A Designer's Guide to Processing, Arduino, and openFrameworks

Programming

Interactivity

O'REILLY®

Joshua Noble

Programming Interactivity

If you want to create rich interactive experiences with your artwork, designs, or prototypes using electronics and programming, this is the place to start. *Programming Interactivity* helps you explore important themes in interactive art and design: 2D and 3D graphics, sound, physical interaction, computer vision, geolocation, and much more. This book also examines cutting-edge techniques for interaction design, and includes discussions with leading artists and designers on projects and theory.

Previous programming experience, while helpful, is not required. The book covers some of the basics of programming and electrical engineering, and provides a complete introduction to three freely available tools created specifically for artists and designers.

- **Processing**, a Java-based programming language and environment for building projects on the desktop, Web, or mobile phones
- **Arduino**, a system that integrates a microcomputer prototyping board, IDE, and programming language for creating your own hardware and controls
- **openFrameworks**, a coding framework for designers and artists that uses the powerful C++ programming language

Inside, you'll find working code samples you can use right away, along with the background and technical information you need to design, program, build, and troubleshoot your own projects. You'll finish this book knowing how to design interaction and incorporate software and electronics into your projects.

"This book is a great way for anyone to get started making interactive applications. It teaches the fundamentals of programming and electronics so that beginners can start making cool projects right away."

—**Mark Frauenfelder,** *editor-in-chief,* MAKE

Joshua Noble, a consultant, freelance developer, and Rich Internet Application designer, has taught coding and electronics to art and design students at the School of the Museum of Fine Arts in Boston. He works extensively with the tools discussed in this book, and shares his knowledge in workshops at colleges and elsewhere.

INTRODUCTORY INTERMEDIATE ADVANCED

Previous programming experience, while recommended, is not required.

O'REILLY®
oreilly.com

US $49.99 **CAN $62.99**
ISBN: 978-0-596-15414-1

Safari Books Online
Free online edition
for 45 days with purchase of this book. Details on last page.

Programming Interactivity

A Designer's Guide to Processing, Arduino, and openFrameworks

Joshua Noble

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Programming Interactivity
by Joshua Noble

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steve Weiss
Production Editor: Sumita Mukherji
Copyeditor: Kim Wimpsett
Proofreader: Sumita Mukherji
Production Services: Newgen

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
July 2009: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-15414-1

[M]

1247251884

Table of Contents

Part III. Explorations

Preface

This is a book about creating physical interaction with computer systems. It focuses on designing hardware and programming for systems that use either physical input or physical feedback. This book has been a dream of mine since I was an art student beginning to create interactive installations and finding that there was no simple introduction to the topics that I wanted to explore. At the time, I didn't know what platforms, tools, and programming languages were available for creating interactive art, and I didn't know where to find more information about these topics that a relative novice programmer could understand. As I began teaching, I was asked the same question again and again by students: "where do I begin?" Much has changed in the seven years since then, though, and now many excellent projects are helping beginners program, artists create, and programmers rapidly prototype applications. We'll cover three of these projects in this book: Processing, Arduino, and openFrameworks. This book intends to answer the question "Where do I begin?" in as comprehensive a manner as possible. It is the intention of this book to be useful for almost any type of project. This book will provide technical advice, critical commentary for you to consider, code that you can use, hardware diagrams that you can use, and further resources for you to explore.

Ten years ago, the idea of artists or designers writing code or designing hardware was almost unheard of. Today, not only has it become commonplace, but it has become an important arena of expression and exploration. The dialogue between technology and design is a vital and vibrant one that shapes art and technology alike. I hope that this book can be, in some small way, another path into this conversation for more artists and designers.

Who This Book Is For

This book is aimed at designers, artists, amateur programmers, or anyone interested in working with physical interaction in computing. No assumption is made about your technical background or previous experience. The only assumption is that you are interested in learning to program and build hardware. This book is an introduction to a great number of topics, and throughout the book we list links to further resources so you can expand your knowledge or explore a particular topic that interests you.

We encourage you to make as much use as possible of these resources and to use this book as a map for exploring a great number of technologies and techniques.

How This Book Is Organized

This book is broken into three parts. The first introduces the three projects that will be used throughout this book, the second introduces some of the most common themes in creating interaction in designs and applications, and the third introduces some of the more advanced topics that you may want to explore further. Also included with some of the chapters are interviews with programmers, artists, designers, and authors who work with the tools covered in this book. Covering such a massive range of topics means that this book doesn't go into great depth about most of them, but it is filled with references to other books, websites, designers, and artists that you may find helpful or inspiring.

What Is—and Isn't—in This Book

My excitement about the ideas and rapid growth of the field of interaction design is hard to contain. However, as exciting and far-reaching as interaction design is, the limitations of time and physical book size dictate that I be selective about what is and isn't covered in this book.

What's in

This book covers Processing, Arduino, and openFrameworks. To help novice programmers, it covers some of the core elements of programming in C and C++ for Arduino and openFrameworks and also covers the Processing language. We introduce dozens of libraries for openFrameworks and Processing—too many to list here. Some of these are official libraries or add-ons for the two frameworks, and some are simply extensions that have been created for this book or provided by altruistic coders.

We also introduce some of the basics of electronics and how computer hardware functions, as well as many tools and components that you can use with an Arduino. The Arduino and Processing IDEs are covered, as are two different IDEs for openFrameworks, namely, CodeBlocks, and Xcode. The Arduino Duemilanove and Mini are covered in depth, and we discuss other boards only briefly. We cover many electronic components that have designed expressly for the Arduino, called *shields*, in depth as well.

What's Not in

While this book shows how to create some circuits, it doesn't cover a great deal of the fundamentals of electronics or hardware, how to create circuits, or electronics theory.

Chapter 18 lists some excellent tutorials and references. While the book does cover the Processing subset of the Java programming language, to conserve space and maintain focus, it doesn't cover Java. The book doesn't cover many aspects of C++, such as templates, inline functions, operator overloading, and abstract classes. Again, though, listed in Chapter 18 are several excellent resources that you can use to learn about these deeper topics in C++.

There are so many Arduino-compatible boards now that it's almost impossible to cover them all in depth; the book mentions the Mega, the Nano, and several other boards only in passing and leaves out many of the Arduino-compatible boards that are not created by the Arduino team. Quite a few components and other tools that we would have liked to discuss in depth could not be included to maintain scope and to save space. A good camera for computer vision was not included either, though a glance at the openFrameworks or Processing forums will likely provide a more up-to-date discussion than could have been given here.

Many topics that we would have liked to include have been left out because of space considerations: artificial intelligence, data visualization, and algorithmic music, among others. Though these are all potentially interesting areas for artists and designers, the focus of the book is on teaching some of the theory and techniques for interaction design as well as the basics of hardware and programming. The resources listed at the end of the book can provide the names of some materials that might help you explore these topics.

Companion Website

All the code included in this book are available for download from the book's companion website, *http://www.oreilly.com/catalog/9780596154141*.

Typographical Conventions Used in This Book

The following typographical conventions are used in this book:

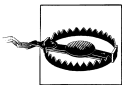
Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, path names, and directories.

`Constant width`
: Indicates direct references to code, text output from executing scripts, XML tags, HTML tags, and the contents of files.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Hexadecimal numbers in this book are denoted with the prefix `0x`.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Answering a question by citing this book and quoting example code does not require permission. On the other hand, selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Programming Interactivity* by Joshua Noble. Copyright 2009 Joshua Noble, 978-0-596-15414-1.

If you think your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

We'd Like to Hear from You

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596154141

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

I need, first and foremost, to thank the wonderful engineers, artists, programmers, and dreamers who created the platforms that I've covered in this book. It is to all of them that I would like to dedicate this book. A woefully short list has to include Massimo Banzi, Tom Igoe, David Cuartielles, Gianluca Martino, David A. Mellis, Ben Fry, Casey Reas, Zach Lieberman, Theo Watson, Arturo Castro, and Chris O'Shea, the creators of the frameworks covered in this book. There are dozens, if not hundreds, of other names that should be on this list, but space is too limited to list them all. All I can say is thank to you to all the creators of these frameworks and to everyone who uses them to inspire, invent, amaze, and enrich the dialogue about design, technology, and art. This book is a humble attempt to thank you all for everything that you've given to me and to every other programmer, artist, or designer interested for working with computing in novel and interesting ways and bringing more people into the conversation. I would also like to extend my deepest thanks to all my interviewees for taking the time to respond to my questions and enrich this book and for so enriching the world of interaction design and art. To everyone who provided code for this book as well, created open source code, or answered questions on any of the forums for beginners, thank you for your efforts to create a community.

This book is as much my effort as it is the sum of the efforts of the editorial team that worked on it. My technical editors, Michael Margolis, Adam Parrish, and Jeremy Rotzstain, have been absolutely fantastic. Their expertise, suggestions, and fresh look at what I was working on shaped not only this book but enlightened me, showed me new ways of solving problems, introduced me to new tools and techniques, and sharpened my thinking and broadened my horizons for the better. This book is a collaboration between all four of us in every sense of the word. I cannot pay them enough thanks for their excellent work. I would also like to thank Justin Hunyh and Mike Gionfriddo from LiquidWare as well as Nathan Seidle from Sparkfun for all of their help. My editors, Robyn Thomas and Kim Wimpsett, have been incredible, helping me with my sometime torturous grammar and patiently working with my propensity for sending in extremely rough drafts to bounce ideas off of them. They have made this book better than it ever could have been without their watchful eyes and guidance. Finally, I need to thank Steve Weiss for listening to my idea when I first proposed it and helping guide it through to completion.

I need to thank all of my friends in New York, Amsterdam, Geneva, London, Zurich, Boston, Paris, and Toulouse for their support, their ideas, their Internet, and their encouragement. I would like to thank my family as well, and particularly my mother, for their support and humor.

PART I

Introductions

Part I of this book is an introduction not only to writing code and working with hardware, but also to the three tools that you'll be learning about in this book: Processing, Arduino, and openFrameworks. You'll learn about writing software in general and about writing code for each of the three platforms in particular. Each of the three platforms approaches the applications that you'll use to actually write code in a slightly different way, but the general concepts of working with the code are the same across all three. One thing you'll come to realize as you learn more about programming is that the core concepts of programming are quite similar across a lot of different programming languages. This means that the knowledge you have about Processing, for instance, can help you if you decide to create a project using Arduino. Understanding how these different frameworks are similar will help you leverage your knowledge in one to work with the other. Understanding how they're different will help you choose the right one for the kind of project that you want to create.

Chapter 1, *Introducing Interaction Design*, will give you an introduction into what interactive design means and the tools available to build interactivity. In Chapter 2, *Programming Basics*, you'll be introduced to the fundamental concepts of programming. There you'll learn how programming an application works, learn what code really is, see some of the key terminology and concepts, and get ready to dive into actually writing code in each of the three programming tools that you'll be exploring. Chapter 3, *Processing*, introduces you to the Processing language and environment and shows you some code to get you started working with video, sound, images, and simple graphics. Chapter 4, *Arduino*, introduces the Arduino language, hardware platform, and environment. Since the Arduino platform consists of a programming language, a hardware device, and an IDE that you use to write code, you'll be learning about how to use all three. In Chapter 5, *Programming Revisited*, you'll learn about some more advanced topics in programming that will prepare you to work with openFrameworks, like classes and object-oriented programming. Finally, in Chapter 6, *openFrameworks*, you'll be introduced to the C++ programming language and the openFrameworks way of using it.

This first part might involve taking a lot of first steps for you, but once you have stepped your way through it, you'll understand a great deal about three tools that can help you build almost any kind of interactive project by creating your own hardware and software.

CHAPTER 1

Introducing Interaction Design

The scientist and philosopher Alfred Korzybski once remarked, "The map is not the territory," and it's in that spirit that this book was written. The map may not be the territory, but it is helpful for getting around the territory and for finding where you are and where you want to go. This book covers a vast range of topics from programming to electronics to interaction design to art, but it doesn't cover any one of them in great depth. It covers all of these topics because they are part of an emerging territory that is often called *interaction design*, and that territory encompasses art, design, psychology, engineering, and programming. It's also a territory that is becoming more and more accessible thanks to excellent projects like the ones that we'll be exploring in the book—tools that have been created to make code and coding easier to do.

You should use this book like a map to see what technologies exist and the areas in interaction design that you might want to explore. This isn't a cookbook or an in-depth technical manual, but it will point you in the direction of other books, researchers, designers, projects, and artists as you go along. This book will also give you the technical understanding to know how to find information on almost any kind of project that you want to explore and what to do with that information once you find it.

What This Book Is for

This book was created under the premise that technology and code are not tools solely for computer scientists or engineers to create applications and that no one be intimidated by or shy away from working with and exploring electronics, hardware, and code. Artists and designers can be interested in enabling interaction between users and between applications in ways that can be accentuated by the addition of custom computer applications or that can be realized only through the use of custom computer applications. You can focus on creating applications that emphasize their technological nature or on creating applications that feel very high-tech or use familiar metaphors like a keyboard and mouse or touchscreen. You can also choose to accentuate other aspects of the interaction or hide the technology behind a more organic interface. This book is specifically about the interactions that users or viewers can have with computers,

electronics, tools, and the platforms that artists and designers can use to create applications and electronics that users can interact with. You'll be learning about three tools: Processing, openFrameworks, and Arduino.

These frameworks are designed specifically for artists and designers and as such are perfect for discussing how we can begin to create interactive designs and artworks. Each of them has a different background and uses different kinds of technology, but all of them are created with the goal of helping you explore and create applications more painlessly and quickly. In addition to showing you specifics of those three tools, this book focuses on three slightly more abstract concepts: code, interaction design, and ideas. Creating code is a similar activity whether you're writing something in C++ for openFrameworks or you're creating some logic in a circuit with Arduino. In both cases, you're creating a process that will run many times, perhaps even thousands of times, and that will generate the outcome you want. That outcome could be lighting up an LED when someone presses a button, or it could be creating graphics when a certain color is detected in a video feed.

This book also makes a few assumptions about you, the reader. I assume that you don't have a deep, or even any, programming or technical background. I also assume that you're a designer, artist, or other creative thinker interested in learning about code to create interactive applications in some way or shape. You might be a designer wanting to begin playing with interactive elements in your designs, wanting to create physically reactive applications to explore some interaction design concept, or wanting to prototype an idea for a product. You might be an artist wanting to begin working with interactive installations or with interactive computer graphics. You might be an architect wanting to get a basic understanding of programming and hardware to explore reactive architecture. You might be none of these at all, which is fine, too, as long as you're interested in exploring these themes while you learn about the three frameworks this book describes.

You'll explore the nature of interaction through common tools and techniques as well as through some discussions with designers, engineers, and artists working with interaction. In all likelihood, this book will not radically alter your perception of what interaction is, nor will it introduce you to radically new modes of interaction. This book will introduce to you to methods of creating common interactive elements that you can then use to explore further techniques of facilitating interactions between users or creating interactive elements that a user or viewer can experience.

Programming for Interactivity

This book is called *Programming Interactivity* because it's focused primarily on programming for interaction design, that is, programming to create an application with which users interact directly. There are many styles of programming, and some techniques and ways of thinking about code are better suited to programming servers or databases than interaction. In this book, we're going to concentrate explicitly on things

you can use to tell users something or to have users tell your application something. One of the great challenges in interaction design is actually creating real interactions between what you're designing and the user who will be using it.

The Nature of Interaction

So then, what exactly is *interaction*? Interaction could be defined as the exchange of information between two or more active participants. The writer and video game designer Chris Crawford describes interaction as "an iterative process of listening, thinking, and speaking between two or more actors." Generally, when we're talking about interaction and programming it's because one element in the interaction is a computer system of some sort or some control element that a person is trying to get to do something. The person for whom the computer or mechanical system is being designed is called the *user*, and what the user is using is called the *system*. There are many different terms floating around today, such as *human computer interaction*, *computer human interaction*, or *experience design*. All mean more or less the same thing: designing a system of some sort that a person can interact with in a way that is meaningful to them. As an interaction designer, you're trying to understand what the user wants to do and how the system that you're creating should respond. That system can be almost anything: a game, a menu, a series of connected sensors and lights, a complicated physically interactive application, or even a group of other people.

There is another key concept in interaction design that you should understand: the *feedback loop*. The feedback loop is a process of an entity communicating with itself while checking with either an internal or external regulatory system. That sounds a little more complex than it actually is. You're actually already quite familiar with biological regulatory systems; sweating keeps your body cool, breathing keeps oxygen flowing through your body, and blinking keeps your eyes from drying out. When you need more oxygen, your body breathes harder. This isn't something you have to tell your body to do; it simply does it. To maintain a constant level of oxygen, it sends out signals to breathe more and more deeply or frequently until it reaches the correct level. It feeds back on itself, sending signals to itself to breathe more again and again until it doesn't need to send those signals anymore. You can also think of the feedback that you give yourself while staying upright on a bicycle. You're constantly adjusting your balance minutely, with your brain feeding data to your body and your body feeding data back in a constant loop that helps you stay balanced. These loops are important in the notion of a system that does something constantly. Without feedback, systems can't regulate themselves because they won't know what they're doing.

Let's start at *messaging* and work our way up to *interaction*. While one participant certainly may be more active than the other, the "interaction" doesn't really apply when we use it to describe a *transmission*, that is, a message sent to someone with no way of handling a response. Think of a television commercial or a radio broadcast: it's simply a signal that you can listen to if you're in the right place at the right time and you have

the right equipment. These broadcasts flow on regardless of whether you or anyone else is listening, and they occur on their own time, in their own tempo.

When you give a user a way of *rewinding* or controlling the tempo of information, an extra layer of user control is added. You can't really *interact* with a book or a static web page, or even the vast majority of dynamic web pages, but you can control the speed at which you read them, and you can rewind information that you're not sure about. These are really guided transmissions in that they give you a chunk of information that is more or less established and ask you which part of it you want to view. Scrolling, linking, fast-forwarding, and rewinding are all the techniques of guided transmissions.

When you give a user a way to accomplish a task or input data into the system that changes it in a substantial way and you create a means for that system to respond to what the user is doing, then you're creating interaction. Reactive interaction is really the beginning of interaction because it gets you started thinking about what the user will do and how your system or object will react. For everything that user does, the system or object needs to have a response, even if that response is "I didn't understand" or another kind of error message. This can also be built into a single system. Many kinds of applications monitor their own performance, checking the state of a property in the system or the number of boxes available in a warehouse, for instance. If you imagine this as being an interaction between two people, then you might imagine a parent giving a child an order.

A somewhat more complex model of interaction is one where the system is constantly doing a task and the users' input regulates that task. Many industrial monitoring systems function this way, as do the underlying parts of game engines, and many interactive installations. The difficulty of creating this kind of interaction is ensuring that users always know what the system is doing at any given time, understand how they can modify it, and understand exactly how their modifications to one aspect of the system might affect another. If you imagine this between two people, then you might imagine a parent helping a child walk, ensuring that she doesn't don't fall over as she goes. You can also imagine how a regulatory system might function, where the system regulates the user as they're executing a task. This isn't really two entities fully communicating because the regulated system doesn't respond—it simply changes its behavior—but it does involve continuous systems. Systems can perform this task on their own as well, monitoring a process and providing regulation of an ongoing process.

This last mode of interaction blends into another. It is a very similar but slightly more complex model of creating interaction that might be described as the *didactic*, or learning, mode of interaction. Here, the system is still running continuously, and the user can see into the system, but instead of regulating the behavior, the user is learning from the output data. A lot of monitoring applications function this way, providing a view into relevant data and data points that the user can use to learn about a process. Again, the system isn't actively conversing with a user; it's just running and reporting information to the user. The user also has his process driven by the reporting from the system

but not really modified by it, which is why it's a learning model. Both systems and people are more than capable of learning from themselves, albeit in quite different ways.

A more complex mode of interaction is a management type model where the user communicates something to a system and the system communicates something back that allows the user to carry on with a secondary task. This is where you begin to see the real complexities of communication between users and systems. The user is communicating with a system and asks the system to perform some task. The system responds in a way that allows a user to continue with a secondary task. The system continues to run, and the user continues to run even while she has her own internal feedback loop occurring. One can find this in many real-time monitoring applications in fields from finance to medicine.

Finally, we have the most complex mode of interaction: a full-fledged conversation. This is something that humans have mastered doing amongst one another, but it's another matter altogether to create this between a human and a machine because of how complex the notion of a conversation really is. When you think about how much data is communicated in a conversation through words, tone of voice, facial expressions, body posture, subtext, and context, you realize it's a substantial amount of information being exchanged and processed at extremely high rates. Most user-system conversations are a great deal less complex.

A simple but good example of this is navigating using a mobile device: the device is constantly updating its position and displaying that back to the user and providing directions, while the user is actively traveling and querying the device for information. Enabling this conversational mode of interaction between users and systems is one of the most pressing challenges in interaction design and engineering. These modes of interaction all present different challenges and help users do different kinds of things. You'll find that the appropriate mode depends on the users, the task, and the context in which the interaction is taking place.

Messages and Interaction

Interaction happens via messages sent from systems to users, and vice versa. These messages can be text, speech, colors, visual feedback, or mechanical and physical input or feedback. Depending on the kind of application, winking can be just as clear and important a message as pushing a button. One thing that interaction designers talk about a great deal is how to construct and receive messages in a way that is simple and unambiguous for users and for the system.

One of the most difficult tasks in creating interactive applications is to understand how the system sees messages from users and how the user sees messages from the system. With applications that have a great degree of interactivity, allow more tasks for the user and the system, and allow for more sophisticated messages, it is easy for a conversation to become unclear to one party. When a message isn't understood, it's quite important

to help the other party understand not just what wasn't understood but also how it can be fixed. If I don't understand something that someone says to me, I ask that person to repeat it. If I ask for a web page that doesn't exist, the server responds with an error page that tells me the page doesn't exist. The more freedom each party has, the greater the possibility of erroneous, unintended messages, and the greater the need for educating one party about what the other party understands and how that understanding is being constructed.

Think for a moment about a conversation between two adults. Communicating like this requires years of what could be described as *user training*: learning a language, learning appropriate and inappropriate behavior, learning a value system, and so on. It is because of this that the interaction between two humans can be as rich as it is. This idea of training the user to understand what messages the system understands and what a message from the system means is a tricky process. Creating a program with a datagrid where a user can select items is quite simple for the user to begin to understand because most computer-literate users are familiar with the notion of a datagrid. We see datagrids quite frequently, and we generally have an understanding of what they can do, what they can't do, a rough understanding of what error messages coming from datagrids might mean, and how to use them. If you're using a new kind of control or interface, you'll have to make sure that you provide ways for users to learn what your system is, how it works, and what they can do with it.

There is a correlation between the richness of interactive system and the difficulty of creating it: the richer the interaction, the more that can go wrong. This is part of why designers spend so much time and energy attempting to create *anticipatable experiences*: interactive experiences where a user or viewer can leverage other realms of knowledge or other experiences interacting. Popular slogans in design like "principle of least surprise" or express the notion that the familiar interaction is the preferable interaction because the learning curve for the user is much more shallow than a truly novel interaction. Users must learn how feedback is returned to them and how to modify their behavior based on the feedback, both of which can be a lengthy process.

Interfaces and Interaction

One part of the feedback from a system is actual messages sent back and forth—text prompts, for example—but the interface is another important part of the communication of an interaction. An interface sits between two actors and facilitates their communication. This can be a screen, a control panel, an interactive wall, or simply a microphone and a pair of speakers. The interface is whatever shared materials the user and the system use to send and receive messages. Interface design is a very large topic unto itself, but it gets a little more manageable if you consider it in terms of what it means for designing an interaction.

The interface is the medium of the communication between the user and the system. It drives a lot of what is possible and what is not possible, what is efficient and what isn't, and what the tone of the interaction is. If you think about how you talk to someone on the phone versus how you talk to them in person, you're probably using more hand gestures, facial expressions, and other forms of nonverbal communication in person and being more direct and using your tone of voice more when you are on the phone. What we use to do something affects a lot of how we do that thing. Having a functional, expressive, and attractive interface is very important in creating the means for an interaction to occur. The attractiveness of an interface is an important part of making an interaction pleasant to a use; the colors, text, symmetry, sounds, and graphics are important and are communicative elements that shape a great deal about what a user thinks about your system. This shouldn't come as a great surprise to anyone, but users prefer good-looking interfaces. What makes those interfaces attractive is largely a matter of context, both for your users and for the task that they're trying to accomplish with your system. While users *prefer* attractive interfaces, they *need* functional interfaces. The functionality of an interface is part of what makes a system good for a task and what makes a user able to use your system. Even if what that system does is rather opaque, the user still needs a functional interface that shows him what his input does and gives him feedback.

It's important to remember that interaction is more than the use of an interface. When we consider the most common interactions between a user and a machine—for example, a cell phone call—they're quite simple in terms of the interaction between the user and the object. For a cell phone, you simply dial numbers to find someone else in a system; it alerts you if you're being sought, and it sends and receives sound. This relatively simple interaction is important for reasons other than the interaction between the person and the object; it's important because of the context of that interaction: you can make a cell phone call from almost anywhere. Before cell phones, you needed a phone line available to you, but now, with a cell phone, you simply need a phone and an account. You can reach people while both of you are away from home, and you can be reached when you are away from your home or office. When the cell phone first emerged, cell phone users already understood how to make and receive telephone calls, and the general pattern of the user interface was already established. True innovations in user interfaces are very difficult to realize because they often require very substantial engineering efforts and serious thinking by the interaction designer to ensure that the interface will function properly. Also, they require a lot of user training and retraining. There aren't a great deal of true revolutions in user interfaces: the creation of the keyboard, Doug Englebar's mouse (the prototype of the mouse we know today), Ivan Sutherland's sketchpad, the desktop GUI, and now the capacitive touchscreen. These were technological changes and impressive feats of engineering, and they were also shifts in the way the people used computers. Revolutionary interfaces shape more than just the way that a tool appears; they redefine the possibilities of how a tool can be used.

Languages of Interaction

All interactions have a certain vocabulary that they use. If you think of how you delete something from the desktop with a mouse, you might say, "I select the file and drag it to the trash." The actual actions that you're performing when you do this are a little different from what the system understands you to be doing, but that's not really what's important. What's important is that you understand what the actions you can perform are and you know that the system understands those actions in the same way and will perform them in the same way that you expect. Having a meaningful, efficient, and productive interaction, just like creating a language or a code, requires that both parties agree on the meaning of the symbol and the meaning of the order in which actions occur. Those particular understandings are going to be quite different depending on the interface and type of interaction that the user undertakes.

In this book, we'll examine some of the many different kinds of interactions, but don't take this next section as a list of categories. Considering the pervasiveness of computing and interactions that exist with computing, there are so very many kinds interaction between humans and computers that it is difficult to even reference some of the most common modes of interaction without some overlap between categories.

Physical manipulation

These are the first interfaces that were created for electronics and some of the first designed multifunction man/machine interactions. Typically, before the advent of the car and radio, which were the first two common machines with multiple interface elements, a machine had a single switch or use. The user's attention was focused on a single task at a time. Radios and automobiles presented novel challenges because both required multiple actions by nonspecialists; in the case of the automobile, this included speed and direction at all times and other tasks at irregular times. The interface might be a control that represents either a state that can be activated by flipping a switch or pushing a button or a range that can be set by turning a knob or pushing a slider. The interface lets users not only control the values that they are setting but also check values via labeling of sliders, knobs, dials, and switches. Dials, oscilloscopes, and other feedback interface elements let users verify information more quickly without referring to the actual interface element that they were manipulating. This requires that the user monitor multiple sources of information at a given time while manipulating controls. Physical manipulation of a control is one of the most important and interesting ways of creating interaction with system.

Input using code

At the dawn of the age of computing, the classic user interaction model was a terminal where a user input code commands that were then run and the results were reported to the screen in the form of text. The driving interactive concept was to command the machine via a system of commands that the computer had been preprogrammed to recognize. The user had to be knowledgeable or at the very least

comfortable with requesting help from a very bare interface. This is certainly not the end of keyboard-based interactive behaviors, though. Consider the notion of the *hot key*, for instance Ctrl+Z for undo, beloved by so many programmers and ubiquitous in all applications from word and image processing applications to browsers. The hot key is no different from the command line but accentuates the user interface by allowing the user to automate repetitive tasks or perform a task quickly without diverting their attention from another task.

Mouse manipulation

This is the most common method of interacting with a computer at this moment and the interface for which almost all commonly used applications have been designed. Consider the language of working with the mouse, the techniques that have been implemented by designers and learned by users: drag-and-drop, double-click, and click-and-hold. These movements and the meanings behind them in different applications are not completely standard, nor are they entirely fixed. One application may use a given gesture in many different ways in a single application and rely on the user understanding the feedback given to them by the application to know which meaning of the gesture will be used in the current context.

Presence, location, and image

The use of the presence and absence of the participant or user is an extremely simple but profoundly intuitive way of interacting. This can be detected by weight, motion, light, heat, or, in certain cases, sound. The reaction to simple presence or absence acts as a switch, begins a process, or ends a process. The presence of the body, though simple, is a powerful basis of interaction; it engages users and asks users to engage with their presence, their position, and their image. This can be as simple as an automatic door sliding open as we approach, or as complex as Theo Watson's *Audio Space*, where visitors don a headset equipped with earphones and a microphone and record messages that are then placed in the spot where they were recorded. As another user enters the location where a message was left, the message is played back along with any recorded by previous visitors. Each message sounds as if it is coming from the spot where it was recorded. We can imagine the body as a switch, or we can imagine the body as the image of the body and analyze this using photos or videos in any great number of ways. This theme of embodiment drives a great deal of fascinating interactions using what is called *computer vision*, that is, the analysis of images input using a camera, turned into pixels, and then analyzed. Later in this book, we'll examine using computer vision to detect movement in an image and even to detect the location of a human face within an image.

Haptic interfaces and multitouch

At the time of the writing of this book, Apple iPhone, Microsoft Surface, and a great number of new tools for multiple touch-based interfaces have already been introduced. Given the excitement around these technologies, the speed of change and innovation will likely outstrip any attempts by myself or my editors to keep this text abreast of the most cutting-edge products or technologies. Nevertheless, the fundamentals of designing and structuring interactions using these

gesture-based interfaces will likely not change. These essentials are based on what will be familiar gestures to anyone who has used any of these products: using two fingers to expand or contract, turning two fingers to rotate, tapping to select. These are not used simply for software applications, either. Consider how often the waving gesture is used in an airport bathroom with sinks to turn on the water, paper towel dispensers, and hand driers. The language of these gestures becomes a language that we can use to enable interaction much as a common natural language, an icon, or a pattern of buttons pressed on a video game controller.

Gesture

The gesture is a fascinating interactive model because it so readily associates itself with signs, writing, and physicality. This notion of the interaction that is not driven by a keyboard or a mouse is particularly powerful because mouse and key interaction is often nonintuitive for certain kinds of tasks. Gestures are often implemented with touchscreen interfaces or mouse movements or pens and are very often used for drawing applications, simple navigation elements, adaptive technologies, or applications for children. There are many different cutting-edge interactive approaches that are being explored, from writing recognition systems and novel key input strategies like Swype to hand motion recognition systems via video.

Voice and speech recognition

Voice recognition is the programming of a computer to recognize certain words or phrases and perform certain tasks based on those commands. Commands can be as simple as voice activation, that is, having the voice act as a switch to turn something on, and as complex as recognizing different words as commands. For a computer, words or commands are recognized as patterns of sounds that are then strung together and compared with a dictionary of patterns to determine what the command could be. Speech recognition is a much more advanced topic, using roughly the same approach as a simple command recognition engine, but with a far larger dictionary and more powerful tools to determine the input. Beyond speech, the voice itself can be used to provide input, volume, tone, and duration, and can be used to drive the interaction between users and applications.

This is just a short list of some of the most prevalent themes in interaction design. In this book, there won't be space to cover all of these approaches to interactivity, but you will learn some of the basics behind each of them and get information about further resources that you can use for your own design work.

Design and Interaction

The great industrial designer Henry Dreyfuss called design "the measure of man." By this, he meant that the design of things is an excellent way to understand and analyze the activities of human beings. Defining the word *design* is a task better left to others, so I'll leave my contribution at this: interaction design is the creation of tools for how we do specific things. The more specific the thing, the more finely the tool can be honed

for it, and the more specific the interaction design can be. Interaction is sometimes confused with "doing something with a tool," and although that's important, it's a little less specific than "how we do things with a tool." Thinking about tools in terms of *how*, rather than just *what*, *when*, or *why*, isolates those things about the interaction that define the experience of doing that task. A lot depends on the task as well. A singular task with a singular action does not foster much dissonance; therefore, it can bear a lot more dissonance before it becomes meaningless. A task of multiple actions creates much greater dissonance and can lose meaningfulness much more quickly.

The design of an interaction is a complex process that involves a lot of modeling of how a system will work, how a user will approach the goal she's trying to accomplish, and how the interface needs to be configured to allow for all of these different operations. All of these taken together create the *context of the interaction* that you're making. The context is very important to what choices you should make for the design of an interaction. You might want to make the interaction very cut and dry so that everything that the user expects is given to her as quickly as possible and in an unambiguous manner. Most business applications or very task-based applications function this way; users know what they can do in unambiguous terms, and the interaction doesn't deviate much from that initial information. There is a real pleasure in knowing what to expect and getting it so that you can make the interaction—and by extension the application or object—attractive. Or, you might want to make something much more playful, where the reward is in discovering the interaction and seeing it change throughout the use of it. Either way, a good understanding of the context of the user will help you create a better system and a better experience.

One of the tricks of interaction design is that fundamentally what users are trying to do when they're interacting with a system is to correlate it to something else that they're more familiar with. Anyone who has ever said or heard anyone else say "the computer is thinking" has seen a little bit of anthropomorphic thought applied to a computer. As human beings, we are very good at a few different things, and when it comes to interaction design, one of the more important is using our understanding of the inner processes of other people. Interaction with a system doesn't really involve understanding what someone else is thinking, but it does use some of the same cognitive processes. To that end, as an interaction designer, you want to give good cues that will help users understand what's going on. They may not need to know exactly what the process of your system is, and probably shouldn't, but they do need to know more or less what your system is doing with the information that they give it.

Art and Interaction

Interactivity in art has been a hotly discussed and debated topic for at least 20 years now, and the kinds of interactivity that you see in art pieces are constantly changing to expand the definitions of *art* and *interaction*. There are many computer games that can be considered art, many art pieces that can be considered industrial design, and a vast

and ever-increasing number of projects that can fit comfortably into art galleries and design shows.

For the purposes of this book, there isn't much point in differentiating between the fields of interactive art, industrial design, interaction design, and traditional software engineering. Although these different fields might seem quite different from one another, they actually all share common goals. They all attempt to create objects and experiences for users, they use similar tools and processes, and they all share a common workflow that goes from sketch to prototype to final product to showing. You can think of a continuum, where at one end there are predictable and well-defined things that may be more suited for completing a task, and at the other end are more unpredictable and dissonant works that challenge and provoke us but may not be useful in our everyday lives. There is a curious dance between art and design in interactive art that plays on the relationship between simplicity and complexity, usefulness and uselessness, and goals and open interpretations. Deciding which end of that spectrum is more interesting to you has a lot of bearing on how you think about the interaction but it doesn't change the code that you write or the way that you design your hardware.

Making interactive art is quite different from making noninteractive art because the real object of interactive art is the situation. In painting, the object is the painting itself; in sculpture, it is the object and the space around it; in a video piece, the object is the video projection. In an interactive artwork, the object of the art is really the interaction between the viewer and the system that the artist has created. That system can be very technologically complex, it can have a single simple technical element, or it can have none at all. This book discusses artists and artworks that make heavy use of technology, what are often called *new media artists*, because they are artists who use or develop the tools that you'll be learning how to use. I distinguish new media art from interactive art because projects that use programming (but that aren't interactive) don't have that key characteristic of being created in the situation where the viewer encounters them. There are many technologically sophisticated projects that use the tools covered in this book but that are not actually interactive. For the most part, this book covers artists who work with interactivity and projects that generate feedback in response to the actions of a user.

One of the interesting challenges of working with interactive art is that the art can be truly useful and functional in many ways while still being art. You also have a great deal of control over the context of what that art is; an artwork can be viewed or experienced in any location that the user chooses, altered to become something unrecognizable, or used in a way that it was not intended to be used when it was first created. Many designers are exploring what they call *critical design*, designed objects that not only function but exist to be thought-provoking as well, making users think in critical ways that are usually associated with art rather than with design. This overlap between the design of an object and the creation of an art experience is part of what makes interactivity such a rich topic for artists to explore because you can open the realm of what a user can experience and explore in deeply expressive ways.

Data Exchange and Exploration

The task or goal that an interaction facilitates is as important as the way in which an interaction is carried out between a user and a system. Again, the types listed here aren't being included to make a list of types of interactive work, but to show some of the common themes that run through interactive art and design and to help you get an idea of what you'll be exploring in this book:

Supporting data visualization

Data visualization is an increasingly relevant theme given the amount of data that we as members of an increasingly information-centric society must process. A well-formed data visualization is a powerful tool because it lets a user not only comprehend individual data points but also understand the relationship between what are called *data points*, detect patterns in the data, and even reconfigure and recontextualize information. Data visualization accelerates the ability of the user to process and synthesize new information by not simply learning a fact but by locating the fact within a discourse quickly.

As the designer and writer Frank van Ham notes, "They should be massively collaborative...not focus on analysis but on communication...it should be visual and end user driven." The goal of data visualization is to generate, for the user, a view into data. This can be a view that will help the user understand the data better, as in the work of Ben Fry, where he creates beautiful diagrams that let a viewer more quickly and more fully understand the relationships between the objects in the data. His approach is informed by aesthetic considerations and by careful cognitive and psychological research. *I Want You to Want Me* by Jonathan Harris and Sep Kamvar retrieves data from online data sites and uses that information to generate interactive visualizations. While still data visualization, this piece is a far more dynamic and whimsical approach than the usual graph and chart approach used in standard visualizations, and yet it performs deep and meaningful data parsing and analysis.

The interaction of the user can be a process of refining, exploring juxtaposition, mining new data, or storytelling. When designing data and the interaction with it, we must consider not only what data is presented, but also how users will interpret that data, what they might want to do with it, and how they would want to interact with it. Interaction is far easier when the view into data and the visual representation of that view are clearly related. For instance, the visual representation of the data and the visual representation of filtering should be clear so that the user easily understands what is being filtered and how to change it.

Organizing tasks

Some interactions are interesting because of what they allow us to accomplish. The organization of tasks or actions or of discrete and discontinuous objects is the driving force behind much of the thinking in interface design. When you look back at the history of interfaces for machines and computers, you can see an evolution

of organizing tasks, applications, information, and acts. The now ubiquitous desktop model allowed the user to organize tasks in a way that leveraged both natural cognitive abilities, like the ability to organize things spatially, and a familiar motif for any office worker, namely, the desktop.

One challenge for interaction design is to conceive of ways to effectively put working spaces and organizational tools in places other than the traditional desktop environment. Computing is everywhere, and users want access to programs and data at more places. How to enable interaction on very small screens or with no screen at all is an increasingly relevant challenge given the environments in which users are interacting with environments.

Some of the themes of exploration are how to develop novel desktops environments using new tools like multitouch screens, how to create tools for users to create their own interfaces to fit their needs at a particular time, and how to create interfaces for data visualization and tasks around data visualization.

These types of interactive applications tend much more to practical and functional concerns, enabling users to complete tasks, organize information, and save information. This certainly does not mean that they need to attempt to replicate an operating system in functionality, but rather that they draw on that vocabulary of interaction. This can be used in a somewhat more subversive ways as well, as with Adrian Ward's *Auto-Illustrator*.

Creating experiences

Not all interactive designs need to rely on the traditional application model. In fact, one of the most common and powerful modes of interaction is what might be called the experiential model of interaction. These are often computer games, reactive drawings, or eye-catching graphic displays that engage and entertain without a set purpose. They often use novel connections between audio and visual stimulation and create either spectacles that entertain users or have entertaining interactions. The experiential interaction is very evident in the design of many kinds of computer games, where the user can play a character in the game and see the world through that character's eye. Many times the interaction in these kinds of games is goal oriented, whether that goal be moving to the next level, killing enemies, or scoring points in some way or another. Many interactive installations use a similar model of interaction, where the interaction is playful but often lacks the goal-driven nature of gaming and instead focuses on enabling the viewing of a spectacle or a playing with some engine that creates a sound or graphics. The goal of this kind of interaction is often simply to entertain or engage.

Both games and interactive installations often allow for fast switching between multiple views, perspectives, or models within the flow of the applications. This can be useful not just in gaming but also in an architectural fly-through, to show what a building will be like to walk through, or in data visualization. Gaming style interfaces are also quite common, with first-person views onto a 3D world or with a 2D view onto a world with the user controlling the view onto that world. These

also often involve creating an environment that is reactive to user's actions and independently active when the user is inactive to create the illusion of a world. Interactive installations or more playful and less task-oriented pieces will sometimes also involve inverted mediums, where one draws a sound, a sound creates an image, a physical object becomes a virtual one, or vice versa.

Enabling collaboration between users

The interactiveness of an art piece most certainly does not need to be driven by data or a virtual world; it can be driven by multiple participants in concert with one another. We can conceive of this in a very straightforward way, such as in a whiteboard collaborative application, or in an unpredictable and experimental way, where the input of one user is added to the input of the others in a way that can't easily be anticipated. As with many of these topics, a range exists of predictable and consonant works to unpredictable and dissonant works. Locative gaming where the game play is driven by the users' locations is another kind of interactive application that uses collaboration between users to drive the interaction. Many network-based applications also use the model of collaboration between users to drive the interaction. The system in these kinds of interactions tends to facilitate communication and ensure that messages from one user are received by another instead of generating messages and feedback for a user like a single-player game.

These applications can use chat-based metaphors or presence-based metaphors like some of the large multiplayer games that have become popular lately, or they can create a physical manifestation of each user. As long as the user has some indication of how many other users are interacting and how their actions are affecting the environment or those other users, the interaction can be very compelling.

Controlling mechanics

One of my favorite teachers, writers, and engineers, Tom Igoe, wrote, "Computers should take whatever physical form suits our needs for computing." It is very limiting to think of computing strictly in terms of the computer itself and the traditional interface to the computer, that is, the screen. In fact, interactive designs and artworks can be far more. With the Arduino, we can easily create computers that control machines out in the physical world. These machines can perform tasks as simple as turning lights on and off or as complex as the control of robotics. The machine can be controlled manually by a user or by many users, or it can be reactive, controlled by a program that dictates its responses in reaction to stimulus from users or viewers or from a physical environment.

The control of mechanics can be very task-oriented with rigidly defined effects for each user action, or it can be very playful as in a collaborative application. In the latter case, the controls that the user has can be very loosely defined; that is, the user may have to play with the installation to discover the action that the control performs. In the case of task-oriented controls, the labeling and structure of the controls should be very clearly delineated.

Using tools for performance and as performance

An application can be used as a way of creating an aspect of performance, aiding a performance, or accentuating a performance. You can think of examples as simple as the modification of an electric guitar to projects as complex as completely new interfaces for musical development. An interactive application or tool is a means to a performance or interaction, driven by a performer or driven by the audience. Some of the most interesting uses of this mode of interaction involve sharing a control between the performer and the audience, though this does require some user training to ensure that the audience understands what they are supposed to be doing. These tools don't have to be as conceptually simple as creating a new tool for a performer to use on a stage. They can allow users to create new elements in a performance, control objects from remote locations, or create performers out of the audience.

Creating environments

Many architects are beginning to explore what is called *reactive architecture*, the marriage of architectural practices with computing to create houses and environments that react to users, environmental factors, and external commands. The model of a feedback system is quite important to this model of interaction. The environment needs to monitor itself using sensors or timers to know when to change its state and when to maintain its state. At the simplest, a timer can be used to tell a house to turn the lights on and off, a temperature sensor can be used to maintain a certain temperature, a motion sensor can detect presence, or a humidity sensor can be used to control a dehumidifier. However, by using more complex sensors and systems, you can track movement in a space by using tags, cameras, microphones, or wireless radios and have a system use that data to make changes in the environment to make it more comfortable, to make it louder, or to configure it correctly. Many architects and technologists are designing spaces and buildings that can be configured with a command by a user to change the way a space is used or to make the space itself more interactive. These sorts of spaces are often called *smart rooms* or *enabled architecture*, and they are an important area of research for both architects and engineers. Computing doesn't have to be limited to indoor spaces, though; outdoor spaces like parks, walking trails, squares, or streets can also be sites for interesting technological interventions that can be playful, helpful, or thought-provoking. It is important, though, to always consider the appropriateness of an application for the space in which it exists and how the user engages that interaction. In a public space, this becomes especially important since a user should have the decision of whether to interact with it.

Telling a narrative or story

One of the more interesting themes beginning to emerge in interactive design is the notion of using interaction to tell a story or narrative. These sorts of works typically rely on the interface to allow the user to control the flow or direction of the narrative using techniques cribbed from data visualization or gaming. Despite using con-

cepts familiar from gaming and visualization, narratives offer a different set of challenges more familiar to filmmakers and authors than to engineering concerns.

Working Process

The actual process of creating interactive work generally follows a combination of any of the following processes:

Conception

Conception can consist of sketches in a notebook, a daydream, a product or process of research or something developed in consultation with a client who wants the application for commercial purposes, or any combination of the three. You should try to map out in as much detail as possible what you would like the application to do, how the interaction should feel to the user, and the goals of the application. All projects will require research and planning of some kind. You shouldn't let this suffocate you or stifle your creativity, but you should include it. Starting to sketch without a clear plan of what you're doing can often lead to great and new ideas. For most people, I dare say that starting to write code without a clear plan of what they're doing usually doesn't produce something usable.

Research

When you've decided what you would like your application to look like and how you would like it to function, you'll need to do the research on what components you might need and what libraries or existing code might be available that can help you create your project. If you need hardware, you should determine your budget for your application and determine which components you'll need and how they fit into your budget. It's important to ask questions either on forums or of colleagues to ensure that the components or approaches you're considering will work. Most projects will require different technologies, but almost all the requisite pieces will be available, or you will be able to leverage existing projects that have been created by you or by another artist or developer. Twenty years ago, this may not have been true, but it is now.

Design

The design phase is one of the more amorphous because it blends so easily into the research, conception, and actual building of your project. Sometimes you may not have a design phase, and sometimes all you will have is a design phase, depending on the requirements and nature of what you are building. At the very least, you should define all the parts of your application clearly, have a clear vision of how they will appear to a user and how they will respond to a user, and understand exactly how a user's action should correlate to an action by the system. It may help to create diagrams that show the flow of actions and responses from the user to the system and back. It may also help to create diagrams of different parts of the system and show how those will appear to the user, what the user might want to do with them, and how they relate to the interaction overall. You might not want

to plan everything in your application, but the more that you can plan, the easier the actual building of your application will be.

Build

This is the process of actually putting together hardware and writing code. This is where you'll write, debug, probably research some more, and ultimately assemble your application.

Test

Once you're finished building your application, it's important to test it. Testing can be as complex as creating situations for users to use your application and observing how they use it, or it can be as simple as using the application or hardware yourself and ensuring that everything works.

After testing, you might be ready to present your project by installing it or making it available for download. If your project is a piece of hardware, then you may be ready to prepare it for manufacture, if that's your aim, or to place it in the location and environment that you mean it to be used. You'll want to make sure that everything works as intended before sending a project off, so having a good testing phase is important.

Now that you've learned a little bit about what this book is about and the different ideas that you'll be exploring, you can start learning how to code.

CHAPTER 2
Programming Basics

Writing code is never extremely easy; however, it is also not as difficult as you might imagine. The basis of all programming is simple logic, and programming languages use lots of simple math symbols and English words. So, if you're comfortable with things like equals signs, some very basic algebra, and a smattering of English words, you'll probably do just fine. This chapter is by necessity far shorter than it could be. Learning all of the intricacies of writing code in even a single programming language, much less multiple languages, takes years of study. However, you can easily learn some of the basics, read documents, ask questions on user forums, and use what other people have created, and then find yourself able to create projects and designs within a short amount of time. With that in mind, this chapter is a simple introduction to the fundamentals that the rest of this book relies on.

There are a few types of readers using this book. The first says, "I just want it to work." For you, this chapter will explain just enough that you'll be able to follow along with the examples in this book and the ones that you find online, and be able modify them slightly to get them to do what you need them to do. You'll definitely want to read this chapter, probably more than once, but know that I understand your predicament. I was there once myself. You should know, though, that simply copying and pasting code with no understanding of what it's doing is a very easy way to become seriously frustrated. You should know at least the basics of what your code is doing. Another kind of reader says, "I want it to work, and I want to know why it works." For you, this chapter will be an introduction. You'll probably read it, read some examples, and come back to this chapter again later. That's a good thing. You'll probably also want other books soon. Some of the best are listed in Chapter 18. Another kind of reader may be familiar with some material covered in this book but not others. For you, this chapter will probably be a review and may not even be necessary, but you might want to flip through it just in case you're not familiar with the Processing or Arduino languages or some of basics of C++. If you are familiar with the basics of all of these languages, then you might want to skip ahead to the chapters on the tools themselves or to Chapter 5 for some more advanced material on programming. Whichever type

you are, you should read enough to understand what the code listings in the rest of the book describe.

Why You'll Read This Chapter More Than Once

In all likelihood, the first time you read this chapter, some of it will not make sense. That's perfectly normal and to be expected. As you read through the rest of this book, when you have questions, return to this chapter. There's no substitute for seeing how code functions within the context of something interesting to you. Some of the ideas might not make sense at first, but after seeing them put into practice, being patient, and hacking at them, you'll find that none of this is wildly complicated. Don't be afraid when you don't understand something, don't be afraid to look back at this chapter, and above all, don't be afraid to copy code from this book and change it until it breaks. Hacking at code, taking stuff that works, breaking it, and then figuring out why it breaks is the best way to understand. Of course, the goal throughout this book is to try to provide code that will just work and satisfy the first kind of reader, those of you who simply want it to work and be done with it. In either event, though, this is going to require some patience, because it simply won't *make sense* the first time through. With a little bit of patience, though, and the willingness to experiment, fail, and try again, you'll find that this isn't all that difficult.

The Nature of Code

Throughout this chapter, you'll be learning about programming—writing code, to be more exact—so it's important to know not only what code *is* but how it fits into the process of creating a program. There are a few key terms that you should understand before starting to program:

Code

Code is a series of instructions that a computer will execute when the code is run. It is written in a programming language that, like natural languages, is essentially a contract between two parties. In the case of code, though, the two parties are the programmer and the compiler. You'll learn more about a compiler shortly; for the moment, we'll just think of it as the listener who understands our code. Code can be as simple as adding two numbers or as complex as rendering an animation. What matters is that you write correct instructions to the compiler using a programming language and a text editor of some sort and then tell the compiler what files contain the instructions. Writing code is typing code instructions into a text file that will later be passed to a compiler of some sort. To write a program can mean writing source code from scratch, or it can mean putting several programs together and creating a way for them to communicate. This can also mean configuring prebuilt projects. Creating applications and code doesn't always require

writing code, but if you have this book, then you're probably interested in creating code.

Files
: Code is stored in text files that usually any text editor can open. These files contain your code and nothing else. For larger and more complex projects, you can link multiple files together. Sometimes, larger projects will have many hundreds of code files that will all be linked together to create the sum of the code for the application. Arduino projects use *.pde* files and sometimes *.c* files. Processing projects also use *.pde* files and sometimes *.java* files. openFrameworks projects use *.cpp* and *.h* files. In each case, the different file types do different things and have different purposes that you'll learn more about later in this book.

Compiler
: A compiler is a program that takes a code file (or many code files) and turns it into a series of instructions that a computer will run as a program. Most modern computers do not directly process any instructions that you write; instead, you ask a compiler to turn your code into machine instructions. The compiler optimizes machine instructions for the computer to run very quickly, but they would be very difficult for a person to write, so the better step is to write code in a more human-friendly way and convert it to machine-friendly instructions. This means that when you write code for an Arduino controller or write some Java code, you don't simply run that code; you compile it and have the compiler create an executable file that your computer can run. You can imagine the process of writing code as a series of translations in which you tell the compiler what you want to do with the program that it will create in a high-level programming language like Processing or C++ and the compiler then creates a machine language file that will run that file.

Executable
: An executable is a file that can be run as an application. It is the result of writing code and compiling it. Sometimes the terms *application* and *executable* are used interchangeably, but they're not the same. An application may consist of many executable files, or it may consist of only one. In either case, for the kinds of projects you'll be learning to build in this book, you'll always have an executable.

Now you're ready to get started writing your own code.

Variables

Looking at variables is a good place to start. Everyone remembers variables from their first algebra classes. You might have seen something like this written on a blackboard or whiteboard:

```
x = 5
```

That says, "There's something called *x*, and it is equal to the number 5." Variables in computing are very similar. They represent something, and in fact they always

represent a certain kind of something. That's a little different than the algebra example because that example didn't need to say that `x` is going to be a number; it just says, "x is a number." If you wanted to use code to make a number called `x` and use it as a variable, you would do it like this:

```
int x;
```

The `int` indicates (or *sets*) the type of the variable. You can't just make any type of variable; you have to make specific kinds of variables. The kind of variable is called its *type*; types are dealt with a lot more in the next section, so for right now, let's concentrate on getting the basics of variables down.

```
int x = 5;
```

This code creates a variable named `x` that is an integer and has the value of `5`. What this means is that somewhere in your computer's memory there is something called `x` that is storing the value `5`. You can set the value of `x` right away, as in `int x = 5`, or create it and leave it for later use, as in `int x`. Look at another example:

```
int y = 10;
int x = 5;
int z = x + y;
```

This snippet shows some code that creates variables, gives two of them values, and sets a third variable's value to be the sum of the first two.

Simple Types

In the platforms covered in this book, all variables are *typed*. This means that they have a type that tells the computer what sorts of things are going to be stored in the variable. This is because numbers, letters (usually called *characters*), and `true`/`false` (called *boolean* values) all require different amounts of space to store and move around.

Here are some of the most common types that you'll encounter in each of the programming languages in this book.

int

This is the datatype for integers, which are numbers without a decimal point, like 2 or 20,392. Now, we're about to run into our first real problem in this chapter: there are three programming languages used in this book; they all have similar things, but they sometimes work just a little bit differently. In Processing, integers can be as large as 2,147,483,647 and as low as –2,147,483,648. In Arduino and C++, the languages that openFrameworks (oF) uses, things work a little differently, and understanding why requires a quick explanation of signed and unsigned variables. The next small section explaining signed and unsigned variables might be a bit heavy at first, so you may want to skim over it the first time, particularly if you're more interested in working with Processing.

Signed versus unsigned

As mentioned earlier, variables need to be declared with a type because the computer needs to know how much space they will take up. That means that sometimes an int is 4 bytes or 32 bits of information all the time and that it can store 4 bytes worth of information. So, what happens when you need a negative int? Well, you have to store whether it's negative or positive, and the way you do that is by moving all the values of the int down into the negative values. For the moment, you'll see how this works in C++, the language that oF uses, and then the rest of the languages will be addressed. In C++, where the int represents a value in between –2,147,483,647 to 2,147,483,647, if you have an int that is never going to use a negative number, you can use an unsigned int that stores anything from 0 to 4,294,967,295. This is what's called having *signed* or *unsigned* variables. Unsigned variables don't store negative numbers, whereas signed variables do. The way this works requires a little bit of thinking about binary numbers.

An int is 32 binary values. Binary counting uses 1 and 0 to count instead of 0 through 9 like we're used to doing. This means that when you want to set a variable to 1, you'll actually be storing the value 1 in the place where the variable is stored. When you want to store the value 2, you'll be storing 10, which means 1 in the 2's place and nothing in the 1's place. Additionally, 3 is represented as 11, which means 1 in the 2's place and 1 in the 1's place, for 2+1, which is 3. Further, 4 is represented as 100, which is 1 in the 4's place and nothing in the 2's or 1's place. Therefore, 16 is 1,000, 17 is 1,001, 18 is 1,010, and so on. This is called *two's complement math* because it counts everything up by squaring for the next value. That's a very quick introduction to binary math, but don't worry, there's a much more comprehensive discussion of this in several chapters later in this book where you begin using bits and binary.

Figure 2-1 shows the ways that signed and unsigned variables work. Remember that in C++ the int is 32 bits, so there will be 32 boxes to show what each bit stores.

In unsigned numbers, the first bit is counted like any other bit. In signed numbers, the first bit is used to store whether the number is positive or not, and this is why unsigned variables can store larger values than signed variables.

Arduino and C++ use unsigned variables, and all variables are signed unless you indicate otherwise by putting `unsigned` in front of the variable type:

```
unsigned int = 5;
```

Processing does not use unsigned variables; all numerical variables in Processing are signed.

In C++, the language you'll use with oF, signed ints are –2,147,483,647 to 2,147,483,647, and unsigned ints are between 0 and 4,294,967,295. In Arduino, the int can be between –32,768 to 32,767. Tricky? Not really. Lots of times you won't have to think about what you're storing in the int. When you do have to think about how large the value you're going to be storing is, it's time to use a bigger number type, like long or double. We'll discuss these later in this section. When using Processing, you

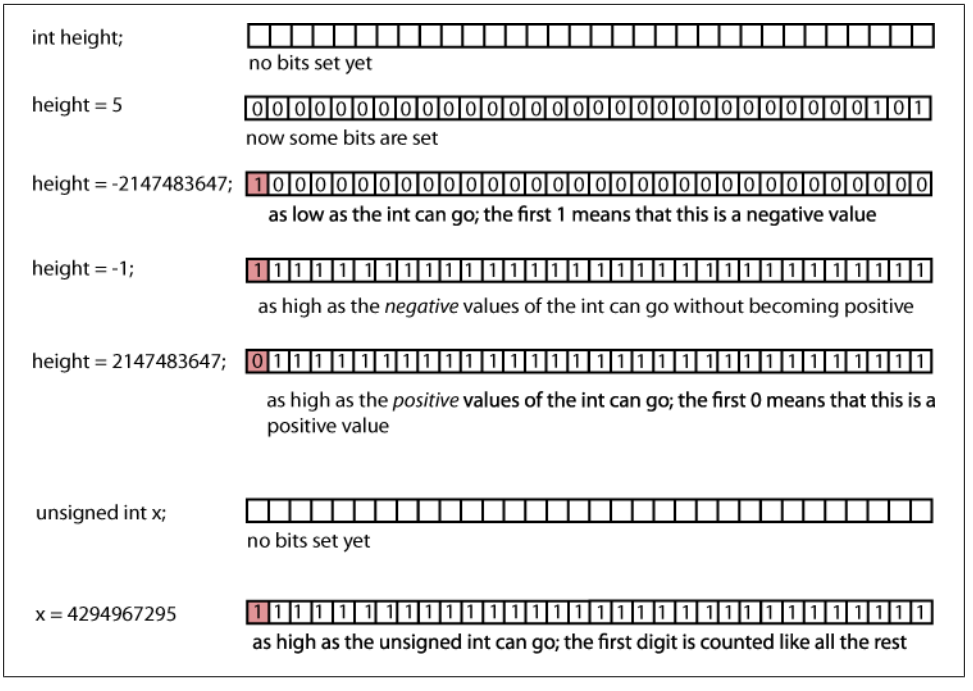


Figure 2-1. Setting the bits of signed and unsigned variables

never have to think about how large the value will be at all, so it isn't a problem. Simply figure out whether you'll need a decimal point. If you do, then use a float; if you don't, then use an int.

float

The *float* is the datatype for floating-point numbers, which are numbers that have a decimal point. Floating-point numbers are often used to approximate analog and continuous values because they have a greater range of values than integers. Signed float variables can be as large as 2,147,483,647 and as low as –2,147,483,647. Float variables aren't signed in C++ or Arduino.

char

This type can contain characters, that is, single letters or typographic symbols such as A, d, and $. Here are two `char` variable declarations:

```
char firstLetter = 'a';
char secondLetter = 'b';
```

So, what is different about floats, ints, and chars? char variables can sometimes be added together like numbers, but it's not going to work the way you think it will. When working with Processing or Arduino, it's not going to work period, and with C++ in openFrameworks it will work, but not as you might expect:

```
char thirdLetter = a + b; // this won't become 'ab' or 'c'
```

Like I mentioned earlier, Processing and Arduino aren't going to like this at all. They're just going to throw errors at you and complain. C++, however, will work, and the next section will explain why.

ASCII

There are a few ways that characters in a computer are stored, and one way is as numbers in a system called American Standard Code for Information Interchange, or ASCII. In ASCII, `a` is 97, and `b` is 98. That's why if you run the following snippet of code through a C++ compiler or through Processing or Arduino, it'll happily say that `c` is 195:

```
char a = 'a';
char b = 'b';
int  c = a+b; // notice that a+b is being stored as an int
```

You can store the result of two characters being added together *if you store the value of the result as an int* because adding the characters together is really adding the numbers that they are stored as together.

All keyboard-related things can be stored as a char. That includes particular keys on the keyboard. These are called *character escapes*, and they have a particularly special place in programming because this is how you determine when the user has clicked a key that can't be represented with a letter or number. There are other encoding systems that correlate numbers to letters, Unicode Transformation Format, or UTF, is a very popular one that you might hear mentioned. Encoding is a complex topic and the most important thing to understand is that the char is both a number and character.

bool or boolean

Boolean values store two possible values: `true` and `false`. In C++ and Arduino, the `true` is actually a 1, and the `false` is actually a 0. It's usually clearer to use `true` and `false`, but you can use 1 and 0 if you like. They're very handy for storing very simple results: you received a message from the server or you didn't, the user clicked the F key or they didn't.

For Arduino and Processing, you do the following:

```
boolean b = true;
```

In C++, you do this:

```
bool  b = true;
```

Note that the Processing and Arduino version uses the word `boolean`, while C++ uses the word `bool`.

Both `true` and `false` are reserved words. That means that the compiler only recognizes them as being values that you can use to set Boolean variables. This means that you can't, for example, do this:

```
boolean true = false;
```

It will cause errors, which is something that, although unavoidable, you generally want to restrict to more interesting things that will help you create your projects.

string

A *string* is a sequence of characters. This is both a helpful, metaphorical way to think about it and a matter of fact, and it is something that you'll investigate more closely later. The string includes helpful methods for looking at individual characters, comparing strings to see whether they're the same or different, searching strings for parts of the string (for example, finding "toast" in "toaster"), and extracting parts of strings (getting "toast" out of "toaster"). The differences between a string and a char is that a string is always defined inside double quotes (`"Abc"`) and can consist of multiple characters, while char variables are defined inside single quotes (`'A'`) and can consist of only one character.

Here are some strings in Processing:

```
String f = "foo";
String b = "bar";
String fb = f+b;// this will be "foobar"
```

Here are some strings in C++:

```
string f = "foo";
string b = "bar";
string foobar = f+" "+b;// this will be "foo bar" note the space w/in quotes
```

Note that Processing uses `String`, while C++ uses `string`. Why isn't there any Arduino example? Arduino doesn't have a `String` or a `string`, because it doesn't need one. We'll have to leave the explanation for this until the next section. Suffice to say, it's not a huge problem.

Since a String is defined within quotes:

```
String super = "super";
```

including quotes in a String requires the (backslash) character to be used preceding the quote:

```
String quoted = "He said \"It's super\".";
```

This is known as an *escape sequence*. Other escape sequences include `t` for the tab character, which is written `\t` with the escape character, and `n` for newline, which is written `\n`. A single backslash is indicated by `\\` because without the `\` in front, the compiler would think that you're writing an escape sequence. If you wanted to write the characters *\that's great!*, you'd write this:

```
String quoted = "\\that's great!";
```

Otherwise, without the double backslash, you'd end up with this:

```
"    hat's great!"
```

The compiler would assume that the `\t` is a tab.

byte

The *byte* is the datatype for bytes, that is, 8 bits of information storing numerical values. In Processing the byte stores values from –128 to 127 (a signed value), and in the Arduino it stores from 0 to 255 (an unsigned value). Bytes are a convenient datatype for sending information that can't or shouldn't be represented by strings or other kinds of numbers such as the contents of a file, data sent to or from hardware, data about the pixel of an image, or a millisecond of sound.

Here are bytes in Processing and Arduino:

```
byte firstByte = 55;
byte newByte = 10+firstByte;
```

Now, the byte `newByte` is just a value waiting to have something done with it. If you were to treat `newByte` like a char, it would be A and if you treat it like an int, it would be 65. Because bytes are so lightweight, they often are used to represent data that needs to be read or written, or sent quickly, or as data that doesn't have an easy representation in other formats.

C++ does not have a byte type. Where you would use a byte in Processing or Arduino, you use a `char` in C++. Why is this? Remember that the `char` is an ASCII number value, that is, a number between either –128 and 127 or 0 through 255, that represents a letter or character. The designers of C++, taking a cue from the C programming language that preceded it, being as efficient as possible, recognized that this value could be used not only for characters but for anything. Hence, in C++ you use the `char` more frequently than in Processing or Arduino.

long

The long variable allows you to store very large nonfloating-point numbers, like an int. In C++ and Arduino, the long can store values from –2,147,483,648 to 2,147,483,647. In Processing, the maximum value is considerably larger: 18,446,744,073,709,551,615.

Arrays

The array is a slightly more complex datatype than the ones that were shown earlier. An *array* contains one or more variables in a list. Remember one of the problems that you're going to run into in this book is that there are three different programming platforms used in this book, each of which does things slightly differently. We'll look at three different arrays filled with integers in each of our platforms to give you a feel for the similarities and differences.

The array is a list of multiple elements; in the code snippet below, it will contain integers. You're going to create an array that holds three `int` values: 1, 2, 3. It's nothing too thrilling, but it's a good place to start.

Note the markers above each of the elements in the array in Figure 2-2: `numbers[0]`, `numbers[1]`, `numbers[2]`. These are the array access operators. That's not a typo; they do count up from 0. We'll come back to these after we look at creating the array.

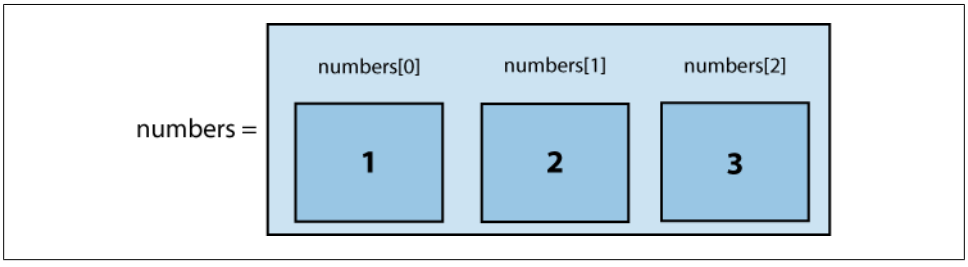


Figure 2-2. An array with three integers

Here's what the array looks like in Processing:

```
int[] numbers = new int[3];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
```

First, look at the declaration of the array, as shown in Figure 2-3.

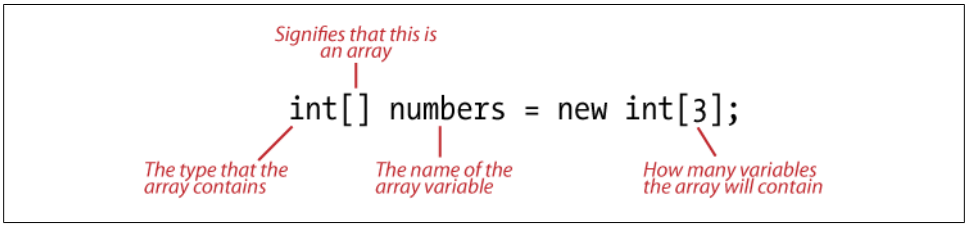


Figure 2-3. An array containing three integers as declared in Processing

You can declare arrays to be filled with any type, even types that you create yourself. We'll get to this later. The code snippet sets all the values of the array. When you want to access an element in the array, you use the access operators, `[]` to access the index position in the array that is specified within the square brackets:

```
int x = numbers[0];
```

The following sets `x` to `1`, because the first item in our `numbers` array is `1`:

```
int y = numbers[1] + numbers[2];
```

The following sets `y` to the values of the second plus the third elements of `numbers`, that is, 2 plus 3, which equals 5. If you use the `=` operator in front of the array with the access operator, it sets the value of the element in the array at that point:

```
numbers[0] = 1;
numbers[0] = 5;
```

Whereas `numbers[0]` originally was `1`, it is now `5`. The array is really storage, and each element in the array is like a box that can store any variable of the type declared when the array is created. When any item within the array is set, it will store that value until it is changed.

Instead of putting each number into the array on a separate line of code, the array values can be created all at once:

```
int arr[] = {1, 2, 3};
```

or:

```
int arr[3] = {1, 2, 3};
```

Note that above the array does not need to have a number length assigned to it. This results in the same array as creating the array in the first example; however, it assumes that you know all the values that will go into the array. If you do not, use the first method described.

The next language to look at is the declaration of an array in Arduino. Luckily for the sake of brevity, the Arduino platform deals with arrays very similarly to Processing. Creating an array in Arduino or C++ can be done in any one of the three following ways.

Here's what the array looks like in Arduino or C++:

```
int arr[] = {1, 2, 3};
```

or:

```
int arr[3];
```

or:

```
int array[3] = {1, 2, 3};
```

Figure 2-4 breaks down the parts in the array.

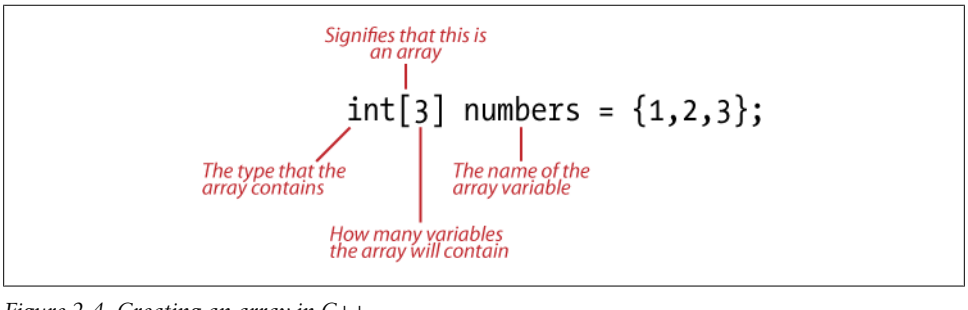


Figure 2-4. Creating an array in C++

The only difference between the array in Processing and in Arduino is that while Processing supports the `new` keyword when creating an array, Arduino does not. If you know all the elements when you're creating the array, do it like so:

```
int array[3] = {1, 2, 3};
```

Otherwise, you should construct an array like this:

```
int[] arr = new int[3];
```

Now that we've covered the basics of arrays, you'll look at some of the more complex aspects of dealing with the array. The first is the error that you're sure to encounter at some point, specifically, the "out of bounds access" error in all its many forms. What this means is that you've tried to access an element in the array that does not exist.

In Arduino and C++, arrays can be initialized as shown here:

```
char arr[3] = {'a', 'b', 'c'};
char badArrayAccess = arr[3];
```

When you try to do something with the `badArrayAccess` variable, you'll find that not only is it not a char, but that accessing it might just crash your program. This is because the array `arr` contains three char variables, and you're trying to access a fourth one, which violates the number of elements established when declaring the array. Another mistake is trying to store the wrong kinds of variables in the array, as shown here:

```
char arr[3]; // an array of chars, with space for 3 variables
float f = 2.3;
arr[0] = f; // oops! arr expects chars, so will turn f into "2"
bool b = true;
arr[1] = b; // oops! arr expects chars, so will convert b into "1"
```

This won't cause an error; however, it won't return the variables correctly, because the array declared is expecting char variables, so when you place float and bool variables within that array, it automatically converts them to char, with frequently unpredictable results. Table 2-1 is a quick comparison of the datatypes and how the three languages use them.

Table 2-1. Comparison of datatypes

Arduino	Processing	C++	Use
int	int	int	A number without a decimal point, for example: 4 or –12
float	float	float	A number with a decimal point, for example: 1.23 or –128.12
char	char	char	A single character or number that will be treated as a character, for example: `a`, `1`, `!`
None (use char[] instead)	String	string	A grouping of characters together, for example: hello, 172 Cherry Street
byte	byte	None (use char instead)	The value of a byte, between –128 and 127 if the byte is signed and 0 and 255 if it is not signed
boolean	boolean	bool	A `true` or `false` value
double (but same as float)	double	double	A floating-point number with higher precision than float

Casting

What do you do if you have a number that is an integer but you need it to be a float? You *cast* it. This means that you convert it from its original type into a new type:

Here's what casting looks like in Processing, Arduino, and C++:

```
int i = 5;
float f = (float)i; // float is now 5.0
```

Why can't you simply set `f` to `i`?

```
float f = i;
```

Types won't allow you to do that, because that's the point of a type. You know what specific type values are, how much memory they require, and how they can be used. The previous code will cause an error if you try to run it. Instead, cast the variable. You can't change the type of a variable once it's declared. In this example, `i` will always be an int, but if you need to use it as a float, you can very easily cast it. Some other examples of casting involving char variables because they are such a fundamental datatype. For instance, if you want to cast a char to a float and then back you do the following:

```
char ch = 'f';
int fAsInt = (int)ch;// now fAsInt is 102
char newChar = char(fAsInt); // newChar is now 'f'
```

One important thing to note is that you cannot cast an array:

```
char[] foo = {'a', 'b', 'c', 'd'};
int[](foo); // ERROR
```

To cast anything in an array, you need to cast every element of the array. We'll cover this in greater detail in the sections on loops, but for now, we'll look at this:

```
char[] foo = {'a', 'b', 'c', 'd'};
int i = int(foo[0]);
```

Here you create an array of char variables and then read one of those char values out of the array and cast it as an int.

Operators

Operators are the symbols that a compiler uses to perform commands and calculations in your program. Operators let you set variables like the `=` operator, compare variables like the `==` operator, add variables like the `+` operator, and so on. There are three major types of operators. The first operators are the mathematical operators that perform mathematical operations. These should be familiar from any math class you've ever taken. The second are assignment operators that change values. The third are comparison operators, which determine whether two variables are equal, different, greater than, or less than another variable.

Mathematical operators work pretty much as expected, with one notable exception. `+` adds two values, for example:

```
int apples = 5
int moreApples = apples + 3; // moreApples is equal to 8.
```

The exception occurs when you add strings; you end up with the two strings stuck together:

```
string first = "John";
string second = "Brown";
string full = first+second; // now full is JohnBrown
```

This is because of something called *operator overloading*. If you're curious enough, you can check some of the recommended programming books from Chapter 18 for a full description, or look online because it's a more advanced topic.

The other simple mathematical operators are `-` (subtraction), `*` (multiplication), and `/` (division). The last mathematical operator is `%`, the modulo.

The modulo tells you what is left over (the *remainder*) when the value on the left is divided by the value on the right. Here are some examples:

```
8 % 2 // equals 0 since there's no remainder when 8 is divided by 2
17 % 2 // equals 1 since there's a remainder of 1 when 17 is divided by 2
19 % 5 // equals 4 since the remainder is 4 when 19 is divided by 5
12 % 11 // equals 1 since the a remainder of 1 when 12 is divided by 11
19.0 % 5 // equals 4.0 since we're dealing with floats
```

Assignment operators work from right to left. That is, the operator uses the value on the right to assign or change the variable on the left. For example:

```
int age = 6;
```

The `=` operator sets the variable on the left to `6`. There are other types of assignment operators that change the value of the variable on the left, just like the `=` operator. For instance:

`+=`
: Adds the value on the right to the value on the left:

```
int height = 6;
height += 1; // height is now 7
height += 10; // height is now 17
```

`-=`
: Subtracts the value on the right from the variable on the left:

```
int size = 16;
size -= 8; // height is now 8
size -= 6; // height is now 2
```

`++` and `--`
: Add one or subtract one from the variable on the left:

```
int hearts = 2;
hearts++; // hearts is now 3
hearts--; // hearts is now 2 again
hearts--; // hearts is now 1
```

*= or /=
: These work roughly the same as the += and -= statements, multiplying or dividing the variable on the left by the value on the right:

```
int i = 15;
i /= 3; // i is now 5
int j = 20;
j /= 2; // j is now 10
float k = 100.0;
k /= 3.333333; // k is now 30.000004
float m = 100.0;
m /= '2'; // not a good idea
i *= 4; // i is now 20
m *= 0.5; // m was 2.0 and is now 1.0
```

Comparisons are very important in programming and particularly important in using control statements, which will be discussed in the next section. Before we get to that, though, you need to become familiar with the various operators. These operators are common across all the programming languages discussed in this book and, with slight variations, across all programming languages. Comparisons allow you to determine whether variables are the same, different, greater than, or less than one another:

== (equal to)
: Compares whether two things are equal. For example:

```
5 == 4 // false
'a' == 'a' // true
(12 / 3) == (2 * 2); // true
4.1 == 4 // false
char(102) == int('f') // true, because 102 is 'f' in ASCII
"duck" == 0.002 // false, because it doesn't make any sense
```

!= (not equal to)
: Checks whether things are not equal, for example:

```
3 != 1 //true, they're not equal
'B' != 'b' // also true, they're not equal
```

> (greater than)
: Checks whether the value on the left is *greater* than the value on the right, just like in math class:

```
4 > 3 // true
5 > 192901.2 //false
"fudge" > 8 // false, because it doesn't make any sense
```

< (less than)
: Checks whether the value on the left is *smaller* than the value on the right, again, just like in math class:

```
3 < 2 // false
'g' < 106 // since 'g' is 103 in ASCII this is true
-100 < 100 // true
```

`>= (greater than or equal to)`
: Checks whether the value on the left is *greater than or equal to* the value on the right, just like in math class:

```
3 >= 3 // true, since they're equal
0 >= -0.001 // since 0 is greater than -0.001, this is true
'?' >= 'h' // true, since '?' is 63 in ASCII and 'h' is 104
4 >= 28 // false
"happy" >= "sad" // false, because it doesn't make any sense
```

`<= (less than or equal to)`
: Checks whether the value on the left is *smaller than or equal* to the value on the right, again, just like in math class:

```
13.001 <= 13.001 // true, since they're equal
0 <= -0.001 // since 0 is greater than -0.001, this is false
'!' <= '7' // true, since '!' is 33 in ASCII and 'h' is 55
```

`&&`
: Evaluates the statement on the left and the statements on the right and returns `true` if they are *both true*:

```
(4 > 3) && ('f' > '1') // true
((5 * 2) == 10) && ((6 - 3) != 4) // true
(5 < 10) && (2 > 4) // false, even though the left is true, the right isn't
```

`||`
: Evaluates the statement on the left and the statements on the right and returns `true` if *either one of them is true*:

```
(4 < 3) || ('f' > 'e') // true, left isn't true but the right is
((5 * 2) == 10) || ((6 - 3) != 4) // both are true
('b'=='g') || (2 > 4) // false, none of them are true
```

You may be wondering what to do with all these evaluations and comparisons. The answer is control statements. Table 2-2 lists operators and their uses.

Table 2-2. Operators and their uses

Operator	Use
+, -, *, /	Adds, subtracts, multiplies, and divides.
%	Modulo; returns the remainder of a division.
=	Assignment; assigns the value on the right to the variable on the left.
+=, -=, *=, /=	Mathematical assignment; adds, subtracts, multiples, or divides the value on the left by the value on the right and sets the value on the right to that result.
++	Adds 1 to the value to the left.
--	Subtracts 1 from the value to the right.

Operator	Use
==	Compares the value on the left with the value on the right. If they are equal, then the expression is true.
!=	Compares the value on the left with the value on the right. If they are not equal, then the expression is true.
>, >=	Compares the value on the left with the value on the right. If the value on the left is greater than or greater than or equal to the value on the left, the expression is true.
<, <=	Compares the value on the left with the value on the right. If the value on the left is lesser than or greater than or equal to the value on the left, the expression is true.
&&	Checks the truth of the expression to the left of the operator and to the right; if both are true, the entire expression is true.
||	Checks the expression to the left of the operator and to the right of the operator; if either is true, the entire expression is true.

Control Statements

You'll often want to control how the logic of your program flows. If one thing is `true`, then you'll want to do something different if it's `false`. You might also want to do something a certain number of times or until something becomes `false`. There are two kinds of control statements that you can use to create this kind of logic flow in your application: *conditional logic* statements, which check whether something is `true` or `false`, and *loop* statements, which do something a set number of times or until something becomes `false`.

if/then

The `if`/`then` is a conditional logic statement, and it works just like it does in English: "If it's raining, then I'll bring an umbrella." These statements look like this:

```
if(condition) {
    result if the condition is true
} else {
    result if the condition is false
}
```

There must be a `true`/`false` expression in the brackets next to the `if`. Here's an example:

```
int myWeight = 72;

if(myWeight > 100) {
    print(" you're getting heavy! ");
} else {
    print(" you're still doing ok ");
}
```

You can also use the `if` statement without an `else`:

```
int myHeight = 181;

if(myHeight > 200) {
```

```
    print(" you're too tall ");
}
```

This just means that the actions inside the brackets will execute if the statement is `true`; otherwise, you don't do anything. You can also check that things are not `true`:

```
boolean allDone = false;

if(!allDone) {  // if not all done
    print(" keep going! ");
} else {
    print(" ok, quit! ");
}
```

There is one last permutation on this pattern:

```
if(age == 5){
    print(" you're 5!");
} else if(age == 25) {
    print(" you're 25!");
} else {
    print(" can't login "); // if neither of the above is true
}
```

In this example, there is a new term introduced, `else if`. What this does is evaluate another statement before going on to the `else` statement. This means that if the first statement doesn't evaluate to `true`, then check each `else if` looking for the first one that is `true`. If none is `true`, then do the `else` statement.

for Loop

The `for` statement lets us do things over and over again, for a specified number of repetitions. Loops are frequently used to loop through arrays and examine, use, or alter each element within them. This is going to be particularly useful when dealing with the pixels in images or frames of video, as well as sounds, data from the Internet, and many other kinds of information that needs to be sorted through:

```
int i;
for(i = 0; i < 10; i++) {
    print(char(i)+", "); // this will print 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
}
```

The `for` statement uses three different statements, as shown in Figure 2-5.

The integer `i` is used to set the number of times that the loop will execute, running all the code inside the code block. In the initialization of the `for` loop, you set `i` to 0; as long as `i` is less than 10, you continue looping, and each time the `for` loop is passed, `i` is incremented by 1.

Of course, it is also entirely possible to use subtraction in the `for` loop:

```
for(int i = 5; i>-1; i--){
   print(i);
}
```

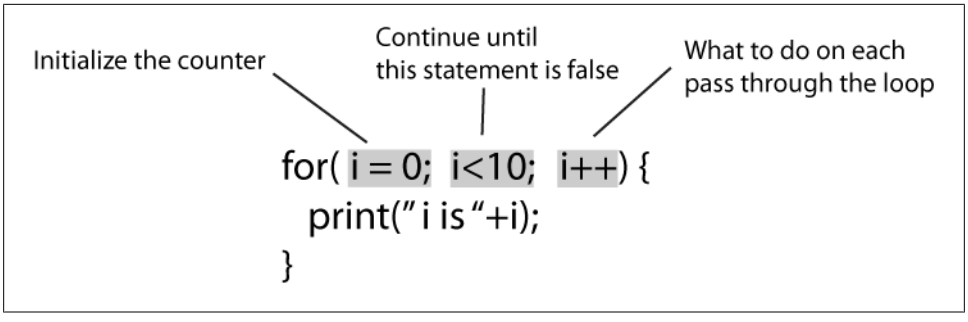


Figure 2-5. A for loop

Another great use of the `for` loop is to go through an array and look at each item in the array:

```
int[] intArr = {0, 1, 2, 3, 4, 5};
int sum = 0;
for(int j = 0; j < 6; j++){
    sum += intArr[j]; // we'll get the item in the array using the loop
}
```

Each time the loop executes, `sum` will be incremented using the next integer in the `intArr` array.

while Loop

The `while` loop is similar to the `for` loop, but it is slightly less sophisticated:

```
while(trueOrFalse){
//something to do each time
}
```

As long as the expression that the `while` is evaluating each time through remains `true`, the loop will continue executing. This is important to note because if the evaluated variable does not become `false`, the loop will never exit, and your program could become locked up while the loop runs. Creating an *infinite loop*—that is, a loop that never exits—is easy and will certainly make your program unusable, so take care when using any of the control statements. Here's an example using a counter, much like in the `for` loop:

```
int j = 0;

while(j < 100) {
    print(" what's up? "+j);
    j++; // if j is not incremented we end up with an infinite loop
}
```

Each time the loop executes, the integer `j` is incremented. When `j` is no longer less than 100, the loop is exited, and the program continues on:

```
boolean ready = false;
float number = 0.0;

while(ready != true) {
    print(" we're just waiting" );
    number += 0.1;
    if(number > 1.0){
        ready = true;
    }
}
print(" we're ready ");
```

In this example, you increment a number and use a second variable, a Boolean value, to determine whether you are in fact ready. Until the `ready` variable is `true`, the loop will continue executing, and the "we're ready" message will not be printed.

continue

The `continue` statement tells a loop to skip over any further instructions and go on to the next repetition. For instance, if you wanted to loop through a certain number of items and process only odd numbers, you could do this:

```
for(int i = 0; i< 10; i++) {
    if( i % 2 == 0){
        continue;
    }
    println(i + " is not divisible by 2");
}
```

This will print the following:

```
1 is not divisible by 2
3 is not divisible by 2
5 is not divisible by 2
7 is not divisible by 2
9 is not divisible by 2
```

The `continue` statement here starts the loop over again. If you have a complicated operation to perform, then using the `continue` statement allows you to evaluate whether you need to run that operation and to skip it if you do not. This means your code does only what is necessary.

break

The `break` statement breaks the loop. It is a great tool to use when you're looping through an array looking for something and suddenly you find it. Once you've found it, you can quit looping easily by using the `break` statement:

```
int[] intArr = {1, 2, 3, 4, 5, 6, 2, 12, 2, 1, 19, 123, 1, 3, 13};
int counter = 0;

while(counter < intArr.length)
{
```

```
    if(intArr[counter] == 5) { // we're looking for 5
        print(" found it at ");
        break;
    }
    counter++;
}
// code will continue from here after the break
print(counter); // found the value 5 in intArr[4]
```

Functions

What is a function?

A *function* is a name for a grouping of one or more lines of code and is somewhat like a variable in that it has a type and a name. It's very much unlike a variable in that it doesn't just store information; it manipulates it. Going back to the basic algebra analogy you saw earlier when first looking at variables, a variable is something like this:

```
x
```

A function is something more like an instruction with something that is given at the beginning of the instruction and something that is expected in return. Writing out a function in simple English, you might see something like this: "When someone gives you some money, add the amount given to you to the amount that you already have and report that number back to me."

There are a few parts there that can be broken out:

- An amount of money that you are given
- Adding the amount given to you to the amount that you already have
- Reporting that number

Those three things can be thought of in order as follows: what the function takes or what will be passed in to it, what it does, and what it returns. Functions are set patterns of behavior in your program that take certain kinds of values, do something with those values, and return something when they are finished.

Defining a Function

To turn the written example into code, you would define a variable to be used to keep track of all the money that you have:

```
int myBank = 0;
```

Then you create a function that takes money, adds it to `myBank`, and returns that value:

```
int moneyReceived(int money){
  myBank += money;
  return myBank;
}
```

Now, you have defined a function called `moneyReceived()`. Using pseudocode or talking though what a function is supposed to do can be quite helpful when trying to understand it. "Take an integer, add it to the money already in the bank, and return that value." Figure 2-6 is a diagram that helps walk you through what the function is supposed to do.

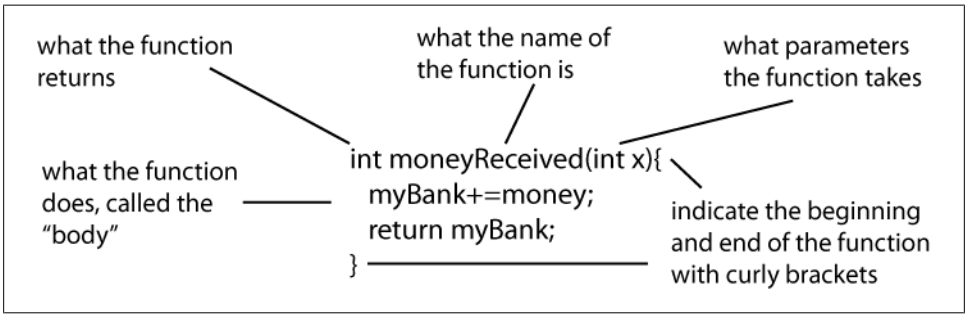


Figure 2-6. Declaring a function

Notice that the return statement is returning something that is of the same type as the type listed in front of the function name. The variable `myBank` is an int that the `moneyReceived()` function returns.

Passing Parameters to a Method

Once the function is defined, it's time to call it. To call the function, you need to pass the correct kinds of parameters to it, which requires knowing what parameters the function requires. In the `moneyReceived()` function example, the required parameter is a single integer. Both of the following examples are legal:

```
moneyReceived(4);
```

or:

```
int priceOfDinner = 17;
moneyReceived(priceOfDinner);
```

This next one is not legal:

```
float f = 1.32;
moneyReceived(f);
```

because the variable being passed to the function is not the correct type. This is why being able to see the function declaration, that is, where its return type, name, and parameters are established, is so important.

The return statement indicates the type of value that the method will return, just as a type for a variable determines what kind of value the variable will store. Functions that don't return anything are declared `void`, and any other kind of return value needs to be

declared. For example, if you created a function that would return a `char`, you would declare it like so:

```
char myFunction()
```

Note that the function always has the following pattern: type, function name, parentheses, and any parameters that need to be passed to the function:

```
int multiplyByTwo(int value){
    return value * 2;
}
```

This next function takes an integer value, multiplies that value by 2, and then returns whatever that value is. When a function has a return type, it can be used to set the values of variables:

```
int x = 5;
int twoTimesX = multiplyByTwo(x); // twoTimesX is now 10
```

Here, you use a function to return a `char` based on the value passed into the function:

```
char convertIntToChar(int i){
    char ch = char(i);
    return ch;
}
```

New variables can be set using the values returned from a function:

```
string addExclamationPoints(string s) {
    return s+"!!!";
}
string myStr = addExclamationPoints("hello"); // myStr will be 'hello!!!'
```

That's a bit tricky to see what's going on at first, but you simply have to understand that the function is going to become equal to a string when it's all done with its operations, which means that you can treat it like a string. Your program will take care of making sure that everything within the function will be run before it gets used to set the value of the variable.

That's why having functions typed is so important. Anything typed as an integer can be used to create a new integer:

```
int squareOfEight = square(8);
```

This will work if the `square()` function looks like this:

```
int square(int val) {
    return val*val;
}
```

Since `square()` returns an `int`, you can use it set an `int`. If it returned a float or another datatype, it wouldn't work, which makes reading and understanding the return types of functions so important.

Some Suggestions on Writing Functions

Name them well. Functions should do what their names indicate. A function called `square` is very well named if it takes an integer or float as a parameter and returns the square of that value. It's not so well named if it doesn't do that. Generally thinking of functions as verbs is a good way to go because it makes you think about what that thing should do in the first place and help you remember what you meant it to do later when you have to look back at it.

Make them no smaller than sensible and no larger than necessary. A function of 200 to 300 lines of code should probably be broken down into smaller functions. This helps you reuse different parts of the code in different places, where necessary, and helps you locate problems more quickly when they occur. Being able to isolate a problem to several lines of code instead of several hundred can save you hours.

When you find yourself needing to do the same tasks repeatedly, put the code into a function and call that function. For instance, if you are frequently resizing images and saving them to a folder somewhere online, you could make a function like this:

```
resizeAndSave(int picHeight, int picWidth, String urlToSave)
```

This cleans up your code, saves you typing, and makes debugging problems easier. The less stuff you have to look at, the more quickly you'll be able to find things, locate problems, and remember where you left off in a piece of code.

Overloading Functions

Function declarations are important for two reasons. First, the function declaration tells you what parameters to pass to the function and what kinds of things you can do with any values returned from a function. Second, the function declaration and the parameters that the function take are unique for the function that the compiler uses. A function that accepts two strings is considered different from a function that accepts three strings, *even if those functions have the same name*. Making multiple functions with the same name that accept different parameters allows you to use the same functionality to handle different situations. This is called *overloading* a method, allowing it to accept multiple sets of parameters.

Think of a verb like "draw." It's easy for us to differentiate between "drawing a picture" and "drawing a card" because of the context. The compiler treats functions that same way, which makes it easier for us to name functions with the same functionality by the same name and rely on the compiler to differentiate between them using the context of the kinds of parameters passed. Sizes for a video could be set using int values or float values, for example. The compiler considers each of these functions to be completely separate, even if they have the same name. When the function is called, if float values are passed, then the function of that name that accepts floats is used.

Here is an example of an overloaded function in Processing:

```
char multiplyByTwo(char value){

    return char(int(value) * 2);
}

String multiplyByTwo(String value) {
    return value+value;
}

int multiplyByTwo(int value){

    return value * 2;
}

int[] multiplyByTwo(int value[]){
    for(int i = 0; i<value.length; i++) {
        value[i] *= 2;
    }
    return value;
}
```

This function accepts an int, a String, a char, and an array of ints (`int[]`). The version of the function that will be called is dependent on what is passed to it.

Here it is in Processing:

```
println(multiplyByTwo('z'));// this will print ô
println(multiplyByTwo(5)); // this will print 10
println(multiplyByTwo("string"));// this will print stringstring
int[] foo = {1, 2, 3, 4};
println(multiplyByTwo(foo));//this will print 2, 4, 6, 8
```

Overloading functions is a great tool because it allows you to make a single way to do something and modify it slightly to take different kinds of variables as parameters. One thing to note is that although having different types of parameters, as shown here, is fine:

```
int function(int i) {
}

int function(float f) {
}

int function(char c) {
}
```

having different return types does not always work:

```
int function(int i) {
}

float function(float f) {
}

char function(char c) {
}
```

This will throw an error in Arduino and C++ but not in Processing. Generally, it's not the best idea, but it's your code, your project, and your call.

Objects and Properties

So, what's an object? An *object* is a grouping of multiple properties into a single thing that represents all those different properties. That sounds wretchedly abstract, but it's actually quite simple: think of a chair. A chair has a height, a weight, and a number of legs. So, those are three different properties of a thing called a chair. If you were to think about how you'd represent that in code, you'd probably say something like this:

```
Chair myChair;
myChair.height = 22;
```

See the dot (`.`) after the name of the `Chair` instance but before the property? That's the *dot operator*, and in a lot of the most popular programming languages today it's the way that you access the properties of an object to use or set them. Let's look more at what to do with the chair:

```
myChair.legs = 4;
myChair.weight = 5;
```

So, now you've made a chair called `myChair` that has some different properties: `legs`, `weight`, and `height`. Well, that's not particularly useful. Let's think of something that might be a little bit more useful to you. The following code snippets will all be using Processing:

```
PImage img = loadImage("mypicture.jpg");
image(img, 20, 10);
img.x = 300; // set the x position of the image
img.y = 300; // set the y position of the image
```

The previous code might be a little mysterious at first, but let's look at the parts that you can make sense of right away. `PImage` is an object in Processing that lets you load and display image files, resize them, and manipulate their pixels. The explanation of all that needs to wait until the next chapter, which focuses on the Processing language. For the moment, you want to focus on these two lines:

```
img.x = 300; // set the x position of the image
img.y = 300; // set the y position of the image
```

What you know after looking at these lines is that the `PImage` has an `x` property and a `y` property. So, when you create a `PImage` object, like our `img` object created in the previous code snippet, you can get or set those properties. That means that you can tell the object where the image should be, or you can find out where the image is. It all depends on which side of the equals sign you're placing the image on:

```
int xPosition = img.x; // get the image x position
img.x = 400; // set the image x position
```

Now, let's look at another thing that an image can do, which is copy all of its pixels and return them to be used by another image. The `PImage` does this by using the `copy` function. Let's take a look at calling that function:

```
img.copy();
```

Notice the `.` operator there again? Just like the properties, the `.` operator is how you call functions as well. You might think that's confusing, but it's actually sort of helpful. The `.` indicates that you're using something within that object. So, the `PImage copy` function is another property of the `PImage`, just one that happens to be a function.

Methods versus functions: usually functions that are part of something like the `PImage`—for instance, `copy`—are called *methods*. The difference between them is mostly semantic and isn't something you should worry about. Just remember that if you see the term *method*, it's referring to a function that belongs to an object like `PImage`.

Now, if you happen to look at the documentation for the `copy` function on the Processing website, you'll see something that looks like Figure 2-7.

Class PImage

Name copy()

Examples

```
PImage img = loadImage("tower.jpg");
img.copy(50, 0, 50, 100, 0, 0, 50, 100);
image(img, 0, 0);
```

Description

Copies a region of pixels from one image into another. If the source and destination regions aren't the same size, it will automatically resize source pixels to fit the specified target region. No alpha information is used in the process, however if the source image has an alpha channel set, it will be copied as well.

The **imageMode()** function changes the way the parameters work. For example, a call to **imageMode(CORNERS)** will change the width and height parameters to define the x and y values of the opposite corner of the image.

Syntax

```
img.copy(sx, sy, swidth, sheight, dx, dy, dwidth, dheight);
img.copy(srcImg, sx, sy, swidth,  sheight, dx, dy, dwidth, dheight);
```

Figure 2-7. The documentation for the PImage copy from Processing.org

So, what does this tell you? It shows you an example of using the PImage copy, and it shows you what the signature of the function looks like. Here it is in Processing:

```
img.copy(sx, sy, swidth, sheight, dx, dy, dwidth, dheight);
img.copy(srcImg, sx, sy, swidth, sheight, dx, dy, dwidth, dheight);
```

This tells you that the PImage has a function called copy that has two different ways it can be called. You'll need to read the rest of the PImage documentation on the Processing website to know what parameters you need to pass to each of these functions, but the names of those parameters should be somewhat helpful. Looking at the documentation shown in Figure 2-7 for the first function, you see several parameters with an s appended to them and several with a d appended to them. The s stands for source, and the d stands for destination, making them a little easier to understand. The function copies the pixels from one PImage object to another PImage object. Now that you have a rough idea of what sorts of parameters these functions take, you can call them on two different PImage objects:

```
PImage  p1 = loadImage("baby.jpg");
image(p1, 0, 0);
PImage p2 = new PImage(400, 400);
p2.copy(p1, 0, 0, p1.width, p1.height, 10, 10, p1.width * 2, p1.height * 2);
image(p2, 100, 100);
```

Take a moment to read the fourth line carefully. You're calling a function on the p2 PImage, on p2, and passing the height and width of p1 as parameters. Let's look at that function signature one more time:

```
img.copy(srcImg, sx, sy, swidth, sheight, dx, dy, dwidth, dheight);
```

srcImg is the PImage you want to copy pixels from, and sx and sy are where you want to copy the image from. dx and dy is where the image will be copied to and how wide and high the image will appear. You can use the properties of a PImage to pass values to a function of another PImage. You'll see this quite frequently, where properties of objects are stored, passed around, and accessed later.

Let's imagine for a moment that you've made some object that has a whole bunch of PImage objects inside it. You'll call it a PhotoAlbum. That object has a function on it called getNewestPicture. In using your PhotoAlbum class, you might have something that looks like this in Processing:

```
PhotoAlbum newAlbum;
newAlbum.getNewestPicture().filter(BLUR);
```

So, even though we haven't covered the filter function, I'd bet you can guess what doing filter(BLUR) will do. The interesting thing to note here is how a value from the getNewestPicture function has another function called on it right away. You could just as easily say this in Processing:

```
PhotoAlbum newAlbum;
PImage img = newAlbum.getNewestPicture();
img.filter(BLUR);
```

The two code snippets here will do exactly the same thing, but they'll do them in slightly different ways. It isn't important which way you do these things; it's only important that you're able to follow both the ways that they're done because you'll encounter them both.

Scope

Scope is one of those nefarious things that can create some of the most mysterious bugs known. Luckily for you, it's quite easy to keep it straight because there's a simple rule that will keep you out of trouble.

Let's first look at the following little Processing code snippet:

```
void setup() {

    if(true)
    {
        int i = 0;
    }
    i+=1;
    print(i);

}
```

OK, that was rather pointless, but if you try to compile it, you'll see something interesting pop up above the Processing console window (we'll cover how to do that in Chapter 3). You'll see this:

```
Cannot find anything named "i"
```

What was that? The scope of something means the place in your program where things are accessible. Figure 2-8 shows where in the function that each variable is accessible.

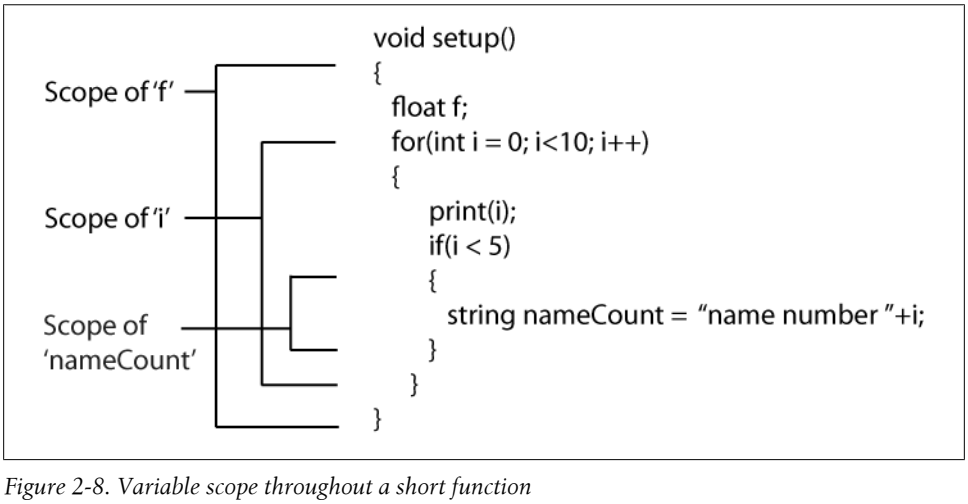


Figure 2-8. Variable scope throughout a short function

The general rule of variable scope is this: *all variables exist within the brackets that surround them*. If you have a variable within a function, you have this:

```
void someFunction() {
    int anInt;
}
```

The variable `anInt` is going to exist in that function, until the closing bracket for the function, which is the `}` at the end of the function. If a variable is defined within an `if` statement, then you have this:

```
if(something) {
    int anInt;
}
anInt = 5; // can't do this! anInt doesn't exist out here
```

In this case, `anInt` exists only inside that `if` statement, not anywhere else. Why is this? The short answer is that it makes it easier for your program to figure out what things are being used and what things aren't so that it can run more efficiently. Just remember the rule of the brackets: if there's a bracket before a variable, it exists only until another bracket closes it.

Some variables are available everywhere in a program, so they never go out of scope. These kinds of variables are called *global variables*. In a Processing or Arduino application, you can create a global variable by declaring it outside of the `setup()` method of your application. In an oF application, it's a little different to do. You'll learn more about variables and global variables in each chapter relating to the specific language that you'll be looking at, because each one handles them in a slightly different fashion.

Review

Variables represent data stored in the computer's memory for your program to access.

Variables have a type that describes what kind of information can be stored in them and specified when the variable is declared, for instance:

```
int weight = 150;
```

or:

```
float f = 1.12;
```

Variables are *declared* and then *initialized*. These can be done in separate steps:

```
int apples;
apples = 5;
```

or at the same time:

```
int apples = 5;
```

Variables can be cast to another variable type by using the name of the new type and parentheses to cast the variable:

```
float f = 98.9;
int myInt = int(f); //myInt is now the float f converted into an int
```

All variable types can be stored in arrays. These are declared with the length of the array and the type of variable stored in the array:

```
int arr[6]; // creates an array that can hold up to 6 integers
```

Accessing an array at any point within it is done by using the index, which begins at 0 and extends to the length of the array minus 1:

```
arr[0] = 1;
arr[1] = 2;
print(arr[0]); // will print '1', the first element in the array
```

Arrays can be filled when they are declared, as shown here:

```
int arr[3] = {1, 2, 3};
```

or the elements in the array can be set one at a time.

Control structures allow you to control the flow your program. The most commonly used control structure are branching structures:

```
if(somethingTrueOrFalse){
    //if the evaluated statement is true
} else {
    //if the evaluated statement is false
}
```

and loops:

```
for(int i = 0; i<someNumber; i++) {
    //do something someNumber of times
}
```

Functions are declared with a name, a return type, and any parameter that the function requires:

```
returnType functionName(parameterType parameterName) {
//what the function does
}
```

Using a function is usually referred to as *calling* the function. This is separate from *declaring* the function:

```
splitString(0, 10, 'hello there'); // this calls the function declared above
```

Functions can be overloaded, which means that multiple method declarations can be made with the same name and different parameters. When you call one of the functions, the compiler will determine which version of the function you meant to call.

CHAPTER 3
Processing

Processing was one of the first open source projects that was specifically designed for simplifying the practice of creating interactive graphical applications so that nonprogrammers could easily create artworks. Artists and designers developed Processing as an alternative to similar proprietary tools. It's completely open source and free to download, use, and modify as you see fit. Casey Reas and Ben Fry started the project at MIT under the tutelage of John Maeda, but a group of developers maintain it by making frequent updates to the core application. The Processing project includes an integrated development environment (IDE) that can be used to develop applications, a programming language specifically designed to simplify programming for visual design, and tools to publish your applications to the web or to the desktop.

One of the reasons that we brought up the Java Virtual Machine (JVM) in the introduction is that Processing is a Java application; that is, it runs in the JVM. A Processing application that you as the artist create is also a Java application that requires the JVM to run. You can take two different approaches to running Processing applications. The first is running your Processing application on the Web, which means putting your application in a Hypertext Markup Language (HTML) page where a browser like Firefox, Internet Explorer, or Safari will encounter your application and start the JVM to run your application. The second is running your Processing application on the desktop, using the JVM that is installed on the user's machine.

What can you do with Processing? Because Processing is built in Java and runs using Java, it can do almost anything Java will do, and although Java can't quite do *everything* you'll see in computational art and design, it certainly comes close. With it you can do everything from reading and writing data on the Internet; working with images, videos, and sound; drawing two- and three-dimensionally; creating artificial intelligence; simulating physics; and much more. If you can do it, there's a very good chance you can do it using Processing.

In this chapter, we'll cover the basics of getting started with Processing, including the following: downloading and installing the environment, writing some simple programs, using external libraries that let you extend what Processing is capable of, and running applications on the Internet and on the desktop.

Downloading and Installing Processing

The first step to installing Processing is to head to *http://processing.org* and look for the Download section. You'll see four downloads, one for each major operating system (Windows, Mac OS X, and Linux) and one for Windows users who don't want to install another instance of the JVM.

Processing includes an instance of the JVM, so you don't need to worry about configuring your computer. In other words, everything Processing needs is right in the initial download. Download the appropriate archive for your operating system, and move the files in the archive to somewhere you normally store applications. On a Windows computer, this might be a location like *C:\Program Files\Processing*. On a Mac, it might be something like */Applications/Processing/*. On a Linux machine, it might be somewhere like *~/Applications/*.

Once you've downloaded and uncompressed the application, you're done—you've installed Processing. Go ahead and start the application, and then type the following simple program in the window:

```
fill(0, 0, 255);
rect(0, 0, 100, 100);
print(" I'm working ");
```

Click the Run button (shown in the next section in Figure 3-1) to execute the program. This makes sure you have everything set up correctly. If this opens a new small window with a blue square in it and prints "I'm working" into the Console window, everything is good to go.

Exploring the Processing IDE

Before we talk about the IDE in detail, it's important to understand the setup of the Processing environment. While the application is stored in your *C:\Program Files* directory (for Windows users) or in your */Applications/* directory (for Mac or Linux users), the sketches that you create in Processing are usually stored in your home documents folder. This means that for a Windows user the default location for all your different sketches might be *C:\Documents And Settings\User\My Documents\processing*, for a Mac user it might be */Users/user/Documents/Processing/*, and for a Linux user it might be *home/user/processing/*. You can always set this to a different location by opening Processing, opening the Preferences dialog and changing the "Sketchbook location" option to be wherever you'd like. The important thing is that you remember where it is.

Each Processing sketch you create and save will have its own folder. When you save a Processing sketch as *image_fun*, for example, a folder called *image_fun* will be created in the Processing project directory. If you're planning on working with images, MP3 files, videos, or other external data, you'll want to create a folder called *data* and store it in the project folder for the project that will use it. We'll discuss this in greater detail in the section "Importing Libraries" on page 77. You can always view the folder for a particular sketch by pressing Ctrl+K (⌘-K on a Mac) when the sketch is open in the IDE.

As you can see in Figure 3-1, the Processing IDE has four main areas: the controls, the Code window, the Messages window, and the Console window.

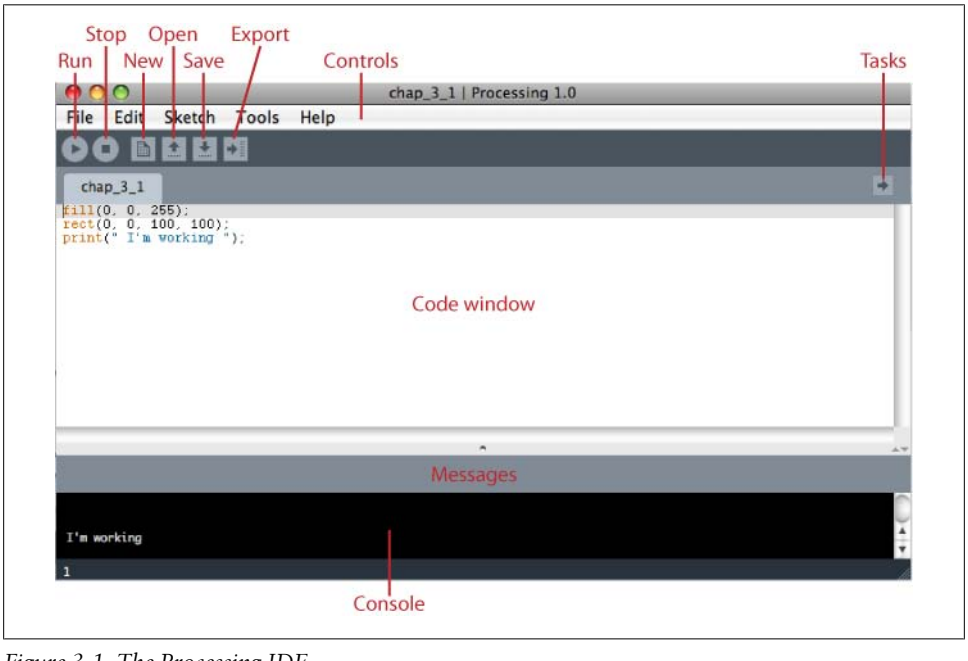


Figure 3-1. The Processing IDE

The controls area is where all the buttons to run, save, stop, open, export, and create new applications appear. After you've entered any code in the Code window and you're ready to see what it looks like when it runs, click the Run button, which opens a new window and runs your application in that window. To stop your application, either close the window or click the Stop button. The Export button creates a folder called *applet* within your project folder and saves all the necessary files there. You can find more information on this in the section "Exporting Processing Applications" on page 86 later in this chapter.

The Code window is where you enter all the code for the application. It supports a few of the common text editor functions, like Ctrl-F (⌘-F on a Mac) for Find and Ctrl-Z (⌘-Z on a Mac) for Undo.

The Messages window is where any system messages that might appear will be shown. It isn't widely used.

The Console window is where any trace statements or errors that occur while your application is running will be shown. Trace statements and printing messages to the consol will be discussed a little later on in this chapter.

The Basics of a Processing Application

A Processing application has two fundamental methods. These methods are the instructions for your application at two core moments. The first, `setup()`, is invoked when the application first starts. The second, `draw()`, is invoked over and over again from right after startup until the application closes. Let's take a look at the first of these and dissect what a method really is.

The setup() Method

Any instructions put in the `setup()` method run when the application first starts. Conceptually, you can think of this as any kind of preparation that you might be used to in daily life. Getting ready to go for a run: stretch. Getting ready to fly overseas: make sure you have a passport. Getting ready to cook dinner: check to see everything you need is in the refrigerator. In terms of our Processing application, the `setup()` method makes sure any information you want to use in the rest of the application is prepared properly.

Here's an example:

```
void setup(){
    size(200, 200);
    frameRate(30);
    print(" all done setting up");
}
```

So, what's going on in this code snippet? Well, first you have the return type of this method, and then you have its name. Taken together these are called the *declaration* of this method. If you don't remember what a *method declaration* is, review Chapter 2. The rest of the code snippet is methods that have already been defined in the core Processing language. That is, you don't need to redefine them, you simply need to pass the correct values to them.

The size() method

On the second line of the previous code snippet is the `size()` method. It has two parameters: how wide the application is supposed to be and how tall it's supposed to be, both in pixels. This idea of measuring things in pixels is very popular; you'll see it in any graphics program, photo or video editing program, and in the core code that runs all of these programs. Get inches or centimeters out of your head; in this world,

everything is measured in pixels. This `size()`method makes the window that our application is going to run in. Go ahead and write the previous code in the Processing IDE, and click Run. Then change the size, and click Run again. You'll see that the `size()` method takes two parameters:

```
void size(width, height)
```

Frequently when programmers talk about a method, they refer to the *signature* of a method. This means what parameters it takes and what values it returns. If this doesn't sound familiar, review Chapter 2, specifically, the section on methods. In the `setup()` example, the `size()` method takes two integer (numbers without any decimal values) values and returns `void`, that is, nothing.

The frameRate() method

The next method in the previous code snippet is the `frameRate()` method, which determines how many frames per second your application is going to attempt to display. Processing will never run faster than the frame rate you set, but it might run more slowly if you have too many things going at once or some parts of your code are inefficient. More frames per second means your animations will run faster and everything will happen more rapidly; however, it also can mean that if you're trying to do something too data intensive, Processing will lag when trying keep up with the frame rate you set. And how is this done? Well, the number you pass to the `frameRate()` method is the number of times per second that the `draw()` method is called.

The print() method

The `print()` method puts a message from the application to the Console window in the Processing IDE. You can print all sorts of things: messages, numbers, the value of certain kinds of objects, and so forth. These provide not only an easy way to get some feedback from your program but also a simple way to do some debugging when things aren't working the way you'd like them to work. Expecting something to have a value of 10? Print it to see what it really is. All the print messages appear in the Console window at the bottom of the Processing IDE, as shown in Figure 3-2.

The draw() Method

The `draw()` method is where the drawing of the application happens, but it can be much more than that. The `draw()` method is the heartbeat of your application; any behavior defined in this method will be called at the number of times per second specified as the frame rate of your application.

This is a simple example of a `draw()` method:

```
void draw() {
println("hi");
}
```

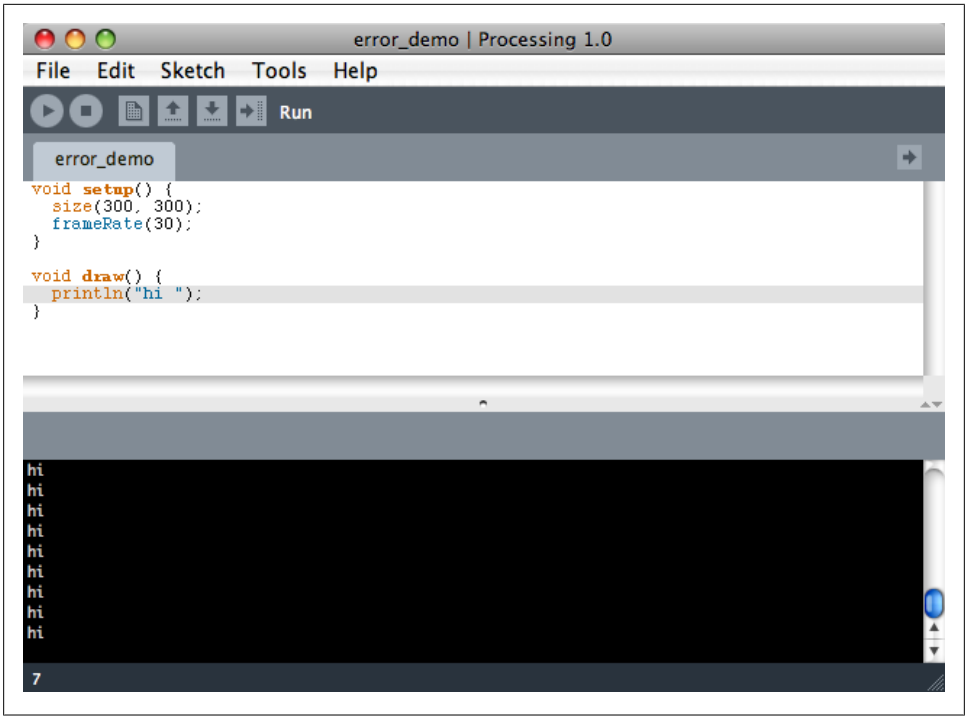


Figure 3-2. The Console window in an application

Assuming that the frame rate of your application is 30 times a second, the message `"hi"` will print to the Console window of the Processing IDE 30 times a second. That's not very exciting, is it? But it demonstrates what the `draw()` method is: the definition of the behavior of any processing application at a regular interval determined by the frame rate, after the application runs the `setup()` method.

Now, let's look at a slightly more interesting example and dissect it:

```
int x = 0;

void setup() {
    size(300, 300);
}

void draw() {
    // make x a little bit bigger
    x += 2;
    // draw a circle using x as the height and width of the circle
    ellipse(150, 150, x, x);
    // if x is too big, we can't see it in our window, so put it back
    // to 0 and start over
    if(x > 300) {
        x = 0;
    }
}
```

First things first—you're making an `int` variable, `x`, to store a value:

```
int x = 0;
```

Since `x` isn't inside a method, it's going to exist throughout the entire application. That is, when you set it to 20 in the `draw()` method, then the next time the `draw()` method is called the value of `x` is still going to be 20. This is important because it lets you gradually animate the value of `x`.

This refers to the idea of *scope*; if that concept isn't ringing any bells for you, review Chapter 2.

To set up the application using the `setup()` method, simply set the size of the window so that it's big enough. Nothing much is too interesting there, so you can skip right to the `draw()` method:

```
void draw() {
```

Each time you call `draw()`, you're going to make this number bigger by 2. You could also write `x = x+2;`, but the following is simpler and does the same thing:

```
    x += 2;
```

Now that you've made `x` a little bit bigger, you can use it to draw a circle into the window:

```
ellipse(150, 150, x, x);
```

Look ahead in this chapter to the section "The Basics of Drawing with Processing" on page 60 for more information about the `ellipse()` method.

If the value of `x` is too high, the circle will be drawn too large for it to show up correctly in your window (300 pixels); you'll want to reset `x` to 0 so that the circles placed in the window begin growing again in size:

```
    if(x > 300) {
        x = 0;
    }
}
```

In Figure 3-3, you can see the animation about halfway through its cycle of incrementing the `x` value and drawing gradually larger and larger circles.

The `draw()` method is important because the Processing application uses it to set up a lot of the interaction with the application. For instance, the `mousePressed()` method and the `mouseMove()` methods that are discussed in the section "Capturing Simple User Interaction" on page 67 will not work without a `draw()` method being defined. You

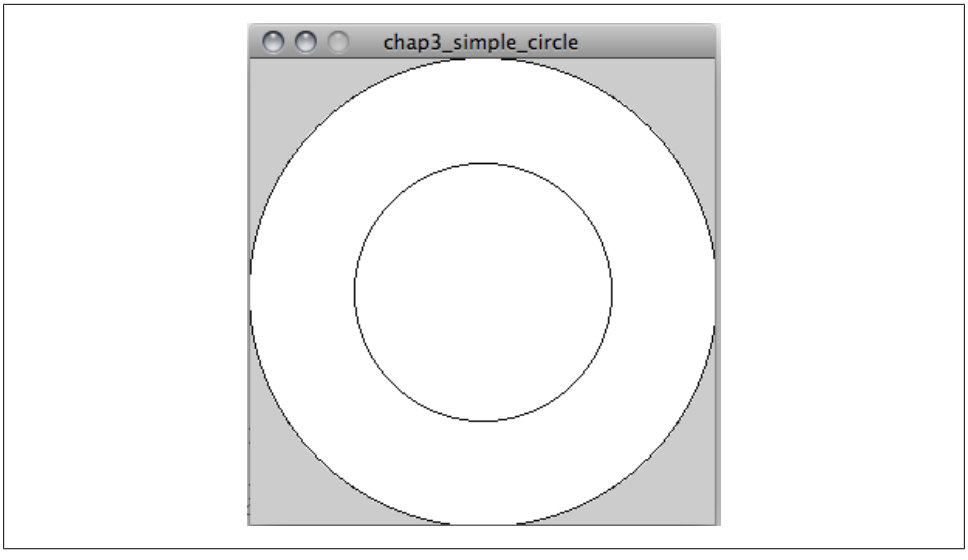


Figure 3-3. The demo application drawing circles

can imagine that the `draw()` method tells the application that you want to listen to whatever happens with the application as each frame is drawn. Even if nothing is between the brackets of the `draw()` method, generally you should always define a `draw()` method.

The Basics of Drawing with Processing

Because Processing is a tool for artists, one of the most important tasks it helps you do easily is drawing. You'll find much more information on drawing using vectors and bitmaps in Chapters 9 and 10, and you'll find information about OpenGL, some of the basics of 3D, and how to create complex drawing systems in Chapter 13. For right now, you'll learn how to draw simple shapes, draw lines, and create colors to fill in those shapes and lines.

The rect(), ellipse(), and line() Methods

Each of the three simplest drawing methods, `rect()`, `ellipse()`, and `line()`, lets you draw shapes in the display window of your application. The `rect()` method draws a rectangle in the display window and uses the following syntax:

```
rect(x, y, width, height)
```

Each of the values for the `rect()` method can be either `int` or `float`. The `x` and `y` position variables passed to the rectangle determine the location of the upper-left corner of the rectangle. This keeps in line with the general rule in computer graphics of referring to

the upper-left corner as 0, 0 (to anyone with a math background this is a little odd, since Cartesian coordinates have the `y` values flipped).

The `ellipse()` method is quite similar to `rect()`:

```
ellipse(x, y, width, height)
```

Each of the values passed to the `ellipse()` method can be int or float. If you make the `height` and `width` of the ellipse the same, you get a circle, of course, so you don't need a `circle()` method.

Finally, the `line()` method uses the following syntax:

```
line(x1, y1, x2, y2)
```

The `x1` and `y1` values define where the line is going to start, and the `x2` and `y2` values define the end of the line. Here's a simple example incorporating each of the three methods:

```
void setup() {
    size(300, 300);
}

void draw() {
    rect(100, 100, 50, 50);
    ellipse(200, 200, 40, 40);
    line(0, 0, 300, 300);
}
```

Notice in Figure 3-4 that because the line is drawn after the rectangle and ellipse, it's on top of both of them. Each drawing command draws on top of what is already drawn in the display window.

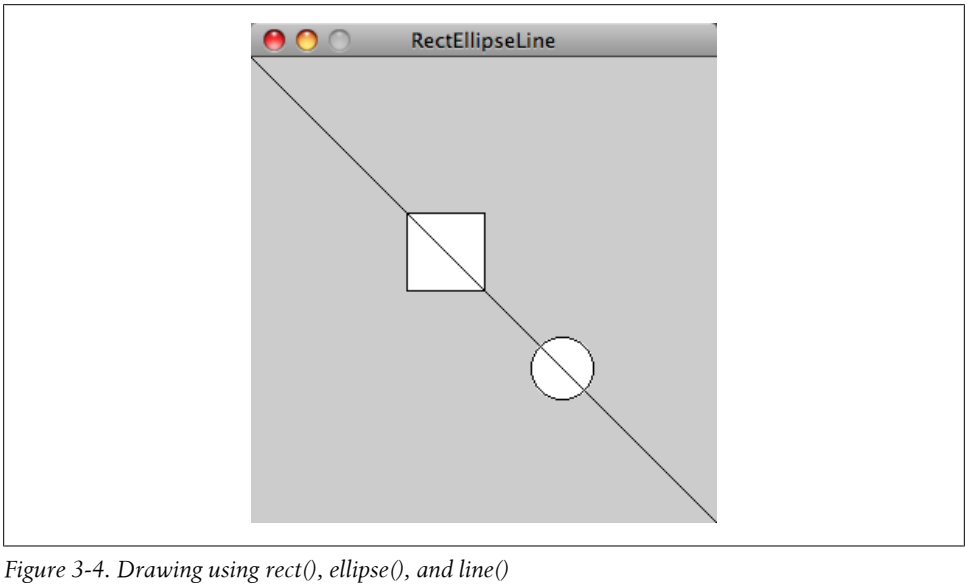


Figure 3-4. Drawing using rect(), ellipse(), and line()

Now, let's take a brief interlude and learn about two ways to represent color.

RGB Versus Hexadecimal

There are two ways of representing numbers in Processing in particular and computing in general: RGB and hexadecimal. Both are similar in that they revolve around the use of red, green, and blue values. This is because the pixels in computer monitors are colored using mixtures of these values to determine what color a pixel should be. In RGB, you use three numbers between 0 and 255 and pass them to the `color()` method. For example, the following defines white:

```
int fullred = 255;
int fullgreen = 255;
int fullblue = 255;
color(fullred, fullgreen, fullblue);
```

Conversely, the following defines black:

```
int fullred = 0;
int fullgreen = 0;
int fullblue = 0;
color(fullred, fullgreen, fullblue);
```

All red and no green or blue is red, all green and blue and no red is yellow, and anytime all the numbers are the same the color is a gray (or white/black).

RGB is an easy way of thinking about color perhaps, but another way is commonly used on the Internet for HTML pages and has grown in popularity in other types of coding as well: hexadecimal numbers.

What's a hexadecimal number? Well, a decimal number counts from 1 to 9 and then starts incrementing the number behind the 1. So, once you get to 9, you go to 10, and once you get to 19, you go to 20. This is the decimal system. The hexadecimal system is based on the number 16 and is very efficient for computers because computers like to process things in numbers that are cleanly divisible by 4 and 16, which 10 is not. A hexadecimal counting looks like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. So, that means that to write 13 in hexadecimal, you write D. To write 24, you write 18. To write 100, you write 64. The important concept is that to write 255, you write FF, which means representing numbers takes fewer characters to represent ("255" being three characters while "FF" is only two), which is part of where the efficiency comes from. Less characters means more efficient processing. Hexadecimal numbers for colors are written all together like this:

```
0xFF00FF
```

or:

```
#FF00FF
```

That actually says red = 255, green = 0, blue = 255, just in a more efficient way. The `0x` and `#` prefixes tell the Processing compiler you're sending a hexadecimal number,

not just mistyping a number. You'll need to use those if you want to use hexadecimal numbers.

Here are some other colors:

- `000000` = black
- `FFFFFF` = white
- `00FFFF` = yellow

We'll cover one final concept of hexadecimal numbers: the alpha value. The *alpha value* controls the transparency of a color and is indicated in hexadecimal by putting two more numbers at the beginning of the number to indicate, on a scale of 0–255 (which is really 0–FF), how transparent the color should be.

`0x800000FF` is an entirely "blue" blue (that is, all blue and nothing else) that's 50% transparent. You can break it down into the following:

- Alpha = 80 (that is, 50%)
- Red = 00 (that is, 0%)
- Green = 00 (that is, 0%)
- Blue = FF (that is, 100%)

`0xFFFF0000` is an entirely "red" red, that is, 0% transparent or totally opaque with full red values and no green or blue to mix with it, and `0x00FFFF00` is a magenta that is invisible. This is why computer color is sometimes referred to as ARGB, which stands for "alpha, red, green, blue."

Now that you know the basics of colors, you can move on to using more interesting colors than white and black with shapes.

The fill() Method

The `fill()` method is named as such because it determines what goes inside any shapes that have empty space in them. Processing considers two-dimensional shapes like rectangles and circles to be a group of points that have an empty space that will be colored in by a fill color. Essentially, imagine that setting the fill sets how any empty shapes will be *filled in*. Without a fill, Processing won't know what color you want your shapes to be filled in with. The `fill()` method lets you define the color that will be used to fill in those shapes.

Before you really dive into the `fill()` method, it's important to remember that methods can be *overloaded*; that is, they can have multiple signatures. If this doesn't seem familiar to you or you'd like a refresher on this, review Chapter 2.

The `fill()` method has many different ways of letting you set the color that will be used to fill shapes:

`fill(int gray)`
: This is an `int` between 0 (black) and 255 (white).

`fill(int gray, int alpha)`
: This is an `int` between 0 (black) and 255 (white) and a second number for the alpha of the fill between 0 (transparent) and 255 (fully opaque).

`fill(int value1, int value2, int value3)`
: Each of these values is a number between 0 and 255 for each of the following colors: red, green, and blue. For instance, `fill(255, 0, 0)` is bright red, `fill(255, 255, 255)` is white, `fill(255, 255, 0)` is yellow, and `fill(0, 0, 0)` is black.

`fill int(value1, int value2, int value3, int alpha)`
: This is the same as the previous example, but the additional `alpha` parameter lets you set how transparent the fill will be.

`fill(color color)`
: The `fill()` method can also be passed a variable of type `color`, as shown here:

```
void draw(){
    color c = color(200, 0, 0);
    fill(c);
    rect(0, 0, 100, 100);
}
```

`fill(color color, int alpha)`
: When passing a variable of type `color`, you can specify an `alpha` value:

```
color c = color(200, 0, 0);
fill(c, 150);
```

`fill(int hex)`
: This is a fill using a hexadecimal value for the color, which can be represented by using the `0x` prefix or the `#` prefix in the hexadecimal code. However, Processing expects that if you use `0x`, you'll provide an alpha value in the hexadecimal code. This means if you don't want to pass a separate `alpha` value, use the `0x` prefix. If you do want to pass a separate alpha value, use the `#` prefix.

`fill(int hex, int alpha)`
: This method uses a hexadecimal number but should be used with the `#` symbol in front of the number, which can be a six-digit number only. The alpha value can be anywhere from 0 to 255.

The `fill()` method sets the color that Processing will use to draw all shapes in the application until it's changed. For example, if you set the fill to blue like so:

```
fill(0, 0, 255);
```

and then draw four rectangles, they will be filled with a blue color.

The background() Method

To set the color of the background window that your application is running in, use the `background()` method. The `background()` method uses the same overloaded methods as the `fill()` method, with the method name changed to `background()`, of course. The `background()` method also completely covers up everything that was drawn in the canvas of your application. When you want to erase everything that has been drawn, you call the `background()` method. This is really helpful in animation because it lets you easily draw something again and again while changing its position or size without having all the previous artifacts hanging around.

The line() Method

You can use the `line()` method in two ways. The first is for any two-dimensional lines:

```
line(x1, y1, x2, y2)
```

The second is for any three-dimensional lines:

```
line(x1, y1, z1, x2, y2, z2)
```

Now, of course, a three-dimensional line will simply appear like a two-dimensional line until you begin moving objects around in three-dimensional space; we'll cover that in Chapter 13. You need to take smaller steps first. The `line()` method simply draws a line from the point described in the first two (or three parameters) to the point. You can set the color of the line using the `stroke()` method and thickness of the line using the `strokeWeight()` method.

The stroke() and strokeWeight() Methods

The `stroke()` method uses the same parameters as the `fill()` method, letting you set the color of the line either by using up to four values from 0 to 255 or by using hexadecimal numbers. The `strokeWeight()` method simply sets the width of the lines in pixels. Let's take a look:

```
void draw(){
    stroke(0xFFCCFF00); // here we set the color to yellow
    strokeWeight(5); // set the stroke weight to 5
    // draw the lines
    line(0, 100, 600, 400);
    line(600, 400, 300, 0);
    line(300, 0, 0, 100);
}
```

This draws a rectangular shape with yellow lines, 5 pixels thick.

The curve() Method

The `curve()` method draws a curved line and is similar to the `line()` method except it requires you to specify anchors that the line will curve toward.

You can use the `curve()` method in two dimensions:

```
curve(x1, y1, x2, y2, x3, y3, x4, y4);
```

or in three dimensions:

```
curve(x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4);
```

All the variables here can be either floating-point numbers or integers. The `x1`, `y1`, and `z1` variables are the coordinates for the first anchor point, that is, the first point that the curve will be bent toward. The `x2`, `y2`, and `z2` variables are the coordinates for the first point, that is, where the line begins, in either two- or three-dimensional space. The `x3`, `y3`, and `z3` parameters are the coordinates for the second point, that is, where the line ends, and the `x4`, `y4`, and `z4` parameters are the coordinates for the second anchor, that is, the second point that the curve will be bent toward:

```
void setup(){
    size(400, 400);
}

void draw(){
    background(255);
    fill(0);
    int xVal = mouseX*3-100;
    int yVal = mouseY*3-100;
    curve(xVal, yVal, 100, 100, 100, 300, xVal, yVal);
    curve(xVal, yVal, 100, 300, 300, 300, xVal, yVal);
    curve(xVal, yVal, 300, 300, 300, 100, xVal, yVal);
    curve(xVal, yVal, 300, 100, 100, 100, xVal, yVal);
}
```

When this code snippet runs, you'll have something that looks like Figure 3-5.

The vertex() and curveVertex() Methods

What is a vertex? A *vertex* is a point where two lines in a shape meet, for example, at the point of a triangle or point of a star. In Processing, you can use any three or more vertices to create a shape with or without a fill by calling the `beginShape()` method, creating the vertices, and then calling the `endShape()` method. The `beginShape()` and `endShape()` methods tell the Processing environment you intend to define points that should be joined by lines to create a shape. Three vertices create a triangle, four a quadrilateral, and so on. The Processing environment will create a shape from the vertices that you've selected, creating lines between those vertices, and filling the space within those lines with the fill color you've defined. The following example creates three vertices: one at 0, 0, which is the upper-left corner of the window; one at 400,

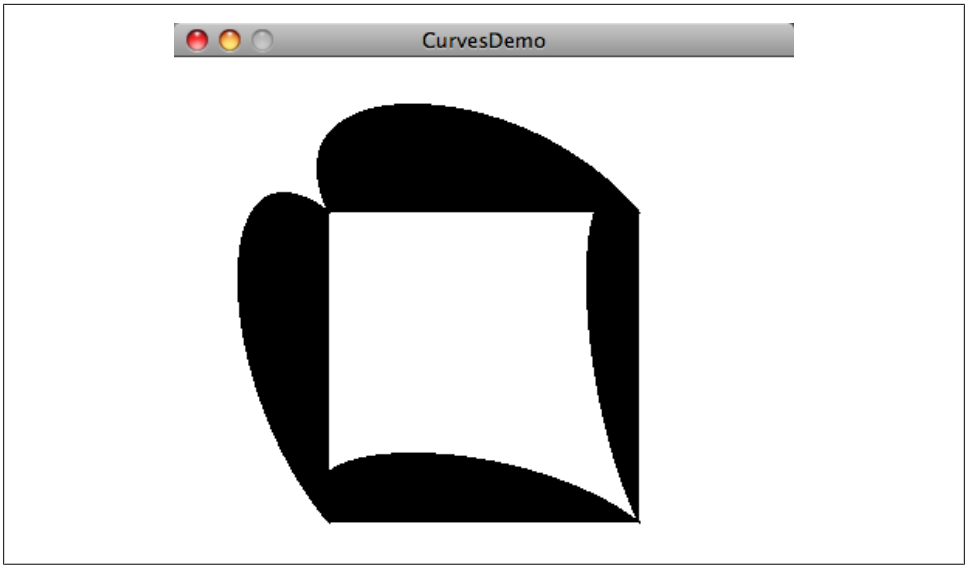


Figure 3-5. Running the curve drawing sample

400, which is the lower-right corner of the window; and one at the user's mouse position. The snippet results in the shape shown in Figure 3-6.

```
void setup() {
    size(400, 400);
}

void draw(){
    background(255);
    fill(0);
    beginShape();
    vertex(0, 0);
    vertex(400, 400);
    vertex(mouseX, mouseY);
    endShape();
}
```

Capturing Simple User Interaction

To begin at the beginning, you'll see how Processing handles the two most common modes of user interaction: the mouse and the keyboard. To capture interactions with these two tools, what you really need to know is when the mouse is moving, when the mouse button has been pressed, when the user is dragging (that is, holding the mouse button down and moving the mouse), whether a key has been pressed, and what key has been pressed. All these methods and variables already exist in the Processing application. All you need to do is invoke them, that is, to tell the Processing environment

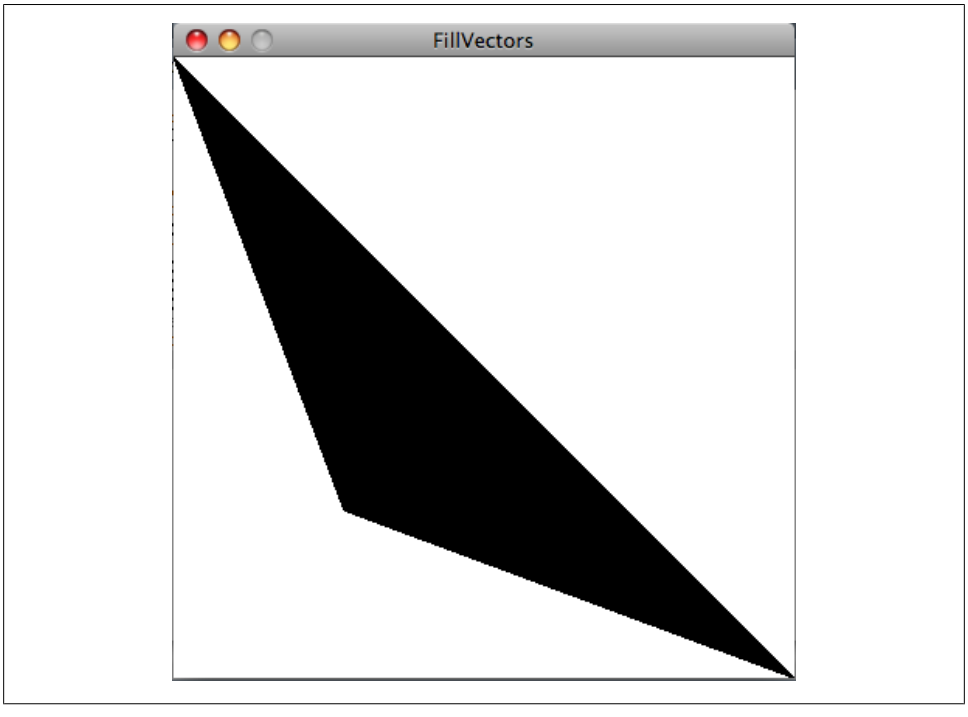


Figure 3-6. Creating vertices and a fill

that you want to do something when the method is invoked or access the variable either when one of the methods is invoked or in the `draw()` method of your application.

The mouseX and mouseY Variables

The `mouseX` and `mouseY` variables contain the position of the mouse in x and y coordinates. This is done in pixels, so if the mouse is in the furthest upper-left corner of the window, `mouseX` is 0 and `mouseY` is 0. If the mouse pointer is in the lower-right corner of a window that is 300 × 300 pixels, then `mouseX` is 300 and `mouseY` is 300. You can determine the position of the mouse at any time by using these variables. In the following code snippet, you can determine the position of the mouse whenever the `draw()` method is called:

```
PFont arial;

void setup() {
    // make the size of our window
    size(300, 300);
    // load the font from the system
    arial = createFont("Arial", 32);
    // set up font so our application is using the font whenever
    // we call the text method to write things into the window
```

```
    textFont(arial, 15);
}

void draw() {
    // this makes the background black, overwriting anything there
    // we're doing this because we want to make sure we don't end up
    // with every set of numbers on the screen at the same time.
    background(0);
    // here's where we really do the work, putting the mouse position
    // in the window at the location where the mouse is currently
    text(" position is "+mouseX+" "+mouseY, mouseX, mouseY);
}
```

As you can see from the code comments, this example shows a little more than just using the mouse position; instead, it uses the `text()` method to put the position of the mouse on the screen in text at the position of the mouse itself. You do this by using the `text()` method, which takes three parameters:

```
void text(string message, xPosition, yPosition);
```

The first parameter is the message, and the second and third parameters can be floats or ints and are for positioning the location of the text. In the previous example, you put the text where the user's mouse is using the `mouseX` and `mouseY` variables.

The mousePressed() Method

Processing applications have a `mousePressed()` method that is called whenever the user clicks the left mouse button. This *callback method* is called whenever the application has focus and the user presses the mouse button. As mentioned in the previous section, without a `draw()` method in your application, the `mousePressed()` method will not work. Let's look at a quick example that uses a few drawing concepts and also shows you how to use the `mousePressed()` method:

```
int alphaValue = 0;

void setup() {
    size(350, 300);
    background(0xFFFFFFFF);
}

void draw() {
    background(0xFFFFFFFF);
    fill(255, 0, 0, alphaValue);
    rect(100, 100, 100, 100);
}

void mousePressed() {
    print(mouseX + "\n");
    alphaValue = mouseX;
}
```

The `mousePressed()` method here sets the `alpha` value of the fill that will be used when drawing the rectangle to be the position of the user's mouse. The application will draw using the `alphaValue` variable that is set each time the mouse is clicked.

The mouseReleased() and mouseDragged() Methods

The `mouseReleased()` method notifies you when the user has released the mouse button. It functions similarly to the `mousePressed()` method and lets you create actions that you would like to have invoked when the mouse button is released. The `mouseDragged()` method works much the same way, but is generally (though certainly not always) used to determine whether the user is dragging the mouse by holding the button down and moving the mouse around. Many times when looking at mouse-driven applications you'll see the `mouseDragged()` method being used to set a boolean variable that indicates whether the user is dragging. In the following example, though, you'll put some actual drawing logic in the `mouseDragged()` method:

```
int lastX = 0;
int lastY = 0;

void setup() {
     size(400, 400);
}

void draw() {
    lastX = mouseX;
    lastY = mouseY;
}

void mouseDragged() {
    line(lastX, lastY, mouseX, mouseY);
}
```

Now you have a simple drawing application. You simply store the last position of the user's mouse and then, when the mouse is being dragged, draw a line from the last position to the current position. Try running this code and then comment out the lines in the `draw()` method that set the `lastX` and `lastY` variables to see how the application changes.

An, easier way to do this is to use the `pmouseX` and `pmouseY` variables. These represent the x and y mouse positions in the previous frame of the application. To use these variables, you would use the following code:

```
void mouseDragged() {
    line(pmouseX, pmouseY, mouseX, mouseY);
}
```

To extend this a little bit, you'll next create a slightly more complex drawing application that allows someone to click a location and have the Processing application include that location in the shape. This code sample has three distinct pieces that you'll examine one by one. The first you'll look at is a class that is defined in this example.

First, you have the declaration of the name of the class, `Point`, and the bracket that indicates you're about to define all the methods and types of that class:

```
class Point{
```

Next, you have the two values that are going to store the `x` and `y` values of `Point`:

```
float x;
    float y;
```

Here is the constructor for this class:

```
Point(float _x, float _y){
    x = _x;
    y = _y;
}
```

The constructor takes two values, `_x` and `_y`, and sets the `x` and `y` values of `Point` to the two values passed into the constructor. What this `Point` class now lets you do is store the location where someone has clicked.

Now that you've defined the `Point` class, you can create an array of those `Point` objects. Since you want to allow for shapes with six vertices, in order to store them, you'll need an array of `Point` instances with six elements:

```
Point[] pts = new Point[6];
```

Now all that's left to do is set the `mousePressed()` method so that it stores the `mouseX` and `mouseY` positions using the `Point` class and then draws all those `Point` instances into the window. We'll break down each step of the code because this is pretty complex:

```
Point[] pts = new Point[6];
int count = 0;
void setup(){
    size(500, 500);
}
```

In the `draw()` method, the background is drawn filled in, the fill for any drawing is set using the `fill()` method, and a `for` loop is used to draw a vertex for each of the points:

```
void draw(){
    background(255);
    fill(0);
    beginShape();
    for(int i = 0; i<pts.length; i++){
```

Just to avoid any errors that may come from the `Point` object not being instantiated, you check to make sure that the `Point` in the current position in the array isn't `null`:

```
if(pts[i] != null) {
```

If it isn't `null`, you'll use it to create a `vertex` using `Point` from the `pts` array:

```
vertex(pts[i].x, pts[i].y);
        }
    }
```

Now you're done drawing the shape:

```
endShape();
}
```

When the user presses the mouse, you want to store their mouse position for use in the `pts` array. You do this by creating a new `Point` object, passing it the current `mouseX` and `mouseY` positions, and storing that in the `pts` array at the `count` position:

```
void mousePressed(){
    if(count > 5){
        count = 0;
    }
    Point newPoint = new Point(mouseX, mouseY);
    pts[count] = newPoint;
    count++;
}
```

Finally, you have the declaration for the `Point` class:

```
class Point{
    float x;
    float y;
    Point(float _x, float _y){
        x = _x;
        y = _y;
    }
}
```

So, what happens when you click in the window? Let's change the `Point` class constructor slightly so you can see it better:

```
Point(float _x, float _y){
        println(" x is: "+_x+" and y is "+_y);
        x = _x;
        y = _y;
    }
```

Now, when you click in the Processing window, you'll see the following printed out in the Console window (depending on where you click, of course):

```
x is: 262.0 and y is 51.0
x is: 234.0 and y is 193.0
x is: 362.0 and y is 274.0
x is: 125.0 and y is 340.0
x is: 17.0 and y is 155.0
```

So, why is this happening? Look at the `mousePressed(d)` method again:

```
void mousePressed(){
...
    Point newPoint = new Point(mouseX, mouseY);
    pts[count] = newPoint;
}
```

Every time the `mousePressed` event is called, you create a new `Point`, calling the constructor of the `Point` class and storing the `mouseX` and `mouseY` positions in the `Point`.

Once you've created the `Point` object, you store the point in the `pts` array so that you can use it for placing vertices in the `draw()` method.

So, in the following example, you'll use `for` loops, classes, and arrays, as well as the `vertex()` method. The last code sample was a difficult one that bears a little studying. There's a lot more information on classes in Chapter 5.

The keyPressed and key Variables

Many times you'll want to know whether someone is pressing a key on the keyboard. You can determine when a key is pressed and which key it is in two ways. The first is to check the `keyPressed` variable in the `draw()` method:

```
void draw() {
    if(keyPressed) {
        print(" you pressed "+key);
    }
}
```

Notice how you can use the `key` variable to determine what key is being pressed. Any keypress is automatically stored by your Processing application in this built-in variable.

Processing also defines a `keyPressed()` method that you can use in much the same way as the `mousePressed()` or `mouseMoved()` method:

```
void keyPressed(){
    print(" you're pressing a key \n that key is "+key);
}
```

Any code to handle key presses should go inside the `keyPressed()` method. For instance, handling arrow key presses in a game, or the user pressing the Return button after they've entered their name in a text field.

Interview: Ben Fry

Ben Fry was one of the originators of the Processing project along with Casey Reas. He has lectured worldwide on data visualization, media art, and computer science; published two books, *Processing: A Programming Handbook for Visual Designers and Artists* (coauthored with Casey Reas; MIT Press) and *Visualizing Data* (O'Reilly); and has created numerous data visualizations, illustrations, and essays.

What made you get interested in data visualization? Was it an aesthetic thought or more of a problem set that you realized wasn't being properly addressed, or some combination of the two?

Ben Fry: It's the combination of the two. I'd been interested in graphic design and computer science since an embarrassingly young age: I found advertising and logos and typography interesting even in middle school and did my first BASIC programs even younger. In college I studied graphic design but especially enjoyed information design (and motion graphics and kinetic information display). For a long time I thought that UI design was the way to go and did internships and a first job in that. But it wasn't

until grad school at MIT that I was able to bring the design and visualization sides together. At school my focus was primarily in graphic design actually, with a minor in computer science (at least for undergrad). Design was the primary subject because I thought it would give me more flexibility and teach me more about how to think. I was led to computational art because of how it melded these two (until then) very distinct interests together. One of the disappointments was actually being a few months in and realizing I was no longer "cross disciplinary," because my two primary disciplines of interest were now merged. The positive side was that I could focus on the combination in a more direct way, but the negative was that I could no longer play the two against one another the way I could when I was younger.

I see a theme in your work of the role of the designer in society, in science, in computation, and in places where visual design considerations were usually taken as being secondary to other considerations. Is that an accurate evaluation? Do you see a need for visual design in other places where it's not being valued correctly?

Ben: I think that's correct, and I see two sides to it. First, the society/science/computation aspects are what I'm most drawn to and curious about. Second, there's a more pragmatic point in that fewer people are interested in those issues, so I can carve out my own niche.

Looking at your work visually, I very often have this reaction that goes something like "Wow...oh." At first the visual beauty and complexity really strike me, and then I realize what is being visualized. Is this something intentional or something you hear a lot and is something desired?

Ben: People tend to separate making things beautiful or informative, and this disparity is especially wide when dealing with information. But unless you have some of both, I don't think it's particularly fulfilling. So, my hope is always that the project is still rewarding after the initial "Ooh!" Otherwise, the "pretty" part is easy to write off. On some level, I'm just making things that *I* want to look at, and I want them to be visually and intellectually stimulating. Of course, this quickly gets into highly subjective territory, and I'm unwilling to refer to my own work as "beautiful"—that's a personal opinion that may or may not be shared by the viewer. I also consider it a failure if people stop at "Ooh!" If the image doesn't make people curious enough to delve deeper and curious about the subject matter, then I'm not doing my job (at least as I see it).

A piece like the graphics piece "aligning humans and animals" is both an excellent illustration and a piece that sort of alludes to a world of information in a reasonably scientifically rigorous way. In that way, it reminds me of some contemporary artists (Carsten Nicolai jumps to mind) who work with difficult concepts and make them visible and legible. Can you talk about some of the contemporary artists or designers who you feel inspired by?

Ben: I get a lot of inspiration from typography, motion graphics, and film. But I don't have a good list of specific people or pieces, since it's always little tidbits here and there: the cinematography found in Kubrick movies, cheesy film info graphics, and the balance of beauty and structure (and ubiquity) in the typographic work of someone like Matthew Carter. I started a blog partly so that I could write about such things and assemble them in one place. When I find something like the scientist who's pouring molten aluminum into ant holes (*http://benfry.com/writing/archives/98*) so that he can

see their structure (after letting the aluminum cool and removing the dirt), I have somewhere to post that. But those are the sort of ideas that stick with me—associating the organic shape from the ant colony and how it connects to other shapes like it, whether spindly arrangements of lakes in northern Canada or a complicated network topology diagram.

In glancing through your writing on your blog and looking at the books you've written, the breadth of topics you cover is quite interesting—from typography to baseball players to storyboarding the government and privacy—but they all seem to have a common skein of the interpretation or misinterpretation of data. Do you or have you had a methodology for finding topics to approach?

Ben: It's largely just the things I'm curious about, but it's a way for me to assemble my thoughts. I don't tend to enjoy blogs, books, and so on that seem to have a direct connection to my work (they tend to muddle my thinking), but I'd never been able to figure that out why. As I've started assembling things, it's easier to understand where my head is. It's partly to improve at writing, and it's partly therapeutic.

As for methodology, I'm desperately trying to stick to themes around information and data, though it strays a little bit into film and motion, usually as it relates to drawing (which has to do with communication) or narrative (since that's what visualization is about). It's tough for me to stay away from politics or, say, the Space Olympics skit on last week's *Saturday Night Live*, but I don't want to spiral into general interest or an online diary.

Do you think your design work may ever lead you away from information visualization and toward something more abstractly aesthetic, or is that something you're not interested in?

Ben: Nah, I think I'm too obsessed with information. It's more likely the other way around—that it might lead me away from design and into more about how we think about data in general (for instance, along the vector of all the privacy and security posts on the blog that get to the "we have all this data, now what?"). But I love the visual side equally, so I don't really think I could ever give that up completely.

How much of the creation of Processing was simply wanting a tool for your own uses, and how much was an interest in creating a tool for others to use, and how did these two fit together?

Ben: Hmm, I think it may have been equal parts initially but then became more about the tool for others. The amount of time it takes to make a tool for yourself and to make (and support and maintain) a tool used by strangers is multiple orders of magnitude different. Even faced with that, we've chosen to continue the public part of it (often to the detriment of our own work).

That said, it's still very much driven by my personal work—specific improvements are often aligned with projects that I'm developing at one time or another. And to be honest, there are areas that we don't cover well enough (sound, for instance) because Casey and I don't make as much use of it in our own work.

I think that Processing is more of a "giving back" project. I learned programming because lots of people shared code with me (whether directly or indirectly) when I was

10 or 12 years old, so I owe them. It's also a matter of opening up this sort of work to people who are more talented and can take it further. Every once in a while I see something built with Processing that makes me want to give up on doing my own projects and just work a lot harder on Processing itself.

As a programmer I really love the Deprocess and Revisionist pieces because they document two different processes that center around a single thing: the developing of code and the running of code.

Ben: These are both about making an image of what's in one's head...that I have a general *idea* of what code changes over time look like, or what running code looks like, but I really want to *see* that. And I suspect that's what other people respond to as well—that the images aren't particularly groundbreaking (we've seen visual diffs and code traces before), but it's a bit like seeing a dream realized or reading an article that perfectly articulates the way you feel about, say, a complicated political situation or societal phenomenon.

Do you have a clear vision for the future of Processing, or has the community of processing, which is large and very active, taken over guiding the project in a sense?

Ben: I think the community has guided us in a lot of our decision making. If you look at our goals when we started, the project (online web delivery, plus support for features like serial I/O, hardware devices, and so on) and where we are now (full-scale applications, accelerated graphics, and dozens upon dozens of libraries), you can see how that invisible hand has pushed progress.

Some of the more recent developments have to do with using the API in other languages (JavaScript, Ruby, and Python), which is something that wasn't really feasible when we started but yet I'm really excited about. That's something where the community has really picked it up and run.

Are there sorts of projects that people aren't really exploring with Processing that you ever think, "I wish people were doing this more"?

Ben: One of Casey's and my personal goals in the project was that if we made the easy things easy, and made the hard things less painful, the overall level of quality in such work could perhaps improve a bit—or put another way, that you'd be able to get away with a lot less B.S. I don't think we've been as successful in this as I would like.

How did you approach the idea of creating a language for nonprogrammers? Did you refer to other "friendly" languages like Ruby or Python? Or was it driven by what you imagined users would be doing with it? Was it driven by the idea of making a "teaching" language as well?

Ben: It was a combination of all those things, plus all our accumulated biases and quibbles based on past experiences.

But to be sure, we haven't created an ultimate language of some sort—we simply tried to assemble what we knew and make an evolutionary jump forward toward where we thought things should be. If it were more revolutionary, I think the syntax and mental model would be quite different. But we had to balance that against the practical (how

far from existing languages people were willing to diverge and how quickly software we created needed to run).

Beyond the cognitive processes of analyzing data and understanding data, you also work with allowing users to interact with it. Do you find that the notion of interacting with data to be as rich a territory as the notion of displaying it legibly and informatively?

Ben: Oh, it's absolutely a rich territory, because it gets closer to involving more of our senses and/or our bodies. There are few things in our environment that we just stare at statically—we reach and manipulate them and move them around to see what they do (look at any 6-month-old baby). The more we can do this with information, the more we'll be able to learn.

Importing Libraries

One of the great aspects of using Processing is the wide and varied range of libraries that have been contributed to the project by users. Most processing libraries are contained within *.jar* files. JAR stands for *Java archive* and is a file format developed by Sun that is commonly used for storing multiple files together that can be accessed by a Java application. In Processing applications, the Java application that will be accessing the *.jar* files is the Processing environment. When you include a library in a Processing code file and run the application, the Processing environment loads the *.jar* file and grabs any required information from the *.jar* and includes it in the application it's building.

Downloading Libraries

Many Processing libraries are available for download at *www.processing.org/reference/libraries/index.html*. The libraries here include libraries used for working with 3D libraries, libraries that communicate with Bluetooth-enabled devices, and simple gesture recognition libraries for recognizing the movements made by a user with a mouse or Wii remote controller.

For this example, you'll download the ControlP5 library, install it to the Processing directory, and write a quick test to ensure that it works properly. First, locate the ControlP5 library on the Libraries page of the Processing site under the Reference heading. Clicking ControlP5 on the Libraries page brings you to the ControlP5 page at *http://www.sojamo.de/libraries/controlP5/*. Once you've downloaded the folder, unzipping the *.zip* file will create the controlP5 folder. Inside this is a *library* folder that contains all the *.jar* files that the Processing application will access.

Now that you've downloaded the library, the *libraries* folder of your Processing sketchbook. The Processing sketchbook is a folder on your computer where all of your applications and libraries are stored. To change the sketchbook location, you can open the Preferences window from the Processing application and set value in the

"Sketchbook location" field. You'll need to copy the contributed library's folder into the "libraries" folder at this location. To go to the sketchbook, you hit Ctrl-K (⌘-K on Mac OS X). If this is the first library you've added, then you need to create the "libraries" folder. For instance, on my computer the Processing sketchbook is installed at */Users/base/processing*, so I place the *controlP5* folder at */Users/base/processing/libraries/*. Your setup may vary depending on where you've installed Processing and your system type. Once the library is in the correct location, restart the Processing IDE, and type the following code in the IDE window:

```
import controlP5.*;
```

Then, run the application. If a message appears at the bottom of the IDE saying this:

```
You need to modify your classpath, sourcepath,
 bootclasspath, and/or extdirs setup. Jikes could not find package
 "controlP5" in the code folder or in any libraries.
```

then the ControlP5 library has not been created properly. Double check that the folder is in the correct location. If you don't get this message, you've successfully set up ControlP5. We'll talk about it in greater depth in Chapter 7. For now, we'll talk about some of the more common libraries for Processing and what they do:

Minim by Damien Di Fede
: This uses the JavaSound API to provide an easy-to-use audio library. This is a simple API that provides a reasonable amount of flexibility for more advanced users but is highly recommended for beginners. It's clear and well documented.

OCD by Kristian Linn Damkjer
: The Obsessive Camera Direction (OCD) library allows intuitive control and creation of Processing 3D camera views.

surfaceLib by Andreas Köberle and Christian Riekoff
: This offers an easy way to create different 3D surfaces. It contains a library of surfaces and a class to extend.

Physics by Jeffrey Traer Bernstein
: This is a nice and simple particle system physics engine that helps you get started using particles, springs, gravity, and drag.

AI Libraries by Aaron Steed
: These are a great set of libraries to assist with artificial programming tasks such as genetic algorithms and the AStar pathfinding algorithm. We'll discuss this in greater detail in Chapter 18.

bluetoothDesktop by Patrick Meister
: This library lets you send and receive data via Bluetooth wireless networks.

proMidi by Christian Riekoff
: This library lets Processing send and receive MIDI information.

oscP5 by Andreas Schlegel
: This library is an OpenSound Control (OSC) implementation for Processing. OSC is a protocol for communication among computers, sound synthesizers, and other multimedia devices.

Almost all of these will be discussed in later chapters when demonstrating how to extend Processing to help you work with physical interactions, other programming languages like Arduino, and other devices like GPS devices and physical controls.

Loading Things into Processing

Now that we've covered some of the basics of drawing with processing and some of the basics of capturing user interaction, you'll learn how the Processing environment loads data, images, and movies. Earlier, we mentioned the default setup of a Processing project, with the folder *data* stored in the same folder as the *.pde* file. The Processing application expects that anything that's going to be loaded into the application will be in that folder. If you want to load a file named *sample.jpg*, you should place it in the *data* folder within the same folder as the *.pde* file you're working with.

Loading and Displaying Images

First, you'll learn how to load images. One class and two methods encapsulate the most basic approach to loading and displaying images: the `PImage` class, the `loadImage()` method, and the `image()` method.

The PImage class

Processing relies on a class called `PImage` to handle displaying and sizing images in the Processing environment. When you create a `PImage` object, you're setting up an object that can have an image loaded into it and then can be displayed in the Processing environment. To declare a `PImage` object, simply declare it anywhere in the application, but preferably at the top of the code, as shown here:

```
PImage img;
```

Once you've declared the `PImage` object, you can load an image into it using the `loadImage()` method.

The loadImage() method

This method takes as a parameter the name of an image file and loads that image file into a `PImage` object. The name of the file can be either the name of a file in the filesystem (in other words, in the *data* folder of your Processing applications home folder) or the name of a file using a URL. This means you can do something like this to load a JPEG file from the Internet:

```
PImage rocks;
rocks = loadImage("http://thefactoryfactory.com/images/hello.jpg");
```

Or you can do the following to load an image from the *data* folder within your Processing applications folder:

```
PImage rocks;
rocks = loadImage("hello.jpg");
```

The `loadImage()` method lets you load JPEG, PNG, GIF, and TGA images. Other formats, such as TIF, BMP, or RAW files, can't be displayed by Processing without doing some serious tinkering.

Now that you've looked at implementing the class that helps you display images and you've seen how to load images into the application, let's look at actually displaying those images using the `image()` method.

The image() method

The `image()` method takes three parameters:

```
image(PImage, xPosition, yPosition);
```

The first is the `PImage` object that should be displayed. If the `PImage` has not been created with `loadImage()`, then trying to put it on the screen using the `image()` method will cause an error. There are ways around this, but we'll leave those to Chapter 10. For now, let's say that until a `PImage` object has had an image loaded into it, don't try to use it with the `image()` method. The next two parameters are the x and y positions of the upper-left corner of the image. This is where the image will be displayed. Here's a simple example:

```
PImage img;

void setup() {
    size(400, 400);
    img = loadImage("sample.jpg");
    image(img, 0, 0);
}
```

As soon as the `setup()` method is called, the window is given a size, the *sample.jpg* file is loaded into the `PImage`, and the `PImage` is placed in the window at the 0, 0 position (that is, in the upper-left corner of the window). You can also size the `PImage` using the `image()` method. Like so many methods in Processing, the `image()` method is overloaded, letting you pass optional `width` and `height` values that will size the image:

```
image(img, x, y, width, height)
```

By default, the image is displayed in the Processing window at its default height and width, but you can also set the `height` and `width` of the image. Be aware, though, that the image may have a different aspect ratio and might look stretched or squashed slightly if using a size that doesn't match the original aspect ratio.

Displaying Videos in the Processing Environment

The Processing environment contains a `Movie` class to display videos. This class relies on the Apple QuickTime video libraries, so if your computer doesn't have the QuickTime libraries installed on it, you'll need to download and install them for your operating system. Note that at this time, working with video in Processing on Linux is a fairly difficult and involved process. The GSVideo library is fairly new and has a somewhat difficult setup procedure; however, it does appear quite workable and allows Linux users to work with different video formats in Processing. For the sake of brevity in this book, we'll leave you to explore it on your own, if you're curious.

Using the Movie Class

The `Movie` class lets you load QuickTime movies; load movie files with *.mov* file extensions; play, loop, and pause movies; change the speed; and tint the video. Creating a `Movie` object is somewhat similar to creating an image using the `PImage` class, with one important distinction: you need to first import all the libraries that contain the information about the `Movie` class and how it works. These are all stored in a separate location that can be accessed by using the `import` statement, as shown here:

```
import processing.video.*;
```

Next, declare a `Movie` variable:

```
Movie mov;
```

In the `setup()` method, you need to *instantiate*, or create, a new instance of that `Movie` class. You do this by calling the constructor of the `Movie` class. The constructor of the `Movie` class is another overloaded method that has four signatures:

```
Movie(parent, filename)
Movie(parent, filename, fps)
Movie(parent, url)
Movie(parent, url, fps)
```

The `parent` parameter is the Processing application that will use and display the `Movie` class. For a simpler application, the parent is almost always the application itself, which you can reference using the `this` keyword:

```
Movie(this, http://sample.com/movie.mov");
```

This isn't vital to understand, but all Processing applications are instances of a Java class called `PApplet`. The `PApplet` class handles calling the `setup()` and `draw()` methods and handling the mouse movement and key presses. Sometimes, if you're referencing a Processing application from a child object loaded into it, the parent `PApplet` will be the application that your child is loaded into. This is fairly advanced stuff and not something you'll likely be dealing with, but if you come across the parent parameter, the `this` keyword in an application, or the `PApplet`, you'll know that all these things refer to the main application class.

The `filename` and `url` parameters work much the same as they do in the `loadImage()` method. When loading a local file stored on the machine that the Processing application is running on, use the filename of the *.mov* file. When loading a file from a site on the Internet, use a URL. Finally, as an optional parameter, you can pass a frame rate that the movie should be played at, if the frame rate of your movie and the frame rate of the Processing application are different. In the `setup()` method of this application, using the reference to the Processing application and the local filename, you'll instantiate the `Movie` object:

```
void setup() {
    size(320, 240);
    mov = new Movie(this, "sample.mov");
    mov.play();
}
```

In this code snippet, take a look at the `play()` method. This is important because it tells the Processing environment to start reading from the movie file right away. Without calling this method, the Processing application won't read the frames from the movie to display them. It's important to call either the `play()` or `loop()` method whenever you want the Processing environment to begin displaying your movie.

To read a movie, you need to define a `movieEvent()` method in the application. Why? Well, as the QuickTime Player plays the movie, it streams its video in pieces or frames (not too different from the frames of a film movie) to the Processing application, which then displays them in the window. The `movieEvent()` method is the QuickTime Player notifying the Processing environment that a frame is ready to be displayed in much the same way that the `mousePressed()` method is the machine notifying the Processing environment that the mouse has moved. To get the information for the frame, you want to call the `read()` method of the `Movie` class. This reads the information for the frame into the `Movie` class and prepares the frame for display in the Processing environment:

```
void movieEvent(Movie m) {
    m.read();
}
```

To draw the current frame of the movie into the window, you use the `image()` method again just the same as you did for a `PImage` that contained all the information for a picture:

```
void draw() {
    image(mov, 0, 0);
}
```

This draws the current frame of the movie into the 0, 0, or upper-left, corner of the window. Figure 3-7 shows the results of the complete code listing:

```
import processing.video.*;
Movie mov;

void setup() {
    size(320, 240);
```

```
    mov = new Movie(this, "sample.mov");
    mov.play();
}

void movieEvent(Movie m) {
    m.read();
}

void draw() {
    image(mov, 0, 0);
}
```

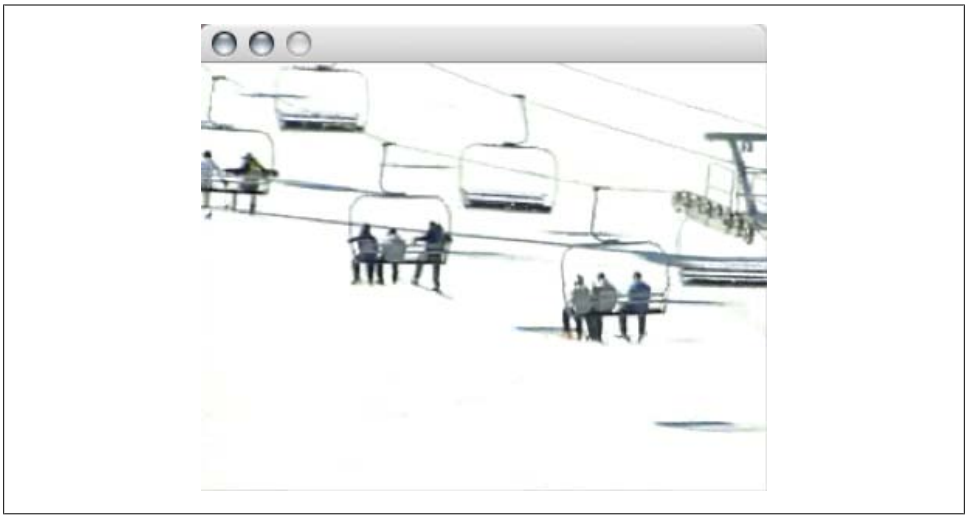


Figure 3-7. Showing a video using the Movie class

Reading and Writing Files

The last thing to look at is reading in files. You can read in a file in two ways, and the one you should use depends on the kind of file you want to load. To load a picture, you would use the `loadImage()` method; to load a simple text file, you would use the `loadStrings()` method; and to load a file that isn't a text file, you would use the `loadBytes()` method. Since loading and parsing binary data with the `loadBytes()` method are much more involved, we'll leave those topics for Chapter 12 and focus instead of reading and writing simple text files.

The loadStrings() method

The `LoadStrings()` method is useful for loading files that contain text and only text. This means that loading more complex text documents like Microsoft Word files isn't going to work as smoothly as you'd like because they contain lots of information other than just the text. Generally, files with a *.txt* extension or plain-text files without an

extension are OK. This means that files created in Notepad or in TextEdit will be loaded and displayed without any problems. To load the file, you call the `loadStrings()` method with either the filename or the URL of the text that you want to load:

```
loadStrings("list.txt");
```

Now, that's not quite enough, because you need to store the data once you have loaded it. For that purpose, you'll use an array of `String` objects:

```
String lines[] = loadStrings("list.txt");
```

This code makes the `lines` array contain all the lines from the text file, each with its own string in the array. To use these strings and display them, you'll create a `for` loop and draw each string on the canvas using the `text()` method, as shown here:

```
void setup(){

    size(500, 400);

    PFont font;
    font = loadFont("Ziggurat-HTF-Black-32.vlw");
    textFont(font, 32);

    String[] lines = loadStrings("list.txt");
    print("there are " + lines.length + " lines in your file");
    for (int i=0; i < lines.length; i++) {
        text(lines[i], 20 + i*30, 50 + i*30);//put each line in a new place
    }
}
```

As a quick reminder, the text file that you load must be in the *data* folder of your Processing application or be downloaded from the Internet.

The saveStrings() method

Just as you can read strings from a text file with the `loadStrings()` method, you can write them out to a file just as easily. The process is somewhat inverted. First, you make an array of strings:

```
String[] lines = new String[4];
lines[0] = "hello";
lines[1] = "out";
lines[2] = "there";
lines[3] = "!";
```

After creating an array, you can write the data back to a file by passing its name to the `saveStrings()` method:

```
saveStrings("data/greeting.txt", lines);
```

This creates and saves a file called *greeting.txt* with each element from the `lines` array on a separate line in the file.

Running and Debugging Applications

To run your application, simply click the Run button in the Processing IDE; you'll see the output shown in Figure 3-8.

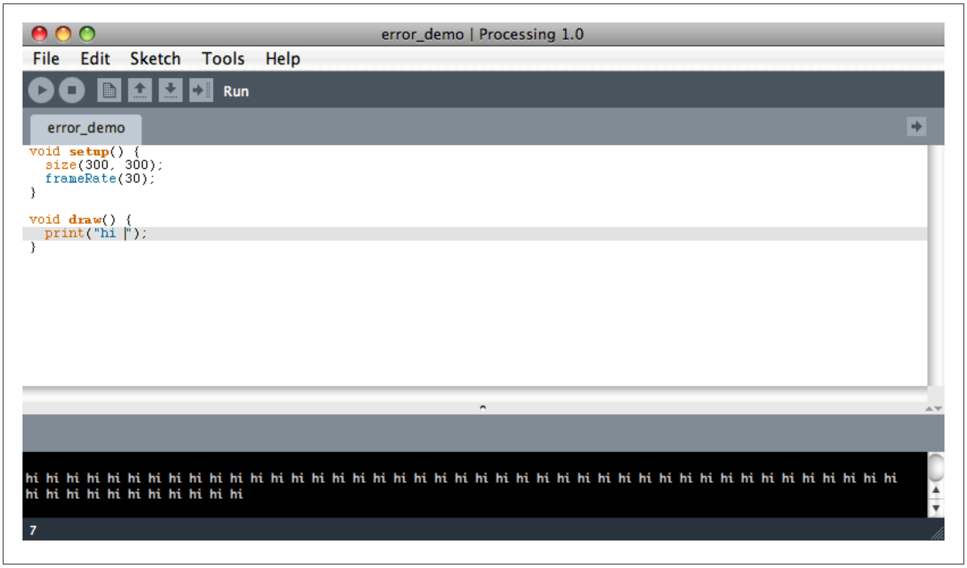


Figure 3-8. Running a Processing application

It couldn't be easier, could it?

What happens when it doesn't run? Look at the Message window in Figure 3-9.

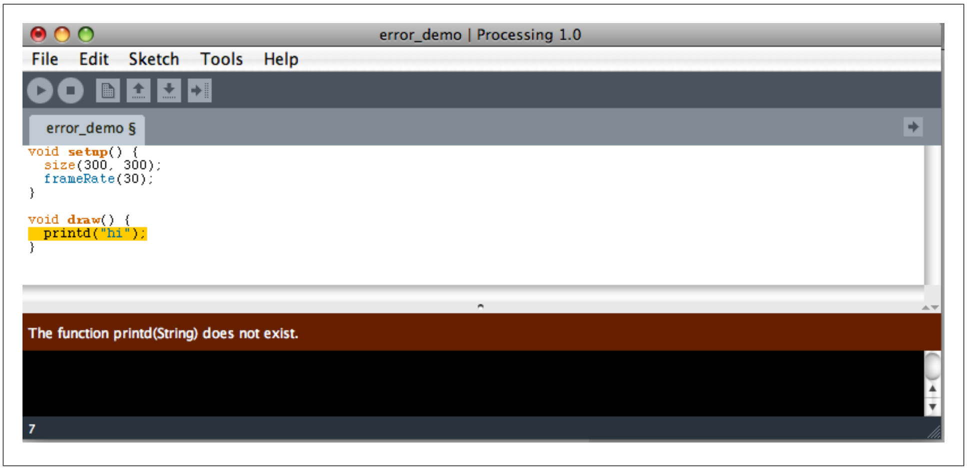


Figure 3-9. Viewing an error in the Message window

Note the message, which is, in this case, very helpful:

```
The function printd(String) does not exist.
```

You see that the method you tried to call, `printd()`, doesn't exist. The Processing environment will also return deeper errors. For example, if you enter the following line in a `setup()` method:

```
frameRate(frames);
```

you may see this in the Console window:

```
No accessible field named "frames" was found in type "Temporary_85_2574".
```

This error indicates you've forgotten to define the variable frames. Change that line to this:

```
String frames = "foo";
frameRate(frames);
```

You'll see the following in the Console window:

```
Perhaps you wanted the overloaded version "void frameRate(float $1):" instead?
```

This tells you the `frameRate()` method does not accept a string as a parameter. Instead, it takes a float or an integer. Since the Processing IDE always highlights the offending line, figuring out which line is causing the problems tends to be quite easy. Some errors, though, aren't so easy to figure out. In these cases, checking the Processing forums at *http://processing.org/discourse/yabb_beta/YaBB.cgi* is usually your best bet. Thousands if not tens of thousands of Processing users from all over the world ask and give advice on these forums. If you're having a problem, chances are someone has had it before, asked there, and gotten an answer.

Exporting Processing Applications

Now, while running and debugging certainly helps when you want to see how the application is doing and check for errors in the code, it doesn't much help when you're trying to share it with a friend. To do that, you need to export the application.

The first step is to either select the Export Application option in the File menu or hit Ctrl-E (⌘+E on OS X). This will bring up a dialog, shown in Figure 3-10, asking you what operating systems you would like to create a dialog for.

Once you click the Export button, you'll be shown the location of your created executable files in your filesystem (Figure 3-11). There should be four folders, one for each operating system that you've created an application for and one for an applet that can run in a web browser. Each folder for the operating system version contains your application, compiled and ready to run on a desktop machine. The fourth folder contains the file necessary to place your application in a website where it can be viewed in a browser.

Figure 3-10. Exporting an application

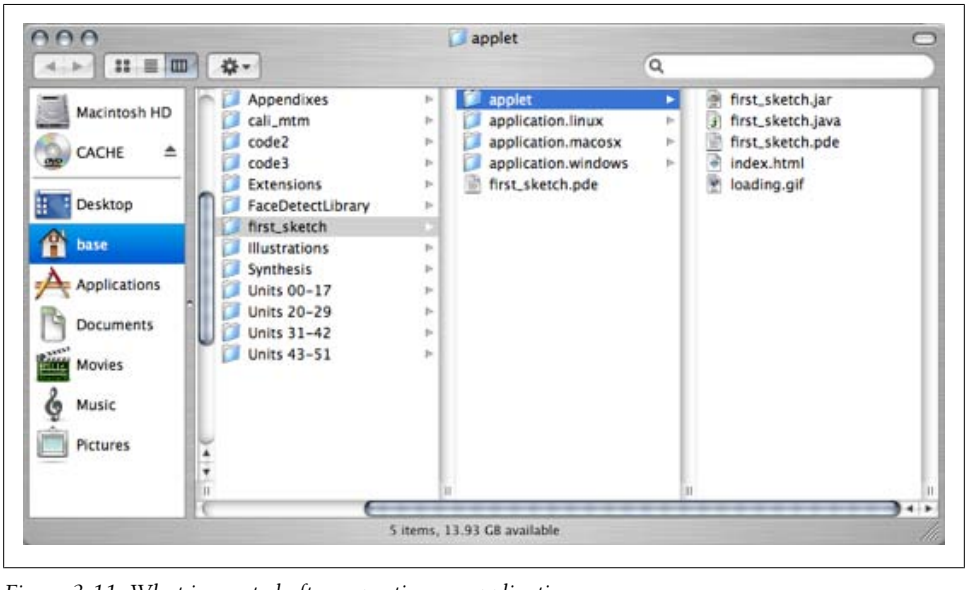


Figure 3-11. What is created after exporting an application

If you take a slightly closer look at what is generated by the export, you'll see four distinct folders. The first folder is the *applet* folder. This folder contains all the files that you'll need in order to put your Processing application on the Internet. You'll see each file in the order that it appears in. This application is named *first_sketch*, so all the files will have that name:

first_sketch.jar
: This is the Java archive, or *.jar* file, that contains the Java runnable version of the sketch. When someone views the web page that contains your application, this file contains information about your application that will be used to run it.

first_sketch.java
: This is the Java code file that contains the code that the Java Virtual Machine will run to create your application.

first_sketch.pde
: This is your Processing code file.

index.html
: This is an HTML web page generated by the export that has the *first_sketch.jar* file embedded in it. When users open this page in a browser, if they do not have the correct Java runtime installed, a message directs them to the Sun site, where they can download it. This file is created, so that you can simply post the file to the Internet after you've created your application and have a ready-made page containing your Processing application.

loading.gif
: This *.gif* file is a simple image to show while the Java runtime is preparing to display your application.

So, in order to display your application online, simply place all the files in this folder in a publicly accessible location on a website.

The other three folders contain runnable executable versions of your processing application for each of the three major home operating systems: Windows, Mac OS X, and Linux. A Windows application for instance, has an *.exe* extension; an OS X application has an *.app* extension; and a Linux executable application doesn't have any extension. Each of these three options lets you show your application outside the browser. If you have a friend who has a Windows computer that you want to send the Processing application to, you would simply compress the folder and all the files it contains into an archive and send it to that person. They could then unzip the file and run it by clicking the executable file. Each of these folders contains somewhat different files, but they all contain the *.java* file for their application and the Processing *.pde* file within their source folder. This is important to note because if you don't want to share the source of your application, that is, the code that makes up your Processing application, remove this folder. If you want to share (remember that sharing is a good thing), then by all means leave it in and share your discoveries and ideas.

Conclusion

If you're already thinking of exploring more about Processing, head to the Processing website at *http://processing.org* where you'll find tutorials, exhaustive references, and many exhibitions of work made with Processing. Also, the excellent book,

Processing: A Programming Handbook for Visual Designers and Artists (MIT Press), written by Casey Reas et al., provides a far more in-depth look at Processing than this chapter possibly could.

What next? Well, if you're only interested in working with Processing, then skip ahead to Part II, *Themes*, where we explore different themes in computation and art and show you more examples of how to use Processing. If you're interested in learning about more tools for creating art and design with, then continue with the next chapter. In Chapter 4, we discuss the open source hardware initiative that helps create physical interactions and integrates nicely with Processing.

Review

Processing is both an IDE and a programming language. Both are included in the download, which is available at *processing.org/download*.

Processing is based on the Java programming language, but is simplified to help artists and designers more easily prototype visual and interactive applications.

The Processing IDE has controls on the top toolbar to run an application; stop an application; and create, open, save, or export an application. All of these commands are also available in the File menu or as hotkeys.

Clicking the Run button compiles your code and starts running your application.

A Processing application has two primary methods: the `setup()` method, which is called when the application starts up, and the `draw()` method, which is called at regular intervals.

You set the number of times the `draw()` method is called per second using the `frameRate()` method.

The Processing drawing methods can be called within the `draw()` or `setup()` method, for example, with `rect()`, `ellipse()`, or `line()`. The method `background()` clears all drawing currently in the display window.

When drawing any shape, the `fill()` method determines what color the fill of the shape will be. If the `noFill()` method is called, then all shapes will not use a fill until the `fill()` method is called again.

Processing defines `mouseX` and `mouseY` variables to report the position of the user's mouse, as well as a `key` variable and a `keyPressed()` method to help you capture the user's keyboard input.

You can import libraries into Processing by downloading the library files and placing them in the Processing *libraries* folder and restarting the Processing IDE. Once the library has been loaded, you can use it in your application with the `import` statement to import all of its information, as shown here:

```
import ddf.minim.*; // just an example, using Minim
```

The `PImage` object and `loadImage()` method allow you load image files into your Processing application and display and manipulate them.

The `Movie` class enables you to load QuickTime movies into your application, display them, and access their pixel data.

The `saveStrings()` and `loadStrings()` methods let you load data from text files, with a *.txt* extension, and also to save data to those files.

The `print()` method is helpful when debugging because it lets you trace the values of variables at crucial points where their values are being changed.

The Processing IDE prints any error messages to the Console window at the bottom of the IDE. These messages can be helpful when trying to determine the cause of an error or bug. This Console window is also where any print messages are displayed.

You can export a Processing application as either a standalone executable, an *.exe* file for Windows, an *.app* file for Mac OS X, or you can export it for the Web as a Java applet.

CHAPTER 4
Arduino

The word *Arduino* refers to three separate tools, which, bundled together, create the toolkit that we refer to as Arduino. First, there is the Arduino controller, which exists in several forms, from large to small to a freely available schematic that anyone with the requisite knowledge can assemble for about $12. Second, there is the language and compiler, which creates code for this microcontroller, and which, much like the Processing language, simplifies many of the tasks that challenge designers and developers alike when working with hardware and physical interaction. Finally, there is the Arduino programming environment, which, again like the Processing IDE, is a simple open source IDE built in Java. Therefore, because the word *Arduino* can have several meanings, I'll be quite specific when referring to a particular aspect of the environment, such as the *Arduino language*. When I simply refer to *Arduino*, I'm referring to the environment as a whole.

So, with that established, what is Arduino? The *Arduino environment* is a goal, and that goal is to simplify the creation of interactive applications or objects by simplifying the programming language used to create instructions and by providing a powerful yet basic controller that can easily be used for many common programming tasks while still being robust enough to support more complex projects. The Arduino controller is one of the most remarkable and popular open source projects because it enables so much. Programming microcontrollers can be, to the untrained, daunting and frustrating. Yet, being able to create working computers the size of a matchbox that can easily interact with hardware opens up an entirely new world of possibilities to interactive designer and artists.

> Tinkering is what happens when you try something you don't quite know how to do, guided by whim, imagination, and curiosity. When you tinker, there are no instructions, but there are also no failures, no right or wrong way of doing things. It's about figuring out how things work and reworking them.
>
> —Massimo Banzi, one the originators of the Arduino project

What sorts of things do people make with Arduino? They create physical controls that people can interact with, including buttons, dials, levers, and knobs. They create interactive environments that use weight sensors, ultrasonic and infrared sensors, and temperature sensors. They create ways to control their appliances, control the lights in their houses, and control video cameras or still cameras (what is generally called *home automation* or a *smart house*). They create small robotics or mechanical toys. They create small autonomous programs that communicate with one another and send signals, thereby creating miniature or not-so-miniature networks. The potential of the controller, what it can do, and what it can enable you to do is enormous. So, with little further ado, let's get started.

Starting with Arduino

First, and perhaps most important, you'll need an Arduino controller. This can be any one of many controllers, though I recommend that beginners use the Duemilanove controller or the Arduino Mini. Of the two, the Duemilanove is probably easier for beginners to start. The differences between these two will be covered in the section "Duemilanove Versus Mini" on page 97. If you're already comfortable with electronics and wiring and you have a compelling reason, then you can easily apply the information in this chapter to one of the other Arduino controllers. Note that if you have decided to use the Mini controller, you'll need a Mini USB adapter in order to upload applications to the Arduino controller. We'll explain exactly what that means later, but for the time being, make sure that if you get a Mini, you also get a Mini USB adapter with it. The Duemilanove doesn't require a Mini USB adapter because a USB port is attached to the board. There is also, as of the writing of this book, the Mega, which provides a great deal more computational power and pins for input and output. In addition, there is the Pro and the Pro Mini; you might explore these later after you've grown more comfortable with the Arduino. All of these are available from a great number of online electronics supply stores; a quick online search will be able to guide you to a few places.

You'll also need a USB cable with one end that is USB A and the other USB B, as shown in Figure 4-1.

You'll use the USB cable to connect the Arduino controller to your computer, since the Arduino controller connects only with the USB B (the most square shape), and most computers use only the USB A (the more rectangular shape). If you don't have a Duemilanove or similar board that has a built-in LED, it's also recommended that you purchase an LED light from an electronics store for testing purposes. Using a blinking light is a great way of verifying that the board is working and is a great tool for debugging code.

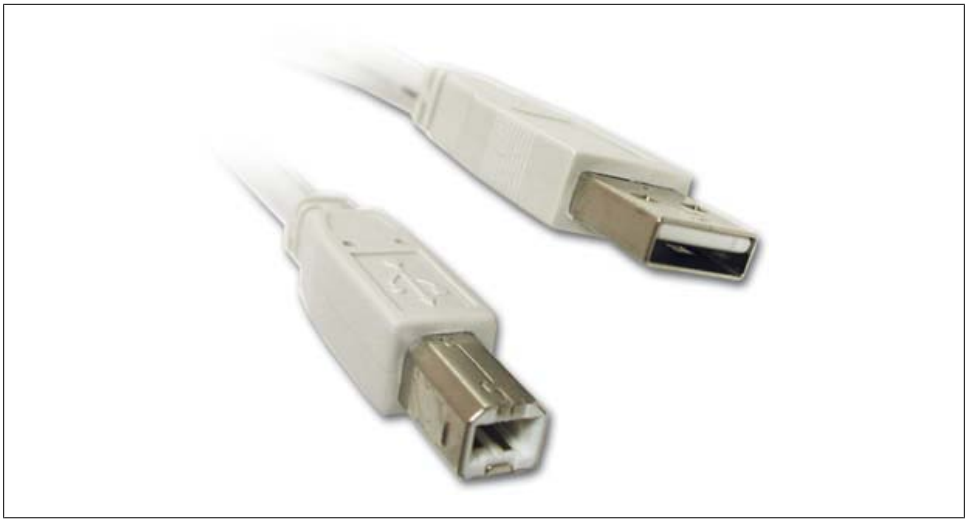


Figure 4-1. USB A and USB B cables

Installing the IDE

The next part of the Arduino controller that you'll need is the IDE, which you can download from the Arduino website at *www.arduino.cc/en/Main/Software*. Select your operating system, and download the zipped files. Once you've downloaded the files, unzip them. On a Mac, you should see something like Figure 4-2.

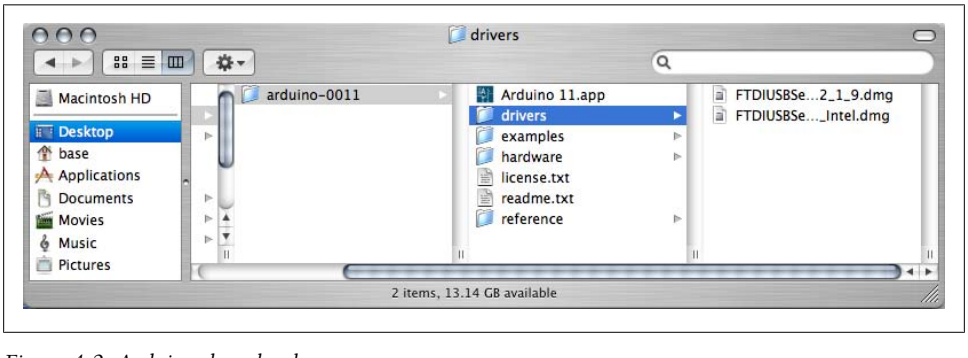


Figure 4-2. Arduino download

The drivers are what allow your computer to communicate with the Arduino board. Without these drivers installed, you won't be able to communicate with the Arduino, so it's important that they are installed before you try to upload any code. The file structure on a Windows or Linux computer will be the same, but the process for installing will be slightly different.

Mac OS X

To install the Mac drivers, simply double-click the appropriate drivers for your processor type. Once the *.dmg* file is mounted, double-click the *.pkg* file contained in the *.dmg* file to install the drivers. You can now plug in the controller.

Windows

The easiest way to install the Windows drivers for the Arduino controller is simply to plug the controller into your computer and wait for the Found New Hardware Wizard to appear. You may need to open the Device Manager, as shown in Figure 4-3.

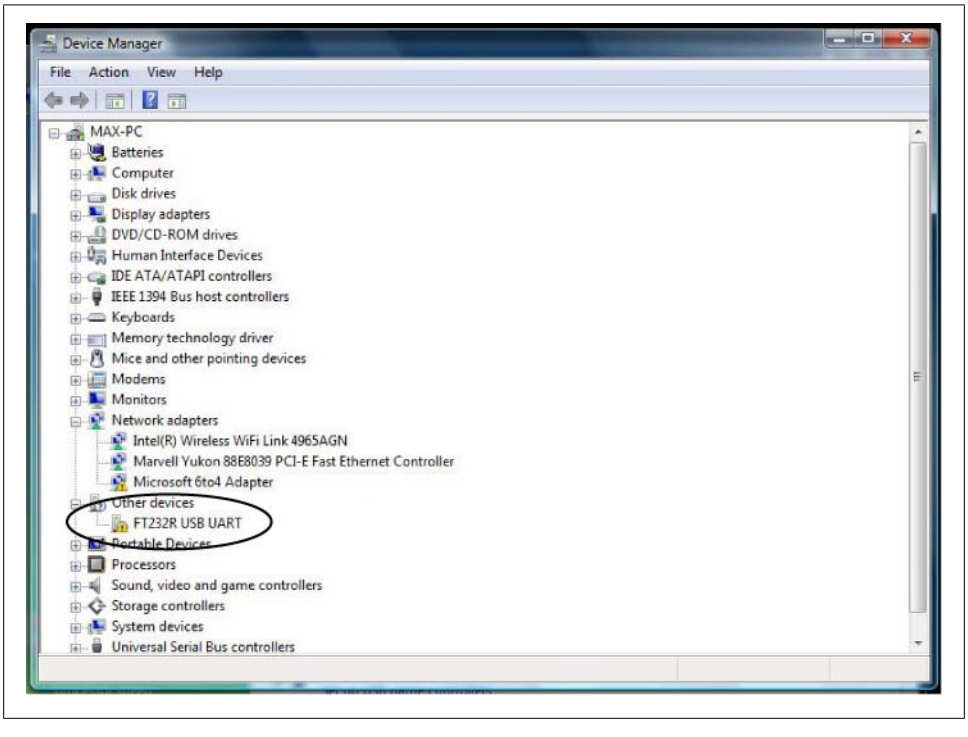


Figure 4-3. Selecting devices

Continue to the "Please choose your search and installation options" window, and add the location of the drivers folder to the locations that Windows will search for the driver in by adding it to the "Search for driver software in this location" field, as shown in Figure 4-4.

Click Next until you get to the final screen, and then click Finish. The drivers are now installed, and you're ready to go.

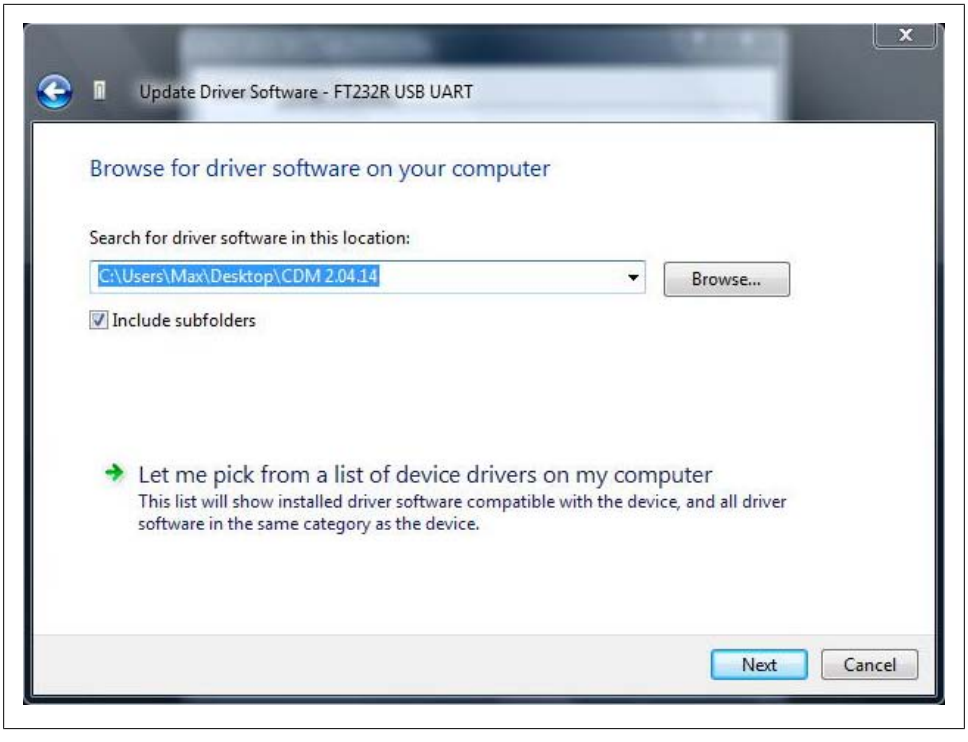


Figure 4-4. Browsing for the installer files

Linux

I hate to perpetuate stereotypes about Linux users, but I'm going to assume that if you're running Linux, you're clever enough to troubleshoot a lot of the difficulties and peculiarities of your particular distribution. With that in mind, here's a slightly more high-level overview of what's involved in getting Arduino running on a Linux machine, because it's a slightly more involved process. You will need to install the following programs:

- The Java Runtime Engine (called the JRE)
- avr-gcc (aka "gcc-avr")
- avr-libc

How you install these programs is going to depend on which distribution you are using. You will also need a compatible kernel. If you have `brltty` installed (the default on recent versions of Ubuntu), you'll need to remove it. Once you've done this, download the latest Arduino Linux distribution, extract these files to a directory, and run the *arduino* script. If you encounter problems, consult the Linux install instructions at *www.arduino.cc/playground/Learning/Linux*, where there are links to more detailed instructions for several of the most popular Linux distributions.

Configuring the IDE

Once you've installed the drivers, you can run the IDE. On all systems, the Arduino IDE is an executable in the folder where the download was unzipped. You shouldn't move the actual executable because it requires the */hardware* folder in order to operate correctly. Once you start up the IDE, you'll need to configure it correctly. This means telling the IDE which board you're using and which serial port is connected to the Arduino controller. To select the board, go to Tools→Board, and select the board type, as shown in Figure 4-5.

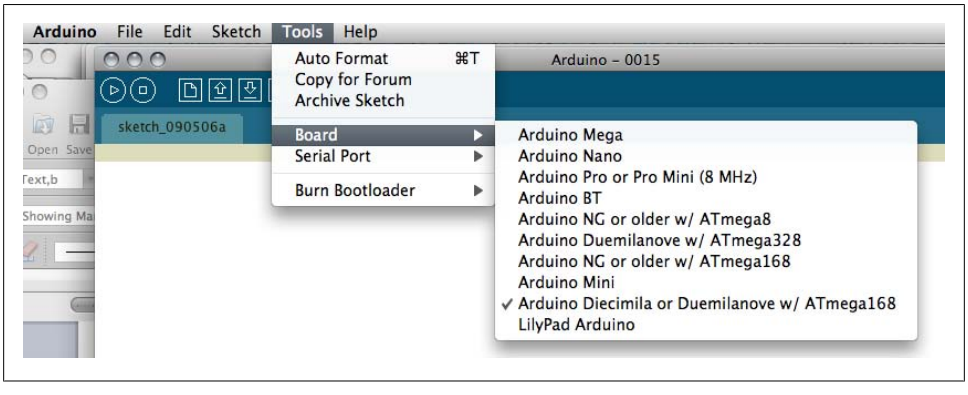


Figure 4-5. Selecting the Arduino board

Next, you'll need to select the port on which the board will operate. On a Macintosh, this will be the option with the letters *tty* in it, as shown in Figure 4-6.

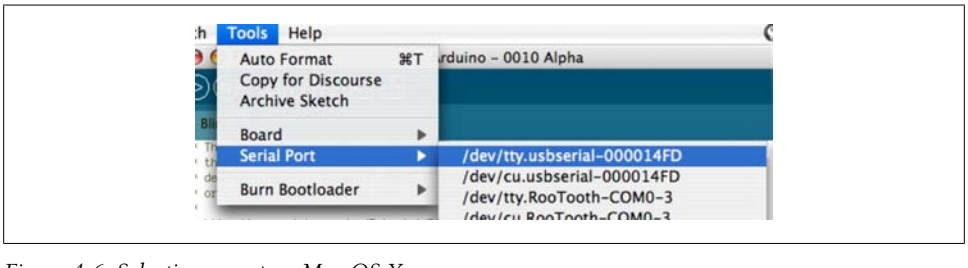


Figure 4-6. Selecting a port on Mac OS X

On a Windows computer, the ports will have the word *COM* in the title. You may need to do a little testing to determine which port is the correct one on your computer. Once you get the correct port, you're ready to go. If you'd like to run a simple script to check whether your board is working correctly, look ahead to the section titled "Touring the Arduino IDE" on page 102.

Touring Two Arduino Boards

Now that you know how to install the Arduino IDE and drivers, you can move on to looking at the boards themselves. The Mini and Duemilinove are quite similar, but the differences between them are substantial.

The Controller

So, what is this controller? Really, it's just the same thing as your average computer, minus the hard drive and a few other things you might be used to having. The core elements are there, however. Input/Output (I/O) is done through pins and the USB port, the processor is there, and there is a small amount of random access memory (RAM) that acts much the same as RAM in a larger computer. Of course, all of these things are on a far smaller scale than a laptop or desktop computer, which means that the engineering is quite different in some respects, and there are many different issues that you need to consider with a microcontroller that you would not need to consider with a normal computer, particularly when working with powering other devices and powering the controller itself. That said, the general principles still apply, which makes working with microcontrollers a good way to learn more about the inner workings of computers and of computing.

Duemilanove Versus Mini

We'll be looking at two controllers. The Duemilanove controller, shown in Figure 4-7, is a slightly larger controller that is a little easier to work with for beginners, since components can be attached directly to the controller.

The other Arduino controller that we'll examine is the Mini. This controller is much smaller, and that changes the types of projects that it can be used with. The Mini requires a separate USB adapter (see Figure 4-8) that allows you communicate with the controller over USB for uploading code and receiving messages over the Serial port. The Mini also requires that you purchase a prototyping board in order to wire any components to it or to power it separately from a computer using a battery. This isn't a great impediment, and you should purchase a prototyping board anyway for doing any kind of serious work with the Arduino controller.

The Duemilanove comes with small LED lights that you'll find next to the USB port on the Duemilanove. When you connect the controller to power, if there is a program currently loaded into the memory of the controller, a light should begin blinking. The Mini does not have such an easy way to tell that it's plugged in and powering up correctly, but making a small program that can turn on an LED that is connected to one of the Mini's pins is easy enough to do. We'll cover doing that a little bit later in the section "Hello World" on page 117.

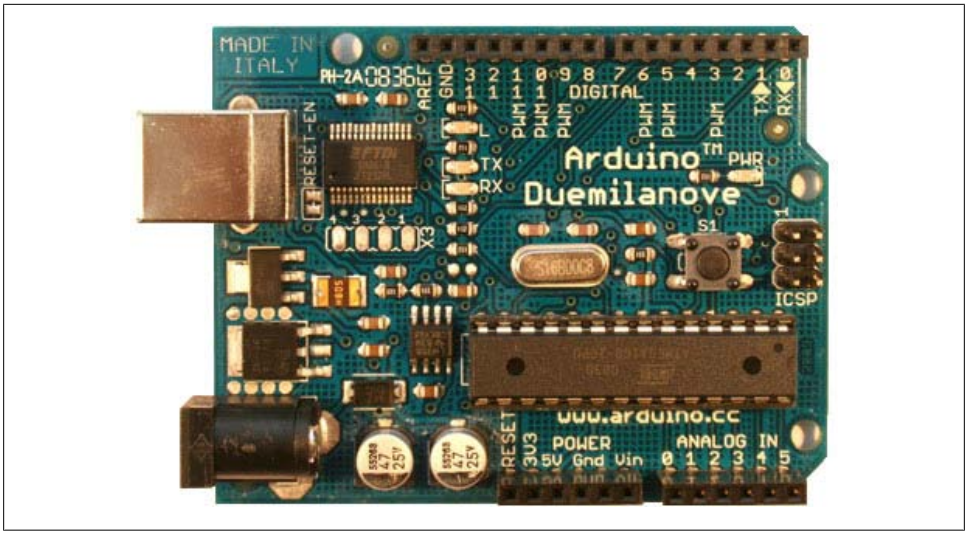


Figure 4-7. Arduino Duemilanove controller

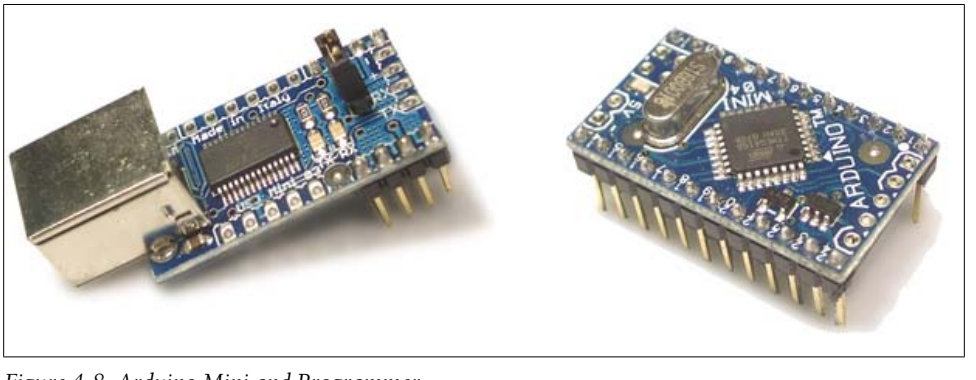


Figure 4-8. Arduino Mini and Programmer

Duemilanove means *2009* in Italian, the first language of Massimo Banzi, who is one the originators of the Arduino project. It is designed to be easy to understand and to work for those without experience in electronics. It has 20 pins (6 analog and 14 digital), 6 pulse-width modulation-enabled pins, a TX and RX pin pairing, and an I2C port. None of that makes sense? Then read on.

What's a pin?

A *pin* provides an input or output through which the controller can communicate with components. Smallish copper wires can be inserted into the pins connectors, and you're off to the races. When you look at the Duemilanove, you'll see a row of small black holes along either side of the controller that a wire can be inserted into.

Digital versus analog pins

Digital pins have two values that can be read or written to them: high and low. *High* means that 5 volts (V) is being sent to the pin either from the controller or from a component. *Low* means that the pin is at 0V. Now, start to imagine the sorts of information that this can encompass: on/off, there/not there, ready/not ready. Any kind of binary information can be read or written to a digital pin.

Analog pins are slightly different; they can have a wide range of information read or written to them. How wide a range? Well, from 0 to 255 can be written, which represents 256 steps of voltage information, once again starting with 0V and going up to 5V. Analog values from 0 to 1,023 can be read (representing voltages from 0 to 5V). These pins are what we use to read and write information that has a range of values, such as the position of a dial, the distance of an object from an infrared sensor, or the brightness of an LED light.

In addition to the digital and analog pins, there are connectors relating to the powering of components.

Figure 4-9 shows the Duemilanove. Note that the digital pins are at the top of the picture. These are where you'll plug in any controls that communicate using digital signals, that is, either an on or off state. We'll talk more about this later in this chapter, but for the moment, understand that when we're referring to the digital ports, the ports at the top of the board are what we mean. Some of the ports are also enabled for pulse width modulation (PWM), which means, in essence, that they can be used to send analog information to a component. Those are ports 3, 5, 6, 9, 10, and 11. That means that these ports can be set to do things other than just digital communication. The possible values of these pins is IN (that is, the information is coming from the component to the control) and OUT (meaning the information is going to the component from the controller).

At the bottom of the controller, we have the power connectors that provide 5V power, provide 3.3V power, and provide two ground pins that can be used to ground any components that you attach to the Arduino controller. Power, voltage, and ground are all defined in the Programming Glossary at the end of the book, so if you're not quite familiar with those terms, then make sure to read it.

Just to the right of the power pins are the Analog In pins. These allow you receive analog information, that is, information in a range of voltage from 0 to 5V, in 4.9mV (millivolt) increments, which means that the analog information can be between 0 and 1,023. These pins are quite important, because many different types of controls send analog information in a range of values, such as a dial being turned or an infrared sensor sending range information.

Above the analog pins is the processor that the Arduino controller runs. Just above the processor is the reset button. This allows you to restart your program. This is important to know because the controller saves any program uploaded to it. So, if you write a

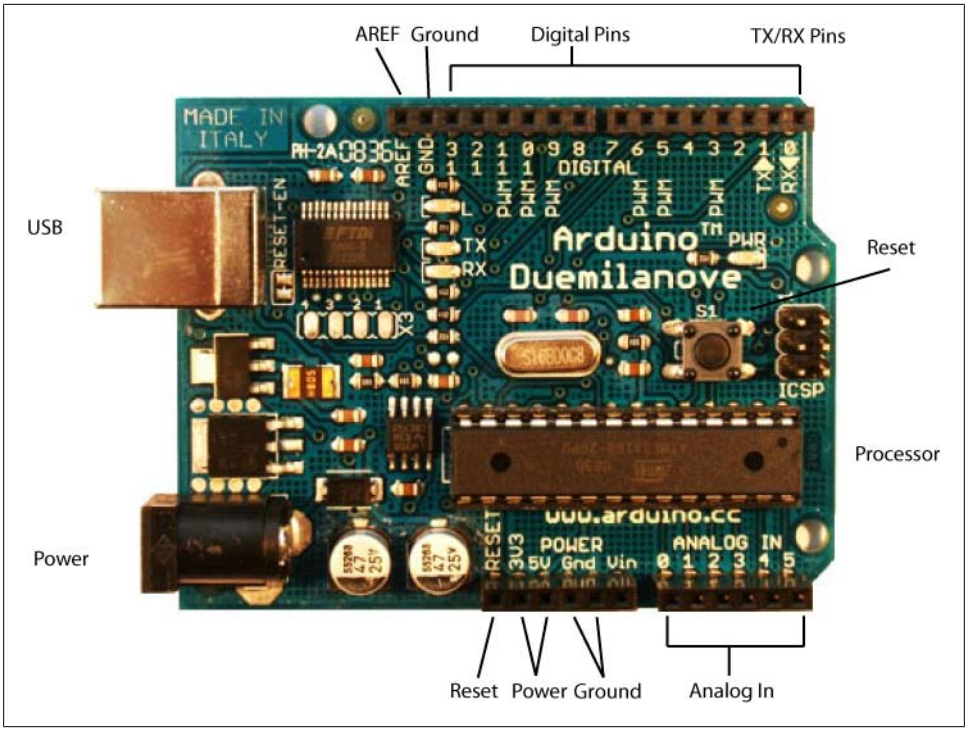


Figure 4-9. Arduino Duemilanove

program for your controller that is made to blink an LED on or off, it will begin running that program when you power up the Arduino. The reset button stops the program that is currently running. To change the program saved on the Arduino, you'll need to clear the memory by uploading a new program.

Just to the right of the reset button are the ICSP pins. These allow you to communicate using the I2C protocol, which allows for much more complex kinds of communication than the other ports. You can also use analog pins 4 and 5 to communicate over I2C. In Chapter 8, we'll be communicating with an accelerometer to detect pitch and acceleration over I2C. I2C also allows you to easily communicate many devices using your Arduino controller, up to 127. It's not easy, but it not impossible either.

The Arduino Mini (see Figure 4-10) was first introduced in 2006 and is a space-saving alternative to the Duemilanove. It can be so much smaller because it uses a different processor package, does not have the same easy access pins that the Duemilanove has, and requires a USB adapter that can communicate with the computer over USB. That said, the size of the controller means you can use it in projects where space is truly at a premium.

Pins on the Mini versus Duemilanove

When you turn over the Mini controller, you'll see a row of small copper pins that represent each pin. These function in the same way as the connectors on the Duemilanove, but whereas you can simply insert a wire into the Duemilanove, you'll need to insert the Mini into a prototyping board and then attach a wire to the same line on the board, as shown in Figure 4-10.

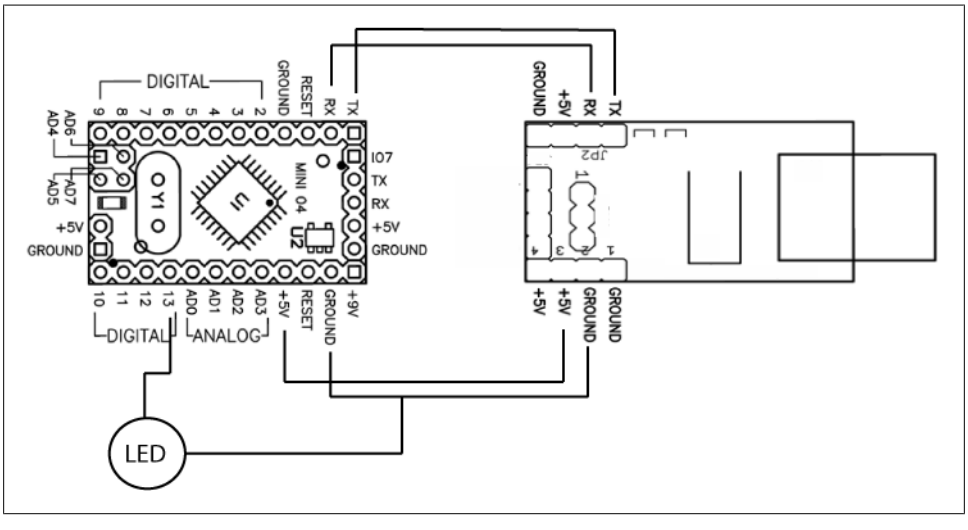


Figure 4-10. Connecting the Arduino Mini to the Programmer

The Mini has 14 digital pins, which, as you can see in the diagram in Figure 4-10, has 8 along the left side of the board and 4 more along the bottom of the right side. Since the pins are not marked on the Mini as they are on the Duemilanove, it's important to know which pin is which. Above the four digital pins on the right side of the controller are the four analog pins. Unlike the Duemilanove, all of these can be set to read data in or write data out. The other four analog pins (4–7) are at the bottom of the controller in a square-shaped formation.

Next to analog pins 4–7 are a pair of power pins, a ground pin, and a +5V pin to power components. Looking at the Mini, there are 3 +5V power pins and 4 ground pins, one on each side of the board. This makes the Mini even better suited for working in tight places, because grounding pins can be accessed from all sides of the controller, a real blessing when space is at a premium.

As of the writing of this book, there are eight other Arduino controllers:

Nano
: A compact board designed for breadboard use, the Nano connects to the computer using a Mini USB B cable.

Bluetooth
: The Arduino BT controller contains a Bluetooth module that allows for wireless communication and programming. It is compatible with Arduino shields.

LilyPad
: Designed for wearable application, this board can be sewn onto fabric and is a stylish purple.

Pro
: This board is designed for advanced users who want to leave a board embedded in a project. It's cheaper than a Duemilanove and easily powered by a battery, but it requires additional components and assembly.

Pro Mini
: Like the Pro, the Pro Mini is designed for advanced users requiring a low-cost, small board and willing to do some extra work.

Serial
: This is a basic board that uses RS232 as an interface to a computer for programming or communication. This board is easy enough to assemble—you can use it as a learning exercise.

Serial Single Sided
: This board is designed to be etched and assembled by hand. It is slightly larger than the Duemilanove, but it's still compatible with many accessories for the Duemilanove.

Mega
: This is a larger, more powerful Arduino board compatible with many accessories for the Duemilanove.

There are also numerous non-Arduino microcontroller boards that are quite similar to the Arduino: Freeduino, Sanguino, Bare Bones Board, LEDuino, and Miduino, among others. The nice thing is that once you've learned the basics of using one of these boards, you can apply that knowledge to using any of the others depending on your preference and the particulars of your project.

Touring the Arduino IDE

First, we'll take a look at the IDE, and then we'll look at the language that the Arduino controller uses. Looking around the IDE you'll notice that it's quite similar to the Processing IDE. This is no accident; the Processing project shared the code and design philosophy for the Arduino IDE, keeping a light, simple, and intuitive design with the IDE.

Before discussing the IDE, we'll explain how Arduino works. In Processing, for example, you compile and then run the code. Arduino adds an extra step to this: You compile the code, but the next step is to upload it to the Arduino controller so you can run it. Your computer doesn't run Arduino code, so you'll need to upload your code to a

controller that can run the code in order to test what you've written. As soon as code is uploaded to the Arduino controller, it runs right away, so the *run* step of the Processing workflow is entirely removed. Now, take a look at the buttons at the top of the controller (see Figure 4-11).

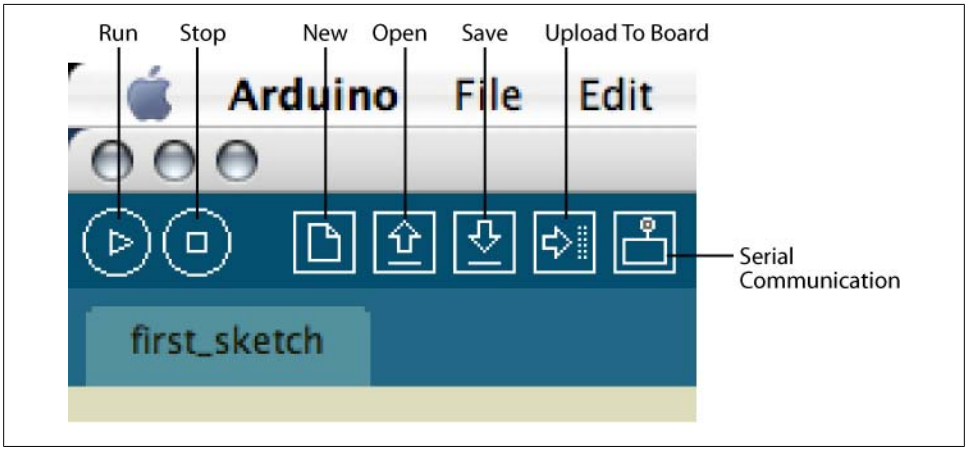


Figure 4-11. Controls of the Arduino IDE

Clicking the Run button does not in fact run your code; it checks for errors and compiles your code. Still, when you're writing code, it's quite handy to know what your errors are before you've uploaded anything to the board. The Stop button stops the IDE from listening on the Serial port. There's more information on how this works in the section "Debugging Your Application" later in this chapter. The New button simply creates a a new application but, note that it doesn't save it automatically. The Open button opens a project from your local machine. The Save button saves your project. The Upload to Board button actually uploads your code to the board, assuming that the board is properly connected and all the drivers are properly installed. Finally, the last button on the right opens the Serial Monitor window. This is quite important if you want feedback from the controller. Other than what the controller itself is doing obviously, you'll want to view the serial monitor on occasion.

The Sketch menu of the toolbar (shown in Figure 4-12) contains some duplicates of the functionality in the IDE controls but also contains some other interesting options.

The Import Library option allows you to import functionality from a library created for a specific purpose, for instance, working with Motors or I2C communication. These can be either the default libraries that come with Arduino or a library that you have created yourself. If you select the Stepper library, for example, you'll see the following line appear in the code window:

```
#include <Stepper.h>
```

This imports the Stepper library into the project, making all of its functionality available to your application. You'll find more information on importing libraries and

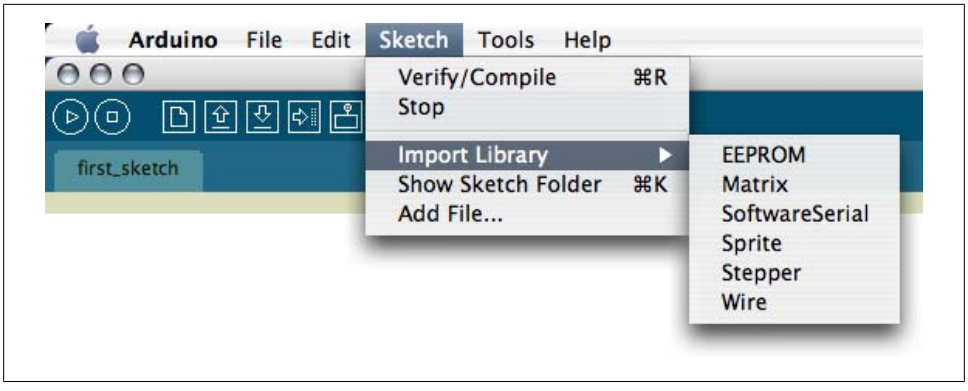


Figure 4-12. The Sketch menu options

functionality later in this chapter. The Show Sketch Folder option is a shortcut that brings up the folder in your operating system (OS) where the application files are stored. This is helpful if you're storing other files with your application or need to copy or change a file in the file system. Finally, the Add File option allows you to select a file easily from anywhere in your operating system and save it to the folder where your application is located.

The Tools menu (shown in Figure 4-13) contains menu buttons for selecting the controller and port on which the controller is connected to your computer. Additionally, this menu item contains the Auto Format option, which formats all your code to standardize the indentations and spacing. It also contains a Copy for Forum option, which copies all the code in an application to the system clipboard of your computer in an HTML-ready format so that it can be pasted into a web page without losing its formatting. The Archive Sketch option simply creates a *.zip* file of your application. Finally, the Burn Bootloader option is a rather advanced topic. Suffice to say that you won't need to burn a bootloader with any of the controllers that we'll be looking at unless you're building your own board.

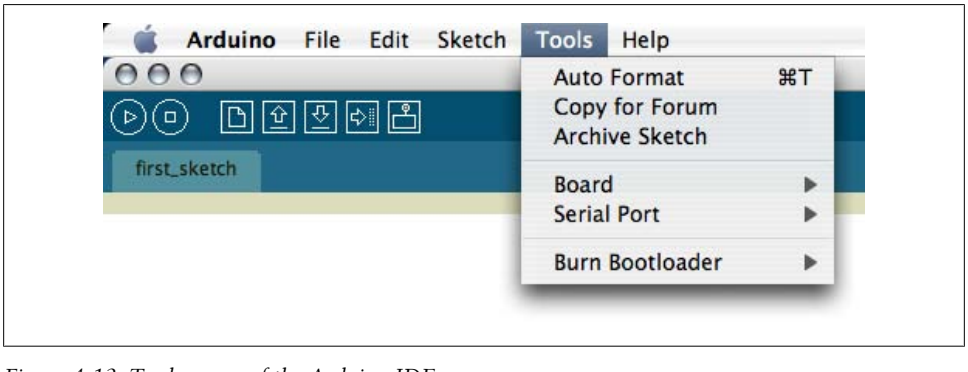


Figure 4-13. Tools menu of the Arduino IDE

So, now that we've examined the basics, let's start writing some code. The code window is the main window of the IDE, which is where all your code will go. Once you've written something, you'll want to compile the code, which you can do by clicking the Run button. Give the following code a spin:

```
void setup() {
    pinMode(13, OUTPUT);      // sets digital pin 13 as output
}

void loop() {
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
    delay(500);
}
```

For those of you with Duemilanove controllers, if there are no errors, you can click the button to upload the code to your board. You should see the TX and RX lights on your controller blink as the code is being uploaded and the memory for the program initialized, and you'll then see a message displayed in the console window of the IDE that says something like this:

```
Binary sketch size: 1092 bytes (of a 14336 byte maximum)
```

If that works, you'll see the LED on your board start blinking, and you're ready for the next section. If it doesn't, your controller is probably not connected properly. Check the settings for your controller and the port in the Arduino IDE, and try again.

For those of you with Mini controllers, skip ahead to the section "How to Connect Things to Your Board" on page 115 if you want to see something blink, and then return to this section. Why is this? Well, you still need to wire the USB Adapter and the Mini together. It's not too difficult, but there are a few little things that can go wrong.

The Basics of an Arduino Application

We're going to jump into code now rather than wiring for one simple reason: you've already seen code, so what you learned in the previous chapter will help you understand the new things in this chapter. In fact, the Arduino language is structured quite similarly to the Processing language with regard to how the application itself is structured. There is a `setup()` statement, and code within that statement runs once when the application first starts up. Then there is `loop()` statement which runs over and over again. Almost all Arduino applications consist of the declaration of some variables that will be needed throughout the lifetime of the application, one-time initialization to set things up ready to run, and functions to use in the main loop.

The setup Statement

The `setup` statement is the first thing called in the Arduino application. As in a Processing application, this is where things that need to be initialized should be placed. For instance, if you're going to use the Serial port for debugging your application, you'll need to start the Serial connection using the `Serial.begin()` method in the `setup()` method. Some devices need to be initialized when the microcontroller starts up; other devices need to have a signal sent to them just after the device starts up but before it's used. These are the sorts of tasks that should be handled in the `setup` statement. All applications must have a `setup()` method, even if nothing is done in them. This is because the compiler will check for this method, and if it isn't defined, an error will occur.

The loop Method

The `loop()` method contains anything that needs to happen repeatedly in the application; that could be checking for a new value from a control, sending information to a computer, sending a signal to a pin, or sending debug information. Any instructions in this method will be run repeatedly until the application is terminated. Usually we want to check any values that might be input from a control or another program, determine what to do with the input, and then send any information or instructions to any other components or programs that might need them. We'll look at two examples, one simple and one more complex.

In Figure 4-14, you'll see the organization of the code for a simple program to turn a light on and off when a button is pressed. There are three distinct elements to this program:

`initialization`
: This element contains all the variables and values that will be used throughout the program. Recall that if a variable is declared within a method, then it exists only within that method. This is called *variable scope*. If this doesn't sound familiar, take a look back at Chapter 2.

`setup`
: This element contains the code to configure the pin for the button to receive information from a control and set the pin for the light to send information.

`loop`
: This element contains the code to check the value of the button. If it is sending a LOW signal, that is, if the button is pressed, then send a signal to the light to turn on; otherwise, it tells the light to turn off.

Figure 4-14 shows how an Arduino application breaks these elements down.

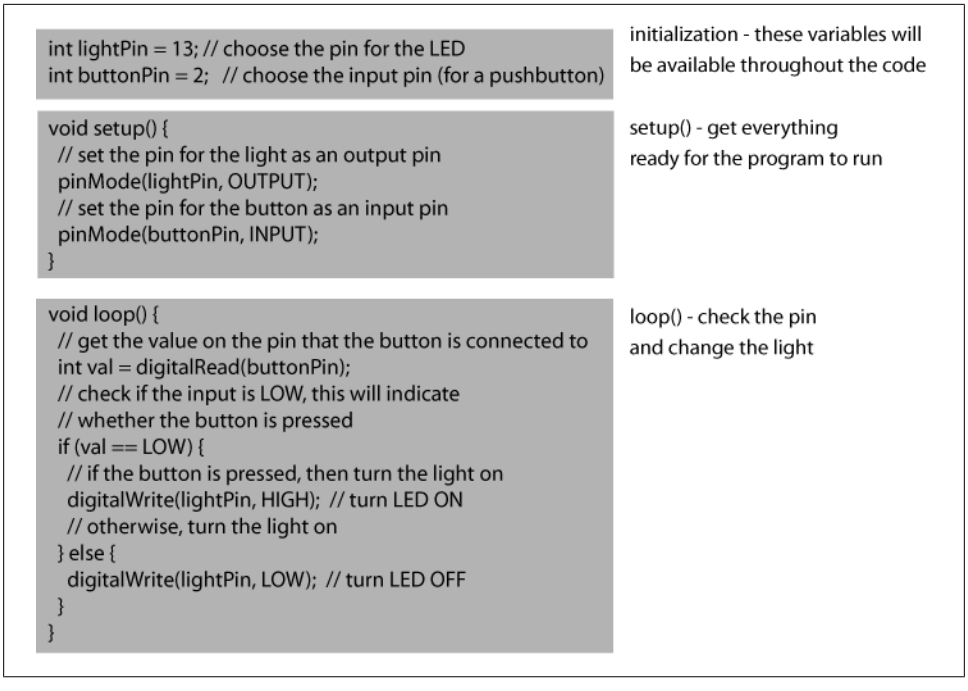


Figure 4-14. The organization of a simple Arduino application

Another, slightly more complex example is to use a potentiometer (more commonly but less accurately called a dial) to set the position of a stepper motor. Stepper motors are covered in much greater detail later in this chapter and in Chapter 11, but for the moment, think of a stepper motor as being a motor that has distinct positions that it can be placed at, rather than a simple motor like a fan that simply spins. To use a stepper motor in Arduino, import the `Stepper` library into the application. Once again, you'll learn more about importing libraries into an Arduino application later in this chapter; for the moment, we want more to familiarize you with the structure of an Arduino application. So, our steps for this application will be as follows:

1. `import`: before you do anything else, you need to import any libraries that you'll use in the application.
2. `initialization`: this initializes the variables, which in this case includes the `Stepper` object that you'll use to handle positioning the stepper motor correctly.
3. `setup`: this sets the speed of the `Stepper` object.
4. `loop`: this checks the value of the dial and sets the position of the stepper motor accordingly.

Figure 4-15 shows the organization of the Arduino application.

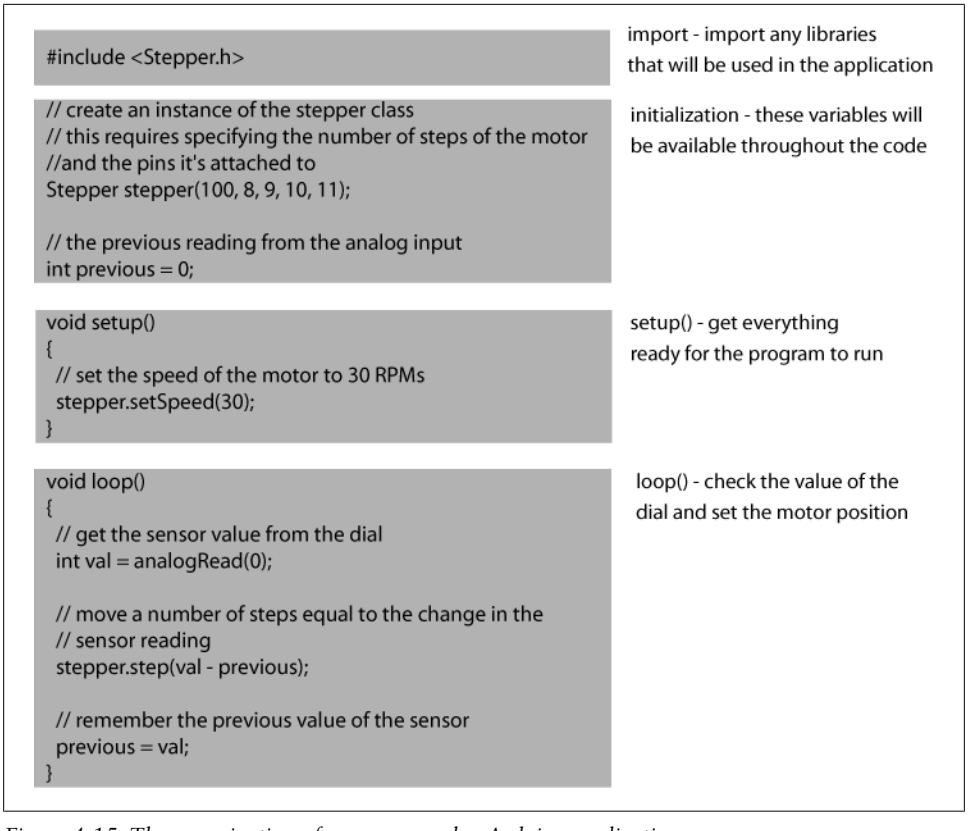


Figure 4-15. The organization of a more complex Arduino application

Features of the Arduino Language

The Arduino language is designed to support communication with electronic components. It works in a similar way to Processing, but it is structured a little bit differently because it deals with different problems. When dealing with electrical components, you are usually sending information as voltage or, in the case of I2C, sending communication in binary strings. Still, these forms of communication are at the very root and core of computing; they are, to use two very geeky terms, *low level* and *close to the metal*. When one sends an instruction as a pattern of electrical bursts, they're really working at the level that electrical components speak rather than asking a *higher-level* or more abstract programming language to do it for them.

The language that Arduino uses is built on the C language. Now, a quick word on C is relevant here because it's important to understand what C is and how Arduino is and isn't like C. C is an old programming language that makes it very well suited for our purposes because it was designed back when computing memory and processing power were at a premium. This means that while C may not be friendly to read and write for

the new programmer, it certainly is stingy with its resources, making it perfect for doing things on a microcontroller where resources are more limited than on a desktop machine or laptop. The Arduino language has a relationship with C somewhat similar to the relationship that the Processing language has with Java. If you understand C, you'll understand a lot of Arduino, and conversely, if you understand Arduino, you'll understand *some* C. Now, it's important to note that Arduino is not quite C, which is to say that if you find some neat C code somewhere that you want to run, pasting it into the Arduino IDE and expecting it to work might lead to disappointment. You can do this, but it requires some planning and some serious knowledge of what you're doing.

C is a general-purpose, block-structured, procedural imperative computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories to use with the Unix operating system. It was named C because many of its features were derived from an earlier language called B. Compilers, libraries, and interpreters of other higher-level languages are often implemented in C.

The basics of the Arduino programming language are similar to C++ and Processing. Methods are defined with a return type and any parameters like so:

```
return methodName(params...) {}
```

Variables are defined like so:

```
variableType variableName;
```

Some of the other features of the Arduino language aren't going to be so immediately familiar, but the next few paragraphs will help explain some of the more important.

Constants

The Arduino language has the following constant variables that can be used throughout any Arduino application:

`true/false`
: You can always use `true` and `false` just as they are. These are predefined, never change, and are always available:

```
if(variable == true) {
doSomething();
} else {
doSomethingElse();
}
```

`HIGH/LOW`
: These define the voltage level on a digital pin, either 5V or 0V. These make your code more readable:

```
digitalWrite(13, HIGH);
```

`INPUT/OUTPUT`
: These are constants used for setting pins that can be used either for output or for input:

```
pinMode(11, OUTPUT);
```

As you'll notice, the `OUTPUT` constant is used to set the value of the pin using the `pinMode()` method. This seems like a good place to segue into the core methods of the Arduino language.

Methods

Now you should learn about some of the methods of the Arduino language:

`pinMode(pinNumber, mode)`
: Remember that the digital pins of the Arduino controller can be set to either input or output, which is to say that they'll send values to a controller or receive values from a controller. Before we use a digital pin, though, we need to establish in which direction the information on that controller will be flowing. Usually you do this in the `setup()` method, because you'll rarely need to do it more than once in an application.

`digitalWrite(value)`
: This method sets a digital pin to `HIGH` if value is high or `LOW` if value is low, which is to say that it sends 5V or 0V through the pin. This can work only on pins that have been set to `OUTPUT` using `pinMode()`. For example:

```
pinMode(11, OUTPUT);
digitalWrite(11, HIGH);
```

If the `pinMode()` method hasn't been used to set the pin to be `OUTPUT`, then calling `digitalWrite` will have no effect.

`int digitalRead(pinNumber)`
: This method reads the state of a pin that is in input mode. This can be either `HIGH` or `LOW`, that is, 5V or 0V, and no other value. The `digitalRead` method is used for reading buttons, switches, anything that has a simple on and off, and any control that returns a simple `true` or `false`, `1` or `0`, or other type of binary value. It returns the value of the pin read as an integer.

`analogRead(pinNumber)`
: This reads the value of an analog pin, returning a range from 0 to 1,023, that represents the voltage being passed into the pin. As you'll discover later, it is important that any controllers you connect to your Arduino controller send analog signals in the range between 0 and 5V because higher values will not be read and could damage the board. This method returns an integer and so can be used as shown here:

```
int analogVal = analogRead(11);
```

This code snippet uses the value returned from the `analogRead` method and stores it in the `analogVal` integer. This is a common technique that allows you to retrieve the value from a pin and use that value later in your code.

`analogWrite(pin,value)`
: This writes an analog value to a pin and can be any value between 0 and 255. This means that the value sent to this method can be used as an adjustable voltage and sent through the pin to any controller on the side:

```
analogWrite(11, 122);
```

`delay(ms)`
: This method tells the program to wait for a given number of milliseconds before executing the next instruction. In practice, this means that in the following lines of code:

```
digitalWrite(13, HIGH);
delay(1000);
digitalWrite(13, LOW);
```

there will be a delay of one full second between when the first instruction and the third instruction are executed. In practice, this is often used for timing, such as controlling how long, for example, a light stays lit.

`millis()`
: This returns the number of milliseconds since the program started running. This can be useful if you need to keep track of time, which you can do by storing the result of the `millis` call and then comparing with another `millis` call at some later point, as shown here:

```
long timer = 0;
void setup() {
    timer = millis();// get the timer the first time
}
void loop() {
    int lengthOfALoop = millis() - timer; // compare it
    timer = millis(); // now set the timer variable again
}
```

Of course, you might want to do something more interesting with `millis`, but it's a start.

Arrays

Arduino treats arrays in much the same way as C++ and somewhat like Processing, though you cannot use the constructor with an array as with Processing. The same rules about what can be stored in the array apply:

```
int intArray[10];
intArray[4] = 4;
intArray[3] = "hello!"; // doesn't work!
```

You can also declare and initialize an array at the same time:

```
int preinitialized[3] = {1, 2, 3};
```

Strings

Working with Strings in Arduino is different from working with Strings in Processing or openFrameworks because the Arduino language doesn't use a `String` class like C++ and Processing. This might seem like a big deal at first, but you'll realize after working with Arduino for a little while that the types of things that you do with Arduino don't require strings or text as frequently. If you do need to work with a `String` class, you need to make an array of characters, as shown here:

```
char name[5] = {'j', 'o', 's', 'h', 0};
or
char name[] = "josh";
```

If you need to, for example, store multiple strings in an array, you can create an array with multiple `char` arrays stored within it:

```
char* multichar[4];
char name1[5] = {'j', 'o', 's', 'h',0};
char name2[5] ={'t', 'o', 'd', 'd',0};
or
char * multichar[] = {"josh", "todd"};
```

OK, so what's going on here? You'll notice that there's a little pointer magic being used to store the arrays. Pointers are covered in greater detail in Chapter 5. When you declare the array, you're declaring that it will store pointers to char variables. Though this may seem a little strange at first, it isn't hard to get used to, and generally, in Arduino, you're not dealing very extensively in strings.

Interview: David Cuartielles

David Cuartielles was one of the originators of the Arduino project along with Massimo Banzi. He currently teaches and mentors at the University of Malmo in Sweden. You can find more information about his work and thoughts at *www.0j0.org*.

What sorts of things are you most excited about working on?

David Cuartielles: We take projects that we consider challenging. We could say that we don't do this for the money but for the pleasure of making things the best way possible. We are in a constant learning process. For example, today I came to Seoul (Korea) to teach about generative music using Arduino boards, a topic I have been researching for the last months, just for fun. Now it is time to see if this research was worth the effort!

Can you explain the history of the Arduino controller and what the original motivations for it were?

David: For the individual people in the team, the motivations were a little different. To me, the important thing was to create an educational tool that would be so cheap that people could afford giving it away inside their pieces.

We made it in a couple of days of hacking during after-work hours at Interaction Design Institute Ivrea (IDII) in Italy. Massimo Banzi and I were discussing the need of something like this. A couple of his students joined the discussion, and one of them, David Mellis, started to craft the software in a very neat way, wrapping functionality together in a way that was very easy for people without a true technical background to understand. We were trying to create a new way to prototype and a new way to think about prototyping things.

How did you begin working with electronics and coding?

David: I burned my first computer at the age of 9. It was a Commodore 64, and I was trying to make an LED screen. It was fun and sad at the same time. It took a couple of months until I could get replacement parts to fix it. I use all the Arduino controllers extensively; it is very powerful when used in, I guess, "expert mode." I like to code aloud, letting people around me look at the code while I write it...I guess that is why I became a teacher.

You run a toy-making company (according to a bio I found of you online). Can you talk a little bit about making toys and how you began doing that and what sorts of toys you find yourself making.

David: Right now we are focusing on sound. Everything you see around has the shape of a keyboard, as if we didn't know how to play with any other tools. Our first toy collection is dealing with this issue. At the same time, we are documenting all our processes, and we are trying to make the first open source toy collection in the world.

What is the philosophy of open hardware? How is it different/same as the open source software philosophy?

David: To me the only difference is that one of those two happens to have a physical representation. Hardware costs money beyond the personal effort of creation. Software can be given completely for free; hardware cannot.

You do a lot of workshops and teaching. What sorts of things do you try to focus on with beginning students?

David: I focus on learning. Many people come to me with a motor in one hand or a screen or whatever weird gadget and ask, "Can I hook this up to Arduino?" Then I answer, "Yes, what for?" I don't fancy technology. To me, it is just a way to understand more about our current way of living.

For me, it is more important that people understand how it works than them making the fanciest project in the world. On the other hand, once people understand, it becomes much easier to make fun projects.

This is a very broad question, so feel free to answer however you would like: How are interactions developed? What sorts of processes do you go through from envisioning a project to creating a rough prototype to creating a finished piece?

David: This is a lot of value in knowing what may or not work while developing a piece. However, many times we give a try to things we don't fully believe in just because we can try them easily, and we can clarify our potential doubts by actually physically interacting with the object.

To me, the idea of "designing an interaction" is based either in the concept of aesthetically pleasant interaction or, sometimes, in the more functionalist of meaningful interaction. The issue for me is, "Is it really possible to anticipate the audience's feelings toward whatever object or experience we produce?" Shall we then "design the interaction" or just make something that is pleasant for us?

You very often work with other designers, engineers, and artists. Can you talk a little bit about how you find it best to work with others on a particular art piece? Is this different from how the core Arduino team has collaborated on the design of the various Arduino controllers?

David: Yes, with Arduino we have kind of a higher mission than making just one piece. We want to help people make their own pieces, and that requires an extra effort in documentation and community building. Working on making a piece for/together with someone means solving technical issues and accommodating as much as possible the project's expectations. Making Arduino means trying to set up a quality standard in how to speak about things.

How do you see the relationship between industrial design and engineering and fine arts and more visual design changing now and also in the next few years?

David: Right now we are experiencing a revival of the craft, not only in design but also in the arts. Conceptual art is being left behind, and artists are going back to the galleries bringing illustrations and beautifying our lives with objects. It is a cycle that in my opinion was a logical step to take after postmodernism. Visual design (understood as graphic design) is in my eyes a very powerful tool for artists to express their ideas and point controversial topics. Illustration is a technique to do so.

Industrial design and engineering are experiencing some kind of similar romance. Now we are seeing new tools that allow industrial designers to jump from conceptualizing to actually crafting the objects to a certain extent. They don't just do boxes and let us imagine the rest; they prototype the interaction and start to give extra thoughts to the physical affordances of objects in terms of how they will be used as digital artifacts.

Do you see a difference between "interactive art" and "interactive design"?

David: Yes and no. I jump from one to the other constantly, because I come from the line of action of "critical design" where we create objects to express existential conflicts. This ends up being very close to art. On the other hand, during the last years, I have gotten to know a lot of people coming from the conceptual arts and moving toward electronic art, and their theoretical framework is so much heavier than mine. There is where I think that art and design differ a little. Design has sometimes that quality of not needing to express anything, and just entertain instead.

What do you see as the most exciting areas or themes of exploration in physical computing?

David: Now that we have both Wii Remote and iPhone, anything else is rediscovering the wheel…just kidding, of course. I think there is a lot to do. The field of *haptics* (touch-based feedback) is still in an early stage. There is no common language for physical programming of objects. We have to learn about new smart materials—not just textiles, but all sort of intelligent plastics, and so on.

As I see it, considering the state of the art of technology, projects are constructed from either physical computing objects with embedded intelligence or computer vision installations, or hybrids. Knowing about one of the fields is a must for interaction designers. Many times there are two ways of solving the same issue, and those two ways can be framed within these two categories.

What are some of the projects or designs that you've seen that you feel best embody the idea of an interactive device?

David: I love the Wii Remote because it has everything people have been discussing physical computing should be. It is nothing but a magic wand, a link to the infrastructure that does almost nothing by itself, but it has a very good cause-effect relationship (thanks to the well-crafted software). I also like that it is hackable; it is a property of digital technologies to be hackable, modifiable, and copyable.

On the other hand, there is a project by some of my students called *Table Talk* that anyone can find on YouTube (created by Marcus Ericsson et al.; see *http://tabletalk.se/*); it shows the idea of subtle interaction that happens without having to jump on things or dance or clap your hands. It is the total opposite to the Wii Remote, and it best expresses this second category of devices that do things without demanding users to be crazily active.

How to Connect Things to Your Board

Of course, the one thing that we haven't talked about much yet is how to actually connect things to your board. To connect a component to your board, you need to create an electrical connection between the correct pin on your board and the correct corresponding pin on the component. There are two ways of getting this done: using a project board and solder or using a prototyping board.

Soldering should be used once your designs are really ready. Prototyping boards save time and solder because they allow you to fix mistakes easily. Just pop the wires out, and connect them in the right place; it's much easier than needing to undo a bunch of solder and resolder it all. There are lots of situations in electronics where you don't want to use a prototyping board because they can allow stray current to leak around places you might not want it, but since the controller of the Arduino is already somewhat protected, a breadboard is fine to use. Figure 4-16 shows a prototyping board.

To connect the controller to the board, you can use either solid 24-gauge wire or jumper wires like the kind shown in Figure 4-17.

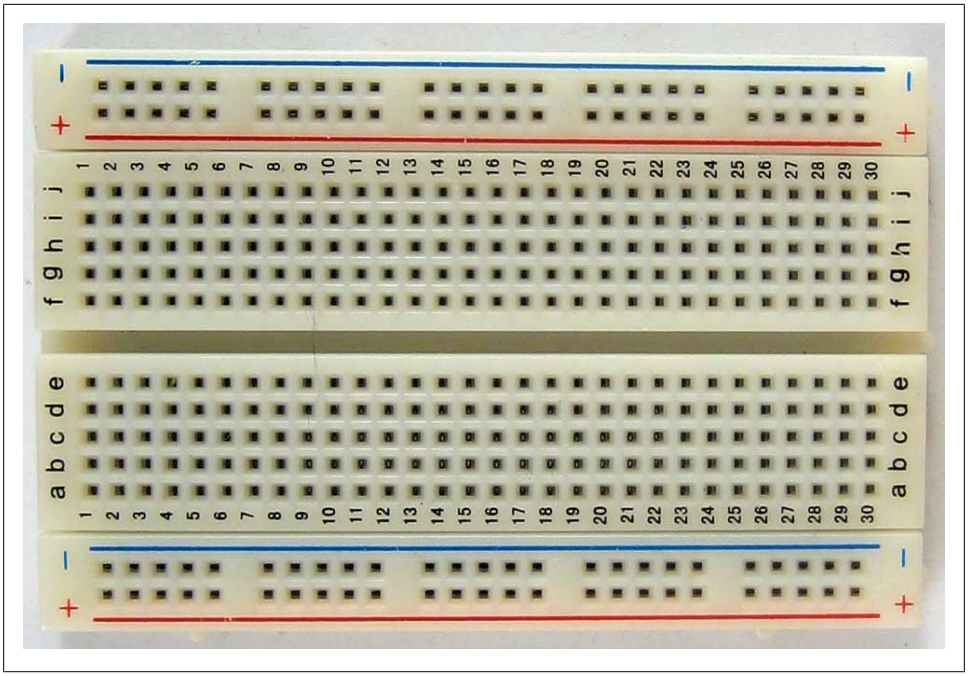


Figure 4-16. A breadboard or prototyping board

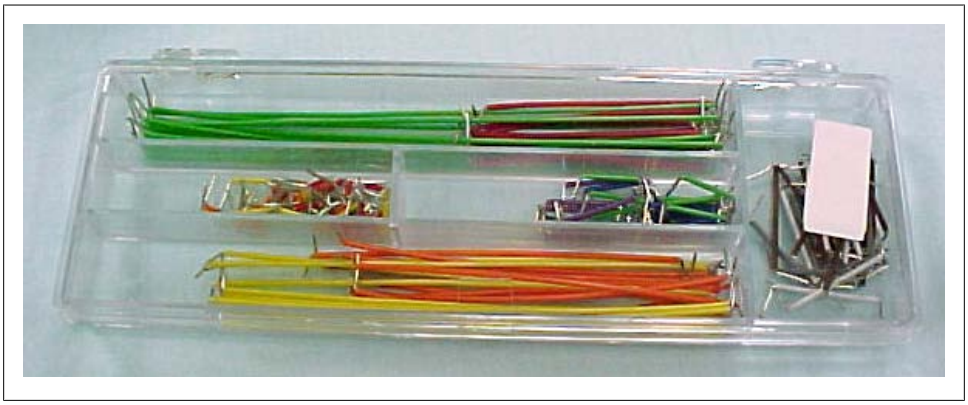


Figure 4-17. A box of jumper wires

Both the prototyping board and the wires can be secured from an electronics store and will make your Arduino experimenting life a lot easier. A prototyping board has two different directions for its rows. The rows on the edge of the board travel the length of the board, as you can see in Figure 4-18.

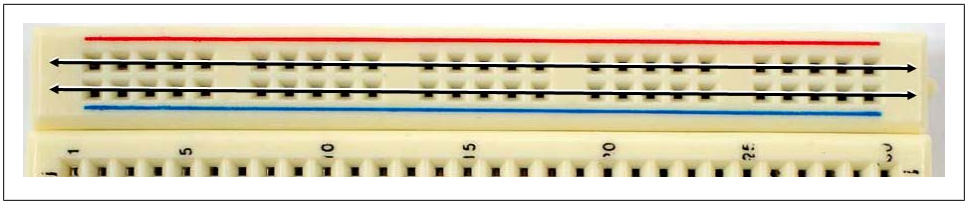


Figure 4-18. Top of a breadboard with lengthwise rows

The top two rows are usually used with one row providing access to power for all components and the other providing a ground for all components on the board. The middle of the board is a little different, as shown in Figure 4-19.

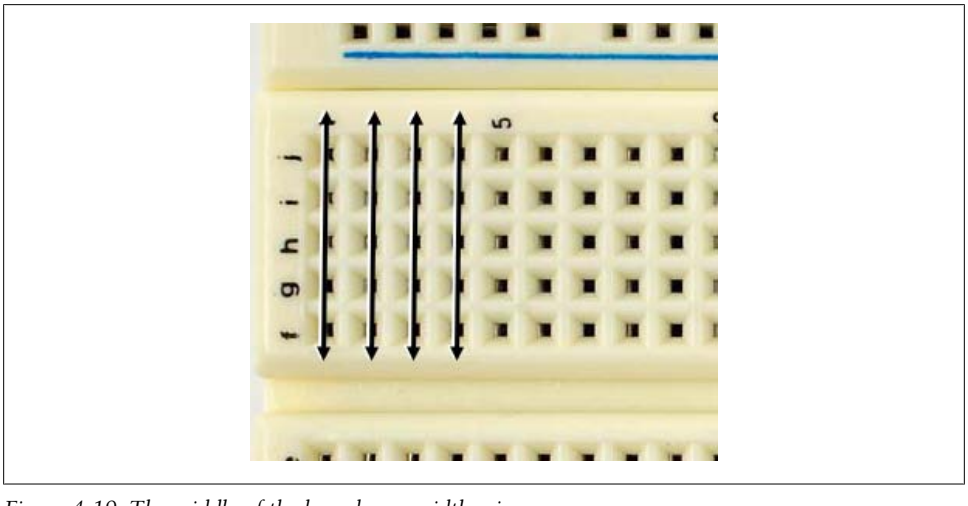


Figure 4-19. The middle of the board runs width-wise

The middle of the board is where you connect components and controllers together. Each row can be connected to the others using either 24-gauge wire or jumper wires. Connecting two rows together makes them share a circuit.

Once you have your design ready to go, you can easily pick up a drilled breadboard, some solder, and a soldering iron and get started making a more permanent version of your project. Until then, though, it's easier to use the prototyping board.

Hello World

The Hello World of the Arduino controller is a blinking light. For this to work, you'll need to do your first little bit of wiring with the Arduino controller, attaching an LED to pin 13 of your controller, as shown in Figure 4-20.

To attach the Mini and upload code, you'll want to configure it like Figure 4-21.

Figure 4-20. Diecimilia with LED attached

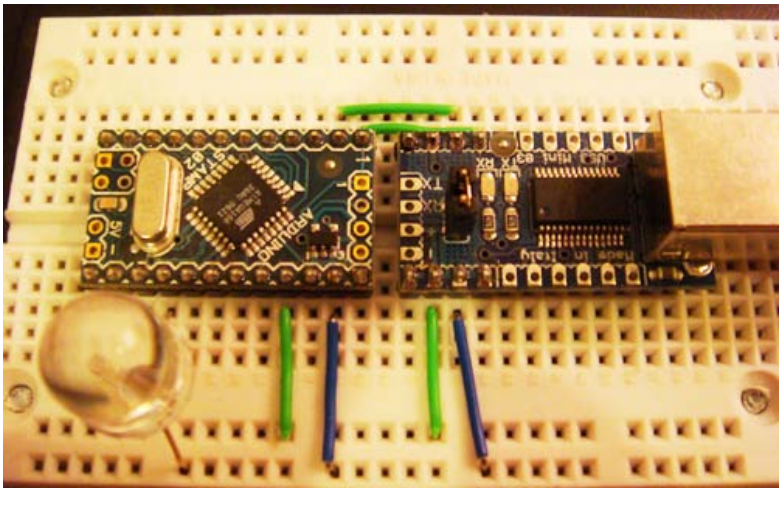


Figure 4-21. Connecting the Mini to the programmer

That might be a bit hard to follow what's going on; let's look at a schematic of it in Figure 4-22.

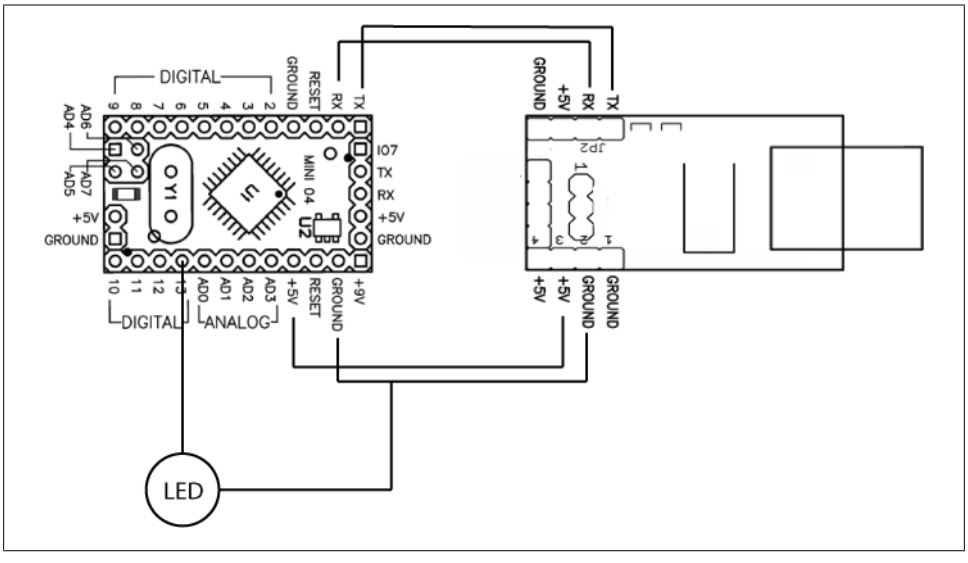


Figure 4-22. Connecting the Mini to the programmer part 2

Once you've attached the pin to the controller, put the following code in the code window of the IDE, and click Run. After the IDE has finished compiling the code, click Upload. It should take a few seconds to upload the code to the controller, and then you should see the LED blink on and then off in a somewhat syncopated rhythm:

```
int ledPin = 13;

void setup() // run once, when the application starts
{
    pinMode(ledPin, OUTPUT); // sets the pin where our LED is as an output
}

void loop() // run over and over again
{
    digitalWrite(ledPin, HIGH);   // sets the LED on
    delay(1000);                  // waits for a second
    digitalWrite(ledPin, LOW);    // sets the LED off
    delay(500);                   // waits for a half second
}
```

That's not too exciting yet, is it? In the spirit of interactivity, let's give you some control over that LED. We're going to use the `INPUT` mode of the pin that a button is connected to so we can send a signal and control the LED with the button:

```
int ledPin= 13; // choose the pin for the LED
int buttonPin = 10;   // choose the input pin (for a pushbutton)

void setup() {
    // set the pin for the light as an output pin
    pinMode(ledPin, OUTPUT);
    // set the pin for the button as an input pin
    pinMode(buttonPin, INPUT);
}

void loop() {
    // get the value on the pin that the button is connected to
    int val = digitalRead(buttonPin);
    // check if the input is LOW, this will indicate
    // whether the button is pressed
    if (val == 0) {
        // if the button is pressed, then turn the light on
        digitalWrite(lightPin, HIGH);  // turn LED ON
        // otherwise, turn the light on
    } else {
        digitalWrite(lightPin, LOW);  // turn LED OFF
    }
}
```

To get this to work properly, you'll need to add a new element to the wiring scheme: the resistor. A *resistor* is aptly named because what it does is resist the flow of electricity. Resistors are used to control the flow of current. Current can move either way through a resistor, so it doesn't matter which way they're connected in a circuit. To read the button, what we'll do is send 5V through a resistor over to the button and allow a button press to complete that circuit. When the circuit is completed, flowing all the way through, the value from pin 10 will read LOW, and the light will be given power to turn it on. With the Arduino Mini, your schematic will look like Figure 4-23.

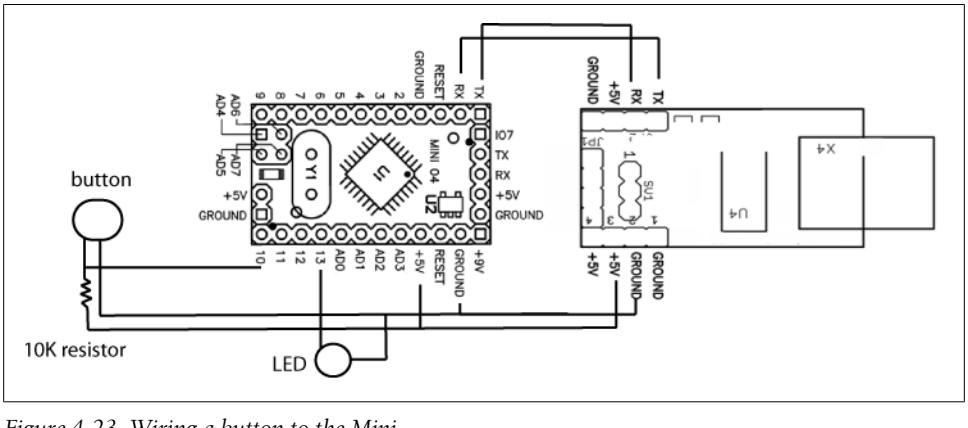


Figure 4-23. Wiring a button to the Mini

For the Duemilanove, you'll want to create the circuit in Figure 4-24.

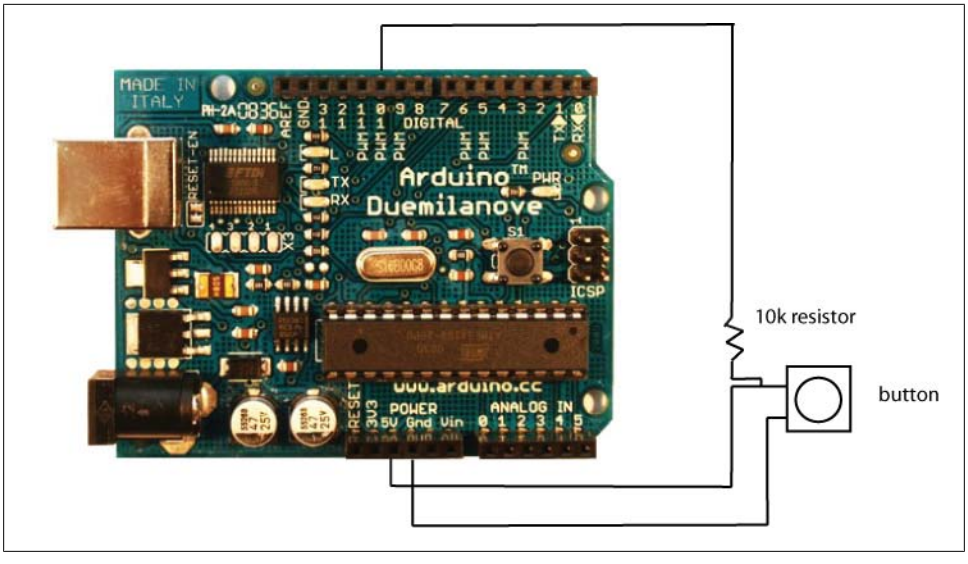


Figure 4-24. Wiring a button to the Decimilia

This is where using a breadboard pays off. If you're using the Mini, plug the +5V pin into one of the connection lines that runs the length of the board and a ground pin into the other connection line. Next, use the 10K resistor to create a connection between one pin of the button and the connection line that carries the 5V. The button will have two connection points, so one gets connected with the resistor and the other gets connected to the ground connection line. Then, connect the LED to the ground connection line and pin 13. Take a look at Figure 4-25.

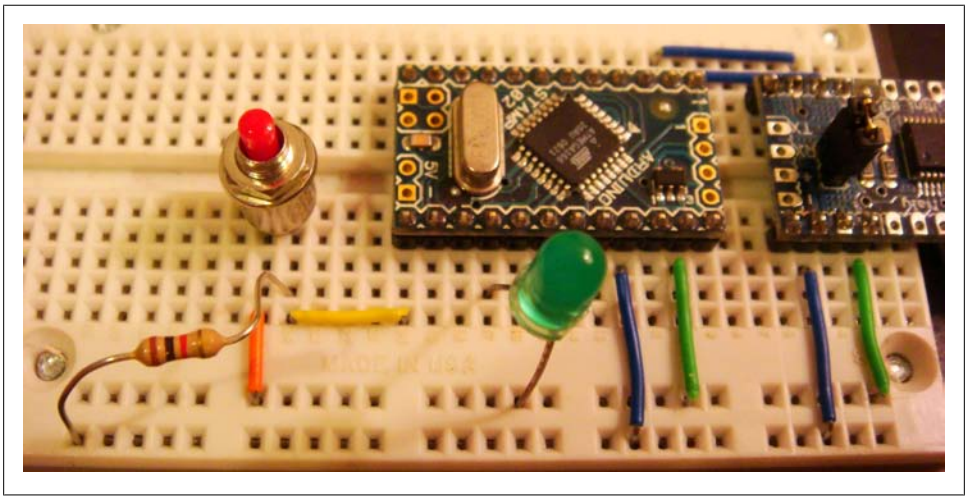


Figure 4-25. Wiring up the button

Even though Figure 4-25 shows the Mini version, the Duemilanove version is even simpler. Simply connect the resistor directly from the 5V to the same connection line on a breadboard as the button. Then, run a wire from that connection line to pin 10. Ground the other button connection by running a wire to the Decimiliar's ground pin. Plug the LED in, and you're off to the races.

To upload the code, simply click the Upload button, as shown in Figure 4-26.

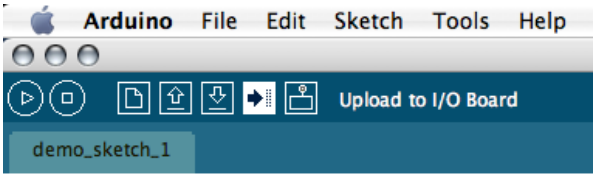


Figure 4-26. Uploading code to the Arduino board

You should see the LEDs on the board begin flashing, and you should see a message printed out in the console window. Your Arduino controller may take a second or two to begin running the code, but you'll be able to turn the light on and off.

Debugging Your Application

So, what do we do if we need to know the value being sent back from a sensor to a controller? Or, what if there's something going wrong that you can't quite sort out? You'll use the serial library to send messages, character strings actually, back from the controller to the computer so you can see what's happening. This lets you send messages to know that something is working or not working and to monitor the values being passed around in your application. Wherever you have some code that you need to monitor, you can use the `Serial` `print()` method to allow you to check the values of variables or the execution of code and see those messages in real time using the Serial window of the Arduino IDE.

Use the `begin()` method of the `Serial` class in the `setup()` method of your application to set the communications speed:

```
int speed = 9600;

void setup() {
    Serial.begin(speed)
}
```

This sets the data rate in bits per second (baud) for serial data transmission. For communicating with the computer, use one of these rates: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. The higher the number, the faster the Arduino controller will send messages to the computer but also the higher demand you'll make on your controller. Note, too, that the speed needs to be the same on the

sending and receiving, so if you're trying to read data on the Serial Monitor of your Arduino IDE, you'll need to make sure that you're using the same speed in the IDE as you've set in the `begin()` method. Once you've called the `begin()` method of the Serial, you can send data from your Arduino controller to your computer using the `print()` method:

```
Serial.print(data)
```

The `print()` method is more or less like the Processing `print()` method that you encountered in Chapter 3 and the `printf()` you'll encounter in Chapter 6, but it is optimized to save space for Arduino and so works slightly differently. By default, the `print()` method, with no format specified, prints an ASCII string. For example, the following:

```
int b = 79;
Serial.print(b);
```

prints the string `79` to the Serial Monitor window. The following little application will print the message every two seconds:

```
int i = 0;

void setup() {
    Serial.begin(9600);
}

void loop() {
    Serial.print(" hey there, just saying hi for the ");
    Serial.print(i);
    Serial.print("th time. \n");
    delay(2000);
    i++;
}
```

Once you've uploaded the application to the board, you start monitoring the output of the Serial by clicking the Serial Monitor button, as shown in Figure 4-27.

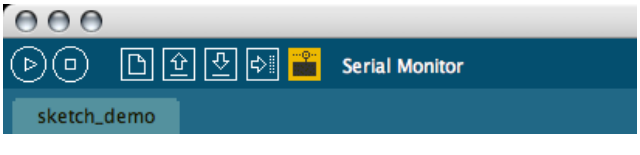


Figure 4-27. Starting the Serial Monitor

Once the Serial Monitor is started, you can see the output in the bottom of the Arduino IDE, as in Figure 4-28.

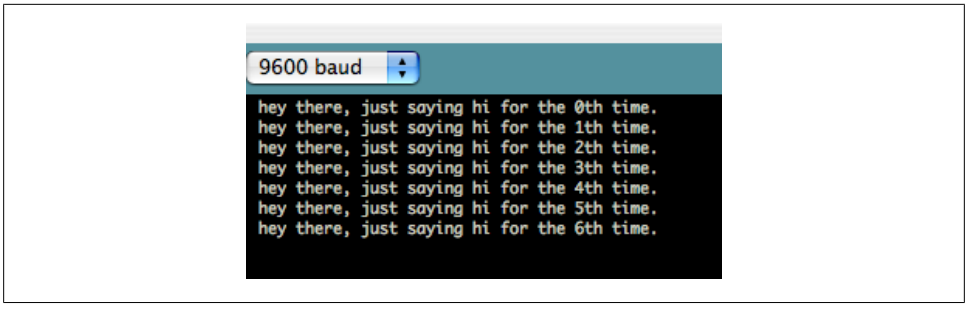


Figure 4-28. Monitoring the Serial output

You'll need to make sure that the baud rate selected in the drop-down list above the output window matches the rate that you passed to the Serial `begin()` method, or the Serial Monitor won't be listening at the same rate that the controller is sending.

You can also pass additional parameters to the `print()` method to get the output to be formatted a specific way:

```
int b = 79;
Serial.print(b, HEX);
```

This prints the string `4F`, which is the number 79 in hexadecimal. Table 4-1 lists most of the format specifiers for the Serial `print()` method.

Table 4-1. Output with the Serial print() method

Format specifier	Output
DEC	Prints the value as a decimal
HEX	Prints the value as a hexadecimal
OCT	Prints the value as a octal value
BIN	Prints the value as a binary number
BYTE	Prints the value as a single byte using an ASCII characters
No format specified	Prints the value as a string

Importing Libraries

There are many extremely useful libraries for the Arduino controller that help you do things like work with LED lights, control motors, use LCD screens, or even control certain kinds of cell phones with your Arduino controller.

For this section, we'll primarily use the Stepper library for demonstrations. The Stepper library comes with the Arduino download and allows you to reduce the number of connections required by a stepper motor from four to two. We won't get into why this is good and what this allows us to achieve in any great detail until Chapter 8, focusing instead here on how the library functions in the Arduino environment.

An Arduino library is generally made up of four parts:, an *.h* file, a *.cpp* file, a *keywords.txt* file that highlights Arduino library words in the IDE, and the actual binary library code. Arduino uses *.h* files to contain the information about a library, the variables it contains, the methods that it has, and so forth. The *.cpp* file contains the definitions for those methods and variables and is compiled into the binary library files used at runtime by the Arduino controller. For those of you experienced working with C or C++ already (which, after you read through the sections on openFrameworks, will be all of you), you'll recognize the *.h* file and what it does. The *.h* file is what is imported into the Arduino application, using the `#include` statement. If you want to know what is in a library, glancing at the *.h* file will tell you. For example, looking at the *Stepper.h* file, you'll see the following:

```
// constructors:
Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2);
Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2, ¬
    int motor_pin_3, int motor_pin_4);
// speed setter method:
void setSpeed(long whatSpeed);
// mover method:
void step(int number_of_steps);
int version(void);
```

This tells you that to create a new instance of the stepper, you pass it the number of steps, the pin that the first motor is connected to, and the pin to which the second motor is connected. Once you've created the stepper, you can call two methods on it: `setSpeed()` and `step()`. Learning to read *.h* files is a great asset in working with Arduino, openFrameworks, C, and C++, because it saves you the time of digging through documentation, and helps you get acquainted with the inner workings of the programming languages.

So, now that you've had a crash course in *.h* files, you'll look at importing some functionality. It's actually quite simple; you add the following line to the top of the application file:

```
#include <Stepper.h>
```

This will import all the stepper functionality, allowing you to create new `Stepper` objects and call their methods. For instance:

```
Stepper stepper(100, 8, 9, 10, 11); // note that we don't use = new Stepper();
void setup()
{
    // set the speed of the motor to 50 RPM
    stepper.setSpeed(50);
}
```

The Arduino download comes with a few core libraries that we'll outline in a rough way, just to give you an idea what's out there:

EEPROM

Depending on the board, the microcontroller on the Arduino board has 4 kilobytes, 1 kilobyte, or 512 bytes of EEPROM, which is permanent memory, like an extremely small little hard drive that you can save information to. So, how big is 512 bytes? It's 128 integers, or 512 boolean values, so it's not a lot, but it's enough.

So, you ask what you might do with it. You can save the last values input by a user, save the last state of a controller, save the highest reading ever for a component, save just a little tiny bit of GPS data, or store a URL or some other kind of valuable information.

Stepper

This library allows you to control stepper motors. To use it, you will need a stepper motor and the appropriate hardware to control it.

What you might use it for: making things that use stepper motors, of course, such as making small robot arms, rotating cameras, manipulating toys, pointing things, or driving things.

Wire

Two Wire Interface (TWI/I2C) is for sending and receiving data over a net of devices or sensors. This library allows you to communicate with I2C/TWI devices. On the Arduino, SDA (data line) is on analog input pin 4, and SCL (clock line) is on analog input pin 5.

So, you ask what you might use it for. You can read GPS sensors, read from RFID readers, control accelerometers, and in fact do many of the things that we're going to do later in this book.

Running Your Code

Once you have loaded your program onto the board, it is saved in the program memory of the board whether the board is being powered or not. There's no worry about the controller *forgetting* its program. When the controller starts up again, it will automatically run the last program that it had loaded into it.

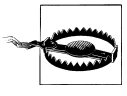
You can clear the RAM memory of the board either by grounding the reset pin, which works on the Duemilanove and the Mini, or by simply pressing the reset button, which works only on the Duemilanove. This restarts the board and the program that is saved on the board, but doesn't clear the program. That will happen only when a new program is uploaded to the board.

Running Your Board Without a USB Connection

There are two ways to run your Arduino, both of which are available to you with the Duemilanove controller, and one of which is available to you with the Mini controller. The first is to plug the controller directly into a wall using a 9V or 12V DC adapter, and

this can be done only with the Duemilanove. Most electronics supply shops will have one of these adapters with the correct attachment end that will fit in the Duemilanove DC port.

The other option, which is available on both controllers, is to simply use a 9V battery with a snap-on cap that allows you to get two wires from the battery to send to the controller. On the Mini, attach the positive lead (the red wire) of the battery cap to the +9V pin and the ground lead (the black wire) of the battery cap to the ground pin. On the Duemilanove, attach the positive lead to the 9V Voltage In (Vin) pin and the ground lead to the ground pin. You're now wireless and free to wander around with your Arduino.



When connecting to a power source, be careful. You can fry your controller by sending voltage to the wrong pin. If working with a battery, always connect the ground first and the positive lead later.

All right, you've seen the beginnings of the Arduino; now you can either flip ahead to Chapter 8, where you will work with the Arduino to provide some real interactivity, or you can read the next chapter, which delves a little deeper into programming to get you ready for openFrameworks and C++.

Review

The Arduino project is several different components: a microcontroller board, a simplified version of C and C++ for nonengineers to use in programming their microcontroller, and an IDE and compiler used to create and load code onto the microcontroller board.

To get started with the Arduino, you'll need to download the Arduino IDE and compiler from the *http://arduino.cc* website and purchase one of boards from one of the online suppliers.

Once you have the board and the IDE ready, you'll need to configure the IDE correctly for your board by plugging the board in and then, in the IDE, going to the Tools menu, selecting the board option, and selecting the board that you're using.

You can run a simple blinking light script to ensure that your board is working properly.

Pins can be either input, which means they will read values from a control, or output values to a control. Communication between the Arduino and any controls attached to it is done via the amount of voltage being sent either to or from the Arduino. Arduino controllers have two kinds of pins: analog pins that can read and write analog values, from 0 to 5V, and digital pins that simply read or write binary values represented as 0 volts or 5 volts.

Running an Arduino program has two steps: the first is to compile the code on your computer in the Arduino IDE. You can do this by pressing Ctrl+R (⌘-R on OS X). Once your code compiles, you upload it to the Arduino board using the Upload to I/O Board button on the Arduino IDE or by pressing Ctrl+U (⌘-U on OS X).

An Arduino application has two required methods: the `setup()` method that is called once when the application first starts up, and the `loop()` method that is called over and over.

To attach an LED to your Arduino, connect the first lead of the LED to a resistor, connect the resistor to any digital pin, and connect the second lead of the LED to Ground. Once you use the `digitalWrite()` method to set that pin to `HIGH`, the LED will light up.

To connect a button to your Arduino, attach a button with a resistor to a digital pin and the other end of the button to the 5V pin of the Arduino. Buttons are an easy way to get started creating some physical interactivity in your designs.

To debug your application, you can use the `Serial.print()` method to send messages from your Arduino controller to the console of the Arduino IDE at critical junctions to help you see what your application is doing.

The `Serial.print()` method can be formatted by passing a second parameter to the `print()` method to specify what format should be used.

You can import libraries to your Arduino application using the Sketch menu item of the toolbar at the top of the Application menu and then selecting the Import Library option. You can also write the name of the *.h* file that the library uses, if you've placed it in the *libraries* folder of your Arduino installation, like so:

```
#include <Wire.h>
```

Other Arduino libraries allow you to work with stepper motors, Two-Wire Interface, the memory of the Arduino controller, and many other techniques or controls.

Once you've written a working program and have uploaded it to the Arduino, the board will save the code in program memory and begin executing the program. The code will start as soon as the board is powered up again.

CHAPTER 5

Programming Revisited

In this chapter, we'll cover several more advanced topics that you will need to understand to use some of the other code samples in this book. First among these is *object-oriented programming* (OOP), that is, programming using classes and objects as the fundamental units to organize your code. This topic is a fundamental part of the nature of C++, which is quite important to understand if you're going to work with openFrameworks (oF) in any meaningful way. We will cover file structure and how to create classes and objects in C++. Finally, we will cover pointers and references, which are quite advanced topics but important ones if you plan on doing any serious work with openFrameworks. It also helps a lot when looking at libraries created for Arduino or if you decide to build your own library.

Object-Oriented Programming

To work with oF—and, in many cases, to work with Processing—you need to understand object-oriented programming, which is a way of organizing and assembling your code that is used in many different kinds of programming languages and for many different purposes. Hundreds of programming languages support OOP, and the principles don't vary too greatly across languages, so once you understand the basics of it, you'll find yourself better able to follow along with code that you encounter and better able to construct your own code when you're working on a project. At its core, OOP is the philosophy of creating *classes* that represent different tasks or objects in your application to delegate responsibility and organize the structure of your application. Classes can handle specific functionality like reading data from a camera and displaying it to a screen, they can represent a data object like a file, and they can be events for one part of an application to tell another about something that has happened. The class is the blueprint for what kind of object the class will create. When you create one of these classes to use in your application, you call that particular instance of the class an *object*. There may be unfamiliar terms in that previous sentence, but there probably aren't any unfamiliar concepts. My favorite analogy is the car analogy.

Imagine a car. What are the abstract qualities of a car? Let's stick with four right now: doors, windows, wheels, and the motor. So, we have a really basic abstract car that has a few different characteristics that we can identify. We're not talking about any *particular* car; we're talking about *any car*. Now, let's picture a particular car. I'm thinking of the red 1975 Volvo 240L station wagon that my parents had when I was a kid; you're most likely thinking of a different car. We can both identify our particular cars as being two instances of the abstract kind-of-thing car that share all the common characteristics of a car as well as having their own particular characteristics that make them unique objects. If you can handle that leap of taxonomic thinking, objects and classes should be fairly easy for you. Let's go forward.

Classes

All three tools profiled in this book—Arduino, Processing, and openFrameworks—provide a wide and rich variety of data types and methods. That said, it is inevitable that as you work with these tools you'll need to add your own constructions to these frameworks to create the works that you envision. These things that you need to add can be different kinds, but in all likelihood, they'll consist of some data and some functionality. In other words, they will probably consist of some variables and some methods. Once again, we need to approach this with the three distinct programming languages discussed in this book in mind. Arduino doesn't use classes as much as Processing or oF. This is because sending and receiving bytes from a microcontroller or from hardware tends to be a very "low-level" operation; that is, it doesn't frequently require the use of classes and new types. Arduino, and the C programming language on which it is based, can use something called a *struct*, which is collection of properties with a name, but we won't be discussing structs in this book. This isn't because structs aren't important, but rather because they aren't particularly important in the common tasks that are done in Arduino. Lots of times in Arduino when you need to do new things, you write new functions. This isn't always the case; when you're creating a library, for instance, you'll end up creating classes. But for beginners, OOP isn't something that you'll be doing a lot in Arduino. In Processing and C++, however, the kinds of things that these languages are good at frequently require new data types and new objects with their own methods. In Processing and C++, when you need to create a new different type to do something in your application or to store some data, you create a class.

Why do you use classes? The short answer is that you use classes to break functionality into logical pieces that represent how you use them, how the application regards them, and how they are organized. An application that handles reading data from multiple cameras and saving images from them as pictures will probably have a `Camera` class, a `Picture` class, and maybe a `File` class. A game with spaceships might have the classes `Spaceship`, `Enemy`, `Asteroid`, and so on. Using classes helps make the code for an application easier to read, modify, debug, and reuse.

The Basics of a Class

The concept of a class is a difficult one for many people at first; however, with a little bit of patience, you'll find that it's not all that different from the way you see the world. A *class* is a grouping of variables and methods into an object that contains and controls access to them all. Think of a very simple description of dog. A dog has a breed, an age, and a weight, for example. These are all traits that, if you were trying to describe the dog in code, you would use variables to describe. A dog also has some actions it can perform; it can run, it can eat, or it can bark. If you were to describe these actions in code, you would make methods to describe them. Let's go ahead and make a simple class in Processing that shows what this `Dog` class might look like:

```
class Dog{

    String breed;
    int age;
    int weight;

    Dog(){} // we'll talk about this one much more
    void run(){}
    void bark(){}
    void eat(){}
};
```

In C++, that class will look exactly the same. The differences between Processing and C++ are going to become more apparent later.

That's all. When you want to create a `Dog` in your application, just like when you make a String, you create a new instance of a `Dog`, declare the variable, and call the constructor:

```
Dog rover = new Dog();
```

Now that you've made `rover`, you can describe `rover` and call his methods:

```
rover.breed = "shepard"; // set the breed of the dog
rover.age = 3; // set his age
rover.weight = 50; // set his weight
rover.run(); // tell him to run
rover.bark(); // tell him to bark
```

That was nothing terribly challenging, was it? So, the concept of classes really isn't that difficult, because the idea was created to make programming easier for people, to make writing code mimic the way that people perceive and classify the world. Classes have properties (variables), like the `breed` property of the `Dog`, and methods (behaviors), like the `run()` method of the `Dog`. All methods of a class are created equal, except one, which we call the *constructor*. Let's revisit making the `Dog` variable `rover` once again:

```
Dog rover = new Dog();
```

Let's look more closely at `= new Dog();`. This calls the constructor, performing any actions that you want to perform when the `Dog` is first created. This will be the first thing that this instance of the `Dog` does, because this is where the `Dog` really begins, in

the constructor call. The `Dog()` method is a special method, which is why it doesn't return anything; that is, it doesn't declare a return type. Let's continue thinking about the `Dog` example. A `Dog`, `rover` in particular, has to have an age. It's impossible to have a dog without an age, so let's go ahead and say that when the `Dog` is created, you set its age:

```
Dog() {
    age = 1;
}
```

This means now that by default whenever you make a `Dog`, its `age` will be `1`:

```
Dog rover = new Dog();
println(rover.age); // prints 1, because the dog is 'just born' ;)
```

So now, you've created a simple class that you can use to create objects possessing all of the properties that you require. A well-written class lets you group functionality and data together in logical objects that allow you to focus on what the code needs to do, rather than how it is structured. This topic will be revisited many times in the following chapters.

Class Rules

Classes can be declared in files outside your primary code file. To work with a class or multiple classes that have been defined in separate files, those files and classes need to be imported. This is handled slightly differently in each of the different programming environments discussed here, so we'll discuss it in the chapter relevant to each of these languages. The filenames that these classes should be saved in are particular to each environment and language as well. In C++, classes are saved across two file types: *.cpp* files and *.h* files. The reason for this and the way that it works will be discussed in Chapter 6. In Processing, these external class files are saved in *.pde* files that are then imported. There are also *.java* files that can be imported into the Processing environment.

Both in Processing and in C++, the class declaration must always look like this:

```
class ClassName{
    // all the things the class has
};
```

Note that the class declaration is followed by curly brackets that enclose the definition of the class, which, in C++, is followed by a semicolon. In Processing and Java, the semicolon isn't required, but the brackets function the same way.

Classes should have a declared constructor unless they're just for storing data. For example:

```
class Point{
    int xPosition;
    int yPosition;
};
```

This doesn't require a constructor, but when you use your code, you'll have to use it like this:

```
Point pt = new Point();
pt.x = 129;
pt.y = 120;
```

If a constructor is provided for this class, as shown here:

```
class Point{

    int xPosition;
    int yPosition;
    // this is the constructor for this class
    Point(int xPos, int yPos){
        xPosition = xPos;
        yPosition = yPos;
    }
};
```

then the class can be used as follows:

```
Point pt = new Point(129, 120);
```

This saves you some typing and makes your code more compact. It also avoids the classic problem that you might run into later where you have `Point` objects that don't have any values because none were set. This way, all `Point` objects will have an *x* and *y*. In this case, the problem is taken care of, and you can concentrate on your project and not hunting down bugs.

A class generally should have good method names linked to the job that class is supposed to do. A good rule of thumb is to say that classes should be nouns and methods should be verbs. This isn't for the compiler; the compiler doesn't care what you call your class, variable, or method. This is more for you, the person writing the code. Remembering what you were thinking when you wrote a class or what the class should be used for is far easier if you have names for the class and the methods that make sense to you. For instance, a `Dog` should have `run()`, `bark()`, and `eat()` methods, but not a `paper()` method. A method called `fetchThePaper()` would be far more appropriate, but ultimately, your code is your own, and you can do whatever you like with it.

Public and Private Properties

A class has a name, and it has two basic kinds of properties: public and private. *Public* ones are available to the outside world, which means that other classes can use those properties and methods. *Private* properties are not available to the outside world, only to methods and variables that are inside the class, which means that other classes cannot use those properties and methods. When we create a class, we have the following shell in C++:

```
class Dog {
    public:
    //all public stuff goes here
    private:
    //all private stuff goes here
}; // a closing bracket and a ; to mark the end of the class
```

To give the `Dog` class some methods, you simply put the method definitions underneath the `public` and `private` keywords, depending on whether you needed that method to be public or private like so:

```
class Dog {
    public:
    void bark() {
        printf("bark");
    }

    void sleep() {
       // sleep() can call dream, because the dream() method
       //is within the Dog class
       dream();
}

    private:
    void dream() {
        printf("dream");
    }
};
```

In Processing, you do not need to use the `public:` keyword on top of all the public variables. Instead, each method is marked as public and private separately, like so:

```
class Dog{

    public void bark(){
        println("bark");
    }

    // sleep() can call dream, because the dream() method
    //is within the Dog class
    public void sleep() {
        dream();
    }

    private void dream() {
        println("dreaming");
    }

};
```

Now, if you create a `Dog`, called `Rover`, and try to call the `dream()` method on him, you'll get an error because `dream()` is private and cannot be accessed by external operations, that is, operations that are not going on inside the class itself:

```
Dog d;
d.bark();
d.dream(); // this doesn't work
```

Now, that may not make a ton of sense right away: why make things that we can't use? The answer to that is design and usage. Sometimes you want to make certain methods or properties of a class hidden to anything outside of that class. Generally, you want a class to expose the interface that something outside of your class can access and have the class do what it is supposed to do. Methods or properties that aren't necessary for the outside world to use or manipulate can be declared private to keep objects outside the class from accessing them or using them inappropriately. You might not ever make methods private, but if you work with C++ and Processing long enough, you're guaranteed to run into methods and variables that have been marked private when you're looking at other people's code. Now you'll understand what it is and why they might have done it.

Inheritance

Discussing public and private brings us to inheritance. This is a big part of OOP, and, like many of the other topics in OOP, seems tricky at first but ends up being fairly easy and common sense. We'll look at some simple examples using `Dog` and then some more complicated examples using shapes later to help you get the hang of it.

Inheritance means that a class you're making extends another class, getting all of its variables and methods that have been marked public. Imagine there is a general kind of `Dog` and then specialized kinds of dogs that can do all the things that all dogs can do, such as run, bark, smell, *plus* some other things. If I were to make a class called `Retriever`, I could rewrite all the `Dog` stuff in the `Retriever` class, *or* I could just say this: A `Retriever` *is a* `Dog` with some other special stuff added on. Let's say the `Retriever` does everything the `Dog` does *plus* it fetches.

Here's the code in Processing:

```
class Retriever extends Dog{

    public void retrieve() {
        println("fetch");
    }

}
```

Here's the code in C++:

```
class Retriever : public Dog {
public:
    void retrieve() {
        printf("retrieve");
    }
```

```
private:

};
```

Does this mean that you can have the `Retriever` bark? Since the `Retriever` is a `Dog`, the answer is yes, because all the methods and variables from the `Dog` are available to the `Retriever` class that extends it.

Here's the code in Processing:

```
Retriever r = new Retriever();
r.bark(); // totally ok, from the parent class
r.retrieve();
```

Here's the code in C++:

```
Retriever r; // note, you don't need to do = new...()
r.bark(); // totally ok, from the parent class
r.retrieve();
```

In Figure 5-1, you can see how the methods of classes are passed down what is sometimes called the *chain of inheritance*.

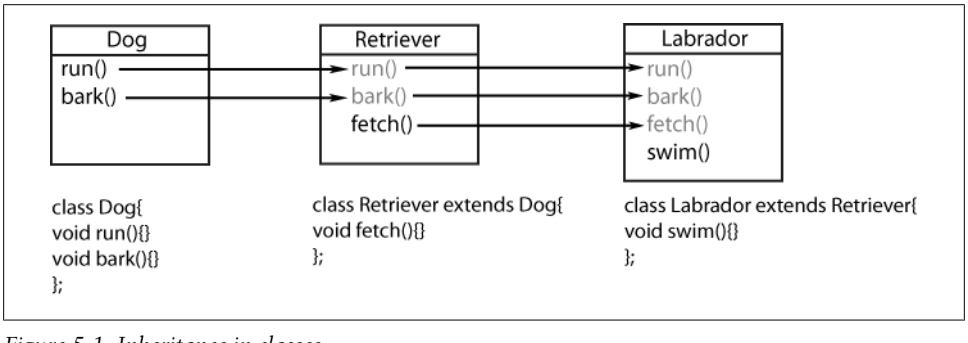


Figure 5-1. Inheritance in classes

The `Dog` class defines two methods: `run()` and `bark()`. The `Retriever` logically extends the `Dog`, since the `Retriever` is a dog, but it is also a specific kind of dog, namely, the kind of `Dog` that can fetch things. So, the `Retriever` defines only one method, but actually it has three because it inherits from the `Dog`. The `Labrador` is a kind of `Retriever` and hence is a `Dog` as well. A `Labrador` can do all the things a plain old `Retriever` can do because it extends the `Retriever`, but it also can `swim`. So, even though the class for the `Labrador` class has only one method, it can do four things, `run()`, `bark()`, `fetch()`, and `swim()`, because it always inherits all the methods of its parent classes.

Why is this important? Well, if you're going to make interactive designs and artwork that use code, you're going to end up reading a lot of code. You're inevitably going to run into a class that is calling methods that you won't see anywhere when you look at the code listing. It's probably defined in one of the parent classes that the class extends.

You also might start making types of things that you realize share some common characteristics. If you're making an application that draws lots of things to the screen, it might make sense to put a `draw()` method on them all. If at some point later in writing code you want to change some core aspect of the `draw()` method, you would go through every class and change it. It would make things easier if the `draw()` method was in one place, like a parent class, for instance. This is another one of the advantages of creating class structures: to avoid duplicating code unnecessarily and make your code simpler to read, change, and share.

There's a caveat to using inheritance with your classes, which is that *only methods and variables marked as public* will be inherited. You heard right: anything marked private doesn't get inherited. There's one small catch to this: in both C++ and Java, there are a few additional levels of privacy for methods and objects, but they're not as important for the purposes of this book. So, in the class shown in Figure 5-2, again using the `Dog` and `Retriever` analogy, the `Retriever` will have a `bark()` method that it will inherit from the `Dog` class, but it will not have a `dream()` method because the `Dog` has that method marked as private.

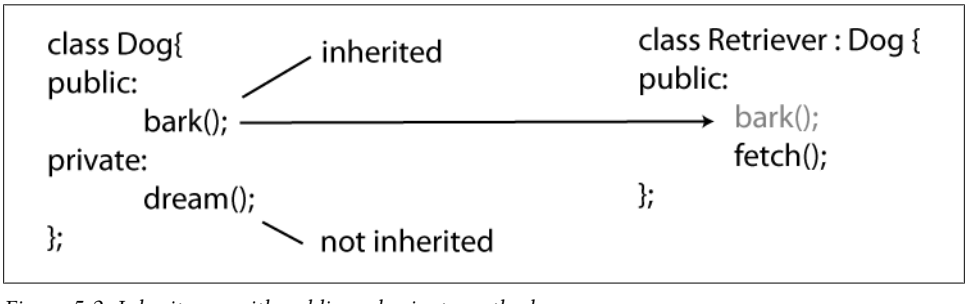


Figure 5-2. Inheritance with public and private methods

The same would go for any variables defined in the `Dog` class; the `Retriever` will inherit *all public* variables and *no private* ones. This is a good thing to know when reading through the code of libraries that you might want to use, like the `ofImage` or `ofVideoGrabber` classes in openFrameworks or some of the libraries available in Processing.

Processing: Classes and Files

In Processing, classes often are defined within the main file for the application that you see when you're working with a Processing application. If you open the folder that your application is saved in, you'll see that this main file is a *.pde* file that has the name of your application. This is the main file of your application, and lots of times if you want to make a new class, you'll just add it to this file. However, in a complex application, you may want to make multiple classes, and you want to break them apart into separate files so they're easier to read and maintain, or so you can use a file from another place.

To do this, all you need to do is create a new *.pde* file in the same folder and put your class declaration and description into that file.

To create a new *.pde* file within your application folder that will automatically be imported into your application, click the New Tab icon at the right of the Processing IDE, as shown in Figure 5-3.

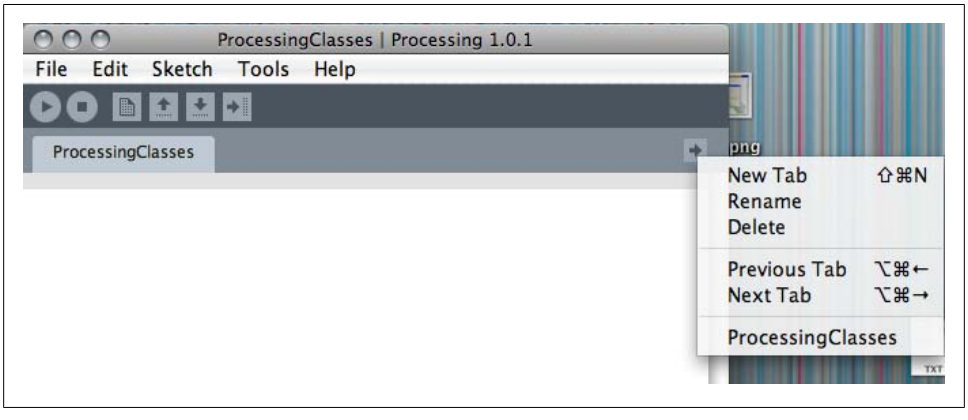


Figure 5-3. Creating a class in Processing

This opens a small dialog box asking you to name your new file (Figure 5-4).

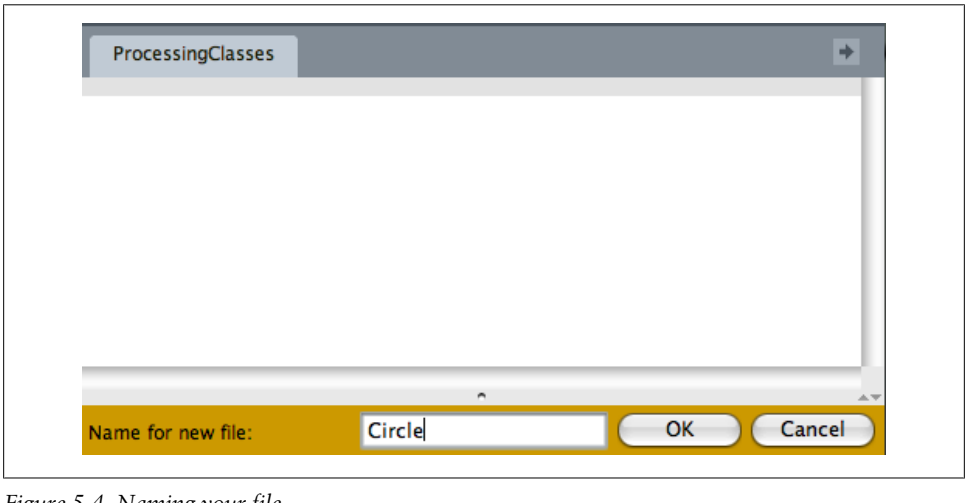


Figure 5-4. Naming your file

After you've given your file a name, Processing automatically creates the new *.pde* file in your sketch folder. You can begin adding your class to the file and then using it in your main application file, that is, the file that contains the `draw()` and `setup()` methods (Figure 5-5).

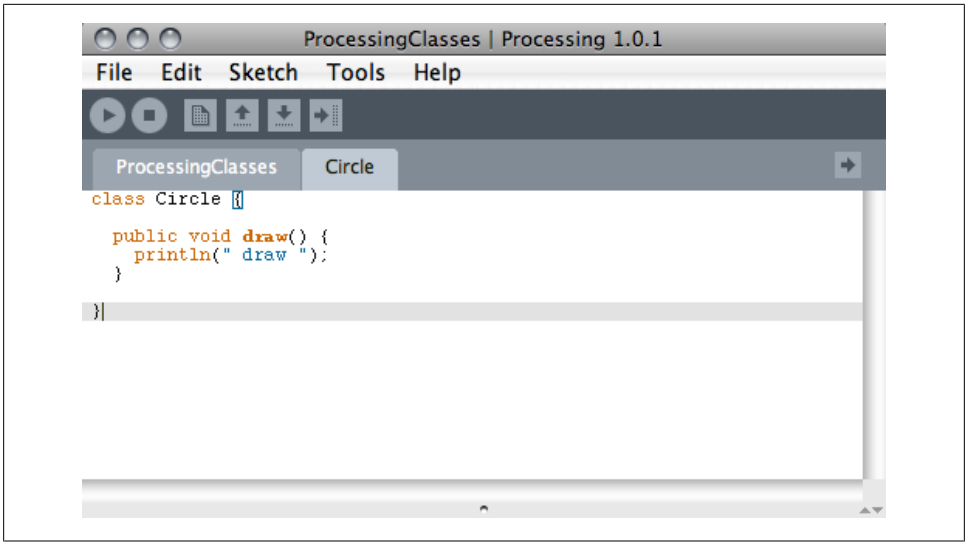


Figure 5-5. The newly created class file

What do you get for putting this extra work into creating new files and classes? There are two distinct advantages to creating files and separate classes: it makes organizing your code and debugging your application easier, and once you've written some classes that work well and do something that you'd like to use again, it's easy to copy and paste the file into a new project and reuse all your code from the class. This last bit is, although oversimplified, a lot of the basis of the Processing, Arduino, and openFrameworks projects: providing reusable code to speed up creating your projects.

C++: Classes and Files

C++ is an object-oriented language. It's quite difficult to write anything in C++ without creating at least a few classes, and as you work with oF, you'll find classes and class files all over the place. If you've read this chapter up to this point, you should have a fairly good idea of what a class is and what it does.

So, now we're going to define an absurdly simple class called `Name`:

```
class Name {
public:
        string firstName;
        string secondName;

        Name(string first, string second) {
                firstName = first;
                secondName = second;
        }
};
```

Next up is creating an instance of `Name`. So, somewhere else in our code, we create an instance of `Name` like so:

```
Name name("josh", "noble");
```

That's it. We've made a class, and we've instantiated an instance of it. We're sort of off to the races.

So, what are some things that are commonly created and stored as classes? A good example is one of the oF core classes, the `ofSoundPlayer` class. This class represents what it sounds like it would represent: a sound player. It has methods to play, stop, set the volume, and set the position that the player reads from. We wouldn't expect it to do anything else, and in fact it doesn't. It also provides lots of useful properties that help us determine whether a sound is currently playing and whether there was a problem loading the sound.

Why do we use classes? In the short answer, we use classes to break functionality into logical pieces that represent how we use them, how the application regards them, and how they are organized. It makes sense to think of a spaceship game having spaceships, enemies, asteroids, planets, high scores, and so on. For example, we use classes to organize all the functionality that an enemy would need into one place so that when we need to make a enemy, we know we're always getting the same characteristics, and when we need to make a change to the enemies, we can do it in just one place. It's far more difficult to introduce classes later than it is to simply begin with them. The theory of how to best organize classes is a serious topic that has spawned many a serious nerd-spat. We're not going to delve into it deeply, but in Chapter 18, there are some pointers to introductory texts as well as heavy classics.

.cpp and .h

.cpp and *.h* shall now forever be burned into your mind as *the two file formats of a C++ class*. That's right: two file formats and two separate files. The why of this is a bit irrelevant; what's important is that you follow how the two work together and how you'll use them. We're going to take this slowly because it's important.

In an oF application, there will be two types of files: *.cpp* and *.h* files. *.cpp* files contain the implementation of a class, while the *.h* file contains the prototype of that class. The prototype of that class, the *.h* file, is going to contain the following:

- Any `import` statements that the class needs to make
- The name of the class
- Anything that the class extends (more on this later)
- Declarations of variables that the class defines (sometimes referred to as *properties*)
- Declarations of methods that the class defines

The definition of the class, the *.cpp* file, will contain the following:

- The actual definition of any methods that the class defines

So, why are there so many things in the *.h* file and so few in the *.cpp*? Well, usually the definition of a method takes up a lot more space than the declaration of it. If I want to define a method that adds two numbers and returns them, I define it like so:

```
int addEmUp(int a, int b);
```

This is the kind of thing that you'll see in an *.h* file—methods with signatures but no bodies. Take a moment to let that sink in: the method is being *declared*, not *defined*. Now, if I want to do something with this method, I need to actually define it, so I'll do the following:

```
int addEmUp(int a, int b){
    return a+b;
}
```

Here, you actually see the meat of the method: add the two numbers and return it. These are the kinds of things that you'll see in the definition of a class in the *.cpp* files.

If you want to have your class contain a variable, and you will, then you can define that in the *.h* header file as well. Just remember that the declarations go in the *.h* file and the definitions go in the *.cpp* file, and you'll be OK. When you're working with oF, you'll frequently see things that look like Figure 5-6.

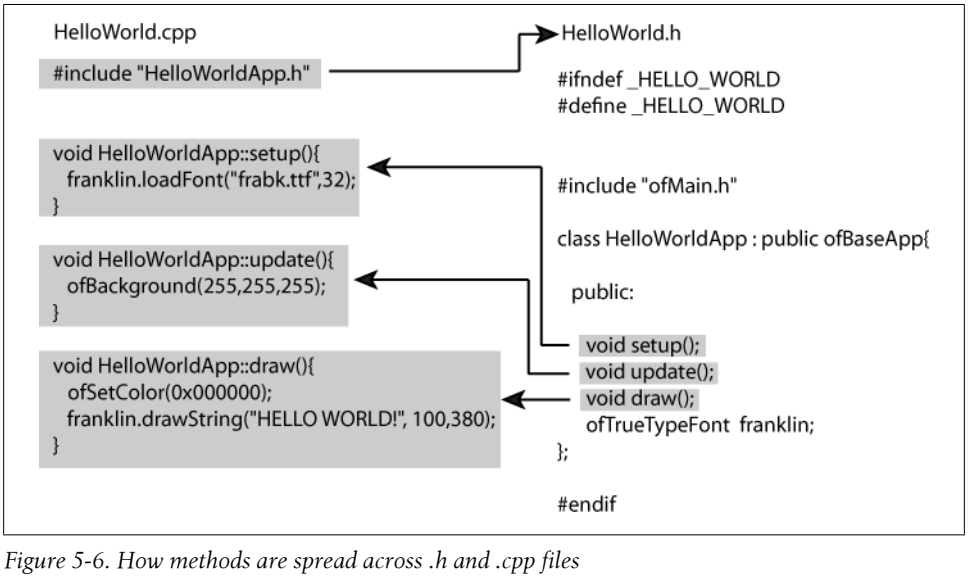


Figure 5-6. How methods are spread across .h and .cpp files

Don't worry about all the specifics of this diagram; we're going to return to it in Chapter 6 when we discuss oF in greater detail. For now, just notice how the `setup()`, `update()`, and `draw()` methods are declared but not defined in the *.h* file, and how they are defined in the *.cpp* file with the name of the class in front of them. This is going to be a wonderfully familiar pattern for you by the end of this book.

A Simple C++ Application

Example 5-1 is your first C++ class, helpfully named `FirstClass`. First up you have the *.h* file where all the definitions are made:

Example 5-1. FirstClass.h

```
// make sure we have one and only one "First class"
#ifndef _FIRSTCLASS
#define _FIRSTCLASS

// give the class a name
class FirstClass{
// declare all the public variables and methods, that is, all the values that can
// be shared
public:
    FirstClass();
    int howManyClasses();
// declare all the private variables and methods, that is, all the values that are
// only for the internal workings of the class, not for sharing
private:
    int classProperty;
};

#endif
```

We'll talk this through, because the first thing you see is this really odd `#ifndef`. Don't let this scare you away, because it actually has a perfectly good explanation. Frequently in a larger project, you'll have a single class that is referenced in lots of other classes. Going back to our little space game example, our `Enemy` class is going to be needed in more than one place. When the compiler is compiling our classes, we don't want it to compile `Enemy` multiple times; we want it to compile the enemy only once. This weird little `#ifndef` statement tells the compiler that if it hasn't already defined something called `_FIRSTCLASS`, then it should go ahead and compile everything in here. If it *has* already defined something called `_FIRSTCLASS` and, by extension, also compiled all the code in our header file, then we don't want it to do anything. The `#ifndef` acts just like any other `if` statement. Take a glance at the end of the class, and note the `#endif`; this is just like the closing `}` on a regular C++ `if` statement. The rest of it is much more straightforward. The `class` keyword goes in front of the name of the class and a `{`. All the class data is contained within the opening class statement `{` and the closing class statement `};`.

Inside of that you can see the word `public`. This means that after the `public` keyword, everything will be publicly accessible, which means that it is available to other classes that have access to instances of `FirstClass`. Everything up until the `private` keyword will be `public`, and, logically, everything after the `private` keyword will be `private`.

So, by looking at the *.h* file, you know the name of the class, what properties your class defines, what methods it defines, and whether those properties and methods are public or private. That's great, but it's not quite everything, so we'll list the *.cpp* file so you can see how all these methods are actually defined in Example 5-2.

Example 5-2. FirstClass.cpp

```
// first import our header file
#include "FirstClass.h"
// then import the file that contains the 'print' method
#include <iostream>


// this is our constructor for the FirstClass object
FirstClass::FirstClass()
{
    // this is a little magic to get something printed
    // to the screen
    printf(" FirstClass \n");
    classProperty = 1; // initialize our property
}

int FirstClass::howManyClasses()
{
    // once again, just a little message to say 'hello'
    printf(" howManyClasses method says: 'just one class'' \n");
    // note that we're returning classProperty, take a look at the method
    // declaration to understand why we're doing this (hint, it says 'int')
    return classProperty; // do something else with our property
}
```

Note that for each of the methods that are declared in the *.h* file there is a corresponding definition in the *.cpp* file. Here, you can see how each of the methods of the class is put together with both a definition in the *.cpp* file and a declaration in the *.h* file. So far, so good. You have an instance of a class, but you aren't creating an instance of it anywhere yet. That's about to change with the final file. This last file is not a class but is a single *.cpp* file called *main*. There is a rule in C++ that says you must have one file that has a method called `main()`. When your application is launched, this is the method that will get called, so it's where all your initialization code goes. In an oF application, this is just creating an instance of your application and running it; this will all be covered in greater detail in Chapter 6. There is no rule that says you must have a file called *main* in order to run your application, but it's the way it's done in oF, and that means it's the way we'll demonstrate in Example 5-3.

Example 5-3. main.cpp

```
#include "FirstClass.h"

// all applications must have a 'main' method
int main() {
    FirstClass firstClass; // here is an instance of our class
    firstClass.howManyClasses(); // here we call a method of that inst.
    return 0; // here we're all done
}
```

So, is this the only way to make a simple C++ application (Figure 5-7)? The answer is no; however, this is the way that oF is set up, so this is the way that we'll show.

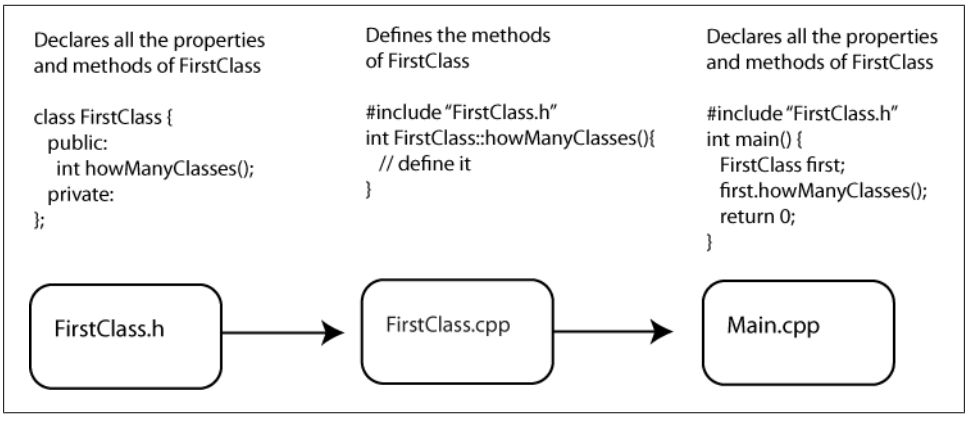


Figure 5-7. The organization of a C++ application

There are many more tricks to classes in C++ and many more wonderful mysterious things that you can do. If you want to learn more, you're in luck, because there are dozens of good manuals and guides to help you learn C++. If you don't really want to learn more, you're in luck, because oF hides much of the complication of C++, allowing you to concentrate on other parts of your project.

Pointers and References

C++ is a very powerful and very fast programming language because it is very "low-level." That is, it doesn't hide the inner workings of the computer nearly as much as a language like Processing. This has upsides and downsides. To follow many of the openFrameworks examples in this book, you'll need to have at least a cursory familiarity with two concepts: the *pointer* and the *reference*. If you're not interested in working with openFrameworks and C++ right now, then feel free to skip this section and move on. You might come back to it later; you might not. Above all, this book is meant to be helpful to you, not to shove things at you that you're not interested in.

The pointer and the reference are complementary concepts that are inextricably linked. To understand how they work, though, you need to first understand what happens when a variable is declared and see how the memory of a computer works. Take a look at this:

```
int gumballs = 6;
```

What you're doing here is telling the computer to store something the size of an `int` with a value of `6` and a variable name of `gumballs`. It's important to understand that the value and the variable name are separate. They are exchanged for one another when needed. One example is when you add `5` to `gumballs`:

```
int moreGumballs = gumballs + 5; // "gumballs" is replaced with 6
```

This is easy to do because `gumballs` is really representing a part of memory. That section of memory is represented by hexadecimal numbers like 0x2ac8 that simply stand for the beginning of the location in the computer's memory where the value of `gumballs` is stored. The variable `gumballs` simply stores that location so that you can access it easily. When you set `gumballs` to `8`:

```
gumballs = 8;
```

what you're doing is changing the value that is stored in the section of memory that `gumballs` represents. When you *set* `gumballs`, you set that section of the computer's memory. When you *get* `gumballs`, you read that section of the computer's memory. You're reading the bytes stored in the computer's memory by location. That location is referred to by the *friendly* variable name, but in reality, it's a physical location. It really doesn't matter a ton, until you start to deal with pointers and references, because the pointer and the reference let you declare up front whether you want to work with a pointer to a location in memory, or whether you want to deal with that memory explicitly. We'll talk more about when and why you would do each of those things later.

Here's a simple example that you can run with any C++ compiler to see how this works:

```
#include <iostream>
int main () {
    int gumballs = 6;
    int gumdrops = 12;
    // here's where we make the magic
    printf(" the variable gumdrops has the value of %i and ¬
        the address of %p  \n", gumdrops, &gumdrops);
    printf(" the variable gumballs has the value of %i and ¬
the address of %p  \n", gumballs, &gumballs);
}
```

This will print out something that might look like this:

```
the variable gumdrops has the value of 12 and the address of 0xbffff998
the variable gumballs has the value of 6 and the address of 0xbffff99c
```

Notice two things. First, notice the `&` in front of the two variables:

```
&gumdrops
&gumballs
```

These are both references, which indicate that you want the name of the location in memory where these are stored. When you use the variable name, you just get back what's stored there. When you use the reference, though, you get the name of the location where the actual value is stored. Second, notice the actual location in memory: `0xbffff998`. That's just on my computer at one particular moment in time, so if you run the code, your output will almost certainly look different. This is interesting because this is the place that `gumdrops` is stored. What is stored there is an integer, `12`, and the way you get at that integer is by using the variable name `gumdrop`.

Reference

The reference, as mentioned earlier, is the location in memory where a variable is stored. Any variable's address can be accessed using the reference operator (`&`). Though much more can be done with references, in this book we're going to limit the use of references solely to initializing pointers. With that in mind, we'll talk about a pointer and show how the pointer works.

Pointer

Pointers are so called because they do in fact point at things; in particular, they point at locations in memory much like you just saw in the discussion of references.

The pointer, like a variable and a method, has a *type*. This is so you know what kind of thing is stored in the memory that the pointer is pointing at. Let's go ahead and make a pointer to point at `gumdrops`:

```
int* pGumdrops = &gumdrops;
```

There are two important things to notice here. First is that when the pointer is created, the `pGumdrops` variable uses the same type as the variable whose address the pointer is going to point to. If you are making a pointer to point to a float, then the pointer must be of type `float`; if you are making a pointer to point to an object of a class called `Video`, then the pointer must be of type `Video` (assuming that `Video` is a class). The second is that the pointer is marked as a pointer by the presence of the `*` after the type declaration. A pointer to a float would look like this:

```
float* pointerToAFloat;
```

In the following code snippet, you have *declared* the pointer, but have not *initialized* it. This means that it doesn't point at anything yet:

```
pointerToAFloat = &someFloat;
```

Now you have initialized the pointer, which means that it can be passed around in place of the variable itself, which is all well and good, but it doesn't help you do things with the pointer. To do things with the pointer, you need to *dereference* the pointer:

```
int copyOfGumdrops = *pGumdrops;
```

To dereference the pointer—that is, to get the information that is stored in the location that the pointer points to—you dereference the pointer by using the * symbol in front of the name of the pointer. Think of the * as indicating that you're either telling the pointer to point to some place in particular, initializing it, or getting what it points at. Here's another example showing how dereferencing the pointer works:

```
int gumdrops = 12;
int* pGumdrops = &gumdrops;

printf(" pGumdrops is %i  \n", *pGumdrops);// will be 12
gumdrops = 6;
printf(" pGumdrops is %i  \n", *pGumdrops);// will be 6
gumdrops = 123;
printf(" pGumdrops is %i  \n", *pGumdrops);// will be 123
```

Any time you change the value of `gumdrops`, the value that `pGumdrops` points to changes. Figure 5-8 illustrates the relationship between the variable and the pointer. When you do the following:

```
int gumdrops = 12;
int* pGumdrops = &gumdrops;
```

you are creating the relationship shown in Figure 5-8.

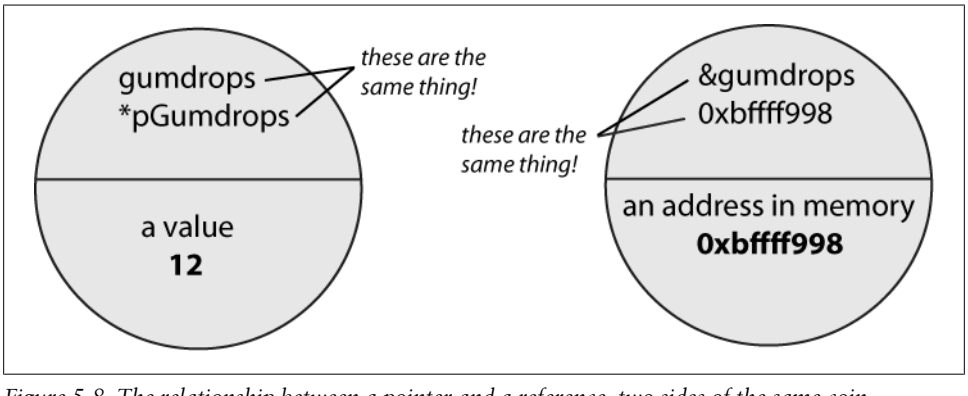


Figure 5-8. The relationship between a pointer and a reference, two sides of the same coin

When to Use Pointers

As you may recall from the earlier discussion on methods, when something is passed to a method, a copy of it is used for manipulation. This generally means that when you pass something to a method, it doesn't alter the original variable:

```
void addExclamation(string s){
    s+="!";
}

string myName = "josh";
addExclamation(myName);
printf( myName ); // will still be 'josh', not 'josh!'
```

If you wanted to alter the variable that is passed in to the `addExclamation()` method, you would pass a pointer to the method:

```
void addExclamation(string* s){
    *s+="!";
}

string myName = "josh";
string* pName = &myName;
func(pName);
printf( myName ); // will now be 'josh!', not 'josh'
```

When you pass a pointer you are passing an object that points to the actual memory, and dereferencing the pointer allows you to change what is actually in memory. If you have a very large picture that you want to alter, you want to alter the picture, not create a copy of the picture and alter that. Why not? Because it might not be the picture that's currently open, because you might want to alter a particular picture in the filesystem, or because the picture might be really large and altering might take up too much memory. This brings us to the second extremely common situation in which you might use pointers.

Large Data Objects

If you have a video file, a huge photo, or a lot of sound data, you don't want to be making copies of that all over your program, though you'll probably need to access it all over the place. The judicious use of the pointer is a way to allow different parts of a program to access a variable and alter that variable by making multiple copies of it.

There are several very important rules to pointers. Write these down somewhere, commit them to memory, or otherwise preserve them:

- Don't try to access a pointer before it has been initialized. This:

  ```
  int* pInt;
  printf(" printing out pInt %i ", *pInt);
  ```

  will not print anything meaningful.

- Check whether a pointer is `NULL` before doing anything with it:

  ```
  if(ptr != NULL) {
      // now you can use the pointer
  }
  ```

  If you try to use the pointer and it isn't assigned to anything, then your application won't do what you want it to do and might crash.

- Although you can do this:

  ```
  int* ptInt = 212;
  ```

  you really shouldn't. This can create bugs that are really difficult to track down. When using a pointer, unless you're very confident in your skills, brave, or just

very smart, you should set pointers to initialized variables. If you're comfortable with dynamically allocating memory, it can be a very powerful tool.

- You might come across code that looks like this:

```
FirstClass* fc = new FirstClass();
```

  This creates a new `FirstClass` and keeps a pointer to the object that can be used to access that object. If this is something that you're curious about understanding better, look in Chapter 18 for a list of books focusing entirely on C++.

- If you dynamically create an object, you have to clean up after yourself by calling `delete` on the pointer:

```
FirstClass* fc = new FirstClass();
// do some things with fc
...
// now we're all done with it, so clean up
delete fc;
```

  If you don't delete pointers that you've dynamically allocated when you're done using them, and especially if you assign them in a method and don't delete them before the method is finished, you're creating what's called a *memory leak*. These can cause your application to run slowly or crash.

Pointers and Arrays

Pointers and arrays have a very close relationship. An *array* is in fact either a pointer to a section of memory or a pointer to a bunch of pointers. Really, when you make an array of `int` variables:

```
int gumballs[10];
```

what you're saying is that there are going to be a list of `10` integers next to one another. It's the same thing as this:

```
&gumballs[0]
```

How's that? Take a look at what declaring the array actually does in Figure 5-9.

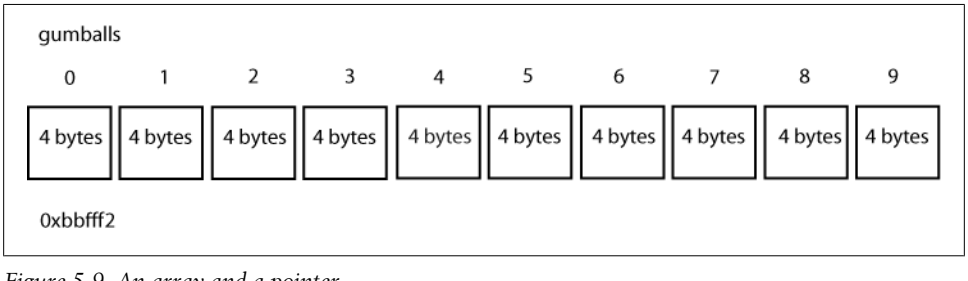


Figure 5-9. An array and a pointer

What you are doing is making a series of 10 integers, all in a row, within memory. Nothing links them together, except that they are all right next to one another. When you get the second element in the array, what you're doing is *getting the next integer after the initial integer*. The array itself is just a pointer to the location in memory where the first element of the array is stored, along with a number to indicate how many more places are stored for the array. In this case, there are 9 after the initial place, for a total of 10. So, if the beginning of the array is the location of the first element in memory, how do we get the next element in memory? By adding one to the pointer. Don't let this throw you for a loop, because it's deceptively simple:

```
int myInt = 4;
int* ptInt = &myInt;
int* nextPointer = ptInt + 1;// point to the next piece of memory
```

Imagine a row of shoe boxes. Imagine that you make a metaphorical pointer to the second shoe box by saying "that red shoe box there." Saying "the next shoe box" is like saying "the shoe box to the right of the red shoe box." You're not doing anything with the shoes inside those shoe boxes; you're just getting the next shoe box. That's exactly what adding to the pointer does. You're not adding to the value that the pointer is pointing to, you're adding to the location in memory that you're interested in accessing. Going back to our `gumballs` array of integers, when you get the second element in the array, what you're doing is this:

```
gumballs + 1
```

because `gumballs` is a pointer to the location of the first element in the array. You can do some really clever things with pointer arithmetic, but they aren't particularly important at the moment.

When Are You Going to Use This?

If you follow along with this book, you're going to encounter pointers first when getting sound data in an openFrameworks application. The main application class in oF is called `ofBaseApp`, and it has an `audioReceived()` method that you can use to get data from the sound card of a computer. The `audioReceived()` method looks like this:

```
void audioReceived (float * input, int bufferSize, int nChannels)
```

That pointer to `float` is a pointer to the beginning of an array of sound samples. All the data of a sound is stored in a computer in an array of floating-point variables. When you want to work with the sound that your computer's sound card has captured, you'll get a pointer to an array and a `bufferSize` variable that tells you how large the array is. To use that sound data, you simply loop through the values in that array using a `for` loop:

```
for (int i = 0; i < bufferSize; i++){
    printf("left channel is %f and right channel is %f",input[i*2],
        input[i*2+1]);
}
```

The reasoning behind the way the elements are accessed from the input array will be explained in Chapter 7. For the time being, just rest assured that, if you've followed everything in this chapter and the previous section in particular, you're going to be able to put it to use to make some interesting stuff. If you didn't really follow that last bit, don't sweat it too much, because you're not going to really need to work with pointers if you don't want to quite yet. You'll find that as you spend more time around these things and see how they're used more, they'll start to make a lot more sense to you.

Review

A class is a group of variables and methods that are stored together within a single object. For example:

```
class Point{
    int xPosition;
    int yPosition;
};
```

Once classes are defined, they can be used like other variable types. First defined, then initialized:

```
Point pt = new Point();
```

Classes can define multiple methods and variables. One of the special methods that a class defines is the *constructor*. The constructor is a method that has the same name as the class and is called when the class is instantiated.

```
class Point{
    int xPosition;
    int yPosition;
    Point(){
        // anything to do when the object is first constructed
    }
};
```

The constructor is the only method that does not use a return type.

Pointers are a special type that exist in C++ and Arduino that point to a location in memory where a value is stored. They are declared with a type, like any other variable, but use an * immediately after the type declaration to indicate to the compiler that the variable is a pointer:

```
int* intPt;
```

A reference refers to the actual location in memory where the value of variable is defined and is indicated by the `&` in front of the variable. Once the pointer is declared, it can be set to point to the reference of a variable:

```
int apples = 18;
intPt = &apples;
```

Pointers can be set only to the reference of a variable or a value but should usually be set only to the reference of a variable.

Pointers can be incremented using the + and decremented using the - symbol. This moves the location in memory to which the pointer points.

CHAPTER 6
openFrameworks

openFrameworks (oF) is the third and last tool you'll learn about in this part of the book. As you may have guessed from its name, oF is a *framework*, which is a collection of code created to help you do something in particular. Some frameworks are designed for working with databases, and others are designed for working with Internet traffic. Frameworks don't provide you with a prebuilt tool in the way that Processing and Arduino do, but they provide you with code that you can use to create your own programs.

Specifically, oF is a framework for artists and designers working with interactive design and media art. That sounds simple enough, right? It is written in C++ and expects that you write programs in C++. As mentioned in Chapter 2, C++ is a big, powerful, old, and (again for emphasis) powerful programming language. You can create features using oF that would not be possible in Processing because the underlying language is so much more flexible and low-level. Although this isn't always ideal or necessary, when it is, you'll appreciate it. It isn't always necessary to use oF, though. Sometimes Processing will work just as well. Let's take a quick look at some tasks you can do in Processing that aren't as easy to do in oF, and vice versa:

Make a project visible on the Internet
: Use Processing; this is much trickier to do with oF.

Make a project that renders a lot of 3D graphics
: Use oF; this is where C++ really shines.

Make a project that can run on many different computers without a lot of configuration
: Use Processing; this is rather tricky to do with oF.

Make a project that uses a computer vision library like OpenCV
: Use oF Processing doesn't communicate with a C++ library like OpenCV as easily as oF.

Make a project that interfaces with the Arduino controller
: Use either, depending on what you need to do with the information from the Arduino controller.

openFrameworks is developed by Zach Lieberman, and Theo Watson, Arturo Castro, and Chris O'Shea, along with help from collaborators at Parsons School of Design, MediaLabMadrid, and Hangar Center for the Arts, as well as a far-flung network of developers, artists, and engineers. The framework originated in the Design and Media Arts program at Parsons School of Design, where Lieberman was a student. He found himself needing increasingly more power for his projects and began to learn C++. As he worked, he realized that although thousands of different libraries were available to C++ developers, nothing provided the same kind of ease of use and low barrier to entry for artists like Processing does for Java development. Hence, openFrameworks was born.

Your IDE and Computer

Your computer and its operating system are about to become very important for a moment. That's because oF, unlike Arduino and Processing, does not have an integrated development environment (IDE). It requires that you use another IDE, and that IDE depends on your platform and what's available to you. So, we'll break it down by operating system. Because the requirements and instructions for getting your IDE set up may change without warning, the following sections are going to be less of step-by-step instructions and more of a general overview of what is involved. The openFrameworks website (*www.openframeworks.cc*) will have the most current instructions, and you can probably resolve any problems you run into on its forums.

Roughly, each of these OS + IDE combinations requires two steps. The first is to install and configure the IDE correctly, and the second is to download the appropriate oF package. Since oF really is just a collection of C++ code, you simply put the code files in the appropriate location for the IDE. You can then get started running the sample programs that come with oF, or can start building your own programs.

Windows

Those wanting to work with Windows are well taken care of. The structure and philosophy of oF work well with Windows, and several major Windows-based projects have been built in oF on Windows already.

Visual Studio

Visual Studio is probably the most popular development environment for Windows. It's developed by Microsoft and is very powerful, but as of this writing, it does not always play well with oF. That said, it is a well-designed IDE that provides a lot of tools for developers. The current version of Visual Studio is Visual Studio 2008, and the free version of Visual Studio is Visual Studio C++ Express Edition.

Code::Blocks for Windows

Code::Blocks is an open source IDE for C++ development. Currently, it is the preferred tool for working with oF on Windows. Code::Blocks comes bundled with a compiler system called MinGW that replicates the free and open source GNU Compiler Collection. If you don't really follow that last bit, don't worry about it—just know that because of MinGW, Code::Blocks is free. To get started with Code::Blocks, first download the IDE from *www.codeblocks.org* and install it. Once you've gotten Code::Blocks set up, download the oF for Windows Code::Blocks project files. Once you've downloaded the packages, you'll be able to open any one of the example programs and run them. Simply navigate to the *apps/examples/* section of the download and open up any one of the applications in the Code::Blocks IDE to get started.

Mac OS X

Anyone with a Mac will have access to Xcode, which means you have access to an excellent IDE. Take a look at the top-level directory on your main hard drive. If you see a folder called *Developer*, then you're set. Otherwise, you'll need to install the Developer Tools packages. You can do this in two ways: insert the operating system DVD that came with your computer and install the Developer Tools, or go to *http://developer.apple.com/technology/xcode.html* and download and install the Developer Tools from the *.dmg* file you've downloaded. You'll need admin privileges on the computer to do this.

Once you've gotten set up with Xcode, you can download the Xcode oF package and get started by opening any one of the *xcodeproj* files in the example *applications* folder.

Linux

Lovers of Linux have many options, but we'll cover just two. In either event, you're going to need to grab the following libraries via APT or RPM (your package management tools):

- freeglut3
- alsa-dev
- libxmu-dev
- libxxf86vm-dev

Code::Blocks for Linux

Code::Blocks for Linux is quite a bit different from Code::Blocks for Windows. To get Code::Blocks up and running, you'll need to get the following tools installed:

- gdb
- wxWidgets

Once you have those installed, you can go ahead and grab the appropriate Code::Blocks IDE and get started. You'll have to take a few more steps, so I highly recommend you refer to the instructions on the openFrameworks wiki.

Using makefiles

Ah, the makefile—the classic build tool. If you're thinking about using makefiles, you're probably at least a little C++ savvy already. You don't need anything more than the libraries listed earlier in the section "Linux" on page 95 and the *.sh* files included with the download to build and clean all the sample projects.

Taking Another Quick Tour of C++

In the previous tools you looked at in this book, Processing and Arduino, you learned about the more popular language that each of those languages extends and resembles: Java in the case of Processing, and C in the case of Arduino. In both those cases, however, we had to note that although the tools' languages were similar and derived from Java and C, respectively, they were not quite the same. You cannot just use Java code in a Processing program; you need to take certain steps for that to happen. The same goes for Arduino regarding C. oF is different because oF is C++, end of story. Anything you can do with C++, you can do with oF, because oF is simply a C++ library that was built with the intention of making C++ easier to use. This has a wonderful upside to it, in that oF is very straightforward, incredibly extensible, and is as powerful as C++, which is to say, it's about as powerful as programming languages get.

The downsides, though, are that C++ can be difficult to learn and that the tools across platforms aren't completely standardized. So, in order to get you an appropriate tour of oF, you'll need to take a rudimentary tour of C++. Now, as with many other things in this book, this tour is an introduction, not a complete guide. Hundreds of C++ books exist for a reason: C++ is a massive topic. This quick tour is intended to provide you with the information you'll need to get started with oF C++, and creating interactive programs using them. At some point, though, you'll need to refer to another resource, be it the oF forums, an online reference (of which there are many good ones), or another book. That said, oF makes working with C++ easy and relatively painless while letting you (once you're comfortable) reach into some pretty tricky territory.

Bjarne Stroustrup developed C++ in 1979 at Bell Labs as an enhancement to the C programming language and named it C with Classes. In 1983, it was renamed C++. The idea of creating a new language originated from Stroustrup's experience in programming for his Ph.D. thesis. C++ is one of the most widely used programming languages, and is used to create programs and systems, from video games to software for the space shuttle to other programming languages.

Basic Variable Types

The basic variable types in C++ are more or less the same as the variable types in Arduino with one fairly crucial difference: where Arduino uses the byte type, C++ uses the char. This is worth mentioning because at some point you'll probably encounter some code that does something strange like get all the bytes from a JPEG file into a char array. The reason that the bytes of a JPEG are stored in char variables is that the char stores exactly 1 byte of information, which can be a character like *B* (hence the name *char*); a small number, up to 32,647; or some bytes, like a pixel in a JPEG. Just be aware that you'll see the char type used in some places where the information being represented isn't an alphanumeric character.

These are the other most common C++ variable types:

`bool`
: For storing `true`/`false` values

`int`
: For storing integer numbers, for example, `1` or `89`; in all likelihood, this has a maximum value of `32767` on your computer

`long`
: For storing large integer values, for example, `3831080`; in all likelihood, this has a maximum value of `2147483647` on your computer

`float`
: For storing floating-point numbers, for example, `3.14` or `0.01`

`char`
: For storing character values, for example, `'f'` or `'g'`

`string`
: For storing strings of characters, for example, `"C++"` or `"openFrameworks"`

Arrays

Arrays in C++ are identical to arrays in Arduino because Arduino is written in C++. To create an array, give the type of object that the array will store and a length for the array:

```
int arr[5] = { 5, 10, 15, 20, 25 };
```

One thing to note is that you can't initialize an array like this in the class declaration in your *.h* file. You need to declare the array first: `int arr[5];` and then assign values to it in the `setup()` method of your application. There are other ways to create an array as well, using static variables, but this is probably the most common approach. This should be pretty familiar; if it isn't, refer to the discussion of arrays in Chapter 2.

Methods

Methods have signatures:

```
returnType methodName(params) { }
```

Methods can be overloaded:

```
String overloadedMethod(bool b);
String overloadedMethod(char c);
String overloadedMethod(String s);
```

Each of these methods will be treated as a separate method, although this isn't extremely common. In oF, it does crop up now and again, and it's important to know what's going on if you see it.

Methods can be scoped to an object or be static. This might be a little complex; then again, it might be a cinch for you to understand. It isn't a vital aspect of working with oF, but it is something you might come across, and it is helpful to understand. To start, think back to the idea of scope. Sometimes methods or variables are scoped to a class. This means you can't call the method without an instance of a class. For instance, an oF program has a `draw()` method. You can't call `draw()` without having a program in which to draw. That's fairly simple, isn't it?

```
ofBaseApp myApp;
myApp.draw();
```

So far, so good. Imagine you have a method you want to call without making any objects at all, for instance, the local date and time. You don't want to have to make a date object just to get the time; you just want to know what time it is. In those cases, you use what are called *static methods*. Now, this is a sticky topic with a lot of nuances to it, but *generally* (and for our purposes) this means a method is available without being called on an object. This means that calling a static method is *not like* the previous example where the `draw()` method of a particular instance of `ofBaseApp` (you'll learn exactly what this class is a little later on in this chapter) is being called:

```
myApp.draw(); // calling a method of an *instance*
```

Sometimes, however, you'll see a method being called like this:

```
NameOfClass::nameOfMethod();// calling a method of a *class*
```

For instance, this might be something like the following:

```
// this calls the static method of the ApplicationTime class
// the method is the same for every ApplicationTime instance
// and it always does the same thing
ApplicationTime::staticTimeMethod();
```

You look around and don't see any instance of `ApplicationTime` anywhere, but the method is being called and works fine. What gives? This means the `ApplicationTime` class has a static method called `staticTimeMethod()` that can be called anywhere, at any time, without having to create an instance of the `ApplicationTime` class.

The `ApplicationTime` class would probably look like this:

```
class ApplicationTime {

    public :
    static int staticTimeMethod();  // can be accessed from anywhere
};
```

The `static` keyword in front of that method marks it as being accessed from anywhere by using the name of the class, followed by two colons:

```
ApplicationTime::staticTimeMethod()
```

You might not ever need to use static methods in your code, but you probably will run into it them some point. As with many other things in this book, static methods have a lot of very subtle and complex nuances, and this book isn't the place to cover them.

Classes and Objects in C++

If you skipped over it, or weren't aware that it existed, now would be an excellent time to review the section "C++: Classes and Files" on page 139 in Chapter 5. Generally, and particularly in oF, a class is stored in two separate files, an *.h* file that *declares* all the variables and methods of the class, and a *.cpp* file that *defines* all the methods of a class.

A class is declared in the *NameOfClass.h* file, as shown here:

```
class NameOfClass {
public:
    void publicMethod(int);
private:
    int privateMethod();
}
```

In the *NameOfClass.cpp* file, the class is defined as follows:

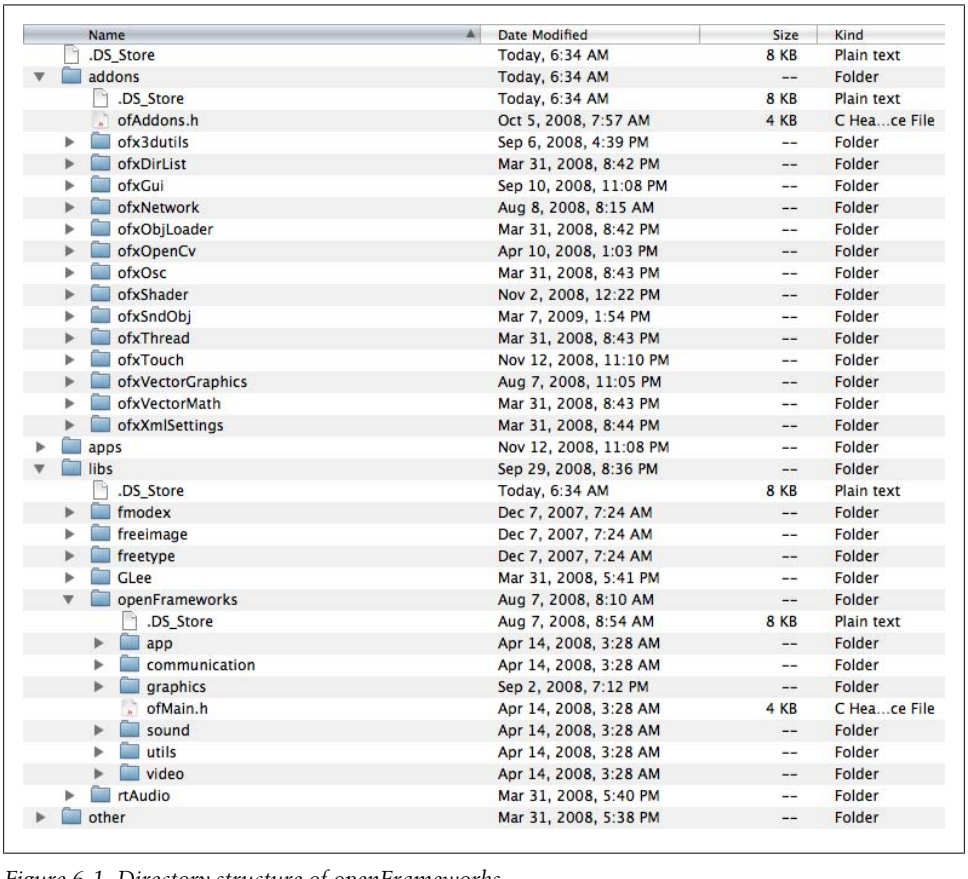
```
// we have to include the header file
#include "NameOfClass.h"

// all class declarations require the name of the class
// followed by :: to ensure that the method is included
// with the class
void NameOfClass::publicMethod(int i) {
//do some stuff
}
int NameOfClass::publicMethod() {
//do some other stuff
}
```

You should make note of four elements here: the class is declared in the *.h* file, the methods are declared but not defined in the *.h* file, the *.h* file is included in the *.cpp* file, and all the methods in the class use the name of the class and double colons before the method definition. If this seems unfamiliar at all, review Chapter 5.

At this point, you're ready to start exploring openFrameworks. Again, although C++ is complex and sometimes a bit obtuse, openFrameworks is designed to make working with C++ as easy and straightforward as possible so that you can get started experimenting and making your projects quickly.

Getting Started with oF

Getting started with oF is as simple as downloading the appropriate package for your computer, compiling it, and starting to look through the sample programs that have been built by the oF team. These should appear in something like the configuration shown in Figure 6-1.



Name	Date Modified	Size	Kind
.DS_Store	Today, 6:34 AM	8 KB	Plain text
addons	Today, 6:34 AM	--	Folder
.DS_Store	Today, 6:34 AM	8 KB	Plain text
ofAddons.h	Oct 5, 2008, 7:57 AM	4 KB	C Hea...ce File
ofx3dutils	Sep 6, 2008, 4:39 PM	--	Folder
ofxDirList	Mar 31, 2008, 8:42 PM	--	Folder
ofxGui	Sep 10, 2008, 11:08 PM	--	Folder
ofxNetwork	Aug 8, 2008, 8:15 AM	--	Folder
ofxObjLoader	Mar 31, 2008, 8:42 PM	--	Folder
ofxOpenCv	Apr 10, 2008, 1:03 PM	--	Folder
ofxOsc	Mar 31, 2008, 8:43 PM	--	Folder
ofxShader	Nov 2, 2008, 12:22 PM	--	Folder
ofxSndObj	Mar 7, 2009, 1:54 PM	--	Folder
ofxThread	Mar 31, 2008, 8:43 PM	--	Folder
ofxTouch	Nov 12, 2008, 11:10 PM	--	Folder
ofxVectorGraphics	Aug 7, 2008, 11:05 PM	--	Folder
ofxVectorMath	Mar 31, 2008, 8:43 PM	--	Folder
ofxXmlSettings	Mar 31, 2008, 8:44 PM	--	Folder
apps	Nov 12, 2008, 11:08 PM	--	Folder
libs	Sep 29, 2008, 8:36 PM	--	Folder
.DS_Store	Today, 6:34 AM	8 KB	Plain text
fmodex	Dec 7, 2007, 7:24 AM	--	Folder
freeimage	Dec 7, 2007, 7:24 AM	--	Folder
freetype	Dec 7, 2007, 7:24 AM	--	Folder
GLee	Mar 31, 2008, 5:41 PM	--	Folder
openFrameworks	Aug 7, 2008, 8:10 AM	--	Folder
.DS_Store	Aug 7, 2008, 8:54 AM	8 KB	Plain text
app	Apr 14, 2008, 3:28 AM	--	Folder
communication	Apr 14, 2008, 3:28 AM	--	Folder
graphics	Sep 2, 2008, 7:12 PM	--	Folder
ofMain.h	Apr 14, 2008, 3:28 AM	4 KB	C Hea...ce File
sound	Apr 14, 2008, 3:28 AM	--	Folder
utils	Apr 14, 2008, 3:28 AM	--	Folder
video	Apr 14, 2008, 3:28 AM	--	Folder
rtAudio	Mar 31, 2008, 5:40 PM	--	Folder
other	Mar 31, 2008, 5:38 PM	--	Folder

Figure 6-1. Directory structure of openFrameworks

Figure 6-1 shows the Mac OS X Finder, so obviously if you're on Windows or Linux, you'll see something different, but the general idea is the same. The oF folder contains the following folders:

addons

This contains all the added-on features for openFrameworks that have been contributed by users over the past year or so. These libraries change often, and new ones are added frequently, so it's important to remember that these are dynamic if you're planning on using these libraries in a program because you need to explicitly include them in your program.

apps

This is where your programs should be stored. If you have a strong preference, you can organize them in other ways, but that would go against the flow and would break some of your code. The example programs that use the core oF libraries are stored here, as are the example programs that show you how to use each of the add-on libraries. These are some of your best learning resources because they give you wonderful starting points for exploring oF and C++, and because they're usually well commented and documented.

libs

This is where the libraries that oF relies on are stored. These range from external libraries like fmodex, which is used for two- and three-dimensional sound manipulation, to the GLee library, which is used for OpenGL graphics. Most important, you'll see the core *openFrameworks* folder, which stores the header files that define all the core oF functionality in six folders. As you find your ideas and techniques becoming more and more sophisticated, you'll probably need more libraries than those provided here.

openFrameworks

The *openFrameworks* folder contains the core of the oF framework within six folders. The *app* folder contains the header file for the `ofBaseApp` class, which is the base program class of any oF program. The `ofBaseApp` class lets you easily toggle between full-screen and window views, set up code to handle mouse movements and keyboard presses, and integrate the simplified drawing routines with the graphics environment of your operating system. The *communication* folder contains tools for communicating over serial ports that can be used to communicate with the Arduino controller. The events folder contains ofEvents, which handles sending and receiving events in an oF application.

The *graphics* folder contains five classes, each of which is broken into a *.h* and *.cpp* file: *ofBitmapFont*, for working with bitmapped fonts, *ofGraphics* for working with 2D drawing, *ofImage* for working with bitmapped image data, *ofTexture* for working with OpenGL textures, and *ofTrueTypeFont* for working with fonts.

The *sound* folder contains two classes for working with sound: the `ofSoundPlayer` that is a more simplified class for creating and playing sound files and the `ofSoundStream` class that provides more low-level access to the sound capabilities of your computer.

The *video* folder contains code for capturing video from a video camera and for playing back captured video or video files loaded from your computer. We'll talk more about all these topics in later chapters, which are organized thematically.

Finally, the *utils* folder contains useful code for helping you work with math, useful code to get and measure time, and some core data types that are used throughout the program.

One of the first steps you'll want to take after you've downloaded oF is to go to the *apps/examples* folder and try running one or two of these programs. After you've run a program, open the code files, and explore how the program is put together. Keep an eye out for the references to `ofBaseApp`, and in the *.h* header file, look at the `#import` statements. You might not understand everything in these sections, but exploring these files is a good way to get familiar with what the libraries are, how they are imported, and how an oF program is structured.

oF is built out of multiple layers of libraries and code that uses those libraries. As you grow more comfortable with oF and with C++, you'll probably delve deeper into the official oF libraries and the underlying third party libraries that they use. At the beginning though, you'll just want to focus at the top. The diagram in Figure 6-2 shows the structure of an oF application.

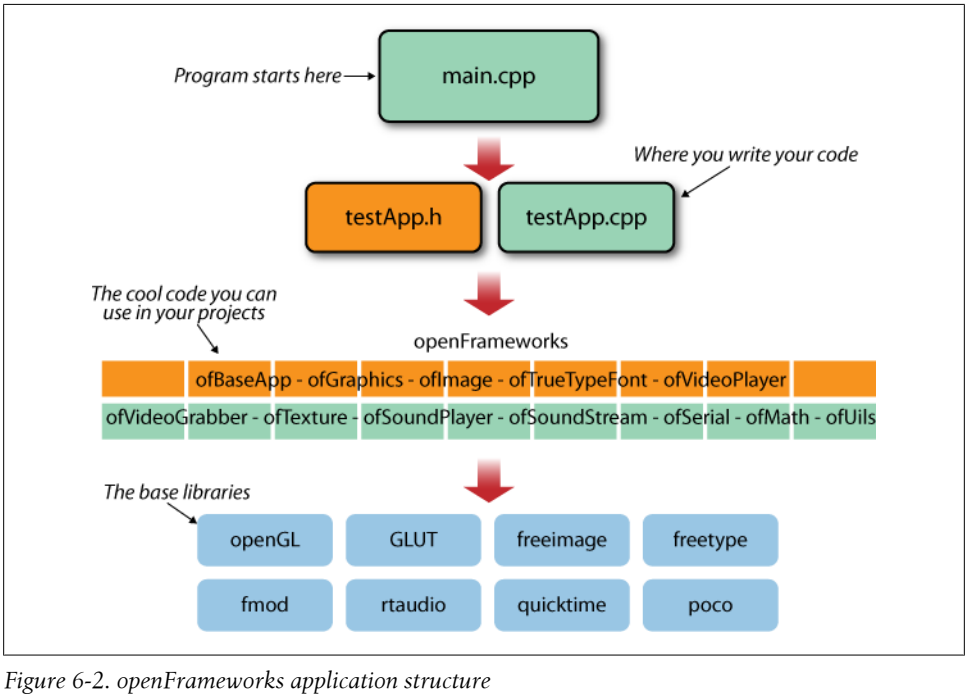


Figure 6-2. openFrameworks application structure

As you can see, your application sits atop the openFrameworks libraries, which sit atop a series of third-party libraries. As you learn more and more about oF and how all the libraries interrelate, you'll be delving deeper into both the oF and third-party libraries to do more.

Interview: Theo Watson

Theo Watson, along with Zach Lieberman, is head of the openFrameworks project. Theo's work ranges from creating new tools for artistic expression, experimental musical systems, and immersive environments with full-body interaction. He lives in Amsterdam.

Can you talk a little bit about how you got started programming and working with interactive art?

Theo Watson: Before I was doing programming work, I was really obsessed with sound. I was at Parsons, and I was skipping a lot of classes because I was so interested in working with music and, more particularly, with making instruments. I started off doing Max/MSP and was creating synthesizers that would take radio signals and use them to modulate the sounds that the synthesizers would make. That way, the synthesizers wouldn't just be making the sounds that you thought they would be making, but would also be driven by where you were based on which radio stations could be picked up in that area. I was very stubborn about doing any visual work; I wanted to keep everything based in sound.

Later, I had a class with Zach Lieberman called Audio Visual Synthesis. I had lost the use of my arms over the summer because of carpal tunnel syndrome, so I couldn't use a computer, and I wanted to make something that I didn't need to actually touch—something I could control using just my voice. That got me thinking...what if I didn't even need to control it at all? It would create a performance for itself, and then it could not only create a performance for itself but appreciate itself as well. It remembers what it has played so it builds up an interpretation of that. And because the microphone information is imperfect, it creates these very subtle variations that were stored and carried into the next performance.

When I played it back through MIDI using the information as a piano, it sounded very much like a jazz pianist. It totally blew my mind. It wasn't anything intentional, because of the feedback loop that was delayed and imperfect; what it was hearing was quite different from what it was playing.

Do you see your artistic work and your work on oF as being distinct from one another, or are they part of the same project for you?

Theo: We approached openFrameworks and our own work from the perspective of having an idea and really wanting to make it. I'm really taken with the idea of learning the skills as you go and doing whatever it takes to achieve your vision. I really firmly believe that you can get something unexpected back as feedback from your art.

From the process of making something, I can have something that I want to do in an art project and not have any idea how to accomplish that and then treat that as a process

of learning how to do something. As something that shapes my ideas about what I'm doing, there's this constant back and forth and feedback. Neither Zach nor I are engineers; we're just people with funny ideas who have a bit of a nerdy background and a technical fascination with learning what we need to know.

What kinds of things have you done with openFrameworks?

Theo: It's really good that both Zach and I make projects; it gives us a real impetus to make sure that everything works because we're using oF ourselves. In my latest project, for example, I put it together in about two weeks, and if it weren't for oF, it would have taken about two months to make. It wasn't something that I was always excited about, but it's beginning to become more appealing to me now.

There were things that I had never used in oF. For example, I was trying to figure out how to use 3D vector rotations, and then I just happened to glance into `ofVectorMath`, and there was a bunch of code to help me. I was sitting there with a pencil and a piece of paper, and then it hit me: "Oh, yeah, I know there's something like this." I looked, and there it was, so I didn't have to figure out. I could spend time on the more interesting stuff. So, it's not like we're just making sure the code compiles and then putting it out there; we're using it in installations that have to run sometimes for weeks straight without crashes, so oF is just as important to us as it is to anyone using it.

Zach and I worked together on LaserTAG. LaserTAG was a really nice project because it reached so many people—both people who played with it and people who downloaded the software, set up the projector, and used the project themselves. It's working at a different scale both in terms of physical size and in terms of audience size. It's something that people are getting really excited about, and that's really amazing to me. I want to push those ideas and put people into situations where they can shape their environment and experience their environment. I think Zach and I are going to perhaps collaborate on something else soon, aside from oF, of course.

Can you talk a little bit about the philosophy of openFrameworks?

Theo: The motive of oF is to get rid of all the frustration of getting a window, getting something drawn in the window, and getting an image displayed in it. For some people first starting out, getting a window displayed and getting a shape drawn in it are really big deals. You have to deal with all this stuff that really isn't going to be interesting to you at all. You really should be thinking about how to make your idea richer, not "Why am I having this strange windowing error?"

When learning to code, you can start anywhere. My entry was with PHP and with Max/MSP. What I loved about Max/MSP that is a bit underrated was that in some ways it teaches you to be a good programmer because it teaches you how to break a problem into discrete parts. So, it helps you visualize how to break something into parts. Lots of times when I run into people who've never programmed before, a big hurdle is that they frequently don't know how to break something into the correct kinds of parts or they underestimate what it means to make something.

How did openFrameworks get started?

Theo: The idea of oF came about because Zach was teaching these classes, and in every class he was essentially trying to do the same thing every time—to get people up to

speed, so to speak, with programming, with building an application, with being able to understand graphics and how graphics are drawn on the computer screen, with playing a sound, with capturing a sound, and so forth.

It actually started that summer, and I wasn't involved in it at all, because I was working on my thesis all that year. On that project, I was very determined that I write every line of code myself, and I ended up writing all the computer vision code for my project. I wrote every single function—including contour detection—that I used in that project. I mean everything, so I was very much busy at that point, working on my own stuff.

But then what happened was, after Parsons, we had a very rough outline of oF that was in use, and it was great for these students, so you could say "use `ofCircle`" instead of needing to work with cosine and sine, for example, and your class could be more interesting. It became something where we were incorporating more and more functionality, and it was stuff that was more and more interesting to me. When someone asked, "How do I do this?" and we didn't know, we would end up spending three or four days trying to solve the problem, and it would be a really valuable thing to learn. Later I'd find myself using that same information or knowledge in my own work, so it became something I was doing to teach myself as much as because it was something we wanted to add to oF.

What kinds of people have you found using oF?

Theo: The forums have sprung up in a really interesting way, because there's such a wide range of kinds of things that people are talking about and backgrounds that people are coming from. A lot of people who are a lot smarter than us are participating, which is great because it's really helping to lower the barrier to entry, because not only is there this framework that we've put together but additionally there are people other than us to answer questions. For example, there's Arturo Castro. We wanted to do a Linux version of oF, but neither of us really knew a lot about Linux, especially video and audio capture in Linux, but he did, so he really took over that part of it, which is amazing because the community around this project is allowing us—all of us involved with oF or using oF—to make and use things that maybe we aren't as familiar with as we'd need to be without it. So, we're very lucky.

We really want people to use it in the most unusual way they can imagine. We don't want it to be just about using computer vision, where people are going to say, "OK, I need to do computer vision, so I'll use oF." We're quite interested in things that use machinery, sound, or nontraditional means of interacting. I want to take it away from computer vision, graphics, and installation work and allow people to work more richly with sound.

Are there particular types of projects that you want to see people using openFrameworks for or particular ways you want to see people using it?

Theo: We don't want to have to tell people, "This is the way you have to use it." We just want to see it used in new and different ways, so every time there's a new project, I feel like there's very often a new plug-in for that project that's born out of it. As long as people don't claim they wrote oF, we're fine with anything they do with it. If that is something having to do with maximizing profits in a design project, we're fine with

that. We don't want to force people to share their code; we won't use GPL, because we don't want to restrict what people can do with it. We want them to feel they can do whatever they want with it. We think it should be up to the end user—to the people playing with it—to decide what they need.

Touring an oF Application

The core of an oF program is the main application class, `ofBaseApp`. Since oF is a bare-bones C++ library, the best way to understand it is to look at the code, and the first place to look is the program. Everything you make in oF will have an `ofBaseApp` class somewhere in it.

Methods

All the methods in the `ofBaseApp` class are event-handling methods. This means they are triggered in response to events that happen within your program, such as the user moving the mouse, the user pressing a key, the program quitting, or your program redrawing. At its simplest, working with oF is adding new code to the appropriate method and waiting for the program to call your methods. If you want to work with the users mouse actions, add some code to the `mouseMoved()` method of the `ofBaseApp`. The core oF framework handles creating the window and adding the appropriate code behind the scenes. To do something when these events occur, you simply need to create a class that extends `ofBaseApp` and add the appropriate methods. If that sounds a little intimidating, don't worry, because we'll walk you through it after looking at each of the methods in the `ofBaseApp` class.

The `setup()` method is called when the program first starts up, the same as in Arduino and Processing. If something has to run only once or it is important to run at the beginning of your program, it should be in `setup()`. Also, if you're defining variables in your *.h* file, you should initialize them. For instance, in a *.h* file you might have:

```
int numberOfVideos;
```

In your setup method, you should initialize that if you're going to need it once your application starts running:

```
numberOfVideos = 5;
```

The `update()` method is called just before the `draw()` method. This is important because in oF you generally want to put any data processing you need to do in the `update()` method so you're drawing with new data. If you need to check and see, for example, whether any new sound is coming from a microphone before you draw the sound wave, you'll want to do that in the `update()` method.

The `draw()` method is called when the program draws its graphics, much like in Processing. In contrast with Processing, however, in oF you should have drawing code only in your `draw()` method to keep your program running smoothly.

The `exit()` method is called when the program exits and closes down. If you want to do any final processing or data logging in your program, or to free memory, which we'll discuss later, this is the place to do it.

The `keyPressed()` and `keyReleased()` methods are callbacks that handle any key actions that the user makes:

- `keyPressed(int key)`sends the code of the key just pressed as an integer
- `keyReleased(int key)`sends the code of the key just released as an integer

The `ofBaseApp` class defines four callback methods to handle mouse movements. You can add any code to handle these events to these methods. The `mouseMoved()` method passes in the current `x` and `y` position in pixels of the mouse. The `mouseDragged()` and `mousePressed()` methods pass in the `x` and `y` positions and also the button that has been pressed. The `mouseReleased()` method is triggered whenever either mouse button is released:

```
mouseMoved(int x, int y)
mouseDragged(int x, int y, int button)
mousePressed(int x, int y, int button)
mouseReleased()
```

The `audioReceived()` method is a little more complex than the other methods because it receives a pointer to the stream of audio data and because dealing with a pointer is a little tricky. This requires that you remember what a pointer really is—a piece of information that tells you the location of something:

```
audioReceived(float * input, int bufferSize, int nChannels)
audioRequested (float * output, int bufferSize, int nChannels)
```

Imagine you get some data in the `audioReceived()` callback. That data represents the waveform of the sound that the computer's microphone has captured in a list of floating-point numbers, maybe something like 0.000688 or –0.000688 for a loud noise, or 0 for silence. The numbers represent the waveform of the sound, just like a waveform display that you might see on an oscilloscope or in recording software. Instead of sending all the data—all those floating-point numbers—to `ofBaseApp`, the program sends just a single value: the location in memory where all this data is stored. That means instead of needing to send a few hundred values, the program needs to send only one (Figure 6-3), saving the space that would be needed if it had to copy all the data every time.

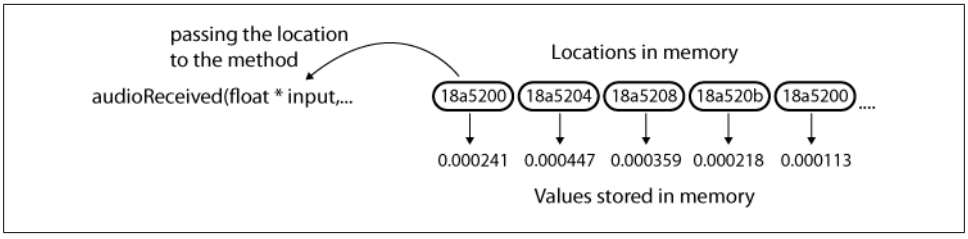


Figure 6-3. Passing a pointer to an array of data for sound processing

So, you simply pass the location of the first item in the array of sound data and then start processing the data in the `audioReceived()` method. You'll learn more about this in Chapter 7, so we'll leave it be for the moment. You might also want to look back at Chapter 5 in the section "Pointers and References" on page 144.

Variables

The two variables that `ofBaseApp` defines are the `mouseX` and `mouseY` values:

```
int mouseX, mouseY;
```

`mouseX` and `mouseY` let you always determine the location of the user's mouse. Since we're talking about the mouse position, we'll point out two very handy methods you can call to figure out the size of the screen where your program is being displayed. This might not be as important if you always know the computer that your program is going to be running on, but if you're going to be moving it around or you don't feel like measuring your screen, you can use these methods to find the height and width of the screen in pixels:

```
int ofGetScreenHeight();
int ofGetScreenWidth();
```

You'll be using these later in this chapter to help control the speed of a movie file with the mouse position.

Now that you have a basic idea of what is included in an oF program, you'll create one of your own.

Creating "Hello, World"

In this section, you'll create a simple program that will print the words "Hello oF World" to the screen. As described in Chapter 5, your program will be spread across three different files, *HelloWorldApp.h*, *HelloWorldApp.cpp*, and *main.cpp*. In all the oF downloads, you can use the empty project templates so you don't have to create a new project yourself. Simply copy the *emptyProject* template and give it a new name, and you have all the classes, libraries, and paths you need to create your program.

All oF programs extend the `ofBaseApp` class. Any methods you want your program to use will be defined in the *.h* file of your application class. In this case, you need to use only the `setup()` and `draw()` methods. Any methods you do not extend in your program are simply left as they were in the original program.

Here's the *HelloWorldApp.h* file:

```
#ifndef _HELLO_WORLD
#define _HELLO_WORLD

#include "ofMain.h" // we need to include all the oF files, linked here

class HelloWorldApp : public ofBaseApp{

    public:

        void setup(); // we need this to load the font
        void draw(); // we need this to draw the font to the screen

        ofTrueTypeFont  franklin; // this is the font that we'll be loading

};

#endif
```

These two lines may jump out at you from *HelloWorldApp.h*:

```
#ifndef _HELLO_WORLD
#define _HELLO_WORLD
```

These two lines are what are called *preprocessor instructions*, and they're important to the compiler that's going to create your oF program from your code, because it tells the compiler to make the class only once. In a simple program like the one you have here, this isn't a big deal, but in more complicated programs, the compiler may try to compile the class multiple times, which would cause all kinds of problems.

It's interesting to note the declaration of the font object. To use a font in oF, you need to create an object that can load a font file, in this case, a TrueType font (TTF) file, and create the correct characters to display on the screen from that font file:

```
ofTrueTypeFont  franklin
```

The next part of your program is the definition of the methods of the `HelloWorld` class, which in this case loads the font from a TTF file, sets the background color of your program, and then actually writes the text to the screen. This all happens in the *.cpp* file. Here's the *HelloWorldApp.cpp* file:

```
#include "HelloWorldApp.h"

void HelloWorldApp::setup(){
```

To use the font, you need to load the file. Since you need to do this only once, you do it in the `setup()` method. The `loadFont()` method takes two parameters: the name of the file and the size of the font you want to use. Each font size must be loaded separately,

so if you wanted to have your Franklin Black font in 32 and 64, you'd need to create two separate `ofTrueTypeFont` objects to store those settings:

```
        // load our font and give it a font name
        franklin.loadFont("frabk.ttf",32);
```

Next, you set up the background of your program, passing in the red, green, and blue values, all set to `255` here, making your program's background white:

```
        ofBackground(255,255,255);
    }
```

The `draw()` method redraws the graphics of the program, just the same as in a Processing program:

```
    void HelloWorldApp::draw(){
        ofSetColor(0, 0, 0);
        franklin.drawString("HELLO WORLD!", 100,380);
    }
```

The `setColor()` method sets the color of any drawing that the program will do, and in this case, that drawing is going to be the text. Since you're passing `0`, `0`, `0`, which is RGB black, the text will appear in black. The `drawString()` method writes the text to the location on the screen specified by the x and y coordinates.

Now that the program is ready, you need to run it. In C++, as you may recall, you do this in a `main()` method, which is a special method that the compiler looks for when it starts compiling your code. In an openFrameworks program, this method is always in the *main.cpp* file. For an oF program, the `main()` method should set up the window size and then start the program. The framework handles everything else.

The first step you'll see here is the setup of the window. You can specify a window with a size, or you can just say `oF_FULLSCREEN` to make the program fill the screen of the computer that it's being run on. The next line of code contains the call to the `ofRunApp()` method that starts the program. You pass it a new instance of the program, and the `ofRunApp()` method takes care of starting the program and calling its `update()` and `draw()` methods.

Here's the *main.cpp* file:

```
    #include "ofMain.h"
    #include "HelloWorldApp.h"

    int main( ){

        // can be oF_WINDOW or oF_FULLSCREEN
        // pass in width and height too:
        ofSetupOpenGL(1024,768, oF_WINDOW);// <-------- setup the GL context
        // this kicks off the running of my app
        ofRunApp(new HelloWorldApp);

    }
```

And that's the "Hello, World" of openFrameworks. Now, the preceding code sample contained one method that did not receive the attention it merits because it raises an important point about how oF applications are organized. Take a look at the call to the `loadFont()` method:

```
franklin.loadFont("frabk.ttf",32);
```

This is actually loading the *ttf* file from the *data* folder into your application. Every application can have a *data* folder to store any images or other files it might need to load.

openFrameworks assumes that a folder called *data* is in the same directory as your program. Whenever you load an image, font, Apple QuickTime movie, or other kind of file, by default your oF program will look in the *data* folder. Unless you have a good reason to do so, you should always store data within this folder to leverage the work that the creators of oF have put into the framework.

You've now seen how to write your first oF application.

Drawing in 2D

To begin, drawing in oF is quite similar to drawing in Processing; you set the color that the drawing will use, set the fill mode of the drawing (whether the shape will be filled in with color), and then use one of the predefined drawing methods to draw the shape to the screen. For example, to draw two circles to the screen, you would simply add the following methods to the `draw()` method of a program:

```
void simpleGraphics::setup(){
    ofSetCircleResolution(100);
    ofBackground(255,255,255);
}

void simpleGraphics::draw(){

    ofSetColor(0xFF0000);
    ofFill();        // draw "filled shapes"
    ofCircle(100,400,100);

    // now just an outline
    ofNoFill();
    ofSetColor(0x333333);
    ofCircle(400,400,200);

}
```

The *.h* header file for the class would look like so:

```
#ifndef _SIMPLE_DRAW
#define _CHAP_SIX_GRAPHICS
```

```
#include "ofMain.h"
#include "ofAddons.h"

class simpleGraphics : public ofBaseApp{

    public:

        void setup();
        void draw();

};

#endif
```

Since you're not overriding any of the other methods of the `ofBaseApp` class, you don't need to include them in the header file for your class. The result will look something like Figure 6-4.

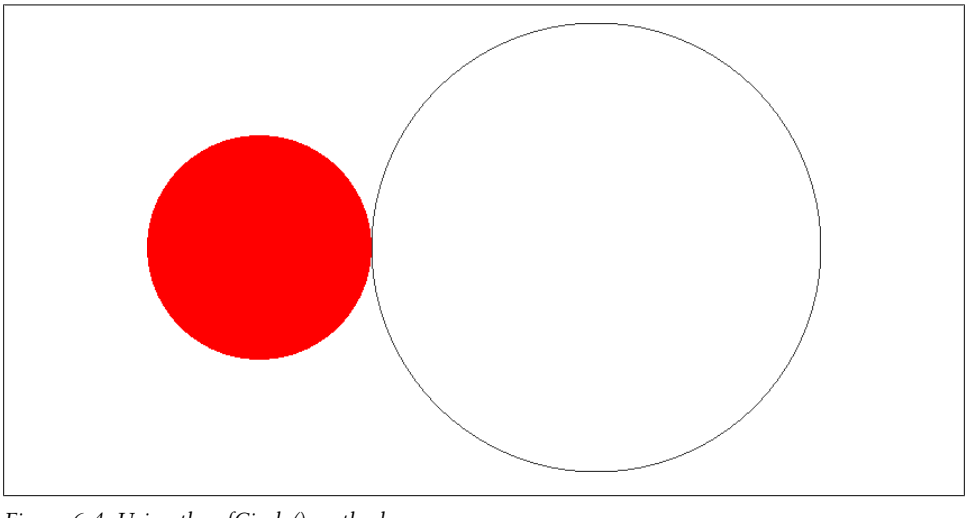


Figure 6-4. Using the ofCircle() method

Setting color in oF is similar to the way you set it in Processing. The `ofSetColor()` method takes either RGB values, RGB and alpha values, or hex color values:

```
void ofSetColor(int r, int g, int b); // 0-255
void ofSetColor(int r, int g, int b, int a); // 0-255
void ofSetColor(int hexColor); // hex, like web 0xFF0033;
```

Setting Drawing Modes

The drawing mode of a program indicates whether lines or shapes should have fills. In oF, you set these drawing modes using one of two methods. If you want your drawing to have a fill, that is, a color that fills in the shape, you call the method `ofFill()` before doing any drawing; if not, then call `ofNoFill()`:

```
ofSetColor(0xCCCCFF); // set our color to light blue
ofFill();        // drawing will use "filled shapes"
ofRect(100, 100, 200, 200); // draw a rectangle 200x200 pixel

// now just an outline
ofNoFill();
ofRect(400, 100, 200, 200);
```

This code draws two rectangles that will sit next to one another. You'll see what that looks like in a few moments after you add a little more detail to the rectangles. oF provides quite a few convenience drawing methods for drawing simple 2D shapes:

```
void ofTriangle(float x1,float y1,float x2,float y2,float x3, float y3);
void ofCircle(float x,float y, float radius);
void ofEllipse(float x, float y, float width, float height);
void ofLine(float x1,float y1,float x2,float y2);
void ofRect(float x1,float y1,float w, float h);
```

Putting a few of these methods together, you can make a simple drawing (as shown in Figure 6-5) that overlaps squares and circles with varying alpha values.

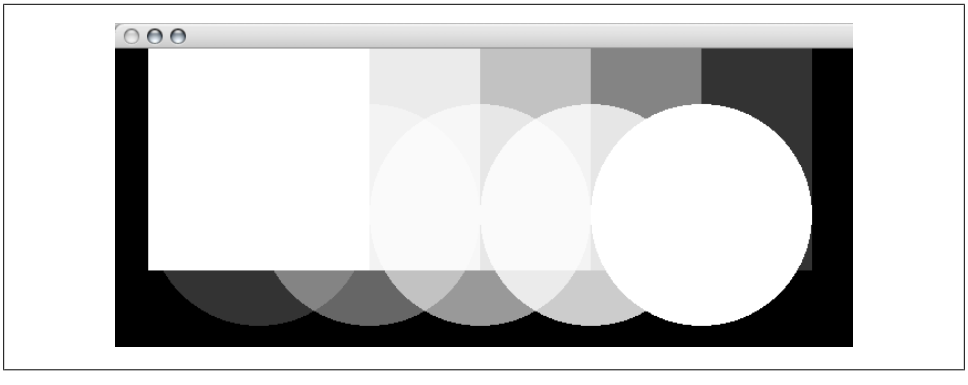


Figure 6-5. Drawing with varying alpha values

```
void simpleGraphics::setup(){
    ofSetCircleResolution(100);
    ofBackground(0, 0, 0);
}

void simpleGraphics::draw(){
    ofEnableAlphaBlending();
    int i = 0; // make a variable to use for our loop
    ofFill(); // draw filled shapes
    while(i < 6) {

        ofSetColor(255, 255, 255, 255 - (i * 51)); // decreasing alpha
        ofRect((i * 100)+30, 0, 200, 200); // draw a rectangle 200x200 pixel

        ofSetColor(255, 255, 255, i * 51); // increasing alpha
        ofEllipse((i * 100)+30, 150, 100, 100); // draw an ellipse
```

```
            i++;
        }

    }
```

Drawing Polygons

The Open Graphics Language (OpenGL) handles all the drawing in oF. Chapter 13 in this book is dedicated to OpenGL, so in this chapter you'll learn how to get oF to draw without touching any actual OpenGL code. Sometimes you don't want to draw a circle, square, ellipse, or triangle, and you don't want to delve into OpenGL. For this, you can use several polygon-drawing functions in oF to draw shapes by placing points and setting fills. This works much the same as drawing vectors in Adobe Flash, Adobe Illustrator, and many other paint programs: you place three or more points, and the program creates the fill between those points that defines the shape. In oF, you tell the program to begin to draw a polygonal shape, you add points (*vertices* in oF) that define the edges of the point, and then you close the shape, which triggers the filling in of your shape.

This method is called before the shape is drawn, telling the oF application that you're starting to draw a shape:

```
void ofBeginShape();
```

This method is called after the shape is drawn, telling the oF application that you're finished drawing the shape:

```
void ofEndShape(bool bClose);
```

This method takes a single parameter: a boolean that indicates that whether the shape should be closed, that is, whether or any space between the first point and the last point should just be closed up by OpenGL. The default is `true`.

The following places a vertex at the position on the screen indicated by the `x` and `y` values passed into it:

```
ofVertex(float x, float y);
```

When you place a vertex, OpenGL automatically connects the previous point (if one exists) and the next point (when one exists) via the location of the just added point with two straight lines.

If you don't want to draw straight lines (and sometimes you just won't), then you might want to use the `ofCurveVertex()` method:

```
ofCurveVertex(float x, float y);
```

This method places a vertex that a line will curve to from the previous point and that will be curved from to the next vertex. `ofCurveVertex()` has one tricky part: for reasons that have to do with the way OpenGL draws vertex curves, you need to add the first

and last vertices in the shape twice. The following code snippet shows how to use the `ofCurveVertex()` method:

```
ofSetColor(0xffffff);
ofFill();
ofBeginShape();
    ofCurveVertex(20, 20);// that's not a typo; you need to add the first
    ofCurveVertex(20, 20); // point twice
    ofCurveVertex(220, 20);
    ofCurveVertex(220, 220);
    ofCurveVertex(20, 220);// that's not a typo; you need to add the last
    ofCurveVertex(20, 20);// point twice as well
        ofCurveVertex(20, 20);
ofEndShape(false);
```

One drawback to the `ofVertexCurve()` method is that you have no control over how curved the curve is. To control the curviness of the curve, you'll want to use the `ofBezierVertex()` method, which lets you set the starting point of the curve and then define two control points (Figure 6-6) that will be used to define the shape of the line on its way to the next vertex:

```
ofBezierVertex(float x1, float y1, float x2, float y2, float x3, float y3)
```

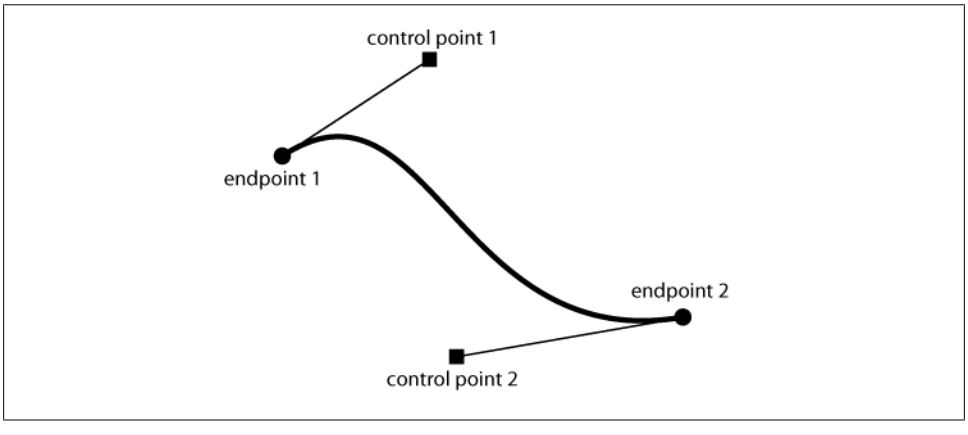


Figure 6-6. A Bezier curve

In the `ofBezierCurve()` method, the first two floats, `x1` and `y1`, set the location of the line; the next two, `x2` and `y2`, set the location of the first control point; and the last two, `x3` and `y3`, set the location of the second control point. This lets you create quite complex shapes by combining multiple Bezier curves. For example, the following code snippet in the `draw()` method of an oF program creates two moving Bezier curves:

```
ofEnableAlphaBlending();
float bez1X1 = 350+50 * cos(ofGetElapsedTimef()*2.0f);
float bez1Y1 = 200+100 * sin(ofGetElapsedTimef()/3.5f);
float bez1X2 = 400+30 * cos(ofGetElapsedTimef()*5.0f);
float bez1Y2 = 300+100 * sin(ofGetElapsedTimef());
```

```
float bez2X1 = 350+50 * cos(ofGetElapsedTimef()*2.0f);
float bez2Y1 = 200-100 * sin(ofGetElapsedTimef()/3.5f);
float bez2X2 = 400+30 * cos(ofGetElapsedTimef()*5.0f);
float bez2Y2 = 300-100 * sin(ofGetElapsedTimef());

ofFill();
ofSetColor(255, 0, 0, 127);
ofBeginShape();
    // first we create the point from which to begin drawing
    ofVertex(100,300);
    // then create the Bezier vertex
    ofBezierVertex(100,300,bez1X2,bez1Y2,650,300);
    ofBezierVertex(650,300,bez2X2,bez2Y2,100,300);
    // finish the shape
    ofVertex(650,300);
ofEndShape();

ofFill();
ofSetColor(255, 0, 0, 127);
ofBeginShape();
    ofVertex(100,300);
    ofBezierVertex(bez1X1,bez1Y1,bez1X2,bez1Y2,650,300);
    ofBezierVertex(bez2X1,bez2Y1,bez2X2,bez2Y2,100,300);
    ofVertex(650,300);
ofEndShape();
```

Figure 6-7 shows what the previous code will look like when run.

Figure 6-7. Drawing with Bezier curves

Displaying Video Files and Images

Now that you can draw some simple shapes with oF, you might be interested in loading and displaying video files and images.

Images

You can display images in an oF program by using the `ofImage` class, which handles most of the dirty work of loading an image into memory and displaying it on the screen. Though, as with all things, you can do this work yourself, but it's often easier to have oF do it for you until you really need to do it for yourself.

The `ofImage` class is a core class of oF, which means you don't have to use any `import` statements to include this class in your program. Simply declare an instance of the `ofImage` class in the header file of your program, and you're ready to go. The following is an example of a header file for an oF program that will use an `ofImage` class:

```
#ifndef _IMAGE_APP
#define _IMAGE_APP
#include "ofMain.h"

class imageApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        ofImage milkShakeImg;
        ofImage friesImg;
};

#endif
```

Most of the time when you're dealing with an image, you're dealing with image files: JPEG, PNG, and so on. To load an image from a file, call the `loadImage()` method, and pass the relative or absolute name of the file that you want to load as an argument. The *relative* filename is the file path from the program to the location of the image. The *absolute* name is something that starts with *C:\...* if you're using Windows and */Folder/...* if you're using OS X or Linux. Generally, you'll want to use the *data* folder within your oF program:

```
void loadImage(string fileName);
```

This loads the image into memory so that you can display it or manipulate it as you see fit. Once you've loaded the image, you need to draw it into the graphics window. You do this using one of the `draw()` methods of the `ofImage` class, like so:

```
void draw(float x, float y); // use the images default size
void draw(float x, float y, float w, float h); // specify image size
```

In the previous code the `x` and `y` parameters dictate where the image will be drawn in the program window. This overloaded method also lets you either use the default size of the image file you've loaded or set a new height and width for the image:

```
void resize(int newWidth, int newHeight);
```

To dynamically resize an image, in an animation, for example, use the `resize()` method to change the size of the image:

```
void saveImage(string fileName);
```

The `saveImage()` method lets you save an image to a file that you pass in as a string to the method name. You might want to do this if you're manipulating an image or creating a new image from a screen capture. You'll learn how to use a screen capture later

in this chapter. If you want to look at doing pixel-level manipulations with an image file, look ahead to Chapter 10.

Video

You can work with video from a prerecorded movie file and from a live video source. The first option is to display a video that has been saved in a QuickTime, WMV, or MPEG file. You might want to load the file, play it, rewind it, speed it up, or even grab and manipulate the individual pixels that make up the image. oF helps you do all these tasks with its `ofVideoPlayer` class, which you'll learn about in this section. The second option for working with video is using a camera attached to your computer. You might want to display the video from your camera, resize the display, or manipulate the pixels of your video to create new images. oF helps you do all these tasks with its `ofVideoGrabber` class, which you'll learn about in Chapter 10.

Working with video is a huge topic, and because oF lets you do so many things by extending its functionality and integrating so many different libraries, this chapter is only scratching the surface of the topic. We'll save the majority of the other functionality for the later chapters in this book and concentrate on the basics in this chapter.

Creating a program that loads a movie is a cinch. Simply declare an instance of the `ofVideoPlayer` class in an *.h* file for the program, and then load the movie in the `setup()` method of the program:

```
#ifndef _VIDEO_APP
#define _VIDEO_APP


#include "ofMain.h"
#include "ofAddons.h"

class videoApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        ofVideoPlayer player;
        float screenHeight;
        bool isPaused;

        void mouseMoved( int x, int y );
        void keyPressed  (int key);
};

#endif
```

Once you have declared all the methods and variables of the `videoApp` class, you can move on to defining it in the *videoApp.cpp* file. You'll use two very important methods here. The first is the `draw()` method for `ofVideoPlayer` that draws the pixels from the current frame of the movie onto the screen. This needs to be called in the `draw()` method of your program. The second is the `idleMovie()` method, which in essence tells the movie to load the next chunk of data from the file and prepare to display itself. This is called in the `update()` method of the oF app, keeping with the philosophy of doing all nondrawing related processing in the `update()` method before the `draw()`. If you omit this method, the movie will play fine, but skipping around in the movie won't work, so it's important to remind the player to do its internal housekeeping:

```
#include "videoApp.h"

void videoApp::setup()
{
    ofBackground(255,255,255);
    screenHeight = float(ofGetScreenHeight());
    player.loadMovie("a_movie2.mpeg");// this loads from the data folder
    player.play();
    isPaused = false;
}

void videoApp::update()
{
    player.idleMovie();
}

void videoApp::draw()
{
    player.draw(20, 20);
}

void videoApp::mouseMoved( int x, int y )
{
    float yPos = (mouseY / screenHeight ) * 2.0;
    player.setSpeed(yPos);
}

void videoApp::keyPressed  (int key)  {
    if(    isPaused ) {
        isPaused = false;
    } else {
        isPaused = true;
    }
    player.setPaused(isPaused);
}
```

You won't recognize a few methods here unless you've done a little creative investigation into the header files already. The `ofVideoPlayer` class uses the `loadMovie()` method to load a video file into memory:

```
bool loadMovie(string name);
```

Remember that, by default, oF expects you're loading files from the *data* folder of your program, so the value of `name` you pass into the `loadMovie()` method should be relative location of your movie file.

To control the speed of the movie, you use the `setSpeed()` method; 1.0 is normal speed, which means 0.5 is half speed, and 2.0 is double speed:

```
void setSpeed(float speed);
```

Next, try using the `setPosition()` method of the `ofVideoPlayer` class:

```
void setPosition(float pct)
```

This sets the position of the video as a percentage of its total length, so 0.5 is halfway through the movie file, and 1.0 is the end of the movie file.

This concludes your brief introduction to working with video in oF, though you'll find plenty more information in Chapter 10.

Importing Libraries

One of the wonderful things about the community that has grown up around oF is that new libraries are being developed all the time to fit not only the needs of the people developing them, but also the needs of the community of people who are using oF. While oF includes libraries to do many of the simpler things that you might want to do, you might be interested in working with more complex kinds of functionality. For this, you can look at the add-ons to oF that oF users have created.

These libraries are all stored in the *addons* folder of the file you downloaded. Each of those folders contains an add-on that has been created with a specific purpose. An add-on isn't included in your program by default, so to include one, you simply tell the compiler you want to include that add-on, and away you go. The ofxOpenCV add-on is a great example. With the ofxOpenCV library, you can use the OpenCV computer vision library created by Intel in your oF program. You can do motion tracking, face detection, and much more with this library, but you have to include it first. To do that, look at the *ofxOpenCv* directory in the *addons* folder; you'll see a file called *ofxOpenCv.h*, which contains all the import statements needed for the add-on to run correctly. Opening the *install.xml* file, you see the following:

```
#ifndef OFX_CV_H
#define OFX_CV_H


//--------------------------
// constants
#include "ofxCvConstants.h"

//--------------------------
// images
#include "ofxCvImage.h"
#include "ofxCvGrayscaleImage.h"
```

```
#include "ofxCvColorImage.h"
#include "ofxCvFloatImage.h"
#include "ofxCvShortImage.h"

//--------------------------
// contours and blobs
#include "ofxCvContourFinder.h"


#endif
```

When you include the *ofxOpenCv.h* file in your application, you're including everything that *ofxOpenCv.h* includes. This means that the whole library is being imported, allowing you to get it all with only one statement: `#include "ofxOpenCv.h"`. Now your application is ready to go:

```
// standard defining of app
#ifndef _OPENCV_APP
#define _OPENCV_APP
#include "ofxOpenCv.h"

// everything else as usual
class opencvApp : public ofBaseApp{
// rest of our app
};

#endif
```

And that's it.

The following are some of the add-ons you might want to use.

ofxOpenCv

Computer vision is a huge and fascinating topic that can open all kinds of new possibilities, from face detection to analyzing movement to blue screening images. We'll discuss the ofxOpenCv package in great detail in Chapter 14.

You might use ofxOpenCv for tracking motion, detecting faces in a video screen, recognizing user gestures, or compositing images in real time.

ofxVectorGraphics

Generally speaking, you'll encounter two kinds of graphics in computer programming: bitmap and vector. Vector graphics are defined by paths, lines, and fills that create shapes and are a lot more lightweight than a bitmaps because an image can simply be some mathematical information that the graphics card will use to create an image. The size of a vector image doesn't matter as much, because it's only the values that describe the shape. Bitmaps are defined by data about each pixel in the image, which means that as the image gets bigger, the number of pixels gets bigger, and the amount of data that

your program needs to work with gets much larger. Vectors are perfect for quickly drawing graphics that use basics shapes.

You might use ofxVectorGraphics for creating large animations, creating an EPS file, creating precise curves and lines, and graphing data.

ofxVectorMath

In graphics programming, you'll encounter two types of vectors: one type is a vector graphic, and the other is the mathematical concept of a vector, which you might remember from high school math. Vectors are essentially an array of numbers that represent the position and heading (or orientation) of an object; they are an important part of creating 2D and 3D graphics. We'll talk more about them when we discuss OpenGL in Chapter 13. When working with vectors, as anyone with a mathematical background or some experience with graphics programming will tell you, you'll need to work with some rather complex math on the odd occasion. Luckily, a lot of the work has been done for you and is stored in the ofxVectorMath add-on, which provides a few of the most important operations for you.

You might use ofxVectorMath for helping you out when working with vector graphics or OpenGL and you have some heavy-duty calculations to do.

ofxNetwork

This add-on helps you send and receive data over a network. With it, an oF program can act as a client and read data from another computer, a server, or another device altogether. A program can also act as a server and send information to a series of client programs.

You might use ofxNetwork for sending data to a website, creating a program that runs in multiple geographic locations at the same time and communicates, or storing or saving data from a program.

ofxOsc

Open Sound Control (OSC) is a protocol for communication between computers, sound synthesizers, and other multimedia devices. OSC is used to create robotics, help programs communicate with one another, and help devices communicate with a computer. OSC is really just a message format that ensures that devices creating OSC messages and devices receiving OSC messages are all using the same specification.

You might use ofxOsc for communicating with a keyboard, drum pad, or other MIDI device; having an oF program communicate with non-Arduino hardware; or communicating with a Processing or Flash program.

Compiling an oF Program

Depending on the IDE you use, compiling your program is a little bit different.

Compiling in Xcode

If you're using a Mac with OS X, you'll be using Xcode. To build your program in Xcode, make sure the Active Build Configuration drop-down menu is set to Release, as shown in Figure 6-8.

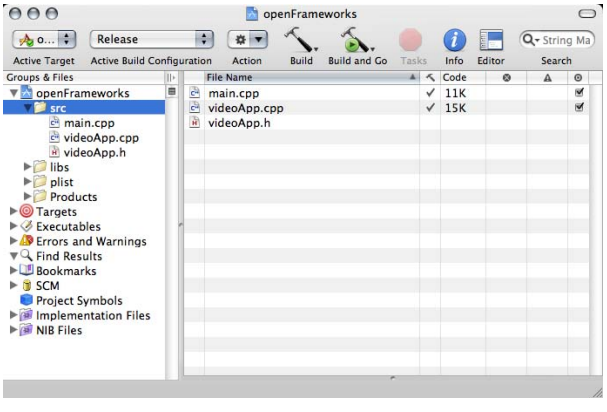


Figure 6-8. The build configuration set to Release

Then, click the Build and Go button, and select the Build and Run option, as shown in Figure 6-9.

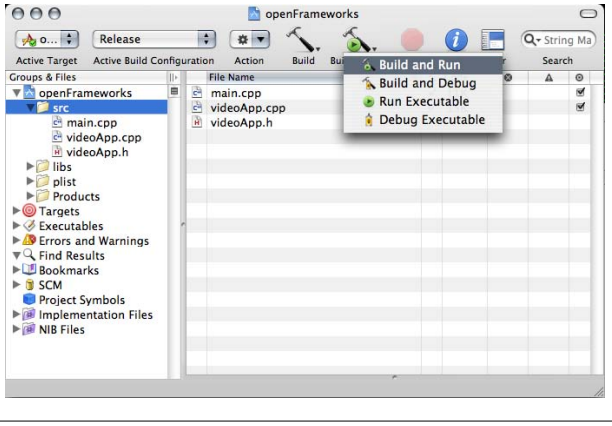


Figure 6-9. Building and running your program

Compiling in Code::Blocks

If you're on Linux or Windows, you'll probably be using Code::Blocks (unless you're using the makefile approach). To build and run your project in Code::Blocks, click F9, or click the Build and Run button, as shown in Figure 6-10.

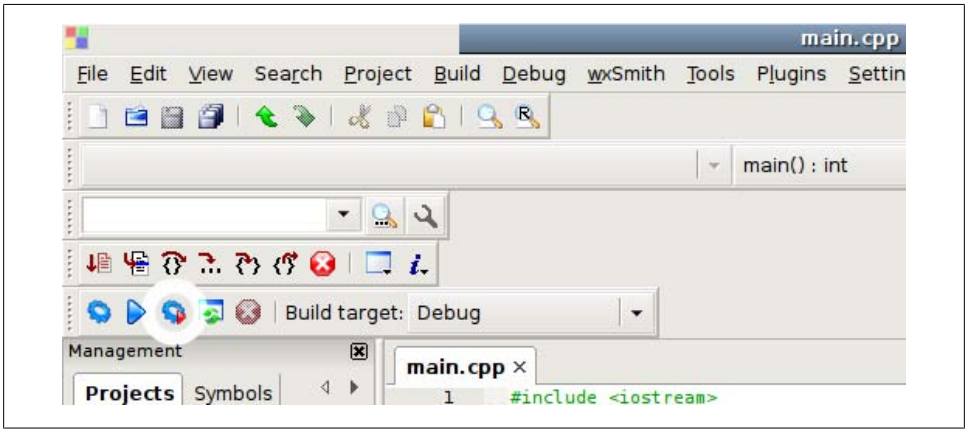


Figure 6-10. Building your project

You can also change the configuration of your project, whether you're preparing it for build or for release, by using the "Build target" drop-down menu shown in Figure 6-10. To just build your project and check it for errors, click Ctrl+F9. This won't run your program, but it will tell you whether you have any errors, which is often quite useful.

Debugging an oF Application

Invariably, things will go wrong with the code you write. Luckily, you can use several tools to figure out what went wrong and how to fix it. The following sections highlight some strategies to use when writing code to help make debugging easier and faster.

Using the printf Statement

The `printf` statement is one of the greatest tools available to you when working with code. Anywhere you have code that might be executing in an unexpected way or you have a variable that isn't behaving as expected, you can use the `printf` statement and pass it the variable name to determine the value of the variable at a given point in time. The `printf` method is a bit of a strange one because it can take an essentially unlimited number of parameters. It takes a string that includes holders for the values you want to print, followed by those values. For example, if you wanted to print an integer, you would use the `%i` placeholder, like so:

```
int widthOfSomething = 5;
printf(" this is the width of something: %i", widthOfSomething);
```

This prints the following:

```
this is the width of something: 5
```

See how the `%i` is replaced by the value of the integer that is passed at the end? Say, for instance, that you have an `ofVideoPlayer` object with a variable height and width that you want to check at a vital point in your program. You would use the following:

```
printf(" video is %i by %i ", player.width, player.height);
```

which would print this:

```
video is 240 by 320
```

So, the two `%i` symbols are replaced by the two integers that are passed to the `printf` method. This does require that you know what kinds of things you're trying to print, though. If you have a float, for example, `1.8`, and you print it as an integer using `%i`, you won't see "1.8" printed; you'll see "–1073741824," which is just a bit different. To print a float, you'll want to use `%f`. Printing a string is a little bit tricky. I'll show you the snippet, and if you're curious, you can look it up in greater detail (though it's a bit tricky, it isn't all that important right now):

```
string s = "I'm a string!";
printf("The string says: %s ", s.c_str());
```

Notice the call to the `c_str()`method of the `string` class. It returns a version of the `string` that `printf()` can use. Without it, you won't see anything printed to the Console.

Table 6-1 lists the flags you can use for `printf`.

Table 6-1. printf flags

Specifier	Output	Example
c	Character	a
d or i	Signed decimal integer	-392
f	Decimal floating point	392.65
g	Use the shorter of %e or %f	392.65
s	String of characters	sample
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
p	Pointer address	0xbffffa18

Using the GNU Debugger

Beyond `printf`, both Xcode and Code::Blocks provide a debugger that interfaces with the GNU Debugger, which is a wonderful tool that lets you pause your program in the middle of the program, examine values, and move to the next line of code. This book

doesn't have enough space to permit going deeply into debugging, but you'll learn how to get a debugging session started in Xcode and Code::Blocks; the rest of the discovery is left to you.

Using the Debugger in Xcode

First, set a breakpoint anywhere in your code by double-clicking a line number to the left of your code in your code view (Figure 6-11).

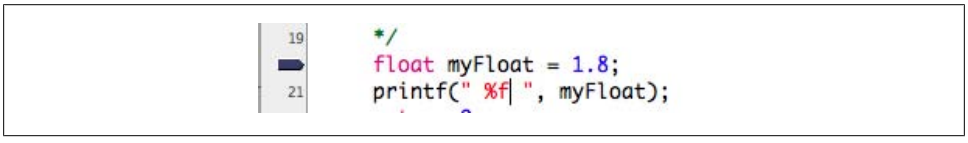


Figure 6-11. Setting a breakpoint in Xcode

Then, make sure your project is set to Debug mode, as shown in Figure 6-12.

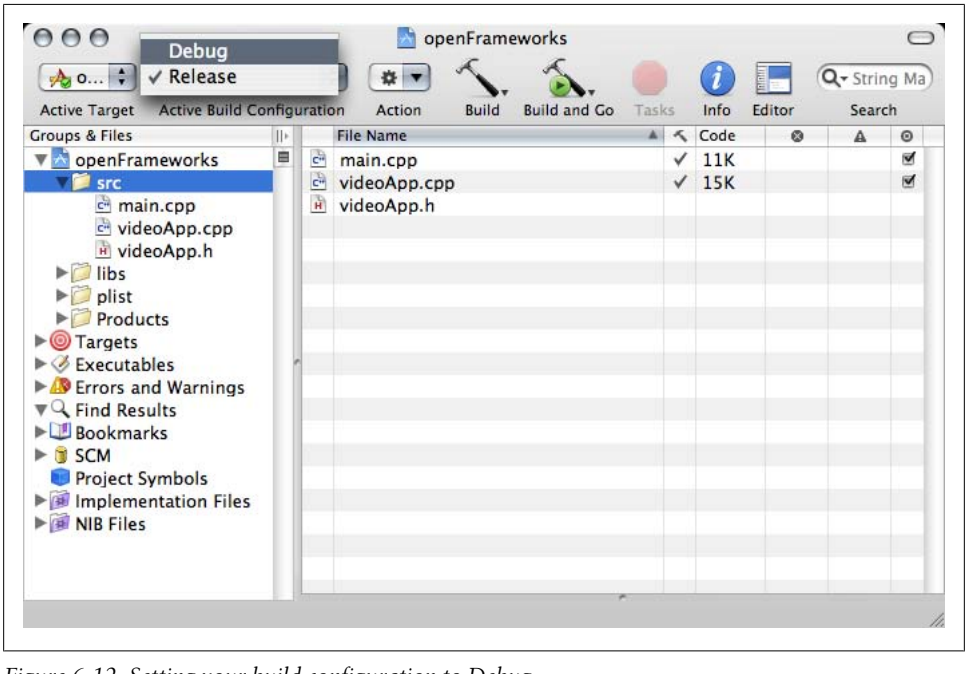


Figure 6-12. Setting your build configuration to Debug

Initiate a debugging session by clicking the Build and Go button and selecting the Build and Debug option, as shown in Figure 6-13.

Once you've started a debugging session, the debugger will run your code until it encounters the line where you've set a breakpoint, as shown in Figure 6-14.

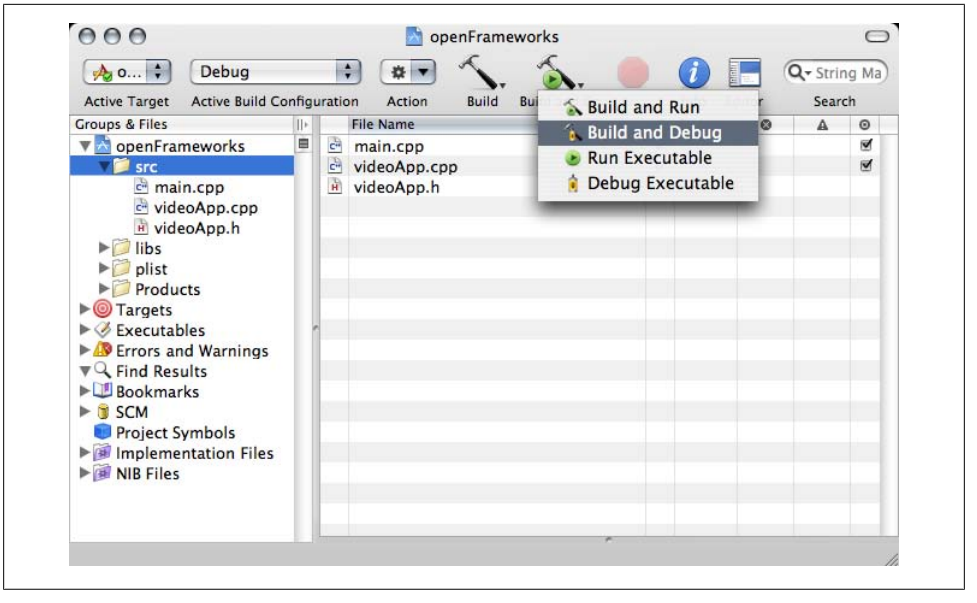


Figure 6-13. Building and debugging your program

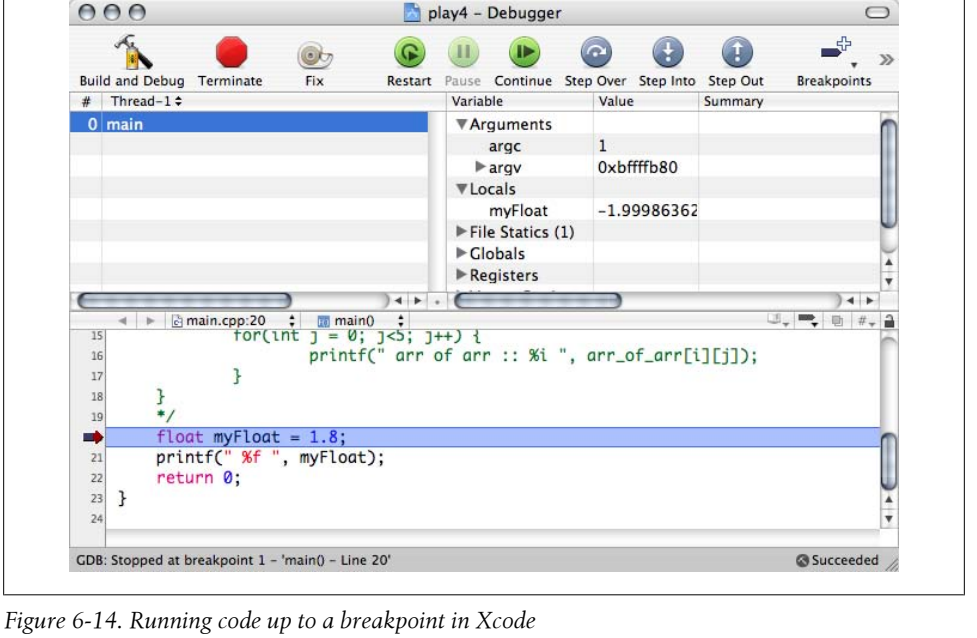


Figure 6-14. Running code up to a breakpoint in Xcode

By clicking the Step Over button that you see above the variable, you can *step through* the code, which means you can advance the debugger line by line to see what

is going on in your code. If you encounter a variable that contains a value you didn't expect, set a breakpoint in front of where that variable is altered, and step through the code. This helps you understand the flow of your program and how values are being passed around.

You may find that certain methods aren't doing what you expect them to do. If you know that the logic inside the method is sound, you can set breakpoints just inside the method to see what's getting passed in when the method gets called. This can help you understand how the program is calling that method and what's going on outside it. There are many other tricks to using debugging effectively; what's most important to understand is that the debugger lets you check the value of a variable at a given point in time.

Using the Debugger in Code::Blocks

Setting breakpoints in Code::Blocks is similar to setting a breakpoint in Xcode; simply click on a line number. A bright red circle appears. This tells you where the debugger will stop when it is executing code. Next, you simply start the debugging session. From the Debug menu, select Debug → Start (Figure 6-15) or click F8.

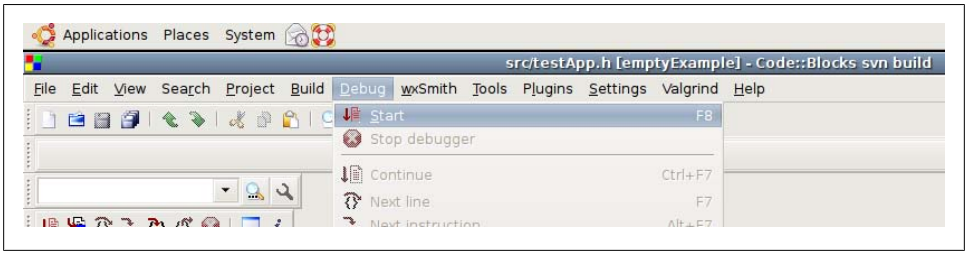


Figure 6-15. Starting the debugger in Code::Blocks

Once the debugger is running, you can check the values of your variables by looking at the variables in the Watches window (Figure 6-16).

By clicking F7, you can step through the code line by line to see what is going on in your code. Of course, you can do many other things with the Code::Blocks debugger, but we'll leave that to you to discover if you're interested.

Review

openFrameworks is a C++ framework created to simplify advanced graphics and sound programming for artists and designers.

openFrameworks does not supply an IDE and requires that you use one of the available IDEs for your operating system. Xcode is recommended for Mac OS X, and Code::Blocks is recommended for Linux and Windows.

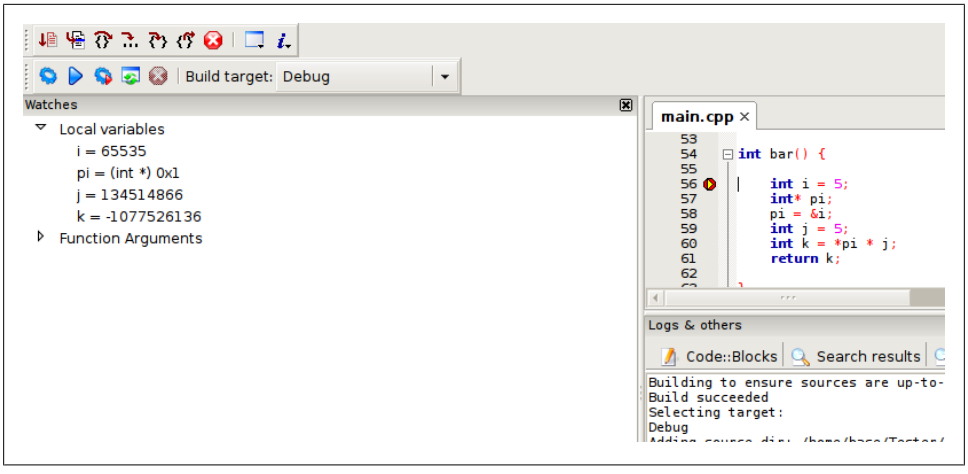


Figure 6-16. Viewing the values of variables at runtime in the Code::Blocks debugger

All oF programs use a base class called `ofBaseApp`. To create an oF program, you create your own application class that extends this `ofBaseApp`.

A very basic oF program consists of three files: the *.cpp* file for the `ofBaseApp` class, the *.h* file for the `ofBaseApp` class, and the *main.cpp* file.

The `ofBaseApp` class has three primary routine methods: `setup()`, which is called when the program starts up; `update()`, which is called just before `draw()` and should be where you put any code that changes data; and `draw()`, which is where the program does its actual drawing.

The `ofBaseApp` class defines callback methods for letting your program know when the user has moved a mouse or clicked a key. You can find all these methods in the *ofBaseApp.h* file.

A few simple 2D drawing methods are provided in oF that you can use to create images and animations; they're all stored in the *ofGraphics* files. These methods help you easily draw circles, lines, rectangles, and curves.

To load images into an oF program, declare an object of the `ofImage` class in your *.h* file. Then, in your `setup()` method, load an image source into your `ofImage` using the `load()` method for `ofImage`. In your program's `draw()` method, call the `draw()` method of the image to display the pixels from the images on the screen.

To play videos, declare an object of the `ofVideoPlayer` class in your *.h* file. Then, in your startup method, load a video file into your `ofVideoPlayer` using the `load()` method `ofVideoPlayer`. In your program's `draw()` method, call the `draw()` method of the image to display the pixels from each of the videos on the screen.

To import add-on libraries into your program, add the definition for that add-on into the *.h* file of your `ofBaseApp` class.

The easiest way to debug your program is to use the `printf` statement and check the values of your program as it executes your code.

If you're using Xcode or Code::Blocks, you can use the built-in debugger to set breakpoints by clicking the line numbers of the code in which you're interested. When you start a debugging session, the debugger will stop once it reaches that line of code; this lets you see the values of your variable during runtime. This is a fairly advanced operation, so consult the manual for your IDE to learn more.

PART II

Themes

Now that you've been introduced to the different tools that you can use to create interactive applications using hardware and software, this book will move on to some of the major themes in interactivity. Understanding how you can use sound, physical controls, video, and graphics in interaction is more than a matter of just looking at code; it also requires that you understand some of the theory and fundamentals behind each of these methods. This "Themes" part looks at some of the most commonly explored means of creating interaction between users and systems and shows you how to create examples of those themes in the appropriate technology. You can create some kinds of applications only in Arduino or openFrameworks, while in other cases, the technology that you use is entirely up to you.

Interaction design is more than just writing code; it's also understanding how the approach that you're taking to interaction creates meaning for the user and for your system. Good interaction requires good input and feedback, and will vary depending on the medium that you're using to communicate. For example, audio presents much different challenges than video does, and integrating different mediums for input and output requires that you match up the strengths, weaknesses, and differences of the mediums creatively.

Throughout this part, you'll be exploring different techniques for writing code, using sensors and controls, and integrating preexisting libraries into openFrameworks, Processing, and Arduino applications. Chapter 7, *Sound and Audio*, introduces working with audio in applications and shows you some of the techniques and libraries for processing signals, playing files, and generating sound. Chapter 8, *Physical Input*, introduces you to working with physical input via controls and sensors for Arduino. Chapter 9, *Programming Graphics*, shows you some more advanced techniques for creating vector graphics and basic physics on a screen with Processing and openFrameworks. Chapter 10, *Bitmaps and Pixels*, focuses on using bitmaps and videos and how to work with and manipulate individual pixels in openFrameworks and Processing. Chapter 11, *Physical Feedback*, introduces you to components and techniques that you can leverage to create physical feedback in your application with the Arduino

controller. Finally, Chapter 12, *Protocols and Communication*, shows you how to work with networks and data protocols with all three of the tools.

By the end of this part, you will have a good introduction to several of the key methods of input and feedback that you can use in your own interactive designs.

CHAPTER 7
Sound and Audio

Our primary experience of computing and of computers is through the screen. The feedback to the user and the input of the user are both most often communicated visually. This isn't, however, how our experience of the world works, or how our cognitive and perceptual facilities function. One of our most finely turned and emotionally evocative senses, our hearing, is often relegated to a lesser role or is ignored altogether in interactive design. In some scenarios, sound isn't a viable method of input or feedback. We don't want to have to talk to a computer or have it making noise in a quiet office. In other situations, neglecting the possibility of aural interaction with a user is a great loss to both an art piece or a product. Video game designers have put great effort and attention into their sound engines and the quality of the sounds in their games. Architects and interior designers, stage designers, sociologists, and of course musicians and sound artists all understand how our experience of the world is shaped by sound. Sound isn't always vital to a particular task, but it's an essential component of a rich and complete experience. You don't need to hear anything to enter your credit card number into a form, but a well-crafted, logical, and sensible use of sound in an application helps you perceive the world.

When sound is married effectively with a visual element, both elements are strengthened. Imagine a ball hitting a wall with and then again without the sound of it. Being able to hear the sound tells you a great deal about the room, including whether it echoes, whether it's muffled, and how large it is. It tells you a great deal about the wall, about the ball, and about the strength with which the ball was thrown. These shape the mental image and shape the understanding of the world around that image. When you create applications, it's of utmost importance that you help the user understand the application. From this little thought experiment, you can begin to see how sound can shape, heighten, and enrich a user's understanding of the world.

Sound physically affects us beyond its movement of the air that surrounds us; a sound wave causes the area of the human brain that is responsible for processing audio signals to generate electrical pulses at the same frequency. The sound wave for an A note, 440 Hz, causes the brain to produce electrical pulses at 440 Hz. So, your physical reaction to sound reaches beyond the pressure on your ears and, in some extreme cases, your

stomach and skin and reaches into your brain as well. Our experience of hearing is a physical phenomenon on many different levels.

The physical nature of sound also makes it a very powerful tool for input. Asking a participant to interact with a device through sound is, while not completely novel, engaging in that it asks them to draw more attention to themselves than asking them to input something through a keyboard. An application that asks you to talk to it demands your attention and your presence in a way that keyboard input does not. This kind of vocal input also tells your application a lot more about the user. An incredible amount of information is contained within the voice that can be used by application: the volume, pitch, cadence, rhythm, pauses, and starts.

In this chapter, you'll learn some of the techniques for creating and processing audio in Processing and openFrameworks as well as learn more about how computers process audio and how you can analyze audio signals.

Sound As Feedback

Sound can influence a person's perception of objects and information. A higher sound is in a different location than a lower one. A major chord is a different type of object than a minor chord. A sudden change in pitch signals a change in the state of your system. Typically, humans can hear any sounds between 20 Hz and 20,000 Hz. This is an immense data set; however, we'll offer a few caveats. First, the human ear is far more sensitive to certain ranges of sound than others. These ranges correlate generally to normal human vocal ranges. Second, the ear can't detect all changes in frequency. The amount of change of a frequency depends greatly on the volume of the sound and the range that it's in. Despite these limits, the amount of information that can be transmitted in audio is immense.

Here are a few general examples about how you can use sound to give a user direct feedback on an action or on information:

As a recognition or audio icon
: Think of an action, one without a visible indication of what has happened. If you can't provide any visual signal to a user, then a sound can signal the user. This can be a simple clicking sound that accompanies pressing a button or a small, subtle sound that acknowledges that a transaction at an ATM has completed. The sound does the same thing as a pop-up message. Given the myriad of contexts in which users interact with applications today—in a busy office, on a bus, in their cars—it's difficult to create omnicontextually appropriate sounds, so many designers avoid the use of sound altogether. This doesn't mean that using sounds as feedback isn't useful, appropriate, or even enriching. It does, however, shape a lot of the thinking about the use of sound. Some research suggests that the first time a user performs a task, audio feedback is extremely helpful, but that with repeated

performances of that same task, sonic feedback becomes a potential nuisance. It's important to consider this when designing for a task-driven activity.

As an associative or textural message

In more passive activities, sound becomes something quite different. Think of almost any product, and you'll probably hear a jingle dancing through your head. A sound can be, just like an image, an iconic signifier or a branding tool. Movies, television shows, commercials, radio broadcasts, and even operating systems all have musical signatures. This is because sound is a very ephemeral and, frequently, a deeply emotional experience. It's partly that ephemeral quality that makes sound such a powerful experience.

As an aid to simulation

In less task-driven interactions, in play, for example, or in exploratory applications, sound becomes a vital part of the feedback to the user because we experience so much of the world in a fully audiovisual environment. The experience of a world or environment without sound is at least a little limited. The higher the fidelity of the sound and the more closely the sound is appropriately related to the event, the more fully realized the environment will seem. Think of a video game that involves computer-generated enemies shooting at you. When enemies are right next to you, you would expect that the sound of their gunfire would be louder than when they are far away from you. The sound "seems right" if it matches our experiential expectation. The audio can give a user information if it's contextually correct. If you see someone walking down a hallway and you hear echoes, this is contextually correct and gives a sense of the hallway, their footsteps, and the scene as a whole. The sound of footsteps echoing in a field doesn't really give a user information because it isn't contextually accurate.

Humans are very skilled at positioning sound. This is a product of the development of our ability, via hearing, to detect danger from potential predators or enemies. This means that auditory signals can transmit spatial information. Playing any video game will bear this out. Using two or four speakers, you can experience a rich world of physical data, positioning Doppler effects, echoes, and other facts that give you so much information. You'll learn more about creating 3D sound in openFrameworks later in this chapter.

As a product of play or nondirected action

Creating tools for making sound is one of the oldest endeavors. Creating instruments is a complex and vast topic. Nearly all of us have at least some experience with an instrument for creating sound, whether it's a guitar, a drum, the tapping of our foot, our own voice, a computer program, or a programming language. One of our favorite pieces for working with sound is the Sonic Wire Sculptor by Amit Pitaru. It combines the ability to visualize sound, to make something out of the most ephemeral of art forms, with a tool that gives the user the ability to create both drawings and music. The Reactable project (which Sergi Jordà et al.,

developed at Pompeu Fabra University of Barcelona) is another excellent example of a music and sound-making tool.

As a way of telling us information about the objects making the sounds

This ties in to a certain degree to the third point, as an aid to simulation. The source of a sound is oftentimes indicated in the sound itself. Hitting a glass with a fork sounds like a glass, a person whistling sounds like air rushing past something, a rock landing in water tells you a great deal about the size of the rock, and when you hear someone's voice, you gather a great deal of information about them. You can hear what makes the sound. You also hear where the sound is coming from, the location, the echo, the shape of the room or space, and how much background noise distracts from the primary noise.

As an emotional trigger

The beeps, clicks, and verification sounds that you're accustomed to hearing in an application are small, subtle tones. Those small audible feedback signals are appropriate for use in a business application, an operating system, or a website. Many books and discussions of using sound in interaction are limited to these sorts of auditory signals because, on a pragmatic level, the majority of work done in interaction design is for websites and business applications. Part of the premise of this book is to consider the possibilities for communication outside these sorts of contexts.

The powerful subconscious emotional significance of sound as a signifier is almost always on display in limited and subtle ways in traditional application interaction design and surrounds us in our day-to-day lives, in the soaring strings in a poignant moment of a film, in the cheer when a goal is scored in football, or in the opening stanza of a national anthem. Because the context of interaction with programs changes rapidly and dramatically, the nature of the sound and the information that it conveys changes as well. At the dawn of the video gaming industry, the 8-bit sound employed by game designers was aimed at producing simple feedback signals to accompany visuals. Now sound design is employed in games to add emotional elements, drive the narrative, heighten tension or rewards, or alert players to events offscreen.

The best way to determine how to create the signals and understanding in your users or audience is to experiment and observe closely how the sounds you use are received. Although engineering is often the art of avoiding failure, interaction and industrial design are the art of failing and understanding the failures correctly. Sound in particular is difficult to understand in the context of an application without seeing how others react to it. A good sound design is a deeply enriching aspect to any experience that is immersive, emotionally affective, and truly communicative, and takes a great amount of practice, testing, and tweaking to get right. That said, getting it right is well worth the effort.

Sound and Interaction

Interactive art that works with sound is bound to the interaction that the user has with the sound. That depth of interaction is determined by the control that the user or participant has over the sound.

The key element to understand is how the audience will control the sound that they will hear or that they create. If the user doesn't have any control over the sound they hear, that person is a spectator. The control that users have over the sound can vary. The acoustics of an automobile, the road noise, the sound of the engine, and the acoustic qualities of the interior are very important to the driver. All of these become elements that the driver controls and alters while driving. Spend any time with someone who loves cars, and they'll invariably tell you about the sound of an engine when it's revved up. The interaction there is simple: press on the pedal. What it communicates, though, is quite rich and complex. If it were simply a sound that the driver did not control, then it would not have the same attractiveness to the driver. It's part of the experience of driving and an element of the experience that industrial engineers pay close attention to. The industrial designer Raymond Loewy wrote, "A fridge has to be beautiful. This means it also has to sound good."

Interaction is generally processing-oriented. This means that the elements of interaction never stand on their own: they are always interrelated and linked to a temporal development. Sounds themselves are inherently bound to processes in time. As interactive systems become more and more common, where the processes of the system are more and more difficult for users to understand, providing continuous feedback becomes more important. While visual information relies on movement and color, a sound can create a more continuous signal. The idea isn't to have an application buzz or beep the entire time the user is interacting with it, because some of the aspects of sound make working with it less desirable, such as public space, low sound quality, and bandwidth problems.

Let's examine some of the most common ways of using sound as the driver of an interaction:

The user creates a sound that she or someone else can hear

This is essentially a recap of using sound as feedback for play or nondirected action. In these sorts of scenarios, the user's goal is to manipulate, control, or make a sound. This doesn't have to be as directed and purposeful as playing a piano or as finely controlled as a recording studio mixing board. Helping people make novel sounds can be a sort of play, a way of communicating whether someone can control the generation of sound, or a way of sending a signal to someone else. Some sound-generating interactive instruments are the theremin, RjDj, the iPhone DJing application, and of course the classic DJ turntable setup. What these all have in common isn't just that they make sound, but that the sound is produced interactively. In the case of the theremin and turntables, the sound is produced by a user's gestures. Think of using the Wii to drive a theremin, using the accelerometer to change

the tempo of a drum, or having a painting application create different sounds based on the color and stroke that users create on a canvas.

The user creates a sound input using a tool

Sound is everywhere and has such a vast range of tempos, timbres, pitches, and volumes that it's nearly impossible to prune it down into neat input data, unless you know exactly what kind of sound you're expecting. You may remember rotary dial phones—the old phones that sent a series of clicks for each number. The call routing system used those clicks to determine what number you were dialing. That system was later replaced by push-button phones that used a different tone to represent each number. These are both prosaic but excellent examples of sound-creating tools that create input. Any well-known music instrument can create a very precise tone that could be used as a command. A MIDI keyboard or a tool like the Monome control board offers even more possibilities. Any sound that can be reproduced with any degree of fidelity can be mapped to a command. The sort of fidelity required for complex commands isn't well suited to the human voice, but it can be very well suited to any instrument.

The user creates a sound that is the input

Two different types of applications use sound that users generate on their own, that is, without an extra tool: those that use sound data and those that use speech data. While it's true that spoken words are sound data, getting a computer to recognize speech is different from having it process sound. If you're interested only in processing the sound that users make, you can use a few different approaches. The first, and perhaps more obvious, is to use a microphone. From a microphone, you can analyze the volume of the user's speech, varying some input based on how loud they are talking. You can also determine whether they have just begun talking by taking a "snapshot" of the sound in a given period of time (a second, for instance) and comparing it with the sound in the next snapshot. Big changes in sound can indicate an activation that can act like a switch. Volume is a spectrum that means that it doesn't report data that's on or off but rather data from 0 to 100. Pitch, the frequency of the user's voice or the sound he is creating, works much the same way. It can be turned into any kind of spectrum values. Another thing that you can determine from a voice is its tempo. You can do this using beat detection, which a library like Minim for Processing makes very easy. The speed of sound is another spectrum of data that you can use. Adding another microphone lets you create positional data by simply comparing which mic is louder. Finally, as mentioned earlier, sound is vibration, and many piezo elements or small microphones can be used with Arduino to create up to 10 sound and vibrational inputs. These could be used to determine the position of a user within a room, enable many different elements to communicate, or create an interface that uses highly sensitive sounds to catch taps, touches, or rattling.

The user talks or says a word to create the input

Speech recognition is exciting, filled with interactive possibilities, and is, at the moment, very difficult to do. *Speech recognition* doesn't mean recognizing who is

talking (that's called *voice recognition*), but rather recognizing what someone is saying. As you might imagine, the difficulty of the task grows as the vocabulary grows. Recognizing a "yes" or a "no" is quite easy. Understanding a full-speed English sentence with a possible vocabulary of 25,000 words is very, very difficult. Beyond the computational difficulties, there aren't any easy ways to get started working with speech recognition. A few open source speech recognition engines exist that you can play with, but they're a little difficult to get started with. Ideally, this will change soon so that artists and designers can get started making voice-activated and speech-driven applications.

You should consider creating an application that enables sound interactions for several reasons. On a practical level, sound creates opportunities for you to let users with physical or situational mobility issues interact with your application. Sound interaction also enables communication where no keyboard or physical input can be used. On a completely different level, interacting through sound can be fun. Making noises is like play, and making noises to drive an application that does something playful in response is a fun interaction all around: the input is fun, and the feedback is fun.

How Sound Works on a Computer

Before we can discuss a representation of sound, we need to discuss what sound actually is. *Sound* is a wave of air pressure that we can detect with our ears. That wave has both a minimum and maximum pressure that define the sound frequency. Higher sounds have a higher frequency, and lower sounds have a lower frequency, which means that the maximum and minimum values occur much more closely together. The volume of the sound is, roughly speaking, the amount of air that the sound displaces. So, a pure single tone is single wave, which might look something like one of the two waves in Figure 7-1.

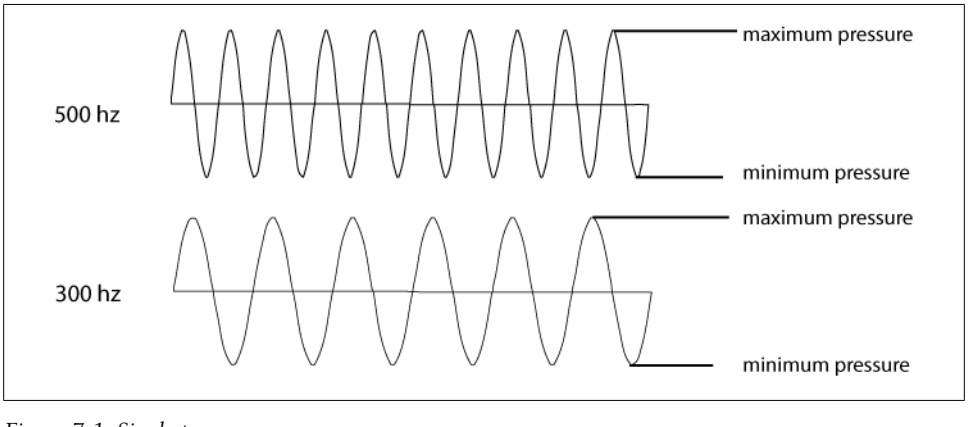


Figure 7-1. Single tones

It's interesting to note that when two or three tones are combined, they create a complex tone that looks like the ones in Figure 7-2.

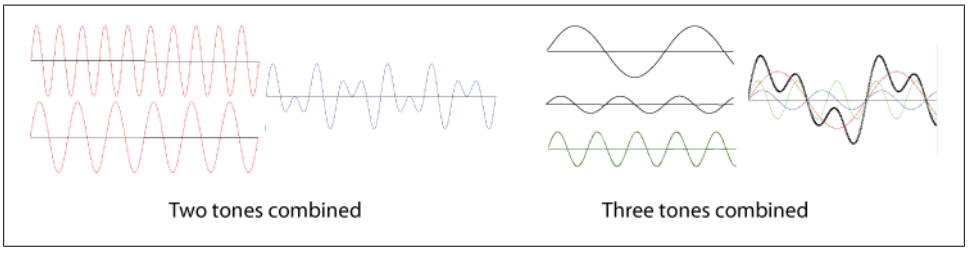


Figure 7-2. Two and three tones combined into a single tone

For you to work with sound on a computer, you or someone else needs to have captured the sound using a microphone; alternatively, you might have synthesized it using a sound synthesizer. In the case of capturing sound, you're converting the wave in the air to an electromagnetic wave. That electromagnetic signal is then sent to a piece of hardware called an *analog to digital converter*, which converts the electromagnetic wave into a series of integers or floating-point numbers. This is where one of the trickiest parts of working with sound comes into play.

A wave is a smooth curve, whereas a series of numbers trying to represent that curve needs to reduce some of the complexity of that curve. Why is that? Take a look at Figure 7-3.

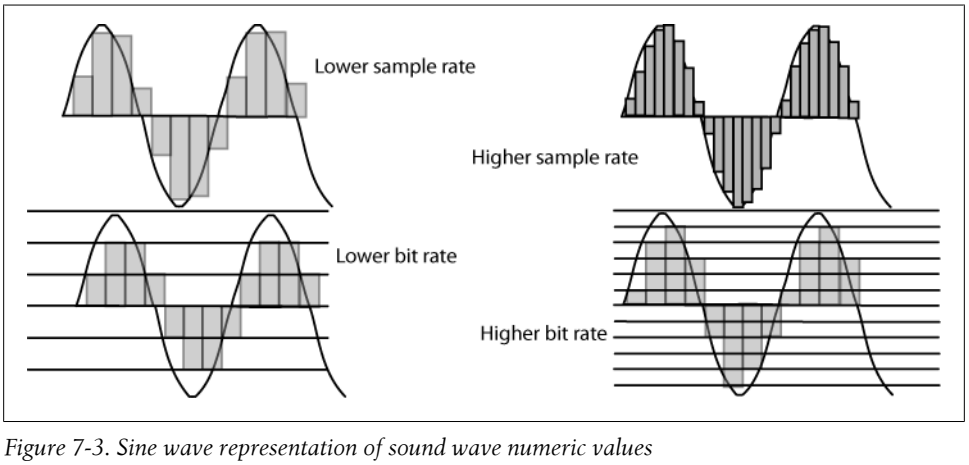


Figure 7-3. Sine wave representation of sound wave numeric values

Notice in the upper two diagrams of Figure 7-3 how a higher sampling rate means that the values of the blocks, which represent the values in the array of floating-point numbers for passing audio data around an application, more closely approximate the curves. The sample rate is actually the number of samples taken per second. In CD audio, the

waveform is sampled at 44.1 KHz (kilohertz), or 44,100 samples per second. That's the most common modern sample rate for consumer audio, but it's not the only possibility. Older audio files used to go as low as 11 KHz or 22 KHz, whereas modern video uses an audio sample rate of 48 KHz. Greater sample rates mean larger data sets, and lower sample rates mean smaller data sets. Depending on your needs, one sample rate might be more appropriate than another. A nice compromise is 44.1 KHz.

In the lower two diagrams of Figure 7-3, the one on the left shows how a lower bit rate makes it more difficult to accurately approximate a sound. Specifically, 8-bit audio (256 possible values per sample) isn't a very good representation of a sound: think of the first Nintendo Entertainment System. That has its own charm, but you wouldn't want to record most music with it. CDs use 16-bit audio, allowing more than 65,000 possible values for each sample. Modern digital audio software also starts at 16-bit, but it's increasingly popular to record at 24-bit, with a massive 16 million possible values per sample.

For uncompressed audio, such as a *.wav* or *.aiff* file, sound approximation is done by measuring the wave many, many times a second and converting that measurement into a binary number of a certain size. Each measurement is called a *sample*, and the overall process is referred to as *sampling*. This creates a representation of the wave using *slices* of discrete values; the representation looks like stairs when you lay it over the sine-wave diagram. These factors (the number of measurements per second, or *sample rate*, and the precision of measurements, or *bit depth*) determine the basic quality of an uncompressed audio file. What about MP3 files? They operate in much the same way, but with an extra level of compression to save space. You'll notice that an MP3 file is smaller than the same data saved in WAV format. This saves space but also reduces quality, and means that the actual data stored in an MP3 file is different.

As you work through the examples in this chapter, you'll find arrays of floating-point numbers used again and again. This is because floating-point numbers are the root of audio processing, storing and manipulating data as numbers, and then writing the numbers to the audio card of a computer. Listed below is some of the equipment and hardware interaction used to make, capture, or play sound:

Sound card
: A *sound card* (also known as an *audio card*) is a computer expansion card that facilitates the input and output of audio signals to/from a computer with the help of specialized software. Typical uses of sound cards include providing the audio component for multimedia applications such as music composition, video or audio editing, presentation/education, and entertainment (games). Many computers have sound capabilities built-in, while others require additional expansion cards to provide for audio capability.

Buffering and buffers
: Computers grab only what you need from a file to send to the sound card. This is called *buffering*. You'll notice that much of the sound data that you receive when

working with audio is grabbed in small chunks that are often passed around either as arrays of floating-point numbers or as pointers to arrays of floating-point numbers.

Drivers and devices
: Most libraries enable communication between your software and your computer sound card. The average sound card handles both writing sound out to a speaker and reading and digitizing sound coming from a microphone. For most other devices, like keyboards, instruments, or other devices, you'll need to have your application enable the device either by using a library for Arduino, Processing, or openFrameworks, or by having your application interface with another system. The applications that enable an operating system to communicate with other applications are called drivers and are often used to facilitate communication over USB or Serial ports. This chapter explains some of the basics of how to use the MIDI and OSC protocols, both of which let you communicate with other systems and devices, many of which are quite relevant to processing audio signals.

Audio in Processing

While Processing has a great deal of support for working with graphics, video, and OpenGL built into the core libraries, it doesn't provide nearly as much functionality for working with audio. Luckily, the Processing community has developed several excellent libraries for working with sound. The Minim library, developed by Damien Di Fede, is one of the best known and most complete, though several others are available as well. It's available with the Processing download as one of the core libraries.

Instantiating the Minim Library

The philosophy in creating Minim was, as Damien puts it, to "make integrating audio into your sketches as simple as possible while still providing a reasonable amount of flexibility for more advanced users. There are no callbacks, and you do not ever need to directly manipulate sample arrays; all of the dirty work is handled for you."

The core of the Minim library is a class called `Minim`. Every time you use the Minim library, you need to instantiate a Minim object and call its constructor with the `this` keyword. You can perform four tasks with the `Minim` object: play an audio file that you load into your application, play audio that you create in your program, monitor audio and get data about it, and record audio to disk. Different classes in the Minim library handle these tasks and you can obtain instances of those classes by calling the appropriate methods of `Minim`:

```
Minim minim;
// remember that the 'this' keyword refers to your Processing application
minim = new Minim(this);
```

Now that you have the library initialized, you can begin doing things with it, like loading MP3 files and playing them back:

```
import ddf.minim.*;
AudioPlayer song;
Minim minim;
void setup()
{
    size(800, 800);
    // don't forget to instantiate the minim library
    minim = new Minim(this);
    // this loads song.mp3 from the data folder
    song = minim.loadFile("song.mp3");

}
```

This is a common pattern in the Minim library and in a lot of sound manipulation libraries, like the C++ Sound Object library. A core class is instantiated, and then new objects are created from that core object to do specific tasks like playing files, creating filters, or generating tones. To load an audio file, the `AudioPlayer` class provides mono and stereo playback of WAV, AIFF, AU, SND, and MP3 files. `AudioPlayer` is instantiated by the static Minim `loadFile()` method. Once you've created `AudioPlayer`, you can pause, play, and apply filters, as well as read data from the raw audio file using the `right` and `left` arrays, all from `AudioPlayer`. The `left` and `right` properties of `AudioPlayer` are arrays filled with floating-point numbers that represent the left and right channels in the audio file. In the `draw()` method, those arrays can be used to draw a line of small ellipses. You'll be hearing the sound, and seeing a representation of it as well:

```
void draw()
{
    fill(0x000000, 30);
    rect(0, 0, width, height);
    //background(0);
    stroke(255);
    noFill();
    for(int i = 0; i < song.bufferSize() - 1; i++)
    {
        ellipse(i * 4, 100 + song.left.get(i)*100, 5, 5);
        ellipse(i * 4, 250 + song.right.get(i)*100, 5, 5);
    }
}
```

The `AudioPlayer` class also defines `play()` and `pause()` methods for the playback of the audio information:

```
bool isPlaying = false;
void mousePressed()
{
    if(isPlaying) {
        song.pause();
        isPlaying = false;
    } else {
```

```
            song.play();
            isPlaying = true;
        }
    }
    void stop()
    {
      song.stop();
      minim.stop();
      super.stop();
    }
```

In these few lines of code, you have an instantiated an audio library that can load an audio file, control the playback, and read the data from the audio file as it plays to create graphics.

Generating Sounds with Minim

Minim also defines methods to generate new sounds from equations. Four fundamental kinds of waves can generate sounds: triangle, square, sine, and sawtooth. The names are derived from the appearance of the waves when they are graphed, as in Figure 7-4.

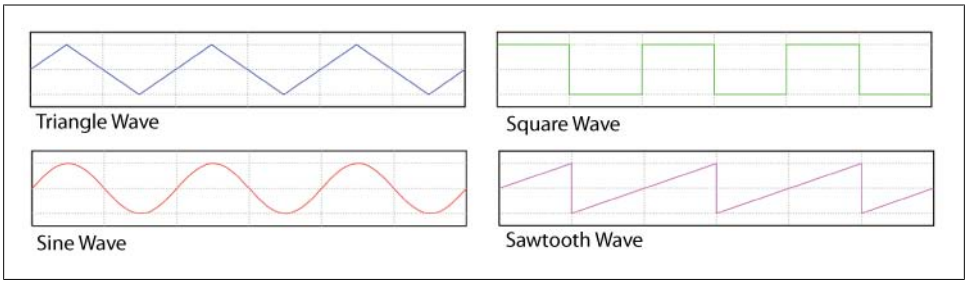


Figure 7-4. Sound wave pattern types

Each of these creates a slight variation of the familiar single tone that can be the base of a more complex tone. Minim simplifies generating tones for you by providing several convenience classes that generate tones and let you manipulate their frequency and amplitude in your application. While generating a sine wave may not seem useful when you need to provide feedback to a user, it's an excellent way to begin building complex layers of sound that increase in pitch or volume depending on input.

Before we go over the code for generating the waves, it's important to understand how the Minim library gets access to the sound card of the computer on which it's running. Generating a sine or sawtooth wave is really a process of feeding floating-point values to the sound card so that it can convert them into analog signals. The `AudioOutput` class is used to store information about the data being sent to the sound card for manipulation while the application is running. Though the `AudioOutput` class has several dozen methods, in the interest of space we'll discuss these two:

`addSignal(AudioSignal signal)`
: Adds a signal to the chain of signals that will be played.

`removeSignal(AudioSignal signal)`
: Removes a signal from the chain of signals that will be played.

Additionally, the `AudioOutput` class defines the following variables:

`left`
: Is an array containing all the samples for the left channel of the sound being sent to the sound card.

`right`
: Is an array containing all the samples for the right channel of the sound being sent to the sound card.

`mix`
: Is an array of data containing the mix of the left and right channels.

Any time you need access to the data being sent to the sound card—say you're mixing several tones together and want the final mix of all the sounds—the `mix` property of the `AudioOutput` class will give you access to that information.

You can't use `AudioOutput` before it has had all of its information set by the Minim framework. To do this, you use the main `Minim` class, which defines a `getLineOut()` method with the following four signatures:

```
getLineOut(int type)
getLineOut(int type, int bufferSize)
getLineOut(int type, int bufferSize, float sampleRate)
getLineOut(int type, int bufferSize, float sampleRate, int bitDepth)
```

You'll notice that all the methods require a type parameter, which can be one of the following:

`MONO`
: Sets up a mono, or single-channel, output.

`STEREO`
: Sets up a stereo, or two-channel, output.

The `getLineOut()` method instantiates and gives data to `AudioOutput`, as shown here:

```
AudioOutput out;
out = minim.getLineOut(Minim.STEREO);
```

Without calling the `getLineOut()` method, none of the audio data generated by the `SineWave` or `SquareWave` classes will be routed correctly to the sound card. Now that you know how to start the sound, take a look at generating a square wave and a sine wave:

```
import ddf.minim.*;
import ddf.minim.signals.*;

AudioOutput out;
SquareWave square;
```

```
SawWave saw;
Minim minim;

void setup()
{
    size(800, 800);
    //don't forget, you always need to start Minim first
    minim = new Minim(this);

    //get system access to the line out
    out = minim.getLineOut(Minim.STEREO, 512);

    // create a SquareWave with a frequency of 440 Hz,
    // an amplitude of 1 with 44100 samples per second
    square = new SquareWave(440, 1, 44100);
    // create a SawWave with a frequency of 600Hz and
    // an amplitude of 1
    saw = new SawWave(600, 1, 44100);

    // now you can attach the square wave and the filter to the output
    out.addSignal(square);
    out.addSignal(saw);
}

void draw() {
    saw.setFreq(mouseX);
    square.setFreq(mouseY);
}

void stop() {
  minim.stop();
  super.stop();
}
```

Running this program should give you an idea of what setting the frequency on a sine or square wave sounds like. In the next chapter, which covers using controls with the Arduino board, you'll learn how to use physical knobs to tune a sine wave. That is just the beginning of what is possible when you combine means of input and means of feedback.

Another thing that you'll probably want to do with the Minim library is playback and manipulate audio files. To play audio files in Minim, you use the AudioPlayer. Like many of the other objects in the Minim library, you create the object by having it returned from a method on the main Minim object. For the AudioPlayer, it looks like this:

```
AudioPlayer player = Minim.loadFile("myfile.mp3");
```

This starts the file streaming from its location. The name of the file can be just the filename, like "*mysong.wav*", in which case Minim will look in all of the places Processing looks (the data folder, the sketch folder, etc.), it can be the full path to the file, or it can be a URL:

`void play()`
: Starts playback from the current position.

`void play(int millis)`
: Starts playback millis from the beginning.

One of the difficulties in working with sound in Java is that certain implementations of the JVM have small quirks. One of them affects how Minim changes the volume of the sound when you play back an MP3 file. On some machines, you'll use the `setGain()` method and on others you'll use the `setVolume()` method. Both take floating point values, usually between –80 and 12 for `setGain()` and 0 to 1.0 for `setVolume()`. The easiest way to check whether some functionality is supported is to call the `hasControl()` method of the `AudioPlayer` class:

```
AudioPlayer player = Minim.loadFile("myfile.mp3");
boolean hasVolume = player.hasControl(Controller.VOLUME);
```

This code snippet shown below loads up two files and then allows the user to fade between them using the movement of the mouse. The code will be broken up into the following:

```
import ddf.minim.*;

AudioPlayer player;
AudioPlayer player2;
Minim minim;

boolean hasVolume;

void setup()
{
  size(512, 200);
  minim = new Minim(this);

  // load a file, give the AudioPlayer buffers that are 1024 samples long
  player = minim.loadFile("two.mp3", 1024);
  player2 = minim.loadFile("one.mp3", 1024);
  // play the file
  player.play();
  player.printControls();
  player2.play();
  hasVolume = player.hasControl(Controller.VOLUME);
}

void draw()
{
  background(0); // erase the background
  stroke(0, 255, 0);
  float gain1 = 0;
  float gain2 = 0;
If the setup on the users computer can alter the volume, then set that using the mouse
position, otherwise, set the gain using the mouse position. The effect will be more
or less the same.
  if(hasVolume) {
```

```
    player.setVolume(mouseX / width);
    gain1 = map(player.getVolume(), 0, 1, 0, 50);
  } else {
    player.setGain(map(mouseX, 0, width, -20, 1));
    gain1 = map(player.getGain(), -20, 1, 0, 50);
  }
```

As mentioned earlier in this chapter, the sound buffer is really just a big array of floating point numbers that represent the sound wave. To draw the sound wave to the screen, you can just loop through all the values in each channel and use the value however you'd like. Here, it's just being used to draw a line:

```
for(int i = 0; i < player.left.size()-1; i++) {
    line(i, 50 + player.left.get(i)*gain1, i+1, 50 + player.left.get(i+1)*gain1);
    line(i, 150 + player.right.get(i)*gain1, i+1, 150 + player.right.get(i+1)*gain1);
  }
  stroke(255, 0, 0);
  if(hasVolume) {
    player2.setVolume(width - mouseX / width);
    gain2 = map(player2.getVolume(), 0.0, 1, 0, 50);
  } else {
    player2.setGain(map(width - mouseX, 0, width, -20, 1));
    gain2 = map(player2.getGain(), -20, 1, 0, 50);
  }
  for(int i = 0; i < player2.left.size()-1; i++) {
    line(i, 50 + player2.left.get(i)*gain2, i+1, 50 + player2.left.get(i+1)*gain2);
    line(i, 150 + player2.right.get(i)*gain2, i+1, 150 + player2.right.get(i+1)*gain2);
  }
}

void stop()
{
  // always close Minim audio classes when you are done with them
  player.close();
  player2.close();
  minim.stop();

  super.stop();
}
```

Filtering Sounds with Minim

Filtering sounds is an important element of manipulating them. Filtering a sound can involve removing a narrow band of its sound frequency, removing the highest or lowest frequency part of a complex sound, changing its pitch, or removing popping sounds from audio to smooth it out, among other things. Being able to create and tune filters in Minim is quite simple. The following example creates a `SquareWave` tone and then applies a `LowPass` filter to it, the frequency of which is controlled by the mouse position:

```
import ddf.minim.*;
import ddf.minim.signals.*;
import ddf.minim.effects.*;
```

```
AudioOutput out;
SquareWave square;
LowPassSP lowpass;
Minim minim;

void setup()
{
    size(800, 800);
    // don't forget to instantiate the minim library
    minim = new Minim(this);
    // get a stereo line out with a sample buffer of 512 samples
    out = minim.getLineOut(Minim.STEREO, 512);
    // create a SquareWave with a frequency of 440 Hz,
    // an amplitude of 1, and the same sample rate as out
    square = new SquareWave(440, 1, 44100);
    // create a LowPassSP filter with a cutoff frequency of 200 Hz
    // that expects audio with the same sample rate as out
    lowpass = new LowPassSP(200, 44100);

    // now we can attach the square wave and the filter to our output
    out.addSignal(square);
    out.addEffect(lowpass);
}

void draw()
{
    try {
    if(out.hasSignal(square)) {
        out.removeEffect(lowpass);
    }
    // set the frequency of the lowpass filter that we're using
    lowpass.setFreq(mouseY);
    out.addEffect(lowpass);
    } catch(Exception e) {
    }
}
```

You'll want to make sure that you include a `stop()` method to close out the Minim library:

```
void stop()
{
    out.close();
    minim.stop();
    super.stop();
}
```

This is very small taste of what the Minim library can do, but it is a starting place to explore how to create sophisticated sound in your Processing applications. For more information on the Minim library, go to the Minim website and look at the thorough, well-written, and extremely helpful documentation by Damien Di Fede.

Interview: Amit Pitaru

Amit Pitaru is an artist, designer, and researcher. He develops instruments and installations for music and animation, and has exhibited and performed at the London Design Museum, Paris Pompidou Center, Sundance Film Festival, and ICC Museum in Tokyo. He is also a designer with a particular interest in assistive technologies and universal design. In addition, he creates toys and software that are inclusively accessible to people with various disabilities. He also teaches at the ITP program of New York University and the Cooper Union in New York City.

You've mentioned the idea of there being an infinite learning curve when learning how to make interactive art or interactive design objects. Can you elaborate on that?

Amit Pitaru: Think about other crafts, for instance, photography. When the camera was introduced and it became accessible to anyone who could afford it, there were perhaps five years where people had to figure out how to use it. During these five years, much of the work that was done, if you judge from history, was more experimental than artistic. The really good stuff came afterward. There's value in the experimental new work, in finding new territory and such, but it's not the stuff that we look at today as the remastery of the medium. So, what do you do when you have a medium like new media, when a certain period of five years of technical mastery is expanded to become infinite? Do people ever snap out of the learning mode, and will artists be able to overcome this learning period? Or will they forever be dealing with technical mastery?

You can look at that a different way and say that the ultimate goal of any artist is to spark this creative feedback loop where they could get into this little tunnel of creativity and come out with something really beautiful at the end. You want to be in the zone, in the flow—whatever you want to call it. It's the same when you're writing or you're drawing, or anything. The question is, how do you start that spark? How do you spark this process with new media? Having that technical proficiency (or hiring someone who knows how to do what you can't do) is a relevant issue because of what we said before: the learning curve is infinitely expanded because the technologies keep changing all the time. Sometimes it's better to learn a single technology and then stop worrying about the technology and realize that it's better to then concentrate on making your artwork than learning new skills and techniques.

I know some people who really enjoy the process of learning everything and really thrive from exploring technical challenges, and I know other people who really want to bypass that stuff and get to the part where they make their work or their object.

Amit: It's very different if you actually enjoy programming or you want to use programming as a method to do something that you know is possible. It's the bottom-up versus top-down approach. You learn because you want to get somewhere, but at some point, you get new ideas because of what you just learned. For instance, say someone wants to do a motion-tracking thing that can output whatever. That's an artistic idea, but by its definition someone needs to build all of this technology. I think everyone should try to learn the technology, but I'm not sure that person should go and do everything themselves. I think the trick for people like that is to respect the craft. They need to understand that anybody who makes it look easy has had just as much practice

as any other craft or sport, so just jumping into it—unless you're some sort of superhacker—will make you understand that this craft requires practice and attention.

How did you get started programming?

Amit: My background is in music, not programming. I started programming on a leap of faith, but I had some musical instruments that I wanted to play and ended up enjoying the process of learning how to program very much. I think this was because of my musical training and the analogies that I made very quickly between the type of abstractions that code has and the abstraction that a musical rotation would have or the understanding of what a musical piece would have when you're composing it. I think you'll find a lot of commonalities there.

What do you actually learn when you want to start programming?

Amit: One of the things that happens with new artists who are trying to get into this is that they still don't have a connection between what they want to see and the amount of work it requires. For example, suppose you want to do sound analysis. If all you want to do is grab the volume of the microphone, then it's something you could do with a beginner's exercise. If you want to grab the pitch of the microphone, you need a few months because it's not as straightforward. If you want to do speech recognition and you don't understand the gaps in difficulties, you won't understand the huge mountains between those three things. If you want to start programming and you're set on doing speech recognition, before you even know how to write a lot of code, you may put yourself in a bad situation, especially once you realize how big of a task this is. I think it's really important to learn a little bit of code first, understand what's possible, and come up with ideas and goals that somehow relate to the type of things you can do.

You teach at NYU and Cooper Union. What do you teach there?

Amit: I'm very interested in assisted technology, so the latest course I taught was making games, video games, and hardware for children with disabilities. We hooked up with a hospital and a school, and we actually developed games and hardware that the students are now using. Those things were delivered to them as products. So, it was an interesting exercise in constraints at the end because, as a person who does art for people to use, it is so important to understand usability. We're tapping now into a different subject, but I find that if we have to list the things that new media artist need to know—for example, if you're working on something interactive—you have to learn usability design. All the courses that I teach heighten students' sensibilities to how people use the things that they make, whether it's art or not. I pay close attention to this in any work that I do.

What do you find that your students have the most difficulty learning, other than the basics of code?

Amit: I try not to mix topics too much. If I'm teaching about disability design for a particular target or particular niche, I don't want necessarily to teach them the ABCs of code at the same time. For example, let's talk about teaching a person how to start programming. I think the most difficult thing is to be able to visualize the system that they are creating. Once they are able to control visual representation of how program

flows and how program is structured, that's when they really have to simulate (if not all of the program, then parts of your program at a time) in their own brain. Basically, you can't just have the machine run the program and then look at the output and say, "OK, I know what to do." Usually, programming involves running the code together with the machine, to some extent, and that is a skill that takes time to learn, and that is a skill that needs to be learned. For example, if I draw 10 horizontal lines and I ask a painter, "What do you see?", they'll respond "I see 10 horizontal lines". What I want them to think about now is a program that creates 10 lines evenly spaced horizontally. Then it could be 10, 20, or 30. I want them to see the process that creates those lines, the computational process, and then understand how that system can be flexed and expanded and how different variables go in there.

What do you find when you're teaching a class more centered on usability or interaction design? What are the first things you have to get people to think about?

Amit: You need to start by understanding how you react to the world. In art school they would tell you, "OK, stand upside down on your hands and describe what you see," just to break the usual way of looking. Here, you have to do the same thing. Ask yourself how you perceive a certain situation if you are interacting with anything, whether it's an ATM machine or artwork at the mall. And break those different things that are happening in your mind down as much as you can, and try to find patterns in the way that you see things. Then, you have to look at how other people perceive the same thing, and once you are open to that, the realizations that you have to test everything that you do daily is important. At the end of the day, most of the works that are being done fall down upon the stupidest and smallest details. Art should be communicated before it's understood. You can't build a piece of art that delivers on your intentions without learning usability.

What do you think it means to be making new media art?

Amit: If you want to be a new media artist today, you are in an excellent place, almost in all aspects. First of all, the system isn't defined yet, so you're allowed to be inside of the accepted genres or on the outside, because there really is no outside; you can do whatever you want. I often see people who try to define what new media is, and I run away from that like the plague. I think it's our luxury right now not to be defined. The only time it's worth trying to define yourself and what exactly you want to be doing in this field is if you want to be a commercial new media artist and the art world has figured out a system for artists to make money, which it hasn't.

How did you think about the making of the interface for the Sonic Wire Sculptor?

Amit: When I built it, I built it for myself. The evolution of this thing started from looking at musical instruments. Musical instruments evolved over centuries to be as perfect as they are today. There's the guitar, the piano—revision after revision after revision of becoming perfect in the way we think, the way we move. It'd be really hard to make a better piano, but it's that kind of thinking that got me really interested in what would is important for the interface of an instrument. Turntables in particular are enjoyable to touch, so I knew when I was making an instrument that it had to be enjoyable to touch. I realized how much I enjoy touching the instruments that I know

how to play. That was an important part of learning to play them, and that's an important part of expressing myself with them.

The next step was constraining myself and saying, "Well, instead of asking what kind of sound and what kind of drawing, if I can draw a line—a simple, single wave line—what would that sound like?" Another issue in interactive art, and usability in general, is about how you map things into another, such as how you map a line to a sound.

If I wanted to hold a note, the same way as holding a note on a piano or when playing the trumpet, I don't want to move anything, but I still want that note to come out. That's one thing that must happen, so you must give a dimension there, where the note can actually visually extract, even if you're not actually moving anything.

Another thing was notation. I was really interested in notation, and I was really interested in the texture of the sound, rather than the notes themselves. I wanted the ingredient of the note, not the note. So if I made a three-dimensional sculpture of the piece and then I actually move it in space to see what happens to notation, I thought that would be interesting. So, all those things came together, and I don't know what came first, but they were all on my mind, and they made enough sense for me to put a lot of time into programming it.

Once I had created the Sonic Wire Sculptor, I got some interest from museums to show it in the context of art. When I was performing, people wanted to use it all the time, so I started the second version of the tool. I thought, "How do I make this enjoyable to the person who walks into the room and who hasn't seen this before and doesn't know what it is if I'm not there to explain it to them?"

It took me about six or seven shows to get it right. I noticed that if I just put it with the computer and the touch screen, people felt very intimidated. It just didn't work right, and it wasn't enjoyable. I couldn't break the ice and get people to touch it, without committing to it. And I noticed that when you're looking at artwork, like a drawing or painting in a museum, you control the level of engagement that you want to have with the piece. But with interactive work specifically like Sonic Wire Sculptor, it's all or nothing. So, how do you create an atmosphere where the viewer has control over the engagement? How do you create an atmosphere where the engagement pulls the viewer in and they want more of it?

Now, I have an actual device that looks like a moon crossed with a Russian missile control system that I fabricated out of wood and metal, and that wasn't arbitrary. You know, it's really retro, so when you come into the room and see this sculptural thing that's kind of heavy and bulky and weird, it makes sense. There's an embodiment to the sound, and it makes sense that sounds would come out of this. Then I put this thing in the center of the room, and I have a projection that mirrors everything that's happening on the interface of this instrument. So, you can be next to the tool, playing and looking at your little screen at the same time, and everybody sees what you're doing on the big screen. This is actually surround installation, and the sound actually spins around depending on the space of the sculpture. So, that got more people into it, and then I put couches and an air conditioner there. And the result is that now when I put it there, people will stay there, sometimes up to 40 minutes, and people will clap. The difference between the first time when no one wanted to stay in the room for more than

30 seconds and now is unbelievable! And pay mind that I almost didn't change anything in the software.

When you're in play mode, you don't mind failing, like killing yourself 50 times in a game and restarting a game, when you feel there's no consequence to failing. This is where you learn at your best. When I look at a piece of interactive art that doesn't have this learning scaffold—where you want to be playful with it, you want to engage with it, you want to try it out, and the system accommodates you to do that—I feel that that piece is lacking in soul. It's as extreme as that for me.

Sound in openFrameworks

It's certainly not this book's intention to overwhelm you with library names and strange acronyms, but they are an important part of working with openFrameworks. For many purposes, the code included in the openFrameworks core library will do just fine. In equally as many cases, though, you'll want to work with the underlying engines to tweak something, add some functionality, get a different value to work with, or do any of several tasks. When you glance into the *libs* folder of openFrameworks, you'll see the libraries with the classes that openFrameworks uses to make an openFrameworks application, specifically, `ofGraphics`, `ofTexture`, `ofSoundStream`, and so on. When working with sound, you'll work with two classes most of the time: `ofSoundPlayer` and `ofSoundStream`. `ofSoundStream` is used for more low-level access to the sound buffer and uses the RtAudio library developed at McGill University by Gary P. Scavone. RtAudio provides an API that lets you control and read data from the audio hardware of your computer. The other library used in openFrameworks for the `ofSoundPlayer` class is the FMOD Ex library. It provides more high-level methods to play and manipulate sounds.

You can manipulate sound with openFrameworks using two approaches. The first option is to directly manipulate the sound data sent from the sound card by using the `ofSoundStream` class that is included as a part of the core oF distribution. The second option is to use a library like the `ofxSndObj` add-on. This wraps the Sound Object library developed by Victor Lazzini to provide you with a powerful, flexible library that includes a wealth of tools for generating, manipulating, and mixing sounds. First, let's look at the `ofSoundStream` class.

The `ofBaseApp` class defines two callback methods that let you work with sound: `audioReceived()` is called when the system receives any sound, and `audioRequested()` is called before the system sends sound to the sound card. Both of these callbacks both require that the `ofSoundStreamSetup()` method is called before they will be activated. This tells the RtAudio library to start up, begin processing audio from the system microphone (or line in), and send data to the system sound card:

```
ofSoundStreamSetup(int nOutputs, int nInputs, int
 sampleRate, int bufferSize,  int nBuffers)
```

The `ofSoundStreamSetup()` method has five parameters:

`nOutput`
: Is the number of output channels that your computer supports. Usually this will be two: left and right. If you have a surround sound setup, it might be four or five.

`nInputs`
: Is the number of input channels that your system uses.

`sampleRate`
: Is usually 44,100 KHz, or CD quality, though you may want to make it higher or lower depending on the needs of your application.

`bufferSize`
: Is the size of the buffer that your system supports. At the time of writing this book, on any operating system, it's probably 256 values.

`nBuffers`
: Is the number of buffers that your system will create and swap out. The more buffers, the faster your computer will write information into the buffer, but the more memory it will take up. You should probably use two for each channel that you're using. Here's an example call:

```
ofSoundStreamSetup(2, 0, 44100, 256, 4);
```

The previous snippet will send two channels of stereo sound to the `audioReceived()` method each time the underlying RtAudio library sends information from the sound card. This should be called in the setup method of your openFrameworks application. Now, look at the first of two callback methods. The `audioReceived()` method is called whenever the system microphone detects sound:

```
void audioReceived(float * input, int bufferSize, int nChannels)
```

`input`
: Is a pointer to the array of data.

`bufferSize`
: Is the size of the buffer, the number of floating point values in the input array.

`nChannels`
: Is the number of sound channels represented in the sound data.

The input parameter is always an array of floating-point numbers with the length given in the `bufferSize` variable. This sounds a little tricky to work with, but as you can see, by using a `for` loop with a length determined by `bufferSize`, it isn't that difficult:

```
float samples[bufferSize];
for (int i = 0; i < bufferSize; i++) {
    // increment the sample counter
    samples[sampleCounter] = input[i];
}
```

Remember that the pointer to a float is actually just the first element in an array. If this doesn't ring any bells, look back at Chapter 5. Also, note that this callback won't be triggered unless you call `ofSoundStreamSetup()` with one or two channels set as the input, like so:

```
ofSoundStreamSetup(0, 2, 44100, 256, 4);
```

Next, the `audioRequested()` method is called when the system needs one buffer worth of audio to send to the sound card. The method sends the array of floating-point information that represents the buffer of audio data, the size of the buffer, and the number of channels:

```
void audioRequested() (float * output, int bufferSize, int nChannels)
```

To have the `audioRequested()` callback triggered by the system, you would need to call `ofSoundStreamSetup()` with one or two channels in the output. If you want to alter the data before it's sent to the sound buffer, you must do it within this method.

Now you'll use these two methods to create a way to record sound from the microphone and shift the pitch around. Example 7-1 shows the header file for the *audioReceived1* application, which is followed by the *.cpp* file for the application (Example 7-2).

Example 7-1. audioReceived1.h

```
#ifndef AUDIO_RECIEVED1
#define AUDIO_RECIEVED1

#include "ofMain.h"
#define LENGTH 220500 // 1 channel, 5 sec
class audioReceived1 : public ofBaseApp{
    public:

        float lsample[LENGTH];
        float rsample[LENGTH];
        float ltemp[LENGTH];
        float rtemp[LENGTH];

        bool recording;
        int lastKey;
        int sampleCounter;
        int playbackCounter;
        int recordingLength;
        void setup();
        void update();
        void draw();
        void keyPressed(int key);
        void octaveDown ();
        void octaveUp ();
        void audioReceived(float * input, int bufferSize, int nChannels);
        void audioRequested(float * output, int bufferSize, int nChannels);

};

#endif
```

Example 7-2. audioReceived1.cpp

```
#include "audioReceived1.h"

// set up the audio
void audioReceived1::setup(){
    ofBackground(255,255,255);
    // 2 output channels, 2 input channels, 44100 samples per second
    // 256 samples per buffer, 4 num buffers (latency)
    ofSoundStreamSetup(2, 2, this, 44100, 256, 4);
    recording = true;
}
```

The `audioReceived()` method notifies the application when sound has been sent from the microphone. If the recording variable is `true` and you haven't gotten too many samples, say 10 seconds worth, then capture the left channel input to the `lsample` array and the right channel input to the `rsample` array:

```
void audioReceived1::audioReceived (float * input, int bufferSize,      int nChannels)
{
    if(sampleCounter < 220499) { // don't get too many samples
        for (int i = 0; i < bufferSize; i++) {
            lsample[sampleCounter] = input[i*2];
            rsample[sampleCounter] = input[i*2+1];
            sampleCounter++;
        }
    } else  {
        if(recording) {
            recording = false;
        }
    }
}
```

Next, define the `audioRequested()` method. The values sent from the microphone were stored in the `lsample` and `rsample` arrays. If your application is recording, then you don't want to mess up the recording by writing data to the speakers, so send all zeros; otherwise, play back the sampled sound from the `lsample` and `rsample` buffers:

```
void audioReceived1::audioRequested(float * output, int bufferSize,
    int nChannels){
    if (!recording) {
        if(playbackCounter >= LENGTH) {
            playbackCounter = 0;
        }
        // loop over the buffer of samples
        for (int i = 0; i < bufferSize; i++) {
            // increment the sample counter
            output[i*2] = lsample[playbackCounter];
            output[i*2+1] = rsample[playbackCounter];
            playbackCounter++;
        }
    }
    // if we are recording, output silence
    if (recording) {
        for (int i = 0; i < bufferSize; i++) {
```

```
            output[i] = 0;
        }
    }
}
```

The `octaveDown()` and `octaveUp()` methods mirror one another. The `octaveDown()` method takes small sections of the `lsample` and `rSample` arrays, divides them by 2.0, and then adds the current value in each array to the next value to smooth out the sound. This helps avoid empty popping spots in the playback of the sound. The `octaveUp()` method works in much the same fashion, though it doesn't require the smoothing of values that the `octaveDown()` method does. This uses a technique called *windowing*, which is often used in digital signal processing to reduce noise and processing time when processing a signal. Windowing is a fairly complex topic and the mathematics behind it aren't entirely relevant to how you'll use it. The basic idea of it though is simpler to understand—a portion of the signal is taken and then clamped down to zero at both ends:

```
void audioReceived1::octaveDown (){
    int winLen = 5000;
    int halfWin = winLen / 2;
    if (!recording) {
        int numWins = int(LENGTH / winLen);
```

This is where the creation of the windows begins, looping through the sound from each channel, averaging each number with the following value, and storing it in the temporary array for that channel. Each value is spaced out more in the array, a somewhat crude way of lowering the tone of the sound:

```
for (int i = 0; i < numWins; i++) {
            int windowStart = i * winLen;
            for (int j = 0; j < halfWin; j++) {
             ltemp[windowStart + (j*2)] = lsample[windowStart + j];
             ltemp[windowStart + (j*2) + 1] = (lsample[windowStart + j] +
                 lsample[windowStart + j + 1]) / 2.0f;
             rtemp[windowStart + (j*2)] = rsample[windowStart + j];
             rtemp[windowStart + (j*2) + 1] = (rsample[windowStart + j] +
                 rsample[windowStart + j + 1]) / 2.0f;
            }
        }
        for (int i = 0; i < LENGTH; i++) {
            rsample[i] = rtemp[i];
            lsample[i] = ltemp[i];
        }
    }
}

void audioReceived1::octaveUp (){
    int winLen = 5000;
    int halfWin = winLen / 2;
    if (!recording) {
        int numWins = int(LENGTH / winLen);
```

Here the inverse of the `octaveDown()` method is used—every other value from the sample is used to populate the temporary array for each channel:

```
for (int i = 0; i < numWins; i++) {
         int winSt = i * winLen; // store the start of the window
         for (int j = 0; j < halfWin; j++) {
          ltemp[winSt + j] = lsample[winSt + (j * 2)];
          ltemp[winSt + halfWin + j] = lsample[winSt + (j*2)];
          rtemp[winSt + j] = rsample[winSt + (j * 2)];
          rtemp[winSt + halfWin + j] = rsample[winSt + (j*2)];
        } // now average the values a little to prevent loud clicks
        ltemp[winSt + halfWin - 1] =
                (ltemp[winSt + halfWin - 1] + ltemp[winSt + halfWin]) / 2.0f;
        rtemp[winSt + halfWin - 1] =
                (rtemp[winSt + halfWin - 1] + rtemp[winSt + halfWin]) / 2.0f;
        }

for (int i = 0; i < LENGTH; i++) {
            rsample[i] = rtemp[i];
            lsample[i] = ltemp[i];
        }
    }
}

void audioReceived1::keyPressed(int key) {
    if(key == 357) // up
        octaveUp();
    if(key == 359) // down
        octaveDown();
}
```

You'll notice that if you use the octave up and down methods repeatedly, the sound quality deteriorates very quickly. This is because the algorithms used here are pretty simplistic and lead to dropping values quickly. A much more sophisticated approach is to use a library called SMBPitchShift, which was written by Stephen Bernsee (his name will come up later in this chapter in the section "The Magic of the Fast Fourier Transform" on page 233). This library lets you change the pitch of a sound in semitones. A *semitone*, also called a *half step* or a *half tone*, is the smallest musical interval; an example is the shift from C and D♭. It also preserves the sound through the shifts in pitch with much greater accuracy than the previous example. Example 7-3 shows the header file for that library.

Example 7-3. SMBPitchShift.h

```
#ifndef _SMB_PITCH_SHIFT
#define _SMB_PITCH_SHIFT

#include <string.h>
#include <math.h>
#include <stdio.h>

#define M_PI 3.14159265358979323846
#define MAX_FRAME_LENGTH 8192
```

```
class smbPitchShifter
{
    public:
    static void smbPitchShift(float pitchShift, long numSampsToProcess,
      long fftFrameSize, long osamp, float sampleRate, float *indata,
      float *outdata);
    static void smbFft(float *fftBuffer, long fftFrameSize, long sign);
    static double smbAtan2(double x, double y);

};
```

For the sake of brevity, we'll omit the *.cpp* file for `smbPitchShifter`. You can check it out in the code downloads for this book in the code samples for Chapter 7. Understanding the methods that this function defines and what parameters they take is enough for the moment:

`float pitchShift`
: Is the amount to shift the pitch up or down. To determine the amount to shift, take the number of semitones that you want to step up or down, divide it by 12 to account for the number of notes, and since the harmonic scale is logarithmic, raise it to the power of 2:

```
int semitones = -1;    // go down one semitone
float pitchShift = pow(2., semitones/12.);    // convert semitones to factor
```

`long numSampsToProcess`
: Is the number of samples total that need to be processed.

`long fftFrameSize`
: Is the size of the frame that the method will try to shift at a given time. You should probably stick with 2,048.

`long osamp`
: Is the amount to overlap frames. Essentially, a higher number here, say 32, will produce higher-quality but slower shifts, while a lower number, like 4, will operate faster but with less quality.

`float sampleRate`
: Is the sample rate of your sound.

`float *indata`
: Is a pointer to the audio data that you want to pitch shift.

`float *outdata`
: Is a pointer to the data that will contain the pitch-shifted sound. In the following code snippets, the `indata` and `outdata` are the same.

If you've correctly placed `smbPitchShifter` in the *src* folder of your application, you can import it and use `smbPitchShifter` in the `octaveUp()` and `octaveDown()` methods, as shown here:

```
void audioReceived2::octaveDown (){
    int semitones = -3;    // shift up by 3 semitones
    float pitchShift = pow(2., semitones/12.);    // convert semitones
                                                    //to factor
    int arrayBitLength = LENGTH * sizeof(float);
    // the call to the memcpy method copies the values from the ltemp
        //array into
    // the lsample array
    memcpy(ltemp, lsample, arrayBitLength);
    memcpy(rtemp, rsample, arrayBitLength);
    smbPitchShifter::smbPitchShift(pitchShift, (long) LENGTH, 2048, 4,
        44100, ltemp, ltemp);
    smbPitchShifter::smbPitchShift(pitchShift, (long) LENGTH, 2048, 4,
        44100, rtemp, rtemp);
    memcpy(lsample, ltemp, arrayBitLength);
    memcpy(rsample, rtemp, arrayBitLength);
}

void audioReceived2::octaveUp (){

    long semitones = 3;    // shift up by 3 semitones
    float pitchShift = pow(2., semitones/12.);    // convert semitones to
                                                        //factor
    int arrayBitLength = LENGTH * sizeof(float);
    memcpy(ltemp, lsample, arrayBitLength);
    memcpy(rtemp, rsample, arrayBitLength);
    smbPitchShifter::smbPitchShift(pitchShift, (long) LENGTH, 2048, 4,
        44100, ltemp, ltemp);
    smbPitchShifter::smbPitchShift(pitchShift, (long) LENGTH, 2048, 4,
        44100, rtemp, rtemp);
    memcpy(lsample, ltemp, arrayBitLength);
    memcpy(rsample, rtemp, arrayBitLength);
}
```

One of the greatest strengths of C++ is its popularity. That popularity means that there is always a vast body of code written by researchers over the past 20 years and made available to the world for free by programmers and artists. You can almost always find a working example of something that you want to do.

The `ofSoundStream` part of the openFrameworks sound is simple, fast, and requires that you do most of the audio processing yourself. In the next section, we'll look at `ofSoundPlayer`, which leverages the powerful FMOD Ex library to provide some very useful shortcuts for playing back and processing sound.

openFrameworks and the FMOD Ex Library

The `ofSoundPlayer` class offers higher-level access and uses the FMOD Ex library developed by Firelight Technology. FMOD Ex is used in many major video games and is available for all the major operating system platforms: Xbox, PlayStation 3, and iPhone. If you look at the `ofSoundPlayer` header file, *ofSoundPlayer.h*, in the *sound* folder of *oF/libs*, you'll see some of the core functionality that the `ofSoundPlayer` class enables:

```
void     loadSound(string fileName, bool stream = false);
void     unloadSound();
void     play();
void     stop();

void     setVolume(float vol);
void     setPan(float vol);
void     setSpeed(float spd);
void     setPaused(bool bP);
void     setLoop(bool bLp);
void     setMultiPlay(bool bMp);
void     setPosition(float pct);    // 0 = start, 1 = end;
```

These methods are all very straightforward to understand and use, so in this section, we'll move on to a different aspect of the FMOD libraries: using the 3D sound engine.

FMOD Ex is the low-level sound engine part of the FMOD suite of tools. This library is included with openFrameworks. FMOD Ex input channels can be mapped to any output channel and output to mono, stereo, 5.1, 7.1, and Dolby Pro Logic or Pro Logic 2 with ease. The API includes a whole suite of 14 DSP effects, such as echo, chorus, reverb, and so on, which can be applied throughout the DSP mixing network. The API can play back *.wav*, *.midi*, *.mp3*, *.xma*, *.ogg* and *.mod* files. FMOD Ex also lets you work with 3D sound and supply 3D positions for the sound source and listener. FMOD Ex will automatically apply volume, filtering, surround panning, and Doppler effects to mono, stereo, and even multichannel samples.

Because the implementation of FMOD Ex in openFrameworks is all contained within `ofSoundPlayer`, looking at the *ofSoundPlayer.h* file will give you an idea of what sort of functionality is built-in. Take note of these two methods:

```
static void initializeFmod();
static void closeFmod();
```

These methods start up and close down the FMOD Ex engine. If you glance at the definitions of these methods in *ofSoundPlayer.cpp*, you'll see the calls to the FMOD Ex engine:

```
FMOD_System_Init(sys, 32, FMOD_INIT_NORMAL, NULL);  //do we want
                                                      //just 32 channels?

FMOD_System_GetMasterChannelGroup(sys, &channelgroup);
bFmodInitialized = true;
```

It's these sorts of calls that you're going to add and modify slightly. The FMOD Ex engine isn't set up to run in 3D sound mode the way that openFrameworks has implemented it. The solution is to create a new class that extends the `ofSoundPlayer` class that we'll call Sound3D.

Here's what the header file for that class looks like:

```
#ifndef SOUND_3D
#define SOUND_3D
#include "ofMain.h"
```

```
class Sound3D : public ofSoundPlayer {
    public:
        Sound3D();
```

These two methods are the most interesting:

```
        static void initializeFmod();
        static void closeFmod();
        void loadSound(string fileName, bool stream = false);
        void play();
        static FMOD_CHANNELGROUP * getChannelGroup();
        static FMOD_SYSTEM * getSystem();
};
#endif
```

The definition of the `Sound3D` class in Example 7-4 sets up the FMOD Ex library to operate in 3D mode.

Example 7-4. Sound3D.cpp

```
#include "Sound3D.h"
bool bFmod3DInitialized = false;

static FMOD_CHANNELGROUP * channelgroup;
static FMOD_SYSTEM       * sys;

Sound3D::Sound3D(){

    initializeFmod();
}

// this should only be called once
void Sound3D::initializeFmod(){

    if(!bFmod3DInitialized){
        FMOD_System_Create(&sys);
        FMOD_System_Init(sys, 32, FMOD_INIT_NORMAL, NULL);
        //do we want just 32 channels?
```

Here, the FMOD Ex engine is set to use 3D mode. Now that the two static variables are declared, the `FMOD_CHANNELGROUP` and the `FMOD_SYSTEM` are passed to the `FMOD_System_GetMasterChannelGroup()` method. The `FMOD_SYSTEM` instance is initialized and then the `FMOD_CHANNELGROUP` is set as the channel that the system will use:

```
        FMOD_System_Set3DSettings(sys, 10.0f, 10.0f, 10.0f);
        FMOD_System_GetMasterChannelGroup(sys, &channelgroup);
        bFmod3DInitialized = true;
    }
}
```

These two methods are provided to allow access to the `channelgroup` and the `sys` variables. These are used by the oF application to set the locations of the sounds and the listeners:

```
FMOD_CHANNELGROUP * Sound3D::getChannelGroup() {
    return channelgroup;
}

FMOD_SYSTEM * Sound3D::getSystem() {
    return sys;
}

void Sound3D::loadSound(string fileName, bool stream){
    result = FMOD_System_CreateSound(sys, ofToDataPath(fileName).c_str(),
        FMOD_3D, NULL, &sound);
    result = FMOD_Sound_Set3DMinMaxDistance(sound, 1.f, 5000.0f);

    if (result != FMOD_OK){
        bLoadedOk = false;
        printf("ofSoundPlayer: Could not load sound file %s \n",
            fileName.c_str() );
    } else {
        bLoadedOk = true;
        FMOD_Sound_GetLength(sound, &length, FMOD_TIMEUNIT_PCM);
        isStreaming = stream;
    }
}

void Sound3D::play(){

    FMOD_System_PlaySound(sys, FMOD_CHANNEL_FREE, sound, bPaused, &channel);
    FMOD_VECTOR pos = {  0.0f, 0.0f, 0.0f };
    FMOD_VECTOR vel = {  0.0f, 0.0f, 0.0f };
    FMOD_Channel_Set3DAttributes(channel, &pos, &vel);
    FMOD_Channel_GetFrequency(channel, &internalFreq);
    FMOD_Channel_SetVolume(channel,volume);
}
```

Now you're ready to create an actual application that uses FMOD Ex. All the positions of the listeners and the sound emanating from the channel are positioned using `FMOD_VECTOR` vector objects. Chapter 9 discusses vectors in far greater detail. A vector is an object with a direction and a length. In the case of a 3D vector, like the `FMOD_VECTOR` type, there are x, y, and z values that each represent a direction. FMOD uses four different vectors for the listener and two for the channel, as shown in Figure 7-5. The listener vectors represent the position, facing, relative up, and velocity of the listener, who is most likely the user. The channel vectors represent the position and velocity of the sound origin.

In the following header file (Example 7-5), you'll see the four vectors for the listener and the two for the sound defined, along with a pointer to the actual `FMOD_SYSTEM` variable that the `ofSoundPlayer` class defines. Other than that, the rest of the header file is rather straightforward.

Example 7-5. fmodApp.h

```
#ifndef _FMOD_APP
#define _FMOD_APP
```

```
#include "ofMain.h"
#include "Sound3D.h"

class fmodApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();
        void keyPressed( int key );
        void mouseDragged( int x, int y, int button );
        Sound3D player;
        // note these reference to the FMOD_SYSTEM
        // this is why you added the getSystem method to Sound3D
        FMOD_SYSTEM* sys;
        FMOD_VECTOR listenerVelocity;
        FMOD_VECTOR listenerUp;
        FMOD_VECTOR listenerForward;
        FMOD_VECTOR listenerPos;

        FMOD_VECTOR soundPosition;
        FMOD_VECTOR soundVelocity;
        bool settingListener;
        bool settingSound;
};

#endif
```

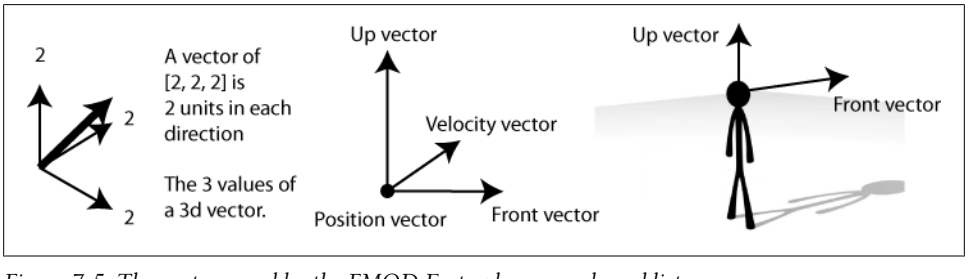


Figure 7-5. The vectors used by the FMOD Ex to place sounds and listeners

The *fmodApp.cpp* file contains a few other "newish" methods specific to the FMOD engine that require a little bit of explanation. First, all the vectors are initialized in the `setup()` method. Neither the listener nor sound is given a velocity here. You can experiment with changing these values on your own:

```
#include "fmodApp.h"
void fmodApp::setup(){

    listenerVelocity.x = 0;
    listenerVelocity.y = 0;
    listenerVelocity.z = 0;
    listenerUp.x = 0.f;
```

```
        listenerUp.y = 0.f;
        listenerUp.z = 0;
        listenerForward.x = 0.f;
        listenerForward.y = 0.f;
        listenerForward.z = 1.0;
        listenerPos.x = 3.f;
        listenerPos.y = 3.f;
        listenerPos.z = 1.f;
        soundPosition.x = 3.f;
        soundPosition.y = 3.f;
        soundPosition.z = 1.0;
        soundVelocity.x = 0;
        soundVelocity.y = 0;
        soundVelocity.z = 0.0;
```

Next, the player has a sound loaded from the openFrameworks application's data folder, and the `play()` method is called on it:

```
        player.loadSound("synth.wav");
        player.setVolume(0.75);
        player.setMultiPlay(true);
        player.play();
```

Next, use the `getSystem()` method that you added to the `ofSoundPlayer` class. You'll need to use the scoping resolution operator `::` to call the static method. If you're not familiar with this, take a look back at Chapter 5. Set the `sys` variable to the `FMOD_SYSTEM` instance that the `Sound3D` class has initialized:

```
        sys = Sound3D::getSystem();
    }

    void fmodApp::update() {
        if(!player.getIsPlaying())
            player.play();
```

Next, ensure that the sound loops, but check whether the player is playing. If it isn't, restart the playback of the *.wav* file. The listener attributes are set in the `update()` method of the oF application using the `FMOD_System_Set3DListenerAttributes()` method, and the channel attributes are set using the `FMOD_Channel_Set3DAttributes()` method. These are both rather longish methods and are filled with pointers, which means that you'll need to use the reference operator `&` in front of the vectors that you pass to these methods. Note also that you must pass the system needs to set the listener attributes, which is why the `getSystem()` method was added to the `Sound3D` class. Since this example has only one listener, just pass 0 for the listener and the correct vectors for the rest of the parameters to represent the position of the listener in 3D space:

```
    FMOD_System_Set3DListenerAttributes(FMOD_SYSTEM *system, int listener,
        const FMOD_VECTOR *pos, const FMOD_VECTOR *vel, const FMOD_VECTOR *forward,
        const FMOD_VECTOR *up);
```

Setting the properties of the channel is a little simpler. Pass the channel, which is the vector that represents the position of the sound in 3D space and its velocity:

```
FMOD_Channel_Set3DAttributes(FMOD_CHANNEL *channel, const FMOD_VECTOR *pos,
    const FMOD_VECTOR *vel);
```

Here are the actual calls that you'll want to add to your application:

```
    FMOD_System_Set3DListenerAttributes(sys, 0, &listenerPos,
        &listenerVelocity, &listenerForward, &listenerUp);
    FMOD_Channel_Set3DAttributes(player.channel, &soundPosition,
        &soundVelocity);
```

Right after this, you'll need to tell the system to update the sounds based on all the values that you set:

```
FMOD_System_Update(sys);
}
```

Next, in the `draw()` method, draw circles at the positions of the listener and the sound in the application window so you have a visual representation of the sound and listener positions:

```
void fmodApp::draw(){
    ofSetColor(0xff0000);
    ofEllipse(soundPosition.x * 100, soundPosition.y * 100, 10, 10);
    ofSetColor(0x0000ff);
    ofEllipse(listenerPos.x * 100, listenerPos.y * 100, 10, 10);
}
```

Allow the user to set the sound and listener positions by dragging the mouse. This could be anything—a user's movement in a space, an accelerometer, a joystick, or almost anything else that can create three numbers. The important thing is to get the three numbers:

```
void fmodApp::mouseDragged( int x, int y, int button ) {
    if(settingListener) {
        soundPosition.x = float(x) / 100.f;
        soundPosition.y = float(y) / 100.f;
        soundPosition.z = -2.0 - float( x/100.f )
    } else {
        listenerPos.x= float(x) / 100.f;
        listenerPos.y = float(y) / 100.f;
        soundPosition.z = -2.0 - float( x/100.f )
    }
}
```

Finally, to let the user toggle between editing the sound and listener positions, make any key press change the object for which the position is being modified:

```
void fmodApp::keyPressed( int key ){
    if(settingListener) {
        settingListener = false;
        settingSound = true;
    } else {
        settingListener = true;
```

```
            settingSound = false;
        }
    }
```

This is just the bare surface of what the FMOD Ex library can do. It can create up to 16 channels and position each of them in 3D space, navigate a listener or even multiple listeners through multidimensional space, create echoes based on the geometry of a space, simulate Doppler effects, and do a lot more. It lets you use sophisticated techniques to place, shape, and simulate sound. Having the ability to create complex and realistic sound helps with one of the main points of working with sound that was mentioned at the beginning of this chapter: sound gives us a lot of information about the world. In the case of a virtual world, correct sound goes a long way toward giving us information about a world that may not immediately be visible to the user.

The Sound Object Library

One thing that FMOD Ex doesn't do is generate new sounds. To create and mix complex new sounds in openFrameworks, you can use the Sound Object library (also called *SndObj*).

The Sound Object library is an object-oriented audio processing library created by Victor Lazzarini. It provides objects for the synthesis and processing of sound that can be used to build applications for computer-generated music. The core code, including sound file and text input/output, is fully portable across several platforms. Platform-specific code includes real-time audio I/O and MIDI input support for Linux (OSS, ALSA, and Jack), Windows (MME and ASIO), and Mac OS X (CoreAudio but no MIDI at the moment). The source code for the core library classes can be compiled under any C++ compiler.

At the core of the Sound Object library is SndObj (pronounced "sound object"), which can generate signals with audio or control characteristics. It has a number of basic attributes, such as an output vector, a sampling rate, a vector size, and an input connection, which points to another SndObj. Depending on the type of `SndObj`, other attributes will also be featured; for example, an oscillator will have an input connection for a function table, a delay line will have a delay buffer, and so on.

The Sound Object library provides tools for loading, playing, and analyzing audio files; creating sine, sawtooth, and square waves; mixing sounds; and recording to audio files. To make working with the Sound Object library a little easier in openFrameworks, you can download the ofxSndObj add-on from *addons.openframeworks.cc*. This means that you can use the Sound Object library in openFrameworks in two ways: on its own, which requires that you download and compile the library yourself, or using the ofxSndObj add-on, which requires only that you download and install ofxSndObj. For the purposes of this book, we'll concentrate primarily on using the add-on, though if you're interested, you can check out the Sound Object library's website and look at the examples of using it on its own.

The core of the ofxSndObj add-on is the `ofxSndObj` class. Any time you create an application that uses the ofxSndObj add-on, you need to make sure you include an instance of the `ofxSndObj` class. At a minimum, you'll need to follow these three steps:

1. Instantiate an instance of `ofxSndObj`.
2. Call either the `startOut()` or `startSystemInOut()` method to initialize the output and input for the Sound Object library.
3. Call the ofxSndObj add-on's `startProcessing()` method.

We'll now look at a skeletal application that uses the ofxSndObj library. The following is the header *.h* file:

```
#ifndef _TEST_APP
#define _TEST_APP

#define MACOSX

#include "ofMain.h"
#include "ofxSndObj.h"
#include "AudioDefs.h"

class testApp : public ofBaseApp{

    public:

        void setup();
        ofxSndObj sndobj;
};

#endif
```

The following are the definitions in the *.cpp* file:

```
void testApp::setup(){
    sndobj.startOut(true, 1);
    sndobj.startProcessing();
}
```

This creates the Sound Object library and starts the processing thread that will assemble and mix all the different sounds that you create and send them to your computer's sound card. Next, you'll create a pair of tones.

An *oscillator* is an object that creates repetitive variation, typically in time, of some measure about a central value (often a point of equilibrium) or between two or more different states. In Sound Object, oscillators are used to create tones. Just as the actual sound that our ear detects possesses a frequency at which the wave oscillates and an amplitude, the Sound Object oscillator object has a frequency, an amplitude, and a type of wave that it represents. The type of wave that the oscillator creates is determined by the `ofxSOTable` object passed to it. The `ofxSOTable` object contains a function that determines what changes to make to the signal every time it's processed. This table object is where you set the type of wave that the oscillator will create and how many

milliseconds the wave will play before repeating. This may seem a bit strange at first, but seeing an example should help you. First, we'll look at how the two objects are constructed.

The `init()` method for `ofxSOOscillator` looks like this:

```
void init(ofxSndObj& parent, ofxSOTable t, float frequency, float amp,
    ofxBaseSndObj b, int type);
```

The `ofxSOTable()` method accepts the following parameters:

`ofxSOTable table`
: Determines the oscillator, what kind of wave it will produce (`SINE`, `SAW`, `SQUARE`), and how long its cycle will be.

`float fr`
: Is the frequency offset of the wave, and it controls the pitch. By default this is 440 Hz.

`float amp`
: Is the amplitude offset, and it controls the loudness of the wave. By default this is set to 1.0.

`ofxBaseSndObj inputfr`
: Is the frequency control input, which is another `ofxSndObj` object. The fundamental frequency can be controlled by another object, letting you link the frequency that the wave produces to another object.

  The `init()` method for `ofxSOTable` looks like this:

```
void init(ofxSndObj& parent, long length, int type, float phase);
```

The parameters are shown here:

`long length`
: Is the table length.

`int type`
: Sets the wave that the table will generate to one of these preset wave shapes: SINE, SAW, SQUARE, or BUZZ.

Now, look at the following header file for an application using the ofxSndObj add-on. A lot of the extra methods have been stripped out to make it easier to read:

```
#ifndef _OFXSNDOBJ_OSCIL
#define _OFXSNDOBJ_OSCIL

#define MACOSX

#include "ofMain.h"
#include "AudioDefs.h"


class ofxSndObjOscilEx : public ofBaseApp{
```

```
    public:

        void setup();

        void mouseDragged(int x, int y, int button);

        ofxSOTable t1;
        ofxSOTable t2;
        ofxSOOscillator o1;
        ofxSOOscillator o2;

        ofxSndObj sndobj;
};

#endif
```

Since the objects are going to have their constructors called when the application is first run, you'll need to always make sure that you call the `init()` methods on each object. The general pattern for working with both the ofxSndObj and Sound Object libraries (if you choose to do so at some point) is to initialize the library, start communication with the system sound card, initialize any objects that you want to use in generating your sound, and then call the `startProcessing()` method of the main `ofxSndObj` object to process all the sound objects and create the final sounds. After that, it's easy to tune the sound objects, change their frequencies or amplitudes, change the components that go into a mix, and even add new objects. This is essentially the pattern that is on display here. All the objects are initialized in the `setup()` method, and the two oscillators are then altered by the user's mouse movement in the `mouseDragged()` method:

```
#include "ofxSndObjOscilEx.h"
#include "stdio.h"


void ofxSndObjOscilEx::setup(){

    ofBackground(255,0,0);
    sndobj.startOut(true, 1);

    t1.init(sndobj, 2500l, SINE, 1.f);
    t2.init(sndobj, 2500l, SQUARE, 1.f);

    o1.init(sndobj, t1, 1.f, 100.0f);
    o2.init(sndobj, t2, 200.f, 10000.f, o1, OFXSNDOBJOSCILLATOR);

    sndobj.setOutput(o2, OFXSNDOBJOSCILLATOR, 1);
    sndobj.setOutput(o2, OFXSNDOBJOSCILLATOR, 2);

    sndobj.startProcessing();
}


void ofxSndObjOscilEx::mouseDragged(int x, int y, int button){
    float xFreq, yFreq;
    o1.setFrequency(x * 3);
```

```
        o2.setFrequency(y * 3);
    }
```

The tones are a sort of "Hello, World" for the Sound Object library. One of the next steps to take is to begin working with loops, including creating them from other sounds, setting their duration, and mixing them together. This next example will use three of the ofxSndObj objects: `ofxSOLoop`, `ofxSOWav`, and `ofxSOMixer`.

`ofxSOLoop` lets you create and tune loops of other tracks. You can loop either an audio file loaded into the Sound Object library or a sound or series of sounds that have been created in Sound Object. The `init()` method of this class takes the following parameters:

```
void init(ofxSndObj& parent, float xfadetime,
 float looptime, ofxBaseSndObj inObject, int intype, float pitch);
```

`parent`
: References the main (or parent) `ofxSndObj` object that all `ofxSndObj` objects must use to communicate with the underlying system.

`xfadetime`
: Is the amount of time that the end of a loop will play over the beginning of the loop when it restarts. This is given in seconds.

`looptime`
: Is the length of the loop in seconds.

`inObject`
: Is the object that is the source of the loop.

`intype`
: Is the type of object that is being passed in as a loop. It's important to get this correct. Take a look at *ofxSndObj.h* for all the different types along with their integer values listed in the enum at the top of the header.

`pitch`
: Is the bend to the pitch of the loop.

`ofxSOWav` lets you load a *.wav* file into the Sound Object library. At the moment, MP3 files aren't supported. The following is the `init()` method:

```
void init(ofxSndObj& parent, string filePath, bool stereo, int channel);
```

`filePath` is the path to the sound file that is being loaded. The `stereo` and `channel` variables simply indicate whether the sound should be played in stereo and on which channel the sound should be played.

`ofxSOMixer` creates a mix of all the objects added to it. The `init()` method simply takes the reference to the `ofxSndObj` object to start the mixer, and all the objects are added using the `addObject()` and `removeObject()` methods:

```
void init(ofxSndObj& parent);
void addObject(ofxBaseSndObj b, int type);
void removeObject(ofxBaseSndObj b, int type);
```

Again, it's important that the correct types be passed in to the `addObject()` and `removeObject()` methods. Without these types, the mixer will not know what type of object is being added or where to find it. Now, let's move on to the example; we'll keep the code sample here short. The only references will be to those objects in the header file that are different from the previous example. First, declare `ofxSOMixer`, the two `ofxSOWav` instances, the two `ofxSOLoop` instances, and of course the `ofxSndObj` instance:

```
ofxSOMixer m;
ofxSOWav wavFile;
ofxSOWav wavFile2;
ofxSOLoop looper;
ofxSOLoop looper2;
ofxSndObj sndobj;
```

The `setup()` method calls the `init()` method of each of these objects, passing the `ofxSOWav` objects to `ofxSOLoop` to create their sources and passing the `ofxSOLoop` objects to `ofxSOMixer`:

```
void testApp::setup(){

    ofBackground(255,0,0);

    sndobj.startOut(true, 1);
    // initialize the mixer first
    m.init(&sndobj);
    // load the wav files
    wavFile.init(&sndobj, "tester.wav", true, 2);
    wavFile.init(&sndobj, "tester.wav", true, 2);
    // now create the two loopers
    looper.init(&sndobj, 1.f, 10, wavFile, OFXSNDOBJWAV, 1.f);
    looper2.init(&sndobj, 1.f, 10, wavFile2, OFXSNDOBJWAV, 1.f);
    // add the two loops to the mixer
    m.addObject(looper, OFXSNDOBJLOOP);
    // always remember to call startProcessing when you're ready
    // to have sndobj startup
    sndobj.startProcessing();

}
```

You can do a great deal more interesting work with the ofxSndObj add-on as well as with the Sound Object library. For example, you can input audio via a microphone, manipulate it, mix it with loaded sound, tune it with great precision, and then send it to audio files or speakers. There's much more information on the SndObj library at *http://sndobj.sourceforge.net/*.

The Magic of the Fast Fourier Transform

You've all probably seen a equalizer readout on a stereo or audio program that shows you how loud a song is in different frequencies. The lower frequencies are often shown in bands to the left, and the higher frequencies to bands in the right. There are a few different approaches to analyzing a sound this way, but the most common is called the

Fourier transform, named after the French mathematician Joseph Fourier. A transform is a mathematical term for the multiplying of a differential equation by a function to create a new equation that is simpler to solve. The Fourier Transform is very useful in signal processing, but involves a lot of computation and processing time. The *fast Fourier transform* is essentially a fast version of the discrete Fourier transform that was developed in 1969 so that Fourier Transforms could be calculated more quickly.

In looking at the fast Fourier transform (FFT), we'll give you two different explanations of varying difficulty. You'll see the terms *fast Fourier transform* and *FFT* everywhere in audio-processing code. In its most simplified and pragmatic form, the fast Fourier transform is used to decompose a signal into partial waves that can be more easily analyzed. When you want to analyze the spectrum or amplitude of each frequency of an audio buffer, you'll probably in one way or another be using the Fourier transform.

In a little more detail, the Fourier transform helps represent any signal as the sum of many different sine or cosine or *sinusoidal* waves (the name given to a wave that is a combination of a sine wave and a cosine wave). This is important because it lets you figure out how loud a complex signal is at a given frequency, for instance, 440 Hz. Complex signals are tested by taking a sine wave and multiplying it by many smaller signals to determine which smaller signals show the greatest correspondence to the complex signal that you're attempting to analyze. These smaller signals are either sine or cosine waves or sinusoids. You could almost imagine this as many small filters, each of which isolates a small frequency range in a signal and determines how loud those frequencies are in the given range. This is computationally and mathematically quite efficient, which is why it's in widespread use. In the discrete Fourier transform, for any number of samples that you're trying to determine a frequency for the signal at, you'll need to determine the same number of sine waves to approximate that signal over time. This is where the *fast* in the fast Fourier transform comes in. Imagine that the signal you're trying to process is a simple sinusoidal wave. You don't need to process the entire signal; you probably just want some information about its phase, frequency, and magnitude instead of sine and cosine waves from some predefined frequencies. The fast Fourier transform is a way of doing this quickly and without needing to calculate the number of values that the less efficient discrete Fourier transform requires. One of the drawbacks of the *fast* in the FFT is that it almost always requires a number of samples divisible by 2, as you'll see in both the openFrameworks and Processing examples.

Many distinct FFT algorithms exist that involve a wide range of mathematics, and you'll find a great body of literature related to Fourier transformations and to digital signal processing as a whole. An excellent tutorial, created by Stephen Bernsee, is available at *www.dspdimension.com/admin/dft-a-pied/*.

This doesn't quite explain why you would want to use the FFT, though. If you want to know how loud a sound is at a particular Hertz range, say, between 300 Hz and 325 Hz, you can use the FFT. This lets you analyze the tonal makeup of a complex sound. In the implementations of the FFT in Processing and openFrameworks, performing the

FFT returns a spectrum of sound. The spectrum does not represent individual frequencies, that is, 300 and then 301 and then 302, but actually represents frequency bands centered on particular frequencies. In other words, 300 to 325 is actually roughly an estimate of the amplitude of the sound at 312.

To use the FFT in openFrameworks, you need to create an `ofSoundPlayer` instance and use the `ofSoundGetSpectrum()` method, which looks like this:

```
float * ofSoundGetSpectrum(int nBands)
```

`nBands` is the number of slices of the sound spectrum that you want to receive. More slices means more accurate data, but it also means slower processing, so keep that in mind if you're using this to drive complex animations. The `float *` part means a pointer to a floating-point number. Remember that an array of floating-point numbers is really just a pointer to the first number in the array and an idea of the length of the array. To dynamically create an array, that is, create and store data on the fly, you simply need to give the address of the first element and have an idea of how many values are following it. If this is a little foggy for you, look back at Chapter 5. You'll see how the returned value is used in the code snippets that follow. The first step is to declare a pointer to a float in the header file of the application:

```
float * fftSmoothed;
```

Then, you can call `ofSoundGetSpectrum()` like so:

```
fftSmoothed = ofSoundGetSpectrum(128);
```

You can find a call to the `ofSoundGetSpectrum()` method in the example applications that come with the openFrameworks downloads:

```
int nBandsToGet = 128;
int numberOfFreqsToGet = 256;
float * val = ofSoundGetSpectrum(nBandsToGet);// request 128 values for fft
    for (int i = 0;i < nBandsToGet; i++){
        // smooth the values out a little
        fftSmoothed[i] *= 0.96f;
        // take the max, either the smoothed or the incoming:
        if (fftSmoothed[i] < val[i]) fftSmoothed[i] = val[i];
    }
```

Notice that here the values are smoothed out by comparing the previous values with the new values and taking the larger of the two. This prevents any really sudden drops.

The `draw()` method of that application has the values being used to draw rectangles, which will create a display not unlike the classic equalizer display of bars from an amplifier or audio player:

```
float width = (float)(5*128) / nBandsToGet;
    for (int i = 0;i < nBandsToGet; i++){
        // (we use negative height here, because we want to flip them
        // because the top corner is 0,0)
        ofRect(100+i*width,ofGetHeight()-100,width,-(fftSmoothed[i] * 200));
    }
```

In Processing, Minim provides a way to perform the FFT via the `FFT` object. You construct it as shown here:

```
FFT(int timeSize, float sampleRate);
```

For instance:

```
FFT fft = FFT(256, 44100);
```

To use the FFT implementation that the Minim library provides, you need to use an `AudioSource` object or some object that extends `AudioSource`. In this example, the `AudioPlayer` object is used to load an MP3 and then read the spectrum from that audio. Note the two `import` statements at the top. The first one is necessary to access the Minim `FFT` object:

```
import ddf.minim.analysis.*;
import ddf.minim.*;

Minim minim;
AudioPlayer player;
FFT fft;
String windowName;

void setup() {
  size(512, 200);
  minim = new Minim(this);

  player= minim.loadFile("song.mp3", 2048);
  player.loop();
  fft = new FFT(player.bufferSize(), player.sampleRate());
  windowName = "None";
}

void draw() {
  background(0);
  stroke(255);
  // perform the FFT on the samples in the mix of the songs, left and right channel
  fft.forward(player.mix);
  for(int i = 0; i < fft.specSize(); i++)
  {
    // draw the line for frequency band i, scaling it by 4 so
 we can see it a bit better
    line(i, height, i, height - fft.getBand(i)*4);
  }
  fill(255);
}

void stop() {
  // always close Minim audio classes when you finish with them
  player.close();
  minim.stop();

  super.stop();
}
```

You can also use the fast Fourier transform with the Sound Object library for openFrameworks. The `ofxSndObj` class defines a `createFFT()` method that you pass the object containing the sound for which you would like a sound spectrum along with the type of object that you're passing in. Here, you're passing an `ofxSOMixer` object to the `createFFT()` method:

```
sndobj.createFFT(table, m, OFXSNDOBJMIXER);
```

Note that you must also pass in an `ofxSOTable` object:

```
createFFT(ofxSOTable table, ofxBaseSndObj base, int type)
```

The base object that is passed in will provide the sound, while the `ofxSOTable` object defines any functions that will be performed on the sound to trim values from the returned spectrum. You can access the sound spectrum that is created by the Sound Object library by passing an array of floating-point numbers to the `getFFT()` method, like so:

```
float fft[1024];
sndobj.getFFT(fft);
```

Now you can use the `fft` array for creating graphics or performing any other kinds of calculations based on the returned spectrum. Take a look at the following application:

```
#ifndef _SNDOBJ_FFT
#define _SNDOBJ_FFT

#define MACOSX

#include "ofMain.h"
#include "ofxSndObj.h"
#include "AudioDefs.h"


class ofxSndObjFFT : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        ofxSOMixer m;
        ofxSOWav wavFile;

        float fft[1024];

        ofxSOTable table;
        ofxSOLoop looper;
        ofxSndObj sndobj;
};

#endif
```

Example 7-6 shows the *.cpp* file for this application. The `setup()` method contains all the initialization code for the Sound Object library, and the `update()` and `draw()` methods read the FFT spectrum and draw circles based on the amplitude values for each frequency.

Example 7-6. testApp.cpp

```
#include "testApp.h"
#include "stdio.h"


void testApp::setup(){

    ofBackground(255,0,0);
    sndobj.startOut(true, 1);

    m.init(sndobj);
    // load the file
    wavFile.init(sndobj, "tester.wav", true, 2);
    // create a loop
    looper.init(sndobj, 1.f, 10, wavFile, OFXSNDOBJWAV, 1.f);
    // add the loop to a mix
    m.addObject(looper, OFXSNDOBJLOOP);
    table.init(sndobj, 2500, 1, 1.f);
    // create the FFT based on the sound in the mixer
    sndobj.createFFT(table, m, OFXSNDOBJMIXER);
    sndobj.startProcessing();
}

void testApp::update() {
    sndobj.getFFT(fft);
    ofBackground(255, 255, 255);
}

void testApp::draw()
{
    ofSetColor(0, 255, 0);
    ofNoFill(); // draw empty circles
    int i;
    // the constant DEF_VECSIZE tells how large our system thinks
 a sound buffer should be
    for(i = DEF_VECSIZE; i>0; i--) {
        // draw a circle based on the values in the spectrum
        ofCircle(300, 300, fft[i] / (i*2));
    }
}
```

Physical Manipulation of Sound with Arduino

This chapter began by talking about the physical nature of sound. Yet we haven't really looked at any direct electrical manipulation of sound. We're about to rectify that. One of the most commonly used sensors for the Arduino controller is a piezo sensor. You

can find a more thorough discussion of piezoelectricity and how to find these sensors in Chapter 8, so the introduction to this sensor will be brief here.

Piezo sensors are usually used to detect vibration and pressure by outputting more electricity when they are bent or altered. However, it's possible to do more with a piezo sensor than detect pressure. In fact, one of the more common uses of a piezo sensor is in a buzzer. Many older electronics and primitive speakers use piezo elements. Their output range is quite limited, but for simple tones without a great deal of depth, a piezo is quite adequate. Thanks to some excellent work by David Cuartielles, who figured out the amount of time to push a common piezo sensor `HIGH` to replicate each note of the common eight-note scale, you can use Table 7-1 to play notes from a piezo sensor.

Table 7-1. Piezo sensor notes

Note	Frequency	Period[a]	Pulse width[b]
C	261	3830	1915
D	294	3400	1700
E	329	3038	1519
F	349	2864	1432
G	392	2550	1275
A	440	2272	1136
B	493	2028	1014
C (high C)	523	1912	956

[a] (Length of time It takes to make the tone)

[b] (Length of the high signal in the tone)

This concept of *pulse width* is one that will require a little bit more exploration. For the Arduino, the *pulse* is a short blast of 5-volt current, and the *width* is the length of time that the controller sends those 5 volts. You'll notice on your controller (or, if you have an Arduino Mini, in the schematic for your controller) that some pins are marked PWM.

A Quick Note on PWM

PWM stands for *pulse width modulation*, and it's important because your Arduino controller can't actually output analog voltages. It can output only digital voltages, either 0 or 5 volts. To output analog voltages, the computer uses averaged voltages, flipping between 0 and 5 volts at an appropriate interval to simulate the desired output voltage, as in Figure 7-6.

To make a tone—for example, an A—you send a 5-volt signal to the piezo sensor for 1,136 microseconds and then send a 0-volt signal or no signal to the piezo for 1,136 microseconds. In the following code, you'll see two arrays filled with the notes and their relative microsecond delays.

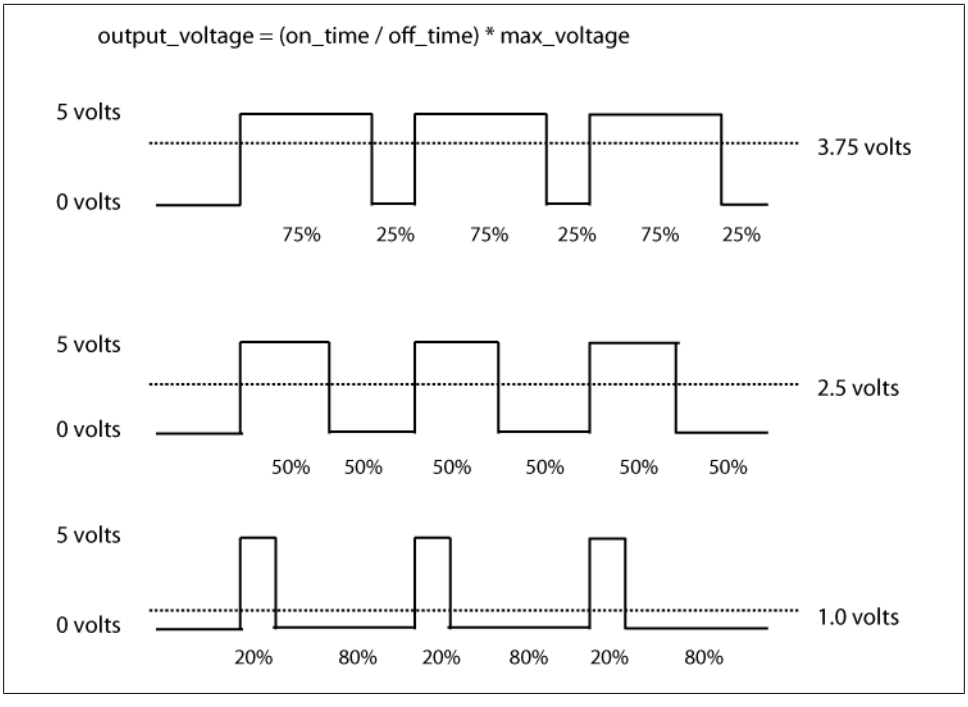


Figure 7-6. Pulse width modulation

```
// this enumerates the different notes
byte names[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
// enumerate the tones and the period required for them
int tones[] = {1915, 1700, 1519, 1432, 1275, 1136, 1014, 956};
```

The duration of the pulse is in *micro*seconds, not *milli*seconds; 1,136 microseconds is 1.136 milliseconds and 0.001136 seconds.

A melody can be input as a series of notes with lengths in front of them, for example, `4a`, which indicates a note with a length about four times the base length. Notice next that the notes in the `names` array correlate to the correct lengths in the `tones` array. For each note, the PWM value will be looked up in the `tones` array. The microsecond value is used to send a `HIGH` signal and then a `LOW` signal from the digital pin. The rest of the code that follows finds the correct note and adjusts the length of the note. This is because high notes will be shorter than longer notes since the amount of time in the signal required to make the sound is longer.

The piezo sensor will almost certainly have a red wire and a black wire. If not, look at the data sheet for your piezo sensor to determine which wire is the ground and which is the power. The power line is connected to digital pin 3 of the Arduino controller,

and the ground is connected to the ground pin on the Arduino. The rest of this code is carefully annotated for you:

```
int speakerOut = 3; // this is the pin that the piezo element should be connected to
// this enumerates the different notes
byte names[] = {'c', 'd', 'e', 'f', 'g', 'a', 'b', 'C'};
// enumerate the tones and the frequency required for them
int tones[] = {1915, 1700, 1519, 1432, 1275, 1136, 1014, 956};
// here's the melody we'll play, with the length of the note and the note itself
byte melody[] = "4c4d4e4f4g4a4b4C";
int eachNote = 0;
int noteLength = 0;
int findNote = 0;
int melodyLength = 8;
int theRightNote;

void setup() {
  pinMode(3, OUTPUT);
}

void loop() {
  //start our program
  digitalWrite(speakerOut, LOW);
  int correctedNoteLength;

  //loop through each notes in our melody
  for (eachNote = 0; eachNote < melodyLength; eachNote++) {
      // find the note that we're suppposed to play
      for (findNote=0;findNote<8;findNote++) {
        //store that note
        if (names[findNote] == melody[eachNote*2 + 1]) {
          theRightNote = findNote;
        }
      }

      // adjust the note because higher notes take less time to play
      // so we need to add some time to higher notes and subtract time
      // from lower notes
      int adjustmentAmt = (1450 - tones[theRightNote])*3;
      correctedNoteLength = (((melody[eachNote*2]) * 200) + adjustmentAmt) / 100;

      //make sure that we play the note for the length specified in the length
      for (noteLength = 0; noteLength <= correctedNoteLength; noteLength++) {
          digitalWrite(speakerOut,HIGH);
          delayMicroseconds(tones[theRightNote]);
          digitalWrite(speakerOut, LOW);
          delayMicroseconds(tones[theRightNote]);
      }
    }
  }
```

If you liked this, you can find an updated example of this code by David Cuartielles on the Arduino website and a more complex example that uses an actual speaker instead of a piezo element created by Alexandre Quessy.

Creating Interactions with Sound

Now that you've learned some basic sound generation and manipulation techniques, the question becomes, how do you create interactions with these libraries and techniques? FFT lets you do rather sophisticated analysis of sounds. Analyzing sounds lets you do things based on a pitch or on the amplitude of a sound in a particular frequency range. This lets you create an input based on the volume or pitch of people's voices, based on tones as old touch-tone telephones once did, or based on certain patterns of sounds. With all the libraries that have been presented in this chapter, we've just scratched the surface of what they can do, and all of them deserve a closer look.

Using the Minim library, you can create tones and sounds; load, play back, and process audio files; mix and loop multiple complex series of sounds; and apply effects to those sounds. With the Sound Object library, you can create tones, loops, mixes of sounds, and tones, and you can perform exact manipulations of those tones. This opens up two options: first, you can programmatically create audible feedback for a user or listener, and second, you can let users create their own sounds, loops, and mixes based on an interface and input system that you devise. Applying audio effects, pitch bending, tuning, and control of the timing of sounds is rich interactive terrain both as input and as feedback. Finally, with the FMOD Ex library, you can create 3D sound and sound effects to provide physically realized feedback that can position, respond, and inform a user.

Further Resources

For anyone serious about making computer sound, the following four tools are invaluable: PureData, Max/MSP, Csound, and SuperCollider.

PureData, created by Miller Puckette and maintained by Puckette and a large group of collaborators, is a real-time graphical programming environment for audio, video, and graphical processing. Max/MSP shares some core ideas and lineage with PureData and has a very similar interface: a graphical programming environment bolstered by a wealth of plug-ins. It's a commercial product and must be purchased, but it provides a substantial community and a wide range of tools that make it a worthwhile investment. It's very popular with composers, sound artists, and interactive artists, and has been used to generate music by DJs, in sound installation, and in live performances of all kinds all over the world.

Csound is a system and programming environment for creating sounds, mixing, and creating filters. Finally, SuperCollider is an environment and programming language released in 1996 by James McCartney. It consists of a server that processes commands to perform real-time audio synthesis and algorithmic composition and a scripting language that lets you pass commands to that server. Since then, it has been evolving into a system used and further developed by both scientists and artists working with sound.

It's an efficient and expressive dynamic programming language, and it's an interesting framework for acoustic research, algorithmic music, and interactive programming.

For understanding music, digital signal processing, and sound on a computer, *The Computer Music Tutorial* by Curtis Roads (MIT Press) is a must. It's a little bit dated at this point, but the explanations of fundamental concepts in signal processing, synthesizer generation, and the mathematics behind sound are invaluable.

HCI Beyond the GUI, edited by Philip Kortum (Morgan Kaufmann), also provides a lot of very valuable information for designers or artists thinking about working with sound as an interactive element, particularly for application development or more commercially or product-oriented design efforts. David Huron wrote a wonderful book on the topic of sound, surprise, and expectation called *Sweet Anticipation* (MIT Press) that addresses the psychology of music and sound. Leonard Meyer's *Emotion and Meaning in Music* (University of Chicago Press) is also a wonderful book on similar topics that would be well suited to anyone with a background in music.

If you're interested in working with electronics for music and audio production either by assembling components from low-level components or by repurposing other electronics the book, *Handmade Electronic Music* by Nic Collins (Routledge) is a gold mine of tutorials, information, and inspiration. It's also a fine primer for many of the basic concepts of electronics that we don't have space to cover in this book.

Voice user interfaces are one of the most tantalizing interactive elements of working with sound. Unfortunately, at this time they are also difficult to begin working with right away. In the interest of preventing this book from becoming out-of-date too quickly, we'll make only a few mentions of projects that might be of interest. A fully open-source library called Sphinx, developed at Carnegie Mellon University, was primarily intended to run on Linux but has been ported to Mac OS X and Windows. The Julius library has been developed over the past dozen or so years by a rotating team of researchers at Japanese universities. Currently, the project is headed up out of Kyoto University. The Julius project is being updated frequently and has a fairly large user base. It has successfully been used in iPhone applications and seems quite promising. By the time this book is published, better documentation and resources may be available for this Julius and Sphinx.

Review

Sound is a wave of air pressure that has both a maximum and minimum pressure as well as a frequency.

The sound card is a device that converts sounds into digital signals. Devices that need to communicate with an operating system use applications called drivers to exchange information with an operating system. Buffering is the technique of storing a small portion of the audio data received from the audio card.

In a Processing application, you can use the Minim library to load sounds and play them back by using the `load()` and `play()` methods of the Minim object. The `AudioOutput` class of the Minim library allows you to output sounds through your sound card, the `AudioInput` allows you to input sounds.

The Minim library allows you to create filters, for instance a LowPass filter, and waves, for instance a SquareWave or a SineWave.

Sound in oF is done through two classes `ofSoundPlayer` and the `ofSoundStream`. The `ofSoundPlayer` wraps the FMOD Ex library for higher-level sound access and the `ofSoundStream` uses the rtAudio library for lower-level access to data directly from the sound card.

Pitch shifting is a technique for changing the pitch of a sound by shifting the value of each number in the sound data up or down a certain amount. This often involves the use of technique called windowing, which is used to create a small section of a signal for analysis.

The FMOD Ex library can also be used to create 3D sounds that lets you place both the listener and the sound origin.

To generate sounds in oF, you can use the ofxSndObj library that uses the Sound Object library for oF to load sounds, create waves, and mix these together.

You can also play sounds with the Arduino controller by using Pulse Width Modulation to send signals to a peizo element at a specific interval to create a sound.

The *Fast Fourier Transform* is an algorithm used to determine how loud a signal is within a certain frequency range. This is often used to create equalizer views or to do rudimentary beat detection.

CHAPTER 8

Physical Input

Interaction design opens new possibilities of interaction between humans and technology. Physical computing is a particular type of interaction design that centers on the interaction between the human body and the computer. All the types of interaction explored in this chapter are driven by physical actions, both the sending of information out into the physical world and the sending of information from the physical world into the digital world. The power of physical interaction is in the connection of a physical action to an abstracted result, be that an auditory or visual result.

Some of the controls explored in this chapter are instantly familiar, such as knobs, buttons, and sliders, and some aren't so instantly familiar, such as accelerometers and infrared sensors. All the code will be Arduino, and we'll include diagrams to help you get your components connected to your board.

Interacting with Physical Controls

Human beings are flat perceivers; that is, we perceive the world as a flat 2D space. This is advantageous, evolutionarily speaking, because it's a natural limit to the amount of information humans can accept. However, humans are quite skilled at mentally manipulating a series of two-dimensional images into a fully realized three-dimensional space, assembling it piece by piece. To record information, we are quite accustomed to drawing flat representations, maps, and diagrams, with which a three-dimensional mental model can be mentally assembled. The classic way of turning two-dimensional information into three-dimensional information is to physically manipulate the object or environment. You want to see how it fits together, so you turn it over; you want to see the other side, so you walk around it. This physicality is an inseparable aspect of information gathering, and as anyone who has spent time interacting with alternative interfaces like a multitouch user interface or any other kind of haptic interface can attest, varying the physical interaction frequently leads to new experiences, new realizations, and new ways of thinking.

One of the great strengths of using physical interaction is the ability to separate the feedback from the action that the user takes. Anyone who has spent time making music can relate to the intuitiveness of turning knobs to speed or slow a beat or change a pitch. We have tools that help us physically and cognitively understand and work with an interface using physical objects that we manipulate with our hands.

In the simplest model of physical input: you press a button, and a light goes on. In a more complex model, you tilt and turn a sensor, and the tone a computer plays changes; you turn a pair of dials, and a camera pans and rotates. What these all have in common, of course, is that they're physically driven by controls that are instantly recognizable as controls. In this chapter, we'll focus on such controls. An entirely separate realm of physical computing uses implicit interfaces and focuses on gathering physical or environment data in ways that aren't directly controlled by the user, and we'll cover some of those later in this book. When you think about an explicit interface, you're dealing with quite different challenges than when you're dealing with an implicit interface.

All controls require some discovery to use—some instruction given to us by a combination of the controls and the labels around the controls. With anything that we make for others to use or that we ourselves learn to use, the creator must provide a discovery process that the participant actively takes part in. When we are faced with a new control or a new series of controls, those controls have unknown consequences and side effects. To understand those consequences and side effects, we have to discover how to work the equipment. How we approach engendering this discovery process, and specifically, how we engineer the discoverability of an object, is one of the most important decisions in making an application or device.

Any equipment that we interact with via controls has three elements to it: the things that we touch, the electricity that powers and communicates with the system, and the software that handles that communication. While you as the developer and creator will, of course, be concerned with all three, you have to remember that the user will deal only with the first.

Thinking About Kinetics

Here's an interesting question: is kinetic motion thinking? Well, yes. To paraphrase the psychologist Scott Kelso, it's important to remember that the brain and the mind that it engenders didn't evolve just to picture and analyze the world. The brain and mind are both parts of a larger system that always exists within a physical realm. A lot of research suggests that the ways in which humans think is an extension of their existence as embodied minds. The way that we think is driven by our physicality, and our physicality drives the way that we think. Your goal in interaction design is to tailor the design of your application to your users' capabilities and to the way that they process information and make decisions—in short, to the way that they think. Interacting with physical controls allows users to act and react kinetically.

We've all seen muscle memory in action, abstractly. An experienced guitarist isn't consciously aware of each finger. They see through the action to the task. In a sense, this is the goal of industrial interaction design: to allow workers to perform their action instinctively and without the need to consider each action but to instead consider its larger consequence. A task itself is often an impediment to seeing through it to the desired result. Most often when making an interactive system, the goal should be enabling the user to realize a goal or see through the interaction to the system below it. When we speak or listen to our native language, we don't have to actively think to process and produce the information. Physical interaction is much the same; we're adept with our hands, and we learn certain movements quickly and well enough that we can perform them before our conscious mind is fully aware of what we are doing. Tapping into this fundamental human ability lets your users engage your system more fully and develop habits around that system more quickly.

Muscle memory is more than the movement of the body. Many linguists have noted how native Chinese speakers frequently "write characters in the air" with their fingers when attempting to recall the stroke order for a particular character. This is a linguistic example, but it extends to other tasks. Ask a piano player to write down the keys that they hit in a piece of music they know. Usually, unless they can sit at the piano and use their muscle memory of the order of the keys, they can't do it. This is because the memory isn't a visual memory; it's isolated in the act of finger movements. A great many tasks are far easier to accomplish with a kinetic or physical interface.

Getting Gear for This Chapter

This chapter, and the rest of the Arduino chapters, will be a bit challenging because of all the controls and components you'll use. For each type of control we discuss, parts may go in and out of production and in and out of stock at popular electronics stores. Updates may be made to a component that may necessitate minor changes in the code shown here. But the controls we discuss are common and classic enough that they will likely remain consistent for several years. What follows is a list of essential gear that *all* the examples will use:

Arduino board
: This chapter and its diagrams assume that you're using or referring to the Arduino Diecimila or Duemilanove board. If you're using the Mini or the Nano, you can still run all the code in this chapter, but you'll need to check the differences between the two boards to make sure you're making connections to the correct pins.

Prototyping board
: This is just a board with electrical conductive metal and a plastic grid atop it that allows circuits to be created easily without soldering, saving you time, solder, and the pain of having to unsolder mistakes.

10 KiloOhm resistors (always written as 10K)
: You can find these at any hobby electronics shop and even at some hardware stores.

22- or 24-gauge single-core plastic-coated wire
: *Wire gauge* refers to the thickness of the wire. The smaller the gauge, the larger the wire; the thinner the wire, the easier it is to bend and turn. Getting several different colors of wire lets you follow what you're doing more easily when you create circuits with a prototyping board.

For everything else, please refer to the later section "Getting Gear for This Chapter" on page 247, which will include names, models, manufacturers, and likely sources for all the controls and components in this chapter.

Controlling Controls

We've already discussed how to attach a button to an Arduino controller, but now let's talk about what attaching a button to an Arduino controller "really does." The "really does" is in quotation marks, because any element in any interactive object exists as several different things. Even the simple button is rather complex.

The Button As an Electrical Object

A button works by completing a circuit external to the controller. The Arduino board sends a small amount of electricity to the button. When the button itself is pressed, the circuit is completed, and the input pin to which the button has been connected detects the change in voltage, alerting the controller that the button has been pressed.

The Button As an Interactive Object

Buttons are far more commonly encountered in the digital age than knobs, largely because buttons define a binary operation that shape a user's behavior so quickly: on/off, start/stop. The button guides behavior by breaking down all the possible interactions, even a complex series of interactions, into a set of binary oppositions. Even when the button controls a range, it controls the range in fixed increments. You push a button to increase or decrease the temperature of an oven 5 degrees for each click, or the volume of a stereo increases in fixed amounts with a click.

The Button As a Value in Code

The Arduino controller registers the button press as voltage passing to a pin on the controller. In Chapter 4, you saw how the button is wired to the Arduino controller: the button has two wires, voltage is passed through one wire, and the other is connected to a port that listens for a 5-volt signal. When the button is pressed, the circuit is completed, and the Arduino controller registers the voltage change. All the controls

that the Arduino interfaces with send electrical signals that the controller reads and processes in an application.

Turning Knobs

Buttons operate using a digital signal, which indicates two things: we read the button using the `digitalRead()` method (see Chapter 4), and the possible values for the button are `HIGH` or `LOW`, or 0 and 1. A button can't be anything other than pressed or not pressed. Analog signals, in contrast, can be a range of values. In Arduino, you read the analog values as integers from 0 to 1,023. These analog values correlate to the amount of voltage sent into the analog pin on the analog input of the controller. Any value from 0 to 5 volts means that each 4.8 millivolts correlates to an increase or decrease of an integer value.

The Dial As an Interactive Object

A dial fully represents a range and allows for a smooth transition from one value in the range to the next. Although buttons allow for greater precision in setting the value, they jarringly move from one state to the next. Imagine a stereo suddenly turned on or an engine roaring to life. Think of the dial for the volume of an old stereo or the hands of a clock.

Each turn of the dial increments or decrements a value in a fixed immutable amount, but we tend to experience these amounts as fluid changes. The hour hand of a clock seems to turn imperceptibly; turning the volume on a mixing console changes the volume subtly enough that the music seems to simply rise in volume.

Potentiometers

Potentiometers, perhaps known to you as knobs or a dials, work by receiving a current and increasing or decreasing the amount of voltage on a pin as it's turned. For the Arduino controller, this means that the potentiometer is connected to three places on the board (Figure 8-1) to provide the three things that the potentiometer needs to communicate with the board:

- Voltage (in this case, 5 volts from the +5-volt pin),
- A ground (from the ground bus),
- An analog input to read how much voltage is being returned (in this case, the An In 0 pin)

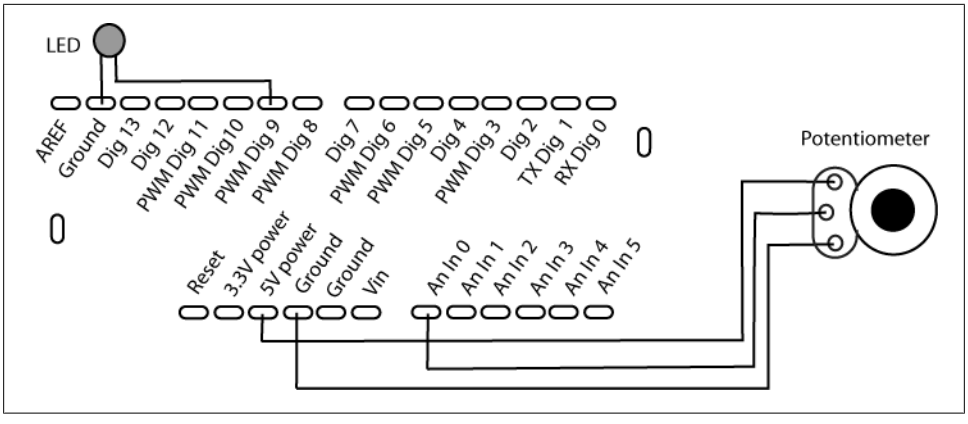


Figure 8-1. Connecting a potentiometer to an Arduino board

You can configure the Arduino board to read data from the analog pins by using the `analogRead()` method. This method, like the `digitalRead()` method, takes the number of the port that should be read and returns the value that is read from the port. In the case of analog data, that value can be from 0 to 1,023.

In the following code snippet, the analog value sets the blinking rate of an LED:

```
int potentiometerPin = 0;
int ledPin = 13;
int val = 0;

void setup(){
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);
}

void loop(){
    val = analogRead(potentiometerPin);
    Serial.println(val);
    digitalWrite(ledPin, HIGH);
    delay(val);
    digitalWrite(ledPin, LOW);
    delay(val);
}
```

To use the value from the potentiometer to set the brightness of the LED, you need to remember that analog output can be any value from 0 to 255, so you divide the analog input value by 4, converting the 0 to 1,023 range to the 0 to 255 range:

```
int potentiometerPin = 0;
int ledPin = 9;
int val = 0;

void setup(){
    // setup isn't doing anything, but we still need it here
}
```

```
void loop(){
    val = analogRead(potentiometerPin);
    Serial.println(val);
    analogWrite(ledPin, val/4);
}
```

A slightly different twist on the potentiometer is the 0.5-millimeter-tall soft potentiometer that provides a very different form factor that you can use for flatter, fingertip-tactile interfaces.

The soft potentiometer shown in Figure 8-2 is manufactured by Spectra Symbol and is widely available. A few other suppliers are creating them, so checking at any of the recommended suppliers should give you an idea of what's available for your region and price range. This particular soft potentiometer can be wired the same as a standard dial potentiometer, while providing a much different appearance and experience for a user.

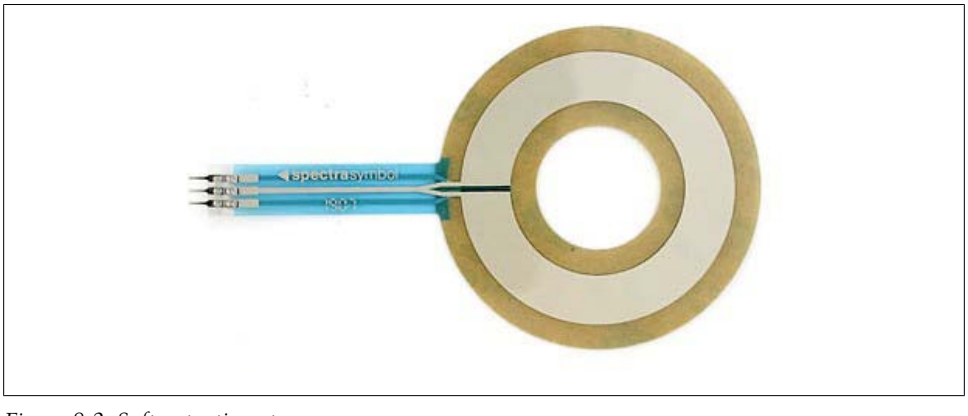


Figure 8-2. Soft potentiometer

Using Lights

A light is generally used as a feedback mechanism to tell users that something is on or has changed, to inform them of progress by blinking, or to warn them. When a light blinks, you instinctively pay attention to it. When a light suddenly comes on, you notice it. When a light goes off, you tend not to notice. If a light is always on, you'll notice only if you're searching for the cause of something gone wrong.

Lights are one of the oldest electronic feedback mechanisms because they're simple to engineer, easy to comprehend with a well-placed label or icon, and cheap.

Wiring an LED

Attaching a small light to the Arduino board is quite easy. Create a circuit with a 220 Ohm resistor to the 5-volt pin of the Arduino and then connect that to the longer leg of the LED. Connect the shorter leg of the LED to the ground pin of the Arduino.

Figure 8-3 shows the electrical diagram.

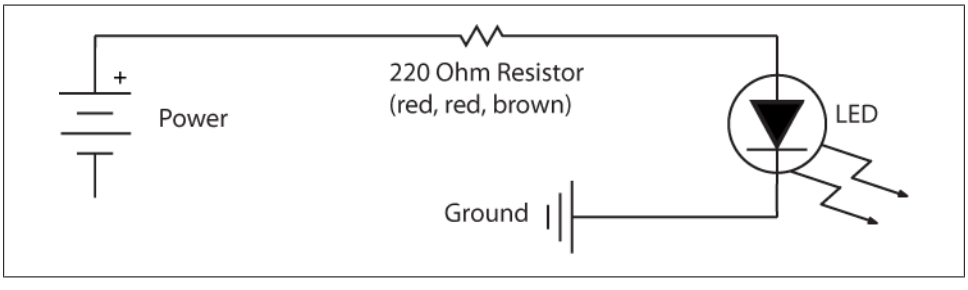


Figure 8-3. Connecting an LED

This raises an interesting question: how do you wire multiple lights? Since the light simply requires a complete circuit and enough voltage to power the bulb, you can easily add multiple lights to the circuit and power them all at the same time, as shown in Figure 8-4.

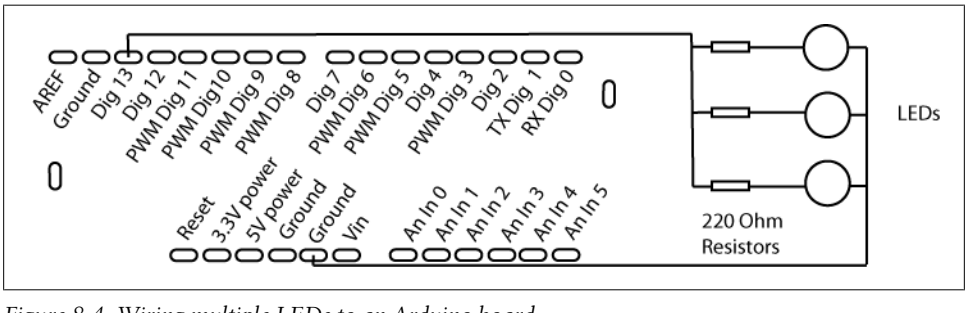


Figure 8-4. Wiring multiple LEDs to an Arduino board

When the lights are wired to the digital out, both are turned on or off by the digital port either sending a `HIGH` value or sending a `LOW` value. You can also send analog values to the lights by using the `analogWrite()` and using any one of the pins marked PWM to power the lights, but you'll need to add a resistor to the circuit:

```
int ledPin = 9;
int amountOfLight = 0;

void setup(){
}

void loop(){
```

```
        if(amountOfLight > 254) {
            amountOfLight = 0;
        }
        analogWrite(ledPin, amountOfLight);
        amountOfLight+=1;
        delay(10);
    }
```

Since the `loop()` method is going to repeat every 10 milliseconds, you can use that loop to slowly increase the amount of power that is sent to the lights, creating a slow linearly increasing glow. Of course, the next thing you might want to do is add multiple lights and control them independently. While this is fairly easy to do with a limited number of lights by using the same number of lights as digital out or analog out ports that the Arduino controller possesses, using more lights in more complex configurations is an entirely different matter. Chapter 11 will cover more complex ways of using LEDs in your applications.

Detecting Touch and Vibration

A *piezoelectric* sensor (*piezo* sensor for short) is a device that uses the piezoelectric effect to measure pressure, acceleration, strain, or force by converting these factors to an electrical signal. Piezo sensors use a phenomenon called *piezoelectricity*, the ability of some materials (notably crystals and certain ceramics) to generate an electric potential in response to physical stress. What this means is that the material of a piezo sensor, usually a crystalline coating over metal, returns more current when bent or otherwise disturbed. This lets you detect very slight changes in the sensor, for example, a breeze or the touch of a finger.

The principles behind the piezo sensor should give you an idea of how the sensor connects to the Arduino board. A certain amount of current is sent through the sensor, and that current is fed through the sensor. When the sensor is bent, the resistance of the sensor increases, and the amount of current returned from the sensor is reduced. Many electrical supply stores carry small hobbyist piezo sensors (Figure 8-5) that have two wires attached to them.

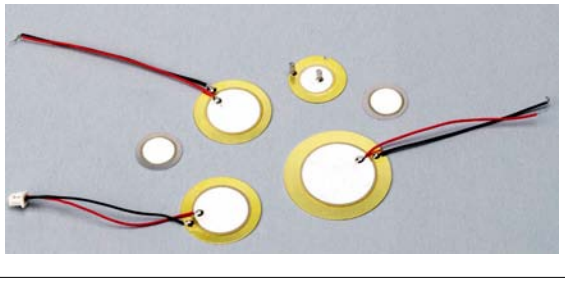


Figure 8-5. Piezo elements

To connect the piezo sensor, you'll want to connect 5 volts to the input of the sensor, the red wire, and connect the output of the sensor, the black wire, to an Analog In pin on the Arduino controller.

Reading a Piezo Sensor

The following code will detect vibration in a piezo sensor attached to your Arduino with 5 volts of power (+5V pin) connected to the red wire of the piezo and the other end of the piezo sensor connected to Analog In pin 2:

```
int piezoPin = 2;    // input for the piezo
int ledPin = 13;   // output pin for the LED
int val = 0;       // variable to store the value coming from the sensor

void setup() {
    pinMode(ledPin, OUTPUT);  // declare the ledPin as an OUTPUT
}

void loop() {
    val = analogRead(piezoPin);    // read the value from the sensor
    // a nice even number that could represent a little bit of change
    if(val < 100) {
        // if there's resistance on the piezo, turn the light on
        digitalWrite(ledPin, HIGH);
    } else {
        // otherwise, turn the light off
        digitalWrite(ledPin, LOW);
    }
}
```

Detecting force in a piezo sensor can also detect the amount of force up to a certain range; however, the piezo sensor reaches a maximum value quickly. If you want to detect more force than the piezo sensor allows, you can use a flexible pressure sensor. These sensors read the amount of force that is exerted on a flexible piece of electro-conductive material and commonly range from 1 to 100 pounds of maximum force detection. The sensor in Figure 8-6, manufactured by FlexiForce, is accurate from 0 to 25 pounds of pressure and is about 0.2 millimeters thick, making it unobtrusive and easy to conceal inside levers, soft objects, punching bags, and shoes, among other possibilities.

Figure 8-6. FlexiForce sensor

Getting Piezo Sensors

RadioShack stocks a small piezo sensor kit that consists of a piezo sensor held inside a small plastic case. Carefully open the case to remove the sensor, or simply leave it inside the case. It's easier to work with the sensor outside of the case, but it's also more delicate, so be careful not to disconnect the wires from the back or to tear the material of the piezo element itself. You can also buy piezo sensors from a well-stocked electronics supplier.

Interview: Tom Igoe

Tom Igoe is the author of *Making Things Talk* (Make Books) and *Physical Computing* (Course Technology). He teaches at the Interactive Telecommunications Program at New York University.

You wrote a great post on your blog about the themes that emerge again and again in physical computing. Why do you think artists and designers are drawn to these themes?

Tom Igoe: I think people return to the same ways of creating interaction because of the technology involved. Certain things like gloves, interactive mirrors, mechanical pixels, tilt controllers, and so on, all come up a lot because they're easy to do and they're things that are already popular. People tend to build what they already know exists. If you look at that list of themes and ideas from that blog post you mention, one of the things that really strikes me is that the interaction in almost all of them is quite vague. In other words, with the theremin, for example, you're waving your hands to make this "wooooo" sound. Your body is a kind of cursor, and as a result, people don't have to mess with the nuances of interactive design as much. You seldom see a class in an art school or a design school that focuses on designing a control panel or control surface, because, frankly, it's kind of boring to artists and designers who are just learning how to think about interaction design. They think, "Why would I want to design a power plant control system?" The thing is that power plant control systems are really cool, and there are a lot of interesting problems there, but you can't see that by looking at it purely aesthetically. It's only when you get people to think about it in a performative sense—how the needs of system and the needs of the users map to a set of controls—that you realize it's quite complex and quite interesting.

One of the biggest difficulties in beginning to work with interactive design is deciding which skills are important to you. I think people often don't realize how important being able to design an interaction is. Being able to design the functionality of a control panel is a real skill.

Tom: It's true—I think it's a skill-born experience. I think that you're always learning new skill sets, and a woodworker is always learning new tools and new variations on tools, too. But what I think you're referring to in terms of the practice of craft for years is that you do have to repeat and hone a skill. People who work a lot with their product, whether their product is a physical interface or software interface, have gained a craft that you can't teach easily. That's a natural extension from dealing with all the limitations that you run into. Often people who come from an industrial design background or an art background are used to the idea that they're making an object or product that

does one thing. They're the ones who have the most trouble establishing the boundaries of their project when they start to do physical computing projects. It's because it's the first time they use software, and software always makes them want to do too many things. For them to suddenly have the ability to do almost anything is a little mind-blowing for them. There's a difference between an application and a platform. When you're making something, you have to decide whether that thing is a platform or whether it's an application. I think if anything is the craft of building applications, it is having an understanding of the limitations that a particular task needs.

What are the most important and or difficult things for students to learn when it comes to physical interaction?

Tom: I think that one of the biggest challenges in the physical computing is learning to think specifically. If a designer can't describe the action that they want to bring about, then they can't really design for that action. The ones who can do it best are those who come from theater and performance—those who've taken an acting class—because they know that you can't get up on stage and think. You have to get up on stage and take an action, and that action has to communicate something. So, often I see students who say, "Well, then I'm going to make it possible for them to interact." I'll try to push them to be very specific, and they respond, "Well, you know, they're gonna make the thing happen." And that just doesn't work. I think that a lot of people who deal with physical interaction don't understand that you have to be that specific when you think about the interaction because at the guts of that "thing," somewhere there's a piece of software that's going to be looking for that series of actions.

How do you teach students how to make an interaction that can be understood by the user...that the interaction is appropriately discoverable?

Tom: People like to use the dirty word *intuitive* for that, and I always tell them that there's no such thing as intuition when it comes to user interface. I make students read the first two chapters of Donald Norman's *Design of Everyday Old Things* (Basic Books). They read it and realize that there is this idea of a mental model and of affordances and that you can lead a person to a decision. You should be leading people to the next thing to do, and there are some good "next" things to do. You have to organize things in a way that allows a user to read the interface and follow it along. It's really interesting to just sit down and analyze physical interfaces. The writer and designer Dan Safer does this quite a bit, as do I. If you do figure out how to lead people in the physical design of a layout and the performance of it, you solve a lot of your software and electronic problems because you don't have to write code to listen to every sensor in the system. The user is going to pay attention to the switch that's blinking, so you can stop checking the other 16. Someone else who's very bright about these things is Chris Crawford. His book *Art of Interactive Design* (No Starch Press) talks about conversational error messages. Recognizing that all device interaction is a conversation, it's an iterative loop of thinking and speaking. So often, artists, more so than designers, get really good at the speaking side of the conversation and don't develop the listening skills. This is the area where I find the great difference between people who are attempting to make interactive art as opposed to those who are attempting to make interactive design. The distinction comes in that art is primarily an act of self-expression, whereas design is primarily an act of facilitating communication. Artists have to make things that do something or say

something. This is one of the reasons *interactive art* is kind of a fallacy. I always tell students, look, it doesn't matter which side of the divide between art and design you fall on, but be clear on the distinction, and when you make your choices, be clear on which of the two is governing your choices.

You attended the Interactive Telecommunications Program at NYU as a student and now you teach there. What and why are things different now as opposed to when you were there as a student?

Tom: This program is often jokingly described as the center for the study of the recently possible. What that means is that we don't necessarily do bleeding-edge technology. We look at what's being done and say, "OK, how can this be applied?" It has to change every three years. When I first heard about the program, people were working on interactive video with laser discs and Mac Pluses. One of the interesting things here when I was a student was you had a lot of people trying to be the next Nam June Paik and doing big video installations. Over the years there was a shift from space-based applications like installations to device-based applications—partially out of its immediacy and partially out of recognition that you don't always get to build a space to show your work in. You might have to make things that don't need a space to be presented in. The cell phone really changed how people think about devices and communication. Streaming media has really been a constant line of research. What has stayed constant is the question of how you communicate with a user through the limitations of a device.

As someone who does physical computing and interactive design, in what sort of role in society do you see yourself? How do see yourself or refer to yourself?

Tom: I don't know what I call myself. I use the word *designer* a lot. It gets overused a bit and abused a bit. It has a lot of baggage, but I still find it useful because for me designers facilitate communication and facilitate action. I come from a theater design background where your role is to serve the action, and everything else is secondary. And nobody remembers your name. People in industrial design and people in architecture of course wouldn't necessarily agree with me. I very much doubt Santiago Calatrava feels that it's his role for everybody to forget his name. But nonetheless, he's still there to facilitate you getting things done.

Recognizing that a designer's job is to make it easier for people to do things through tools is another big part of it. *Facilitator* I think comes up a lot, too, because, again, what you're really doing is figuring out how to get this person to talk to that person. How do you get people interested in something? The answer is that you start with something that they're already interested in. Games are a great example of this, too. So, facilitation, direction, and engagement, all of that; ultimately my feeling is that whatever we are as technologists or designers, our role should be to serve, and to make people's lives better in some way.

Do you think there are fundamental things that everyone who wants to do things with interactive design or art has to learn?

Tom: Yes, but I can't say that there are specifics. So, for example, I think that everyone doing this should learn some programming regardless of the language. It doesn't mean you have to get good at it; you just have to know and understand it. In fact, some of

the best people coming out of the ITP program are crappy programmers, but they know what can be achieved through programming.

I think everybody should get his or her hands on digital hardware at some point. Understanding the connection between code and physical reality is huge. At the end of the day, everything we do is grounded in our physical reality, and you need to have a handle on that. An understanding of formative communication is the key, and that means a lot of different things. You need to understand that the communication is the performance. You need to have an understanding of what hardware can do and how it interfaces with a person and you need to understand that physical interaction is always a performance by the user.

How did you get involved with the Arduino project?

Tom: I got involved in 1995 when I visited the Ivrea school. Massimo Banzi told me that they were working with this board called Arduino, and I already knew Wiring at that point, which was Hernando Barragan's project. Massimo and David Cuartielles were using the Arduino in the workshop that we did. I saw it and I thought that I'd use it in my own classes and knew a lot of other people would use it in their classes, too. Later, they decided they needed a core team of developers, and I joined. What I do on the team is be opinionated. David Mellis does the software work; David Cuartielles, Massimo, and Gianluca Martino do the hardware work. I do a lot of the beta testing and looking at the design of the board. Occasionally, I'll contribute real hardcore technical things. I guess I'm a bit like the fifth wheel in that way; I remind them about things they may have forgotten.

What are the things that you find the most exciting in physical computing right now?

Tom: In the past couple of years, I've gotten interested in communication between things, and specifically between physical things, which is more possible than it has ever been. When I first started teaching physical computing, you just didn't do wireless, right? Now, you can spend 40 bucks and buy two XBee wireless controllers, and in 15 minutes you have wireless communication. So, that's huge, and I think that changes things in terms of interactive design, too, because even the people who are the most technology obsessed now are interested in tackling the problem of getting two things to talk to each other. So, they're designing a conversation. It's easy to step from there to "Now why don't you design a conversation between two people?" What's gone on with social networking is huge. I think it's interesting when people start to think about the information flow of social networking. How can we incorporate that flow of information in the everyday environment for use by other people because we're already letting it be used when it's online. What happens when my daily thoughts are on the wall, what happens when it's on my car, or anywhere else?

On a technological level and on an ecological level, it's still not simple and cheap to do WiFi from a microcontroller. When it is, in terms of both money and electrical power, that's a change I'm very interested in.

The other one is sustainable technology development. I'm curious to see that get easy; by that, I mean I want it to be so easy to think not only about how to make something but also how to unmake it that people automatically think about this stuff ecologically.

By that I mean they think about the destruction of the thing as a fuel for technical nutrients. That's still a ways off, but I think it's essential, and I think it's really interesting. There's such an obsession with how to make things, and it's almost seductive to get hooked into how to make it, not even worrying about what it does. I'd love to see that same kind of seductive energy linked into how to recycle it.

Communicating with Other Applications

So far, we've covered getting input, which is one half of the interaction or conversation; however, at some point, you'll probably want to use the input from the Arduino to communicate with other applications built in Processing and openFrameworks. Communicating with Processing and openFrameworks is done via the serial port. In Chapter 4, we looked at using the `Serial` class in an Arduino application to send debugging information to the Arduino IDE. So, thinking about that for a moment, you're sending a message from the Arduino controller to the computer and listening to that message. That's not some magic in the Arduino IDE; it's something that all modern computers are able to do. Therefore, you can use the same `Serial` methods to communicate with a Processing or openFrameworks application.

So, what is the `Serial` object and what is serial communication? It's a way of sending information from your computer to another machine that could be another computer but is more frequently another kind of device. Serial communication uses a protocol called RS-232. This protocol dictates how messages are sent and received using the serial port. Many devices use serial communication to communicate with a parent computer: Bluetooth-enabled devices, GPS sensors, older printers and mice, bar-code scanners, and, of course, the Arduino board. How do you do it? Well, it's pretty low-level and can be pretty simple or quite complex. To send a message from a Processing application to the Arduino board, the event flow is like the following:

```
import processing.serial.*;

// Declare the serial port that we're going to use
Serial arduinoPort;

void setup() {
    // uncomment the next line if you want to
    // see all the available controllers
    // println(Serial.list());
```

Now you set up the communication between the Arduino and the Processing application. The following line may need to be changed if the Arduino isn't connected on the first serial port of your computer:

```
    arduinoPort = new Serial(this, Serial.list()[0], 9600);
}

void draw() {}
```

```
void keyPressed() {
    // Send the key that was pressed
    arduinoPort.write(key);
}
```

When you send the key, it will be the ASCII representation of the key, which means that on the Arduino side, you need to listen for a number, not a character. This is important.

The constructor of the `Serial` object has a few different signatures:

```
Serial(parent, name, rate)
Serial(parent)
Serial(parent, rate)
Serial(parent, name)
```

`parent:PApplet`
: Is the Processing application that owns the `Serial` object. You usually just use `this`.

`rate:int`
: Is the serial rate at which you're communicating; 9600 is the default, so if you've set your Arduino controller to communicate at a different rate, you'll want to change this.

`name:String`
: Is the name of the port. If you uncomment the `Serial.list()` call in the previous code snippet, you'll see the names of all the available ports.

You can create the serial port in some other ways, but we'll leave those for you to explore, because at the moment, we need to move on to the Arduino code.

The new stuff here is all in the `loop()` method of the Arduino application:

```
int message = 0;    // for incoming serial data
int ledPin = 13;

void setup(){
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);    // opens serial port, sets data rate to 9600 bps
}

void loop() {
```

The `Serial` objects `available()` method tells how much data has arrived and is available on the serial port from the Processing application. This is important because were you to not wait until you were sure that all the messages had been received, you might get incorrect data. It's important to always make sure that there is something in the buffer before you try to use it. In the following example, you'll turn the light on if you get a `!` from the Processing application, and you'll turn the light off if you get a `?`. Otherwise, just ignore it:

```
    // do something only when we receive data:
    if (Serial.available() > 0) {
        // read the incoming byte:
```

```
            message = Serial.read();
             if(message == '!') {
                digitalWrite(ledPin, HIGH);
             }

             if(message == '?') {
                 digitalWrite(ledPin, LOW);
              }
        }
    }
```

In openFrameworks, you send messages to an Arduino controller by using the `ofSerial` object. It's quite similar to the `Serial` object that Processing contains, so the methods that it defines should be familiar to you:

`bool setup(string portName, int baudrate)`
: Starts up the communication between the openFrameworks application and the Arduino board using the name of the port, something like `"/dev/tty.usbserial-A6004920"` if you're on OS X or `"COM1"` if you're on Windows.

`bool setup(int deviceNumber, int baudrate)`
: Starts up the communication between the openFrameworks application and the Arduino board using the device number. This can be problematic if you have anything else plugged into your computer and don't know the device number of the Arduino port.

`int readBytes(unsigned char * buffer, int length)`
: Reads multiple bytes from the Arduino into a character array.

`int writeBytes(unsigned char * buffer, int length)`
: Writes multiple bytes out to the Arduino.

`bool writeByte(unsigned char singleByte)`
: Writes a single byte out to the controller.

`int readByte()`
: Reads a single byte from the controller. This is an int, so if the controller sends a character, you'll get its ASCII representation.

In the previous Processing application, you simply sent the ASCII key number over the serial port to the Arduino board. To duplicate the same functionality in openFrameworks, you'd simply do what's shown in Example 8-1.

Example 8-1. OFSendSerial.h

```
#ifndef _OF_SEND_SERIAL
#define _OF_SEND_SERIAL

#include "ofMain.h"

class OFSendSerial : public ofBaseApp{

    public:
        void setup();
```

```
        void keyPressed(int key);
        ofSerial serial; // here's our ofSerial object that we'll use
};
#endif
```

You would implement it as shown in Example 8-2.

Example 8-2. OFSendSerial.cpp

```
#include "OFSendSerial.h"
void OFSendSerial::setup(){
    serial.setup("/dev/tty.usbserial-A6004920", 19200);

}

void OFSendSerial::keyPressed  (int key){
    serial.writeByte(key);
}
```

And that's all there is to it.

Sending Messages from the Arduino

To have a Processing or openFrameworks application receive messages from Arduino, you need to make sure that something is listening in the application. In Example 8-3, you send a message to the serial port if digital pin 13 is `HIGH`, which could be any digital signal, a button press, or something similar.

Example 8-3. Arduino

```
int buttonPin = 13;

void setup() {
    // open the serial port at 9600 bps:
    Serial.begin(9600);
}

void loop() {
    if(digitalRead(buttonPin ) == HIGH) {
        Serial.print("!");
    } else {
        Serial.print("?");
    }
    delay(200);
}
```

In the following Processing code (Example 8-4), you listen on the port for any incoming information from the serial port and print the data that you receive from the Arduino controller. Of course, you can do far more sophisticated things with the data you receive, but for this demonstration, let's keep it simple.

Example 8-4. Processing

```
import processing.serial.*;

Serial arduinoPort;
void setup() {
    // set up the communication between the Arduino and the Processing app
    arduinoPort = new Serial(this, Serial.list()[0], 9600);
}

void draw() {
    // Expand array size to the number of bytes you expect
    byte[] inBuffer = new byte[7];
    while (arduinoPort.available() > 0) {
        inBuffer = arduinoPort.readBytes();
        arduinoPort.readBytes(inBuffer);
        if (inBuffer != null) {
            String myString = new String(inBuffer);
            println(myString);
        }
    }
}
```

openFrameworks

Communication between an Arduino board and an openFrameworks application uses the `ofSerial` object. The `ofSerial` class defines the following methods:

`void enumerateDevices()`
: Prints out to the console all the devices that are connected to the serial port.

`void close()`
: Stops the openFrameworks application from listening on the serial port.

`bool setup()`
: Is the setup method without the port name and rate, by default it uses port 0 at 9600 baud.

`bool setup(string portName, int baudrate)`
: Uses the `portName` to set up the connection.

`bool setup(int deviceNumber, int baudrate)`
: Uses the `devicenumber` if there are multiple devices connected.

`int readBytes(unsigned char * buffer, int length)`
: Reads any bytes from the buffer that the Arduino controller has sent.

`int writeBytes(unsigned char * buffer, int length)`
: Writes some data to the serial buffer that the Arduino controller can receive.

`bool writeByte(unsigned char singleByte)`
: Writes a single byte to the serial buffer for the Arduino controller.

`int readByte()`
: Reads a single byte from the buffer; it returns –1 on no read or error.

`void flush(bool flushIn, bool flushOut)`
: Flushes the buffer of incoming serial data; you can specify whether both the incoming and outgoing data are flushed. If there's some data in the serial buffer, it will be deleted by calling `flush()`.

`int available()`
: Returns 0 if the serial connection isn't available and another number, usually a 1, if it is.

We show a common pattern for working with the `ofSerial` application here. First, define the number of bytes that you're expecting. A Boolean value, like a button being pressed, would just be a single byte. The values from a potentiometer would be integers and so would be 2 bytes. This is the header file for the application:

```
#define NUM_BYTES
```

Next, define the application:

```
class SerialDemo : public ofBaseApp {
    public:
        void setup();
        void update();
```

Next, use a Boolean value to determine whether you should try to read from the buffer. An openFrameworks application runs faster than the serial protocol can send and receive information, so you should try to read from the serial buffer only every five frames or so. This value will be set to `true` when you're ready to read from the serial port; otherwise, it remains `false`:

```
        bool bSendSerialMessage;    // a flag for sending serial
        // this will be used to count the number of frames
        // that have passed since the last time the app reads from the serial port
        int countCycles;
```

Here's the `ofSerial` object:

```
        ofSerial serial;
};
#endif
```

In the *.cpp* file for your application, the `setup()` method of your application will need to call the `setup()` method of the `ofSerial` object:

```
void testApp::setup(){
    bSendSerialMessage = false;
    serial.enumerateDevices(); // this will print all the devices
    // this is set to the port where your device is connected
    serial.setup("/dev/tty.usbserial-A4001JEC", 9600);
}
```

The `update()` method of `ofApplication` should do all the reading from and writing to the serial port. When the `bSendSerialMessage` flag is set to `true`, the openFrameworks application sends a message to the Arduino board telling it that it's ready to receive data:

```
void testApp::update(){
    if (bSendSerialMessage){

        // send a handshake to the Arduino serial
        sIf the erial.writeByte('x');
        // make sure there's something to write all the data to
        unsigned char bytesReturned[NUM_BYTES];
        memset(bytesReturned, 0, NUM_BYTES);
```

The openFrameworks application reads bytes from the serial port until nothing is left to read:

```
        // keep reading bytes, until there's none left to read
        while( serial.readBytes(bytesReturned, NUM_BYTES) > 0){}
    }
```

Once you've read all the data out of the serial port and into the `bytesReturned` variable, set the `bSendSerialMessage` flag to `false`:

```
    // wait a few cycles before asking again
    bSendSerialMessage = false;
}
```

Now you can use the signal sent from the Arduino however you'd like. The `countCycles` variable is incremented until it hits 5; then the `bSendSerialMessage` variable is set to `true`, and the next time the `update()` method is called, the openFrameworks application will read from the serial port:

```
    countCycles++;
    if(countCycles == 5) {
        bSendSerialMessage = true;
        countCycles = 0;
    }
}
```

Detecting Motion

Motion detection is one of the most immediately satisfying experiences to create for a user or viewer, and it's also quite simple to implement. You can use motion detection in an environmental way, with the motion detection not being apparent to the user, or you can use it more like a control to signal the computer. The difference between being apparent or hidden for the user is determined by the placement and obviousness of the control. A motion detector for security should not be apparent to the user, or it will not be as effective in securing the area around it. A motion detector to detect whether a person is putting their hands under a faucet to start the flow of water, while not readily visible, is a control that the user is aware of, for example.

PIR Motion Sensor

You provide motion detection for your application in several ways. One is a passive infrared (PIR) motion sensor, shown in Figure 8-7.

Figure 8-7. PIR motion sensor

If your PIR sensor comes with a connector like the one shown in Figure 8-7, you may need to cut this off so you can connect the sensor to the Arduino controller, as shown in Figure 8-8.

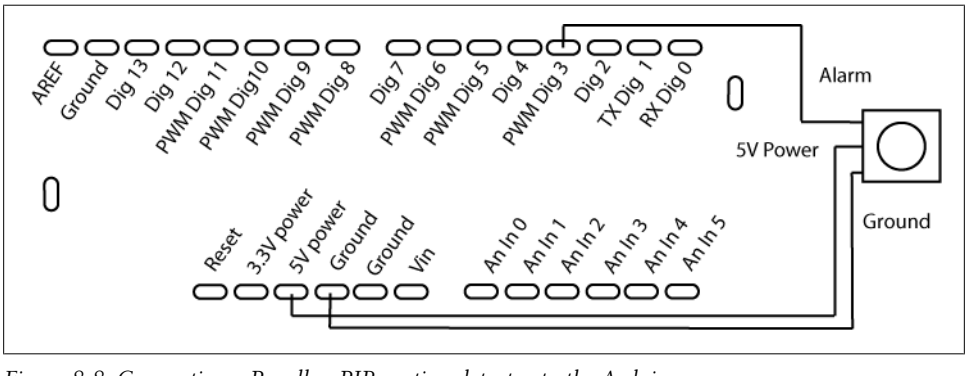


Figure 8-8. Connecting a Parallax PIR motion detector to the Arduino

The following code will light the LED on the Arduino controller when motion is detected by the PIR sensor. The source code is quite simple:

```
int alarmPin = 3;     // motion alarm output from PIR
int ledPin = 13;      // output pin for the LED
int motionAlarm = 0;
```

```
void setup(){
    pinMode(ledPin, OUTPUT);
    digitalWrite(ledPin, LOW);
}

void loop(){
    motionAlarm = digitalRead(alarmPin);
    // this is a very simple loop that lights an LED if motion detected
    if(motionAlarm == HIGH){
        digitalWrite(ledPin, HIGH);// we've detected motion
        delay(500);
        digitalWrite(ledPin, LOW); // turn off the light
    }
}
```

The kind of motion detection offered by the PIR motion detection sensor is really useful only for very simple interactions. Combined with another system, though, it can be quite powerful. Applications or machines that turn on or enter a ready state when a user approaches create a powerful affordance: preparedness. Other, more precise ways exist for not only detecting motion but also gathering more information about the motion and position of an object; you'll learn about these methods in the next section.

Reading Distance

Some sensors provide information about the distance to an object, and these are extremely useful because they detect presence and distance. While a simpler motion sensor can tell you whether something has moved within its range, a distance sensor will also tell you how far away that thing is. They're used extensively in hobbyist robots because they're simple to program, do not require a great deal of power, and provide very important information, such as how far away objects are. With that sort of information, you can go further than simply giving a user or audience binary feedback (such as you're moving or you're not) and give them analog feedback (such as you're this far away, now this far, now this far). It creates a more fluid interaction and experience because, while it may start rather suddenly when a person comes into range of the sensor, it does not end there.

You'll see two technologies widely used: ultrasonic and infrared. *Ultrasonic* sensors work by detecting how long it takes a sound wave to bounce off an object. The sensor provides an output that indicates the amount of time it takes for the echo signal to return to the sensor, and the magnitude of the echo delay is proportional to distance. One of the most popular lines of ultrasonic sensors is the Daventech Range Finder series, which has several different models varying in price and the range at which they function. The two most common are the SRF04 and SRF08. The SRF04 is a good and cheap sensor that is accurate to about 4 meters. The SRF08 is a little more expensive and has a range of about 8 meters.

Daventech sensors provide an output signal that has a pulse width proportional to distance. Using the Arduino `pulseIn()` method, this signal can be measured and used as a distance value:

```
long echo = 0;
int usPin = 9; // Ultrasound signal pin
long val = 0;
void setup() {
    Serial.begin(9600);
}

long ping(){
    pinMode(usPin, OUTPUT); // Switch signalpin to output
    digitalWrite(usPin, LOW); // Send low pulse
    delayMicroseconds(2); // Wait for 2 microseconds
    digitalWrite(usPin, HIGH); // Send high pulse
    delayMicroseconds(5); // Wait for 5 microseconds
    digitalWrite(usPin, LOW); // Holdoff
    pinMode(usPin, INPUT); // Switch signalpin to input
    digitalWrite(usPin, HIGH); // Turn on pullup resistor
    echo = pulseIn(usPin, HIGH); // Listen for echo
```

Now the value returned from the `pulseIn()` method can be used to determine how far away the object is in centimeters from the ultrasonic sensor by multiplying the echo by 58.138. This value comes from the manufacturer's specifications sheet, so you might want to check your sensor before assuming that this code will work. If you want the value in inches, you can multiply the value by 0.39:

```
    long ultrasoundValue = (echo / 58.138);
    return ultrasoundValue;
}

void loop() {
    long x = ping();
    Serial.println(x);
    delay(250); //delay 1/4 seconds.
}
```

Try adding a video or audio application written in Processing or openFrameworks that uses an ultrasonic sensor to control the speed of a video or audio file playing. Another common use of distance sensing is to detect presence or the absence of presence: starting an application or device when motion is detected within a certain range or when something that the device is programmed to expect is absent. An alarm on a door or window could function this way, as could a simple trip wire type of interaction. Motion detection is also an interesting possibility for mobile or handheld devices if you indicate to the user how to direct the sensor. This allows them to point the device at an object or surface to activate a reaction to it based on distance. Used in tandem with an accelerometer within the same handheld device or as part of a large set of controls, buttons, or potentiometers, a distance sensor can be a rich input for an interface to utilize.

Another way to measure distance, which uses technology borrowed from autofocus cameras, is *infrared*. The infrared sensor has two lenses, the first of which emits a beam

of infrared light and the second of which detects any infrared light reflected back. If the second lens detects any light, then the angle of the beam is measured by an optical sensor and used to determine the distance. The greater the angle, the greater the distance. The Sharp Ranger line is a popular choice, particularly for measuring accurate distances up to a meter.

Figure 8-9 shows how to wire an infrared sensor to your Arduino board.

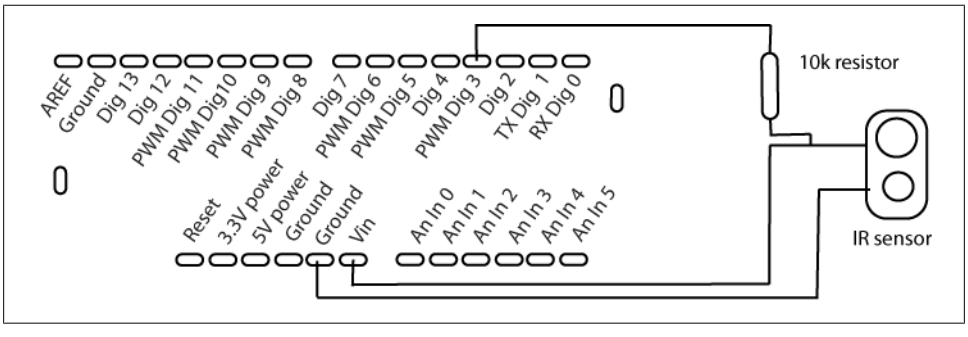


Figure 8-9. Connecting an infrared sensor to the Arduino board

Reading Input from an Infrared Sensor

To read data from an IR sensor, you need to initialize the pin that will be connected to the IR sensor and use the `analogRead()` method to read values from the sensor.

Generally, the sensor will report slight differences in readings from the sensor. You can smooth out your reading using the average of several readings. This may not be necessary, but if you find that your readings are slightly erratic, then using an average may be the answer. The following snippet is an example of averaging the readings from 10 IR sensor readings:

```
#define NUMREADINGS 10

int readings[NUMREADINGS];              // the readings from the analog input
int index = 0;                          // the index of the current reading
int total = 0;                          // the running total
int average = 0;                        // the average

int inputPin = 0;

void setup()
{
    Serial.begin(9600); // start serial communication
    for (int i = 0; i < NUMREADINGS; i++)
        readings[i] = 0; // initialize all the readings to 0
}

void loop()
{
```

```
    total -= readings[index]; // subtract the last reading
    readings[index] = analogRead(inputPin); // read from the sensor
   total += readings[index]; // add the reading to the total
    index++;

    if (index >= NUMREADINGS) {// if at the end of the array
        index = 0; // start again at the beginning
    }
    average = total / NUMREADINGS;// calculate the average
    Serial.println(average); // send it to over the Serial
}
```

Understanding Binary Numbers

As you may have heard at some point, computers and electronic components do not generally use decimal numbers. That means values like 17, 4, and 1,977—while certainly recognizable and usable in any computer program—aren't recognizable by the computer at runtime in that decimal form. Instead, computers and some sensors use binary counting. What follows is a quick introduction to binary numbers and, more importantly, how binary numbers and bit shifting work.

Binary Numbers

When you count from 1 to 13, you'll notice that you're counting up to 9 and then shifting a 1 over; then, you're continuing counting until you get to the next 9:

9, 10, 11....19, 20, 21....29, 30, 31...99, 100, 101...999, 1,000, 1,001

Notice how every time a 9 appears in the last place, you increment the value on the left and put a 0 on the right, and if all the numbers are nine, you add a 1 to the front. This counting system is called *decimal*. It's something so automatic that it takes a moment to realize that this is just one counting system among many. A binary counting system is similar to a decimal counting system, but instead of using 10 numbers to count, it uses only 2, which is why it's called *bi*nary.

When you count in binary numbers, the numbers reset every two values, so to count to four, you do the following:

1, 10, 11, 100

Notice how every time you have a 1 in the right column, you add another 1 on the left and transform the number on the right to a 0. Replicating the counting that you did in decimal system would look like this:

1001, 1010, 1011...10011, 10100, 10101...11101, 11110, 11111...1100011, 1100100, 1100101, 1100110...1111100111, 1111101000, 1111101001

That requires more numbers for certain, but why? Remember that underneath all the niceties that a computer provides to make it easier for us to work with code and numbers, the computer represents everything by the presence or absence of an electrical

current. For representing values simply by a presence or an absence, what could be better than a numbering system with two values: 0 and 1?

Bits and Bit Operations

Now that you have a rough idea of how counting in binary works, let's look at using objects that are represented in binary using bit shifting.

Bit shifting means shifting bits around by places. Although the uses of it are a little tricky to understand, the principles behind it aren't. Consider a situation where people are discussing salaries. Since they all make salaries that are measured in the thousands, they might say things like "52K" or "65." What they're doing, really, is shifting the numbers by three places so that 48,000 becomes 48. Now, some information isn't explicitly expressed, but if you know you're talking in thousands, then no information is actually lost. If you know that when you say "70" you really mean 70,000, it isn't a big deal, because you can always mentally "shift" those three places. The same works with percentages. Everyone knows that the "50" in a 50/50 chance is really 0.5. You just shift the value up two places to make it easier to think about because percentages use nice round numbers and those are easier to think and talk about. Bit shifting is essentially the same principle, except that shifting in binary is a bit more natural for a computer than for a human so it looks a bit strange at first. Bit shifting looks like this:

Let's say you have the integer 3:

```
int val = 3;
```

You know that as an int this variable has 16 bits available to it, so that, as far as your computer is concerned, you're really storing this:

0000 0000 0000 0011

and all those extra zeros are just placeholders. The only things doing anything are the last two numbers. Normally, this isn't a big deal at all, because computers have enough memory that wasting a few extra spaces is nothing to get bent out of shape about. That's fine for a laptop or an Arduino controller, but sometimes for very small sensors that's not fine, and they might do something like send you an integer in two pieces. And that could be a problem. Why? Well, what if you had a relatively big integer, for example, 30,000?

0111 0101 0011 0000

If you had to send that integer as two pieces in binary the first time, you'd get the following:

0111 0101

This is 117 in decimal, and the second time you'd get the following:

0011 0000

and this is 48 in decimal. Now, even if you know that you're supposed to be getting something kind of like 30,000, how are you going to turn those two numbers into what you're expecting back? The answer lies in the magic of shifting. Let's say that you're getting a value from an accelerometer sensor that sends information in two pieces. Since you know that the sensor is sending the first 8 bits of the number and then the second 8 bits (that is, the two halves of the integer), just shift the first 8 bits over and then put the next 8 bits behind it. To do this, use a bit shift operator:

`<<`
: Returns the value on the left *up* by the number of places indicated on the right.

`<<=` and `+=`
: Sets the variable on the left by shifting up by the number of bits indicated on the right, the same way that `+=` adds the value on the right to the value on the left:

```
int val = 3;
val <<= 1;
```

In the code snippet above, the variable `val` is now 6, because 3 in binary is 11, so moved over 1, it's 110, or 6.

Back to the sensor example, say that the total is 30,000, or 0111 0101 0011 0000 in binary. To shift the integer values over correctly so you get the first half of the number, you'd do the following:

```
int valueFromSensor = firstHalfOfValue; //this is 0111 0101
valueFromSensor <<= 8; //now our value is shifted over correctly
// since the second half won't affect the top, just add it
valueFromSensor += secondHalfOfValue;
```

Another bit shift operation is the right shift, which works the opposite of the left shift. It's not used as commonly in the types of code that you'll encounter because there isn't as much of a need to make big numbers into small ones, but it's available, and it's good to know:

`>>`
: Shifts the value on the left *down* by the number of places indicated on the right.

`>>=`
: Shifts the value on the left *down* by the number of bits indicated on the right, just like `+=` adds the value on the right to the value on the left:

```
int val = 100;// val is 1100100
val >>= 2; // removing the last two digits, val is now 11001 or 25
```

So, to recap in a slightly more academic fashion, you have this:

```
00010111 << 1 =  00101110
00010111 >> 1 =  00001011
```

In the first case, the left digit is shifted out, and a new 0 was shifted into the right position. In the second case, the last digit, a 1, was shifted out, and a new 0 was placed

into the left position. Multiple shifts are sometimes shortened to a single shift by some number of digits. For example:

```
00010111 << 2 =  01011100
```

A word of warning and a word of explanation as well: for the Arduino controller, an integer is 16 bits, that is, a binary number with 16 places. It's very important to note that *the first of those digits indicates whether the number is negative or positive*, so the number is only 15 digits long, with a final bit to indicate either a positive or negative number. Here's a few integer values:

```
0111111111111111 = 32767
```

Now, you would think that the value of that number if it was shifted to the left would be 65,534, but it's not, because the first bit says whether the number is negative or positive. It would actually be the following:

```
1111111111111110 = -2
```

Why is that important? When you start bit shifting numbers around, you may encounter some surprise negative numbers. This would be one source of those sorts of errors.

Why Do You Need to Know Any of This?

The complete answer lies at the end of the section "Introducing I2C" on page 278, where you begin working with I2C. Beyond that, though, this section was an excellent introduction to how the fundamentals of a computer actually function. It's also good to understand how bits and bit shifting actually works, because when working with sensors and the Arduino controller, you'll frequently see the bit shifting operators used in example code. As mentioned earlier, frequently smaller electrical components do not use larger numerical values and so will assume that any controller accessing their data can assemble those larger values using bit shifting. Now that you've read this section, you'll be able to follow along with how those controls send data.

Detecting Forces and Tilt

Accelerometers are sensors that detect forces acting upon them, specifically, the force of acceleration. Much the same way as you can detect the acceleration of your car when you press on the gas by the feeling that you have of being pushed back into the seat, the accelerometer detects the force of any acceleration onto it by detecting the shift in the forces acting on the internals of the sensor and reporting how much they've changed. Two-dimensional accelerometers detect forces registering changes in the x- and y-axes, and three-dimensional accelerometers detect changes in the x-, y-, and z-axes.

In this example, you're going to use an SEN-00849, which is a simple board that holds the accelerometer and provides easy access to its x, y, and z pins. Later in this chapter,

we'll look at another accelerometer, but for now, the SEN-00849 is easy to use; match the pin to an Analog In, and you're off to the races.

Figure 8-10 shows the wiring diagram to connect to an accelerometer.

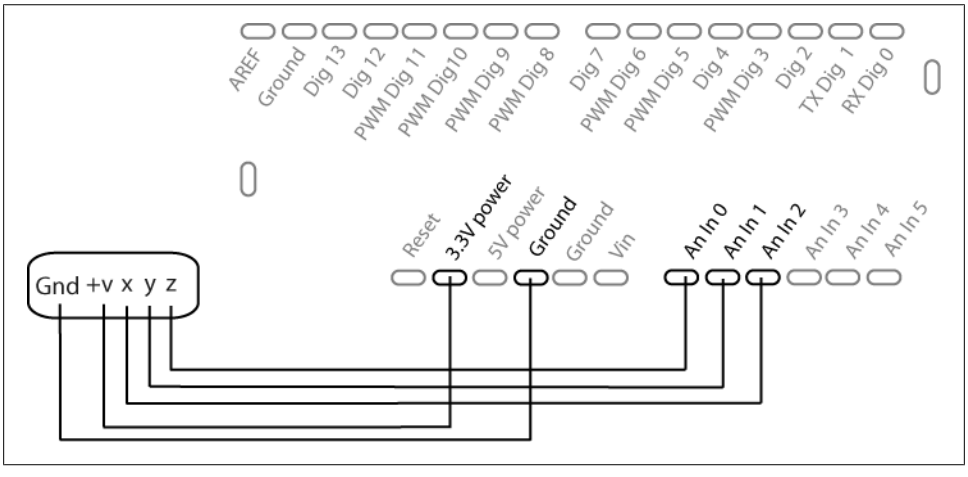


Figure 8-10. Connecting an accelerometer to the Arduino controller

The following snippet is a simple example of reading from an accelerometer using the `analogRead()` method to read the value for the plane that the accelerometer reports values for:

```
int xpin = 2;         // x-axis of the accelerometer
int ypin = 1;         // y-axis
int zpin = 0;         // z-axis (only on 3-axis models)

void setup()
{
    Serial.begin(9600);
}

void loop()
{
    Serial.print(analogRead(xpin));
    Serial.print(" ");
    Serial.print(analogRead(ypin));
    Serial.print(" ");
    Serial.println(analogRead(zpin));
    delay(50);
}
```

Although this is a simple application, it's an effective one. Once you get the values from the accelerometer, you can do all kinds of things with them, such as create a drawing tool, create a pointer, control a sound, change a game, or rotate in a 3D world.

In the following example, you're going to send a message from an accelerometer to an openFrameworks application. First up is the code for the Arduino application. You'll

see a few familiar things here, including the averaging of data from the controller, just like in when you saw a sketch with the infrared sensor. As you'll notice after working with an accelerometer for any length of time, the data from the accelerometer is jumpy, varying quite quickly by as much as 100 with no input. To avoid this, you'll simply average that value over eight readings:

```
int groundpin = 18;             // analog input pin 4
int powerpin = 19;              // analog input pin 5
int xpin = 5;                   // x-axis of the accelerometer
int ypin = 3;                   // y-axis
int zpin = 1;                   // z-axis (only on 3-axis models)

int xVal = 0;
int yVal = 0;
int zVal = 0;

int xVals[8];// an array of the last 8 x coordinate readings
int yVals[8];// an array of the last 8 y coordinate readings
int zVals[8];// an array of the last 8 z coordinate readings
int xAvg = 0; // the x value we'll send to our oF application
int yAvg = 0;// the y value we'll send to our oF application
int zAvg = 0;// the z value we'll send to our oF application

int currentSample = 0;

void setup()
{
  Serial.begin(19200);
}

void loop()
{
    // we use currentSample as an index into the array and increment at the
    // end of the main loop(), so see if we need to reset it at the
    // very start of the loop
    if (currentSample == 8) {
       currentSample = 0;
    }

    xVal = analogRead(xpin);
    yVal = analogRead(ypin);
    zVal = analogRead(zpin);

    xVals[currentSample] = xVal;
    yVals[currentSample] = yVal;
    zVals[currentSample] = zVal;
```

Here is where the values are averaged to avoid having strong spikes or dips in the readings:

```
    for (int i=0; i < 8; i++) {
        xAvg += xVals[i];
        yAvg += yVals[i];
        zAvg += zVals[i];
    }
```

These will under read for the first seven cycles, but that shouldn't make a huge difference unless you need to read the value from the accelerometer right away, in which case you could just not send the data for the first seven cycles:

```
            xAvg = (xAvg / 8);
            yAvg = (yAvg / 8);
            zAvg = (zAvg / 8);
        // --------------------
        // print the value only if we get the 'handshake'
        // --------------------
        if( Serial.available() > 0) {
             Serial.read();
            printVal(xAvg);
            printVal(yAvg);
            printVal(zAvg);
        }
        currentSample++; // increment the sample

    }

    //here's the tricky stuff: break the number into two
    // bytes so we can send it to oF without any problems
    void printVal(int val) {
         byte highByte = ((val >> 8) & 0xFF);
         byte lowByte = ((val >> 0) & 0xFF);

         Serial.print(  highByte, BYTE );
         Serial.print(  lowByte, BYTE );

    }
```

You'll notice that the value prints to the serial only if you've gotten something back from the openFrameworks application. Why is this? An Arduino controller and a C++ application run at slightly different speeds. If you were to send a message to the C++ application every time the controller ran its `loop()` method, you'd send a message while the application was still in the middle of processing the previous message. It's as if you were trying to write down what someone was saying as they were talking: it's quite helpful if they stop at the end of a sentence and wait for you to catch up. This is the same thing as when the openFrameworks application has finished processing a message from the Arduino board and sends a message back, in this case a single character, *x*, just to say: "OK, I'm ready for the next message." This ensures that the openFrameworks application hears the right values, and the Arduino board sends information only when the openFrameworks application is ready.

Let's look at the openFrameworks code in Example 8-5. It's short, and it's not very exciting, but it does show you something important: how to get complex data from the Arduino board into your openFrameworks application. What to do with that information once you get it is entirely up to you. You could use it to draw, you could use it to create a sound like an electronic theremin, or you could use it control the speed and distortion of a video; it's entirely up to you.

Example 8-5. oFArduino.h

```
#ifndef _OF_ARDUINO_COMM
#define _OF_ARDUINO_COMM

#include "ofMain.h"

#define NUM_BYTES 6

class OFArduino : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        int xVal;
        int yVal;
        int zVal;

        bool    bSendSerialMessage;    // a flag for whether to
 send                                          // our 'handshake'
        //data from serial, we will be reading 6 bytes, two bytes for each integer
        unsigned char bytesRead[NUM_BYTES];
        int countCycles; // this is how to keep track of our time
        ofSerial serial; // this is ofSerial object that enables all this
};

#endif
```

Note the `countCycles` variable. You're going to use that to make sure that you've gotten all information from the serial buffer before sending another handshake message to the Arduino board. Simply count to 5, and then get the next set of data. You'll see this in the `update()` method, shown here in Example 8-6.

Example 8-6. oFArduino.cpp

```
#include "OFArduino.h"


void OFArduino::setup(){

    countCycles = 0; // start our count at 0
    bSendSerialMessage = true; // send a message right away
    // serial.enumerateDevices(); // uncomment this line to see all your devices

    // set this to our COM port, the same one we use in the Arduino IDE
    // the second part is the baud rate of the controller
    serial.setup("/dev/tty.usbserial-A6004920", 19200);
}

void OFArduino::update(){

    if (bSendSerialMessage){
```

```
        // send a message to the Arduino controller telling it that we're
        // ready to get accelerometer data
        serial.writeByte('x');

        unsigned char bytesReturned[NUM_BYTES];
        memset(bytesReturned, 0, NUM_BYTES);

        // keep reading bytes, until there's none left to read
        while( serial.readBytes(bytesReturned, NUM_BYTES) > 0){
        };

        // make our integers from the individual bytes
        xVal = bytesReturned[0];
        xVal <<= 8;
        xVal += bytesReturned[1];

        yVal = bytesReturned[2];
        yVal <<= 8;
        yVal += bytesReturned[3];

        zVal = bytesReturned[4];
        zVal <<= 8;
        zVal += bytesReturned[5];

        printf("first  %i %i %i \n", xVal, yVal, zVal);
        // get ready to wait a few frames before asking again
        bSendSerialMessage = false;
    }

    countCycles++;

    if(countCycles == 5) {
        bSendSerialMessage = true;
        countCycles = 0;
    }
}
```

Accelerometers are very rich territory for interaction, particularly when dealing with mobile or handheld devices. Anything that a user can directly manipulate in a physical way becomes an excellent candidate for an accelerometer. Tilting, shaking, and positioning can all be used as input methods via an accelerometer. Coupled with an unwired transmission via the XBee wireless data controller, the accelerometer allows the user free motion around a space while still retaining their ability to send input.

Introducing I2C

With Inter-Integrated Circuit (I2C), you very quickly get into some complicated concepts. If you're interested, come along for the ride; if not, return to this section when you need it. When might you need it? Well, you need it when you need to communicate with an LCD display, a complex temperature sensor, a compass, or a touch sensor.

First, you should understand the concept of a clock in device communication. A device needs to keep track of its timing, in other words, how frequently it should be sending high and low signals out to any other devices connected to it. If a device has an internal clock, like the Arduino controller, then it can keep its own time. That's fine for the Arduino controller, because you expect that you're going to be sending it instructions and then having it run on its own. But what about something that you don't expect to run on its own? For example, what about a compass sensor? You expect that you're going to plug it into something to help it display information and know when to turn on and off, probably even providing power for it. It's a reasonable assumption that whatever the compass sensor is going to be plugged into will have a clock, so why can't you just use that clock instead of needing to have one in the compass? The answer is, you can. The catch, though, is that in order to use the clock of the controller, you need to be connected to that clock and get messages on it. This is what I2C is for—allowing multiple devices to be connected to a larger device in a network of sorts that will tell it when to update and read data from it. This network looks something like Figure 8-11.

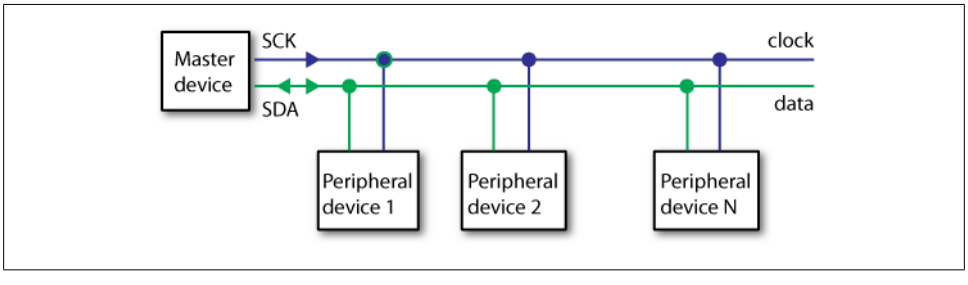


Figure 8-11. A controller using multiple devices over I2C

From the diagram in Figure 8-11, you'll notice that multiple devices are connected to the master device. This is one of the great powers of I2C. It allows you to communicate with up to *127 different devices*. They can't all communicate at the same time, because there's only one data line, but with a little clever code you can ensure that all the devices on the line know when they should be *talking*, or sending information, and when they should be *listening*, or reading the information on the data line. The clock ensures that synchronizing all these devices is fairly easy, because the master device simply tells all the slave devices when to talk and when to listen. The slave devices can be anything, even other Arduino controllers, which should give you not only a hint of how powerful I2C can be but also some bright ideas about distributing your application across multiple Arduino controllers.

I2C is built into the Arduino controller. Analog Pin 4 is the SDA, or *data pin*, and Analog Pin 5 is SCK, or *clock pin*. Plugging a device into the Arduino controller and creating an I2C circuit are both quite easy. I2C communication, however, is generally a little more difficult and would involve some rather intricate-looking code if not for the Wire library created by Nicolas Zambetti to work with I2C communication.

First, here's an example of how to set up I2C using the Wire library in an Arduino sketch:

```
#include <Wire.h>

void setup(){
    Wire.begin(); // this 'joins' or initializes the circuit
}
```

That's it. It's not so bad, right? Next, here's how to send a command to a slave device:

```
Wire.beginTransmission(20); // transmit to device #20
Wire.send("x is ");        // sends five bytes
Wire.send(x);              // sends value of x as one byte
Wire.endTransmission();    // stop transmitting
```

The `beginTransmission()` method indicates the device to which you want to send data. Now, this is very important to understand and admittedly counterintuitive. The number of the device doesn't actually have anything to do with the physical location of the device—it's the device identifier. We'll talk more about this later, but suffice to say that either the device will generally be set by the manufacturer of the device or you will have to set it manually. The `send()` method indicates what you're going to send to the device, and it's very important because devices have different commands in different formats and send and receive information in formats tailored to their function. This means that it's important to read the manual or data sheet for any device that you're trying to communicate with. To finish the transmission, you call the `endTransmission()` method.

So, that's the first step of using I2C with Arduino. The next step is to request information from a device. You need to know how much data you're expecting from the device that you're communicating with. For example, if you're expecting a single character, you need to know that and expect that from the device. Again, this is where the data sheet and documentation for any device will come in handy. You'll know just what data any device will send in response to any particular request and what that data represents.

In the following snippet, you expect that the device with the address of 2 will be sending 6 bytes of information, and you indicate that with the `requestFrom()` method:

```
Wire.requestFrom(2, 6);    // request 6 bytes from slave device #2
while(Wire.available()){    // slave may send less than requested
    char c = Wire.receive(); // receive a byte as character
    Serial.print(c);         // print the character
}
```

The `requestFrom()` method uses the following format:

```
Wire.requestFrom(whichDevice, howManyBytes);
```

The `requestFrom()` method uses two parameters:

`whichDevice`
: Is the identification number of the device from which you're requesting data.

`howManyBytes`
: Is the number of bytes that you're expecting the device to send. This is yet another reason to consult the data sheet and a fine justification, if one was needed, for spending a little time studying the number of bytes that different data types require.

The Wire library also defines methods for having your Arduino controller act as a slave to another master control (another Arduino or another kind of device), but we're not going to get into those at the moment. Instead, let's look at how to use the Arduino controller for a slightly different kind of LED light than you first looked at, using the BlinkM controller.

A great site and resource for information on electronics, hacking, and good ideas all around is Tod Kurt of ThingM. He and his partner have created a nifty little light called the BlinkM, shown in Figure 8-12, that lets you control the color of the light and set sequences of colors and blinking patterns.

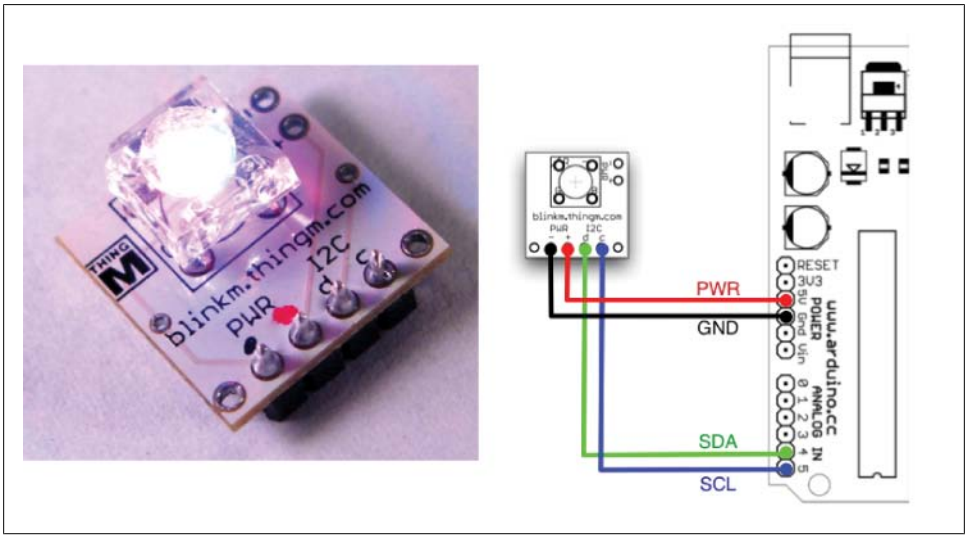


Figure 8-12. The BlinkM and connecting it to your Arduino controller

The following code is quite simple and liberally borrows from the BlinkM examples but does show a couple of really interesting things about I2C. The first thing to notice is that you address the BlinkM, which means that you give it an address that it will use for I2C communication. This means that you could connect multiple BlinkM controllers and send commands to each of them independently of the other. The address is a single byte, and you use it for all the BlinkM commands that you send. The Wire library is hidden within the BlinkM functions that are imported in the `#include` statement at the top of the file, which is nice, because it lets you concentrate on what's possible with the I2C library before delving too deeply into it. You can give a controller an address

and then use that address to control that particular component. To start, the following code sends a hexadecimal-based RGB color to the BlinkM via a serial command:

```
#include "Wire.h"
#include "BlinkM_funcs.h"
```

The address can be any value that you set BlinkM to. If you don't know how to set the BlinkM address, then you can leave it at the default value: `0x10`.

```
byte blinkm_addr = 0x10; // the address to set the BlinkM to
char serInStr[6];  // array to hold the HEX color the user inputs

void setup()
{
```

Here is where you initialize the BlinkM. The `BlinkM_stopScript()` method stops the BlinkM from running any script that has been uploaded to it previously:

```
    BlinkM_beginWithPower(); // start up the blinkM
    BlinkM_setAddress( blinkm_addr );  // set the address
    BlinkM_stopScript(blinkm_addr); // stop the BlinkM from running its script
    Serial.begin(9600);

}

void loop()
{
    int num;
    int i = 0;
    if(Serial.available()) {
         delay(10);
        while (Serial.available())
        {
          serInStr[i] = Serial.read();
          if(++i >= 6)
            break;
        }

void loop()
{
    if(Serial.available() >=6) {
        for(int i=0; i < 6; i++) {
            serInStr[i] = Serial.read();
        }
    }
```

For each RGB value in the color that you want the BlinkM to display, create a `byte` value that contains that value. In this example, the hexadecimal numbers sent over serial port from the Arduino console (or any other kind of application, for that matter) are used to create those bytes. Once you've created all the values, you can pass them to the `BlinkM_fadeToRGB()` method:

```
        byte a = toHex( serInStr[0],serInStr[1] );
        byte b = toHex( serInStr[2],serInStr[3] );
        byte c = toHex( serInStr[4],serInStr[5] );
```

```
        BlinkM_fadeToRGB( blinkm_addr, a,b,c);
    }
}
```

This method takes a hexadecimal number up to 255 and converts it into a byte value:

```
// copied directly from the BlinkM examples
#include <ctype.h>
uint8_t toHex(char hi, char lo)
{
    uint8_t b;
    hi = toupper(hi);
    if( isxdigit(hi) ) {
        if( hi > '9' ) hi -= 7;      // software offset for A-F
        hi -= 0x30;                  // subtract ASCII offset
        b = hi<<4;
        lo = toupper(lo);
        if( isxdigit(lo) ) {
            if( lo > '9' ) lo -= 7;  // software offset for A-F
            lo -= 0x30;              // subtract ASCII offset
            b = b + lo;
            return b;
        } // else error
    }  // else error
    return 0;
}
```

What would you add to have another BlinkM connected to your Arduino? If you answered "another address," you would be correct. All that is required to wire them up is to attach the second BlinkM to Analog Out Pins 4 and 5 of the Arduino controller to connect it to the I2C circuit. The rest is up to you to complete as a project.

What Is a Physical Interface?

Not all interactive applications require a physical interface for a user to send input into a system, but many do. The more controls that you provide to your user, the more important the organization used to present those controls becomes. The more complex the interaction between your application and the user, the more control the user will need over the application. In these situations, an interface that provides some sense of what can be input and how the system regards that input helps a user understand what can be done with an application and how to do it. You can use the following to help when creating an interface:

Appearance and labeling
: Any control can be labeled by a piece of text, by its color, or by its appearance. This can be quite simple or quite complex, whimsical or purely functional. The labeling of a control can be revealed, animated, variable, or based on the state of an application. Consider all the different elements that can contribute to the labeling of a control: color, typography, material, size, weight, and texture. Any or

all of these can be used in novel and interesting ways or in predictable ways that will reassure a user.

Task-based organization
: Controls that perform the same task should be organized by their appearance and by their location within the interface. The layout of many commonly used programs displays this characteristic; tool palettes, menus, mixing consoles, and computer keyboards all group their individual controls by the larger task.

Alignment organization
: Elements that are visually aligned with one another become related to one another when a user's eye tracks across the interface. This is particularly important in the *discovery period* of an application, which is the period when a user is trying to figure out what the system does and can do by the interface. Alignment can indicate which objects are related to one another, which objects are subordinate to another.

The organization of controls does two things: it aids the user in understanding the system, and it provides a personality for the system. The system must provide some feedback of what it's doing for the user to understand not only why the system has done what it already has done, but also what it's going to do in the future. You can script some things for a user, but you can't script all things. For anything that you can't script, you should provide meaningful allowances and meaningful descriptors. Chapter 11 examines physical feedback in much greater detail.

What's Next

Now that you have a grasp on how to use some basic and not so basic controls, you can begin to play with them and play with the interactions that people might have with them. Remember, even though you have to worry about all the technical details, your users and participants don't. They're concerned only with what they can touch or use and what they can get back from that.

A physical interface lets you work with the material qualities of your interface. While many of the visual effects that you associate with screen-based interfaces are very difficult if not impossible to replicate in a physical interface, the materiality is available to you. The texture, color, environmental context, tactile characteristics, and emotional resonance of materiality are all richly communicative elements that, even if they don't provide a way for you to get information from your users, provide invaluable ways to subtly communicate with your users. Industrial designers obsessively focus on the materials of their products for a good reason. Take a look at some of the classic works of industrial design by Raymond Loewy, or *Digital By Design* (Thames and Hudson) by Conny Freyer et al., the principals at Troika design studios in London. The boundaries of the physical interface are nearly limitless, and new technologies are constantly redefining the materials that you can use in an interface. Though many of the types of physical input that are covered in this chapter are rather basic, other components and controls are readily available on Sparkfun at sparkfun.com, on Newark at newarkelec

tronics.com, and from many other hobbyist electronics suppliers for creating physical interaction, not to mention some of the more exotic and experimental materials and technologies recently made available.

Just as important as the material and componentry chosen are the action and the actual physical interaction that a user will employ. Our everyday world is filled with objects for which you already have muscle memory, a cultural language, and an emotional attachment that can be leveraged and used by you, the interaction designer. Consider a piano. An innumerable number of associations and kinds of familiarity exist with the piano in Western culture. People will instinctively know how to use a piano, know what sort of data that their input is creating, and have a certain set of associations with the type of feedback they will receive. This is a powerful affordance for you to design an interaction around. The immediate problem is a technical one: how do you extract data from a piano? Analog information must be transformed into digital information in order to process and use the information in any kind of digital realm. In the case of the piano, either you can capture the sound with a microphone and analyze it in a piece of software, or you can use a MIDI device that digitizes the sound for you and then process that digital data. (The MIDI protocol and how to use MIDI devices will be covered in detail in Chapter 12.) Some of the techniques you learned in Chapter 7 will help you get started processing audio.

Despite the increasing digitalization of our lives, we still frequently interface through the world in a broadly physical fashion. The tools that you can use to input data into your application are all around you in everyday life. Some careful observation of how people interact with their physical environments will open up a whole new way to think about what you can do to make applications more intuitive, expressive, enjoyable, and meaningful.

Two excellent books on controls or physical computing are by Tom Igoe's and Dan O'Sullivan's: *Physical Computing* (Course Technology) and Igoe's *Making Things Talk* (Make Books). They're invaluable resources and inspirations to anyone interested in working with physical computing and physical interfaces.

The book *Practical Electronics for Inventors* by Paul Scherz (McGraw-Hil) is another wonderful reference book that will help you with the basic to intermediate concepts of working with electronics. It covers everything from integrated circuits, semiconductors, stepper motors and servos, and LCD displays to rectifiers, amplifiers, modulators, mixers, and voltage regulators.

Brendan Dawes wrote a book called *Analog In, Digital Out* (New Rider's Press) for thinking about physical interfaces and how to design them. They've already been mentioned in Chapter 1, but *Universal Principles of Design* by William Lidwell (Rockport) and *Design of Everyday Things* by Don Norman (Basic Books) are both particularly relevant to designing physical interfaces.

Review

Potentiometers, perhaps known to you as knobs or a dials, work by receiving a current and increasing or decreasing the amount of voltage on a pin as it's turned.

A piezoelectric sensor (piezo sensor for short) is a device that uses the piezoelectric effect to measure pressure, acceleration, strain, or force by converting these factors to an electrical signal.

The Serial port is a way of sending information from your computer to another machine that could be another computer but is more frequently another kind of device. Serial communication uses a protocol called RS-232.

To communicate by serial with a Processing application, use the `Serial` class and set it to use the port that the Arduino is connected to and baud rate that your Arduino is configured to use. For instance:

```
new Serial(this, Serial.list()[0], 9600);
```

To communicate by serial with an oF application, use the `ofSerial` class and set it to use the port that the Arduino is connected to and baud rate that your Arduino is configured to use. For instance:

```
serial.setup("/dev/tty.usbserial-A6004920", 19200);
```

To send messages from the Arduino to another application, write data to the Serial port using `Serial.print()`.

In a Processing application, you can read data sent over the Serial port by using the `Serial.readBytes()` method. You should make sure that data is available first by using the `Serial.available()` method:

```
while (arduinoPort.available() > 0) {
        arduinoPort.readBytes(inBuffer);
}
```

In an oF application the `ofSerial` class operates similarly, with a `readBytes()` method to read from the Arduino controller. It takes an additional parameter, which is the number of bytes you're expecting:

```
readBytes(unsigned char * buffer, int length);
```

When reading data from an infrared sensor, it is good to average readings by taking multiple readings, adding them together, and dividing by the number of total readings.

Binary numbers count using only the numbers 1 and 0 and counting up using places of 2. For instance, 1, 10, 11, 100, 101, 110, 111, 1000....

Bit shifting is moving the bits of a number up or down in binary. Shift left means to move the bits up; for instance, 5 shifted left one place becomes 10 because 5 is 101, and shifted left one place it becomes 1010 or, in decimal, 10.

Shift right means to move the bits down; for instance, 4 shifted right one place becomes 2 because 4 is 100, and shifted down you get 10 or, in decimal, 2.

I2C or Inter-Integrated Circuits can be used with the Arduino by attaching a device to Analog Pins 4 and 5. Pin 4 is the SDA, or data pin, and Analog Pin 5 is SCK, or clock pin. I2C is requently used to communicate with devices that requires timing signals from a microprocessor.

To make I2C communication easier, you can use the Wire library that comes bundled with the Arduino.

CHAPTER 9

Programming Graphics

There's an excellent reason for the hundreds of books out there on computer graphics, programmatic animation, and the mathematics and techniques that make it possible; they're vast topics. That said, as with so many other topics in this book, our goal is not to comprehensively cover this area, but to provide an introduction to some basic techniques, inform you of some more advanced techniques, and point you to places to go for more information so that you'll have a grounding when you encounter more advanced topics or technical challenges in your projects.

It's quite difficult to overstate the importance of providing graphical feedback and guidance for users. That said, it makes sense to describe exactly what this chapter is going to cover. In both Processing and oF, you've learned some of the basics of drawing using the API that each framework provides. Chapter 13 will cover using OpenGL, drawing in 3D, improving drawing performance, and using textures. So in this chapter, we'll focus on a few simple topics that will stitch the basics to the more advanced stuff: how to create animations, how to structure your code when you're creating an animation, and how to do some more sophisticated vector drawing. We'll also talk about how to create graphics that are useful to users, provide information, encourage exploration, and help the user easily understand what is going on behind the screen and how their interactions are driving those systems and processes.

We'll also explain how to use and create graphical elements that users can interact with, both drawing them and using libraries that have been created in Processing and in openFrameworks. There's great power in a familiar screen-based control. By working with screen-based controls and nonscreen-based controls in tandem, you can leverage not only the familiarity of a control, a button, a dial, or a slider, but you can link them to a physical reaction or create user input by using a control in a new way that reimagines them for the viewer.

The Screen and Graphics

A screen is a surface, and any graphics that appear on that screen define the space within that surface, whether it is deep or flat, representational or abstract, open or closed.

Roughly speaking, those images and graphics can be divided into a few different types that have quite different uses:

Diagrams

Diagrams are graphics that represent particular views into information or an instruction. Graphs, manuals, assembly instructions, and warnings are all examples of diagrams. The purpose of a diagram or visualization is insight: the view of a particular relationship between two or more represented elements. Accurate diagrams are not just conveniences wrapped in graphics. There is substantial evidence to show that visualizations and graphics improve cognition in many ways. Generally, the expectation of a diagram is that all of its signs are unambiguous; that is, the signs that it uses and data that it represents are clear. Diagrams tend to encourage interaction within their parameters. That is, a diagram that consists of a map of election statistics will lead viewers to want to be able to highlight data, call it out, select additional parameters, or alter existing ones.

A recent wealth of excellent projects use data visualization to create visually stunning graphics that simultaneously encourage playful and engaging interaction. Some excellent example projects include *NYTE* by SENSEable City Lab, *Similar Diversity* by Philipp Steinweber and Andreas Koller, any of the excellent projects by Santiago Ortiz and the Bestiario collective, *They Rule* by Josh On, and *State of the Union* by Brad Borevitz. These projects use engaging graphics and animations to create an interactive view into a particular dataset. There are also several wonderful books on creating diagrams and data visualizations. The two bibles in this field are *The Visual Display of Quantitative Information* and *Envisioning Information*, both by Edward R. Tufte (and both by Graphics Press). *Visualizing Data* by Ben Fry is an excellent primer not only on Processing but information graphics as a whole. *Information Graphics* by Robert L. Harris is also a helpful book for thinking about how to work with graphs and instructions in a more traditional way. On a more academic and scientific level, both *Information Visualization, Second Edition: Perception for Design* by Colin Ware and *Readings in Information Visualization: Using Vision to Think* by Stuart K. Card et al. (both published by Morgan Kaufmann) are dense but marvelous books filled with essays on vision, perceptual psychology, and how cognition and vision interrelate.

Scenes

Scenes are narratives that bring the user's eye around the image in a constructive act. This is not to say that the only things that draw the eye in a certain trajectory to create a temporal experience of a graphic or image are those that are narrative in a linear sense. There is, however, a difference between something that is ordered by how you shape the user's gaze than by more explicit signals such as numbering or boxes. Comic books frequently have wonderful examples of this, as do many classical paintings, motion graphics, and, of course, video games. Scenes urge exploration—the ability to zoom, explore, and change perspective. The graphics are viewed as representing a world and should provide functionality that allows

exploration. There is no better developed or more educational thing to do when thinking of devising a scene or world-based graphics than to play video games, particularly first-person games. The interaction demands of a user will be largely contextual to the needs of the world but will include, at the very least, a way to navigate in any direction and a way to increase or decrease speed.

Immersive graphics have been a trope of interactive graphics since the advent of computer graphics and animations. Everything from simulations to first-person shooters to architectural walk-throughs to novel interactive worlds like Zoltan Szegedy-Maszak's Promenade to Jeffery Shaw's Legible City all have used the idea of a navigable 2D or 3D space. There are several great books on designing worlds and narrative graphics systems. *Chris Crawford on Interactive Storytelling* (New Riders) is a wonderful primer to the possibilities of creating and fleshing out a narrative within a world for a user. *The 3D Math Primer for Graphics and Game Development* by Fletcher Dunn is a valuable reference for anyone looking to seriously develop an interactive world.

Algorithmic drawings

Algorithmic drawings are fundamentally meant to express the output of a system to a viewer. They also encourage interaction in the same way that a diagram does, changing the underlying system through some interaction to alter the output of the system. So, what kinds of interaction does a user require for an algorithmic system? If you're looking to have user interaction drive the algorithms, then you'll want to include controls to manipulate variables, alter patterns, and explore the visualization and easy and appropriate control types for each variable. An analog variable requires an analog type control such as a slider or a dial, a complex multimodal variable requires a complex input like an accelerometer or 3D control, and a binary variable requires an on or off switch.

It is quite difficult to discuss algorithmic art without mentioning Casey Reas right away, since he is not only one of the most forward-thinking visual artists working with software today but also has created one of the most popular tools for generating algorithmic drawings: Processing. It is very difficult to talk about algorithmic art without mentioning the artist Sol Lewitt as well. Neither of these artists creates interactive works per se, but they are touchstones for anyone interested in the algorithmic art as something other than a screen saver. In the 1970s, people like Myron Kreuger, Ben Laposky, and Stan Vanderbeek created works that were the front-runners of algorithmic art. In the last few years, artist/programmers like Thomas Straum, Robert Hodgin, Erik Natzke, Karsten Schmidt, Paul Prudence, and Marius Watz have shaped much of the ideas that people have about what an algorithmic piece of artwork looks like, particularly a piece made with Processing. The book *Processing* written by Casey Reas et al. (MIT Press) is a wonderful place to start thinking about algorithmic pieces. John Madea's *Creative Code* (Thames and Hudson), likewise, introduces the reader to some of the most important ideas and thinkers. A much more advanced text, worth its trouble, though, is *The Art of Artificial Evolution* by Juan Romero and Penousal Machado (Springer). Finally,

browsing the Processing website, at *http://processing.org*, will introduce you to several relevant artworks and artists.

Drawing tools

Drawing tools can utilize user interaction to create the graphics, either as a part of a larger algorithm or as the sole provider of information. User input can be taken as data and thrust in a larger system—physics for example, as in the whimsical case of the classic Flash game Linerider—or the novelty of their manner of input, like many of Camille Utterbach's pieces. It's important to give a user fine-grained control over their drawing. The more interesting the tool, the less the user will focus on the drawing itself; the less interesting the tool, the more the user will focus on the drawing.

The interactive possibilities for a drawing tool are very rich indeed. By looking at a piece like Amit Pitaru's Sonic Wire Sculputor, you can see how to make even a tool that draws only simple lines fascinating by interfacing it with another system. A drawing system very often needs to interface with something else, such as a printer, an orchestra, a camera, or some machine that alters the physical environment in some fashion. It is important when combining systems, however, to ensure that the user is given a complete view into the system with which they are interacting.

Although this isn't in any way an exhaustive look at the tropes in creating interactive graphics, it gives you a sense of some of the various ways of presenting interactive graphics, rather than just images for people to view.

Seeing Is Thinking, Looking Is Reading

It's important to realize that a graphic or a visual signal to a user is not an isolated object. Any piece of feedback given to a user is contextualized by the grammar of your application and the understanding that the user brings to the application. Any interactive application is ultimately a conversation with a user, and any symbol within that application is modified by the grammar that is used to organize any signs. That word *grammar* is a slightly odd one. Think of the relationship that a subject has to the object of a sentence and the relation that the verb of a sentence has to the subject and object: "John throws the ball." Now imagine how we describe the actions of users from the perspective of users. When they see a control, a shape, an instruction, or an image, they are seeing the image as it is, but they are also seeing how it exists within the context of the application up to that point, how it relates to what they expect to be seeing, and how it relates to what they think they can do with it as information. So, design is the conscious effort to impose order, but the use of or interaction with a design is the effort to discover how to use that order.

Donald Hoffman makes an excellent case for the rules of visual thought in his very readable and very entertaining book *Visual Intelligence* (W. W. Norton) that's worth looking at for anyone interested in visual communication. Thinking is the attempt by

the brain to make a rational explanation for what it encounters. One part of your job in designing an interaction and an interface is to help guide the brain of the user toward the appropriate rational description of the object that they see. This is where patterns are important. When hearing a rapid, verbal conversation in a language that you understand, it is quite easy to fill in the blanks to make sense of what the speakers are saying. The same goes for the workings of a system and the interaction that a user has with it. If they understand the visual grammar of an interface, then they will find it much easier to understand the feedback given by the system and easier to create meaningful input.

Ask anyone to multiply 57 by 91 in their head. Then give her a piece of paper and ask her to multiply two other two-digit numbers. Almost everyone will be able to perform this task more quickly on paper. This is not because there is any inherent value in the act of writing, but because being able to refer to the numbers, line them up visually, and break the calculations into smaller calculations is far easier when you can refer to the material that you are working with. This goes for almost all tasks: having a visual reference makes tasks easier. The easier a visual reference is to identify, organize, and manipulate, the more it will aid thinking. Creating distinct, good icons; having an easily legible color scheme; and making objects easy to manipulate are aids that help the user think. These things may seem a little bit like common sense, but it's important to contextualize them correctly. An interaction does not require visual affordances just because they are pleasant; an interaction requires visual affordances because they let a user employ vision to think.

Math, Graphics, and Coordinate Systems

In any computer drawing that you do, you'll be using a coordinate system. This may sound complex, but it is actually quite simple. A coordinate system is just a way of determining in any given mathematical measurement, pixels or feet or miles, where things are and how big they are. The easiest coordinate system to imagine is the one used whenever you draw a rectangle. For example, in Processing, you draw a rectangle using the following syntax:

```
rect(50, 50, 50, 50);
```

This syntax creates a rectangle that starts at 50 pixels in the x position and 50 pixels in the y position that is 50 pixels high and 50 pixels wide, as shown in Figure 9-1.

If you recall Cartesian coordinates from your high-school algebra class, you'll remember that the y position in a Cartesian system decreases as you go further down the y-axis. In screen coordinates, this is inverted; the y values go up as you look further down the y-axis. (Figure 9-2 shows a comparison of the two coordinate systems.) This creates some difficulties when figuring out where something is supposed to go on a screen, because any mathematical calculations assume that the coordinate system has y increasing as you look further up the y-axis.

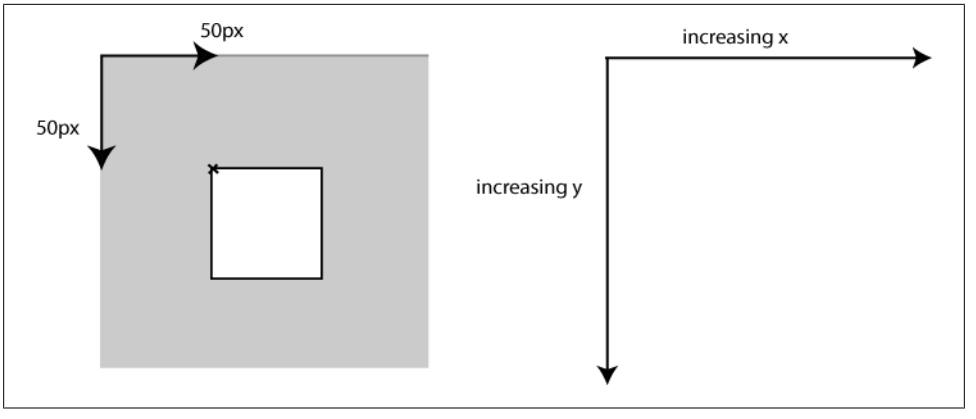


Figure 9-1. Positioning a rectangle in screen space

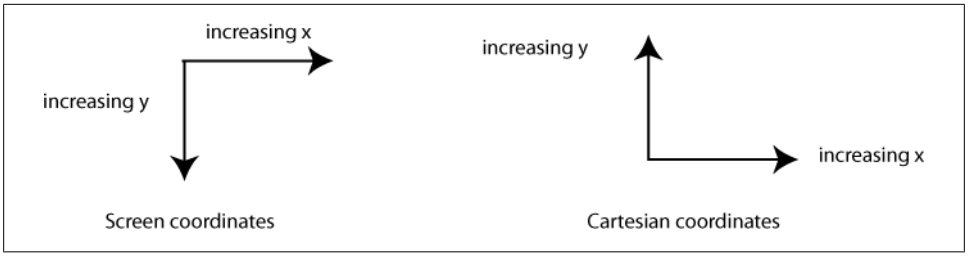


Figure 9-2. Screen coordinate and Cartesian coordinate systems

Most of the mathematics involved in determining the distance and direction of an object stay the same, but the meaning of the values change from one system to another because a negative value in a Cartesian system can be a positive value in a screen system, and vice versa. Luckily, for the most part, though, the frameworks that you'll use to draw—Processing and openFrameworks—have convenience methods that let you avoid converting from one system to another. The coordinate systems are a good thing to understand, though, when you're creating drawing commands for your computer.

Another thing you'll almost certainly remember from your geometry class is the notion of a point, a location in two or three-dimensional space. On a screen, the point is a pair of x and y coordinates; in three-dimensional space, those coordinates are joined by a z coordinate. One thing that you'll frequently find yourself doing is calculating the distance between points. This is easily done by using the Pythagorean theorem: distance = square root of $x^2 + y^2$. You can express this simply by using one of the built-in square root methods of Arduino, C++, or Processing. The following code snippet is in C++, though the Processing version is similarly straightforward:

```
float dist(int startX, int startY, int endX, int endY)
{
    return sqrt((endX-startX)*(endX-startX) + (endY-startY)*(endY-startY));
}
```

In addition to drawing lines, rectangles, triangles, ellipses, and other primitive graphic types at some point, you'll probably want to rotate objects that you've drawn. One oddity of working with graphics in computing is that rotate operations are almost always calculated and reported using radians instead of degrees. As you may recall from your geometry classes, a circle is 2π radians around, which makes 360 degrees equal to 2π, 90 degrees equal to π/2, and 180 degrees equal to π. Now, since π is an irrational and infinite number, most programs use a constant value that stands in for an approximate value of π, or they use convenience methods to allow you to continue thinking and calculating in degrees and simply convert those values to radians when you're finished with your calculations to rotate your objects or drawings easily.

Given that both Processing and openFrameworks are designed for users without a great deal of mathematical experience, a few of the most commonly used operations have convenient methods. Processing provides, among others, the following:

`lerp(value1, value2, amt)`
: Calculates a number between two numbers at a specific increment. `lerp` is actually short for linear interpretation, which means translating a value to a given range. For example, you might want to get a number at 60 percent of a particular range, say, 0 to 255:

```
lerp(0, 255, 0.6); // returns 153.0
```

`dist(x1, y1, x2, y2)` and `dist(x1, y1, z1, x2, y2, z2)`
: Calculate the distance between 2D and 3D points, respectively.

`map(value, low1, high1, low2, high2)`
: Remaps a number from one range to another. You might need to take a number from one range, 0 to 255 for example, and put it into a different range, 0 to 10:

```
print(map(100, 0, 255, 0, 10));//prints 3.9215686
```

`atan2(y, x)`
: Calculates the angle (in radians) between the two points.

These are just a few of the many methods both provided in the Processing language and expanded by user-contributed libraries. You'll notice that these methods revolve mostly around calculating values in one range and then converting that to another range. This is because conversions are a very important part of working with graphics: converting not only between coordinate systems but between ranges set by your UI elements and data, locations within one visual element to locations within another visual element, and much more. The other vital group of methods revolves around trigonometry. At the heart of graphics programming and graphical processes is a lot of good old-fashioned high school trig. Some of the methods that openFrameworks provides for working with vectors will be examined later in this chapter. The core C++ libraries provide many of the basic methods that you'll need for trigonometric calculations, such as `sin()` to calculate the sine of a number in radians, `cos()` to calculate the cosine, and so on.

All of this may sound quite dull, and in fact, it's quite often mathematically complex to put together a drawing. However, one of the beautiful aspects of algorithmic drawing is that once you've struck upon something interesting to you, usually with very small tweaks to the equations that you're using you can generate drawings that vary immensely.

Two excellent books on mathematics for graphics are *Mathematics and Physics for Programmers* by Danny Kodicek (Charles River Media), which has more of a general focus, and *3D Math Primer for Graphics and Game Development* by Fletcher Dunn (Jones and Bartlett), which is specific to 3D graphics. Two other Processing books that have a lot of great information about mathematics and drawing are Ira Greenberg's *Processing: Creating Coding and Computational Art* (Springer) and Daniel Shiffman's *The Nature of Code* (*http://www.shiffman.net/teaching/nature/*). Though all the code in these books is in Processing, it can very easily be converted to C++ for use in an oF application.

Drawing Strategies

Anyone can see what a house or a dog or a face looks like. They can perceive it in very great detail when they attend to each element of it. That perception, though, doesn't translate to the ability to draw that face until that perception is accompanied by a knowledge of how to decompose the object into requisite pieces and assess them not only in terms of how it appears but also how it must be drawn. Anyone who has spent time trying to draw themselves—or, for that matter, anything in the world—can attest that vision and construction are different. So, it is with creating graphics in computing. Graphics that you see are composed of thousands of triangles, squares, simple lines and fills between them, or pixel-by-pixel constructions. This is important to understand in the same way that understanding that seeing something and being able to draw it are different. Depending on your intended outcome, you can employ several different strategies. Let's look at a few.

Use Loops to Draw

The `for` and `while` loops are useful drawing tools, because they let you consistently repeat a pattern, an idea, to vary what you draw as your loop unfolds or to vary the number of drawings with response to some input or to provide a certain feedback. Let's look at drawing a repeating pattern, an ellipse, and varying it with each loop with a simple example in Processing:

```
void setup() {
    size(800, 800);
}

void draw() {
    background(0, 0, 0);
    for(int i = 0; i < 50; i++)
```

```
        {
          // draw 50 circles
          // in the center of the screen with the circles
          // progressively getting closer and closer to the mouse
          ellipse(400 - (i*10 - (0.02*i*mouseX)),
            400 - (i*10 - (0.02*i*mouseY)),
            i*2,
            i*2); // we're just increasing the size of the circles here
          fill(255, 255, 255, 20);
        }
    }
```

So, here the ellipses are progressively drawn closer and closer to the mouse position. Notice how the code just increases the integer i and uses that value each time the program passes through the `for` loop. This draws circles that gradually increase in size and modifies the value that is used to place those circles incrementally. Since the `draw()` method itself loops over and over again, the drawing is redrawn every frame, and the reaction to the user's mouse is re-created.

The `MouseResponseColor` class has the `getDistance()` method that calculates the distance from one point to another. Let's look at another example of using the mouse position in drawing, this time using oF, in Example 9-1.

Example 9-1. MouseResponseColor.h

```
#ifndef _MOUSE_COLORATION
#define _MOUSE_COLORATION
#include "ofMain.h"

class MouseResponseColor : public ofBaseApp{

public:

    void setup();
    void draw();

    float getDistance(int startX, int startY, int endX, int endY);

    float max_distance;
};

#endif
```

You can see how this is used in the *.cpp* file for this application shown in Example 9-2.

Example 9-2. MouseResponseColor.cpp

```
#include "MouseResponseColor.h"

void MouseResponseColor::setup(){

    ofSetWindowShape(700,700);
    max_distance = ofDist(0, 0, 700, 700);
    ofEnableSmoothing();
```

```
    ofEnableAlphaBlending();
}

void MouseResponseColor::draw(){
    int i, j;
    int height = ofGetHeight();
    int width = ofGetWidth();

    for(i = 0; i <= height; i += 20) {
        for(j = 0; j <= width; j += 20) {
            float dist_color = getDistance(mouseX, mouseY, i, j);
            dist_color = dist_color/max_distance * 100;
```

To get the colors into the range between 0 and 255, multiply the values by 5:

```
                ofSetColor(dist_color*5,dist_color*5,dist_color*5, 123);
                ofEllipse(i, j, 20, 20);
            }
        }
    }

    float MouseResponseColor::getDistance(int startX, int startY, int endX,      int endY)
    {
        return sqrt((endX-startX)*(endX-startX) + (endY-startY)*(endY-startY));
    }
```

We moved quickly into following the mouse because it gets us to what we want to talk about, which is using the graphics as a feedback loop. Average users will figure out in several milliseconds that the mouse position is determining the drawing. They'll figure out in several seconds how to control the drawing with their mouse. Unless your drawing changes, gives them greater control, or does something fantastically novel, a user can get bored rather quickly. That said, the techniques of drawing for a mouse-driven drawing, laser pointer-driven drawing, or bodily driven drawing are not that different.

Use Arrays to Draw

The drawings shown in the previous examples are vector-based, which means that they consist of a series of points that are connected by lines and filled in by a color. If you want to store a particular drawing, you could try to store the data about the drawing that you've just made, but this can be a lot of information, saving and manipulating each pixel for each drawing. Instead, you can simply use an array to store the data that was used to create the drawing, its points, and the color used to fill it. This a common strategy because it lets you store multiple drawings in a single place, access them, and manipulate them easily each time you draw the screen.

In the following oF application (Examples 9-3 and 9-4), there are 20 circles that are redrawn in each frame. However, those circles maintain a somewhat steady position because their positions are each drawn from two different arrays, one storing x positions and another storing y positions. This allows you to create the illusion of consistent objects, even though they are redrawn in each frame. This is, in fact, the basis of all

computer animation: storing data points to represent the locations of objects, updating them each frame, and then redrawing them all.

Example 9-3. MouseCircleFollow.h

```
#ifndef _MOUSE_C_FOLLOW
#define _MOUSE_C_FOLLOW

#include "ofMain.h"

#define numOfCircles 20

class MouseCircleFollow : public ofBaseApp{

    public:

        void setup();
        void draw();
        void update();

        float xPos[numOfCircles];
        float yPos[numOfCircles];

        float dist(int startX, int startY, int endX, int endY);
        float createRandomizedNewPos(float in, float mouse);
        float max_distance;
};

#endif
```

Example 9-4. MouseCircleFollow.cpp

```
#include "MouseCircleFollow.h"

void MouseCircleFollow::setup(){

    ofSetWindowShape(700,700);
    max_distance = dist(0, 0, 700, 700);
    ofEnableSmoothing();
    ofEnableAlphaBlending();
    int i;
```

Each circle should start at a random location to give the application some initial movement before the circles begin to converge on the mouse location:

```
        for(i = 0; i<numOfCircles; i++) {
            xPos[i] = ofRandomf() * 700;
            yPos[i] = ofRandomf() * 700;
        }

    }

    void MouseCircleFollow::update(){
        int i, j;
```

```
        float modifier;
        for(i = 0; i<numOfCircles; i++)
        {
            if(mouseX > xPos[i]) {
                xPos[i] += createRandomizedNewPos(xPos[i], mouseX);
            } else {
                xPos[i] -= createRandomizedNewPos(xPos[i], mouseX);
            }

            if(mouseY > yPos[i]) {
                yPos[i] += createRandomizedNewPos(yPos[i], mouseY);
            } else {
                yPos[i] -= createRandomizedNewPos(yPos[i], mouseY);
            }
        }
    }

    void MouseCircleFollow::draw(){
        int i;
        ofSetColor(255,255,255, 100);
        for(i = 0; i<numOfCircles; i++) {
            ofEllipse(xPos[i],yPos[i],20,20);
        }
    }
```

Here, a new position for the circle is created that will more or less follow the mouse, but not too precisely to create the illusion of somewhat randomized movement. It's purely an aesthetic decision and isn't necessary for this code to run correctly:

```
    float MouseCircleFollow::createRandomizedNewPos(float in, float mouse)
    {
        return ofRandomf()/2 * tanhl(sqrt((mouse - in)*(mouse - in)))+0.05;
    }

    float MouseCircleFollow::dist(int startX, int startY, int endX, int endY)
    {
        return sqrt((endX-startX)*(endX-startX) + (endY-startY)*(endY-startY));
    }
```

You can extend the use of arrays with a little additional mathematics to track not only the position of the x and y positions but also the angle of an object. Figure 9-3 shows the results of the MouseFollow code.

There are a few discrete sections to this code (Example 9-5): determining the location of each block relative to the mouse position, updating the arrays that store the positions of each block, and finally, drawing each rectangle.

Example 9-5. MouseFollow.h

```
#ifndef _MOUSE_FOLLOW
#define _MOUSE_FOLLOW


#include "ofMain.h"
```

```
#define numSegments 200
#define segLength 10

class MouseFollow : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();
```

Figure 9-3. The preceding application running

Here are the three methods that perform each of the discrete steps mentioned earlier:

```
            void reachSegment(int i, float xin, float yin);
            void positionSegment(int a, int b);
            void drawSegment(float x, float y, float a, float sw);

            float xPos[numSegments];
            float yPos[numSegments];
            float angle[numSegments];
            float targetX, targetY;
            float randX, randY;
    };

    #endif
```

Example 9-6. MouseFollow.cpp

```
#include "MouseFollow.h"

void MouseFollow::setup()
```

```
{
    ofSetWindowShape(1000, 1000);
    ofEnableSmoothing();
    ofEnableAlphaBlending();
    ofSetColor(255, 255, 255, 123);
}
```

The `update()` method uses two loops to determine the position of each segment in the array by calculating a corkscrew pattern between the origin of the mouse position and the new position. The "corkscrew" is calculated in the `positionSegment()` method:

```
void MouseFollow::update() {
    reachSegment(0, mouseX-500, mouseY-500);
    int i;
    for(i = 1; i<numSegments; i++) {
        reachSegment(i, targetX, targetY);
    }
    for(i = numSegments-1; i>=1; i--) {
        positionSegment(i, i-1);
    }
}
```

Now that the positions for each segment have been calculated, they can be drawn to the screen:

```
void MouseFollow::draw()
{
    for(i=0; i<numSegments; i++) {
        drawSegment(xPos[i], yPos[i], angle[i], (i+1)*2);
    }
}

void MouseFollow::positionSegment(int a, int b) {
    xPos[b] = xPos[a] + cos(angle[a]) * segLength;
    yPos[b] = yPos[a] + sin(angle[a]) * segLength;
}
```

The `reachSegment()` method determines where the next segment should be placed so that the segments, as a total, start at the origin and reach to the mouse position:

```
void MouseFollow::reachSegment(int i, float xin, float yin) {
    float dx = xin - xPos[i];
    float dy = yin - yPos[i];
    angle[i] = atan2(dy, dx);
    targetX = xin - cos(angle[i]) * segLength;
    targetY = yin - sin(angle[i]) * segLength;
}
```

Now draw the actual segment and rotate it, if it needs to be rotated at all:

```
void MouseFollow::drawSegment(float x, float y, float a, float sw) {

    glPushMatrix();
        glTranslatef(x,y,0);
        glRotatef(a, 0, 0, 0);
        ofRect(500,500, sw, segLength);
```

```
        glPopMatrix();

    }
```

Draw Only What You Need

One of the most important rules of making efficient graphics code is to draw only what you need. It's important to do only the drawing that is necessary in each frame. Each time your application draws the processor of the computer and the memory that your graphics require increase. One strategy is to set Boolean values to `true` or `false` that indicate whether a particular drawing operation needs to be done:

```
bool updateLines;
bool updateBackground;
```

You can then check the variables in the `draw()` method of the application:

```
app::draw() {
    if(updateLines) {
        // ... do some drawing for the lines
        // and set the value to false, since you've just updated the lines
    }
    if(updateBackground) {
        // ... redraw the background
        // and set the value to false, since you've just updated the background
    }
}
```

Use Sprites

Another strategy is to break the drawing into requisite parts. For instance, the background of a graphic will not need to be redrawn as often as the foreground, which is the object of visual interest. This means that it makes sense to define your visual objects, or *sprites* as they are sometimes called, as separate classes. For instance, a shape that has a variable color and a label should be broken into a class with a `draw()` method on it that will set the color appropriately and set the text on the label. In your main application, you simply call `draw()` on each sprite instance, and they handle creating their own colors and graphics.

Processing and Transformation Matrices

Imagine for a moment that the window of a Processing application is a piece of paper and you are seated at a desk in front of this piece of paper with a pencil in your hand. Your hand is sitting at the 0, 0 point of the paper, the upper-left corner. If you want to draw something in the lower-right corner of that piece of paper, you can move your hand down to the lower right of the page, or you can push the page so that the lower-right corner is beneath where your hand already sits. Take that thought and apply it to

Processing: drawing a circle in the lower right of a 300 × 300 pixel window would look like this:

```
ellipse(270, 270, 30, 30);
```

The ellipse is drawn 270 pixels down and 270 pixels to the right of the window. Now take a look at the following bit of code and think of moving the piece of paper:

```
ellipse(270, 270, 30, 30);
translate(-30, -30);
ellipse(270, 270, 30, 30);
```

This bit of code will create the drawing shown in Figure 9-4 on the left by moving the coordinate system of the Processing application up and to the left by 30 pixels.

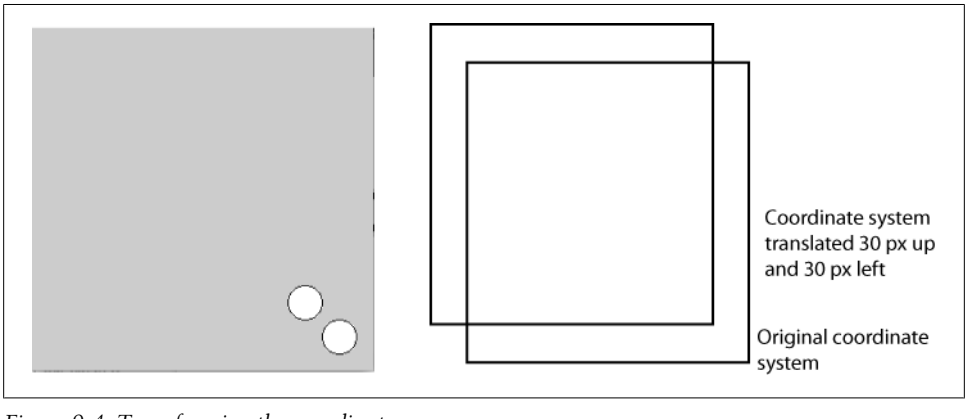


Figure 9-4. Transforming the coordinate space

One easy way of thinking of the `translate()` method is to imagine that it moves the upper-left corner of the drawing space. Move the drawing space down 20 pixels, and all drawings will appear 20 pixels lower on the screen. The proper way of thinking of the `translate()` method is that it modifies the coordinate space of the application; that is, it moves the position of the 0, 0 point in the application, what you might know as the origin of the coordinate system.

You can call the `translate()` method in two ways: the first is for two-dimensional space, and the second is for three-dimensional space:

```
translate(int x, int y);
translate(int x, int y, int z);
```

or:

```
translate(float  x, float y);
translate(float x, float y, float z);
```

This simplifies your drawing greatly by allowing a single algorithm, for example one that creates a particular shape, to be reused without modifying any of its coordinates. In algorithmic drawing, this is quite important because a drawing composed of several

dozen or more shapes is painstaking work to put together correctly. Using `translate()` allows you to use the same piece of code again and again, moving the coordinate system of the application to place each instance of a drawing.

Returning to the drawing metaphor that we began with, imagine now that each piece of paper can be moved around for a moment and then put back to the original position. This is done frequently when drawing—moving a page here and there to draw more easily. Processing uses a similar concept called a *matrix stack* to let you make changes to the coordinate system of your application and then undo or modify the changes. Look at the matrix stack in Figure 9-5.

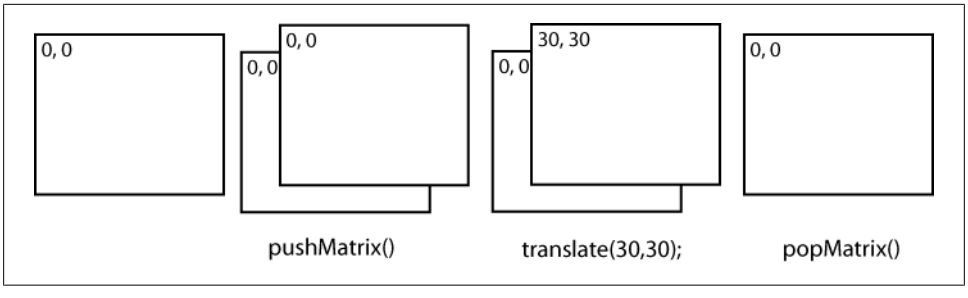


Figure 9-5. The coordinate matrix

Initially, there is only one transformation in the matrix stack, the original coordinate system. When a call is made to `pushMatrix()`, a new coordinate system is added to the stack. All drawing goes on in that new coordinate system, and any changes made are made to that system. Next, in Figure 9-5, a translation is made to the coordinate system using the `translate()` method to move it 30 pixels to the right and 30 pixels down. This affects only the current coordinate system in the matrix stack. Finally, `popMatrix()` is called, and the translated matrix is removed from the matrix stack, meaning that any changes made to the old coordinate system will not be used in new drawings.

To recap the methods in the matrix stack:

`pushMatrix()`
: Saves the current coordinate system to the matrix stack, making it available for use.

`popMatrix()`
: Removes the current coordinate system from the matrix stack, removing all of its transformations from any future drawings.

Once again, when do you want to use a matrix stack? You want to use a matrix stack when animating is a great time consumer, because modifying and switching out coordinate systems is much easier than doing the math and redrawing each shape when rotating or scaling shapes. The following code creates a series of four matrices. Each new matrix saves the previous one and then uses its coordinates as the origin, meaning that changes are cumulative:

```
float rotateAmt = 0.0;

void setup() {
    size(700, 700);
}

void draw() {
    background(122);
```

The first matrix will be translated over 100 pixels and down 100 pixels:

```
    pushMatrix();
        translate(100, 100);
        ellipse(30, 30, 300, 300);
        rect(250, 250, 200, 200);
        pushMatrix();
```

The second matrix will take the position of the first matrix, move it 100 pixels further over and use the `mouseX` to set the y position of the matrix:

```
          fill(255, 100);
          translate(100, mouseX - 100);
          ellipse(0, 0, 300, 300);
          rect(100, 100, 200, 200);
          pushMatrix();
              fill(255, 100);
```

The third matrix will take the position of the second matrix and further modify it by using the `mouseY` to set the x position of the matrix:

```
              translate(mouseY - 100, 100);
              ellipse(0, 0, 300, 300);
              rect(100, 100, 200, 200);
              pushMatrix();
```

The final matrix adds rotation and then draws three more rectangles into the new matrix:

```
                rotate(PI*rotateAmt);
                translate(200, 200);
                rect(0, 0, 50, 50);
                rect(-200, -200, 50, 50);
                rect(-100, -100, 50, 50);
```

Now all of the matrices need to be cleared by calling `popMatrix()` for each of them:

```
              popMatrix();
          popMatrix();
      popMatrix();
    popMatrix();

// modify the rotate variable
    rotateAmt += 0.01;

}
```

Take a moment to run this code, and notice how the three coordinate systems build off the changes to the first. Looking at the code, you'll notice a new method, `rotate()`. This is in fact the reason for the discussion of radians earlier in this chapter, since the `rotate()` method takes a floating-point value for the amount that the coordinate system should be rotated in radians, not degrees.

The `rotate(float value)` method, used in Figure 9-6, rotates the drawing coordinates by the specified amount.

For convenience sake, there is also a `radians()` method that takes degrees and returns radians.

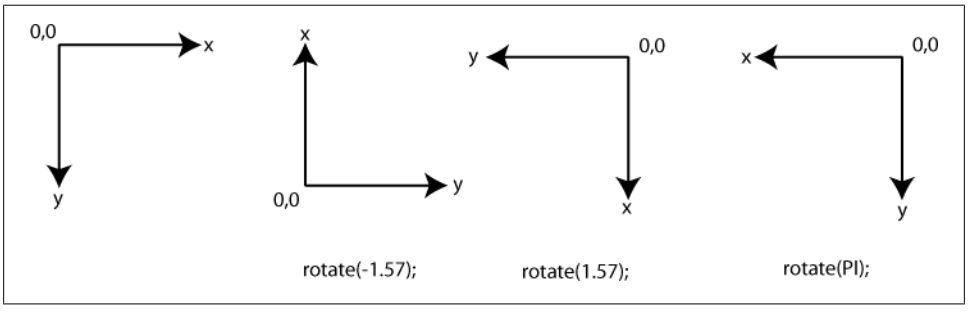


Figure 9-6. The rotate() method of the Processing language

The same technique can be used in openFrameworks, but it relies on OpenGL, so it will be covered in Chapter 13.

Creating Motion

Most motion is perceptual. In other words, we know what looks right and what looks wrong. There are lots of ways to quantify data around animation and animated motions, but animation is really a matter of looking right to the viewer. There are two fundamental types of animation: frame-based animation and cast-based (sprite) animation. You can think of these as ways to organize your drawing. Either you can draw everything at once, or you can organize your code to make sprites or graphical objects that have their own drawing routines. You'll find that many times the ideal way to create your animations and graphics is to combine frame animation and sprite animation. Consider a simple maze game like Pac-Man. In Pac-Man, the main character moves around the game screen eating dots. To convey the effect of the main character eating, his mouth opens and closes in a loop of frames played over and over again without change as he moves around. The simple movement of the character around the maze is done by redrawing the sprite in a new location, but the change in his

appearance brought on by his mouth moving is done by playing a loop of predrawn images over and over again.

Lots of animation is done around equations. The equations determine the speed of motion of objects. There are three basic kinds of motion: uniform motion, accelerated motion, and chaotic motion. *Uniform motion* is motion that does not change—it's the same direction and same speed all the time. The following snippet uses uniform motion:

```
int circleXPos = 1;
void draw() {
    circleXPos += 3;
    ellipse(circleXPos, 10, 10, 10); // never changes
}
```

Uniform motion is pretty well described by points, either in two or three dimensions.

Accelerated motion is what you encounter in the world: mass, gravity, wind, forces pushing and pulling on all objects. Accelerated motion is made a lot easier by using *vectors*, a mathematical entity like a point, but with a direction and velocity in addition to a location.

Finally, *chaotic motion* is motion that you don't often encounter in the world. It doesn't mean something that is necessarily purely chaotic, just that it is affected by some degree of randomness. Adding a call to Processing's `random()` method or oF's `ofRandom()` and adding it into a perfectly stable accelerated motion or uniform motion equation will do the trick.

Shaping the Gaze

When you look at how most interfaces allow the user to navigate through them, you'll find the same metaphors of static screens that display a certain view and that can be manipulated by the user. There is a reason for this: the injudicious use of motion can be very disruptive. For interactivity, you want the animation and motion that you employ to help people understand what the application is doing, what the user can do next, what they should pay attention to, and how their input is affecting the application. To drive participation, you must provide users with a sense not only that they are driving an animation or a motion but also *how* they are doing it. Animations and movements create anticipation (or focal points of attention), provide feedback on an action, or create relations.

Setting the Mood

Motion, like color, has a tone or a sense associated with it. A smooth, gradual motion feels orderly. A jittery, fast motion can be something that demands attention or something that has gone wrong. A twisting, meandering motion can be something idling. A rapid turning or twisting can be something dynamic. A rapid camera shift implies a

sudden event. A good motion artist or animator can use all these types of movement to give their artwork life, tone, and texture, or drive a narrative.

Two of the simplest generalizations of motion are those that appear mechanical and those that appear organic. Mechanical motions are geometric, efficient, repetitive and frequently less complex to compute. A piston turning, or the hand of a clock turning, is a crisp, clean, orderly movement that speaks to the engineering that has gone on beneath these surfaces or around these objects. Organic motions are smoothly nonlinear, are idiosyncratic, and vary greatly in response to their environments, such as a leaf falling as the wind blows, a person dancing, water rippling, or smoke rising in a still room. These two types of motion evoke much different emotional and cognitive responses from people, just as different colors do.

It's often difficult at first to think about how to represent or break down motion. You can refer to diagrams from physics classes to show linear, quintic, or quadratic motion, as in Figure 9-7.

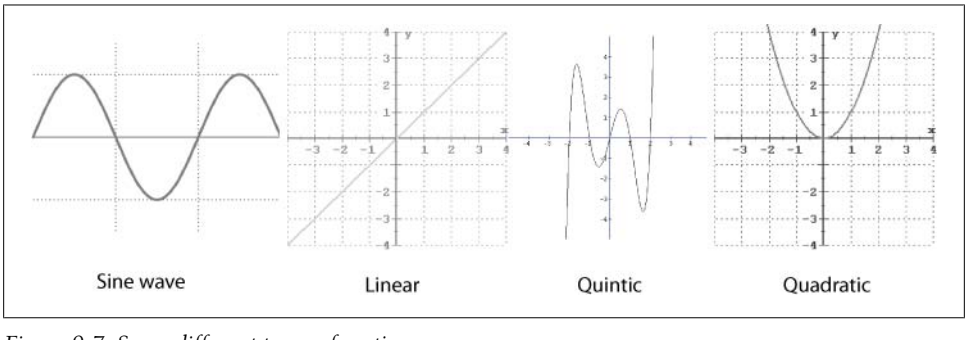


Figure 9-7. Some different types of motion

The most straightforward way to create this kind of motion is to use the equation to draw. The following snippet uses the Sine wave to draw the position of a series of circles:

```
float theta = 0.0f;       // Start angle at 0
float waveAmplitude = 75;  // how high you want the wave to be drawn
float dx;                 // Value for incrementing X, to be calculated
                             //as a function of period and xspacing
float[] yvalues;          // Using an array to store height values for
                             //the wave (not entirely necessary)

void setup() {
    size(300,300);
    smooth();
    stroke(255);
    yvalues = new float[500];
}

void draw() {
    fill(0, 50);
    rect(0, 0, 300, 300);
```

```
        noFill();
        // Increment theta (try different values for 'angular velocity' here
        theta += 0.05;

        // For every x value, calculate a y value with sine function
        float x = theta;
        int i;
        for (i = 0; i < yvalues.length; i++) {
            yvalues[i] = sin(x)*waveAmplitude;
            x+=0.5;
        }
        for (i = 0; i < yvalues.length; i++) {
            ellipse(i*mouseX/10, width/2+yvalues[i], 16, 16);
        }
    }
```

This is an easy way of creating motion for many sprites using the same equation, but it can be a problem if you have to create many different kinds of motion for several different objects. A better way of moving several elements in different styles is to use tweens.

Creating Tweens

Setting an initial point and then using random or semi-random values to set the next position that will be drawn can easily create some organic looking motion, but not all organic motions have to be random, though. Gravity, acceleration, and deceleration are all organic and natural motions that can be simulated with simple equations. One way of doing this is by using the ShapeTween library. ShapeTween is a library for Processing that provides an easy way of animating elements within a sketch. *Tween* is short for "between" and is used to describe an animation that occurs by interpolating the state of an animation between two defined states. This differs from traditional cell animation, where every frame is defined.

To use the `tween()` method, make an instance of the `Tween` class and then pass it four parameters: `parent`, an object to draw in (your Processing application); `duration`, the duration of the tween; `durationType`, what units to measure the duration in (seconds or milliseconds); and `easing`, an optional easing method. The `easing()` method is a way to use mathematical modeling to give the tween more character:

```
Tween( parent, duration )
Tween( parent, duration, durationType )
Tween( parent, duration, durationType, easing )
```

The following is a snippet that creates a tween using a cosine equation to determine the motion and moves an ellipse:

```
import megamu.shapetween.*;

Tween tween;
void setup(){
    tween = new Tween(this, 2, Tween.SECONDS, Shaper.COSINE);
```

```
}

void draw(){
    background(255);
    ellipse(tween.time()*width, tween.position()*height, 4, 4);
}
```

Run this application so that you can see how the tween works because animations are very difficult to capture in a still image; you'll learn far more looking at it yourself running in a Processing application. Here's another example of a tween. This time, instead of tweening height and width, you'll tween a color:

```
import megamu.shapetween.*;

Tween tween;

color[] colorArr = new color[3];

int colorInd1 = 0;
int colorInd2 = 1;

void setup(){

    size( 200, 200 );
    colorMode( RGB );
    smooth();
    tween= new Tween(this, 3.0, Tween.SECONDS, Shaper.COSINE);
    tween.start();

    colorArr[0] = color(  255, 0, 0 );
    colorArr[1] = color( 0, 0, 255 );
    colorArr[2] = color( 0, 255, 0 );
}

void draw(){
```

When the code stops running, switch the colors that will be tweened to by advancing the array indexes to get the next elements in the array and restart the tween:

```
    if(!tween.isTweening()) {

        colorInd1++;
        colorInd2++;

        if(colorInd1 > 2) { colorInd1 = 0; }
        if(colorInd2 > 2) { colorInd2 = 0; }

        tween.start();
    }
    // tween the colors
    color c = lerpColor( colorArr[colorInd1], colorArr[colorInd2],
        tween.position() );

    noStroke();
    fill( c ); // set the color to our new tweened color
```

```
        ellipse( 100, 100, 140, 140 ); // draw a circle using that new color
    }
```

Tweening in oF uses all the same mathematical principles but doesn't have a nice easy tween library to use. However, the semi-famous easing equations of Robert Penner have been ported to C++ specifically for use in oF. You can grab these in the code downloads for this book and get started building your own tweens using those equations.

Interview: Casey Reas

Casey Reas is a digital artist, teacher, writer and one of the founders of the Processing project. His works have been shown at the Seoul Museum of Art (Seoul, South Korea), Laboral (Gijon, Spain), and The Cooper-Hewitt Museum (New York). He is also an associate professor and chair of the department of Design Media Arts at the University of California, Los Angeles.

You've talked about a perceived divide between artists who are algorithmically inclined and artists who are more straightforward "algorithmic artists." Do you feel that this division has lost its relevance or power?

Casey Reas: I would prefer for this division to be irrelevant in the present. I do make distinctions among the long list of pioneers, but not along this division. I feel that my work relates to the concepts of [Sol] LeWitt and [Manfred] Mohr, but not strongly to [Nam June] Paik or [Ben] Laposky. It's fascinating to dig deeper and deeper into this history to learn about the different perspectives. Most of the first people to use a computer to make visual images were researchers and academics who had access to the rare and expensive computers of the time. They saw the potential and explored it. There were fewer people with backgrounds in the arts (for example Manfred Mohr and Vera Molnar) who were working and thinking algorithmically using traditional media. They started to use computers to further explore their ideas.

How much of the creation of Processing was simply wanting a tool for your own uses and how much was an interest in creating a tool for others to use, and how did these two fit together?

Casey: It was more of a desire to make a tool for teaching and for sketching software. We weren't happy with the tools that were used to teach programming within the visual arts context, and we also thought we could improve upon the tools we used to write quick programs to explore ideas. At that time (2001) we were writing all of our software in C++, using OpenGL for graphics. We had a robust development system for writing efficient and reliable software, but it didn't let us quickly prototype concepts. Early versions of Processing allowed us to do two important things. First, we could teach workshops where students could start to write programs after only a few minutes of instruction. Second, we could sit down for a few hours and quickly write many short programs. Processing has now evolved to a production environment, and Ben and I both write software using Processing from start to finish.

Can you talk a little about how you first started working with John Maeda?

Casey: In 1998, I was working at a studio in New York called i/o 360, and John stopped by one day to see Gong Szeto, one of the studio's owners. I was fascinated with John's work (I had seen his Reactive Books, posters, and a stunning promotional piece for Gilbert Paper), and we started a conversation that eventually led to my application to the MIT Media Lab. I worked within his Aesthetics and Computation Group as a research assistant for two years from 1999 to 2001. I didn't know how to program when I first met John (I could script a little Lingo and wrote some BASIC and LOGO as a kid). He encouraged (enforced, actually) that I learn enough to be able to hit the ground running if I was accepted to MIT. I started learning on my own and took classes at NYU. I started at MIT about 18 months later.

Pieces like "The Protean Image" play with the notion of users assisting in generating a process and altering software through their input and decisions. There's both a very anonymous survey-like element to the work and, on the other hand, a sense that one is creating something.

Casey: "The Protean Image" was an exploration to give people access to the same systems that I had been using in my work for the last three years. It's a metaprocess that can be used to generate software. The goal of working with other people is to go beyond my personal preferences and limitations. Some unique configurations that I hadn't yet explored came out of watching people use the machine for a few hours. The cards for the project are a way to easily and inexpensively encode the decisions so they can be easily compared, changed, and categorized. They let people think about the decisions they are making and not about how to use an interface. A person can easily grab a group of cards off the wall and try them in the machine. If they like one, they can pull it out and look at the marks. It's very easy to use. Also, the Protean Image cards are the identical size to the standardized mainframe punch cards. It's a wink to the past.

Some of your talk about systems and evolution has some parallels with how people describe working with artificial intelligence or artificial life research.

Casey: It was an interest in artificial life that got me started on this work in the first place—also the related study of emergence and distributed thinking. Books by Resnick, Kelly, Holland, Braitenberg, and Levy planted some fertile seeds in my head. I'm also interested in the new artificial intelligence—people building embodied behavior following the ideas articulated by Rodney Brooks, et al. I'm not an expert in either area, and I'm not personally interested in building life or intelligence. I'm interested in the ideas that underlie that work.

When do you consider that a system you've made is successful? What sorts of things do you find yourself looking for in it?

Casey: It's both an intellectual evaluation and an intuitive reaction. My recent work is about the relationship between a simple set of rules and the result. I look for the simplest rules to create the most unexpected results. I want the viewer to have access to both, and I'm interested in how he or she is able to imagine the space between the two. I also don't place the emphasis on objects, I'm very focused on systems and processes and the relations among objects. I'm also interested in the relation between the natural and the artificial. I strive for an ambiguous space between.

In some cases the data in your works are obvious and important to understanding the context of the work, and in other cases the data is hidden. How do you regard the interrelation of data and art?

Casey: Data and art can be synonymous or distinct. It depends on the context.

What does the notion of artistic process give to you as an artist?

Casey: It shifts the focus from things to concepts. From a realization to potential. For me, potential is always more exciting than the resolution.

What's your impression of how people are using Processing?

Casey: I'm so impressed with how people are using the ideas behind Processing as well as their explorations with the tool itself. The list of exciting subprojects is too long to mention. An area of new emphasis is OpenGL and GPU integration. The kind of advanced work created by Aaron Koblin, Robert Hodgin, and Karsten Schmidt requires more power than plain Processing. New libraries are in development to make these features more accessible. I'm very excited by how people are extended Processing. This has been an amazing year for people exploring and hacking. There have been impressive implementations in JavaScript (Processing.js by John Resig), Ruby (Ruby-Processing by Jeremy Ashkenas), and ActionScript (Procesing.as by Tim Ryan), and people have made prototypes for Python and Scala integration. The original Processing offshoot, Mobile Processing, is doing well. And the sister electronics projects Arduino and Wiring are making a tremendous impact within their domain. We've never thought of Processing exclusively as a Java wrapper; it's more of a point of view and set of priorities. We're excited to see this propagate. Ben and I both wish there was more emphasis on mobile development, but this is now seriously happening as well (it's too early to talk about this).

Do you feel that your artwork and your work in creating and advancing the Processing language and IDE are part of the same project for you—that they're somehow integrated?

Casey: They all overlap to an extent, but they also compete for focus. I have three areas of professional focus: Processing development, teaching, and artistic practice. Teaching and Processing are easy to align. The book *Processing: A Programming Handbook for Visual Designers and Artists* (MIT Press) grew out of my interactions with the students at UCLA, and some of my graduate students have written interesting Processing libraries. I use the Processing software for most of my artwork, but the time that I spend teaching and working on Processing completes with this pursuit.

As a teacher, how do you approach getting art and design students to think about the creation of systems in code?

Casey: Everything is done through exercises and projects. In the first undergraduate class, we work on short exercises to teach technical skills and mix in longer projects that allow them to apply their emerging technical skills to ideas about interactivity. The subsequent classes focus on developing one 10-week project. It starts with conceptual development, moves into prototyping, and then moves to refinement. In this class, we've done live visual performances to music, we've visually explored emergent systems, and we have worked on ideas for navigating through different types of software

space. We also read and discuss relevant texts and spend hours talking about examples and the students' work.

Do you have any advice for artists who want to work with code but who are lacking a technical or engineering background?

Casey: Some people can teach themselves with the help of online examples and books, and others need the structure of a class to get started. There are some great books and excellent classes to sample. It doesn't take an extensive technical background, just motivation and time. I think people should realize, though, that although learning to program is not difficult, it can take years to be proficient and more than a lifetime to master. Using software such as Photoshop or Illustrator to edit photographs or draw is another way to indirectly get into programming. The rules of programming are embedded within the menus and interfaces. The type of thinking necessary for programming can be introduced indirectly.

Are there ways to work with code that perhaps don't immediately generate a product that can still be helpful to an artist?

Casey: Yes, of course. Code can be used as part of a larger process. For example, I've used code to generate data that I've transformed into sculpture. The same way many artists make sketches in Photoshop before they begin to paint, programs can be written to work through or explore ideas in preparation for another medium.

Are there particularly important things for an artist to understand as they attempt to integrate their artistic ideas with programming-based ideas?

Casey: Among burgeoning artist/programmers, I think the most interesting software is typically written by artists who were using ideas and concepts related to software before they actually started to write code or collaborate with a programmer. This is a natural progression that puts ideas before technique.

Using Vectors

What are vectors for? At its root, the act of creating computer graphics consists of two fundamental activities: creating a world inside a computer that is defined by mathematical entities and some rules binding those all entities together, and producing two-dimensional images of that world. A graphics program is a camera onto that world. When the camera says that it's about to take a picture, all those mathematical entities are converted into shapes, colors, and lines. Since all the entities that you'll use to draw are mathematical entities, you'll have to familiarize yourself with a little bit of mathematics. Not much mathematical knowledge is needed; basic geometry and some basic algebra is plenty to get you started. That said, a quick review is in order.

In geometry, a *point* is a location in space. A point does not have any size. Its only property is a location. A geometrical vector has two properties: length and direction. For example, a vector value might be something like "50 kilometers an hour west." A vector does not have a fixed location in space, though you can use vectors to draw lines

if you provide a starting location for the vector. This combination of "distance and direction" is sometimes called a *displacement*. So, what kinds of things are vectors used for? They're used to find the point at which two lines intersect, the distance of a point to a line, or whether a shape is convex or concave. They're used to find objects closest to the eye or determine whether a plane is facing away from the camera. They're used to determine how much light hits a surface (illumination), how much of that light is seen by the viewer (reflection), and what other objects are reflected in that surface (ray tracing). Vectors are also quite important when trying to do any physics, creating collisions between objects, creating barriers, and making gravity or other forces that affect objects.

Vectors are widely used in Processing code, particularly the `PVector` class that is included with the core Processing code. It allows you to easily represent a 3D vector and that greatly simplifies trying to create animations in three dimensions. The basics of working with vectors can be grasped quite easily. Every object that is affected by a force should be assigned a vector. Any forces that need to act on that object will also be assigned a vector.

The `PVector` describes a two or three-dimensional vector with either two or three properties to describe the vector. In the case of a 3D vector, those are x, y, and z. It's commonly used to store position, velocity, or acceleration. You might be wondering why position is included in that list because, in physics at least, position is a point. Think of a ball moving across the screen: at any given instant, it has a position, a velocity, and acceleration. To determine what the position will be you'll want to combine the velocity, acceleration, and position together, but since velocity and acceleration are represented as vectors, it's easier to use the vector math methods of the `PVector` class. The `PVector` has two constructors that you can use for creating 2D and 3D vectors respectively:

```
PVector(float x, float y)
PVector(float x, float y, float z)
```

That vector can be used to calculate the force and direction of that object at a given moment. So first things first, you'll need an object, `Ball`, to move around (see Example 9-7). It will need to have a few vectors to keep track of the different forces on it. An object moving around in two-dimensional space will have a location, a velocity, and a rate of acceleration. Each of these can be represented by a vector.

Example 9-7. Ball

```
class Ball {
    PVector location;
    PVector velocity;
    PVector acceleration;
    float mass = 20; // how heavy are we?
    float maximum_velocity = 20; // we'll use this to make sure
 things don't get too fast
    // How bouncy are we? 1 means we don't lose any speed in bouncing,
```

```
// higher means we gain speed, lower means we lose it
float bounce = 1.0;
```

Now the `Ball` needs a constructor. Since Processing provides overload methods, you can use those to make two constructors: one that you create vectors for and one that initializes values on its own:

```
Ball(PVector initialAcceleration, PVector initialVelocity, PVector initialLocation) {
    acceleration = initialAcceleration.copy();
    velocity.set(initialVelocity);
    location.set(initialLocation);
}

Ball() {
    acceleration = new PVector (0.0, 0.0, 0.0);
    location = new PVector (0.0, 0.0, 0.0);
    velocity = new PVector (1.0, 0.0, 0.0);
}
```

Now that the basic elements of the `Ball` have been established, you'll need to add forces to the `Ball` to represent wind, gravity, and any other elements that you might want to use. The location vector shouldn't be manipulated, but the velocity and acceleration should, since those are the values that will be used to determine the new position of the `Ball` when the `draw()` method is called.

To accurately represent the way that the mass of an object affects its movement, you need to modify the force using the mass. Mathematically, this is done by dividing the force vector by the mass. The `Vector3D` class provides a simple convenience method that you can use to do this, the `div()` method. The acceleration of the `Ball` also needs to be modified by the force; a negative force slows the acceleration, a positive force accelerates it. This is done mathematically by adding the new force vector to the acceleration vector to ensure that all the values of the acceleration are affected:

```
void addForce(Vector3D force) {
    force.div(mass); // make sure the force is modified by the mass
    acceleration.add(force); // the acceleration is affected by the force
}
```

Now that new forces can modify the object's acceleration, all that's left to do is make a method that can do all the necessary calculations in each frame of the animation. This is going to be broken into two methods to make it more readable and better organized. The first method, `update()`, does all the calculations required to determine the current position of the `Ball`:

```
void update() {
    velocity.add(acceleration); // add the accleration to the velocity
    velocity.limit(maximum_velocity);
    location.add(velocity);
    // the acceleration all comes from the forces on the Ball which are reset
    // each frame so we need to reset the acceleration to keep things within
    // bounds correctly
    acceleration.set(0.0f,0.0f,0.0f);
```

```
    // bounce off the walls by reversing the velocity
    if (location.y > height) {
        velocity.y *= -bounce;
        location.y = height;
    }
    if (location.y < 1) {
        velocity.y *= -bounce;
        location.y = 1;
    }
    if ((location.x > width) || (location.x < 0)) {
      velocity.x *= -bounce;
    }
```

The `drawFrame()` method handles actually drawing a small white ball to the screen:

```
// Method to display
void drawFrame() {
    update();
    ellipseMode(CENTER);
    noStroke();
    fill(255);
    ellipse(location.x, location.y, 20, 20);
 }
}
```

Now it's time to put the `Ball` in play:

```
Ball ball;
PVector wind;

void setup() {
    size(200,200);
    smooth();
    background(0);
    ball = new Ball();
    wind = new PVector(0.01, 0.0, 0.0);
}

void draw() {
    fill(0, 10);
    rect(0, 0, 200, 200);

    // Add gravity to thing
    // This isn't "real" gravity, just a made up force vector
    Vector3D grav = new PVector (0,0.05);
    ball.add_force(grav);

    // give our Wind some random variation
    float newWind = noise(mouseX, mouseY)-0.5;
    wind.x += newWind;
    wind.limit(1.0);
    ball.addForce(wind);
    ball.drawFrame();
}
```

openFrameworks also provides three different classes in the `ofxVectorMath` add-on that is available in the oF download: `ofxVec2f`, `ofxVec3f`, and `ofxVec4f`. The numbers within those class names tell you how many dimensions the class operates on. Why an `ofxVec4f`? Without getting too deep into the mathematics of it, among other things, using a 4D vector helps prevent a certain mathematical error when rotating 3D objects that's famous enough to have its own name: "gimbal lock." Kodicek has a good explanation of this in *Mathematics and Physics for Programmers* (Charles River Media), as does Fletcher Dunnin's *3D Math Primer for Graphics and Game Development* (Jones and Bartlett).

Each of these classes defines a few key methods:

`align()`
: Takes another vector as a parameter and returns `true` if the two vectors are looking in the same direction and `false` if they aren't. This is quite useful when trying to line things up or determine the position of objects.

`rescale()`
: Takes a float that represents the new length of the vector. Remember that the vector has a direction and a length.

`Rotate()`
: Turns the vector around its origin. This is one of the tasks you'll do most frequently with a vector.

`normalize`
: Reduces the length of vector to one, which makes it faster and easier to compare the directions of two vectors. A normalized vector also prevents any odd behavior from overly large or small vectors.

`limit`
: Sets a limit on the size of the vector, both in the x and y values. This is quite handy if you need to restrain a value for some reason, keeping something from going too fast, for example.

`angle`
: Takes a vector as a parameter and compares the angles between the two vectors in degrees. This is a very helpful method for determining the orientation of objects to one another.

All the `ofxVectorMath` vector classes override the mathematical functions, `+`, `-`, `/`, and `*`, to perform vector addition, subtraction, division, and multiplication. This is, again, very convenient, as you'll see in the following examples. The `Ball` example in the previous Processing example will be translated to C++ use in oF so that you have an object you can manipulate (see Examples 9-8 and 9-9).

Example 9-8. Ball.h

```
#ifndef BALL
#define BALL
```

```
#include "ofMain.h"
#include "ofxVectorMath.h"

#define otherObjsCount 2

class Ball{
    public:
        Ball();

        void addForce(ofxVec3f force);
        void updateBall();

        ofxVec3f location;
        ofxVec3f velocity;
        ofxVec3f acceleration;
        float mass; // how heavy are we?
        float maximum_velocity;
        // How bouncy are we? 1 means we don't lose any speed in bouncing,
        // higher means we gain speed, lower means we lose it
        float bounce;
        int color[3];

        // Method to display
        void drawFrame();
};

#endif
```

Example 9-9. Ball.cpp

```
#include "Ball.h"

Ball::Ball() {


    color[0] = 255;
    color[1] = 255;
    color[2] = 255;

    location.set(ofRandom(0.0f,800.0f), 0.0f, 0.0f);
    velocity.set(0.0f, 0.0f, 0.0f);

    acceleration.set(0.0f, 0.0f, 0.0f);

    mass = ofRandom(0, 20); // how heavy are we?
    maximum_velocity = 5;
    // How bouncy are we? 1 means we don't lose any speed in bouncing,
    // higher means we gain speed, lower means we lose it
    bounce = 1.0;

}

void Ball::addForce(ofxVec3f force) {
    force /= mass;
```

```
    acceleration += force;
}
```

To update the `Ball`, add the acceleration to the velocity, and then add the velocity to the location. If the acceleration is constant then the `Ball` will keep going faster and faster; if the acceleration is negative then the `Ball` will begin slowing down:

```
void Ball::updateBall() {

    velocity += acceleration;
    location += velocity;
    acceleration.set(0.0f,0.0f,0.0f);
    // this assumes that the height of the window is 800px and the
       //width is 1000
    // you can also use the getWindowSize() method to determine how
       //large the window is
    if (location.y > 800) {
        velocity.y *= -bounce;
        location.y = 1000;
    }
    if(location.y < 0) {
        velocity.y *= -bounce;
        location.y = 0;
    }
    if (location.x > 1000) {
        velocity.x *= -bounce;
        location.x = 1000;
    }
    if (location.x < 0) {
        velocity.x *= -bounce;
        location.x = 0;
    }

}

// Method to display
void Ball::drawFrame() {
    ofSetColor(color[0], color[1], color[2]);
    ofCircle(location.x, location.y, 50);
}
```

That application that uses the `Ball` class will simply instantiate several of them and apply gravity and wind forces to them with two additional features added. Notice the `mouseMove()` method of the application; the mouse movement is altered by the user's mouse position. The center of the screen has the x and y gravity both at 0. Moving the mouse pointer alters the direction of the gravity, so positioning the mouse pointer at the top of the screen means the gravity will draw the objects to the top of the screen (Examples 9-10 and 9-11).

Example 9-10. ch09Vector.h

```
#ifndef CH9_VECTOR
#define CH9_VECTOR
```

```
#include "ofMain.h"
#include "Ball.h"

#define numBalls 3

class ch09vector : public ofBaseApp {

    public:

        void setup();
        void update();
        void draw();

        void mouseMoved(int x, int y );
        void drawConnectors();

        ofxVec3f wind;
        ofxVec3f gravity;

        Ball b1;
        Ball b2;
        Ball b3;

        Ball* balls[3];
};

#endif
```

Example 9-11. ch09Vector.cpp

```
#include "ch09vector.h"

void ch09vector::setup() {
```

Here are the two forces that will be applied to each `Ball`, creating constant forces on them:

```
        gravity = ofxVec3f(0.0, 1.0f, 0.0);
        wind = ofxVec3f(0.5f, 0.0, 0.0);

        b1.color[0] = 0;
        b2.color[1] = 0;
        b3.color[2] = 0;

        balls[0] = &b1; //use a reference so we can access the pointer later
        balls[1] = &b2;
        balls[2] = &b3;

        ofEnableAlphaBlending();
        //for smooth animation, set vertical sync if we can
        ofSetVerticalSync(true);
        // also, frame rate:
        ofSetFrameRate(60);
    }
```

On each call to `update()`, the application adds the forces to the balls again so that their velocities and accelerations will reflect constant forces acting on them like wind and gravity:

```
void ch09vector::update(){

    b1.addForce(gravity);
    b2.addForce(gravity);
    b3.addForce(gravity);

    b1.addForce(wind);
    b2.addForce(wind);
    b3.addForce(wind);

    b1.updateBall();
    b2.updateBall();
    b3.updateBall();

}

void ch09vector::draw(){

    b1.drawFrame();
    b2.drawFrame();
    b3.drawFrame();

    drawConnectors();
}


void ch09vector::mouseMoved(int x, int y ){
    gravity.set(float((x -500)/100), float((y - 500)/100), 0);
}
```

The `drawConnectors()` method draws a line from the center of each `Ball` drawn on the screen by reading the location vector of each `Ball` and using that to supply values to an `ofLine()` call:

```
void ch09vector::drawConnectors() {
    int i, j;
    for(i=0; i<numBalls; i++)
    {
        for(j=i+1; j<numBalls; j++) {
            ofLine(balls[i]->location.x, balls[i]->location.y,
                balls[j]->location.x, balls[j]->location.y);
        }
    }
}
```

So far so good, but now you'll add some collision detection to the balls. Add the following to Example 9-11:

```
void ch09vector::checkCollision()
{
    int i, j;
```

```
    for(i=0; i<numBalls; i++)
    {
        for(j=i+1; j<numBalls; j++)
        {
            if(balls[i]->location.x+100 > balls[j]->location.x &&
                balls[i]->location.x < balls[j]->location.x+100)
            {
                if (balls[i]->location.y+100  > balls[j]->location.y &&
                    balls[i]->location.y < balls[j]->location.y+100)
                {
                    balls[i]->collision(balls[j]);
                }
            }
        }
    }

}
```

This determines whether the balls are actually colliding with one another. You'll also need to add a `checkCollision()` call to the `update()` method of the application. The call to the `collision()` method of the `Ball` implies that you'll be adding a `collision()` method:

```
void Ball::collision(Ball* b1)
{
    // this helps cut down on typing :)
    float m1 = mass;
    float m2 = b1->mass; // note pointer access using the ->
    float x1 = location.x;
    float x2 = b1->location.x;
    float y1 = location.y;
    float y2 = b1->location.y;

    // we'll just declare all these things we'll need at once
    float newMass, diff, angle, newX, newY, newVelocityX,
          newVelocityY, fy21, sign;

    newMass = m2/m1;
    newX = x2-x1;
    newY = y2-y1;
    newVelocityX = b1->velocity.x - velocity.x;
    newVelocityY = b1->velocity.y - velocity.y;

    //  If the balls aren't heading at one another, we don't want to alter them
    //  because they could be heading away from each other.
    if ( (newVelocityX*newX + newVelocityY*newY) >= 0) return;

    fy21=1.0E-12*fabs(newY);
    if ( fabs(newX)<fy21 ) {
        if (newX<0) { sign=-1; } else { sign=1;}
        newX=fy21*sign;
    }

    // Now that we've figured out which direction things are heading,
    // set their velocities.
```

```
        angle=newY/newX;
        diff = -2 * (newVelocityX + angle * newVelocityY)/((1 + angle * angle) *
            ( 1 + newMass )) ;
        b1->velocity.x = b1->velocity.x + diff;
        b1->velocity.y = b1->velocity.y + angle * diff;
        velocity.x = velocity.x - newMass * diff;
        velocity.y = velocity.y - angle * newMass * diff;

    }
```

The mathematics behind this method is simpler than the code implies. For now, it's more appropriate to look at the general algorithm and explain how vectors play into that than explain the equations. The general steps for doing collision detection follow:

1. Determine the vector that results from subtracting the two `Ball` vectors from one another.
2. Determine the angle of that vector so that you know how direct the collision is: is it head-on or a glancing collision?
3. Determine the difference in velocity based on the initial velocities and the angle.
4. Modify the velocity vector of each `Ball`.

Using Graphical Controls

Graphical controls provide an easy way for a user to modify a value, to provide yourself an interface for testing, to provide a simpler alternative to a more complex interaction, or to explicitly request information from a user. In certain situations, an interaction can be very complex and novel, and in other situations, it's more appropriate to use a simple text input or button that a user can click. Determining what type of control is most appropriate for a given situation is largely a matter of understanding how a user will approach your application, what they'll want to accomplish, and what will help them do that. These are commonly referred to as *use* cases, and it involves considering carefully the environment, the users, the nature of your application, the understanding that users will have of the system, and the actions or tasks that they may want to accomplish.

Any interface is a map of the possible interactions that a user can have with a system. As such, the user will always make great assumptions about your system based on the interface you provide them. Just as you need to ensure that the control is appropriate for the context and the data required of a user, the feedback for the interaction should map to the action that a user takes. When a user inputs a value, it is quite important that they can understand directly how that value has changed the system that they are interacting with. Without this, the interaction loses meaning, the users role is reduced, and your application ceases to be a communication.

ControlP5 Library

ControlP5 is a controller library for Processing that enables you to easily add controls, toggle them on or off, set and read their feedback, and organize control panels of grouped controls. The German artist, designer, and programmer Andreas Schlegel developed the ControlP5 library as well as the oscP5 OSC communication library and sDrop file-system library. The root of the ControlP5 library is the `ControlP5` class, which is declared in the Processing application and then has controls references added to it.

ControlP5 looks something like this:

```
import controlP5.*;
ControlP5 controller; // declare the controller

void setup() {
    // tell the controller what application will be running it
    controller = new ControlP5(this);
}
```

Once the controller has been created, buttons can be added by using the `addButton()` method using the syntax:

```
addButton(String buttonName, Object value, int x, int y, int width,
    int height)
```

This method adds a button with the name passed as the `buttonName` parameter and with the value set as the value parameter. The value will tell your program that the button has been clicked.

Event Handling

An important concept in working with controls is an event. An easy way to think of an event is to think about an alarm clock. You'll hear lots of things at night—a dog barking, a car horn, maybe the people next door—but when you hear your alarm clock, you know it's time to get up. You can think of this as there being lots of events during the wee hours of the morning but only one event that tells you that it's time to get up. In programming terms, you've set an event handler for the alarm clock event. Only that event will trigger the action that you've associated with it.

In programming terms, an *event handler* is a method or function that handles a specific kind of event. Usually that event is defined by a class; for instance, in the case of the ControlP5 library, the event is called a `ControlEvent`, and the event handler function is always called `controlEvent()`. The event handler looks like so:

```
void controlEvent(ControlEvent event) {
    println("got a control event from controller "+event.controller().value());
}
```

The `ControlEvent` object has a reference to whatever the last value of the last control that was modified in it was, so you can access that by accessing the `controller()` method of the event and the `value()` method of the controller:

```
event.controller().value()
```

You'll see this same pattern used differently in the openFrameworks GUI add-on later in this section. For now, though, you'll put the ControlP5 library to use. Here, the button values set the background color:

```
import controlP5.*;

ControlP5 controlP5;
int buttonValue = 0;

void setup() {
    size(400,400);
    controlP5 = new ControlP5(this);
    controlP5.addButton("white",255,100,160,80,20);
    controlP5.addButton("black",0,100,190,80,20);
}

void draw() {
    background(buttonValue);
}

void controlEvent(ControlEvent event) {
    buttonValue = int(event.controller().value());
}
```

Another feature that the ControlP5 library offers is the ability to set `id` values for each control so that the `controlEvent()` method can determine which control has been altered. Here, a slider and knob control are kept synchronized by examining the ID of the control that was most recently edited:

```
void controlEvent(ControlEvent event) {
    int id = event.controller().id();
}
```

Now you know the `id` of the control that sent the signal. For this to be truly helpful, though, you need to set an `id` for each control. Luckily, that's quite easy. Declare a slider:

```
Slider s;
```

Then, add it to the `ControlP5` instance and assign it an `id`:

```
    s = control.addSlider("slider",0,255,128,100,160,10,100);
    s.setId(1);
```

The same pattern used previously to set the background color is used here with a knob:

```
import controlP5.*;

// Declare the ControlP5
ControlP5 control;
int bgColor = 0;
int sliderVal = 100;
// Declare the two controls that will be used
Knob k;
```

```
Slider s;

float sliderValue;
float knobValue;


void setup() {
    size(400,400);
    control = new ControlP5(this);
    s = control.addSlider("slider",0,255,128,100,160,10,100);
    s.setId(1);
    k = control.addKnob("knob", 0,255,128, 200, 160, 40);
    k.setId(2);
}

void draw() {
    background(0, 0, bgColor);
    k.setValue(knobValue);
    s.setValue(sliderValue);
}

void controlEvent(ControlEvent event) {
    int id = event.controller().id();
    if(id == 1) {
        knobValue = event.controller().value();
    } else {
        sliderValue = event.controller().value();
    }
    bgColor = int(event.controller().value());
}
```

Importing and Exporting Graphics

One aspect of 2D graphics that we haven't discussed yet is how to work with graphics in formats other than what's provided in the Processing or openFrameworks environments. You might want to save graphics in a file format that another application can open and run; you might want to print your graphics, or you might want to import vector graphics from a file and display them on the screen. All this can be done in two different ways. The first is to use bitmapped images like JPEG files. This approach will be discussed Chapter 10. The other approach is to use Encapsulated PostScript (EPS) files that define vector information about a graphic. If you've ever used a vector drawing program such as Adobe Illustrator, OpenOffice Draw, or Inkscape, you've probably seen how a drawing can be created in these programs and then exported. The EPS format is a lightweight way to save graphics because, like all vector graphics, it doesn't require all the bitmap data to save the information of an image; it requires only the positions of the vectors and the fills that those shapes use. PostScript is primarily designed to present publication-quality text because it can scale up to billboard size or down to postcard size without affecting the quality of the image.

Using PostScript in Processing

SimplePostscript is a library that can be used to output *.ps* files. It implements most of the basic PostScript functionality used for drawing. The following code creates a simple drawing and saves it out to a *.ps* file:

```
import SimplePostscript.*;
SimplePostscript ps;

void setup() {
    size(200,200);
    ps=new SimplePostscript(this);
```

If this file exists, then we just open it, otherwise this line will create the file in the same folder as the application:

```
    ps.open("pattern.ps",0,0, 200,200);
    noLoop();
}

void draw() {
    background(255);
    stroke(0);
    ps.setlinewidth(0.25);
```

Now this begins drawing to the file itself:

```
    ps.setgray(0);
    int step=2;
    for(int y=0; y<height; y+=step) {
        beginShape();
        for(int x=0; x<width; x+=step/2) {
            float z=y+calc(x,y);
            vertex(x,z);
            if (x==0) ps.moveto(x,z);
            else ps.lineto(x,z);
        }
        endShape();
        ps.stroke();
    }
    ps.close();
}

float calc(float x,float y) {
    return 10*(sin(x*0.1)*cos((y+x*0.33)*0.1));
}
```

There is another option for importing vector graphics files: Scalable Vector Graphics (SVG). SVG files are, for the sake of this discussion, equivalent to PostScript files. They have the same type of information and similar uses. PostScript is great, but there's not currently an easy way to import PostScript files into Processing. Importing SVG files is easy and is built into Processing. All that's required is declaring a `PShape` file and then drawing it using the `shape()` method:

```
PShape s;
void setup() {
    s = loadShape("svgFile.svg");
    smooth();
    noLoop();
}

void draw() {
    // the shape method signature looks like this:
    // shape(PShape s, int x, int y, int width, int height)
    shape(s, 10, 10, 80, 80);
}
```

Both SVG and PostScript are ways to save, store, or prepare to print vector graphics.

Using PostScript Files in oF

Using PostScript files in oF is similar to using them in Processing. There isn't a way to work with SVG at the moment. All the PostScript functionality is contained in an add-on called ofxVectorGraphics. You access the ofxVectorGraphics library through an object called ofxVectorGraphics. Simply create one of these in your header file, and call the `beginEPS()` method to indicate that all the points you're creating should be included in the generated EPS file. All the drawing for an EPS file is done using the graphics methods of the ofxVectorGraphics object that you've created. These are similar to the standard 2D drawing methods of oF: `rect()`, `ellipse()`, `arc()`, `bezier()`, `setColor()`, and so on. Take a look at the *addons/ofxVectorGraphics/src/ofxVectorGraphics.h* file to see them all. When you're finished drawing, call `endEPS()` to save all your graphics data to the file.

The `beginEPS()` method takes a string that will be the name of the generated *.ps* PostScript file:

```
ofxVectorGraphics graphics;
graphics.beingEPS("myFile.eps");
```

When you begin drawing a distinct shape, call `graphics.beginShape()`, and when you're finished drawing that shape, call `graphics.endShape()`. Once you're sure you have all the graphics that you want to be included in your file, you call `graphics.endEPS()`, which writes the file to the data folder of your application.

Here's a simple example of using some of the ofxVectorGraphics drawing methods to draw shapes to a graphics file:

```
#ifndef _VECTOR_GRFX_APP
#define _VECTOR_GRFX_APP

#include "ofMain.h"
#include "ofxVectorGraphics.h"

class VectorGrfxApp : public ofBaseApp{

    public:
```

```
        void setup();
        void update();
        void draw();

        void keyPressed(int key);
        void mouseMoved(int x, int y );
        void mouseDragged(int x, int y, int button);
        void mousePressed(int x, int y, int button);

        ofxVectorGraphics output;
        bool capture;

        vector <ofPoint> pts;
        float angles[30];
        int phaseoffset;
        int steps;
};

#endif
```

The *.cpp* file of the application takes care of actually creating the *.ps* file in the `draw()` method (Example 9-12).

Example 9-12. VectorGrfxApp.cpp

```
#include "VectorGrfxApp.h"
#include "stdio.h"

void VectorGrfxApp::setup(){
    capture = false;
    output.enableDraw();
    ofSetCircleResolution(50);
    phaseoffset = 0;
    steps = 30;
    for(int i = 0; i < 30; i+=2) {
        angles[i] = 2 * PI / steps * i + phaseoffset;
    }
}

void VectorGrfxApp::update(){
    ofBackground(255, 255, 255);
}

void VectorGrfxApp::draw(){
```

You don't want to capture every frame, only capture one frame when the user has pressed the space bar, which sets the capture `property` to `true`:

```
        if(capture){
            output.beginEPS("test.ps");
        }
        // draw all the shapes in red
        output.setColor(0xff0000);
        output.fill();
```

The `ofxVectorGraphics` class defines a few methods for drawing simple shapes. It is used here to create a circle, rectangle, and triangle:

```
output.circle(100, 100, 80);
output.rect(200, 20, 160, 160);
output.triangle( 460, 20, 380, 180, 560, 180);
// set the color we'll be using
output.setColor(0x999999);
output.noFill();
output.setLineWidth(1.0);
float ang;
// use all the angle values set in the mouseMoved method to draw some circles
for(int i = 0; i < 30; i++){
    ang = angles[i] * 180 / PI;
    output.ellipse(ang + 20, ang + 250, mouseX * 0.1 * cos(angles[i]) +
        ang, mouseY * 0.25 * sin(angles[i]) + ang);
}
```

Now, the application draws to the screen using all the points set in the `mouseDragged()` method:

```
if( pts.size() > 0 ){

    int numPts = pts.size();
    output.setColor(0x0000ff);
    output.beginShape(); // begin the shape
    int rescaleRes = 6;  //create nice smooth curves between points

    for(int i = 0; i < numPts; i++){
        //we need to draw the first and last point
        //twice for a catmull curve
        if(i == 0 || i == numPts -1){
            output.curveVertex(pts[i].x, pts[i].y);
        }
        if(i % rescaleRes == 0) {
            output.curveVertex(pts[i].x, pts[i].y);
        }
    }
    output.endShape(); // end the shape
}
// this will write the PS file to the data folder
if(capture){
    output.endEPS();
    capture =false;
}
}
```

If the user presses the space bar, set the capture variable to `true` so that the `draw()` method will write to the *.ps* file:

```
void VectorGrfxApp::keyPressed(int key){
    if(key == ' '){
        capture = true;
    }
}
```

```
void VectorGrfxApp::mouseMoved(int x, int y ){
        // make some values that will be used in the draw() method
           //to draw circles
        for(int i = 0; i < 30; i++) {
            angles[i] = 2 * PI / steps * i + phaseoffset;
        }
        phaseoffset += PI / 180 * 1;
}

void VectorGrfxApp::mouseDragged(int x, int y, int button){

    //we add a new point to our line
    pts.push_back(ofPoint());
    int last = pts.size()-1;

    pts[last].x = x;
    pts[last].y = y;

}
// start storing the line
void VectorGrfxApp::mousePressed(int x, int y, int button){
    pts.clear();
    //store the first point of the line
    pts.push_back(ofPoint());
    pts[0].x = x;
    pts[0].y = y;

}
```

Vector graphics need not be limited to the screen. By creating SVG and PostScript files you can create print ready graphics or lightweight graphics that can easily be stored, sent over the Internet to be displayed on other computers, or reloaded and manipulated later.

What's Next

There are far too many techniques to use in creating vector graphics for it to ever fit in a single book much less in a single chapter. Depending on what you want to do, you might want to consider a few different directions. Chapter 13 of this book covers OpenGL, which is really important to understand if you want to create 3D graphics or complex animations. Although you can get away without knowing any OpenGL when working with Processing, it's much more commonly used in oF applications, and any complex blending and drawing will most likely require it.

If you're not familiar with vector math and linear algebra, then continuing to study a little more wouldn't hurt. Vector math is a very important topic to understand in graphics and computation. I would recommend that anyone seriously interested in learning more about graphics programming study up on matrix math. Beyond math, there are many algorithmic tricks to learn that can help you figure out how to do things that you might not know how to do. *The Graphics Gems* series of books is an interesting

and important series that contains a lot of the core algorithms used in graphics programming. I wouldn't recommend that you buy the books, though, but I would recommend that you go to the online code repository and download the free source code there; it's worth browsing. All the books that have been published about Processing are excellent resources for learning more about graphics in general and Processing in particular. In addition, the oF and Processing forums are both active and helpful. Andrew Glassner is one of the giants of computer graphics, and in addition to the academic works for which he is best known, he has written three introductory books on graphics that might be interesting to you: *Andrew Glassner's Notebook* (Morgan Kaufmann), *Andrew Glassner's Other Notebook* (AK Peters), and *Morphs, Mallards, and Montages* (AK Peters). These are a less technical read and are full of interesting techniques and ideas.

Finally, you should look at the graphics of applications that you like, both on the Processing exhibition page of the Processing site, where many wonderful projects are highlighted, and on the oF website, where several of the most interesting projects are profiled along with example code. Find artists using Processing, oF, other graphics applications such as Flash, Director, and even animation, and think about their graphics and animations both in terms of what makes them visually appealing and in terms of their construction.

Review

There are two coordinates to consider when working with graphics in the screen: the Cartesian coordinate set used in mathematics and the screen coordinate set used when placing pixels on the screen.

Points on the screen are represented by two-dimensional or three-dimensional points. Forces on the screen are represented as two or three-dimensional vectors, that is, a speed and a direction.

A few common strategies will help you organize your drawing code more effectively and create more efficient code: draw using loops, use arrays to store values you'll need for an animation, and use sprite objects that can handle their own drawing routines.

Processing and oF use matrices to transform the drawing space that your graphics objects have been put into. This is an easy and efficient way to assign the positions and rotations of many objects at once.

There are two simple descriptions for motion: organic and mechanical. Organic motion has elements of randomness in it, accelerations and decelerations, while mechanical motion is more constant and repetitive.

Using tweening equations, like the ShapeTween library for Processing, helps you create more natural movement that has interesting acceleration and deceleration characteristics.

Vectors allow you to assign speed, assign direction, and check for collisions between graphical sprites. The PVector class in Processing and the ofxVectorMath library for oF provide many of the mathematical methods that you'll need in order to calculate positions and manipulate vectors.

Graphical controls such as sliders, buttons, scrollbars, and dials are a powerful and familiar way for users to give input and also for you to tune an animation or graphic. Processing users should check out the ControlP5 library; oF users will want to look at the ofxSimpleGUI library.

Particles are another powerful strategy for working with graphics. A particle system consists of three elements: the collection of particles, the emitter for the particles that places them on the screen, and the physics calculations that determine what those particles should do once they are on the screen.

Graphics can also be imported and exported using SVG or EPS file formats that save out vector graphics. The import and export of vector graphics is included in Processing, while in oF that functionality is provided by the ofxVectorGraphics library.

To save out *.ps* files in Processing create a new instance of the `SimplePostscript` class. When you call the `open()` method on the file it will create a *.ps* file in the folder of the application. The `close()` method of the `SimplePostscript` class writes the file. The ofxVectorGraphics uses the `beginEPS()` and `endEPS()` in a similar way to begin and finish writing the file.

CHAPTER 10
Bitmaps and Pixels

In this chapter, you'll learn about video and images and how your computer processes them, and you'll learn about how to display them, manipulate them, and save them to files. Why are we talking about video and images together? Well, both video and photos are bitmaps comprised of pixels. A *pixel* is the color data that will be displayed at one physical pixel in your computer monitor. A *bitmap* is an array of pixel data.

Video is several different things with quite distinct meanings: it is light from a projector or screen, it is a series of pixels, it is a representation of what was happening somewhere, or it is a constructed image. Another way to phrase this is that you can also think of video as being both file format and medium. A video can be something on a computer screen that someone is looking at, it can be data, it can be documentation, it can be a surveillance view onto a real place, it can be an abstraction, or it can be something fictional. It is always two or more of these at once because when you're dealing with video on a computer, and especially when you're dealing with that video in code, the video is always a piece of data. It is always a stream of color information that is reassembled into frames by a video player application and then displayed on the screen. Video is also something else as well, because it is a screen, a display, or perhaps an image. That screen need not be a standard white projection area; it can be a building, a pool of water, smoke, or something that conceals its nature as video and makes use of it only as light.

A picture has a lot of the same characteristics. A photograph is, as soon as you digitize it, a chunk of data on a disk or in the memory of your computer that, when turned into pixel data to be drawn to the screen, becomes something else. What that *something else* is determines how your users will use the images and how they will understand them. A picture in a viewer is something to be looked at. A picture in a graphics program is something to be manipulated. A picture on a map is a sign that gives some information.

Using Pixels As Data

Any visual information on a computer is comprised of pixel information. This means graphics, pictures, and videos. A video is comprised of frames, which are roughly the same as a bitmapped file like a JPEG or PNG file. I say *roughly* because the difference between a video frame and a PNG is rather substantial if you're examining the actual data contained in the file that may be compressed. Once the file or frame has been loaded into Processing or openFrameworks, though, it consists of the same data: pixels. The graphics that you draw to the screen can be accessed in oF or Processing by grabbing the screen data. We'll look at creating screen data later in this chapter, but the real point to note is that any visual information can be accessed via pixels.

Any pixel is comprised of three or four pieces of information stored in a numerical value, which in decimal format would look something like this:

```
255 000 000 255
```

which is full red with 100 percent alpha, or this:

```
255 255 255 255
```

which is white with 100 percent alpha. In the case of most video data, you'll find that the alpha value is not included, supplying only three pieces of information as in the value for red:

```
255 000 000
```

Notice in Figure 10-1 that although the hexadecimal representation of a pixel has the order alpha, red, green, blue (often this will be referenced as ARGB), when you read the data for a pixel back as three or four different values, the order will usually be red, green, blue, alpha (RGBA).

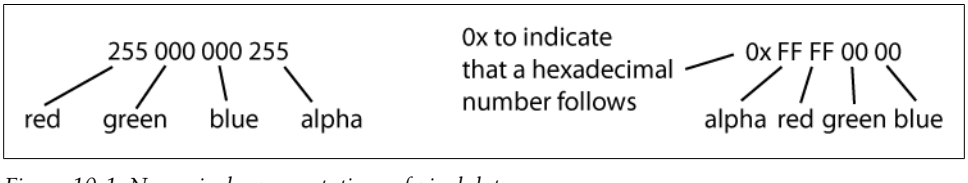


Figure 10-1. Numerical representations of pixel data

The two characters 0x in front of the number tell the compiler that you're referring to a hexadecimal number. Without it, in both Processing and oF, you'll see errors when you compile.

In oF, when you get the pixels of the frame of a video or picture, you'll get four unsigned char values, in RGBA order. To get the pixels of an `ofImage` object, use the `getPixels()` method, and store the result in a pointer to an unsigned char. Remember from Chapter 5 that C++ uses unsigned char where Arduino and Processing use the `byte` variable type:

```
unsigned char * pixels = somePicture.getPixels();
```

So, now you have an array of the pixels from the image. The value for `pixels[0]` will be the red value of the first pixel, `pixels[1]` will be the green value, `pixels[2]` will be the blue, and `pixels[3]` will be the alpha value (if the image is using an alpha value). Remember that more often than not, images *won't* have an alpha value, so `pixels[3]` will be the red value of the second pixel.

While this may not be the most glamorous section in this book, it is helpful when dealing with video and photos, which, as we all know, can be quite glamorous. A bitmap is a contiguous section of memory, which means that one number sits next to the next number in the memory that your program has allocated to store the bitmap. The first pixel in the array will be the upper-left pixel of your bitmap, and the last pixel will be the lower-right corner, as shown in Figure 10-2.

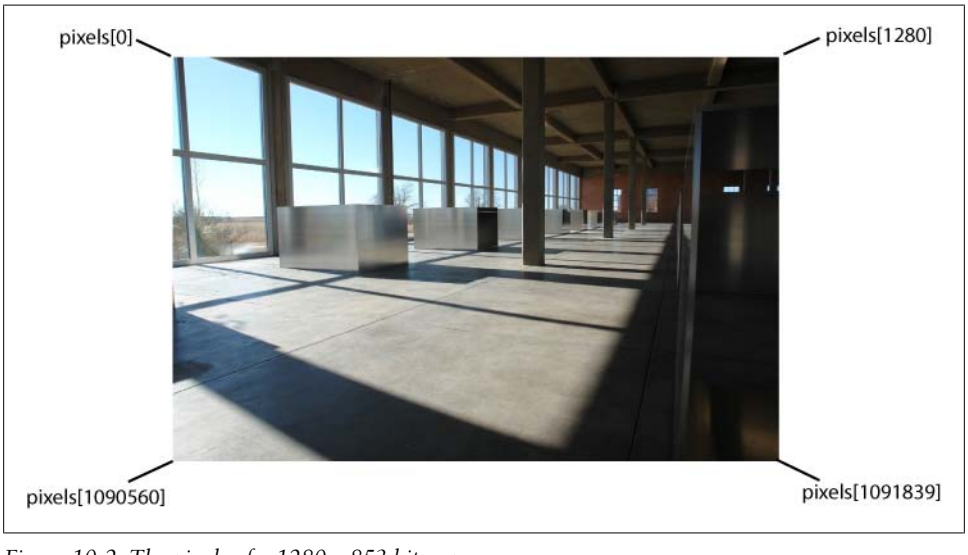


Figure 10-2. The pixels of a 1280 × 853 bitmap

You'll notice that the last pixel is at the same index as the width of the image multiplied by the height of the image. This should give you an idea of how to inspect every pixel in an image. Here's how to do it in Processing:

```
int imgSize = b.height * b.width;
for(int i = 0; i < imgSize; i++) {
   // do something with myImage.pixels[i]);
}
```

And here's how to do it in oF:

```
unsigned char * pixels = somePicture.getPixels();
// one value for each color component of the image
int length = img.height * img.width * 3;
int i;
```

```
for(i = 0; i < length; i++) {
    // do something with the color value of each pixel
}
```

Notice the difference? The Processing code has one value for each pixel, while the oF code has three because each pixel is split into three parts (red, green, and blue) or four values if the image has an alpha channel (red, green, blue, alpha).

Using Pixels and Bitmaps As Input

What does it mean to use bitmaps as input? It means that each pixel is being analyzed as a piece of data or that each pixel is being analyzed to find patterns, colors, faces, contours, and shapes, which will then be analyzed. Object detection is a very complex topic that attracts many different types of researchers from artists to robotics engineers to researchers working with machine learning. In Chapter 14, computer vision will be discussed in much greater detail. For this chapter, the input possibilities of the bitmap will be explored a little more simply. That said, there are a great number of areas that can be explored.

You can perform simple presence detection by taking an initial frame of an image of a room and comparing it with subsequent frames. A substantial difference in the two frames would imply that someone or something is present in the room or space. There are far more sophisticated ways to do motion detection, but at its simplest, motion detection is really just looking for a group of pixels near one another that have changed substantially in color from one frame to the next.

The tone of the light in a room can tell you what time it is, whether the light in a room is artificial, and where it is in relation to the camera. Analyzing the brightest pixels in a bitmap is another way of using pixel data for creating interactions. If your application runs in a controlled environment, you can predict what the brightest object in your bitmap will be: a window, the sun, a flashlight, a laser. A flashlight or laser can be used like a pointer or a mouse and can become a quite sophisticated user interface. Analyzing color works much the same as analyzing brightness and can be used in interaction in the same way. A block, a paddle, or any object can be tracked throughout the camera frame through color detection. Interfaces using objects that are held by the user are often called *tangible* user interfaces because the user is holding the object that the computer recognizes. Those are both extremely sophisticated projects, but on at a simpler level you can do plenty of things with color or brightness data: create a cursor on the screen, use the position of the object as a dial or potentiometer, create buttons, navigate over lists. As long as the user understands how the data is being gathered and analyzed, you're good to go. In addition to analyzing bitmaps for data, you can simply use a bitmap as part of a conversion process where the bitmap is the input data that will be converted into a novel new data form. Some examples of this are given in the next section of this chapter.

Another interesting issue to consider is that for an application that does not know where it is, bitmap data is an important way of determining where it is, of establishing context. While GPS can provide important information about the geographic location of a device, it doesn't describe the actual context in which the user is using the application. Many mobile phones and laptops now have different affordances that are contextual, such as reading the light in the room to set the brightness of the backlighting on the keyboard, lighting up when they detect sudden movement that indicates that they are about to be used, autoadjusting the camera, and so on. Thinking about bitmap data as more than a picture can help you create more conversational and rich interactions.

Once you move beyond looking at individual bitmaps and begin using arrays of bitmaps, you can begin to determine the amount of change in light or the amount of motion without the great deal of the complex math that is required for more advanced kinds of analysis.

Providing Feedback with Bitmaps

If you're looking to make a purely abstract image, it's often much more efficient to create a vector-based graphic using drawing tools. One notable exception to this is the "physical pixel," that is, some mechanical object that moves or changes based on the pixel value. This can be done using servo motors, solenoid motors, LED matrices, or nearly anything that you can imagine. Chapter 11 contains information about how to design and build such physical systems; however, this chapter focuses more on processing and displaying bitmaps.

Sometimes the need for a video, bitmap, or a photo image in an application is obvious. A mapping application begs for a photo view. Many times, though, the need for a photograph is a little subtler or the nature of the photograph is subtler. Danny Rozins's *Wooden Mirror* is one of the best examples of a photograph that changes our conception of the bitmap, the pixel, and the mirror. In it is a series of mechanical motors that flip small wooden tiles (analogous to pixels in a bitmap) to match an incoming video stream so that the image of the viewer is created in subtle wooden pixels. He has also developed *The Mirrors Mirror*, which has a similar mechanism turning small mirrors. These mirrors act as the pixels of the piece, both reflecting and representing the image data.

Another interesting use of the pixel is Benajmin Gaulon's *PrintBall*, a sort of inkjet printer that uses a paintball gun as the printhead and paintball as ink. The gun uses a mounted servo motor that is controlled by a microcontroller that reads the pixels of an image and fires a paintball onto a wall in the location of the pixel, making a bitmap of brightly colored splashes from the paintball. Though the application simply prints a bitmap, it prints in an interesting way that is physically engaging and interesting.

These works both raise some of the core questions in working with video and images: who are you showing? What are you showing? Are you showing viewers videos of themselves? Who then is watching the video of the viewers? Are you showing them

how they are seen by the computer? How does their movement translate into data? How is that data translated into a movement or an image? Does the user have control over the image? If so, how? What sorts of actions are they are going to be able to take, and how will these actions be organized? Once they are finished editing the image, how will they be able to save it?

So, what is a bitmapped image to you as the designer of an interactive application? It depends on how you approach your interaction and how you conceive the communication between the user and your system. Imagery is a way to convey information, juxtaposing different information sources through layering and transparency. Any weather or mapping application will demonstrate this with data overlaying other data or images highlighting important aspects, as will almost any photo-editing application. With the widespread availability of image-editing tools like Photoshop, the language of editing and the act of modifying images are becoming commonplace enough that the play, the creation of layers, and the tools to manipulate and juxtapose are almost instantly familiar. As with many aspects of interactive applications, the language of the created product and the language of creating that product are blending together. This means that the creation of your imagery, the layering and the transparency, the framing, and even the modular nature of your graphics can be a collaborative process between your users and your system. After all, this is the goal of a truly interactive application.

The data of a bitmap is not all that dissimilar from the data when analyzing sound. In fact, many sound analysis techniques, fast Fourier transforms among one of the more prominent that was discussed in Chapter 7 are used in image analysis as well. This chapter will show you some methods for processing and manipulating the pixels that make up the bitmap data of an image or of a frame of a video.

Looping Through Pixels

In both Processing and oF, you can easily parse through the pixels of an image using the `getPixels()` method of the image. We'll look at Processing first and then oF. The following code loads an image, displays it, and then processes the image, drawing a 20 × 20 pixel rectangle as it loops using the color of each pixel for the fill color of the rectangle:

```
PImage pic;
int location = 0;
int fullSize;

void setup()
{
    pic = loadImage("test.jpg");
    fullSize = pic.height * pic.width;
    size(pic.width, pic.height);
}

void draw()
{
```

```
        background(255, 255, 255);
        image(pic, 0, 0);
        if(location == fullSize) {
            location = 0;
        } else {
              location++;
        }

        fill(pic.pixels[location]);
        int row = location / width;
        int pos = location - (row * width);
        rect(pos, row, 20, 20);
    }
```

This code will work with a single picture only. To work with multiple pictures, you'll want to read the pixels of your application, rather than the pixels of the picture. Before you read the pixels of your application, you'll need to call the `loadPixels()` method. This method loads the pixel data for the display window into the `pixels` array. The `pixels` array is empty before the pixels are loaded, so you'll need to call the `loadPixels()` method before trying to access the `pixels` array. Add the call to the `loadPixels()` method, and change the `fill()` method to read the pixels of the `PApplet` instead of the pixels of the `PImage`:

```
loadPixels();
fill(pixels[location]);
```

Looping through the pixels in oF is done a little differently. In your application, add an `ofImage` and a pointer to an `unsigned` char:

```
#ifndef _PIXEL_READ
#define _PIXEL_READ

#include "ofMain.h"
#define bytesPerPixel = 3

class testApp : public ofBaseApp

    public:
        void setup();
        void update();
        void draw();

        ofImage pic;
        int location;
        int fullSize;
        unsigned char * pixels;
};
#endif
```

In the `setup()` method of your application, get access to the pixels of the image using the `getPixels()` method. The rest of the code is about the same as the Processing version with one exception as mentioned earlier—for each pixel, there are three unsigned char values in the `pixels` array:

```
#include "PixelRead.h"

void testApp::setup()
{
    location = 0;
    pic.loadImage("image_test.jpg");
    fullSize = pic.width * pic.height;
    ofSetWindowShape(pic.width, pic.height);
    pixels = pic.getPixels();
    ofSetVerticalSync(true);
    ofEnableAlphaBlending();

}

void testApp::update(){}

void testApp::draw() {
    ofSetupScreen();
    pic.draw(0,0);

    //location = (mouseY * pic.width) + mouseX; // the interactive version

    if(location == fullSize) { // the noninteractive version
        location = 0;
    } else {
        location++;
    }

    int r = pixels[3 * location];
    int g = pixels[3 * location+1];
    int b = pixels[3 * location+2];

    ofSetColor(r, g, b);

    int col = location % pic.width;
    int row = location / pic.width;

    ofCircle(col, row, 20);
    ofSetColor(0xffffff);
}
```

To grab the pixels of your entire application, create an `ofImage` instance, and then call the `grabScreen()` method to load the pixels from the screen into the image object:

```
void grabScreen(int x, int y, int w, int h);
```

An example call might look like this:

```
int screenWidth = ofGetScreenWidth(); // these should be in setup()
int screenHeight = ofGetScreenHeight();
// this would go in draw
screenImg.grabScreen(0, 0, screenWidth, screenHeight);
```

The `ofGetScreenWidth()` and `ofGetScreenHeight()` methods aren't necessary if you already know the size of the screen, but if you're in full-screen mode and you don't know the size of the screen that your application is being shown on, then it can be helpful.

Manipulating Bitmaps

A common way to change a bitmap is to examine each pixel and modify it according to the value of the pixels around it. You've probably seen a blurring filter or a sharpen filter that brought out the edges of an image. You can create these kinds of effects by examining each pixel and then performing a calculation on the pixels around it according to a convolution kernel. A *convolution kernel* is essentially a fancy name for a matrix. A sample kernel might look like this:

```
.11 .11 .11
.11 8 .11
.11 .11 .11
```

This indicates that each pixel in the list will have this kernel applied to it; all the pixels around the current pixel will be multiplied by 0.11, the current pixel will be multiplied by 8, and the result will be summed. Take a look at Figure 10-3.

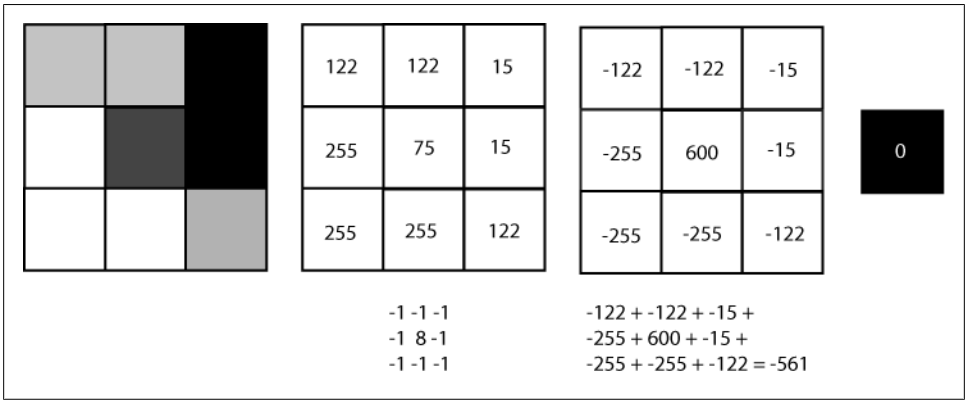


Figure 10-3. Performing an image convolution

On the left is a pixel to which the convolution kernel will be applied. Since determining the final value of a pixel is done by examining all the pixels surrounding the image, the second image shows what the surrounding pixels might look like. The third image shows the grayscale value of each of the nine pixels. Just below that is the convolution kernel that will be applied to each pixel. After multiplying each pixel by the corresponding value in the kernel, the pixels will look like the fourth image. Note that this doesn't actually change the surrounding pixels. This is simply to determine what value will be assigned to the center pixel, the pixel to which the kernel is currently being applied. Each of those values is added together, and the sum is set as the grayscale value of the pixel in the center of the kernel. Since that value is greater than 255, it's rounded down to 255. This has the net result, when applied to an entire image, of leaving only dark pixels that are surrounded by other dark pixels with any color. All the rest of the pixels are changed to white.

Applying the sample convolution kernel to an image produces the effects shown in Figure 10-4.

Figure 10-4. Effect of a convolution filter

Now take a look at the code for applying the convolution kernel to a grayscale image:

```
PImage img;
float[][] kernel = { { .111, .111, .111 }, { .111, 8, .111 },
      { .111, .111, .111 }};

void setup() {
    img  = loadImage("street.jpg"); // Load the original image
    size(img.width, img.height); // size our Processing app to the image
}

void draw() {
img.loadPixels(); // make sure the pixels of the image are available
// create a new empty image that we'll draw into
PImage kerneledImg = createImage(width, height, RGB);
// loop through each pixel
for (int y = 1; y < height-1; y++) { // Skip top and bottom edges
    for (int x = 1; x < width-1; x++) { // Skip left and right edges
        float sum = 0; // Kernel sum for this pixel
        // now loop through each value in the kernel
        for (int kernely = -1; kernely  <= 1; kernely ++) {
            for (int kernelx = -1; kernelx <= 1; kernelx++) {
                // get the neighboring pixel for this value in the
                     // kernel matrix
                int pos = (y + kernely)*width + (x + kernelx);
                // Image is grayscale so red/green/blue
 are identical,                    //it doesn't matter
                float val = red(img.pixels[pos]);
                // Multiply adjacent pixels based on the kernel values
                sum += kernel[kernely+1][kernelx+1] * val;
            }
        }
        // For this pixel in the new image, set the gray value
        // based on the sum from the kernel
        kerneledImg.pixels[y* width + x] = color(sum);
```

```
    }
}
// State that there are changes to edgeImg.pixels[]
edgeImg.updatePixels();
image(edgeImg, 0, 0); // Draw the new image

}
```

This algorithm is essentially the same one used in the book *Processing* by Casey Reas et al. (MIT Press) in their section on image processing. You'll notice that this algorithm works only on grayscale images. Now is the time to take a moment and look at how color data is stored in an integer and how to move that information around.

Manipulating Color Bytes

You'll find yourself manipulating colors in images again and again. In Processing, it is quite easy to create colors from three integer values:

```
int a = 255;
int r = 255;
int g = 39;
int b = 121;
// this only works in Processing. Everything else in this section
  //is applicable to oF and Processing
color c = color(r, g, b);
```

An interesting thing happens if you put the color inside a call to the `hex()` method to convert the color:

```
println(hex(c));
```

You'll see FFFF2779.

You may remember that a hexadecimal number consists of either three or four values from 0 to 255 that are stored in the same integer value in the order alpha, red, green, blue: AARRGGBB.

The value FFFF2779 broken down into its individual values is as follows: FF = alpha, FF = red, 27 = green, 79 = blue.

These values can be written into a single integer value by simply assembling the integer from each value by bit shifting each value into the integer. This means pushing the binary value of each part of the color by using the left shift operator (`<<`). This operator shifts the value on its left side forward by the number of places indicated on the right side. It's a little easier to understand when you see it:

```
int r = 255;
```

In binary, `r` is 0000 0000 0000 0000 0000 0000 1111 1111. Now, shift it to the left by 16 places:

```
r = r << 16
```

In binary, `r` is now 0000 0000 1111 1111 0000 0000 0000 0000. See how the value is shifted over? This is how the color is assembled. Take a look at the way to assemble a color value:

```
int intColor = (a << 24) | (r << 16) | (g << 8) | b;
```

So, what's going on here? Each piece of the color is shifted into place, leaving you with the following binary number: 11111111 11111111 00100111 01111001. That's not so pretty, but it's very quick and easy for your program to calculate, meaning that when you're doing something time intensive like altering each pixel of a large photograph or manipulating video from a live feed, you'll find that these little tricks will make your program run more quickly. So, now that you know how to put a color together, how about taking one apart?

Instead of using the left shift operation, you'll use the right shift operation to break a large number into small pieces:

```
int newA = (intColor >> 24);
int newR = (intColor >> 16) & 0xFF;
int newG = (intColor >> 8) & 0xFF;
int newB = intColor & 0xFF;
```

The AND (`&`) operator compares each value in the binary representation of two integers and returns a new value according to the following scheme: two 1s make 1, 1 and 0 make 0, and two 0s make 0. This is a little easier to understand looking at an example:

11010110 & 01011100 = 01010100

See how each digit is replaced? This is important to do because the `intColor` variable shifted to the right 16 digits is 31231 or 0111 1001 1111 1111. That's not quite what you want. You want only the last eight digits to store in an integer. The easiest way to do this is to AND the value with 255 or 0xFF as it's represented in the earlier code:

0111 1001 1111 1111 & 1111 1111 = 0000 0000 1111 1111

And there you have it: the *red* value of the number. As mentioned earlier, this isn't the easiest way to work with colors and pixels, but it is the fastest by far, and when you're processing real-time images, speed is of the essence if your audience is to perceive those images as being in real time. Note also in the previous code example that the alpha value doesn't have the `&` applied to it. This is because the alpha is the first four digits of the image, so you don't need to mask it to read it correctly.

Using Convolution in Full Color

In the convolution kernels example, the kernels were applied to grayscale images. This means that the image data contained only a single value to indicate the amount of white in each pixel, from 0 or black to 255 or completely white. Working with color images is a little different. If you want to use a convolution kernel on a color image, you would change the `draw()` method of the sample convolution kernel application to do the following:

```
void draw() {
img.loadPixels();
// Create an opaque image of the same size as the original
PImage copyImg = createImage(width, height, RGB);
// Loop through every pixel in the image.
for (int y = 1; y < height-1; y++) { // Skip top and bottom edges
    for (int x = 1; x < width-1; x++) { // Skip left and right edges
```

The major change is here: for each pixel, instead of applying the kernel values to a single value for each pixel, it is applied to three values: red, green, and blue:

```
        int rsum = 0; // red  sum for this pixel
        int gsum = 0; // green  sum for this pixel
        int bsum = 0; // blue sum for this pixel
        for (int ky = -1; ky <= 1; ky++) {
            for (int kx = -1; kx <= 1; kx++) {
                // Calculate the adjacent pixel for this kernel point
                int pos = (y + ky)*width + (x + kx);
```

Just as in the previous grayscale example, the adjacent pixels are multiplied based on the kernel values, but again, in this case since the image is color and has RGB values, the color values of each pixel must be altered as well. Note the bold lines:

```
                int val = img.pixels[pos];
                rsum += kernel[ky+1][kx+1] * ((val >> 16) & 0xFF);
                gsum += kernel[ky+1][kx+1] * ((val >> 8) & 0xFF);
                bsum += kernel[ky+1][kx+1] * (val & 0xFF);
            }
        }
        copyImg.pixels[y* width + x] = color(rsum, gsum, bsum);
    }
}
// State that there are changes to edgeImg.pixels[]
edgeImg.updatePixels();
image(edgeImg, 0, 0); // Draw the new image
}
```

Remember that in Processing a pixel is represented as a single integer with three values in it. To get the red, green, and blue values, simply slice the integer into three different values, one for each color; perform the calculations; and then reassemble the color using the `color()` method. This sort of shifting around to get color values is quite common in oF and Processing.

Analyzing Bitmaps in oF

Bitmap analysis is something that you'll do again and again when programming interactive applications. Chapter 16 will look at analyzing bitmaps using OpenCV for face detection and gesture detection, but this chapter will look at simpler examples.

While these examples show oF code, they are equally as applicable to Processing and with some slight tweaks can be reused. The largest difference between the two is how

they handle the pixel colors. Adapting the example to either oF or Processing involves swapping out method names and changing the way that the colors are processed.

Analyzing Color

Analyzing color in an oF application is a matter of reading each of the three `char` values that contain all the pixel information. The following code (Examples 10-1 and 10-2) draws a simple color histogram, which is a representation of the distribution of colors in an image, showing the amount of each color in an image. While there are 16,581,375 colors that can be drawn using the RGB values, the following spectrograph uses 2,048 values, so that each item in the array represents a range of 8,096 colors.

Example 10-1. histoSample.h

```
#ifndef _HISTO_APP
#define _HISTO_APP

#include "ofMain.h"

class histoSample: public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        ofImage img;
        int pxColors[2048]; //the array that will store all the colors

};

#endif
```

Example 10-2. histoSample.cpp

```
#include "histoSample.h"

void histoSample::setup(){
    bChange = true;
    img.loadImage("cyclists.jpg");
    ofEnableAlphaBlending();
    for(int i = 0; i < 2048; i++) {
        pxColors[i] = 0;
    }

    unsigned char * pixels = img.getPixels();
    int w = img.width;
    int h = img.height;
    int colorVal;
```

Since each pixel is split into three values, to store them as an integer you'll need to shift the values as discussed in the section "Manipulating Color Bytes" on page 347. Once the value of the integer is set, it is added to the `pxColors` array. The amount of each range of colors is represented by the size of each value in `pxColors`:

```
        for (int i = 0; i < h * w * 3; i+=3) {
                colorVal = ( pixels[ i ] << 16) | ( pixels[ i + 1 ] << 8) |
                  pixels[ i + 2 ];
                pxColors[ colorVal / 8096 ] += 1;
        }
    }

    void histoSample::update(){ }
```

Since the `pxColors` array contains integer values, those need to be broken into the individual color values. Those values are then used to set the color that will be drawn into a rectangle:

```
    void histoSample::draw(){
        if(bChange) {
            for(int i = 0; i < 2048; i++) {
                int intColor = i * 8096;
                int newR = (intColor >> 16) & 0xFF;
                int newG = (intColor >> 8) & 0xFF;
                int newB = intColor & 0xFF;
                ofSetColor( newR, newG, newB );
                ofRect( i/2, 0, 2,  pxColors[i] / 4 );
            }
        }
    }
```

Analyzing Brightness

Brightness is another valuable and simple way of analyzing a bitmap. You can easily find the brightest pixel by looping through each pixel grabbed from a video and comparing it to the brightest pixel found (Examples 10-3 and 10-4).

Example 10-3. OFBrightness.h

```
#ifndef _OF_BRIGHTNESS
#define _OF_BRIGHTNESS

#include "ofMain.h"

#define grabbedVidWidth 320
#define grabbedVidHeight 240

class OFBrightness : public ofBaseApp
{
    public:

        void setup();
        void update();
```

```
        void draw();

        unsigned char* drawingPixels;
        int brightestLoc[2];
        ofVideoGrabber videoIn;

};

#endif
```

Example 10-4. OFBrightness.cpp

```
#include "OFBrightness.h"

void OFBrightness::setup(){

    videoIn.initGrabber(grabbedVidWidth ,grabbedVidHeight);
}

void OFBrightness::update(){

    videoIn.grabFrame();

    int brightest = 0;
    int index = 0;

    if (videoIn.isFrameNew()) { //check to make sure the frame is new
        drawingPixels = videoIn.getPixels();
        int length  = grabbedVidWidth*grabbedVidHeight*3;
        for (int i = 0; i < length; i+=3) {
            unsigned char r = drawingPixels[i];
            unsigned char g = drawingPixels[i+1];
            unsigned char b = drawingPixels[i+2];

            if(int(r+g+b) > brightest) {
                brightest = int(r+g+b);
                brightestLoc[0] = (i/3) % grabbedVidWidth;
                brightestLoc[1] = (i/3) / grabbedVidWidth;
            }
        }
    }
}

void OFBrightness::draw(){
    ofSetColor(0xffffff);
    videoIn.draw(0, 0);
    ofEllipse(brightestLoc[0],brightestLoc[1], 10, 10);
}
```

Using the brightest point in an image or camera feed is an easy way to enable a user to create a cursor from a flashlight, an LED laser, or any other light source. If you can control the environment that will be used to generate the pixels, then other kinds of data can be used: most red, darkest, and so on.

Detecting Motion

To detect motion in an oF or Processing application, simply store the pixels of a frame and compare them to the pixels of the next frame. Examples 10-5 and 10-6 are pretty simple. If the difference between the pixels is greater than an arbitrary number (70 in the following examples), then the pixel to be displayed is colored white. This highlights the movement in any frame.

Example 10-5. OFMovement.h

```
#ifndef _TEST_APP
#define _TEST_APP

#include "ofMain.h"

#define GRABBED_VID_WIDTH 320
#define GRABBED_VID_HEIGHT 240

class OFMovement : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        unsigned char drawingPixels[GRABBED_VID_WIDTH * GRABBED_VID_HEIGHT *3];
        unsigned char dataPixels[GRABBED_VID_WIDTH * GRABBED_VID_HEIGHT *3];
        ofVideoGrabber videoIn;
        ofTexture text;
        int totalPixels;

};

#endif
```

Example 10-6. OFMovement.cpp

```
#include "OFMovement.h"


void OFMovement::setup(){
    totalPixels = GRABBED_VID_WIDTH * GRABBED_VID_HEIGHT *3;
    videoIn.initGrabber(GRABBED_VID_WIDTH ,GRABBED_VID_HEIGHT);
    text.allocate(GRABBED_VID_WIDTH, GRABBED_VID_HEIGHT, GL_RGB);
}

void OFMovement::update(){

    videoIn.grabFrame();

    if (videoIn.isFrameNew()){
        int totalPixels = GRABBED_VID_WIDTH*GRABBED_VID_HEIGHT*3;
        unsigned char * tempPixels = videoIn.getPixels();
```

```
for (int i = 0; i < totalPixels; i+=3)
{
    unsigned char r = abs(tempPixels[i] - dataPixels[i]);
    unsigned char g = abs(tempPixels[i+1] - dataPixels[i+1]);
    unsigned char b = abs(tempPixels[i+2] - dataPixels[i+2]);
```

In the next portion of *oFMovement.cpp*, if the difference across all three color components is greater than 70, it sets the pixel to white; otherwise, it uses the pixel from the most recent video frame:

```
        int diff = r+g+b;
        if (diff > 70) {
            drawingPixels[i] = 255;
            drawingPixels[i+1] = 255;
            drawingPixels[i+2] = 255;
        } else {
            drawingPixels[i] = tempPixels[i];
            drawingPixels[i+1] = tempPixels[i+1];
            drawingPixels[i+2] = tempPixels[i+2];
        }

    }
```

Now that all the changed pixels have been colored white, they are loaded into an `ofTexture` object. You'll see a new method in here: `memcpy()`, which is a C++ method that copies data from one location to another. It's a very fast way to copy all the values of one array into another, which is what it's doing here. Here's the signature, followed by the attributes:

```
void * memcpy ( void * destination, const void * source, size_t num );
```

`destination`
: Pointer to the destination array where the content is to be copied

`source`
: Pointer to the source of data to be copied

`Num`
: Number of bytes to copy

The `memcpy()` method copies the values from `source` directly to the memory pointed at by `destination`. The underlying type of the objects pointed by both the source and destination pointers are irrelevant for this function; the result is a binary copy of the data, and it always copies exactly `num` bytes. To avoid errors, the size of the arrays pointed by both the `destination` and `source` parameters should be at least `num` bytes long. This means that if you want to copy an array of 1,000 floats, you should make sure that the array you're copying it into has at least enough space for 1,000 floats. The `num` parameter in this case would be 4,000, because a float is 4 bytes:

```
        memcpy(dataPixels, tempPixels, totalPixels); // copy all the
                                                     //pixels over
        text.loadData(drawingPixels, GRABBED_VID_WIDTH,GRABBED_VID_HEIGHT,
```

```
                GL_RGB);
        }
    }
```

Next, the texture is drawn to the screen:

```
void OFMovement::draw(){
    ofSetColor(0xffffff);
    videoIn.draw(20,20);
    text.draw(20+GRABBED_VID_WIDTH,20,GRABBED_VID_WIDTH,GRABBED_VID_HEIGHT);

}
```

Figure 10-5 shows the application in action.

Figure 10-5. Detecting the changed pixels between images

Using Edge Detection

Another interesting piece of information that pixels can tell us is the location of edges in the image. This verges on the territory that will be covered in Chapter 14, but it's a nice way to think a little more in depth about what you can do when processing pixels.

Edges characterize boundaries between objects, which makes them very interesting and important in image processing. *Edges* in images are areas with strong intensity contrasts, a jump in intensity from one pixel to the next. The goal in edge detection is to reduce the amount of data and filter out any nonedge information from an image while still preserving the important structural properties of an image, which is the basic shape of what is displayed in the image. This is done in much the same way as in the bitmap filtering example. Though the Processing code in that example filtered the image, increasing the contrast, the fundamental way that the operation works is the same: we create a 3 × 3 array that will serve as the kernel, process each pixel according to that kernel, and set the value of the pixel to the result of multiplying it by the kernel.

Two of the most popular edge detection algorithms are called Sobel and Prewitt. These are named after their respective authors, Irwin Sobel and JMS Prewitt. They operate in very similar ways: by looping through each pixel and comparing the amount of change across the surrounding nine pixels in both the x and y directions. Then the algorithm sums the differences and determines whether the change is great enough to be considered an edge. Whichever algorithm you choose, once it has done its job by returning the amount of change around a given pixel, you can decide what you'd like to do with it. In the following example, if the change in a pixel isn't great enough to considered an edge, then it's painted darker by darkening the color values of each pixel. If it is an edge, then it's painted lighter by lightening the pixel. This way, an image where the edges are white and the nonedges are dark can be produced. You are of course free to use these detected images in whatever way you like.

First, here's the EdgeDetect header file for the oF application (Examples 10-7 and 10-8).

Example 10-7. EdgeDetect.h

```
#ifndef _EDGE_APP
#define _EDGE_APP


#include "ofMain.h"

class EdgeDetect : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();
        int setPixel(unsigned char* px, int startPixel, int dep,
              int depthOfNextLine, int depthOfNextNextLine,
              const int matrix[][3]);

        void edgeDetect2D(); // this is for grayscale images,
                                // with only gray pixels
        void edgeDetect3D(); // this is for color images with an RGB

        int sobelHorizontal[3][3]; // here's the sobel kernel
        int sobelVertical[3][3];

        int prewittHorizontal[3][3]; // here's the prewitt kernel
        int prewittVertical[3][3];

        unsigned char* edgeDetectedData;
        ofImage img;
        ofImage newImg;

        bool updateImg;

};

#endif
```

Example 10-8. EdgeDetect.cpp

```
#include "EdgeDetect.h"

void EdgeDetect::setup(){
```

The following code sets up the different kernels that will be used for the edge detection:

```
        sobelHorizontal[0][0] = -1; sobelHorizontal[0][1] = 0;
            sobelHorizontal[0][2]= 1;
        sobelHorizontal[1][0] = -2; sobelHorizontal[1][1] = 0;
            sobelHorizontal[1][2] = 2;
        sobelHorizontal[2][0] = -1; sobelHorizontal[2][1]= 0;
            sobelHorizontal[2][2] = 1;

        sobelVertical[0][0] = 1; sobelVertical[0][1] = 2;
            sobelVertical[0][2]= 1;
        sobelVertical[1][0] = 0; sobelVertical[1][1] = 0;
            sobelVertical[1][2] = 0;
        sobelVertical[2][0] = -1; sobelVertical[2][1]= -2;
            sobelVertical[2][2] = -1;
```

This is the matrix for the Prewitt edge detection that will be used with color images:

```
        prewittHorizontal[0][0] = 1; prewittHorizontal[0][1] = 1;
            prewittHorizontal[0][2]= 1;
        prewittHorizontal[1][0] = 0; prewittHorizontal[1][1] = 0;
            prewittHorizontal[1][2] = 0;
        prewittHorizontal[2][0] = -1; prewittHorizontal[2][1] = -1;
            prewittHorizontal[2][2] = -1;

        prewittVertical[0][0] = 1; prewittVertical[0][1] = 0;
            prewittVertical[0][2]= -1;
        prewittVertical[1][0] = 1; prewittVertical[1][1] = 0;
            prewittVertical[1][2] = -1;
        prewittVertical[2][0] = 1; prewittVertical[2][1] = 0;
            prewittVertical[2][2] = -1;
```

Next, load an image. If you want to use a color image, then you can load the *test.jpg* image that is included with the downloadable code file for this chapter. Otherwise, you can use the grayscale image, *test.bmp*, also included in the code file. The difference between the two is rather important—the BMP image has 1 byte per pixel because each pixel is only a grayscale value, and the JPG image has 3 bytes per pixel because it contains the red, green, and blue channels:

```
    #ifdef USING_COLOR
        img.loadImage("test.jpg");
    #else
        img.loadImage("test.bmp");
    #endif
        // set aside memory for the image
        edgeDetectedData = new unsigned char[img.width * img.height * 3];

        updateImg = true;

    }
```

Here the type of edge detection is set. If the `USING_COLOR` variable is set and the image is color, then you'll want to use the `edgeDetect3D()` method. Otherwise, you'll want to use the `edgeDetect1D()` method:

```
void EdgeDetect::update(){
    ofBackground(255, 255, 255);
    if(updateImg) {
        #ifdef USING_COLOR
        edgeDetect3D();
        #else
        edgeDetect1D();
        #endif
        updateImg = false;
    }
}
```

The simple part is drawing the images, first the original image and then the edge-detected image that's been created from running the edge detection algorithms:

```
void EdgeDetect::draw(){
    img.draw(0, 0);
    newImg.draw(400, 0);
}
```

This is the edge detection for images that are grayscale, that is, that have only one byte per pixel, which is why there's an "1D" at the end of the method name. It loops through each pixel of the image and multiplies it by the Sobel kernel, just like you've seen in the other convolution kernel examples from earlier in this chapter:

```
void EdgeDetect::edgeDetect1D() {
        int xPx, yPx, i, j, sum, sumY, sumX = 0;

        unsigned char* originalImageData = img.getPixels();

        int heightVal = img.getHeight();
        int widthVal = img.getWidth();

        for(yPx=0; yPx < heightVal; yPx++)  {
            for(xPx=0; xPx < widthVal; xPx++)  {
                sumX = 0;
                sumY = 0;
```

If you're analyzing the image boundaries, then just set `sum` to 0, because there won't be surrounding pixels on at least one side to run the algorithm correctly:

```
                if( yPx == 0) {
                    sum = 0;
                } else if( xPx == 0) {
                    sum = 0;
                } else   { // Convolution starts here
```

Here, you find the change on the x-axis and the y-axis by multiplying the value of each pixel around the current pixel being examined by the appropriate value in the Sobel kernel. Since there are two kernels, one for the x-axis and one for the y-axis, there are two `for` loops that loop through each pixel surrounding the current pixel:

```
                for(i=-1; i<=1; i++)  {
                    for(j=-1; j<=1; j++)  {
                        sumX = sumX + (int) originalImageData [(yPx + j) *
                            img.width + xPx + i] *
                            sobelHorizontal[i + 1][j + 1];
                    }
                }
```

Now find the amount of change along the y-axis:

```
                for(i=-1; i<=1; i++)  {
                    for(j=-1; j<=1; j++)  {
                        sumY = sumY + (int) originalImageData[(yPx + j) *
                            img.width + xPx + i] *
                            sobelVertical[i + 1][j + 1];
                   }
                }
                // add them up
                sum = abs(sumX) + abs(sumY);
            }
```

Here the values are *thresholded*; that is, results that are less than 210 are set to 0. This makes the edges appear much more dramatically:

```
            if(sum>255) sum=255;
            if(sum<210) sum=0;
```

Now all the edge-detected pixels are set to white by taking the `sum` value and subtracting it from 255 to make a white pixel out of any values that are 0:

```
            edgeDetectedData[ yPx * img.width + xPx ] =
                  255 - (unsigned char)(sum);
         }
        }
```

Now that the application has loaded the data into the `edgeDetectedData` array, copy the values from `edgeDetectedData` into the `newImg` instance for `ofImage`:

```
        newImg.setFromPixels(edgeDetectedData, img.width,
            img.height, OF_IMAGE_GRAYSCALE, false);
}


void EdgeDetect::edgeDetect3D() {

    unsigned char* imgPixels = img.getPixels();
    unsigned int x,y,nWidth,nHeight;
    int firstPix, secondPix, dwThreshold;
```

Now determine the number of pixels that are contained in a single horizontal line of the image. This will be important because as you run through each pixel, you'll need to know the locations of the pixels around it to use them in calculations:

```
    int horizLength = (nWidth * 3);
    long horizOffset = horizLength - nWidth * 3;
```

```
        nHeight = img.height- 2;
        nWidth  = img.width - 2;
```

Now, as in the `edgeDetect1D()` method, loop through every pixel in the image. Since this method is supposed to be checking all three values of the pixel, R, G, and B, you'll notice there are three different values being set in the `edgeDetectedData` array. The code is a little more compact than the `edgeDetect1D()` method, because the actual multiplication of each surrounding pixel by the appropriate value in the kernel is now in the `setPixel()` method, making things a little tidier:

```
        for( y = 0; y < nHeight;++y) {
            for( x = 0 ; x < nWidth; ++x) {
```

It's important to keep track of where the locations of the pixels around the current pixel are. This code will be the location above the current pixel in the array of pixels. `center` is the value of the pixel currently being calculated, and `below` is the value of the pixel below the pixel being calculated:

```
                long above = (x*3) +(y*horizLength);
                long center = above + horizLength;
                long below = center + horizLength;
```

Next, compare the red values of the pixels:

```
             firstPix = setPixel(imgPixels, 2, above, center, below,
                prewittHorizontal);
            secondPix = setPixel(imgPixels, 2, above, center, below,
                prewittVertical);
            edgeDetectedData[5 + center] = max(dwThreshold,
                min( (int)sqrt(pow(firstPix,2)+pow(secondPix,2)), 255 ));
```

Compare the blue values of the pixels:

```
            firstPix = setPixel(imgPixels, 1, above, center, below,
                prewittHorizontal);
            secondPix = setPixel(imgPixels, 1, above, center, below,
                prewittVertical);
            edgeDetectedData[4 + center] = max(dwThreshold, min( (int)
                sqrt(pow(firstPix,2)+pow(secondPix,2)), 255 ));
```

Compare the green values:

```
            firstPix = setPixel(imgPixels, 1, above, center, below,
                prewittHorizontal);
            secondPix = setPixel(imgPixels, 1, above, center, below,
                prewittVertical);
            edgeDetectedData[3 + center] = max(dwThreshold,min(
               (int) sqrt(pow(firstPix,2)+pow(secondPix,2)), 255 ));

            }
        }
```

Now set the bitmap data for the new image:

```
        newImg.setFromPixels(edgeDetectedData, img.width, img.height,
            OF_IMAGE_COLOR, true);
    }
```

Last up is multiplying the value of the pixel and the pixels around it by the kernel to get the value that the pixel should be set to.

```
int EdgeDetect::setPixel(unsigned char* px, int
 startPixel, int above,      int center, int below, const int matrix[][3]) {
    return (
        (px[startPixel + above] * matrix[0][0]) +
            (px[startPixel + 3 + above] * matrix[0][1]) +
        (px[startPixel + 6 + above] * matrix[0][2]) +
            (px[startPixel + center] * matrix[1][0])+
        (px[startPixel + 3 + center ] * matrix[1][1]) +
            (px[startPixel + 6 +center] * matrix[1][2]) +
        (px[startPixel + below]* matrix[2][0]) +
            (px[startPixel + 3 +below]* matrix[2][1]) +
        (px[startPixel + 6 + below]* matrix[2][2])) ;
}
```

Figure 10-6 shows the result of running the edge detection on a photo.

Figure 10-6. Edge detection results on a photo

The most famous and one of the most accurate edge detection algorithms is called the Canny edge detection algorithm after its inventor John Canny. The code is a little more complex and lengthy than there is room for here but there are easy implementations for Processing and oF that you can find by looking on the Processing and oF websites.

Using Pixel Data

One of the most common things to *do* with pixel data is to mark an object or a location. While we've focused on having the computer locate pixel data, sometimes a human

being does a far better job of finding that *something*. Once you have data about the location of an object, you can use it creatively. A great example of using pixel data is Evan Roth and Ben Engelbrethe's *White Glove Tracking* project. They asked Internet users to help isolate Michael Jackson's white glove in all 10,060 frames of his nationally televised first performance of *Billie Jean*. After only 72 hours, all 125,000 gloves had been located and stored as data that was then released for anyone to use in their own projects. These projects were then collected into an online gallery.

Another way of looking at using pixel data is to use it to generate a sound. Often you see pixel data generated from a sound like in music visualizations and waveforms, but going the other way around can be interesting as well. Sound can be described as a fluctuation of the acoustic pressure in time, while images are spatial distributions of values of luminance or color, the latter being described in its RGB or HSB components. Any signal, in order to be processed by numerical computing devices, has to be reduced to a sequence of discrete samples, and each sample must be represented using a finite number of bits. The first operation is called *sampling*, and the second operation is called *quantization* of the domain of real numbers.

This example uses the ofxSndObj add-on introduced in Chapter 7 to create a buzz tone that reflects the amount of each color in the pixel that the users hovers over with the mouse pointer:

```
#ifndef _PIXELSOUND_APP
#define _PIXELSOUND_APP

#define MACOSX

#include "ofMain.h"
#include "ofxSndObj.h"
#include "AudioDefs.h"


class pixSoundApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        void mouseMoved(int x, int y );
        void exit();
```

Declare all the objects that will generate the sound:

```
        ofxSOBuzz b1;
        ofxSOBuzz b2;
        ofxSOBuzz b3;

        ofxSOADSR adsr1;
        ofxSOADSR adsr2;
        ofxSOADSR adsr3;
```

```
        ofxSOMixer m;
```

Now, declare the image:

```
        ofImage img1;
        unsigned char* imgPixels;

        int pixPlayIndex;
        long currentPixelData[3];
```

Declare the `ofxSndObj` instance that will generate and play back the sound:

```
        ofxSndObj sndobj;
};

#endif
```

Example 10-9. pixSoundApp.cpp

```
#include "pixSoundApp.h"
#include "stdio.h"


void pixSoundApp ::setup(){

    ofHideCursor();

    ofBackground(255,255,255);
    ofSetBackgroundAuto(false);
    img1.loadImage("all.png");
```

Set up the `ofxSndObj` instance:

```
        sndobj.startOut(true, 1);

        b1.init(sndobj, 600, 1, 1);
        b2.init(sndobj, 700, 1, 1);
        b3.init(sndobj, 800, 1, 1);
        m.init(sndobj);
        adsr1.init(sndobj, 1, 1, 1.f, 1, 5, 1.f, b1, OFXSNDOBJBUZZ);
        adsr2.init(sndobj, 1, 1, 1.f, 1, 5, 1.f, b2, OFXSNDOBJBUZZ);
        adsr3.init(sndobj, 1, 1, 1.f, 1, 5, 1.f, b3, OFXSNDOBJBUZZ);
        m.addObject(adsr1, OFXSNDOBJADSR);
        m.addObject(adsr2, OFXSNDOBJADSR);
        m.addObject(adsr3, OFXSNDOBJADSR);
        sndobj.startProcessing();
```

Set the index of the pixel being "played" to 0:

```
        pixPlayIndex = 0;

        imgPixels = img1.getPixels();

    }

    void pixSoundApp ::update() {
```

Get the red, green, blue values:

```
float r = (float)imgPixels[pixPlayIndex];
float g = (float)imgPixels[pixPlayIndex+1];
float b = (float)imgPixels[pixPlayIndex+2];
```

Use the color values to set the volume and frequency of each of the buzz sounds:

```
b1.tune(r*g*b, r*100);
b2.tune(r*g*b, g*100);
b3.tune(r*g*b, b*100);
```

Call `sustain()` to create a slight reverb on the sound:

```
    adsr1.sustain();
    adsr2.sustain();
    adsr3.sustain();
}

void pixSoundApp ::draw() {
    img1.draw(0, 0);
    ofSetColor(0x88FFFFFF);
    ofRect(mouseX, mouseY, 4, 4);
}

void pixSoundApp ::exit() {
    sndobj.cleanup();
}
```

If the mouse is moved, then you can set the location of the pixel being "played" to the pixel underneath the user's mouse pointer:

```
void pixSoundApp ::mouseMoved(int x, int y ) {
    // set the location of the current pixel
    pixPlayIndex = ((img1.width * y) + (x * 3))*3;
}
```

We've been discussing pixels as data, and that extends beyond manipulating the pixels on the screen. Those pixels are valid data to use as physical information as well on controls like an LED matrix. An LED matrix is simply an array of 8 × 8 LED lights. You can build your own matrix, or you can buy a prebuilt one from Sparkfun or Newark suppliers. Controlling an LED matrix requires the use of a controller called the MAX7221. This controller is covered in greater detail in Chapter 11, so in the interests of space, we won't be rehashing all of the information on this controller. Look ahead to the chapter on LED matrices if you'd like. The Arduino code to use the MAX7221 is shown next. It uses the LedControl library that greatly simplifies the code needed to set individual LEDs to display on an LED matrix. We start by including the library:

```
#include "LedControl.h"
```

The LedControl library is initialized by calling its constructor, telling it which pins are connected to the Arduino and the number of the MAX7221 controller attached to the Arduino:

```
    LedControl lc = LedControl(12,11,10,1);
    int incomingRow;
    int incomingColumn;

    void setup() {
        Serial.begin(9600);
    }

    void loop() {
       if (Serial.available() > 1) {
```

If information is being sent, then clear the display by calling the `clearDisplay()` method:

```
          lc.clearDisplay(0);
          // read the incoming byte:
```

Needs to read into an array:

```
          if(Serial.read() == 'x') {
            incomingRow = Serial.read();
            incomingColumn = Serial.read();
            lc.setLed(0, incomingRow, incomingColumn, true);
        }
      }
    }
```

Now take a look at the oF code in Example 10-10.

Example 10-10. LEDMovement.h

```
#ifndef _LED_MOVE
#define _LED_MOVE

#include "ofMain.h"

#define GRABBED_VID_WIDTH 320
#define GRABBED_VID_HEIGHT 240

class LEDMovement : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        int motionVals[64];
        int totalPixels
        unsigned char dataPixels[GRABBED_VID_WIDTH * GRABBED_VID_HEIGHT * 3];
        ofVideoGrabber videoIn;
        ofSerial serial;
};

#endif
```

The oF code is much the same as the motion detection code shown earlier in this chapter. The incoming pixels from the `ofVideoGrabber` instance are compared to the pixels of the previous frame. Since the LED matrix has only 64 LEDs, one way to fix the pixels of the video into those 64 values is to divide the screen into 64 pieces. Create an array of 64 values, and increment the appropriate value in the array if there is movement in that quadrant of the frame (Example 10-11).

Example 10-11. LEDMovement.cpp

```
#include "LEDMovement.h"

void LEDMovement::setup() {

    videoIn.initGrabber(GRABBED_VID_WIDTH ,GRABBED_VID_HEIGHT);
    serial.setup("/dev/tty.usbserial-A4000Qek", 9600);
    totalPixels = GRABBED_VID_WIDTH*GRABBED_VID_HEIGHT*3;
}

void LEDMovement::update() {

    videoIn.grabFrame();

    if (videoIn.isFrameNew()) {
        unsigned char * tempPixels = videoIn.getPixels();
        int i;
        for(i = 0; i < 64; i++) {
            motionVals[i] = 0;
        }

        for (i = 0; i < totalPixels; i+=3) {
            unsigned char r = abs(tempPixels[i] - dataPixels[i]);
            unsigned char g = abs(tempPixels[i+1] - dataPixels[i+1]);
            unsigned char b = abs(tempPixels[i+2] - dataPixels[i+2]);

            int diff = r+g+b;
            if (diff > 70)  {
                motionVals[totalPixels/3600]++;
            }

            tempPixels[i] = dataPixels[i];
            tempPixels[i+1] = dataPixels[i+1];
            tempPixels[i+2] = dataPixels[i+2];

        }
```

For each value in the array, check whether the motion in that quadrant of the frame is above an arbitrary amount (in this case, 100), and if it is, then send it to the serial port for the Arduino to read:

```
            for(i = 0; i < 64; i++)
            {
                if(motionVals[i] > 100) {

                    serial.writeByte(i % 8);
```

```
                serial.writeByte(i / 8);
                serial.writeByte('x');
            }
        }
    }
}

void LEDMovement::draw(){
    ofSetColor(0xffffff);
    videoIn.draw(20,20);
}
```

Another natural physical extension of the pixel is the BlinkM light introduced in Chapter 8. Since the BlinkM can fade from one RGB value to another, it's easy to send those values over the Serial connection from an oF or Processing application. I'm going to leave the creation of the oF or Processing side of this up to you and just show you the Arduino code that will set the BlinkM to the desired color:

```
#define b_addr = 0x00

byte r,g,b; // make some bytes to store the values of the data sent from the oF app

void setup() {
  Serial.begin(9600);
  BlinkM_beginWithPower();
  BlinkM_stopScript( b_addr );  // turn off startup script
}

void loop() {

  if(Serial.available > 3) {
    if(Serial.read() == 'x') {
      r = Serial.read();
      g = Serial.read();
      b = Serial.read();
      BlinkM_fadeToRGB( blinkm_addr, r,g,b );
    }
  }
}
```

This is just the tip of the iceberg, as they say. A few other more playful ideas that come to mind are connecting multiple household lights to an Arduino and turning them on or off based on some pixel data, attaching a servo to a USB web camera and using the motion analysis to turn the camera toward whatever is moving in the frame, and creating a simple system of notation that uses color and edges to play the notes of the musical scale. On a more practical level, pixel analysis is the start of computer vision techniques. What you've learned in this chapter isn't quite enough to begin to develop gestural interfaces that a user interacts with via a touchscreen or simply by using their hands or use marked symbols as UI objects (these are called *fiducials*. It is, however, a start on the road to being able to do that and should give you some ideas as you consider how to create the interaction between your users and your system.

Using Textures

Textures are a way of using your bitmaps in a more dynamic and interesting way, particularly once they're coupled with OpenGL drawing techniques. You've already used textures without knowing it because the `ofImage` class actually contains a texture that is drawn to the screen when you call the `draw()` method. Though it might seem that a texture is just a bitmap, it's actually a little different. Textures are how bitmaps get drawn to the screen; the bitmap is loaded into a texture that then can be used to draw into a shape defined in OpenGL. I've always thought of textures as being like wrapping paper: they don't define the shape of the box, but they do define what you see when you look at the box. Most of the textures that we've looked at so far are used in a very simple way only, sort of like just holding up a square piece of wrapping paper. Now, we'll start to look at some of the more dynamic ways to use that metaphorical paper, curling it up, wrapping boxes, and so on. Textures are very much a part of learning OpenGL, and as such we're going to need to tread lightly to avoid talking too much about things that you'll learn in Chapter 13.

Processing and oF both use textures but in very different ways. oF draws using the Graphics Language Utility Toolkit (GLUT) library, which in turn uses OpenGL. Processing draws to OpenGL optionally, and it uses textures only when you use the OpenGL mode. You set this mode when declaring your window size, which we'll see later. Now, onto the most important part: when you draw in OpenGL, any pixel data that you want to put on the screen must be preloaded into your computer's RAM memory before you can draw it. Loading all this pixel data to your graphic card's RAM is called *loading* your image into a texture, and it's the texture that tells the OpenGL engine on your graphics card how to draw those pixels to the screen.

The Processing `PImage` is a good place to start thinking about textures and bitmaps. The `PImage` is often used to load picture files into a Processing application:

```
PImage myPImage; //allocate space for variable
// allocate space for pixels in ram, decode the jpg, and
// load pixels of the decoded sample.jpg into the pixels.
myPImage = loadImage("sample.jpg");
image(myPImage,100,100); //draw the texture to the screen at 100,100
```

The `PImage` is a texture object that has a built-in color array that holds pixel values so that you can access the individual pixels of the image that you have loaded in. When you call `loadImage()`, you're just pointing to the file, loading the data from that file into an array of bytes, and then turning that array of bytes into a texture that can be drawn to the screen.

Remember how you access the individual pixels of the screen? You first call `loadPixels()`, make your pixel changes, and then call `updatePixels()` to make your changes appear. Although you use a different function altogether, what happens is the same as what happened in the previous Processing application with `PImage`: Processing is loading your pixels from the screen into a texture, essentially a `PImage`, and then

drawing that texture to the screen after you update it. The point here is that you've already been working with textures, those drawable arrays of bitmap data, all along.

The `ofImage` class also has a texture object inside it. The oF version of the previous Processing application code is shown here:

```
ofImage myImage;
// allocate space in ram, then decode the jpg, and finally load the pixels into
// the ofTexture object that the ofImage contains.
myImage.loadImage("sample.jpg");
myImage.draw(100,100);
```

The `ofImage` object loads images from files using `loadImage()` and images from the screen using the `grabScreen()` method. Both of these load data into the internal texture that the `ofImage` class contains. When you call the `draw()` method of the `ofImage` class, you're simply drawing the texture to the screen. If you wanted to change the pixels on the screen, you would also use an `ofImage` class to capture the image and then load the data into an array using the `getPixels()` method. After that, you could manipulate the array and then load it back into the image using `setFromPixels()`:

```
ofImage theScreen; //declare variable
theScreen.grabScreen(0,0,1024,768); //grab at 0,0 a rect of 1024x768.
                                    //similar to loadPixels();
unsigned char * screenPixels = theScreen.getPixels();
//do something here to edit pixels in screenPixels
//...
// now load them back into theScreen
theScreen.setFromPixels(screenPixels, theScreen.width, theScreen.height,
    OF_IMAGE_COLOR, true);
// now you can draw them
theScreen.draw(0,0); //equivalent to the Processing updatePixels();
```

You can edit the pixels of an `ofImage` because `ofImage` objects contain two data structures: an array of `unsigned char` variables that stores all the colors of every pixel in the image and a texture (which is actually an `ofTexture` object, the next thing that we'll discuss) that is used to upload those pixels into the RAM after changes.

Textures in oF

Textures in openFrameworks are contained inside the `ofTexture` object. This can be used to create textures from bitmap data that can then be used to fill other drawn objects, like a bitmap fill on a circle. Though it may seem difficult, earlier examples in this chapter used it without explaining it fully; it's really just a way of storing all the data for a bitmap. If you understand how a bitmap can also be data, that is, be an array of `unsigned` char values, then you basically understand the `ofTexture` already. The `ofTexture` creates data that can be drawn to the screen. A quick tour of the methods of the `ofTexture` class will give you a better idea of how it works:

`void allocate(int w, int h, int internalGlDataType)`
: This method allocates space for the OpenGL texture. The width (`w`) and height (`h`) do not necessarily need to be powers of 2, but they do need to be large enough to contain the data you will upload to the texture. The internal datatype describes how OpenGL will store this texture internally. For example, if you want a grayscale texture, you can use `GL_LUMINANCE`. You can upload whatever type of data you want (using `loadData()`), but internally OpenGL will store the information as grayscale. Other types include `GL_RGB` and `GL_RGBA`.

`void clear()`
: This method clears/frees the texture memory, if something was already allocated. This is useful if you need to control the memory on the graphics card.

`void loadData(unsigned char * data, int w, int h, int glDataType)`
: This method loads the array of unsigned chars (`data`) into the texture, with a given width (`w`) and height (`h`). You also pass in the format that the data is stored in (`GL_LUMINANCE`, `GL_RGB`, `GL_RGBA`). For example, to upload a 200 × 100 pixel wide RGB array into an already allocated texture, you might use the following:

```
unsigned char pixels[200*100*3];
for (int i = 0; i < 200*100*3; i++){
    pixels[i] = (int)(255 * ofRandomuf());
}
myTexture.loadData(pixels, 200, 100, GL_RGB);
```

`void draw(float x, float y, float w, float h)`
: This method draws the texture at a given point (x, y) using a given width and height. This can be used if you simply want to draw the texture as a square. If you want to do something a little more complex, you'll have to use a few OpenGL calls. You'll see in the following application how to draw an `ofTexture` object to an arbitrarily sized shape. This is the first look at OpenGL in this book and it might look a bit strange at first, but the calls are very similar to a lot of the drawing API methods that you've learned in both Processing and oF. Since the point here is to show how the `ofTexture` is used, we won't dwell too much on the GL calls and instead concentrate on the methods of the `ofTexture`. Chapter 13 is entirely dedicated to OpenGL and 3D graphics, so you may want to look ahead to that chapter after you finish this one.

In the header file for this application, there is an `ofTexture` instance and a pointer to pixels that will be used to store data for the texture. Everything else is pretty standard:

```
#ifndef _TEXT_APP
#define _TEXT_APP

#include "ofMain.h"

class TextureApp : public ofBaseApp{

    public:
```

```
        void setup();
        void update();
        void draw();
        void mouseMoved(int x, int y );

        ofTexture colorTexture;
        int             w, h;
        unsigned char     * colorPixels;

};

#endif
```

Example 10-12. TextureApp.cpp

```
#include "TextureApp.h"

void TextureApp::setup(){

    w = 250;
    h = 250;
    colorTexture.allocate(w,h,GL_RGB);
    colorPixels = new unsigned char [w*h*3];

    // color pixels, use w and h to control red and green
    for (int i = 0; i < w; i++){
        for (int j = 0; j < h; j++){
            colorPixels[(j * w+i) * 3 + 0] = i;    // r
            colorPixels[(j * w+i) * 3 + 1] = j;    // g
            colorPixels[(j * w+i) * 3 + 2] = 0; // b
        }
    }
    // this is important, we load the data into the texture,
    // preparing it to be used by the OpenGL renderer
    colorTexture.loadData(colorPixels, w,h, GL_RGB);
}


void TextureApp::update(){}

void TextureApp::draw(){
    ofBackground(255,255,255);
```

The call to `glEnable()` tells the graphics card to enable some specific functionality; in this case, we're enabling the target of the texture. This essentially ensures that your texture is enabled and ready to be used:

```
glEnable(colorTexture.getTextureTarget());
```

The `glBegin()` method sets what kind of shape the graphics engine should be placing in between all the points that are listed. Passing the `GL_QUADS` constant to `glBegin()` indicates that you want to draw a quadrilateral shape with all the points that you set using the `glVertex3i()` call. Some of the other commonly used values are `GL_TRIANGLES` and `GL_POLYGONS`. The `glBegin()` and `glEnd()` methods delimit the vertices

that define a primitive or a group of like primitives. `glBegin()` accepts a single argument that specifies in which of 10 ways the vertices are interpreted. There's going to be much more information on this in Chapter 13, so to save space, for the time being understand that passing `GL_QUADS` to the `glBegin()` method draws a rectangular shape:

```
        glBegin( GL_QUADS );
```

Now, call `glBindTexture()` to begin drawing with the texture and then place the points that the quadrilateral should be drawn using:

```
            glBindTexture (colorTexture.getTextureData().textureTarget,
    colorTexture.getTextureData().textureTarget);
```

Here, points for where the texture should be mapped to and the points for the shape that the texture will be drawn into are set. These points are drawn clockwise, from the upper-left of the screen to the x and y positions of the mouse pointer:

```
            glTexCoord2f(0,  0); // set a point for the texture to be drawn to
                glVertex3i(0, 0, 0); // set a point for the quad to be drawn to
            glTexCoord2f(mouseX, 0);
                glVertex3i(mouseX, 0, 0);
            glTexCoord2f(mouseX, mouseY);
                glVertex3i(mouseX, mouseY, 0);
            glTexCoord2f(0, mouseY);
                glVertex3i(0, mouseY, 0);
```

Now, call `glEnd()` to stop drawing:

```
        glEnd();
    }
```

The `mouseMoved()` method updates all the values in the `colorPixels` array to make the texture react to the user's mouse movements:

```
    void TextureApp::mouseMoved(int x, int y ){

        // when the mouse moves, we change the color image:
        float pct = (float)x / (float)ofGetWidth();
        for (int i = 0; i < w; i++){
            for (int j = 0; j < h; j++){
                colorPixels[(j * w + i) * 3 + 0] = i;    // r
                colorPixels[(j * w + i) * 3 + 1] = j;    // g
                colorPixels[(j * w + i) * 3 + 2] = (unsigned char)(pct*255); // b
            }
        }
        // finally, load those pixels into the texture
        colorTexture.loadData(colorPixels, w,h, GL_RGB);
    }
```

In Chapter 13, you'll learn a lot more about some of the OpenGL calls that were used in this sample code, so for the moment we'll move on to covering textures in Processing.

Textures in Processing

In Processing, when you use the OpenGL mode, a texture is stored inside a `PImage` object. To draw the `PImage` to the screen, either you can use the `image()` method to draw the `PImage` directly to the screen or you can use the `PImage` as a texture for drawing using the `fill()` and `vertex()` methods. There are five important methods that you need to understand to draw with a texture:

`size(400, 400, P3D);`
: When you call the `size()` method to size the application stage, you need to pass three parameters. The first two are the dimensions, and the third parameter is the P3D constant that tells the `PApplet` to use a 3D renderer. That means that all the calls to `vertex()` and `fill()` need to reflect that they are being drawn into a three-dimensional space rather than a two-dimensional space.

`textureMode(NORMALIZED);`
: This method sets the coordinate space for how the texture will be mapped to whatever shape it's drawn onto. There are two options: `IMAGE`, which refers to the actual coordinates of the image being used to create the image, and `NORMALIZED`, which refers to a normalized space of values ranging from 0 to 1. This is very relevant to how the `vertex()` method is used because the fourth and fifth parameters are the locations of the texture that should be mapped to the location of the vertex. If you're using the `IMAGE textureMode()` method, then the bottom-right corner of a 200 × 100 pixel texture would be 200, 100. If you're using `NORMALIZED`, then the bottom-right corner of the same texture would be 1.0, 1.0.

`texture(PImage t);`
: This method sets a texture to be applied to vertex points. The `texture()` function must be called between `beginShape()` and `endShape()` and before any calls to `vertex()`. When textures are in use, the fill color is ignored. Instead, use `tint()` to specify the color of the texture as it is applied to the shape.

`beginShape(MODE);`
: This shape begins recording vertices for a shape, and `endShape()` stops recording. The value of the `MODE` parameter tells it which types of shapes to create from the provided vertices, the possible values are `POINTS`, `LINES`, `TRIANGLES`, `TRIANGLE_FAN`, `TRIANGLE_STRIP`, `QUADS`, and `QUAD_STRIP`. The upcoming example uses the `QUADS` mode, and the rest of these modes will be discussed in Chapter 13. For the moment, you just need to understand that every vertex created by a call to `vertex()` will create a quadrilateral.

`vertex(x, y, z, u, v);`
: All shapes are constructed by connecting a series of vertices. The method `vertex()` is used to specify the vertex coordinates for points, lines, triangles, quads, and polygons and is used exclusively within the `beginShape()` and `endShape()` functions. The first three parameters are the position of the vertex, and the last two indicate the horizontal and vertical coordinates for the texture. You can think of

this as being where the edge of the texture should be set to go. It can be larger than the vertex or smaller, but this will cause it to be clipped if it's greater than the location of the vertex. One way to think of this is like a tablecloth on a table. The cloth can be longer than the table or smaller, but if it's longer, then it will simply drape off the end, and if it's shorter, then it will end before the table.

In the following code, an image is used as a texture and followed by a color interpolation and the default illumination. The shading of the surfaces, produced by means of the illumination and the colors, is modulated in a multiplicative way by the colors of the texture:

```
PImage a;

void setup() {
    size(400,400,P3D);
    a = loadImage("test2.png");
}

void draw() {
    background(255);
    textureMode(NORMALIZED);
    beginShape(QUADS);
    texture(a);
        vertex(0, 0, 0, 0, 0);
        vertex(200, 0, 0, 0, 100);
        vertex(200, 200, 0, 100, 100);
        vertex(0, 200, 0, 100, 100);
    endShape();
    beginShape();
        texture(a);
        vertex(100, 0, 0, 0, 0);
        vertex(mouseX, 0, 0, 100, 0);
        vertex(mouseX, 200, 0, 100, 100);
        vertex(100, 200, 0, 0, 100);
    endShape();
}
```

In addition to generating textures from JPG files, you can also generate a texture. For example, the pattern of a chessboard can be generated by creating a `PImage`, filling it with an empty image, and then setting the value of each pixel in the image:

```
PImage textureImg;
void setup() {

    size(300, 300);
    // dummy image colorMode(RGB,1);
    textureImg = loadImage("test.png");

    color col;
    int squareWidth = 20;
    boolean isBlack = true;
```

Look through each pixel of the image and use the area of each square on the chessboard to determine what color the pixel should be colored. If the square being currently drawn is black, then when the end of the square is reached, switch to white, and so on, until the edge of the image:

```
        for( int i = 0; i < textureImg.height; i++) {
            for ( int j = 0; j < textureImg.width; j++) {
                if(j % squareWidth == 0) {
                    isBlack = !isBlack;
                }
                if(isBlack) {
            textureImg.pixels[ i * textureImg.width + j] = color(255);
                } else {
                    textureImg.pixels[ i * textureImg.width + j] = color(0);
                  }
            }
            if(i % squareWidth != 0 && i != 0) {
                   isBlack = !isBlack;
            }
        }
    }

    void draw() {
      image(textureImg, 0, 0);
    }
```

Saving a Bitmap

You've learned how to load bitmaps, now take a look at saving them. In oF, saving an image is handled through the `ofImage` class. The `saveImage()` method allows you to write a file by default to the data folder of your oF application, though you can write it anywhere else:

```
void saveImage(string fileName);
```

This means that you can also take a screenshot and then save the image by calling the `grabScreen()` method and then calling the `saveImage()` method. You can save in all the common file formats, and if you try to save to format that oF doesn't understand, then it will be saved as a BMP file.

In Processing the same thing can be accomplished by calling the `save()` method, as shown here in the `keyPressed()` handler of an application:

```
void keyPressed() {
   if(key == ENTER) {
       save("file.png");
   }
}
```

Images are saved in TIFF, TARGA, JPEG, and PNG format depending on the extension within the filename parameter. If no extension is included in the filename, the image will save in TIFF format, and *.tif* will be added to the name. These files are saved to the sketch's folder, which may be opened by selecting "Show sketch folder" from the Sketch menu. You can't call `save()` while running the program in a web browser. All images saved from the main drawing window will have an opaque background; however, you can save images without a background by using the `createGraphics()` method:

```
createGraphics(width, height, renderer, filename);
```

The `filename` parameter is optional and depends on the renderer that is used. The renderers that Processing can use with the `createGraphics()` methods that *don't* require a filename are the P2D, P3D, and JAVA2D. There's more information on these different renderers in Chapter 13. The DXF renderer for DXF files and PDF renderer for creating PDF files both require the `filename` parameter. It's not possible to use `createGraphics()` with OPENGL, because it doesn't allow offscreen use. Unlike the main drawing surface, which is completely opaque, surfaces created with `createGraphics()` can have transparency. So, you can use `save()` to write a PNG or a TGA file and the transparency of the graphics that you create will translate to the saved file. Note, though, that with transparency, it's either opaque or transparent, there's no half transparency for saved images.

What's Next

Now that you've learned some about textures and about how your computer creates and interprets graphics, a logical next step is to look ahead to Chapter 13 on OpenGL. If you're particularly interested in learning more about image processing, motion detection, or face detection, look ahead to Chapter 14. You can also, of course, continue reading the book straight through as well.

For more image processing examples, take a look at some books such as Casey Reas's and Ben Fry's *Processing* (MIT Press), Daniel Shiffman's *Learning Processing* (Morgan Kaufmann) and and *The Nature of Code* (*http://www.shiffman.net/teaching/nature/*), and Ira Greenberg's *Processing: Creative Coding and Computational Art* (Springer). There aren't a great deal of introductory-level image processing or signal processing texts out there, but *Practical Algorithms for Image Analysis* by Lawrence O'Gorman et al. (Cambridge University Press) is the closest thing I've found. Be advised, though, it isn't a simple or quick read. There are also several websites that offer tutorials on image processing that are of varying quality. Some are worth a perusal. If you search for *image processing basics*, you'll come across more than one. Once you've figured out what you want to do, you'll find a wealth of information online that can help, from code snippets to algorithms to full-featured toolkits that you can put to work for yourself.

If you're looking for ideas or thoughts on how to design with images, *Graphic Design: The New Basics* by Ellen Lupton and Phillips Jennifer Cole (Princeton Architectural Press) is an excellent text for ideas. *Design Elements: A Graphic Style Manual* by Timothy Samara (Rockport) is also worth checking out. Both of these books will give you ideas for how to frame, layer, and present video and images.

Review

All videos and photo are bitmaps once they have been read into the memory of the computer. You can read the pixel values of those bitmaps to alter them, analyze them, and save them to the user's computer.

The Processing `PImage` can actually be a texture object that has a built-in color array that holds pixel values so that you can access the individual pixels of the image that you have loaded. Images cannot draw themselves using a `draw()` method, like in oF, but they can be drawn by the `image()` function.

In Processing, if you want to access the individual pixels of the screen call `loadPixels()`, make your pixel changes, and then call `updatePixels()` to make your changes appear.

In openFrameworks, any image file can be loaded into the application by creating an `ofImage` object and calling the `loadImage()` method of that object, passing the name of the file to be loaded to the method.

To read the pixels from an `ofImage` object, call the `getPixels()` method. This will return an array of unsigned char values, with one value for each color component and, if the image has alpha values, the alpha value of each pixel. This means that for a 100 × 100 pixel image without an alpha channel, there will be 30,000 values, while an image with an alpha value will have 40,000 values.

In an oF application you can read video from a camera attached to your computer by creating an `ofVideoGrabber` object and calling the `initGrabber()` method on it. To draw the pixels of the video to the screen, use the `draw()` method. To retrieve the pixels from the camera, call the `getPixels()` method.

A convolution kernel is a 3 × 3 matrix used to calculate how to change each pixel in an image to achieve some kinds of spatial effects on the bitmaps.

Motion detection can be done by storing each incoming bitmap of data from a video stream and comparing the next incoming frame to the previous frame. A difference of more than an arbitrary value indicates movement in that pixel of the frame.

You can do simple edge detection by creating convolution kernels and applying them to a bitmap. Two of the more well-known edge detection algorithms are known as the Sobel and the Prewitt algorithms.

Textures in Processing are created and altered using the `PImage` class. You can load an image into the `PImage`, draw it to the screen, alter its pixels, and use it as a texture for OpenGL drawing.

Textures in openFrameworks are handled by the `ofTexture` class, which contains methods for loading bitmap data into a drawing-ready texture, for copying bitmap data into textures, and for drawing the pixels to the screen.

Saving images is done in Processing by simply calling the `save()` method. In oF, the `saveFile()` method of the `ofImage` class enables you to save images.

CHAPTER 11
Physical Feedback

Although there are projects that allow us to alter an aspect of reality as it confronts us (the *Parallel Tracking and Mapping* project at the Oxford Robots Lab or the ARToolkit library come to mind), nothing can replace the richness of the actual physical sensation of the world around us. This includes, of course, both the sound and the images of the world around us, as discussed in earlier chapters, but too often the easily manipulated senses are the only ones that interaction art and design addresses. Any industrial or product designer, architect, or sculptor can tell you about the emotional and cognitive effects that the physical product has upon us. Objects that are carried, lifted, or felt; that change the nature of space around us; or that let us change the nature of the space around us engage us differently than those that do not.

When you're thinking about creating physical feedback, you're thinking about using mechanics and about creating machines. More than any other topic in this book, physical feedback involves thinking about and using the practices of mechanical and electrical engineering and coupling them with programming. Luckily for you, the Arduino community on the bulletin boards is very knowledgeable and helpful in answering questions about these sorts of topics, and a host of other resources are available. This does require a slightly different approach and way of thinking, though, because you'll need to follow along quite closely with electrical diagrams as well as copying down code. Debugging circuits is usually done using a voltmeter or, if you're really dedicated or lucky, an oscilloscope. It's a much different process than coding, but it has some rich rewards: enabling you to create things that move and that act in the world.

So, from the perspective of designing an interaction, how do you create good physical interaction? The rules are pretty much the same: spectrum feedback mechanisms usually should correlate to spectrum inputs, and binary feedback should correlate to binary input. User input that changes a state leads a user to expect a changed state. Thinking about this in terms of the physical environment that one or more viewers inhabit presents interesting challenges and wonderful opportunities to engage. You can explore a whole new range of senses and kinds of thinking with physical feedback, though. The feel of something, its actual sensation, or the changing of the physical space around us are all quite different sensory experiences than looking at a screen or hearing a sound.

The physical nature of what a user is doing is going to shape drastically their perception of what should be happening, so the digital states of a system or an application and the physical states that the user perceives should be kept synchronized when possible.

When you design a system that presents physical feedback, it is necessary to understand and plan around every possible aspect of the physical environment of the user. The vibrate setting on my cell phone works because I keep my cell phone in my pocket. Just having a blinking light to indicate to me that my turn signal is on in my car doesn't work because I might not be looking at the dashboard. Coupled with a clicking sound, I notice it.

Innumerable tools, from backhoes to bombs, alter our physical environment. Their actions in the environment are not intended to help a viewer or participant understand something that has occurred, to give or request information from them, or to further an exchange. Simple tools alter the world as the result of an action. Although the design and construction of such tools is fascinating, this chapter will focus on tools that use physical feedback to further an interaction or enable communication between a system and a user.

There is another element to some of the tools that we'll be touching on in this chapter that takes the discussion outside the realm of this book, but it's an important one nonetheless. Lots of times in physical computing there is no user, or the user really doesn't matter much. That's not often true with any of the other themes of interaction design that we've discussed so far, but it can be true with physical machines. Sometimes all the user does is watch or press a button to start a long process. That's not the focus of this book, but it is a reality in physical computing, so it's worth mentioning: lots of times the design of machines is pretty nonpeople-friendly, because the machine isn't going to be interacting with people very much. However, in this chapter, we're going to be looking at things with the assumption that someone will be actively operating or experiencing them.

Using Motors

The motor is the primary tool for creating motion. At its simplest, you can use a motor to make something spin. With a little more mechanical work, using gears and other mechanical devices, you can use a motor to make something move, vibrate, rise, fall, roll, creep, or perform almost any other type of motion that does not require precise positioning. There are several different kinds of motors: servos, stepper motors, or unidirectional DC motors. In this chapter, you'll look at all three kinds and how you can use them. The simplest motors are good for designs that need motion forward or backward, like a remote control car or a fan, but not for things that need to move to a precise position, like a robotic arm or anything that points or moves something to a controlled position.

Motors are all around us; just look inside moving toys, and you'll find a number of excellent motors and gears you can repurpose if you'd like. Any electronics supplier will have a wide range of motors that will suit many purposes from spinning small objects to driving large loads. In Figure 11-1, you'll see two small motors that can be controlled by an Arduino controller and a diagram that shows how the internals of a brushed DC electric motor work.

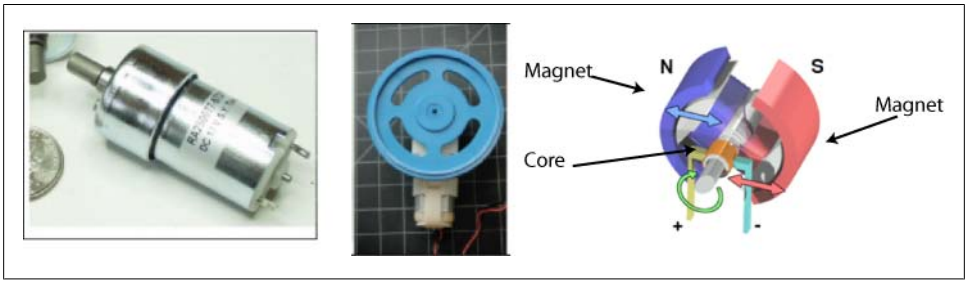


Figure 11-1. Two small motor examples and a cutaway of another

DC Motors

In a motor, when current passes through the coil wound around the core, the side of the positive pole is acted upon by an upward force, while the other side is acted upon by a downward force. This creates a turning effect on the coil, making it rotate. To make the motor rotate in a constant direction, the current reverses every half a cycle, which ensures that the motor rotates in the same direction. To work through the example in this section, you'll need a DC motor that runs on low-voltage DC, somewhere between 5V and 15V.

This part of the example reverses the direction that the motor spins. To reverse a DC motor, you need to be able to reverse the direction of the current in the motor. The easiest way to do this is using an H-Bridge circuit. There are many different models and brands of H-Bridge. This example uses one of the most basic, an L293E from Texas Instruments (a similar chip is the SN754410). If you simply want to turn a motor on and off and don't need to reverse it—for example, if you're controlling a fan—you can simply control the motor by controlling the current sent to the motor.

The example in Figure 11-2 uses an H-Bridge integrated circuit, the Texas Instruments L293E. You can find these chips or equivalent at most large electronics suppliers. This particular chip can control two motors. It can drive up to 1 amp of current and between 4.5 and 36V. If your motor needs more than that, you'll need to use a more powerful H-Bridge.

There's another component included in this circuit that you'll need to understand: the capacitor. A *capacitor* is a pair of conductors (or plates) separated by an insulator (Figure 11-3), and it stores energy as an electrostatic field between the plates. *Capacitance* is a capacitor's ability to store that energy. The basic unit of capacitance

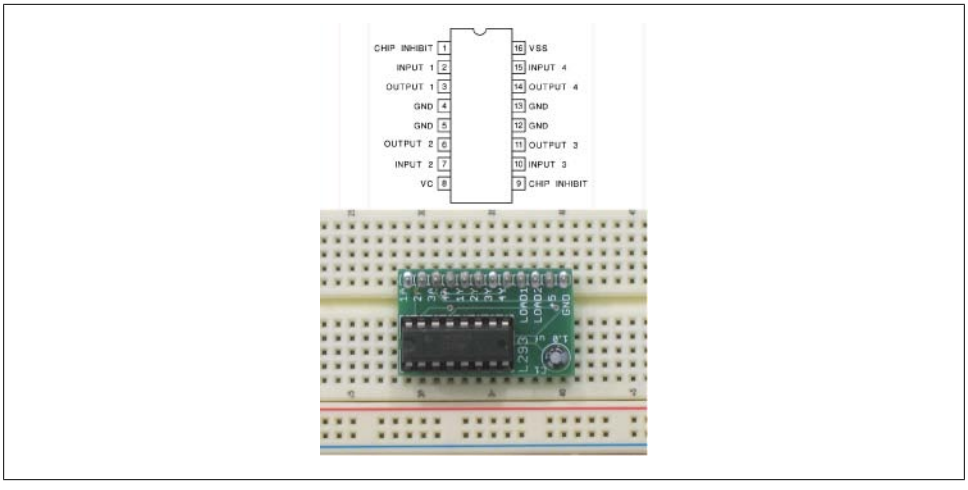


Figure 11-2. The pinout and image of the 1293 H-Bridge chip

Figure 11-3. The electrical symbol for a capacitor and a representation of a capacitor

is the farad (F). So, you'll see capacitors marked as 10 microFarads, or μF, as in Figure 11-4. Usually there are slight swings in the amount of current passed around a circuit. Sometimes variations in the voltages of these circuits can cause problems; if the voltages swing too much, the circuit may operate incorrectly. In the case of the motor in Figure 11-3, the capacitor smoothes the voltage spikes and dips that occur as the motor turns on and off.

Now you're ready for the wiring diagram in Figure 11-4.

Let's break the wiring diagram down a little bit. The switch connected to the Arduino pin 2 controls which direction the motor will turn. When the switch is high or on, Arduino pin 3 will send a low signal, and pin 4 will send a high signal. This tells the H-Bridge to send the motor one direction. When the switch is low, the Arduino pin 3 will send a high signal, and pin 4 will send a low signal, telling the H-Bridge to send the motor in the other direction:

```
int switchPin = 2;    // switch input
int motorPin1 = 3;    // H-bridge leg 1
int motorPin2 = 4;    // H-bridge leg 2
int speedPin = 9;     // H-bridge enable pin
int ledPin = 13;      //LED
```

```
void setup() {
    // set the switch as an input:
    pinMode(switchPin, INPUT);

    // set all the other pins you're using as outputs:
    pinMode(motorPin1 , OUTPUT);
    pinMode(motorPin2 , OUTPUT);
    pinMode(speedPin, OUTPUT);
    pinMode(ledPin, OUTPUT);

    // set speedPin high so that motor can turn on:
    digitalWrite(speedPin, HIGH);

    // this is borrowed from Tom Igoe and is a nice way to debug
    // your circuit, if the light blinks 3 times after the initial
    // startup that's probably an indication that the Arduino has reset itself
    // because the motor caused a short
    blinkLED(ledPin, 3, 100);
}

void loop() {
```

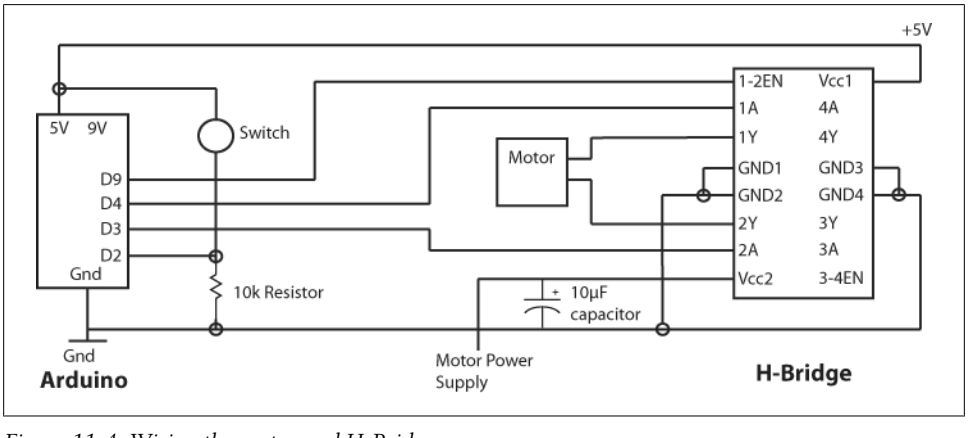


Figure 11-4. Wiring the motor and H-Bridge

If the switch is set to high, the motor will turn in one direction:

```
    if (digitalRead(switchPin) == HIGH) {
        digitalWrite(motorPin1, LOW);   // set 1A on the H-bridge low
        digitalWrite(motorPin2, HIGH);  // set 2A on the H-bridge high
    }
```

If the switch is low, the motor will turn in the other direction:

```
    else {
        digitalWrite(motorPin1, HIGH);  // set 1A on the H-bridge high
        digitalWrite(motorPin2, LOW);   // set 2A on the H-bridge low
    }
}
```

```
// this method just blinks an LED
void blinkLED(int whatPin, int howManyTimes, int milliSecs) {
    int i = 0;
    for ( i = 0; i < howManyTimes; i++) {
        digitalWrite(whatPin, HIGH);
        delay(milliSecs/2);
        digitalWrite(whatPin, LOW);
        delay(milliSecs/2);
    }
}
```

Now that you know how to wire a motor, you can begin to use a motor to create motion in any of your projects. If you're looking to use a motor to drive a more complex motion, you'll probably need to use some gears or wheels in different kinds of configurations. A number of websites worldwide sell tracks, wheels, and gear and axle kits.

Stepper Motors

Stepper motors operate differently from normal DC motors. Instead of just rotating when voltage is applied to their terminals, they have multiple toothed electromagnets that are given power by an external control. To make the motor shaft turn, first one electromagnet is given power, which makes the gear's teeth magnetically attracted to the electromagnet's teeth. When the gear's teeth are thus aligned to the first electromagnet, they are slightly offset from the next electromagnet. When the next electromagnet is turned on and the first is turned off, the gear rotates slightly to align with the next, and so on, all the way around the gear. Each rotation is called a *step*, with an integral number of steps making a full rotation that lets you set the position of the motor to a precise angle.

To work with a Stepper motor and an Arduino, you'll probably need to use a chip to control a stepper, called a *driver*, like the one pictured in Figure 11-5.

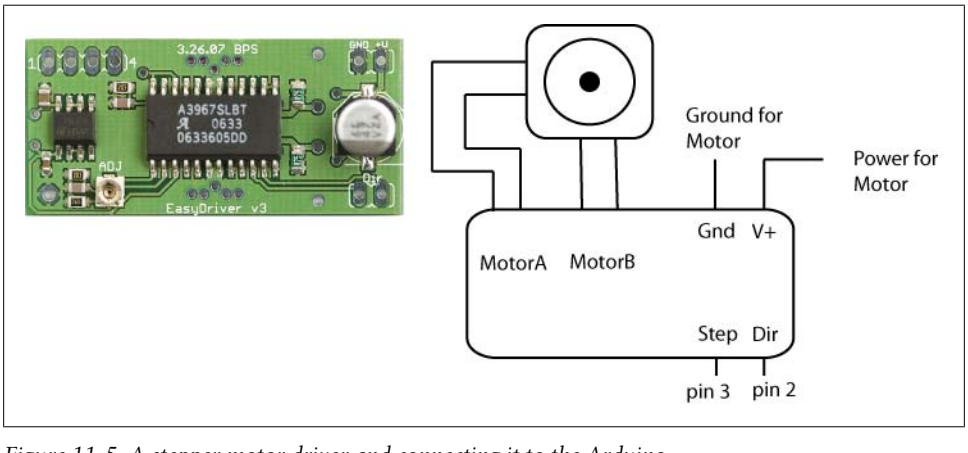


Figure 11-5. A stepper motor driver and connecting it to the Arduino

The driver lets the Arduino control the motion of the motor with pin 3 connected to the Step input as marked on the diagram and pin 2 connected to the `Dir` input as marked in the diagram. To power your motor, you'll send the current to the pin on the EasyDriver marked +V. How much current you'll need to send is dependent on the motor that you're using, so check the datasheet for your motor.

The Arduino code to control a stepper motor looks like this:

```
int dirPin = 2;
int stepperPin = 3;

void setup() {
    pinMode(dirPin, OUTPUT);
    pinMode(stepperPin, OUTPUT);
}

void step(boolean dir,int steps){
    digitalWrite(dirPin,dir);
    delay(50);
    for(int i=0;i<steps;i++){
        digitalWrite(stepperPin, HIGH);
        delayMicroseconds(100);
        digitalWrite(stepperPin, LOW);
        delayMicroseconds(100);
    }
}

void loop(){
    step(true,1600);
    delay(500);
    step(false,1600*5);
    delay(500);
}
```

Another option is to use the Stepper library that comes with the Arduino IDE distribution. You can find examples of using this library both on the Arduino site and on the MAKE magazine site at makezine.com. Anything that you need to move a very controlled amount, such as a camera, a light, a small door, a slide for a slide whistle, can all be controlled with a stepper motor.

In the next section, we'll look at servo motors, which are quite similar but have a few differences from a stepper motor. A servo is much smoother and better for continuous motion, while stepper motors can much more easily lock into fixed positions and can hold those fixed positions with much greater torque than a servo, making them more appropriate for moving into preset positions. You can see stepper motors at work in printers, laser cutters, tool and die machines of all flavors, as well as throughout industrial production lines.

Other Options

Another option for working with stepper motors is a motor shield from Adafruit Industries, shown on the right in Figure 11-6. As of the writing of this book, the motor shield supports two smaller (5V) servos, up to four bidirectional DC motors, and two stepper motors, as well as providing a block of connectors to easily hook up wires (10-22AWG) and power. These are worth looking into because they simplify working with multiple motors.

Pololu is a robotics supply company, which is also making motor drivers, as shown on the left of Figure 11-6, that can be used to drive stepper and bidirectional motors. While the boards made by Pololu are more versatile, the AdaFruit shield is much more Ardunio-ready. If any of these different tools is appropriate for your project, that is dependent of course on how much you can spend.

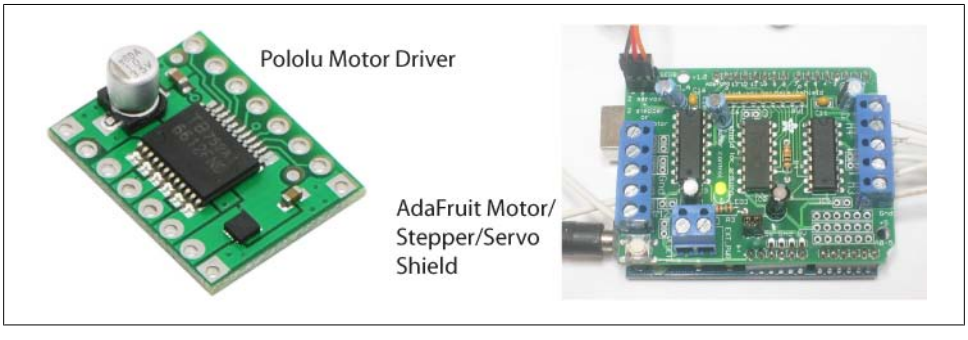


Figure 11-6. Two different motor driver shields

Using Servos

The motor is not, on the face of it, an interactive element; it is a reactive element. For example, I turn the key, and my car turns on. But motors are important and relevant to interaction and interactivity because working with motors and servos allows you to make the first step into the one of the most important areas of human computer interaction and computing: robotics. Now, given the space constraints that I frequently mention, it's impossible for us to delve too deeply into robotics, but it is quite easy to show some of the basics of controlling a servo and creating *programmable motion*.

A servo is a type of motor but with a few added advantages. It, like the motor, receives a PWM signal that it uses to power itself; however, a servo reads the amount of voltage passed in to determine how far to rotate within a limited range. The most common range for a servo motor is 180 degrees, and the servo can move from any point in that range to any other point clockwise or counterclockwise, making it a powerful tool for controlling anything that needs a controllable, repeatable movement. Robots, puppets, cars, and in fact most automated machinery make extensive use of servos. For our

purposes in this chapter, we'll be looking at small servos, but you are encouraged to think big. One of the really wonderful aspects of servos is that though there are many to choose from with different sizes and performance, the signals used to control the servo are almost always the same, meaning that you could apply the same code to move a small hobbyist servo as you would to move a very large servo.

Now, although this won't quite make sense until we look at code to control the servo, it's worth mentioning here that the PWM pin on the Arduino control isn't set at the correct frequency to control a servo motor. That means that we can't write simple instructions to tell a servo to move to a certain position. We have to send the servo a 5V pulse where the duration of the pulse tells the servo to move to a given position. This isn't as difficult as it sounds, although it does make our code a little strange. But don't worry, because it will be explained in full.

Connecting a Servo

To connect a servo motor to the Arduino controller, you need to provide the servo with power from the 5V pin on the Arduino controller, to ground the servo, and to send commands to the servo PWM port. Now, as we mentioned, the PWM port of the Arduino controller doesn't operate at the proper frequency to communicate with a servo. If you're savvy with AVR microcontrollers, the chip that the Arduino uses, and C programming, you can change it, but that's beyond the scope of this book and isn't necessary to communicate with the servo.

Communicating with the Servo

To control the servo, you send it a command every 20 milliseconds. The servo responds to a short pulse (1ms or less) by moving to one extreme, and it moves around 180 degrees in the other direction when it receives long pulses (around 2ms or more). The amount that the servo moves is determined by the length of time that the pulse is sent to the servo (Figure 11-7). You should be aware that there is no standard for the exact relationship between pulse width and position; different servos may need slightly longer or shorter pulses.

Servo motors have three wires: power, ground, and signal, as Figure 11-8 shows. The power wire is typically red and should be connected to the 5V pin on the Arduino board, though some servo motors require more power than the Arduino can provide. The ground wire is typically black or brown and gets connected to a ground pin on the Arduino board. The yellow (or white) wire of the servo goes to the digital pin that you're using to control the servo. In Figure 11-8 it's pin 9, and the red and black wires go to +5V and ground, respectively.

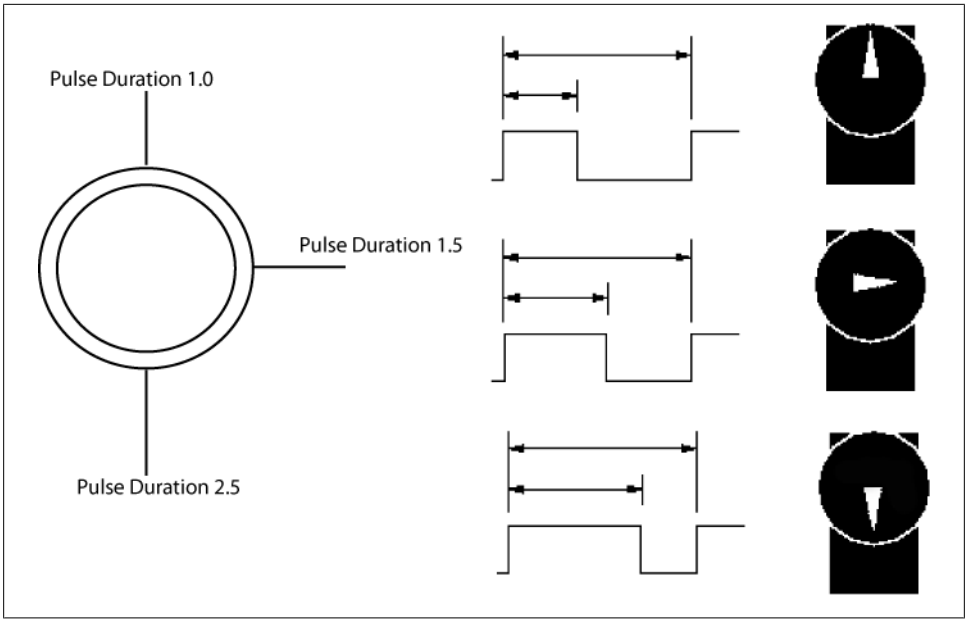


Figure 11-7. Positioning a servo using the pulse width or duration

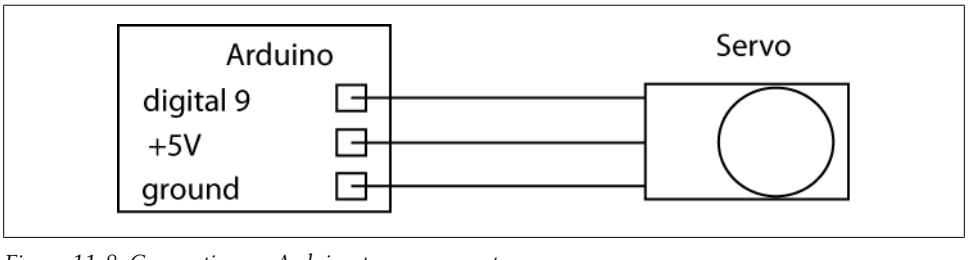


Figure 11-8. Connecting an Arduino to a servo motor

Wiring a Servo

As you'll see in the code snippet that follows, controlling a servo is really just a matter of determining how long in microseconds you need to send each command to the servo. Sending commands to the servo is a matter of writing a pulse to the motor that is the desired length. In this example, a value between 0 and 180 degrees (followed by a nondigit character) is sent using the serial connection to a computer to the servo. For example, 180 microseconds will move the servo fully in one direction, and 0 microseconds will move it fully in the other:

```
int refreshTime = 20;
int command;
int pulseWidth = 1500; // this is the default value
long lastServoPulse;
```

```
int servoPin = 9;

void setup(){
  pinMode(servoPin, OUTPUT);
}

void loop(){
    if(Serial.available()){
        command = Serial.read(); // get the keyboard input
        if(command > '0' && command < '9')
            command = command - '0';
        else{
             pulseWidth = (command * 9) + 700; // get the right amount of time
             command = 0; // reset to get ready for the next command
        }
    }
    sendPulse();
}

void sendPulse(){
    if(millis() - lastServoPulse > refreshTime) {
        digitalWrite(servoPin, HIGH);
        delayMicroseconds(pulseWidth);
        digitalWrite(servoPin, LOW);
        refreshTime = millis(); // make sure we only update at the right moments
    }
}
```

Many objects that you want to turn or rotate can be manipulated with a servo, cameras, mirrors, directed lighting etc. To control these kinds of movement there's an easy correlation between a potentiometer and a servo; turn the potentiometer, and the servo turns. You can use buttons as well, both to set the servo to a preset position or by turning the servo as the button is held down. Other input devices that can be matched to a servo are infrared and ultrasonic sensors, which were covered in Chapter 8.

Another way of programming the servo is to use the Servo library, which is included with the Arduino IDE. This library provides a somewhat easier way of communicating with the servo motor.

This library lets an Arduino board control one or two RC (hobby) servo motors on pins 9 and 10. Servos have integrated gears and a shaft that can be precisely controlled. Standard servos let the shaft be positioned at various angles, usually between 0 and 180 degrees. Continuous rotation servos let the rotation of the shaft be set to various speeds.

This library uses functionality built in to the hardware on pins 9 and 10 (11 and 12 for the Mega) of the microcontroller to control the servos without interfering with other code running on the Arduino. If only one servo is used, the other pin cannot be used for normal PWM output using `analogWrite()`. For example, you can't have a servo on pin 9 and PWM output on pin 10.

The Servo library provides three methods:

`attach()`
: Starts reading the `Servo` instance on the pin passed in to the method and has the following signatures:

```
servo.attach(pin)
servo.attach(pin, min, max) // optionally set the min and max pulse widths
```

`write()`
: Writes a value to the servo. On a standard servo, this will set the angle of the shaft (in degrees). On a continuous rotation servo, this will set the speed of the servo (with 0 being full speed in one direction, 180 being full speed in the other, and a value near 90 being no movement).

`read()`
: Reads the current angle of the servo (the value passed to the last call to `write()`).

To start the Servo library, declare an instance of it in your application code:

```
Servo servoInstance;
```

In this example, a potentiometer is attached to analog pin 0, and then the value from the potentiometer is used to set the position of the servo:

```
#include <Servo.h>
Servo myservo;  // create the servo object

int potpin = 0;  // analog pin used to connect the potentiometer
int val;    // variable to read the value from the analog pin

void setup()
{
    myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{
    val = analogRead(potpin); // reads the value of the potentiometer
        //(value between 0 and 1023)
    val = map(val, 0, 1023, 0, 180);// scale it to use it with the servo
        //(value between 0 and 180)
    myservo.write(val); // sets the servo position according to the
        scaled value
    delay(15); // waits for the servo to get there
}
```

One issue to keep in mind is that when you send a command to the servo, it takes a moment for the actual position of the arm to reach that point. If you want the servo to go from 0 degrees to 180 degrees, you'll probably need to send the command multiple times so that the motor can arrive at the position. In the following example by Hernando Barragan, the servo slowly rotates around its full range. Note the delay to allow the motor to move into the correct position:

```
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
```

```
int pos = 0;    // variable to store the servo position
void setup() {
    myservo.attach(9);  // attaches the servo on pin 9 to the servo object
}

void loop()
{
    for(pos = 0; pos < 180; pos += 1)  // goes from 0 degrees to 180 degrees
    {                                  // in steps of 1 degree
        myservo.write(pos);            // tell servo to go to position in
                                             //variable 'pos'
        delay(15);                     // waits 15ms for the servo to
                                             //reach the position
    }
    for(pos = 180; pos>=1; pos-=1)     // goes from 180 degrees to 0 degrees
    {
        myservo.write(pos);            // tell servo to go to old position
        delay(15);                       // waits 15ms for the servo to
                                               //reach the position
    }
}
```

Lots of different types of servo motors are available. Most have only a 180-degree turning range, but some have 1.5- and 3-turn radii. Others, like the Parallax Continuous Rotation Servo, allow for full and complete continuous rotation.

There are several different shields and boards to simplify the wiring for servo motors. The Adafruit stepper shield pictured in Figure 11-6 is among them. Driving multiple servos—for instance, to create a simple robotic arm or a piece of animatronics—can get complex quickly, and though using a board will increase the price of your project, it can also make your wiring cleaner and your assembly quicker.

Another option for working with servo motors that you may be interested in is the MegaServo Hardware Servo library developed by Michael Margolis that is available for download on the Arduino website. This library allows an Arduino board to control 1 to 12 RC (hobby) servo motors on any digital pin on a standard Arduino board or up to 48 servos on an Arduino Mega. It can be used just like the Servo library. It also allows you to control up to 48 servos on the Arduino Mega or 12 servos on the other boards. Any digital pin can be used with any servo, and pulse widths can be written and read in degrees or microseconds that makes writing the pulse widths easier.

Some experiments with servos that you might be interested in trying are creating a motion detector using several infrared motion sensors and a servo to control a webcam. By using four infrared motion sensors pointed in each of the four cardinal directions, you can determine where motion is occurring and use a servo to point a camera in that direction. You'll need a 360-degree servo to do this. Another interesting idea is creating a drawing machine by using servos to control a pen or pencil. By pairing servos, you can have one mechanical control to change the position of the pen and another to control the location of the paper. In tandem, they would be able to draw pictures and write letters.

Using Household Currents

The Arduino controller can easily power a small LED light or a very small incandescent bulb, but to power up anything larger than that, you'll need to use more current than you can safely run through the Arduino pins. This means that you'll need to run the power through a separate piece of hardware called a *transistor* and control that piece of hardware using the Arduino controller. A transistor can act as a switch to a circuit that carries a certain amount of current, in this case, way more current than you want to allow near your Arduino pins. Normally the transistor doesn't allow the circuit to complete, and no current flows through the high power circuit. But when the base of the transistor is turned on, the transistor completes the larger circuit, and whatever you're powering with the big circuit turns on.

In Figure 11-9, you can see the solid-state relay with its input and output posts labeled. The input posts are attached to the Arduino so that the switch inside can be thrown by the 5V signal from the Arduino. The output posts are used to control the voltage for the larger appliance, such as a light. You can control larger voltages using a combination of a diode, a transmitter, and a relay, but an easier approach is to use a solid-state relay, which combines all three of these into a single unit and makes wiring substantially less difficult. Note that most of these are used to control AC current like household current, not DC current.

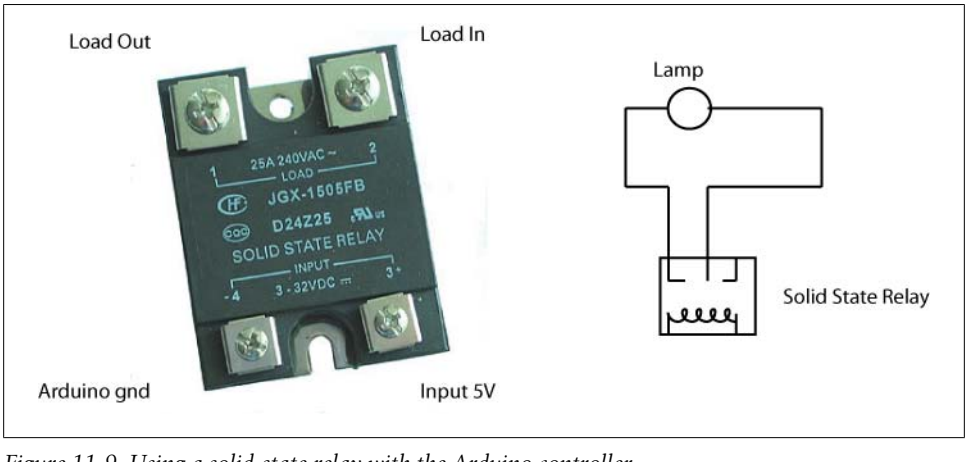
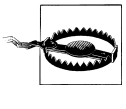


Figure 11-9. Using a solid-state relay with the Arduino controller

To control household AC voltages such as a standard household light, you need something like a relay to switch the high voltage and isolate it from the Arduino.



You'll need to be very careful when following along with these instructions because you're dealing with enough electrical current to hurt yourself and start a fire. This is important enough that it bears repeating: household current can hurt you and start fires.

Figure 11-10 shows how to hook up a household AC light to the solid-state relay.

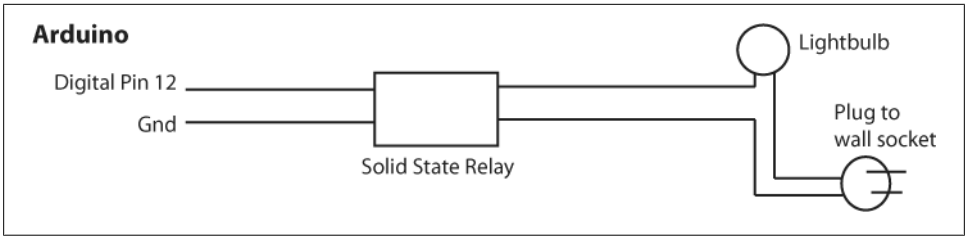


Figure 11-10. Connecting a lightbulb and a solid-state relay to the Arduino controller

The code for this can be quite simple:

```
int relayPin = 12; // choose the pin for the relay
void setup() {
    pinMode(relayPin, OUTPUT);  // declare relayPin as output
}

void loop(){
    digitalWrite(relayPin, LOW);  // turn relay OFF
    delay(1000);
    digitalWrite(relayPin, HIGH);  // turn relay ON
    delay(1000);
}
```

Beware that turning lights on and off frequently in a short span of time could burn the bulb out quickly. However, you can do more than turn lights on and off with this controller: you can control any household device as long as it does not exceed the current capacity of the relay you use.

Working with Appliances

One simple way to create physical feedback is to use preexisting appliances. There's a great device for doing this created by the guys at LiquidWare called the RelaySquid, shown in Figure 11-11. The RelaySquid is a board that has four relays, each of which controls an extension cord with three outlets. Each relay can be individually controlled by the Arduino by sending it a digital signal.

The pins marked "Relay drive" should each be attached to the pins on the Arduino. The following bit of code assumes that relay drive pin 1 is attached to digital pin 4, that relay drive pin 2 is attached to digital pin 5, and so on. It will turn each outlet on the RelaySquid for one second and then go onto the next one:

```
void setup() {

    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);
    pinMode(7, OUTPUT);
```

```
}

void loop() {

    digitalWrite(HIGH, 4);
    delay(1000);
    digitalWrite(LOW, 4);
    digitalWrite(HIGH, 5);
    delay(1000);
    digitalWrite(LOW, 5);
    digitalWrite(HIGH, 6);
    delay(1000);
    digitalWrite(LOW, 6);
    digitalWrite(HIGH, 7);
    delay(1000);
    digitalWrite(LOW, 7);

}
```

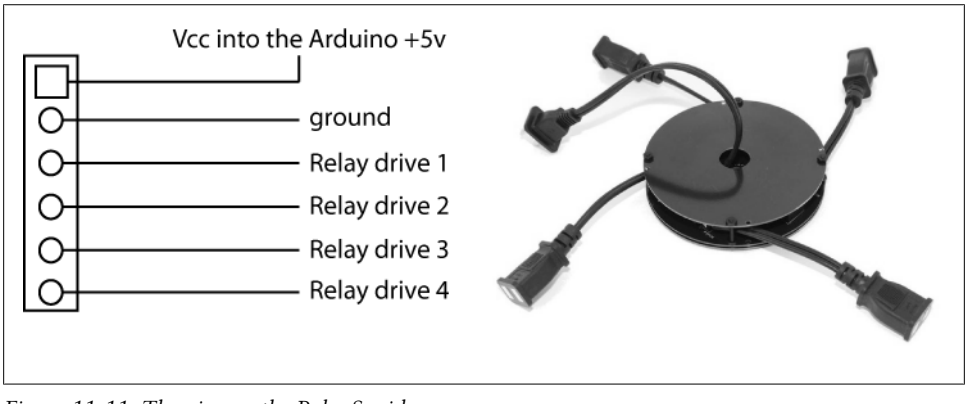


Figure 11-11. The pins on the RelaySquid

So, beyond turning lights on and off, what can you do with the RelaySquid? For starters, you could attach a clock sensor to the Arduino and turn an appliance on or off at a specific time for home security. You could program a schedule for grow lights to turn on and off for plants. You could turn a television, a generator, or any other appliance on or off. You need to take a few things into account: some devices work well with rapid cycling, and some do not. Some types of lights work well being turned on and off quickly. In my experience, halogen bulbs work particularly well, and incandescent bulbs burn out quickly when turned on and off in rapid succession. Televisions do not work well when turned on and off quickly. A little bit of searching and asking questions on forums can save you unpleasant surprises later on.

Other preassembled components are available for switching large loads like the one shown in Figure 11-12 on the left from *electronics-lab.com* and the one shown on the right created at *sparkfun.com*.

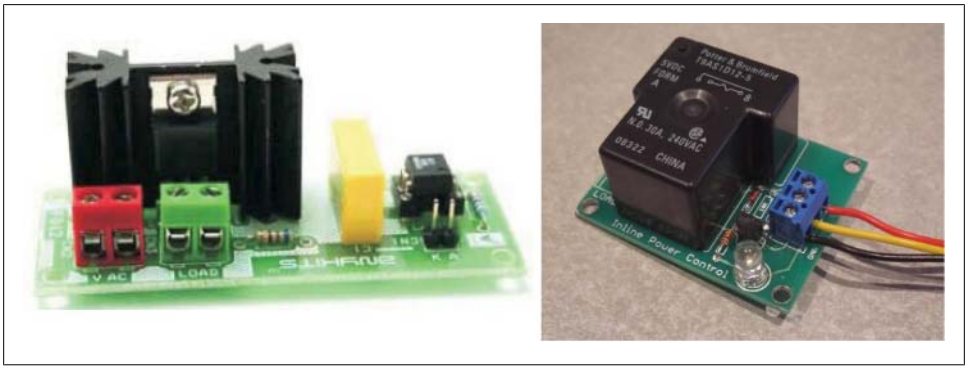


Figure 11-12. Two shields for working with household currents: the anykit Solid State Relay Switch and the sparkfun Inline Power Control

Introducing the LilyPad Board

There's another Arduino-compatible board that has a particular relevance to providing physical feedback: the LilyPad board shown in Figure 11-13 and developed by Leah Buechley. One of the challenges of providing physical feedback is always deciding on the scale and scope of that feedback. A signal should be appropriate to its message and also to the context in which the user finds themselves, which is why there's a discrete vibrate setting on almost all cell phones and mobile devices. Removing this would make some signals inappropriate for their context, like a loud ringing during an opera. And so it goes with interactive design: ensuring that the correct feedback is provided for the context is a vital affordance for the user. What does this have to do with the LilyPad? The LilyPad Arduino is a microcontroller board designed for wearables and e-textiles. It can be sewn to fabric and mounted to power supplies, sensors, and actuators with conductive thread. Leah Buechley has provided code and schematics and links to several projects on her website at *web.media.mit.edu/~leah/*, including a turn signal jacket for bicyclists. So, the means of input and feedback can become much more subtle, becoming invisible or inaudible to anyone except the user. It also allows you to use preexisting form factors. That is, you can simply sew the LilyPad into a shirt, a jacket, a bag, or any other common object and let that object create the form of your device.

The board is based on the ATmega168V, which is a slightly lower-power version of the ATmega168 chip that the Arduino uses. It is programmed using the Arduino IDE just like any other Arduino device and can be powered from a USB if you either get a USB connector or use one of the USB modules for the Arduino Mini, as shown Figure 11-14.

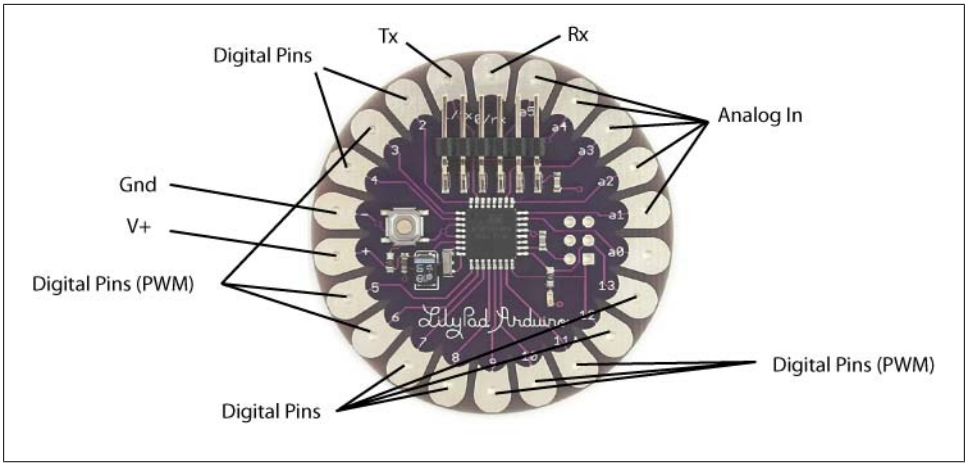


Figure 11-13. The LilyPad

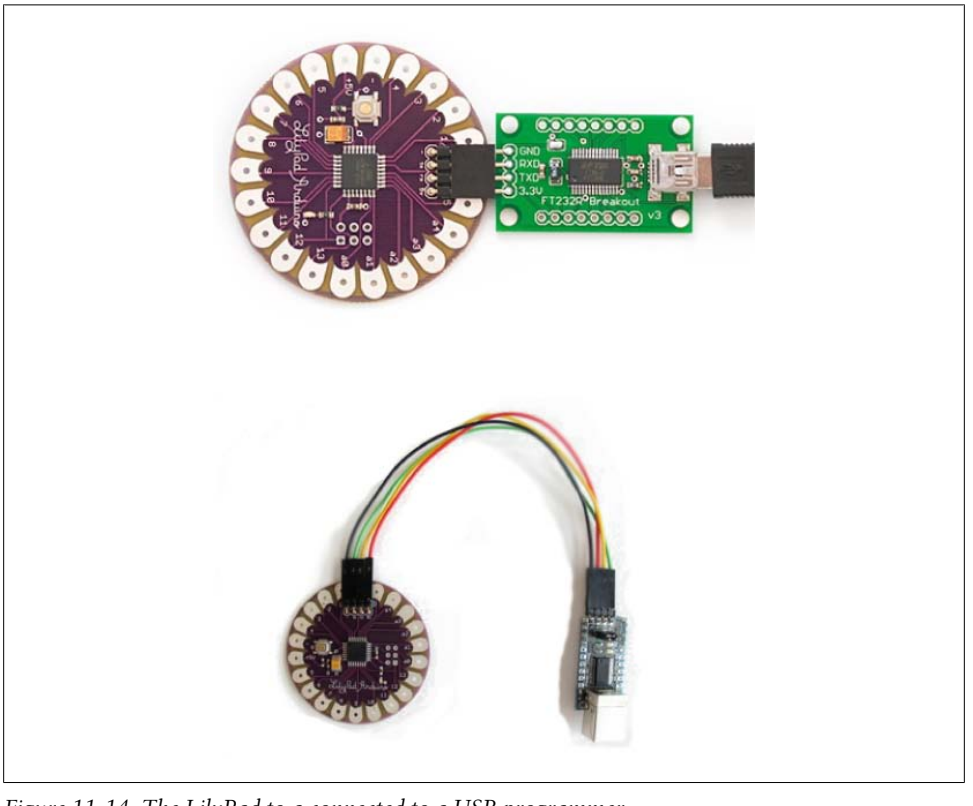


Figure 11-14. The LilyPad to a connected to a USB programmer

It has 14 pins; all can be used as digital outputs, 6 can be used for analog (PWM) outputs, and 6 can be used for analog inputs. The LilyPad is also a little more fragile than a regular Arduino board, so you'll need to be a bit careful when working with it and particularly when wiring it up. Also, the LilyPad is not particularly sensitive to water. If the LilyPad is not powered up, it can be washed with nonabrasive detergent. This doesn't help when your project is caught in a sudden torrential rainstorm, but it does make the device a little less susceptible to the sweat, small spills, and environmental moisture that a wearable piece of technology will invariably encounter.

Using Vibration

Vibration is a simple and subtle signal that, given the correct contextualizing information, can be a great alert to a user. The LilyPad comes with a number of prebuilt components that can easily be connected either as inputs or as feedback devices, one of which is the LilyPad Vibe Board shown in Figure 11-15. This small motor works in much the same way as the vibrate motor on a cell phone. A 5V signal is enough to power the vibration motor, alerting the user to whatever you'd like to alert them to.

Figure 11-15. LilyPad Vibe Board

Since the LilyPad is generally meant to be used in a completely mobile application, you'll need to provide power to it. One AAA battery is enough to power the LilyPad for a reasonable amount of time using the LilyPad power supply. You can also use the LiPower module with a Lithium Ion battery, shown in Figure 11-16.

The LiPower board is small, is inconspicuous, and lets you use a rechargeable Lithium Polymer battery like the one shown on the left in Figure 11-16. These batteries are smaller, are flatter, and last much longer than AAA cells. Simply attach a single cell for Lithium Polymer (LiPo) battery, flip the power switch, connect the LiPower battery to the LilyPad, and you have a small, portable, short circuit–protected 5V supply.

The last part of the puzzle with the LilyPad is the connection between the LilyPad and all of its component pieces. When working with an Arduino Duemilanove, most of your connections are made with 22- or 24-gauge wire or with solder. This approach doesn't translate as well to clothing, so you might considering using a special kind of conductive thread that can be used to connect the LiPower, the LilyPad, and the Vibe Board. Conductive thread is an excellent way to connect various electronics onto clothing. It isn't as conductive as traces on a printed circuit board, but it lets you create a complete circuit that remains flexible, near invisible, and, most important, wearable.

Figure 11-16. Lithium Ion battery and the LiPower board

Note that when designing applications with the LilyPad, you should always keep your power supply and LilyPad main board as close to each other as possible. If they are too far apart, you are likely to have problems with your LilyPad resetting or just not working at all because of the resistance in the conductive thread connecting the boards. The resistance usually isn't as big of a deal when it's on the line connecting the LilyPad to a sensor or component, but between the LilyPad and its power supply, it makes a big difference in the reliability and durability of your project.

Figure 11-17 shows how to connect the LilyPad, LiPower, and Vibe Board.

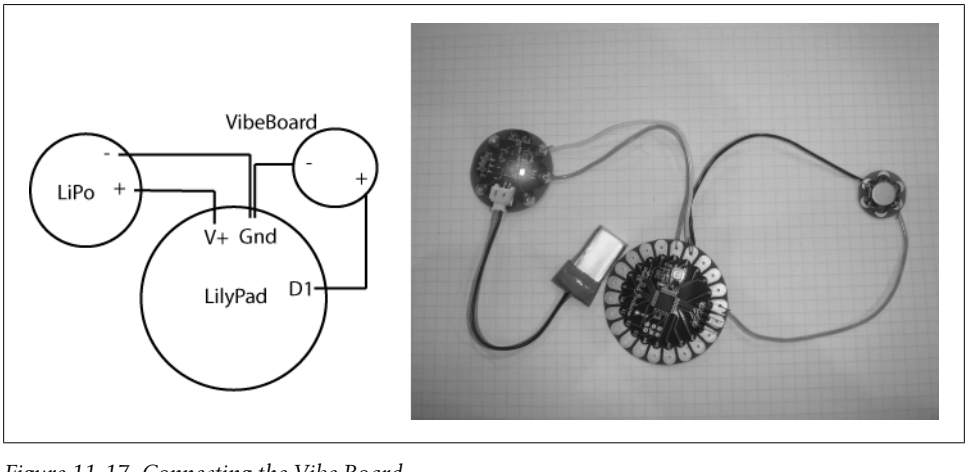


Figure 11-17. Connecting the Vibe Board

The code to turn on the Vibe Board can be as simple as this:

```
void setup() {

    pinMode(1, OUTPUT);

}

void loop() {

    digitalWrite(HIGH, 1);
    delay(1000);
    digitalWrite(LOW, 1);
    delay(1000);

}
```

As of the writing of this book, the LilyPad has several other available types of components for input: an accelerometer, a light sensor, a temperature sensor, a button board, and an XBee-compatible breakout board for integrating an XBee antenna into a LilyPad system. Some of the other feedback options that the LilyPad provides are sound-generating buzzers and three-color LED components. At the moment, all LilyPad components are available through the Sparkfun website as well as other online retailers.

Interview: Troika Design Studios

Troika is a design studio based in London that works with graphic design, product design, technology development, sculptural projects, and interactive installations. Conny Freyer, Eva Rucki, and Sebastien Noel founded Troika in 2003. They are also the authors of the book *Digital by Design* (Thames and Hudson).

Where do you think the boundary between fine art and design is located with regard to new media art or interactive art? Is this a relevant or important question for you?

Troika Design Studios: We describe ourselves as an art and design studio, but wouldn't describe ourselves as new media artists or interactive artists. Although terms like *new media art* and *interactive art* are very vague, they seem rather limiting because they define the work you are making by the tools you are using.

Although any attempt to classify new movements is interesting, we enjoy being part of a movement that has so far escaped any coinage and defines itself through blurring the boundaries between established genres and disciplines.

Can you address the notion of collaboration throughout the conception and execution of a project? Is the collaboration something continuous or something more akin to a traditional software development practice where each team does its part and hands it to the next team? How do you handle communicating across disciplines?

Troika: We are continuously working on a range of self-initiated and commissioned projects, which follow similar processes from concept to design development to production and implementation.

All three partners take part in the concept stage after which the project is led by one partner who is responsible for the planning, the communication with the client, and the division of work between the partners as well as other collaborators. Tasks are naturally distributed according to different skill sets. We collaborate with a close-knit network of specialists, which we employ depending on the project. An iterative process is crucial to our way of working, which happens internally between the team members as much as with our clients and manufacturers.

Communicating across disciplines takes time and practice. It's literally like learning a different language. We were fortunate enough to follow each other's developments throughout the master's degree program at Royal College of Art in London, which helped to develop a common vocabulary.

Can you give a little background on the history of Troika?

Troika: Conny Freyer, Sebastien Noel, and Eva Rucki founded Troika in 2003, after we graduated from the Royal College of Art. Since then we have worked for clients such as MTV International, British Airways, the BBC, Warner Music UK, Thames and Hudson, and the London Science Museum.

Although the work of our studio spans various disciplines from graphics to products to installations, a lot of the themes—like creative use of technology and cross-fertilization between the art and design disciplines—are recurring subjects.

Since the beginning we have frequently taken part in national and international exhibitions and conferences, including Get it Louder (in 2007 in Guangzhou, Shanghai, and Beijing), Responsive (in 2007 in Tokyo), the Science of Spying (in 2007 at the Science Museum London), Noise of Art (in 2007 at Tate Britain, London), Design and the Elastic Mind (in 2009 at the Museum of Modern Art in New York), Volume(s) (in Casino Luxembourg, Forum d'art contemporain), and ExperimentaDesign (in Amsterdam in 2008).

Are there are good precedents for companies or collectives working across fine art and design that influenced you and the way that you structured Troika?

Troika: Collectives like Tomato and Antirom were certainly an influence during our studies, because of their multidisciplinary approach and because they were working as collectives rather than presenting themselves as one person (who employs others to complement his or her skill set), as is often the case in the product design world. But their mix of skills was quite different from ours, and the commercial landscape in which they started out had changed quite a bit when we set up our studio.

Troika's structure developed naturally based on the backgrounds of the founding members and a shared interest in the creative use of technology and cross-fertilization between the art and design disciplines.

These interests are apparent in our work for the Science Museum (*Identity for the Science Museum Art Projects* galleries and concept products for the Spymaker exhibition,

which examine the future of spy technologies) as much as in our forthcoming book *Digital by Design*. It's an overview of the fusion of digital technology and art and design production, as in our installation work for Terminal 5 of the Heathrow airport.

Do you consider yourselves part of a lineage of artists and designers or of a school of design in any way?

Troika: Troika is part of a generation of artists and designers who creatively engage with technology in order to provoke questions, experiment, and explore its potentials and impacts. A lot of these artists grew up during the transient phase where analog technologies were gradually replaced—the record players and View-Master. As such, they understand the benefits, appeal, and importance of the materiality and the tangible component of technologies versus the all-digital, immaterial, which prevailed at the start of the digital era.

The influence of sci-fi—from films such as Kubrick's *2001 Space Odyssey*, Scott's *Blade Runner*, Orwell's novel *1984* (Plume), Gibson's novel *Neuromancer* (Ace), Ballard's collection of short stories, or the more popular *Star Wars*, to name but a few—sparked a willingness to think and create in technological terms, finding new avenues and alternative futures. A lot of artists working in this field share a more or less intense fascination for technology itself, its potentials for artistic expression, and its social and cultural impacts. Other notable influences include kinetic and media art, the counterculture of the '80s and '90s (more precisely in DJ culture, which influenced the process in terms of remixing), hacker communities (as the Berlin-based Computer Chaos Club), street art, and graffiti.

How do you view what I've been calling "the ecology of new media art," by which I mean economic possibilities, exhibition opportunities, exchange of dialogue, and so on. Do you see it forming interesting discussions and fueling work or not? Do you have a sense of how it this is different across different parts of the world?

Troika: The general approach of artist and designers working in this field is very collaborative. Sharing knowledge as well as the joy of discovery online as well as at conferences is a common feature and encourages a constant exchange and debate. Part of the reason may be that a lot of small structures are operating in this field that have adopted a flexible and associative process, pulling together the right competences when needed, rather than growing into a bigger apparatus in order to ensure their autonomy.

Who are the artists and designers working today that you are most interested in?

Troika: We have just written a book called *Digital by Design* that features artists and designers who employ technology in unexpected, playful, disruptive, functional, or critical ways. What was most challenging, but also most interesting, to us when writing the book was that the works we selected successfully escape any coinage such as *media art* or *interactive design*. The selected artists and designers have one thing in common: an eagerness to explore the intersection between science, art, design, and technology in the search for humanist values, humor, magic, and sensory experiences.

We love Daniel Rozin's *Mechanical Mirrors* for the tangible beauty, Dunne and Raby's work for their critical approach toward design, and Ron Arad's sense of exploring the

new. What we look for in all the works we admire is the increasingly lacking desire for thought, enjoyment, quality, and craftsmanship.

A piece like Sonic Marshmallows *appears at first very nontechnological, but it allows you to think about things that are enabled by technology without using any electronics or power. Can you talk a little about how a piece like that fits in with the rest of your works and how you see it functioning in relation to people's desires to interact with things?*

Troika: We are fascinated by technology—being kinetic, optical, sonic, or electronic—and its impact on people. So, a lot of our work is informed by technology, but it is not a prerequisite. What we are striving for in our work is immediacy. We want our works to trigger emotions or thoughts like a story or film might do, and we love when art, magic, and science appear to be crossing paths. *Sonic Marshmallows* is a great example; it's technology in its purest form. Older technologies, especially the ones directly linked to natural optical and acoustic phenomena, carry a very simple and innate beauty that reveal their depth and magical surprise once you experience them. We see *Sonic Marshmallows* more as a manifestation of people's desire to interact with each other, rather than with things. *Sonic Marshmallows* can be used only in conjunction with another user. Sound mirrors were originally used on the coast of Kent to detect incoming enemy planes, not far from the location were *Sonic Marshmallows* is installed now. We used the same technology in a way that enables people to communicate with each other instead.

You've created the electroprobe, a device for listening to electrical currents that surround us in our everyday environment. Can you talk a little bit about working with this tool and why you've come back to this theme several times?

Troika: We first exhibited the electroprobes as an exploration tool for the immediate surrounding; for example, in galleries people could discover the sounds produced by the radio magnetic radiations of the surrounding lamps, closed-circuit TV cameras, electricity wires, and so on.

We then became interested in the wide range of sounds that electroprobes can pick up, including sounds from objects, which might not appear in your common gallery context, such as LED boards, fans, fridges, and so on. In order to enable visitors to hear the varied sounds of their radio magnetic environment, we curated installations of electronic objects.

When we were invited by the British Council to China, we decided to make a location-specific sculpture using only electronics from the Chinese markets. The objects were arranged according to the sounds they were producing to create one big electromagnetic organ. The user becomes in a certain sense the composer because he will determine which sounds become audible through navigating this electromagnetic soundscape with the aid of the electroprobe.

Do you approach conceiving of a print work differently than you approach conceiving of an interactive piece?

Troika: Projects that are most interesting to us are the ones where we can bring all media together: print, installation, product, and animation. The main difference between one work or another is usually whether it is something that is applied or

not—something that is about answering a brief or creating a work that is self-initiated. We approach all in the same way. We want to engage and surprise people through sensory experiences or through making them think. And this counts for a piece of print as much as for an installation.

This is a very broad question, so feel free to answer however you like. What are the fundamental rules for designing an interaction?

In Troika's practice, *interaction design* or *experience design* might have a very different meaning than, for instance, for an interaction designer working in product development.

Troika: Troika strives to engage people on more intuitive and emotional levels. A good example is our kinetic sculpture *Cloud*. Supported by the organic fluid movement of its surface, it provokes associations of a living creature. The flicking sound of a flip dot, which is a small circle that is flipped over to create a sort of pixel and which was a technology used in old airport displays, touches on memories of traveling when you were a child.

Some people have asked us if *Cloud* is interactive—if it moves only when you walk by. Our answer to that is that interaction can mean many things and goes far beyond touching a button and getting a response once you have done so. A sculpture such as *Cloud* is not based on such a linear approach.

Nevertheless, designing interactivity is an interesting task. Martin Heidegger sums it up pretty well. He describes that the only time he really noticed his hammer was when it was broken and that he was thinking about the nail instead of the hammer when the hammer was working properly. It is about making the interface or the object invisible and tapping into what people already know, where they are able to recognize form and behavior without having to think about it.

What software or hardware tools do you use, and how do you feel they shape your vision or your working process?

Troika: We use the common 2D, 3D, and animation software packages. We draw and write to formulate our ideas. We have a workshop in which we experiment and build our prototypes and smaller pieces.

Our ideas and designs are to a great extent influenced by the tools we are using. Try to formulate a concept without words! We believe that good craftsmanship and intimate knowledge of your field is essential. This counts for digital/virtual solutions as much as tangible solutions.

A piece like All the Time in the World *has an extremely complicated technological element, a beautiful typeface, an element of motion graphics, and a conceptual element that plays on locations well-known and well-known but rarely considered. How do all these ideas come together in a piece? Is it born more of necessity or a desire to be able to shape all aspects of a piece?*

Troika: Our approach is quite holistic, and we find it an enriching and creatively challenging experience to develop the different aspects of a project.

All the Time in the World developed—like many of our projects—in a nonlinear fashion. We had created the Firefly typeface after we learned about the possibilities of electroluminescent technology. When British Airways commissioned for an art installation for its reception hall in T5 using the *Firefly* display, we started from the semantics of the location to develop the content for the installation.

Troika: We are investigating possibilities for technological innovation—mainly in the form of reappropriating existing technologies—on an everyday basis. At the same time, we are very aware of the contexts in which these technologies were previously used or developed, and this informs the way we will use it in our projects. For example, the flip dot technology used in *Cloud* stems from the signage boards used in the '70s and '80s in train stations and evokes notions of travel. When those little inventions and context-specific concepts come together in an installation, it creates a multilayered end result, which carries meaning on more than one level.

Using an LED Matrix

A light-emitting diode (LED) can indicate something to a user, it can provide recognition of a user action, or it can act as an alert. Using multiple LEDs in concert with an Arduino is usually limited by the number of digital out pins that the Arduino has. By using an LED driver chip, though, you can control up to an 8 × 8 matrix of LEDs or an eight-digit LED display. This lets you draw simple shapes, characters, and digits to begin creating simple animations.

This section is going to be rather long because of the great number of ways that you work with LEDs and in particular with LED *matrices* (group LEDs arranged in rows and columns). These are the three relevant libraries for working with LEDs in Arduino:

Matrix
: This library enables you to work with a single LED driver.

LedControl
: This library enables you to work with multiple LED drivers and newer drivers as well.

Sprite
: This library allows you to create image sprites to use with the Matrix library.

Using the Matrix Library

We'll focus on the Matrix library first. The chip and the idea of an LED matrix requires a bit of explanation. First the chip: the most common approach to drive an LED matrix, and the one that we'll examine in this section, is to use a chip called the Maxim MAX7221 to drive the LEDs.

Figure 11-18 shows the pins of the MAX7221.

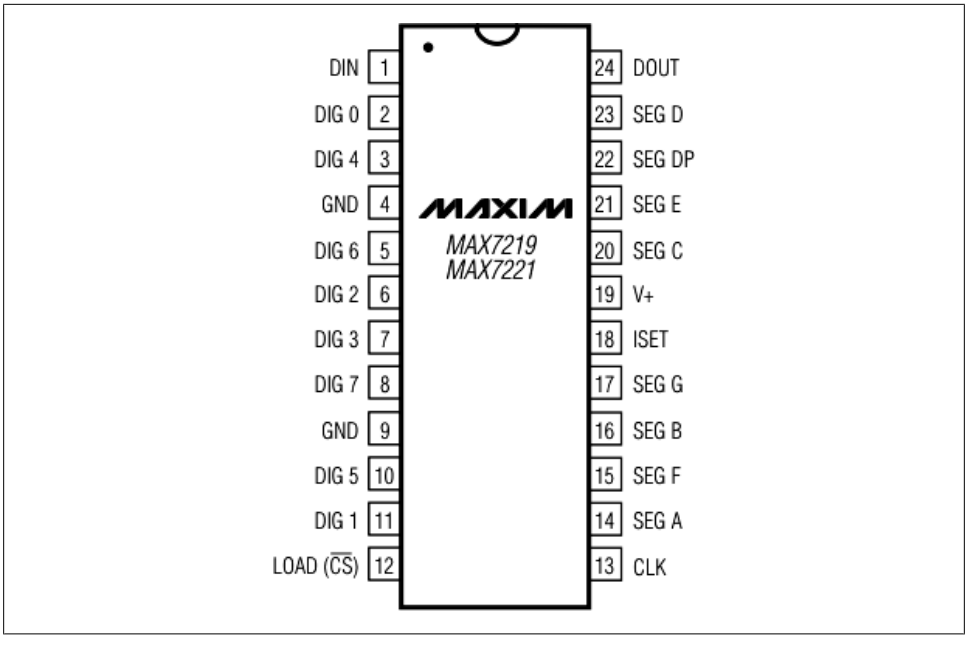


Figure 11-18. Pins on the MAX7221

In Figure 11-19, you'll see how it can be wired to an 8 × 8 LED matrix and an Arduino.

You'll notice that the wiring from the MAX7221 looks a bit complex, but it's actually made easier by the chip's pin diagram in Figure 11-19. The pins on the LED matrix go from DIG 0 to DIG 7, left to right, and SEG DP (comes before SEG A) to SEG G, top to bottom.

The Matrix library allows you easily communicate with the MAX7221 chip. You can turn individual LEDs on or off, as well as setting the brightness or clearing the LED matrix. For example, to make a smiley face, you import the Matrix library and tell it which pins are connected to the MAX7221.

The constructor for the Matrix library looks like this:

```
Matrix mat = Matrix(2, 3, 4);
```

Those three parameters specify which pins are connected to the MAX7221 control pins. The first is which pin is connected to the data (din), then which pin is connected to the load (load), and lastly which pin is connected to the clock (marked `clk` on the MAX7221). These pins are used internally by the `Matrix` class to drive the chip:

```
#include <Matrix.h>

Matrix mat = Matrix(2, 3, 4);

void setup() {
    // nothing here at the moment
```

```
}

void loop()
{
    myMatrix.clear(); // clear display
    delay(1000);

    // turn some pixels on
```

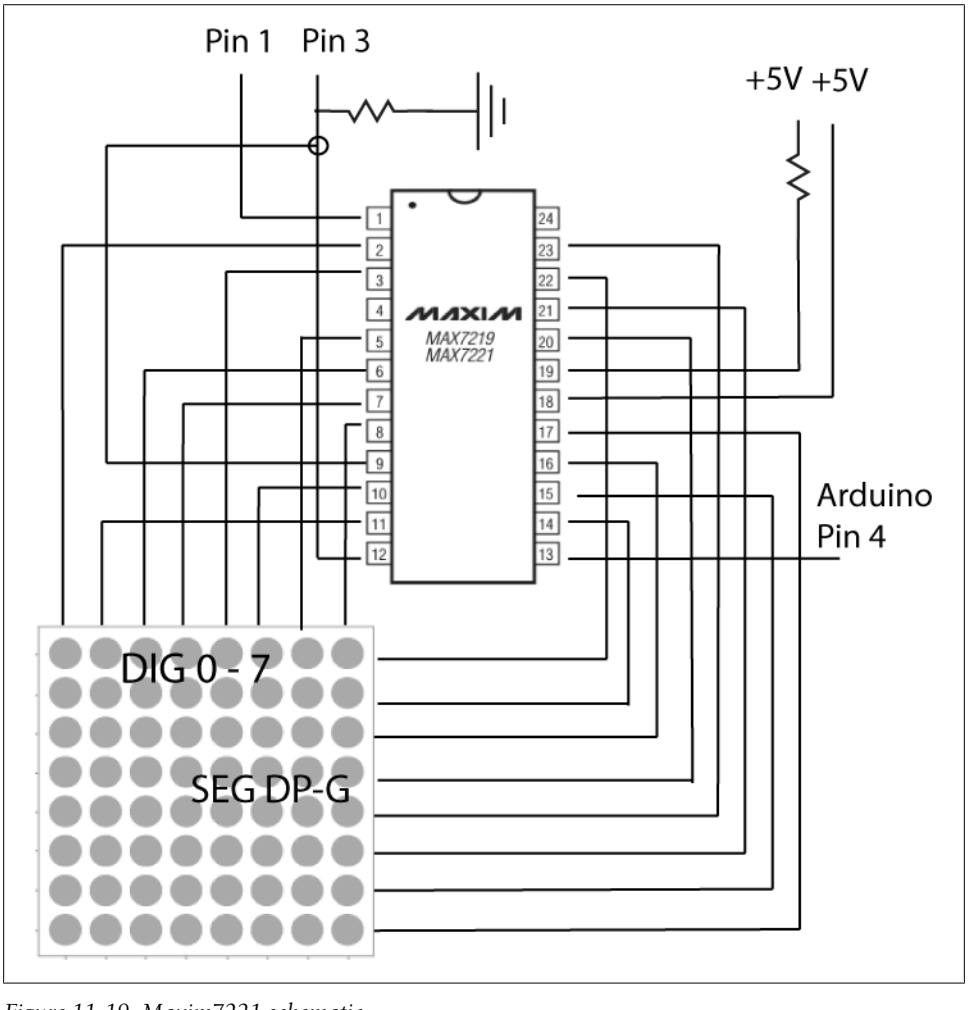


Figure 11-19. Maxim7221 schematic

The `write()` method take two parameters, the first to indicate the row of the LED that should be turned on and the second to indicate the column of the LED that should be turned on. The following calls to the `write()` method will create a smiley face:

```
    mat.write(1, 5, HIGH);
        mat.write(2, 2, HIGH);
        mat.write(2, 6, HIGH);
        mat.write(3, 6, HIGH);
        mat.write(4, 6, HIGH);
        mat.write(5, 2, HIGH);
        mat.write(5, 6, HIGH);
        mat.write(6, 5, HIGH);

        delay(1000);
    }
```

Aside from the tricky wiring situation, creating a single LED matrix isn't too difficult. Creating multiple LED matrices is also fairly easy, but it requires that you use a different Arduino library: the LedControl library.

Using the LedControl Library

The LedControl library is quite similar to the Matrix library except that it allows you to easily work with multiple MAX7221 and LED matrix pairings. Note, though, that for each LED matrix that you want to use, you'll need to have a driver chip for it. You can use up to eight driver and matrix pairs with this library. The constructor looks like this:

```
#include "LedControl.h"

LedControl lc1=LedControl(12,11,10,1);
```

The initialization of the LedControl library takes four arguments. The first three arguments are the pin numbers on the Arduino that are connected to the MAX7221, just like the Matrix library. You are free to choose any of the digital I/O pins on the Arduino, but since some of the pins are also used for serial communication or have an LED attached to them, it's best to avoid pins 0, 1, and 13. I chose pins 12, 11, and 10 in my example.

The fourth argument for the LedControl constructor is the number of cascaded MAX72XX devices you're using with this LedControl. Valid values can be between 1 and 8. There is a little performance penalty implied with each device you add to the chain, but the amount of memory used by the library code will stay the same, no matter how many devices you set. This is quite important with a limited memory device like the Arduino. Here is the syntax for the constructor, followed by a description of the four parameters:

```
LedControl(int dataPin, int clkPin, int csPin, int numDevices);
```

`dataPin`
: An `int` that represents the pin on the Arduino where data gets shifted out

`clockPin`
: An `int` that represents the pin for the clock

`csPin`
: An `int` that represents the pin for selecting the device when data is to be sent

`numDevices`
: An `int` that represents the maximum number of devices that can be controlled

If you need to control more than eight driver and matrix pairings, you can always create another LedControl variable that uses three different pins on your Arduino. This means that if you're using the Arduino MEGA, you can have up to 17 MAX7221 drivers connected. Figure 11-20 shows how to connect the signal lines for multiple MAX7221 chips.

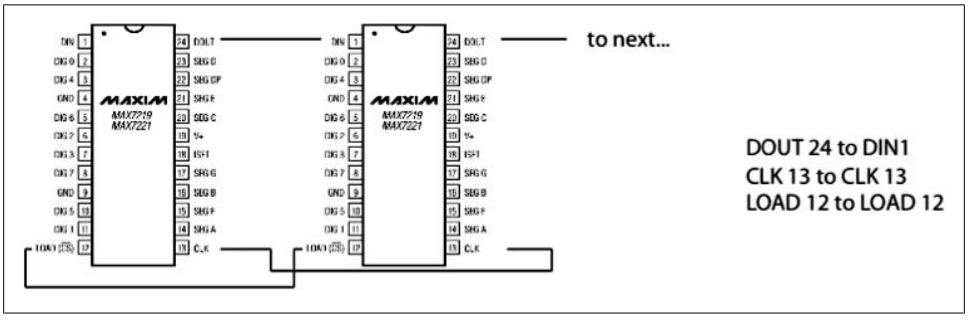


Figure 11-20. Connecting multiple MAX7221 chips

Now you can create an LedControl for two devices:

```
LedControl lc1=LedControl(12,11,10, 2);
```

If you've attached multiple MAX7221 drivers to your LedControl, you can loop through them all using the `getDeviceCount()` method, as shown here:

```
    for(int index=0;index<lc1.getDeviceCount();index++) {
      lc1.setLed(1, 0, 0, 1);
   }
}
```

The `setLed()` method sets the LED value at the address indicated:

```
setLed(int addr, int row, int col, boolean state)
```

The `setLed()` method takes four parameters:

`int addr`
: The address of the display to control, between 0 and 7.

`int row`
: The row in which the LED is located, between 0 and 7.

`int col`
: The column in which the LED is located, between 0 and 7.

`boolean state`
: If `true`, the led is switched on; if false, it is switched off.

To initialize the MAX7221, you may need to wake the device by calling the `shutdown()` method with `false` passed to the method instead of `true`:

```
shutdown(int addr, bool shutdown);
```

This method takes two parameters:

`addr`
: Indicates which MAX7221 should be altered.

`shutdown`
: Indicates whether the device should go into power-saving mode if `true` is passed or whether it should come out of power-saving mode if `false` is passed.

When a new LedControl is created, the library will initialize the hardware with the display blanked, the intensity set to the minimum, the device in power-saving mode, and the maximum number of digits on the device activated. To completely initialize the LED, you'll want to set the brightness on it so that it's ready to display data as soon as the setup is complete on your Arduino. A common display sequence might look like this:

```
void setup() {
    //wake up the MAX72XX from power-saving mode
    lc.shutdown(0,false);
    //set a medium brightness for the Leds
    lc.setIntensity(0,8);
    }
```

To clear a display, call the `clear()` method, and pass the address of the controller that you want to clear:

```
void clearDisplay(int addr);
```

I'll mention two other useful methods in the LedControl library:

The `setRow()` method sets an entire row on or off and uses the following signature:

```
void setRow(int addr, int row, byte value)
```

The `setRow()` method takes three parameters:

`int addr`
: The address of the display to control.

`int row`
: The row on which the LEDs are to be set. Valid values are between 0 and 7.

`byte value`
: A bit set to 1 in this value will light up the corresponding LED.

The `setColumn()` method sets an entire column on or off and uses the following signature:

```
void setColumn(int addr, int col, byte value)
```

The `setColumn()` method takes three parameters:

`int addr`
: The address of the display to control.

`int col`
: The column on which the LEDs are to be set. Valid values are between 0 and 7.

`byte value`
: A bit set to 1 in this value will light up the corresponding LED.

So, now that you have an understanding of how to use multiple LED devices, you can begin creating small displays using multiple LED matrices. Some fun ideas might be a version of the classic video game Pong controlled using potentiometers, small graphics or animations, or a readout for an application.

You might be interested in working with three-color LEDs, that is, LED matrices where each LED segment contains three LEDs (red, green, and blue), meaning that each segment can be adjusted to many different colors by turning the brightness up or down on each LED. These three-color LED matrices can be controlled using multiple MAX7221 controllers, or you can look at using the serial interfaced LED matrices from Sparkfun that include a logic board that handles connecting the logic controller and the pins. To connect those, you'll simply need to connect the Arduino to the backpack included with the Serial matrix and send commands to it using the `spi_transfer()` method. This is a pretty advanced technique that shifts data into the data register to be read by any device that communicates using the SPI protocol.

Using the SPI Protocol

There's only space for a very short tutorial on the Serial Peripheral Interface (SPI) protocol. SPI is generally used to communicate with other microcontrollers. In an SPI connection, there is always a master device that controls all the peripheral devices connected to it over three common lines:

Master in slave out (MISO)
: The line that the slave uses to send data to the master controller

Master out slave in (MOSI)
: The line that the master uses to send data to the slave devices

Serial clock (SCK)
: The clock pulses that synchronize data transmission generated by the master

The really tricky thing about SPI is that generally each device implements it a little bit differently. Information on the nuances for each will always be available on the datasheet for the device in online documentation.

All SPI settings are determined by the Arduino controller's SPI control register (SPCR). A register is just a byte of microcontroller memory that can be read from or written to. Registers generally serve three purposes: control, data, and status.

Control registers control settings for various microcontroller functions. Usually each bit in a control register affects a particular setting like the communication speed.

Data registers simply hold bytes. The SPI data register (SPDR) holds the byte that is about to be shifted out the MOSI line for all the peripheral devices to receive and the data that has just been shifted in the MISO line from any peripheral devices.

Status registers change their state based on various microcontroller conditions. For example, the seventh bit of the SPI status register (SPSR) is set to 1 when a value is shifted in or out of the SPI.

The SPI control register (SPCR) has 8 bits, each of which controls a particular SPI setting:

7 SPIE
: Enables the SPI interrupt when set to 1.

6 SPE
: Enables the SPI when set to 1.

5 DORD
: Sends data last bit first when set to 1 and sends the first bit first, usually the bit that determines whether a number is negative or positive, when set to 0.

4 MSTR
: Sets the Arduino in master mode when set to 1 and sets slave mode when set to 0.

3 CPOL
: Sets the data clock to idle when high if set to 1 and to idle when low if set to 0.

2 CPHA
: Samples data on the falling edge of the data clock when set to 1 and on the rising edge when set to 0.

1 and 0 SPR1 and SPR0
: Sets the SPI speed; 00 is the fastest (4MHz), and 11 is the slowest (250KHz).

This next example will help you understand what this method means when you encounter it, such as if you try to control one of the Serial RGB LED matrices mentioned earlier in this chapter:

```
char spi_transfer(volatile char data)
{
  SPDR = data;                    // Start the transmission
  while (!(SPSR & (1<<SPIF)))     // Wait for the end of the transmission
  {
  };
  return SPDR;                    // return the received byte
}
```

The `spi_transfer()` function loads the output data into the data transmission register, thus starting the SPI transmission. It checks the value of the SPSR to detect when the transmission is complete and, when it is, returns complete. This is how external

memory can be used with Arduino, such as when you need to write to external non-volatile memory. Here's what setting an application up to use SPI looks like:

```
#define DATAOUT 11//MOSI
#define DATAIN  12//MISO
#define SPICLOCK  13//sck
#define SLAVESELECT 10//ss

void setup()
{
    Serial.begin(9600);
```

Here, pin 13 will be the clock pin to connect to the SPIClock pin of the LED matrix controller, pin 12 will go to the DataOut pin, pin 11 to the DataIn, and pin 10 will go to the SlaveSelect pin.

```
    pinMode(DATAOUT, OUTPUT);
    pinMode(DATAIN, INPUT);
    pinMode(SPICLOCK,OUTPUT);
    pinMode(SLAVESELECT,OUTPUT);
}
```

Figure 11-21 shows how the peripheral devices are connected to the Arduino.

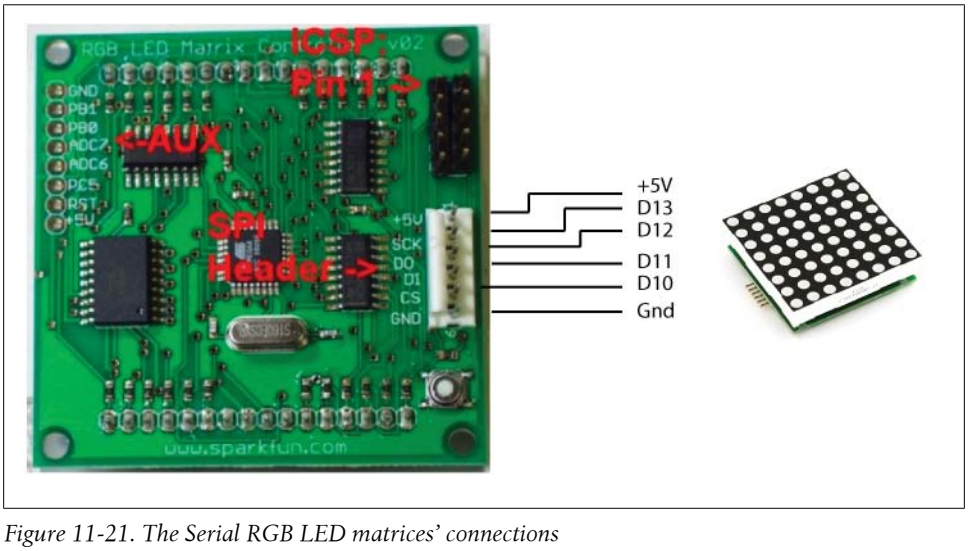


Figure 11-21. The Serial RGB LED matrices' connections

There are dozens of kinds of SPI devices that you might encounter, from memory storage devices to GPS devices to LCD screens.

Using LCDs

We are surrounded by liquid crystal display (LCD) screens, and although what they provide is not necessarily a physical sort of feedback, a small screen is an important

element of many small devices because it lets you return data that is more complex than an analog range or a digital value. Although representing "Hello World" in sound is a very interesting exercise, it's often a lot easier to simply get an LCD screen and print the characters to it. Moreover, a small screen can be held in the hand, attached to another object, and embedded in the environment in a way that provides supplementary information to a user.

There are many kinds of LCD screens and no foolproof way to communicate with all of them. With that in mind, we'll look at one of the more common types of LCDs that has an Arduino library. LCD panels that are controlled by the Hitachi HD44780 LCD controller chip or equivalent can be used with the LCD Interface library, you can check which chip a LCD uses online. This library has methods to initialize the screen, print characters and strings, and clear the screen, making your coding substantially easier. The library is included with the Arduino IDE, so to get started, simply import the library using the Tools tab in the IDE, choose Import Library, and then choose LiquidCrystal. Figure 11-22 shows the next example.

Figure 11-22. A simple 16 × 2 LCD screen, a 20 × 4 LCD, and a Serial Miniature OLED

The LCD Interface library provides five main methods:

`void clear()`
: Clears out anything shown in the LCD screen.

`void home()`
: Sets the cursor back to the beginning of the display.

`void setCursor(int row, int column)`
: Sets the cursor to the position indicated by the row and column. So, in a display with 2 rows of 16 columns, to set the 19th character or the character in the 4th position of the 2nd row, you would use this:

```
setCursor(1, 3); // rows and columns start from 0
```

`void write(byte value)`
: Writes the character to the current cursor position.

`void command(byte value)`

This method is a little trickier, but for the ambitious among you, it will be interesting. The HD44780 chip defines different commands that you can use to do things like set the display in and out of display mode quickly, control whether the cursor blinks, and so on.

Table 11-1 shows how the Arduino pin maps to an LCD using the HD44780 chip.

Table 11-1. Arduino to LCD Panel Chip Pin Map

Arduino pin	Pin on the LCD
2	14 (DB7)
3	13 (DB6)
4	12 (DB5)
5	11 (DB4)
6	
7	
8	
9	
10	Enable
11	Read/Write (RW)
12	Register Select (RS)

Keep in mind, though, that the pins might not be in the same order, so you'll need to check the datasheet. Figure 11-23 shows how the HD44780 LCD screen is connected to the Arduino controller.

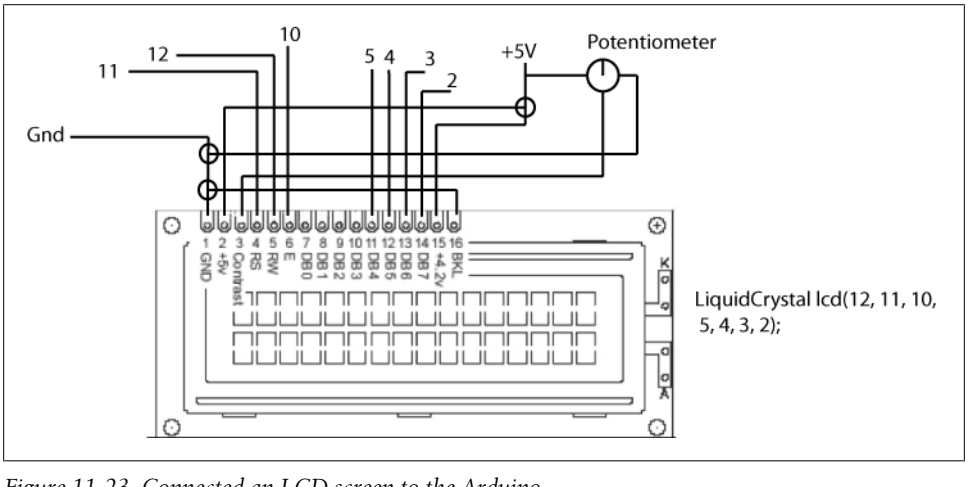


Figure 11-23. Connected an LCD screen to the Arduino

To control the contrast of the display, you will need to connect a 10k (or 20k) potentiometer to provide the voltage to LCD pin 3 because without the correct voltage on this pin, you may not see anything displayed. As shown in the figure, one side of the potentiometer connects to Gnd (ground), the other side connects to Arduino +5v, and the center of the potentiometer goes to LCD pin 3. Many LCD screen have an internal lamp called a *backlight* that illuminates the display. The datasheet for your LCD screen should indicate whether the backlight needs a resistor. Many do; you can use 220 ohms or so if you are not sure.

This allows you to use the LiquidCrystal library that is included with the Arduino distribution:

You have two different versions of the constructor available. One lets you pass the pins that will be used for all seven control pins:

```
LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)
```

Another lets you just pass four pins:

```
LiquidCrystal(rs, rw, enable, d4, d5, d6, d7)
```

Both options use the following three parameters:

`rs`
: Specifies the number of the Arduino pin that is connected to the RS pin on the LCD

`rw`
: Specifies the number of the Arduino pin that is connected to the RW pin on the LCD

`enable`
: Specifies the number of the Arduino pin that is connected to the enable pin on the LCD

Whichever constructor version you decide to use, make sure that those pins are correctly attached to the LCD device. Now, to test, you could do the following:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 10, 5, 4, 3, 2);

void setup()
{
    lcd.print("I'm in an LCD!");
}

void loop() {}
```

You should see the words “I’m in an LCD!” printed to your screen.

Serial LCD

Another option for working with LCD screens is the Serial LCD style. Some LCD screens can be controlled simply by connecting them to the Arduinio's RX and TX pins, which are the digital 0 and 1 pins on your controller. The Serial LCD works by having certain command bytes that indicate that the next byte will be a message. For instance, sending 0x01 over the Serial port to the LCD might clear it, but sending 0x10 would move the cursor left one space.

To communicate with the Sparkfun LCD, you can use the SLCD library, written by Ian McDougall, which makes working with LCD screens a great deal easier. Different manufacturers use different commands. They aren't standardized, so you'll need to check the datasheet for your board if you're using a different one. To use this library, download the SLCD library from the Arduino site, and place it in the *hardware/libraries* folder of your Arduino application. Next, you'll need to load the following code onto your Arduino:

```
#define numRows 2
#define numCols 16
```

Pass the number of rows and columns to the library:

```
SLCD lcd = SLCD(numRows, numCols);

void setup()
{
```

Now, call the `init()` method to initialize it:

```
    lcd.init();
}

void loop()
{
    lcd.clear();
 // can use string, line, col
    lcd.print("Help I'm stuck", 0, 0);
    // or line, col, string
    lcd.print(1, 11, "in an LCD!");
    delay(500);
    lcd.clear();
 // can use string, line, col
    lcd.print("Just Kidding", 0, 0);
}
```

As you can see, the `print()` method writes characters to the screen, and the `clear()` method erases the screen. The SLCD library also defines a few more methods for the Serial LCD:

`brightness()`
: This takes an integer to set the brightness of the display.

`lcd.flash(5, 500)`
: The first parameter sets the number of times the screen should flash, and the second is the length of time in milliseconds that the screen should flash.

`vscroll(numCols, 100)`
: This scrolls the text on the LCD screen to the right or left depending on whether the number passed is negative or positive.

`setCursorPosition()`
: The LCD screen has a small cursor that can be shown at the cursor position. This is especially useful if you want to prompt the user to enter something.

The Serial LCD enables a few other methods. You can find out more about these either by checking the datasheet available from an electronics supplier or by checking online.

Another option for working with the Serial LCD is to use the SoftwareSerial libraries. If you want to connect multiple LCD screens to your Arduino, take a look at Chapter 15 where we discuss these different approaches to sending and receiving serial data.

You can use the LCD screen in a few different ways; since it's small, it can be embedded in touchable devices to send messages or can be put in plain view in an interface. Since the LCD is such a common way of providing feedback, to make it noteworthy or playful, you'll need to use it in a novel fashion, such as embedding it in novel location or using its lightness and relative thinness to your project's advantage. Of course, you can always use an LCD screen in a simple and straightforward manner: to provide simple textual feedback to a user without requiring a large screen that consumes more power.

Using Solenoids for Movement

This section requires that a few terms be defined for you first, so let's start at the beginning.

A *solenoid* is a coil of wire with a magnetic core and usually with a rod resting inside that coil of wire. It works by sending a current through the wire that either pushes or pulls the rod, depending on the type of solenoid. When the current triggers the solenoid, the rod moves in its primary direction, either pushing if it's a *push solenoid* or pulling if it's a *pull solenoid* (see Figure 11-24). The names simply indicate in which direction the magnetic field tries to move the solenoid. When the current is off, then the rod in the center of the solenoid returns to its resting position. Solenoids are used frequently in machines, in automobile transmissions, and in robotics.

Diodes are used to ensure current flows in only a single direction, acting almost like a valve that lets water flow in only one direction (see Figure 11-25). Diodes use a small amount of voltage to operate, typically 0.7V, so a diode receiving the voltage from a 7.2V battery would reduce it to 6.5V. They are used for many purposes in more advanced electronics, but the most common use for beginners is to protect a microcontroller from "noise" that could destroy the microcontroller or interfere with other

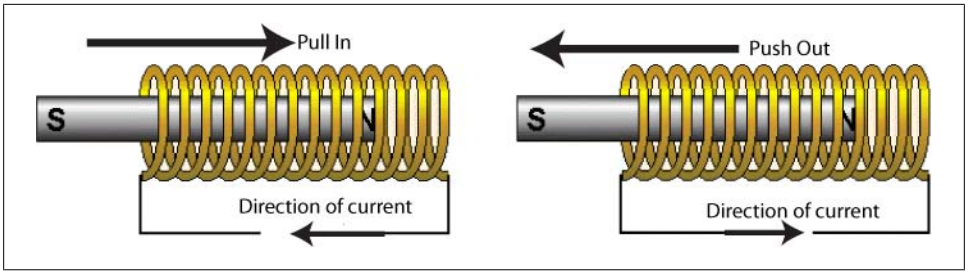


Figure 11-24. How a solenoid functions

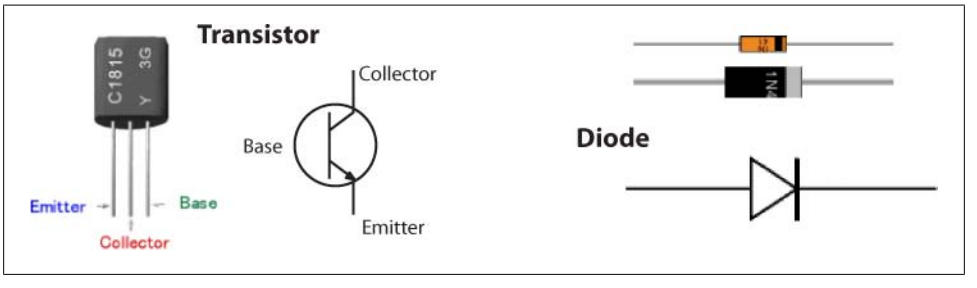


Figure 11-25. Transistors and diodes

components. Diodes called *Zener diodes* are also useful for dropping high voltages to a lower voltage.

Transistors are, at the moment, probably the important electrical component ever invented. They may even be one of the most important inventions of all time (see Figure 11-25). Almost everything that is electronic probably has at the very least one of them and, in the case of your computer, probably millions, if not billions, of them (in the digital world). A transistor is not much different from a simple mechanical on/off switch. Instead of flipping a switch, a signal is sent to the transistor telling it to connect the circuit that runs through it from your microcontroller.

When you send 0 volts to the base of the transistor, the collector is turned off, and when you apply a signal of 5V, the transistor collector is turned on, which is why it's like a light switch. Transistors can be used to control things other than lights. By sending a small signal, you can control huge flows of water through a pipe. In the case of an Arduino board, the transistor controls a larger flow of current by sending a digital on or off signal from one of the digital pins on the board. With the solenoid, the transistor will be used to power the magnetic coil of the solenoid to push out. The Arduino controller wouldn't have enough power on its own to power up the solenoid, but by feeding the current through a transistor, you can use the small 5V signal of the Arduino to control the 12V solenoid.

The circuit that you'll be building to control the solenoid looks like Figure 11-26.

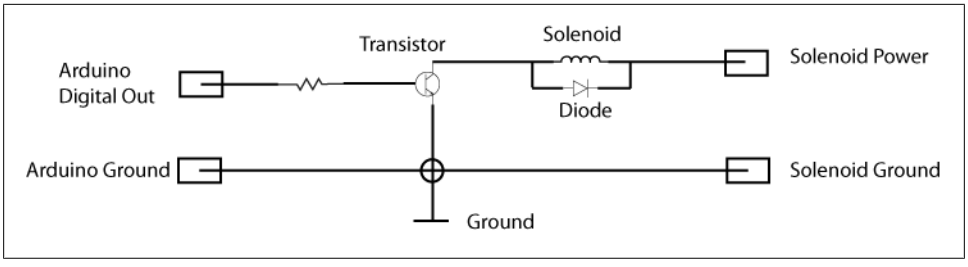


Figure 11-26. A circuit to connect a Solenoid to the Arduino

The solenoid power is going to depend on the solenoid that you're using. A 12V solenoid will require a 12V power source. You'll also want to make sure that the diode is "facing" the right direction, that is, that power from the solenoid power source is not flowing into your Arduino, because the solenoid won't work and the solenoid power source will be shorted when the transistor is turned on.

Another way to do this is to use a low power relay switch, like the one shown in Figure 11-27 on the left.

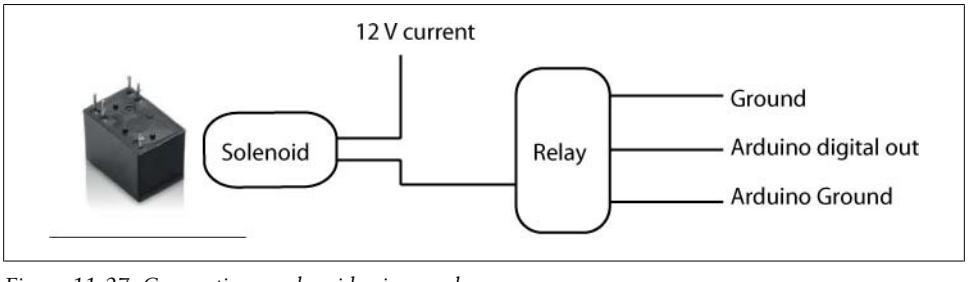


Figure 11-27. Connecting a solenoid using a relay

The relay functions in much the same way as the transistor and diode but integrates the two devices into a single unit. The relay that you use will need to be rather specific because it needs to have a coil current of 40 milliamps and a coil voltage of 5V. Some models you can investigate are the JQ1AP-5V or the Panasonic Electric Works TQ2SA-L2-5V. You can now connect your solenoid to your Arduino in the way shown on the right of Figure 11-27.

Since solenoids can be used to push, one possibility is to use them to create a drumbeat of sorts. You can easily do this by purchasing two solenoids and attaching one of them to a cymbal and another to a snare drum with some tape or with small clamps. The beat would be defined in two arrays of Boolean values to save space. Your code could be as simple as this:

```
boolean hihats[] = {1, 0, 0, 0, 0, 1, 0, 1};
boolean snare[] = {1, 0, 0, 0, 1, 0, 0, 0};

int hatPin   = 2;
```

```
int snarePin = 3;
int counter;

void setup() {
    counter = 0;
}

void loop() {

    digitalWrite(hatPin,   hihats[counter]);
    digitalWrite(snarePin, snare[counter]);

     if(counter == 7) {
        counter = 0;
    } else {
        counter++;
    }
    delay(200); // delay between beats
}
```

You could, of course, make your rhythms much more complex and, if you wanted, use as many relays and solenoids as you have digital pins on your Arduino. Another option is to have a button control a solenoid, making user-driven controls. Yet another option is to drive the solenoids with a Processing or oF application by sending data over the Serial port. This, coupled with the computer vision techniques in Chapter 14 could allow you to create a remote virtual drumming machine. Or, coupled with the network communication techniques from Chapter 12 it could enable a user to remotely drum or control any other kind of movement. The difficult part of working with a solenoid is not the programming; it's getting the power and wiring right. However, as mentioned earlier, using a relay switch can make that quite a bit easier for you.

What's Next

This chapter mentioned a few different topics that you can explore further. Mechanical engineering is a vast field that many of the devices here have touched on only just briefly. Working with several stepper motors or servos requires some engineering thinking to ensure that equipment doesn't fail or destroy other parts of the project. For instance, using multiple servos to create complex motion requires some careful planning; controlling multiple appliances or motors also requires a fair amount of planning and research. That said, there are plenty of resources to help. There are a few good introductory texts on robotics that can be of great help. *Practical Electronics for Inventors* by Paul Scherz (McGraw Hill), *Robot Building for Beginners* by David Cook (Apress), *Getting Started in Electronics* by Forrest M. Mims III (Master Publishing), and *Physical Computing* by Tom Igoe and Dan O'Sullivan (Course Technology) are all resources that will help you. Also, the Arduino website contains a wealth of information in its Playground section. The Arduino community is also very prolific in publishing books, tutorials, and libraries to make working with components easier. Another excellent book

for learning more about hardware, electronics, and components is *Designing Embedded Hardware* by John Catsoulis (O'Reilly).

You might also be interested in taking a look at some of the more far-out projects that people in the Arduino community have been exploring. Some of my favorites are the RepRap Research Foundation's self-replicating machines; the Asurino library and development project, which uses the Arduino to control the Asuro hobby robot; and the *Braitenberg Vehicle*, the light-seeking robot designed by Alexander Weber. Another fascinating breed of projects that has emerged in the Arduino community is research into unmanned autonomous vehicles (UAVs). There are already several small projects underway to provide componentry and drivers for UAV projects built with Arduino controllers, among them the ArduPilot. There are also non-Arduino UAV platform kits available if the idea of creating things that run around is interesting.

Review

DC motors come in several different kinds. Brushed DC motors, for example, are composed of two magnets and a core that spins as the voltage around the core changes. To reverse a DC motor, you need to be able to reverse the direction of the current in the motor. The easiest way to do this is using an H-Bridge circuit.

In addition to buying an H-Bridge circuit, you can purchase several kinds of driver boards that will allow you to control motors with prebuilt kits. Several major suppliers have created different boards for running motors.

Another kind of motor is a stepper motor. Instead of spinning in one direction or another depending on the current, a stepper motor advances to predefined steps. Stepper motors can provide more torque than a servo. There are several different libraries, most prominently the Stepper library, included with the Arduino IDE that can be used to control stepper motors. As with DC motors, there are also driver shields that are ready to be fit onto an Arduino controller.

A servo allows for programmable motion. It receives a signal of varying pulse position to determine how far to rotate within a limited range. The most common range for a servo motor is 180 degrees.

You can use the Servo library or control the servo using pulse width depending on which is more appropriate for your application.

To use household currents, you can either use a solid-state relay, or you can use a device like the RelaySquid from LiquidWare. It's very important to check your circuit carefully when working with high voltage because you can hurt yourself or damage your electronics.

Using the Vibe Board in conjunction with the LilyPad is another interesting way to provide feedback because it lets you send small vibratory signals to a user from a small controller that can be sewn into clothing or carried.

An LED matrix is an array of either 5 × 7 or 8 × 8 LED lights. They can be controlled using a chip such as the MAX7721. To control a single LED matrix, you can use the Matrix library; to control up to eight LED matrices, you can use the LedControl library.

The `spi_transfer()` method is used to send and receive data from another microcontroller or peripheral using the Serial Peripheral Interface protocol. This is a low-level protocol that lets you set other devices to be the slave devices for the Arduino controller, which will act as the master device.

LCD devices can be controlled by the LiquidCrystal library. This lets you write to the LCD screen using the `print()` method and lets you clear the screen using the `clear()` method. Most LCDs also support using cursors and scrolling.

Solenoids are magnetic devices that can push or pull depending on the kind of solenoid it is. These can be used in many different kinds of mechanical devices and are quite common in electrical engineering and robotics.

You'll need to use a transistor and a diode to control a solenoid, or you can use a relay switch to control whether the solenoid is powered on or off.

CHAPTER 12
Protocols and Communication

Throughout this book, we've explored data and different conceptions of what data is and how it is understood in an application. A user gets data from an application that helps them understand what the application is doing, how their actions have been interpreted, and how to get the application to do what they want it to do. This data is called the *feedback*. The data that the user sends to the application is called the *input*. The input tells the application what the user wants it to do, how it should do it, and what kind of feedback to provide. In this chapter, we'll focus on a different sort of exchange of data: exchanging data between an application and devices. This isn't new; you've already looked at many different components that work with the Arduino controller: receiving commands from the Arduino, sending data to the Arduino, using your computer in tandem with the Arduino, and creating an exchange of information and commands between the two of them. What you haven't looked at yet are the ways that communication between different machines and applications depends on the context, the means of communication, the machines used, and the amount of data being exchanged.

Cell phones, communication over the Internet, and devices plugged into your computer all rely on protocols that dictate how devices communicate with one another. Enabling your devices to communicate with other devices is really a matter of ensuring that your system knows the protocol that the other devices speak. Think of it like a conversation between two people: they have to be speaking the same language to know what the other one is talking about. Without those protocols established, a conversation can't occur, and your application and the devices it needs to communicate with need the same kinds of protocols. There needs to be an understanding of what is being communicated and how that communication is being encoded.

When you allow your application or applications to communicate with one another, communicate with other devices, and communicate over common protocols, you enable your applications to work remotely, get information from remote locations, and send information to other devices, applications, and locations. You can send commands to a remote server, save data remotely, gather data from the Internet, network multiple devices and machines, communicate with other types of machines like MIDI keyboards, send text messages, and even make phone calls.

How does this affect interaction? The interaction of your designs doesn't always need to be a matter of the user interacting with the interface of your application. In fact, in most interactive applications, a great deal of the communication goes on between the application and other devices or applications. Most of this communication is governed by protocols. However you use the input or data, the important thing when using a protocol is making sure that any input is correctly translated into the protocol and that any data coming in can be correctly translated from that protocol. Users probably don't care how you're getting the data, and often the format of the data won't be one that your users will want or be able to understand themselves.

What kinds of data can you receive? That ends up being a hardware question. Without a something like a microphone, you can't receive sound data. Without a network interface you can't receive data over the Internet. So, you'll have to rely on hardware to provide access points between your system and the external world.

Think for a moment about the data that you receive from an accelerometer, namely, the x, y, and z data that reports movement of the accelerometer. If the only thing that you need to do with that data is send it to the Arduino controller, then you don't need to convert the data; you can use it as received and be done with it. If you want to have that accelerometer control MIDI commands, however, then you'll need to not only process the data but also figure out a meaningful way to turn it into a different protocol, the MIDI protocol. If you want to send that information to a remote server and turn it into a visualization or another transformation of data, then you'll need to understand the protocol that the server uses and the program that you're using to provide the visualization or other transformation. A great deal of effort in computation is spent ensuring that data is correctly converted from one format to another and from one protocol to another.

The translation of data is more than a hidden computational matter. It can be relevant, interesting, and engaging for your users. A gesture can become a signal over the Internet. A sound in a remote location can become a mechanical motion in an installation. A command sent from a mobile device can become a message written on the side of a building.

In this chapter, you'll learn about communicating over networks, both local networks and the Internet; communicating over the Bluetooth protocol with cell phones and other Bluetooth devices; and communicating with MIDI devices.

Communicating Over Networks

Interaction designers of all types have been excited about the possibilities of networked communication for years. Since networked communication became affordable, accessible, and widespread, artists, designers, engineers, and dreamers of all types have experimented with the possibilities of remote data, remote access, and remote control. Nothing expands the place or location of an application like spreading it across the globe via a network. Some of the most innovative and thought-provoking computer-aided artworks from the 1990s brought the possibilities of networked communication into sharp focus.

Now designers and artists can use networks to send a video feed across the Atlantic Ocean as in the *Telectroscope* or mirror public benches in locations across the world as in the *World Bench* project by GreyWorld. Game designers like Area/Code can use networked mobile devices to create real-time locative games that track a user's location through their cell phone signals. Ken Goldberg's classic *Tele-Garden* installation is a garden that is controlled remotely by thousands of remote gardeners from all over the globe who operate robotic arms that arrange, light, and water the plants. Projects like Heath Buntings Border Crossings can use the geolocation of the user to provide or deny them access to certain material on a site. Another project that uses the power and idea of networking is *Screening Circle* by Andy Wilson that creates shared space where multiple users can create images together in a sort of sewing-circle collaborative effort. Another common use of networking and networked capabilities is to mine the Internet for data. Jonathan Wilson and Sep Kamvar built a project called *I Want You To Want Me* that mined data from online data sites to create beautiful representations of how users of online data sites thought of themselves and what they wanted from a prospective partner. All these projects use networks and networking in different ways, but all communicate with the outside world and outside applications.

Networks can be used to cover greater amounts of space between parts of an application by spreading an application over multiple machines, to gather data from another source, to allow access to users who aren't physically present, to allow users to connect with one another, or to pass processing off to more powerful computers. At this point, the question is not so much "What is networked?" as "What isn't networked?" Most applications in some way or another use information from the Internet, whether to gather data, check whether a serial number is valid, enable searching, or allow easy communication. A rapidly increasing number of devices network as well, such as laptops, cell phones, and PDAs. All are beginning to blend into one another as the need for computing power shrinks and the reliance on other networked computers for information or processing increases.

For you the designer or artist, adding networking capability to your application is much easier with the tools that are provided in oF, Processing, and Arduino. The hardest part of networking and networks is making them efficient and fast, but if you're making small-scale projects, they may not need to be fast or efficient in the way that a large,

powerful site needs to be. That's not to say there are not new things to learn, tricky concepts to understand, and lots of new things to get your head around. In this chapter, you'll be learning how to send and receive data from the Internet, send data to other applications that are local or remote, read wireless Internet signals, and use MIDI signals to communicate between applications or over a network.

Using XML

Extensible Markup Language (XML) is a way of giving a document structure. It's important because XML is very commonly used by Internet web services like Twitter and RSS feeds, and it's also a very common way to store and send data. Chances are that at some point you'll want to get some data from another source and that the data will be in XML format. Things that you might want to represent as a piece of data have a few common characteristics. They have some specific *traits*, they *have* things that belong to them, and they *belong to* other things. That's a little overly abstract, so think of a library:

- *Characteristics:* street address, name
- *Has:* books, magazines, DVDs
- *Belongs-to:* library system, street, city

XML is a way to represent a structure in a document that is standardized so that any application can read it. An XML document has to be well formed. This means that every opening tag `<tag>` must have a corresponding closing `</tag>`:

```
<book>Wuthering Heights</book>
```

The following is generally called a *node*, a single object within the document that has some associated data, and this node is a well-formed one because the opening and closing tags match:

```
<book>Wuthering Heights</magazine>
```

That is not a well-formed node, because the opening and closing tags don't match. Some nodes don't have a closing node and those are indicated like this:

```
<DVD/>
```

Usually a node that consists of a single tag will have attributes to describe its data. I'll discuss them later in this section. You can also have multiple nested nodes, like this:

```
<library>
    <book>
        <title>Great Expectations</title>
        <author>Charles Dickens</author>
        <publication_year>1883</publication_year>
    </book>
</library>
```

In this case, the library *has a* book and the book *has an* author and a title. This idea of ownership is expressed by nesting the nodes within one another.

XML documents sometimes have a declaration at the top of the document that looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

This tells whatever is parsing the XML document what version of the XML specification is being used and what encoding to expect. If you're writing an XML document that one of your own applications is going to read, then you probably don't need to put a declaration at the top of it. If you're planning on sending the XML document to a service on the Internet to be read, then you probably do want to add one at the top of your document to make sure that the other application reads it correctly. XML also allows you to add comments into the document, just like in any other code. XML comments start with `<!--` and end with `-->`:

```
<!-- This is a comment. -->
```

Two consecutive dashes (`--`) may not appear anywhere in the text of the comment, or the program reading the XML will most likely throw an error:

```
<!-- This comment -- will throw an error. -->
```

In addition to the values contained within the element, XML elements can have attributes as well:

```
<state id="0">
<name>Ohio</name>
</state>
```

The `id` property here gives another value to the state that you can use for sorting or accessing the names more quickly than looking through all the names. Using IDs or other values in an XML node is a useful way to add extra data that describes the node but maybe doesn't need a whole other child node. A common use for this is to add an `id` property. This is another example:

```
<site id="1" url="http://programminginteractivity.com">
    Programming Interactivity</site>
```

So, what kinds of things can you store with XML? All sorts of data: dates, places, ZIP codes, cities, states, and statistics of all sorts. XML is also a very common format for websites to return data. If you request something from another website, a series of photos from Flickr, stock quotes, or the weather, the results are often returned from websites or from programs that have a publicly available service called a *web service*. In some cases, XML is used to store data on a computer as well as for an application to load configuration files.

Processing contains an XML library to make creating and parsing XML easier. This functionality is contained in the `XMLElement` class; it contains methods for loading XML files, creating new XML files, looking through data for specific pieces of information, or setting new values in an existing XML file or dataset.

The functionality of this class provides a good explanation of how you might work with XML data in an application. To load an XML file from the home folder of a Processing application, you would create a new `XMLElement` object as shown here:

```
XMLElement xml  = new XMLElement(this, "xmlfile.xml");
```

Let's take a look at a simple XML file of three states:

```
<?xml version="1.0" encoding="UTF-8"?>
<states>
<state id="0">
    <name>Ohio</name>
</state>
<state id="0">
    <name>Michigan</name>
</state>
<state id="0">
    <name>Indiana</name>
</state>
</states>
```

To find out how many states are in the file, call the `getChildCount()` method. In this case, that method will return 3, since there are three state nodes. The following code snippet gets the number of nodes and loops through them:

```
int stateCount = xml.getChildCount();
for (int i = 0; i < stateCount; i++) {
    //do something with each node
}
```

To get each state node of the XML file, call the `getChild()` method inside the `for` loop:

```
for (int i = 0; i < stateCount; i++) {
    XMLElement firstChild = xml.getChild(i);
}
```

Notice in the previous code how both the main XML document and the individual `<state>` node of the file are represented by `XMLElement` objects. The value returned by the `getChild()` call is another `XMLElement` object. This means that the `getChild()` method can be used to get any children of the `<state>` node as well. In the case of the state node in the example XML file, the first child will be the name. You would grab the name of a `<state>` node as shown here:

```
String name = firstChild.getChild(0).getContent();
```

The `getContent()` method returns the content of the node, that is, the data inside. In the case of the name of the state, it looks like so:

```
<state id="0">
<name>Ohio</name>
</state>
```

So, looking at the previous snippet, the content of the name is "Ohio":

```
String name = firstChild.getChild(0).getContent();
```

So, the `<name>` node, which is the first child of the `<state>` node, is returned by the `getChild()` call; then the content of that node, which is the value inside the XML tags, is stored as a separate string.

To get the `id` of the state node, you need to read the attribute of the node. The `XMLElement` class has three ways to read an attribute: `getFloatAttribute()`, `getStringAttribute()` and `getIntAttribute()`. If you want the value of the attribute as a float number, use `getFloatAttribute()`. If you want the value of the attribute returned as a number so that you can compare it numerically, sort it, or perform other operations that aren't available with a string, then use `getIntAttibute()`. If the attribute is a name, a URL, or another character-based data, then use the `getStringAttribute()` method:

```
int id = firstChild.getIntAttribute("id");
```

You'll put this library to use in the next section when learning how to communicate over a network with a Processing application.

Understanding Networks and the Internet

Network can mean many things: two computers linked together, hundreds of computer linked together, computers attached to the Internet, machines connected to a wireless router, and so on. All networks have a few things in common, though:

- Machines on the network need to be identifiable to each other and to themselves.
- Machines need to know how to connect with one another.
- Machines need to know what protocol the other machines are using.

Network Organization

There are few different ways of organizing a network. Figure 12-1 shows some of the most common setups that you might use.

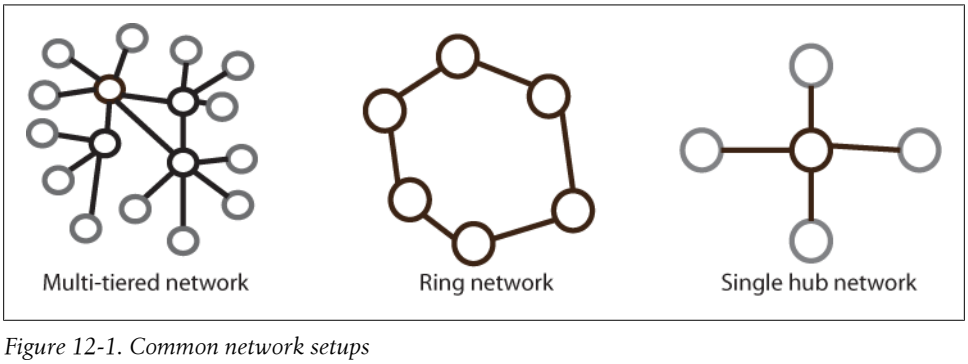


Figure 12-1. Common network setups

You've already used a sort of hub network when connecting multiple devices to your Arduino controller or when connecting your Arduino to your computer. Other examples of hub networks are Bluetooth-enabled networks (which you'll learn about later in this chapter) and a group of computers connected to a single central server that sends commands out to each of the nodes on the networks. In a *ring* network, any message has two possible paths, but it also means, in a worst-case scenario, that to get a message from one machine to another, you might have to send $n/2$ messages where n is the number of machines in your network. One simple example of a ring network is a group of Arduino controllers connected together. A multitiered network is just a collection of single-hub networks that have ways to determine which of the central nodes is acting as a hub to the desired child node. This is how the Internet is set up, with large relays bouncing traffic to its intended destination.

Network Identification

Over a network, all the machines are identified by their Internet Protocol (IP) addresses. Most machines that connect to the Internet also use the Ethernet protocol. Finally, many devices have a Media Access Control (MAC) address that is burned into the devices memory. This is similar to the way that devices using the I2C protocol (that you learned about in Chapter 7) have an address that is always associated with that object.

You've probably seen your computer's IP address at some point. On Mac OS X you can see it by opening your Preferences window and opening the Network pane. On Windows Vista you can see it by going into your Control Panel, then selecting Properties and then Manage Network Settings. You'll see an IP address that might look something like this:

```
192.168.0.25
```

That number is the identifier of your computer, with four sets of numbers, each from 0–255. The address shown here is a bit tricky to explain because it shows an IP of a computer that is connected to a router. The value `192.168` at the beginning of the IP address indicates the mask and subnet of a local network. That means this number won't mean anything to another computer *outside* your network, but to another computer *inside* your network this value specifies your computer on the local network and the router will send information destined for that address to your computer. A computer that is publicly available outside of a local area network might have an IP address like `208.75.87.131`.

Usually you won't be typing in IP addresses to get to the server and website that it hosts, usually you type in something like *http://programminginteractivity.com* to get the server with that IP address. There's a separate protocol called the Domain Name System that is used to convert names into numbers. Domain Name System servers take the domain name and convert it to the IP address that will route the traffic to the correct server. Why do you need to remember all these acronyms? In short, when you send information

over the Internet, whether you know it or not, you're using these protocols to send your data. You won't need to worry a great deal about them and how they work, but it helps to conceptually understand what's going on when you are configuring your application to send or receive data from another networked machine.

So, what is the data being sent over the Internet? In short: packets. *Packets* are your data, broken into small pieces for easier sending. A way to see this at work is the `ping` command. Open a terminal, and type `ping` followed by the name of a web server. You'll see something like the following:

```
64 bytes from 208.75.87.131: icmp_seq=1 ttl=47 time=60.217 ms
64 bytes from 208.75.87.131: icmp_seq=2 ttl=47 time=66.809 ms
64 bytes from 208.75.87.131: icmp_seq=3 ttl=47 time=70.654 ms
64 bytes from 208.75.87.131: icmp_seq=4 ttl=47 time=62.126 ms
```

This says that 64 bytes were sent to IP address 208.75.87.131 and that the round-trip was on average around 63 milliseconds. You'll learn more about packets later, but for right now, just know that most messages you send will be broken into multiple smaller packets that will be sent to the server or client your application is communicating with.

When you type the `ping` command, you put the name of a host that your computer is going to connect with. This hostname can be either an IP address (208.75.87.131) or a domain name (programminginteractivity.com). There is one reserved IP address and hostname that it's good to know about: 127.0.0.1 or localhost. This is your computer, the local machine. You'll notice that if you type `ping localhost`, the return times will be extremely fast, since the computer is just pinging itself. When you're building clients and a server, it's very helpful to have a local version of the server to test on. Usually accessing this will involve using either the 127.0.0.1 or localhost, and it might involve putting a port number at the end so that the server name looks like this: 127.0.0.1:8080.

Any host has many ports that a client can connect on. For instance, HTTP web traffic goes over port 80. File Transfer Protocol (FTP), used for uploading and downloading large files, uses port 21. All the port numbers between 0–1023 are reserved for predefined uses, so sometimes you'll see port numbers like 8080 or 4000 appended to the end of the IP address. You can think of the port as a way to ensure that all the messages for a certain type of traffic are grouped together.

Network Data Flow

So, how does a network request work if you want to get an HTML file, for example? Let's say that your application is running on your local computer that is connected to the Internet through a wireless router, which is a fairly common setup. First, your application sends its data to the router. The router sends your message to your network provider. The message is routed to the correct network provider for the server you want to make a request from. Finally, the server will receive the request for the file. This should give you an indication of why so many different protocols are needed for Internet communication.

Handling Network Communication in Processing

Communicating over a network is a very important capability in Processing. Your applications can communicate with other Processing applications, download data from the Internet, send data to be stored or processed remotely, and even peek into data being sent wirelessly. This is all done through the Network library. The Network library enables you to easily create clients to read data from other servers on local or remote networks as well as servers that can send data to remote clients or to other clients on a local ring network. As with many networking libraries, there are two classes: the `Client` class and the `Server` class.

Client Class

The `Client` class is used to create client objects that connect to a server to exchange data. To create a client within a Processing application, import the Network library by including `import processing.net.*` at the top of your application, call the constructor, and pass in a server name that you would like to connect to and port number that you want to use:

```
import processing.net.*
Client networkClient = new Client(this, "www.oreilly.com", 80);
    // Connect to server on port 80
```

The next most important method to understand is the `write()` method. This allows you to make requests to a server. The most common method to send to a server is the `GET` command, which tells the web server that you're simply asking for a web page. If you're curious about other commands, look around for information on Representational State Transfer (REST), and you'll learn about the `POST`, `PUT`, and `DELETE` commands. For the purposes of this introduction, the `GET` variable is perfectly sufficient. Here's an example of writing a `GET` request:

```
networkClient.write("GET / HTTP/1.1\r\n");
```

This starts the request to the server, but it's not enough. Most servers won't respond to this because it's not a complete request. You also need to add a server name:

```
networkClient.write("Host: oreilly.com\r\n");
```

The "Host:..." is required for reasons that are a little more complex than we need to get into here. If you want to read more about HTTP, check out the HTTP specification at *www.w3.org*, the home of the World Wide Web Consortium, which writes and maintains all the web specifications. You can write as many lines as you need in the request, breaking it up over multiple calls or all as one call without affecting the request that the server will receive.

After you send the request, you'll want to receive the response from the server. There are two methods that you'll use to do this: the `available()` method that returns how

many bytes are being sent to the client and the `readString()` method that returns all the data as a string:

```
if (networkClient.available() > 0) { // If any incoming data
    String data = networkClient.readString(); // store as a string
}
```

The drawback of the `available()` method is that it requires that you poll it constantly. If you're expecting a lot of data from a network connection, then this makes sense; otherwise, it can be excessive. If the communication between the network and your application is going to be infrequent, you might want to add the `clientEvent()` event handler for the network events into your application to have a separate method where you can store all your logic for handling communications from a client:

```
// the ClientEvent is triggered whenever the server sends data to a client.
void clientEvent(Client client) {
    String dataIn = client.readString();
    println("Server data is:: " + dataIn);
    // do something else with the data
}
```

Server Class

The `Server` class is used to create server objects that can send and receive data to and from any client connected to it. You create a new server object by passing the port number that the server should be available on:

```
Server srv = new Server(this, 5204);
```

You might want to make your server available on port 80, the standard HTTP traffic port, but you'll probably find that your computer doesn't like that. That's in all likelihood because another service is already on port 80. The simplest thing to do is to set a different port number, but if you want to use port 80, or another port number lower than 1023, and you run into problems, check on the Processing forums for instructions on how to get this working.

Once you've gotten the server started, you'll want to listen for any client that connects up to it. You can do this by using the `available()` method, which returns the clients that are connected to this server:

The `available()` method returns a client with updates, and if no clients have updates, then this returns null. The `available()` method can be done in the `draw()` loop of your application; when it returns a value that is not null, use the `readString()` method to read any data sent from the client:

```
void draw() {
Client client = myServer.available();
      // is there a current client?
if (client !=null) {
String clientMsg = thisClient.readString();
    if (clientMsg!= null) {
      //do something with the data
```

```
    }
}
```

Just like the `clientEvent()` callback, the `serverEvent()` callback can be used to handle any new client connections to any server within the application:

```
// called when a new client connects to one of the servers in the application
// the server and the client will both be passed in to the method
//     by the Processing framework
void serverEvent(Server server, Client client) {
    print("New connection from IP :: " + someClient.ip());
}
```

Now, by using XML parsing within a client-server setup, some reasonably complex data can be exchanged. The following example has components. The server loads an XML file from Flickr (the popular photo-sharing site) using the Processing `XMLElement` object, which parses the XML into a simpler format and then sends all of its clients the URLs of the pictures, along with the x, y locations to display the pictures:

```
import processing.net.*;

Server fsSrv;
XMLElement flickrXML;
XMLElement picList;
PFont myFont;
String lastmsg = "";

void setup() {

    size(600, 300);
    fsSrv = new Server(this, 8180);
    flickrXML = new XMLElement(this,
    "http://api.flickr.com/services/rest/?method=flickr.photosets.getPhotos&api_key=
        0367b4b7b7ab07af3c04d8d6d839467d&photoset_id=72157594290642861");

    XMLElement pics = flickrXML.getChild(0);
    picList = new XMLElement("<pictures></pictures>");
    // you could use them all, or only use 4
    int totalPics = 4;//pics.getChildCount();
    int xp = -400;
    int yp = 0;

    for(int i = 0; i<totalPics; i++) {
```

Flickr uses picture URLs that look like this: `http://farm{farm-id}.static.flickr.com/{server-id}/{id}_{secret}.jpg` so you have to build the URL up from the XML file:

```
        String url = "http://farm"+pics.getChild(i).getAttribute("farm")+
            ".static.flickr.com/" +pics.getChild(i).getAttribute("server")+
            "/"+pics.getChild(i).getAttribute("id")+"_"+
            pics.getChild(i).getAttribute("secret")+".jpg";
        if(xp > 399 ) {
           xp = 0;
           yp += 400;
      } else {
```

```
            xp += 400;
          }
```

Now that it's built, add it to the XML file to send to all the clients:

```
            picList.addChild(new XMLElement("<pic id=\""+i+"\" url=\""+url+"\" x=\
                ""+xp+"\" y=\""+yp+"\" />"));
        }
        myFont = createFont("Arial", 16);
        textFont(myFont);
    }

    void draw() {
        background(0);
        text("ok", 10, 20);
        Client client = fsSrv.available();
```

If one of the clients has sent a message, read it, and write it to all the other clients:

```
        if(client != null) {
            lastmsg = client.readString();
            if(lastmsg != null) {
                fsSrv.write("<msg>"+lastmsg+"</msg>");
            }
        }
        text(lastmsg, 30, 20);
    }
    void serverEvent(Server srv, Client clt) {
        fsSrv.write(picList.toString());
    }
```

Next, the client loads all the pictures sent in the file XML:

```
    import processing.net.*;

    Client fsClient;
    XMLElement picList;
    ArrayList picArray;

    void setup() {
        size(800, 800);
        fsClient = new Client(this, "127.0.0.1", 8180);
        picArray = new ArrayList();

    }

    void draw() {
        background(0);
```

If data is sent to the client, then read it using the `readServer()` method:

```
        if(fsClient.available() > 0) {
            readServer();
        }
```

Draw all the images to the screen:

```
        int sz = picArray.size();
        IImage fi;
        for(int i = sz-1; i >= 0; i-- ) {
            fi = (IImage)picArray.get(i);
            image(fi.img, fi.x, fi.y);
        }
    }
```

This next code is called when the client has data that has been sent to it:

```
    void readServer() {
        if(picList == null) {
                picList = new XMLElement(fsClient.readString());
                int totalPics = picList.getChildCount();
```

For each `<pic>` object in the XML, create a new `IImage` object and load the image specified in the URL property of the XML:

```
                for(int i = 0; i<totalPics; i++) {
                        IImage fi = new IImage();
                        fi.setXMLData(picList.getChild(i));
                        PImage p = loadImage(picList.getChild(i).
                             getStringAttribute("url"));
                        fi.img = p;
```

Add it to the `ArrayList` so that it can be drawn later:

```
                        picArray.add(fi);
                }
            }
    }

    class IImage{

        PImage img;
        public int x;
        public int y;

        public void setXMLData(XMLElement s)
        {
            x = s.getIntAttribute("x");
            y = s.getIntAttribute("y");
          }
    }
```

Now that you're finished, you should have something that looks like Figure 12-2.

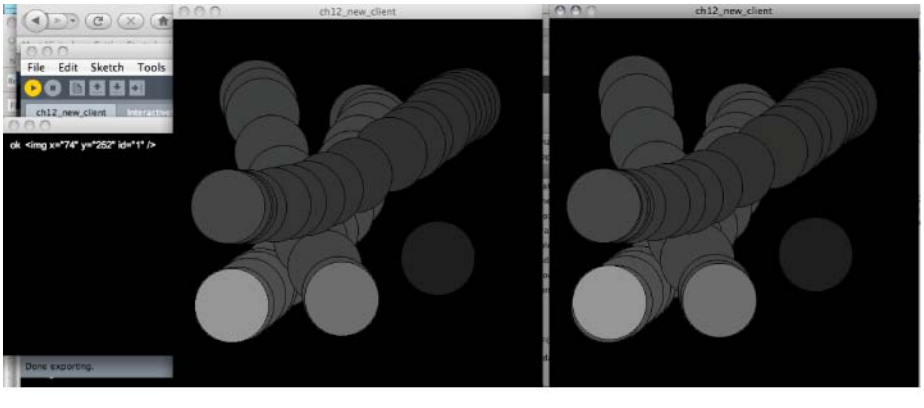
Sharing Data Across Applications

Next, you'll use the same principle in a more dynamic way. The clients will be reactive to the user's actions and will send data up to the server whenever the user changes something in the application. The server will then send the information to all the

listening clients, which will place the images according to the incoming data, as in Figure 12-3.

Since the server sends each application the same data, two separate instances of the application will look exactly the same.

Figure 12-2. Synchronizing two applications



The server prints out the data it receives in its window before passing it on to all the clients.

The two clients are kept in sync via the XML data sent between them as the client moves the objects in each window.

Figure 12-3. Sharing drawing data across applications

Example 12-1 works similarly to previous example. The client opens a connection to the server on a certain port and IP address and listens for any data coming from that IP. The difference is that the server sends the client information only when one of the other clients has changed something and sends the information to the server.

Example 12-1. Server

```
import processing.net.*;

Server fsSrv;
PFont myFont;
String lastmsg = "";

void setup() {
    size(600, 300);
    fsSrv = new Server(this, 8180);
    myFont = createFont("Arial", 13);
    textFont(myFont);
}

void draw() {
    background(0);
    text("ok", 10, 20);
    Client client = fsSrv.available();
    if(client != null) {
```

The `readStringUntil()` method is used here to listen *until* the server receives the end of the message. Before that, it doesn't do anything:

```
            lastmsg = client.readStringUntil('>');
            client.clear();
            if(lastmsg != null) {
```

Once it receives the end of the message, it will send it to all its listening clients:

```
                fsSrv.write("<msg>"+lastmsg+"</msg>"+'\0');
            }
        }
        if(lastmsg != null) {
            text(lastmsg, 30, 20);
        }
    }
```

The client (see Example 12-2) is a little more complex because the client needs to tell the server whether the user has dragged one of the shapes around, and it needs to make sure that it updates only if the information is coming from a different client. Since the server sends information to all clients, without checking to make sure that it hasn't sent a message, the client would be stuck in a loop of updating its data with the data it just sent out. The `Client` uses an instance of the `InteractiveImage` class to draw into, which has been omitted to conserve space, but it's available in the code downloads for this chapter.

Example 12-2. Client

```
import processing.net.*;

Client fsClient;
ArrayList iiArr;

int millisSinceSent = 0;

void setup() {
  size(500, 500);
  fsClient = new Client(this, "127.0.0.1", 8180);
  iiArr = new ArrayList();
```

Using a slower frame rate helps make sure that you're not sending too many messages to the server:

```
        frameRate(10);
        for(int i = 3; i>-1; i--) {
            // check in the downloads for this class
            InteractiveImage ii = new InteractiveImage();
            ii.id = i;
            ii.colr = 30 + 40 * i;
            iiArr.add(ii);
        }
        background(0);
    }

    void draw() {
        fill(0, 2);
        rect(0, 0, 500, 500);
```

If a message comes in from the server, the process waits until the `\0` character is sent to read it. This character could be anything, but `\0` is nice because it's very unlikely to appear anywhere else. Once the client hears the `\0` signal, it will send the message to the `readServer()` method to process the data:

```
        if(fsClient.available() > 0) {
            String msg = fsClient.readStringUntil('\0');
            readServer(msg);
        }
```

Now draw all the circles in their correct locations:

```
        int sz = iiArr.size();
        InteractiveImage ii;
        for(int i = sz-1; i > -1; i-- ) {
            ii = (InteractiveImage)iiArr.get(i);
            fill(ii.colr);
            ellipse(ii.x, ii.y, 100, 100);
        }
    }
```

The `readServer()` method is the most important method of this application. It reads the string sent from the server and determines how long ago it sent a message to the server. If it was more than 500 milliseconds, then it's safe to assume that it wasn't sent from this application. This requires that users take turns to an extent. If there were going to be more than a few clients connected to the server, you'd want to do something different, but for this scenario it works well enough:

```
void readServer(String di) {
    int timing = millis() - millisSinceSent;
    if(timing < 500) {
        print(" too recent "+timing);
        return;
    }
    print(" not too recent "+timing);

    if(di == null) {
        println(" di null ");
        return;
    }

    XMLElement changedXML;
    fsClient.clear();
```

There's a possibility that the XML might get mangled, so the conversion to XML is wrapped in a `try/catch` block. This shouldn't happen very often, but if it does, the `try/catch` block ensures that the application doesn't throw an error:

```
    try {
        changedXML = new XMLElement(di);
    } catch (Exception e) {
        println(" can't convert ");
        return;
    }
    println(changedXML);
    int inId;
    if(changedXML.getChildCount() > 0) {
        inId = changedXML.getChild(0).getIntAttribute("id");
    } else {
        return;
    }

    for(int i = 0; i<iiArr.size(); i++) {
        InteractiveImage fi = (InteractiveImage) iiArr.get(i);
                if(fi.id == inId) {
                        int xv = changedXML.getChild(0).getIntAttribute("x");
                        int yv = changedXML.getChild(0).getIntAttribute("y");
                        fi.x = xv;
                        fi.y = yv;
                }
            }

}
```

When the mouse is dragged, loop through all the `InteractiveImage` instances and find the one that has been clicked. Once the image that has been clicked is found, then create XML with the data of that `InteractiveImage` instance:

```
void mouseDragged() {

        for(int i = 0; i<iiArr.size(); i++) {
            InteractiveImage img = (InteractiveImage) iiArr.get(i);
            if(img.isClicked(mouseX, mouseY)) {
                img.x = mouseX;
                img.y = mouseY;
                try {
                    fsClient.write(createXMLFromII(img));
                } catch (Exception e) {
                    return;
                }
```

Set the `millisSinceSent` to the current value of `millis` so that the application knows that it has recently sent data to the server and shouldn't listen to data sent from the server for the next half second:

```
                millisSinceSent = millis();
                return;
            }
    }
}
```

The `createXMLFromII()` method simply takes the properties of the `InteractiveImage` instance passed to it and creates XML from it:

```
String createXMLFromII(InteractiveImage i) {

    String s = "<img x=\""+i.x+"\" y=\""+i.y+"\" id=\""+i.id+"\" />";
    return s;
}
```

Now, you have an application that maintains state across multiple clients using a server that dispatches XML. You can use these same methods to maintain multiple applications throughout a room, a building, or on opposite ends of the world if you want.

Understanding Protocols in Networking

Think for a moment about Internet communication: a machine makes a request to a server for a particular file and the server responds with that file. That request uses a certain protocol to tell the server what file it wants, where the request is coming from, and how it would like the file returned. The response uses a similar protocol to tell the client that is requesting the file how large the file is, what type of file it is, what kind of encoding is used with the file, and other data that the client might want to know. This additional data, the order in which it is placed in the file, and how the values are signified are all defined in the Hypertext Transmission Protocol (HTTP). I use that as an example because it's probably one that you're familiar with. It's only one of the many kinds of

communication that your application can have with another computer using a network. HTTP is a protocol that uses another protocol: Transmission Control Protocol (TCP). Here's how it works: a client establishes a TCP connection to a particular port on a host. An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server returns a status line, such as `HTTP/1.1 200 OK`, and a message of its own, the body of which is perhaps the requested resource, an error message, or some other information.

Later in this chapter, we'll look at HTTP communication in greater detail. For the moment, though, let's focus on TCP. TCP provides transmission control, presents the data in order, and provides error correction when packets get out of order. In a large document where the order of all the bytes is important, this is very important. The plus side of TCP is that it is a way to connect a client to a server that ensures that all the data is in the correct order and notifies the client if something is broken in the message. The downside of TCP is that it can be comparatively slow because of this error checking. There is another connection protocol called the User Datagram Protocol (UDP). UDP is much faster than TCP, but it also has drawbacks: it has no concept of acknowledgment, retransmission, and timeout, and its messages aren't ordered.

Using ofxNetwork

There are two key classes in the ofxNetwork add-on that enable you to work with TCP communication over the Internet. The first is `ofxTCPClient`, which is for creating clients that will connect to and read information from a server, and the second is `ofxTCPServer`, which will send information to any listening client.

To make a client that can receive and send information to a server, create an `ofxTCPClient` instance, and call the `setup()` method on the client. For instance, to connect to a server running on the same machine as the client that is listening for connections on port 8180, you would call the `setup()` method like so:

```
ofxTCPClient client;
client.setup("127.0.0.1", 8180);
```

To check at any time whether the client is connected, you can call the `isConnected()` method. This method returns `true` if the client is connected to server and returns `false` if it isn't connected. This is a good way to figure out whether your client has connected to the server before you try to run any other code and to set up a retry loop to try to connect to the server again.

To send information to the server, use the `send()` method, which returns `true` if the message is successfully sent and `false` if it is not, as shown here:

```
If(client.send("hello")) {
    printf(" client sent hello ");
} else {
printf(" client didn't send hello ");
}
```

The `send()` method sends only strings, so if you want to send another data type, for instance, the bytes of an image or audio, you'll need to use the `sendRawBytes()` method:

```
bool sendRawBytes(const char * rawBytes, const int numBytes)
```

This method allows you to send byte or unsigned char arrays without modifying or needing to wrap the data. The `numBytes` parameter needs to be the length of the array that you're sending.

To receive information from a server that the client is connected to, first check to see whether anything has been sent using the `receive()` method. This gets the message sent from the server as a string. One thing to note is that this method works only with messages sent using the server's `send()` or `sendToAll()` methods or with messages terminating with the string `"[/TCP]"`. If you want to send `"Hello"` from a server that isn't an `ofxTCPServer`, you would send `"Hello[/TCP]"`.

To check the number of bytes received, use `getNumReceivedBytes()`, which returns an integer value.

When you want to load binary data, a video file, an image, an MP3, or any other kind of data that isn't a string, use the `receiveRawBytes()` method.

`int receiveRawBytes(char * receiveBytes, int numBytes)`.

You pass in a pointer to a buffer to be filled with the data, and the buffer will be loaded with the incoming information. You need to make sure the buffer is at least as big as `numBytes`.

A simple client application would have something like the following: in the `update()` method, check to see whether the client is connected. If not, connect, and then call the `setup()` method of the client. If connected, then check for any data being sent from the server:

```
void simpleClient::update()
{
    // store whether the client is connected to the server somewhere
    if(connected){
        string str = tcpClient.receive();
        if( str.length() > 0 ){
            prtinf("From ther server:: %s",str);
        }
    }else{
    // don't retry all the time, just every few hundred frames or so
        if( retryCounter > 500 ){
            connected = tcpClient.setup("127.0.0.1", 11999);
            retryCounter = 0;
        } else {
            retryCounter++;
        }
    }
}

void simpleClient::keyPressed(int key){
```

```
        //you can only type if you're connected
        if(connected){
            tcpClient.send("hello");
        }
    }
```

Setting up an `ofxTCPServer` instance is fairly similar to how the Processing Network library server is created. Declare an instance in the *.h* file of your application, and then in the `setup()` method of your application call the `setup()` method of the `ofxTCPServer`:

```
ofxTCPServer  server;
server.setup(8180, true);
```

The second parameter, `blocking` bool value is a new one and indicates whether the server should do one thing at a time, like write data to client or establish a new connection, or try to do multiple things at a time. You're usually going to have this be `false`, unless you need to write to a server that will be accepting lots of connections.

The `ofxTCPServer` has a `connected()` method that returns true if any clients are connected to the server. If there are no clients attached, then `connected()` returns `false`. It's a good idea to use this to determine whether you need to run any client communication logic and save processing time if you don't. The TCP server gives every connected client a number that is simply a count starting with 0. The `clientID` in the signatures of the methods that follow is just the index number of the connected client:

```
bool send(int clientID, string message)
```

This next line of code sends data to a client as a string, and there is also a `sendToAll()` method that omits the `clientID` and just sends the message:

```
bool sendRawBytes(int clientID, const char * rawBytes, const int numBytes)
```

Using `sendRawBytes()` lets you send and receive byte (char) arrays without modifying or appending the data and is the better option if you are trying to send something other than just ASCII strings. There is also a `sendRawBytesToAll()` method that omits the `clientID`.

To receive data from a client you can use the two following methods:

`string receive(int clientID)`
: This gets the message as a string, which will work only with messages coming via `send()` and `sendToAll()` or from messages terminating with `[/TCP]`.

`int receiveRawBytes(int clientID, char * receiveBytes, int numBytes)`
: To use this method, pass in a buffer to be filled, you'll need to make sure the buffer is at least as big as `numBytes`. As an example:

```
char buffer[6000];
receiveRawBytes(4, buffer, 6000);
```

As a demonstration of the ofxNetwork add-on, look at the following example, which is an application to capture bytes from a camera on a server and send them to a client

application that will display them there (credit goes to Theo Watson for helping out with this code).

An important concept that you'll see used here is the idea of packets. All the pixels of even a small frame of video is a very big dataset. To make sure that the server and client don't choke trying to read it all, you can write small bits of the data to the client, over and over, until it's all written. Figure 12-4 is a diagram that shows how these different classes will work together.

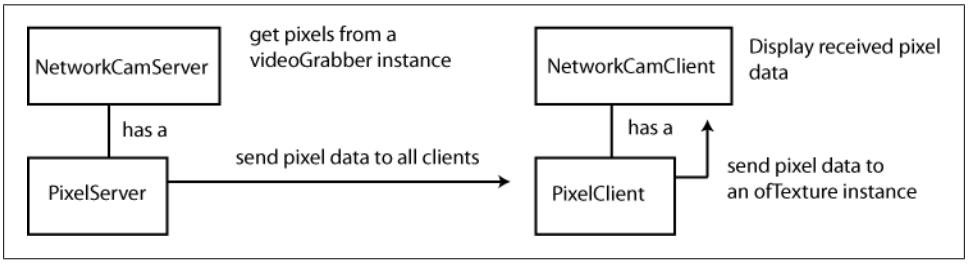


Figure 12-4. Communication between the server and clients

Example 12-3. pixelClient.h

```
#include "ofxNetwork.h"
#include "ofMain.h"

#define NUM_BYTES_IN_IMG (320 * 240 * 3)

enum clientState{
    NONE, READY, RECEIVING, COMPLETE
};

class pixelClient{
    public:
```

`pixelClient` is going to store all the bytes received and know whether it is finished receiving bytes or should expect to receive more:

```
        pixelClient(){
            state = NONE;
            totalBytes = NUM_BYTES_IN_IMG;
            bytesRecieved = 0;
            memset(pixels, 255, totalBytes);
        }

        int getState(){
            return state;
        }

        string getStateStr(){
            if(state == NONE) { return "NONE"; }
            else if(state == READY) { return "READY"; }
            else if(state == RECEIVING) { return "RECEIVING"; }
```

```
        else if(state == COMPLETE) { return "COMPLETE"; }
        else { return "ERR"; }
    }

    void reset(){
        state = READY;
        bytesRecieved = 0;
    }
```

Start the connection to the server:

```
    void setup(string ip, int port = 11999){
       TCP.setup(ip, port);
       state = READY;
       bytesRecieved = 0;
    }
```

The size of the packet here is 2,048 bytes. Each time, the client receives a packet and determines whether it still has more bytes to get:

```
    void update(int bytesToGet = 2048){

        if( state == READY || state == RECEIVING ){
            if( bytesToGet + bytesRecieved >= NUM_BYTES_IN_IMG ){
                bytesToGet -= ( ( bytesToGet + bytesRecieved ) -
                      NUM_BYTES_IN_IMG );
            }

            char tmpBuffer[bytesToGet];
            int numRecieved = TCP.receiveRawBytes(tmpBuffer, bytesToGet);

            if( numRecieved > 0 ){
                state = RECEIVING;
                memcpy(&pixels[bytesRecieved],tmpBuffer, numRecieved);
                bytesRecieved += numRecieved;
            }

            if( bytesRecieved >= NUM_BYTES_IN_IMG ){
                state = COMPLETE;
            }
        }
    }

    clientState state;
    int bytesRecieved;
    int totalBytes;
    ofxTCPClient TCP; // the TCP client that handles the actual communication
    unsigned char pixels[NUM_BYTES_IN_IMG];
};
```

The *networkCamClient.h* (Example 12-4) is the header file for the application that uses the `pixelClient` and displays its bytes to the screen when the complete image has loaded.

Example 12-4. networkCamClient.h

```
#ifndef _NETCAM_APP
#define _NETCAM_APP

#include "ofMain.h"
#include "pixelClient.h"

class NetworkCamClient : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();
        void keyPressed  (int key);

        pixelClient client;
        ofTexture tex;
        bool pixelsLoaded;

};

#endif
```

Example 12-5. networkCamClient.cpp

```
#include "NetworkCamClient.h"

void NetworkCamClient ::setup(){
    ofBackground(60,60,70);
```

Make sure to allocate the texture before trying to draw pixels into it:

```
    tex.allocate(320, 240, GL_RGB);
        pixelsLoaded = false;
    }

    void NetworkCamClient::update(){
        client.update(2048);

        if( client.getState() == COMPLETE ){
            tex.loadData(client.pixels, 320, 240, GL_RGB);
            pixelsLoaded = true;
        }
    }

    void NetworkCamClient::draw(){

        string statusStr =  "status: " + client.getStateStr();

        ofSetColor(255, 0, 255);
        ofDrawBitmapString(statusStr, 10, 20);
        ofDrawBitmapString("client - launch server than hit c key to
              connect - r to reset the state", 10, ofGetHeight()-20);
```

```
        ofSetColor(255, 255, 255);
```

If the pixels are all loaded, then they can be drawn into the `ofTexture` object so that it can be displayed:

```
        if( pixelsLoaded ){
            tex.draw(0, 25);
        }
    }

    void NetworkCamClient::keyPressed  (int key){
        if( key == 'c'){
            client.setup("127.0.0.1", 44999);
        }
        if(key == 'r'){
            client.reset();
        }
    }
```

Now, take a look at the pixel server that handles writing all the bytes in small packets over to the client (see Example 12-6). Note that the entire class is stored within the *.h* file, there is no *.cpp* file used with this class.

Example 12-6. pixelServer.h

```
#include "ofxNetwork.h"
#include "ofMain.h"

#define NUM_BYTES_IN_IMG (320 * 240 * 3)

serverState state;
int numSentBytes;
int totalBytes;
ofxTCPServer TCP;
unsigned char pixels[NUM_BYTES_IN_IMG];
```

An `enum` is used here to store all the different states that the `pixelServer` class can be in:

```
    enum serverState{
        NONE, READY, SENDING, COMPLETE
    };

    class pixelServer{
        public:


        pixelServer(){
            state        = NONE;
            totalBytes    = NUM_BYTES_IN_IMG;
        }


        void pixelServer::setup(int port = 11999){
           //setup the server to listen on 11999
           TCP.setup(port);
```

```
        state = READY;
    }

    void pixelServer::sendPixels(unsigned char * pixelData) {
        if( state == NONE ) return;
```

Here, all the pixels are copied over to the new object to ensure that if the parent application wants to overwrite them by grabbing a new frame, then it doesn't break things with the server:

```
        memcpy(pixels, pixelData, totalBytes);
        state = SENDING;
        numSentBytes    = 0;
    }

    void update(int numToSend = 1024){
        if( state == SENDING && numSentBytes < totalBytes ){
            if( numToSend + numSentBytes > totalBytes ){
                numToSend -= ( (numToSend + numSentBytes) - totalBytes );
            }
```

Here's the call to `sendRawBytesToAll()` to write the packet of bytes to all connected clients:

```
            TCP.sendRawBytesToAll( (char *)&pixels[numSentBytes], numToSend);
            numSentBytes += numToSend;
        }

        if( numSentBytes >= totalBytes ){
            state = COMPLETE;
        }
    }
};
```

The `networkCamServer` that follows is a simple oF application that allows the user to press a key and capture an image from a webcam that will then be written to the clients (see Example 12-7).

Example 12-7. networkCamServer.h

```
#ifndef _NETCAMSRV
#define _NETCAMSRV

#include "ofMain.h"
#include "ofxThread.h"
#include "ofxNetwork.h"
#include "pixelServer.h"

class networkCamServer: public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();
        pixelServer server;
```

```
        ofImage testImg;
       ofVideoGrabber grabber;
};

#endif
```

In the implementation, all that's left to do is start the `videoGrabber` and send the pixels to the `pixelServer` instance when the user hits the spacebar (see Example 12-8).

Example 12-8. networkCamServer.cpp

```
#include "networkCamServer.h"

void networkCamServer::setup(){
    ofBackground(20,20,20);
    server.setup(44999);
    grabber.initGrabber(320, 240, true);
}

void networkCamServer::update(){
    server.update(2048);
    grabber.grabFrame(); // grab a frame form the video
}

void networkCamServer::draw(){
    grabber.draw(0, 25);
}
```

When the user hits the spacebar, start sending the pixels:

```
    void networkCamServer::keyPressed  (int key){
        if( key == ' '){
            unsigned char * pix = grabber.getPixels();
            server.sendPixels(pix);
        }
    }
```

Now, you have an application that sends pixels across to another application, and with a little work you could create an application that sends images across the Internet to another application that could be located in another room or in another state.

Creating Networks with the Arduino

You can use the Arduino Ethernet Shield (Figure 12-5) with your Arduino controller to send and receive data from a network. The shield has a chip called the Wiznet W5100 that provides a network (IP) stack capable of both TCP and UDP communication and can handle up to four simultaneous connections. The Arduino team has also created the Ethernet library to make connecting to the Internet using the shield simpler.

The shield connects to an Arduino board using the long pins called *headers* that you see in the image. This keeps the pin layout intact and also allows another shield to be stacked on top if you need to attach another shield to your Arduino controller. The

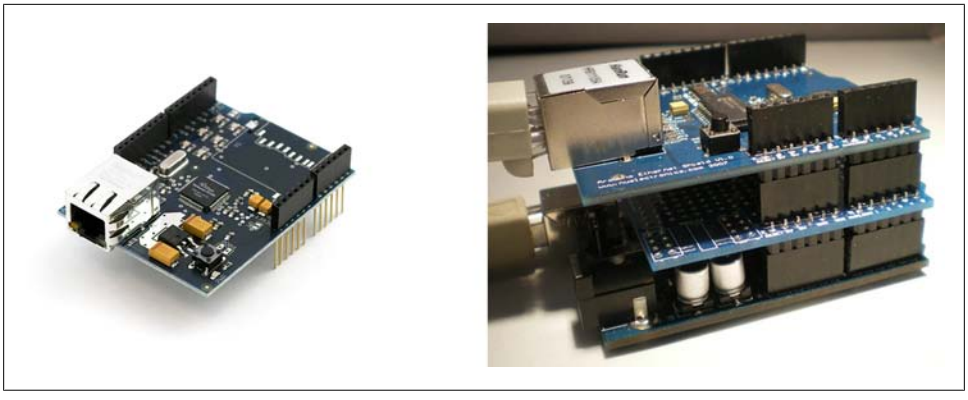


Figure 12-5. The Arduino Ethernet Shield

Arduino uses the digital pins at 10, 11, 12, and 13 to communicate with the Ethernet shield chip so you can't use these for anything else. The rest of the pins, even those that the Ethernet shield is connected to, are fine to use. The Ethernet jack on the shield is a standard Ethernet jack and will work with the same kind of cable you use for connecting a computer to the network. The reset button on the shield resets both the chip on the shield and the Arduino board.

Initializing the Ethernet Library

To use the chip on the board, you'll need to use the Arduino Ethernet library. There are a few things about the Ethernet library to note. All the communication with servers must be done using the IP address. At present, the Ethernet library doesn't work with domain names, but this might change in the future. Also, the Ethernet shield needs a Media Access Control (MAC) number. In computer networking, a MAC address is a quasi-unique identifier assigned to most network adapters or network interface cards (NICs) by the manufacturer for identification. Many times a MAC address uses the manufacturer's registered identification number. In the case of the Ethernet shield, you create a unique MAC address with 6-byte hexadecimal values.

Use the following code to initialize the Ethernet library. Notice the MAC address in line 2 and the IP (not domain name) in line 3. Also, note that if you're going to be using multiple Ethernet shields on a single network you'll need to give each of them unique addresses:

```
#include <Ethernet.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 10, 0, 0, 177 };

void setup()
{
    Ethernet.begin(mac, ip);
}
```

The `begin()` method in the code initializes the Ethernet library and network settings and has three overloaded signatures:

`Ethernet.begin(mac, ip);`
: This is the simplest signature, using only the MAC and IP addresses.

`Ethernet.begin(mac, ip, gateway);`
: This signature also uses the gateway, which is the IP address of the network gateway. In the case of a router, it is the IP address of the router. This is needed for communication on the Internet because the shield sends and receives Internet messages through the router.

`Ethernet.begin(mac, ip, gateway, subnet);`
: This is the subnet mask of the network (array of 4 bytes). By default this is 255.255.255.0. This tells the shield how to interpret IP addresses.

Creating a Client Connection

To make a client, you need to import the Ethernet library and then declare an instance of the `Client` class. Remember that C++ constructors don't use the `new` keyword:

```
#include <Ethernet.h>
bool connected = false;
```

Next, these arrays of bytes are used to start up the client and initialize the Ethernet library:

```
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192,168,0,51 }; // make sure this address is free on your
                              // network and is appropriate for your router
byte serverIp[] = {64, 233, 187, 99}; // this is googles address
```

Now, create the new client using the `serverIp` array to connect to Google and port 80 to make sure that you're connecting to the standard HTTP port. Then call `setup()` to initialize the Ethernet shield:

```
Client client(serverIp, 80);

void setup() {
    //initialize the Ethernet library
    Ethernet.begin(mac, ip);
    Serial.start(9600);

}

void loop()
{
```

If the client hasn't connected yet, try to connect again. When the `connect()` method of the client returns `true`, then send a search request to Google:

```
    if(!connected) {
       if (client.connect()) {
           connected = true;
```

```
            Serial.println("connected");

            // you could also change this line to anything else
            // "GET /search?q=programming+interactivity HTTP/1.1" for example
            client.println("GET /search?q=arduino HTTP/1.1");
            client.println("Host: www.google.com");
            // say who you are, just like in the Processing example
            client.println("User-Agent: AVR ethernet");
            client.println("Accept: text/html");
            // then insert an empty line
            client.println();
        } else {
            Serial.println(" can't connect ");
            delay(500);
        }

    } else {
```

If the client has connected already, you can begin reading the bytes from the server. The `read()` method returns the char values one at a time. This means that you have to think carefully about what data you're reading from the server and how you're going to parse it to find the bits you want. Searching through a lot of HTML returned from an HTML page can be very time-consuming and inefficient on an Arduino:

```
    if (client.available()) {
        char c = client.read();
        Serial.print(c);
    }
}
```

Creating a Server Connection

You might also want to connect an Arduino to your Internet connection to act like a server, accepting requests from other clients and sending them data in response. The Ethernet library uses a class called `Server` to create servers. There's a few things that you'll need to decide on before using the `Server` class. Will your Arduino connect directly to your modem? If you're using a router, you'll need to determine your router address. You can use several web services, or you can go to your router's admin page to check. To find the admin page of your server, you can either try 192.168.1.1 (the default address for Linksys and several other brands of routers) or look up the IP and default username and password for your brand and model. Once you've determined that, you can set the IP of the Arduino. You may also need to enable the demilitarized zone (DMZ) settings on your router to send traffic to the Arduino. By setting the DMZ, you're telling your router to stop blocking traffic that tries to visit it and send them to whatever IP is set as the DMZ. Doing a web search on any combination of these terms can help you find good and thorough tutorial. It's a bit simpler if you connect the Arduino directly to the modem.

The quickest way to see how the server works is to look at a simple example:

```
#include <Ethernet.h>

// network configuration.  gateway and subnet are optional.
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
```

You need a MAC address and an IP address to initialize the Ethernet library. Although the client uses the IP address to connect to a server, in the case of the server, the IP is the address that others will use to connect to your Ethernet Arduino. If you're going to connect your Arduino directly to your modem, you'll need to know what IP address you'll be connecting to. On Linux or Mac OS X, type `ifconfig -a` into a terminal, on Windows type `ipconfig /all`. This will spit out something that looks like this: `inet 192.168.1.100 netmask 0xffffff00 broadcast 192.168.1.255`. On Windows, the broadcast address will be the IP address with the last digits set to 255. You'll want to use the broadcast address as the IP address:

```
byte ip[] = { 24, 189, 101, 90 };
```

An array of bytes for the gateway will be needed if you're using a router to connect to the Internet. Use the gateway address found on your router's admin page and the netmask from the `ifconfig` or `ipconfig` command for the subnet value. Pass these to the call to the Ethernet `begin()` method. Also, if you're using a router to connect, you'll need the subnet address. Again, you can find this on the admin screen for your router or by the using the `ifconfig` or `ipconfig` terminal commands. In our example, we are using the simplest `begin()` method signature, so the gateway and subnet lines are shown for you, commented out:

```
// byte gateway[] = { 10, 0, 0, 1 };
//byte subnet[] = { 255, 255, 0, 0 };
```

Now, set up the server to communicate on port 80, handling HTTP requests:

```
// HTTP uses port 80 by default
Server server = Server(80);

void setup()
{
    // initialize the ethernet device
    //Ethernet.begin(mac, ip, gateway, subnet); // with subnet, ie through router
    Ethernet.begin(mac, ip); // without subnet, ie directly to modemå
    // start listening for clients
    server.begin();
}

void loop()
{
```

Next, the `available()` method of the server gets the next client that's trying to connect to the server:

```
    Client client = server.available();
    if (client) {
```

Now, print an HTTP response. HTTP uses codes to tell clients what the response is indicating. A 200 response means the request was all right and the server is responding. 404 means the client is requesting a file that doesn't exist. 500 indicates a server error, and 401 indicates a protected resource that the client doesn't have permission to access yet. There are quite a few more, but those are the most common ones. You'll always need to include a response code as shown here in the first line of the `println()` methods that the server uses to reply to clients:

```
            server.println("HTTP/1.1 200 OK");
            server.println("Content-Type: text/html");
            server.println();
            server.println("Hello World!");
        }
    }
```

And that's it. Setting up clients and servers in Arduino is fairly easy. The most difficult part of creating and connecting a client is ensuring that your addressing is correct and your requests are properly formatted. The most difficult part of using a server is correctly connecting to your server instance to your Internet connection and ensuring that requests to your modem's IP address are correctly passed on to your Arduino.

In both of these examples, we've used HTTP requests over port 80. As mentioned earlier, there are quite a few different communication protocols that you can use to connect and send and receive information. Another common protocol to use with a microcontroller like the Arduino Ethernet is the Telnet protocol, which uses port 23.

So, what can you do with the Arduino Ethernet shield and the Ethernet library? At the easiest, you can create a network between multiple Arduino Ethernet controllers using a router. Since the router can enable available servers to assign themselves IP addresses, you could create a network of multiple Arduino Ethernet servers that could broadcast signals and events to themselves all the time. An entire room or environment could be enabled to send and listen to events from Arduino controllers and physical controls. Since the Ethernet shields are not communicating with the Internet outside of the local network, all that's required is to set them all to IP addresses that the router doesn't reserve for itself (Figure 12-6).

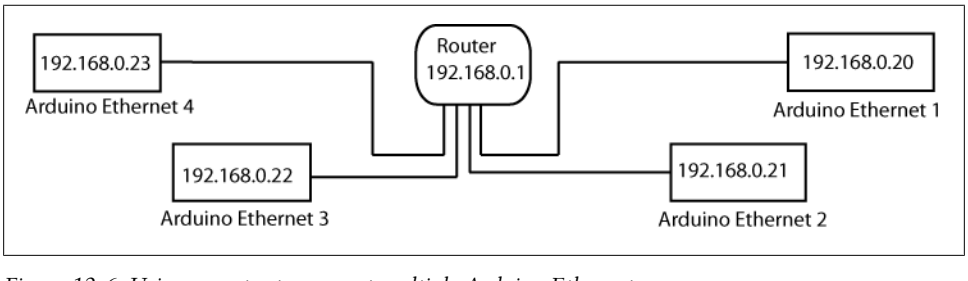


Figure 12-6. Using a router to connect multiple Arduino Ethernet servers

Another interesting use of the Arduino Ethernet is being explored at the Pachube project, initiated by Usman Haque. Pachube is a service that enables servers and clients to send and receive data from all over the world in an XML-based format. They're particularly interested in enabling and sharing real-time sensor data from objects, devices, buildings, and environments around the world. If you're looking for an interesting data feed to connect to, that's a place to begin. Many other web services return data in a format that is too data heavy for the Arduino to parse and process correctly, but with some work, you can use some of them.

There's more plenty more information on using the Arduino Ethernet both on the Arduino website and on the Pachube website. If you're interested in learning more about networking, the World Wide Web Consortium has free tutorials. *Network Know-How* by John Ross (No Starch) is also a pretty useful book for learning the basics of creating networks and enabling networked devices. The Internet itself is also filled with tutorials, hints, tips, tricks, and forums where you can ask questions about these kind of topics; in particular the Processing, Pachube, and Arduino sites are good resources.

Using Carnivore to Communicate

Let's look at another type of communication that occurs all around us in the city, particularly in any coffee shop or office building: wireless Internet. That communication, as we mentioned in the section "Understanding Protocols in Networking" on page 441, on TCP and UDP, takes place using small packets of information. Those packets can, if they aren't encrypted, be sniffed or intercepted and read as they travel from a computer to a wireless router. This can be quite fun and can be quite interesting and is also in certain circumstances illegal, so *caveat utilitor* (user beware).

A packet itself is generally split into three different parts:

Header
: The header contains information about the data carried by the packet. Usually it contains the protocol that the packet is using, destination address (where the packet is going), and originating address (where the packet came from). The header is usually 96 bits (that's not byte; that's bit, of which there are 8 in a byte) long, though this is dependent on the protocol.

Payload
: This is the actual data that the packet is carrying. This is often 896 bits, but, as with the header, it can be a different length.

Trailer
: This is a few bits that tell the receiving device that it has reached the end of the packet. It may also have some type of error checking.

Installing the Carnivore Library

To get started sniffing packets, you'll of course need a computer with a wireless card on it. To install the Carnivore library, go to *http://r-s-g.org/carnivore/processing.php*, and download the library. Drop it in your libraries folder of your main Processing folder.

If you're on Mac OS X or Linux, you'll need to run the following command every time you restart your computer before you start the Carnivore library to ensure that the Carnivore library can access and read the packets that the wireless card receives:

```
sudo chmod 777 /dev/bpf*
```

On Windows you'll need to install the WinPcap library, which you can download from *www.winpcap.org*.

Once you have the library installed, restart Processing, and then you're ready to begin. The `Carnivore` class is constructed like so:

```
CarnivoreP5 c = new CarnivoreP5(this); // this is your application
```

In your application, you can add the `packetEvent()` event handler that the Carnivore library will call every time a new packet has been detected and sniffed:

```
void packetEvent(CarnivorePacket packet)
```

One of the most important classes in the Carnivore library is the `CarnivorePacket`. Once you've received a packet, you'll want to get some of the information about that packet:

```
void packetEvent(CarnivorePacket packet){
  println(" the packet is in this protocol : " + packet.strTransportProtocol);
  println( " the sender IP and socket is : " + packet.senderSocket());
  println( " the receivers IP and socket is " + packet.receiverSocket());
}
```

This will print something like this:

```
[CarnivoreP5] newCarnivorePacket from 192.168.1.100
the packet is in this protocol : TCP
the sender IP and socket is : 192.168.1.100:61362
the receivers IP and socket is 74.125.45.83:80
```

The packet can also be read using the `data` property of the `CarnivorePacket`, which returns an array of byte objects. There are a few interesting methods and properties of the `CarnivorePacket` that might be helpful to you:

`String dateStamp()`
: This method gets the date of the packet in the format hour:minute:second:millisecond.

`int intTransportProtocol`
: This gets the packet type as an int, either 6 (TCP) or 17 (UDP).

`IPAddress receiverAddress`
: This gets the packet's receiver IP address. Use `receiverAddress.ip` to get the IP address as a string.

`IPAddress senderAddress`
: The packet's sender IP address. Use `senderAddress.ip` to get the IP address as a string.

`String toString()`
: This method returns the entire packet as a String.

Creating a Carnivore Client

In the next example, you'll see how to build a full Carnivore client that detects any network traffic around your computer and draws ellipses to represent the senders' and receivers' IP addresses and connection lines to show the connections between the packages. First, create a data object to represent the data about the client:

```
class Client {

    String location;
    int id;
    int xPos;
    int yPos;

    String port;
    String ip;

}
```

Here is the actual application. Note the `import` statements at the top of the application that import the Carnivore library:

```
import org.rsg.carnivore.*;
import org.rsg.lib.Log;

int clientCount = 0;
Client clients[];

void setup(){
    size(600, 600);
    clients = new Client[10];
    background(0);
    //Log.setDebug(true); // Uncomment for verbose mode
```

Next is the constructor call to the Carnivore library, notice that the Processing version of the library is actually called CarnivoreP5, so the constructor method is `CarnivoreP5()`. You can also set how much traffic you want to detect using the `setVolumeLimit()` method, in case of very busy areas:

```
    CarnivoreP5 c = new CarnivoreP5(this);
    //c.setVolumeLimit(4); //if there is too much traffic set this to 1
    //too little 10
}

void draw(){
```

```
        fill(0, 2);
        rect(0, 0, 600, 600);
```

Simply draw an ellipse for each client that has connected:

```
        for(int i = 0; i < 10; i++) {
            if(clients[i] != null) {
                fill(122);
                ellipse(clients[i].xPos, clients[i].yPos, 50, 50);
            }
        }
    }
```

Next, the `packetEvent()` method, which receives the packets and determines whether the sender and receiver of the packet have already been captured, then draws the connection between the two:

```
    void packetEvent(CarnivorePacket packet){

        int x1 = 0;
        int y1 = 0;
        int x2 = 0;
        int y2 = 0;

        String firstip = packet.receiverAddress.ip.toString();
        String secondip= packet.senderAddress.ip.toString();
        boolean foundFirst = false;
        boolean foundSecond = false;

        for(int i = 0; i < 10; i++) {
            if(clients[i] != null ) {
                if(firstip.equals(clients[i].ip)) {
                    x1 = clients[i].xPos;
                    y1 = clients[i].yPos;
                    foundFirst = true;
                }
                if(secondip.equals(clients[i].ip)) {
                   x2 = clients[i].xPos;
                   y2 = clients[i].yPos;
                   foundSecond = true;
                }
            }
        }
        if(foundFirst && foundSecond) {
            println(" draw "+x1+" "+y1+" "+x2+" "+y2);
            line(x1, y1, x2, y2);
        }
        if(!foundFirst && clientCount < 10){
            clients[clientCount] = new Client();
            clients[clientCount].ip = firstip;

            String ip_as_array[] = split(firstip, '.');
            // Scale to applet size
            clients[clientCount].xPos = int(ip_as_array[2]) * width / 255;
            // Scale to applet size
            clients[clientCount].yPos =  int(ip_as_array[3]) * height / 255;
```

```
            clientCount++;
        }
        if(!foundSecond && clientCount < 10){
            clients[clientCount] = new Client();
            clients[clientCount].ip = secondip;

            String ip_as_array[] = split(secondip, '.');
            // Scale to applet size
            clients[clientCount].xPos = int(ip_as_array[1]) * width / 255;
            // Scale to applet size
            clients[clientCount].yPos =  int(ip_as_array[3]) * height / 255;

            clientCount++;
        }
    }
```

There's quite a bit more you can do with the Carnivore library, but this simple example should get you started working with it.

What other things can you do with the Carnivore library? The easiest answer is to inspect the traffic in a public place and inspect the destination of traffic. That data can be used to create a world map using one of the many free IP-to-location services, a sound, a map of activity, or any type of data visualization that you can imagine. By inspecting the packet data of a wireless network computer further, you can determine what activity is being used and from there create a more specialized data set: search engine terms, languages used, or any other special set of data that you can imagine. It all comes down to a matter of parsing the data, stripping out what's useful to you, and leaving behind what isn't.

Next, you'll learn about another way of transmitting data wirelessly: Bluetooth.

Communicating with Bluetooth

You may have heard of Bluetooth already, because many cell phones, laptops, and other devices use it to enable communication between peripheral devices like headsets and cell phones or phones and computers. Bluetooth is basically a networking standard like the 802.11 wireless Internet protocol that works at two levels: as a radio frequency standard and as a data standard. Bluetooth is a protocol and that means it provides a standard for when bits are sent, how many will be sent at a time, and how the parties in a conversation can be sure that the message received is the same as the message sent. The big benefits of Bluetooth are that it is wireless, inexpensive, and automatic. Bluetooth sends out very weak signals of about 1 milliwatt (most cell phones can send a signal of about 3 watts), so the range of a Bluetooth device is about 10 meters. Despite the relative weakness of the signal, Bluetooth is wireless and can transmit through walls, making the standard useful for controlling several devices in different rooms without needing to run any wires. You can connect up to eight devices simultaneously in a single area.

There are two approaches to working with Bluetooth that will be covered in this chapter: the Bluetooth Arduino board and the Processing bluetoothDesktop library.

Using Bluetooth in Processing

bluetoothDesktop allows Processing sketches to send and receive data via Bluetooth wireless networks on any computer that can support the Java Bluetooth library called JSR-82, which is what Processing uses to facilitate Bluetooth communication. It is an adapted version of a Bluetooth library written by Francis Li for the Processing Mobile project that enables you to run Processing on certain cell phones. Using this library, a Processing sketch running on a computer with a JSR-82 implementation can connect to other Bluetooth devices as well as act as a service to which other devices can connect.

Installing Bluetooth on Linux and Windows

Download the library, unzip it, and place the *bluetoothDesktop* folder into the Processing libraries folder.

Installing Bluetooth on Mac OS X

Download the library and unzip it. The bluetoothDesktop library includes two different libraries that you can use to connect to BlueTooth devices: bluecove and avetana. bluecove is open source, free, and the generally preferred option. However, it might not work properly on your machine. If it doesn't, you can use the commercial avetana library instead. To start, you'll need to remove one or other of the libraries to ensure that the Processing application doesn't initialize both. I'd recommend trying the bluecove library first, so move the *avetana.jar* file to a separate folder outside your library folder. If you run into problems with the bluecove library and you really want to use Bluetooth, it might be worth looking into purchasing a license.

Using the bluetoothDesktop Library

Once you've installed bluetoothDesktop, you're ready to look at the four main classes that the bluetoothDesktop library uses: `Bluetooth`, `Client`, `Device`, and `Service`. Bluetooth works a little differently than the server-client communication that we've been looking at so far. Any Bluetooth-enabled device can create a service that other clients can connect to, and any Bluetooth-enabled device can connect to any other devices that have made themselves available. The services that these devices create and connect to are simply referenced by strings in the bluetoothDesktop library, so you could create a service called `myGreatService`. Other devices that find this service can connect to it, or the service itself can actively look for clients with which to connect and attempt to connect. That connection allows a service-providing device to send and receive bytes, strings, or integer values from a connected client device. Once the devices have finished communicating, the client simply closes the connection.

With that high-level overview in mind, the four primary classes in the bluetoothDesktop library will make a little more sense. The primary interface for both finding other Bluetooth services that exist and creating new services is called *Bluetooth*, and it, like many other Processing libraries, has five important methods that allow you to start an action and use callback methods to tell your application when those actions have completed. For instance, to find other Bluetooth-enabled devices, create a new Bluetooth instance, and use the `discover()` method. The Bluetooth instance will call the `deviceDiscoverEvent()` method whenever it finds a new device and the `deviceDiscoveryCompleteEvent()` method when all devices that can be found have been located:

```
import bluetoothDesktop.*

Bluetooth bluetoothInstance;

void setup() {
    bluetoothInstance= new Bluetooth(this);
    bluetoothInstance.discover();
    noLoop(); // tell the Processing application that we don't need a loop
}

void deviceDiscoverEvent(Device dev) {
    println(dev.name + " discovered.");
}

void deviceDiscoveryCompleteEvent(Device[] devices) {
    println(" Found " + devices.length+ " devices.");
    for( int i = 0; i<devices.length; i++) {
    println("i: "+devices[i].name;
    }
}
```

If you want to create a new service using your Bluetooth instance, use the `start()` method:

```
Bluetooth bluetoothInstance;
String amazingService = "AMAZING";

void setup() {
    bluetoothInstance= new Bluetooth(this);
    bluetoothInstance.start(amazingService);
}
```

To stop the service, simply pass the name of the service to the `stop()` method:

```
stop(amazingService);
```

The Bluetooth object can also call the following callback methods on a Processing application:

`serviceDiscoverEvent()`
: This method is triggered when a new service is found. If you're creating a client object, you'll use this method and the next quite frequently because it allows you to search out new services and discover information about them.

`serviceDiscoveryCompleteEvent()`
: This method is triggered when the Bluetooth object is finished searching for services and will pass an array of service objects to the method for you to use.

`clientConnectEvent()`
: This method is triggered whenever a new client connects to the service.

The `Device` class, which was used in the `deviceDiscoverEvent()` method in the previous example, is used to represent any devices discovered on the Bluetooth network. The `Device` class has properties that give you its name and address, a `discover()` method that searches for available services, and a `cancel()` method that cancels the search for a service.

The service represents software running on devices that can be connected to via the Bluetooth network. It, like the `Device`, has a `name` property but adds a `device` property that returns the device that hosts the service, as well as a `connect()` method that returns a connected `Client` object.

The `Client` object is where the bulk of the actual reading and writing of data between two devices takes place. There are a series of read methods for a client to read information from a server:

`read()`
: Reads one byte of data.

`readBytes(buffer, offset, length)`
: Reads information into an array of bytes that will hold the data. `offset` is the index into `buffer` to start storing data, and `length` is the maximum number of bytes to store.

There are also methods for reading data with specific types:

`readChar()`
: Reads char values

`readInt()`
: Reads int values

`readUTF()`
: Reads String values

The write methods work similarly:

`write(int value)` *or* `write(byte[] buffer)`
: Writes information to a server that the client is connected to

`writeBoolean()`
: Writes Boolean values to a server that the client is connected to

`writeChar()`
: Writes char values to a server that the client is connected to

`writeInt()`
: Writes int values to a server that the client is connected to

`writeUTF()`
: Writes String values to a server that the client is connected to

Finally, the `stop()` method closes the connection and ends any communication that the client is conducting.

Using the Arduino Bluetooth

So far, you've used only the USB cables to connect Processing applications and Arduino controllers. By using the Bluetooth library, you can have your Arduino communicate with your Processing application wirelessly. There are two approaches to having your Arduino communicate wirelessly: the first is to use a Bluetooth broadcaster like the BlueSmirf Gold, and the other is to use the Arduino BT microcontroller. Figure 12-7 shows both.

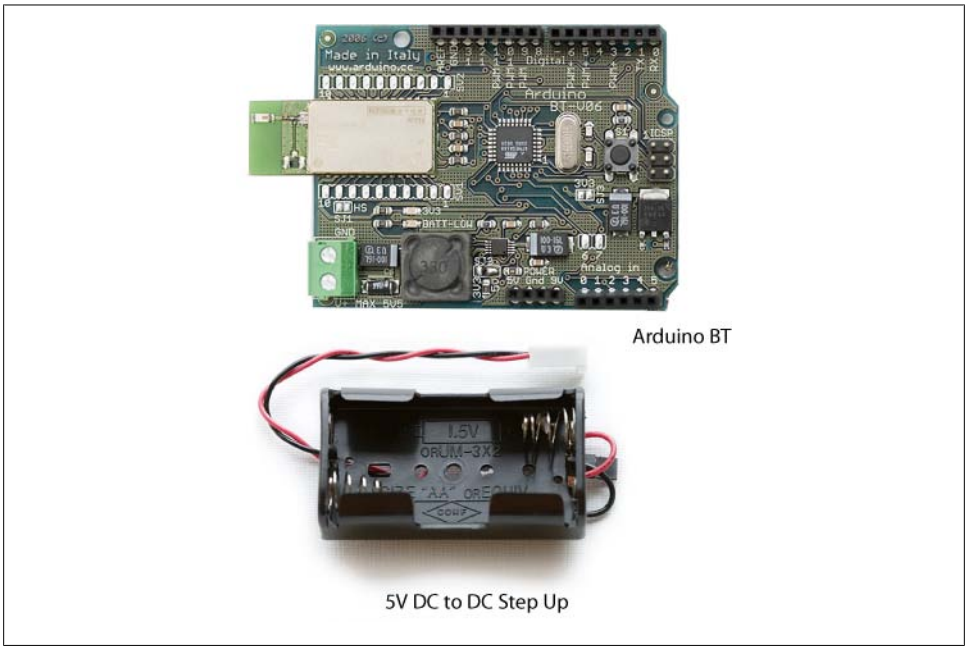
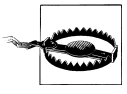


Figure 12-7. The Arduino BT module and a battery pack to power it

Unlike the other Arduino controllers, it doesn't have a USB port. Arduino BT comes with a bootloaded program that allows you to upload sketches to it via Bluetooth, so you need to make sure that your computer has a Bluetooth modem enabled and working on it.



The Arduino BT can't be powered by more than 5.5V, which means that plugging a 9V battery or other power source to it *will fry it*. The simplest way to get an Arduino BT powered up at first is to connect it to the 5V power port of an Arduino Decimilia or Duemilnove. You can also use a 6V or 9V battery with a Rectifier Diode (they're usually part number 1N4001~7), which will reduce the voltage by about 0.7V per diode, so using them in sequence will cut the voltage down to an amount that won't destroy your Arduino BT. Another option is to use a device like 5V DC to DC Step Up from Bodhilabs, which provides 5V of power to the Arduino BT.

Once you have the Arduino powered up, you'll need to be able to connect to it. This can be a bit tricky and the instructions may change so take a look at the Arduino website for the most up-to-date documentation on how to get the Arduino BT running with your particular operating system.

Once you have configured your Arduino BT and power it up, your Arduino is totally wireless and can transmit messages to a computer without needing to be physically connected to the machine. All the pins on the Arduino work the same as on the Decimilia with one exception. You'll notice that on the Arduino BT that digital pin 7 is marked with a dash (–) instead of a 7. This is because this pin is reserved for the Bluetooth chip on the Arduino BT and causes the chip to reset.

So now, we'll look at connecting the Arduino BT to a Processing application. There are two approaches. The first is to use the Firmata library. To do this, you'll need to first upload the Firmata library to your Arduino BT. You may find yourself getting messages that look like this:

```
avrdude: stk500_recv(): programmer is not responding
```

If you see these remember to press the Reset button on the Arduino BT (it looks the same as the Reset button on the Decimilia) and then try uploading your sketch. Now you have the Firmata library loaded onto your Arduino, which means that you can control your Arduino from a Processing or oF application. The next step is to run the following code as a Processing application. This lets you open a connection to Arduino BT:

```
import processing.serial.*;
import cc.arduino.*;

String port = "/dev/tty.ARDUINOBT-BluetoothSeri-1";
Arduino arduino;
```

```
void setup()
{
```

The `list()` method gets a list of all the Arduino-enabled ports. From this, find the one that matches the connection settings that you've used to upload code to the Arduino BT in your Arduino IDE. In this example, it's `/dev/tty.ARDUINOBT-BluetoothSeri-1`, but it might be different for you:

```
    String arduinoList[] = Arduino.list();
    size(512, 200);
    // find the port that you've configured your Arduino BT to use
    int correctPort = 0;
    for(int i = 0; i < arduinoList.length; i++)
    {
        if(arduinoList[i] == port) {
            correctPort = i;
        }
    }
    // once you find it, create a new Arduino instance
    arduino = new Arduino(this, arduinoList[correctPort ], 115200);
}

void draw() {
    if(arduino != null) {

        arduino.pinMode(12, Arduino.INPUT);
        int analogValue = 0;
```

Read the value of analog pin 0 using the `analogRead()` method. You could connect any control that returns analog values—a potentiometer, distance sensor, and one pin of an accelerometer, among others:

```
        analogValue = arduino.analogRead(0);
        // mapping the analogSensor values (0-1023) to values between 0-255
        int inByte0 = int(map(analogValue,0,1023,0,255));
        println(" anlg "+analogValue);

       int digitalValue = 0;
```

Read the value of pin 12. If it's equivalent to `Arduino.HIGH`, then print the value. You can connect any control that returns digital values:

```
      // mapping the analogSensor values (0-1023) to values between 0-255
         if(digitalValue == Arduino.HIGH) {
         println(" dig "+digitalValue);
        }
    }
}
```

Now you can receive data in a Processing application sent wirelessly from an Arduino controller. You can attach a battery to your Arduino BT, making sure not to run more than 5.5V through the controller, and create an entirely wireless device that will broadcast data to your Processing application.

You can also look at using a different Bluetooth module with your Arduino, some of the more common are shown in Figure 12-8.

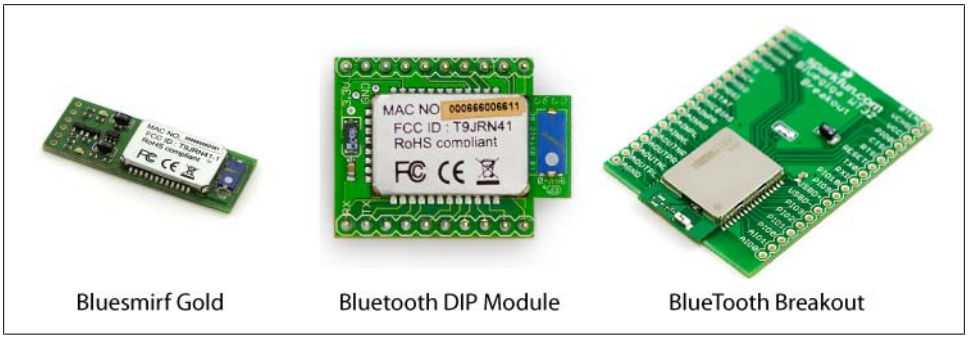


Figure 12-8. Other Bluetooth modules

These have the advantage of being less expensive and more multipurpose, though they do require some manual configuration. Both the Arduino website and Tom Igoe's *Making Things Talk* (O'Reilly) contain excellent resources for getting up and running with the Bluesmirf module pictured in Figure 12-8. Any of these modules can communicate with the Arduino by connecting them to the TX and RX pins of the Arduino and using the hardware serial and the `Serial` library.

Communicating Using MIDI

In interaction design, you always want to look for actions or paradigms of interaction that users are already familiar with. Some of the richest examples of interfaces come from the world of music. Taking the piano keyboard as an example, you have two approaches for getting data from the interface into your application: by analyzing the notes that are played on the piano or by using an electronic keyboard and reading the notes that the user plays. Techniques, the first approach, were covered in Chapter 7. To use the second approach with an electronic device, you'll want to use the Musical Instrument Digital Interface (MIDI) protocol. This is an easy way to get an Arduino or your laptop communicating with electronic musical instruments of all sorts, not just pianos.

If you decide to work with MIDI, two other elements are added to the system design. The first is the MIDI device itself, and the second is the data format that all MIDI devices use. The device can be any MIDI keyboard, mixing board, or even computer program. Which device and how to present it to the user is left up to you. The way that the MIDI communicates is quite straightforward. The MIDI specification consists of many different signals. Like so many of the other protocols that have been discussed in this chapter, MIDI has a way of defining what devices should be listening to which signals. In MIDI, this is done by using a channel, which is sent in the last 4 bits of the message

byte. In programming for audio, a device or program will respond to messages sent with the channel ID that it is tuned to and will ignore all other channel messages. The most relevant four are shown in Table 12-1.

Table 12-1. MIDI messages

Message	Meaning	First byte	Second byte
0x80 or 128	Note Off	Key	Velocity
0x90 or 144	Note On	Key	Velocity
0xA0 or 160	Key Pressure	Key	Pressure
0xE0 or 224	Pitch Bend	Least Value of Bend	Maximum Value of Bend

A MIDI message from a keyboard will look like Figure 12-9, with each block representing a byte of data.

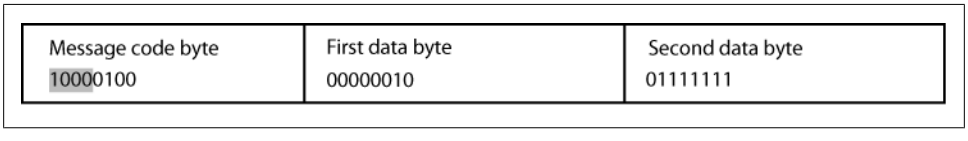


Figure 12-9. A MIDI message block

If a MIDI device received the following message:

```
10000100  00000010  01111111
```

it would break it down like this: the first four bits of the status byte, 1,001 or 0x80, tell MIDI that the message is a "note on" command, and the last four bits of that first byte tell MIDI what channel the message is for (0000=MIDI channel #1, 1111=MIDI channel #16). The first data byte tells MIDI what note to play, 00000010 is decimal 12, the lowest C on a piano; while the second data byte tells MIDI how loud to play the note, in this case the maximum velocity of 127. The device should play the note until a "note off" message on the same channel and with the same note is received. If you're thinking of communicating with an Arduino using MIDI devices, you'll probably be most interested in the message indicating what the user has done and the note itself since it would indicate what key the user has pressed on a MIDI device. Not all MIDI devices send velocity, but if they do, then you can read that information too to create quite interesting and responsive interfaces.

Sending data from an Arduino to a MIDI device is considerably easier than receiving data from a MIDI device. A good number of tutorials are available online to show you how to produce MIDI data from an Arduino. In this next code example, you'll see how to send data to an Arduino from a MIDI device like a keyboard. First, look at the schematic (Figure 12-10).

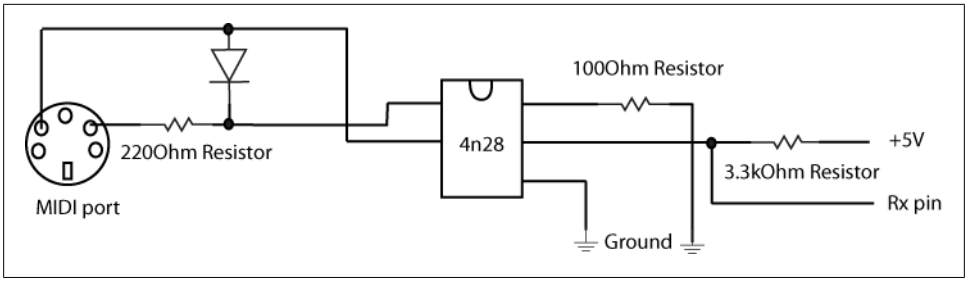


Figure 12-10. Wiring a MIDI in connection

Notice in the schematic the box in the middle marked 4n28. This is an opto-isolator that is an electrical component made up of a light-emitting device and a light-sensitive device with no electrical connection between the two. An opto-isolator uses a beam of light to tell a device on one side of the circuit that a certain amount of current has been reached on the other side. The reason for this is that the MIDI connection uses a lot of voltage, enough to fry your Arduino at worst and interfere with it at best. The opto-isolator allows the MIDI signals to be sent to the Arduino without the dangerous amounts of voltage that they use. The other relevant part of the diagram is the MIDI 5 pin DIN connector. This is what is used to connect to the MIDI device. Figure 12-11 shows images of the two components.

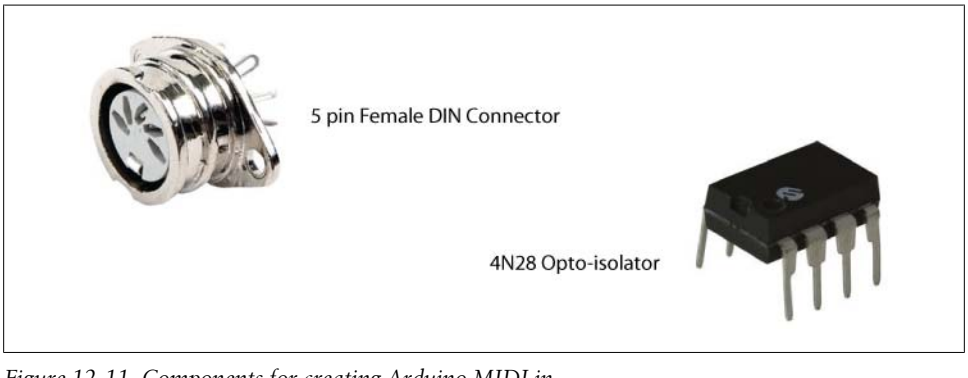


Figure 12-11. Components for creating Arduino MIDI in

Adafruit also has created a MIDIsense Analog+Digital I/O kit that will save you needing to replicate the wiring described above. It has two MIDI ports that can be powered by a battery, and can easily to configured to communicate with an Arduino.

Now take a look at the code to read a MIDI in signal (special thanks to Stephan C. from the Arduino message boards for providing this code):

```
byte incomingByte;
byte note;
byte velocity;
```

```
int statusLed = 13;   // select the pin for the LED

int action=2; //0 =note off ; 1=note on ; 2= nada
#include <SoftwareSerial.h>


//setup: declaring iputs and outputs and begin serial
void setup() {
```

Start the Arduino serial with a baud rate of 31250 or 38400 to read the MIDI signal. Since the MIDI cable is connected to the RX pin of the Arduino board, the data can be read via the serial port:

```
    Serial.begin(31250);
    digitalWrite(statusLed,HIGH);
}

//loop: wait for serial data, and interpret the message
void loop () {
    if (Serial.available() > 0) {
```

The serial port reads the incoming byte:

```
        incomingByte = Serial.read();
```

Here, the application waits for a status byte at the beginning of the message to determine what action the MIDI message is indicating. If the message starts with 144, then it is a "note on" message, so the applications sets the `action` variable and expects that the next bits sent will be the note that has been played:

```
if (incomingByte== 144){ // note on message starting starting
  action=1;
}else if (incomingByte== 128){ // note off message starting
  action=0;
```

If the serial port has already received a "note off" message, then the incoming byte must be indicating which note has just been released:

```
}else if ( (action==0)&&(note==0) ){
  note=incomingByte;
  playNote(note, 0);
  note=0;
  velocity=0;
  action=2;
```

If the serial port receives a "note on" message, then the incoming byte must be indicating which note has just been played:

```
}else if ( (action==1)&&(note==0) ){
  note=incomingByte;
```

If both the action and the note have already been set, then the incoming byte must be the velocity, so we can use that in playing the note:

```
}else if ( (action==1)&&(note!=0) ){ // ...and then the velocity
  velocity=incomingByte;
  playNote(note, velocity);
```

```
        note=0;
        velocity=0;
        action=0;
      }
    }
  }
```

This method is left blank for you to play around with how to handle the data. You can use the data to control a servo or another machine; you can use the values to send to an oF or Processing application; or you can create a lightshow driven by a keyboard:

```
void playNote(byte note, byte velocity){
    // play with the notes and velocities
}
```

If you're interested in sending this data to an oF or Processing application, you might notice a potential problem with this code. The serial port can do only one thing at a time. This isn't quite true, but it's true enough for the moment. If you're interested in learning more about creating software serials, look up the AFSoftwareSerial library, the NewSoftSerial library, or flip ahead to Chapter 15, *Movement and Location*, where both of these libraries are discussed. It isn't going to work for us now, but it is a solution worth looking into. At the moment, for this example to communicate with Processing or oF, we need to have two hardware serial connections, so the easiest thing to do is connect the digital pins to another Arduino and then write to a listening oF or Processing application from the second Arduino. There are a few different ways to do this. The easiest is to wire the digital pin set to OUTPUT of one Arduino to a digital pin set to INPUT on the second Arduino. Another approach is to use Inter Integrated Circuit (I2C), as discussed in Chapter 7. You might not be interested in sending this information on to another machine. You might want to drive a Piezo sensor used as a speaker, control servo motors or even home appliances with this information, drive LED displays, or try any number of interesting ways of creating feedback.

Review

XML is a hierarchical data format consisting of nested nodes that represent data and the relationship of one piece of data to another. It's often used to save or read data, particularly on the Internet.

All XML documents consist of nodes shaped like `<node>Value</node>`. Nodes can have children, defined like `<parent>Value<child>ChildVal</child></parent>`.

To send or receive information from a network in Processing, use the Processing Network library. This defines `Client` objects that can read information from local and remote servers and `Server` objects that can respond to requests from local or remote clients.

Transmission Control Protocol (TCP) is a heavier and slower network protocol more often used for documents or text data. Universal Data Protocol (UDP) is a more

lightweight network protocol most often used for streaming video, audio, or other binary data where the bit order is not as important as the speed of communication.

To send or receive information from a network in oF, use the ofxNetwork add-on for openFrameworks. This allows you to send or receive data via TCP or UDP. The ofxNetwork add-on defines an `ofxTCPClient` object for reading Internet data or other TCP format communications, an `ofxTCPServer` for serving up data in the TCP, and an `ofxUDPServer` and `ofxUDPClient` for using UDP.

To use Ethernet connections with an Arduino, you can use the Arduino Ethernet Shield, which allows you to send and receive both TCP and UDP message directly from an Arduino controller. The Ethernet library for the Arduino provides the functionality that you need to read or write network data.

If you're interested in gathering wireless data, you can use the Carnivore library for Processing. This allows you to packet sniff data passing through a wireless network and determine the origin and destination of the data. Bluetooth is a data and transmission protocol for low power wireless communication over short distances. It is often used in cellphones and electronic peripherals like ear-pieces and keyboards.

The Arduino BT board allows you to communicate with another Arduino BT board or a Processing application over Bluetooth.

MIDI is another useful data protocol that allows you to read data from a MIDI device like a keyboard and use it as input in an Arduino board.

PART III
Explorations

In this part, you'll look at some of the directions that you might be interested in exploring further. This is only a small sampling of the range of how designers and artists are using the tools you've learned about in this book, but it should give you some good ideas as you begin developing your own interactions.

Chapter 13, *Graphics and OpenGL*, covers 3D graphics both in Processing and openFrameworks and introduces you to the graphics programming language OpenGL as well as to some additional libraries in oF and Processing for working with 3D graphics. In Chapter 14, *Detection and Gestures*, you'll learn about OpenCV and computer vision for openFrameworks and for Processing. Chapter 15, *Movement and Location*, explores ways of using location and movement for input into your application using GPS, networks, and other tools. Chapter 16, *Interfaces and Controls*, discusses controls and tools in greater depth and shows you how to integrate other tools and devices into your application to create richer interfaces. In Chapter 17, *Spaces and Environments*, you'll learn about how to work with spaces and environments, about some techniques for monitoring environmental data, and about some potential applications for the Arduino in reactive architecture and smart environments.

The last chapter in the book, Chapter 18, *Further Resources*, points you to some of the other areas of research and exploration that you might find interesting and lists the most relevant texts that have been referenced throughout this book.

CHAPTER 13

Graphics and OpenGL

If you're interested in displaying visual feedback or gathering input from a screen, learning how to create three-dimensional graphics is an important skill to learn. We see in three dimensions, we experience in three dimensions, and increasingly we expect our graphics to be in three dimensions. It's not just a matter of attempting to mimic our experience of the world or meet the expectations of what something should look like in a realistic depiction of a scene; it's also a matter of providing more data in a manner that humans are accustomed to receiving information.

The ability to make accurate and precise 3D graphics is core to being able to represent objects in the world effectively. There's a marked difference between making accurate graphics and realistic graphics. Most architectural drawings, diagrams, and visualizations aren't particularly realistic, but they are very data rich. They behave the way we expect them to behave when we change the view, rotate the object, or change the distance at which we view the object. When you're creating an interface, this is far more important than accurately replicating a real-world object. Many times, but not all times, creating a usable interface that provides legibility and usability trumps wowing a user the first time they see something.

A lot of, but certainly not all, advanced graphics code in Processing is written in OpenGL. You can do a great deal with the core graphics classes that are provided in Processing, but many times you'll find that you need to create an effect or shape that simply runs too slowly or that you need more control over your graphics than is provided in the default Processing libraries. By contrast, a lot of the visual code in oF is written directly using OpenGL commands and not library functions that wrap oF calls, though that of course depends on what the developer needs and the project demands.

What Does 3D Have to Do with Interaction?

Since the design of an interaction is a matter of creating meaningful ways to input data and meaningful ways of receiving feedback, the use of 3D graphics allows you to do both effectively. We're used to perceiving in three dimensions, so when we encounter visual information presented in three dimensions, it mirrors how we perceive the world.

Depth, scale, and distance can all become elements in the feedback loop that your users perceive and hence change the way that they respond. Video games are a wonderful example of this. The two-dimensional scrolling game provided a vastly different range of information to a user than the three-dimensional first-person game. A user couldn't perceive the depth, their own position, or the position of other elements in the game properly without a way of accurately representing three-dimensional space. Once three-dimensional space could accurately be mapped, the user could provide a whole range of different input to the game, new controllers were required, and games changed dramatically.

Games aren't the only paradigms in which the ability to map 3D and graphics is a massive aid. Architectural walk-through applications are invaluable to architects, designers, and urban planners. Mapping applications provide vastly more useful data to users when the data presented can simulate the 3D world in which we live. Accurate three-dimensional mapping allows teams in all sort of different situations, from surgical to military applications, to collaborate in real time. These sorts of spaces allow us to understand, move, and manipulate spaces the way we're used to doing in real life.

We're used to doing things in three-dimensional space: turning objects around, looking at objects side by side, rotating objects, and placing things in 3D. When you allow for 3D feedback, you create the possibility of 3D input and actions, turning things over, rotating them in space, stacking them, comparing them in 3D. This chapter is going to have a lot more technical information and a lot less exposition than other chapters because the technical challenges of getting you started with OpenGL and 3D are considerable, and many applications require the kinds of complex graphics that are best done in OpenGL.

Understanding 3D

There are a few important but simple concepts to understand about three-dimensional objects and three-dimensional space:

Points
: Any object in 3D space will have `x`, `y`, and `z` coordinates; that is, the object will be located at a 3D point. In OpenGL, points that connect multiple lines or objects together are called *vertices*. You can simply think of a vertex as a point that can be used to represent a location or a velocity.

Vectors
: An object in 3D space can also have a heading, which is a direction that it is moving in or looking toward. This is represented by a three-dimensional vector, which is a vector with x, y, and z directions and a magnitude that describes how quickly it is moving in that direction. In the case of a simple heading or looking-at described in a three-dimensional vector, then the magnitude is 0. Fundamentally, a point and

a vector are the same thing: an array of values that represent data about a location in space.

Lines and objects
: Any line in 3D space will connect two 3D points. Any object will have two more points or vertices.

Surfaces
: Surfaces in 3D are made up of points that have a shader or texture assigned to them to fill the space between the points on the *front* of the surface. This is a little strange because although we generally expect a surface in real life to be more or less the same on both sides, in computer 3D graphics surfaces often only one side. It's as if when you looked at a table from the top, it was a table but when you looked at it from the bottom, it vanished. You can set the drawing mode to have two sides when you draw, but it's an additional step.

Matrices
: Matrices are one of the most important things to understand when working with 3D and in particular when working with OpenGL. Matrices were introduced in Chapter 9, but it's worth reviewing them again. A matrix is a mathematical structure consisting of multiple columns and rows:

```
[m11] [m21] [m31] [m41]
[m12] [m22] [m32] [m42]
[m13] [m23] [m33] [m43]
[m14] [m24] [m34] [m44]
```

  Most matrices that you'll encounter in this chapter are 4 × 4 matrices, and most of the operations that you'll see for manipulating how objects are drawn or how they appear involve multiplying matrices. You can change either the matrix that represents the world in which the objects are drawn or the matrix that represents the view onto that world.

Transformations
: You change matrices by using a *transformation*, which is simply the name for the mathematical operation for modifying a matrix. The `translate()`, `rotate()`, or `scale()` methods are all examples from Processing of methods that change a matrix. You use transformations to change the matrices that represent the camera view into the 3D world, the locations that objects are drawn in the 3D world, or the way that the world is projected into the camera. There's plenty more on this later in this chapter.

Working with 3D in Processing

Rendering graphics requires trade-offs between speed, accuracy, and general usefulness of the available features. None of the renderers is perfect, so we'll provide multiple options in the following sections so that you can decide what trade-offs make the most sense for your project. It would be nice if all of them had perfect visual accuracy, to

have high performance, and to support a wide range of features, but that's simply not possible.

The examples in this chapter add the third parameter, `P3D`, to `size()` to draw in 3D in Processing. All these examples can also run faster by importing the OpenGL library from the Import Library menu and using the constant `OPENGL` as the third parameter to `size()`.

`P3D` *(Processing 3D)*
: This is a faster 3D renderer for the web that sacrifices rendering quality for quick 3D drawing.

`OPENGL`
: This is a high-speed 3D graphics renderer that uses OpenGL-compatible graphics hardware if available. Keep in mind that OpenGL is not magic pixie dust that makes any sketch faster (though it's close), so other rendering options may produce better results depending on the nature of your code. Also note that with OpenGL, all graphics are smoothed: the `smooth()` and `noSmooth()` methods are ignored.

Lighting in Processing

Creating the two simplest primitive types of 3D objects is quite easy: call either `box()` or `sphere()`. If you don't want the box or sphere to have stroke lines, then call the `noStroke()` method before the drawing calls; otherwise, leave it in:

```
void setup() {
    size(400, 400, P3D); // set up the 3D renderer
}
void draw() {
    noStroke(); // commenting this line out will show the lines in the sphere
    //lights(); // uncommenting this will show the model with lights
    fill(255);
    translate(100, 100, 0);
    sphere(40);
}
```

Figure 13-1 shows the different ways that a sphere can be rendered depending on the options set.

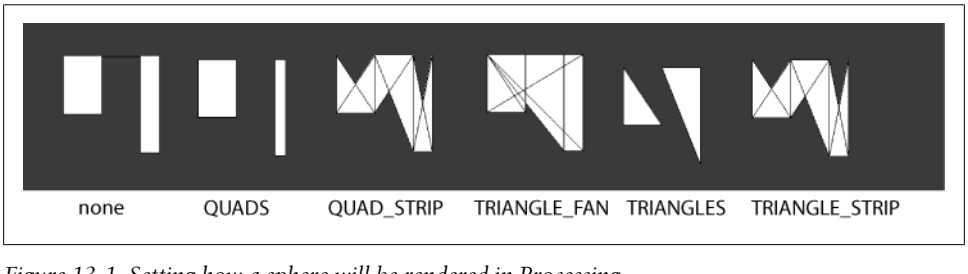


Figure 13-1. Setting how a sphere will be rendered in Processing

Notice that without calling the `lights()` method, the sphere looks suspiciously like a regular circle. Calling the `lights()` method makes the sphere look three-dimensional because without any lights positioned anywhere, there's no way for the renderer to know which parts of the sphere are darker and which are lighter. Lighting is one of the keys of representing three-dimensionality. Without it, there is no way for a viewer (or the renderer, for that matter) to know what the three-dimensional shape of an object is.

Lights in Processing are created using a number of methods that let you create and control the lighting in your scene:

`lightFalloff()`
: This sets how the light falls off, or dims, away from the source. Just like the `fill()` method, it affects only the elements created after it, not all elements in your code. If you wanted a region of your scene to be lit ambiently by one color and another region to be lit ambiently by another color, you would use an ambient light with a specific location and falloff.

`lightSpecular()`
: Sets the specular color for lights. *Specular* refers to light that bounces off a surface in a preferred direction and is used for creating highlights. Later in this section when you learn about the `specular()` and `shininess()` methods, you'll see how lights interact with surfaces. For now just imagine that the specular quality of a light is the amount of reflectivity that a surface will have.

  There are four kinds of lights that you can create in Processing. Remember that the lighting methods need to be called in the `draw()` method of your application so that they will be redrawn each time the program loops through the drawing method.

`ambientLight()`
: Adds an ambient light. Ambient light doesn't come from a specific direction. The rays of light have bounced around so much that objects are evenly lit from all sides. Ambient lights are almost always used in combination with other types of lights.

`directionalLight()`
: Adds a light that comes from one direction without having a particular location. These lights are stronger if they hit a surface squarely and weaker if they hit at an angle. It is called as shown here:

  ```
  directionalLight(v1, v2, v3, nx, ny, nz)
  ```

  The color is set using the first three parameters: red, green, blue. The `nx`, `ny`, and `nz` parameters indicate the direction the light is facing.

`pointLight()`
: Adds an ambient light with a position. Lights need to be included in the `draw()` method to remain persistent in a looping program:

  ```
  pointLight(v1, v2, v3, x, y, z)
  ```

  The color is set using the first three parameters: red, green, blue. The `x`, `y`, and `z` parameters set the position of the light.

`spotLight()`

Adds a spotlight that is like a directional light, except that it has a location that can be altered:

```
spotLight(v1, v2, v3, x, y, z, nx, ny, nz, angle, concentration)
```

The `x`, `y`, and `z` parameters specify the position of the light, and `nx`, `ny`, `nz` specify the direction or light. The `angle` parameter affects the angle of the cone of the spotlight and how wide it is, and the `concentration` parameter determines how much brighter the light is at the center of the cone.

Controlling the Viewer's Perspective

Lights are only half the story. They are important to show how *objects* are three-dimensional, but to make the *world* fully three-dimensional, you need to have control over how the viewer perceives the world. In most 3D platforms, this is by providing methods or access to an object that a programmer can modify to more easily manipulate the point from which the viewer sees the scene. In Processing, this is just called the `Camera`. The lighting objects and the `Camera` interact in much the way that lights and cameras do on a film set. The methods of the `Camera` class are also quite similar to the properties and operations of a real-life camera. The easiest way to add a 3D camera to a scene is to call the `camera()` method. A camera actually has three vectors: its location, the position that it's pointed at, and which direction is up. Figure 13-2 shows how these three vectors relate.

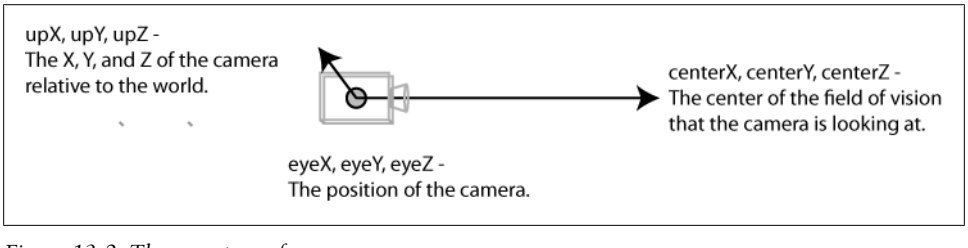


Figure 13-2. Three vectors of a camera

Moving the eye position and the direction it is pointing allows the objects to be seen from different perspectives. The version without any parameters sets the camera to the default position, pointing to the center of the display window with the y-axis of the world as the `upY` value.

You can also call the `camera()`method with the following optional parameters:

```
camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
```

One thing to remember is that the center coordinates and eye coordinates operate independently of one another, so moving the camera does not change what the camera is looking at. You'll usually want to do the two in tandem. The following small

application draws a cube and moves the camera around depending on the user's mouse position:

```
float eyeX, eyeY, eyeZ;
float centerX, centerY, centerZ;

boolean setCenter = false;

void setup() {
    size(300, 300, P3D);
    noFill();
    eyeX = width/2.0;
    eyeY = height/2.0;
    eyeZ = 20;
    centerX = width/2.0;
    centerY = height/2.0;
    centerZ = 0;
}

void draw() {
    lights();
    background(255);
    camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ,  0.0, 1.0, 0.0);
    fill(122);
```

Draw a simple box so that there's some reference for how the camera is moving:

```
    box(45);
}

void mouseMoved() {

        eyeX = width/2 - mouseX;
        eyeY = height/2 - mouseY;
}

void mouseDragged() {
    centerX += mouseX - pmouseX;
    centerY += mouseY - pmouseY;
}

void mouseReleased() {
    setCenter = false;
}

void mousePressed() {
    setCenter = true;
}

void keyPressed() {
```

Here the `printCamera()` method prints the values of all the camera's vectors. When you run this code and press Enter, you'll notice that 16 values are printed. This is because the camera's matrix is actually a 4 × 4 matrix, not a 3 × 3:

```
        if(keyCode == 10) { // enter button
            printCamera();
        }
        if(setCenter) {
            if(keyCode == 38) { eyeZ++; } // up arrow
            if(keyCode == 40) { eyeZ--; } // down arrow
        } else {
            if(keyCode == 38) { centerZ++; } // up arrow
            if(keyCode == 40) { centerZ--; } // down arrow
        }
    }
```

This code could just as easily be changed to use an accelerometer (introduced in Chapter 7), a set of dials, a light, or another color source (as shown in Chapter 10); or a gesture detection (as you'll see in Chapter 14). The interesting part is how quickly most users will be able to figure out the view onto the 3D world.

There are three more advanced methods for changing the way that the camera behaves. The `frustum()` method allows you to change the perspective of the camera. This method lets you set the locations and angles of the clipping planes that the camera uses. The *clipping plane* is a somewhat mathematically complex topic, but understanding how to use it and how it functions is fairly simple. Take a look at the diagram in Figure 13-3.

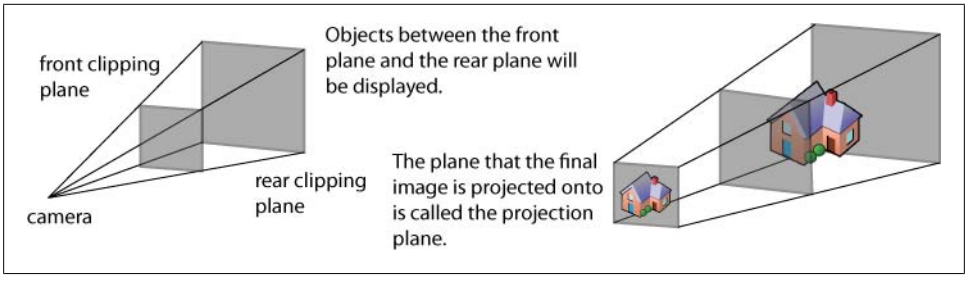


Figure 13-3. Clipping plane and projection plane

You can see in the diagram that the two planes define what appears in the projection that's created from the world and shown on the user's screen. This is called the *frustum*. A frustum is the portion of a cone or pyramid that lies between two parallel planes. In computer graphics the term frustum describes the 3D area that is visible on the screen because the visible area is formed by a clipped pyramid. Calling the `frustum()` method changes the positions of the rear clipping and front clipping planes and hence changes what appears on the screen. Here are the parameters that the `frustum()` method takes:

```
frustum(float left, float right, float bottom, float top, float near, float far)
```

Figure 13-4 shows how the `left`, `right`, `bottom`, and `top` parameters correlate to the positions where the new plane will be positioned. The `near` and `far` parameters control the positioning of the front and back of the frustum.

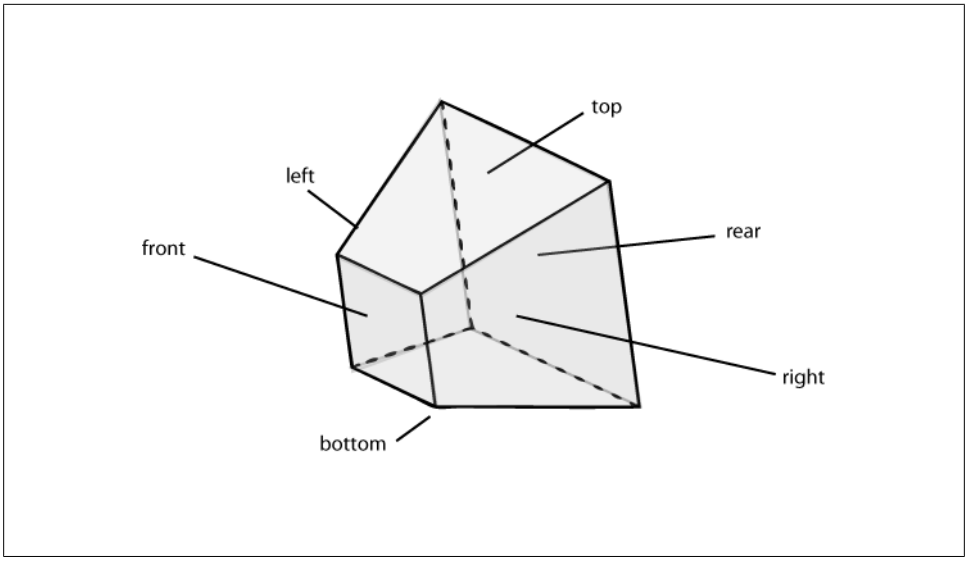


Figure 13-4. The viewing frustum

If you want to experiment with the `frustum()` methods, try setting up different variables for the camera settings and then setting up `keyPressed()` event handler methods to change each variable. What you'll see is that the frustum formed by the front and rear clipping panes of the application is transformed by the values passed into the `frustum()` method.

Another way of changing the perspective of the camera is to call the `perspective()` method. This simulates the perspective of the world more accurately than orthographic projection. The `perspective()` method without parameters sets the default perspective, but with four parameters it lets you set the area precisely. Here is the signature of the `perspective()` method followed by the definition of the parameters:

```
perspective(fov, aspect, zNear, zFar)
```

`fov`
: The field-of-view angle for vertical direction expressed in radians. This represents the amount of angular shift that should be used to change what the user sees.

`aspect`
: The ratio of width to height.

`zNear`
: The z-position of nearest clipping plane, that is, the closest something can be and still be seen.

`zFar`
: The z-position of farthest plane, that is, the farthest something can be and still be seen.

Calling the `perspective()` method won't change the object or camera, but it will change how the viewer sees the object (Figure 13-5).

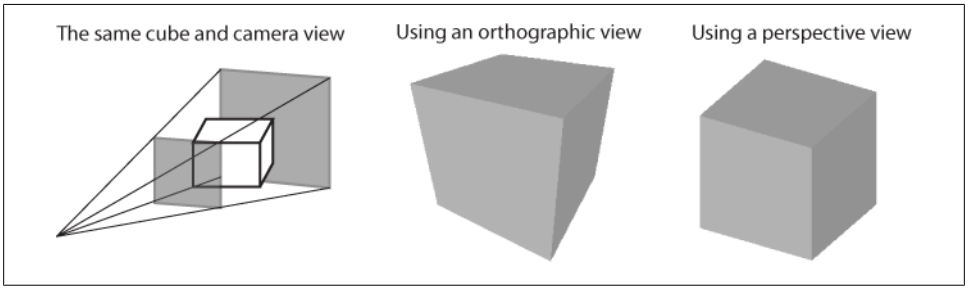


Figure 13-5. Changing the appearance of an object by setting the perspective view

Making Custom Shapes in Processing

The first element to understand in making three-dimensional shapes for Processing is the vertex. A *vertex* is a point in 3D space that has `x`, `y`, and `z` properties that determine where it is in relation to the 0, 0, 0 point (you can think of this as the center) of the world. A vertex might have the values 100, 100, 100. This means that it is 100 pixels on the x, y, and z coordinates from the center of the world in which it is located. In Processing, all shapes are the result of the connections between vertices. A pyramid is a construction made of the connections between five vertices, a cube is made up of the connections between eight vertices, a line is made up of the connections between two vertices. You create a vertex by using the `vertex()` method and passing in the coordinates. A 2D vertex looks like this:

```
vertex(x, y);
```

A 3D vertex looks like this:

```
vertex(x, y, z);
```

The `x`, `y`, and `z` parameters can be float or int values. It's important to note that this call doesn't create anything, and there isn't a "Vertex" class that you create instances of and store. Instead, the vertex communicates with the underlying graphics engine of your Processing application, telling it where to place a vertex. Drawing a vertex in 3D using the `z` parameter requires the `P3D` or `OPENGL` parameter in combination with size.

The main use of the `vertex()` method is to set the points for drawing a 3D shape. There is another use that we'll examine later in the section "Using Matrices and Transformations in OpenGL" on page 493, in the discussion on textures and mapping them, but for now, consider a vertex as the way to set the points to be used for the corners of a 2D or 3D shape.

To create a custom 3D shape, you'll need to tell the underlying graphics engine that you're going to creating the points for shape first using the `beginShape()` method, then

define all the points for the shape, and finally tell the graphics engine that you're finished setting the points for the shape using the `endShape()` method.

So, the first step is to call the `beginShape()` method, then call the `vertex()` method as many times as you need to set all the points for your shape, and then call the `endShape()` method. The call to the `beginShape()` method is what determines the way that all the subsequent vertices will be connected. The following are the different modes for this method. Figure 13-6 shows how these modes are used.

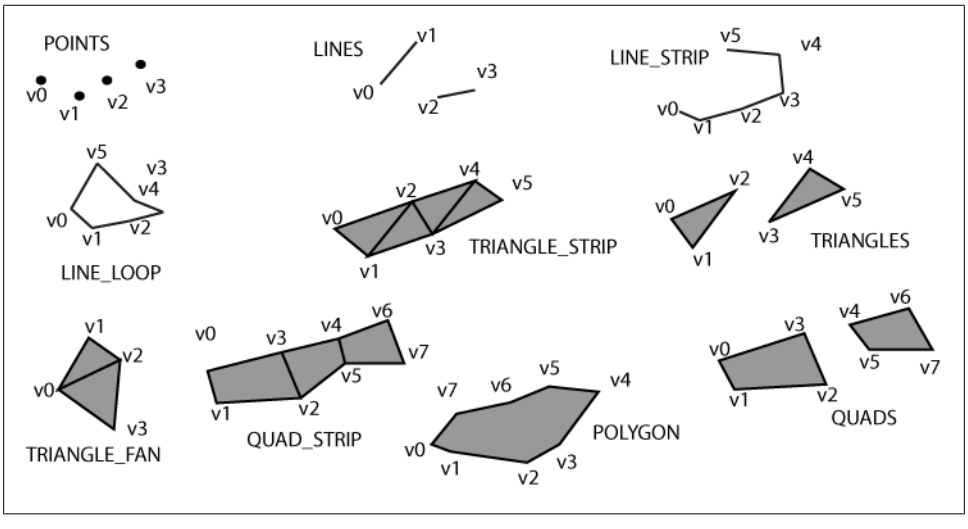


Figure 13-6. The GL modes that can be passed to the beginShape() method

`POINTS`
: Creates a shape without any connections in between the vertices.

`LINES`
: Connects the vertices with lines but no fill.

`TRIANGLES`
: Connects the vertices with triangles. Each vertex will be used to create one point of a triangle.

`TRIANGLE_FAN`
: Connects the vertices with multiple triangles. Each vertex will be connected with multiple equilateral triangles.

`TRIANGLE_STRIP`
: Connects the vertices with multiple triangles. Each vertex will be connected with multiple isosceles triangles.

`QUADS`
: Connects the vertices with as few quadrilaterals as possible.

`QUAD_STRIP`
: Connects the vertices with as many quadrilaterals as possible, placing multiple quadrilaterals at each vertex.

The following code creates eight vertices and connects them according to the primitive mode passed to the `beginShape()` method:

```
void setup(){
    size(640, 360, P3D);
}

void draw(){
    background(50, 50, 50);
    translate(width/2, height/2);
```

The default shape is POLYGON:

```
    beginShape();
        vertex(0, 0);
        vertex(0, 75);
        vertex(50, 75);
        vertex(50, 0);
        vertex(100, 0);
        vertex(100, 125);
        vertex(125, 125);
        vertex(125, 0);
    endShape();
}
```

This will produce the drawing shown on the far left of Figure 13-7. Note that the difference between the `QUADS` and the default `beginShape()` is the small connecting line between the two rectangles. The default drawing mode connects all vertices with lines and creates fills where the lines enclose an areas. The `QUADS` mode simply creates quads and fills for all of the vertices.

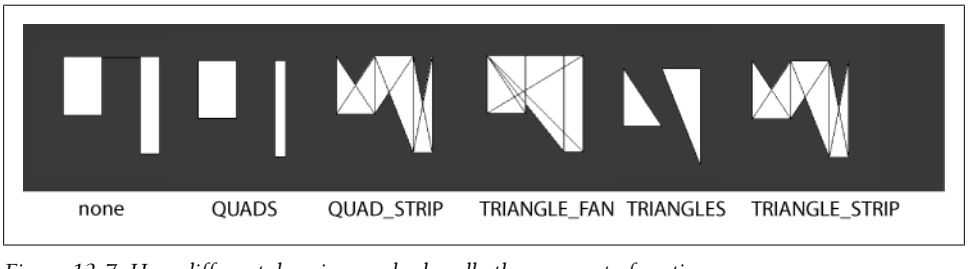


Figure 13-7. How different drawing modes handle the same set of vertices

Note that you can't use transformations like `translate()`, `rotate()`, and `scale()` inside a `beginShape()` block, nor can you use the shape drawing methods like `ellipse()` or `rect()`. Also, don't forget to always finish with a call to `endShape()`. Only vertex calls can be used in that drawing block. After you've drawn the shape, then you can use `translate()`, `rotate()`, and `scale()` to change the position of the drawing objects.

Using Coordinates and Transforms in Processing

The previous code snippets have touched on how to move objects and views around in 3D in a Processing application, but these all merit a closer examination.

There are two coordinate systems in Processing to consider: the coordinates of the world and the coordinates of the screen. The difference is the location of the 0, 0, 0 point. Although a model may have its 0, 0, 0 point in one location, because of rotations and transformation applied to the model, it will appear in an entirely different location on the screen as in Figure 13-8.

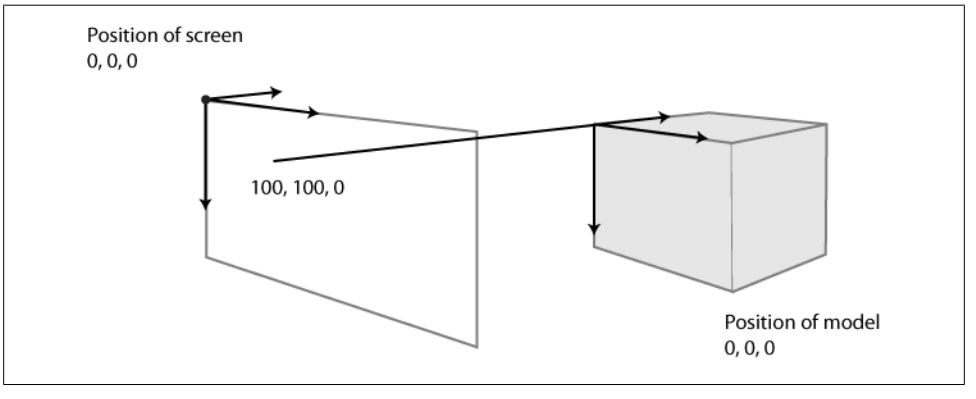


Figure 13-8. The model co-ordinate system and the screen co-ordinate system may not be the same values

Imagine that you have a cube you've drawn as shown in the code snippet below:

```
void setup()
{
    size(400, 400, P3D);
}

void draw()
{
    pushMatrix();
        translate(200, 200, 20);
        rotateY(PI/3.0);
        rotateZ(PI/3.0);
        box(100);
    popMatrix();
    // not very exciting ☺
    rect(100, 100, 100, 100);

}
```

After the call to `popMatrix()`, what is the screen position of the corner of the box? What is the actual position of the box relative to the point at the center of the world? You could do the math to get these, but there's an easier way. Three methods return a

position in 3D space relative to a model: `modelX()`, `modelY()`, and `modelZ()`. These return the values for a given coordinate based on the current set of transformations (scale, rotate, translate, and so on). For instance, you transform the x value to be 200 pixels to the right, then you tilt the world matrix a little. As soon as you call `popMatrix()`, the value of the 0 point of the model vanishes. You won't be able to place something else at that same value without calling all the same methods again and that might be a problem. Luckily, if you store the value from `modelX()`, before you call `popMatrix()`, then you can use that value wherever you would like. If you want to place another cube in the same location as the first cube, before the call to `popMatrix()`, add the `modelX()`, `modelY()`, and `modelZ()` calls as shown here:

```
void draw()
{
    pushMatrix();
        translate(200, 200, 20);
        stroke(255);
        fill(0);
        rotateY(PI/3.0);
        rotateZ(PI/2.0);
```

This next code gets the location of the models 0, 0, 0 point in screen coordinates with all the transforms applied:

```
        float transX = modelX(0, 0, 0);
        float transY = modelY(0, 0, 0);
        float transZ = modelZ(0, 0, 0);
        box(50);

    popMatrix();

    translate(transX, transY, transZ);
    stroke(0);
    fill(255, 20);
    box(100);

}
```

In the code snippet, adding calls to `modelX()`, `modelY()`, and `modelZ()` methods records the location of the box in space after being placed using a series of `translate()` and `rotate()` commands. After `popMatrix()` is called, those transformations vanish along with any data about what would have been the 0 point within them, but those coordinates can be captured by the `modelX()`, `modelY()`, and `modelZ()` methods.

All three of those methods take the same parameters:

`x`
: int or float; 3D x coordinate to be mapped

`y`
: int or float; 3D y coordinate to be mapped

`z`
: int or float; 3D z coordinate to be mapped

These enable another cube to be placed in the same location as the original cube using all the original transform data, but this doesn't always do what you might want. For example, while the world position of a cube might be in one location, it appears in an entirely different place on the screen because of camera movements, or rotations. To place an object based on another object's position according to the screen coordinates, you'll want to use the `screenX()`, `screenZ()`, and `screenZ()` methods. These take the same parameters and are used in the same way as the model coordinate methods but use the x-, y-, and z-positions of the screen instead of the world within the screen.

Working with 3D in OpenGL

If you're not interested in Processing, you might have skipped the opening sections to this chapter that discussed OpenGL. If you did, you might want to take a look at those sections. Though they don't explicitly discuss OpenGL, they do help you understand the basics of how OpenGL works. If you're using the OpenGL mode Processing methods, simply call the underlying OpenGL methods that your graphics card uses to create the graphics that you see on your screen. If you're not using the OpenGL mode, then the Processing graphics methods don't actually use OpenGL but they do mimic OpenGL calls. The graphics methods contained in `ofGraphics` class do the same things. Like the Processing calls, you can use the `ofGraphics` methods to do all your drawing, but if you're looking to create something sophisticated and complex, you'll very quickly find that you might need a little more control over and a better understanding of your code.

So, What Is OpenGL?

OpenGL is an API written in a programming language that will look pretty familiar to you from using Arduino and oF code in particular, because it's loosely based on C. At its core, OpenGL does one thing: it hides a lot of the complexities of interfacing with different 3D drivers by making one way to program for any drivers that communicate between an operating system and a graphics card. There is another specification for a very similar language called DirectX that is used on Windows computers but since that isn't used by Processing or openFrameworks, it won't be addressed in this book.

OpenGL's core functionality is to help a programmer create code that creates points, lines, and polygons and then convert those objects into pixels. The conversion of objects into pixels is called the *pipeline* of the OpenGL renderer. You'll see that OpenGL is pretty low level, in much the same way that the Processing graphics API can create complex objects from sets of vertices and instructions on how to connect them. OpenGL is a low-level, procedural API, requiring the programmer to dictate the exact steps to render a scene. This contrasts with descriptive approaches like the Processing drawing API, where a programmer needs only describe a scene and can let the library manage the details of rendering it. OpenGL's low-level design requires programmers to have a good knowledge of the graphics pipeline but also gives a certain amount of

freedom to implement novel rendering algorithms that can help your code run substantially faster.

Transformations

Transformations are the modification of one of the coordinate systems, either the model coordinate system space as relative to a single object or the world coordinate system space relative to all the objects. They are performed using the `translate()` method of Processing. This section introduces you to something that is very important in OpenGL: the transformation stack. Let's review. The `translate()` method moves the location where objects will be drawn, so if you call the following:

```
translate(200, 200, 0);
```

and then the following:

```
rect(0, 0, 100, 100);
```

your rectangle will be drawn 200 pixels from the left and 200 pixels from the top of the application window. When you call `pushMatrix()`, you're setting up a transform that will be used for all drawing operations until you call the `popMatrix()` method.

OpenGL uses the same ideas of transforms and a transformation stack. When you create a transform, it alters all the operations that follow it until it's pushed off the stack. That stack can consist of multiple transforms that all affect any drawing operation that follows it. The other way of drawing that you've seen in this chapter—creating vertices and having the renderer generate shapes based on those vertices—are all based on the way that OpenGL works.

OpenGL in Processing

OpenGL in Processing uses a class called `GL` that provides access to all the OpenGL methods. Now, there's a few caveats to using OpenGL in Processing that make it exciting, interesting, powerful, and difficult, because the purpose of Processing and the intent of the designers of the language was to ensure that you, the Processing user, didn't have to get into using OpenGL. There are things that may work in one release of Processing and suddenly break in another, there are things will work fine in oF code that won't work correctly in Processing, and there's all kinds of hardware incompatibilities and strangeness that you may encounter. That said, if you're confident that you want to work in Processing and you really need the power that the OpenGL renderer is going to provide you, read on. We'll get you started, but because there are so many potential pitfalls and problems, it doesn't make sense to delve too deeply into OpenGL in Processing.

The first step in using OpenGL is making sure that the Processing application knows to use the OpenGL renderer to create any graphics that are supposed to be displayed

on the screen. You indicate that by passing `OPENGL` to the `size()` method when you initialize your application:

```
size(800, 600, OPENGL);
```

Once you've done that, Processing uses the GL rendering engine to create all the graphics. That doesn't mean that your calls to `vertex()`, `color()`, `rotate()`, or any other Processing 3D methods will change. You'll still be using the same methods to create your graphics. If you want to delve deeper into OpenGL in Processing, even more caveats apply. If you glance at the reference page for the OpenGL library in the Processing documentation, you'll see the following:

> These were added against my better judgment and handle setting up the camera matrices (though still no lighting or other parameters). Many features still will not work, but it opens up lots of control for geometry. All the above caveats about "don't cry when it breaks" apply.

So, keep that in mind. That said, to get started, you'll simply need to add the following `import` statements to the top of your application:

```
import processing.opengl.*;
```

This imports not only the Processing OpenGL library but also the underlying Java OpenGL library called JOGL (Java OpenGL). The calls to the JOGL library are nested between two calls to initialize and then finalize the JOGL drawing operations: `beginGL()` and `endGL()`. All Processing applications contain a `PGraphics` variable called `g`, and that variable can be cast to the `PGraphicsOpenGL` variable type. You might remember casting from Chapter 5. (If not, take a look back at it.) You should place these two calls in the `draw()` method of your application:

```
void draw() {
PGraphicsOpenGL pgl = (PGraphicsOpenGL) g;
GL gl = pgl.beginGL();
```

`GL` is the Java media class that provides you with access to the raw OpenGL methods. Earlier in this chapter you saw how to pass the OPENGL renderer to the `size()` method of your Processing application. That tells the Processing engine to turn your Processing calls, like `vertex()`, into actual OpenGL calls, like `glVertex3fv()`. That might look unfamiliar but it actually does the same thing as the `vertex()` call. You can only call an actual OpenGL method like `glVertex3fv()` on the `PGraphicsOpenGL` object.

Once you've made the `GL` object, you can call `GL` methods as shown here:

```
gl.glColor4f(1.0, 1.0, 1.0, 1.0);
```

Finally, the call to the `endGL()` method ends the OpenGL calls:

```
pgl.endGL();
```

Now, as mentioned, this is not necessarily the best way to create graphics for two or three dimensions in Processing, but it does have some bonuses, and it does allow you to leverage OpenGL in a Processing application. For the rest of this chapter though, the OpenGL information will focus on openFrameworks.

Open GL in openFrameworks

openFrameworks uses OpenGL for all of its graphics drawing, but most of the calls are hidden. It actually uses an implementation of OpenGL called Graphics Language Utility Toolkit (GLUT) by default. However, if you want to change to a different GL implementation, you're free to do so. All graphics calls in the `ofGraphics` class use calls to common OpenGL methods, which you can see if you open the class and take a look at what goes on in some of the methods. For instance, the `ofLine()` method contains the following lines:

```
glBegin( GL_LINES );
    glVertex2f(x1,y1);
    glVertex2f(x2,y2);
glEnd();
```

You'll see something that should look familiar from other ways of drawing that you've encountered earlier in this chapter. The method of connecting the vertices is set when initializing the drawing. Next, the vertices are set, and then a method is called to finish the drawing. This looks pretty familiar. Using OpenGL with oF doesn't come with any caveats like Processing, since all the drawing code is, in fact, OpenGL, and the GL context isn't dependent on JOGL like Processing. You don't need to do anything special to begin using OpenGL code in an oF application. openFrameworks does a lot of the work to initialize and set up the GL context: creating the 3D window for drawing, setting the initial size of that window. If you open *ofAppGlutWindow.cpp* and look at the `setupOpenGL()` function, you'll see the following:

```
glutInit(&argc, vptr);
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH | GLUT_ALPHA );
```

When you start up an oF application that is going to use the GLUT mode, this is one of the first functions that is called. You'll see that this function calls `glutInit()` to initialize the GLUT library, followed by `glutInitDisplayMode()` to set the mode of the display.

The `ofRunApp()` method initializes what is called the OpenGL context. It calls a series of functions that set up the OpenGL window and indicates the GLUT callback functions that take place during the draw loop, first `update()` and then `draw()`. The `ofAppGlutWindow` class contains the critical GLUT functions that take place every frame that your application runs. If you look at the `display()` function, you'll see that it sets up the screen, calls your application's `draw()` function, and increases the frame count. The `idle_cb()` method is the idle function for GLUT, which is called after each frame is drawn. Inside `idle_cb()`, you'll see that it maintains your application's frame rate and calls its `update()` function before calling the `draw()` method. Understanding how all these calls work isn't vital to being able to use OpenGL, but it can be helpful to see what's going on under the hood in the oF core.

Using Matrices and Transformations in OpenGL

There are three important matrices that you'll deal with in 3D. Two of them are shown in Figure 13-9.

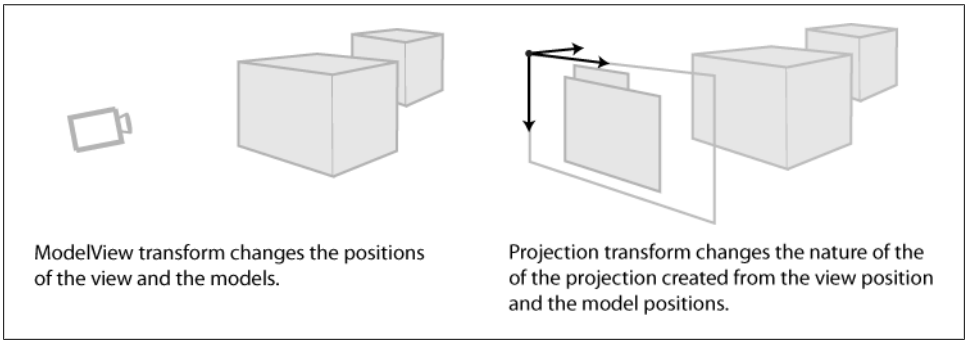


Figure 13-9. How ModelView and Projection transforms change the models and view

`ModelView`

This matrix defines the position and orientation of the camera and the world. A model is any object that you draw, and the view is defined by the position of the viewer. Changes made to this matrix alter the coordinates of the world in which the camera and objects are positioned. It is loaded by passing the `GL_MODELVIEW` constant to the `glMatrixMode()` method. You need to call the `glMatrixMode()` method before you try and do any model transformations and then load the identity as shown here:

```
glMatrixMode(GL_MODELVIEW); // specify the modelview matrix stack
glLoadIdentity(); // load it with an empty matrix
```

The *identity matrix* is like a blank slate matrix. It has an identical number of rows and columns and all the elements in which i=j are set to 1 and all others are set to 0. A 3 × 3 identity matrix looks like this:

```
| 1 0 0|
| 0 1 0|
| 0 0 1|
```

We'll be talking more about the identity matrix and how it's used later in this section. To get back to the transformations, it's a bit complex to understand how the viewing and the modeling transformations act because in the end they have the same effect: moving the view forward in the direction of the objects or moving all the objects backward in the direction of the camera is more or less the same thing. The term *ModelView* indicates that there is no distinction between the modeling and the viewing transformations, and the principle behind which transformation it really is, is left to the programmer. So, the `ModelView` transformation can be used to transform the camera position or the model, depending on how it is used.

`Projection`
: This defines how the camera, that is, the viewer, sees the world. Changing this matrix alters how the "world" is seen for the viewer, though it does not change the positions of any of the objects in the world including the view position. It is loaded by passing the `GL_PROJECTION` constant to the `glMatrixMode()` method and is the last transformation performed. It finalizes what you will see on the screen by establishing the 2D view that the camera will see based on 3D coordinates of all visible objects and the camera position.

`Texture`
: This is the matrix that defines the position of the texture that will be applied to shapes. For each texture, you'll have a different texture matrix.

A transformation is the modification of one of the coordinate systems, and you'll perform these by either using the `glTranslate()`, `glScale()`, or `glRotate()` method. For instance, to change the positions of objects or the camera position, you load the `ModelView` matrix as shown here and then set some transformations:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // load the identity matrix to clear out the matrix values
glTranslatef(230, 130, 0);
```

Once a transformation is performed, each new object you create is drawn according to the position of the new coordinate system, which is what makes the objects appear from a different point of view. You're not directly moving the camera; you're simply creating a mathematical model of how everything should be positioned when it's drawn. That's the idea behind transformations: they are a mathematical shortcut that helps imitate real-world movements and appearances.

As in Processing, once you modify the coordinate system, any objects created subsequently will be modified. You can even save the current state of the coordinate system using `glPushMatrix()`, retrieve it later after more transformations are applied, and then use it as a starting point for subsequent transformations. This allows you to place objects at any position and angle you want and be able to draw objects that rotate around other rotating objects (simulating orbiting planets, for example).

All transformations are consecutive. It is up to the programmer to come up with the right order of transformations to achieve a desired effect. For example, consider a modeling transformation of an object. Let's say you have a sphere built around the center of its local coordinates. Take a look at Figure 13-10. In the diagram on the left, the position of the circle does not change with respect to the origin, only the rotation changes. In the diagram on the right, both the position and the rotation change. This is because the order of the transformation and rotation calls are reversed.

To help you get a quick grasp on how these are used in an oF application, take a look at the following `draw()` method from an oF application. This will place two `ofImage` objects on the stage and modify them using the `glTranslate()` and `glRotate()` methods:

```
void testApp::draw(){
    // enable the tranformation matrix
    glMatrixMode(GL_MODELVIEW);
    // load the identity matrix
    glLoadIdentity();
    glTranslatef(230, 130, 0);
    // start a new tranformation matrix (based on the state of the previous one)
    glPushMatrix();
         // move 230-pixels across and 30-pixels up
         glTranslatef(130, 30, 0);
         // rotate the image 45 degrees along the z-axis
         glRotatef(90, 0, 0, 1);
         // draw an image
         img1.draw(0, 0);
    // jump to the previous transformation matrix (no rotation,
 no translation, no scaling)
    glPopMatrix();
    // start a new tranformation matrix
    glPushMatrix();
         // rotate the image 45 degrees along the z-axis
          glRotatef(90, 0, 0, 1);
         // draw the second image (translating 230-pixels across and 30-pixels up)
         img2.draw(130, 30);
    // go back to the previous transformation matrix
    glPopMatrix();
}
```

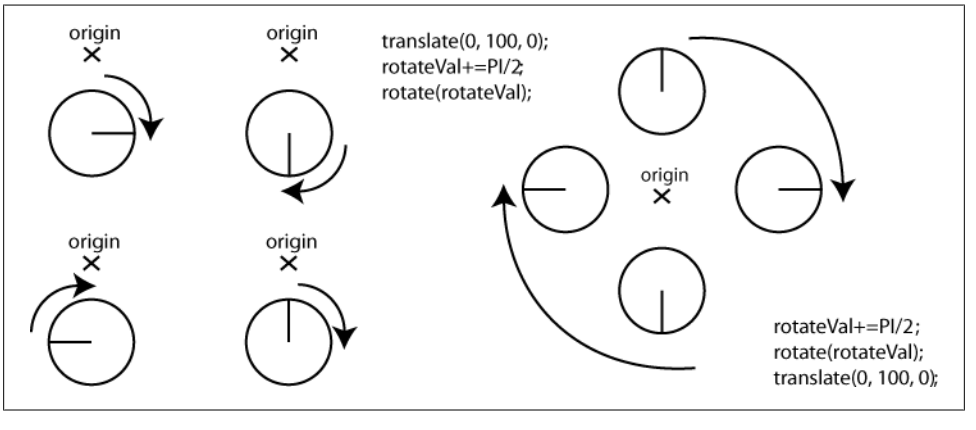


Figure 13-10. How the order of transforms affects the result of a drawing operation in Processing

Note the explicit call to `glMatrixMode()` to set the `GL_MODELVIEW` as the current matrix stack. By default, the `GL_MODELVIEW` is the stack altered by transform methods, so if you want to alter the view onto the world instead, you'll use `glMatrixMode(GL_VIEWPORT)`. In this case, though, it's the model and the view that are being altered.

Using Vertices in OpenGL

If you remember how the vertices in Processing work or how the `ofPoint()` method in oF works, you already understand a lot of what you need to know about making vertices in OpenGL. An OpenGL vertex is what you're creating by using the Processing `vertex()` method or the oF `ofPoint()` method. It defines a point in space that will be connected with other points by the renderer according to the primitive drawing mode that you indicate. Creating a simple square in OpenGL looks like this:

```
glBegin(GL_QUADS);
    glVertex2i(10, 10); // top left
    glVertex2i(100, 10); // top right
    glVertex2i(100, 100); // bottom right
    glVertex2i(10, 100); // bottom left
glEnd();
```

That probably looks quite familiar. Set the way that the vertices are going to be connected, create some vertices, and then signal the end of the drawing operation to connect them. Both in Processing and oF, the vertices can handle different values passed into them. In OpenGL, you need to be more specific. You'll notice that the vertex methods in the previous code have the number of parameters and type appended to them:

```
glVertex2i(100, 100);
```

This specifies that there are two parameters and that both are integers. To use three floating-point variables, you would use the `glVertex3f()`method. To use four double variables, you would use `glVertex4d()`.

The drawing modes that can be passed to the `GL_QUADS`, for example, works for drawing rectangles and squares, and `GL_POLYGON` works for drawing multisided shapes. `GL_LINE` works for drawing lines. See the *The OpenGL Programming Guide* (also sometimes called "The Red Book") by Mason Woo et al. (Addison-Wesley) for more information on primitives. The valid values to pass to the `glBegin()` method are `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_QUADS`, `GL_QUAD_STRIP`, and `GL_POLYGON`. These should look pretty familiar to you because they're the same options that can be passed to the Processing `beginShape()` method except with a `"GL_"` appended to the front. These function in the same way, too, so to avoid unnecessary repetition if you're curious how each of these modes work, glance back to the section "Making Custom Shapes in Processing" on page 484.

Drawing with Textures in oF

In pure OpenGL, if you want to draw a cube, there isn't a convenience method like the Processing `box()` method. However, drawing a cube is really just a matter of drawing six planes and arranging them. In the following example, you'll do just that. The header

file (Example 13-1) defines an `ofImage` to load a picture and an `ofTexture` that will use the image to create a texture.

Example 13-1. glCubeApp.h

```
#include "ofMain.h"

class glCubeApp: public ofBaseApp{

    public:

        ofTexture text;
        ofImage img;

        float ang;

};

#endif
```

Next, the glCubeApp (Example 13-2) has a `setup()` method to load the image and create an `ofTexture` from it.

Example 13-2. glCubeApp.cpp

```
void glCubeApp::setup(){

    img.loadImage("tmp2.jpg");
    text.allocate(400, 400, GL_RGB, true);
    ang = 0;
```

There's a slight problem with drawing six quadrilateral planes to represent a cube. You have no way to tell the graphics card which quads should be drawn in front of the others. They'll be shown in the order that they're drawn, which is hard if you're going to turn the cube around because you would need to draw different sides in different orders depending on which is facing the camera. There's a concept that can help you with this problem and save you some time: depth testing. *Depth testing* simply means that the graphics card will look at which quad should be placed in front, so it saves you the trouble. To turn depth testing on, call the `glEnable()` method, and pass it the `GL_DEPTH_TEST` constant:

```
        glEnable(GL_DEPTH_TEST); // enable depth testing, otherwise things will
            //look really werid
        glDepthFunc(GL_LEQUAL);    // set the type of depth testing
        text.loadData(img.getPixels(), 400, 400, GL_RGB);
    }

    void glCubeApp::update(){
        ofBackground(122,122,122);
        ang+=0.2;
    }

    void glCubeApp::draw() {
```

You'll need to enable the texture, telling the graphics card to use that texture for any drawing operations that occur until the `glDisable()` method is called. Since drawing the cube is the only drawing operation that will be performed in this section, there's no need to call `glEnable()` multiple times:

```
glEnable(text.getTextureData().textureTarget);
glTranslated(400, 150, 0);
glRotated(ang, 1.0, 0.0, 0.0);
```

Tell the graphics card to connect all the vertices with quads:

```
glBegin(GL_QUADS);
```

Now, draw each side of the cube, one side at a time. As in the Processing example earlier, each vertex is defined, and then the position of the texture attached at that vertex is set:

```
// left side
glTexCoord3f(0.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 20.0);
glTexCoord3f(0.0, 200.0, 0.0); glVertex3f(0.0, 200.0, 20.0);
glTexCoord3f(200.0, 200.0, 0.0); glVertex3f(0.0, 200.0, 200.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 200.0);

// bottom side
glTexCoord3f(0.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 20.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 200.0);
glTexCoord3f(200.0, 200.0, 0.0); glVertex3f(200.0, 0.0, 200.0);
glTexCoord3f(0.0, 200.0, 0.0); glVertex3f(200.0, 0.0,20.0);

// top side
glTexCoord3f(0.0, 0.0, 0.0); glVertex3f(0.0, 200.0, 20.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(0.0, 200.0, 200.0);
glTexCoord3f(200.0, 200.0, 0.0); glVertex3f(200.0, 200.0, 200.0);
glTexCoord3f(0.0, 200.0, 0.0); glVertex3f(200.0, 200.0, 20.0);

// back side
glTexCoord3f(0.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 20.0);
glTexCoord3f(0.0, 200.0, 0.0); glVertex3f(0.0, 200.0, 20.0);
glTexCoord3f(200.0, 200.0, 0.0); glVertex3f(200.0, 200.0, 20.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(200.0, 0.0, 20.0);

// front side
glTexCoord3f(0.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 200.0);
glTexCoord3f(0.0, 200.0, 0.0); glVertex3f(0.0, 200.0, 200.0);
glTexCoord3f(200.0, 200.0, 0.0); glVertex3f(200.0, 200.0, 200.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(200.0, 0.0, 200.0);

// right side
glTexCoord3f(0.0, 0.0, 0.0);glVertex3f(200.0, 0.0, 20.0);
glTexCoord3f(0.0, 200.0, 0.0);glVertex3f(200.0, 200.0, 20.0);
glTexCoord3f(200.0, 200.0, 0.0);glVertex3f(200.0, 200.0, 200.0);
glTexCoord3f(200.0, 0.0, 0.0); glVertex3f(200.0, 0.0, 200.0);
```

```
        glEnd();

    }
```

Your next question might be, how do I draw a sphere and apply a texture to it? The answer is not too difficult, but it involves touching on a lot of topics that we're not going to be able to explore fully. In the spirit of this book, though, we'll show places you can go for further research in the section "What to Do Next" on page 513. Creating a sphere is as simple as calling the `gluSphere()` method. It has the following signature:

```
gluSphere (GLUquadric* quad, GLdouble radius, GLint slices, GLint stacks );
```

This method takes three parameters that should make sense and one that probably won't. `radius` is the size of the circle. `slices` is the number of vertical strips in the sphere, and `stacks` is the number of horizontal strips in the sphere, so the more `stacks` and `slices`, the slower the rendering but the smoother the sphere. The one element in there that you're not going to recognize is the `GLUquadric` object, and it's a big topic. Basically, all these objects are created from groups of squares, called *quads*. A cube has six quads, a roughly drawn sphere might have 500 quads, and a smoothly drawn large sphere might have 4,000. The `GLUquadric` object is simply a pointer to the first element in an array of `GLUquadric` objects, and each of those quadrics represents one quad in a 3D object like a sphere. To create a new `GLUquadric` pointer, you use the `gluNewQuadric()` method as shown here:

```
GLUquadric * sphere;
sphere = gluNewQuadric();
```

To change the previous cube example to create a sphere, you'll need to make a few changes. Add the following line to the *.h* file:

```
GLUquadric * sphere;
```

In the *.cpp* file, change the `setup()` method of the application to the following:

```
void glSphereApp::setup(){

    img.loadImage("tmp2.jpg");
```

Usually—and this is important—this last parameter, the `bUseARBExtention` parameter, needs to be set to false:

```
    text.allocate(512, 512, GL_RGBA, false);
    ang = 0;

    glEnable(GL_DEPTH_TEST); // enable depth testing, otherwise
      //things will look really werid
    glDepthFunc(GL_LEQUAL);    // set the type of depth testing
```

Make sure to enable the type of texture that you want to use. In this case, our texture is a 2D texture represented by the `GL_TEXTURE_2D` flag passed to the `glEnable()` method:

```
    glEnable ( GL_TEXTURE_2D );
    sphere = gluNewQuadric();
    glPixelStorei ( GL_UNPACK_ALIGNMENT, 1 );
```

```
        gluQuadricDrawStyle( sphere, GLU_FILL);
        gluQuadricNormals( sphere, GLU_SMOOTH);
        gluQuadricTexture( sphere, GL_TRUE );
        text.loadData(img.getPixels(), 512, 512, GL_RGB);

    }
```

You'll also need to change the `draw()` to draw the sphere:

```
    void glSphereApp::draw() {
```

The image data is still loaded into the texture in the same way:

```
        glRotated(ang, 1.0, 0.0, 0.0);
        glTranslatef(350, 250, 0);
        // enable texturing
```

The `glEnable()` call is necessary and tells the graphics card how to use the texture on the sphere that will be drawn. In this case, the texture is a 2D image, so you'll use `GL_TEXTURE_2D`:

```
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, text.getTextureData().textureID);
```

Finally, call the `gluSphere()` method to draw the sphere, and then call `glDisable()` to signal that you're finished with the texturing operation:

```
    gluSphere(sphere, 200, 50, 50);
        glDisable(GL_TEXTURE_2D);
    }
```

Lighting in OpenGL

Lighting in OpenGL looks somewhat like lighting in Processing, but uses a slightly more complex syntax. First, in what should be becoming a familiar pattern, to turn the lights on, you'll need to call `glEnable()`:

```
    glEnable(GL_LIGHTING);
```

Then, you'll need to set a light:

```
    glEnable(GL_LIGHT0);
```

This creates a light that can be used when you assign the properties of those lights, like so:

```
    glLightf(GLenum light, GLenum pname, GLfloat param);
```

The first parameter, `light`, would be name of the light, in this case `GL_LIGHT0`. The second, `pname`, would be the type of light, such as `GL_AMBIENT or GL_DIFFUSE`. The third parameter, `param`, is an array of four floats that set the color and brightness of the light. To add lighting to any of the previous oF examples, change the `setup()` method to look like the following:

```
    void lightingApp::setup(){
        glEnable(GL_LIGHTING);
```

```
        glEnable(GL_LIGHT0);

        glEnable(GL_DEPTH_TEST); // enable depth testing, otherwise things
                                 //will look really weird
        glDepthFunc(GL_LEQUAL);    // set the type of depth testing
```

Create light components with 3-color floating point values and a fourth for brightness:

```
        GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
        GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
        GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
        GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

        // Assign created components to GL_LIGHT0
        glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
        glLightfv(GL_LIGHT0, GL_POSITION, position);
    }
```

Now you've added lighting to your application.

Blending Modes in OpenGL

At some point you're going to want to blend your objects and images. You might want to draw transparent 3D object, or you might want to overlay and blend multiple images, or you might want to filter out a certain color in an image. In OpenGL, this is done via blending, which combines the color data already in the graphics card with the incoming color data from the new object being drawn. If you've drawn one cube and are drawing another, then the `blendFunc()` method tells the graphics card how to display the two cubes together. By default, the most recently drawn cube will be drawn over the other cube, but you can set how they appear using the blending modes. As with many other things, a number of constants are defined in OpenGL that allow you to select the blending mode. In this list, you're going to find two terms: *source*, which means the color data from the objects already in the buffer, and *destination*, which means the new incoming colors from the object being drawn.

OpenGL thinks of colors as floating-point numbers, so adding them, subtracting them, and even multiplying them are all perfectly valid operations. Here's the list of all the different flags that you can pass to the `blendFunc()`, telling it how to handle both the source and the destination data:

`GL_ZERO`
: Tells the graphics card to set all the color data to which this flag is applied, either the source or the destination, to zero.

`GL_ONE`
: Tells the graphics card to set this data to full.

`GL_DST_COLOR`
: Tells the graphics card to blend the destination color data into the incoming color data.

`GL_ONE_MINUS_DST_COLOR`
: Tells the graphics card to invert the destination color data by subtracting it from 1 and returning that value.

`GL_SRC_ALPHA`
: Tells the graphics card to use the source alpha values to determine which of the destination colors will appear.

`GL_ONE_MINUS_SRC_ALPHA`
: Tells the graphics card to subtract the invert all the colors depending on the strength of the sources alpha value.

`GL_DST_ALPHA`
: Uses only the alpha values of the destination data.

`GL_ONE_MINUS_DST_ALPHA`
: Uses the values of the destination minus 1.

`GL_SRC_ALPHA_SATURATE`
: Takes the minimum value of each the two alpha values compared.

Each of these can be set to handle the incoming data and the existing data by passing it to the following method:

```
glBlendFunc( GLenum sfactor, GLenum dfactor );
```

The `glBlenFunc()` method takes two parameters:

`sfactor`
: Specifies how the red, green, blue, and alpha source blending factors are computed.

`dfactor`
: Specifies how the red, green, blue, and alpha destination blending factors are computed.

It's quite difficult to visualize how these operate; perhaps the best way to do this is to create a simple application that will enable you to apply different blend modes to both the source and destinations of the `glBlendFunc()` method. A short sample application that does just this is shown here.

To the standard *testApp.h* file add the following, as in Example 13-3 (I'd like to thank Ira Greenberg for providing the inspiration for this demo).

Example 13-3. blendApp.h

```
int limit;
float ang;
int blendL[14]; // an array to store all the blending values
int blendR[14]; // an array to store all the blending values
int currentL, currentR; // indexes to store the current blending value
```

```
ofImage img; // an image to load a picture for a texture
ofTexture text; // a texture to use
int cubePos[60]; // an array to store all the text values
```

The definition of the file contains only one new method, the `drawCube()` method. The following code listing is truncated to conserve space and avoid repetition, though it is available in full in the code downloads for this book:

```
#include "blendApp.h"
```

In the `setup()` method, you'll simply set the values of two arrays to each possible blending value. Each array will have an index integer stored that indicates which blending mode should be used:

```
void blendApp::setup(){
    limit = 20;
```

These are set to `GL_ONE_MINUS_SRC_ALPHA` for the source and `GL_SRC_ALPHA` for the destination since these are the most commonly used settings:

```
currentL = 7;
    currentR = 6;
    img.loadImage("tmp.jpg");
    text.allocate(256, 256, GL_RGB, true);
    text.loadData(img.getPixels(), 256, 256, GL_RGB);

    blendL[0] = GL_ZERO;
    blendL[1] = GL_ONE;
    blendL[2] = GL_SRC_COLOR    ;
    blendL[3] = GL_ONE_MINUS_SRC_COLOR;
    blendL[4] = GL_DST_COLOR    ;
    blendL[5] = GL_ONE_MINUS_DST_COLOR;
    blendL[6] = GL_SRC_ALPHA    ;
    blendL[7] = GL_ONE_MINUS_SRC_ALPHA;
    blendL[8] = GL_DST_ALPHA    ;
    blendL[9] = GL_ONE_MINUS_DST_ALPHA;
    blendL[10] = GL_SRC_ALPHA_SATURATE;
    blendL[11] = GL_CONSTANT_COLOR    ;
    blendL[12] = GL_ONE_MINUS_CONSTANT_COLOR ;
    blendL[13] = GL_CONSTANT_ALPHA ;
    blendL[14] = GL_ONE_MINUS_CONSTANT_ALPHA;

    blendR[0] = GL_ZERO;
    blendR[1] = GL_ONE;
    blendR[2] = GL_SRC_COLOR    ;
    blendR[3] = GL_ONE_MINUS_SRC_COLOR;
    blendR[4] = GL_DST_COLOR    ;
    blendR[5] = GL_ONE_MINUS_DST_COLOR;
    blendR[6] = GL_SRC_ALPHA    ;
    blendR[7] = GL_ONE_MINUS_SRC_ALPHA;
    blendR[8] = GL_DST_ALPHA    ;
    blendR[9] = GL_ONE_MINUS_DST_ALPHA;
    blendR[10] = GL_CONSTANT_COLOR    ;
    blendR[11] = GL_ONE_MINUS_CONSTANT_COLOR ;
```

```
        blendR[12] = GL_CONSTANT_ALPHA ;
        blendR[13] = GL_ONE_MINUS_CONSTANT_ALPHA ;
```

Here, we populate the `cubePos` array with random values to place each of the cubes:

```
        for (int i = 0; i< limit * 5; i++){
            cubePos[i] = ofRandom(-200.0, 200.0);
            printf("%i ", cubePos[i]);
        }
```

Don't forget to enable depth testing using the `glEnable()` method:

```
        glEnable(GL_DEPTH_TEST); // enable depth testing, otherwise
            //things will look really werid
        glDepthFunc(GL_LEQUAL);  // set the type of depth testing
        ofSetFrameRate(20); // slow things down a little
    }

    void testApp::update(){
    }

    void blendApp::draw(){
        ofBackground(122, 122, 122);

        int w, h;
        h = ofGetHeight();
        w = ofGetWidth();
```

To blend different objects, you need to enable blending by calling `glEnable()` with the `GL_BLEND` constant passed to it:

```
        glEnable(text.getTextureTarget());
        glEnable(GL_BLEND);
        glTranslated(w/2, h/2, -300+mouseX);
        //rotate around y and x axes
        glRotated(ang, 1.0, 1.0, 0.0);
```

The call to `glBlendFunc()` sets the blending mode to use on all the preceding drawing operations:

```
        glBlendFunc( blendL[currentL], blendR[currentR] );

        for (int i = 0; i < limit * 3; i+=3) {
            drawCube(cubePos[i], cubePos[i+1], cubePos[i+2],
     cubePos[i+3]/2, cubePos[i+4]/2, cubePos[i+5]/2);
        }
```

Finally, indicate that the drawing and blending operation is finished:

```
        glDisable(GL_BLEND);
        //used to rotate the view
        ang+=0.2;
    }
```

In the `keyPressed()` method, there is simply a listener that lets you or the user toggle through the different blending modes available:

```
void testApp::keyPressed  (int key){
    switch( key ){
    case 'e':
          if(currentL > 13)
              currentL = 0;
          else
              currentL++;
          printf(" changed r to %i ", currentL);
          break;
    case 'd':
          if(currentL < 0)
              currentL = 13;
          else
              currentL--;
          printf(" changed r to %i ", currentL);
      break;
    case 'r':
          if(currentR > 13)
              currentR = 0;
          else
              currentR++;
          printf(" changed r to %i ", currentR);
      break;
    case 'f':
          if(currentR < 0)
              currentR = 13;
          else
              currentR--;
          printf(" changed r to %i ", currentR);
      break;
  }
}
```

The `drawCube()` method draws a cube to the screen:

```
void blendApp::drawCube(int shiftX, int shiftY, int shiftZ, int h,
    int w, int d)  {
    //front face
    glBegin(GL_QUADS);
    glTexCoord3f(0, 0, 0); glVertex3f(-w/2 + shiftX, -h/2 + shiftY,
        -d/2 + shiftZ);
    glTexCoord3f(256, 0, 0); glVertex3f(w + shiftX, -h/2 + shiftY,
        -d/2 + shiftZ);
    glTexCoord3f(256, 256, 0); glVertex3f(w + shiftX, h + shiftY,
        -d/2 + shiftZ);
    glTexCoord3f(0, 256, 0); glVertex3f(-w/2 + shiftX, h + shiftY,
        -d/2 + shiftZ);
    . . .
}
```

To conserve space and avoid repetition, the rest of the method is omitted from this code listing but is available in the code downloads for this book.

Blending modes in Processing work much the same way as in oF, so with just a little bit of tweaking you can turn the preceding example into a Processing application. The only major difference is that `glBlendFunc()` is a method of the Processing `GL` object rather than a call like `glBlendFunc()` which is not called on an object.

Using Textures and Shading in Processing

Processing has a way of creating textures and shading that avoids a lot of the complexities of working with textures and shading in OpenGL. The idea of a *texture* is one that you've already encountered in Chapter 8, where you learned how bitmaps and bitmapped textures are displayed. In this section, you'll see how to apply those textures to 3D objects created in Processing. The first step to understanding this is to revisit the `vertex()` method. Remember that the `vertex()` method creates a vertex by using three numbers to define its location in space, but there are two additional parameters that you can pass to the `vertex()` method:

```
vertex(x, y, z, u, v); // note the u and v parameters
```

`u`
: Can be an int or float and designates the horizontal coordinate for the texture mapping.

`v`
: Can be an int or float and designates the vertical coordinate for the texture mapping.

In Figure 13-11, a 500 × 500 pixel image is used as a texture, which is then applied to a set of vertices. Notice, the `u` and `v` values are referring to the pixel position in the image, so the full image is seen only in the final example where the values passed to the `u` and `v` parameters of the lower-right vertex are the same as the size of the image.

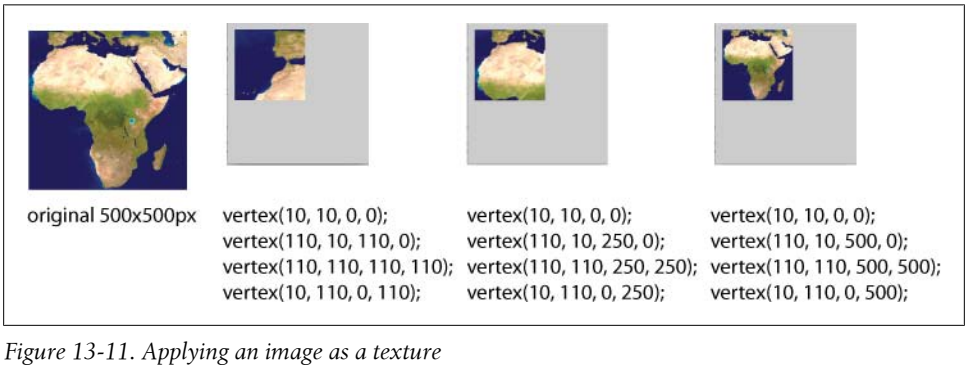


Figure 13-11. Applying an image as a texture

Here is how you apply the image as a texture:

```
void setup() {
    size(200, 200, P3D);
}

void draw() {
    noStroke();
    PImage a = loadImage("tmp.jpg");
    beginShape();
    texture(a);
    vertex(10, 10, 0, 0);
    vertex(110, 10, 250, 0);
    vertex(110, 110, 250, 250);
    vertex(10, 110, 0, 250);
    endShape();
}
```

You can't simply apply the texture to a box or sphere created with the `box()` or `sphere()` methods. There is another method of setting the way that objects appear when drawn: you can set the materials that will be applied as the texture and appearance of those objects.

Applying Material Properties

You can set a few different properties for any object that is drawn in Processing. If you've added an ambient light component to an application using the `ambientLight()` method, you can set the reflectiveness of an object by using the `ambient()` method. There are two ways to call the method: `ambient(R, G, B)`, `ambient(gray)`, or `ambient(color)`, where a `color` is passed in. These set how the objects surface reflects the ambient light. Higher values mean it reflects more light and is brighter, and lower values mean it reflects less light.

If you want your objects to emit light, like a light bulb, then use the `emissive()` method to set the amount of brightness and color that the object emits. As with the `ambient()` method, the `emissive()` method can be called like so: `emissive(gray)`, `emissive(color)` where a `color` object is passed in, or `emissive(red, greed, blue)` with each of the values representing the channel.

To set the amount of gloss in the surface of shapes, use the `shininess()` method, which is passed a floating-point value from 0.0 (no shine) to 1.0 (which reflects all light from any directional light that touches it). It doesn't, however, reflect other objects. To do that, you'll need to use a more powerful renderer, like the surfaceLib library by Andreas Köberle and Christian Riekoff.

Finally, you can use the `specular()` method to set the `specular` property of the material for an object. This sets how the material reflects a directional light, but unlike the `shininess()` method, the `specular()` method sets how a directional light is reflected in the same direction that the light emanates from. It is called in the same way as the other

methods: `specular (grey)`, `specular (color)` where a `Color` object is passed in, or `specular (v1, v2, v3)`.

Using Another Way of Shading

GLSL (Graphics Language Shading Language) is an abbreviation for the official OpenGL Shading Language. GLSL is a high-level programming language that's similar to C/C++ for several parts of the graphics card. With GLSL, you can code short programs, called *shaders*, which are executed on the GPU.

What Does GLSL Look Like?

A shading language is a special programming language adapted to easily map on shader programming. Those kinds of languages usually have special data types such as color and normal. Because of the various target markets of 3D graphics, different shading languages have been developed.

GLSL shaders themselves are a set of strings that are passed to the graphics card drivers for compilation from within an application using the OpenGL API's entry points. Shaders can be created on the fly from within an application or read in as text files, but they must be sent to the driver in the form of a text string.

There are two fundamental aspects of a shader: *the vertex shader* and *the fragment shader*. The OpenGL shader pipeline works roughly as shown in Figure 13-12.

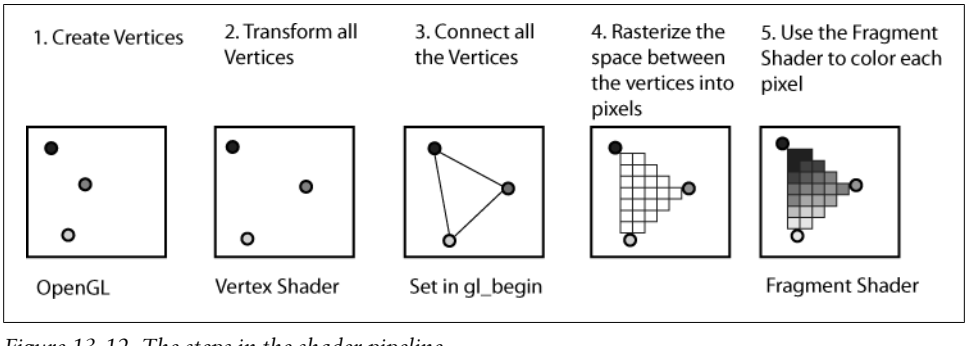


Figure 13-12. The steps in the shader pipeline

In step 2 of Figure 13-12, the vertex shader can be applied to alter the positions of different vertices. In step 5, the fragment shader is applied to control the shading of each pixel. We'll look at each part of this process individually.

Vertex Shaders

A *vertex shader* has attributes about a location in space or vertex, which means not only the actual coordinates of that location but also its color, how any textures should be

mapped onto it, and how the vertices are modified in the operation. A vertex shader can change the positions of each vertex, the number of lighting computations per vertex, and the color that will be applied to each vertex.

Your application certainly doesn't need to do all of these operations to use a vertex shader. As you'll see in the next code example, you don't need to apply lights to your objects. Even though you don't need to use all the functionality of the vertex shader, when you create and apply a vertex shader, your program will assume that you're replacing all the functionality of your other vertex transformations. If you apply changes to particular vertex, the vertex shader will probably change those according to the instructions contained in the vertex shader. When a vertex shader is used, it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

The vertex processor processes each vertex individually and doesn't store any data about the other vertices that it has already processed or that it will process. It is responsible for at least writing a variable, `gl_Position`, usually transforming the vertex with the `ModelView` and `Projection` matrices. A vertex processor has access to OpenGL state, so it can perform operations that involve lighting for instance, and use materials. It can also access textures.

So, what does a vertex shader look like? They can be quite simple. Take a look at the following example. It's a single function that sets the value of each vertex passed to it. That vertex can be accessed within that method as `gl_Vertex`, and its final position after the operation can be set using the `gl_Position` variable:

```
void main() {
```

Here the vertex that is currently being operated on is used to create a new vertex:

```
    vec4 v = vec4(gl_Vertex);
```

As a quick example, the `x` position of the vertex is shifted over by 1 pixel:

```
    v.x += 1;
```

Now the final position of the vertex is set:

```
    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

That's a very simple vertex shader that doesn't alter your image in any immediately noticeable way, but it does show you how the vertex shader functions.

Fragment Shader

The fragment shader is somewhat misleadingly named because what it really allows you to do is to change values assigned to each pixel. The vertex shader operates on the vertices, and the fragment shader operates on the pixels. By the time the fragment shader gets information passed into it by the graphics card, the color of a particular pixel has already been computed and in the fragment shader can be combined with an

element like a lighting effecting, a fog effect, or a blur among many other options. The usual end result of this stage per fragment is a color value and a depth for the fragment.

The inputs of this stage are the pixel's location and the fragment's depth, whether it will be visible or altered by an effect, and color values.

Just like vertex shaders, fragment shaders must also have a `main()` function. Just as the vertex shader provides you access to the current vertex using the `gl_Vertex` variable, the fragment shader provides you access to the current fragment, that is, the fragment that the fragment shader is working with when it is called, using the `gl_FragColor` variable. Like the `gl_vertex`, this is a four-dimensional vector that lets you set the RGB and alpha properties of the fragment. In the following example, the color of the fragment is set to a medium gray with full alpha:

```
void main() {
    gl_FragColor = vec4(0.5, 0.5, 0.5, 1.0);
}
```

Notice how each of the channels is set using a floating point number from 0.0 to 1.0. With this in mind, we can move on to the oF add-on for working with vertex and fragment shaders.

Variables Inside Shaders

Shaders can have three kinds of variables. *Uniform* variables are the same for all pixels or vertices they describe, and *attribute* variables are calculated separately for each vertex. One way to think of a uniform variable is that it works in the same way that static variables work in C++ as discussed in Chapter 7: setting the uniform once sets it for all vertices. An attribute is calculated for each vertex. Finally, there are variables marked as *varying* variables, which are shared across both the shader and the fragment rendering stages of the pipeline. This means that if you have a 4D vector value declared as varying in your vertex shader:

```
varying vec4 normalColor
```

then you'll need to make sure that you have the same declaration in your fragment shader. This is so that your graphics card can share the value of the variable through each stage of the rendering process. A value can be set in the vertex rendering stage and then used in the fragment rendering stage. Lots of complex lighting effects are done this way.

Using an ofShader Addon

As with so many other things, if you're looking to work with shaders in oF, there's an add-on for you to use. There's no getting around the difficult nature of working with shaders, but they're very interesting to play with and can be very powerful if you learn

how to use them. It's worth showing a simple example of how to use the ofShader add-on.

To get started with a using shaders in oF, you'll need to define a vertex and fragment shader file and save both of those in the data folder of your oF application. You can use either the examples from later in this section or from the previous section. Once you've done that, you can download and then import the ofShader add-on the same as you would with any other add-on file. Then declare an `ofShader` instance in your application's *.h* file:

```
ofShader shader;
```

This won't do anything until the `loadShader()` method of the `ofxShader` object is called:

```
shader.loadShader("color"); // this loads color.frag and color.vert
```

The string should be the name of both the fragment shader file and the vertex shader file. They both have to have the same name.

Once you've loaded a shader, you can decide whether you want to activate or deactivate it using the `setShaderActive()` method:

```
void setShaderActive(bool bActive);
```

There is a `setUniform()` method that allows you to set uniform variables in your shaders. Say you have a value that you want to be able to set in your shader, for instance `red`. In your *color.frag* file, you would have the uniform declared like in Example 13-4.

Example 13-4. color.frag

```
uniform float red;
void main() {
    gl_FragColor = vec4(red, 1.0, 1.0, 1);
```

In your oF application can set this value by using the `setUniform()` method:

```
Shader.setUniform ("red", 100);
```

Here's the signature of the method:

```
void setUniform (char * name, float v1);
```

The method also has overloaded versions to accept up to four floating-point values, up to four short values, or up to four double values. You can also set the attributes of the shader using the `setAttribute()` method.

You're probably ready to see a working oF application that uses these methods. In the following example, a fragment shader and a vertex shader are loaded into the application and then used to shade a cone created using `glutSolidCone()` and a cube using `glutSolidCube()` as in Examples 13-5 and 13-6.

Example 13-5. color.vert

```
uniform float xTweak;
uniform float yTweak;
```

```
void main()
{
    gl_FrontColor = gl_Color;
    vec4 v = vec4(gl_Vertex);
    v.x +=  xTweak;
    v.y +=  yTweak;
    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Example 13-6. color.frag

```
uniform float red;
uniform float green;
void main()
{
    gl_FragColor = vec4(sin(5.0 * red), sin(5.0 * green), 1.0, 1);
}
```

These two files should be saved in the data folder of the application. Next is the oF application, which creates an `ofShader` object and loads the shader files using the `loadShader()` method, as shown in Example 13-7.

Example 13-7. shaderApp.h

```
#ifndef _SHADER_APP
#define _SHADER_APP
#include "ofMain.h"
#include "ofShader.h"

class shaderApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        void mouseMoved(int x, int y );

        ofShader shader;
};

#endif
```

Now the *.cpp* file for the application that handles actually loading and displaying the shader (Example 13-8).

Example 13-8. shaderApp.cpp

```
#include "shaderApp.h"

void shaderApp::setup(){
```

Here is the call to actually load the shader and then to set it to active:

```
        shader.loadShader("color");
        shader.setShaderActive(true);
    }

    void shaderApp::update(){
    }
```

Next, two shapes are drawn. The vertex shader file modifies the vertices of each shape, and the fragment shader file alters the pixel values of each shader:

```
    void shaderApp::draw(){
        ofSetupScreen(); // call this to do the plain vanilla GL environment setup
        ofBackground(255,255,255);
        glPushMatrix();
            glTranslated(900, 400, 0);
            glutSolidCube(300);
        glPopMatrix();

        glPushMatrix();
            glTranslated(250, 400, 0);
            glRotatef(mouseX, 1.0, 0, 0);
            glutSolidCone(100, 200, 50, 50);
        glPopMatrix();
    }
```

The uniform values in the shaders can be set according to the mouse movement or any other kind of input data or movement:

```
    void shaderApp::mouseMoved(int x, int y ){
        shader.setUniform("xTweak", float( sin(x / 50.0) * 20.0));
        shader.setUniform("yTweak", float (cos(y / 50.0) * 20.0));
        shader.setUniform("green", (float) x / ofGetWidth());
        shader.setUniform("red", (float)y / ofGetHeight());
    }
```

That's just the very tip of what you can do with shaders and GLSL, but it should be a start that will get you going.

What to Do Next

If you're interested in learning more about OpenGL, you'll want to pick up *The OpenGL SuperBible* by Richard Wright (Sams), which is also known as the "Blue Book" because of the color of its cover. This is the oldest and best way to learn all that you to know about OpenGL to become a proficient graphics programmer. Another useful text is *The OpenGL Programming Guide* by Mason Woo et al., also called the "Red Book," which is the resource for learning about the GL language. If you're interested in shading or working more with shaders, you'll want to look at *The OpenGL Shading Language Guide* by Randi J. Rost (Addison-Wesley), also called the "Orange Book." Both the Red Book and the Orange Book, as well as their code samples, are freely available online as of the writing of this book.

A lot of other books promise to teach you about graphics, but you're probably better off buying one of the official OpenGL books, reading online tutorials, or picking up one of the Processing books that covers graphics in a more in-depth manner such as *Processing: Creative Coding and Computational Art* by Ira Greenberg (Apress), *The Nature of Code* by Daniel Shiffman (*http://www.shiffman.net/teaching/nature/*), or *Processing* by Casey Reas et al. (MIT Press).

Review

To draw 3D in Processing, pass the `P3D` or `OPENGL` constant as the third parameter for the `size()` method when starting your application.

If you've set up the rendering correct in Processing, you can create 3D points, shapes, and the `sphere()`, `box()`, and `cone()` methods.

In Processing, calling the `lights()` method sets up basic lights in your applications. You can change how the lighting appears by calling the `lightFalloff()` and `lightSpecular()` methods, and you can add new lights using `directionalLighting()`, `pointLight()`, `spotLight()`, or `ambientLight()`.

Processing makes working with views easy by providing you with a `camera()` method to set up a camera. You can also call the `camera()` method like so: `camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)`.

In Processing, vertices are 3D points created with the `vertex()` method that will be connected to form a shape by the mode passed to the `beginShape()` method.

In Processing, to change the position or rotation of the viewer or the location of an object to be drawn, use the `pushMatrix()` method to get a new matrix, and then use the `rotation()` and `transformation()` methods to change the coordinates at which things will be drawn. Call `popMatrix()` when you are finished.

OpenGL can be used in Processing and oF applications, though it is somewhat more standardized in oF applications. It communicates with the graphics card of your computer.

OpenGL uses matrices and transformations quite similarly to Processing. OpenGL has three matrices: `ModelView`, `Projection`, and `Texture`. There are different transformations that allow you to work with these matrices easily: the `Model` transform, `ModelView` transform, `Projection` transform, and the `Viewport` transform.

To draw custom shapes with textures in oF, use the `glTexCoord()` method to set how an `ofTexture` will be mapped to a vertex created with `glVertex()`.

You can set blending modes using the `glBlendFunc()` method, which takes two parameters: a value for the existing data already rendered and a value for the incoming data for all incoming activities.

Processing lets you create materials by using the `ambientLight()`, `emissive()`, `shininess()`, and `specular()` methods. These determine whether an object emits light and how it reacts to environmental light.

Graphics Language Shading Language (GLSL) can be used in oF by using the ofShader add-on. Shading happens in two distinct steps: the vertex shader creates values for each vertex in the model, and the fragment shader creates values for each pixel in the rendered object. To define a shader, create a *.frag* file for the fragment shader and a *.vert* file for the vertex shader.

CHAPTER 14
Detection and Gestures

This chapter is going to be largely about one tool, OpenCV, and will be quite heavy on openFrameworks code. This is an important chapter because it is where you'll learn how to turn the gestures and movements of a user, or other kinds of movement in an image, into meaningful input for an application. As anyone paying attention to the development of devices and the advancement of user interface and experience concepts over the past few years can tell you, this topic is one of the most interesting and important ideas in contemporary device design. Surfaces, tangible interaction devices, and free gesture interfaces have been incorporated in many different kinds of devices in the past few years.

Computer vision is a field of computing concerned with creating "seeing" applications by feeding an application a series of images, like a video, or a single image. Usually these images are analyzed for something in particular, such as looking for a light or for a face. Unlike human sight, a computer needs a good deal of instruction on what it should be looking for in an image. Computer vision techniques can be used to find fingertips, track gestures, find faces, or simply detect motion. Images are analyzed by looking for patterns in color values of the pixels. Those changes can be grouped into objects and tracked over multiple images, or they can be analyzed using statistical algorithms to determine how likely something is to represent a certain object like a face or other specific object.

This chapter talks about computer vision, recognition, and gestures together because one of the more interesting ways to create interaction using computer vision is to use gesture and recognition. This allows the user to input data into a program through an image given to the computer. *Recognition* is the detection of a particular kind of object in an image, a face, a hand, or a particular image. A *gesture* is a pattern of movement over time. You can use both of these with Processing or oF applications by using the OpenCV library for computer vision. Gestures recognition and face recognition are really a particular way of generalizing patterns from data, and in the case of computer vision, that data is image data. Any system trying to use gestural input needs to accurately distinguish one gesture from another, determine the extent of each gesture, and

determine and communicate whether it is a gesture that communicates a range of input values or functions as a binary input.

Computer Vision

Computer vision (CV) is a massive and complex topic that ranges in the disciplines it draws from. Statistical analysis, physics, and signal processing all are important contributors to the development of computer vision techniques and libraries. Some of the most important arenas where computer vision is being used are robotics, surveillance machines, weapons systems, games, toys, and touchscreen interfaces. Computer vision allows interaction designers to do what seemed impossible: create interfaces from the movement of the human body, without a need for an interface that the user physically manipulates. For a long time, applying computer vision techniques to interactive applications was unrealistic because the techniques were too time- and processor-intensive to allow real-time interaction and provide feedback quickly enough. However, newer processors and better algorithms and techniques for image processing have made it possible to analyze images from a video feed in near real time.

Artists and engineers have used gesture recognition and computer vision since the pioneering work of Myron Krueger in the 1970s. Krueger often talked about an art that was fundamentally about interactivity, rather than art that used interactivity as a gimmick. He focused on the idea of the interaction being the medium of communication, just like language, text, and pages are to a book or canvas and paint are to a painting. His ideas and work are worth a look for designers and for artists alike because even in the 1960s and 1970s he had a clear vision of how to connect people and virtual spaces.

There are a few key areas of research in computer vision. *Image processing* and *image analysis* tend to focus on 2D images and deal with how to transform one image into another. Some techniques for altering images were covered in Chapter 10. Another area of computer vision research focuses on the 3D scene projected onto one or several images and deals with reconstructing the scene from one or several images. *Machine vision* is concerned with processing of image data to control a robot or other kind of machine. It's most often concerned with inspection or measurement and focuses on precise calculations in controlled environments like factories. This gives the research and algorithms used by machine vision a slightly different bend than those used in analyzing uncontrolled situations where light changes quickly and unpredictably. *Pattern recognition* is another field that uses various methods to extract information from image data, looking for particular data that can represent a written letter or word, a fingerprint, or a kind of airplane, to give some common examples.

Computer vision generally, though not always, follows a particular sequence. First, the program needs to acquire the image through some process of digitalization. A digital image is produced by one or several image sensor that might be a light-sensitive camera, a range sensor, a radar, or an ultrasonic device. The pixel values typically correspond to light intensity in one or several spectral bands (gray images or color images) but can

also be related to various physical measures, such as depth, absorption, reflectance of sonic or electromagnetic waves, or nuclear magnetic resonance. Next, the program will do some kind of preprocessing, specifically, resampling the image, reducing noise in the image, blurring it, increasing the contrast, or performing other kinds of changes. You'll see this in the examples shown in this chapter; usually the images are converted to grayscale or blurred, or both, before being processed. Next, the program will look for features that it's interested in—lines, edges, ridges, corners, blobs, or points. This is often called *feature extraction*. Finally, the program will perform high-level processing on a small part of the data that it's actually interested in. For instance, in an application looking for faces, all things that could be faces are checked more closely to see whether they do or do not actually appear to be faces. In an application that finds a specific part of the body, an arm for instance, the actual processing of the shape of the arm will take place in this final step.

Interfaces Without Controls

One of the wonders of working with OpenCV and computer vision is that the physical interface can become optional. There is no requirement that the user touch anything. This dramatically frees you as the designer, allowing you to literally create an interface out of air. With the recent emphasis on lighter, more transparent, and more minimal interfaces using computer vision to capture user input seems thrilling. Before you get too excited by the concept, though, you need to consider the ramifications of that lack of interface. Interfaces are what they are because users require information about what information they can send, what might be done with the information that they send, and how that system to which they are sending the information will use the information.

Your messages to the user about what they are doing and how the system receives information about what they are doing becomes absolutely vital when making computer vision interfaces. You'll want to avoid the vague "you do something, it does something back" mode of interaction. While this can be fun to experiment with and is an excellent way to learn about computer vision and test certain ideas, it doesn't make for a particularly rich experience. One of the great realizations that started the flurry of touchscreen interfaces was the precise reactivity of the prototypes of Jeff Han's *Media Wall*. These were touchscreens that reacted to user movements in real time and tracked fingertips and gestures with great precision, making the interaction both precise and natural. Once the touchscreen started to become an interactive surface that was not only dynamic but was precise as well, it became possible to develop new and different kinds of applications with them. Touchscreens had been around for a long time before 2004, but they were ATM screens or kiosks where the user touched a button much the same way as if they clicked a mouse. When working with purely gestural interfaces, it becomes even more important to ensure that the modes of input and the feedback to the user are precise, timely, and intelligible. How you go about doing that is entirely up to you.

Creating a *gesture library*, a set of specific gestures that your application will look for and react to, is important if you're going to make an application that will use gestures, whether those gestures will be input into a touchscreen, will be in the air, or will be the movement of a body. Defining ahead of time what gestures your application will be looking for will make your decisions about how to structure your application and how to test it a great deal easier for you. With all the recent developments in touchscreens, there is an informal but growing library of recognized gestures: two fingers slide, pinch, tap to select, and so on. Dan Saffer wrote an excellent book called *Designing Gestural Interfaces* (O'Reilly) that details many of these gestures and what meaningful uses for them might be.

You should think about how to signal what a device is interpreting a gesture to be by providing either some textural feedback or some visual feedback. Many touchscreen devices draw a small white dot around the user's finger so that the user can see where the application thinks their finger is located. A few other suggestions to think about include making an easy help gesture that users can call and making sure that users are given prompts. If you're using gestures or specific actions, make sure you signal them clearly.

Example CV Projects

There are a wide range of projects that use motion tracking and computer vision techniques. The projects range from LaserTag, which tracks a laser pointer on a surface several meters away; to medical imaging, which helps detect cancerous tumors; to military applications; to projects like the NOR_/D Touchscreen that will be explored later in this chapter. This notion of the computer-mediated environment has percolated throughout the design, architectural, and artistic communities. Public artists like Greyworld make frequent use of computer vision. Pieces like *The Monument to the Unknown Artist* use the gestures of passersby to pose a robotic sculpture, relying on computer vision to determine the pose of the onlooker. Performance pieces like *Messa di Voce* by Golan Levin and Zachary Lieberman use computer vision to track the movements of a performer on a stage and particularly the motion of their heads. Computer vision is widely used in dance, in musical performance, and in video pieces in dance clubs to widen the physical space in a sense or to make a performer or a participant metaphorically "exist" in more than one place. Advertising or product demonstrations use computer vision to grab a viewer's attention with graphics that mirror their movements. These computer vision techniques are becoming popular for video artists in dance clubs, as attention getters in shopping malls, in building lobbies, and on the street to attract attention for an advertisement, as well as being tools for artists.

Some of the most remarkable achievements in computer vision have been much more oriented toward engineering. In 2005 a race called the Grand Challenge was held where entrants equipped a car with a computer vision system to navigate desert terrain for several hours. Gary Bradski was a member of the winning team and used OpenCV in the vision system. Almost all robots and robotics applications have some element of

computer vision to help them navigate terrain or analyze objects. Several companies are working with computer vision and gesture recognition to create gaming systems that operate without touch and without the need for a controller.

OpenCV

A team of developers at Intel headed by Gary Bradski started developing OpenCV in 1999. In 2007, Intel released version 1.0 as an open source project. Since it's an open source library, it's free to download, and you can use and modify it for your own purposes. Some of the core features that it provides are an interface to a camera, object detection and tracking, and face and gesture recognition. The library itself is written in C and is a little tricky to get started using right away without another library or framework to interface with it. Luckily, there are libraries that you can use with both Processing and oF.

ofxOpenCV is an add-on for openFrameworks that Stefan Hechenberger and Zachary Lieberman developed to let people working with oF use OpenCV easily in their projects. For Processing, both the OpenCV library—developed by Stephane Cousot and Douglas Edric Stanley—and the FaceDetect library utilize the OpenCV library. Because OpenCV is so massive, all the libraries implement a portion of its functionality. The ofxOpenCV library focuses on contour detection, as well as blob tracking and detection functionality. The Processing OpenCV library, as of the writing of this book, provides for real-time capture, image manipulation, and face and blob detection. The FaceDetect library for Processing is set up primarily to do face detection.

Using Blobs and Tracking

Movement tracking in OpenCV is done through what is commonly called a *blob*, which is a data object that represents a contiguous area of a frame. Blobs can be all sorts of different things: hands, faces, a flashlight, a ball. Blobs are usually identified by the contrast caused by the difference between adjoining pixels. This is quite similar to how edge detection, which you saw in Chapter 9, is done. A group of adjoining pixels that have the same relationship to another group of adjoining pixels, such as an area of darker pixels that is bordered by an area of whiter pixels, usually indicates a specific object (Figure 14-1).

Behind all the ethereal magic of gestural interfaces, particularly in using multiple tracking points, is the notion of a *blob* and of *tracking*. *Tracking* is the process of identifying one or more objects that display consistent movement within the field of the computer's vision and identifying those objects over time. A hand is an excellent and commonly used example. A finger pressed on a surface that is light reactive is another commonly used concept in many surface-based interfaces. Colored objects, particularly those with a specific design that can be recognized by a computer vision system like those used in tangible interfaces, are another common interface concept that makes use of the

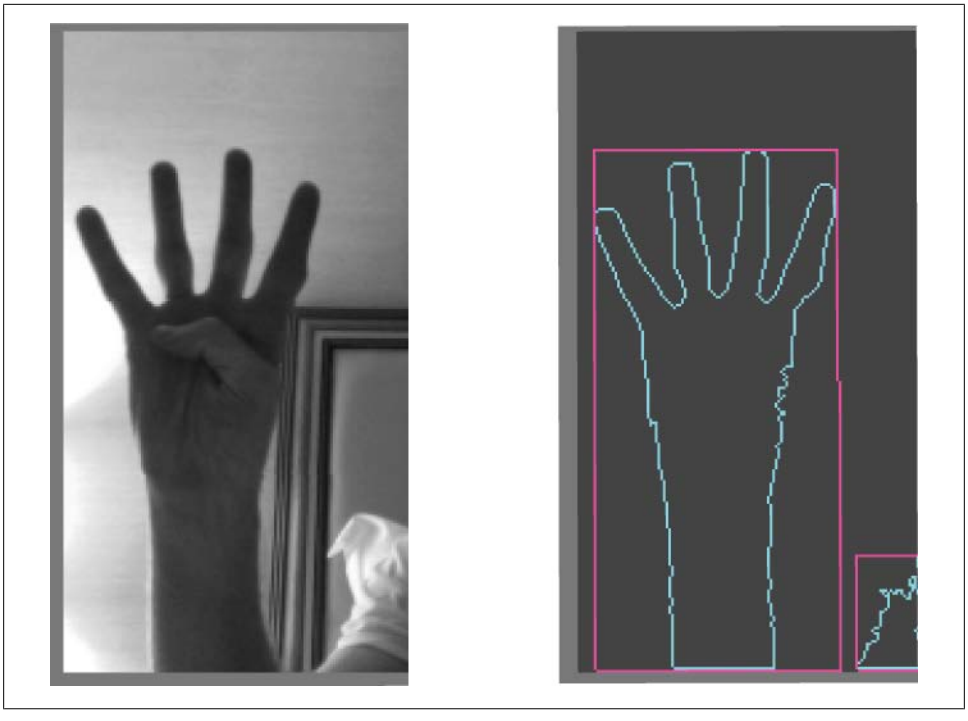


Figure 14-1. Contour detection by examining pixel darkness

tracking of blobs. Once a blob has been found, let's say a hand against a white wall, the next step is to track its movement over time and ensure that the blob's position is consistently updated to reflect the movement of the object throughout the images. This is what allows the movement of a single object in a visual field to be turned into a gesture, that is, a movement over time. This is substantially easier when you're dealing with a single object, but when dealing with multiple blobs, it becomes mathematically and computationally more complex. The tracking of blobs is one of the difficult tasks in gestural interfaces that OpenCV makes a great deal easier. Later in this chapter you'll learn how to track fingers in a controlled environment as well as learn how to use two different simple gesture recognition libraries, one for Processing and one for oF.

Starting with ofxOpenCV

ofxOpenCV is an add-on, but it's included with the "fat distribution," the full distribution containing all of the add-ons, so you won't need to download it separately. Take a look in the add-ons directory of your oF folder, and see whether the ofxOpenCV folder is there. If so, you don't need to do anything. If you don't find it, you might need to download it from *http://addons.openframeworks.cc* or grab a different version of the oF download. As with other add-ons for oF, you'll need to include the main file for the add-on in an `include` statement:

```
include "ofxOpenCv.h"
```

Once you've done that, you should be ready to go. You may want to run the sample application that comes with the ofxOpenCv add-on.

One of the most important aspects of this add-on is its contour finding capability. This is handled through the `ofxContourFinder` class that performs the blob finding and tracking task. In the *.h* file of your application, define an instance of the `ofxCvContourFinder` class:

```
ofxCvContourFinder contourFinder;
```

You'll see more about how to use this method in the following example, but for the moment, since this is one of the two key methods of `ofxCvContourFinder`, you should understand how the `findContours()` method works. Here is the signature followed by a description of the parameters:

```
virtual int  findContours( ofxCvGrayscaleImage& input, int minArea,
    int maxArea, int nConsidered, bool bFindHoles, bool bUseApproximation = true);
```

`input`
: This is an `ofxCvGrayscaleImage` reference (this is written `ofxCvGrayscaleImage&`) to a grayscale image that will be searched for blobs. Note that grayscale images only are considered. So if you're using a color image, you'll need to highlight the particular color that you're looking for beforehand. You can do this by looping through the pixels and changing the color values of any pixel with the desired color to white or black, for instance.

`minArea`
: This is the smallest potential blob size as measured in pixels that will be considered as a blob for the application.

`maxArea`
: This is the largest potential blob size as measured in pixels that will be considered as a blob for the application.

`nConsidered`
: This is the maximum number of blobs to consider. This is an important parameter to get right, because you can save yourself a lot of processing time and possibly speed up the performance of your application by pruning this number down. An interface that uses a user's fingers, for instance, needs to look only for 5 points, one for each finger. One that uses a user's hands needs to look only for two points.

`bFindHoles`
: This tells the contour finder to try to determine whether there are holes within any blob detected. This is computationally expensive but sometimes necessary.

`bUseApproximation`
: This tells the contour finder to use approximation and to set the minimum number of points needed to represent a certain blob; for instance, a straight line would be represented by only two points if `bUseApproximation` is set to `true`.

Once you've detected the blobs in an image, there are two useful properties of `ofxContourFinder` that you'll use to determine what has been found in the contour finding:

`blobs`
: The vector `ofxCvBlob` blobs returns each blob that was found in the image. These should, if all has gone well, correlate to the blobs in previous examples so that you can begin to perform gesture detection.

`nBlobs`
: This is an int that returns the number of blobs found by the contour finder.

Commonly after contour detection, you'll want to use the `nBlobs` variable to loop through the blobs contained in the blobs vector:

```
for (int i = 0; i < contourFinder.nBlobs; i++){
    contourFinder.blobs[i].draw(360,540);
}
```

This merits a look at `ofxCvBlobs`. As mentioned earlier, a blob defines an area identified by the OpenCV plug-in as being a contiguous object in the scene. Here are some of the properties of this class:

`area`
: This gives the size of the blob in pixels.

`boundingRect`
: This is an `ofRectangle` instance that can be drawn to the screen and that shows the height and width of the blob. This can be helpful to determine large regions of interest, or it can lead to some inaccurate results depending on the shape of your object. For instance, a squarish shape will be well represented by a rectangle, whereas a long thin shape with an angle in the middle will not.

`centroid`
: This is an `ofPoint` instance with its x and y positions set at the center of the boundaries of the blob.

`hole`
: This is a Boolean value that indicates whether the blob contains a whole. This is also dependent on whether the call to `findContours()` in `ofxContourFinder` has the `findHoles` parameter set to `true`.

`pts`
: This is a `vector` of `ofPoint` objects that represent the contour of the blob. This is different from the bounding rectangle. It's listing the different points around the edge of the blob. The bounding rectangle is a rectangle around the extreme points of the blob.

`nPts`
: This is an int that represents the number of points that are contained within the contour.

`void draw( float x, float y )`
: This method draws the blob to the screen with the upper-left corner located at the point specified by the x and y values.

When `contourFinder` detects a contiguous contour, it stores the information about that region as a blob. Later in this chapter you'll use those blobs to track objects over time. First, though, look at the code for reading the contours of an image, detecting blobs, and drawing them to the screen:

```
#ifndef _CTRS_APP
#define _CTRS_APP

#include "ofMain.h"
#include "ofxOpenCv.h"

class contoursApp : public ofBaseApp{

public:

    void setup();
    void update();
    void draw();

    void keyPressed  (int key);

    ofVideoGrabber vidGrabber;
    ofxCvColorImage colorImg;
```

The three `ofxCvGrayscaleImage` images are used together to determine what has changed between the current frame and the background. The `grayImage` is used to convert the color image from the video into grayscale, and the `grayBg` image stores an image of the background. These two are then compared using the `absDiff()` method of the `ofxCvGrayscaleImage` object, which creates a new image based on the differences between the current image and the image of the background and stores it in the `grayDiff` image:

```
    ofxCvGrayscaleImage grayImage;
    ofxCvGrayscaleImage grayBg;
    ofxCvGrayscaleImage grayDiff;
```

`ofxCvContourFinder` is the object that will detect the contours in the image, finding borders of objects or shifts in color and light:

```
    ofxCvContourFinder contourFinder;
    int threshold;
    bool bLearnBackground;
};
#endif
```

Here's the *.cpp* file:

```
#include "contoursApp.h"

void contoursApp::setup(){

        bLearnBackground = false;

    vidGrabber.setVerbose(true);
    vidGrabber.initGrabber(320,240);
```

Now, allocate memory for the array of pixels that the image will use to save the incoming pixel data:

```
    colorImg.allocate(320,240);
    grayImage.allocate(320,240);
    grayBg.allocate(320,240);
    grayDiff.allocate(320,240);
}

void contoursApp::update(){
    vidGrabber.grabFrame();
    //do we have a new frame?
    if (vidGrabber.isFrameNew()){
```

Now `ofxCvColorImage` has all of its pixels set from the newest frame that the video camera has captured. If you don't have a camera attached to your computer when you run this code, you can always use the `ofVideoPlayer` class to play back a video to use in `contourDetection`:

```
        colorImg.setFromPixels(vidGrabber.getPixels(), 320,240);
        grayImage = colorImg; // convert our color image to a grayscale image
        if (bLearnBackground == true) {
            grayBg = grayImage; // update the background image
            bLearnBackground = false;
        }
```

Now, find the difference between the background and incoming image to look for changes in the two images:

```
        grayDiff.absDiff(grayBg, grayImage);
```

To ensure that the `grayDiff` image doesn't contain too much noise, increase the contrast on it. You might want to make this higher or lower depending on the lighting situation in the video image:

```
        grayDiff.threshold(20);
```

Here's the call to the `findContours()` method. In this case, the smallest blobs that `contourFinder` is looking for are 10 pixels, and the largest are one tenth the size of the screen. `contourFinder` is looking for two blobs and for holes in those blobs:

```
        contourFinder.findContours(grayDiff, 10, (340*240)/4, 2, true);
    }
}
```

```
void contoursApp::draw(){
    colorImg.draw(0, 0, 32, 240);
    ofSetColor(0xffffff);
    ofRect(0, 0, 320, 240);
    ofSetColor(0x000000);
```

Now, loop through all the blobs that `contourFinder` has detected, and draw them:

```
    for (int i = 0; i < contourFinder.nBlobs; i++){
        contourFinder.blobs[i].draw(0, 0);
    }
}
```

If the user hits a key, then set the `learnBackground` parameter to `true`, meaning that the background should be captured again so the new images will be compared with the correct background. This can be good to do if the light changes in a room or if the camera moves. Otherwise, you'll find too many false matches in your image:

```
void contoursApp::keyPressed  (int key){
    bLearnBackground= true;
}
```

Now that you've seen the basics of detecting blobs, the next step is to track them over a period of time.

Tracking Blobs with ofxOpenCV

Now that you have an understanding of both how to work with ofxOpenCV in oF and what blob tracking is, you can begin to look at tracking blobs over time. Tracking a blob or a shape across a surface is, at its core, a matter of comparing all the blobs found in a scene with the blobs found in the previous frame of the scene. Just as in the earlier ofxOpenCV examples, this can be done using the `ofxContourFinder` class within the application to detect the contours of the objects. The key difference is that once a blob has been detected, then its data is stored in a vector. On the next frame, when the update method of the application is called, all the blobs are captured again and compared with the previous blobs to see which blobs from the previous frame best correspond to the blobs of the current frame. Figure 14-2 shows a single blob being tracked over several frames.

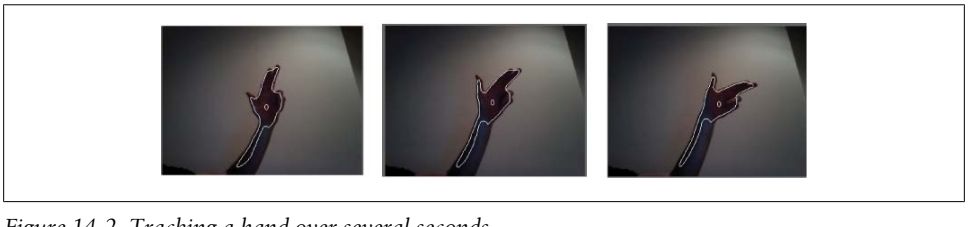


Figure 14-2. Tracking a hand over several seconds

The 0 in the center of the blob indicates the position of the blob within the vector of discovered blobs. That 0 is drawn at the *centroid*, or center point of the blob. When looking through tracking code, you'll often see references to the centroids of the blobs. This example makes the blob tracking easy because the object is in high contrast to its background and there is really only one object of interest in the frame. Generally, the more objects there are to track, the less accurate the blob tracking becomes, and the lower the contrast between objects. The more precise the control that you need over the tracking data, the more careful you'll need to be when setting up your environment to ensure that the images have the proper contrast.

Here is the general algorithm for blob tracking:

1. Compare the the incoming image against the previous image by getting the different pixels between the two images.
2. Get the threshold of the image.
3. Find the contours of the thresholded image.
4. Compare the blobs from the previous image against the blobs from the current image to determine which blobs from the current image correlate reasonably well to blobs from the previous image. Then mark them as being the same object.

The first three steps are handled by an instance of `contourFinder`, and the fourth step is handled by a new file, the `ofxCVBlobTracker`. This file is available in the code downloads for this chapter, so to conserve space, only the *.h* header file of the blob tracker is shown here:

```
#ifndef oF_CV_BLOBTRACKER_H
#define oF_CV_BLOBTRACKER_H

#include <map>
#include <vector>
#include "ofMain.h"

#include "ofxCvTrackedBlob.h"
#include "ofxCvConstants_Track.h"

class ofxCvBlobTracker {
    public:
        vector<ofxCvTrackedBlob>  blobs;
        ofxCvBlobTracker();
        void trackBlobs( const vector<ofxCvBlob>& blobs );
```

Determine the order by which the present blobs came into existence:

```
        int findOrder( int id );
```

The `getById()` method returns a reference (note the `&` symbol) to the blob in the blobs vector that has the ID passed to the method:

```
        ofxCvTrackedBlob& getById( int id );
```

The `draw()` method draws all the tracked blobs, drawing the contour of the detected blob and drawing the ID number at its center:

```
void draw( float x, float y );
```

The blobs' history, that is, all the tracked and detected blobs, is stored in the history vector, which is a vector of vectors. Each vector represents a capture of all the blobs from the screen, which are then compared to the newest blobs coming in to determine which ones most likely represent the same objects:

```
protected:
    vector<vector<ofxCvTrackedBlob> > history;
```

You can find the *.cpp* file of the `ofxCvBlobTracker` class in the downloads for this chapter.

Now, take a look at `ofBaseApp`, which uses `ofxCvBlobTracker` to find and track blobs in a live camera feed and draw them to the screen:

```
#include "ofMain.h"

#include "ofxCvBlobTracker.h"
#include "ofxCvTrackedBlob.h"
#include "ofxCvConstants_Track.h"

class trackerApp: public ofBaseApp {

    public:
        int cwidth;
        int cheight;
```

Here is the `ofVideoGrabber` instance to capture video:

```
ofVideoGrabber  vidGrabber;
```

As in the previous application, there are three images. The first stores bitmap data from the camera, the second is a grayscale image that will be thresholded to sharpen the contrast of the changed pixels, and the third stores the background for comparison against the incoming image:

```
ofxCvColorImage  colorImg;
ofxCvGrayscaleImage  grayImg;
ofxCvGrayscaleImage  bgImg;
```

`ofxCvBlobTracker` doesn't actually capture the blobs; it simply stores blobs detected by the `ofxCvContourFinder` instance and compares them for similarities. In the *.cpp* file of the application, you'll see how they are used together:

```
    ofxCvContourFinder  contourFinder;
    ofxCvBlobTracker  blobTracker;

    int threshold;
    bool bLearnBackground;
};
```

Here is the *trackerApp.cpp* file:

```
#include "trackerApp.h"

void trackerApp::setup() {
    ofSetFrameRate( 60 );
    threshold = 60;
    bLearnBackground = true;
```

Start the `videoGrabber`, and allocate memory for all the images:

```
    vidGrabber.initGrabber( 320, 240 );
    colorImg.allocate( 320, 240 );
    grayImg.allocate( 320, 240 );
    bgImg.allocate( 320, 240 );
```

Since the `blobTracker` is set up to notify a listening application when detected blobs have been moved, the tracker is set to notify this application. You do this by passing the `blobTracker` the `this` pointer so that it will call the `blobMoved()`, `blobOn()`, and `blobOff()` methods of the application:

```
    blobTracker.setListener( this );
}
```

In the application's `update()` method, a new frame is grabbed from the `videoGrabber` and converted into a grayscale image:

```
void trackerApp::update() {
    ofBackground( 100, 100, 100 );
    vidGrabber.grabFrame();
    if( vidGrabber.isFrameNew() ) {
        colorImg = vidGrabber.getPixels();
```

The `=` here copies the color pixels of the `colorImg` variable into the `ofxCvGrayscale Image` after converting them to grayscale. This uses something called *operator overloading* that allows different operators, `=`, `+`, or `-`, for instance, to have different meanings depending on the class. It's too advanced a topic for this book, but you can learn more about it in one of the more advanced C++ texts listed in Chapter 18:

```
        grayImg = colorImg;
        if( bLearnBackground ) {
            bgImg = grayImg;
            bLearnBackground = false;
        }
        grayImg.absDiff( bgImg );
        grayImg.blur( 11 );
        grayImg.threshold( threshold );
```

`contourFinder` is passed the grayscale image, and then the discovered blobs are passed to `ofxCvBlobTracker`:

```
        contourFinder.findContours( grayImg, 50, 20000, 10, false, true);
        blobTracker.trackBlobs( contourFinder.blobs );
    }
}
```

```
void trackerApp::draw() {
    ofSetColor( 0xffffff );
    colorImg.draw( 20,200 );
    grayImg.draw( 360,200 );
```

This time, you draw from `blobTracker` so that all the blobs will be drawn to the screen rather than all the detected contours as you did in the previous example:

```
    blobTracker.draw( 20,200 );
  }
```

Taking a quick foray into virtual methods

The blob tracker can also work by using callback methods, which the introductory chapters to Processing and oF discussed. A *callback* is a method that is called in response to a particular event like the `mouseMoved()` method in an oF application or the `keyPressed()` method of a Processing application. By creating a class that will call a main application when an event occurs, you can save the processing time needed to check a class again and again and simply call a method on the main application to notify it that something has happened. The downloads for this chapter contain a file called *ofxCVTrackingConstants.h* that contains the following class declaration:

```
class ofxCvBlobListener {
    public:

        virtual void blobOn( int x, int y, int id, int order ) = 0;
        virtual void blobMoved( int x, int y, int id, int order ) = 0;
        virtual void blobOff( int x, int y, int id, int order ) = 0;

};
```

This declares some virtual methods that the class that extends this interface will have to override. The reasons for this may seem a bit confusing at first, but imagine an abstract representation of a person. We know that (in our little example at least) people will be able to speak, but we don't really know what language they'll be speaking. If we were to make a person class, though, we wouldn't really want to include that kind of information, so we'd just assume that once we made some more concrete people, we'd figure out how they would speak. So, the person would look like this:

```
class Person {

public:
    virtual void speak() = 0; // here's the virtual method

};
```

This says, in essence, all people will need to speak, but you can't make a person without extending this class and creating an overridden version of this method. For example, if you wanted to make a French person, they might look like this:

```
class FrenchPerson : public Person {
public:
```

```
    void speak() { printf(" Je parle Francais. "); }
};
```

whereas an Argentine person might look like this:

```
class ArgentinePerson : public Person {
public:
    void speak() { printf(" Hablo castellano "); }
};
```

Both override the `speak()` method in their own ways, doing things that are particular to that class. The virtual method simply sets a method that all classes that extend it must override, thereby creating their own versions.

Using the velocity of blobs

Let's go back to the earlier example of the `ofxCvBlobListener` class. The `ofBaseApp` instance that your application uses can extend multiple classes. Take a look at the class declaration for the following application that declares two different classes that it is extending:

```
class testApp : public ofBaseApp, public ofxCvBlobListener {
```

This means that this class will have the three additional methods of the `ofxCvBlobListener` class, in addition to the methods of the `ofBaseApp` class. Adding those three methods allows the blob tracking engine to notify the application when a blob has moved and allows the application to react to it. To do this, the application should use the `blobMoved()` method to get information about the blob that has moved and use it. In the following example, the movement of the blob is used to change the velocity of two objects in different ways: using the movement of the blob as a force upon a small circle drawn on the screen and using the blob as an object that a ball can collide with. Chapter 9 introduced a simple sprite class called `Ball` for drawing objects that can be affected by vector forces. This example reuses that class to draw the objects affected by the blob force:

```
#ifndef _TEST_APP
#define _TEST_APP

#include "ofMain.h"
#include "Ball.h"
#include "ofxVectorMath.h"
#include "ofxCvBlobTracker.h"
#include "ofxCvMain.h"

#define APP_WIDTH 1000
#define APP_HEIGHT 800

#define vid_width 640
#define vid_height 480
```

Notice that the `ofBaseApp` extends `ofxCvBlobListener` so that it will receive events from the blob tracker when they become available:

```
class blobForceApp : public ofBaseApp, public ofxCvBlobListener {

public:

    void setup();
    void update();
    void draw();
```

Here are the two `Ball` objects that will be used by the forces in the simulation. The first instance will be affected by the hand that is detected and tracked, and the second instance will be affected by the other ball:

```
    Ball pushedBall;
    Ball physicsBall;

    int cwidth;
    int cheight;
    ofVideoGrabber  vidGrabber;
    ofxCvColorImage  colorImg;
    ofxCvGrayscaleImage  grayImg;
    ofxCvGrayscaleImage  bgImg;

    ofxCvContourFinder  contourFinder;
    ofxCvBlobTracker  blobTracker;

    int threshold;
    bool bLearnBackground;

    ofxVec3f blobCenter;
    bool checkCollision();

    void keyPressed( int key );
    void blobOn( int x, int y, int id, int order );
    void blobMoved( int x, int y, int id, int order );
    void blobOff( int x, int y, int id, int order );
```

These are the two vectors that will track the directions and amount of force that should be applied to the two `Ball` objects:

```
    ofxVec3f gravityForce;
    ofxVec3f handForce;

};

#endif
```

Example 14-1 shows the *.cpp* file for the application:

Example 14-1. blobForceApp.cpp

```
#include "blobForceApp.h"

void blobForceApp::setup() {
```

```
    ofSetFrameRate( 60 );
    threshold = 60;

    blobCenter.x = 400;
    blobCenter.y = 500;
```

Initialize all the forces and objects:

```
        gravityForce = ofxVec3f(0.0, 1.0f, 0.0);
        handForce = ofxVec3f(0.0, 0.0f, 0.0);
        pushedBall.bounce = 0.8;
        pushedBall.maximum_velocity = 2;
        physicsBall.bounce = 0.8;
        physicsBall.maximum_velocity = 2;

        vidGrabber.initGrabber( vid_width, vid_height );
        colorImg.allocate( vid_width, vid_height );
        grayImg.allocate( vid_width, vid_height );
        bgImg.allocate( vid_width, vid_height );
        blobTracker.setListener( this );
    }
```

The `update()` method grabs the frame from the video, depending on the setting either from the camera or from a video. If the frame is new, then continue processing the image:

```
    void blobForceApp::update() {

        bool bNewFrame = false;
        #ifdef _USE_LIVE_VIDEO
            vidGrabber.grabFrame();
            bNewFrame = vidGrabber.isFrameNew();
        #else
            vidPlayer.idleMovie();
            bNewFrame = vidPlayer.isFrameNew();
        #endif
        if( bNewFrame ) {
```

If you're using live video, then set the color pixels from the `videoGrabber`; otherwise, set them from the `vidPlayer`:

```
            #ifdef _USE_LIVE_VIDEO
                    colorImg.setFromPixels(vidGrabber.getPixels(), 320,240);
            #else
                    colorImg.setFromPixels(vidPlayer.getPixels(), 320,240);
            #endif
```

Set the grayscale image from the `colorImg`:

```
            grayImg = colorImg;

            if( bLearnBakground ) {
                bgImg = grayImg;
                bLearnBakground = false;
            }
            grayImg.absDiff( bgImg );
```

```
        grayImg.blur( 11 );
        grayImg.threshold( threshold );
```

In this case, the application is looking for only one blob, the largest one. It will simply reject all the blobs it finds until it finds one that is the appropriate size. Depending on the relative position of the camera and where you're tracking the blobs, you might want to make this different:

```
        contourFinder.findContours( grayImg, 200, 20000, 1, false, false );
        blobTracker.trackBlobs( contourFinder.blobs );
    }
```

Now, all the forces are added to the `Ball` instances:

```
    pushedBall.addForce(gravityForce);
    pushedBall.addForce(handForce);
    pushedBall.updateBall();
    physicsBall.addForce(gravityForce);
    physicsBall.updateBall();
```

Here, the force of the vector is reduced over time so that the results of actions are lessened over time:

```
    float tampedVal;
    float absVal;
    absVal = fabs(handForce.x);
    if(absVal > 0) {
```

If the value of the `handForce` is bigger than you want, you can shrink it reasonably by taking the square root of the value. This way, if it's a value like 9 or 12, then it will very quickly become a usable value, and if it's a value like 3, it will reduce more gradually:

```
        if(absVal > 2 ) {
            tampedVal = sqrt(absVal);
        } else {
            tampedVal = absVal - 0.1;
        }
        handForce.x = ( handForce.x > 0 ) ? tampedVal : -1 * tampedVal;
    }
    absVal = fabs(sqrt(handForce.y));
    if(absVal > 0) {
        if(absVal > 2 ) {
            tampedVal = sqrt(handForce.y);
        } else {
            tampedVal = absVal - 0.1;
        }
        handForce.y = ( handForce.y > 0 ) ? tampedVal : -1 * tampedVal;
    }
    if(    blobCenter.x == 0 && blobCenter.y == 0 ) { return; }
```

Here, the `checkCollision()` method looks to see whether the blob, in this case, the user's hand, has collided with the `Ball` instance. If it has, then determine what the result of the collision should be:

```
    if(checkCollision()) {
        physicsBall.collision(&blobTracker.blobs[0]);
```

```
    }
}

void blobForceApp::draw() {
    ofSetColor( 0xffffff );
    ofSetLineWidth(1);
    blobTracker.draw( 20,200 );
    pushedBall.drawFrame();
    physicsBall.drawFrame();
    ofSetColor( 0xffffff );
    ofSetLineWidth(3);
    ofLine(500, 400, 500+(handForce.x * 50), 400+(handForce.y * 50));
    ofCircle(500+(handForce.x * 50), 400+(handForce.y * 50), 10);
}
```

When the key is pressed, reset everything:

```
void testApp::keyPressed( int key ) {
        bLearnBackground = true;
        handForce.set(0.0, 0.0, 0.0);
        physicsBall.location.x = 400;
        physicsBall.location.y = 0;
}

void testApp::blobOn( int x, int y, int id, int order ) {}
```

Whenever a blob moves, the blob tracker calls the `blobMoved()` callback method to determine the amount of force that should be set on the `handForce` vector variable:

```
void testApp::blobMoved( int x, int y, int id, int order) {
    ofxCvTrackedBlob b = blobTracker.getById(id);
    handForce.set(0.5 * b.deltaLoc.x, 0.5 * b.deltaLoc.y, 0.0);
    blobCenter.x = ( APP_WIDTH / vid_width ) * b.centroid.x;
    blobCenter.y = ( APP_HEIGHT / vid_height) * b.centroid.y;
}

void testApp::blobOff( int x, int y, int id, int order ) {}

bool testApp::checkCollision() {
    //difference vector of the 2 circles' positions
    ofxVec3f cDiff = physicsBall.location - blobCenter;
    float c = cDiff.dot( cDiff ) - 10000;
    if( c < 0 ) {
        return true;
    }
    return false;
}
```

This example is a very simple example of how to generate forces from computer vision and how to add forces to drawings. Of course, you can improve this example in a great number of ways or extend these basic techniques to create different kinds of reactivity, or different kinds of animations, from simple particle fields to games to water simulations.

Using OpenCV in Processing

So far in this chapter, you've looked only at OpenCV in oF applications, but as mentioned earlier, there is an OpenCV library for Processing as well. The OpenCV library is made accessible to your Processing application through a class called `OpenCV`. First download the library from the Processing website, and then move the library to the *libraries* folder of your Processing installation. Import the libraries by calling the following `import` statement:

```
import hypermedia.video.*;
```

Create an instance of this class, and pass this reference to your application to the constructor; you're ready to go:

```
OpenCV opencv;
opencv = new OpenCV(this);
```

To begin capturing video, call the `capture()` method, passing in the width and height you want to capture. This automatically reads from the first video camera attached to your computer.

```
int IMG_WIDTH = 400;
int IMG_HEIGHT = 300;
opencv.capture(IMG_WIDTH, IMG_HEIGHT);
```

To capture a frame of video, in each `draw()` method of your application you'll want to call the `read()` method of the OpenCV library:

```
opencv.read();
```

Here is a simple example that captures, flips, blurs, inverts, and then redisplays the captured image. The resulting images are shown in Figure 14-3.

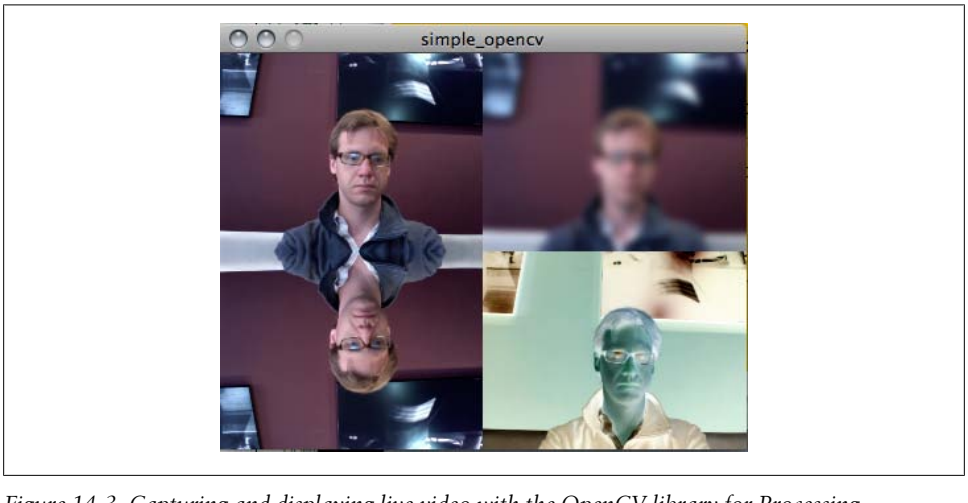


Figure 14-3. Capturing and displaying live video with the OpenCV library for Processing

```
import hypermedia.video.*;

// OpenCV instance
OpenCV opencv;
// blur value
int blurAmount = 17;

// image dimensions
int IMG_WIDTH  = 200;
int IMG_HEIGHT = 150;

int COLOR_SPACE = OpenCV.RGB;
//final int COLOR_SPACE = OpenCV.GRAY;

void setup() {

    size( IMG_WIDTH*2, IMG_HEIGHT*2 );

    opencv = new OpenCV(this);
    opencv.capture(IMG_WIDTH, IMG_HEIGHT);

}

void draw() {
    // grab image
    opencv.read();
```

Here, create a blur by calling the `blur()` method and passing the type of blur. The options as of this writing are `BLUR` for a simple blur, `MEDIAN`, `GAUSSIAN`, or `BLUR_NO_SCALE`. Experiment with the different options to see how they differ because they're rather difficult to describe in print:

```
    opencv.blur( OpenCV.GAUSSIAN, blurAmount );
```

Now, draw the image to the screen:

```
    image( opencv.image(), IMG_WIDTH, 0 );
```

The `restore()` method is called to reload the original image data. This is quite important because without calling this method, changes are applied sequentially. For instance, if you comment out the following call to `restore()`, the inverted image will have the blur applied to it. With the call to restore, the original image captured by the camera is used again:

```
    opencv.restore( COLOR_SPACE );
    opencv.flip(OpenCV.FLIP_VERTICAL);
    image( opencv.image(), 0, IMG_HEIGHT);

    opencv.restore( COLOR_SPACE );
    opencv.invert();
    image( opencv.image(), IMG_WIDTH, IMG_HEIGHT);
```

Another way to draw the original image is to call the `image()` method of the OpenCV library and pass `OpenCV.SOURCE` to it. This returns the original pixels captured from the

camera. In the following line, these are passed to the `image()` method of the Processing application to be drawn:

```
        image( opencv.image(OpenCV.SOURCE), 0, 0 );
    }
```

Here, update the amount of blur when the mouse is dragged:

```
    void mouseDragged() {
        blurAmount = (int) map(mouseX,0,width,0,255);
    }
```

When the application is stopped, make sure to call the `stop()` method of the `opencv` instance:

```
    public void stop() {
        opencv.stop();
        super.stop();
    }
```

Now, let's look at blob tracking using the OpenCV library. Like the ofxOpenCV add-on, the Processing version of this library includes a `Blob` class that used to define the area of a tracked blob. The properties of this class should look familiar to you if you read the previous section:

`float area`
: This is the area of the blob in pixels.

`float length`
: This is the length of the perimeter in pixels.

`Point centroid`
: This is the binary center of the blob.

`Rectangle rectangle`
: This is a Processing `Rectangle` instance that contains the blob, defined by its boundaries.

`Point[] points`
: This is the list of points defining the shape of the blob.

`boolean isHole`
: This returns `true` if the blob is determined to be a hole inside of another blob.

To capture the blobs, simply call the `opencv.blobs()` method, passing the following parameters:

```
    blobs( int minArea, int maxArea, int maxBlobs, boolean findHoles )
```

In the following example, the image from the camera will be captured and compared with a background image. One difference between the oF add-on and the Processing library is that in Processing the `remember()` method is used to save the current image to memory. Once the image is saved, it can then be accessed by using the `opencv.image()` method and passing `OpenCV.MEMORY`, as shown here:

```
opencv.image(OpenCV.MEMORY)
```

The difference between the image in memory and the current image from the camera will be used to find blobs in the image:

```
import hypermedia.video.*;
OpenCV opencv;
int w = 320;
int h = 240;
int threshold = 80;

boolean find=true;

void setup() {

    size( w*2+30, h*2+30 );
    opencv = new OpenCV( this );
    opencv.capture(w,h);
}

void draw() {
    background(0);
```

Read the latest image from the camera using the `read()` method. This saves the image into memory so it can be accessed later:

```
    opencv.read();
    image( opencv.image(), 10, 10 );                // RGB image
    image( opencv.image(OpenCV.GRAY), 20+w, 10 );   // GRAY image
```

Each time the image is altered, the image in memory is updated, allowing you to stack alterations to the image. The call to `opencv.image(OpenCV.MEMORY)` accesses the image captured when the `opencv.remember()` method is called:

```
    image( opencv.image(OpenCV.MEMORY), 10, 20+h ); // image in memory
```

Like in the previous oF examples, the difference of the background image and the incoming image is used to determine what pixels have changed between the two images. This is then drawn to the screen using the `image()` method:

```
    opencv.absDiff();
    opencv.threshold(threshold);
    image( opencv.image(OpenCV.GRAY), 20+w, 20+h ); // absolute difference image
```

Here, the blobs are detected in the image:

```
    Blob[] blobs = opencv.blobs( 100, w*h/3, 20, true );
    noFill();
    pushMatrix();
    translate(20+w,20+h);
```

Now, draw all of the blobs:

```
    for( int i=0; i<blobs.length; i++ ) {
```

The `rectangle` property of the `Blob` class can be used to draw a rectangle using the `rect()` method:

```
Rectangle bounding = blobs[i].rectangle;
noFill();
rect( bounding.x, bounding.y, bounding.width, bounding.height );
```

Now capture the rest of the properties of the blob:

```
float area = blobs[i].area;
float circumference = blobs[i].length;
Point centroid = blobs[i].centroid;
Point[] points = blobs[i].points;
// centroid
stroke(0,0,255);
```

Draw a small cross at the center of the blob:

```
line( centroid.x-5, centroid.y, centroid.x+5, centroid.y );
line( centroid.x, centroid.y-5, centroid.x, centroid.y+5 );
fill(255,0,255,64);
stroke(255,0,255);
```

Now, draw a simple shape to describe all the points of the blob:

```
            if ( points.length>0 ) {
                beginShape();
                for( int j=0; j<points.length; j++ ) {
                    vertex( points[j].x, points[j].y );
                }
                endShape(CLOSE);
            }
        }
        popMatrix();
    }
```

Listen for the spacebar being pressed to record a new background image:

```
void keyPressed() {
    if ( key==' ' ) opencv.remember();
}
```

Drag the mouse to change the threshold:

```
void mouseDragged() {
    threshold = int( map(mouseX,0,width,0,255) );
}

public void stop() {
    opencv.stop();
    super.stop();
}
```

Figure 14-4 shows this application running.

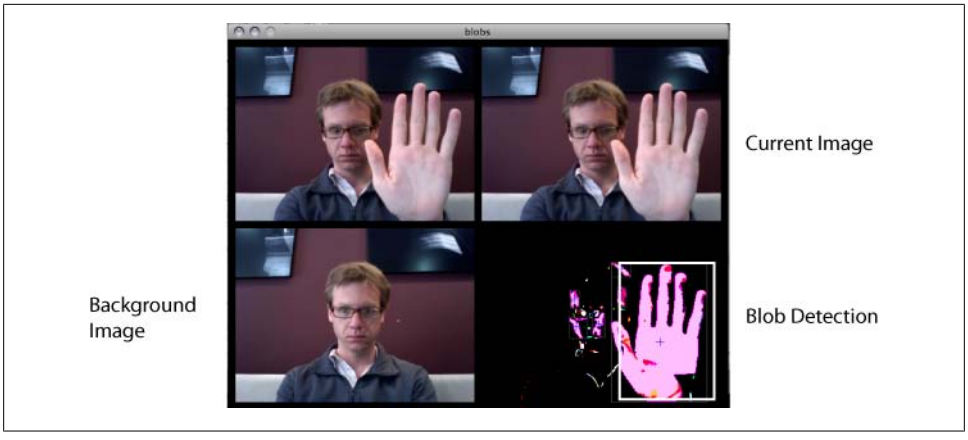


Figure 14-4. Blob tracking in Processing with OpenCV

The Processing application is slightly more restrictive than the oF version because of how the core Java classes of the Processing version interact with the OpenCV core, but much of the basic functionality is available for you to use. Working with OpenCV in Processing is an excellent way to get started with computer vision and tracking.

Exploring Further in OpenCV

There are many other topics that you can explore in OpenCV that there isn't space in this chapter to explain and explore properly. These are all used frequently and you can find more information on using them in Gary Bradskis *OpenCV*, in the OpenCV documentation, or on the oF forums at *openframeworks.cc/forums*.

Line fitting
: Line fitting is the process of examining an image to see whether it fits a certain line. A series of points can be "fitted" by examining the series and finding an equation that creates a line that fits those points. This is rather complex mathematically, but it has some very practical applications. For instance, using the change in the rate of change around a contour, you can determine whether a blob has a finger contained within it. Several examples are available on the oF forums that use this technique to find and track hands or fingers in a video feed.

Convex hulls
: A convex hull is an object detection technique that attempts to determine the boundaries of an object given a set of points using an algorithm to determine the shortest distance between different points. Figure 14-5 shows an example.

  Convex hulls are very useful for creating collision detection, For instance, you can use a convex hull to determine whether a virtual object has collided with an object detected in an image or a video. You can find examples of creating and using a convex hull in OpenCV on the oF forums.

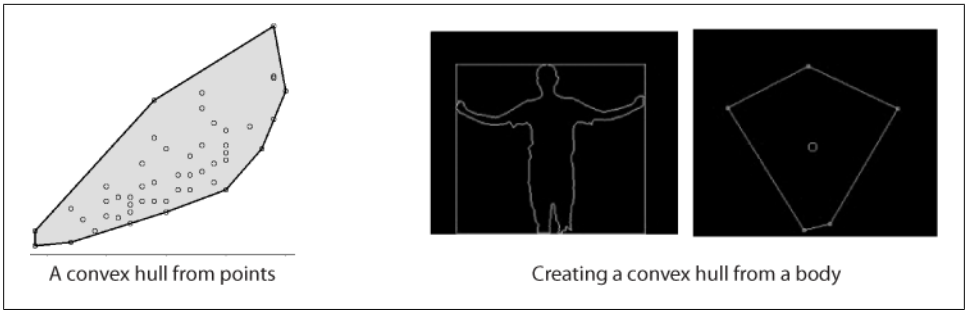


Figure 14-5. Creating a convex hull from points and from the image of a body

Optical flow
: Optical flow or optic flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion. It works by reading the apparent motion of brightness patterns in the image. Generally, optical flow corresponds to the motion field, but not always, as shown in Figure 14-6.

Usually apparent motion matches actual motion, which can be helpful for detecting motion where you don't need to track blobs but simply want to detect movement. For instance, many reactive installations merely want to react to the general movement in front of the camera, not track blobs.

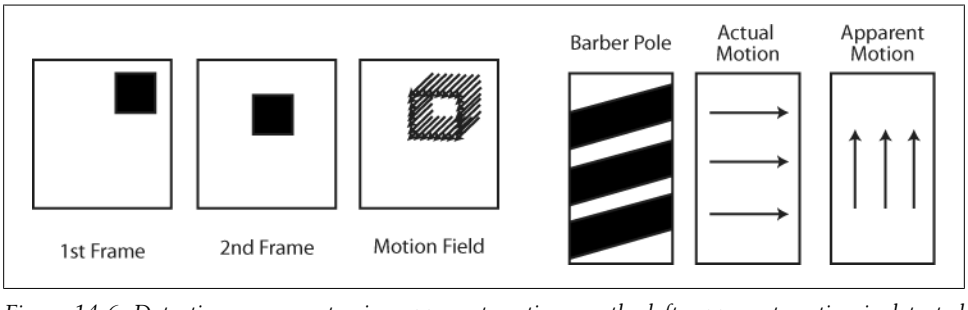


Figure 14-6. Detecting movement using apparent motion; on the left, apparent motion is detected correctly, while on the right, you can see how apparent motion might not match actual motion

Detecting Gestures

You might have used a pen-driven interface at some point, and if you have, then you're probably familiar with how it detects what character you're trying to write when you input text. The characters are similar to the ones normally used but are simplified to reduce ambiguity for the program processing them. You might also be familiar with some other more complex gestures from other devices. Those sorts of gestures are best done with a single point of input, a pen, or a single finger, but they can be done with multiple points of input, though this is substantially more difficult. Often, gestures aren't the best and easiest ways to get user input. In the case of writing, it's far less

efficient than typing on a keyboard. In some cases, though, they're extremely effective and efficient ways for a user to input a simple command like clearing a grouping of photos, selecting an object or an area of an object, and doing some kinds of sign language recognition. The simplest way to detect gestures is by using a controller to act as an extension of the body so that the motion can be captured by software. Both mouse gestures and the Wii Remote work very well as gesture controllers. In this chapter, we'll look at using the mouse for gesture detection so that you can easily examine the fundamentals of gesture detection.

Another object that works well for 2D gesture detection is a camera, as long as the environment is appropriate for using computer vision. An appropriate environment is bright and clear enough and has sufficient differentiation between the object of interest and the background. The object of interest can be a pen, a hand, or the user themselves.

To understand how gesture recognition is done, think about the gesture itself: a movement in a particular direction and with a certain pattern. Thinking about this in the way that a computational system could understand it, it would be a map of x, y, and possibly z points across a surface. This map is called a *gesture map*. These gesture maps are made by recording the vector of a movement to compare to all the gesture maps that the engine recognizes. Commonly, gesture recognition engines contain different algorithms to compare sets of movements to the gestures that they contain. One of the challenges of working with gestures is determining whether a gesture is complete. In the following examples, since the mouse is used as the gesturing instrument, the mouse button indicates the start and end of the gesture. In the case of a camera-based gesture, entering and leaving an area can indicate the beginning and end of a gesture, as can simply holding the tracked object still for a predetermined length of time.

Using ezGestures in Processing

One way to use gestures in a Processing application is to use ezGestures, a gesture recognition library. It analyzes mouse or Wiimote movements that you make while dragging the mouse or while holding the A button of the Wiimote down. Then it compares the motion against a gesture map that you've created for all the gestures in which your application should be interested. It's an excellent way to experiment and prototype gestural applications. The library understands four directions: `UP`, `DOWN`, `LEFT`, and `RIGHT`. Using these simple directions, the application can recognize, for instance, the difference between a circle and an *L* shape or the difference between clockwise or counterclockwise gestures. It can't, however, tell the difference between a square and a circle or recognize diagonal gestures, and it doesn't pay attention to how large an area is being covered, among other things.

There are two types of `GestureListener`s used in the ezGestures library. In the first type, *concurrent* listeners try to match the gesture to the pattern while the gesture is still being performed. This means that a gesture can be recognized and acted upon before the button is released. The second type starts with *Post*; for instance, `PostVShakeListener`

tries to match the gesture to the pattern after the gesture is complete, which means after the button is released. This is how the following example works.

One thing to understand about the gesture detection library is that it uses a *regular expression* to determine what gesture is being drawn. A regular expression is really just a way of making a pattern of letters. While they can be extremely complex and difficult to read, they can be powerful tools. You don't need to understand too much about how they work to use the gesture detection library. You'll see a few strange symbols at the beginning of the string that is passed to the `PostGestureListener` constructor method; for instance, a circle is described as `LDRU`, or left, down, right, up. To make sure that any and all circles are found in the messages that are sent to the `PostGestureListener` method, you'd use the caret (`^`) to indicate that anything can come before the circle gesture and the dollar sign (`$`), which is the last thing in the string passed to the `PostGestureListener`. Here is an example of creating a gesture listener that listens for a circle gesture:

```
PostGestureListener pgl = new PostGestureListener(this, brain, "^(LDRU)$");
```

For each gesture that you want to listen for, you'll need to create a new `PostGestureListener` object that will have the string that represents the gesture passed to it in the constructor for that object:

```
import net.silentlycrashing.gestures.*;

GestureAnalyzer brain;
PostGestureListener boxListener;


void setup() {
    // initialize the gesture listeners
    size(500, 500);

    brain = new MouseGestureAnalyzer(this);
    brain.setVerbose(true);
    boxListener = new PostGestureListener(this, brain, "^(RDLU)$");
    boxListener.registerOnAction("drawBox", this);
}

void draw() {
    line(pmouseX, pmouseY, mouseX, mouseY);
}

void drawBox() {
    rect(mouseX, mouseY, 200, 200);
}
```

The previous example used `PostGestureListener`, but there are four other classes that use concurrent gesture recognition:

`ConcurrentHShakeListener`
: Listens for a horizontal shake while the movement is being made

`ConcurrentVShakeListener`
: Listens for a vertical shake while the movement is being made

`PostHShakeListener`
: Listens for a horizontal shake while the movement is being made

`PostVShakeListener`
: Listens for a vertical shake while the movement is being made

In the following very simple example, the listeners will listen concurrently; that is, they won't wait for the gesture to be finished. Instead, they'll simply call their action method whenever the gesture being listened for is heard. `ConcurrentHShakeListener` just listens for a horizontal sweep back and forth, or in ezGestures terms `RLRL` or `LRLR`. In this simple application, drawing the square gesture creates a new 3D box using the `box()` method, and the horizontal shake clears all the boxes:

```
import net.silentlycrashing.gestures.*;
import net.silentlycrashing.gestures.preset.*;

GestureAnalyzer brain;
ConcurrentGestureListener boxListener;
ConcurrentHShakeListener shakeListener;

int numBoxes = 0;
SimplePoint[] boxes;

void setup() {
    // initialize the gesture listeners
    size(500, 500, P3D);
    lights();

   boxes = new SimplePoint[100];
```

As before, the `MouseGestureAnalyzer` is created and passed a reference to the application itself:

```
    brain = new MouseGestureAnalyzer(this);
    brain.setVerbose(true);
```

`ConcurrentGestureListener` is initialized and given the box string as its gesture to listen for:

```
    boxListener = new ConcurrentGestureListener(this, brain, "(RDLU)$");
    boxListener.registerOnAction("createBox", this);
```

`ConcurrentHShakeListener` is created and passed a reference to the `MouseGestureAnalyzer` instance. Since the gesture that it's listening for, the horizontal shake, is already programmed, there's no need to pass it another string to use as its gesture:

```
    shakeListener = new ConcurrentHShakeListener(this, brain);
    shakeListener.registerOnAction("shakeHeard", this);
}

void draw() {
```

```
// line(pmouseX, pmouseY, mouseX, mouseY);
    background(122);
    int i;
```

Using the `camera()` method as in Chapter 13, the camera is positioned based on the user's mouse position. This could, of course, be the relative position of a Wiimote or another kind of accelerometer connected to the computer via an Arduino:

```
    camera(0.0, mouseY, 0, // eyeX, eyeY, eyeZ
         width/2, height/2, 0.0, // centerX, centerY, centerZ
         0.0, 1.0, 0.0); // upX, upY, upZ
```

Here, the `translate()` method sets the location where the box will be drawn, and the `s` property of the `SimplePoint` is used to determine the size of the box:

```
    for( i = 0; i<numBoxes; i++) {
       pushMatrix();
           translate(width/2 - 100, height/2 - 100, 0);
           translate(boxes[i].x, boxes[i].y, boxes[i].z);
           box(boxes[i].s);
       popMatrix();
    }
}

void createBox() {
    SimplePoint p = new SimplePoint();
    p.x = random(100);
    p.y = random(100);
    p.z = random(50);
    p.s = random(40);
    boxes[numBoxes] = p;
    numBoxes++;
}
```

When the horizontal shake gesture is heard, all the boxes will be cleared from the screen:

```
void shakeHeard() {
    println(" shaken ");
    numBoxes = 0;
}
```

This is a simple class to store the data about where the box should be drawn:

```
class SimplePoint {
    float x;
    float y;
    float s;
    float z;
}
```

One of the great advantages of the ezGestures library is that it works well with the Wiimote remote control. To help you work with the Wiimote, the ezGestures library has several different listeners that listen for buttons and actions specific to the Wiimote.

Using Gestures in oF

ofxSimpleGesture is an add-on for openFrameworks that lets you do simple gesture recognition using either a mouse or a video camera. To start, download the add-on from *http://addons.openframeworks.cc*. It's loosely based on an algorithm developed by Didier Brun and was originally written in ActionScript for Flash Player. You can add your own custom shapes to it or use a predefined alphabet. The ofxSimpleGesture plug-in has a `createStandardAlphabet()` method that creates the base alphabet using modified shapes that can easily be recognized by the controller.

Figure 14-7 shows the shapes of each letter. The *M*, for instance, is broken into four separate gestures. As the user makes the shapes, the ofxSimpleGesture plug-in stores a record of the gestures and then attempts to determine what shape the user might be trying to make. The drawing that the user makes may be slightly different than the model for that gesture. The ofxSimpleGesture plug-in will take a best guess as to what the intended gesture is.

As you can see from the letter shapes shown on the left, the strokes to create the letters are simplified in a few cases to make it easier for the application to read the user's gestures. You certainly don't need to use these preprogrammed alphabetic gestures; you can easily create your own.

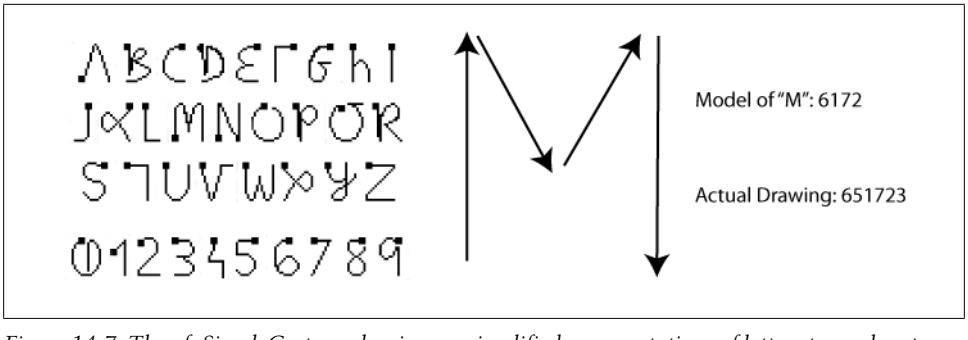


Figure 14-7. The ofxSimpleGesture plug-in uses simplified representations of letters to read gestures.

The ofxSimpleGesture requires that the application extend the `IGestureListener` class, in addition to the `ofBaseApp` class. This means that the application has a new method, `onGesture()`, that can be called by the ofxSimpleGesture plug-in when a gesture has been detected.

The declaration for an application using the ofxSimpleGesture might look like this:

```
#ifndef _TEST_APP
#define _TEST_APP
#include "ofMain.h"
#include "ofxGestures.h"

class gestureApp : public ofBaseApp, public IGestureListener {
```

```
    public:

        void setup();
        void update();
        void draw();

        void mouseDragged(int x, int y, int button);
        void mousePressed(int x, int y, int button);
        void mouseReleased();
```

Here's the `onGesture()` method that the `ofxGestureRecognizer` instance uses to tell the application that it has detected a gesture:

```
        void onGesture(const string& name);
```

Here's the declaration of the `ofxGestureRecognizer` instance:

```
        ofxGestureRecognizer ofxg;
        string gesture;

    };

#endif
```

In the *.cpp* file of the `gestureApp`, the `ofxGestureRecognizer` instance calls the `setListener()` method to set the callback methods of the application:

```
#include "gestureApp.h"

void gestureApp::setup(){
    ofBackground(255,255,255);
    ofxg.setListener(this);
```

The `createStandardAlphabet()` method creates all the letters in the standard alphabet, as shown previously in Figure 14-7:

```
    ofxg.createStandardAlphabet();
    gesture = "";
}

void gestureApp::update(){
    // nothing here
}

void gestureApp::draw(){
    ofBackground(255,255,255);
    ofSetColor(0x000000);
    ofDrawBitmapString(gesture, 100, 100);
}

void gestureApp::mouseDragged(int x, int y, int button){
    ofxg.addMousePosition(x, y);
}
```

Start the capture by calling the `startCapture()` method of the `ofxSimpleGesture` instance. This begins logging all the user's actions to determine the gesture:

```
void testApp::mousePressed(int x, int y, int button){
    ofxg.startCapture(x, y);
}
```

The gesture recording stops when the mouse is released:

```
void gestureApp::mouseReleased(){
    ofxg.endCapture();
}
```

Next is the callback method for the `ofxGestureRecognizer` that indicates to the application that the gesture has been found:

```
void gestureApp::onGesture(const string& name) {
    gesture = "Your gesture was ";
    gesture += name;
}
```

Implementing Face Recognition

Face recognition doesn't mean recognizing a person's face; it means recognizing things that look like a face in an image. It's accomplished by "training" a program to tell it what to look for in an image. The process of training applications to learn things is called *machine learning* and is a very interesting topic. In this section, you'll learn about a specific machine learning technique. The most commonly used approach for face detection and recognition in OpenCV is based on a technique called the *Haar Classifier*. The Haar Classifier is a data file generated from a training process where an application is "taught" how to recognize something in different contexts. This can be things like recognizing whether a certain sound is a word being spoken by a user, whether a gesture is a certain shape, or, in this case, whether a pattern of pixels is a face. If you're interested in learning more about Haar Classifiers and how machine languages work in OpenCV, once again refer to Gary Bradski's and Adrian Kaehler's book, *Learning OpenCV* (O'Reilly).

Creating a Haar Classifier is done by generating multiple images and setting the regions of interest in the image. This is usually called *training*, and Figure 14-8 shows some example images, generously donated by Todd Vanderlin.

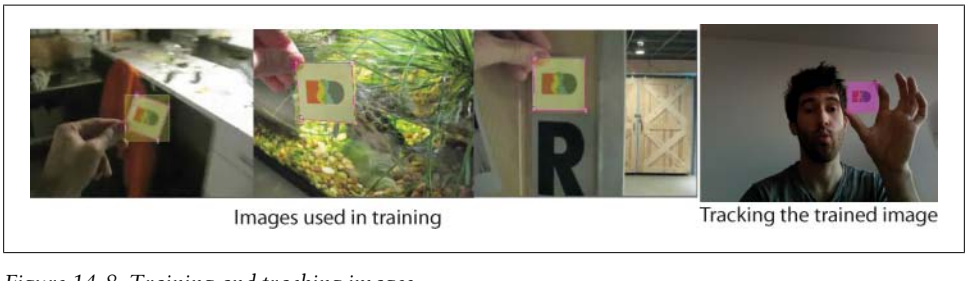


Figure 14-8. Training and tracking images

Notice how the three images to the left all show different variations of images containing the same object. After correct training, an OpenCV application can properly track the object in a video feed. Training and creating a Haar Classifier is beyond the scope of this book, but it's helpful to understand roughly how the training works. In Figure 14-8, the training is for the small card image. The characteristics of that card, meaning the size, shapes, and shading of it, are encoded into an XML file that the OpenCV engine can read and compare to the contours of images that it is sent. In training faces, hundreds or thousands of images of faces are used to create a data model of what a face looks like, and then similar images of objects other than faces are used to create a data model of what a face does not look like.

One of the things that OpenCV generously provided to the open source community is a few sample trained XML files that represent a Haar Classifier that can be used in a few general situations. For instance, in the following example, the *haarcascade_frontalface_alt_tree.xml* file contains data about the characteristics of a face looking at the camera. Other files included in the OpenCV training files help detect faces seen in profile, full bodies, upper torsos, and lower torsos. In the downloads for this chapter, you'll find the *haarcascade_frontalface_alt_tree.xml* file. If you open the XML file, you'll see XML that defines information about the shapes that could potentially match the objects being defined. This XML file defines regions that contain certain attributes that are common to faces. If you're curious what it looks like, open the file up and take a look.

You'll want to make sure that you add the Haar training file to the data folder of your application and the *ofxCvHaarFinder.h* and *ofxCvHaarFinder.cpp* files to your application folder. The following application simply looks for faces in the images coming from the camera and draws a red rectangle wherever it detects a face using the `ofxCvHaarFinder` object:

```
#ifndef _HAAR_APP
#define _HAAR_APP

#include "ofxCvMain.h"
#include "ofMain.h"
```

Since the HaarFinder isn't part of the ofxOpenCV add-on, you'll need to include it separately in your application by including the *.h* file:

```
#include "ofxCvHaarFinder.h"

class haarApp : public ofBaseApp{

    public:

        void setup();
        void update();
        void draw();

        void keyPressed  (int key);
        ofVideoGrabber vidGrabber;
```

```
        ofxCvColorImage colorImg;

        ofxCvGrayscaleImage grayImage;
        ofxCvGrayscaleImage grayBg;
        ofxCvGrayscaleImage grayDiff;
```

As with the blob tracking example earlier in this chapter, the images are read from the `ofVideoGrabber` instance and then converted to grayscale images. This is then passed to `ofxCvHaarFinder`, which contains an instance of the `CvHaarClassifierCascade` object from the OpenCV library:

```
        ofxCvHaarFinder haarFinder;

        int threshold;
        bool bLearnBackground;
};

#endif
```

Here's the *haarApp.cpp* file:

```
#include "haarApp.h"

void haarApp::setup(){
    vidGrabber.setVerbose(true);
    vidGrabber.initGrabber(320,240);

    colorImg.allocate(320,240);
    grayImage.allocate(320,240);
    bLearnBackground = true;
    threshold = 80;
```

Now, the XML file is loaded into `ofxCvHaarFinder` so that it can use that data to analyze the images passed into it:

```
    haarFinder.setup("haarcascade_frontalface_default.xml");
}

void haarApp::update(){
    ofBackground(100,100,100);

    bool bNewFrame = false;
    vidGrabber.grabFrame();
    bNewFrame = vidGrabber.isFrameNew();
    if (bNewFrame){
            colorImg.setFromPixels(vidGrabber.getPixels(), 320,240);
            grayImage = colorImg;
        if (bLearnBackground == true){
            grayBg = grayImage;
            bLearnBackground = false;
        }
```

Now the image is passed to the `ofxHaarFinder` instance that will try to find any objects that match the characteristics defined in the XML file that was loaded:

```
        haarFinder.findHaarObjects(grayImage, 10, 99999999, 10);
    }
}

void haarApp::draw(){
```

The grayscale image is drawn, and then rectangles are drawn where `ofxHaarFinder` has detected faces:

```
    ofSetColor(0xffffff);
    grayImage.draw(20, 20);
```

`ofxHaarFinder` contains the number of faces that it has discovered in the `blobs` vector. You can use the `size()` method of the `blobs` vector to create a loop to draw all the detected faces:

```
    int numFaces = haarFinder.blobs.size();
    glPushMatrix();
    glTranslatef(20, 20, 0);
```

Loop through all the faces, and draw their bound rectangles:

```
    for(int i = 0; i < numFaces; i++){
        float x = haarFinder.blobs[i].boundingRect.x;
        float y = haarFinder.blobs[i].boundingRect.y;
        float w = haarFinder.blobs[i].boundingRect.width;
        float h = haarFinder.blobs[i].boundingRect.height;
        float cx = haarFinder.blobs[i].centroid.x;
        float cy = haarFinder.blobs[i].centroid.y;
        ofSetColor(0xFF0000);
        ofRect(x, y, w, h);
        ofSetColor(0xFFFFFF);
        ofDrawBitmapString("face "+ofToString(i), cx, cy);
    }
    glPopMatrix();
}
```

The `keyPressed()` method sets the background and will be stored again so that incoming frames can be compared to the backdrop of the image:

```
void haarApp::keyPressed (int key){
    bLearnBackground = true;
}
```

As shown in Todd Vanderlin's example, anything can be trained for recognition, but since the OpenCV download comes with pretrained Haar files, it makes sense to demonstrate how to use them. Face tracking can be a very interesting way of providing responsiveness to an application. Since humans naturally tend to look at one another's faces as an important source of information, the interactive possibilities of having an application that does the same are worth exploring. Golan Levin's Opto-Isolator is an excellent example of an interactive art piece that uses face tracking, following the eyes of the viewer. Beyond face tracking, you can apply the same techniques to detect eyes, mouths, or noses, using the face recognition as a first pass and then determining within the face what shapes are most likely such as the eyes, mouth, and so on.

Exploring Touch Devices with oF

In 2007, the touch device was one of the most revolutionary concepts in interface design. Two years later, it has become remarkably commonplace for phones, computer screens, and interactive walls. Touchscreens have gone from being simple interfaces that allowed simple dragging to being complex user interface tools where users can enter text, draw, and perform many tasks with both hands that could previously be done only with a single mouse. There isn't sufficient space in this book to cover the design and implementation of any of the touch devices, tools, or libraries that have been developed to help you in creating touch applications, but, as with so many things in this book, we'll take a look at some of the projects that use similar techniques or tools as discussed in this chapter.

TouchKit

TouchKit, shown in Figure 14-9, is a modular multitouch development kit with the aim to make multitouch readily available in an open source fashion. TouchKit is comprised of software and a hardware component. The TouchKit team provides the source files and welcomes you to use, study, and appropriate the code and schematics for your work or projects. TouchKit is a plug-and-play solution for simple projects and an easily extendable base for experimental and cutting-edge endeavors.

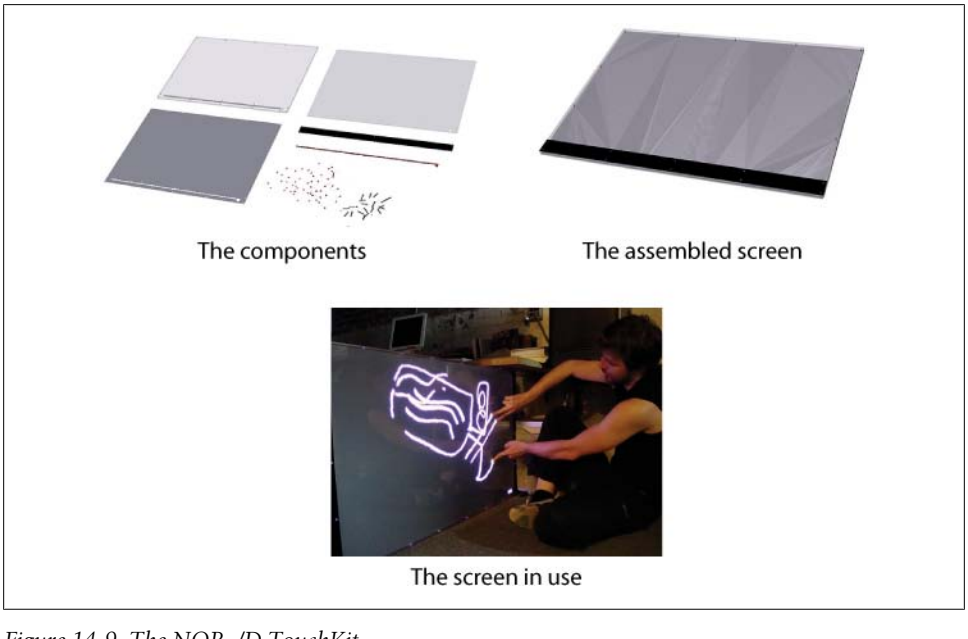


Figure 14-9. The NOR_/D TouchKit

You can find more information on NOR_/D and its projects at *touchkit.nortd.com/*. TouchKit will also be explored in greater detail in Chapter 16.

Tuio

Tuio is a protocol for tabletop tangible user interfaces that helps provide a common way to describe events. It's used in a few different existing table interfaces like the reacTable, developed in Barcelona, and the tDesk, developed in Germany. It defines information about objects on a table surface, such as finger and hand gestures performed by the user or the movement of controller objects with fiducial markers like those shown in Figure 14-10.

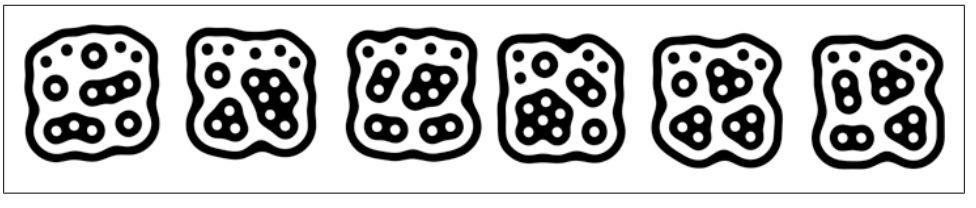


Figure 14-10. Fiducial markers that can be tracked across a table and used as controls by a user

The Tuio protocol is implemented using the OpenSound Control (OSC) that was discussed in Chapter 12, and is usable on any platform supporting this protocol. There is an ofxTuio add-on available for oF that lets a table interact with an oF application. The Tuio library can be used with the Touchlib library for Processing or oF as well. You might want to look into some of the development efforts around the Tuio implementation if you're interested in developing your own touchscreen devices.

Touchlib

Touchlib is another library that can be used to create multitouch interaction surfaces. It tracks blobs of infrared light and sends your information about these blobs as events. These events are named things like finger down, finger moved, and finger released. It includes a configuration application and a few demos to get you started. It will interface with most types of webcams and video capture devices. It currently works only under Windows, but it is being ported over to other platforms.

reacTIVision

reacTIVision is another open source computer vision framework. It can be used for tracking fiducial markers and for multitouch finger tracking in table-based tangible user interfaces (TUIs) and multitouch interactive surfaces. It is being developed by Martin Kaltenbrunner and Ross Bencina at the Music Technology Group at the Universitat Pompeu Fabra in Barcelona, Spain, as part of the reacTable project. reacTable is a novel electronic music instrument based using a tabletop multitouch tangible user interface.

Martin Kaltenbrunner has created a library for working with reacTIVision that can be used for creating touchscreen applications using Processing.

What's Next

If you're really interested in using OpenCV, a great book to check out is *Learning OpenCV* by Gary Bradski and Adrian Kaehler (O'Reilly), one of the creators of OpenCV. It'll teach you a great deal about how to use OpenCV to its fullest potential, which is something that this chapter has barely begun to do. The OpenCV manual, while not light reading, is available online as well as a free PDF download.

A number of online resources are available for people interested in working with computer vision and gesture tracking online; the oF forums are a particularly active place to get advice and ideas. The OpenCV project has its own mailing list and forum that might be an excellent resource for any one looking to troubleshoot a problem or come up with an idea for how a certain image could be processed. A great number of courses that use the OpenCV library or other image and motion processing libraries have materials online that you'll be able to peruse.

The ARToolkit, developed at the University in Washington, is a library for developing augmented reality applications. *Augmented reality* (AR) is the process of placing computer graphics images onto live feeds from a camera and is a very interesting way of approaching creating an interactive application, since a representation of the world can be interwoven with computer synthesized images (Figure 14-11).

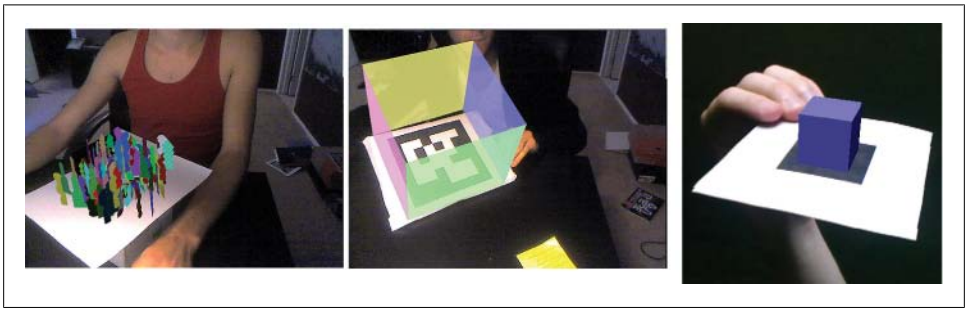


Figure 14-11. Some examples of ARToolKit in use

AR has been used in video tracking applications, children's books, magic tricks, industrial applications, and many other areas. There are efforts underway to develop addons for oF that use the ARToolkit, and currently several examples in the oF forums are available for download that show how the ARToolkit can be integrated into an oF application.

Review

Computer vision is the use of algorithms that help identify certain patterns in bitmap images and track those objects over multiple frames. These can be used with live video from a camera or with video that has already been captured.

Tracking usually utilizes "blobs," which are distinct areas within an image that the program identifies as being contiguous areas representing an object in the camera's field of view.

Most tracking in openFrameworks is done using the OpenCV library and engine. The OpenCV project was initiated by Gary Bradski in 1999 and is a free and open source project computer vision library. It provides tools for tracking bobs, finding faces in images, and manipulating images among other things.

To use OpenCV in openFrameworks, use the ofxOpenCV add-on created by Zach Lieberman and Stephan. To use OpenCV in Processing, you can use the OpenCV library created by Douglas Edric Stanley.

The ofxOpenCV add-on allows you to create an instance of `ofxContourFinder` to detect contours in an image passed to the plug-in. This class has a `findContours()` method that can be passed a grayscale image to detect contours as well as parameters to determine the maximum and minimum size of blobs and the number of blobs that the contour finder should look for in the image.

A virtual method is a method that is declared but not defined in a base class that must be defined in any class that extends the base class. This is often done to ensure that a class has a particular method that can be called by another class, as in the case of the `ofxCvBlobListener` class in this chapter. Creating an application that extends `ofxCvBlobListener` requires that the application can define the `blobOn()`, `blobMoved()`, and `blobOff()` methods that will be called when the application detects blobs moving.

To use OpenCV in Processing, create an instance of the OpenCV object in your application, and call its `capture()` method in the `setup()` method of your application. In the `draw()` method of your application, call the `read()` method of the OpenCV object.

Blob detection using OpenCV in Processing is done using the `blobs()` method of the OpenCV library.

Gesture recognition is the process of checking the movement of an object, either the mouse or a finger gesture. Gesture recognition in Processing can be done using the ezGestures library. In oF, this can be done using the ofxSimpleGesture library.

Face detection can be done in OpenCV by using a Haar Classifier XML file that is loaded into the application at runtime. The example shown in this chapter uses a Haar file that has been set up to recognize faces that are looking directly at the camera. Other files are available for full body tracking or tracking the sides of faces. Haar files can also be trained to find many other types of shapes as well.

Many touch devices use computer vision and blob tracking to create multitouch interfaces, such as the reActivision project and the NOR_/D TouchScreen.

Tuio is protocol designed by Martin Kaltenberger that helps describe common properties of controller objects on the table surface as well as of finger and hand gestures performed by the user. For instance, the number of objects on a table, the movement of touches, rotation, and gestures can all be communicated between applications using Tuio.

CHAPTER 15

Movement and Location

Throughout this book we've examined the types of data that users can generate and ways to use that data to help users perform certain tasks. In this chapter, we'll look at using movement and location as a dataset, creating what are often called *locative applications*, that is, applications that are location aware.

One way of doing this is by using GPS data. GPS stands for Global Positioning System, and it relies on a system of satellites to provide triangulated data about the location of a GPS device at a given time. Another way of doing this through an Internet connection. The GPS devices that you'll be learning about in this chapter are quite small, some no more than 2 inches square. This makes them appropriate for projects and devices that need to be embedded within another object or that need to be extremely small. There are other ways of getting location information than just using GPS. If a computer or browser-enabled device such as a mobile phone is connected to the Internet, we can determine with some reasonable accuracy the location of the computer. This becomes an easy way to work with the tools that a user already has and is familiar with.

Using Movement As and in Interaction

Our location is information—what does determining where someone is tell you?

We can look at how to use movement and location in interaction in two fundamental ways. The first is the notion of knowing where someone is. The ability to determine users' positions allows you to retrieve information relevant to their current locations. The weather is one easy piece of data that you can use once you know a user's general location. If you know more precise information, such as the actual location with a few hundred meters of accuracy, you can begin to provide many different kinds of information. Mobile device applications use location information to send relevant information about that particular location: public transport options for commuters, shops, and sites of historical significance for self-guided GPS tours are just some of the possibilities. However, these require a high degree of accuracy that isn't always possible, though you can use certain libraries to get exact location information. We'll discuss these later in

this chapter. At their core, these sorts of applications use a user's location as a parameter for determining what data to retrieve.

Another very interesting genre of interactive applications that uses much the same idea, though in a much different paradigm, is location-based gaming. These types of applications use the users' positions to create positions in a game world that in some way or another mirrors the actual world. Thinking about movement for a moment, it inherently lends itself to games and to play, such as with races, games of movement, capture the flag, and tag. This is another theme that we've looked at again and again in this book: leveraging existing experiences and situations from life to create interactions that are intuitive and immediately understandable for the user. Locative and geolocative games also easily allow for very social game play that provides many different sorts of experiences than those games that orient the user toward a screen or console.

Locative gaming has a rich and long history. One of the earliest locative games, BotFighters, was released in 2000 in Sweden. It was a mobile game developed by It's Alive! It was revolutionary in mobile gaming for taking advantage of location-based services as a key element in the game play. This is a mode of game play that is just beginning to be fully explored, both in terms of the geographical potential and in terms of the social interactions, the kinds of cooperation and competition that it enables, and in how the user perceives game play. It wasn't until recently that these sorts of games were possible. Mobile devices have only recently become powerful enough to run many kinds of games or interactive applications. Network strength in many areas was until recently insufficient for the real-time communication that this kind of game play requires.

Some of the more famous uses of GPS positioning for games include Pac-Manhattan, which re-created the classic video game Pac-Man using actual players in Washington Square Park in New York City. Area/Code, the principals of which are interviewed in this chapter, also has created games that use the idea of location both in its more obvious implementations, such as where the user is, and in more expected ways. Area/Code's game Shark Hunt used the actual locations of real sharks in the Atlantic and Pacific oceans mapped by GPS to drive the game play of the user. A great number of companies, artists, and designers have worked with the idea of location as a core element of play and games. The designer and theorist Thomas Dreyer identified seven fundamental modes of using locative data and locative devices in game play that provide a very interesting window into how to use this technology in play. Loosely paraphrased, these seven approaches follow:

- The hardware or interface is how the players get information about what they can do and what their goals are.
- The hardware or interface augments the actual environment by adding game play elements that are visible or detectable only through the interface.
- The hardware or interface is a way to connect to other players.

- The hardware or interface is a way to keep track of scores or changes in the game play via turns.
- The hardware or interface is a way to provide the players with directions or instructions to facilitate game play.
- The virtual world and the physical world can be connected or disconnected by the players as elements of game play. When connected, the virtual world correlates in some way or another to the physical world.
- The movements of the player are in one way or another used as data to drive game play that happens in the virtual world.

The last approach points to another way to think of movement data: as a history of the movement of a device. In this way, the location data stops being simply an input into a system and becomes instead a database to be mined. Movement can be a narrative or a mapping. This requires storing, logging, and transmitting this data back to a machine that can parse the data. There are a few different approaches to doing this that we'll examine later in this chapter, but the core idea remains the same: instead of using the data to drive the nature of the feedback given to the user, the data is itself used as the generation of a database that will then be parsed and used later. This lends itself easily to the idea of research, such as how people navigate a space, city, or a park, for instance.

Many mapping applications like Google Maps can map this data to photographs or to terrain maps using a markup language called KML. The breadcrumbs project introduced later in this chapter will show you how to store GPS data in the memory of your Arduino and then write it back in KML format. Creating digital maps or mappings often relies on geographical data though in a much different sense. The Pastiche project by Christian Marc Schmidt and Ivan Safrin is one such project that remaps New York City using keywords found on blogs and newspaper articles and arranged around the neighborhoods they refer to creating a map of the impression of the neighborhood. Masaki Fujihata is another artist who has made great use of GPS devices and technology in creating artwork using GPS and GPS data, working with ideas of location and geography for almost 25 years.

Using Software-Based Serial Ports

The Arduino includes a hardware-based communication capability that you've been using throughout this book. This serial capability is actually a hardware device called a UART. The nice thing about working with this device is that it allows your Arduino to read data even while the processor is busy. The problem is that there are limited numbers of UART ports on an Arduino. However, the Arduino also allows device communication using software that emulates serial communications. One library to do this, called SoftwareSerial, was developed for the Arduino controller and is included in the standard Arduino download.

Note that the `read()` method of the hardware `Serial` object asks the library to read information from the buffer; the buffer gets filled with data without software intervention. In the software version, only data received when the `read()` method is called will be heard, and data sent while not explicitly reading will be lost.

Several other software-based serial libraries address certain limitations of the SoftwareSerial library. One other such library is AFSoftSerial developed by Adafruit Industries. Another is the NewSoftSerial library developed Mikal Hart, who is also the developer of the TinyGPS library that you'll be using to parse GPS data in the next section.

The biggest advance in these two newer libraries is the introduction of interrupt-driven receives, which was a dramatic improvement over the polling (constant attention) required by the native SoftwareSerial. Without interrupts, a program running on the Arduino has to poll the serial port at very short, regular intervals. So, SoftwareSerial has difficulty receiving GPS data and parsing it into a usable form because the program is too busy trying to keep up with characters as they arrive to actually spend time assembling them into meaningful data. The AFSoftSerial or the NewSoftSerial libraries can process data while receiving information from a device like a GPS controller because of the hardware interrupts that they use. Both of these libraries are quite small as well; the NewSoftSerial library adds about 2,000 bytes to the size of an application, while AFSoftSerial adds about 2,400.

All three libraries have similar interfaces, which makes it easy to switch between one or the other. They all have roughly the same implementation. SoftwareSerial and NewSoftwareSerial use the `constructor(uint8_t receivePin, uint8_t transmitPin)` method. It sets up the library and sets the receive and transmit pins that the library will use. The Arduino SoftwareSerial library is set up to call this as follows:

```
SoftwareSerial(2, 3);
```

while a new instance of the NewSoftwareSerial library is constructed like so:

```
NewSoftSerial nss = NewSoftwareSerial(2, 3);
```

All the libraries contain a `begin()` method to start the application listening at the baud rate passed to it:

```
nss.begin(9600);
```

All the libraries contain a `read()` method that reads 1 byte at a time from the serial port as an integer value:

```
int val = nss.read();
```

In this chapter, the examples will use the NewSoftSerial library because of a few advantages that it has: it can have devices connected on all Arduino pins 0–19 (0–21 on Arduino Mini), it supports multiple simultaneous software serial devices, and it supports the ATmega328P chips that newer versions of the Arduino boards come with. NewSoftSerial is written on the principle that you can have as many devices connected

as resource constraints allow, as long as you use only one of them at a time. This means that you need to read the devices serially, one after the other, as this small example from Mikal Hart shows:

```
#include <NewSoftSerial.h>
// Here's a GPS device connect to pins 3 and 4
NewSoftSerial gpsDevice(4,3);
// A serial thermometer connected to 5 and 6
NewSoftSerial therm(6,5);
// An LCD connected to 7 and 8
NewSoftSerial LCD(8,7); // serial LCD
void loop() {
    // collect data from the GPS unit for a few seconds
    read_gps_data();  // use gps as active device
    // collect temperature data from thermometer
    read_thermometer_data(); // now use therm
    // LCD becomes the active device here
    LCD.print("Data gathered...");
}
```

The important thing to note is here that any time you call the `begin()`, `available()`, or `read()` method of one of the NewSoftSerial library instances, that instance becomes the active instance, the previously active instance is deactivated, and any information in its buffer is thrown out. When you use the Serial library, you can often use the `available()` method of the Serial library to determine when the device is active by looking at the integer value returned from the method like so:

```
if (arduinoSerial.available() > 0){
int c = nssDevice.arduinoSerial();
// do something with c;
}
```

This doesn't work the same way with the NewSoftSerial library. Instead, you need to do this:

```
if (nssDevice.available()){
int c = nssDevice.read();
// do something with c;
}
```

Now that you have an overview of the alternative ways of reading serial communication, you'll learn how to use those libraries to communicate via GPS.

Understanding and Using GPS

GPS is a way of determining location that has become amazingly common. It is used with airplanes, cars, cyclists, and hikers, as well as to track animals and boats. Last but most certainly not least, it can be used in creating games and designs. GPS is two things: a data format and a method of communicating with a network of satellites sending information to devices that request it. Any device that sends a GPS signal is referred to as a GPS *transmitter*, and any device that receives a GPS signal is called a GPS

receiver. Most devices that you'll be working with are receivers since they receive information rather than explicitly requesting it. This requires far less power and fewer additional electronics.

The receiver calculates its position by precisely timing the signals sent by the GPS satellites. Each satellite sends messages containing the time that the message was sent, information about its current location in orbit, and the general system health and rough orbits of all GPS satellites. When the receiver gets a message, it measures the transit time of the message and uses that to compute the distance to each satellite. The receiver used the calculated distances to create an accurate estimate of the current location, called the *fix* for the device.

The receiver uses triangulation, combining the distance that it took to receive the message with the location of the satellites to determine the receiver's location. GPS receivers are usually composed of an antenna that is tuned to the frequencies transmitted by the satellites, a processor, and a clock. Originally receivers monitored four or five channels to receive messages, but this number has increased, and today most receivers typically have between 12 and 20 channels on which they listen for messages.

GPS receivers send positional data to a PC or other device as strings of characters in the National Marine Electronics Association (NMEA) protocol. Most receivers typically send four different data strings:

- GPGGA: Global Positioning System Fix Data
- GPGSV: GPS satellites in view
- GPGSA: GPS DOP and active satellites
- GPRMC: Recommended minimum specific GPS/transit data

The strings containing the data are *sentences*. Each of these sentences contains a lot of data, but the most interesting part for the purposes of this chapter is the last one, GPRMC.

Let's take a closer look at the sample GPRMC sentence in Figure 15-1.

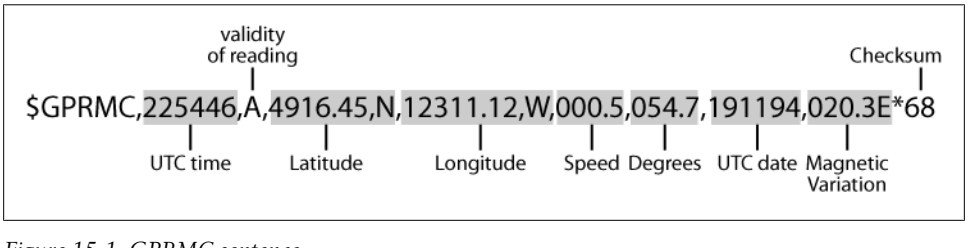


Figure 15-1. GPRMC sentence

That's a lot of data mashed into a tiny area. Table 15-1 shows just how much information is contained.

Table 15-1. GPRMC sentence data breakdown

Segment	Description
GPRMC	The type of sentence
225446	Time of fix 22:54:46
A	Navigation receiver warning. Valid values include: A = Valid position V = Warning
4916.45,N	Latitude 49 degrees 16.45 minutes north
12311.12,W	Longitude 123 degrees 11.12 minutes west
000.5	Speed over ground in knots
054.7	The heading in degrees
191194	UTC date of the fix, 19 November 1994. 2009 would be 09
020.3,E	Magnetic variation, 20.3 degrees east
*68	Mandatory checksum

There is another variation of this string, actually just a new version of the protocol, called the *NMEA 0183 version 3.00* that includes altitude data and a mode indicator field appended to the end of the string like so:

$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a,m*hh

When working with GPS data and an Arduino controller that are moving around, you probably won't be able to send the data to another computer, so you'll probably want to save the data for later use. This requires using the memory on the Arduino controller, which will be the next topic that you'll look at. First, though, we'll need to know how to read the GPS data.

Let's look at the TinyGPS library designed by Mikal Hart that is a very elegant and easy-to-use library for parsing GPS signals. Since GPS tends to take a lot of memory and memory is such a precious commodity when working with microcontrollers, TinyGPS is designed to be as small as possible. It provides position, date, time, altitude, speed, and course but ignores the rest to save space.

First you'll need a GPS chip. There are several different ones (Figure 15-2) available on different electronic supplier sites; the Parallax GPS chip is one of the more popular and well tested.

GPS chips communicate with the Arduino library over its serial port, which means that you'll need to use either the SoftwareSerial library or the NewSoftSerial library, also developed by Mikal Hart.

To get started with the TinyGPS library, download it from *http://sundial.org/arduino/index.php/*, place it in your Arduino *libaries* folder, and create an instance of the library:

```
#include "TinyGPS.h"
TinyGPS tgps;
```

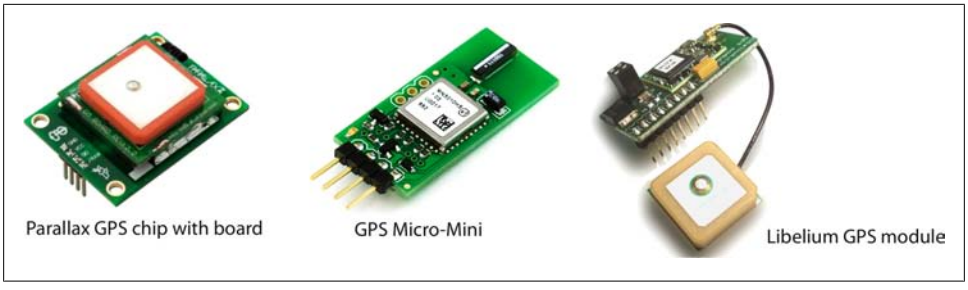


Figure 15-2. GPS chips

To feed the library data to parse, you pass one character at a time to the `encode()` method of the TinyGPS instance. The way to do this is by reading all the bytes from the software serial port one at a time until none remain inside the `loop()` method, as the following example shows. When `encode()` returns `true`, the TinyGPS library has received a valid NMEA string:

```
#define RXPIN 3  // connect digital pin 3 to the GPS output pin
#define TXPIN 2  // input to the GPS (not used in this example)
NewSoftSerial nss(RXPIN, TXPIN);
void loop()
{
    while (nss.available())
    {
```

Note that the data from the NewSoftSerial instance is an `int` rather than a `char` value:

```
        int incoming = nss.read();
        if (tgps.encode(incoming))
        {
            // process the NMEA strings
        }
    }
}
```

Now you can get the data from the TinyGPS instance using the following properties:

`lat`
: Latitude as a long or 32 bit integer. The long is just like an int, but instead of being limited to 16 bits it's a 32-bit representation, which means that it can store values up to 4,294,967,295 if it's unsigned or 2,147,483,647 if it's signed. The latitude value is a signed long, which means it can be positive or negative.

`lon`
: Longitude as a long integer.

`TinyGPS` objects depend on an external source, like the Arduino application that has created an instance of the `TinyGPS`, to feed valid and up-to-date NMEA GPS data. This is the only way to make sure that the `TinyGPS` notion of the fix is current. To test whether the `TinyGPS` object contains valid fix data, pass the address of an unsigned long variable for the `fix_age` parameter in the methods that support it. If the returned value is

TinyGPS::GPS_INVALID_AGE, then you know the object has never received a valid fix. Otherwise, the fix_age is the number of milliseconds since the last valid fix. If you are sending data to the TinyGPS instance regularly, the fix_age shouldn't be more than 1,000 milliseconds. Larger values may be a sign that the device had a fix but lost it. The best way to check, as shown in this code snippet by Mikal Hart, is to check the fix_age value every so often to ensure that the device fix is current:

```
float flat, flon;
unsigned long fix_age;

gps.f_get_position(&flat, &flon, &fix_age);
if (fix_age == TinyGPS::GPS_INVALID_AGE) {
    Serial.println("No fix detected");
} else if (fix_age > 5000) { // last fix more than seconds ago?
    Serial.println("Warning: possible stale data!");
} else {
    Serial.println("Data is current.");
}
```

The time, date, speed, and course are all available as unsigned long values. There are also a few methods that TinyGPS defines that you can use to get more general data:

gps.get_position(&lat, &lon, &fix_age)
: Retrieves +/– latitude and/or longitude in 100,000ths of a degree. Note that it takes a reference to the lat, lon, and fix_age parameters, which means that you would call it like so:

```
long latVal, lonVal;
unsigned long fixVal; // these are all null right now
get_position(&lat, &lon, &fixVal); //this method sets them to the correct values
```

gps.get_datetime(&date, &time, &fix_age);
: This method works in a similar way as get_position(). It sets the passed-in values to the time in hhmmsscc and to the date in ddmmyy. You can call this method like so:

```
unsigned long dateVal, timeVal, fixVal;
gps.get_datetime(&dateVal, &timeVal, &fixVal);
```

speed()
: This method returns an integer representing the speed in 100ths of a knot.

course()
: This method returns an integer representing the course in 100ths of a degree.

You might have noticed that all the values returned by the core TinyGPS methods are integers or long integers, which means that get_position() returns a longitude value of 10,050,000, or 100.5 degrees. This can seem odd at first, but it saves a lot of space since floating-point numbers such as 100.5 require another Arduino library. For applications that don't have strict size requirements and that can afford the extra 2,000 or so bytes that these libraries add, it might be more convenient to use floating-point

numbers. If you enable floating-point libraries in your application, the following methods are available and provide floating-point values:

`f_get_position(&flat, &flon, &fix_age)`
: Same as `get_position` but with floating-point values.

  Latitude is available as a floating-point number in the parameter `flat`, and longitude is available as a floating-point number in the parameter `flon`.

`f_altitude()`
: Altitude in meters.

`f_course()`
: Course in degrees.

`f_speed_knots()`
: Speed in knots.

`f_speed_mph();`
: Speed in miles/hour.

`f_speed_mps()`
: Speed in meters/second.

`f_speed_kmph()`
: Speed in kilometers/hour.

Now that you see how the library works, you can connect a GPS unit to your Arduino. Now, we'll look at an example connecting the Parallax breakout board to an Arduino.

With your Arduino connected to a computer, the following code will print the location of the GPS device. You'll need to make sure that both the TinyGPS and NewSoftSerial libraries are included in your Arduino application's *hardware/libraries* folder, or the code won't compile. Once it does, though, you should see the latitude, longitude, and age of the fix printed to the console:

```
// Use NewSoftSerial for greater reliability
#include "NewSoftSerial.h"
#include "TinyGPS.h"
#define RXPIN 3
#define TXPIN 2

NewSoftSerial nss(RXPIN, TXPIN);
TinyGPS gps;

long lat, lon;
unsigned long age;


void setup() {
    Serial.begin(9600);
}

void loop()
{
```

```
    while (nss.available())
    {
        int c = nss.read();
        if (gps.encode(c))
        {

            gps.get_position(&lat, &lon, &age);
            Serial.println(" ---------- ");
            Serial.println(" latitude ");
            Serial.println(lat);
            Serial.println(" longitude ");
            Serial.println(lon);
            Serial.println(" age ");
            Serial.println(age);
        }
    }
}
```

Since most GPS devices don't require any more power than the Arduino provides, both units can run from a battery and be completely mobile. So, what can you do with this? The distance to a particular location can be measured and used to drive behavior. A light that brightens as the users get closer to a particular location and an LCD screen that provides some feedback about the users' distance are two easy examples that mimic the play of hide-and-seek.

You can also aggregate the data that the movement of the device generates, creating a record of the amount of movement or the total change in elevation over the course of a journey. This sort of averaging creates a summed record of movement that can be interesting for looking at hikes, the movements of animals or bicycle riders, the travels of a balloon, or any other number of projects. This is different from logging the data, that is, creating a record of each movement. Logging requires a different approach because the Arduino does not have the memory necessary to store more than a few GPRMC data strings. Later in this chapter, you'll learn more about data logging.

Interview: Area/Code

Area/Code makes cross-media games and entertainment. Area/Code takes advantage of today's environment of pervasive technologies and overlapping media to create new kinds of entertainment. Games and media define imaginary spaces to enter and explore. Area/Code highlights the connections between these imaginary spaces and the world around them.

These connections can take many forms: urban environments transformed into spaces for public play, online games that respond to broadcast TV in real time, simulated characters and virtual worlds that occupy real-world geography, game events driven by real-world data, and situated media that corresponds to specific locations and contexts. Kevin Slavin, Frank Lantz, and Kevin Cancienne responded to these interview questions and gave insight into their work.

Explain in broad terms what your company does.

Area/Code: When we started the company four years ago and took the name Area/Code. We focused on the idea that there were connections between something distinct and the cloud, the area, and the code. Four years ago when we were first talking about physical and locative media as a direction that this entertainment was going take us, nobody had any idea what we were talking about, even to some degree us. Now there are so many examples of that the need for that conversation diminishes over time. Four years ago nobody knew what Nike+ was, the Wii hadn't come out yet, and there was no Guitar Hero. The idea of things that had hybrid physical and digital forms is now much more present; we just spend less and less of our time having to explain that premise, which is great.

If you look at how we evolved over the years and how we've come to express or pursue that premise, it's an interesting metaphor for one of those challenges that you've said you've noticed, which is that people look at technology very deterministically. That is, "because my board can do this, my project or the thing I'm trying to express must look like that." We've always allowed creative to lead the technology, even if the technology is sometimes the kernel and core inspiration around which we build the experience. I think that we've been lucky that we've been able to really pursue our crazy ideas and principles about game design and about interactivity as opposed to being beholden to expressing every feature that this one buzzword or knickknack might afford you.

It's using technology to inspire an idea for an experience, not to determine what the experience is going to be.

What are the tools that you're using and noteworthy things about those tools?

Area/Code: One of the challenges of Area/Code is that in every project that we do, part of its value—whether it's for us or for the people who are funding it—is that nobody has ever done it before, and that includes us. What that means is that we're not really platform based. We don't have the toolkits and the systems so that the next time we just reuse an existing game and alter it to use dogs that have GPS receivers attached to them instead of sharks. So, the portfolio of the technologies that we've used is insanely diverse because the projects are so diverse.

I think another one of our tricks is that a lot of our technology is actually fairly mundane stuff, like LAMP (Linux, Apache, MySQL, PHP). PHP, MySQL, Flash, and other fairly ubiquitous technologies are already doing what we need to do. Interactivity has for us come less from devices or technology as from data sources. In Shark Hunters or the *Sopranos* games, it looked like a cool technology trick, but it was just clever data mining that drove the behavior. That's been one of our insights—that there's all kinds of raw material out there from which you can create interesting interactivity if you have the right eye for it and if you allow the creative side take the lead, instead of the technology. We've begun with a text file and turned it into a game that feels like augmented reality.

Provided you have the *stuff* that you're interested in, you can create a project as a small-scale experiment. Or you can wait around for the technology to catch up with the idea. Or you can figure out how to do it for real, using the existing technology. We've taken the third approach. Back when we were doing the location-aware gaming project with

multiple teams running around New York City, we would use QR (Quick Response) codes, because it allowed us to rig a system that was location aware.

We didn't know where the teams playing the game were, but we could get them to take pictures with cell phones by giving them points for doing that, but we knew that if a poster that contains the QR code is located in that photo we know where that QR code is so we can figure out where they are without anything sophisticated. That's not the best example, but the point is that this was done in 2004 when every phone was location aware, but we couldn't get to that information as a third-party developer. So, we created a ground-up location-sensing system and mapped the codes to the location. Then we knew where the phone was when they sent us a picture. We knew where each of these teams were, because the activity that those phones were engaging in was tied to places, which was a very kind of weird idea.

One of the things I keep trying to get people to think about is the relationship between data and feedback. What's good and what's bad input?

Area/Code: That's one of the pitfalls of sight-based interactive art and physical computing projects: using an overambiguous form of interaction, where the users know they're affecting the system in some way. They're waving their hand, they're entering their Social Security number, or it's scanning their DNA, and as a result, *something* is happening but what that something is doesn't get defined clearly enough. Game design is sort of the art form of the type of feedback system you're describing. Making my interaction with the system interesting and meaningful is the art form of a game design. Not that you can't also do it in other projects that are not games, but it's often not as essential. In games, you are forced to wrestle with making it clear to the player how they're affecting the system, what their choices are, why those choices are meaningful, what the result of their action is, how the system is behaving, and what all the various ramifications or possibilities are based on, why they should explore it, and which direction they're moving in.

How do you describe what you do?

Area/Code: Well, we started out with games that took place in the real world with an element of location awareness or physicality to them, and as we evolved, we started to branch to other kinds of projects and games that cut across other media, such as games that you play on your laptop while watching TV or *Parking Wars* on Facebook. There's really no physical location aware expectation at all, and the game has some of the same flavor of inserting itself into your life in a weird way, into your relationship with your friends, into when you go to sleep, and into when you're at work. Or when you're on vacation. It has an intersection with your real life. The common thread between all those things is that we make games that have some interesting relationships to the real world outside of the game.

A lot of it is about taking the various aspects of ubiquitous computing and making games that aren't happy to live in any one context or aren't happy to live in just the conventions that you might assume when you think of video games. Now that we have a place like Facebook, a place where millions of people come together virtually, what does a game look like there? We want to allow it to really seep out into those edges, as opposed to working within the box that Frogger was in.

That same thought process applies to all sorts of situations. If you have a laptop that knows where it is, what kind of game would you like to play with that? What happens when millions of people tune into a really popular TV show at the same time? Imagine that as a space where we create a game, and what does that look like?

There isn't a neat box that we work within, but there is a neat box that we work outside of, not in a cliché sense but in a sense of what video games have generally been about: setting up a fiction world, and the games take you into the fiction world, which is bounded on all sides. For us, what's interesting about the stuff that we work on is that the system of the game starts to leak outside of the boundaries of the game itself.

We also draw a lot of inspiration from the history of games, from precomputer games, from sports, and from board games. The history of games is lighter on the pretending aspect, lighter on the make-believe, and stronger on the social interaction. It's a form of stylized social interaction. That's a flavor we like, and we feel that the industry as a whole is going to move in that direction. I think there will be video games where there will be virtual worlds that you can disappear in, but we're more interested in, you know, other directions that the games can evolve.

What are your goals for the players of your games? What effect do you want to have?

Area/Code: I want people's lives to be improved by interacting with stuff I make, by playing the games. If you play one of my games, I want you to walk away having had your mind expanded, your soul enlarged, and your view of the world more interesting. I think there's a deep relationship between the aesthetic goals and ethical questions that they kind of bring up. What are our responsibilities for any of those things? You are creating a form of stylized social interaction.

It's based on empathy. I ask, what are the types of games that when I played them I felt like I benefited, my life has been made better, I'm glad that I played, and I'm glad that I put tons of hours into them? That's the kind of game that I want to make. I think it's rooted in that sort of empathy with the player. The person who's going to be participating with the system that you're designing is a good compass.

One of the things that you're doing when you're creating a game is bounding people's behavior, creating rules, and in a sense restricting them. What is the design process for accomplishing this?

Area/Code: Well, that's the fundamental paradox of game design: when you're designing a game system, you're really designing separate rules that constrain the behavior of the player. People think that rules are restricting, eliminating, fixed, and tell you what you can't do. But when you submit yourself to the rules in a game system, you end up producing play, which is the opposite of the system. It's spontaneous, creative, imaginative, and improvisational. So, by the same token in the creative process when you're designing a game, we often struggle with constraints and the constraints of the technology. Wrestling with the material reality of the clay is what produces the dig. Sometimes it's the client. When we're doing a client-based project, it's what they want to say and what they want to express, and we wrestle with those things, and it's useful.

In *Parking Wars* what's unexpected is that it's a game in which there's almost nothing that you can do. The only thing that you can do is park on somebody else's street or

ticket somebody who's parked on your street. Those are the only rules of the game. And yet this produces such deep forms of engagement. You have 1.5 million players, one of them is going to die, and a woman who was a hardcore player of the game did die. Her presence and subsequent absence were felt so strongly within the hardcore players; just because she stopped parking on other people's street, the other players felt the loss, which means that she was that present. In the narrowest possible vocabulary, they found ways to express grief. They left one spot open all day on their street. And in fact, if they could've done anything in the world, they might not have done anything, but because their vocabulary was so limited, it became wildly expressive. That's what strange about this notion of bounding behavior.

It's one of the things that games are good at doing—creating debates, drawing lines, and allowing us to think about our social behaviors. Like saying "You like the Red Sox, screw you!" And you can do that because it's absurd, because it's the weirdly expressive abstract thing.

How do you approach thinking about the actual object with which users interface?

Area/Code: It's a really interesting and important aspect of the design process to the degree that it's where, if what you're designing is an interactive system that's interesting and beautiful to interact with and participate in, communicating that to the player is the most important thing. What is the state of the system, what can I do to change it? We just released our first iPhone game; it's called *Rock 7*. During the design process, I spent a lot of time fine-tuning the behavior of the animation, and I remember being struck by how much the underlying system, the actual rules of how the system behaved, was a really important aspect to it. It was almost like a discovery. We discovered this set of behaviors that was inherently fascinating and deep and interesting. Then so much of the design process was communicating that to the player: communicating what's happening, why it's interesting. We're doing that by the rate at which animation plays, where we draw the viewer's eye, we're communicating about the system. Game design is so often the aesthetics of decision making, meaningful interaction, and these kinds of more crowd-sourced interactions.

One of the benefits we have that a lot of big mainstream people developing large-scale projects don't is that they have to devote almost all their attention to constructing a simulacra reality that's more and more realistic. What they're doing is creating a simulated 3D space. You can devote so many resources to just polishing the kind of realism aspects of it. We get to devote all those resources to this question: what is interesting and important for the user to know and understand at each point in order to enhance their appreciation of what's going on in the system? What we do has a greater range. Things can be more stylized and abstract. We can think more about these issues and less just about making it look "more real."

I think we're lucky to have a pretty good rule of thumb about what constitutes good feedback and what constitutes knowing whether or not you're doing your job and giving the user all the information that they need to know. We have our aesthetics well defined, we know what makes one of these systems good, we know that we want you to know what's good in there, and we just check to make sure that we're communicating that

clearly. I think that's part of what attracts us to working in game design as opposed to some broader form of interactivity.

What will a social or locative game look like in a few years?

Area/Code: I think what social games have done in part is to make it very clear in everybody's head that games are built out of people. For instance, the fundamental difference in the ethos about MySpace and Facebook is that MySpace was really built around the idea that there was content that had social activity around it, and there was media and people would congregate around that. Facebook was the opposite of that. What people are going to congregate around is each other, and then there's going to be these things that hook on to the stuff that we do. Therefore, games that are emerging, for example on Facebook, are tapping into existing social activities rather than trying to just produce them from scratch.

When creating social games, we're exploring the ideas of ubiquitous computing and gameplay that is always on as you're running around the world and we're exploring how that changes gameplay. I worry a little that it will take a direction that I'm not as interested in seeing it go which is, taken to its logical extreme, a world where everything you're doing is part of the game somehow, you're pumping gas and getting points in some game. I feel like something is lost in there about the things we've been talking about, the good aesthetics of games.

We're looking at things that are explicitly designed as game experiences and have a really interesting weird interaction with the real world. It's not that it blurs what is the game or isn't a game, it's just that it creates all kinds of connections that you haven't thought up before. It could be representational, the way this game inverts my normal relationship with my friends, or it can be in terms of making you aware of your habits, such as plotting my trip by air. I'm thinking about how I'm going to a place where I can pick up some rare thing in this game I'm playing. There's a concept referred to as "the magic circle." The magic circle is the game that is stylized rituals of social interactions. You step into a game, and by doing so, you are able to participate in this really interesting system.

We continue to be interested in getting people to realize or think about the idea that the magic circle doesn't really happen on a TV screen or doesn't just exist in a football field, and probably throughout human history. It might sound contradictory to say that I don't want to imagine a future where we're constantly playing a game where every activity, where every real-world activity is imbued with this frivolous game meaning. However, I do think it's interesting to have people continually engage with the idea that games have this real value that's nonfrivolous—that there's a good reason to be interested in playing a game. Most social games are built on ubiquitous technology and get different kinds of people interacting with each other and interacting with this stuff and engaging with issues that have value. That's a good thing, and that's what I want more of.

Genuinely interesting, genuinely good, mathematical models, and things that expand your mind are worth spending time doing.

Storing Data

As mentioned earlier, the Arduino can't really store GPS data in its RAM memory. You can store some data but not enough to be useful. Another way to store the data without using the RAM memory is to save the data to the EEPROM memory of the Arduino. EEPROM stands for Electrically Erasable Programmable Read-Only Memory and is a type of nonvolatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, for example, data in calibration tables or device configuration data. The Arduino has an EEPROM library that you can use to store such data. This can be very important if you want to store variables without taking up runtime space.

This requires a little explanation of how memory works. There are three types of memory in the microcontroller (either the ATmega168, ATmega328, or ATmega1280, depending on which version of the Arduino you have) used on the Arduino boards.

The Flash memory is where your actual program is stored on the Arduino. This information is nonvolatile, which means that it exists after the power to the board is turned off. This is why you can turn the power to the board off and then on again with no ill effect. You can think of this as being like the hard disk of your desktop or laptop computer.

Static Random Access Memory (SRAM) is where the sketch creates and manipulates variables when it runs. This is volatile, which means that when the power is cut or any time the program restarts, the values in memory are lost.

Finally, there's EEPROM, memory space that programmers can use to store long-term information. It is space that is not needed to store the application or any variables that the application will create while running. This is nonvolatile, so values can be written into it and recovered later, even after multiple restarts. EEPROM is slower to read and write than SRAM, and it can be written only a fixed number of times before the memory cannot be reset again, though this does take about 100,000 cycles before that happens.

As of the writing of this book, a few different types of chips are used with the Arduino and with Arduino-compatible devices. One thing to note is that 2 KB is used for the bootloader that starts up the application when the power is turned on. Without this, you wouldn't be able to simply plug the Arduino in and have it respond to your PC for updates over USB.

Right now, Arduino devices have either the ATmega168 or ATmega328 processors on them. There are some other Arduino compatible boards, such as the Sanguino that has an Atmega644 processor or the Illuminato board with a ATmega645 processor, that have even more memory and EEPROM available. Table 15-2 compares the different processor types that these boards have.

Table 15-2. Chips and memory

Chip	Flash memory	SRAM memory	EEPROM
ATmega168	16 KB	1 KB	512 bytes
ATmega328	32 KB	2 KB	1 KB
ATmega644	64 KB	4 KB	2 KB
ATmega645	64 KB	4 KB	2 KB
ATmega1280	128 KB	8 KB	4 KB

When you're running a GPS sketch, one of the most important things to do is to get data from a GPS unit and save it so it can be retrieved later. First, we'll concentrate on how to save the data; then, we'll look at learning how to read GPS data from a GPS unit.

Writing into the EEPROM memory is done using the EEPROM library. You use the `write()` method:

```
void write(int location, uint8_t value)
```

The `write()` method takes two parameters: the location at which to write into the memory and the byte written into that location in memory. As shown in Table 15-2, the Atmega168 has 512 bytes of EEPROM memory, which means that you can write up to 512 byte values into the memory, for instance:

```
void setup(){
int i;
    for( i = 0; i< 512; i++) {
        EEPROM.write(i, i/2);
    }
}
```

You would only want to run this in startup, not in the `loop()` method because, as mentioned previously, the EEPROM memory has a limited number of read/write cycles. This snippet of code would fill the EEPROM memory of an ATmega168 with the numbers 0 to 255. On an ATmega328, you would be able to store up to 1,023 values; on an ATmega644, it would store up to 2,048.

To read the values back from the memory, you call the `read()` method:

```
uint8_t read(int location)
```

This returns the value in memory at the location passed as a parameter. In the following example, the serial port is used to set the mode of the microcontroller. The `read()` method is called on startup, and if a serial communication is sent, then the mode is set to that value using the `write()` method:

```
#include <EEPROM.h>

#define TEST_MODE 1
#define DEMO_MODE 2
#define DISPLAY_MODE 3
// start reading from the first byte (address 0) of the EEPROM
```

```
int address = 0;
byte value;
byte incoming;
byte mode;

void setup() {
    Serial.begin(9600);
    mode = EEPROM.read(0);
}

void loop() {

    if(Serial.available() > 0 ) {
        incoming = Serial.read();
        boolean diffd = false;

        if(incoming == 1 && mode != TEST_MODE) {
            mode = TEST_MODE;
            diffd = true;
        }
        if(incoming == 2 && mode != DEMO_MODE) {
            mode = DEMO_MODE;
            diffd = true;
        }
        if(incoming == 3 && mode != DISPLAY_MODE) {
            mode = DISPLAY_MODE;
            diffd = true;
        }

        if(diffd) {
            EEPROM.write(0, mode);
        }

        // now do different things based on the mode

    }
}
```

You'll notice in that example that some pains are taken to ensure that the EEPROM memory is written to only if necessary. This is because there is a limit to the number of times that this memory can be completely written to and then erased. After more than 100,000 erase/write cycles, the EEPROM will stop working. This is a hardware limitation of the processor itself that you need to be aware of if you're going to be making a project that needs to read and write very frequently over a long period of time. Take care to only write values that you need, and you'll make your EEPROM writable and reliable for much longer.

Logging GPS Data to an Arduino

As discussed earlier, aggregating GPS data on an Arduino controller is fairly easy. Logging data is another matter and is slightly more difficult. Luckily, as with so many

things, there's an open source library that's been developed that makes this task substantially easier. There are also some open source hardware solutions.

Using the Breadcrumbs Library

breadcrumbs is an Arduino GPS tracking project developed by Brian Griffin. Its goal is to log GPS position information to the Arduino's onboard EEPROM. With only 512; 1,024; or 2,048 bytes of storage, breadcrumbs stores only heading information. The breadcrumbs code saves space by scaling the heading value and only making a measurement after a certain distance has been traveled and so can store several kilometers of GPS data.

breadcrumbs has the ability to read back stored GPS data, translating it to KML sentences via its serial port. This output can be stored in a file that Google Earth can directly read.

The breadcrumb code is an Arduino sketch, which means that you simply upload it to your board and run it.

These are the key methods of the breadcrumbs application:

`void gpsEncodePosition(float lat, float lon)`
: This method either erases the EEPROM memory if this is the first write operation for a session or calculates the current direction based on the last latitude and longitude position and saves it to the EEPROM in a condensed form.

`void gpsDecodeEeprom(void)`
: This method converts all the GPS data from the EEPROM data into KML and writes it to the Serial port so that it can be seen in the data window of the Arduino IDE.

If you find the variable `selPin` the *breadcrumbs.pde* file, you'll see the following:

```
int selPin = 5;                  /* Mode select pin */
```

The breadcrumbs library expects that if pin 5 reads HIGH at any point that the Arduino should be put into decode mode and write all the GPS data to the Serial port. You can connect a button to pin 5 or create another more appropriate way of setting the mode to decode if you like.

Once you receive the KML data from breadcrumbs, it will be something that looks more or less like the following:

```
<kml>
<Placemark>
<Style id="lineStyle">
<LineStyle>
<color>800000ff</color>
<width>10</width>
</LineStyle>
</Style>
```

```
<Point>
<coordinates>-118.283638,34.150879
</coordinates>
</Point>
<LineString>
<coordinates>
-118.283676,34.151073 <!--29-->
-118.283867,34.151134 <!--24-->
-118.284012,34.151268 <!--26-->
</coordinates>
</LineString>
</Placemark>
</kml>
```

Breadcrumbs is a very useful way to log GPS data without needing to purchase extra hardware, and it's also an excellent way to start thinking about how to log data yourself to the EEPROM memory of your Arduino controller. There are other more powerful ways to log data using SD flash cards that allow you to store several megabytes of data.

Implementing Hardware-Based Logging

You can consider a few different hardware-based logging strategies if you're looking to log more data than the breadcrumbs project or another EEPROM-based approach allows.

Adafruit industries, run by the prolific Lady Ada, has created the GPSShield, an Arduino-compatible shield that can be used to log data to a flash memory card. This shield supports four different GPS modules and stores data on a standard SD flash memory card. The Adafruit team has created a library to help simply plug it into your computer when you've finished your data capture, and the plain-text files are ready for importing into Google Earth, GPSvisualizer, or a spreadsheet. It's quite light, with the total weight of shield, card, GPS module, and Arduino at only 75 grams. It also fits nicely inside an Otter box or other waterproof case for all-weather logging. Their website *www.adafruit.com*, also includes some example sketches that show how to parse NMEA data and log data to a file on the SD card using the AF_SDLog library developed for the GPSShield and the NewSoftSerial library. It can run for about 3 hours with a 9V battery and up to 12 hours using the Adafruit MintyBoost power supply. The GPSShield and the MintyBoost are both available from the ladyada store along with a lot of other creations that extend the Arduino platform for all sorts of interesting projects and applications.

Another powerful option is to use the microSD module for the Arduino controller built by Libelium (Figure 15-3).

The SD socket is connected to SPI port. This module enables you to store a lot of information, because a 1-gigabyte microSD card is included with the module, meaning that you can store a massive amount of GPS data. The microSD module can be attached to digital pins 8–13 on the Arduino. Pin 8 should be set as HIGH to output power to

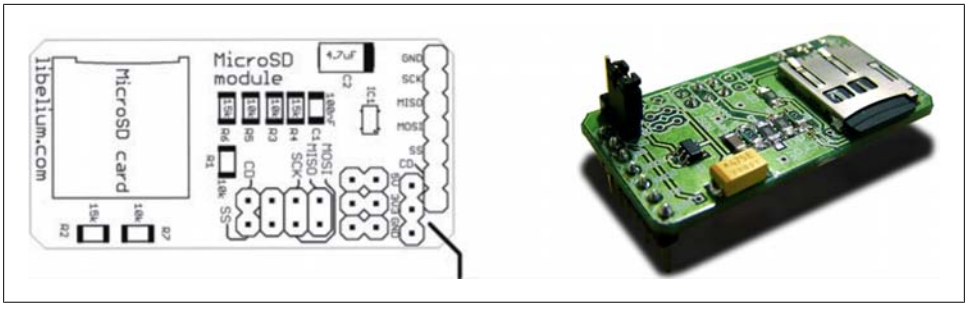


Figure 15-3. microSD module for Arduino

the module. This means that you still have a good number of pins available to connect a GPS device and any other peripheral devices that you'd like to connect as well.

Since the SD card acts more or less like a hard drive, you'll need some additional libraries that have been developed by the Libelium team to help you create applications that store and read back substantial amounts of data. The example applications included in the downloads for the microSD shield published by the Libelium team shows the writing and reading of a text file.

Sending GPS Data

So far in this chapter, GPS and locational data has been either saved or sent over the serial port to a listening computer. There are some other options for sending data, each of which has its own advantages and limitations.

There is an interesting module that Libelium developed for allowing an Arduino to send SMS data and make phone calls, provided that you connect a valid SIM card for a phone network to it. It includes the HiLo SAGEM communication module that handles the communication with the SIM card and Arduino, but you need to provide a working SIM and an antenna for the communication module to use to make calls. Figure 15-4 shows the Libelium GPS shield and its schematic.

As you can see from the following code snippet, the commands to instruct the GPRS shield to send an SMS message are a bit difficult to read and understand at first glance, but they are mercifully short and simple:

```
Serial.print("AT+CMGS="); // send the SMS the number
Serial.print(34,BYTE); // send the " char
Serial.print("2125551212"); // you'll want to use an actual number
Serial.println(34,BYTE); // send the " char
delay(1500);
Serial.println("hey"); // the SMS body
delay(500);
Serial.println(0x1A,BYTE); // end of message command 1A (hex)
```

As of the writing of this book, the Libelium store sells a kit that includes both its GPS module pictured in Figure 15-4 and the GPRS module as well as a demonstration of

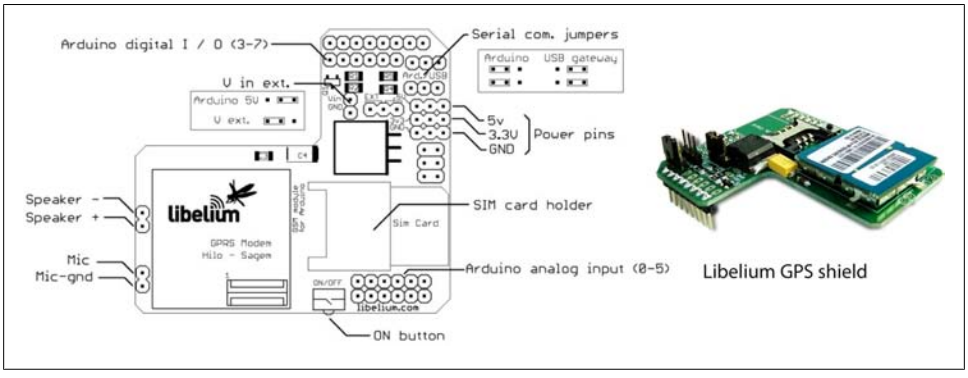


Figure 15-4. The Libelium GPS shield and schematic

how to send GPS signals as SMS messages. Though this is a fairly inefficient way to handle large amounts of data, such as data logging, it would be effective for notifying a remote user or machine that a certain location has been reached or that the user wants to notify the system of their location at that time. You can find more information on the GPRS shield and information on how to purchase one of these devices on the Libelium site at *http://libelium.com*.

A second option for wirelessly communicating is to send the location using a wireless technology such as Bluetooth if you're using either a Bluetooth Transmitter such as the Bluesmirf Gold or the Arduino Bluetooth module. This would be practical for situations where you know that the Arduino will be making periodic contact with a Bluetooth network, like a circuit race, or a controlled route where you know that the controllers will be within a certain location for the several seconds necessary to initiate communication with a Bluetooth network and send data from the EEPROM to the listening network. Using some the techniques for communicating with Bluetooth networks from Chapter 12, you could configure a Processing application to listen for the Bluetooth connection. *Making Things Talk* by Tom Igoe contains a good example of a Processing application that is configured to listen for Bluetooth communication containing GPS data.

A third option worth exploring is the Zigbee wireless radio transmitters. These have a range of 100 feet with some of the same limitations as sending data over Bluetooth, but they're slightly easier to use than Bluetooth because creating the connections is so much simpler. Again, if you knew that the GPS device would be regularly within range of a Zigbee wireless radio at intervals short enough that the EEPROM data wouldn't be overflowed, then these provide an excellent way of setting up a data dump between a remote device and a listening application.

For instance, using the XBee Pro 900 you could send data up to 6 miles outdoors with a good line of sight between a receiver and another XBee to receive the data. While these module are quite expensive, they are also quite powerful and paired with a

controller like this, could make logging data from a moving GPS shield possible. The XBee is covered in much greater detail in Chapter 17, so for the purposes of this demonstration, we'll cover just the most basic details. You can use the Libelium Xbee Shield to easily and compactly attach an XBee module to your controller and still have access to all the pins except digital pins 0 and 1:

```
#include "NewSoftSerial.h"
#include "TinyGPS.h"
#define NSSRXPIN 3
#define NSSTXPIN 2
```

The GPS connection will use the NewSoftSerial library because the XBee will be using the RX and TX pins and the hardware Serial connection:

```
NewSoftSerial nss(NSSRXPIN , NSSTXPIN );
TinyGPS gps;

long lat, lon;
char myName[] = "alberto c";

void setup() {
```

The XBee communicates at a rate of 9600 baud, so you'll want to use that baud rate to send messages over wireless:

```
    Serial.begin(9600);
}

void loop()
{
    while (nss.available())
    {
        int c = nss.read();
        if (gps.encode(c))
        {
```

The XBee module will route any hardware Serial communications through the XBee and send them out wirelessly, so print the name and then the position to any listening XBee devices:

```
            gps.get_position(&lat, &lon, 0);
            Serial.print(myName);
            Serial.print(':');
            Serial.print(lat,DEC);
            Serial.print(',');
            Serial.print(lon,DEC);
        }
    }
    delay(60000); // send our name and location every minute;
}
```

As a way to extend this further, you might want to consider storing the GPS data with the method used in the breadcrumbs application until the receiver XBee sends an message indicating that it is within range and available, then dump the data to the receiver.

Determining Location by IP Address

Another way to get users' location information is by their IP address. For instance, a Processing or oF application can get information about the users' location by determining what network they are connected to and looking that IP address up in a listing of IP addresses. If you've ever visited a website that shows a map of users currently using the website and their geographical locations, you've seen this technique used. You'll need to be able to not only connect to the Internet but also to parse the data received from a website based on that IP address.

Once you can get the users' location data, you can start to determine more things about them, like what the weather is like where they are or sometimes even exactly where they are. In the following example, the location of the user is determined by using a web service and then using another web service to determine the weather in that user's location. This is broken into a few different steps. The first step is to get the IP address of the user and translate that into a physical address. That can be done with a single call to a web service, at *http://infosniper.net*. There are a number of other sites that provide similar services as well. The data will probably be more or less the same. This example uses XML, which was explained earlier in Chapter 12, so you might want to take a look back at that chapter if you have any questions. The following is the XML I receive when requesting this data at the moment:

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
    <result>
        <ipaddress>75.121.140.47</ipaddress>
        <state>Wisconsin</state>
        <city>La Crosse</city>
        <areacode>608</areacode>
        <postalcode></postalcode>
    </result>
</results>
```

Your results, most likely, will look different. You'll notice that the `postalcode` property isn't returned in the XML data, which is a problem, because you'll need that to get the current weather for your location. For this you can use the *.csv* file that is included in the code downloads for this chapter. That file contains data like this:

```
35085, AL,  32.965120,  -86.74405, Jemison, Alabama
35087, AL,  34.303718,  -86.58323, Joppa, Alabama
35089, AL,  32.941708,  -86.06098, Kellyton, Alabama
35091, AL,  33.771090,  -86.80672, Kimberly, Alabama
35094, AL,  33.530698,  -86.55506, Leeds, Alabama
35096, AL,  33.605233,  -86.12079, Lincoln, Alabama
```

There are of course other data files that you can use to parse the same data, but this will suffice. There are a few different approaches you can take to getting the ZIP code. If it's important to be very accurate, then you'll want to use the latitude and longitude to determine what part of a city the user is in. In this case, though, since we're looking

for weather, we can assume that the weather report is going to be pretty much the same for any location in a city. That's not always true, but it's close enough for this example. It should be noted though that many web services will return the ZIP code, so the *.csv* file is a backup.

Once you have the ZIP code parsed out, you can use that value to get the weather in an area. There are many different weather services you can use. This example uses the Yahoo Weather API, though a little bit of web searching should point you to an acceptable one if for some reason the one used here isn't. The most important bit of data will be the condition node that tells the temperature, the time, and a description:

```
<yweather:condition  text="Light Snow"  code="14"  temp="29"
    date="Tue, 10 Mar 2009 8:53 pm CDT" />
```

The weather data returned from the Yahoo! Weather API is an RSS feed and looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com
/ns/rss/1.0" xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
<channel>
<title>Yahoo! Weather - La Crosse, WI</title>
...
</rss>
```

There's a lot of data in there that you won't need, so you'll need to parse through it to get to just the weather condition. Another option is to download the XML of the National Weather Service and parse through that to find the weather data for the current weather in the given location. Now, look at the code:

```
import processing.net.*;

Client c;
String locdata;
String weatherdata;
String weatherHost = "weather.yahooapis.com";
String locHost = "www.infoSniper.net";

int appState = 0;
XMLElement xml;
XMLElement xmlChildren[];
PFont font;
```

The different values retrieved from each web service are stored in the following string variables:

```
String city;
String state;
String country;
String ip;
String zipcode;
String weather;
```

The `zips` variable will store all the ZIP code data loaded from the *.csv* file in case the ZIP code isn't returned with the location data:

```
String[] zips;
Hashtable zip;

void setup() {

    size(1000, 200);
```

Initialize the `zip Hashtable`, and start parsing the *.csv* file:

```
    zip = new Hashtable();
    String[] lines = loadStrings("tmp.csv");
    int i;
    for (i = 0; i < lines.length; i++) {
        String[] bits = split(lines[i], ",");
```

This grabs the name of the city and state and turns it in a single string that will be the key for the key-value pair, so the name La Crosse, Wisconsin, for example, will become the key `LaCrosseWisconsin`, and the value will become `54601`:

```
        bits[4] = trimStr(bits[4]);
        bits[5] = trimStr(bits[5]);
        String fullname =bits[4]+bits[5];
        zip.put(fullname, bits[0]);
    }
    fill(200);
```

Connect to the infoSniper server on port 80:

```
    c = new Client(this, locHost, 80);
    font = createFont("FFScala", 32);
    textFont(font);
}
```

The `draw()` method has a large `switch` statement that determines where the application is in the process of loading and processing data:

```
void draw() {
    background(0);

    switch(appState) {
```

As of the writing of this book, the site *http://infosniper.net* is one source for this, though it cuts you off after 15 requests in a 24-hour period. You can find plenty of other services with a quick web search. This service is being used to find the user's IP address to geolocate them. There are other ways to do this, but they're more complex. Take a look in the earlier section "What to Do Next" on page 513 to find alternative methods.

The `switch` statement allows the application to use multiple steps to call the two different servers, receiving and then parsing the data, by setting the `appState` variable to different values:

```
case 0:
    println(" connect ");
    createReq("/xml.php", "www.infosniper.net", "1.1");
    appState = 1;
    // this gives a little more time to the client
break;
case 1:
    if (c.available() > 0) { // If there"s incoming data from the client...
        println(" available ");
```

If data hasn't been received from the location server, that is, then the string storing the location data, called `locdata` will still be blank, so the application starts reading it; otherwise, it continues reading data from the server until the end of the XML file:

```
        if(locdata == "") {
            locdata = c.readString(); // ...then grab it and print it
        } else {
            locdata+= c.readString();
        }
        if(locdata.indexOf("</results>") == -1) {
            return;
        } else {
            appState = 2;
        }
     }
break;
case 2:
    try {
        xml = new XMLElement(locdata.substring(locdata.indexOf("<?xml"),
            locdata.length()-1));
    } catch ( Exception e) {
  appState = 1;
        return;
    }
```

Here, all the location data properties are retrieved from the XML file:

```
    city = findElementContent(xml.getChild(0), "city");
    state = findElementContent(xml.getChild(0), "state");
    country = findElementContent(xml.getChild(0), "country");
    zipcode = findElementContent(xml.getChild(0), "postalcode");
    if(zipcode.equals("") || zipcode.equals("null")){
        String zipKey = new String();
        zipKey+=city+state;
        zipcode = (String) zip.get(zipKey);
    }
    println("found the zipcode :: "+zipcode);
    appState = 3;
break;
case 3:
    c = new Client(this, weatherHost, 80);
    appState = 4;
break;
case 4:
```

Now, fetch the weather information:

```
            createReq("/forecastrss?p="+zipcode, weatherHost, "");
            appState = 5;
        break;
        case 5:
               if (c.available() > 0) { //If there's incoming data from the client...
                println(" available ");
                if(weatherdata == "") {
                    weatherdata = c.readString(); // ...then grab it and print it
                } else {
                    weatherdata+= c.readString();
                }
                if(weatherdata.indexOf("</rss>") == -1) {
                    println(" not done w/weather "+weatherdata);
                    return;
                } else {
                    println(" done w/weather ");
                    appState = 6;
                }
            }
        break;
```

Once all the weather data has returned, store it and display the information:

```
        case 6:
            weather = getWeather(weatherdata);
            text("You're in "+city+" "+state+" "+country+" "+zipcode, 20, 40);
            text("The weather is "+weather, 20, 100);
        break;

        }

    }
```

This application has a few other utility methods. The `createReq()` method writes the request to the server. You'll notice a lot of calls to the `write()` of the `Client` class that are writing information about the request. These aren't always necessary, but servers require different parameters, and it can be helpful to see what a full request, like the one sent by a browser, looks like:

```
    void createReq(String reqURL, String reqHost, String HTTPv){
            if(HTTPv.equals("1.1")) {
            c.write("Get "+reqURL+" HTTP/1.1\r\n");
        } else {
            c.write("Get http://"+reqHost+reqURL+" \r\n");
        }
        c.write("Host: "+reqHost+"\r\n");
        c.write("User-Agent: Processing \r\n");
        c.write("Accept: text/html,application/xhtml+xml,application/xml;
             q=0.9,*//*;q=0.8\r\n");
        c.write("Accept-Language: en-us,en;q=0.5\r\n");
        c.write("Accept-Encoding: gzip,deflate\r\n");
        c.write("Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n");
        c.write("Keep-Alive: 300\r\n");
        c.write("Connection: keep-alive\r\n");
```

Sometimes it helps a server to understand that a request is finished by sending it a blank line:

```
        c.write("\r\n\r\n");
    }
```

Here, we search for the element specified in the name parameter of the `findElement Content()` method and return the content of that node as a string. It's an easy way to get the value of a particular node in an XML file:

```
String findElementContent(XMLElement x, String name) {
    String s = "";
    int len = x.getChildCount();
    for(int i =0; i<len; i++) {
        if(name.equals(x.getChild(i).getName())) {
            if(x.getChild(i).getContent() != null) {
                s = x.getChild(i).getContent();
            }
            return s;
        }
    }
    return s;
}

String trimStr(String source) {
    return source.replaceAll("\\s+", "");
}
```

Here the weather report is parsed. This could be done using XMLElement as well, but it's just as easy to chop the string up:

```
String getWeather(String source) {
    int ind = source.indexOf("yweather:condition");
    if(ind == -1) {
        return "Whoops. No weather for you.";
    }
    String report = source.substring(ind+18);
    String cond = report.substring(0, report.indexOf("/>"));
    String desc = cond.substring(cond.indexOf("text=")+6, cond.indexOf("code=")-3);
    String temp = cond.substring(cond.indexOf("temp=")+6, cond.indexOf("date=")-3);
    return desc+" "+temp+"F";
}
```

You could, for example, leave messages at particular locations by only sending a message from a server to a user if they are in a certain location. Area/Code made a very interesting game called *Plundr*, which was the world's first location-based PC game. Using Wi-Fi Positioning System (WPS) technologies, the game locates the user's computer in physical space and uses the location as part of the game. The game itself is a pirate adventure, in which players move from island to island to buy, sell, and fight for goods. Depending on where you are in the physical world, you'll find different islands, different market prices, and different ships to fight.

If you're interested in taking the idea further, you can look into some of the of the other WPS libraries available. One example is the Skyhook Wireless library, which is quite easy to use and provides readings accurate to 20 meters in some cases and to about 100 meters in rural settings. It, and most libraries like it, can be used on laptop computers as well as wireless-enabled mobile phones.

What to Do Next

There is some substantial bleed over from this chapter into almost every other chapter. GPS isn't just a neat data format; it's a way of understanding movement and location and using that to generate location appropriate feedback as well as data for input. Once again, Tom Igoe's *Making Things Talk* (Make Books) contains several examples of using GPS data and locational data that are worth looking into if you're curious about working with location-aware applications and devices. Another project that might be of interest to you is the *Mobile Processing* project that was initiated by Francis Li that allows you to create Processing sketches that will run on Windows or OS X Java-enabled mobile devices. There is a similar project in openFrameworks that has ported the oF architecture to the iPhone, enabling you to create oF applications that can run on the iPhone and take advantage of that platform.

As mentioned earlier, there is another group of microcontrollers that provide even more memory for storage of GPS data for instance and processing speed for generating that data. The Arduino MEGA has a great deal more space than the standard Arduino controller, though it is also substantially larger as well. The Sanguino developed by Zach Hoeken is compatible with the Arduino operating system while providing more power than the basic Arduino controllers. You can find information on this controller at *http://sanguino.cc*. Another option developed by the people at Liquidware is the Illuminato, which is quite similar in capabilities and power to the Sanguino. You can find more information on that controller at *http://liquidware.com*.

Review

Both the SoftwareSerial and NewSoftwareSerial libraries allow serial communication on pins other than 0 and 1 with the Arduino. This is important because it means that multiple serial devices—for instance, a GPS device—can be connected to the Arduino and information read at the same time.

GPS is a system of satellites and receivers that enable small receiving devices to calculate their positions based on their distance from multiple satellites that send out signals at a regular interval. The receivers use triangulation to determine how far they are from each satellite signal to determine their current position.

GPS devices send information in a format called NMEA. This contains information about the position of the device, the state of the network, and the current date and time. Within an NMEA, a *sentence* is the GPRMC string that contains the locational information in the following format:

```
$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a,m*hh
```

To parse NMEA string, you can use the TinyGPS library developed by Mikal Hart to simplify reading the latitude, longitude, speed, direction, and other data from the NMEA sentences.

GPS devices can communicate with the Arduino using one of the software serial libraries. There are a few options that you can use: the SoftwareSerial library supplied with Arduino or the NewSoftSerial developed by Mikal Hart.

EEPROM provides nonvolatile storage on the Arduino; that is, data is not lost when the device is powered down. It is between 512 and 4,096 bytes of data depending on the microcontroller being used.

GPS logging can be written to either the EEPROM or to an SD card attached to the Arduino using either the GPSShield from Adafruit or the microSD card from Libelium.

CHAPTER 16
Interfaces and Controls

Creating controls is one of the most interesting and challenging tasks that an interactive designer can take on because the very nature of tools implies an application that is defined by its use in a given task. A game or a toy or even a piece of art can all be defined by their playfulness, their aesthetic qualities, or their novelty. A tool will be judged by how it aids the completion of a task, which is a much more severe test of the quality of an application or system. As any industrial designer will tell you, creating a successful tool is one of the most challenging and rewarding tasks you can undertake. You need to have a good understanding you must have about a task, its context, and its challenges, and it's a different way to think about a task. When you see a task, you can analyze the different aspects of it: the hard parts and the easy parts or the subtasks that take a long time or those that don't. To understand how that task is understood by those who perform it and understand the needs that they have for a tool in that task is a different matter altogether.

However, this chapter is not just to introduce some of the things you have to think about when creating tools; it's about creating systems with which a user will interact. This can certainly be a tool, or it can be any object that a user will use for a long period of time with a specific goal in mind. That includes controllers, instruments, and systems. The kind of thinking required to successfully create a tool is invaluable when creating interactive objects. It requires a level of thinking about the human mind and body and how to adapt an object to them and for them. This process will teach you one of the fundamental aspects of all great design.

In this chapter, you'll examine a few ready-built modules that let you use familiar interfaces in new and exciting ways. These are complex input interfaces; specifically, they're complex in how the user can input information and complex in how the user perceives their input.

Examining Tools, Affordances, and Aesthetics

One of the interesting successes in device design is the popularization of interface design. In the same way that the design and aesthetics of the shape of an automobile or a fine shirt can be appreciated and understood, all the aesthetics of the interface have become popularized. Even nontechnical audiences are now regularly examining interfaces in sophisticated and critical ways and thinking not only in terms of the way that the interface helps them to function but also about the interface as a design object separate from its use. To you, the designer, that means that novel and new interfaces are interesting to users not only for what they enable the user to do but in the way that the interface itself functions. The emotional relationship between users and the interfaces of their machines has become more than a simple matter of inputting data to accomplish a task. It can now be playful, fascinating, multivariate, and engaging.

This is not to say that the playfulness, aesthetic appearance, or other nonfunctional aspects of an interface are the most important. Very often, those who use a tool most understand and value its affordances most. Something that we use every day and that we have to use repetitively for important tasks should not surprise us or take us away from performing our task. Those who either do not use a tool often or do not use it for its intended purpose understand its aesthetic affordances most. A beautiful violin that has an imperfectly crafted neck is still beautiful when it is not being played, but the person who plays music with it determines the value of the object as the user of the tool. Interfaces that are used for specific tasks have many of the same characteristics: they can have many secondary characteristics, but their core user base will judge them by how they perform their primary task.

This discrepancy between the different ways of valuing an interface sometimes leads to an incorrect estimation of its value because the commercial value of something is often determined by its popular appeal. The use value of an interface is different and should never be underestimated. The aesthetics are important, and are in fact central, but the definition is in the functional affordances. That is what defines the tool or control and separates it from a novel object that exists.

Think of an extremely simple object like a flute. It makes a specific set of tones when it is blown into in a very particular way. Now think of how complex what we do with that tone is and how many different factors can alter that tone and the way that it functions in how we experience it. From a very simple but somewhat difficult interface, we have a very wide range of potential uses and meanings and ranges of apparent application. Notice that I say "apparent applications." These aren't actual applications; a flute is designed only to facilitate making a certain set of tones. One interesting thing that happens with well-designed and simple objects and interfaces that do interesting things is that they allow for virtuosity. They let users become very good at using them in the way that some people are very good at video games or operating a backhoe or using a piece of musical software. Because a user *can* become a virtuoso with a tool, it becomes absolutely imperative that the tool be predictable even at the expense of

changes or new features or aesthetic considerations. If a user can't predict how a tool or a control will behave, then they can't practice because practice is repeating.

For artists, the notions of tools and controls are interesting elements to play with because when a viewer "uses" an art piece, it means that not only will they be interacting with it but that they will be trying to perform some action with it as well. That's not a common experience for viewers in a traditional museum, but artists like Amit Pitaru, Toshio Iwai, and Marianne Weems have all used this to great effect in installations and performance. For interaction designers, the ability to think clearly about task performance and the way to design for tasks is very important whether designing web interfaces, industrial objects, gadgets, games, or machinery. In part because this is such a fundamental aspect of design and of being a designer, there are a number of wonderful books that address these designing interactions around tools. Again, I'll reference Don Norman and two of his books: *The Design of Everyday Things* and *The Design of Future Things* (both by Basic Books). In the book *Where the Action Is* (MIT Press), Paul Dourish examines different ideas of rationality and also of tools and action in Western philosophy. He examines how thinking about tools is relevant to some of the most important questions in physical computing: What is embodiment? How does that change how we think about interaction design? How do we help users act and perform? Another interesting idea in describing and understanding how interactions work is a notion called *activity theory*, which originated in the Soviet Union but has become popular across a range of disciplines, including design. The crux of thinking in this practice draws heavily on psychology and tends to concentrates on interactions as being between a "subject" and an "object" rather than a "user" and a "system." Particularly in the design of controls and tools, a book like *Context and Consciousness* (MIT Press), edited by Bonnie A. Nardi, might be of interest when designing tools and systems for users.

The design of tools and creation of controls is probably the most complex and fundamental task in interaction design, and it's also the one that provides the greatest potential rewards. Whether you're considering more artistically oriented projects or some more practical ones, thinking clearly about an interaction in terms of it enabling a task that a user might want to perform and seeing how a good control can couple with a system are important steps in designing an interaction. First, though, a short technical reintroduction to accelerometers is needed to show an important detail.

Reexamining Tilt

For technical reasons that will become apparent later in this chapter, it's worth reexamining the accelerometer that Chapter 8 introduced. It is also a good chance to reexamine a very simple control, LIS3LV02DL (shown in Figure 16-1), that can provide input for a tool and that has a great number of potential applications in controls and tools.

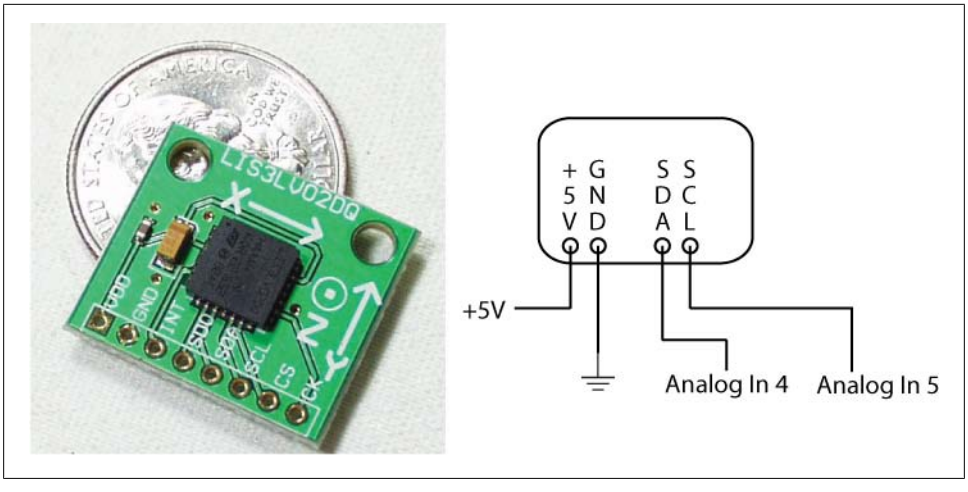


Figure 16-1. The LIS3V02D and connecting it to the Arduino board

The LIS3LV02DL is a more accurate sensor than the other accelerometers introduced in this book. However, it requires that you use the Wire library and I2C, which makes it a little more, but not a lot more, difficult to set up. Before we start looking at code, I should mention this code is inspired by the work of Julian Bleecker.

The Wire library allows you to communicate with I2C/TWI devices. On the Arduino, SDA (data line) is on analog input pin 4, and SCL (clock line) is on analog input pin 5. Since this section of Arduino code is a little longer, the code is going to be broken up into little sections with explanations nested in between. The first bit defines a few constants that are going to be used throughout the application and that correlate to the high and low values of each of the different dimensions in which the accelerometer can detect changes:

```
#include <Wire.h>
#define i2cID 0x1D
#define outXhigh 0x29
#define outYhigh 0x2B
#define outZhigh 0x2D
#define outXlow 0x28
#define outYlow 0x2A
#define outZlow 0x2C
```

Next, the `setup()` method starts the Wire library so that we can receive reports back from the accelerometer and the Serial library so we can print what's happening on the serial monitor:

```
void setup()

{
    Wire.begin(); // join i2c bus (address optional for master)
    Serial.begin(9600);

    Serial.println("Wire.begin");
```

The first message to the accelerometer sets the ID of the device that we're going to be listening for information on. All devices have an ID that they use when communicating over I2C so that the computer or processor can tell which device is sending a signal and which device a signal should be sent to. In the case of our accelerometer, it's `0x1D` or `29`:

```
Wire.beginTransmission(0x1D);
```

The next message tells the accelerometer the destination of the message, that is, which register we want to send the message to. The *register* is a location in memory of the accelerometer, and the register that we're sending the message to here, `0x20`, is the one that determines what mode the device should be in: regular mode or testing. The next message sets the device mode. If you're interested in more about the messages for the accelerometer, check the user manual for the accelerometer:

```
    Wire.send(0x20); // Tell the device the next message will set up the control
    Wire.send(0x87); // Power up the device, enable all axes, don't self-test
    Wire.endTransmission(); // all done!

}// end of setup

void loop() {
```

Next, we create variables to hold all the different values that we're going to be getting back from the accelerometer:

```
byte z_val_l, z_val_h, y_val_l, y_val_h, x_val_l, x_val_h;
```

The `x_val` is being stored as a word, which is the same as an unsigned int:

```
word x_val;
```

There's an important point to understand about the values that we'll be getting back from the accelerometer: the messages are so small that in order to get a whole value, we have to put two messages together, the low part of the int, called `x_val_l`, and the high part of the int, called `x_val_h`, in the following example. We'll be using this technique for x, y, and z values, so we'll see it again.

Here, we tell the accelerometer which value we want:

```
sendWireMessage(outXlow); // we want the low x value

if(Wire.available()) {
    x_val_l = Wire.receive();
}
sendWireMessage(outXhigh); // we want the high x value

if(Wire.available()) {
    x_val_h = Wire.receive();
}
```

With this code, the value for the x value is assembled by bit-shifting the high and low values into place. You could do this like so:

```
x_val = x_val_h;
x_val <<= 8;
x_val += x_val_l;
```

Or, you can use the `word()` method. The `word()` method has the following signature:

```
word(h, l);
```

`h` is the high-order (rightmost) byte of the word, and `l` is the low-order (leftmost) byte of the word.

This simplifies the code greatly, as you can see here:

```
x_val = word(x_val_h, x_val_l);
```

Figure 16-2 shows what is going on.

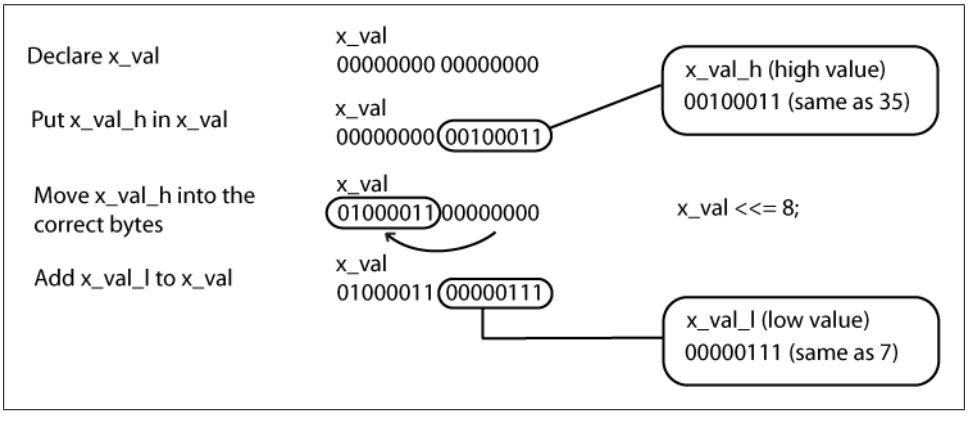


Figure 16-2. Creating an integer from two different bytes

The rest of the code follows the same pattern for the y and z values:

```
int y_val;
sendWireMessage(outYlow);
while(Wire.available()) {
    y_val_l = Wire.receive();
}

sendWireMessage(outYhigh);

if(Wire.available()) {
    y_val_h = Wire.receive();
}

y_val = word(y_val_h, y_val_l);

// ------ read the z-axis
int z_val;
sendWireMessage(outZlow);
while(Wire.available()) {
    z_val_l = Wire.receive();
```

```
}

sendWireMessage(outZhigh);
if(Wire.available()) {
    z_val_h = Wire.receive();
}

z_val = word(z_val_h, z_val_l);
delay(250);
}
```

The `sendWireMessage()` method is where sending the data over the I2C connection is taken care of. While this code could be repeated over and over again, it's far more efficient to write this functionality once, in one place, and use it again and again:

```
void sendWireMessage(byte message) {
    Wire.beginTransmission(i2cID);
    Wire.send(message);
    Wire.endTransmission();
    Wire.requestFrom(i2cID, 1);
}
```

You may ask yourself whether we really just go through bytes, bits, and all that stuff, *just* to look at another accelerometer. In a word, no. Now you have a more solid footing in working with binary information and that's going to be very important when reading and storing GPS data.

Exploring InputShield

The InputShield shown in Figure 16-3 is one of the two open source hardware products from Liquidware that you'll be seeing in this chapter. The InputShield is a small shield that fits on top of the Arduino controller and provides a small joystick, two input buttons, and a vibration motor in much the same configuration as the classic Nintendo Game Boy. You could think of this as being a gaming interface, but its familiarity and its simplicity make it just as useful in physical computing, in the control of robotic instruments, or even as a rudimentary musical instrument.

Figure 16-3. The Liquidware InputShield

One of the interesting transformations that has been going on in the Arduino community is the move toward more and more sophisticated devices and configurations. When the Arduino, or its ancestor the Wiring board, were first introduced, the difference between an embedded device and a laptop computer was much more pronounced than it is today. As users in the real world have become more familiar with small computers, the notion of what an embedded device can be and what an embedded computing platform can do has changed greatly. Liquidware is one group that is pushing this the furthest, creating peripheral devices for the Arduino team that let you create devices as complex as some PDA devices while still using the Arduino environment. There are other companies like Libelium and Adafruit that are pursuing similar projects as well.

Reading data from the InputShield is quite easy. Depending on the mode, you simply need to read the pin mapped to the joystick and the button. Table 16-1 shows how these are mapped.

Table 16-1. Arduino pin to mode map

Arduino pin	Mode A	Mode B
4	Button B	None
5	Button A	None
6	Joystick button	None
7	Vibration enable	None
8	None	Button B
9	None	Button A
10	None	Joystick button
11	None	Vibration enable
Analog 0	Joystick lateral movement	None
Analog 1	Joystick vertical movement	None
Analog 2	None	Joystick lateral movement
Analog 3	None	Joystick vertical movement

The different modes let you use the remaining pins on the Arduino for different purposes, such as controlling a servo motor, reading GPS data, sending data to an LCD display, and so on.

A small vibration motor is attached to the bottom of the shield. The vibration motor will vibrate when pin 7 (mode A) or pin 11 (mode B) is grounded. This lets you send the user physical feedback quickly and easily. The buttons output digital signals, and the joystick output uses varying voltage based on the rotation angle in both lateral and vertical directions. For instance, in mode A, the following code checks and prints both lateral and vertical values of the joystick:

```
void loop() {
    unsigned int joyLatValue;
    unsigned int joyVertValue;
    joyLatValue  = analogRead(0);
    joyVertValue  = analogRead(1);
    Serial.println(joyLatValue);
    Serial.println(joyVertValue);
}
```

Now you can connect the InputShield to an Arduino and use the serial port to create a video game control. If you're interested in using a regular video game controller using a USB port, then you can look into a library like the `proCONTROLL` that allows you to communicate with joysticks and video game controllers in a Processing application. What might be more interesting is the easy control of robotics and physical computing that the Arduino affords you. Using the InputShield to control multiple servo motors, for instance, is a excellent way to get fine-grained and natural control over some simple robotics. To save a bit on space, the discussion on controlling multiple servos will wait until later in this chapter, but rest assured that the code there can very easily be used with the InputShield as well.

Understanding Touch

The cultural moment of the touchscreen has most certainly arrived. By the time you read this book, it's quite likely that many consumer laptops will be equipped with sophisticated touchscreens. The wild popularity of the iPhone, and indeed of its interface, have taken the touchscreen from a sophisticated tool seen in interface research labs and expensive trade demos to the primary interface for a tool that enables interaction in a few different areas in contemporary society. It is soon to become as ubiquitous, as commonplace, and as well understood as the keyboard, the mouse, or even the dial. There is, however, quite a difference between these common input devices and the touchscreen. The first factor is the multimodal data: we can evaluate the location of the touch, the pressure on the screen, the relative distance of other touches in a multitouch environment, and, over time, the gestures consisting of one or many points.

There are also a great number of simple and common tasks that are difficult to do on a touchscreen. Typing, for instance, is an interesting challenge that a great number of different approaches and devices have attempted to simplify and standardize from the use of swipe-based typing systems to projected keyboards. There are numerous approaches and design philosophies that are tackling this problem. Another problem is the language of touchscreen gestures. Since one of the strengths of the touchscreen is the ability to use a gesture expressively, using two fingers to zoom in, for instance, the standardization of these gestures and the way that they are used becomes important to consider. A user will expect to use certain gestural behaviors across all systems and applications; ensuring consistency is an important part of the designers' job.

Exploring Open Source Touch Hardware

The idea of open source software is a fairly simple one: When you allow an end user to see and modify the source code, your project is *open source*. In other words, the source can be read and reused with some reservations. Open source hardware is a little different. In one sense, hardware can't be truly free and open source because, somewhere, someone along the line will need to pay for materials and hardware. However, the schematics and the source code that runs on the hardware can be open source and available. This is the philosophy of two interesting touchscreen projects that are currently providing schematics and hardware, selling kits, and sharing source code openly. The first project is Nort_/D, a touchscreen project built on and around openFrameworks. The other is the Liquidware TouchScreen built on the Arduino platform.

Nort_/D

Nort_/D is a design collective that creates hardware and software for multitouch screens. The collective provides both instructions for creating touchscreens and also kits that include screens, cameras, and supports. Its aim is to make multitouch readily available in an open source fashion. Over the past few years, Nort_/D has created a few different multitouch systems: the CUBIT multitouch system, the TouchKit, and Tactility, which uses LCD screens. The TouchKit includes a Nort_/D Multitouch Screen, Nort_/D's Multitouch Software (computer vision, calibration, API), and a camera (calibrated for use with the TouchKit).

The `ofxTouchApp` class is the main superclass for ofxTouch applications. It is a substitution for the typical openFrameworks `ofBaseApp` from which all applications usually inherit. The `ofxTouchApp` class adds to `ofBaseApp` multitouch-specific functionality. You'll recall that all the event handlers in the `ofxTouchApp` class are sent a reference to this object to notify the application of the position, side, and order of the finger touching the surface.

In the `vector<ofxTouchFinger> fingers` vector, `fingers` stores all the blobs, which are areas detected in the video of your application that are moving together, that are detected and tracked on the touchscreen. They are stored as the `ofxTouchFinger` type, which is a simple class that the `ofxTouchApp` uses to store information about the location and movement of a finger on the touchscreen.

The class provides these event handlers to notify your application that a finger event has happened and pass the reference to the `ofxTouchFinger` instance representing the blob that triggered the event. The names are quite self-explanatory:

- `fingerDragged( const ofxTouchFinger& )`
- `fingerPressed( const ofxTouchFinger& )`
- `fingerReleased( const ofxTouchFinger& )`

The `ofxTouchApp` class also has some convenience methods that make it easier for you to test your application with stored video files:

`setSimulationMode( bool value )`
: Sets whether the application is going to be live (using a touchscreen) or a simulation using the mouse instead of finger touches.

`setVideoPlayerMode( bool value )`
: Passes whether you want the application to be running from a camera or from a test video file that will be used for setup and demo purposes.

`setVideoPlayerFile( string path )`
: Passes a file that you would like the `ofxTouchApp` class to use in place of live finger detection. This is excellent for testing or configuring your touchscreen when you first set it up.

The other key component of the `ofxTouch` add-on is `ofxTouchFinger`, which is used to store information about finger events. These are the two methods useful for keeping track of the fingers:

`int id()`
: Every finger object has an ID assigned to it.

`int initialOrder()`
: This is the order of the touches.

To determine information about the finger, there are four integer values about the finger (`x`, `y`, `width`, and `height`) and a float value (`radius`).

Your application's *.h* file will look something like this:

```
class testApp : public ofxTouchApp {
```

Note that instead of extending the `ofBaseApp` class, the application extends the `ofxTouchApp` class:

```
    public:
        bool bDragging;
        void setup();
        void update();
        void draw();
        void keyPressed  (int key);
        void keyReleased  (int key);
        void mouseMoved(int x, int y );
        void mouseDragged(int x, int y, int button);
        void mousePressed(int x, int y, int button);
        void mouseReleased();
        void fingerDragged( const ofxTouchFinger& finger );
        void fingerPressed( const ofxTouchFinger& finger );
        void fingerReleased( const ofxTouchFinger& finger );

};
```

In the *main.cpp* file of an `ofxTouchApp`, instead of seeing the standard call to `ofRunApp()`, you'll see a call to `ofxTouchAppRelay()` wrapped inside the `ofRunApp()` call:

```
ofRunApp(new ofxTouchAppRelay(new testApp));
```

If you take a look at the *ofxTouchApp.h* file, you'll see the following properties. If you read Chapter 14, the first two objects should be familiar to you.

`ofxTouchContourFinder contourFinder`
: This is an instance of the `ofxOpenCV contourFinder` that you read about in Chapter 14.

`ofxTouchBlobTracker blobTracker`
: This is an instance of the blob tracker that you looked at Chapter 14.

`ofxTouchGraphicsWarp graphicsWarp`
: This is an instance of the `ofxTouchGraphicsWarp` class that is used to calibrate the camera and screen. This is a more advanced setting that allows you to configure the application for specific camera types and screen configurations. There isn't space to cover it in full, but in short, combined with the `ofxTouchVisionWarp` class to set the properties of your particular screen and camera, you can set up the camera and project in different configurations. By using `ofxTouchGraphicsWarp` instead of matching the camera and project perfectly to ensure that the projector covers the entire TouchKit, you can adjust graphics warp so the projector and the TouchKit screen match.

Moving on, a very simple application might look like the following:

```
void fingerApp::setup() {
    setSetupMode( true );
    cwidth = 800;
    cheight = 800;

    //setSimulationMode( true );  //uncomment this to use mouse simulation
    setVideoPlayerMode( true );   //comment this out to run live from cam
    setVideoPlayerFile("touchkit-fingers.mov");
    bDragging = false;
    ofEnableAlphaBlending();
}


void fingerApp::update(){
    // nothing here
}


void fingerApp::draw() {
```

Since the `fingers` vector stores all the detected touch fingers, you can use this code to display the fingers:

```
    for( int i=0; i<fingers.size(); i++ ) {
        ofSetColor( 255, 255, 255);
        ofCircle( fingers[i].x, fingers[i].y, 4*fingers[i].radius );
```

```
            ofSetColor( 255, 0, 0 );
            ofCircle( fingers[i].x, fingers[i].y, 3*fingers[i].radius );
            ofSetColor( 255, 255, 255);
            ofCircle( fingers[i].x, fingers[i].y, 2*fingers[i].radius );
            ofSetColor( 255, 0, 0);
            ofCircle( fingers[i].x, fingers[i].y, 1*fingers[i].radius );
            ofSetColor( 255, 255, 255 );
            ofCircle( fingers[i].x, fingers[i].y, 0.5*fingers[i].radius );

        }
    }
```

This will create a simple bull's-eye at the location of each detected finger touch. There are some other excellent examples for download from the Nort_/D site at *http://touchkit.nortd.com*. For information on their other projects, look at nortd.com.

Liquidware TouchShield

Another much smaller touchscreen device that you might be interested in exploring is the TouchShield developed by Liquidware (Figure 16-4). It plugs into the Arduino and lets you use small touchscreen devices both to send information to a listening laptop or desktop computer or to simply create a small device that relies entirely on the Arduino for its processing power. The TouchShield greatly simplifies working with resistive touch sensing and drawing.

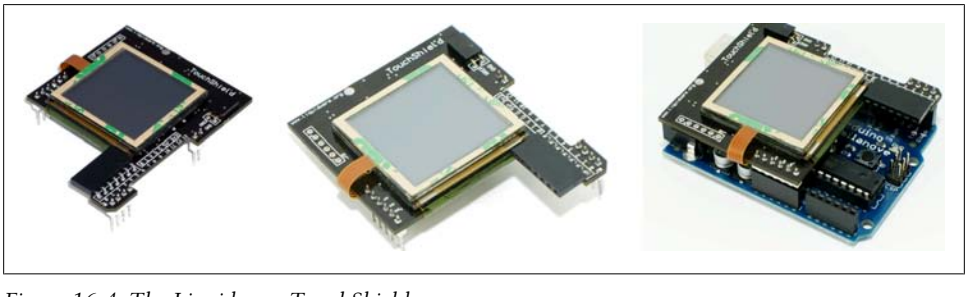


Figure 16-4. The Liquidware TouchShield

To connect the TouchShield onto an Arduino Diecimila or Duemilanove, you simply mount it as shown previously in Figure 16-2 on the right. The TouchShield does quite different things than the standard Arduino components. In addition to handling touch data, the TouchShield also draws to its screen. The Liquidware team took a logical direction in how to best providing methods and a programming interface for the developer to draw to the screen: they emulated the Processing language and application structure.

The very simplest TouchShield application looks much like a Processing application. For instance, to draw a rectangle to the screen of the TouchShield, your application would look like this:

```
COLOR red = {255,0,0};
COLOR blue = {0,0,255};

void setup() {
    rect(20,20,80,80, red, blue);
}
void loop(){}
```

To upload your code to the TouchShield, you'll need to use a slightly modified version of the Arduino IDE that the Liquidware team has put together and that is available for download on the Liquidware site. The only modifications that have been made are to allow for the uploading of code to the TouchShield. When you have an Arduino application that communicates with the TouchShield, you are in essence running two applications at the same time: one on the TouchShield and one on the Arduino. You could think of this metaphorically as being like the relationship between code running on your central processor unit communicating with code running on your graphics processor unit.

While the code for the TouchShield is quite similar to Processing, it is not actually Processing, and so it has a few slightly different methods:

`beginCanvas()`
: This begins the drawing, allowing the TouchShield to accept drawing functions from the Arduino. It starts the serial connection that the TouchShield uses to read data from an Arduino at 9600 baud (the equivalent of calling `Serial.begin(9600)` in the Arduino) and sets up the TouchShield to receive drawing commands from the Arduino. These can be things like the following:

  ```
  |RECT20208080
  ```

  Note the pipe (|) at the beginning of the command. You would send this like so:

  ```
  Serial.write("|RECT20208080");
  ```

  This will draw a rectangle at 20, 20, with a width and height both of 80 pixels. Almost all the drawing commands for the TouchShield can be called in this way via the Serial port. These commands are documented on the Liquidware site in greater detail.

`delay()`
: Like the Arduino version of this method, this delays for the specified millisecond amount of time passed to it.

`random()`
: This generates a random number (long number).

To read the input locations and information about the user touching the screen, the TouchShield library defines the following methods and variables:

`gettouch()`
: This checks the touchscreen for any registered touches. The OLED screen on the TouchShield is resistive, so a user will need to be pushing on it a little bit harder than they would need to with a capacitive screen. Once the `gettouch()` method is called, the `mouseX` and `mouseY` variables will have values assigned to them for that loop. The best way to do this is to call `gettouch()` in the `loop()` method of your application and then do something with the newly updated position values:

```
void loop(){
    gettouch();
    ellipse(mouseX, mouseY, 20,20);
}
```

`mouseX`
: This stores the X location of the touch when the `gettouch()` method is called.

`mouseY`
: This stores the Y location of the touch when the `gettouch()` method is called.

The TouchShield also defines quite a few different methods to draw on the canvas:

`ellipse()`
: Draws an ellipse with the first two parameters specifying the location of the ellipse and the second two determining the width and height, like so: `ellipse(x, y, width, height)`.

`line()`
: Draws a line from point to point, like so: `line(x1, y1, x2, y2)`.

`point()`
: Draws a single pixel.

`rect()`
: Draws a rectangle with a specified beginning point, width, and height, like so: `rect(x, y, width, height)`.

`triangle()`
: Draws a triangle with three specified points, like so: `triangle(x1, y1, x2, y2, x3, y3)`.

`quad()`
: Draws a polygon with four specified lines from four specified points, like so: `quad(x1, y1, x2, y2, x3, y3, x4, y4)`.

To set the color properties and line properties when you're drawing, you can use the following methods (note that all the colors are passed as three values, red, green, and blue with values between 0–255):

`background()`
: Colors the whole screen with the color value passed to it. This also overwrites all the graphics on the screen.

`fill()`
: Sets the color used to fill the inside of shapes. This method takes a color value.

`noFill()`
: Allows a shape to be drawn with no fill color.

`stroke()`
: Stores the color of a shape's outlines or of text. This method takes a color value.

`noStroke()`
: Allows a shape to be drawn without an outline around it, just like the Processing application.

`strokeWeight()`
: Changes the outline thickness of an ellipse or rectangle in pixels.

One of the great strengths of open source software is that it allows developers to alter some of the internals of the program or add new features to support different functionality. Since the TouchShield is in a sense like an Arduino controller, in that it has its own processor that the Arduino IDE must compile and load code onto, you'll need a version of the Arduino IDE that knows how to create and upload code from the TouchShield. Liquidware created a version of the IDE that adds this new board into the available configurations and didn't change anything else, so you can use the TouchShield as well as all your other Arduino boards with this altered Arduino IDE. Those altered IDEs are available on the Liquidware site, *http://liquidware.com*. To work with the TouchShield, you'll need to download and install the appropriate one for your operating system. Once you've done that, you'll see the TouchShield appear in your Arduino IDE (Figure 16-5).

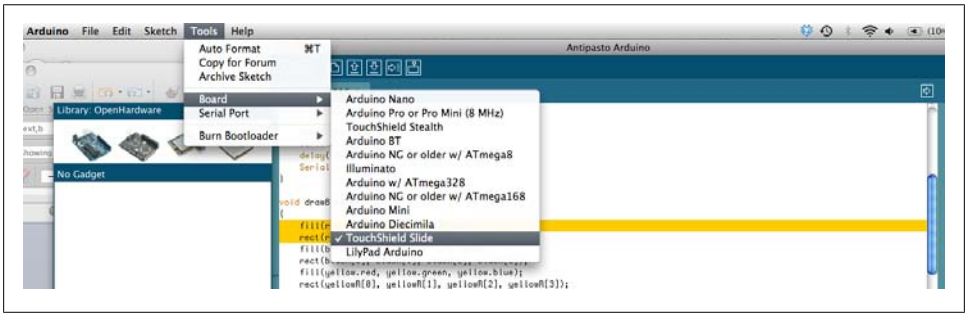


Figure 16-5. Selecting the TouchShield in the modified Arduino IDE

In the Board menu, you'll notice a couple new boards, the TouchShield Slide and TouchShield Stealth. There are currently two different touch-enabled shields that Liquidware manufacturers; one is the TouchShield Slide and the other is the TouchShield Stealth. The workflow for uploading code to a TouchShield attached to a Duemilanove would look like this:

1. Select Tools→Boards→TouchShield Slide.
2. Write the TouchShield code.
3. Touch the programming button on the TouchShield.
4. Compile and upload the application code to the TouchShield from the Arduino IDE by hitting the Upload button.
5. Select Tools→Boards→Arduino Duemilanove (or the name of your board).
6. Write Arduino code.
7. Compile and upload the application code to Arduino controller from the Arduino IDE by hitting the Upload button.

That's all there is to it. Let's take a look at a couple of examples.

Drawing to the TouchShield Screen

The following example shows how to draw different shapes and characters to the screen:

```
#define SHAPE_SQUARE  0
#define SHAPE_CIRCLE  1
#define  SHAPE_X   2
#define  SHAPE_TRI   3

int nextShape = SHAPE_SQUARE;

int pMouseX;
int pMouseY;
char xposStr[4];
char yposStr[4];

void setup() {
    pMouseX = 0;
    pMouseY = 0;
}

void loop()
{
```

The `getTouch()` method polls the screen to see whether any touches have been registered. When you want to detect touches on the screen, you'll want to do this each time the `loop()` is called:

```
    gettouch();

    if(pMouseX != mouseX || pMouseY != mouseY)
        touched();   // screen touched so draw new shape

    delay(50);

} //end loop
```

```
void touched()

{
    nextShape++;
    if(nextShape > 3) { // shape 3 is the last shape
        nextShape = 0;
    }
```

Just as in a Processing application, this clears the stage by overwriting everything:

```
    background(0);
```

Now the location of the touch is stored for comparison to the next time the `loop()` method is called:

```
    pMouseX = mouseX;
    pMouseY = mouseY;
```

The `text()` method writes characters to the screen, the first parameter are the characters to be written, and the next two are the x and y locations on the screen:

```
    text(mouseX, mouseX, mouseY+20);
    text(mouseY, mouseX+30, mouseY+20);
```

Depending on the value of the `nextShape` variable, a different shape is drawn to the stage:

```
    switch(nextShape)
    {
    case SHAPE_SQUARE:
        fill(0, 255, 0);
        rect(mouseX-20, mouseY-20, 30, 30);
        break;

    case SHAPE_CIRCLE:
        fill(0, 0, 255);
        ellipse(mouseX-10, mouseY-10, 15, 15);
        break;

    case SHAPE_X:
        stroke(255, 255, 255);
        line(mouseX - 10, mouseY, mouseX+10, mouseY);
        line(mouseX, mouseY - 10, mouseX, mouseY+10 );
        break;
    case SHAPE_TRI:
        fill(255, 255, 0);
        triangle(mouseX+10, mouseY-10, mouseX-10, mouseY+10,
              mouseX+20, mouseY+10);
        break;
    }
}
```

The application will look something like Figure 16-6.

Now that you've seen how to do simple drawing operations with the TouchShield, you can use the touch information and send it to the Arduino to control other objects. The following example allows the user to select the color of the circle they are drawing, draw a circle, and send those values to the Arduino controller. From there, the Arduino

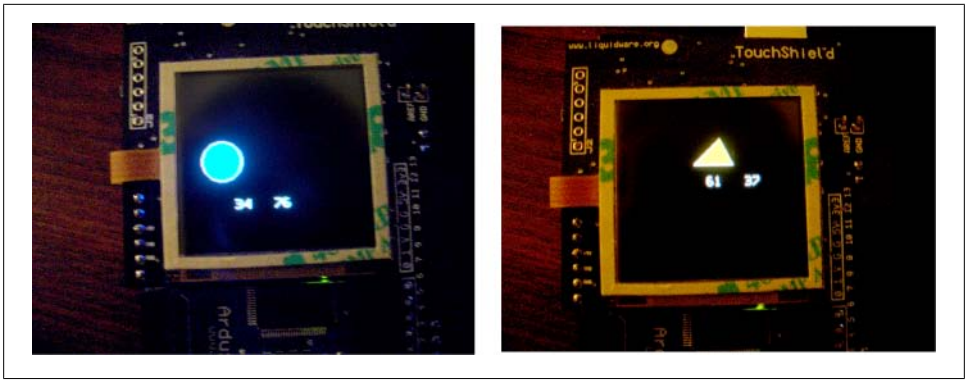


Figure 16-6. Drawing a circle and a triangle to the TouchShield

controller can, for example, use the x and y positions to set the rotation of a servo, relay those values to another listening controller, or, as shown in the next example, position a light.

Controlling Servos Through the TouchShield

In this example, though, those values will be used to position two servo motors that will shine a three-color LED with the color selected by the user. Before putting this code into the Arduino IDE, make sure that you have the TouchShield board selected, or the code won't compile because the compiler will not have access to the right libraries:

```
COLOR curr;
```

Notice that the color is stored as a `COLOR` type. This is a `struct` that is defined in the TouchShield libraries and has `red`, `green`, and `blue` integer properties that can be read and written:

```
int redR[] = { 0, 0, 20, 20 };   // rectangle coordinates
int blueR[] = { 0, 20, 20, 40 };
int yellowR[] = { 0, 40, 20, 60 };
int greenR[] = { 0, 60, 20, 80 };
int whiteR[] = { 0, 80, 20, 100 };
int blackR[] = { 0, 100, 20, 120 };

COLOR red = {255,0,0};
COLOR green = {0,255,0};
COLOR blue = {0,0,255};
COLOR yellow = {255,255,0};

boolean changedColor = false;

void setup() {

    drawBackground();
    Serial.begin(9600);
    delay(3000);
```

```
        Serial.print('U');
    }
```

The `drawBackground()` method draws six boxes on the left of the TouchShield that the user can select. These will be used to set the color value of the color LED attached to the two servo motors:

```
    void drawBackground()
    {
```

Draw the first rectangle in the top left of the screen, first setting the fill using the `fill()` method. The `fill()` method takes the different values of the predefined color values included in the TouchShield API:

```
        fill(red.red, red.green, red.blue);
        rect(redR[0], redR[1], redR[2], redR[3]);
```

Now draw the other rectangles for blue, yellow, green, and white:

```
        fill(blue.red, blue.green, blue.blue);
        rect(blueR[0], blueR[1], blueR[2], blueR[3]);
        fill(yellow.red, yellow.green, yellow.blue);
        rect(yellowR[0], yellowR[1], yellowR[2], yellowR[3]);
        fill(green.red, green.green, green.blue);
        rect(greenR[0], greenR[1], greenR[2], greenR[3]);
        fill(white.red, white.green, white.blue);
        rect(whiteR[0],whiteR[1], whiteR[2], whiteR[3]);
    }

    void loop() {
```

The refresh rate on the TouchShield is quite a bit slower than a laptop computer. If you call the `background()` method on a loop and then do a relatively expensive drawing operation like the `drawBackground()`, then you'll see the screen begin to flicker:

```
    gettouch();
        check_touch();

        Serial.print(mouseX);
        Serial.print(mouseY);
```

If the color has changed, then send the values to over the hardware Serial port to the Arduino controller:

```
        if(changedColor) {
            Serial.print(curr.red);
            Serial.print(curr.green);
            Serial.print(curr.blue);
        }
    }

    void check_touch(){
        changedColor = true;
        // if the mouse is greater than 20, it's not on any of the rectangles
        if(mouseX > redR[0]) {
            return;
```

```
            changedColor = false;
        }
        if(mouseY < redR[2]) {
            curr = red;
        }
        else if(mouseY > blueR[2] && mouseY < blueR[4]) {
            curr = blue;
        }
        else if(mouseY > yellowR[2] && mouseY < yellowR[4]) {
            curr = yellow;
        }
        else if(mouseY > greenR[2] && mouseY < greenR[4]) {
            curr = green;
        }
        else if(mouseY > whiteR[2] && mouseY < whiteR[4]) {
            curr = white;
        }
        else {
            changedColor = false;
        }
    }
```

Setting Up Communication Between Arduino and TouchShield

In this example, you'll see how the Arduino board and the TouchShield can communicate. The principle idea in this example is to use two servos together to create a hemispheric range for a light, that is, to enable an LED to cover an area that more or less would look like the top half of a sphere. The way you do that is by combining two servos and using one to provide 180 degrees of motion on the x-axis and the other to provide 180 degrees of motion on the y-axis. There are quite complex ways to do this, but there are simple ones that work almost as well. You can simply attach one servo to the other and then attach the LED to the second servo. This provides you with the range of motion to point the light. Using a "super-bright LED" means that you can attach a small object to a small servo and still create a good amount of light.

Figure 16-7 shows a diagram of how to connect the two servos together and how they would be wired to the Arduino.

Now, the Arduino code that receives the x and y values that will be used to set the two servos and the color values to set the LED. This code is for the Arduino, so don't forget to make sure that you have the correct board selected in your Arduino IDE, or the code may not compile:

```
#include <Servo.h>

// create servo object to control a servo
Servo xServo;
Servo yServo;
int redPin = 5;
int greenPin = 6;
int bluePin = 11;
```

```
// variable to store the servo position
int xpos = 0;
int ypos = 0;

void setup()
{
```

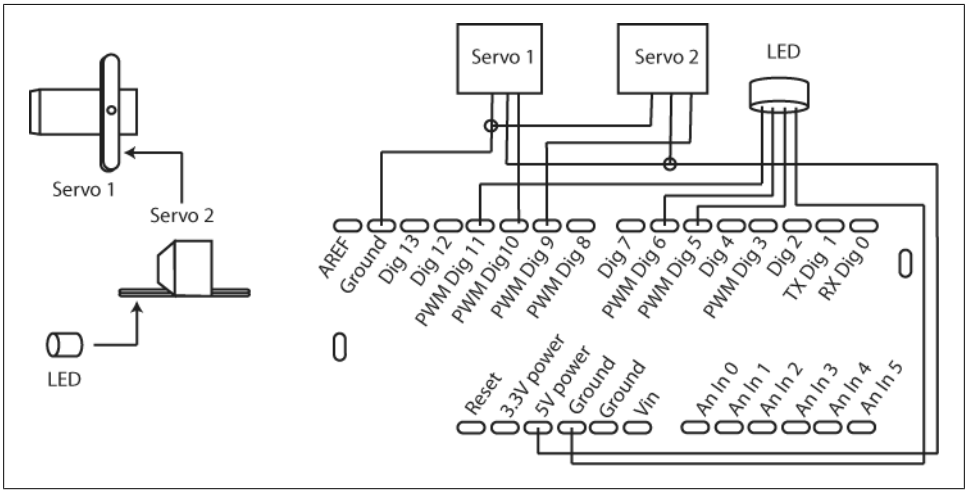


Figure 16-7. Connecting the servos and LED to the Arduino

Here you start up the communication between the Arduino and the two servos by calling the `attach()` method of the `Servo` instance:

```
    xServo.attach(9);
    yServo.attach(10);

    Serial.begin(9600);
}

void loop()
{
    int tmpx, tmpy, tmpred, tmpblue, tmpgreen = 0;
```

Get all the available bytes from the TouchShield that have been sent over the Serial port. I've set my hardware up so this data is being sent from the TouchShield, but you could also use the TouchShield and have the data be sent from a Processing or oF application with which the user is interacting. The positions and colors will be stored in the five integers:

```
if(Serial.available() > 4) {
    tmpx = Serial.read();
    tmpy = Serial.read();
    tmpred = Serial.read();
    tmpblue = Serial.read();
    tmpgreen = Serial.read();
```

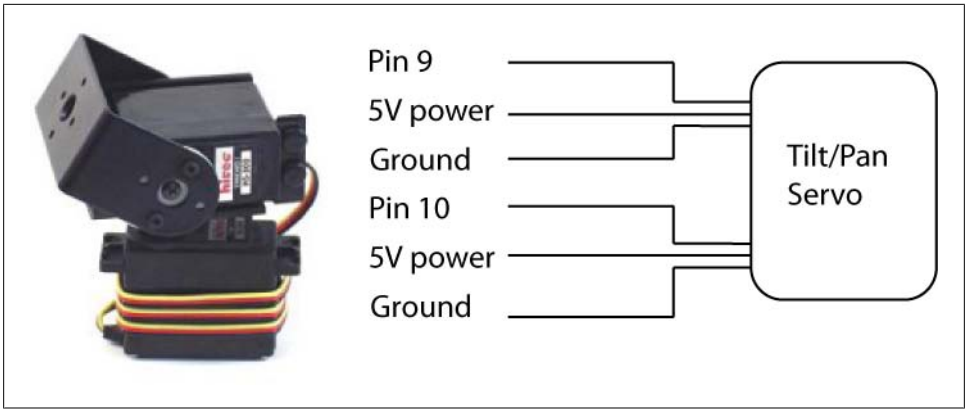


Figure 16-8. The Trossen pan-tilt servo

After five values are received, set each respective color values, red, green, and blue, of the three color LED lights:

```
            analogWrite(redPin, int(tmpred));
            analogWrite(greenPin, int(tmpgreen));
            analogWrite(bluePin, int(tmpblue));
```

Now, set the location of each servo. Since the locations are being sent as integers, they'll need to be cut down into smaller values to set the rotation of the servo.

```
            xpos = map(tmpx, 0,255, 0,180);
            ypos = map(tmpy,0,255,0,180);
            xServo.write(xpos);
            yServo.write(ypos);
            delay(15);

        }
    }
```

You could use this same technique to position a camera, a microphone, or many other kinds of devices. The pairing of multiple servos is one of the basics of complex mechanical motion and is a very useful thing to look into. You might check out *Robot Building for Beginners* by David Cook (Apress). It is a good introduction to mechanics and robotics that is aimed at newcomers.

Another approach to this is to use a pan and tilt servo unit like the one shown in Figure 16-8, made by Trossen Robotics. It's actually just two servo motors with a stronger mounting, which means you can attach larger lights or cameras to the servo.

The TouchShield is a wonderful tool for creating small touch-enabled applications. One thing to consider when working with the TouchShield is the relatively slow refresh rate of the drawing on the TouchShield. The code used with TouchShield is based on Processing, but the hardware will not be like a laptop or desktop computer. So, you'll need to consider what drawing functions to call and when to call them to ensure that you don't overtax the hardware. Another issue to consider is the power that the

TouchShield requires. If you're going to use the TouchShield as a mobile device, then you might want to consider using something like the Lithium Backpack that provides a few extra hours of battery power.

Communicating Using OSC

OpenSoundControl (OSC) was originally designed as a protocol for communication between computers, sound synthesizers, and any other devices to be used in a networked setting. It allows any device that can send and receive OSC messages to share music performance data in real time over a network. In this way it's similar to MIDI but is designed expressly for network communications, whereas MIDI was designed more as a low-level way of communicating between devices that couldn't be networked. This is largely because MIDI was developed in 1983 before networks were fast enough to consider sending live, real-time audio and video data over them. Since it's a network protocol, OSC allows musical instruments, controllers, and multimedia devices to communicate on any network, including over the Internet.

You may remember the UDP protocol for network communication that we mentioned earlier. OSC is communication over UDP just as sending and receiving web pages (HTTP) is communication over TCP. Now, there's an important caveat here: nothing says OSC has to be over UDP, that's just the way most people use it. You might want to make your OSC application communicate over TCP, and that's perfectly valid as well. The advantages of UDP being light, fast, and easy to parse all make it extremely valuable for real-time audio communication. OSC enables real-time interactions between almost any kind of device that can be connected to a network and also allows you a lot of flexibility in terms of the data you can send over the wire. This means your applications or devices can communicate with each other at a high level or a low level, however you see fit for your application.

What can you use it for? Since OSC is a data protocol, any application that can send OSC messages can use it to communicate with any other application that can understand OSC messages. You can use OSC to communicate between applications like oF and Processing. You can also use OSC to communicate between Processing or oF and another platform like the Max/MSP, PureData, or Supercollider. Since OSC is really just communication using the UDP protocol, by using an Ethernet shield an Arduino can send and receive any OSC messages as well. So, what does an OSC message look like? Usually something like this:

```
'/pitch', '122'
```

That looks a little strange, but it's actually fairly straightforward. The message has a title, in this case `pitch`, and a value, in this case `122`. A message can have any number of values attached to it, but generally the messages are quite short. They send things like a note, a change in pitch, a mouse location, a camera action, movement detection,

and so on. In practice, OSC isn't all that well suited for sending massive amounts of data at a time; it's far better at sending lots of small messages quickly.

For working with OSC in oF, you can use the ofxOSC library, created by Damian Stewart and available from *http://addons.openframeworks.cc*. It is built on top another OSC library, called *oscpack*, that was built by Ross Bencina. It consists of three main classes that should have a familiar breakdown:

`ofxOSCSender`
: This is the object that dispatches messages. It doesn't need to accept connections from a client. Instead, it simply dispatches the messages to anything listening on the port that it uses.

`ofxOSCReceiver`
: This is the object that receives and parses messages; it listens on a particular port number and then can parse messages by name and get their values.

`ofxOSCMessage`
: This represents the message. OSC can have int, String, or Float values. The `OSCMessage` class provides methods for adding all of these to a message.

For using OSC in Processing, check out oscP5, an OSC implementation for the programming environment processing developed by the prolific Andreas Schlegel. If you already have been working with oscP5 in the past, please note that you will have to import `netP5.*` in addition to importing `oscP5.*`:

`OscP5`
: This is the main class that enables you to initialize the oscP5 library and set up a server that can send and receive messages. This class handles both sending and receiving messages.

`OscProperties`
: This is used to start oscP5 with more specific settings, and it is passed to the OscP5 object in the constructor when initializing a new OscP5 object.

`OSCArgument`
: This represents the value of the message sent and allows you to read the message back in several different formats: int, boolean, float, String, or char.

`OscMessage`
: Like the `ofxOSCMessage` class, this contains an address, a type, and multiple values.

Though space constraints don't allow us to delve further into OSC, it might be worth checking out the netP5 and the oscP5 libraries if you're looking to share real-time data across several different machines. One immediately practical application for this protocol is communication with the Wii Remote (unofficially nicknamed Wiimote), which you'll learn how to do in the next section using the ofxOSC library.

Using the Wiimote

By now most of you will have seen the Nintendo WiiMote controller, the small accelerometer-driven controller for the Wii system. The Wii Remote is the primary controller for Nintendo's Wii console. A main feature of the Wii Remote is its motion-sensing capability, which allows the user to interact with and manipulate items on the screen via movement and point toward objects through the use of accelerometer and optical sensor technology. There are two different controllers, the Wii Remote and the Wii Nunchuk, that attach to the Wii Remote to allow for two-handed control.

Using the Wii Nunchuck in Arduino

You can communicate with the Wii in two ways, either by plugging it into an Arduino using the Wii stick developed by Tod Kurt of Todbot or by using the Bluetooth connection on the Wii Remote. You can also connect the Nunchuck directly to the Arduino by cutting off the connector and using the wiring, as shown in Figure 16-9.

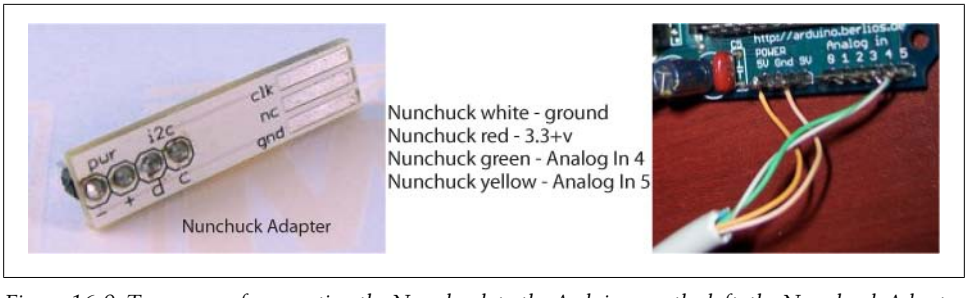


Figure 16-9. Two ways of connecting the Nunchuck to the Arduino: on the left, the Nunchuck Adapter and, on the right, the Nunchuk with the connector removed and wires connected directly to the Arduino

The following Arduino code uses the values from the Nunchuck to control a servo and also sends the accelerometer data from the Nunchuck to a Processing application:

```
#include <Wire.h>
#include <Servo.h>

// create servo object to control a servo
Servo xServo;
Servo yServo;

int angleX = 90;    // angle to move the X servo
int angleY = 90;    // angle for the Y servo

int refreshTime = 20;  // the time in millisecs between updates
int loop_cnt = 0;

int joy_x_axis;
int joy_y_axis;
int accel_x_axis;
```

```
int accel_y_axis;
int accel_z_axis;

int z_button;
int c_button;

void setup()
{
    Serial.begin(9600);
    xServo.attach(9);
    yServo.attach(10);
    nunchuck_init(); // send the initialization handshake to nunchuck
}

void loop()
{
    checkNunchuck();
    xServo.write(angleX);
    xServo.write(angleY);
    if( nunchuck_zbutton() ){  // light the LED if z button is pressed
        Serial.print(angleX,DEC);
        Serial.print(":");
        Serial.print(angleX,DEC);
        Serial.print('\0');
    }
    delay(refreshTime);        // this is here to give a known time per loop
}

void checkNunchuck()
{
    if( loop_cnt > 5) {  // loop every 20msec, this is every 100msec

        nunchuck_get_data();
        nunchuck_print_data();

        angleX = map(accel_x_axis,70,185,0,180); // nunchuck range is ~70 - ~185
        angleY = map(accel_y_axis,70,185,0,180); // Arduino function maps this
                                                 // to angle 0-180
        loop_cnt = 0;  // reset
    }
    loop_cnt++;
}
```

The rest of the code defines the Nunchuck functions to initialize communication between the Arduino and the Nunchuck and to send and receive data from the Nunchuck. These methods all use the Wire library to read the data. Credit for these methods goes to Tod Kurt:

```
static uint8_t nunchuck_buf[6];   // array to store nunchuck data,

// initialize the I2C system, join the I2C bus,
// and tell the nunchuck we're talking to it
void nunchuck_init()
{
    Wire.begin();                    // join i2c bus as master
```

```
    Wire.beginTransmission(0x52);    // transmit to device 0x52
    Wire.send(0x40);        // sends memory address
    Wire.send(0x00);        // sends sent a zero.
    Wire.endTransmission();    // stop transmitting
}

// send a request for data to the nunchuck
// was "send_zero()"
void nunchuck_send_request()
{
    Wire.beginTransmission(0x52);    // transmit to device 0x52
    Wire.send(0x00);        // sends one byte
    Wire.endTransmission();    // stop transmitting
}

// receive data back from the nunchuck,
// returns 1 on successful read. returns 0 on failure
int nunchuck_get_data()
{
    int cnt=0;
    Wire.requestFrom (0x52, 6);    // request data from nunchuck
    while (Wire.available ()) {
```

The Wire library will receive a byte that needs to be converted to an integer, by using the `nunchuck_decode_byte()` method, defined later in this application:

```
        nunchuck_buf[cnt] = nunchuck_decode_byte(Wire.receive());
        cnt++;
    }
    nunchuck_send_request();  // send request for next data payload
    // If we recieved the 6 bytes, then go print them
    if (cnt >= 5) {
        return 1;   // success
    }
    return 0; //failure
}
```

This method prints out the input data received from the Nunchuck:

```
void nunchuck_print_data()
{
    static int i=0;
    joy_x_axis = nunchuck_buf[0];
    joy_y_axis = nunchuck_buf[1];
    accel_x_axis = nunchuck_buf[2];
    accel_y_axis = nunchuck_buf[3];
    accel_z_axis = nunchuck_buf[4];

    z_button = 0;
    c_button = 0;
```

The sixth byte of the data from the Nunchuck, `nunchuck_buf[5]`, contains bits that indicate whether the z or c buttons have been pressed:

```
// so we have to check each bit of byte outbuf[5]
    if ((nunchuck_buf[5] >> 0) & 1)
        z_button = 1;
```

```
    if ((nunchuck_buf[5] >> 1) & 1)
        c_button = 1;
```

The sixth byte also contains the last bits of the accelerometer data for all the axes:

```
if ((nunchuck_buf[5] >> 2) & 1)
        accel_x_axis += 2;
    if ((nunchuck_buf[5] >> 3) & 1)
        accel_x_axis += 1;

    if ((nunchuck_buf[5] >> 4) & 1)
        accel_y_axis += 2;
    if ((nunchuck_buf[5] >> 5) & 1)
        accel_y_axis += 1;

    if ((nunchuck_buf[5] >> 6) & 1)
        accel_z_axis += 2;
    if ((nunchuck_buf[5] >> 7) & 1)
        accel_z_axis += 1;

    i++;
}

// encode data to format that most wiimote drivers except
// only needed if you use one of the regular wiimote drivers
char nunchuck_decode_byte (char x)
{
    x = (x ^ 0x17) + 0x17;
    return x;
}

// returns zbutton state: 1=pressed, 0=notpressed
int nunchuck_zbutton()
{
    return ((nunchuck_buf[5] >> 0) & 1) ? 0 : 1;  // voodoo
}
```

The next step would be to use the progress of the two servo motors in a Processing application. This could be easily extended to update another set of servo motors in a different location using a network, to drive complex graphics, to control sounds or video, or to implement any number of other approaches. For the sake of brevity, in this example the Processing application simply draws current positions of the servo motors:

```
import processing.serial.*;
Serial serial;  // The serial port
int firstAng = 0;
int secAng = 0;
byte[] inbyte = new byte[4];

void setup() {
    size(400, 400);
    String[] arr = Serial.list();
    if(arr.length > 0) {
        serial = new Serial(this, Serial.list()[0], 9600);
    }
```

```
}

void draw() {
    background(122);
    int i = 0;
    boolean changed = false;
    if(serial != null) {
        while (serial.available() > 0) {
            inbyte[i] = byte(serial.read());
            changed = true;
            i++;
        }
        if(changed) {
            firstAng = inbyte[0];
            secAng = inbyte[2];
        }
    }

    ellipse(40, 40, 50, 50);
    line(40, 40, 40+(25*cos(firstAng)), 40+(25*sin(firstAng)));
    ellipse(120, 40, 50, 50);
    line(120, 40, 120+(25*cos(secAng)), 40+(25*sin(secAng)));

}
```

The Nunchuck is a very cool tool, but the Wii Remote is a far more versatile tool because it doesn't require you to have your input device tethered to a laptop. The Wii Remote can also communicate with your laptop or desktop over a Bluetooth connection. Once the remote and your computer are connected, you send OSC messages to an oF or Processing application and use the accelerometer data from the remote in your application.

Many times you'll find yourself piecing together different libraries to create a working solution. In this case, creating a working solution that is truly cross-platform is difficult and involves using lots of libraries. Windows users should look into an application called GlovePIE, OS X users should look into an application called DarwiinOSC, and Linux users can utilize an application called WiiOSC.

Once the OSC messages are sent from the Wii controller to the oF application, you can use the ofxOSC add-on to read the messages and use them for drawing, controlling video, and so on:

```
#ifndef _TEST_APP
#define _TEST_APP

#include "ofMain.h"
#include "ofxOSC.h"

class remoteApp: public ofBaseApp{

public:

    void setup();
```

```
    void update();
    void updateOSC();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y );
    void mouseDragged(int x, int y, int button);
    void mousePressed(int x, int y, int button);
    void mouseReleased();

    ofxOscReceiver    receiver;
```

These are the variables that will be store the values received from the Wii Remote:

```
    float pitch, roll, yaw, accel;
    };

    #endif
```

Now on to the *.cpp* file, where the `ofxOSC` instance will read OSC messages from the application that handles the communication between the Wii Remote and the computer and then draws that data to the screen:

```
    #include "remoteApp.h"
    void remoteApp::setup(){

        receiver.setup( 9000 );
        ofSetVerticalSync(true);

    }
    void remoteApp::update(){
```

The call to the `updateOSC()` method ensures that the OSC listener grabs any new data in the OSC buffer:

```
        updateOSC();
    }

    void remoteApp::updateOSC() {

        // check for waiting messages
        while( receiver.hasWaitingMessages() ) {
```

Get the next message:

```
            ofxOscMessage m;
            receiver.getNextMessage( &m );
```

Check to make sure that the new message is from the Wii Remote. The `getAddress()` call here is using an OS X address. Depending on your operating system your address will look different:

```
            if ( strcmp( m.getAddress(), "/wii/1/accel/pry" ) == 0 ) {

                // verify the type
                if( m.getArgType(0) != OFXOSC_TYPE_FLOAT ) break;
```

```
            if( m.getArgType(1) != OFXOSC_TYPE_FLOAT ) break;
            if( m.getArgType(2) != OFXOSC_TYPE_FLOAT ) break;
            if( m.getArgType(3) != OFXOSC_TYPE_FLOAT ) break;
            // get the new score
            pitch = m.getArgAsFloat( 0 );
            roll = m.getArgAsFloat( 1 );
            yaw = m.getArgAsFloat( 2 );
            accel = m.getArgAsFloat( 3 );
        }
    }
}

void remoteApp::draw(){
```

Draw the background according to the pitch, and then use those values to create the background color of the application:

```
    ofBackground(pitch*255, roll*255, yaw*255);
    ofSetColor(yaw*255,pitch*255,roll*255);
    ofFill();
```

Draw the two circles using the pitch, yaw, and roll values sent from the Wii Remote:

```
    ofEllipse(pitch*ofGetWidth(), yaw*ofGetHeight(), 20, 20);
    ofEllipse(yaw*ofGetWidth(), roll*ofGetHeight(), 10, 10);

}
```

It's important to remember that Wii data is, compared to the touchscreens discussed earlier, quite difficult data to work with; it's erratic, and it's often difficult to determine the exact position of a cursor as one would with a mouse or a pair of potentiometers. For this reason, accelerometer-based controls like the Wii Remote are sometimes not appropriate for building applications that require cursor control. However, there is another way to use the Wii Remote as an infrared pointing device. This is often more appropriate than accelerometer data because it is so much more precise. Accelerometer data can be used to position a cursor as well, but it requires some rather tricky calculations that, although certainly within the realm of possibility, are outside of the scope of this book.

Tracking Wii Remote Positioning in Processing

There's another possibility for working with the Wii Remote that provides other sorts of opportunities to work with interaction: in addition to accelerometers, the Wii Remote can also calibrate its position using an infrared sensor (Figure 16-10). To use this, you need to get four small IR lights and put them in front of the Wii Remote where it will be able to detect them. To create a 360-degree experience, you could use more IR lights and position them in multiple locations around the Wii Remote, though this is substantially more complex.

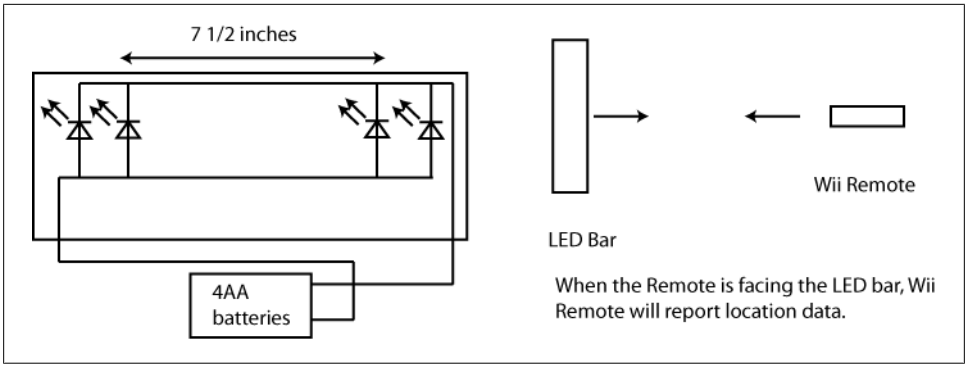


Figure 16-10. Creating an LED array to use with the Nunchuck

The first step in using the Wii Remote as a pointer is to read the IR data sent from the controller. This is sent as 12 floating-point numbers. The oscP5 code calls `ir` when new IR data is received and gives us 12 parameters. Those are four times x, y, and brightness values for each tracked LED. If the brightness value is bigger than 15, no point is recognized. To get usable x and y position values, you will want to pass them through a method that might look like this:

```
float ir[12]
void ir( float f10, float f11,float f12, float f20,float f21,
    float f22, float f30, float f31, float f32, float f40,
    float f41, float f42 ) { // first all the numbers
    ir[0] = f10;
    ir[1] = f11;
    ir[2] = f12;
    ir[3] = f20;
    ir[4] = f21;
    ir[5] = f22;
    ir[6] = f30;
    ir[7] = f31;
    ir[8] = f32;
    ir[9] = f40;
    ir[10] = f41;
    ir[11] = f42;

    points = 0;
```

We want every third value for the x position and every fourth digit for the y position:

```
    for (int i = 0; i < 4; i += 3) {
        if (ir[i+2] < 15) {
            x[points] = 0.5 - ir[i]; // average out the x values
            y[points] = 0.5 - ir[i+1]; // average out the y values
          points++;
        }
    }
}
```

The trickiest part of getting and using the infrared data is correctly parsing out the x and y values. Once you've done that, it's quite easy to use them in an oF or Processing application. Over OSC, the IR data will be passed as 12 floating-point values, and you simply call this to have all the infrared data messages plugged into the `ir` array:

```
osc.plug(this,"ir","/wii/irdata");
```

This simple Processing application draws a rectangle to the screen at the locations of the infrared data:

```
import oscP5;

float ir[12];
OscP5 osc;

void setup() {
    size(800,600);

    // open an udp port for listening to incoming osc messages
       //from darwiinremoteOSC
    osc = new OscP5(this,5600);

    osc.plug(this,"ir","/wii/irdata");
    osc.plug(this,"connected","/wii/connected");
}

void ir(
float f10, float f11,float f12,
float f20,float f21, float f22,
float f30, float f31, float f32,
float f40, float f41, float f42
) {
    ir[0] = f10;
    ir[1] = f11;
    ir[2] = f12;
    ir[3] = f20;
    ir[4] = f21;
    ir[5] = f22;
    ir[6] = f30;
    ir[7] = f31;
    ir[8] = f32;
    ir[9] = f40;
    ir[10] = f41;
    ir[11] = f42;
}

void draw() {
    for(int i=0;i<12;i+=3) {
```

Every third value in the array is the size, and a size of either 0 or 15 indicates that IR point is not available:

```
        if(ir[i+2]<15 && ir[i+2]>0) {
            fill(255, 0, 0);
```

Now we'll draw a rectangle at the point where the Wii Remote detected the light:

```
            rect(ir[i] * width, ir[i+1] * height, 5, 5);
        }
      }
    }
```

What's Next

As you've seen in this chapter, you can approach creating tools from so many different angles that the unifying concept we've explored isn't so much a technical idea but rather a design concept: that of carefully measured feedback and particularly controlled input. There are a number of really great controller technologies and designs out now that might be interesting to play with or use for inspiration.

The Monome project is a series of boards with extremely minimal interfaces that can be used to create music, send MIDI and OSC messages, or do almost any other thing you can imagine for a device with 64 soft buttons and very little extraneous hardware. The Monome creators provide both prebuilt boards in extremely small quantities or boards with instructions to build your own. They also have a small but active community of developers, musicians, artists, and hackers working with the boards and creating new applications. Take a look at *http://monome.org/* for more information.

The Nintendo DS may seem to be an odd platform for a designer or artist to hack at but the DS has a few aspects that might make it interesting to you. It's has a fully rewritable firmware, which means that like the Arduino you can completely rewrite and alter the core software running on the controller. It also has a WiFi connection and uses SD storage. There are also library and toolkits that allow you to program the DS in C++ or C as well as several tools to help you make games without heavy programming.

LadyAda and AdaFruit industries have created a tool called the x0xb0x, which is a board that can act as both a synthesizer and a sequencer. It has a MIDI in, out, and through port; a CV and Gate (1/8-inch jacks); and an headphone, mix-in line, and line-level out port with 1/4-inch jacks; and finally a USB jack.

Another option is the Wacom drawing tablet. As a user interface, it can be instinctive and very intuitive; the pressure sensitivity provides another range of input data as well. It can be inherently very controlled and precise but also playful, making it a great tool for artists, as demonstrated by the Sonic Wire Sculptor by Amit Pitaru. There are a few different libraries that allow you to use Wacom with Processing and oF that you can find by looking on the Processing website in the Libraries section or on the oF website in the forums.

Review

Open source hardware designs are where the schematics and the source code that runs on the hardware schematics can be open sourced and made available.

The Liquidware InputShield allows you plug interface controls that are similar to the classic video game controls into your Arduino controller.

The Liquidware TouchShield is a small Arduino-compatible shield that can be used to capture touches and send and receive messages over a Serial port with an Arduino-compatible board.

To use the TouchShield Slide or Stealth, you'll need to use a modified version of the Arduino IDE.

The TouchShield is programmed using a language influenced by the Processing language, making it easier to write visual code to display on the TouchShield.

OpenSoundControl (OSC) is a protocol for communication between computers and other devices or for between applications. They are very simple messages that usually consist of a key and value pair, like:

```
'/name', 'josh'
```

To work with OSC in oF, you can use the ofxOSC add-on developed by Damian Stewart. To work with OSC in Processing, you can use the oscP5 library developed by Andreas Schlegel.

Another option for interaction is the Wii controllers. The Nunchuck can be plugged in an Arduino, or you can use the Nunchuk Adapter developed by Tod Kurt. The tilt data and the buttons of the Nunchuck can be read in an Arduino application.

The Wii Remote can be paired with a computer over Bluetooth using one of several libraries and can send accelerometer data to a Processing or oF application over OSC.

By creating a grouping of four LED lights and shining them toward the Wiimote, you can create an oF or Processing application that can read the positional data of the Wiimote, where it is pointing in relation to the infrared sensors. This allows you to create a simple mouse-like pointer from the Wiimote.

CHAPTER 17

Spaces and Environments

In this book, we've covered a lot of different means of getting input and creating feedback, but we haven't discussed to a great extent where that feedback and input takes place. The location of the interaction is a very important consideration, because it provides context for that interaction. Using the user's location as a data point is one thing; using the location as an element of the interaction itself is a different proposition. Sculptors, architects, and installation artists have explored the notion of communicating meaning through spatial relationships and architectural elements for many years.

You might want to sense data about an environment: reading the light in a room, listening to sound in a room, detecting motion in a room, or detecting gas with a sensor. This could mean making a smart environment or an enabled environment by allowing multiple devices in an environment to communicate with one another. It could also mean using space itself to communicate the message, as in the case of X10 communication, which sends messages over the electrical lines of a building. It could mean thinking sculpturally and helping design space that reacts to users' commands, like making a room that is reconfigurable from a remote control or that changes based on the time of day, number of people in the room, or heat outside the room. There are so many different conceptions of space both in architectural senses and in aesthetic senses that it's difficult to formulate any coherent meaning for how to approach space. In this chapter, the focus will be on technical strategies for using space in an interactive way that allows engagement.

Using Architecture and Space

One of the primary questions of architecture is "How do we think about space in a way that allows people to live, work, and play better in it?" This is not too dissimilar from the types of questions that you must ask yourself when designing an interface or control for a user, and in fact what many artists and designers have come to realize and explore is that space and structures are, in many ways, tools that shape what we can do and how we can do it.

In traditional architecture, the elements of a building are normally static and fixed at their time of construction. By using interactive design techniques and technologies, you can enhance and enable a building, making it transformable, reactive, and interactive. This is often called *interactive architecture*. The design approach of interactive architecture is different from traditional architecture because interactive architectural objects have behaviors and appearances that are molded together by the architect to create a reactive space and place. Creating successful architectural objects means designing their spatial characteristics and behavior in a way that fully opens up the possibilities of interaction with their environment. You can see hybrid uses of interactive architecture in product displays, trade show demonstrations, wall displays, sculptural pieces, installation pieces in galleries or museums that use multimedia equipment, and interior architecture.

Many techniques and ideas discussed elsewhere in this book are very relevant to how you might approach using space in your application. For instance, machinery can subtly change the nature of a room; for example, with a strong enough motor you could shift the location of a façade or wall, or with a very small electric motor you could raise or lower curtains. Artists and architects like Usman Haque, Jason and Zena Bruges, and Kas Oosterhuis, among others, have explored the overlap and potential of interactive and transformative architecture. Sound shapes a space in subtle ways as well: echoes, soft hums, and sound reverberations all change the nature of how we perceive a space. Lighting is of course vital, too, and a great number of both artists and architects are doing very interesting things with how lighting can transform spaces and objects; James Turrell, Pablo Valbuena, and H. C. Gilje have all used lighting and carefully controlled perceptual effects to sculpt spaces and re-create both rooms and outdoor areas.

Recent advances in materials for construction and fabrications mean that creating reactive environments is becoming easier and more affordable. Light-conductive materials and LEDs make lighting easier and more flexible, three-dimensional printing technologies mean that you can print plastic prototypes cheaply, and wireless controllers like the XBee mean that you can easily send information from one sensor to another without needing to run wires through a space. Interaction design is very rapidly becoming a valid way to approach design in the spaces in which people act.

Some excellent books are worth checking out if you're interested in some of the thought around architecture and computing or interactive architecture. *Digital Ground* by Malcolm McCullough (MIT Press), *Where the Action Is: The Foundations of Embodied Interaction* by Paul Dourish(MIT Press), *Digital by Design* by Troika, and *Responsive Environments* by Lucy Bullivant (Victoria and Albert Museum) are just a few.

Sensing Environmental Data

You can detect change in an environment to create a response for a user in many different ways: detecting Bluetooth signals, heat, movement detected by ultrasonic sensors, computer vision techniques, weight sensors in a floor, the time of day, the amount

of noise...the list is nearly endless. The idea of gathering ambient data about a space or an area and using that in an artwork or design object has been around for a long time. Audio artists have used ambient and environmental sound for a long time to create sound works. The techniques to create these works are vastly different and range from John Cage's use of the sound of the audience in his piece *4'33* to Barry Truax's *Dominion* to David Cunningham's *Listening Room*.

The last 40 years have seen a steady architectural dialogue about how to shape environmental data to create a meaning or a particular sensation. The art and design collective Plaplax creates simple reactive spaces and spatial interfaces, children's toys, and collaborations with dance companies, always working with both installation space and the body of movement of the user to create interaction. The group Random Internationals has created pieces like *Audience*, which has several hundred small mirrors attached to servos that track people as they walk past, creating a kaleidoscopic reactive mirror. What makes this approach to artwork different from what you'll be reading about in this chapter is that in those works the processing of the environment data is done by the person in the space rather than by a device. The things that we can perceive in a space can be very different from the things that a device perceives in a space or an environment.

The data of your application will depend on the physical relationship that the user has to the space where the data is being gathered. It's important to consider carefully how the data that you're sensing in an environment is being used, whether you'll make the user aware of what your system is detecting and what it will do with that data, how you'll alert them to that, and how you'll allow them to control the system.

On a more pragmatic level, what are the easily available data points for a space or environment? The temperature is one; the amount of ambient light in the room can tell you whether the lights are on or not, whether the room gets natural light, and whether it's day or night. The color of light in a room is another one, especially if the space receives some natural light. Magnetic sensors can detect direction or a change in the location of magnetic elements in a space. Ultrasonic, infrared, and PIR motion sensors can all be used to detect movement; taken over time, this data indicates the amount of movement in a space as a whole or just as a detection that there is something in a particular part of a space.

Using an XBee with Arduino

To understand what the XBee is, we first have to talk about the ZigBee protocol. ZigBee is a standard for communicating over wireless networks that is designed to be inexpensive and not require a lot of power, making it perfect for small devices and ubiquitous computing. Its low cost means that you can deploy lots of devices cheaply to create a larger network, and its low power use means you can use small batteries and make devices run for longer periods of time. ZigBee is particularly well designed for mesh networks, which connect from node to node, rather than networks that are routed

through a single router, like most wireless connections. Mesh networks are slower, but they also are set up to allow many devices to communicate with many other devices instead of assuming that all devices want to connect to the same single device. Mesh networks are very good at *self-healing*. That is, when one node goes down, the other nodes that are not down can redirect messages around that down controller. The goal of the ZigBee protocol is to have an inexpensive and general-purpose way to create mesh networks to use in many different settings. Just some of the applications where ZigBee can be used are the following: smart lighting, advanced temperature control, safety and security, movies and music, water sensors, power sensors, smoke and fire detectors, smart appliances, access sensors, industrial controls, and monitoring.

The XBee device is a ZigBee-enabled device commonly used with the Arduino. This means that there are numerous good tutorials by Tom Igoe and other member of the Arduino community; there are shields developed by Libelium; and if you're working with the XBee controllers, you can count on being able to find someone in the Arduino community who can help you answer questions that you might have. Depending on the model, an XBee can communicate over distances up to 100 meters indoors. Using the XBee outdoors with line of sight between controllers, you can send and receive signals at a distance of up to 2 kilometers; if you use high-gain antennas, you can send signals up to 6 kilometers. You can use the XBee as a serial replacement, or you can put it into a command mode and configure it for a variety of broadcast and mesh networking options. You can do quite a number of things with such a small module. Figure 17-1 shows the different XBee options.

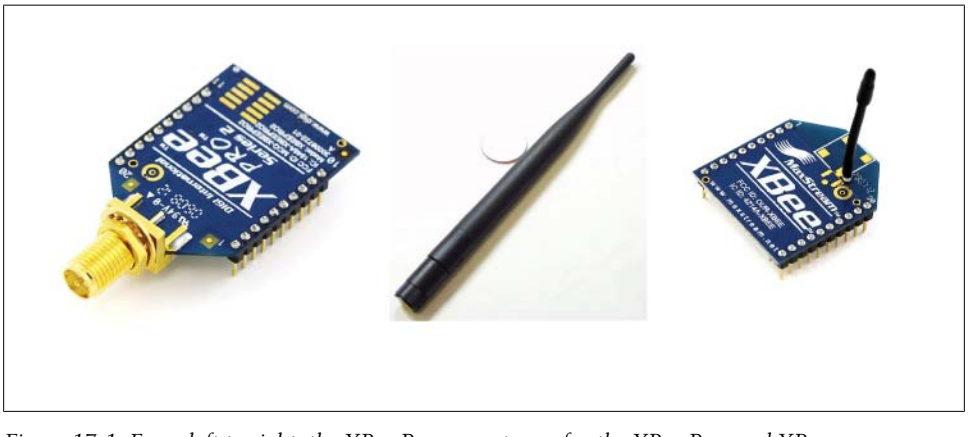


Figure 17-1. From left to right: the XBee Pro, an antenna for the XBee Pro, and XBee

The XBee does have some limitations. The XBee 900 and XBee DigiMesh 900 transmit and receive at 900MHz and cannot communicate with the 2.4GHz frequency XBee controllers. Table 17-1 lists some information about the capabilities of specific models.

Table 17-1. XBee types and frequencies

Name	Frequency	Type	Range (outdoor)	Notes
XBee 802.15.4	2.4GHz	Point-multipoint	100m (1.5km)	Antenna-ready. Compatible with Digimesh 2.4.
XBee DigiMesh 2.4	2.4GHz	Mesh	100m (1.5km)	Compatible with XBee 802.15.4.
XBee Pro	900MHz	Point-multipoint	10km	Compatible with Digimesh 900.
XBee DigiMesh Pro	900MHz	Mesh	10km	Compatible with XBee 900.
XBee XSC Pro	900MHz	Both	25km	Compatible with DigiMesh Pro.
XBee ZB	2.4GHz	Mesh	120m (1.5km)	Antenna-ready.

Getting the XBee up and running can be a bit tricky. It requires a little configuration and the use of a terminal application, such as the command prompt on Windows or the Terminal application on OS X.

Note the circle on the jumpers for the XBee shield in Figure 17-2. These indicate whether the XBee will use the serial connection from the USB port of the Arduino or whether the Arduino controller itself will be using the serial port. If you're not using a shield because you're using an Arduino Mini or Pro, then you can use either the USB programmer to communicate with the XBee or an Arduino board with the processor removed, but it's strongly recommended that you use a shield. You can find instructions for configuring the XBee without using a shield on the Arduino website, if you're interested. In this book, though, we'll be covering using the XBee shield.

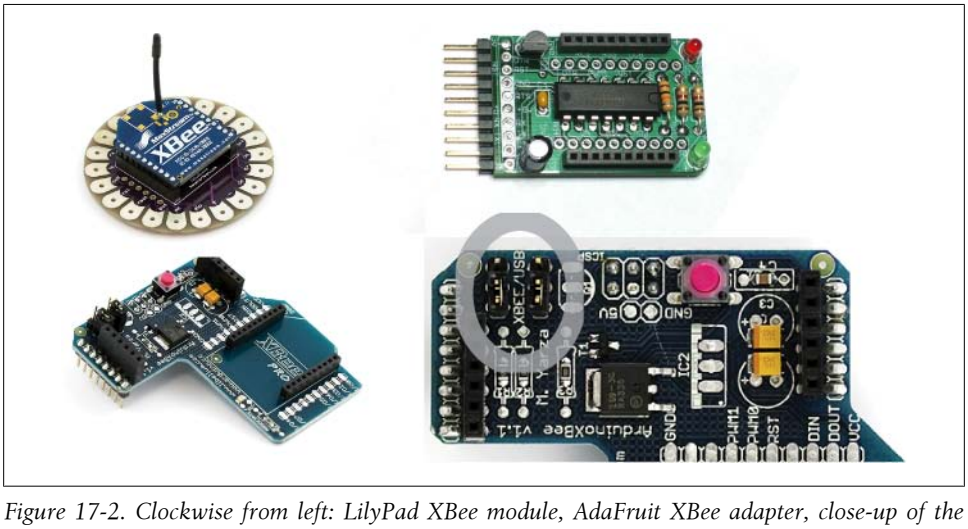


Figure 17-2. Clockwise from left: LilyPad XBee module, AdaFruit XBee adapter, close-up of the jumper pins on the XBee Shield, the Arduino XBee shield

The jumper on the shield, shown in Figure 17-3, sets the mode of the XBee shield. You set the shield to XBee mode to configure the XBee chip itself when you're setting the baud rate for communication or setting the names of the controller. You set the shield to Arduino when you want to run the Arduino and have it communicate with the XBee shield.

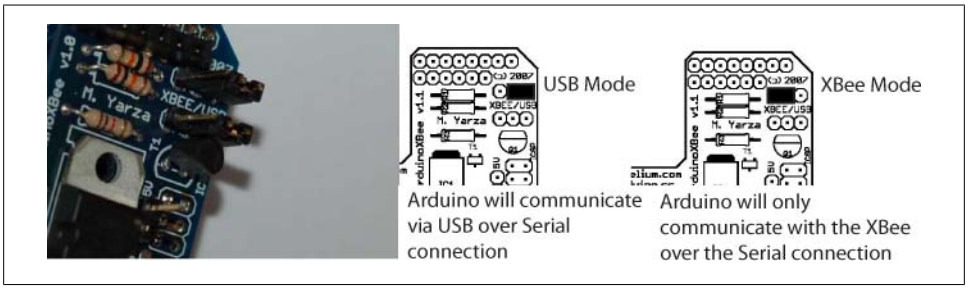


Figure 17-3. Jumpers on the XBee shield

When the jumpers are in the XBee position, data sent from the microcontroller will be transmitted to the computer via USB as well as being sent wirelessly by the XBee module. The Arduino microcontroller will be able to receive data only from the XBee module, not over USB from a computer.

When the jumpers are in the USB position and the microcontroller is left in the Arduino board, the Arduino will be able to talk to the computer normally via USB, but neither the computer nor the microcontroller will be able to talk to the XBee module. If the microcontroller has been removed from the Arduino board, the XBee module can communicate directly with the computer. You'll do this if you want to configure the XBee controller.

Creating a Simple Test

To upload an application to an Arduino board with an XBee shield, the shield needs to have its jumpers set to USB. This means ensuring that the two pins to the right of the jumper are connected, rather than the two pins on the left of the jumper. Now you can upload an application from the Arduino IDE as you normally would. The XBee module on the shield is set up to work at 9600 baud by default, so unless you reconfigure it, you'll need to make sure you're passing 9600 to the `Serial.begin()` command in your sketch (as shown in the following code). If your first Arduino doesn't have an LED on pin 13—that is, if it isn't a Duemilanove—then connect an LED to pin 13 and upload the following sketch (code). This sketch *reads* a value sent over the XBee:

```
void setup() {
    Serial.begin(9600);
    pinMode(13, OUTPUT);
}
```

```
void loop() {
    if (Serial.available()) {
        byte val = Serial.read(); // this will read from the XBee
        if (val == 'X') {
            digitalWrite(13, HIGH);
        }
        if (val == 'O') {
            digitalWrite(13, LOW);
        }
    }
}
```

Now unplug the first Arduino board from the computer, and switch the jumpers to XBee with the center pin and the pin farthest from the edge of the board connected. On the second Arduino controller, make sure the jumpers are in the USB setting, and upload the following sketch to the board. This sketch is *sending* data over XBee that the receiving sketch will read:

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    Serial.print('X');
    delay(1000);
    Serial.print('O');
    delay(1000);
}
```

Turn off the serial monitor, and unplug the board. Switch the jumpers to the XBee setting. Now connect both boards to the computer to power them up. After a few seconds, you should see the LED on the first board running the receiving program turn on and off, once every second. This means that your Arduino boards are communicating wirelessly. Now you can power the Arduinos using 9V batteries, taking care to connect them correctly, and move them to different rooms in a house; you should see them continue to communicate.

The jumpers are important because when they're set to USB, the Arduino Serial `print()` and `println()` methods will send and receive information over the USB port. When the jumpers are set to XBee, then those methods will send information over the XBee. The `read()` method of the Serial works the same way, so when the shield is set to XBee, you'll find that the Arduino will communicate only with the XBee and not with a computer attached to it. This is important to keep in mind when you're working with the XBee shield: If things aren't working, check your jumpers first. You can also communicate directly with the XBee shield from a computer by connecting it to an Arduino board whose microcontroller has been removed and placing its jumpers in the USB configuration. This works only with Arduino controllers that use the Atmel 168 or 328 like the Duemilanove and Decimilia. With this configuration, you can send data to and receive data from the XBee module with any terminal program (such as the

HyperTerminal in Windows or the Terminal in Linux and OS X). This allows you to see the data that the module is receiving from other XBee shields.

Configuring the XBee Module

The XBee is in some ways like the Arduino itself in that it is programmable using commands. Table 17-2 lists some of the most commonly used commands.

Table 17-2. XBee programming commands

Command	Description	Possible values	Default
ID	The network ID that the XBee will use. Only controllers that have the same network ID can communicate.	Between 0 and 65,535 or between 0x0 and 0xFFFF	3332 or 0xD04
CH	The channel of the XBee module.	Between 0x0B and 0x1A	0x0C
MY	The address of the module.	Between 0 and 0xFFFF	0
DH and DL	The destination address for the wireless communication (DH is the high 32 bits; DL is the low 32).	Between 0 and 0xFFFFFFFF	Starts at 0
RE	This restores the factory settings of the XBee.	N/A	N/A
WR	This writes the parameter values to the memory of the XBee so that the next time it is powered up, the XBee will use its new settings.	N/A	N/A
BD	This sets the baud rate that the shield will communicate at.	0 (1200 bps)	3
		1 (2400 bps)	
		2 (4800 bps)	
		3 (9600 bps)	
		4 (19200 bps)	
		5 (38400 bps)	
		6 (57600 bps)	
		7 (115200 bps)	

You can configure the XBee module with code running on the Arduino board or with software on the computer. To configure it from the Arduino board, you'll need to have the jumpers in the XBee position. To configure it from the computer, you'll need to have the jumpers in the USB configuration and have removed the microcontroller from your Arduino board.

You'll need to follow a few steps to configure your XBee. If you're using the Libelium shield, you'll need to remove the processor from the Arduino module that the shield is connected to and then connect the Arduino to a computer. If you're using the AdaFruit shield, then you simply need to connect the shield to the FTDI cable that comes with the shield. Now, you're ready to get the module into configuration mode, which you can do in the Arduino IDE by sending three plus symbols (+++).

There's one trick here: If you're trying to configure the module from the computer, you need to make sure your terminal software is configured to send characters as you type them, without waiting for you to press Enter. Otherwise, it will send the plus signs immediately followed by a newline (that is, you won't get the needed one-second delay after the +++). Make sure that when you type in the three +++ signs that you do it quickly; there can't be more than a second delay in between each +, or it won't work correctly. If you successfully enter configuration mode, the module will send back the two characters `OK`.

Now, you're in configuration mode, and you can begin sending the commands listed in Table 17-2. Note, though, that all those commands are prefaced by the letters `AT` and that no space is placed between the command and the value. For instance, to set the XBee to use a baud rate of 19200 instead of 9600, you would type the following:

```
ATBD19200
```

Then press Enter. The controller should respond with the characters `OK`. Take note, though, that if you change the baud rate, you'll have to change the terminal baud rate too, or the terminal won't be able to communicate with the XBee any longer. You may need to restart your terminal controller.

You can also use the commands to query the controller. To get the ID of the controller, send the `ATID` command without a value:

```
ATID
```

Then press Enter. The controller should respond with the values `3332` (the default), unless you've configured it to be something else.

It's important to note that unless you tell the module to write the changes to nonvolatile (long-term) memory, the changes will be in effect only until you power off the module. To save the changes permanently, use the `ATWR` command followed by pressing Enter to which the module should respond with `OK`.

To reset the module to the factory settings, use the `ATRE` command, and then save the changes using `ATWR`. Note that like the other commands, the reset won't be saved into the module unless you follow it with the `ATWR` command.

You can also use multiple commands like so:

```
ATID3333,BD4,WR
```

This sets the ID and the baud rate and then saves it to the module.

Addressing in the XBee

The XBee uses a few different ways of addressing and finding objects in its network. There are three different kinds of values that can be used to define what can communicate with a particular XBee module and what other modules that XBee module will listen to messages from:

Individual module addresses
: Each XBee controller has an address that is set at the factory. You can also set the value of the `ID` using the `DL` and `DH` commands. If a message is sent with a destination `ID`, the XBee whose address is this destination `ID` will hear the message.

Personal area network (PAN) IDs
: This is another way that the XBee can be configured that has a coordinator sending out signals to any device using a particular PAN ID and one or more end devices, like a broadcaster and listeners. If a message is sent with a `PAN ID` and that module has that `PAN ID` set, then it will hear the message.

Channels
: Any number of XBee modules can be set to use a certain channel, and that channel determines what messages the XBee module receives. If that module has the same channel as a message, then it will receive and process the message.

For two modules to communicate, they must be on the same channel, they must have the same PAN ID, and the destination address of the sender must match the address of the receiver.

Each XBee module has its own unique address, as well as a destination address to which it sends its messages. The destination address can specify a single destination, or it can be a broadcast address, which will be received by all XBee modules within range. The broadcast address is `65535` (`0xFFFF` in hexadecimal).

Now, this may seem a bit complex, but if you have only a couple modules, you'll probably never need to change the PAN ID or channel. But if you're in an environment with lots of modules, you might want to be sure that nobody else's messages are getting mixed up with yours, so this offers a nice option to handle that.

If you're on a Windows computer, then to configure your XBee controllers further, you'll probably end up working with the XCTU terminal controller from Digi, available from *www.digi.com/*, to configure your controller. If you're on Linux or OS X, you'll open a terminal and type something like the following.

First, list the serial ports, and find the one you want:

```
$ ls -l /dev/tty.*
```

This is what I use, but what you need to type will depend on the port you use. The following line of code opens a program called `screen` and tells it to open the USB port `/dev/tty.usbserial-A9003QNn` using 9600 baud:

```
$ screen /dev/tty.usbserial-A9003QNn 9600
```

Now enter command mode:

```
+++
```

Finally, get XBee firmware version; in `screen`, the `ctrl-A b` indicates a carriage return indicating that the command is finished:

```
ATVR ctrl-A b
```

If you're not using `screen`, you won't need to use the `ctrl-A b` command.

A tutorial on XCTU is out of the scope of this book and far better documented on the Arduino forums and in *Making Things Talk* by Tom Igoe than I could do here, and the forums and Tom's book are also substantially friendlier and easier to use.

So, what can you do now? Well, there are a few interesting ideas. If parts of a space can communicate with one another wirelessly, then opening a door could turn on the lights in the kitchen quite easily; power usage for a house could be monitored; blinds could be raised at a certain time to wake someone; and a single command could customize the temperature, lighting, and sound in a room for a particular user, all without running wires throughout the space. Health care, education, and business are all fields where ways to integrate responsive and intelligent architecture with the social activities, environmental concerns, and tasks in those fields are being explored in quite interesting ways. Another option is to simply synchronize machines or components throughout space, making the movements of one small robotic instrument match another, synchronize behavior across machines, or discover new machines as they are introduced to the environment. You can also synchronize data-gathering behavior across machines, creating simple networks that don't require much power to communicate; sensors can communicate with a computer handling all the incoming data, and adjustments can be made to sensors.

Later in this chapter, you'll look at an example of sending temperature data from multiple sensors to an Arduino linked to a home automation system.

XBee Library for Processing

Another option for working with the XBee and the Arduino is the XBee library for Processing that Rob Faludi, Daniel Shiffman, and Tom Igoe worked together to create. One of the difficulties in working with the XBee controller is parsing out the data from the messages sent by the XBee. By using Processing and the XBee library for Processing, you can parse out the data from a message quickly and easily. The following example assumes that you have two XBee modules that are sending messages to a receiver that is connected to a computer via a Serial connection. This means either having an Arduino with its chip removed, allowing it to communicate directly with the XBee, or alternatively using the following:

```
import processing.core.*;
import processing.serial.*;
import xbee.XBeeDataFrame;
import xbee.XBeeReader;
```

Here is a Serial port that will communicate with the XBee module:

```
    Serial port;
```

An `XBeeReader` object will read the data from the XBee module and put it in a friendly and easy-to-read format:

```
XBeeReader xbee;
int[] analog;
int[] digital;

public void setup() {
    size(400, 400);
```

Remember that the XBee will be using the 9600 baud rate unless you change the settings of the module:

```
    port = new Serial(this, Serial.list()[0], 9600);
    xbee = new XBeeReader(this, port);
    xbee.startXBee();
    println("XBee Library version " + xbee.getVersion());
}
```

Chapter 8 introduced the Processing Serial library that allows a Processing application to read data from the serial port of your computer. The XBee library works quite similarly to the Serial library with an event handling function called `xBeeEvent()`, which, like the `serialEvent()` method of the Serial library, is called for you when new data is available to be read. Of course, the XBee library reads data only from the XBee and not from the Serial, but the concept is the same:

```
public void xBeeEvent(XBeeReader xbee) {
    println("Xbee Event!");
```

Next, grab a chunk of data from the XBee, often referred to as a *frame*:

```
    XBeeDataFrame data = xbee.getXBeeReading();
```

The Processing library checks to make sure that the received data is in the correct series. The XBee library supports several different packet types, but each contains different data:

```
    if (data.getApiID() == xbee.SERIES1_IOPACKET) {
```

Now, we want to loop through all the data received in the frame and determine what data has arrived in the packet:

```
        int totalSamples = data.getTotalSamples();
        for (int n = 0; n < totalSamples; n++) {
            print("Sample: " + n + "  ");
            // Current state of each digital channel
               //(-1 indicates channel is not configured)
            digital = data.getDigital(n);
            // Current state of each analog channel
              //(-1 indicates channel is not configured);
            analog = data.getAnalog(n);

            for (int i = 0; i < digital.length; i++) {
                print(digital[i] + " ");
            }
```

```
                for (int i = 0; i < analog.length; i++) {
                    print(analog[i] + " ");
                }

            }
        } else {
            println("Not I/O data: " + data.getApiID());
        }
    }

    public void draw() {
        background(0);
        fill(255);
```

Draw the data received in the last packet to the screen:

```
        for (int i = 0; i < digital.length; i++) {
           ellipse(50, i* 20, digital[i]);
        }
        for (int i = 0; i < analog.length; i++) {
           ellipse(250, i* 20, analog[i]);
        }
    }
```

Now that you see how to get data from the XBee, you'll probably want to send commands to it using the `AT` commands. The XBee library defines methods to send most of the primary XBee commands for getting and setting destinations and channels. To test some of the key `AT` commands, you can run the `This` method to respond to key presses when the program window is active: These methods test a few of the different `AT` commands:

```
public void keyPressed() {

    switch (key) {

    case '1':
        println(" Do node discovery and find any available nodes: ");
        xbee.nodeDiscover();
        break;
    case '2':
        println("Set the destination node of the XBees ");
        // this can be whatever you would like, just make sure that
        // there is a valid node first
        xbee.setDestinationNode("205");
        println();
        break;
    case '3':
        println("Get the Channel of the XBee using the CH command");
        xbee.getCH();
        break;
    case '4':
        println(" Send a datastring over the XBee ");
```

Here, the XBee will send a data string out using the address passed in. The two hexadecimal numbers that you see as the first two parameters are the high byte and low byte

to send as the destination of the message. This way, an XBee can send a message with an address, and any other XBee can decide whether it wants to listen to the message. You can change these address values:

```
            xbee.sendDataString(0x0013A200, 0x403E17E6, "Bonjour!");
            break;
        case '5':
            println("Getting the high byte of the destination of the XBee");
            xbee.getDH();
            break;
        case '6':
            println("Getting the low byte of the destination of the XBee");
            xbee.getDL();
            break;
        case '7':
            println("Getting the ID of the XBee that sent data");
            xbee.getID();
            break;
        case '8':
            println("Get the Node Identifier using the NI command");
            xbee.getNI();
            break;
```

Another aspect of the XBee that we haven't brought up yet is the ability of the XBee to write digital or PWM data out via one of its digital input or output pins:

```
        case '9':
            xbee.setIOPin(1, 5);
            break;
        case '0':
            xbee.setIOPin(1, 4);
            break;
        case '-':
            println("get the address of the XBee that sent data ");
            xbee.sendRemoteCommand(0x0013A200, 0x403E17E5,0xFFFE,"MY",-1);
            break;
        }
    }
```

Paired with a Processing application, the XBee is a very powerful way to send commands remotely, across spaces such as a garden, field, or even, if you use a powerful enough XBee, across a small town, and easily integrate that with a Processing application. Of course, you can also use the XBee without connecting it to a Processing application. Paired with a relay like the RelaySquid or with a hand-built relay circuit as you saw in Chapter 11, the XBee can be used to control lights, appliances, or other large devices. It could also be used to set the values on a servo, drive a LCD display, or do anything else that an Arduino can do.

With some careful planning, you can use the node discovery capabilities of the XBee to create networks that configure themselves when new nodes are added or removed, with XBees on the network alerting other XBee controllers about the node's capabilities are and what kind of information they expect.

Placing Objects in 2D

One of the more common tasks that you might find yourself wanting to do when working with a space is to locate objects in two dimensions. You might want to determine whether a person is in a room so that you can turn the lights on, whether someone is approaching a doorway, or whether an object has been moved to a certain location. Behind a lot of complex reactive effects is the simple determination of where a person is in a room or space so that changes can be made to the environment.

Before you get started placing objects in two dimensions, it's important to know how to place them in one dimension—the distance between a sensor and an object in a straight line. Here's a simple piece of code to do just that with an ultrasonic sensor like the ones discussed in Chapter 8:

```
unsigned long echo = 0;
```

Connect the PW pin on the ultrasonic sensor to pin 9 on the Arduino:

```
int ultraSoundSignal = 9;
unsigned long ultrasoundValue = 0;

void setup()
{
    Serial.begin(9600);
     pinMode(ultraSoundSignal,OUTPUT);
}

unsigned long ping(){
     pinMode(ultraSoundSignal, OUTPUT); // Switch signalpin to output
     digitalWrite(ultraSoundSignal, LOW); // Send low pulse
     delayMicroseconds(2); // Wait for 2 microseconds
     digitalWrite(ultraSoundSignal, HIGH); // Send high pulse
     delayMicroseconds(5); // Wait for 5 microseconds
     digitalWrite(ultraSoundSignal, LOW); // Holdoff
     pinMode(ultraSoundSignal, INPUT); // Switch signalpin to input
     digitalWrite(ultraSoundSignal, HIGH); // Turn on pullup resistor
     echo = pulseIn(ultraSoundSignal, HIGH); //Listen for echo
     ultrasoundValue = (echo / 58.138); //convert to centimeters
     return ultrasoundValue;
}

void loop()
{
     int x = 0;
     x = ping();
     Serial.println(x);
```

Next, delay the next ping by one-fourth of a second to make sure that the signals don't interfere with one another:

```
    delay(250); //delay 1/4 seconds.
}
```

There are a few different strategies to place an object in space. This depends on what you are defining as "space." You can always use a GPS for placing an object in geographical space, but it doesn't help when placing an object in a room. Using a pair of cameras to position an object is an easy way to position an object in a room, but it requires that you have multiple cameras and that the cameras are connected to a computer that can handle multiple input streams from cameras. This is more a matter of picking the right camera and configuring your hardware than programming, but it is quite possible to do in an oF application. You can find more information on how to do this and what cameras would be best suited to your system on the oF forums and website.

To locate the object in a room, you'll probably want to use a set of sensors, such as in Figure 17-4, that will provide enough data for you to triangulate the position of the object. You'll need to determine the position of the object along at least two axes, usually the x and y. This presents a few problems, though, if you're trying to track multiple objects through space or if the space that you're interested in working with has multiple obstructions in it.

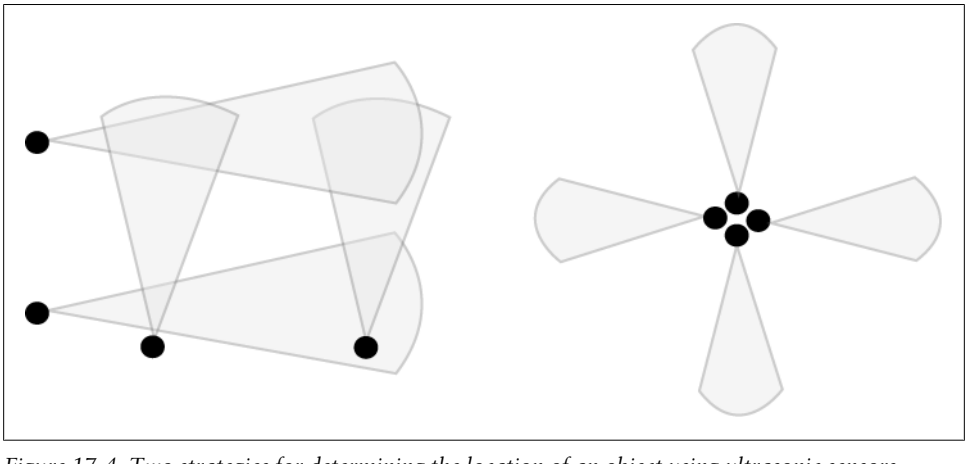


Figure 17-4. Two strategies for determining the location of an object using ultrasonic sensors

Using ultrasonic sensors in tandem presents a challenge: Numerous sensors often interfere with each other when placed within "hearing" distance of each other. This is because each sensor has no way of knowing which short pulse of sound came from which sensor and the data readings can get very unpredictable. The solution is to synchronize the sensors to prevent one sensor from listening while any other sensor is clicking. Infrared sensors present somewhat the same problem, except with light instead of sound; but they have an additional disadvantage: They cannot be easily sequenced, while the ultrasonic sensor manufactured by Maxbotix can.

To chain the Maxbotix ultrasonic sensors and have them to operate in sequential daisy-chained fashion (as in Figure 17-5), you link the TX pin of unit 1 to the RX pin of unit

2, and so on. The BW pin is connected to +5 volts on each sensor. Then just set the pin connected to the first sensor's RX pin to `HIGH`, (shown in Figure 17-5 as pin 6), to start the chain reading, and all of the sensors will read in sequence. The analog values can then be read. In the example figure, you use one pin to command the chain and three analog in pins to read the data from the sensors.

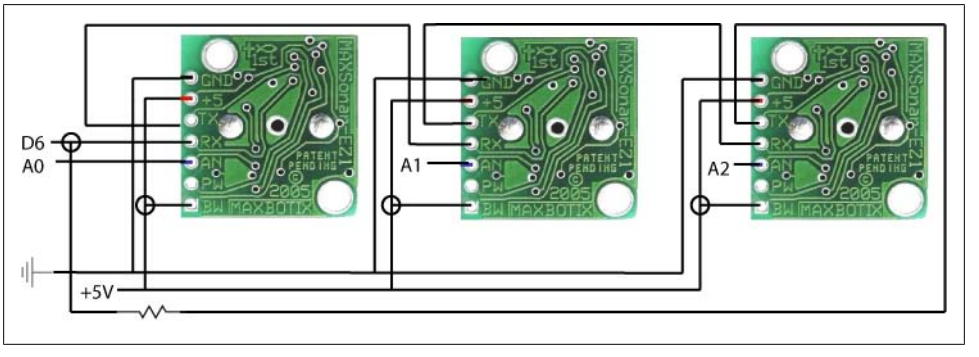


Figure 17-5. "Daisy-chaining" multiple ultrasonic sensors together

This chains the sensors together so that the values from each sensor will be sent to the analog pin 1. You'll need to add a resistor between the last sensor's TX pin back to the RX pin of the first unit through a 1K resistor.

Now you can begin reading them in sequence. In your `setup()` method, you want to have a delay of 250 milliseconds after powering on the sensors to give them time to boot up. Each time you want to read the ring of sensors, you have to "kick start" them by setting the RX pin `HIGH` on the first sensor for 20 microseconds. Now all of the sensors in the chain will run in sequence. This "ring of sensors" will cycle around and around, constantly maintaining the validity of their analog values. You can then read the latest range reading at any time. This is the easiest way to use them. After setting the pin on the Arduino connected to the RX of the ultrasonic sensor to `LOW`, you should wait 100 milliseconds for the sensor as it calibrates itself. After that you can read the analog pin roughly every 50 milliseconds. The most recent range reading is always ready to be read on the analog voltage pin, so once you start the chain and if you are using it in continuous mode, you can read the values at any time:

```
int ultraSoundSignalPins[] = {0,1,2}; // 3 Ultrasound signal pins
int ultraSoundTriggerPin = 6;        // output pin to start Ultrasound signals

void setup() {
    Serial.begin(9600);

    for(int i=0; i < 3; i++) {
        pinMode(ultraSoundSignalPins[i], INPUT); // Switch signalpin to input
    }
    pinMode(ultraSoundTriggerPin, OUTPUT); // set this pin to output
   //give the sensors time to boot up
    delay(250);
```

```
    // send RX pin high to signal to chain to ping
    digitalWrite(ultraSoundTriggerPin, HIGH);
    delayMicroseconds(20);
    digitalWrite(ultraSoundTriggerPin, LOW);
    pinMode(ultraSoundTriggerPin, INPUT); // electrically disconnects the pin
    delay(50);
}

void loop()
{
    unsigned long ultrasoundValue;
    unsigned long echo;
    delay(50);
    for(int i=0; i < 3; i++)
    {
```

Now the values from each sensor can be read:

```
        echo = analogRead(ultraSoundSignalPins[i]); //Listen for echo
        ultrasoundValue = (echo / 58.138);
    delay(50);
    }
}
```

The configuration shown in Figure 17-5 shows how multiple sensors could be configured. Setting them at the center of a room and reading them around from left to right is probably the easiest option since it allows you to determine the location of a reading very easily. Getting good coverage of a room requires quite a few sensors, though, and it can be less than optimal to put sensors in the middle of a room. Setting the sensors to read across from one another is perhaps more precise because it allows you to actually determine in two dimensions the location of an object, but it's slightly trickier to put the values together and easier to get false readings if the sensors are not timed correctly or if there are multiple objects in the room. In either event, some planning ahead and experimentation will successfully let you determine the location of one or more people or objects in a room to a fair degree of accuracy.

Interview: Usman Haque

Usman Haque is a London-based architect and artist who designs interactive architectural systems and develops both physical spaces and the software and systems that bring them to life. He currently teaches at the Bartlett School of Architecture in London, England. His installations have been exhibited at the Institute of Contemporary Arts (London), the Hillside Gallery (Tokyo), the Tokyo Metropolitan Museum of Photography (Tokyo), and the Plymouth Arts Centre (Plymouth). His projects have been published in several magazines and journals including *The Architects' Journal*, *Artifice*, *Wallpaper*, *Wired Online*, *WebMaster* magazine, *.net* magazine, *Architectural Design*, ZDNet, and the *RIBA Journal*.

For my readers who might not be familiar with you and your work, could you give a very brief summary of the kinds of projects that you're working on currently?

Usman Haque: There have been three particular directions that my work in the past has taken, guided particularly by my background in architecture and general interest in how we relate to each other and to the space we form around ourselves: (1) those involving participation and interaction, often in large-scale productions in urban parks; (2) experiments exploring the process of perception, both in humans and machines; and (3) designing design systems, for example by finding ways to involve people who might not otherwise call themselves "designers" in the processes of design.

The direction of the work I'm involved in these days has changed a little. I've become particularly interested in tying together the previous strands, looking particularly at self-powered, dynamic, and permanent systems. I'm spending a lot of time nurturing Pachube.com. I'm also getting much more rigorous about what I describe as "interactive."

Can you explain a little bit about how you and your collaborators came to work together and provide a little insight into how your actual day-to-day collaborations on a project takes shape?

Usman: There's no conscious process involved in developing a collaboration. I usually find that I'm involved in a collaboration without specific planning. The project just takes shape in conversation; by contrast, when I have a project already in mind and I need specific additional input from others, I get a little embarrassed asking people whether they would like to work on it with me.

My favorite collaborations (and the usual way that these occur) are those that ensue from just hanging out with people I enjoy being with. The *Evolving Sonic Environment* project developed from having lunch with Rob Davis one day and having a conversation where we speculated about trying to go "totally analog"; the *Wifi-Camera* project came about when Adam Somlai-Fischer and Bengt Sjolen and I were huddled together at an exhibition opening; Seth Garlock got involved in *Sky Ear* during one of our (quite frequent) three-hour phone calls. (This was in pre-Skype days, and up until that point we hadn't talked about working together. Our conversations were usually about books and politics and egg-cream sodas. We just slipped unknowingly into the world of collaboration when he said, "Why don't your balloons talk to each other?"). When Rolf Pixley and I started working together, I think it was largely because we misunderstood each other, and the ongoing collaboration was a process of discovering that we might in fact be in agreement (!); working with Yu Nishibori on *Infinitum Ad Nauseam* came about because I really liked his meal recommendation after I met him in a restaurant (I also liked his rubber ducks); working with Despina Papadopoulos on *1000* ensued after several years of being friends—partly we just wanted to find a way to get funding to carry on the kinds of conversations we were having. Anyway, I work with people I like. I don't worry about how/whether our skills overlap, complement, or jar. That usually gets clarified in the process of collaborating.

Working with Rolf on *Open Burble* is something I consider very precious. Partly through his pressure and partly because I want to know and understand every part of the projects I'm involved in, our conversations would swing rapidly from discussions on the timing of setting individual bits and registers within a microchip, all the way through conceptual discussions about the way the human nervous system functions, up through the

perceptual mechanisms involved in looking at a lit balloon against the rapidly changing colors of an evening sky. I do think it's important for someone who is going to put his or her name on a project, even in a collaboration, to understand exactly what's going on in the project at all levels.

I'm quite interested in the idea of architecture being something "no longer considered static and immutable." Can you elucidate this a little bit?

Usman: Architecture has long been considered something permanent, static, and unchanging—the walls, roofs, and floors that surround us and that condition the way we relate to space and to each other. In the last few decades, however, a new idea of architecture has come about (largely influenced by Cedric Price): one in which architecture is all the stuff in between; it's ephemeral, permeable, and dynamic. I've sometimes described this as the "software" of space, rather than the "hardware," analogizing computer terminology. The point of this distinction is to say that there are bits that appear to stay the same, and there are also bits that change in terms of their relationship to the whole. I'm interested in the kind of architecture that is constantly constructed, something that is constantly responsive (even if that response is to "do nothing"), something that is constantly on the precipice of change.

Do you see parallels between developing a system that replicates life and a system that integrates with life?

Usman: The reason I undertake projects such as *Evolving Sonic Environment* is less about trying to make an "intelligent" space and more about trying to understand how "we humans" understand space. By attempting to build a space that "understands" us, I'm really trying to understand "understanding."

When I say "a space that understands," I mean one that modifies its perceptual processes and creates its own categories of occupancy. The point is not so much to integrate so-called "intelligent" systems with living systems but rather to develop an approach to environment design in which humans are not at the mercy of technology designed and determined by others—to develop adaptive, responsive, conversational systems, where the outcome is determined by both participants, not just the machine (which is the usual determinant of outcome in human-machine relationships).

What sorts of strategies or thinking do you use to ensure that a participant is free to participate in the continued design and evolution of a space or project or structure?

There are two important limits to the idea of a collaborative design system in architecture.

Usman: First, the more "open" a system is, the more bewildering it is for those who are introduced to it anew. In a design system in particular, people are often aided, not hindered, by having constraints applied to them—the real key is to find a way that the constraints are not necessarily absolute. If the constraints themselves can be modified over time, then that really helps in fostering ongoing participation. It is, in a sense, what learning is all about.

Second, I don't believe that it's possible to construct a "totally open system"—the very nature of a "system" (which is a description of a set of relationships, dependent in all cases upon a particular subjective approach) precludes it. There will always be pro-

cesses that involve "topdownness" and "bottomupness"—the key to helping perpetuate the design and evolution of the kinds of conversational environments we're talking about is to ensure that the relationship between the two processes is not frozen.

It's also important to ensure a varied "granularity" of participation: Different people have different interests or skill sets, are willing to participate to a greater or lesser degree, or desire to have a smaller or larger impact on the entity being constructed. Everyone will have their own point at which they wish to engage with something, so it's vital that a wide variety of entry points and methods are available.

You've written before about Gordon Pask and his conversation theory; can you explain a little about how that's useful to you?

Usman: In the context of architecture, conversation theory is useful in two ways: (1) in considering the architectural construct as an "individual" in conversation with another "individual" (often a human being, who happens to occupy the architectural construct) and how these two can interact to their mutual benefit (and coherence); and (2) in considering an "environment" (such as architecture) as being something that is constructed when two individuals interact (for example, a human and a machine) and create "coherence." In this second example, interaction itself gives rise to architecture, and it is an arbitrary distinction whether the occupant is engaged with the "architecture" or the "computer-system-that-drives-the-architecture."

For an understanding of how I define many different flavors of "interaction," it might be useful to see the "What is Interaction? Are there different types?" article in January 2009's *ACM Interactions*, an article I coauthored with Hugh Dubberly and Paul Pangaro, based largely on the Paskian cybernetic approach.

Who are some of the artists and thinkers that you refer to most in your own work and thinking about interactivity and architecture?

Usman: I tend to be influenced really easily. So, if I stand next to someone in the subway, it probably affects the way I'm thinking about the world. However, apart from Gordon Pask, whom you have already mentioned, there are others who I look to often, including architect Cedric Price; Ross Ashby, a pioneer in systems theory; Kenneth Craik (who wrote *The Nature of Explanation* [Oxford University Press]); Stephen Groak (former director of "blue skies" research at Ove Arup and author of *The Idea of Building* [Spon Press]); and Natalie Jeremijenko, whose work on the "structures of participation" has been very influential. I often get lost in the mountainsides of the Pataphysicians (pataphysics being to metaphysics what metaphysics is to physics); my favorite artists are probably the group Gelitin.

Do have a vision of the way that computing and computation will engage with the kinds of thinking and tasks that architecture and the design of spaces requires?

Usman: The vision I have is what I fear will happen: that our technological systems will be designed and built by corporate entities with their own rationales for existing, that these systems will be imposed upon us, and that we will be unable to enter into them in order to change the way they respond to us. They will be private, proprietary, unhackable, and unchallengeable. Imagine having a closed system, just like Microsoft Windows, controlling your physical desktop, your thermostat, or your home life.

That's the vision I'm trying to escape when thinking about ways to push the architecture and computation direction.

What I would like to head toward instead, and this comes out particularly in Pachube.com, is an ecosystem of conversant devices, environments, architectural entities—an ecosystem that is open, constantly being reinvented and repurposed, and constantly adjusting to new perturbations. The goal is not to make the world high-tech—rather, it's to accept that the world is heading in a high-tech direction and to attempt to nudge it toward a more humanist outcome.

I'm curious how you see a viewer/participant's perception of the kinds of spaces that your projects construct. Do you view the space that the viewer inhabits more like a medium in which you communicate with them, or do you see the space as a message unto itself?

Usman: The point is that the designer of any system is involved in a process of specifying things. If a designer specifies all parts of a design and hence all behaviors that the constituent parts can conceivably have at the beginning, then the eventual identity and functioning of that design will be limited by what the designer can predict. This entity is therefore closed to novelty and can respond only to preconceptions that were explicitly or implicitly built into it by the designer. If, on the other hand, a designed construct (say an environment) can choose what it senses (rather than having that imposed by the designer), either by having ill-defined inputs/sensors or by dynamically determining its own perceptual categories (that is, the means for comparing things), then it moves a step closer to true autonomy, which would be required in an authentically interactive system. In an environmental sense, the human component of interaction then becomes crucial because a person involved in determining input/output criteria is productively engaging in collaborative "conversations" with his or her environment. Building on the rather prosaic model of the thermostat, imagine a temperature-controlling system, which has the possibility of altering both inputs and outputs to the system (while always retaining the goal of "controlling the ambient temperature"). Rather than solely measuring temperature as an input, and outputting instructions to a heater to switch on, such a system might dynamically reconnect to different types of input or output.

These input criteria might range from "energy consumption measurements over the last month" to "the exterior temperature for this week last year" to "seasonal rainfall" to "the color of my clothes today" to "the fifth letter of the second paragraph on the front page of today's newspaper." The system might try all of these inputs and continue monitoring how much effect each appears to have on the goal (to control temperature). The system would evolve weightings for each of these input criteria in order to provide satisfactory output, again according to criteria determined dynamically with the person (who is controlling the goal of this super-thermostat). Output criteria might include "degree of thermal comfort," "energy bill amount," "neighbor's energy bill amount," "hot chocolate drinking tendency," "number of friends who come to visit and how long they stay."

In all cases, both input and output criteria are dynamically constructed and continually monitored with respect to how well they achieve the goal set by the person. In this situation we can imagine that the inhabitants and the environment are in some sense

collaborating to develop a temperature-controlled system, because not only is the person contributing the input criteria, but they are also involved in influencing the output criteria. Such a calculation is complex enough as to be pretty much impossible without the aid of the super-thermostat; but without the human contribution, it's also impossible.

Are you still working on the Low Tech Sensors *and* Actuators *project?*

Usman: The point of *Low Tech Sensors* and *Actuators* (a collaboration with Adam Somlai-Fischer) was to make it simple for nontechnical people (or people who thought they were nontechnical) to get involved in hacking toys and gadgets, not being too respectful of the packaging that a mechanism is found in, to be able to prototype responsive systems relatively easily. We didn't take it much further than the first manual (though we did hold, and occasionally continue to hold, low-tech workshops) mostly because it is now quite a popular concept, and so there are lots of such guides and workshops available.

The manual has led us to create the *Reconfigurable House* (which amassed hundreds of low-tech gadgets, devices, and toys in a system where occupants could determine how these devices were connected to each other, building interactive systems in which "they decide" how the house would respond to them—essentially a critique of the closed nature of "smart homes") and later *Scattered House*, which was like *Reconfigurable House* but split across multiple cities and networked using Pachube.com (a system I have been developing to enable people from around the world to connect up and share their devices and environments in real time in order to build planetary-scale interactive environments; it's a little like YouTube, but for sharing real-time environmental data rather than videos). Pachube has evolved from these earlier systems: attempting to foster a reconfigurable planet, if you will, through a kind of global generalized data brokerage.

Do you see architecture as a domain that is truly distinct from, say, designing an online community space?

Usman: I'm not sure I agree necessarily that interaction or participation are more meaningful when conducted solely in nonelectronic physical space; certainly, being in the same room with someone offers a lot more bandwidth, and when time dimensions change, it's possible to drift in and out to greater or lesser degrees. In online situations, of course, it's difficult to have degrees of opt out. Either you're part of a conversation or you're lurking (or you're not even part of the community at all). Skype gets a little closer by having different status conditions you can assign yourself to, paralleling the ability to be "in the next room" as it were. But participation is mostly provoked into being through systems that are open to be changed, and that's why we have the rich possibilities of something like Wikipedia. There are certainly parallels between the processes involved in constructing architecture and processes involved in constructing online spaces, though I'm hesitant to analogize since it feels as though there are already too many analogies used by each for the other. However, I'm not someone who believes in a great distinction between real and virtual—this distinction has been compared to the now-quaint 19th-century distinction between mind and body.

Reading about your 1,000 (little tips of communication) *project I came across a quote that really struck me: "At a time when we can build whatever we imagine, device, building, or experience, it is vital to consider the wider aesthetic, ethical, and lyrical implications that this condition affords us." Do you have specific ideas on what those implications are and what they mean to an artist, architect, or designer?*

Usman: The point is that when we can build/design almost anything, it becomes very important to consider why it is that we choose to bring any particular thing into this world: Does the world really need another XYZ product? What does it mean to be able to connect to more people? How can we continue to produce magic? In this statement, I was influenced a lot by Anthony Dunne's *Hertzian Tales* (MIT Press), where he talks about the "post-optimal" object (that is, objects you design once practicality and functionality can be taken for granted). In a world where anything is possible, poetry becomes imperative for generating the unexpected. I'm not sure that I can respond more specifically to your question, other than to say that that's really the hope for most of my projects: to find an answer to the question!

What inspired you to design the EEML?

Usman: Ultimately, with Pachube and Extended Environments Markup Language (EEML), my goal is to have some little effect on the construction industry, which determines the kinds of spaces we inhabit and how we relate to them.

EEML came about for two reasons: (1) we needed something that would do what current construction industry modeling standards cannot do (such as describe buildings that are dynamic, responsive, networked), while still being able to operate alongside them; and (2) because Pachube.com needed a protocol that was able to describe both the physical environments of buildings, devices, and human interaction and the virtual environments of *Second Life* and web browsers without predicating one upon the other. The point was to develop something that was robust enough to be used by building management software and also flexible and simple enough that individual developers and nonprofessionals can use the format in their own smaller devices and projects. This facilitates communication between all types of entity and therefore is exactly what Pachube.com requires, with its goal being to enable people to tag and share real-time sensor data between buildings, devices, and environments all over the world.

An understanding of what an environment actually constitutes is crucial to EEML's ongoing development. I believe that one of the major failings of the usual *ubicomp* (ubiquitous computing integrates common, everyday computational devices and systems, often without the user knowing) approach is to consider the connectivity and technology at the object level, rather than at the environment level. It's built into much of contemporary Western culture to be object-centric, but at the environment level, we talk more about context, disposition, and subjective experience. An *environment* has dynamic frames of reference, all of which are excluded when simply focusing on devices, objects, or mere sensors. If you really study deeply what an environment is (by this I mean more than simply saying that "it's what things exist in"), you begin to understand that an environment is a construction process and not a medium, not just a state and not just an entity. In this I would refer to Gordon Pask's phenomenally important text "Aspects of Machine Intelligence" in Nicholas Negroponte's *Soft Architecture Machine*

(MIT Press), which makes for extremely tough reading (Negroponte compared it in importance to Alan Turing's contributions to the discipline).

As a result of this conception of environment, we remove the need for a distinction between real and virtual. We can consider equally as environments a mountainside, the interior of a building, the context of a web page, the internal status and external context of a mobile device, the interactions within something like *Second Life*—all these are environments and can communicate with each other on equivalent terms. More importantly, a single environment can be expressed as a snapshot in time, or it can be expressed as a sequence of many snapshots over several years. Chris Leung (who has been working on this) and I have a very similar approach to systems design, but at the same time we are looking for quite different things in the protocol. And so, in ensuring that we are both happy with it, it has been useful to make sure that we cover a lot of territory—attempting to be all things to all people...and machines!

Using the X10 Protocol

X10 is a protocol used to send digital data between devices over household electrical wiring. This means that you can send data around a house using the wiring a little like Internet cables are used for TCP or UDP communication. It works by sending a single digit over the wire every time the waveform of the current hits 0 in its cycle, called the *zero crossing*. This looks like the images in Figure 17-6.

When the waveform is at 0, an X10 device can send a quick burst of data to any other connected X10 device. That data, once reassembled into a message, usually consists of an address and a command sent from a controller to a controlled device. That command could be telling the device to turn on or off or checking status, such as the dimming level of lights, the temperature in a room, the state of the coffee maker, or other sensor readings. Devices usually plug into the wall where a lamp, television, or other household appliance plugs in; however, some built-in controllers are also available for wall switches and ceiling fixtures.

The signal can't pass through a power transformer or across the phases of a multiphase system, so more complex electrical systems might not work with X10 as expected. There are devices that will help you work around this, but for the purposes of this quick introduction, we'll assume that you're in a space with a single-phase system where you're not attempting to cross any transformers. Most X10 modules fall into a few general types. *Controllers* send out an address and a command to control the receiver modules. Controllers do things like work as timers and interface with computers, telephone responders, universal transmitters, and alarm systems. *Transceivers* convert IR (infra red) or RF (radio frequency) signals to X10 signals, allowing you to easily set up controls using TV-style remote controls, wireless switches, motion detectors, and other means of wireless communication. There are also modules to control lights and equipment plug-in modules, built-in switches, micromodules, and modules for professional use.

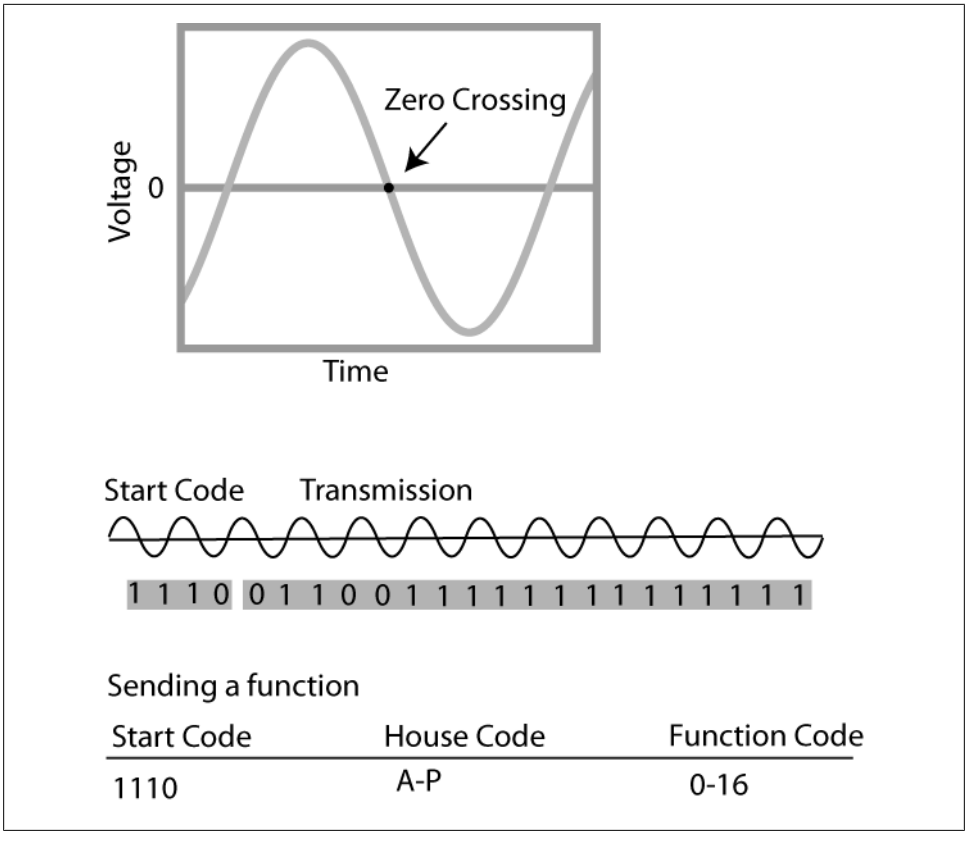


Figure 17-6. X10 sending data across the zero crossing of electrical current

The Arduino X10 library is built to send X10 commands through a unit that can be connected to the Arduino and provide access to a X10 network. The X10 protocol uses a few different constant values to send commands from one module to another module, and the Arduino X10 library includes constants that make working with these commands easier. The names of the constants should give you a good idea of what they're supposed to do.

```
ALL_UNITS_OFF   BRIGHT            PRE_SET_DIM
ALL_LIGHTS_ON   ALL_LIGHTS_OFF    EXTENDED_DATA
ON              EXTENDED_CODE     STATUS_ON
OFF             HAIL_REQUEST      STATUS_OFF
DIM             HAIL_ACKNOWLEDGE  STATUS_REQUEST
```

X10 also allows you to label and name different parts of an X10 circuit in a house, as well as naming different modules on each circuit, so that you can communicate with

an individual module by name. To control specific devices, all modules are assigned an address, which consists of a House code and a Unit code. There are 16 House codes (A through P) and 16 Unit codes (1 through 16). Each House code has 16 Unit codes, so this means there are 256 possible addresses. House/Unit codes are referred to like this: A5, C7, M13, P4, and so on.

In this book, we're most interested in using the Arduino with an X10 module, and that requires a module that the Arduino can interface with like the TW523, which is shown in Figure 17-7.

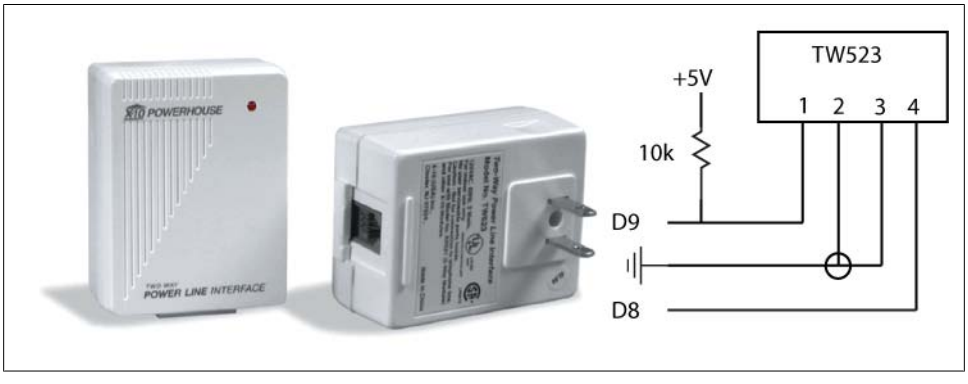


Figure 17-7. The TW523 X10 module

The TW323 X10 module can be connected to the Arduino as shown in the small schematic to the right in the figure. The Arduino X10 library will work with the PL513 one-way X10 controller and the TW523 two-way X10 controller. They simply provide a gateway into the X10 network that the Arduino can send signals to, much like a modem.

To make a simple application that writes an `ALL_LIGHTS_ON` message simply import the X10 library and call the constructor on the X10 object, passing the two pins that you'll be using for communication with the X10 module:

```
#include <x10.h>
#include <x10constants.h>
```

Call the constructor:

```
x10 myX10 = x10(8, 9);

int buttonPin = 5;
void setup() {
```

Set pin 8 to read input from the X10 controller and pin 9 to write to it:

```
    pinMode(8,INPUT);
    pinMode(9,OUTPUT);
    pinMode(buttonPin, INPUT);
    digitalWrite(buttonPin, HIGH); // turn on pull-up resistor
}
```

```
void loop() {
    int val = digitalRead(buttonPin);
    if(val == LOW) {
        // turn lights on if switch is pushed
        myX10.write(A, ALL_LIGHTS_ON, 1);
    }
    else {
        myX10.write(A, ALL_LIGHTS_OFF, 1);
    }
}
```

The possibilities that the X10 creates for you are in some ways similar to what the AC/DC switchers that you learned about in Chapter 7 enable, with a slight difference: X10 doesn't simply control devices that use household current; it uses household current to communicate with devices. You can use a few ways to identify things of interest occurring in a specific space: RFID tags, motion sensors, and pressure sensors in a doorway.

In the next section, you'll learn about RFID and see how to create an X10 network that sends signals when an RFID tag is read.

Setting Up an RFID Sensor

Radio Frequency Identification (RFID) is a technology that allows you to read and write to tags that can be read from a distance. RFID is used in credit cards, books, shipping containers of all sorts, and many other places. RFID is really a way to tag objects and physically port data. You can attach an RFID tag to any object to give it an ID, or if you are using an RFID writer, then you can both read and write to tags attached to an object.

Most RFID tags contain at least two parts. One is an integrated circuit for storing and processing information, modulating and demodulating an RF signal, and doing other specialized functions. The second is an antenna for receiving and transmitting the signal.

Some RFID readers can read tags from 6 meters away or more, but most have a short range of 5–10 centmeters. There are two kinds of RFID devices: readers and readers/writers. Devices that can write RFID data are generally more expensive. The frequency that a tag operates at is important because the reader and the tag need to be operating at the same frequency, for instance, 125Hz or 13.56MHz.

The Parallax RFID reader is a less expensive option for reading 125Hz RFID tags that has been used with the Arduino on many projects. Another Arduino-compatible option using the 125Hz tags is the Innovations ID-12 or ID-20 RFID readers. All of these can be powered from the Arduinos +5Vcc pin. Both the Parallax reader and the Innovations chips are shown, along with a breakout board for the ID-12 to make connecting it to your Arduino easier, in Figure 17-8.

Figure 17-8. RFID readers from Parallax and Innovations

If you're interested in working with writing RFID tags, less expensive RFID reader/writers are available. As of the writing of this book, APSX makes 13.56MHz reader/writers available for less than $100 that have been used with the Arduino.

To connect the Parallax reader or the ID2 breakout board to the Arduino, follow the schematic in Figure 17-9.

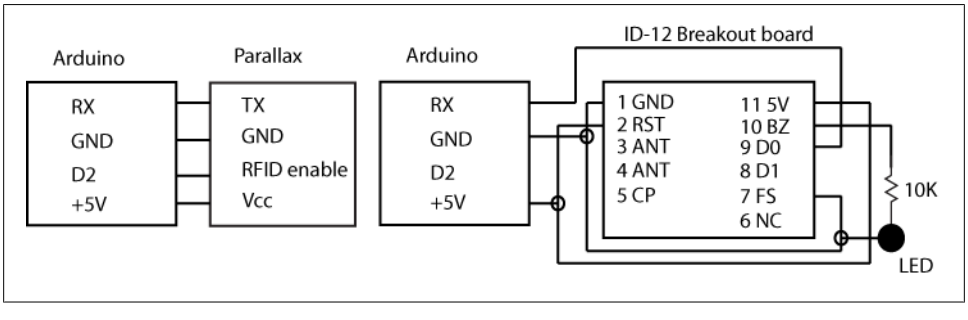


Figure 17-9. Connecting the Parallax RFID and ID-12 to the Arduino

Now you're ready to read data from the RFID reader. In the case of the Parallax RFID reader, this operates only at a baud rate of 2400:

```
int  val = 0;
char code[10];
int bytesread = 0;

void setup() {
```

Start communication between the RFID reader and the Arduino at 2400:

```
  Serial.begin(2400);
  pinMode(2,OUTPUT); // connected to the RFID ENABLE
```

Digital pin 2 is connected to the RFID enable pin and sending it `LOW` activates the RFID reader. If you wanted for any reason to deactivate the RFID reader, you would need to send it `HIGH:`.

```
        digitalWrite(2, LOW);
    }

        void loop() {
            if(Serial.available() > 0) { // check if there's data
```

The beginning of RFID messages from the Parallax will contain a header, so you'll know to continue reading the message. The Parallax RFID reader sends 10-digit tags, so you'll know that after 10 values, you've read the entire tag. There is also a stop byte sent as 0x13 that is sent if the tag is complete, so the loop created checks for that value:

```
            if(Serial.read() == 10) { // look for a linebreak
                bytesread = 0;
                while(bytesread<10) { // read 10 digit code
                    if( Serial.available() > 0) {
                        val = Serial.read();
                        if((val == 10)||(val == 13)) { // if header or stop bytes
                                                       // before the 10 digit
                                                            // reading
                            break;
                        }
                        code[bytesread] = val;
                        bytesread++;// ready to read next digit
                    }
                }
                if(bytesread == 10) { // if 10 digit read is complete
                    code(bytesread] = 0; // terminate the string
                    Serial.print("RFID tags code is: ");
                    Serial.println(code);
                }
                bytesread = 0;
                delay(500);
            }
        }
    }
```

If you're using the ID-12, the code is almost the same but with one exception: When checking the start and stop values for the ID-12, you'll use 0x02 for the start and 0x03 for the stop, so in the previous code you'll want to check for that value instead:

```
if(Serial.read() == 2) {
    // do the rest of the reading
```

You also could add some constants to define these values:

```
#define RFID_START 2  // Parallax start is 10, ID12 is 2
#define RFID_STOP 3    // Parallax stop is 13, ID12 is 3
```

One other thing to be aware of is that you'll want to make sure that you disconnect the RX serial wire from the ID-12 when uploading the sketch.

Now that you understand the basics of RFID, you can combine it with other technologies. You can use Firmata library or serial communication to read and write RFID data from a tag and send it to Processing or oF applications, or an Arduino can use the RFID data directly. In the following example, you'll combine RFID with X10 to turn the lights in a house on when the correct RFID tag is read. You could set this up near the front door of a house and turn your lights on simply by waving the tag in front of the reader.

This example uses one method that you probably have not seen before: the `strcmp()` method. This is a C method that is part of the standard C library, which means that you can use it in an Arduino application, an oF application, or any other C or C++ application. The method has this signature:

```
int strcmp(const char *s1, const char *s2);
```

The `strcmp()` method returns 0 if the strings are equal and a nonzero value if they are different. You'll get a positive value if the `s1` is greater than the second and a negative value if the `s2` is greater. In this example, that method is used to compare the RFID tag values to the constant that we're expecting. This example uses the following:

```
#include <x10.h>
#include <x10constants.h>
// RFID reader variables
#define TAG_LEN 12
#define RFID_START 2  // Parallax start is 10, ID12 is 2
#define RFID_STOP 3    // Parallax stop is 13, ID12 is 3
```

This is the value for the RFID tag that I'm using, but you'll need to change it to the RFID tag that you have:

```
char tag[12] = { "0F03037185"};
char code[12];
int bytesread = 0;
int ledPin = 13; // Connect LED to pin 13
int rfidPin = 2; // RFID enable pin connected to digital pin 2
int val=0;
```

Now, add some variables for the X10 control:

```
int repeat = 1;
boolean lightsOn = false;

x10 house = x10(8, 9);

void setup() {
```

Now, begin serial communications with the RFID reader. The `SOUT` pin of the RFID reader should be connected to Serial RX (Digital Pin 1) at 2400 baud. If you want to connect the RFID reader and a PC you can use the NewSoftSerial library to set up communication between the RFID reader and the Arduino and leave the hardware serial for computer to Arduino communication:

```
    Serial.begin(2400);
    // X10 Module
    house.write(A, ALL_UNITS_OFF, repeat);
```

Set the pins that the X10 unit will use:

```
pinMode(8,INPUT);
pinMode(9,OUTPUT);
```

Set the pins that the RFID reader will use:

```
pinMode(rfidPin,OUTPUT); // Set digital pin 2 as OUTPUT to connect it
                         // to the RFID /ENABLE pin
pinMode(ledPin,OUTPUT); // Set ledPin to output
digitalWrite(rfidPin, LOW); // Activate the RFID reader
}
void loop() {
```

This RFID-reading code assumes that the ID-12 is being used, but you just need to change the expected start and end values to use the Parallax RFID reader:

```
if(Serial.available() > 0) {
    if((val = Serial.read()) == 2) {
        bytesread = 0;
        while(bytesread<10) {
            if( Serial.available() > 0) {
                val = Serial.read();
                if((val == RFID_START)||(val == RFID_STOP)) {
                    break;
                }
                code[bytesread] = val; // add the digit
                bytesread++; // ready to read next digit
            }
        }

        if(bytesread >= 10)
        {
            Serial.flush(); // clear out the serial
            code(bytesread] = 0; // terminate the string
```

Use the `strcmp()` method to figure out whether the string matches:

```
            if(strcmp(code, tag) == 0)
            {
```

If lights are off, turn them on, and if they're off, turn them on:

```
                if (lightsOn == false) {
                    house.write(A, UNIT_1, repeat);
                    house.write(A, ON, repeat);
                    lightsOn = true;
                } else  {
                    house.write(A, UNIT_1, repeat);
                    house.write(A, OFF, repeat);
                    lightsOn = false;
                }
```

After you're done sending the X10 signal using the matched tag, turn the RFID reader on and off and then `flush()` the serial port:

```
                    Serial.flush();
                    delay(1000); // prevent readings from the same tag.
                } else {
                    Serial.flush();
                    delay(1000);
                    Serial.flush();
                }
            }

            bytesread = 0; // clear system and prepare for next cycle
            delay(500); // wait
        }
    }
}
```

Now you have an application that communicates with an X10 network as well as reads RFID tags. There's much more that you can do with an X10 network, but this should get you started with the basic controllers.

Reading Heat and Humidity

Another interesting and meaningful data point in an environment is temperature. Humidity and temperature are both fairly easy data points to read with an appropriate sensor, and they tell you a lot about a place or environment. Although you learned how to grab weather from online services in Chapter 12, there's another way that provides more localized data: using a temperature sensor. We'll look at two different sensors. First, the LM35 (Figure 17-10) is a heat detection chip that is inexpensive. It reads heat data and is quite easy to use. It's accurate within about 1.5 degrees C at the limits of its ranges of –55C and 150C, and it's accurate within about 0.5 degrees C between –25C and 100C.

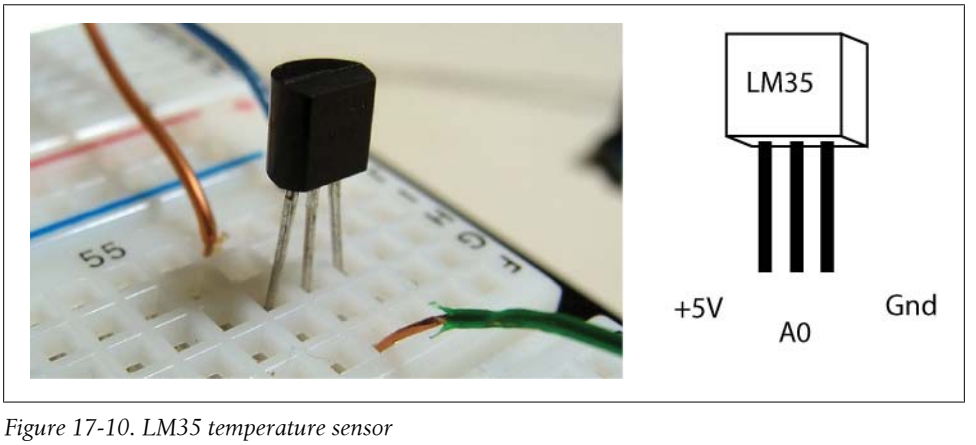


Figure 17-10. LM35 temperature sensor

The code to read the temperature is shown here and is based on some wonderful work by Daniel Andrade:

```
int pin = 0; // analog pin
int tempCelsius = 0, tempFahrenheit=0; // temperature variables
int samples[8]; // average of readings
int i;

void setup(){
    // this is where you'd start Serial communication if you want to use that
}
```

The data from the LM35 can be a little bit dirty, so it's best to take several readings and average them. This adds the values received from the sensor together and then divides by the number of readings to ensure that any odd readings are averaged out:

```
void loop()
{

    for(i = 0;i< =7;i++){
        samples[i] = ( 5.0 * analogRead(pin) * 100.0) / 1024.0;
        tempc = tempc + samples[i]; // add the  eight samples together
        delay(1000);
    }

    tempc = tempc/8.0; // this is the average of the eight samples
    tempf = (tempc * 9)/ 5 + 32; // converts to fahrenheit

    delay(1000); // delay before loop
}
```

To read both heat and humidity, a company called Sensirion makes a sensor called the SHT15 that can be used to read both temperature and humidity anywhere. It connects to the Arduino using the I2C protocol, which was introduced in Chapter 8, using a pin to control the clock and another to read the data sent from the SHT15. The SHT15 uses two commands, one to read temperature and another to read the humidity, so when you send the appropriate command, the chip will respond with the correct data. The trick is that these commands need to be sent in binary, as in the following code:

```
int temperatureCommand  = B00000101;  // command used to read temperature
int humidityCommand = B00000011;  // command used to read humidity
```

Note the B in front of the binary numbers. This tells the compiler that you're using a binary value to set the value of the integer. For the SHT15, the first three bits are the address, 000, and the last 5 bits are the command, so in the case of the temperature command, the actual command is 00101.

Next, declare the pins that will be used to communicate with the chip:

```
int clockPin = 2;  // clock
int dataPin  = 3;  // data
int error;  // to track whether any errors have occurred
float temperature;
float humidity;
```

```
void setup() {
```

Open the Serial port for communication with a listening computer. If you're not sending the data to a computer, then you don't need to do anything in this method:

```
    Serial.begin(9600); // open serial at 9600 bps
}

void loop() {
```

I'm a firm believer in the metric system, and so is the SHT15, so this application will read the temperature value from the chip and convert it to Centigrade. A conversion to Fahrenheit just requires changing the received value and was shown in the previous example:

```
    sendCommandSHT(temperatureCommand, dataPin, clockPin);
    waitForResultSHT(dataPin);
    int val = getData16SHT(dataPin, clockPin);
    skipCrcSHT(dataPin, clockPin);
    temperature = (float)val * 0.01 - 40;
    Serial.print("temperature: ");
    Serial.print((long)temperature, DEC);
    //Now we read the humidity
    sendCommandSHT(humidityCommand, dataPin, clockPin);
    waitForResultSHT(dataPin);
    val = getData16SHT(dataPin, clockPin);
    skipCrcSHT(dataPin, clockPin);
```

The relative humidity is calculated by taking the returned value and tweaking the value a bit to get a usable number that indicates the humidity in a percentage from 0 to 100:

```
    humidity = -4.0 +  (.0405 * val) + (-.0000028 * (val*val));

    Serial.print("humidity: ");
    Serial.print((long)humidity, DEC);

    delay(300000); // wait for 5 Minutes for next reading
}
```

This reads the values from the `dataPin` in between flashing the `clockPin`. Setting a pin `HIGH` and then `LOW` again is called *flashing* and, in the `shiftIn()` method here, is used on the `clockPin` to tell the SHT15 to send the next bit of data:

```
int shiftIn(int dataPin, int clockPin, int numBits) {
int ret = 0;

for (int i=0; i<numBits; ++i) {
    digitalWrite(clockPin, HIGH);
    ret = ret*2 + digitalRead(dataPin);
    digitalWrite(clockPin, LOW);
}
return(ret);
}
```

This method sends a command to the SHT15 sensor:

```
void sendCommandSHT(int command, int dataPin, int clockPin) {
```

Set the mode on the pins and then flash both pins, setting them `HIGH` and then `LOW` quickly, so that the chip knows that the transmission is about to start. The clock pin is set `HIGH` and then `LOW` two times, and the data pin is set `HIGH`, then `LOW`, and then back to `HIGH`. Now the SHT15 is ready to have data shifted out to it:

```
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    digitalWrite(dataPin, HIGH);
    digitalWrite(clockPin, HIGH);
    digitalWrite(dataPin, LOW);
    digitalWrite(clockPin, LOW);
    digitalWrite(clockPin, HIGH);
    digitalWrite(dataPin, HIGH);
    digitalWrite(clockPin, LOW);
```

As mentioned earlier, the commands have 3 bits in the address section and 5 bits in the command section. This uses the `shiftOut()` command, which shifts out a byte of data one bit at a time starting from either the most (left) or least (right) significant bit. Each bit is written in turn to the `dataPin`, after which the `clockPin` is toggled to indicate that the bit is available. This is known as *synchronous* serial protocol and is a common way that microcontrollers communicate with sensors and with other microcontrollers. The advantage of using this method of sending data is that the two devices always stay perfectly synchronized and communicate at very high speeds. In this code, the data is being sent with the most significant bit first. The third parameter, `MSBFIRST`, is an Arduino-defined constant, which means that the bit that determines whether a value is positive or negative is sent first:

```
    shiftOut(dataPin, clockPin, MSBFIRST, command);

    // check for errors
   digitalWrite(clockPin, HIGH);
    pinMode(dataPin, INPUT);
    error = digitalRead(dataPin);
    if (error != LOW)
        Serial.println("got an error 0");
    digitalWrite(clockPin, LOW);
    error = digitalRead(dataPin);
    if (error != HIGH)
        Serial.println("got an error 1");
}
```

In this next code snippet, the `waitForResultSHT()` method waits for the SHT15 to answer back by polling the data pin until it goes `LOW`. This is how the chip indicates to the Arduino that it's finished generating data and is ready to be read. If the pin does not go `LOW`, after 1 second we can assume that there's some sort of error with the chip and the reading isn't going to come back:

```
void waitForResultSHT(int dataPin) {

    pinMode(dataPin, INPUT);
    for(int i=0; i < 50; ++i) {
        delay(20);
        error = digitalRead(dataPin);
        if (error == LOW)
            break;
    }
    if (error == HIGH)
        Serial.println("got an error 2");
}
```

Getting data back from the SHT15 is a bit tricky. This next example follows the example code provided by Maurice Ribble and Hernando Berrigan to create and use a `shiftIn()` method, which more or less imitates how the Arduino `shiftOut()` method works except in reverse:

```
int getData16SHT(int dataPin, int clockPin) {
    unsigned int val;

    // get the MSB (most significant bits)
    pinMode(dataPin, INPUT);
    pinMode(clockPin, OUTPUT);
    val = shiftIn(dataPin, clockPin, 8);
    val << 8;

    pinMode(dataPin, OUTPUT);
    digitalWrite(dataPin, HIGH);
   digitalWrite(dataPin, LOW);
    digitalWrite(clockPin, HIGH);
    digitalWrite(clockPin, LOW);

    // get the LSB (less significant bits)
    pinMode(dataPin, INPUT);
    val |= shiftIn(dataPin, clockPin, 8);
    return val;
}
```

The SHT15 sends Cyclical Redundancy Check data. We don't need this data, so we want to tell the SHT15 that we don't want it:

```
void skipCrcSHT(int dataPin, int clockPin) {
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    digitalWrite(dataPin, HIGH);
    digitalWrite(clockPin, HIGH);
    digitalWrite(clockPin, LOW);
}
```

Now you can check the temperature and humidity for both inside and outside environments and use that data in your Arduino applications. You could use this for smart-home projects where the temperature of a house is adjusted according to the weather outside or a certain room is kept at a certain temperature. Connecting an SHT15 to an

X10 system allows you to engineer quite sophisticated reactions to shifts in temperature or humidity. A few projects that become easier with temperature sensing include monitoring a wine cellar, detecting weather, or monitoring chemicals for photography.

There are other ways to sense temperature as well. There is a Lilypad-ready temperature sensing module; there is also a One Wire Digital Temperature Sensor with the product number DS18B20 that uses the 1wire protocol to sense temperature data. If you don't need great precision, you can use very inexpensive 10k thermistors, a type of resistor that returns temperature values as voltage, that can return temperature data.

What's Next

There are a great number more environmental sensors to explore. To read the amount of light in a room, you can use a light intensity sensor to determine how much light is shining on the sensor, or you can use a light color sensor to determine what color the light in a room is. Both of these can be quite useful for determining what's happening or what factors are changing in a certain environment. Tutorials are available on the Arduino website for using the light color sensor. Another way to detect the color of light in an area is by using a chip like the PICAXE color sensor, which is very widely available online and gives you excellent readouts of light intensity and color.

You might want to look up the time in a location, which can be done using the DS1307 clock that returns the date and time without requiring that the Arduino run expensive calculations or that the Arduino always be connected to another computer.

If you're interested in having control over lots of lights or other devices commonly used in staging, there is a whole series of devices that use the Digital MultipleXed (DMX) protocol for controlling projectors, LED lighting arrays, dimmers, lighting boards, and other devices. DMX is commonly used in theater or dance productions where a single operator needs to have control over a large number of devices and there are many sophisticated lights and projectors that can be controlled via DMX. The Arduino can use a driver chip like MAX485 or 75176 to talk to DMX enabled devices, sending and receiving DMX commands. There are several tutorials posted on the Arduino website that include code, wiring diagrams, and more technical information on the DMX protocol.

Review

ZigBee is a standard for communicating over wireless networks that is designed to be inexpensive and to not require a lot of power, making it perfect for small devices and ubiquitous computing.

Depending on the model, an XBee can communicate up to 100 meters indoors or outdoors within line of sight, a maximum of about 2 kilometers, and if you use of the high-gain antennas, you can send signals up to 6Km.

XBee controllers can be used for point-multipoint communication or for mesh networks.

There are several different types of XBee units. Different XBee modules are not always compatible, so make sure to either use the same type of module or check the compatibility.

AT commands are used to set the destination, ID, or other properties on an XBee and can be sent from the host computer or from an Arduino.

The XBee Processing library allows you to route information directly from an XBee to a Processing application and parse the information as well as send commands. It also helps you parse XBee data frames into all the respective parts.

Both ultrasonic and infrared sensors can be used to detect movement and the distance between an object and a sensor.

The Maxbotix ultrasonic sensor can be chained so that multiple sensors will read without causing interference with one another, which lets you use multiple sensors to place objects in two dimensions or to detect range over a larger part of a space or room.

X10 is a protocol used to send digital data between devices over household electrical wiring. The Arduino X10 library can be used to simplify sending and receiving data.

The X10 protocol can address different devices on a network and has specific commands to communicate with the system available.

RFID is a technology that can read tags that have a specific address written to them and send that information to an Arduino. There are also RFID writers that can write to a tag.

RFID tags are often 10-digit ASCII values that can be used to identify a tag, if you're looking for a specific tag, you simply need to read the 10 digits from the Serial port and compare them to the value that you're expecting.

You can read temperature using a device like the LM35 or a sensor like the SHT15 to read temperature and humidity data.

CHAPTER 18
Further Resources

Throughout this book, you've learned about the basics of code, hardware, and ways of designing and programming interactivity in applications, objects, and art pieces. Given the wide scope of this book and the limits of bookbinding, a great number of topics were only cursorily introduced. You'll almost invariably find yourself wanting more, and you should; there is a world of programming techniques, components, and different ideas to learn and explore. Your next steps are probably going to be driven by what you want to make and do with code and hardware. This last chapter includes some resources that you might be interested in exploring and some further ideas that you can use as a jumping-off point as you work on your projects. It also includes a short list of manuals and books that you might find beneficial in helping you understand a topic introduced in this book, that might inspire you, or that can help you find solutions to a problem you've encountered.

What's Next?

This section contains pointers to topics you might be interested in exploring further. A lot of tools and projects exist for artists and designers working with code and hardware. We've covered the three platforms that I prefer and that I think have best user communities and are in most widespread use right now, but you might want be aware of others as well. Some of these are being integrated with the Arduino, Processing, or openFrameworks projects, and some are entirely separate projects.

Software Tools

Beyond the tools that we've examined in this book, the following tools may be of interest to you.

ARToolKit

ARToolKit is a software library for building augmented reality (AR) applications. These are applications that involve the overlay of virtual imagery on the real world. Augmented reality composites real-time video and computer-generated imagery to create the appearance of an object that exists in the real world. ARToolKit was developed at the University of Washington and has already been successfully used in projects made with openFrameworks. The potential of AR as an assistive tool, as a tool for learning, as a visualization aide, and as a way to perform tasks that are too small or too dangerous for a person to do has already been demonstrated in commercial, industrial, and educational applications. There is also a massive potential for using AR in gaming applications, particularly in locative or physically reactive games, since a layer of computer graphics representing additional information or a virtual reality can be overlaid atop a player's natural perception. You can find more info at *www.hitl.washington.edu/artoolkit/*. There are other AR projects currently in development like the Parallel Tracking and Mapping (PTAM) project at Oxford University that you may be interested in looking into as well.

PureData

PureData (PD) is an open source graphical programming environment for audio, video, and image processing that, instead of using text-based programming like Processing, C++, or Arduino, uses a visual patching environment to create applications. PD coding is done by connecting elements in a graphical environment rather than by text-based input. PD is often called a *patching* language because each application can be used in another element or patch in another application. An application that reads the microphone of a computer, sends it through a series of filters, and then outputs it to the speakers can be run alone, or it can be used as an element, or *patch*, in another application. This idea of patching means that Pd development often consists of developing multiple patches and then combining them in different orders or with different parameters to create an application. PD was originally developed by Miller Puckette and company at IRCAM and has an active development community that is building out the core of PureData as well as plug-ins and extra patches for it. For instance, the development community has created a system of abstractions for building performance environments, a library of objects for physical modeling, and a library of objects for generating and processing video in real time.

PD shares a lot of core concepts with Max/MSP. PD was created to explore ideas of how to further refine the Max paradigm while allowing data to be treated in a more open-ended way and opening it up to applications outside of audio and MIDI, such as graphics and video. It is easy to extend PD by writing object classes (externals) or patches (abstractions) in C. PD is free software and can be downloaded either as an OS-specific package or as source code. It's also possible to write externals and patches that work with Max/MSP and PD (see Figure 18-1). You can find more info at *http://puredata.info*.

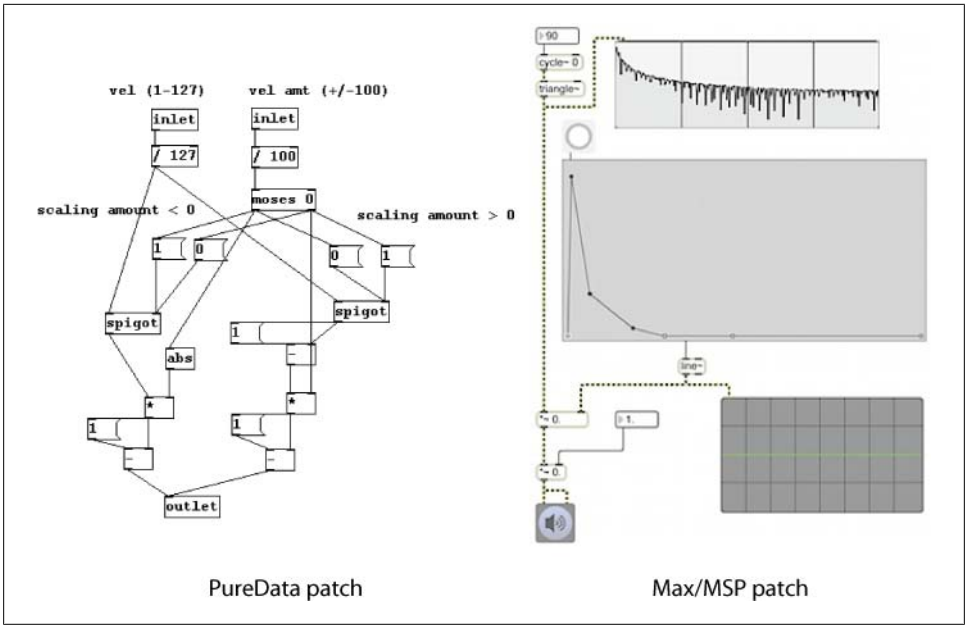


Figure 18-1. PD patch and a Max/MSP patch

Max/MSP

Max is a graphical development environment for music and multimedia developed and maintained by San Francisco-based software company Cycling '74. It is used by musical composers, performers, software designers, researchers, and artists to create music, stand-alone applications to be used in live performance or as part of a larger system. Max can use many different libraries and that creating and incorporating libraries is quite easy. Like the other programming environments that we've explored, Max has a large community of programmers who enhance the software with commercial and noncommercial extensions. Like PD and vvvv, Max/MSP uses a graphical programming interface. There are many libraries that allow Max to be extended into realms far beyond music. Jitter for video processing, the Lemur panel allows you to create novel musical interfaces, the Monome project can be integrated with Max; and a whole series of physical inputs and outputs can be integrated with Max/MSP applications. Unlike the other projects presented in this book, Max/MSP is not free and open source, but it does provide a level of support, libraries, and documentation that make it worth its price.

vvvv

vvvv is graphical programming project that shares a few similarities with both PureData and Max/MSP in that it provides a visual programming interface rather than a text-based programming interface. It can be used to provide feedback for physical interfaces, generate real-time motion graphics, work with audio and video, and create

high-performance applications that can interact with many users simultaneously. vvvv is also very powerful when used for live coding, that is, writing code as it is in front of users. This isn't something that can be done as well in a platform like Processing where the code is compiled and then run, but since vvvv is always compiling its code into graphics, it has one mode—runtime. Its primary limitation is that it runs only on Windows because the graphics code is built in the Windows graphical programming language DirectX. It's a free and open source project and, if you're working with Windows applications, you may find it an excellent tool.

Flash

Adobe Flash is a very popular tool mainly used for web applications and animations, but it can also be used for creating desktop applications. The name *Flash* refers to a few different things: a runtime player that plays Flash files once they've been compiled and an IDE that can be used to design animations and write code. Flash can create vector graphics; load, display, and manipulate images, videos, and other Flash animations; play sound files; and communicate with other applications online and offline. Flash Player is widely used for watching videos online, creating web-based games, and displaying advertisements. Flash uses ActionScript, a programming language that has some resemblance to Java. While you need to buy a copy of the Flash IDE to use the Flash animation tool, the Flash compiler that compiles ActionScript into Flash applications is available for free from the Adobe website.

Construction Processes

Any project that creates 2D images can transfer those images onto physical objects ranging from traditional printing on paper to laser cutting on wood to printing onto t-shirts. The Processing core libraries include tools for creating shapes, loading shapes from vector graphics files, and saving them back out so they can be used with most graphics programs and most types of printers. Laser cutters are other kinds of printers can be used to etch graphics into wood, vinyl, or nearly any other kind of material both in relief and in print. Graphics that are generated algorithmically in response to data or user actions and modified as part of an interactive process can be used to create prints. These more traditional ways of creating 2D output are not the entire extent of what can be done with 2D output, though. For instance, rAndom International created a project called PixelRoller that used the head of a printer to create a paint roller that prints images as it runs over a surface.

Another technique being explored more and more frequently in art projects is 3D printing. 3D printing is a unique form of printing that is often used in rapid prototyping. A 3D model of an object can be created and used to "print" a 3D object by layering and connecting successive cross sections of material. On a Windows computer, you can use two libraries, OpenGLExtractor (OGLE) and GLIntercept, to export 3D models from Processing and then have a 3D-enabled printer create models from your

application. Marius Watz and Casey Reas, among others, have explored generating algorithmic graphics and creating models from them. Another interesting extension of this technology is the RepRap project, which seeks to create machines that can reproduce any part of themselves using 3D printing technology.

Artificial Intelligence

In the 1970s, artificial intelligence made great and broad promises that computer science would deliver thinking machines that would learn, communicate, and consider just like human beings. While these haven't come to pass in quite the way originally conceived, there are a still a great number of technical advances that involve the creation of programs that can come up with sophisticated ways of making decisions. If this interests you, then you can begin by looking into two libraries for Processing: the Alice library and the AILibrary create by Aaron Steed. Here's a look into a few of the technical advances in artificial intelligence.

Neural nets

Neural nets are a technique for creating decision-making engines. Neural nets function in ways somewhat similar in the way that human brains function: As they attempt tasks and are corrected, they are told whether their actions are correct or incorrect, and the neural net instructs itself to make the correct decision given the same input the next time. This process is called *training*, and as the neural net is trained, it structures the nodes that the neural net is comprised of with prejudices towards a certain response.

Pathfinding

Pathfinding is another common task that artificial intelligence applications need to perform. Given a set of obstacles, an origin, and a destination, the program must find a path from one location to another. This is a very common task in games or applications where there are *sprites* that act and move independently. While this isn't pure artificial intelligence in the strict sense, algorithms like the AStar algorithm or the Diikstra pathfinding algorithm provide ways for programs to make decisions. You can also take a look at the work that Daniel Shiffman has been publishing in advance of his book *Nature of Code* at *http://www.shiffman.net/teaching/nature/*.

Genetic algorithms

Genetic algorithms are a conception of programming and creating solutions to problems inspired by Darwin's theory of evolution. Solutions to a problem solved by genetic algorithms are evolved over time by a program that checks which solutions come closest to solving the problem. Solutions that are selected to form new solutions (*offspring*) are selected according to their fitness—the more suitable they are, the more chances they have to reproduce.

Artificial life

AL is another area that falls roughly under the category of artificial intelligence applications. AL has been used widely in biology and sociology to make predictive models of how different species will respond to different changes in climate, food supply, and population. Many artists and engineers have used some of these algorithms and techniques to create ecosystems of independent agents that can be controlled or affected by user input, environmental changes, or other stimuli. Some of the most interesting work on this topic has been done by Karl Sims, his website *www.karlsims.com/*, is worth a look if you're at all interested in this topic.

Interview: Daniel Bisig

Daniel Bisig received a Ph.D. in Protein Crystallography at the same university. In 2001, he joined the Artificial Intelligence Laboratory at the University of Zurich as a senior researcher. He has also been working as a research associate at the Department of Art and Design, University of Applied Sciences, Aargau in 2003 and at the Institute Cultural Studies, University of Art and Design, Zurich in 2004. Since 2006, he has an additional research position at the Institute for Computer Music and Sound Technology in Zurich.

Some of his projects include the Interactive Swarm Orchestra installation that generates artificial life and sound in response to user actions, the *BioSonics*, which also generates complex life systems, images, and sound in response to user movements.

Do you have a clear vision of how an interaction with a machine can help humans understand themselves?

Daniel Bisig: The computer is a machine that excels in its capability to simulate reality. Simulations represent both abstracted and mediated forms of reality that mirror our notions and preconceptions about reality. Simulations of living systems that are important in the scientific field of artificial life confront us with our self-awareness as living beings. The observation of artificial creatures that seems to possess drives and fend for themselves challenges our notions of what constitutes the peculiarities of natural organisms. What is unique about being alive? What is unique about being human? These simulated realities provide an excellent environment for interactive experimentation and exploration in order to help us understand the idiosyncrasies of our own natural reality.

Can you explain a little bit about why you made all the code for ISO open source on the i-s-o.ch website? What sorts of things do you hope people will do with that code?

Daniel: The project Interactive Swarm Orchestra (ISO) attempts to establish a public platform for discussion, experimentation, and realization within the field of swarm-based art. I believe that swarm-based art still possesses a lot of untapped potential and that the advancement of this field benefits from an open exchange of conceptual ideas, technical implementations, and artistic intentions. Making all ISO-related software developments open source forms an important part of this strategy. The software libraries that are available on the ISO website provide the necessary functionality to realize software applications for swarm-based art. ISO-Flock constitutes the most important library in that it supports the implementation of swarm simulations. A swarm

simulation is supposed to act as a generative system that communicates with other applications that deal with interaction and feedback. It is possible to use the swarm simulation library as a black-box system that provides standard swarm behaviors as an underlying mechanism in a swarm-based art piece. Unfortunately, this form of application is not particularly interesting for the ISO project. I would prefer if artists devise and implement new agent behaviors and swarm interactions that are specifically adapted to their own artistic goals. This form of software customization and development that is driven by artistic ideas constitutes an important contribution and matches the transdisciplinary scope of the ISO project.

Do you see the challenges for the design of the interaction in these sorts of situations as presenting different design challenges than, say, a simpler system? What do you think about how the participant/user understands what is occurring in the system?

Daniel: I believe that the design of autonomous artifacts should encourage collaborative forms of interaction that are based on an intuitive understanding of the artifact's behavior at the cost of the amount of control that the user can exert. Such an understanding can arise via the creation of a language of interaction, a set of nonverbal cues based on the artifact's appearance, and behavior that allows the users to deduce the system's current internal state and activity. This design of a body language, and its possibilities for interpretation, represents a novel form of affordance. The analysis of interaction between humans and their animal pets might provide an important source of inspiration for this design challenge.

In reading some of your writings, I get a very strong sense of this concept of a symbiotic relationship between humans and AI systems. Is that accurate, and if so, can you talk a little bit about that?

Daniel: This impression is indeed correct. As design methodologies from AI and in particular from AL become more prevalent in artifact design, these artifacts cease to function as mere tools but adopt characteristics of natural organisms and therefore possess their very own needs, drives, and goals. For this reason, new forms of relationships between the AI systems and humans need to be explored, such as relationships that no longer solely emphasize human interests but that also benefit the AI system. Such relationships possess aspects of symbiotic relationships known from biology.

Your primary academic studies were in natural sciences. Can you explain a little bit about how you began working with interactive art and systems design?

Daniel: It is true that my primary background is in the natural sciences, or more accurately in molecular biology. Throughout my PhD, I became interested in using software to model the structure and behavior of proteins, first as a user of such software and later when developing my own software. Once I finished my PhD, I acquired a second background in web design. At that time, I was particularly interested in building dynamic websites to devise algorithms to automatically create designs and structure content. I believe that my involvement in interactive art and systems design is an attempt to combine these backgrounds. To me, computer simulations of natural systems provide interesting forms of interaction and exploration with mediated forms of reality. These simulations represent an algorithmic means to grasp some of the complexity and diversity of natural systems. They can form an important part of an interactive and

dynamic artwork in that they serve as a source of richness and surprise. The aesthetic output that is driven by these simulations exhibits a naturalness in behavior and spatiotemporal patterns that shows even through layers of abstraction.

You mention in your paper on ISO (iso_icmc_demo_2008.pdf) that you're interested in working with gestural interfaces and in connecting them with the ISO project. Do you have any views on how this would change the project and the experience of it that a viewer would have? Do you have any views on how this gestural interaction would be designed and implemented?

Daniel: While creating interactive simulations, I was constantly facing a similar challenge: how to create an immersive experience and intuitive proximity between two totally separated worlds: the real world of the viewers and the digital world of the computer simulation. We had to confront how we would penetrate or remove the screen that is a brick wall between user and simulation. Our current plan is to create interfaces that possess both a physical presence and a virtual presence. They exist in the real world of the user and in the virtual world of the simulation. The physicality of these interfaces provides feedback modalities that can help to merge the physical and virtual. For example, the interface may resist being manipulated, and the amount of resistance depends both on the interfaces physical properties (mass and friction) and on its virtual presence being blocked by an obstacle in the simulation such as a moving swarm. In addition, the interfaces should be realized as gestural interfaces that can be operated via normal scale body movements (arm movements, walking, and so on). I believe that interfaces that combine mixed feedback modalities and gestural characteristics increase the naturalness and intuitively of interaction. And this again serves a prerequisite for the creation of immersive spaces that promote a feeling of a shared presence between viewer and artificial system.

The contemporary notion of complexity is quite heavily rooted in biology, sociobiology, mathematics, particularly information sciences, and artificial intelligence studies as well. Do you see working with these ideas more as a way to explore them further or more as a way to explain them to an audience?

My main motivation is to explore complexity from the point of view of interactive art to complement the complexity sciences with an artistic research focus. This focus deals with the following questions: what are appropriate forms of interaction with complex systems? What is the aesthetic expressivity of complex systems? How do complex systems and artificial forms of creativity interrelate? How can complex systems and humans engage into shared forms of creativity?

But I have come to realize that most viewers are entirely unfamiliar with interactive systems that behave in a complex fashion and whose response to user input is harder to predict and reproduce (as opposed to simple and purely reactive systems). For this reason, the role of the interactive artworks that I'm creating has really become twofold. They possess educational aspects in that they let visitors explore and experiment with complex systems and thereby develop an awareness for the promising capabilities of these systems. And the installations are products of my artistically and scientifically motivated research.

You've been working with flocking and swarming behaviors in several different projects. Can you explain a little bit about why you return to this theme again and again?

Daniel: I'm fascinated by swarm behavior since it represents one of the most fundamental forms of group behavior and because it exemplifies very nicely how the combination of simple individual behaviors can give rise to complex phenomena. These individual behaviors can in fact be abstracted to something so simple that they can even be exhibited by purely physical systems (a principle that is appreciated, for example, by physical models of social phenomena). Swarm simulations therefore create a fascinating link between physical and biological phenomena. Since swarm simulations model social phenomena, they are particularly attractive for mixed reality applications that combine real and simulated participants. Furthermore, swarm simulations are very flexible. Swarms can be easily extended with novel behaviors and social relationships. Swarm simulations can therefore be adapted to achieve specific aesthetic and interactive capabilities and functionality. So, to wrap it all up, swarm simulations are a paradigmatic example of a complex system, they lend themselves to interaction, and they can easily be expanded and adapted for particular artistic goals.

What sorts of challenges do you face when trying to have an audience interact with such a complete and complex system?

Daniel: Most visitors are not prepared to interact with autonomous systems. Complex systems are driven by underlying processes that are active regardless of any interaction, and visitor input never dominates the system but rather constitutes only a single aspect in this web of interrelated processes. As a result, visitors are often baffled by the fact that the system might respond differently to the same repeated input or it might respond only gradually and after some time has passed.

So, the challenge is to acknowledge these user expectations without sacrificing the unique potential for surprise and diversity that complex systems possess. I can't really claim that I have succeeded in this. I usually try to offer two levels of interaction to the visitor: an interface-level interaction that provides immediate feedback via the handling of the interface itself and a system-level interaction that emphasizes the autonomy and complexity of the system.

I've read that you had a difficult time getting the audience to use sounds to interact with the system. Can you explain a little about how you've approached that?

Daniel: It seems that sound-based interaction is hampered both by the fact that audio is rarely used as interaction modality and that art exhibitions represent silence-inspiring environments. So far, I've shied away from the obvious but blunt solution, which would be to hang some written instructions next to the installation. Rather, I try to realize an artwork in such a way that it combines several interaction modalities. For example, in my piece *BioSonics*, interaction combines both camera-based tracking of the visitors' hands as well as audio capturing and analysis. By offering familiar forms of interaction, I hope to motivate visitors to spend some extended time exploring the behaviors of the installation and thereby help the visitors to discover by themselves the less conventional but more rewarding forms of interaction.

A very interesting idea to me is how you work with mutation and selection that is in some way driven by the interaction of the viewer. What kinds of modeling techniques or ideas do you use to enable this kind of interaction and ensure that the user is aware of the system and their effect on it?

Traditionally, interactive evolutionary systems let the users manually assign fitness values to individuals in an evolving population based on their subjective aesthetic preferences. I don't like this approach very much since it creates a passive and user-driven artificial system that demotes evolution to the level of a sophisticated randomizer. Much more interesting approaches implement principles of implicit natural selection that guide the evolution of an artificial biotope. Here, evolution is based on inherent and implicit processes that determine the probabilities of survival and adaptation of artificial creatures. The probability that a virtual creature can produce offspring depends on how well it is able to cope with the artificial biotope environment. In such a system, the user can interact in a much more intuitive and varied fashion than in an aesthetic selection system. By influencing the probability for change and survival (for example by caring for particular creatures and breeding them), the user's actions become part of the evolutionary process but don't dominate it. Many of the most famous and impressive artworks that employ artificial evolution, such as early works by Christa Sommer, Laurent Mignonneau (*A-Volve*, *LifeSpacies*, *PicoScan*) and Mauro Annunziato (*Relazioni Emergenti*, *E-Sparks*), are based on this approach.

What artists or designers do you draw inspiration from? Is this somehow different from the sort of inspiration that you draw from research or more theoretical ideas?

Daniel: I'm very inspired by the artistic realizations and publications by several people who are active at the intersection of artificial life and interactive art. Christa Sommerer, Laurent Mignonneau, Mauro Annunziato, Jeffrey Ventrella, and Jon McCormac are the names that immediately come to my mind. Of particular interest are those people's contribution to explore novel interaction concepts that relate simulated processes and aesthetic feedback and that establish mutual relationships between the inhabitants of physical and virtual spaces. It is interesting that all these people possess a background in art and science/engineering. Accordingly, their projects are of equal relevance to me on conceptual, scientific, and artistic levels. While the activities of this mixed science/art communication are essential for my own work, I also maintain a keen interest in the synthetic natural sciences (such as artificial intelligence and artificial life). I'm fascinated by their engineering-based approach to identify and model universal biological and social phenomena. And I'm also highly influenced by their research methodology that interweaves conceptual and practical activities and therefore becomes a form of experimental philosophy.

What ultimately do you see as the limits of artificial life systems? What can we learn from them? Is there a limit to what can be modeled or what could emerge from a system?

Daniel: At the moment, all artificial life systems lack the diversity and adaptive capabilities of their natural counterparts. In the AL community, there is no agreement about what fundamental properties that natural organisms possess are missing in artificial systems. With regard to simulation-based AL systems, one possible limitation might result from too simplistic modeling of the virtual habitat the artificial creatures exist

in. According to this explanation, simulations fail to mimic the biophysical world at a sufficient level of richness and diversity that would be necessary to give rise to the wide diversity of phenomena that we observe in nature. Rather, simulations tend to be too simplistic and thereby restrict the potential of natural adaptive processes such as evolution and learning. For this reason, some groups in AL move into robotics or chemistry and therefore begin to create artificial systems that exist in the real world. It is still too early to say anything about fundamental limitations of these type of artificial systems since this approach is at least for the moment very much hindered by technological restrictions of challenges.

I believe that AL systems have been very instructive with regard to our understanding of the relationship between simplicity and complexity. They have shown us that simple processes often give rise to very complex phenomena if these processes happen concurrently and influence each other. This principle of complexity from simplicity abounds both in the living and nonliving worlds. And it is this principle that reveals the existence of a fundamental relationship between complex physical, biological, social, and cultural phenomena. For this reason, AL is one of the most fascinating and promising transdisciplinary enterprises that benefits from the contributions of both scientists and artists.

Physics

Any time you want to create physical interactions between elements in your application that appear to be realistic, you'll need something to mathematically calculate the results of those interactions, whether they're gravitational effects, collisions between elements, or a complex combination of all of these. You can write that code yourself, but it's often easier to use an existing library and modify it to your needs or to at least base your code on an existing library. A number of excellent physics libraries are available for both Processing and oF that you might want to investigate.

Chipmunk

Chipmunk is a 2D rigid body physics library created by Scott Lembcke. *Rigid body* means that the library models interactions only between elements that do not bend or transform when they collide with another element. It is fast, numerically stable, and easy to use. For many of the most common tasks you'll want to do in modeling physics, Chipmunk can be quite helpful. For example, rotating rigid bodies is quite easy in Chipmunk. You can also model colliding shapes and define the shape of an element by attaching multiple shapes to it. You can attach many shapes to a single body to define a complex shape, or none if it doesn't require a shape. You can attach joints between two bodies to constrain their behavior, for modeling an object like an arm or leg. The basic simulation unit in Chipmunk is called a *space*, and you add bodies, shapes, and joints to a space and then update the space as a whole, making the update process quite simple.

Box2D

Box2D is a feature-rich 2D rigid body physics engine, which is used for modeling the rigid body interactions in Chipmunk. It is written in C++ by Erin Catto so you could plug it into an oF application with no changes required, and it has been ported and is available for Flash, Java, C#, and Python, among others. For Processing, you can use the BoxWrap2D library, created by E. W. Jordan. As with Chipmunk, any time you want to model interactions between rigid bodies, you might want to at least look at the Box2D engine. It is used widely in many games for many different platforms and in many other interactive applications, including the award-winning and remarkably fun Crayon Physics.

Other Processing physics libraries

For working with physics in Processing, you can use library called Physics that was created by Jeffrey Traer Bernstein and has been used in countless projects. At its core, it's a simple particle system physics engine that generates particles, springs, gravity, and drag on the sprites. It is extremely simple to use and more than sufficient for many uses. There is a port of this over to oF as well that you can use. In addition to this library, a quick glance at the Processing website will show you other libraries you can use to model physics.

ofAddons

Several add-ons for oF have been developed for working with physics in three dimensions in particular. The ofxMSAPhysics library is a powerful add-on developed by Memo Akten to help in modeling physics equations. There is also the ofxMSASpline add-on and a series of fluid equation solvers that he has developed and shared with the oF community.

Hardware Platforms

Arduino is not the only hardware prototyping platform available, there are other projects under development that you might be interested in exploring. Some of the following projects provide similar functionality to the Arduino and others, like the Fritzing project, can be used as a supplement to working with Arduino.

Phidgets

Phidgets are a line of plug-and-play building blocks for physical computing that can be plugged into a computer using a USB cable and set up to communicate with any application. All the USB complexity is handled by the Phidgets API, which enables you to create communication between applications with a minimum of fuss. Since Phidgets are so simple to set up and use, you might consider working with them if you require only a single device for physical input or output. You can't create physical computing applications with the same complexity as you can with Arduino, but your project may

not require that level of sophistication. Phidgets provides Python, Java, and C APIs, which means that connecting the Phidgets API to a Processing or oF application can be quite easy. The bottom level of the Phidgets stack is a cross-platform library, which implements the low-level protocols necessary to communicate with the Phidgets API and exports a unified interface to the software programmer. Built upon this low-level library are higher-level libraries that simplify using Phidgets for many more languages. These higher-level libraries contain logic for interfacing with the C library, thus making maintenance much easier.

Robotshop Rover

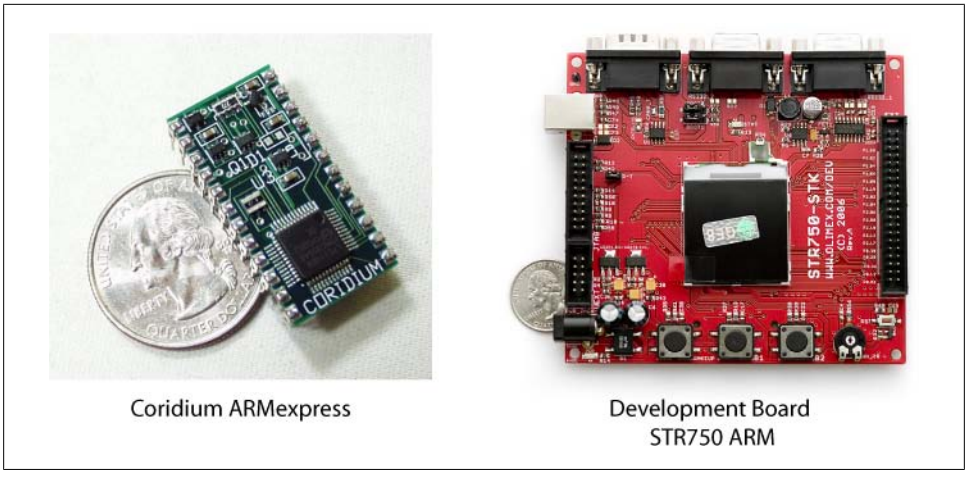
Robotshop Rover is a robot kit built to work with Arduino Duemilanove; it includes one robot in the kit as well the necessary electronics and hardware to build a fully functional robot that can be programmed in Arduino. It does require some soldering to get the robot working, but you can easily create a fully functional robot with two DC motors. You can also customize the robot by adding new electronic parts. For example, a pan and tilt kit is included with the kit and can be used to mount additional sensors or a camera and can even act as the base for a multidegree-of-freedom robotic arm. The Pololu Servo Motor Controller mentioned in Chapter 11 can be used with Robotshop Rover to control Servos as well.

Fritzing

Fritzing is an open source initiative to help designers and artists create actual products from their prototypes. Fritszing consists of software to help design and document circuits and turn them into PCB Printed Circuit Boards (PCB) layouts for manufacturing. Fritzing is what is called *electronic design automation* software that helps create circuit boards. The interface of the software uses the metaphor of the breadboard, which makes it easier to sketch hardware to the software. You can create a prototype using a breadboard and Arduino, create it in Fritzing, and then create a file that can be sent to a manufacturer to produce a working PCB. While Arduino or some of the other physical prototyping tools lets nonengineers quickly turn their ideas into functional interactive prototypes, Fritzing aims to help the designer move to the next stage and create a finished PCB of their individual circuit in the desired shape. This makes the circuit robust and is great for creating permanent installations or even batch production of a project.

AVR

You may remember from Chapter 4 that the Arduino uses a microcontroller called *AVR*, but it's not the only project or device that uses these controllers. In fact, if you're interested in working with different kinds of controllers other than the Arduino while leveraging some of your Arduino knowledge, you might consider working with other AVR controllers. The two libraries that you'll probably want to look into first are the AVR Libc library, which contains the version of the C programming language that you can use with AVR microcontrollers, and the AVRDUDE library, which is a program



Coridium ARMexpress

Development Board
STR750 ARM

Figure 18-2. Two ARM development boards

for downloading and uploading programs onto AVR microcontrollers. You can compile C code for the AVR microcontroller using avr-gcc, which is a version of the Gnu Compiler Collection that, depending on your OS, you might be using to compile an oF application. You can look at many of the electronics supplier sites and find different AVR boards and programming kits that might be of interest to you.

ARM

ARM processors are increasingly being used in a wide range of platforms from *net books* (small computers meant for using the Internet or light word processing) to tiny embedded controllers. You can get ARM boards from many online electronics that will include pins for communicating with the microcontroller as well as documentation. Figure 18-2 shows two small examples.

For more information, you can look at *www.arm.com* or check an electronics supplier for information on a particular chip or board.

PIC

The PIC is another series of chips that are power efficient, are simple to program, have many tools, and are aimed at less experienced developers. PIC controllers are used in a very wide range of projects. The simplest chip to program, find code for, and learn on is the 16F84, but there are other chips as well, namely, the 16F877 and the 18F series. You can find tutorials online or in a book such as *Physical Computing* by Tom Igoe and Dan O'Sullivan (Course Technology) that will show you some techniques to use. *Designing Embedded Systems with PIC Microcontrollers* by Tim Wilmshurst (Newnes) is an excellent text as well.

Bibliography

Throughout our time together I've mentioned different books that might be worth looking into if you're interested in exploring a certain idea further or understanding something in greater detail. Not all of the books mentioned throughout the chapters are listed here, but the most general and useful are.

Interaction Design

Data Flow: Visualising Information in Graphic Design by R. Klanten et al. (Die Gestalten Verlag)

Information Design Workbook by Kim Baer and Jill Vacarra (Rockport)

Envisioning Information by Edward R. Tufte (Graphics Press)

Visual Explanations: Images and Quantities, Evidence and Narrative by Edward R. Tufte (Graphics Press)

The Design of Everyday Things by Donald A. Norman (Basic Books)

Universal Principles of Design by William Lidwell (Rockport)

The Laws of Simplicity by John Maeda (MIT Press)

About Face 3: The Essentials of Interaction Design by Alan Cooper et al. (Wiley)

Designing for Interaction: Creating Smart Applications and Clever Devices by Dan Saffer (New Riders)

Sketching User Experiences: Getting the Design Right and the Right Design (Interactive Technologies) by Bill Buxton (Morgan Kaufmann)

Acting with Technology: Activity Theory and Interaction Design by Victor Kaptelinin and Bonnie A. Nardi (MIT Press)

Human-Machine Reconfigurations: Plans and Situated Actions (Learning in Doing: Social, Cognitive and Computational Perspectives) by Lucy Suchman (Cambridge University Press)

Where the Action Is: The Foundations of Embodied Interaction (Bradford Books) by Paul Dourish (MIT Press)

Digital Ground: Architecture, Pervasive Computing, and Environmental Knowing by Malcolm McCullough (MIT Press)

Reassembling the Social by Bruno Latour (Oxford University Press)

Sweet Anticipation: Music and the Psychology of Expectation by David Huron (MIT Press)

This Is Your Brain on Music: The Science of a Human Obsession by Daniel J. Levitin (Plume/Penguin)

Gaming: Essays On Algorithmic Culture by Alexander R. Galloway (University of Minnesota Press)

Persuasive Games: The Expressive Power of Videogames by Ian Bogost (MIT Press)

First Person: New Media as Story, Performance, and Game by Noah Wardrip-Fruin and Pat Harrigan (MIT Press)

Second Person: Role-Playing and Story in Games and Playable Media by Pat Harrigan and Noah Wardrip-Fruin (MIT Press)

Programming

Processing: A Programming Handbook for Visual Designers and Artists by Casey Reas et al. (MIT Press)

Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction (Morgan Kaufmann Series in Computer Graphics) by Daniel Shiffman (Morgan Kaufmann)

Processing: Creative Coding and Computational Art (Foundation) by Ira Greenberg (Springer)

Making Things Talk: Practical Methods for Connecting Physical Objects by Tom Igoe (Make Books)

Algorithms for Visual Design Using the Processing Language by Kostas Terzidis (Wiley)

Visualizing Data: Exploring and Explaining Data with the Processing Environment by Ben Fry (O'Reilly)

Analog In, Digital Out: Brendan Dawes on Interaction Design by Brendan Dawes (New Riders)

The Computational Beauty of Nature by Gary William Flake (MIT Press)

Mathematics and Physics for Programmers by Danny Kodicek (Charles River Media)

OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R) by OpenGL Architecture Review Board et al. (Addison-Wesley)

OpenGL(R) Shading Language by Randi J. Rost (Addison-Wesley)

OpenGL(R) SuperBible: Comprehensive Tutorial and Reference by Richard S. Wright et al. (Addison-Wesley)

C++ Primer Plus, Fifth Edition by Stephen Prata (Sams)

Learning OpenCV: Computer Vision with the OpenCV Library by Gary Bradski and Adrian Kaehler (O'Reilly)

Hardware

Getting Started with Arduino (Make: Projects) by Massimo Banzi (Make Books)

Arduino Cookbook by Michael Margolis (Available 1/2010; O'Reilly)

Physical Computing: Sensing and Controlling the Physical World with Computers by Tom Igoe and Dan O'Sullivan (Course Technology)

Getting Started in Electronics by Forrest M. Mims III (Master Publishing)

Electronic Sensor Circuits & Projects by Forrest M. Mims III (Master Publishing)

Handmade Electronic Music: The Art of Hardware Hacking by Nicolas Collins (Routledge)

Electronic Projects for Musicians by Craig Anderton (Music Sales Corp.)

Designing Embedded Hardware by John Catsoulis (O'Reilly)

Programming Embedded Systems in C and C++ by Michael Barr (O'Reilly)

Art

Design and the Elastic Mind by Hugh Aldersey-Williams et al. (The Museum of Modern Art)

Relational Aesthetics by Nicolas Bourriaud (Les Presse Du Reel)

Musimathics, Volume 1 and Volume 2: The Mathematical Foundations of Music by Gareth Loy (MIT Press)

Aesthetic Computing by Paul Fishwick (MIT Press)

Information Arts: Intersections of Art, Science, and Technology by Stephen Wilson (MIT Press)

At the Edge of Art by Joline Blais et al. (Thames and Hudson)

Noise, Water, Meat: A History of Sound in the Arts by Douglas Kahn (MIT Press)

4dsocial: Interactive Design Environments (Architectural Design) by Lucy Bullivant (Wiley)

4dspace: Interactive Architecture (Architectural Design) by Lucy Bullivant (Academy Press)

Database Aesthetics: Art in the Age of Information Overflow by Victoria Vesna (University of Minnesota Press)

A Thousand Years of Nonlinear History by Manuel De Landa (Zone Books)

Materializing New Media: Embodiment in Information Aesthetics by Anna Munster (Dartmouth)

Digital By Design by Troika (Thames and Hudson)

Atlas of Novel Tectonics by Jesse Reiser (Princeton Architectural Press)

Interaction of Color by Josef Albers and Nicholas Fox Weber (Yale University Press)

The Art of Color by Johannes Itten (Wiley)

Conclusion

Throughout this book you've seen techniques for making interactive applications, ways of thinking of both code and hardware, libraries to use, and interviews with artists and designers who work with interaction as their primary medium. What's more, all the code that you've seen has been using open source projects that believe firmly in sharing code, resources, and ideas with other artists, designers, and programmers like yourself. This will make it easier for you to find help when you need it, get ideas when you want them, and get feedback on your work when you want it.

In addition to the books and websites mentioned in this book, there are a growing number of conferences and exhibitions throughout the United States, Europe, and Asia highlighting interaction design and artwork by design groups, students, and artists. Some of these are hosted at universities that offer courses and degrees in new media studies, and others are simply occasions for like-minded thinkers to gather, present their own work, and look at the work of others. In the interest of not dating this book too greatly in the near future, I'll direct you to sites like *http://we-make-money-not-art.com* or the websites of each of the three featured projects in this book for more information.

I hope that, whatever your interests or reasons for picking up this book in the first place, you put this book down feeling prepared to begin your own exploration of the artistic and design potential in using technology to create meaningful, playful, or thought-provoking interactions.

APPENDIX

Circuit Diagram Symbols

Figure A-1 shows some of the most commonly used symbols in electrical diagrams. Understanding the symbols will help you interpret the diagrams in this book and the many schematic diagrams that you might find on the web or in other books. It's also useful if you want to make your own diagrams for projects. You can find a more comprehensive list at *www.kpsec.freeuk.com/symbol.htm*.

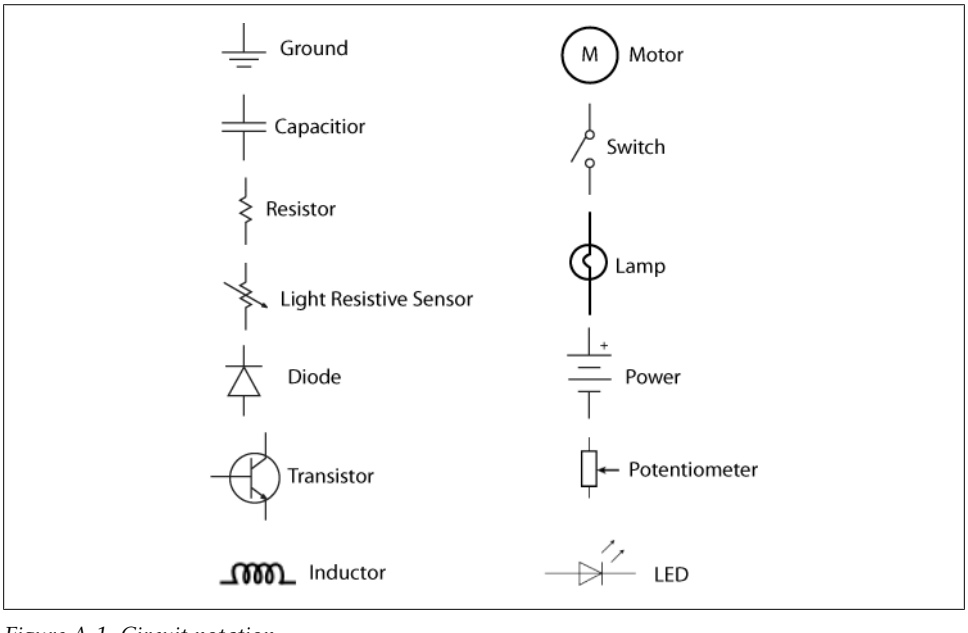


Figure A-1. Circuit notation

Programming Glossary

This programming glossary defines some of the most commonly used programming terms throughout this book. It doesn't include operators and symbols.

#define
: The `#define` operator defines a value that persists throughout an entire application. For instance:

```
#define NUMBER 4
```

Now the constant `NUMBER` will represent the value 4 throughout the application.

#ifdef
: The `#ifdef` operator checks whether something has been defined using the `#define` keyword. It must be followed by `#endif`.

#ifndef
: The `#ifndef` operator checks whether something has *not* been defined using the `#define` keyword. It must be followed by `#endif`.

amp
: This is short for *ampere* and is a electrical measurement of how much electrical current is flowing in a circuit. It is equal to the voltage divided by the resistance.

analog pin
: On the Arduino, the analog pins enable reading of voltage. The voltage on the pin is interpreted as an integer value in the range of 0 to 1,023.

array
: An array is a collection of variables that are accessed with an index number. An array is declared with a length or with initializers that determine the length. The following code declares an array of six elements with the first element at `arr[0]` and the last at `arr[5]`:

```
int arr[6];
```

Processing arrays are created like so:

```
int arr = new int[6];
```

In C++, declaring the array initializes each element in the array when the object is created *only if* the array is declared in the class:

```
class myClass {int arr[6];
// will be created when an instance of
// myClass is created
}
```

The elements would be accessed as follows:

```
int first = arr[0];
int last = arr[5];
```

ASCII
: American Standard Code for Information Interchange (ASCII) is a set of definitions for 128 characters: 33 are nonprinting control characters that affect how text is processed, 94 are printable characters, and the space is considered an invisible graphic. Each character is represented as a number; for instance, *A* is represented as 65, and *a* is represented as 97.

assignment

Assignment is when a value is given to a variable, like so:

```
int j;
j = 5; // now j is assigned a value
```

bitwise operations

Bitwise operations work at the bit level of variables. These are the AND (`&`) operator (not to be confused with the reference operator, which looks the same but is used differently), the OR (`|`) operator, the XOR (`^`) operator, and the two bitwise shift operations (`<<` and `>>`).

Boolean

`boolean` variables or `bool` hold one of two values: `true` and `false` represented as 1 and 0, respectively. They are the same in C++, Arduino, and Processing.

byte

In both Processing and Arduino, the byte stores an 8-bit number, from 0 to 255. byte is an unsigned datatype, so it does not store negative numbers.

capacitor

Capacitors are another element used to control the flow of charge in a circuit. The name derives from their capacity to store a charge. Capacitors consist of two conducting surfaces separated by an insulator; a wire lead is connected to each surface.

cast

A cast translates one variable type into another and forces calculations to be performed in the cast type. It uses the `()` operator and looks like this: `(type)variable`. For instance:

```
int i;
float fl;
fl = 5.5;
i = (int) f; // i is the integer value of f,
             // so it will be equal to 5
```

Casts can also be done like so:

```
i = int(f);
```

char

A char takes up 1 byte of memory and stores a character value written in single quotes, like `'A'`. The char stores the characters in ASCII as numbers, which means a char can be used for arithmetic. The char datatype is a signed type, meaning that it encodes numbers from –128 to 127. The unsigned char can store numbers from 0 to 255 (and is exactly the same as the byte type).

class

A class is a prototype for an object in Processing or C++. It can be used to create an instance of that class that will have its constructor called when it is created. In C++, for instance, class declarations look like this:

```
class Circle {
    int radius;
    int x;
    int y;
};
```

In Processing, class definitions look the same but do not require the semicolon (`;`) at the end of the class.

comment

Comments help you understand (or remember) how your program works or inform others how your program works. There are two different ways of marking a line as a comment:

```
// this is a single-line comment
/* this is a multiline comment
that ends with a */
```

comparison

Comparison operators are operators that compare two values, like `!=` (not equal to) or `==` (equal to), for instance. `3 == 4` returns `false` (because 3 is not equal to 4), while `7.8 != 9` returns `true` (because the values are not equal).

constant

A constant is a value that does not change within a program. It cannot be reassociated with a different value like a variable can. You declare a constant like this:

```
const int constInt = 5; // Arduino and C++
public static final int constInt  = 5;
// Processing
```

constructor

The constructor of a class is the method that will be called when an instance of that class is created. In both C++ and Processing, the constructor has the same name as the class:

```
class Circle {
    Circle() {
       // do something on creation
    }
}
```

cpp file

In an oF or other kind of C++ class, the *.cpp* file is where all the definitions for a class are stored.

dereferencing

Dereferencing a pointer returns the value of the variable that the pointer points to. For instance:

```
int* p;
int j = 5;
p = &I;
int k = *p; // k is now 5
```

digital

In the Arduino controller, the digital pins accept or send digital signals. The signal values are either `HIGH` or `LOW`.

diode

The diode acts like a one-way valve for current, and this is a very useful characteristic. One application is to convert alternating current (AC), which changes polarity periodically, into direct current (DC), which always has the same polarity.

double

A double is a very large floating-point number consisting of 8 bytes (64 bits) that can represent numbers as large as 18,446,744,073,709,551,615. In Arduino, a double is the same size as a float (4 bytes).

event handler

An event handler is called when a certain event happens. In Processing applications, the `mouseMoved()` method is an example of an event handler. In an oF application, the `receivedSound()` method is an event handler that indicates that the system has received data from the sound card.

float

A float is a number that has a decimal point. Floating-point numbers can be as large as 3.4028235E+38 and as low as –3.4028235E+38. They are stored as 32 bits (4 bytes) of information.

for

The `for` loop repeats a block of statements enclosed in curly braces until the condition is met. An increment counter is usually used to increment the loop counter and terminate the loop, though there are other ways of working with `for` loops. There are three parts to the `for` loop header:

```
for (initialization; condition; increment) {}
```

For example:

```
for (int i = 0; i < 10; i++) {
    doSomething(i);
    //call doSomething with values
    //from 0 to 9
}
```

function

A function is a subroutine with a name that optionally takes parameters and returns a value. For instance:

```
int square(int val) {
    return val*val;
}
```

Functions that are defined within a class are called *methods*.

function call

A function call is another way to refer to calling a method that has been declared and defined elsewhere. For instance:

```
methodName();
```

header file or .h file

In an openFrameworks class or Arduino library, the *.h* or header file is where the declarations of a method and a class are stored.

if/else

An `if` is used with a comparison operator and tests whether a certain condition is true or false:

```
if (x > 50) {
    // do something here
} else {
    // do something if x isn't > than 50
}
```

inheritance

Inheritance is the process of having one class extend another class. When a class extends another class, it inherits all the methods and properties of its parent class. In C++, this is done by writing the following:

```
class Circle : public Shape {
};
```

In Processing, it's done by writing the following:

```
class Circle extends Shape {
}
```

integer

An integer is a numerical value that does not have a decimal point. It represents either of the following:

- A 2-byte (16-bit) number, with a value between 0 and 65535 for an unsigned Integer and –32,768 and 32,677 for a signed integer. This is the size of an int in Arduino.
- A 4-byte (32-bit) number, with a value between 0 and 4,294,967,295 for an unsigned Integer and –2,147,483,647 and 2,147,483,647 for a signed integer. This is the size of an int in Processing and oF.

In other platforms not covered in this book, an int can be an 8-byte (64-bit) number.

long

Long variables are extended-size variables for number storage, and on Arduino and oF they can store 32 bits (4 bytes) from –2,147,483,648 to 2,147,483,647 or for an unsigned long from 0 to 4,294,967,295. In Processing, a long stores values up to 18,446,744,073,709,551,615.

method

A method is a function that is declared within a class.

method declaration

This is the declaration of a function within a class, where its signature is defined. In C++, this is often done in the *.h* header file:

```
int charToInt(char ch);
```

In Arduino this is also sometimes done in an *.h* file, though it doesn't always need to be. In Processing, method declarations and definitions are done at the same time.

method definition

In C++, this is done in the *.cpp* file and looks like this:

```
void className::methodName() {
}
```

Methods that are included in the *.cpp* file of a class have to defined in the classes definition in the *.h* file. In Processing and Arduino, the declaration and definition are done at the same time, though sometimes *.h* files with declarations are used as well.

object-oriented programming

Object-oriented programming, or OOP, is the process of using multiple classes to represent different areas of functionality or data objects within your application.

Ohm

An Ohm is an electrical measurement that indicates the amount of resistance that current will encounter as it travels through a circuit. It is equal to the voltage divided by the current.

opamp

An opamp, or operational amplifier, is a DC device that amplifies signals.

operator

An operator is a function that operates on or modifies a value or function. `+`, `=`, and `/` are mathematical operators, while `&` and `?` in C and C++ are nonmathematical operators.

pin
: On the Arduino, the pins are the ports that connect into the microprocessor. There are analog, digital, and PWM-enabled pins.

pointer
: A pointer is a type in C++ and C that points to a section in memory. They are most often used to pass to a method to ensure that the particular variable being pointed to is modified by the method, rather than any other variable.

potentiometer
: A potentiometer is a resistor that is usually controlled directly by the user, allowing the amount of voltage that it resists to be set by a dial or other physical control. This is commonly used in knobs and dials.

PWM
: Pulse Width Modulation (PWM) is a way of simulating an analog output by varying `HIGH` and `LOW` signals at intervals proportional to the value.

recursion
: Recursion is a process by which a method calls itself over again until some process is complete or some condition is met.

reference
: A reference is the location in memory that a variable points to. You can use references to set the value of a pointer. For instance:

```
int* p;
int j = 5;
p = &j; // p is now pointing at the location
        // in memory where j is stored
```

resistor
: Resistors are electrical components that resist the flow of charge. The value of a resistor is measured in Ohms and represented by the Greek letter capital omega.

return
: This statement sets what a function returns. In C++ and Java, the return type is indicated in the signature of the method like this:

```
int addNumbers()
```

scope
: Scope defines the area of an application or method in which a variable is accessible. Most variables are accessible only within the brackets within which they are declared.

Serial
: Serial is a library in the Arduino core software that enables serial communication using the RS232 protocol over a port opened between another serial device. You'll most often hear the serial port discussed in setting up communication between the Arduino controller and another host computer.

short
: A short is a datatype that represent a small int, using 2 bytes instead of 4 like a C++ or Java int. Shorts are useful for calculations that need to be extremely fast where you know that your data value will never exceed the range of two bytes. Short is signed by default, but can also be declared as unsigned.

signed
: Signed numerical values can have negative numbers assigned to them. This usually means that the first bit of the number states whether the number is negative or positive. This means that signed variables represent ranges that start at the lowest possible number, for instance –32,768 for a signed int (in Arduino), and go to the highest number, which for a signed int is 32,677.

static
: Marking a method as `static` means that it is available from a class whether or not an instance of that class has been created. For instance, in C++ you can call a static method like this:

```
ClassName::methodName();
```

In Processing or Java, static methods are called like so:

```
ClassName.methodName();
```

Declaring a variable as `static` means that the variable name refers to the same value

throughout the application whether an instance has been created or not.

string

A string is construct available in Processing and C++, which represents a series of characters that can be split, looped through, or accessed using methods that make working with a string easier than working with an array of characters.

struct

A `struct` is a collection of variables, somewhat like a simplified class. It cannot have methods, but it can have properties. For instance:

```
struct {
    int radius;
    int x;
    int y;
} Circle;
```

The struct can be used only in C++ and Arduino.

switch

`switch/case` statements control the flow of programs by checking a list of "cases" inside a set of bracket. The program checks each `case` for a match with the test variable and runs the code if a match is found.

```
switch (var) {
    case 1:
        //do something when var == 1
    break;
        // break is optional
    case 2:
        //do something when var == 2
    break;
    default:
        // if nothing else matches,
        // do the default
        // default is optional
}
```

type

Type refers to what datatype a variable represents. For instance, int, string, or char are all datatypes. All variables and methods must have a type even if that type is `void`, unless they are the constructor method of a class.

unsigned

Declaring a variable as unsigned means that it will store values only between 0 and their maximum value. For instance, in Arduino, an unsigned int stores 0 to 65,535. The first bit of an unsigned datatype is not used to determine whether the number is positive or negative.

variable assignment

Variable assignment is the assignment of a value to a variable. For instance:

```
int val;
val = 5; // assign to val the value 5
```

Before a value is assigned, the variable is `null`, which means it has no value and may cause errors if you try to read from it.

variable declaration

Variable declaration is where a variable is first given a type and a variable name. For instance:

```
int j;
char name[20];
```

In C++, the methods and variables that the class will contain are declared in the *.h* file.

vector

A vector is an object in C++ that behaves somewhat like an array in that it contains multiple elements that can be accessed with the [ ] operators, but unlike arrays in C++ it can be of a variable size.

voltage

This is the rate at which energy is drawn from a source that produces a flow of electricity in a circuit. It is measured and expressed in volts. The higher the voltage, the more current is flowing around a circuit. It is equal to the current times the resistance.

while

The `while` loop constructs a loop that executes until the statement in the condition of the loop is true. For example, the following will execute 10 times, as long as `I` is less than 10:

```
int I = 0;
While(i<10) { i++;}
```

Index

Symbols

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

About the Author

Joshua Noble is a consultant, freelance developer, and Rich Internet Application designer based in Brooklyn, New York. He is the lead author of *Flex 3 Cookbook* (O'Reilly).

As a graduate student, Joshua Noble studied interactive art, teaching himself programming and electronics using available resources on the Internet. After school, he began teaching coding to art and design students interested in interactive design at the School of the Museum of Fine Arts in Boston.

He found an acute need for a book that taught the technical aspects of programming and computing for interactive art and design as well as some of the theoretical and conceptual aspects of design interaction. He's worked extensively with each of the tools discussed in this book and has taught the subject at workshops, colleges, and to friends.

Colophon

The animals on the cover of *Programming Interactivity* are guinea fowl (family *Phasianidae*, subfamily *Numindinae*). Sometimes known as guinea hen, wild guinea fowl originally hail from western Africa. Featherless heads with black crests and dark gray or deep blue plumage distinguish guinea fowl from other birds.

Domesticated guinea fowl (descended from *Numida meleagris*) make popular additions to farms, as farmers value the birds for their ability to control insects (guinea fowl dine on insects, leafy greens, and seeds). Farmers and other guinea fowl owners also appreciate the birds' paranoid natures; guinea fowl will cry out at provocations as slight as the bark of a dog, the beep of a horn, or a stranger's footsteps.

Their distinctive cries provide an easy way to distinguish the gender of the birds. While females and males both make a piercing "ah, ah, ah" sound when provoked, only the female can produce a two-syllable call that sounds as if she is saying "come back, come back, come back" or "buckwheat, buckwheat, buckwheat."

Gourmands prize cooked guinea fowl for their lean, tender flesh, which possesses a less gamy flavor than pheasant, while others say the prepared bird tastes like chicken (and also a little bit like turkey).

The cover image is from *The Riverside Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.