OpenSceneGraph Quick Start Guide

A Quick Introduction to the Cross-Platform Open Source Scene Graph API

Paul Martz

Front cover image courtesy of Professor Mark Bryden and VRAC at Iowa State University.

Back cover top two images generated with 3DNature's NatureViewExpress and Visual Nature Studio with Scene Express.

Back cover bottom image courtesy of Andes Computer Engineering.

OpenSceneGraph Quick Start Guide

Paul Martz
Skew Matrix Software LLC
284 W. Elm St.
Louisville, CO 80027 USA
`pmartz@skew-matrix.com`

This book is dedicated to all developers new to scene graph technology.

Contents

Preface

This book is a concise introduction to OpenSceneGraph (OSG)—the cross-platform open source scene graph application programmer interface (API). Specifically, this book documents OSG v1.3. OSG plays a key role in the 3D application software stack. It's the middleware above the lower-level OpenGL hardware abstraction layer (HAL), providing extensive higher-level rendering, I/O, and spatial organization functionality to the 3D application.

For many years, OSG has thrived with only its source code as documentation. The OSG distribution includes several examples that illustrate various rendering effects, and methods for integrating OSG with end-user applications. These illustrative examples, along with the ability to step through core OSG in a debugger, have enabled several developers to become proficient in the OSG API.

Although source code has sufficed as documentation in the past, it is no substitute for more traditional forms of documentation. Manuals lend themselves quite easily to pedagogical instruments such as figures and tables, which are difficult to embed in source code. As OSG has grown and become more complex, lack of formal documentation has unacceptably lengthened the learning curve for new users. Prior to this book's release, the lack of formal OSG documentation has caused some developers to wonder if OSG is mature and robust enough to support professional-quality applications.

By mid-2006, both Don Burns and Robert Osfield had recognized the need for an OSG book. Don had a client, Computer Graphics Systems Development Corporation (CGSD), whose contract called for OSG documentation, so he subcontracted the documentation development to Paul Martz. Robert suggested that the first OSG book should be freely available and concise in nature. Thus, the *OpenSceneGraph Quick Start Guide* was born.

The *OpenSceneGraph Quick Start Guide* is a short programming guide that covers the basic and essential elements of the OSG API. It is the first in a series of planned books to document OSG with more comprehensive material to follow. The goals of the *OpenSceneGraph Quick Start Guide* are listed below.

- Provide new OSG developers with a quick and affordable introduction to OSG basics.
- Familiarize the reader with the OSG distribution and source code organization.
- Illustrate proper use of the commonly used elements of the OSG API.
- Direct the reader to sources of more thorough documentation.

In the spirit of open source, the OSGQSG is available for no charge as a PDF file. However, you can contribute to the OSG community by purchasing a full-color softbound copy. To place an online order for a copy of this book, visit the Lulu.com Web site and search for OpenSceneGraph.

```
http://www.lulu.com
```

Proceeds from sales of bound copies fund ongoing documentation revisions to ensure that the manual is always up-to-date.

Regardless of whether you download this book for free, or purchase a bound copy, your feedback on the book is essential in ensuring this documentation remains current and useful. Please post your comments to the osg-users email list. See the **Appendix, Where to Go From Here,** for information on the osg-users email list.

For information about new revisions to the book, visit the *OpenSceneGraph Quick Start Guide* Web site:

```
http://www.openscenegraph.org/osgwiki/pmwiki.php/
        Documentation/QuickStartGuide
```

This URL contains the most up-to-date information on obtaining the latest revision, downloading the book's example source code, and information on related publications.

Target Audience

This is a short book, and making it short was not an easy task. The scope is limited to a very narrow set of useful OSG functionality, and—just as importantly—the book targets a specific set of readers.

This book is intended for software developers who are new to OSG and considering using it in their application. This book doesn't preclude a particular genre of application software, but provides information that will be useful to the visualization and simulation markets, which traditionally have been OSG's strength.

OSG is a C++ API, so it is assumed that you have some knowledge of C++. In particular, you should be familiar with C++ features, such as public and private access, virtual functions, memory allocation, class derivation, and constructors and destructors. OSG makes extensive use of the standard template library (STL), so you should be familiar with STL constructs, especially **list, vector,** and **map**. Some familiarity with design patterns as implemented in C++ is useful, but is not required.

You should be familiar and comfortable working with data structures, such as trees and lists.

If you are considering using OSG in your application, you are probably familiar with 3D graphics. For this book, you should be familiar with OpenGL, the standard cross-platform low-level 3D graphics API. You should know about different coordinate spaces at the conceptual level, and should be comfortable specifying three-dimensional Cartesian coordinates as geometric vertex data. You should know that texture mapping essentially

applies an image to geometry, but you need not know the specifics of how the graphics hardware accomplishes this.

Some linear algebra experience is helpful. You should know that 3D locations are treated as vectors, and that graphics systems transform vectors by matrices as part of the rendering process. You should know that matrix concatenation combines transformations.

Recommended Reading

If you are a little rusty in any of the above areas, you might find the following list of recommend reading material useful.

- *OpenGL® Programming Guide, Fifth Edition,* by OpenGL ARB, Dave Shreiner, Mason Woo, Jackie Neider and Tom Davis (Addison-Wesley) [ARB05]
- *Geometric Tools for Computer Graphics,* by Philip Schneider and David H. Eberly (Morgan Kaufmann).
- *Real-Time Rendering, Second Edition,* by Tomas Akenine-Moller and Eric Haines (AK Peters).
- *Computer Graphics, Principles and Practice, Second Edition,* by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes (Addison-Wesley).
- *The C++ Programming Language, Third Edition,* by Bjarne Stroustrup (Addison-Wesley).

Organization of the Book

The *OpenSceneGraph Quick Start Guide* is composed of three main chapters and an appendix.

Chapter 1, An Overview of Scene Graphs and OpenSceneGraph, opens with a brief history of OSG's origins, followed by instructions for obtaining and installing OSG and how to use some examples and applications included in the OSG distribution. The chapter includes an introduction to the concept of a scene graph, followed by an overview of OSG and its organization.

In **Chapter 2, Building a Scene Graph,** you'll get your hands dirty assembling OSG data structures for storing and rendering geometry. Core OSG fundamentals, such as referenced pointers, scene graph nodes, drawable geometry, and state (including texture mapping and lighting) are described. This chapter also describes the osgText node kit for quickly adding text to your scene, as well as file I/O for accessing stored scene graph data and images. You'll leave the chapter with a firm grasp of how to use OSG to build a scene graph that displays a variety of geometry.

In **Chapter 3, Using OpenSceneGraph in Your Application,** you'll learn to do just that. The final chapter describes rendering, positioning and orienting the viewpoint, and animating and dynamically modifying your scene graph.

Finally, the **Appendix, Where to Go From Here,** explains where you can find more information about OSG and how you can become involved in the OSG community.

Conventions

This book uses the following style conventions:

- **Bold**—OSG classes and namespace names, OSG and OpenGL API methods and entry points, OSG and OpenGL types.
- *Italics*—Variables, parameter names, arguments, matrices, and spatial dimensions (such as *x, y,* and *z*).
- `Monospace`—Code listings, short code segments within a text paragraph, enumerants, and constants.

Furthermore, URL Web addresses are set aside from the text and use a `monospace font` and *italics* are used to introduce new terminology.

About the Author

Paul Martz is the president of Skew Matrix Software LLC, which provides custom software development, documentation, and developer training services. Paul has been involved in 3D graphics software development since 1987 and is the author of *OpenGL® Distilled* [Martz06]. He plays drums and provides music instruction, and is known to enjoy an occasional game of poker.

Acknowledgements

Obviously, there would be no OpenSceneGraph book if there were not an OpenSceneGraph. Both Don Burns and Robert Osfield have been instrumental in OSG's genesis and development—thanks to them both. But I'd be remiss in not thanking the entire OSG community, over 1600 strong, for their support and contribution to this important open source standard.

I'd like to thank Robert Osfield of OpenSceneGraph Professional Services for suggesting that the first OpenSceneGraph book should take the form of a free "quick start" guide. I'd also like to thank Roy Latham of CGSD and Don Burns of Andes Computer Engineering for funding the book's initial revision.

Thanks (again) to Robert Osfield, and also Leandro Motta Barros, who both wrote partial OSG documentation in the past. These efforts served as stepping stones for this book.

Thanks to Ben Discoe and the Virtual Terrain Project. This book's example code uses a tree image from their foliage library.

Many in the OSG community served as technical reviewers for this book, or have provided other assistance. I'd like to thank Sohaib Athar, Ellery Chan, Edgar Ellis, Andreas Goebel, Chris "Xenon" Hanson, Farshid Lashkari, Gordon Tomlinson, and John Wojnaroski. All of these individuals have contributed in some way, whether they were aware of it or not.

Finally, I must thank Deedre Martz for providing professional copyediting services. Once again, she's helped hide my sloppy writing style.

1 An Overview of Scene Graphs and OpenSceneGraph

This first chapter provides you with an introduction to the concept of a scene graph. You'll learn about OSG's history and organization, get a glimpse of its capabilities, learn how to obtain and install OSG, and run a few simple examples. This chapter helps you become familiar with scene graphs and OSG, but it contains no details on writing OSG-based applications. Chapters 2 and 3 cover that topic in more detail. This chapter is purely an introduction.

1.1 History of OpenSceneGraph

In 1997, Don Burns was employed as a software consultant at Silicon Graphics, Inc. (now simply SGI) with an after-hours interest in hang gliding. Inevitably, his dual-interest in computer graphics and hang gliding, along with his access to high-end rendering hardware, resulted in his development of a hang gliding simulator that ran on SGI Onyx systems using the (SGI proprietary) Performer scene graph.

Encouraged by fellow hang gliding enthusiasts to make his simulator available on more affordable hardware, Don began experimenting with Mesa3D on a Linux system with 3dfx Voodoo hardware. While this system provided acceptable OpenGL support, scene graphs were not available on Linux at that time. To fill this requirement, Don began to write "SG," a simplified Performer-like scene graph. His emphasis with SG was simplicity and ease of use. SG filled the scene graph requirement and allowed his hang gliding simulator to run on low-cost Linux systems.

In 1998, Don met Robert Osfield on an email list for hang gliding enthusiasts. Robert was working for Midland Valley Exploration, a Glasgow, Scotland, oil and gas company, so also had an interest in computer graphics and visualization. The two began to collaborate on improving the simulator. Robert was a supporter of open source, and suggested continuing to develop SG as a standalone open source scene graph project, with Robert as

the project lead. The name was changed to OpenSceneGraph, and initially nine people were on the osg-users email list.

In late 2000, Brede Johansen made the first major public contribution to OpenSceneGraph when he added the OpenFlight OSG plugin module. He developed the plugin while employed at Kongsberg Maritime Ship Simulation, Kongsberg, Norway, which ships the OSG-based SeaView R5 visual system product.

Also in the year 2000, Robert left his salaried position to develop OSG on a fulltime basis, doing business as OpenSceneGraph Professional Services. At this time, he designed and implemented many of the core features you see in OSG today. This was accomplished without clients or pay.

Don had moved on to Keyhole Technologies (now the Google Earth department of Google), and subsequently resigned in 2001. He also formed his own company, Andes Computer Engineering, based in San Jose, California, primarily to continue with OSG development. The first OpenSceneGraph birds-of-a-feather (BOF) meeting occurred at SIGGRAPH 2001. While only 12 people attended, the audience included representatives from Magic Earth, who were looking for an open source scene graph library to support their oil and gas application. They decided to contract with both Don and Robert for support and development, and became OSG's first paying customer.

Each year, attendance at the OSG BOF continued to grow. The osg-users email list membership continues to grow at a phenomenal pace, as Figure 1-1 illustrates. As this book went to press, osg-users had more than 1600 subscribers.

OSG features and add-on libraries were developed at a rapid pace. In 2003, the OSG companion library, Producer (originally called OSGMP), was created to provide a multipipe rendering capability for Magic Earth. In 2004, large database paging, terrain support, and shader support were added. 2006 included a complete revamp of the OpenFlight plugin, as well as the creation of osgViewer, an integrated library for managing and rendering views of a scene.

Today, several high-performance applications use OSG to manage rendering complex 2D

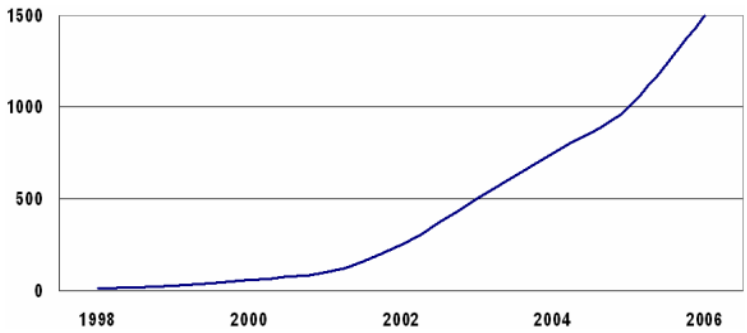


Figure 1-1
osg-users email list growth
The osg-users email list has grown significantly over time.

and 3D scenes. Though most OSG-based applications are in the visualization and simulation industries, OSG is found in nearly every field that employs 3D graphics, including geographic information systems (GIS), computer-aided design (CAD), modeling and digital content creation (DCC), database development, virtual reality, animation, gaming, and entertainment.

1.2 Installing OSG

The previous section describes OSG's origin. This section explains how to obtain and install OSG, so you can run OSG examples and develop your own OSG applications.

The OSG Wiki Web site [OSGWiki] offers many different packages and mechanisms for downloading OSG:

- Runtime binaries—Use an OSG runtime binary package to install the OSG libraries necessary to run OSG examples and applications.
- OSG source code—OSG developers should obtain a copy of the OSG source code. OSG provides many mechanisms for obtaining a complete OSG source code tree. You can obtain a compressed archive of a stable OSG release, download a nightly tarball (a compressed archive of the current source), or use Subversion (SVN) to check out the current source.
- OpenThreads—The core OSG libraries rely on OpenThreads for multithreading support. You must have an OpenThreads development environment to build OSG from source and build OSG-based applications.
- Third-party dependencies—If you're building OSG from source, some optional components require non-OSG software packages, such as libTIFF, libPNG, etc. If these third-party components aren't present on your build system, the optional OSG components fail to build.
- Sample dataset—This is a collection of 2D images, 3D models, and other data files.

The following sections describe how to obtain and install the OSG runtime binaries. Although OSG runs on a wide variety of platforms, only Apple Mac OS X, Fedora Linux, and Microsoft Windows are covered here. For information on obtaining OSG for other platforms, see the OSG Wiki Web site [OSGWiki].

If installable binaries aren't available for your platform, or to create your own OSG development kit, you'll need to build OSG from source code. For information on obtaining OSG source code, the third party dependencies, OpenThreads, and the sample dataset, refer to the OSG Wiki Web site [OSGWiki].

1.2.1 Hardware Requirements

OSG runs on a wide variety of hardware platforms and operating systems, and should run on most computer systems available today.

- Processor—OSG can be compiled to run on most contemporary CPUs. OSG is thread-safe and can take advantage of multi-processor and dual core architectures. OSG runs on both 32- and 64-bit processors.
- Graphics—Your system should feature an AGP or PCI-Express graphics card. OSG runs on most professional- and consumer-grade graphics hardware designed for modeling, simulation, and gaming. OSG requires graphics hardware with robust OpenGL support. Obtain and install the latest OpenGL device driver from your graphics hardware vendor. OSG's onboard graphics RAM requirements vary based on your usage, but 256MB is a good starting point. OSG runs on multi-pipe systems, and can take advantage of multiple graphics cards to increase rendering speed.
- RAM—The minimum system RAM requirement varies depending on the amount and type of data you intend to display with OSG. 1GB is a good starting point, but you might need more for larger data sets.
- Disk—Like RAM, amount of secondary storage depends on your data requirements. As with any application, fast RPM and large disk caches can reduce data load times.

1.2.2 Apple Mac OS X

OSG for Apple Mac OS X is available from the OSG Wiki Web site [OSGWiki] as a disk image (.dmg) file. It contains both run-time binaries and a full development environment. To install this package, perform the following steps.

1. From the OSG Wiki Web site [OSGWiki], select Downloads.
2. Download the OSG Universal Binaries for OSG v1.3. This is a .dmg file.
3. After the download completes, mount the .dmg file.
4. Drag the contents of the .dmg Frameworks folder into /System/Frameworks.
5. In /Library/Application Support, create a new folder called OpenSceneGraph. Drag Plugins from the .dmg into this new folder.

1.2.3 Fedora Linux

OSG is also available for other flavors of Linux. Many Linux environments provide a package installer interface that allows you to search for a package and install it. For example, in Ubuntu Linux, run the Synaptic Package Installer and search for

OpenSceneGraph. The search should find OSG runtime binary and development environment packages, which you can select and install.

To obtain the latest OSG binaries for Fedora Core 4, visit the OSG Wiki Web site [OSGWiki] and select Download. Under Binaries, select the Fedora Core 4 link.

1.2.4 Microsoft Windows

OSG run-time binaries for Microsoft Windows operating systems are available from the OSG Wiki Web site [OSGWiki] as an InstallShield executable. To install this package, perform the following steps.

1. From the OSG Wiki Web site [OSGWiki], select Downloads.
2. Download the OSG Win32 Binaries for OSG. This is an .exe file.
3. After the download completes, double-click the .exe file and follow the instructions for installation.

The default installation modifies environment variables in the registry. To make these changes take effect, either log out and log back in, or reboot your system.

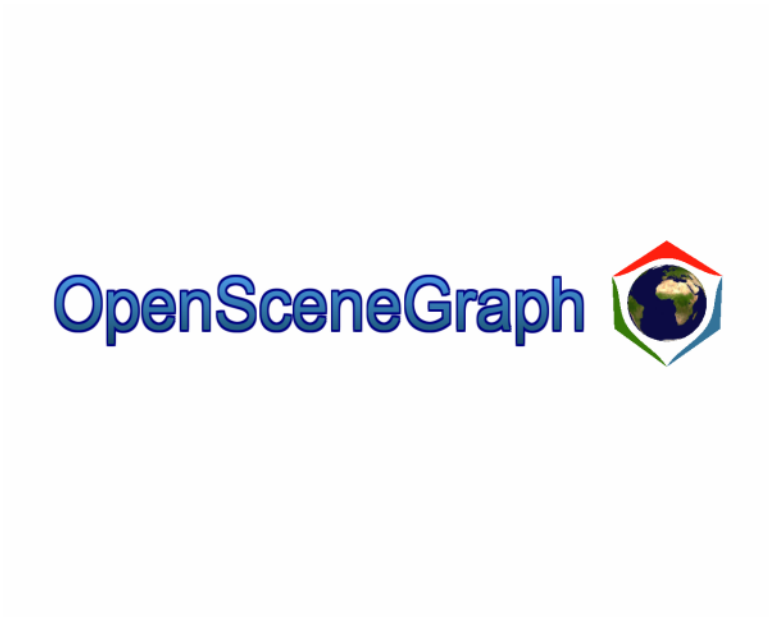


Figure 1-2
osglogo output
This figure displays the results of the `osglogo` command.

1.2.5 Verifying Your OSG Installation

After installing OSG, you should verify your installation. Perform the following steps:

1. Open a command shell prompt on your computer.
2. Enter the following command:

```
osgversion
```

This executes the osgversion application, which should output the OSG version number, as follows:

```
OpenSceneGraph Library 1.3
```

This simple step verifies that the system is able to find OSG executables (your PATH is set correctly), tells you what release version of OSG you're running, and minimally ensures that OSG is functional.

To verify that OSG can render on your system, execute the following command:

```
osglogo
```

This should display am image similar to Figure 1-2.

osglogo dynamically updates its scene to rotate the Earth. It also supports a trackball interface that allows you to spin the logo with the left mouse button.

1.3 Running osgviewer

In the last section, you executed osgversion and osglogo. While this allowed you to verify your installtion, the program are limited in their functionality. This section shows you how to run osgviewer, OSG's flexible and powerful model viewing tool. Load a simple model of a cow and display it with the following command:

```
osgviewer cow.osg
```

The results are shown in Figure 1-3.

The cow model is in OSG's own .osg file format. However, osgviewer supports the same file formats as OSG, many of which are enumerated in the section **OSG Plugins** later in this chapter.

Like osglogo, osgviewer allows you to interact with the model. By default, osgviewer exposes a trackball-like interface. To rotate the cow model, drag with your left mouse button. When you release the mouse button, the model will continue to rotate. You can zoom in or out using the right mouse button. Hit the space bar to return to the initial view.

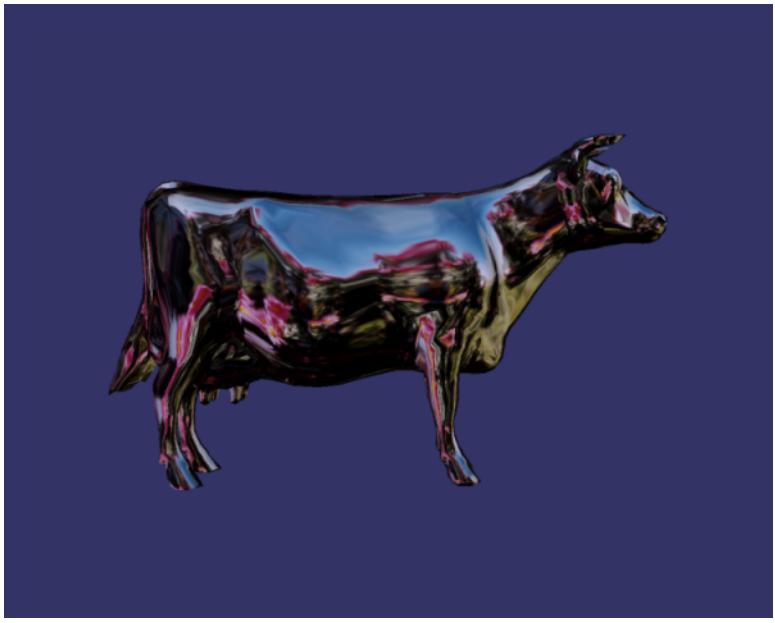


Figure 1-3
osgviewer output
This figure displays the results of the command `osgviewer cow.osg`. The osgviewer application displays a wide range of image and model files.

1.3.1 Getting Help

While in osgviewer, press the 'h' key to display a list of key commands and their functions. The '1' through '5' keys allow you to switch to different camera manipulation modes, which modify the way that the mouse controls the camera position; for now, stick with mode '1', trackball mode, which is the default. Many of the key commands control display modes, which the next section describes.

The help text shows that Escape causes osgviewer to exit. Press Escape now; osgviewer exits and returns you to the command line prompt.

Enter the following command to see the osgviewer command line options:

```
osgviewer --help
```

This causes osgviewer to display all command line options. The following describes a few of the commonly used options.

- --clear-color—This option allows you to set the clear, or background, color. If you issue the following command, for example, osgviewer uses a white background color:

```
osgviewer --clear-color 1.0,1.0,1.0,1.0 cow.osg
```

- --samples—This option enables hardware multisampling, more commonly known as full screen antialiasing. Its single numeric parameter is the number of samples per pixel. To enable multisampling with 16 subpixels, for example, use the following command:

  ```
  osgviewer --samples 16 cow.osg
  ```

- --image—This option causes osgviewer to load a single image and display it as a texture on q quadrilateral primitive, as shown in the following command.

  ```
  osgviewer --image osg256.png
  ```

In addition to command line arguments and key commands, you can also control osgviewer with several environment variables. To see the full help text for osgviewer, issue the following command at a shell prompt:

```
osgviewer --help-all
```

The following sections provide more details about using the osgviewer application.

1.3.2 Display Modes

Many of the osgviewer key commands specify display modes to control the appearance of the loaded model. Some of the commonly used commands are listed below.

- Polygon mode—Hit the 'w' key repeatedly to cycle between wireframe, point, and filled polygon rendering mode.
- Texture mapping—Hit the 't' key to toggle between textured and non-textured.
- Lighting—Disable and enable lighting with the 'l' key.
- Backface culling—The 'b' key toggles backface culling. This doesn't change the appearance of the cow.osg model, but it could affect other models' appearance and rendering performance.
- Fullscreen mode—Use the 'f' key to toggle between fullscreen and windowed rendering.

Take some time and experiment with combinations of these osgviewer commands. For example, to clearly see the polygonal structure of a model, go to wireframe mode, and disable texture mapping and lighting.

1.3.3 Environment Variables

Although OSG and the osgviewer application support many environment variables, there are two that you should become very familiar with. You will use them often when working with OSG.

File Search Path

The OSG_FILE_PATH environment variable specifies the search path used by OSG when loading image and model files. If you run `osgviewer cow.osg` and cow.osg isn't in the current directory, OSG finds and loads cow.osg because its directory path is specified in OSG_FILE_PATH.

Your runtime installation sets the OSG_FILE_PATH variable. You can add more directories to this variable. On Windows, separate each directory with a semicolon, and use a color on other platforms. If the variable is empty or not set, OSG only searches the current directory when loading image and model files.

Debug Message Display

OSG is capable of displaying a large amount of debugging information to **std::cout.** This is useful in developing OSG applications, because it provides insight into what OSG is doing. The OSG_NOTIFY_LEVEL environment variable controls how much debugging information OSG displays. You can set it to one of seven values for varying verbosity levels: ALWAYS (least verbose), FATAL, WARN, NOTICE, INFO, DEBUG_INFO, and finally DEBUG_FP (most verbose).

For typical OSG development, set OSG_NOTIFY_LEVEL to NOTICE, and adjust the value up or down the verbosity scale as necessary to receive more or less output.

1.3.4 Statistics Display

Particularly useful for performance measurements is the 's' key, which uses the **Statistics** class in the osgUtil library to gather and display rendering performance information. The 's' key cycles between four display modes:

1. Frame rate—osgviewer displays the number of frames rendered per second (FPS).
2. Traversal time—osgviewer displays the amount of time spent in each of the update, cull, and draw traversals, including a graphical display, Figure 1-4 illustrates.
3. Geometry information—osgviewer displays the number of rendered **osg::Drawable** objects, as well as the total number of vertices and primitives processed per frame.
4. None—osgviewer disables the statistics display.

Pressing the 's' key twice displays traversal time. Figure 1-4 illustrates the traversal time graphical display.

Figure 1-4
The traversal time graphical display
This shows a typical traversal time display for an application synchronized with a 60Hz monitor. The time spent in the update traversal, 0.04 milliseconds, is so insignificant that it doesn't appear in the graphical display. However, the cull and draw traversals, at 2.68 and 3.05 milliseconds respectively, display clearly as cyan and dark yellow bars spanning part of a single frame. The text output also indicates that the GPU takes 1.87 milliseconds to process the rendering commands, as measured by OpenGL.

The graphical display represents a series of rendered frames. Typically, rendering is synchronized to the monitor refresh rate to avoid rendering artifacts such as image tearing. In Figure 1-4, the monitor refresh rate is 60Hz, so each frame occupies $1/60^{th}$ of a second, or about 16.67 milliseconds. This display illustrates how much time is spent in the update, cull, and draw traversals. This feedback is essentially for analyzing performance problems to help determine the rendering stage of any application performance bottlenecks.

1.3.5 Recording an Animation

Developers require repeatable test cases to effectively tune and measure application rendering performance. To facilitate performance tuning, osgviewer allows you to easily record a camera motion sequence and play it back. This sequence is called an animation path.

While osgviewer is running, press the 'z' key. This causes osgviewer to immediately begin recording an animation path. Rotate the model and zoom in or out using the mouse; OSG records all camera movements. Finally, press the 'Z' (shift-z) key. This stops recording the animation path, and immediately plays it back. This will show you every camera motion made while the path was being recorded.

Exit osgviewer with the Escape key, and get a list of files in your current directory. You will see a new file, saved_animation.path. As its name implies, this file contains the recorded animation path. osgviewer wrote this file out when you hit the 'Z' key. To play the animation path, issue the following command:

```
osgviewer –p saved_animation.path cow.osg
```

When playing an animation path, osgviewer displays the elapsed time for the sequence to **std::cout.** If osgviewer doesn't display this information in your shell, press Escape to exit osgviewer. Set the OSG_NOTIFY_LEVEL environment variable to INFO and restart osgviewer.

1.4 Compiling OSG Applications

To build OSG-based applications, you need an OSG development environment consisting of header files and libraries. The OSG runtime binary distributions contain header files and optimized libraries. To create debuggable libraries, download and build the OSG and OpenThreads source code. Obtain both OSG and OpenThreads source from the OSG Wiki Web site [OSGWiki], Downloads section. The OSG Wiki Web site contains instructions for how to build OSG.

To successfully compile an OSG-based application, you must set the correct include file paths so that the compiler can find the necessary header files. Add the following two directories to the compiler's include search path.

```
<parent>/OpenSceneGraph/include
<parent>/OpenThreads/include
```

Replace <parent> with the top-level directory where you installed both OpenSceneGraph and OpenThreads. On a Linux system using gcc, specify these include paths with the –I command line option. <parent> is uaually /usr/local/include on a Linux system, so the include paths on a gcc command line appear as follows.

```
-I/use/local/include/OpenSceneGraph/include
-I/usr/local/include/OpenThreads/include
```

In Microsoft Visual Studio, add the appropriate directories as Additional Include Directories in the Project Properties dialog, C/C++ options.

Similarly, you need to tell the linker where to fnd the OSG libraries. On Linux systems, OSG libraries reside in /usr/local/lib, and the linker can find them there without use of the –L option.

The Microsoft Visual Studio build leaves libraries in the following two source code tree directories.

```
<parent>/OpenSceneGraph/lib/win32
<parent>/OpenThreads/lib/win32
```

Add these directories as Additional Library Directories in the Project Properties dialog, Linker options.

Finally, tell the linker which OSG libraries to link with. As section **1.6.3 Components** describes, OSG is made up of several different libraries, each providing different functionality. For a simple OSG-based application using the osgViewer, osgDB, osgUtil, and osg libraries, the gcc command line looks like the following.

```
-losgViewer –losgDB –losgUtil –losg
```

In Microsoft Visual Studio, add the library file names as Additional Dependencies in the Project Property dialog, Linker options. Under Microsoft Windows, OSG builds debug and release libraries with different names. For a release build, add the following library file names.

```
libosgViewer.lib libosgDB.lib libosgUtil.lib libosg.lib
```

For a debug build, insert 'd' before the file extension.

```
libosgViewerd.lib libosgDBd.lib libosgUtild.lib libosgd.lib
```

These libraries are an example, and the actually libraries your application links with depends on what OSG functionality your application uses. You might need to link with other libraries, such as the osgText, osgShadow, and osgGA libraries.

All this is handled for you on a Mac OS X system if you simply tell Xcode to use the OSG framework.

If you fail to configure the compiler and linker with the correct options, your application build fails with errors such as "unable to open include file," "unable to find library file," and "unresolved symbol." If you encounter any of these errors, examine the error message

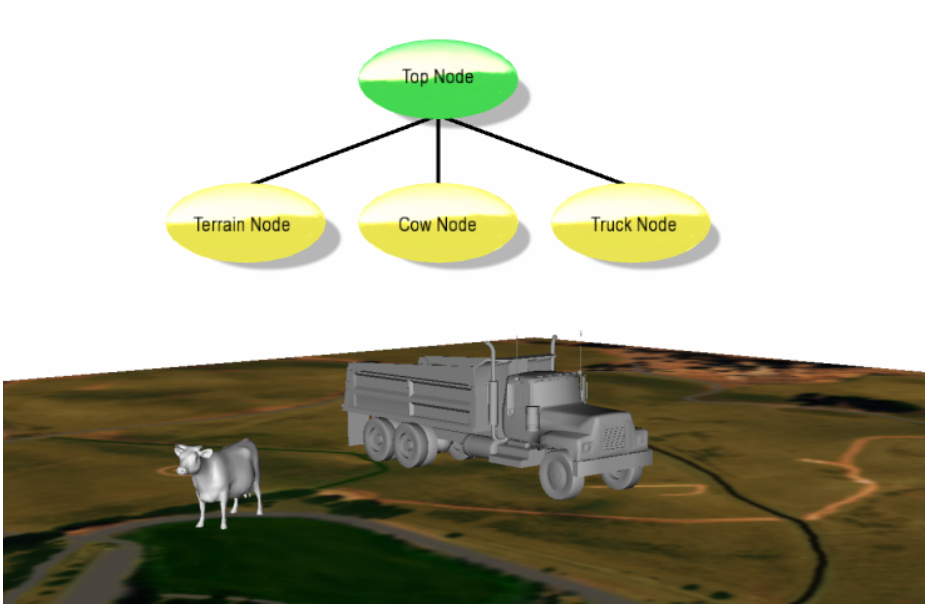


Figure 1-5
A simple, abstract scene graph
To render a scene consisting of terrain, a cow, and a truck, the scene graph takes the form of a top-level node with three child nodes. Each child node contains the geometry to draw its object.

closely and verify that you've specified the correct compiler and linker options.

1.5 Introduction to Scene Graphs

The previous sections focus on where OSG came from and how to install and run it on your system. If you've followed the instructions in this chapter so far, you've succeeded in creating a few interesting images on your screen using OSG. The rest of this book explores OSG in increasing depth. The current section describes scene graphs at the conceptual level. Section **1.6 Overview of OpenSceneGraph** provides a high-level overview of OSG's feature set. Finally, **Chapter 2, Building a Scene Graph,** and **Chapter 3, Using OpenSceneGraph in Your Application,** describe portions of OSG's application interface.

A scene graph is a hierarchical tree data structure that organizes spatial data for efficient rendering. Figure 1-5 illustrates an abstract scene graph consisting of terrain, a cow, and a truck.

The scene graph tree is headed by a top-level root node. Beneath the root node, group nodes organize geometry and the rendering state that controls their appearance. Root nodes and group nodes can have zero or more children. (However, group nodes with zero children are essentially no-ops.) At the bottom of the scene graph, leaf nodes contain the actual geometry that makes up the objects in the scene.

Applications use group nodes to organize and arrange geometry in a scene. Imagine a 3D database containing a room with a table and two identical chairs. You could organize a scene graph for this database in many ways. Figure 1-6 shows one example organization. The root node has four group node children, one for the room geometry, one for the

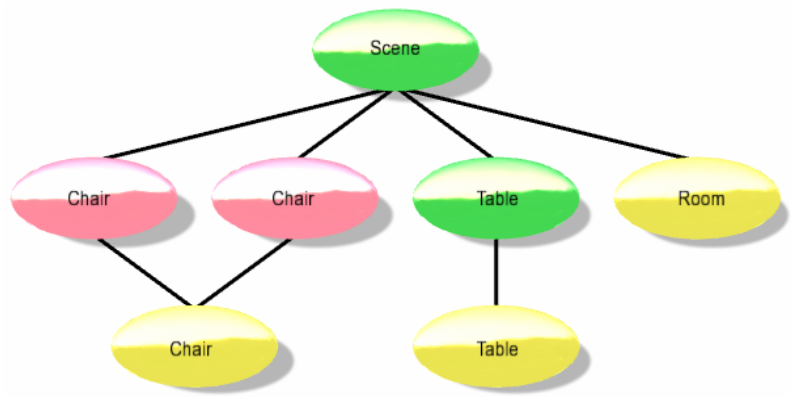


Figure 1-6
A typical scene graph
Group nodes can have several children, and allow applications to logically organize geometric and state data. In this case, the two chair group nodes translate their single child to two different locations, producing the appearance of two chairs.

table, and one for each chair. The chair group nodes are color-coded red to indicate that they transform their children. There is only one chair leaf node because the two chairs are identical—their parent group nodes transform the chair to two different locations to produce the appearance of two chairs. The table group node has a single child, the table leaf node. The room leaf node contains the geometry for the floor, walls, and ceiling.

Scene graphs usually offer a variety of different node types that offer a wide range of functionality, such as switch nodes that enable or disable their children, level of detail (LOD) nodes that select children based on distance from the viewer, and transform nodes that modify transformation state of child geometry. Object-oriented scene graphs provide this variety using inheritance; all nodes share a common base class with specialized functionality defined in the derived classes.

The large variety of node types and their implicit spatial organization ability provide data storage features that are unavailable in traditional low-level rendering APIs. OpenGL and Direct3D focus primarily on abstracting features found in graphics hardware. Although graphics hardware allows storage of geometric and state data for later execution (such as display lists or buffer objects), low-level API features for spatial organization of that data are generally minimal and primitive in nature, and inadequate for the vast majority of 3D applications.

Scene graphs are middleware, which are built on top of low-level APIs to provide spatial organization capabilities and other features typically required by high-performance 3D applications. Figure 1-7 illustrates a typical OSG application stack.

1.5.1 Scene Graph Features

Scene graphs expose the geometry and state management functionality found in low-level rendering APIs, and also provide additional features and capabilities, such as the following:

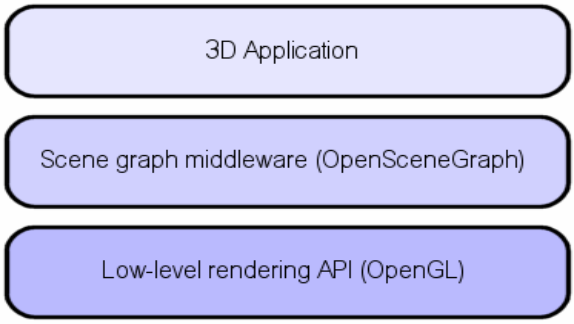


Figure 1-7
The 3D application stack
Rather than interface directly with the low-level rendering API, many 3D applications require additional functionality from a middleware library, such as OpenSceneGraph.

- Spatial organization—The scene graph tree structure lends itself naturally to intuitive spatial organization.
- Culling—View frustum and occlusion culling on the host CPU typically reduces overall system load by not processing geometry that doesn't appear in the final rendered image.
- LOD—Viewer-object distance computation on bounding geometry allows objects to be efficiently rendered at varying levels of detail. Furthermore, portions of a scene can load from disk when they are within a specified viewer distance range, and deleted from memory when they are beyond that distance.
- Translucency—Correct and efficient rendering of translucent (non-opaque) geometry requires all translucent geometry to render after all opaque geometry. Furthermore, translucent geometry should be sorted by depth and rendered in back-to-front order. These operations are commonly supported by scene graphs.
- State change minimization—To maximize application performance, redundant and unnecessary state changes should be avoided. Scene graphs commonly sort geometry by state to minimize state changes, and OpenSceneGraph's state management facilities eliminate redundant state changes.
- File I/O—Scene graphs are an effective tool for reading and writing 3D data from disk. Once loaded into memory, the internal scene graph data structure allows the application to easily manipulate dynamic 3D data. Scene graphs can be an effective intermediary for converting from one file format to another.
- Additional high-level functionality—Scene graph libraries commonly provide high-level functionality beyond that typically found in low-level APIs, such as full-featured text support, support for rendering effects (such as particle effects and shadows), rendering optimizations, 3D model file I/O support, and cross-platform access to input devices and render surfaces.

Nearly all 3D applications require some of these features. As a result, developers who build their application directly on low-level APIs typically resort to implementing many of these features in their application, which increases development costs. Using an off-the-shelf scene graph that already fully supports such features enables rapid application development.

1.5.2 How Scene Graphs Render

A trivial scene graph implementation allows applications to store geometry and execute a draw traversal, during which all geometry stored in the scene graph is sent to the hardware as OpenGL commands. However, such an implementation lacks many of the features described in the previous section. To allow for dynamic geometry updates, culling, sorting, and efficient rendering, scene graphs typically provide more than a simple draw traversal. In general, there are three types of traversals:

- Update—The update traversal (sometimes referred to as the application traversal) allows the application to modify the scene graph, which enables dynamic scenes. Updates are accomplished either directly by the application or with callback functions assigned to nodes within the scene graph. Applications use the update traversal to modify the position of a flying aircraft in a flight simulation, for example, or to allow user interaction using input devices.
- Cull—During the cull traversal, the scene graph library tests the bounding volumes of all nodes for inclusion in the scene. If a leaf node is within the view, the scene graph library adds a reference to the leaf node geometry to a final rendering list. This list is sorted by opaque versus translucent, and translucent geometry is further sorted by depth.
- Draw—In the draw traversal (sometimes referred to as the render traversal), the scene graph traverses the list of geometry created during the cull traversal and issues low-level graphics API calls to render that geometry.

Figure 1-8 illustrates these traversals.

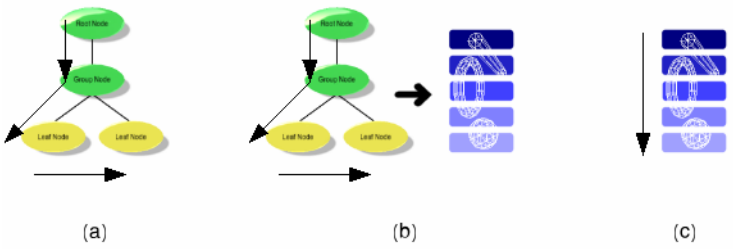


Figure 1-8
Scene graph traversals
Rendering a scene graph typically requires three traversals. In (a), the update traversal modifies geometry, rendering state, or node parameters to ensure the scene graph is up-to-date for the current frame. In (b), the cull traversal checks for visibility, and places geometry and state references in a new structure (called the *render graph* in OSG). In (c), the draw traversal traverses the render graph and issues drawing commands to the graphics hardware.

Typically, these three traversals are executed once for each rendered frame. However, some rendering situations require multiple simultaneous views of the same scene. Stereo rendering and multiple display systems are two examples. In these situations, the update traversal is executed once per frame, but the cull and draw traversals execute once per view per frame. (That's twice per frame for simple stereo rendering, and once per graphics card per frame on multiple display systems.) This allows systems with multiple processors and graphics cards to process the scene graph in parallel. The cull traversal must be a read-only operation to allow for multithreaded access.

1.6 Overview of OpenSceneGraph

OSG is a set of open source libraries that primarily provide scene management and graphics rendering optimization functionality to applications. It's written in portable ANSI C++ and uses the industry standard OpenGL low-level graphics API. As a result, OSG is cross platform and runs on Windows, Mac OS X, and most varieties of UNIX and Linux operating systems. Most of OSG operates independently of the native windowing system. OSG includes code to support some windowing system specific functionality, such as PBuffers, however.

OSG is open source, and is available under a modified GNU Lesser General Public License, or *Library GPL* (LGPL) software license. OSG's open source nature has many benefits:

- Improved quality—OSG is reviewed, tested, and improved by many members of the OSG community. Over 200 developers contributed to OSG v1.2.
- Improved application quality—To produce quality applications, application developers need intimate knowledge of the underlying middleware. If the middleware is closed source, this information is effectively blocked and limited to vendor documentation and customer support. Open source allows application developers to review and debug middleware source code, which allows free access to code internals
- Reduced cost—Open source is free, eliminating the up-front purchase price.
- No intellectual property issues—There is no way to hide software patent violations in code that is open source and easily readable by all.

OSG support is easy to find by subscribing to the osg-users email list or by contracting with professional support. For more information, see the **Appendix, Where to Go From Here.**

1.6.1 Design and Architecture

OSG is designed up front for portability and scalability. As a result, it is useful on a wide variety of platforms, and renders efficiently on a large number and variety of graphics hardware. OSG is designed to be both flexible and extensible to allow adaptive development over time. As a result, OSG is able to meet customer needs as they arise.

To enable these design criteria, OSG is built with the following concepts and tools:

- ANSI standard C++.
- C++ Standard Template Library (STL).

- Design patterns. [Gamma95]

These tools allow developers using OSG to develop on the platform of their choice and deploy on any platform the customer requires.

1.6.2 Naming Conventions

The following list enumerates the naming conventions used by the OSG source code. These conventions are not always strictly enforced. The OSG plugins contain many convention violations, for example.

- Namespaces—OSG namespace names start with a lower-case letter, but can be upper case for clarity. Examples: **osg, osgSim, osgFX.**
- Classes—OSG class names start with an upper case letter. If the class name is composed of multiple words, each word starts with an upper-case letter. Examples: **MatrixTransform, NodeVisitor, Optimizer.**
- Class methods—Names of methods within an OSG class start with a lower-case letter. If the method name is composed of multiple words, each additional word starts with an upper-case letter. Examples: **addDrawable(), getNumChildren(), setAttributeAndModes().**
- Class member variables—Names of member variables within a class use the same convention as method names.
- Templates—OSG template names are lower case with multiple words separated by underscores. Examples: **ref_ptr<>, graph_array<>, observer_ptr<>.**
- Statics—Static variables and functions begin with *s_* and otherwise use the same naming conventions as class member variables and methods. Examples: *s_applicationUsage, a_ArrayNames().*
- Globals—Global class instances begin with *g_*. Examples: *g_NotifyLevel, g_readerWriter_BMP_Proxy.*

1.6.3 Components

The OSG runtime exists as a set of dynamically loaded libraries (or shared objects) and executables. These libraries fall into five conceptual categories:

- The Core OSG libraries provide essential scene graph and rendering functionality, as well as additional functionality typically required by 3D graphics applications.
- NodeKits extend the functionality of core OSG scene graph node classes to provide higher-level node types and special effects.
- OSG plugins are libraries that read and write 2D image and 3D model files.

- The interoperability libraries allow OSG to easily integrate into other environments, including scripting languages such as Python and Lua.
- An extensive collection of applications and examples provide useful functionality and demonstrate correct OSG usage.

Figure 1-9 illustrates OSG's architectural organization. The following sections discuss these libraries in more detail.

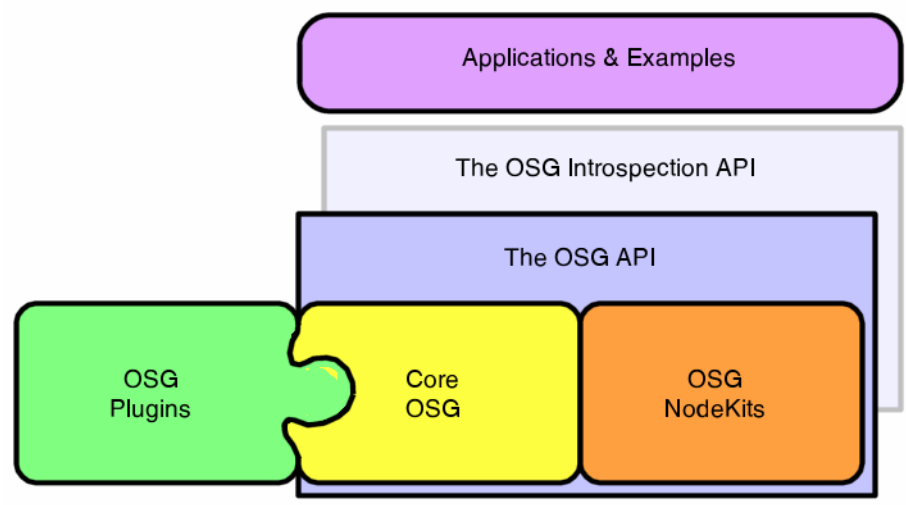


Figure 1-9
OSG architecture
The Core OSG libraries provide functionality to both the application and the NodeKits. Together, the Core OSG libraries and NodeKits make up the OSG API. One of the Core OSG libraries, osgDB, provides access to 2D and 3D file I/O by managing the OSG plugins.

Core OSG

Core OSG provides core scene graph functionality, classes, and methods for operating on the scene graph, additional application functionality typically required by 3D graphics applications, and access to the OSG plugins for 2D and 3D file I/O. It consists of four libraries:

- The osg library—This library contains the scene graph node classes that your application uses to build scene graphs. It also contains classes for vector and matrix math, geometry, and rendering state specification and management. Other classes in osg provide additional functionality typically required by 3D applications, such as argument parsing, animation path management, and error and warning communication.

- The osgUtil library—This utility library contains classes and functions for operating on a scene graph and its contents, gathering statistics and optimizing a scene graph, and creating the render graph. There are also classes for geometric operations, such as Delaunay triangulation, triangle stripification, and texture coordinate generation.
- The osgDB library—This library contains classes and functions for creating and rendering 3D databases. It contains a registry of OSG plugins for 2D and 3D file I/O, as well as a class for accessing those plugins. The osgDB database pager supports dynamic loading and unloading of large database segments.
- The osgViewer library—New in OSG v1.3, this library contains classes that manage views into the scene. osgViewer integrates OSG with a wide variety of windowing systems.

The v1.3 release contains a library called osgGA for adaptation of GUI events. However, in the near future, OSG will be redesigned to move portions of osgGA functionality into osgViewer, and eliminate it as a standalone library.)

The following sections discuss the four core libraries in detail.

The osg Library

The osg library is the heart of OpenSceneGraph. It defines the core nodes that make up the scene graph, as well as several additional classes that aid in scene graph management and application development. Some of these classes are described briefly below. Chapter 2 covers them in greater detail, and shows you how to use them in your application.

Scene Graph Classes

Scene graph classes aid in scene graph construction. All scene graph classes in OSG are derived from **osg::Node.** Conceptually, root, group, and leaf nodes are all different node types. In OSG, these are all ultimately derived from **osg::Node,** and specialized classes provide varying scene graph functionality. Also, the root node in OSG is not a special node type; it's simply an **osg::Node** that does not have a parent.

- **Node**—The **Node** class is the base class for all nodes in the scene graph. It contains methods to facilitate scene graph traversals, culling, application callbacks, and state management.

The osg Library

Namespace: osg

Header files: <OSG_DIR>/include/osg

Windows library files: osg.dll and osg.lib

Linux and Mac OS X library file: libosg.lib

- **Group**—The **Group** class is the base class for any node that can have children. It is a key class in the spatial organization of scene graphs.
- **Geode**—The **Geode** (or *Geometry Node*) class corresponds to the leaf node in OSG. It has no children, but contains **osg::Drawable** objects (see below) that contain geometry for rendering.
- **LOD**—The **LOD** class displays its children based on their distance to the view point. This is commonly used to create a varying level of detail representation for objects in a scene.
- **MatrixTransform**—The **MatrixTransform** class contains a matrix that transforms the geometry of its children, allowing scene objects to be rotated, translated, scaled, skewed, projected, etc.
- **Switch**—The Switch class contains a Boolean mask to enable or disable processing of its children.

This is an incomplete list of OSG node types. Other node types exist, such as **Sequence** and **PositionAttitudeTransform**. Refer to the osg library header files for information on these node types.

Geometry Classes

The Geode class is the OSG leaf node, and it contains geometric data for rendering. Use the following classes to store geometric data in a Geode.

- **Drawable**—The **Drawable** class is the base class for storing geometric data, and **Geode** maintains a list of **Drawable**s. **Drawable** is a pure virtual class and can't be instantiated directly. You must use a derived class, such as **Geometry** or **ShapeDrawable** (which allows your application to draw predefined shapes such as spheres, cones, and boxes).
- **Geometry**—The **Geometry** class, in conjunction with the **PrimitiveSet** class, act as high-level wrappers around the OpenGL vertex array functionality. **Geometry** stores the vertex arrays vertex, texture coordinate, color, and normal arrays.
- **PrimitiveSet**—The **PrimitiveSet** class provides high-level support for the OpenGL vertex array drawing commands. Use this class to specify the types of primitives to draw from the data stored in the associated **Geometry** class.
- Vector classes (**Vec2, Vec3,** etc.)—OSG provides a set of predefined 2-, 3-, and 4-element vectors of type float or double. Use these vectors to specify vertices, colors, normals, and texture coordinates.
- Array classes (**Vec2Array, Vec3Array,** etc.)—OSG defines several commonly used array types, such as **Vec2Array** for texture coordinates. When specifying vertex array data, your application stores geometric data in these arrays before passing them to **Geometry** objects.

This might sound very confusing, but it can be summarized as follows: **Geode**s are leaf nodes in the scene graph that store **Drawable**s. **Geometry** (one type of **Drawable**) stores vertex array data and vertex array rendering commands specific to that data. The data itself is composed of arrays of vectors. **Chapter 2, Building a Scene Graph,** covers all of this in greater detail.

State Management Classes

OSG provides a mechanism for storing the desired OpenGL rendering state in the scene graph. During the cull traversal, geometry with identical states are grouped together to minimize state changes. During the draw traversal, the state management code keeps track of the current state to eliminate redundant state changes.

Unlike other scene graphs, OSG allows state to be associated with any scene graph node, and state is inherited hierarchically during a traversal.

- **StateSet**—OSG stores a collection of state values (called modes and attributes) in the **StateSet** class. Any **osg::Node** in the scene graph can have a **StateSet** associated with it.
- Modes—Analogous to the OpenGL calls **glEnable()** and **glDisable(),** modes allow you to turn on and off features in the OpenGL fixed-function rendering pipeline, such as lighting, blending, and fog. Use the method **osg::StateSet::setMode()** to store a mode in a **StateSet.**
- Attributes—Attributes allow your application to specify state parameters, such as the blending function, material properties, and fog color. Use the method **osg::StateSet::setAttribute()** to store a mode in a **StateSet.**
- Texture attributes and modes—These attributes and modes apply to a specific texture unit in OpenGL multitexturing. Unlike OpenGL, there is no default texture unit; your application must supply the texture unit when setting texture attributes and modes. To set these state values and specify their texture unit, use the **StateSet** methods **setTextureMode()** and **setTextureAttribute().**
- Inheritance flags—OSG provides flags for controlling how state is inherited during a scene graph traversal. By default, state set in a child node overrides the same state set in a parent node. However, you can force parent state to override child node state, and you can also specify that child state be protected from parent overriding.

This state system has proven itself to be very flexible. All new state added to the OpenGL specification, including the addition of the OpenGL Shading Language [Rost06], has fit easily into the OSG state system.

Utilities and Other Classes

Finally, the osg library contains several useful classes and utilities. Some of these deal with the OSG reference-counted memory scheme, which helps avoid memory leaks by deleting

unreferenced memory. **Chapter 2, Building a Scene Graph,** discusses reference-counted memory in detail.

- **Referenced**—The **Referenced** class is the base class for all scene graph nodes and many other objects in OSG. It contains a reference count to track memory usage. If an object is of a type derived from **Referenced** and its reference count reaches zero, its destructor is called and memory associated with the object is deleted.
- **ref_ptr<>**—The **ref_ptr<>** template class defines a *smart pointer* to its template argument. The template argument must be derived from **Referenced** (or support an identical interface for reference counting). When the address of an object is assigned to a **ref_ptr<>,** the objects' reference count automatically increments. Similarly, clearing or deleting a **ref_ptr<>** decrements the object reference count.
- **Object**—The pure virtual **Object** class is the base class for any object in OSG that requires I/O support, cloning, and reference counting. All node classes and several other objects in OSG are derived from **Object.**
- **Notify**—The osg library supplies a set of functions for controlling debug, warning, and error output. You control the amount of output by specifying one of the **NotifySeverity** enumerant values. Most code modules within OSG display notification messages.

The osg library contains several other classes that this section doesn't mention. Refer to the osg library source and header files to learn about other classes and features.

The osgUtil Library

The osgUtil library is a broad collection of utilities for processing a scene graph and modifying the geometry within it.

The osgUtil library is probably best known for the set of classes that support the update, cull, and draw traversals. In a typical OSG application, these traversals are handled by higher-level support classes such as **osgViewer::Viewer,** and you do not have to interact with them directly.

Intersection

Typically, 3D applications need to support some type of user interaction or selection, such as *picking*. The osgUtil library efficiently supports picking with a large variety of classes that test the scene graph for intersection. When your application receives input from the user that requires picking, your application should use the following classes to obtain information about what part of the scene graph was selected.

The osgUtil Library

Namespace: osgUtil

Header files: <OSG_DIR>/include/osgUtil

Windows library files: osgUtil.dll and osgUtil.lib

Linux and Mac OS X library file: libosgUtil.lib

- **Intersector**—This is a pure virtual class that defines an interface for intersection testing. The osgUtil library derives several classes from **Intersector,** one for each type of geometry (line segment, plane, etc.). To perform an intersection test, your application instantiates one of the classes derived from **Intersector,** passes it to an instance of **IntersectionVisitor,** and queries it for intersection results.
- **IntersectionVisitor**—The **IntersectionVisitor** class searches a scene graph for nodes that intersect a specified piece of geometry. Intersection testing is done in classes derived from **Intersector.**
- **LineSegmentIntersector**—Derived from **Intersector,** the **LineSegmentIntersector** class tests for intersections between a specified line segment and a scene graph, and provides methods for the application to query the results.
- **PolytopeIntersector**—Like **LineSegmentIntersector,** the **PolytopeIntersector** class tests for intersections against a polytope defined by a list of planes. This class is especially useful for picking, in which the polytope defines a bounded area in world space around the mouse click point.
- **PlaneIntersector**—Like **LineSegmentIntersector,** the **PlaneIntersector** class tests for intersections against a plane that is bounded by a list of planes.

Optimization

The scene graph data structure is ideally suited for optimization and easy statistics gathering. The osgUtil library contains classes that traverse the scene graph to modify it for optimal rendering and gather statistical data about its contents.

- **Optimizer**—As its name implies, the **Optimizer** class optimizes the scene graph. Its behavior is controlled by a set of enumerant flags, which each indicate a specific type of optimization to be performed. For example, the `FLATTEN_STATIC_TRANSFORMS` flag transforms geometry by non-dynamic Transform nodes, which optimizes rendering by eliminating modifications to the OpenGL model-view matrix stack.

- **Statistics** and **StatsVisitor**—To properly tune a 3D application, a developer must have as much information as possible about what is being rendered. The **StatsVisitor** class returns the amount and type of node in a scene graph, and the **Statistics** class returns the amount and type of geometry rendered.

Geometry Manipulation

Many 3D applications require modification of loaded geometry to achieve desired performance or rendering results. The osgUtil library contains classes to provide several types of common geometrical operations.

- **Simplifier**—Use the **Simplifier** class to reduce the amount of geometry in a **Geometry** object. This can aid in the automatic construction of lower levels of detail.
- **Tessellator**—OpenGL doesn't directly support concave or complex polygons. The **Tessellator** class generates an **osg::PrimitiveSet** from a list of vertices describing such a polygon.
- **DelaunayTriangulator**—As its name implies, this class implements the Delaunay triangulation algorithm, which generates a set of triangles from a collection of vertices.
- **TriStripVisitor**—In general, strip primitives render more efficiently than individual primitives due to vertex sharing. The **TriStripVisitor** class traverses a scene graph and converts polygonal primitives to triangle and quadrilateral strips.
- **SmoothingVisitor**—The **SmoothingVisitor** class generates per vertex normals that are the average of the normals for all facets sharing that vertex.
- Texture map generation—The osgUtil library contains support routines to aid in creating reflection maps, half-way vector maps, and specular highlight maps. There is also a **TangentSpaceGenerator** class that creates arrays of per-vertex vectors to aid in bump mapping.

The osgUtil library contains several other classes that aren't mentioned in this section. Refer to the osgUtil library source and header files to learn about other classes and features.

The osgDB Library

The osgDB library allows applications to load, use, and write 3D databases. It provides support for a wide variety of common 2D image and 3D model file formats using a plugin architecture. The osgDB maintains a registry of and oversees access to the loaded OSG plugins.

OSG supports its own file formats. .osg is a plain ASCII text description of a scene graph, and .osga is an archive (or group) of .osg files. The osgDB library contains support code for these file formats. (OSG also supports a binary .ive format.)

The osgDB Library

Namespace: osgDB

Header files: <OSG_DIR>/include/osgDB

Windows library files: osgDB.dll and osgDB.lib

Linux and Mac OS X library file: libosgDB.lib

Large 3D terrain databases are often created in sections that tile together. In this case, applications require that portions of the database load from file in a background thread without interrupting rendering. The **osgDB::DatabasePager** provides this functionality.

The osgViewer Library

The osgViewer library defines several *viewer* classes that integrate OSG with a wide range of windowing toolkits, including AGL/CGL, FLTK, Fox, MFC, Qt, SDL, Win32, WxWindows, and X11. The viewer classes support single window / single view applications, as well as multithreaded applications using multiple views and render surfaces. All viewer classes support camera manipulation, event handling , and support for the **osgDB::DatabasePager.** The osgViewer library contains three viewer classes that your application can use.

- **SimpleViewer**—The **SimpleViewer** class manages a single view into a single scene graph. To use **SimpleViewer,** your application must create a window and set a current graphics context.
- **Viewer**—The **Viewer** class can manage multiple synchronized cameras to render a single view spanning multiple monitors. Viewer creates its own window(s) and graphics context(s) based on the underlying graphics system capabilities, so a single Viewer-based application executable runs on single or multiple display systems.
- **CompositeViewer**—The **CompositeViewer** class supports multiple views into the same scene and multiple cameras with different scenes. You can feed the results of one rendering into another by specifying the render order of each view. Use **CompositeViewer** to create HUDs, prerender textures, and display multiple views in a single window

The osgViewer Library

Namespace: osgViewer

Header files: <OSG_DIR>/include/osgViewer

Windows library files: osgViewer.dll and osgViewer.lib

Linux and Mac OS X library file: libosgViewer.lib

The osgViewer library contains additional support classes for statistics display, window abstraction, and scene handling.

NodeKits

NodeKits extend the concept of **Node**s, **Drawable**s, and **StateAttribute**s, and can be thought of as extensions to the osg library in core OSG. NodeKits must do more than derive from an OSG class. They must also provide a *dot OSG wrapper* (an OSG plugin to support reading from and writing to an .osg file). As a result, a NodeKit is composed of two libraries the NodeKit itself, and a dot OSG wrapper plugin library.

OSG v1.3 has six NodeKits.

- The osgFX library—This NodeKit provides additional scene graph nodes for rendering special effects, such as anisotropic lighting, bump mapping, and cartoon shading.
- The osgParticle library—This NodeKit provides particle-based rendering effects, such as explosions, fire, and smoke.
- The osgSim library—This NodeKit supports the special rendering requirements of simulation systems and OpenFlight databases, such as terrain elevation query classes, light point nodes, and DOF transformation nodes.
- The osgText library—This NodeKit is a powerful tool for adding text to your scene. It fully supports all TrueType fonts.
- The osgTerrain library—This NodeKit provides support for rendering height field data.
- The osgShadow library—This NodeKit provides a framework for supporting shadow rendering techniques.

It is beyond the scope of this book to cover in detail the extensive capabilities of the OSG NodeKits. Section **2.6 NodeKits and osgText,** explains the basics of using the osgText NodeKit, however. What you learn about using osgText in Chapter 2 should empower you to explore and use the other NodeKits.

OSG Plugins

The core OSG libraries support file I/O for a large variety of 2D image and 3D model file formats. The **osgDB::Registry** automatically manages the plugin libraries. Your application simply makes a function call to read or write a file, and as long as an appropriate plugin is available, the **Registry** finds and uses it.

The osg library allows your application to build scene graphs directly in a node-by-node fashion. In contrast, OSG plugins allow your application to load entire scene graphs from disk with just a few lines of code, or load scene graph parts that your application can arrange into a complete scene graph.

OSG v1.3 supports a selection of common 2D image file formats, including .bmp, .dds, .gif, .jpeg, .pic, .png, .rgb, .tga, and .tiff. OSG supplies a QuickTime plugin for loading movie files, and a plugin for loading font files using the FreeType library.

OSG's support for 3D model file formats is comprehensive and includes the following common file formats: 3D Studio Max (.3ds), Alias Wavefront (.obj), Carbon Graphics' Geo (.geo), Collada (.dae), ESRI Shapefile (.shp), OpenFlight (.flt), Quake (.md2), and Terrex TerraPage (.txp).

In addition to the standard formats listed above, OSG defines its own file formats. The .osg format is an ASCII text representation of a scene graph that you can edit in a text editor. The .ive format, on the other hand, is a binary format, which is optimized for fast loading.

In addition to 2D image and 3D model files, OSG plugins support I/O on archives, or collections of related files. There are plugins for the common .tgz and .zip formats, as well as OSG's own .osga archive format.

OSG plugins also allow file loading over the Internet using the .net plugin.

Finally, there is a collection of plugins known as *pseudoloaders*, which provide additional functionality beyond simply loading a file.

- scale, rot, and trans—These pseudoloaders load a file, but additionally place a **Transform** node above the loaded scene graph root node, and configure the **Transform** according to specified scale, rotation, or translation values.
- logo—The logo pseudoloader allows image files to display HUD-style over the loaded 3D scene.

Section **2.5 File I/O** provides additional information about how to use the OSG plugins in your application.

Interoperability

You can use OSG in any programming environment that allows linking to C++ libraries. To ensure OSG operates in other environments, OSG provides an interface that allows language-independent runtime access.

The osgIntrospection library allows software to interact with OSG using the reflective and introspective programming paradigms. Applications or other software use osgIntrospection classes and methods to iterate over OSG types, enumerants, and methods, and can call those methods without any compile- or link-time knowledge of OSG.

Languages such as Smalltalk and Objective-C contain built-in support for reflection and introspection, but these features are normally unavailable to C++ developers, because C++ doesn't retain the necessary metadata. To compensate for this C++ shortcoming, OSG provides a set of automatically generated wrapper libraries created from OSG source code. Your application doesn't need to interact with OSG wrappers directly; they are managed entirely by osgIntrospection.

As a result of osgIntrospection and its wrappers, many languages can now interface with OSG, including Java, TCL, Lua, and Python. For more information on language interoperability, visit the OSG Wiki Web site [OSGWiki], Community page, and select LanguageWrappers.

Applications and Examples

The OSG distribution includes five handy OSG utility applications that are useful for debugging and general OSG development activities.

- osgarchive—This application allows you to add files to an OSG .osga archive file. It also lets you extract files and list archive contents.
- osgconv—This application converts from one file format to another. It's particularly useful for converting any file format to the optimized .ive file format.
- osgdem—This application is a tool for converting elevation and image data into paged terrain databases.
- osgversion—This application dumps the current OSG version to **std::cout,** as well as some additional support code to track changes and contributors to the OSG source.
- osgviewer—This is OSG's flexible and powerful model viewer. Section **1.3 Running osgviewer** demonstrates osgviewer use in detail.

The OSG distribution also comes with example programs that demonstrate the capabilities of the API. The example source code illustrates many programming concepts and techniques used in OSG application development.

2 Building a Scene Graph

This chapter shows you how to write code that builds an OSG scene graph. It covers both bottom-up nuts and bolts scene graph construction, as well as OSG's facility for loading entire scene graphs from 3D model files.

The first topic covered is memory management. Scene graphs and their data typically consume large amounts of memory, and this section discusses OSG's system for managing that memory so that you avoid dangling pointers and memory leaks.

The simplest scene graph consists of a single leaf node with some geometry and state. Section **2.2 Geodes and Geometry** describes geometry, normal, and color specification. Following that, you'll learn how to control the appearance of geometry by specifying OSG state attributes and modes.

Real applications require more complex scene graphs than a single node, however. This chapter also covers OSG's family of group nodes, which provide the broad feature set found in most scene graph libraries.

Most applications need to read geometric data from 3D model files. This chapter describes OSG's simple file loading interface, which provides support for many common 3D file formats.

Finally, the chapter concludes by showing you how to add text to your application. OSG encapsulates a large amount of advanced functionality in NodeKits. This section looks at one NodeKit, osgText, in detail.

2.1 Memory Management

Before you start building a scene graph, you need to understand how OSG manages memory occupied by scene graph nodes and data. A firm grasp of this concept will make it easy for you to write clean code that avoids memory leaks and dangling pointers.

The previous chapter shows diagrams of some fairly simple scene graphs, headed by a root node. In a fairly typical usage scenario, the application keeps a pointer to the root node,

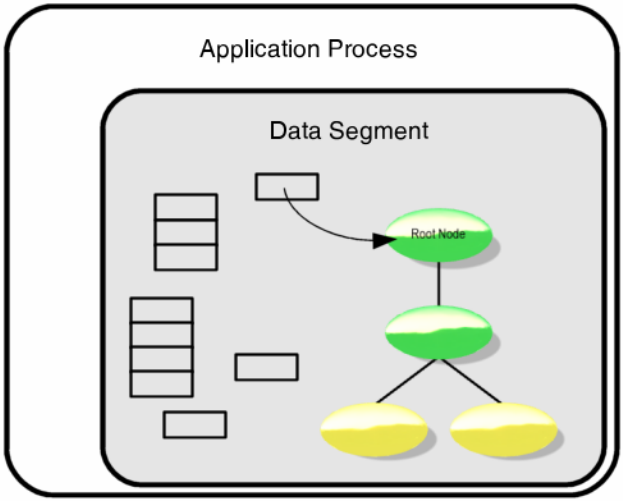


Figure 2-1
Referencing a scene graph
Typically, an application references a scene graph with a single pointer storing the address of the root node. The application doesn't keep pointers to other nodes in the scene graph. All other nodes are referenced, directly or indirectly, from the root node.

but not to any other nodes in the scene graph. The root node, directly or indirectly, references all nodes in the scene graph. Figure 2-1 illustrates this typical scenario.

When your application is done using this scene graph, the memory occupied by each node must be deleted to avoid memory leaks. Writing code to traverse the entire scene graph, deleting each node and its data along the way, would be tedious and error prone.

Fortunately, OSG provides an automated garbage collection system that uses reference-counted memory. All OSG scene graph nodes are reference counted, and when their reference count is decremented to zero, the object deletes itself. As a result, to delete the

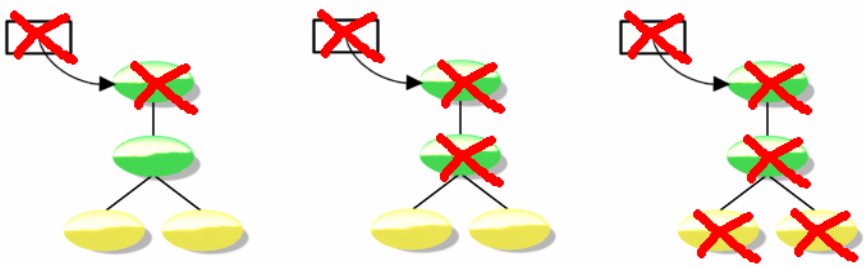


Figure 2-2
Cascading scene graph deletion
OSG's memory management system deletes the entire scene graph when the last pointer to the root node is deleted.

Warning

Never use a regular C++ pointer variable for long-term storage of pointers to objects derived from **Referenced.** As an exception, you can use a regular C++ pointer variable temporarily, as long as the pointer is eventually stored in a **ref_ptr<>.** Using a **ref_ptr<>** is always the safest approach, however.

scene graph illustrated in Figure 2-1, your application simply deletes the pointer to the root node. This causes a cascading effect that deletes all the nodes and data in the scene graph, as shown in Figure 2-2.

There are two components to this garbage collection system:

- OSG node and scene graph data classes all derive from a common base, **osg::Referenced,** which contains an integer reference count.
- OSG defines a smart pointer template class called **osg::ref_ptr<>.** When code assigns a **Referenced** object pointer to a variable of type **ref_ptr<>,** the reference count in **Referenced** is automatically incremented.

Any code that stores a pointer to an object derived from **Referenced** must store that pointer in a **ref_ptr<>** rather than a regular C++ pointer variable. If all code adheres to this rule, then memory automatically deletes itself when the last **ref_ptr<>** referencing it goes away.

When you create any scene graph node or data that derives from **Referenced,** your application code can't explicitly delete that memory. With very few exceptions, all **Referenced** subclasses have protected destructors. This ensures that objects derived from **Referenced** can only be deleted as a result of decrementing their reference count to zero.

The following text describes both **Referenced** and **ref_ptr<>** in detail, and then shows some example code.

2.1.1 The Referenced Class

Referenced (namespace: **osg**) implements a reference counted block of memory. All OSG nodes and scene graph data, including state information and arrays of vertices, normals, and texture coordinates, derive from **Referenced**. As a result, all OSG scene graph memory is reference counted.

There are three main components to the **Referenced** class:

- It contains a protected integer reference count member variable, *_refCount,* initialized to 0 in the class constructor.
- It contains public methods **ref()** and **unref(),** which increment and decrement *_refCount*. **unref()** deletes the memory used by the object if *_refCount* reaches zero.

- The class destructor is protected and virtual. Creation on the stack and explicit deletion aren't possible because the destructor is protected, and the virtual declaration allows subclass destructors to execute.

As a general rule, your code should never need to call the **ref()** and **unref()** methods directly. Instead, allow **ref_ptr<>** to handle this.

2.1.2 The ref_ptr<> Template Class

ref_ptr<> (namespace: **osg**) implements a smart pointer to an object of type **Referenced** and manages its reference count. The **Referenced** object is guaranteed to be deleted when the last **ref_ptr<>** referencing it goes away. **ref_ptr<>** eases deletion of scene graph memory and ensures object deletion during an exception call stack unwind.

There are four main components to the **ref_ptr<>** template class:

- It contains a private pointer, **_ptr,** to store the address of the managed memory. The **get()** method returns the value of **_ptr.**
- It contains several methods that allow your code to use **ref_ptr<>** as a normal C++ pointer, such as **operator->()** and **operator=().**
- The **valid()** method returns true if the **ref_ptr<>** is non-NULL.

When code assigns an address to a **ref_ptr<>** variable, the **ref_ptr<>** assignment operator, **operator=(),** assumes the address points to an object derived from **Referenced,** and automatically increments the reference count by calling **Referenced::ref().**

There are two situations when the **ref_ptr<>** variable needs to decrement the reference count: during **ref_ptr<>** deletion (in the class destructor), and during reassignment (in **operator=()**). In both of these cases, **ref_ptr<>** decrements the reference count by calling **Referenced::unref().**

2.1.3 Memory Management Examples

The following examples make use of OSG's **osg::Geode** and **osg::Group** classes. **Geode** is the OSG leaf node, which contains geometry for rendering; see **2.2 Geodes and Geometry** for more information. **Group** is a node with multiple children; see **2.3 Group Nodes** for more information. Both of these classes derive from **Referenced.**

In a typical usage scenario, you create a node and add it as a child to another node in a scene graph:

```
#include <Geode>
#include <Group>
#include <ref_ptr>
...
{
    // Create a new osg::Geode object. The assignment to the
```

```
    //   ref_ptr<> increments the reference count to 1.
    osg::ref_ptr<Geode> geode = new osg::Geode;

    // Assume 'grp' is a pointer to an osg::Group node. Group
    //   uses a ref_ptr<> to point to its children, so addChild()
    //   again increments the reference count to 2.
    grp->addChild( geode.get() );
} // The 'geode' ref_ptr<> variable goes out of scope here. This
  //   decrements the reference count back to 1.
```

A **ref_ptr<>** really isn't required in this case, because your code doesn't keep the *geode* pointer long-term. In fact, in the simple case shown above, a **ref_ptr<>** just adds unnecessary overhead to the creation process. A simple C++ pointer suffices here, because the **osg::Group** parent node's internal **ref_ptr<>** manages the memory occupied by the new **osg::Geode:**

```
// Create a new osg::Geode object. Don't increment its reference
//   count.
osg::Geode* geode = new osg::Geode;

// The internal ref_ptr<> in Group increments the child Geode
//   reference count to 1.
grp->addChild( geode );
```

Use caution when using regular C++ pointers for **Referenced** objects. In order for OSG's memory management system to work correctly, the address of the **Referenced** object must be assigned to a **ref_ptr<>** variable. In the code above, that assignment happens in the **osg::Group::addChild()** method. If the **Referenced** object is never assigned to a **ref_ptr<>** variable, its memory leaks:

```
{
    osg::Geode* geode = new osg::Geode;
} // Don't do this! Memory leak!
```

As stated previously, you can't explicitly delete an object derived from **Referenced** or create one on the stack. The following code generates compile errors:

```
osg::Geode* geode1 = new osg::Geode;
delete geode1; // Compile error: destructor is protected.

{
    osg::Geode geode2;
} // Compile error: destructor is protected.
```

Variables of type **ref_ptr<>** can point only to objects derived from **Referenced** (or objects that support the same interface as **Referenced**):

```
// OK, because Geode derives from Referenced:
osg::ref_ptr<Geode> geode = new osg::Geode;
```

```
int i;
osg::ref_ptr<int> rpi = &i; // NOT okay! 'int' isn't derived from
    //   Referenced, doesn't support the Referenced interface.
```

As discussed earlier in this chapter, OSG's memory management feature facilitates cascading deletion of entire scene graph trees. When the sole **ref_ptr<>** to the root node is deleted, the root node reference count drops to zero, and the root node destructor deletes both the root node and the **ref_ptr<>** pointers to its children. The following code doesn't leak the child **Geode** memory:

```
{
    // 'top' increments the Group count to 1.
    osg::ref_ptr<Group> top = new osg::Group;
    // addChild() increments the Geode count to 1.
    top->addChild( new osg::Geode );
} // The 'top' ref_ptr goes out of scope, deleting both the Group
  //   and Geode memory.
```

In summary:

- Assigning an object derived from **Referenced** to a **ref_ptr<>** variable automatically calls **Referenced::ref()** to increment the reference count.
- If a **ref_ptr<>** variable is made to point to something else, or is deleted, it calls **Referenced::unref()** to decrement the reference count. If the count reaches zero, **unref()** deletes the memory occupied by the object.
- When allocating an object of type **Referenced,** always ensure it is assigned to a **ref_ptr<>** to allow OSG's memory management to operate correctly.

This may seem a little long-winded for a book with "Quick Start Guide" in its title. The concept is important, however, and a firm grasp of OSG memory management is important for any OSG developer.

The sections that follow describe several classes derived from **Referenced,** and the code snippets make extensive use of **ref_ptr<>** variables. As you read this chapter, keep in mind that OSG uses **ref_ptr<>** internally for any long-term pointer storage, as in the calls to **osg::Group::addChild()** in the previous examples.

2.2 Geodes and Geometry

The previous section introduced the concept of OSG memory management. If you're new to reference counted memory, you should look at a real OSG example to increase your understanding. This section presents a simple OSG example program that uses the memory management techniques described previously, and also introduces you to building a scene graph with OSG's geometry-related classes. The code might appear overwhelming

at first glance, because you're not familiar with many of the classes. A full explanation of the geometry classes follows the code listing.

Listing 2-1
Building a simple scene graph

This is a listing of the Simple example from the book's accompanying example code. The function createSceneGraph() specifies the geometry for a single quadrilateral primitive. The quad has a different color at each vertex, but has the same normal for the entire primitive.

```
#include <osg/Geode>
#include <osg/Geometry>

osg::ref_ptr<osg::Node>
createSceneGraph()
{
    // Create an object to store geometry in.
    osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;

    // Create an array of four vertices.
    osg::ref_ptr<osg::Vec3Array> v = new osg::Vec3Array;
    geom->setVertexArray( v.get() );
    v->push_back( osg::Vec3( -1.f, 0.f, -1.f ) );
    v->push_back( osg::Vec3( 1.f, 0.f, -1.f ) );
    v->push_back( osg::Vec3( 1.f, 0.f, 1.f ) );
    v->push_back( osg::Vec3( -1.f, 0.f, 1.f ) );

    // Create an array of four colors.
    osg::ref_ptr<osg::Vec4Array> c = new osg::Vec4Array;
    geom->setColorArray( c.get() );
    geom->setColorBinding( osg::Geometry::BIND_PER_VERTEX );
    c->push_back( osg::Vec4( 1.f, 0.f, 0.f, 1.f ) );
    c->push_back( osg::Vec4( 0.f, 1.f, 0.f, 1.f ) );
    c->push_back( osg::Vec4( 0.f, 0.f, 1.f, 1.f ) );
    c->push_back( osg::Vec4( 1.f, 1.f, 1.f, 1.f ) );

    // Create an array for the single normal.
    osg::ref_ptr<osg::Vec3Array> n = new osg::Vec3Array;
    geom->setNormalArray( n.get() );
    geom->setNormalBinding( osg::Geometry::BIND_OVERALL );
    n->push_back( osg::Vec3( 0.f, -1.f, 0.f ) );

    // Draw a four-vertex quad from the stored data.
    geom->addPrimitiveSet(
        new osg::DrawArrays( osg::PrimitiveSet::QUADS, 0, 4 ) );

    // Add the Geometry (Drawable) to a Geode and
    //   return the Geode.
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( geom.get() );
```

```
    return geode.get();
}
```

Listing 2-1 makes extensive use of the **ref_ptr<>** template class described in the previous section. All the memory allocated in Listing 2-1 is reference counted. The createSceneGraph() function even returns a **ref_ptr<>** to the created scene graph. (Strictly speaking, the code in Listing 2-1 could be written completely with regular C++ pointers, as long as the calling code stores the return address in a **ref_ptr<>.** However, it's good practice to use **ref_ptr<>** in your application, because it automates memory deletion in the event of an exception or shortcut return. This book and its example code use **ref_ptr<>** throughout to encourage this good practice.)

The code in Listing 2-1 creates a scene graph with a single node. Figure 2-3 shows a diagram of this extremely simple scene graph. Real scene graphs are much more complex, though this single-node scene graph has educational value for new developers.

Figure 2-3
The Listing 2-1 scene graph
Listing 2-1 creates a scene graph consisting of a single Geode.

In addition to creating a scene graph, as shown in Listing 2-1, you'll also want to render it to create an image or animation. The examples in this chapter use osgviewer to view the scene graph, because writing viewer code for your application isn't covered until **Chapter 3, Using OpenSceneGraph in Your Application.** To view a scene graph in osgviewer, you need to write it to disk. Listing 2-2 shows code that calls the function in Listing 2-1 and writes the scene graph to disk as an .osg file. Once the scene graph exists as a file on disk, you can use osgviewer to see what it looks like.

Listing 2-2
Writing a scene graph to disk
This listing shows the main() entry point for the Simple example program. main() calls createSceneGraph() in Listing 2-1 to create a scene graph, then writes the scene graph to disk as a file named "Simple.osg".

```
#include <osg/ref_ptr>
#include <osgDB/Registry>
#include <osgDB/WriteFile>
#include <osg/Notify>
#include <iostream>

using std::endl;

osg::ref_ptr<osg::Node> createSceneGraph();
```

```
int
main( int, char** )
{
    osg::ref_ptr<osg::Node> root = createSceneGraph();
    if (!root.valid())
        osg::notify(osg::FATAL) <<
                "Failed in createSceneGraph()." << endl;

    bool result = osgDB::writeNodeFile(
            *(root.get()), "Simple.osg" );

    if ( !result )
        osg::notify(osg::FATAL) <<
            "Failed in osgDB::writeNode()." << endl;
}
```

After calling the function in Listing 2-1 to create the scene graph, the code in Listing 2-2 writes it to disk as a file called "Simple.osg". The .osg file format is OSG's proprietary ASCII-text based file format. As an ASCII file, .osg files are slow to load and rather large, so the format is rarely used in production code. However, it's extremely useful for debugging during development and quick demos.

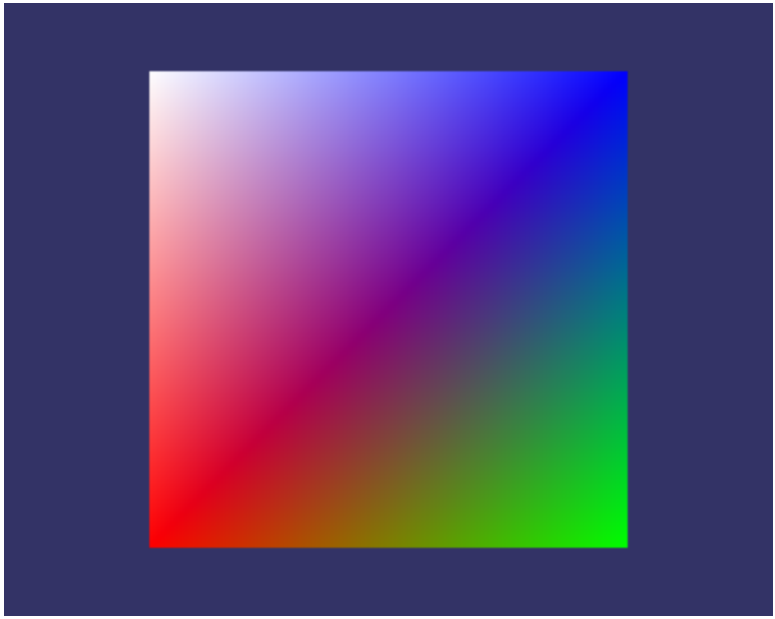


Figure 2-4
The simple example scene graph displayed in osgviewer
This figure shows the quadrilateral primitive created in Listing 2-1, after being written out as an .osg file by the code in Listing 2-2 and displayed in osgviewer.

The code in Listings 1 and 2 is from the Simple example in the accompanying source code. If you haven't already done so, obtain the example source code from the book's Web site and compile and run Simple. After running the example, you'll find the output file, Simple.osg, in your working directory. To see what this scene graph looks like, use osgviewer:

```
osgviewer Simple.osg
```

osgviewer should display an image similar to Figure 2-4. Chapter 1 describes osgviewer and its user interface. You can spin the rendered geometry with the left mouse, and zoom in or out with the right mouse, for example.

The code in Listing 2-1 makes extensive use of OSG's geometry-related classes. The following text provides a high-level explanation of how to use these classes.

2.2.1 An Overview of Geometry Classes

The code in Listing 2-1 might look confusing, but in essence it performs only three operations.

1. It creates arrays of vertex, normal, and color data.
2. It instantiates an **osg::Geometry** object and adds the arrays to it. It also adds an **osg::DrawArrays** object to specify how the data should be drawn.
3. It instantiates an **osg::Geode** scene graph node and adds the **Geometry** object to it.

This section examines each of these steps in detail.

Vector and Array Classes

OSG defines a rich set of classes for storing vector data such as vertices, normals, colors, and texture coordinates. **osg::Vec3** is an array of three floating point numbers; use it for vector and normal data. Use **osg::Vec4** for color data, and **osg::Vec2** for 2D texture coordinates. In addition to simple vector storage, these classes provide a complete set of methods for calculating length, dot and cross products, vector addition, and vector-matrix multiplication.

OSG defines a template array class for storing objects. The most common use of the array template is for storage of vector data. In fact, this use is so common, that OSG provides type definitions for arrays of vector data: **osg::Vec2Array, osg::Vec3Array,** and **osg::Vec4Array.**

Listing 2-1 creates individual three-element vectors for each *xyz* vertex using **Vec3,** then pushes each **Vec3** onto the back of a **Vec3Array**. The code uses **Vec3Array** in an almost identical fashion to store *xyz* normal data. Listing 2-1 uses **Vec4** and **Vec4Array** for color

data, because colors have four elements (red, green, blue, and alpha). Later, this chapter present example code that uses **Vec2** and **Vec2Array** to store two-element texture coordinates.

The array types derive from **std::vector,** so they support the **push_back()** method to add new elements, as shown in Listing 2-1. As a subclass of **std::vector,** the array classes also support the resize() and operator[]() methods. Here's an example that creates a vertex data array using resize() and operator[]().

```
osg::ref_ptr<osg::Vec3Array> v = new osg::Vec3Array;
geom->setVertexArray( v.get() );
v->resize( 4 );
(*v)[ 0 ] = osg::Vec3( -1.f, 0.f, -1.f );
(*v)[ 1 ] = osg::Vec3( 1.f, 0.f, -1.f );
(*v)[ 2 ] = osg::Vec3( 1.f, 0.f, 1.f );
(*v)[ 3 ] = osg::Vec3( -1.f, 0.f, 1.f );
```

Drawables

OSG defines a class, **osg::Drawable,** to store data for rendering. **Drawable** is a virtual base class that isn't instantiated directly. Core OSG derives three subclasses from **Drawable: osg::DrawPixels,** which is a wrapper around the glDrawPixels() command, **osg::ShapeDrawable,** which provides access to several predefined shapes such as cylinders and spheres, and **osg::Geometry.** The example code uses **Geometry,** which is the most flexible and commonly used subclass.

If you're already familiar with vertex arrays in OpenGL, the **Geometry** class will be easy for you to use. **Geometry** provides an interface that lets your application specify arrays of vertex data and how that data should be interpreted and rendered. This is analogous to the OpenGL entry points for specifying vertex array data (such as **glVertexPointer()** and **glNormalPointer()**) and vertex array rendering (such as **glDrawArrays()** and **glDrawElements()**). The code in Listing 2-1 uses the following **Geometry** methods:

- **setVertexArray(), setColorArray(),** and **setNormalArray()**—These methods are analogous to **glVertexPointer(), glColorPointer(),** and **glNormalPointer()** in OpenGL. Your application uses these methods to specify arrays of vertex, color, and normal data. **setVertexArray()** and **setNormalArray()** each take a pointer to a **Vec3Array** as a parameter, and **setColorArray()** takes a pointer to a **Vec4Array.**
- **setColorBinding()** and **setNormalBinding()**—These methods tell **Geometry** how to apply the color and normal data. They take an enumerant defined in the **Geometry** class as a parameter. Listing 2-1 sets the color binding to `osg::Geometry::BIND_PER_VERTEX` to assign a different color to each vertex. However, the code sets the normal binding to `osg::Geometry::BIND_OVERALL` to apply the single normal to the entire **Geometry.**

- **addPrimitiveSet()**—This method tells **Geometry** how to render its data. It takes a pointer to an **osg::PrimitiveSet** as a parameter. **PrimitiveSet** is a virtual base class that you don't instantiate directly. Your code can add multiple **PrimtiveSet**s to the same **Geometry.**

The **addPrimitiveSet()** method allows your application to specify how OSG should draw the geometric data stored in the **Geometry** object. Listing 2-1 specifies an **osg::DrawArrays** object. **DrawArrays** derives from **PrimitiveSet**. Think of it as a wrapper around the **glDrawArrays()** vertex array drawing command. The other **PrimitiveSet** subclasses (**DrawElementsUByte, DrawElementsUShort,** and **DrawElementsUInt**) emulate OpenGL's **glDrawElements()** entry point. OSG also provides the **DrawArrayLengths** class, which has no equivalent in OpenGL. Functionally, it's similar to repeated calls to **glDrawArrays()** with different index ranges and lengths.

The most commonly-used **DrawArrays** constructor has the following declaration:

```
osg::DrawArrays::DrawArrays(
    GLenum mode, GLint first, GLsizei count );
```

mode is one of the 10 OpenGL primitive types, such as `GL_POINT`, `GL_LINES`, or `GL_TRIANGLE_STRIP`. The **PrimitiveSet** base class defines equivalent enumerants, such as `osg::PrimitiveSet::POINTS`, and your code can use either.

first is the index of the first element in the vertex data array that OSG should use when rendering, and *count* is the total number of elements that OSG should use. For example, if your vertex data contains 6 vertices, and you want to render a triangle strip from those vertices, you might add the following **DrawArrays** primitive set to your **Geometry:**

```
geom->addPrimitiveSet( new osg::DrawArrays(
    osg::PrimitiveSet::TRIANGLE_STRIP, 0, 6 );
```

After adding vertex data, color data, normal data, and a **DrawArrays** primitive set to the

How OSG Draws

While the **PrimitiveSet** subclasses provide near-equivalent functionality to OpenGL's vertex array features, don't assume that **PrimitiveSet** always uses vertex arrays under the hood. Depending on rendering circumstances, OSG might use vertex arrays (with and without buffer objects), display lists, or even **glBegin()**/**glEnd()** to render your geometry.

Objects derived from **Drawable** (such as **Geometry**) use display lists by default. You can modify this behavior by calling **osg::Drawable::setUseDisplayList(false).**

OSG resorts to using **glBegin()**/**glEnd()** if you specify the `BIND_PER_PRIMITIVE` attribute binding, which sets an attribute for each individual primitive (for each triangle in a `GL_TRIANGLES`, for example.

Geometry object, the code in Listing 2-1 performs one final operation with the **Geometry:** it attaches it to a node in the scene graph. The next section describes this operation.

Geodes

The **osg::Geode** class is the OSG leaf node that stores geometry for rendering. Listing 2-1 creates the simplest scene graph possible—a scene graph consisting of a single leaf node. At the end of Listing 2-1, the createSceneGraph() function returns the address of its **Geode** as a **ref_ptr<>** to an **osg::Node.** This is legal C++ code because **Geode** derives from **Node.** (By definition, all OSG scene graph nodes derive from **Node.**)

osg::Geode has no children because it's a leaf node, but it can contain geometry. The name **Geode** is short for "geometry node"—a node that contains geometry. Any geometry that your application renders must be attached to a **Geode. Geode** provides the **addDrawable()** method to allow your application to attach geometry.

Geode::addDrawable() takes a pointer to a **Drawable** as a parameter. As described in the previous section, **Drawable** is a virtual base class with many subclasses such as **Geometry.** Look at Listing 2-1 to see an example of adding a **Geometry** object to a **Geode** using **addDrawable().** The code performs this operation at the end of the createSceneGraph() function, just before returning the **Geode.**

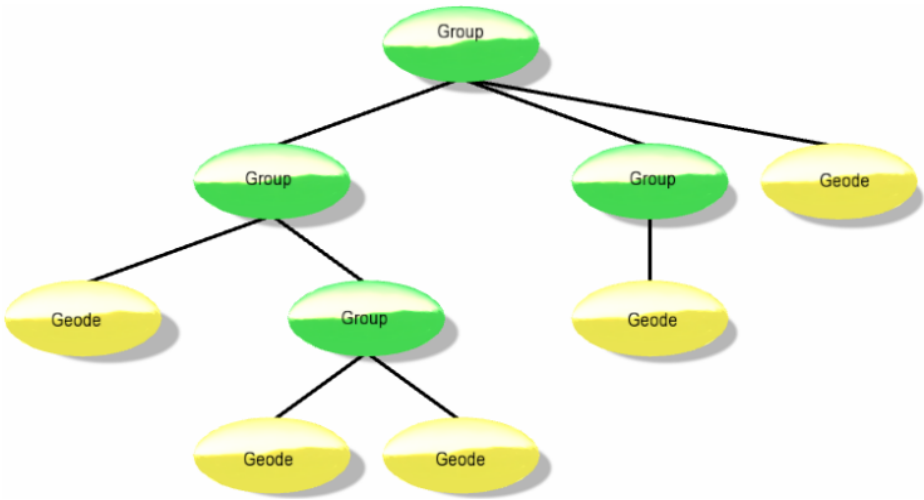


Figure 2-5
The Group node
Shown in green, the **Group** node can have multiple children, which in turn can also be **Group** nodes with their own children.

2.3 Group Nodes

OSG's group node, **osg::Group,** allows your application to add any number of child nodes, which in turn can also be **Group** nodes with their own children, as shown in Figure 2-5. **Group** is the base class for many useful nodes, including **osg::Transform, osg::LOD,** and **osg::Switch,** which are described in this section.

Group ultimately derives from **Referenced.** In a typical scenario, the only code referencing a given **Group** is its parent, so if the root node of the scene graph is deleted, a cascading deletion ensures there are no memory leaks.

Group is really the heart of OSG because it allows your application to organize data in the scene graph. **Group**'s strength comes from its interface for managing child nodes. **Group** also has an interface for managing parents, which it inherits from its base class, **osg::Node.** This section provides an overview of both the child and parent interfaces.

Following a description of the child and parent interfaces, this section describes three frequently-used classes derived from Group, the **Transform, LOD,** and **Switch** nodes.

2.3.1 The Child Interface

The **Group** class defines the interface for having children, and all nodes that derive from **Group** inherit this interface. Most OSG nodes you use derive from **Group** (with **Geode** being an exception), so in general you can assume that most nodes support the child interface.

Group stores pointers to its children in a **std::vector< ref_ptr<Node> >**—an array of **ref_ptr<>**s to **Node.** You can access a child by index because **Group** uses an array. **Group** uses **ref_ptr<>** to allow OSG's memory management system to work.

The following code snippet shows part of the declaration for **Group**'s child interface. All classes are in the **osg** namespace.

```
class Group : public Node {
public:
    ...
    // Add a child node.
    bool addChild( Node* child );

    // Remove a child node. If the node isn't a child, do nothing
    //   and return false.
    bool removeChild( Node* child );

    // Replace a child node with a new child node.
    bool replaceChild( Node* origChild, Node* newChild );

    // Return the number of children.
    unsigned int getNumChildren() const;
```

```
    // Return true if the specified node is a child node.
    bool containsNode( const Node* node ) const;
    ...
};
```

A very simple scene graph might consist of a **Group** parent node with two **Geode** children. You can construct such a scene graph with the following code.

```
{
    osg::ref_ptr<osg::Group> group = new osg::Group;

    osg::ref_ptr<osg::Geode> geode0 = new osg::Geode;
    group->addChild( new osg::Geode0 );

    osg::ref_ptr<osg::Geode> geode1 = new osg::Geode;
    group->addChild( new osg::Geode1 );
}
```

Note that **Group** uses a **ref_ptr<>** to point to its children. In this example, the memory in *geode0* and *geode1* is referenced by *group*. That memory remains allocated after *geode0* and *geode1* go out of scope, and is freed only after *group* is deleted.

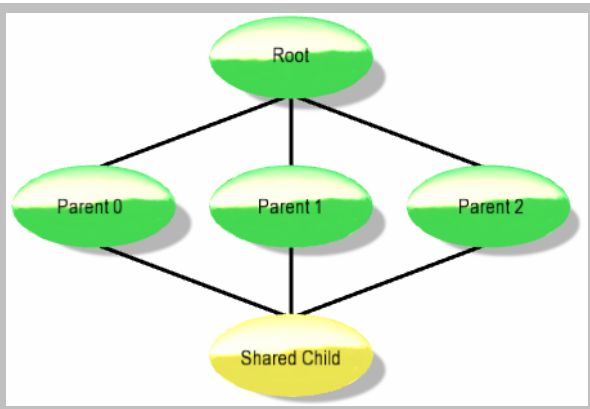
2.3.2 The Parent Interface

Group inherits an interface for managing parents from **Node. Geode** has the same interface because it also derives from **Node.** OSG allows nodes to have multiple parents. The following code snippet shows part of the declaration for **Node**'s parent interface. All classes are in the **osg** namespace.

```
class Node : public Object {
public:
    ...
    typedef std::vector<Group*> ParentList;

    // Return a list of all parents.
    const ParentList& getParents() const;

    // Return a pointer to the parent with the specified index.
    Group* getParent( unsigned int I );

    // Return the number of children.
    unsigned int getNumParents() const;
    ...
};
```

Node derives from **osg::Object. Object** is a virtual base class that your application doesn't instantiate directly. **Object** provides an interface for storing and retrieving a name string and specifying whether stored data is static or dynamically modifiable, and derives

Multiple Parents



When you add the same node as a child to multiple nodes, the child node has multiple parents, as illustrated in the diagram. You might want to do this to render the same subgraph many times without creating multiple copies of the subgraph. Section 2.4.3 Example Code for Setting State shows how to render the same **Geode** with different transformations and different rendering state, for example.

When a node has multiple parents, OSG traverses the node multiple times. Each parent keeps its own **ref_ptr<>** to the child, so the child isn't deleted until all parents have stopped referencing it.

from **Referenced** to support OSG's memory management system. Section **3.2 Dynamic Modification** discusses **Object**'s name and dynamic data interfaces in more detail.

Note that **osg::Node::ParentList** is a **std::vector** of regular C++ pointers. While a **Node** has the address of its parents, the **Node** doesn't need to use OSG's memory management system to reference its parents. When a parent is deleted, the parent removes itself from any child node parent lists.

In the typical case where a node has a single parent (**getNumParents()** returns zero), obtain a pointer to that parent with a call to **getParent(0).**

You will use the child and parent interfaces often when constructing and manipulating scene graphs. However, classes derived from Group provide additional functionality required by many applications. The following sections describe three

2.3.3 Transform Nodes

OSG supports transformation of geometric data with the **osg::Transform** family of node classes. **Transform** derives from **Group,** and can have multiple children. **Transform** is a virtual base class that your application can't instantiate directly. Instead, use **osg::MatrixTransform** or **osg::PositionAttitudeTransform,** both of which derive from **Transform.** These two classes provide different transformation interfaces. Depending on your application requirements, you may use one or the other or both.

Transform affects the OpenGL model-view matrix stack. Hierarchically-arranged **Transform** nodes create successively concatenated transformations in the same way that

Matrices and osg::Matrix

The **osg::Matrix** class stores and allows operations on a 4×4 matrix consisting of 16 floating point numbers. **Matrix** doesn't derive from **Referenced** and isn't reference counted.

Matrix provides an interface that is somewhat backwards from the notation used in the OpenGL specification and most OpenGL textbooks. Matrix exposes an interface consistent with two-dimensional row-major C/C++ arrays:

```
osg::Matrix m;
m( 0, 1 ) = 0.f; // Set the second element (row 0, column 1)
m( 1, 2 ) = 0.f; // Set the seventh element (row 1, column 2)
```

OpenGL matrices are one-dimensional arrays, which OpenGL documentation usually displays as column-major:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

```
GLfloat m;
m[1] = 0.f; // Set the second element
m[6] = 0.f; // Set the seventh element
```

In spite of this apparent difference, both OSG and OpenGL matrices are laid out in memory identically—OSG doesn't perform a costly transpose before submitting its matrix to OpenGL. As a developer, however, remember to transpose an OSG matrix in your head before accessing individual elements.

Matrix exposes a comprehensive set of operators for vector-matrix

OpenGL matrix manipulation commands (such as **glRotatef()** or **glScalef()**) concatenate matrices onto the current top of matrix stack.

Transform allows you to specify its reference frame. By default, the reference frame is relative (`osg::Transform::RELATIVE_RF`), resulting in the concatenation behavior described previously. Just as OpenGL allows you to call **glLoadMatrixf(),** however, OSG allows you to set an absolute reference frame:

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
mt->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
```

multiplication and matrix concatenation. To transform a **Vec3** *v* by a rotation *R* around a new origin *T,* use the following code:

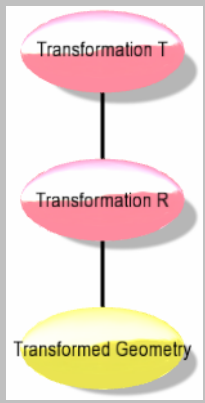
```
osg::Matrix T;
T.makeTranslate( x, y, z );
osg::Matrix R;
R.makeRotate( angle, axis );
Vec3 vPrime = v * R * T;
```

Matrix's premultiplication-style operators are the opposite of what you usually find in most OpenGL documentation:

v′ = TRv

This OpenGL notation produces the same result because OpenGL's postmultiplication notation with column-major matrices is equivalent to OSG's premultiplication-style operators with row-major matrices.

To process an entire **Geode** with this transformation, create a **MatrixTransform** node containing *T,* add a child **MatrixTransform** node containing *R*, and add a child **Geode** to the rotation **MatrixTransform,** as shown in the diagram. This is equivalent to the following sequence of OpenGL commands:




```
glMatrixMode( GL_MODELVIEW );
glTranslatef( ... ); // Translation T
glRotatef( ... ); // Rotation R
...
glVertex3f( ... );
glVertex3f( ... );
...
```

In summary, while OSG exposes a row-major interface that differs from OpenGL documentation's column-major notation, OSG and OpenGL perform equivalent operations internally and their matrices are 100% compatible.

The MatrixTransform Node

MatrixTransform uses an **osg::Matrix** object internally (see the sidebar). To create a **MatrixTransform** node that performs a translation, create a translation **Matrix** and assign the **Matrix** to the **MatrixTransform.**

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
m.setTranslate( x, y, z );
mt->setMatrix( m );
```

Matrix doesn't derive from **Referenced.** You can create local **Matrix** variables on the stack. The **MatrixTransform::setMatrix()** method copies the **Matrix** parameter onto the **MatrixTransform** node's **Matrix** member variable.

Matrix provides an interface for common transformation such as translation, rotation, and scale. You can also set the matrix explicitly:

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;

// Set all 16 values of the Matrix:
m.set( 1.f, 0.f, 0.f, 0.f,
       0.f, 1.f, 0.f, 0.f,
       0.f, 0.f, 1.f, 0.f,
       10.f, 0.f, 0.f, 1.f ); // translate by x=10
mt->setMatrix( m );
```

The State example in the book's source code, illustrated in Section **2.4.3 Example Code for Setting State,** uses multiple **MatrixTransform** nodes to render the same multiply-parented **Geode** at several different locations, each with their own unique state settings.

The PositionAttitudeTransform Node

Use the **PositionAttitdeTransform** node to specify a transformation using a **Vec3** position and a quaternion. OSG provides the **osg::Quat** class for storage of quaternion orientation data. **Quat** isn't derived from **Referenced** and therefore isn't reference counted.

Quat provides a rich configuration interface. The following code demonstrates several methods for creating and configuring quaternions.

```
// Create a quaternion rotated theta radians around axis.
Float theta( M_PI * .5f );
osg::Vec3 axis( .707f, .707f, 0.f );
osg::Quat q0( theta, axis );

// Create a quaternion using yaw/pitch/roll angles.
osg::Vec3 yawAxis( 0.f, 0.f, 1.f );
osg::Vec3 pitchAxis( 1.f, 0.f, 0.f );
osg::Vec3 rollAxis( 0.f, 1.f, 0.f );
osg::Quat q1( yawRad, yawAxis, pitchRad, pitchAxis,
        rollRad, rollAxis );

// Concatenate the two quaternions
Q0 *= q1;
```

Configure a **PositionAttitudeTransform** by using its **setPosition()** and **setAttitude()** methods

```
osg::Vec3 pos( x, y, z );
osg::Quat att( theta, axis );
osg::ref_ptr<osg::PositionAttitudeTransform> pat =
    new osg::PositionAttitudeTransform;
pat->setPosition( pos );
pat->setAttitude( att );
```

You can add as many child nodes to a **PositionAttitudeTransform** node as required for your application because **PositionAttitudeTransform** inherits the child interface from **Group**. Like **MatrixTransform,** the **PositionAttitudeTransform** node transforms child geometry by its position and attitude.

2.3.4 The LOD Node

Use the **osg::LOD** node to render objects at varying levels of detail. **LOD** derives from **Group** and therefore inherits the child interface. **LOD** also allows you to specify a range for each child. The range consists of a minimum and maximum value. These values represent distances by default, and **LOD** displays a child if its distance to the viewer falls within the child's corresponding range. **LOD** children can be in any order and don't need to be sorted by distance of amount of detail.

Figure 2-6 illustrates an **LOD** node with three children. The first child is a **Group** node with children of its own. When distance to the viewpoint falls within the first child's range, OSG traverses it and its children. The **LOD** node applies the same logic to its second and third child. OSG can display none, any, or all of an **LOD**'s children, based on their distance range.

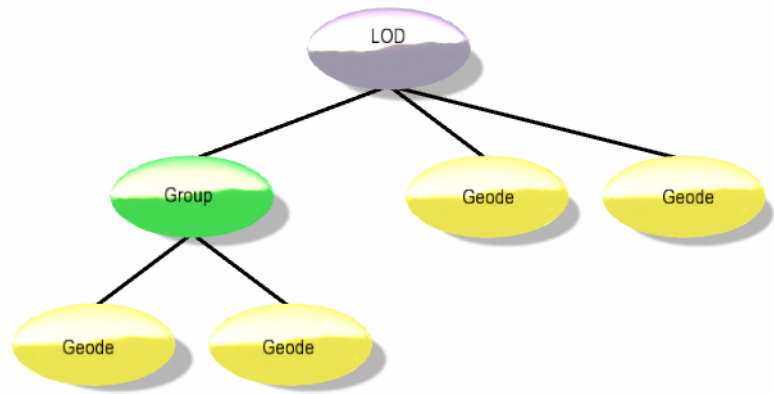


Figure 2-6
The LOD node
This figure shows an LOD node with three children. Each child has a distance range. The LOD node allows its children to be rendered if the distance to the viewpoint falls within a child's range.

The following code adds a **Geode** child with a range of 0 to 1000. The parent **LOD** node displays it when the distance to the eye is less than 1000 units.

```
osg::ref_ptr<osg::Geode> geode;
...
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// Display geode when 0.f <= distance < 1000.f
lod->addChild( geode.get(), 0.f, 1000.f );
```

LOD displays multiple children simultaneously if their ranges overlap.

```
osg::ref_ptr<osg::Geode> geode0, geode1;
...
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// Display geode0 when 0.f <= distance < 1050.f
lod->addChild( geode0.get(), 0.f, 1050.f );
// Display geode1 when 950.f <= distance < 2000.f
lod->addChild( geode1.get(), 950.f, 2000.f );
// Result: display geode0 and geode1 when 950.f <= distance <
1050.f
```

LOD computes the distance from the eye to the center of its bounding volume by default. If this isn't appropriate for your rendering situation, you can specify a user-defined center. The following code configures and LOD node to use a user-defined center.

```
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// Use a user-defined center for distance computation
lod->setCenterMode( osg::LOD::USER_DEFINED_CENTER );
// Specify the user-defined center x=10 y=100
Lod->setCenter( osg::Vec3( 10.f, 100.f, 0.f ) );
```

To restore the distance computation to the default behavior of using the bounding sphere center, call **osg::LOD::setCenterMode(osg::LOD::USE_BOUNDING_SPHERE_CENTER).**

The minimum and maximum values represent distance by default. However, you can configure **LOD** so that it interprets the range values as pixel sizes, and **LOD** displays a child if its on-screen pixel size falls within the child's corresponding range. To configure an **LOD** node's range mode, call **osg::LOD::setRangeMode()** and pass in either **osg::LOD::DISTANCE_FROM_EYE_POINT** or **osg::LOD::PIXEL_SIZE_ON_SCREEN.**

2.3.5 The Switch Node

Use the **osg::Switch** node to selectively render or skip over specific child nodes. Typical uses of **Switch** include implementing your own load-balancing scheme to selectively draw certain children based on current rendering load, or to switch between screens or levels in

a game. **Switch** has been used in the Present3D application to display successive slides in a presentation.

Like **LOD, Switch** inherits the child interface from **Group.** Each **Switch** child node has an associated Boolean value. **Switch** renders a child when the child's corresponding Boolean value is true and skips over the child when the value is false. The following code creates a **Switch** node and adds two Group children, one visible and one not visible.

```
osg::ref_ptr<osg::Group> group0, group1;
...
// Create a Switch parent node and add two Group children:
osg::ref_ptr<osg::Switch> switch = new osg::Switch;
// Render the first child:
switch->addChild( group0.get(), true );
// Don't render the second child:
switch->addChild( group1.get(), false );
```

If your code simply adds a child without specifying a value, **Switch** assigns the default value *true*.

```
// Add a child. By default, it's visible
switch->addChild( group0.get() );
```

You can change the default value for new children by calling **Switch::setNewChildDefaultValue().**

```
// Change the default for new children:
switch->setNewChildDefaultValue( false );
// These new children will be turned off by default:
switch->addChild( group0.get() );
switch->addChild( group1.get() );
```

Once you've added a child to a **Switch,** you can change its value. Use **Switch::setChildValue()** and pass in the child and its new value.

```
// Add a child, initially turned on:
switch->addChild( group0.get(), true );
// Disable grouo0:
switch->setChildValue( group0.get(), false );
```

The above code snippet is extremely contrived. To experience the full potential of enabling and disabling **Switch** children at runtime, you need to use an update callback (**osg::NodeCallback**) or *NodeVisitor* (**osg::NodeVisitor**) to manipulate your scene graph node values between rendered frames. This book discusses update callbacks and the **NodeVisitor** class in **Chapter 3, Using OpenSceneGraph in Your Application.**

2.4 Rendering State

The Simple example code in Section **2.2 Geodes and Geometry** creates a scene graph for you to examine using the osgviewer application. If you rotated the quad in osgviewer, you almost certainly noticed that the quad was lit from a light source at the viewpoint. osgviewer enables lighting by configuring OSG *rendering state*.

Lighting is one of many rendering state features supported by OSG. OSG supports most of the OpenGL fixed function pipeline rendering state (such as alpha function, blending, clip planes, color mask, face culling, depth and stencil state, fog, point and line rasterization state, and several others). OSG rendering state also allows applications to specify vertex and fragment shaders.

Your application sets rendering state in an **osg::StateSet.** You can attach a **StateSet** to any node in the scene graph, and you can also attach it to a **Drawable.** Most OpenGL application developers know that they need to minimize state changes and avoid setting state redundantly; the **StateSet** object handles these optimizations automatically.

StateSets also manage the OpenGL state attribute stack as OSG traverses the scene graph. This allows your application to set different state in different scene graph subtrees. OSG effectively saves and restores rendering state as it traverses each subtree.

You should minimize the number of **StateSet**s you attach to a scene graph. Fewer **StateSet**s consume less memory and reduces the amount of work OSG has to do during scene graph traversals. To facilitate **StateSet** sharing, **StateSet** derives from **Referenced.** This means that **Node**s or **Drawable**s that share the same **StateSet** don't require extra code to manage memory cleanup.

OSG organizes rendering state into two groups, attributes and modes. Attributes are the state variables that control the rendering features. For example, fog color and blend function are both OSG state attributes. Modes have a nearly 1-to-1 mapping with the OpenGL state features toggled with **glEnable()** and **glDisable().** Your application sets modes to enable or disable functionality such as texture mapping and lighting. In short, modes are rendering features, and attributes are the variables that control those features.

To set a state value, your application needs to do the following:

- Obtain the **StateSet** for the **Node** or **Drawable** whose state you want to set.
- Call into that **StateSet** to set the state modes and attributes.

To obtain a **StateSet** from a **Node** or **Drawable,** call the following method:

```
osg::StateSet* state = obj->getOrCreateStateSet();
```

In the above code snippet, *obj* is either a **Node** or a **Drawable; getOrCreateStateSet()** is defined for both classes. This method returns a pointer to *obj*'s **StateSet.** If *obj* doesn't already have a **StateSet** attached to it, this method creates a new one and attaches it.

StateSet derives from **Referenced.** The owning **Node** or **Drawable** uses a **ref_ptr<>** internally to reference the **StateSet,** so it's safe for your application to use a regular C++ pointer as long as you don't need to reference the **StateSet** long-term. The above code demonstrates the typical usage, in which *state* is a local variable within a function call, and the application doesn't need a long-term reference to the **StateSet.**

The *state* variable in the above code snippet is a pointer to *obj*'s **StateSet.** Once your application has obtained a pointer to a **StateSet,** you can set attributes and modes. The following sections look at both in detail, and also present a simple example.

2.4.1 Attributes and Modes

OSG defines a different class for each state attribute that your application can set. All state attribute classes derive from **osg::StateAttribute,** which is a virtual base class that your application won't instantiate directly.

The classes that derive from **StateAttribute** number in the dozens. This book provides a glimpse of a few attribute classes, and examines the lighting and texture mapping attributes in greater detail. A comprehensive examination of all attribute classes is outside the scope of this book. To explore on your own, examine the header files in include/osg in your OSG development environment, and look for classes that derive from **StateAttribute.**

OSG divides attributes and modes into two groups: texture and non-texture. This section discusses setting non-texture state. Setting a texture state is covered in Section **2.4.4 Texture Mapping.** OSG provides a different interface to set texture attributes because texture attributes require a texture unit for multitexturing.

Setting an Attribute

To set an attribute, instantiate the class corresponding to the attribute you want to modify. Set values in that class, then attach it to the **StateSet** using **osg::StateSet::setAttribute().** The following code snippet demonstrates how to set the face culling attribute:

```
// Obtain the StateSet from the geom.
osg::StateSet* state = geom->getOrCreateStateSet();

// Create and add the CullFace attribute.
osg::CullFace* cf = new osg::CullFace(
    osg::CullFace::BACK );
state->setAttribute( cf );
```

In the above code snippet, *geom* is a **Geometry** object (though it could be any other object derived from **Drawable** or **Node**). After obtaining a pointer to *geom*'s **StateSet,** the code creates a new **osg::CullFace** object, and then attaches it to *state.*

CullFace is an attribute, and therefore derives from **StateAttribute.** Its constructor takes a single parameter: an enumerant that specifies whether front or back faces are to be culled: `FRONT, BACK,` or `FRONT_AND_BACK.` These enumerants are defined in the

CullFace header, and are equivalent to OpenGL's `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK` enumerants.

If you're familiar with OpenGL, think of the above code as calling **glCullFace(** `GL_BACK` **).** Keep in mind that OSG is a scene graph, however. When your application attaches a **CullFace** attribute to a **StateSet,** you're storing a desired state, not issuing a command to OpenGL. During the draw traversal, OSG tracks changes to state and only issues the **glCullFace()** command when it's required.

Like most objects in OSG, **StateAttribute** derives from **Referenced.** After you instantiate any object that derives from **StateAttribute** and attach it to a **StateSet,** the **StateSet** references the attribute. You don't have to worry about deleting that memory later. In the typical use case shown above, you reference the **StateAttribute** temporarily using a regular C++ pointer. After you've attached the **StateAttribute** to a **StateSet,** the owning **StateSet**'s **ref_ptr<>** manages the memory.

Setting a Mode

To enable or disable a mode, call **osg::StateSet::setMode().** For example, the following code snippet enables fog:

```
// Obtain the StateSet.
osg::StateSet* state = geom->getOrCreateStateSet();

// Enable fog in this StateSet.
state->setMode( GL_FOG, osg::StateAttribute::ON );
```

setMode()'s first parameter is any **GLenum** that is valid in a call to **glEnable()** or **glDisable().** The second parameter can be `osg::StateAttribute::ON` or `osg::StateAttribute::OFF.` (Actually, this is a bit mask, as discussed in **2.4.2 State Inheritance**.)

Setting an Attribute and a Mode

OSG provides a convenient interface to set both an attribute and a mode with a single function call. In many cases, an obvious correspondence exists between attributes and modes. The **CullFace** attribute's corresponding mode is `GL_CULL_FACE`, for example. To attach an attribute to a **StateSet** and simultaneously enable its corresponding mode, use the **osg::StateSet::setAttributeAndModes()** method. The following code snippet attaches the blending function and enables blending:

```
// Create the BlendFunc attribute.
osg::BlendFunc* bf = new osg::BlendFunc();
// Attach the BlendFunc attribute and enable blending.
state->setAttributeAndMode( bf );
```

setAttributeAndModes() has a second parameter that enables or disables the first parameter attribute's corresponding mode. The second parameter defaults to `ON.` This

allows your application to specify an attribute and conveniently enable its corresponding mode with a single function call.

2.4.2 State Inheritance

When you set state on a node, that state applies to the current node and its children. If a child node sets the same state to a different value, however, the child state value overrides the parent state. In other words, the default behavior is to inherit state from parent nodes unless a child node changes it. This concept is illustrated in Figure 2-7.

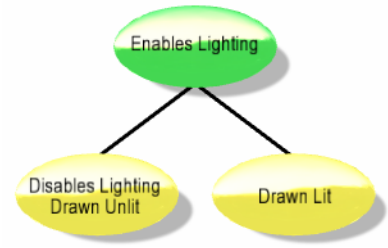


Figure 2-7
State inheritance
In this scene graph, the root node enables lighting. Its first child disables lighting, which overrides the lighting state from its parent. OSG renders the first child with lighting disabled. The second child doesn't change state. As a result, OSG renders the second child using its parent's state, with lighting enabled.

The default behavior of inheritance is useful in many cases. Some rendering situations require different behavior, however. Imagine a scene graph with nodes that specify a filled polygon mode. To render this scene graph in wireframe, your application must override the polygon mode state regardless of where it occurs in the scene graph.

OSG allows you to change the state inheritance behavior individually for each attribute and mode at any point in the scene graph. The following enumerants are available:

- `osg::StateAttribute::OVERRIDE`—If you set an attribute or mode to `OVERRIDE`, all children inherit this attribute or mode regardless of whether they change that state or not.
- `osg::StateAttribute::PROTECTED`—There is an exception to `OVERRIDE`, however. You can cause an attribute or mode to be immune from overriding by setting it to `PROTECTED`.
- `osg::StateAttribute::INHERIT`—This mode forces the child state to inherit from its parent. In effect, it removes the state from the child and uses the parent state instead.

You can specify these values by bitwise ORing them into the second parameter of the **setAttribute(), setMode(),** and **setAttributeAndModes()** methods. The following code snippet forces wireframe rendering on a scene graph:

```
// Obtain the root node's StateSet.
osg::StateSet* state = root->getOrCreateStateSet();

// Create a PolygonMode attribute
osg::PolygonMode* pm = new osg::PolygonMode)
      osg::PolygonMode::FRONT_AND_BACK, osg::PolygonMode::LINE );
// Force wireframe rendering.
state->setAttributeAndModes( pm,
      osg::StateAttribute::ON | osg::StateAttribute::OVERRIDE );
```

Use `PROTECTED` to ensure that parent state never overrides child state. For example, you might create a scene containing lights that use luminance lighting for the light source geometry. If a parent node disables lighting, the light geometry would render incorrectly. In this case, use `PROTECTED` on the light geometry state set for `GL_LIGHTING` to ensure it remains enabled.

2.4.3 Example Code for Setting State

Section **2.3.2 The Parent Interface** describes adding the same node as a child to multiple parents. The section, **The MatrixTransform Node,** describes the **MatrixTransform** node for transforming geometry. Section **2.4 Rendering State** describes OSG state. The following example code combines all three concepts.

This section presents a simple example of how to modify OSG rendering state. The code creates a **Geode** with a **Drawable** that contains two quadrilaterals, but multiply parents it

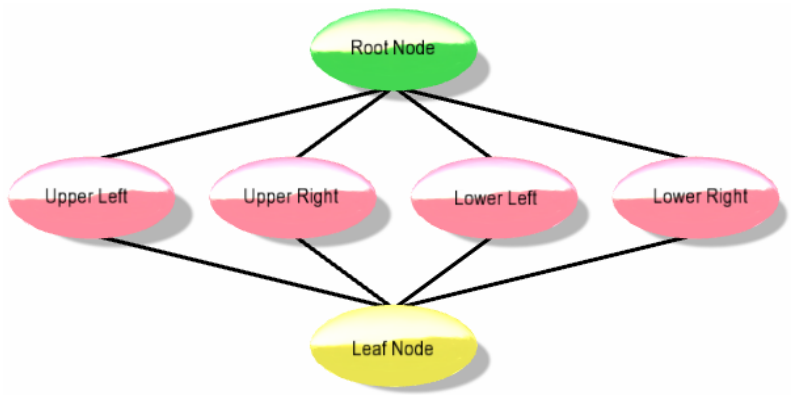


Figure 2-8
Scene graph for the state example program
The State example renders the same **Geode** four times. It positions the **Geode** using four **MatrixTransform** nodes, each with their own **StateSet.**

to four **MatrixTransform** nodes, each with their own **StateSet.** Figure 2-8 shows this scene graph, and Listing 2-3 presents the code for creating it. The code is from the State example in the book's source code.

Listing 2-3

State modifications

This portion of the State example adds several Drawables to a single Geode. The code sets different state for each Drawable, and also disables lighting for all geometry by disabling it in the Geode's StateSet..

```
#include <osg/Geode>
#include <osg/Group>
#include <osg/MatrixTransform>
#include <osg/Geode>
#include <osg/Geometry>
#include <osg/StateSet>
#include <osg/StateAttribute>
#include <osg/ShadeModel>
#include <osg/CullFace>
#include <osg/PolygonMode>
#include <osg/LineWidth>

...

osg::ref_ptr<osg::Node>
createSceneGraph()
{
    // Create the root node Group.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    {
        // Disable lighting in the root node's StateSet. Make
        //   it PROTECTED to prevent osgviewer from enabling it.
        osg::StateSet* state = root->getOrCreateStateSet();
        state->setMode( GL_LIGHTING,
            osg::StateAttribute::OFF |
            osg::StateAttribute::PROTECTED );
    }

    // Create the leaf node Geode and attach the Drawable.
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( createDrawable().get() );

    osg::Matrix m;
    {
        // Upper-left: Render the drawable with default state.
        osg::ref_ptr<osg::MatrixTransform> mt =
                new osg::MatrixTransform;
        m.makeTranslate( -2.f, 0.f, 2.f );
        mt->setMatrix( m );
        root->addChild( mt.get() );
```

```
    mt->addChild( geode.get() );
}
{
    // Upper-right Set shade model to FLAT.
    osg::ref_ptr<osg::MatrixTransform> mt =
            new osg::MatrixTransform;
    m.makeTranslate( 2.f, 0.f, 2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );

    osg::StateSet* state = mt->getOrCreateStateSet();
    osg::ShadeModel* sm = new osg::ShadeModel();
    sm->setMode( osg::ShadeModel::FLAT );
    state->setAttribute( sm );
}
{
    // Lower-left: Enable back face culling.
    osg::ref_ptr<osg::MatrixTransform> mt =
            new osg::MatrixTransform;
    m.makeTranslate( -2.f, 0.f, -2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );

    osg::StateSet* state = mt->getOrCreateStateSet();
    osg::CullFace* cf = new osg::CullFace(); // Default: BACK
    state->setAttributeAndModes( cf );
}
{
    // Lower-right: Set polygon mode to LINE in
    //   draw3's StateSet.
    osg::ref_ptr<osg::MatrixTransform> mt =
            new osg::MatrixTransform;
    m.makeTranslate( 2.f, 0.f, -2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );

    osg::StateSet* state = mt->getOrCreateStateSet();
    osg::PolygonMode* pm = new osg::PolygonMode(
        osg::PolygonMode::FRONT_AND_BACK,
        osg::PolygonMode::LINE );
    state->setAttributeAndModes( pm );

    // Also set the line width to 3.
    osg::LineWidth* lw = new osg::LineWidth( 3.f );
    state->setAttribute( lw );
}

return root.get();
```

```
}
```

The State example code creates a **Group** node, called *root,* to act as the root node of the scene graph, and configures *root*'s **StateSet** to disable lighting. It uses the `PROTECTED` flag to prevent osgviewer from enabling it.

The code uses a function called createDrawable() to create a **Geometry** object containing two quadrilaterals with color per vertex. Listing 2-3 doesn't show createDrawable(). Download the example code to see this function. As you can imagine, it's similar to the code in Listing 2-1. The code attaches the returned **Drawable** to a new **Geode,** called *geode.*

To render *geode* at four different locations, the code creates four **MatrixTransform** nodes, each with a different translation, then adds geode as a child to each **MatrixTransform.** To change the appearance of *geode,* each **MatrixTransform** has a unique **StateSet.**

- The first **MatrixTransform,** which translates *geode* to the upper-left, uses an unmodified **StateSet.** *geode* inherits all state from its parent. In this case, it uses default state.

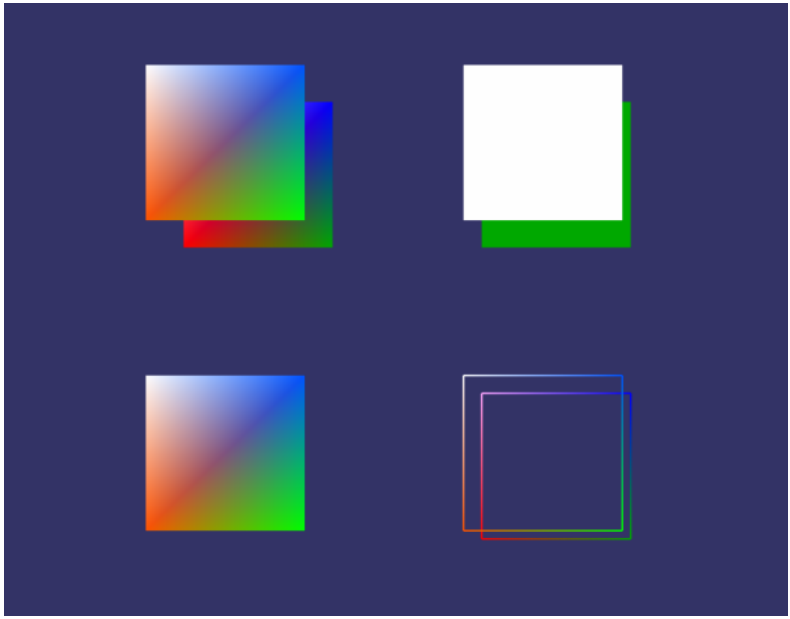


Figure 2-9
The state example scene graph displayed in osgviewer
This rendering of the scene graph created in Listing 2-3 demonstrates different state attributes and modes. Upper left: default state. Upper right: shade model set to FLAT. Lower left: face culling enabled for back faces; Lower right: polygon mode set to LINE and line width set to 3.0. Lighting is disabled for the entire scene graph.

- The second **MatrixTransform,** which translates *geode* to the upper-right, uses a **StateSet** with a **ShadeModel StateAttribute** set to `FLAT`. This causes *geode*'s quadrilaterals to render with flat shading. Their colors come from the color of the final triggering vertex.
- The third **MatrixTransform,** which translates *geode* to the lower-left, uses a **StateSet** with a **CullFace StateAttribute.** By default, **CullFace** culls `BACK` faces, though you can change this with a constructor parameter. The call to **setAttributeAndModes(** *cf* **)** attaches the **CullFace** and simultaneously enables `GL_CULL_FACE`. (The two quadrilaterals returned by createDrawable() have opposite vertex winding order, so one will always be a back face regardless of the viewpoint.)
- The final **MatrixTransform** translates *geode* to the lower-right. Its **StateSet** contains two **StateAttribute**s, a **PolygonMode** and a **LineWidth.** The code sets the polygon mode to `LINE` for both `FRONT_AND_BACK` faces, and sets the line width to 3.0.

Like the Simple example, the State example writes its scene graph to a file, State.osg. After running the State example, display the output in osgviewer with the following command:

```
osgviewer State.osg
```

Figure 2-9 shows how this file appears when loaded in osgviewer.

2.4.4 Texture Mapping

OSG fully supports OpenGL's texture mapping features. To perform basic texture mapping in your application, your code must do the following:

- Specify texture coordinates with your geometry.
- Create a texture attribute object and store texture image data in it.
- Set the appropriate texture attributes and modes in a **StateSet.**

This section provides information on each step. In the book's example code, the TextureMapping example demonstrates basic texture mapping techniques. To conserve space, that code is not reproduced in this text.

Texture Coordinates

Section **2.2 Geodes and Geometry** explains the **Geometry** object's interface for setting vertex, normal, and color data. **Geometry** also allows your application to specify one or more arrays of texture coordinate data. When you specify texture coordinates, you must also specify the corresponding texture unit. OSG uses the texture unit value for multitexture.

Multitexture

OpenGL was originally released without multitexture support. After adding multitexture, OpenGL continued to support the non-multitexture interface for backwards compatibility. Under the hood, OpenGL interprets the non-multitexture interface as a modification to texture unit 0.

Unlike OpenGL, OSG doesn't provide a non-multitexture interface. As a result, your application must specify a texture unit for texture coordinate data and texture state. To use a single texture, simply specify texture unit 0.

The code snippet that follows creates an **osg::Vec2Array,** stores texture coordinates in it, and attaches it to the **Geometry** for use with texture unit 0. To apply multiple textures to a single **Geometry,** attach multiple texture coordinate arrays to the **Geometry** and assign a different texture unit to each array.

```
// Create a Geometry object.
osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;

// Create a Vec2Array of texture coordinates for texture unit 0
//   and attach it to the geom.
osg::ref_ptr<osg::Vec2Array> tc = new osg::Vec2Array;
geom->setTexCoordArray( 0, tc.get() );
tc->push_back( osg::Vec2( 0.f, 0.f ) );
tc->push_back( osg::Vec2( 1.f, 0.f ) );
tc->push_back( osg::Vec2( 1.f, 1.f ) );
tc->push_back( osg::Vec2( 0.f, 1.f ) );
```

The first parameter to **osg::Geometry::setTexCoordArray()** is the texture unit, and the second parameter is the texture coordinate data array. There's no need for an **osg::Geometry::setTexCoordBinding()** entry point. Texture coordinates are always bound per vertex.

Loading Images

Your application uses two classes for basic 2D texture mapping: **osg::Texture2D** and **osg::Image. Texture2D** is a **StateAttribute** that manages the OpenGL texture object, and **Image** manages image pixel data. To use a 2D image file for a texture map, set the file name on an **Image** object and attach the **Image** to a **Texture2D.** The following code snippet uses the file sun.tif as a texture map.

```
#include <osg/Texture2D>
#include <osg/Image>
...
osg::StateSet* state = node->getOrCreateStateSet();
```

```
// Load the texture image
osg::ref_ptr<osg::Image> image = new osg::Image;
image->setFileName( "sun.tif" );

// Attach the image in a Texture2D object
osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
tex->setImage( image.get() );
```

After configuring the **Texture2D** attribute, attach it to a **StateSet.** The next section, **0Texture State,** describes this step in more detail.

Applications that make heavy use of texture mapping require tight control over memory management. The **Image** object derives from **Referenced,** and **Texture2D** keeps a **ref_ptr<>** to the **Image.** During the first render pass, OSG creates an OpenGL texture object to store the image data, resulting in two copies of the texture image: one in the **Image** object, and the other owned by OpenGL. In a simple single-context rendering scenario, you can reduce memory consumption by configuring the **Texture2D** to release its reference to the **Image.** If the **Texture2D** is the only object referencing the **Image,** this will cause OSG to delete the Image and the memory it occupies. The following code demonstrates how to configure **Texture2D** to release its reference to the **Image** object:

```
// After creating the OpenGL texture object, release the
//   internal ref_ptr<Image> (delete the Image).
tex->setUnRefImageDataAfterApply( true );
```

By default, **Texture2D** doesn't release its **Image** reference. This is the desired behavior in a multi-context rendering scenario if texture objects aren't shared between the contexts.

Texture State

The interface for setting texture state allows your application to specify state for each texture unit. The texture state interface is very similar to the non-texture state interface, however. To attach a texture attribute to a **StateSet,** use **osg::StateSet::setTextureAttribute().** The first parameter to **setTextureAttribute()** is the texture unit, and the second parameter is a texture attribute derived from **StateAttribute,** There are six valid texture attributes, one for each of the five texture types (**osg::Texture1D, osg::Texture2D, osg::Texture3D, osg::TextureCubeMap,** and **osg::TextureRectangle**) and an attribute for texture coordinate generation (**osg::TexGen**).

Given a **Texture2D** attribute *tex* and a **StateSet** *state,* the following code attaches *tex* to *state* for use on texture unit 0:

```
// Create a Texture2D attribute.
Osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
// ...

// Attach the texture attribute for texture unit 0.
state->setTextureAttribute( 0, tex.get() );
```

Similarly, to set a texture mode, use **osg::StateSet::setTextureMode().** This method is analogous to the **setMode()** method. You can set the following modes with **setTextureMode():** `GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP, GL_TEXTURE_RECTANGLE, GL_TEXTURE_GEN_Q, GL_TEXTURE_GEN_R, GL_TEXTURE_GEN_S,` and `GL_TEXTURE_GEN_T.`

Like **setTextureAttribute(),** the first parameter to **setTextureMode()** is the desired texture unit. The following code snippet disables 2D texture mapping for texture unit 1.

```
state->setTextureMode( 1, GL_TEXTURE_2D,
        osg::StateAttribute::OFF );
```

Of course, use **osg::StateSet::setTextureAttributesAndModes()** to attach an attribute to a **StateSet** and simultaneously enable its corresponding mode. If the attribute is a **TexGen** object, **setTextureAttributesAndModes()** sets the texture coordinate generation modes `GL_TEXTURE_GEN_Q, GL_TEXTURE_GEN_R, GL_TEXTURE_GEN_S,` and `GL_TEXTURE_GEN_T.` The mode is implicit for other texture attributes. For example, in the following code, **setTextureAttributesAndModes()** enables `GL_TEXTURE_2D` because the attribute passed as the second parameter is a **Texture2D** object.

```
// Create a Texture2D attribute.
osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
// ...

// Attach the 2D texture attribute and enable GL_TEXTURE_2D,
//   both on texture unit 0.
state->setTextureAttributeAndModes( 0, tex );
```

setTextureAttributeAndModes() has a third parameter that defaults to ON to enable the texture mode. Like **setAttributeAndModes(),** you can modify the inheritance behavior of texture attributes by bitwise ORing `OVERRIDE, PROTECTED,` or `INHERIT` into this final parameter. You can also specify inheritance flags in the third parameters to **setTextureMode()** and **setTextureAttribute().**

2.4.5 Lighting

OSG fully supports OpenGL lighting, including material properties, light properties, and lighting models. Like OpenGL, light sources aren't visible—OSG doesn't render a bulb or other physical manifestation. Also, light sources create shading effects, but don't create shadows—use the osgShadow NodeKit for that.

To use lighting in your application, your code must do the following:

- Specify normals with your geometry.
- Enable lighting and set other lighting state.
- Specify the light source properties and attach it to your scene graph.

- Specify surface material properties.

This section provides information on each step.

Normals

Correct lighting requires that unit length normals accompany your geometric data. Section **2.2 Geodes and Geometry** describes how to set normal arrays and bindings in a **Geometry** object.

As in most 3D APIs, your normals must be unit length to obtain acceptable results. Keep in mind that scale transformation can alter the length of your normals. If your **Geometry**'s normal array contains unit length normals, but lighting results appear too bright or too dim, a scale transformation could be the culprit. The most efficient solution is to enable normal rescaling in the **StateSet.**

```
osg::StateSet* state = geode->setOrCreateStateSet();
state->setMode( GL_RESCALE_NORMAL, osg::StateAttribute::ON );
```

As in OpenGL, enabling this feature will only restore your normals to unit length if they were affected by uniform scaling. If the scaling in your scene graph is non-uniform, you can enable normalization to restore them to unit length.

```
osg::StateSet* state = geode->setOrCreateStateSet();
state->setMode( GL_NORMALIZE, osg::StateAttribute::ON );
```

Normalization is typically more expensive that normal rescaling; avoid it if possible.

Lighting State

To obtain any lighting effects from OSG, you must enable lighting and at least one light source. The osgviewer application does this by default, by setting the appropriate modes in a root node **StateSet.** You can do the same in your application. The following code snippet enables lighting and two light sources (`GL_LIGHT0` and `GL_LIGHT1`) on *root*'s **StateSet.**

```
osg::StateSet* state = root->getOrCreateStateSet();
state->setMode( GL_LIGHTING, osg::StateAttribute::ON );
state->setMode( GL_LIGHT0, osg::StateAttribute::ON );
state->setMode( GL_LIGHT1, osg::StateAttribute::ON );
```

The following sections describe how to control individual light source properties, such as its position and color, and how to control the OpenGL color material feature (and set surface material colors).

OSG also provides the **osg::LightModel StateAttribute** to control the global ambient color, local viewer, two-sided lighting, and separate specular color OpenGL features.

Light Sources

To add a light source to your scene, create an **osg::Light** object to define the light source parameters. Add the **Light** to an **osg::LightSource** node, and add the **LightSource** node to your scene graph. **LightSource** is effectively a leaf node containing a single **Light** definition. The light source defined by **Light** affects your entire scene.

OSG supports eight light sources, `GL_LIGHT0` through `GL_LIGHT7,` and possibly more depending on your underlying OpenGL implementation. Enable these lights with **setMode()** as described previously. To associate a **Light** object with an OpenGL light source, set the light's number. To associate a **Light** object with `GL_LIGHT2,` for example, set its number to two:

```
// Create a Light object to control GL_LIGHT2's parameters.
osg::ref_ptr<osg::Light> light = new osg::Light;
light->setLightNum( 2 );
```

The light number is zero by default.

The **Light** class exposes much of the functionality found in the OpenGL **glLight()** command. **Light** methods allow your application to set the light's ambient, diffuse, and specular color contributions. You can create point, directional, or spot lights, and you can specify attenuation to diminish the light intensity with distance. The following code creates a **Light** object and sets some commonly-used parameters.

```
// Create a white spot light source
osg::ref_ptr<osg::Light> light = new osg::Light;
light->setAmbient( osg::Vec4( .1f, .1f, .1f, 1.f ));
light->setDiffuse( osg::Vec4( .8f, .8f, .8f, 1.f ));
light->setSpecular( osg::Vec4( .8f, .8f, .8f, 1.f ));
light->setPosition( osg::Vec3( 0.f, 0.f, 0.f ));
light->setDirection( osg::Vec3( 1.f, 0.f, 0.f ));
light->setSpotCutoff(25.f );
```

To add the **Light** to your scene, create a **LightSource** node, add the **Light** to the **LightSource,** and attach the **LightSource** to your scene graph. The location within your scene graph where you attach the **LightSource** node affects the **Light** position. OSG transforms the **Light** position by the current transformation for the **LightSource** node. OSG application developers typically attach the **LightSource** as a child to a **MatrixTransform** to control the **Light** position, as shown in the following code:

Positional State

When an OpenGL application issues a **glLight()** command to set the light position, OpenGL transforms the position by the current model-view matrix. In OSG this concept is known as *positional state.* During the cull traversal, OSG adds positional state items to a positional state container to ensure they are transformed correctly during the draw traversal.

Warning

Many new OSG developers mistakenly assume that **LightSource** child subgraphs are automatically lit. This is not the case! OSG lights your scene graph based on current state, not based on any hierarchical relationship to the **LightSource** node. `GL_LIGHTING` and at least one light source (`GL_LIGHT0`, for example) must be enabled for OSG to light your scene graph.

Think of **LightSource** as a leaf node that contains a single **Light.** However, you can attach children to a **LightSource** node because **LightSource** derives from **Group.** Typically, this is where applications attach geometry to render the physical manifestation of the light.

```
// Create the Light and set its properties.
osg::ref_ptr<osg::Light> light = new osg::Light;
...

// Create a MatrixTransform to position the Light.
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
m.makeTranslate( osg::Vec3( -3.f, 2.f, 5.f ) );
mt->setMatrix( m );

// Add the Light to a LightSource. Add the LightSource and
//   MatrixTransform to the scene graph.
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;
parent->addChild( mt.get() );
mt->addChild( ls.get() );
ls->setLight( light.get() );
```

OSG transforms the **Light** position by the current transformation for the **LightSource** by default. You can disable this by setting the **LightSource**'s reference frame, in the same way that you set the reference frame for a **Transform** node as described in **2.3.3 Transform Nodes.** The following code causes OSG to ignore the **LightSource**'s current transformation and treat the **Light** position as an absolute value.

```
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;
ls->setReferenceFrame( osg::LightSource::ABSOLUTE_RF );
```

Material Properties

The **osg::Material StateAttribute** exposes the functionality available in OpenGL's **glMaterial()** and **glColorMaterial()** commands. To set Material properties in your application, create a **Material** object, set colors and other parameters, and attach it to a **StateSet** in your scene graph.

Material allows you to set ambient, diffuse, specular, and emissive material colors, as well as the specular exponent (or shininess) parameter. Material defines the enumerants `FRONT`, `BACK`, and `FRONT_AND_BACK` to allow your application to set material properties for both front and back faces. For example, the following code sets the diffuse, specular, and specular exponent parameters for front-facing primitives:

```
osg::StateSet* state = node->getOrCreateStateSet();
osg::ref_ptr<osg::Material> mat = new osg::Material;
mat->setDiffuse( osg::Material::FRONT,
    osg::Vec4( .2f, .9f, .9f, 1.f ) );
mat->setSpecular( osg::Material::FRONT,
    osg::Vec4( 1.f, 1.f, 1.f, 1.f ) );
mat->setShininess( osg::Material::FRONT, 96.f );
state->setAttribute( mat.get() );
```

Like OpenGL, the specular exponent must be in the range 1.0 to 128.0 unless an implementation relaxes this rule with an extension.

Setting material properties directly can be expensive for some OpenGL implementations to process. The color material feature allows your application to change certain material properties by changing the primary color. This is more efficient than changing material properties directly, facilitates coherency between lit and unlit scenes, and satisfies application material requirements in many cases.

To enable the color material feature, call **Material::setColorMode(). Material** defines the following enumerants, `AMBIENT`, `DIFFUSE`, `SPECULAR`, `EMISSION`, `AMBIENT_AND_DIFFUSE`, and `OFF`. By default, the color mode is `OFF` and color material is disabled. When your application sets the color mode to any other value, OSG enables color material for the specified material properties, and subsequent changes to the primary color change that material property. The following code segment enables color material so that front-facing ambient and diffuse material colors track changes to the primary color:

```
osg::StateSet* state = node->getOrCreateStateSet();
osg::ref_ptr<osg::Material> mat = new osg::Material;
mat->setColorMode( osg::Material::AMBIENT_AND_DIFFUSE );
state->setAttribute( mat.get() );
```

Note that **Material** automatically enables or disables `GL_COLOR_MATERIAL` based on the color mode value. Your application doesn't need to call **setAttributeAndModes()** to enable or disable this feature.

Lighting Example

The Lighting example in this book's source code creates two light sources and renders geometry using seven different material properties. To conserve space, the source code hasn't been reproduced here. The example writes the scene graph to a file called Lighting.osg. Display the scene graph with the following command:

```
osgviewer Lighting.osg
```

Figure 2-10 shows the scene graph displayed in osgviewer.

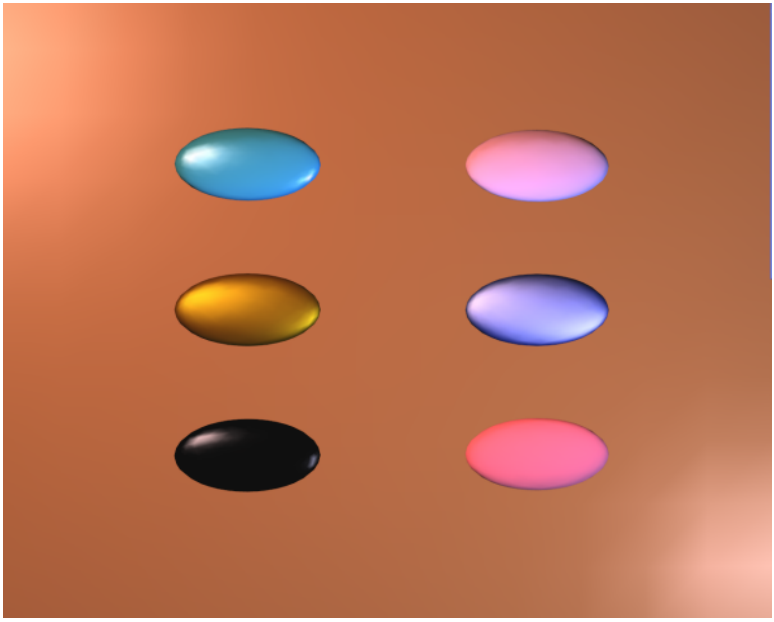


Figure 2-10
The Lighting example scene graph displayed in osgviewer
This example renders six lozenges and a background plane, each with their own material settings. Two light sources illuminate the scene.

2.5 File I/O

This chapter has described programmatic techniques for creating scene graphs with geometry and state, and most applications create some geometry programmatically. However, applications commonly load and display large, complex models from file. Applications require a function that loads a model from file and returns it as a prebuilt scene graph.

The osgDB library provides an interface that lets your application read and write 2D image and 3D model files. The osgDB manages the OSG plugin system to support different file types. Section **1.6.3 Components** introduced the concept of plugins, and Figure 1-9 shows how plugins fit into the overall OSG architecture.

The examples discussed in this chapter use the osgDB for file I/O. All the examples use the .osg plugin to write the scene graph to an .osg file. The Lighting example uses the .osg plugin to load a subgraph from a file called lozenge.osg, and the TextureMapping example

uses the .png plugin to load its texture image. However, the previous text doesn't explain this functionality. This section describes plugins in greater detail so that you can use them effectively in your application. It describes the interface for reading and writing files, how OSG searches for files, and how OSG selects plugins to load those files.

2.5.1 Interface

The osgDB provides a file I/O interface that completely hides the underlying plugin system from the application. Two osgDB header files define this interface:

```
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
```

To use the osgDB for file I/O in your application, include these header files in your source code. They define several functions in the osgDB namespace for performing file I/O.

Reading Files

Use the functions **osgDB::readNodeFile()** and **osgDB::readImageFile()** to read 3D model and 2D image files.

```
osg::Node* osgDB::readNodeFile( const std::string& filename );
osg::Image* osgDB::readImageFile( const std::string& filename );
```

Use **readNodeFile()** to load a 3D model file. OSG recognizes the file type from the file name extension and uses the appropriate plugin to convert the file to a scene graph. **readNodeFile()** returns a pointer to the root node of the scene graph to your application. Similarly, **readImageFile()** loads 2D image files and returns a pointer to an **Image** object.

The *filename* parameter can contain an absolute path or a relative path. If you specify an absolute path, OSG looks for the file in the specified location.

If *filename* contains a relative path (or contains just a file name), OSG searches for the file using the osgDB *data file path list*. Users can set this list of directories using the OSG_FILE_PATH environment variable, as described in section **1.3.3 Environment Variables.** D

To add specific data directories to the data file path list, use the function **osgDB::Registry::getDataFilePathList().** The **osgDB::Registry** is a singleton—to call this function, access the singleton instance. This function returns a reference to an **osgDB::FilePathList,** which is simply a **std::deque<std::string>.** To add a directory stored in the string *newpath,* for example, use the following line of code:

```
osgDB::Registry::instance()->getDataFilePathList().push_back
        ( newpath );
```

If OSG is unable to load your file for any reason, both of these functions return NULL pointers. To determine why the file isn't loaded, set the OSG_NOTIFY_LEVEL

environment variable to a higher verbosity level (such as WARN), attempt to load the file again, and check for warnings or error messages displayed in your application's console.

Writing Files

Use the functions **osgDB::writeNodeFile()** and **osgDB::writeImageFile()** to write data to 3D model and 2D image files.

```
bool osgDB::writeNodeFile( const osg::Node& node,
        const std::string& filename );
bool osgDB::writeImageFile( const osg::Image& image,
        const std::string& filename );
```

If OSG can't write the files for any reason, these functions return false. Again, set OSG_NOTIFY_LEVEL to WARN to view messages about why the operation failed. If the write operation succeeds, these functions return true.

If the *filename* parameter contains an absolute path, **writeNodeFile()** and **writeImageFile()** attempt to write the file to the absolute location. If *filename* contains a relative path (or no path at all), these functions attempt to write the file relative to the current directory.

OSG will overwrite an existing file with the same name without warning. To prevent this behavior, your application should check for existing files and take an appropriate action.

2.5.2 Plugin Discovery and Registration

The OSG plugins are a group of shared libraries that implement the interface described in the osgDB header file ReaderWriter. In order for OSG to find the plugins, their directory must be listed in the PATH environment variable on Windows or within the LD_LIBRARY_PATH environment variable on Linux. End users can specify additional search directories in the OSG_LIBRARY_PATH environment variable.

OSG recognizes plugin libraries only if they conform to the following naming convention.

- Apple—`osgdb_<name>`
- Linux—`osgdb_<name>.so`

Warning

Plugins might not support both read and write operations. As an example, the OSG v1.3 3D Studio Max plugin supports reading .3ds files, but doesn't support writing .3ds files. In fact, most OSG plugins support file import, but don't support file export.

For the most up-to-date information on supported file types, see the OSG Wiki Web site [OSGWiki].

- Windows—`osgdb_<name>.dll`

`<name>` is usually the file extension. For example, the plugin to read GIF image files is called `osgdb_gif.so` on Linux systems.

It's not always practical for developers to name their plugins using the file extension because some plugins support multiple file extensions. The plugin that loads RGB image files, for example, can load files with the following extensions: .sgi, .int, .inta. .bw, .rgba, and .rgb. The **osgDB::Registry** constructor contains special code to support such plugins. The **Registry** maintains an extension alias map that associates many different extensions with the plugin that supports them.

OSG doesn't look for all plugins and load them to discover which file types they support. This would be too great an expense to incur at application startup. Instead, OSG implements the Chain of Responsibility design pattern [Gamma95] to load as few plugins as possible. When your application attempts to read or write a file using the osgDB, OSG performs the following steps to find an appropriate plugin:

1. OSG searches its list of registered plugins for a plugin that supports the file. Initially, the list of registered plugins contains only those plugins that were registered in the **Registry** constructor. If it finds a plugin that supports the file type and the plugin performs the I/O operation successfully, OSG returns the loaded data.
2. If no registered plugins support the file type, or the I/O operation failed, OSG creates an appropriate plugin name using the file name rules described previously, and attempts to load that plugin library. If the load succeeds, OSG adds that plugin to its list of registered plugins.
3. OSG repeats step 1. If the file I/O operation fails again, OSG returns failure.

In general, plugins just work, and you rarely need to know how OSG supports file I/O internally. When a file I/O operation fails, however, this information will help you track down the source of the issue.

2.6 NodeKits and osgText

OSG provides a rich feature set. Nonetheless, developers commonly need to derive their own specialized nodes from OSG's core node classes. Moreover, derived functionality usually doesn't belong in core OSG and is more appropriately made available as an additional support library. A NodeKit is a library that extends core OSG functionality with specialized **Node**s, **StateAttribute**s, or **Drawable**s, and also adds support for these new classes to the .osg file format with a dot OSG wrapper.

Section **1.6.3 Components** introduces the concept of NodeKits and provides a high-level overview of the NodeKits available in the OSG distribution. This section provides an example of using one popular NodeKit, osgText, to display texture mapped text in your scene graph.

2.6.1 osgText Components

The osgText library defines a namespace, **osgText.** Within that namespace are a small handful of related classes for loading fonts and rendering strings of text.

The osgText library's key component is the **osgText::Text** class. **Text** derives from **Drawable,** and your application must add **Text** instances to a **Geode** using **addDrawable()** (in the same way that you add instances of **Geometry**). **Text** displays a character string of arbitrary length. Typically, your application creates a **Text** object for each string to display.

The other key osgText component is the **osgText::Font** class. osgText provides a convenience function to create a **Font** from a font file name. **Font** uses the FreeType plugin to load a font file. When your application associates a **Font** with a **Text** object, **Font** creates a texture map containing only the glyphs necessary to render the text string. At render time, **Text** draws a texture mapped quadrilateral for each character in its text string using texture coordinates that display the corresponding glyph from the texture.

osgText also defines a **String** class to support multibyte and international text encodings.

2.6.2 Using osgText

Two header files define the **Text** and **Font** objects. The following code illustrates how to include them in your application.

```
#include <osgText/Font>
#include <osgText/Text>
```

To use osgText in your application, you usually need to perform three steps:

1. If you will display multiple text strings using the same font, create a single **Font** object that you can share between all **Text** objects.
2. For each text string to display, create a **Text** object. Specify options for alignment, orientation, position, and size. Assign the **Font** object you created in step 1 to the new **Text** object.
3. Add your **Text** objects to a **Geode** using **addDrawable().** You can add multiple **Text** objects to a single **Geode,** or create multiple **Geode**s depending on your application requirements. Add your **Geode**s as child nodes in your scene graph.

The following code demonstrates how to create a **Font** object using the Courier New TrueType font file, cour.ttf.

```
osg::ref_ptr<osgText::Font> font =
        osgText::readFontFile( "fonts/cour.ttf" );
```

osgText::readFontFile() is a convenience function that uses the FreeType OSG plugin to load the font file. It uses the osgDB to find the file, so it looks in the directories

specified in OSG_FILE_PATH, as described in section **2.5 File I/O**. However, **readFontFile()** also searches a list of font directories for various platforms. If **readFontFile()** is unable to find the specified file, or the file isn't a font, **readFontFile()** returns NULL.

Use the following code to create a **Text** object, assign a font to it, and set the text to display.

```
osg::ref_ptr<osgText::Text> text = new osgText::Text;
text->setFont( font.get() );
text->setText( "Display this message." );
```

Although **Text::setText()** appears to take a **std::string** as a parameter, it actually takes an **osgText::String** to support multibyte encodings. **osgText::String** has several non-explicit constructors that accept **std::string** or string literal parameters. In the code above, calling setText() with a string literal parameter causes the string literal to be converted to an **osgText::String,** probably at runtime.

If all attempts to load a font with **readFontFile()** fail and your application is unable to find any usable fonts on the runtime system, don't call **Text::setFont().** In this case, **Text** will use a default font that is always available.

Text has several methods that control its size, appearance, orientation, and position. The following sections describe how to control many of these parameters.

Position

Text, like **Geometry,** transforms its object coordinate position during the cull and draw traversals. By default, the position is the object coordinate origin. You can change this value with the **Text::setPosition()** method, which takes a **Vec3** as a parameter.

```
// Draw the text at (10., 0., 1.).
text->setPosition( osg::Vec3( 10.f, 0.f, 1.f ) );
```

Position alone doesn't determine where the text appears in your final image. **Text** uses the transformed position, along with the orientation and alignment values, to determine where to render the text. Orientation and alignment are discussed next.

Orientation

Orientation determines which direction the rendered text faces in 3D space. Set the orientation with the **Text::setAxisAlignment()** method, and pass one of the **Text::AxisAlignment** enumerants as a parameter. To create billboard-style text that always faces the viewpoint, use `Text::SCREEN.`

```
text->setAxisAlignment( osgText::Text::SCREEN );
```

Alternatively, you can make the text lie in an axis-aligned plane. The default orientation is `Text::XY_PLANE`, which places text in the *xy* plane facing positive *z*.

```
text->setAxisAlignment( osgText::Text::XY_PLANE );
```

There are seven `Text::AxisAlignment` enumerant values: `Text::XY_PLANE` (the default), `Text::XZ_PLANE`, and `Text::YZ_PLANE` place text in the specified axis plane, facing the positive remaining axis; `Text::REVERSED_XY_PLANE`, `Text::REVERSED_XZ_PLANE`, and `Text::REVERSED_YZ_PLANE` are similar, but text faces the negative remaining axis; and `Text::SCREEN` renders text that always faces the screen.

Alignment

Alignment is analogous to text alignment in a word processor, or cell alignment in a spreadsheet. It determines the horizontal and vertical alignment of the rendered text relative to its position (as set with **setPosition()**). **Text** defines a set of enumerants called `Text::AlignmentType`. Each enumerant name encodes first the horizontal alignment, then the vertical alignment. The default is `Text::LEFT_BASE_LINE`, which aligns the left edge of the text with the position horizontally, and aligns the font's base line with the position vertically. Figure 2-11 illustrates how different alignment types affect the text location relative to its position.

To change the text alignment, call **Text::setAlignment().** The following code specifies horizontal center alignment and top vertical alignment:

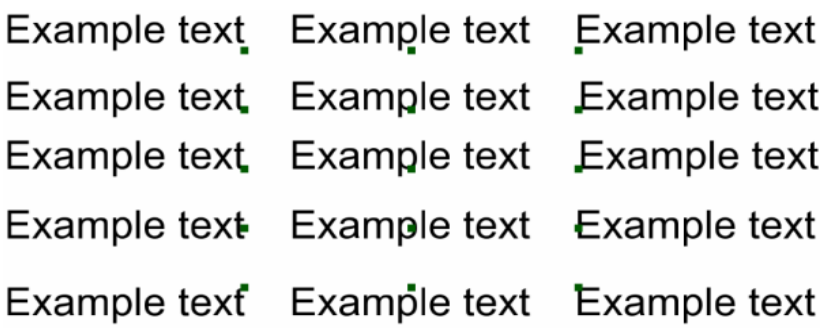


Figure 2-11
Text alignment types
This figure shows the effect of the 15 different `AlignmentType` enumerants. For each example, the text's position (set with **setPosition()**) is illustrated by a dark green point. Left to right, top to bottom: `RIGHT_BOTTOM`, `CENTER_BOTTOM`, `LEFT_BOTTOM`, `RIGHT_BOTTOM_BASE_LINE`, `CENTER_BOTTOM_BASE_LINE`, `LEFT_BOTTOM_BASE_LINE`, `RIGHT_BASE_LINE`, `CENTER_BASE_LINE`, `LEFT_BASE_LINE`, `RIGHT_CENTER`, `CENTER_CENTER`, `LEFT_CENTER`, `RIGHT_TOP`, `CENTER_TOP`, and `LEFT_TOP`.

```
text->setAlignment( osgText::Text::CENTER_TOP );
```

Size

The default character height is 32 object coordinate units. The character width is variable and depends on the font. **Text** renders characters with the correct aspect ratio according to information stored in the **Font** object.

To change the default character height, call **Text::setCharacterSize().** The following code reduces the character height to one object coordinate unit.

```
text->setCharacterSize( 1.0f );
```

By default, **Text** interprets the parameter to **setCharacterSize()** as an object coordinate value. **Text** also allows you to specify the character height in screen coordinates rather than object coordinates. Use the **Text::setCharacterSizeMode()** method to specify screen coordinates.

```
text->setCharacterSizeMode( osgText::Text::SCREEN_COORDS );
```

After changing the character height mode to screen coordinates, **Text** scales text geometry to maintain a constant screen size regardless of perspective effects. Note that OSG sizes the text during the cull traversal based on the last frame's projection information, which causes a single frame latency. This latency is usually not noticeable for applications that have high frame coherency.

Resolution

Applications regularly need to vary the glyph resolution used by the font texture map to avoid blurred characters. By default, the osgText NodeKit allocates 32×32 texels per glyph. To change this value, use the **Text::setFontResolution()** method. The following code increases the font resolution so that osgText allocates 128×128 texels per character.

```
text->setFontResolution( 128, 128 );
```

When multiple **Text** objects with different resolutions share the same **Font** object, and the same characters are present in the two text strings, the font texture map will contain multiple copies of the redundant characters at different resolutions.

Note that increasing the font resolution also increases demand for hardware resources such as graphics card texture memory. You should use the smallest font resolution that produces acceptable results for your character height.

Color

Text colors character strings white by default. You can change this default with the **Text::setColor()** method. To specify the color, pass an **osg::Vec4** *rgba* color value as a parameter to **setColor().** The following code causes Text to render blue character strings.

```
// Set the text color to blue.
text->setColor( osg::Vec4( 0.f, 0.f, 1.f, 1.f ) );
```

The osgText library's **Text** and **Font** classes allow you to control several additional parameters that are beyond the scope of this book. Peruse the include/osgText/Text and include/osgText/Font header files for more information.

2.6.3 Text Example Code

The Text example in this book's source code demonstrates placing text relative to a single piece of geometry. The code creates a simple scene graph consisting of a single **Geode.** The **Geode** contains four **Drawable**s, a Gouraud-shaded quadrilateral in the *xz* plane and three **Text** objects. Two of the **Text** objects are screen-oriented and label the top-left and top-right corners of the quadrilateral, while the third **Text** object lies in the *xz* plane just

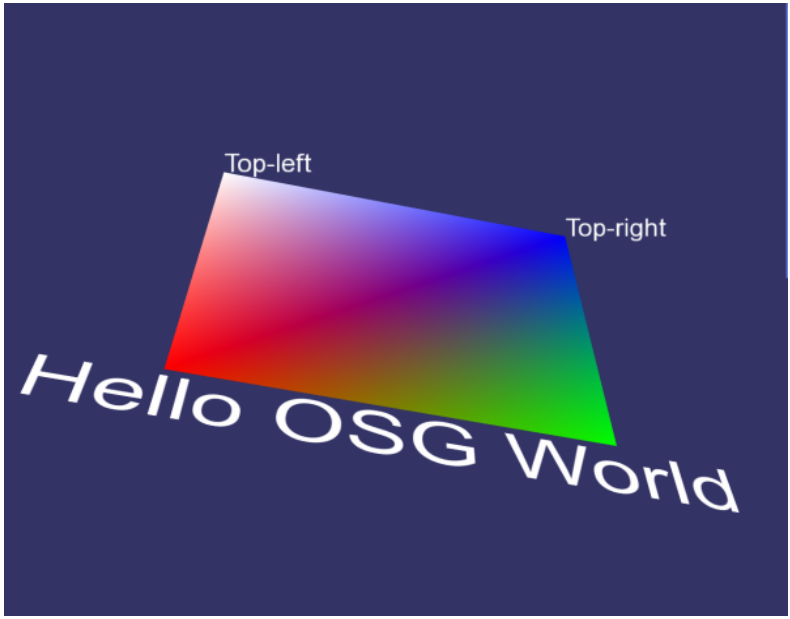


Figure 2-12
The Text example scene graph displayed in osgviewer
The Text example creates three Text objects with two different font resolutions and orientation styles (**SCREEN** and **XZ_PLANE**).

below the quadrilateral.

Like all the examples in this chapter, the Text example simply creates a scene graph and writes it as an .osg file. To view the scene graph, issue the following command:

```
osgviewer Text.osg
```

Figure 1.2 shows the scene graph displayed in osgviewer.

2.6.4 The .osg File Format

All NodeKits exist as two libraries. The first implements the NodeKit's primary functionality using new classes derived from **Node, Drawable,** or **StateAttribute.** Your application must link with this library in order to use these new classes. The second library is the dot osg wrapper that allows these new classes to load from and write to the .osg file format.

The .osg file format is plain ASCII text and not designed to be an efficient mechanism for storing and loading data. It's an excellent debugging tool, however. All the examples in this chapter write their scene graphs as .osg files. This is an unnecessary step in the rendering process—real applications create (or load) their scene graph and display it. The examples use .osg to illustrate a useful scene graph debugging technique.

During application development, you should write subgraphs or entire scene graphs as .osg files when you encounter unexpected rendering behavior. You can often determine the root cause of rendering errors by examining .osg files in a text editor. If you find something that doesn't look quite right, you can often edit the file manually to verify your suspicion.

Many OSG developers post to the osg-users email list when they encounter rendering problems. An .osg file containing a small subgraph that reproduces the problem, attached to such posts, greatly increases the chance that another reader can correctly diagnose the problem.

Listing 2-4 shows the Text example's .osg file output. It shows how the osg library stores its **Geode** and **Geometry** classes, as well as vertex and color data. It also illustrates how the osgText library's dot osg wrapper stores **Text** objects and their parameters.

The format is indented for readability. Child nodes or stored data are indented two spaces relative to their parent object, and curly braces wrap nesting levels.

Listing 2-4
The Text example scene graph .osg File
The Text example creates a scene graph consisting of a single **Geode** that contains four **Drawable**s, a **Geometry** and three **Text** objects.

```
Geode {
  DataVariance UNSPECIFIED
  nodeMask 0xffffffff
```

```
  cullingActive TRUE
  num_drawables 4
  Geometry {
    DataVariance UNSPECIFIED
    useDisplayList TRUE
    useVertexBufferObjects FALSE
    PrimitiveSets 1
    {
      DrawArrays QUADS 0 4
    }
    VertexArray Vec3Array 4
    {
      -1 0 -1
      1 0 -1
      1 0 1
      -1 0 1
    }
    NormalBinding OVERALL
    NormalArray Vec3Array 1
    {
      0 -1 0
    }
    ColorBinding PER_VERTEX
    ColorArray Vec4Array 4
    {
      1 0 0 1
      0 1 0 1
      0 0 1 1
      1 1 1 1
    }
  }
  osgText::Text {
    DataVariance UNSPECIFIED
    StateSet {
      UniqueID StateSet_0
      DataVariance UNSPECIFIED
      rendering_hint TRANSPARENT_BIN
      renderBinMode USE
      binNumber 10
      binName DepthSortedBin
    }
    supportsDisplayList FALSE
    useDisplayList FALSE
    useVertexBufferObjects FALSE
    font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
    fontResolution 32 32
    characterSize 0.15 1
    characterSizeMode OBJECT_COORDS
    alignment LEFT_BASE_LINE
    autoRotateToScreen TRUE
    layout LEFT_TO_RIGHT
```

```
    position 1 0 1
    color 1 1 1 1
    drawMode 1
    text "Top-right"
  }
  osgText::Text {
    DataVariance UNSPECIFIED
    Use StateSet_0
    supportsDisplayList FALSE
    useDisplayList FALSE
    useVertexBufferObjects FALSE
    font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
    fontResolution 32 32
    characterSize 0.15 1
    characterSizeMode OBJECT_COORDS
    alignment LEFT_BASE_LINE
    autoRotateToScreen TRUE
    layout LEFT_TO_RIGHT
    position -1 0 1
    color 1 1 1 1
    drawMode 1
    text "Top-left"
  }
  osgText::Text {
    DataVariance UNSPECIFIED
    Use StateSet_0
    supportsDisplayList FALSE
    useDisplayList FALSE
    useVertexBufferObjects FALSE
    font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
    fontResolution 128 128
    characterSize 0.4 1
    characterSizeMode OBJECT_COORDS
    alignment CENTER_TOP
    rotation 0.707107 0 0 0.707107
    layout LEFT_TO_RIGHT
    position 0 0 -1.04
    color 1 1 1 1
    drawMode 1
    text "Hello OSG World"
  }
}
```

A **Geode** appears at the top level of Listing 2-4. This file shows some parameters, such as *NodeMask* and *CullingActive that* are outside the scope of this book, followed by the *num_drawables* parameter, set to four. The four **Drawable**s are indented one level.

The first **Drawable** is the **Geometry** object that renders the quadrilateral. It contains all the parameters and data that the Text example specifies, as well as some additional parameters required by **Geometry.**

The three **Text** objects follow the **Geometry** object. The first two have an *autoRotateToScreen* parameter set to TRUE, which causes them to always face the screen. The third Text object contains a *rotation* parameter followed by four **Quat** values that force the text into the *xz* plane. The **Text** objects contain other more familiar parameters as well, such as the color (white), position values, and font file name.

As an experiment, edit the *rgba* color value of one of the **Text** objects, save the file, and view it with osgviewer. The following line sets the **color** parameter to purple, for example.

```
color 0.6 0 1 1
```

This might appear to be a trivial change, but if you are debugging a lighting problem and you suspect the light diffuse color is too dark, editing the .osg file and to brighten the light diffuse color is a quick and easy way to test your suspicion.

The first **Text** object contains a **StateSet.** The StateSet parameters are outside the scope of this book, but they essentially specify that OSG should render the **Text** object last and in back-to-front ordering for proper transparency and blending. (Internally, Text enables blending as it renders.) The other two **Text** objects don't appear to contain a **StateSet** because the osgText library shares **StateSet**s between **Text** objects to conserve memory. If you examine the other two Text objects, you'll see that they contain the following line:

```
Use StateSet_0
```

When OSG loads the .osg file, the *Use* parameter indicates data sharing. In this case, it tells OSG that the last two **Text** objects share the **StateSet** identified by *StateSet_0* with the first **Text** object.

As an OSG developer, you should become familiar with the .osg file format. Take the time to examine each of the .osg files created by the examples in this chapter. You might not understand all of the parameters, but you should at least understand the structure and how it corresponds to the code that creates the scene graph.

3 Using OpenSceneGraph in Your Application

Real applications need to do more than build a scene graph and write it out to a file. This final chapter explores techniques for integrating OSG into your application. You'll learn how to render a scene graph, change the view, perform picking operations, and dynamically modify scene graph data.

3.1 Rendering

OSG doesn't hide any functionality. OSG exposes the lowest levels of its functionality to your application. If you want complete control over rendering a scene graph, you can write code to perform the following operations in your application.

- Develop your own view management code to modify the OpenGL model-view matrix.
- Create a window and an OpenGL context and make them current. Write the code to manage multiple windows and contexts, if your application requires it.
- If your application uses paged databases, start the **osgDB::DatabasePager.**
- Instantiate **osgUtil::UpdateVisitor, osgUtil::CullVisitor,** and **osgUtil::RenderStage** objects to implement the update, cull, and draw traversals. If you really want total control, design your own classes to perform these traversals.
- Write a main loop that handles events from the operating system. Call into your view code to update the model-view matrix.
- Call **glClear()** before rendering a frame. Execute update, cull, and draw traversals to render, then swap buffers.

- Write additional code to support stereo and multipipe rendering if your application or target platform requires it.
- Finally, write all this code in a platform-independent manner so that it works with all target platforms.

This is possible to do but tedious and time consuming. It's also potentially incompatible with future versions of OSG that might modify some of the low-level interfaces used by such an application.

Fortunately, OSG has evolved over time to increasingly incorporate functionality that makes it easier for applications to render. As you work with OSG, you might encounter some of these utilities and libraries.

- **osgUtil::SceneView**—A class that wraps the update, cull, and draw traversals, but doesn't start the **DatabasePager.** There are several applications that use **SceneView** as their main interface for rendering in OSG.
- Producer and osgProducer—Producer is an external camera library that supports multipipe rendering. osgProducer is a library that unifies OSG and Producer for application usage. Producer has an active user base, and there are many Producer- and osgProducer-based OSG applications today.

OSG v1.3 adds a new library to the core OSG libraries—the osgViewer library. osgViewer contains a set of viewer classes that encapsulate a large amount of functionality commonly required by applications, such as display management, event handling, and rendering. It uses the **osg::Camera** class to manipulate the OpenGL model-view matrix. Unlike the **SceneView** class, the osgViewer library's viewer classes provide full support for the **DatabasePager.** osgViewer also simplifies support for multiple independent views into the same scene graph.

Section **1.6.3 Components** provides an overview of the osgViewer library's three viewer classes, **SimpleViewer, Viewer,** and **CompositeViewer.** This chapter demonstrates how the Viewer class can be used by your application to implement OSG rendering functionality.

3.1.1 The Viewer Class

The Viewer example in this book's source code demonstrates the minimal code required to render OSG in an application. Viewer instantiates an **osgViewer::Viewer** object, attaches a scene graph to it, and allows it to render. The source code is effectively three lines long, as shown in Listing 3-1.

Listing 3-1
The Viewer example
This code demonstrates the minimum code for rendering OSG in your application.

```
#include <osgViewer/Viewer>
```

```
#include <osgDB/ReadFile>

int main( int, char ** )
{
    osgViewer::Viewer viewer;
    viewer.setSceneData( osgDB::readNodeFile( "cow.osg" ) );
    return viewer.run();
}
```

The Viewer example might remind you of executing the following osgviewer command from section **1.3 Running osgviewer.**

```
osgviewer cow.osg
```

The similarity is no coincidence. Under the hood, the osgviewer application uses **Viewer** for its rendering. osgviewer configures its Viewer for additional functionality, however, just as you will certainly want your application to do more than the code in Listing 3-1.

Changing the View

Under the hood, **Viewer** creates an **osg::Camera** object to manage the OpenGL model-view matrix. There are two ways you can control the **Camera.**

- Attach a camera manipulator to the **Viewer.** If your application doesn't do this, **Viewer::run()** creates an **osgGA::TrackballManipulator** to control the **Camera.** The osgGA library defines several manipulators that you can use. Call **Viewer::setCameraManipulator()** to specify your own manipulator.
- Set the **Camera**'s projection and view matrices to matrices that you define. This gives your application complete control over the view.

If you choose to set the Camera matrices directly, using Viewer::run() is impractical because it doesn't allow view changes per frame. Instead, you'll need to code a small loop that iteratively updates the view and renders a frame. Listing 3-2 shows an example of how to do this.

Listing 3-2
Direct view control

This code snippet demonstrates how to control **Viewer**'s **Camera** object to change the view each frame.

```
osgViewer::Viewer viewer;

viewer.setSceneData( osgDB::readNodeFile( "cow.osg" ) );

viewer.getCamera()->setProjectionMatrixAsPerspective(
        40., 1., 1., 100. );

// Create a matrix to specify a distance from the viewpoint.
```

```
osg::Matrix trans;
trans.makeTranslate( 0., 0., -12. );

// Rotation angle (in radians)
double angle( 0. );
while (!viewer.done())
{
    // Create the rotation matrix.
    osg::Matrix rot;
    rot.makeRotate( angle, osg::Vec3( 1., 0., 0. ) );
    angle += 0.01;

    // Set the view matrix (the concatenation of the rotation and
    //   translation matrices).
    viewer.getCamera()->setViewMatrix( rot * trans );

    // Draw the next frame.
    viewer.frame();
}
```

The code in Listing 3-2 sets the **Camera**'s projection matrix once, outside the rendering loop. Camera provides several methods for specifying the projection matrix, which should look familiar to most OpenGL programmers.

```
void setProjectionMatrix( const osg::Matrix& matrix );
void setProjectionMatrixAsOrtho( double left, double right,
        double bottom, double top, double zNear, double zFar );
void setProjectionMatrixAsOrtho2D( double left, double right,
        double bottom, double top );
void setProjectionMatrixAsFrustum( double left, double right,
        double bottom, double top, double zNear, double zFar );
void setProjectionMatrixAsPerspective( double fovy,
        double aspectRatio, double zNear, double zFar );
```

Camera::setProjectionMatrix() takes an osg::Matrix as a parameter and is analogous to the following sequence of OpenGL commands:

```
glMatrixMode( GL_PROJECTION );
glLoadMatrixf( m );
```

Camera::setProjectionMatrixAsOrtho() creates a projection matrix using an algorithm identical to the **glOrtho()** OpenGL command, while the **setProjectionMatrixAsOrtho2D()** method is more analogous to the GLU entry point **gluOrtho2D(). Camera** also provides methods for setting a perspective projection analogous to the **glFrustum()** and **gluPerspective()** commands.

Inside the rendering loop, the code updates the **Camera**'s view matrix each frame to increment the rotation angle. Again, Camera provides several entry points that should be familiar to OpenGL developers. The code in Listing 3-2 sets the view matrix explicitly with

the **setViewMatrix()** method, and **Camera** also supplies the **setViewMatrixAsLookat()** method that takes parameters similar to the **gluLookAt()** entry point.

Setting the Clear Color

The **Viewer**'s **Camera** object provides interfaces for several operations besides setting the view. Your application uses **Camera** to set the clear color. The following code shows how to set the clear color to black.

```
viewer.getCamera()->setClearColor( osg::Vec4( 0., 0., 0., 1. ) );
```

By default, Camera clears the depth and color buffers. To change this default behavior, use the **Camera::setClearMask()** method and pass in the appropriate OpenGL buffer flags.

```
viewer.getCamera()->setClearMask(GL_COLOR_BUFFER_BIT |
       GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
```

The above code snippet clears the color, depth, and stencil buffers.

3.1.2 SimpleViewer and CompositeViewer

The osgViewer supplies two additional viewer classes that are outside the scope of this book. This section describes them only at a high level.

Viewer provides extensive functionality and is designed to simplify the process of writing OSG applications from scratch. If you're porting an existing application to OSG, you need a simpler viewer class that functions within your existing framework. The **SimpleViewer** class was designed for this purpose. Unlike **Viewer, SimpleViewer** doesn't automatically create a window or graphics context. Instead, it depends on your application to create a window and context and make them current. If you do this correctly, **SimpleViewer::frame()** will render into your application window.

While **Viewer** manages a single view into a scene (possibly with a group of **Camera**s to support multipipe rendering), **CompositeViewer** supports multiple views into one or more scenes and allows your application to specify their rendering order. **CompositeViewer** supports render-to-texture (RTT) operations, which allows your application to use the rendered image from one view as a texture map in a subsequent view.

For the latest information on **SimpleViewer** and **CompositeViewer,** see the OSG Wiki Web site [OSGWiki].

3.2 Dynamic Modification

OSG allows you to dynamically modify the scene graph so that it changes each frame. This capability is a requirement of any interactive graphics application. You can modify the geometric data, state parameters, **Switch** node settings, or even the structure of the scene graph itself.

As chapter 1 explains, the cull traversal stores references to geometry and state in a render graph for processing during the draw traversal. The osgViewer library supports many threading models, one of which runs the cull and draw traversals in a separate thread. For optimum performance, OSG doesn't impose locks for thread safety. Instead, it requires that applications only modify the scene graph outside the cull and draw traversals.

There are a few ways to ensure that your modifications don't collide with the cull and draw thread. One simple solution— modifying the scene graph outside of the **Viewer::frame()** call—requires additional code within the main rendering loop. If you desire a more elegant solution, you should perform modifications during the update traversal.

This section covers some basic topics related to dynamic scene graph modification.

- For optimum performance and thread safety, you need to tell OSG which parts of the scene graph you intend to modify. You do this by setting the *data variance* for any **Object** (**Node, Drawable, StateSet,** and so on).
- OSG allows applications to assign callbacks to **Node**s and **Drawable**s. OSG executes these callbacks during specific traversals. To modify a **Node** or **Drawable** during the update traversal, applications set an update callback.
- Applications don't always know in advance which parts of a scene graph they'll modify. They might need to search a scene graph to find the node of interest, or they might allow users to pick a node using the mouse or other input mechanism.

The following text described each of these topics.

3.2.1 Data Variance

The osgViewer library supports a threading model that allows the application main loop to continue before the draw traversal completes. This means **Viewer::frame()** can return while the draw traversal is still active. It also means that the draw traversal from the previous frame could overlap with the update traversal of the next frame. When you consider the implications of this threading model, it might seem impossible to avoid

Cause of the Crash

As you develop code to dynamically modify the scene graph, you might encounter an application crash or segmentation violation that occurs during the scene graph modification. Such crashes are almost always caused by modifying the scene graph during the cull and draw traversals.

colliding with the draw traversal thread. However, OSG supplies a solution with the **osg::Object::DataVariance()** method.

To set an **Object**'s data variance, call **setDataVariance()** with one of the `Object::DataVariance` enumerant values. Initially, data variance is `UNSPECIFIED.` Your application should change the data variance to either `STATIC` or `DYNAMIC.`

OSG will ensure that the draw traversal returns only after all `DYNAMIC` nodes and data have been processed. The draw traversal may still be processing the render graph even after it has returned, but only `STATIC` data remains in the render graph at that point. If your scene graph contains very little data marked as `DYNAMIC,` the draw traversal will return very quickly, freeing your application for other tasks.

3.2.2 Callbacks

OSG allows you to assign callbacks to **Node**s and **Drawable**s. **Node**s can have callbacks that OSG executes during the update and cull traversals, while **Drawable**s can have callbacks that OSG executes during the cull and draw traversals. This section describes how to dynamically modify a **Node** during the update traversal using an **osg::NodeCallback.** OSG's callback interface is based on the Callback design pattern [Gamma95].

To use a **NodeCallback,** your application should perform the following steps.

- Derive a new class from **NodeCallback.**
- Override the **NodeCallback::operator()** method. Code this method to perform the dynamic modification on your scene graph.
- Instantiate your new class derived from **NodeCallback** and attach it to the **Node** you want to modify using the **Node::setUpdateCallback()** method.

OSG will call the **operator()** method in your derived class during each update traversal, allowing your application to modify the **Node.**

OSG passes two parameters to your **operator()** method. The first parameter is the address of the **Node** you attached the callback to. This is the Node that your callback will dynamically modify within the **operator()** method. The second parameter is the address of an **osg::NodeVisitor.** The next section describes the **NodeVisitor** class, and for now you can ignore it.

To attach your **NodeCallback** to a **Node,** use the **Node::setUpdateCallback()** method. **setUpdateCallback()** takes one parameter, the address of a class derived from **NodeCallback.** The following code segment shows how to attach a **NodeCallback** to a node.

```
class RotateCB : public osg::NodeCallback
{
    ...
};
```

```
...
    node->setUpdateCallback( new RotateCB );
```

Callbacks can be shared by multiple nodes. **NodeCallback** derives (indirectly) from **Referenced,** and **Node** keeps a **ref_ptr<>** to its update callback. When the last node referencing a callback is deleted, the **NodeCallback**'s reference count drops to zero and it is also deleted. In the code above, your application doesn't keep a pointer to the RotateCB object and doesn't need to.

The books example code contains a Callback example that demonstrates the use of update callbacks. The code attaches a cow to two **MatrixTransform** nodes. The code derives a class from **NodeCallback** and attaches it to one of the two **MatrixTransform** objects. During the update traversal, the new **NodeCallback** modifies the matrix to rotate one of the cows. The output of the callback example is shown in Figure 3-1.

The example code, shown in Listing 3-3, consists of three main parts. The first part defines a class called RotateCB, which derives from **NodeCallback.** The second part is a function called createScene(), which creates the scene graph. Note that when this function creates the first **MatrixTransform** object, called *mtLeft,* it assigns an update callback to *mtLeft* with the function call `mtLeft->setUpdateCallback( new RotateCB ).` If you were to comment this line out and run the example, the cow wouldn't rotate. The final part of the

Figure 3-1
Dynamic modification using an update callback
This figure shows the output of the Callback example program. The code dynamically rotates the cow on the left around its vertical axis while the cow on the right remains unmodified.

example is the main() entry point that creates a viewer and renders.

Listing 3-3
The Callback Example Source Code

This example demonstrates the process of creating a **NodeCallback** to update the scene graph during the update traversal.

```
#include <osgViewer/Viewer>
#include <osgGA/TrackballManipulator>
#include <osg/NodeCallback>
#include <osg/Camera>
#include <osg/Group>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>

// Derive a class from NodeCallback to manipulate a
//   MatrixTransform object's matrix.
class RotateCB : public osg::NodeCallback
{
public:
    RotateCB() : _angle( 0. ) {}

    virtual void operator()( osg::Node* node,
            osg::NodeVisitor* nv )
    {
        // Normally, check to make sure we have an update
        //   visitor, not necessary in this simple example.
        osg::MatrixTransform* mtLeft =
                dynamic_cast<osg::MatrixTransform*>( node );
        osg::Matrix mR, mT;
        mT.makeTranslate( -6., 0., 0. );
        mR.makeRotate( _angle, osg::Vec3( 0., 0., 1. ) );
        mtLeft->setMatrix( mR * mT );

        // Increment the angle for the next from.
        _angle += 0.01;

        // Continue traversing so that OSG can process
        //   any other nodes with callbacks.
        traverse( node, nv );
    }

protected:
    double _angle;
};

// Create the scene graph. This is a Group root node with two
//   MatrixTransform children, which multiply parent a single
//   Geode loaded from the cow.osg model file.
osg::ref_ptr<osg::Node>
```

```
createScene()
{
    // Load the cow model.
    osg::Node* cow = osgDB::readNodeFile( "cow.osg" );
    // Data variance is STATIC because we won't modify it.
    cow->setDataVariance( osg::Object::STATIC );

    // Create a MatrixTransform to display the cow on the left.
    osg::ref_ptr<osg::MatrixTransform> mtLeft =
            new osg::MatrixTransform;
    mtLeft->setName( "Left Cow\nDYNAMIC" );
    // Set data variance to DYNAMIC to let OSG know that we
    //   will modify this node during the update traversal.
    mtLeft->setDataVariance( osg::Object::DYNAMIC );
    // Set the update callback.
    mtLeft->setUpdateCallback( new RotateCB );
    osg::Matrix m;
    m.makeTranslate( -6.f, 0.f, 0.f );
    mtLeft->setMatrix( m );
    mtLeft->addChild( cow );

    // Create a MatrixTransform to display the cow on the right.
    osg::ref_ptr<osg::MatrixTransform> mtRight =
            new osg::MatrixTransform;
    mtRight->setName( "Right Cow\nSTATIC" );
    // Data variance is STATIC because we won't modify it.
    mtRight->setDataVariance( osg::Object::STATIC );
    m.makeTranslate( 6.f, 0.f, 0.f );
    mtRight->setMatrix( m );
    mtRight->addChild( cow );

    // Create the Group root node.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->setName( "Root Node" );
    // Data variance is STATIC because we won't modify it.
    root->setDataVariance( osg::Object::STATIC );
    root->addChild( mtLeft.get() );
    root->addChild( mtRight.get() );

    return root.get();
}

int main(int, char **)
{
    // Create the viewer and set its scene data to our scene
    //   graph created above.
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );

    // Set the clear color to something other than chalky blue.
    viewer.getCamera()->setClearColor(
```

```
            osg::Vec4( 1., 1., 1., 1. ) );

    // Loop and render. OSG calls RotateCB::operator()
    //   during the update traversal.
    viewer.run();
}
```

RotateCB::operator() contains a call to traverse(). This is a member method of the **osg::NodeVisitor** class. If the scene graph contained more than one update callback, this call would allow OSG to visit other nodes and execute their callbacks. The following section discusses the **NodeVisitor** class in more detail.

The scene graph that the Callback example creates is shown in Figure 3-2. The **Group** root node has two child **MatrixTransform** nodes that transform the single cow **Geode** to two different locations. As the figure shows, one of the two **MatrixTransform** objects has its data variance set to `DYNAMIC`, and the other uses `STATIC` data variance because the code never modifies it. The **MatrixTransform** on the left has the update callback attached to it, which dynamically modifies the matrix during the update traversal.

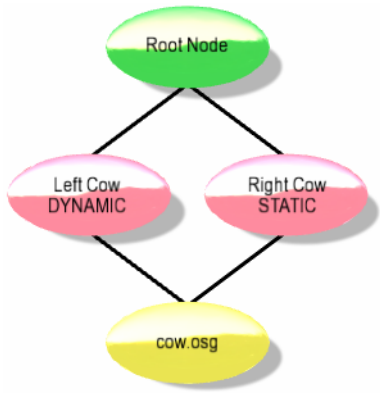


Figure 3-2
The Callback example program scene graph
This figure shows the Callback example program's scene graph hierarchy. Note the two **MatrixTransform** nodes have different data variance.

As this example illustrates, dynamically modifying a node is straightforward because attaching an update callback to a known node is trivial. The problem becomes more complex if your application modifies nodes deep within a scene graph loaded from a model file, or selected interactively by a user. The next sections describe some methods in OSG for runtime node identification.

3.2.3 NodeVisitors

NodeVisitor is OSG's implementation of the Visitor design pattern [Gamma95]. In essence, **NodeVisitor** traverses a scene graph and calls a function for each visited node. This simple technique exists as a base class for many OSG operations, including the **osgUtil::Optimizer,** the osgUtil library's geometry processing classes, and file output. OSG uses the **osgUtil::UpdateVisitor** class (derived from **NodeVisitor**) to perform the update traversal. In the preceding section, **UpdateVisitor** is the **NodeVisitor** that calls the **NodeCallback::operator()** method. In summary, **NodeVisitor**s are used throughout OSG.

NodeVisitor is a base class that your application never instantiates directly. Your application can use any **NodeVisitor** supplied by OSG, and you can code your own class derived from **NodeVisitor. NodeVisitor** consists of several **apply()** methods overloaded for most major OSG node types. When a **NodeVisitor** traverses a scene graph, it calls the appropriate **apply()** method for each node that it visits.

After loading a scene graph from file, applications commonly search the loaded scene graph for nodes of interest. As an example, imagine a model of a robot arm containing articulations modeled with transformations at the joints. After loading this file from disk, an application might use a **NodeVisitor** to locate all the **Transform** nodes to enable animation. In this case, the application uses a custom **NodeVisitor** and overrides the **apply(osg::Transform&)** method. As this custom **NodeVisitor** traverses the scene graph, the **apply()** method is called for each node that derives from **Transform,** and the application can perform the operations necessary to enable animation on that node, such as saving the node address away in a list.

Searching for a node with a specific name is a simple and often useful operation. The code in Listing 3-4 shows how to implement a class called FindNamedNode. This class takes a string as a parameter to its constructor, and stores the address of a node with a matching name.

Listing 3-4
Definition of the FindNamedNode class

This listing shows the declaration and definition of a simple NodeVisitor that finds a node with a specified name. The class, FindNamedNode, is part of the FindNode example program.

```
// Derive a class from NodeVisitor to find a node with a
//   specific name.
class FindNamedNode : public osg::NodeVisitor
```

Enable NodeVisitor Traversals

The **NodeVisitor** base class, by default, disables traversal. In your derived class, you should initialize the base class with the enumerant `NodeVisotor::TRAVERSE_ALL_CHILDREN` to enable traversal. Otherwise, OSG won't call any of your **apply()** methods.

```
{
public:
    FindNamedNode( const std::string& name )
      : osg::NodeVisitor( // Traverse all children.
                osg::NodeVisitor::TRAVERSE_ALL_CHILDREN ),
        _name( name ) {}

    // This method gets called for every node in the scene
    //   graph. Check each node to see if its name matches
    //   out target. If so, save the node's address.
    virtual void apply( osg::Node& node )
    {
        if (node.getName() == _name)
            _node = &node;

        // Keep traversing the rest of the scene graph.
        traverse( node );
    }

    osg::Node* getNode() { return _node.get(); }

protected:
    std::string _name;
    osg::ref_ptr<osg::Node> _node;
};
```

To traverse your scene graph with a **NodeVisitor,** pass the **NodeVisitor** as a parameter to **Node::accept().** You can call **accept()** on any node, and the NodeVisitor will traverse the scene graph starting at that node. To search an entire scene graph, call **accept()** on the root node.

The FindNamedNode class in Listing 3-4 is part of the FindNode example program. The FindNode example loads a scene graph from disk, finds a node with a specific name in the loaded scene graph, and modifies that node's data before rendering it. The FindNode example works with the State example discussed in section **2.4.3 Example Code for Setting State,** which outputs a scene graph to a file. When FindNode loads this file, it finds the **MatrixTransform** node who's **StateSet** is configured for flat shading, and changes the state to smooth shading.

3.2.4 Picking

Most 3D applications require some form of picking functionality to allow the end user to select a portion of the displayed image. In its simplest form, the user positions the mouse over the displayed scene and clicks a mouse button. Internally, the application performs an operation to map the 2D *xy* mouse location to the corresponding scene graph node and stores the node address for future operations.

In essence, OSG-based applications perform two operations to implement picking.

- Receive mouse events. The osgGA library provides event classes that allow applications to receive mouse events in a platform-independent manner.
- Determine which part of the scene graph is under the mouse cursor. The osgUtil library provides intersection classes that create a volume around the mouse *xy* location and allow you to intersect that volume with your scene graph. osgUtil returns a list of nodes intersected by the volume sorted in front-to-back order.

This section describes how to implement both of these operations.

Capturing Mouse Events

As section **1.6.3 Components** states, the osgGA library provides platform-independent GUI event support. The examples in this chapter use **osgGA::TrackballManipulator** for manipulating the view matrix. **TrackballManipulator** takes mouse events as input and modifies the viewer's **osg::Camera** view matrix.

TrackballManipulator derives from **osgGA::GUIEventHandler. GUIEventHandler** is a virtual base class that your application doesn't instantiate directly. Instead, applications derive classes from **GUIEventHandler** to perform operations based on GUI events. To perform mouse-based picking, derive a class from **GUIEventHandler** and override the **GUIEventHandler::handle()** method to receive mouse events. Create an instance of your new class and attach it to your application's viewer.

The **handle()** method takes two parameters, an **osgGA::GUIEventAdapter** and an **osgGA::GUIActionAdapter,** as shown below.

```
virtual bool GUIEventHandler::handle(
        const osgGA::GUIEventAdapter& ea,
            osgGA::GUIActionAdapter& aa );
```

Your implementation of handle() receives GUI events including mouse events in the **GUIEventAdapter.** The GUIEventAdapter header file declares an `EventType` enumerant that allow your application to examine only events of interest, such as mouse events. Retrieve the event type with the **GUIEventAdapter::getEventType()** method.

GUIActionAdapter is your application's interface back to the GUI system. In the case of mouse picking, you attach your picking **GUIEventHandler** to your viewer class, so the **GUIActionAdapter** is your viewer class. You need this so you can perform an intersection with your scene based on the current view.

Before rendering with your viewer, create an instance of your new **GUIEventHandler** class and attach it to your viewer with the **Viewer::addEventHandler()** method. As its name implies, viewers can have multiple event handlers, and Viewer adds your event handler to a list of possibly several event handlers. **Viewer** calls **handle()** for each GUI event until one of the **handle()** methods returns true.

The code in Listing 3-5 contains a class called PickHandler that derives from **GUIEventHandler.** The implementation of the **handle()** method supports the `PUSH`, `MOVE`, and `RELEASE` mouse event types. It records the mouse xy location on `PUSH` and

MOVE events, and if the mouse doesn't move the RELEASE event triggers a pick operation. If the pick succeeds, **handle()** returns true. It returns false in all other cases to allow other event handlers to examine the event.

Listing 3-5
The PickHandler class

To implement picking in OSG, use a subclass derived from **osgGA::GUIEventHandler.** This listing shows the PickHandler class from the Picking example program. The class defines two methods, one for receiving mouse events, and the other for implementing the pick operation on mouse release.

```
// PickHandler -- A GUIEventHandler that implements picking.
class PickHandler : public osgGA::GUIEventHandler
{
public:

    PickHandler() : _mX( 0. ),_mY( 0. ) {}
    bool handle( const osgGA::GUIEventAdapter& ea,
            osgGA::GUIActionAdapter& aa )
    {
        osgViewer::Viewer* viewer =
                dynamic_cast<osgViewer::Viewer*>( &aa );
        if (!viewer)
            return false;

        switch( ea.getEventType() )
        {
            case osgGA::GUIEventAdapter::PUSH:
            case osgGA::GUIEventAdapter::MOVE:
            {
                // Record mouse location for the button press
                //   and move events.
                _mX = ea.getX();
                _mY = ea.getY();
                return false;
            }
            case osgGA::GUIEventAdapter::RELEASE:
            {
                // If the mouse hasn't moved since the last
                //   button press or move event, perform a
                //   pick. (Otherwise, the trackball
                //   manipulator will handle it.)
                if (_mX == ea.getX() && _mY == ea.getY())
                {
                    if (pick( ea.getXnormalized(),
                                ea.getYnormalized(), viewer ))
                        return true;
                }
                return false;
            }
```

```
            default:
                return false;
        }
    }

protected:
    // Store mouse xy location for button press & move events.
    float _mX,_mY;

    // Perform a pick operation.
    bool pick( const double x, const double y,
            osgViewer::Viewer* viewer )
    {
        if (!viewer->getSceneData())
            // Nothing to pick.
            return false;

        double w( .05 ), h( .05 );
        osgUtil::PolytopeIntersector* picker =
                new osgUtil::PolytopeIntersector(
                    osgUtil::Intersector::PROJECTION,
                        x-w, y-h, x+w, y+h );

        osgUtil::IntersectionVisitor iv( picker );
        viewer->getCamera()->accept( iv );

        if (picker->containsIntersections())
        {
            osg::NodePath& nodePath =
                    picker->getFirstIntersection().nodePath;
            unsigned int idx = nodePath.size();
            while (idx--)
            {
                // Find the LAST MatrixTransform in the node
                //   path; this will be the MatrixTransform
                //   to attach our callback to.
                osg::MatrixTransform* mt =
                        dynamic_cast<osg::MatrixTransform*>(
                            nodePath[ idx ] );
                if (mt == NULL)
                    continue;

                // If we get here, we just found a
                //   MatrixTransform in the nodePath.

                if (_selectedNode.valid())
                    // Clear the previous selected node's
                    //   callback to make it stop spinning.
                    _selectedNode->setUpdateCallback( NULL );
```

```
                _selectedNode = mt;
                _selectedNode->setUpdateCallback( new RotateCB );
                break;
            }
            if (!_selectedNode.valid())
                osg::notify() << "Pick failed." << std::endl;
        }
        else if (_selectedNode.valid())
        {
            _selectedNode->setUpdateCallback( NULL );
            _selectedNode = NULL;
        }
        return _selectedNode.valid();
    }
};

int main( int argc, char **argv )
{
    // create the view of the scene.
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );

    // add the pick handler
    viewer.addEventHandler( new PickHandler );

    return viewer.run();
}
```

Listing 3-5 also shows the main() function of the Picking example program to illustrate using the **Viewer::addEventHandler()** method to attach an event handler to a viewer.

In summary, to receive mouse events for picking, perform the following steps:

- Derive a class from GUIEventHandler. Override the handle() method.
- In **handle(),** examine the event type in the GUIEventAdapter parameter to select events of interest and perform any necessary operations. Return true to prevent other event handlers from receiving an event.
- Before rendering, create an instance of your event handler class and add it to your viewer with the **addEventHandler()** method. OSG passes your viewer to your handle() method as the GUIActionAdapter parameter.

These techniques aren't limited to mouse-based picking. Your application can implement classes similar to **TrackballManipulator** by receiving mouse events in the same manner. You can also receive keyboard events and implement operations in responses to key presses.

The following section completes the discussion of mouse-based picking by describing how to determine which part of your scene graph is under a user mouse press.

Intersections

Think of Mouse picking as shooting a ray from the mouse position into your scene. The part of the scene under the mouse has an intersection with that ray. Ray intersections don't meet application picking requirements when the scene consists of line and point primitives because mouse location round off prohibits exact mathematical intersection with such primitives. Furthermore, in a typical perspective rendering, ray intersection precision is inversely proportional to distance from the viewer.

Instead of a ray, OSG intersects with a pyramid volume called a polytope to overcome both issues. The pyramid has its apex at the viewpoint, and its central axis passes directly through the mouse location. It widens away from the viewpoint as a function of the field of view and application-controlled width parameters.

OSG employs the inherent hierarchical nature of the scene graph to efficiently compute intersections on the host CPU, avoiding OpenGL's often-sluggish selection feature. The **osgUtil::IntersectionVisitor** class derives from **NodeVisitor** and tests each **Node**'s bounding volume against the intersection volume, allowing it to skip subgraph traversals if the subgraph has no possibility of a child intersection.

IntersectionVisitor can be configured for intersection testing with several different geometric constructs including planes and line segments. Its constructor takes an **osgUtil::Intersector** as a parameter, which defines the pick geometry and performs the actual intersection testing. **Intersector** is a pure virtual base class that your application doesn't instantiate. The osgUtil library derives several classes from **Intersector** to represent different geometric constructs, including the **osgUtil::PolytopeIntersector,** which is ideal for mouse-based picking.

Some applications require picking individual vertices or polygons. Other applications simply need to know the parent **Group** or **Transform** node containing any selected geometry. To meet these requirements, **IntersectionVisitor** returns an **osg::NodePath. NodePath** is a **std::vector<osg::Node>** that represents the path through the node hierarchy leading from the root node down to a leaf node. If your application requires an intermediate group node, search the **NodePath** from back to front until you find a node that meets your applications requirements.

In summary, to perform mouse-based picking in OSG, write your application to perform the following steps.

- Create and configure a **PolytopeIntersector** using the normalized mouse location stored in the **GUIEventAdapter.**
- Create an **IntersectionVisitor** and pass the **PolytopeIntersector** as a parameter in the constructor.
- Launch the IntersectionVisitor on your scene graph's root node, usually through the **Viewer**'s **Camera,** as in the following code:

```
// 'iv' is an IntersectionVisitor
viewer->getCamera()->accept( iv );
```

- If the **PolytopeIntersector** contains intersections, obtain the **NodePath** and search it to find the node of interest.

The PickHandler::pick() method in the Picking example program, shown in Listing 3-5, illustrates these steps. The Picking example program creates a scene graph similar to the scene graph created by the Callback program. However, the Picking scene graph uses a hierarchy of two **MatrixTransform** nodes, one to store a translation and the other to store a rotation. Upon a successful pick, the code searches the **NodePath** until it encounters the rotation **MatrixTransform.** It attaches an update callback to that node to dynamically rotate the child geometry.

When you run the Picking example program, it displays two cows, like the Callback example program. However, you can pick either cow, and the program rotates the selected cow in response.

Appendix: Where to Go From Here

Hopefully this book is a good introduction to OSG. As a quick start guide, however, this book isn't a complete OSG resource. This section describes additional sources of information that many OSG developers find useful.

Source Code

From a developer's perspective, the primary benefit of an open source product is that the source code is available. As you develop OSG-based application software, many issues you encounter can be resolved quickly and easily by stepping through the OSG source code to find out what's happening internally.

If you haven't done so already, download the full OSG source code, as described in section **1.2 Installing OSGOSG}** and create your own OSG binaries with debugging information. Building OSG for the first time can be confusing and time consuming, but this investment pays excellent dividends during the development phase of your application software.

The OSG source code distribution also comes with a rich collection of well-written and informative example programs that demonstrate correct usage of many OSG features not covered in this short book. These example programs are invaluable to anyone doing OSG development work.

It's possible to develop OSG applications using only the OSG binaries, but access to the source code distribution, examples, and debug binaries speeds the development process.

The OSG Wiki

The OSG Wiki Web site [OSGWiki] contains an enormous wealth of information pertaining to OSG, including the latest OSG news, tips for downloading, building, and installing, additional documentation contributed by members of the OSG community, example data, information on OSG community events, OSG-compatible components created by the OSG community, and support information.

```
http://www.openscenegraph.org/
```

The osg-users Email List

The osg-users email list puts you in touch with other OSG users and developers. If you run into an issue while trying to build OSG, can't figure out how to code a sticky problem, or have a question about some aspect of OSG's internal operation, a post to osg-users usually generates a handful of helpful replies. To subscribe to the osg-users email list, visit the following URL.

```
http://www.openscenegraph.net/mailman/listinfo/osg-users
```

Professional Support

As a testament to OSG's success, several companies provide OSG development, consulting, training, and documentation services. Rates, fees, and availability vary by company. For the most current information on services, post an inquiry to the osg-users email list, or visit the following URL.

```
http://www.openscenegraph.com/osgwiki/pmwiki.php/Support/Support
```

Glossary

.osg	This is an ASCII-based OSG native format, capable of storing all scene graph elements.
Data variance	An **osg::Object** property that specifies whether the application intends to dynamically modify the **Object**'s data or not. Set this property with the **Object::setDataVariance()** method, and pass in `Object::DYNAMIC` or `Object::STATIC`. See *Object*.
Data file path list	OSG searches this list of directories when your application attempts to read a 2D image or 3D model file using the osgDB interface.
Dot OSG wrapper	This is an OSG plugin library that allows NodeKits to perform file I/O on .osg files.
Drawable	The **osg::Drawable** class contains geometry to be rendered. Objects of type **Geode** (see *Geode*) contain lists of **Drawable**s in the scene graph. The render graph (see *render graph*) contains references to **Drawable**s.
Geode	The **osg::Geode** class is the OSG leaf node. **Geode**s have no children, but have a list of **osg::Drawable** objects (see *Drawable*) and may also have an **osg::StateSet** (see *StateSet*). The name is a combination of the words "geometry" and "node". See also *leaf node*. See **osg::Geode** in the Geode header.
Group	The **osg::Group** class supports the generic scene graph group node concept. It can serve as both a group node and a root node. Many scene graph classes derive from **osg::Group** to support multiple children. See also *group node*.
Group node	Group nodes have children. The group node has one or more parents, unless it is the root node (see *root node*).
Leaf node	Scene graph nodes that have no children. In most scene graphs, leaf nodes contain rendering data, such as geometry.

Library GPL	Formally known as the GNU Lesser General Public License. This less-restrictive form of the GNU General Public License is the basis of OSG's licensing.
Multipipe rendering	A parallel process that spreads the rendering workload over multiple graphics rendering cards or systems. In a typical multipipe scenario, displays are arranged side-by-side or in an array (or wall), and each graphics card renders part of a scene to one display.
Node	The base class of all OSG nodes. See the **osg::Node** class in the Node header.
NodeKit	An OSG NodeKit is a module that enhances core OSG functionality by adding new scene graph Node classes.
NodeVisitor	A class that traverses a scene graph, performing operations on (or collecting data from) nodes encountered during the traversal. The **osg::NodeVisitor** class implements the Visitor design pattern [Gamma95].
Object	A pure virtual base class that defines some basic properties and methods common to **Node**s, **Drawable**s, **StateAttribute**s, **StateSet**s, and other OSG entities.
Picking	A common interaction between the user and a 3D graphics software application in which the user selects an object of interest from the rendered image, usually by positioning the mouse cursor over the object and clicking a button.
Plugin	An architecture that allows libraries or modules conforming to a standard interface to be dynamically loaded at runtime by compatible software; any library or module conforming to a plugin architecture. OSG employs a plugin architecture for 2D and 3D file support. Libraries conforming to the interface defined in **osgDB::ReaderWriter** are collectively known as the OSG plugins. Applications access OSG plugins through the osgDB library.
Positional State	Any state value containing a position that is affected by the current transformation matrix. Clip plane and Light source position are two examples of positional state.
Pseudoloader	An OSG plugin that provides additional functionality beyond simply loading a file. The trans pseudoloader, for example, places a **Transform** node above the root node of the loaded file.
Render graph	A collection of references to Drawables and StateSets. The cull traversal creates the render graph from the scene graph. The draw traversal sends geometry and state data from the render graph to the underlying graphics hardware for final display.

Rendering state	Internal variables that control geometry processing and rendering. OSG rendering state is composed of modes (Boolean feature variables such as lighting and fog that can be enabled or disabled) and attributes (variables that configure enabled features, such as fog color and blending equations).
Root node	The parent node of all nodes in a scene graph. By definition, the root node has no parent node.
Smart pointer	A C++ class that contains a pointer and maintains a reference count associated with the memory it points to. An instance of a smart pointer usually increments the reference count in the constructor and decrements it in the destructor. When the reference count reaches zero, the memory is deleted. In OSG, **ref_ptr<>** is a smart pointer.
StateSet	An OSG object that stores state values. They are associated with **Nodes** and **Drawables** and can be shared to improve efficiency. During the cull traversal, OSG sorts some Drawables by their **StateSet.**
Stripification	The process of converting a collection of individual triangles that implicitly share vertices into a more efficient collection of triangle strip primitives with explicit vertex sharing.
Viewer	An OSG class that manages one or more views into the scene. The Viewer class can also manage different render surfaces such as windows or frame buffer objects, camera manipulators for changing the view(s), and event handlers.

Bibliography

[ARB05] OpenGL ARB, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis, *OpenGL® Programming Manual, Fifth Edition*. Boston: Addison-Wesley, 2005.

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995.

[Martz06] Martz, Paul, *OpenGL® Distilled*. Boston: Addison-Wesley, 2006.

[OSGWiki] OpenSceneGraph Wiki Web site, `http://www.openscenegraph.org/`

[Rost06] Rost, Randi. *OpenGL® Shading Language, Second Edition*. Boston: Addison-Wesley, 2006.

Revision History

Date	Comments
7 April 2007	Initial revision.