

**VOXEL-BASED TERRAIN  
FOR REAL-TIME VIRTUAL SIMULATIONS**

By

ERIC STEPHEN LENGYEL

B.S. Mathematics, Virginia Polytechnic Institute and State University, 1994  
M.S. Mathematics, Virginia Polytechnic Institute and State University, 1996

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA  
DAVIS

Copyright © 2010

# **VOXEL-BASED TERRAIN FOR REAL-TIME VIRTUAL SIMULATIONS**

## **Abstract**

This dissertation provides the theoretical basis and implementation details for a complete and practical real-time voxel-based terrain rendering system. We first present a modified Marching Cubes algorithm designed to eliminate choices arising from ambiguities in the original algorithm and its successors in order to facilitate a faster implementation and to simplify the design of a level-of-detail algorithm. The modified Marching Cubes algorithm is extended to operate on voxel data at multiple resolutions in such a way that triangle meshes produced at all levels of detail correctly match geometrical features. We introduce a robust method for seamlessly joining voxel-based terrain meshes of different levels of detail and establish a transition structure that both simplifies the triangulation problem and eliminates the potential for shading artifacts. Finally, we discuss methods for applying texture maps and advanced shading techniques to voxel-based terrain meshes. These methods are designed to be fast and compatible with the widest possible range of graphics hardware across multiple platforms.

# Contents

<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>Acknowledgements</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Research Goals	2
1.2 Contributions	4
1.3 Organization	4
<b>2 Previous Work</b>	<b>6</b>
2.1 Height-Based Terrain	6
2.1.1 Crack Filling Strategies	6
2.1.2 Height-Based Terrain Algorithms	8
2.2 Volume Rendering	9
<b>3 Voxel Terrain</b>	<b>11</b>
3.1 Isosurface Extraction	11
3.1.1 Marching Cubes	11
3.1.2 Ambiguous Cases	13
3.2 Modified Marching Cubes	16
3.3 High-Performance Implementation	18
<b>4 Level of Detail</b>	<b>27</b>
4.1 Introduction	27
4.2 Geomip Terrain Meshes	28
4.2.1 Surface Shifting	28
4.2.2 Alternate Triangulations	31
4.3 Transition Cells	33
4.4 Transition Rendering	44
4.5 Implementation Details	51
<b>5 Texturing and Shading</b>	<b>55</b>
5.1 Introduction	55
5.2 Texture Mapping	55
5.3 Material Specification	59
5.4 Texture Organization	64
5.4.1 Texture Arrays	65
5.4.2 Texture Palettes	65

5.4.3 Palette Sampling	70
5.5 Terrain Shading	73
5.5.1 Texture Fetches	73
5.5.2 Tangent Space	75
5.5.3 Shading	79
5.6 Results	81
<b>6 Future Work</b>	<b>85</b>
6.1 Geomorphing	85
6.2 Sharp Features	86
6.3 Enhanced Transition Triangulations	87
6.4 Texture Map Orientation	88
<b>7 Conclusions</b>	<b>90</b>
<b>A Editing Voxel Terrain</b>	<b>92</b>
A.1 Terrain Creation Process	92
A.2 Terrain Editing Tools	92
A.3 Terrain Shaders	95
<b>Bibliography</b>	<b>97</b>

# List of Figures

<b>Figure 2.1.</b> Methods for filling cracks.	7
<b>Figure 3.2.</b> Edge placement on an ambiguous face.	14
<b>Figure 3.3.</b> A hole appearing on an ambiguous face.	14
<b>Figure 3.4.</b> Bilinear interpolation parameters on a cell face.	15
<b>Figure 3.5.</b> Edge configurations allowed by modified Marching Cubes.	17
<b>Figure 3.6.</b> Ambiguous face without a hole.	19
<b>Figure 3.7.</b> Slice of a single block of voxels.	21
<b>Figure 3.8.</b> Corner voxel numbering.	22
<b>Figure 3.9.</b> Cell vertex locations and edge codes.	23
<b>Figure 4.1.</b> Multiresolution terrain block dimensions.	29
<b>Figure 4.2.</b> Surface misalignment in multiresolution Marching Cubes.	30
<b>Figure 4.3.</b> Vertex placement in low-resolution Marching Cubes.	30
<b>Figure 4.4.</b> Shadow artifact caused by poor triangulation.	33
<b>Figure 4.5.</b> Cracks and holes on voxel resolution boundary.	34
<b>Figure 4.6.</b> Sample positions for border cells.	35
<b>Figure 4.7.</b> Division of a transition cell.	36
<b>Figure 4.8.</b> Topological layout of a transition cell.	36
<b>Figure 4.9.</b> Configuration of multiple transition cells.	37
<b>Figure 4.10.</b> Transition cell edge configurations for lateral faces.	44
<b>Figure 4.11.</b> Large terrain shown with three levels of detail.	45
<b>Figure 4.12.</b> Projection of transition cell vertices onto the tangent plane.	47
<b>Figure 4.13.</b> Transition configurations at a block corner.	48
<b>Figure 4.14.</b> Elimination of cracks and holes, example 1.	49
<b>Figure 4.15.</b> Elimination of cracks and holes, example 2.	50
<b>Figure 4.16.</b> Transition cell sample locations.	52
<b>Figure 4.17.</b> Transition cell vertex case index codes.	52
<b>Figure 4.18.</b> Transition cell vertex locations and edge codes.	53
<b>Figure 4.19.</b> Transition cell corner codes.	54
<b>Figure 5.1.</b> Texture stretching in single-plane projection.	56
<b>Figure 5.2.</b> Texture coordinates in cubic projection.	57
<b>Figure 5.3.</b> Comparison of triplanar blending parameters.	59
<b>Figure 5.4.</b> Textured terrain using cubic projection.	60
<b>Figure 5.5.</b> Holes created with poison material.	61
<b>Figure 5.6.</b> Material composition and mapping.	62
<b>Figure 5.7.</b> Material selection by nontrivial subcells.	63
<b>Figure 5.8.</b> Per-vertex geometric data and material attributes.	64

<b>Figure 5.9.</b> A texture array.	66
<b>Figure 5.10.</b> A texture palette.	69
<b>Figure 5.11.</b> Two tangent fields induced by cubic projection.	76
<b>Figure 5.12.</b> Terrain rendered with diffuse reflection.	83
<b>Figure 5.13.</b> Terrain rendered with diffuse and specular reflection.	84
<b>Figure 6.14.</b> Edge configurations for extracting sharp features.	87
<b>Figure 6.15.</b> Interior vertex placement for sharp features.	87
<b>Figure 6.16.</b> Lateral face edge configurations for enhanced transition cells.	88
<b>Figure 7.17.</b> Terrain screenshot in C4 Engine.	91
<b>Figure A.18.</b> Terrain editor tool panel.	93
<b>Figure A.19.</b> A terrain fragment shader graph.	96

# List of Tables

<b>Table 3.1.</b> Original Marching Cubes equivalence classes.	13
<b>Table 3.2.</b> Modified Marching Cubes equivalent classes.	19
<b>Table 4.1.</b> Equivalence classes having multiple triangulations.	32
<b>Table 4.2.</b> Transition cell equivalence classes, part 1.	39
<b>Table 4.3.</b> Transition cell equivalence classes, part 2.	39
<b>Table 4.4.</b> Transition cell equivalence classes, part 3.	39
<b>Table 4.5.</b> Transition cell equivalence classes, part 4.	40
<b>Table 4.6.</b> Transition cell equivalence classes, part 5.	41
<b>Table 4.7.</b> Transition cell equivalence classes, part 6.	41
<b>Table 4.8.</b> Transition class inverses, stabilizers, and sizes.	42
<b>Table 5.1.</b> Texture palette parameter comparison.	69
<b>Table 5.2.</b> Texture palette space utilization.	70
<b>Table 6.3.</b> Example enhanced transition cell triangulations.	89

# Acknowledgements

I wish to thank the dissertation committee, consisting of Associate Professor John Owens (chair), Professor Kenneth Joy, and Professor Nina Amenta, for their time and the valuable feedback they have provided.

I would also like to thank the many people in the C4 Engine community who tested my implementation of the voxel terrain methods developed in this dissertation and took the time to send me feedback. They discovered special cases that caused early versions to fail in strange ways, and this led to several improvements that made the whole architecture more robust.

Finally, I would like to thank my fiancée Andrea and my parents Steve and Penny for the continuous support they gave me throughout the entire time I worked on this project.

# Chapter 1

## Introduction

In real-time<sup>1</sup> applications, large terrain geometry has traditionally been organized as a scalar representation of elevation over a regular two-dimensional grid, and such a function is commonly referred to as a *height field*. The vast majority of existing software products having a real-time terrain rendering component use height fields due to the simplicity of triangle mesh generation for the terrain, the ability to create or modify the elevation data using ordinary painting programs such as Adobe Photoshop, the availability of a wide variety of two-dimensional terrain generation programs, and free access to high-resolution survey data such as the National Elevation Dataset [US Geological Survey 2006].

A plethora of algorithms have been developed to allow very large elevation datasets to be rendered with high performance, and these serve many types of applications very well. However, the restriction to a two-dimensional domain places significant limitations on the shapes of terrain features, and these limitations have become less acceptable for some types of applications as the power of graphics hardware has continued to increase. There can be no caves, overhangs, arches, or any other kind of feature that would prevent the terrain surface from being a continuous function on  $\mathbb{R}^2$ . Even a truly vertical cliff is impossible.

These limitations can be overcome by defining terrain using a scalar function over a three-dimensional grid, or *voxel map*. The function value no longer represents elevation, but a form of distance from the terrain surface, where a positive value means that a sample point is outside the terrain in empty space, and a negative value means that it is inside the terrain in solid space. The terrain surface itself corresponds to the set of points in space where the distance function is zero. Voxel maps provide the means to represent terrain having far greater topographical and topological complexity, and they enable the creation of all of the features that are named as impossible above when using height fields.

---

<sup>1</sup> The definition of “real time” has been distorted by many academic papers in order to include processes that can only be iterated a few times per second, and only if they are able to consume all available computing power on a particular machine. Throughout this dissertation, however, the use of the term “real time” means that the process to which it pertains is able to execute at a rate of 30 frames per second or better in combination with all other tasks necessary to render a complete virtual scene.

Although it provides many greater capabilities, the use of voxel-based terrain in real-time virtual simulations also introduces several new difficulties, enumerated below:

- The algorithms used to extract the terrain surface from a voxel map produce far greater numbers of vertices and triangles when compared to conventional two-dimensional terrain. This is true even for a voxel map constructed to match a given two-dimensional height field because the surface is split into polygons along grid planes in three dimensions instead of just two. The larger quantity of data needed to render the terrain can lead to performance problems while rendering, and this issue necessitates both the use of triangle mesh simplification techniques and the development of level-of-detail techniques to reduce the amount of geometric data required to display a large region of terrain.
- The development of a seamless level-of-detail algorithm for voxel-based terrain is vastly more complex than the analogous problem for height-based terrain. By “seamless”, we mean a method by which different parts of a terrain surface can be rendered next to each other without any cracks, holes, seams, or overlap.
- Texturing and shading of voxel-based terrain is significantly more difficult than it is for height-based terrain due to the topological possibilities that must be considered. This is complicated by the fact that modern shading techniques (for example, normal mapping) require a tangent field over the vertices of a mesh in addition to a normal field, and there are many topological configurations enabled by voxel-based terrain for which the existence of a smooth tangent field is a mathematical impossibility.

The subject of this dissertation is a set of robust solutions to the problems stated above with the goal of providing practical methods for the implementation of voxel-based terrain rendering in real-time virtual simulations. We pay particular attention to the quality of rendered images, compatibility across an array of current-generation and next-generation graphics hardware, rendering performance, and high-speed construction of terrain meshes.

## 1.1 Research Goals

The methods developed to construct and render voxel-based terrain during the course of research for this dissertation were influenced by the following design goals:

- Rendering performance is extremely important. Our methods must be designed so that large regions of voxel-based terrain can be rendered in combination with a variety of other geometry

in an outdoor scene at frame rates deemed acceptable for virtual simulations, which is ordinarily 30 frames per second or better. The other types of geometry that can be rendered in addition to the terrain include foliage, man-made structures, animated characters, vehicles, projectiles, and special effects.

- The terrain must be broken into pieces that can be independently rendered. This is essential for efficient culling of portions of the terrain that are not visible to the camera. Furthermore, it enables a discrete level of detail to be selected for each independently renderable piece of the terrain.
- We must be able to modify terrain at a local level without having to regenerate the entire terrain surface. This requirement is a good fit with the requirement that terrain be broken into pieces of reasonable size.
- It must not be necessary for the entire terrain to be loaded into memory all at once. At some granularity, it must be possible to dynamically load parts of the terrain from storage media and dynamically unload parts of the terrain from memory. This is also a good fit with the requirement that terrain be broken into pieces.
- No visible artifacts can be produced where pieces of terrain rendered at different levels of detail meet. Not only should the geometry appear to be smooth at such a boundary, but the surface shading should not exhibit any artifacts such as smearing or sudden changes in color or texture.
- Our methods must function correctly for arbitrary input data. We can place no constraints on the voxel data used to generate a terrain surface because it can often be modified automatically in unpredictable ways by an application, and any requirement that constraints be satisfied would only create annoying difficulties for engineers using our work.

In addition to high rendering performance, there is a strong desire to retain the ability to edit a voxel map and generate the resulting modified terrain meshes in real time. Such an ability serves two main applications. First, it would be possible to manually edit voxel-based terrain using an interactive program specifically designed for such a purpose (see Appendix A). This would allow an artist to quickly modify terrain while receiving nearly instant feedback. Second, it would allow terrain to be built or damaged in real time in response to events taking place during a simulation. For example, an explosion could create a spherical hole in the voxel map that produces a crater in the regenerated terrain.

## 1.2 Contributions

This dissertation provides the theoretical basis and implementation details for a complete and practical real-time voxel-based terrain rendering system. The specific contributions that make such a system possible are the following:

- We present a modified Marching Cubes algorithm and provide details for a high-performance single-processor implementation. Our modifications are designed to eliminate the choices arising from ambiguities in the original Marching Cubes algorithm and its successors in order to facilitate a faster implementation and to simplify the design of our level-of-detail algorithm.
- Our modified Marching Cubes algorithm is further extended to operate on voxel data at multiple resolutions. Vertex positions and triangle arrangements in terrain meshes generated for lower resolutions are adjusted to match corresponding geometric features in terrain meshes generated for higher resolutions.
- We introduce a robust method for seamlessly joining voxel-based terrain meshes of different levels of detail. Our method follows design concepts similar to those behind Marching Cubes, but operates on a different arrangement of voxel data. It also establishes a transition structure that both simplifies the triangulation problem and eliminates the potential for shading artifacts.
- We discuss methods for applying texture maps and advanced shading techniques to voxel-based terrain meshes. These methods are designed to be fast and compatible with the widest possible range of graphics hardware across multiple platforms.

## 1.3 Organization

A background discussion of real-time terrain rendering and work related to level-of-detail methods for voxel-based datasets appears in Chapter 2. The main contributions of this dissertation are organized into the following three chapters:

- In Chapter 3, we discuss the Marching Cubes algorithm and identify the well-known ambiguity problems that are associated with it. We then present modifications to Marching Cubes that allow for an efficient and robust implementation, and we eliminate the ambiguity problem by imposing a specific set of rules for edge placement.
- The topic of Chapter 4 is level of detail. Here, we explore the problems that arise in multiresolution Marching Cubes methods, and we make further modifications to our surface extraction technique to achieve higher mesh quality. This chapter also introduces a new algorithm that

makes seamless transitions between multiresolution voxel-based terrain meshes not only possible, but straightforward and practical.

- Chapter 5 discusses topics pertaining to texturing and shading. We examine the difficulties of applying texture maps to large terrain surfaces, and we present techniques that allow smooth mapping and blending of many textures over complex topography. We also present a method for applying tangent-space per-pixel lighting to arbitrarily-shaped terrain features.

These three core chapters are followed by a discussion of future research directions and concluding remarks about publications and industry deployment of our work.

# Chapter 2

## Previous Work

This chapter reviews previously published real-time algorithms and techniques for rendering large terrain surfaces with multiple levels of detail. We also review published methods for rendering multiresolution volumetric data. A large portion of this dissertation concerns itself with bridging the gap between these two subjects to produce a practical solution for rendering large terrains based on volumetric data with support for view-dependent level of detail.

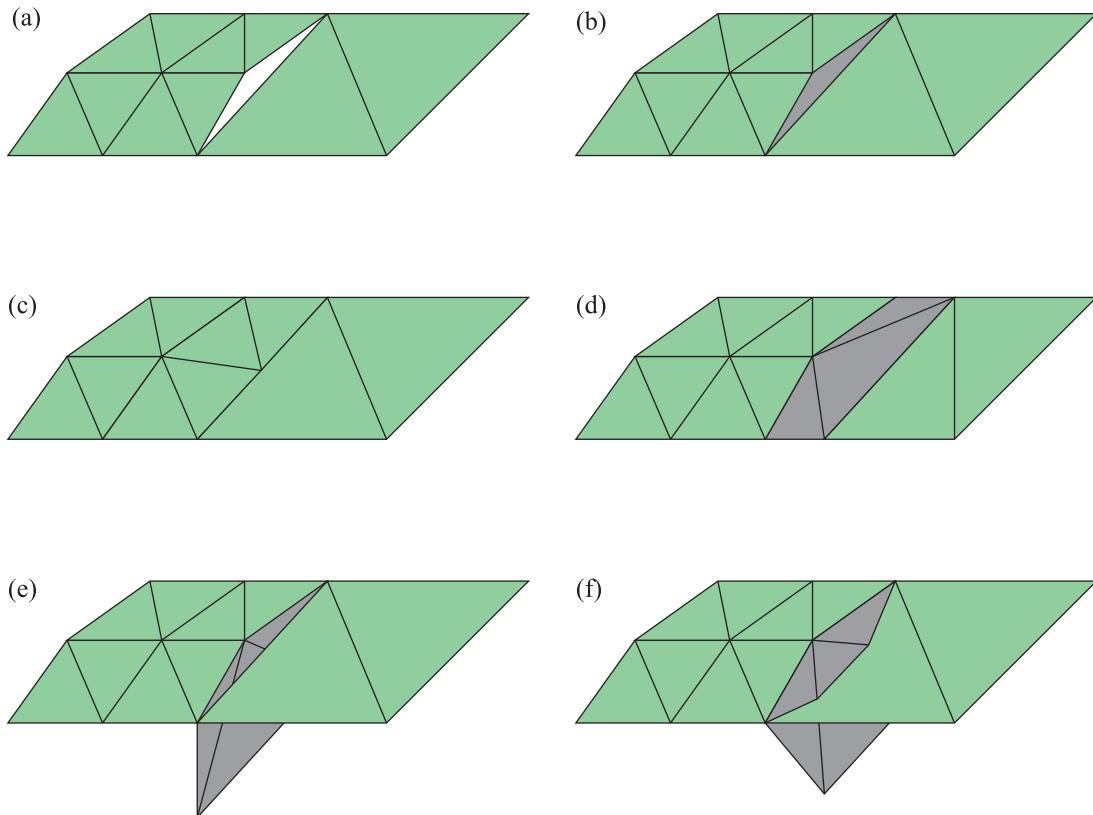
### 2.1 Height-Based Terrain

The existing literature pertaining specifically to terrain deals almost exclusively with height fields, and virtually all of the publications provide some form of level-of-detail management for high rendering performance. Several methods have been proposed for filling cracks that appear between terrain meshes rendered at different levels of detail, and some are more elegant than others. The process of joining terrain meshes together in order to fill or hide these cracks is known as “stitching”, and we briefly examine the common methods for stitching before reviewing specific terrain rendering publications.

#### 2.1.1 Crack Filling Strategies

As illustrated in Figure 2.1(a), a crack appears along the boundary between terrain meshes representing different levels of detail. In Figure 2.1(b), the crack is precisely filled with a triangle that uses existing vertices from both levels of detail, but the result creates serious difficulties for texturing and shading. This is due to the fact that the new triangle’s normal vector points horizontally in stark contrast to the normal vectors of the surrounding triangles. In Figure 2.1(c), vertices in the higher level of detail are moved to the adjacent edge in the lower level of detail, and a zero-area triangle connects the two meshes. This method also carries negative consequences when it comes to shading because the density of normal vector data is higher on one side of the crack. This creates a mismatch in the interpolated normal vectors used for shading, and the result is a visible seam along the boundary. In Figure 2.1(d), the crack is intentionally expanded and filled with a greater number of triangles compared to the patch applied in Figure 2.1(b). This enables the ability to render with much smoother shading and texturing.

The methods shown in Figures 2.1(e) and 2.1(f) are known as attaching “skirts” and “flanges”, respectively. In both cases, the crack is covered up by extra triangles attached to the higher-detail mesh, and they extend downward far enough to ensure that the crack is completely hidden. For a skirt, the triangles extend straight downward and thus exhibit the same shading problems seen in Figure 2.1(b). These problems are alleviated to a small degree by a flange because it also extends outward from the mesh to which it is attached. However, the triangles belonging to a flange penetrate the terrain surface, and this can cause depth buffer artifacts when additive lighting passes are rendered.



**Figure 2.1.** (a) An unstitched crack. (b) A triangle uses existing vertices to exactly fill the crack, but its normal starkly contrasts with the surrounding triangles. (c) The center vertex along the boundary is moved to the adjacent edge in the lower LOD, and a zero-area triangle stitches the meshes together. (d) An additional strip of triangles is created between the meshes, and its finite width maintains smooth normals. (e) A skirt is attached to the higher LOD. (f) A flange is attached to the higher LOD.

### 2.1.2 Height-Based Terrain Algorithms

Numerous methods for rendering large height-based terrain surfaces have been developed (see survey by Pajarola and Gobbetti [2007]), and they generally fall into two broad classes. Early methods focused on eliminating triangles in a view-dependent manner on the CPU at a very fine granularity, and then sending as little mesh data as possible to be rendered by the GPU. The following list summarizes the original publications that describe techniques falling into this class.

- The CLOD (continuous level of detail) algorithm [Lindstrom et al. 1996] takes a two-part approach in which terrain is first divided into blocks for which a detail level can be selected at a coarse granularity. Then, each block is retriangulated using a viewport-space error metric to dynamically control vertex density.
- The ROAM (real-time optimally adapting meshes) algorithm [Duchaineau et al. 1997] builds upon the CLOD algorithm by organizing terrain meshes into a triangle bintree structure for which view-independent error bounds are precalculated.
- Röttger et al. [1998] introduce geomorphing to the continuous level of detail algorithms. The geomorphing process smoothly moves vertices from a high-detail position to a low-detail position right before the mesh is changed in order to avoid visible popping.
- The progressive mesh technique [Hoppe 1996] was extended to height-based terrain [Hoppe 1998], and it enables smooth view-dependent terrain rendering with geomorphs.
- The Chunked LOD algorithm [Ulrich 2002] is another method in which terrain is divided into preprocessed blocks, and each block is independently triangulated in a view-dependent manner. This algorithm is designed to allow geomorphing, higher GPU utilization, and the ability to page “chunks” of a large terrain mesh out to disk. Mismatches between adjacent meshes rendered at different levels of detail are handled by rendering skirts along the boundaries, as shown in Figure 2.1(e).

With the widespread availability of high-performance commodity graphics hardware, the above techniques have been largely displaced by less CPU-intensive methods. The rapidly increasing triangle throughput possible on the GPU turned the fine-grained adjustment of terrain meshes into an unnecessary CPU expense that tends to dominate the processing time. Algorithms belonging to a newer class of techniques instead select levels of detail at a coarse granularity and simply pass precomputed meshes to the GPU with almost no CPU processing. These techniques all divide terrain data into blocks and store a set of triangle meshes generated at multiple resolutions. Which meshes to render are selected based on the distance from the camera.

The notion of texture clipmaps [Tanner 1998] inspired the creation of geometry clipmaps for height-based terrain [Losasso and Hoppe 2004; Asirvatham and Hoppe 2005]. In this work, a geometric mip (geomip) pyramid is precomputed for a large terrain mesh and stored on disk at multiple resolutions differing by powers of two. As the user explores the terrain, blocks are demand-loaded at the needed resolutions into a toroidal mesh buffer. The mismatches between neighboring blocks rendered at different resolutions are stitched together using zero-area triangles, as shown in Figure 2.1(c).

The work of Livny et al. [2009] extends geometry clipmaps so that strips of finite width are inserted between blocks of differing resolutions, as shown in Figure 2.1(d). This avoids several lighting and shading artifacts that can occur when zero-area triangles are used.

The possibility of using ray casting on the GPU to render terrain in real-time was demonstrated by Dick et al. [2009]. Their method uses multiresolution height field data and has the advantage that stitching between levels of detail is unnecessary because the terrain surface is never polygonalized. The primary disadvantage is, of course, slower rendering speed because GPUs are optimized for rasterization. Furthermore, the applicability of image quality techniques such as hardware-based multisampling are lost.

## 2.2 Volume Rendering

There is a vast reservoir of literature pertaining to the extraction of polygonal isosurfaces from volumetric data, and a large portion of it involves the Marching Cubes algorithm (see survey by Newman et al. [2006]). In the cases that triangle meshes are generated for multiple resolutions, we again encounter the cracking problem. Several unsophisticated methods proposed for eliminating these cracks are described in this section, but none of these methods are satisfactory for reasons discussed below. The only known robust solution is presented in Chapter 4 of this dissertation.

A method for patching cracks on the boundary plane between cells triangulated at different voxel resolutions was described by Shu et al. [1995], and it works by generating new polygons that fill the two-dimensional holes formed on the shared face. This has the effect of introducing polygons with harshly contrasting surface normals, however, making it unsuitable for an environment in which complex shading is applied and shadows are rendered.

Shekhar et al. [1996] employed a simpler technique in which vertices on the faces of the high-resolution cell are moved so that they are collinear with the endpoints of corresponding edges on the shared face of the low-resolution cell. This technique creates T-junctions that produce seams in the mesh when it is rendered, but this artifact could be eliminated by adding vertices to the low-

resolution edges where necessary. A more significant drawback of this technique is that the triangles generated for high-resolution cells are deformed when neighboring cells are triangulated at a lower resolution. In cases when the boundary between detail levels moves with a dynamic camera, this can cause unwanted surface morphing.

Importantly, both of the methods of Shu et al. and Shekhar et al. fail to handle situations in which the low-resolution cell is trivial and thus generates no edges, but neighboring high-resolution cells are nontrivial. There are no cracks to patch, but large holes appear in the overall mesh due to the lack of continuation for fine details in low-resolution cells. The solution that we present in Chapter 4 handles these cases correctly.

More recently, Kazhdan et al. [2007] approached the crack-patching problem from the opposite direction. They subdivide edges belonging to low-resolution cells and insert new vertices matching the locations of vertices belonging to the adjacent high-resolution cells. This again creates a problem if the low-resolution cell is trivial and has no edges to subdivide. Furthermore, when level-of-detail boundaries are not fixed, multiple triangulations must be stored for each low-resolution cell, one for each configuration in which different neighboring cells are triangulated at the same resolution or higher resolutions.

Finally, the notion of GPU-based ray casting has been put forth by Crassin et al. [2009]. This work constitutes a three-dimensional analog of the same idea in the two-dimensional domain, and it comes with the same set of advantages and disadvantages. In particular, cracks between levels of detail are not an issue, but performance becomes a problem, and image quality is reduced.

# Chapter 3

## Voxel Terrain

### 3.1 Isosurface Extraction

In order to render voxel-based terrain, we must be able to convert a three-dimensional scalar field representing the terrain into a set of vertices and triangles that can be rendered by the graphics hardware. A particular sample value in the scalar field corresponds to the perpendicular distance between the value's sample location and the surface of the terrain, or the boundary between empty space and solid space. A positive value indicates that a sample lies outside the volume occupied by the terrain in empty space, and a negative value indicates that a sample lies inside the terrain in solid space. The range of distances is limited by some choice of precision based on the number of bits used to store each sample value, and distances outside that range are clamped to the minimum and maximum representable values. Eight bits of storage per sample value typically provide sufficient precision, and that is the size that we use in this dissertation.<sup>2</sup>

A voxel map consists of a set of  $n \times m \times l$  sample values placed on a regular three-dimensional grid measuring  $n$  units in the  $x$  direction,  $m$  units in the  $y$  direction, and  $l$  units in the  $z$  direction. Using eight bits per voxel, a voxel map of this size obviously requires a total of  $nml$  bytes of raw storage. Each  $2 \times 2 \times 2$  subset of the voxel map delineates a cubic volume of space called a *cell*, and there are thus  $(n - 1) \times (m - 1) \times (l - 1)$  individual cells in the entire voxel map.

The conversion from a voxel map to a triangle mesh is accomplished through a process called *isosurface extraction*. In general, this process finds the set of points in space at which the values in a continuous extension of the discrete scalar field attain some constant value  $\alpha$  called the *isovalue*. If the continuous extension is constructed through trilinear interpolation, as is normally done, then the isosurface extraction process can always be transformed into one for which the isovalue is zero by subtracting  $\alpha$  from all sample values in the voxel map.

#### 3.1.1 Marching Cubes

A popular method for constructing a triangle mesh whose vertices coincide with the zero-valued isosurface is the Marching Cubes algorithm [Lorensen and Cline 1987]. This algorithm extracts the

---

<sup>2</sup> Even though a signed 8-bit storage type can represent values between -128 and 127, we restrict our sample values to the range -127 to 127 for symmetry.

isosurface by processing each cell in a voxel map independently of the others. For each cell, it determines what part of the isosurface passes through the cell, if any, and generates a triangulation inside the cell’s boundary based on the sample values at the eight corners of the cell. The results from all cells are combined to form the complete isosurface.

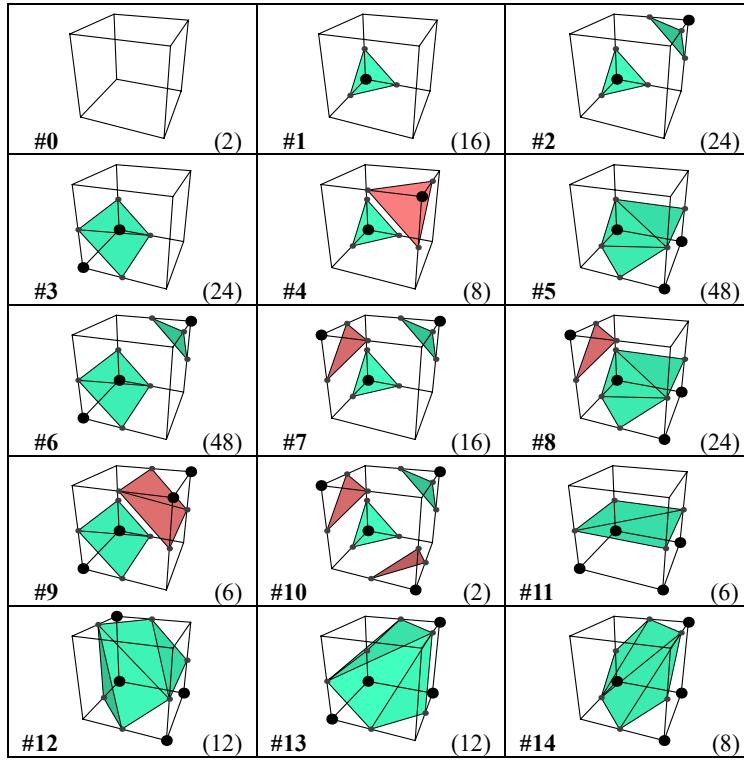
When processing a single cell, Marching Cubes first classifies each of the eight sample values at the corners of the cell as being either in empty space or in solid space, corresponding to the sign of the sample being positive or negative, respectively. A choice must be made globally as to whether a sample value of exactly zero is considered to be in empty space or in solid space, and a consistent classification either way is acceptable. Vertices belonging to the cell’s internal triangulation lie on the edges of the cell for which one endpoint has been classified as lying “outside” in empty space and the other endpoint has been classified as lying “inside” in solid space. However, as explained below, we do not explicitly check each of the 12 edges for this condition.

The classification into a binary inside/outside state at all eight corners gives rise to precisely  $2^8 = 256$  possible distinct cases to consider for a single cell. The original Marching Cubes algorithm partitions these 256 cases into the 15 equivalence classes shown in Table 3.1. Two cases belong to the same equivalence class when one case or its inverse can be rotated about some axis passing through the center of the cell so that it matches the other case exactly. The inverse of a case is obtained by inverting the inside/outside state of all eight of its corners (which is not equivalent to negating the sample values at all eight corners). Cases that are reflections of each other are not considered to be equivalent in the original Marching Cubes algorithm. If we include reflection in the equivalence relation, then classes #12 and #13 in Table 3.1 collapse into a single equivalence class.

For a particular cell, bits representing the inside/outside state of each corner are concatenated to form a single 8-bit case index, and this index is used to fetch an equivalence class index from a 256-entry lookup table. Once the index of the equivalence class has been selected, it is used to fetch additional information from a 15-entry table (one for each equivalence class). This information includes the vertex count and triangulation data that is identical for all cases belonging to the equivalence class. A separate 256-entry table identifies the edges on which the vertices lie for each case.

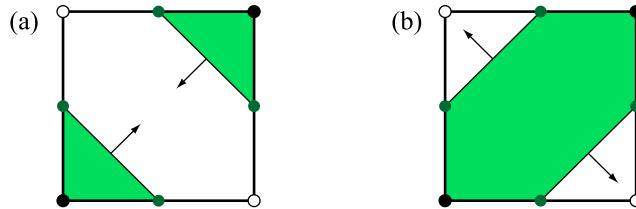
All of the equivalence class representatives in Table 3.1 show four or fewer corners as being inside solid space. When a particular case has five or more corners inside solid space, it belongs to the equivalence class represented by its inverse. The same set of vertices and triangles are generated as would be for its inverse, but the winding order of the vertices for each triangle is reversed so that the face normals are negated. This ensures that the resulting mesh surface consistently faces outward and that corners inside solid space are always beneath the surface.

**Table 3.1.** These are the 15 equivalence classes identified by the original Marching Cubes algorithm. The number in the lower-left corner of each entry is the class index, and the number in the lower-right corner is the number of cases belonging to the equivalence class out of the 256 total cases. A black dot indicates a corner that is inside the solid volume, and corners without a dot are outside. Green triangles are front-facing with respect to the view point, and red triangles are back-facing.



### 3.1.2 Ambiguous Cases

There is a well-known defect in the original Marching Cubes algorithm involving the so-called ambiguous cases, first identified by Dürst [1988]. This problem becomes evident when we consider a single face for which two diagonally opposing corners are inside and the other two corners are outside, as shown in Figure 3.2. A vertex is placed on all four edges of the face since a transition from solid to empty space takes place along each edge. We call such a face an *ambiguous face* because there are two ways to connect these four vertices with edges so that the inside corners lie in the interior of the triangulation and the outside corners lie in the exterior. In Figure 3.2(a), triangle edges connect vertices lying on face edges that share a corner having the inside state, and two separated components of solid space are created on the face. In Figure 3.2(b), triangle edges connect vertices lying on face edges that share a corner having the outside state, and a single component of solid space is created.

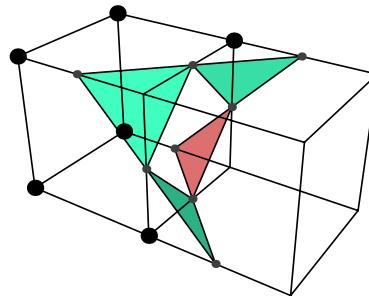


**Figure 3.2.** There are two ways to connect vertices on a cell face for which diagonally opposing corners have the same inside/outside state. Corners with solid dots are classified as inside, and corners with open dots are classified as outside. The green region represents solid space, and the arrows represent the outward surface normal direction along the edges.

If we choose to enforce a consistent rule for connecting vertices by edges in the representative cases of the 15 equivalence classes in Table 3.1, then we create a problem in some situations when two adjacent cells share an ambiguous face. A mismatch occurs between the edges generated for one cell and those generated for the other cell whenever exactly one of the two cells must be inverted. The result of this potential mismatch for each pair of adjacent cells is a strong likelihood of holes appearing in the final triangle mesh when all cells are considered. In the example shown in Figure 3.3, both cells belong to equivalence class #2 in Table 3.1, but the left cell must be inverted. The two triangulations do not produce matching edges along the shared face, and a large rectangular hole is created.

Equivalence classes #2, #6, #7, #8, #9, and #10 in Table 3.1 all contain ambiguous faces and are capable of producing holes under an implementation of the original Marching Cubes algorithm. Potential solutions to this problem abound in the literature and vary widely in complexity.

A notable solution that determines how edges should connect vertices on an ambiguous face was put forth by Nielson and Hamann [1991]. It works by examining how the bilinear interpolation



**Figure 3.3.** Two cells share an ambiguous face, and the cell on the left must be inverted to match its equivalence class representative. The edges of the resulting triangulations do not match along the shared face, and a large rectangular hole is created. Green polygons are front-facing, and red polygons are back-facing.

function  $F(s, t)$  of the four corner sample values behaves in the interior of the face. Referring to Figure 3.4, the function  $F(s, t)$  is defined as

$$F(s, t) = (1 - t)[(1 - s)A + sB] + t[(1 - s)C + sD], \quad (3.1)$$

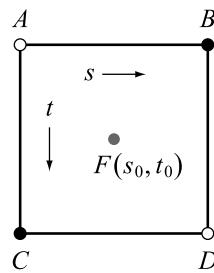
where  $A, B, C$ , and  $D$  are the sample values at the four corners, and  $s$  and  $t$  vary linearly from zero to one over the entire face as shown. We can locate the saddle point  $(s_0, t_0)$  of the function  $F$  by setting partial derivatives  $\partial F / \partial s$  and  $\partial F / \partial t$  to zero, from which we obtain

$$\begin{aligned} s_0 &= \frac{A - C}{A - B - C + D} & 1 - s_0 &= \frac{D - B}{A - B - C + D} \\ t_0 &= \frac{A - B}{A - B - C + D} & 1 - t_0 &= \frac{D - C}{A - B - C + D}. \end{aligned} \quad (3.2)$$

Plugging these into the function  $F$  gives us

$$F(s_0, t_0) = \frac{AD - BC}{A - B - C + D}. \quad (3.3)$$

The value  $F(s_0, t_0)$  of the interpolated scalar field at the saddle point decides whether we are to use a triangulation with two separated components as in Figure 3.2(a) or a triangulation with a single component as in Figure 3.2(b). Since  $A$  and  $D$  are both nonnegative and  $B$  and  $C$  are both negative, the denominator in Equation (3.3) is always positive. Thus, the value of the numerator can be used by itself to classify the inside/outside state of the saddle point. If  $AD < BC$ , then the saddle point lies inside solid space, and we choose the triangulation with a single component. Otherwise, the saddle point lies outside in empty space, and we choose the triangulation with two separated components.



**Figure 3.4.** An ambiguous face is parameterized for bilinear interpolation by letting  $s$  range from zero to one in the horizontal direction and letting  $t$  range from zero to one in the vertical direction.

Of course, the possibility of two different edge configurations for each ambiguous face requires that many more triangulation equivalence classes be recognized in addition to the 15 identified by the original Marching Cubes algorithm. This is especially true in light of the fact that classes #7, #8, #9, and #10 in Table 3.1 each have multiple ambiguous faces. These equivalence classes are explicitly identified by the Marching Cubes 33 algorithm [Chernyaev 1995], the title of which reveals that 33 equivalence classes are required to cover all possible combinations, but we do not explore them here. The extra per-cell computation needed to implement a decider function for ambiguous faces is not a significant disadvantage, but the combinatorial growth of the number triangulation classes becomes a serious problem when we consider level of detail in Chapter 4. We therefore prefer a solution that simplifies our choices when it comes to triangulating cells.

## 3.2 Modified Marching Cubes

When generating the polygonal surface for voxel-based terrain, our most important requirement is that the resulting mesh be robust, meaning that each connected component is continuous and free of holes no matter what values are present in the voxel map. Assuming this requirement is satisfied, our primary goal among the remaining concerns is maximum performance. We want to be able to modify the voxel map underlying a chunk of terrain in real-time during the course of an interactive simulation and regenerate its triangle mesh without causing a noticeable hesitation for the user.

In order to achieve the highest speed possible, we make the choice not to concern ourselves with the quality by which a single cell's triangulation matches the trilinear interpolation of the sample values at its corners, as would be determined by applying Equation (3.3) to the cell's faces. As long as the vertices used for the triangulation lie on the isosurface and the robustness requirements are met, then any minimal cell triangulation is acceptable.

With these goals of robustness and speed in mind, we have developed a modified version of the Marching Cubes algorithm that eliminates the ambiguous cases without adding any extra per-cell computation. The key observation about the original Marching Cubes algorithm that leads to our modified version is that the ambiguous cases always arise due to the inclusion of inverses in the equivalence classes shown in Table 3.1. We find it necessary to split some of these equivalence classes into two distinct classes. When we make a split, one of the new classes includes all cases having fewer than four corners inside solid space, and the other new class includes all cases having more than four corners inside solid space.

The robustness requirement for a terrain mesh built from a voxel map can be summarized in a single statement that, when applied to all cell faces in a voxel map, guarantees a mesh whose connected components are continuous and free of holes. This statement is the following:

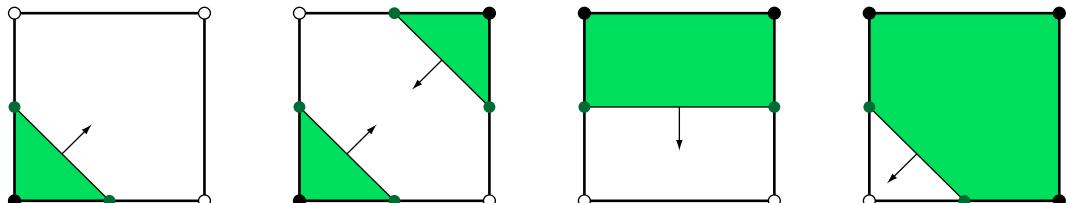
For any two adjacent cells  $A$  and  $B$ , any edge belonging to the triangulation of cell  $A$  lying on the shared face between the two cells must be matched by an edge belonging to the triangulation of cell  $B$  connecting the same two endpoints with the opposite orientation.

Here, the orientation of an edge is the direction along the edge that winds counterclockwise about its owning triangle when viewed from the front side.

When we consider the possible inside/outside states of the four corners of a cell face, there are really only four nontrivial cases after we remove those that are equivalent through rotation, and one of them is the ambiguous case shown in Figure 3.2. We can satisfy the edge matching requirement for mesh robustness by enforcing a consistent edge placement in the ambiguous case: we always connect vertices on adjacent edges sharing an inside corner. (Such a rule is called preferred polarity by Ning and Bloomenthal [1993].) Following this rule, the four cases shown in Figure 3.5 are the only nontrivial edge configurations allowed by our algorithm (and a fifth configuration is the trivial face having no edges because all four corners have the same inside/outside state).

Our modified Marching Cubes algorithm does not permit the edge configuration shown in Figure 3.2(b), and thus the triangulation of the left cell in Figure 3.3 is not legal. This situation requires that we introduce a new set of triangulation equivalence classes that only exhibit the per-face edge configurations shown in Figure 3.5.

As noted earlier, the ambiguous face problem arises from the inclusion of inverses in the equivalence classes of the original Marching Cubes algorithm. For the equivalence classes having four corners inside and four corners outside, including inverses is not necessary because the inverse of



**Figure 3.5.** These are the only four nontrivial edge configurations allowed on a cell face by our modified Marching Cubes algorithm modulo rotation. Corners with solid dots are classified as inside, and corners with open dots are classified as outside. The green region represents solid space, and the arrows represent the outward surface normal direction along the edges.

each member case is identical to a rotation of the class representative. There are three remaining equivalence classes that contain ambiguous faces, and they are classes #2, #6, and #7 in Table 3.1. Upon inversion, these three classes each produce the disallowed edge configuration shown in Figure 3.2(b) for their ambiguous faces.

We remedy this situation by removing inversion from the equivalence relation for the classes having an ambiguous face, leaving only rotation. We then establish three new equivalence classes to contain cases having more than four corners inside solid space from the original equivalence classes #2, #6, and #7. The result is the set of 18 equivalence classes shown in Table 3.2, where the three new classes are displayed in the last row. Classes #15, #16, and #17 are composed of the inverses of the representative cases of classes #2, #6, and #7, respectively, and their rotations.

Using the new set of equivalence classes shown in Table 3.2, the pair of cells sharing an ambiguous face shown in Figure 3.3 is now triangulated as shown in Figure 3.6. The left cell no longer belongs to equivalence class #2 (through inversion), but to the new equivalence class #15. Because the consistent edge configuration rule is followed, there is no longer a hole on the shared face.

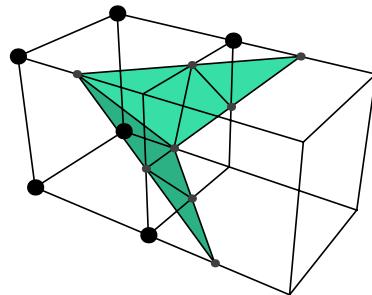
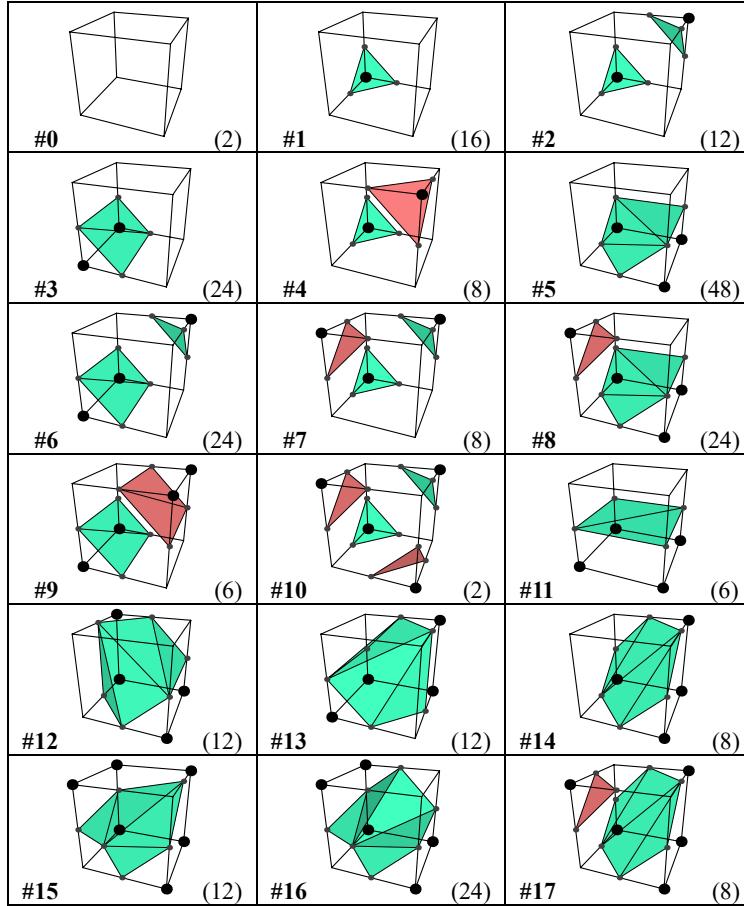
### 3.3 High-Performance Implementation

An implementation of the modified Marching Cubes algorithm that is both efficient and robust can be surprisingly difficult to achieve. There are several delicate intricacies that, if not handled properly, can lead to artifacts such as seams and duplicate triangles in the final terrain mesh. This section describes many of the details of our implementation with a particular focus on speed of execution and exactness of the results.

Our implementation of the modified Marching Cubes algorithm was designed with the following goals in mind:

- It must be possible to edit small regions of terrain without having to reconstruct the entire terrain surface.
- When rendering the resulting terrain mesh, it must be possible to efficiently cull reasonably-sized parts of it when they are not visible in the view frustum.
- The resulting terrain mesh must be perfectly watertight without occurrences of any tiny gaps between triangles or tiny regions where triangles overlap.
- Vertices should be shared as often as possible among neighboring cells in order to take full advantage of hardware post-transform caches and to minimize the storage requirements for the resulting terrain mesh.
- The amount of memory required while generating a terrain mesh should be kept to a minimum.

**Table 3.2.** These are the 18 equivalence classes identified by our modified Marching Cubes algorithm. The number in the lower-left corner of each entry is the class index, and the number in the lower-right corner is the number of cases belonging to the equivalence class out of the 256 total cases. A black dot indicates a corner that is inside the solid volume, and corners without a dot are outside. Green triangles are front-facing, and red triangles are back-facing.



**Figure 3.6.** The two cells first shown in Figure 3.3 share an ambiguous face. Since inversion is no longer part of the equivalence relation, the cell on the left is a member of one of the new equivalence classes. Its triangulation now follows the rules shown in Figure 3.5 on the shared face, and as a result, there is no longer a hole in the mesh as there was before.

Dividing a large voxel map into relatively small equal-sized blocks helps achieve some of these goals. In our implementation, factors involving typical edit dimensions and level-of-detail hierarchies influenced an ultimate decision to divide voxel maps into blocks measuring  $16 \times 16 \times 16$  cells. Each block owns the voxels located at the minimum corners of the cells it contains, where the minimum corner of a cell is the one having the smallest  $x$ ,  $y$ , and  $z$  coordinates.

When generating a triangle mesh for voxel-based terrain, we operate on each  $16 \times 16 \times 16$  block in isolation, but we must also have access to voxels belonging to neighboring blocks. If any part of the entire terrain mesh intersects a particular block, then that part becomes an individually renderable and individually cullable geometry node in the scene. Whenever a region of the voxel map is edited, all blocks intersecting the region's bounding box have their triangle meshes regenerated.

Due to the volumetric nature of the data, the storage requirements for a raw voxel map representing a large terrain can grow beyond the limits of addressable memory for many machines.<sup>3</sup> This necessitates that a voxel map be stored in a compressed form. We have found that a simple run-length encoding (RLE) technique achieves high lossless compression rates due to the tendency for terrain data to contain large regions of solid space and large regions of empty space.<sup>4,5</sup> The compressible runs are particularly long when voxel data is stored sequentially in horizontal spans since terrain data tends to include regions in which the terrain surface is nearly horizontal.

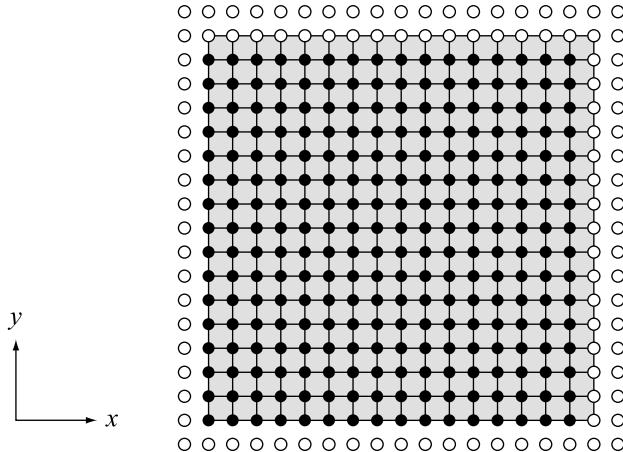
When we generate the triangle mesh for a particular block, we must have access to a volume of  $17 \times 17 \times 17$  voxels in order to run the modified Marching Cubes algorithm on a  $16 \times 16 \times 16$  set of cells. Some corner voxels belonging to the cells on the positive boundaries of the block are owned by neighboring blocks. Furthermore, in order to calculate a normal vector field, we need access to one additional voxel preceding and succeeding each voxel used by the block so that we can compute central differences in the directions of the three coordinate axes. In total, generating the triangle mesh for one block requires access to a volume of  $19 \times 19 \times 19$  voxels, where one layer of voxels precedes the negative boundaries of the block, and two layers of voxels succeed the positive boundaries of the block. A two-dimensional slice of this volume is shown in Figure 3.7.

---

<sup>3</sup> This is true even if only one byte representing the signed distance to the terrain surface is stored per voxel. Later in Chapter 5, we add two more bytes per voxel containing information about materials, tripling the amount of raw storage space.

<sup>4</sup> This relies on voxel values being clamped to a small range so that large groups of voxels lying in solid space or empty space have equal values.

<sup>5</sup> For example, the compression ratio for the terrain shown in Figure 7.1 is 33:1.

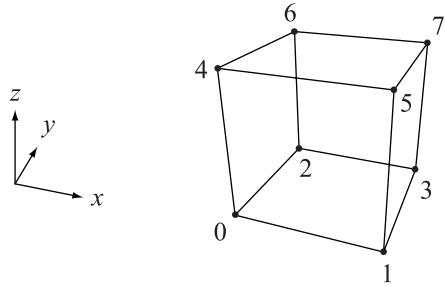


**Figure 3.7.** This is a horizontal slice of the three-dimensional set of voxels to which access is necessary when generating the triangle mesh for a single block of  $16 \times 16 \times 16$  cells. The shaded area corresponds to the volume occupied by the block, and the solid dots represent the voxels owned by the block. Open dots represent voxels owned by neighboring blocks.

The first step in generating the triangle mesh for a particular block is decompressing the set of blocks for which we need access to voxel data. In general, this means that 27 blocks must be decompressed to obtain a raw voxel map containing all of the required voxel data. If the block being triangulated lies along the boundary of the entire data set, then fewer blocks are decompressed, and voxel values are duplicated along borders as necessary.

Once the raw voxel data is available in memory, we visit each of the 4096 cells in the block and run the modified Marching Cubes algorithm. We start with the cell at the coordinates  $(0, 0, 0)$  in the block and then increment the  $x$  coordinate to visit each cell in a row. After each entire row has been visited, we increment the  $y$  coordinate and reset the  $x$  coordinate to zero. This continues until the entire  $16 \times 16$  *deck* is visited for a constant  $z$  coordinate, and then we increment the  $z$  coordinate to start a new deck at  $(0, 0, z)$ . It is important that we visit cells in this order since vertex sharing depends on data being available for cells that were previously visited.

Within a cell, the corner voxels are numbered as shown in Figure 3.8. The eight corners are classified as inside or outside the terrain volume by examining the sign bits of the 8-bit integer sample values. If the sign bit is zero, then the voxel lies outside, and if the sign bit is one, the voxel lies inside. (Consequently, a sample value of zero is always classified as outside.) The sign bits are concatenated into a single 8-bit quantity for which the least significant bit corresponds to corner 0, and the most significant bit corresponds to corner 7, as demonstrated in Listing 3.1. This quantity provides the cell's case index, used to fetch the equivalence class index from a 256-entry lookup table. If the case index is either 0 or 255, then the cell contains no triangles. Both of these



**Figure 3.8.** Voxels at the corners of a cell are numbered as shown.

cases can be skipped after making only one comparison by using the “bit twiddling” trick shown in Listing 3.1. We perform a signed right shift on one of the sample values to fill a byte with copies of its sign bit and then exclusive OR that byte with the case index. The result is zero for case indexes 0 and 255, but is nonzero for all others.

The equivalence class index for a cell maps to information about the vertex count, triangle count, and the vertex sequence for the cell’s triangulation. This data is the same for all members of the equivalence class, but the actual location of the vertices varies among class members. The full 8-bit case index is used to fetch a list of edges on which the vertices lie from a table. Vertices never occur in the interior of a cell, but only on *active edges* of the cell, where an active edge is an edge for which one endpoint is classified as inside and the other is classified as outside.

For each cell, it is possible to generate up to 12 distinct vertices and up to 5 triangles. In order to maximize vertex sharing in the triangle mesh generated for the block, we need to be able to reuse

**Listing 3.1.** The sign bits of the eight corner sample values are concatenated to form the case index for a cell, and the trivial cases are immediately eliminated from further processing. The type of `corner` is an array of eight signed byte values.

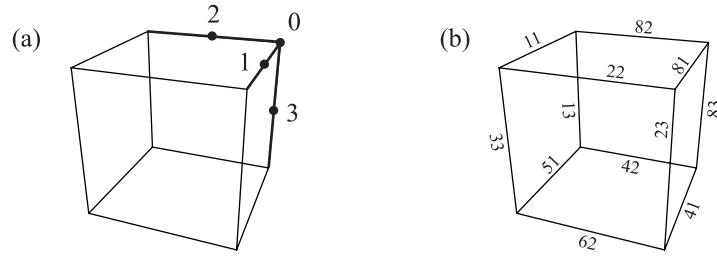
```

unsigned long caseCode = ((corner[0] >> 7) & 0x01)
| ((corner[1] >> 6) & 0x02)
| ((corner[2] >> 5) & 0x04)
| ((corner[3] >> 4) & 0x08)
| ((corner[4] >> 3) & 0x10)
| ((corner[5] >> 2) & 0x20)
| ((corner[6] >> 1) & 0x40)
| (corner[7] & 0x80);

if ((caseCode ^ ((corner[7] >> 7) & 0xFF)) != 0)
{
    // Cell has a nontrivial triangulation.
}

```

vertices that were previously output when possible. This is accomplished by limiting the locations in which a new vertex can be created for any cell and reusing a vertex from a previously triangulated cell where the creation of a new vertex is not allowed. As shown in Figure 3.9(a), we only allow new vertices to be created at the maximal corner and maximal edges of a cell, where the maximal corner is the one having the largest  $x$ ,  $y$ , and  $z$  coordinates, and a maximal edge is one having the maximal corner as an endpoint.



**Figure 3.9.** For most cells in a block, new vertices can only be created at the maximal corner and three maximal edges. (a) The locations at which new vertices are allowed are numbered 0 through 3 as shown. (b) An 8-bit hexadecimal code is assigned to each edge to provide a mapping to a reusable vertex belonging to a preceding cell.

Each edge of a cell is assigned an 8-bit code, as shown in Figure 3.9(b), that provides a mapping to a preceding cell and the coincident edge on that preceding cell for which new vertex creation was allowed. The high nibble of this code indicates which direction to go in order to reach the correct preceding cell. The bit values 1, 2, and 4 in this nibble indicate that we must subtract one from the  $x$ ,  $y$ , and/or  $z$  coordinate, respectively. These bits can be combined to indicate that the preceding cell is diagonally adjacent across an edge or across the minimal corner. The bit value 8 indicates that a new vertex is to be created for the current cell. The low nibble of the 8-bit code gives the index of the vertex in the preceding cell that should be reused or the index of the vertex in the current cell that should be created.

In the event that the sample value at a corner is exactly zero, the vertex lying on any active edge adjacent to that corner is placed precisely at the corner location. The only corner at which a new vertex is allowed to be created is corner 7, so vertices for any other corners must be reused from preceding cells. A 3-bit direction code leading to the proper cell can easily be obtained by inverting the 3-bit corner index (bitwise, by exclusive ORing with the number 7), where the bit values 1, 2, and 4 indicate that we must subtract one from the  $x$ ,  $y$ , and/or  $z$  coordinate, respectively.

For cells occurring along the minimal boundaries of a block, the preceding cells needed for vertex reuse may not exist. In these cases, we allow new vertex creation on additional edges of a cell. While iterating through the cells in a block, a 3-bit mask is maintained whose bits indicate whether corresponding bits in a direction code are valid. When a direction code is used to locate a preceding cell, it is first ANDed with the validity mask to determine whether the preceding cell exists, and if not, the creation of a new vertex in the current cell is permitted. (This test always fails for 4-bit direction codes having the bit with value 8 set. This is the motivation for assigning the codes 0x81, 0x82, and 0x83 to the maximal edges instead of 0x01, 0x02, and 0x03.)

For each cell, we must have space to store four vertex indexes corresponding to the locations shown in Figure 3.9(a) so that they can be reused by cells triangulated at a later time. It is never the case that all four vertex slots are used at once (because a vertex lying on the interior of an edge implies no vertex at the corner), but we must be able to identify the vertices using a fixed indexing scheme. The vertices used by a cell are always owned by the cell itself, the preceding cell in the same row, one of two adjacent cells in the preceding row, or one of four cells in the preceding deck. We can therefore limit the vertex history that we store while processing a block to two decks containing  $16 \times 16$  cells each and ping-pong between them as the  $z$  coordinate is incremented.

When generating the vertices for a cell, the equivalence class index maps to a data structure containing the number of vertices, and the case index maps to a list of 16-bit codes providing information about the edges on which the vertices lie. The low byte of each 16-bit code contains the corner indexes of the edge's endpoints in one nibble each, and the high byte contains the mapping code shown in Figure 3.9(b). For example, the 16-bit code for the edge connecting corners 1 and 5 is 0x2315. The higher-numbered corner always comes last to easily enable a consistent ordering of endpoints when interpolating sample values. This 16-bit code and the sample values at the cell's corners constitute all the information needed to generate new vertices or reuse previously-generated vertices.

For each edge on which we need to generate a new vertex, we calculate a fixed-point interpolation parameter  $t$  using the sample values  $d_0$  and  $d_1$  at the edge's endpoints:

$$t = \frac{d_1}{d_1 - d_0}. \quad (3.4)$$

It is extremely important that this parameter is calculated using a consistent ordering of the endpoints, or seams will appear between adjacent cells belonging to different blocks. We always associate the distance  $d_0$  with the lower-numbered endpoint and the distance  $d_1$  with the higher-

numbered endpoint (see Figure 3.8). Calling the endpoint positions  $\mathbf{P}_0$  and  $\mathbf{P}_1$ , the position  $\mathbf{Q}$  of the new vertex is given by

$$\mathbf{Q} = t\mathbf{P}_0 + (1 - t)\mathbf{P}_1. \quad (3.5)$$

In Listing 3.2, we use an 8-bit fraction, allowing the new vertex to be located at one of 257 possible positions along the edge when both endpoints are included. If the vertex is coincident with one of the endpoints, then the fraction bits are all zeros. If this condition is detected, then separate branches of execution are followed because sharing of corner vertices is handled differently than sharing of vertices lying on the interiors of edges.

It is possible to generate triangles having zero area when one or more of the corner sample values for a cell is zero. For example, when we triangulate a cell for which one corner sample value is zero and the seven remaining corner sample values are negative, then we generate the single triangle of equivalence class #1 (see Table 3.2). However, all three vertices lie exactly at the corner having zero sample value. Such triangles are eliminated after a simple area calculation (using fixed-point vertex coordinates) indicates that they are degenerate.

The final step in generating a triangle mesh for a block is to build the vertex array that is used for rendering. When triangulating individual cells, we store coordinates of new vertices in a fixed-point format at a scale in which each cell edge measures one unit in length. When building the final vertex array, we convert each set of vertex coordinates to floating-point format, scale them to the block's actual size, and then translate them to the block's global position. Vertices that are unused due to triangle elimination are not included in the final vertex array, and vertex indexes in the triangle array are remapped to account for the removed vertices.

**Listing 3.2.** The lowest two nibbles of the edge code identify the corner indexes corresponding to the endpoints of the edge. The sample values at those corners are used to calculate a fixed-point interpolation parameter (with eight fraction bits) for the edge, giving the location of the isosurface.

```
unsigned short v0 = (edgeCode >> 4) & 0x0F;
unsigned short v1 = edgeCode & 0x0F;

long d0 = corner[v0];
long d1 = corner[v1];

long t = (d1 << 8) / (d1 - d0);

if ((t & 0x00FF) != 0)
{
    // Vertex lies in the interior of the edge.

    long u = 0x0100 - t;
    Q = t * P0 + u * P1;
}
else if (t == 0)
{
    // Vertex lies at the higher-numbered endpoint.

    if (v1 == 7)
    {
        // This cell owns the vertex.
    }
    else
    {
        // Try to reuse corner vertex from a preceding cell.
    }
}
else
{
    // Vertex lies at the lower-numbered endpoint.
    // Always try to reuse corner vertex from a preceding cell.
}
```

# Chapter 4

## Level of Detail

### 4.1 Introduction

Due to the high numbers of vertices and triangles that are produced, even after mesh simplification procedures have been applied, a level-of-detail (LOD) system is required in order to maintain high performance when rendering large terrain surfaces. A LOD system generally aims to flatten the viewport-space density of vertices and triangles used to render a scene, and without it, per-vertex calculations can dominate the processing time on the GPU. Furthermore, rendering triangles of very small size is inefficient because a large portion of the fragments processed correspond to dead pixels belonging to quads<sup>6</sup> covering the triangles' boundaries.

Many methods exist for transforming a general mesh into a lower-detail version comprising fewer vertices and triangles. If such methods were applied to the terrain blocks described in the previous chapter, then the amount of mesh data for each block would become smaller at lower detail levels, but the number of discrete meshes covering an entire terrain would remain the same. This can lead to unnecessarily large numbers of triangle batches being sent to the GPU to render large terrains, which can be very inefficient.

Our solution extends the concept of screen-space geometry density flattening to the meshes themselves by having lower-detail meshes represent larger volumes within the terrain's voxel map. The method presented here is a multiresolution technique that bears similarity to texture mipmaps in that lower-detail meshes are created from lower-resolution samplings of the voxel map, bringing about the term *geomip* (geometric *multum in parvo*) meshes.

The primary problem that arises is the formation of a variety of prominent artifacts along the boundary between meshes representing different detail levels. In this chapter, we introduce a robust method for patching the seams, cracks, and holes that appear due to the mismatches between edges lying in the boundary plane, and we call our method the Transition Cubes algorithm. Our method

---

<sup>6</sup> A *quad* is a  $2 \times 2$  group of pixels corresponding to the smallest unit of fragment processing performed by the GPU. When a triangle is rasterized, a fragment program is executed for all four pixels in each of the quads intersected by the triangle. The fragment program outputs for pixels lying outside the triangle's boundary, called *dead pixels*, are discarded. The GPU renders in quad units so that valid interpolant and local variable derivatives can always be obtained through simple differencing.

follows in the footsteps of the Marching Cubes algorithm, but it operates on a different type of spatial structure called a “transition cell”. Transition cells are inserted between ordinary cells where the voxel map sampling frequencies differ by a factor of two. Our method provides the means to efficiently generate triangles that smoothly and seamlessly connect terrain blocks rendered at different levels of detail, producing an artifact-free multiresolution rendition of large voxel-based datasets.

Note that all of the previous methods described in Section 2.2 allow arbitrary differences in resolution between neighboring cells, but we restrict the difference to a factor of two in this chapter. Doing so allows us to efficiently address a fixed set of possibilities and to generate seamless transitions between levels of detail in a constant per-cell time.

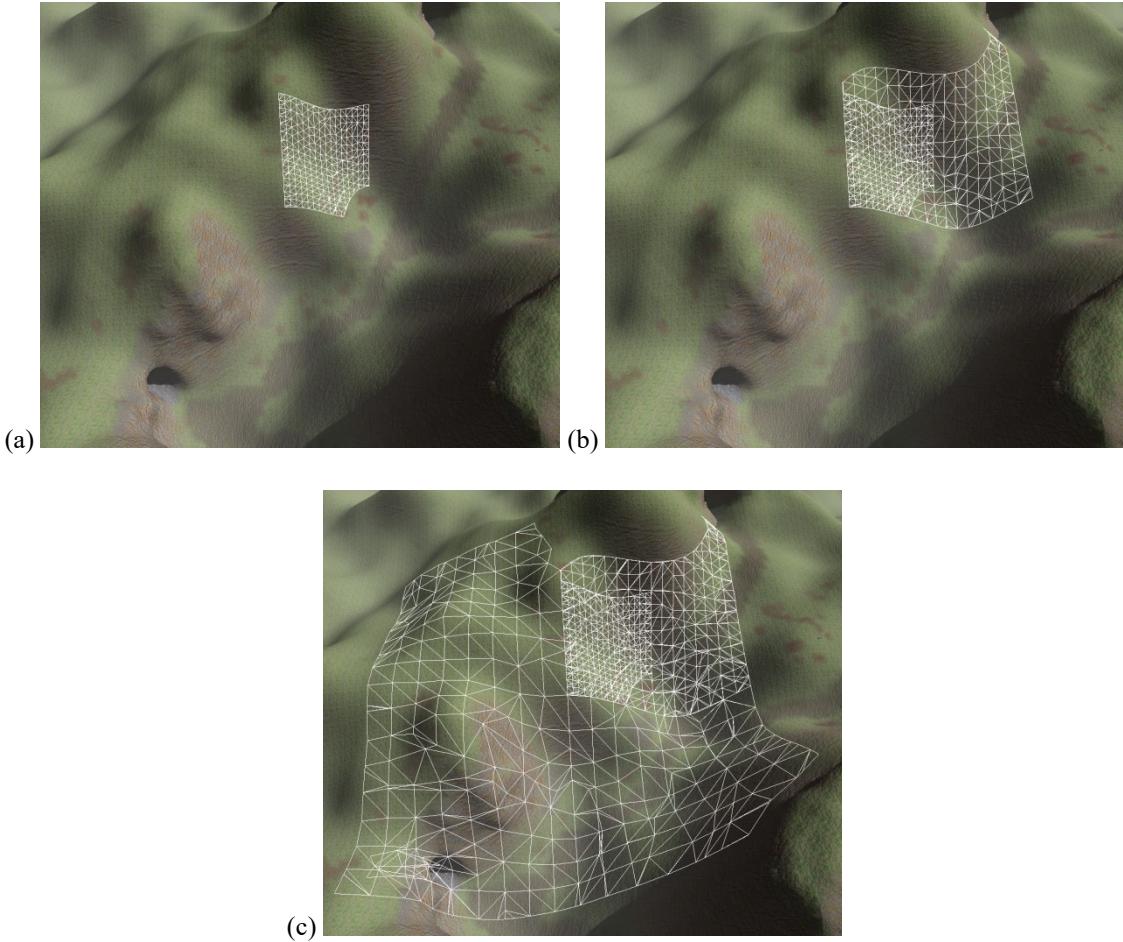
## 4.2 Geomip Terrain Meshes

When designing a level-of-detail system for voxel-based terrain, a rather obvious avenue to pursue is triangulating the voxel data at lower resolutions. Lower levels of detail are created by sampling the volumetric data at one-half resolution, one-quarter resolution, and so on. In order to prevent the amount of geometric data generated for each block from decreasing exponentially, we do not triangulate terrain blocks having  $8 \times 8 \times 8$  cells,  $4 \times 4 \times 4$  cells, etc. Instead, we double the physical size of a block each time the resolution is cut in half so that blocks always measure  $16 \times 16 \times 16$  cells at the resolution for which they are to be triangulated. This ensures that the average number of triangles rendered in each batch sent to the GPU is the same across all levels of detail, and it prevents the density of the number of meshes in viewport space from becoming too high.

Figure 4.1 shows a terrain surface for which one block’s triangle mesh is rendered in wireframe for each of three different detail levels. As done with texture mipmaps, we number the highest level of detail as LOD 0 and progressively lower levels of detail as LOD 1, LOD 2, and so on. Each block of a lower detail level precisely contains eight blocks sampled at double the resolution, and this leads to a natural octree structure that can be used for efficient visibility culling and detail level selection during rendering.

### 4.2.1 Surface Shifting

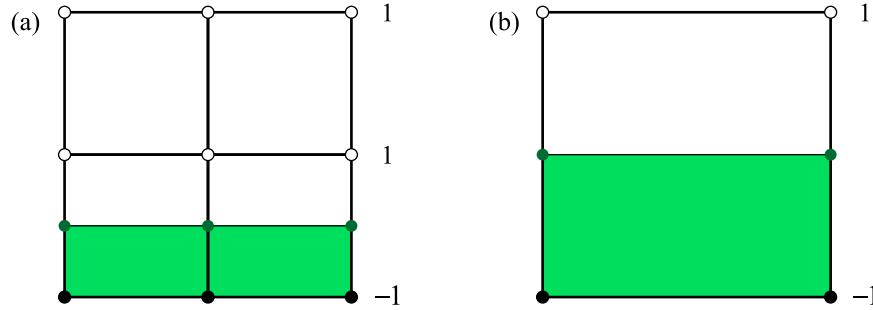
When triangulating a lower-resolution sampling of a voxel map, running the same Marching Cubes algorithm used for the highest LOD tends to produce meshes that are smoothed out and a poor match for their higher-resolution counterparts. The result is a very noticeable shift in the terrain surface whenever the level of detail rendered for part of a terrain surface changes. As exemplified



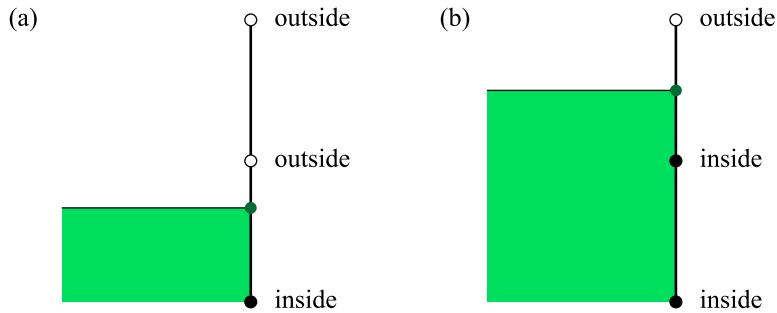
**Figure 4.1.** The dimensions of a terrain block belonging to a low-resolution detail level are twice as large in each direction as those of the next highest detail level. Shown here are meshes corresponding to terrain blocks of (a) the highest level of detail (LOD 0), (b) the first lower level of detail (LOD 1), and (c) the second lower level of detail (LOD 2). Images (b) and (c) show the meshes for the higher levels of detail for comparison purposes, but these are not actually part of the lower-detail meshes.

by Figure 4.2, this effect can be particularly bad even for flat ground because the terrain surface can shift as much as half of a cell width. Any structures placed on the terrain would suddenly appear sunken into the ground when the detail level changed, which is not acceptable.

To eliminate the terrain shifting problem, we must modify the way in which vertex coordinates are chosen when running Marching Cubes over low-resolution cells so that vertices belonging to low-resolution terrain meshes coincide exactly with vertices belonging to the highest-resolution mesh. This is accomplished by first examining the voxel sample at the midpoint of each active edge of a low-resolution cell. As shown in Figure 4.3, if a transition from inside to outside occurs on an



**Figure 4.2.** These are side views showing where ground triangles would be placed for cells of two different resolutions occupying the same space when the voxel values transition from  $-1$  to  $1$  along the vertical edges. In both (a) high-resolution cells and (b) low-resolution cells, the ground vertices are placed halfway between the voxel map sample locations by the Marching Cubes algorithm. The misalignment of these halfway points causes an undesirable shift in the terrain surface when detail levels change.



**Figure 4.3.** If a transition from inside to outside occurs along an edge in a low-resolution sampling of the voxel map, represented by the top and bottom sample locations, then such a transition can only occur in one half of the edge at double the resolution where the sample at the midpoint is considered. (a) The transition occurs in the bottom half of the edge because the center sample's inside/outside state does not match the bottom sample's state. (b) The transition occurs in the top half because the center sample's state does not match the top sample's state.

edge of a cell, then such a transition occurs inside one of the halves of the edge at double the resolution, but it cannot occur inside both halves. So we choose the sub-edge for which the transition does occur and continue recursively until the highest resolution is reached. We know a vertex is placed on the edge to which we ultimately converge in the highest-resolution terrain mesh, so we calculate the vertex position for the low-resolution mesh by applying Equations (3.4) and (3.5) to the endpoints of the highest-resolution sub-edge. This process is deterministic even when multiple transitions occur at LOD 2 or higher, and it guarantees that all vertices in lower LODs coincide exactly with vertices of the highest-resolution mesh. Note that the search described here only needs to be performed for vertices lying in the interior of an edge, and not for vertices placed at cell corners.

### 4.2.2 Alternate Triangulations

Although not particularly important for the highest level of detail, the specific placement of internal edges can make a significant difference when triangulating cells for lower levels of detail. For most of the equivalence classes utilized by our modified Marching Cubes algorithm, the triangulation shown in Table 3.2 is not the only legal triangulation that could be generated when cases belonging to those equivalence classes arise. The triangulation for each class is composed of one to four connected components that are topologically equivalent to convex polygons having between three and seven vertices, inclusive. For components having three vertices, we have no choice but to generate a single triangle. However, for components having a larger number of vertices, we do have some choice in the matter, and it may not be the case that the triangulation shown in Table 3.2 is the best option because it is a poor match for higher-resolution voxel data.

Even though they don't participate in the selection of the equivalence class to which a cell belongs, we still have access to the highest-resolution voxel samples inside each cell. Each of these voxel samples represents a signed distance to the terrain surface, so our goal is to choose the triangulation that minimizes the error between these values and the actual distances to the surface created by the triangulation. Since the interior edges are the only ones that can be changed, we only consider voxels that lie in the interior of a cell, and not those lying on a cell's boundary.

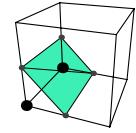
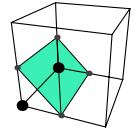
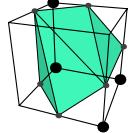
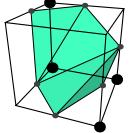
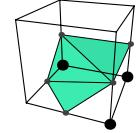
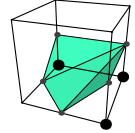
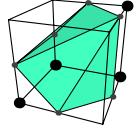
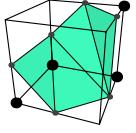
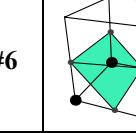
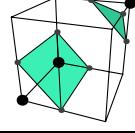
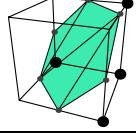
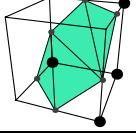
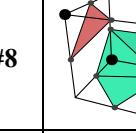
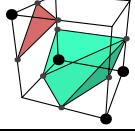
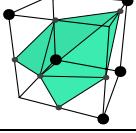
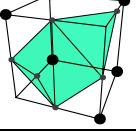
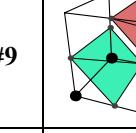
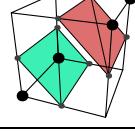
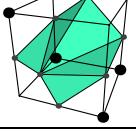
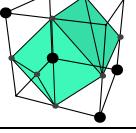
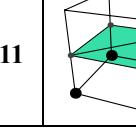
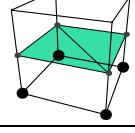
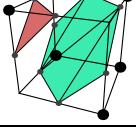
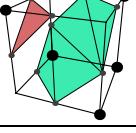
In the case that a triangulation component has four vertices, there are obviously two possible ways to connect vertices to form a triangulation. As the number of vertices increases, however, the number of possible triangulations grows very rapidly. In general, the number of ways to triangulate a convex polygon having  $n$  vertices is given by the Catalan number  $C_{n-2}$ , where

$$C_n = \frac{1}{n+1} \binom{2n}{n}. \quad (4.1)$$

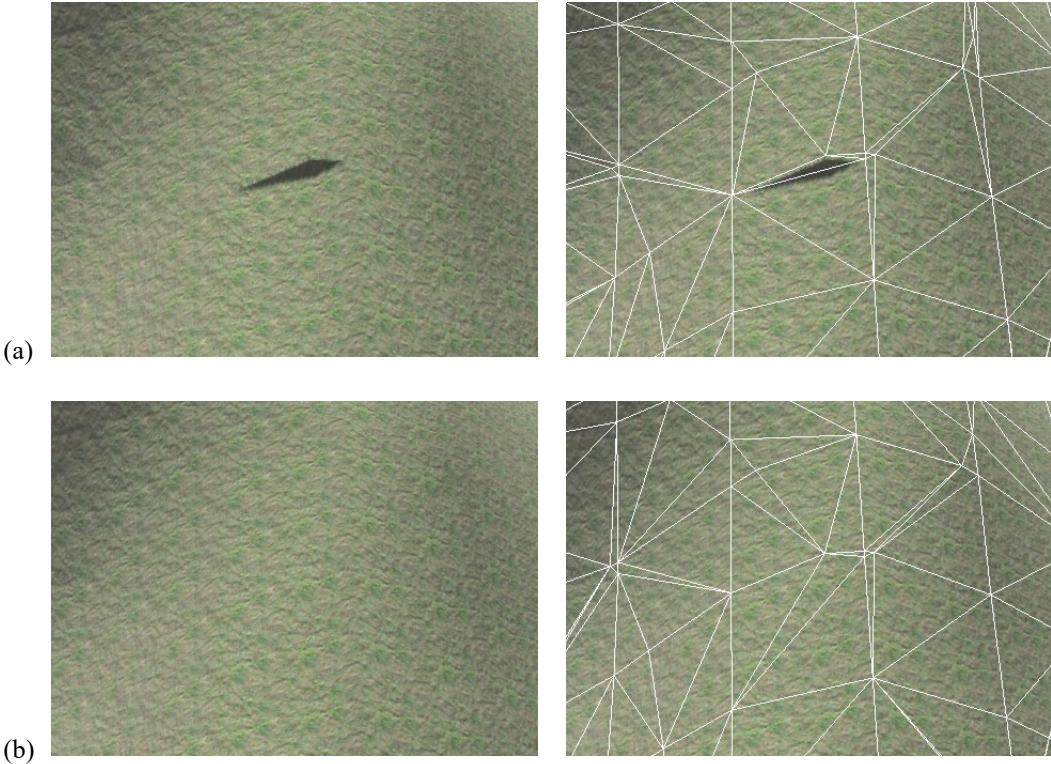
The values of  $C_n$  for  $n = 1, 2, 3, 4, 5$  are 1, 2, 5, 14, and 42, corresponding to the number of possible triangulations of polygons having 3, 4, 5, 6, and 7 vertices [Stanley 2001].

There are two equivalence classes for which 5 triangulations are possible, five classes for which 14 triangulations are possible, and one class for which 42 triangulations are possible. Searching for the best triangulation out of all the possibilities in these cases would have a negative impact on performance, so we instead make a compromise and limit the number of triangulations tested per cell to two. For components having five or more vertices, we choose the two triangulations to be as dissimilar as possible in terms of their principal directions of curvature. Table 4.1 lists the pairs of triangulations that we test for the 12 equivalence classes having multiple possibilities.

**Table 4.1.** These are the 12 equivalence classes for which multiple triangulations are possible. The two triangulations shown are the only ones that we test for a best match to the high-resolution voxel data, and they are chosen to be as dissimilar as possible in terms of their principal directions of curvature. The last column shows how many total triangulations are possible for each equivalence class.

Class Index	Primary Triangulation	Secondary Triangulation	Number Possible	Class Index	Primary Triangulation	Secondary Triangulation	Number Possible
#3			$C_2 = 2$	#12			$C_4 = 14$
#5			$C_3 = 5$	#13			$C_4 = 14$
#6			$C_2 = 2$	#14			$C_4 = 14$
#8			$C_3 = 5$	#15			$C_4 = 14$
#9			$C_2^2 = 4$	#16			$C_5 = 42$
#11			$C_2 = 2$	#17			$C_4 = 14$

Because lower levels of detail are typically rendered only when the camera is somewhat far away, which triangulation we assign to any particular cell only makes a subtle difference that can be directly observed in the terrain mesh. However, there are two indirect ways in which the ability to choose between triangulations does make a prominent difference. The first way involves the time at which the level of detail changes for a group of terrain blocks. There is a perceptible “pop” that occurs in the mesh when the detail level is abruptly changed, but the visible magnitude of this transition is reduced when the low-resolution triangulation is a better match for the high-resolution surface. The second way involves the appearance of shadowing artifacts. As shown in Figure 4.4(a), a very noticeable artifact can appear when a poor choice of triangulation is made because it can

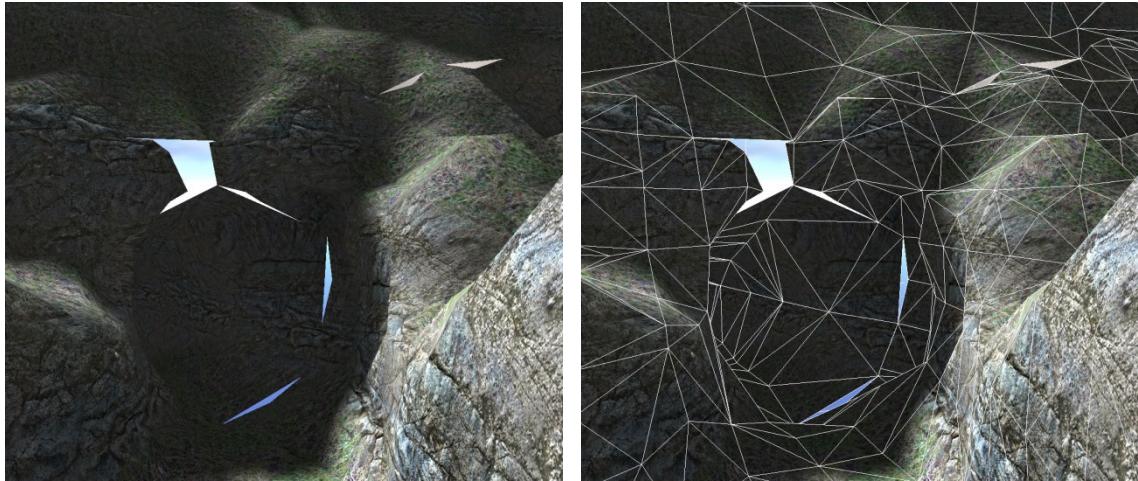


**Figure 4.4.** (a) A prominent shadow artifact appears in a low-resolution mesh, shown with and without wireframe, because a poor choice of triangulation creates a concavity that blocks light. (b) Choosing the best triangulation from Table 4.1 eliminates the artifact.

create a concavity that produces an unwanted shadow when aligned in an unfavorable manner with the light source. Choosing the best triangulation from Table 4.1 helps eliminate these artifacts by creating a smoother low-resolution terrain mesh, as shown in Figure 4.4(b).

### 4.3 Transition Cells

Of course, rendering terrain blocks of differing resolutions next to each other produces cracks along the mesh boundaries, as shown in Figure 4.5. These cracks can be quite large in spite of the fact that vertices along the boundary in the low-detail mesh always coincide exactly with vertices in the high-detail mesh. The most robust solutions to this problem are called “stitching” techniques, and they work by filling the cracks with additional triangles that seamlessly join the border triangles of the adjacent meshes. Stitching techniques are easily implemented in the realm of height-based terrain, but the analogous crack-filling problem for voxel-based terrain is vastly more complicated due to the topographical freedom allowed by the underlying volumetric data. As an example, a giant hole appears in Figure 4.5 where a low-resolution surface is roughly tangent to the planar

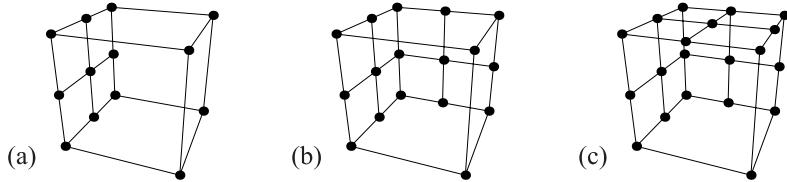


**Figure 4.5.** Adjacent terrain blocks whose meshes are rendered at two different resolutions are shown with and without wireframe. Cracks are visible along the boundary between the blocks, allowing the sky in the background to show through, and a large hole appears where the surface is nearly tangent to the shared faces of the blocks.

boundary of a high-resolution block, and this creates a topography that cannot exist in a height-based terrain mesh. Existing stitching methods can be carried over to voxel-based terrain only under very limited conditions, and none of them are capable of handling all of the cases that can arise in an unrestricted voxel map. In this section, we introduce a new stitching method that solves the general crack-filling problem for multiresolution voxel-based surface meshes.

Let us consider two adjacent blocks of voxel data such that one block is sampled at half the resolution of the other block. We call a cell inside the half-resolution block that lies along the border with the full-resolution block a *transition cell*, and we see that a triangulation compatible with the Marching Cubes algorithm for any such cell must account for a total of 13 sample values lying on its boundary. As shown in Figure 4.6(a), nine of these samples come from the full-resolution data, and the remaining four come from the half-resolution data. Classifying each of these sample values as either inside or outside the solid space produces  $2^{13} = 8192$  possible cases to be triangulated. This is quite large compared to the 256 cases arising in a Marching Cubes implementation for a single resolution, but there is no reason to believe that an analogous algorithm using a 13-bit lookup table would not produce a satisfactory and robust result.

However, the half-resolution block can be bordered on more than one side by full-resolution blocks. If we further consider a single cell in the half-resolution block that is bordered by full-resolution cells on two adjacent faces, then a triangulation of that cell must account for the 17 sample values shown in Figure 4.6(b), giving rise to  $2^{17} = 131,072$  distinct cases. Finally, if a half-

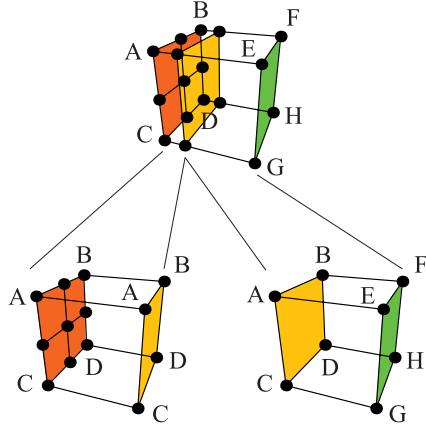


**Figure 4.6.** The points shown on the boundaries of these cubes illustrate the sample positions for a single cell in a half-resolution block when it is bordered by a full-resolution block on (a) one side, (b) two sides, and (c) three sides.

resolution cell is bordered on three adjacent faces by full-resolution cells, a triangulation must account for the 20 sample values shown in Figure 4.6(c), resulting in a whopping  $2^{20} = 1,048,576$  distinct cases. To implement an algorithm that would connect the full-resolution volume to the half-resolution volume by triangulating all possible cases arising within individual transition cells along the border between them, we would have to account for all  $2^{13} + 2^{17} + 2^{20} = 1,187,840$  configurations.

From an engineering perspective, determining the set of topological equivalence classes for the nearly 1.2 million cases and then generating a lookup table containing triangulation data for all of those cases is a monumentally tedious job. Furthermore, even if such a lookup table could be easily constructed, it would require dozens of megabytes of storage space. This size by itself could be prohibitively large on some platforms, but a more important issue is the high number of data cache misses that an algorithm using this lookup table would experience on any platform, severely affecting performance. In light of these disadvantages to the implementation of a direct analog of Marching Cubes for transition cells, we are compelled to search for some way of simplifying the fundamental problem at hand.

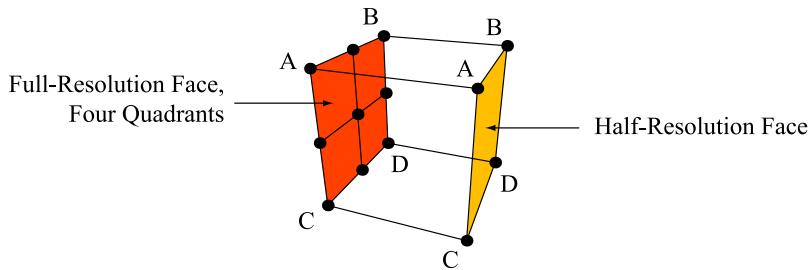
Ideally, we would like the number of distinct triangulation cases for a transition cell to be on the same order of magnitude as the number of cases arising in the conventional Marching Cubes algorithm, and we would prefer that the number of possible cases be the same for all transition cells. Both of these properties can be achieved by dividing each transition cell into two smaller cells in the manner illustrated by Figure 4.7. The left cell is triangulated using sample values only from the face bordering the full-resolution block, and the four corner values labeled A, B, C, and D in the figure are duplicated on the opposite face of the cell. The transition from full resolution to half resolution takes place entirely inside this cell, for which we now have a very manageable nine distinct sample values to consider. The right cell is triangulated in the conventional manner using only the eight sample values from the half-resolution data.



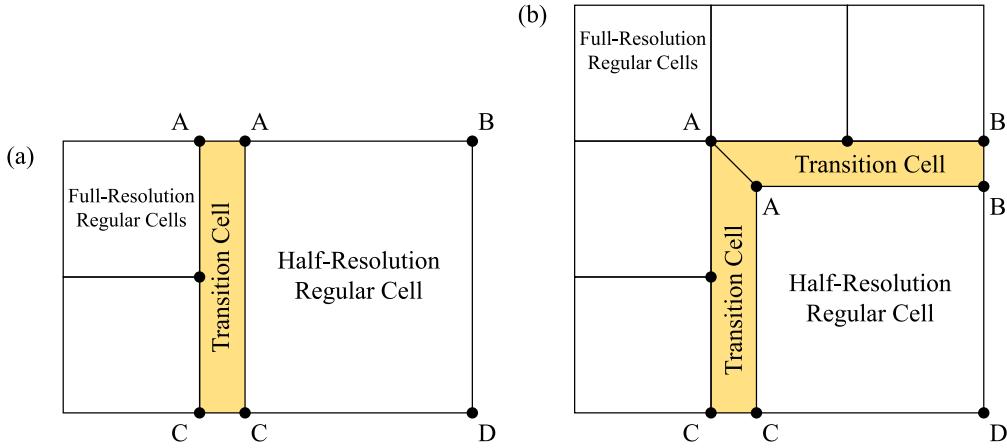
**Figure 4.7.** A transition cell is divided into two parts along a plane parallel to the boundary face between full-resolution and half-resolution blocks. Our new algorithm triangulates the left part using sample values only appearing on the full-resolution face. The right part is triangulated with the conventional modified Marching Cubes algorithm.

We call cells (at any resolution) that are influenced by eight corner voxels and triangulated using the modified Marching Cubes algorithm *regular* cells. We redefine the meaning of a *transition* cell to be one of the form shown in Figure 4.8, which is influenced by nine voxels lying on a single face. In a transition cell, we call the face on which nine sample values appear the *full-resolution* face. The face opposite the full-resolution face is called the *half-resolution* face, and the sample values at its four corners are identical to those of the corresponding corners of the full-resolution face. The remaining four faces are called *lateral* faces.

The width of a transition cell (i.e., the distance between the full-resolution and half-resolution faces) is a parameter that can be freely adjusted at the global level. The triangles that we construct inside a transition cell to connect voxel data of different resolutions is simply scaled to the proper width. The transition cell width is normally chosen to be a large fraction of the size of a full-



**Figure 4.8.** This cube represents the topological layout of a transition cell. It has a full-resolution face (shown in red) that is divided into four quadrants corresponding to four full-resolution regular cells. Opposite the full-resolution face is the half-resolution face (shown in orange), and its corner voxel sample values are the same as the corresponding corners of the full-resolution face. The remaining four faces of the cube are the lateral faces.



**Figure 4.9.** These are two-dimensional slices showing what is done in the cases in which a half-resolution cell is bordered by full-resolution cells on (a) one side and (b) two sides. All regular cells are triangulated using the conventional modified Marching Cubes algorithm, and all transition cells have the same data point configuration.

resolution cell. It is possible to use a width of zero and still produce results that seamlessly stitch multiresolution meshes together, but this width leads to severe shading problems (see Chapter 5).

Transition cells always have the same configuration with respect to the number and location of their voxel samples, and this is true even when a half-resolution block is bordered by full-resolution blocks on two or three sides. Figure 4.9(a) shows a two-dimensional slice of a transition cell connecting full-resolution regular cells to a half-resolution regular cell on one face, and Figure 4.9(b) shows two transition cells connecting full-resolution regular cells to a half-resolution regular cell on two adjacent faces. In the case of a half-resolution regular cell connected full-resolution regular cells on three adjacent faces, the configuration is an analogous extension of that shown for two faces. The transition cells like those shown in Figure 4.9(b) are no longer boxes whose edges always meet at right angles, but they are still topologically equivalent to the cube shown in Figure 4.8.

The transition cell triangulation problem has now been reduced to one for which there are  $2^9 = 512$  possible cases to handle. It is natural for us to attempt to identify a set of equivalence classes whose members are related through an orthogonal transform in the same spirit as the Marching Cubes algorithm. Since the nine voxel sample values influencing a transition cell lie in a square region of a plane, we determine the set of equivalence classes by considering the action of the dihedral group  $D_8$  on the sample locations of the individual cases.<sup>7</sup> Given a particular case  $x$ , the other members of the equivalence class containing  $x$  are precisely the cases belonging to the orbit

<sup>7</sup> The *dihedral group*  $D_8$  is the set of operations mapping a square onto itself. It is a non-Abelian group consisting of 8 elements that represent combinations of rotations and reflections.

of  $x$  under the group action.<sup>8</sup> The size of the equivalence class containing  $x$  is given by the index<sup>9</sup> of the stabilizer subgroup of  $x$  in  $D_8$ .<sup>10</sup> (Consequently, the size of each equivalence class must divide 8.) Because  $D_8$  includes reflections, the equivalence relation induced by the group action equates cases that are mirror images of each other.

If a case  $x$  does not have an ambiguous half-resolution face and does not have any ambiguous quadrants on its full-resolution face, then we also add an inversion operation to the group acting on  $x$  that reverses the inside/outside state of all nine sample values. For these cases, the group inducing our equivalence classes is isomorphic to  $D_8 \times \mathbb{Z}_2$ , and the net effect is that the sizes of the equivalence classes are doubled.

Observing the orbits of each of the 512 cases under the action of the group  $D_8$  or  $D_8 \times \mathbb{Z}_2$ , as appropriate, yields exactly 73 distinct equivalence classes. One equivalence class is the trivial class containing the two cases for which the inside/outside state of all nine sample values is the same. The 72 nontrivial classes are enumerated in Tables 4.2 through 4.7, where all of the cases appearing in a particular table are related in some way. For each table entry, the number in the lower-left corner is the class index (that we have assigned somewhat arbitrarily), and the number in the lower-right corner is the number of cases belonging to the equivalence class. A black dot indicates a corner that is inside solid space, and corners without a dot are outside. Green triangles are front-facing with respect to the view point, and red triangles are back-facing.

Table 4.8 lists the 51 transition cell equivalence classes having four or fewer sample values classified as inside solid space. For equivalence classes that do not include geometric inverses due to the presence of an ambiguous face, the inverse equivalence class is listed, and this accounts for the remaining 22 classes. For the representative case  $x$  shown for each equivalence class, Table 4.8 also identifies the stabilizer subgroup  $H = G_x$ , where  $G \cong D_8 \times \mathbb{Z}_2$  for classes that include geometric inverses, and  $G = D_8$  for those that do not. The stabilizer subgroup is expressed in terms of the generators  $r$  and  $f$  of the group  $D_8$ , where we have chosen  $r$  to be the 90-degree counterclockwise rotation and  $f$  to be the flip about the horizontal axis. Equivalence classes containing cases that exhibit no symmetry are the largest because the stabilizers are simply the subgroup  $\{e\}$  containing only the identity element  $e$ .

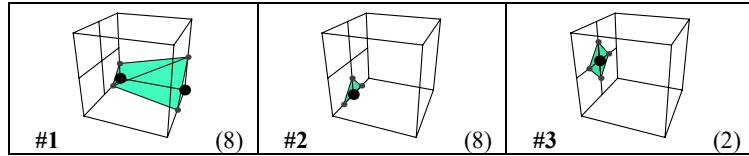
---

<sup>8</sup> The *orbit* of  $x$  under the action of a group  $G$  is the set  $\{g \cdot x \mid g \in G\}$ . That is, the orbit of  $x$  is the set of all elements reachable by acting on  $x$  with elements of  $G$ .

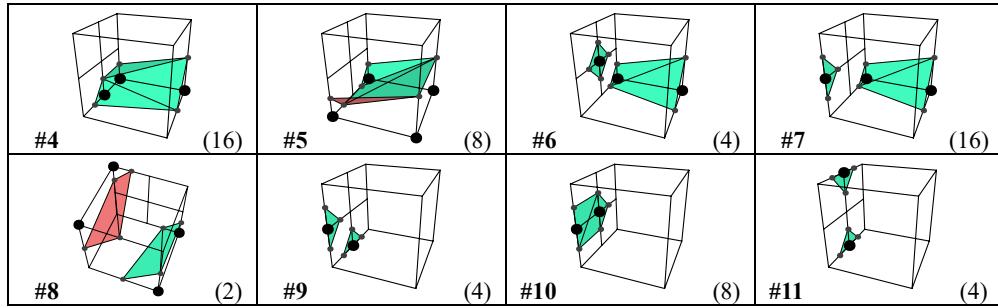
<sup>9</sup> The *index* of a subgroup  $H$  in a group  $G$ , written  $|G : H|$ , is equal to the number of cosets of  $H$  in  $G$ . For finite groups,  $|G : H| = |G|/|H|$ .

<sup>10</sup> The *stabilizer subgroup* of  $x$  in a group  $G$ , denoted by  $G_x$ , is the group  $\{g \in G \mid g \cdot x = x\}$ . That is, the stabilizer is the subgroup in  $G$  that fixes  $x$ .

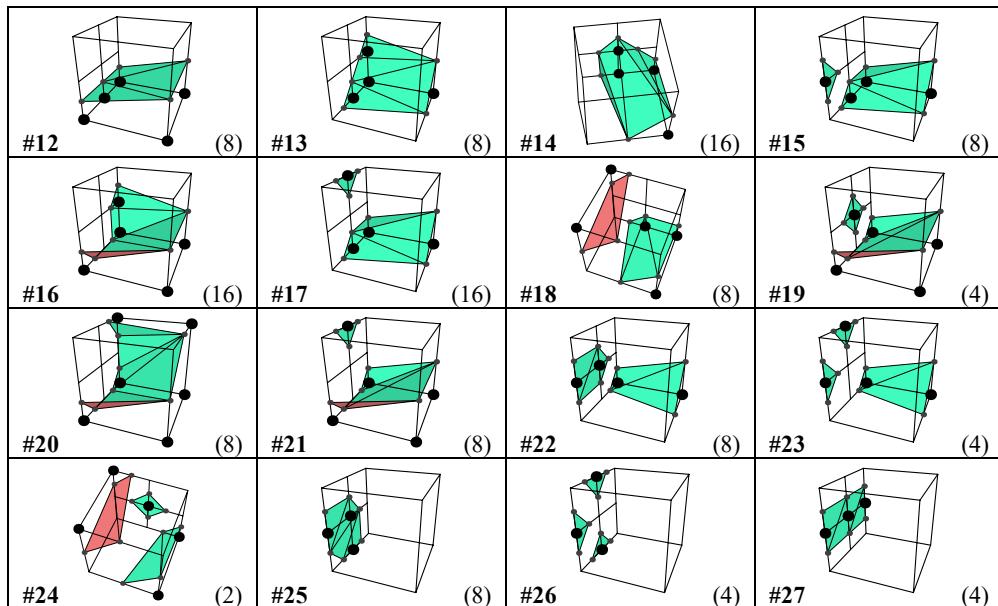
**Table 4.2.** These are the 3 transition cell equivalence classes for which exactly one sample value is inside solid space. These classes represent 18 of the 512 distinct cases.



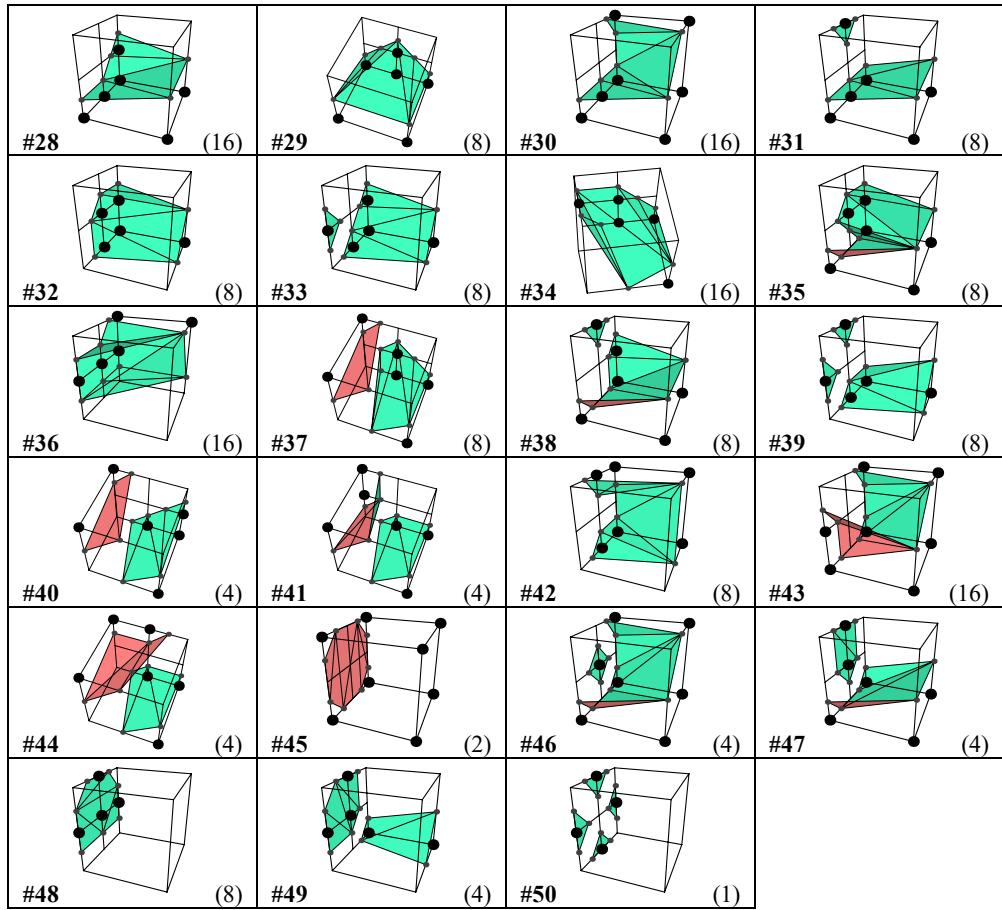
**Table 4.3.** These are the 8 transition cell equivalence classes for which exactly two sample values are inside solid space. These classes represent 62 of the 512 distinct cases.



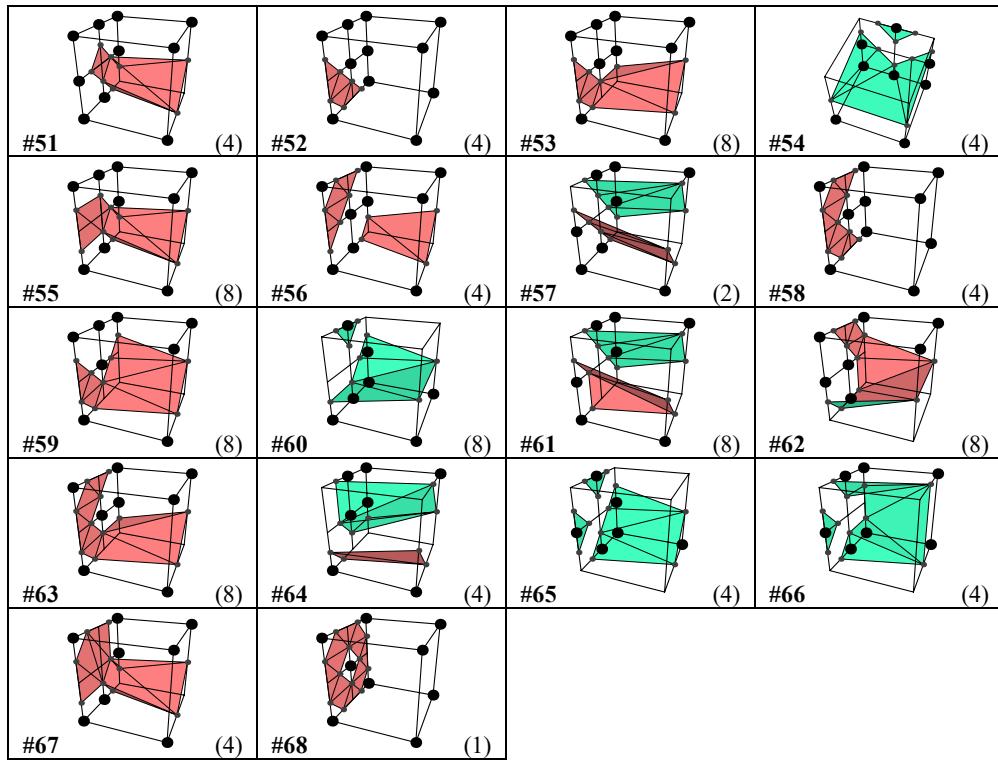
**Table 4.4.** These are the 16 transition cell equivalence classes for which exactly three sample values are inside solid space. These classes represent 130 of the 512 distinct cases.



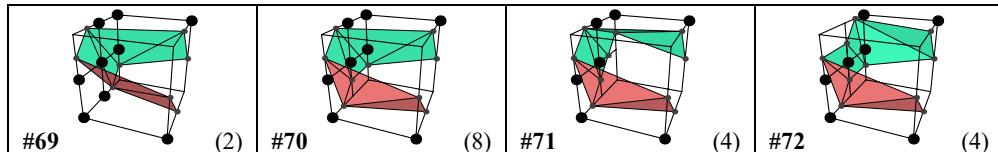
**Table 4.5.** These are the 23 transition cell equivalence classes for which exactly four sample values are inside solid space. These classes represent 187 of the 512 distinct cases.



**Table 4.6.** These are the 18 transition cell equivalence classes corresponding to the inverses of classes appearing in Tables 4.3 through 4.5 that have at least one ambiguous quadrant on the full-resolution face. These classes represent 95 of the 512 distinct cases.



**Table 4.7.** These are the 4 transition cell equivalence classes corresponding to the inverses of classes appearing in Tables 4.3 through 4.5 for which the half-resolution face is ambiguous, but no full-resolution quadrants are ambiguous. These classes represent 18 of the 512 distinct cases.



**Table 4.8.** This table lists all 73 transition cell equivalence classes and identifies the stabilizer subgroup  $H$  for the representative element shown for each class. For classes in Tables 4.2 through 4.5 that do not include geometric inverses, the associated inverse equivalence class from Table 4.6 or 4.7 is indicated. The size of each equivalence class is given by  $|G : H|$ , where  $G = D_8 \times \mathbb{Z}_2$  for classes that include geometric inverses, and  $G = D_8$  for those that do not.

Class	Inverse class	Stabilizer subgroup $H$	Class size
#0 		$D_8$	2
#1 		$\langle r^3 f \rangle$	8
#2 		$\langle r^2 f \rangle$	8
#3 		$D_8$	2
#4 		$\{e\}$	16
#5 		$\langle r^2 f \rangle$	8
#6 	#51 	$\langle r^3 f \rangle$	4
#7 		$\{e\}$	16
#8 	#69 	$\langle r^2, rf \rangle$	2
#9 	#52 	$\langle rf \rangle$	4
#10 		$\langle f \rangle$	8
#11 		$\langle r^2 f \rangle$	4
#12 		$\langle r^2 f \rangle$	8
#13 		$\langle r^3 f \rangle$	8
#14 		$\{e\}$	16
#15 	#53 	$\{e\}$	8
#16 		$\{e\}$	16
#17 		$\{e\}$	16
#18 	#70 	$\{e\}$	8
#19 	#54 	$\langle r^2 f \rangle$	4
#20 		$\langle r^3 f \rangle$	8
#21 		$\langle r^2 f \rangle$	8
#22 	#55 	$\{e\}$	8
#23 	#56 	$\langle r^3 f \rangle$	4
#24 	#57 	$\langle r^2, rf \rangle$	2
#25 		$\langle rf \rangle$	8
#26 	#58 	$\langle f \rangle$	4
#27 		$\langle r^2, f \rangle$	4
#28 		$\{e\}$	16
#29 		$\langle r^2 f \rangle$	8
#30 		$\{e\}$	16
#31 		$\langle r^2 f \rangle$	8
#32 		$\langle r^3 f \rangle$	8
#33 	#59 	$\{e\}$	8
#34 		$\{e\}$	16
#35 	#60 	$\{e\}$	8
#36 		$\{e\}$	16
#37 	#61 	$\{e\}$	8

**Table 4.8.** (Continued.)

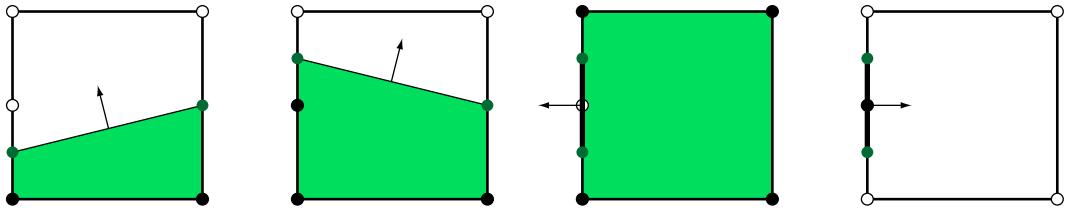
Class	Inverse class	Stabilizer subgroup $H$	Class size		
#38		#62		$\{e\}$	8
#39		#63		$\{e\}$	8
#40		#71		$\langle r^3 f \rangle$	4
#41		#64		$\langle rf \rangle$	4
#42				$\langle f \rangle$	8
#43				$\{e\}$	16
#44		#72		$\langle r^2 \rangle$	4

Class	Inverse class	Stabilizer subgroup $H$	Class size		
#45			$D_8$	2	
#46		#65		$\langle r^3 f \rangle$	4
#47		#66		$\langle r^2 f \rangle$	4
#48				$\langle r^2 f \rangle$	8
#49		#67		$\langle r^3 f \rangle$	4
#50		#68		$D_8$	1

As with our modified Marching Cubes algorithm, the triangulations we use for transition cells follow a consistent set of rules. Clearly, we must match edges occurring on the regular cells adjacent to a transition cell on both the full-resolution and half-resolution faces in order to ensure a continuous mesh. For any nontrivial quadrant of the full-resolution face, open edges must be placed within the quadrant at the locations specified by Figure 3.5. The same is true for the half-resolution face when it is nontrivial.

What remains are the lateral faces of a transition cell. Any open edges placed on a lateral face must be matched by an edge having the same endpoints on the coincident lateral face of an adjacent transition cell, and this edge must have the opposite winding direction. Figure 4.10 shows the four possible nontrivial configurations of a lateral face in terms of the inside/outside state of the three voxel sample values affecting it. In the two configurations for which the sample states alternate, either inside-outside-inside or outside-inside-outside, a mesh edge is placed on the boundary edge between the lateral face and the full-resolution face, and it is thus not connected to the half-resolution face.

Unlike most of the triangulations for regular cells, many of the triangulations for transition cells contain internal edges that lie entirely on the full-resolution face, and they can span multiple



**Figure 4.10.** These are the only four nontrivial edge configurations allowed on a lateral face of a transition cell modulo a vertical flip. The left edge represents the full-resolution face, and the right edge represents the half-resolution face. Sample locations with solid dots are classified as inside, and sample locations with open dots are classified as outside. The green region represents solid space, and the arrows represent the outward surface normal direction along the edges. In the two rightmost configurations, the edge lies on the boundary between the lateral face and the full-resolution face.

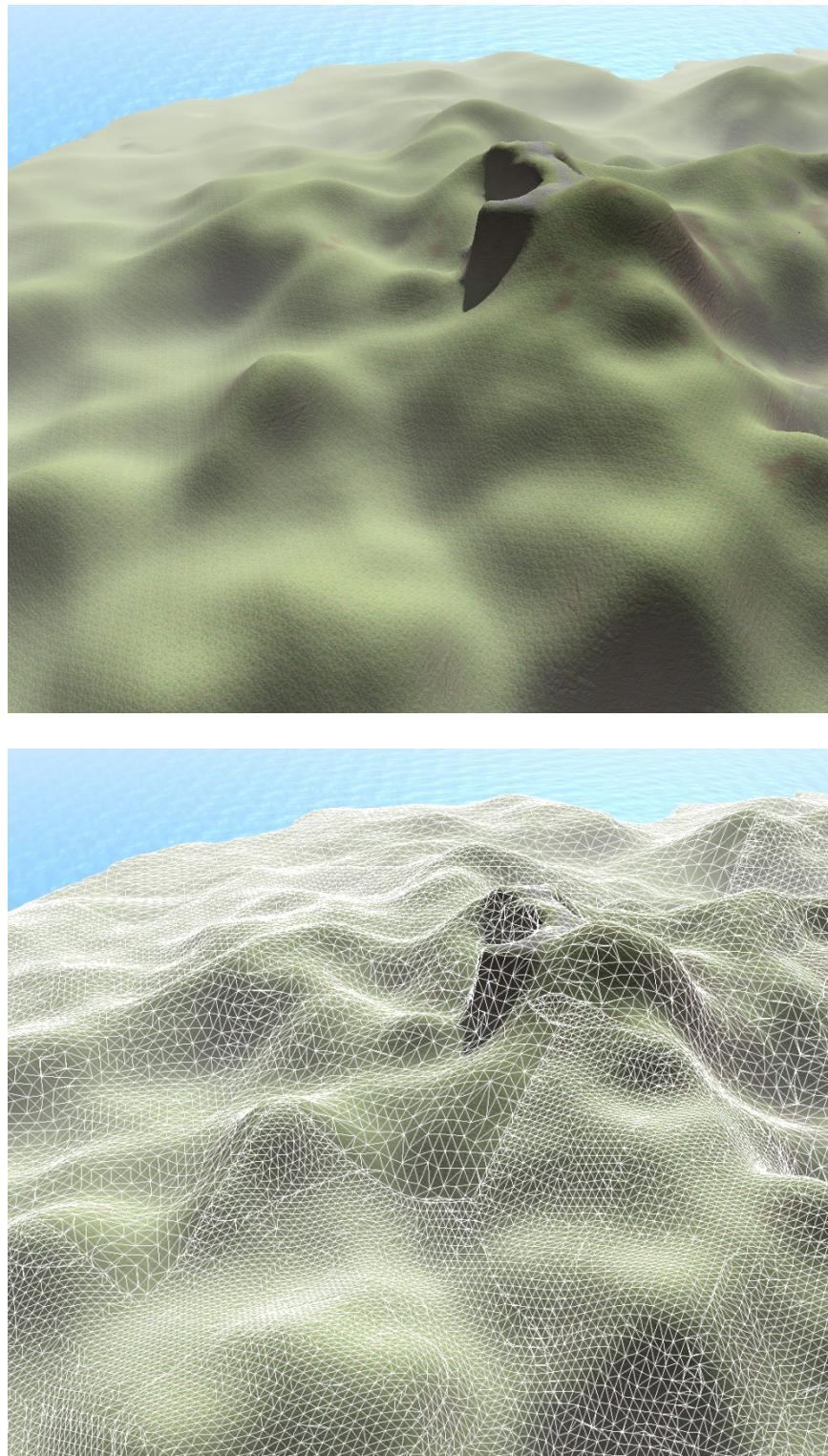
quadrants.<sup>11</sup> The placement of these edges on the boundary of the transition cell cannot be avoided without introducing additional vertices. In most cases, these edges connect two triangles that both lie in the full-resolution face and form part of a cap for features in the full-resolution voxel map that exceed the sampling frequency in the half-resolution voxel map. The four classes shown in Table 4.7 are the only exceptions to this pattern, and they each contain internal edges connecting triangles lying in the full-resolution face with triangles that span the width of the transition cell. These triangulations form a saddle-like surface in the local region of the overall mesh.

## 4.4 Transition Rendering

A large terrain surface is composed of a hierarchy of blocks measuring  $16 \times 16 \times 16$  cells, and these blocks are arranged in an octree for which each level corresponds to a separate mesh resolution. Each block not of the highest level of detail neatly contains exactly eight blocks of the next-highest resolution. The octree is traversed when the terrain is rendered, and any block not intersecting the view frustum is culled along with its entire subtree. When a block is visited in the octree and determined to be visible to the camera, its projected size in the viewport is calculated. If the size falls below a threshold value, then that block is rendered, and its subtree is skipped. This process efficiently selects the proper levels of detail for the entire terrain surface, and the threshold value can be adjusted to control at what distances LOD transitions occur. An example scene using this process is shown in Figure 4.11.

---

<sup>11</sup> The only exceptions for which an internal edge of a regular cell does lie entirely on its boundary are the secondary triangulations for classes #15 and #16. See Table 4.1.



**Figure 4.11.** A large region of terrain, rendered with and without wireframe. Three levels of detail are visible, and transition cells can be seen between blocks of differing resolutions. The width of a transition cell is half of the size of the adjacent full-resolution regular cell.

The triangles belonging to transition cells should only be rendered between adjacent blocks of differing resolutions. Each block selected for rendering needs to be aware of which of its siblings in the octree have also been selected for rendering, and the transition cells occurring along the boundaries with those blocks must be disabled. Thus, for all but the highest-resolution blocks, the mesh generated for a block is composed of up to seven distinct parts: the primary mesh for the block and a transition mesh for each of the six faces of the block on which nontrivial transition cells occur. The primary mesh is always rendered, but the transition meshes are only rendered under specific conditions.

Regular cells that occur along the boundary of a low-resolution block can be rendered in one of two states. If an adjacent block is rendered at the same level of detail, then the regular cells along the boundary are rendered normally with no transition cells in between. If adjacent blocks are rendered at different levels of detail, however, then regular cells in the block for the lower level of detail must be scaled in one or more directions to make space for transition cells as illustrated in Figure 4.9. In order to avoid extra computation in the vertex program, we store two positions for each vertex belonging to regular cells on the boundary of a low-resolution block, a primary position used when transition cells are not rendered, and a secondary position used when transition cells are rendered.

Secondary vertex positions can be calculated by linearly transforming positions inside boundary cells so that the full-size cell is scaled to a smaller size that allows space for between one and three transition cells, as necessary, depending on the location with respect to the edges and corners of the entire block. This can be accomplished by computing offsets  $(\Delta x, \Delta y, \Delta z)$  for the coordinates  $(x, y, z)$  in any boundary cell using the formula

$$\Delta x = \begin{cases} (1 - 2^{-k})w(k), & \text{if } x < 2^k; \\ 0, & \text{if } 2^k \leq x \leq 2^k(s-1); \\ (s-1 - 2^{-k})w(k), & \text{if } x > 2^k(s-1), \end{cases} \quad (4.2)$$

where  $k$  is the LOD index, and  $s$  is the size of a block in one dimension when measured in cells (e.g., if a block measures  $16 \times 16 \times 16$  cells, then  $s = 16$ ). The function  $w(k)$  specifies the width of transition cells for LOD index  $k$ , and it is defined as  $w(k) = 2^{k-2}$  in our implementation. Formulas for  $\Delta y$  and  $\Delta z$  are similar to Equation (4.2).

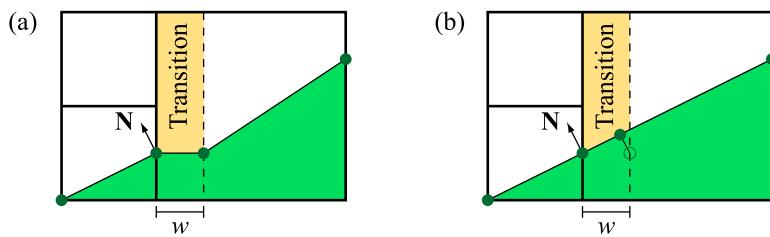
Along a block face on which a transition is occurring between levels of detail, we apply Equation (4.2) to all vertices generated for the regular cells and to any vertices generated on the half-resolution faces of transition cells. Vertices generated on the full-resolution faces of transitions cells

are not moved. Adding the offsets directly to the original vertex positions has the effect of creating a flattened strip of triangles in the transition region as shown in Figure 4.12(a), and it deforms the surface in the regular cell such that unwanted concavities can be created. These problems can be eliminated by projecting the offset vector for a vertex onto the tangent plane passing through the vertex's original position to obtain the new vertex position  $(x', y', z')$  using the formula

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} 1 - N_x^2 & -N_x N_y & -N_z N_x \\ -N_x N_y & 1 - N_y^2 & -N_y N_z \\ -N_z N_x & -N_y N_z & 1 - N_z^2 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix}, \quad (4.3)$$

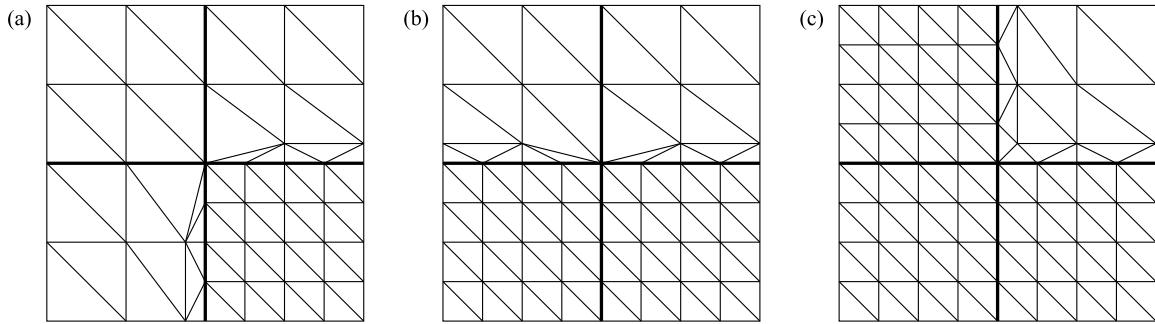
where  $\mathbf{N}$  is the unit-length vertex normal. The effect of the projection given by Equation (4.3) is shown in Figure 4.12(b).

In addition to the primary and secondary positions, we also store a three-component value for each vertex that indicates which faces of the block the vertex is near. The value stored for each component corresponds to one of three states possible for each coordinate of the primary vertex position: near the positive face, near the negative face, or near neither of those faces. When a block is selected for rendering, six boolean values are calculated to indicate whether the adjacent block at the same level of detail is also being rendered, and these values are made available to the vertex program. If the vertex program determines that none of the blocks are being rendered at the same level of detail adjacent to faces for which a vertex is near, then it selects the secondary vertex position because all transition cells affecting the vertex are being rendered. If any of the adjacent blocks adjacent to faces for which a vertex is near are being rendered at the same level of detail as the block owning the vertex, then the primary position is selected.



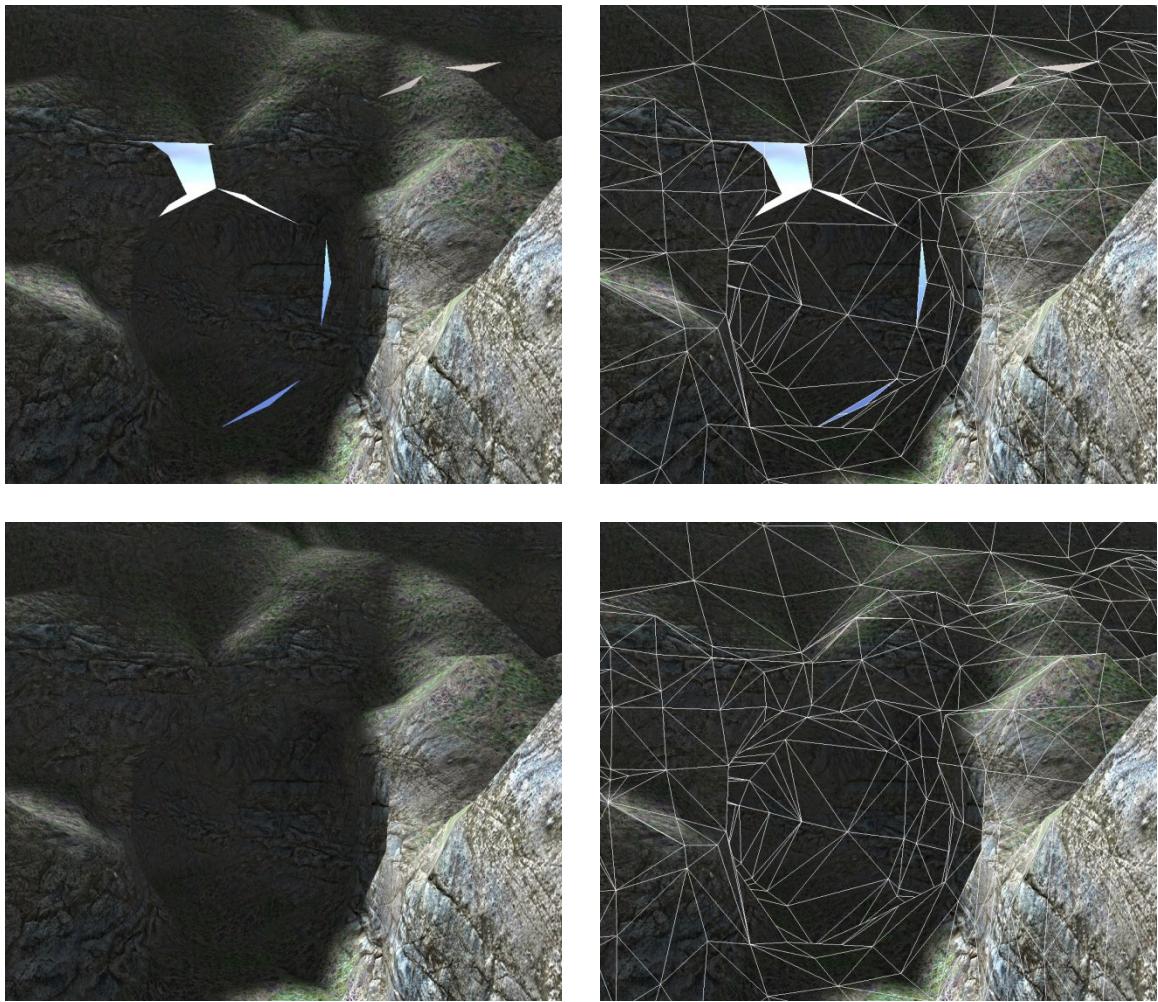
**Figure 4.12.** (a) The linear transformation given by Equation (4.2) is applied to vertices in the low-resolution cell to make space for a transition cell, but this has the unwanted effects of creating a flattened region and a concavity. (b) Using Equation (4.3) to project the difference in vertex positions onto the tangent plane with respect to the vertex normal  $\mathbf{N}$  eliminates these problems.

This procedure for selecting between the primary and secondary vertex positions tends to cause the primary vertex position to be used at corners where blocks meet and more than one block is rendered at the lower level of detail. Figure 4.13 shows examples in the case where four blocks meet and are rendered at different levels of detail. In images (a) and (b), the primary vertex position is always selected at the corner location because not all adjacent blocks are being rendered at the higher resolution. In image (c), the secondary vertex position is selected since all three adjacent blocks are rendered at the higher resolution.

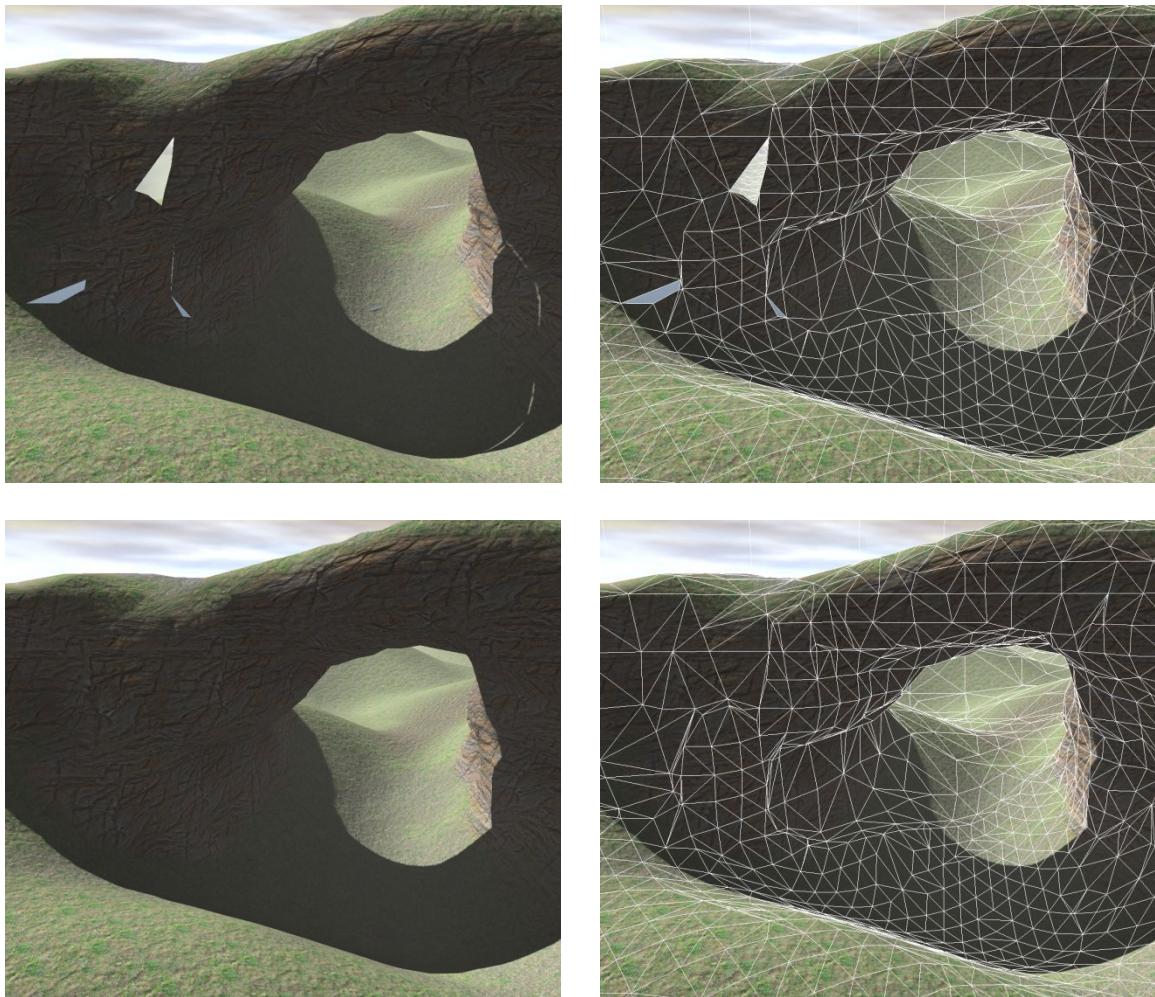


**Figure 4.13.** These are the three possible transitions that can take place at the corner where four blocks meet and are not all rendered at the same level of detail. All of the transition cells shown here use equivalence class #12 from Table 4.4. (a) Three blocks are rendered at low resolution, and one is rendered at high resolution. (b) Two adjacent blocks are rendered at low resolution, and the other two are rendered at high resolution. (c) One block is rendered at low resolution, and three are rendered at high resolution.

The results produced by adding transition cells to our terrain and following the rendering procedure described above are shown in Figures 4.14 and 4.15. The triangles generated by transition cells stitch together all the cracks, seams, and holes arising from mismatched edges between blocks rendered at differing resolutions that were first demonstrated in Figure 4.5. Furthermore, these stitching triangles are created in such a way that a smooth surface is maintained, avoiding potential texturing and shading artifacts.



**Figure 4.14.** The example cave entrance from Figure 4.5 is shown again in the top row, with and without wireframe. In the bottom row, triangles generated by transition cells have been added, and all artifacts resulting from mismatches between blocks of differing resolutions have been eliminated.



**Figure 4.15.** The detail level changes at the block boundary near the left side of this arch. No transition cells are rendered in the top row, and large artifacts can be seen both directly and in the shadow cast by the terrain mesh on the right side of the image. In the bottom row, transition cells are rendered to stitch together the different levels of detail.

## 4.5 Implementation Details

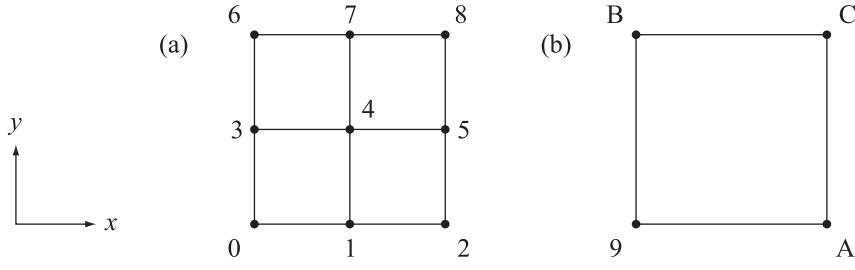
For any block representing a voxel sampling resolution less than the highest level of detail, we need to generate a triangle mesh corresponding to the layer of transition cells situated upon each of the six block faces. (In our implementation, a block face measures  $16 \times 16$  cells.) We call the collection of all triangles generated for any single block face the *transition mesh* for that face. A typical block does not have nontrivial transition cells on all six faces, so it is usually the case that fewer than six transition meshes are generated. As discussed in Section 4.4, each transition mesh can be individually enabled or disabled for rendering depending on whether higher-resolution blocks are being rendered in the position adjacent to the corresponding face.

An implementation that efficiently generates a transition mesh shares many similarities with the implementation of the modified Marching Cubes algorithm used to generate meshes for the regular cells of a block, described in Section 3.3. The biggest difference is that transition mesh generation is a two-dimensional process since the layer of transition cells on a block face is only one cell thick. We describe the process here for the positive  $z$  block face on which the transition cells are arranged on a grid parallel to the  $x$ - $y$  plane. For the remaining five faces, the coordinate basis is adjusted appropriately.

Transition meshes are generated for a block either in parallel with the block's interior mesh or immediately after the block's interior mesh, so the voxel data needed for the transition cells is already available in memory. On a single block face, we visit each of the 256 transition cells in order of increasing local  $x$  coordinate, followed by increasing  $y$  coordinate. We maintain a history buffer in memory large enough to store reusable vertex positions for two rows, the current row and the preceding row.

Within a transition cell, the voxel sample locations are numbered separately for the full-resolution face and half-resolution face as shown in Figure 4.16. Even though the four corner locations are numbered differently on the two faces, the corresponding voxel values are always equal. That is, the voxel values for locations 0 and 9 are the same, as are those for locations 2 and A, 6 and B, and 8 and C.

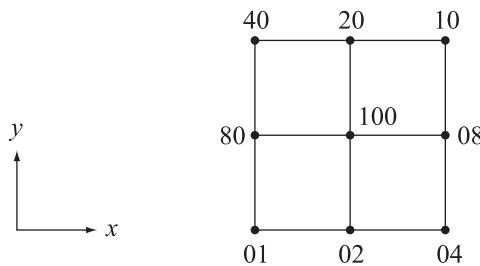
The case index for the transition cell is assembled from the sign bits of the voxel values at the nine locations on the full-resolution face in a manner similar to that of the modified Marching Cubes implementation. The bits composing the case index are given by the hexadecimal quantities shown in Figure 4.17. These quantities are summed for the locations where the voxel values are negative to construct the 9-bit case index. (This particular order was chosen because the lowest two nibbles of the case index are transposed under a 180-degree rotation of the cell.)



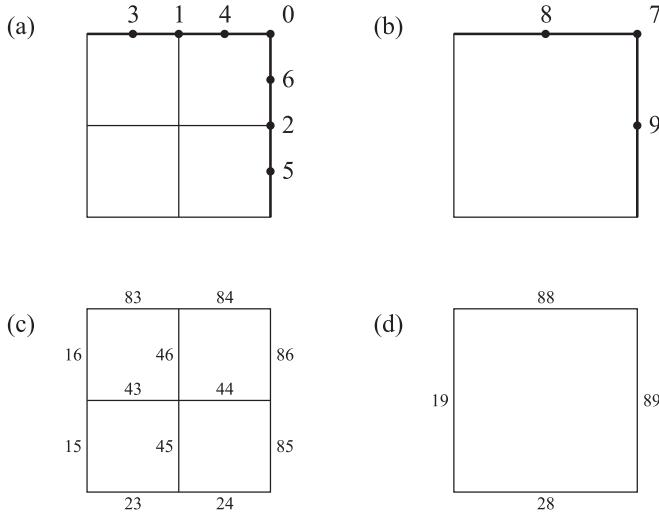
**Figure 4.16.** The voxel sample locations for (a) the full-resolution face and (b) the half-resolution face of a transition cell are numbered, in hexadecimal, as shown.

If the case index is 0 or 511, then the transition cell contains no triangles. Otherwise, the case index is used to fetch the equivalence class index from a 512-entry lookup table. Since some transition cell equivalence classes include geometric inverses, we also need to be able to identify which cases are inverted relative to the representative case for their classes so that we know to wind the triangles in the reverse direction. We indicate this property by setting the high bit of the equivalence class index in the lookup table for any case that is inverted.

As was done in our modified Marching Cubes implementation, we limit the locations at which new vertices can be created for a transition cell and reuse vertices created for preceding cells whenever possible. The creation of new vertices is permitted everywhere except along the minimal edges of the full-resolution and half-resolution faces with respect to the local coordinate system. Of the locations where vertex creation is permitted, vertices lying at the ten locations shown in Figures 4.18(a) and 4.18(b) can be reused by adjacent cells in the same row or the succeeding row. Vertices lying on the four interior edges or at the center location are never reused.



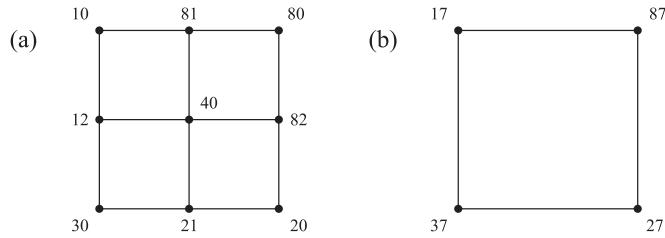
**Figure 4.17.** For each negative voxel value, the corresponding hexadecimal quantity shown here is contributed to the transition cell case index.



**Figure 4.18.** Vertices placed on the maximal edges of a transition cell can be reused by succeeding transition cells. Reusable vertices on (a) the full-resolution face and (b) the half-resolution face are numbered 0 through 9 as shown. An 8-bit hexadecimal code is assigned to each edge of (c) the full-resolution face and (d) the half-resolution face to provide a mapping to a reusable vertex belonging to a preceding transition cell.

Each of the 16 edges on which a vertex can lie is assigned an 8-bit code, as shown in Figures 4.18(c) and 4.18(d), that provides a mapping to a preceding cell and the edge from within that preceding cell from which a vertex may be reused. The high nibble of this code indicates which direction to go in order to reach the correct preceding cell. The bit values 1 and 2 in this nibble indicate that we must subtract one from the  $x$  or  $y$  coordinate, respectively, and these two bits are never simultaneously set. The bit value 4 indicates that a new vertex is to be created on an interior edge where it cannot be reused, and the bit value 8 indicates that a new vertex is to be created on a maximal edge where it can be reused. The low nibble of the 8-bit code gives the index of the vertex in the preceding cell that should be reused or the index of the vertex in the current cell that should be created.

When the voxel value at one of the 13 sample locations (see Figure 4.16) is exactly zero, a vertex is always placed precisely at that location. We allow new vertex creation only at locations not lying along a minimal edge, so a new vertex can be created at locations 4, 5, 7, 8, and C. For any other locations, we reuse vertices previously created for a preceding cell. Each sample location is assigned an 8-bit code, as shown in Figure 4.19, that has a meaning similar to the code assigned to each edge. The high nibble contains a direction code for which the bit values 1 and 2 correspond to a subtraction in the local  $x$  and  $y$  directions, and the low nibble specifies one of the storage slots shown in Figures 4.18(a) and 4.18(b).



**Figure 4.19.** An 8-bit hexadecimal code is assigned to each voxel sample location on (a) the full-resolution face and (b) the half-resolution face of a transition cell to provide a mapping to a reusable vertex belonging to a preceding cell.

Of course, for transition cells in the first row and first column of a block face, the preceding cells needed for vertex reuse may not exist. In these cases, we allow new vertex creation on minimal edges where necessary. A 2-bit mask is maintained as we iterate over the transition cells on a block face indicating which bits in the direction codes from Figures 4.18 and 4.19 are valid. If the bit value 1 or 2 is set in a direction code, but is not set in the validity mask, then a new vertex is created for the current cell. The bit values 4 and 8 are never valid, and these are used for the interior edges and maximal edges to force new vertex creation in those locations all the time.

Vertex positions are calculated as described for the modified Marching Cubes algorithm in Section 3.3, but endpoints for edge interpolation must be chosen in the manner described in Section 4.2 (see Figure 4.3) on faces where the voxel data is not sampled at the highest resolution. This is always true on the half-resolution face of a transition cell, and is true on the full-resolution face when the level of detail is 2 or greater.

# Chapter 5

## Texturing and Shading

### 5.1 Introduction

The unlimited array of geometric formations made possible by a voxel terrain system present new challenges when it comes to applying texture maps and complex shading to a terrain surface. The methods used to map textures to conventional height-based terrain and produce shading effects such as normal mapping are inadequate for voxel-based terrain in many respects. This chapter discusses more advanced methods that have we have developed to account for the topographies of voxel-based terrain and produce high-quality results where conventional methods fail.

### 5.2 Texture Mapping

In order to apply texture maps to a terrain mesh, we must be able to specify a mapping function that provides texture sampling coordinates for each vertex position. This is ordinarily accomplished by performing some kind of projection on the vertex positions themselves to obtain two-dimensional texture coordinates. In the case of height-based terrain, it is common to simply drop the  $z$  coordinate and scale the  $(x, y)$  coordinates to produce the  $(s, t)$  texture coordinates, and this can be done very efficiently in a vertex program. The problem with this approach is that texture images become stretched along one direction as the slope of the terrain surface increases, severely distorting the aspect ratio of the texels. This problem becomes much worse for voxel-based terrain, as shown in Figure 5.1, because the slope can be infinite with respect to the  $x$ - $y$  plane. Furthermore, such a planar projection reverses the handedness of the tangent frame for surfaces where the normal points downward.

One solution to the texture stretching problem is to generate multiple sets of texture coordinates by projecting the vertex position onto multiple planes, sampling one or more texture maps at these locations, and blending the results in some way. The technique known as *triplanar projection* [Geiss and Thompson 2007; Geiss 2008] generates three sets of texture coordinates by projecting the vertex position onto planes perpendicular to each of the three coordinate axes. Up to three different texture maps are then sampled, and the results are blended based on the direction in which the vertex normal points.



**Figure 5.1.** Texture coordinates obtained by projecting vertex positions onto the  $x$ - $y$  plane exhibit severe stretching problems where the terrain surface is nearly vertical. The tangent frame on the undersides of the terrain features also has a reversed handedness.

For each of the three planes onto which vertex positions are projected, if the surface normal is oriented such that it forms an obtuse angle with the plane's normal, then the texture image is mirrored in one direction. This undesirable effect can be eliminated by negating one of the texture coordinates generated for each plane whenever the surface normal forms a negative dot product with the plane normal. Since the plane normals are aligned to the coordinate axes, computing the dot products amounts to nothing more than picking off individual components of the surface normal, so we calculate our three sets of texture coordinates  $(s_x, t_x)$ ,  $(s_y, t_y)$ , and  $(s_z, t_z)$  as follows, where  $\mathbf{p}$  is the scaled vertex position:

$$\begin{aligned}
 s_x &= \begin{cases} p_y, & \text{if } N_x \geq 0, \\ -p_y, & \text{if } N_x < 0; \end{cases} \\
 s_y &= \begin{cases} -p_x, & \text{if } N_y \geq 0, \\ p_x, & \text{if } N_y < 0; \end{cases} \\
 s_z &= \begin{cases} p_x, & \text{if } N_z \geq 0, \\ -p_x, & \text{if } N_z < 0; \end{cases} \\
 t_x &= t_y = p_z \\
 t_z &= p_y.
 \end{aligned} \tag{5.1}$$

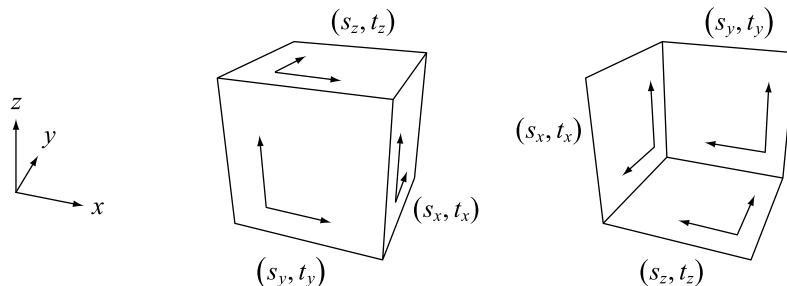
What this equation really establishes is a cubic projection in which the texture coordinates for each face are aligned as shown in Figure 5.2. Using these coordinates, a texture image is never mirrored when a cube face is viewed from its front side. The first line of Listing 5.3 calculates a vector holding ones for values of  $s$  that should be negated and zeros otherwise. This is later used to interpolate between  $\mathbf{p}$  and  $-\mathbf{p}$  in order to calculate texture coordinates.

Once the three sets of texture coordinates have been determined, we can sample one texture map at three different locations, or we can choose to sample up to three different texture maps. In any case, we need to combine the color samples fetched from the texture maps in such a way that stretching caused by any one of the projections is not visible. An effective method for calculating blend weights  $b_x$ ,  $b_y$ , and  $b_z$  is to use the formulas

$$\begin{aligned} b_x &= \left( \max \left\{ \frac{|N_x|}{\|\mathbf{N}\|} - \delta, 0 \right\} \right)^m \\ b_y &= \left( \max \left\{ \frac{|N_y|}{\|\mathbf{N}\|} - \delta, 0 \right\} \right)^m \\ b_z &= \left( \max \left\{ \frac{|N_z|}{\|\mathbf{N}\|} - \delta, 0 \right\} \right)^m, \end{aligned} \quad (5.2)$$

where  $\mathbf{N}$  is the interpolated vertex normal,  $\delta$  is a real number in the interval  $[0, \sqrt{3}/3]$ , and  $m$  is a positive integer. These weights will not generally sum to unity, so we normalize them as follows:

$$\begin{aligned} b'_x &= \frac{b_x}{b_x + b_y + b_z} \\ b'_y &= \frac{b_y}{b_x + b_y + b_z} \\ b'_z &= \frac{b_z}{b_x + b_y + b_z}. \end{aligned} \quad (5.3)$$



**Figure 5.2.** The texture coordinates given by Equation (5.1) establish a cubic projection for which the axes are aligned as shown for each of the six cube faces.

An example fragment shader snippet that calculates these blend weights is shown in Listing 5.3. The blended texture sample  $C$  is given by

$$C = b'_x C_x(s_x, t_x) + b'_y C_y(s_y, t_y) + b'_z C_z(s_z, t_z), \quad (5.4)$$

where  $C_x$ ,  $C_y$ , and  $C_z$  are functions representing the value returned by sampling the texture maps associated with the plane normal directions  $x$ ,  $y$ , and  $z$ .

The offset  $\delta$  in Equation (5.2) creates a plateau on each of the cube faces for which texture maps associated with a very small component of the normal vector do not contribute to the final color. To avoid dividing by zero, we must have  $\delta < \sqrt{3}/3$  since the largest component of a unit normal vector can be as small as  $\sqrt{3}/3$  in absolute value. The exponent  $m$  controls how fast the transition between texture maps is as the dominant component of the normal vector moves from one coordinate to another on a curved surface. Higher values of  $m$  produce a sharp transition that takes place over a small angle, and lower values produce a smooth transition that takes place over a large angle. Figure 5.3 shows the affects on texturing blending caused by varying the values of  $\delta$  and  $m$  in Equation (5.2).

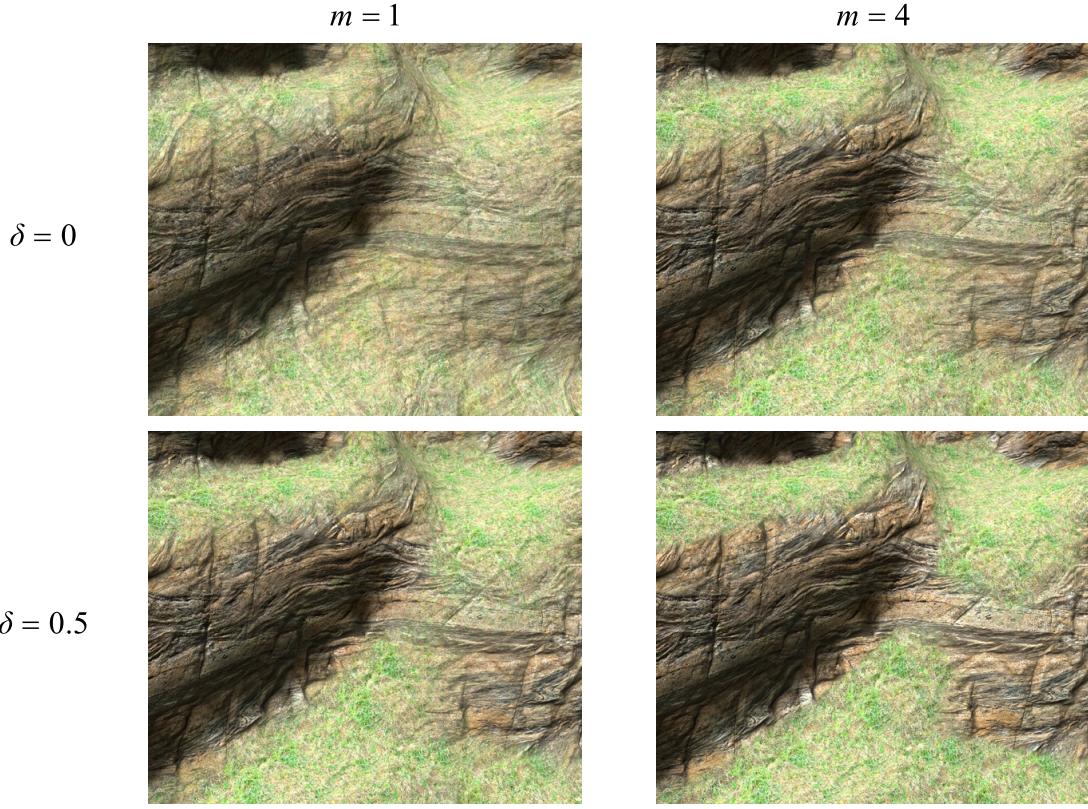
Figure 5.4 shows the cubic projection applied to the same terrain that was previously shown in Figure 5.1. Two different texture maps are sampled, one for the planes perpendicular to the  $x$  and  $y$  axes, and another for the plane perpendicular to the  $z$  axis. Even though only two texture maps are used, one of them is sampled at two locations to produce a total of three colors to be blended together. The blend weights used in Figure 5.4 are those given by Equations (5.2) and (5.3) with  $\delta = 1/2$  and  $m = 4$ .

**Listing 5.3.** This snippet of GLSL fragment shader code determines which values of  $s$  in Equation (5.1) should be negated, and it calculates the blend weights given by Equation (5.2) for  $\delta = 1/2$  and  $m = 4$ . The interpolated vertex normal is stored in `nrml`.

```
// Calculate a vector that holds ones for
// values of s that should be negated.
vec3 flip = vec3(nrml.x < 0.0, nrml.y >= 0.0, nrml.z < 0.0);

// Raise each component of normal vector to 4th power.
vec3 blend = saturate(abs(normalize($NRML)) - 0.5);
blend *= blend;
blend *= blend;

// Normalize result by dividing by the sum of its components.
blend /= dot(blend, vec3(1.0, 1.0, 1.0));
```

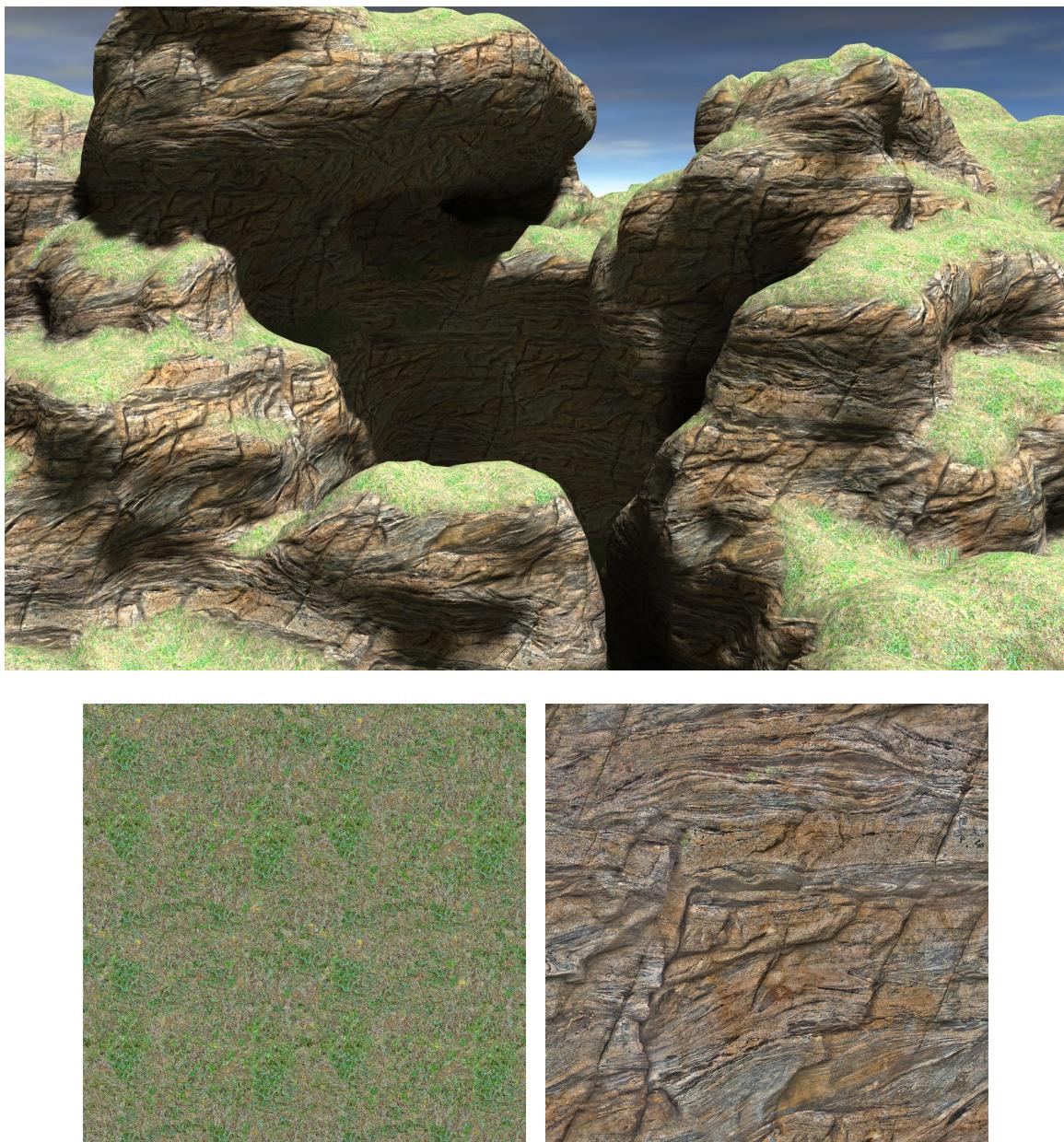


**Figure 5.3.** These screenshots show how the triplanar texture blending is affected by using different values of  $\delta$  and  $m$  in Equation (5.2). In the upper-left image, streaking can be seen because the texture used on the positive  $z$  face is blended into the texture for the  $x$  and  $y$  faces of the cubic projection. This is eliminated by using the higher value of  $\delta$  in the bottom row. In both rows, the higher value of  $m$  causes the transition between textures to be sharper.

### 5.3 Material Specification

At any particular point on a terrain surface, we may want to be able to specify multiple texture maps for each of the six faces of the cubic projection, and we may want to include information about how our shaders should blend samples fetched from these texture maps. It is natural to store this kind of data at the same resolution that the voxel map generating the terrain geometry is stored. Instead of storing only the distance to the terrain surface at each voxel, we store multiple channels of data containing different kinds of information.

Each channel of per-voxel data potentially consumes a large amount of storage space, so we want to keep the total number of channels to a minimum. Storing a full set of texture map indexes at each voxel would consume too much space, so we instead store a single 8-bit material identifier. In a second channel, we store another 8-bit value representing a blending parameter. This gives us



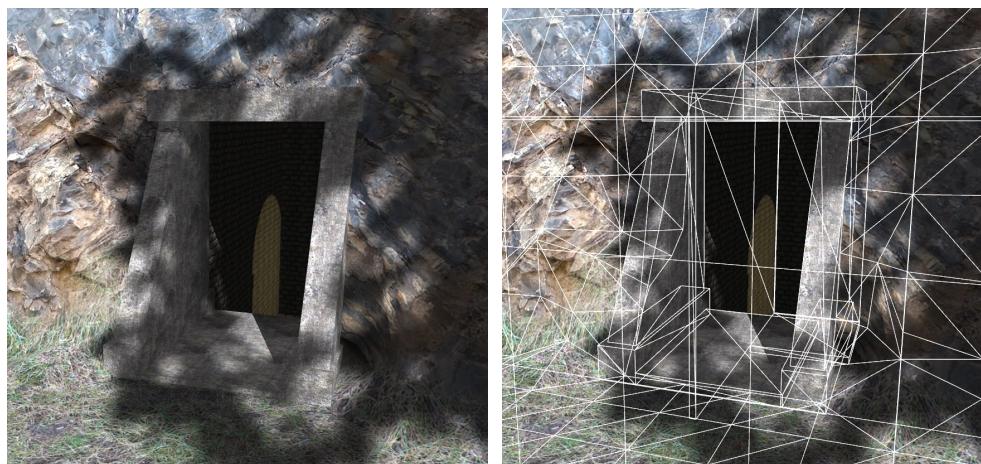
**Figure 5.4.** The terrain previously shown in Figure 5.1 has been retextured using the cubic projection given by Equation (5.1). The left texture map shown in the bottom row is applied for the positive  $z$  projection face, and the right texture map is applied to all other projection faces. The texture samples have been blended using Equation (5.2) with  $\delta = 1/2$  and  $m = 4$ .

a total of three channels that are stored for the entire voxel map. The channels are compressed independently because material information normally changes at much lower frequencies over large areas of the terrain compared to the data in the primary distance channel.

The material identifier is a value in the range  $[0, 254]$  that is interpreted as an integer and used to index into a table containing information about texture map selections for each material. A separate material table can be maintained for each of the lowest-resolution terrain blocks to enable the use of a large number of distinct materials over a large area, or a single table can be maintained globally for the entire voxel map if the total number of distinct materials is no greater than 255.

The material identifier 255 is treated as a special value. It's a “poison” material that prevents triangles from being generated for any cell that uses it. Having this capability is useful in cases when an entrance to some kind of interior or underground structure is needed, and this structure is not modeled with the voxel map, as demonstrated in Figure 5.5. Without a poison material, it would be necessary to dig a large cavern in the voxel data to provide a cavern for the interior structure, and this would generate a large number of triangles that would most likely never be visible when rendering the scene.

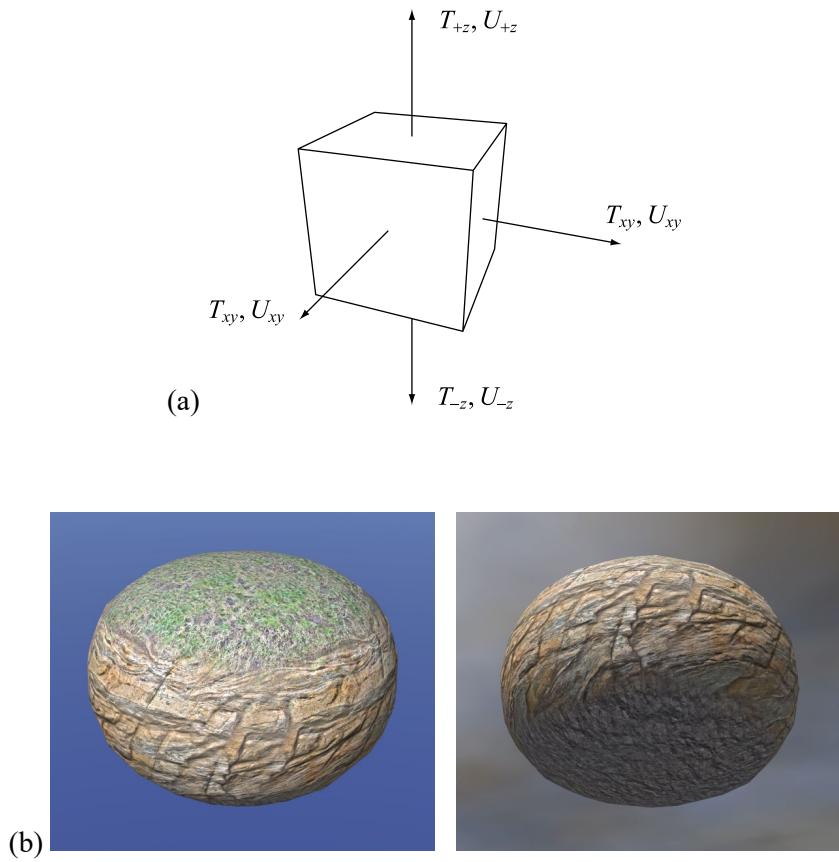
The table into which the material identifier indexes contains a set of texture map indexes. For each material, we can specify one or more texture maps (stored in either a texture array or a texture palette—see Section 5.4) for subsets of the six faces of the cubic projection discussed in Section 5.2. Since we use a single blending channel in our implementation, we specify two texture maps for each subset, and the blend value determines how they are combined in the fragment shader.



**Figure 5.5.** A poison material identifier prevents triangles from being generated for cells that use it so that holes can be punched into the terrain surface at entrances to interior structures.

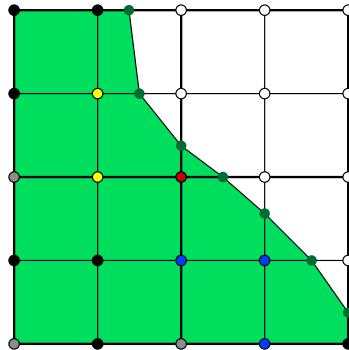
It is possible to store separate texture map information for all six faces, but is it usually the case that the same texture maps are used on all of the vertically aligned faces. In our implementation, a material comprises a total of six texture map selections: two for the positive  $z$  face, two for the negative  $z$  face, and two for all four of the remaining faces. This divides the sphere into three separate material domains, as illustrated in Figure 5.6.

When a cell is triangulated by our modified Marching Cubes implementation, each of the triangles generated for the cell use the same material. For the highest-resolution cells, we choose the material associated with the voxel at the minimal corner. For lower-resolution cells, we select the material by examining its eight subcells and choosing the material that is most populous among those subcells that are not trivial. This process recurs until the highest resolution is reached, and it ensures that the materials rendered at lower levels of detail are materials that actually participate in



**Figure 5.6.** (a) A material comprises three pairs of texture maps, and the indexes of the texture maps in each pair are labeled  $T$  and  $U$ . One pair of texture maps is applied to the positive  $z$  face, another is applied to the negative  $z$  face, and the third is applied to the remaining four faces. These are identified by the subscripts  $+z$ ,  $-z$ , and  $xy$ , respectively. (b) A different texture map is applied to each of the three material domains, shown from the top (left) and bottom (right).

the rendering of the highest-detail terrain surface. This process is necessary because materials are often painted by hand, and the voxel data may only be updated at the highest resolution near the terrain surface. If we simply selected the material specified by one of the corner voxels for a low-resolution cell, we are likely to choose some default material identifier that is assigned to a voxel deep inside the terrain in solid space or far outside in empty space. An example showing which voxels influence material selection for a lower level of detail is shown in Figure 5.7.



**Figure 5.7.** For the triangles generated in each of the cells shown here, the colored voxel at the cell’s minimal corner determines which material is applied. For the next lower level of detail, the minimal corners are at the center voxel and the three voxels colored gray. Using information stored at these voxels could alter the material applied to the mesh surface, so we only allow the material selection to be influenced by data stored for nontrivial subcells. The two yellow voxels determine the material for the upper-left cell, the red voxel determines the material for the upper-right cell, and the three blue voxels determine the material for the lower-right cell.

Once a material has been selected for a cell, we look up the texture map indexes in the material table and assign them to the vertices used by the cell. If a vertex is reused from a preceding cell, and the preceding cell selects a different material, then the vertex must be duplicated so that the different texture map indexes can be assigned to it. The attribute data associated with a vertex now consists of a three-component floating-point position, a three-component floating-point normal vector, and two four-component 8-bit unsigned integer material attributes, as shown in Figure 5.8. The first material attribute contains the two texture map indexes for the four vertically aligned faces of the cubic projection, and the second material attribute contains the two indexes for the positive  $z$  face and the two indexes for the negative  $z$  face. Listing 5.4 demonstrates how the correct pair of indexes for the  $z$  faces is selected in a fragment shader. The blend parameter  $\alpha$  is stored in the  $z$  component of the first material attribute, and the  $w$  component is left open for other uses. (In our implementation, we often store a per-vertex ambient occlusion factor in the  $w$  component.)

	$P_x$				$P_y$				$P_z$	
	X				Y				Z	
	$N_x$				$N_y$				$N_z$	
	X				Y				Z	
$T_{xy}$	$U_{xy}$	$\alpha$								
X	Y	Z	W							
$T_{+z}$	$T_{-z}$	$U_{+z}$	$U_{-z}$							
X	Y	Z	W							

**Figure 5.8.** The attributes generated for each vertex include a three-component floating-point position  $\mathbf{P}$ , a three-component floating-point normal vector  $\mathbf{N}$ , and two four-component 8-bit unsigned integer material attributes. The letters  $T$  and  $U$  represent the indexes of the primary and secondary texture maps, and the subscripts  $xy$ ,  $+z$ , and  $-z$  indicate the set of four vertically aligned faces, the positive  $z$  face, and the negative  $z$  face. The blend parameter is represented by the value  $\alpha$  stored in the  $z$  component of the first material attribute.

**Listing 5.4.** This snippet of GLSL fragment shader code selects either the pair of texture map indexes  $(T_{+z}, U_{+z})$  or the pair of texture map indexes  $(T_{-z}, U_{-z})$ . The choice depends on the value of the  $z$  component of `flip`, calculated in Listing 5.3.

```
// Declare the interpolants that hold the material attributes.
// (See Figure 5.8.)
varying vec4    material1;
varying vec4    material2;

// Select the texture map indexes T and U for either
// the positive z face or the negative z face.
vec2 zindex = lerp(material2.xz, material2.yw, flip.z);
```

## 5.4 Texture Organization

It is usually necessary to apply a variety of different materials to a large terrain mesh in order to achieve a desired result. For instance, different parts of the terrain may need to be rendered with the appearance of grass, dirt, mud, rock, asphalt, or the leaf-covered floor of a forest. In addition to the ability to blend between texture maps based on the cubic projection and the direction of the normal vector, we usually have the need to blend between two or more texture maps along the transition from one type of material to another. Furthermore, many materials require that samples be fetched from several texture maps, such as diffuse maps, normal maps, and gloss maps, whose texels have different meanings. What this all means is that we need to have random access to a large set of texture maps at a reasonably fine granularity.

### 5.4.1 Texture Arrays

The number of distinct texture maps needed to render the mesh for a single terrain block can exceed the number of texture units available in the graphics hardware, so we do not have the option of addressing all the required texture maps as separate texture objects in a shader. Even if the number of texture units was not a limiting factor, modern graphics hardware does not have the ability to select a texture unit using a variable parameter when a texture map is sampled. Our only option is to pack many texture maps into a single texture object and address the individual texture maps by modifying the texture coordinates used to fetch samples. Beginning with the fourth generation of programmable graphics hardware,<sup>12</sup> this is particularly easy to accomplish because they support *texture arrays* [Leech and Kilgard 2008].

A texture array is a set of independent two-dimensional texture maps like that shown in Figure 5.9. All of the texture maps have the same dimensions, and all of them have a complete mipmap chain. A texture array is addressable as a single unit by the hardware through a three-dimensional set of coordinates  $(s, t, r)$ . Unlike a three-dimensional texture map, however, there is no filtering between neighboring layers based on the  $r$  coordinate. Instead, the  $r$  coordinate is rounded to the nearest integer to select one layer in the texture array, and the  $(s, t)$  coordinates are then used to fetch a filtered sample from a two-dimensional texture map in the conventional manner. Listing 5.5 demonstrates how the texture coordinates given by Equation (5.1) can be calculated for use with a texture array by incorporating the material indexes.

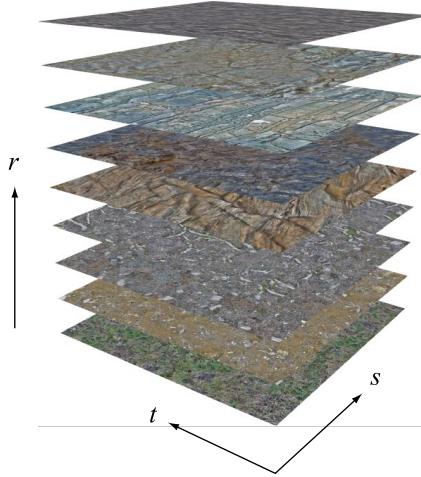
### 5.4.2 Texture Palettes

For graphics hardware that does not have texture array capabilities, which includes current-generation game consoles such as the PlayStation 3 and Xbox 360, we must resort to another method of packing multiple texture maps into a single texture object. Three-dimensional texture maps do not provide a viable solution because there is no way to prevent filtering between adjacent layers when using mipmaps necessary for good performance. The only practicable course of action is to pack multiple images into a larger two-dimensional texture map called a *texture palette*.

When rendering with a texture palette, we select the index of the texture map that needs to be sampled in the same way that it would be done when using a texture array. Instead of simply setting the  $r$  coordinate to the texture map index, however, we transform the  $s$  and  $t$  coordinates so that they correspond to the correct subimage of the overall texture map. Unfortunately, this process

---

<sup>12</sup> The first GPUs in the fourth generation are the Nvidia GeForce 8000 series and the ATI Radeon HD 2000 series.



**Figure 5.9.** Modern graphics hardware supports texture arrays, which are stacks of two-dimensional images that are stored as a single unit of addressable texels, but for which each image is independently filtered. The  $r$  coordinate is rounded to the nearest integer to select a texture image, and the  $s$  and  $t$  coordinates specify the sample location within that image.

**Listing 5.5.** This snippet of GLSL fragment shader code calculates the three sets of texture coordinates given by Equation (5.1) for use with a texture array. The texture coordinates  $(s_x, t_x)$  are stored in the  $x$  and  $z$  components of both `texcoord1` and `texcoord2`. The texture coordinates  $(s_y, t_y)$  are stored in the  $y$  and  $z$  components of both `texcoord1` and `texcoord2`. The texture coordinates  $(s_z, t_z)$  are stored in the  $x$  and  $z$  components of `texcoord3`. The variables `material1` and `zindex` are declared in Listing 5.4.

```
// texcoord1 = (sx, sy, tx|ty, Txy)
texcoord1.xy = lerp(p.yx, -p.yx, flip.xy);
texcoord1.z = p.z;
texcoord1.w = material1.x * 255.0;

// texcoord2 = (sx, sy, tx|ty, Uxy)
texcoord2.xyz = texcoord1.xyz;
texcoord2.w = material1.y * 255.0;

// texcoord3 = (sz, tz, Tz, Uz)
texcoord3.x = lerp(p.x, -p.x, flip.z);
texcoord3.y = p.y;
texcoord3.zw = zindex * 255.0;
```

leads to a number of difficulties that produce visible artifacts in the rendered scene. These are discussed in the remainder of this section and in the next section.

First, we need to be able to extract a properly filtered sample at the borders of each subimage because the individual textures are often repeated across a terrain mesh. Typical graphics hardware can only perform such filtering with proper wrapping at the boundaries of the entire texture map, so we cannot rely on the hardware to perform wrapped filtering for any subimage. This leaves us with the following two options:

1. Hardware filtering can be disabled, and we can instead perform the filtering calculations explicitly in the fragment shader.
2. A portion of each texture map can be repeated along the border on all four sides of every subimage packed into the larger texture map.

The first option works correctly, but it is slow because it requires manual calculation of the sample interpolation parameters, and it requires eight times as many texture fetches as would normally be required for trilinear filtering. The second option takes full advantage of hardware filtering capabilities, but requires more memory for storing the repeated portions of each texture map. Since memory space is a relatively inexpensive resource, we choose the second solution for its greater speed.

Suppose that the individual images in our texture palette are square and have a power-of-two width and height of  $w$  texels. When we pack multiple images into a texture palette, the overall dimensions of this larger texture map must be rounded up to the next highest power of two for reasons of performance and compatibility with a wide variety of graphics hardware. If we pack  $k$  images in either the horizontal or vertical direction, then the overall width  $W$  required for the texture palette in that direction is given by

$$W = 2^{\lceil \log_2 kw(1+2b) \rceil}, \quad (5.5)$$

where  $b$  is the fraction of each image repeated along its borders. The value of  $b$  is typically a small negative power of two so that mipmaps can be generated for all but the lowest resolutions without leakage from one image into another. When rendering, we set the maximum mipmap level  $lod_{max}$  for the texture palette to

$$lod_{max} = \log_2 wb \quad (5.6)$$

in order to prevent these mipmaps from being accessed by the hardware. (Consequently, mipmaps levels beyond this one don't even need to exist.)

Because it is used frequently in this section, we define the function  $K(k, b)$  as

$$K(k, b) = 2^{\lceil \log_2 k(1+2b) \rceil}. \quad (5.7)$$

The overall width of a texture palette can then be expressed as the function

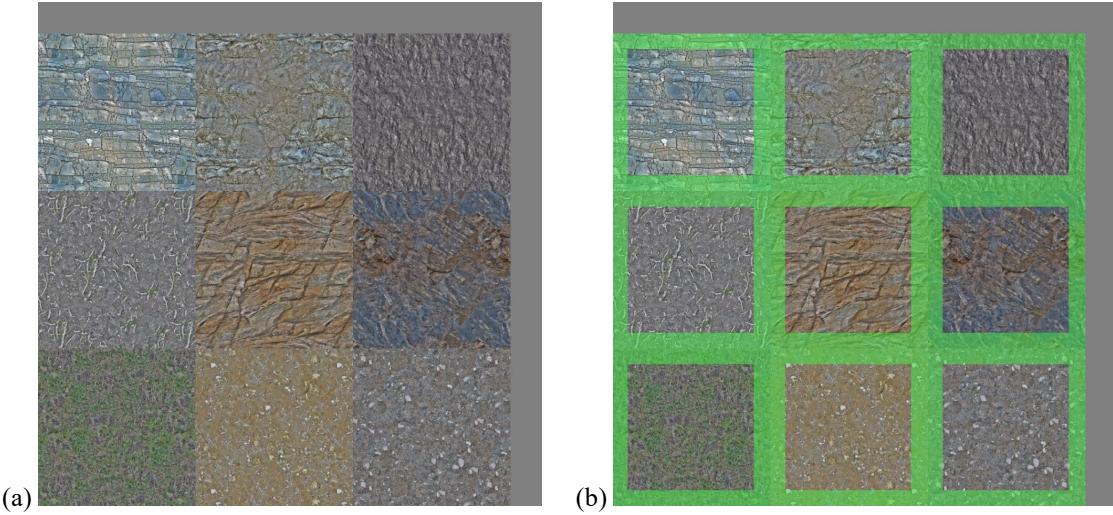
$$W(w, k, b) = wK(k, b) \quad (5.8)$$

since  $w$  is equal to an integer power of two.

Figure 5.10 shows an example texture palette containing nine  $1024 \times 1024$  images packed into a single texture map in a  $3 \times 3$  arrangement. Around the border of each subimage, one-eighth of the texels belonging to the subimage are copied from the opposite sides to produce a partial repetition. Thus, the width and height of each texture map in the palette are 125% those of the original texture map, and the total number of texels occupied is just over 156% of the original size. Plugging the values  $w = 1024$ ,  $k = 3$ , and  $b = 1/8$  into Equation (5.5) gives us an overall size of  $4096 \times 4096$  texels for the texture palette. By Equation (5.6), we are able to utilize mipmap levels 0 through 7.

Because we must round the dimensions of the texture palette up to the next power of two in each direction, there is always some space that goes unused. Table 5.1 shows the value of  $k(1 + 2b)$  for a number of typical values of the variables  $k$  and  $b$  used in Equation (5.5). The ceiling of the base-2 logarithm of this value is evaluated in Equation (5.5), so we want this value to make a small difference with the next higher power of two. Table 5.2 shows the actual percentage of space used in texture palettes derived from the best combinations of the values of  $k$  and  $b$  from Table 5.1.

The best space utilization occurs when  $k = 7$  and  $b = 1/16$ , but such a large value of  $k$  causes us to reach the size limit of 4096 texels for the dimensions of a texture map on targeted graphics hardware (i.e., hardware that does not support texture arrays) when individual palette entries are only  $512 \times 512$  texels in size. Also, the smaller value of  $b$  limits the number of mipmaps that can be used for texture minification a little too much. In our implementation, we fix  $b$  to be  $1/8$ , and we allow  $k$  to be 3 or 6. A  $3 \times 3$  texture palette contains images up to  $1024 \times 1024$  texels in size, and a  $3 \times 6$  or  $6 \times 6$  texture palette contains images up to  $512 \times 512$  texels in size.



**Figure 5.10.** On hardware that does not support texture arrays, we must pack multiple images into a conventional two-dimensional texture map. (a) Nine images of size  $1024 \times 1024$  texels are packed into a single texture map of size  $4096 \times 4096$  texels. (b) The area highlighted in green shows where each of the nine images is repeated for 128 texels to allow smooth filtering on its border. The actual area used in the texture map is  $3840 \times 3840$  texels.

**Table 5.1.** This table shows values of  $k(1 + 2b)$  for typical values of the variables  $k$  and  $b$  used in Equation (5.5). The entries highlighted in green represent the best choice of  $b$  for each value of  $k$  because they have the smallest difference with the next higher power of two.

	$b = 1/16$	$b = 1/8$	$b = 1/4$
$k = 2$	2.25	2.5	3
$k = 3$	3.375	3.75	4.5
$k = 4$	4.5	5	6
$k = 5$	5.625	6.25	7.5
$k = 6$	6.75	7.5	9
$k = 7$	7.875	8.75	10.5

**Table 5.2.** This table shows the space utilization in a texture palette for the best combinations of  $k$  and  $b$  in Table 5.1. The numbers in the second and third columns are expressed in multiples of the width of a single image in the palette without including repetitions around the borders.

Parameters	Image Array Size with Borders $k(1 + 2b)$	Texture Palette Size $K(k, b)$	Texture Map Space Utilization $\left(\frac{k(1 + 2b)}{K(k, b)}\right)^2$
$k = 2, b = 1/4$	3	4	56.25%
$k = 3, b = 1/8$	3.75	4	87.89%
$k = 4, b = 1/4$	6	8	56.25%
$k = 5, b = 1/4$	7.5	8	87.89%
$k = 6, b = 1/8$	7.5	8	87.89%
$k = 7, b = 1/16$	7.875	8	96.90%

### 5.4.3 Palette Sampling

A texture palette must be sampled in a fragment shader using a specific process in order to achieve high performance while avoiding rendering artifacts. Let  $(s, t)$  be one of the sets of texture coordinates that are given by Equation (5.1), and suppose that a texture map index  $r$  has been calculated. Then for a texture palette having an array of  $k \times k$  subimages and a border fraction  $b$ , the coordinates  $(i, j)$  where we actually sample the texture palette are given by

$$i = \frac{(r \bmod k)(1 + 2b) + b + \text{fract}(s)}{K(k, b)}$$

$$j = \frac{\lfloor r/k \rfloor(1 + 2b) + b + \text{fract}(t)}{K(k, b)}. \quad (5.9)$$

(The `fract` function returns the fractional part of its argument modulo the integers, defined as  $\text{fract}(x) = x - \lfloor x \rfloor$ .) The values of  $r \bmod k$  and  $\lfloor r/k \rfloor$  represent the grid coordinates of the texture map within the palette. The multiplication by  $1 + 2b$  accounts for the size of each palette entry, the offset by  $b$  accounts for the repetition border surrounding each entry, and the coordinates  $(\text{fract}(s), \text{fract}(t))$  provide the sample position within the entry. Finally, the division by  $K(k, b)$  normalizes the texture coordinates against the overall size of the texture palette. An example fragment shader that calculates the sample coordinates  $(i, j)$  with Equation (5.9) for all three pairs of texture maps used by our terrain material is shown in Listing 5.6.

**Listing 5.6.** This snippet of GLSL fragment shader code calculates the sample coordinates  $(i, j)$  using Equation (5.9) for the three pairs of texture maps specified by the material for the positive  $z$  face, negative  $z$  face, and the four vertically aligned faces. This code is written for a  $3 \times 3$  texture palette (and thus  $k = 3$ ) with a border fraction  $b = 1/8$ .

```
// matindex = (Txy, Uxy, Tz, Uz)
vec4 matindex = vec4(material1.xy * 255.0, zindex * 255.0);

vec3 tmp1, tmp2;
tmp1.xy = fract(-p.xy);
tmp2 = fract(p);
tmp1 = lerp(tmp2.yxx, tmp1.yxx, flip);

// tmp1 = (fract(sx, sy, sz) + b) / K(k,b)
tmp1 = tmp1 * 0.25 + 0.03125;

// tmp2 = (fract(tx, ty, tz) + b) / K(k,b)
tmp2 = tmp2.zzy * 0.25 + 0.03125;

// tmp4 = floor((Txy, Uxy, Tz, Uz) / k)
vec4 tmp4 = floor(matindex * 0.3334);

// tmp3 = (Txy, Uxy, Tz, Uz) mod k
vec4 tmp3 = matindex - tmp4 * 3.0;

// Calculate three (i,j) pairs using Equation (5.9).
// texcoord1 = (sx1, sy1, tx1, ty1)
texcoord1.xy = tmp3.x * 0.3125 + tmp1.xy;
texcoord1.zw = tmp4.x * 0.3125 + tmp2.xy;

// texcoord2 = (sx2, sy2, tx2, ty2)
texcoord2.xy = tmp3.y * 0.3125 + tmp1.xy;
texcoord2.zw = tmp4.y * 0.3125 + tmp2.xy;

// texcoord3 = (sx3, sy3, tx3, ty3)
texcoord3.xy = tmp3.zw * 0.3125 + tmp1.z;
texcoord3.zw = tmp4.zw * 0.3125 + tmp2.z;
```

When either of the texture coordinates  $s$  or  $t$  crosses an integer boundary, it creates a discontinuity in the derivative of the corresponding coordinate  $i$  or  $j$  in Equation (5.9) due to the application of the `fract` function. The sudden jump from one end of the palette subimage to the opposite end causes the graphics hardware to select a high-numbered mipmap for any quad containing the discontinuity. This results in the appearance of a prominent two-pixel-wide seam where any texture subimage wraps when terrain is rendered with a texture palette.

This problem can be eliminated by replacing ordinary texture fetches in the fragment shader with texture fetches that take explicit derivatives for the texture coordinates as arguments. Additional instructions would be necessary to compute partial derivatives of  $s/K(k, b)$  and  $t/K(k, b)$  with respect to the  $x$  and  $y$  viewport axes. This process solves the filtering problem at the texture

wrap boundaries perfectly from a visual standpoint, but it introduces a serious performance problem. Ordinarily, the graphics hardware can assume that the derivatives used for a particular texture fetch are equal for all four pixels in a quad, but this assumption may not be valid when explicit derivatives are supplied by the fragment shader. Where the hardware would normally execute all four texture fetches in parallel using the same derivatives, it now has to serialize the four texture fetches using distinct derivatives, and this reduces performance to 25%.<sup>13</sup> Since a typical terrain shader can contain a dozen fetches from a texture palette, the overall impact on performance is very significant, making this solution unattractive.

An alternative is to use texture fetch instructions that take an explicit mipmap level. We can compute the mipmap level one time at the beginning of the fragment shader, and then pass the result to each texture fetch instruction. This method has the advantage that it runs at full speed, but it comes with one drawback. Since derivatives are no longer available to the texture units, the hardware is not able to perform anisotropic filtering. We consider this to be an acceptable trade-off for the large gain in performance.

As described in the OpenGL Specification [Segal and Akeley 2009], a legal approximation of the mipmap level  $\lambda$  is given by the following calculation for a texture map measuring  $w \times h$  texels:

$$\lambda = \log_2 \left[ \max \left\{ w \left| \frac{ds}{dx} \right|, w \left| \frac{ds}{dy} \right|, h \left| \frac{dt}{dx} \right|, h \left| \frac{dt}{dy} \right| \right\} \right] + bias. \quad (5.10)$$

The *bias* term is a constant level-of-detail bias that can be specified for the texture object. In order to avoid calculations involving  $w$  and  $h$  in the fragment shader, we set

$$bias = \log_2 (\min \{w, h\}) \quad (5.11)$$

at the time the texture palette is loaded into memory, and we calculate

$$\lambda = \log_2 \left[ \max \left\{ \left| \frac{ds}{dx} \right|, \left| \frac{ds}{dy} \right|, \left| \frac{dt}{dx} \right|, \left| \frac{dt}{dy} \right| \right\} \right] \quad (5.12)$$

in the fragment shader for each of the three sets of texture coordinates given by Equation (5.1). An implementation of this calculation is shown in Listing 5.7. One should note that the value of  $\lambda$  is not rounded to an integer, either explicitly or by the graphics hardware—its fractional part controls trilinear filtering between consecutive mipmap levels.

---

<sup>13</sup> Hardware implementing a new texture fetch instruction that takes explicit derivatives, but also allows the programmer to guarantee that the derivatives are constant over any primitive, would prove very useful in this situation.

**Listing 5.7.** This snippet of GLSL fragment shader code calculates the value of  $\lambda$  given by Equation (5.12) for all three sets of texture coordinates given by Equation (5.1). The value of  $p$  is the scaled vertex position.

```
// Calculate componentwise max derivatives.
vec3 temp = max(abs(dFdx(p)), abs(dFdy(p)));

// Determine max s and t derivative for each projection plane.
temp = max(temp.yxx, temp.zzy);

// Take logarithms to obtain final lambda values.
vec3 lambda = vec3(log2(temp.x), log2(temp.y), log2(temp.z));
```

## 5.5 Terrain Shading

All of the mathematics and fragment shader code developed so far in this chapter has involved the selection of texture maps, the calculation of texture coordinates, and the calculation of texture blend weights. We now turn our attention toward the use of this information in the implementation of complex terrain shaders. This section first briefly describes how texture samples are fetched from both texture arrays and texture palettes using the data calculated by shader code provided in the previous sections. The focus then turns to tangent-space per-pixel shading calculations. We provide a solution that enables the proper normal mapping in conjunction with the cubic texture coordinate projection discussed throughout this chapter.

### 5.5.1 Texture Fetches

What would normally constitute a single texture fetch in an ordinary fragment shader is replaced by six texture fetches in a shader using the cubic projection and blending discussed in this chapter. (This can be reduced to three fetches if we do not blend between two textures on each projection plane, but this makes it difficult to conceal transitions between materials.) By combining the results of all the calculations appearing in the previous sections, Listings 5.8 and 5.9 demonstrate how texture samples are fetched for a texture array and a texture palette, respectively, and how a single color is derived from these samples. In both cases, we acquire two sets of three color samples  $C_x$ ,  $C_y$ , and  $C_z$ , and these two sets are each blended using Equation (5.4) to produce two colors  $C_1$  and  $C_2$ . The final color is then given by

$$C = (1 - \alpha)C_1 + \alpha C_2, \quad (5.13)$$

where  $\alpha$  is the blend parameter shown in Figure 5.8.

**Listing 5.8.** This snippet of GLSL fragment shader code fetches color samples  $C_x$ ,  $C_y$ , and  $C_z$  for both the primary and secondary materials from a texture array. The swizzles appended to each set of texture coordinates specify the  $(s, t, r)$  coordinates passed to the graphics hardware, where the  $r$  coordinate corresponds to the texture map index in the array. Each group of three samples is blended using Equation (5.4), and the two resulting colors are then linearly interpolated using Equation (5.13).

```
uniform sampler2DArray textureArray;

// Fetch primary Cx, Cy, and Cz.
vec4 x1 = texture2DArray(textureArray, texcoord1.xzw);
vec4 y1 = texture2DArray(textureArray, texcoord1.yzw);
vec4 z1 = texture2DArray(textureArray, texcoord3.xyz);

// Fetch secondary Cx, Cy, and Cz.
vec4 x2 = texture2DArray(textureArray, texcoord2.xzw);
vec4 y2 = texture2DArray(textureArray, texcoord2.yzw);
vec4 z2 = texture2DArray(textureArray, texcoord3.yzw);

// Blend triplanar samples independently.
vec4 color1 = x1 * blend.x + y1 * blend.y + z1 * blend.z;
vec4 color2 = x2 * blend.x + y2 * blend.y + z2 * blend.z;

// Blend primary and secondary colors.
vec4 output = lerp(color1, color2, material1.z);
```

**Listing 5.9.** This snippet of GLSL fragment shader code fetches color samples  $C_x$ ,  $C_y$ , and  $C_z$  for both the primary and secondary materials from a texture palette. The swizzles appended to each set of texture coordinates specify the  $(s, t)$  coordinates passed to the graphics hardware, and the components of the variable `lambda` are the mipmap levels given by Equation (5.12). Each group of three samples is blended using Equation (5.4), and the two resulting colors are then linearly interpolated using Equation (5.13).

```
uniform sampler2D texturePalette;

// Fetch primary Cx, Cy, and Cz.
vec4 x1 = texture2DLod(texturePalette, texcoord1.xz, lambda.x);
vec4 y1 = texture2DLod(texturePalette, texcoord1.yw, lambda.y);
vec4 z1 = texture2DLod(texturePalette, texcoord3.xz, lambda.z);

// Fetch secondary Cx, Cy, and Cz.
vec4 x2 = texture2DLod(texturePalette, texcoord2.xz, lambda.x);
vec4 y2 = texture2DLod(texturePalette, texcoord2.yw, lambda.y);
vec4 z2 = texture2DLod(texturePalette, texcoord3.yw, lambda.z);

// Blend triplanar samples independently.
vec4 color1 = x1 * blend.x + y1 * blend.y + z1 * blend.z;
vec4 color2 = x2 * blend.x + y2 * blend.y + z2 * blend.z;

// Blend primary and secondary colors.
vec4 output = lerp(color1, color2, material1.z);
```

### 5.5.2 Tangent Space

The tangent-space bump mapping technique requires that we have information about the orientation of texture maps in the two-dimensional tangent space at each vertex of a triangle mesh  $M$ . Let  $s(\mathbf{p})$  and  $t(\mathbf{p})$  be functions that map a vertex position  $\mathbf{p}$  to a set of texture coordinates  $(s, t)$ , such as those given by Equation (5.1). To quantify the texture map orientation induced by the functions  $s(\mathbf{p})$  and  $t(\mathbf{p})$ , we select sections  $\mathbf{T}(\mathbf{p})$  and  $\mathbf{B}(\mathbf{p})$  from the tangent bundle<sup>14</sup> of  $M$  such that the following conditions are satisfied:

$$\begin{aligned}\mathbf{T}(\mathbf{p}) \cdot \nabla s(\mathbf{p}) &\geq 0 \\ \mathbf{T}(\mathbf{p}) \cdot \nabla t(\mathbf{p}) &= 0 \\ \mathbf{B}(\mathbf{p}) \cdot \nabla t(\mathbf{p}) &\geq 0 \\ \mathbf{B}(\mathbf{p}) \cdot \nabla s(\mathbf{p}) &= 0.\end{aligned}\tag{5.14}$$

The function  $\mathbf{T}(\mathbf{p})$  maps vertex positions to the texture-aligned tangent field for the mesh  $M$ , and the function  $\mathbf{B}(\mathbf{p})$  maps vertex positions to the texture-aligned bitangent field. Given an arbitrary mesh with nondegenerate texture coordinates assigned to each vertex, the tangent field and bitangent field can be calculated using a little linear algebra [Lengyel 2002].

It is not strictly necessary that the tangent and bitangent be perpendicular at any vertex position, but the bitangent is usually adjusted so that the two vectors form an orthogonal basis for the tangent space. This being the case, we use the normal vector  $\mathbf{N}(\mathbf{p})$  and tangent vector  $\mathbf{T}(\mathbf{p})$  to calculate the bitangent vector (up to sign) as

$$\mathbf{B}(\mathbf{p}) = \mathbf{N}(\mathbf{p}) \times \mathbf{T}(\mathbf{p}).\tag{5.15}$$

The vectors  $\mathbf{T}$ ,  $\mathbf{B}$ , and  $\mathbf{N}$  then form an orthogonal basis for a three-dimensional vector space associated with the vertex position  $\mathbf{p}$ , and the term *tangent space* normally refers to this vector space in the context of real-time rendering. An orthogonal basis makes it easy to transform a vector such as the direction to a light source into tangent space.

Ideally, the tangent field for a triangle mesh would be a continuous function over its surface for which each tangent can be normalized to unit length. This would correspond to a smooth set of texture coordinates that does not produce any seams in the texture maps applied to the mesh. However, for many meshes, choosing a set of texture coordinates that yields a continuous tangent field

---

<sup>14</sup> The *tangent bundle*  $TM$  of a manifold  $M$  is the set  $\{(\mathbf{p}, T_{\mathbf{p}}M) \mid \mathbf{p} \in M\}$ , where the notation  $T_{\mathbf{p}}M$  means the vector space tangent to  $M$  at the point  $\mathbf{p}$ . A *section* of a tangent bundle is a vector field formed by choosing a single vector in the tangent vector space at each point  $\mathbf{p}$  to form a function  $s : M \rightarrow T_{\mathbf{p}}M$ .

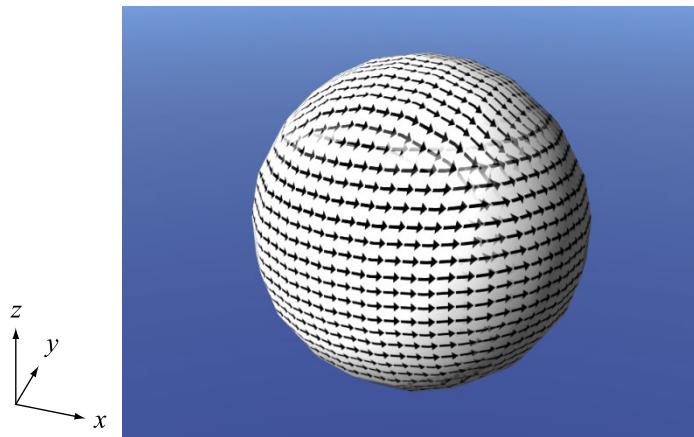
does not produce a mapping that is easy for artists to work with. Worse yet, any mesh that is topologically equivalent to a sphere cannot have a nonvanishing tangent field by the Poincaré-Hopf theorem [Hazewinkle 2001]. That is, it is impossible to smoothly map a texture image to any continuous terrain surface (without boundary) that doesn't have any holes in it. These difficulties force us abandon the notion of one continuous field of unit tangent vectors, as could be achieved with height-based terrain. We instead choose texture coordinates that are easy to work with, as we have already done in this chapter, and deal with the consequences.

The cubic projection described in Section 5.2 induces two distinct tangent fields with respect to the texture coordinates. The sets of coordinates  $(s_x, t_x)$  and  $(s_y, t_y)$  both induce a tangent field that is parallel to the  $x$ - $y$  plane everywhere and flows in the positive (counterclockwise) direction about the  $z$  axis. The set of coordinates  $(s_z, t_z)$  induces a tangent field that is parallel to the  $x$ - $z$  plane everywhere and flows in the positive direction about the  $y$  axis. These two tangent fields are illustrated on the surface of a sphere in Figure 5.11.

For a particular vertex position where the unit normal vector is  $\mathbf{N}$ , the tangent vector  $\mathbf{T}_1$  belonging to the first tangent field can be calculated by projecting the normal vector onto the  $x$ - $y$  plane, rotating it 90 degrees counterclockwise, and normalizing it to unit length as follows:

$$\mathbf{T}_1 = \frac{(-N_y, N_x, 0)}{\sqrt{N_x^2 + N_y^2}}. \quad (5.16)$$

The bitangent vector  $\mathbf{B}_1$  belonging to the first tangent field is then given by



**Figure 5.11.** The cubic projection produces three sets of texture coordinates that induce the two tangent fields shown on the surface of this sphere.

$$\mathbf{B}_1 = \mathbf{N} \times \mathbf{T}_1 = \frac{(-N_z N_x, N_y N_z, N_x^2 + N_y^2)}{\sqrt{N_x^2 + N_y^2}}. \quad (5.17)$$

A similar procedure with respect to the  $x$ - $z$  plane produces the tangent vector  $\mathbf{T}_2$  and bitangent vector  $\mathbf{B}_2$  in the second tangent field:

$$\begin{aligned} \mathbf{T}_2 &= \frac{(N_z, 0, -N_x)}{\sqrt{N_x^2 + N_y^2}} \\ \mathbf{B}_2 &= \mathbf{N} \times \mathbf{T}_2 = \frac{(-N_x N_y, N_x^2 + N_z^2, -N_y N_z)}{\sqrt{N_x^2 + N_z^2}}. \end{aligned} \quad (5.18)$$

The sets  $\{\mathbf{T}_1, \mathbf{B}_1, \mathbf{N}\}$  and  $\{\mathbf{T}_2, \mathbf{B}_2, \mathbf{N}\}$  form orthogonal bases for the two tangent spaces existing at the associated vertex position  $\mathbf{p}$ . Each set also forms the columns of a matrix that transforms vectors from one of the tangent spaces to the object space in which the coordinates of  $\mathbf{p}$  and  $\mathbf{N}$  are specified. A vector  $\mathbf{V}_{\text{tan}}$  expressed in the coordinates of a tangent space having the basis  $\{\mathbf{T}, \mathbf{B}, \mathbf{N}\}$  is transformed into a vector  $\mathbf{V}_{\text{obj}}$  expressed in object-space coordinates through the product

$$\mathbf{V}_{\text{obj}} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \mathbf{V}_{\text{tan}}. \quad (5.19)$$

In per-pixel shading, calculations are performed in tangent space, so we often need to transform vectors from object space into tangent space. Since the matrix in Equation (5.19) is orthogonal, its transpose gives the inverse transformation that we need:

$$\mathbf{V}_{\text{tan}} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} \mathbf{V}_{\text{obj}}. \quad (5.20)$$

The types of vectors that are transformed into tangent space include the direction from a vertex to a light source and the direction from a vertex to the camera position. The tangent-space representations of these vectors are calculated in the vertex shader and interpolated over triangles to which they belong. In the vertex shader, Equation (5.20) is implemented as the three dot products  $\mathbf{T} \cdot \mathbf{V}_{\text{obj}}$ ,  $\mathbf{B} \cdot \mathbf{V}_{\text{obj}}$ , and  $\mathbf{N} \cdot \mathbf{V}_{\text{obj}}$ . This transformation must be performed twice in order to transform vectors into both of the tangent spaces arising from our texture mapping. Using the cross product

relationships shown in Equations (5.17) and (5.18), the dot products with the bitangents can be calculated as

$$\begin{aligned}\mathbf{B}_1 \cdot \mathbf{V}_{\text{obj}} &= (\mathbf{N} \times \mathbf{T}_1) \cdot \mathbf{V}_{\text{obj}} \\ \mathbf{B}_2 \cdot \mathbf{V}_{\text{obj}} &= (\mathbf{N} \times \mathbf{T}_2) \cdot \mathbf{V}_{\text{obj}}.\end{aligned}\quad (5.21)$$

By permuting the factors of the scalar triple products appearing in this equation, we can eliminate one of the cross products as follows:

$$\begin{aligned}\mathbf{B}_1 \cdot \mathbf{V}_{\text{obj}} &= (\mathbf{V}_{\text{obj}} \times \mathbf{N}) \cdot \mathbf{T}_1 \\ \mathbf{B}_2 \cdot \mathbf{V}_{\text{obj}} &= (\mathbf{V}_{\text{obj}} \times \mathbf{N}) \cdot \mathbf{T}_2.\end{aligned}\quad (5.22)$$

Expanding the tangent vectors into their components, the full transformation from object into each of the two tangent spaces is given by

$$\begin{aligned}\mathbf{T}_1 \cdot \mathbf{V}_{\text{obj}} &= \frac{N_x(\mathbf{V}_{\text{obj}})_y - N_y(\mathbf{V}_{\text{obj}})_x}{\sqrt{N_x^2 + N_y^2}} \\ \mathbf{T}_2 \cdot \mathbf{V}_{\text{obj}} &= \frac{N_z(\mathbf{V}_{\text{obj}})_x - N_x(\mathbf{V}_{\text{obj}})_z}{\sqrt{N_x^2 + N_z^2}} \\ \mathbf{B}_1 \cdot \mathbf{V}_{\text{obj}} &= \frac{N_x(\mathbf{V}_{\text{obj}} \times \mathbf{N})_y - N_y(\mathbf{V}_{\text{obj}} \times \mathbf{N})_x}{\sqrt{N_x^2 + N_y^2}} \\ \mathbf{B}_2 \cdot \mathbf{V}_{\text{obj}} &= \frac{N_z(\mathbf{V}_{\text{obj}} \times \mathbf{N})_x - N_x(\mathbf{V}_{\text{obj}} \times \mathbf{N})_z}{\sqrt{N_x^2 + N_z^2}}.\end{aligned}\quad (5.23)$$

The dot product  $\mathbf{N} \cdot \mathbf{V}_{\text{obj}}$  is the same for both transformations and only needs to be calculated once. A vertex shader implementing these calculations is shown in Listing 5.10. To avoid division by zero and the associated rendering artifacts of such an operation, we clamp the denominators in Equation (5.23) to a small  $\varepsilon$  value.

**Listing 5.10.** This snippet of GLSL vertex shader code transforms a constant object-space vector into both tangent spaces. Since the  $z$  component of the vector is the same in both tangent spaces, it is only calculated for the first output vector.

```

attribute vec3 nrml;           // The vertex normal.
uniform vec3 vector;          // The vector V to transform.
varying vec3 tanvec1;         // Output V in tangent space 1.
varying vec2 tanvec2;         // Output (Vx, Vy) in tangent space 2.

// mag = max((nx * nx + ny * ny, nx * nx + nz * nz), 1/32)
vec3 mag = nrml * nrml;
mag.xy = mag.x + mag.yz;
mag.xy = max(mag.xy, 0.03125);

// Calculate V x N one time.
vec3 vxn = cross(vector, nrml);

// Perform calculations from Equation (5.23).
temp.xy = nrml.xz * vector.yx - nrml.yx * vector.xz;
temp.zw = nrml.xz * vxn.yx - nrml.yx * vxn.xz;

tanvec1.xy = temp.xz * inversesqrt(mag.x);
tanvec2 = temp.yw, inversesqrt(mag.y);

// Calculate N * V. Same for both tangent spaces.
tanvec1.z = dot(nrml, vector);

```

### 5.5.3 Shading

The standard Blinn shading model [Blinn 1977] determines the color  $C$  rendered at a particular point  $\mathbf{p}$  by evaluating the formula

$$C = AI_{\text{ambient}} + \left[ D \max \{ \mathbf{N} \cdot \mathbf{L}, 0 \} + S (\max \{ \mathbf{N} \cdot \mathbf{H}, 0 \})^e \right] I_{\text{direct}}, \quad (5.24)$$

where the factors are defined as follows:

- $A$  is the fraction of ambient light reflected.
- $D$  is the fraction of direct light that is diffusely reflected.
- $S$  is the fraction of direct light that is specularly reflected.
- $e$  is the specular reflection exponent controlling the sharpness of the specularity.
- $\mathbf{N}$  is the unit normal vector at the point  $\mathbf{p}$ .
- $\mathbf{L}$  is the unit vector pointing from  $\mathbf{p}$  to the position of the light source.
- $\mathbf{H}$  is the unit halfway vector, defined as the normalized average between  $\mathbf{L}$  and the unit vector  $\mathbf{V}$  pointing from  $\mathbf{p}$  to the camera position.

- $I_{\text{ambient}}$  is the intensity of the ambient illumination. This could be constant or a function of position in the global coordinate space.
- $I_{\text{direct}}$  is the intensity of the direct illumination, possibly attenuated by the distance between  $\mathbf{p}$  and the position of the light source.

The values  $A$ ,  $D$ , and  $S$  are colors that can each be a constant, a sample fetched from a texture map, or a combination of both. (The values  $I_{\text{ambient}}$  and  $I_{\text{direct}}$  may also be colors.) In the case that they involve texture maps, the samples are fetched in the manner described in Section 5.5.1.

For a smooth surface, the vector  $\mathbf{N}$  is a barycentric interpolation of vertex normals, and it is renormalized in the fragment shader. The difference between the vertex position  $\mathbf{p}$  and the object-space light source position is calculated at each vertex, and its interpolated value is normalized in the fragment shader to form the vector  $\mathbf{L}$ . Similarly, the difference between the vertex position  $\mathbf{p}$  and the object-space camera position is interpolated and normalized to form the view vector  $\mathbf{V}$ . The halfway vector is then given by  $\mathbf{H} = (\mathbf{L} + \mathbf{V})/\|\mathbf{L} + \mathbf{V}\|$ . All of the vectors appearing in Equation (5.24) remain in object space throughout the shading process.

When a surface is rendered with bump mapping, the normal vector  $\mathbf{N}$  is fetched from a texture map. In the case of our terrain shader, it is a combination of up to six different normal vectors fetched from six different texture maps. These normal vectors are inherently expressed in tangent-space coordinates, so to calculate the dot products  $\mathbf{N} \cdot \mathbf{L}$  and  $\mathbf{N} \cdot \mathbf{H}$ , we need to have the vectors  $\mathbf{L}$  and  $\mathbf{H}$  in tangent space as well. However, the normal vectors belong to two different tangent spaces, and the dot products involving each normal vector must be evaluated in the correct tangent space induced by the texture coordinates used to fetch it.

For a vertex at the position  $\mathbf{p}$ , we define the vectors  $\mathbf{L}_{\text{obj}}$  and  $\mathbf{V}_{\text{obj}}$  as

$$\begin{aligned}\mathbf{L}_{\text{obj}} &= \mathbf{P}_{\text{light}} - \mathbf{p} \\ \mathbf{V}_{\text{obj}} &= \mathbf{P}_{\text{camera}} - \mathbf{p},\end{aligned}\tag{5.25}$$

where  $\mathbf{P}_{\text{light}}$  is the object-space position of the light source and  $\mathbf{P}_{\text{camera}}$  is the object-space position of the camera. Both of the vectors  $\mathbf{L}_{\text{obj}}$  and  $\mathbf{V}_{\text{obj}}$  need to be transformed into both tangent spaces in the vertex shader in the manner described in Section 5.5.2. The results become interpolants  $\mathbf{L}_{\text{tan}}^1$ ,  $\mathbf{L}_{\text{tan}}^2$ ,  $\mathbf{V}_{\text{tan}}^1$ , and  $\mathbf{V}_{\text{tan}}^2$  that are accessible by the fragment shader, where superscripts indicate the tangent space in which the coordinates are expressed. As demonstrated in Listing 5.11, these vectors are combined in the fragment shader to create the vectors  $\mathbf{L}_1$ ,  $\mathbf{L}_2$ ,  $\mathbf{H}_1$ , and  $\mathbf{H}_2$  for use in Equation (5.24).

**Listing 5.11.** The snippet of GLSL fragment shader code calculates the vectors  $\mathbf{L}_1$ ,  $\mathbf{L}_2$ ,  $\mathbf{H}_1$ , and  $\mathbf{H}_2$ . The interpolants  $\text{ltan1}$  and  $\text{ltan2}$  are the outputs of the vertex shader shown in Listing 5.10 for the vector  $\mathbf{L}_{\text{obj}}$ , and the interpolants  $\text{vtan1}$  and  $\text{vtan2}$  are the same outputs for the vector  $\mathbf{V}_{\text{obj}}$ .

```

varying vec3    ltan1, vtan1;
varying vec2    ltan2, vtan2;

vec3 l1 = normalize(ltan1);
vec3 l2 = normalize(vec3(ltan2.x, ltan2.y, ltan1.z));

vec3 h1 = normalize(l1 + normalize(vtan1));
vec3 h2 = normalize(l2 + normalize(vec3(vtan2.x, vtan2.y, vtan1.z)));

```

Let  $\mathbf{N}_x$ ,  $\mathbf{N}_y$ , and  $\mathbf{N}_z$  be normal vectors fetched for the texture coordinates  $(s_x, t_x)$ ,  $(s_y, t_y)$ , and  $(s_z, t_z)$  using the cubic projection. Since these normal vectors are not all expressed in the same tangent-space coordinate systems, they cannot be blended together in a manner similar to that given by Equation (5.4) before they participate in the dot products appearing in Equation (5.24). Instead, we calculate the dot products with vectors belonging to the appropriate tangent spaces first, and then blend the scalar results as follows:

$$\begin{aligned}\mathbf{N} \cdot \mathbf{L} &\mapsto b'_x(\mathbf{N}_x \cdot \mathbf{L}_1) + b'_y(\mathbf{N}_y \cdot \mathbf{L}_1) + b'_z(\mathbf{N}_z \cdot \mathbf{L}_1) \\ \mathbf{N} \cdot \mathbf{H} &\mapsto b'_x(\mathbf{N}_x \cdot \mathbf{H}_1) + b'_y(\mathbf{N}_y \cdot \mathbf{H}_1) + b'_z(\mathbf{N}_z \cdot \mathbf{H}_1).\end{aligned}\quad (5.26)$$

(Here, the expressions  $\mathbf{N} \cdot \mathbf{L}$  and  $\mathbf{N} \cdot \mathbf{H}$  appearing on the left-hand sides of the mappings do not represent actual dot products, but represent components of Equation (5.24) that are replaced by the right-hand sides.) In the case that two sets of normal vectors are fetched from texture maps, we first interpolate them using Equation (5.13) and then blend using Equation (5.26). This process is demonstrated in Listing 5.12.

## 5.6 Results

Figure 5.12 shows a terrain surface rendered with the shading techniques developed in this chapter. The same terrain is shown with and without tangent-space normal mapping applied. Only the diffuse reflection terms of Equation (5.24) are calculated in this pair of images.

A specular reflection term is demonstrated in Figure 5.13, where we again show a terrain surface with and without normal mapping. The specularity has been exaggerated beyond the levels that would ordinarily be used so that the bumps become very apparent. This makes it easy to see the smooth transition where textures corresponding to different projection planes are blended together.

**Listing 5.12.** The snippet of GLSL fragment shader code fetches two sets of three normal vectors from texture maps, interpolates them using the value  $\alpha$  shown in Figure 5.8, and expands their components to the range  $[-1, 1]$ . It then evaluates the dot products  $\mathbf{N} \cdot \mathbf{L}$  and  $\mathbf{N} \cdot \mathbf{H}$  using Equation (5.26) and the values calculated in Listing 5.11. The results are stored in the variables `diff` and `spec`, respectively.

```

uniform sampler2DArray textureArray;
vec3 temp;

// Fetch primary and secondary Nx, interpolate, and expand.
vec3 tmp1 = texture2DArray(textureArray, texcoord1.xzw).xyz;
vec3 tmp2 = texture2DArray(textureArray, texcoord2.xzw).xyz;
vec3 nx = lerp(tmp1.xyz, tmp2.xyz, material1.z) * 2.0 - 1.0;

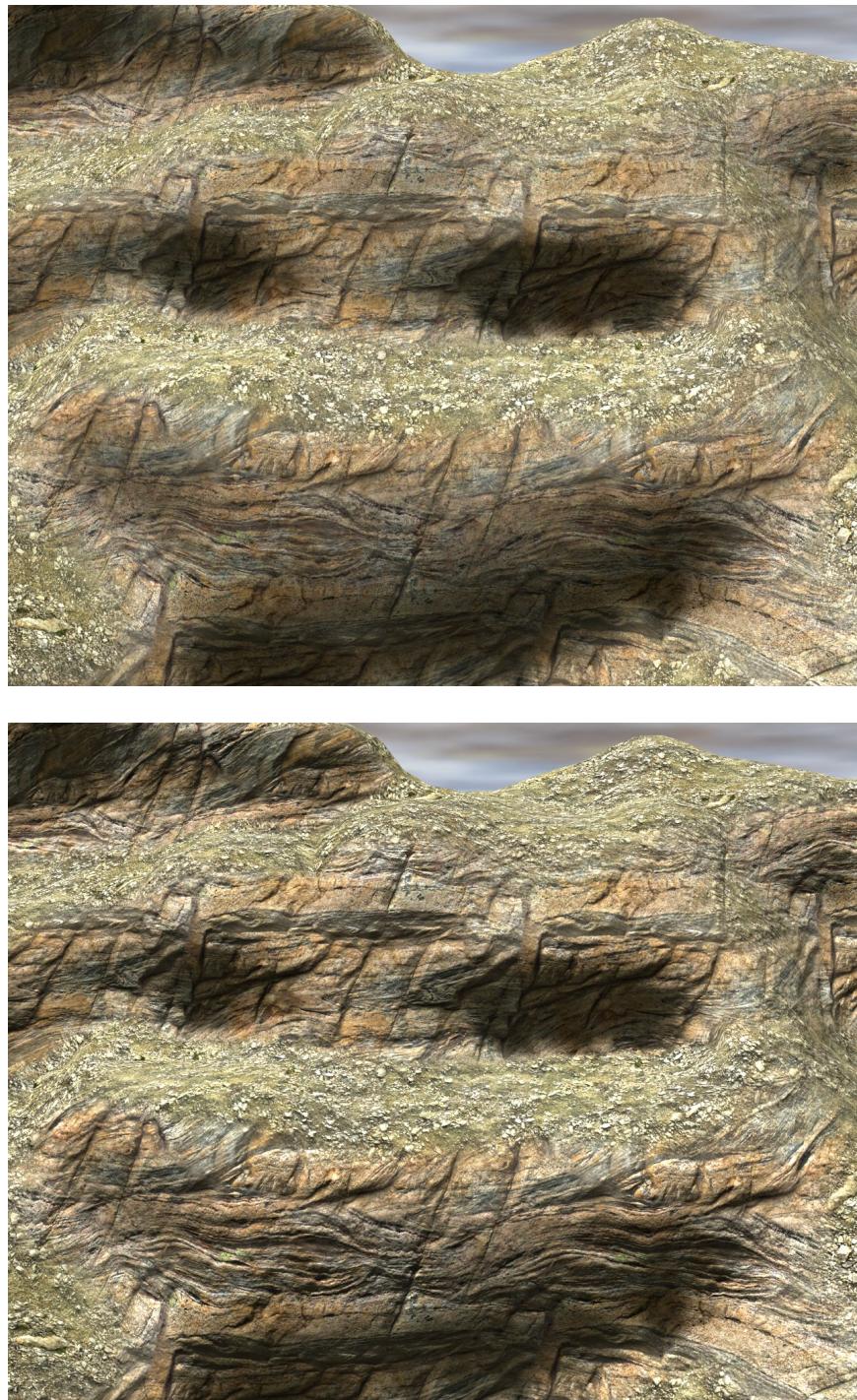
// Fetch primary and secondary Ny, interpolate, and expand.
tmp1 = texture2DArray(textureArray, texcoord1.yzw).xyz;
tmp2 = texture2DArray(textureArray, texcoord2.yzw).xyz;
vec3 ny = lerp(tmp1.xyz, tmp2.xyz, material1.z) * 2.0 - 1.0;

// Fetch primary and secondary Nz, interpolate, and expand.
tmp1 = texture2DArray(textureArray, texcoord3.xyz).xyz;
tmp2 = texture2DArray(textureArray, texcoord3.xyw).xyz;
vec3 nz = lerp(tmp1.xyz, tmp2.xyz, material1.z) * 2.0 - 1.0;

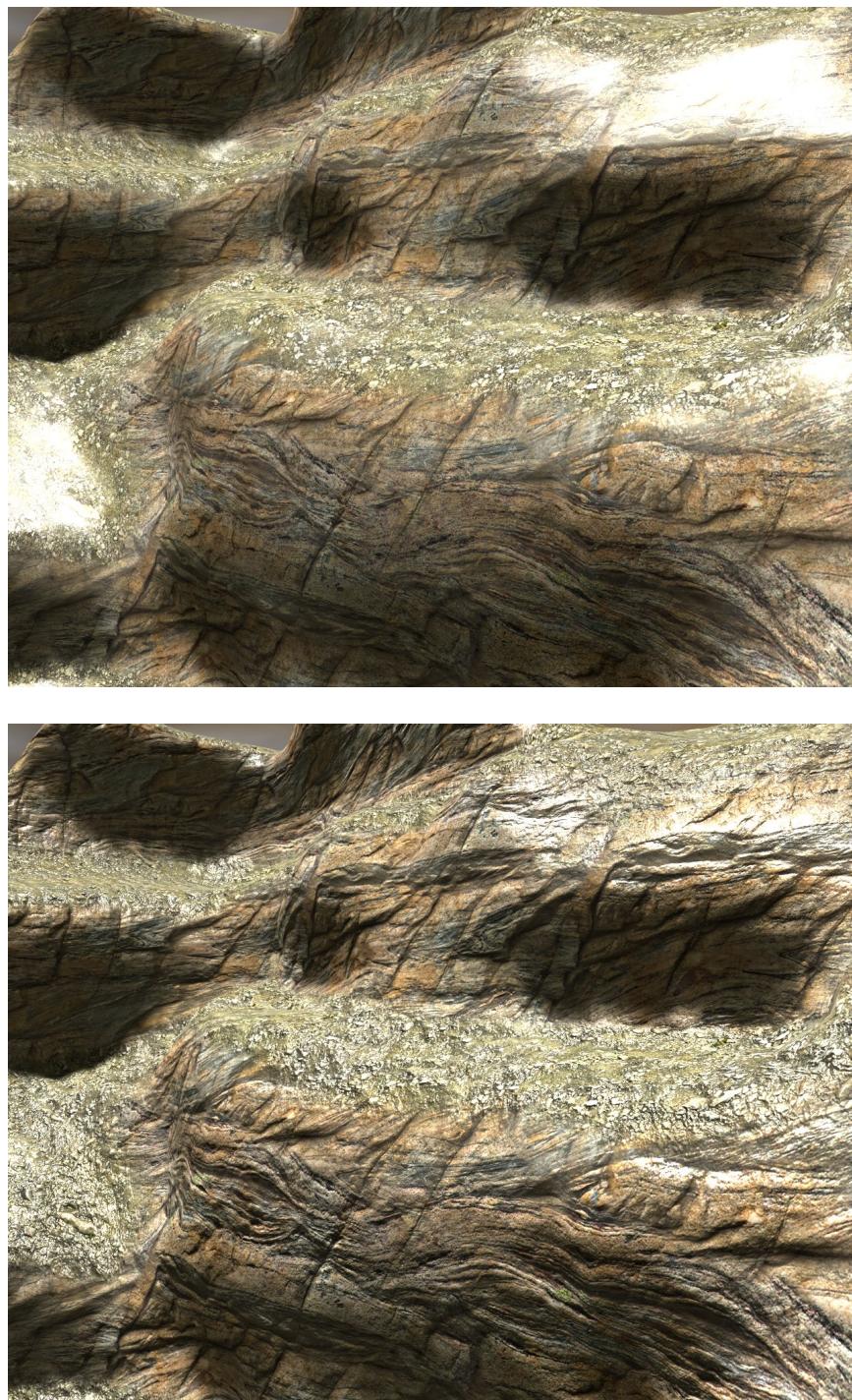
// Calculate blended N dot L.
temp.x = saturate(dot(nx, l1));
temp.y = saturate(dot(ny, l1));
temp.z = saturate(dot(nz, l2));
float diff = dot(temp, blend);

// Calculate blended N dot H.
temp.x = saturate(dot(nx, h1));
temp.y = saturate(dot(ny, h1));
temp.z = saturate(dot(nz, h2));
float spec = dot(temp, blend);

```



**Figure 5.12.** A terrain surface rendered (top) without normal mapping and (bottom) with normal mapping. Only diffuse reflection terms have been included in this pair of images.



**Figure 5.13.** A terrain surface rendered (top) without normal mapping and (bottom) with normal mapping. An exaggerated specular reflection term has been applied to make the bump mapping more apparent.

# Chapter 6

## Future Work

In this chapter, we discuss four topics related to voxel-based terrain that may be the focus of our continued research and engineering efforts. These topics generally pertain to generating higher-quality visual results and providing better control over the shape of geometric features and texture mapping.

### 6.1 Geomorphing

The vertex placement technique described in Section 4.2 ensures that every vertex belonging to a low-detail terrain mesh coincides exactly with a vertex belonging to the corresponding highest-detail mesh. Nevertheless, popping between different levels of detail can still be quite visible as the camera moves through a scene. Geomorphing is an effective method for lessening the popping effects caused by the sudden transition from one detail level to another, and it works by smoothly moving vertices in the higher detail level to the nearest point on the surface of the lower detail level. This motion is a function of camera position only, and it is very efficiently implemented in a vertex program.

The main problem associated with geomorphing is the extra space required to store a secondary position for every vertex in a terrain mesh. It may also be necessary to store a secondary normal vector to prevent a shading pop that could occur even though the morphed vertices lie in the surface of the lower-detail terrain mesh when a LOD transition occurs. Even if a 16-bit-per-component format is used so that a vertex position and normal vector both occupy six bytes of storage, an extra 12 bytes per vertex can be too costly, especially on platforms such as game consoles where memory is limited.

One solution that would reduce the secondary vertex position to a single scalar value is to morph the vertex only along its normal vector. When the terrain mesh is built, we would construct a line using the vertex position and normal, find all of the intersections with the lower-detail mesh in the cell containing the high-detail vertex, and select the closest intersection point whose interpolated normal vector makes a positive dot product with the normal vector of the high-detail vertex. (It is unclear whether this point always exists.)

Unfortunately, there is no analogous solution for the morph between the primary and secondary normal vectors. However, it may be possible to encode an 8-bit tangent offset and an 8-bit angle in

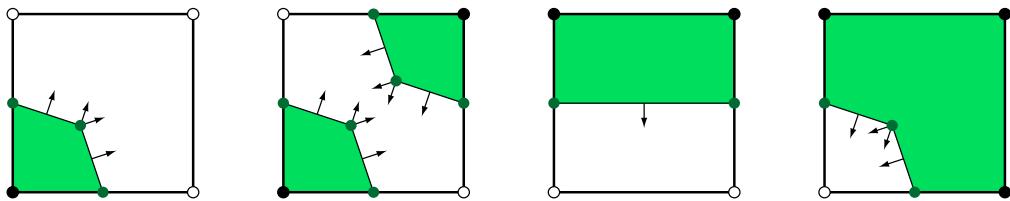
a single 16-bit scalar value. These could be decoded in a vertex program and used to morph the normal vector as the vertex position is morphed. The combined data for a morph that eliminates both geometric pops and shading pops would then be only four bytes per vertex.

## 6.2 Sharp Features

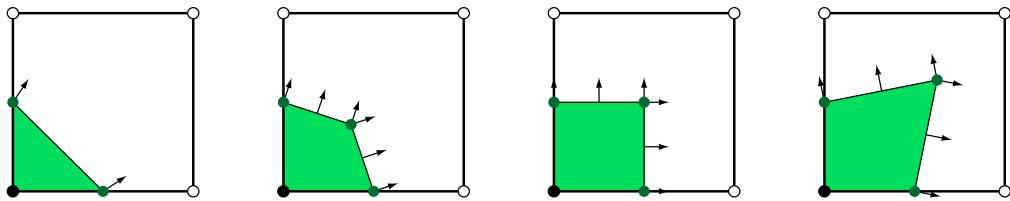
The original Marching Cubes algorithm is known for its inability to extract sharp features from volumetric data. A terrain surface generated by our modified Marching Cubes algorithm is generally smooth and free of sharp edges and corners, but this is sometimes not the desired result. An artist may wish to model a rocky surface with sharp features in the same vicinity as smooth areas covered by dirt or grass.

Several methods have been developed for capturing sharp features during polygonalization, and they include Extended Marching Cubes [Kobbelt et al. 2001], Dual Marching Cubes algorithm [Nielson 2004], and Dual Contouring [Ju et al. 2002]. Each of these techniques involves placing extra vertices in the interior of a cell in order to generate edges having a smaller dihedral angle. Using similar methods, the per-face edge configurations shown in Figure 3.5 are transformed into those shown in Figure 6.14, and these would be applied to triangulations for both regular cells and transition cells. The positions of the interior vertices are determined by the normal vectors at the vertices lying on the cell edges, as shown in Figure 6.15. In order to create a sharp edge while shading, interior vertices would ordinarily be duplicated and assigned different normal vectors corresponding to their adjacent faces. Whether this duplication occurs can be controlled by setting a dihedral angle threshold above which a smoothed normal vector is used instead. This prevents faceting on smoothly curving surfaces while also allowing sharp edges to be produced. It is also possible to eliminate the interior vertex altogether if the dihedral angle is close to 180 degrees, as in the leftmost example in Figure 6.15.

So that an artist could choose whether smooth or sharp triangulations are produced on a per-cell basis, we could introduce a new channel in the voxel data that stores an 8-bit sharpness parameter. A value of 0 would indicate a smooth surface generated without any interior vertices, and a value of 255 would indicate the maximum sharpness. For intermediate values, the positions of interior vertices would be interpolated between the position for maximum sharpness and its projection onto the edge that would be created without an interior vertex. Sharpness would need to be determined independently for each edge based on interpolated sharpness values stored for each corner of a cell, and this would greatly increase the number of triangulations that could be generated for each equivalence class.



**Figure 6.14.** These are the four possible edge configurations on a nontrivial cell face when using methods to extract sharp features.

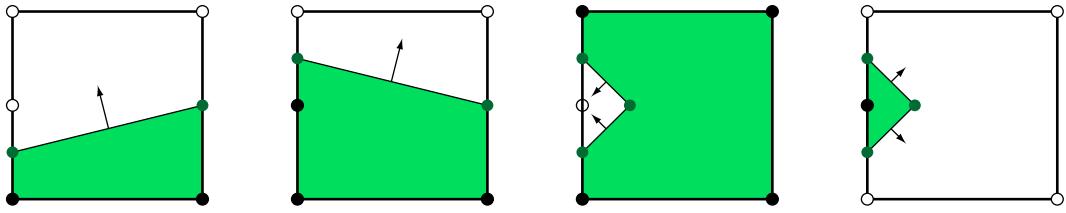


**Figure 6.15.** The normal vectors for the vertices lying on cell edges determine the position of the interior vertex. In the leftmost example, the interior vertex is removed because the dihedral angle is close to 180 degrees. In the other three examples, the interior vertex is duplicated, and each copy is assigned the normal vector of one of the adjacent faces to produce a sharp edge while shading.

### 6.3 Enhanced Transition Triangulations

Of the 73 transition cell equivalence classes shown in Tables 4.2 through 4.7, 45 generate triangles that lie entirely in the plane of the full-resolution face. Even though the terrain mesh is watertight in these cases, the vertex normals on either side of the coplanar triangles can be strongly opposing, and this tends to give texturing and shading a poor appearance on those triangles. Fortunately, these triangles usually occur in somewhat hidden spots in the terrain and are only rendered far away from the camera at a level-of-detail boundary, so they often go unnoticed. If a minimal vertex count is critical, then we make no changes to alleviate the problem, but otherwise, we can construct meshes that give us better shading by adding vertices to the interiors of transition cells.

The key observation to make is that triangles coplanar with the full-resolution face almost always occur when an alternating edge arises, where an alternating edge is an edge on the full-resolution face whose center voxel sample has the opposite inside/outside state of its neighbors either above and below or to the right and left. (The only exceptions are the four special classes shown in Table 4.7.) In order to generate triangulations that do not include triangles coplanar with the full-resolution face, we can modify the set of edge configurations allowed on lateral faces, previously shown in Figure 4.10, so that a new interior vertex is created whenever a lateral face has an alternating edge. The new set of edge configurations is shown in Figure 6.16. There are 16 equivalence



**Figure 6.16.** The four nontrivial edge configurations allowed on a lateral face shown in Figure 4.10 are replaced by this set of configurations for an enhanced transition cell. An interior vertex is added in each of the two rightmost configurations, and the edge that originally laid on the boundary between the lateral face and the full-resolution face is replaced by two edges that extend inward.

classes for which an alternating edge occurs on a lateral face, but it is not connected to any triangles coplanar with the full-resolution face. In these cases, we would still need to use the new edge configurations to make sure that adjacent transition cells always match on lateral faces.

Table 6.3 shows several examples in which the original transition cell triangulations are modified to follow the new lateral face rules shown in Figure 6.16. There are 15 equivalence classes for which the center voxel sample on the full-resolution face is covered by coplanar triangles, excluding the four special classes shown in Table 4.7. In these cases, we would create an additional interior vertex for the center location and use it in the new triangulations as shown in the table. The interior vertices are all placed at the same distance from the full-resolution face in all but one special case—for equivalence class #45, the interior vertex corresponding to the center voxel is moved in further than the others to avoid creating a flattened area parallel to the full-resolution face.

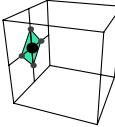
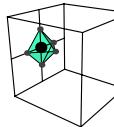
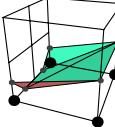
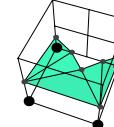
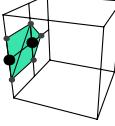
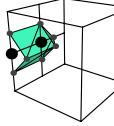
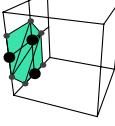
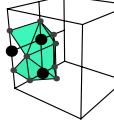
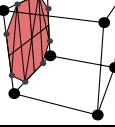
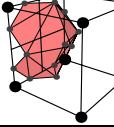
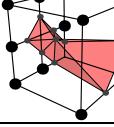
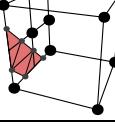
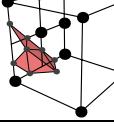
## 6.4 Texture Map Orientation

The methods developed in Chapter 5 only allow texture maps to appear at a fixed orientation relative to the voxel grid. If an artist wanted to paint a texture map onto the terrain in a different orientation, the only option would be to include pre-rotated versions of the same texture map as a separate entry in the texture array or texture palette. However, this still would not allow a texture to follow a curve representing the course of a path or roadway.

There are two approaches from which we can attempt to solve this problem. The first is to store additional channels of information in the voxel map that specify a texture coordinate transformation of some kind. Since, at very least, a rotation angle and a two-dimensional offset vector would be required at each voxel in order to provide arbitrary texture orientation along specified curves, the amount of new volumetric information could be prohibitively large. We might also need to store separate data for each of the three planes for which texture coordinates are generated.

A better approach may be to store minimal information about various flow fields separately from the voxel data. For example, a space curve with a radius function could represent a path on the terrain for which the direction of the  $s$  texture coordinate is always aligned to the curve's tangent direction. When the mesh is generated for a terrain block, we would determine which curves intersect the block and generate per-vertex texture coordinates to match them. Since texture coordinates need to be stored with high precision, they would only be generated for vertices belonging to blocks that have at least one curve running through them in order to save space.

**Table 6.3.** These are several examples showing how the transition cell triangulations change when the edge configurations for lateral faces follow the rules given by Figure 6.16 and interior vertices are added.

Class Index	Original Triangulation	Enhanced Triangulation
#3		
#5		
#10		
#25		
#45		
#51		
#52		

# Chapter 7

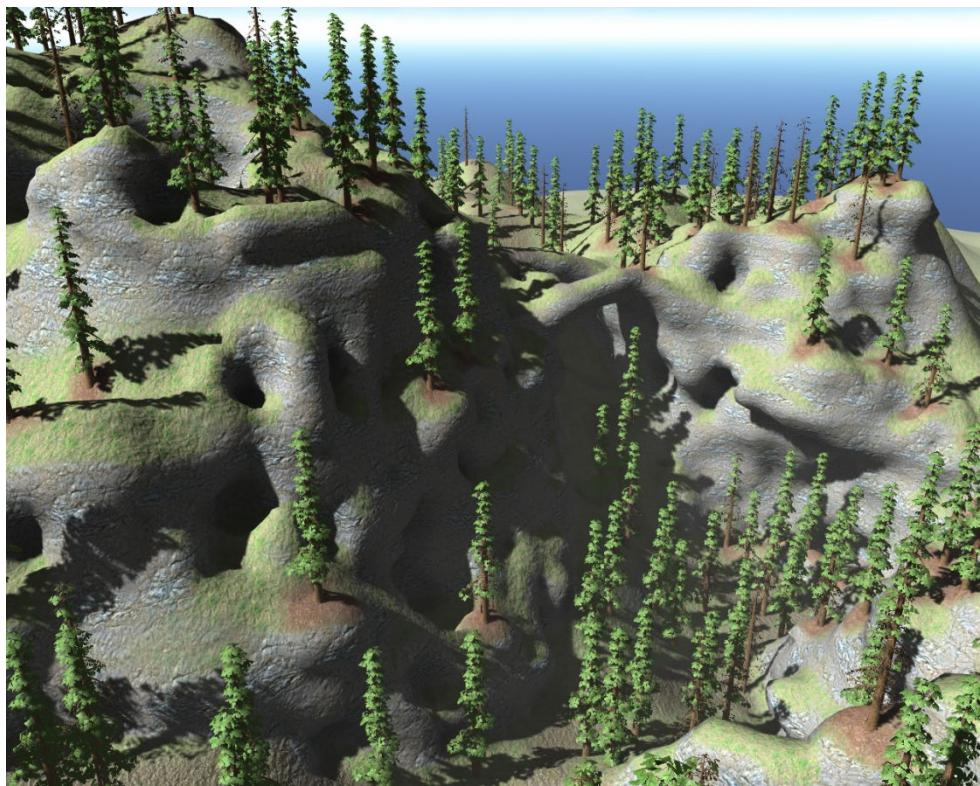
## Conclusions

This dissertation provides a complete set of robust and practical methods for constructing and rendering large voxel-based terrain in a real-time virtual simulation. The real-world implementation of such a system relies upon the following ingredients:

- A fast isosurface extraction method that dispenses with the difficulties associated with ambiguous faces greatly simplifies various aspects of the terrain construction process. This is provided by the modified Marching Cubes method discussed in Chapter 3.
- A level-of-detail system that aims to limit the viewport-space density of vertices, triangles, and individual meshes is essential for high rendering performance. The Transition Cubes algorithm introduced in Chapter 4 provides a method for seamlessly stitching multiresolution voxel-based meshes, and the set of equivalence classes arising in this algorithm is kept as simple as possible by our insistence on a preferred polarity for ambiguous faces in our modified Marching Cubes algorithm.
- A set of texture mapping and shading methods allowing a large number of different materials to be applied to a terrain surface without smearing artifacts is necessary for terrain to have a high-quality visual appearance. The methods described in Chapter 5 provide this functionality across multiple generations of graphics hardware.

The Transition Cubes algorithm introduced in this dissertation has also been described in the technical paper “Transition Cells for Multiresolution Voxel Terrain Rendering”, and it is under review at the time of this writing. Additional papers discussing the triangulation of low-resolution cells and techniques for texturing and shading are planned.

The full content of this dissertation has been implemented and deployed in the *C4 Engine*, a multi-platform computer game engine [Lengyel 2005]. The robustness of the techniques we have developed has been field tested and proven by a user base consisting of thousands of engineers, designers, and artists working on many types of applications. We conclude with the screenshot from the C4 Engine in Figure 7.1.



**Figure 7.17.** This screenshot taken from the C4 Engine shows a large voxel-based terrain having three levels of detail rendered in real-time with several thousand trees and multiresolution cascaded shadow maps.

# Appendix A

## Editing Voxel Terrain

This dissertation provides the details for constructing a terrain mesh from volumetric data and rendering it, but it says nothing about how that data is created in the first place. We have implemented a set of voxel terrain editing tools in the C4 Engine [Lengyel 2005], and this appendix describes how they work.

### A.1 Terrain Creation Process

A voxel map is created in the C4 Engine by drawing a box that encloses the entire terrain volume and then specifying a resolution. The data in the voxel map can be initialized in two different ways. First, it can be initialized so that the terrain surface is simply a horizontal plane cutting halfway through the cells at a given  $z$  coordinate. The voxels below the plane are initialized to  $-1$ , and the voxels above the plane are initialized to  $+1$ . Second, the voxel map can be initialized so that it matches a height field read from a grayscale image file. For cells containing the terrain surface, the voxel values are set so that the interpolated values along vertical cell edges yield the value zero at the correct height for any  $(x, y)$  coordinates in the height field.

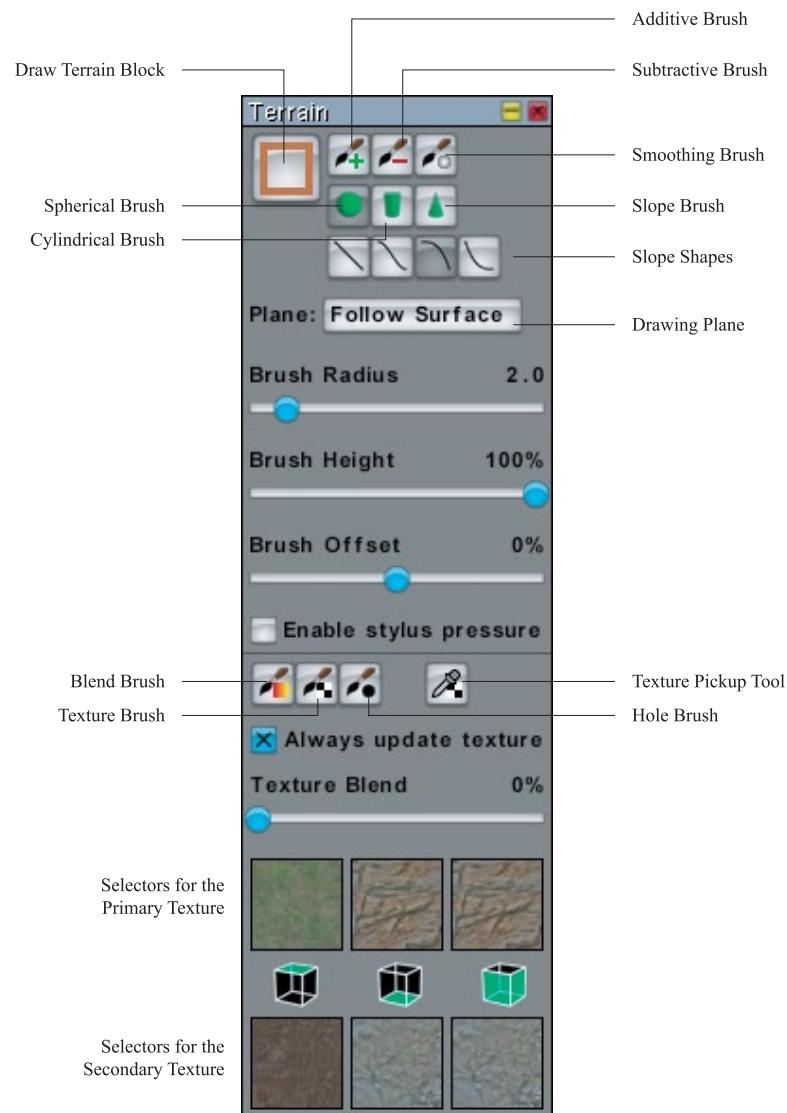
The material data for the entire voxel map is initialized to a single material identifier with a blend factor of zero. The texture selections for this material identifier are those selected in the terrain tool panel (see Figure A.18) at the time the terrain is created.

### A.2 Terrain Editing Tools

Once a voxel map has been created, it can be edited using a number of drawing tools. These tools are displayed in the terrain tool panel shown in Figure A.18. The user modifies the voxel map by setting attributes for a three-dimensional paint brush and then drawing in a viewport using a mouse or tablet.

There are six different types of paint brushes, and they affect the three channels of voxel data in different ways. (Recall that the three channels are distance, material ID, and blend factor.) They have the following functions:

- The additive brush increases values in the distance channel so that solid formations are added to the terrain. If we consider the brush to be a small 3D voxel map itself, then the actual operation applied to the terrain is a maximum between the existing voxel values and those belonging to the paint brush.
- The subtractive brush decreases values in the distance channel so the empty space is created. The actual operation applied here is a minimum.
- The smoothing brush applies a three-dimensional blur filter to the distance channel in order to smooth the terrain surface.



**Figure A.18.** The terrain tool panel.

- The blend brush sets the values in the blend channel to the current value specified by the texture blend slider (scaled to the range  $[0, 255]$ ).
- The texture brush sets the values in the material ID channel to match the current texture map selection at the bottom of the tool panel. If the “Always update texture” box is checked, then every other paint brush also updates the material ID wherever it is applied.
- The hole brush sets the values in the material ID channel to the identifier for the poison material so that holes can be painted onto the terrain surface. A hole can be removed by painting an ordinary material over the same region with the texture brush.

In addition to these six paint brushes, there is a texture pickup tool that can be used to grab the material information from the voxel map at the point where the mouse is clicked.

The shape and size of each paint brush is controlled by several settings in the tool panel. First, the general shape is selected by the spherical brush, cylindrical brush, and slope brush buttons, and four different slope functions can be selected for the slope brush. The radius of each brush shape is specified, in units of voxels, by the brush radius slider, and the height of a brush is specified, as a percentage of the brush diameter, by the brush height slider.

The drawing plane can be set to four different values:

- **Horizontal plane.** The brush is constrained to move only in the  $x$  and  $y$  directions.
- **Tangent plane.** The brush is constrained to the plane tangent to the terrain surface where the mouse is first clicked.
- **Camera plane.** The brush is constrained to move only in directions perpendicular to the current camera view direction.
- **Follow surface.** The brush always stays on the terrain surface that existed before the mouse was first clicked. Changes made by the brush do not affect the brush position until the user releases the mouse and begins drawing again.

The brush offset slider controls the position of each paint brush with respect to the current drawing plane. A value of  $-100\%$  means the brush is completely submerged beneath the surface, and a value of  $+100\%$  means the brush is just grazing the surface on the outside. Naturally, a value of  $0\%$  means the brush is split halfway between inside and outside. If the “Enable stylus pressure” box is checked, then the brush offset is dynamically adjusted based on the current pressure exerted by the user on a tablet device, if available. For an additive brush, the offset is increased with greater pressure, and for a subtractive brush, the offset is decreased with greater pressure.

The texture selectors at the bottom of the tool panel are used to select two sets of texture maps for the triplanar projection. In each set, one texture map can be chosen to be applied to the positive  $z$  direction, one can be chosen to be applied to the negative  $z$  direction, and one can be chosen to be applied to all four directions in the  $x$ - $y$  plane. Clicking on a texture image causes a palette to appear from which a texture map can be selected from the available palette entries. Textures from the two sets are mixed on the terrain using the blend brush and the texture blend slider.

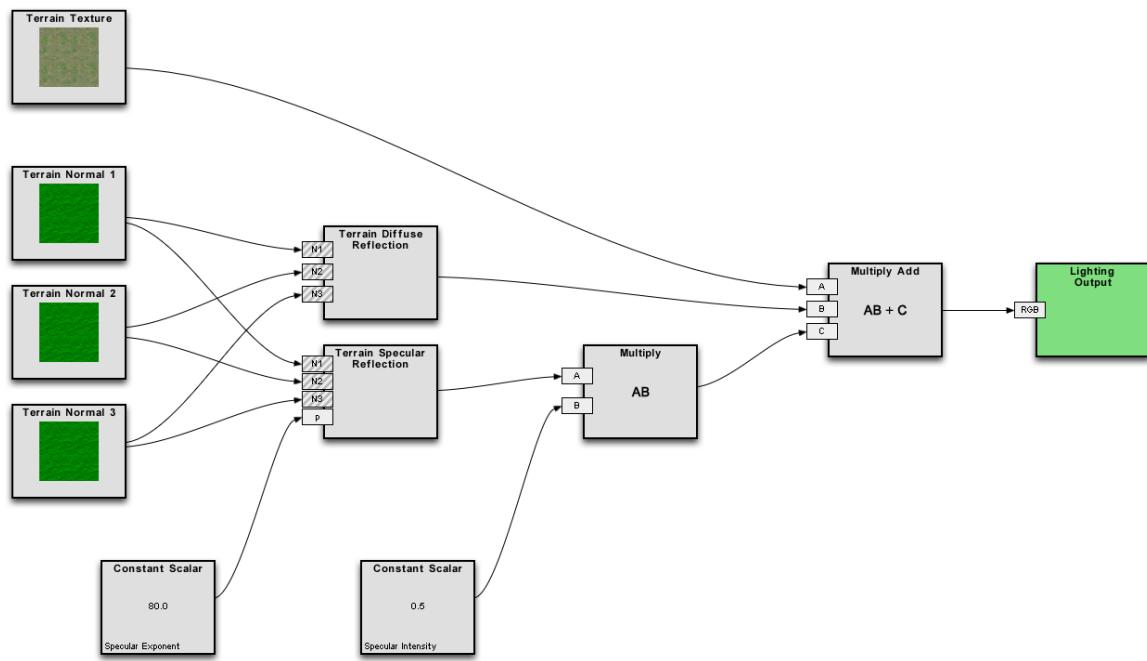
### A.3 Terrain Shaders

The C4 Engine includes a graphical fragment shader editor that provides an abstraction of the underlying textual shading language passed to the low-level rendering library. We added special types of nodes to this editor that encapsulate all of the shader code developed in Chapter 5 so that the user need not be concerned with the internals of texture fetches from texture arrays or texture palettes and the specifics of triplanar blending.

An example fragment shader graph showing the operations for bump-mapped diffuse and specular reflection is shown in Figure A.19. The “Terrain Texture” node fetches color samples from the texture maps assigned to the triangle being rendered and performs triplanar blending based on the interpolated normal vector. This node can be configured to blend between two different sets of textures using the blend factor  $\alpha$  or selecting only one set or the other.

The three “Terrain Normal” nodes function a little differently. Each one fetches normal vector samples from one or two of the texture maps assigned to just one plane of the triplanar projection. As before, these samples can be blended, or just one texture map can be sampled and have its value passed through by itself. The results of the “Terrain Normal” nodes are passed into nodes called “Terrain Diffuse Reflection” and “Terrain Specular Reflection”. These two nodes calculate the quantities  $\max\{\mathbf{N} \cdot \mathbf{L}, 0\}$  and  $(\max\{\mathbf{N} \cdot \mathbf{H}, 0\})^e$  for each input normal vector in the appropriate tangent space and then perform triplanar blending on the results in the manner shown in Listing 5.12.

The presence of the “Terrain Texture” and “Terrain Normal” nodes in the graph implicitly cause the three sets of texture coordinates and the triplanar blending weights to be generated and shared among the nodes that need them.



**Figure A.19.** A terrain fragment shader graph that includes bump-mapped diffuse and specular reflection.

# Bibliography

- ASIRVATHAM, ARUL, AND HOPPE, HUGUES. 2005. "Terrain Rendering Using GPU-Based Geometry Clipmaps". *GPU Gems 2*, Addison-Wesley, Chapter 2, pp. 27–45.
- BLINN, JAMES F. 1977. "Models of Light Reflection for Computer Synthesized Pictures". *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 192–198.
- CHERNYAEV, EVGENI V. 1995. "Marching Cubes 33: Construction of Topologically Correct Isosurfaces". Technical Report CERN CN 95-17, CERN.
- CRASSIN, CYRIL; NEYRET, FABRICE; LEFEBVRE, SYLVAIN; AND EISEMANN, ELMAR. 2009. "Giga-Voxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering". *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pp. 15–22.
- CRASSIN, CYRIL; NEYRET, FABRICE; LEFEBVRE, SYLVAIN; SAINZ, MIGUEL; AND EISEMANN, ELMAR. 2009. "Beyond Triangles : Gigavoxels Effects In Video Games". SIGGRAPH 2009 Technical Talk.
- DICK, CHRISTIAN; KRÜGER, JENS; AND WESTERMANN, RÜDIGER. 2009. "GPU Ray-Casting for Scalable Terrain Rendering". *Proceedings of Eurographics 2009*, pp. 43–50.
- DUCHAINEAU, MARK; WOLINSKY, MURRAY; SIGETI, DAVID E.; MILLER, MARK C.; ALRICH, CHARLES; AND MINEEV-WEINSTEIN, MARK B. 1997. "ROAMing Terrain: Real-time Optimally Adapting Meshes". *Proceedings of the 8th Conference on Visualization '97*, pp. 81–88.
- DÜRST, M. J. 1988. "Additional reference to marching cubes". *Computer Graphics*, Volume 22, Number 2, pp. 72–73.
- GEISS, RYAN. 2008. "Generating Complex Procedural Terrains Using the GPU". *GPU Gems 3*, Addison-Wesley, Chapter 1, pp. 7–37.
- GEISS, RYAN, AND THOMPSON, MICHAEL. 2007. "NVIDIA Demo Team Secrets—Cascades". Game Developers Conference. <http://developer.download.nvidia.com/presentations/2007/gdc/CascadesDemoSecrets.zip>
- HAZEWINKEL, MICHIEL (editor). 2001. "Poincaré–Hopf Theorem". Encyclopaedia of Mathematics, Kluwer Academic Publishers.

- HOPPE, HUGUES. 1996. "Progressive Meshes". *Proceedings of SIGGRAPH 1996*, pp. 99–108.
- HOPPE, HUGUES. 1998. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering". *Proceedings of the Conference on Visualization '98*, pp. 35–42.
- JU, TAO; LOSASSO, FRANK; SCHAFER, SCOTT; AND WARREN, JOE. 2002. "Dual Contouring of Hermite Data". *Proceedings of SIGGRAPH 2002*, pp. 339–346.
- KAZHDAN, MICHAEL; KLEIN, ALLISON; DALAL, KETAN; AND HOPPE, HUGUES. 2007. "Unconstrained Isosurface Extraction on Arbitrary Octrees". *Proceedings of the 5th Eurographics Symposium on Geometry Processing*, pp. 125–133.
- KOBBELT, LEIF P.; BOTSCHE, MARIO; SCHWANECKE, ULRICH; AND SEIDEL, HANS-PETER. 2001. "Feature Sensitive Surface Extraction from Volume Data". *Proceedings of SIGGRAPH 2001*, pp. 57–66.
- LEECH, JON AND KILGARD, MARK. 2008. `GL_EXT_texture_array` OpenGL extension specification. [http://www.opengl.org/registry/specs/EXT/texture\\_array.txt](http://www.opengl.org/registry/specs/EXT/texture_array.txt)
- LENGYEL, ERIC. 2002. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, Chapter 6.
- LENGYEL, ERIC. 2005. *C4 Engine*. Terathon Software LLC. <https://c4engine.com/>
- LINDSTROM, PETER; KOLLER, DAVID; RIBARSKY, WILLIAM; HODGES, LARRY F.; FAUST, NICK; AND TURNER, GREGORY A. 1996. "Real-Time, Continuous Level of Detail Rendering of Height Fields". *Proceedings of SIGGRAPH 1996*, pp. 109–118.
- LIVNY, YOTAM; KOGAN, ZVI; AND EL-SANA, JIHAD. 2009. "Seamless Patches for GPU-Based Terrain Rendering". *The Visual Computer: International Journal of Computer Graphics*, Volume 25, Number 3, pp. 197–208.
- LORENSEN, WILLIAM E., AND CLINE, HARVEY E. 1987. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". *Computer Graphics (Proceedings of SIGGRAPH 87)*, Volume 21, Issue 4, pp. 163–169.
- LOSASSO, FRANK AND HOPPE, HUGUES. 2004. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids". *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, Volume 23, Issue 3, pp. 769–776.
- NEWMAN, TIMOTHY S., AND YI, HONG. 2006. "A Survey of the Marching Cubes Algorithm". *Computers & Graphics*, Volume 30, Number 5, pp. 854–879.
- NIELSON, GREGORY M. 2004. "Dual Marching Cubes". *Proceedings of the Conference on Visualization '04*, pp. 489–496.

- NIELSON, GREGORY M., AND HAMANN, BERND. 1991. "The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes". *Proceedings of the 2nd Conference on Visualization '91*, IEEE Computer Society Press, pp. 83–91.
- NING, PAUL, AND BLOOMENTHAL, JULES. 1993. "An Evaluation of Implicit Surface Tilers". *IEEE Computer Graphics and Applications*, Volume 13, Number 6, pp. 33–34.
- PAJAROLA, RENATO, AND GOBBETTI, ENRICO. 2007. "Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering". *The Visual Computer: International Journal of Computer Graphics*, Volume 23, Number 8, pp. 583–605.
- RÖTTGER, STEFAN; HEIDRICH, WOLFGANG; SLUSALLEK, PHILIPP; AND SEIDEL, HANS-PETER. 1998. "Real-Time Generation of Continuous Levels of Detail for Height Fields". *Proceedings of WSCG '98*, pp. 315–322.
- SEGAL, MARK, AND AKELEY, KURT. 2009. *The OpenGL Graphics System: A Specification, Version 3.2*. The Khronos Group.
- SCHNEIDER, JENS, AND WESTERMANN, RÜDIGER. 2006. "GPU-Friendly High-Quality Terrain Rendering". *Journal of WSCG*, Volume 14, pp. 49–56.
- SHEKHAR, RAJ; FAYYAD, ELIAS; YAGEL, RONI; AND CORNHILL, J. FREDRICK. 1996. "Octree-Based Decimation of Marching Cubes Surfaces". *Proceedings of the 7th Conference on Visualization '96*, pp. 335–342.
- SHU, RENBEN; ZHOU, CHEN; AND KANKANHALLI, MOHAN S. 1995. "Adaptive Marching Cubes". *The Visual Computer*, Volume 11, pp. 202–217.
- STANLEY, RICHARD P. 2001. *Enumerative Combinatorics, Volume 2*. Cambridge University Press, Chapter 6.
- TANNER, CHRISTOPHER C.; MIGDAL, CHRISTOPHER J.; AND JONES, MICHAEL T. 1998. "The Clip-map: A Virtual Mipmap". *Proceedings of SIGGRAPH 1998*, pp. 151–158.
- ULRICH, THATCHER. 2002. "Super-size it! Scaling up to Massive Virtual Worlds". SIGGRAPH 2002 Courses.
- UNITED STATES GEOLOGICAL SURVEY. 2006. National Elevation Dataset. <http://ned.usgs.gov/>
- VAN GELDER, ALLEN, AND WILHELM, JANE. 1994. "Topological Considerations in Isosurface Generation". *ACM Transactions on Graphics*, Volume 13, Issue 4, pp. 337–375.

## Author contact information

Eric Lengyel  
Terathon Software  
[lengyel@terathon.com](mailto:lengyel@terathon.com)



All mathematical expressions in this document  
were typeset with the [Radical Pie](#) equation editor.  
(Updated in April 2025.)

© 2010 Eric Lengyel (the Author).

The Author provides this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at <https://creativecommons.org/licenses/by-nd/3.0/>. The Author further grants permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.

