CAD Tools for Creating Space-filling 3D Escher Tiles

Mark Howison

CAD Tools for Creating Space-filling 3D Escher Tiles

by Mark Howison

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Carlo H. Séquin
Research Advisor

(Date)

* * * * * * *

Professor Jonathan R. Shewchuk
Second Reader

(Date)

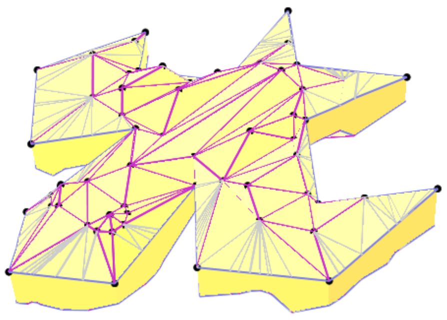
CAD Tools for Creating Space-filling 3D Escher Tiles

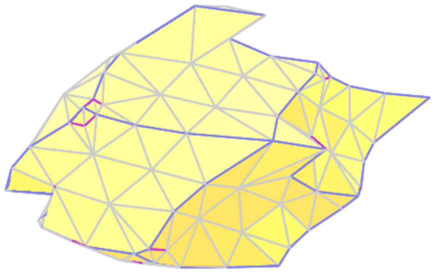
Mark Howison
Computer Science Division
University of California, Berkeley
mhowison@berkeley.edu

May 1, 2009

Abstract

We discuss the design and implementation of CAD tools for creating decorative solids that tile 3-space in a regular, isohedral manner. Isohedral tilings of the plane, as popularized by M. C. Escher, can be constructed by hand or using existing tools on the web. Specialized CAD tools have also been developed for tiling other 2-manifolds. This work addresses the question: *How can we generate interesting tilings of 3-space?* To generate boundary representations of 3D tiles, we have implemented an interactive constrained Delaunay triangulation algorithm. In addition, we have designed a specialized mesh-cutting algorithm used in layering extruded 2D tiles to create intricate space-filling designs. We describe visual debugging methods used during the implementation of these two geometric algorithms, and also explain user-interface decisions we made in designing the CAD tools. Finally, we show examples of 3D tilings that are derived from extruded 2D shapes and from 3D cubic lattices.



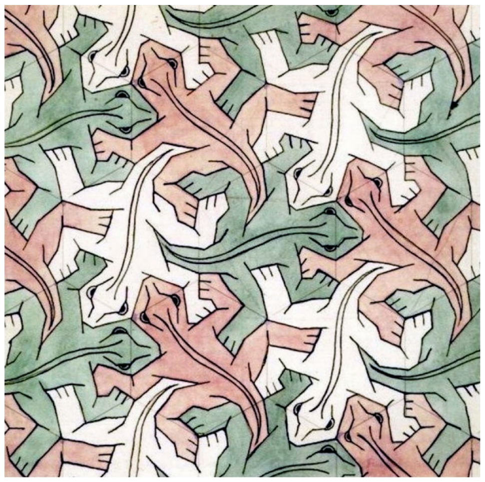


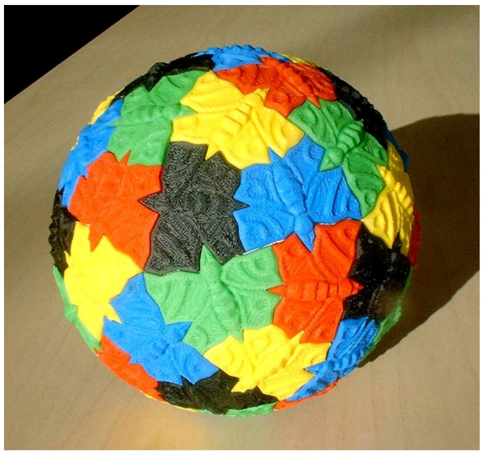
1 Introduction

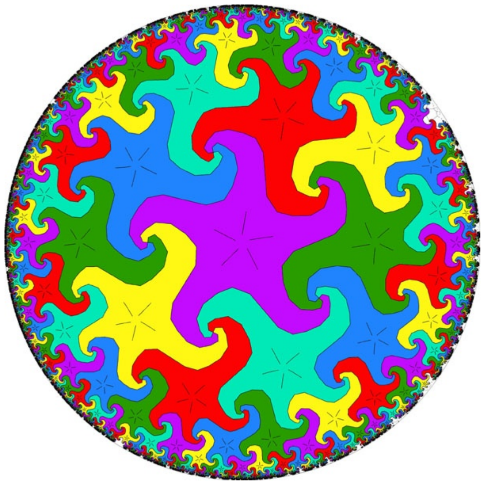
M. C. Escher's intricate tilings [12] are well known and appreciated by many people; the intriguing, natural looking shapes that tile the plane in a regular manner have fascinated mathematicians, artists, and tiling hobbyists (Fig. 1). Without the help of computer graphics tools, it is difficult and labor-intensive to create aesthetically pleasing tilings of this kind. Because of the widespread interest in such patterns, many easy-to-use graphics tools have been created and made available on the web, allowing people with no training in the graphics arts or in computer science to experiment with and generate innovative regular patterns [11].

Such tilings can also be created on surfaces other than the plane. Figure 1b shows a spherical tiling made from 60 identical tiles made on a rapid prototyping machine [25], and a hyperbolic tiling in the Poincaré disks, where the tiling becomes infinitely dense towards the rim of the circular domain. In fact, all planar tilings can be generalized to hyperbolic patterns by simply packing more instances of the tile around its sides. Spherical tilings, on the other hand, are limited to the symmetries of the Platonic solids, since they have the added constraint of closing smoothly around the back of the sphere. There are several tiling generators on the web for hyperbolic tilings [7], and also for spherical tilings [24]. In some isolated experiments, Escher-like tiling patterns have also been placed on symmetric surfaces of higher genus, e.g., onto a torus [21] and onto a genus-3 surface with tetrahedral symmetry (Fig. 1d) [14]. In both cases, specially designed CAD tools were created to address the particular challenges of those tasks.

Prompted by the emergence of affordable layered manufacturing machines and rapid-prototyping services, we began to explore the possibility of making Escher-like tilings that would regularly and seamlessly fill 3-space. This exploration space is much larger than in the case of planar and 2D tilings. First, there are many more symmetry groups in 3-space than in the plane. Second, the 3D tiles can be of a


(a)


(b)


(c)

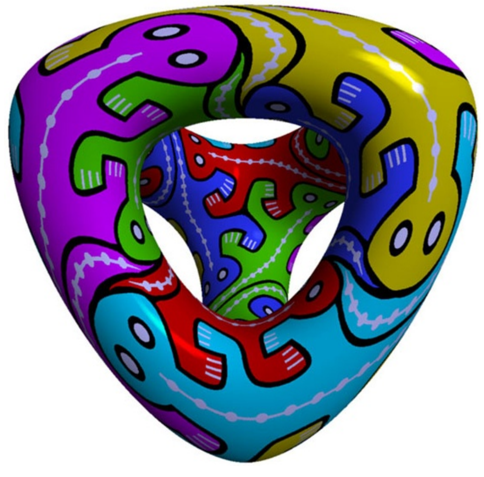

(d)

Figure 1: Escher-like tilings on 2-manifolds: (a) in the plane; (b) on a sphere; (c) in the Poincaré disk; and (d) on a genus-3 “Tetrus” surface.

genus higher than zero, they can interlink with their neighbors, and they can even be knotted! An exploratory paper [15] surveys many of these possibilities, and concludes that different approaches and tools would be needed to design such tiles.

We present CAD tools that aid in the construction of isohedral tiles of genus zero, with complex surfaces that may or may not resemble shapes found in nature. In 3-space, new challenges arise for the development of appropriate CAD tools. The data structures and geometrical algorithms are more complex; but also there are user interface issues arising from the limitations of projecting a 3D object onto a 2D viewing screen and the geometric interdependences caused by the imposed symmetries. With 2D tilings, a single comprehensive view can show the prototype tile and its nearest neighbors, but this is not true for 3D tilings. If we display only one isolated tile, then we can see at most half of its surface, and if we display more than one tile, the neighboring tiles can occlude features of the prototype tile. Furthermore, it is important to view all faces that are modified as the result of an editing operation, yet because of the tile's symmetries, these faces are typically opposite each other on the tile's surface. In the following, we address these issues and present CAD solutions.

2 Simple 2½-Dimensional Tilings

As a warm-up exercise, we started by constructing an editing tool for a 2½D tile. A tile that tessellates 2-space is extruded into a slab, and layers of these tiles are stacked to fill 3-space. The 2D outline of the tile can be designed with one of the many available 2D tools, but additional facilities are needed for shaping the top and bottom surfaces of this tile. This intermediate 2½D design tool allowed us to explore suitable data structures and geometrical algorithms, and to debug them in a less complicated context than the full 3D case.

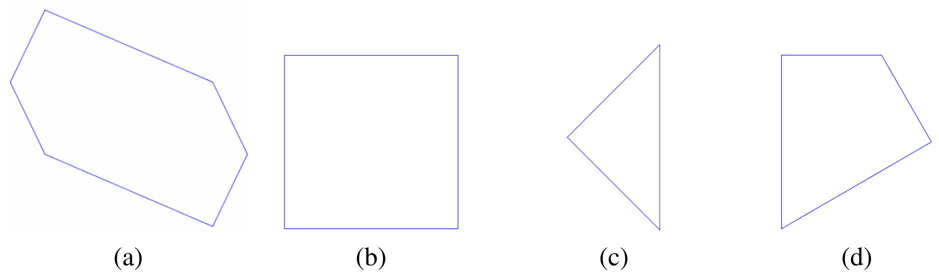


Figure 2: Four 2D symmetry groups: (a) IH01, hexagonal domain with translational symmetry; (b) IH41, rectangular domain with translational symmetry; (c) IH79, right-triangle domain with 4-fold rotational symmetry; and (d) IH31, kite-shape domain with 6-fold rotational symmetry.

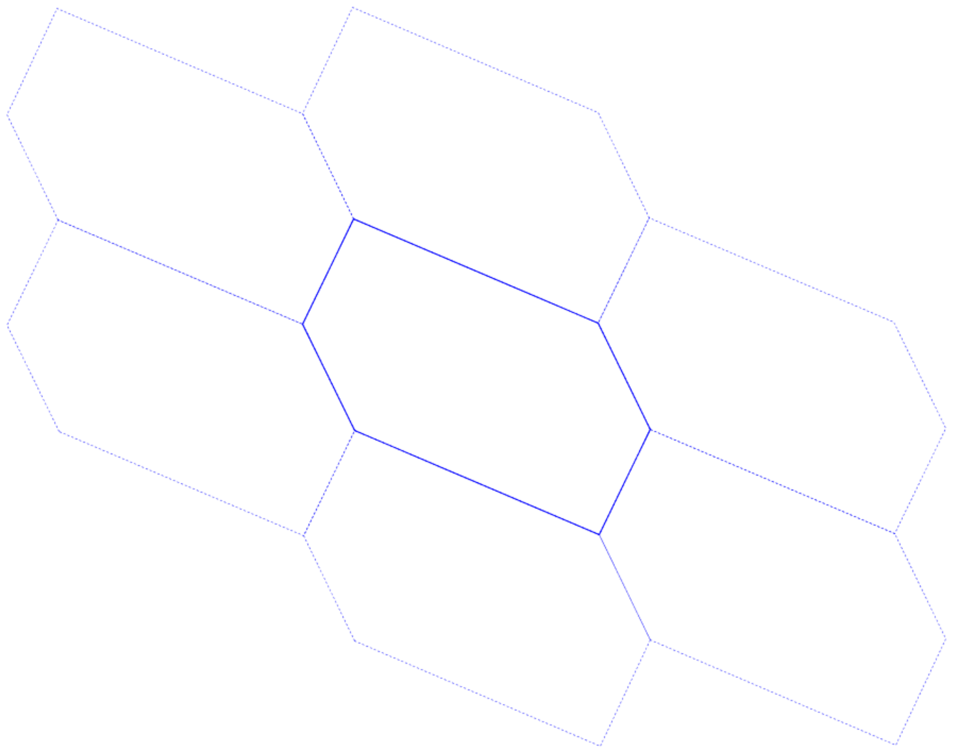


Figure 3: IH01 shown with nearest neighbors.

In our 2½D editor, we have implemented four symmetry groups (Fig. 2). The first is a simple isohedral tiling with only translational symmetry (IH01; type p1 [5]; Conway notation: o [1]). The simplest repeatable unit of this tiling (its *fundamental domain*) is a skewed hexagon in which opposite sides are identical, translated copies of one another. A similar group, IH41, uses a rectangular instead of hexagonal domain with the same translational symmetries. A third example uses higher-order symmetries (IH79; type p4; Conway notation: 442). Its fundamental domain is an isosceles right triangle in which the two legs transform into one another by a 90° rotation around the shared vertex, and half the hypotenuse maps into the other half by a 180° rotation around its midpoint. Finally, IH31 (type p6; Conway notation: 632) is similar to IH79, but has 6-fold symmetry around its shared vertex and a kite-shaped fundamental domain.

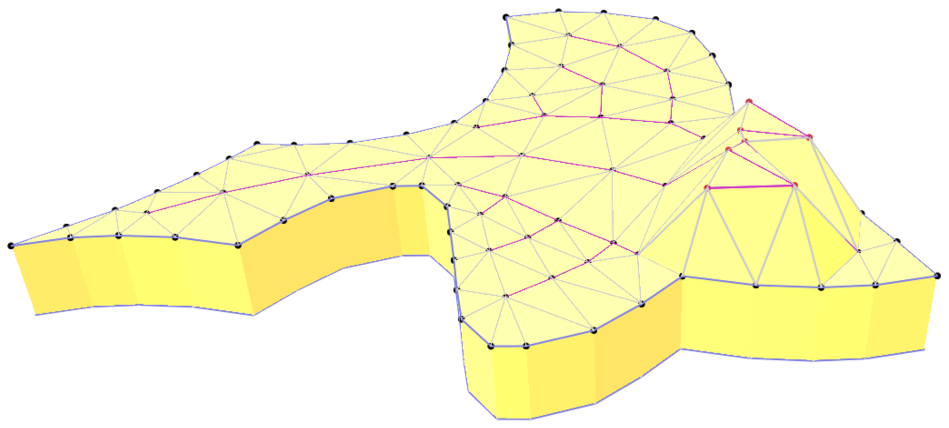


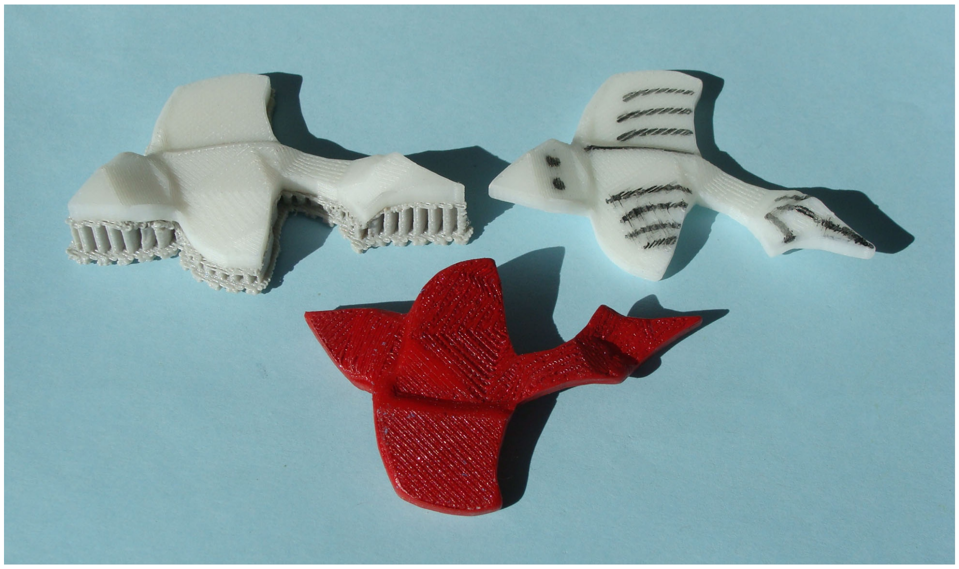
Figure 4: 2½D Escher tiles: height-editing of the top surface.

The construction of a 2½D Escher tile occurs in two distinct phases. From a user's perspective, the first phase resembles that of existing 2D Escher tile editors. The user picks a symmetry group and is given a basic shape that represents the fundamental domain of this group. This tile can be modified in the context of all its neighbors and the whole tiling array. Any change made to a segment of the edge of a tile is readily replicated on all corresponding edge segments on all other displayed tile instances. The user can also decorate the interior of the tile with extra points and line segments.

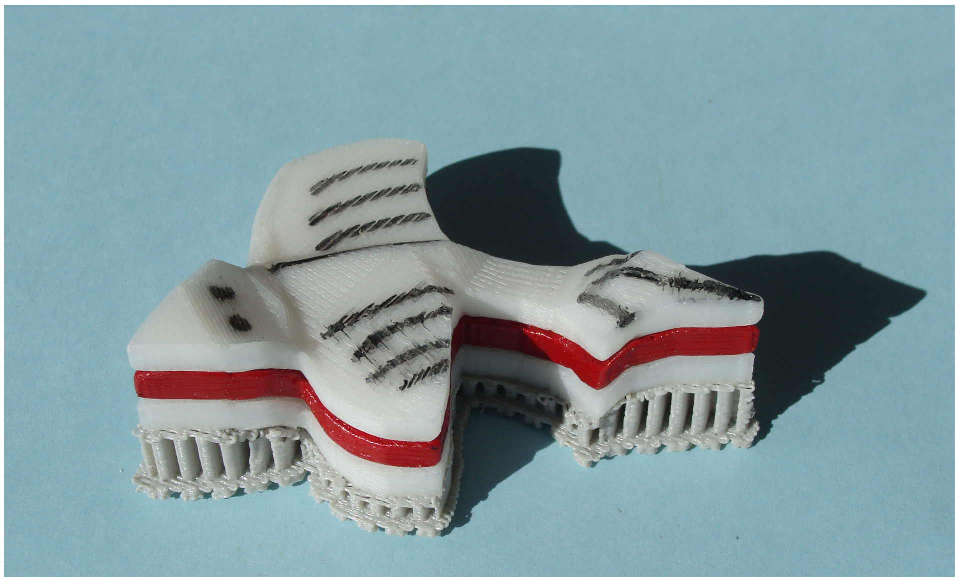
In the second editing phase, the whole tile is extruded uniformly to a chosen thickness, and then the top surfaces can be further modeled into a non-planar height field by moving vertices—both on the boundary and in the interior of the tile—up or down in the vertical direction. During this edit phase, a single tile is displayed as a 3D object that can be arbitrarily rotated around its center of gravity, allowing the designer to choose a convenient viewing direction that best shows the 3D deformations being performed. To allow some structured height-editing, multiple vertices can be selected (highlighted in red in Fig. 4), and all their heights can be changed simultaneously by the same amount by simply dragging one of those vertices up or down in a direction perpendicular to the base plane of the tile.

As a default case, the bottom surface of the tile is modified in exactly the same way as the top surface, so that the 3D tiles stack on top of one another in a single prismatic column that also fits together laterally with neighboring columns to fill 3-space (Fig. 5).

In a more interesting 3D isohedral tiling, subsequent layers of these tiles can be shifted with respect to one another, so that, for instance, the belly of a bird-like tile rests on top of the wings of the bird in the layer below. In our CAD tool, we have restricted ourselves to isohedral tilings; thus, the lateral offset from one layer to the next one above must always be the same. This lateral offset is conveniently specified at the end of Phase I of the editing process, by grabbing the single edited proto-tile and shifting it laterally with respect to the complete 2D tiling (Fig. 6). The bottom surface of each tile is given by a combination of different parts of the top surfaces of the tiles in the layer below. In essence, the outline of the prototype tile is used as a "cookie cutter" to carve out a suitable mesh from the height field formed by all the top surfaces of all the tiles in the layer below. This carved-out mesh is then connected with vertical, prismatic side walls to the top surface of the proto-tile to form a closed, watertight, 2-manifold boundary representation of the 2½D Escher tile, which is suitable as input to any layered manufacturing machine.

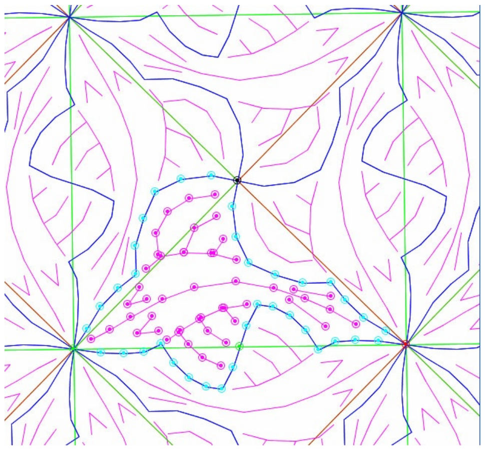


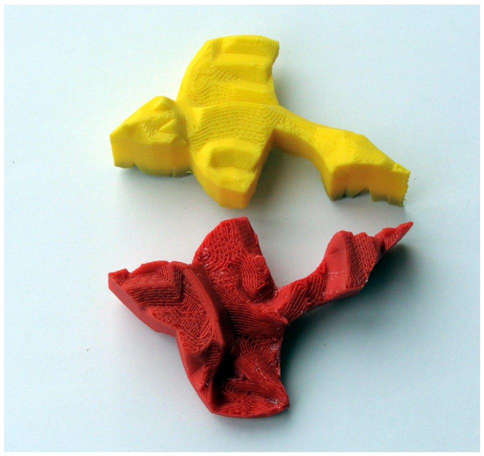
(a)

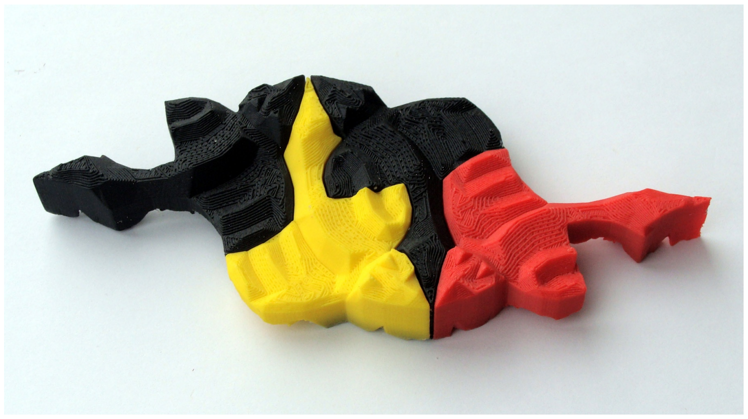


(b)

Figure 5: 2½D Escher tiles: (a) 3 identical tiles, white tiles seen from top, red seen from bottom; and (b) the 3 tiles stacked on top of one another.


(a)


(b)


(c)

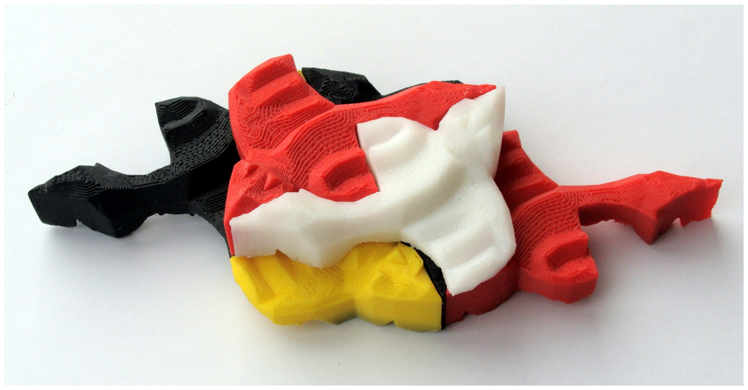

(d)

Figure 6: Offset 2½D Escher tiles: (a) proto-tile of symmetry IH79; (b) top and bottom view of the extruded 2½D tile; (c) four tiles put together in one layer; and (d) two more tiles placed in the laterally offset layer above.

3 Interactive Constrained Delaunay Triangulation

We create a mesh for the final boundary representation of the Escher tile by performing a constrained Delaunay triangulation of the interior of the proto-tile during Phase I. It is advantageous to show the designer the resulting triangulation after every edit step in Phase I, so that she can readily judge whether that triangulation is rich and robust enough to allow for the formation of the desired extruded features during Phase II. Also, having a mesh available during Phase I provides a convenient data structure in which to locate new points and check for illegal edit moves, e.g., boundary deformations that would lead to a self-intersecting boundary.

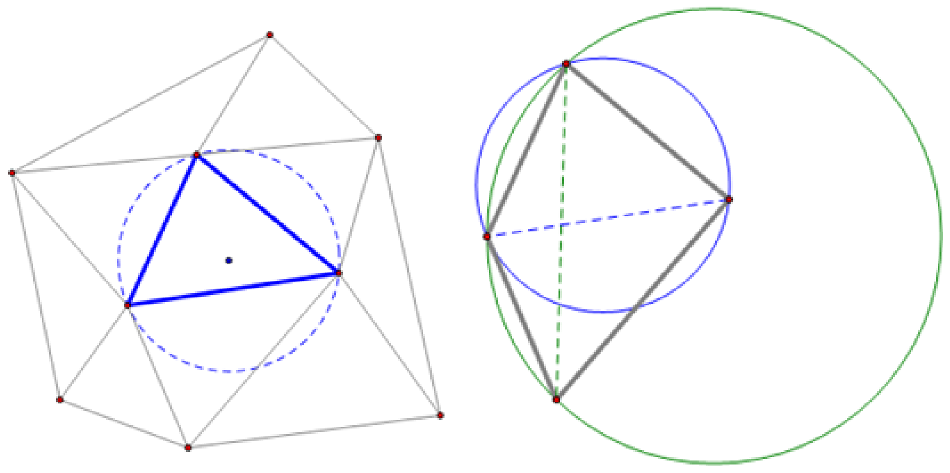


Figure 7: The Delaunay triangulation has the properties that (a) the circumcircle through any triangle encloses no other vertices of the triangulation; and (b) a non-Delaunay triangulation can be converted into a Delaunay one by flipping any edge (green) shared by two adjacent non-Delaunay triangles to an new edge (blue) shared by Delaunay triangles.

We perform Delaunay triangulation because it generates aesthetically pleasing triangulations with few sliver triangles (i.e., elongated triangles with one or two small angles). This is due to its "locally equiangular" [22] guarantee that, given any two adjacent triangles, their shared edge will be the one that maximizes the minimum of the six angles within both triangles. This is equivalent to the property of Delaunay triangulations that the circumcircle of any face (i.e., the unique circle that passes through all three of the face's vertices) does not enclose any other vertices of the triangulation [10], [22] (see Fig. 7). The Delaunay triangulation is also "globally equiangular" in that it maximizes the minimum angle over all faces of the triangulation, as compared to any other triangulation of the same vertices [22].

Writing a truly robust library for constrained Delaunay triangulation is a difficult task, and we contemplated using the well-tested Triangle package [17]. However, several considerations discouraged us from taking this approach. First, Triangle only operates in batch mode, delivering the Delaunay triangulation after all the constrained points and line segments have been placed. It seemed wasteful to run this process after every new placement or slight movement of a point in the proto-tile. Second, we wanted to develop our tile editor as a web-ready Java application, but a Java implementation of Triangle does not exist. Triangle's C codebase could be retooled to use the Java Native Interface, or a platform-specific Triangle executable could be called remotely on ASCII mesh data files from within Java, but both of these solutions would compromise the portability of our application.

Therefore, we decided to implement an interactive algorithm for creating and modifying a Delaunay triangulation as the designer edits a tile. Potentially, this could be a daunting task, considering how much effort was spent in the development of Triangle on issues of precision and numerical stability. Triangle uses adaptive precision arithmetic to achieve robustness without sacrificing much speed [17], but these methods are complicated and difficult to implement. We opted for a simplified version of adaptive arithmetic that was easier to implement while still providing similar performance benefits.

Although we represent coordinates as floating-point values, our editor is limited by screen resolution. Designers never input coordinates, and they judge the results visually to decide whether the resulting mesh fits their needs. We want to discourage the generation of many narrow sliver triangles, since overly complicated geometry cannot be realized accurately by the layered manufacturing machines used to fabricate the tiles. Thus, our triangulation algorithm automatically merges vertices that are placed too closely together, and subdivides and snaps line segments to new vertices that are placed too closely to them. If the result of a merge is not acceptable, the designer can always grab a vertex in question and move it to a slightly different location.

3.1 Robustness

Delaunay triangulation algorithms use two geometric predicates: 1) the *orientation* test, which determines if a vertex lies left, right or on top of a line; and 2) the *in-circle* test, which determines if a vertex lies inside, outside, or on top of the circumcircle of a triangle. Round-off errors from floating-point arithmetic can cause wrong results, such that an algorithm cannot correctly determine which side of a line a vertex lies on, or whether a point is inside or outside of a circumcircle.

As an alternative to using floating-point arithmetic, vertex locations can be restricted to rational numbers, allowing for integer arithmetic that is always exact when there is enough bit precision. There are several downsides to this approach, however. First, integer arithmetic requires error-handling code to prevent overflow conditions. Second, implementing rotations is awkward because many common angles have irrational values in their rotation matrices (e.g. $45°$ and $\sqrt{2}$), which have to be rounded. Finally, modern CPUs are optimized for floating-point operations, not integer arithmetic.

Exact (or arbitrary-precision) arithmetic can be used to circumvent floating-point round-off errors. Unfortunately, most implementations of exact arithmetic are notoriously slow. Alternatively, adaptive precision methods can provide the same robustness, but with less sacrifice in performance. Because both the *orientation* and *in-circle* tests rely on the sign of a determinant, a "floating-point filter" can be used to determine if the magnitude of the determinant as calculated by floating-point arithmetic is small enough to be affected by round-off error; if it is, exact arithmetic routines are used to increase the precision of the calculation until it can be guaranteed to have the correct sign [4]. While Shewchuk's [18] adaptive precision method uses several levels of filter, we have opted for a simpler one-level method that invokes an arbitrary precision routine whenever a floating-point calculation's uncertainty surpasses the filter threshold. Although this approach is not as efficient as the multi-level method, it is easier to implement while exhibiting similar performance benefits.

3.2 Delaunay Triangulation Algorithms

There are many existing algorithms for computing unconstrained Delaunay triangulations in batch mode [19]. Shamos and Hoey [16] developed an $\mathcal{O}(n \log n)$ *divide-and-conquer* algorithm for computing the Voronoi diagram, the dual of the Delaunay triangulation, which can then be converted into the Delaunay triangulation in linear time. Subsequent divide-and-conquer algorithms [6] bypass the Voronoi diagram, construct the Delaunay triangulation directly, and have simpler implementations.

An $\mathcal{O}(n \log n)$ *sweepline* algorithm by Fortune [3] sorts vertices along an axis, then uses a moving *front* perpendicular to the axis that accrete edges as it intersects vertices. Faces can be added as soon their circumcircles fall behind the front, and hence can't contain new vertices.

One of the easist algorithms to implement is Lawson's [10] *incremental insertion* algorithm that starts with a global bounding triangle and iteratively adds vertices by splitting the triangle that encloses the inserted vertex and performing incircle tests and edge flips to maintain a Delaunay triangulation. Proofs of the mathematical properties backing this technique can also be found in a succinct treatment by Sibson [22]. The running time of Lawson's algorithm depends on the search routine used to identify the triangle in the existing triangulation that encloses the vertex to be inserted.

We chose to implement Lawson's algorithm, because of its ease of implementation, and in particular because there is a straightforward modification to support constrained Delaunay triangulations. Constrained edges are simply flagged and then ignored during the *in-circle* test phase of the algorithm so that they cannot be flipped.

Rather than use a global bounding triangle for our incremental insertion algorithm, we chose to handle boundary edges and vertices explicitly since we represent that data to enforce the symmetry constraints of the tile. Boundary vertices contain a field that specifies the other boundary vertices they are paired with via symmetry contraints, and are stored in a doubly-linked list to facilitate operations that need to treat the boundary as a polygon separate from the triangulation (e.g. when testing to prevent situations where the boundary could become self-intersecting.) Although we could technically support mirror symmetries via scaling by a negative value, in practice these are not currently supported because of the additional complexity of inverting the direction of all half-edges in a mirror copy of a tile.

The insertion search is normally the bottleneck for an incremental insertion algorithm. Several optimizations exist that use tree or graph data structures to more efficiently locate insertion sites in the existing triangulation [19]. Our situation is different, however, since we are running in interactive mode rather than batch mode, and the time between user inputs is sufficient for even a slow search routine to finish. That said, we have implemented two refinements to Lawson's search routine, which follows triangles in the direction of the insertion site until it finds the containing triangle. First, we use a heuristic that stems from our specific application:

designers will likely add features as localized groups of vertices and constraints. Therefore, we use the last inserted vertex as the search origin for the subsequent insertion. Second, we temporarily load neighboring copies of the tile so that we do not have to walk around concavities in the boundary (Fig. 8). Because we are not using a convex global bounding triangle, this modification is neccessary, as concavities in the boundary can cause a naive triangle walking routine to hang, which is what initially led us to pursue the neighbor-copy approach.

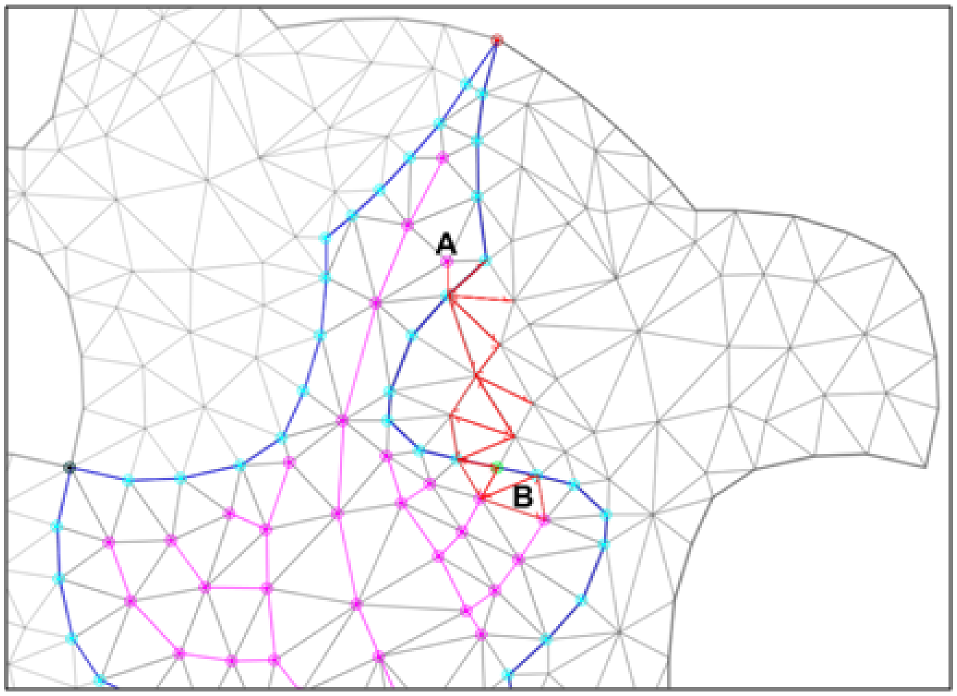


Figure 8: A location search for a point in triangle B starts at vertex A, the last inserted vertex. The search walks along the faces in as straight a line as possible. If it encounters the boundary of the tile, it continues into the adjacent tile, which is loaded temporarily for this purpose.

3.3 Constrained Edge Insertion and Polygon Filling

We support constrained edges to provide the designer with explicit control of the interior features of the tile. Our insertion routine is the same as Kao and Mount's [8]. Assuming that the endpoints of the constraint segment already exist in the triangulation, the edge is inserted by first removing all existing edges that lie between the endpoints. If one of these existing edges is itself a constrained edge, we instead add an additional vertex at the intersection, bisect the existing edge, then add two constrained edges that connect the endpoints and the new vertex. Clearing the edges between the endpoints and adding the new edge leaves two polygonal holes in the triangulation, which we fill with a polygon-filling algorithm. For ease of implementation, we use a worst-case $\mathcal{O}(n^2)$ algorithm published by Anglada [1], although in practice it typically runs in $\mathcal{O}(n \log n)$ time. The algorithm starts at the newly inserted constrained edge, which is "visible" to the other vertices of the polygon, meaning that those vertices can be connected to the visible edge's endpoints without leaving the polygon. Next, the algorithm finds the vertex with the largest circumcircle through the visible edge to form a triangle lying inside the polygon. This procedure is recursively applied to the polygons that remain on either side of the new triangle.

3.4 Interactive Triangulation and Non-convex Boundaries

Because of the user's interaction with the design environment, we implemented a constrained Delaunay algorithm that incrementally performs triangulation operations as the user specifies modifications. Originally, we did not update the triangulation as the user modified the tile boundary and interior, but instead required a separate "triangulation" stage. The triangulation algorithm, which added in edges from a length-sorted list of all possible edges, was slow, non-Delaunay, and had problems with numerical instability. The incremental algorithm described here is superior in its running time, its stability, and its ability to provide immediate feedback to the user.

We support both removal and relocation of vertices in the existing triangulation. Interior vertices are removed by flipping edges in the star of edges eminating from the point until there are no more than four edges left in the star. The remaining three or four edges are removed, leaving behind either a single face or a quadrilateral which is filled with a valid diagonal. The triangulation is restored to a Delaunay triangulation by performing in-circle tests on all edges affected by the vertex removal and flipping them when needed. Relocation of an interior vertex is implemented simply as a removal plus a re-insertion. However, special care has to be taken to preserve constrained edges. Whenever a constrained edge is removed, its endpoints are added to a queue so that it can be re-inserted after the vertex has been relocated.

Removing boundary vertices is more difficult. We connect the adjacent boundary vertices with a new boundary edge and perform a flood search to find and remove any interior edges that intersect the new boundary edge or lie outside the new boundary. This may leave a polygonal hole along the new boundary edge, but the new boundary edge is a visible edge and we apply the same polygon-filling algorithm as for constrained edge insertion.

We allow boundary vertex relocations as long as they do not create a self-interseting boundary. This leaves the following three cases:

1. *Relocating the boundary vertex along one of its two existing boundary edges.* This is the simplest case and is similar to relocating an interior vertex. We insert a boundary vertex at the new location and then remove the boundary vertex at the old location.

2. *Relocating the boundary vertex into the exterior of the tile.* We reassign the vertex's coordinates, but there may be edges attached to the vertex that now intersect the boundary. Therefore, we remove all attached non-boundary edges and then fill the resulting polygon (Fig. 9).

3. *Relocating the boundary vertex into the interior of the tile.* In this case, there

may be edges and vertices that now lie outside the boundary. We walk along both boundary edges of the relocated vertex to remove any intersecting edges, perform a flood search to identify and remove any remaining geometry that lies outside the boundary, and fill the resulting polygon (Fig. 10).

In cases 2 and 3, the polygons do not necessarily have visible edges. Therefore, we must use a polygon-filling algorithm that works for arbitrary simple polygons. We borrowed an existing Java implementation of Seidel's [13] algorithm from the open-source J3D[1] project.

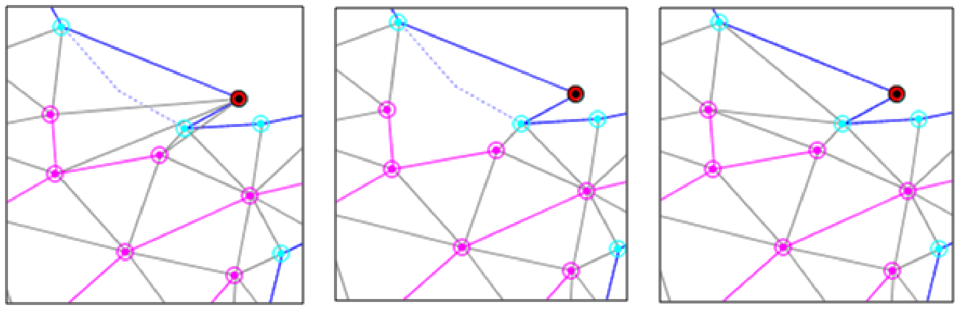


Figure 9: Relocating a boundary vertex into the exterior: (a) moving the vertex causes the attached edges to intersect the boundary; (b) the attached edges are removed, leaving a hole; and (c) the hole is filled.

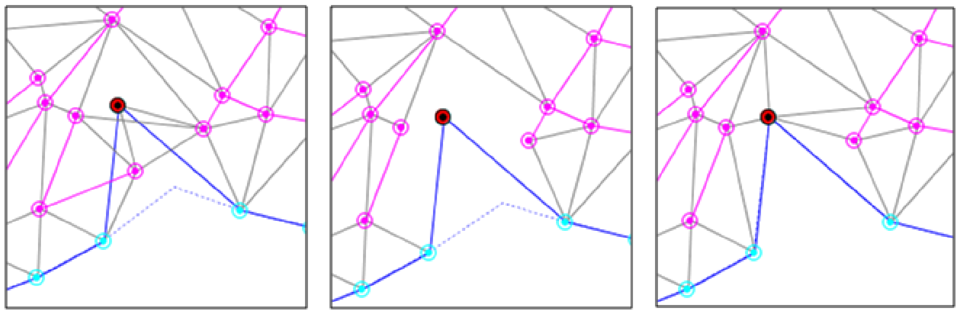


Figure 10: Relocating a boundary vertex into the interior: (a) moving the vertex creates intersections and leaves geometry dangling outside the boundary; (b) the intersecting and dangling geometry is removed; and (c) the hole is filled.

[1]`http://code.j3d.org/`

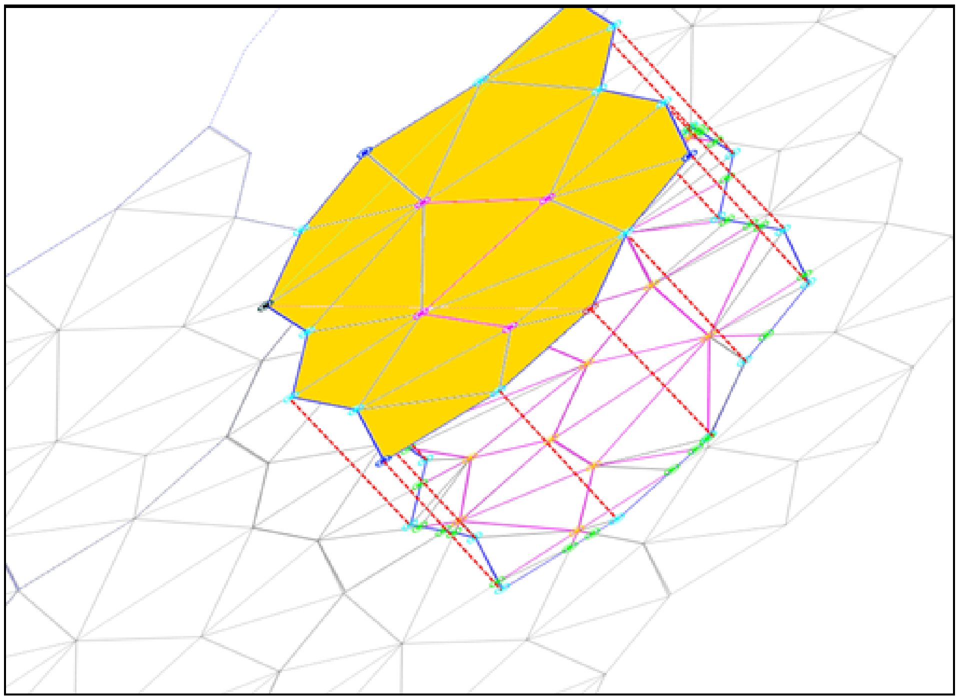


Figure 11: The yellow tile mesh serves as the top face of the 2½D tile, but the bottom face must be cut out of the underlying layer that has been laterally offset.

4 Mesh-Cutting Algorithm

The bottom mesh of a laterally-offset 2½D Escher tile is generated using a cookie-cutter operation that crops the appropriate geometry from the top meshes of the tiles in the underlying layer (Fig. 11), allowing the layers to align properly and fit together seamlessly (Fig. 6). Once a lateral offset has been specified (Fig. 12a), we form the cookie cutter by copying the tile mesh, retaining only the boundary edges and triangulating the interior with temporary edges (Fig. 12b) to facilitate subsequent vertex insertions. Then we move the cookie cutter to the offset location and walk along its boundary to find all the intersections with the underlying landscape. The underlying landscape is made up of many copies of the tile mesh, which have been subjected to the tile's symmetries. But when we start the boundary walk, we only load the copy that contains the first vertex of the cookie-cutter boundary. As the cookie cutter traverses boundary edges in the landscape, we load on demand the appropriate adjoining mesh copy. This on-demand approach bypasses the problem of determining *a priori* the planar extent of the cookie cutter across the mesh copies forming the underlying landscape.

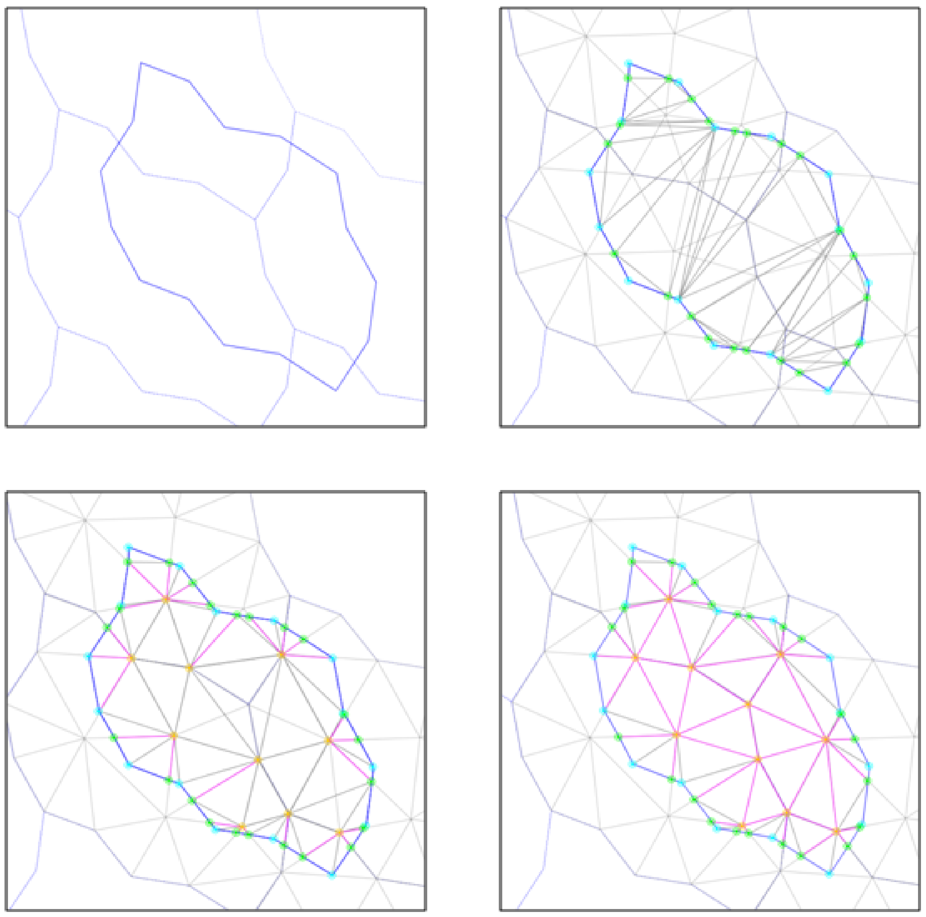


Figure 12: Cookie cutter operation: (a) a cookie cutter contour (blue) lies offset from the underlying landscape (gray), whose meshes will subsequently be loaded on demand; (b) intersection points (green) are found by walking the cookie cutter boundary, which is temporarily filled with interior edges (gray) to enable later vertex insertions; (c) edge fragments in the active queue are added as constrained edges (magenta); and (d) the remaining landscape edges lying inside the cookie cutter are found via flood fill and added as constrained edges (magenta).

When the cookie cutter crosses an underlying edge, the intersection point is added to its boundary (Fig. 12b), and the fragment of the edge projecting inside the cookie cutter is retained and placed in an *active fragment* list. To catch all the fragments that might arise when an underlying edge is bisected multiple times from different directions, new fragments are checked for intersections with all existing fragments in the list. As an additional optimization, we have implemented separate lists for each underlying edge to reduce unnecessary intersection tests. When the boundary walk finishes, all active fragments and their endpoints are added to the interior of the cookie cutter, and are marked as constrained edge segments (Fig. 12c). The algorithm performs a flood search on the endpoints to collect any remaining edges of the landscape that lie inside the cookie cutter (Fig. 12d). The resulting cookie-cutter mesh now contains all of the cropped geometry of the underlying landscape and becomes the bottom mesh of the 2½D tile. Quadrilateral side-wall faces are added between corresponding boundary edge segments in the top and bottom meshes, and the resulting watertight boundary representation can be output in .STL format, which is understood by almost all layered manufacturing machines.

The cookie-cutter problem was another motivation to implement our own constrained Delaunay triangulation library in Java. Tackling this problem with Triangle, for example, is possible but not straightforward. In particular, we would have to identify the extent of the underlying mesh needed to cover the contour, and copy, transform, and aggregate that geometry at the start (as opposed to loading the geometry on demand). Next, we would constrain all of the edges to prompt Triangle's automatic splitting of intersecting edges, and finally remove all geometry exterior to the contour using Triangle's "triangle-eating virus" mechanism [17]. However, many 2½D tile designs utilize offsets that create coinciding vertices and/or partly coinciding edge segments between the cookie cutter and the landscape, e.g., when an interior feature on the top mesh is lined up with a boundary feature on the bottom mesh. While Triangle would compute exact intersections and retain finely detailed geometry, our algorithm instead merges vertices, so that multiple intersections within an ϵ-neighborhood are represented by only a single vertex in the cookie-cutter mesh. Although this merging does not result in an exact solution, the resulting cookie-cutter mesh retains enough detail to be accurately printed on a layered manufacturing machine.

5 Visual Debugging

Sophisticated algorithms are difficult to debug using traditional text-based debuggers and methods. This is especially true of geometric algorithms, because the data is inherently visual and difficult to interpret when presented as textual output, e.g. as coordinate values. Visual debugging output can leverage the tuned visual processing capabilities that humans enjoy.

In the case of the triangulation and mesh-cutting algorithms described above, we found that animating and visualizing the algorithms helped us to identify extreme, difficult, and unexpected cases. Moreover, we were able to isolate and visualize subtasks within the algorithm, as opposed to waiting to receive one visual output at the termination of the algorithm and then working backwards to determine the faulty subtask.

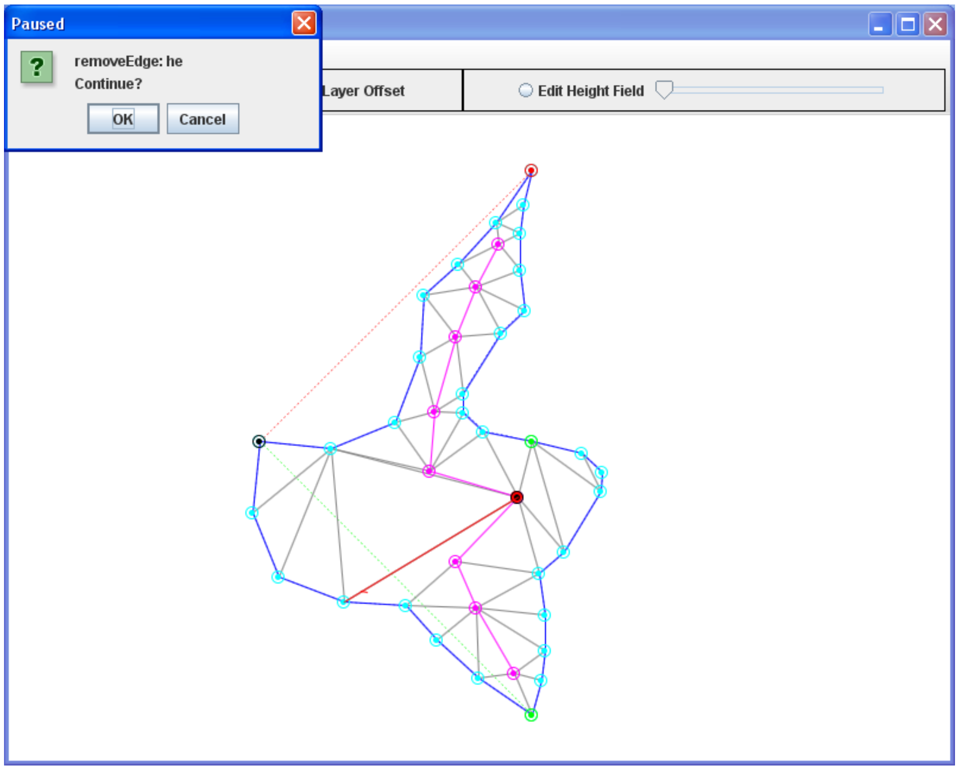
We implemented our visual debugging system in Java2D by overriding the event-queue repainting mechanism and inserting events mid-algorithm to create visual "breakpoints." In addition, these breakpoints can take geometric objects, such as points or halfedges, as arguments, causing them to be highlighted in the visual debugging output (Fig. 13).

The idea of implementing a visual debugging system for geometric algorithms is not new. GeoLab[2] and GeoSheet[3] are examples of feature-rich environments

[2] `http://www.ic.unicamp.br/~rezende/GeoLab.htm`

[3] `http://www.ece.northwestern.edu/~dtlee/theory/gs_tech_1_html/gs.html`

for developing, testing, visualizing, and animating geometry algorithms in C/C++. Both packages wrap low level graphics libraries such as Xfig and Xlib to provide convenient graphical output calls that can be inserted mid-algorithm, similar to our breakpoint feature. We chose to implement our own visual debugging system because we needed a Java-based solution and because of the ease of integrating it into our existing application, which already displayed triangulation output in Java2D and supported saving and loading dynamically created test cases.



removeEdge: he
Continue?

Figure 13: A breakpoint during a visual debugging session: a halfedge and vertex are highlighted (red) and the system is waiting for user input to proceed.

6 Three-Dimensional Cubic-Lattice Tilings

There exist many more symmetry groups and tiling groups in 3-space than in the plane. In our 2½D tile editor, we have realized only four planar symmetry groups, but have created a modular framework that allows for incorporation of other tiling symmetries at a later time. For instance, we could use the complete set of 91 isohedral tiling parameterizations as categorized by Kaplan and Salesin [9]. To our knowledge, there is no similar categorization of 3D tiling groups, and the number of possibilities is much larger than in 2D.

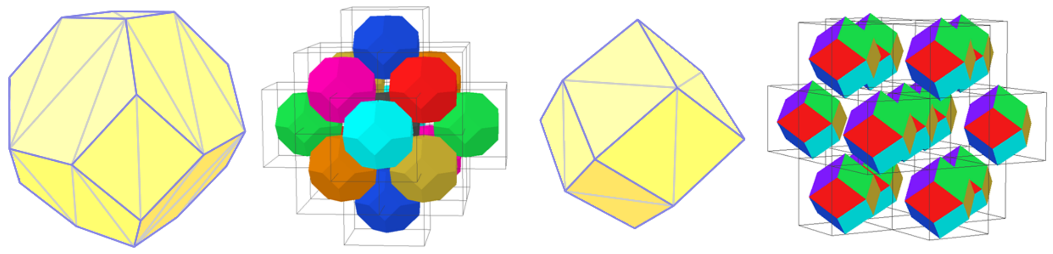


Figure 14: Fundamental domains of 3D tilings, and nearest neighbors: (a,b) a truncated octahedral cell based on the body centered cubic lattice; and (c,d) a rhombic dodecahedral cell based on the densest sphere packing.

In our 3D tile editor, we have created two tiling groups. For both, we start by displaying a corresponding polyhedron representing the fundamental domain. The first tiling is derived from the body centered cubic lattice, with a truncated octahedron as its fundamental domain (Fig. 14a, b). However, we only consider translational symmetries, and thus allow the user to perform arbitrary affine distortions of the underlying coordinate system. In this scheme, each cell has 14 nearest neighbors, and its fundamental domain can always be represented as a polyhedron with 7 pairs of opposite, parallel, and identical faces. A second tiling that we have explored is based on the densest sphere packing, with the rhombic dodecahedron as its fundamental domain (Fig. 14c, d). Again, since we only consider translational symmetries, this domain can be distorted into a polyhedral shape with 6 pairs of opposite, parallel, and identical faces.

6.1 Pane-based Editing Workflow

Initially, the 3D fundamental domains are not offered to the user as objects that can be freely edited in a 3D domain. Instead, there is again a Phase I of the editing process, where we present the individual faces of the fundamental domain to the user as 2D "panes" that can be decorated with extra vertices and edges. These are later manipulated to create 3D free-form shapes during a second editing phase. To provide context, the 14 or 12 panes of the whole domain are always shown as a 3D object that can readily be rotated around its center of gravity with a "crystal ball" or "orbit" interface [2]. A "special" click into one of these panes of the fundamental domain snaps that pane into the display plane and loads the 2D editor described above. "Special clicking" subsequently into any of the other panes will initiate the most direct rotation that will bring that pane to the front, so as to preserve the orientation context and minimize the user's confusion [2].

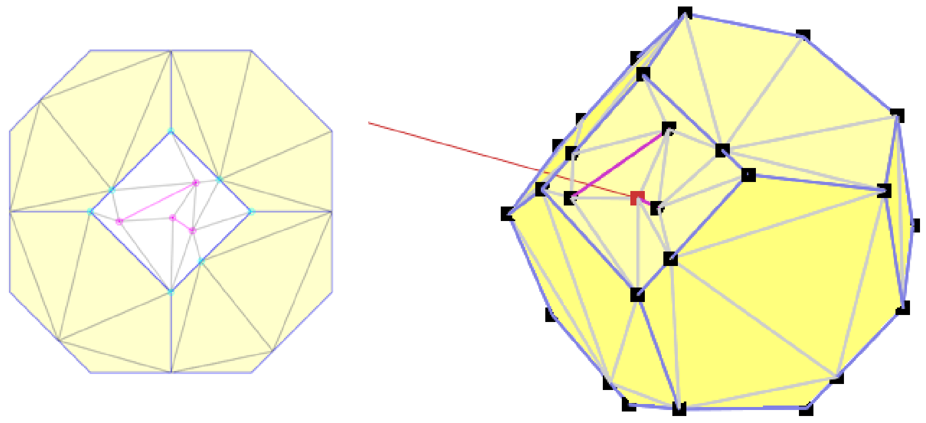


Figure 15: Pane-based construction of a 3D Escher tile: (a) Phase I: editing the triangulation of an individual pane; and (b) Phase II: free-form 3D editing of the pane.

In this 2D edit mode, any of the face's internal vertices and line segments can be added, deleted and moved, and will instantly result in a clean Delaunay triangulation obtained through our incremental algorithm (Fig. 15a). Boundary vertices can be added onto existing boundary line segments, but cannot be moved, because this would cause other faces of the fundamental polyhedron to become non-planar. For these 2D edits, each face uses its own local coordinate system with the origin at the centroid of the face. Another special click restores the 3D crystal ball view.

In Phase II of the 3D shape editing process, local 2D coordinates for each pane are transformed into 3D vertices that can now be manipulated in 3-space. The last vertex the user selects defines an axis through the center of mass of the tile, and the user can translate the entire selection parallel along that axis, or in the plane perpendicular to the axis. We interpret the cursor's position in the XY-plane of the view screen as if it were in the perpendicular plane after it has been rotated through the minimal angle that brings it parallel with the view plane (Fig. 15b).

If additional detail is needed beyond what is possible with the tesselated panes, new vertices can be added by subdividing a face or edge. These edits are kept as purely local changes, however, with no attempt to optimize the resulting mesh or clean up sliver triangles. If the designer cannot achieve the desired result, and needs many more vertices in a particular pane of the fundamental polyhedron, we provide a limited roll-back option. The designer may return to the 2D edit mode for that particular pane and modify its triangulation, then return to the Phase II 3D edit mode. During this transition, the 3D information of all the vertices associated with any other panes is maintained, and only the Delaunay mesh for the rolled-back pane is recalculated. Any 3D information that belongs solely to this modified pane needs to be re-entered from scratch.

We chose this pane-based workflow model as a trade-off between the needs for maintaining the logical equivalence between corresponding panes in the fundamental polyhedron and the desire for a truly free-form shape editor. We quickly rejected the idea of representing the whole 3D tile as a volumetric object partitioned into a collection of 3D tetrahedra. A constrained 3D Delaunay tetrahedralization does not even exist for all sets of constraints, and implementing a conforming Delaunay tetrahedralization code would be considerably more work than the 2D triangulation code we have already created [20]. Moreover, there is no need to modify the interior of the tile. Instead, we require only an appropriate boundary representation to display the tile on the screen and to manufacture it on a rapid prototyping machine.

6.2 User Interface Issues

Occlusion is probably the largest obstacle to free-form editing of 3D tiles. Because of the translation symmetry of the fundamental domains we have chosen, editing a face that is visible in the current view will cause changes to the opposite face of the tile, but the opposite side is occluded when the tile is rendered opaquely. Often, it is important to see the symmetry-induced changes on the opposite face. Creating a convex feature such as a fish fin on one side of the tile will create a concave feature such as an eye socket on the opposite side, and the designer may need to strike a delicate balance between concavity and convexity to achieve the desired artistic effect. To address this issue, we have implemented a dual-camera view that shows the complete convex/concave pairings (Fig. 16).

Because both 3D fundamental domains are based on a cubic lattice, they can be scaled and skewed into parallelepiped lattices and remain space-filling. What is the easiest way to manipulate this affine transform, which has nine degrees of freedom (three for the scale factors in each direction, and six additional skew factors)? We have created a widget with exactly nine control points, each restricted to one degree of freedom. The widget maintains the same orientation as the camera, and dragging a control point projects the XY motion of the mouse in the view plane onto the one-dimensional axis of the control point (Fig. 17).

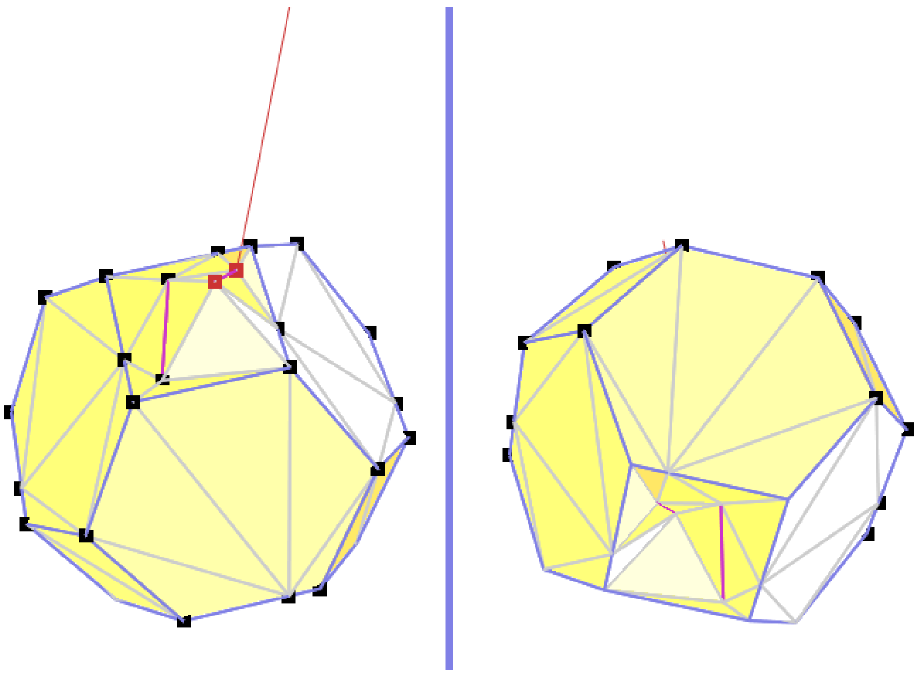


Figure 16: Dual cameras simaultaneously show a convex feature (left camera) and its corresponding concave feature (right camera).

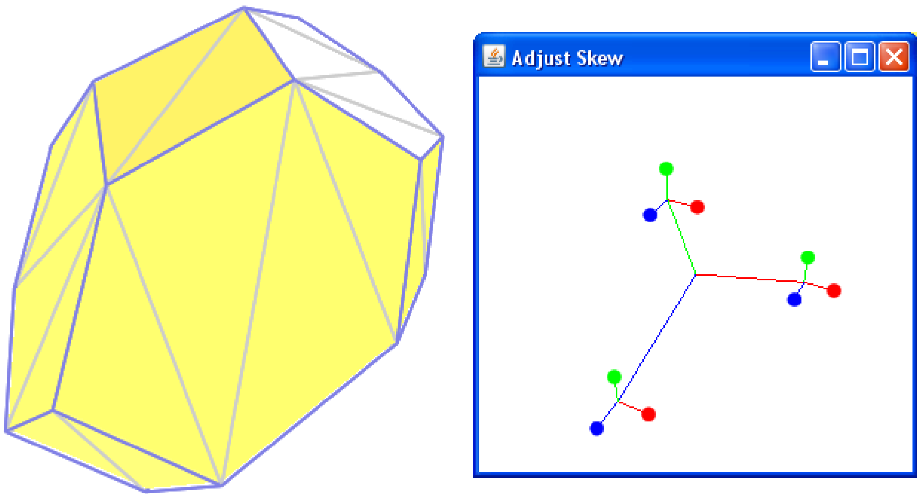


Figure 17: A skew widget provides 9 control points, each with one degree of freedom.

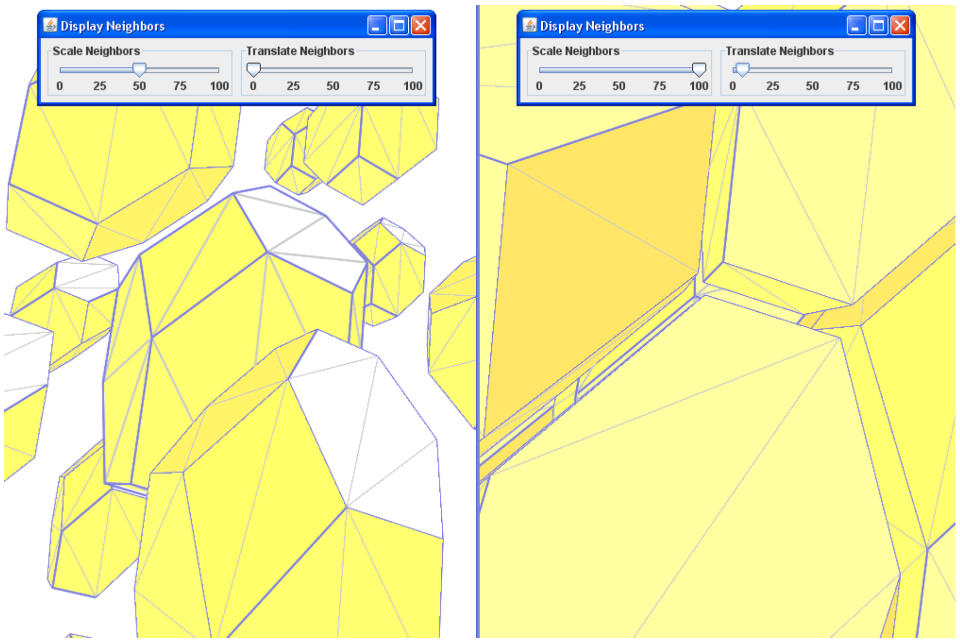


Figure 18: Displaying the nearest neighbors: (a) neighbors are scaled to 50% around their center of mass; and (b) neighbors are scaled to 100%, but translated slightly along the normal of their adjoining face, to reveal the interface between tiles.

3D tilings can have complicated interlocking features. We have experimented with rendering faces transparently and with omitting faces to reveal the interior faces of the opposite side of the tile, but in both cases the resulting display is cluttered and confusing. Instead, we display nearest neighbors that are scaled and translated to reveal the interface between adjoining tiles (Fig. 18). The scaling is applied in relation to each neighbor's center of mass, and the translation factor moves them radially outward along the normals of the faces where they adjoin.

6.3 2D vs. 3D Editing

Editing a 3D tile is inherently more restrictive than a 2D tile because of the lack of interior decorations, which in 3D would lie inside the tile's volume and hence be occluded in any opaque, solid rendering. A decoration of a 3D tile can only make use of the tile's boundary surface, which is entirely subject to symmetry constraints. Creating a desirable feature on one face of the tile will lead to a symmetrically linked feature on the opposite side of the tile, only with opposite concavity. Adding a concave feature such as an eye to a fish-shaped tile requires accommodating the symmetrically opposite convex feature, perhaps as a fin (e.g., Fig. 21c). In contrast, a 2D tile allows unrestricted addition of decoration vertices and edges within its interior, so that features that clarify the content of the tile, such as eyes and fins, can be freely "painted" in the tile's interior without regard to symmetry constraints.

In designing 2½D tiles, we can draw on an existing "vocabulary" of aesthetically pleasing 2D tilings from Escher's sketchbook, and use these as starting points (e.g., Fig. 19a). Unforunately, there is no similar 3D sketchbook. Possibly, as artists attempt to create 3D tilings, a larger vocabulary of complementary shapes will emerge.

7 Results

We have produced both 2½D and 3D tilings of 3-space. Starting with sketch 127 from Escher's sketchbook [12] (Fig. 19a), we traced a bird-shaped contour and added constraint edges (Fig. 19b) that could be used later to form ridges along the bird's back when we formed the height field (Fig. 19c). An offset was chosen to allow the thickness of the bird's head and body to complement the thinness of its wings (Fig. 20a). After editing the height field, the resulting 2½D tile fills 3-space in laterally offset sheets (Fig. 20b).

To create a 3D tile of a fish based on the rhombic dodecahedron lattice, we started with an earlier prototype that was created in the SLIDE [23] environment and uses Bézier patches for the 12 faces (Fig. 21a). We added control points analagous to Bézier control points to the initial rhombic dodecahedron in our 3D tile editor (Fig. 21b), along with additional control points and constraints for forming the fish's eyes and mouth. After free-form editing of these control points, we generated a 3D fish tile complete with fins, eyes, and mouth (Fig. 21c). The nearest neighbors (Fig. 22) join together to form a "school" of fish that fill 3-space.

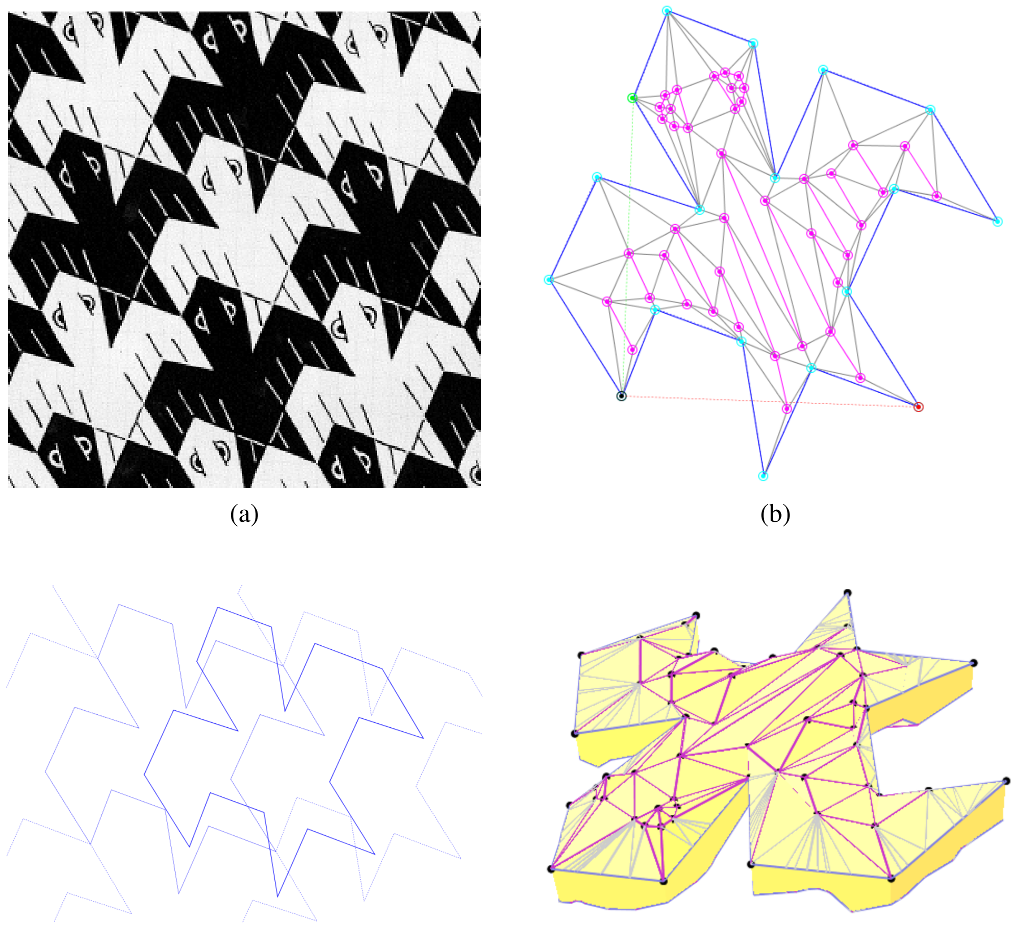
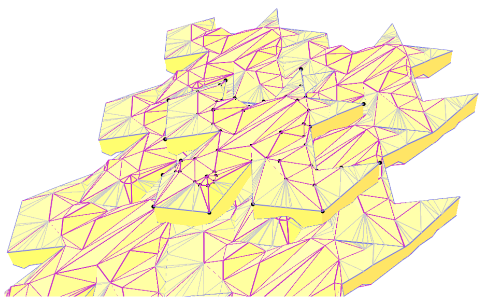

(a) (b) (c) (d)

Figure 19: A 2½D bird tile: (a) Escher's sketch 127 provides the basic contour; (b) interior vertices and constrained edges are added to provide control points for editing the height field; (c) the chosen offset; and (d) the edited height field.


(a)

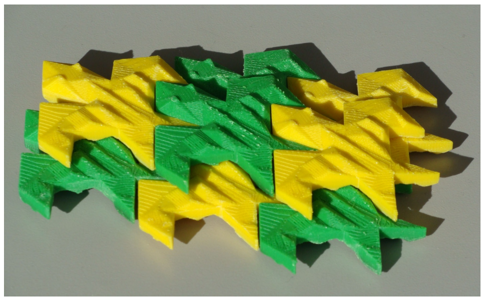

(b)

Figure 20: A 2½D offset tiling: (a) displaying the laterally offset layer that lies below the tile; and (b) manufactured tiles.

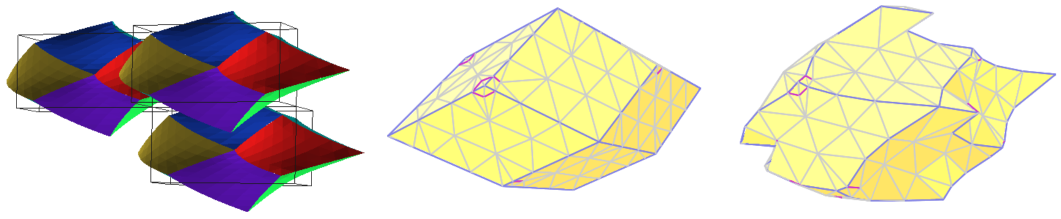
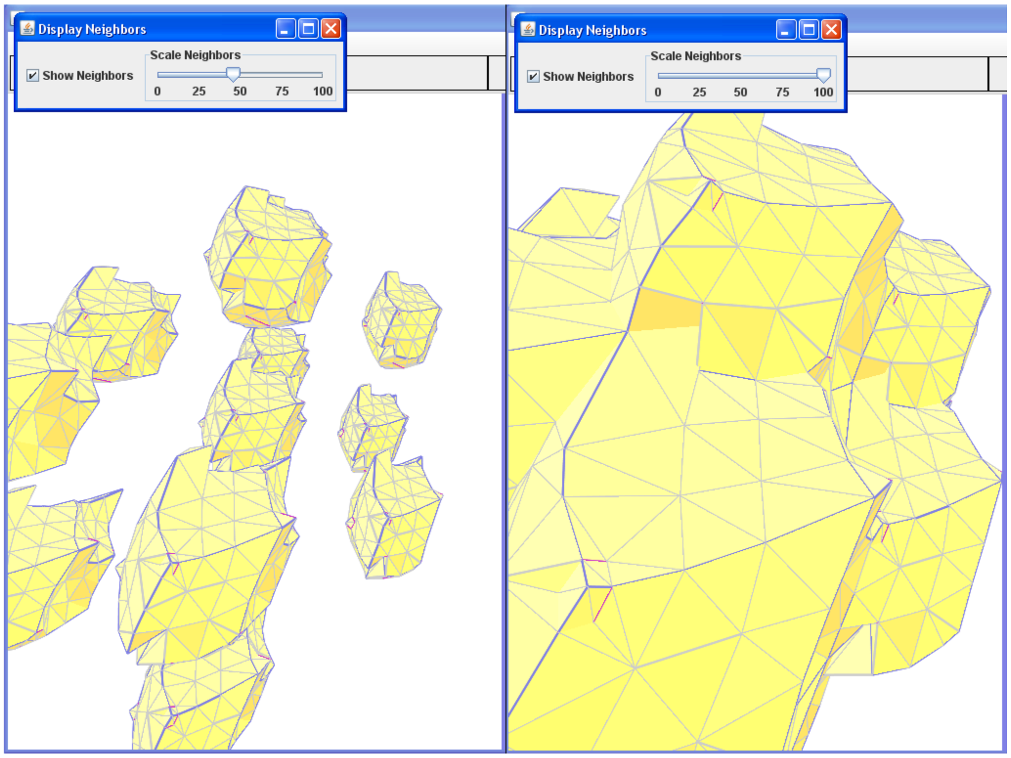


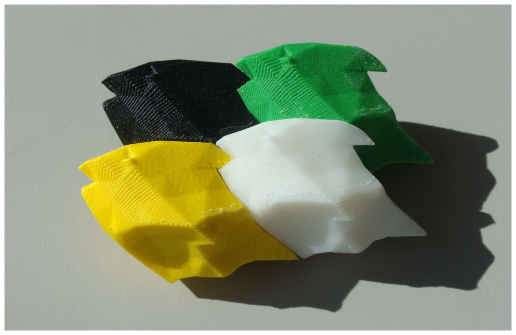
Figure 21: A 3D tiling based on the rhombic dodecahedron lattice: (a) an initial prototype that uses Bézier patches for the faces; (b) creating similar Bézier patch control points in our 3D editor, along with control points for eye and mouth features; and (c) the finished tile after free-form editing.

8 Conclusions

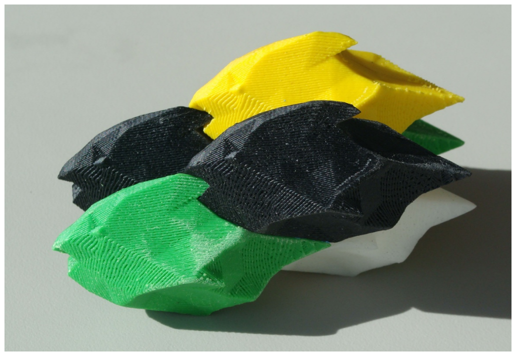
As part of the development of a 3D Escher-tile editor, we have addressed and overcome several interesting challenges. In particular, we have implemented a tile editor that is simultaneously easy to use while offering a fair amount of flexibility in the design of 2½D Escher-tile surfaces. A generalization of this 2½D "pane" editor forms the basis for our 3D tile editor, which allows more general deformation of the surface elements of an arbitrary genus-zero 3D Escher tile. An important side product of this work is our Java constrained Delaunay triangulation library, which



(a)



(b)



(c)

Figure 22: A "school" of 3D fish tiles: (a) nearest neighbors at 50% and 100% scaling, to demonstrate how they fill 3-space; and (b,c) manufactured tiles.

we have released as jmEscher under an open-source license on Google Code.[4] We anticipate that such a library will prove useful for other Java-based interactive mesh-editing applications.

The move from 2½D to 3D tile design introduces new user interface problems. When should the application perform Delaunay triangulation and how should a user specify control points, then manipulate them in 3D using the 2D input of a mouse? We have addressed these with our pane-based editing workflow and our use of radial selection axes. We also implemented dual cameras to help reveal features that are occluded during the editing process due to the symmetry constraints of the tile, and interactive display of the tile's nearest neighbors to show how the tiling interlocks. Finally, we have created a widget for specifying an affine transform, which enables global editing operations such as elongating the tile or skewing it.

Despite the implementation of specialized user interface features, designing 3D Escher tiles remains difficult. This is due to the nature of the symmetry contraints imposed on the surface of the tile, and a lack of existing 3D tessellated artwork to draw on. With some careful thought and planning, however, we have successfully produced examples of 2½D and 3D space-filling Escher tilings using these CAD tools.

References

[1] J. H. Conway, H. Burgiel, and C. Goodman-Strauss. *The Symmetries of Things*. A. K. Peters, Wellesley, MA, 2008.

[2] G. Fitzmaurice, J. Matejka, I. Mordatch, A. Khan, and G. Kurtenbach. Safe 3D navigation. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, pages 7–15, 2008.

[4] `http://code.google.com/p/jmescher`

[3] S. Fortune. A sweepline algorithm for Voronoï diagrams. *Algorithmica*, 2(2):153–174, 1987.

[4] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 163–172. Association for Computing Machinery, 1993.

[5] B. Grünbaum and G. C. Shephard. *Tilings and Patterns*. W. H. Freeman and Co., New York, 1986.

[6] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoï diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[7] D. E. Joice. Hyperbolic tessellations. `http://aleph0.clarku.edu/~djoyce/poincare/PoincareApplet.html`.

[8] T. C. Kao and D. M. Mount. Incremental construction and dynamic maintenance of constrained Delaunay triangulations. In *Proceedings of the 4th Canadian Conference on Computational Geometry*, pages 170–175, 1992.

[9] C. S. Kaplan and D. H. Salesin. Escherization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 499–510, 2000.

[10] C. L. Lawson. Software for C^1 surface interpolation. In J. Rice, editor, *Mathematical Software III*, New York, USA, 1977. Academic Press.

[11] D. Schattschneider. Computer software for tiling. `http://www.geom.uiuc.edu/software/tilings/TilingSoftware.html`.

[12] D. Schattschneider. *M. C. Escher: Visions of Symmetry*. W. H. Freeman and Co., New York, 1990.

[13] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[14] C. H. Séquin. Patterns on the Klein quartic. In *Bridges Conference*, pages 245–254, London, August 4–9 2006.

[15] C. H. Séquin. Intricate isohedral tilings of 3D Euclidean space. In *Bridges Conference*, pages 139–148, Leeuwarden, The Netherlands, July 24–28 2008.

[16] M. I. Shamos and D. Hoey. Closest-point problems. In *16th Annual Symposium on Foundations of Computer Science*, pages 151–162, Berkeley, CA, October 1975. IEEE Computer Society Press.

[17] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, pages 203–222, 1996.

[18] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18:305–363, 1997.

[19] J. R. Shewchuk. Lecture notes on Delaunay mesh generation. Computer Science Division, University of California at Berkeley, 1999.

[20] J. R. Shewchuk. Constrained Delaunay tetrahedralizations and provably good boundary recovery. In *Eleventh International Meshing Roundtable*, pages 193–204, 2002.

[21] Y. Shon. Escher tiling on the torus. CS285 course project, U.C. Berkeley, Berkeley, CA, May 2002.

[22] R. Sibson. Locally equiangular triangulations. *The Computer Journal*, 21(3):243–245, 1978.

[23] J. Smith. SLIDE: Educational rendering system for 3D interactive dynamic environments. Master's thesis, U.C. Berkeley, Berkeley, CA, May 2003.

[24] J. Weeks. Kaleidotile. http://www.geometrygames.org/KaleidoTile.

[25] J. Yen and C. H. Séquin. Escher sphere construction kit. In *Interactive 3D Graphics Symposium*, pages 95–98, March 19–21 2001.