OpenGL Volumizer™ 2.1 Programmer's Guide

007-4389-002

CONTRIBUTORS
Written by Ken Jones
Edited by Susan Wilkening
Illustrated by Chrystie Danzer
Production by Glen Traefald
Engineering contributions by Praveen Bhaniramka

COPYRIGHT
© 2001 Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND
The electronic (software) version of this document was developed at private expense; if acquired under an agreement with the USA government or any contractor thereto, it is acquired as "commercial computer software" subject to the provisions of its applicable license agreement, as specified in (a) 48 CFR 12.212 of the FAR; or, if acquired for Department of Defense units, (b) 48 CFR 227-7202 of the DoD FAR Supplement; or sections succeeding thereto. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy 2E, Mountain View, CA 94043-1351.

TRADEMARKS AND ATTRIBUTIONS
Silicon Graphics, Infinite Reality, O2, OpenGL, and Onyx are registered trademarks and SGI, the SGI logo, Open Inventor, OpenGL Multipipe SDK, OpenGL Performer, OpenGL Volumizer, and Reality Center are trademarks of Silicon Graphics, Inc.

Cover Design By Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

New Features in This Guide

This revision includes the following noteworthy changes:

- Added a description of two new shaders, a gradient shader (vzTMGradientShader) and a tag shader (vzTMTagShader), to Chapter 4.
- Added the new section “A Closer Look at TMRenderAction” to Chapter 4.
- Added Chapter 5, “Advanced Topics”.

Record of Revision

Version	Description
001	August 2001 Original publication.
002	November 2001 Updated for the 2.1 release of OpenGL Volumizer.

Contents

Figures

Tables

About This Guide

This publication documents the release of OpenGL Volumizer 2.1 running on SGI systems.

OpenGL Volumizer 2.1 is 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 2.1 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 program interfaces is strongly recommended.

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.

Convention	Meaning
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.

Obtaining Publications

To obtain SGI documentation, go to the SGI Technical Publications Library at `http://techpubs.sgi.com`.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. 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 us 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 World Wide Web page:

  `http://techpubs.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
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043-1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

We value your comments and will respond to them promptly.

Overview

This overview consists of the following sections:

- "What Is OpenGL Volumizer 2.x?"
- "Product Components"
- "Supported Platforms"
- "Comparison with OpenGL Volumizer 1.x"

What Is OpenGL Volumizer 2.x?

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, Shirley-Tuchman), users demand more interactivity and immersion capabilities with their large volumetric datasets.

To help application programmers answer these needs, SGI has developed OpenGL Volumizer, a software development kit that provides a simple interface to the high-end graphics features available on InfiniteReality systems (such as 3D texture mapping and texture lookup tables).

OpenGL Volumizer 2.*x* is a new design that allows better incorporation and sharing of capabilities across our application programming interfaces (APIs) and facilitates management of extremely large volumetric datasets. OpenGL Volumizer 2.*x* 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.

OpenGL Volumizer 2.*x* Versus Other APIs

OpenGL Volumizer 2.*x*, like other SGI graphics APIs, is a layer of functionality that sits on top of OpenGL, as shown in Figure 1-1.

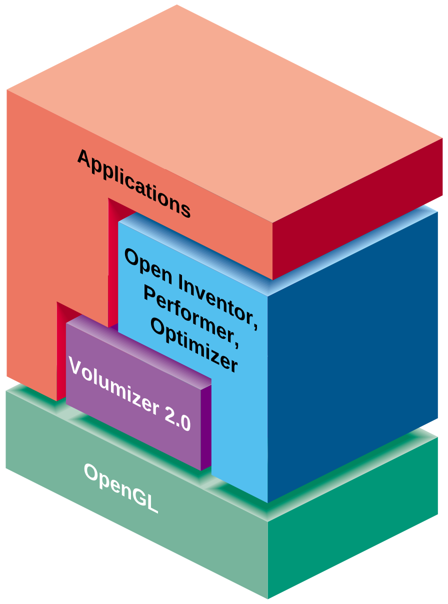


Figure 1-1 OpenGL Volumizer 2.*x* in Relation to Other Graphics APIs

OpenGL Volumizer 2.*x* 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 2.*x* is a rich toolkit with features that include the following:

- A high-level, descriptive C++ API
- Thread safety
- Integrated volume lighting
- Immediate-mode rendering without transient geometry overhead
- Built-in support for rendering slice planes
- Optimized for InfiniteReality systems
- A volume-viewer example application based on OpenGL Multipipe SDK
- A transfer function editor

Product Components

The product components fall into the following two categories:

- Core API

`libVo2.so`	Volumetric shape description API
`libVo2RenderTM.so`	3D texture-based render action

- Sample code

`volview`	Scalable volume-viewer application
`simple/tmRenderAction`	Simple volume rendering demos based on 3D texture mapping
`tfeditor`	Transfer function editor
`loaders`	Sample volume data loaders

Figure 1-2 shows the relationships among the components of OpenGL Volumizer 2.*x*.

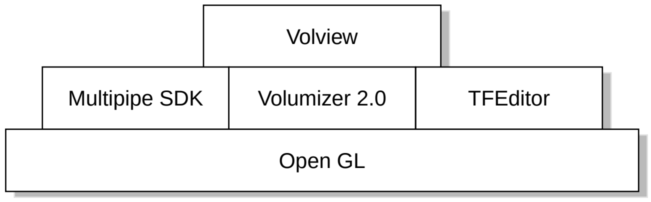


Figure 1-2 The Relationships among the Product Components

OpenGL Volumizer 2.*x* includes a number of example modules to help you use the API. They are not part of the supported API and are all located in the directory `/usr/share/Volumizer2/src`. In this directory are a 3D IFL loader, a sample transfer function editor, a volume-viewer application based on OpenGL Multipipe SDK (`volview`), and simple examples for each of the render actions.

Supported Platforms

OpenGL Volumizer 2.*x* 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 2.*x* is optimized for InfiniteReality graphics and is targeted at SGI scalable servers. Specifically, the texture mapping render action is optimized for InfiniteReality systems. Also, as the product name implies, it targets OpenGL applications.

Comparison with OpenGL Volumizer 1.x

OpenGL Volumizer 2.*x* should be distinguished from its predecessor, OpenGL Volumizer 1.*x*. OpenGL Volumizer 2.*x* is optimized to take advantage of SGI high-end scalable servers running InfiniteReality graphics, while OpenGL Volumizer 1.*x* is available on all SGI platforms. OpenGL Volumizer 2.*x* also provides a much higher-level API than OpenGL Volumizer 1.*x*, enabling you, the application writer, to solve large data problems with greater ease.

See Table 1-1 for a more complete comparison of features contained in OpenGL Volumizer 2.*x* and OpenGL Volumizer 1.*x*.

Table 1-1 OpenGL Volumizer 2.*x* Versus OpenGL Volumizer 1.x Features

	Volumizer 2.*x*	**Volumizer 1.*x***
API	High-level, descriptive	Low-level, procedural
Interoperability	Integrates with other toolkits	Integrates with other toolkits
Cross-platform	SGI systems supporting 3D texture mapping	Runs on multiple SGI platforms
Texture mapping render action	Yes	Yes
Arbitrary regions of interest	Yes	Yes
Texture lookup tables	Yes	Yes
Volume lighting	Integrated support	Unsupported example code
Thread safety	Yes	No
Polygonization technique	Immediate mode	Retained mode
Virtualized volumes	Transparent support	Exposed support

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 2.*x*:

- Introduction of a *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.

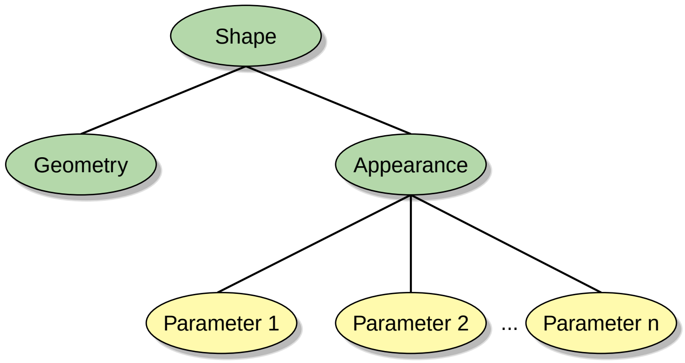


Figure 2-1 The Shape Node

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.

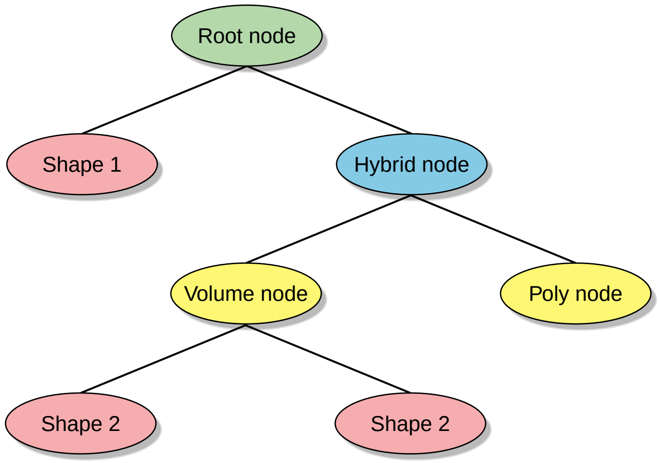


Figure 2-2 A More Complex Scene Graph

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.

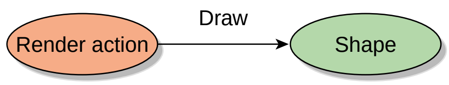


Figure 2-3 Render Actions

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.

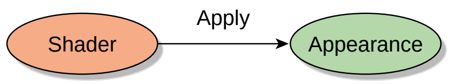


Figure 2-4 Shaders

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 `/usr/share/Volumizer2/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 and Running the Application"
- "Program Components"

Prerequisites

The following software must be installed on your system:

- OpenGL Volumizer 2.*x*
- Source for the sample application, installed in `/usr/share/Volumizer2/src/apps/simple/pguide`
- GLUT (available and free on the Web)

Note that the application links against `libVo2Loaders.so`, which is generated by compiling the source given in `/usr/share/Volumizer2/src/loaders`.

Compiling and Running the Application

To compile the application, enter the following commands:

```
% CC -o glut.o glut.cxx -c -I/usr/share/Volumizer2/src/
% CC -o viewer glut.o -L/usr/share/Volumizer2/src/lib/ -lVo2
 -lVo2RenderTM -lVo2Loaders -lglut -lGLU -lGL -lXmu -lX11
```

To run the application, enter the following commands:

```
% viewer /usr/share/Volumizer2/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 related include files are located in the directory `/usr/include/Volumizer2`.

This program also uses the IFL data loaders, which are installed in the directory `/usr/share/Volumizer2/src/loaders`.

Creating the Shape Node

The function **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.

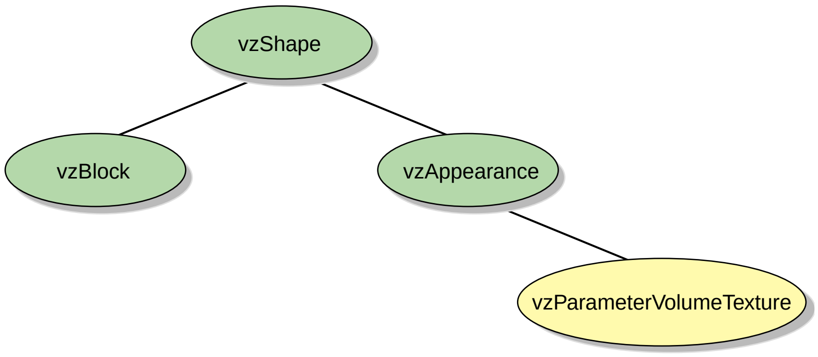


Figure 2-5 Shape Node in Sample Application

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 function **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 function **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 the 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:

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

Libraries

As an application writer, you need to be concerned about only two of the libraries that are installed as part of the OpenGL Volumizer installation. The following are the two libraries:

Library	**Description**
`libVo2.so`	Volumetric shape description and management constructs
`libVo2RenderTM.so`	3D texture-based render action

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 de-allocation 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 de-allocation of OpenGL Volumizer objects.

This might be useful in many cases. 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 de-allocation.

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);
}
```

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 de-allocation. 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

Table 3-2 (continued) Shape-Related Classes

Class	Description
vzVertexArray	An array of floating-point vertex coordinates
vzIndexArray	An array of integral indexes
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.

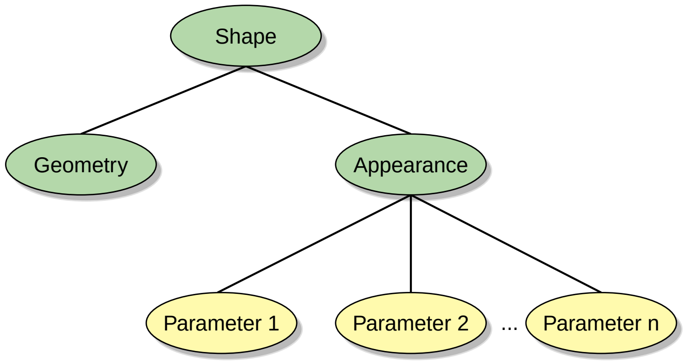


Figure 3-1 The Shape Node

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"

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 26.

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 26.

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.

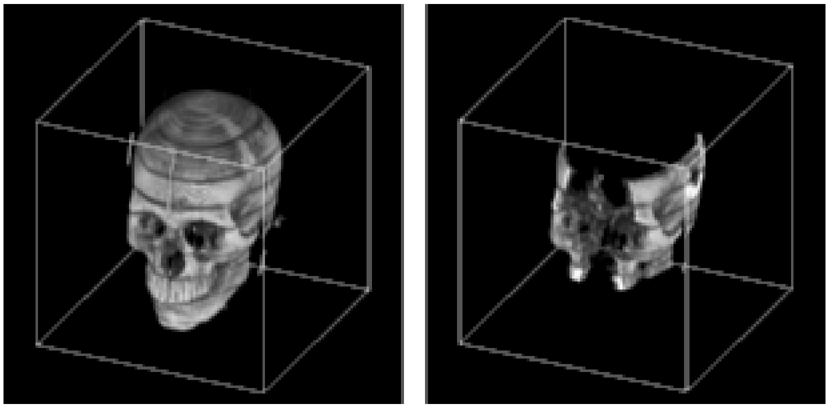


Figure 3-2 Modification of Shape Node from Sample Application

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.

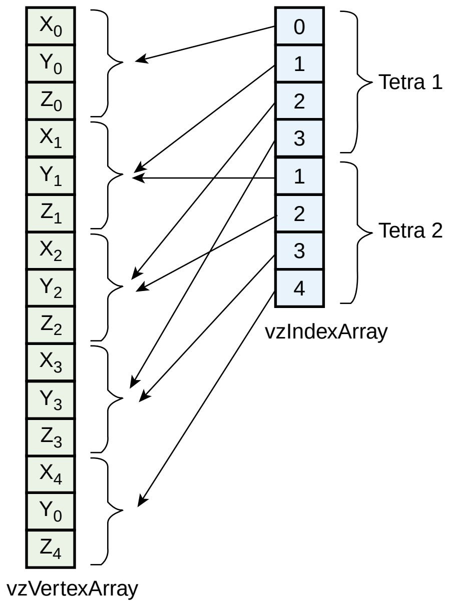


Figure 3-3 Construction of vzUnstructuredTetraMesh with Two Tetras

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 tetra mesh approximation.

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.

The 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".

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.

Renderers

A render action, as mentioned before, 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, the 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 currently the only render action shipped with OpenGL Volumizer 2.1. 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:

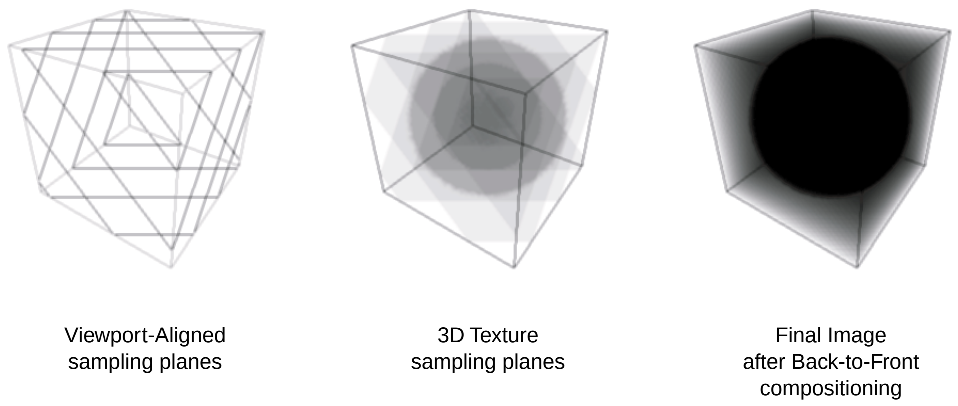


Figure 4-1 Viewport-Aligned Sampling Planes, 3D Textures Sampling Planes, and Final Image after Back-to-Front Compositing

The following are advantages of using 3D texture mapping:

- Using 3D texture mapping for volume rendering is very fast since all the interpolations for each fragment are done by the OpenGL hardware. Also, the texture data is resident in texture memory, which reduces the data access time considerably.
- Since the volume rendering process generates a polygonal approximation of the data, the technique allows you to mix volumes with other polygonal data.
- Many other techniques like *maximum intensity projection* can be implemented simply by changing the OpenGL blending functions.
- Arbitrary volumetric geometry can be used to specify regions of interest in the volume data.

Algorithm Used by TMRenderAction

TMRenderAction implements the 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.

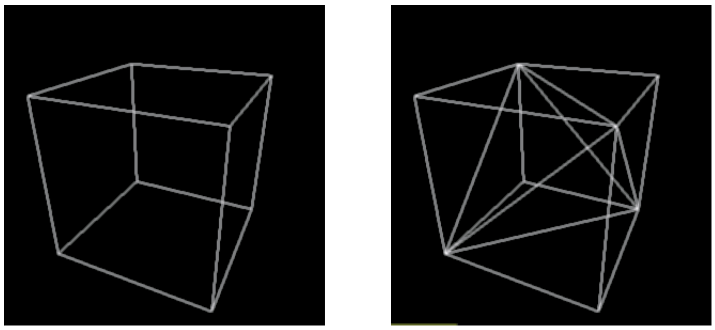


Figure 4-2 Original vzBlock and Corresponding Tessellation

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.

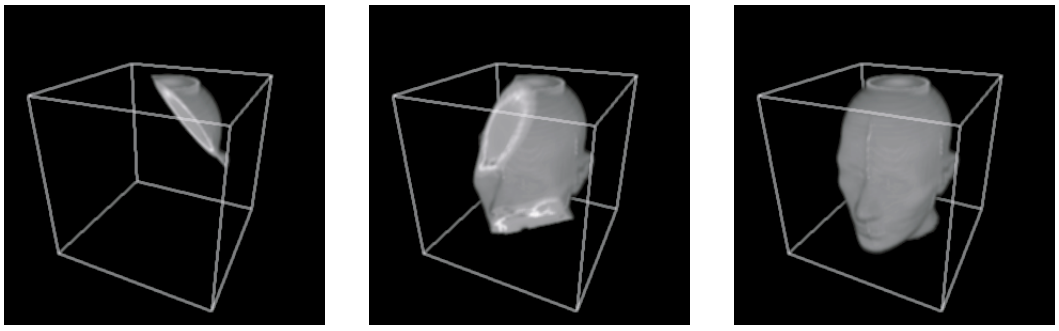


Figure 4-3 Back-to-Front Composited Slices for One, Three, and Five Tetrahedra (Final Image)

Note: In OpenGL Volumizer 1.*x*, the sliced geometry was stored and returned to the application. In OpenGL Volumizer 2.*x*, there is no such overhead. The geometry is rendered as it is generated.

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::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.

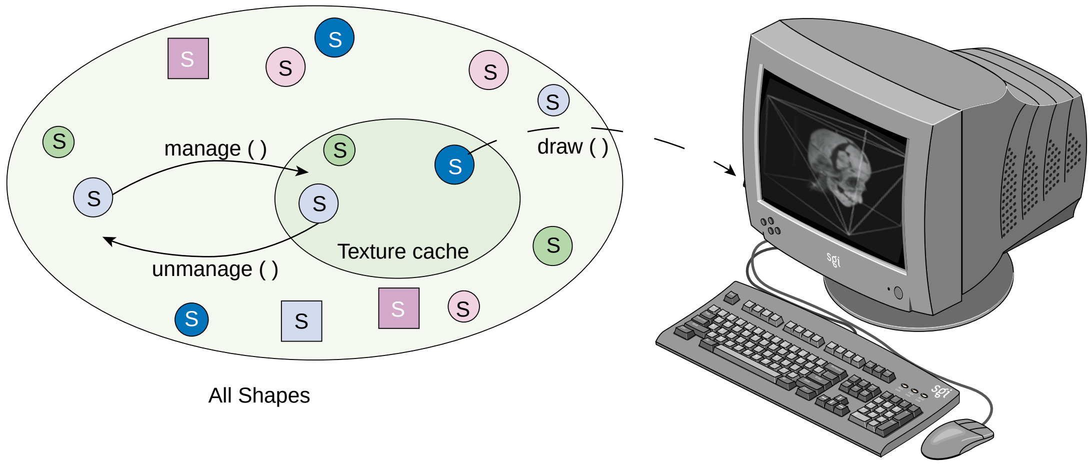


Figure 4-4 Managing, Unmanaging, and Drawing Shapes

TMRenderAction tries to load all the managed shapes into texture memory. Similarily, 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 three 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 shaders currently supported by TMRenderAction:

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

The vzTMSimpleShader

The vzTMSimpleShader has the following parameter:

Parameter Name	**Type**
`volume`	`vzParameterVolumeTexture`

As the name implies, the 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.

The vzTMLUTShader

The vzTMLUTShader has the two following parameters:

Parameter Name	**Type**
`volume`	`vzParameterVolumeTexture`
`lookup_table`	`vzParameterLookupTable`

The 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 the 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.

The vzTMLUTShader applies the transfer function using color tables, which are applied to the texel values in the imaging pipeline. 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 re-downloaded, which is usually much faster than downloading the whole volume texture.

The vzTMTangentSpaceShader

The vzTMTangentSpaceShader has the three following parameters:

Parameter Name	Type
`volume`	`vzParameterVolumeTexture`
`lookup_table`	`vzParameterLookupTable`
`lightdir`	`vzParameterVec3f`

The TMTangentSpaceShader 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.

The technique implemented by the 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 manipulating the texture matrix and rendering the sliced geometry in two passes.

Note:
The vzTMTangentSpaceShader does not generate correct lighting of volumetric data. It simply creates the appropriate visual effect by manipulating the texture matrix.

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 the vzTMGradientShader for correct volumetric lighting of shapes.

The vzTMGradientShader

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

The vzTMGradientShader has the following four parameters:

Parameter Name	**Type**
`volume`	`vzParameterVolumeTexture`
`gradient`	`vzParameterVolumeTexture`
`lookup_table`	`vzParameterLookupTable`
`lightdir`	`vzParameterVec3f`

The vzTMGradientShader implements a three-pass shading algorithm to perform gradient shading of volume data. The algorithm uses two perfectly overlapping volumes to perform gradient shading. The `volume` texture defines the actual volume data, while the other `gradient` texture defines the gradient for `volume`. The RGB values of the gradient texture provide the (a, b, c) coefficients for the gradient at each texel in the original volume. It is the application's responsibility to compute the gradient texture and add it to the shape's appearance.

The shader computes the dot product of the gradient values with the light direction using the OpenGL Imaging pipeline. This is done efficiently by setting the appropriate color matrix before downloading the gradient texture. The result of this dot product is a scalar value and, hence, can be stored internally as an intensity texture. So, the `gradient` texture should have an internal texture format of `VZ_INTENSITY`*n* (where *n* can be `8`,`12` or, `16`). Changing the light direction forces the gradient texture to be re-downloaded in order to re-compute the dot products with the new light direction. In addition, the shader accepts a lookup table parameter to apply transfer functions to the volume data.

Note:
The shading algorithm uses destination alpha to compute the gradient lighting. Hence, the application should ensure that the appropriate visual is selected.

Using the vzTMGradientShader has the overhead of potentially using two times the texture memory than the vzTMTangentSpaceShader. The gradient shader generates accurate lighting effects and does not have artifacts associated with the bricking of shapes, as opposed to the vzTMTangentSpaceShader, which does not generate correct lighting and produces seams for bricked shapes.

The vzTMTagShader

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

The vzTMTagShader has the following three parameters:

Parameter Name	Type
`volume`	`vzParameterVolumeTexture`
`tag`	`vzParameterVolumeTexture`
`lookup_table`	`vzParameterLookupTable`

The vzTMTagShader implements a two-pass algorithm 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.

The tagging algorithm uses stencil and alpha tests 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.

Note:
The tagging algorithm uses the stencil buffer to mask out the volume data. Hence, the application should ensure that the appropriate visual is selected.

Using the 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 4-5 illustrates the relationship between the data ROI and the geometry ROI of a texture.

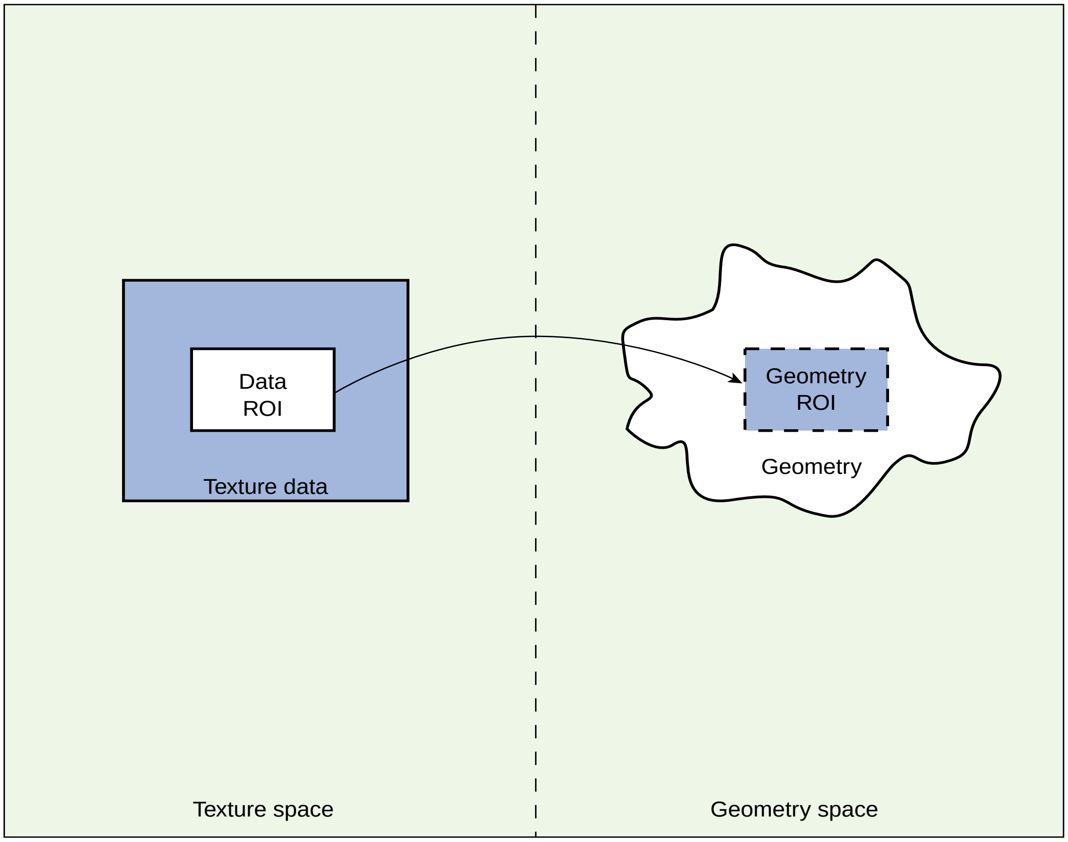


Figure 4-5 Data ROI and Geometry ROI of a Texture

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 4-6 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.

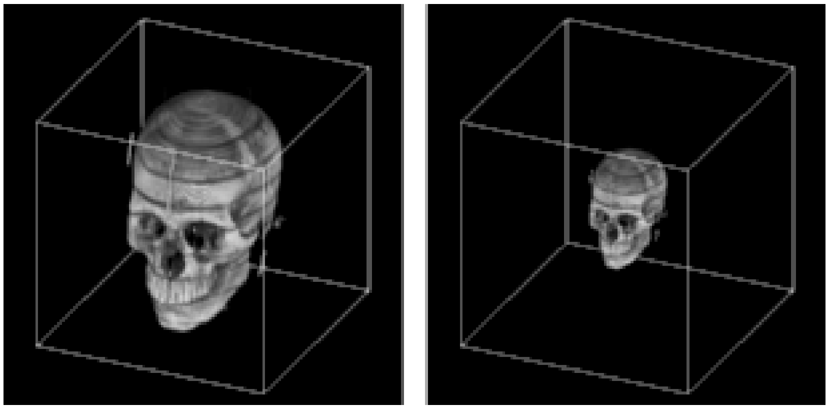


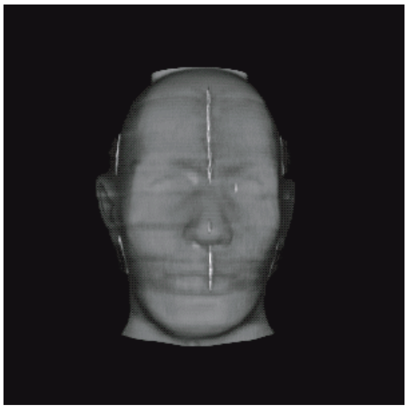
Figure 4-6 Original Texture and Texture after Modifying the Geometry ROI

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 4-7 shows a head image along with its lookup table.



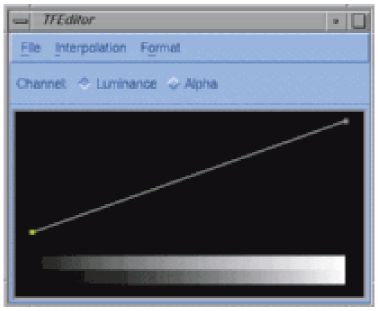
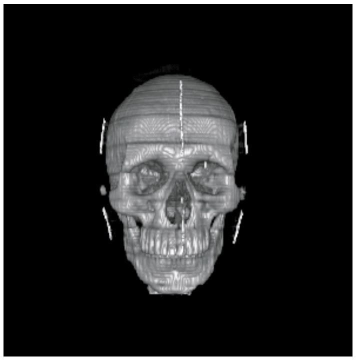


Figure 4-7 Head Image and Its Lookup Table

Figure 4-7 was generated using both the transfer function editor and the demo code provided with OpenGL Volumizer 2.*x*.

Figure 4-8 shows the skull of the head along with its lookup table.



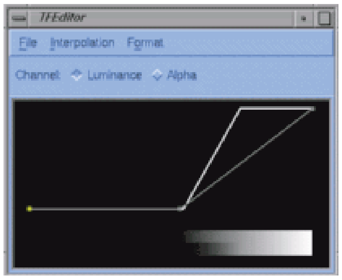


Figure 4-8 Skull of the Head and Its Lookup Table

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 the TMTangentSpaceShader 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 4-9 shows the pipeline used by a typical volume rendering application using the render action.

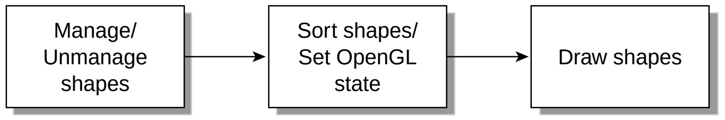


Figure 4-9 Volume Rendering Pipeline

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. The OpenGL hardware is more restrictive with respect to texture dimensions and sizes though. 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. All of these processes are transparent to you, requiring no intervention in brick creation, management, and sorting.

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 1-voxel overlap between the adjacent textures. Figure 4-10 illustrates this in 2D.

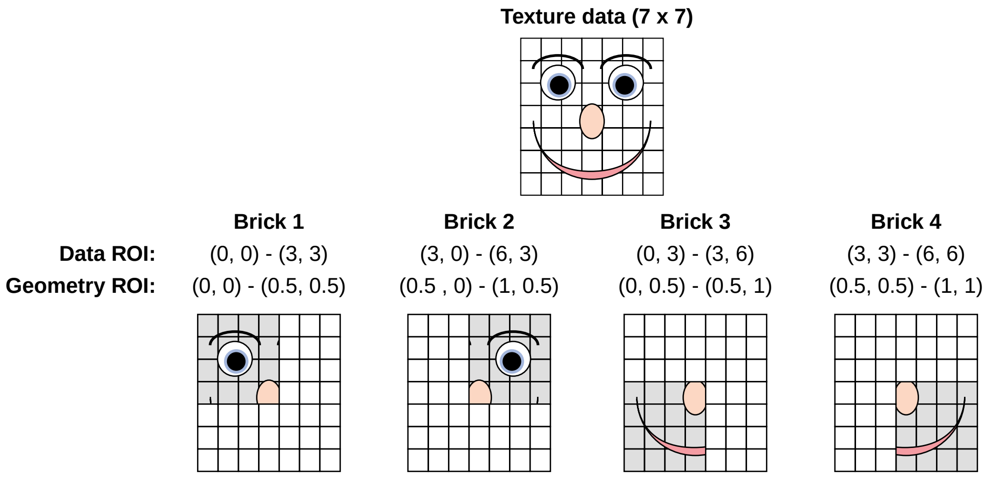


Figure 4-10 Texture Bricking

Figure 4-10 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 4-11.

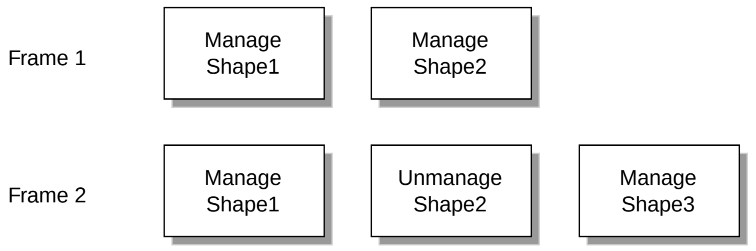


Figure 4-11 Reusing Texture Objects

In the first frame, the render action would allocate OpenGL texture objects for `Shape1`and `Shape2`. In the second frame, even though `Shape2` is not managed, the render action does not delete the texture objects for it. Instead, it re-uses 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, multi-resolution 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_INTENSITY`*n* (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_INTENSITY`*n* (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 responsilibility 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 `/usr/share/Volumizer2/src/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:

```
/usr/share/Volumizer2/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

Understanding the OpenGL state management in TMRenderAction can help you implement alternative functionality not supported by the render action. For example, you can render arbitrary polygonal geometry with the shape's `volume` texture applied to it. This can be done after calling the draw for a shape since the draw method does not restore any OpenGL state. 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. Also, 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 and shapes that have been bricked internally by the render action (if their textures do not fit in texture memory). You can ensure that the render action does not draw any polygons either by setting the volumetric geometry to be degenerate or by using slicing planes with all the planes disabled.

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 4-12 illustrates the principle.

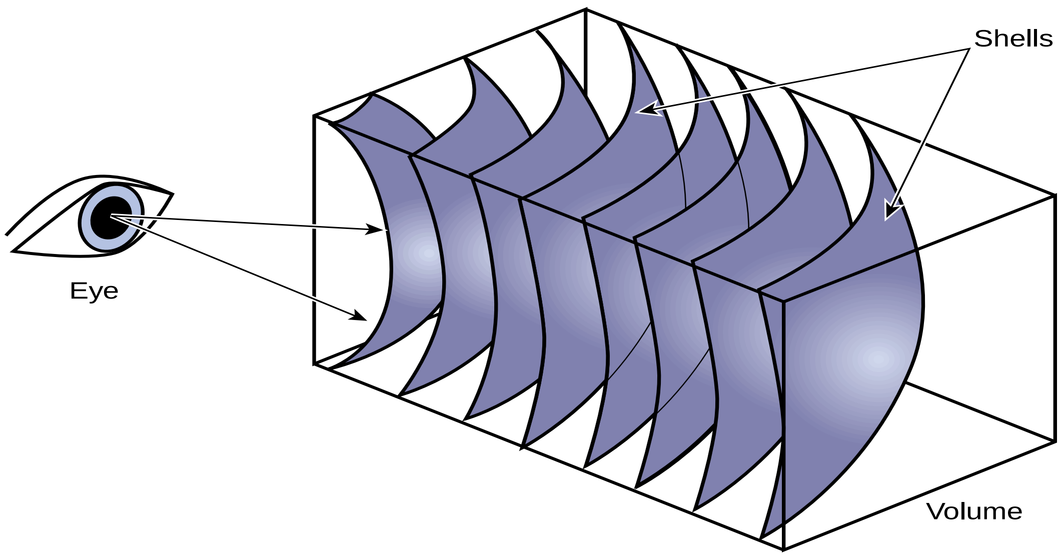


Figure 4-12 Spherical Slicing

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.

Advanced Topics

This chapter consists of the following topics:

- "Integration with Other Toolkits"
- "Using Multiple Graphics Pipes"
- "Visualizing Large Data"

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 5-1 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 polygonal geometry.

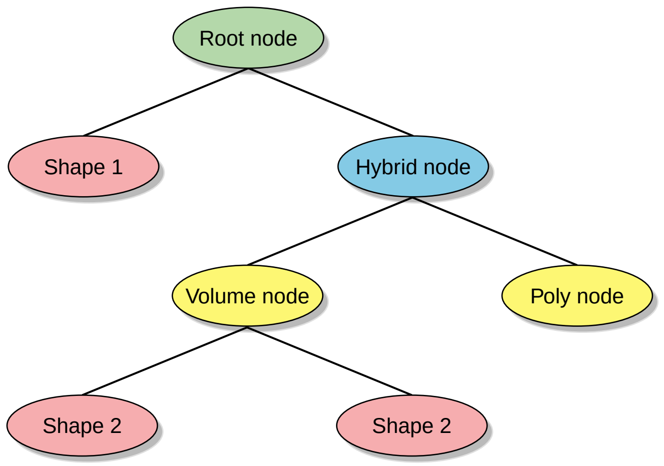


Figure 5-1 A Complex Scene Graph

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 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 5-2 shows a volumetric data set rendered along with opaque geometry using the preceding technique.

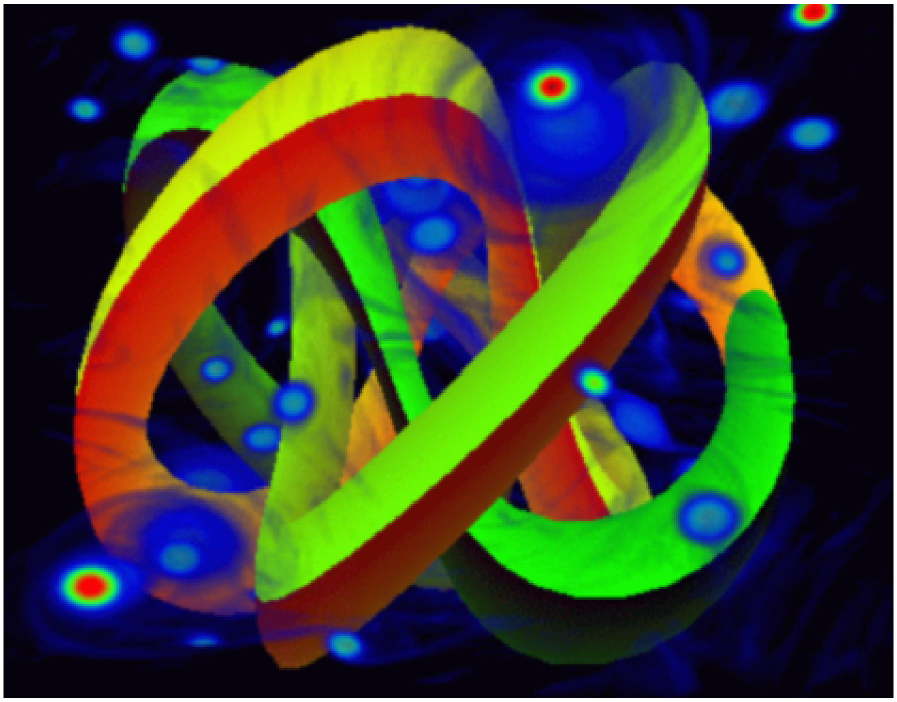


Figure 5-2 Volume and Opaque Geometry Integrated in a Single Scene

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 resources 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 5-3 shows `n` pipes rendering the same scene using one thread/process per pipe.

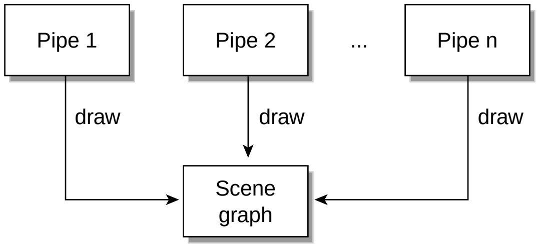


Figure 5-3 Multipipe Architecture

Rendering performance can be scaled using multiple compositing schemes. Figure 5-4 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.

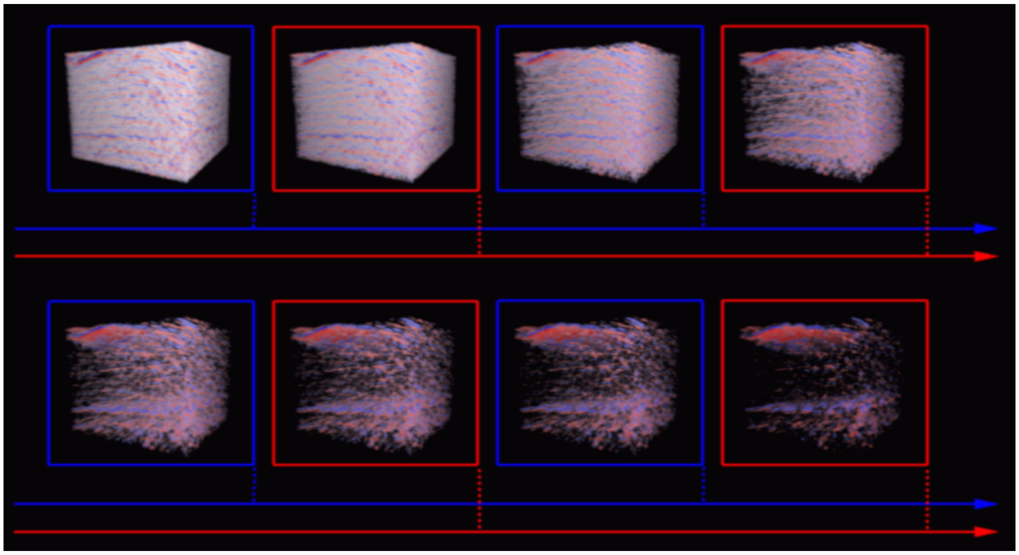


Figure 5-4 DPLEX Decomposition

Figure 5-5 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.

DB decomposition allows applications to linearly scale the texture memory size and fill rate performance with the number of graphics pipes on the system.

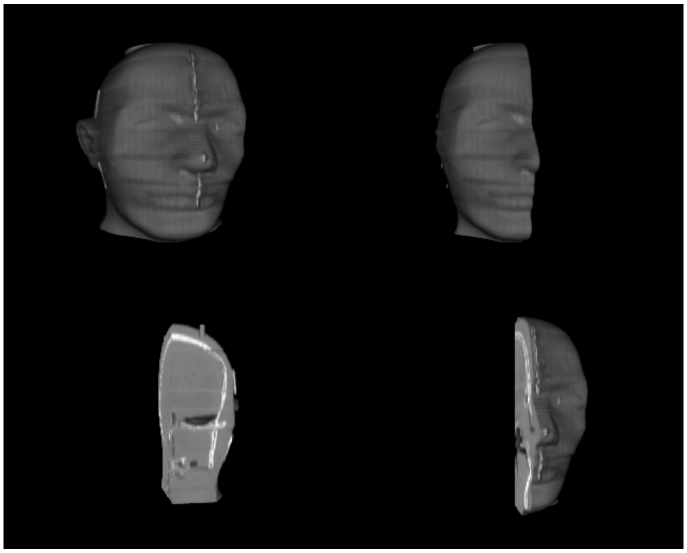


Figure 5-5 DB Decomposition

Visualizing Large Data

As the power of computing platforms and acquisition devices increase, applications using numeric simulations or data-acquisition techniques must process more and more data, also called *large data*. Some examples of these applications are in the scientific and energy domain. Here, *large data* means data larger than what the local resources can handle. This means the data to be visualized will reside on slower and larger storage peripherals like main memory or disks instead of local graphics resources. This data must migrate from the storage media to the graphics pipeline within the frame rate constraint. From this point of view, data migration becomes the main bottleneck for visualization. Figure 5-6 illustrates the management of large data and graphics resources.

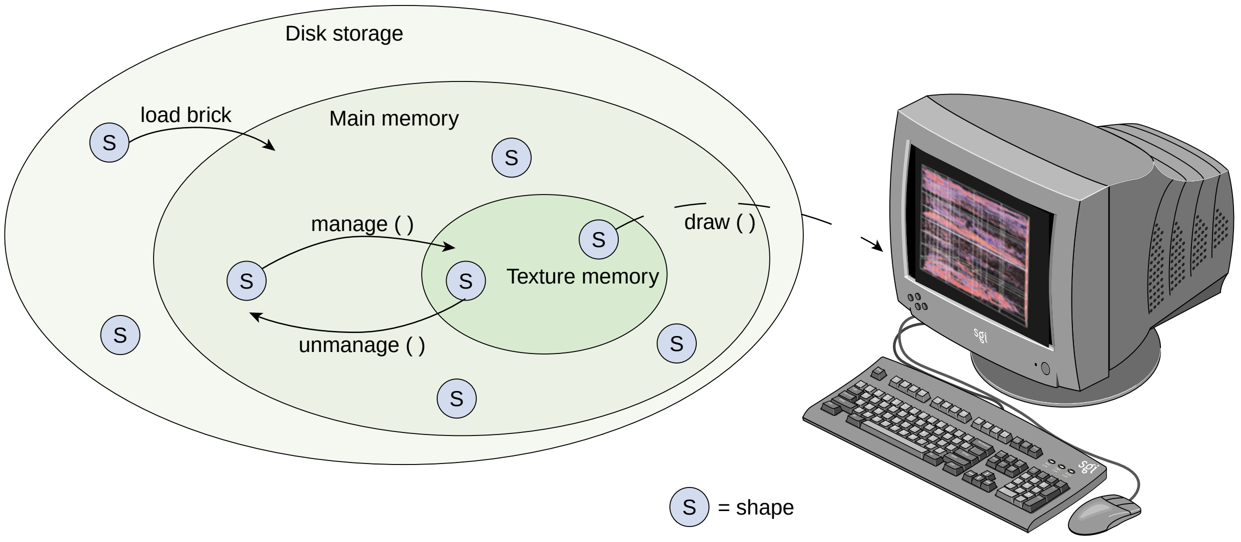


Figure 5-6 Large Data and Resource Management across Multiple Devices

Bricking

To handle these issues, OpenGL Volumizer can benefit from the Onyx 3000 series architecture by exploiting the high bandwidths and low latencies of such systems. The data transfer process can be supported by dividing the whole volumetric data into smaller components called *bricks*. A brick represents one volume shape. The application controls the frame rate by moving these data bricks to the local texture memory from the various storage devices. This 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. In addition, TMRenderAction automatically bricks textures too big to fit in texture memory to allow them to be rendered using OpenGL. The following section briefly mentions two techniques that can be used by large data visualization applications for interactive rendering of the data.

Volume Roaming

Volume roaming is an efficiency technique that allows the user to explore large volumetric data using a volumetric probe, which can be interactively moved inside the volume. The probe allows the user to have a viewing window and to concentrate on a specific section of the whole data set.

Figure 5-7 illustrates volume roaming with a 3D probe.

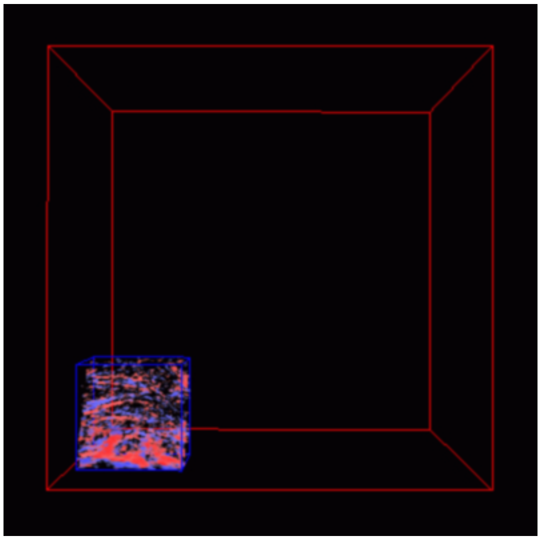


Figure 5-7 Volume Roaming with a 3D Probe

Volume roaming can employ the following techniques to achieve improved performance:

Intelligent texture management	Uses predictive texture downloads to maintain near constant frame rates during user motion.
Intelligent memory management	Allows roaming through a data set that may not fit in main memory.
Toroidal mapping	Extends the concepts of toroidal mapping from clip-textures to volume roaming. The granularity of the texture element is a volume brick here rather than a texel.

Multi-resolution Volume Rendering

Multi-resolution volume rendering allows applications to interactively render huge volume data by assigning varying levels of volume detail (LOD), thus, making a trade-off between performance and image quality. Lower resolutions help improve performance since it limits the texture memory as well as the fill-rate consumption of the application. Many researchers have worked on multi-resolution techniques for interactive volume rendering, typically using an octree decomposition of the whole volume. Figure 5-8 illustrates multi-resolution volume rendering.

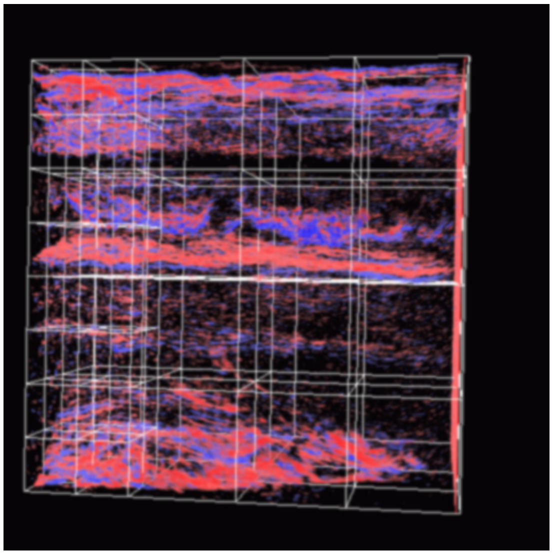


Figure 5-8 Multi-resolution Volume Rendering

The following techniques can be used to improve the performance while maintaining acceptable image quality:

- Coupling texture management with LOD switching to ensure near-constant frame rates
- Using the sorted order of bricks to determine the LOD to be rendered
- Using clipping geometries to optimize the use of texture memory available on the graphics subsystem
- Rendering higher resolutions during stages of minimal user interaction

In addition, time-varying techniques allow users to run a volume movie and can be easily implemented with the same techniques. Such techniques can be used, for example, to visualize animated fluid dynamics or crash analysis data.