Modeling and Rendering Cellular Textures

by

Justin Legakis

B.S., Computer Science and Engineering
University of California at Davis (1996)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1998

© Massachusetts Institute of Technology 1998. All rights reserved.
Modeling and Rendering Cellular Textures

by

Justin Legakis

Submitted to the Department of
Electrical Engineering and Computer Science
on September 14, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Cellular patterns are all around us, in masonry, tiling, shingles, and many other materials. Their geometric nature gives rise to many interesting challenges. This thesis presents a collection of techniques for modeling and rendering cellular textures. First, an interactive framework for cellular texturing is presented, along with insight gained from the experience of implementing and using the system. A strategy for writing pattern generators to cover a model with cells is then presented. This leads to a cellular texturing language designed to facilitate rapid specification of cellular textures and experimentation with different combinations of patterns. Two compatible solutions to the problem of rendering complex scenes with detailed cellular textures are discussed: instancing, which reduces the amount of data needed to store a scene, and caching, which allows a scene to be rendered when it does not fit in memory. The structure of novel shading language is then described, and finally a rendering architecture is presented for rendering images using a network of workstations. Examples are shown of images created using this system, and a discussion of the results is presented.

Thesis Supervisor: Julie Dorsey
Title: Associate Professor
Acknowledgments

I would like to thank my advisor Julie Dorsey and the many other people that we worked with on cellular textures: Hans Køhling Pedersen, Sami Shalabi, Michael Monks, Maneesh Agrawala, and Chris Schoeneman.

I would also like to thank the other people who have influenced me most in my interest in computer graphics: Ken Joy, Bernd Hamann, Henry Moreton, and Seth Teller.

In addition, I would like to thank Michael Capps, Kavita Bala, and the rest of the MIT Computer Graphics Group for their support.
# Contents

1 Introduction
   1.1 Cellular Patterns ............................................. 13
   1.2 Modeling Challenges ............................................. 15
   1.3 Rendering Challenges ............................................. 18
   1.4 Overview of Thesis ............................................. 18

2 Previous Work ....................................................... 20
   2.1 Surface Tiling .................................................... 20
   2.2 Volumetric Textures ............................................. 21
   2.3 Particle Systems ................................................. 22
   2.4 Implicit Cells ..................................................... 23

3 Modeling Cellular Textures ........................................... 25
   3.1 Interactive Framework for Cellular Texturing ................. 25
      3.1.1 Software Architecture ....................................... 25
      3.1.2 Lessons Learned ............................................... 27
   3.2 Strategy for Filling a Model with Cells ....................... 27
      3.2.1 Corners, Edges, and Regions ............................... 28
      3.2.2 Model Analysis ............................................... 29
   3.3 Cellular Texturing Language ................................... 32
      3.3.1 Modules ....................................................... 32
      3.3.2 Types .......................................................... 34
      3.3.3 Cell Assembly Lines ......................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.4</td>
<td>Feature Identification</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Generating Cell Geometry</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Optical/Material Properties</td>
<td>39</td>
</tr>
</tbody>
</table>

4 Rendering Cellular Textures | 40

4.1 Managing Complex Geometry | 40
4.1.1 Instancing | 41
4.1.2 Caching | 42
4.2 Shading | 42
4.2.1 Overview of Shading | 44
4.2.2 “Shaders” | 45
4.2.3 “Materials” | 46
4.2.4 How Shaders and Materials Work Together | 46
4.3 Rendering System Architecture | 48
4.3.1 Render Workers | 48
4.3.2 Render Clients | 49
4.3.3 Render Server | 50

5 Results | 51

6 Conclusion | 53

A Color Plates | 55
# List of Figures

1-1 Examples of Cellular Textures: Bricks ........................................ 14
1-2 Examples of Cellular Textures: Stones and Tiles .......................... 15
1-3 Turning Corners ............................................................................ 16
1-4 Examples of Cellular Textures: Chimney Shafts ............................ 17

3-1 Software Architecture ................................................................. 26
3-2 Simple Strategy ............................................................................ 29
3-3 Edge Segment Types ................................................................... 30
3-4 Vertex Types ................................................................................ 31

4-1 Rendering Block Diagram ............................................................ 48
4-2 Render Client GUI ........................................................................ 49

A-1 Implicit Cells ................................................................................ 55
A-2 Building Tiled With Cells ............................................................. 56
A-3 Brick Cube .................................................................................... 57
A-4 Complex Base Mesh ..................................................................... 58
A-5 Cell Decimation ............................................................................ 59
A-6 Feature Identification ................................................................. 59
A-7 Optical Parameters ....................................................................... 60
A-8 Castle ........................................................................................... 61
List of Tables

3.1 Module Input Types ............................................. 35

4.1 Shading Variables ............................................. 44
Chapter 1

Introduction

1.1 Cellular Patterns

Cellular patterns are all around us, in masonry, tiling, shingles, and many other materials. Their complexities and imperfections give life and texture to real-world scenes[3]. Individual cells shape the surface appearance of such patterns with their color, orientation, and geometry, but they also provide the important underlying structure. Consequently, the spacing and characteristics of each cell guarantees that each pattern is unique. Cells are attractive not only in large expanses, but also in small accents — framing a doorway, outlining a walkway, or lining a niche.

Figure 1-1 shows some examples of buildings depicted with cellular textures. Much of the visual interest of these drawings lies in the patterns formed by the assemblies of bricks tiling the surface. The cellular textures are designed to adapt to the underlying geometry to which they are applied — they are aligned with important geometric features of the model, and different patterns interact to tile the model in an aesthetic fashion.

Figure 1-2 shows some more complex examples. These photographs are of the F.L. Ames Gate Lodge in North Easton, Massachusetts, designed by the architect H.H. Richardson. The roof of the building is covered with ceramic tiles that follow the shape and curvature of the surface. Tiles on flat portions of the roof are longer, and those on curved portions are shorter. The walls of the building are built out
Figure 1-1: Examples of Cellular Textures: Bricks (from [4])
of stones of various sizes and shapes, forming patterns that wrap seamlessly around corners and fill irregularly-shaped regions. Still more cellular patterns can be found on arches and window frames, where stones adapt naturally to the underlying geometric structure of the building.

1.2 Modeling Challenges

There are many interesting challenges associated with modeling cellular textures. First, cellular textures differ from other types of textures in that they are geometric in nature. Applying a cellular texture to a model amounts to tiling geometry with geometry. Second, the geometric aspect of cellular textures leads to challenges in
creating cells that turn corners, a situation in which the true 3D nature of the cells cannot be ignored. A third challenge is the need for a general methodology for working with cellular textures.

Traditionally, geometry and surface detail have largely been treated as separate entities in computer graphics. However, in the case of cellular textures, the border between geometry and texture is blurry. The geometry of the model and the placement of cells that cover it are closely tied. In Figure 1-4, for example, the bricks do not simply tile the surface of the chimneys, the bricks are the chimneys.

When attempting to place patterns seamlessly around corners, it is difficult to avoid cracking, distortion, self intersections, and other artifacts (see Figure 1-3). In

![Incorrect Correct](image)

Figure 1-3: Turning Corners

order to make a pattern naturally turn a corner or otherwise adapt to the geometry of the underlying object, intelligent decisions must be made about how to deal with boundary cases. Patterns placed on a model must be done so with regard to how they will interact with adjacent patterns on neighboring portions of the model. Patterns turning a corner cannot be considered independently on each region; cells lying on the boundary between two patterns are usually part of both patterns.

The interactive process of designing the surface detail of a geometric model is at present largely a batch process of trial and error, with long intermediate waiting periods. The lack of a general methodology for creating such images is unfortunate, since decorative patterns are visually striking and have great potential for enhancing the appearance of 3D models. Clearly, a more structured approach for modeling cellular textures is desirable.
Figure 1-4: Examples of Cellular Textures: Chimney Shafts (from [4])
1.3 Rendering Challenges

Once the modeling problems of cellular texturing have been overcome, the rendering of cellular textures presents a new host of challenges. Tiling a model with a cellular texture results in a substantial increase in scene complexity, both in terms of the number of cells and the geometric detail of each individual cell. For all but the simplest of scenes on the largest of computers, this geometry, if fully instantiated, will not fit in memory. Modifications must be made to a standard renderer to allow it to work with such large scenes and to take advantage of the repetitive nature of cellular textures.

Many cellular textures represent natural materials. To create a compelling image of such materials, simply rendering the geometry of the cells is not sufficient. The surface appearances of natural materials are highly complex, and require a renderer with sophisticated shading capabilities.

1.4 Overview of Thesis

In this thesis, the challenges of modeling and rendering cellular textures are addressed. An interactive framework for cellular texturing is presented, along with insight gained from the experience of implementing and using the system. Next, a strategy for writing pattern generators to cover a model with cells is presented. This leads to a cellular texturing language designed to facilitate rapid specification of cellular textures and experimentation with different combinations of patterns. Two compatible solutions to the problem of rendering complex scenes with detailed cellular textures are discussed: instancing, which reduces the amount of data needed to store a scene, and caching, which allows a scene to be rendered when it does not fit in memory. The structure of novel shading language is then described, and finally a rendering architecture is presented for rendering images using a network of workstations.

The remainder of this thesis is organized as follows. Chapter 2 discusses previous work in the areas of modeling and rendering cellular textures. Chapter 3 presents
work in the area of modeling cellular textures, and Chapter 4 presents solutions to
the challenge of rendering them. Results are discussed in Chapter 5, and conclusions
are presented in Chapter 6.
Chapter 2

Previous Work

Researchers have taken a wide variety of approaches to modeling and rendering cellular textures. These include: surface tiling algorithms that procedurally subdivide surfaces into cellular regions (Section 2.1), volumetric textures that tile a surface with texels that store the radiometric properties of the geometry they represent (Section 2.2), particle system techniques that place cells on a surface using biologically-motivated simulations (Section 2.3), and implicit cellular texturing in which cells are defined by a distribution of seed points and cell geometry is resolved per-sample at render-time (Section 2.4). In this chapter, each of these approaches is reviewed and their strengths and weaknesses are discussed.

2.1 Surface Tiling

For generating patterns, it is natural to consider a procedural approach. Yessios described a prototype computer drafting system for common materials such as stones, wood, plant, and ground materials, which included a number of algorithms for generating regular patterns[23]. Miyata also described an algorithm for automatically generating stone wall patterns[14]. In related work, tiling and packing problems have attracted the interest of mathematicians for centuries[5, 8].

A major limitation of these approaches is that the resulting patterns are ostensibly 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, visible discontinuities may occur at their boundaries. Consequently, the user is left with the tedious task of trying to generate textures that can be applied seamlessly across the surface.

2.2 Volumetric Textures

In contrast to 2D surface tiling methods, volumetric textures are true 3D entities. Inspired by volume densities[2, 11], volumetric textures represent objects as a 3D distribution of density and shading parameters. Introduced by Kajiya and Kay as a solution to the problem of rendering furry surfaces[10], “texels” are volumetric entities that seamlessly tile a surface to represent repetitive geometric patterns.

The classical model of a volume density function is that of a uniform distribution of spherical particles, each whose radius represents the local density. Shading inside the volume is performed solely as a function of the direction to the eye and the light source. Texels extend such volumes by replacing the spherical partials with oriented microsurfaces. Two properties are stored at each point in the volume: a density and a shading model. The shading model is comprised of a coordinate frame and a reflectance function.

Texels are mapped onto bilinear surfaces with user-specified vectors at their shared corners. This defines a trilinear deformation that maps each texel to its multiple positions on the surface, giving the user the ability to “comb” the orientation of the geometry. For rendering, rays are intersected with the trilinear extent of each texel. Each interval of intersection is mapped back into texel space, and then shading calculations are performed at uniform increments.

Volumetric textures have been extended by Neyret to allow for a more general reflectance function and for volumetric filtering[16]. The microsurfaces (and their corresponding coordinate frames) of Kajiya and Kay are replaced by ellipsoids, providing a compact form for normal repartitioning, and general enough to approximate
spherical, cylindrical, and planer surface elements. Texels are pre-filtered and rendered using an octree structure, analogous to the way 2D textures are pre-filtered and rendered with mip-maps[21]. In addition, Neyret has done work in animating texels[15] and generating texels from more traditional geometric representations.

Volumetric texturing works well for adding repetitive fine detail to a scene, where the radiometric properties are more important than the geometric properties. However, this method is not well suited for highly detailed foreground objects. Once the finest resolution of the texture is less than the sampling resolution of the image, it is necessary to default to an object's geometric representation.

### 2.3 Particle Systems

An alternative approach is to treat a "cell" as a generalization of a particle. Fleischer et al. used particle systems to simulate cells constrained to lie on an underlying geometric surface[6]. After the simulations have determined the position, orientation, size, and various shape and appearance parameters of each particle, the particles are converted to geometry. This geometry is then ray traced to produce a final rendered image.

Cells are modeled as particles with associated state information. Their behavior is controlled by "cell programs," which are evaluated by the simulation. These cell programs update the particles' state information based on their neighbors, the surface properties, and other environmental information. Cell programs can be superposed to build up complex behavior out of simpler components.

After the simulation is complete, the particles are converted to geometry. The estimated screen-space size of the cell can be used to control the geometric level of detail. The shape can depend upon the orientation and other state information of the particle. In addition, values computed by the simulation can be used to set shading parameters and variation.

Particle systems are well suited for generating cellular patterns that are biological in nature. The authors successfully wrote cell programs to create reptilian scale
textures, to cover a complex bear model with fur, and to tile a human head with thorns. However, the authors warn that cell programs can be difficult to write, and their effects can be hard to predict. As cell programs control local interactions between cells, they are not practical for achieving a particular desired global structure.

2.4 Implicit Cells

Implicit cells differ from the previous types of cells in that the geometry for each cell is never actually instantiated. Instead, the cells are resolved implicitly during rendering, on a per-sample basis. Worley introduced a cellular texture basis function[22] that takes as input a set of “seed points” that can either be generated procedurally or specified by the user. He defines the functions $F_n(P)$ as the distances from $P$ to the $n$th closest seed point. Linear combinations of $F_n$ can be used to create many interesting functions, all controlled by the placement or distribution of the seed points.

Worley intended for his cellular texture basis function to augment Perlin’s noise and turbulence functions[18] as tools to be used for creating solid textures. However, this original intention was overshadowed by the utility of one special form of his basis function: $F_2 - F_1$. This evaluates to zero for every point which is equidistant from two or more seed points, and to a positive value for all other points. Furthermore, this value is proportional to the distance of the nearest boundary. This implicitly defines the voronoi diagram of the seed points, dividing space into convex regions.

Implicit cells can be used for cellular texturing by specifying the center of each cell, and using these centers as the seed points. The renderer computes $F_2 - F_1$ at each sample and uses this value, along with the unique ID for the first and second closest seeds, in the shading computation for the surface. By altering the surface color based on the closest seed, and the surface normal based on the value of $F_2 - F_1$, convincing cellular textures can be realized (see Figure A-1).

This technique has the advantage of giving the user precise control over the global structure of the cellular texture, and relieves both the user and the system of the responsibility of generating geometry for each cell. In addition, a single seed point
per cell is an extremely efficient form in which to store a cellular texture. The main drawback to this method is that it is no more than a shading operation. Since no cell geometry is created or rendered, all cells are strictly two-dimensional and cannot extend beyond the original surface.
Chapter 3

Modeling Cellular Textures

This chapter addresses the problem of designing a collection of cellular textures, applying them to a model, generating the actual cells that make up the texture, and assigning optical and material parameters to the cells.

3.1 Interactive Framework for Cellular Texturing

In this section, a general approach to texturing models with an interesting range of cellular materials is presented. The goal of the approach is to provide an interactive setting in which the user can specify regions of the model, assign cellular textures, adjust the parameters of the textures, preview the textures directly on the 3D model, and produce a final rendering of their creation.

3.1.1 Software Architecture

The approach is organized as five software components (see Figure 3-1). As input, the system takes a polyhedral CAD model.

- **Region Editor**: Tools to interactively outline regions of the model.

- **Pattern and Material Editor**: Provides interaction tools for assigning a pattern and materials to a particular region and adjusting parameters (such as the sizing and spacing of cells).
- **Pattern Generators**: Generates cells to fill a region. Uses the region's pattern's parameters.

- **Sketch Visualization**: Draws the cellular textures using a rough pen and ink style rendering.

- **Renderer**: Takes as input the model geometry, cells, and surface material information. The renderer used to create the images in this thesis was a ray tracer, which was optimized to operate on large data sets.

![Software Architecture Diagram](image)

Figure 3-1: Software Architecture
3.1.2 Lessons Learned

Figure A-2 shows examples rendered with this system that illustrates many typical characteristics of cellular textures. For example, the irregularly shaped stones on the walls meet seamlessly around sharp corners but are clipped to the sides of the window frames. The roof tiles are arranged in parallel lines orthogonal to the y-axis, and subtle random variations add richness to the overall appearance. Also note that the individual stones and tiles are three-dimensional. The roof tiles are slanted slightly, and the stones extend out from the mortar joints between them.

Not surprisingly, the writing of the pattern generators was an exceedingly difficult part of implementing this system. It became evident that it was necessary to distinguish between different types of cells. Cells in the interior of a region are essentially defined by a 2D outline. They interact only with other cells in the same region, generated as part of the same pattern. However, cells lying on the interface between adjacent regions are really part of two or more separate patterns, and it is insufficient to represent these cells with 2D outlines.

Additionally, each pattern generator in this system is completely independent. Clearly, there would be some benefit to having reusable components that could be plugged together in different ways to create new and interesting pattern generators.

One last difficulty in working with the system was that each design was tied to a specific model. It was not possible to specify a set of pattern generators while working with one model, and then apply that design to other models — the design had to be recreated by hand each time. A clean separation of the design of a set of cellular textures from the model to which it is applied was clearly in order.

3.2 Strategy for Filling a Model with Cells

In this section, a strategy for implementing pattern generators is described. This strategy is motivated by the difficulty of filling the interface between two adjacent regions with cells, and the desire for a cellular texturing language that cleanly encapsulates the complete design of a cellular texture, independent of the model to which
it is applied. Such a language is described in Section 3.3.

### 3.2.1 Corners, Edges, and Regions

_Corners, edges, and regions_ are geometric features of a model. A region is a connected planar portion of the surface of the model. It may be of any shape, and may contain holes in its interior. For example, the front region of the castle in Figure A-8 is concave along its top edge, and has three holes for the windows. Adjacent non-planar regions meet at an edge. Edges are always incident to two regions, one on either side. Edges are also always incident to two corners, one on either side. A corner is where three or more edges and regions meet.

One main difficulty in filling a model with cells is that cells of adjacent regions must interact in a reasonable and desirable fashion. Cells that lie on an edge take up space in two regions, and are really part of two cellular textures. Cells that lie on corners are part of textures of three or more edges and regions.

One key idea of this strategy is to treat cells that lie on corners, edges, and regions differently. This implies three distinct types of pattern generators: those that place cells on corners of a model, those that place cells on edges of a model, and those that place cells in the regions of a model. The three different types of model features require three fundamentally different types of cells. Cells in the interior of a region generally have one “out-facing” side, and their shapes are characterized by a planar curve. Cells lying on an edge have two “out-facing” sides, and must be part of the textures of two adjacent regions. Corner cells can be viewed from three or more sides, so their 3D geometry is more evident. They are part of the textures of several regions.

Pattern generators can be applied to a model consistently if the system restricts the order of their application. A simple strategy would be to first fill all corners, then to fill all edges, and finally to fill all regions (see Figure 3-2). In this way, a corner pattern generator does not have to worry about fitting cell(s) adjacent to those of its incident edges and regions. An edge pattern generator only has to consider the cells of the two corners on either end, not the cells of the two regions on either side. A region pattern generator only needs to generate the cells that fill the interior of the
region. The border of the region has already been filled.

Figure 3-2: A simple strategy: cells are applied to a cubic model by (a) first filling the corners, (b) then filling the edges, (c) and then filling the regions.

This simple strategy can be made less restrictive. All that is required is that an edge cannot be filled after a region on either side, and that a corner cannot be filled after an incident edge or region. In this way, some features of the model such as doorways and windows can be filled (corners, edges, and regions) by one set of pattern generators, and the rest of the model can be filled by a different set.

3.2.2 Model Analysis

The base mesh of the model is stored using a winged-edge data structure[1, 20] and is assumed to be closed and manifold. It consists of three types of features: vertices, edge segments joining two vertices, and faces formed by loops of edge segments. Each feature of the model is analyzed and annotated with information according to its geometric properties. This information is available for use by the pattern generators.

First, edge segments are labeled according to the normal vectors of the faces on either side of the edge. Then, vertices are labeled according to the labels of all incident edges. Vertices are identified as “corners” if they are necessary to define the geometric structure of the model, as opposed to lying on an edge or in a region. Finally, collinear connected edge segments are identified as a single “edge,” and coplanar connected faces are identified and grouped into a “region.”
Edge Segment Types

Edge segments are labeled according to the normal vectors of their two incident faces (see Figure 3-3). If both normal vectors point away from the center of the opposite face, the edge is labeled “convex” (Figure 3-3a). If both normal vectors point in the same direction, the edge is labeled “flat” (Figure 3-3b). Flat edge segments separate coplanar faces, and are not necessary to define the geometric structure of the model. If both normal vectors point towards the center of the opposite face, the edge is labeled “concave” (Figure 3-3c). Assuming that the model is closed and manifold, these are the only three cases that can occur.

![Figure 3-3: Edge segments are labeled (a) convex, (b) flat, or (c) concave, based on the normal vectors of the faces on either side of the edge.](image)

Vertex Types

Vertices are labeled according to the labels of their incident edge segments (see Figure 3-4). Vertices that define the geometric structure of the model are labeled as “corners.” If all incident edge segments are convex, the vertex is labeled “corner-convex” (Figure 3-4a). If all incident edge segments are concave, the vertex is labeled “corner-concave” (Figure 3-4b). If there are both convex and concave edge segments incident to the vertex, it is labeled “corner-saddle” (Figure 3-4c).

Other vertices lie on edges or in regions, and do not define the geometric structure of the edge. These are labeled “on-edge” or “in-region.” If the vertex is incident to two collinear convex edge segments, and all other incident edge segments are flat, the vertex is labeled “on-edge-convex” (Figure 3-4d). Similarly, if the vertex is incident
to two collinear concave edge segments, and all other incident edge segments are flat, the vertex is labeled “on-edge-concave” (Figure 3-4e). Finally, if all incident edge segments are flat, the vertex is labeled “in-region-flat” (Figure 3-4f).

Only vertices identified as one of the three types of “corners” are filled by the cellular texturing system.

![Figure 3-4: Vertices are labeled (a) corner-convex, (b) corner-concave, (c) corner-saddle, (d) on-edge-convex, (e) on-edge-concave, or (f) in-region-flat, based on the types of all incident edges.](image)

**Joining Edge Segments into Edges**

After all edge segments and vertices are labeled, collinear connected edge segments are identified as a single “edge.” There are only two types of edges: convex and concave. Flat edge segments are not joined. A convex edge must consist of convex edge segments joined by on-edge-convex vertices, and a concave edge must consist of concave edge segments joined by on-edge-concave vertices. It is these *edges* that are filled by a pattern generator, not the *edge segments* of the base mesh.
Joining Faces into Regions

The last step of the model analysis is to identify connected coplanar faces of the mesh and group each into a single "region." For each face of the mesh not already assigned to a region, a search is performed over all incident edge segments labeled "flat." It is these regions that are filled by a pattern generator, not the faces of the base mesh.

3.3 Cellular Texturing Language

This section presents the design of a language to specify the application of cellular textures to a polygonal base mesh, based upon the strategy outlined in Section 3.2. The goals of the language are to allow a user to easily experiment with different combinations of cellular texturing operations, and to provide a general framework to which the user can add new components.

3.3.1 Modules

Cellular textures are specified in the language with a hierarchical collection of "modules." Simply stated, a module is something that performs an action on the model, either creating new cells or modifying existing cells. Modules can either be defined within the cellular texture program ("in-line modules") or written in C++ ("plug-in modules").

In-Line Modules

In-line modules are defined within the cellular texture program. An in-line module is simply a list of modules, each of either type. When an in-line module is applied, each module in its list is applied sequentially.

Plug-In Modules

Plug-in modules are C++ classes, derived from a base PlugInModule class. They are compiled as dynamic shared objects, and loaded and linked with the system upon
demand at run-time. Many useful plug-in modules are built in to the system, and users are free to write their own.

Plug-in modules implement the ApplyCellModule() method, which performs the action of the module. A plug-in module can accept any number of parameters to control its behavior. In general, the more flexibility a plug-in module offers through its parameters, the more varied applications a user will find for it.

Plug-in modules can accept other modules as parameters, and apply them as a part of its own application. For example, a module that fills windows with cells might take 2 modules as parameters, and apply one to the top and bottom of the window and the other to the sides.

An Example

The module \texttt{RedBricksWithStoneTrim} defined below was used to fill the cube in Figure A-3.

\begin{verbatim}
Define Module RedBricksWithStoneTrim {
 Module FillCorners {
 FillWith = StoneCorner
 }
 Module IdentifyEdges {
 FillVertWith = RedBrickEdge
 FillHorzWith = StoneEdge
 }
 Module FillRegions {
 FillWith = RebBrickRegion
 }
}
\end{verbatim}

\texttt{RedBricksWithStoneTrim} is an in-line module. It consists of a list of three other modules: \texttt{FillCorners}, \texttt{IdentifyEdges}, and \texttt{FillRegions}. In this example these three modules are all plug-in modules.

The \texttt{FillCorners} plug-in module takes one argument, \texttt{FillWith}, which is also a module. \texttt{FillCorners} searches the base mesh, and applies its argument to each empty corner. The \texttt{IdentifyEdges} plug-in module takes two modules as arguments, \texttt{FillVertWith} and \texttt{FillHorzWith}. \texttt{IdentifyEdges} searches the base mesh, and applies one
argument to all vertical edges, and a second argument to the rest. This is a simple
form of feature identification, discussed in Section 3.3.4. Similarly, the FillRegions
plug-in module takes one module as the parameter FillWith, searches the base mesh,
and applies the parameter to each empty region.

In this example, the parameter to FillCorners is StoneCorner, an in-line module
defined as follows:

```plaintext
Define Module StoneCorner {
    Module CornerBlocks {
        Size = 0.10
        Thickness = 0.05
    }
    Module Set {
        Shrink = 0.005
        Bevel = 0.02
        R = 0.75
        G = 0.75
        B = 0.75
    }
}
```

StoneCorner is defined as a list of two modules, CornerBlocks and Set. These are
both plug-in modules built in to the system. Given an empty corner of the base mesh,
CornerBlocks creates a cell of a specified size and thickness. Set sets some geometric
and optical parameters of the newly created cell. Section 3.3.3 discusses which cells
of the base mesh are modified by the Set module.

### 3.3.2 Types

Every module has a type based on two properties: what the module takes as input,
and what action the module performs. A plug-in module’s type is declared by its
author; an in-line module’s type is deduced from the modules within itself. Plug-in
modules that take other modules as parameters expect a module of a specific type,
and the language performs this type-checking. (For example, the FillCorners module
above requires a module that, given a corner of the base mesh, creates a cell to fill
that corner.)
Input Type

The first half of a module's type is its *input type*. The 9 different inputs types are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>One Corner</th>
<th>All Corners</th>
<th>Entire Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Edge</td>
<td>One Edge</td>
<td>All Edges</td>
<td>Special Feature</td>
</tr>
<tr>
<td>One Region</td>
<td>One Region</td>
<td>All Regions</td>
<td>One Cell</td>
</tr>
</tbody>
</table>

Table 3.1: Module Input Types

A module with an input type of *one corner* takes a single corner of the base mesh as input. Two examples are the *StoneCorner* module in Section 3.3.1 and the *CornerBlock* plug-in module built in to the system. Similarly, modules of input types *one edge* and *one region* operate on a single edge of the base mesh.

A module of input type *all corners* can potentially operate on every corner in the mesh. Similarly, modules of input types *all edges* and *all regions* potentially operate on every edge or region of the mesh. And example is the *IdentifyEdges* plug-in module that loops through every edge of the mesh, and applies one of two modules of type *one edge* to each, depending on each edge’s spatial orientation.

A module of input type *entire model* takes the whole base mesh as input. A program written in the cellular texturing language must define such a module to be used by the system to fill the input model with cells.

The input type *special feature* is provided for the user to write modules that identify special geometric features of the model, such as oriented edges, windows, or columns. (Feature identification is discussed in Section 3.3.4).

The last input type *one cell* is different. Rather than taking a portion of the base mesh as input, it takes an existing cell. The built in plug-in module *Set*, which sets geometric and optical properties of a cell, is an example of such a module.
Operation Type

The second half of a module’s type is its operation type. There are two operation types: create cells and modify cells. Modules of type create cells take unfilled portions of the base mesh and fill them with cells. Modules of type modify cells take filled portions of the base mesh or individual cells, and modify them. Note that there is only one illegal type for a module: input type one cell with operation type: create cells.

3.3.3 Cell Assembly Lines

In-line modules act as cell assembly lines. When an in-line module is applied, the modules in its list are applied in order. Its input and operation type are defined to be the input and operation type of the first module in its list. For example, the type of the in-line module StoneCorner in Section 3.3.1 is “input: one corner, operation: create cells,” because that is the type of CornerBlocks, the first module in its list.

All modules in the assembly line operate, in order, on the input to the in-line module, and on cells created within the module. All modules in the list must either be of the same type at the first, or of the type “input: one cell, operation: modify cells.” Modules that create cells add cells to the assembly line. Modules that modify cells are applied to all cells passing through that point of the assembly line.

3.3.4 Feature Identification

In all but the simplest cases, the user will not want to tile an entire model with the same set of modules. For example, it may make sense to apply one edge pattern generator to horizontal edges, and another to vertical edges, or to decorate window borders with their own set of pattern generators. Examples of these features are demonstrated in Figure A-6.

The input type special feature is provided for the user to write modules that identify special geometric features of the model, and apply a module to those features. Typically, the user will write two modules, one that searches the base mesh for a particular type of special feature, and a second that fills that type of special feature
with cells. The first module is of type *entire model*, and takes the second module of type *special feature* as a parameter.

For example, the module *BrickWallWithWindows* first identifies all windows with the plug-in module *FillWindows*. This module takes one parameter of type *special feature*. In this case, the parameter is the in-line module *MyWindowBlocks*. After the windows are filled, *BrickWallWithWindows* proceeds to fill the remaining empty corners, edges, and regions.

```plaintext
Define Module MyWindowBlocks {
    Module WindowBlocks {
        Size = 0.025
        Thickness = 0.04
        DivideBlocks = 2
    }
    Module Set {
        Shrink = 0.0025
        Bevel = 0.01
        R = 0.55
        G = 0.25
        B = 0.1
    }
}

Define Module BrickWallWithWindows {
    Module FillWindows {
        FillWith = MyWindowBlocks
    }
    Module FillCorners {
        FillWith = StoneCorner
    }
    Module IdentifyEdges {
        FillVertWith = RedBrickEdge
        FillHorzWith = StoneEdge
    }
    Module FillRegions {
        FillWith = RebBrickRegion
    }
}
```
3.4 Generating Cell Geometry

Modules that create cells output only a coarse representation of the cells' final shapes. With the exception of the cells in Figure A-2, all cells in every picture in this thesis were initially defined simply as bounding boxes. While the coarse representations of cells are sufficient to display the geometric patterns of the cellular textures, images of much greater visual interest can be created by converting the coarse cells into more naturally-shaped objects.

The creation and processing of cell geometry is performed using a winged-edge data structure[1, 20], similar to that used to store the base mesh of the model. In this system, after an initial coarse triangle mesh for each cell is created, the final cell geometry is generated by five mesh-processing steps: shrinking, beveling, decimation, smoothing, and displacement.

First, cell meshes are shrunk by an amount specified in the cellular texturing program. This creates space between the cells, which could be filled with mortar. Then, cell edges are beveled, again, by an amount specified in the cellular texturing program. This is an inexpensive way to add geometric interest to otherwise less detailed cells. See Figure A-5a for cells that have been shrunk and beveled.

Third, cell meshes are decimated. This serves two purposes: to create a nice mesh with roughly equilateral triangles, and to control the amount of detail added by the following two steps. The cell meshes are decimated by repeated splitting of the longest edge in the mesh until all edges are below a threshold length.

After cell meshes are decimated, they are subdivided using Loop subdivision[13], which smoothes out sharp corners of the mesh. Each iteration of the subdivision process creates a smoother mesh, and quadruples the number of triangles.

Finally, the cell meshes are displaced with a fractal turbulence function[18]. The amplitudes of the frequencies of the turbulence function can be adjusted to create various roughly-shaped cells. See Figure A-5b for cells that have been shrunk, beveled, decimated, smoothed, and displaced.
3.5 Optical/Material Properties

Modules are not restricted to specifying the geometric description of cells — they can control optical and material properties as well. This has already been demonstrated by the Set plug-in module, built into the system. In addition to setting the geometry generation parameters of “Shrink” and “Bevel,” Set can also specify the color of newly created cells.

A simple yet extremely effective effect is to apply a random color variation to each cell (see Figures A-2b, A-3d, and A-8). The renderer has access to a unique ID for each cell, and use that ID to vary the color. In these examples, the color for each cell is converted to HSV space, and only the saturation and value are changed.

A similar technique is to use the cell ID to select between a collection of different materials. For example, see the rocks in Figures A-2a and A-4.

As a more elaborate example, consider the Mosaic plug-in module. Its type is “input: one region, operation: modify cells,” and it takes the name of an image file as a parameter. For each cell in the region, it maps the center of the cell to a point on the image. It then colors the cell based on the color of the image at that point (see Figure A-7).

Define Module MosaicOnFrontFace {
  ...
  ... // (fill model with cells)
  ...
  Module SelectRegion {
    Face = front
    ModuleToApply =
      Module Mosaic {
        Texture = gargoyles.rgb
      }
  }
}
Chapter 4

Rendering Cellular Textures

Producing a compelling rendered image of a model tiled with cellular textures requires fine geometric detail, complex shading capabilities, and as much compute power as possible. In order to capture important rendering effects, such as rough silhouette edges and accurate self-shadowing, there is no substitute for an enormous number of triangles. The final stage of the system is to save all cell geometry to disk as triangle meshes, and to render the scene with a ray tracer.

This chapter is organized as follows. Section 4.1 discusses improvements made to a ray tracer that allow it to deal with extremely large input scenes. Section 4.2 presents the shading language used to specify the optical properties of the cells in the rendered images of this thesis. Section 4.3 describes a rendering architecture that takes advantage of the computing power available on a network of workstations.

4.1 Managing Complex Geometry

The primitive objects of the ray tracer are triangle meshes, stored in a two-level spatial hierarchy. On the higher level, the bounding boxes of all the meshes are stored in a single hierarchical tree[12]. On the lower level, each triangle mesh is subdivided into its own adaptive octree[7]. Similarly, each ray intersection is performed in two stages. First, the ray is tested against the top level tree to generate a list of bounding boxes that are sorted by distance. Second, the ray is intersected against the octree of each
mesh in the list, in order, until the closest triangle intersection is found.

This scheme is straightforward if the geometry of all cells in the scene can fit in memory. Unfortunately, for most interesting scenes this is rarely the case. The remainder of this chapter discusses two techniques, instancing and caching, that allow the ray tracer to work with scenes that contain a large number of highly complex cells.

4.1.1 Instancing

A typical scene can contain tens or hundreds of thousands of cells, many of which are nearly identical. It is impractical and often unnecessary to create and use a separate triangle mesh for each individual cell. The idea behind instancing is for similar cells to share the same triangle mesh. This dramatically saves the system time in creating the cell geometry, reduces the necessary disk storage space, and reduces the memory requirements of the renderer.

After the system has evaluated the cellular texturing description file and tiled the base mesh with cells, the newly created cells are analyzed for similarity. Cells whose spatial dimensions are all within a small threshold of each other are grouped, and a representative cell is chosen. A single triangle mesh is generated, and a transformation is created for each cell to place the mesh at the appropriate location in the scene. In addition, a scaling transformation is used to correct for small variations between the size of each individual cell and the representative cell.

In Figure 4-1, R's represent actual triangle meshes, and I's represent instanced meshes. There is one instanced mesh per cell. Each stores a reference to an actual mesh, along with the modeling transform to place the shared mesh at the correct location in the scene. The top-level hierarchical tree of the scene stores the transformed bounding box of each instanced mesh in world space, while the octrees for each actual mesh are stored in object-space. After intersection with the top level tree, the ray is transformed into the object space of each potentially-hit mesh for intersection with its octree.
4.1.2 Caching

Even with instancing, the geometry created by the system can be on the order of gigabytes, many times larger than can fit in memory. To handle this much data, a geometry cache[19] is used, on a per-mesh basis. A skeleton of the scene — the bounding box of each triangle mesh and the top-level tree — is always present in memory. The actual triangle meshes, however, are cached using a least-recently-used criteria.

A limit is placed on the total memory space available for triangle meshes. Initially, just the bounding boxes of each mesh are loaded by the renderer. The full meshes are loaded only as needed. When the total size of all the meshes in memory exceeds the limit, the mesh accessed least recently is deleted to free up space.

The caching scheme works best when rays that hit the same mesh are traced near each other in time. Tracing rays in square blocks of the image, rather than in scanline order, produces a more coherent mesh access pattern and reduces cache misses.

In Figure 4-1, M’s represent actual triangle meshes on disk, C’s represent meshes cached into main memory, and R’s represent references to actual meshes, either in the cache or not. Intersection with the bounding boxes of the top level tree produces a list of instanced cells to be intersected. For each instanced cell, the ray is transformed into object space and tested against the (tighter) object space bounding box. If this test succeeds, then the ray is tested against the actual triangle mesh, loading it into the cache if necessary. Loading the mesh can be avoided if the ray misses its bounding box, or if an intersection is found with another mesh closer to the camera.

4.2 Shading

Shading languages, such as RenderMan\textsuperscript{TM}[9], are a popular method for defining the surface appearance of objects in sophisticated rendering systems. The user writes a function, usually called a “shader,” that is invoked for each sample and evaluates to the color of the light reflecting off the object. The renderer supplies the shader with all the relevant information, such as the color of the object, the surface normal, and
Surface appearance is defined in the ray tracer used in this thesis with a novel shading language. The language is designed to meet four competing goals of a system that provides shading functionality in computer graphics. These four goals are: speed, power, flexibility, and ease-of-use.

- **Speed** is always an issue. Computers will never be as fast as we want them to be, so a well-designed system should run as efficiently as possible. In computer graphics, speed not only determines how long it takes to render a frame, but how complex a scene the user is willing to try to render.

- **Power** directly affects the productivity of the user and determines the overall effectiveness of the system. A shading system should allow the user to express and implement his or her ideas in a natural way, rather than make the user conform to the system. Power determines what new ideas the user is willing to attempt with the system before looking elsewhere for a solution.

- **Flexibility** is important in the design phase of a rendering project. Not only should the system provide flexibility to the users, but the system should allow users to provide flexibility to themselves. A shading system should allow the user to build a library of components that can be reused and combined in a variety of interesting and unique ways. Most importantly, a system should facilitate experimentation.

- **Ease-of-Use** is the fourth goal. A novice with no programming experience should be able to use the system to produce interesting and novel results. The user should be able to experiment with different parameters and structures of components quickly, easily, and intuitively.

Typically, shading languages are powerful and flexible, yet fail in the areas of speed and ease-of-use. An ideal shading system should excel with respect to all four of these goals. Towards this end, we introduce the following architecture for shading in a 3D computer graphics system and use it with this ray tracer. Rather than
attempt to meet all four design goals at once, two tightly integrated components are used: *shaders*, which provide speed and power, and *materials*, which provide flexibility and ease-of-use. After an overview of the shading process, shaders and materials are presented in more detail.

### 4.2.1 Overview of Shading

The shading process in the ray tracer is centered around a structure of shading variables (see Table 4.1). For each sample, this structure is passed from the visibility algorithm, to the shading system, and then to the lighting model. The shading system is allowed to access all variables and to do whatever it wants to them before the sample is lit. For example, a simple material may just change the diffuse color based on the surface point. A bump map shader would alter the normal vector based on the $s$ and $t$ texture coordinates and partial derivatives.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse Color</td>
<td>color</td>
<td>Material</td>
</tr>
<tr>
<td>Ambient Coeff.</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Diffuse Coeff.</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Specular Coeff.</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Shininess</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Reflection Coeff.</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Transmission Coeff.</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Index of Refraction</td>
<td>float</td>
<td>Material</td>
</tr>
<tr>
<td>Surface Point</td>
<td>point</td>
<td>Renderer</td>
</tr>
<tr>
<td>Normal</td>
<td>vector</td>
<td>Renderer</td>
</tr>
<tr>
<td>$s$ Texture Coord.</td>
<td>float</td>
<td>Renderer</td>
</tr>
<tr>
<td>$t$ Texture Coord.</td>
<td>float</td>
<td>Renderer</td>
</tr>
<tr>
<td>$\partial P/\partial s$</td>
<td>float</td>
<td>Renderer</td>
</tr>
<tr>
<td>$\partial P/\partial t$</td>
<td>float</td>
<td>Renderer</td>
</tr>
<tr>
<td>Cell ID</td>
<td>int</td>
<td>Renderer</td>
</tr>
<tr>
<td>Cell Center</td>
<td>point</td>
<td>Renderer</td>
</tr>
<tr>
<td>Correct Normal</td>
<td>boolean</td>
<td>Renderer</td>
</tr>
</tbody>
</table>

Table 4.1: Shading Variables

The variables are initialized in one of two places. Some variables are initialized by
the material. These variables include the diffuse color and the shading coefficients. If no value is specified by the user, reasonable defaults are used. The other variables are initialized by the renderer for each sample. These variables include the surface point, normal, and texture coordinates.

4.2.2 "Shaders"

Shaders are the more powerful of the two shading components, and most closely resemble the shaders of other shading languages, such as RenderMan™[9]. Shaders are written in C++, compiled by the user, and automatically loaded and linked by the renderer at run-time. The user is essentially extending the renderer with his or her own compiled code inserted right in to the middle of the rendering pipeline. Since shaders are written in C++, programmers can immediately draw upon their existing skill, access the full power of the language, and even incorporate existing code.

Every shader is derived from the base class Shader, and implements the following virtual function:

```cpp
void ApplyShader(ShadingVariables &SV) const;
```

This function has access to all of the shading variables and is allowed to change any or all of them. This is similar to the global variables to which RenderMan™ shaders have access.

In addition, shaders can define parameters to be set by the user upon instantiation. These parameters are implemented as data members of the class, giving the `ApplyShader()` function access to read (but not modify) them. Parameters are the means by which a shader's functionality can be controlled. The more parameters the shader's author provides, the more different applications another user will find for the shader.

Shaders do not have to produce a complete shading effect. In fact, it is better for a shader to do something very specialized, such as creating veins of marble, applying a bump map, or dividing a surface into implicit cells. Many shaders, each which
implement their own specific effect, can be combined using materials to produce the final appearance for an object.

4.2.3 “Materials”

Materials are the other half of the shading language. They are written by the user in the scene description file, using a very simple script-like language. Simply stated, a material is a list of shaders, each of which are applied in turn. A material allows the user to combine as many shaders as the user wants, to apply the shaders in any order, and to specify parameters to control the functionality of each shader. This makes materials extremely flexible. However, since the user is limited to using existing shaders, materials are the less powerful of the two shading elements. It is the flexibility of being able to combine shaders in novel ways that allows the user to create new and interesting appearances.

Listing a shader in a material instantiates the shader. At this point, a Shader object is allocated, and the parameters (data members of the class) are initialized to the specified parameters. Thus, a shader can be used multiple times with different parameters. Internally, a material is represented as a set of initial shading variables and list of Shader objects. When a material is evaluated, a ShadingVariables structure is initialized with some variables from the material and others set by the renderer (refer to Table 4.1). Then the ApplyShader() method is simply called with the ShadingVariables structure for each Shader in the list.

4.2.4 How Shaders and Materials Work Together

Even though shaders and materials are two very different constructs, they are designed to be used together seamlessly. Nearly all complex surface appearances in this system are built out of a combination of several materials and shaders. Materials apply a list of shaders, but this is only half of the story. The other half, that closes the co-recursive link between shaders and materials, is that shaders can take materials as parameters.
As an example, consider the Marble shader. For each point, it determines if that point is inside a vein, and if so, evaluate a new material. The user can specify any vein material they like:

```
Material Veins {
    Color < 0.25 0.05 0.15 >
    Apply VaryColor {
        Variation = 0.3
        Scale = 500
    }
}

Material BumpyBlueMarble {
    Color < 0.1 0.2 1.0 >
    Ambient 0.1
    Diffuse 0.7
    Specular 0.2
    Shininess 75

    Apply Marble {
        Mat = Veins
        Frequency = 3
        Amplitude = 1.5
        Scale = 10
    }

    Apply NoiseBumps {
        Scale = 1000
        Height = 0.75
    }
}
```

There is no bound to the depth of nested shaders and materials. The example above is nested four levels deep: the BumpyBlueMarble material applies the Marble shader, which evaluates the Veins material, which applies the VaryColor shader. The BumpyBlueMarble shader could either be assigned to an object, or it could be used as a parameter to yet another shader defining a more complex surface appearance.
4.3 Rendering System Architecture

A block diagram of the ray tracer is shown in Figure 4-1. Rendering jobs are managed using a Worker/Client/Server model. The meshes for each scene are stored on a file server (M’s in the figure) and accessed by each render worker individually. Each render worker caches meshes (C’s in the figure) from the complete list of actual meshes for a scene (R’s in the figure). Each cell instances a mesh from this list (I’s in the figure).

Figure 4-1: Rendering Block Diagram

4.3.1 Render Workers

One render worker is run on each individual machine. At startup, they register with the render server to let it know they are available to do work. Workers can connect to or disconnect from the server at any time.
A render worker waits for a job from the server, and then invokes the ray tracer. Once the scene is loaded, the worker waits for the server to tell it which part of the image to render. The worker receives one tile of the image at a time, ray traces that tile, and then sends the rendered pixels back to the server.

The render workers take advantage of multiprocessor machines, requesting a different tile for each processor. Workers running on different architectures can also connect to the same server. The images in this thesis were rendered with workers both on SGI machines and a 4-processor PC running Linux.

### 4.3.2 Render Clients

Render clients are the user's interface to the server. The client allows the user to send commands to the server, and to request information from the server. The user can submit jobs to the server to be rendered, view a list of the current jobs in the queue, and check the status of a job currently being rendered.

![Render Client User Interface](image)

Figure 4-2: Render Client User Interface

A graphical user interface (see Figure 4-2) lets the user select a scene to render, and set various rendering parameters. The job can then be submitted to the server.
A web-based client accesses and displays the status of all jobs and workers connected to the server.

### 4.3.3 Render Server

The render server is the heart of the rendering system architecture. The server maintains connections to all render workers and clients. In addition, it keeps a queue of all jobs submitted by the clients, and assigns workers to each job.

When the server receives a job from a client, it places it in a queue. If there are workers available, the server assigns workers to the job and tells them to begin. When a worker requests a tile to render, the server hands it the next tile to be rendered, and when a worker sends back pixel data, the server pastes it into the final image. Once an image is complete, the server informs all workers that their job is done and sends the image back to the client that requested the job. Then, if the queue is not empty, the server assigns the workers a new job.
Chapter 5

Results

Figure A-2 contains images created with the interactive framework for cellular texturing, discussed in Section 3.1. Image (a) shows the result of applying various cellular textures to a large CAD model. We used our system to define regions on the surface of the model, and then to assign patterns to each region. This image contains 2000 unique triangle meshes, composed of a total of approximately 14 million triangles. The pattern generator that was used to tile the sides of the building with rocks produced many cells, each with a unique shape. For a pattern such as this, instancing is of no benefit. Rendering this scene would not have been possible without the geometry cache, which kept at most only 1 million triangles in memory at any given time.

Image (b) in Figure A-2 shows a close up view of cells generated by a roof tiles pattern generator. Notice how the tiles conform to the surface of the model and how the random color variations make each cell unique. Image (c) shows stones placed by an arch pattern generator, this time with more dramatic shading variation. Image (d) shows both bricks of the window pattern and stones placed by the stone wall pattern generator. The initial cells generated by the pattern generators contain at most 20 polygons. The cell geometry generation techniques discussed in Section 3.4 give this image much of its visual appeal.

Figure A-4 shows a complex model tiled with cellular textures specified with the cellular texturing module BrickWallWithWindows discussed in Section 3.3. There are
8388 cells in this image. However, the instancing technique of Section 4.1.1 enabled the system to output only 14 distinct triangle meshes for this scene.

Figure A-6 demonstrates feature identification as well as an interesting pattern generator. The base mesh has two rectangular windows, and these are identified by looking for cycles of four edges labeled "concave" connected by four corners labeled "saddle." A window pattern generator creates random-length cells to frame the window. The interior of the regions are filled with a pattern generator that repeatedly grows rectangular cells from random locations until the surface is covered. This scene contains 27822 cells, instanced from only 200 unique actual meshes.

The final example, Figure A-8, shows a castle model tiled with cellular textures. The cellular texturing program first identifies windows and frames them with large stone blocks, then horizontal edges are identified and filled with smaller stone blocks. Vertical edges, which are part of the brick pattern of two adjacent regions, are filled next. The regions are filled last with bricks of the same size, blending with the edges, and thus seamlessly wrapping around the corners. This scene contains 99485 cells, with an average of 84,300 triangles per cell. There are 839 million triangles in this image. The instancing algorithm compressed the scene to 37 distinct meshes, with a total of 3.5 million triangles. The image was rendered at a resolution of 4096x4096 pixels, with 4 radiance samples per pixel and 6 shadow rays per sample, and took 9.5 hours on a network of workstations using a total of 8 CPUs.
Chapter 6

Conclusion

We have explored many aspects of the modeling and rendering of cellular textures. Our experience with implementing and using the system has taught us a great deal, and has revealed many interesting directions for further research.

We believe it would be valuable to experiment with a broader range of interactive controls. For example, it would be natural to allow the user to manually outline cells within regions to augment the cells created by procedural pattern generators. It would also be interesting to be able to paint directly on the cellular patterns to modulate their material and optical properties. Finally, it would be useful to be able to slide patterns around on the surface similar to the approach proposed by Pedersen[17].

We experimented with a variety of procedural pattern generators, with mixed success. It is clear to us that it would be possible to make even better use of the underlying 3D geometry for pattern generation. Employing a three-dimensional occupancy map that could support the placement of cells relative to one another in 3D appears to be an interesting direction. We are also interested in experimenting with pattern generators based on a small number of pre-defined components and optimized fills [5]. For example, cells could be placed in a region so as to minimize the gaps between them.

The strategy for filling a model for cells proved to be a powerful technique for cellular texturing. The system used to generate the images in this thesis placed the restriction on the input model that all faces must be rectangular with axis-aligned
edges, ensuring that all edges measure either 0 or 90 degree. As a further restriction, all cells generated by the system were described by axis-aligned bounding boxes. The design of the language does not assume any such restrictions, and our initial results have encouraged us to continue with a more general implementation.

A final promising area for future investigation is level-of-detail. The cell geometry generation techniques are procedural, and there is obvious potential for them to create meshes of various complexity, or even to create multi-resolution meshes, and allow the renderer to select the appropriate level-of-detail for each cell. For more distant regions, the system may be able to simply output 2D texture maps, doing away with geometry completely. Such techniques will be essential if we wish to scale the system up to modeling and rendering cellular textures on highly complex models, or even entire cities.
Appendix A

Color Plates

Figure A-1: Implicit cells, (a) before and (b) after
Figure A-2: Building Tiled With Cells
Figure A-3: Brick cube (a) base mesh, (b) after corners have been filled, (c) after edges have been filled, (d) after regions have been filled.
Figure A-4: Complex Base Mesh
Figure A-5: Cell Decimation, (a) before and (b) after

Figure A-6: Feature Identification
Figure A-7: Modules can set the optical parameters of cells
Figure A-8: Castle
Bibliography


generation. In Robert Cook, editor, SIGGRAPH 95 Conference Proceedings, 
Annual Conference Series, pages 239–248. ACM SIGGRAPH, Addison Wesley, 
August 1995. held in Los Angles, California, 06-11 August 1995.


In Forest Baskett, editor, Computer Graphics (SIGGRAPH '90 Proceedings), 


