OpenGL® Volumizer Programmer's Guide

Document Number 007-3720-001

CONTRIBUTORS

Written by George Eckel
Illustrated by Dany Galgani and others
Edited by Christina Cary
Production by Carlos Miqueo
Engineering contributions by Robert Grzeszczuk

OpenGL® Volumizer Programmer's Guide
Document Number 007-3720-001

Contents

List of Figures xi

About This Guide xiii
Audience for This Guide xiii
How to Use This Guide xiii
What This Guide Contains xiv
Part One: Using the Volumizer API xiv
Part Two: Advanced Topics xiv
Recommended Reference Materials xv
Silicon Graphics Publications xv
Third-Party Publications xv
Conventions Used in This Guide xvi

1. **Volumizer Advantages** 3
Volumizer Advantages 4
Faster Development Cycle 5
High Performance 5
Handles Large Data Sets 5
Easy-to-Use API 5
Low-Level Control 6
Familiar Environment 6
Cross-Platform Support 6
Interoperability With Other Toolkits 6
Multi-Pipe Capability 7
Feature-Rich API 7
Preparing for the Future 7
Where OpenGL Volumizer Fits In 8

2. Basic Concepts in Volumizer 9
Volumetric vs. Surface-Based Models 9
Volumetric Properties 11
Decoupling Geometry and Appearance 12
Advantage of Separating Geometry and Appearance 14
Tetrahedron as a Primitive 17
Higher-Level Primitives 18
Volume Rendering Techniques 20
Ray Casting vs. Volume Slicing 20
Ray Casting Processing Order 21
Volumizer Processing Order 22
Advantage of Volume Slicing 23
Using Minimal Tessellations 23
Minimal Tessellation Advantages 23
Minimal Tessellation Disadvantages 23
Virtualizing Voxel Memory 25
Bricks 25
Brick Sets 25
Brick Set Collections 26
Brick Overlap 27
Clip Boxes 28
Brick Size Restriction 29
Polygonizing Shapes 30
Faces Clipped to Brick Boundaries 30
Displaying Volumetric Geometry with Other ToolKits 31

3. Programming Template 33
Audience for this Chapter 33
High-Level Overview of Procedure 35
Step 1: Inclusions and Declarations 36

Step 2: Define Appearance 36
Select Optimal Brick Parameters 37
Construct an Instance of a voBrickSetCollection 39
Allocate Storage for Bricks 40
Read Brick Data from Disk 40
Work with Brick Data 42
Scaling Voxel Values 42
Converting Brick Data 42
Transposing Brick Data 44
Optimize Voxel Data 44
Step 3: Define Geometry 45
Allocating Storage for Transient Geometry 47
Step 4: Drawing Volumes 50
Polygonizing Volumes 50
aTetraSet and aBrickSet 51
modelMatrix and projMatrix 51
samplingMode 52
samplingSpace 54
samplesNumber 55
samplingPeriod 56
State Sorting 56
Drawing Polygonized Volume 57
Step 5: Using Lookup Tables 59
voTextureLookup 59
Table Formats 59
Pre-Interpolation Lookup 60
Post-Interpolation Lookup 60
Choosing Between Pre- and Post-Interpolation 61
Lookup Example 61
Selecting Lookup Table Types and Formats 62

Step 6: Handling Errors 63
Setting Up Error Handlers 63
Setting Error Conditions 64
Debugging Information 64
Step 7: Clean Up 65

4. Sample OpenGL Volumizer Application 67
glwSimpleVolume.cxx 68
glwSimpleMain.cxx 83

5. Volumizer API at a Glance 93
Functional Categories 94
Class Hierarchy 94
Brief Descriptions of the Volumizer Classes 96

6. Volumetric Geometry 101
Data Structures 102
voVertexData 102
Preferred Order of Values in Records 103
Creating an Instance of voVertexData 104
voVertexData Methods 104
voIndices 104
voIndexedSet 105
voIndexedSetIterator 105
voIndexedFaceSet 106
Creating an Instance of voIndexedFaceSet 106
Populating Face Sets with Polygons 107
voIndexedFaceSet Methods 108
Deallocating Indexed Face Sets 108
voIndexedFaceSetIterator 108
voIndexedTetraSet 109
Creating a voIndexedTetraSet 109
Clipping Planes 111
Arbitrarily Shaped Volumes of Interest 111
Using Higher-Level Geometric Primitives 113

Higher-Level Geometric Primitives and Solids 113
Mixing Volumes and Surfaces 114
Rendering Opaque Geometry with Volumes 115
Rendering Translucent or Overlapping Geometry with Volumes 115
Transient Geometry Caching During polygonize 115
Rendering Multiple Volumes 116
Overlapping Volumes 117
Combining Overlapping Volumes 117
Texture Memory Overlap 117
Interpenetrating Polygons 118
Polygonizing Arbitrarily Oriented Cross Sections 118
Stack of Two-Dimensional Textures 119
Using Angled Slices 120
Multi-planar Reformatting Polygonization 121
Shading 122
Tangent Space Shading 123
Volume Roaming 124
Implementing a Volume of Interest 124
Picking Volumetric Objects 125
Auxiliary Methods 127

7. Volumetric Appearance 129
Texture Bricking 130
Data Types 130
Data Formats 131
Data Format Domains 132
Data Format Values 132
Optimal Formats 133
Converting Between Formats 134
Scaling Data 134
Interleaving Bricks 135
Advantages of Interleaving Bricks 135
Manually Interleaving Multiple Bricks 136
Creating Texture Objects 136
When Not to Use Texture Objects 136
Voxel Values 137
Tagged Voxels 137
Voxel Reference Frame 138
Generating Voxel Coordinates 140
Texture Matrices 141
Setting Brick Sizes 141
Setting Brick Sizes Automatically 142
Drawing Brick Outlines 142
Brick Sets 143
Brick Set Collections 143
Allocating Brick Data 144
Loading Brick Data 145
Forcing Download of Texture Data 145

Creating Custom Loaders 146
Loading Three-Dimensional Data 146
Loading Two-Dimensional Data 147
Unsupported File formats 147
File Format Utilities 147
Converting Voxel Data to the TIFF Format 147
Merging a List of Two-Dimensional Files Into One Three-Dimensional File 148
Test Volumes 148

8. Customized Volume Drawing 149
Customized Volume Drawing Procedure 150
clip() 152
General Clipping 152
Other Applications 153
Drawing Brick Set Collections 154
Example Use of findClosestAxisIndex() 155

A. Volume Rendering Examples 157
OpenGL Examples 157
voglBasic 160
voglCache 160
voglRaw 160
voglSpaceLeap 161
voglMorph 161
voglSphere 162
voglPick 162
voglShade 163
voglUnstructured 163
voglMirror 163
voglMPR1 and voglMPR2 164
Open Inventor Examples 164
IRIS Performer Example 166

Index 171

List of Figures

Figure 1-1 Volumetric Rendering 3
Figure 1-2 OpenGL Volumizer in Relation to Other Graphics APIs 8
Figure 2-1 Geometric Primitives 10
Figure 2-2 Abstract Data Set 10
Figure 2-3 Colored Cubes 11
Figure 2-4 Similarities Between Two- and Three-Dimensional Shapes 13
Figure 2-5 Volume Deformation 14
Figure 2-6 Jaw Modeled as a Separate Part 15
Figure 2-7 Per-part Properties. 16
Figure 2-8 Finely Tessellated Shape 16
Figure 2-9 Tetrahedral Decomposition of Cubic Geometry 17
Figure 2-10 Tessellation of Spherical Geometry 18
Figure 2-11 Volumetric Primitives 19
Figure 2-12 Processing Order 20
Figure 2-13 Volume Rendering as Texture Mapping 21
Figure 2-14 Ray Casting Processing order 21
Figure 2-15 Volume Slicing 22
Figure 2-16 Non Uniform Interpolation 24
Figure 2-17 Brick Set 26
Figure 2-18 Data Replication for Two-dimensional Texturing 26
Figure 2-19 Brick Overlap 27
Figure 2-20 Bricks Overlapping 28
Figure 2-21 Polygonal Slices of a Tetrahedron 30
Figure 2-22 Face Sets Spanning Two Bricks Stacked Vertically 31
Figure 3-1 Five Tetrahedra Defining a Cube 45
Figure 3-2 Indexed Vertices 46
Figure 3-3 voFaceSets 51

Figure 3-4 Sampling Surface 52
Figure 3-5 A Volume Polygonized Using VIEWPORT_ALIGNED and AXIS_ALIGNED Sampling 53
Figure 3-6 Sampling Spaces 54
Figure 3-7 Increasing Sampling Rate 55
Figure 5-1 Volumizer Class Hierarchy Divided By Function 95
Figure 6-1 Indexed Face Sets 106
Figure 6-2 Arbitrarily Shaped VOI 112
Figure 6-3 Geometric (Sunglasses) and Volumetric Objects Rendered Together 114
Figure 6-4 Overlapping Volumes 116
Figure 6-5 Merging Multiple Volumes 117
Figure 6-6 MPR with Two-Dimensional Texture Mapping 118
Figure 6-7 MPR Bands 120
Figure 6-8 Shading Off and On 122
Figure 6-9 Volume of Interest 124
Figure 7-1 Data Formats 131
Figure 7-2 Voxel Coordinates 138
Figure 8-1 Polygonization of a Single Tetrahedron. 150
Figure A-1 voglSpaceLeap: Space Leaping Example 161
Figure A-2 voglSphere: A Spherical Region of Interest 162
Figure A-3 voglPick: Voxel Picking Example 162
Figure A-4 voglMirror: A Muti-pass Reflection Algorithm on a Heterogeneous Scene 163
Figure A-5 Clipping Volumes to Arbitrary Surfaces 165
Figure A-6 Free-form Deformation of Volumes 166

About This Guide

The OpenGL Volumizer API is a library of C++ classes that facilitates the display and manipulation of volumetric shapes. This guide provides a developer's introduction to the API in two parts: Part One describes all the basic concepts in Volumizer and the most-commonly used classes in the API. A full-length, annotated example application shows the Volumizer concepts and API in context.

Part Two of the book presents advanced topics, including concepts, classes, and methods not presented in Part One.

Audience for This Guide

This book is intended for C++ developers of volumetric applications who understand the basic concepts of computer graphics programming.

Familiarity with OpenGL concepts and programmatic interfaces is strongly recommended.

How to Use This Guide

The first chapter introduces you to the central concepts of OpenGL Volumizer. These concepts are described in terms of the OpenGL Volumizer API in Chapter 2. Chapter 2 also presents a step-by-step guide for creating a OpenGL Volumizer application.

Subsequent chapters explore the OpenGL Volumizer API in greater detail in a task-oriented format.

What This Guide Contains

This book contains the following chapters:

Part One: Using the Volumizer API

Chapter 1, “Volumizer Advantages,” provides an overview of OpenGL Volumizer and lists its advantages.

Chapter 2, “Basic Concepts in Volumizer,” describes, at a high level, the important concepts you need to understand before developing an OpenGL Volumizer application.

Chapter 3, “Programming Template,” describes in terms of the OpenGL Volumizer API the concepts discussed in Chapter 1. This chapter also presents the basic steps you take to create your own OpenGL Volumizer application.

Chapter 4, “Sample OpenGL Volumizer Application,” presents annotated code for a sample OpenGL Volumizer application.

Chapter 5, “Volumizer API at a Glance,” provides a high-level overview of the entire API.

Part Two: Advanced Topics

Chapter 6, “Volumetric Geometry,” describes the OpenGL Volumizer classes that relate to creating volumetric geometry.

Chapter 7, “Volumetric Appearance,” describes the OpenGL Volumizer classes that relate to applying appearances to volumetric geometry.

Chapter 8, “Customized Volume Drawing,” describes how to create your own draw action for rendering volumes.

Appendix A, “Volume Rendering Examples,” present OpenGL and OpenInventor examples using the OpenGL Volumizer API.

A glossary and index follow the appendix.

Recommended Reference Materials

Silicon Graphics Publications

The following are found in IRIS InSight:

- *IRIS Performer Programming Guide* (SGI_Developer bookshelf)
- *MIPS Compiling and Performance Tuning Guide* (SGI_Developer bookshelf)
  For information on dynamically shared objects (DSOs).
- *OpenGL on Silicon Graphics Systems* (SGI_Developer bookshelf)

Third-Party Publications

- Farin, Gerald. *Curves and Surface for Computer Aided Geometric Design*. San Diego, CA.: Academic Press, Inc., 1988.
- D. Voorhies and J. Foran, "Reflection Vector Shading Hardware" in *Computer Graphics Proceedings, Annual Conference Series*, ACM, 1994.
- The *OpenGL WWW Center* at http://www.sgi.com/Technology/OpenGL.

The following are all produced by Addison-Wesley Publishing:

- Foley, J. D., A. VanDam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 1990.
- Gamma, E., R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995.
- Kilgard, M. J., *Programming OpenGL for the X Window System, 1996.* (Also known as "the Green book.")
- Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, 1992. Note Chapter 6, "Mapping Techniques: Texture and Environment Mapping."
- Wernecke, J., *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor,* 1994.
- Wernecke, J., *The Inventor Toolmaker,* 1994.

Conventions Used in This Guide

All class and function names, and other literals in the OpenGL Volumizer API have names that begin with prefix **vo**, followed by a string beginning with an upper case letter, for example, **voBrick**. Related utilities use **vout** prefix, for example **voutPerfMeter**.

These type conventions and symbols are used in this guide:

Bold C++ class names, C++ member functions, C++ data members, and function names.

Italics Filenames, manual/book titles, new terms, and variables.

`Fixed-width type`
Code.

`Bold fixed-width type`
Keyboard input keys.

ALL CAPS Environment variables, defined constants.

() (Bold Parentheses)
Follow function names. They surround function arguments if needed for the discussion or are empty if not needed in a particular context.

PART ONE

Volumizer API Basics

Part One of this book presents the most-commonly used OpenGL Volumizer concepts, classes, and methods This part presents the steps you take to create an OpenGL Volumizer application and presents an annotated example as an extension of the step-by-step programming template.

Chapter 1, “Volumizer Advantages,” provides an overview of OpenGL Volumizer and lists its advantages over other volume rendering APIs.

Chapter 2, “Basic Concepts in Volumizer,” describes, at a high level, the important concepts you need to understand before developing an OpenGL Volumizer application.

Chapter 3, “Programming Template,” describes in terms of the OpenGL Volumizer API the concepts discussed in Chapter 1. This chapter also presents the basic steps you take to create your own OpenGL Volumizer application.

Chapter 4, “Sample OpenGL Volumizer Application,” presents an annotated, sample OpenGL Volumizer application.

Chapter 5, “Volumizer API at a Glance,” provides a high-level overview of the entire OpenGL Volumizer API.

Chapter 1

Volumizer Advantages

The OpenGL Volumizer API is a library of C++ classes that facilitates the display and manipulation of volumetric data. Typical examples of volumetric shapes include:

- Clinical diagnostic images, for example, CT, MRI, and PET.
- Seismic data.
- Unstructured meshes that are common in CFD analysis.

The OpenGL Volumizer API facilitates rapid development of efficient, portable, and easily maintainable applications that have the ability to manipulate volumes.

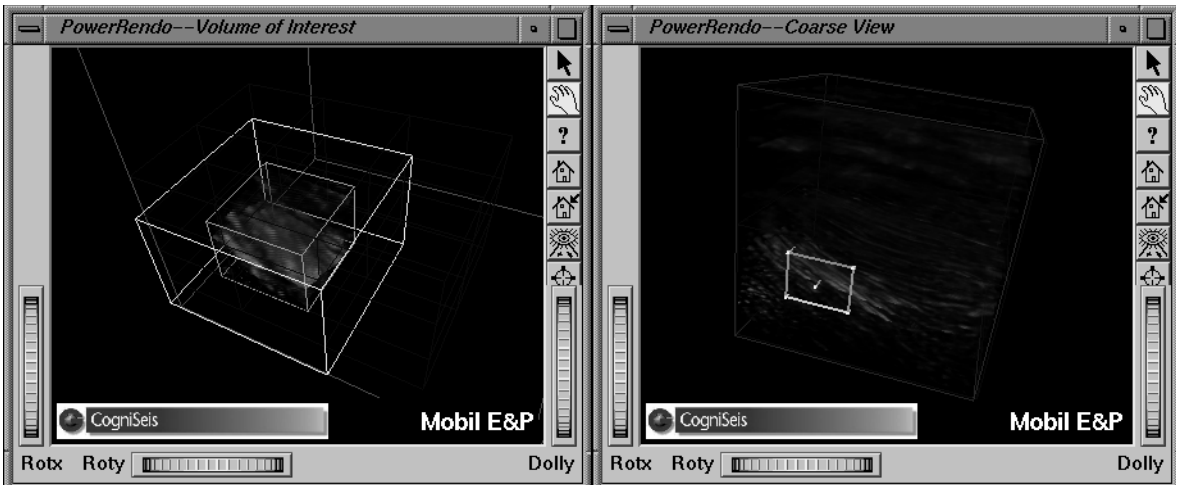

Figure 1-1 Volumetric Rendering

OpenGL Volumizer facilitates displaying and manipulating volumetric data by providing a level of abstraction that shelters the application programmer from the mundane task of dealing with platform-specific details thereby shortening prototyping and development cycles and allowing the programmer to focus on the application rather than the mechanics of the volume rendering process. At the same time, OpenGL

Volumizer allows the implementers of the API to create highly optimized code that can be tailored to specific platforms and updated as new hardware and software are introduced. Applications can thus take full advantage of the underlying hardware while maintaining a single code base spanning multiple platforms.

This chapter provides a high-level overview of OpenGL Volumizer and differentiates it from other volume rendering APIs, in the following sections:

- "Volumizer Advantages" on page 4
- "Where OpenGL Volumizer Fits In" on page 8

Volumizer Advantages

OpenGL Volumizer is specifically designed for volume visualization applications. It hides the details of low-level graphics languages and exposes only those functions that are necessary for viewing volumetric data.

OpenGL Volumizer provides an unparalleled set of sought-after features. The following sections describe some of them:

- "Faster Development Cycle" on page 5
- "High Performance" on page 5
- "Handles Large Data Sets" on page 5
- "Easy-to-Use API" on page 5
- "Low-Level Control" on page 6
- "Familiar Environment" on page 6
- "Cross-Platform Support" on page 6
- "Interoperability With Other Toolkits" on page 6
- "Multi-Pipe Capability" on page 7
- "Feature-Rich API" on page 7
- "Preparing for the Future" on page 7

Faster Development Cycle

Volumizer facilitates displaying and manipulating volumetric data by providing a level of abstraction that shelters the application programmer from the mundane task of dealing with platform-specific details. Volumizer thereby shortens prototyping and development cycles.

By handling the mechanics of volume rendering, application development time is radically shortened.

High Performance

The driving force behind Volumizer is optimal handling and display of volumetric models. Volumizer enables developers to create highly optimized code tailored to specific platforms that can be updated as new hardware and software is introduced.

Handles Large Data Sets

Volumizer's 64-bit support enables access and manipulation of data files larger than 4 GB. A typical seismic data set could be as large as 10 GB. OpenGL Volumizer hides the details of handling such large data sets.

Easy-to-Use API

Volumizer is the higher-level alternative to OpenGL that allows developers to focus more directly on the large picture rather than on the fine granularity characteristic of low-level graphics programming languages, such as OpenGL.

By handling the mechanics of volume rendering, Volumizer enables developers to focus on the unique qualities that differentiate their application.

Low-Level Control

Volumizer provides the ease of a higher-level graphics programming language. However, Volumizer supports full access to OpenGL for programmers needing low-level control in their applications. Applications can thereby take full advantage of the underlying hardware while maintaining a single code base spanning multiple platforms.

Familiar Environment

Volumizer operates in the familiar, immediate-mode graphics, which is characteristic of OpenGL based applications.

Cross-Platform Support

Developing graphics applications for different platforms and different operating systems is both time consuming and expensive. Now, with Volumizer, developers can create one application that runs on many platforms and operating systems without modification. While the current release of Volumizer is for IRIX only, support for other hardware and software platforms is being considered.

Interoperability With Other Toolkits

Volumizer API is an immediate mode toolkit built on top of OpenGL. Therefore, Volumizer can be used in the familiar framework of many existing, immediate mode applications.

Volumizer enables the integration of its volumetric shapes with the geometric shapes produced by other, higher-level, retained mode toolkits, such as:

- IRIS Performer
- Open Inventor
- OpenGL Optimizer

For example, you can render volumetrically defined clouds in a flight simulator (IRIS Performer), or a CAD model (OpenGL Optimizer) of a hip prosthesis in the anatomical context provided by a diagnostic CT scan of a pelvis. This intermixing is possible because Volumizer converts volumes into a set of polygons; each polygon is a slice of a volume and each slice can be rendered with polygons from the other toolkits.

OpenGL Volumizer does not explicitly provide support for any of the toolkits mentioned above. Instead, simple examples of how to create node “wrappers” in these toolkits are provided.

Multi-Pipe Capability

While Volumizer does not explicitly support multi-pipe operation, Volumizer can be used in conjunction with other software to provide this functionality. For example, Volumizer can operate in the frameworks of Optimizer, Performer and Multi Pipe Utility (MPU) all of which provide this type of functionality.

Feature-Rich API

Volumizer provides a robust set of features, including:

- Volume deformation
- Arbitrarily-shaped volumes of interest
- Volume modeling decoupled from appearance
- Roaming very large data sets
- A unified approach to regular grids and unstructured meshes
- Support for imbedded geometry
- Support for handling multiple volumes
- Voxel picking

Preparing for the Future

Volumizer provides a well-defined transition strategy to the Fahrenheit Scene Graph environment, which is the anticipated Microsoft-Silicon Graphics graphics API.

Where OpenGL Volumizer Fits In

The Volumizer API is a layer of functionality that, like other Silicon Graphics graphics APIs, sits on top of OpenGL, as shown in Figure 1-2.

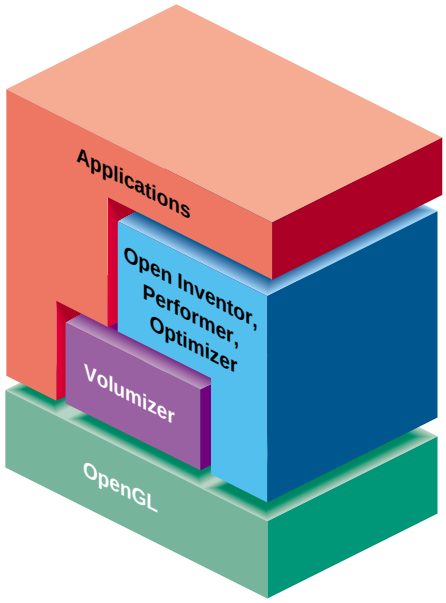

Figure 1-2 OpenGL Volumizer in Relation to Other Graphics APIs

Volumizer is positioned on top of OpenGL because Volumizer can:

- Be decomposed into more primitive actions already supported by OpenGL
- Provide a high degree of configurability, extensibility (low level control), and ease-of-use (high level of abstraction)

Chapter 2

Basic Concepts in Volumizer

This chapter describes the central concepts in Volumizer in the following sections:

- "Volumetric vs. Surface-Based Models" on page 9
- "Decoupling Geometry and Appearance" on page 12
- "Volume Rendering Techniques" on page 20
- "Using Minimal Tessellations" on page 23
- "Virtualizing Voxel Memory" on page 25
- "Polygonizing Shapes" on page 30

The next chapter describes the API calls associated with these concepts.

Volumetric vs. Surface-Based Models

In computer graphics, a three-dimensional object is any object that exists in three-dimensional space. Strictly speaking, however, the triangles and other surface elements used to represent such objects are two-dimensional primitives.

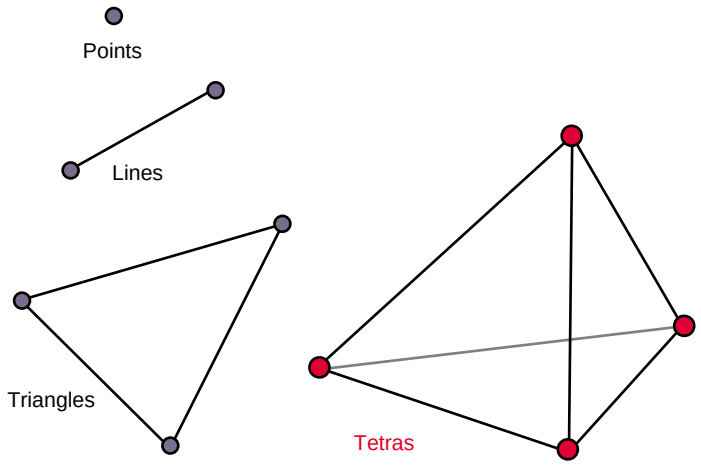

Figure 2-1 Geometric Primitives

Two-dimensional primitives suffice in many cases because most objects around us are adequately represented by their surface. Objects with interesting interiors, however, are abundant in everyday life. Clouds, smoke, and anatomy are all examples of volumetrically interesting objects.

Abstract volumetric objects, such as medical (CT, MRI, PET), geophysical, and computational data sets also contain interesting interior information that cannot easily be represented by surfaces, as shown in Figure 2-2.

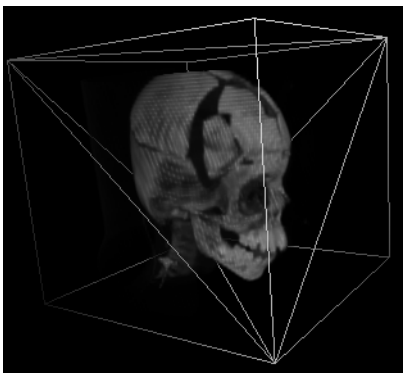

Figure 2-2 Abstract Data Set

Despite their abundance and importance, volumetric objects are either not handled at all or their treatment is substantially different from that of surface-based models. Handling heterogeneous scenes that contain volumes and surfaces, as a result, is very challenging and often done as a special case.

OpenGL Volumizer extends the concepts of surface-based models to include volumetric shapes. As a result, Volumizer arrives at a single, unified framework capable of handling both types of models equally well.

Volumetric Properties

A volumetric model is not hollow; it has some property—for example, color or opacity—that varies throughout the interior of the object. Consider a color cube represented using two- and three-dimensional primitives:

- Traditionally, a colored surface-only cube is rendered by drawing six polygonal faces and interpolating the colors across each face.
- A colored volumetric cube interpolates the colors across the entire space occupied by the cube.

These two cubes look identical from the outside. However, the differences become apparent as soon as we try to fly through such object, or make the surface semi-transparent.

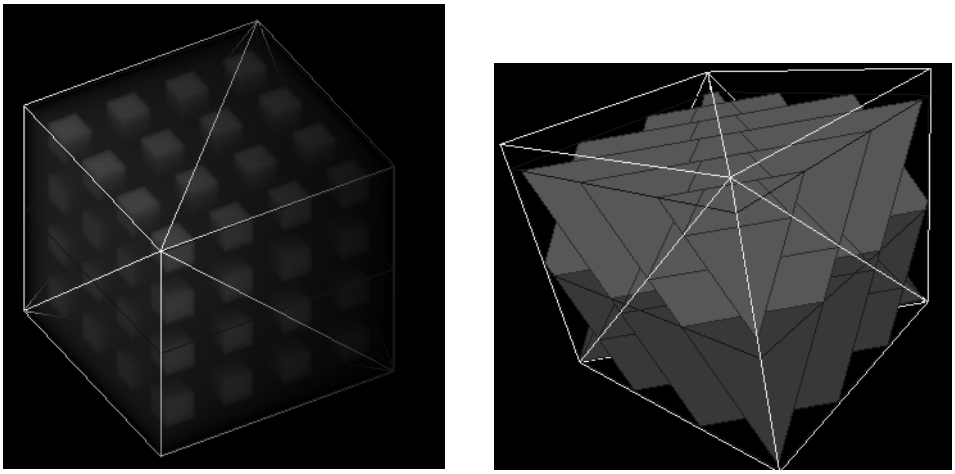

Figure 2-3 Colored Cubes

Decoupling Geometry and Appearance

In traditional three-dimensional graphics, graphical objects, called models or shapes in this document, are commonly described in terms of geometry and appearance. For example, a rectangular shape may be described by a quad mesh (geometry) with a two-dimensional texture (appearance) mapped onto it. In many cases there is a close relationship between appearance and geometry. For example, two-dimensional texture maps, which are rectangular arrays of values (pixels), are often mapped onto a single rectangular polygon of equal size.

Having an appearance match the size of a geometry, however, is a special, not a general case. For example, it is possible to move a circular geometry around inside a (larger) texture image like a magnifying glass, or to texture map an image onto a (smaller) sphere or a bicubic patch capable of squashing and twisting. In these examples, the shape's geometry and appearance are clearly decoupled.

It is the combination of geometry (for example, a sphere) and appearance (for example, voxels representing your data) that compose a volumetric shape. In the remainder of the book, the term "volume" describes the pairing of geometry and appearance.

Note: Although a geometry can be drawn without a texture, a texture cannot be drawn directly; it can only be used to modify the way a geometry is rendered. This is in stark contrast to conventional volume rendering techniques, which consider a voxel to be a drawable entity by itself.

In the simplest case, a volume's dimensions match that of its data cube. In general, however, the dimensions of a volume and its appearance differ. Conventional approaches to volume rendering do not explicitly separate volume and appearance; volume and appearance in those APIs tend to have the same dimensions. OpenGL Volumizer liberates your models from that restriction and thereby unifies the approaches for rendering geometric and volumetric shapes.

Figure 2-4 shows the similarity between polygonal and volumetric definitions of shapes.

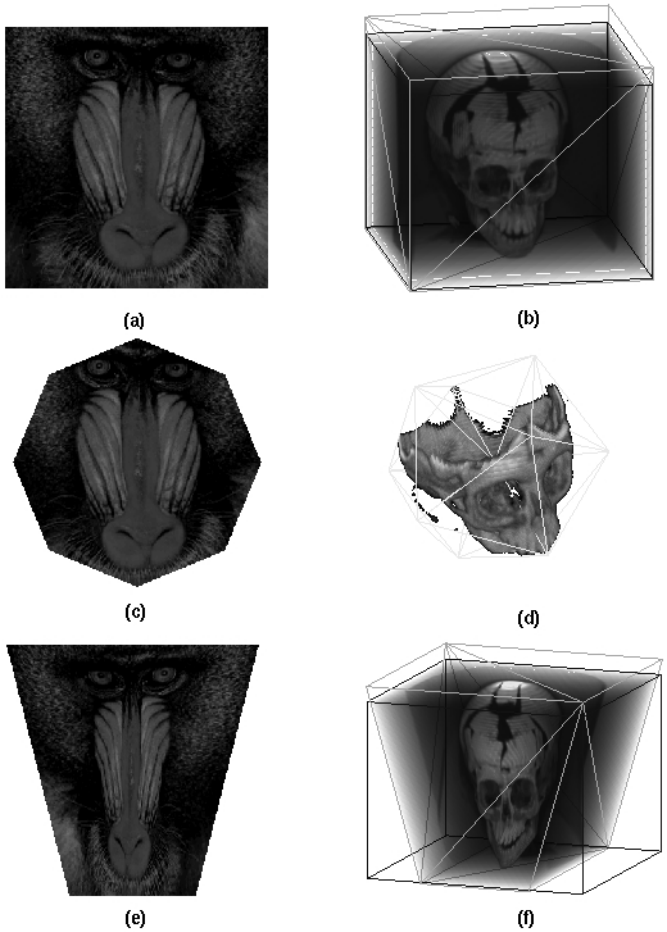

Figure 2-4 Similarities Between Two- and Three-Dimensional Shapes

- A two-dimensional array of pixels mapped onto a matching size rectangle, (a), is similar to a three-dimensional array of voxels mapped onto a matching size cube, (b).
- An octagonal “cookie-cutter” used to focus on a portion of the texture, (c), is similar to an icosahedral geometry used to focus on a portion of a volume, (d).
- A rectangular texture mapped onto an arbitrary quadrangle and the resulting distortion, (e), is similar to a regular volume mapped onto an irregular geometry and the resulting distortion, (f).

Advantage of Separating Geometry and Appearance

Because geometry and appearance are defined independently of one another in OpenGL Volumizer, you never need to write special code that changes the rendering engine for special features, such as:

- Arbitrarily-shaped volumes of interest (VOIs)

  Applications can choose to render only a portion of the volume, for example, a sub-cube or a spherical region containing interesting features. To do this you pass a different set of primitives (tetrahedra) as a parameter to the renderer. No changes to the renderer itself are needed.

- Volume deformation

  You can change a shape by changing its volume but without changing its voxels. This three-dimensional deformation is analogous to stretching a texture-mapped polygon. Figure 2-5 shows how a cube is transformed into a truncated pyramid.

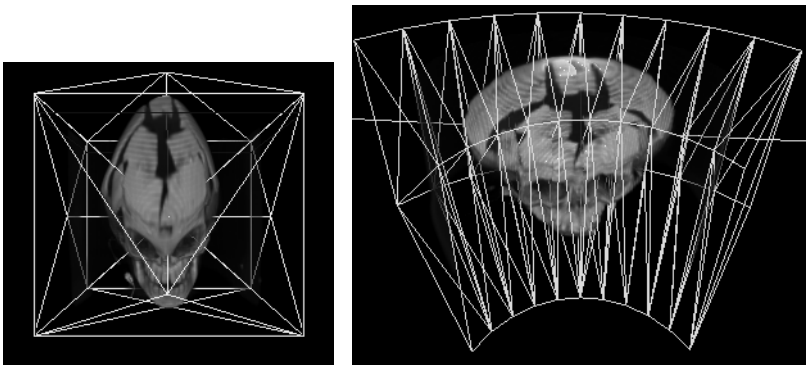

Figure 2-5 Volume Deformation

The left panel in Figure 2-5 shows a simple free form deformation application in which the vertices defining the volume's geometry are moved around affecting the shape of the model. The right panel illustrates a contrained deformation: the geometry is distorted radially. This feature can prove useful in applications that deal with ultrasonic and radar data, which need to be "dewarped."

- Hierarchical modeling

  Applications can create tessellations that define sub-parts within shapes. For example, rather than use a canonical five-tetrahedron tessellation of a volume, for example a skull, a separate set of tetrahedra can be specified to model the jaw, or a bone flap. These subsections can be manipulated as separate objects (with different material properties and transformations) to simulate maxillofacial or brain surgery.

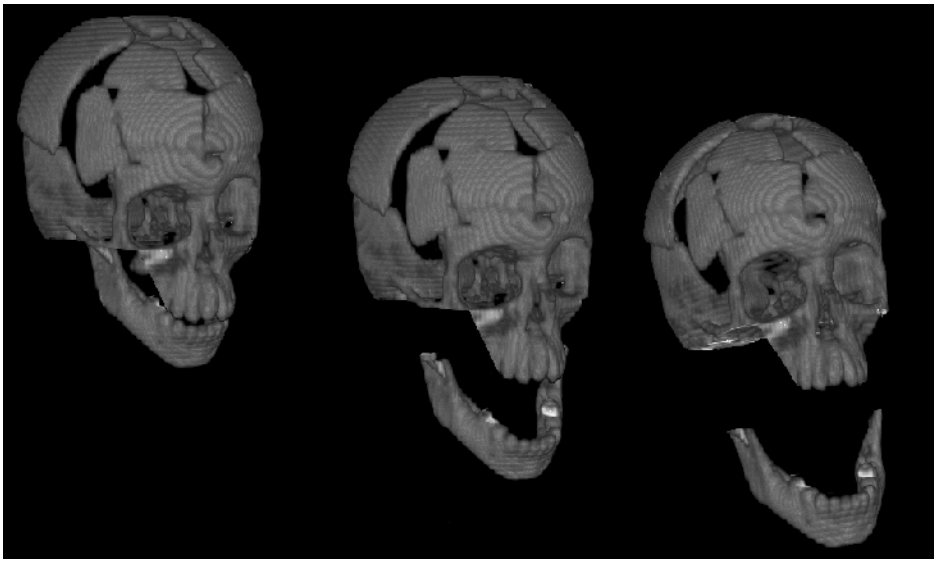

Figure 2-6 Jaw Modeled as a Separate Part

- Per-part appearance.

Applications can assign different properties (e.g., colors) to highlight or otherwise distinguish individual subparts of the model. This can be useful in labeling geological material in a seismic data interpretation application or for diagnostic data set segmentation.

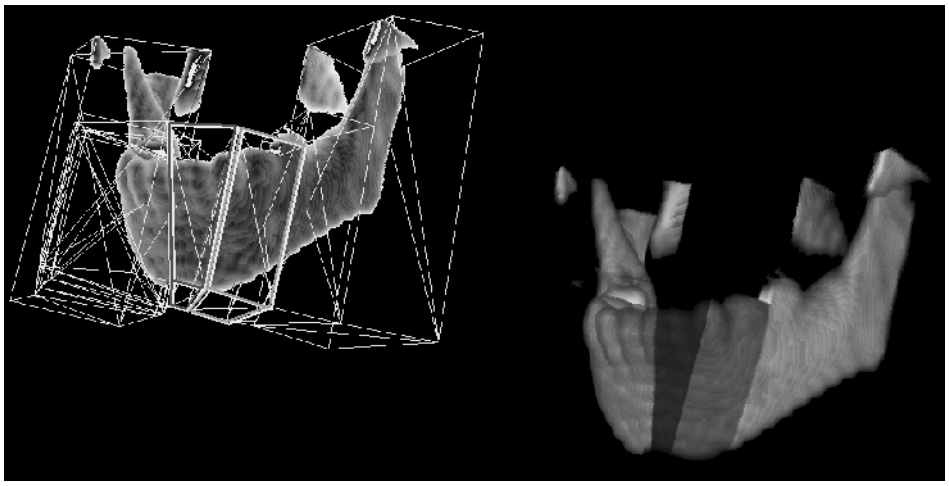

Figure 2-7 Per-part Properties.

- Space leaping techniques

  You can use a very fine tessellation to produce a large number of small cubes to skip areas of void in the scene (analogous to polygon assisted ray casting (PARC) and other space-leaping techniques) in a simple pre-processing step, as shown in Figure 2-8.

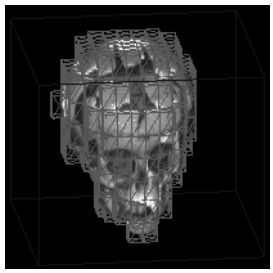

Figure 2-8 Finely Tessellated Shape

This technique, called space leaping, can sometimes produce a dramatic performance increase for sparse data sets by reducing pixel fill calculations.

Tetrahedron as a Primitive

The triangle is the simplest and most flexible primitive you can use to represent polygonal geometry; any surface can be decomposed into a collection of triangles to approximate the surface of a geometry. A circle, for example, can be approximated by thirty triangles all sharing a common vertex in the center of the circle.

Similarly, a tetrahedron is the simplest and most efficient primitive you can use to represent volumetric geometry. Any shape can be tessellated into tetrahedra; for example, any cube can be decomposed into as few as five tetrahedra, as shown in Figure 2-9.

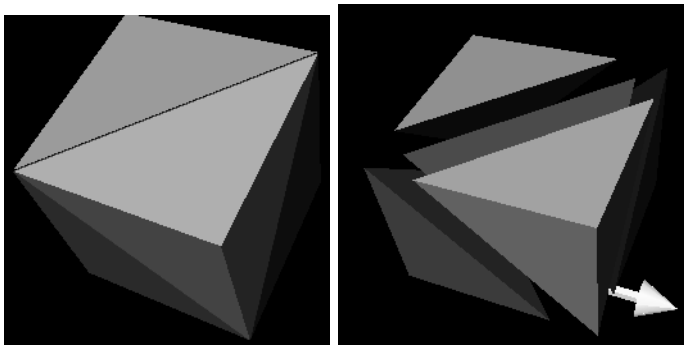

Figure 2-9 Tetrahedral Decomposition of Cubic Geometry

Or, for example, an icosahedron (a rough approximation of a sphere) can be tessellated into 20 tetrahedra by connecting the center of the icosahedron with triangles on the surface, as shown in Figure 2-10.

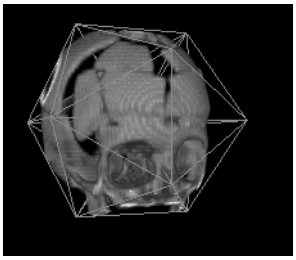

Figure 2-10 Tessellation of Spherical Geometry

Higher-Level Primitives

At times it is more convenient to specify higher-level geometric shapes, such as boxes or solid spheres, as primitives. You might use a higher-level primitive, for example, when using complex-shaped volumes of interest, for example, a logo of the company filled with smoke. Some of the polyhedral primitives are shown in Figure 2-11.

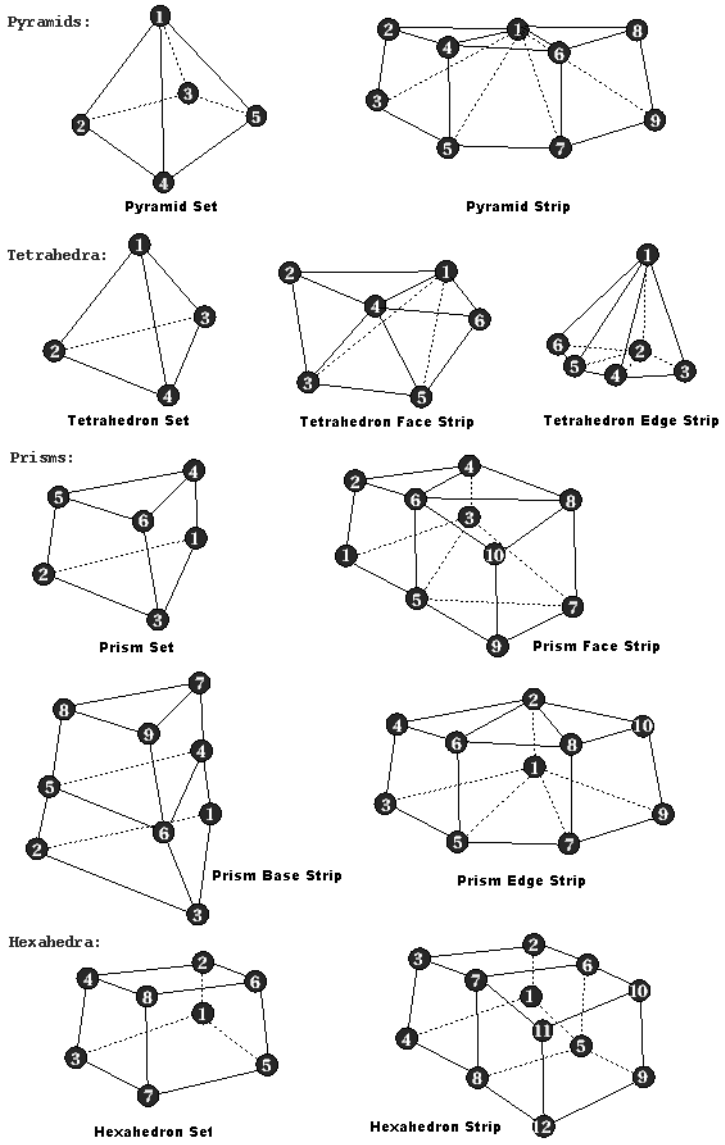

Figure 2-11 Volumetric Primitives

For more information about higher-level primitives, see "Using Higher-Level Geometric Primitives" on page 113.

Volume Rendering Techniques

There are many traditional ways to render a volume, including ray casting, splatting, and shear warp. OpenGL Volumizer uses a technique, similar to ray casting, called volume slicing, to leverage the texture-mapping hardware many workstations now have.

Ray Casting vs. Volume Slicing

Ray casting can be performed in ray-order or sampling-surface order using planes or spherical surfaces parallel to the line of sight, as shown in Figure 2-12.

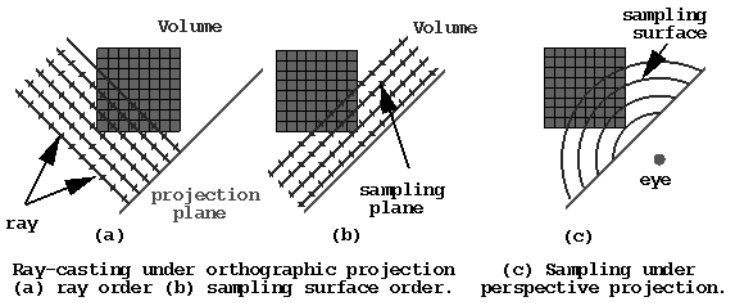

Figure 2-12 Processing Order

Volume slicing and ray casting are equivalent in the following ways:

- Ray casting under orthographic projection ((a) and (b) in Figure 2-12) is equivalent to taking a series of slices along planes parallel to the viewport and compositing them.
- Ray casting under perspective projection ((c) in Figure 2-12) is equivalent to sampling along a series of concentric spherical shells centered at the eye.

The end result is volume rendering according to texture mapping, as shown in Figure 2-13.

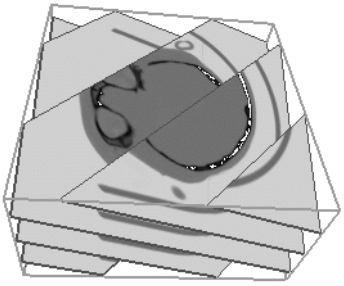

Figure 2-13 Volume Rendering as Texture Mapping

Ray Casting Processing Order

In ray casting, each point on a ray projected from the eye position through the volume is processed sequentially. In this technique:

1. The colors, opacity and shading of a volume are sampled, filtered, and accumulated at a point on a ray a specific distance from the eye position.
2. The distance is incremented along the ray, and the same colors, opacity and shading computations are repeated by the CPU, as shown in Figure 2-14.

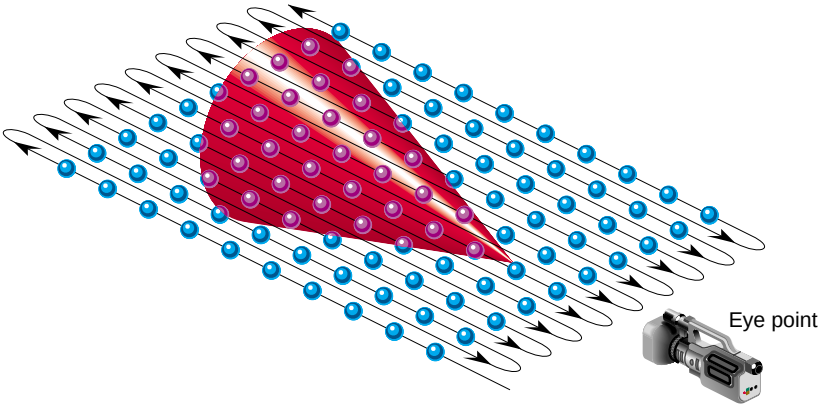

Figure 2-14 Ray Casting Processing order

3. When the traversal finally extends beyond the viewing frustum, the point of computation moves to the next ray near the eye point.

Conventionally, the main processing loop operates in ray order.

Volumizer Processing Order

In Volumizer, all points on a plane orthogonal to the line of sight are computed sequentially in the texture mapping hardware. In this technique:

1. The volume is sampled in surfaces orthogonal to the viewing direction, as shown in Figure 2-15.

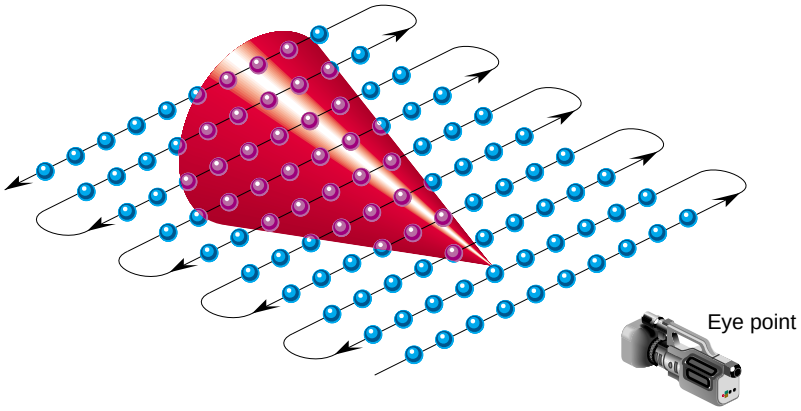

Figure 2-15 Volume Slicing

2. After the points along one plane on all the rays intersecting the volume are processed, the distance is incremented and the processing occurs again for all points on all the rays in the new plane.
3. Processing the points continues until the plane of points moves beyond the viewing frustum, at which point the processing terminates.

Advantage of Volume Slicing

The results of ray casting and volume slicing are identical. Orthographic projection is equivalent to volume slicing using sampling planes, and perspective ray casting is similar to using sampling along generalized (non-planar) surfaces in Volumizer.

There are, however, some important differences between the two techniques in processing the volumes:

- Volume slicing is faster than ray casting because computations are performed by the dedicated texture mapping hardware, whereas ray casting computations are performed by the CPU.
- Volume slicing reduces the volume to a series of texture-mapped, semi-transparent polygons. These polygons are in no way special and can be merged with any other polygonal data base handed to any graphics API (for example, OpenGL or Optimizer) for drawing.

Using Minimal Tessellations

In many common examples, volumes are hexahedral. The minimum tessellation of a hexahedron, as shown in Figure 2-9, is five tetrahedra. There are, however, advantages and disadvantages to using minimal tessellations.

Minimal Tessellation Advantages

When voxel coordinates and vertex coordinates coincide, you can use a single, large tetrahedron that fully encloses a voxel array to increase performance. For example, given a voxel array of $SIZE^3$, you would use a tetrahedron with vertices at (0, 0, 0), ($3 \times SIZE$, 0, 0), (0, 3*SIZE, 0), and (0, 0, $3 \times SIZE$). Such a single-tetrahedron tessellation can reduce polygonization calculations by a factor of five.

Minimal Tessellation Disadvantages

While compact, a minimal tessellation is not uniform and tends to introduce interpolation artifacts. For example, Figure 2-16 shows how colors are not uniformly interpolated across a face.

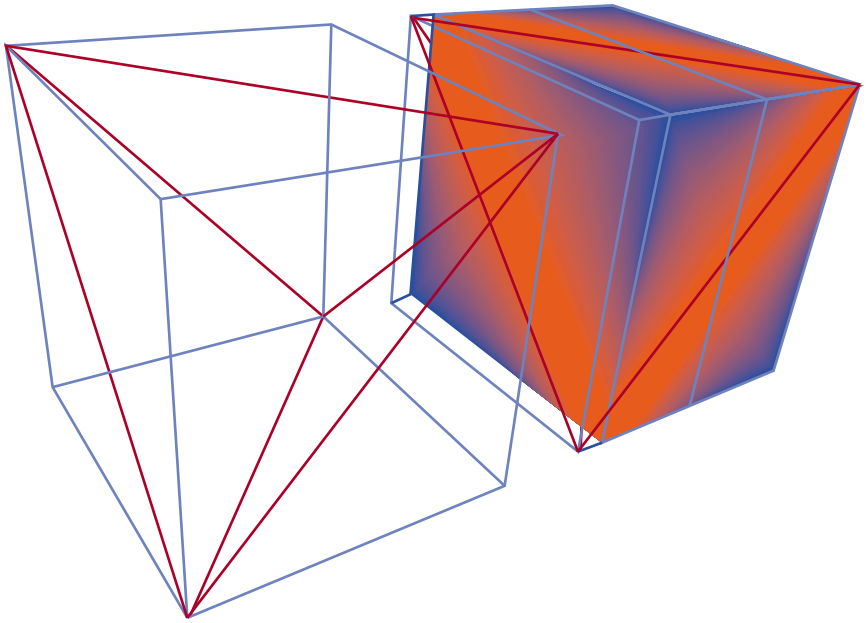

Figure 2-16 Non Uniform Interpolation

The vertices of the cube alternate between red (bright) and blue (dark). One would expect the faces of the cube to be smoothly interpolated between the respective vertices. However, the interpolation only occurs within a tetrahedron. Therefore, the resulting faces will have either a red (bright) or a blue (dark) diagonal band running along the edge of the tetrahedron that divides them. This artifact is analogous to creating T-junctions in polygonal tessellations.

Connecting multiple boxes through face adjacency leads to inconsistent (and highly noticeable) interpolation bands. For example, the wire frame box in Figure 2-16 has the same tessellation and vertex coloring as the solid one.

Yet, due to the asymmetry inherent in the tessellation, the adjacent faces have opposite diagonals as bases for their tessellations. Therefore, one of the two adjacent faces (solid box) is rendered with red (bright) running along one diagonal, while the other (wire frame box) has a similar band running along the opposite diagonal.

Worse yet, moving one of the vertices shared between two adjacent boxes results in "cracking."

These artifacts make the use and manipulation of complex volumetric primitives cumbersome.

All of the artifacts discussed can be minimized or avoided by splitting the cube into a larger number of more uniformly distributed tetrahedra. For example, it is possible to split a cube into 6 pyramids with the apex in the center of the cube using the faces of the cube as their bases. These pyramids can be further subdivided into four tetrahedra each for a total of 24 tetrahedron.

Virtualizing Voxel Memory

Some image data bases contain more voxel data than can be stored in a machine's texture mapping hardware. To take advantage of hardware acceleration, voxel data is broken up into subsections, called bricks.

Bricks

A *brick* is a subset of voxel data that can fit into a machine's texture mapping hardware. Bricks are regular hexahedra (boxes) with non-zero width, height, and depth. Displaying a volume requires paging the appropriate brick data into texture memory. Anticipating which bricks to page can speed up your application's performance.

Applications have control over individual bricks; for example, it is an application's responsibility to provide voxel data for each brick.

Bricks are three-dimensional objects but can also be used to represent two-dimensional textures by setting one of the dimensions to 1. This way stacks of 2D images can easily be handled.

Brick Sets

One or more adjacent bricks constitute a *brick set*. Volumetric appearance is defined as a collection of one or more brick sets. Figure 2-17 shows a brick set containing eight bricks, shown as eight cubes within the large cube.

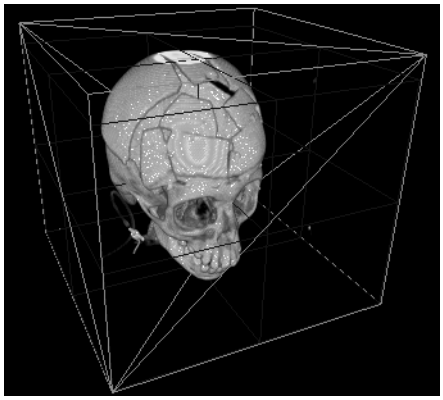

Figure 2-17 Brick Set

Brick Set Collections

Typically, a shape can be described by a single brick set. In certain situations, however, more than one brick set is required. For example, on machines that do not support three-dimensional texture mapping, three separate copies of the data set may have to be maintained, one for each major axis, as shown in Figure 2-18, to minimize sampling artifacts.

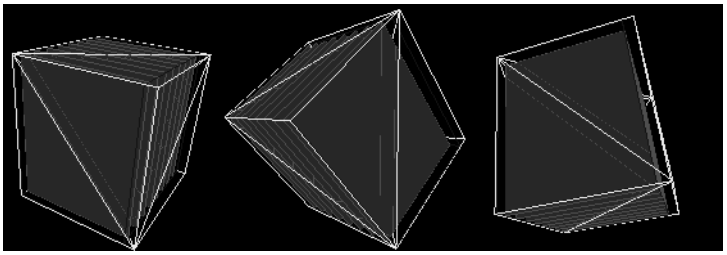

Figure 2-18 Data Replication for Two-dimensional Texturing

For more information about brick set collections, see “Brick Set Collections” on page 143.

Brick Overlap

Bricks typically have to overlap to prevent seams from appearing at the brick boundaries.

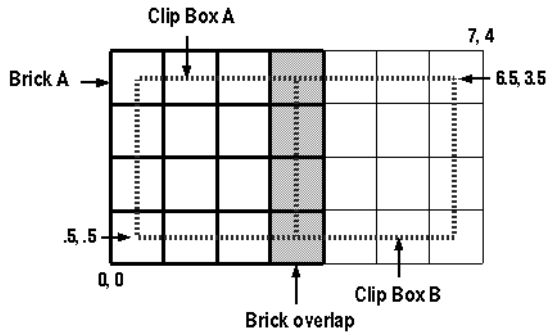

Figure 2-19 Brick Overlap

Figure 2-19 shows that bricks overlap but clip boxes, described in “Clip Boxes” on page 28, do not; they are adjacent. the difference in size between the brick and the clip box determines the brick overlap which helps prevent artifacts, such as seams, in the image.

The amount of overlap depends on the filtering scheme used. For example, if an application requested:

- Tri-linear interpolation, the bricks have to overlap by a layer of voxels one voxel thick in each direction
- Cubic interpolation, three layers are needed in each direction

Another factor that determines the width of brick overlap is gradient computation. When the gradient is computer, the x, y, and z dimensions for each voxel are graded over x + 1 to x - 1, y + 1 to y - 1, and z + 1 to z - 1. If any of these values fall outside the brick, the overlap might be greater.

Overlaps create storage overhead that is typically negligible, but may be substantial in certain situations. For example, requesting that a volume divided into bricks that are one voxel thick and insisting on 1 voxel overlap in all directions triples the storage overhead.

Clip Boxes

Clip boxes are used to clip geometry that spans several bricks in a way that guarantees that individual pieces join seamlessly. A clip box is centered where a brick is, but is smaller by the amount of overlap. Clip boxes of adjacent bricks are strictly adjacent; that is, there is no overlap or void between them.

Figure 2-20 illustrates the spatial relationships for two 4-by-4 bricks overlapping by one layer of voxels. The clip boxes are represented by the dotted lines.

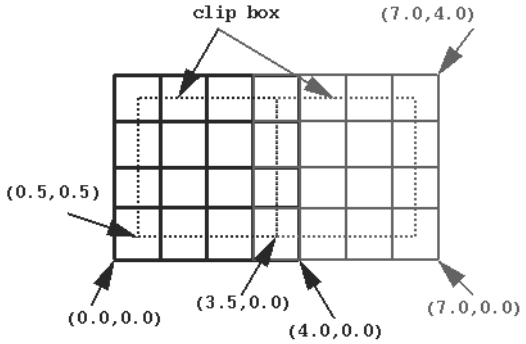

Figure 2-20 Bricks Overlapping

It is convenient to associate a clip box with a brick. A brick is characterized by its origin and its size. In Figure 2-20, the brick on the left has its origin at (0,0). The brick on the right has its origin at (3,0). The left and right clip box origins are (0.5, 0.5) and (3.5, 0.5), respectively.

Brick Size Restriction

Brick sizes are required to be a power of 2. This restriction allows implementations to take advantage of the underlying graphics API, OpenGL. This requirement does not mean, however, that the volume itself has any size restrictions.

If the volume dimensions do not divide evenly into brick dimensions, you can do one of the following:

- Ignore the voxels that fall outside of the brick set that evenly divides the volume.

  For example, a $256 \times 256 \times 190$ volume bricked into 128^3 bricks with one voxel overlap in each direction creates 8 bricks discarding a single layer of voxels in the X and Y directions, and 62 layers in the Z direction (Figure 2-12).

- Add an additional layer of bricks in each direction that are partially empty.

  In the previous example, that would mean creating 27×128^3 bricks.

- Try to make the brick as small as possible: determine the smallest power of 2 that fits the partial brick.

  In the previous example, that would result in creating 27 bricks, but the X and Y dimensions of the partial bricks would be 2, and the Z dimension would be 64. This means that bricks may be of sizes other than the one requested.

  For more information about carrying out these options using the API, see “Work with Brick Data” on page 42.

Polygonizing Shapes

To render a shape, Volumizer slices a volume along a set of parallel surfaces; this slicing process is called *polygonization*. The result is a set of polygons, called *faces*. Textures are associated with each of these polygons, as shown in Figure 2-21.

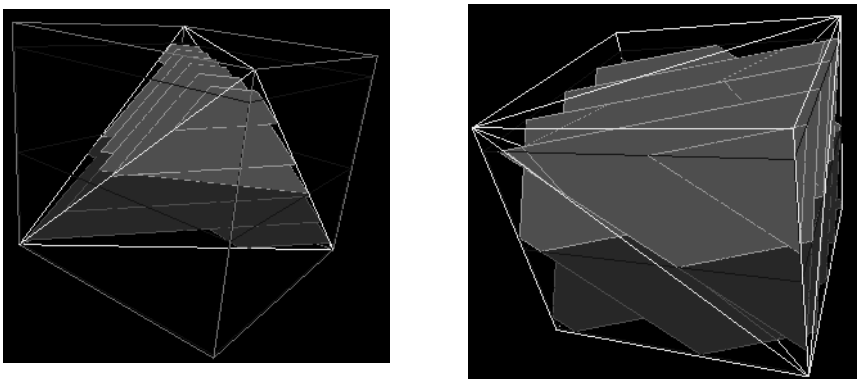

Figure 2-21 Polygonal Slices of a Tetrahedron

The figure on the left in Figure 2-21 shows the polygonization of a tetrahedron. The figure on the right shows the polygonization of five tetrahedra that define a cube.

The orientation of the surfaces is configurable; the simplest case is when the surfaces are orthogonal to the line of sight of the viewer. The surfaces, however, can be aligned arbitrarily.

The slices in these figures are planar; they can, however, be of any shape. In particular, you might choose to shape them spherically to render equidistant points from the viewpoint.

Faces Clipped to Brick Boundaries

Polygonization clips polygons to brick and volume boundaries to facilitate texture paging. Figure 2-22 shows a single tetrahedron spanning two bricks, which are stacked vertically.

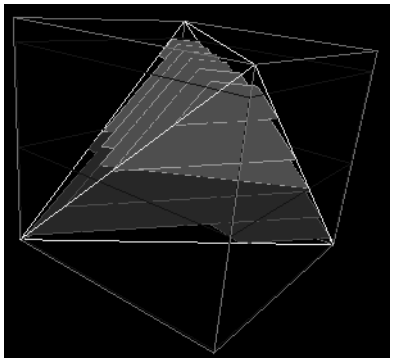

Figure 2-22 Face Sets Spanning Two Bricks Stacked Vertically

Figure 2-22 shows a separate set of polygons for each brick. **polygonization()** depth sorts these polygons and hands them to the application for drawing.

For more information about polygonization, see “Polygonizing Volumes” on page 50.

Displaying Volumetric Geometry with Other ToolKits

After Volumizer slices shapes into a set of parallel polygons, the slices can be rendered by many rendering toolkits, such as IRIS Performer and OpenGL Optimizer. In this way, it is easy to intermix volumes and surface-only geometries in the same display.

For more information about intermixing volumetric and polygonal shapes, see “Mixing Volumes and Surfaces” on page 114.

Chapter 3

Programming Template

This chapter presents a programming template for Volumizer applications. Rigorously following this template is not required when creating your own Volumizer application. The template does, however, provide a programming framework in which to understand the Volumizer API.

This chapter explains the most commonly-used elements of the Volumizer API in the context of creating an application. Chapter 2, “Basic Concepts in Volumizer,” presented the concepts behind the API described in this chapter.

The structure of this chapter is parallel to Chapter 4, “Sample OpenGL Volumizer Application,” in which a full code example is explained. By understanding the API calls presented in this chapter, you can better understand the complete code example in the next chapter.

Audience for this Chapter

Developers working with the Volumizer API can be divided into two groups:

- Those who want to use the supplied Volumizer application with their own image database.
- Those who want to create their own Volumizer application.

Developers in the first group need only to learn about file input and output. OpenGL Volumizer provides several loaders in sample applications. If none of these loaders work, then you need to create your own loader. For help doing that, refer to “Creating Custom Loaders” on page 146.

Note: The examples are provided as illustrative and often do not provide full functionality.

The other parts of this chapter present topics relevant to the second group of developers. The basic steps you take to create your own OpenGL Volumizer application are discussed in the following sections:

- "Step 1: Inclusions and Declarations" on page 36
- "Step 2: Define Appearance" on page 36
  - Selecting Optimal Brick Parameters
  - Defining Appearance (**voBrickSetCollection**)
  - Allocating Storage for Voxel Data
  - Reading Voxel Data from Disk
  - Working with the Brick Data
  - Optimizing Appearances
- "Step 3: Define Geometry" on page 45Defining Geometry (**voIndexedTetraSet**)
  - Allocating Storage for Transient Polygons
- "Step 4: Drawing Volumes" on page 50
  - Polygonizing a Volume
  - State Sorting
  - Loading Voxel Data into Texture Cache
  - Drawing Polygonized Volume
- "Step 5: Using Lookup Tables" on page 59 (optional)
- "Step 6: Handling Errors" on page 63
- "Step 7: Clean Up" on page 65

High-Level Overview of Procedure

This section provides a conceptual overview of the programming template. The sections following this one describe in much more detail how to accomplish each of the programming tasks discussed in this section.

For more information about any of the concepts in this section, see Chapter 2, "Basic Concepts in Volumizer."

When creating a Volumizer application, you perform the following tasks:

1. Define a volumetric shape:
    - Define appearance.
    - Define geometry.
2. Render the shape:
    - Sample the shape along a set of surfaces, often planes parallel to the viewport.

      In this process, called polygonization, the surfaces are clipped to the boundaries of the unbounded volume's geometry; as a result, the clipped planes become polygons.

      The polygons are then clipped to brick boundaries to take advantage of hardware acceleration. A brick is the amount of texture that can be cached into a machine's texture memory.
    - Depth sort the set of polygons from back to front.
    - Map textures onto the polygons from the voxel information and composite the polygons together in the frame buffer to produce the final image.

The polygonization process converts the volume shape into a set of semi-transparent textured polygons that can be handed back to applications. These polygons are in no way special (other than being semi-transparent) and can subsequently be mixed with other polygons in a scene.

The following sections explain how these concepts are implemented by the Volumizer API.

Step 1: Inclusions and Declarations

The first step in creating a Volumizer application is to include all necessary OpenGL, Motif (or other Graphical User Interface API), and C++ header files and to declare global variables. Most applications need to include the following header files to assure that all Volumizer-specific declarations are present:

```
#include <vo/GeometryActions.h>
#include <vo/AppearanceActions.h>
```

These header files include a number of other Volumizer headers. The other header files are listed in “glwSimpleVolume.cxx” on page 68.

Step 2: Define Appearance

To define appearance, use the following procedure:

1. “Select Optimal Brick Parameters” on page 37
2. “Construct an Instance of a voBrickSetCollection” on page 39
3. “Allocate Storage for Bricks” on page 40
4. “Read Brick Data from Disk” on page 40
5. “Work with Brick Data” on page 42
6. “Optimize Voxel Data” on page 44

Select Optimal Brick Parameters

Different hardware platforms provide varying functionality and feature sets making it difficult for application developers to write portable, efficient and maintainable code. To insulate the API proper from machine dependencies, several helper routines are provided that localize such dependencies, such as **voAppearanceActions::getBestParameters()**. Its purpose is to suggest optimal values for several machine dependent variables:

```
xBrickSize = xVolumeSize;
yBrickSize = yVolumeSize;
zBrickSize = zVolumeSize;
```

```
voAppearanceActions::getBestParameters(
    interpolationType, renderingMode, dataType, // In
    diskDataFormat,                             // In
    internalFormat, externalFormat,             // Out
    xBrickSize, yBrickSize, zBrickSize);        // In/Out
```

interpolationType can have the following values:

```
enum voInterpolationType {
    DEFAULT,
    _2D,
    _3D
};
```

_2D and _3D select two-dimensional and three-dimensional textures as internal voxel representation, respectively. Setting this parameter to **DEFAULT** selects _3D on machines that support hardware three-dimensional texture mapping, _2D otherwise.

renderingMode can have the following values:

```
enum voRenderingMode{
    DEFAULT,
    MONOCHROME,
    COLOR
};
```

The value depends on the requested rendering mode: gray scale (MONOCHROME) or RGBA.

dataType and *diskDataFormat* are based on the numerical representation of the voxel data used by the application.

voDataType describes the data type of the voxel data. The values of the enumerant are of type **voDataTypeScope** and can be one of the following values:

```
enum voDataType {
    DEFAULT,
    UNSIGNED_BYTE,
    BYTE,
    UNSIGNED_BYTE_3_3_2_EXT,
    UNSIGNED_SHORT,
    SHORT,
    UNSIGNED_SHORT_4_4_4_4_EXT,
    UNSIGNED_SHORT_5_5_5_1_EXT,
    UNSIGNED_INT,
    INT,
    UNSIGNED_INT_8_8_8_8_EXT,
    UNSIGNED_INT_10_10_10_2_EXT,
    FLOAT
};
```

The dimensions, *xBrickSize*, *yBrickSize*, *zBrickSize,* specify the optimal brick size; If possible, the brick size will match the volume size.

Based on the above set of input parameters, the **getBestParameters()** method determines the optimal *internalFormat*, *externalFormat* and brick sizes for a given platform. For example, given the disk format of INTENSITY, a rendering mode of MONOCHROME and brick sizes of [256, 256, 256], and an SGI OCTANE with 4 MB of texture memory, the method returns LUMINANCE_ALPHA for *externalFormat*, LUMINANCE8_ALPHA8_EXT for *internalFormat*, and [128, 128, 64] for brick sizes. The same call, with identical input parameters returns LUMINANCE, INTENSITY8_EXT, and [256, 256,256], respectively, on an SGI Onyx2 IR with 64 MB of texture memory.

voInternalFormatType describes the format used internally by the texture subsystem to maintain voxel data. It can be one of the following values:

```
enum voInternalFormatType{
    DEFAULT,
    INTENSITY8_EXT,
    LUMINANCE8_EXT,
    LUMINANCE8_ALPHA8_EXT,
    RGBA8_EXT,
    RGB8_EXT,
    RGBA4_EXT,
    DUAL_LUMINANCE_ALPHA8_SGIS,
    DUAL_INTENSITY8_SGIS,
    QUAD_LUMINANCE8_SGIS
};
```

voExternalFormatType is the format that the application uses to pass the voxel data to the texture subsytem. The values of the enum are of type **voExternalFormatTypeScope** and can be one of the following values:

```
enum voExternalFormatType {
    DEFAULT,
    INTENSITY,
    LUMINANCE_ALPHA,
    LUMINANCE,
    RGBA,
    ABGR_EXT
};
```

getBestParameters() does not create anything, but merely suggests suitable parameters to use with subsequent calls. The application can overwrite the suggested values with the understanding that some changes may affect performance or the validity of subsequent operations.

For example, overwriting the internal format results in an additional format conversions every time the volume is downloaded. Or, for example, increasing the total size of a brick is likely to cause failure during the first download because of memory overrun. It is okay, however, to make bricks smaller.

Construct an Instance of a voBrickSetCollection

Once the optimal brick parameters are selected, they can be used to construct an instance of a **voBrickSetCollection** given the voxel array dimensions, brick sizes, voxel formats, and interpolation type to encapsulate the appearance, as follows:

```
voBrickSetCollection (
    int xVolumeSize, int yVolumeSize, int zVolumeSize,
    int xBrickSize, int yBrickSize, int zBrickSize,
    voPartialBrickType _fractionalBricks,
    int _overlap,
    voInternalFormatType _internalFormat,
    voExternalFormatType _externalFormat,
    voDataType _dataType,
    voInterpolationType _interpolationType);
```

A brick set collection is a collection of brick sets. The brick set collection contains the voxel data and all information about the data types, formats, and memory layout of the brick sets in the collection.

Note: This constructor does not allocate memory for voxel data.

For more information about **voBrickSetCollection**, see "Brick Set Collections" on page 143. For a discussion of advanced topics in appearance, see Chapter 7, "Volumetric Appearance."

Allocate Storage for Bricks

Each brick holds an array of voxel data. To allocate voxel memory for all of the brick sets in the brick set collection, interate over the bricks using the following nested loop:

```
// iterate over all brick sets in the brick set collection
for (voBrickSet * brickSet; brickSet = collectionIter();) {
    // iterate over all bricks within the brick set
    voBrickSetIterator brickSetIter(brickSet);
    for (voBrick * brick; brick = brickSetIter();)
    voAppearanceActions::dataAlloc(brick);
    }
aVolume->setCurrentBrickSet(voPlaneOrientationScope::XY);
```

Applications that prefer to manage their own data can set the data pointer of the brick directly to point to the memory area that contains the voxel data by calling the *setDataPtr()* member method. In either case, the application is responsible for memory management. In particular, the voxel memory has to be deallocated with a call to **voAppearanceActions::dataFree()** or some application-specific way once the application is done using it.

Read Brick Data from Disk

Applications read voxel data from the disk brick by brick. To read a brick from a disk into host memory (not texture memory, which is covered in "Read Brick Data from Disk" on page 40), you can use one of the supplied IFL loaders, for example:

```
extern int myReadBrickIfl(
    char *fileName, void *data,
    int xBrickOrigin, int yBrickOrigin, int zBrickOrigin,
    int xBrickSize, int yBrickSize, int zBrickSize,
    int xVolumeSize, int yVolumeSize, int zVolumeSize);
```

The above routine reads a set of voxels that fall within a subvolume, determined by the arguments, from any file supported by the Image Format Library (IFL), for example, a three-dimensional TIFF file. Another loader provided with the API reads a brick of voxels from a stack of "raw" two-dimensional images:

```
int myReadBrickRaw(char **fileNames, void *data,
    int xBrickOrigin, int yBrickOrigin, int zBrickOrigin,
    int xBrickSize, int yBrickSize, int zBrickSize,
    int xVolumeSize, int yVolumeSize, int zVolumeSize,
    int headerLength, int bytesPerVoxel)
```

For more information about stacks of two-dimensional images, see "Stack of Two-Dimensional Textures" on page 119.

A "raw" two-dimensional image is a voxel stream in row-major order possibly proceeded with a fixed-size header (the content of which is ignored).

OpenGL Volumizer provides several loaders. However, if you find that your data set cannot be read by any supplied Volumizer sample application, you need to make your own loader. "Creating Custom Loaders" on page 146 provides code for a loader that you can modify to handle your data.

All bricks of a brick set can be read in using the following construct:

```
voBrickSetIterator brickSetIter(aVolume->getCurrentBrickSet());
for (voBrick * brick; brick = brickSetIter();) {

    int xBrickOrigin, yBrickOrigin, zBrickOrigin;
    int xBrickSize, yBrickSize, zBrickSize;

    void *vdata = brick->getDataPtr();

    // get brick sizes -- they may different than those requested
    brick->getBrickSizes(xBrickOrigin, yBrickOrigin, zBrickOrigin,
    xBrickSize, yBrickSize, zBrickSize);
    // read the data; OK to use brick data area as a temp buffer
    myReadBrickIfl(dataFileName, vdata,
    xBrickOrigin, yBrickOrigin, zBrickOrigin,
    xBrickSize, yBrickSize, zBrickSize,
    xVolumeSize, yVolumeSize, zVolumeSize);
}
```

Multiple copies of the brick sets with different memory layouts can be read this way, or **voAppearanceActions::volumeMakeTransposed()** can be used to create the remaining copies.

Work with Brick Data

Once you read the brick data from disk into host memory, you might need to:

- Scale it, page 42
- Convert it, page 42
- Replicate it, page 44
- Transpose it, page 44
- Optimize it, page 44

Scaling Voxel Values

To scale the voxel values within a brick from an application-specific dynamic range to a standard range (such as <0,255> for unsigned char, <0,65535> for unsigned short, and <0.0,1.0> for floats), you use the following **voAppearanceActions** method on each brick:

```
static int dataScaleRange( voTexture3D* aBrick, float& loValue,
    float& hiValue)
```

loValue and *hiValue* are the minimum and maximum values, respectively, for the original range. For example, while working with 12-bit unsigned short data, the application should use values of 0.0 and 4095.0, respectively.

Converting Brick Data

In certain situations the disk data format does not match the external format, for example, when the data format is INTENSITY on disk and the external (that is, application-side) format is LUMINANCE_ALPHA. In this situation, each voxel within the brick needs to be duplicated so the sequence *I1I2I3I4...* becomes *I1I1I2I2I3I3I4I4....* The following **voAppearanceActions** method converts the brick data to the format requested in the constructor:

```
static int dataConvert(voTexture3D* aTexture3D, void* data,
    voExternalFormatType diskFormat);
```

voTexture3D and **voExternalFormatType** are defined in “Step 2: Define Appearance” on page 36 .

Converting data may require additional buffer space, such as in the case where the converted voxel data occupies twice as much memory area as the original voxels. Applications can choose to use an additional buffer to facilitate the conversion, as follows:

```
// select next brick
voBrick *brick = ...

// IO buffer
unsigned char data[BIG_ENOUGH];
// allocate storage for a brick
(void)voAppearanceActions::dataAlloc(brick);
// read the data into buffer
myReadBrickIfl(fileName, data, minX, minY, minZ, xBrickSize,
     yBrickSize, zBrickSize);
// replicate to the desired externalFormat
voAppearanceActions::dataConvert(brick,data,
    UNSIGNED_BYTE);
```

dataConvert(), however, can perform most conversions using the brick data area for intermediate storage. In this approach, voxel data is read directly into the data area of the brick, and is converted in place, for example:

```
// allocate storage for each brick
unsigned char *data = voAppearanceActions::dataAlloc(brick);
// read the data; OK to use brick data area as a temporary buffer
myReadBrickIfl(fileName, data, minX, minY, minZ, xBrickSize,
     yBrickSize, zBrickSize);
// replicate to the desired externalFormat
voAppearanceActions::dataConvert(brick,data,_DATA_TYPE_UNSIGNED_BYTE);
```

Using the brick data area for intermediate storage is preferable because, in some cases, it may minimize the amount of data copying.

Transposing Brick Data

In some instances, the API needs to maintain multiple copies of the same data set with different memory layouts, for example, to avoid sampling artifacts while using two-dimensional texturing.The API enables you to create the copies using the following method:

```
static int volumeMakeTransposed(voBrickSetCollection* aVolume);
```

This action is a no-op for three-dimensional volumes using tri-linear interpolation that were defined with **voInterpolationTypeScope::_3D.** However, you should always include this call during initialization to assure platform independence.

Optimize Voxel Data

Voxel data can be optimized for better performance in the following ways:

- Interleave bricks
- Create texture objects

To enable appearance optimization, set the last argument in **voAppearanceActions::volumeOptimize()** to one or more of the following values logically ORd together:

- INTERLEAVE, to interleave bricks
- TEXTURE_OBJECTS, to create texture objects
- BEST, to interleave bricks and create texture objects

For example, for full optimization, use a statement similar to the following:

```
voAppearanceActions::volumeOptimize(myBrickSetCollection,
    BEST_PERFORMANCE );
```

For more information about interleaving bricks, see “Interleaving Bricks” on page 135.

For more information about texture objects, see “Creating Texture Objects” on page 136.

Step 3: Define Geometry

To define a volume, an application needs to define both its appearance and its geometry. Volumetric geometry is defined by a set of tetrahedra. For example, a cube can be minimally represented as a set of five tetrahedra as shown inFigure 3-1.

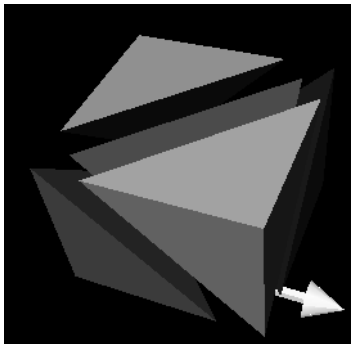

Figure 3-1 Five Tetrahedra Defining a Cube

A volume is generally tessellated into a collection of tetrahedra. Volumizer provides a **voIndexedTetraSet** object type to represent such collections. The vertex coordinates of the tetrahedra are indexed because the same set of coordinates can be shared by multiple tetrahedra. Rather than record the same point for multiple tetrahedra, index pointers for each of the tetrahedra point to one set of vertex coordinates, as shown in Figure 3-2.

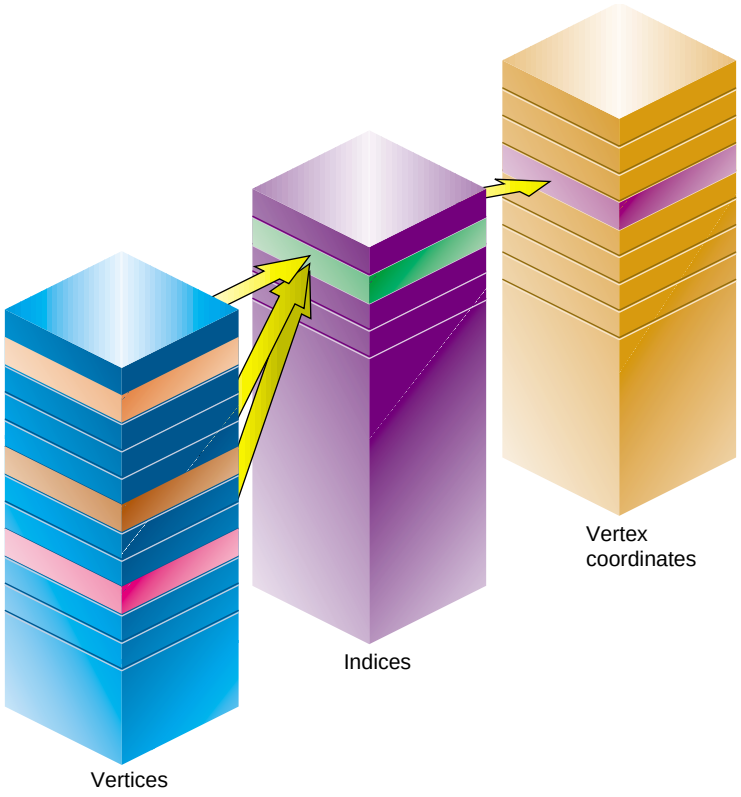

Figure 3-2 Indexed Vertices

For example, consider a solid sphere represented by 100 tetrahedra, all of which share a vertex in the center of the sphere. Instead of having 100 sets of the same coordinates, one index from each tetrahedra can point at one vertex coordinate substantially reducing memory requirements for storing vertex coordinates.

Each group of four indices associated with **voIndexedTetraSet** form a single tetrahedron. For example, in the following array, each row represents a tetrahedron; there are five tetrahedra in the array in total defined by 20 indices:

```
static int cubeIndices[] =
    {
        0, 2, 5, 7,
        3, 2, 0, 7,
        1, 2, 5, 0,
        2, 7, 6, 5,
        5, 4, 0, 7,
    };
```

These indices point into the vertex data array. The array of vertex coordinates might look like the following:

```
float cubeVertices[] = {

    0,            0,            0,
xSize,            0,            0,
xSize,        ySize,            0,
    0,        ySize,            0,
    0,            0,        zSize,
xSize,            0,        zSize,
xSize,        ySize,        zSize,
    0,        ySize,        zSize
};
```

One can create an instance of volumetric geometry describing a minimal tessellation of a cube using the following call:

```
voIndexedTetraSet *aTetraSet = new
      voIndexedTetraSet(cubeVertices, 8, 3, cubeIndices, 20);
```

Allocating Storage for Transient Geometry

To display a volume, the tetrahedra that make up the volume are polygonized into a set of polygons, called faces, by slicing them with a family of sampling surfaces, for example, planar surfaces parallel to the viewport. Each face is a textured slice of the volume clipped to brick and volume boundaries. To prepare for the results of polygonization, you need to create a set of faces equal to the number of bricks times the number of faces in each brick.

When a volume is polygonized, it is sliced into faces. Each face is a textured slice of the volume clipped to brick and volume boundaries. To handle the faces, use **voVertexData** and **voIndexedFaceSet**.

voVertexData is an array of records that holds per-vertex information for a list of vertices, defined as follows:

```
voVertexData(int _countMax, int _valuesPerVertex, float* data=NULL)
```

Each record describes a set of properties of a vertex, such as vertex coordinates, colors, normals, texture coordinates, and optional user-defined data.

_countMax is the number of vertices.

_valuesPerVertex indicates how many floats describe a vertex.

The number of properties per vertex is application dependent. The information may include vertex coordinates, colors, textures, normals, and user-defined per-vertex scalar or vector properties. All the properties are resampled and clipped during the polygonization process. Vertex coordinates (three floats) are required; optional data include colors, texture coordinates, and arbitrary user-supplied data. By convention, the order of per-vertex data is:

1. User
2. Texture coordinates
3. Colors
4. Vertex coordinates

The texture coordinates do not have semantics associated with them directly. By convention, however, the texture coordinates are specified in voxels; for example, in a 256^3 volume the last vertex has coordinates [255, 255, 255].

The allocated storage is guaranteed to be contiguous in memory. Optionally, the application can pass a pre-allocated array of floating point values to be used by reference, that is, no copy of data is made. Keep in mind, however, that **voVertexData**'s destructor deletes the data storage only if **voVertexData**'s constructor allocated it. Therefore, if an application passes a pre-allocated array to a constructor, it is the application's responsibility to delete this storage.

For more information about **voVertexData**, see "voVertexData" on page 102.

Each polygon resulting from polygonization is called a face. Polygonization creates a set of parallel faces, which are generally coplanar polygons. **voIndexedFaceSet** specifies a collection of indexed polygons, defined as follows:

```
voIndexedFaceSet (int _countMaxV, int _valuesPerVertex,
    int _countMaxI);
voIndexedFaceSet (voVertexData* _vertexData, int _countMaxI)
```

_countMaxV is the number of vertices in the face set.

_valuesPerVertex indicates how many floats describe a vertex.

_countMaxI is the number of indices in the face set.

_vertexData is a pointer to a data structure that holds per-vertex information for a list of vertices.

Each polygon in the face set is described by a group of indices. The first index within a group specifies the number of the vertices in the polygon; the following indices point to individual vertex records.

For more information about **voIndexedFaceSet**, see "voIndexedFaceSet" on page 106.

To iterate through a **voIndexedFaceSet**, use **voIndexedFaceSetIterator**, as described in "voIndexedSetIterator" on page 105.

Step 4: Drawing Volumes

A volumetric shape is drawn in three phases.

1. The geometry representing the shape is polygonized by sampling it with a family of surfaces.
2. The resulting polygons are sorted by state and depth.
3. The polygons are composited from back-to-front (or front-to-back) into the frame buffer.

The following sections describe these steps in greater detail.

Polygonizing Volumes

To render a volumetric shape, the set of tetrahedra representing the volume's geometry needs to be polygonized. This is accomplished by "slicing" the volume along a family of sampling surfaces. In the simplest case, the sampling surfaces form a set of planes parallel to the viewport. Each such plane is intersected with each tetrahedron resulting in a single polygon (a triangle or a a quadrangle). This polygon needs to be clipped to the brick boundaries. The number of polygons produced is equal to or less than the number of bricks. Each of the resulting polygons is at most a 10-gon (clipping an n-gon to a box produces at most an (n+6)-gon). This whole procedure is accomplished with help of **polygonize()**.

```
int polygonize(
    voIndexedTetraSet  *aTetraSet,
    voBrickSet *aBrickSet,
    GLdouble modelMatrix[16],
    GLdouble projMatrix[16],
    voSamplingMode samplingMode,
    voSamplingSpace samplingSpace,
    int &samplesNumber,
    float samplingPeriod[3],
    voIndexedFaceSet ***aIndexedFaceSet);
```

polygonize() returns a set of polygons. There is one polygon set for each sampling surface (i.e., depth) and per each brick. Therefore, there are samplesNumber * bricksNumber polygons that get generated. An instance of **voIndexedFaceSet** stores the polygons resulting from the polygonization process:

```
voIndexedfaceSet *faces[brickNumber][sampleNumber]
```

voGeometryActions::polygonize() performs the following tasks:

1. Slices the tetrasets into sets of planes parallel to the viewport, as shown in Figure 3-3.

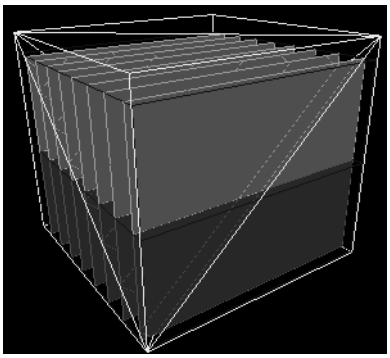

Figure 3-3 voFaceSets

2. Clips each plane of the tetraset to brick boundaries.

   The **voFaceSets** in Figure 3-3 are shown as belonging to two bricks stacked vertically. Faces are colored differently depending on which brick they belong to.

The following sections discuss each of the arguments of **polygonize()**.

aTetraSet and aBrickSet

aTetraSet describes the set of tetras to be polygonized. *aBrickSet* provides the coordinates of the bricks for clipping.

modelMatrix and projMatrix

modelMatrix and *projMatrix* describe the current position and orientation of the volume and the viewing parameters. They help determine the orientation of the sampling surfaces. They can be obtained from the following OpenGL state:

```
GLdouble modelMatrix[16];
GLdouble projMatrix[16];
glGetDoublev(GL_MODELVIEW_MATRIX, modelMatrix);
glGetDoublev(GL_PROJECTION_MATRIX, projMatrix);
```

samplingMode

samplingMode determines what type of sampling surface family to use. The following are valid values:

```
enum voSamplingMode{
    DEFAULT,
    VIEWPORT_ALIGNED,
    AXIS_ALIGNED,
    SPHERICAL,
};
```

DEFAULT allows the API to pick a family that is optimized for performance.

Figure 3-4 shows the graphical implication of these values.

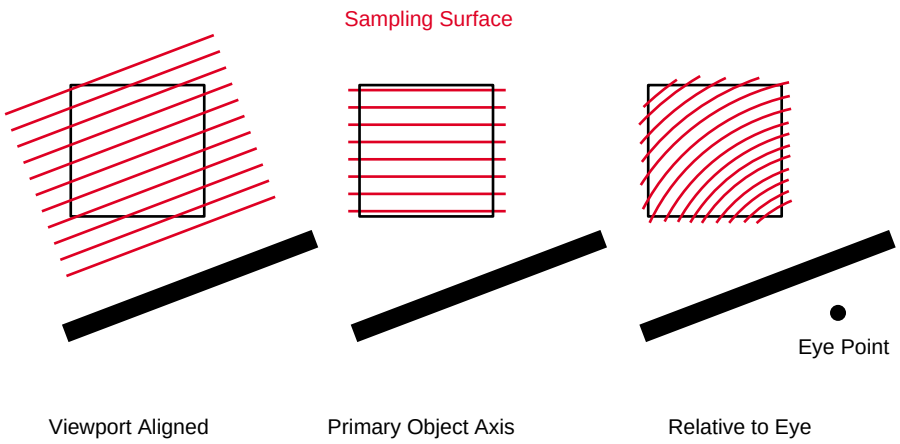

Figure 3-4 Sampling Surface

VIEWPORT_ALIGNED selects planes parallel to the viewport. This is the optimal sampling strategy in terms of rendering quality on systems which support three-dimensional texture mapping (it requires properly sampled and filtered oblique slices through the volume). On systems that do not support three-dimensional texture mapping this mode may not be supported or may be slow.

AXIS_ALIGNED samples volumes along planes that are aligned with the primary axes (X,Y,Z) of the voxel cube. Three copies of the volume (each sliced in one of the primary directions) may be required. This type of sampling allows machines that do not support three-dimensional texture mapping to render volumes. In this mode, the axis that is most parallel to the line of sight is selected.

It is possible to request this mode on a machine that does support three-dimensional texture mapping to improve performance: the polygonize action does not have to be invoked nearly as often as for in AXIS_ALIGNED mode and the texture data is accessed in a more predictable, cache-friendly fashion. There is no quality degradation in this case, as one can still request tri-linear interpolation (providing that three-dimensional textures are perspectively corrected).

SPHERICAL (not currently supported) allows for sampling of the data set with a family of concentric spherical shells centered at the camera position. This simulates more accurately ray casting in wide-angle perspective camera, for example, during fly-throughs. This mode is not currently implemented in Volumizer. Applications may choose to implement their own if need arises.

In Figure 3-5, a volume is viewed from one of its corners. The volume is polygonized with sampling planes parallel to the viewport (left) and the primary axes (right). The volume consists of two bricks coded in green and red.

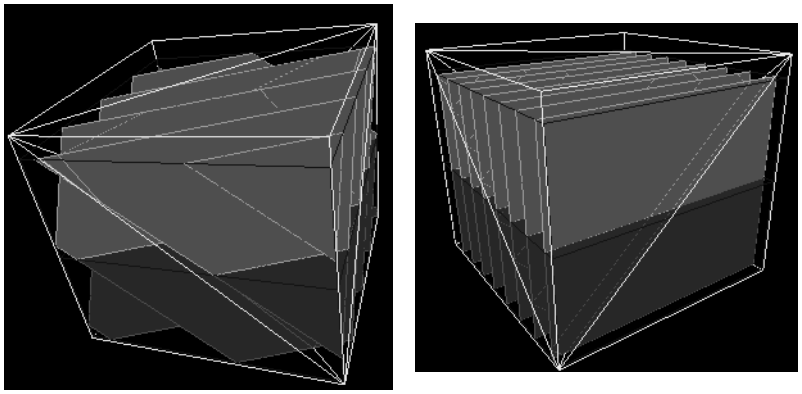

Figure 3-5 A Volume Polygonized Using VIEWPORT_ALIGNED and AXIS_ALIGNED Sampling

samplingSpace

During the polygonization process *samplesNumber* (e.g., 256) different sampling surfaces will be used to sample the volume. By default, they are evenly distributed over the whole sampling space. *samplingSpace* determines what "sampling space" is and how to compute the bounds (e.g., the locations of the first and last surface). Here are the allowable values:

```
enum voSamplingSpace {
    DEFAULT,
    VIEW_VOLUME,
    OBJECT,
    PRIMITIVE,
};
```

Figure 3-6 shows the graphical implication of these values.

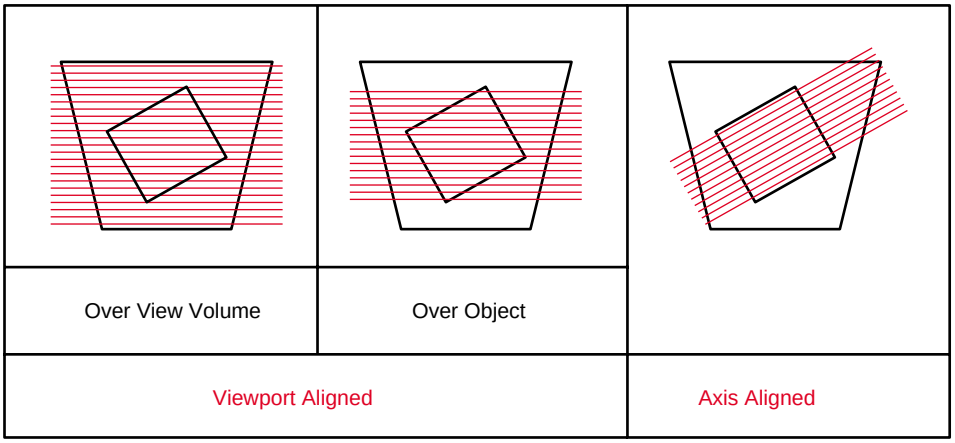

Figure 3-6 Sampling Spaces

VIEW_VOLUME divides the view frustum into *samplesNumber* evenly distributed samples. For example, in the VIEWPORT_ALIGNED sampling mode, the first plane coincides with the ZFar clipping plane, and the last with ZNear. The disadvantage of this technique is that typically some of the sampling surfaces never intersect the object being sampled. Similarly, translating the object though the frustum results in subtle sampling artifacts. On the other hand, it produces a uniform, static sampling grid which makes merging multiple volumes and overlapping transparent geometry somewhat easier.

OBJECT divides the bounding box of the volume, rather than the whole viewing frustum, into *samplesNumber* of evenly distributed samples. Therefore, this option adjusts to the position and orientation of the volume to assure optimal sampling rate.

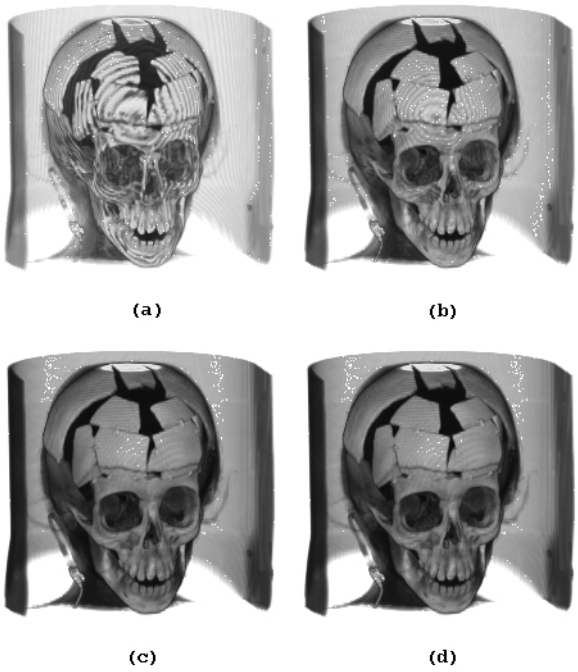

Figure 3-7 Increasing Sampling Rate

samplesNumber

samplesNumber has overloaded meaning. If *samplingPeriod* is NULL, *samplesNumber* determines how many sampling surfaces to use during the polygonization process. If *samplingPeriod* is not NULL, *samplesNumber* determines the upper bound of the number of sampling surfaces. Regardless of the value of *samplingPeriod*, no more than *samplesNumber* of sampling planes are generated. This functionality can guard against buffer overflows.

samplesNumber should be set in accordance with the Nyquist sampling limit. The higher the sampling rate, the higher the quality (to a point), but the slower the performance.

samplingPeriod

If *samplingPeriod* is NULL, it is ignored. Otherwise, it is used to change the number of sampling surfaces depending on the position, orientation of the surfaces, and viewing parameters.

For example, requesting *samplingPeriod* of (1.0, 1.0, 1.0) under **voSamplingSpaceScope::OBJECT** sampling results in *SIZE* samples when the volume is head on, and sqrt(3) × *SIZE* when a view of a cube *SIZE* on the size along the major diagonal is requested. Requesting a *samplingPeriod* of 0.5 assures that the Nyquist criterion is satisfied in that direction.

The number of samples computed is always clamped to the *samplesNumber* parameter to avoid unexpected overflows. In either case, *samplesNumber* is modified to reflect the actual number of slicing planes used.

State Sorting

Once the volume is polygonized it needs to drawn. Because volumes are generally semi-transparent, all processing must take place in depth-sorted order. In the case of a multi-brick volume, the bricks are processed one at a time:

1. Each brick is selected.
2. The selected brick is loaded into the texture cache.
3. The polygonization faces that fall within the brick are drawn.
4. The next brick is selected for processing.

To assure proper results, the bricks need to be sorted from back-to-front (or front-to-back). This is done to minimize the graphics API state changes. Use the following method for state sorting:

```
voSortAction aSortAction(
   aVolume->getCurrentBrickSet(), modelMatrix, projMatrix);
```

The bricks can then be accessed in depth-sorted order, as follows:

```
voBrickSetCollection *aVolume;
int brickSortedNo = aSortAction[brickNo];
voBrick *aBrick aVolume->getCurrentBrickSet()->getBrick(brickSortedNo);
```

This code returns brick number, *brickNo,* in depth-sorted order.

Drawing Polygonized Volume

Once the volume is polygonized, and its bricks are sorted by depth, the application needs to draw the resulting polygons for each brick in order. Before that happens, you need to establish a graphics state by doing the following:

1. Enable texture mapping.
2. Enable the OpenGL texgen feature, if the application did not specify the per-vertex texture coordinates explicitly.
3. Set up a suitable blending function.

To enable texture mapping, the applications can use **voAppearanceActions::textureEnable()**, where the argument, *interpolationType,* is one of the following:

```
enum voInterpolationType{
    DEFAULT = -1,
    _2D,
    _3D
};
```

If the application did not explicitly specify per-vertex texture coordinates, a one-to-one mapping from vertex coordinates to voxel coordinates is assumed. In this case, it is convenient to use texgen functionality of OpenGL to compute the texture coordinates from the vertex coordinates. To enable this feature use

```
if (!hasTextureComponent(interleavedArrayFormat))
   voAppearanceActions::texgenEnable().
```

To define the blending function, use OpenGL directly before any drawing happens:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```

Once the graphics state was set up all the transient geometry generated by the **polygonize()** can be drawn:

```
// Iterate over all bricks.
    for (brickNo = 0; brickNo < BrickCount; brickNo++) {
        int brickSortedNo = aSortAction[brickNo];
        voBrick *aBrick =
            aVolume->getCurrentBrickSet()->getBrick(brickSortedNo);

        // Update texgen equation for the current brick.
        if (!hasTextureComponent(interleavedArrayFormat))
          voAppearanceActions::texgenSetEquation(aBrick);

        // load the texture from host to texture memory
        voAppearanceActions::textureBind(aBrick);

        // iterate over all sampling planes
        for (int binNo = 0; binNo < samplesNumber ; binNo++) {
            voGeometryActions::draw(
                aPolygonSetArray[brickSortedNo][binNo],
                interleavedArrayFormat);
        }
    }
```

Step 5: Using Lookup Tables

Lookup table values control the opacity and colors of volumes. For example, you might choose to vary the opacity of skin such that when it becomes transparent, you can see through the skin to muscular or skeletal systems. Or you might choose to color parts of a volume differently, for example, you might enable the user to color a tumor differently from its surroundings.

voTextureLookup

voLookupTable is an abstract class that is used as a base class for both pre- and post-interpolation lookups. **voLookupTable()** constructs a lookup table in which the external and internal values of the table are defined in the argument of the method, as follows:

```
voLookupTable (GLenum _internalFormat, int _width,
    GLenum _externalFormat, GLenum _dataType, void* _data=NULL);
```

The arguments are defined in “Step 2: Define Appearance” on page 36.

Table Formats

The internal table format can contain either of the following values:

- Luminance and alpha: LUMINANCE8_ALPHA8_EXT
- RGBA: RGBA8_EXT

Lookup tables can be enabled and disabled using **voTextureLookup::enable()** or **voTextureLookup::disable()**, respectively.

Pre-Interpolation Lookup

voTextureLookupPre() performs a pre-interpolation lookup, which occurs in the image pipeline as the texture is loaded into the texture memory. Typically this class implements functionality provided in hardware with GL_COLOR_TABLE_SGI or any other tables along the imaging pipeline. The constructor is defined as follows:

```
voLookupTablePre(GLenum _internalFormat, int _width,
    GLenum _externalFormat, GLenum _dataType, void* _data);
```

The arguments are defined in “Step 2: Define Appearance” on page 36.

For changes caused by the lookup table to take effect, pre-interpolation lookups require reloading of the entire volume, even if it is already in texture memory. For this reason, pre-interpolation lookups are generally slower than post-interpolation lookups.

Pre-interpolations lookups are used to replace default functionality.

Post-Interpolation Lookup

voTextureLookupPost() performs a post-interpolation lookup, which occurs during the rasterization phase when the image is on its way from the texture cache to the frame buffer. Typically this class implements functionality provided in hardware with GL_TEXTURE_COLOR_TABLE_SGI. The constructor is defined as follows:

```
voLookupTablePost(GLenum _internalFormat, int _width,
    GLenum _externalFormat, GLenum _dataType, void* _data);
```

The arguments are defined in “Step 2: Define Appearance” on page 36.

Volumes do not need to be reloaded for post-interpolation changes to take effect. For this reason, post-interpolation lookups are generally faster than pre-interpolation lookups.

Post -interpolations lookups are used to alter default functionality.

Choosing Between Pre- and Post-Interpolation

The results of pre- and post-interpolation lookups are usually different. Make sure to choose the one appropriate to your intent.

For example, in post-interpolation lookup, two values, 100 and 200, are linearly interpolated and the resulting values (spanning the range 100-200) are looked up. Alternatively, in pre-interpolation lookup, the two values, 100 and 200, are looked up first, and the looked-up values are interpolated. If the values in a lookup table are linear, the results of pre- and post-interpolation lookups are identical; with a non-linear table, however, the results of pre- and post-interpolation differ.

Some platforms may impose restrictions on the use of post-interpolation lookups. For example, it is not possible to do a full RGBA post-interpolation lookup on Indigo2 Impact where the table length is limited to 256 entries.

Lookup Example

Example 3-1 shows an example of a table lookup use.

Example 3-1 Lookup Example

```
float lookupEntries[2*256];
voTextureLookupPost *aLookupTable = new voTextureLookupPost(
    LUMINANCE8_ALPHA8_EXT,
    256,
    LUMINANCE_ALPHA,
    FLOAT,
    lookupEntries);

for(int j1=0;j1<256;j1++)
    lookupEntries[2*j1] = lookupEntries[2*j1+1] = 0.5*(float)j1/256.0;

aLookupTable->load();
aLookupTable->enable();
```

Note: If you use a linear ramp for table lookups, you could replace the **for()** loop with the following method:

```
aLookupTable->set(voTextureLookup::LINEAR,195,50);
```

This construct also allows certain types of optimizations to be performed.

Selecting Lookup Table Types and Formats

The Volumizer API provides a routine, **getBestParameters()**, that selects a suitable table type, and internal and external table formats given the following parameters:

- Volumetric shape
- Required rendering mode (MONOCHROME or COLOR)
- Requested table length

Example 3-2 provides a sample implementation of **getBestParameters()**.

Example 3-2 Selecting Lookup Table Types and Formats

```
voLookupTableType lookupType = POST_INTERPOLATION;  // preferred type
voInternalFormatType internalFormat;
voExternalFormatType externalFormat;

voLookupTable::getBestParameters(aVolume, renderingMode, lookupLength,
                     lookupType, internalFormat, externalFormat);

    if (lookupType == PRE_INTERPOLATION)
        aLookupTable = new voLookupTablePre(
            internalFormat,
            lookupLength,
            externalFormat,
            _DATA_TYPE_FLOAT,
            lookupEntries);
    else
        aLookupTable = new voLookupTablePost(
            internalFormat,
            lookupLength,
            externalFormat,
            _DATA_TYPE_FLOAT,
            lookupEntries);
```

Step 6: Handling Errors

The **voError** class handles errors and prints debugging information. **voError** can provide simple error checking and call error handling routines if used after each API call.

After each API call the application can test for any new error conditions with a call to **getErrorNumber()** and get the description of any such errors with a call to **getErrorString()**. For example, to test for errors after polygonizing a volume, your code segment would look similar to the following:

```
voGeometryActions::polygonize(...);

if( voError::getErrorNumber() != voError::NO_ERROR )
    fprintf(stderr,”voError %s\n”,voError::getErrorString());
```

The error conditions that can arise are listed below:

```
enum voErrorType{
    NO_ERROR ,
    BAD_VALUE,
    BAD_ENUM,
    OUT_OF_MEMORY,
    UNSUPPORTED
};
```

Calls to **getErrorNumber()** and **getErrorString()** return the most recently set error code number and string, respectively, and then reset the current error number and string to **NO_ERROR** and **NULL,** respectively.

Setting Up Error Handlers

An alternative way of handing API errors is to set up an error handler with a call to **setErrorHandler(),** which allows the application to intercept all the errors and act on them immediately, as shown in the following code fragment:

```
voGeometryActions::polygonize(...);

if( voError::getErrorNumber() != voError::NO_ERROR )
    fprintf(stderr,"voError %s\n",voError::getErrorString());

voGeometryActions::draw(...);

if( voError::getErrorNumber() != voError::NO_ERROR )
    fprintf(stderr,"voError %s\n",voError::getErrorString());
```

This fragment has a similar effect to the following fragment:

```
void myExceptionHandler(void)
   {
     // inform the user AND reset the error flags and messages
     fprintf(stderr,"OpenGL Volumizer error %d: %s\n",
       voError::getErrorNumber(),voError::getErrorString());

     // take other actions e.g., reallocate storage
     ...
   }

   ...
   voError::setErrorHandler(myExceptionHandler);

   voGeometryActions::polygonize(...);
   voGeometryActions::draw(...);
   ...
```

The solution based on the error handler always reacts to the first error condition (even if it occurred internally within the API), it is less error prone, and less burdensome, which results in cleaner code. A pointer to the previously set handler (or NULL, if none was set) is returned to facilitate building of stacks of error handling functions.

Setting Error Conditions

Applications can set their own error number and error string using **setErrorNumber()** and **setErrorString()**, respectively.

Debugging Information

Volumizer prints out some debugging information.

setDebugLevel() enables you to determine the minimum level of debugging information you want to print. If it is set to 0, no debugging information is printed.

setDebugStream() sets the output for the debug information.

Step 7: Clean Up

Applications are required to clean up upon completion. This involves freeing all the objects used to describe the volumetric shape (tetra sets, brick set collections), but also all the voxel memory, transient geometry, and texture objects:

```
// free storage for TRANSIENT geometry
    for (int j1 = 0; j1 < maxBrickCount; j1++) {
        for (int j2 = 0; j2 < maxSamplesNumber + 1; j2++)
            delete aPolygonSetArray[j1][j2];
        delete aPolygonSetArray[j1];
    }
    delete[]aPolygonSetArray;
    delete allVertexData;

    // free all voxel data storage
    voBrickSetCollectionIterator collectionIter(aVolume);
    for (voBrickSet * brickSet; brickSet = collectionIter();) {
        // iterate over all bricks within the brickCollection
        voBrickSetIterator brickSetIter(brickSet);
        for (voBrick * brick; brick = brickSetIter();)
          voAppearanceActions::dataFree(brick);
    }

    // delete the volumetric object itself
    delete aVolume;

    delete aTetraSet;
```

Chapter 4

Sample OpenGL Volumizer Application

This chapter presents a sample OpenGL Volumizer application. It employs a simple trackball user interface and displays a textured cube.

The sample application is divided into two parts:

- "glwSimpleVolume.cxx" on page 68.
- "glwSimpleMain.cxx" on page 83

This chapter explains in detail the steps you take to create an OpenGL Volumizer application.

glwSimpleVolume.cxx

glwSimpleVolume.cxx sets the geometry and appearance parameters and displays the shape.

The protgramatic structure of *glwSimpleVolume.cxx* is characteristic of OpenGL Volumizer applications:

1. Inclusions and declarations
2. Set the appearance parameters
3. Allocate memory for geometry
4. Allocate memory for bricks
5. Read in the brick data
6. Create the geometry
7. Set drawing parameters
8. Polygonize the shape
9. Draw the shape
10. Clean up

Inclusions and Declarations

OpenGL Volumizer and standard C++ includes.

```
#include <stdio.h>
#include <stdlib.h>

#include <vo/GeometryActions.h>
#include <vo/AppearanceActions.h>
```

Method declaration that returns data about the geometry's file.
The reader must provide this.

```
extern int myGetVolumeSizesIfl(
    char *fileName, int &xSize, int &ySize,
    int &zSize,
    voExternalFormatType & diskDataFormat,
    voDataType & dataType);
```

Method declaration that returns texture and geometry values.
The reader must provide this.

```
extern int myReadBrickIfl(
    char *fileName, void *data,
    int xBrickOrigin, int yBrickOrigin,
    int zBrickOrigin,
    int xBrickSize, int yBrickSize,
    int zBrickSize,
    int xVolumeSize, int yVolumeSize,
    int zVolumeSize);
```

Declaration of scaling factor, one value for each dimension.

```
extern float modelScale[3];
```

The number of bricks in the largest copy of the volume.

```
int maxBrickCount = 0;
```

Determines maximum number of sampling planes.

```
int maxSamplesNumber = 256;
```

Value determines that every voxel is sampled.

```
float samplingRate = 1.0;
```

Vertex format; here only vertex coordinates are specified, not textures or colors.

```
voInterleavedArrayType interleavedArrayFormat =
    voInterleavedArrayTypeScope::V3F;
```

Storage for transient polygons.

```
voVertexData *allVertexData;
voIndexedFaceSet ***aPolygonSetArray;
```

Set the Appearance Parameters

Method for displaying an error string. This also resets the error condition state.

```
void
my_ExceptionHandler(void)
{
    fprintf(stderr,
        "OpenGL Volumizer error %d: %s\n",
        voError::getErrorNumber(),
        voError::getErrorString());
}
```

Method for handling appearance, which is a collection of bricksets.

```
voBrickSetCollection *
    my_InitAppearance(int argc, char **argv)
{
```

The size of the geometry and texture.

```
    int xVolumeSize, yVolumeSize, zVolumeSize;
    int xBrickSize, yBrickSize, zBrickSize;
```

Data formats to use externally and internally.

```
    voExternalFormatType diskDataFormat;
    voExternalFormatType externalFormat;
    voInternalFormatType internalFormat;
```

Describes what format the voxels are in.

```
    voDataType dataType;
```

Specifies whether the texture is sampled using bilinear or trilinear interpolation.

```
voInterpolationType interpolationType =
    voInterpolationTypeScope::DEFAULT;
```

Sets rendering mode to monochrome.

```
voRenderingMode renderingMode =
   voRenderingModeScope::MONOCHROME;
float loValue = -1.0, hiValue = -1.0;
```

Prompts user if filename of geometry was not included on the command line.

```
if (argc < 2) {
   fprintf(stderr, "Usage: %s inFileName\n",argv[0]);
   exit(1);
}
```

Retrieves the file name of the volume in the first argument of the command.

```
char *dataFileName = argv[1];
```

Get volume sizes and voxel format and exits.

```
if (myGetVolumeSizesIfl(dataFileName,
   xVolumeSize, yVolumeSize, zVolumeSize,
   diskDataFormat, dataType))
   exit(1);
```

Sets the texture size to the geometry size.

```
xBrickSize = xVolumeSize;
yBrickSize = yVolumeSize;
zBrickSize = zVolumeSize;
```

Returns, in the last three arguments, the optimal size for bricks based on the current machine's hardware.

```
voAppearanceActions::getBestParameters(
  interpolationType, renderingMode, dataType,
  diskDataFormat, internalFormat, externalFormat,
  xBrickSize, yBrickSize, zBrickSize);
```

Defines the brick collection. This method does not read or allocate any data, it only computes bricks' origins and sizes given the volume dimensions.

```
voBrickSetCollection *aVolume =
new voBrickSetCollection(
   xVolumeSize, yVolumeSize, zVolumeSize,
   xBrickSize, yBrickSize, zBrickSize,
   voPartialBrickTypeScope::TRUNCATE,
   1,
   internalFormat,
   externalFormat,
   dataType,
   interpolationType);
```

Handles errors.

```
if (voError::getErrorNumber() !=
  voErrorTypeScope::NO_ERROR) {
   fprintf(stderr, "%s", voError::getErrorString());
   exit(1);
}
```

Allocate Memory for Bricks

Uses *aVolume* as the brick set collection to iterate through.

```
voBrickSetCollectionIterator collectionIter(aVolume);
```

Iterates over all of the brick sets within the collection of brick sets.

```
for (voBrickSet * brickSet; brickSet =
  collectionIter();) {
```

Iterates over all of the bricks within each brick set.

```
  voBrickSetIterator brickSetIter(brickSet);
  for (voBrick * brick;brick = brickSetIter();)
```

Inside the double for loop, this line allocates memory for all of the bricks in all of the bricksets.

```
    voAppearanceActions::dataAlloc(brick);
}
```

Get the brick set of *aVolume* and then iterate over all the bricks in the brick set. *brick* is a pointer to a brickset.

```
 aVolume->setCurrentBrickSet(
    voPlaneOrientationScope::XY);

voBrickSetIterator
brickSetIter(aVolume->getCurrentBrickSet());
for (voBrick * brick; brick = brickSetIter();) {

    int xBrickOrigin, yBrickOrigin, zBrickOrigin;
    int xBrickSize, yBrickSize, zBrickSize;

    void *vdata = brick->getDataPtr();
```

Gets brick sizes and locations; the brick sizes might be different from those requested.

```
    brick->getBrickSizes(
        xBrickOrigin, yBrickOrigin, zBrickOrigin,
        xBrickSize, yBrickSize, zBrickSize);
```

Load the Brick Data

Reads in the brick data from disk. It is okay to use the brick's data area as a temporary buffer.

```
    myReadBrickIfl(dataFileName, vdata,
      xBrickOrigin, yBrickOrigin, zBrickOrigin,
      xBrickSize, yBrickSize, zBrickSize,
      xVolumeSize, yVolumeSize, zVolumeSize);
```

Replicate to the desired external format

```
    voAppearanceActions::dataConvert(
      brick, vdata, diskDataFormat);
```

Converts geometry to the desired external format. Scales the voxel values within a brick from a specified dynamic range to the following standard ranges: (<0,255> for ubyte, <0,65535> for ushort, and <0.0,1.0> for floats).

```
    voAppearanceActions::dataScaleRange(brick,
      loValue, hiValue);
}
```

If the texture is sampled using bilinear interpolation, transpose the volume.

```
if (aVolume->getInterpolationType() ==
  voInterpolationTypeScope::_2D)
    voAppearanceActions::volumeMakeTransposed
      (aVolume);
```

Determines the largest number of bricks in a brick set, in the brick set collection. This number is used to allocate the buffers.

```
voBrickSetCollectionIterator aCollectionIterator
  (aVolume);
voBrickSet *aBrickSet;

for (int j1 = 0; aBrickSet = aCollectionIterator();
  j1++)
```

Revise the value for *maxBrickCount* if the number of bricks in the brickset exceeds the current value for *maxBrickCount*, which was initialized as 256.

```
    if (j1 == 0 || aBrickSet->getBrickCount() >
      maxBrickCount)
        maxBrickCount = aBrickSet->getBrickCount();
```

Set the maximum number of sampling surfaces to twice the largest dimension.

```
maxSamplesNumber = xVolumeSize;
if (maxSamplesNumber < yVolumeSize)
    maxSamplesNumber = yVolumeSize;
if (maxSamplesNumber < zVolumeSize)
    maxSamplesNumber = zVolumeSize;
maxSamplesNumber *= 2;

return aVolume;

}
```

Create the Containers to Hold the Faces

Procedure to create a geometry.

```
voIndexedTetraSet *
 my_InitGeometry(
      voBrickSetCollection * aVolume)
{
    voError::setErrorHandler(my_ExceptionHandler);
    int xVolumeSize, yVolumeSize, zVolumeSize;
    aVolume->getCurrentBrickSet()->getVolumeSizes(
        xVolumeSize, yVolumeSize, zVolumeSize);
```

Set vertex coordinates. Texgen will be used to generate tex coords.

```
static float vtxData[8][3] = {
                0,            0,            0,
      xVolumeSize,            0,            0,
      xVolumeSize,  yVolumeSize,            0,
                0,  yVolumeSize,            0,
                0,            0,  zVolumeSize,
      xVolumeSize,            0,  zVolumeSize,
      xVolumeSize,  yVolumeSize,  zVolumeSize,
                0,  yVolumeSize,  zVolumeSize,
};
```

Defines the geometry to be drawn. Each row defines one tetrahedron, five of which define a cube. Values index into **vtxData[]**.

Constructs the tetra set.

```
static int cubeIndices[] =
   {
       0, 2, 5, 7,
       3, 2, 0, 7,
       1, 2, 5, 0,
       2, 7, 6, 5,
       5, 4, 0, 7,
   };

int tetraCount =
    sizeof(cubeIndices) / (VO_TETRA_VERTICES *
    sizeof(cubeIndices[0]));

int valuesPerVtx = sizeof(vtxData[0]) / sizeof(float);
```

Allocates storage for transient polygons; note that all the polygons share vertex data area **allVertexData[]** to minimize fragmentation **boundFaceCount()** determines the maximum number of indices required to store a **polygon (11).**

Sizeof refers to the number of vertices.

```
   voIndexedTetraSet *aTetraSet = new voIndexedTetraSet(
      (float *) vtxData,
       8,
       valuesPerVtx,
       cubeIndices,
       20);
```

Holds all intermediate PER_VERTEX information.

```
    allVertexData = new voVertexData(100000,
        valuesPerVtx);

    aPolygonSetArray = new voIndexedFaceSetPtrPtr
      [maxBrickCount];
    for (int j1 = 0; j1 < maxBrickCount; j1++) {
      aPolygonSetArray[j1] = new
        voIndexedFaceSetPtr[maxSamplesNumber + 1];
      for (int j2 = 0; j2 < maxSamplesNumber + 1; j2++)
         aPolygonSetArray[j1][j2] = new
            voIndexedFaceSet(allVertexData,
             boundFaceCount(tetraCount));
    }
    return aTetraSet;
}
```

Set Drawing Parameters

Initializes graphics.

```
void my_InitGfx(voIndexedTetraSet *,
    voBrickSetCollection * aVolume)
{
```

Optimizes based on performance.

```
    voAppearanceActions::volumeOptimize(aVolume,
       voOptimizeVolumeTypeScope::BEST_PERFORMANCE);
}
```

Draw the Volume

Defines draw method.

```
void my_DrawVolume(voIndexedTetraSet * aTetraSet,
    voBrickSetCollection * aVolume)
{
```

Chooses wireframe mode.

```
    GLboolean wireframe = GL_TRUE;
```

Type definitions of the number of bricks in the geometry and the modelview and projection matrices.

```
int brickNo;
GLdouble modelMatrix[16], projMatrix[16];
```

Returns the modelview matrix, which is the cumulative product of multiplying viewing and modeling transformation matrices.

```
glGetDoublev(GL_MODELVIEW_MATRIX, modelMatrix);
```

Returns the projection matrix, which is the viewing frustum.

```
glGetDoublev(GL_PROJECTION_MATRIX, projMatrix);
```

If using 2D textures, select an appropriate BrickSet from among XY,XZ,YZ. Note, that individual brick sets may have differrent dimensions and thus change certain PER_FRAME properites, for example, *BrickCount*.

```
if (aVolume->getInterpolationType() ==
  voInterpolationTypeScope::_2D

   aVolume->setCurrentBrickSet(
   voGeometryActions::findClosestAxisIndex(
     modelMatrix, projMatrix,
     voSamplingModeScope::AXIS_ALIGNED));

int BrickCount = aVolume->getCurrentBrickSet()->
    getBrickCount();
```

Disables texture and draws opaque (embedded) geometry.

```
voAppearanceActions::textureDisable(
   aVolume->getInterpolationType());
```

Draw tetrahedron wireframes in yellow.

```
if (wireframe == GL_TRUE) {
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    voGeometryActions::draw(aTetraSet,
     interleavedArrayFormat);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

Draw brick outlines in blue, but only if bricks are 3D.

```
  if (aVolume->getInterpolationType() ==
    voInterpolationTypeScope::_3D) {
      glColor4f(0.0, 0.0, 1.0, 1.0);
      for (brickNo=0;brickNo<BrickCount; brickNo++)
          voGeometryActions::draw(
            aVolume->getCurrentBrickSet()->getBrick(
            brickNo));
  }
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}
```

Polygonize the Shape

Given a *aTetraSet*, bricks' dimensions, and *count*, sampling mode and rate, find all slicing polygons and clip them to brick boundaries.

```
  int samplesNumber;
  float samplingPeriod[3] = {
      samplingRate / modelScale[0],
      samplingRate / modelScale[1],
      samplingRate / modelScale[2] };

  voGeometryActions::polygonize(
      aTetraSet, aVolume->getCurrentBrickSet(),
      interleavedArrayFormat,
      modelMatrix, projMatrix,
      aVolume->getInterpolationType() ==
      voInterpolationTypeScope::_3D ?
      voSamplingModeScope::VIEWPORT_ALIGNED :
      voSamplingModeScope::AXIS_ALIGNED,
      voSamplingSpaceScope::OBJECT,samplingPeriod,
      maxSamplesNumber,
      samplesNumber,aPolygonSetArray);
```

If *aVolume* is three-dimensional, make the face sets viewport aligned; otherwise make them sampling-axis aligned.

Specify the number of faces in the face set, call the face set *aPolygonSetArray*.

If not using **texgen(),** transform texture coords explicitly.

```
if (hasTextureComponent(interleavedArrayFormat))
    voAppearanceActions::xfmVox2TexCoords(
      aVolume->getCurrentBrickSet(),
      samplesNumber,aPolygonSetArray,
      interleavedArrayFormat);
```

Depth sort the bricks.

```
voSortAction aSortAction(
  aVolume->getCurrentBrickSet(),
  modelMatrix, projMatrix);
```

End of polygonize.

Draw the Shape

Enable 2D or 3D texturing depending on the interpolation type.

```
voAppearanceActions::textureEnable(
  aVolume->getInterpolationType());
```

If no explicit texture coordinates were given, enable **texgen()**.

```
if(!hasTextureComponent(interleavedArrayFormat))
  voAppearanceActions::texgenEnable();
```

Set up drawing mode and color

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glColor4f(1.0, 1.0, 1.0, 1.0);
glDepthMask(GL_FALSE);
```

Iterate over all bricks and sort them back to front.

```
for (brickNo = 0; brickNo < BrickCount; brickNo++) {
    int brickSortedNo = aSortAction[brickNo];
    voBrick *aBrick = aVolume->
      getCurrentBrickSet()->
      getBrick(brickSortedNo);
```

Update texgen equation for the current brick.

```
    if (!hasTextureComponent(
       interleavedArrayFormat))
       voAppearanceActions::texgenSetEquation(
        aBrick);
```

Load the texture to texture memory unless it is already there.

```
    voAppearanceActions::textureBind(aBrick);
```

Iterate over all sampling planes and draw them.

```
    for (int binNo=0; binNo < samplesNumber; binNo++)
{
        voGeometryActions::draw(
          aPolygonSetArray[brickSortedNo][binNo],
          interleavedArrayFormat);
    }
}
```

Disable the texture and disable the texture generation.

```
    voAppearanceActions::textureDisable(
      aVolume->getInterpolationType());
      voAppearanceActions::texgenDisable();
      glDepthMask(GL_TRUE);
} // end of procedure my_DrawVolume
```

Clean Up

Deletes geometry and textures.

```
void my_Cleanup(voIndexedTetraSet *aTetraSet,
    voBrickSetCollection *aVolume)
{
```

Frees storage for transient geometry.

```
for (int j1 = 0; j1 < maxBrickCount; j1++) {
    for (int j2 = 0; j2 < maxSamplesNumber + 1; j2++)
        delete aPolygonSetArray[j1][j2];
    delete aPolygonSetArray[j1];
}
delete[]aPolygonSetArray;
delete allVertexData;
```

Frees all voxel data storage.

```
voAppearanceActions::volumeUnoptimize(aVolume);
voBrickSetCollectionIterator
  collectionIter(aVolume);
for (voBrickSet * brickSet; brickSet =
  collectionIter();) {
```

Iterate over all bricks within the brick collection and release the memory used for the bricks.

```
    voBrickSetIterator brickSetIter(brickSet);
    for(voBrick * brick; brick=brickSetIter();)
        voAppearanceActions::dataFree(brick);
}
```

Delete volumetric geometry.

```
delete aVolume;
delete aTetraSet;

} // end of my_Cleanup
```

glwSimpleMain.cxx

glwSimpleMain.cxx contains a simple, trackball user interface, global declarations, and the main loop. The structure of the main loop follows the Motif programming model.

The programatic structure of *glwSimpleMain.cxx* is:

1. Inclusions
2. Global State Declarations
3. GUI Definitions
4. Handling Input From a Trackball
5. Main Routine

Inclusions

X, OpenGL, and standard C++ includes.

```
#include <X11/X.h>
#include <X11/Intrinsic.h>
#include <X11/keysym.h>
#include <Xm/Xm.h>
#include <GL/gl.h>
#include <GL/GLwMDrawA.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
```

Include statements for Volumizer actions.

```
#include <vo/AppearanceActions.h>
#include <vo/GeometryActions.h>
```

Global State Declarations

Declare a volume's geometry.

```
voIndexedTetraSet *aTetraSet;
```

Declare a volume's appearance.

```
voBrickSetCollection *aVolume;
```

The coordinates for the center of the cube, the dimensions of its sides, and the scaling factor in three dimensions.

```
float modelCentroid[3] = { 64.0,  64.0, 32.0};
float modelSize[3] = {128.0, 128.0, 128.0};
float modelScale[3] = { 1.8, 1.8, 3.0 };
```

GUI Definitions

Type declarations.

```
float latitude = 0, longitude = 0,
    lastLatitude = 0, lastLongitude = 0;
int lastX = 0, lastY = 0, lastButton = Button1;
Widget drawArea;
GLXContext glCtxt;
```

External symbols for appearance.

```
extern voBrickSetCollection *my_InitAppearance(
    int argc, char **argv)
```

External symbols for geometry.

```
extern voIndexedTetraSet *my_InitGeometry(
     voBrickSetCollection *);
```

External symbols for graphics intialization. X and Motif functions are declared as external C functions because the C++ preprocessor mangles C++ function names.

```
extern void my_InitGfx(voIndexedTetraSet *,
    voBrickSetCollection *);
```

External symbols for draw action.

```
extern void my_DrawVolume(voIndexedTetraSet *,
    voBrickSetCollection *);
```

External symbols for deleting geometry and appearance.

```
extern void my_Cleanup(voIndexedTetraSet *,
    voBrickSetCollection *);
```

Intialize internal data structures, including the visual data structure, XVisualInfo.

```
void GinitCB(Widget w, XtPointer , XtPointer )
{
    Arg arg;
    XVisualInfo *vip;

    XtSetArg(arg, GLwNvisualInfo, &vip);
    XtGetValues(w, &arg, 1);
    glCtxt = glXCreateContext(XtDisplay(w), vip,
        NULL, GL_TRUE);
    glXMakeCurrent(XtDisplay(w), XtWindow(w), glCtxt);

    my_InitGfx(aTetraSet, aVolume);
}
```

Draw method begins by clearing the screen to black.

```
void Draw()
{
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);
```

OpenGL methods to clear the buffer then load it with prescribed rotation, scaling, and translation values.

```
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glLoadIdentity();
    glRotated(-latitude-90, 1.0, 0.0, 0.0);
    glRotated(-longitude, 0.0, 0.0, 1.0);
    glScalef(modelScale[0], modelScale[1],
      modelScale[2]);
    glTranslatef(-modelCentroid[0],
        -modelCentroid[1], -modelCentroid[2]);
```

Swap buffers to display the loaded image.

```
    my_Draw(aTetraSet, aVolume);

    GLwDrawingAreaSwapBuffers(drawArea);
}
```

Callback for X Window System “expose” event.

```
void ExposeCB(Widget w, XtPointer , XtPointer cal)
{
    GLwDrawingAreaCallbackStruct *csp =
      (GLwDrawingAreaCallbackStruct *) cal;
```

Makes *glCtxt* the graphics context and attaches it to an X window.

```
    glXMakeCurrent(XtDisplay(w), XtWindow(w),
        glCtxt);
```

Specifies that the current matrix pertains to the projection **view**.

```
    glMatrixMode(GL_PROJECTION);
```

Initializes the current projection matrix so that it is unaffected by pre**vious** matrix values.

```
    glLoadIdentity();
```

Specifies the transformation between normalized-de**vice** and window coordinates.

```
    glViewport(0, 0, csp->width, csp->height);
    glOrtho(
        -modelSize[0] * 1.2, modelSize[0] * 1.2,
        -modelSize[1] * 1.2, modelSize[1] * 1.2,
        -modelSize[2] * 2.5, modelSize[2] * 2.5);
```

Specifies that the current matrix and all future transformations pertains to the model **view**.

```
    glMatrixMode(GL_MODELVIEW);
```

Then do the actual draw.

```
    Draw();
}
```

Handling keyboard input. The Escape key terminates the application and deletes the geometry and textures used in the application.

```
void KeybdInput(XKeyEvent * keyEvent)
{
    KeySym keySym;
    char buf[32];

    XLookupString(keyEvent, buf, 32, &keySym, NULL);

    switch (keySym) {
    case XK_Escape:
        my_Cleanup(aTetraSet,aVolume);
        exit(0);
    }
}
```

Handles button input for a virtual trackball:

This code temporarily stores, in *lastButton*, *lastLatitude* and *lastLongitude*, which button was pushed and the location of the trackball cursor when the button was pushed.

```
void ButtonInput(XButtonEvent * btnEvent)
{
    switch (btnEvent->button) {
    case Button1:
        lastButton = Button1;
        lastLatitude = latitude;
        lastLongitude = longitude;
        break;
    }
    lastX = btnEvent->x;
    lastY = btnEvent->y;
}
```

Uses the **ButtonInput()** data to move the shape accordingly. The left button rotates the shape. The movement is scaled by one-tenth.

```
void MotionInput(XMotionEvent * motionEvent)
{
    switch (lastButton) {
    case Button1:
        latitude  = (float) (lastLatitude  + (lastY
            - motionEvent->y) / 10.0);
        longitude = (float) (lastLongitude + (lastX
            - motionEvent->x) / 10.0);
        break;
    }
}
```

Callback function for trackball. Handles trackball button press, trackball motion, trackball button release, keyboard input, and notification.

```
void InputCB(Widget w, XtPointer, XtPointer callData)
{
    XmDrawingAreaCallbackStruct *dacs =
        (XmDrawingAreaCallbackStruct *) callData;
    glXMakeCurrent(XtDisplay(w), XtWindow(w),glCtxt);

    switch (dacs->event->type) {
    case ButtonPress:
        ButtonInput((XButtonEvent *) dacs->event);
        break;
    case MotionNotify:
        MotionInput((XMotionEvent *) dacs->event);
        break;
    case KeyRelease:
        KeybdInput((XKeyEvent *) dacs->event);
        return;
    case ButtonRelease:
        return;
    default:
        break;


    Draw();
}
```

Main Routine

Handles command line arguments and prints out basic information: version number and mouse button functionality.

```
void main(int argc, char **argv)
{
    fprintf(stderr,"%s\n",voVersion());

    fprintf(stderr,"\nLeft mouse button to rotate.\n");
```

Creates a geometry and an appearance.

```
    aVolume = my_InitAppearance(argc, argv);
    aTetraSet = my_InitGeometry(aVolume);
```

Type and value declarations.

```
    Widget toplevel;
    XtAppContext appCtxt;
    XVisualInfo *visual;
    Cardinal n1 = 0;
    Arg args[4];
```

Elements of the attribute group set the graphics state.

```
    GLint attribs[] =
    {GLX_RGBA, GLX_DOUBLEBUFFER, GLX_RED_SIZE, 1,
     GLX_GREEN_SIZE, 1, GLX_BLUE_SIZE, 1,
        GLX_DEPTH_SIZE, 1, None};
```

Sets up display.

```
    toplevel = XtAppInitialize(&appCtxt,
        "Lean", NULL, 0, &argc, argv, NULL, NULL, 0);
    XtSetArg(args[n1], XmNwidth, 400); n1++;
    XtSetArg(args[n1], XmNheight, 400); n1++;
    XtSetArg(args[n1], GLwNattribList, attribs);  n1++;
    drawArea = GLwCreateMDrawingArea(toplevel, "pb",
      args, n1);
```

visual is a pointer to an XVisualInfo structure, which uses the display, *toplevel*, and the graphics state, *attribs*. An error message is printed if the pointer is NULL.

```
visual = glXChooseVisual(XtDisplay(toplevel),
    0, attribs);
if (!visual)
    fprintf(stderr, “Bad visual\n”);
```

Callback registration. Callbacks take action upon user input.

```
XtAddCallback(drawArea, GLwNinputCallback,
    InputCB, NULL);
XtAddCallback(drawArea, GLwNginitCallback,
    GinitCB, NULL);
XtAddCallback(drawArea, GLwNexposeCallback,
    ExposeCB, NULL);
```

Create a hierarchy of shapes to group shapes into a logical collection.

```
XtManageChild(drawArea);
```

Creates an X window on the screen in which the shapes are displayed.

```
XtRealizeWidget(toplevel);
```

Enter event loop to handle user input.

```
XtAppMainLoop(appCtxt);
}
```

Chapter 5

Volumizer API at a Glance

In chapters one through four, you learned about a subsection of the Volumizer API. This chapter provides a high-level overview of the entire API.

Part Two of this book, “Advanced Topics,” describes in greater depth the advanced Volumizer concepts, classes, and methods not already covered in Part One.

Sections in this chapter include:

- “Functional Categories” on page 94
- “Class Hierarchy” on page 94
- “Brief Descriptions of the Volumizer Classes” on page 96

Functional Categories

The classes in the Volumizer API fall into one of the following functional categories:

- Classes that define a geometry, such as **voVertexData**, **voFaceSet**, and **voIndexedFaceSet**.
- Classes that perform operations on volumetric geometry, such as **voGeometryActions**.
- Classes that define appearance, such as **voTexture3D**, **voBrick**, and **voBrickSet**.
- Classes that perform operations on an appearance, such as **voAppearanceActions**.
- All other classes.

Class Hierarchy

Figure 5-1 shows the hierarchy of Volumizer classes. Child classes are shown to the right of their parent class.

Because the hierarchy is shallow, the classes have been grouped according to function.

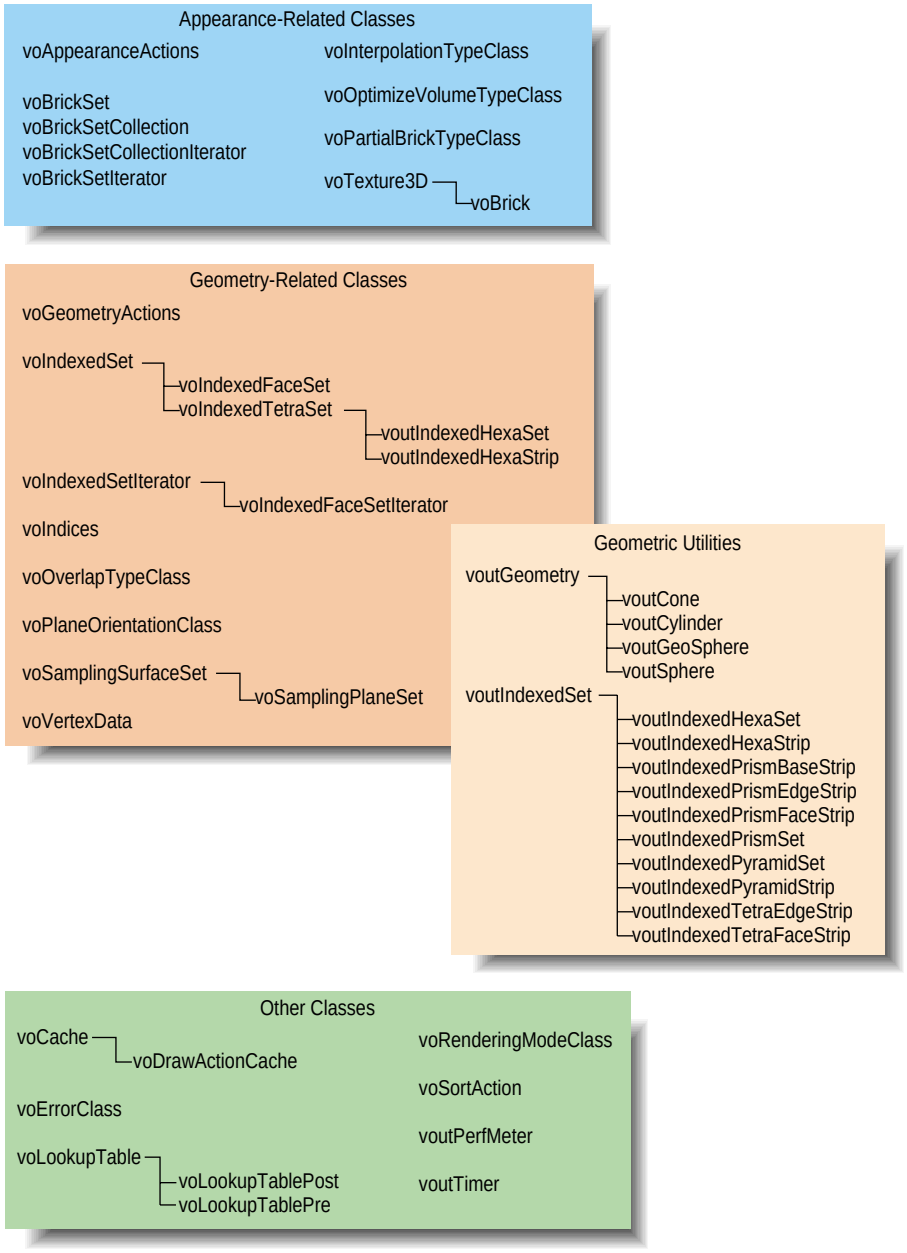

Figure 5-1 Volumizer Class Hierarchy Divided By Function

Brief Descriptions of the Volumizer Classes

This section provides a very brief description of each class in the Volumizer API. The classes are divided into the same sections shown in Figure 5-1.

Appearance-Related Classes	
voAppearanceActions	Implements operations on volume appearance, for example, textures.
voBrick	An anchored voTexture3D; that is, a texture that has position, in addition to sizes associated with it.
voBrickSet	A set of one or more bricks that define a large array of voxels.
voBrickSetCollection	Implements a collection of voBrickSets, that is, a set of sets of bricks.
voBrickSetCollectionIterator	Iterator for stepping through the collection of **voBrickSets**.
voBrickSetIterator	Iterator for stepping through the set of bricks.
voInterpolationTypeScope	An enum that specifies two- or three-dimensional interpolation.
voOptimizeVolumeTypeScope	An enum that specifies types of optimizations.
voPartialBrickTypeScope	An enum that specifies how to handle partially filled bricks.
voTexture3D	Atomic chunk of texture (3D or 2D) that maps directly on the hardware-supported textures of the underlying graphics API implementation, for example, GL_TEXTURE_3D_EXT and GL_TEXTURE_2D in OpenGL.

Geometry-Related Classes	
voGeometryActions	Implements operations on volume geometry, for example, TetraSets, and transient geometry generated in the polygonization process.
voIndexedFaceSet	An indexed geometry set describing a collection of polygons.
voIndexedFaceSetIterator	An interator for the **voIndexedFaceSet** class.
voIndexedSet	Abstract base class for indexed geometry sets.
voIndexedSetIterator	An abstract base class for geometry set iterators.
voIndexedTetraSet	An indexed geometry set describing a collection of tetrahedra.
voIndices	Data structure maintaining generic index information.
voPlaneOrientationScope	Specifies the alignment of the volume.
voSamplingPlaneSet	An class describing a set of parallel planes.
voSamplingSurfaceSet	An abstract class describing a set of sampling surfaces, for example., planes and spheres.
voVertexData	Data structure maintaining per-vertex information.
voutCone	Implements a cone abstract geometry class
voutCylinder	Implements a cylinder abstract geometry class
voutGeoSphere	Implements a sphere abstract geometry class tessellated via sub division
voutGeometry	Base class for all utility geometry classes
voutIndexedHexaSet	Implements a strip of hexahedra (boxes).
voutIndexedHexaStrip	Implements a strip of hexahedra (boxes).
voutIndexedPrismBaseStrip	Implements a strip of prisms, each connected to each other by a top/bottom face.
voutIndexedPrismEdgeStrip	Implements a strip of prisms, each connected to each other at a single edge.

voutIndexedPrismFaceStrip	Implements a strip of prisms, each connected to each other by a side face.
voutIndexedPrismSet	Implements a set of prisms
voutIndexedPyramidSet	Implements a set of pyramids.
voutIndexedPyramidStrip	Implements a strip of pyramids.
voutIndexedSet	Abstract base class for indexed utility geometry sets.
voutIndexedTetraEdgeStrip	Implements a strip of tetrahedra, connected at a single edge.
voutIndexedTetraFaceStrip	Implements a strip of tetrahedra, connected at a single point.
voutSphere	Implements a sphere abstract geometry class
Other Classes	
voCache	A structure for preserving transient data (for example, geometry resulting from polygonization) from frame to frame.
voDrawActionCache	A class for maintaining transient data (primarily geometry) from frame to frame.
voError	An error handling class.
voLookupTable	An abstract class describing lookup tables.
voLookupTablePost	Post-interpolation lookup table.
voLookupTablePre	Pre-interpolation lookup table.
voRenderingModeScope	Determines monochrome or color rendering
voSortAction	Implements sorting of a brick set.
voutPerfMeter	An API-specific set of timers used to time the phases of the volume rendering process. (utility)
voutTimer	Generic cumulative timer. (utility)

PART TWO

Advanced Volumizer Topics

Chapters 6 through 8 discuss advanced concepts and the Volumizer API not discussed in Part One.

Chapter 6, “Volumetric Geometry,” describes volumetric geometry.

Chapter 7, “Volumetric Appearance,” describes three-dimensional textures.

Chapter 8, “Customized Volume Drawing,” describes how to create a customized draw method.

Chapter 6

Volumetric Geometry

This chapter covers advanced topics in volumetric geometry. For an understanding of basic volumetric terms and concepts, see "" on page 9. For an understanding of the API associated with these basic concepts, see "Read Brick Data from Disk" on page 40.

This chapter discusses the following advanced, volumetric topics:

- "Data Structures" on page 102
- "Clipping Planes" on page 111
- "Arbitrarily Shaped Volumes of Interest" on page 111
- "Higher-Level Geometric Primitives and Solids" on page 113
- "Mixing Volumes and Surfaces" on page 114
- "Rendering Multiple Volumes" on page 116
- "Polygonizing Arbitrarily Oriented Cross Sections" on page 118
- "Shading" on page 122
- "Volume Roaming" on page 124
- "Picking Volumetric Objects" on page 125
- "Auxiliary Methods" on page 127

Data Structures

The Volumizer API defines several data structures to facilitate representation of volumetric geometry as well as the polygonal geometry that is derived to render the volumetric geometry, including **voIndexedTetraSet** and **voIndexedFaceSet**. Both classes are derived from the base class, **voIndexedSet**, and use **voVertexData** and **voIndices** internally.

voVertexData

voVertexData is an array of records. Each record describes a set of properties of a vertex, such as:

- Vertex coordinates
- Vertex colors
- Vertex normals
- Texture coordinates
- Optional user-defined data

Note: The records are similar in structure and function to interleaved vertex arrays, as specified by the OpenGL 1.2 specification.

The number of properties per vertex is application-dependent. Vertex coordinates (three values) are always required; other properties are optional. The only supported value type is float.

To see the use of **voVertexData** in the context of an application, see "Allocate Storage for Bricks" on page 40.

Preferred Order of Values in Records

The API associates very little semantics with the per-vertex values. For example, all per-vertex properties are re-sampled and clipped during **polygonize()** action regardless of their interpretation. In rare cases, for example, **voGeometryActions::draw()**, the order of data actually matters. The preferred order of per-vertex data is:

1. User-defined
2. Normals
3. Voxel coordinates
4. Colors
5. Vertex coordinates

Any of the optional properties can be omitted. For example, if the application specifies only texture and vertex coordinates, each record consists of six values (s,t,r,x,y,z); the record type is **T3F_V3F** (Texture 3 floats, Vertex 3 floats).

The following commonly used combinations are supported:

```
enum voInterleavedArrayType {
    DEFAULT,
    V3F,
    T3F_V3F,
    C3F_V3F,
    C4F_V3F,
    T3F_C3F_V3F,
    T3F_C4F_V3F,
    USER_V3F
};
```

USER_V3F is a catch-all type that can be used by applications that provide less common combinations of vertex properties. Applications that use USER_V3F or an unconventional order of properties should provide their own version of **voGeometryActions::draw()**.

Creating an Instance of voVertexData

The constructor for **voVertexData** is

```
voVertexData(int _countMax, int _valuesPerVertex, float* data=NULL);
```

You can construct an instance of **voVertexData** by requesting either:

- An empty instance of *_countMax* vertices, each with *_valuesPerVertex* values.

  In this case, a new data area is allocated.

- Passing a pre-allocated array of floats to the constructor.

  For more information about this option, see "Creating an Instance of voIndexedFaceSet" on page 106.

voVertexData Methods

In addition to a conventional set of constructor and accessors functions, **voVertexData** implements two methods:

- **empty()** sets the vertex count to zero, thereby marking the vertex array as empty.
- The **array operator []** returns a pointer to the requested vertex record.

voIndices

voIndices implements a simple array of indices, defined as follows:

```
voIndices(int _countMax, int* indices=NULL);
```

voIndices is a component of several **voIndexedSets**. **voIndices** has no semantics associated with it.

In addition to a conventional set of constructor and accessors functions, **voIndices** implements the following two methods:

- **empty()** sets the index count to zero, thereby marking the object as empty.
- The **array operator []** returns a requested element of the index array.

voIndexedSet

voIndexedSet is an abstract class that contains one **voVertexData** and one **voIndices** and serves as a base class for both **voIndexedTetraSet** and **voIndexedFaceSet**. **voIndexedSet** has no semantics associated with its data.

In addition to a conventional set of constructor and accessors functions, **voIndexedSet** provides several methods that are shared by all the derived classes:

- **empty()** marks the given set as not containing any data.
- The **array operator []** returns a pointer to an individual vertex record specified by the given index. (The semantics of such access varies among derived classes.)

voIndexedSetIterator

voIndexedSetIterator is an abstract class that enforces consistent traversal of **voIndexedSets**. For example, given a **voIndexedFaceSet**, that is, a set of polygons, you can use an instance of **voIndexedFaceSetIterator** to step through every vertex of each polygon.

In addition to a constructor, a **voIndexedSetIterator** has three methods:

- **nextVertex()** returns the next vertex of the current primitive, for example, a polygon, or NULL if all vertices of the current primitive were already traversed.
- **nextPrimitive()** steps to the next primitive.
- **doneP()** returns TRUE or FALSE depending on whether the whole indexed set was traversed.

voIndexedFaceSet

voIndexedFaceSet describes a collection of indexed polygons. These polygons are often different, planar slices of the same volume, as shown in Figure 6-1.

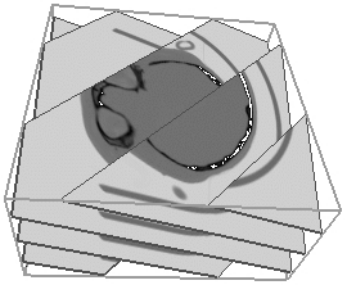

Figure 6-1 Indexed Face Sets

The polygons are the product of **voGeometryAction::polygonize(),** in which the volume is sliced into polygons.

Each group of indices describes a single polygon.

For more information about **polygonize()**, see "Polygonizing Volumes" on page 50.

Creating an Instance of voIndexedFaceSet

An empty instance of **voIndexedFaceSet** can be created by requesting a **voIndexedFaceSet** of *_countVertices* vertices, each with *_valuesPerVertex* fields, with, at most, *_countIndices* indices using the following constructor:

```
voIndexedFaceSet (int _countMaxV, int _valuesPerVertex,
    int _countMaxI);
```

For example,

```
voIndexedFaceSet *aFaceSet1 = voIndexedFaceSet(1000, 3, 5000);
```

This call constructs an empty instance of **voIndexedFaceSet** with a private vertex data array capable of holding 1,000 vertices each with three fields per vertex, and a private index array of 5,000 entries.

Another way to construct a face set is to specify a pre-allocated array of **voVertexData** and the maximum number of indices using the following constructor:

```
voIndexedFaceSet (voVertexData* _vertexData, int _countMaxI);
```

For example:

```
voVertexData *vertexData = voVertexData(1000, 3);
voIndexedFaceSet *aFaceSet2 = voIndexedFaceSet(vertexData, 5000);
```

The advantage of supplying a pre-allocated array is that a single instance of **voVertexData** can be shared among many **voIndexedFaceSets**, for example:

```
voVertexData *vertexData = voVertexData(1000, 3);
voIndexedFaceSetPtr aFaceSetArray[10];
for(int i1;j1<10;i1++)
   aFaceSetArray[i1] = voIndexedFaceSet(vertexData, 5000);
```

Pre-allotting the array reduces data fragmentation and lends itself more easily to parallelization.

Populating Face Sets with Polygons

All instances of **voIndexedFaceSet** constructed as above are initially empty. To populate them with polygons, use **voIndexedFaceSet::appendPoly()**, as follows:

```
float vdata[] = { 100,100,100,  200,100,100, 200,200,100, };
aFaceSet->appendPoly(vdata, 3);
```

This code excerpt adds a triangle, defined by *vdata[]*, to the face set.

If *vdata in* **appendPoly()** is NULL or points to the first empty record of the shared **voVertexData** array, Volumizer assumes that the application already placed the vertex data in the vertex array, so only indices are updated. Otherwise, the vertex data is copied from *vdata* into the vertex data array starting with the first available record. Appended vertex data are guaranteed to be copied into contiguous records.

voIndexedFaceSet Methods

voIndexedFaceSet inherits **empty()** from **voIndexedSet**. Calling **empty()** on an instance of **voIndexedFaceSet** constructed with a private vertex data area sets both the vertex and index counts to zero. The contents of the vertex data array and the index array is undefined after a call to **empty()**.

Calling **empty()** on an instance constructed with a shared vertex data sets the index count to zero, but leaves the vertex data count unchanged. This is done in order to assure that other objects that may be sharing the vertex array are unaffected. Application should empty the vertex array explicitly in such situations:

```
for(int i1;j1<10;i1++)
    aFaceSetArray[i1]->empty();
vertexData->empty();
```

Deallocating Indexed Face Sets

Calling a destructor on an instance of **voIndexedFaceSet** deallocates all storage allocated by the constructor. That means that vertex data storage is deallocated only on instances created with the first version of the constructor. For instances that use a pre-allocated array, the application must deallocate the storage, for example:

```
for(int i1;j1<10;i1++)
    delete aFaceSetArray[i1];
delete vertexData;
```

voIndexedFaceSetIterator

voIndexedFaceSetIterator facilitates traversal through a **voIndexedfaceSet**. For example, the following code steps though all polygons and prints out all the vertex coordinates for each polygon on a separate line:

```
for(voIndexedFaceSetIterator iter(aFaceSet);
    iter.done();iter.nextPrimitive())
        {
          while( (ptr = iter.nextVertex()) != NULL )
            fprintf(stderr,”(%f %f %f) “,
              ptr[valuesPerVertex-3],
              ptr[valuesPerVertex-2],
              ptr[valuesPerVertex-1]);
          fprintf(stderr,”\n”);
        }
```

voIndexedTetraSet

voIndexedTetraSet describes a collection of indexed tetrahedra; a collection of tetrahedra in Volumizer defines a solid shape. Each group of four indices in a tetraset form a single tetrahedron. The indices point into a vertex data array as in all other derivations of **voIndexedSet**.

Creating a voIndexedTetraSet

An empty instance of **voIndexedTetraSet** can be created by calling a constructor with *countVertices*, *valuesPerVertex,* and countIndex as arguments using the following constructor:

```
voIndexedTetraSet(int _countMaxV, int _valuesPerVertex,
    int _countMaxI);

voIndexedTetraSet(float* data, int _countMaxV, int _valuesPerVertex,
    int* _indices, int _indexMaxI);
```

_countMaxV is the number of vertices in the tetraset.

_valuesPerVertex are floating point values describing the vertices.

_countMaxI is the number of indices in the tetraset.

data must point to a linear array of floats that is at least *_countMax*_valuesPerVertex* long and the count is set to *_countMax*. If no data is specified, the storage is allocated and the vertex count is set to zero.

_indices points to index values.

_indexMaxI is the number of indices in the tetraset.

An alternative is to pass an array of floating point numbers describing the vertex data together with an array of indices using the following constructor:

```
voIndexedTetraSet(voVertexData* data, voIndices* indices);
```

For example, the code below constructs an instance of **voIndexedTetraSet** describing a five-tetrahedron decomposition of a cube. Each vertex of the cube has color and coordinates associated with it.

```
        const static float VSizeX = 128;
        const static float VSizeY = 128;
        const static float VSizeZ = 128;

        static float vtxData[8*6] = {
          1, 0, 0,  0,           0,       0,
          0, 1, 0,  VSizeX,      0,       0,
          0, 0, 1,  VSizeX, VSizeY,       0,
          0, 1, 1,  0,      VSizeY,       0,
          1, 0, 1,  0,           0,  VSizeZ,
          1, 1, 0,  VSizeX,      0,  VSizeZ,
          1, 1, 1,  VSizeX, VSizeY,  VSizeZ,
          0, 1, 1,  0,      VSizeY,  VSizeZ,
        };

        static int cubeIndeces[20] = {
          0, 2, 5, 7,
          3, 2, 0, 7,
          1, 2, 5, 0,
          2, 7, 6, 5,
          5, 4, 0, 7,
        };

aTetraSet = new voIndexedTetraSet( vtxData, 8, 6, cubeIndeces, 20);
```

Clipping Planes

Clipping planes into face sets is handled automatically either directly though OpenGL or with Volumizer API calls. In almost all cases, planes are clipped as part of **voGeometryActions::polygonize()** before drawing polygonized volumes.

In rare cases, an application might clip planes explicitly using **voGeometryActions::clip()**. For an example, see "Polygonizing Arbitrarily Oriented Cross Sections" on page 118.

Arbitrarily Shaped Volumes of Interest

Decoupling appearance from geometry for volumetric shapes and using tetrahedral tessellations allows you to select VOIs bounded by arbitrarily complex constraints. For example, a spherical VOI can be crudely approximated with an icosahedron tessellated into tetrahedra. Including a conventional clip plane produces a semi-spherical VOI. It is important to notice, that this kind of modeling does not require any special programming, but merely a different set of geometry to be rendered. This way, the task of modeling is clearly decoupled from the rendering stage.

Similarly, one can render everything outside of a sphere by tessellating the space between two concentric spherical shells and making sure that the radius of the external sphere is large enough to encompass the whole volume. Rendering a volume minus a cylinder can be useful for looking at occluded objects. For example, in radiation therapy planning, it is common to visualize the prostate un-obscured by the hip bone and the bladder, but with enough anatomical context to facilitate treatment planning.

Figure 6-2 shows a tessellation that enables the mandible to be treated as a separate model part.

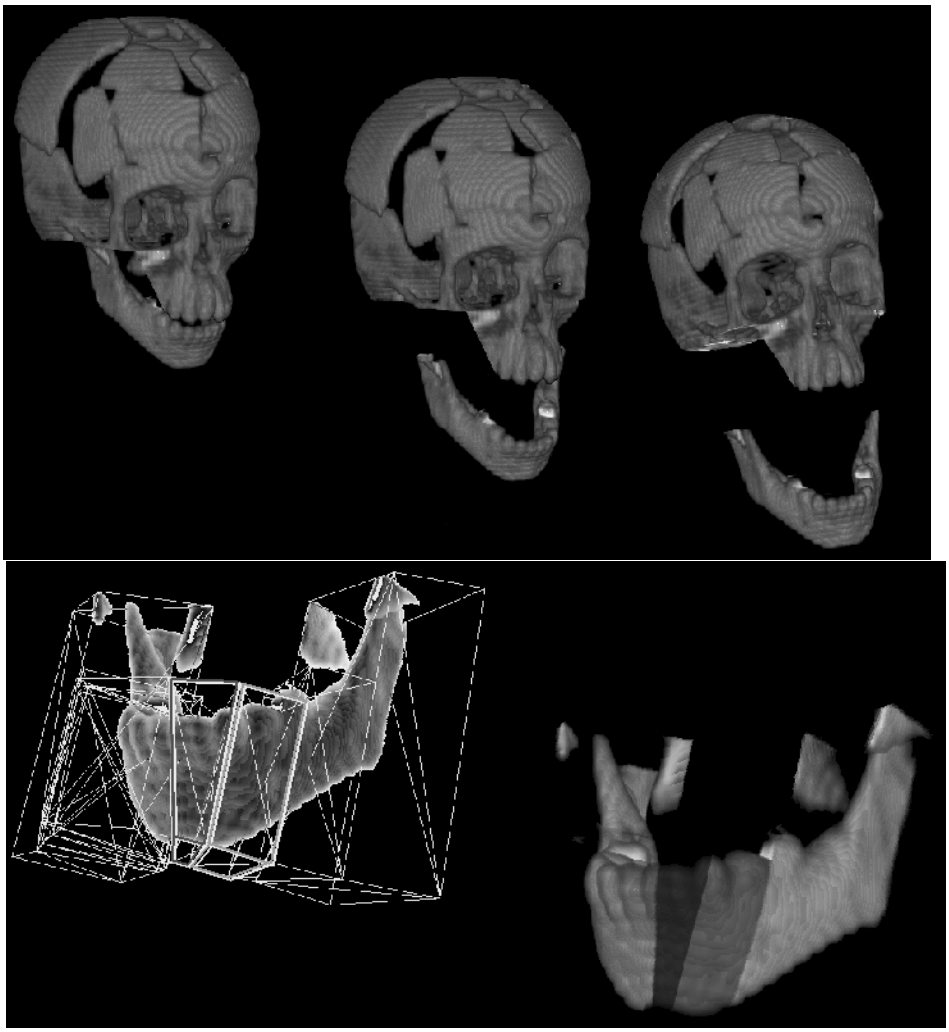

Figure 6-2 Arbitrarily Shaped VOI

Using Higher-Level Geometric Primitives

The Volumizer API can render only those geometric shapes that are sets of tetrahedra. At times, however, it is more convenient to specify higher level geometric shapes, such as boxes or spheres. This section describes a set of helper routines that facilitate such high-level volumetric modeling.

The utility functions provide polyhedral shapes based on tetrahedra, pyramids, prisms, and hexahedra, as shown in Figure 2-11 on page 19. In addition, there are utilities that enable applications to easily model cylinders, cones, and spheres.

Higher-Level Geometric Primitives and Solids

Volumizer offers the following higher-level polyhedral primitives:

- **voutIndexedTetraFaceStrip**
- **voutIndexedTetraEdgeStrip**
- **voutIndexedHexaSet**
- **voutIndexedHexaStrip**
- **voutIndexedPyramidSet**
- **voutIndexedPyramidStrip**
- **voutIndexedPrismSet**
- **voutIndexedPrismFaceStrip**
- **voutIndexedPrismBaseStrip**
- **voutIndexedPrismEdgeStrip**

There is also support for higher order solids (these are non-indexed):

- **voutCone**
- **voutCylinder**
- **voutSphere**
- **voutGeoSphere**

All these utility types are derived from an abstract base class, **voutGeometry,** and all the indexed types are also derived from **voutIndexedSet**. All of these classes provide a **tessellate()** method that returns a **voIndexedTetraSet** that is a tessellation of the given shape. For example:

```
voutSphere *aSphere(0,0,0,100, 32, 32);
voIndexedTetraSet *tetras = aSphere->tesselate();
```

voutSphere implements a sphere abstract geometry class.

Mixing Volumes and Surfaces

Rendering scenes containing conventional, surface-based and volumetric objects is essential for many applications. For example, for seismic data interpretation, it is often necessary to display oil wells and horizontal surfaces separating layers of geological material. Or, for the purpose of surgical simulation, you might like to render a scalpel, or a CAD model of a prosthetic device in the context of a patient's anatomy, as specified by a CT volume. Figure 6-3 was created by rendering a polygonal sphere inside of a CT volume.

Figure 6-3 Geometric (Sunglasses) and Volumetric Objects Rendered Together

The surface geometry being combined with volumes can be either opaque or translucent.

Rendering Opaque Geometry with Volumes

To add opaque geometry to a scene, follow these steps:

1. Enable Z-buffering.
2. Render the surfaces in the scene.
3. Disable Z buffer writes (while leaving Z test enabled).
4. Render the three-dimensional volumes.

Rendering Translucent or Overlapping Geometry with Volumes

The Volumizer API reduces all volumes to a set of semi-translucent polygons during the polygonization stage. To render translucent or overlapping geometry with volumes, you can use any renderer that knows how to handle transparency. You can merge the polygon lists from the volume and the surface models and render them in visibility order.

There are a number of techniques that can be used for that purpose. For example, Binary Space Partition (BSP) trees are commonly used to repeatedly depth sort a polygonal scene. Applications can take advantage of highly coherent structures of a volume-derived polygonal scene (many polygons share a supporting plane, and all planes are parallel) to create well balanced trees.

Transient Geometry Caching During polygonize

The **voIndexedFaceSets** generated by **polygonize()** can often be reused. For example, for AXIS_ALIGNED sampling, the set of **voIndexedFaceSets** remains unchanged from view to view and can be cached without any consequences. Similarly, in VIEWPORT_ALIGNED mode, you can reuse sampling geometry from the previous frame if the two viewpoints are not very different. While such lazy evaluation may result in some quality degradation, it can be used for increased interactivity.

voGeometryActions::draw() initializes and maintains cache content. *glwCache.cxx* provides an example implementation of **voGeometryActions::draw()**.

One thing to watch for when not using viewport-parallel sampling is perspective correction for your textures. When the sampling planes are parallel to the viewport, the textures do not need to be perspectively corrected because they are the same depth. In other cases, however, the underlying hardware correct the perspective otherwise artifacts may occur.

While two-dimensional textures are often perspective corrected, three-dimensional textures may not be (e.g., Impact) graphics in SGI's Indigo2 and Octane workstations). Therefore, it is safe to render in axis-aligned mode using two-dimensional interpolation, but it may not be safe to use other types of sampling with tree-dimensional textures.

Rendering Multiple Volumes

Often you want to display images like these:

- Two non-associated, non-overlapping volumes, such as clouds.
- Two associated, overlapping volumes, such as a volume of radiation dose distribution and the corresponding CT data set describing the patient's anatomy, as shown in Figure 6-4.
- Two unassociated, overlapping volumes, such as a a CT and PET scan of a patient's head.

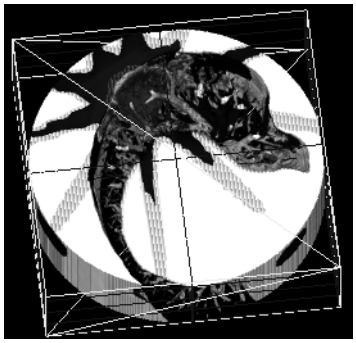

Figure 6-4 Overlapping Volumes

Rendering non-associated, non-overlapping volumes is straightforward: the bricks of all the volumes in the scene are merged in a single list and sorted from back to front and the rest of the algorithm proceeds as described in Chapter 3, "Programming Template."

There are, however, additional considerations when volumes overlap.

Overlapping Volumes

When the volumes overlap, you must decide how to:

- Combine overlapping parts
- Overlap texture memory

Combining Overlapping Volumes

The semantics of the application often dictate how to combine overlapping volumes. For example, if the volumes are interpenetrating clouds, they need to absorb light according to the laws of optics.

If, on the other hand, a radiation dose distribution is to be “painted” over the anatomy, the values need to be merged through multiplicative blending, as shown in Figure 6-4. In this case, it may be necessary to use an offscreen buffer to produce individual slices.

Texture Memory Overlap

Every pair of bricks that overlaps has to be resident in the texture memory simultaneously. This condition halves the maximum brick size value and requires some of the bricks to be loaded more than once. Because texture downloading is expensive, you should create as few downloads as possible.

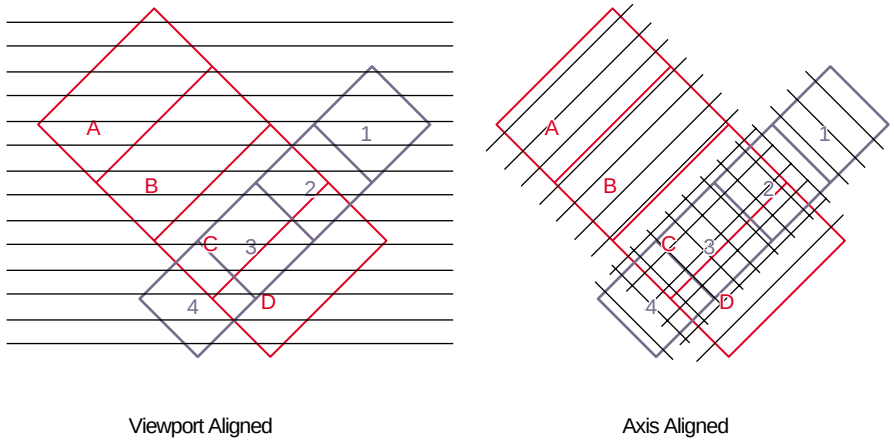

Figure 6-5 Merging Multiple Volumes

Figure 6-5 shows multiple volumes merged under viewport aligned sampling (on the left) and under axis-aligned sampling (on the right). In this example, it's best if the bricks are small enough so that three bricks fit the texture memory simultaneously. Otherwise, the same brick may have to be reloaded several times.

Interpenetrating Polygons

If the sampling of two volumes by **voGeometryActions::polygonize()** does not occur along an identical set of surfaces, for example, in axis-aligned planes, the situation becomes complicated. In this scenario, the polygons resulting from the polygonization of each volume need to be depth sorted. The depth sorting is complicated by the fact that some of the polygons might be interpenetrating. To correctly depth sort the polygons, you need to use a BSP tree or some similar mechanism.

Polygonizing Arbitrarily Oriented Cross Sections

Arbitrarily oriented cross sections, also called Multi-Planar Reformations (MPRs), of volumes are easily implemented on machines that support three-dimensional texture mapping hardware: the slicing plane is clipped, using **voGeometryActions::clip()**, to the volume's geometry and the brick boundaries and the resulting polygons are drawn with texturing enabled.

Note: The above technique may produce, what appears like geometric distortions on machines that do not perspectively correct three-dimensional textures (e.g., Impact graphics).

Without three-dimensional texture-mapping hardware, however, computing a tri-linearly filtered oblique slice through a volume is more challenging. This section describes how to accomplish this task using two-dimensional texture mapping and blending circuits commonly available on even the lowest-end machines.

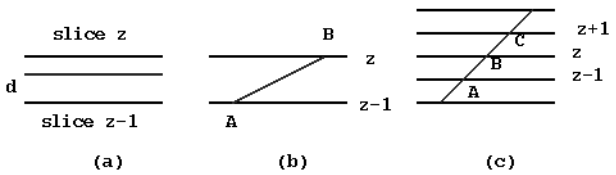

Figure 6-6 MPR with Two-Dimensional Texture Mapping

Figure 6-6 shows a tri-linearly interpolated slice using two-dimensional texture mapping. The three instances in the figure show:

(a) A weighted average between two two-dimensional images

(b) An oblique slice between two two-dimensional images

(c) An oblique slice through a stack of two-dimensional images.

Stack of Two-Dimensional Textures

Consider a stack of two-dimensional textures representing a three-dimensional volume of voxels. One can sample such volume only along any plane that coincides with any of the image planes to obtain a properly filtered image using only bi-linear interpolation.

Figure 6-6 shows a simple technique to sample along any plane that is parallel to the acquisition plane, but not necessarily coincident with any individual slice.

To obtain an image for the planes between the slices, use a weighted average of the two images, using the following procedure:

1. Enable the first two-dimensional texture.
2. Set the current color to (1-d, 1-d, 1-d, 1-d).
3. Draw the requested polygon.
4. Enable the second two-dimensional texture.
5. Set the current color to (d, d, d, d).
6. Enable additive blending before drawing the same polygon a second time.
7. Disable Z-testing or use the polygon offset has to assure that Z-fighting does not prevent the second pass from appearing the frame buffer.
8. Draw the second polygon.

Using Angled Slices

The above procedure can be extended to obtain a properly-textured polygon that is cutting at an angle through two adjacent two-dimensional images, as shown in Figure 6-6(b).

The polygon is drawn twice using additive blending. However, rather than using constant opacity for the polygon, the opacity of each vertex is set to either 0 or 1, depending on whether the vertex falls into the active texture or not. For example, when texture Z is enabled, the opacity of vertex A is set to 0 and the opacity of vertex B is set to 1. Therefore, when the first texture is enabled, the pattern mapped onto the polygon is effectively multiplied by a linear ramp fading away in the parts of the polygon that are further away from the active image. Subsequently, the other texture is enabled, and the opacities of the vertices are reversed.

To minimize state changes once the texture is enabled, all polygons that use the same state should be drawn; for example, in Figure 6-6(c), once texture Z was enabled, polygons A-B and B-C should be drawn.

Figure 6-7 shows the partial results of the algorithm applied to every other slice.

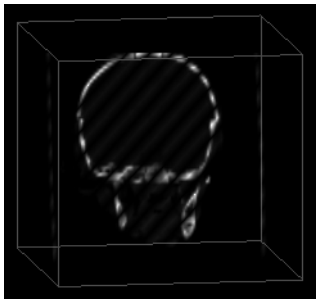

Figure 6-7 MPR Bands

You can compute an oblique slice through a stack of two-dimensional images by bounding each texture at most once and drawing the polygon exactly twice. The cross section, then, is computed at half the fill rate for two-dimensional textured polygons plus the overhead of computing the polygon's coordinates.

In the worst case, slicing through a n^3 volume requires computing $2 \times n^2$ polygonal vertices. However, they can be easily found using tri-linear interpolation. This technique significantly reduces the overall complexity of tri-linearly sampling the entire slice.

Multi-planar Reformatting Polygonization

polygonizeMPR() actions, described as follows, computes a set of polygons along an oblique section (Multi-planar Reformatting) clipped to brick boundaries. Two alternative actions differing by their inputs are provided. The first instance of the action will clip a plane, given its equation Ax+By+Cz+D=0. The second will clip an arbitrary polygon, described by their vertices.

```
static int polygonizeMPR(
    float planeEquation[4], voBrickSet* aBrickSet,
    voIndexedFaceSet*** polygonSet,
    voInterleavedArrayType interleavedArrayFormat);

static int polygonizeMPR(
    voVertexData* vData, voBrickSet* aBrickSet,
    voIndexedFaceSet*** polygonSet,
    voInterleavedArrayType interleavedArrayFormat);
```

The plane is given by its equation, *planeEquation[4]*.

The input polygon is described by **voVertexData**.

Note: Currently, this method is implemented only for two-dimensional textures.

Shading

Shading increases the realism of an image by applying lighting calculations to the object being rendered. For example, the image on the right in Figure 6-8 was shaded (using a Z-buffer shading technique), while the one on the left was rendered with simple opacity blending only.

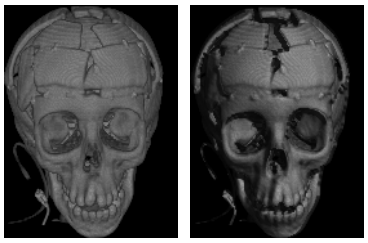

Figure 6-8 Shading Off and On

In addition to looking more realistic, the shaded image enhances many surface details that are not apparent in the blended rendition, for example, the small bones in the nasal cavity, and staples holding the pieces of forehead together.

Currently, the API does not define how polygons produced by **polygonize()** are blended. Applications can use OpenGL's **glBlendFunc()** to select a suitable blending, for example, OVER or MIP.

A more sophisticated scenario calls for computation of per-voxel gradient. The gradient is used as an approximation for the surface normal in lighting calculations. You can implement gradient-based shading by maintaining a gradient volume in addition to the original voxel data.

Tangent Space Shading

A significantly more flexible technique that allows for on-the-fly per-voxel gradient computation is based on the tangent-space techniques that were recently proposed for bump mapping (consult SIGGRAPH97 Proceedings, and SIGGRAPH98 Course 17 by Tom McReynolds for details). In this multi-pass approach, each sampling polygon is drawn twice using blending operations to estimate the component of the gradient in the direction of the light source.

More specifically, the polygon is drawn first into a off-screen memory. Subsequently, the vertex coordinates of the polygon are offset by a unit in the direction of the light source (alternatively, the texture coordinates are shifted by a corresponding fraction in the opposite direction).

The offset polygon is drawn for the second time subtracting it from the first image (this operation can also be implemented efficiently with an accumulation buffer). Effectively, this operation computes the projection of the gradient vector onto the direction of the light vector, or the dot product of these two vectors. Once the subtracted image is computed it is copied into the frame buffer and blended using conventional OVER operator.

The main advantage of the technique is the ability to compute the shading directly from the intensity data thus avoiding memory bloat and bandwidth issues which plague algorithms that require storing of the pre-computed gradient volume. Further, it allows for post-lookup, post-interpolation gradient calculation that may be required by high quality applications. The main drawback is use of the un-normalized gradient, which can produce shading artifacts similar to using un-normalized vertex normals with conventional models.

A sample implementation of this technique is provided with the distribution.

Volume Roaming

Many applications deal with very large volumes. In these cases, only a subregion of the volume can be displayed at any given time. This subregion is called the Volume of Interest (VOI).

The VOI can be of any volumetric shape. Figure 6-9 shows a cubical VOI in the midst of geological data.

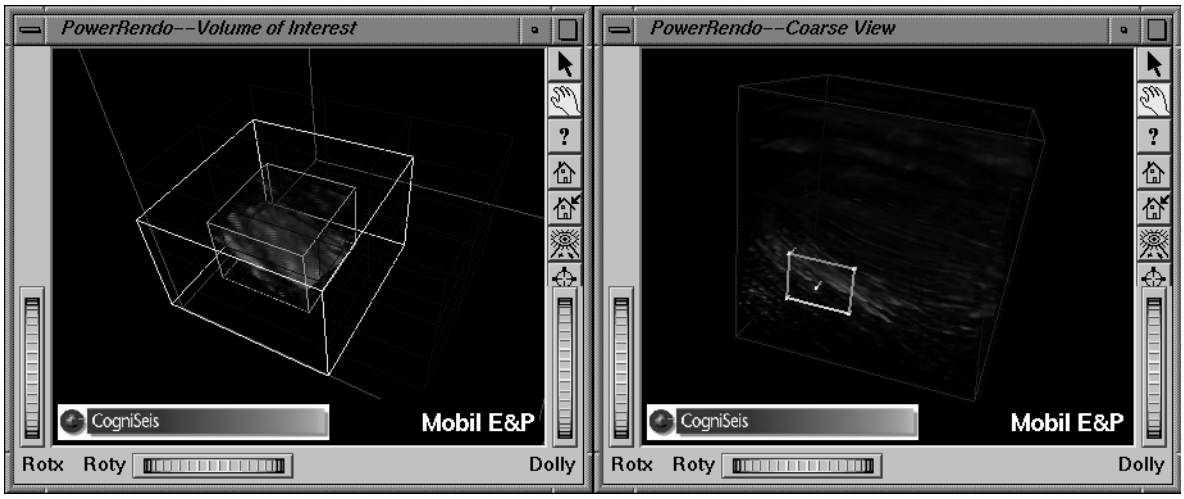

Figure 6-9 Volume of Interest

Implementing a Volume of Interest

You can enable a user to translate and scale a VOI, such as a cube or a sphere. For example, suppose you roam a 2048^3 (8 GB) volume bricked into 64 256^3 bricks. The VOI is a 512^3 cube, which can be translated and scaled. Every time the position or size of the VOI changes, the application has to:

1. Determine which bricks have shifted into the field of view.
2. Read those bricks in from the disk.
3. Determine which bricks have shifted out of the field of view.
4. Discard those bricks and mark them as inactive.

These tasks can be accomplished by defining the vertices of the geometry, **voIndexedTetraSet**, so they coincide with the vertices of the VOI rather than the vertices of the voxel data cube. The same approach can be used to manipulate arbitrarily shaped VOIs.

More importantly, thanks to virtualized (bricked) appearance, applications can control the amount of voxel data present in the memory and can implement just-in-time brick loading.

Picking Volumetric Objects

Many applications are interested in selecting objects being drawn. For volumetric shapes such selection can be on two different levels: sub-part level and voxel level.

Applications that tessellate objects into individual parts may require that an individual part be selectable. For example, a mandible may have to be selected and moved around for maxillofacial surgery planning. For this type of per-part selection it is best to use OpenGL picking and selection mechanisms. Simply draw the **faces** of the tetrahedra defining the part just as if you were dealing with a conventional, surface-based model, using **voGeometryActions::draw()**.

At the low level, an application may want to inspect individual voxels of a model. For example, a distance between two three-dimensional anatomical landmarks may be required for a morphometric measurement. Similarly, a seismic data interpretation application may want to automatically segment out a layer of geological material by simply pointing at it on the screen.

Voxel picking was provided for the purpose of analyzing the volume on a point by point basis. In order to make the picking functionality general purpose, the API does not make any application-specific decisions. Instead, an array of voxels along a line (typically the line of sight through the cursor) is returned. It is application's responsibility to do its own signal processing to extract features of interest, for example, find the closest voxel along the line of sight that has a value above the threshold.

To return an array of voxels and their coordinates for an intersected volumetric object, use the following **voGeometryActions** method:

```
static voBool pick(
    int pixelPosition[2], int viewport[4], voIndexedTetraSet* tetraSet,
    voBrickSet* aBrickSet, voInterleavedArrayType
    interleavedArrayFormat, GLdouble modelMatrix[16],
    GLdouble projMatrix[16], float samplingPeriod[3],
    voBool flag, voVertexData* coordinates, void* voxels);
```

This method determines whether or not *aBrickSet* is intersected by a line-of-sight ray through the cursor. The method returns VO_TRUE if the brickset is intersected or VO_FALSE otherwise.

The equation of the line is determined from the *(row, col)* screen position, the viewport, and modelview/projection matrices. This line is intersected with the volume described by *aTetraSet* and *aBrickSet*. Sampling period determines how often to sample along this line. The routine computes the values of voxels and their respective coordinates.

The requested voxel data is returned in *voxels* in the format and voxel type of the brick set. If *flag* is set to VO_FALSE, the original voxel values along the line of sight are returned using nearest neighbor interpolation. If *flag* is set to VO_TRUE, the current graphics state is applied, for example, tri-linear interpolation, lookup tables, or interpolated-per-vertex colors.

The first voxel-coordinate pair refers to the intersection point on the volume closest to the viewpoint; the last voxel-coordinate pair refers to the intersection point on the volume furthest from the viewpoint. The coordinates along the line are returned in coordinates as specified by the **voInterleavedArrayFormat** enum, as defined in "Preferred Order of Values in Records" on page 103.

Setting *flag* to VI_TRUE, if it is implemented in hardware, is generally faster for a single-brick brick set, but may result in fully reloading each brick in a multi-brick brick set through the texture memory. Setting *voxels* to NULL returns vertex data but no voxel data so that the application can compute their own voxel values.

The other form of the same **voGeometryActions** method is:

```
static voBool pick(int pixelPosition[2], int voewport[4],
    voIndexedTetraSet* tetraSet, voBrick* aBrick,
    voInterleavedArrayType interleavedArrayFormat,
    GLdouble modelMatrix[16], GLdouble projMatrix[16],
    float samplingPeriod[3], voBool flag, voVertexData* coordinates,
    void* voxels);
```

The second version of the method tests to see if a **voBrick** instead of a **voBrickSet** is intersected. Applications can use this overloaded method to apply hardware-assisted picking on a brick by brick basis. For example, the application can sort the bricks by depth and only apply the pick action to the closest brick thus reducing download requirements significantly.

Auxiliary Methods

The following **voGeometryActions** auxiliary methods determine the position of the camera and the direction it is facing from OpenGL matrices, respectively:

```
static void findCameraPosition(
    double eyePos[3], double viewDir[4], GLdouble modelMatrix[16],
    GLdouble projMatrix[16]);

static void findViewDirection (
    double viewDir[3], GLdouble modelMatrix[16],
    GLdouble projMatrix[16]);
```

findCameraPosition() determines the view direction and eye position given the modelview and projection matrices. If the projection matrix describes an orthographic projection, set **viewDir[3]** to 0.0 and do not try to find the eye position. If the matrices were singular, for example., scaled by factor of 0.0 for projective shadows, **voErrorNo** is returned as BAD_VALUE and no eye position is found.

findViewDirection() determines the viewing direction given the modelview and projection matrix. This should generally not be a problem, unless the matrices are singular; for example, the view frustum is reduced to 0 depth by a scaling factor of 0.0, which is a common practice for projective shadows technique. If the direction cannot be computed, **voErrorNo** is set to BAD_VALUE, and **viewDir[]** is set to (0.0,0.0,1.0).

Chapter 7

Volumetric Appearance

This chapter discusses advanced topics in volumetric appearance. For an understanding of basic appearance concepts, see “Decoupling Geometry and Appearance” on page 12. For an understanding of the API associated with these concepts, see “Step 2: Define Appearance” on page 36.

This chapter discusses the following topics:

- “Texture Bricking” on page 130
- “Brick Sets” on page 143
- “Brick Set Collections” on page 143
- “Allocating Brick Data” on page 144
- “Loading Brick Data” on page 145
- “Creating Custom Loaders” on page 146
- “Test Volumes” on page 148

Texture Bricking

Voxel data sets often exceed the size of texture memory. To handle such large data sets, volumes are subdivided into a number of chunks, each of which are small enough to fit into texture memory. These chunks are called bricks.

A **voBrick** is a hexahedral (box-like), three-dimensional textures that you use to approximate volumes. A **voBrick** is the same as a **voTexture3D** object except that a **voBrick** has a defined position; **voTexture3D** does not.

Note: A **voBrick** can also represent a two-dimensional texture when you set one of the **voBrick**'s dimensions to one.

Data Types

A **voBrick** can hold voxels represented by various data types. OpenGL Volumizer supports the following types:

```
enum voDataType{
    DEFAULT,
    UNSIGNED_BYTE,
    BYTE,
    UNSIGNED_BYTE_3_3_2_EXT,
    UNSIGNED_SHORT,
    SHORT,
    UNSIGNED_SHORT_4_4_4_4_EXT,
    UNSIGNED_SHORT_5_5_5_1_EXT,
    UNSIGNED_INT,
    INT,
    INT_8_8_8_8_EXT,
    INT_10_10_10_2_EXT,
    FLOAT
};
```

These data types map directly onto types supported by OpenGL (consult documentation on **glDrawPixels(3G)** or **glTexImage2D(3G)** for details). For example, a data type of BYTE indicates that every voxel holds a single, 8-bit value represented as an unsigned char. Similarly, INT_8_8_8_8_EXT indicates that four 8-bit values, for example, RGBA components, are packed into a single voxel of type unsigned int.

The choice of data type is driven by convenience, performance, and memory usage. More compact types consume less memory and may download associated textures faster. Some types are faster to convert to the native format of the graphics accelerator.

Data Formats

The data format refers to the number of channels encoded in the voxel information, for example, a one-channel data format specifies the INTENSITY of the voxel; a four-channel data format might specify the red, green, blue, and alpha (RGBA) values of the voxel.

In Volumizer, data can be used in a variety of formats at different processing stages, as shown in Figure 7-1.

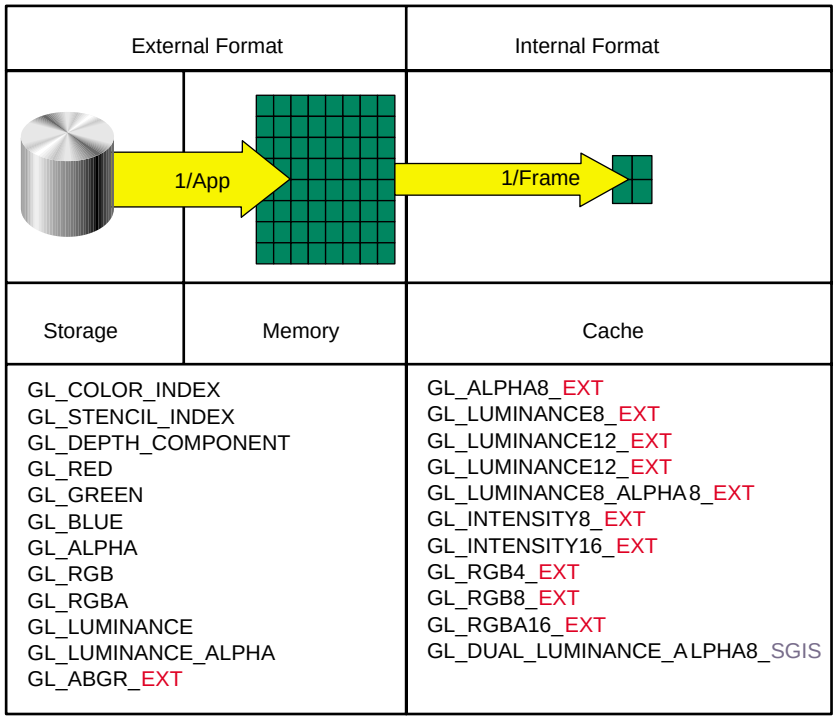

Figure 7-1 Data Formats

Data Format Domains

As voxel data is processed at various stages of the visualization pipeline, the data can be stored in various formats. In Volumizer, there are three places where the data can be stored in different formats:

- Native format—is the data format that is used to store data on the disk.
- External format—is the data format used by the application.
- Internal format—is the format that the underlying graphics API uses to save the voxel data in texture memory.

We refer to the transitions between these stages as “texture read” and “texture download” respectively.

For example, your data may be stored on disk as a single value, INTENSITY, per voxel. You may choose to use a LUMINANCE_ALPHA external format for your application-side processing. This choice requires that each voxel value be duplicated after a brick is read. Finally, you could use the RGBA internal format as the storage format in texture memory. The voxel data may need to be converted as it goes from one processing stage to another.

You specify each of these formats in the **voTexture3D** constructor.

Every time a brick is downloaded from the host memory to texture memory, the formats are converted. Format choices can be driven by convenience, performance, and memory usage considerations.

Data Format Values

Because native and external formats are conceptually the same, they can both be any of the following values:

```
enum voExternalFormatType{
    DEFAULT,
    INTENSITY,
    LUMINANCE_ALPHA,
    LUMINANCE,
    RGBA,
    ABGR_EXT
};
```

The following internal formats, on the other hand, are more sensitive to the internal memory layout:

```
enum voInternalFormatType{
    DEFAULT,
    INTENSITY8_EXT,
    LUMINANCE8_EXT,
    LUMINANCE8_ALPHA8_EXT,
    RGBA8_EXT,
    RGB8_EXT,
    RGBA4_EXT,
    QUAD_LUMINANCE8_SGIS,
    DUAL_LUMINANCE_ALPHA8_SGIS
};
```

The conversion from the native format to the external format is typically only done once per-brick during the initialization phase. The conversion from external to internal format has to be done every time a brick is downloaded from the host to texture memory.

There may be a performance penalty associated with specific formats depending on the underlying hardware. Therefore, it may be advantageous to keep all the formats as similar as possible.

Optimal Formats

There is typically an optimal format for any given platform. For example, on Impact graphics hardware, LUMINANCE_ALPHA and LUMINANCE8_ALPHA8_EXT produce optimal performance for grayscale rendering. However, it results in a times-two memory bloat because the same piece of information, the voxel's intensity, is replicated in the ALPHA channel.

On the other hand, using a "tighter" external format reduces your application's memory requirements. So an application can request INTENSITY as the external format and LUMINANCE8_ALPHA8_EXT as internal and minimize its memory usage at the expense of run-time performance penalty during texture download.

Converting Between Formats

In situations where the disk format differs from the external format, voxel data can be suitably converted with help of **voAppearanceActions::dataConvert()** after it was read:

```
static int dataConvert(voTexture3D* aTexture3D, void* data,
    voExternalFormatType diskFormat);
```

In this method the voxel data, *aTexture3D,* is converted to the specified format, *diskFormat*, and the result is pointed to by *data*.

data can point at the brick's data storage, in which case the conversion is done in place. This may be a little faster (and certainly more economical in terms of memory usage and code size) than using an additional I/O buffer.

Scaling Data

Some data values have to be scaled into the range expected by the graphics API. For example, if the voxel data type is float, it has to fit in the range of <0.0, 1.0>. Similarly, unsigned short voxels need to be scaled to <0, 65535>.

voAppearanceActions::dataScaleRange() expands the values of *data* to span the entire dynamic range of the underlying data type.

Example 7-1 initializes the voxel data by loading, converting, and expanding it.

Example 7-1 Initializing Voxel Data

```
voBrickSetIterator aBrickSetIter(aBrickSet);
for(voBrick *brick; brick = aBrickSetIter(); ) {
    unsigned char *vdata = (unsigned char *)
        voTexture3DActions::dataAlloc(brick);
    myReadBrickIfl(fileName, vdata, xBrickOrigin, yBrickOrigin,
        zBrickOrigin, xBrickSize, yBrickSize, zBrickSize);

    // convert to the desired externalFormat
    Texture3DActions::dataConvert(brick,vdata,INTENSITY);

    // expand the values to span the whole dymanic range
    voAppearanceActions::dataScaleRange(brick, loValue, hiValue);

}
```

Consult *InitAppearance.cxx* for sample source code.

Interleaving Bricks

On certain architectures, texel values are internally stored and manipulated as RGBA quadruples regardless of their original format. Therefore, even if the external, application side voxel format is INTENSITY, this single value is replicated 4 times during the download and stored internally as 1111. This means that a 4 MB texture memory can hold at most 1 MVoxel of intensity data. Other forms of packing texels are also possible. On some machines it is possible to increase the effective size of the texture memory and improve the download performance by brick interleaving.

When two bricks are interleaved, their voxels are combined together: the first voxel of the first brick is followed by the first voxel of the second, and so on. This way two values (one from each brick) in the INTENSITY_ALPHA format can be packed on the host into a single RGBA value, one in each channel pair. This 2-way interleaved texture can be transferred into the texture memory with a single command (reducing download effort 2-fold) and either one of the two original textures can be selected.

To determine if two bricks are interleaved, use **voTexture3DActions::cliqueTest(brick1, brick2)**. Only 2-way interleaving is currently supported in Volumizer.

Advantages of Interleaving Bricks

Interleaving bricks provides two important advantages:

- Transfer of appearance information is dramatically increased.

  If, for example, two LUMINANACE_ALPHA voxels are stored in a single RGBA value, the texture transfer rate is increased by 200%.

- Memory is conserved.

  If, for example, two LUMINANACE_ALPHA voxels are stored in a single RGBA value, the number of voxels that can be held in memory is increased by 200%.

Manually Interleaving Multiple Bricks

To perform two-way interleaving of bricks manually, use one of the following methods, respectively:

```
static int textureInterleave (
    voTexture3D* aBrick1, voTexture3D* aBrick2);
```

In order for two bricks to be suitable for interleaving they both have to be in compatible format. That is, their sizes, with a call to data types, and external formats have to be identical.

Note: An application can easily interleave all of the bricks of the brick set with a call to **oAppearanceActions::volumeOptimize()** using the INTERLEAVE value in the argument. **voAppearanceActions::textureInterleave()** is reserved for unconventional uses.

Creating Texture Objects

The TEXTURE_OBJECTS flag takes advantage of the Texture Manager in OpenGL by creating texture objects out of textures thus relieving the application from explicit texture memory management. Without texture objects, the application itself has to keep track of the texture memory usage. For example, if the whole volume fits into the texture memory, the application should make sure not to repeatedly re-load the texture.

With texture objects, the texture manager takes over this bookkeeping task. If a texture object is bound repeatedly, no explicit downloads occur. For more information about texture objects, see *OpenGL Programming Guide*.

Note: Texture object creation requires a valid graphics context.

When Not to Use Texture Objects

If a texture object is used for only one or two frames and then discarded, textures should not be converted into texture objects because the advantage of single-time processing is lost. Performance might be harmed because the creation of a new texture object for every frame is more costly than processing a texture for each new frame.

For example, if your application displays a beating heart, texture object creation should not be enabled because the texture changes in each frame.

Voxel Values

Voxel values do not have to represent the intensity or color, but can describe other values, such as stencils, also called tags. Tags are discussed in “Tagged Voxels” on page 137.

Similarly, data in the voxel may be subject to interpretation by applications. For example, an RGB value may contain gradient components rather than colors.

Tagged Voxels

You might tag individual voxels with class identifiers to, for example:

- Render different parts of a volume with different parameters.

  For example, in clinical applications, rendering a variety of tissues with different color/transparency parameters.

- Clip textures to arbitrary surfaces in situations where the surface is too fine to be tessellated into individual tetrahedra.

  For example, when interpreting seismic data, you volumetrically render only the sub-volume that falls between two horizontal planes. Defining the sub-volume as *WIDTH* x *HEIGHT* number of hexahedra may not be a viable alternative so a tagged volume should be used instead.

The functionality provided in the current release of the API allows application programmers to render tagged volumes using a simple multi-pass algorithm. Namely, the application can maintain an additional volume of tags and use stencil planes to select individual classes for rendering.

Voxel Reference Frame

Just as you can map two dimensional texture coordinates onto a polygon, you can map voxel coordinates onto volumetric geometry (tetrasets), as shown in Figure 7-2.

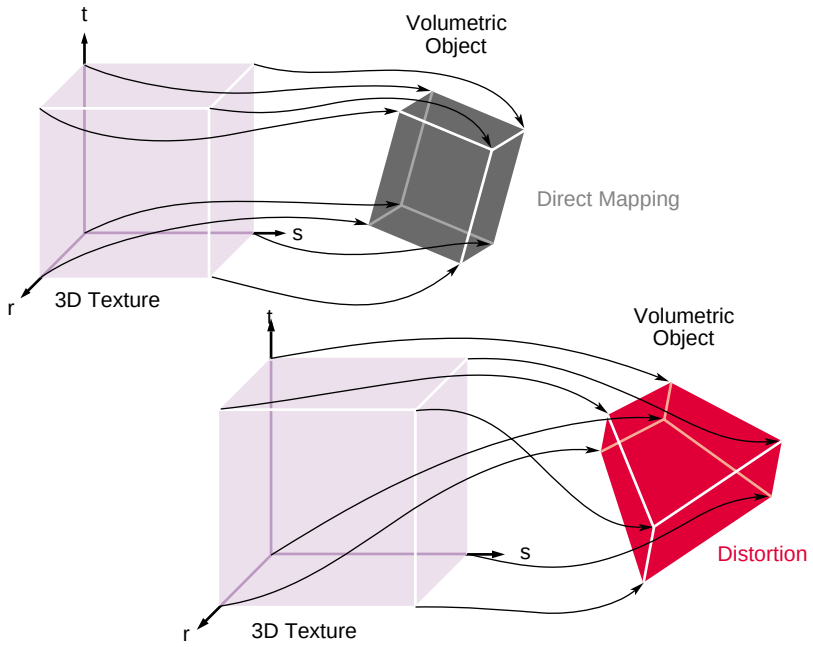

Figure 7-2 Voxel Coordinates

In the OpenGL Volumizer API, the geometry and voxel reference frames coincide: the voxel coordinates span [0,0,0] to [*xVolumeSize*-1, *yVolumeSize*-1, *zVolumeSize*-1]. When the size of the volumetric geometry matches the size of the voxel data array, the voxel and vertex coordinates are identical. Since this is a very common occurrence, applications do not have to explicitly specify texture coordinates in this situation.

However, there are situations where there may not be a 1-to-1 mapping from coordinates to voxel coordinates. For example, that may be required for modeling reasons: the volumetric object may have to exist in the world space that has dimensions of <1.0,1.0,1.0> even though the voxel array has sizes of <256.0,256.0,256.0>. In a more involved scenario, the volume may be deformed. In such situations, the application is required to explicitly specify both voxel and vertex coordinate triples.

The voxel reference frame is different from the texture reference frame. For a single brick volume, there is a simple linear mapping that takes the voxel coordinates and scales them back to <0.0, 1.0> range so that they can be used as texture coordinates. In the general case of a multi-brick volume, there is a series of such piecewise linear mappings that map a voxel coordinate into texture coordinates (s, t, r) within each individual brick by applying a suitable scale and bias.

During polygonization, the voxel coordinates, if specified, are sampled and clipped with the vertex coordinates. To draw the resulting polygons, however, these clipped voxel coordinates have to be transformed into the brick texture's reference frame such that the coordinate system, (s, t, r), spans [0,0,0] to [1,1,1] for each brick.

voAppearanceActions::xfmVox2TexCoords(), defined as follows, performs this transformation for each **voFaceSet** within a brick. It must be used before any draw calls.

```
static void xfmVox2TexCoords(voBrick* brick,
    voIndexedFaceSet* aFaceSet,
    voInterleavedArrayType interleavedArrayFormat,
    voPlaneOrientation orientation);

static void xfmVox2TexCoords(voBrickSet* aBrickSet,
    int samplesNumber, voIndexedFaceSet*** aFaceSet,
    voInterleavedArrayType interleavedArrayFormat);
```

In the case where the voxel and vertex coordinates match, it is unnecessary to specify the voxel coordinates. Instead you use **voAppearanceActions::texgenEnable()** once per drawing loop to generate the texture coordinates, and **voAppearanceActions::texgenSetEquation()** for each brick before drawing any face sets, as follows:

```
  if( interleavedArrayFormat == _V3F )
    voAppearanceActions::texgenEnable();
  else
    voAppearanceActions::xfmVox2TexCoords(
      aBrickSet,
      samplesNumber,
      aPolygonSetArray,
      interleavedArrayFormat);

  // iterate over all bricks
  for(brickNo=0;brickNo < BrickCount;brickNo++)
  {
    ...

    // update texgen equation for the current brick
    if( interleavedArrayFormat == _V3F )
      voAppearanceActions::texgenSetEquation(aBrick);

    // iterate over all sampling planes
    for(int binNo=0;binNo < samplesNumber;binNo++)
      voGeometryActions::draw(

aPolygonSetArray[brickSortedNo][binNo],GL_FILL,interleavedArrayFormat);

  }
    if( interleavedArrayFormat == _V3F )
    voAppearanceActions::texgenDisable();
```

Generating Voxel Coordinates

Generating voxel coordinates with **voAppearanceActions::texgenEnable()** is convenient and it may increase performance because fewer data items are transmitted down the graphics pipeline. In some situations, however, voxel coordinate generation can harm performance because the coordinates cannot be cached and the generation relies on hardware acceleration, which a machine may not have.

Texture Matrices

Instead of using **xfmVox2TexCoords()** to explicitly transform the voxel coordinates into the brick texture reference frame, you can use a texture matrix to apply scale and bias to brick coordinates on the fly. Use **voAppearanceActions::textureMakeMatrix()**, defined as follows, to construct a suitable matrix and push it onto the texture matrix stack with standard OpenGL calls.

```
static void textureMakeMatrix (voBrick* aBrick, float* matrix);
```

Setting Brick Sizes

You can set and return the size of a **voBrick** using the following **voBrick** methods:

```
void setBrickSizes(float _xmin, float _ymin, float _zmin,
    float _xSize, float _ySize, float _zSize)

inline void getBrickSizes (int& _xmin, int& _ymin, int& _zmin,
    int& _xSize, int& _ySize, int& _zSize);

inline void getBrickSizes (float* _position, float* _size);
```

These methods use, as arguments, the coordinates of kitty-corner corners of a hexahedral brick to set or return its size. The *size* values are relative to the *min* values rather than the coordinate-axis origin. For example, if the *min* values are (1, 1, 1), the *size* values could also be (1, 1, 1), in which case, the kitty-corner coordinates are (2, 2, 2).

Note: The brick origin can be an arbitrary number, but the brick sizes are typically (but not necessarily) powers of two.

Setting Brick Sizes Automatically

Selecting optimal brick sizes for a shape is tedious. Volumizer, therefore, provides the following helper routine to select the optimum brick size:

```
xBrickSize = xVolumeSize;
yBrickSize = yVolumeSize;
zBrickSize = zVolumeSize;

voAppearanceActions::getBestParameters(
    interpolationType, renderingMode, dataType, // In
    diskDataFormat,                             // In
    internalFormat, externalFormat,             // Out
    xBrickSize, yBrickSize, zBrickSize);        // In/Out
```

For more information about **voAppearanceActions::getBestParameters()**, see "Step 2: Define Appearance" on page 36.

To see sample code that determines brick size, refer to *InitAppearance.cxx*.

Drawing Brick Outlines

You can draw the outline of a brick using the following action:

```
static void voGeometryActions::draw(voBrick *);
```

You might draw the outline of a brick for debugging purposes.

Brick Sets

Typically, it takes more than one brick to represent a volume. A number of such adjacent, possibly-overlapping, bricks constitute a **voBrickSet**. Applications can use **voBrickSet** to access volume size, requested brick sizes, handles to individual **voBrick** objects in the set, and the set's orientation.

For more information about implementing brick sets, see "Allocate Storage for Bricks" on page 40.

Brick Set Collections

Some volumes must be represented by multiple **voBrickSets**. For example, machines that do not support three dimensional texture mapping require maintaining three separate copies of a brickset: one for each major axis, as shown in Figure 2-18 on page 26. In this case, each copy is represented by a separate **voBrickSet** and the entire volume is represented by the collection of the three **voBrickSets**.

voBrickSetCollection works like a switch that allows applications to select one of the **voBrickSet**s in the collection. For example, the following call selects the **voBrickSet** within the collection sliced in planes that are the most perpendicular to the line of sight:

```
aBrickSetCollection->setCurrentBrickSet(
   findClosestAxisIndex(modelMatrix,projMatrix,AXIS_ALIGNED) );
```

For multiple **voBrickSets,** it is convenient to know the maximum number of bricks in a collection. For example, the maximum number of bricks in a 256x256x128 volume represented by three stacks of two dimensional images is 256. this value can be used to allocate a single buffer for transient geometry that will be large enough to hold the results of polygonization regardless of the volume's orientation.

Allocating Brick Data

Creating a **voBrick** does not allocate voxel data storage. Applications must call the **voAppearanceActions** method, **dataAlloc()**, defined as follows, before operating on any data.

```
static void* dataAlloc(voTexture3D* aTexture3D);

static void* dataAlloc(voBrickSet* brickSet);
```

The following code allocates data for all of the bricks in a **voBrickSet**:

```
voBrickSetIterator aBrickSetIter(aBrickSet);
for(voBrick *brick; brick = aBrickSetIter(); )
    (void)voTexture3DActions::dataAlloc(brick);
```

dataAlloc() returns a pointer to the voxel data storage. This pointer can be used by applications to directly reference voxel data.

voTexture3D::**getDataPtr()** returns the same pointer but only after the brick has been created.

An application can use this pointer as a destination for any application-specific, disk I/O routines and thereby avoid allocation of additional buffer space, for example,

```
voBrickSetIterator aBrickSetIter(aBrickSet);
for(voBrick *brick; brick = aBrickSetIter(); )
    {
    unsigned char *vdata = (unsigned char *)
    voAppearanceActions::dataAlloc(brick);
    myReadBrickIfl(fileName, vdata, xBrickOrigin, yBrickOrigin,
        zBrickOrigin, xBrickSize, yBrickSize, zBrickSize);
    }
```

myReadBrickIfl() is a utility function provided with the API that reads a a block of data from (*xBrickOrigin*, *yBrickOrigin*, *zBrickOrigin*) to (*xBrickOrigin*+*xBrickSize*-1, *xBrickOrigin*+*xBrickSize*-1, *xBrickOrigin*+*xBrickSize*-1) from a three dimensional TIFF file using the Image Formal Library (IFL).

Loading Brick Data

Applications need to load the voxel data into the texture memory before the volume that is associated with this data can be drawn. For multi-brick volumes, each brick has to be loaded in turn once per frame. However, for single-brick volumes it is enough to load the voxel data once. Applications can keep track of such situations and use **voAppearanceActions::textureLoad()** to explicitly transfer the voxel data from the host to texture memory.

Once the volume is optimized with TEXTURE_OBJECTS as a parameter, the application can use **voAppearanceActions::textureBind()**. This action uses the OpenGL Texture Manager to determine if the requested brick data already resides in texture memory. If it does, **textureBind()** does nothing; otherwise, **textureLoad()** is called, which downloads the texture from host to texture memory.

```
static int voAppearanceActions::textureBind(voTexture3D* aTexture3D);
```

Using **textureBind()** instead of **textureLoad()** results in a significant performance boost when the brick data is already in texture memory.

Applications must call **textureBind()** or **textureLoad()** before drawing any shapes.

Forcing Download of Texture Data

voAppearanceActions::textureLoad(), invoked as follows, forces a download of texture data associated with a brick from the host memory into the texture memory regardless of whether or not the texture is already resident in memory.

```
static int voAppearanceActions::textureLoad(voTexture3D* aTexture3D);
```

Creating Custom Loaders

File I/O is not a part of the OpenGL Volumizer API. The sample applications in this section illustrate how to read voxel data from three-dimensional TIFF files using the IFL library, and from files that are stored as raw, two-dimensional images.

If your voxel data set is in a format that can be loaded by one of the demonstration OpenGL Volumizer programs, you can use the utilities in this section to load your two- or three-dimensional data set, as demonstrated in "Read Brick Data from Disk" on page 40.

Loading Three-Dimensional Data

To load three-dimensional, TIFF files and to determine the size of the volumes they contain, use the following two methods in *InitAppearance.cxx*:

```
int myGetVolumeSizesIfl(
    char *fileName, int &xSize, int &ySize, int &zSize,
    voExternalFormatType & diskDataFormat,
    voDataType & dataType);

int myReadBrickIfl(char *fileName, void *data,
    int xBrickOrigin, int yBrickOrigin, int zBrickOrigin,
    int xBrickSize, int yBrickSize, int zBrickSize,
    int xVolumeSize, int yVolumeSize, int zVolumeSize);
```

myGetVolumeSizesIfl() determines the volume sizes, format (for example, INTENSITY or RGBA) and data type (for example, ubyte or float).

myReadBrickIfl() reads a brick of voxels with their origin at (*xBrickOrigin*, *yBrickOrigin*, *zBrickOrigin*) and sizes (*xBrickSize*, *yBrickSize*, *zBrickSize*). See *myBrickIO.cxx* for the complete source code.

Loading Two-Dimensional Data

If your data is stored on disk as a sequence of 2D raw images, use the following method:

```
int myReadBrickRaw(char **fileNames, void *data,
    int xBrickOrigin, int yBrickOrigin, int zBrickOrigin,
    int xBrickSize, int yBrickSize, int zBrickSize,
    int xVolumeSize, int yVolumeSize, int zVolumeSize,
    int headeLength, int bytesPerVoxel);
```

By a “raw” image we understand a verbatim voxel stream in row-major order possibly proceeded by a fixed size header.

The file information, header length, dimensions, format, and data types, should be provided by the application, for example, read from the command line or read from the proprietary header.

Unsupported File formats

Applications that prefer to perform their own file I/O must provide their own API equivalents for **myReadBrickIfl()** or **myReadBrickRaw()** that read an arbitrary block of voxels from (*xBrickOrigin*, *yBrickOrigin*, *zBrickOrigin*) to (*xBrickOrigin*+*xBrickSize*-1, *xBrickOrigin*+*xBrickSize*-1, *xBrickOrigin*+*xBrickSize*-1) into a contiguous memory area. should replace.

File Format Utilities

OpenGL Volumizer provides a variety of voxel manipulation utilities in the *util* directory.

Converting Voxel Data to the TIFF Format

iflfrombin.cxx illustrates how to convert a raw image into a two-dimensional TIFF file (or any other two dimensional image recognizable by IFL). The syntax is:

```
% iflfrombin hdrlen xsize ysize csize bpp infile outimage
```

hdrlen is the length of the image file’s header in bytes.

xsize and *ysize* are file dimensions, csize is the number of channels (1 for INTENSITY, 4 for RGBA data).

bpp, the data type, is uchar, ubyte, or float.

infile and *outimage* are the input file and the TIFF file, respectively. The TIFF file should have an extension of *.tif* by convention.

Merging a List of Two-Dimensional Files Into One Three-Dimensional File

A number of two-dimensional images representing a volume can be combined into a single, three-dimensional TIFF file using the following command in *iflto3Dtiff.cxx*:

`% iflto3Dtiff` *inFileName1 inFileName2*`...` *outFileName*

Test Volumes

To facilitate a simple and repeatable testing environment, a utility for generating a volumetric test pattern is provided in *Volumizer/util/mkcubes.cxx*. Invoking this program with the following parameters produces a volume in a three-dimensional TIFF format:

```
mkcubes xVolumeSize yVolumeSize zVolumeSize cubeSize fileName.tif
```

The volume has dimensions specified on the command line and it contains a pattern of small cubes of size, *cubeSize,* with intensity varying linearly from their centers to their surface. For example, use:

```
mkcubes 256 256 256 128 out.tiff
```

to produce a 256^3 volume filled with 8 cubes each having 128 voxels on the side.

Chapter 8

Customized Volume Drawing

Drawing a volume can be as simple as calling **voGeometryActions::draw()**. This method renders polygons, called **voIndexedFaceSets**, each of which is a slice of a volume. The slices are often planar, but they also can be arbitrary surfaces. Each of these slices, clipped to volume and brick boundaries, can then be drawn.

Applications that choose to use unconventional array types or provide user-defined per-vertex data have to implement their own draw method.

This chapter provides a procedure for implementing a custom draw method in the following sections:

- "Customized Volume Drawing Procedure" on page 150
- "clip()" on page 152
- "Drawing Brick Set Collections" on page 154

Customized Volume Drawing Procedure

Use the following procedure to create a custom draw method. For each frame, your draw method must perform the following tasks.

1. Polygonize a volume using **voGeometryActions::sample()**.

   This method slices a volume's geometry with a plane to create a set of slices, as shown in Figure 8-1.

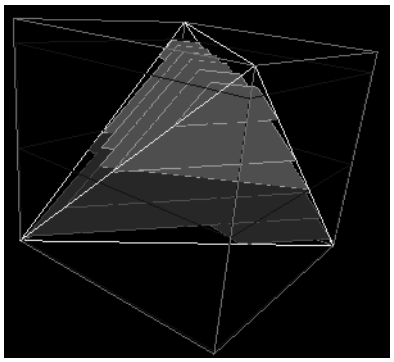

Figure 8-1 Polygonization of a Single Tetrahedron.

   sample() returns the vertices of all the polygon slices. The orientation of the planes can be specified using one of the following flags:

   - VIEWPORT_ALIGNED
   - AXIS_ALIGNED
   - SPHERICAL

   voSamplingPlaneSet describes the set of planes and can be used to iterate through them to draw the volume.

   If you want the sampling surfaces not to be planar, you can implement your own, application-specific **voSamplingSurfaceSet**. For more information, see "General Clipping" on page 152.

2. Once the sampling surfaces are clipped to volume's geometry, the resulting polygons need to be further clipped to brick boundaries to facilitate state sorting. You clip polygons to brick boundaries using **voGeometryActions::clip()**.

   For more information about this method, see "clip()" on page 152.

3. Sort the bricks from back to front using **voGeometryActions::voSortAction**.

   Given a brick set, **voSortAction** produces a sorted array of indices that can be accessed in sorted order. For example, the following code will visit each brick in visibility sorted order:

```
voSortAction aSortAction(aVolume->getCurrentBrickSet(),
modelMatrix, projMatrix);

   for(brickNo=0;brickNo<BrickCount;brickNo++) {
     int brickSortedNo = aSortAction[brickNo];
       voBrick *aBrick =
       aVolume->currentBrickSet->getBrick(brickSortedNo);
     ...
   }
```

4. Process bricks individually in a depth-sorted order. Load the current brick of texture using **voAppearanceActions::textureLoad()** or **voAppearanceActions::textureBind()**.

5. Draw all the polygons with texture using the vertices returned from **voGeometryActions::clip()**.

6. Repeat steps four and five with each successive brick until the volume is fully rendered.

   Some applications, may want to compute the area of a **voIndexedFaceSet** after it was projected onto the screen. This may be required for the sake of fill rate computation, for example. If you need to determine the area of a face in the set, use **voGeometryActions:: findProjectedArea()**. Please, note, that this is a rather heavy weight routine and should be used only sparingly.

clip()

voGeometryActions::cpolygonize() calls **voGeometryActions::clip()**, defined as follows, to clip a single polygon computed by **voSampleAction()** to brick boundaries, specified by its lower-left and upper-right vertices:

```
static int clip (voVertexData* vertexData,
    voInterleavedArrayType interleavedArrayFormat,
    float lowerLeft[3], float upperRight[3], voIndexedFaceSet* cSet);
```

The arguments specify the coordinates of the polygon's vertices and possibly other per-vertex attributes. The input format is described by **voVertexData** in "Drawing Brick Set Collections" on page 154. The clipped polygon ends up as a **voIndexedFaceSet**, which then can be drawn using **voGeometryActions::draw()**.

General Clipping

An overloaded version of **voAppearanceActions::clip()** clips a polygon to a convex polyhedron determined by set of positive half-spaces:

```
static int clip(voVertexData* vertexData, voInterleavedArrayType
   interleavedArrayFormat,float* planes,int planeCount,
   voIndexedFaceSet* cSet);
```

The convex polyhedron is specified by a set of positive half planes given by their equations {A, B, C, D]. The input array specifies the coordinates of the polygon's vertices and possibly other per-vertex attributes. The input format is described by **voVertexData**. The clipped polygon ends up as a **voIndexedFaceSet**.

This action allows applications to implement alternative ways of sampling and clipping. For example, applications may want to implement a box volumetric primitive by directly sampling a cube without tessellating it into tetrahedra. In this case, the default instance of **voAppearanceActions::sample()** can be replaced with a call to **clip()**.

Similarly, for primitives that are convex polyhedra (for example, tetrahedra) it is possible to collapse **sample()** and **clip()** into a single routine. After all, "slicing" a tetrahedron with a plane followed by clipping the resulting polygon to a box, is equivalent to clipping the plane to the convex region determined by the ten positive half-planes defining the region of intersection between the tetrahedron and the box.

Other Applications

clip() is a general purpose utility and may be used by applications to accomplish general-purpose clipping to a clip box. (For more information about clip boxes, see “Clip Boxes” on page 28.)

For example, it is common in many diagnostic applications to display an arbitrarily oriented plane through the data set that may not necessarily coincide with the original acquisition direction (this task is referred to as Multi-Planar Reformation (MPR). For more information about MPR, see “Polygonizing Arbitrarily Oriented Cross Sections” on page 118.

Similarly, in seismic data interpretation it is desired to visualize data along an arbitrarily shaped surface (e.g., geological horizon). In both of these situations, the planes/polygons need to be clipped to the brick boundaries so that the resulting polygons can be texture mapped correctly.

Consider a situation where an MPR along a plane described by [A, B, C, D] is required. One can, apply the **sample**() in order to clip the plane to each individual tetrahedron and feed the results into **clip**() to clip them to each brick.

```
float planeEquation[4] = { A, B, C, D,};
float tetraVerts[4][3] = { ... };  // vertices of the tetrahedron
float brickLowerLeft[3]  = { 0.5, 0.5, 0.5 };
float brickUpperRight[3] = { 254.5, 254.5, 254.5 };
voVertexData vData(4,valuesPerVertex); // intermediate polygon verts

sample(tetraVerts,planeEquation,&polygonVerts);
clip(&polygonVerts,brickLowerLeft, brickUpperRight, aFaceSet);
```

Given a brick, one should clip to the clip box rather than the brick boundary to avoid seams:

```
voBrick *aBrick;

aBrick->getClipBrickSizes(brickLowerLeft, brickUpperRight);
```

Note: For applications that deal exclusively with canonical volumes, where the geometry coincides with appearance, it is not necessary to clip to tetrahedra boundaries. For example, the polygons that comprise the geological horizon can be fed directly to **clip**().

Drawing Brick Set Collections

In a **voBrickSetCollection**, you want to draw the copy of the volume that has the least sampling artifacts.

voGeometryActions::findClosestAxisIndex() identifies the orientation (XY, XZ, or YZ) that minimizes sampling artifacts under some sampling modes. The result is generally used in conjunction with the **voBrickSetCollection** to select the most suitable copy of a volume for the current view:

voGeometryActions::findClosestAxisIndex(), defined as follows, returns a **voPlaneOrientation**.

```
static voPlaneOrientation findClosestAxisIndex(
    GLdouble modelMatrix[16], GLdouble projMatrix[16],
    int samplingMode);
```

modelMatrix[] and projMatrix[] are the ModelView and Projection matrices that can be easily obtained from OpenGL state:

```
glGetDoublev(GL_MODELVIEW_MATRIX, modelMatrix);
glGetDoublev(GL_PROJECTION_MATRIX, projMatrix);
```

samplingMode can have several values:

- **voSamplingModeScope**::VIEWPORT_ALIGNED, in which the planes are parallel to the viewport, that is, they change with every view.
- **voSamplingModeScope**::AXIS_ALIGNED, in which the planes are aligned with the axis that is most parallel to the line of sight.

The return value is one of the following:

```
enum voPlaneOrientation{
    UNSPECIFIED,
    XY,
    XZ,
    YZ,
};
```

The value is the axis that is most aligned parallel either to the line-of-sight vector or the viewport, depending on *samplingMode*. For proper 3D volumes (i.e., none of the brick dimensions is 1) the value of UNSPECIFIED is returned.

Example Use of findClosestAxisIndex()

Example 8-1 shows how to use **findClosestAxisIndex()**.

Example 8-1 findClosestAxisIndex() Example

```
if( aVolume->interpolationType == voInterpolationTypeScope::_2D)
    aVolume->setCurrentBrickSet(
        findClosestAxisIndex(
            modelMatrix,projMatrix,
            voSamplingModeScope::AXIS_ALIGNED) );

draw()
// voGeometryActions::draw() can be applied to a voFaceSet to render
it into the // frame buffer:

void
draw(
    voIndexedFaceSet *aFaceSet,
    GLenum mode,  // GL_LINE or GL_FILL
    voInterleavedArrayType interleavedArrayFormat);
```

interleavedArrayFormat is one of the interleaved array types described for **voVertexData** in “Read Brick Data from Disk” on page 40.

Appendix A

Volume Rendering Examples

A number of OpenGL Volumizer application examples are included in */usr/share/Volumizer/demos/Volumizer.* They include an extensive set of OpenGL, Open Inventor, and IRIS Performer samples. In addition, a set of utilities for voxel data manipulation is provided in */usr/share/Volumizer/util* as described earlier.

It should be noted that all the examples were intended as an illustration of a specific feature of the API and are thus kept rather simple. No full-featured viewer is provided at this time.

OpenGL Examples

There are a number of examples provided here that illustrate the use of Volumizer in the framework of an OpenGL application. They all share a common structure and much of the source. For example, most of the examples share the same main() routine (to be found in *voglMain.cxx*). This is done in order to emphasize the similarities and differences, to facilitate code reuse, and to provide consistent user interface for all examples. The next section discusses the common structural elements.

All the programs define five functions: **my_InitAppearance()**, **my_InitGeometry()**, **my_InitGfx()**, **my_DrawVolume()**, and **my_Cleanup()**. The contents of these functions vary from application to application, but their purpose remains the same:

- **my_InitAppearance()** parses the command line arguments for input file name, selects optimal parameters for representing voxel data, creates an instance of **voBrickSetCollection**, reads and converts the data. It is called from main().
- **my_InitGeometry()** creates an instance of **voIndexedTetraSet** to represent the region of interest of the volumetric object to be rendered. It also allocates storage for transient geometry (**voIndexedFaceSet**) produced during the polygonization stage. It is called from main().
- **my_InitGfx()** performs certain types of initialization and optimization on the voxel data (e.g., creation of texture objects) that require the graphics context to be present. This is also where the lookup tables get initialized. It is called once the connection to the server is established (e.g., from the Expose callback).
- **my_DrawVolume()** is called for every frame to polygonize the volumetric shape and draw the resulting geometry.
- **my_Cleanup()** is called once the application is done with the volume and wants to reclaim the system resources (e.g., memory, texture objects, etc.).

For example, *voglSimpleMain.cxx* and *voglSimpleVolume.cxx* illustrate the above routines in action in a minimalist fashion. This is the smallest self-contained example. However, the resulting application, *voglSimple*, has very limited capabilities.

All the remaining examples share a substantial amount of code. In particular, they share much of the implementation of the user interface. For example, dragging the left mouse button causes all applications to rotate the scene around its center. Similarly, middle mouse button will modify the current transfer function using the cursor position to determine the center and width of a grayscale linear ramp (-lutFile on the command line disables this feature, as described below).

All the applications, except as noted below, take a single command line argument, which is the name of the 3D TIFF file containing the voxel data. Sample data sets in this format are provided in *Volumizer/data/volumes/*. Look in *Volumizer/data/volumes/*/README* for detailed format description. *voglRaw* takes as argument a list of file names of a sequence of 2D images stored as raw voxel streams. A sample data set in this common format is provided in *Volumizer/data/volumes/*/raw.*

It is also possible to generate a simple 3D test pattern consisting of an array of cubes with varying density. *Volumizer/util/mkcubes* was provided for this purpose. It creates output in form of a 3D TIFF file. For example:

```
mkcubes 128 128 128 32
```

will create a volume 128 units on the side filled with a series of 8 cubes (each 32 units on a side) of varying density.

All sample applications take the same set of command line options. These are:

- -2D -- render using 2D textures (default is 3D if supported)
- -3D -- render using 3D textures
- -color -- render in RGBA mode (default is monochrome)
- -dataRange %d %d --specify the dynamic range of your data
- -lut %d %d -- generate a grayscale linear ramp with center and width
- -lutFile %s -- read in the transfer function from a file
- -monochrome -- render in grayscale mode (saves memory and bandwidth)
- -tessellationSize %d -- for *voglSpaceLeap* the size of the adaptive subdivision

The options can come anywhere on the command line and can be abbreviated (ambiguity resolved in lexicographic order, e.g., -lu expands to -lut not -lutFile). Individual programs are discussed below. All programs are executed with the following command:

```
program volume.tif
```

unless indicated otherwise. Consult the *README* file in the demo directory for specific instructions and options use.

voglBasic

voglBasic is a version of *voglSimple* extended to use a lookup table, parse the command line arguments, and select different options. Typical invocations include:

```
voglBasic volume.tif
voglBasic volume.tif -2D
voglBasic volume.tif -color
voglBasic volume.tif -color -lutFile lookup.table
```

Sample lookup tables are provided in *Volumizer/data/tables*.

voglCache

voglCache replaces the contents of the basic **my_DrawVolume()** modifying it to use transient geometry caching from frame to frame in order to amortize the cost of polygonization. In addition, it uses a call to **voGeometryAction::drawUtility()** which is a high level utility function that will try to draw the volumetric object efficiently. While this utility simplifies the volume rendering code even further it has one drawback: **drawUtility()** does not have precise semantics associated with it and may change in future releases. Applications that rely on the specifics of the rendering process may want to exercise more control by adopting the more explicit version provided by *voglBasic*.

voglRaw

This application provides functionality identical to that of *voglCache*. However, it accepts input from a list of files of raw 2D images, instead of a single 3D tiff file. By a "raw" 2D file, we mean a voxel stream in row-major order possibly proceeded with a fixed length header. File names, image sizes, header length, and voxel data type are specified on the command line:

```
voglRaw 0 256 256 128 ushort -dataRange 0 4095 -lut 135 155 vol*.ima
```

voglSpaceLeap

All the previous examples use a single cube matching the voxel array in size as geometry. *voglSpaceLeap* replaces the content of **my_InitGeometry()** to tessellate the volume into a large number of small sub-cubes. The sub-cubes that are empty (outside of the data range of interest) are discarded, and the adjacent non-empty sub-cubes are coalesced to reduce the number of tetrahedra in the scene (each cube is still represented as 5 tetrahedra). The size of the sub-cubes and the voxel range are specified on the command line:

```
voglSpaceLeap volume.tif -tess  8 40 255
```

The results are shown below.

voglMorph

All the previous examples specified the vertex data in **V3F** format, thus implicitly assuming that the voxel and vertex coordinates be identical. *voglMorph* replaces the contents of **my_InitGeometry()** to explicitly specify the voxel coordinates on per-vertex basis. If the mapping between the vertex and voxel coordinates is not an identity, the rendered volume appears warped. Consult the figures below.

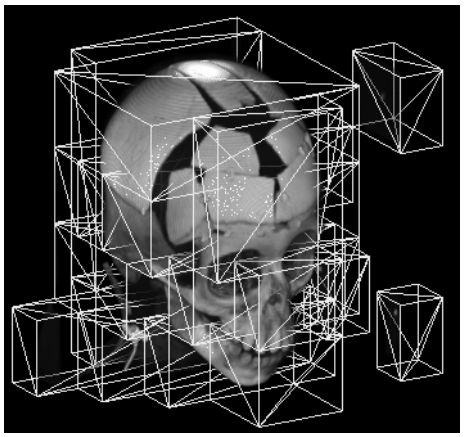

Figure A-1 voglSpaceLeap: Space Leaping Example

voglSphere

voglSphere replaces the content of **my_InitGeometry()** to define a spherical region of interest. Extended volumetric primitives are used for that purpose.

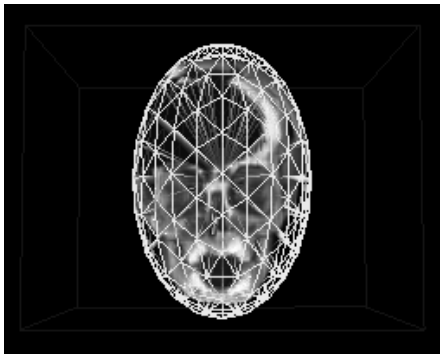

Figure A-2 voglSphere: A Spherical Region of Interest

voglPick

voglPick illustrates Volumizer's ability to pick individual voxels from within the volume. Press the right mouse button to select the line of sight through the cursor. A signal corresponding to the values of voxels along this line is drawn at the bottom of the screen. See the figure below.

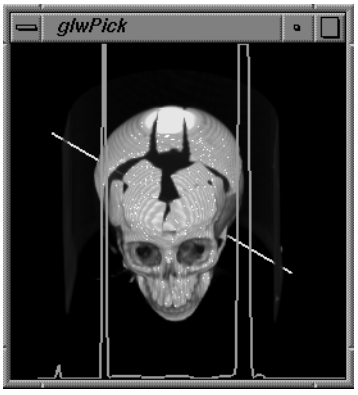

Figure A-3 voglPick: Voxel Picking Example

voglShade

voglShade modifies the **my_DrawVolume()** routine to apply per voxel shading using the tangent space method (Brian Cabral, SIGGRAPH97). Each slice is drawn twice using slightly different texture coordinates to effectively compute the component of the gradient along the light vector. Use the right mouse button to move the light vector around.

voglUnstructured

voglUnstructured virtually eliminates **my_InitAppearance()** in order to render unstructured grids: a tetrahedral tessellation with per-vertex colors but no texture associated with it.

voglMirror

voglMirror illustrates multi-pass rendering of a heterogeneous scene containing both surface and volume objects in order to accomplish special effects such as reflection and refraction. The scene is rendered twice, once from the imaginary and again from the actual eye position and the results are combined. Use the left mouse button to tilt the scene and the middle button to move the sphere around.

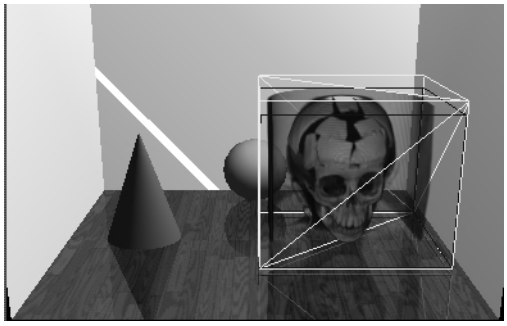

Figure A-4 voglMirror: A Muti-pass Reflection Algorithm on a Heterogeneous Scene

voglMPR1 and voglMPR2

These two programs illustrate drawing of Multi-Planar Reconstructions (MPRs) or cross sectional images on stacks of 2D images. This code is useful for efficiently computing oblique slices through volumes on machines that do not support 3D texture mapping efficiently. *voglMPR1* slices the volume with a plane, while *voglMPR2* with a couple of polygons.

Open Inventor Examples

A set of examples that use Open Inventor as the underlying software platform is also provided. Their main purpose is to illustrate how to integrate the functionality provided by Volumizer with the retained mode framework of Open Inventor. Note that this is not an endorsement of Open Inventor as a viable development environment for new applications; these examples are provided solely to illuminate Volumizer to existing and past Open Inventor users.

OpenGL Volumizer operates in the immediate mode characteristic of OpenGL. In order to take advantage of its volume rendering capabilities in an Open Inventor environment we have to extend the core set of classes and traversal methods provided by Inventor.

In the simplest case, we have to implement two classes to represent a volumetric object: one to represent geometry, the other, appearance. These are **voivGeometry** and **voivAppearance** respectively. **voivGeometry** will be subclassed from **soShape**, and **voivAppearance** from **soNode,** becoming a part of the current state. In addition, we will also need **voivAppearanceElement** which will be derived from **SoReplacedElement** and which will be used to get the volume appearance from the current state.

voivViewer is an application that uses these two classes to implement a simple volume viewer. It is essentially identical to *ivview* (it is possible to build a shared library from the objects provided and convince *ivview* to load them as well) capable of displaying volumetric objects. *voivViewer* takes a single command line argument that is a file in an Open Inventor format. Sample input files were provided in /usr/share/*Volumizer/data/Inventor*. For example:

```
voivViewer Simple.IV
```

voivFreeFormDeform is another application that builds on the Open Inventor classes previously described. It illustrates Volumizer's ability to clip the volumetric model to arbitrarily shaped surfaces and to accomplish free form deformations of volumes. In order to grab on a vertex press ESCAPE key or click on the "arrow" button in the upper part of the right panel on the viewer.

In the first mode of operation, *voivViewer* takes as input the same input file as *voivViewer* (i.e., V3F vertex format):

```
voivFreeFormDeform Simple.IV
```

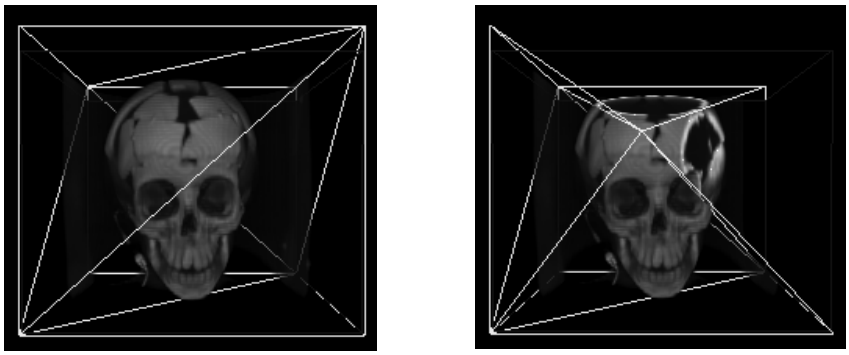

Figure A-5 Clipping Volumes to Arbitrary Surfaces

Grab one of the vertices outlining the volume's geometry (drawn as a yellow wireframe) and drag it around. If you move the vertex towards the center, the surfaces of the distorted geometry will act as clipping planes as shown above.

In the second example, the input volume uses T3F_V3F per-vertex data:

```
voivFreeFormDeform UniformT3F.IV
```

Dragging a vertex will cause the volume to morph:

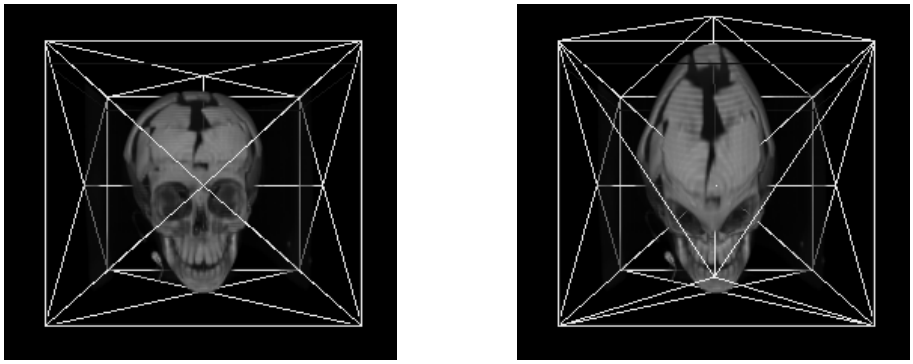

Figure A-6 Free-form Deformation of Volumes

IRIS Performer Example

A sample implementation of a Performer volume node is also provided. The executable takes a single command line argument that is the path to a 3D TIFF file containing the voxel data.

```
perfvol volume.tif
```

Use the left, middle, and right mouse buttons to translate, rotate, and scale the model respectively.

Glossary

brick

is a set of voxels that is small enough to be cached by underlying hardware, generally texture memory.

brick set

is composed of one or more adjacent bricks.

brick set collection

is one or more brick sets for applications and platforms that require several copies of the same volume with possibly different memory layouts, for example, XY, XZ, and YZ stacks of slices.

caching

maintains objects in anticipation for future reuse; transient geometry can be cached from frame to frame in order to amortize polygonization cost.

clip box

is an abstract construct associated with overlapping bricks. If two bricks overlap, all geometry, for example, sampling surfaces, has to be clipped to each brick's clip box to avoid seaming.

data type

is voxel's numeric representation, for example, unsigned byte, or float.

external format

is an application side voxel format, for example, INTENSITY.

faces

are polygons resulting from polygonization. Each face is a slice of a volume and may consist of a number of polygons.

internal format

is a voxel format internal to the graphics adaptor, for example, INTENSITY8_EXT.

model (or shape)

is a combination of geometry and appearance attributes; for example a triangle with per-vertex color, or a tetrahedron with 3D textures mapped on it; it is the graphical object that appears in the scene.

multi-planar reformation

is a cross section through a volume

native (or disk) format

is a format of the voxels as they are store on the disk, for example, INTENSITY.

polygonization

is the process of slicing a volumetric geometry along with a family of surfaces. The result is a set of polygons, each with a different texture. These polygons are used to display the volume and can be passed to other rendering toolkits, such as IRIS Performer.

ray casting

In this approach, a ray is traversed from the viewer's eye trough each drawable pixel with colors and opacities accumulated along the way. Can proceed in ray-order or sampling-distance order.

slicing

is a process of sampling of a solid model with a family of surfaces in order to polygonize it. Sampling-distance order ray casting.

tessellation

subdividing a shape into primitives. For example, a concave polygon can be tessellated into triangles, a solid torus into tetrahedra.

transient geometry

is face sets resulting from polygonizations; these are often discarded after each frame.

volume

is a pair of geometry and appearance defining a volumetric model.

volume roaming

displaying an interactively manipulated volume of interest that is substantially smaller than the whole volume.

volumetric (or solid) model

is defined as the combination of geometry (for example, a sphere) and appearance (for example, the voxels representing your data) to represent a solid (i.e., not hollow) object.

volumetric primitive

is a building block used in modeling of solid shapes. For example: tetrahedron, prism, pyramid, hexahedron, cylinder, cone, sphere; primitives are used to build more complex shapes, for example, sub-parts of a volume, region of interest, or a deformable grid.

voxel coordinates

is a reference frame for specifying per-vertex properties, for example, **T3F_V3F** T3F refers to voxel reference frame. Voxel coordinates map piecewise linearly onto the per-brick texture coordinates.

voxel

is a value in a three-dimensional array.

Index

Symbols

_SAMPLE_OVER_OBJECT, 79
_SAMPLING_AXIS_ALIGNED, 79

Numbers

2D and 3D mixing, 114
2D to 3D, changing, 148

A

allocate, voxel data, 144
allocate memory, 40, 72
API, 3
API, common classes, 33
API, functional categories, 94
API, hierarchy, 93
appearance, 12, 70, 84, 129
appendPoly(), 107
application, template, 34
array, 104
axis, find closest, 155
AXIS_ALIGNED, 53, 154

B

bilinear, 71, 74
blending function, 57
book, structure, xiv
boundFaceCount(), 76
brick, 25
brick, collection, 72
brick, coordinates, 138
brick, count, 74
brick, deleting, 82
brick, determine number of, 74
brick, interleave, 44
brick, load, 40
brick, loading, 151
brick, optimize, 44
brick, overlap, 27
brick, scale, 42
brick, size restrictions, 29
brick, sorting, 151
brick set, 25, 143
brick set collection, 26, 39, 143
button input, 88

C

callback, 92
camera, 127

classes, description of, 96
classes, Volumizer, 94
clean up, 65
clip(), 152
clip()., 151
clip box, 27, 28
clip planes, 151
clip texture, 137
collection, brick set, 26
color, 59
conventions, xvi
convert data, 43
converting 2D to 3D, 148
coordinates, 84, 102
coordinates, brick, 138
count, brick, 74
cracking, 25
cube, 45

D

data, convert, 43
data, delete, 40
data, multiple copies, 44
data, optimize, 44
data, scale, 42
data, scaling, 134
dataAlloc(), 40, 72, 144
dataConvert(), 43, 134
data file, handles large, 5
data format, 131
dataFree(), 40, 65, 82
dataScaleRange(), 42, 134
data structures, 102
data types, 130
deallocate, 65
deallocate, face set, 108
debugging, 64
declarations, 36
deformation, 14
delete, 40, 65
destructor, for face sets, 108
disk, load from, 40
display, 91
documentation, xv
draw, 50
draw(), 78, 81, 115, 149
draw, customized, 149
drawing outline of brick, 142

E

error, number, 63, 64
error, string, 63
error handler, 63
errors, handling, 63
external format, 132
eye position, 127

F

face, 49
faces, 47
face set, 107
face set, deallocate, 108
file format utilities, 147
file name of a volume, 71
findCameraPosition(), 127
findClosestAxisIndex(), 154, 155
findClosestAxisIndex()), 78

findProjectedArea(), 151
findViewDirection(), 127
format, convert, 42
format, external, 132
format, internal, 133
format, optimal, 133
format, table, 59
format, utilities, 147
format, voxels, 70
free data, 40
freeing data, 65

G

geometry, 12, 70, 84, 101
geometry, create, 75
geometry, defining, 45
geometry, delete, 81
geometry, drawing, 50
geometry, roaming, 124
geometry, storage, 47
getBestParameters(), 37, 62, 71, 142
getBrickCount(), 74
getCurrentBrickSet(), 79
getDataPtr(), 144
getErrorNumber(), 63
getErrorString(), 63
getInterpolationType(), 79
GL_MODELVIEW_MATRIX, 78
GL_PROJECTION_MATRIX, 78
GL_TEXTURE_COLOR_TABLE_SGI, 60
glBlendFunc(), 122
glwSimpleMain.cxx, 83
graphics intialization, 85
graphics state, 57

H

header files, 36
hexahedra, 25
higher-level, primitives, 18
higher level primitives, 113

I

iflto3Dtiff, 148
includes, 69
index, 47
indices, 104
interate, face sets, 108
interleave, advantages, 135
interleave, manually, 136
interleave bricks, 44
intermix 2D and 3D, 31
internal format, 133
interpolation, 71, 74
interpolationType, 57
intialize internal data structures, 86
Inventor, 6
IRIS Performer, 6, 31
iterate, 72

K

keyboard input, 88

L

load brick, 40
loaders, custom, 146
lookup, post, 60

lookup, pre, 60
lookup table, 59
low-level, 6

M

main(), 91
memory, 72
memory, allocation, 40
memory, deallocating, 65
memory, texture, 117
mixing 2D and 3D, 114
model, hierarchical, 15
modelMatrix, 51
modelview, 78
MPR, 121, 153
multi-pipe, 7
Multi-planar Reformatting, 121
multiple copies, 44
multiple copies of data, 44
myGetVolumeSizesIfl(), 146
myReadBrickIfl(), 40, 144, 146

N

nextPrimitive(), 105
nextVertex(), 105
NO_ERROR, 63

O

OBJECT, 55
opacity, 21, 59
opaque, 120
opaque geometry, 78
OpenGL, 6, 8
OpenGL Optimizer, 6, 31
OpenGL Volumizer, 3
Open Inventor, 6
optimal format, 133
optimize, 44, 77
optimize, brick size, 71
optimize, brick sizes, 142
Optimizer, 6, 31
orientation, 154
orthographic projection, 20
outline of brick, brick, outline, 142
overlap, 27
overlapping, 117
overlapping volumes, 116
overview, of Volumizer, 4

P

PARC, 16
Performer, 6, 31
per-part appearance, appearance, per part, 15
perspective projection, 20
pick(), 126
picking, 125
platform support, 6
polygon, 30
polygon, append, 107
polygonization, 30
polygonize(), 50, 79, 118
polygonizeMPR(), 121
polygons, 115, 118
post-interpolation lookup, 60
pre-allocated array, 104, 107

pre-interpolation lookup, 60
primitive, complex, 25
primitive, higher level, 18
primitive, tetrahedron, 17
primitives, 10
primitives, higher level, 113
programming algorithm, 35
programming template, 33, 35
projection matrix, 78
projMatrix, 51

R

ray casting, 20
ray order, 22
ray-order, 20
read brick data, 40
reference frame, 138
rendering, slicing, 20
renderingMode, 37
RGBA, 59
RGBA8_EXT, 59
roaming, 124

S

sample(), 150, 153
sampling, 56
sampling mode, 52
sampling plane, 81
sampling-surface order, 20
scale, voxel values, 42
scale data, 42, 134
selecting, 125
setBrickSizes(), 141
setDebugLevel(), 64
setErrorHandler(), 63
setErrorNumber(), 64
shading, 21, 122
shape, 12
shape, drawn, 50
shear warp, 20
Silicon Graphics APIs, 8
size, 70, 73
size, brick, 141
size, bricks, 29
size, optimal, 142
slicing, 20
sort bricks, 151
sorting, 56
space leaping, 16
SPHERICAL, 53
splatting, 20
state, 56, 57
storage, for transient geometry, 47
storage, free, 65
storage, freeing, 82
surface-only models, 11

T

table, format, 59
tagged voxels, 137
template, 33, 35
template, application, 34
tesselation, minimal, 23
tessellation, fine, 16
tetrahedra, 45
tetrahedra, indexed, 109
tetrahedron, as primitive, 17

tetraset, indexed, 47
texgen, 57
texgen(), 80
texgenDisable(), 81
texgenEnable(), 140
texgenSetEquation(), 81, 140
texture, 70, 78, 80
texture, memory, 81
texture, stack of 2D, 119
TEXTURE_OBJECTS, 136
textureBind(), 145
textureDisable(), 78, 81
textureEnable(), 57, 80
textureInterleave(), 136
textureLoad(), 145, 151
textureMakeMatrix(), 141
texture mapping, 119
texture memory, 117
texture objects, 136
texture paging, 30
TIFF files, 146
T-junction, 24
toolkits, 31
trackball, 88
transient storage, 76
translucent, 115
transpose, 41, 44, 74
traversal of geometries, 105
triangle, as primitive, 17
trilinear, 71
two-dimensional v. Volumizer, 9

U

user-defined data, 102
user-supplied data, data, user-supplied, 48

V

vertex, 104
vertex, data order, 103
vertex coordinates, 75
vertex parameters, 102
VIEW_VOLUME, 54
viewing frustum, 22
viewpoint, 127
VIEWPORT_ALIGNED, 52, 54, 154
VO_SAMPLING_AXIS_ALIGNED, 115
voAppearanceActions, 40, 57, 72, 96
voBrick, 96, 130
voBrickSet, 96
voBrickSetCollection, 39, 70, 75, 84, 96, 143
voBrickSetCollection, drawing, 154
voBrickSetCollectionIterator, 72, 96
voBrickSetIterator, 96
voBrickSets, 143
voCache, 98
voDataType, 38, 70, 130
voDataTypeScope, 38
voError, 63, 98
voErrorType, 63
voExternalFormatType, 39, 70, 132
voFaceSet, 106
voFaceSets, 51, 115
voGeometryActions, 97
VOI, 14
VOI, arbitrary shape, 111
voIndexedFaceSet, 48, 49, 50, 77, 97, 105, 106, 107
voIndexedFaceSetIterator, 97, 105, 108
voIndexedFaceSetPtr, 77
voIndexedSet, 97, 105
voIndexedSetIterator, 97, 105

voIndexedTetraSet, 45, 47, 75, 84, 97, 105, 109
voIndices, 97, 104
voInitAppearance, 70
voInterleavedArrayFormat, 126
voInternalFormatType, 38, 70, 133
voInterpolationType, 57, 71
voInterpolationTypeScope, 44, 78, 96
voLookupTable, 59, 98
voLookupTablePost, 98
voLookupTablePre, 98
volume, defining, 45
volume, deformation, 14
volume, drawing, 149
volume, multiple, 116
volume, properties, 11
volume, roaming, 124
volume, slicing, 20, 22
volumeMakeTransposed, 74
volumeMakeTransposed(), 41, 44
Volume of Interest, 124
volume of interest, 14
volumeOptimize(), 44, 77
Volumizer, 3
Volumizer, v. 2D APIs, 9
voOptimizeVolumeTypeScope, 96
voPartialBrickTypeScope, 96
voPlaneOrientation, 154
voPlaneOrientationScope, 97
voRenderingMode, 71
voRenderingModeScope, 98
voSamplingMode, 52
voSamplingModeScope, 154
voSamplingPlaneSet, 97, 150
voSamplingSpaceScope, 56
voSamplingSurfaceSet, 97
voSortAction, 98, 151
voTexture3D, 96, 130
voTextureLookupPost(), 60
voTextureLookupPre(), 60
voutIndexedHexaSet, 113
voutPerfMeter, 98
voutTimer, 98
voVertexData, 48, 97, 102, 104
voxel, coordinates, 140
voxel, delete, 40
voxel, scale values, 42
voxel, tagged, 137
voxel data. storage, 40
voxel values, 137

X

xfmVox2TexCoords(), 80, 139
X window, 92

Z

Z-buffer, 122
Z-buffering, 115

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3720-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389