sgi®

SGI® OpenGL Volumizer™ 2
Programmer's Guide

Version 2.8

OpenGL™

CONTRIBUTORS

Written by Ken Jones and Jenn McGee

Illustrated by Chrystie Danzer

Production by Karen Jacobson

Engineering contributions by Praveen Bhaniramka and Vaibhav Saxena

New Features in This Release

OpenGL Volumizer 2.8 provides the following enhancements:

- Scalability enhancements for Silicon Graphics Prism platforms
- SUSE LINUX support
- High-quality shading algorithms

Record of Revision

Version	Description
001	August 2001 Original publication.
002	November 2001 Updated for the 2.1 release of OpenGL Volumizer 2.
003	April 2002 Updated for the 2.2 release of OpenGL Volumizer 2.
004	June 2002 Updated for the 2.3 release of OpenGL Volumizer 2.
005	November 2002 Updated for the 2.4 release of OpenGL Volumizer 2.
006	July 2003 Updated for the 2.5 release of OpenGL Volumizer 2.
007	December 2003 Updated for the 2.6 release of OpenGL Volumizer 2.
008	July 2004 Updated for the 2.7 release of OpenGL Volumizer 2.
009	February 2005 Updated for the 2.8 release of OpenGL Volumizer 2.

Contents

Figures

Tables

About This Guide

This publication documents OpenGL Volumizer 2, a C++ volume rendering toolkit optimized for SGI scalable servers. It provides the developer with the tool set needed to solve the problems inherent in high-quality, interactive volume rendering of large datasets. This guide gives an introduction to the OpenGL Volumizer 2 application programming interface (API) and examples of its use.

Audience for this Guide

This guide is intended for C++ developers of volume rendering applications who understand the basic concepts of computer graphics programming.

Familiarity with OpenGL and programming interfaces is strongly recommended.

Related Publications

The following books might be helpful:

- *OpenGL Performer Programmer's Guide*
- *SGI OpenGL Multipipe SDK Programmer's Guide*

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.

- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.

- You can also view man pages by typing `man` *<title>* on a command line.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
function	This bold font indicates a function or method name. Parentheses are also appended to the name.
variable	Italic typeface denotes variable entries and words or concepts being defined.
user input	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
`manpage(x)`	Man page section identifiers appear in parentheses after man page names.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the

manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library webpage:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, CA 94043-1351

SGI values your comments and will respond to them promptly.

Overview

This overview consists of the following sections:

- "What Is OpenGL Volumizer 2?"
- "Product Components"
- "Supported Platforms"

What Is OpenGL Volumizer 2?

As the volume of information produced by instrumentation devices and simulation increases in size, complexity, and level of detail, so does the need for better, more powerful interpretation tools. In particular, the requirements for volume-data interpretation software keeps expanding. Utilizing various computational techniques (such as marching cubes, segmentation, region growing, isosurface extraction, flow streamlines, and flow volumes) and visualization techniques (such as 3D texture mapping, ray casting, projected tetrahedra), users demand more interactivity and immersion capabilities with their large volumetric datasets.

To help application programmers answer these needs, SGI has developed OpenGL Volumizer 2, a software development kit that provides a simple interface to complex hardware features available through OpenGL.

OpenGL Volumizer 2 is a new design that allows better incorporation and sharing of capabilities across the SGI application programming interfaces (APIs) and facilitates management of extremely large volumetric datasets (See Chapter 6, "The Large-Data API: 3D Clip Textures" for detailed information). OpenGL Volumizer 2 provides a supported pathway for new application writers and current application writers who will want to migrate to new technologies (like visual serving) in the coming years.

> **Note:** Hereafter in this manual, OpenGL Volumizer unqualified refers to OpenGL Volumizer 2.

OpenGL Volumizer Versus Other APIs

OpenGL Volumizer, like other SGI graphics APIs, is a layer of functionality that sits on top of OpenGL, as shown in Figure 1-1.
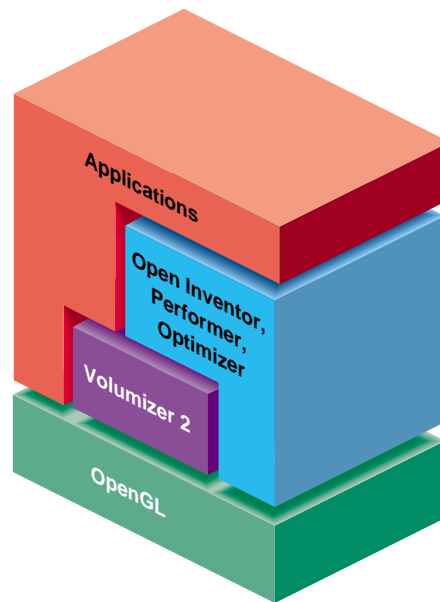
Figure 1-1   OpenGL Volumizer in Relation to Other Graphics APIs

OpenGL Volumizer is a toolkit designed to handle the volume rendering aspect of an application. You can use other toolkits, like OpenGL Performer and Open Inventor, to structure the other elements of your application. The API is designed to allow seamless integration with other scene graph APIs.

Features

OpenGL Volumizer is a rich toolkit with features that include the following:

- A high-level, extensible, C++ API that segments classes and methods based on the corresponding procedural-versus-descriptive nature of the component members. The core API consists of a volumetric-shape description API and a procedural 3D texture-based render action. See Chapter 3, "The OpenGL Volumizer API" and Chapter 4, "Texture Mapping Render Action" for more information.

- Integrated shading capabilities to perform volumetric shading, which allows techniques like multivolume blending and volumetric lighting to improve realism and to implement very high quality visualizations. See Chapter 5, "Custom Volumetric Shading" for more information.

- Large data management capabilities, including support for 3D clip textures, which allow interactive visualization of extremely large datasets. See Chapter 6, "The Large-Data API: 3D Clip Textures" for more information.

- Direct rendering of irregular grids using projected tetrahedra. See Chapter 7, "Irregular Grids: The Projected Tetrahedra Render Action" for more information.

- The Volume Data Description Language (VDDL). VDDL allows you to change volume data descriptions at run time by simply editing a text file. See Chapter 8, "OpenGL Volumizer File Loader" for more information.

- Thread safety, which allows implementation of multithreaded applications that run on multiple processors and graphics engines in conjunction with APIs like OpenGL Multipipe SDK and OpenGL Performer.

- A container for volume rendering techniques. Developers can integrate their own scene graph parameters and rendering algorithms in the API structure. The ability to incorporate such custom-tailored parameters and renderers gives the flexibility to advanced developers to implement and experiment with new rendering methods.

- Multiplatform support: IRIX, Linux, and Windows systems

- Examples that include a transfer function editor, data loaders, and a volume rendering application for multipipe systems, along with sample integration with existing APIs.

Supported Platforms

As the product name implies, OpenGL Volumizer targets OpenGL applications. This section describes the following platforms:

- SGI IRIX platforms
- 32-bit Linux platforms
- 64-bit Linux platforms
- Windows platforms

Note: This guide refers to *GPU-based* systems. They include the Silicon Graphics Onyx4 UltimateVision and Silicon Graphics Prism families of graphics systems.

SGI IRIX Platforms

OpenGL Volumizer supports all SGI graphics systems with 3D texture mapping and color tables (for instance, Silicon Graphics O2 systems do not support 3D texture mapping). However, OpenGL Volumizer is optimized for Onyx systems with InfiniteReality or UltimateVision graphics and is targeted at SGI scalable servers. Specifically, the texture mapping render action is optimized for InfiniteReality systems.

IRIX 6.5 or later is required.

32-Bit Linux Platforms

OpenGL Volumizer supports the following 32-bit Linux platforms:

- SUSE LINUX 9.1 or later
- Red Hat 8
- Red Hat 9
- Red Hat Enterprise Linux

On 32-bit Linux systems, the OpenGL Volumizer libraries are compiled with GCC 3.2 libraries and are **not** binary compatible with GCC 2.*x*. The Red Hat systems ship with GCC 3.x compiled libraries by default.

Graphics cards must support 3D textures. Specific shaders require other OpenGL extensions (see Chapter 5, "Custom Volumetric Shading"). The following graphics cards are supported:

- ATI RADEON 9000 series or later
- ATI FireGL series
- NVIDIA GeForce FX or later

64-Bit Linux Platforms

On 64-bit Linux systems, the OpenGL Volumizer libraries are compiled with GCC 3.2. OpenGL Volumizer supports Silicon Graphics Prism platforms, which require SGI ProPack 3 Service Pack 2 or later.

Graphics cards must support 3D textures. Specific shaders require other OpenGL extensions (see Chapter 5, "Custom Volumetric Shading"). The following graphics cards are supported:

- ATI RADEON 9000 series or later
- ATI FireGL series
- NVIDIA GeForce FX or later

Windows Platforms

OpenGL Volumizer supports the following Windows platforms:

- Windows NT
- Windows XP
- Windows 2000

Graphics cards must support 3D textures. Specific shaders require other OpenGL extensions (see Chapter 5, "Custom Volumetric Shading"). The following graphics cards are supported:

- ATI RADEON 9000 series or later
- ATI FireGL series
- NVIDIA GeForce FX or later

Product Components

This section describes the product components in the following subsections:

Product Libraries

The product libraries fall into the following categories:

- Core OpenGL Volumizer library

    - Shape description API (See Chapter 3.)

    - Render actions

        * Texture Mapping Render Action (See Chapter 4.)
        * Clip Texture Render Action (See Chapter 6.)
        * Projected Tetrahedra Render Action (See Chapter 7.)

- Additional modules:

    - ISL integration library (See Chapter 5.)

    - XML file loader interface (See Chapter 8.)

    - DICOM data loader library (See Chapter 8.)

    - Plot3D data loader library (See Chapter 7.)

Table 1-1 shows the names of the libraries on the various platforms.

Table 1-1   Product Library Names

Library	IRIX and Linux	Windows
Core OpenGL Volumizer library	`libvz.so`	`vz.dll`
ISL integration library	`libvzisl.so`	`(Not available)`
XML file loader	`libvzxml.so`	`vzxml.dll`
DICOM data loader	`libvzdicom.so`	`vzdicom.dll`
Plot3d data loader	`libvzplot3d.so`	`vzplot3d.dll`

For the location of these libraries, see the subsection "OpenGL Volumizer File Locations" on page 9.

Demos, Example Modules, and Utilities

As demos, OpenGL Volumizer provides sample applications and plug-in libraries. In addition, OpenGL Volumizer includes a number of example modules and utilities to help you use the API. These modules are not part of the supported API . These modules include the following items:

- Plug-in libraries
  - Sample transfer function editor
  - Examples of volume loaders and writers
- Toolkit libraries
  - Additional utility programs to reformat data into a format readable by OpenGL Volumizer sample applications
  - Integration libraries for the APIs OpenGL Performer, Open Inventor, and Visualization Toolkit (VTK)
- Sample applications
  - Simple examples of each of the render actions
  - Volume-viewer application, Volview, based on OpenGL Multipipe SDK

Note: Volview is not supported on Windows.

Figure 1-2 shows the structure of the Volview application, which is written on top on OpenGL Volumizer, OpenGL Multipipe SDK, and uses the plug-in libraries provided with OpenGL Volumizer sample distribution.
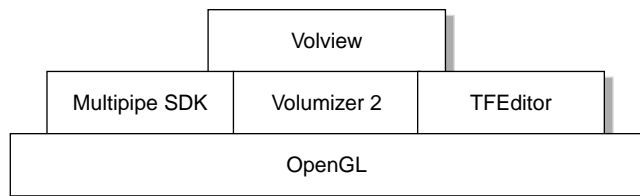
Figure 1-2   Volview: OpenGL Multipipe Volume Viewer Application

For the location of the demos and example modules, see the subsection "OpenGL Volumizer File Locations" on page 9.

OpenGL Volumizer File Locations

The environment variable VZROOT defines the root directory for OpenGL Volumizer files.

Default value for VZROOT on IRIX and Linux systems:

`/usr/share/Volumizer2/`

Default value for VZROOT on Windows systems:

`c:\Program Files\Silicon Graphics\OpenGL Volumizer`

Table 1-2 shows the directories for the major program components.

Table 1-2   File Locations

Files	IRIX and Linux	Windows
Product libraries	IRIX: `/usr/lib32` or `/usr/lib64` Linux: `/usr/lib`	`%VZROOT%\bin` (DLLs) `%VZROOT%\lib` (import libraries)
Conversion utilities	`VZROOT/bin`	`%VZROOT%\bin`
Demos	`/usr/demos/Volumizer2`	`%VZROOT%\bin\demos`
Sample source code	`VZROOT/src`	`%VZROOT%\src`
Header files	`/usr/include/Volumizer2`	`%VZROOT%\include`
Sample datasets	`VZROOT/data`	`%VZROOT%\data`
Developer reference pages	`VZROOT/doc`	`%VZROOT%\doc`
Release notes	`VZROOT/release_notes`	`%VZROOT%\release_notes`

Note: Hereafter in this guide, file paths will be denoted in this manner: VZROOT/*<relative-path>*. The Windows equivalent, `%VZROOT%\`*<relative-path>*, will not be shown.

Getting Started

This chapter describes how you use OpenGL Volumizer to build an application. The chapter consists of the following sections:

- "Basic Concepts"
- "Sample Volume Rendering Application"

Basic Concepts

There are two key notions in OpenGL Volumizer:

- Introduction of a *volumetric shape node* to the scene graph
- Highly parameterized control of rendering, termed *render actions*

The following subsections introduce these two concepts. Chapter 3, "The OpenGL Volumizer API" describes these concepts in greater detail.

The Shape Node

The shape node encapsulates a volume in a manner that allows you to separate its geometry from its appearance. The volume's geometry defines its spatial attributes and a region of interest while the volume's appearance defines its visual attributes. The appearance itself consists of a list of parameters that are specific to the particular rendering technique being applied to the shape. Figure 2-1 illustrates this concept.
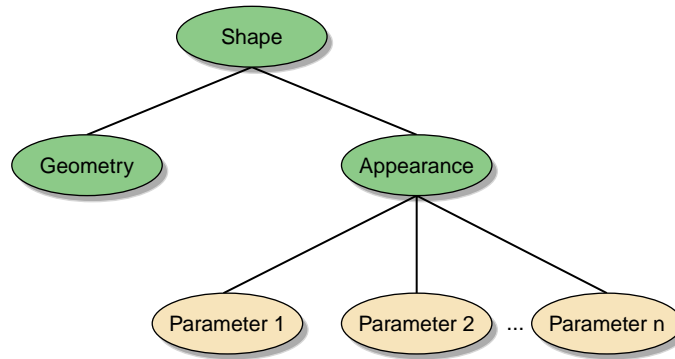
Figure 2-1   The Shape Node

The shape node contains all information required to render itself. Hence, it can be treated as the leaf node of a scene graph. You can create a more complex scene graph by inserting these shape nodes to represent the volumetric components of the scene, as shown in Figure 2-2.
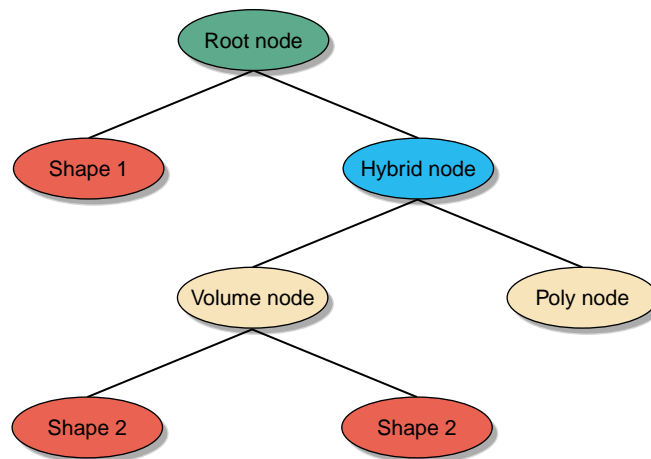
Figure 2-2   A More Complex Scene Graph

Figure 2-2 shows an example of a scene graph that has polygonal data mixed with volumetric shapes. Such a scene graph can sit on top of the OpenGL Volumizer API in conjunction with other scene graph APIs like OpenGL Performer or Open Inventor.

Render Actions

A render action primarily implements a visualization algorithm that accepts a shape node and renders it. Hence, in OpenGL Volumizer, there is a clear distinction between the descriptive components of the scene (the shape nodes) and the procedural components (the render actions). Your control of render actions allows you flexibility in employing known visualization algorithms. Figure 2-3 illustrates rendering actions.
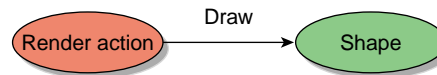
Figure 2-3   Render Actions

Closely related to render actions are shaders, which are used to apply specific rendering techniques to generate a desired visual effect. Shaders deal with the specific OpenGL state settings that need to be applied during the rendering process. The shaders are attached to the shape's appearance and expect a list of parameters for rendering the shape. Figure 2-4 illustrates the function of shaders.
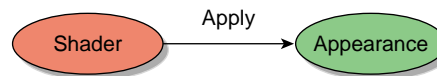
Figure 2-4   Shaders

Sample Volume Rendering Application

Example 2-1 shows a simple volume rendering application that uses the OpenGL Utility Toolkit (GLUT) to manage the user interface. This application demonstrates how to create a shape node and how to render it. The source for this application can be found in the directory `VZROOT/src/apps/simple/pguide/`.

Example 2-1   Sample Volume Rendering Application

```
// C / C++
#include <stdlib.h>
#include <iostream.h>

// OpenGL / GLUT
#include <GL/gl.h>
#include <GL/glut.h>

// IFL
#include <loaders/IFLLoader.h>

// Volumizer2
#include <Volumizer2/Version.h>
#include <Volumizer2/Shape.h>
#include <Volumizer2/Block.h>
#include <Volumizer2/Appearance.h>
#include <Volumizer2/ParameterVolumeTexture.h>
#include <Volumizer2/TMRenderAction.h>
#include <Volumizer2/TMSimpleShader.h>

// Global variables
vzShape *shape = NULL;
vzTMRenderAction *renderAction = NULL;
GLint viewport[4];
int lastPosition[2] = {0, 0};
float angles[2] = {0, 0}, lastAngles[2] = {0, 0};

///////////////////////// Volumizer /////////////////////////////

//  Load the volume data and initialize the shape node.
void loadVolumeData(char *fileName)
{
    // Print the volumizer version string
    cerr<<vzGetVersionString()<<endl;

    // Create a data loader
    IFLLoader *loader = IFLLoader::open(fileName);
    if (loader == NULL) {
        cerr<<"Error: couldn't open file "<<fileName<<endl;
        exit(0);
    }
```

```
        // Load the volume data
        vzParameterVolumeTexture *volume = loader->loadVolume();
        if (volume == NULL) {
            cerr<<"Error: couldn't read volume data"<<endl;
            delete loader;
            exit(0);
        }

        // Initialize appearance
        vzShader *shader = new vzTMSimpleShader();
        vzAppearance *appearance = new vzAppearance(shader);
        shader->unref();
        appearance->setParameter("volume", volume);
        volume->unref();

        // Initialize geometry
        vzGeometry *geometry = new vzBlock();

        // Initialize shape node
        shape = new vzShape(geometry, appearance);
        geometry->unref();
        appearance->unref();

        // Initialize the render action
        renderAction = new vzTMRenderAction(1);
        renderAction->manage(shape);
        }

// Draw the volume data
void renderVolumeData()
{
    // Begin drawing
    renderAction->beginDraw(VZ_RESTORE_GL_STATE_BIT);
    renderAction->draw(shape);
    renderAction->endDraw();
}

// Clean up the shape node and the render action
void cleanup()
{
    // Delete the render action and unref() the shape node
    renderAction->unmanage(shape);
    delete renderAction;
    shape->unref();
}
```

```
//////////////////////// GLUT callback functions////////////////////

// glutDisplayFunc() callback function
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDisable(GL_DEPTH_TEST);

    // Viewport
    glViewport(viewport[0], viewport[1], viewport[2], viewport[3]);

    // Projection matrix
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);

    // Modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glRotated( 90 + angles[1], 1, 0, 0);
    glRotated(180 + angles[0], 0, 0, 1);
    glScalef(1.5, 1.5, 1.5);
    glTranslatef(- 0.5, - 0.5, - 0.5);

    // Enable back-to-front alpha blending
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Render the volume data
    renderVolumeData();
    glutSwapBuffers();
}

// glutKeyboardFunc() callback function
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:
            cleanup();
            exit(0);
    }
}

// glutReshapeFunc() callback function
```

```
void reshape(int width, int height)
{

    // Update viewport
    viewport[0] = 0;      viewport[1] = 0;
    viewport[2] = width; viewport[3] = height;
    glutPostRedisplay();
}

// glutMouseFunc() callback function
void mouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN) {
        lastPosition[0] = x;
        lastPosition[1] = y;
        lastAngles[0] = angles[0];
        lastAngles[1] = angles[1];
    }
}

// glutMotionFunc() callback function
void motion(int x, int y)
{
    angles[0] = lastAngles[0] + (lastPosition[0] - x) / 4.0;
    angles[1] = lastAngles[1] + (y - lastPosition[1]) / 4.0;
    glutPostRedisplay();
}

// main
void main(int argc, char *argv[])
{
    if(argc < 2) {
        cerr<<"Usage: "<<argv[0]<<" <filename>"<<endl;
        exit(0);
    }
    glutInit(&argc, argv);
    loadVolumeData(argv[1]);

    // Initialize window
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutCreateWindow("Simple Volume Viewer");

    // Initialize callbacks
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
```

```
        glutKeyboardFunc(keyboard);
        glutMouseFunc(mouse);
        glutMotionFunc(motion);
        glutMainLoop();
}
```

The following subsections describe the sample application:

- "Prerequisites"

- "Compiling the Application"

- "Running the Application"

- "Program Components"

Prerequisites

The following software must be installed on your system:

- OpenGL Volumizer

- Source for the sample application, installed in `VZROOT/src/apps/simple/pguide`

- GLUT (available and free on the Web)

Note that the application links against `libvzLoaders.so` (`vzLoaders.dll` on Windows), which is generated by compiling the source given in `VZROOT/src/lib/loaders`.

Compiling the Application

To compile the application, enter the following commands:

(On IRIX)

```
% CC -o glut.o glut.cxx -c -I/usr/share/Volumizer2/src/lib
% CC -o viewer glut.o -L/usr/share/Volumizer2/lib/irixn32 -lvz
-lvzLoaders -lglut -lGLU -lGL -lXmu -lX11
```

Note that the preceding commands create an n32 executable for the application. To compile a 64-bit version, add the −64 flag to both commands and change the library path to `irix64`.

(On Linux)

```
% g++ -o glut.o glut.cxx -c -I/usr/share/Volumizer2/src/lib
% g++ -o viewer glut.o -L/usr/share/Volumizer2/lib/linux32 -lvz
-lvzLoaders -lglut -lGLU -lGL -lXmu -lX11
```

Note that the preceding commands assumes 32-bit Linux. On 64-bit Linux systems, change the library path to `linux64`.

(On Windows)

```
C:\> cl.exe /c glut.cxx /I ..\..\..\..\include
/I ..\..\..\..\lib /D VISUALCXX
C:\> link.exe /out: viewer glut.obj /libpath: ..\..\..\..\lib
vzLoaders.lib vz.lib opengl32.lib glut32.lib
```

Note that you need to run the `VCVARS32.BAT` batch file provided by Microsoft Visual Studio before compiling the sample code. Alternatively, you can create your own project in Microsoft Visual Studio to compile the application. Also, note that you need to have `libtiff` for Windows installed on your system in order to run this application.

Running the Application

To run the application, enter the following commands:

(On IRIX or Linux)

```
% viewer /usr/share/Volumizer2/data/medical/Phantom/
  CT.Head.Bone.char.tif
```

(On Windows)

```
C:\> viewer.exe ..\..\..\..\data\medical\Phantom\CT.Head.Bone.char.tif
```

Program Components

The program can be divided into the two main components:

- OpenGL Volumizer—Manages the scene graph and draws the volume data.

- GLUT—Manages the display and user interaction.

The following subsections describe program initialization and the OpenGL Volumizer component of the program:

- "Basic Initialization"

- "Creating the Shape Node"

- "Creating the Render Action"

- "Rendering the Volume Data"

- "Freeing the Allocated Memory"

Basic Initialization

The OpenGL Volumizer include files are located in the directory `/usr/include/Volumizer2` for IRIX and Linux and `%VZROOT%\include` for Windows..

This program also uses the IFL data loaders, which are installed in the directory `VZROOT/src/loaders`.

Creating the Shape Node

The method **loadVolumeData()** loads in a volumetric data set from the disk and then creates the vzShape node. The following are the key actions required to create the shape node.

1. Create a loader for the volume data.

   The following line from the function **loadVolumeData()** creates an IFL loader.

   ```
   IFLLoader *loader = IFLLoader::open(fileName);
   ```

   Upon success, **open()** returns the data loader; otherwise, a NULL pointer is returned. The value *fileName* should point to a valid file in the IFL TIFF format.

2. Load the volume data.

   The following line uses the loader just created to load in the volume data:

   ```
   vzParameterVolumeTexture *volume = loader->loadVolume();
   ```

The **loadVolume()** method returns a `vzParameterVolumeTexture` value. The `vzParameterVolumeTexture` value corresponds to the only shader parameter attached to the shape's appearance in this example. One could attach multiple shader parameters to an appearance. See "Shader Parameters" in Chapter 3 for details.

3. Create a shader for the appearance.

   The following line creates a shader:

   ```
   vzShader *shader = new vzTMSimpleShader();
   ```

   The shader determines the particular rendering technique to be applied to the shape while rendering it. The `vzTMSimpleShader` shader performs simple volume rendering using 3D texture mapping. See Chapter 4, "Texture Mapping Render Action" for details.

4. Create the shape's appearance.

   The following line creates the shape's appearance:

   ```
   vzAppearance *appearance = new vzAppearance(shader);
   ```

   The appearance for the shape determines how the shape looks when rendered. It accepts a `vzShader` value as an argument to its constructor.

5. Add the volume texture as a parameter to the appearance.

   The following line adds the parameter:

   ```
   appearance->setParameter("volume", volume);
   ```

   The shader `vzTMSimpleShader` needs a parameter named `volume`, which should be of the type `vzParameterVolumeTexture`. The appearance adds the parameter to its list of parameters.

6. Decrement the reference counts of the shader and the volume texture.

   On initialization, the reference count of any OpenGL Volumizer object is set to 1. The previous two calls cause the appearance to increase the reference counts of the shader and the volume texture. The following **unref()** calls decrease the reference counts by one. This ensures that `shader` and `volume` will be deleted when `appearance` is deleted.

   ```
   shader->unref(); // shader ref count = 1
   volume->unref(); // volume ref count = 1
   ```

7. Initialize the geometry.

   The following line creates a simple cuboidal geometry:

```
vzGeometry *geometry = new vzBlock();
```

The `vzBlock` object represents a simple axis-aligned cube. By default, the extents of the cube are set to (0, 0, 0) and (1, 1, 1).

8. Initialize the shape node.

   The following line creates the shape node `shape` with the given `geometry` and `appearance`:

   ```
   shape = new vzShape(geometry, appearance);
   ```

   Again, the reference counts of `geometry` and `appearance` are increased by one.

9. Decrement the reference counts of the `geometry` and `appearance`.

   The following lines ensure that `geometry` and `appearance` will be deleted when `shape` is deleted.

   ```
   geometry->unref();// geometry ref count = 1
   appearance->unref(); // appearance ref count = 1
   ```

Figure 2-5 depicts the resulting shape node.
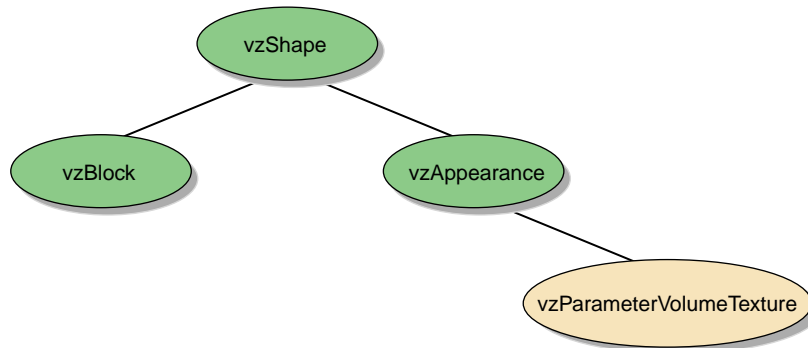
Figure 2-5   Shape Node in Sample Application

Creating the Render Action

The render action used in this example is Texture Mapping Render Action (TMRenderAction). It renders the given geometry by slicing it using sampling planes and then compositing them in a back-to-front order with alpha blending.

The next two steps create the render action and manage the shape.

1. Create a `vzTMRenderAction`.

   ```
   renderAction = new vzTMRenderAction(1);
   ```

   The integral argument specifies the number of threads the render action is allowed to create.

2. Manage the `vzShape`.

   ```
   renderAction->manage(shape); // shape ref count = 2
   ```

   The render action adds the given shape to its list of managed shapes. In this case, it ensures that the volume textures in the shape are made resident in the texture memory of the graphics subsystem. The render action also maintains a reference count for the shape inside the **manage()** method.

Rendering the Volume Data

The method **renderVolumeData()** draws the created shape node using the `vzTMRenderAction`.

The following lines render the shape node:

```
renderAction->beginDraw(VZ_RESTORE_GL_STATE_BIT);
renderAction->draw(shape);
renderAction->endDraw();
```

The **beginDraw()** method tells the render action that the application is done creating and managing the shape nodes for this frame and now it needs to render the shapes. The actual rendering is done inside the **draw()** calls for the individual shapes to be rendered. The **endDraw()** method marks the end of the rendering phase.

Freeing the Allocated Memory

The method **cleanup()** deletes the shape node and the render action. The reference counting ensures that all the other components of the shape node are also deleted when the shape node is deleted.

The following lines delete the render action and the shape node:

```
renderAction->unmanage(shape); // shape ref count = 1
delete renderAction;
shape->unref(); // shape ref count = 0. Deletes itself
```

For the details about the shape node and related classes, refer to Chapter 3, "The OpenGL Volumizer API". Chapter 4, "Texture Mapping Render Action" describes in detail TMRenderAction and related shaders.

The OpenGL Volumizer API

Chapter 2, "Getting Started" provides an overview of the basic concepts of OpenGL Volumizer. This chapter describes in greater detail how you use the OpenGL Volumizer API. For details on the individual classes, refer to their respective man pages.

This chapter has the following sections:

- "Base Classes"
- "Shape-Related Classes"
- "Rendering Classes"
- "Error Reporting"

Base Classes

Table 3-1 summarizes the base classes for all the OpenGL Volumizer object classes.

Table 3-1 Base Classes

Class	Description
vzMemory	Memory allocation and deallocation routines
vzObject	Reference counting and deletion notification

The following subsections describe the roles of these classes:

- "Memory Allocation and Deallocation"
- "Reference Counting and Deletion Notification"

Memory Allocation and Deallocation

All OpenGL Volumizer object classes are derived from the base class vzMemory. It provides you with the ability to control memory allocation and deallocation of objects by providing two static operators new and delete. By default, the operators new and delete simply use the **malloc()** and **free()** functions. By overriding this default behavior, you can customize the allocation and deallocation of OpenGL Volumizer objects.

For example, consider the case of designing a volume rendering application using OpenGL Performer where OpenGL Volumizer shape nodes are used to represent the volumetric components of the scene. OpenGL Performer uses a multiprocess model of execution, using the **fork()** system call to set up separate processes for APP, CULL, and DRAW. To share the objects between the processes, you would need to allocate them in shared memory. To accomplish this, simply override the default new and delete operators by setting two callback functions: one for allocation and one for deallocation.

For instance, the following lines force the API to use OpenGL Performer shared arenas:

```
// Set the callbacks for vzMemory base class
vzMemory::setMemoryManagementCallbacks (myNewCB, myDeleteCB, NULL);
```

The callbacks **myNewCB()** and **myDeleteCB()** look like the following:

```
// Callback for memory allocation - uses pfMalloc()
void *myNewCB (size_t nBytes, void *userData) {
   return pfMalloc (nBytes, pfGetSharedArena());
}

// Callback for freeing memory - uses pfFree()
void myDeleteCB (void *ptr, void *userData) {
   pfFree (ptr);
}
```

Any subsequent calls to **vzMemory::malloc()** and **vzMemory::free()** will use the callbacks specified above to allocate and de-allocate memory, respectively. All application data that needs to be shared across processes in the preceding case should be allocated using these methods. Refer to the vzMemory man page for details of the functions used in the preceding code.

Reference Counting and Deletion Notification

The vzObject class encapsulates the notions of reference counting and deletion notification, which this section describes separately.

Reference Counting

Reference counting allows painless memory management of objects that are shared between multiple objects. The basic idea is to maintain a counter for each object to indicate the number of outside references currently being held for it. Thus, the counter value indicates the number of users and objects that have a reference for the object. A count of zero indicates that there are no references to the object and, hence, it is safe to delete it.

All OpenGL Volumizer objects are derived from the vzObject class, which provides simple reference counting and deletion notification facilities. When an object is created, its reference count is initialized to one. If the reference count of an object reaches zero, the object calls its own destructor.

The vzObject class provides two public methods: **ref()** and **unref()**, which can be used to increase and decrease the reference count for the object, respectively. For each invocation of **ref()**, the count is increased by one and similarly for **unref()**, the count is decreased by one. If inside an **unref()** call the counter reaches zero, the object deletes itself.

The following code snippet from the example used in "Sample Volume Rendering Application" in Chapter 2 illustrates the use of reference counts for the shader.

```
// shader ref count = 1
vzShader *shader = new vzTMSimpleShader();

// shader ref count = 2
vzAppearance *appearance = new vzAppearance(shader);

// shader ref count = 1
shader->unref();
```

The shader is unreferenced since the appearance would invoke a **ref()** on it inside the constructor. Unreferencing the shader ensures that it would get deleted when the appearance is deleted. This is because, in its destructor, the appearance would invoke an **unref()** on the shader, which brings its reference count to 0, hence, deleting it. The following code illustrates the use of reference counts for the geometry and appearance classes.

```
// geometry ref count = 1
vzGeometry *geometry = new vzBlock();

// geometry ref count = 2, appearance ref count = 2
shape = new vzShape(geometry, appearance);

// geometry ref count = 1
geometry->unref();

// appearance ref count = 1
appearance->unref();
```

If you are not careful, you might make mistakes with the reference counting system. Two possible symptoms result from mismanagement of reference counts:

- Your program leaks memory. This is caused by forgetting to use **unref()** on an object once you are done using it.

- You have called methods on objects that have already been deleted. Once an object's reference count drops to zero, it is invalid to call methods on it. Doing so will have unpredictable results.

The OpenGL Volumizer API in itself is very consistent with the use of reference counts— that is, every object A that keeps a reference for another object B invokes a **ref()** on B. Also, A is supposed to invoke an **unref()** on B when it removes that reference. If you were to create a new geometry and use it for the shape node in the sample application from "Sample Volume Rendering Application" in Chapter 2, you would do something like the following:

```
// create a new geometry
vzGeometry *newGeometry = createNewGeometry();

// update the geometry for the shape to the new one
shape->setGeometry(newGeometry);
```

The following steps occur inside the **setGeometry()** method of vzShape:

```
void setGeometry(vzGeometry *newGeometry) {

    // ref() the new geometry
    newGeometry->ref();

    // unref() the old geometry
    currentGeometry->unref();

    // update the geometry
    currentGeometry = newGeometry;
}
```

To debug reference counts more effectively, you can set the debug level to 4 (see the vzError class for details). This causes the API to print the value of the reference count every time a **ref()** or **unref()** call is issued.

Deletion Notification

The OpenGL Volumizer API maintains a consistent system for memory allocation and deallocation. If you allocate any memory, then it is your responsibility to free that chunk of memory. To do this, it is essential for you to know when an object is about to be deleted—that is, when its reference count drops to zero. The API provides you the ability to specify deletion callbacks that are invoked just before an object is deleted. These callbacks can be used to do the necessary cleanup for the particular object.

The following code illustrates the use of this deletion notification system for freeing memory. Suppose you allocated a floating point array of vertex data and passed a pointer into the vzVertexArray class as in the following:

```
int numVerts = 20;
float *myData = new float[numVerts*3];
vzVertexArray *array = new vzVertexArray (numVerts, myData);
```

Since you allocated the memory for the array, you are responsible for freeing it. Using the deletion notification system, this can be accomplished very easily by installing a deletion

callback on the vertex array. This callback can be used to free the array since it is no longer needed, as shown in the following example:

```
// Add a deletion notification callback to the vertexArray just created
vertexArray->addDeletionCallback (myArrayDeletionCB, myData);

// Deletion notification callback - frees allocated memory
void myArrayDeletionCB (vzObject *object, void *userData) {
   delete [] userData;
}
```

It is valid to add multiple deletion callbacks with the same function pointer but different user data pointers. Refer to the `vzObject` man page for details of the callbacks and functions used in this section.

Shape-Related Classes

Table 3-2 summarizes the shape-related classes.

Table 3-2   Shape-Related Classes

Class	Description
vzShape	Container node for a volume's geometry and appearance
vzGeometry	Geometry of a shape node
vzVolumeGeometry	Volumetric geometry associated with a shape node
vzBlock	Volumetric geometry representing an axis-aligned cuboid
vzStructuredHexaMesh	Volumetric geometry representing a structured hexahedral mesh
vzUnstructuredMesh	Unstructured volumetric geometry
vzUnstructuredTetraMesh	Volumetric geometry representing an unstructured tetrahedral mesh
vzUnstructuredHexaMesh	Volumetric geometry representing an unstructured hexahedral mesh
vzVertexArray	An array of floating-point vertex coordinates
vzIndexArray	An array of integral indexes

Table 3-2   Shape-Related Classes **(continued)**

Class	Description
vzAppearance	Appearance description of a shape node
vzParameter	Shader parameter for a shape's appearance
vzSlicePlaneSet	A set of slice planes

This section describes how to use the shape-related classes in the following subsections:

- "Shape Node Construction"
- "Geometry Description"
- "Appearance Description"
- "Shader Parameters"

Shape Node Construction

As mentioned briefly in Chapter 2, the shape node encapsulates a volumetric representation in the form of its geometry and appearance. The shape node is the basic unit of rendering in the OpenGL Volumizer API. This means that the shape node is atomic; hence, you cannot render part of a shape. Shape nodes form the leaf nodes of a potentially more complex scene graph. The scene graph can be built upon the existing infrastructure provided by OpenGL Volumizer.

The geometry of a shape provides a region of interest while the appearance controls how it looks. In other words, the geometry of the shape describes **what** is rendered and the appearance describes **how** the geometry is rendered. Figure 3-1 illustrates this separation.
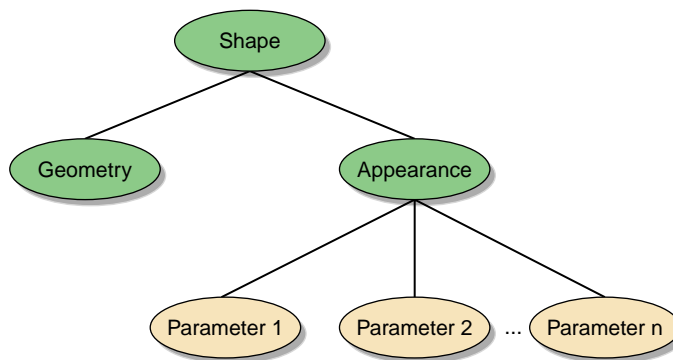
Figure 3-1   The Shape Node

Geometry Description

As mentioned before, the geometry of a shape defines what is rendered or the spatial attributes of the shape. In general, geometry can have any dimension. For example, a triangle is a 2D geometry type whereas a tetrahedron is 3D.

2D objects can be directly rendered using OpenGL primitives like triangles and polygons while 3D objects cannot. In order to render 3D objects using OpenGL, you must generate 2D primitives first and then use them to render the 3D objects.

The vzGeometry class is an abstract class which can be used to represent the geometry associated with a shape node. The class has one public method that allows you to retrieve the bounding box of the geometry. You can use the bounding box, which is an attribute of every geometric object, for culling to the viewing frustum, collision detection, or applying other special algorithms.

This following subsections further describe how to define your geometry:

- "Volumetric Geometry"
- "Simple Cuboidal Geometry"
- "General Tetrahedral Meshes"
- "Creating Your Own Volumetric Geometry Classes"
- "Arbitrary Polygonal Geometry"

Volumetric Geometry

OpenGL Volumizer allows you to specify 3D geometry using the vzVolumeGeometry class, which is derived from the vzGeometry class. The vzVolumeGeometry class can be used to represent a set of polyhedral primitives that define the volumetric structure of the shape node. On one hand, OpenGL Volumizer simplifies the description for the most commonly used cases of volumetric geometry like cuboids. On the other hand, it provides other constructs to allow specifying much more complex geometry types like structured hexahedral meshes and unstructured tetrahedral meshes. This is done by providing built-in classes that support these representations. For a complete list of the built-in volumetric geometry classes, see Table 3-2 on page 30.

All volumetric geometry types can be represented using a set of tetrahedra. Hence, internally OpenGL Volumizer uses the tetrahedron as the basic unit for representing volumetric geometry. The volumetric geometry class that represents arbitrary tetrahedral meshes is vzUnstructuredTetraMesh, which is described later in section "General Tetrahedral Meshes". All of the classes derived from the vzVolumeGeometry class need to know how to tessellate themselves into such a tetrahedal mesh. The following subsections describe the two most important volumetric geometry classes, vzBlock and vzUnstructuredTetraMesh. For a description of the others, refer to the man pages of the classes listed in Table 3-2 on page 30.

In addition to specifying the volumetric geometry, the vzVolumeGeometry class allows you to set arbitrary slice planes that pass through it. In many volume rendering applications, slice planes passing through the volume data can be a very powerful visualization technique. See the vzSlicePlaneSet man page for more details on how to use these slice planes in conjunction with volumetric geometry.

Simple Cuboidal Geometry

The vzBlock class is used to represent the simple case of an axis-aligned cuboid. This is the simplest and the most commonly used construct used to represent volumetric data. The vzBlock class has routines that allow you to set the offsets and dimensions of this cuboid.

The sample application "Sample Volume Rendering Application" in Chapter 2 uses a vzBlock object to represent the geometry of the volume data. By default, the constructor creates a cuboid at the offsets (0, 0, 0) and with dimensions (1, 1, 1). Try adding the following lines of code to the application before the renderAction->beginDraw() line:

```
// New offset and dimensions
float offset[3] = {0.25,0.25,0.25}, dimensions[3] = {0.5,0.5,0.5};

// Shape's geometry
vzBlock *block = (vzBlock *) shape->getGeometry();

// Modify the offsets for the cuboid
block->setOffsets(offset);

// Modify the dimensions of the cuboid
block->setDimensions(dimensions);
```

The result should be similar to the one shown in Figure 3-2. This simple example illustrates how modifying the geometry can allow you to *carve* your shape node.
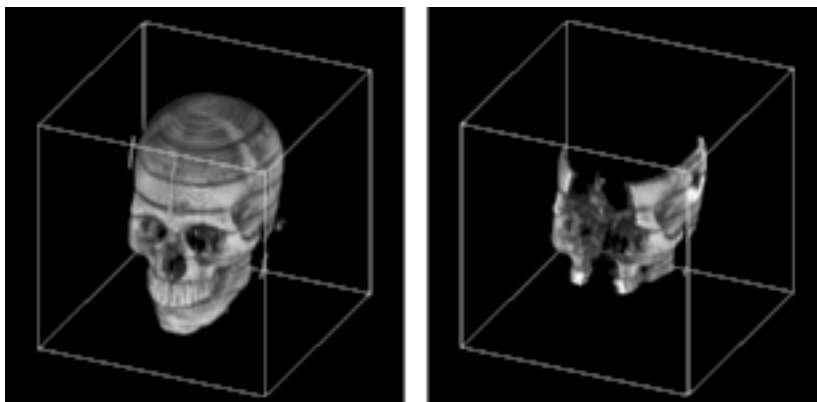
Figure 3-2   Modification of Shape Node from Sample Application

General Tetrahedral Meshes

The vzUnstructuredTetraMesh class is derived from the vzUnstructuredMesh class and represents indexed sets of tetrahedra. Each tetrahedron is represented by four integers that index a list of vertex coordinates. Figure 3-3 illustrates the structure of an unstructured tetrahedral mesh.
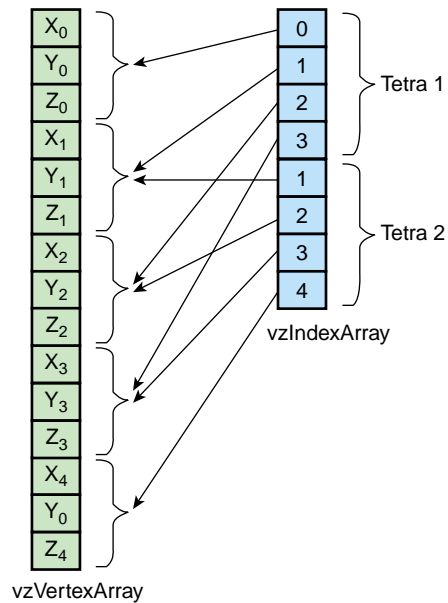
Figure 3-3   Construction of vzUnstructuredTetraMesh with Two Tetras

For example, you can represent an octahedron using a tetrahedral mesh consisting of six vertices and four tetrahedra.

Creating Your Own Volumetric Geometry Classes

It is possible to derive your own subclass of volumetric geometry simply by overriding the virtual **tessellate()** method of a vzVolumeGeometry object.

The **tessellate()** method is intended to take your geometry type and tessellate it into tetrahedra, which can then be used as geometry by the render actions. For example, you could design a vzSphere class that knew how to tessellate itself into tetrahedra. Simply create and initialize a vzIndexArray object and a vzVertexArray object for the resulting tetrahedral-mesh approximation.

Arbitrary Polygonal Geometry

In addition to the basic geometry elements outlined earlier in this section, OpenGL Volumizer allows applications to use arbitrary polygonal geometry within a

shape node. The vzPolyGeometry class and its associated virtual draw method provides a vehicle for such implementations.

The vzPolyGeometry class represents any polygonal geometry attached to a shape node. Derived from the vzGeometry abstract class, the class provides a pure virtual draw method, invoked by the render action while rendering the shape node. When the render action invokes the draw method, it passes the appropriate bounding box of the polygonal geometry to be rendered.This method allows applications to skip the "polygonization" step of the render action and instead, render arbitrary polygonal geometry. The OpenGL state used is the same as for the render action with polygonized geometry. The example below illustrates how to use this class:

```
class myPolyGeometry: public vzPolyGeometry {
public:
        virtual void draw(double bounding_box[6]) const
        {
            // Render geometry
        }
};
```

Appearance Description

The vzAppearance class encodes the visual attributes of a shape node. Volumetric appearance includes all descriptive characteristics that control the way a volumetric shape will look when it is rendered. The render actions are responsible for interpreting and applying this appearance description during the rendering process.

The appearance contains a list of parameters and a shader. Shaders associated with an appearance are specific to the render action to be applied to the shape. The list of parameters are attributes that are used by the shader to generate a desired visual effect. The appearance associates each parameter attached to it with a name and type.

Each render action supports one or more built-in shaders. Each shader in turn expects parameters of a given name and type, which are necessary for its use. For example, the sample application "Sample Volume Rendering Application" in Chapter 2 creates a simple appearance that uses the shader vzTMSimpleShader to volume render the given shape using 3D texture mapping.

TMRenderAction, used in the sample application, supports another built-in shader called vzTMTangentSpaceShader, which expects three parameters (see Chapter 4,

"Texture Mapping Render Action" for more details). The following code creates the appearance to be used to perform gradient-less shading of volumetric data:

```
// Create a tangent space shader
vzTMTangentSpaceShader *shader = new vzTMTangentSpaceShader();

// Create the appearance
vzAppearance *appearance = new vzAppearance(shader);

// Set the parameters required by the shader
appearance->setParameter ("volume", volumeTextureParameter);
appearance->setParameter ("lookup_table", lookupTableParameter);
appearance->setParameter ("lightdir", lightDirectionParameter);
```

The appearance stores a reference to the supplied parameters and associates them with the given names. Invoking the **setParameter()** method with a name already used but with a different parameter would overwrite the previous value.

Shader Parameters

Parameters are attached to the shape's appearance and provide the necessary information to complete a volumetric appearance description. Examples of parameters include 3D textures, texture lookup tables, lighting directions, and per-vertex floating point values.

The vzParameter class forms an abstract base class for all the shader parameters. For complete descriptions of the parameter classes and their usage, see Chapter 4, "Texture Mapping Render Action" and Chapter 7, "Irregular Grids: The Projected Tetrahedra Render Action".

Rendering Classes

Table 3-3 summarizes the function of the rendering classes.

Table 3-3   Rendering Classes

Class	Description
vzRenderAction	Renderer for drawing shape nodes
vzShader	Shader for generating a desired visual effect from an appearance

This section describes the use of the two classes listed in Table 3-3.

Render Actions

A render action, as mentioned earlier, implements a certain visualization algorithm to render the given shape nodes. Depending on the available resources and desired effect, you can apply different render actions to render your volume data. For example, TMRenderAction shipped with OpenGL Volumizer renders shape nodes using 3D texture mapping. You can also write your render action to implement different visualization algorithms if you want.

Render actions are responsible for more than just implementing a particular visualization algorithm. They can also perform the resource management for improving the performance of the rendering. This might also include doing their own OpenGL state management.

In order for a render action to implement intelligent resource management techniques, it should have some knowledge of the total size of resources available on the system and what is required to render the given shape nodes. You can provide information about the latter using the **manage()** and **unmanage()** methods of the render action. You can add a shape to the render action's list of managed shapes using **manage()** and remove it using **unmanage()**. Finally, the shapes can be drawn by calling **draw()** on the shapes. A shape that has not been managed cannot be drawn, but a shape that has been managed does not need to be drawn. Refer to the documentation specific to the render action you are using for the details on its implementation.

Shaders

Each render action recognizes a certain set of built-in shaders. Each built-in shader expects certain parameters to be defined. You must provide all of the parameters required for a given shader; failing to do so will generate an error. Shaders extract the required parameters from the respective appearances using the **getParameter()** method with the name of the respective parameters as an argument. For information on the built-in shaders available for the render action, see the documentation specific to the render action you are using.

Shaders are more lightweight as opposed to render actions in the sense that they are only concerned with the specific OpenGL state settings required to generate a particular visual effect. On the other hand, the render action performs more complex resource management for the list of shapes that are managed and need to be rendered. Hence, switching the shader for an appearance by using the **setShader()** method of the vzAppearance class would have minimal overhead. But using a different render action would involve more complex resource management to be done for the shape.

Error Reporting

The vzError class implements a mechanism for logging and reporting errors. It can also be used to print debug messages at run time. The class consists of a collection of static methods that allow you to do the error processing.

The following two subsections describe error processing:

- "Logging and Reporting Errors"
- "Printing Debug Messages"

Logging and Reporting Errors

The **vzError::log()** method is used by the library to log errors. Depending on the severity of the error (see vzErrorSeverity), you can issue a **log()** call with a severity of VZ_ERROR or VZ_WARNING. You can use your own error routine to handle all the logged errors. The default handler simply prints out an error message if the severity is VZ_WARNING. If the severity is VZ_ERROR, it calls **abort()** after printing the error message. The error handler installed applies to all threads.

You can use the convenience methods **error()** and **warn()** to log errors and warnings, respectively. Calling **error()** or **warn()** is equivalent to calling **log()** with the severity passed in as VZ_ERROR or VZ_WARNING.

The following example shows how to install your own error handling routine.

```
// Set the error handler for vzError::log()
vzError::setHandler (myHandler, NULL);
```

The handler might look like the following:

```
static void myHandler(vzErrorSeverity severity, vzErrorType type,
                      const char *format, va_list args, void* data)
{
    if(severity == VZ_ERROR)
        cerr<<"myHandler::Error!!!";
    else if(severity == VZ_WARNING)
        cerr<<"myHandler::Warning!!!";

    // Print the error message
    vfprintf(stderr, format, args);

    // Use the vzErrorType to do whatever else is needed!!!
    ....
}
```

Regardless of the error handler in effect, the first error encountered will be recorded and can be queried later using **getError()**. The **clear()** method resets the saved error to VZ_NO_ERROR. Errors are recorded and cleared on a per-thread basis.

Printing Debug Messages

The vzError class also provides the **message()** method to print debug messages that are neither errors nor warnings. Each debug message is given a particular debug level, passed as a parameter to the **message()** method.

The message will be output to stderr only if the debug level of the message is less than or equal to the current debug level. Therefore, the higher you set this debug level, the more debug information you will see. This is useful for debugging reference counts, monitoring texture memory usage, and so on.

The API internally does not use messages of levels 0 or 1. The guidelines in Table 3-4 are used by the API to print debug messages.

Table 3-4   Guidelines for Debug Messages

Level	Message
2	Major changes in execution model—setting error or memory callbacks, etc.
3	Changes caused by using set methods on object classes or managing and unmanaging shape nodes
4	Reference count changes
5	GL state-related changes

To debug applications effectively, you can print out the right level of debug messages by setting the environment variable VOLUMIZER_DEBUG_LEVEL to the appropriate value.

Texture Mapping Render Action

The Texture Mapping Render Action (TMRenderAction) is the standard render action provided by OpenGL Volumizer. This render action uses the 3D texture mapping hardware to perform volume rendering of the given shape nodes.

This chapter describes the following topics:

- "Volume Rendering Using 3D Texture Mapping"
- "Algorithm Used by TMRenderAction"
- "Volume Rendering Using TMRenderAction"
- "A Closer Look at TMRenderAction"

Volume Rendering Using 3D Texture Mapping

The main steps involved in volume rendering using 3D texture mapping are as follows:

1. Sample the volumetric data using sampling planes parallel to the viewport.

2. Render these planes using 3D texture mapping with the volumetric data as the currently bound 3D texture.

3. Composite the planes in a back-to-front manner using the over operator.

Figure 4-1 depicts the previous steps, respectively:

<div align="center">

Viewport-Aligned
sampling planes

3D Texture
sampling planes

Final Image
after Back-to-Front
compositioning
</div>
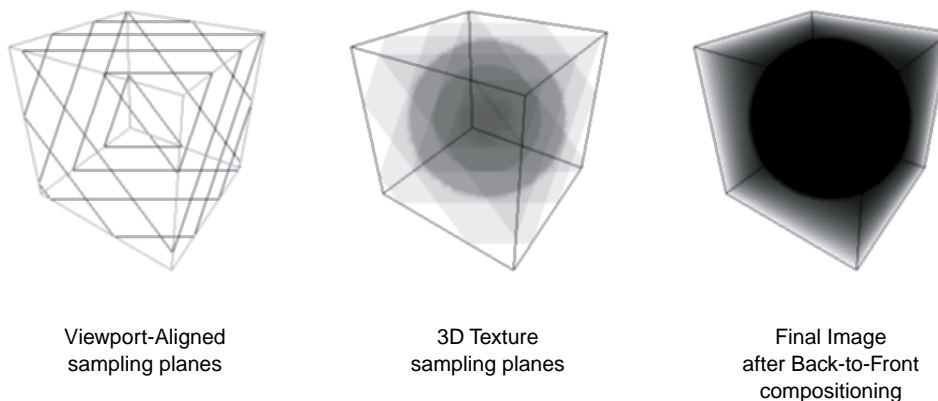
Figure 4-1   Viewport-Aligned Sampling Planes, 3D Textures Sampling Planes, and Final
Image after Back-to-Front Compositing

The following are advantages of using 3D texture mapping:

- Immediate-mode execution to prevent the overhead of storing transient geometry
  from polygonization.

- Optimized texture managment for improved texture download performance. [This
  includes the case of texture memory oversubscription.]

- Support for custom volumetric shading techniques along with built-in shaders for
  volumetric lighting and tagging.

- Transparent bricking and interleaving of texture data.

- Support for applications using multi-resolution and volume roaming techniques.

Algorithm Used by TMRenderAction

TMRenderAction implements a 3D volume rendering technique. The render action uses
the tetrahedron as the basic unit for representing volumetric geometry. The rendering
algorithm used by TMRenderAction consists of the following steps:

1. Tessellate the given volumetric geometry into a tetrahedral mesh.

   Figure 4-2 depicts the tessellation.
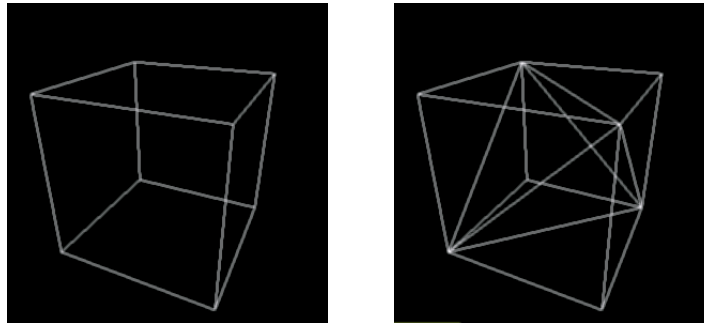
Figure 4-2   Original vzBlock and Corresponding Tessellation

2. Sort the tetrahedral mesh in a back-to-front visibility order.

3. Set the OpenGL state for a given shader.

4. Starting with the rearmost element, slice the tetrahedra one-by-one and render the polygonal geometry generated.

    Figure 4-3 illustrates the slicing and the final rendering.
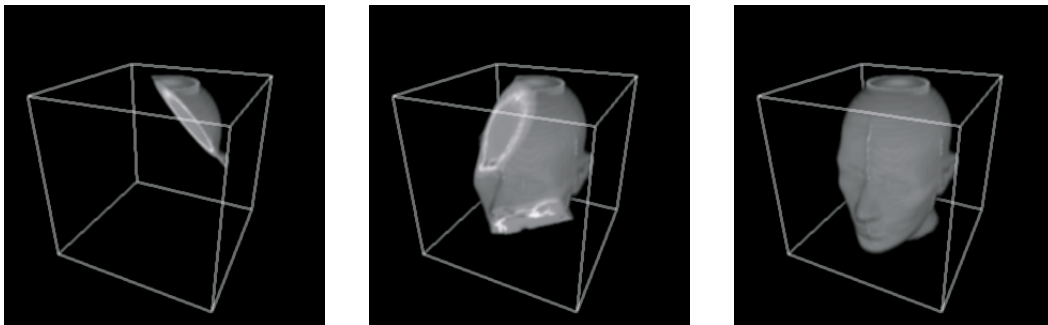
Figure 4-3   Back-to-Front Composited Slices for One, Three, and Five Tetrahedra

Volume Rendering Using TMRenderAction

The sample application in Chapter 2, "Getting Started" shows how simple it is to use the vzTMRenderAction class to render a simple volume shape. However, most real-life volume rendering applications need to do more complex operations than just render a simple volume shape. The vzTMRenderAction class has been designed with such applications in mind.

The following sections describe how to use the various components of TMRenderAction:

- "Creating the Render Action"
- "Managing and Drawing Shapes"
- "Using the Built-in Shaders"
- "Using Shader Parameters"

Creating the Render Action

The constructor to the render action takes an integer as a parameter, which represents the maximum number of threads the render action is allowed to create, as shown in the following:

```
vzTMRenderAction (int maxThreads);
```

The render action is not thread safe. Hence, do not share render actions across multiple threads. Also, for efficiency reasons, create only one render action per graphics pipe.

Managing and Drawing Shapes

The vzTMRenderAction base class has the following pure virtual methods:

- **manage()**
- **unmanage()**
- **draw()**

They allow the application to tell the render action about the shapes it wants to be cached and rendered. The process is shown in Figure 4-4.
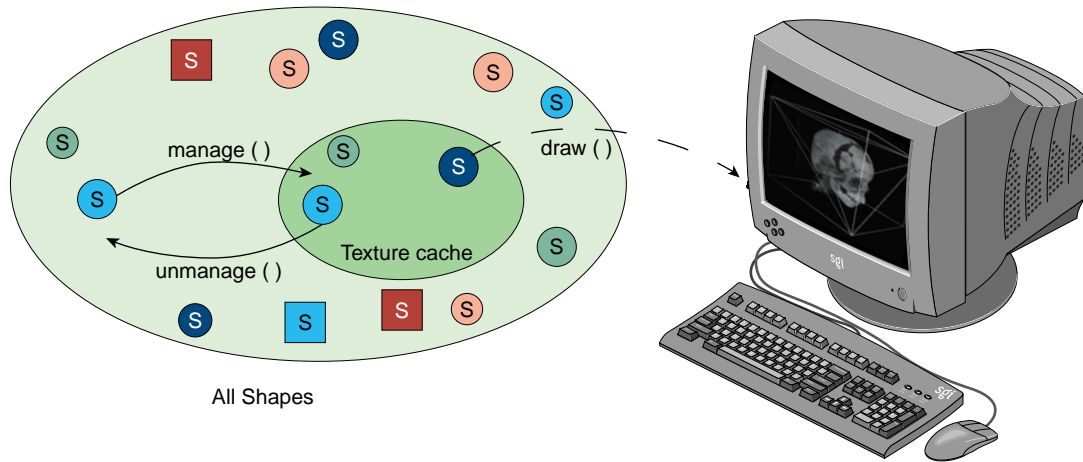
Figure 4-4   Managing, Unmanaging, and Drawing Shapes

TMRenderAction tries to load all the managed shapes into texture memory. Similarly, it removes any unmanaged shapes from the texture memory. All shapes that are drawn need to be managed first, even though it is not necessary to draw all the shapes currently managed.

The **beginDraw()** and **endDraw()** methods are used to inform the render action about the end of the management phase and the beginning of the rendering phase. The render action performs all the texture management in the **beginDraw()** method. Hence, all the **manage()** and **unmanage()** calls are queued until the application issues a **beginDraw()** call, when the actual management is done.

Using the Built-in Shaders

TMRenderAction currently supports a number of built-in shaders. All of them use 3D texture mapping to do volume rendering and implement specific techniques to generate a desired visual effect. All shaders render the shapes using one or more passes over the polygonal geometry generated from the slicing of the volumetric geometry. As you might expect, there is one parameter common to all shaders supported by TMRenderAction: volume. This parameter specifies the actual volume data to be rendered and is of the type vzParameterVolumeTexture.

The following subsections describe the list of built-in shaders currently supported by TMRenderAction:

- "vzTMSimpleShader"
- "vzTMLUTShader"
- "vzTMTangentSpaceShader"
- "vzTMGradientShader"
- "vzTMTagShader"

In addition to the built-in shaders, TMRenderAction also supports custom shaders. See Chapter 5, "Custom Volumetric Shading".

vzTMSimpleShader

The shader vzTMSimpleShader has the following parameter:

Parameter Name	Type
volume	vzParameterVolumeTexture

As the name implies, vzTMSimpleShader performs simple volume rendering of the given volume texture. The polygonal geometry to be rendered is generated as described earlier in section "Volume Rendering Using 3D Texture Mapping". This geometry is rendered in a back-to-front order with the given "volume" texture as the currently bound texture. Figure 4-5 shows a 256 MB size CT of a human head rendered using vzTMSimpleShader.
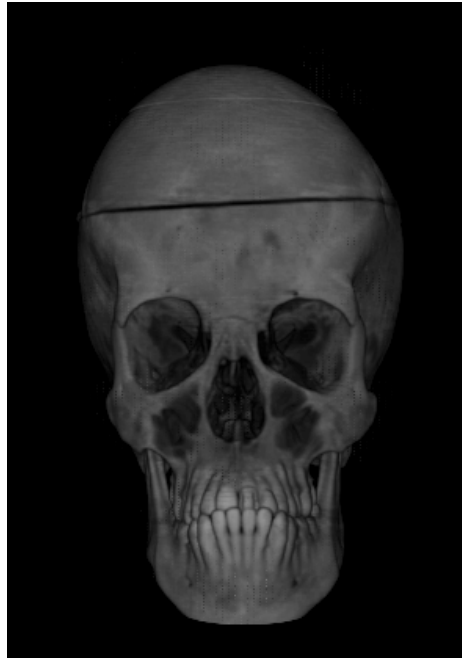
Figure 4-5   vzTMSimpleShader

vzTMLUTShader

The shader vzTMLUTShader has the two following parameters:

Parameter Name	Type
volume	vzParameterVolumeTexture
lookup_table	vzParameterLookupTable

The shader vzTMLUTShader allows you to apply transfer functions to the volume data by using a one-dimensional lookup table, which maps the interpolated texel values to color values. You can achieve a similar effect by applying the transfer function to precompute the color values for each texel in the volume and then use it as the volume texture for vzTMSimpleShader. This technique, however, would have a huge overhead due to the amount of computation involved. In addition, for every change to the transfer function the whole volume data will need to be re-downloaded to texture memory.

Using the OpenGL hardware, the shader vzTMLUTShader applies the transfer function in a system-dependent fashion. This process is much faster than doing the computation in software. Moreover, for every change to the transfer function, only the lookup table needs to be downloaded again, which is usually much faster than downloading the whole volume texture. Figure 4-6 shows the head data set used in Figure 4-5 on page 49 now rendered with vzTMLUTShader. Notice how use of the LUT shader allows getting rid of unwanted values in the data set.
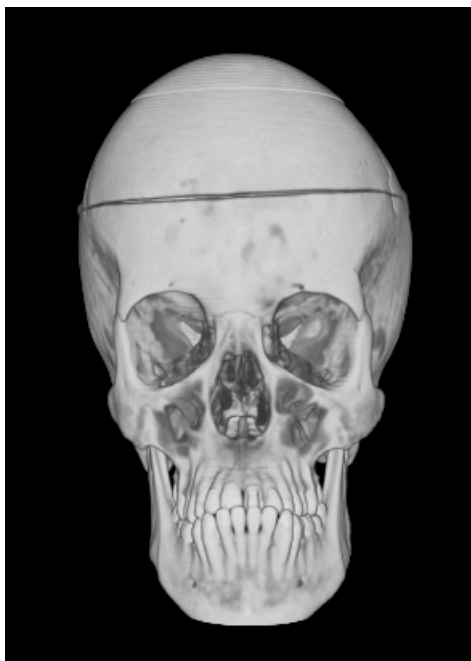
Figure 4-6   vzTMLUTShader

vzTMTangentSpaceShader

The shader vzTMTangentSpaceShader has the three following parameters:

Parameter Name	Type
volume	vzParameterVolumeTexture
lookup_table	vzParameterLookupTable
lightdir	vzParameterVec3f

The shader vzTMTangentSpaceShader implements a shader to perform lighting of volumetric data. The shader also uses lookup tables to apply transfer functions to the volumetric data. In order to perform the lighting computations, the shader also expects a parameter to specify the direction of the light source. Figure 4-7 shows the head data set used in Figure 4-5 on page 49 now rendered with vzTMTangentSpaceShader. Notice how use of the Tangent Space shader provides higher image quality and better depth cues.

Figure 4-7　　vzTMTangentSpaceShader

The technique implemented by vzTMTangentSpaceShader is a *gradient-less lighting* technique. It does not use the gradients for every texel of the volume data. The lighting computations are performed by using an approximation to the diffuse lighting component of the Phong shading model.

Note: The shader vzTMTangentSpaceShader does not generate correct lighting of volumetric data. It simply creates the appropriate visual effect by using a system-dependent technique.

The technique used here produces seams for bricked shapes along the borders of the bricks (see the later section "Texture Management" for more information on bricks). Use vzTMGradientShader for correct volumetric lighting of shapes.
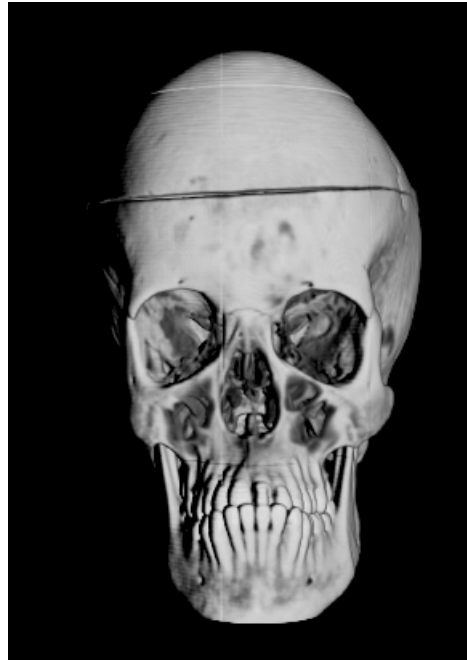
vzTMGradientShader

Note: This shader is not available in versions prior to OpenGL Volumizer 2.1.

The shader vzTMGradientShader has the following parameters:

Parameter Name	Type
volume	vzParameterVolumeTexture
gradient	vzParameterVolumeTexture
lightdir	vzParameterVec3f
lookup_table	vzParameterLookupTable
ambient	vzParameterVec3f (optional)
diffuse	vzParameterVec3f (optional)
specular	vzParameterVec3f (optional)
shininess	vzParameterVec3f (optional)

The shader vzTMGradientShader is a cross-platform shader that implements volumetric shading based on the Phong illumination model. The actual implementation used by the shader is platform-dependent. The shader uses per-voxel *gradient* information in order to compute the contributions from diffuse and specular coefficients to the output color. This gradient information can be specified as a parameter to the shape's appearance. In this case, the parameter volume defines the actual volume data, while the parameter gradient defines the corresponding gradient values. The RGB values of gradient provide the (a, b, c) coefficients for the gradient at the corresponding voxel in volume. The application must compute the gradient texture and add it to the shape's appearance.

Relative to the vzTMTangentSpaceShader, the use of a separate gradient texture generates more accurate lighting effects and does not have the artifacts associated with the bricking of shapes. However, this can potentially increase the texture memory usage by a factor of two to four times the size of the original volume texture.

Non-GPU-Based Systems

On systems which do not support the ARB_fragment_program OpenGL extension, the parameters `specular` and `shininess` do not have any effect.

On systems which do not support ARB_fragment_program or the ATI_fragment_shader OpenGL extensions, the shading algorithm uses the destination alpha to compute the gradient lighting. Hence, the application should make sure that the appropriate visual is selected. In this case, the system should support ARB_imaging extensions and the `gradient` texture should have an internal texture format of `VZ_INTENSITY`n, where n can 8, 12, or 16. The shader accepts a lookup table parameter to apply transfer functions to the volume data.

GPU-Based Systems

On GPU-based systems like Onyx4 and Silicon Graphics Prism systems that support the ARB_fragment_program extension, if parameter `gradient` is not specified, the shader computes the gradient values internally using *central differencing*.

vzTMTagShader

Note: This shader is not available in versions prior to OpenGL Volumizer 2.1.

The shader vzTMTagShader has the following three parameters:

Parameter Name	Type
volume	vzParameterVolumeTexture
tag	vzParameterVolumeTexture
lookup_table	vzParameterLookupTable

The vzTMTagShader implements system-dependent techniques to perform volumetric tagging. The algorithm uses two perfectly overlapping volumes. The `volume` texture defines the actual volume data, while the `tag` texture defines a 3D stencil buffer for

volume. Each value in `tag` contains the mask for the corresponding texel in `volume`. If the value of the tag texel is greater than 0.5, then the corresponding texel in the volume data is rendered; otherwise, the texel is masked out. Figure 4-8 shows the head data set used in Figure 4-5 on page 49 now rendered with vzTMTagShader. Notice how use of the vzTag Shader allows masking out arbitrary regions in the data set.

Figure 4-8   vzTMTag Shader

The vzTMTagShader uses a system-dependent technique to perform tagging. Ideally, the tag volume should require only one bit to represent each texel. However, on most graphics hardware, each texel will use at least one byte to represent a texel. On InfiniteReality graphics systems, the application can specify the internal texture format to be `VZ_QUAD_INTENSITY4` and ask the API to optimize the texture. The texture would then be interleaved so that each texel requires only four bits to represent it; this reduces texture memory consumption and improves the texture download rate. See the man page for vzParameterVolumeTexture for more details.

Notes:

- On systems that do not support the ARB_fragment_program or ATI_fragment_shader OpenGL extensions, the vzTMTagShader uses the stencil buffer to implement the tagging algorithm. Hence, the application should ensure that the appropriate visual is selected.

- Using vzTMTagShader has the overhead of storing an additional 3D texture in the texture memory. You can also generate the same effect by actually modifying the `volume` texture to remove the unwanted texels by setting their opacity to zero explicitly. This, however, has the disadvantage of modifying the original volume data.

Using Shader Parameters

The preceding section describes the list of shaders that are supported by TMRenderAction. The following subsections briefly describe the shader parameters used by the shaders:

- "The vzParameterVolumeTexture Parameter"

- "The vzParameterLookupTable Parameter"

- "The vzParameterVec3f Parameter"

For details on the specific methods, refer to the man pages of the individual classes.

The vzParameterVolumeTexture Parameter

The vzParameterVolumeTexture class provides a simple abstraction of a 3D texture and its position in 3D space. This section describes each of the components of the class by looking at the constructor for the class. The following is the constructor:

```
vzParameterVolumeTexture( const int dataDimensions[3],
                          const int dataROI[6],
                          void* dataPtr,
                          vzTextureType dataType,
                          vzExternalTextureFormat externalFormat,
                          vzInternalTextureFormatinternalFormat=
                          VZ_DEFAULT_INTERNAL_FORMAT);
```

The `dataDimensions` values are the dimensions of the texture data along the X, Y, and Z axes, respectively. The `dataROI` value specifies a cuboidal region of interest (ROI) "contained" within the volumetric data. This will be useful if, for example, you have a

volumetric data of size 256 x 256 x 256 and you want to render texture data of size 128 x 128 x 128 starting at offsets (64, 64, 64). This can be done simply by choosing a `dataROI` defined as in the following:

```
int dataROI[6] = {64, 64, 64, 191, 191, 191};
```

This prevents you from having to create a separate buffer for the subtexture and then copying the data over to it. TMRenderAction will use only the data that lies in the data ROI for all subsequent operations.

The `dataPtr` value specifies the actual texture data. The `dataType` value specifies the type of the texture data stored in the `dataPtr` variable (unsigned byte, integer, float, etc.), while the `externalFormat` value specifies the format of the data (luminance, RGBA, etc.). One can also specify the internal format to be used for the OpenGL texture. The internal format is the format used internally by OpenGL to store the texture in texture memory. The texture data has to be specified in a row-major order, as when creating a 3D texture in OpenGL using the **glTexImage3D()** function call. For example, if the external format is RGBA, the data should be stored as in the following:

```
{ {R1, G1, B1, A1}, {R2, G2, B2, A2}, .....}
```

Note the following:

- The texture data is only "shallow copied" by the API. This means that there is no allocation done internally for the texture data. The class just stores the data pointer and uses it for all subsequent operations.

- The texture data can be modified by using the **setDataPtr()** method. This call would force TMRenderAction to reload the texture into texture memory before using the texture again.

- The `dataDimensions`, `dataROI`, `dataType`, `externalFormat`, and `internalFormat` values of a texture **cannot** be modified once the texture has been created. In order to change any of the above, you will need to create a new texture and use the **setParameter()** method of the shape's appearance to use the new texture.

- The texture dimensions **do not** need to be powers of two as required by OpenGL. TMRenderAction will internally pad the texture data to create the appropriate power-of-two texture.

- The complete texture need not fit in texture memory. If the texture does not fit in texture memory, TMRenderAction will break the texture into smaller bricks internally and use them to create the actual OpenGL textures.

- If a default value is used for the internal format, then the render action would infer a suitable value from the data type and external format of the texture.

In addition to specifying the texture data for the 3D texture, the vzParameterVolumeTexture class also contains information for mapping the texture data to geometry space. This mapping is specified by the `geometryROI` parameter of the volume texture. The geometry ROI of the texture represents the bounding box for the region in world space to which the texture maps. Figure 4-9 illustrates the relationship between the data ROI and the geometry ROI of a texture.
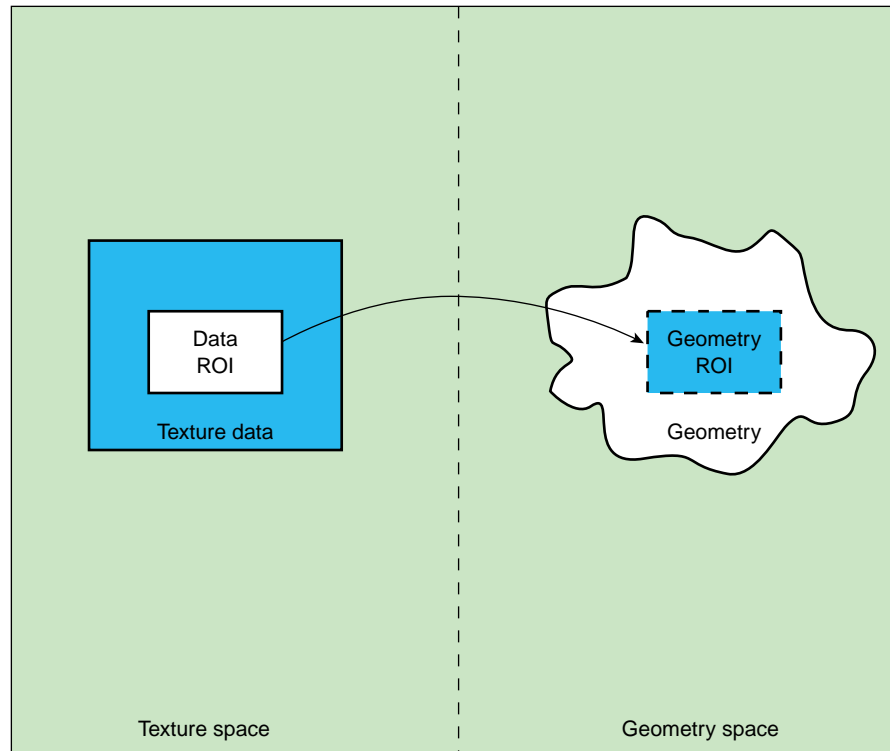
Figure 4-9   Data ROI and Geometry ROI of a Texture

The geometry lying outside the geometry ROI is clipped out by TMRenderAction using clipping planes. If a particular OpenGL clipping plane is enabled before calling the **draw()** method, then TMRenderAction uses software clipping planes to clip the geometry. Otherwise, it uses OpenGL clipping planes to do the clipping. This allows you

to use OpenGL clipping planes in your application. The values for the geometry ROI are set to (0, 0, 0) to (1, 1, 1) by default inside the constructor. Try adding the following lines of code to the sample program in Chapter 2:

```
// Get the parameter "volume" from the shape's appearance
vzParameter *parameter =shape->getAppearance()->getParameter("volume");

// Cast the parameter to a vzParameterVolumeTexture
vzParameterVolumeTexture *texture =
                                (vzParameterVolumeTexture*)parameter;

// Set the geometryROI for the texture
double geometryROI[6] = {0.25, 0.25, 0.25, 0.75, 0.75, 0.75};
texture->setGeometryROI(geometryROI);
```

Figure 4-10 shows the original texture and the modified texture. This illustrates how you can arbitrarily scale and translate your texture to fit the shape's geometry.
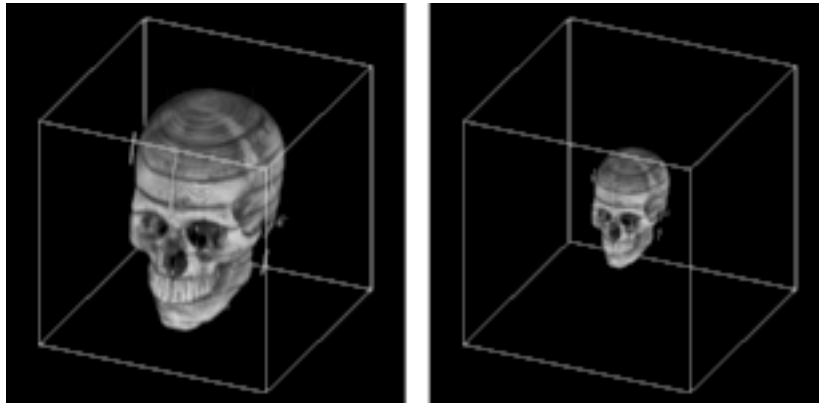

Figure 4-10   Original Texture and Texture after Modifying the Geometry ROI

Note the following:

- If specified, only the data ROI gets mapped to the geometry ROI and not the entire texture.

- The voxel samples along the border of the data ROI are mapped so that they lie exactly along the boundaries of the geometry ROI.

The vzParameterLookupTable Parameter

The vzParameterLookupTable class provides a mechanism for specifying transfer functions to be applied to the volume texture. A transfer function provides the mapping from data values to color values. In this case, it provides the mapping from texel values in the volume texture to color values to be rendered. Using transfer functions, you can visually "remove" unwanted values from the volume rendered image by setting an alpha value of zero for such values. Similarly, you can emphasize other values by giving them different colors and high opacity values. This could be used, for example, to see only the skull from the head data set by assigning an opacity of zero to the other components. Figure 4-11 shows a head image along with its lookup table.

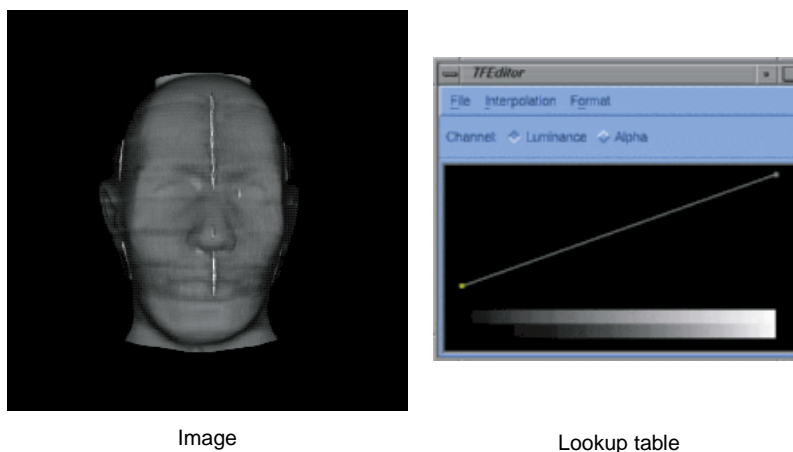Image   Lookup table

Figure 4-11   Head Image and Its Lookup Table

Figure 4-11 was generated using both the transfer function editor and the demo code provided with OpenGL Volumizer.

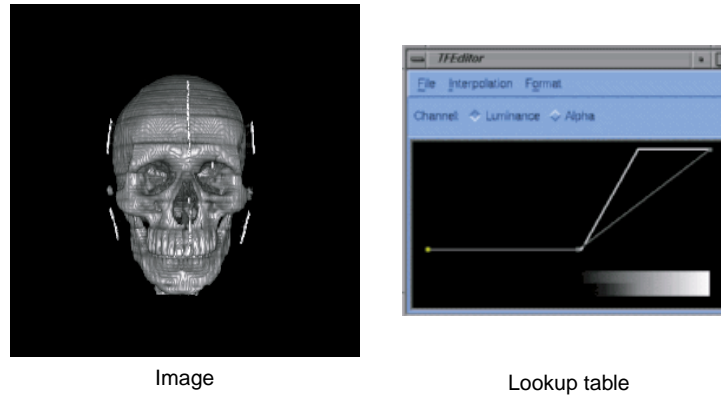Figure 4-12 shows the skull of the head along with its lookup table.

Image   Lookup table

Figure 4-12 Skull of the Head and Its Lookup Table

TMRenderAction implements the transfer function using post-interpolation lookup tables. These lookup tables get applied in the imaging pipeline after the texture interpolation stage. The interface for specifying the lookup table is similar to that of the vzParameterVolumeTexture parameter since a lookup table can be thought of as a one-dimensional texture. The constructor for the class looks like the following:

```
vzParameterLookupTable( int width,
                        void* dataPtr,
                        vzTextureType dataType,
                        vzExternalTextureFormat externalFormat);
```

The `width` value specifies the number of entries in the table. The `dataPtr` value is the address of the table entries in memory. The `dataType` and `externalFormat` values specify the data type and format, respectively, similar to that of a vzParameterVolumeTexture parameter.

Note the following:

- Unlike the vzParameterVolumeTexture parameter, the width of the lookup table **must** be a power of two.

- The `dataPtr`, `dataType`, and `dataFormat` values of a lookup table can be modified once it is created. For any of these modifications, the table would be reloaded.

- Like the vzParameterVolumeTexture parameter, the `dataPtr` value is shallow copied—that is, no memory is allocated internally for the data. Also, the data should be specified in an interleaved format similar to that of the volume texture.

- The maximum size of the lookup tables on InfiniteReality systems is 1024 for `RGBA`, 2048 for `LUMINANCE_ALPHA`, and 4096 for `INTENSITY` formats.

The vzParameterVec3f Parameter

The vzParameterVec3f class is used to specify a vector of three floating point values. It is used by vzTMTangentSpaceShader to specify the light direction for the volumetric lighting. It can potentially be used by other shaders that require parameters such as color values, material properties, and so on. The constructor is simply the following:

```
vzParameterVec3f( );
```

The vector is given a default value of (1, 0, 0). You can modify the value by using the **setValue()** method of the class.

A Closer Look at TMRenderAction

TMRenderAction implements the 3D texture slicing technique (described earlier in section "Algorithm Used by TMRenderAction") to render volumetric shapes. This section explains some of the details of the render action and mentions a few techniques that you can employ for added functionality and performance. Included are the following subsections:

- "The Volume Rendering Pipeline"

- "Texture Management"

- "Sampling Rate"

- "Arbitrary Polygonal Geometry"

The Volume Rendering Pipeline

Figure 4-13 shows the pipeline used by a typical volume rendering application using the render action.

Figure 4-13   Volume Rendering Pipeline

First, the application computes the number of shapes it needs to keep resident in texture memory for the given frame. The list of shapes might be the outcome of visibility culling in an immersive application, the current frame index of a time-varying simulation, or the like. Note that it is not necessary to draw all shapes that are managed, but a shape that needs to be drawn must be managed.

Next, it is the application's responsibility to sort the rendered shapes in the correct order since TMRenderAction does not perform any visibility sorting of the rendered shapes. After the sort, the application sets the appropriate OpenGL state, such as enabling blending and setting the appropriate blending functions, for performing volume rendering. TMRenderAction renders the polygonal geometry in a back-to-front sorted order. Hence, the blending function for the most common volume rendering case would be the over operator **glBlendFunc(**GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**)**. The following is a typical example of the OpenGL state settings using the over operator:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The flexibility in choosing the blending function allows you to implement other techniques. For example, you can implement Maximum Intensity Projection by using the blending equation **glBlendEquation(**GL_MAX**)**. See the man page for **glBlendEquation()** for a complete list of modes.

After these two steps, the application lets the render action know that it is ready to start drawing the shapes by calling **beginDraw()**. The **beginDraw()** method marks the end of the texture management phase and the beginning of the rendering phase. Inside the method, the render action does the following:

- Computes total resources required for the list of managed shapes.

- Manages the OpenGL state (push application's OpenGL state, store transformation matrices, etc.).

- Manages the OpenGL resources (creates and downloads texture objects, lookup tables, etc.).

Then the application draws all the shapes in the visibility sorted order just described in the preceding paragraphs. Inside each draw method, the render action does the following:

- Invokes the shader's initialization routine, which sets the appropriate OpenGL state (bind texture objects, enable lookup tables, etc.).

- Polygonizes the volumetric geometry using the transformation matrices.

- Draws the polygonized geometry in a back-to-front order.

Note that the polygonized geometry is always parallel to the viewport, unless the application has set slicing planes on the volumetric geometry. The transformation matrices are queried directly from OpenGL in the **beginDraw()** method. These matrices are stored and used for all the subsequent draws before the next **endDraw()** call.

Finally, in the **endDraw()** method, the render action restores the OpenGL state that it has modified. This includes texture related settings, lookup tables, and pixel store.

Texture Management

Texture memory is a very valuable resource that needs to be managed very efficiently if one is to achieve interactive rates for volume rendering using 3D texture mapping. TMRenderAction makes this job easier for you by hiding all the machine-specific details of texture management and giving you transparent access to the graphics hardware. The render action optimizes the texture management process by using techniques to prevent fragmentation of texture memory and optimizing the flow of texture data to the graphics subsystem.

The following subsections provide some specific details of the texture management performed by TMRenderAction:

- "Texture Dimensions and Sizes"

- "Custom Bricking of Textures"

- "Texture Memory Usage"

- "Intelligent Texture Management"

- "Texture Interleaving"

Texture Dimensions and Sizes

TMRenderAction allows specifying textures of arbitrary dimensions and sizes using the vzParameterVolumeTexture class. All texture dimensions have to be powers of two for the textures to be valid. Also, the texture size should be less than or equal to the amount of texture memory available on the graphics subsystem.

TMRenderAction removes this restriction by appropriately padding the textures of invalid dimensions to the next higher power-of-two dimensions. Also, TMRenderAction is capable of virtualizing textures that are too big to fit in texture memory. Requiring no intervention in brick creation, management, and sorting, all of these processes are transparent to you.

Custom Bricking of Textures

For some applications, you might want to implement your own bricking of the texture data. In this case, you will have to create one vzShape per brick. Each of these shapes will contain one volume texture corresponding to the texture data for the brick. Once the shape is created, you should manage, unmanage, and draw these shapes as required. TMRenderAction will try to optimize the texture management, depending on the total size of the textures that you have created.

For your custom bricking, you should make sure that the geometry ROIs of the texture bricks are such that the boundaries match with those of the adjacent bricks. You should invoke the draw function in such a manner that the shapes are rendered in a back-to-front sorted order. TMRenderAction assumes linear filtering of textures; so, you should have a one-voxel overlap between the adjacent textures. Figure 4-14 illustrates this in 2D.
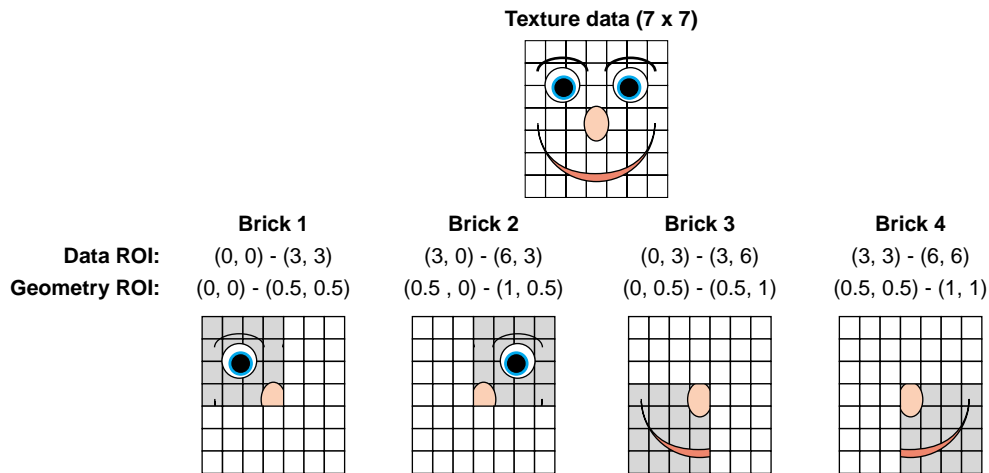
Texture data (7 x 7)

	Brick 1	**Brick 2**	**Brick 3**	**Brick 4**
Data ROI:	(0, 0) - (3, 3)	(3, 0) - (6, 3)	(0, 3) - (3, 6)	(3, 3) - (6, 6)
Geometry ROI:	(0, 0) - (0.5, 0.5)	(0.5 , 0) - (1, 0.5)	(0, 0.5) - (0.5, 1)	(0.5, 0.5) - (1, 1)

Figure 4-14 Texture Bricking

Figure 4-14 shows a 7 x 7 texture, which is divided into 4 bricks of size 4 x 4 each. These textures use the same data pointer of the original texture and do the bricking by using a different data ROI for each of the bricks. The first row gives the data ROIs of each of the bricks. In order for the brick boundaries to match, you need to adjust the geometry ROIs of each of the bricks so that they match on their boundaries. The second row gives potential values for the geometry ROIs of each brick.

Texture Memory Usage

TMRenderAction by default uses all of the texture memory available on the graphics subsystem. It uses `GL_PROXY_TEXTURE_3D` to figure out the amount of texture memory available on the system.

Intelligent Texture Management

Understanding the texture management can help you improve the performance of the rendering by the render action in many common cases. TMRenderAction computes the total amount of resources required to render the given set of managed shapes in the **beginDraw()** call and compares it to the amount available on the graphics pipe. Depending on the outcome of the comparison, the render action uses different texture management schemes. One optimization common to all the schemes is that the render

action tries to reuse OpenGL texture objects whenever possible. Note the sequence of frames in Figure 4-15.
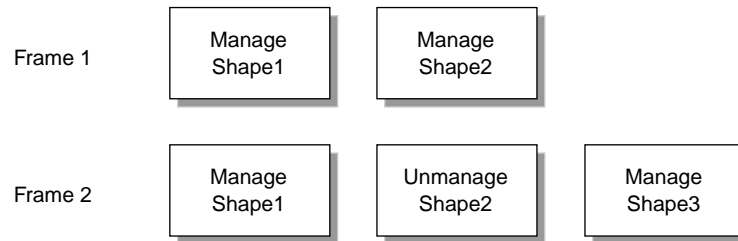
Frame 1	Manage Shape1	Manage Shape2	
Frame 2	Manage Shape1	Unmanage Shape2	Manage Shape3

Figure 4-15   Reusing Texture Objects

In the first frame, the render action would allocate OpenGL texture objects for Shape1and Shape2. In the second frame, even though Shape2 is not managed, the render action does not delete the texture objects for it. Instead, it reuses the texture objects for downloading and binding the textures in Shape3. This has two advantages. First, reusing texture objects prevents fragmentation of texture memory, since not all texture managers do garbage collection immediately after the texture object has been deleted. Second, for downloading the textures in Shape3, the render action uses **glTexSubImage3D()** calls, which are considerably faster than the corresponding **glTexImage3D()** calls.

 The preceding discussion assumes that the textures in the shapes fit in texture memory and have the same data ROI dimensions and internal texture formats. Hence, if your application uses multiple shapes and needs to constantly manage and unmanage them in order to improve the download performance of your application, you should try to divide the whole scene into multiple shapes such that the textures in the shapes are all of equal sizes. Typical examples of such applications are volume roaming, multiresolution volume rendering, and time-varying volumes.

You can use the **manage()** and **unmanage()** methods to do predictive texture downloads of volumetric textures. For example, you could manage a shape in frame n which you need to render in frame $n+1$. This process can help you split the cost of downloading the textures over multiple frames. This can be very useful for applications like volume roaming, time-varying volumes, and the like.

Texture Interleaving

Note: This section is intended for advanced users.

On InfiniteReality graphics systems, the smallest texel supported by the hardware is 16 bits. Hence, even if your textures are single-byte textures, they would end up taking twice the amount of texture memory. Texture interleaving allows you to efficiently fill up the space in texture memory using the texture-select extension.

Texture interleaving has two main benefits:

1. Efficient use of texture memory

    Texture interleaving allows you to use all the texture memory available on InfiniteReality systems. This would not be true if you had single-byte LUMINANCE textures rendered with an internal format of VZ_INTENSITYn (where n can be 8, 12 or 16).

2. Increase in texture download rate

    With an internal texture format of VZ_DUAL_INTENSITY8, the texture download takes only half the time as compared to the format VZ_INTENSITYn (where n can be 8, 12 or 16).

TMRenderAction currently supports interleaving of LUMINANCE textures using either two-way (DUAL) or four-way (QUAD) interleaving. Interleaving can be used in multiple ways depending upon the application. The following are the three ways that interleaving can be used with TMRenderAction:

*   Transparent interleaving

    If you create a vzParameterVolumeTexture with an external format of VZ_LUMINANCE and data type of VZ_BYTE or VZ_UNSIGNED_BYTE; then, on InfiniteReality systems, TMRenderAction would internally interleave the texture data and use it to download and bind the appropriate texture. Requiring no interference from you, this process is completely transparent to the application. This, however, can have some computational disadvantages in dynamic applications such as time-varying volumes because the interleaving process itself can be slow. For such applications, you can prevent the render action from interleaving the textures by specifying the appropriate internal texture format—for example, VZ_INTENSITY16 instead of VZ_DEFAULT_INTERNAL_FORMAT.

- Forced interleaving

  In order to avoid the cost of interleaving every time you manage a texture, you can force the render action to interleave the texture data and store the results. This can be done by specifying the desired internal texture format—for example, `VZ_DUAL_INTENSITY8` or `VZ_QUAD_INTENSITY4`—and calling the method **optimize()** on the vzParameterVolumeTexture after creating it. The results of the interleaving process will be stored in the texture and will be available to the render actions for all subsequent operations. If you use the internal format of `VZ_DEFAULT_INTERNAL_FORMAT`, then an appropriate internal format will be inferred from the external data format and type.

- Pre-interleaved textures

  You can also provide pre-interleaved texture data to the render action. In this case, it is the application's responsibility to interleave the texture data and provide the appropriate internal and external texture formats. Also, the texture data should be compliant with the texture specifications of OpenGL. For example, the textures should fit in texture memory and should have power-of-two dimensions. The sample code in `VZROOT/src/lib/apps/appsUtil/` demonstrates how to create interleaved textures from a given input texture.

Note: In the interleaving interface, the interleaving is done within the same texture and the data is rendered appropriately. The render action assumes the texture is decomposed into two textures along the X dimension of the data. Rendering with these textures involves sorting and using the appropriate OpenGL state settings, but this procedure is completely transparent to the application, even for pre-interleaved textures.

Sampling Rate

The sampling rate used to polygonize the volumetric geometry controls the number of slices that are used to render the shape. Theoretically, the minimum data slice spacing is computed by finding the longest ray cast through the volume in the view direction, and then finding the highest frequency component of the texel values and using double that number for the minimum number of data slices for that view direction. Practically, the rendering process tends to give a pixel-fill limitation; and, in many cases, choosing the number of data slices to be equal to the volume's dimensions, measured in texels, works well. Trading performance and image quality can be a key issue for numerous applications.

You can control the sampling rate by setting the appropriate value using the **setSamplingRate()** method. By default, TMRenderAction uses a sampling rate of (1, 1, 1), which implies that the slicing is done once per voxel along each of the data dimensions. This default usually provides acceptable image quality.

However, when zooming into the volume data, you might see artifacts due to undersampling in the image space. In order to remove this, you might need to increase the sampling rate accordingly. Varying the sampling rate is also necessary for anisotropic data to compensate for the difference sampling rate along the various data dimensions. The sample medical data set in the following file is an example of such a data set:

```
VZROOT/data/medical/Phantom/CT.Head.char.tif
```

Using a sampling rate of (1, 1, 3.32) would usually give better image quality for this data set.

Arbitrary Polygonal Geometry

You can render arbitrary polygonal geometry with the shape's `volume` texture applied to it by using the vzPolyGeometry class described in "Arbitrary Polygonal Geometry" in Chapter 3. When the draw method is invoked on the vzPolyGeometry, the appropriate geometry ROI for the polygonal data is passed with the method. The render action also sets the appropriate OpenGL state, including 3D texture state and clipping planes, before calling the draw method. So, if the shape's appearance used vzTMSimpleShader or vzTMLUTShader, the corresponding `volume` texture will still be bound with the appropriate lookup tables and texgen settings. Using this scheme, applications can implement the spherical sampling technique (described in the following paragraphs) by rendering the appropriate tessellated shells after the corresponding draw. There is one notable caveat, however: the technique would not work correctly with multipass shaders like vzTMTangentSpaceShader.

Slicing with planes is common but artifacts can appear when the observer is very close to the model. As an implementation alternative, spherical slicing provides a more accurate visualization in perspective projection. Figure 4-16 illustrates the principle.
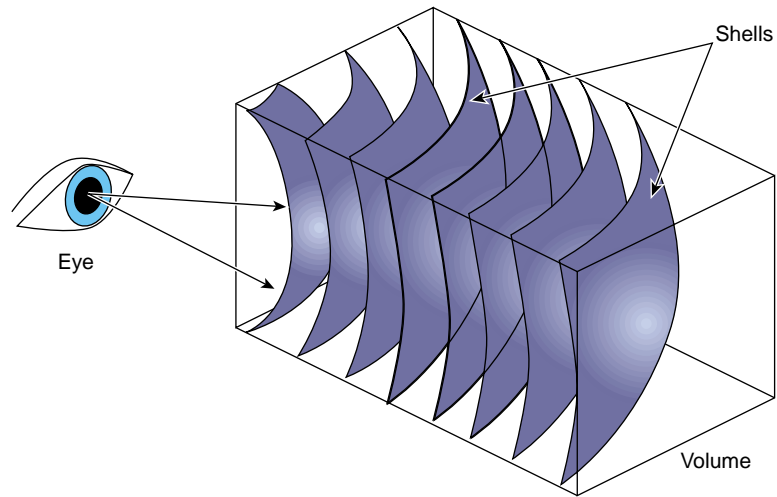
Figure 4-16   Spherical Slicing

In this case, the polygonization process might become the performance bottleneck. Using a parallel algorithm to perform the polygonization on multiple processors will help maintain a good level of performance.

Custom Volumetric Shading

The built-in shaders, described earlier in section "Using the Built-in Shaders" in Chapter 4, implement the more common volumetric shading models. These built-in shaders are cross-platform and are optimized for SGI visualization systems. To implement less common shaders and to support more graphics platforms, OpenGL Volumizer allows you to write custom shaders.

All lighting and shading has to be implemented in a fragment processing pipeline. The built-in shaders all use multiple passes with OpenGL's fixed-function pipeline. In your custom shaders, you can also use this multipass, fixed-function pipeline approach, or you can use per-fragment computations for programmable GPUs.

This chapter describes how you can write your own custom shaders. The supported custom shaders can be characterized as follows:

- Multipass, callback-based shaders
- Interactive Shading Language (ISL)-based shaders for IRIX and 32-bit Linux systems
- Hardware-specific, GPU-based shaders

In general, OpenGL Volumizer provides the following functions:

- Polygonization
- OpenGL state management
- Data management

The following sections describe what you must provide to implement custom shaders.

Multipass, Callback-Based Shaders

Using multiple passes with different OpenGL states set before the various passes, the built-in shaders render the polygonized geometry. Note that a built-in shader, not the application, handles the multiple passes and OpenGL states. The polygonized geometry is essentially the set of triangles corresponding to each slice of the volume data. Rendering each slice using different OpenGL states allows you to create shading effects that are otherwise difficult to generate using the OpenGL hardware. The SGI OpenGL Shader uses the same approach for surface shading.

TMRenderAction provides a built-in shader class that allows you to write your own custom, multipass shaders. The vzTMShader class provides such an interface to apply arbitrary shading to shapes being rendered using the render action. The class provides a set of callbacks that are invoked by the shader while rendering the polygonized geometry. These callbacks can be used to set the appropriate OpenGL state for the rendered geometry. There are three types of callbacks used by vzTMShader:

- Shape callbacks

- Slice callbacks

- Bind-Parameter callbacks

The section describes these callbacks and provides guidelines for writing your custom shaders.

Shape and Slice Callbacks

The shape callbacks are invoked before and after the current shape is rendered. The shader typically uses the pre-render shape callbacks to set the OpenGL state, which does not change while rendering the shape. This might include enabling 3D textures, binding lookup tables, and the like. The pre-render shape callback can also be used to do error checking. For example, the shader can verify the list of required parameters in the current appearance (described later in this section). The post-render shape callback can be used to restore the OpenGL state modified by the shader.

The slice callbacks are invoked before each slice of the polygonized geometry is rendered. These callbacks are used to set the OpenGL state, which needs to be modified for every slice. The OpenGL state includes the currently bound texture, enabling and disabling blending, modifying the color mask, and the like.

The shape and slice callbacks can be set using the following methods (see man pages for vzTMShader for more details):

```
void setShapeCallbacks(vzTMShaderCB preCB, vzTMShaderCB postCB, void*
userData);
```

```
void setSliceCallbacks (int numCB, vzTMShaderCB* callbacks, void*
userData);
```

The **vzTMShaderCB** callback is defined as the following:

```
void (*vzTMShaderCB)(vzTMShaderData *shaderData);
```

Bind-Parameter Callbacks

The shader can request TMRenderAction to "bind" the shader parameters using the **vzTMBindParameterCB** callback function, defined as the following:

```
bool (*vzTMBindParameterCB)(const char *name, vzTMShaderData
*shaderData, vzTMShaderOp operation = VZ_TM_BIND);
```

The bind-parameter callbacks accept the following three arguments: the name of the parameter to be bound, shader data, and the shader operation to be applied on the given parameter. If the shader operation is `VZ_TM_ENABLE` or `VZ_TM_DISABLE`, the corresponding OpenGL state is simply enabled or disabled, respectively. The default shader operation is `VZ_TM_BIND`, which binds the given parameter.

The bind-parameter callbacks can be invoked from the shape or slice callbacks to bind the appropriate parameters. The bind-parameter callbacks return a true value if the call succeeds. Otherwise, it returns a false value. The `vzTMShaderData` argument passed to the callbacks contains shading information. The `vzTMShaderData` argument has the following data members:

```
vzTMBindParameterCB bindVolumeTextureCB;
vzTMBindParameterCB bindLookupTableCB;
vzAppearance* appearance;
void* userData;
```

TMRenderAction primarily manages two OpenGL resources for the application. These resources are the shader parameters corresponding to the texture data (usually vzParameterVolumeTexture) and the transfer functions (vzParameterLookupTable). These parameters can be bound using the **bindVolumeTextureCB** and **bindLookupTableCB** callbacks. The `appearance` structure can be used to access the other parameters required by the shader.

While the two bind-parameter callbacks can be used to bind the appropriate parameters, the current appearance can be used to access the other parameters required by this shader. The user-data pointer can be used to store arbitrary user data that the shader might need. The shader-data pointer needs to be passed back to the render action as an argument to the bind-parameter callbacks.

The following code shows an example of a shader that blends two volume textures together. This is done using two slice callbacks that bind the different volume textures. The following are pre-render and post-render shape callbacks that enable and disable 3D texturing, respectively:

```
void preShape(vzTMShaderData *shaderData) {
shaderData->bindVolumeTextureCB("volume", shaderData, VZ_TM_ENABLE);
}
void postShape(vzTMShaderData *shaderData) {
shaderData->bindVolumeTextureCB("volume", shaderData,VZ_TM_DISABLE);
}
```

The following are the two slice callbacks that bind the different volume textures:

```
void pass1(vzTMShaderData *shaderData) {
    if(!shaderData->bindVolumeTextureCB("volume", shaderData))
      vzError::error(VZ_OPERATION_FAILED,"Could not bind volume
        texture 'volume'");
}
void pass2(vzTMShaderData *shaderData) {
    if(!shaderData->bindVolumeTextureCB("volume2", shaderData))
        vzError::error(VZ_OPERATION_FAILED,"Could not bind volume
        texture 'volume2'");
}
```

Guidelines for Creating Your Shaders

Use the following guidelines when creating your shaders:

- Try to minimize the number of state settings in the callbacks. For example, do not bind the same parameter repeatedly in the slice callbacks if this is not necessary.

- All the shaders need to have a parameter of type `vzParameterVolumeTexture` and name `volume`. TMRenderAction uses this parameter to compute the number of slices from the geometry ROI and sampling rate.

- Each appearance can have any number of volume texture parameters but only one lookup table parameter.

- The data ROI and geometry ROI of all the volume textures in the appearance should be the same. Different data ROI and geometry ROI would work correctly if the shape is not bricked internally.

Interactive Shading Language (ISL)-Based Shaders

Note: This section pertains only to IRIX and 32-bit Linux systems.

The SGI OpenGL Shader provides a high-level language, ISL, for custom shading. Unlike the multipass, callback-based shaders just described, ISL-based shaders do not require you to change your application. You place your shading descriptions in an external file that is loaded by the application at run time. These run-time descriptions can be used to control the operations that can be applied to the data while it is rendered.

OpenGL Volumizer Implementation

TMRenderAction uses the vzTMISLShader class to interface with ISL. The ISL interface provides the following features:

- Placement of the run-time shading description in a text file

- Use of application parameters

- Transparent OpenGL state management application-wise

- Cross-platform code

The TMISLShader provides a static **load()** method for loading shader files written in ISL format. The following code creates a TMISLShader by loading it from file `fileName`:

```
// Load the ISL shader from file 'fileName'
vzShader *shader = vzTMISLShader::load(fileName);
```

The TMISLShader also provides an alternative constructor that accepts an islAppearance as its only argument. This interface is useful if you want to use the OpenGL Shader API to create a shader and then apply it to an OpenGL Volumizer appearance.

In either case, the application needs to ensure that the set of parameters used by the shader are attached to the shape's appearance.

ISL Examples

Example 5-1 shows a simple ISL code segment that takes the two textures `volume` and `volume2` as input and then computes and renders their per-voxel product.

Example 5-1   A Simple ISL Code Segment

```
surface multiply() {
   varying color tex1 = FB;
   varying color tex2;

   // sample texture 'volume'
   FB = texture("volume");

   // sample texture 'volume2' and multiply to above
   FB *= texture("volume2");

   // do the final blend with frame buffer contents
   tex2 = FB;
   FB = tex1;
   FB = under(tex2);
}
```

Figure 5-1 illustrates the texture multiplication described in Example 5-1.
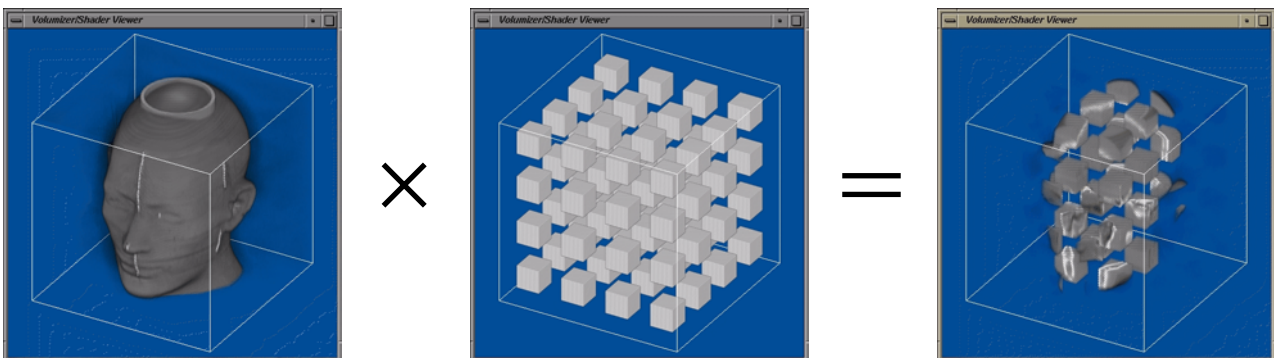
Figure 5-1   Run-Time Multiplication of Two Textures

Example 5-2 shows a more complex ISL example that uses conditionals to apply a per-voxel threshold to selectively remove parts of the rendered volume.

Example 5-2   A Complex ISL Example

```
surface threshold() {
   uniform color trans = color(0, 0, 0, 0);
   varying color tex1 = FB;
   varying color tex2;

   // apply the texture tag
   FB = texture("tag");

   // See if the tag value is below the threshold
   if (FB[3] < 0.5)
       FB = texture("volume");
   else
       FB = trans;

   // final blend with the frame buffer contents
   tex2 = FB;
   FB = tex1;
   FB = under(tex2);
}
```

Hardware-Specific, GPU-Based Shaders

Rendering volume datasets differs from polygonal rendering. The traditional OpenGL lighting and shading models for polygonal rendering do not apply to rendering volume datasets. In the context of OpenGL Volumizer, you can think of shaders as a programmable interface to the GPU. All lighting and shading has to be implemented in the fragment processing pipeline using a fragment program (per-fragment computations).

GPU-based shaders have the following features and constraints:

- Placement of the run-time shading description in a text file

- Use of application parameters

- Application management of the OpenGL state

- Platform support of the OpenGL `ARB_fragment_program` extension

  To check for the presence of this extension on IRIX and Linux systems, enter the following:

  $ **glxinfo | grep -i ARB_fragment_program**

OpenGL Volumizer Implementation

TMRenderAction provides the vzTMFragmentProgram class to interface with GPUs. The OpenGL Volumizer programming model uses the following three callbacks to structure the shading:

Callback	Description
Pre-shader callback	Sets the OpenGL state. Binds non-texture parameters.
Texture unit callback	Binds the texture parameters for a texture unit.
Post-shader callback	Restores the OpenGL state.

The following code illustrates the structure:

```
void preShader(){
   // bind 'diffuse' parameter
   // set other OpenGL state
}
void unit1() {
   // bind 'lookup_table' parameter
}
void unit2() {
   // bind 'volume' parameter, e.g. fMRI
}
void unit3() {
   // bind 'volume2' parameter, e.g. CT
}
void postShader() {
   // restore OpenGL state
}
```

To provide the per-fragment computations, you can use ISL from OpenGL Shader or a fragment program using the OpenGL `ARB_fragment_program` extension.

The TMFragmentProgram provides a static **load()** method for loading fragment programs written in the language specified by the `ARB_fragment_program` extension. For the complete specification of the `ARB_fragment_program` extension, see http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt. The following code creates a TMFragmentProgram by loading it from file `fileName`:

```
// Load the fragment program file 'fileName'
vzTMFragmentProgram *fp = vzTMFragmentProgram::load(fileName);
```

In addition to specifying the fragment program, the application also needs to provide the shader parameters that are used by the fragment program. These parameters can be accessed inside the fragment program either using textures or local program parameters. The TMRenderAction manages the textures used in the program transparently, but the application needs to bind them to the appropriate texture units using the multitexture callbacks. The application can also set other OpenGL state or shader parameters required by the fragment program in the pre-shape callback. See Example 5-3 for a sample program.

You can also use the XML file format to load and initialize appearances with fragment programs. For more information, see section "Loading Fragment Programs" on page 118.

Fragment Program Examples

Example 5-3 shows a simple fragment program that takes the two textures `volume` and `volume2` as input and then computes and renders their per-voxel product.

Example 5-3   A Simple Fragment Program

```
!!ARBfp1.0
# Temporary variables
TEMP volume;
TEMP volume2;
# Sample textures 'volume' and 'volume2'
TEX volume1, fragment.texcoord[0], texture[0], 3D;
TEX volume2, fragment.texcoord[1], texture[1], 3D;
# Multiply the results of 'volume' and 'volume2'
MUL result.color, volume2, volume;
END
```

Notice that the `volume` texture is the 3D texture bound on texture unit 0 while the `volume2` texture is bound on unit 1. This can be initialized using the following multitexture callbacks:

```
// Allocate memory for the multitexture callbacks
vzTMShaderCB *multiTexCB = new vzTMShaderCB[2];

// Initialize the multitexture callbacks for the two texture units
multiTexCB[0] = texUnit0;
multiTexCB[1] = texUnit1;

// Set the callbacks for the shader.
fp->setMultiTextureCallbacks(2, multiTexCB, fp);
```

The multitexture callbacks used in the preceding code sample are very similar to those used by the vzTMShader class and might look like the following:

```
void texUnit0(vzTMShaderData *data) {

    // enable texture 'volume'
    data->bindVolumeTextureCB("volume", data, VZ_TM_ENABLE);

    // bind texture 'volume1'
    data->bindVolumeTextureCB("volume", data));
}

void texUnit1(vzTMShaderData *data) {
    // enable texture 'volume2'
    data->bindVolumeTextureCB("volume2", data, VZ_TM_ENABLE);

    // bind texture 'volume2'
    data->bindVolumeTextureCB("volume2", data));
}
```

Note: Graphics systems using graphics processing units (GPUs) like those on Onyx4 or Silicon Graphics Prism systems do not support texture lookup tables (TLUTs) in the hardware. However, LUTs can be implemented using dependent textures, where the sampled texel values of the volume texture can be used as the texture coordinates for sampling the 1-dimensional, dependent texture.

Example 5-4 shows how to implement LUTs using a dependent texture.

Example 5-4   Implementing LUTs Using Dependent Textures

```
!!ARBfp1.0
# Temporary variable for storing sample volume texel values
TEMP volume;
# Sample 3D "volume" texture bound on texture unit 1
TEX volume, fragment.texcoord[1], texture[1], 3D;
# Sample 1D "lookup_table" texture bound on texture unit 0
# Use values from above instruction as texture coordinates
TEX result.color, volume, texture[0], 1D;
END
```

Example 5-5 shows a more complex fragment program that uses a conditional to apply a per-voxel threshold to selectively remove parts of the rendered volume.

Example 5-5   A Complex Fragment Program

```
!!ARBfp1.0
# threshold for tag shader
PARAM thresh = {0.5, 0.5, 0.5, 0.5};
# temporary variable for 'tag' texture
TEMP tag;
# sample the 'tag' texture
TEX tag, fragment.texcoord[1], texture[1], 3D;
# subtract threshold value from the 'tag' value
SUB tag, tag, thresh;
# if tag value is smaller than threshold, kill fragment
KIL tag;
# sample 'volume' and output resulting color
TEX result.color, fragment.texcoord[0], texture[0], 3D;
END
```

Note that the blending is controlled by the OpenGL state, which is set by the application before the shader is called. This contrasts with ISL-based shaders, where ISL manages the OpenGL state and, consequently, blending transparently.

A set of ISL shaders and fragment programs are installed under VZROOT/src/shaders/. As an exercise, you can modify them to generate different visual effects.

The Large-Data API: 3D Clip Textures

OpenGL Volumizer includes a new application programming interface (API) for large-data volume rendering. This API allows volume rendering of datasets that exceed main memory and texture memory and addresses bottlenecks associated with pixel-fill performance using a multiresolution rendering scheme.

This chapter contains the following topics:

- "Problems in Large-Data Visualization"
- "3D Clip Textures"
- "Clip Texture Representation: Class vzParameterClipTexture"
- "Clip Texture Rendering: Class vzClipRenderAction"
- "Visualization Pipeline for the Large-Data API"

Problems in Large-Data Visualization

Chapter 4 introduces the render action Texture Mapping Render Action (TMRenderAction). TMRenderAction uses 3D texture mapping hardware to volume render a shape node. TMRenderAction manages the data resident in texture or graphics memory. In the context of volume rendering and 3D texture mapping, when the size of volume data is larger than what the local resources can handle, volume visualization also becomes a data management problem. This chapter shows how OpenGL Volumizer can efficiently manage resources to maintain interactive frame rates.

In the context of 3D texture-based rendering, large data implies that the size of the volume data exceeds one or more of the following:

- Rendering efficiency of the graphics hardware, such as the fill rate limitation of the graphics subsystem
- Amount of texture or graphics memory of the graphics subsystem

- Amount of main memory of the computer system

- System data bandwidth, especially between the various peripherals and the graphics pipe

Current graphics subsystems provide unified graphics memory or dedicated texture memory in addition to necessary framebuffer and ancillary-buffer support. With graphics hardware providing ever larger texture- or graphics-memory configurations, it is possible to render sizable volumes at frame rates approaching near real-time performance (10/15 fps). However, even if the volume data fits in graphics memory, the pixel fill rate of the graphics hardware can limit an application's rendering performance. Moreover, if the size of the data exceeds graphics-memory resources, the data to be visualized will partly reside on slower and larger storage peripherals, such as system memory, and disks. It is the task of the visualization application to manage the data among memory devices within the time constraint in addition to taking into account concerns over pixel fill-rate limitations.

Bricking

Applications can improve volume rendering performance when rendering large data by addressing bandwidth bottlenecks—for instance, during data transfer. One approach is to divide the whole volumetric data into smaller components called *bricks*. Using bricks provides an application more control over frame rates by moving these data bricks to the local texture memory from the various storage devices. This level of control gives applications the capability to visualize huge data located in memory or on high-performance disks by paging them into texture memory using intelligent schemes. Thus, bricking allows applications to page in to an application space as needed smaller units of a large volume that would not otherwise fit into main or graphics memory.

In addition to the usefulness of bricks in the implementation of texture-paging mechanisms, view-frustum culling, and load balancing on multipipe systems, this representation is useful in circumventing the inefficiencies due to the padding of textures to the next higher power-of-two dimensions, as required by OpenGL. Brick size plays an important role in the overall efficiency. Short data transfers may require frequent interrupts in the data flow and can consequently affect performances. On the other hand, long data transfers optimize the overall bandwidth but are not interruptible. The choice of brick size depends upon the hardware architecture; therefore, applications should select values taking system parameters into consideration.

Bricking alone addresses data representation for efficient data transfer. Data visualization techniques used in 3D texture-based rendering applications, such as volume roaming and multiresolution volume rendering can affect application performance as well. These rendering methods use bricking as a base for exchanging data from main memory to texture memory or from disks to main memory.

Volume Roaming

Volume roaming is a technique that allows the user to explore large volumetric data using a volumetric probe, through which users interactively move inside a rendered volume. The probe allows users to navigate the dataset using a viewing window and helps them concentrate on a specific section of the whole dataset. The key components of the technique are texture bricking, intelligent texture and main memory management, and asynchronous disk paging of volume data. The application maintains a hierarchy of windows, which contain smaller subsets of the total volume data and are updated during user motion. Each window is subdivided into multiple shapes, one for each brick. As the window moves, the bricks are updated with new texture data. The application is responsible for controlling all of the window management and data transfer between the various peripherals. TMRenderAction efficiently pages in the new data into texture memory from main memory. Roaming allows an application to overcome fill rate, texture memory, and main memory constraints. The technique renders only a subvolume of data at a time and cannot guarantee constant frame rates during fast user motion.

Multiresolution Volume Rendering

To achieve interactive rendering for a volume dataset of a given size, applications can control the sampling of the data that they render. Making a tradeoff between performance and image quality, multiresolution volume rendering allows applications to interactively render large volume data by rendering bricks that vary in volume levels-of-detail (LOD). Lower resolutions help improve performance since it limits the texture memory as well as the fill rate consumption of the application. To improve rendering performance while maintaining acceptable image quality, applications can do the following:

- Couple texture management with LOD switching to ensure near-constant frame rates.

- Take advantage of the sorted order of bricks to determine the LOD to be rendered.

- Use clipping geometries to optimize the use of texture memory available on the graphics subsystem.

- Progressively render higher-resolution volumes during minimal stages of user interaction or for bricks closer to the user.

Using LODs requires the volume data to be reformatted at different resolutions by filtering and decimation to produce a more compressed representation. Because this compression is usually lossy, the rendering process trades off interactivity against rendering performance and image quality against visualization accuracy.

The key components of multiresolution volume rendering include texture bricking, intelligent texture memory management, and proper computation of the various LODs. Applications can improve the performance by rendering low-resolution data during user interaction and then successively improving the image quality during an absence of user interaction. In this case, a shape is used to represent each node in the octree. The TMRenderAction manages the texture data and multiple lookup tables used to compensate for the different opacities at the LODs. Low resolutions help improve rendering performance by limiting texture memory and fill-rate consumption of the application. A limitation of this technique is that all of the volume data along with the various LODs need to be present in main memory, thereby constraining the size of the dataset that can be rendered using this method.

OpenGL Volumizer provides a large-data API: an interface to a hierarchy of 3D clip textures and their associated renderer. 3D clip textures allow applications to visualize arbitrarily large volumetric data by merging the advantages of volume roaming and multiresolution techniques.

3D Clip Textures

While multiresolution volume rendering and volume roaming are attractive techniques for rendering volumetric data, they are restricted when dealing with extremely large datasets. 3D clip textures allow applications to visualize arbitrarily large volumetric data by combining the advantages of bricking, volume roaming, and multiresolution techniques.

Figure 6-1 illustrates the concept of 2D clip textures. 2D clip textures have been used successfully to provide interactive navigation of very large terrain data. Clip textures are mipmap versions of the original texture data, except that each mipmap level maintains a roaming window (physical memory window in Figure 6-1) to limit the amount of texture

data resident in main memory. These clipped mipmap levels are called clip levels. The highest level of resolution in the hierarchy corresponds to the original texture data. The remaining levels are computed by filtering and decimating the preceding clip level.
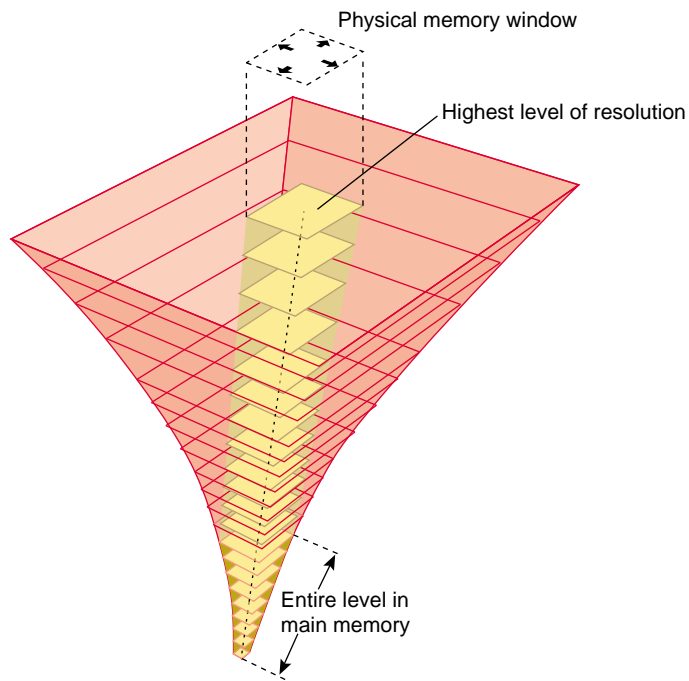
Figure 6-1   Clip Texture Hierarchy in Two Dimensions

The center of the physical memory window is usually the viewer's center of interest. As the viewer moves, the center of interest is updated and the texture data that is not in the window anymore is replaced by new data from disk. This data is paged into slots vacated by data being paged out of the window. This mapping ensures constant memory usage during user interaction. As shown in Figure 6-1, lower resolutions of texture data fit completely in main memory. During periods of fast user motion, these low-resolution textures are rendered while high-resolution data is being paged in. As higher-resolution texture data is available, it is rendered to improve the image quality of the visualization. This mechanism provides the capability to interactively visualize huge amounts of texture data resident in main memory or on high-performance disks.

Special-purpose SGI graphics hardware, such as InfiniteReality systems, provides built-in support for 2D clip textures. OpenGL Volumizer implements a software emulation of 3D clip textures. A 3D clip texture is an extension of a 2D clip texture scheme combining volume roaming with multiresolution volume rendering. In this case, the data transfer process is supported by representing the whole clip texture hierarchy as a collection of smaller 3D bricks at each level of resolution. This volume data representation combines the benefits of bricked volume files, asynchronous disk paging, multiresolution and volume roaming methods to overcome memory and pixel-fill constraints.

Like their 2D counterparts, 3D clip textures maintain a window of textures, which fits in main memory of the system at each level of the hierarchy. A clip texture loader replaces all texture data not contained in the window with new texture data from disk. To maintain the highest data transfer rate, OpenGL Volumizer represents the clip texture hierarchy as a collection of smaller 3D bricks at each level of resolution.

Figure 6-2 shows different resolutions for the same volume data. Each brick in the figure has the same data dimensions but they have different geometry ROIs. For example, if the brick has dimensions of 128 x 128 x 128 (2MB for 1 byte/texel), approximately 4 terabytes of volume data require 256 x 256 x 256 blocks at the highest resolution. By subsampling the data to lower resolutions using a kernel of size 2 x 2 x 2 or 4 x 4 x 4 texels, the data can be reduced by a factor of 8 and 64, respectively.

res = n voxels/dim unit

LOD1
LOD2
LOD3
LOD4
LOD5
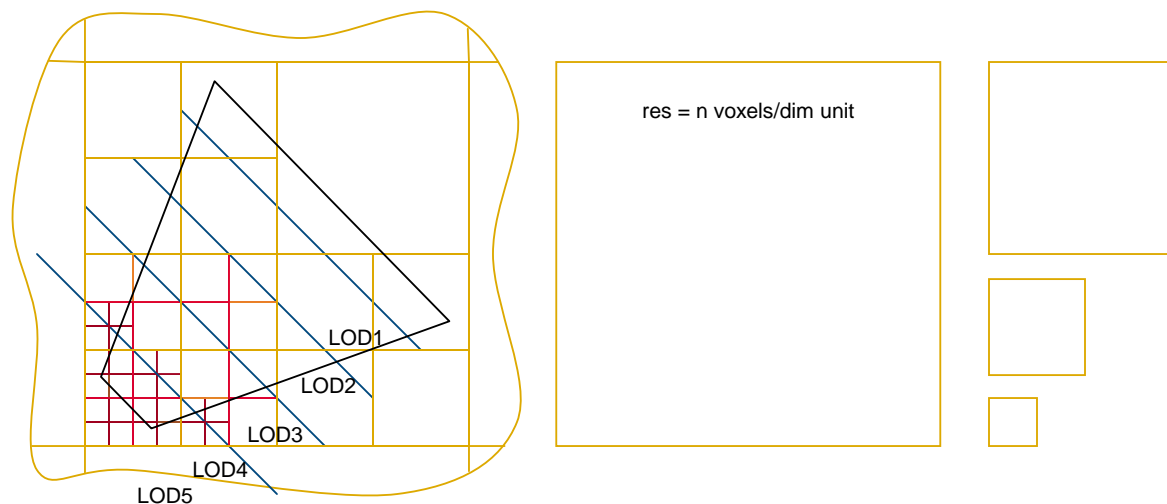
Figure 6-2   Subsampled Volume Data

The core of the large-data API for OpenGL Volumizer is in the abstraction of a 3D clip texture and its associated renderer. The implementation of the clip texturing is exposed as a new class, vzParameterClipTexture, and an associated render action, vzClipRenderAction. Figure 6-3 and Figure 6-4 show the similarities between shape node components for the core OpenGL Volumizer API and the 3D clip texture API.
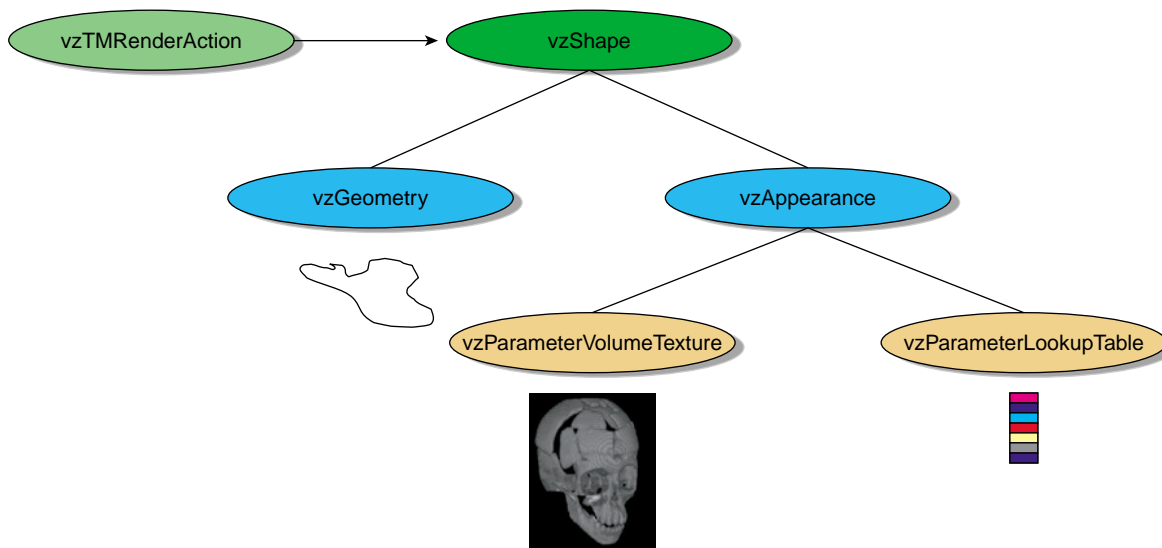
Figure 6-3   A Shape Node Using a Volume Texture
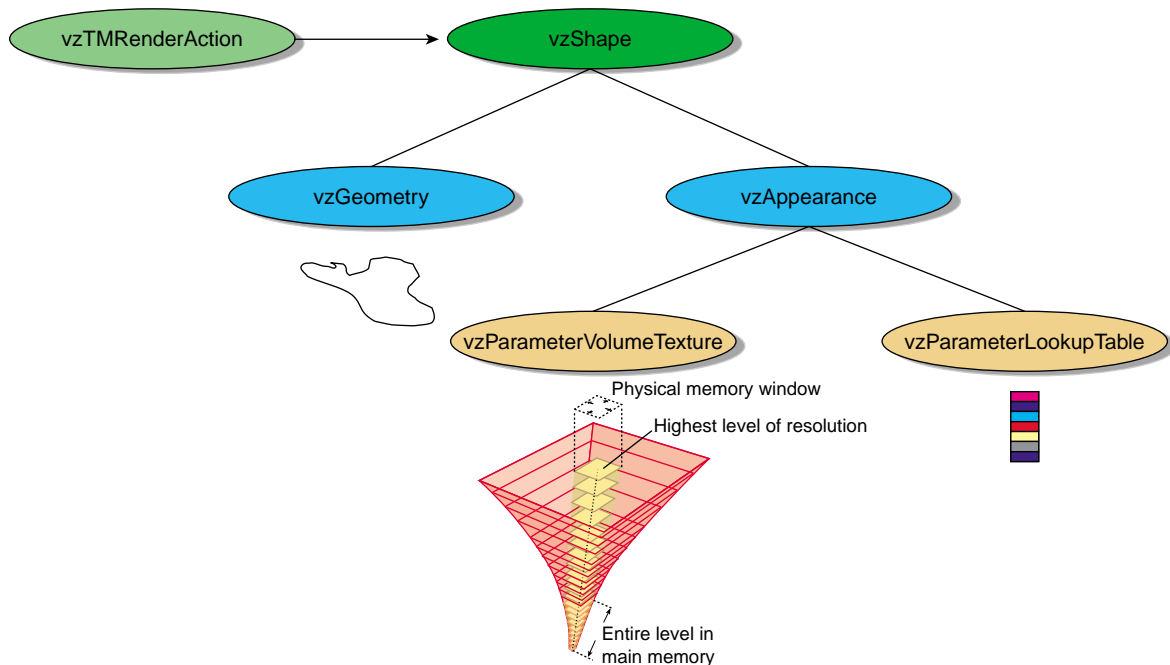
Figure 6-4   A Shape Node Using a Clip Texture

Clip Texture Representation: Class vzParameterClipTexture

The class vzParameterClipTexture is a new parameter class that provides an abstraction for a 3D clip texture hierarchy. The class maintains a set of clip levels and manages the amount of physical memory necessary to store the texture data. In addition, the class handles texture bricking and paging based on application-specific parameters. For efficiency, texture bricks are of constant size.

Applications control the following parameters to initialize a clip texture hierarchy:

Parameter	General Function
Brick dimensions	Optimizes the data transfer on the underlying hardware architecture, and it is constant for all bricks.

| Physical memory size | Limits the amount of physical memory used to load texture data. |
| Data loader callback | Allows a clip texture to load the texture data from the disk. |

Depending on the preceding parameters, the clip texture class creates a number of clip levels. Each clip level is assigned a maximum physical memory window, the size of which is computed from the total physical memory allowed for the clip texture. When the application starts, multiple loader threads are created, which invoke data-load callbacks to load texture bricks from disk. These callbacks are invoked in a sorted order using a cost function proportional to the distance of the brick from the center of the physical memory window. When a callback returns, the appropriate flag in the brick is updated to reflect the presence of the brick in main memory. As illustrated in Figure 6-5, this memory management and window management mechanism is implemented using a 4D toroidal mapping technique. OpenGL Volumizer maintains a separate toroidal map for each clip level.
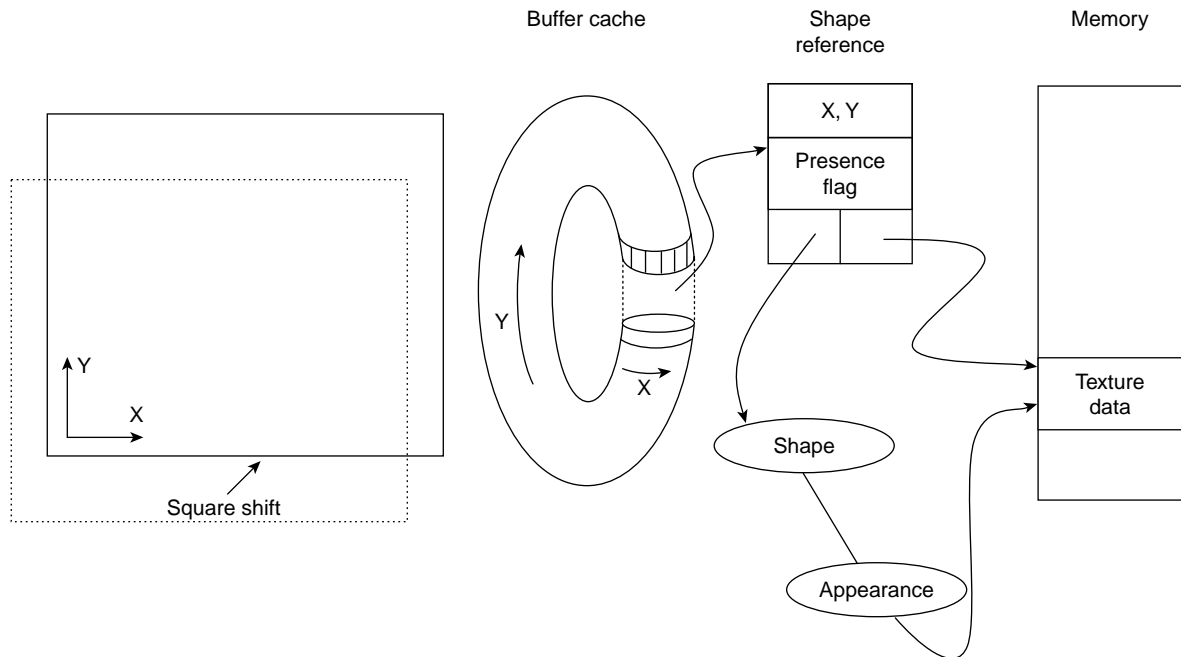
Figure 6-5   The Volume Buffer

Applications control the following parameters on a per-frame basis:

Parameter	General Function
Center of interest	Updates the center of the physical memory windows of the clip levels.
Roaming window size	Updates the size of the physical memory windows.

When the physical memory window is updated, all of the bricks that are not in the window anymore are marked to be dirty and, hence, need to be reloaded by the loader threads.

The following list defines the parameters for this class:

Parameter	Description
Data dimensions	Size of the whole volume data. This value is useful in computing the total number of bricks in the volume and the number of levels in the heirarchy.
Texture type and formats	Texture format, type, and internal format. These values are necessary to compute the number of bytes per texel and to create the volume textures representing the sub-bricks.
Center of interest	The (X, Y, Z) index in volume data space, corresponding to the current center of interest. Sets the center of interest for the current frame. Texture bricks closer to the center of interest will be paged in earlier compared to the bricks that are farther away.
Roaming window size	The dimensions of the actual roaming window containing the visible volume data. By default, all the data is visible. The **setRoamingWindowSize()** method requires the dimensions of a window as input. The method uses this window as the roaming window in conjunction with the center of interest. The dimensions are assumed to be specified according to the original data dimensions. Moreover, the actual roaming window dimensions for each clip level is computed depending on the total physical memory that clip texture is allowed to use.

The remaining parameters, shown in the following list, have default values computed internally. Applications can override the default values using the corresponding set methods. All these methods affect the way the clip texture is set up, applications should be modified immediately after the construction of the clip texture and before it is managed and drawn for the first time.

Parameter	Description
Texture brick dimensions	Size of the texture brick in terms of the number of texels in (X, Y, Z).
Geometry ROI	A mapping of the texture data onto world coordinates. Each of the subtextures will be assigned an ROI accordingly. The default values are (0, 0, 0) to (1, 1, 1).
Physical memory size	Limit for the size of the volumetric data to be kept in main memory. The method **setPhysicalMemorySize()** takes as input the maximum volume size kept in main memory. This value should be set depending on the total memory available on the system. Specifying a small value will mean that fewer high-resolution bricks will be resident in main memory at any given instant and, hence, lower-resolution levels of the clip texture will be rendered. Conversely, if a high value is specified, then higher resolution textures will be rendered.

Most of the set methods also have corresponding get methods. The man pages for OpenGL Volumizer describe the complete set of methods.

To load the volume data from disk, the clip texture class uses a data-load callback. OpenGL Volumizer provides sample callbacks with the demo code. The following is the format for the callback:

```
static void loadData(int offset[3], int level, int dimensions[3], void
*data, void *userData);
```

The callback that an application uses to load in volume data passes through the position `offset` and `level`. The dimensions of the data to be loaded are in the argument `dimensions`. The resulting data will be set in argument `data`. For example, the following sample code uses IFL loaders to load in the volume data.

```
void loadDataCB(int offset[3], int level, int dimensions[3], void
*data, void *userData) {
IFLLoader *loader=((ClipLoaders **)userData)->getLoaderForLevel(level);
loader->loadBrick(data, offset, dimensions);}
```

Clip Texture Rendering: Class vzClipRenderAction

The class vzClipRenderAction implements a render action for 3D clip textures. This render action implements intelligent, texture-paging techniques to render a clip texture hierarchy in a view-dependent fashion. It renders bricks that are closer to the viewpoint at a higher resolution compared to those farther away. It also performs view-frustum and geometry culling to discard bricks that are not visible in the current frame. This render action supports both looking at the visible frustum as well as roaming the dataset using multiple levels of detail. The renderer encapsulates a vzTMRenderAction and implements view-dependent rendering of clip textures. Depending on the geometry bounding box, the renderer is capable of rendering the whole volume or a subcubical roaming window in the data. Like vzTMRenderAction, it provides methods to manage and unmanage a vzShape and the methods **beginDraw()** and **endDraw()**. Unlike the draw method of vzTMRenderAction, the vzClipRenderAction draw method uses the current center of interest and clip window of the clip texture. The renderer culls out the texture bricks that do not lie in the visible frustum of the current window. In addition, the render action provides a debug utility that draws the wireframe for the bounding boxes of the texture bricks being rendered in the current frame.

The LOD used to decide whether a given brick is rendered or not is computed by projecting the brick's bounding box to screen space and then comparing it with a user-supplied threshold value. A brick that is closer to the viewpoint (that is, in front of other bricks at the same LOD in the visibility sorted order) gets higher priority in the selection process. The render action selects a brick for rendering if the following conditions are true:

• The brick intersects the viewing frustum and intersects the shape's geometry.

• Its associated texture data is in main memory.

• Its projection-to-screen space meets the LOD threshold criteria.

• The associated texture fits in texture memory.

The renderer allows applications to roam the clip texture by modifying the volumetric geometry for the shape. This geometry provides the region of interest in the volume data and can be moved around to navigate the dataset interactively. In order to maintain near-constant frame rates during user motion, the render action performs predictive texture downloads to distribute the overhead of the data transfer over multiple frames. The predictive texture download is a function of the direction of motion of the probe and a differential of the current and predicted positions. The texture download will proceed over a sequence of multiple frames. In the roaming mode, only textures at the same level of resolution are rendered. To find such textures, the renderer traverses the clip level

hierarchy to find the collection of bricks with the highest resolution possible and which satisfy the preceding criteria. Roaming mode is more efficient when using a smaller geometry as a probe, because the render action can provide finer control over the frame rate.

The vzClipRenderAction provides methods to control texture memory resources and trades off such resources against rendering time. For example, applications can control the amount of texture memory that the render action uses to store texture data by using the **setTextureMemorySize()** method. Applications can set an upper bound on the total size of textures that the renderer can download in a given frame using **setMaxDownloadSize()**. This constraint in conjunction with the LOD threshold helps improve the interactivity of the render action by reducing the time spent in downloading textures. In addition, the **setMaxDrawSize()** method allows applications to set an upper bound on the total size of textures that the renderer can draw in a given frame.

To get an approximate measure of texture-download time in a previous frame, use the **getDownloadTime()** method. It is only applicable if the total texture being rendered in the frame does not exceed the size of texture memory. The **getDrawTime()** method allows an application to elicit the time taken in microseconds for the actual draw of the shapes, including the time for the polygonization and the polygon rendering. Applications can control the ability to roam using the **setRoam()** method, which enables or disables the roaming mode by setting the argument roaming to true or false, respectively. If roaming is enabled, only texture bricks at the same level of resolution are drawn. In the "normal" multiresolution rendering mode, textures closer to the viewpoint are drawn at a higher resolution than textures farther from the viewpoint.

Visualization Pipeline for the Large-Data API

A visualization pipeline for large-data applications needs to address constraints imposed by large data and limited resources. Because of the amount of data involved, OpenGL Volumizer uses a two-step approach: offline data preparation followed by a visualization method that supports 3D clip textures. The first step in the data visualization is volume data preprocessing, which is done before the visualization process begins by an offline application. OpenGL Volumizer provides the ClipGen3d application (installed in the directory bin/clipGen3d under the sample distribution), which performs this pre-processing. See the README file in the directory for details for the various options, some of which will be discussed in the following sections.

Preprocessing

The vzParameterClipTexture class represents the clip texture hierarchy as a set of clipped mipmap levels of the original texture data. The clip renderer roams through each of the clip levels individually to provide interactive visualization. These mipmap levels are stored on the disk so that they can be paged in by the clip texture through a data-load callback and then rendered by the clip render action. Hence, before creating the clip texture, the original texture data needs to be preprocessed to compute these mipmap levels. At run time, the clip texture invokes a previously provided data-load callback to load texture bricks from disk. It is essential to understand the clip texture mechanism to be able to create your own clip texture levels.

The preprocessing step involves filtering and decimation of the input texture data to compute the clip levels, which are compressed representations of the data. In this process, the filtering step is applied to retain valuable features in the dataset that might be lost during decimation. Figure 6-6 illustrates the preprocessing.
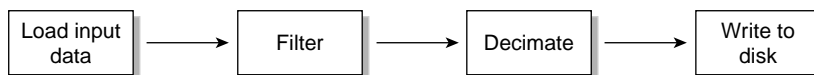
Figure 6-6   Preprocess Texture Data

The preceding process is repeated for each clip level that needs to be computed.

Number of Clip Levels

In this discussion, we assume that the level 0 corresponds to the original (highest) resolution texture data. Currently, vzParameterClipTexture assumes a decimation factor of 2 x 2 x 2. Hence, each of the clip levels has data dimensions that are half in size of the next higher resolution level. Let X, Y, and Z denote the texture dimensions of the original texture. Figure 6-7 illustrates the data decimation.

X	Y	Z	Level 0
X/2	Y/2	Z/2	Level 1
X/4	Y/4	Z/4	Level 2
	⋮		
			Level n-1
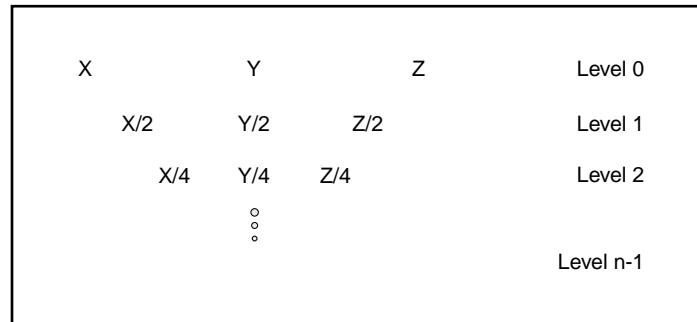
Figure 6-7   Data Decimation

Chapter 4, "Texture Mapping Render Action" describes the bricking of textures and the need to have a one-voxel overlap among adjacent bricks. The clip texture class handles the bricking of texture data internally, but the application has to ensure that it has taken the one-voxel overlap into consideration to compute the appropriate number of clip levels for the hierarchy.

To compute the correct number of clip levels, given the data dimension of the input data and the texture brick dimensions used for bricking the data, you must determine the following:

- The number of bricks along each dimension

```
Bricks = Data Dimensions / (Brick Dimensions – 1)
```

- The number of levels

```
Clip Levels = log2(max (bricks_x, bricks_y, bricks_z)) + 1
```

For example, using a brick size of 128^3 on a dataset with dimensions of 1024^3 results in the following:

```
Bricks = ceil ( 1024 / (128 – 1) ) = 9
Clip Levels = ceil ( log2(9) ) + 1 = 5
```

The lowest resolution level in the hierarchy contains only one texture brick. Hence, using a brick size of 128 x 128 x 128 would generate five clip levels with the following data dimensions:

```
1024 x 1024 x 1024
512 x 512 x 512
```

```
256 x 256 x 256

128 x 128 x 128

64 x 64 x 64
```

Writing Clip Levels for Optimal Disk Paging

Taking advantage of the fact that clip textures always load in bricks of constant sizes, the application can optimize the disk-to-memory transfer rate by computing the clip level files appropriately. The clip files can be stored using a scheme in which the data for the appropriate bricks is stored contiguously to account for a one-voxel overlap among adjacent bricks. This technique is used by the `RoamLoader` example loaders provided in `VZROOT/src/lib/loaders`.

The following pseudocode reorganizes an ordinary data file into such a bricked file:

```
for texture bricks index = 0 to numBricks - 1 {
    offset = index * (brickSize - 1)
    loadBrickDataFromFile(offset, brickSize, brickDataBuffer)
    writeBrickDataToFile(index, brickDataBuffer)
}
```

The **writeBrickDataToFile()** function writes the brick data to a file using contiguous bricks. Hence, while loading this file, the loader needs to perform only one seek-and-read per brick to load the data.

Data Filtering

The clip generator reads the volume data at a given clip level, computes the appropriate input and output parameters, filters, and then decimates the data to generate the following clip level in the hierarchy. To compute clip level n, a clip-generating application may choose to use clip level n–1 as the input texture. The exception to this rule is that an application should only brick and potentially filter clip level 0, but it should not be decimated, unless required by the application.

Users must be aware of available computational and memory resources and of application-specific requirements, such as specific filters, when implementing the filtering and decimation portions of a clip-generating utility in light of the data sizes involved when designing for the large-data API.

In general, you have two choices for implementing a 3D filtering routine. For example, you may choose to implement spatial filtering using convolution in three dimensions.

The second choice is to use a frequency-filtering method, which requires that the volume data, as well as the filter, be Fourier-transformed into frequency space, multiplied, and inverse-transformed to spatial coordinates. The desired method depends on the following:

- Number of available CPUs

- Characteristics of the filtering method to be implemented

- Availability of high-performing mathematical libraries for optimal, multithreaded or parallelizable computation

- Amount of memory resources available

The README file in that directory provides instructions on how to generate your clip levels. This utility generates a ClipData.ct file and a series of clip level files based on the size of the original datasets and the user-specified brick dimensions. The ClipData.ct file indicates the location of the source texture, its format, brick dimensions, the clip level formats, and the names of the clip level files. The text format of the ClipData.ct configuration file is the following:

```
Brick dimensions - dim_x dim_y dim_z
Original volume - <file format>
<outFileName>0
Clip volumes - <file format>
<outFileName>1
<outFileName>2
.....
<outFileName>n - 1
```

Each clip level file contains a filtered and decimated copy of the texture. In ClipGen3d, the filter implementation is through basic 3D convolution, assuming non-separable, odd-sized kernels. For voxels not covered by the filter support, such as edges, the convolution copies the corresponding voxels within the original dataset. Decimation takes a step in each of X, Y, and Z and subsamples the already filtered volume by the corresponding factor.

The file format for filters for this first release of ClipGen3d is a text file where the first three entries denote the dimensions of the filter kernel in three dimensions, followed by the filter taps or coefficients, as shown in the following:

```
dim_x dim_y dim_z
k_0 ... (k_x*k_y*k_z - 1)
```

The dim_i values are the dimensions of the filter kernel along the respective directions. The k_i values are the corresponding filter components or taps. The utility `ClipGen3d` does not assume that filters are normalized; so, it will normalize it if necessary upon reading the coefficients.

Clip Texture Visualization

A clip texture visualization architecture can be structured into the following separate threads that run at different rates, each depending on the resources available to the thread:

- An application thread that manages user interaction, including the center-of-interest region and the roaming window

- A clip texture loader that manages the physical memory window

- A clip texture renderer that manages and renders the texture memory window (a subset of the physical-memory window) and supports geometry and view frustum culling

Depending on the user interaction, the volume data is loaded from the storage device into memory in a predictive way. The view dependency affects the functions of the loader and the renderer in that it affects visibility windows, data loading, culling, and rendering.

The clip texture loader uses a visibility window, a cache cuboid from the clip texture class, and a second cache cuboid from the clip rendering class, representing the contents of main memory and texture memory. The clip texture loader uses that information to manage volume data in main memory. As needed by the application and based on the user's view, the loader asynchronously loads additional data from disk when the main memory cuboid moves. The computation of the visibility windows is based on the movement the user gives from mouse and button events.

The clip texture uses texture bricks of constant size for efficiency, as explained in Chapter 4, "Texture Mapping Render Action". The data-load callback provided to the clip texture is invoked to load bricks of this size only. Different bricks are loaded by modifying the offset and clip level parameters in the loader callback. The application should be able to properly load boundary bricks by appropriately padding the bricks to the requested brick dimensions.

For example, in the preceding example, in order to load a boundary brick along the X dimension, the clip texture might invoke the following:

```
int off[3] = {1016, 0, 0};
int level = 0;
int dims[3] = {128, 128, 128};
void *data = malloc(texelSize * 128 * 128 * 128);
(*loadDataCB)(off, level, dims, data, userData);
```

In this case, the brick to be loaded exceeds the data dimensions of clip level 0. The callback `loadDataCB` should be able to handle this case properly by loading the correct dataset into a temporary buffer of brick dimensions (8, 128, 128) and then copying it appropriately to the given brick data buffer.

Before rendering a particular brick, the renderer checks for its availability in main memory. As illustrated in Figure 6-8, the culler extracts boxes from the data representation according to the field of view, the position of the user, and the culler's direction inside the texture memory cuboid. This functionality also selects the level of detail needed from back to front of the volume. The optimizer selects the resolution level according to the presence of volume data in buffer memory and maintains bricks in texture memory. In addition, it produces a list of bricks to reside in texture memory as bricks to be rendered.
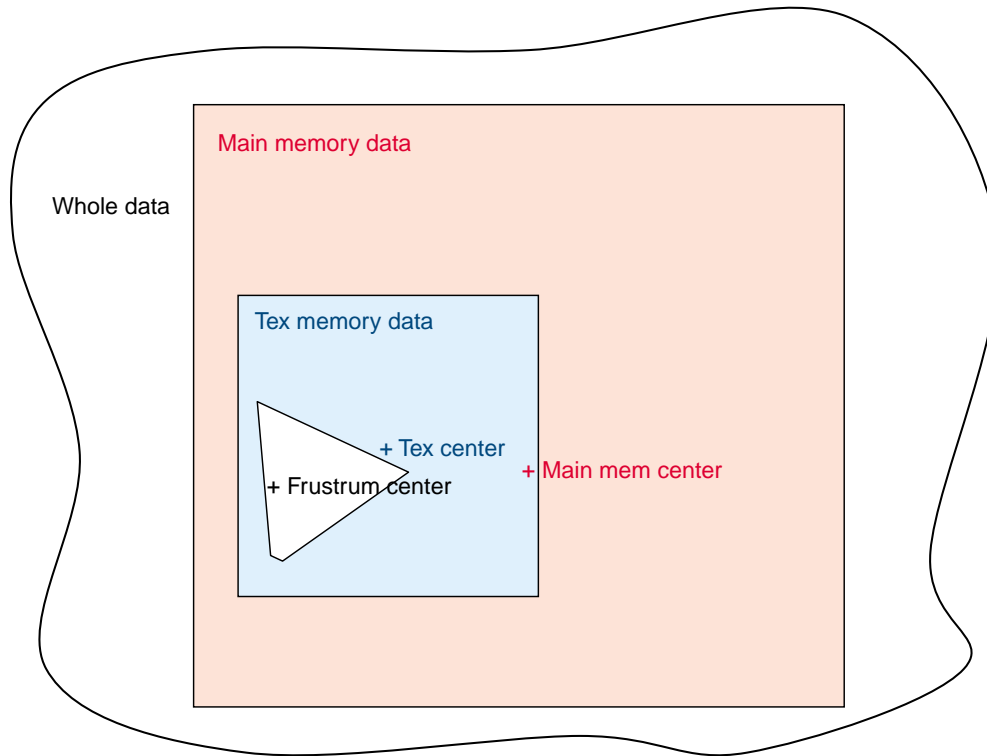
Figure 6-8   Rendering a Scene

As soon as the culler retrieves cuboids for rendering a new frame, the loader starts reading additional volume data blocks on disks. The renderer thread polls for the presence of data and determines whether to render the shape inside the cull space. Once all the shapes in the culling frustum have their data loaded in main memory and then in texture memory, they are rendered.

Mapping the preceding description to the clip texture API, a basic clip texture viewing application must do the following:

1. Initialize the vzParameterClipTexture class.

2. Attach a parameter volume to the appearance.

3. Use vzClipRenderAction to render the clip texture.

Irregular Grids: The Projected Tetrahedra Render Action

Volumetric data produced by acquisition devices, such as computer tomography (CT) or magnetic resonance imaging (MRI) devices used in medical imaging, is usually regularly spaced data where the space between adjacent sample points remains constant throughout the volume data. In certain applications, however, the sample points can be at arbitrary positions in world space. These kinds of *irregular grids* are commonly found in computational fluid dynamics (CFD) applications. For example, a CFD application can use these sample points to represent the space around an aircraft and then use computer simulations to compute the various fluid parameters, such as pressure and energy, at these sample points. Using irregular grids allows the grid to be more densely sampled at special regions of interest and sparsely sampled at other locations. This approach provides more accurate simulations for critical parts of the object under consideration— the nose of the aircraft, for example. Figure 7-1 shows the wireframe model of such a grid, which models the flow around a wing. Note how the data is highly sampled at the tip of the wing and sparsely so at other locations.
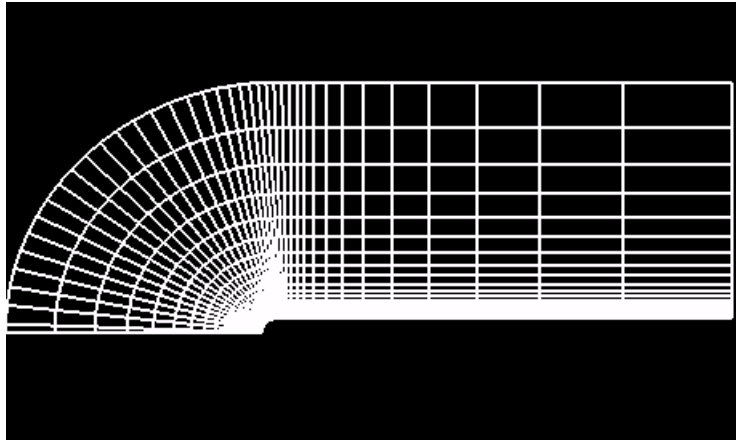
Figure 7-1   Blunt Fin Dataset (Wireframe Model)

Volume visualization of such datasets is critical in detecting the specific features in these datasets since conventional techniques like the use of isosurfaces and particle tracing are not able to reveal certain 3D structures in the data. Various techniques can be used to volume render such datasets. A commonly used approach is to resample the irregular grid to a conventional volume dataset using interpolation techniques. Such a process can be computationally expensive and, also, using a regular sampling can result in loss of critical details at areas of high density.

This chapter describes how OpenGL Volumizer renders irregular grids in the following sections:

• "The Projected Tetrahedra Render Action"

• "Rendering Using PTRenderAction"

• "Volumetric Geometry"

• "Shaders"

• "Plot3D Loaders"

The Projected Tetrahedra Render Action

The Projected Tetrahedra Render Action (PTRenderAction) provides a new render action for volume rendering irregular grids. The render action uses the following steps to render the grid:

1. Tessellate the input volumetric geometry into a tetrahedral representation.

2. Sort the tetrahedra in back-to-front visibility order using a fast sorting algorithm.

3. Render the grid by projecting each tetrahedron to the screen in the sorted order.

4. Apply color and opacity to the rendered geometry using the shading parameters (lookup tables, and the like).

The main difference between the algorithm used by the Texture Mapping Render Action (TMRenderAction) and PTRenderAction is in the polygonization step. Polygonization is the process of approximating the volumetric geometry (tetrahedra) using polygonal geometry (triangles) to allow direct rendering using OpenGL hardware. TMRenderAction approximates each rendered tetrahedron by computing proxy geometry by intersecting with viewport-aligned slices. PTRenderAction projects each tetrahedron into screen space using a *footprint evaluation* technique.

Figure 7-2 shows two different approximations of a tetrahedron projected into screen space. The first approximation uses three triangles and the second one uses four triangles.
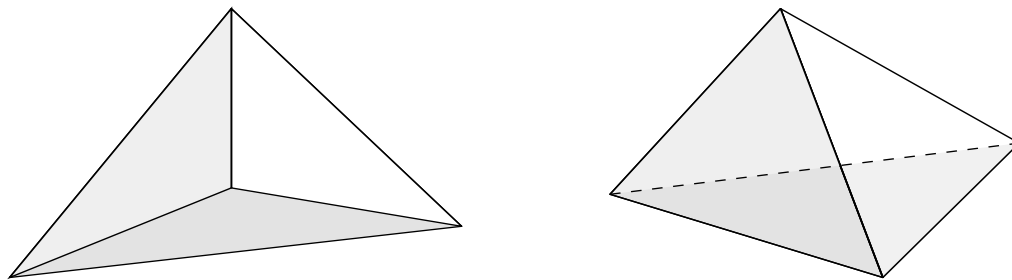
Figure 7-2   Tetrahedron Projected into Screen Space

Rendering Using PTRenderAction

Both PTRenderAction and TMRenderAction are derived from the base class vzRenderAction. Both provide the **manage()**, **unmanage()**, and **draw()** methods for resource management and rendering of volumetric shapes.

The following code sample creates a PTRenderAction and uses it to manage and render a shape node:

```
vzPTRenderAction *renderer = new vzPTRenderAction;
renderer->manage(shape);
renderer->beginDraw(0);
renderer->draw(shape);
renderer->endDraw();
```

The render order of a shape is determined by the invocation of its draw method. Consequently, the application must sort the multiple shapes being rendered in the desired visibility order. The render action provides all the data management internally using a *lazy evaluation* technique.

The PTRenderAction supports multiple sorting algorithms to sort the tetrahedral grids before rendering. Choose a sorting algorithm that matches the input grid that is being rendered. PTRenderAction supports the following sorting modes:

- `VZ_CONCAVE`

- `VZ_CONVEX`

- `VZ_NO_SORT`

The default mode is `VZ_CONCAVE`, which uses a linear order sorting algorithm for fast sorting of concave meshes. The sorting provides good visual quality but does not guarantee the correct order for certain boundary anomalies.

Use the `VZ_CONVEX` flag to provide accurate sorting if the grid is known to be convex. Use the `VZ_NO_SORT` flag to disable visibility sorting, in which case the volumetric geometry will be rendered in the order specified by vzIndexArray. The following code sample selects the concave sorting algorithm:

```
renderer->setSorter(VZ_CONCAVE);
```

Note that the sorting and tetrahedral projection steps are computed entirely in software and, consequently, constitute the bulk of the rendering time. Due to lack of hardware support, the rendering is usually not as interactive (between software and hardware) as

in the case of 3D textures, for which there is support in the graphics hardware. On Onyx4 and Silicon Graphics Prism systems, the projection algorithm can be implemented using the programmable vertex and fragment engines. This process is transparent to the application.

For certain datasets, it is useful to use other rendering techniques in conjunction with the projected tetrahedra algorithm to visualize the dataset. PTRenderAction also supports the following two rendering methods:

drawBoundary()   Renders the grid boundary.

drawCells()   Renders cells.

The rendered primitives can be controlled by setting the OpenGL state with **glPolygonMode()** to the desired primitive—namely, `GL_FILL`, `GL_LINE`, or `GL_POINT` (see the `glPolygonMode` man page). Figure 7-1 on page 106 was generated using the following code:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
renderer->drawCells(shape);
```

Volumetric Geometry

When using regular volumetric data, the volumetric geometry simply provides a region of interest for the 3D volume. Consequently, this geometry usually is a block geometry. The irregular grids produced by CFD applications typically tend to be of two kinds: structured hexahedral grids and unstructured tetrahedral grids. Structured hexahedral grids (also referred to as curvilinear grids) can be thought of as a "warped" and "scaled" regular volume because each vertex in the grid can be indexed using a 3-tuple, just like a regular volume. Therefore, the ordering of the vertices in the grid is implicit, and each set of eight neighboring vertices represent a hexahedron. The Blunt Fin dataset, shown in Figure 7-3, is an example of a structured hexahedral grid with dimensions of 40x32x32. An unstructured tetrahedral grid, however, can have vertices at arbitrary locations and, therefore, needs to be indexed linearly using a vzIndexArray (see "Shape-Related Classes" on page 30). An application can use its own volumetric geometry class as long as it overrides the virtual tessellate method to create a tetrahedral representation of the geometry.

The Blunt Fin dataset, shown in Figure 7-3, represents airflow over a flat plate with a blunt fin rising from the plate. The free stream flow direction is parallel to the plate and to the flat part of the fin, entirely in the X axis direction. The flow is assumed to be

symmetrical about a plane through the center of the fin. Consequently, only one half of the "real" geometry is present and used in the computation.
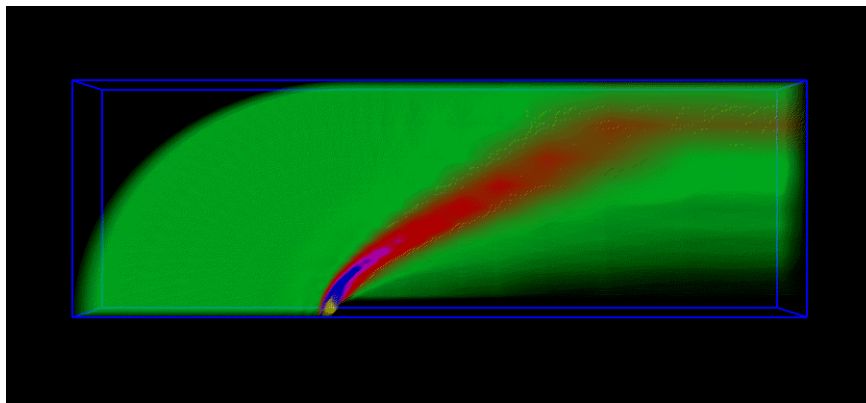
Figure 7-3   Blunt Fin Dataset

Shaders

PTRenderAction supports only one shader, vzPTLUTShader. The vzPTLUTShader volume renders the data using color and opacity, which are specified using the shader parameter vzParameterLookupTable. The following are the required parameters for vzPTLUTShader:

vzParameterVertexData   *vertex_data*

vzParameterLookupTable   *lookup_table*

The following sections describe the shader parameters vzParameterVertexData and vzParameterLookupTable .

Specifying Per-Vertex Data (vzParameterVertexData)

Each vertex in the output grid is associated with a set of floating point values which correspond to the computed data values at that vertex. The application can specify these data values using the class vzParameterVertexData, which allows storing a set of floating point values for each vertex in the grid. For example, Plot3D solution files store five

values per vertex—namely, pressure, momentum (3-tuple), and energy. This can be represented using a vzParameterVertexData as follows:

```
// Five values per vertex
vzParameterVertexData *vertexData =
    new vzParameterVertexData(5, numVertices, dataPtr);
```

The data values need to be interleaved in the form of *n*-tuples, similar to the vertex coordinates in the vzVertexArray class.

Transfer Functions (vzParameterLookupTable)

The *lookup_table* value specifies the transfer function used to map the per-vertex data values to color and opacity values. Because the values in vzParameterVertexData can be arbitrary floating point values, the mapping is done linearly between the minimum and maximum data values for the whole grid. For example, if the range of data values is [-0.56..1.25], then the first entry in the table would correspond to -0.56 and the last to 1.25. The indices for the intermediate values can be computed by linearly interpolating between the range. Figure 7-3 on page 110 shows a snapshot of the Blunt Fin dataset rendered using vzPTLUTShader.

The vzPTLUTShader uses a simple approximation for the volume density model to compute the volume rendering integral when rendering the projected tetrahedra. The transfer function for the volume renderer is implemented using a 2D texture map. This texture map is indexed using the scalar value at the rendered vertex as the "s" coordinate and the cell thickness as the 't' coordinate.

For more details on transfer functions, see "The vzParameterLookupTable Parameter" on page 60.

Plot3D Loaders

Plot3D is a commonly used data format for storing CFD datasets. OpenGL Volumizer includes a new loader library to load Plot3D data grid and solution files into OpenGL Volumizer geometry and vertex data objects, respectively. The loader library, libvzplot3d.so, is installed in /usr/lib* and the corresponding header file is installed under VZROOT/src/lib/loaders/Plot3dLoader.h. The loader supports reading in both single-zone and multizone grids. The grids can be structured as well as unstructured and with or without an IBLANK array. The data files are assumed to be in

binary format and all the floating-point data are assumed to be in 32-bit IEEE format, SGI endian. The following code loads a Plot3D grid and the corresponding solution file into an OpenGL Volumizer application:

```
// Create the Plot3d loader. Use the grid and solution (or scalar)
// field file names
Plot3dLoader *loader = Plot3dLoader::open(gridFileName, solnFileName);

// Load the Plot3d grid
vzGeometry *geometry = loader->loadGrid();

// Load the Plot3d solution data
vzParameterVertexData *vertexData = loader->loadSolution();
```

OpenGL Volumizer File Loader

This chapter describes the OpenGL Volumizer file loader and consists of the following topics:

- "The OpenGL Volumizer Shape Loader"
- "A Simple Code Example"
- "Related Classes"
- "The Volume Data Description Language"

The OpenGL Volumizer Shape Loader

Applications based on OpenGL Volumizer have the option of creating volumetric shapes using different components. These components form the *descriptive part* of the OpenGL Volumizer API. The OpenGL Volumizer file loader provides an easy-to-use interface for creating and modifying volumetric shapes using a shape description file. This interface allows applications to modify the shape nodes at run time using the Volume Data Description Language (VDDL), a simple XML-based text file. This prevents application writers from having to modify the application for every such change.

The file format greatly reduces the effort involved in inserting OpenGL Volumizer nodes in existing applications. These applications might be based on other toolkits such as Open Inventor or OpenGL Performer. These scene graph toolkits treat the volumetric nodes as leaf nodes (see Chapter 9, "Advanced Topics") and usually do not need to have any knowledge of the contents of these nodes. With the new interface, the leaf nodes can be modified simply by changing the XML-based, node description text file, which is loaded at run time by the application. Additionally, the interface supports using plug-in modules to allow application-defined loaders for loading different components in the data description hierarchy.

The file format supports loading of multiple OpenGL Volumizer shapes into a collection of shapes (see "The vzShapeSet Class" on page 115). This collection of shapes can then be rendered using vzClipRenderAction, described in the following sections.

A Simple Code Example

See the following file for a minimal OpenGL Utility Toolkit (GLUT) application that uses the new file interface to load and render OpenGL Volumizer shape nodes:

```
VZROOT/src/apps/simple/xml/viewer.cxx
```

The application is a slightly modified version of the example program used in Chapter 2, "Getting Started" and uses the file loader interface instead. The code can be compiled using the following commands:

(On IRIX)

```
% CC -c viewer.cxx
% CC -o viewer viewer.o -lvz -lvzxml -lglut -lGL -lGLU -lXm -lXt -lX11
      -lm -lXmu
```

(On Linux)

```
% g++ -c viewer.cxx
% g++ -o viewer viewer.o -lvz -lvzxml -lglut -lGLU -lGL -lXmu -lX11
```

(On Windows)

```
C:\> cl.exe /c viewer.cxx /I ..\..\..\..\include /D VISUALCXX
C:\> link.exe /out: viewer viewer.obj /libpath: ..\..\..\..\lib vz.lib
vzxml.lib opengl32.lib glut32.lib
```

Note that you need to run the VCVARS32.BAT batch file provided by Microsoft Visual Studio before compiling the sample code. Alternatively, you can create your own project in Microsoft Visual Studio to compile the application. Also, note that you need to have libtiff for Windows installed on your system in order to run this application.

Note: To provide the XML loader for the shape set, the application needs to link against libvzxml on IRIX and Linux and against vzxml on Windows. The XML loader library uses the Xerces XML Parser from Apache Software Foundation for parsing and validating the XML file.

The following are the two main components that change:

- Loading a shape set

  The following code loads a shape set and prints some debugging information about the shape set:

  ```
  vzShapeSet *shapeSet = vzShapeSet::load(filename);
  if(!shapeSet) {
      cerr<<"Error loading shape set from "<<filename<<endl;
  } else {
      cerr<<"Shape set has "<<shapeSet->getNumShapes()<<" shapes"<endl;
  }
  ```

- Rendering a shape set

  The following code demonstrates how to use vzClipRenderAction to render a shape set. It uses the **draw(**vzShapeSet ***)** method, which performs the management, sorting, and rendering of the shapes in the shape set.

  ```
  vzClipRenderAction *renderAction = new vzClipRenderAction();
  renderAction->draw(shapeSet);
  ```

Related Classes

The following classes are closely tied to the use of the XML file loader:

- vzShapeSet
- vzClipRenderAction
- vzGraphicsState

The vzShapeSet Class

The vzShapeSet class is a collection of vzShapes and can be loaded directly using the method **vzShapeSet::load()**. The load method accepts the filename for the XML file, parses and validates the file, and then creates and returns the appropriate shape set. If the file format is invalid or if an error occurred while loading the file, the method returns NULL.

In addition to the file loader, the vzShapeSet class provides utility routines for querying the bounding box of the collection of shapes and for sorting the shape set in a

back-to-front visibility-sorted order. Other routines are provided to add, remove, and retrieve individual shapes in the set. This allows applications to create and modify shape sets using a programmatic interface.

The vzClipRenderAction Class

The vzClipRenderAction class, described in Chapter 6, "The Large-Data API: 3D Clip Textures", can be used to render shape sets directly. A shape set can be rendered either using the **draw(**vzShapeSet ***)** method (as in the example described in the section "A Simple Code Example" on page 114) or using the regular manage, unmanage, and draw interfaces provided by the render action. In the former case, all the shape management, sorting, and rendering of the shapes in the set is handled by the render action. This scheme allows rendering of shapes with simple volume textures and those with clip textures in a unified manner using the same render action. The render action first allocates the resources required for rendering volume textures and then uses the remaining resources to render any clip textures in the shape set.

The vzGraphicsState Class

A shape set is associated with its own graphics state, which is useful for controlling dynamic rendering parameters for the set. This encapsulation is provided by the vzGraphicsState class, which is used by applications to pass the necessary viewing and other graphics state-related information to the shape set. For example, the graphics state associated with a vzShapeSet is used by the graphics state of vzParameterClipTexture in the set to update the centers of interest when the view direction changes.

The following code shows how to update the modelview matrix for the graphics state by querying the current matrix from OpenGL:

```
GLdouble modelview[16];
glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
shapeSet->getGraphicsState()->setModelviewMatrix(modelview);
```

The preceding code needs to be called by the application to notify the shape set of changes to the viewing parameters when relevant. For example, if the viewing direction changes and the render action is using the multiresolution mode, the preceding code should be called.

The Volume Data Description Language

The Volume Data Description Language (VDDL) allows you to change volume data descriptions at run time by simply editing a text file.Using XML tags, VDDL mimics the hierarchy of an OpenGL Volumizer shape description. The file loader uses appropriate default values for unspecified components in the file. For the complete syntax of the file format, see the `vzxml.dtd` file in the directory `VZROOT/data/XML/`. This directory also contains other files that can help you get started. The XMLViewer application installed in directory `VZROOT/src/apps/XMLViewer` loads and renders the sample VDDL files installed in directory `VZROOT/data/XML`.

The following topics are described in this section:

- "Simple Shape Example"
- "Adding Shaders and Parameters"
- "Loading Fragment Programs"
- "Modifying Geometry"
- "Loading Different Data Formats Using Built-in Modules"
- "Writing Custom Modules for Loading Datasets"

Simple Shape Example

The following VDDL file is used to represent a shape with an appearance containing a single `volume` texture parameter, specified using a 3D TIFF file. In the following example, a `vzBlock` will be used as the default geometry and a vzTMSimpleShader as the default shader.

```
<?xml version="1.0"?>
<!-- DTD file for the XML file format -->
<!DOCTYPE vzShapeSet SYSTEM "vzxml.dtd">
<vzShapeSet>
    <vzShape>
        <!-- Appearance -->
        <vzAppearance>
            <!-- Parameter: name - 'volume', type -
            vzParameterVolumeTexture -->
            <vzParameterVolumeTexture name="volume">
            <filename>VZROOT/data/medical/Phantom/CT.Head.char.tif</fi
            lename>
```

```
                    </vzParameterVolumeTexture>
                </vzAppearance>
            </vzShape>
        </vzShapeSet>
```

Adding Shaders and Parameters

The file shown in the preceding section "Simple Shape Example" can be modified to define more complex shape sets containing multiple shapes, more sophisticated shaders, parameters, and so on. Augmenting the appearance description in the preceding file to add a new shader and the required parameters allows rendering the same shape with a volumetric lighting shader applied to it, as shown in the following code:

```
<!-- Tangent space shader -->
<vzTMTangentSpaceShader/>

<!-- Param: name - 'lookup_table', type - vzParameterLookupTable -->
<vzParameterLookupTable name="lookup_table"/>

<!-- Parameter: name - 'lightdir', type - vzParameterVec3f -->
<vzParameterVec3f name="lightdir"/>
```

Loading Fragment Programs

On Onyx4 and Silicon Graphics Prism systems, the VDDL can be used to load and initialize appearances with fragment programs. The TMFragmentProgram interface requires the application to set texture unit callbacks to set up each texture unit appropriately. The XML loader, however, takes care of setting up the callbacks and other parameters for the shader automatically.

The following example illustrates the use of an XML file to initialize a fragment program and the required parameters :

(See file VZROOT/data/XML/fpWithBiasAndAdd.vz.)

```
<vzAppearance>

    <!--        Shader         -->
    <vzTMFragmentProgram>
        <filename>VZROOT/src/shaders/BiasAndAdd.fp</filename>
    </vzTMFragmentProgram>
```

```
<!--       Parameters        -->
<vzParameterVec3f name="bias">
    <value>1.0 0.0 0.0</value>
</vzParameterVec3f>

<vzParameterLookupTable name="lookup_table"/>

<vzParameterVolumeTexture name="volume">
   <filename>VZROOT/data/medical/Phantom/
     CT.Head.char.tif</filename>
</vzParameterVolumeTexture>

<vzParameterVolumeTexture name="volume2">
   <filename>VZROOT/data/medical/Phantom/
     CT.Head.Bone.char.tif</filename>
</vzParameterVolumeTexture>
```

```
</vzAppearance>
```

The following operations occur when the preceding file is loaded:

1. The loader sets up a per-texture unit callback for each parameter that requires a texture to be bound—that is, vzParameterLookupTable, vzParameterVolumeTexture, and vzParameterClipTexture:

Texture 0	lookup_table
Texture 1	volume
Texture 2	volume2

2. The loader sets up a local parameter for each vzParameterVec3f attached to appearance:

Local 0	bias

The fragment program and the XML file need to be written so that the textures used in the program are the ones specified in the XML file.

The following code corresponds to the fragment program referred to in the preceding XML file (see file `VZROOT/src/shader/BiasAndAdd.fp`):

```
!!ARBfp1.0

PARAM bias = program.local[0];

# Temporary variables
TEMP volume, volume2, lut;

# Sample texture bound on unit 1 - CT Head "volume"
TEX volume, fragment.texcoord[1], texture[1], 3D;

# Lookup 1D texture on unit 0 "lookup_table" with output of unit 1
TEX lut, volume, texture[0], 1D;

# Sample texture bound on unit 2 - CT Bone "volume2"
TEX volume2, fragment.texcoord[2], texture[2], 3D;

# Apply the bias to output of texture unit 2
MUL volume2.rgb, bias, volume2;

# Add the result of volume with value in volume2
# Highlight the Bone structure in rendered output using the bias color
ADD result.color, lut, volume2;

END
```

Modifying Geometry

The following example defines a new block geometry that is scaled to a range of (–1, –1, –1) – (1, 1, 1):

```
<!-- Geometry -->
<vzBlock>
    <offsets>-1 -1 -1</offsets>
    <dimensions>2 2 2</dimensions>
</vzBlock>
```

In order to match the geometry in this example, the geometry ROI for the texture can also be updated to the following:

```
<vzParameterVolumeTexture name="volume">
    <filename>VZROOT/data/medical/Phantom/CT.Head.char.tif</filename>
    <geometryROI>-1 -1 -1 1 1 1</geometryROI>
</vzParameterVolumeTexture>
```

Loading Different Data Formats Using Built-in Modules

The XML loader allows you to get a well-formatted proprietary dataset into an application without too much programming. The loader has built-in modules for loading the following data formats:

- Image Format Library (IFL)-based

- Raw binary

- Digital Imaging and Communication in Medicine (DICOM)

IFL-Based Datasets

IFL-based datasets, such as Tag Image File Format (TIFF) files, are quite common. The example shown in the earlier section "Simple Shape Example" on page 117 shows the use of the built-in loader module for such files.

Raw Binary Datasets

Raw binary datasets have a data header followed by an array of binary data values. The following example loads a volume texture provided in raw binary format using a built-in module:

```
<vzParameterVolumeTexture name="volume">
    <filename>VZROOT/data/misc/bonsai.raw</filename>
    <!-- Built-in Module named "bin" -->
    <module name="bin">
        <!-- User string for passing additional info to the loader -->
        <!-- Header size, data dimensions, data format, data type, -->
        <!-- endianess-->
        <userString>0 256 256 256 luminance ubyte</userString>
    </module>
</vzParameterVolumeTexture>
```

Note for advanced users regarding endianess:

To account for cross-platform datasets, you can specify an additional flag as part of the XML `userString` to indicate the endianess of the data. The valid values for endianess field are `intel` and `mips`. For example, this is important if your dataset was computed on an SGI IRIX system with MIPS processors and you are loading and rendering it on an Silicon Graphics Prism system with Intel Itanium2 processors. Note that this is only relevant if the data type for the volume data is larger than one-byte. The following example shows the appropriate `userString` for a dataset with unsigned short data type stored on a MIPS/IRIX system to be loaded on a Silicon Graphics Prism system:

```
<userString>0 256 256 256 luminance ushort mips</userString>
```

If the endianess is not specified, the loader assumes that the endianess of the system used to compute the dataset is the same as that of the current system. Hence, the loader does not perform any byte swapping.

DICOM Datasets

The built-in module for DICOM datasets allows you to load a series of DICOM files into a 3D texture. The data can also be processed using different image processing parameters passed as a user string. See the `dicom.vz` example in the directory `VZROOT/data/XML`.

The following example loads a set of DICOM files using the built-in DICOM loader module:

```
<vzParameterVolumeTexture name="volume">
        <filename>dicom/picker/I.%03d</filename>
        <module name="dicom">
                <userString>-s 1 -e 87 +Wh 1 +obt3</userString>
        </module>
</vzParameterVolumeTexture>
```

The XML file can specify the DICOM filenames and any other data processing options using the `filename` and `userString` tags. The same options are accepted by the `dicomToIFL` utility as command-line options. You can use the –h option to get a complete list of options.

Writing Custom Modules for Loading Datasets

For loading proprietary data formats that have no built-in loader in OpenGL Volumizer, you can write custom (plug-in) modules. These modules can be invoked from the XML file. A custom loader module is basically a shared library or a dynamic shared object (DSO) that provides a loader routine for loading the dataset in a format compliant to OpenGL Volumizer.

The following example shows how to use an application-provided plug-in module to load polygonal geometry from a data file using the appropriate DSO. On Windows, replace the DSO file in the example with the DLL file `vzLoaders.dll`.

```
<!-- Polygonal Geometry -->
<vzPolyGeometry>
    <!-- Geometry data file name -->
    <filename>VZROOT/data/polygonal/knot.sgo</filename>
    <module>
        <!-- Module DSO's file name -->
        <filename>VZROOT/lib/libvzLoaders.so</filename>
    </module>
</vzPolyGeometry>
```

The loader module is responsible for implementing the callback routine create*ObjectClassName*, where *ObjectClassName* specifies the name of the class without the vz prefix. The callback is defined as the following:

```
typedef vzObject * (*vzObjectLoaderCB)(const char *filename, const char
*userString);
```

For the preceding example, the callback routine might look like the following:

```
extern "C" vzObject *createPolyGeometry(const char *filename,
   const char *userString)
{
   vzPolyGeometry *geom = SimplePolyGeometry::open(filename);
   if(!geom) {
       cerr<<"Could not open file "<<filename<<" for
       vzPolyGeometry"<<endl;
       return NULL;
   }
   return (vzObject *)geom;
}
```

For sample code, see file `VZROOT/src/XMLPlugins.cxx`.

Advanced Topics

This chapter consists of the following topics:

- "Integration with Other Toolkits"
- "Using Multiple Graphics Pipes"

Integration with Other Toolkits

OpenGL Volumizer is an API designed to handle the volume rendering aspect of an application. You can use other toolkits, such as OpenGL Performer and Open Inventor, to structure the other elements of your application. The API allows seamless integration with other scene graph APIs because the shape node can be used as the leaf nodes of such scene graphs. Figure 9-1 illustrates a hypothetical scene graph that contains polygonal data mixed with volumetric data. In this case, the shape nodes are used to represent the volumetric components of the scene while the Poly node is used to represent the polygonal geometry.
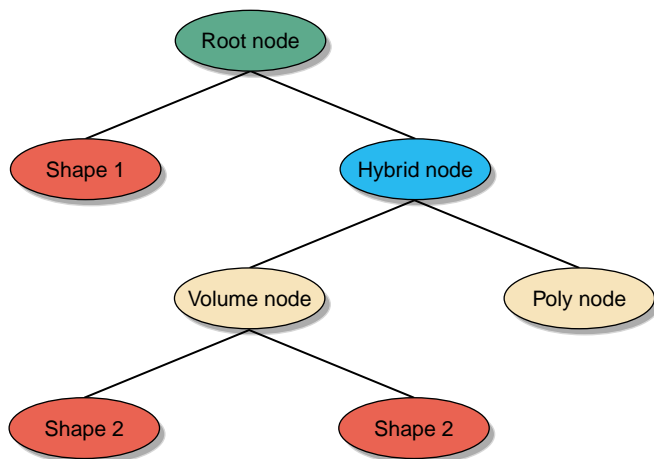
Figure 9-1   A Complex Scene Graph

Mixing geometric objects with volume-rendered data is a useful technique for many applications. For opaque objects, the geometry is rendered first using depth buffering and then the volume data is rendered with depth testing enabled. When using APIs such as OpenGL Performer, the scene graph traversal should be done in the appropriate order to ensure correct alpha compositing. The application can ensure this by *marking* the volumetric nodes as transparent so that the scene traverser renders it after the opaque geometry. In the case of OpenGL Performer, this can be accomplished by creating the appropriate pfGeoState and attaching it to the volume node. Figure 9-2 shows a volumetric data set rendered along with opaque geometry using the preceding technique.
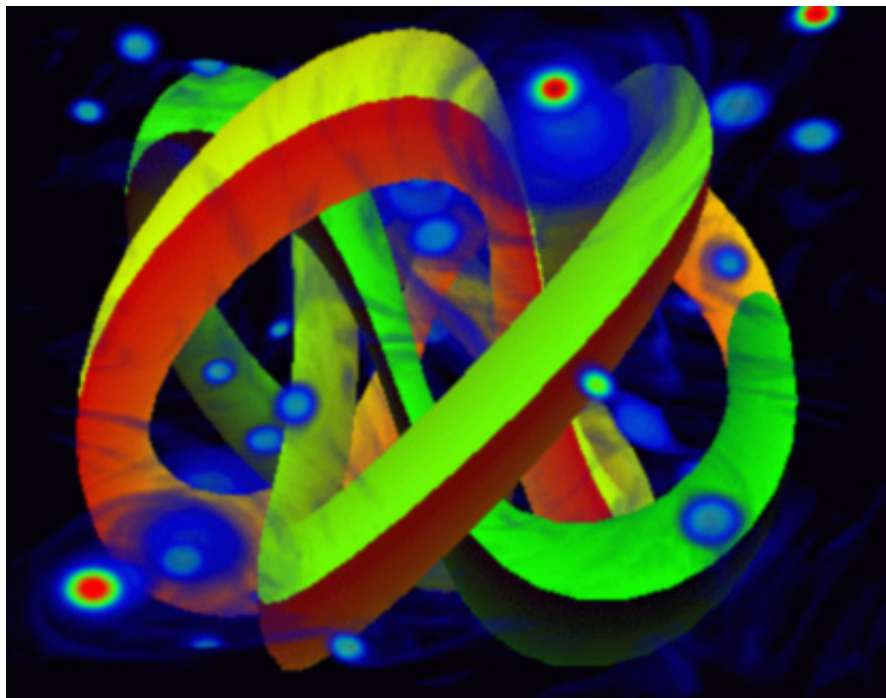
Figure 9-2   Volume and Opaque Geometry Integrated in a Single Scene

The design of OpenGL Volumizer permits integration with visualization toolkits as well. To demonstrate this aspect of that design, OpenGL Volumizer provides a new volume rendering method for the Visualization Toolkit (VTK) using 3D textures and based on the OpenGL Volumizer API. VTK is open source, free software for imaging and visualization. VTK supports a number of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods. Its volume rendering method is an implementation of ray casting and 2D texture-based rendering.

In the directory `VZROOT/src/`, the VTK example shows how to add 3D texture-based volume rendering to VTK, encapsulated in a class that provides all the necessary steps to support 3D texture-based volume rendering. The joint implementation (OpenGL Volumizer coupled with VTK) consists primarily of an OpenGL, 3D, texture-based mapper class. This class initializes the necessary OpenGL Volumizer volume parameters (such as shape and appearance), creates the geometry, creates an appropriate shader, and manages the shape. The new class,

vtkOpenGLVolumeTextureMapper3D, takes as input a vtkVolume and creates an equivalent vzShape node. Then the virtual renderer method uses TMRenderAction to draw the shape node.

Using Multiple Graphics Pipes

Thread safety allows applications the ability to run on large platforms for large immersive displays or to scale the graphics performance and resource use by sharing the scene graph among multiple rendering threads/processes. Typically used with OpenGL Multipipe SDK, the applications will be scalable and able to run in a Reality Center environment. Applications can scale the rendering performance of the system by compositing the intermediate results from different pipes to get the final image. Figure 9-3 shows n pipes rendering the same scene using one thread/process per pipe.
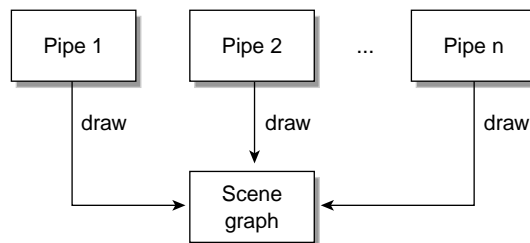
Figure 9-3   Multipipe Architecture

There are several topics to consider when using OpenGL Multipipe SDK with OpenGL Volumizer:

- "Execution Modes" on page 129

- "Rendering" on page 129

- "Decomposition Schemes" on page 130

For sample code that illustrates techniques described in this section, see the `volview` application that ships with OpenGL Volumizer. The application is a full volume-viewer application that uses OpenGL Multipipe SDK as the underlying software layer to provide run-time configurability and scalability.

Execution Modes

OpenGL Multipipe SDK supports the `pthread`, `fork`, and `sproc` multiprocess mechanisms. The following table describes their use with OpenGL Volumizer:

Mode	Prescribed Use with OpenGL Volumizer
pthread	Works well with OpenGL Volumizer since OpenGL Volumizer is thread-safe.
fork	Works well with OpenGL Volumizer since OpenGL Volumizer is thread-safe. Use the vzMemory class to ensure allocation of OpenGL Volumizer objects from shared memory by using the code shown in Example 9-1.
sproc	Does not work with OpenGL Volumizer since OpenGL Volumizer links against the `pthread` library.

Example 9-1   Using vzMemory to Allocate Objects from Shared Memory

```
// Set the allocation and deallocation callback functions
vzMemory::setMemoryManagementCallbacks(allocate, deallocate, NULL);

// The allocator callback function
void *allocate(size_t size, void *userData) {
   return mpkMalloc(size);
}

// The de-allocator callback function
void deallocate(void *pointer, void *userData) {
        mpkFree(pointer);
}
```

Rendering

OpenGL Volumizer render actions manage the graphics resources on a per-pipe basis. Use the following guidelines to get proper rendering:

- Ensure that OpenGL Volumizer nodes are stored as part of the shared data for the application. The data should not be replicated across multiple pipes unless there are special needs since volume data tends to be quite big in practice.

- Create only one render action per MPKWindow.

- Do not use multiple MPKWindows per MPKPipe since this will lead to inefficient resource management and also force unnecessary context switches. However, this practice might be acceptable for testing purposes on single-pipe machines.

- Do not share the same render action among multiple MPKWindows since each window has its own GLX context and execution thread.

Decomposition Schemes

Rendering performance can be scaled using multiple compositing schemes. This section describes the following topics:

- "2D Decomposition" on page 130
- "DB Decomposition" on page 131
- "3D Decomposition" on page 132
- "DPLEX Decompositon" on page 133

2D Decomposition

2D decomposition scales fill rate in a straightforward manner. If you can divide the volume data into smaller bricks, you can use view-frustum culling to scale the texture memory size also by moving these bricks in and out of the texture memory of the pipes. Section "Managing and Drawing Shapes" on page 46 describes how to do this.

You can achieve even greater scalability if you use 2D decomposition in conjunction with one or both of the following techniques:

- Hardware compositing with the SGI Scalable Graphics Compositor
- Adaptive load balancing

Figure 9-4 contrasts dynamic and static load balancing for a 2D compound using `volview`. Volume rendering is bound by fill rate; therefore, the load balancing can adjust the compound's region so that each pipe has approximately the same amount of volume to rasterize. When using static tiling, one pipe may have to render the whole volume as it is moved around. Since the slowest child dictates overall performance, the frame rate is better, in this case, when using load balancing.
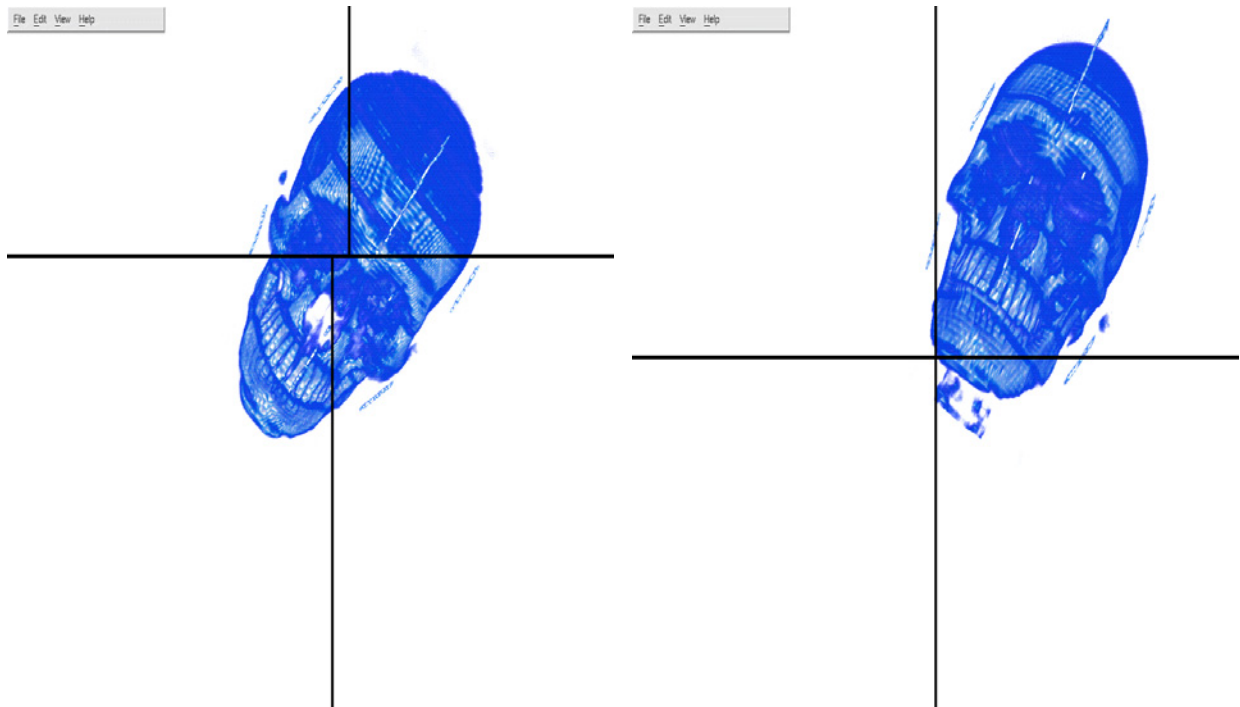
Figure 9-4   Dynamic versus Static Load Balancing

DB Decomposition

DB decomposition scales fill rate and texture memory size. Divide the volume data into multiple bricks and render equal number of bricks on each of the pipes. You need to pre-modulate the transfer function (lookup table) before rendering to compensate for the image blending operation used to composite the results together.

Figure 9-5 illustrates database (DB) decomposition using OpenGL Volumizer and OpenGL Multipipe SDK. The application partitions the volume data into four separate bricks. Each of these bricks are rendered on four different pipes to generate partial images. These images are then composited and displayed on the destination channel (which is also a source in this case) to give the final image.
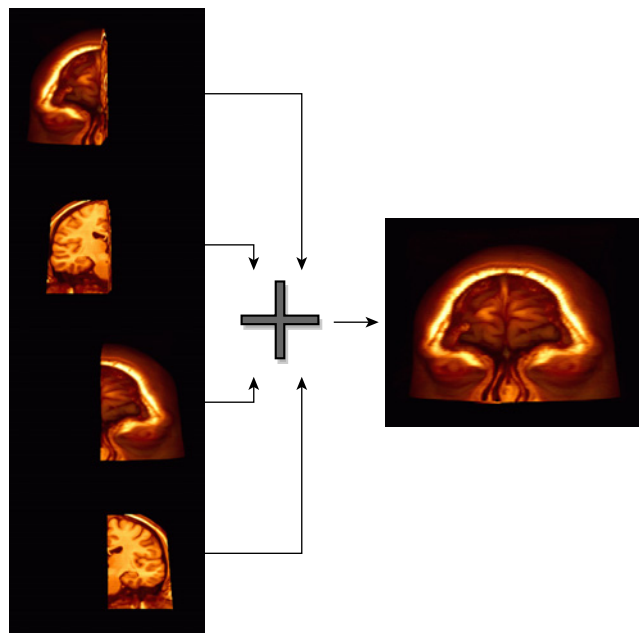
Figure 9-5   DB Decomposition

DB decomposition allows applications to linearly scale the texture memory size and fill rate performance with the number of graphics pipes on the system. For example, on a Silicon Graphics Prism system, a GPU supports over 200 Mbytes of texture memory. Using four graphics pipes in conjunction with DB decomposition provides over 800 Mbytes of texture memory.

3D Decomposition

Data streaming (3D decomposition) is not supported by OpenGL Volumizer since the order in which the source channels are composited cannot be controlled in this mode. However, true volume decomposition can be achieved using DB decomposition, which enables the application to scale in texture memory by distributing the data across multiple pipes.

DPLEX Decompositon

DPLEX decomposition is supported by OpenGL Volumizer. Full-scale software recomposition (that is, where the destination channel is also contributing actively in the rendering) simply requires the application to set the OpenGL Volumizer slice callback appropriately so that **mpkChannelSyncDPlex()** is invoked after every slice in the rendering thread.

Figure 9-6 shows an example of DPLEX decomposition, where consecutive frames are rendered over different pipes. This example shows a sequence of frames as the user modifies the transfer function for this seismic data set. The even frames are rendered on pipe 1 (red) and the odd frames on pipe 2 (blue), respectively. This technique effectively doubles the frame rate with minimal application effort.
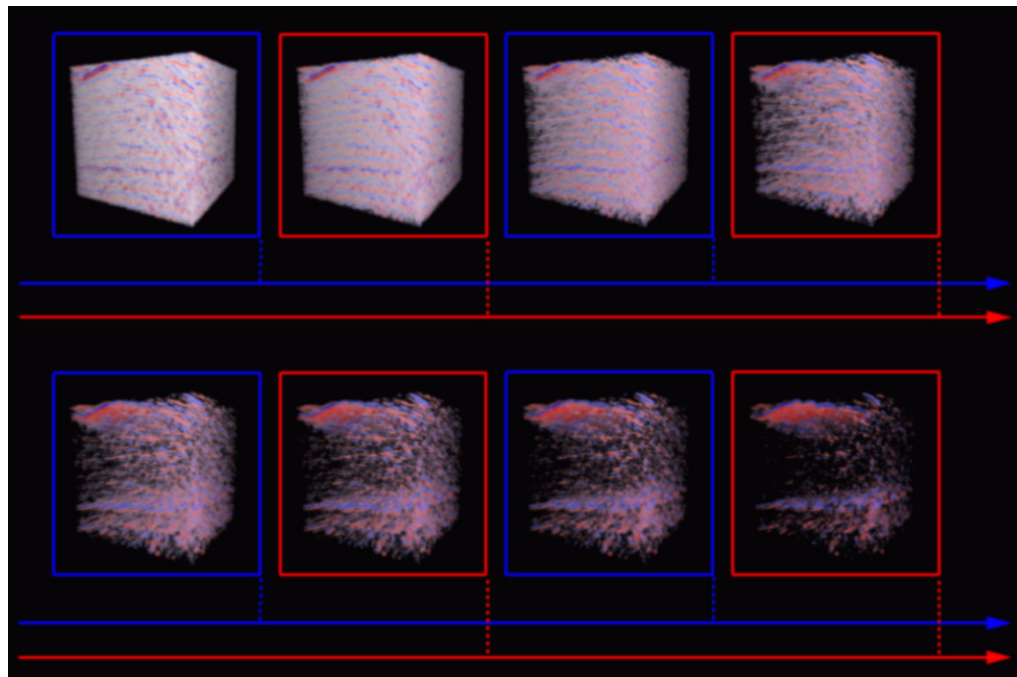
Figure 9-6   DPLEX Decomposition

Index

W

warn() method, 40
Windows platforms, 5
writeBrickDataToFile() function, 100

X

Xerces XML Parser, 114
XML
   file format, 117
   file loader, 7, 113, 114
   parser, 114