Volumetric Surface Sculpting

by

Aseem Agarwala

S.B., Computer Science and Engineering
Massachusetts Institute of Technology (1998)

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

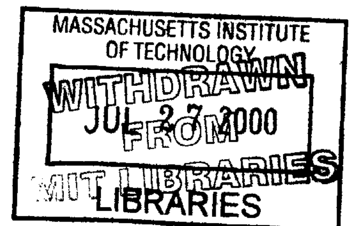
© Massachusetts Institute of Technology 1999. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 8, 1999

Certified by ....
Julie Dorsey
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

ENG



Volumetric Surface Sculpting

by

Aseem Agarwala

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In computer graphics, two types of geometric representations are widely used: *surfaces* and *volumes*. Surfaces are infinitely thin skins defining the boundary of an object, and have been successful in modeling realistic appearances. However, their lack of information about the solid nature of objects make them inadequate for many applications. Volumes represent the entire solid geometry of an object in a three-dimensional grid. Volumes, however, are inefficient and cannot achieve high resolutions since they must maintain information about internal geometry that is not pertinent to rendering realistic appearances.

We have created a new, hybrid data structure, the *volumetric surface*, which combines the benefits of surfaces and volumes. A volumetric surface represents solid materials in a thin region near the surface of an object in an efficient, multi-resolution data structure. Volumetric surfaces are thus designed to exploit the coherency of roughly surface aligned-layers of materials which are ubiquitous in real-world materials. Examples of such materials include wood, stone, and building facades.

To demonstrate the utility of volumetric surfaces, we present a system for the interactive sculpting of weathering and erosion effects on building facades. High-resolution sculpting of layers of solid brick, plaster, and mortar are not possible with surfaces or volumes alone. Volumetric surfaces are well suited to the representation and intuitive editing of such complex appearances. This thesis presents detailed algorithms for the sculpting system with a focus on efficient rendering and manipulation of volumetric surfaces.

Thesis Supervisor: Julie Dorsey
Title: Associate Professor

Acknowledgments

First I would like to thank Professor Julie Dorsey for giving me this project and supporting me with funding, office space, equipment, and other resources. I would also like to thank Justin Legakis for the use of his ray tracer, his excellent C++ library JLLib, and general graphics and C++ advice. Bob Sumner and Hans Pedersen were also very helpful, along with the rest of the MIT Computer Graphics Group.

More thanks go to my family, including my parents Om and Kala Agarwala, my sister Anjalika, and my brother Amit. Their support is much appreciated. I also want to thank all the other people I have worked for over my time here at MIT including Matt Brand at the Media Lab and Carol Strohecker and Joe Marks at MERL. Their support really helped to cultivate my interests in computer graphics and user interaction, and their faith in me is much appreciated.

And finally, thanks to the 249-foolz, the TDC crew, the bone boyz, and the brew-dogg. Peace.

Contents

1 Introduction **9**

1.1 Goal . . . 11

1.2 Organization of Thesis . . . 12

1.3 Traditional Modeling Primitives . . . 12

1.3.1 Surfaces . . . 12

1.3.2 Volumes . . . 13

1.4 Related Work . . . 14

1.5 Organization of System . . . 16

2 Hierarchical Volumetric Surfaces **17**

2.1 Slabs . . . 17

2.1.1 Complications . . . 19

2.1.2 Advantages . . . 19

2.2 Inside the Slab . . . 20

3 Creating Solid Geometry **23**

3.1 Slab Geometry . . . 23

3.2 Solid Textures . . . 24

3.2.1 Materials . . . 25

3.3 Sampling Solid Textures into Slabs . . . 25

3.3.1 Subdivision . . . 26

3.3.2 Sampling the Solid Textures . . . 26

3.4 Discussion . . . 27

4 Interactive Rendering 29

4.1 Design Requirements 29

4.2 Design Alternatives 30

4.2.1 Hardware Accelerated Volume Rendering 30

4.2.2 Ray Casting 30

4.2.3 Marching Cubes 30

4.2.4 Height Fields 31

4.3 Final Design 32

4.3.1 Culling Unnecessary Geometry 32

4.3.2 Quadtree Traversal with Neighbors 33

4.3.3 OpenGL Rendering 33

4.3.4 Results 34

4.3.5 Further Optimizations 34

4.4 Shading 36

4.4.1 Coordinate System Transformations 37

4.4.2 T-joints 37

4.4.3 Single Corner Traversal 38

4.4.4 Normal Storage 39

4.4.5 Results 40

4.5 Model Navigation 42

5 Interactive Sculpting 43

5.1 Graphics Updating 43

5.2 Tool Editing of Geometry 46

5.2.1 Tools 46

5.2.2 Tool Modification Steps 47

5.2.3 Tool Specifics 49

5.2.4 Modifying Normals 51

5.3 Results 52

6 Final Rendering 54

6.1 Design Alternatives 55

6.1.1 Marching Cubes 55
6.1.2 Direct Ray Tracing 55
6.2 Final Design 56
6.2.1 Restricted Quadtree Triangulation 56
6.2.2 Growing Height Fields 59
6.2.3 Patching Height Fields 64
6.2.4 Shading 66
6.3 Results 67

7 Discussion and Conclusion **69**
7.1 Shaded Results 69
7.1.1 Crumbled Brick 69
7.1.2 Plaster Over Bricks 70
7.1.3 Building Corner 70
7.2 Future Work 73
7.3 Conclusion 75

List of Figures

1-1 Examples of weathering effects on Venetian building facades. 10
1-2 Flow chart of the sculpting system. 16

2-1 A slab and its local coordinate system. 18
2-2 A brick wall composed of 81 slabs. 18
2-3 A stone column composed of slabs. 18
2-4 A quadtree. Shown is (a) the spatial subdivision, (b) the tree hierarchy. 21
2-5 A linked list of materials, consisting of air, plaster, and brick from top down. 21
2-6 A complete slab with quadtree and materials. 22

4-1 Meshing algorithm on untouched brick wall. 35
4-2 Meshing algorithm on a sculpted section of brick. 35
4-3 Calculation of normals. 36
4-4 T-joint problems in shading. 38
4-5 Touching of the side corners. 40
4-6 Single corner traversal order. 40
4-7 Shading algorithm on untouched brick wall. 41
4-8 Shading algorithm on a sculpted section of brick. 41

5-1 Two views of a chisel. 50
5-2 Wireframe model of a sphere cut. 53
5-3 Shaded model of the same sphere cut. 53

6-1 a 2D example of patched height fields. 56
6-2 An example of a crack problem in quadtree triangulations. 57

6-3 A correct restricted quadtree triangulation. 57
6-4 Differences in brick subdivision that lead to bad shading. 59
6-5 The vertex join data structure. 60
6-6 An SAI's pointers to vertex joins. 60
6-7 Examples of surface sharing for neighboring SAI's. 61
6-8 A case of three found surface sharing agreements. 63
6-9 A contradicting case of four agreements. 63
6-10 Patch case one. 65
6-11 Patch case two. 65
6-12 Patch case three. 66
6-13 An example of final meshing on a tooled area. 68
6-14 Annother example of final meshing. 68

7-1 Interactive rendering of tooled, crumbling brick. 71
7-2 The ray-traced rendering. 71
7-3 Interactive rendering of a tooled, layered model. 72
7-4 The ray-traced rendering. 72
7-5 Interactive rendering of a tooled building corner. 74
7-6 The ray-traced rendering. 74

Chapter 1

Introduction

Real-world appearances have a complex, time-varying nature that is frequently missing from today's computer graphics. While natural materials are crumbled, eroded, and stained, computer graphics models usually have a pristine and plastic look that is not realistic. Even when more complex appearances are modeled successfully they are accomplished through painstaking, time-consuming methods. The fundamental problem is that traditional modeling primitives used to represent appearances in computer graphics fail both conceptually and technically when used to create weathered effects.

A representative gallery of complex, eroded appearances which are hard to represent using traditional computer graphics techniques is shown in Figure 1-1. The first thing to note about these appearances is that they are volumetric in nature. The crumbling brick in 1-1a would not be well represented by a surface mesh. If an artist wished to crumbled away brick, or remove plaster in 1-1c to reveal more brick, a surface model would not provide the information about solid materials that would be necessary to allow such operations. On the other hand, representing such materials with a full volumetric model would not be practical. The information that leads to these appearances is contained in a region near the surface, and the overhead of maintaining internal volumetric information would prevent us from reaching high levels of resolution.

In this thesis we introduce a new data structure, the *volumetric surface*. The volumetric surfaces is a hybrid data structure that combines the benefits of volumes

Figure 1-1: Examples of weathering effects on Venetian building facades.

and surfaces. A thick layer of volumetric data is maintained near the surface in local, surface aligned coordinate systems. We further enhance our data structure by making it adaptive; we allow resolution in the model to vary depending upon where realistic detail is most needed. A key feature of volumetric surfaces is that it is a conceptually intuitive way to model appearances, and thus straightforward methods exist to edit and model using volumetric surfaces. To this end, we present a three-dimensional sculpting system in which users can directly edit geometry in 3D using real-world sculpting tools. To focus our efforts we will work exclusively with the sculpting of layered effects on building facades. The examples in Figure 1-1 depict the kinds of appearances we may be able to model.

1.1 Goal

The goal of our system is to be able to allow artists to sculpt complex appearances at resolutions greater than what has previously been possible. We feel that such appearances can be modeled best at resolutions on the order of millimeters. The specific goal, then, is to create an interactive sculpting system that can keep a whole building facade in memory and allow modifications on the order of millimeters. This is several orders of magnitude higher than what has been possible with other volumetric sculpting systems. To keep the system interactive rendering rates must be maintained above 20 frames per second. This certainly presents a difficult challenge, and some of the key issues are mentioned below.

- The first and hardest challenge is speed. Allowing the modeling of highly complex geometry at interactive rates requires that all algorithms in critical loops be as efficient as possible. For algorithms that are used heavily in interactive loops, it is sometimes necessary to sacrifice visual quality for speed.

- Another challenge is memory conservation. Storing large amounts of geometry at high resolutions can require prohibitive amounts of memory, and so it is necessary to make the data structures as efficient as possible.

- A high-resolution sculpting system is useless if it is not easy to use. Therefore effective tools, an intuitive interface, and good tool control are necessary.

1.2 Organization of Thesis

The remainder of this chapter will describe traditional modeling primitives that volumetric surfaces build on, related work that has inspired volumetric surfaces, and an overview of the system. Once this is accomplished, a more complete description of the data structure is presented. This is followed by a description of the various components of an interactive sculpting system. Results are presented, as well as a discussion of future areas of improvement.

1.3 Traditional Modeling Primitives

The two most common modeling primitives in three-dimensional computer graphics are surfaces and volumes. Surface models are characterized by infinitely thin surfaces around the interface of an object, thus leaving the interior hollow. Volume models are defined by a three-dimensional grid of voxels that describe how space is filled by an object. Both models have their strengths and weaknesses, but neither alone can achieve the appearances we desire.

1.3.1 Surfaces

Surface models have been extraordinarily useful in the history of computer graphics. Indeed, they are the basis of almost all modeling and animation to date. They work well because in many applications only the surface of a model is of interest.

Surfaces are usually composed of many polygons linked together at corner vertices to form a surface mesh [15]. They can be made more realistic by applying textures. More complicated variants are parametric surfaces which allow mathematically defined, curved surfaces. One of the first difficulties involved with surface meshes is they are not easy to create or edit. Many 3D surface modeling packages exist, but they are time-consuming to learn and difficult to use. Another recent option is 3D laser range scanning; real-world objects can be physically scanned to create a surface mesh [9].

Surface models become especially unwieldy when representing eroded appearances. Surfaces have a conceptual problem in that they are hollow shells, and have no notion

of filling solid space. This makes it difficult to model and edit many types of objects, such as the crumbling brick wall in Figure 1-1a. A surface model of this wall would be very difficult to build, and would consist of highly complex and irregular geometry. Even if such a surface model were built, close inspection would reveal that it is only a surface; hence it would appear unrealistic. Even more difficult would be interactive editing of such a surface model. If an artist wished to crumble away more brick or otherwise edit the geometry, this would be next to impossible to do with a surface representation. There is simply no solid material with which to work.

1.3.2 Volumes

The other common data structure for three-dimensional objects is a volume. Here, there is a notion of filled space, and it is described explicitly through an axis-aligned grid of voxels. Volumetric models have proven useful in many applications such as medical imaging where the inside of the human body can be visualized. However, volumes suffer from what is often called the 'curse of dimensionality.' The memory and rendering overhead of volume models increases with the cube of the model resolution. Also, the entire three-dimensional solid volume of a model must be stored, and this incurs huge storage and computational overhead. This internal data is useful for medical imaging or representing diffuse materials such as clouds, but is unnecessary for modeling appearances of solid objects. In these ways the resolution of volume models is severely limited, and is several orders of magnitude below what is required for realistic appearances. Also, since the voxels are axis-aligned, volumetric models often suffer aliasing problems when the surface of the model lies at strange angles relative to the major axes. Axis-aligned voxels are thus not well suited to capturing realistic, high-resolution effects that lie mostly near the surface.

The major benefit of volumes, however, is that it is an intuitive way to represent many of the appearances we seek to create. It is also easy to edit interactively, and several volume sculpting systems have been created to demonstrate this functionality [16, 41].

1.4 Related Work

There has been much work done in the computer graphics field to model complex, natural phenomena. Many interactive modeling tools have also been created. It is useful to mention notable related work in surface sculpting, volumetric sculpting, texels, layered models, erosion models, and interactive systems.

Surface models have been effective in capturing properties that occur only at the surface of the model. Two-dimensional texture maps can be used to specify color, transparency, and other 2D properties at high resolution. There have also been attempts to capture 3D effects on surfaces. Bump maps [6] can be used to perturb normals, thus simulating 3D-like textures. However, this technique is an attempt to capture effects that are not meant to be modeled using thin surfaces, and thus suffer from limitations such as bad silhouette images and unrealistic shadows. Displacement maps [14] are used to specify a position offset from the surface, thus capturing a 3D effect. Displacement maps are similar to our technique, but have the significant limitation that under-cuts beneath the surface can not be represented.

Many attempts have been made to use surfaces as a flexible modeling tool. Various papers have been presented on systems that allow sculpting and free-form deformations of surface models [8, 27, 32, 37, 42]. However, these attempts have met with limited results and are complicated by the cracks, folds, and self-intersections than can occur with surface sculpting. The problem is that infinitely thin surfaces are not a good primitive for sculpting; it makes more sense to sculpt solid materials.

Another approach to capturing complex phenomena using surfaces is to represent a surface as a composite of many layers. RenderMan [2] has been used this way with good success, as evidenced by some of the convincing imagery from Pixar's films. This technique was also used to model metallic patinas on surfaces as they weather over time [12]. While many effects can be captured with layered geometry, they are still limited by their attachment to a surface.

Volumes are another approach to 3D modeling. Volume models treat a model as a solid, filled object in space. This makes many modeling tasks, such as sculpting, much easier. Volume sculpting was introduced in [16]. In this system a user could move a

tool in 3D space and locally edit the volumetric information. At each step marching cubes[24] was performed to calculate a surface model that was then rendered. Volume sculpting was significantly extended in [41], where localized volume ray-casting was used. A haptic interface was added to volume sculpting in [5], which made interacting with a 3D volume much more intuitive. While volume sculpting produced some nice results, all these systems are severely limited in resolution. Since only coarse models can be sculpted the resultant objects are not at all realistic or impressive. By discarding the inner voxels we hope to be able to provide for high-resolution volumetric sculpting near the surface of the object.

Another approach to modeling with volumes is volumetric textures, or texels. Texels were introduced in [21] and were very effective in rendering furry surfaces. Texels are similar to our notion of volumetric surfaces in that it attempts to maintain information of a volumetric nature near the surface of an object. Also, our notion of a thick skin around models is shared by texels. Texels were extended in [29, 28, 26]. Texels work well when attempting to model complex repetitive geometries like fur and foliage near the surface of an object. However, they cannot be interactively sculpted. Also, each texel must contain one of a limited number of pre-defined objects, and these cannot be modified in any way.

There are various other examples of previous systems that have inspired the direction of this work. In [13] eroded appearances caused by water flow were modeled using particles systems. An excellent example of an elegant interactive modeling system was presented in [18], where a user was able to paint directly on 3D surfaces. The painter could even use displacement maps to capture more complex effects. Many of the design metaphors presented in this paper affected interactive systems that have followed it. More sophisticated 3D painting systems have followed [1], but since all these systems work with surfaces they suffer from the same limitations.

And finally, the concept of volumetric surfaces was first introduced in [11]. This system, which was developed concurrently with our sculpting system, uses volumetric surfaces to numerically simulate weathering effects in stone. There are two main differences between this system and ours. First, the stone weathering system fills the volumetric surfaces with a discrete volumetric grid. We use a different, multi-

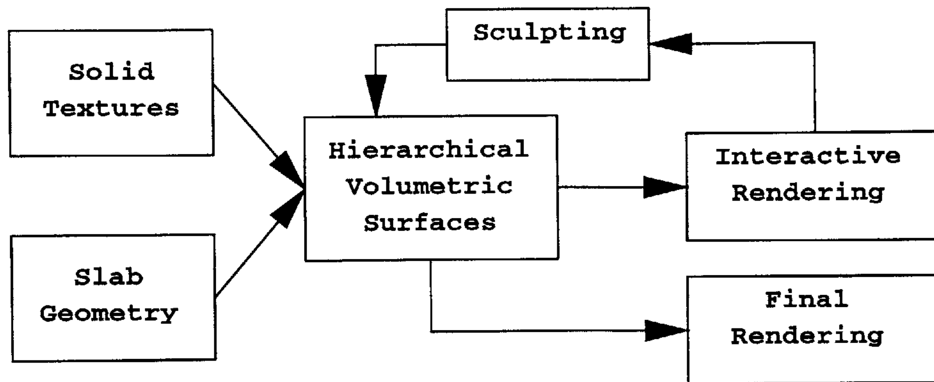


Figure 1-2: Flow chart of the sculpting system.

resolution data representation that will be described in depth. Second, the stone system uses numerical simulation to achieve complex appearances. We focus on interactive sculpting.

1.5 Organization of System

The sculpting system was implemented in C++ [40] using the openGL [44] graphics library. It is designed to run on graphics hardware accelerated machines such as SGI's. The overall sculpting system is organized according to the flow chart shown in Figure 1-2. The first task is to create the initial volumetric surface geometry. This is done by combining volumetric surface containers (called *slabs*) and solid textures (Chapter 3). Once this is done we pass the data to an interactive sculpting system for editing. The two main components of this system are the rendering of geometry (so that the user can see where he or she is sculpting, Chapter 4), and the actual algorithms involved in altering the geometry (Chapter 5). Once the user is done sculpting, a high quality surface mesh of the volumetric surfaces is created for ray-tracing and final image output (Chapter 6).

Chapter 2

Hierarchical Volumetric Surfaces

The hierarchical volumetric surface is a multi-resolution data structure for representing complex, volumetric geometry near the surface of an object. It is designed to exploit the coherency found in models that consist of layers of materials that are approximately surface-aligned. While this data structure has the potential to be useful in a wide range of applications, we will focus our examples on the modeling of weathered building facades consisting of layers of materials.

2.1 Slabs

The fundamental unit of volumetric surfaces is the *slab*. A slab is a volume delineated by eight corners, as shown in Figure 2-1. This volume has a local coordinate system defined as the tri-linear interpolation between eight corners. The interpolation constants u, v, w are valid in the range $[0, 1]$ and are the variables in the local coordinate system.

These slabs are put together to form a thick skin around an object. Slab geometry is constrained such that each slab may only have North, South, East, and West neighbors, and neighbors must share the appropriate vertices. An example of a brick wall composed of a 9×9 grid of slabs is shown in Figure 2-2. Another example of slabs composing a section of a stone column is shown in Figure 2-3. Note that slab geometry is very similar to texel geometry in [29].

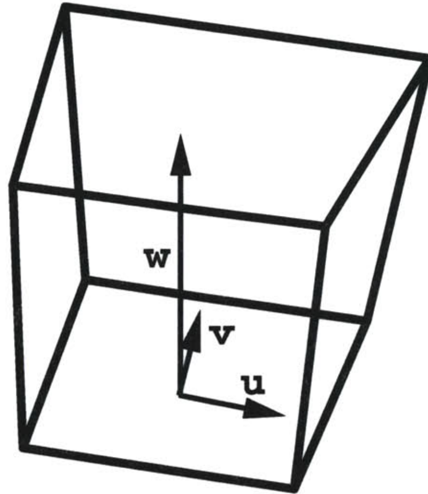


Figure 2-1: A slab and its local coordinate system.

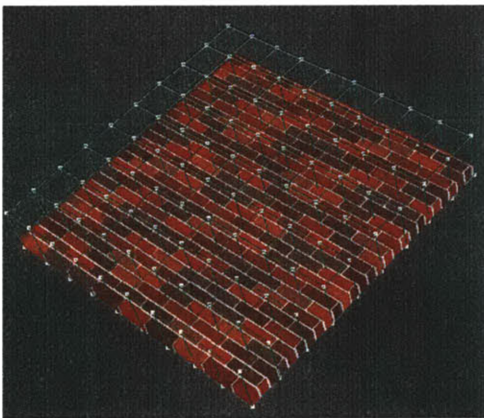


Figure 2-2: A brick wall composed of 81 slabs.

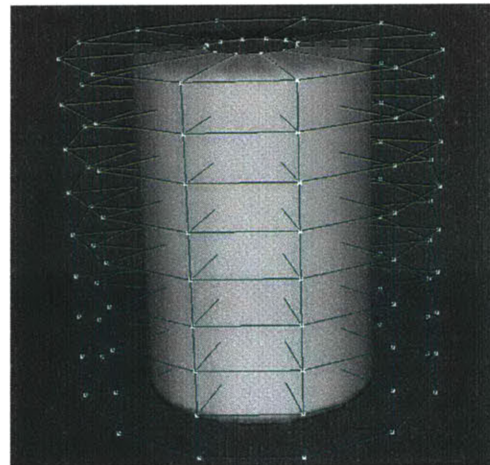


Figure 2-3: A stone column composed of slabs.

2.1.1 Complications

One of the complications this structure creates is that the data is in a separate coordinate system from the global coordinate system where geometry would be rendered. Thus, two of the most heavily used procedures in our system convert local coordinates to global, and vice versa. Converting local coordinates into global is a simple tri-linear interpolation of the slab corners. This procedure is the most heavily used during interactive rendering, and so it must be computed quickly. The fastest algorithm is presented in [20] and can accomplish tri-linear interpolation of three-dimensional coordinates in 21 multiplies. This algorithm was implemented in our system. Converting from global to local coordinates is much more difficult as it involves taking the inverse of a cubic polynomial. So, an iterative numerical scheme must be used to calculate this conversion. As a result it is important to minimize the use of this conversion in any critical loops.

Yet another complication is the distortion problem. The slabs are not constrained so that opposite sides of the slab are parallel. This can cause the local coordinate system to be distorted. This means that a straight line in slab coordinates may be curved in global space, and vice versa. This causes many problems, as many of the assumptions we make in Euclidean geometry will fail as we go between coordinate systems. Examples of these problems will appear later.

2.1.2 Advantages

This slab structure is complicated but has several advantages. For one, slabs of local coordinate systems allow us to roughly align coordinate systems and the surfaces they are representing. In traditional volumetric schemes there is one global, axis-aligned coordinate system. This leads to aliasing effects when the surface of the object cuts at angles across the global coordinate system. Here we minimize this effect by using separate, local coordinate systems that follow the surface. This allows the appearance of higher resolution without additional storage.

Another advantage is that these slabs are well positioned to take advantage of coherency in objects that are composed of layers of materials. Since the local coordinate system is roughly surface-aligned, there should be significant coherency in

the u, v plane, and a constant number of material boundaries in the w direction that correspond to the number of layers of materials in the object. This allows us to use a unique, multi-resolution data structure that in the average case grows $O(n^2)$ with respect to resolution rather than $O(n^3)$ of traditional volumetric data. This is the advantage afforded by volumetric surfaces.

2.2 Inside the Slab

To take advantage of this coherency we use a two-dimensional spatial subdivision scheme called a *quadtree* [36] to fill the slab. A quadtree is a simple, hierarchical data structure that divides squares into four equal components recursively until the desired resolution is reached. An example is shown in Figure 2-4. The tree hierarchy depicts how such a quadtree would be represented in memory. The *leaf nodes* are shown in red, and the non-leaf nodes are shown in black. Such a quadtree is then stretched across the u, v plane of a slab. In local slab coordinates the lower left corner of the root level quadtree is (0,0) and the upper right corner is (1,1).

The advantage of using a quadtree as opposed to a regular grid is that we can approximate using coarse resolution areas where the object being modeled is uniform parallel to the surface. On the contrary, where fine detail is needed the quadtree can subdivide to the needed resolution. The disadvantages are two-fold. First, we must store non-leaf nodes that do not themselves contain geometry. This additional memory, however, is trivial compared to the memory saved by adaptive resolution. There are pointerless representations of quadtrees that store only the leaf nodes, but many of the traversal algorithms that we later need are not possible with such a representation. The second disadvantage is that algorithms operating on the geometry become significantly more complicated. With a regular grid, data can be accessed easily and efficiently. With a quadtree any access algorithms must be written in the form of a tree traversal.

Now that we have described the two-dimensional subdivision in the u, v plane, we can describe how the slabs contain three-dimensional volumetric data. At each leaf node of the quadtree, a linked list is stored that specifies intervals of materials. The

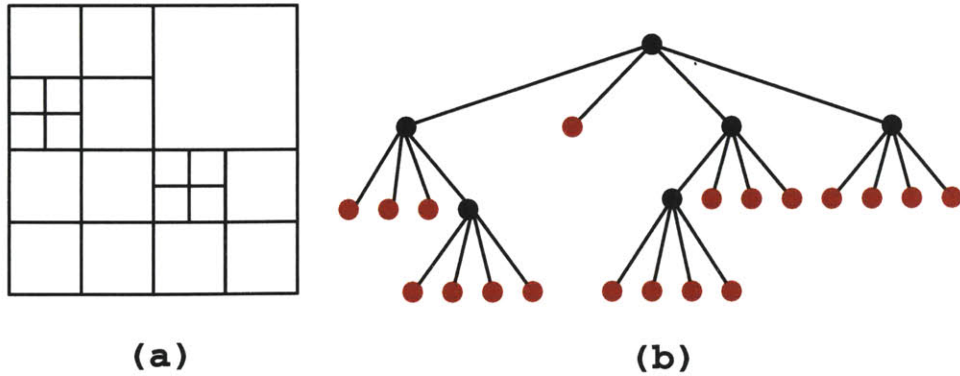


Figure 2-4: A quadtree. Shown is (a) the spatial subdivision, (b) the tree hierarchy.

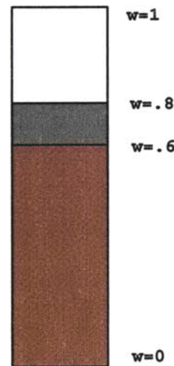


Figure 2-5: A linked list of materials, consisting of air, plaster, and brick from top down.

top of each interval is specified as a w coordinate, giving us floating point precision in the third dimension. An example list of materials is shown in Figure 2-5. This technique is similar to run-length encoding [7], except that instead of counting discrete voxels, we specify floating point intervals. Also, note that this representation allows under-cuts since intervals of air can be placed anywhere in the list.

A simple example of a complete, filled slab is shown in Figure 2-6. This slab contains a small section of a brick wall. The quadtree has subdivided to a maximum depth of 6, which means that a regular grid would need to have 64×64 cells. It can be seen from the figure that many areas were coarse and did not need this resolution. Other areas near the edges of mortar and brick boundaries did need this resolution, and subdivided appropriately. The next chapter will describe how this slab was filled with materials such as brick.

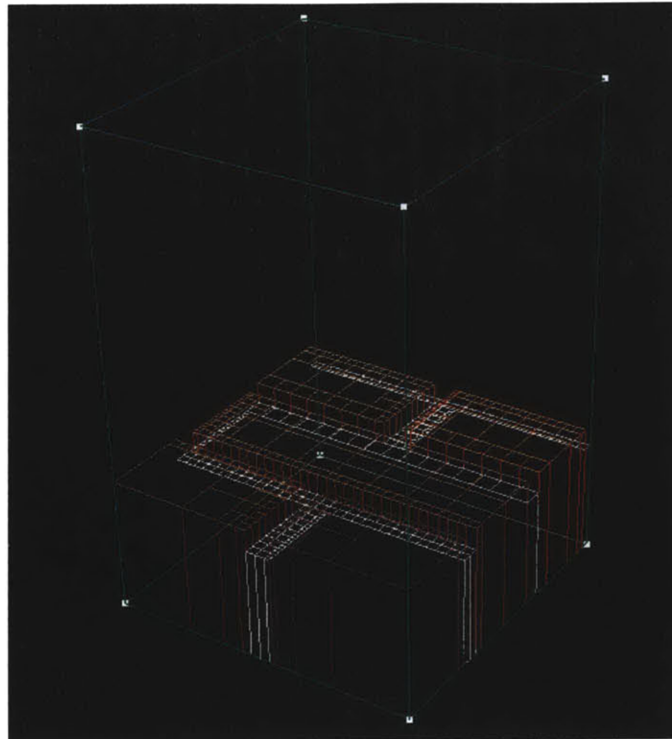


Figure 2-6: A complete slab with quadtree and materials.

Chapter 3

Creating Solid Geometry

The first task in creating a volumetric surface sculpting system is to import or create the original geometry. Our unique data structure must somehow be filled by solid material, and this must be done at adaptive resolutions.

3.1 Slab Geometry

The first task is to specify the geometry of the slabs. This establishes the local coordinate systems that we can later fill, render, and sculpt. This is fairly straightforward. First, the system takes as input a list of vertices in 3D space. The next input is a list of slabs. Slabs are specified as eight vertices that form their corners. The vertices are listed in integers that index back into the original list of vertices. It is up to the user to make sure the slabs are well formed, and that slabs that should be neighbors share the approriate vertices.

Once the slabs are specified, the system passes over the slabs and generates adjacency information. Thus each slab stores pointers to its eight neighbors. This algorithm takes $O(n^2)$ where n is the number of slabs, and is performed by having each slab check every other slab to see if it shares certain vertices. Since generating the adjacency information is a one-time pre-processing step it does not affect the performance of the system.

3.2 Solid Textures

Solid textures are function in 3D space that map a location to a material [33, 31]. That is, given an (x, y, z) point it returns whether the spot is filled or not, and if it is filled it returns the material that resides at that location.

For our simplified version of building facades we have created two general classes of solid textures. The first is simply solid material that always returns the material it represents. The other is brick, which returns *empty* if the point is in a space between bricks, or *solid* if the point is in a brick. The brick solid texture is initialized by the dimensions of a single brick, as well as the width of the gaps between bricks. A brick solid texture, then, can be thought of in brick coordinates where a single unit in x is the width of a brick plus the width of the gap, a unit in y is the height of the brick plus the width of the gap, and a unit in z is the depth of the brick plus the width of the gap. This technique is beneficial for two reasons. For one, it is easy to sample. An input position can be converted to brick coordinates by dividing by the appropriate brick unit length. By storing the percentage of the brick unit in each dimension that is actually brick (as opposed to gap), we simply compare against this number to determine the status of a position. To acheive the off-alignment of different rows of bricks, 0.5 can be added to the y and z coordinates if the floor of either coordinate is odd. Second, each brick can be referenced by a unique ID consisting of its (x, y, z) in brick coordinates. This leads to a straightforward hash function, which will be used in Section 3.2.1.

Solid textures have several other features that make filling volumetric surfaces easier. First, ranges can be set in either direction. For example, a solid texture could be set so that its x range is $(-1, 1)$. Any points outside this range would return empty. Also, solid textures can be rotated, scaled, and transformed. Rather than actually transforming the textures, the same effect is achieved by simply performing the inverse transformation to any location being queried.

Also, solid textures can be organized into layers, where earlier layers take precedence. This makes it easy to compose a brick and mortar wall. The first layer is a brick texture, with its z range set to allow only one layer of brick. The next layer

is mortar, with its z range set slightly smaller so that the brick sticks out from the mortar.

3.2.1 Materials

The material returned by a solid texture is returned (and stored) as a single byte. This byte is used to look up a fuller material description in a global list of materials. This technique saves significant memory since each interval of material in our volumetric surface must know which material it is. Currently our materials only store a color. However, since volumetric surfaces simply point to materials, many more properties could be added without significant overhead. An example would be to store a hardness factor that could control a material's response to sculpting.

This material table is also used to acheive variety in colors of brick. We want different bricks to have slightly different colors, and we do not want discernible patterns on brick coloration across a wall. We thus store 23 (this is an arbitrary prime number) different brick materials which vary in color. This color is generated by a noise function whose expected value is 'brick red.' Then, since each location maps to a specific brick ID, this ID can be hashed into the 23 brick colors. This allows any position's brick color to be computed in constant time while still avoiding any discernable patterns in brick coloration.

For our simple building facades, we have materials for plaster, mortar, and 23 different bricks. There is also a material entry for air. This framework could easily support a variety of additional materials.

3.3 Sampling Solid Textures into Slabs

Now that we have slab geometry and solid textures, the final task is to sample the solid textures into the slabs. This would be much easier if we had a discrete, regular volumetric grid: we could simply somple the solid textures at their voxel centers. Our unique data structure requires more complicated algorithms.

3.3.1 Subdivision

To begin filling a slab we create a root node quadtree, and specify the maximum depth we want from the quadtree. This determines the maximum resolution. We also pass in a solid texture, which itself can be composed of many layers of solid textures. Then, in a depth-first manner we create nodes for the quadtree down to the maximum depth. Each leaf node initializes its material list (a linked list of material intervals, as discussed in Section 2.2) by sampling the solid texture using a technique that we will discuss. Since we are using depth-first traversal, each step back up the tree will stop at a quadtree with four child nodes. This quadtree will then compare the four children nodes and decide if they can put approximated with one node. The criteria for this is as follows:

1. Each child must be a leaf node. If a child instead has four of its own children it clearly can not be approximated by one node.
2. The material lists of each child must have the same number of intervals.
3. The floating point w values for each interval and its neighbors must be the same to within a floating point tolerance (usually set as the resolution determined by the maximum depth of the tree).

If these criteria are met, we can delete each leaf node, and create a material list for this node as an average of the lists of the leaf nodes. Note that since we are using depth-first traversal, we do not initially create all the leaf nodes; this would require prohibitive amounts of memory. Instead, as leaf nodes are created they are tested for deletion as we ascend the tree.

3.3.2 Sampling the Solid Textures

To do this, we need to be able to create an initial material list from the solid texture for each leaf node. Each leaf node has a central (u, v) location, but its w extends from 0 to 1. Thus we need to sample the texture along a line. Note that this line is in global space, which means the line could actually be a curve. But, since the line is along a constant (u, v), the line is always straight.

Each solid texture, then, must be able to take two endpoints and return a linked list of materials alone the line formed between them. This is easy for solid material textures. The list is initialized to one interval of solid material, and then clipped by any range limits.

The task is much more difficult for brick patterns. It is accomplished using the observation that changes in brick material occur along constant x,y, and z planes (rotation transforms do not complicate the situation since they are applied in inverse to the input points). The list is initialized by first adding every possible location where a change in material may occur along the specified line. Thus, every x,y, and z plane within the endpoints is added, along with any range limits. The center point of each interval is then sampled in order to assign a material. Finally, since this procedure may add redundant events, one pass of the list is executed to delete redundant intervals. This allows us to create a list of materials to floating point accuracy.

3.4 Discussion

The above technique serves well to create example volumetric surfaces for our sculpting system. However, it requires that geometry be hardwired and that each solid texture be mathematically defined. This makes it difficult for anyone other than the creator of the system to create initial geometry.

The goal of our sculpting system is not to be able to create geometry, but instead to be able to import existing geometry and add volumetric weathering effects. The idea is to import a building facade, for example, and then use tools to crumble brick or remove paint and plaster to reveal layers of material underneath. The focus of the research in this thesis has been algorithms for fast, high-resolution rendering and sculpting of volumetric surfaces. The ability to import detailed geometry is a time-consuming task to engineer but does not pose interesting research questions. It is a crucial feature, however, to make our system a useful tool. The examples used in this thesis have been hardwired; this is not a feasible option for other users.

To this end, there are several possibilities for future processes to import geometry.

The challenge is that most buildings are CAD models, which can be generalized as collections of polygons. To sample into volumetric surfaces we must be able to assign a list of materials to any line in the scene. This could be done for CAD models by intersecting each line with the polygons in the model using ray-tracing techniques. This poses several problems, however. First, it is highly inefficient, but could be made faster by placing the CAD polygons into octrees [35]. Second, it places requirements on the CAD model. The polygons must correctly indicate which way they face so that we know the boundaries of a material, i.e. when we are entering material or leaving it. Second, the material of the entering polygon should match that of the exit polygon. Otherwise there is ambiguity.

A better idea would to be create a set of building blocks. There would already exist a set of pre-fabricated slabs, the most basic one being a section of brick. Other blocks (such as windows, doors, arches) could be created by allowing users to fill slabs with polygons. The system would then sample these polygon-filled slabs to create volumetric surfaces using the same technique discussed for CAD models. After the building blocks are created, an interactive system could be built to place these blocks together like assembling Legos to make buildings.

The advantage of this technique is two-fold. First, basic brick wall sections form the majority of building facades, and are best done procedurally with the techniques in this chapter rather than through CAD modeling. This will save much pre-processing time. We still allow more complex geometry to be created by a user, but in smaller chunks that can be reused. Second, this technique would allow instancing. Rather than store the volumetric texture for each brick section, we can dump this redundant information by having the slabs point to one reference block. Then, when the user actually sculpts a specific section, the reference slab could be copied in. This would allow untouched expanses of wall to be stored with little memory and would enable the modeling and storage of larger collections of buildings.

Thus, the techniques implemented for filling volumetric surfaces are able to provide us the examples we need. However, to make this system accessible to a wider audience and to make more impressive geometry possible, extensions will be necessary.

Chapter 4

Interactive Rendering

One of the harder challenges when working with hierarchical volumetric surfaces is visualizing them. To create a sculpting system, it is necessary to render the volumetric surfaces to the screen at interactive rates. This requires a technique that can render complex geometry fast and in a manner that is visually faithful to the data structure. Also, there are no established or obvious methods to render hierarchical volumetric surfaces, and so one must be chosen from the many options.

4.1 Design Requirements

There are are many techniques that could be used to interactively render volumetric surfaces. In order to choose the best method it is helpful to consider some of the design requirements for an interactive rendering method.

1. The rendering must be fast. Rendering is a major bottleneck, and the faster our rendering method the more resolution we can get from the sculpting system.

2. The method should be able to faithfully render any under-cuts in the volumetric surfaces.

3. The rendering primitives must be locally updateable. When a tool edits the data in a region of a volumetric surface, we only want to have to render the changed region rather than the entire volumetric surface.

4. The rendering method must be amenable to a multi-resolution data structure.

4.2 Design Alternatives

There are many possible rendering methods, and it is fruitful to consider some of the rejected solutions.

4.2.1 Hardware Accelerated Volume Rendering

There has been a flurry of recent research on hardware accelerated volume rendering [43]. These techniques use 3D texture hardware to render volume data in a series of slices, which is very fast. These methods are perfect for volumetric surfaces in every way except for the fact that they are not amenable to multi-resolution data structures, which is design requirement number 4. This is because the 3D texture buffer used is a regular grid. It would be interesting to see if a variant of this hardware could be built that could render volumetric surfaces at rapid rates.

4.2.2 Ray Casting

Ray casting has been used in volume sculpting systems [41] to render at interactive rates. In this technique a ray is cast from the eye point through each each pixel into the scene. A big advantage of this method is that it is an image-order technique; to render a scene, a task is done for each pixel of the image. This makes it very easy to locally update the rendering during tool operations, since we just need to render those pixels in the area of the tool. However, it also means that renderings that require the whole image to be updated (such as a rotation of the model) are very slow. Also, this technique does not use any hardware acceleration; this makes it significantly slower than polygon-based methods.

4.2.3 Marching Cubes

Marching cubes [24] is a simple, fast, and high-quality method to generate a surface mesh from a volumetric data set. It was initially not possible to do local updating of a marching cubes solution, but a technique called *incremental marching cubes* [16] developed for the first volume sculpting system solved this problem. It is also not

straightforward to use marching cubes for a hierarchical data structure, but it has been done for octrees in a technique called *adaptive marching cubes* [38].

The problem with marching cubes is that it is designed for a discrete volumetric grid, which our data structure does not have. It is possible to build such a grid from our data structure, but adding this layer of indirection would take up precious memory and processor time. It can also lead to contouring problems.

4.2.4 Height Fields

Volumetric surfaces are very similar to height fields. Height fields add 3D effects to surfaces by specifying a displacement at each vertex in a surface mesh along the normal at that vertex. This is certainly similar to our slab data structure except for one major difference: volumetric surfaces can handle under-cuts. Height fields have the advantage that they are usually rendered using polygons. Polygons can be rendered very fast on graphics hardware accelerated machines using openGL. Also, the rendering of height fields is a heavily researched field and we can borrow these previously developed techniques. One area of research [17] is polygonizing a height field accurately while minimizing the number of triangles, since fewer triangles lead to faster rendering. The problem is that these methods generally use Triangulated Irregular Networks, or TINs, which have very irregular meshes. If a local area of the mesh changes, the whole mesh must be regenerated. This violates design requirement 3.

Another area of research is the regular triangulation of hierarchical height fields organized in quadtrees [19, 23, 30]. This is clearly very similar to our data structure. The difficulty in polygonizing a hierarchical height field is the avoidance of cracks, but fast algorithms have been developed to do this. We will explore this method in Chapter 6 when we do final rendering; however, it is unsuitable for interactive rendering. First, these methods require that the quadtree be a restricted quadtree. This means that the quadtree must be balanced so that no leaf node has a leaf neighbor whose depth differs by more than one. Making a quadtree restricted involves adding many nodes, and we do not want to place this restriction on our quadtree. Second, height fields do not render under-cuts. This can be solved by patching together different

height fields where under-cuts occur, and this technique will be discussed. However, it is a complicated task and can not be done efficiently enough for interactive rendering.

4.3 Final Design

The rendering algorithm that we finally settled on satisfies all the design requirements and is fairly simple. Each interval of material is polygonized as a six-sided box, and rendered to the screen as quadrilaterals using openGL. This is done during a depth-first traversal of the quadtree. At each leaf node the list of materials is traversed. The naive algorithm is then to render each solid interval as a six-sided box. The vertices are converted from slab coordinates to global coordinates before output. This technique is very simple but leads to a large number of polygons that are either redundant or not visible, and this can slow rendering. For example, an entire interval could be invisible if there is another interval of material above it and the neighbors are all high enough in the w direction. Also, both neighbors that share a side render that side when really only the higher neighbor need do so. Thus, various techniques are necessary to polygonize in a way that minimizes geometry.

4.3.1 Culling Unnecessary Geometry

To use a minimal number of polygons, we add more information to the nodes of the quadtree. Each node of the quadtree stores the minimum height of material in the North, East, South, and West directions. For a leaf node this number is simply the w value of the first interval in the material list. For non-leaf nodes, the minimum height in a certain direction is the minimum of the the minimum height of the children in that direction (for example, the NE and NW children to compute the N minimum height). This information can be easily computed during the depth-first filling of a slab, and must be maintained after any tooling. The information is relevant because it informs neighbor quadtrees that any polygons below the specified height in that direction will be obscured. For example, if a quadtree finds that its North neighbor has a minimum height of 0.3, this quadtree must polygonize at least as far down as

.3 on the North side. Any polygons beneath that will be obscured. This information adds four floats and thus sixteen bytes to each node of the quadtree; but the increase in rendering speed from saved geometry justifies the memory expenditure.

So, to polygonize a leaf node the material list is traversed until it ends or the w coordinate is below the minimum height in all directions. The top face of the top interval is output. Then, each solid interval outputs the side faces in all four directions (the top or bottom of the box is not necessary). The side faces either descend to the next interval or to the minimum height of the neighbor in their direction. For undercut empty intervals, the top and bottom faces must be drawn, and they must face inwards. When a neighbor in a certain direction does not exist (along the fringes of edge slabs), the minimum height used is 0. In this fashion there is no unnecessary geometry; every part of every polygon is visible from some angle at an external location.

4.3.2 Quadtree Traversal with Neighbors

To use this technique quadtree nodes must be able to communicate with their neighbors. A fast procedure exists to traverse the tree and find a neighbor in any direction [36], but in the worst case it takes on the order of the depth of the tree. We use this procedure in the system (slightly modified to work across different slabs, using the neighbor information stored within each slab), but it is not the best method for this traversal. Since each leaf node needs to talk with all four neighbors during our rendering algorithm, this procedure consumes too much time. Instead, a tree traversal algorithm is used that passes a list of pointers to the eight neighbors to each node as a parameter. The list can be calculated in constant time. This top-down traversal with neighbors is a very useful algorithm, and will be used several times throughout the system. It is implemented directly from [34], modified to work across slabs.

4.3.3 OpenGL Rendering

Finally, openGL calls are used to actually render the polygons. To achieve maximum speed we keep an openGL display list for each slab. During rendering, we simply call the display lists for the slabs that must be rendered; this saves the time necessary to

copy the vertex information from the client to the openGL server. This technique, though, requires that openGL store all the vertex information. This consumes a significant amount of memory, but is justified by the large speed increase in rendering. We also use double-buffering to eliminate flickering. When an event occurs that requires the graphics to be updated, the display lists are rendered into the back buffer, and then the back buffer is copied to the front buffer. A final optimization culls slabs that are outside of the viewing frustum. We use the `gluProject()` [22] command to project the corners of each slab into window coordinates to create a 2D bounding box for the projection of the slab. Then, if the entire slab is outside the render window, the slab is not drawn.

4.3.4 Results

A simple wireframe model of our meshing algorithm can be seen in Figure 2-6 and 4-1. A more complicated result achieved after interactive sculpting is shown in Figure 4-2. Note that the results of this meshing algorithm are very aliased; in fact there are only six different normals across all the polygons (unless there is distortion). If drawn in this way, the shading quality will be unacceptable. Instead, a normal is calculated for each vertex using information from the data structure. This shading helps fool the viewer into not noticing the stair-step nature of the mesh, a problem that will be discussed in Section 4.4.

4.3.5 Further Optimizations

Analysis of the performance of this rendering system reveals that the largest bottleneck is the transmission of data from the client side to the openGL server. The problem is that many of the vertices are called several times, and each time they are called six floats (three floats for location, three floats for a normal) are copied to the server. A fruitful area of future research would be to figure out ways to polygonize our mesh while calling each vertex only once. Vertex arrays, a new feature in openGL 1.1 [44], may be another way to accomplish this.

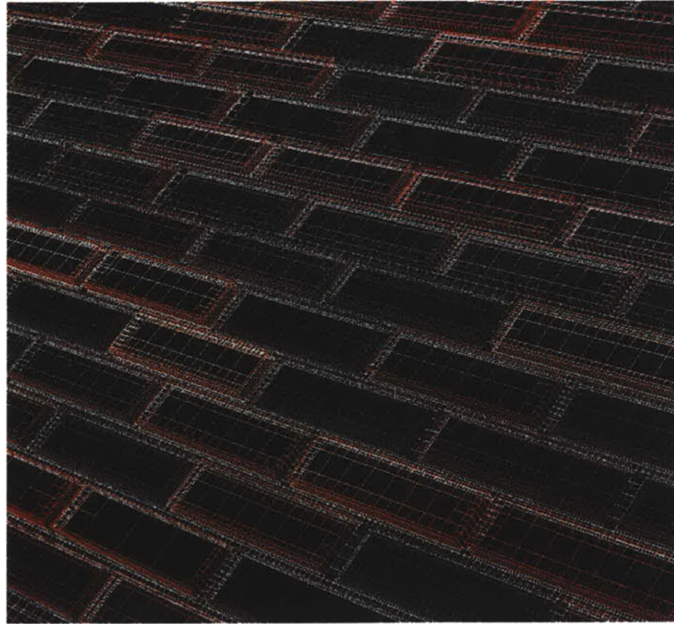


Figure 4-1: Meshing algorithm on untouched brick wall.

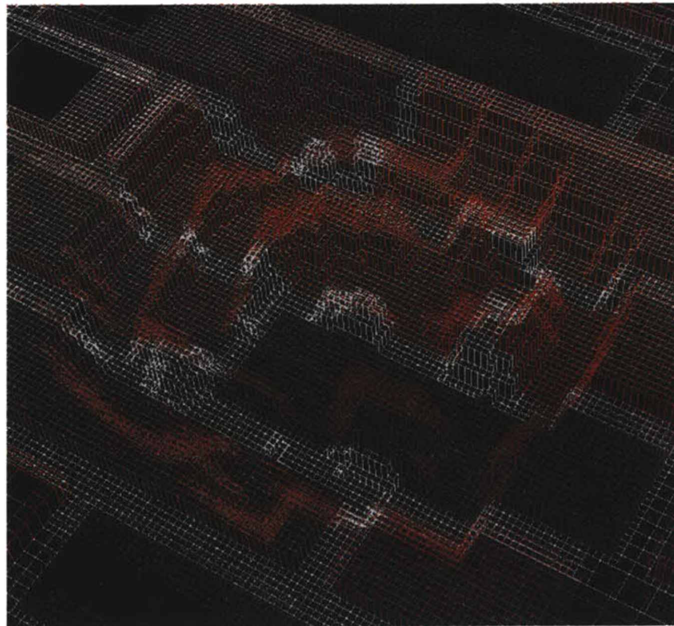


Figure 4-2: Meshing algorithm on a sculpted section of brick.

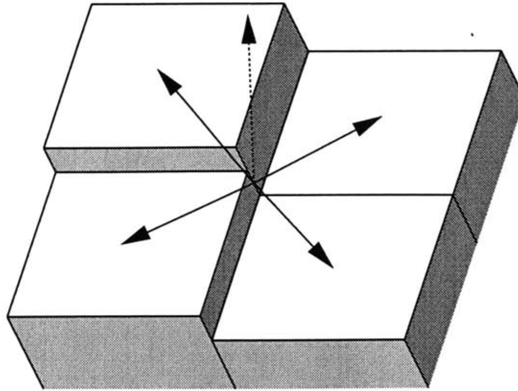


Figure 4-3: Calculation of normals.

4.4 Shading

The generation of attractive normals is crucial to attractive rendering; using normals per face that are calculated directly from our generated mesh would not be satisfactory. However, our data structure does not have a clear definition of normals as there is for a discrete grid; for a discrete grid with floating point densities, a normal at a point can be defined as the density gradient at that point. In our case, since there is no clearly defined notion of normals, an approximation must be found.

The general idea is depicted in Figure 4-3. Here, we see the top materials for four adjacent leaf nodes. Although these leaf nodes may not share the same parent in the quadtree, for our purposes, we can refer to them as the NW, NE, SE, and SW nodes. The NE and SE nodes share their w heights, while the NW and SW differ slightly. We want to calculate all the normals at the corners of quadtree nodes, not at the centers. So, we can calculate one normal from the four nodes shown. To do so we calculate a center point that averages the heights of the adjacent nodes. We then define vectors from this point to the centers of the tops of the adjacent nodes, as depicted by the arrows. To get normal vectors we take the cross-product of adjacent vectors. This gives us four vectors: NE$\times$NW, NW$\times$SW, SW$\times$SE, and SE$\times$NE. We define the normal at this corner as the average of these four cross-product vectors. In the diagram it is depicted as a dotted line.

4.4.1 Coordinate System Transformations

The remaining question is the use of coordinate systems. The obvious method is to compute this normal in slab coordinates, and then convert the normal to world coordinates. The problem is that normals do not transform the same way that points do. The mathematically correct way to transform a normal is to multiply the normal by the transpose of the Jacobian of the tri-linear interpolation. The Jacobian matrix is a 3×3 matrix of partial derivatives describing how slab coordinates change with respect to world coordinates. This matrix is straightforward to calculate, but its value changes from location to location within the slab. This makes the transformation of normals too expensive for interactive rendering.

Next, we attempted to use a linear approximation of this transformation. In this technique, two points are transformed to world coordinates: the center point and a point slightly displaced in the direction of the normal. The subtraction of these two points in world space should be the transformed normal. This approximation worked fine when slabs were not distorted, but caused significant visual artifacts when they were.

Finally, it was decided to do the entire calculation of the normal in world space. Each point used in the calculation was transformed to world coordinates first, and the cross-products were done in this space. This worked fine for both undistorted and distorted slabs. The disadvantage is that five coordinate transformations are necessary to calculate each normal.

4.4.2 T-joints

A *T-joint* in a quadtree occurs when a leaf node neighbors two smaller leaf nodes. An example is shown in Figure 4-4, where the T-joint is indicated by a red circle. T-joints cause problems throughout the system, and one example is shading. The problem occurs when the normal calculated at $\mathbf{b}$ is not the average of the normals at $\mathbf{a}$ and $\mathbf{c}$. This causes a distinct visual artifact as the eye travels left to right across the T-joint. To solve this problem, T-joints are detected during the calculation of normals and handled as a special case. The normal at $\mathbf{b}$ is set to the average of the normals at $\mathbf{a}$ and $\mathbf{c}$.

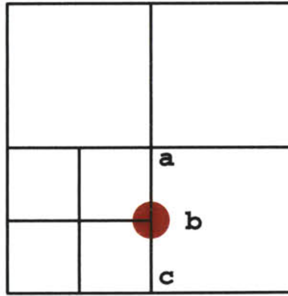


Figure 4-4: T-joint problems in shading.

4.4.3 Single Corner Traversal

The calculation of normals occurs during a traversal of the quadtree. However, the normals are calculated at the corners of the quadtree nodes. The naive algorithm would just calculate the normal for each corner at each leaf node. However, this would mean each normal would be calculated four times (except at T-joints, where it would be calculated two times). To be efficient we need to design a traversal algorithm that touches each corner of a quadtree leaf node exactly once, and it also has to work across slabs as well. To do so we start with a quadtree traversal with neighbors (see section 4.3.2). This passes an array `A` with eight members pointing to neighbors. We then calculate normals in a top-down fashion, as depicted in the following pseudo-code. This procedure is started by calling it on the root node, and then proceeds recursively. The procedure `TouchCorner(Direction D)` calculates a normal at corner D. The procedure `ChildType()` tells us whether this quadtree node is the NW, NE, SW, or SE child of its parent. Depth is 0 at the root node, and increases as we descend. `GetChild(Direction D)` gives us the child in direction D (out of NW,NE,SW,SE). `BuildNeighbors(Neighbors A, Direction D)` calculates a new neighbor array from the current neighbor array and the direction we are traversing.

```
Traverse(Neighbors A) {
    IF (depth==0) {
        TouchCorner(NE);
        IF (A[W]==NULL) TouchCorner(NW);
        IF (A[S]==NULL) TouchCorner(SE);
        IF (A[SW]==NULL) TouchCorner(SW);
    }
    ELSE {
        IF (ChildType()==NE) {
                TouchCorner(NW);
```

```
                TouchCorner(SE);
        }
        IF (ChildType()==SE & (A[S]==NULL | A[S]->getDepth() < depth))
            TouchCorner(SW);
        IF (ChildType()==SW & (A[W]==NULL | A[W]->getDepth() < depth))
            TouchCorner(NW);
    }
    IF (this is a non-leaf node) {
        GetChild(SW)->TouchCorner(NE);
        GetChild(NW)->Traverse(BuildNeighbors(A,NW));
        GetChild(NE)->Traverse(BuildNeighbors(A,NE));
        GetChild(SW)->Traverse(BuildNeighbors(A,SW));
        GetChild(SE)->Traverse(BuildNeighbors(A,SE));
    }
}
```

This algorithm works as follows. At the root level (depth=0) each slab touches its NE corner. The other corners are touched if the neighboring slab that was supposed to take care of it does not exist. The next chunk of code takes care of the side corners, as shown in Figure 4-5. If there are no null neighbors and no T-joints, all side corners should be taken care of by having the NE child touch its NW and SE corners. However, if the S neighbor is absent or is a T-joint (detected by checking if its depth is less than ours), the SE child must touch its SW corner. This is indicated in the figure by the dotted arrow. Likewise, if the W neighbor is absent or a T-joint, the SW child must touch its NW corner. Finally, if this is not a leaf node, we touch the center corner between our four children and then traverse the four children. An example of the traversal order is shown in Figure 4-6. Here, arrows from head to tail show the order of the corners traversed. The first corner traversed is the far NE corner, and its tail comes from outside the quadtree to show that it is first. This traversal example assumes the neighboring slabs are null. Note that each corner is touched exactly once.

4.4.4 Normal Storage

The last issue to consider is the efficient storage of normal data. Notice that since normals are at corners rather than centers, each normal is shared by four different quadtree nodes. It would be wasteful of memory to store normals in each quadtree node, and it would more difficult to update after tooling. Instead, all normals are stored in a bucket array that is a static variable of the quadtree class. Each quadtree

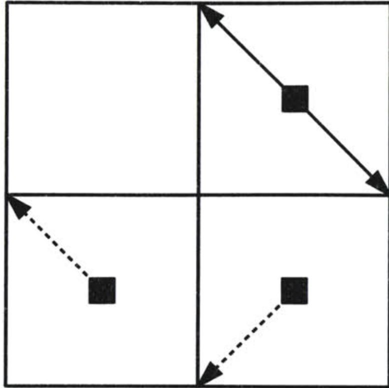


Figure 4-5: Touching of the side corners.

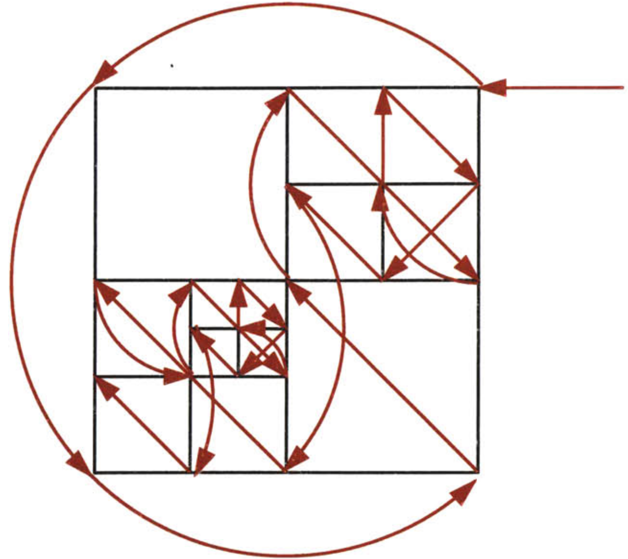


Figure 4-6: Single corner traversal order.

node then stores integer pointers to each of its four corner normals. This cuts memory for normal storage by two-thirds, and makes it easier to update without worrying about consistency.

4.4.5 Results

Normals are initially calculated at each corner in one traversal with neighbors (see Section 4.3.2). They are also recalculated after sculpting, and this is discussed later in Section 5.2.4. Examples of shading can be seen in Figures 4-7 (which is the same brick section as in Figure 4-1) and 4-8 (same as Figure 4-2). Note that this technique does not shade under-cuts perfectly. To do so would add significant processor time to shading, and it is not justified considering that the visual results with this technique are better than expected for under-cuts. Another visual artifact noticeable is that the shading is not uniform across the bricks; some bricks seem more bulgy. This is because of differences in subdivision in the different bricks; the solution is to ensure that the subdivision is fine where big changes occur (such as at the edges of bricks). This will be done for final rendering, but is not expedient for the interactive system. Interactive rendering must favor speed in the tradeoff between quality and efficiency. Final rendering should favor quality, however, and must shade under-cuts better and achieve better uniformity.

Figure 4-7: Shading algorithm on untouched brick wall.

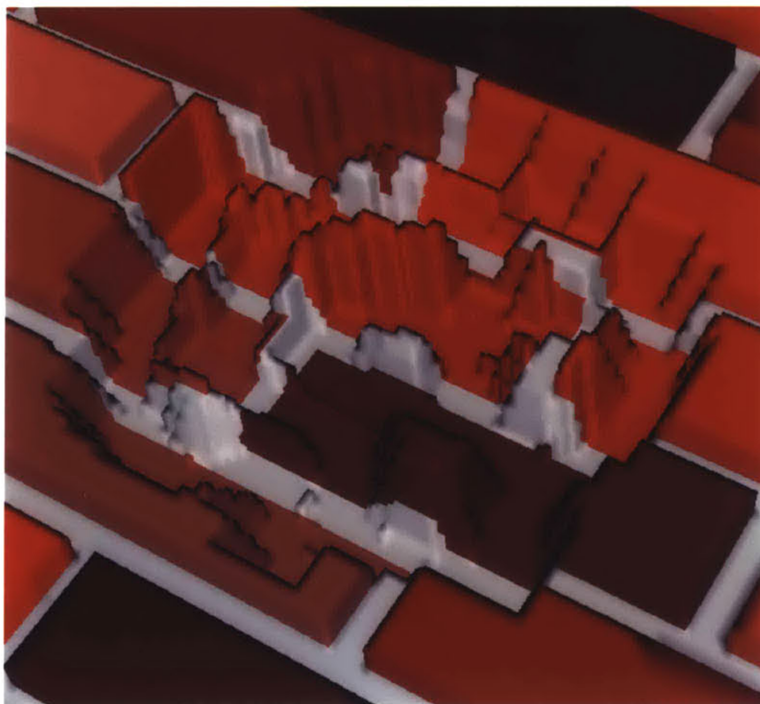


Figure 4-8: Shading algorithm on a sculpted section of brick.

4.5 Model Navigation

Finally, when the user is not doing any sculpting he/she should be able to move the model around to look at the desired parts; this includes rotation, translation, and zoom. At the same time, rendering calls must operate in a constant coordinate system without worrying about user transformations. To do this, two graphics coordinate systems are maintained: the object system and the projection coordinate system. A transformation matrix is kept that converts from object to projection coordinates. When the user uses the mouse to rotate, translate, or zoom, these changes are multiplied into our transformation matrix. To keep in mind the big picture, consider all the transformations that occur when a volumetric surface is drawn. Slab coordinate are converted to object coordinates, then to projection coordinates, and then passed to openGL, where several more transformations occur before reaching the screen.

Also note that to maintain maximum efficiency we cull slabs that are outside of the viewport when rendering images during model navigation. We do this by projecting the eight corners of each slab to window coordinates. If the slab is entirely outside the viewport we do not traverse its quadtree. This speeds up rendering considerably when the user is zoomed in.

Chapter 5

Interactive Sculpting

Now that we can fill our slabs and interactively render them, the next step is to allow the interactive sculpting of volumetric surfaces. There are many components to such a system. The first part is the ability to move a tool and render the part of the graphical output that needs to be updated. It is important for speed to only redraw the section of the output that is necessary. Next, it is necessary to modify the volumetric surface geometry by the tool. Finally, normals must be recalculated.

5.1 Graphics Updating

If the entire scene were redrawn when a sculpting tool moved, sculpting would be very slow. It is therefore necessary to update only the portion of the screen necessary. This turns out to be much more complicated than expected.

The tool is moved with the mouse, but this is difficult since the mouse is a 2D input device moving an object in 3D. To accomplish this, normal mouse movements move the tool in a 2D plane parallel to the screen. When the middle mouse button is held down, mouse movements move the tool in the depth direction. Mouse movements must affect the tool in projection coordinates rather than object coordinates (see Section 4.5) to maintain intuitive, screen aligned movements. At the same time, the tool must affect the geometry in object space.

We have designed a set of steps that accomplish our objective. To begin, when a sculpting tool is activated we make sure an accurate rendering of the entire scene

exists in both the front and back buffer. This sets up the initial conditions to a series of events that are executed each time the tool is moved.

1. Three dimensional coordinates are calculated for the new mouse position. This is done by using `gluProject()` to project the mouse into projection coordinates.

2. A 3D bounding box in object coordinates is calculated that encloses the tool at its previous position and its new position. The projection of this box to the screen should enclose the area that must be updated.

3. This 3D bounding box must be transformed to projection space so that it is aligned with the screen again. To do this, the box is transformed by the object→projection matrix (see Section 4.5). A fast algorithm for transforming axis-aligned bounding boxes is presented in [4], and we implemented this algorithm directly.

4. The front face of this bounding box is projected into the screen and stored for later use. This information gives us a 2D bounding box in window coordinates that tightly encloses the area where updating might occur.

5. Four vectors are calculated from the eye point through the corners of the front face of the 3D bounding box into the scene. These four vectors delineate the edges of a frustum in space such that any geometry within this space must be redrawn. The naive and simpler alternative to this method (simply redrawing any geometry that intersects the 3D bounding box defined in Step 2, without any of the above coordinate transformations) does not work for several reasons. One problem is occlusion; editing the geometry of one slab may reveal geometry in another that does intersect the bounding box. This revealed geometry must be redrawn. The second problem is tool rendering: the movement may reveal geometry that was previously occluded by the drawing of the tool. Therefore, the containing geometry that includes any geometry that must be drawn is a frustum from the eye point into the scene, where the frustum edges travel through the corners of the front face of the 3D bounding box calculated in Step 3.

6. This entire frustum is transformed from projection space into object space.

This is done by multiplying the four frustum vectors by the inverse of the object→projection matrix.

7. Equations of the four planes of the frustum are calculated using the frustum vectors in the format $Ax + By + Cz = D$.

8. We determine which slabs need to be redrawn by testing each slab against the frustum. A plane test is conducted by taking the dot product of all eight corners with the (A, B, C) vector of one of the planes of the frustum. If all eight dot products are greater than D, then the slab falls outside the frustum and does not need to be redrawn. Otherwise, the eight corners are tested against the other three planes. If the slab passes all four plane tests, it must be redrawn, and is marked as such.

9. At this point the geometry of the slabs is edited. This task is discussed in depth in Section 5.2.

10. The back buffer should contain the last rendering done in the previous execution of these steps. The front and back buffers are now swapped, making this rendering visible to the user.

11. The back buffer should now contain the previous contents of the front buffer. The goal is to update the back buffer with the contents of the front buffer. To do so, we will copy the front buffer into the back buffer. However, for maximum efficiency, we wish to copy the smallest region possible. This region in window coordinates was calculated in Step 4. This section of the front buffer is copied into the back.

12. Draw the tool geometry directly into the front buffer. This could in theory lead to flickering, but since the tool geometry is very simple it is drawn faster than the monitor refresh rate and does not flicker.

13. The slabs that were marked for redraw in Step 8 are redrawn directly into the back buffer.

At this point the front buffer should be an image viewable to the user that contains a correct rendering of the geometry and tool, though the geometry will be one step

behind. The back buffer will contain an image of correctly rendered geometry that is the current state, without any rendering of the tool. These conditions are appropriate initial conditions for the next execution of steps when the tool moves again. The fact that the viewable image is one step behind does not cause a problem. This is because tool movement is generally continuous, and the differential in tool position is small.

This series of steps is unfortunately complicated, but is the only method that will guarantee correct rendering including any occlusions while minimizing the amount of geometry rendered and maximizing efficiency. Other, simpler techniques were attempted, but they either failed to be correct under strange occlusion situations, or were very slow.

5.2 Tool Editing of Geometry

The next step is to enable the tool to modify the geometry of our volumetric surfaces. The general idea is to subdivide the quadtrees within the region affected by the tool to a certain resolution. Then, the actual geometry is subtracted from the material lists in the quadtree leaf nodes.

5.2.1 Tools

In our system there are currently only two tools: a sphere and a chisel. It is easy to add more, though. There are several requirements for the satisfactory definition of a tool.

1. Each tool must know the maximum depth that it wants the quadtree to subdivide to (or a resolution in millimeters that it edit at).

2. Each tool must have a bounding sphere, expressed as a radius in millimeters.

3. The geometry of the tool must be convex. As such it must be able to return 0, 1, or 2 intersection points that are the intersection of a line segment and the tool.

4. The tool must be able to draw itself to the screen.

5. The tool must know its normal at any surface point (this is used in Section 5.2.4).

5.2.2 Tool Modification Steps

In Step 9 in the graphics updating loop above, the process of editing geometry by the tool is started. This process occurs in another series of steps.

Intersecting Slabs

The first step is to intersect the bounding sphere of the tool with the axis-aligned bounding box of each slab. This is done quickly using the algorithm presented in [3], which finds the point in the bounding box that is closest to the sphere.

If no slabs are intersected, we are done. Otherwise, we should have one or perhaps a handful of intersected slabs. We iterate over these. The following is done for each such slab.

Entering Slab Space

We have already found the closest point in the bounding box of the slab to the center of the sphere. This location will either be the center of the sphere (if the center of the sphere is in the bounding box), or the point in the bounding box that is furthest inside the sphere. We convert this point to slab coordinates (see Section 2.1.1). This point may not return valid slab coordinates (valid is in the range $[0,1]$), since not all points in the slab bounding box are actually within the slab coordinate system. In this case the slab coordinates are clamped to the valid range. This results in a location in slab space that is closest to the center of the tool bounding sphere. We are not guaranteed that it is in the bounding sphere, however. Also, in the case of distortion this point becomes only an approximation to the closest point within the tool.

Traversing the Quadtree

We now traverse the quadtree. For each non-leaf node traversed, we find the closest point in the geometry of the node to the center of the tool sphere (in slab space a quadtree node is shaped like a box. So, we can use [3] again, as discussed in Section 5.2.2). Note that this is done in slab space, so in the case of distortion the closest point in slab space may not be the closest point in world space. However,

it is a sufficient approximation. If the point is within the tool, we traverse the four children. Otherwise, we no longer explore this section of the tree. Once we reach leaf nodes, we first make sure the leaf node is as deep as the tool resolution calls for. If it is not, we subdivide the node and traverse the children. For leaf nodes that are at the correct resolution and within the tool, we commence to edit their material lists. Note how the hierarchical nature of the data structure allows us to efficiently visit only those nodes that are within the tool. The others are quickly pruned from the search tree before any of their leaf nodes are visited. This would be much more difficult with a regular grid.

Editing Material Lists

The general idea for editing a material list is as follows. The material list exists along a line in space defined by two endpoints (u and v are placed at the center of the node, and w goes between $w = 0$ and $w = 1$). This line is fortunately guaranteed to be straight in both slab and world space. Whichever tool is in use is expected to be able to take these two endpoints converted into world space and return two q values that are linear interpolation values indicating where the line enters and leaves the geometry of the tool (tools are constrained to be convex). The values of q_1 and q_2 lead to five cases.

Tool Intersection Cases

1. $((q_1 < 0) \wedge (q_2 > 1)) \vee ((q_1 > 1) \wedge (q_2 < 0))$
   In this case the entire material list is in the tool. Erase the whole list.

2. $((q_1 < 0) \wedge (q_2 < 0)) \vee ((q_1 > 1) \wedge (q_2 > 1))$
   The tool does not intersect the material list. Leave it as is.

3. $(q_1 > 0) \wedge (q_1 < 1) \wedge (q_2 > 1)$
   The tool intersects the top the material list. In this case, traverse the material list from top down. Remove intervals until we reach q_1 in the list. If the interval containing q_1 is air, do nothing. Otherwise set the top to q_1.

4. $(q_2 > 0) \wedge (q_2 < 1) \wedge (q_1 > 1)$

The tool intersects the material list from behind. Do the same thing as case 3, but backwards.

5. $(q_1 > 0) \wedge (q_1 < 1) \wedge (q_2 > 0) \wedge (q_2 > 1)$
The tool is in the middle of the material list, and thus will create an under-cut. This is the most difficult case to handle. We first find the interval containing q_2. If the interval is solid we clamp it to the value of q_2, and then insert an interval of air. We then traverse the material list until we find q_1, removing intervals along the way. We pull down the start of this interval to the value of q_1, unless it is air in which case we remove the air interval we previously inserted. There are several special cases to handle, such as when the same interval contains q_1 and q_2.

5.2.3 Tool Specifics

The above details work for all tools. There are several capabilities that must be programmed separately for each tool. We explain these inner workings for two tools: a sphere and a chisel.

Spherical Tool

Spherical tools are fairly straightforward. They are defined by a radius, and the bounding sphere has this same radius. To draw the spherical tool we use the function `gluSphere()` [22]. Intersection tests are done by calculating the distance of the point to the center of the sphere, and comparing this to the radius. The normal at a surface point is simply a normalized vector from the center of the sphere to the surface point. Finally, we must be able to intersect the sphere with a line defined by two endpoints and return two linear interpolation values.

Given the center of the spherical tool $\vec{C}$, two endpoints of a line $\vec{P_1}$ and $\vec{P_2}$, a point of intersection $\vec{L}$, and the spherical radius r, the following vector formulas define a sphere and linear interpolation result q.

$$(\vec{C} - \vec{L})^2 = r^2$$

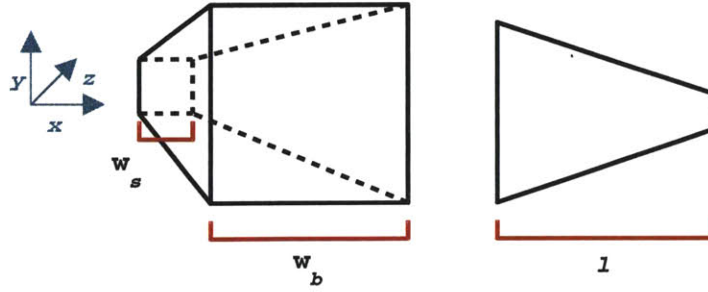


Figure 5-1: Two views of a chisel.

$$\vec{L} = \vec{P_1} + q(\vec{P_2} - \vec{P_1})$$

After significant algebra to eliminate $\vec{L}$ and simplify, these formulas reduce to the following scalar quadratic equation.

$$((\vec{P_2} - \vec{P_1})^2)q^2 + 2[(\vec{P_1} - \vec{C}) \bullet (\vec{P_2} - \vec{P_1})]q + (\vec{P_1} - \vec{C})^2 - r^2 = 0$$

It is first important to take the discriminant of this quadratic. If it is less than 0 the line does not intersect the sphere. Otherwise the quadratic formula should yield values q_1 and q_2.

Chisel Tool

The chisel tool is more complicated. A chisel in our system is a truncated pyramid and is composed of six sides. The two end sides are constrained to be square. Therefore a chisel is defined by three floats specifying the length of a chisel (l), the width of the big side (w_b), and the width of the small side (w_s), as shown in Figure 5-1. The normals to the chisel are the normals to the planes making up the sides of the chisel. Note that since the chisel should be perpendicular to the screen it is always aligned with the projection coordinate system such that the length of the chisel runs parallel to the z axis. This makes calculations easier.

The bounding radius of the tool is the length of a line from the center of the chisel to a corner of the big side, which is $l^2/4 + w_b^2/2$. To draw the tool, 12 lines are drawn depicting the outline of the tool. Intersection tests are done in three stages for maximum efficiency. First the point is tested against the bounding sphere. After this test is passed, the point is converted to projection space (which the tool is always

axis-aligned with), and then translated so that the origin is the center of the tool. The second test checks that the z coordinate is in the range $(-l/2, l/2)$. We then calculate the width w_m that is the width of the square slice of the tool that contains the test point. The third test checks that the x and y coordinates are in the range $(-w_m/2, w_m/2)$. This three-stage intersection is more complicated than it needs to be, but is significantly faster than the naive alternative.

Finally, we must be able to intersect an arbitrary line with the chisel. Both endpoints are converted to projection space, and then translated so the origin is the center of the tool. We first throw out cases where the entire line is outside of the chisel bounding box. Then, the line is intersected with each of the six planes of the tool sides. The normals to the planes can tell us whether the intersection point is an entry or exit point. Note that since we are testing against entire planes there can be spurious intersections. However, some simple case handling can eliminate these. The found intersection points are then converted back to object space.

5.2.4 Modifying Normals

After the tool has edited the geometry we need to recalculate normals. This is a tricky task, as calculating a correct normal requires knowledge of neighboring quadtree nodes. We therefore cannot do it in the same quadtree traversal as the editing of geometry, since neighbors might not have been updated yet. Instead, a second traversal of the quadtree is necessary. However, making two traversals of the quadtree at each motion of the tool is prohibitively slow. So, we instead recalculate normals as a batch job after the user lifts the mouse button, indicating that the tool stroke is over.

However, this means that the user will have poor visual cues while making a tool stroke, since shading helps the user to see depth. Therefore, we make a guess at the new normals during the original traversal of the quadtree. This guess is formed not from the edited geometry but from the shape of the tool itself. The tool is asked for its normal at the entering intersection point of the quadtree node material list and the tool. This normal is reversed and assigned. Such a shading method gives a decent approximation to the appropriate shading of the edited geometry.

While the user is performing a tool stroke, the system keeps track of which slabs

have been touched. When the user finishes a stroke, the system traverses the quadtrees of the touched slabs and recalculates the normals using the same algorithm as in Section 4.4. This provides better shading, and can be done quickly enough to not annoy the user.

5.3 Results

The results of a spherical tool stroke in both wireframe and shaded renderings can be seen in Figures 5-2 and 5-3. Note the subdivision level where the tool has touched the solid materials.

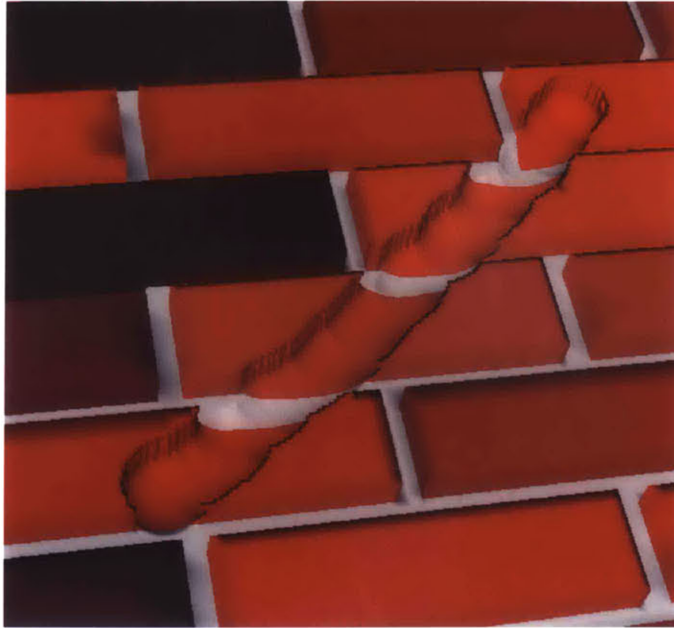


Figure 5-2: Wireframe model of a sphere cut.

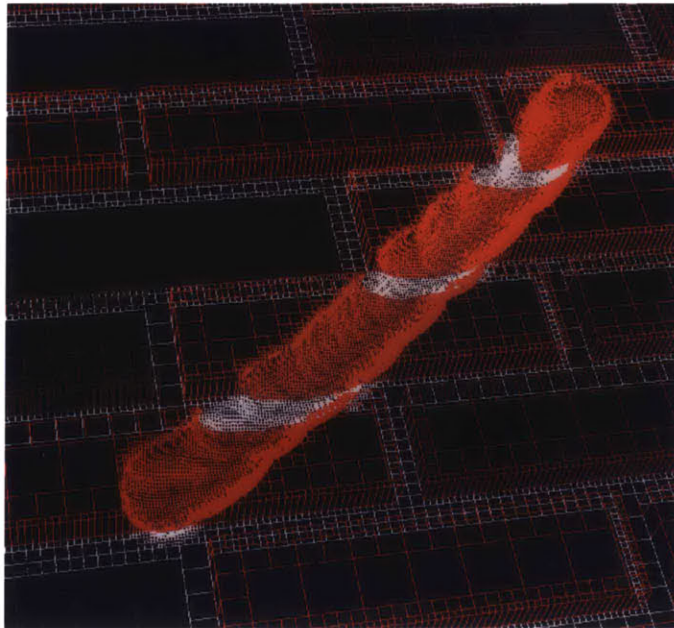


Figure 5-3: Shaded model of the same sphere cut.

Chapter 6

Final Rendering

After a user is finished sculpting erosion effects, it is desirable to be able to create a high-resolution, high-quality surface mesh from the sculpted volumetric surfaces. Such a surface could be imported into a user's animation, or ray-traced to create a photo-realistic image.

The first impulse is to use the same mesh generated for interactive rendering, as discussed in Chapter 4. However, the design requirements of a mesh for final rendering are quite different than those for interactive rendering. Speed is no longer a consideration; rendering can take much longer since there are no interactive requirements. Instead, high-quality is the driving motivation. We also do not need our mesh to be locally updateable; no further modification will occur.

There are several problems with our interactive meshing technique that make it unsuitable for final rendering. For one, since the mesh is extremely aliased the silhouettes are unacceptable. Second, because of differences in subdivision levels there is a lack of uniformity in shading; two identical bricks can look different in our interactive techniques. And finally, our interactive technique does not shade under-cuts correctly. This is very noticeable for significantly large under-cuts.

Instead we need a new meshing algorithm, and there are several alternatives for such an algorithm.

6.1 Design Alternatives

6.1.1 Marching Cubes

We considered using marching cubes in Section 4.2.3. Marching cubes works on a discrete, volumetric grid dataset, which we do not have. However, such a grid could be made from our volumetric surfaces through a form of volumetric scan conversion. This option was rejected for interactive techniques because such a level of data indirection would significantly slow rendering; this is no longer such a problem here. Problems do arise with this technique, however. First, it is hard to generate such a discrete grid that will not exhibit contouring problems when a marching cubes surface is constructed. Second, the discrete grid generated would have to be uniform at the resolution of the deepest subdivison. Hierarchical volumetric surfaces are designed so that whole buildings can be stored in memory by keeping uniform areas at very coarse resolution. Such coarse resolution can coexist with detail that exists at sub-millimeter resolution. If such an entire building were forced to exist at sub-millimeter resolution the size of the polygonal data would become prohibitive even for batch ray-tracing.

A possible solution to this would be to use some sort of adaptive marching cubes, as discussed in Section 4.2.3 and [38]. However, such algorithms operate on volumetric data that is stored in a 3D hierarchical structure like an octree. Our data structure is based on a 2D subdivision scheme, and it is not clear that there is a way to convert our data structure into an octree. Since octrees take advantage of different types of geometric coherency, such a conversion might not result in an efficient data structure.

6.1.2 Direct Ray Tracing

It might be possible to directly ray trace our volumetric surfaces using techniques similar to volumetric ray tracing [39]. Such a technique has the potential to render very realistic images of our volumetric surfaces. There are several problems to tackle, however. First, some sort of interpolation method would have to be conceived to eliminate the aliasing that is inherent in the data structure. Second, the process of casting a ray into volumetric surfaces involves converting world coordinates into slab coordinates. As discussed in Section 2.1.1, this is very slow. Third, a straight line

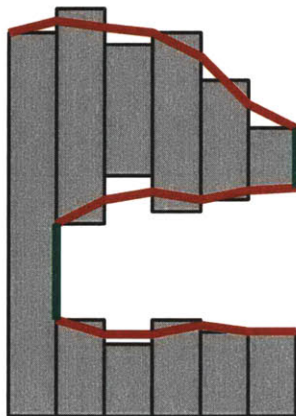


Figure 6-1: a 2D example of patched height fields.

in world space is not necessarily straight in slab space. Therefore, simply finding the entering and exiting intersections of a ray with a slab and then interpolating a straight line between them in slab space would not work for distorted slabs. If these problems can somehow be solved, volumetric surface ray tracing might be an interesting area of future work.

6.2 Final Design

The final design, which was first discussed in Section 4.2.4, uses a hierarchical height field triangulation. We want to take advantage of the fact that volumetric surfaces consist mostly of expanses of height field. The major complications occur where there are under-cuts, and some sort of patching technique must be implemented to patch together height fields that are separated by under-cuts. A drawing that shows our concept for a 2D case is shown in Figure 6-1. Here, three height fields that can be inferred from the geometry are shown with red lines. The green segments show patches that connect the separate height fields.

6.2.1 Restricted Quadtree Triangulation

The triangulation of height fields organized in a quadtree is a well researched field. The main problem is to avoid cracks, which can occur at T-joints. An example of a crack problem is shown in Figure 6-2. A solution to this problem was formulated in [19], and further discussed in [30]. The first step in a restricted quadtree triangula-

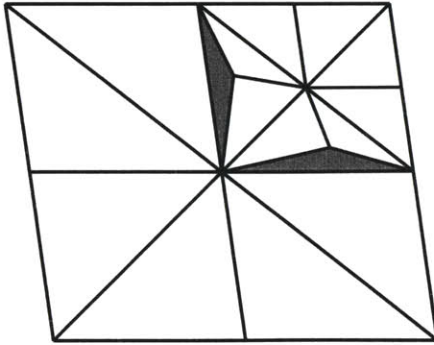


Figure 6-2: An example of a crack problem in quadtree triangulations.

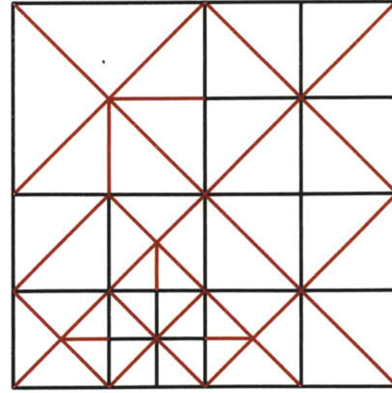


Figure 6-3: A correct restricted quadtree triangulation.

tion is to make the quadtree restricted; a restricted quadtree enforces the constraint that no two neighboring leaf nodes can differ in depth by more than one. To create a restricted quadtree, leaves must be added to the tree. It has been shown in [10] that if the original size of the quadtree is $O(n)$, the restricted quadtree may be larger but is still $O(n)$.

Once a quadtree is restricted, a simple rule can be used to triangulate. For each leaf node, begin by triangulating with two triangles. If any of the neighbors are of greater depth, divide to four triangles. Then, in each direction where a smaller neighbor exists, further divide the triangle in that direction into two. This means that the number of triangles for a leaf node can range from two to eight. This technique can be best depicted by example, and such a triangulation is shown in Figure 6-3. In this figure black lines are those that are both a subdivision line and part of the triangle mesh. A red line is only part of the triangle mesh. Note that the quadtree is restricted, and that no T-joints exist in the triangulation. This solves the problem.

Therefore, to create a restricted quadtree triangulation a restricted copy of the current volumetric surfaces must be created. This is done in two steps.

Making a Restricted Quadtree

An algorithm for making a restricted quadtree copy from an original quadtree was designed in [10]. First, a direct copy of the quadtree is made. While the copy is being made a linked list of all leaf nodes is constructed. The list is then traversed.

Each node is checked against its neighbors in the quadtree, and if the node needs to be split to be balanced with its neighbors, we do so. The children of the newly split node are then added to the end of the list. Also, if any neighbors now need to be split because this split makes the neighbor unbalanced, the neighbor is added to the list. Once the list is exhausted the quadtree should be restricted.

Our new restricted quadtree will be used differently than the original quadtree, and so its material representation should not be the same. We want a leaf node's notion of filled materials to be conducive to the construction of height fields. Thus, instead of a list of material intervals, we construct a list of surface-air intersections. That is, wherever the original material list indicates that a material borders empty space, we construct a surface-air intersection node indicating that a surface needs to exist there. We store the material properties of the intersection, the direction (up or down in the w coordinate) of the intersection, and the height. This eliminates information extraneous to surface construction, such as when two different solid materials border each other in the same material list. No surface should exist there, so the restricted quadtree need not know about it. Note that in most cases there will only be one surface-air intersection for a leaf node. The length of the list will only be greater than one if there is an under-cut. We shall refer to a surface-air intersection as an SAI.

Adding Nodes to Enforce Shading Uniformity

We discussed in Section 4.4.5 that differences in subdivision (stemming from differences in quadtree alignment with the brick pattern) can lead to non-uniformity in the shading of bricks. The problem is shown in Figure 6-4. The normals in the left brick are going to interpolate slower, leading to a bulgy arc-like appearance. In the right brick, the edges will instead be sharper. This is not good, since to the user's eye both bricks should appear identical. Of course, the quadtrees shown are not restricted, and this restriction help some. However, though no longer as severe the non-uniformities still exist in the restricted quadtree.

To solve this problem we need to ensure that edges stay sharp. To do this, we detect edges by detecting changes in the w coordinate that are greater than a certain

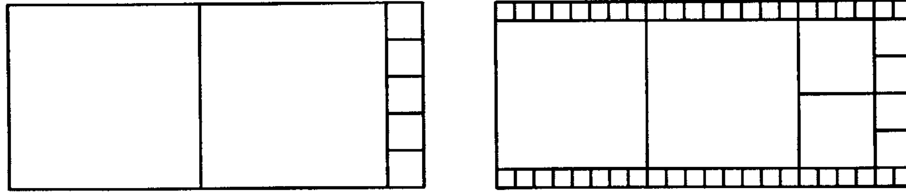


Figure 6-4: Differences in brick subdivision that lead to bad shading.

threshold. The subdivision levels around such edges are then forced to a certain maximum depth. This is done by splitting the nodes during the construction of the restricted quadtree and adding the children and appropriate neighbors to the same linked list. This technique leads to good uniformity in shading.

6.2.2 Growing Height Fields

Now that we have a good restricted quadtree with lists of surface-air intersections we need to construct height fields from them. This is not straightforward since there can be multiple height fields facing different directions. We need to ensure that quadtree nodes that should share a height field do. In effect, we need to grow height fields by having neighboring leaf nodes communicate with each other and decide to share surfaces. In this way we grow connectivity and ensure that connected surfaces grow to their maximum size without jumping under-cuts.

Vertex Joins

The first mechanism we need is a way to latch together the corners of the SAI's in different leaf nodes so that they know that they belong in the same surface. Such a latch is constructed at the corners of quadtrees nodes, and we call them *vertex joins*. A single vertex join is depicted in Figure 6-5. Each vertex join stores four pointers, one for each diagonal direction. Each SAI also stores pointers to vertex joins in eight directions; that is why the arrows in the figure are double-headed. The need for pointers to the NE, NW, SW, and SE vertex joins is clear. We also need pointers to possible N, E, S, and W vertex joins in the case of T-joints. The pointers in the vertex join may be null (though at least one must point to something), and a null pointer indicates that no surface is shared in that direction. Each surface-air

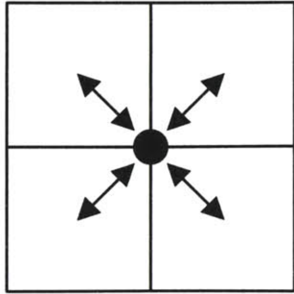


Figure 6-5: The vertex join data structure.

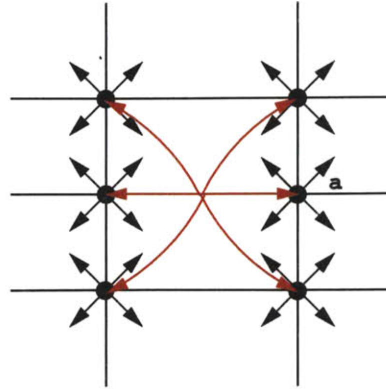


Figure 6-6: An SAI's pointers to vertex joins.

intersection must point to four valid vertex joins in the corner directions. The side directions only need to point if a T-joint exists in that direction. An example for a single SAI is shown in Figure 6-6. Here, the SAI pointers to the six vertex joins are shown in red. The N and S pointers are null, but the W and E are not since T-joints exist in these directions. Notice that the NW and SW pointers in the vertex join labelled a point to the same SAI. This helps us to easily detect T-joints.

Each vertex join stores a height in slab coordinates; this height is the average of the heights of the connected SAI's. This specifies where a surface passing through this corner would be located. Note that if there are under-cuts several vertex joins may exist at different heights at the same corner. Vertex joins are stored in a large bucket array, and SAI's point to them using integer indices. Using this mechanism decisions on the sharing of height fields can be stored entirely within the pointers of these vertex joins. It also makes it possible to walk along a single height field directly without traversing the underlying quadtrees.

Heuristic for Surface Sharing

Before constructing vertex joins we need to be able to decide if two neighboring SAI's share the same surface. The hueristic for doing so is fairly simple. First, the two SAI's must face in the same direction (remember that height fields can face up or down in volumetric surfaces). Second, when travelling from one SAI to the other, no other SAI can get in the way; this constraint eliminates self-intersections. Examples of when SAI's share and when they do not are shown in Figure 6-7. The two examples

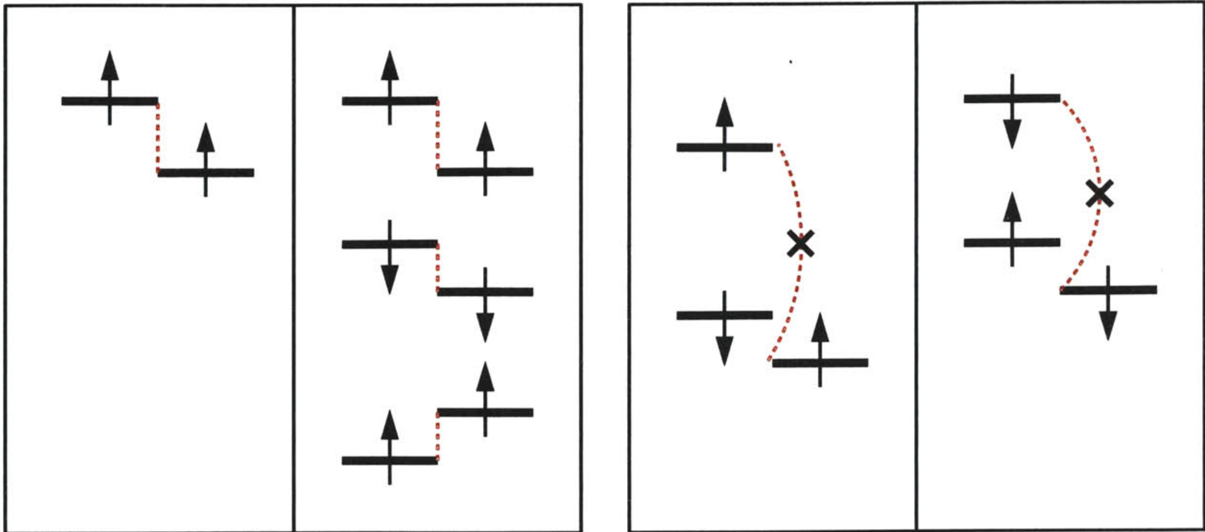


Figure 6-7: Examples of surface sharing for neighboring SAI's.

on the left show SAI's that do share, and the examples on the right are cases where they do not. The dotted red lines indicate the sharing relationships (or lack thereof).

Growing nets of Vertex Joins

Vertex joins exist at the corners of quadtree nodes. To construct vertex joins into nets of height fields we need to traverse each such corner once. Fortunately, we have already devised a single corner traversal algorithm for quadtrees in Section 4.4.3. We use this algorithm again.

At each corner the challenge is to figure out surface sharing. There are four quadtree leaf nodes (NE, NW, SE, and SW), and each leaf node has a list of SAI's. Note that in the case of T-joints, two of the leaf nodes could be the same node. Also, any of the leaf nodes could be null.

During a single corner traversal, each corner that is touched is called by a certain leaf node. We start with the leaf node in this direction, since we are guaranteed that this leaf node actually has a corner at the corner being touched (and thus is not part of the larger node in a T-joint). We traverse the list at this node.

For each SAI we do the following. We ask the node in the CW direction if it has any SAI that should share a surface (as decided by the heuristic in Section 6.2.2). If one is found, we continue in the clockwise direction. We stop either when there

is no sharing SAI, or we have come back to ourself (after four comparisons). The most common case is when there are four SAI's that want to share a surface. In this case, we create a vertex join and set the appropriate pointers. If, instead, we stopped going clockwise because at some point there was a failure to share, we continue by searching counterclockwise from the original node. We do this either until we find a failure again, or we come up upon the node where the last failure was detected. In the case where a node and its neighbor in a certain direction are the same, we have a T-joint. In this case we skip a direction and keep going. We must make a note of this, though, so that we can correctly set its side vertex join pointer.

At this point we can proceed depending upon how many surface sharing agreements we found. If there were two, we are in good shape. This means that the SAI in the direction we started in shares a surface with both its clockwise and counter-clockwise neighbor, but the opposite neighbor is either part of a different surface or on its own. We create a vertex join, set the appropriate pointers, and move on.

If, instead, we found three surface sharing agreements, we have an annoying contradiction. A typical example of this contradiction is shown in Figure 6-8, with sharing agreements shown in red, dotted lines. In this case **a** shares with **b**, **b** shares with **c**, **c** shares with **d**, but **d** does not share with **a** since **e** is in the way. In this case it is not clear how to build a surface. We do know, however, that we want a surface that includes **b** and **c**. We then need to choose to include either **a** or **d** in this surface. We choose the one that adds the lowest overall error to the height of the surface. The other we separate to form its own surface.

Another annoying case can arrive when we have four surface sharing agreements, but they are not consistent. A typical case of this is shown in Figure 6-9. In this case **a** shares with **b**, **b** shares with **c**, **c** shares with **d**, and **d** shares with **f**. This is a problem since we started with **a**, but when we came back to the SE direction we found a different SAI. There is no coherent surface for this situation.

The best way to solve this situation is to assign two different surfaces, one composed of two SAI's and the other composed of three. In this case, one surface will include **a** and **b**, and the other **c**, **d**, and **f**. How do we decide? There are only two possible configurations, so we choose the one that minimizes error in height (when

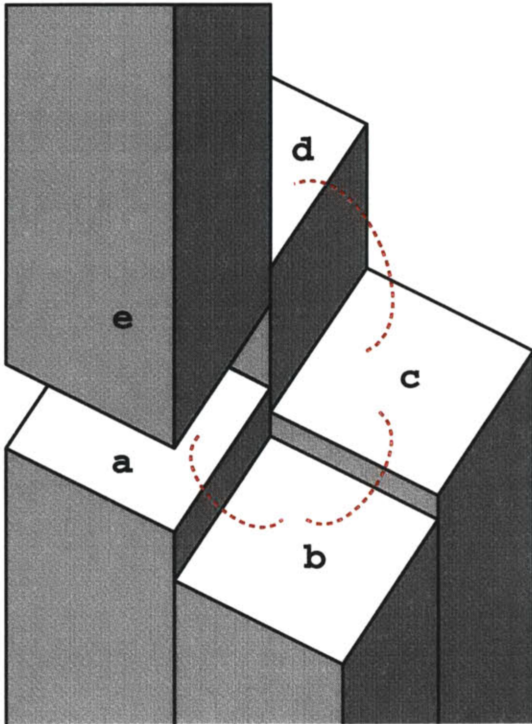


Figure 6-8: A case of three found surface sharing agreements.

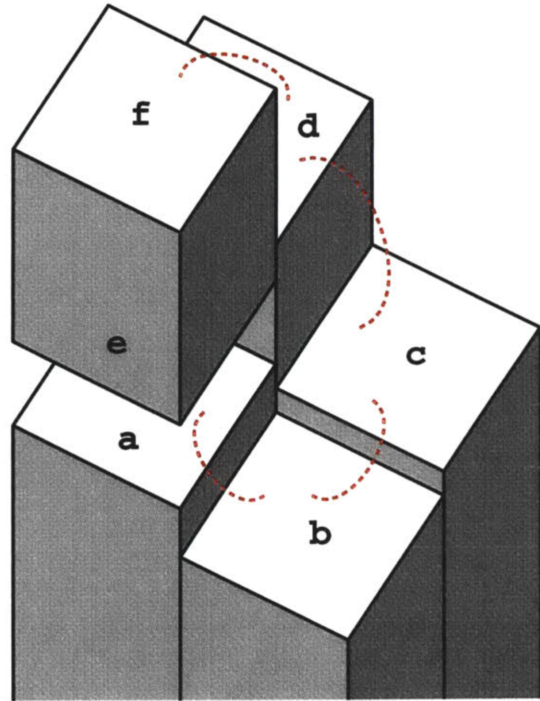


Figure 6-9: A contradicting case of four agreements.

creating a surface vertex at a corner we average the heights of the SAI's included in the surface, so error is the sum of differences from the average).

Note these two cases would become very difficult to handle if there were a T-joint. Fortunately this does not happen since such strange situations only occur from tool operations, which force the area to a common level of subdivision.

After we have traversed the list of SAI's in the initial direction, we continue around the corner and traverse the lists of the other nodes. At this point an SAI may already have its vertex join in the direction of the corner set. If this is the case the SAI is skipped. After we have traversed the four lists of SAI's we should be guaranteed that every SAI will point to a valid vertex join in the direction of the corner. For example, the SAI's in the SW node will all have their NE pointers set.

So, once we finish the single corner traversal all SAI's in all leaf nodes will point to four valid vertex joins. All these vertex joins will have their height set, and so will know their position in both slab and world coordinates. These will form the vertices of the output mesh.

Outputing Height Field Triangles

The height fields are now defined so that we can easily output triangles. The quadtrees are traversed (with neighbors), and the SAI's in each leaf node are traversed. If all the neighbors are the same size or bigger, we output two triangles. Otherwise, we add a vertex for the center point of the SAI. Then, for each neighbor that is the same size or larger, we output one triangle in the neighbor's direction. For the smaller neighbors we output two triangles, querying the neighbor for the middle point of that side.

6.2.3 Patching Height Fields

At this point we have defined height fields that stretch as far as possible across the volumetric surfaces. However, in the many cases where height fields break, we need to patch up the holes between them. In most cases, we do not need to add vertices to patch; we instead need to stretch triangles between the fringe vertices of adjacent height fields. There is also a clear indication of when patching needs to occur. Patching is needed whenever a pointer to an SAI in a vertex join is null.

For simplification, patching can be divided into three cases.

1. The first case occurs when there is a downward facing SAI and two vertex joins have a null pointer in a certain direction. An example is shown in Figure 6-10. Here, for the downward SAI labeled a, the SE pointer in the NE vertex join and the NE pointer in the SE vertex join are null (indicated by the crossed-out pointer arrows), and so we know the E side must be patched. Since this is a downward SAI, the patch will cover an empty interval in our material list. This should only be done if the neighbor on the side in question is solid over this interval, which is the case in our example. We thus stretch a patch on the side in question from the downwards SAI to the next SAI. This is indicated in the figure by a red patch, with an arrow showing the direction of the polygon. If there is no next SAI, the patch must be extended to $w = 0$. To do this, we need to add vertices at $w = 0$, which is straightforward.

2. The second case is very similar to the first, except that it handles upwards SAI's.

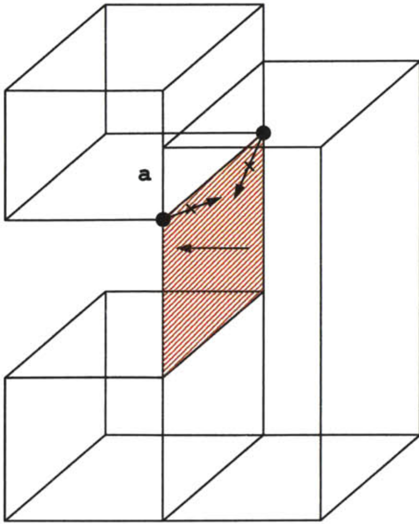


Figure 6-10: Patch case one.

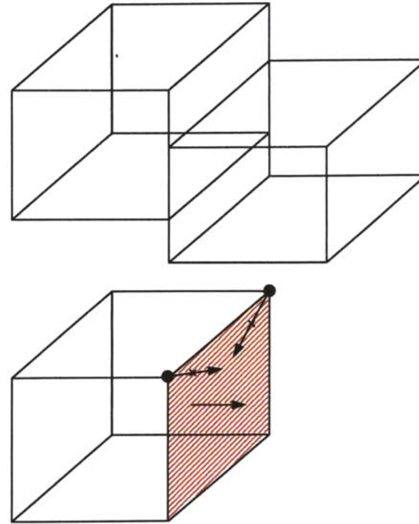


Figure 6-11: Patch case two.

In this case, the patch would cover a solid interval neighboring an empty one. An example is shown in Figure 6-11. In this case, the upward SAI has the vertex joins with null pointers as shown. Hence, a patch is necessary. Since there is no next SAI, the patch must be extended all the way down to $w = 0$ (where new vertices are created).

3. The third case is more complicated and only arises from some of the strange contradictory cases handled in Section 6.2.2. An example of this type of patch is shown in Figure 6-12. This patch arises at the junction of two SAI's that would normally share. In the figure, the SAI's share through the vertex join labelled b. However, due to a contradiction case as discussed earlier, their sharing agreement had to be terminated at the corner labelled a. The two SAI's thus have separate vertex joins at this corner. In the figure, the blue polygons are height field polygons. The single triangle shown in red patches the hole that arises from this situation. To detect such a situation, it is sufficient to detect a mismatch between the vertex join pointers. In this case, from the left SAI perspective the SE pointer of the NE vertex join is not null, while the NE pointer of the SE vertex join is null. Also, care must be taken that this patch is not handled twice, as both the left and right SAI's would detect this situation during traversal. To ensure this an SAI only adds a patch of this type if it is higher than its neighbor

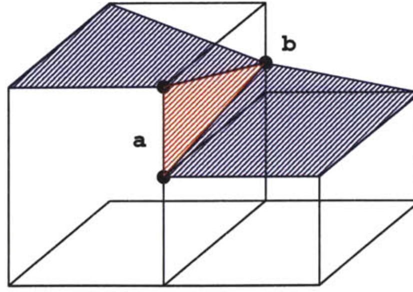


Figure 6-12: Patch case three.

in the direction the patch is to be added.

Patches are added during an additional traversal of the quadtrees with neighbors. All quadtree leaf nodes look in all four directions and add patches as necessary. Since this means each boundary between quadtree nodes is examined twice for patching, care must be taken so that redundant polygons do not occur. Also, if a larger node is neighboring a smaller one (T-joint), two patches may be necessary. In this case, side (as opposed to corner) vertex joins are examined.

6.2.4 Shading

If the resultant triangles are rendered with flat shading, significant aliasing can result. Instead, it is better to calculate normals at the corners of triangles and then smoothly interpolate these normals over the triangles during ray tracing. There are several ways to do this. One obvious way to do this would be to calculate normals using the techniques used in interactive rendering (Section 4.4). However, this failed to give attractive results.

Better results were achieved by ignoring the volumetric surfaces and examining the resulting triangles. The normal at each vertex of the mesh is set to the average of the normals of the adjacent triangles. This achieves shading that is faithful to the mesh being rendered while still achieving smooth shading.

Simple shaders were used to achieve realistic-looking materials. The brick shader involves subtle variation of color across each brick using a noise function. The mortar shader uses a high-frequency bump map to give it a rough appearance. Plaster uses a low-frequency bump map.

6.3 Results

Results of the meshing algorithm can be seen in Figures 6-13 and 6-14. These images show wireframe renderings of the mesh generated by the algorithm. The first figure shows an area that has been sculpted by a chisel tool. The second figure shows fresh brick. Examples of both hierarchical height fields and patching can be seen. Shaded images are presented in Chapter 7.

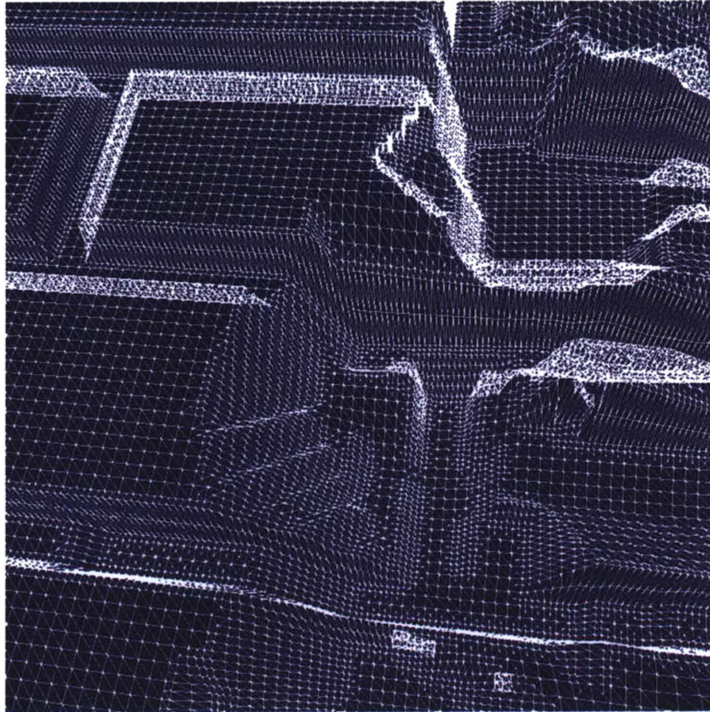


Figure 6-13: An example of final meshing on a tooled area.

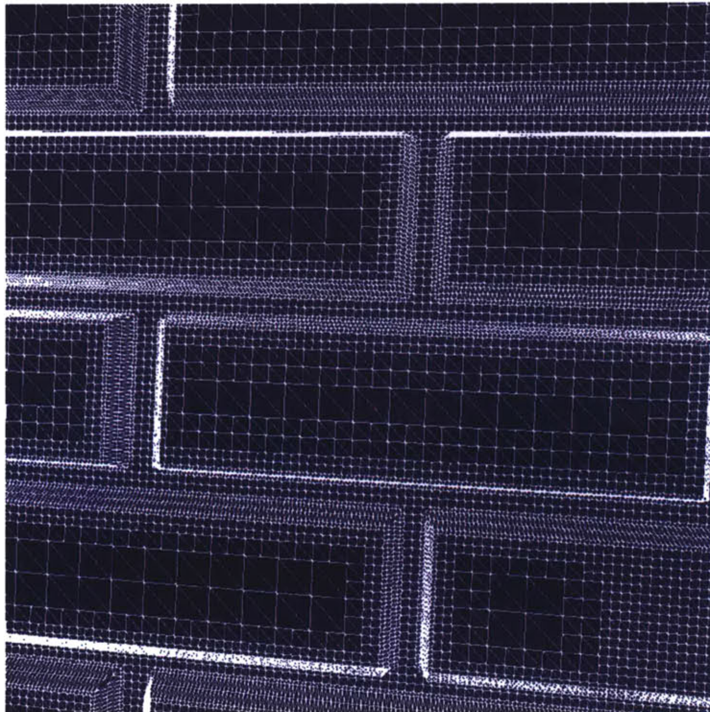


Figure 6-14: Annother example of final meshing.

Chapter 7

Discussion and Conclusion

Now that the fundamental algorithms have been described it is possible to examine final results and discuss both the successes of the system as well as areas for future work.

7.1 Shaded Results

To demonstrate the efficacy of the final system we have generated three examples of eroded appearances that were interactively sculpted with our system. In all the examples we show both the interactive rendering after tooling as well as a shaded, ray-traced image of the final mesh.

7.1.1 Crumbled Brick

The first example depicts a close-up of a crumbling brick wall. Figure 7-1 shows the interactive rendering, and Figure 7-2 shows the final rendering. The sculpting was done entirely with a blunt chisel tool. The light source used during rendering was a spherical source located above and to the left of the brick wall.

The original model on which this sculpting was performed consisted of a 1.8×1.8 brick wall in meters, and was depicted in Figure 2-2. The deepest resolution of the model was about 3.1 millimeters. This model did not come close to stretching the resolution capabilities of the system, and was easily rendered at interactive speeds on an SGI Octane. A traditional volumetric data structure of this brick wall would

require a $576 \times 576 \times 96$ volume raster, which is well beyond the capabilities of current volume sculpting systems.

There are some deficiencies to be noted in the final rendered image. First, the current meshing method has difficulty with well-defined edges, as evidenced by the edges of the bricks. The mortar does not look as inset from the brick as it is in the model. This is because a height field triangulation of a sharp edge between two different materials will be less sharp and will split the triangles along the edge between the materials. This gives the mortar appearance half of the w distance between brick and inset mortar, making the mortar appear to creep up the edge. Some sort of feature detection in the meshing could solve this.

7.1.2 Plaster Over Bricks

The second example, which can be seen in Figures 7-3 and 7-4, shows the capability of volumetric surfaces to efficiently handle layered models. We begin with the same brick wall as in the previous example, but then cover this wall with a thick layer of plaster. The plaster covers the brick and fills in the inset mortar gaps as well. When the chisel tool plunges through the plaster the brick underneath is revealed. In this case, a sharper chisel tool was used than in the previous example.

A second feature added to this example was material-selective tools. It is very straightforward to restrict tools to edit only certain materials. Mortar is a softer material than brick, and so it erodes faster. To model this, we used an additional chisel, which only affects mortar, to further erode this material.

7.1.3 Building Corner

The third and final example shows the corner of a building with walls of brick covered by plaster. Although the images in Figures 7-5 and 7-6 show only the lower portion of the corner, the model maintained in memory is three stories high. The building is 12.35 meters high, the front wall is 5.7 meters, and the side wall is 2.85 meters. All of this is maintained at 7 millimeter resolution, which is clearly beyond the capabilities of volumetric sculpting. It also pushes the limits of *this* system. It requires 265 megabytes of memory (although over half of this consists of openGL's

Figure 7-1: Interactive rendering of tooled, crumbling brick.

Figure 7-2: The ray-traced rendering.

Figure 7-3: Interactive rendering of a tooled, layered model.

Figure 7-4: The ray-traced rendering.

own memory in the form of display lists), and it is difficult to rotate the entire model at interactive speeds. To this end, an additional display mode shows only the edges of the slabs during rotation, and renders the entire model when the rotation handles are released. However, when the user zooms in to chisel at the slabs, interactive rates are maintained. This is because slabs outside the viewport are culled and because our local update algorithms minimize the polygons that need to be rendered during the tooling loop. Hence, even a model of this complexity is easy to position and edit in our system.

Another difficulty posed by this model is that the slabs that comprise the corner are necessarily distorted. The edges of the slabs located at the corner are at 45 degree angles to the edges along the walls. While these slabs require greater subdivision than the non-distorted slabs, all the algorithms in the system still work correctly for the distorted slabs.

7.2 Future Work

While this thesis has established the algorithmic engine for the interactive sculpting of volumetric surfaces, there are many areas of possible improvement that could make this system useful and ground-breaking in computer graphics.

- Tool control: It is fairly difficult to finely control the sculpting tool. This is because the mouse is a 2D input device in a 3D sculpting system, and this is a poor interaction metaphor. It is much easier to use a 3D input device. Even a 3D input device is not enough; force-feedback would greatly improve ease of use. This would make it possible to feel the materials as we sculpt. Beyond this different materials could have different hardnesses such that harder materials would resist sculpting more than softer ones. To this end the system is currently being interfaced to a PHANToM force-feedback device [25].
- Complex geometry: A major improvement would be the capability to import complex geometry. This was already discussed in Section 3.4. A second way to improve the system along these lines would be to import scanned meshes. Scanned meshes are very common in computer graphics, and the ability to volu-

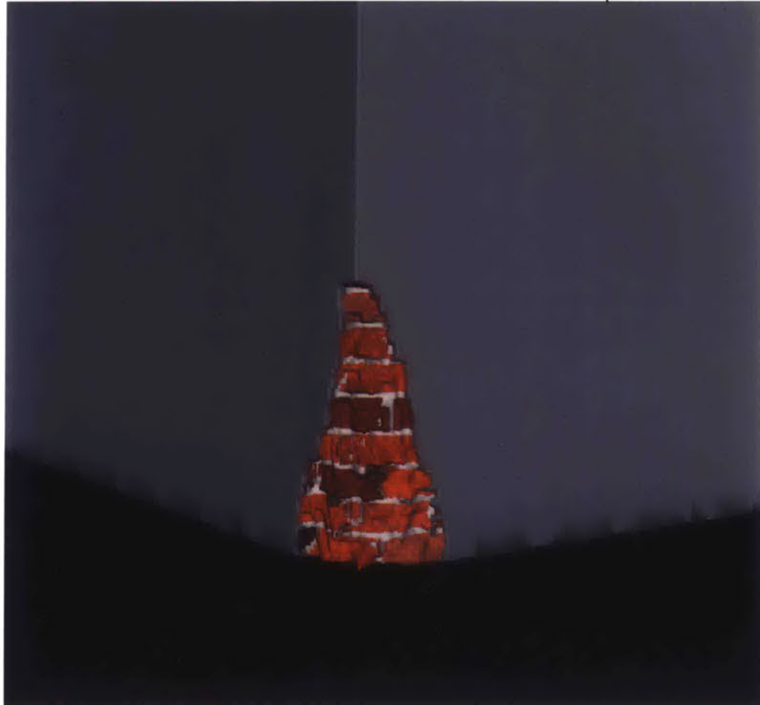


Figure 7-5: Interactive rendering of a tooled building corner.

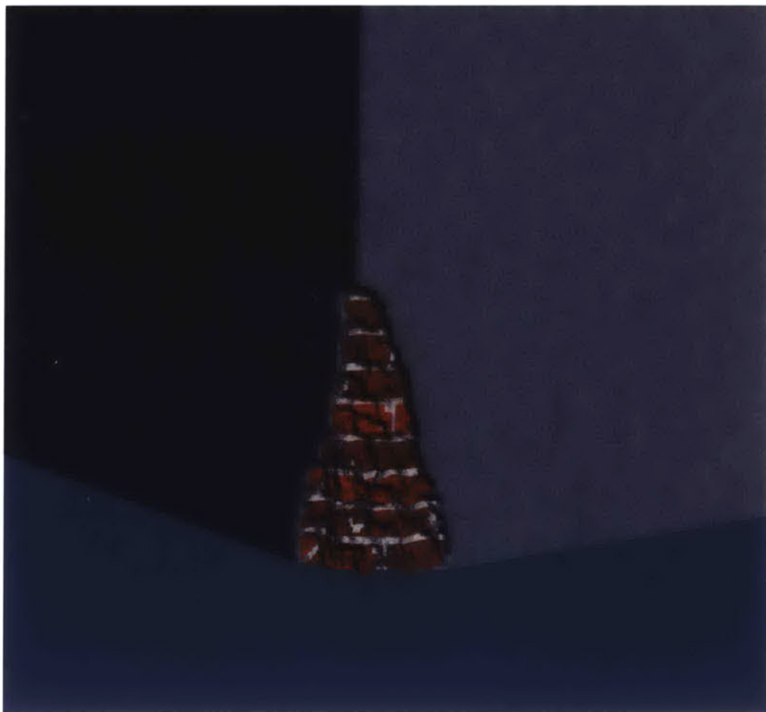


Figure 7-6: The ray-traced rendering.

metrically sculpt near the surface of these models would be very useful. The most difficult part of this task would be to establish the geometry of the slabs so as to achieve even spacing and minimal distortion. An optimization or spring-based technique would be useful in accomplishing this.

- Operators: Additional operators would make the system more flexible. Many different tool shapes could be added. A paint operator could be added to allow the addition of thin layers of paint. Another operator could be devised to allow the flaking off of paint chips. Tools to add volumetric data rather than subtract could also be useful. Physically based operators could be added to simulate cracking, dirt accumulation, and other natural weathering phenomena. There could also be user interface improvements, such as an undo option to reverse the effect of the last tool stroke. Finally, there is still room for further optimization of the critical algorithms to achieve even higher resolutions at interactive rates.

7.3 Conclusion

There is a clearly a need for systems that allow the interactive editing of complex surface appearance effects in computer graphics, and this thesis describes a system that is a step in this direction.

The first part of the system is a new, hierarchical data structure for representing volumetric data near the surface of an object at high resolutions. We then presented algorithms for filling this data structure from solid textures. A system was built that allows a user to sculpt these volumetric surfaces at high resolution and at interactive rates. The core of this system is algorithms for interactive meshing and local updating of the graphics display. Also necessary are algorithms for modifying the data structure after tool operations. An algorithm for generating a high-quality mesh for ray-tracing was also presented. Finally, examples were presented of weathering effects produced using the interactive system.

Hierarchical volumetric surfaces are a promising new direction in computer graphics, and hopefully their potential will continue to be explored. Their unique capabilities for representing realistic appearances could prove very useful in a variety of

applications.

Bibliography

[1] Maneesh Agrawala, Andrew C. Beers, and Marc Levoy. 3D painting on scanned surfaces. In *1995 Symposium on Interactive 3D Graphics*, pages 145–150. ACM SIGGRAPH, April 1995.

[2] Anthony A. Apodaca and M. W. Mantle. Renderman: Pursuing the future of graphics. *IEEE Computer Graphics and Applications*, 10(4):44–49, July 1990.

[3] James Arvo. A simple method for box-sphere intersection testing. In Andrew Glassner, editor, *Graphics Gems*, pages 335–339. Academic Press, 1990.

[4] James Arvo. Transforming axis-aligned boxes. In Andrew Glassner, editor, *Graphics Gems*, pages 548–550. Academic Press, 1990.

[5] Ricardo S. Avila and Lisa M. Sobierajski. A haptic interaction method for volume visualization. In *IEEE Visualization '96*. IEEE, October 1996.

[6] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[7] Wayne E. Carlson. A survey of computer graphics image encoding and storage formats. *Computer Graphics*, 25(2):67–75, April 1991.

[8] Sabine Coquillart. Extended free-form deformation: A sculpturing tool for 3D geometric modeling. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 187–196, August 1990.

[9] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. *Proceedings of SIGGRAPH 96*, pages 303–312, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*, chapter 14, pages 289–301. Springer-Verlag, 1997.

[11] Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Pedersen. Modeling and rendering of weathered stone. In *SIGGRAPH 99 Conference Proceedings*, Annual Conference Series, pages 225–234. ACM SIGGRAPH, Addison Wesley, August 1999.

[12] Julie Dorsey and Pat Hanrahan. Modeling and rendering of metallic patinas. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 387–396. ACM SIGGRAPH, Addison Wesley, August 1996.

[13] Julie Dorsey, Hans Pedersen, and Pat Hanrahan. Flow and changes in appearance. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 411–420. ACM SIGGRAPH, Addison Wesley, August 1996.

[14] B. Fishman and B.Schachter. Computer display of height fields. *Computers and Graphics*, pages 53–60, 1980.

[15] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition.* Addison-Wesley, 1990.

[16] Tinsley A. Galyean and John F. Hughes. Sculpting: An interactive volumetric modeling technique. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 267–274, July 1991.

[17] Michael Garland and Paul Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, School of Computer Science, Carnegie Mellon University, Sept 1995.

[18] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 215–223, August 1990.

[19] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 103–110, July 1987.

[20] Steve Hill. Tri-linear interpolation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 521–525. Academic Press, 1994.

[21] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 271–280, July 1989.

[22] Renate Kempf and Chris Frazier, editors. *OpenGL Reference Manual, Version 1.1*. Addison-Wesley, 1997.

[23] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hughes, Nick Faust, and Gregory Turner. Real-Time, continuous level of detail rendering of height fields. In *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 109–118. ACM SIGGRAPH, Addison Wesley, August 1996.

[24] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 163–169, July 1987.

[25] T.M. Massie and J.K. Salisbury. The phantom haptic interface: A device for probing virtual objects. In *Dynamic Systems and Control 1994*, volume 1, pages 295–301, Nov 1994.

[26] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, July 1998. Eurographics, Springer Wein.

[27] Bruce Naylor. SCULPT an interactive solid modeling tool. In *Proceedings of Graphics Interface '90*, pages 138–148, May 1990.

[28] Fabrice Neyret. A general and multiscale model for volumetric textures. In *Graphics Interface '95*, pages 83-91. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1995.

[29] Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1), January–March 1998.

[30] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *IEE Visualization '98*, pages 19–26. IEEE, October 1998.

[31] Darwyn R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):279–286, July 1985. Held in San Francisco, California.

[32] A. Pentland, I. Essa, M. Friedmann, B. Horowitz, and S. Sclaroff. The thingworld modeling system: Virtual sculpting by modal forces. In *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, volume 24, pages 143–144, March 1990.

[33] Ken Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH '85 Proceedings)*, 19(3):287–296, July 1985. Held in San Francisco, California.

[34] Hanan Samet. A top-down quadtree traversal algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(1), January 1985.

[35] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics*, 13(4):445–60, 1989.

[36] Hanan Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[37] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 151–160, August 1986.

[38] Renben Shu, Chen Zhou, and Mohan S. Kankanhalli. Adaptive marching cubes. *The Visual Computer*, 11(4):202–217, 1995.

[39] Lisa Sibierajski and Arie Kaufman. Volumetric ray tracing. In *1994 Symposium on Volume Visualization*, pages 11–18. ACM SIGGRAPH, October 1994.

[40] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[41] Sidney W. Wang and Arie E. Kaufman. Volume sculpting. In *1995 Symposium on Interactive 3D Graphics*, pages 151–156. ACM SIGGRAPH, April 1995.

[42] William Welch and Andrew Witkin. Free–Form shape design using triangulated surfaces. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, Computer Graphics Proceedings, Annual Conference Series, pages 247–256. ACM SIGGRAPH, ACM Press, July 1994.

[43] Rudiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 169–178, July 1998.

[44] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide, Second Edition.* Addison-Wesley, 1997.