Feature-Based Cellular Texturing for Architectural Models

Justin Legakis Julie Dorsey
Laboratory for Computer Science
Massachusetts Institute of Technology

Steven Gortler
Division of Engineering and Applied Sciences
Harvard University

Abstract

Cellular patterns are all around us, in masonry, tiling, shingles, and many other materials. Such patterns, especially in architectural settings, are influenced by geometric features of the underlying shape. Bricks turn corners, stones frame windows and doorways, and patterns on disconnected portions of a building align to achieve a particular aesthetic goal. We present a strategy for feature-based cellular texturing, where the resulting texture is derived from both patterns of cells and the geometry to which they are applied. As part of this strategy, we perform texturing operations on features in a well-defined order that simplifies the interdependence between cells of adjacent patterns. *Occupancy maps* are used to indicate which regions of a feature are already occupied by cells of its neighbors, and which regions remain to be textured. We also introduce the notion of a *pattern generator* — the cellular texturing analogy of a shader used in local illumination — and show how several can be used together to build complex textures. We present results obtained with an implementation of this strategy and discuss details of some example pattern generators.

CR Categories: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling — geometric algorithms, languages, and systems; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism — color, shading, shadowing, and texture.

Additional Keywords: cellular texturing, computer-aided design, procedural modeling, texturing.

1 Introduction

Scene and object modeling of complicated structures requires substantial human time and effort. Hence, it is important to automate the process as much as possible. One way to achieve this goal is to use procedural methods to help generate complex and repetitive detail. The goal of such methods is to relieve designers of tedious and difficult manual modeling tasks, and to give them more high-level control.

For many models, especially in architectural settings, there is a clear distinction between the basic shape (walls, doors, windows, and arches), and embellishing patterns (bricks, stones, tiles, and shingles). In these cases it can be useful to decompose the design process into these two components. The basic shape can be described with traditional geometric modeling tools, while the placement of embellishing and patterned arrangements of geometric entities, or *cells*, can be encapsulated in an abstraction we refer to as a *cellular texture*. The placement and pattern of these cells can be governed with a procedural cellular texture pattern generator. The complex operations of generating the cellular texture are coalesced into a handful of parameters that drive the process. Ideally, there should be enough parameters that the designer does not give up too much power, but not so many that they become unmanageable.

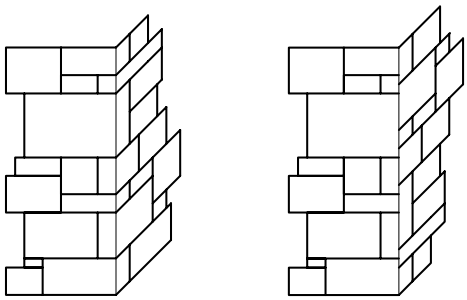


Figure 1: Texturing with 3D cells.

A major limitation of many previous approaches to "embellishment generation" is that the resulting patterns are strictly 2D, having little to do with the 3D surface upon which the pattern is applied. More specifically, the problem of mapping a 2D pattern onto a surface is commonly cast as the problem of associating regions on the surface with corresponding regions of the pattern. If these regions are not aligned carefully, seams may be visible at their boundaries. Consequently, the user is left with the difficult task of trying to generate a set of textures without objectionable discontinuities. (See Figure 1, above.)

Cellular texturing techniques need to recognize that cells are three-dimensional objects, and that their size and shape affect the pattern they create. This is apparent on corners, where multiple sides of a cell are visible. In addition, many cellular patterns are not flush with the surface they create — cells extend outwards from the mortar joints between them. Shadows cast by cells onto each other and onto themselves due to their 3D geometry can have a great effect on the overall appearance of a pattern.

In our framework, the geometry of the base mesh functions as a scaffolding for cells. The base mesh and cellular patterns are therefore separate, allowing the designer to consider and edit each individually. Patterns should be flexible enough to adapt to changes in the model, or even be applied to a model entirely different than the one for which they were designed. The design space can be explored both by changing the original model and by adjusting the parameters of the patterns, with the computer creating the result of new combinations.

1.1 Contributions

The main contribution of this paper is a strategy for applying cellular textures to a base mesh. We treat this as a three-dimensional problem, considering both the geometry of the model and of the

ACM SIGGRAPH 2001, 12-17 August 2001, Los Angeles, CA, USA

cells we create. With our strategy, we show how to create patterns of cells that adapt to features of the model — aligning to edges, wrapping around corners, and responding to annotations supplied by the designer. We discuss geometric requirements of the input base mesh, the organization of cellular patterns into a tree of pattern generators, and the application of pattern generators to the base mesh. We avoid problems at the boundary between adjacent patterns by placing the most constrained cells first, starting with the corners, then texturing the edges, and finishing with the faces.

While we describe the details of how our example pattern generators work, this paper is not about how to create specific patterns. There is a wide array of literature on this subject[1, 5, 6, 9, 11, 14, 21]. We do discuss some useful techniques for creating pattern generators, with the hope of encouraging others to envision what types of patterns they could create.

For simplicity, we require that our input models have nice parameterizations and well-defined structure. We are not claiming that with our strategy one can tile a one-inch cube with two-foot stones, texture a sphere with a regular pattern of square cells, or perform other such impossible or nonsensical tasks. It is easy for one to create a model and a pattern that are simply incompatible, and we leave the burden of ensuring a sensible combination to the human.

1.2 Overview

The remainder of this paper is organized as follows. We discuss related pattern and texture generation work in Section 2. In Section 3, we discuss the input model, or "base mesh" to which cellular textures are applied, and in Section 4 we describe the generation of cellular textures. In Section 5, we show the details of our example pattern generators, along with our results. In Section 6 we conclude and suggest areas for future research.

2 Related Work

There are several algorithms for creating 2D cellular patterns. Yessios described a prototype computer drafting system for common materials, which included a number of algorithms for generating cellular patterns[21]. His algorithms first place the cells that require special treatment, those on the perimeter of regions. We build on this idea, extending it to 3D cells and geometry. Miyata described an algorithm for automatically generating stone wall patterns[14]. He generates displacement data to create natural-looking cells, using it to render his patterns with bump mapping. These techniques produce realistic patterns on the faces of a model, but do not consider how the 3D shape of cells affects the pattern, nor do they consider the 3D geometry of the model to which they are applied. When rendered on a 3D model, seams are apparent in the textures at face boundaries.

Wong *et al.* addressed the problem of creating ornamental patterns to fill arbitrary shapes[19]. The patterns adapt to the shape of the region to be filled, but as with the techniques above, this is strictly a 2D technique.

Fleischer *et al.* used particle systems to simulate cells constrained to lie on a 3D surface[10], governed by a set of "cell programs." The cells react to properties of the model and environmental factors. This technique is well suited for generating cellular patterns that are biological in nature, however the authors warn that cell programs can be difficult to write, and the effects of modifications can be hard to predict. As cell programs control local interactions among cells, they are not practical for achieving a particular desired global structure.

Solid texturing techniques[15] elegantly solve the problem of texturing geometry seamlessly, even across arbitrarily complex face boundaries. Worley introduced a cellular texture basis function[20], a special case of which generates cells as the voronoi diagram of a set of seed points, with remarkable results. Solid textures do tile complex geometry seamlessly, yet they typically do not respond to the geometry itself. Instead, the resulting textured model appears to be carved out of a solid block of texture.

The texture synthesis techniques of Heeger and de Bonet generate tilable texture visually similar to a source image[12, 8]. Praun *et al.* took a different approach, covering an object with overlapping cutouts from a source image[16], relying on the viewer's inability to see the seams between textures. Praun does adapt the texture to the base object, aligning the textures to a vector field defined over the model. However, in general texture synthesis techniques are not responsive to the features of the underlying model. These methods work well for natural materials and patterns, but fail on the types of structured cellular patterns that this paper addresses. Because they operate on images, texture synthesis techniques are also not suitable for generating patterns composed of 3D geometric entities.

A different body of related work is the field of parametric modeling[13]. Models are defined in terms of the steps taken in their construction, together with a set of geometric constraints. The details of the steps and constraints can be edited and reevaluated, and the designer is really generating a parameterized family or class of objects. Cellular texturing is a form of parametric modeling. The resulting cellular texture is a parametric model, parameterized by both the base mesh and the cellular patterns to be applied. An important contribution of parametric modeling that we borrow is the idea of "features." Rather than working with just pure geometry, features of the object are annotated to capture some higher-level meaning of their purpose in the model.

Amburn *et al.* deal with the problem of resolving conflicts while procedurally generating geometry[2]. In their work, elements of the scene communicate with, and respond to, each other while working together to satisfy mutual constraints. We take a different approach to the problem, but our goals are similar. Cook introduced the idea of shade trees[7], building complex shaders out of smaller units. We use this idea for defining cellular patterns using a tree of simpler pattern generators.

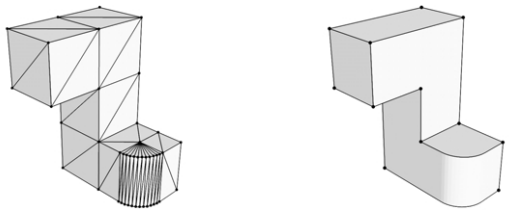
3 Base Mesh

In this section we discuss the base mesh, the geometric input to the system. There are three important aspects of the base mesh. First, the mesh is stored with a two-level geometric hierarchy. This addresses the dual role of the input model — to specify the geometry of the surfaces to which the cellular textures are applied, and to supply annotated features to influence and drive the patterns. Second, edges and faces of the mesh are parameterized, providing the pattern generators with a mapping from parameter space to world space and a projection from world space to parameter space. Third, the base mesh stores *occupancy maps* for edges and faces, to record the portions of features that get covered as cells are placed on the model.

While a hierarchical representation of the model, annotation of features, and parameterization are important parts of our cellular texturing strategy, the actual preparation of the input model is not the focus of this paper. To create, annotate, and parameterize the input models in our examples, we used a combination of automatic and interactive tools. Anderson *et al.* discuss an algorithm for automatically detecting architectural features[3].

3.1 Geometry and Features

In this context, the term "geometry" connotes two things (see Figure 2). The low-level interpretation of the geometry is the actual



Low-level mesh (geometry) High-level mesh (features)

Figure 2: The low-level shape of the object is defined by its "child" mesh, consisting of *vertices*, *polygon edges*, and *polygons*, while the high-level features are defined by its "parent" mesh, consisting of *corners*, *edges*, and *faces*.

shape of the object, as defined by the model's polygons. We expect the form created by a cellular texture to correspond to this shape. The high-level interpretation is the identification of the geometric features of the model: faces (such as walls and roofs), edges (boundaries between faces), and corners (intersections of edges). A cellular texture is defined in terms of how it applies cells to these three types of features.

To capture additional information, we annotate features with labels, indicating which features are part of windows, doors, arches, or other structures. These labels are stored as strings, attached to features.

We use the terms *vertices*, *polygon edges*, and *polygons* to refer to the elements of the low-level mesh that define the shape of the object, to which cells are aligned. We use the terms *corners*, *edges*, and *faces* to refer to the elements of the high-level mesh that are the targets of cellular texturing operations. The mesh representation must provide access to both levels. We use a hierarchical version of a winged-edge adjacency data structure [4, 18], with complete mesh structures maintained for both levels. Elements of the high-level, or "parent" structure correspond to one or more elements of the low-level, or "child" structure. Parent faces correspond to a collection of connected child polygons, and parent edges correspond to a chain of child polygon edges. Parent corners correspond to a single child vertex, while many child vertices (such as those in the middle of parent edges and faces) have no corresponding parent corner.

It is important to note that parent faces may be concave and may also contain holes and multiple edge loops. For example, consider a wall with a window in the middle.

One can envision segmenting a model into more complex features that span multiple geometric elements of the model. We stick with corners, edges, and faces to show how much you can do with just these. In our experience, it has been quite natural to decompose patterns into parts corresponding to corners, edges, and faces.

3.2 Parameterization

We require all edges and faces of the base mesh to have closed-form parameterizations. The parameterizations must support two operations: mapping points in parameter space to a coordinate frame in world space, and projecting points in world space to points in parameter space. Pattern generators use the mapping to place cells they have created in parameter space onto the model. The projection from world space to parameter space is used when filling occupancy maps (Section 3.3) with cells from adjacent features.

For the examples used in this paper, we have implemented parameterizations of edges that are composed of linear and circular segments, faces that are made up of combinations of flat and cylindrical regions, and faces that are sectors of disks. Corners are not parameterized. Figure 7 shows examples of geometry with all of these types of parameterizations.

3.3 Occupancy Maps

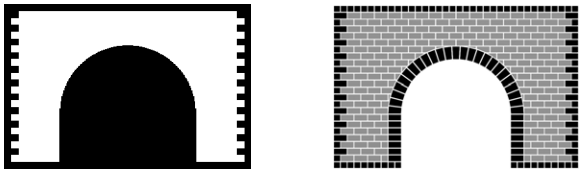
Cells are stored in the base mesh with the feature for which they were created. As cells are generated, adjacent features must know that the space has been occupied. This information is encapsulated in an *occupancy map*, a bit mask that tells a pattern generator which parts of the feature's parameter space have already been occupied by cells of its neighbors, and which areas it is responsible for filling. Occupancy maps are kept for all edges and faces. Corners do not need occupancy maps, because they are always textured first.

Occupancy maps for faces are 2-dimensional, and are initialized to fully "occupied." The polygons of the low-level mesh corresponding to a face are then rasterized into the face's occupancy map, and these areas are set to "empty." This prevents a pattern generator from filling cells in areas of the parameter space that are not actually part of the face, such as the inside of windows, or the underside of arches. When adjacent corners and edges are textured, their cells are projected onto the face and rasterized into the map, setting these areas to "occupied." (See Figure 3.)

Occupancy maps for edges are 1-dimensional, and are initialized to "empty." The geometry of an edge need not be projected onto the occupancy map as is done with a face, because edges never have holes. As the two adjacent corners are textured, their cells are projected onto the edge, setting these intervals to "occupied" in the map.

The space that cells claim in occupancy maps is often not the same as their actual geometry. Most patterns create cells that fill regions completely, and then shrink them to leave gaps between adjacent cells for mortar (see Figure 6). The space-filling shapes are what cells claim in occupancy maps; their actual shapes are used for rendering.

We implement occupancy maps for edges as an array of boolean values. For faces, we implement occupancy maps with a quad tree, both for speed in rasterizing and to allow for higher resolution. The required resolution depends on the size and complexity of the cells. By using a single-valued occupancy map, we can avoid intersections between cells along the curve of an edge or the surface of a face. This is sufficient for many patterns, including all the examples shown in this paper. Representations such as octrees or layered depth images[17] that store volumetric information could be used to allow for patterns in which cells fit together in an overlapping fashion.

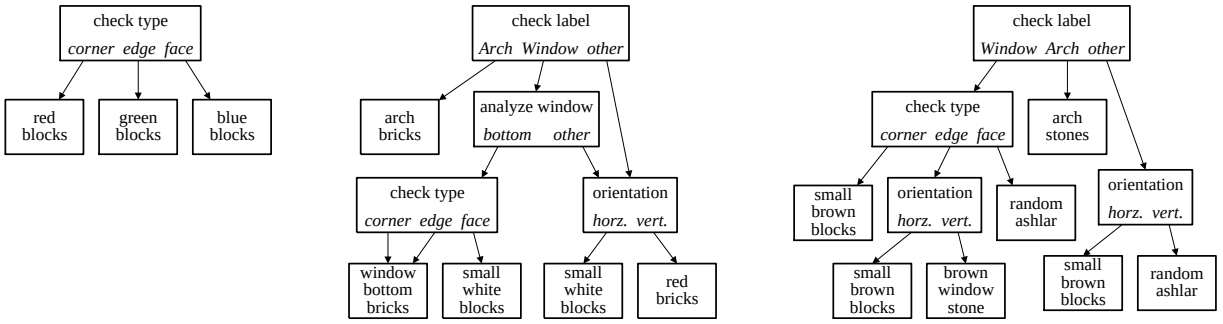


(a) (b)

Figure 3: The *occupancy map* for a face indicates the areas already occupied by cells of adjacent features (a). A pattern generator creates cells to fill the unoccupied region, fitting cells in the available space, and/or clipping them against the occupancy map (b).

4 Pattern Generators

This section presents the details of our cellular texturing framework. Much in the way that shade trees define shaders out of smaller, simpler, and more reusable parts, we discuss the building of cellular textures out of smaller units, which we term *pattern generators*.



(a) RGB Blocks (b) Bricks (c) Random Ashlar

Figure 4: Three examples of pattern generator trees.

4.1 Tree of Pattern Generators

Pattern generators implement the basic building blocks of patterns. Arranged in a tree, a collection of pattern generators applies a cellular texture to the base mesh, operating on one feature at a time. Features of the mesh start at the root of the tree, and are passed down until they are fully textured.

Each individual pattern generator can place cells on the feature, and/or pass the feature on to one of its children for further processing. Pattern generators may pass features to different children based on criteria such as their label, their type (corner, edge, or face), or the result of geometric analysis. For example, a pattern generator might look at a feature's label, and pass "Window" features to one subtree, and the rest to another subtree. Within the subtree for "Window" features, a pattern generator might pass vertical edges to one subtree, and horizontal edges to another.

Pattern generators that create cells can be general and simple, such as nodes that create brick-shaped cells, or specific, such as nodes that create stones for arches of a particular style. Simple nodes are typically more reusable, useful as elements of many different patterns.

Figure 4 shows three examples of pattern generator trees. The simple "RGB Blocks" tree visualizes the cells that are placed on the three different types of features: corners, edges, and faces. The result of applying this pattern to three different base meshes can be seen in the second row of Figure 7. The "Bricks" tree creates a pattern of bricks, with special treatment for features labeled as arches and performing simple geometric analysis to find the bottom of windows. This pattern can be seen in the third row of Figure 7. The "Random Ashlar" tree creates a randomized stone pattern, also with special treatment for features labeled as windows or arches, and the result of applying this pattern can be seen in the bottom row of Figure 7. Details of the individual pattern generators in these examples are given in Section 5.

4.2 Ordering of Texturing Operations

The key to avoiding conflicts and intersections when placing cells on adjacent features is to perform cellular texturing operations in the correct order. Consider the placement of cells on the edge of a stone building. These cells are really part of two patterns, one on each wall. If cells are placed on the walls first, working towards the edge from either side, it may be difficult or impossible to create the final cells along the edge. Likewise, if cells are placed along the multiple edges, working towards a corner, it may be tricky to place the final cells in the corner. There are more constraints on an edges' cells than on a faces' cells, and there are more constraints on a corners' cells than on an edges' cells.

This suggests that it is easier to place cells on the edges before the faces, and on the corners before the edges. The terms *corner*, *edge*, and *face* should be thought of according to these relationships, slightly more general than their usual geometric interpretations:

corner: region of interaction between two or more *edges*
edge: region of interaction between two *faces*

Once the corners have been textured, we can proceed with the edges, without worrying about how the edges interact with each other. The more constrained cells that are part of patterns on multiple edges have already been placed. Likewise, once the edges have been textured, we can proceed with the faces, without worrying about how the faces interact with each other. The difficult cells that are part of the patterns on multiple faces have already been placed on the edges.

What this means to pattern generators is that the patterns they create must be partitioned into 3 parts: cells that go on corners, cells that go on edges, and cells that go on faces. Figure 5 shows the simple example of placing blocks on a cube, after the corners have been textured, then after edges have been textured, and finally after faces have been textured.

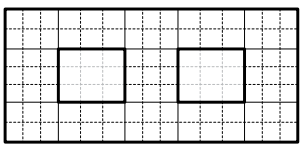
(a) (b) (c)

Figure 5: Applying cells first to all corners (a), then to all edges (b), and then to all faces (c).

5 Examples and Results

5.1 Tools

While experimenting with writing pattern generators, we developed some tools that we found useful for several different patterns. One tool we reused often is a function to impose a grid structure on a face, potentially stretching or shrinking the grid in areas so that



it lines up with with the shape of the face. We make two sorted lists with the s and t coordinates of every corner in the face (solid lines, above). We then divide each interval into segments as close as possible to the desired grid size (dashed lines, above). Our simple "RGB Blocks" pattern generator simply uses the grids of each face, testing each potential block against the occupancy map, and creates cells for those that pass.

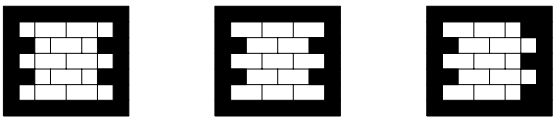
Other tools we found useful for creating pattern generators are functions for clipping the polygonal footprint of a cell against the occupancy map, creating cell geometry from the intersection of a set of planes, and shrinking a cell's full claimed volume to create its actual geometry.

5.2 Patterns

In this section, we describe how several of our individual pattern generators work.

Bricks: To create a pattern of bricks, this pattern generator first overlays a grid on the faces, and marks each grid space that is unoccupied. It then connects pairs of blocks to form bricks. Single bricks are placed on the corners. Bricks in alternating directions are placed on the edges, filling the empty interval(s) in the occupancy map. To fill faces, there are two options for combining bricks (see figure, below). We want to minimize or eliminate the number of half bricks. Since both fill the same area, picking the option that creates the *least* number of bricks yields the better pattern.

If the directions for edge bricks are chosen poorly, it may not be possible to create a nice pattern on the faces. We flip the directions of edge bricks interactively, storing an extra bit with each corner and edge for this purpose.



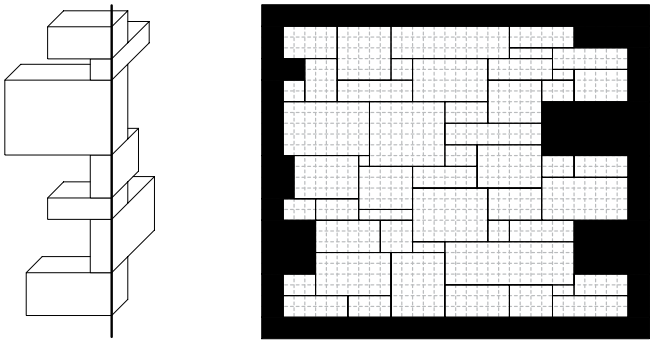
Bad Good ?

Window Bottom Bricks: This pattern generator operates only on corners and edges, creating a row of header bricks along the bottom of a window. For a corner, it creates a simple block and then extrudes it in the opposite direction of its adjacent concave edge. For an edge, it also creates simple blocks, and extrudes them in the direction normal to the edge's adjacent vertical face. It would be an error to construct a tree that passes faces to this pattern generator.

Arch Stones: This pattern generator creates multiple rows of stones for edges. The length and extrusion for each row of cells are parameters. As demonstrated by this pattern, cells do not necessarily touch the feature for which they are created.

Arch Bricks: To create bricks for the edge of an arch, this pattern generator looks at the tangent vector along the edge. For vertical portions of the edge, it behaves identically to the "Bricks" pattern generator, placing bricks in alternating directions. However, once the direction deviates from vertical, bricks are placed along the outer face, determined by choosing the direction vector with the greater y component.

Random Ashlar: Our random ashlar pattern also makes use of the grid tool described above, using a small grid size (dashed lines, below). This pattern places a single stone on corners. For edges, it creates stones in alternating directions as with bricks, but using random heights and lengths. To tile faces, it creates a small rectangle for each unoccupied space in the grid. It then picks rectangles at random, and attempts to merge them with their neighbors, maintaining their rectangular shape and obeying maximum size parameters of the pattern generator. This process is repeated until no more rectangles can be merged (solid lines, below).



5.3 Geometric Analysis

To build interesting patterns, some pattern generators must make a decision about a feature, and pass it onto one of its children for further processing. The most straightforward criteria to use is the type of feature (corner, edge, or face) or the label attached to the feature. However, pattern generators can also make decisions based on the geometry of the model.

Orientation: This pattern generator passes each edge and face to either a "Horizontal" or "Vertical" subtree. Edges are passed based on the vector from one end to another: if the largest component is in the y or $-y$ direction, the edge is considered vertical. Faces are passed based on their normal vector: if the largest component is in the y or $-y$ direction, the face is considered horizontal. Note, these criteria may not always be appropriate for curved edges or faces.

Corners are neither horizontal nor vertical. They are passed to one subtree or the other based on a boolean parameter of the pattern generator.

Analyze Window: Given a feature labeled as a window, this pattern generator determines if the feature is part of the top or bottom. Corners with an outgoing edge in the y direction, edges that have a neighboring face with a normal vector in the y direction, and faces with a normal vector in the y direction are classified as on the bottom, and sent to one subtree. Features on the top are determined similarly. All other features (those on the sides of the windows) are sent to a third subtree.

5.4 Mortar

Many cellular textures are not complete without mortar joints in between the cells. Rather than create actual geometry to fill the space between cells, we shrink the base mesh and use it in our renderings for the mortar (see Figure 6.)

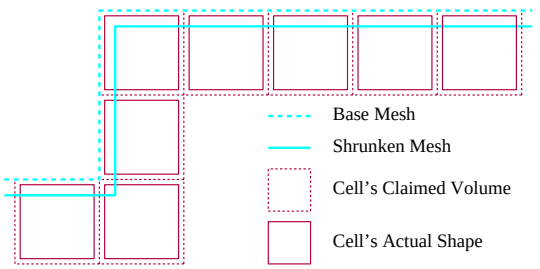


Figure 6: Cell geometry and mortar.

5.5 Results

Figure 7 shows ray traced images of three patterns applied to three different base meshes. The model in the first column has no labels, the model in the second column is annotated with "Window" labels, and the curved edges of the third model are labeled "Arch." Of the three pattern generator trees, the bricks and random ashlar patterns give special treatment to windows and arches. If a base mesh has no features with a particular label, the corresponding portion of the tree may not be utilized. The same base mesh was used in each column, and the same pattern generator tree was used in each row.

Figures 8a and 8b show two different brick patterns applied to a model of some steps and square columns. Intermediate results of applying the first pattern can be seen after creating cells for the corners (c), the edges (d), a decorative pattern on the faces of the columns (e), and the remaining space of all the faces (f).

These brick patterns rely heavily on geometric analysis to place bricks properly. Corners are classified by their valence, the number of convex and concave adjacent edges, and the direction (up, down, or horizontal) of a single concave or convex edge. Even so, in many cases there was a choice of two directions for bricks on corners and edges, and we made these choices interactively.

The first pattern creates two diagonal bricks on the top corners of the steps. At first glance, it may seem that bricks on the edges need to interact with each other, violating the assumptions made in Section 4.2. However, these patterns do fit nicely into our framework, by generating all three bricks as the cells for a corner (see Figure 8c).

The cement stones in the second pattern were placed on the corners and edges of the steps. When the meshes for these cells were created, the pattern generator marked polygons with roughness parameters. The meshes were then decimated and displaced with a turbulence function prior to rendering.

The decorative patterns on the columns are an example of a pattern generator placing some cells on a feature, and then passing the feature down the tree for further processing. Figure 8e shows the decorative pattern before the rest of the face was tiled. The size and position of the herringbone pattern was set carefully so that the regular brick pattern would nicely align.

The handrails were modeled separately by hand, and added to the scene after the cells were generated.

6 Summary and Conclusion

We have described a strategy for feature-based cellular texturing, applied to architectural models. We algorithmically generate cellular textures that are the result of both an underlying model and patterns of cells. The elements of the resulting patterns are full 3D entities that react to features of the model.

We demonstrate our technique with several patterns. However, this is just scratching the surface, and it opens up several interesting areas for future work. There are countless real-world patterns to implement and explore in our framework. We believe that a higher-level classification and specification of patterns would be of great benefit.

We have only been concerned with the outward appearance of our cellular textures. However, real buildings are not one stone deep. An interesting direction to take this work would be to generate textures with full internal structure. We would also like to create more detailed geometry for material (such as mortar) between the cells, as different joint toolings cause different shadows to be cast, affecting the appearance of the texture.

This framework could be applied to more general types of models, including non-manifolds. For example, it may be more natural to specify a stone wall by a single polygon, rather than a closed mesh. We would also like to work with more general curves and surfaces.

Currently, our models and patterns are designed independently, with the user responsible for ensuring a sensible combination. Another area of future research would be to build a constraint-solving layer on top of our framework. Such a system could help coordinate the design of geometry and assignment pattern parameters.

Acknowledgments

We would like to thank Barb Cutler for an endless supply of ideas, encouragement, and constructive criticism. Stephen Duck provided valuable information and suggestions and also modeled the handrails in Figure 8. This work was supported by NSF awards (CCR-9624172 and CCR-9988535), an NSF CISE Research Infrastructure award (EIA-9802220), and a gift from Pixar Animation Studios.

References

[1] ALLEN, E. *Fundamentals of Building Construction*. John Wiley and Sons, Inc., New York, 1999.

[2] AMBURN, P., GRANT, E., AND WHITTED, T. Managing geometric complexity with enhanced procedural models. In *Proceedings of SIGGRAPH 86*, pp. 189–195.

[3] ANDERSON, D., FRANKEL, J. L., MARKS, J., AGARWALA, A., BEARDSLEY, P., HODGINS, J. K., LEIGH, D., RYALL, K., SULLIVAN, E., AND YEDIDIA, J. S. Tangible interaction + graphical interpretation: A new approach to 3d modeling. In *Proceedings of SIGGRAPH 2000*, pp. 393–402.

[4] BAUMGART, B. G. A polyhedron representation for computer vision. In *Proc. AFIPS Natl. Comput. Conf.* (1975), vol. 44, pp. 589–596.

[5] BUEHL, O. B. *Tiles*. Clarkson Potter, New York, 1996.

[6] CHABAT, P., Ed. *Victorian Brick and Terra-Cotta Architecture in Full Color*. Dover Publications, Inc., New York, 1989.

[7] COOK, R. L. Shade trees. In *Proceedings of SIGGRAPH 84*, pp. 223–231.

[8] DE BONET, J. S. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of SIGGRAPH 97*, pp. 361–368.

[9] DOWSLAND, K. A., AND DOWSLAND, W. B. Packing problems. *European Journal of Operational Research 56* (1992), 2–14.

[10] FLEISCHER, K., LAIDLAW, D., CURRIN, B., AND BARR, A. Cellular texture generation. In *Proceedings of SIGGRAPH 95*, pp. 239–248.

[11] GRÜENBAUM, B., AND SHEPHARD, G. *Tilings and Patterns*. W. H. Freeman and Co., New York, 1986.

[12] HEEGER, D. J., AND BERGEN, J. R. Pyramid-based texture analysis/synthesis. In *Proceedings of SIGGRAPH 95*, pp. 229–238.

[13] HOFMANN, C. M., AND JOAN-ARINYO, R. Parametric modeling. *To be published as a chapter in the CADG Handbook*. Current text is available on author's web page at http://www.cs.purdue.edu/homes/cmh/MyHome.html.

[14] MIYATA, K. A method of generating stone wall patterns. In *Proceedings of SIGGRAPH 90*, pp. 387–394.

[15] PERLIN, K. An image synthesizer. In *Proceedings of SIGGRAPH 85*, pp. 287–296.

[16] PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. Lapped textures. In *Proceedings of SIGGRAPH 2000*, pp. 465–470.

[17] SHADE, J., GORTLER, S. J., WEI HE, L., AND SZELISKI, R. Layered depth images. In *Proceedings of SIGGRAPH 98*, pp. 231–242.

[18] WEILER, K. J. *Topological structures for geometric modeling*. Ph.d. thesis, Rensselaer Polytechnic Institute, Aug. 1986.

[19] WONG, M. T., ZONGKER, D. E., AND SALESIN, D. H. Computer-generated floral ornament. In *Proceedings of SIGGRAPH 98*, pp. 423–434.

[20] WORLEY, S. P. A cellular texturing basis function. In *Proceedings of SIGGRAPH 96*, pp. 291–294.

[21] YESSIOS, C. I. Computer drafting of stones, wood, plant and ground materials. In *Proceedings of SIGGRAPH 79*, pp. 190–198.

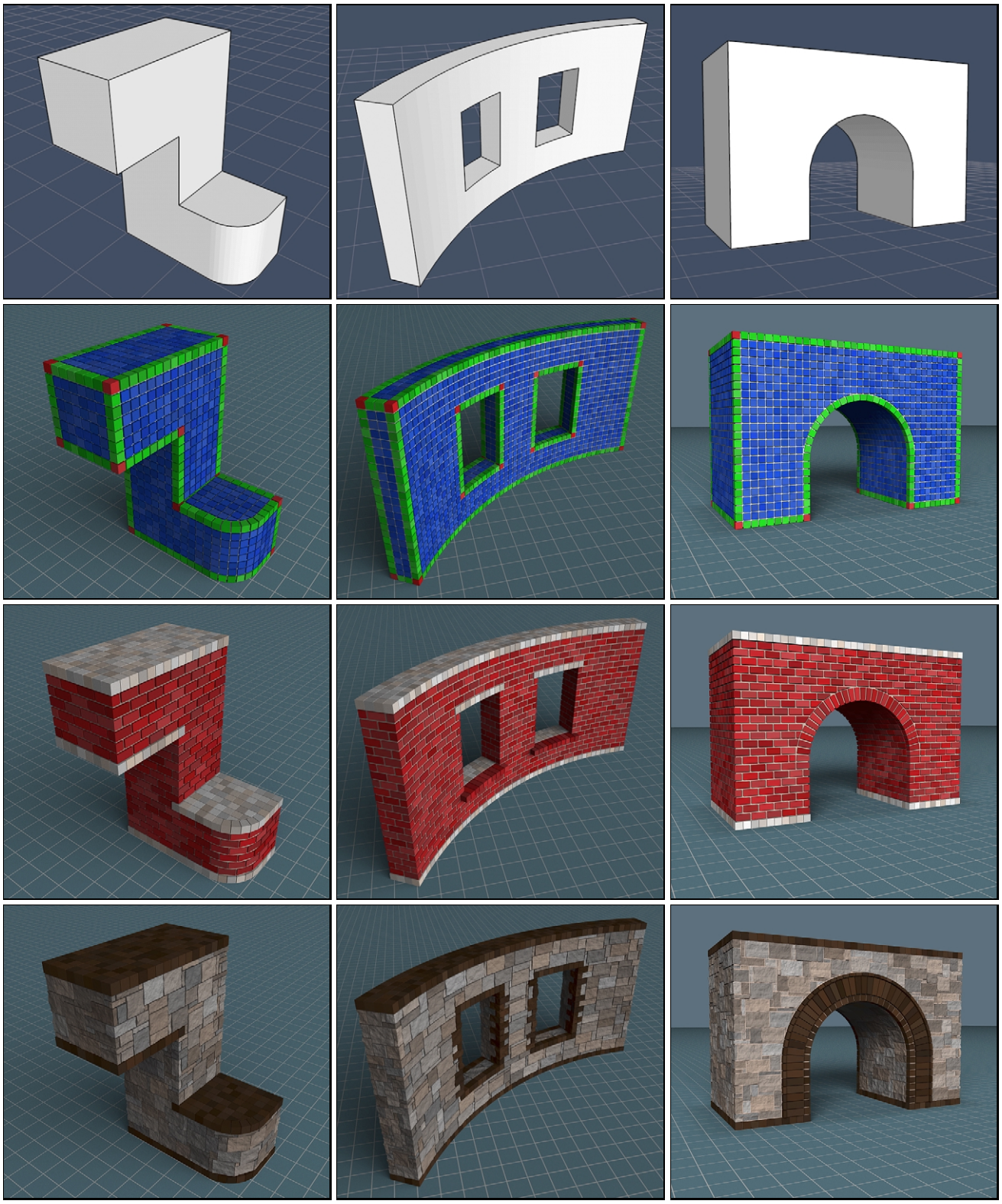
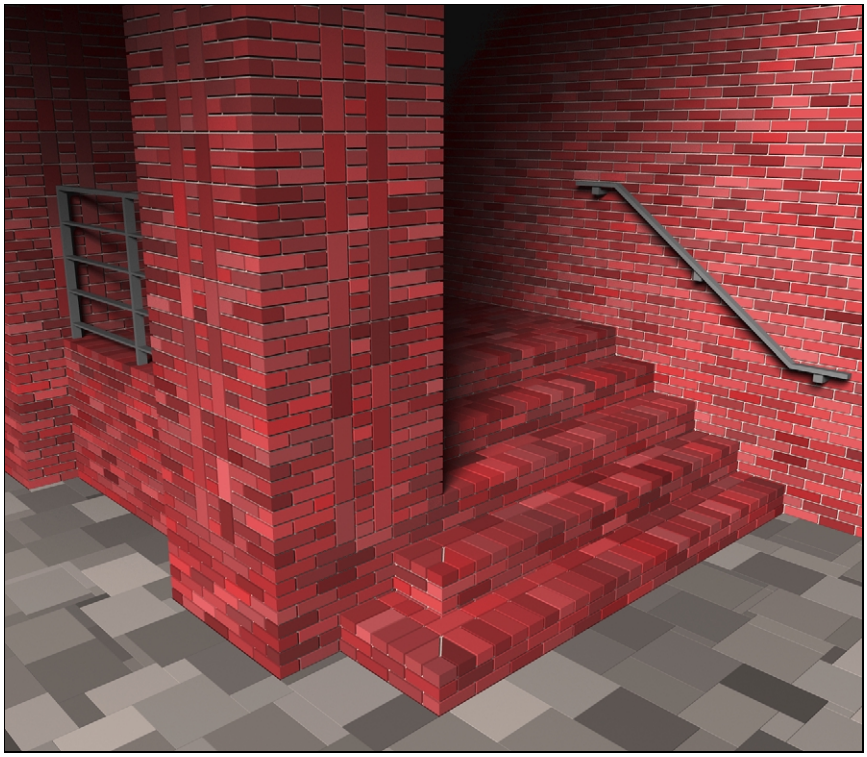
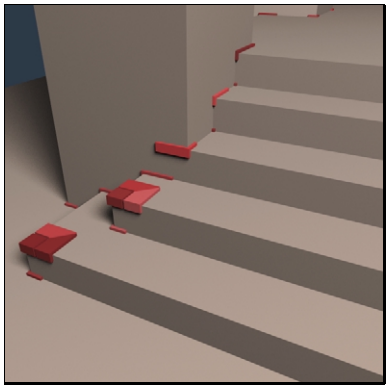


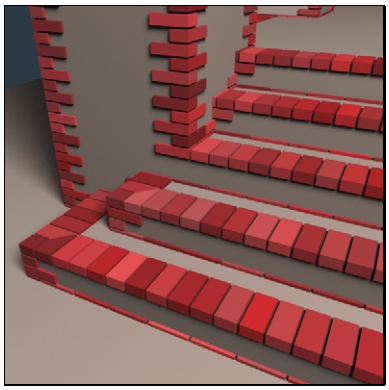
Figure 7: Three Models and Three Cellular Textures



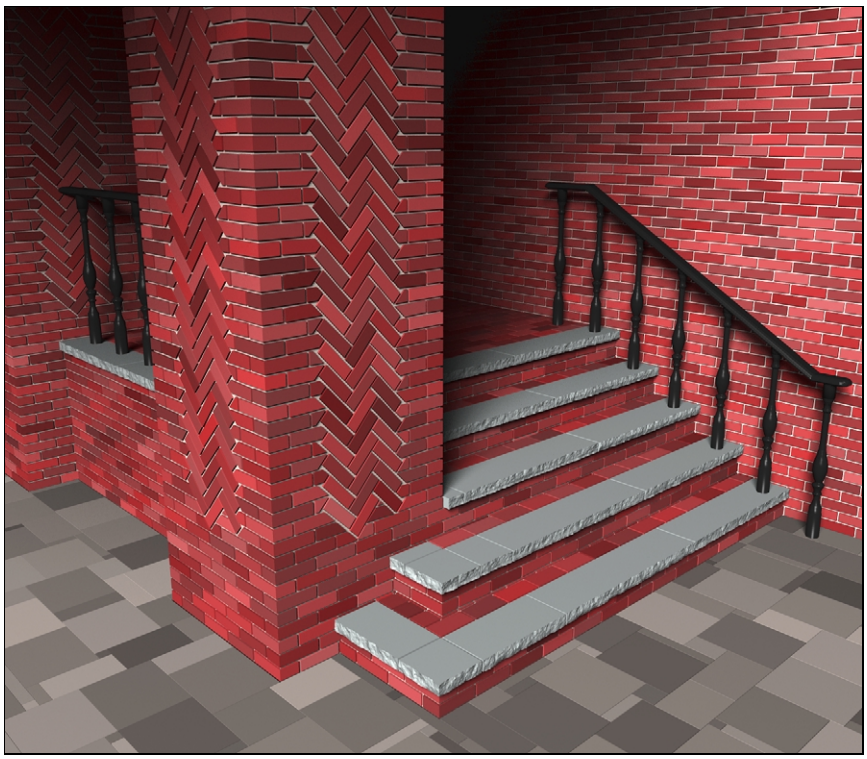
(a)



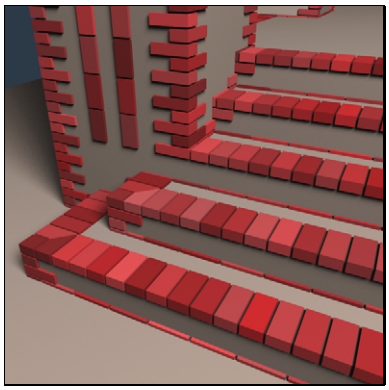
(c)



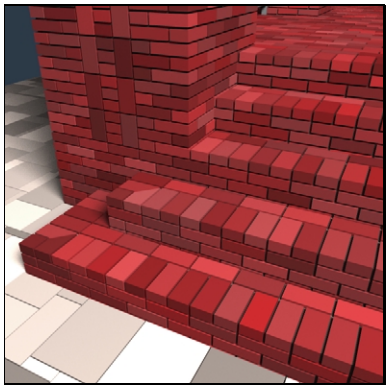
(d)



(b)



(e)



(f)

Figure 8: Brick Stairs