Computer-Aided Design and Applications

http://www.cadanda.com

CAD Tools for Creating Space-filing 3D Escher Tiles

Mark Howison[1] and Carlo H. Séquin[2]

[1]University of California, mhowison@berkeley.edu
[2]University of California, sequin@cs.berkeley.edu

ABSTRACT

We discuss the design and implementation of CAD tools for creating decorative solids that tile 3-space in a regular, isohedral manner. Starting with the simplest case of extruded 2D tilings, we describe geometric algorithms used for maintaining boundary representations of 3D tiles, including a Java implementation of an interactive constrained Delaunay triangulation library and a mesh-cutting algorithm used in layering extruded tiles to create more intricate designs. Finally, we demonstrate a CAD tool for creating 3D tilings that are derived from cubic lattices. The design process for these 3D tiles is more constrained, and hence more difficult, than in the 2D case, and it raises additional user interface issues.

Keywords: isohedral tilings, 3D tile generator, constrained Delaunay triangulation.
DOI: 10.3722/cadaps.2009.737-748

1. INTRODUCTION

M. C. Escher's intricate tilings are well known [9] 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. 1a). However, without the help of computer graphics tools, it is rather difficult and labor intensive to create aesthetically pleasing tilings of this kind. Because of the wide-spread interest in such patterns, many easy-to-use graphics tools have been created and made available on the web, allowing members of the general public with no special training in the graphics arts or in computer science to experiment with and generate innovative regular patterns [10].

Fig. 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.

Such tilings can also be created in non-planar domains. Fig. 1(b) shows a spherical tiling made from 60 identical tiles [18] that were fabricated on a rapid prototyping machine, and Fig. 1(c) displays 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 shared vertices. 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, e.g. [6], and also for the spherical domain, e.g. [19]. In some isolated experiments, Escher-like tiling patterns have also been placed on symmetric surfaces of higher genus, e.g., onto a torus [16] and onto a genus-3 surface with tetrahedral symmetry [12], as in Fig. 1(d). 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 fill 3-space regularly and seamlessly. This exploration space is much larger than the 2D domain. First, there are many more symmetry groups in 3-space than in the plane. Second, the 3D tiles can be of a genus higher than zero, they can interlink with their neighbors, and they can even be knotted! An exploratory paper [13] surveys many of these possibilities, and concludes that different approaches and tools would be needed to design such tiles.

In this paper, we are mainly concerned with CAD tools that help in the construction of isohedral tiles of genus zero, with complex (possibly free-form) surfaces, and which may or may not resemble shapes found in nature. In the 3D domain, 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 both the limitations of projecting a 3D object onto a 2D viewing screen and the geometric interdependencies caused by the imposed symmetries. With 2D tilings, the prototype tile and its nearest neighbors can readily be displayed in one comprehensive view, but this is no longer the case for 3D tilings. If we deal with only one isolated tile, then we can see at most half of its surface, and if we display more than one tile, we may encounter occlusions. 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 then 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.

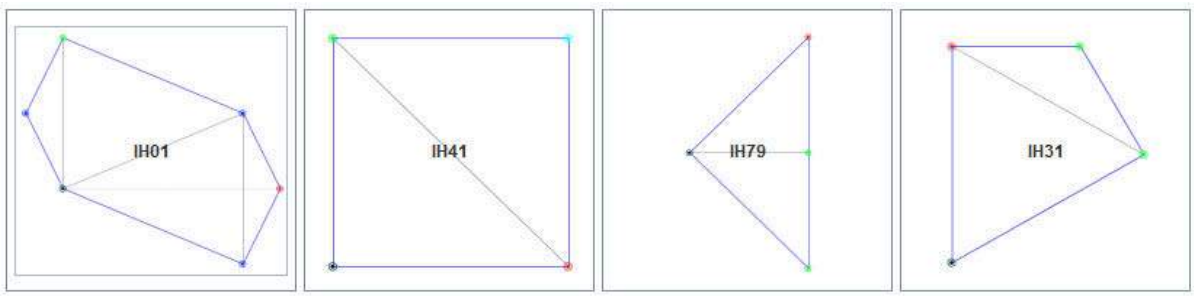

Fig. 2: Four 2D symmetry groups: (a) IH01, hexagonal domain with translational symmetry; (b) IH41, rectangular domain with translational symmetry; (c) IH79, right-triangle domain with 4-fold rotational symmetry; and (d) IH31, kite-shape domain with 6-fold rotational symmetry.

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 [4]; Conway notation: 0 [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 mid point (Fig. 3b,c & 4). 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.

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 one of the symmetry groups and is given a basic shape that represents the fundamental domain of this group. This tile can now be modified in the context of all its neighbors and of 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 constrained line segments (Fig. 3a, 4a).

Fig. 3: Simple 2½D Escher tiles: (a) height-editing of the top surface; (b) 3 identical tiles, the white ones seen from top, the red one seen from bottom; (c) the 3 tiles stacked on top of one another.

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, single-valued height field by moving vertices — both on the boundary and in the interior of the tile — up or down in the vertical direction (Fig 3a). 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 and all their heights 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 (not particularly interesting) default case, the bottom surface of the tile is modified in exactly the same way as the top surface (Fig. 3b), so that the 3D tiles would then stack on top of one another forming a single prismatic column (Fig. 3c). And of course these columns would fit together laterally and fill all of 3-space.

In a more interesting 3D isohedral tiling, subsequent layers of these tiles will 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. So far, we have restricted ourselves to isohedral tilings, so 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 edit process, by grabbing the proto-tile and shifting it laterally with respect to the complete 2D tiling. 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, water-tight, 2-manifold boundary representation

of the 2½D Escher tile. In the translational symmetry cases, the lateral offset between two tile layers can be arbitrary; any constant shift between two subsequent layers will lead to all identical tiles. In the case of the rotational symmetry groups, however, only very special offsets will result in an isohedral tiling, i.e., a shift along one of the legs of the fundamental triangle for IH79 (Fig. 4).

Fig. 4: Fancier 2½D tiles: (a) proto-tile of tiling type IH79; (b) top and bottom view of the extruded 2½D tile; (c) four tiles put together in one layer and two more tiles placed in layer above.

2.1 Interactive Constrained Delaunay Triangulation

Top and bottom surfaces of the resulting 2½D Escher tile are represented as triangle meshes. Only points that were entered into the proto-tile during Phase I of the editing process can be used to define the shape of the final tile. Thus, enough points and poly-lines need to be drawn in Phase I, so that desired features such as the eyes, mouth, and fins of a fish, or the rib-patterns of a leaf, can be formed by vertical extrusion during Phase II of the shape editing.

We create a mesh of good quality 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 (Fig. 5a). 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. Writing a truly robust, high-performance library for constrained Delaunay triangulation is a non-trivial task, and we contemplated using the well-tested *Triangle* package [14]. 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 platform-independence.

Therefore, we decided to implement an incremental algorithm for creating and modifying a Delaunay triangulation on the fly as the designer performs editing operations. Potentially, this could amount to 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 avoid inconsistencies [14], but there are also less efficient ways to achieve robustness that are not as difficult to implement as adaptive routines. Moreover, our editor has different goals and constraints that do not call for a

high level of precision. First, the 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; 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 is not acceptable, the designer can always grab a vertex in question and move it to a slightly different location. We need to invoke robust routines in only a few places in our code. Thus with some careful consideration of critical cases, we have written a quasi-robust triangulation algorithm without resorting to the more complex (but more efficient) adaptive precision approach used by *Triangle*. Having a triangle mesh available during Phase I of the editing process also 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.

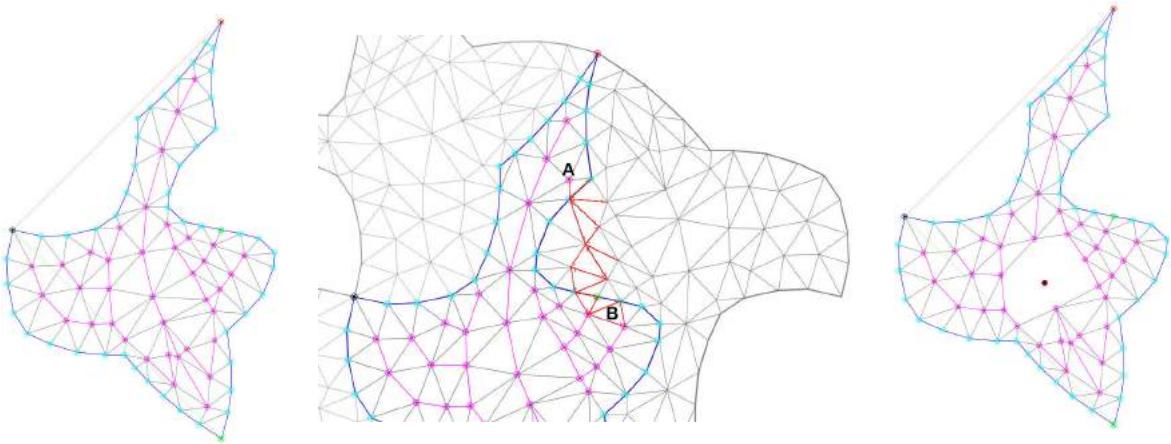

Fig. 5: Incremental Delaunay triangulation: (a) the mesh of a decorated fish tile with constrained edges (magenta); (b) a mesh search from vertex *A* to face *B,* which exploits the symmetry of the tile by moving directly through an adjacent copy; (c) the n-gonal hole resulting from removing a vertex.

Our implementation allows the user to add, move, and remove boundary and interior vertices, to constrain edges, and even to intersect constraint edges, all in real-time. We use Lawson's incremental insertion algorithm [8] to compute the Delaunay triangulation, which iteratively adds vertices to the existing triangulation and is therefore well-suited to our interactive approach. The mesh is backed by a half-edge data structure [3]. Most editing operations involve searches over the half-edge structure and modification and re-linking of the half-edges. For example, the vertex insertion operation starts by searching for the face in the existing mesh that contains the site of the new vertex. Our heuristic for choosing a starting site for the search is to use the last insertion site, since designers will often add features as localized groups of vertices. The search moves from triangle to triangle in the direction of the new vertex location. It exploits the symmetry of the overall tiling and follows a direct path to the insertion site by moving through adjacent, transformed copies of the tile if necessary (Fig. 5b). Once the containing face is found, the new vertex is inserted by creating additional half-edges and re-linking the data structure to split the existing triangle into three new faces. If the inserted point lies too close to an existing edge, we instead snap the point to that edge and split the two adjacent faces.

Temporary polygonal holes do arise (Fig. 5c) when vertices are moved or removed, and they require a polygon filling algorithm that is robust for the types of simple polygons that arise in these situations. For ease of implementation, we use an $O(n^2)$ algorithm published by Anglada [1], which starts with an edge that is "visible" to the other vertices of the polygon, meaning that those vertices can be connected to the edge's endpoints without leaving the polygon. The algorithm finds the vertex with the largest circum-circle through the edge, and uses that vertex and the 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. Although there exist more refined polygon-filling algorithms with running times ranging from $O(n \log n)$ to the theoretical lower bound of $O(n)$ [11], they are more complicated to implement, and in practice we are typically filling small polygons, where speed is not an issue. In an improved version of our library, we could implement Seidel's $O(n \log^* n)$ randomized incremental algorithm [11] as a reasonable trade-off between complexity and speed.

As mesh editing operations are performed, modified half-edges are added to a running "Delaunay test" queue that is cleared by performing an in-circle Delaunay test on each edge, flipping the edge if necessary. When an edge is flipped, its four neighbors are subsequently added back onto the queue. This process is guaranteed to terminate because the circum-radii of the triangles are strictly decreasing and there are only finitely many triangulations. Proofs of this and other mathematical properties of the algorithm are available in a succinct treatment by Sibson [17].

2.2 Cookie-Cutter Operation

The bottom mesh of a laterally-offset 2½D Escher tile has to be generated using a cookie-cutter operation that crops the appropriate geometry from the top meshes of the tiles in the underlying layer, allowing the layers to align properly and fit together seamlessly. Once a lateral offset has been specified (Fig. 6a), we form the cookie cutter by copying the tile mesh, retaining only the boundary edges and triangulating the interior with temporary edges (Fig. 6b) to facilitate subsequent vertex insertions. Then we move the cookie cutter to the offset location and walk its boundary to find all 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.

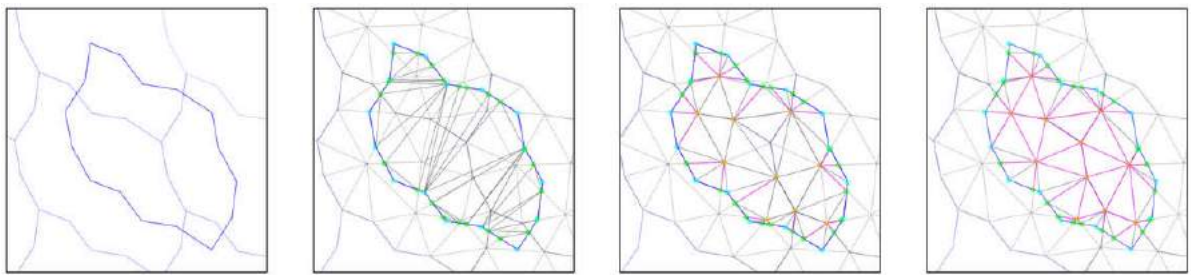

Fig. 6: Cookie-cutter operation: (a) a cookie-cutter contour (blue) has been offset from the underlying landscape (gray), whose meshes will subsequently be loaded on-demand; (b) intersection points (green) are found by walking the cookie-cutter boundary, which is temporarily filled with interior edges (gray) to enable later vertex insertions; (c) edge fragments in the active queue have been added as constraint edges (magenta); and (d) the remaining landscape edges lying inside the cookie cutter have been found via flood fill and added as constraint edges (magenta).

When the cookie cutter bisects an underlying edge, the intersection point is added to its boundary (Fig. 6b), and the fragment of the edge projecting inside the cookie cutter is retained and placed on an "active fragment" queue. In order 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 queue. As an additional optimization, we have implemented separate queues 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. 6c). The algorithm performs a "flood fill" on the endpoints to collect any remaining edges of the landscape that lie inside the cookie cutter (Fig. 6d). 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 water-tight boundary representation can be output in .STL format, which is understood by almost all layered manufacturing machines.

The cookie-cutter problem was another motivation for our decision to implement a constrained Delaunay triangulation algorithm in Java. Tackling this problem with *Triangle,* for example, is possible but not straight-forward. In particular, we would have to identify the extent of the landscape needed to cover the cookie cutter, and copy, transform, and aggregate that geometry at the start (as opposed to loading the landscape 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 cookie cutter by using *Triangle*'s "triangle-eating virus" mechanism [14]. 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. In these cases *Triangle*'s use of adaptive precision arithmetic would likely create many undesirable sliver triangles.

3. THREE-DIMENSIONAL CUBIC-LATTICE TILINGS

There exist many more symmetry groups and tiling groups in 3-space than in the plane. In our prototype 2½D implementation, 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 [7]. To our knowledge, there is no similar categorization of 3D tiling groups, and the number of possibilities is much larger than in the 2D case.

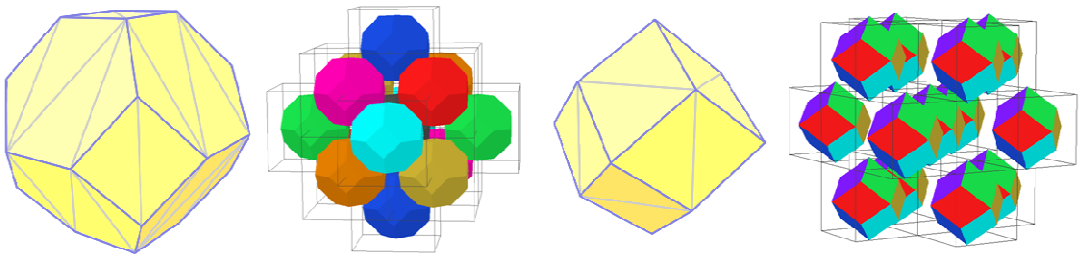

Fig. 7: Fundamental domains of 3D tiling, and nearest neighbors for: (a) a truncated octahedral cell based on the body centered cubic lattice; and (b) a rhombic dodecahedral cell based on the densest sphere packing.

In our 3D tile editor, we have implemented two tiling groups so far. 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. 7a). 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. 7b). 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.

3.1 Pane-based Editing Workflow

Initially, these fundamental domains are not offered to the user as 3D 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 (Fig. 8a). 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 [5]. A right click into one of these panes of the fundamental domain snaps that pane into the display plane and loads the 2D editor described above. Right-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 [5].

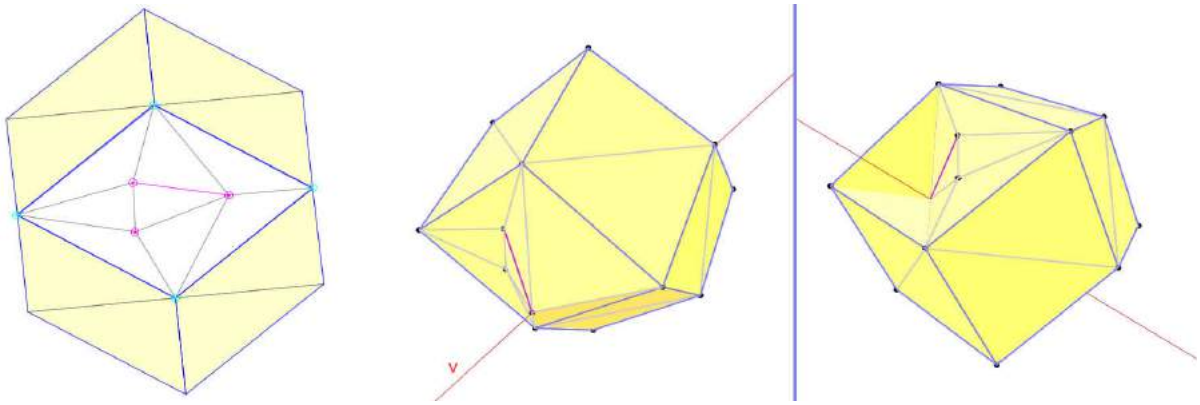

Fig. 8: Pane-based construction of a 3D Escher tile: (a) a pane of the dodecahedral tile available for 2D editing and Delaunay triangulation; and (b) free-form 3D vertex editing with dual cameras and a radial selection-and-extrusion vector v (red).

In this 2D edit mode, any of the face’s internal vertices and line segments can be added, deleted or moved, resulting in a Delaunay triangulation obtained through our interactive algorithm (Fig. 8a). Boundary vertices can be added onto existing boundary line segments, but they cannot be moved, since this would cause other faces of the fundamental polyhedron to become non-planar. For the 2D edits, each face uses its own local coordinate system with the origin at the centroid of the face. Another right 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 be manipulated in 3-space (Fig. 8b). The last vertex the user selects defines an extrusion vector through that vertex. The user can now translate all selected vertices parallel to this vector, or in the plane parallel to the edit pane. We interpret the cursor’s position in the XY -plane of the view screen as if it were in the plane that is parallel to the edit pane, but rotated through the minimal angle that brings it parallel to the screen.

If, during this 3D edit mode, additional detail is needed beyond what is possible with the tesselated panes, new vertices can be added by sub-dividing an individual face or edge. These edits are kept as purely local changes, however, with no attempt to optimize the resulting mesh or to clean up sliver triangles. If the designer still 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 good 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 creating robust 2D triangulation code [15]. Moreover, there is no need to modify the interior of the tile. Instead, we

require only a consistently-oriented boundary representation in order to display the tile on the screen and to manufacture it on a rapid prototyping machine.

3.2 User Interface Issues

Occlusion is probably the largest obstacle to free-form editing of 3D tiles. Because of the translational symmetries associated with the fundamental domains, 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 corresponding 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 simultaneously displays these convex/concave pairings (Fig. 8b).

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 found that displaying nearest neighbors that are scaled about their centers of mass was the clearest way to reveal the interface between adjoining tiles (Fig. 9a).

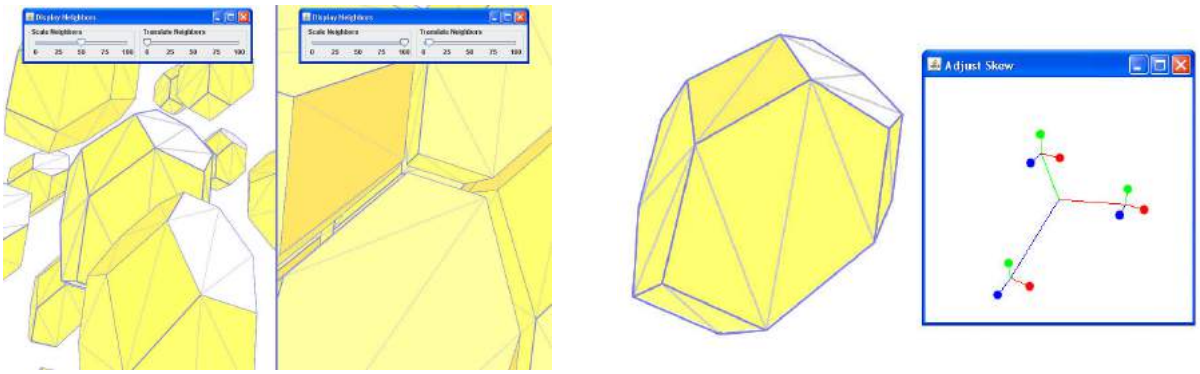

Fig. 9: User interface features: (a) displaying the nearest neighbors scaled about their centers of mass, to reveal the interface between tiles; and (b) a skew widget provides 9 control points each with one degree of freedom.

Because both fundamental domains are based on cubic lattices, but with only the translational symmetries enforced, they can be scaled and skewed into parallelepiped lattices and remain space-filling. Such an affine transform has nine degrees of freedom, or nine entries in its matrix representation. For each of the *X*, *Y*, and *Z* directions, there is a scale factor in that direction and two skew factors in the other directions. We have created a widget with exactly nine control points, each restricted to one degree of freedom and corresponding to an entry in the skew matrix. The widget maintains the same orientation as the view 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. 9b).

4. RESULTS

We have produced both 2½D and 3D tilings of 3-space. In designing 2½D tiles, we can draw on an existing "vocabulary" of aesthetically pleasing 2D tilings from Escher's sketchbook [9]. Starting with Escher's sketch number 127 (Fig. 10a), we traced a bird-shaped contour and added constraint edges (Fig. 10b), which were used later to form ridges along the bird's back when we edited the height field (Fig. 10c). A lateral offset was chosen to allow the thickness of the bird's head and body to complement the thinness of its wings. After editing the height field, the resulting 2½D tile fills 3-space in laterally-offset sheets (Fig. 11).

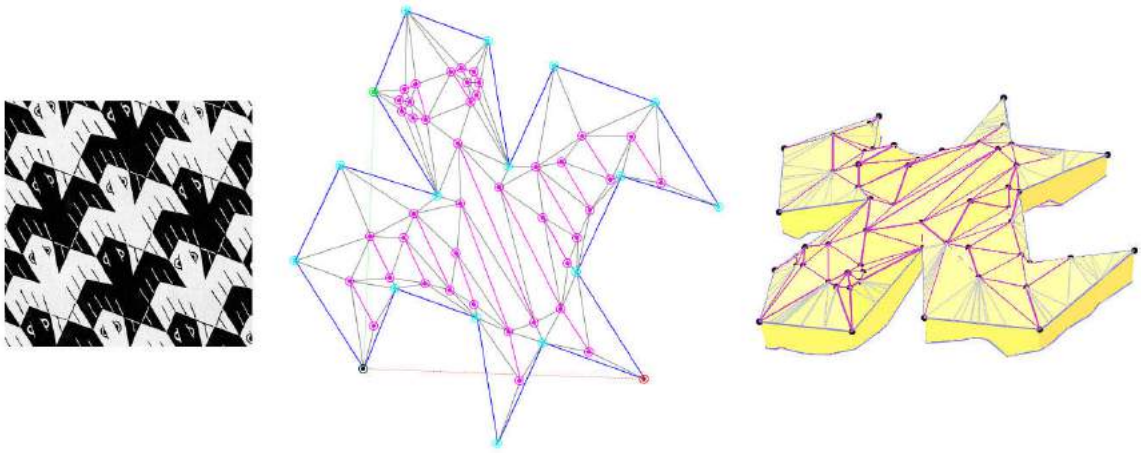

Fig. 10: A 2½D bird tile: (a) Escher's sketch number 127 provides the basic contour; (b) interior vertices and constraint edges are added to provide control points for editing the height field; and (c) the edited height-field.

Fig. 11: An offset, layered 2½D bird tile: (a) manufactured tiles fit together to form a layer in 3-space; and (b) several layers stacked with a lateral offset.

Designing an attractive 3D Escher tile derived from a 3-dimensional lattice is inherently more difficult than creating 2D tilings. Because we assume an opaque solid tile in the 3D case, and are not structuring in any way the interior volume of the tile, only the boundary representation of its surface is editable; and this entire surface is constrained to fit seamlessly with adjacent tiles. In contrast, a 2D tiling only has symmetry constrains on the 1D border, and the fully visible 2D surface of the tile can be decorated freely to clarify the content of the tile. Moreover, any decorations that are imprinted on the surface of a 3D tile will automatically create complementary features in the opposite location on the tile surface. Adding a concave feature such as a mouth to a fish-shaped tile requires accommodating it with a convex feature such as a fin.

Unlike in the 2D case, there is no Escher sketchbook available for tracing an initial 3D shape, although a vocabulary of 3D shapes may emerge as artists attempt to create 3D tilings. In the meantime, we created a 3D tile of a fish based on the rhombic dodecahedron lattice by starting with an earlier prototype that uses Bezier patches for the 12 faces to achieve a fish-like shape (Fig. 12a). With this shape in mind, we created a rhombic dodecahedron with similar control points and skew in our 3D tile editor, and used the pane editor to add control points and constraints for forming the fish's eyes and

mouth (Fig. 12b). After free-form editing, we generated a 3D fish tile complete with fins, eyes, and mouth (Fig. 12c). This boundary representation was then sent to a Fused Deposition Modeling machine to create some physical tiles in different colors. Nearest neighbors join together to form a "school" of fish that fill 3-space (Fig. 13).

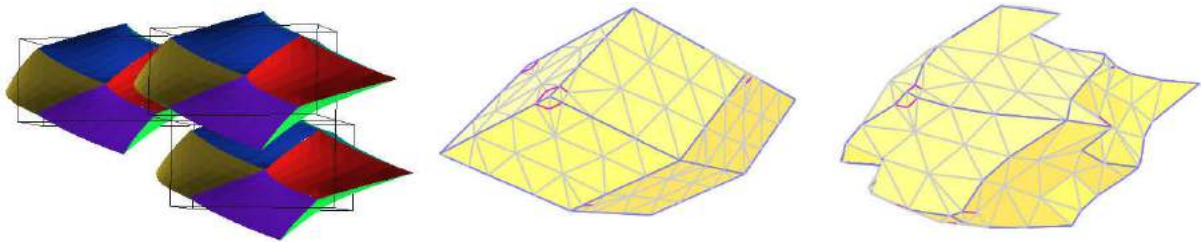

Fig. 12: A 3D tiling based on the rhombic dodecahedron lattice: (a) an initial prototype that uses Bezier patches for the faces; (b) creating similar Bezier patch control points in our 3D editor, along with control points for eye and mouth features; and (c) the finished tile after free-form editing.

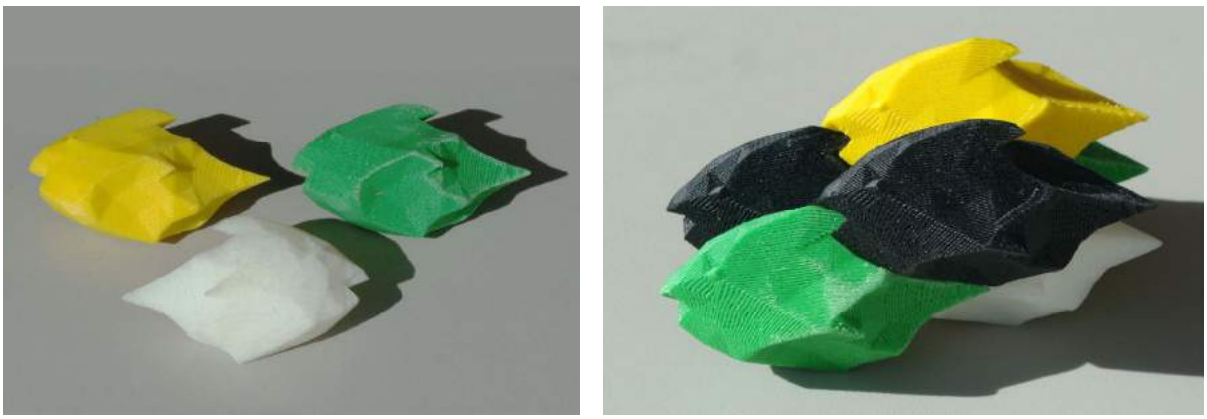

Fig. 13: Prototypes of true 3D fish tiles: (a) isolated tiles, front and back; (b) a tight 3D assembly of several tiles.

5. SUMMARY AND CONCLUSIONS

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 of an arbitrary genus-zero 3D Escher tile. An important side product of this work is our Java constrained Delaunay triangulation library, which we have released as *jmEscher* under an open-source license on Google Code (http://code.google.com/p/jmescher). We anticipate that such a library will prove useful for other interactive mesh-editing applications, especially those targeting Java-enabled web platforms.

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 normal axes to constrain free-form vertex editing. We also implemented dual cameras to help reveal features that are occluded during the editing process due to the symmetry constraints of the tile. An interactively scalable display of the tile's nearest neighbors shows how adjacent tiles fit together. Finally, we have created a widget for specifying an affine transform that enables high-level 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 constraints 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.

6. REFERENCES

[1] Anglada, M. V.: An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations. Comput. & Graphics, 21 (2), 1997, 215–223.

[2] Conway, J. H.; Burgiel, H.; Goodman-Strauss, C.: The Symmetries of Things, A. K. Peters, Wellesley, MA, 2008.

[3] De Berg, M.; van Kreveld, M.; Overmars, M; Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, Springer, New York, 1998.

[4] Grünbaum, B.; Shephard, G. C.: Tilings and Patterns, W. H. Freeman and Co., New York, 1986.

[5] Fitzmaurice, G.: Matejka, J.; Mordatch, I.; Khan, A.; Kurtenbach, G.: Safe 3D Navigation, Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, 2008, 7–15.

[6] Joice, D. E.: Hyperbolic Tessellations, http://aleph0.clarku.edu/~djoyce/poincare/PoincareApplet.html.

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

[8] Lawson, C. L.: Software for C^1 surface interpolation, Mathematical Software III (Ed. J. Rice), Academic Press, New York, 1977.

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

[10] Schattschneider, D.: Computer Software for Tiling, http://www.geom.uiuc.edu/software/tilings/TilingSoftware.html.

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

[12] Séquin, C. H.: Patterns on the Klein Quartic, Bridges Conference, London, August 4-9, 2006, 245–254.

[13] Séquin, C. H.: Intricate Isohedral Tilings of 3D Euclidean Space, Bridges Conference, Leeuwarden, The Netherlands, July 24-28, 2008, 139–148.

[14] Shewchuk, J. R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, Selected papers from the Workshop on Applied Computational Geormetry: Towards Geometric Engineering, May 27-28, 1996, 203–222.

[15] Shewchuk, J. R.: Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery, Eleventh International Meshing Roundtable, Sandia National Laboratories, 2002, 193–204.

[16] Shon, Y.: Escher Tiling on the Torus, CS 285 course project, U.C. Berkeley, May 2002.

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

[18] Yen, J.; Séquin, C. H.: Escher Sphere Construction Kit, Interactive 3D Graphics Symposium, Research Triangle Park, NC, March 19-21, 2001, 95–98.

[19] Weeks, J.: KaleidoTile, http://www.geometrygames.org/KaleidoTile.