Quick answers to common problems

OpenSceneGraph 3 Cookbook

Over 80 recipes to show advanced 3D programming techniques with the OpenSceneGraph API

Rui Wang Xuelei Qian

[PACKT] PUBLISHING open source* community experience distilled

OpenSceneGraph 3 Cookbook

Over 80 recipes to show advanced 3D programming techniques with the OpenSceneGraph API

Rui Wang
Xuelei Qian

BIRMINGHAM - MUMBAI

OpenSceneGraph 3 Cookbook

Copyright © 2012 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: March 2012

Production Reference: 1280212

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-688-4

`www.packtpub.com`

Cover Image by Asher Wishkerman (`wishkerman@hotmail.com`)

Credits

Authors
Rui Wang
Xuelei Qian

Reviewers
Vincent Bourdier
Torben Dannhauer
Chris 'Xenon' Hanson

Acquisition Editor
Usha Iyer

Lead Technical Editor
Meeta Rajani

Technical Editors
Kedar Bhat
Mehreen Shaikh

Project Coordinator
Alka Nayak

Proofreaders
Mario Cecere
Aaron Nash

Indexers
Monica Ajmera Mehta
Hemangini Bari
Tejal Daruwale

Graphics
Valentina D'Silva
Manu Joseph

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Authors

Rui Wang is a software engineer at Beijing Crystal Digital Technology Co. Ltd., and the manager of osgChina, the largest OSG discussion website in China. He is one of the most active members of the official OSG community, who contributes to the OSG project regularly. He translated Paul Martz's "*OpenSceneGraph Quick Start Guide*" into Chinese in 2008, and wrote his own Chinese book "*OpenSceneGraph Design and Implementation*" in 2009. And in 2010, he wrote the book "*OpenSceneGraph 3.0 Beginner's Guide*", which is published by Packt Publishing, along with Xuelei Qian. He is also a novel writer and guitar lover in his spare time.

> I'd like to express my deepest respect to Robert Osfield and the entire OpenSceneGraph community for their continuous contribution and support to this marvelous open source project. Many thanks to the Packt team for the great efforts to make another OpenSceneGraph book published. And last but not least, I'll extend my heartfelt gratitude to my family, for your timeless love and spiritual support.

Xuelei Qian received his Ph.D. degree in applied graphic computing from the University of Derby in 2005. From 2006 to 2008, he worked as a postdoctoral research fellow in the Dept. of Precision Instrument and Mechanology at Tsinghua University. Since 2008, he has been appointed by the School of Scientific Research and Development of Tsinghua University. He is also the Deputy Director of Overseas R&D Management Office of Tsinghua University and Deputy Secretary in General of University-Industry Cooperation Committee, Tsinghua University.

> I will dedicate this book to my family because of all the wonderful things they do for me and supporting me all the way.

About the Reviewers

Vincent Bourdier, a twenty-six years old developer, is a French 3D passionate. After self tuition in 3D modeling and programming, he went to the UTBM (University of Technology of Belfort Montbeliard) in 2003 and received an engineering degree in computer sciences, specializing in imagery, interaction, and virtual reality. A computer graphics and passionate about image processing, he remains curious about new technologies in domains such as AI, Cmake, Augmented reality, and others.

He has been working as a 3D developer at Global Vision Systems (Toulouse, France) since 2008. He is now technical leader on a 3D real-time engine using OpenSceneGraph.

Global Vision Systems (`http://www.global-vision-systems.com`) is a software developer and publisher, offering innovative Human Machine Interfaces for Aeronautics, Space, Plant, and Process supervision.

I would like to thank my parents for their encouragement even if they don't understand a word of my job, my employers for this opportunity to live my passions and to give me challenges to meet, and all the people from the OpenSceneGraph mailing list for their help and advices, especially Robert Osfield, the OpenSceneGraph fundator.

Torben Dannhauer was born in Germany in 1982 and has been working as a freelancer in software development since 2000. He has also provided web and e-mail hosting since 2005. From 2003 until 2009, he studied mechanical engineering, specializing in flight system dynamics and information technology at University of Technology in Munich.

In his term thesis, he developed an airport traffic simulation tool that was also his first experience with OpenGL with which he visualized the simulation results. In his diploma thesis, he analysed the usability of game engines to serve as visualization software for full flight simulators considering qualification requirements. During the thesis, he decided to start the development of a rudimentary but open source visualization framework, which is based on OpenSceneGraph: osgVisual (`www.osgvisual.org`).

osgVisual is designed to provide all necessary elements for implementing a visualization software with multiple rendering hosts, projected on curved screens with multiple video channels. After finishing his studies, he still developed the tool, initially as a freelancer, later on as a hobby; nevertheless osgVisual is still under construction and will be for long time.

During his studies, he also started in 2006, to develop software for Chondrometrics GmbH (`www.chondrometrics.com`), a company providing medical data processing. At the end of 2009, he moved to Ainring (Germany) which is located near to Salzburg (Austria) to join Chondometrics part time and do a Ph.D. in medical science at Paracelsus Medical University Salzburg (`www.pmu.ac.at`).

Torben is experienced in C/C++ with and without Qt. He programmed PHP for a long time and has expertise in Linux and Windows operating systems. Due to his web and e-mail hosting, he knows Postfix & Co in detail and speaks SQL. Last but not least, he develops OpenSceneGraph applications, mostly related to osgVisual.

In his spare time he loves rowing, mountain biking, and skiing with his partner and friends.

I would like to thank Prof. Dr. Holzapfel (Institute of Flight System Dynamics, University of Technology, Munich) for the initial funding of the osgVisual project. It is great to learn OpenSceneGraph and develop an open source application on a paid basis.

Chris 'Xenon' Hanson is co-founder of 3D and digital imaging companies, 3D Nature and AlphaPixel, and is a veteran of the OpenSceneGraph community. Chris has worked on the OpenGL SuperBible, OpenGL Distilled, and OpenCL in Action. His goal in life is to ski all seven continents and build giant intergalactic fighting robots.

I would like to thank my wife Mindy and son Rhys for putting up with the late nights and potato gun explosions.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Customizing OpenSceneGraph 7

Introduction 7
Checking out the latest version of OSG 8
Configuring CMake options 14
Building common plugins 18
Compiling and packaging OSG on different platforms 22
Compiling and using OSG on mobile devices 25
Compiling and using dynamic and static libraries 29
Generating the API documentation 30
Creating your own project using CMake 33

Chapter 2: Designing the Scene Graph 37

Introduction 37
Using smart and observer pointers 40
Sharing and cloning objects 43
Computing the world bounding box of any node 47
Creating a running car 51
Mirroring the scene graph 56
Designing a breadth-first node visitor 58
Implementing a background image node 61
Making your node always face the screen 64
Using draw callbacks to execute NVIDIA Cg functions 67
Implementing a compass node 74

Chapter 3: Editing Geometry Models 81
Introduction 81
Creating a polygon with borderlines 82
Extruding a 2D shape to 3D 86
Drawing a NURBS surface 89
Drawing a dynamic clock on the screen 96
Drawing a ribbon following a model 101
Selecting and highlighting a model 106
Selecting a triangle face of the model 110
Selecting a point on the model 114
Using vertex-displacement mapping in shaders 119
Using the draw instanced extension 124
Chapter 4: Manipulating the View 129
Introduction 129
Setting up views on multiple screens 130
Using slave cameras to simulate a power-wall 133
Using depth partition to display huge scenes 139
Implementing the radar map 143
Showing the top, front, and side views of a model 148
Manipulating the top, front, and side views 152
Following a moving model 155
Using manipulators to follow models 159
Designing a 2D camera manipulator 162
Manipulating the view with joysticks 166
Chapter 5: Animating Everything 171
Introduction 171
Opening and closing doors 172
Playing a movie in the 3D world 177
Designing scrolling text 180
Implementing morph geometry 183
Fading in and out 187
Animating a flight on fire 190
Dynamically lighting within shaders 194
Creating a simple Galaxian game 198
Building a skeleton system 206
Skinning a customized mesh 211
Letting the physics engine be 215

Chapter 6: Designing Creative Effects 227
Introduction 227
Using the bump mapping technique 229
Simulating the view-dependent shadow 233
Implementing transparency with multiple passes 237
Reading and displaying the depth buffer 241
Implementing the night vision effect 245
Implementing the depth-of-field effect 249
Designing a skybox with the cube map 257
Creating a simple water effect 262
Creating a piece of cloud 266
Customizing the state attribute 273
Chapter 7: Visualizing the World 279
Introduction 279
Preparing the VirtualPlanetBuilder (VPB) tool 280
Generating a small terrain database 283
Generating terrain database on the earth 287
Working with multiple imagery and elevation data 290
Patching an existing terrain database with newer data 293
Building NVTT support for device-independent generation 296
Using SSH to implement cluster generation 298
Loading and rendering terrain from the Internet 301
Chapter 8: Managing Massive Amounts of Data 305
Introduction 305
Merging geometry data 306
Compressing texture 310
Sharing scene objects 316
Configuring the database pager 321
Designing a simple culling strategy 324
Using occlusion query to cull objects 331
Managing scene objects with an octree algorithm 333
Rendering point cloud data with draw instancing 342
Speeding up the scene intersections 347
Chapter 9: Integrating with GUI 353
Introduction 353
Integrating OSG with Qt 354
Starting rendering loops in separate threads 359
Embedding Qt widgets into the scene 361

Embedding CEGUI elements into the scene 365
Using the osgWidget library 374
Using OSG components in GLUT 379
Running OSG examples on Android 383
Embedding OSG into web browsers 386
Designing the command buffer mechanism 392

Chapter 10: Miscellaneous Discussion in Depth

This chapter is not present in the book but is available as a free download at the following link: `http://www.packtpub.com/sites/default/files/downloads/6884_Chapter10.pdf`.

Introduction
Playing with the Delaunay triangulator
Understanding and using the pseudo loaders
Managing customized data with the metadata system
Designing customized serializers
Reflecting classes with serializers
Taking a photo of the scene
Designing customized intersectors
Implementing the depth peeling method
Using OSG in C# applications
Using osgSwig for language binding
Contributing your code to OSG
Playing with osgEarth: another way to visualize the world
Use osgEarth to display a VPB-generated database

Index 397

Preface

During the last 12 years, OpenSceneGraph, which is one of the best 3D graphics programming interfaces in the world, has grown up so rapidly that it has already become the industry's leading open source scene graph technology. The latest distribution, OpenSceneGraph 3.0, now runs on all Microsoft Windows platforms, Apple Mac OS X, iOS (including iPhone and iPad), GNU/Linux, Android, IRIX, Solaris, HP-UX, AIX, and FreeBSD operating systems, with the efforts of 464 contributors around the world, and over 5,000 developers of the osg-users mailing list/forum and diverse and growing communities.

In the year 2010, I wrote the book "*OpenSceneGraph 3.0 Beginner's Guide*" with the help of Dr. Xuelei Qian. It was published by Packt Publishing, and could help the readers gain an overview of scene graphs and the basic concepts in OpenSceneGraph. But one book is far less than enough, especially for those who want to continuously study this high-quality library in depth and play with some state-of-art techniques. So the book "*OpenSceneGraph 3 Cookbook*" comes onto the scene, with over 80 recipes demonstrating how to make use of some advanced API features and create programs for industrial demands.

In this book, we will work on different goals, which originate from actual projects and customer needs, and try to make use of the cutting-edge graphics techniques, or integrate with other famous and stable libraries to satisfy various multi-level and multi-aspect demands.

Some of the recipes are too long and too complicated to fit into any of the chapters, so they will only appear in the source code package, which can be downloaded from the Packt website, or the author's Github repository as described later.

What this book covers

Chapter 1, *Customizing OpenSceneGraph*, introduces some advance topics about the configuration and compilation, including build steps on mobile devices and the automatic generation of the API documentation.

Chapter 2, *Designing the Scene Graph*, explains the management of scene graph, as well as the implementation of user-node functionalities in various ways.

Chapter 3, *Editing Geometry Models*, shows how to create polygonal meshes and make them animated. It also introduces some solutions for modifying existing geometric properties.

Chapter 4, *Manipulating the View*, discusses the topics of multi-screen and multi-view rendering, and the camera manipulation using input devices such as the joysticks.

Chapter 5, *Animating Everything*, introduces almost all kinds of real-time animation implementations as well as the integration of physics engines.

Chapter 6, *Designing Creative Effects,* discusses some cutting-edge techniques about realistic rendering with GPU and shaders. It also demonstrates a common way to create post-processing framework for complicated scene effects.

Chapter 7, *Visualizing the World*, is a totally independent chapter that demonstrates the generation of landscape pieces and even the entire earth, using VirtualPlanetBuilder.

Chapter 8, *Managing Massive Amounts of Data*, shows some advanced ways to manage massive data in OpenSceneGraph applications, with the help of some modern hardware features such as occlusion query and draw instancing.

Chapter 9, *Integrating with GUI*, covers the integration of OpenSceneGraph and other graphics user interfaces (GUI), including 2D and 3D widgets, mobile programs, and web browsers.

Chapter 10, *Miscellaneous Discussion in Depth*, introduces a few complicated demands that a developer may face in actual situations, and provides in-depth solutions for them. This chapter is not present in the book but is available as a free download at the following link: `http://www.packtpub.com/sites/default/files/downloads/6884_Chapter10.pdf`.

What you need for this book

To use this book, you will need a graphic card with robust OpenGL support, with the latest OpenGL device driver installed from your graphics hardware vendor.

You will need to download OpenSceneGraph 3.0.1 from `http://www.openscenegraph.org`. Some recipes may require the latest developer version. The CMake utility is also necessary for compiling OpenSceneGraph and the source code of this book. You may download it from `http://www.cmake.org/`.

You will also need a working compiler which transforms C++ source code into executable files. Some recommended ones include: gcc (on Unices), XCode (on Mac OS X), and mingw32 and Visual Studio (on Windows).

Who this book is for

This book is intended for software developers, researchers, and students who are already familiar with the basic concepts of OpenSceneGraph and can write simple programs with it. A basic knowledge of C++ programming is also expected. Some experience of using and integrating platform-independent APIs is also useful, but is not required.

General real-time computer graphics knowledge would be sufficient. Some familiarity with 3D vectors, quaternion numbers, and matrix transformations is helpful.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "For Debian and Ubuntu users, make sure you have the root permission and type the command `apt-get` in the terminal as shown in the following command line."

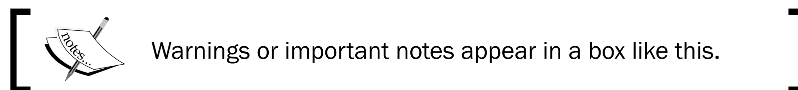
A block of code is set as follows:

```
// Create the text and place it in an HUD camera
osgText::Text* text = osgCookBook::createText(
  osg::Vec3( 50.0f, 50.0f, 0.0f), "", 10.0f);
osg::ref_ptr<osg::Geode> textGeode = new osg::Geode;
textGeode->addDrawable( text );
```

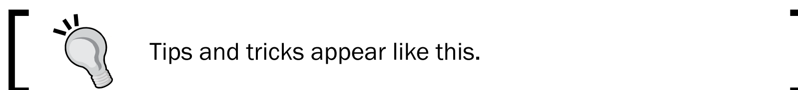
Any command-line input or output is written as follows:

```
# sudo apt-get install subversion
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You may easily download the binary packages of specified platform in the **Binary Packages** section".



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can also download the latest version of the source code package at the author's GitHub repository `https://github.com/xarray/osgRecipes`. All recipes of this book are included in this link, as well as more OSG-related examples written by the author and other contributors.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Customizing OpenSceneGraph

In this chapter, we will cover:

- Checking out the latest version of OSG
- Configuring CMake options
- Building common plugins
- Compiling and packaging OSG on different platforms
- Compiling and using OSG on mobile devices
- Compiling and using dynamic and static libraries
- Generating the API documentation
- Creating your own project using CMake

Introduction

OpenSceneGraph, which will also be abbreviated as OSG in the following parts of this book, is one of the best open source, high performance 3D graphics toolkits. It is designed to run under different operation systems, and even mobile platforms. The **CMake** build system (`http://www.cmake.org/`) is used to configure its compilation process and generate native makefiles, as well as packaging the binaries and development files.

OSG also contains hundreds of plugins for reading and writing files. Some of the plugins require providing external dependencies, and some may not be workable under specified platforms. Meanwhile, there are plenty of options to enable or disable while you are compiling OSG from the source code. These options are designed and implemented using **CMake scripts**, with which we could also create our own projects and provide different choices to the team or the public (if you are working on open source projects) too.

We are going to talk about the following concepts in this chapter: The customization of the OpenSceneGraph library from source code, the understanding of basic CMake scripts, and the construction of your own programs by reusing them.

Of course, you may select to directly download the prebuilt binaries, which is already configured in an irrevocable way, to save your time of compiling and start programming at once. The OpenSceneGraph official download link is:

`http://www.openscenegraph.org/projects/osg/wiki/Downloads`

Binaries provided by the AlphaPixel, including Windows, Mac OS X, and Linux versions, can be found at:

`http://openscenegraph.alphapixel.com/osg/downloads/free-openscenegraph-binary-downloads`

And the online installer for Windows developers is located at:

`http://www.openscenegraph.org/files/dev/OpenSceneGraph-Installer.exe`

Checking out the latest version of OSG

The first step we should do to customize the OpenSceneGraph library is to obtain it. Yes, you may simply get the source code of a stable version from the official download link we just introduced; otherwise, you could also find all kinds of developer releases in the following page:

`http://www.openscenegraph.org/projects/osg/wiki/Downloads/DeveloperReleases`

Developer releases, with an odd minor version number (the first decimal place), always contains some new functionalities but haven't undergone different test rounds. For instance, 2.9.12 is a developer release, and it is one of the stepping stones towards the next stable release, that is, OpenSceneGraph 3.0.

Pay attention to the phrase 'new functionalities' here. Yes, that is what we really care about in this cookbook. It would be boring if we still focus on some very basic scene graph concepts such as group nodes and state sets. What we want here will be the latest features of OSG and OpenGL, as well as examples demonstrating them. So we will try to acquire the latest version of the source code too, using the source control service.

For beginners of OSG programming, please refer to the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*. Some other good resources and discussions can be found in the "osg-users" mailing list and Paul Martz's website (`http://www.osgbooks.com/`).

Getting ready

You have to make use of the **Subversion** tool, which is a popular revision-control system used by OSG. Its official website is:

`http://subversion.apache.org/`

You can easily download the binary packages of the specified platform in the **Binary Packages** section, which are mostly maintained by some third-party organizations and persons. Of course, you may also compile Subversion from source code if you have interest.

For Debian and Ubuntu users, make sure you have the root permission and type the command `apt-get` in the terminal as shown in the following command line:

```
# sudo apt-get install subversion
```

The hash sign (#) here indicates the prompt before the command. It may change due to different platforms.

For Windows users, a GUI client named **TortoiseSVN** is preferred. It is built against a stable enough version of Subversion, and provides easy access to different source control operations. You may download it from:

`http://tortoisesvn.net/downloads.html`

How to do it...

We will take Ubuntu and Windows as examples to check out the latest OSG source code with Subversion. Users of other platforms should first find the correct location of the executable file (usually named `svn`) and follow the steps with appropriate permissions.

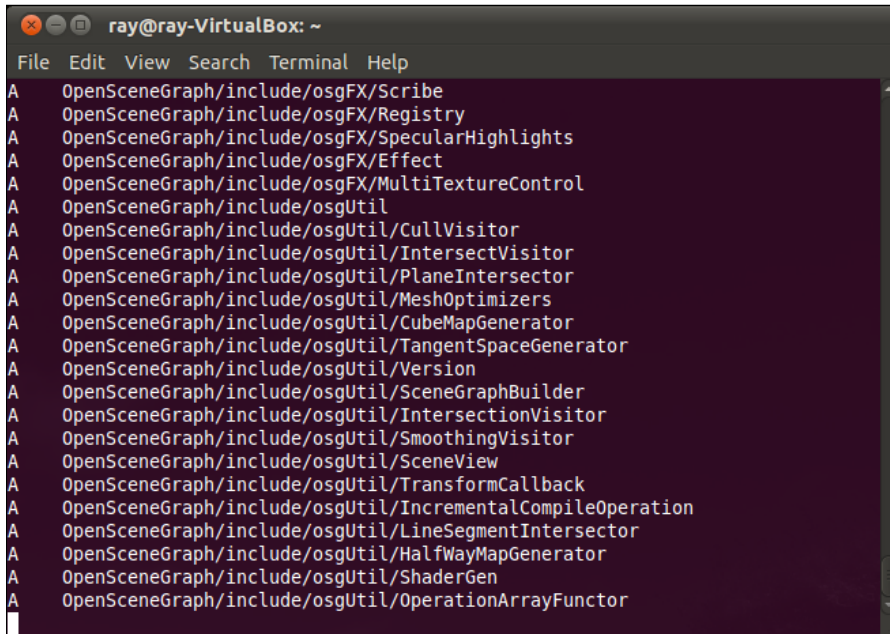
We will split the recipe into two parts—for Ubuntu users and Windows users.

Be aware of the phrase 'check out'. It can be explained as downloading files from the remote repository. Another important word that you need to know is 'trunk'. It is the base of a project for the latest development work. So, 'check out the trunk' means to download the cutting-edge version of the source code. This is exactly what we want in this recipe.

For Ubuntu users

1. Check out the OpenSceneGraph trunk by typing the following command in the terminal (you may have to add `sudo` at the beginning to run as an administrator):

```
# svn checkout
http://www.openscenegraph.org/svn/osg/OpenSceneGraph/trunk
OpenSceneGraph
```



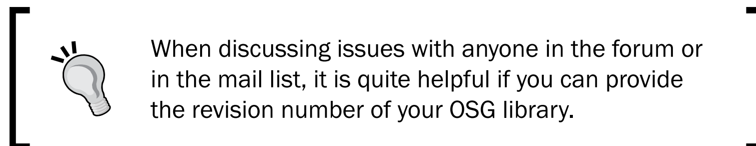
2. The first argument, `checkout` indicates the command to use. The second argument is the remote link to check out from. And the third one, `OpenSceneGraph` is the local path, in which downloaded files will be saved. Subversion will automatically create the local sub-directory if it does not exist.

3. Now you can take a look into the local directory `./OpenSceneGraph`. It contains the entire source code of the latest OSG now! Before configuring and compiling it, there is no harm in checking the source information first. Run the following command in the directory:

```
# cd OpenSceneGraph
# svn info
```

```
ray@ray-VirtualBox: /home/OpenSceneGraph
File  Edit  View  Search  Terminal  Help
ray@ray-VirtualBox:/home/OpenSceneGraph$ svn info
Path: .
URL: http://www.openscenegraph.org/svn/osg/OpenSceneGraph/trunk
Repository Root: http://www.openscenegraph.org/svn/osg
Repository UUID: 16af8721-9629-0410-8352-f15c8da7e697
Revision: 12466
Node Kind: directory
Schedule: normal
Last Changed Author: robert
Last Changed Rev: 12466
Last Changed Date: 2011-05-28 00:04:18 +0800 (Sat, 28 May 2011)

ray@ray-VirtualBox:/home/OpenSceneGraph$
```

4. This screenshot shows some useful information: **URL** is the remote address from which you checked the source code out; **Revision** is an automatically increasing number which could indicate the version of the code.



When discussing issues with anyone in the forum or in the mail list, it is quite helpful if you can provide the revision number of your OSG library.

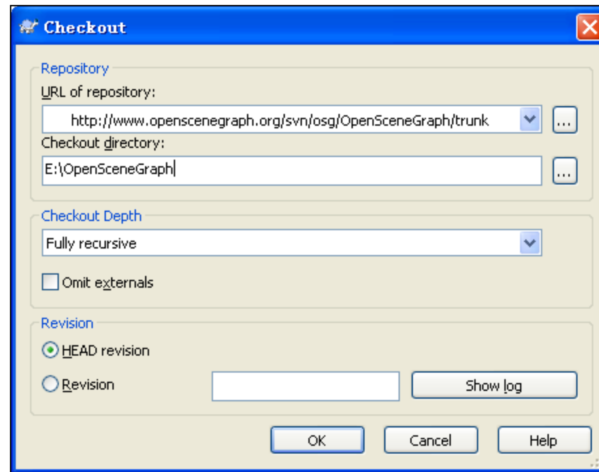
5. Remember that OSG is growing all the time. The source code you have checked out may be outdated in the next few days, or even in the next hour. The **source tree** may be modified to add new features or make fixes to previous functionalities. If you want to update these changes, go to the local directory and type the following:

   ```
   # cd OpenSceneGraph
   # svn update
   ```

   Nothing will happen if no changes are made after the last updating. And there will be conflicts if you have altered some of the source code locally. In that case, you should consider removing these modified files and re-update the trunk to recover them. If you are going to commit your changes to the official OpenSceneGraph developer team, use the "osg-submissions" mailing list instead.

For Windows users

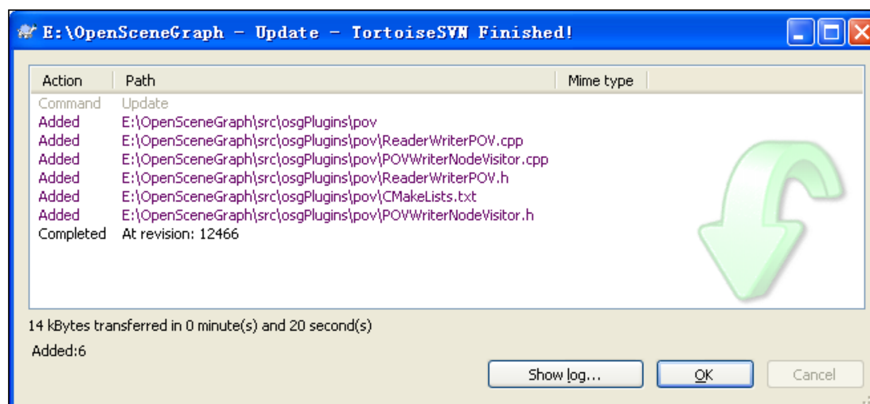
1. It will be a little easier to check out and update the trunk if you are using TortoiseSVN. Right click on the desktop or in a specified folder and you will see an **SVN Checkout** menu item, if TortoiseSVN is installed properly. Click on it and fill in the pop up dialog as shown in the following screenshot:



2. The most important options here are **URL of repository** and **Checkout directory**. Be sure to paste the following address to the former and specify an empty local folder for the latter:

   `http://www.openscenegraph.org/svn/osg/OpenSceneGraph/trunk`

3. Everything will be done automatically after you click on **OK** and you will soon find the source code in the right place. Right click on the newly created directory, and there is a new **SVN Update** menu item. Use it to update to the latest trunk version.



How it works...

Source code control is pretty useful when you are working with a team and have to share sources with other developers. Of course, we may put all source files in one folder on the network driver for everyone to visit and edit. But there may be serious conflicts if more than one developer is modifying the same file at the same time. And in such cases, someone's changes will definitely be lost.

The solution is to save recently added files on a remote server, which cannot be modified directly. Each developer can have an own copy on the local disk by performing the `checkout` operation. Developers who have the 'write' permission can commit their code to the server, and the server will synchronize all changes to every other owner's copy when they perform the `update` operation.

This is what the OSG developer team actually does. Everyone can use the Subversion tool to clone a copy of the latest source code and keep it fresh, but only a few core developers have the rights to upload their code, and help more contributors to submit their code.

There's more...

The Subversion tool can be used to manage the OSG sample data and some other OSG-related projects as well. Some commands will be listed here for convenience.

Here is the command to obtain the latest sample data (you could also set the environment variable `OSG_FILE_PATH` to the local path here):

```
# svn checkout
http://www.openscenegraph.org/svn/osg/OpenSceneGraph-Data/trunk
OpenSceneGraph-data
```

VirtualPlanetBuilder is a terrain database-creation tool which will be used for managing and rendering massive paged data. We are going to demonstrate it in *Chapter 7*. Here it is so you can check it out for later use:

```
# svn checkout
http://www.openscenegraph.org/svn/VirtualPlanetBuilder/trunk
VirtualPlanetBuilder-trunk
```

Besides checking out the source code and updating it with Subversion, sometimes you may also want to export the whole source code directory to a clean one, that is, to clone the source code. The `export` command will work for you here, for example:

```
# svn export this_folder/OpenSceneGraph-trunk
/another_folder/cloned-trunk
```

Remember, don't directly copy the directory. Subversion puts a lot of hidden folders (named `.svn`) inside to help manage the source code. And it is really a waste if we copy these to the target directory too.

It is certainly beyond this book to introduce all other SVN commands one by one. Some additional books would be of help if you are interested in this famous source control system, such as "*Version Control with Subversion*", *Ben Collins-Sussman*, *O'Reilly Media*. The online version can be found at `http://svnbook.red-bean.com`).

Configuring CMake options

If you have an experience of compiling OSG from the source code, you should already be familiar with the CMake system and the `cmake-gui` GUI tool. This book is neither a CMake tutorial book nor a step-by-step OSG compilation guide. Here we will quickly go through the configuration process, and tell you how to make use of some key options to change the behaviors and results.

Again, to come to understand what should be done before and after the configuration procedure, please refer to the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*.

Getting ready

At the least you should have the OSG source code, the CMake software, and a workable C++ compiler. **GCC** is the most common compiler for GNU and BSD operating systems, including most Linux distributions. Windows developers may choose **Visual Studio** or **MinGW**, or use **Cygwin** to construct a Linux-like environment. For Mac OS X users, **XCode** is preferred as the kernel developing toolkit.

CMake binary packages for various systems are available at:

`http://www.cmake.org/cmake/resources/software.html`

You may also use the `apt-get` command here to install the `cmake` (command-line mode) and `cmake-gui` (GUI mode) utilities separately:

```
# sudo apt-get install cmake
# sudo apt-get install cmake-gui
```

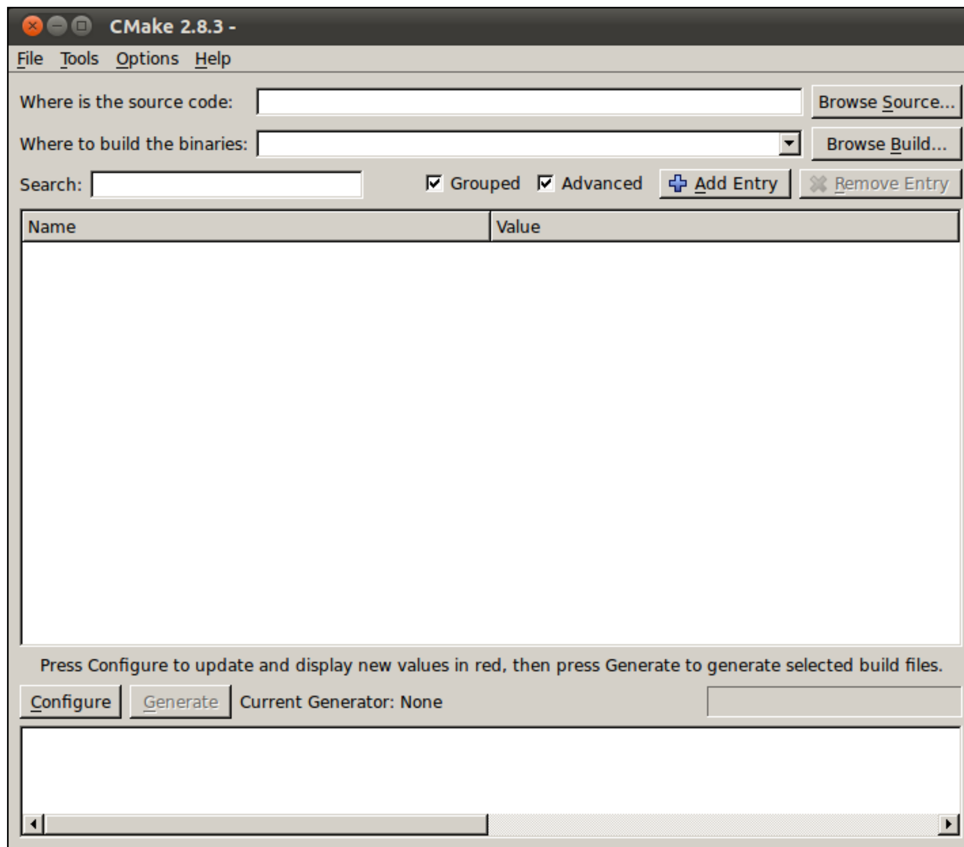
You must have the **OpenGL** library before compiling OSG. This is certainly the bottom line. Install it with `apt-get` if you don't have it by executing the following commands:

```
# sudo apt-get install libgl1-mesa-dev
# sudo apt-get install libglu1-mesa-dev
```

How to do it...

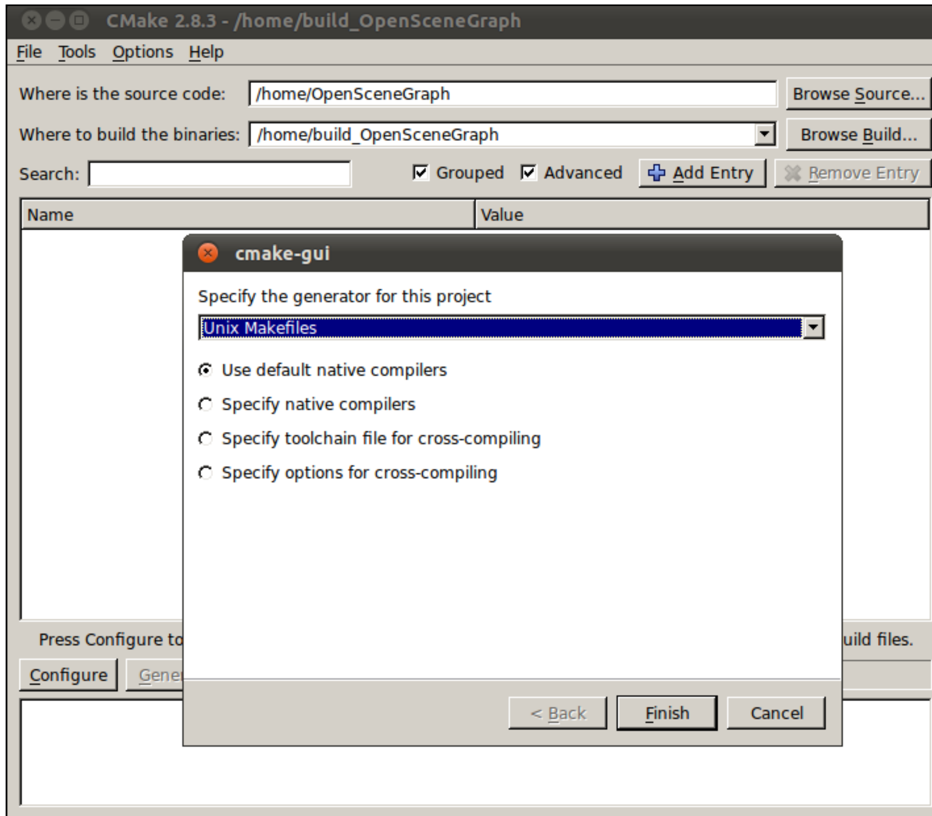
The `cmake-gui` utility has a similar user interface under every platform. And the configuring steps are of little difference too. We will only introduce the Ubuntu case in this recipe.

1. Run `cmake-gui` with administrator permission. Drag and drop `CMakeLists.txt` from OSG root directory to the GUI window. You will see the text boxes of source and build destinations changed immediately.



2. Edit the **Where to build the binaries** box and specify a different place for the generated makefiles or project files, that is, an **out-of-source** build. That is because the `SVN checkout` operation will establish the source directory with an update and revision information of every file. Any newly added items will be marked and considered as 'to be committed to remote repository later', therefore, an **out-of-source** build will prevent the generated project and temporary files from being recorded improperly.

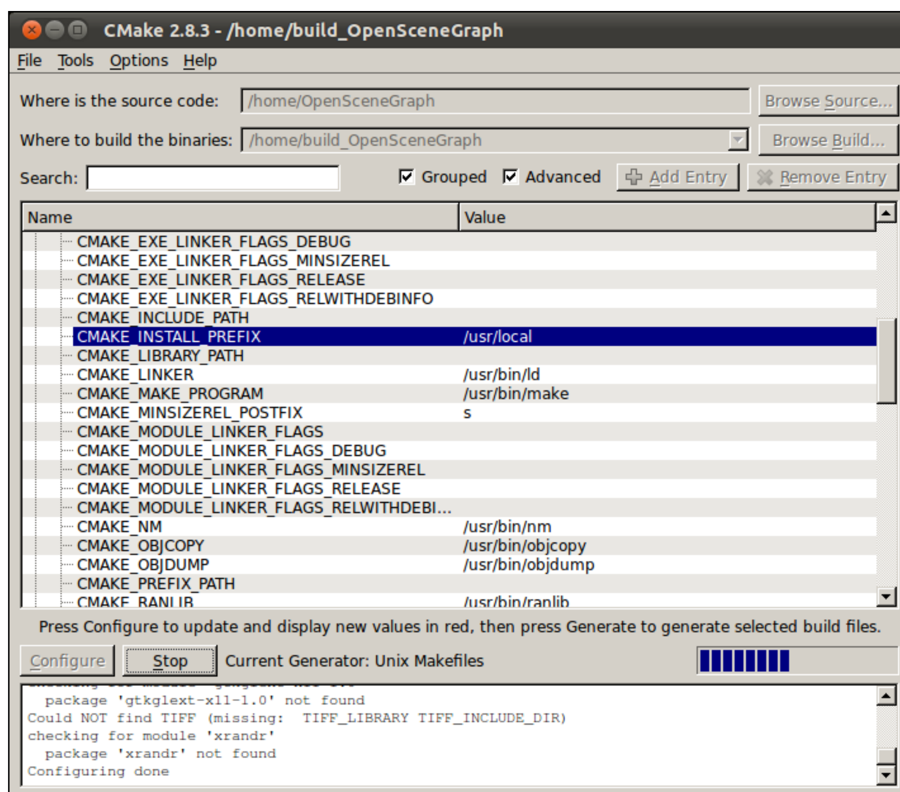
3. Click on **Configure** and select a generator corresponding to your system (**Unix Makefiles** for Ubuntu). After checking the system automatically, you will see a lot of options appear in the GUI. Click on **Finish** as shown in the following screenshot:



4. Next, check the **Grouped** checkbox to put the options in a more readable order. All the options shown here are marked with red at present, which means that they are not set yet. Change one or more of these options and click on **Configure** to confirm them. New unset options may appear as the results of previous choices. All you have to do is to confirm them until there are no red items, and click on **Generate** to finish it up.

   The default values of the configuration options are good enough. So we can just leave them except for setting up the build type (debug or release libraries) and the install prefix under which all OSG binaries and development files will be installed.

5. The `CMAKE_BUILD_TYPE` item in the `CMake` group is used for deciding the build type. Input **Debug** in the **Value** column if you want a debug version of libraries and binaries. Input **Release** or leave it blank if not. By default, it is just empty (if you are using a Visual Studio generator under Windows, it contains multiple configurations).
6. The `CMAKE_INSTALL_PREFIX` item, which is also in the `CMake` group, will help specify the base installation path. By default, it is set to `/usr/local`. Type the value manually or use the browse button on the right-hand side to make decisions.
7. Confirm and generate the makefiles. Have a look into the target directory and if you like, do `make` and `make install` now (but you may have to do this again and again during the next few recipes).



There's more...

CMake supports various kinds of generators, each of which could be used under one or more specified platforms. The following table will provide more details about creating OSG makefiles or solutions using different generators:

Generator	Platform	Required environment	Result
MinGW Makefiles	Windows	MinGW	Makefiles for use with `mingw32-make`
NMake Makefiles	Windows	Visual Studio 7/8/9/10	Makefiles for use with `nmake`
Unix Makefiles	Windows Linux Mac OS X	GCC and G++ (available for Cygwin users under Windows)	Standard Unix makefiles
Visual Studio (Specify the version number)	Windows	Visual Studio 7/8/9/10 (Previous versions are not suitable for compiling OSG)	Visual Studio solutions
CodeBlocks (Specify the makefile type: MinGW, NMake, or Unix)	Windows Linux Mac OS X	Code::Blocks IDE	Code::Blocks projects
XCode	Mac OS X	Apple XCode	XCode projects

Building common plugins

OSG works with hundreds of kinds of plugins with a uniform prefix `osgdb_*`. Most of them can read specified file formats (mainly models and images) into scene objects, and some can also write the nodes or images back to files. The plugin mechanism brings us a lot of convenience as it handles file extensions and chooses a corresponding reader/writer for us internally. Developers will only have to call the `osgDB::readNodeFile()` or `osgDB::readImageFile()` method with the filename while writing OSG-based applications.

In this recipe, we are going to configure CMake options to build with the most common plugins. For general plugins, such as BMP, DDS, and the native OSG format reader/writer, the build process will be faithfully executed and they will always be generated without doubts. But for plugins requiring external dependencies, such as DAE (requires **Collada DOM**) and JPEG (requires **libJPEG**), CMake scripts will automatically remove the plugin sub-directories from the build queue in case the third-party includes a path and libraries are not specified correctly.

It is impossible to get all the plugins built under a certain platform. But we still have some very common plugins depending on external libraries. It is really a pity to leave them alone and thus lose some good features, such as **FreeType** font support and network data transferring with **libCURL**. To avoid missing them, there are two rules to follow: First, download or compile the development packages of external libraries; then, set related options while configuring OSG with CMake.

Getting ready

We will first make a list of the most practical plugins, download their dependent libraries, and set them up in the `cmake-gui` window. The list includes `curl`, `freetype`, `gif`, `jpeg`, `png`, `Qt`, and `zlib` plugins.

Have a look at the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, in case you are interested in other plugins too. In *Chapter 10*, you will find a complete list of file I/O plugins currently supported by OSG.

How to do it...

Thanks to the `apt-get` tool, we can make things easier under Ubuntu as follows:

1. Download all the binary packages of external dependencies with the `apt-get` command:

```
# sudo apt-get install libcurl4-openssl-dev
# sudo apt-get install libfreetype6-dev
# sudo apt-get install libgif-dev
# sudo apt-get install libjpeg62-dev
# sudo apt-get install libpng12-dev
# sudo apt-get install zlib1g-dev
```

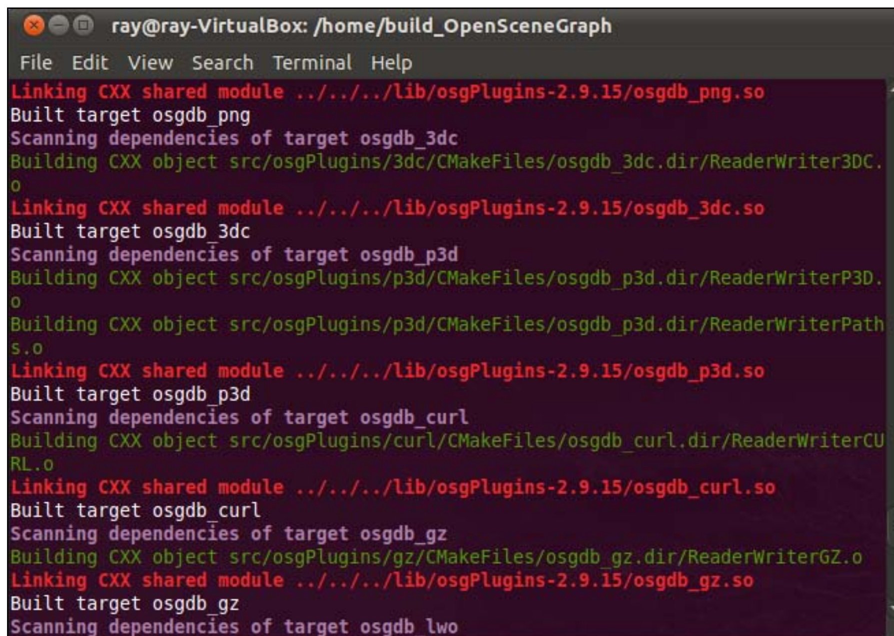
2. Download **Qt** online installer from:

   `http://qt.nokia.com/downloads/sdk-linux-x11-32bit-cpp`

3. Make it executable and run it to download and install Qt SDK:

```
# sudo chmod u+x Qt_SDK_Lin32_online_v1_1_1_en.run
# sudo ./Qt_SDK_Lin32_online_v1_1_1_en.run
```

4. Now it's time to configure these libraries using `cmake-gui`. Just reopen the GUI window and click on the **Configure** button. Check on the **Advanced** checkbox to show all the options and check into related groups, including CURL, FREETYPE, GIFLIB, JPEG, QT, PNG, and ZLIB. If you see nothing unexpected, you will happily find that every `*_INCLUDE_DIR` and `*_LIBRARY` is set correctly. CMake has already queried these installed libraries and got them ready for compiling corresponding plugins.
5. Generate makefiles, and enjoy the compiling work again. It will take much shorter time to finish the entire process this time, unless you remove the build directory containing all the intermediate object files.



How it works...

Let us see what these plugins do and where to learn about their dependencies:

- `osgdb_curl`: This plugin can provide OSG with network transferring functionalities. It helps fetch data from HTTP and FTP servers and use local file readers to parse them. It requires libCURL as the dependence. Binary packages and source code can be downloaded from `http://curl.haxx.se/download.html`.

- `osgdb_freetype`: This plugin is important as it provides osgText with the ability of displaying TrueType fonts (TTF, TTC formats, and so on). The FreeType library is necessary, which could be downloaded from `http://freetype.sourceforge.net/download.html`.
- `osgdb_gif`: This plugin reads GIF and animated GIF images, with GifLib (`http://sourceforge.net/projects/giflib/`) as dependence.
- `osgdb_jpeg`: This plugin reads JPG and JPEG images, with libJPEG (`http://www.ijg.org/`) as dependence.
- `osgdb_png`: This plugin reads PNG images, with Zlib and libPNG (`http://www.libpng.org/pub/png/libpng.html`) as dependence.
- `osgQt`: The osgQt library can be used for OSG and Qt integration and QFont support. It requires the Qt toolkit (`http://qt.nokia.com/downloads/`). Don't miss it as we will talk about some interesting implementations about Qt and OSG in the following chapters.
- Zlib: The Zlib library is used widely as the dependence of core libraries and plugins. For example, the `osgDB` library and the native IVE format can depend on Zlib to support file compression. And the `osgdb_png` plugin needs it too. Its official website is `http://zlib.net/`.

There's more...

For Windows users, it may not be simple to get all these dependencies at once. And to compile them from source code one-by-one is a really harrowing experience for some people (but for somebody else, it may be exciting. Tastes differ!). The following link may be helpful as it contains Win32 ports of some GNU libraries (FreeType, Giflib, libJPEG, libPNG, and Zlib):

`http://gnuwin32.sourceforge.net/packages.html`

Win32 binaries of libCurl and Qt can be found on their own websites.

CMake may not work like a charm under Windows systems, that is, it can hardly find installed libraries automatically. Specify the `ACTUAL_3DPARTY_DIR` option in **Ungrouped Entries** to the root path of uncompressed binaries and development files, and reconfigure to see if it works. Also, you can refer to *Chapter 10* of the "*OpenSceneGraph 3.0*: *Beginner's Guide*" book.

Compiling and packaging OSG on different platforms

You must be familiar with the compilation of OSG under Unix-like systems by simply inputting the following commands in a terminal:

```
# sudo make
# sudo make install
```

Of course, the prerequisite is that you must have already configured OSG with `cmake-gui` or some other command-line tools and generated the makefiles as well. If not, then you may first read the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, for basic knowledge about compiling OSG.

For Windows and Mac OS X users, we always have some slightly different ways to do this because of the wide use of IDEs (Integrated Development Environment), but we may also build from makefiles by specifying the generator type in the CMake configuration window.

Another interesting topic here is the packaging of **built binaries**. It may automatically create RPM, DEB, and GZIP packages under Linux, and even self-extracting installer sunder Windows.

Getting ready

The packaging feature is implemented by the **CPack** packaging system integrated with CMake, so you don't have to download or install it on your own. TGZ (`.tar.gz`) is chose as the default package format under Linux and ZIP is the default one under Windows.

WinZIP (`http://www.winzip.com/win/en/index.htm`) is required by CPack to generate ZIP files. And if you want a cool self-extracting installer/uninstaller, you may get the **NSIS** (**Nullsoft Scriptable Install System**) from:

`http://nsis.sourceforge.net/Download`

Set the environment variable `PATH` to include the location of the executables, and CPack will automatically find and make use of each of them.

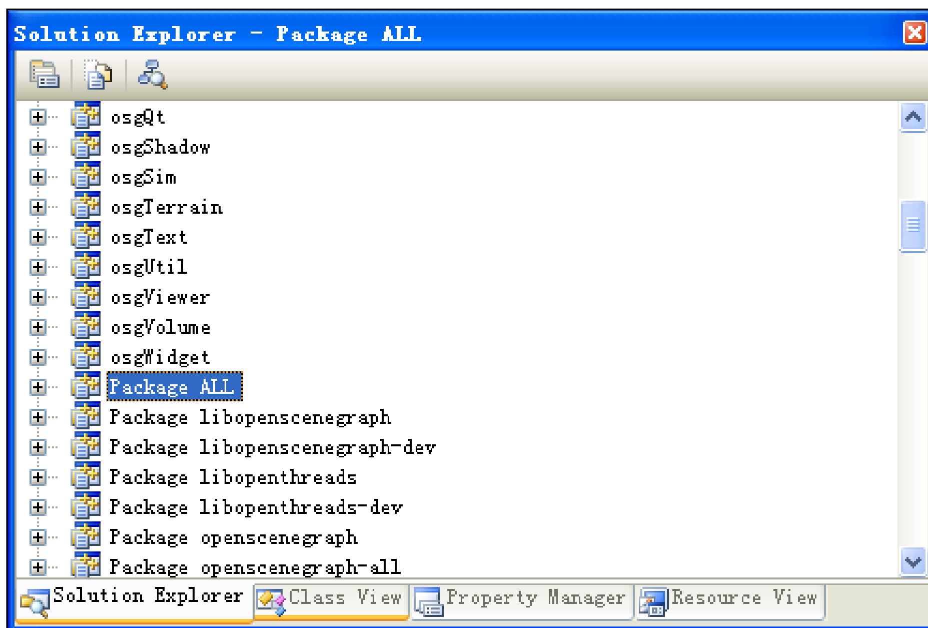
How to do it...

Under any UNIX-link systems, to enable packaging, you have to open `cmake-gui` and set `BUILD_OSG_PACKAGES` to `ON`. Looking at the option `CPACK_GENERATOR` in **Ungrouped Entries**, you can just keep the default value `TGZ`, or change it to `RPM` or `DEB` if you wish to. Corresponding software must be installed to ensure the package generator works.

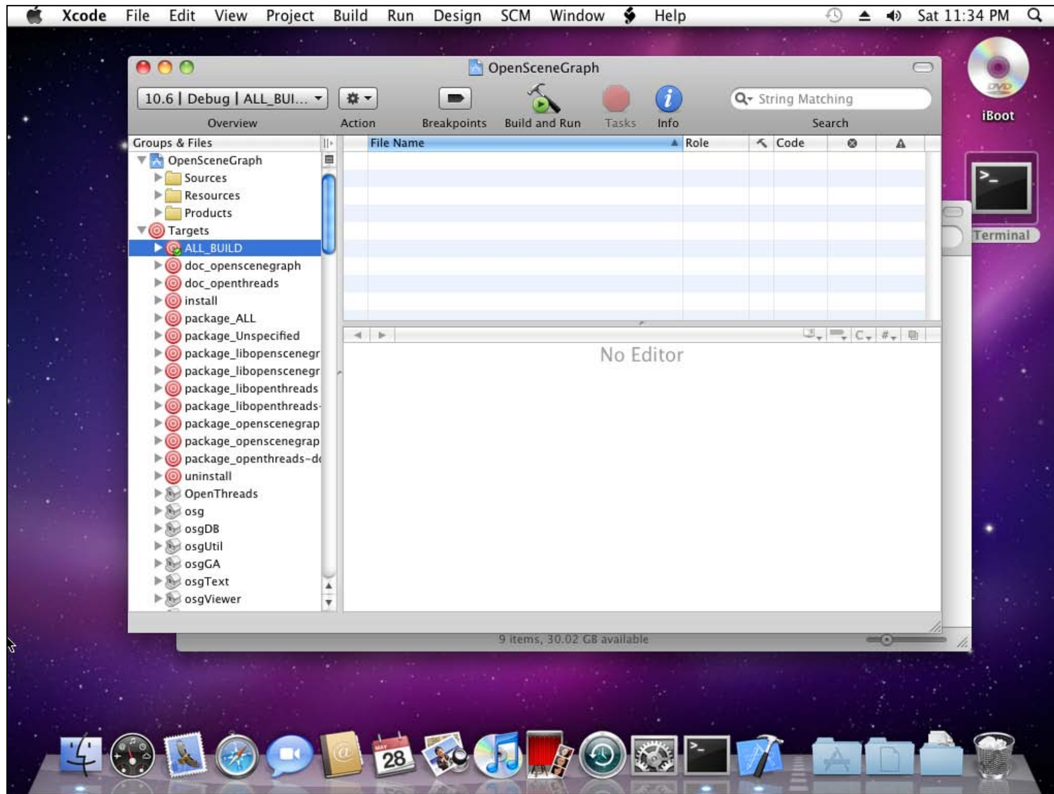
After that, there is nothing besides `make` to build OSG libraries. Open a terminal and type `make` in the build folder to compile the OSG library, or use the generated solution file if you are using Visual Studio products. But instead of installing to a specific directory, this time you could make a series of `.tar` archives containing OSG binaries, development files, and applications and share these developer files with others. Just type the following:

```
# sudo make package_openscenegraph-all
```

Windows users may open the generated Visual Studio solution. Build the sub-project `ALL_BUILD` and click on **INSTALL** to compile one-by-one and install all targets. But similarly, they could also select to build the sub-project **Package openscenegraph-all** instead of clicking on **INSTALL**. This will result in a series of zipped files or NSIS installers.



For Mac OSX users, start XCode and open `OpenSceneGraph.xcodeproj` from the build directory. Choose **Targets** in the **Groups & Files** view and build **ALL_BUILD** and **install** in that order. Again, it is possible to choose **package_ALL** after all libraries built, if you have already had CPack options set before.



How it works...

Generated packages may differ as the result of changing CMake options. The following table shows what you will get after the `make package` operation. The prefix (for example, `openscenegraph-all-3.0.0-Linux-i386-Release-*`) of each package file is just ignored here.

Package name	Required option	Description
libopenthreads	None	The OpenThreads library file
libopenthreads-dev	None	The OpenThreads include files and static-link libraries
libopenscenegraph	None	The OpenSceneGraph core library files
libopenscenegraph-dev	None	The OpenSceneGraph include files and static-link libraries
openscenegraph	BUILD_APPLICATIONS	Applications (osgviewer, osgversion, and so on)
openscenegraph-examples	BUILD_EXAMPLES	Examples
openthreads-doc	BUILD_DOCUMENTATION	The OpenThreads reference documentation
openscenegraph-doc	BUILD_DOCUMENTATION	The OpenSceneGraph reference documentation

Compiling and using OSG on mobile devices

It is really exciting to know that the latest OSG supports some of the most popular mobile platforms. After preparing necessary environments and changing some CMake options, we can then easily build OSG for iOS and Android systems, including iPhone, iPad, most Android based devices, and their simulators.

Remember, your mobile device must support **OpenGL ES** (**OpenGL for Embedded Systems**) to run any OSG applications. And there are also various API redefinitions and limitations that will make some functionalities work improperly. Fortunately, Google Android provides SDKs, simulators, and GLES libraries for development. So it will be an excellent example for demonstrating how to compile and use OSG on mobile devices.

Getting ready

Download Android SDK and Android NDK from their official websites:

`http://developer.android.com/sdk/index.html`

`http://developer.android.com/sdk/ndk/index.html`

Remember that you need NDK r4 or a later version to make the compilation successful. However, the SDK version doesn't matter most of the time.

A very important note before you are going on: At present, some Tegra devices, including Acer IconiaTab and Motorola XOOM, are unable to work with NEON extensions. But OSG at present doesn't provide direct options to disable NEON, so the only way to get these devices to work with OSG libraries in the future is to comment the following line in `PlatformSpecifics/Android/modules.mk.in`:

```
# LOCAL_ARM_NEON := true
```

And compile OSG following the instructions in the next section.

How to do it...

You have to make changes in CMake options to let OSG realize that it is going to work under GLES v1 or v2, and must be cross-compiled with the C++ compiler provided by Android NDK.

In this recipe, we will only show how to configure OSG with GLES v1 support.

1. Start `cmake-gui` and reset the options in the group `OSG` as shown in the table:

Option name	Value	Description
`DYNAMIC_OPENSCENEGRAPH`	`OFF`	Don't build dynamic libraries on Android
`DYNAMIC_OPENTHREADS`	`OFF`	Don't build dynamic libraries on Android
`OSG_BUILD_PLATFORM_ANDROID`	`ON`	Enable to build OSG for Android
`OSG_CPP_EXCEPTIONS_AVAILABLE`	`OFF`	Disable the use of C++ exceptions
`OSG_GL1_AVAILABLE`	`OFF`	No support for OpenGL 1.x
`OSG_GL2_AVAILABLE`	`OFF`	No support for OpenGL 2.x
`OSG_GL3_AVAILABLE`	`OFF`	No support for OpenGL 3.x
`OSG_GLES1_AVAILABLE`	`ON`	Add supports for OpenGL ES 1.x
`OSG_GLES2_AVAILABLE`	`OFF`	No support for OpenGL ES 2.x
`OSG_GL_DISPLAYLISTS_AVAILABLE`	`OFF`	No support for display lists
`OSG_WINDOWING_SYSTEM`	None	Android has its own windowing system, so don't use any others here

2. Configure the `ANDROID_NDK` option which appears after you press **Configure**. Make sure the `ndk_build` executable is in your NDK installation path (we suppose it is `Your_NDK_Root`) and specify `Your_NDK_Root` as the value.
3. Configure the OpenGL include directory and libraries in the `OPENGL` group to support GLES v1:

Option name	Value
`OPENGL_INCLUDE`	The parent directory of EGL. You can find it at `Your_NDK_Root/platforms/android-9/arch-arm/usr/include`. `Your_NDR_Root` and `android-9` may differ according to your NDK installation.
`OPENGL_egl_LIBRARY`	The `libEGL` library. You can find it at `Your_NDK_Root/platforms/android-9/arch-arm/usr/lib`.
`OPENGL_gl_LIBRARY`	The `libGLESv1_CM` library (may differ). You can find it at `Your_NDK_Root/platforms/android-9/arch-arm/usr/lib`.
`OPENGL_glu_LIBRARY`	Don't set it. GLES can't use the GLU library.

4. Nearly done! Now reset all the third-party library options (such as the `JPEG`, `PNG` groups, and so on) to `NOTFOUND`. That is because all of them have to be rebuilt with the Android compiler first, and there may be several limitations and errors. We are not going to discuss this painful process in this book.
5. Now simply run the following command:

   ```
   # sudo make
   ```

6. If you want to add some NDK building options, such as `-B` (to do a complete rebuild) or `NDK_DEBUG=1` (to generate debugging code), use the `ndk_build` executable directly:

   ```
   # Your_NDK_Root/ndk_build NDK_APPLICATION_MK=Application.mk
   ```

7. Now wait for the libraries to finish compiling.

```
ray@ray-VirtualBox: /home/build_OpenSceneGraph
File  Edit  View  Search  Terminal  Help
Compile++ thumb  : osgSim <= LightPoint.cpp
Compile++ thumb  : osgSim <= LightPointDrawable.cpp
Compile++ thumb  : osgSim <= LightPointNode.cpp
Compile++ thumb  : osgSim <= LightPointSpriteDrawable.cpp
Compile++ thumb  : osgSim <= LineOfSight.cpp
Compile++ thumb  : osgSim <= MultiSwitch.cpp
Compile++ thumb  : osgSim <= OverlayNode.cpp
Compile++ thumb  : osgSim <= ScalarBar.cpp
Compile++ thumb  : osgSim <= ScalarsToColors.cpp
Compile++ thumb  : osgSim <= Sector.cpp
Compile++ thumb  : osgSim <= ShapeAttribute.cpp
Compile++ thumb  : osgSim <= SphereSegment.cpp
Compile++ thumb  : osgSim <= Version.cpp
Compile++ thumb  : osgSim <= VisibilityGroup.cpp
StaticLibrary  : libosgSim.a
Compile++ thumb  : osgTerrain <= Layer.cpp
Compile++ thumb  : osgTerrain <= Locator.cpp
Compile++ thumb  : osgTerrain <= TerrainTile.cpp
Compile++ thumb  : osgTerrain <= TerrainTechnique.cpp
Compile++ thumb  : osgTerrain <= Terrain.cpp
Compile++ thumb  : osgTerrain <= GeometryTechnique.cpp
Compile++ thumb  : osgTerrain <= Version.cpp
StaticLibrary  : libosgTerrain.a
Compile++ thumb  : osgWidget <= Box.cpp
```

There's more...

We won't discuss any examples on Android here; this will be done in the *Chapter 9*, *Integrating with GUI*.

If you want to compile OSG with GLES v2, remember to handle the following CMake options in addition to the preceding table, besides enabling `OSG_GLES2_AVAILABLE` (but they must be turned on if you choose GLES v1):

Option name	Value	Description
`OSG_GLES1_AVAILABLE`	`OFF`	No support for OpenGL ES 1.x
`OSG_GL_MATRICES_AVAILABLE`	`OFF`	No support for OpenGL matrix functions
`OSG_GL_VERTEX_FUNCS_AVAILABLE`	`OFF`	No support for OpenGL vertex functions
`OSG_GL_VERTEX_ARRAY_FUNCS_AVAILABLE`	`OFF`	No support for OpenGL vertex array functions
`OSG_GL_FIXED_FUNCTION_AVAILABLE`	`OFF`	No support for all fixed functions

Maybe you are interested in how to configure OSG to use OpenGL ES instead of the standard OpenGL and GLU libraries. The following link will point out the main difference between the options of GLES v1 and GLES v2:

`http://www.openscenegraph.org/projects/osg/wiki/Community/OpenGL-ES`

OSG can also work under Mac OS X to support iOS, and thus support iPhone, iPad, and other Apple devices with GLES. Check the option `OSG_BUILD_PLATFORM_IPHONE` or `OSG_BUILD_PLATFORM_IPHONE_SIMULATOR` and specify the iOS SDK location. Follow the discussions on "osg-users" mailing list and try to work out the compilation yourselves.

The OpenGL ES development files are always necessary when you are developing on mobile devices. And for Windows users, there are some other OpenGL ES emulators for you to test if OSG and your applications work under such environments.

ARM OpenGL ES 2.0 Emulator can be found at `http://www.malideveloper.com/developer-resources/tools/opengl-es-20-emulator.php`.

Qualcomm Adreno SDK can be found at `http://developer.qualcomm.com/showcase/adreno-sdk`.

NVIDIA Tegra's x86 Windows OpenGL ES 2.0 Emulator can be found at `http://developer.download.nvidia.com/tegra/files/win_x86_es2emu_v100.msi`.

Compiling and using dynamic and static libraries

The main difference between dynamic and static libraries is how they are shared in applications. Dynamic libraries (or shared libraries) allow many programs to use the same library at the same time, but static ones don't.

By default, OSG generates dynamic libraries such as `libosg.so` under Linux or `osg.dll` under Windows. The executable only record required routines in a library and all actual modules will be loaded at runtime. This enables multiple executables to make use of only one library, rather than compiling the library code into each program. But it also brings disadvantages such as dependency and distribution problems.

With static linking, target executables will include every referenced part of external libraries and object files at compile time, and there will be no extra shared files. This ensures that all the dependencies are loaded with the correct version. You may not be bothered by the installation of your applications again, as end users won't complain that a **DLL file not found** error is not displayed while using static-linked executables.

Of course, static libraries always make the result a larger size. Fortunately, OSG provides both static and dynamic linking options. It's up to the developers to decide which would win in their case.

Getting ready

Start the `cmake-gui` utility. Now we are going to configure the build system to generate static or dynamic version of OSG libraries. Be careful; each time you switch between `static` and `dynamic`, the whole project including libraries, plugins, and applications will be rebuilt. So it is clever to make a `static-build` and a `dynamic-build` directory separately, if you want to update both configurations.

How to do it...

It is actually very easy to build static-link version of OSG libraries. We will assume you have already had a `static-build` directory for the building process.

1. Start `cmake-gui` and find the `DYNAMIC` group. It contains two options (checked by default)—`DYNAMIC_OPENTHREADS` and `DYNAMIC_OPENSCENEGRAPH`. Unmark them and confirm your changes.
2. That's enough! Let us generate the makefiles/solutions and start the native compilation work.

3. After the files are installed, go to the specified path and see what you have now. You will find that there are no `.so` or `.dll` files, and all the plugins are created as static-link libraries (`.a` or `.lib`), as well as the core libraries.

There's more...

This is what we have just described before—static linking doesn't require extra shared objects any more. All modules referred to by the executable will be directly included at compiling time.

While using static-linked plugins for programming, you must add corresponding file to the dependence list, and declare them in the global scope of the source code, which forces the compiler to look for modules in external dependencies. This can be done with a `USE_OSGPLUGIN()` macro. Another important macros include `USE_DOTOSGWRAPPER_LIBRARY()`, `USE_SERIALIZER_WRAPPER_LIBRARY()`, and `USE_GRAPHICSWINDOW()`. The first two can register native OSG and OSGB formats into your executable, and the last will specify the right windowing system to use. Without dynamic loading of files and dynamic allocating of global proxy variables, OSG wasn't able to automatically check for them this time.

The example `osgstaticviewer`, if you enabled `BUILD_OSG_EXAMPLES` in the `BUILD` group, could be a good example for such developing instructions.

Generating the API documentation

Before we start discussing this recipe, open the following link and have a look at it:

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

Some of you may say: "Oh, this is a wonderful reference guide for me during the programming work. It's impossible to keep all the classes in mind, and it's really rough to search for one method or function in the vast source directory. I'd love to have such a handy API document. But how did you make it, and how do you keep it fresh?"

Believe it or not, all this documenting work could be done by automatic generators, for example, **Doxygen** in our case. It will parse the source code and comments in prescribed forms, and output formatted results to HTML pages, or even CHM files.

And with the well-written build scripts, OSG can create such API documentation with the Doxygen tool in a very simple way.

Getting ready

Download the Doxygen tool first, and you can generate beautiful documents from the source code. The download link is:

http://sourceforge.net/projects/doxygen/files/

There is a **dot** utility created by the **Graph Visualization Software**. It can draw some types of hierarchical graphs and thus makes life more colorful. The toolkit can be found at:

http://www.graphviz.org/Download..php

Ubuntu users can install these two utilities with the `apt-get` command directly by running the following two command lines:

```
# sudo apt-get install doxygen
# sudo apt-get install graphviz
```

Lastly, Windows users may choose to compile a CHM file. **Microsoft HTML Workshop** is required in this situation. If you don't have one, download it at:

http://www.microsoft.com/downloads/en/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580cc

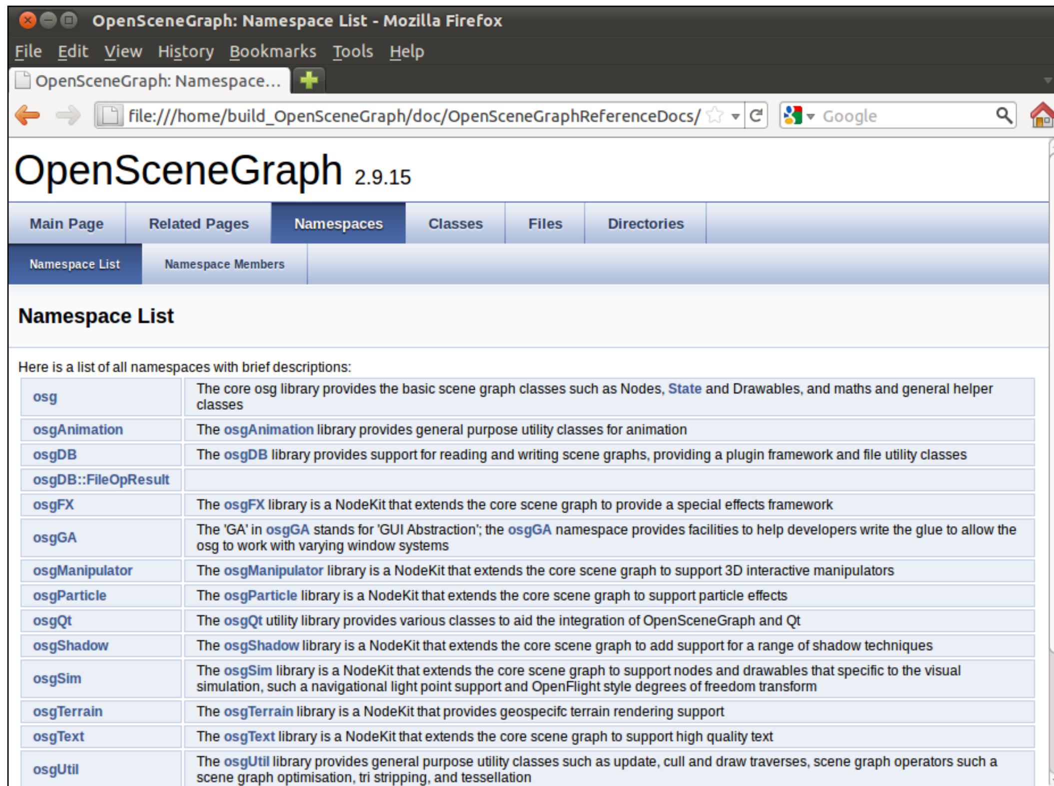
How to do it...

1. Start the `cmake-gui` window. Don't worry about a completely new compilation, which may take another few hours again. This time we are going to configure options for documentation building only.
2. Find the `BUILD` group and click on `BUILD_DOCUMENTATION`. Click on **Configure** for more choices.
3. A new group named `DOXYGEN` appears after reconfiguring. Look into the group and ensure that the `doxygen` and `dot` executables are set properly. Windows users may have to specify the locations of `doxygen.exe` and `dot.exe` manually.
4. Another group `DOCUMENTATION` is used to decide whether we should build with the option `HTML_HELP` (CHM file). Selecting it means we are going to compile HTML documents into a CHM file. It requires `hhc.exe` from the HTML Workshop as the executable.
5. For Windows users only, set up the location of `hhc.exe` and the Html Help SDK library. The latter can be found in the Windows SDK distribution.

6. The common `make` and `make install` commands won't affect the generation of API documents. Use the following commands to obtain the OpenThreads and OpenSceneGraph API documents:

```
# sudo make doc_openthreads
# sudo make doc_openscenegraph
```

7. Use any browser to open the `index.html` file in the `/doc/OpenSceneGraphReferenceDocs/` folder in your build directory. See what great work you have just done!



There's more...

If you have generated Visual Studio solution files, find the sub-project `DoxygenDoc` and build it separately. The `ALL_BUILD` and `INSTALL` projects, which must be run to compile and install all OSG libraries and applications, can never affect the compilation of documents, and vice versa. So you may build the API documents without building OSG.

Interested in the generation of API documents? Or do you want your own project to be documented in such an automatic process too? Change your commenting habit from now on. Doxygen will try to recognize some special forms of comments and create great-looking and practical reference manuals for you. See the link below for more details:

`http://www.stack.nl/~dimitri/doxygen/manual.html`

Creating your own project using CMake

Now, you may have a question like the following—since CMake is a really powerful tool for constructing self-adaptive build systems, is it possible that I could make use of it?

In this recipe, we will create a small enough script file, in which we will design the compilation strategy of a small project with several source files. Our goal is to detect the OSG installation, set up the include directory, and link libraries to the project, and finally generate an executable file in the user-defined path. This build script (in fact only one `CMakeLists.txt` file) will be used throughout this book.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Getting ready

While CMake and the OpenSceneGraph libraries are all installed, you don't have to prepare anything before writing scripts. Open a text editor, such as Visual Studio, UltraEdit, or even Notepad! Make a new folder for your project; save the script file as `.txt` and go ahead.

How to do it...

Let us start coding in a plain text file named `CMakeLists.txt`. The name must not be changed because CMake will use it for parsing scripts.

1. Set the common parts of the script, including the project name, postfix of Debug outputs, and CMake version configurations:

```
PROJECT(OSG_Cookbook)

CMAKE_MINIMUM_REQUIRED(VERSION 2.4.7)
SET(CMAKE_DEBUG_POSTFIX "d" CACHE STRING "add a postfix for
  Debug mode, usually d on windows")

# set cmake policy
IF(COMMAND CMAKE_POLICY)
    CMAKE_POLICY(SET CMP0003 NEW)
ENDIF(COMMAND CMAKE_POLICY)
```

2. Add necessary definitions and C++ compiling flags as you wish:

```
IF(WIN32)
  IF(MSVC)
    ADD_DEFINITIONS(-D_SCL_SECURE_NO_WARNINGS)
    ADD_DEFINITIONS(-D_CRT_SECURE_NO_DEPRECATE)
  ENDIF(MSVC)
ELSE(WIN32)
  SET(CMAKE_CXX_FLAGS "-W -Wall -Wno-unused")
ENDIF(WIN32)
```

3. Find and set the OpenSceneGraph to include the directory and library path in some predicted locations, which will be used as external dependencies of your application. If not found, users who configure the CMake options of your project must specify them manually to make the generator work.

```
# find include directory by looking for certain header file
FIND_PATH(OPENSCENEGRAPH_INCLUDE_DIR osg/Referenced
  PATHS
  $ENV{OSG_ROOT}/include
  /usr/include
  /usr/local/include
)

# find library directory by looking for certain library file
FIND_PATH(OPENSCENEGRAPH_LIB_DIR libso.so osg.lib
  PATHS
  $ENV{OSG_ROOT}/lib
```

```
  /usr/lib
  /usr/local/lib
)

# apply them to following projects
INCLUDE_DIRECTORIES(${OPENSCENEGRAPH_INCLUDE_DIR})
LINK_DIRECTORIES(${OPENSCENEGRAPH_LIB_DIR})
```

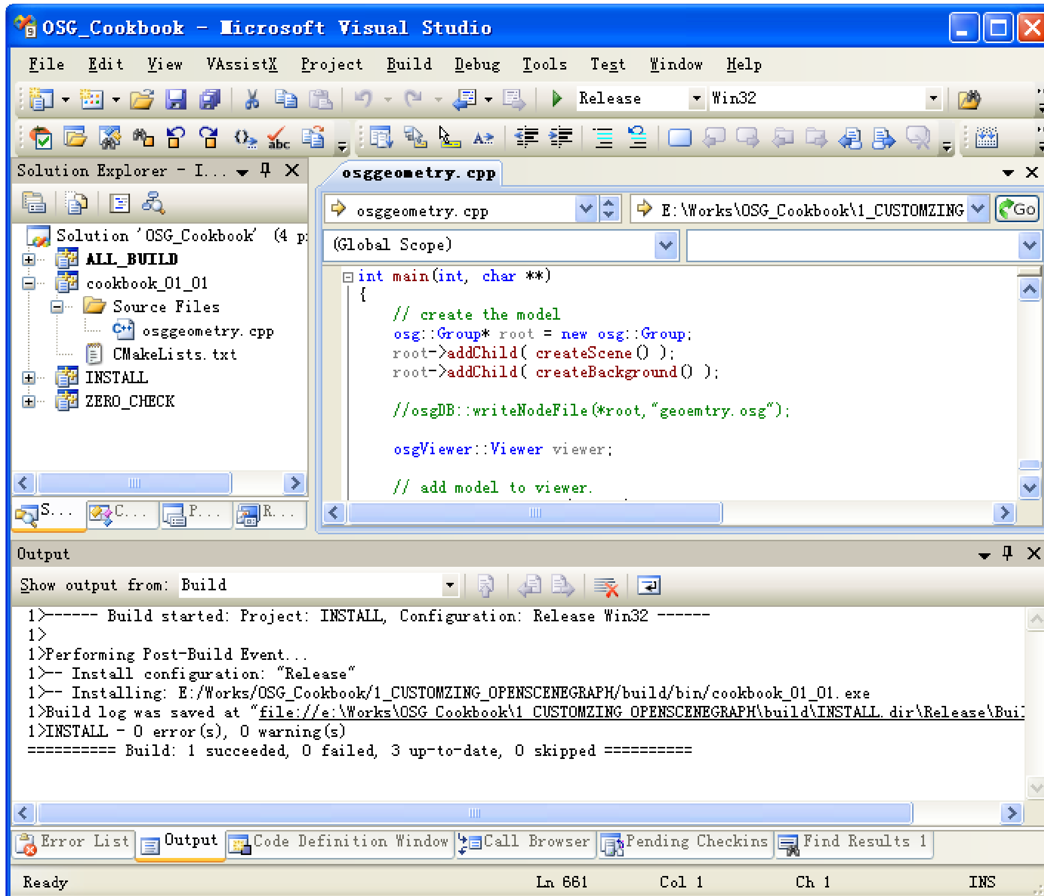
4. Set up your own project files here. We would just take `osggeometry.cpp` as an example. It can be located at `examples/osggeometry` of the OpenSceneGraph source code. Just copy the `.cpp` file to your project's folder:

```
SET(EXAMPLE_NAME cookbook_01_01)
SET(EXAMPLE_FILES osggeometry.cpp)

ADD_EXECUTABLE(${EXAMPLE_NAME} ${EXAMPLE_FILES})
SET_TARGET_PROPERTIES(${EXAMPLE_NAME} PROPERTIES
  DEBUG_POSTFIX "${CMAKE_DEBUG_POSTFIX}")
```

5. Link necessary OSG libraries to the project. The library path is already set before so we don't have to specify the exact file path again. We also want to install the executable to specified directory after build. So here is the installing script too:

```
TARGET_LINK_LIBRARIES(${EXAMPLE_NAME}
  debug osg${CMAKE_DEBUG_POSTFIX} optimized osg
  debug osgUtil${CMAKE_DEBUG_POSTFIX} optimized osgUtil
  debug osgViewer${CMAKE_DEBUG_POSTFIX} optimized osgViewer
  debug osgDB${CMAKE_DEBUG_POSTFIX} optimized osgDB
  debug osgGA${CMAKE_DEBUG_POSTFIX} optimized osgGA
  debug OpenThreads${CMAKE_DEBUG_POSTFIX}
    optimized OpenThreads
)
INSTALL(TARGETS ${EXAMPLE_NAME} RUNTIME DESTINATION
  ${CMAKE_INSTALL_PREFIX}/bin)
```

6. Now use `cmake-gui` to open it and generate the makefiles or solution files as below. Now we have a cool enough framework for creating cross-platform applications in the following chapters!



How it works...

Unfortunately, we don't have space to introduce CMake variables, functions, and grammars. It really requires some hundred pages to make you a CMake expert. Besides the CMake official website, the OSG source code also provides some good code snippets. And you may find a large amount of other information on the Internet, simply using `CMake` or `CMakeLists` as the searching keyword.

The book "*Mastering CMake*", *Ken Martin* and *Bill Hoffman*, *Kitware, Inc*., is also valuable for reading.

2
Designing the Scene Graph

In this chapter, we will cover:

- Using smart and observer pointers
- Sharing and cloning objects
- Computing the world bounding box of any node
- Creating a running car
- Mirroring the scene graph
- Designing a breadth-first node visitor
- Implementing a background image node
- Making your node always face the screen
- Using draw callbacks to execute NVIDIA Cg functions
- Implementing a compass node

Introduction

In this chapter, we will show a series of interesting topics about configuring the scene graph and implementing some special effects with simple but effective methods. We assume that you have already understood the concepts of **group nodes**, **leaf nodes** (**geodes**), and parent and child interfaces. If not, you can read the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, first. So the main objective of the following few recipes will be to use nodes and callbacks in a flexible way.

Before we start, it is necessary to prepare some common functions and classes for use. These utilities can be used to quickly create nodes, event handlers, and other scene objects. As we have just started with the book, we will learn to handle some real programming cases; there will be only three components in the "common use" domain. The first one is the `createHUDCamera()` function:

```
osg::Camera* createHUDCamera( double left, double right, double
bottom, double top )
{
  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  camera->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
  camera->setClearMask( GL_DEPTH_BUFFER_BIT );
  camera->setRenderOrder( osg::Camera::POST_RENDER );
  camera->setAllowEventFocus( false );
  camera->setProjectionMatrix(
    osg::Matrix::ortho2D(left, right, bottom, top) );
  camera->getOrCreateStateSet()->setMode(
    GL_LIGHTING, osg::StateAttribute::OFF );
  return camera.release();
}
```

This function will create an ordinary camera node which will be rendered on the top after the main scene is drawn. It can be used to display some **heads-up display** (**HUD**) texts and images. You may visit the following link to learn more about the concept of HUD: `http://en.wikipedia.org/wiki/HUD_(video_gaming)`

And it is necessary to have a function for creating HUD texts. Its content is shown in the following block of code:

```
osg::ref_ptr<osgText::Font> g_font = osgText::readFontFile("fonts/
  arial.ttf");
osgText::Text* createText( const osg::Vec3& pos, const std::string&
  content, float size )
{
  osg::ref_ptr<osgText::Text> text = new osgText::Text;
  text->setDataVariance( osg::Object::DYNAMIC );
  text->setFont( g_font.get() );
  text->setCharacterSize( size );
  text->setAxisAlignment( osgText::TextBase::XY_PLANE );
  text->setPosition( pos );
  text->setText( content );
  return text.release();
}
```

Of course, it is already designed to work with the HUD camera node smoothly.

The last useful tool to implement is a picking-up handler with which we can quickly select a node or drawable displayed on the screen and retrieve information and parent node paths. It must be derived for practical use.

```
class PickHandler : public osgGA::GUIEventHandler
{
public:
  // This virtual method must be overrode by subclasses.
  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::Intersection& ) = 0;

    virtual bool handle( const osgGA::GUIEventAdapter& ea,
      osgGA::GUIActionAdapter& aa )
    {
      if ( ea.getEventType()!=osgGA::GUIEventAdapter::RELEASE
        ||ea.getButton()!=osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON
        ||!(ea.getModKeyMask()&osgGA::GUIEventAdapter::MODKEY_CTRL) )
      return false;

      osgViewer::View* viewer = dynamic_cast<osgViewer::View*>(&aa);
      if ( viewer )
      {
        osg::ref_ptr<osgUtil::LineSegmentIntersector>
          intersector = new osgUtil::LineSegmentIntersector
            (osgUtil::Intersector::WINDOW, ea.getX(), ea.getY());
        osgUtil::IntersectionVisitor iv( intersector.get() );
          viewer->getCamera()->accept( iv );

          if ( intersector->containsIntersections() )
            {
              osgUtil::LineSegmentIntersector::Intersection&
                result = *(intersector->getIntersections().begin());
              doUserOperations( result );
            }
      }
      return false;
    }
};
```

When you are clicking on the screen to select an object, you must press *Ctrl* at the same time to distinguish the selecting operation with normal scene navigating.

All these utilities will be placed in the `osgCookbook` namespace to avoid ambiguous issues. And we will directly call them in a unified form such as the `osgCookBook::createText()` method, assuming that you have already put them in a suitable place for use.

Also, go through the code bundle of this chapter for the source code.

Using smart and observer pointers

You should be familiar with the smart pointer `osg::ref_ptr<>`, which manages allocated objects using **reference counting**, and deletes them when the counting number decreases to 0. In this case, `osg::ref_ptr<>` is actually a **strong pointer** that contributes to the life of the managed object.

This time we will come across another type of smart pointer, that is, the **weak pointer**. A weak pointer, that is, `osg::observer_ptr<>` in the OSG core library, doesn't own the object and won't change the reference counting number irrespective of it being attached or detached. But it has a property that when the object is deleted or recycled, it will be notified and set to NULL automatically to avoid using invalid pointers.

How to do it...

An interactive program will be created in this recipe to show the main feature of the `osg::observer_ptr<>` template class that checks if the pointer is valid or not spontaneously:

1. Include necessary headers:

```
#include <osg/ShapeDrawable>
#include <osg/Geode>
#include <osgViewer/Viewer>
```

2. We are going to have a `RemoveShapeHandler` class derived from the `osgCookBook::PickHandler` auxiliary class. It simply checks and removes the picked drawable from its parent:

```
class RemoveShapeHandler : public osgCookBook::PickHandler
{
  virtual void doUserOperations( osgUtil::LineSegmentIntersector::
    Intersection& result )
  {
    if ( result.nodePath.size()>0 )
    {
      osg::Geode* geode = dynamic_cast<osg::Geode*>(
        result.nodePath.back() );
      if ( geode ) geode->removeDrawable(
        result.drawable.get() );
    }
  }
};
```

3. The `ObserveShapeCallback` class is used here to keep two drawables with the `osg::observer_ptr<>` template class. As a weak pointer, it will automatically reset the pointer to NULL if the referenced object is recycled for some reason. And the member `_text` variable here will record these changes and display them on the screen:

```
class ObserveShapeCallback : public osg::NodeCallback
{
public:
  virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
  {
    std::string content;
    if ( _drawable1.valid() ) content += "Drawable 1; ";
    if ( _drawable2.valid() ) content += "Drawable 2; ";
    if ( _text.valid() ) _text->setText( content );
  }

  osg::ref_ptr<osgText::Text> _text;
  osg::observer_ptr<osg::Drawable> _drawable1;
  osg::observer_ptr<osg::Drawable> _drawable2;
};
```

4. In the main entry, we will first build the scene graph. It contains a HUD camera with text, and two basic drawables for use in this experiment:

```
// Create the text and place it in an HUD camera
osgText::Text* text = osgCookBook::createText(osg::Vec3(
  50.0f, 50.0f, 0.0f), "", 10.0f);
osg::ref_ptr<osg::Geode> textGeode = new osg::Geode;
textGeode->addDrawable( text );

osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0, 800, 0, 600);
hudCamera->addChild( textGeode.get() );

// Create two simple shapes and add both, as well as the camera,
// to the root node
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( new osg::ShapeDrawable(new
  osg::Box(osg::Vec3(-2.0f,0.0f,0.0f), 1.0f)) );
geode->addDrawable( new osg::ShapeDrawable(new osg::Sphere(osg::Ve
  c3(2.0f,0.0f,0.0f), 1.0f)) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( hudCamera.get() );
root->addChild( geode.get() );
```

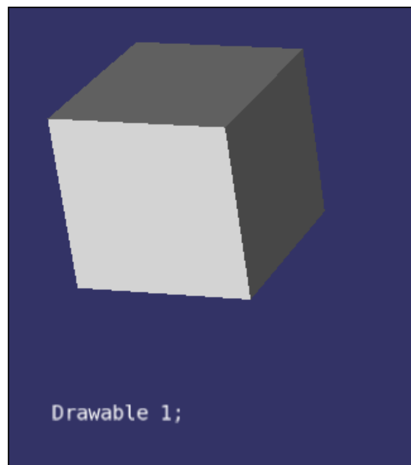
5. Create the update callback for the root node (or any other node in this case). Set up its public member variables in the following way:

```
osg::ref_ptr<ObserveShapeCallback> observerCB =
  new ObserveShapeCallback;
observerCB->_text = text;
observerCB->_drawable1 = geode->getDrawable(0);
observerCB->_drawable2 = geode->getDrawable(1);
root->addUpdateCallback( observerCB.get() );
```

6. Add the `RemoveShapeHandler` instance for interacting with drawables and start the viewer:

```
osgViewer::Viewer viewer;
viewer.addEventHandler( new RemoveShapeHandler );
viewer.setSceneData( root.get() );
return viewer.run();
```

7. Press *Ctrl* and click on one of the shapes to remove it from the scene graph, and you will see that the text shown at the bottom is changed immediately. The observer pointer that has already found the shape is not referenced by other objects anymore and, therefore, resets itself to avoid **dangling pointer** problems.



How it works...

The `RemoveShapeHandler` here re-implements the `doUserOperation()` method of its parent class to check if a shape is picked, and un-references it from the parent `osg::Geode` node. As no other smart pointers are referencing the shape, it is actually deleted from the system memory. The `osg::observer_ptr<>` template class, as a weak pointer will only observe the node's allocation and destroy it, and will automatically switch its data to NULL to prevent further improper usages.

The weak pointer is a great feature if we are going to observe or use a node in callbacks or user processes, without adding redundant references to it. Using a **raw pointer** is troublesome here because you have to always ensure the object is still valid; otherwise, your program may get crashed at once.

In multi-threaded applications, it is safe to use the `lock()` method to convert the weak pointer to a temporary strong pointer, to prevent synchronous object deletion in other threads. The code segments could be as follows:

```
// Define a member variable using osg::observer_ptr<>.
osg::observer_ptr<osg::Node> _memberNode;

// In a thread, when we want to obtain the member node.
osg::ref_ptr<osg::Node> tempRefOfNode;
if ( _memberNode.lock(tempRefOfNode) )
{
  osg::Node* realNode = tempRefOfNode.get();
  // Do something to the realNode.
  // Don't worry if it is unreferenced or deleted in
  // other threads, because tempRefOfNode can ensure
  // it works during the lifetime of the smart pointer.
}
```

There's more...

You may refer to the **Boost** library and read some more about its `shared_ptr` (strong pointer) and `weak_ptr` implementations at the following sites:

`http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/shared_ptr.htm`

`http://www.boost.org/doc/libs/1_46_1/libs/smart_ptr/weak_ptr.htm`

And the **MSDN** site also contains the similar classes for use:

`http://msdn.microsoft.com/en-us/library/bb982026.aspx`

`http://msdn.microsoft.com/en-us/library/bb982126.aspx`

Sharing and cloning objects

The sharing of nodes and drawables is an important optimization for a huge 3D application based on OSG. But sometimes, duplicating a node without sharing any memory chunks between the previous node and the new one is also useful for handling user data. In this example, we will show both implementations in one interactive program and explain the main difference of the scene graphs we have.

How to do it...

We will clone a simple ball shape twice, each with a different mechanism (**shallow copy** or **deep copy**). The user can press *Ctrl* and click on a ball to change its color. Shallow copied balls will change together because they point to the same memory address, but a deep copied one will not.

1. Include necessary headers:

```
#include <osg/ShapeDrawable>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osgViewer/Viewer>
```

2. This time we want to pick up any drawable and change its color if possible. The `SetShapeColorHandler` class here will do this for us. Every time we choose an `osg::ShapeDrawable` object, its color will be inverted. Thus we can quickly find out all the nodes that are sharing the same drawable:

```
class SetShapeColorHandler : public osgCookBook::PickHandler
{
  virtual void doUserOperations( osgUtil::LineSegmentIntersector
    ::Intersection& result )
  {
    osg::ShapeDrawable* shape = dynamic_cast<osg::ShapeDrawable*>
      ( result.drawable.get() );
    if ( shape ) shape->setColor( osg::Vec4(
      1.0f, 1.0f, 1.0f, 2.0f) - shape->getColor() );
  }
};
```

3. The `createMatrixTransform()` function here will just create a transformation node at a specified position, and add an `osg::Geode` node as its child:

```
osg::Node* createMatrixTransform( osg::Geode* geode,
  const osg::Vec3& pos )
{
  osg::ref_ptr<osg::MatrixTransform> trans =
    new osg::MatrixTransform;
  trans->setMatrix( osg::Matrix::translate(pos) );
  trans->addChild( geode );
  return trans.release();
}
```

4. In the main entry, create a basic sphere and disable the use of display lists on it. That is because its color may be dynamically changed later in the simulation loop:

```
osg::ref_ptr<osg::ShapeDrawable> shape = new osg::ShapeDrawable(
  new osg::Sphere );
shape->setColor( osg::Vec4(1.0f, 1.0f, 0.0f, 1.0f) );
shape->setDataVariance( osg::Object::DYNAMIC );
shape->setUseDisplayList( false );
```

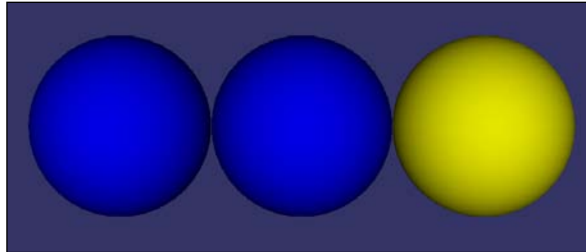
5. Now we will demonstrate different cloning types here. The original `geode1` including the changeable sphere is duplicated into `geode2` (shallow copy) and `geode3` (deep copy). And all the three nodes are added to the root node with a proper translation:

```
osg::ref_ptr<osg::Geode> geode1 = new osg::Geode;
geode1->addDrawable( shape.get() );

osg::ref_ptr<osg::Geode> geode2 = dynamic_cast<osg::Geode*>(
  geode1->clone(osg::CopyOp::SHALLOW_COPY) );
osg::ref_ptr<osg::Geode> geode3 = dynamic_cast<osg::Geode*>(
  geode1->clone(osg::CopyOp::DEEP_COPY_ALL) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( createMatrixTransform(geode1.get(),
  osg::Vec3(0.0f, 0.0f, 0.0f)) );
root->addChild( createMatrixTransform(geode2.get(),
  osg::Vec3(-2.0f, 0.0f, 0.0f)) );
root->addChild( createMatrixTransform(geode3.get(),
  osg::Vec3(2.0f, 0.0f, 0.0f)) );
```

6. Before starting the viewer, don't forget to add the handler, with which you may click on the spheres to make the world colorful:

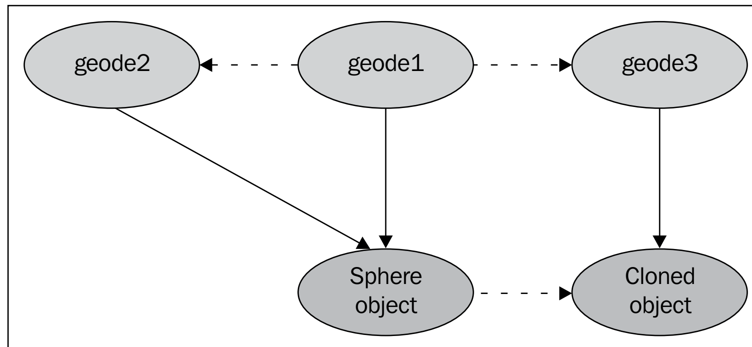
```
osgViewer::Viewer viewer;
viewer.addEventHandler( new SetShapeColorHandler );
viewer.setSceneData( root.get() );
return viewer.run();
```

7. You will soon realize the fact that `geode1` (at the middle) and `geode2` (at the left side) will act together when you pick either of them (press *Ctrl* at the same time). But `geode3` (at the right side) is independent all the time.



How it works...

The `geode3` node is deep copied so that if its member variables point to any objects, the allocated memories of these objects will be copied too. Instead, a shallow copy means the copied member pointers will share the same memory chunks with original ones. The difference between them can be seen in the following diagram:



The `clone()` method here will allocate a new object of the same type by calling the copy constructor. This is just a one-line implementation:

```
virtual osg::Object* clone(const osg::CopyOp& copyop) const
  { return new YourClass(*this, copyop); }
```

`YourClass` here means any class name used as OSG scene objects. And you can read the content of `src/osg/CopyOp.cpp` of the OSG source code for details about the second argument `copyop`.

Computing the world bounding box of any node

You may have already known that a node uses bounding sphere instead of the axis-aligned box by learning some other books and tutorials. You may also learn that the `osg::ComputeBoundsVisitor` class can compute the bounding box by traversing the node and its sub-graph. But in this recipe, we will introduce some more details about the whole computation process and the **local-to-world** transformation used here.

How to do it...

We will create a simple scene with animations and compute the bounding box of some objects in realtime, with the resultant bounding box displayed.

1. Include necessary headers:

```
#include <osg/ComputeBoundsVisitor>
#include <osg/ShapeDrawable>
#include <osg/AnimationPath>
#include <osg/MatrixTransform>
#include <osg/PolygonMode>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The `BoundingBoxCallback` class can compute the real-world bounding box for us. We will have to pass a list of nodes to it and expand the world box by adding the local bound of each node one by one:

```
class BoundingBoxCallback : public osg::NodeCallback
{
public:
  virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
  {

  }

  osg::NodePath _nodesToCompute;
};
```

3. In the `operator()` implementation, we will do the trick: The `osg::ComputeBoundsVisitor` class calculates the bounding box of a node in its parent's reference frame. Then we must re-compute the vertices in the world coordinates before adding them to the world box variable using the `localToWorld` matrix:

```
osg::BoundingBox bb;
for ( unsigned int i=0; i<_nodesToCompute.size(); ++i )
{
  osg::Node* node = _nodesToCompute[i];
  osg::ComputeBoundsVisitor cbbv;
  node->accept( cbbv );

  osg::BoundingBox localBB = cbbv.getBoundingBox();
  osg::Matrix localToWorld = osg::computeLocalToWorld(
    node->getParent(0)->getParentalNodePaths()[0] );
  for ( unsigned int i=0; i<8; ++i )
  bb.expandBy( localBB.corner(i) * localToWorld );
}
```

4. Apply the result (world coordinates) to the transformation node and make it visible in the whole scene:

```
osg::MatrixTransform* trans =
  static_cast<osg::MatrixTransform*>(node);
trans->setMatrix(
  osg::Matrix::scale(bb.xMax()-bb.xMin(), bb.yMax()-bb.yMin(),
    bb.zMax()-bb.zMin()) *
  osg::Matrix::translate(bb.center()) );
```

5. We would like to create a function named `createAnimationPath()` for creating animation path here, and make the computation of the bounding box more dynamic:

```
osg::AnimationPath* createAnimationPath( float radius, float time )
{
  osg::ref_ptr<osg::AnimationPath> path =
    new osg::AnimationPath;
  path->setLoopMode( osg::AnimationPath::LOOP );

  unsigned int numSamples = 32;
  float delta_yaw = 2.0f * osg::PI/((float)numSamples - 1.0f);
  float delta_time = time / (float)numSamples;
  for ( unsigned int i=0; i<numSamples; ++i )
  {
    float yaw = delta_yaw * (float)i;
```

```
    osg::Vec3 pos( sinf(yaw)*radius, cosf(yaw)*radius, 0.0f );
    osg::Quat rot( -yaw, osg::Z_AXIS );
    path->insert( delta_time * (float)i,
      osg::AnimationPath::ControlPoint(pos, rot) );
  }
  return path.release();
}
```

6. In the main entry, we first create the scene with a Cessna flying in a circle, a truck, and the example terrain. All the model files can be found in the OSG sample dataset:

```
osg::ref_ptr<osg::MatrixTransform> cessna =
  new osg::MatrixTransform;
cessna->addChild(
  osgDB::readNodeFile("cessna.osgt.0,0,90.rot") );

osg::ref_ptr<osg::AnimationPathCallback> apcb =
  new osg::AnimationPathCallback;
apcb->setAnimationPath( createAnimationPath(50.0f, 6.0f) );
cessna->setUpdateCallback( apcb.get() );

osg::ref_ptr<osg::MatrixTransform> dumptruck =
  new osg::MatrixTransform;
dumptruck->addChild( osgDB::readNodeFile("dumptruck.osgt") );
dumptruck->setMatrix( osg::Matrix::translate(0.0f, 0.0f, -100.0f) );

osg::ref_ptr<osg::MatrixTransform> models =
  new osg::MatrixTransform;
models->addChild( cessna.get() );
models->addChild( dumptruck.get() );
models->setMatrix( osg::Matrix::translate(0.0f, 0.0f, 200.0f) );
```

7. The Cessna and the truck will be added for computing the world bounding box in a complex way:

```
osg::ref_ptr<BoundingBoxCallback> bbcb =
  new BoundingBoxCallback;
bbcb->_nodesToCompute.push_back( cessna.get() );
bbcb->_nodesToCompute.push_back( dumptruck.get() );
```

8. Construct the box shape for representing the bound in the scene graph:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( new osg::ShapeDrawable(new osg::Box) );

osg::ref_ptr<osg::MatrixTransform> boundingBoxNode =
  new osg::MatrixTransform;
```

```
boundingBoxNode->addChild( geode.get() );
boundingBoxNode->setUpdateCallback( bbcb.get() );
boundingBoxNode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::PolygonMode(osg::PolygonMode::FRONT_AND_BACK,
    osg::PolygonMode::LINE) );
boundingBoxNode->getOrCreateStateSet()->setMode(
  GL_LIGHTING, osg::StateAttribute::OFF );
```

9. Build the scene and start the viewer:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( models.get() );
root->addChild( osgDB::readNodeFile("lz.osgt") );
root->addChild( boundingBoxNode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

10. You will see that the wireframe box is changing its size while the Cessna is moving. But it exactly contains the Cessna and the truck models all the time, as shown in the following screenshot:

11. Try commenting the line inputted in step 3 in the following type:

```
osg::Matrix localToWorld;/* = osg::computeLocalToWorld(
  node->getParent(0)->getParentalNodePaths()[0] ); */
```

Then rebuild to see the difference. Can you figure out the reason for the change?

How it works...

Every node in the scene graph has its own local coordinate system. When you translate and rotate a **transformation node**, it means that you change its position and attitude in its parent's coordinate system. This makes all the transformations occur in local space rather than the world one. And the matrix applied to the node can also be treated as the transpose matrix that maps the node space to its parent's space.

To compute the world coordinates of a specified point, we have to first find out the local space it lives in, and then fetch the node's parent, parent's parent, and so on, until we reach the scene root. Then we will have a node path from the root to the node containing the point. With the node path, we can multiply all the transpose matrices and get a complete local-to-world matrix.

The collection of parental node paths is done with the `getParentalNodePaths()` method. It has multiple paths because an OSG node may have more than one parent node. And to compute the local-to-world matrix, use `osg::computeLocalToWorld()` function directly with the node path as argument. There is another function named `osg::computeWorldToLocal()`, which is for computing the local representation of a point in world space.

Creating a running car

The goal here is easy to understand but not easy to achieve. It requires exactly another book to tell how to make a realistic enough car model and load it into the scene graph efficiently, as well as how to assemble components and have the wheels rotating. So we have to simplify the problem here—we are going to implement some very ugly car parts only with basic shapes, and demonstrate the use of the scene graph in the assembly process.

How to do it...

All we need to do here is use the transformation node, which is one the most basic classes in the OSG library. But it is not easy to be skillful with it.

1. Include necessary headers:

```
#include <osg/ShapeDrawable>
#include <osg/AnimationPath>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The convenient function `createTransformNode()` will create a transformation node for every part shape we have:

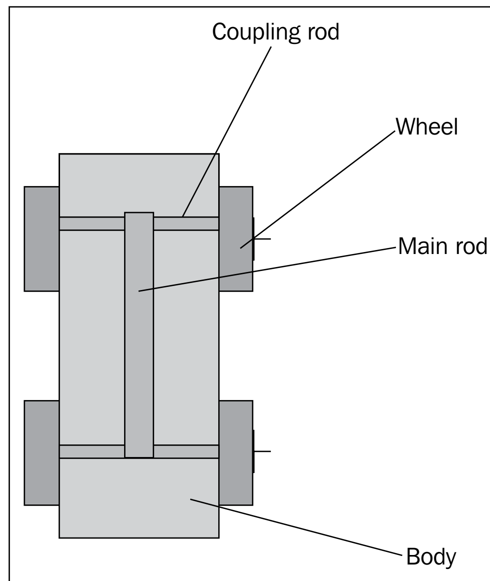
```
osg::MatrixTransform* createTransformNode( osg::Drawable*
  shape, const osg::Matrix& matrix )
{
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( shape );

  osg::ref_ptr<osg::MatrixTransform> trans =
    new osg::MatrixTransform;
  trans->addChild( geode.get() );
  trans->setMatrix( matrix );
  return trans.release();
}
```

3. We want to make the wheel turn rapidly using an animation path callback. Note that we also added a very small offset on the Z axis while rotating the wheel. It makes the animation a little more realistic here:

```
osg::AnimationPathCallback* createWheelAnimation(
  const osg::Vec3& base )
{
  osg::ref_ptr<osg::AnimationPath> wheelPath =
    new osg::AnimationPath;
  wheelPath->setLoopMode( osg::AnimationPath::LOOP );
  wheelPath->insert( 0.0, osg::AnimationPath::ControlPoint(
    base, osg::Quat()) );
  wheelPath->insert( 0.01, osg::AnimationPath::ControlPoint(
    base + osg::Vec3(0.0f, 0.02f, 0.0f), osg::Quat(
      osg::PI_2, osg::Z_AXIS)) );
  wheelPath->insert( 0.02, osg::AnimationPath::ControlPoint(
    base + osg::Vec3(0.0f,-0.02f, 0.0f), osg::Quat(
      osg::PI, osg::Z_AXIS)) );
```

```
  osg::ref_ptr<osg::AnimationPathCallback> apcb =
    new osg::AnimationPathCallback;
  apcb->setAnimationPath( wheelPath.get() );
  return apcb.release();
}
```

4. In the main entry, there are mainly four parts of our ugly car: four wheels, the coupling rod between each two wheels, the car body (here we only use a box to represent it) and the main rod which connects all of them, as shown in the following diagram:



5. By default, the geometric centers of each part's prototype are all placed at the origin point; and the height directions of the cylinders are the Z-axis:

```
// The prototype of the main rod
osg::ref_ptr<osg::ShapeDrawable> mainRodShape =
  new osg::ShapeDrawable( new osg::Cylinder(
    osg::Vec3(), 0.4f, 10.0f) );
// The prototype of the coupling (wheel) rod
osg::ref_ptr<osg::ShapeDrawable> wheelRodShape =
  new osg::ShapeDrawable( new osg::Cylinder(
    osg::Vec3(), 0.4f, 8.0f) );
// The prototypes of the wheel and the car body
osg::ref_ptr<osg::ShapeDrawable> wheelShape =
  new osg::ShapeDrawable( new osg::Cylinder(
    osg::Vec3(), 2.0f, 1.0f) );
```

```
osg::ref_ptr<osg::ShapeDrawable> bodyShape =
  new osg::ShapeDrawable( new osg::Box(
    osg::Vec3(), 6.0f, 4.0f, 14.0f) );
```

6. The wheels will be moved to the ends of the coupling rod:

```
osg::MatrixTransform* wheel1 = createTransformNode(
  wheelShape.get(), osg::Matrix::translate(0.0f, 0.0f,-4.0f) );
wheel1->setUpdateCallback(
  createWheelAnimation(osg::Vec3(0.0f, 0.0f,-4.0f)) );

osg::MatrixTransform* wheel2 = createTransformNode(
  wheelShape.get(), osg::Matrix::translate(0.0f, 0.0f, 4.0f) );
wheel2->setUpdateCallback(
  createWheelAnimation(osg::Vec3(0.0f, 0.0f, 4.0f)) );
```

7. And the coupling rod itself will be rotated and moved to the end of the main rod too:

```
osg::MatrixTransform* wheelRod1 = createTransformNode(
  wheelRodShape.get(),
osg::Matrix::rotate(osg::Z_AXIS, osg::X_AXIS) *
  osg::Matrix::translate(0.0f, 0.0f,-5.0f) );
wheelRod1->addChild( wheel1 );
wheelRod1->addChild( wheel2 );
```

8. For another coupling rod, we will directly copy from the transformation node `wheelRod1`. It is good to do a shallow copy here as the child wheels and animations will be shared. After moving the cloned rod to a suitable place, now we can have a complete wheel system:

```
osg::MatrixTransform* wheelRod2 =
  static_cast<osg::MatrixTransform*>(
    wheelRod1->clone(osg::CopyOp::SHALLOW_COPY) );
wheelRod2->setMatrix( osg::Matrix::rotate(osg::Z_AXIS,
  osg::X_AXIS) * osg::Matrix::translate(0.0f, 0.0f, 5.0f) );
```

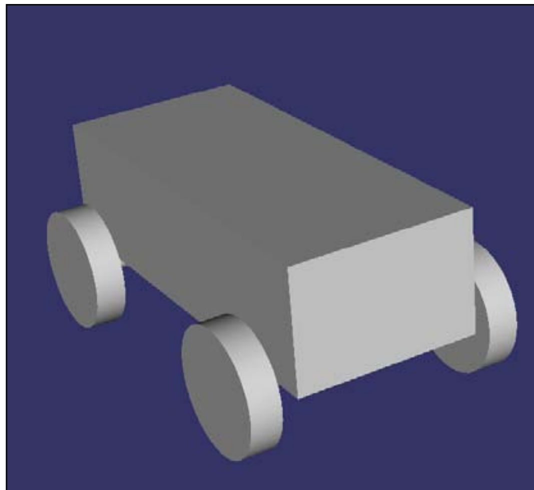
9. Finally, move the car body onto the main rod and finish the assembly work. All three parts should be added to the main rod node to make sure they are under its local coordinates:

```
osg::MatrixTransform* body = createTransformNode(
  bodyShape.get(), osg::Matrix::translate(0.0f, 2.2f, 0.0f) );

osg::MatrixTransform* mainRod = createTransformNode(
  mainRodShape.get(), osg::Matrix::identity() );
mainRod->addChild( wheelRod1 );
mainRod->addChild( wheelRod2 );
mainRod->addChild( body );
```

10. Create the root node and start the viewer now:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( mainRod );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

11. You will see the car running in the viewer. Of course, it has no textures, doors, windows, or aerodynamic bodyworks. But, why don't you just create some beautiful models in some other software like 3dsmax, Maya, or Blender3D, and replace the basic shapes used here? Try it if you have any interest in building a better-looking scene.

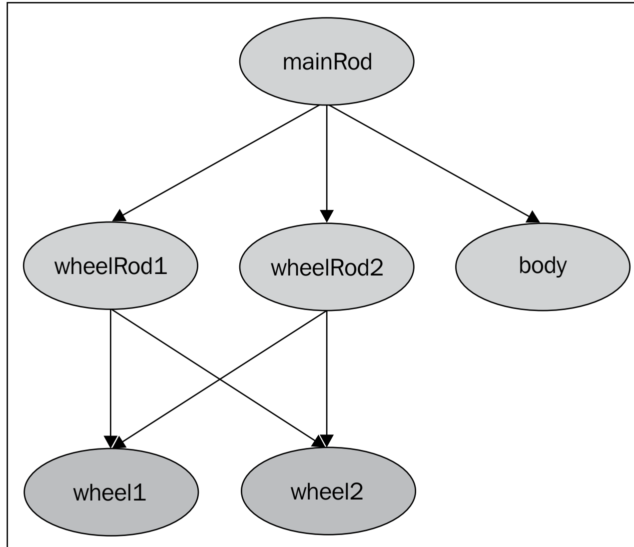


How it works...

Although the result is not so exciting and refined, it should be a good example for demonstrating the use of local coordinates, as well as the basic concepts of character bones (we were working on a car's bones in the previous section).

Of course there are more ways to implement such a composite car model, and you will be able to import some beautiful models to replace our basic ones. Just try it by yourselves.

The structure of the recipe's scene graph is shown in the following diagram:



Mirroring the scene graph

Mirroring the scene graph, or in another words, putting a scene inside a mirror as the "reflection", can also be done by specifying another transformation node as the parent of the origin scene. It requires rendering everything for a second time, and can be integrated with some **render-to-texture** techniques to simulate real mirrors, water reflections, shadows, and some other reflective effects. The solution described here will be used again in *Chapter 6* to create simple water effects.

How to do it...

The following code will be short but can be reused later in other chapters.

1. Include necessary headers:

```
#include <osg/ClipNode>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
Load a model into the scene graph first:
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene = osgDB::readNodeFiles(
  arguments );
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

2. The next important step is to work out the mirror matrix. We are going to make a mirror of the model against the `XOY` plane by flipping it upside down, as well as a small translation along the Z axis too, to represent the height of the mirror. The `osg::Matrix::scale()` function here will put the reflected model opposite on the Z axis, and the `osg::Matrix::translate()` function is used to set the scale pivot point:

```
float z = -10.0f;
osg::ref_ptr<osg::MatrixTransform> reverse =
  new osg::MatrixTransform;
reverse->preMult(osg::Matrix::translate(0.0f, 0.0f, -z) *
  osg::Matrix::scale(1.0f, 1.0f, -1.0f) *
  osg::Matrix::translate(0.0f, 0.0f, z) );
reverse->addChild( scene.get() );
```

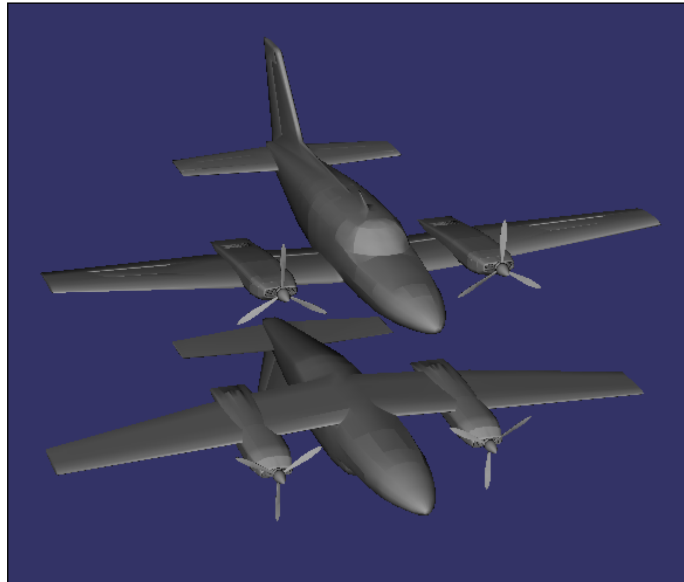
3. Enable clipping to remove anything that is poking out of the mirrored graph through the mirror. It may have no effect here but will help later when we are working on the water simulation example:

```
osg::ref_ptr<osg::ClipPlane> clipPlane = new osg::ClipPlane;
clipPlane->setClipPlane( 0.0, 0.0, -1.0, z );
clipPlane->setClipPlaneNum( 0 );

osg::ref_ptr<osg::ClipNode> clipNode = new osg::ClipNode;
clipNode->addClipPlane( clipPlane.get() );
clipNode->addChild( reverse.get() );
```

4. Now add both the origin scene and the reversed one to the root node and start the viewer:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( scene.get() );
root->addChild( clipNode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

5. The result is shown in the following screenshot. It may not be interesting enough at present, but you will soon find that it is the basis of some other complex effects such as water reflection and shadow implementations.



There's more...

If you have more interests in implementing an inverse scene, there are some more examples for you to understand, for instance, the NeHe OpenGL tutorials at `http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=26`.

The OSG source code also provides a good one using the `osg::Stencil` class for stencil tests. Refer to `examples/osgreflect` for details.

Designing a breadth-first node visitor

In graphics applications, a **breadth-first-search** (**BFS**) is a search algorithm that begins at the root node and traverses all the neighboring nodes before it goes deeper. That is different from the default behavior of the `osg::NodeVisitor` class, which is **depth-first-search** (**DFS**). We will try to implement a BFS visitor in this section to show how to make changes to this basic OSG class for your own use.

How to do it...

First we have to declare a new node visitor class inherited from the `osg::NodeVisitor` class. The headers to be included here are:

```
#include <osg/NodeVisitor>
#include <deque>
```

Two virtual functions must be overrode to make the visitor work: One is the `reset()` method, which will reset all member variables to their initial states; the other one is the `apply()` method, which accepts `osg::Node` class as the input argument. All OSG nodes will be redirected to this method while traversing a scene graph, so we will place our own `traverseBFS()` here for the purpose of creating a BFS visitor:

```
class BFSVisitor : public osg::NodeVisitor
{
public:
  BFSVisitor() { setVisitorType(TRAVERSE_ALL_CHILDREN); }

  virtual void reset() { _pendingNodes.clear(); }
  virtual void apply( osg::Node& node ) { traverseBFS(node); }

protected:
  virtual ~BFSVisitor() {}

  void traverseBFS( osg::Node& node );

  std::deque<osg::Node*> _pendingNodes;
};
```

The new traversal mechanism of scene graph is implemented as follows. We will find all the child nodes and push them into a queue, and then handle the queue from the first. Actually the queue can be treated as a **FIFO** (**first in, first out**) pipe. Neighboring nodes will be explored and handled together, and nodes at a lower level should always wait until nodes at higher levels are finished, and so on:

```
void BFSVisitor::traverseBFS( osg::Node& node )
{
  osg::Group* group = node.asGroup();
  if ( !group ) return;

  for ( unsigned int i=0; i<group->getNumChildren(); ++i )
  {
    _pendingNodes.push_back( group->getChild(i) );
  }
```

```
    while ( _pendingNodes.size()>0 )
    {
      osg::Node* node = _pendingNodes.front();
      _pendingNodes.pop_front();
      node->accept(*this);
    }
  }
```

Now we can use the `BFSVisitor` class in practical work.

1. First include other necessary headers:

   ```
   #include <osgDB/ReadFile>
   #include <osgUtil/PrintVisitor>
   #include <iostream>
   ```

2. We would like to print each node's class name while traversing the scene graph:

   ```
   class BFSPrintVisitor : public BFSVisitor
   {
   public:
     virtual void apply( osg::Node& node )
     {
       std::cout << node.libraryName() << "::"
         <<node.className() << std::endl;
       traverseBFS(node);
     }
   };
   ```

3. In the main entry, let us read a model from file first:

   ```
   osg::ArgumentParser arguments( &argc, argv );
   osg::ref_ptr<osg::Node> root = osgDB::readNodeFiles( arguments );
   if ( !root ) root = osgDB::readNodeFile("osgcool.osg");
   ```

4. The `osgUtil::PrintVisitor` class is used here to show the traversal sequence of DFS visitors:

   ```
   std::cout << "DFS Visitor traversal: " << std::endl;
   osgUtil::PrintVisitor pv( std::cout );
   root->accept( pv );
   std::cout << std::endl;
   ```

5. Now use the new BFS visitor to print the node information. You may have to start a terminal and run the program in text mode:

   ```
   std::cout << "BFS Visitor traversal: " << std::endl;
   BFSPrintVisitor bpv;
   root->accept( bpv );
   return 0;
   ```

6. The comparative result is listed as follows. You can easily find differences between DFS and BFS visitors.

```
DFS Visitor traversal:
osg::Group
  osg::MatrixTransform
    osgParticle::ModularEmitter
    osgParticle::ModularEmitter
    osgParticle::ModularEmitter
  osgParticle::ParticleSystemUpdater
  osg::Geode

BFS Visitor traversal:
osg::Group
osg::MatrixTransform
osgParticle::ParticleSystemUpdater
osg::Geode
osgParticle::ModularEmitter
osgParticle::ModularEmitter
osgParticle::ModularEmitter
```

There's more...

The breadth-first searching can be used to find the shortest path between two nodes, or implement some other algorithms. In a word, it is not suitable for the updating and rendering processes, which encapsulate the OpenGL state transitions and local-to-world transformations in a tree-like structure. The depth-first solution, which goes as far as possible along each branch and then backtracks, is still preferred for most scene graph visitors, such as the `osg::NodeVisitor` class.

Some more information about breadth-first and depth-first searching can be found at the following links:

`http://en.wikipedia.org/wiki/Breadth-first_search`

`http://en.wikipedia.org/wiki/Depth-first_search`

Implementing a background image node

Maybe you have also tried to implement a background image before but failed. The difficulty here is that if you ever try to use an HUD system to apply a background image, which will always be rendered after the main scene, it's difficult to deal with the depth buffer values set by the main scene. Fortunately, in this recipe, we have a solution for this problem using the depth tests.

How to do it...

Specify any image as the background image and load an arbitrary scene, and check if it is displayed before the background to verify the correctness of our solution.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/Depth>
#include <osg/Texture2D>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. Load the background image and map it to a quadrangle geometry:

```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
osg::ref_ptr<osg::Image> image = osgDB::readImageFile(
  "Images/osg256.png" );
texture->setImage( image.get() );

osg::ref_ptr<osg::Drawable> quad =
  osg::createTexturedQuadGeometry( osg::Vec3(),
     osg::Vec3(1.0f, 0.0f, 0.0f), osg::Vec3(0.0f, 1.0f, 0.0f) );
quad->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, texture.get() );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( quad.get() );
```

3. Prepare an HUD camera for the background image. It must completely fill the screen so we have to use orthogonal projection here. Some other important points here include disabling culling on the camera and setting the clear mask to 0. That is because the background should never be culled, and it should neither affect color nor depth buffer generated by the main scene.

```
osg::ref_ptr<osg::Camera> camera = new osg::Camera;
camera->setCullingActive( false );
camera->setClearMask( 0 );
camera->setAllowEventFocus( false );
camera->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
camera->setRenderOrder( osg::Camera::POST_RENDER );
camera->setProjectionMatrix( osg::Matrix::ortho2D(
  0.0, 1.0, 0.0, 1.0) );
camera->addChild( geode.get() );
```

4. Prevent the background from being affected by the light, and set up the depth test values. We will explain the reason later:

```
osg::StateSet* ss = camera->getOrCreateStateSet();
ss->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
ss->setAttributeAndModes( new osg::Depth(
  osg::Depth::LEQUAL, 1.0, 1.0) );
```

5. Now add the background camera and any other scene to the root node and see what we have now:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( camera.get() );
root->addChild( osgDB::readNodeFile("cessna.osg") );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

6. Everything seems to work well, as shown in the following screenshot. Believe it or not, the most important line in the program is the addition of the state attribute `osg::Depth` here. Try hiding it and see what the difference is.

How it works...

The key of the implementation of background images can be concentrated into one line, that is, re-map the depth values of the background image to [1.0, 1.0].

This ensures that each depth value of the post-rendered background is 1.0, and it won't pass the depth test unless the original depth value is equal or greater than 1.0 (the latter is certainly impossible), as shown in the following code segment:

```
setAttributeAndModes( new osg::Depth(osg::Depth::LEQUAL,
  1.0, 1.0) );
```

So when will the original depth value be 1.0? The answer is obvious: It happens when there is nothing displayed! And the real meaning of a "background" is exactly what needs to be shown when there is no other scene object. So we have just finished the work in a perfect way now.

Making your node always face the screen

Make something face the screen? Yes, this is exactly what the `osg::Billboard` class has done for you, and the `osgText::Text` class has a similar feature that rotates the text to screen automatically. But this time we will work on a node, and show how to alter transformation nodes according to the global model-view matrix. The method used here can also be extended to implement other small functionalities, for instance, to show small XYZ axes for reference in a model editor window, or a front sight following the mouse in a shooting game.

How to do it...

This recipe will be simple enough for reading and understanding. But the usage of cull callbacks here may also help in the following chapters to implement some complex examples. Just keep it in mind or place a bookmark if you can.

1. Include necessary headers:

   ```
   #include <osg/MatrixTransform>
   #include <osgDB/ReadFile>
   #include <osgUtil/CullVisitor>
   #include <osgViewer/Viewer>
   ```

2. Declare a node callback and we will change the transformation matrix of specified node to make sure it is always facing the screen, which is actually a billboard node's behavior:

   ```
   class BillboardCallback : public osg::NodeCallback
   {
   public:
   ```

```
    BillboardCallback( osg::MatrixTransform* billboard )
    : _billboardNode(billboard) {}

    virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
    {
     ...
    }

protected:
    osg::observer_ptr<osg::MatrixTransform> _billboardNode;
};
```

3. In the `operator()` implementation, first be careful of the dynamic type casting. We are trying to convert the input-node visitor pointer to an `osgUtil::CullVisitor` object. It can only be retrieved in the cull traversal.

```
osgUtil::CullVisitor* cv =
  dynamic_cast<osgUtil::CullVisitor*>(nv);
if ( _billboardNode.valid() && cv )
{
  osg::Vec3d translation, scale;
  osg::Quat rotation, so;
  cv->getModelViewMatrix()->decompose( translation, rotation,
    scale, so );

  osg::Matrixd matrix( rotation.inverse() );
  _billboardNode->setMatrix( matrix );
}
traverse( node, nv );
```

This code segment decomposes the matrix into translation, rotation, scale vector, and scale orientation.

4. To make a node face the screen all the time, all we should do is remove the rotation component from the model-view matrix applying on it. That is why we set the inverse rotation matrix here. And this and the previous rotation component will cancel each other out during the matrix multiplication process.

5. In the main entry, load the Cessna model and add it to a transformation node, which will only accept the inverse rotation matrix:

```
osg::ref_ptr<osg::MatrixTransform> billboardNode =
  new osg::MatrixTransform;
billboardNode->addChild( osgDB::readNodeFile("cessna.osg") );
```

6. Add the billboard node and a terrain model for reference to the root node:

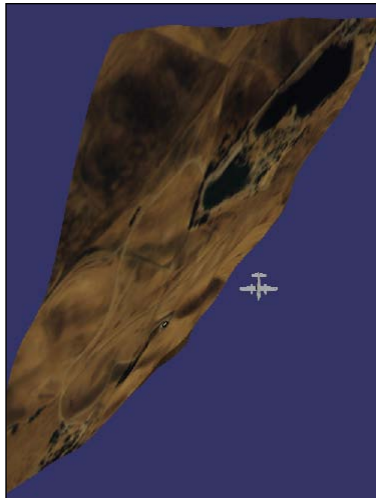
```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( billboardNode.get() );
root->addChild( osgDB::readNodeFile("lz.osg") );
root->addCullCallback(
  new BillboardCallback(billboardNode.get()) );
```

   Later we will explain the reason we put the `BillboardCallback` on the root node, rather than the billboard node itself.

7. Start the viewer:

```
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

8. The Cessna is still at the right place and has correct hiding relations with the terrain. But you will soon find that you can see only one side of the Cessna, as if it is 2D. That is to say, the Cessna is facing the screen now, as shown in the following screenshot:



How it works...

The difference between adding the callback to the root node and to `billboardNode` is the priority order of setting matrix and applying matrix. Let us look at the first case; when the callback is set to root, it will be executed when the **cull visitor** reaches the root node and call the `setMatrix()` method of the transformation node `billboardNode`. After that, when the cull visitor is traversing the node `billboardNode`, the transformation matrix will be applied to the billboard and become effective during the rendering. It leads to the corrected orientation (facing the screen) of the node.

But if we set the callback to the node `billboardNode` directly, some problems may appear. The newly set matrix won't work immediately in a cull callback. So the new orientation value can only take effect in the next frame. In fact, this will cause the model to twinkle, and thus lead to unexpected results.

There's more...

There are several types of visitors that may traverse the scene graph and trigger a callback. You may obtain them by dynamic type casting in the `traverse()` method of your custom node, or in the `operator()` of callbacks. The following table shows these node visitors, type enumerations (can be obtained by calling the `getVisitorType()` method), and descriptions:

Visitor	Type enumeration	Related callback	Description
`osgGA::EventVisitor`	`EVENT_VISITOR`	`setEventCallback()`	The event visitor
`osgUtil::UpdateVisitor`	`UPDATE_VISITOR`	`setUpdateCallback()`	The update visitor
`osgUtil::CullVisitor`	`CULL_VISITOR`	`setCullCallback()`	The cull visitor
`osgUtil::GLObjectsVisitor`	`NODE_VISITOR`	None	The visitor for compiling OpenGL objects
`osg::CollectOccludersVisitor`	`COLLECT_OCCLUDER_VISITOR`	None	The visitor for collecting culling occluders
`osgUtil::IntersectionVisitor`	`NODE_VISITOR`	None	The visitor for intersections

For other implementations of billboards, see the declarations of the `osg::Billboard` and `osg::AutoTransform` classes. And there are some related examples at `examples/osgforest` and `examples/osgautotransform` in the OSG source code too.

Using draw callbacks to execute NVIDIA Cg functions

The **Cg** language (C for Graphics) is a high-level shading language developed by NVIDIA. It is suitable for GPU programming and can support both the DirectX (**HLSL**) and OpenGL (**GLSL**) shader programs. It is widely used in modern PC games and 3D applications.

Of course, although we can't make use of any HLSL features of the Cg language, it is still worth integrating it with the OSG functionalities. Before considering using shader parameters, parameter buffers, **CgFX**, and other advanced Cg features, we could first attempt to run some very easy Cg programs. This time we will use `osg::Camera`'s draw callbacks for such purposes.

Getting ready

Review and download the Cg toolkit at the NVIDIA website first. It supports Linux, Mac OS X, and Windows systems too.

`http://developer.nvidia.com/cg-toolkit`

We don't have space to introduce the Cg grammar and example code here. You can check out some tutorials on the Internet.

The CMake script of your program should be modified to find Cg include directory and libraries. The following code segment should be an easy-to-read example here:

```
FIND_PATH(CG_INCLUDE_PATH Cg/cg.h)
FIND_LIBRARY(CG_GL_LIBRARY CgGL)
FIND_LIBRARY(CG_LIBRARY Cg)

INCLUDE_DIRECTORIES(${CG_INCLUDE_PATH })
TARGET_LINK_LIBRARIES(${EXAMPLE_NAME}
  ${CG_LIBRARY} ${CG_GL_LIBRARY})
```

How to do it...

Now let us first create the draw callbacks for rendering with Cg program states.

1. Include necessary headers and start to construct some classes for integrating Cg shading features:

   ```
   #include <Cg/cg.h>
   #include <Cg/cgGL.h>
   #include <osg/Camera>
   ```

2. The most important steps when using Cg programs and profiles are to enable them before actual drawing, and disable them after; this can enable specific shaders to work before real-drawing operations, and disable them after to make sure they won't affect other processing steps. To implement this with camera callbacks, you have to design a pre-drawing and a post-drawing callback, both with the same Cg variables. Therefore, we can just have a base callback class which manages a list of `CGprofile` and `CGprogram` objects:

```
class CgDrawCallback : public osg::Camera::DrawCallback
{
public:
  void addProfile( CGprofile profile ) {
    _profiles.push_back(profile); }
  void addCompiledProgram( CGprogram prog ) {
    _programs.push_back(prog); }

protected:
  std::vector<CGprofile> _profiles;
  std::vector<CGprogram> _programs;
};
```

3. The `CgStartDrawCallback` and `CgEndDrawCallback` classes will have different behaviors while handling the same Cg objects. Note that the `CgStartDrawCallback` class has an extra `_initialized` variable to help initialize the programs the first time it is executed:

```
class CgStartDrawCallback : public CgDrawCallback
{
public:
  CgStartDrawCallback() : _initialized(false) {}
  virtual void operator()( osg::RenderInfo&
    renderInfo ) const;

protected:
  mutable bool _initialized;
};

class CgEndDrawCallback : public CgDrawCallback
{
public:
  virtual void operator()( osg::RenderInfo&
    renderInfo ) const;
};
```

4. Here is the implementation of these two drawing callbacks. The two `operator()` methods here will be executed just before and after the drawing process of the camera's children:

```
void CgStartDrawCallback::operator()( osg::RenderInfo&
  renderInfo ) const
{
  if ( !_initialized )
  {
    // Load all Cg shader programs
```

```
    for ( unsigned int i=0; i<_programs.size(); ++i )
    cgGLLoadProgram( _programs[i] );
    _initialized = true;
  }

  // Bind the programs to current graphics context
  for ( unsigned int i=0; i<_programs.size(); ++i )
  cgGLBindProgram( _programs[i] );

  // Enable Cg profiles to work under specified devices
  for ( unsigned int i=0; i<_profiles.size(); ++i )
  cgGLEnableProfile( _profiles[i] );
}

void CgEndDrawCallback::operator()( osg::RenderInfo&
  renderInfo ) const
{
  // Disable profiles after the drawing
  for ( unsigned int i=0; i<_profiles.size(); ++i )
  cgGLDisableProfile( _profiles[i] );
}
```

5. After finishing the callback classes, now it's time to create a small example using OSG and NVIDIA Cg. First let us include the headers and create a very simple Cg program rendering vertex normal as final pixel colors:

```
static const char* cgProgramCode = {
  "struct app_input {\n"
    "float4 vertex : POSITION;\n"
    "float4 normal : NORMAL;\n"
  "};\n"

  "struct vertex_to_fragment {\n"
    "float4 position : POSITION;\n"
    "float3 normal3  : TEXCOORD0;\n"
  "};\n"

  "vertex_to_fragment vertex_main(app_input input)\n"
  "{\n"
    "vertex_to_fragment output;\n"
    "output.position = mul(glstate.matrix.mvp,
      input.vertex);\n"
    "output.normal3 = input.normal.xyz;\n"
    "return output;\n"
  "}\n"
```

```
    "float4 fragment_main(vertex_to_fragment input) : COLOR\n"
    "{\n"
      "float4 output = float4(input.normal3.x, input.normal3.y,
        input.normal3.z, 1.0);\n"
    "return output;\n"
    "}\n"
};
```

6. The Cg context must be global, and we will set up an error callback for any Cg-related problems:

```
CGcontext g_context;

void error_callback()
{
  CGerror lastError = static_cast<CGerror>( cgGetError() );
  OSG_WARN << "Cg error: " << cgGetErrorString(lastError)
    << std::endl;

  if ( lastError == CG_COMPILER_ERROR )
  OSG_WARN << std::string(cgGetLastListing(g_context))
    << std::endl;
}
```

7. In the main entry, first we load a model and allocate the two callbacks:

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> root = osgDB::readNodeFiles(
  arguments );
if ( !root ) root = osgDB::readNodeFile( "cow.osg" );

osg::ref_ptr<CgStartDrawCallback> preCB =
  new CgStartDrawCallback;
osg::ref_ptr<CgEndDrawCallback> postCB =
  new CgEndDrawCallback;
```

8. Initialize the viewer, and what is more important, initialize the graphics context by calling the `setUpViewInWindow()` method:

```
osgViewer::Viewer viewer;
viewer.getCamera()->setPreDrawCallback( preCB.get() );
viewer.getCamera()->setPostDrawCallback( postCB.get() );
viewer.setSceneData( root.get() );
viewer.setUpViewInWindow( 100, 100, 800, 600 );
```

9. Initialize Cg variables before adding them to the callback objects. Since the initialization process requires OpenGL context to be created and made current, we must get the graphics context used in the current camera and set up the internal OpenGL rendering context. Now you will understand why we should initialize the graphics context before:

```
CGprofile vertProfile, fragProfile;
CGprogram vertProg, fragProg;

osg::GraphicsContext* gc =
  viewer.getCamera()->getGraphicsContext();
if ( gc )
{
  gc->realize();
  gc->makeCurrent();

  g_context = cgCreateContext();
  cgSetErrorCallback( error_callback );

  vertProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
  vertProg = cgCreateProgram(
    g_context, CG_SOURCE, cgProgramCode, vertProfile,
      "vertex_main", NULL );

  fragProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
  fragProg = cgCreateProgram(
    g_context, CG_SOURCE, cgProgramCode, fragProfile,
      "fragment_main", NULL );

  gc->releaseContext();
}
```

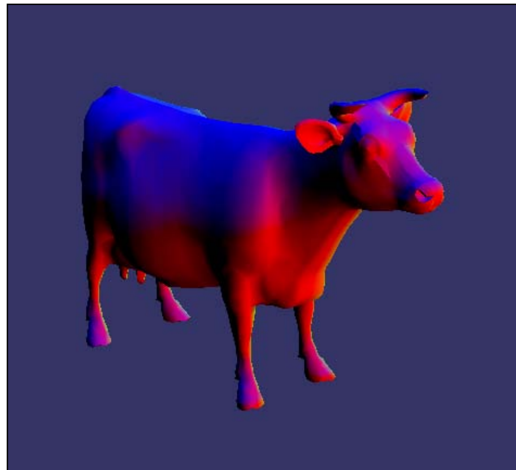
10. Add the initialized variables to the callbacks and start the viewer:

```
preCB->addProfile( vertProfile );
preCB->addProfile( fragProfile );
preCB->addCompiledProgram( vertProg );
preCB->addCompiledProgram( fragProg );

postCB->addProfile( vertProfile );
postCB->addProfile( fragProfile );
postCB->addCompiledProgram( vertProg );
postCB->addCompiledProgram( fragProg );

viewer.run();
```

11. Lastly, don't forget to release allocated Cg variables:

```
if ( gc )
{
  cgDestroyProgram( vertProg );
  cgDestroyProgram( fragProg );
  cgDestroyContext( g_context );
}
return 0;
```

12. OK, now what is the feeling of successfully integrating another shading language into OSG? If you are familiar with the Cg language, just try some other shaders and see if they could also work for you.



How it works...

A remarkable feature of this recipe is that it forces the construction of the graphics context and uses it to specify the OpenGL device and execute commands. You might remember there is a `createGraphicsContext()` method that creates new contexts according to user-specified traits. Yes, it could work here too. And the `setUpViewInWindow()` method actually executes this function internally with an auto-configured traits object.

There are some other `setUpView*()` methods, all of which can build a graphics context of different behaviors for use. You are able to retrieve the `osg::GraphicsContext` object and use it to execute OpenGL calls before the simulation starts.

So there are at least three ways to integrate OpenGL commands and libraries based on OpenGL with OSG now. The first is to derive and customize the `osg::Drawable` class. The second one is the pre-draw and post-draw callbacks defined in the `osg::Camera` class, which can manage some external states of child nodes but may cause OpenGL command coupling sometimes. The last one, that makes use of the rendering context directly, is applicable when you are going to make some initializations or tests; but can also cause serious threading problems in the multi-threaded mode because the same context may have to be used by other OSG graphic objects simultaneously.

Integration with other libraries is an interesting topic, so it will be mentioned again in other chapters. Try to find the pros and cons of the three methods discussed above by learning this and the following recipes, but use them at your own risk in different applications.

There's more...

Another good integration of OSG and NVIDIA Cg can be found in the third-party osgXI project (`http://sourceforge.net/projects/osgxi/`). Its osgCg module now supports Cg and CgFX by accepting them as state attributes.

Last but not least, to learn more about NVIDIA Cg, the free Cg tutorial is always preferred for reading, and can be found at `http://developer.nvidia.com/object/cg_tutorial_home.html`.

Implementing a compass node

Now, here comes the last recipe in this chapter, and we could do something really interesting this time. We will try to implement a compass and use it in a simple earth scene. A compass can help us identify the directions in a 3D world. And as far as we know, it makes our applications look professional and useful, if we are working on some 3D **geographic information systems** (**GIS**) or computer games.

How to do it...

1. Declare the `Compass` class. It contains a transformable dial plate and a needle. The orientation will be read and computed from the current view matrix of the main scene camera, which should be set before the simulation starts:

```
class Compass : public osg::Camera
{
public:
  Compass();
  Compass( const Compass& copy, osg::CopyOp
    copyop=osg::CopyOp::SHALLOW_COPY );
  META_Node( osg, Compass );
```

```
  void setPlate( osg::MatrixTransform* plate ) {
    _plateTransform = plate; }
  void setNeedle( osg::MatrixTransform* needle ) {
    _needleTransform = needle; }
  void setMainCamera( osg::Camera* camera ) {
    _mainCamera = camera; }

  virtual void traverse( osg::NodeVisitor& nv );

protected:
  virtual ~Compass();

  osg::ref_ptr<osg::MatrixTransform> _plateTransform;
  osg::ref_ptr<osg::MatrixTransform> _needleTransform;
  osg::observer_ptr<osg::Camera> _mainCamera;
};
```

2. Implement the copy constructor of the `Compass` class. Without a copy constructor, you will not be able to use the `META_Node` macro to define the standard node methods:

```
Compass::Compass( const Compass& copy, osg::CopyOp copyop ):
  osg::Camera(copy, copyop),
  _plateTransform(copy._plateTransform),
  _needleTransform(copy._needleTransform),
  _mainCamera(copy._mainCamera)
{
}
```

3. The `traverse()` method will be called during the event, update, and cull traversals of the entire scene graph in every frame. Override it and we will have custom behaviors for own node types.
4. For the compass, we have to compute the angle between the present viewer orientation and the north vector (the earth's geographic pole), and rotate the needle or plate node to align itself. Here we will read the current view matrix from the main camera and move the plate to fit it. This can be done during the cull traversal as there are few factors affecting the viewer's position and direction:

```
void Compass::traverse( osg::NodeVisitor& nv )
{
  if ( _mainCamera.valid() &&
    nv.getVisitorType()==osg::NodeVisitor::CULL_VISITOR )
  {
    osg::Matrix matrix = _mainCamera->getViewMatrix();
    matrix.setTrans( osg::Vec3() );
```

```
        osg::Vec3 northVec = osg::Z_AXIS * matrix;
        northVec.z() = 0.0f;
        northVec.normalize();

        osg::Vec3 axis = osg::Y_AXIS ^ northVec;
        float angle = atan2(axis.length(), osg::Y_AXIS*northVec);
        axis.normalize();

        if ( _plateTransform.valid() )
        _plateTransform->setMatrix( osg::Matrix::rotate(
          angle, axis) );
      }
      _plateTransform->accept( nv );
      _needleTransform->accept( nv );
      osg::Camera::traverse( nv );
    }
```

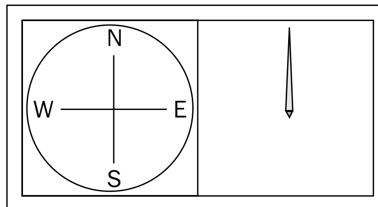
5. Later we will explain why we directly call `accept()` here, and why the `_plateTransform` and `_needleTransform` nodes are never added as the compass' children.

   The compass class can be used in any applications now. Let's try it now.

6. First there are header files for use:

```
#include <osg/ShapeDrawable>
#include <osg/MatrixTransform>
#include <osg/Texture2D>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

7. You may have many ways to design your own compass needle and plate. But in this recipe, we will choose to use textured quads. Create a needle image with transparent background, superimpose it onto the plate image, and the result will be nice enough for our case (a 2D compass). The example images are shown in the following diagram:



8. Create a function for the needle or plate node. The height parameter is for computing the Z-order of these two components:

```
osg::MatrixTransform* createCompassPart( const std::string&
  image, float radius, float height )
{
  osg::Vec3 center(-radius, -radius, height);
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable(
    createTexturedQuadGeometry(center, osg::Vec3(radius*2.0f,0.0f,
0.0f),
      osg::Vec3(0.0f,radius*2.0f,0.0f)) );

  osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
  texture->setImage( osgDB::readImageFile(image) );

  osg::ref_ptr<osg::MatrixTransform> part =
    new osg::MatrixTransform;
  part->getOrCreateStateSet()->setTextureAttributeAndModes(
    0, texture.get() );
  part->getOrCreateStateSet()->setRenderingHint(
    osg::StateSet::TRANSPARENT_BIN );
  part->addChild( geode.get() );
  return part.release();
}
```

9. Create a demo earth model:

```
osg::Geode* createEarth( const std::string& filename )
{
  osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
  texture->setImage( osgDB::readImageFile(filename) );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( new osg::ShapeDrawable(
    new osg::Sphere(osg::Vec3(), osg::WGS_84_RADIUS_POLAR)) );
  geode->getOrCreateStateSet()->setTextureAttributeAndModes(
    0, texture.get() );
  return geode.release();
}
```

10. In the main entry, create the viewer and attach the main camera to the compass, which is shown orthographic:

```
osgViewer::Viewer viewer;

osg::ref_ptr<Compass> compass = new Compass;
compass->setMainCamera( viewer.getCamera() );
compass->setViewport( 0.0, 0.0, 200.0, 200.0 );
compass->setProjectionMatrix( osg::Matrixd::ortho(
  -1.5, 1.5, -1.5, 1.5, -10.0, 10.0) );
```

11. Add the plate and the needle images to the compass node. The needle must appear on top of the plate, so it has a larger height value here:

```
compass->setPlate( createCompassPart("compass_plate.png",
  1.5f, -1.0f) );
compass->setNeedle( createCompassPart("compass_needle.png",
  1.5f, 0.0f) );
```

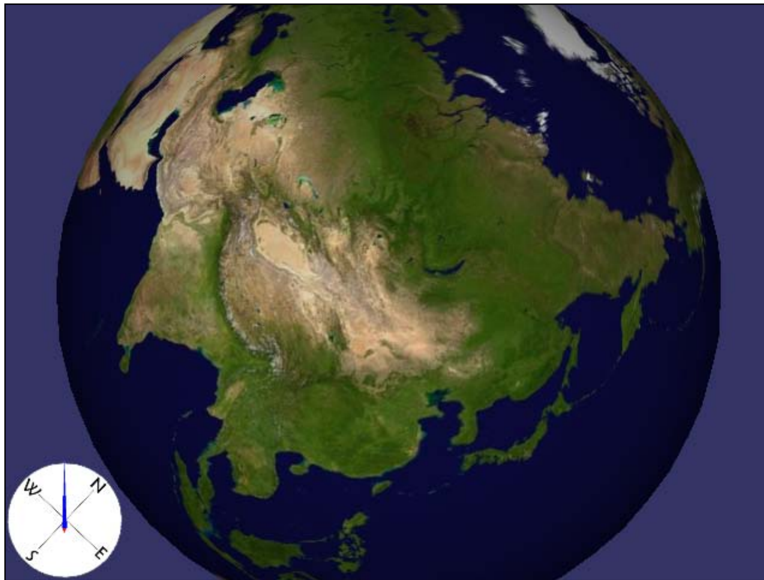
12. The 2D compass is in fact a HUD camera. The following code defines its basic behaviors:

```
compass->setRenderOrder( osg::Camera::POST_RENDER );
compass->setClearMask( GL_DEPTH_BUFFER_BIT );
compass->setAllowEventFocus( false );
compass->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
compass->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );
compass->getOrCreateStateSet()->setMode( GL_BLEND,
  osg::StateAttribute::ON );
```

13. Add the earth and the compass to the root node and start the viewer. The earth image file can be found in the OSG sample dataset.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild(
  createEarth("Images/land_shallow_topo_2048.jpg") );
root->addChild( compass.get() );

viewer.setSceneData( root.get() );
return viewer.run();
```

14. You may start navigating the scene and don't worry about determining the direction in the virtual world. Of course, if you really get lost inside a complex 3D scene one day, maybe quitting the program and restarting will be easier.



How it works...

In this recipe, the north vector is defined as the Z axis in the world coordinates. All we have to do here is figure out how the compass' magnetized needle is pulled towards the North Pole, and rotate the transformation nodes (`_plateTransform` or `_needleTransform`) correspondingly. Watch the **Google Earth** application carefully and you will see that the compass plate is rotating while you look around.

Imagine you are facing the true north in our simple 3D world, your compass needle should point to the top of the screen at that time, that is, actually the positive Y axis. So if there are any orientation changes, it can just be considered as the angle difference between the Y axis in the eye coordinates and the computed earth's north vector in eye coordinates.

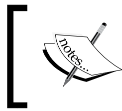
Transform the world north vector with the view matrix, regardless of the position offset. After that, calculate the rotation axis (**cross product** of the Y axis and the north vector, as they are both in the view coordinate system) and angle, and apply them to the needle or plate node at your own discretion.

```
osg::Vec3 axis = osg::Y_AXIS ^ northVec;
float angle = atan2(axis.length(), osg::Y_AXIS*northVec);
axis.normalize();
```

Another question you may have if you have already read the example source code is: Why didn't we add the needle and plate nodes to the compass, and how could they still work without being considered as children? Good question! And if you have ever read the implementation of the `osg::Group` class, you may have found out the answer yourself:

```
void Group::traverse(NodeVisitor& nv)
{
  for(NodeList::iterator itr=_children.begin();
  itr!=_children.end(); ++itr)
  {
    (*itr)->accept(nv);
  }
}
```

While calling the `traverse()` method of its super class, the `Compass` class (and other classes derived from `osg::Group`) will actually iterate each child and call the `accept()` method on them, to make the traversal continue. But here, the work is done by directly calling the `accept()` method on the transformation nodes `_plateTransform` and `_needleTransform`. It means that these two nodes will be traversed as if they were children of the compass. This sometimes brings flexibility.



But be careful, without being added to the scene graph, a node will lose some functionalities such as contributing to the bounding box computation and executing update callbacks applied on it.

Note that the `osg::Camera` class doesn't override the `traverse()` method; it simply calls the `osg::Group`'s `traverse()` method. That is why our strategy works here. And of course, everything would work well if you decide to add the needle and plate as children of the compass.

3

Editing Geometry Models

In this chapter, we will cover:

- Creating a polygon with borderlines
- Extruding a 2D shape to 3D
- Drawing a NURBS surface
- Drawing a dynamic clock on the screen
- Drawing a ribbon following a model
- Selecting and highlighting a model
- Selecting a triangle face of the model
- Selecting a point on the model
- Using vertex-displacement mapping in shaders
- Using the draw instanced extension

Introduction

This chapter is all about creating and manipulating geometries. You will see some well-designed examples showing how to meet specified user demands to create parametric polygons, animate geometry vertices, and make use of advanced techniques such as **displacement mapping** and **draw instanced**.

You have to be very familiar with the construction of `osg::Geometry` objects and manipulation of vertex arrays and primitive sets. We will skip the introduction of basic knowledge of these concepts and directly work on some practical applications in the following recipes.

To help implement some animating models quickly, we will add a new function in the common `osgCookbook` namespace. This `createAnimationPathCallback()` function will generate a circle animation path which can be applied to transformation nodes:

```
osg::AnimationPathCallback* createAnimationPathCallback(
  float radius, float time )
{
  osg::ref_ptr<osg::AnimationPath> path =
    new osg::AnimationPath;
  path->setLoopMode( osg::AnimationPath::LOOP );

  unsigned int numSamples = 32;
  float delta_yaw = 2.0f * osg::PI/((float)numSamples - 1.0f);
  float delta_time = time / (float)numSamples;
  for ( unsigned int i=0; i<numSamples; ++i )
  {
    float yaw = delta_yaw * (float)i;
    osg::Vec3 pos( sinf(yaw)*radius, cosf(yaw)*radius, 0.0f );
    osg::Quat rot( -yaw, osg::Z_AXIS );
    path->insert( delta_time * (float)i,
      osg::AnimationPath::ControlPoint(pos, rot) );
  }

  osg::ref_ptr<osg::AnimationPathCallback> apcb =
    new osg::AnimationPathCallback;
  apcb->setAnimationPath( path.get() );
  return apcb.release();
}
```

Creating a polygon with borderlines

The first recipe in this chapter originates from a common functionality in the **GIS** (**Geographic Information Systems**) field. GIS data always contains the spatial feature part and non-spatial attribute part. The features, including rivers, roads, cities, and so on, are represented using points, lines, and polygons. And mixtures of lines and polygons can be used to describe complex shapes like lakes, country lands, and their boundaries.

For example, we can draw the territory of China with a series of solid triangles and quads, and then draw the national borderlines using the same sample points, but with different primitive types.

Now it's time to start creating such a polygon-and-borderline scene.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/LineWidth>
#include <osgUtil/Tessellator>
#include <osgViewer/Viewer>
```

2. Create the vertex array. It includes a quad (4 vertices in counter-clockwise direction) with a hole (another 4 vertices in clockwise direction) in the center. So we are actually going to create a concave polygon in this recipe:

```
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array(8);
(*vertices)[0].set( 0.0f, 0.0f, 0.0f );
(*vertices)[1].set( 3.0f, 0.0f, 0.0f );
(*vertices)[2].set( 3.0f, 0.0f, 3.0f );
(*vertices)[3].set( 0.0f, 0.0f, 3.0f );
(*vertices)[4].set( 1.0f, 0.0f, 1.0f );
(*vertices)[5].set( 2.0f, 0.0f, 1.0f );
(*vertices)[6].set( 2.0f, 0.0f, 2.0f );
(*vertices)[7].set( 1.0f, 0.0f, 2.0f );
```

3. Specify a unique value for all the normals:

```
osg::ref_ptr<osg::Vec3Array> normals = new osg::Vec3Array(1);
(*normals)[0].set( 0.0f,-1.0f, 0.0f );
```

4. Build the geometry object. Here we just add two primitive sets to describe the quad and the hole separately. Without any polygon tessellation process, they will be treated as two overlapped quads and thus lead to an ugly rendering result:

```
osg::ref_ptr<osg::Geometry> polygon = new osg::Geometry;
polygon->setVertexArray( vertices.get() );
polygon->setNormalArray( normals.get() );
polygon->setNormalBinding( osg::Geometry::BIND_OVERALL );
polygon->addPrimitiveSet( new osg::DrawArrays(
  GL_QUADS, 0, 4) );
polygon->addPrimitiveSet( new osg::DrawArrays(
  GL_QUADS, 4, 4) );
```

5. Start the tessellation work, that is, subdivide the polygon with a hole into convex polygons:

```
osgUtil::Tessellator tessellator;
tessellator.setTessellationType( osgUtil::Tessellator::TESS_TYPE_
GEOMETRY );
tessellator.setWindingType(
  osgUtil::Tessellator::TESS_WINDING_ODD );
tessellator.retessellatePolygons( *polygon );
```

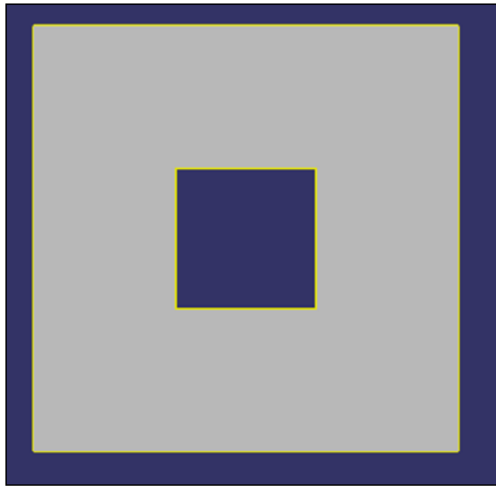
6. Now it's time to create the borderlines. As we know the polygon has a hole inside, so there should be two sets of connected lines to represent the outer boundary and the hole. We will make use of the same vertex array directly, and give the border object a different global color and line width parameter:

```
osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
(*colors)[0].set( 1.0f, 1.0f, 0.0f, 1.0f );

osg::ref_ptr<osg::Geometry> border = new osg::Geometry;
border->setVertexArray( vertices.get() );
border->setColorArray( colors.get() );
border->setColorBinding( osg::Geometry::BIND_OVERALL );
border->addPrimitiveSet( new osg::DrawArrays(
  GL_LINE_LOOP, 0, 4) );
border->addPrimitiveSet( new osg::DrawArrays(
  GL_LINE_LOOP, 4, 4) );
border->getOrCreateStateSet()->setAttribute(
  new osg::LineWidth(5.0f) );
```

7. Add the two geometries to the scene graph and start the viewer:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( polygon.get() );
geode->addDrawable( border.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

8. Now the polygon is shown perfectly with a clear boundary representation, as shown in the following screenshot. You may try some other complex polygons (especially some concave ones) and see if they could be handled in a similar way.



How it works...

You may find that the vertex array is shared by two different geometries. This provides us the benefit of saving memories on both CPU and GPU. OSG makes it flexible enough for multiple geometries to share a single **vertex buffer object** (**VBO**) and use different subsets of it with the help of `osg::PrimitiveSet`'s sub-classes. To note, in this recipe we haven't turned on the VBO property of `osg::Geometry` class yet, but use the traditional display lists for rendering. Dynamic geometry examples with VBO supports will be introduced later in this chapter.

Also pay attention to the `setTessellationType()` method of the tessellator. The value `TESS_TYPE_GEOMETRY` means to tessellate everything added as primitive sets, including triangles, quads, and polygons. If we are only going to handle `GL_POLYGON` faces, consider using `TESS_TYPE_POLYGONS` instead, in which case existing quads and triangles will be left alone.

There's more...

The `osgUtil::Tessellator` class uses the OpenGL tessellation algorithm internally. See the book "*OpenGL Programming Guide*", *Mason Woo*, *Jackie Neider*, *Tom Davis*, *Dave Shreiner*, and *OpenGL Architecture Review Board*, *Addison-Wesley*, or its online version for details:

`http://glprogramming.com/red/chapter11.html`

Extruding a 2D shape to 3D

Extrusion is one of the most common functionalities to create 3D objects quickly in 3D-modeling software such as 3DSMAX and Maya. Extrusion is often used to create models from 2D shapes and curves by dragging 2D shapes along a line with specified direction and length. For instance, a circle will thus be turned into a cylinder if it is extruded along the line perpendicular to the circle plane.

OSG doesn't directly provide such an extrusion utility. So that is what we are going to do in this section.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osgUtil/SmoothingVisitor>
#include <osgUtil/Tessellator>
#include <osgViewer/Viewer>
```

2. The extrusion function requires at least three arguments—an array which contains all the vertices compositing 2D shape; the extrusion direction; and the extrusion length:

```
osg::Geometry* createExtrusion( osg::Vec3Array* vertices,
  const osg::Vec3& direction, float length )
{
  ...
}
```

3. First, we compute all the points of the result 3D model, including the original points of the 2D shape, and new points after being extruded in a certain direction with a certain length value. You may have noticed that new vertices are computed using a reverse interator here. This actually helps build the normal vector of the bottom cap of the extruded geometry as shown in the following code block:

```
osg::ref_ptr<osg::Vec3Array> newVertices = new osg::Vec3Array;
newVertices->insert( newVertices->begin(), vertices->begin(),
  vertices->end() );

unsigned int numVertices = vertices->size();
osg::Vec3 offset = direction * length;
for ( osg::Vec3Array::reverse_iterator ritr=
  vertices->rbegin(); ritr!=vertices->rend(); ++ritr )
{
  newVertices->push_back( (*ritr) + offset );
}
```

4. Add two primitive sets to represent the top and bottom caps. It's uncomfortable for OpenGL to render `GL_POLYGON` primitives, so we have to tessellate them at once. Note that `TESS_TYPE_POLYGONS` is used here instead of `TESS_TYPE_GEOMETRY`, which was used in the *Creating a polygon with borderlines* recipe:

```
osg::ref_ptr<osg::Geometry> extrusion = new osg::Geometry;
extrusion->setVertexArray( newVertices.get() );
extrusion->addPrimitiveSet( new osg::DrawArrays(GL_POLYGON,
  0, numVertices) );
extrusion->addPrimitiveSet( new osg::DrawArrays(GL_POLYGON,
  numVertices, numVertices) );

osgUtil::Tessellator tessellator;
tessellator.setTessellationType(
  osgUtil::Tessellator::TESS_TYPE_POLYGONS );
tessellator.setWindingType(
  osgUtil::Tessellator::TESS_WINDING_ODD );
tessellator.retessellatePolygons( *extrusion );
```

5. The tessellation may sometimes add or remove primitive sets of the geometry. So if we are going to add some other primitive, for instance, the side faces, we would better proceed after handling the cap polygons. Here we simply construct a connected set of quads sharing edges (with the same first and last edges to form a loop) to assemble the surface:

```
osg::ref_ptr<osg::DrawElementsUInt> sideIndices =
  new osg::DrawElementsUInt( GL_QUAD_STRIP );
for ( unsigned int i=0; i<numVertices; ++i )
{
  sideIndices->push_back( i );
  sideIndices->push_back( (numVertices-1-i) + numVertices );
}
sideIndices->push_back( 0 );
sideIndices->push_back( numVertices*2 - 1 );
extrusion->addPrimitiveSet( sideIndices.get() );
```

6. Compute the normals and return the resulting geometry at last:

```
osgUtil::SmoothingVisitor::smooth( *extrusion );
return extrusion.release();
```

7. In the main entry, we give the users some rights to define their own extrusion direction and length values:

```
osg::ArgumentParser arguments( &argc, argv );

osg::Vec3 direction(0.0f, 0.0f, -1.0f);
arguments.read( "--direction", direction.x(), direction.y(),
  direction.z() );

float length = 5.0f;
arguments.read( "--length", length );
```

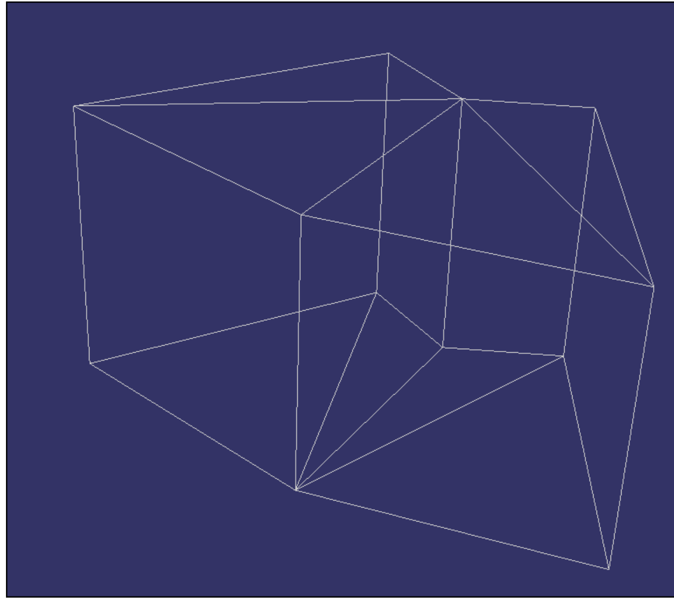
8. Create a list of 2D points for generating the 3D model. In fact, a 3D path here could also work:

```
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array(6);
(*vertices)[0].set( 0.0f, 4.0f, 0.0f );
(*vertices)[1].set(-2.0f, 5.0f, 0.0f );
(*vertices)[2].set(-5.0f, 0.0f, 0.0f );
(*vertices)[3].set( 0.0f,-1.0f, 0.0f );
(*vertices)[4].set( 5.0f, 0.0f, 0.0f );
(*vertices)[5].set( 2.0f, 5.0f, 0.0f );
```

9. Add the extrusion to the scene graph and start the viewer:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createExtrusion(vertices.get(), direction,
  length) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

10. It's done! And now you can say that extrusion is so easy, isn't it? But actually you can find that a large number of objects are made from extrudable 2D shapes, such as a pipe, a pillar, and even a rectangle building. Now, why not try to build one or more of them by yourself?



There's more...

Another good and useful modeling method is called revolution (or lathe in 3DSMAX, the name of an industry machine). It rotates a 2D curve (either open or closed) along a specified axis to create 3D objects such as goblets and pillars. The key parameters are the reference axis and the degrees of revolution. So, how about trying to implement this by yourself with this recipe as a reference?

Drawing a NURBS surface

NURBS (**Non-Uniform Rational B-Splines**) is a powerful way for creating complex curves and surfaces. It requires only a few control points and knot vectors rather than hundreds of sample points on the surface. That means we can mathematically describe a curve or surface using B-Splines or NURBS, instead of approximating it with small pieces of line segments or triangles. It sounds good to our developers who want to characterize their models in a more precise way.

OpenGL provides evaluators and NURBS interface for rendering these parametric models, but OSG doesn't for some efficiency and practicability considerations. Rendering NURBS curves and surfaces always costs more on the graphics hardware than using approximate polygons (LODs can be even better). But it can't stop us from implementing one ourself. In this recipe, we will derive from the `osg::Drawable` class and execute OpenGL commands to draw NURBS surfaces during the rendering traversal of the scene graph.

Some basic knowledge about NURBS can be found at
`http://en.wikipedia.org/wiki/Non-uniform_rational_B-spline`.

Getting ready

You will have to modify the `CMakeLists.txt` we created in the last recipe of *Chapter 1* to find OpenGL and GLU packages before compiling and running this example:

```
FIND_PACKAGE(OpenGL)

INCLUDE_DIRECTORIES(${OPENGL_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(${EXAMPLE_NAME}
  ${OPENGL_gl_LIBRARY} ${OPENGL_glu_LIBRARY})
```

How to do it...

Let us start.

1. We are going to design an as-complete-as-possible NURBS surface class in this example. OpenGL provides some good enough NURBS implementations for us to use here. And the best way to encapsulate them is to derive from the `osg::Drawable` class:

```
class NurbsSurface : public osg::Drawable
{
public:
  NurbsSurface()
  : _sCount(0), _tCount(0), _sOrder(0), _tOrder(0),
    _nurbsObj(0) {}
  NurbsSurface( const NurbsSurface& copy,
    osg::CopyOp copyop=osg::CopyOp::SHALLOW_COPY );
  META_Object( osg, NurbsSurface );

...
};
```

2. Three kinds of arrays can be applied to the `NurbsSurface` class: the control point (vertex) array, the normal array, and the texture coordinate array for texture mapping. For a NURBS surface, we have to also set the non-decreasing knot values along with the knot numbers and orders in the parametric U and V directions. All these requirements will be added as member methods here:

```
void setVertexArray( osg::Vec3Array* va ) { _vertices = va; }
void setNormalArray( osg::Vec3Array* na ) { _normals = na; }
void setTexCoordArray( osg::Vec2Array* ta ) {
  _texcoords = ta; }
```

```
void setKnots( osg::FloatArray* sknots,
  osg::FloatArray* tknots )
{ _sKnots = sknots; _tKnots = tknots; }
void setCounts( int s, int t ) { _sCount = s; _tCount = t; }
void setOrders( int s, int t ) { _sOrder = s; _tOrder = t; }
```

3. The two most important virtual methods to re-implement are `computeBound()` and `drawImplementation()`. Without either of them, we would get an improper rendering result finally:

```
virtual osg::BoundingBox computeBound() const;
virtual void drawImplementation( osg::RenderInfo&
  renderInfo ) const;
```

4. Define protected members:

```
virtual ~NurbsSurface() {}

osg::ref_ptr<osg::Vec3Array> _vertices;
osg::ref_ptr<osg::Vec3Array> _normals;
osg::ref_ptr<osg::Vec2Array> _texcoords;
osg::ref_ptr<osg::FloatArray> _sKnots;
osg::ref_ptr<osg::FloatArray> _tKnots;
int _sCount, _tCount;
int _sOrder, _tOrder;
mutable void* _nurbsObj;
```

5. OpenGL and GLU headers are required before we really implement each class method. And here is the copy constructor which can help do a shallow or deep copy of the object:

```
#include <osg/GL>
#include <GL/glu.h>

NurbsSurface::NurbsSurface( const NurbsSurface& copy,
  osg::CopyOp copyop )
:  osg::Drawable(copy, copyop), _vertices(copy._vertices),
    _normals(copy._normals), _texcoords(copy._texcoords),
    _sKnots(copy._sKnots), _tKnots(copy._tKnots),
    _sOrder(copy._sOrder), _tOrder(copy._tOrder),
    _nurbsObj(copy._nurbsObj)
{}
```

6. The `computeBound()` should be re-implemented for the scene culling process, during which the NURBS surface may be ignored if it is totally out of the view frustum. It is enough to only compute the control points here:

```
osg::BoundingBox NurbsSurface::computeBound() const
{
  osg::BoundingBox bb;
  if ( _vertices.valid() )
  {
    for ( unsigned int i=0; i<_vertices->size(); ++i )
    bb.expandBy( (*_vertices)[i] );
  }
  return bb;
}
```

7. The `drawImplementation()` will be called only once to build a display list for further use unless this default mechanism is disabled. It is OK to keep it because we don't need the NURBS surface to change dynamically here:

```
void NurbsSurface::drawImplementation( osg::RenderInfo&
  renderInfo ) const
{
  ...
}
```

8. In this implementation function, we create the OpenGL NURBS object if it is not already allocated:

```
GLUnurbsObj* theNurbs = (GLUnurbsObj*)_nurbsObj;
if ( !theNurbs )
{
  theNurbs = gluNewNurbsRenderer();
  gluNurbsProperty( theNurbs, GLU_SAMPLING_TOLERANCE, 10 );
  gluNurbsProperty( theNurbs, GLU_DISPLAY_MODE, GLU_FILL );
  _nurbsObj = theNurbs;
}
```

9. Execute proper OpenGL calls to finish the whole NURBS drawing process. There is nothing special in the following code segment. The only point is to pay attention to the changes of OpenGL states, which may affect other drawables in complex applications:

```
if ( _vertices.valid() && _sKnots.valid() && _tKnots.valid() )
{
  glEnable( GL_MAP2_NORMAL );
  glEnable( GL_MAP2_TEXTURE_COORD_2 );

  gluBeginCurve( theNurbs );
```

```
    if ( _texcoords.valid() )
    {
      gluNurbsSurface( theNurbs, _sKnots->size(),
        &((*_sKnots)[0]), _tKnots->size(), &((*_tKnots)[0]),
        _sCount*2, 2, &((*_texcoords)[0][0]), _sOrder, _tOrder,
        GL_MAP2_TEXTURE_COORD_2 );
    }
    if ( _normals.valid() )
    {
      gluNurbsSurface( theNurbs, _sKnots->size(),
        &((*_sKnots)[0]), _tKnots->size(), &((*_tKnots)[0]),
        _sCount*3, 3, &((*_normals)[0][0]), _sOrder, _tOrder,
        GL_MAP2_NORMAL );
    }
    gluNurbsSurface( theNurbs, _sKnots->size(),
      &((*_sKnots)[0]), _tKnots->size(), &((*_tKnots)[0]),
      _sCount*3, 3, &((*_vertices)[0][0]), _sOrder, _tOrder,
      GL_MAP2_VERTEX_3 );
    gluEndCurve( theNurbs );

    glDisable( GL_MAP2_NORMAL );
    glDisable( GL_MAP2_TEXTURE_COORD_2 );
  }
```

10. Now we will make use of the new NURBS class to show a small surface. First we include some other necessary headers:

```
#include <osg/Geode>
#include <osg/Texture2D>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

11. In the main entry, we will create an ordinary NURBS surface by specifying its control points, texture coordinates, and knots:

```
osg::ref_ptr<osg::Vec3Array> ctrlPoints = new osg::Vec3Array;
#define ADD_POINT(x, y, z) ctrlPoints->push_back(
  osg::Vec3(x, y, z) );
ADD_POINT(-3.0f, 0.5f, 0.0f); ADD_POINT(-1.0f, 1.5f, 0.0f);
  ADD_POINT(-2.0f, 2.0f, 0.0f);
ADD_POINT(-3.0f, 0.5f,-1.0f); ADD_POINT(-1.0f, 1.5f,-1.0f);
  ADD_POINT(-2.0f, 2.0f,-1.0f);
ADD_POINT(-3.0f, 0.5f,-2.0f); ADD_POINT(-1.0f, 1.5f,-2.0f);
  ADD_POINT(-2.0f, 2.0f,-2.0f);

osg::ref_ptr<osg::Vec2Array> texcoords = new osg::Vec2Array;
```

```
#define ADD_TEXCOORD(x, y) texcoords->push_back(
  osg::Vec2(x, y) );
ADD_TEXCOORD(0.0f, 0.0f); ADD_TEXCOORD(0.5f, 0.0f);
  ADD_TEXCOORD(1.0f, 0.0f);
ADD_TEXCOORD(0.0f, 0.5f); ADD_TEXCOORD(0.5f, 0.5f);
  ADD_TEXCOORD(1.0f, 0.5f);
ADD_TEXCOORD(0.0f, 1.0f); ADD_TEXCOORD(0.5f, 1.0f);
  ADD_TEXCOORD(1.0f, 1.0f);

osg::ref_ptr<osg::FloatArray> knots = new osg::FloatArray;
knots->push_back(0.0f); knots->push_back(0.0f);
  knots->push_back(0.0f);
knots->push_back(1.0f); knots->push_back(1.0f);
  knots->push_back(1.0f);
```

12. Allocate a `NurbsSurface` drawable and apply all the variables we set before to the newly created object:

```
osg::ref_ptr<NurbsSurface> nurbs = new NurbsSurface;
nurbs->setVertexArray( ctrlPoints.get() );
nurbs->setTexCoordArray( texcoords.get() );
nurbs->setKnots( knots.get(), knots.get() );
nurbs->setCounts( 3, 3 );
nurbs->setOrders( 3, 3 );
```

13. We have nearly finished it! Now apply a texture to the node or drawable's state set, and turn off the light computing to get a better look of the result. Lastly, start the viewer:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( nurbs.get() );
geode->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, new osg::Texture2D(osgDB::readImageFile(
    "Images/osg256.png")) );
geode->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

14. A simple NURBS surface finally comes out. To make it more complex and refined, you may add more control points along the U and V directions and update the knot information simultaneously. Then leave the computation and rendering work to OpenGL, and enjoy your achievements in the rendering window.

How it works...

Now you may gain the experience of how to create a complete, customized drawable. The most important methods to re-implement are `drawImplementation()` and `computeBound()`; and you may have to consider if the default feature of building display lists should be disabled or not. If you need to call `drawImplementation()` method in every frame to execute certain user commands, then call `setUseDisplayList(false)` before the simulation loop; otherwise, you may keep it to improve your application's performance.

Other useful methods to re-implement are `supports()` and `accept()` methods and their overloaded forms. These methods are mainly used by OSG functors such as `osg::TriangleFunctor<>` to collect vertex and primitive information. Without such implementations, functors can't retrieve anything from user drawables, and intersectors will return an empty result on them because of missing data for computing. Fortunately, it doesn't matter in this example.

There's more...

There are some other third-party libraries that can parse and render B-Splines and NURBS objects. A good example is the openNURNS library:

`http://www.opennurbs.org/`

OSG itself also provides a simple example to show how to integrate **Bézier** surfaces. See the example `osgteapot` in the official OSG source code for details.

Drawing a dynamic clock on the screen

Now we will try to face a practical user requirement: design a very simple analogy clock and make it work. This most common kind of clock indicates time using numbered dial and moving hands. It usually includes one hour hand, one minute hand (a little longer and faster) and one second hand (fastest and longest). Their periods are 12 hours, 60 minutes, and 60 seconds respectively.

How to do it...

Let us start.

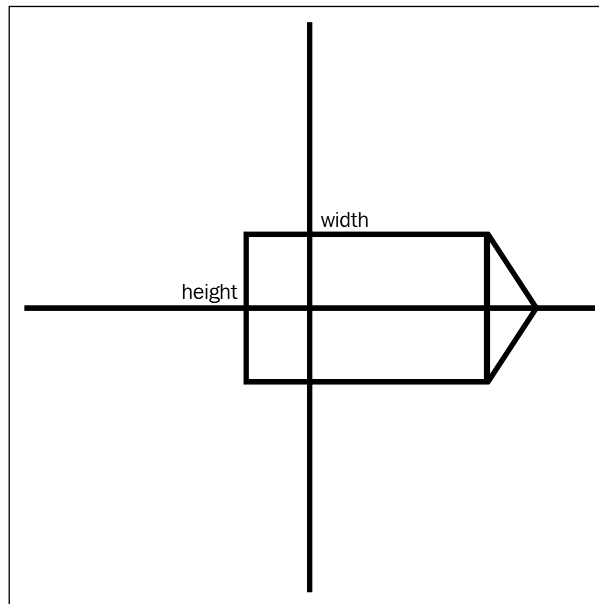
1. Include necessary headers:

   ```
   #include <osg/AnimationPath>
   #include <osg/Geometry>
   #include <osg/Geode>
   #include <osg/MatrixTransform>
   #include <osgViewer/Viewer>
   ```

2. Design a function which will create a needle (a moving hand of the clock). We have to set up the initial angle and period (the time taken to make a revolution) of each needle as well:

   ```
   osg::Node* createNeedle( float w, float h, float depth,
     const osg::Vec4& color, float angle, double period )
   {
     ...
   }
   ```

3. The shape of the needle is designed as shown in the following diagram. It's simple but enough to represent a real analog clock.



4. Now construct the geometry of the needle and add it to an `osg::Geode` node:

```
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array(5);
(*vertices)[0].set(-h*0.5f, 0.0f,-w*0.1f );
(*vertices)[1].set( h*0.5f, 0.0f,-w*0.1f );
(*vertices)[2].set(-h*0.5f, 0.0f, w*0.8f );
(*vertices)[3].set( h*0.5f, 0.0f, w*0.8f );
(*vertices)[4].set( 0.0f, 0.0f, w*0.9f );

osg::ref_ptr<osg::Vec3Array> normals = new osg::Vec3Array(1);
(*normals)[0].set( 0.0f,-1.0f, 0.0f );

osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
(*colors)[0] = color;

osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
geom->setVertexArray( vertices.get() );
geom->setNormalArray( normals.get() );
geom->setNormalBinding( osg::Geometry::BIND_OVERALL );
geom->setColorArray( colors.get() );
geom->setColorBinding( osg::Geometry::BIND_OVERALL );
geom->addPrimitiveSet( new osg::DrawArrays(
  GL_TRIANGLE_STRIP, 0, 5) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( geom.get() );
```

5. The next task is to periodically rotate the needle along the clock face. An animation path callback is used here to simulate the circular motion, which includes three keyframes (three vital points on a circle):

```
osg::ref_ptr<osg::MatrixTransform> trans =
  new osg::MatrixTransform;
trans->addChild( geode.get() );

osg::ref_ptr<osg::AnimationPath> clockPath =
  new osg::AnimationPath;
clockPath->setLoopMode( osg::AnimationPath::LOOP );
clockPath->insert( 0.0, osg::AnimationPath::ControlPoint(
  osg::Vec3(0.0f, depth, 0.0f), osg::Quat(angle, osg::Y_AXIS)) );
clockPath->insert( period*0.5, osg::AnimationPath::ControlPoint(
  osg::Vec3(0.0f, depth, 0.0f), osg::Quat(angle+osg::PI,
    osg::Y_AXIS)) );
clockPath->insert( period, osg::AnimationPath::ControlPoint(
  osg::Vec3(0.0f, depth, 0.0f), osg::Quat(angle+osg::PI*2.0f,
     osg::Y_AXIS)) );

osg::ref_ptr<osg::AnimationPathCallback> apcb =
  new osg::AnimationPathCallback;
apcb->setAnimationPath( clockPath.get() );
trans->addUpdateCallback( apcb.get() );
return trans.release();
```

6. Design the clock face. As a straightforward practice, in this recipe we are not going to draw a true clock face. Instead, we will have a clean disk without text or textures on it. The whole creation is, therefore, easy to understand:

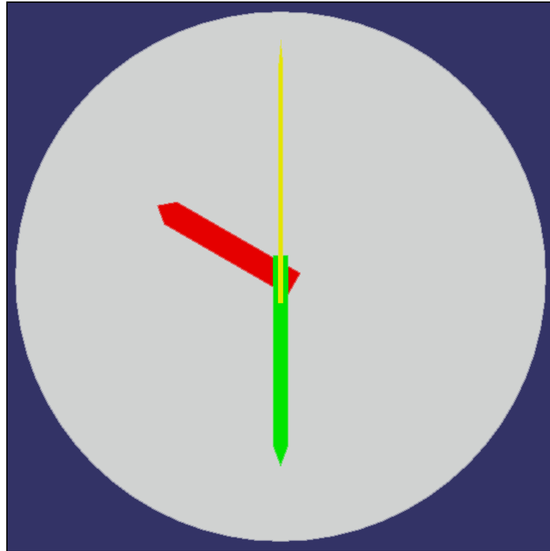
```
osg::Node* createFace( float radius )
{
  osg::ref_ptr<osg::Vec3Array> vertices =
    new osg::Vec3Array(67);
  (*vertices)[0].set( 0.0f, 0.0f, 0.0f );
  for ( unsigned int i=1; i<=65; ++i )
  {
    float angle = (float)(i-1) * osg::PI / 32.0f;
    (*vertices)[i].set( radius * cosf(angle), 0.0f,
      radius * sinf(angle) );
  }

  osg::ref_ptr<osg::Vec3Array> normals = new osg::Vec3Array(1);
  (*normals)[0].set( 0.0f,-1.0f, 0.0f );

  osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
```

```
  (*colors)[0].set( 1.0f, 1.0f, 1.0f, 1.0f );
  // Avoid color state inheriting

  osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
  geom->setVertexArray( vertices.get() );
  geom->setNormalArray( normals.get() );
  geom->setNormalBinding( osg::Geometry::BIND_OVERALL );
  geom->setColorArray( colors.get() );
  geom->setColorBinding( osg::Geometry::BIND_OVERALL );
  geom->addPrimitiveSet( new osg::DrawArrays(
    GL_TRIANGLE_FAN, 0, 67) );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( geom.get() );
  return geode.release();
}
```

7. In the main entry, we first define a fake time—`10:30`. And then we create the hour hand (the shortest), the minute hand, and the second hand (the longest) in turn. They have a small distance between them and the clock face, so there will be no overlapped faces that may cause the Z-fighting problems:

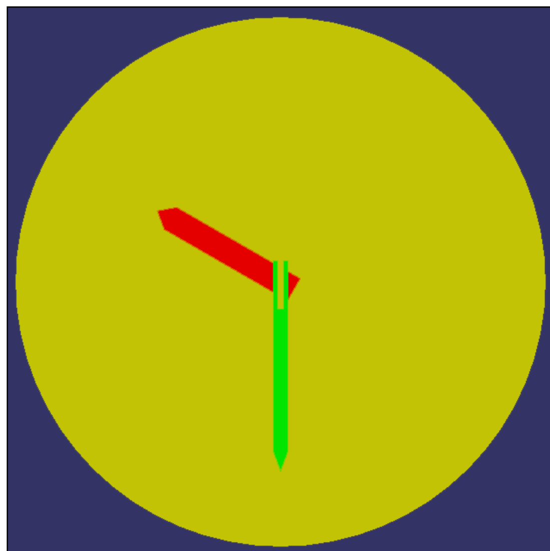
```
float hour_time = 10.0f, min_time = 30.0f, sec_time = 0.0f;
// Hour needle devides the circle into 12 parts
osg::Node* hour = createNeedle(6.0f, 1.0f,-0.02f,
  osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f), osg::PI * hour_time /
    6.0f, 3600*60.0);
// Minute/second needle devides the circle into 60 partsosg::Node*
minute = createNeedle(8.0f, 0.6f,-0.04f,
  osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f), osg::PI * min_time /
    30.0f, 3600.0);
osg::Node* second = createNeedle(10.0f, 0.2f,-0.06f,
  osg::Vec4(1.0f, 1.0f, 0.0f, 1.0f), osg::PI * sec_time /
    30.0f, 60.0);

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( hour );
root->addChild( minute );
root->addChild( second );
root->addChild( createFace(10.0f) );
Start the viewer to see our clock running:
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

8. The result is shown in the following screenshot:



How it works...

Maybe you have already ignored a fact in the `createFace()` function: We might add a 'useless' color array that has only one color to the face geometry. But what will happen if we remove the `setColorArray()` line in this function? Have a try and you may see the following screenshot:



So why does the face inherit a needle's color? Is it an OSG program bug? In fact, it's hard to say because of the famous OpenGL state machine. Thus geometry without any colors set will directly inherit the value sent to the OpenGL pipeline by the previous one, which leads to an unexpected result. The best solution is to apply a color array to every geometry object, no matter whether it needs it or not.

Drawing a ribbon following a model

Trailing ribbon can be thought as a colored ribbon following a flight or helicopter. It can be used to represent banners or streamers dragged by the aircraft, or demonstrate the flight paths in military simulations. A ribbon is never a simple quad. Its points are located on the flight line and they will go after the animated plane all the time. All these demands require a dynamic geometry in which all vertices are moving.

With the information provided in the last paragraph, now we can start working on the interesting topic.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/AnimationPath>
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. Define a few global variables. Of course, you may create a specialized ribbon class and define them as member variables, but in this recipe we will just simplify the work:

```
const unsigned int g_numPoints = 400;
const float g_halfWidth = 4.0f;
The first step is to initialize the ribbon geometry:
osg::Geometry* createRibbon( const osg::Vec3& colorRGB )
{
  ...
}
```

3. Configure the vertex, normal, and color arrays. At the beginning, all vertices are placed together at the origin point, but later they will be treated as the two sides of the ribbon. The colors are computed with a sine function to implement a fade-in and fade-out effect while the ribbon is moving:

```
osg::ref_ptr<osg::Vec3Array> vertices =
  new osg::Vec3Array(g_numPoints);
osg::ref_ptr<osg::Vec3Array> normals =
  new osg::Vec3Array(g_numPoints);
osg::ref_ptr<osg::Vec4Array> colors =
  new osg::Vec4Array(g_numPoints);

osg::Vec3 origin = osg::Vec3(0.0f, 0.0f, 0.0f);
osg::Vec3 normal = osg::Vec3(0.0f, 0.0f, 1.0f);
for ( unsigned int i=0; i<g_numPoints-1; i+=2 )
{
  (*vertices)[i] = origin; (*vertices)[i+1] = origin;
  (*normals)[i] = normal; (*normals)[i+1] = normal;

  float alpha = sinf(osg::PI * (float)i / (float)g_numPoints);
  (*colors)[i] = osg::Vec4(colorRGB, alpha);
  (*colors)[i+1] = osg::Vec4(colorRGB, alpha);
}
```

4. Create the dynamic geometry object. "Dynamic" here means the ribbon geometry will change its points and primitives all the time during the simulation. In this case, we choose to use vertex buffer objects instead of display lists:

```
osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
geom->setDataVariance( osg::Object::DYNAMIC );
geom->setUseDisplayList( false );
geom->setUseVertexBufferObjects( true );
Set up other options and return at the end of the createRibbon()
function:
geom->setVertexArray( vertices.get() );
geom->setNormalArray( normals.get() );
geom->setNormalBinding( osg::Geometry::BIND_PER_VERTEX );
geom->setColorArray( colors.get() );
geom->setColorBinding( osg::Geometry::BIND_PER_VERTEX );
geom->addPrimitiveSet( new osg::DrawArrays(
  GL_QUAD_STRIP, 0, g_numPoints) );
return geom.release();
```

5. The second step is to animate the ribbon while it is following a moving model, and implement the trailing effect. The `TrailerCallback` must be added as an update callback to an `osg::MatrixTransform` node to read and use its transformation matrix at runtime:

```
class TrailerCallback : public osg::NodeCallback
{
public:
  TrailerCallback( osg::Geometry* ribbon ) :
    _ribbon(ribbon) {}

  virtual void operator()( osg::Node* node,
    osg::NodeVisitor* nv );

protected:
  osg::observer_ptr<osg::Geometry> _ribbon;
};
In the operator() method, obtain necessary values and be ready to
edit the vertices and normals:
osg::MatrixTransform* trans =
  static_cast<osg::MatrixTransform*>(node);
if ( trans && _ribbon.valid() )
{
  osg::Matrix matrix = trans->getMatrix();
  osg::Vec3Array* vertices = static_cast<osg::Vec3Array*>(
    _ribbon->getVertexArray() );
  osg::Vec3Array* normals = static_cast<osg::Vec3Array*>(
    _ribbon->getNormalArray() );

  ...
}
traverse( node, nv );
```

6. Compute the new positions and normals of the ribbon points. Dirty the arrays to remind buffer objects to update the graphics memory. And don't forget to recompute the ribbon's bounding box with `dirtyBound()`, for the purpose of the scene-culling process:

```
for ( unsigned int i=0; i<g_numPoints-3; i+=2 )
{
  (*vertices)[i] = (*vertices)[i+2];
  (*vertices)[i+1] = (*vertices)[i+3];
  (*normals)[i] = (*normals)[i+2];
  (*normals)[i+1] = (*normals)[i+3];
}
(*vertices)[g_numPoints-2] = osg::Vec3(0.0f,-g_halfWidth,
```

```
  0.0f) * matrix;
(*vertices)[g_numPoints-1] = osg::Vec3(0.0f, g_halfWidth,
  0.0f) * matrix;
vertices->dirty();

osg::Vec3 normal = osg::Vec3(0.0f, 0.0f, 1.0f) * matrix;
normal.normalize();
(*normals)[g_numPoints-2] = normal;
(*normals)[g_numPoints-1] = normal;
normals->dirty();

_ribbon->dirtyBound();
```

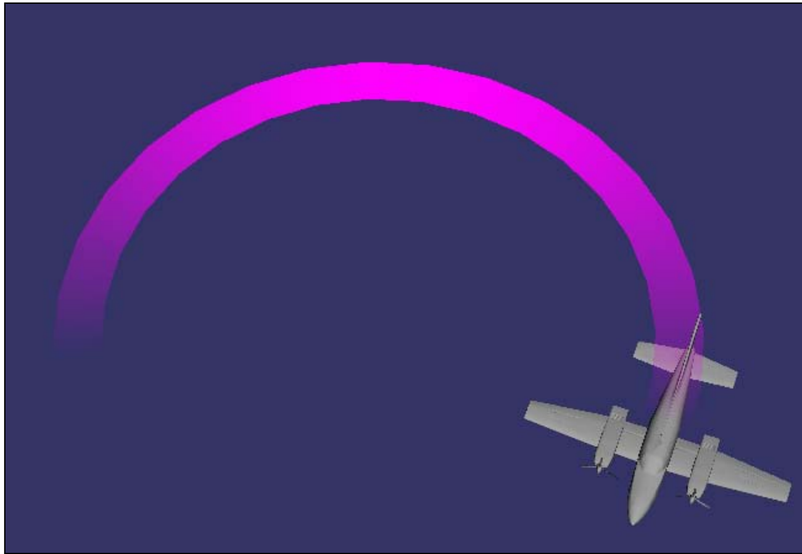
7. Now in the main entry, create a ribbon node and make it transparent if necessary:

```
osg::Geometry* geometry = createRibbon( osg::Vec3(1.0f, 0.0f,
  1.0f) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( geometry );
geode->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );
geode->getOrCreateStateSet()->setMode( GL_BLEND,
  osg::StateAttribute::ON );
geode->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
```

8. Load a Cessna model into the scene graph and make it fly all the time. Add the ribbon to both the trailer callback and the scene graph:

```
osg::ref_ptr<osg::MatrixTransform> cessna =
  new osg::MatrixTransform;
cessna->addChild( osgDB::readNodeFile("cessna.osg.0,0,90.rot") );
cessna->addUpdateCallback(
  osgCookBook::createAnimationPathCallback(50.0f, 6.0f) );
cessna->addUpdateCallback( new TrailerCallback(geometry) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );
root->addChild( cessna.get() );
Start the viewer:
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

9. The result looks good, although we don't make use of shaders and other advanced techniques. You may find similar implementations in some flight simulation software and games. Believe it or not, they might not be as difficult as you thought before.



How it works...

The principle of making a list of trailing vertices is simple: For each two points (at the left and right side of the ribbon), read and accept the values of the next two points in the array, and so forth. The last two points, which can be considered as the front end of the ribbon, should be connected to the transformation node. Their positions and normals will be updated according to the current matrix. And in the next few frames, these values will be delivered to following points, which finally implement a complete trailing effect.

VBO (**Vertex Buffer Objects**) is used for representing dynamic geometries. Display lists are unsuitable here because they don't submit vertex changes to the OpenGL pipeline, unless users destroy the previous display list object and create a new one. This can be done by calling `dirtyDisplayList()` method, but much less efficient than using buffer objects. VBO provides a fast way to communicate between user applications and the GPU while handing vertex attributes and indices. To notify the changes of vertex data, you may just call `dirty()` method of an array object. OSG will update it for you automatically in the back-end rendering.

There's more...

More information about VBO and its features can be found at `http://www.opengl.org/wiki/Vertex_Buffer_Object`.

Selecting and highlighting a model

If you have ever read the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, you may find this topic familiar. Yes, we have done such work as picking a drawable or node in the scene graph and highlighting it. But in this example, we will highlight the selected drawable directly, assuming it is an `osg::Geometry` object (which already implements related intersection algorithms) and has a color array.

Maybe you think it would be uninteresting because you had already done the same exercise before. But don't hesitate to continue reading this and the following two recipes. They actually describe the same requirement existing in many 3D browsing and modeling software—to select a 3D entity or only parts of it (faces, edges, or points).

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osgUtil/SmoothingVisitor>
#include <osgViewer/Viewer>
```

2. Define global color variables (`normalColor` as the base color and `selectedColor` as selected) for selected and unselected objects:

```
const osg::Vec4 normalColor(1.0f, 1.0f, 1.0f, 1.0f);
const osg::Vec4 selectedColor(1.0f, 0.0f, 0.0f, 0.5f);
```

3. Declare a handler class for selecting objects that have intersections with the cursor:

```
class SelectModelHandler : public osgCookBook::PickHandler
{
public:
  SelectModelHandler() : _lastDrawable(0) {}

  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::Intersection& result );
  void setDrawableColor( osg::Geometry* geom,
    const osg::Vec4& color );

protected:
  osg::observer_ptr<osg::Geometry> _lastDrawable;
};
```

4. Out picking strategy in the `doUserOperations()` method is easy to understand—cancel the last selected one, and select the new one. The selected drawable will be painted with a different color (red and translucence) than the normal ones:

```
if ( _lastDrawable.valid() )
{
  setDrawableColor( _lastDrawable.get(), normalColor );
  _lastDrawable = NULL;
}

osg::Geometry* geom = dynamic_cast<osg::Geometry*>(
  result.drawable.get() );
if ( geom )
{
  setDrawableColor( geom, selectedColor );
  _lastDrawable = geom;
}
```

5. In the `setDrawableColor()` method, we assume that all the models in this recipe have a color array with only one element. In other complicated situations, you may have a model bind with per-vertex colors, or without any color settings. Consider rewriting this method to fit such requirements:

```
osg::Vec4Array* colors = dynamic_cast<osg::Vec4Array*>(
  geom->getColorArray() );
if ( colors && colors->size()>0 )
{
  colors->front() = color;
  colors->dirty();
}
```

6. The `createSimpleGeometry()` method will create a box with top and bottom faces, and apply a unique color to all eight vertices:

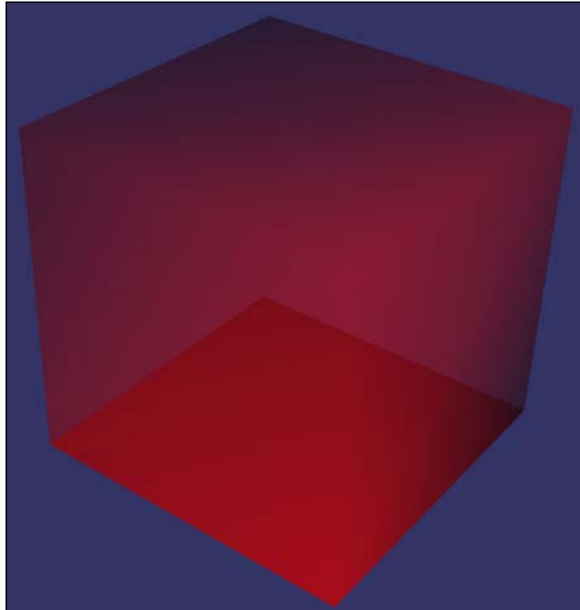
```
osg::Geometry* createSimpleGeometry()
{
  osg::ref_ptr<osg::Vec3Array> vertices =
    new osg::Vec3Array(8);
  (*vertices)[0].set(-0.5f,-0.5f,-0.5f);
  (*vertices)[1].set( 0.5f,-0.5f,-0.5f);
  (*vertices)[2].set( 0.5f, 0.5f,-0.5f);
  (*vertices)[3].set(-0.5f, 0.5f,-0.5f);
  (*vertices)[4].set(-0.5f,-0.5f, 0.5f);
  (*vertices)[5].set( 0.5f,-0.5f, 0.5f);
  (*vertices)[6].set( 0.5f, 0.5f, 0.5f);
  (*vertices)[7].set(-0.5f, 0.5f, 0.5f);
```

```
    osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
    (*colors)[0] = normalColor;

    osg::ref_ptr<osg::DrawElementsUInt> indices =
      new osg::DrawElementsUInt(GL_QUADS, 24);
    (*indices)[0] = 0; (*indices)[1] = 1; (*indices)[2] = 2;
      (*indices)[3] = 3;
    (*indices)[4] = 4; (*indices)[5] = 5; (*indices)[6] = 6;
      (*indices)[7] = 7;
    (*indices)[8] = 0; (*indices)[9] = 1; (*indices)[10]= 5;
      (*indices)[11]= 4;
    (*indices)[12]= 1; (*indices)[13]= 2; (*indices)[14]= 6;
      (*indices)[15]= 5;
    (*indices)[16]= 2; (*indices)[17]= 3; (*indices)[18]= 7;
      (*indices)[19]= 6;
    (*indices)[20]= 3; (*indices)[21]= 0; (*indices)[22]= 4;
      (*indices)[23]= 7;

    osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
    geom->setDataVariance( osg::Object::DYNAMIC );
    geom->setUseDisplayList( false );
    geom->setUseVertexBufferObjects( true );
    geom->setVertexArray( vertices.get() );
    geom->setColorArray( colors.get() );
    geom->setColorBinding( osg::Geometry::BIND_OVERALL );
    geom->addPrimitiveSet( indices.get() );

    osgUtil::SmoothingVisitor::smooth( *geom );
    return geom.release();
  }
```

7. In the main entry, we create the box geometry node and set it to the transparent bin:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createSimpleGeometry() );
geode->getOrCreateStateSet()->setMode(
  GL_BLEND, osg::StateAttribute::ON );
geode->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
Construct the scene graph and start the viewer:
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.addEventHandler( new SelectModelHandler );
viewer.setSceneData( root.get() );
return viewer.run();
```

8. Press *Ctrl* and select the box shown in the center. It will turn red, meaning that the model is selected by the user, as shown in the following screenshot:



How it works...

Here we have highlighted the geometry when the mouse is clickied on it. A `SelectModelHandler` object is used to check the intersections of the eye-direction line and the scene. And when we get any results in the overrode `doUserOperations()` method, we obtain the color array and change it. The VBO data should be updated after the operation.

To highlight a model is easy to achieve except for some prerequisites: The model must be an `osg::Geometry` object, and it should have already set a color array with at least one value to alter. Materials and textures applied on the model may also need to be considered because they also affect the final pixels.

There's more...

There are some other methods to mark a model as "selected", such as placing a bound box around it, drawing the wireframe of the model as an overlay (refer to the OSG example `osgscribe` for details) on it, or drawing an outline around the model (refer to the example `osgoutline` and the `osgFX::Cartoon` node for details).

Selecting a triangle face of the model

Let us go on with the last recipe. While editing a selected model in 3D world, you often have several choices: points, edges, faces (triangles or quads), and entity. Editing one or more faces of a model means to move, rotate, scale, remove, extrude, or do anything else you want to them. In the computer graphics design field, complex polygons including humans and monsters can be created from a simple box with the help of different face modifiers (also called **low-polygon modelization**).

Certainly they are out of the scope of this book, but we will explore the basis of all these advanced operations, that is, the selection of a triangle face of the model.

How to do it...

Let us start.

1. Include necessary headers and define color variables:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osg/PolygonOffset>
#include <osgUtil/SmoothingVisitor>
#include <osgViewer/Viewer>

const osg::Vec4 normalColor(1.0f, 1.0f, 1.0f, 1.0f);
const osg::Vec4 selectedColor(1.0f, 0.0f, 0.0f, 0.5f);
```

2. The `SelectModelHandler` class will manage a selector object (which represents the selected triangle face by overlapping and highlighting it) this time. It is used to represent the selected face when we pick the model:

```
class SelectModelHandler : public osgCookBook::PickHandler
{
public:
  SelectModelHandler() : _selector(0) {}

  osg::Geode* createFaceSelector();

  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::Intersection& result );

protected:
  osg::ref_ptr<osg::Geometry> _selector;
};
```

3. In the `createFaceSelector()` method, the selector geometry should be allocated with three vertices for picking up triangle faces. All the vertices of the selector are reset to the origin so that we won't see it at the beginning. When a face is selected, this geometry will be reset to overlap the selected triangle and highlight it to indicate that it has been chosen:

```
osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
(*colors)[0] = selectedColor;

_selector = new osg::Geometry;
_selector->setDataVariance( osg::Object::DYNAMIC );
_selector->setUseDisplayList( false );
_selector->setUseVertexBufferObjects( true );
_selector->setVertexArray( new osg::Vec3Array(3) );
_selector->setColorArray( colors.get() );
_selector->setColorBinding( osg::Geometry::BIND_OVERALL );
_selector->addPrimitiveSet( new osg::DrawArrays(
  GL_TRIANGLES, 0, 3) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( _selector.get() );
geode->getOrCreateStateSet()->setMode(
  GL_LIGHTING, osg::StateAttribute::OFF );
geode->getOrCreateStateSet()->setMode(
  GL_BLEND, osg::StateAttribute::ON );
geode->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
return geode.release();
```

4. In the `doUserOperations()` method, while we get an intersection result, we have to first check if the selected geometry and its vertex attributes are valid, and obtain them for later use:

```
osg::Geometry* geom = dynamic_cast<osg::Geometry*>(
  result.drawable.get() );
if ( !geom || !_selector || geom==_selector ) return;

osg::Vec3Array* vertices = dynamic_cast<osg::Vec3Array*>(
  geom->getVertexArray() );
osg::Vec3Array* selVertices = dynamic_cast<osg::Vec3Array*>(
  _selector->getVertexArray() );
if ( !vertices || !selVertices ) return;
```

5. Compute the local-to-world matrix of the selected model. This will help us obtain the correct vertex position of the selector. As we know, the `indexList` variable saves all triangles that have intersections with the mouse cursor, in order from the nearest to the farthest. Each triangle is recorded by pushing the indices of its three points into the list. Here we simply take them out and multiply each by the local-to-world matrix, reset the selector's points, and dirty it:

```
osg::Matrix matrix = osg::computeLocalToWorld( result.nodePath );
const std::vector<unsigned int>& selIndices =
  result.indexList;
for ( unsigned int i=0; i<3 && i<selIndices.size(); ++i )
{
  unsigned int pos = selIndices[i];
  (*selVertices)[i] = (*vertices)[pos] * matrix;
}
// Dirty the selector geometry to highlight the picked face
selVertices->dirty();
_selector->dirtyBound();
```

   The `createSimpleGeometry()` function doesn't have any differences from the one in the *Selecting and highlighting a model* recipe.

6. In the main entry, we apply an extra `osg::PolygonOffset` attribute on the simple box geometry for test. The reason is clear: The selected face geometry will overlap with one triangle face of the box because they have the same vertex values, but OpenGL can't handle such cases and will confuse the two faces while rendering the scene. Using the polygon offset functionality is a suitable solution in this example:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createSimpleGeometry() );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::PolygonOffset(1.0f, 1.0f) );
```

7. The box is added as the child of a transformation node. And then we can change the transformation matrix to test if our example code works for models placed anywhere:

```
osg::ref_ptr<osg::MatrixTransform> trans =
  new osg::MatrixTransform;
trans->addChild( geode.get() );
trans->setMatrix( osg::Matrix::translate(0.0f, 0.0f, 1.0f) );
```

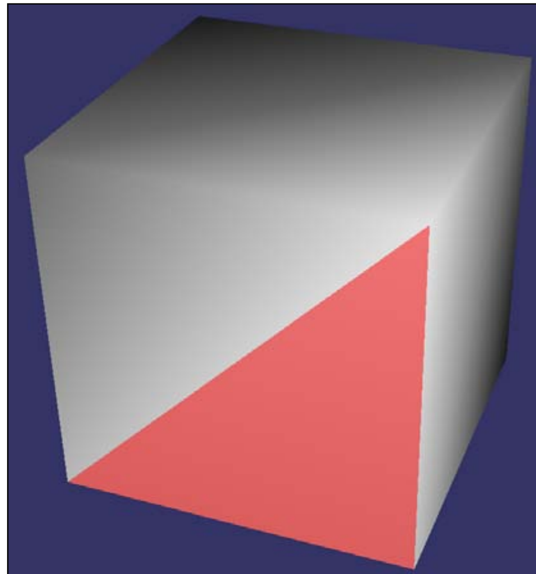
8. Add the selected geometry face to the root node, along with the model node:

```
osg::ref_ptr<SelectModelHandler> selector =
  new SelectModelHandler;

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( trans.get() );
root->addChild( selector->createFaceSelector() );
```

9. Start the viewer:

```
osgViewer::Viewer viewer;
viewer.addEventHandler( selector.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

10. Press *Ctrl* and click on any place on the box model. The `osg::MatrixTransform` node here can demonstrate the importance of converting vertices from local to world. Try commenting the use of `osg::computeLocalToWorld()` and see what will happen.



How it works...

Maybe you are still looking for a way to change the face's color to highlight it. Unfortunately it is impossible to modify the face color in OpenGL because color is in fact a vertex attribute. Shaders may help represent a specified face with a different color or transparency, but it seems to make a mountain out of a molehill in this recipe.

Be careful that the face selector node should be placed under the root node (in the world coordinate system); otherwise, you may have to consider applying an additional transformation matrix from the world to the selector's local coordinates while computing its vertices. The additional matrix is shown in the following block of code:

```
osg::Matrix matrix = osg::computeLocalToWorld( result.nodePath );
osg::Matrix matrix2 = osg::computeWorldToLocal(
  faceSelector->getParentalNodePaths()[0] );
```

```
for ( unsigned int i=0; i<3 && i<selIndices.size(); ++i )
{
  unsigned int pos = selIndices[i];
  (*selVertices)[i] = (*vertices)[pos] * matrix * matrix2;
}
```

There's more...

The classic 'Z-lighting' problem comes out when primitives are coplanar, or they are too near so that their depth values can't be distinguished. In this case, we have to use polygon offset to force adjusting the depth result of one or more of these primitives. More details can be found at `http://www.opengl.org/resources/faq/technical/polygonoffset.htm`.

Selecting a point on the model

The next task of selecting parts of a model is a little challenging: We are going to select an existing point of the model. Points and edges are useful when editing the polygon's topological structure. For example, you can specify one point or edge and collapse all faces sharing it on to a new point or line, which is the basic step of some polygon-simplification algorithms. But no doubt, the first work is to correctly select the point as before.

To note, selecting points is not as easy as selecting a face. The latter has area and can intersect with user-defined line segments. But how can we just "pick up" a point using a line? The solution is: Compute the distances of all possible points and the intersection point on the model, and mark one as selected if the distance is short enough.

How to do it...

Let us start.

1. Include necessary headers and define color variables:

   ```
   #include <osg/Geometry>
   #include <osg/Geode>
   #include <osg/MatrixTransform>
   #include <osg/Point>
   #include <osg/PolygonOffset>
   #include <osgUtil/SmoothingVisitor>
   #include <osgViewer/Viewer>

   const osg::Vec4 normalColor(1.0f, 1.0f, 1.0f, 1.0f);
   const osg::Vec4 selectedColor(1.0f, 0.0f, 0.0f, 1.0f);
   ```

2. This is the third time we meet the `SelectModelHandler` class. In this example, it provides a selection geometry that only contains one vertex to show where the point is:

```
class SelectModelHandler : public osgCookBook::PickHandler
{
public:
  SelectModelHandler( osg::Camera* camera )
  : _selector(0), _camera(camera) {}

  osg::Geode* createPointSelector();

  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::  Intersection& result );

protected:
  osg::ref_ptr<osg::Geometry> _selector;
  osg::observer_ptr<osg::Camera> _camera;
};
```

3. In the `createPointSelector()` method, allocate the selector geometry with one vertex here. To make it clear while rendering, we have to specify the point size attribute too:

```
osg::ref_ptr<osg::Vec4Array> colors = new osg::Vec4Array(1);
(*colors)[0] = selectedColor;

_selector = new osg::Geometry;
_selector->setDataVariance( osg::Object::DYNAMIC );
_selector->setUseDisplayList( false );
_selector->setUseVertexBufferObjects( true );
_selector->setVertexArray( new osg::Vec3Array(1) );
_selector->setColorArray( colors.get() );
_selector->setColorBinding( osg::Geometry::BIND_OVERALL );
_selector->addPrimitiveSet( new osg::DrawArrays(
  GL_POINTS, 0, 1) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( _selector.get() );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::Point(10.0f) );
geode->getOrCreateStateSet()->setMode(
  GL_LIGHTING, osg::StateAttribute::OFF );
return geode.release();
```

4. In the `doUserOperations()` method, acquire the necessary variables from the picked geometry and the selection:

```
osg::Geometry* geom = dynamic_cast<osg::Geometry*>(
  result.drawable.get() );
if ( !geom || !_selector || geom==_selector ) return;

osg::Vec3Array* vertices = dynamic_cast<osg::Vec3Array*>(
  geom->getVertexArray() );
osg::Vec3Array* selVertices = dynamic_cast<osg::Vec3Array*>(
  _selector->getVertexArray() );
if ( !vertices || !selVertices ) return;
```

5. Compute the world-intersection point and the matrix for converting the selected model into world coordinates. Then, we are going to convert both the point and the matrix to projection coordinates, in which the vertex is limited in range from `[-1,-1,-1]` to `[1, 1, 1]`. We will explain the reason for this later.

```
osg::Vec3 point = result.getWorldIntersectPoint();
osg::Matrix matrix = osg::computeLocalToWorld(
  result.nodePath );

osg::Matrix vpMatrix;
if ( _camera.valid() )
{
  vpMatrix = _camera->getViewMatrix() * _camera-
    >getProjectionMatrix();
  point = point * vpMatrix;
}
```

6. Find out all three vertices of the nearest picked triangle, and compute the distance between the intersection point and each of them. If any one of the distances is less than a fixed threshold (`0.1`), we will say that corresponding point is "picked up". To note, the distance values and the threshold are computed with regard to the projection coordinate system:

```
const std::vector<unsigned int>& selIndices =
  result.indexList;
for ( unsigned int i=0; i<3 && i<selIndices.size(); ++i )
{
  unsigned int pos = selIndices[i];
  osg::Vec3 vertex = (*vertices)[pos] * matrix;
  float distance = (vertex * vpMatrix - point).length();
  if ( distance<0.1f )
  {
    selVertices->front() = vertex;
  }
```

```
}
// Dirty the selector geometry to highlight the picked point
selVertices->dirty();
_selector->dirtyBound();
```

The `createSimpleGeometry()` function, of course, has no changes in all three recipes.

7. In the main entry, we will create the sample model with polygon-offset settings, and add it to a transformation node:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createSimpleGeometry() );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::PolygonOffset(1.0f, 1.0f) );

osg::ref_ptr<osg::MatrixTransform> trans =
  new osg::MatrixTransform;
trans->addChild( geode.get() );
trans->setMatrix( osg::Matrix::translate(0.0f, 0.0f, 1.0f) );
```

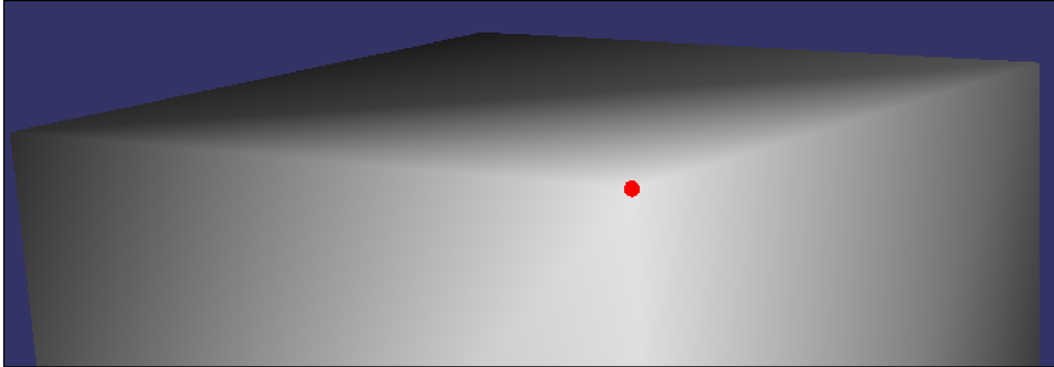
8. Create the selecting handler and add the selected point object to the scene graph:

```
osgViewer::Viewer viewer;
osg::ref_ptr<SelectModelHandler> selector =
  new SelectModelHandler( viewer.getCamera() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( trans.get() );
root->addChild( selector->createPointSelector() );
viewer.addEventHandler( selector.get() );
viewer.setSceneData( root.get() );
```

9. The last thing to pay attention to before starting the viewer is to disable the **small feature culling** mode, with which the back-end rendering will ignore geometries with one vertex automatically:

```
osg::CullSettings::CullingMode mode =
  viewer.getCamera()->getCullingMode();
viewer.getCamera()->setCullingMode( mode &
  (~osg::CullSettings::SMALL_FEATURE_CULLING) );
return viewer.run();
```

10. Press *Ctrl* and click on an end point approximately (in fact you can't precisely pick it). If the cursor is near enough, the point will be selected and a red, bigger point is shown; otherwise, nothing will happen as your cursor is still too far away from your target.



How it works...

In step 6, we use a slightly complex method to compute the distance from a vertex of the selected triangle to the selection point. Both points are transformed into the projection coordinate system before obtaining the distance:

```
float distance = (vertex * vpMatrix - point).length();
```

You may have a question—is it OK if we ignore the view and projection matrix and directly compute distances in the world coordinates? The answer is yes, but the results may not be precise. When we zoom out of the camera and go far away from the model, selecting points becomes a difficult task as we can hardly put the mouse cursor near enough to the expected vertex. That is because the threshold for deciding the selectable distance is fixed, but the pixel size of the model differs according to the current camera.

Now you can imagine why we add an extra matrix transformation here: We transform the points to projection coordinates, and compare the distance with the threshold too. View and projection matrices are not influencing factors anymore, which make the point selection easier for end users.

This solution has another problem: The picking operation must first have an intersection with the model, and then we can check which points are most likely to be selected. If the user clicks a position very near to a point but doesn't have intersections with the model itself, the whole process will fail. In that case, we may have to consider designing a new intersector derived from the `osgUtil::Intersector` class, which will be discussed in the last chapter of this book.

There's more...

This is the first time we come across the concept of small feature culling, or contribution culling. In a word, this is a culling method throwing objects away that do not contribute to the final rendering result.

Using vertex-displacement mapping in shaders

Is this the first time you have heard the name **displacement mapping**? Don't worry. It is just a kind of modern computer-graphics technique. Maybe you are familiar with **bump mapping**, which simulates bumps using a special texture map and makes the result look more realistic. Yes, it has some similar points with **displacement mapping**—both have a smooth surface at the beginning; both make uses of shaders for special effects; and both read data from textures working like parameter lookup tables.

Vertex displacement mapping, as the name suggests, uses textures to modify vertex positions and normals instead of just pixels. It produces dynamic, detailed, and real mesh data, not faked ones (**bump mapping** instead fakes the results).

How to do it...

Let us start.

1. Include necessary headers:

   ```
   #include <osg/Geometry>
   #include <osg/Geode>
   #include <osg/Texture2D>
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   ```

2. The core feature of vertex displacement mapping is to use the texture values to alter vertex positions. This can sometimes generate rough surfaces from one or more textures. In our vertex shader, this is done by applying a simple value, which is read from the texture parameter on each vertex's Z coordinate (the height):

   ```
   const char* vertCode = {
     "uniform sampler2D defaultTex;\n"
     "varying float height;\n"
     "void main()\n"
     "{\n"
       "vec2 uv = gl_MultiTexCoord0.xy;\n"
       "vec4 color = texture2D(defaultTex, uv);\n"
   ```

```
    "height = 0.3*color.x + 0.59*color.y + 0.11*color.z;\n"

    "vec4 pos = gl_Vertex;\n"
    "pos.z = pos.z + 100.0*height;\n"
    "gl_Position = gl_ModelViewProjectionMatrix * pos;\n"
  "}\n"
};
```

3. The fragment shader will decide the pixel color according to the height value. Lower areas are painted with a very dark gray color, and highlands are painted with green:

```
const char* fragCode = {
  "varying float height;\n"
  "const vec4 lowerColor = vec4(0.1, 0.1, 0.1, 1.0);\n"
  "const vec4 higherColor = vec4(0.2, 1.0, 0.2, 1.0);\n"
  "void main()\n"
  "{\n"
    "gl_FragColor = mix(lowerColor, higherColor, height);\n"
    // height won't go beyond 1.0 in this recipe
  "}\n"
};
```

4. Create grid geometry as the displacement mapping container. It is in fact a two-dimensional regular grid object, in which each grid cell has a unique `(x, y)` coordinates. By setting different Z values to these cells, we can thus create 3D terrains easily:

```
osg::Geometry* createGridGeometry( unsigned int column,
  unsigned int row )
{
  ...
}
```

5. Create the grid points and texture coordinates. The latter is much more important as it will be used to read **texels** from the texture object:

```
osg::ref_ptr<osg::Vec3Array> vertices =
  new osg::Vec3Array(column * row);
osg::ref_ptr<osg::Vec2Array> texcoords =
  new osg::Vec2Array(column * row);
for ( unsigned int i=0; i<row; ++i )
{
  for ( unsigned int j=0; j<column; ++j )
  {
    (*vertices)[i*column + j].set( (float)i, (float)j, 0.0f );
    (*texcoords)[i*column + j].set( (float)i/(float)row,
      (float)j/(float)column );
  }
}
```

6. Allocate the geometry and assemble vertices. The `GL_QUAD_STRIP` parameter is suitable here for building such grid geometries:

```
osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
geom->setUseDisplayList( false );
geom->setUseVertexBufferObjects( true );
geom->setVertexArray( vertices.get() );
geom->setTexCoordArray( 0, texcoords.get() );
for ( unsigned int i=0; i<row-1; ++i )
{
  osg::ref_ptr<osg::DrawElementsUInt> de =
    new osg::DrawElementsUInt(GL_QUAD_STRIP, column*2);
  for ( unsigned int j=0; j<column; ++j )
  {
    (*de)[j*2 + 0] = i*column + j;
    (*de)[j*2 + 1] = (i+1)*column + j;
  }
  geom->addPrimitiveSet( de.get() );
}
```

7. Set a customized bounding box here:

```
geom->setInitialBound( osg::BoundingBox(
  -1.0f,-1.0f,-100.0f, 1.0f, 1.0f, 100.0f) );
```

8. Set up the texture and shader attributes. Here we use `LINEAR` to replace the regular `LINEAR_MIPMAP_LINEAR` parameter to set the texture-minifying function. This will disable texture mipmapping, which is of no use in this recipe that treats the texture map as a parameter table:

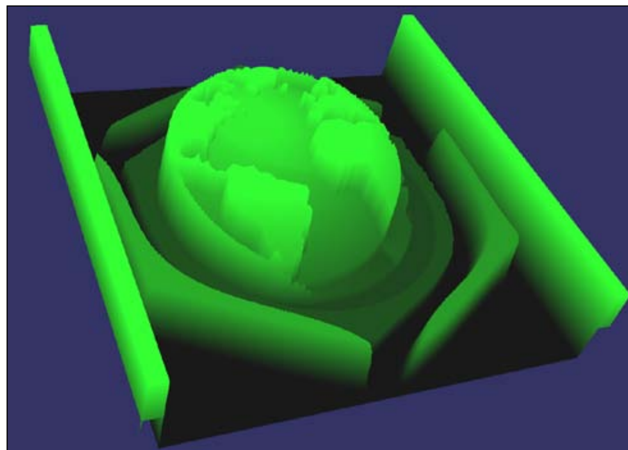
```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( osgDB::readImageFile("Images/osg256.png") );
texture->setFilter( osg::Texture2D::MIN_FILTER,
  osg::Texture2D::LINEAR );
texture->setFilter( osg::Texture2D::MAG_FILTER,
  osg::Texture2D::LINEAR );
geom->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, texture.get() );
geom->getOrCreateStateSet()->addUniform(
  new osg::Uniform("defaultTex", 0) );

osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertCode) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  fragCode) );
```

```
geom->getOrCreateStateSet()->setAttributeAndModes(
  program.get() );
return geom.release();
```

9. In the main entry, there is nothing too much to do. We will just add the grid geometry to the scene graph and start rendering it:

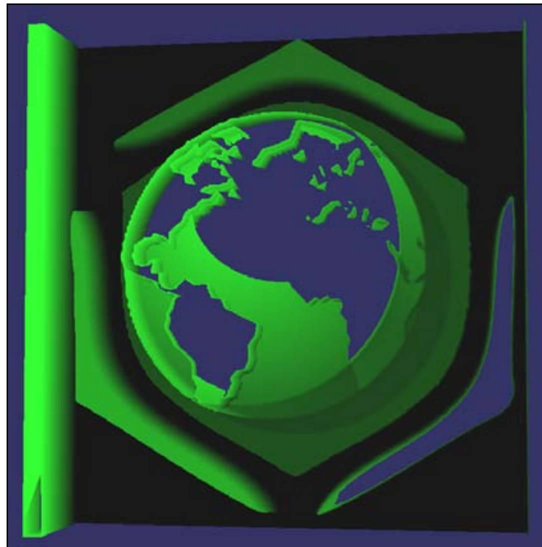
```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createGridGeometry(512, 512) );
geode->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

10. We will choose the OpenSceneGraph logo picture as the texture. Dark parts of this picture form the lowlands, and light parts form mountains and highlands. A real **digital elevation model** (**DEM**) image may implement a much better scene, but it certainly requires more vertices and a large texture resolution.



How it works...

Here we set up a customized bound for the geometry with the `setInitialBound()` method. You may remove it from the program and rebuild to see what the difference is. You will get an image as shown in the following screenshot:



Look strange? Part of the generated model is clipped and replaced by the background. The reason is clear: OSG will automatically compute near and far planes of the projection matrix according to the bounds of scene objects, but it can never know what you have done in shaders. It calculates the bounding box of the geometry referring to the vertices stored in the CPU memory, and ignores position changes in the Z direction.

This leads to wrong near/far values that may clip the geometry in an improper way. To solve this, we would better determine the bounds of special drawables by ourselves. And that is the reason why we use the `setInitialBound()` method here.

There's more...

For more information about displacement, bump, and normal mappings, visit the following websites:

`http://en.wikipedia.org/wiki/Displacement_mapping`

`http://en.wikipedia.org/wiki/Bump_mapping`

`http://en.wikipedia.org/wiki/Normal_mapping`

The osgFX library also has a bump mapping implementation. Read its source code and the example `osgfxbrowser` if you have further interest.

Using the draw instanced extension

It is common in modern 3D applications that a scene can be filled with a huge number of small geometries that represent particles, trees, or people crowds. Rendering such a large number of polygons, no matter how simple they are, will be a heavy burden for computer graphics hardware and APIs. The whole operation is really slow, especially when you are submitting mess data to the graphics pipeline.

In this case, the implementation of hardware geometry instancing (called **draw instanced** in OpenGL) will be of much importance. It enables the same geometry object, or the same sets of vertices and primitives, to be instanced many times and rendered with different transformations. It reduces the number of OpenGL command calls and usage of duplicated data, and makes it possible to render a mess scene full of the same geometries more efficiently. Of course, shaders must be used here to handle any instance of the geometry object.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/Texture2D>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The vertex shader defines the behaviors of instance objects and draws them according to certain rules. We will explain the concrete program later in the *How it works* section.

```
const char* vertCode = {
  "uniform sampler2D defaultTex;\n"
  "const float PI2 = 6.2831852;\n"
  "void main()\n"
  "{\n"
  "float r = float(gl_InstanceID) / 256.0;\n"
  "vec2 uv = vec2(fract(r), floor(r) / 256.0);\n"
  "vec4 pos = gl_Vertex + vec4(uv.s * 384.0, 32.0 *
    sin(uv.s * PI2), uv.t * 384.0, 1.0);\n"
  "gl_FrontColor = texture2D(defaultTex, uv);\n"
  "gl_Position = gl_ModelViewProjectionMatrix * pos;\n"
  "}\n"
};
```

3. Use the `createInstancedGeometry()` function to create multiple instances of the same geometry:

```
osg::Geometry* createInstancedGeometry(
  unsigned int numInstances )
{
  ...
}
```

4. We will only create a quad with four vertices, which is enough for demonstrating the usage here:

```
osg::ref_ptr<osg::Vec3Array> vertices = new osg::Vec3Array(4);
(*vertices)[0].set(-0.5f, 0.0f,-0.5f );
(*vertices)[1].set( 0.5f, 0.0f,-0.5f );
(*vertices)[2].set( 0.5f, 0.0f, 0.5f );
(*vertices)[3].set(-0.5f, 0.0f, 0.5f );

osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
geom->setUseDisplayList( false );
geom->setUseVertexBufferObjects( true );
geom->setVertexArray( vertices.get() );
```

5. Using the draw instanced extension may be easier than you think. Either the `osg::DrawArrays` or `osg::DrawElements*` class has a `numInstances` argument (at the last of the argument list) that indicates the number of instanced objects. Set a non-zero value to enable draw instanced. And set up a customized bounding box again as the system can't decide the actual bound according to only four original points:

```
geom->addPrimitiveSet( new osg::DrawArrays(
  GL_QUADS, 0, 4, numInstances) );
geom->setInitialBound( osg::BoundingBox(
  -1.0f,-32.0f,-1.0f, 192.0f, 32.0f, 192.0f) );
```

6. Apply the texture and shader attributes. This is exactly the same as the *Using vertex displacement mapping in shaders* recipe:

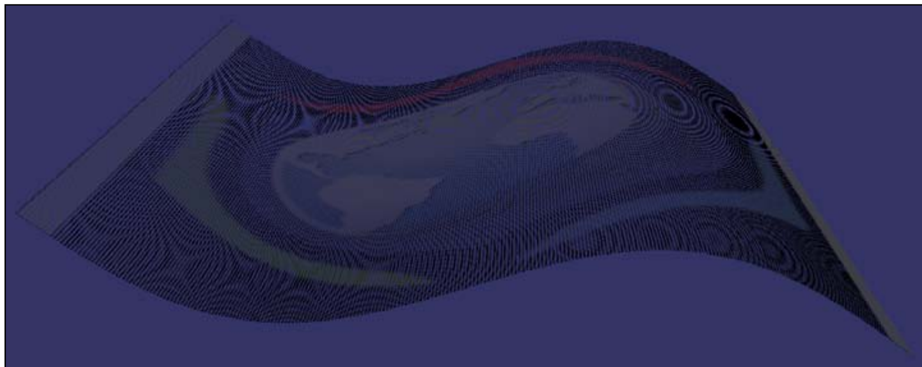
```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( osgDB::readImageFile("Images/osg256.png") );
texture->setFilter( osg::Texture2D::MIN_FILTER,
  osg::Texture2D::LINEAR );
texture->setFilter( osg::Texture2D::MAG_FILTER,
  osg::Texture2D::LINEAR );
geom->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, texture.get() );
geom->getOrCreateStateSet()->addUniform(
  new osg::Uniform("defaultTex", 0) );
```

```
osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertCode) );
geom->getOrCreateStateSet()->setAttributeAndModes(
  program.get() );
return geom.release();
```

7. OK, now we will get into the main entry; add the geometry object to the scene and start the viewer:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( createInstancedGeometry(256*256) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

8. You will see a large number of quads appearing in the 3D world, arranged in a sine surface. It's amazing because we only created one geometry with four points, but now it is over a thousand times greater! A lot of CPU memories and pipeline commands are saved with this great functionality.



How it works...

The draw instanced extension requires OpenGL 2.0 to work properly. It greatly reduces the memory usage of vertices and primitives on the CPU side, but can still perform as effectively as the traditional way to build geometries. It introduces a new read-only, built-in GLSL variable `gl_InstanceID`, which contains the current instance ID (from 0 to number of instances) in the rendering pipeline. With the texture as a parameter table, we can look for data corresponding to the ID and set `gl_Position`, `glTexCoord[*]`, and other outputs to suitable values. This makes it possible to set up the positions and attributes of a low-polygon human, and even some scientific visualization work, for instance, the rendering of point-cloud data (we will discover this in *Chapter 8*).

The `gl_Vertex` variable represents the same vertex data used by each instance, and so do the `gl_Normal` and `gl_MultiTexCoord*` variables. You must apply a transformation matrix or customized offset to them to move the instance to a different location in the 3D world.

In the shader code of this recipe, the `pos` variable represents the position of each instanced quad in the world coordinate (by adding an offset to the original `gl_Vertex` variable). Then, we multiply the model-view-projection (MVP) matrix with it to obtain the position in projection coordinate, which is actually required for final vertex composition in the rendering pipeline.

Note that the shader code here is not perfect because it fixes the number of row and column instances instead of using uniforms. You may try to alter it to provide a more scalable draw instanced implementation.

There's more...

More information about the OpenGL draw instanced support can be found at `http://www.opengl.org/registry/specs/ARB/draw_instanced.txt`.

And the OSG example `osgdrawinstanced` can also help in studying this interesting functionality. You can treat it as an upgraded version of this recipe.

4

Manipulating the View

In this chapter, we will cover:

- Setting up views on multiple screens
- Using slave cameras to simulate a power-wall
- Using depth partition to display huge scenes
- Implementing the radar map
- Showing the top, front, and side views of a model
- Manipulating the top, front, and side views
- Following a moving model
- Using manipulators to follow models
- Designing a 2D camera manipulator
- Manipulating the view with joysticks

Introduction

This chapter is all about the camera and camera manipulation. No matter how beautiful or realistic your scene is, a bad navigating experience will still drive your users away. Your goal is to try changing the view and projection matrices, which represent 3D space transformation, and thus change the world you can see from the camera smoothly and conveniently. But don't forget that manipulating with mouse and keyboard is actually a 2D interaction, with two degrees-of-freedom (DOF) and the mouse buttons. So, you can hardly describe a complete 3D movement (totally 6 DOFs) with mouse move and button click events.

Fortunately, OSG provides us a list of handy in-built camera manipulators, such as the `osgGA::TrackballManipulator`. It also contains a good framework (including manipulator base class and event handlers) for implementing our own navigation strategy. We will cover all these features in this specialized chapter.

Another interesting topic here is the usage of single viewer and composite viewer, as well as the configuration of viewing attributes and multi-monitor options. You will also find examples of them in the later recipes.

Setting up views on multiple screens

Today, more and more people have multiple physical display devices, especially multiple monitors (or screens) at work. Modern graphics cards can always afford at least two outputs nowadays. Also, most operating systems have already fully supported the simultaneous use of multiple screens, including Linux, Mac OSX, and Microsoft Windows.

However, programming multiple screens is still challenging because developers have to handle the graphics buffer of each screen by themselves, which is painful for many inexperienced developers. Fortunately, OSG encapsulates the platform-specific multi-monitor APIs and uses the context ID to characterize screens. It enables users to create and use one or more graphics contexts on each screen, and write OSG programs without having any discomfort.

How to do it...

Carry out the following steps in order to complete the recipe:

1. Include necessary headers:

```
#include <osg/Camera>
#include <osgDB/ReadFile>
#include <osgGA/TrackballManipulator>
#include <osgViewer/CompositeViewer>
```

2. The `createView()` function will create a full-screen window on specified screen:

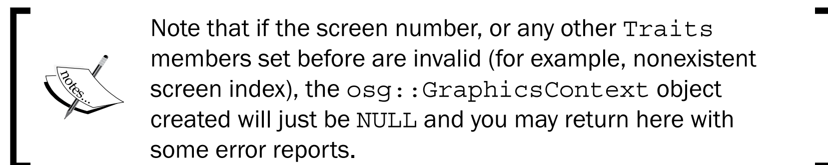
```
osgViewer::View* createView( int screenNum )
{
  ...
}
```

3. Let us configure the desired rendering window attributes according to current screen settings in the function. We may obtain the desired screen size and attributes by calling the `WindowingSystemInterface` class:

```
unsigned int width = 800, height = 600;
osg::GraphicsContext::WindowingSystemInterface* wsi = osg::Graphic
```

```
Context::getWindowingSystemInterface();
if ( wsi )
    wsi->getScreenResolution( osg::GraphicsContext::
ScreenIdentifier(screenNum), width, height );

osg::ref_ptr<osg::GraphicsContext::Traits> traits = new
osg::GraphicsContext::Traits;
traits->screenNum = screenNum;
traits->x = 0;
traits->y = 0;
traits->width = width;
traits->height = height;
traits->windowDecoration = false;
traits->doubleBuffer = true;
traits->sharedContext = 0;
```

4. Construct the graphics context and attach it to a camera to provide a usable rendering window here. The viewport of the camera should also receive the size parameters to display the scene in the full-screen range:



Note that if the screen number, or any other `Traits` members set before are invalid (for example, nonexistent screen index), the `osg::GraphicsContext` object created will just be `NULL` and you may return here with some error reports.

```
osg::ref_ptr<osg::GraphicsContext> gc = osg::GraphicsContext::crea
teGraphicsContext( traits.get() );
if ( !gc ) return NULL;

osg::ref_ptr<osg::Camera> camera = new osg::Camera;
camera->setGraphicsContext( gc.get() );
camera->setViewport( new osg::Viewport(0, 0, width, height) );
camera->setProjectionMatrixAsPerspective(
    30.0f, static_cast<double>(width)/static_cast<double>(height),
1.0f, 10000.0f );

GLenum buffer = traits->doubleBuffer ? GL_BACK : GL_FRONT;
camera->setDrawBuffer( buffer );
camera->setReadBuffer( buffer );
```

5. Create a new view object, set up a default camera manipulator, and return it at the end:

```
osg::ref_ptr<osgViewer::View> view = new osgViewer::View;
view->setCamera( camera.get() );
view->setCameraManipulator( new osgGA::TrackballManipulator );
return view.release();
```

6. In the main entry, we will make use of `osgViewer::CompositeViewer` class in this recipe:

```
osgViewer::CompositeViewer viewer;
```

7. Create a view with any model shown on the first screen:

```
osgViewer::View* view1 = createView( 0 );
if ( view1 )
{
  view1->setSceneData( osgDB::readNodeFile("cessna.osg") );
  viewer.addView( view1 );
}
```

8. Create yet another view on the second screen. Make sure you have enough monitors and already connect them to your graphics card:

```
osgViewer::View* view2 = createView( 1 );
if ( view2 )
{
  view2->setSceneData( osgDB::readNodeFile("cow.osg") );
  viewer.addView( view2 );
}
```

9. Start the viewer:

```
return viewer.run();
```

10. If you have at least two monitors, you will be able to see a Cessna on the main monitor, and a cow model on the other one; otherwise, you can only render the main scene properly, and you may read some failure information in the console window, which indicates that the second view cannot be initialized.

How it works...

In the `createView()` function, we can easily obtain the screen size and other attributes by calling the `WindowingSystemInterface` class, which will have the access to platform-specific Windowing APIs. Then we pass the acquired width and height values to a `Traits` object. Of course, do not forget the `screenNum` value, which is the key for setting up multiple views on multiple displays.

The only difference between single-screen and multi-screen programming in OSG is to set up the screen index (`screenNum`) variable of the `Traits` class. Anything else that is related with a specific platform will be handled by OSG automatically and screen information will be recorded by the `WindowingSystemInterface` class.

There's more...

You may retrieve the number of screens connected with current computer by calling the `getNumScreens()` method of `WindowingSystemInterface`:

```
osg::GraphicsContext::WindowingSystemInterface* wsi =
    osg::GraphicsContext::getWindowingSystemInterface();
if ( wsi ) numScreens = wsi->getNumScreens();
```

In order to obtain the detailed information of a screen (screen index is `num`):

```
osg::GraphicsContext::ScreenSettings settings;
osg::GraphicsContext::WindowingSystemInterface* wsi = osg::Graphic
sContext::getWindowingSystemInterface();
if ( wsi ) wsi->getScreenSettings(
    osg::GraphicsContext::ScreenIdentifier(num), settings );
```

You can get the screen width, height, refresh rate, and color buffer depth attributes from the settings variable.

The example `osgcamera` in the core source code can exactly explain how to display a scene with single and multiple screens, using the screen number identifier.

Using slave cameras to simulate a power-wall

The **PowerWall** is a common virtual reality system displaying extremely high resolution scenes by grouping tiled arrays of projectors or monitors sharing the same data. With the really large screen composed by many small screens, it is possible to see the world clearly and find detailed information effectively.

OSG also provides the **slave camera** feature to implement the same functionality.
Each salve represents a small tile in the PowerWall. You can assign the slave camera to different numbers of screens, but in this recipe, we will simply put them in one screen.

How to do it...

Carry out the following steps in order to complete the recipe:

1. Include necessary headers:

```
#include <osg/Camera>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The `createSlaveCamera()` is used to create cameras following the viewer's main camera. The creation process is similar to the *Setting up views on multiple screens* recipe before. An `osg::Camera` node will be returned, which is already attached with a rendering window:

```
osg::Camera* createSlaveCamera( int x, int y, int width, int
height )
{
    osg::ref_ptr<osg::GraphicsContext::Traits> traits = new
osg::GraphicsContext::Traits;
    traits->screenNum = 0;  // this can be changed for
                           //multi-display
  traits->x = x;
  traits->y = y;
  traits->width = width;
  traits->height = height;
  traits->windowDecoration = false;
  traits->doubleBuffer = true;
  traits->sharedContext = 0;

  osg::ref_ptr<osg::GraphicsContext> gc =    osg::GraphicsContext:
:createGraphicsContext( traits.get() );
  if ( !gc ) return NULL;

  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  camera->setGraphicsContext( gc.get() );
  camera->setViewport( new osg::Viewport(0, 0, width, height) );

  GLenum buffer = traits->doubleBuffer ? GL_BACK : GL_FRONT;
  camera->setDrawBuffer( buffer );
  camera->setReadBuffer( buffer );
  return camera.release();
}
```

3. In the main entry, we will allow the user to decide the row and column numbers of the PowerWall. The greater these two numbers are, the more displays you are going to have, but the side effect is that you will have a less effective system containing too many rendering windows:

```
osg::ArgumentParser arguments( &argc, argv );

int totalWidth = 1024, totalHeight = 768;
arguments.read( "--total-width", totalWidth );
arguments.read( "--total-height", totalHeight );

int numColumns = 3, numRows = 3;
arguments.read( "--num-columns", numColumns );
arguments.read( "--num-rows", numRows );
```

4. Read any model for displaying on all the windows.

```
osg::ref_ptr<osg::Node> scene= osgDB::readNodeFiles(arguments);
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

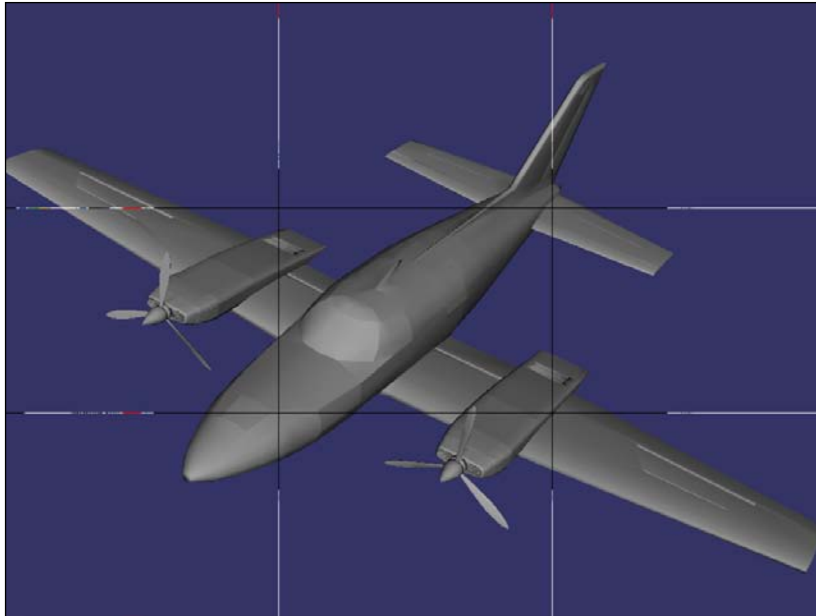
5. Now we are going to add each camera created as a 'slave' into the viewer. A slave camera doesn't have independent view and projection matrices. It uses the main camera's view and adds an offset to each matrix. The computation process will be explained later in the *How it works...* section:

```
osgViewer::Viewer viewer;

int tileWidth = totalWidth / numColumns;
int tileHeight = totalHeight / numRows;
for ( int row=0; row<numRows; ++row )
{
  for ( int col=0; col<numColumns; ++col )
  {
    osg::Camera* camera = createSlaveCamera(
        tileWidth*col, totalHeight - tileHeight*(row+1),
tileWidth-1, tileHeight-1 );
    osg::Matrix projOffset =
        osg::Matrix::scale(numColumns, numRows, 1.0) *
        osg::Matrix::translate(numColumns-1-2*col,
          numRows-1-2*row, 0.0);
    viewer.addSlave( camera, projOffset, osg::Matrix(), true );
  }
}
```

6. Add the scene to the viewer and start the simulation:

```
viewer.setSceneData( scene );
return viewer.run();
```

7. Now, you will find nine windows making up a wall that display one complete scene, as shown in the following screenshot. The power-wall implementation can actually be larger and offer really high-resolution result in practical use. However, it requires more screens or even more computers to collaborate together. This is already beyond the scope of our book:

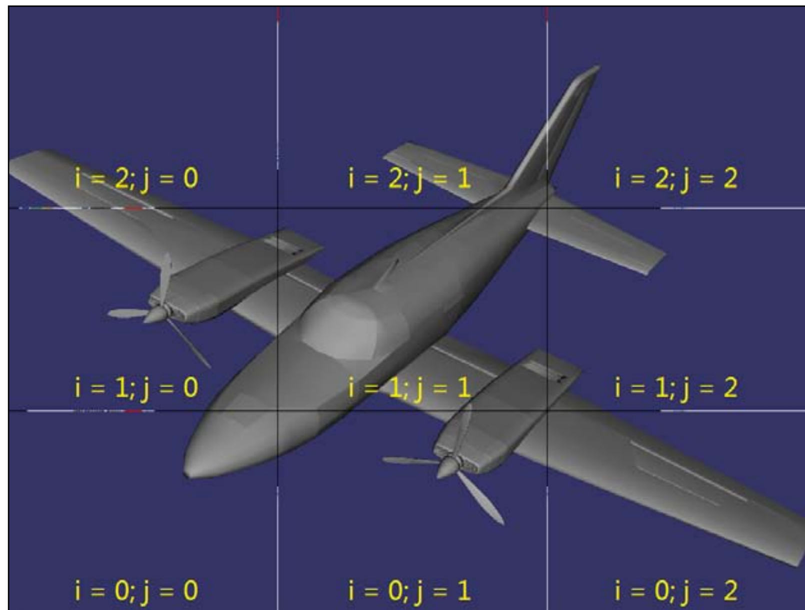


How it works...

The slave camera will read the main camera's view and projection matrices and multiply each with an offset matrix, that is:

```
slaveViewMatrix = mainViewMatrix * viewOffset;
slaveProjectionMatrix = mainProjectionMatrix * projectionOffset;
```

In order to compute the offset matrices here, we must first consider how cameras are arranged in a PowerWall. Let us have a look at the result picture again, with column and row number marked:



In order to render the scene correctly in every slave camera, we can just keep the view matrix as it is, and reset parameters of the frustum of a slave.

The projection matrix can be written as:

$$M_{proj} = \begin{bmatrix} \frac{2 * znear}{right - left} & 0 & 0 & 0 \\ 0 & \frac{2 * znear}{top - bottom} & 0 & 0 \\ \frac{right + left}{right - left} & \frac{top + bottom}{top - bottom} & -\frac{zfar + znear}{zfar - znear} & -1 \\ 0 & 0 & -\frac{2 * zfar * znear}{zfar - znear} & 0 \end{bmatrix}$$

Here **left**, **right**, **top**, and **bottom** specify the horizontal and vertical clipping planes of the frustum, **znear** and **zfar** represent the near and far clipping planes. See the OpenGL document for details:

`http://www.opengl.org/sdk/docs/man/xhtml/glFrustum.xml`

In order to fit the row and column positions of a certain slave camera, we can scale the horizontal and vertical coordinates and reproduce the matrix formula as:

$$M_{tile} = \begin{bmatrix} \frac{2*znear}{right'-left'} & 0 & 0 & 0 \\ 0 & \frac{2*znear}{top'-bottom'} & 0 & 0 \\ \frac{right'+left'}{right'-left'} & \frac{top'+bottom'}{top'-bottom'} & -\frac{zfar+znear}{zfar-znear} & -1 \\ 0 & 0 & -\frac{2*zfar*znear}{zfar-znear} & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{2*znear}{(right-left)/numCols} & 0 & 0 & 0 \\ 0 & \frac{2*znear}{(top-bottom)/numRows} & 0 & 0 \\ \frac{right+left}{(right-left)/numCols} - A & \frac{top+bottom}{(top-bottom)/numRows} - B & -\frac{zfar+znear}{zfar-znear} & -1 \\ 0 & 0 & -\frac{2*zfar*znear}{zfar-znear} & 0 \end{bmatrix}$$

$$= M_{proj} * \begin{bmatrix} numCols & 0 & 0 & 0 \\ 0 & numRows & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ A & B & 0 & 1 \end{bmatrix}$$

Here,

```
left' = left + j *(right - left) / numCols
right' = right - (numCols - j - 1)*(right - left) / numCols
bottom' = bottom + i *(top - bottom) / numRows
top' = top - (numRows - i - 1)*(top - bottom) / numRows
A = numCols - 2 * j - 1
B = numRows - 2 * i - 1
```

The variable `left'`, `right'`, `top'` and `bottom'` are parameters of the slave's projection matrix, which are the same meanings of the OpenGL frustum variables and that is what we have already done in the previous section.

There's more...

Slave cameras are allocated and managed by only one computer. This brings a new problem: how could one system afford too many displays and rendering tasks? Maybe you would like to have more computers, and you may set up a rendering cluster with PCs and workstations with monitors. For this case, have a look at the `osgcluster` example, and consider handling the synchronization of computers by yourself.

Using depth partition to display huge scenes

Is it possible to draw the real sun, earth, mars, and even the solar system in OpenGL and OSG? The answer is absolutely yes. But you may encounter some problems while actually working on this topic.

The most serious one is the computation of near and far planes. As many early devices have only a 16-bit depth buffer or 24-bit one, you may not be able to maintain a huge distance (in the solar system) between the near and the far plane. If you force setting the near plane to a small value and the far plane to a very large one, the rendering of nearby objects will cause the classic Z-fighting problem because there is not enough precision to resolve the distance. The explaination of Z-fighting can be found at: `http://en.wikipedia.org/wiki/Z-fighting`.

The best solution is to buy a new graphics card supporting a 32-bit buffer. But to keep the program portable, we had better find some other solutions, for instance, the depth partition algorithm in this recipe. To introduce this algorithm in one sentence—it partitions the scene into several parts and renders them separately. Every part has its own near and far values and the distance between them is short enough to avoid the precision problem.

How to do it...

Carry out the following steps in order to complete the recipe:

1. Include necessary headers:

```
#include <osg/Texture2D>
#include <osg/ShapeDrawable>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgGA/TrackballManipulator>
#include <osgViewer/Viewer>
```

2. The earth and sun radius, as well as the astronomical unit (AU) are real data and should be set as constant ones. The distance between the earth and the sun is approximately 1 AU.

```
const double radius_earth = 6378.137;
const double radius_sun = 695990.0;
const double AU = 149697900.0;
```

3. The `createScene()` function will create the huge scene which contains a sun and an earth. Compared to our previous scene such as Cessna or a cow, this time it is really a vast expanse.

```
osg::Node* createScene()
{
    ...
}
```

4. Create the earth node and apply a texture to make it more realistic:

```
osg::ref_ptr<osg::ShapeDrawable> earth_sphere = new
  osg::ShapeDrawable;
earth_sphere->setShape( new osg::Sphere(osg::Vec3(),
  radius_earth) );

osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage(
  osgDB::readImageFile("Images/land_shallow_topo_2048.jpg") );

osg::ref_ptr<osg::Geode> earth_node = new osg::Geode;
earth_node->addDrawable( earth_sphere.get() );
earth_node->getOrCreateStateSet()

  ->setTextureAttributeAndModes( 0, texture.get() );
```

5. Create the sun sphere and set its color and radius. We will use a transformation node to move it far from the earth's position, that is, the point of origin:

```
osg::ref_ptr<osg::ShapeDrawable> sun_sphere = new
osg::ShapeDrawable;
sun_sphere->setShape( new osg::Sphere(osg::Vec3(), radius_sun) );
sun_sphere->setColor( osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) );

osg::ref_ptr<osg::Geode> sun_geode = new osg::Geode;
sun_geode->addDrawable( sun_sphere.get() );

osg::ref_ptr<osg::MatrixTransform> sun_node =
  new osg::MatrixTransform;
sun_node->setMatrix( osg::Matrix::translate(0.0, AU, 0.0) );
sun_node->addChild( sun_geode.get() );
```

6. Now, create the scene graph:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( earth_node.get() );
root->addChild( sun_node.get() );
return root.release();
```

7. In the main entry, first we have to set the `depth partition` ranges: the near, middle, and far planes along the Z axis in eye coordinates. Note that there is an additional middle plane compared to the traditional OpenGL near/far mechanism, which divides the whole space into two parts:

```
osg::ArgumentParser arguments(&argc,argv);

double zNear = 1.0, zMid = 1e4, zFar = 2e8;
arguments.read( "--depth-partition", zNear, zMid, zFar );
```

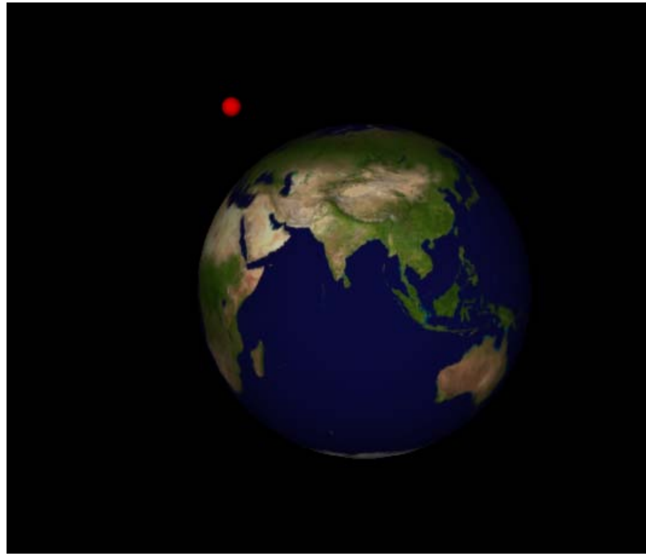
8. The near/middle/far values are set to the `osgViewer::DepthPartitionSettings` object:

```
osg::ref_ptr<osgViewer::DepthPartitionSettings> dps =
  new osgViewer::DepthPartitionSettings;
// Use fixed numbers as the partition values.
dps->_mode = osgViewer::DepthPartitionSettings::FIXED_RANGE;
dps->_zNear = zNear;
dps->_zMid = zMid;
dps->_zFar = zFar;
```

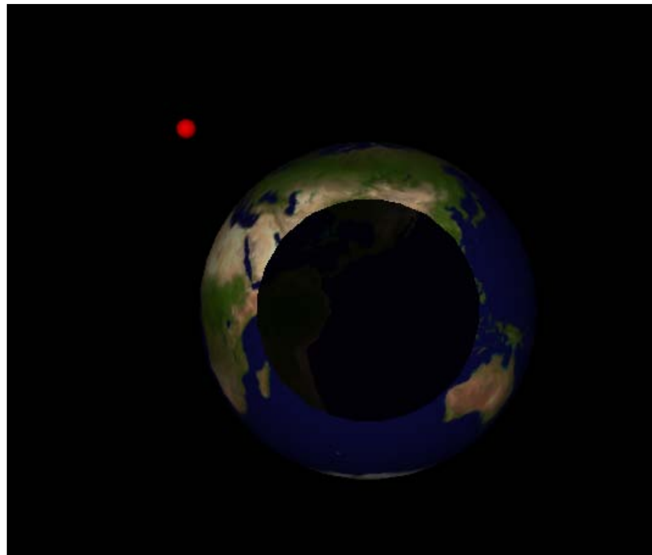
9. Apply the `depth partition` settings to the viewer. We have to preset a home position of the camera manipulator here, which helps us look at the earth at the beginning; otherwise, we will have to look for it in the wide universe ourselves:

```
osgViewer::Viewer viewer;
viewer.getCamera()->setClearColor( osg::Vec4(0.0f, 0.0f, 0.0f,
  1.0f) );
viewer.setSceneData( createScene() );
viewer.setUpDepthPartition(dps.get());
viewer.setCameraManipulator( new osgGA::TrackballManipulator );
viewer.getCameraManipulator()->setHomePosition(
  osg::Vec3d(0.0,-12.5*radius_earth,0.0), osg::Vec3d(),
    osg::Vec3d(0.0,0.0,1.0) );
return viewer.run();
```

10. You will see the earth in front of you. If you rotate the camera slightly, you can see the sun which is only a small red point, as shown in the following screenshot. Everything looks normal here:



11. Now remove the line calling `viewer.setUpDepthPartition()` method and return the recipe. What you have found this time? The earth seems to have been eaten by someone! If you zoom in the view, you will surprisingly find the earth has disappeared. What just happened when we stopped using the **depth partition** functionality?



How it works...

OSG enables automatic near/far planes computation by default. The basic idea is: calculate the Z value of each scene object in eye coordinate frame, and thus get the maximum Z value, which can be approximately treated as the farthest depth value. After that, multiply the far plane value with a very small ratio (must be less than 1.0) and get the near plane value:

```
zNear = zFar * nearFarRatio
```

And then apply the results to the camera's projection matrix.

The process can't be flipped, that is, it is difficult to get the minimum Z value first, because there are always objects with negative Z values (behind or parallel with the viewer), and we can hardly decide the real near plane due to these confusing values.

Now we can explain why the earth disappeared when disabling use of the `setUpDepthPartition()` method. As the far plane is too far away from the viewer, the calculated near plane is too far and the resultant frustum doesn't contain the earth node.

There is more than one solution for this. Besides the **depth partition** (which will render the scene for multiple times and may, therefore, lose efficiency), we could also provide an even smaller ratio to get a smaller near value. This is done by calling the `setNearFarRatio()` method:

```
camera->setNearFarRatio( 0.0001 );
```

By default, the near/far ratio is 0.0005. Be careful of the precision of your depth buffer when you alter it because OpenGL depth buffer always handles 16-bit or 24-bit float values.

There's more...

You may read more about the depth buffer problem, or Z buffer problem at:

`http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html`.

Implementing the radar map

Traditional radar uses electromagnetic pulses to detect the positions and motions of an object. It maps the electromagnetic parameters onto a two-dimensional plane to form a radar map, which indicates all vehicles and aircrafts appearing in a certain place.

In this recipe, we will simulate a very simple radar map by mapping all static and animated models in the main scene onto an HUD camera, and use colored marks to label them.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Include necessary headers.

```
#include <osg/Material>
#include <osg/ShapeDrawable>
#include <osg/Camera>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. We will define two mask constants and a macro providing random numbers for later use.

```
const unsigned int MAIN_CAMERA_MASK = 0x1;
const unsigned int RADAR_CAMERA_MASK = 0x2;
#define RAND(min, max) ((min) + (float)rand()/(RAND_MAX+1) *
((max)-(min)))
```

3. First we can have a function for creating all kinds of scene objects. To make them visible in both the main view and the radar map, we have to do something special besides reading the model file with `osgDB::readNodeFile()` method.

```
osg::Node* createObject( const std::string& filename, const
osg::Vec4& color )
{
    ...
}
```

4. In the function, read the model from file and set up a node mask indicating it to be rendered in the main camera.

```
float size = 5.0f;
osg::ref_ptr<osg::Node> model_node =
  osgDB::readNodeFile(filename);
if ( model_node.valid() ) model_node->setNodeMask(

  MAIN_CAMERA_MASK );
```

5. Create a marker to replace the model itself in the radar map.

```
osg::ref_ptr<osg::ShapeDrawable> mark_shape =
  new osg::ShapeDrawable;
mark_shape->setShape( new osg::Box(osg::Vec3(), size) );

osg::ref_ptr<osg::Geode> mark_node = new osg::Geode;
mark_node->addDrawable( mark_shape.get() );
mark_node->setNodeMask( RADAR_CAMERA_MASK );
```

6. Now add both the model and its marker to a group node and return it as a complete scene object. Apply a material here to paint the object in a specified color:

```
osg::ref_ptr<osg::Group> obj_node = new osg::Group;
obj_node->addChild( model_node.get() );
obj_node->addChild( mark_node.get() );

osg::ref_ptr<osg::Material> material = new osg::Material;
material->setColorMode( osg::Material::AMBIENT );
material->setAmbient( osg::Material::FRONT_AND_BACK,
  osg::Vec4(0.8f, 0.8f, 0.8f, 1.0f) );
material->setDiffuse( osg::Material::FRONT_AND_BACK,
  color*0.8f );
material->setSpecular( osg::Material::FRONT_AND_BACK, color );
material->setShininess( osg::Material::FRONT_AND_BACK, 1.0f );
obj_node->getOrCreateStateSet()->setAttributeAndModes(
  material.get(),
    osg::StateAttribute::ON|osg::StateAttribute::OVERRIDE );
return obj_node.release();
```

7. The `createStaticNode()` function is convenient for creating objects at a fixed position:

```
osg::MatrixTransform* createStaticNode( const osg::Vec3&
  center, osg::Node* child )
{
  osg::ref_ptr<osg::MatrixTransform> trans_node = new
    osg::MatrixTransform;
  trans_node->setMatrix( osg::Matrix::translate(center) );
  trans_node->addChild( child );
  return trans_node.release();
}
```

8. The `createAnimateNode()` is easy-to-use for creating objects moving on a specified path:

```
osg::MatrixTransform* createAnimateNode( const osg::Vec3&
  center, float radius, float time, osg::Node* child )
{
  osg::ref_ptr<osg::MatrixTransform> anim_node = new
    osg::MatrixTransform;
  anim_node->addUpdateCallback(
    osgCookBook::createAnimationPathCallback(radius, time) );
  anim_node->addChild( child );

  osg::ref_ptr<osg::MatrixTransform> trans_node = new
    osg::MatrixTransform;
```

```
  trans_node->setMatrix( osg::Matrix::translate(center) );
  trans_node->addChild( anim_node.get() );
  return trans_node.release();
}
```

9. In the main entry, we will first load and create some composite objects, each including the origin model and a marker box:

```
osg::Node* obj1 = createObject( "dumptruck.osg",
  osg::Vec4(1.0f, 0.2f, 0.2f, 1.0f) );
osg::Node* obj2 = createObject( "dumptruck.osg.0,0,180.rot",
  osg::Vec4(0.2f, 0.2f, 1.0f, 1.0f) );
osg::Node* air_obj2 = createObject( "cessna.osg.0,0,90.rot",

  osg::Vec4(0.2f, 0.2f, 1.0f, 1.0f) );
```

10. Now we randomly place some static and animating objects with the XY range from [-100, -100] to [100, 100]. This can be treated as the region shown in the radar map:

```
osg::ref_ptr<osg::Group> scene = new osg::Group;
for ( unsigned int i=0; i<10; ++i )
{
  osg::Vec3 center1( RAND(-100, 100), RAND(-100, 100), 0.0f );
  scene->addChild( createStaticNode(center1, obj1) );

  osg::Vec3 center2( RAND(-100, 100), RAND(-100, 100), 0.0f );
  scene->addChild( createStaticNode(center2, obj2) );
}
for ( unsigned int i=0; i<5; ++i )
{
  osg::Vec3 center( RAND(-50, 50), RAND(-50, 50),
    RAND(10, 100) );
  scene->addChild( createAnimateNode(center, RAND(10.0, 50.0),
    5.0f, air_obj2) );
}
```

11. Create an HUD camera for representing the radar map screen:

```
osg::ref_ptr<osg::Camera> radar = new osg::Camera;
radar->setClearColor( osg::Vec4(0.0f, 0.2f, 0.0f, 1.0f) );
radar->setRenderOrder( osg::Camera::POST_RENDER );
radar->setAllowEventFocus( false );
radar->setClearMask( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
  );
radar->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
radar->setViewport( 0.0, 0.0, 200.0, 200.0 );
```

12. Set up the view matrix, projection matrix, and culling mask of the HUD camera. The mask set here is the same as the one we set for the marker node (so the AND operation between them will result in true). That means only markers can be seen in the radar camera. The view and projection matrices here are used to describe the view direction and range of the radar map:

```
radar->setViewMatrix( osg::Matrixd::lookAt(osg::Vec3(0.0f,
  0.0f, 120.0f), osg::Vec3(), osg::Y_AXIS) );
radar->setProjectionMatrix( osg::Matrixd::ortho2D(-120.0,
  120.0, -120.0, 120.0) );
radar->setCullMask( RADAR_CAMERA_MASK );
radar->addChild( scene.get() );
```

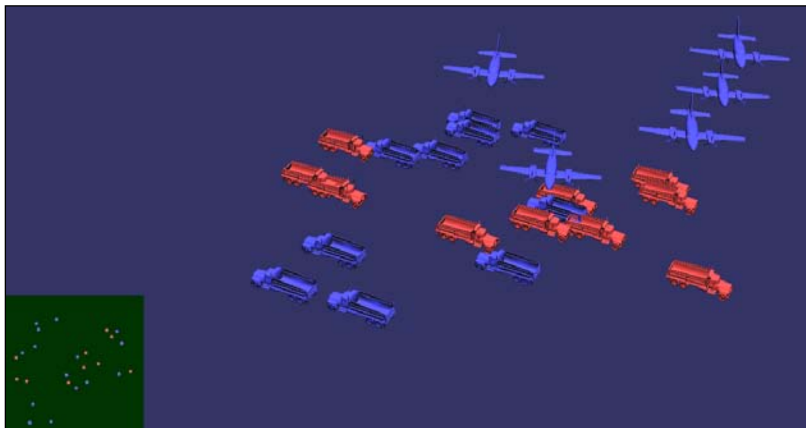
13. Add the radar and the main scene to the root node. To note, scene is already added as the child of the radar camera, so it will be always traversed twice after being added as root's child here:

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( radar.get() );
root->addChild( scene.get() );
```

14. The `MAIN_CAMERA_MASK` constant set to the main camera makes it only render model nodes created before. The lighting mode set here improves the light computation on model surfaces in this recipe.

```
osgViewer::Viewer viewer;
viewer.getCamera()->setCullMask( MAIN_CAMERA_MASK );
viewer.setSceneData( root.get() );
viewer.setLightingMode( osg::View::SKY_LIGHT );
return viewer.run();
```

15. Start the program and you will see a number of trucks and Cessnas in the main view, and only markers in the radar map view, as shown in the following screenshot. That is exactly what we want at the beginning:



How it works...

The cull mask indicates that some types of nodes can be rendered in the camera (the AND operation between node mask and cull mask results in a non-zero value), and some types can't (the AND operation results in zero). It will be checked before other scene culling operations such as **view-frustum culling** and **small feature culling**.

There are two extra methods called `setCullMaskLeft()` and `setCullMaskRight()` besides the `setCullMask()` method. They are mainly used for the stereo-displaying situations, and can determine what kinds of nodes will be shown in the left eye, and what to show in the right eye.

The `setLightingMode()` method controls the global lighting mechanism in OSG. By default the enumeration value is `HEADLIGHT`, which means the light is positioned near the eye and shines along the line of sight. You may change it to `NO_LIGHT` (no global light) or `SKY_LIGHT` (the light is fixed at a position in the world) if required. You may also use the `setLight()` method of the viewer to specify a user light object for global illumination.

Showing the top, front, and side views of a model

Open one of your favorite 3D scene editors, such as 3DS Max, Maya, Blender 3D, and so on. Most of them should default to four views: the top view, side view, front view, and a fourth perspective view. You may change the first three to display the scene from the top, left, right, front, back, or bottom side, which brings great convenience for editors to watch and alter 3D models.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Include necessary headers:

   ```
   #include <osg/Camera>
   #include <osgDB/ReadFile>
   #include <osgGA/TrackballManipulator>
   #include <osgViewer/CompositeViewer>
   ```

2. The `create2DView()` function will be used for creating top, left, and side views of the scene. Set up the view to look at a specific direction but keep it focused on the scene node, and also configure the perspective projection matrix according to the window size:

   ```
   osgViewer::View* create2DView( int x, int y, int width, int
     height, const osg::Vec3& lookDir, const osg::Vec3& up,
   ```

```
    osg::GraphicsContext* gc, osg::Node* scene )
{
  osg::ref_ptr<osgViewer::View> view = new osgViewer::View;
  view->getCamera()->setGraphicsContext( gc );
  view->getCamera()->setViewport( x, y, width, height );
  view->setSceneData( scene );

  osg::Vec3 center = scene->getBound().center();
  double radius = scene->getBound().radius();
  view->getCamera()->setViewMatrixAsLookAt( center -
    lookDir*(radius*3.0), center, up );
  view->getCamera()->setProjectionMatrixAsPerspective(
    30.0f, static_cast<double>(width)/static_cast<double>
      (height), 1.0f, 10000.0f );
    return view.release();
}
```

3. In the main entry, we will always read a renderable scene first. This time, it is a Cessna:

```
osg::ArgumentParser arguments( &argc, argv );

osg::ref_ptr<osg::Node> scene = osgDB::readNodeFiles
  ( arguments );
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

4. Read the screen size and create a new graphics context according to its traits. The same work is done in the first example of this chapter, so you should already be familiar with this process.

```
unsigned int width = 800, height = 600;
osg::GraphicsContext::WindowingSystemInterface* wsi =
  osg::GraphicsContext::getWindowingSystemInterface();
if ( wsi ) wsi->getScreenResolution(
    osg::GraphicsContext::ScreenIdentifier(0), width, height );

osg::ref_ptr<osg::GraphicsContext::Traits> traits = new
  osg::GraphicsContext::Traits;
traits->x = 0;
traits->y = 0;
traits->width = width;
traits->height = height;
traits->windowDecoration = false;
traits->doubleBuffer = true;
traits->sharedContext = 0;
```

```
osg::ref_ptr<osg::GraphicsContext> gc =
  osg::GraphicsContext::createGraphicsContext( traits.get() );
if ( !gc || !scene ) return 1;
```

5. Now, we are going to make the four views share the same graphics context, that is, the same window. Three of them are 2D views (top, left, and front), and they are placed at different X and Y coordinates displaying the same scene:

```
int w_2 = width/2, h_2 = height/2;
osg::ref_ptr<osgViewer::View> top = create2DView(
    0, h_2, w_2, h_2,-osg::Z_AXIS, osg::Y_AXIS, gc.get(), scene.
get());
osg::ref_ptr<osgViewer::View> front = create2DView(
    w_2, h_2, w_2, h_2, osg::Y_AXIS, osg::Z_AXIS, gc.get(), scene.
get());
osg::ref_ptr<osgViewer::View> left = create2DView(
    0, 0, w_2, h_2, osg::X_AXIS, osg::Z_AXIS, gc.get(), scene.
get());
```

6. The main view does not use fixed view matrix, but applies a default manipulator for users to navigate the camera freely:

```
osg::ref_ptr<osgViewer::View> mainView = new osgViewer::View;
mainView->getCamera()->setGraphicsContext( gc.get() );
mainView->getCamera()->setViewport( w_2, 0, w_2, h_2 );
mainView->setSceneData( scene.get() );
mainView->setCameraManipulator( new osgGA::TrackballManipulator );
```

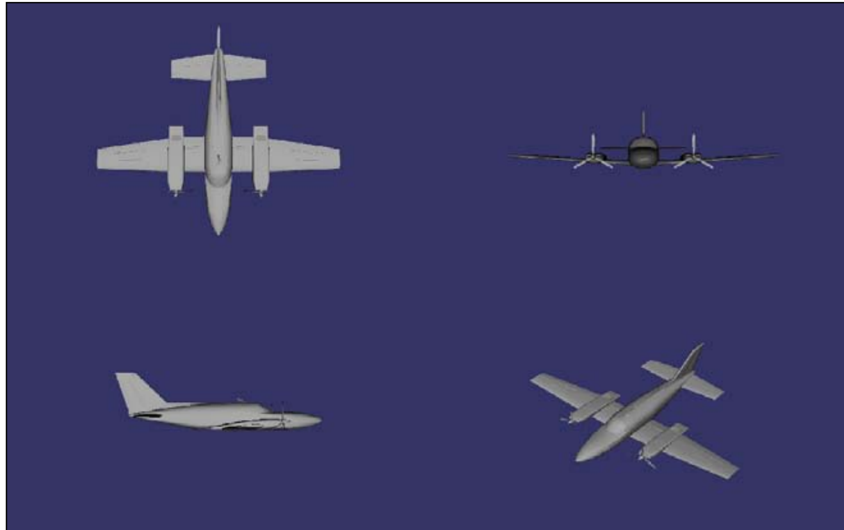
7. Add the four views to the composite viewer:

```
osgViewer::CompositeViewer viewer;
viewer.addView( top.get() );
viewer.addView( front.get() );
viewer.addView( left.get() );
viewer.addView( mainView.get() );
```

8. Start the simulation. Here, we don't use the `viewer.run()` method but instead write a simple loop that calls the `frame()` method all the time. That's because the `run()` method will automatically apply a trackball manipulator to every view unless it already has a valid one. As we don't want the 2D views to be controlled improperly by manipulators, we have to avoid using the `run()` method in this case:

```
while ( !viewer.done() )
{
    viewer.frame();
}
return 0;
```

9. Ok, now we will have a four-view interface a little similar to some of the famous 3D software. The main view can still be rotated and scaled by dragging the mouse, but the 2D ones are fixed and can't be moved, which is certainly inconvenient. In the next recipe, we will continue to work on this issue.



How it works...

In the `create2DView()` function, we need the `lookDir` and `up` vectors to define the view matrix, and the `gc` variable to specify the graphics context attached with the camera.

We may encounter a bifurcation while creating 2D views, that is, there are two ways to set up the projection matrix: to use the perspective projection or the orthographic one. In order to represent a true 2D view of the world, the orthographic projection is preferred. However, in this and the next recipe, we will use perspective projection matrices to make user interactions easier. It's up to you to change the code listed here and make it work as you wish.

There's more...

Now, it's time for us to summarize the relationships of views, cameras, and graphics contexts.

An OSG scene viewer can have only one view (`osgViewer::Viewer`) or multiple views (`osgViewer::CompositeViewer`). Every view has its own scene graph, camera manipulator, and event handlers. A view can have one main camera, which is controlled by the manipulator, and some slave cameras with customized offsets to the main one (see the second recipe in this chapter). Cameras can also be added as a normal node into the scene graph (see the fourth recipe). It will multiply or reset current view and projection matrices, and use current context or choose a different one to render the sub-graph.

The camera must be attached with a graphics context to create and enable the OpenGL rendering window. If different screen numbers are specified while creating the contexts, we may even enable the cameras to work in a multi-monitor environment (see the first recipe). Multiple cameras can share one graphics context and use `setViewport()` and `setRenderOrder()` to set the rendering range and orders.

Thus, to design an application with multiple windows and multiple scenes, we can either make use of the composite viewer or add more cameras in a single viewer framework. The former provides event handlers and manipulators for each view to update user data. But the latter could also implement the similar work with node callbacks. At last, it's up to you to choose a view/camera framework to create multi-windows applications.

Manipulating the top, front, and side views

Let us continue work on the last recipe. This time we are going to handle some interactive events on the 2D views. In order to achieve this, we can just add an event handler to each view and customize the behaviors while the user drags the mouse in the viewport. Code segments that are completely unchanged will not be listed here again.

How to do it...

Carry out the following steps in order to complete this recipe:

1. The most important addition to the previous recipe is the `AuxiliaryViewUpdater` class. It allows users to pan the 2D view with the left button, and zoom in/out with the right button. The `_distance` variable defines the zoom factor. The `_offsetX` and `_offsetY` indicate the offset while panning. And `_lastDragX` and `_lastDragY` are used for recording the mouse coordinates only during the mouse dragging action (otherwise they are set to -1).

```
class AuxiliaryViewUpdater : public osgGA::GUIEventHandler
{
  public:
  AuxiliaryViewUpdater()
  :   _distance(-1.0), _offsetX(0.0f), _offsetY(0.0f),
        _lastDragX(-1.0f), _lastDragY(-1.0f)
  {}

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

protected:
  double _distance;
  float _offsetX, _offsetY;
  float _lastDragX, _lastDragY;
};
```

2. In the `handle()` method, we will obtain the view object and decide operations according to the event type:

```
osgViewer::View* view = static_cast<osgViewer::View*>(&aa);
if ( view )
{
  switch ( ea.getEventType() )
  {
    ...
  }
}
return false;
```

3. In the mouse dragging event, we change the panning offset or the zoom distance by computing the delta values of current and last mouse positions. And, in the mouse pushing event, we will reset the dragging values for next time use:

```
case osgGA::GUIEventAdapter::PUSH:
  _lastDragX = -1.0f;
  _lastDragY = -1.0f;
  break;
case osgGA::GUIEventAdapter::DRAG:
  if ( _lastDragX>0.0f && _lastDragY>0.0f )
  {
    if ( ea.getButtonMask()==osgGA::GUIEventAdapter::LEFT_MOUSE_
BUTTON )
    {
      _offsetX += ea.getX() - _lastDragX;
      _offsetY += ea.getY() - _lastDragY;
    }
    else if ( ea.getButtonMask()==
      osgGA::GUIEventAdapter::RIGHT_MOUSE_BUTTON )
    {
      float dy = ea.getY() - _lastDragY;
      _distance *= 1.0 + dy / ea.getWindowHeight();
      if ( _distance<1.0 ) _distance = 1.0;
    }
  }
  _lastDragX = ea.getX();
  _lastDragY = ea.getY();
  break;
```

4. In the frame event, which is executed in every frame, we will apply the member variables to the view camera. The algorithm will be explained in the next section of the recipe.

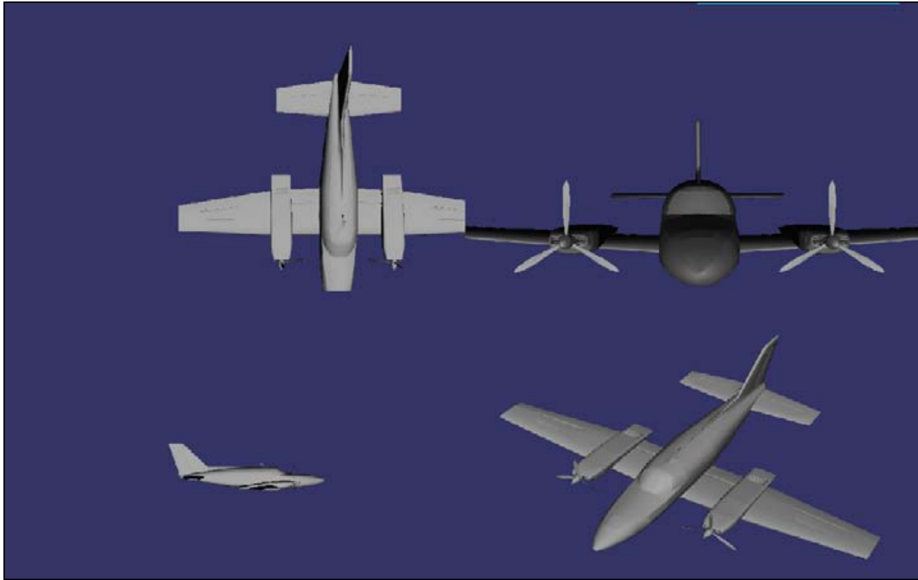
```
case osgGA::GUIEventAdapter::FRAME:
  if ( view->getCamera() )
  {
    osg::Vec3d eye, center, up;
    view->getCamera()->getViewMatrixAsLookAt( eye, center,
      up );

    osg::Vec3d lookDir = center - eye; lookDir.normalize();
    osg::Vec3d side = lookDir ^ up; side.normalize();

    const osg::BoundingSphere& bs = view->getSceneData()->
      getBound();
    if ( _distance<0.0 ) _distance = bs.radius() * 3.0;
      center = bs.center();

      center -= (side * _offsetX + up * _offsetY) * 0.1;
      view->getCamera()->setViewMatrixAsLookAt( center-
        lookDir*_distance, center, up );
  }
  break;
```

5. The `create2DView()` function does not have to be changed compared to the last example. Just leave it as it was earlier.

6. The only change in the main function is to allocate and add the new `AuxiliaryViewUpdater` class to each 2D view. This must be done before the simulation loop begins:

```
top->addEventHandler( new AuxiliaryViewUpdater );
front->addEventHandler( new AuxiliaryViewUpdater );
left->addEventHandler( new AuxiliaryViewUpdater );
```

7. Now, try to drag in the 2D views with your left or right mouse button and see what happens, as shown in the following screenshot. It is now possible to adjust and observe the scene in all-around views, as if you are working with other 3D software like Autodesk 3DS Max and Maya.



How it works...

The computation of new view matrix in the `AuxiliaryViewUpdater` class is not hard to realize. First, we get a current look at the parameters with the `getViewMatrixAsLookAt()` method. Then we can easily obtain the `look direction` vector (`center - eye`) and the side vector (cross product of the look direction and `up` vector). The latter is actually the X axis in the eye coordinates, and the `up` vector is the Y axis.

Panning the scene in a 2D view means to change the X and Y values of the scene in the viewer's eye. Thus we can just use `_offsetX` and `_offsetY` to change the view point along the `lookDir` and `up` directions, and use `_distance` to control the distance between the eye and the view center. That's what we have done in the `handle()` method's implementation.

Following a moving model

It is common in games and simulation programs that you are orbiting around a moving vehicle, and keep focus on the vehicle's center or a specific point. The view center may not move away from the tracking point, no matter the vehicle is animating or not. Except that, the rotation and scale operations are available. This results in the viewer's camera following the vehicle and even working in a first person's perspective.

How to do it...

Carry out the following steps in order to complete this recipe:

1. Include necessary headers:

```
#include <osg/Camera>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgGA/OrbitManipulator>
#include <osgViewer/Viewer>
```

2. In this recipe, we implement a node following functionality, that is, the `FollowUpdater` class, which is derived from the `osgGA::GUIEventHandler`. It requires the target node pointer to be passed in the constructor:

```
class FollowUpdater : public osgGA::GUIEventHandler
{
public:
    FollowUpdater( osg::Node* node ) : _target(node) {}

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

    osg::Matrix computeTargetToWorldMatrix( osg::Node* node )
const;

protected:
    osg::observer_ptr<osg::Node> _target;
};
```

3. In the `handle()` function, when encountering the `FRAME` event, which is called in every frame, we will try to retrieve the `osgGA::OrbitManipulator` object, (which is the super class of the default trackball manipulator) and place its center at the central point of the target node in world coordinates.

```
osgViewer::View* view = static_cast<osgViewer::View*>(&aa);
if ( !view || !_target ||
  ea.getEventType()!=osgGA::GUIEventAdapter::FRAME ) return false;

osgGA::OrbitManipulator* orbit =
  dynamic_cast<osgGA::OrbitManipulator*>( view->
  getCameraManipulator() );
if ( orbit )
{
  osg::Matrix matrix = computeTargetToWorldMatrix(
    _target.get() );
```

```
  osg::Vec3d targetCenter = _target->getBound().center() *
    matrix;
  orbit->setCenter( targetCenter );
}
return false;
```

4. You should already get to be familiar with the method of computing local to world matrix in `computeTargetToWorldMatrix()`, which was introduced in *Chapter 2, Designing the Scene Graph*.

```
osg::Matrix computeTargetToWorldMatrix( osg::Node* node ) const
{
  osg::Matrix l2w;
  if ( node && node->getNumParents()>0 )
  {
    osg::Group* parent = node->getParent(0);
    l2w = osg::computeLocalToWorld( parent->
      getParentalNodePaths()[0] );
  }
return l2w;
}
```

5. In the main entry, we load a moving Cessna and a terrain to form the scene. The Cessna, which is cycling in the sky and worth following, will be specified as the target node of the following updater class:

```
osg::Node* model =
  osgDB::readNodeFile("cessna.osg.0,0,90.rot");
if ( !model ) return 1;

osg::ref_ptr<osg::MatrixTransform> trans = new
  osg::MatrixTransform;
trans->addUpdateCallback( osgCookBook::createAnimationPathCallback
(100.0f, 20.0) );
trans->addChild( model );

osg::ref_ptr<osg::MatrixTransform> terrain = new
  osg::MatrixTransform;
terrain->addChild( osgDB::readNodeFile("lz.osg") );
terrain->setMatrix( osg::Matrix::translate(0.0f, 0.0f,-200.0f) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( trans.get() );
root->addChild( terrain.get() );
```

6. Add the `FollowUpdater` instance to the viewer and start the simulation:

```
osgViewer::Viewer viewer;
viewer.addEventHandler( new FollowUpdater(model) );
viewer.setSceneData( root.get() );
return viewer.run();
```

7. Now, you will find that the main camera is always focusing on the target Cessna model, no matter where it is and what the orientation is. You can still rotate around the model with the left mouse button, and zoom in/out with the mouse wheel or right button. However, the middle button, which can pan the camera by default, doesn't work anymore, which means the Cessna model will always be rendered at the center of the screen. This is a so called node follower in this recipe:

How it works...

Here, we use the event handler to get the node's world center and set up the main camera. Certainly it can be replaced by node callbacks. Applying a node callback to the trackee and pass the main camera object as a parameter of the callback, so that we can implement the same functionality in the same way. Another solution is to use the camera manipulator, which is a more preferred option if you need mouse and keyboard interactions with the main camera. We will introduce it in the next recipe soon.

Using manipulators to follow models

Does OSG already provide us some functionality that implements the node tracking features? The answer is affirmative. The `osgGA::NodeTrackerManipulator` class enables us to follow a static or moving node, which will be demonstrated in this recipe. OSG also provides the `osgGA::FirstPersonManipulator` class for walking, driving, and flight manipulations. You may explore them yourself after reading this chapter.

How to do it...

Let us start.

1. Include necessary headers.

```
#include <osg/Camera>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgGA/KeySwitchMatrixManipulator>
#include <osgGA/TrackballManipulator>
#include <osgGA/NodeTrackerManipulator>
#include <osgViewer/Viewer>
```

2. Load the animating Cessna and the sample terrain. This is exactly the same as the previous example:

```
osg::Node* model =
  osgDB::readNodeFile("cessna.osg.0,0,90.rot");
if ( !model ) return 1;

osg::ref_ptr<osg::MatrixTransform> trans = new
  osg::MatrixTransform;
trans->addUpdateCallback(
  osgCookBook::createAnimationPathCallback(100.0f, 20.0) );
trans->addChild( model );

osg::ref_ptr<osg::MatrixTransform> terrain = new
  osg::MatrixTransform;
terrain->addChild( osgDB::readNodeFile("lz.osg") );
terrain->setMatrix( osg::Matrix::translate(0.0f, 0.0f,-200.0f)
  );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( trans.get() );
root->addChild( terrain.get() );
```

3. Create the node tracker manipulator and set the Cessna as the target. The home position is set here to make sure that the camera is initially not too far from the cessna node:

```
osg::ref_ptr<osgGA::NodeTrackerManipulator> nodeTracker = new
  osgGA::NodeTrackerManipulator;
nodeTracker->setHomePosition( osg::Vec3(0, -10.0, 0),
  osg::Vec3(), osg::Z_AXIS );
nodeTracker->setTrackerMode( osgGA::NodeTrackerManipulator::NODE_
CENTER_AND_ROTATION );
nodeTracker->setRotationMode(
  osgGA::NodeTrackerManipulator::TRACKBALL );
nodeTracker->setTrackNode( model );
```

4. We also create a switcher manipulator to switch between the classic trackball manipulator and the tracker. The first parameter of the `addMatrixManipulator()` `method` indicates the key, which can be pressed to change to corresponding sub-manipulator:

```
osg::ref_ptr<osgGA::KeySwitchMatrixManipulator> keySwitch = new
  osgGA::KeySwitchMatrixManipulator;
keySwitch->addMatrixManipulator( '1', "Trackball", new
  osgGA::TrackballManipulator );
keySwitch->addMatrixManipulator( '2', "NodeTracker",

  nodeTracker.get() );
```

5. It's enough to start the viewer now:

```
osgViewer::Viewer viewer;
viewer.setCameraManipulator( keySwitch.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

6. You may view the scene normally first, and press the number *2* key to change to node tracking mode. It fixes the camera just behind the moving Cessna and prevents you from panning the camera. Press *1* to change back to trackball mode at any time you like.

How it works...

OSG has a complete camera manipulator interface, which is defined as the `osgGA::CameraManipulator` abstract class. It can only work on the main camera of a view and it will alter the view matrix for every frame to implement camera animations and navigations by mouse and keyboard. As we are going to adjust the main camera to track the moving node, the manipulator class will be a good choice. And we can change the tracking target and parameters on the fly by storing and obtaining the manipulator pointer.

You will find it easier to use the in-built node tracker manipulator to follow the nodes here. You can also use the `setHomePosition()` method for the manipulator to have a suitable start position. However, if you need a picture-in-picture effect or want to track the node in an auxiliary camera, callbacks should be more suitable because they can provide more flexible and less methods to override.

In fact, camera manipulators are also derived from the `osgGA::GUIEventHandler` class. However, they are treated separately from common event handlers, and do have extra interface for scene navigation purposes.

There's more...

In order to help you create your own manipulators in the future, we are going to introduce more about camera manipulators, as well as provide a simple manipulator example later in the chapter.

Designing a 2D camera manipulator

In the last few recipes, we have introduced how to control the main camera with event handlers and preset camera manipulators. The manipulator defines an interface, as well as some default functionalities, to control the main cameras of OSG views in response to user events.

The `osgGA::CameraManipulator` class is an abstract base class for all manipulators. Its subclasses include `osgGA::TrackballManipulator`, `osgGA::NodeTrackerManipulator`, `osgGA::KeySwitchMatrixManipulator`, and so on. In this recipe, it is time for us to create a manipulator of our own. In order to make things easier, we will aim at designing a two-dimension manipulator, which can only view, pan, and scale (but not rotate) the scene as if it is projected onto the XOY plane.

How to do it...

Let us start.

1. Include necessary headers.

```
#include <osgDB/ReadFile>
#include <osgGA/KeySwitchMatrixManipulator>
#include <osgGA/TrackballManipulator>
#include <osgViewer/Viewer>
```

2. The `osgGA::StandardManipulator` class is a good start for designing our own manipulators. It handles user events like mouse clicking and key pressing, and sends the event content to different virtual methods according to the event type. There are also virtual methods to be called during the traversal for delivering data. Therefore, the most important work for creating a new manipulator is to derive this class and override the necessary methods.

```
class TwoDimManipulator : public osgGA::StandardManipulator
{
public:
    TwoDimManipulator() : _distance(1.0) {}

    virtual osg::Matrixd getMatrix() const;
    virtual osg::Matrixd getInverseMatrix() const;
    virtual void setByMatrix( const osg::Matrixd& matrix );
    virtual void setByInverseMatrix( const osg::Matrixd& matrix );

    // Leave empty as we don't need these here. They are used by
other functions and classes to set up the manipulator directly.
    virtual void setTransformation( const osg::Vec3d&, const
osg::Quat& ) {}
```

```
    virtual void setTransformation( const osg::Vec3d&, const
osg::Vec3d&, const osg::Vec3d& ) {}
    virtual void getTransformation( osg::Vec3d&, osg::Quat& )
const {}
    virtual void getTransformation( osg::Vec3d&, osg::Vec3d&,
osg::Vec3d& ) const {}

    virtual void home( double );
    virtual void home( const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& us );

protected:
    virtual ~TwoDimManipulator() {}

    virtual bool performMovementLeftMouseButton(
        const double eventTimeDelta, const double dx, const double
dy );
    virtual bool performMovementRightMouseButton(
        const double eventTimeDelta, const double dx, const double
dy );

    osg::Vec3 _center;
    double _distance;
};
```

3. The `getMatrix()` method means to get the current position and the attitude matrix of this manipulator. The `getInverseMatrix()` method should get the matrix of the camera manipulator and inverse it. The inverted matrix is typically treated as the view matrix of the camera. These two methods are the most important when implementing a user manipulator, as they are the only interfaces for the system to retrieve and apply the view matrix. We will explain their implementations later in the *How it works...* section.

```
osg::Matrixd TwoDimManipulator::getMatrix() const
{
  osg::Matrixd matrix;
  matrix.makeTranslate( 0.0f, 0.0f, _distance );
  matrix.postMultTranslate( _center );
  return matrix;
}

osg::Matrixd TwoDimManipulator::getInverseMatrix() const
{
  osg::Matrixd matrix;
  matrix.makeTranslate( 0.0f, 0.0f,-_distance );
  matrix.preMultTranslate( -_center );
  return matrix;
}
```

4. The `setByMatrix()` and `setByInverseMatrix()` methods can be called from user-level code to set up the position matrix of the manipulator, or set with the inverse matrix (view matrix). The `osgGA::KeySwitchMatrixManipulator` object also makes use of them when switching between two manipulators. In our implementations, the `_node` variable, which is a member of `osgGA::StandardManipulator` instance, is used to compute distance from our eyes to the view center. It is set internally to point to the scene graph root.

```
void TwoDimManipulator::setByMatrix( const osg::Matrixd& matrix )
{
    setByInverseMatrix( osg::Matrixd::inverse(matrix) );
}

void TwoDimManipulator::setByInverseMatrix( const osg::Matrixd&
matrix )
{
    osg::Vec3d eye, center, up;
    matrix.getLookAt( eye, center, up );

    _center = center; _center.z() = 0.0f;
    if ( _node.valid() )
        _distance = abs((_node->getBound().center() - eye).z());
    else
        _distance = abs((eye - center).length());
}
```

5. The `home()` method and its overloaded version is used to move the camera to its default position (home position). In most cases, the home position is computed automatically with all the scene objects in the view frustum and Z axis upwards. If you need to change the behavior, use the `setHomePosition()` method to specify its default eye, center, and up vectors for your convenience. However, in this recipe, the default home position is in fact ignored, because we directly compute suitable values in the `home()` method by ourselves.

```
void TwoDimManipulator::home( double )
{
    if ( _node.valid() )
    {
        _center = _node->getBound().center();
        _center.z() = 0.0f;
        _distance = 2.5 * _node->getBound().radius();
    }
    else
    {
        _center.set( osg::Vec3() );
        _distance = 1.0;
```

```
    }
}

void TwoDimManipulator::home( const osgGA::GUIEventAdapter& ea,
  osgGA::GUIActionAdapter& us )
{ home( ea.getTime() ); }
```

6. The `performMovementLeftMouseButton()` method will be invoked when the user is dragging the mouse with left button down. Pan the camera at this time!

```
bool TwoDimManipulator::performMovementLeftMouseButton(
  const double eventTimeDelta, const double dx, const double
    dy )
{
  _center.x() -= 100.0f * dx;
  _center.y() -= 100.0f * dy;
  return false;
}
```

7. The `performMovementRightMouseButton()` method will be invoked when the user is dragging with right button down. Perform zoom in/out actions now!

```
bool TwoDimManipulator::performMovementRightMouseButton(
  const double eventTimeDelta, const double dx, const double
    dy )
{
  _distance *= (1.0 + dy);
  if ( _distance<1.0 ) _distance = 1.0;
  return false;
}
```

8. In the main entry, we will read a sample terrain for testing the new 2D manipulator.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( osgDB::readNodeFile("lz.osg") );
```

9. Use key switch manipulator to switch between the default trackball manipulator and our work. Here, we also set a home position for the trackball manipulator to start at a good place.

```
osg::ref_ptr<osgGA::KeySwitchMatrixManipulator> keySwitch = new
  osgGA::KeySwitchMatrixManipulator;
keySwitch->addMatrixManipulator( '1', "Trackball", new
  osgGA::TrackballManipulator );
keySwitch->addMatrixManipulator( '2', "TwoDim", new
  TwoDimManipulator );

const osg::BoundingSphere& bs = root->getBound();
keySwitch->setHomePosition( bs.center()+osg::Vec3(0.0f, 0.0f,
  bs.radius()), bs.center(), osg::Y_AXIS );
```

10. Finally, start the viewer.

```
osgViewer::Viewer viewer;
viewer.setCameraManipulator( keySwitch.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

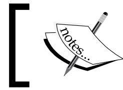
11. When the application starts, you will find the camera stays on the top of the terrain. Press *2* to change to our customized manipulator, and drag the mouse with left or right button down to browse the scene in 2D mode. Press *1* at any time to switch back to trackball mode.

How it works...

As we already know, OpenGL defines the eye position at the origin with the view direction along the negative Z axis by default. So if we want to emulate a 2D camera controller, we can simply move the position of the eye but keep the view direction unchanged. That is what we see in the `getMatrix()` method— the `_center variable` decides the X and Y coordinates of the eye, and `_distance` decides the Z value (using two variables makes it a little easier to handle changes in the `setByMatrix()` method).

Manipulating the view with joysticks

In the last example of this chapter, we would like to add joystick support to the previous 2D manipulator, which means to use joysticks to pan and scale the scene in view. OSG doesn't provide native joystick functionalities so it is time to reply on some external libraries. This time we will choose the famous **DirectInput** library that belongs to **DirectX 8** and higher versions. With some modifications to the last example's code, we can quickly add supports of **DirectInput** devices and make use of them in camera manipulators, callbacks, and event handlers.



Note that this recipe can only work under Windows systems.

A picture of the joystick used here is shown in the following screenshot:

Getting ready

If you don't have the **DirectX SDK** installed, find and download it from the Microsoft website:

http://msdn.microsoft.com/en-us/directx/

After that, configure **CMake** to find the library file `dinput8.lib` and headers for adding DirectInput support.

```
FIND_PATH(DIRECTINPUT_INCLUDE_DIR dinput.h)
FIND_LIBRARY(DIRECTINPUT_LIBRARY dinput7.lib dinput8.lib)
FIND_LIBRARY(DIRECTINPUT_GUID_LIBRARY dxguid.lib)

SET(EXTERNAL_INCLUDE_DIR "${DIRECTINPUT_INCLUDE_DIR}")
TARGET_LINK_LIBRARIES(${EXAMPLE_NAME}
                      ${DIRECTINPUT_LIBRARY}
                      ${DIRECTINPUT_GUID_LIBRARY}
)
```

How to do it...

1. We need some more headers and definitions before the `TwoDimManipulator` class declaration.

```
#define DIRECTINPUT_VERSION 0x0800
#include <windows.h>
#include <dinput.h>
#include <osgViewer/api/Win32/GraphicsWindowWin32>
```

2. Add two new virtual methods in the class.

```
class TwoDimManipulator : public osgGA::StandardManipulator
{
public:
  ...
  virtual void init( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& us );
  virtual bool handle( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& us );
  ...
};
```

3. Use global variables to save DirectInput device and joystick objects.

```
LPDIRECTINPUT8 g_inputDevice;
LPDIRECTINPUTDEVICE8 g_joystick;
```

4. The `EnumJoysticksCallback()` method will look for all usable joysticks and record the first valid one.

```
static BOOL CALLBACK EnumJoysticksCallback( const
  DIDEVICEINSTANCE* didInstance, VOID* )
{
  HRESULT hr;
  if ( g_inputDevice )
  {
    hr = g_inputDevice->CreateDevice( didInstance->
      guidInstance, &g_joystick, NULL );
  }
  if ( FAILED(hr) ) return DIENUM_CONTINUE;
  return DIENUM_STOP;
}
```

5. In the `TwoDimManipulator` constructor, we will create a new input device and try to load an existing joystick object.

```
TwoDimManipulator::TwoDimManipulator()
:   _distance(1.0)
{
  HRESULT hr = DirectInput8Create( GetModuleHandle(NULL),
    DIRECTINPUT_VERSION, IID_IDirectInput8,
      (VOID**)&g_inputDevice, NULL );
  if ( FAILED(hr) || !g_inputDevice ) return;

  hr = g_inputDevice->EnumDevices( DI8DEVCLASS_GAMECTRL,
    EnumJoysticksCallback, NULL, DIEDFL_ATTACHEDONLY );
}
```

6. In the destructor, we release the joystick and device objects.

```
TwoDimManipulator::~TwoDimManipulator()
{
  if ( g_joystick )
  {
    g_joystick->Unacquire();
    g_joystick->Release();
  }
  if ( g_inputDevice ) g_inputDevice->Release();
}
```

7. The `init()` method is called when the manipulator is initialized at the first frame. This is the right place to bind the joystick to OSG window handle and set up necessary attributes.

```
void TwoDimManipulator::init( const osgGA::GUIEventAdapter& ea,
  osgGA::GUIActionAdapter& us )
{
  const osgViewer::GraphicsWindowWin32* gw =
  dynamic_cast<const osgViewer::GraphicsWindowWin32*>(
    ea.getGraphicsContext() );
  if ( gw && g_joystick )
  {
    DIDATAFORMAT format = c_dfDIJoystick2;
    g_joystick->SetDataFormat( &format );
    g_joystick->SetCooperativeLevel( gw->getHWND(),
      DISCL_EXCLUSIVE|DISCL_FOREGROUND );
    g_joystick->Acquire();
  }
}
```

8. The `handle()` method has the same meaning as the one in the `osgGA::GUIEventHandler` class. Here we will try to acquire joystick states for every frame, and parse and perform actions according to the selected joystick directions and buttons.

> Note that OSG has `performMovementLeftMouseButton()` and `performMovementRightMouseButton()` methods for performing mouse movements, which can be called here directly to perform joystick events.

```
bool TwoDimManipulator::handle( const osgGA::GUIEventAdapter&
  ea, osgGA::GUIActionAdapter& us )
{
```

```
    if ( g_joystick &&
      ea.getEventType()==osgGA::GUIEventAdapter::FRAME )
    {
      HRESULT hr = g_joystick->Poll();
      if ( FAILED(hr) ) g_joystick->Acquire();

      DIJOYSTATE2 state;
      hr = g_joystick->GetDeviceState( sizeof(DIJOYSTATE2),
        &state );
      if ( FAILED(hr) ) return false;
      ...  // Please find details in the source code
    }
    return false;
  }
```

9. The main entry has no changes. Now rebuild and run the example. Change to the 2D manipulator. You may press button *1* of the joystick and simultaneously press the direction buttons to move the camera. You may also press button *2* and the directions to zoom in/out. Be careful that different joystick devices may not have exactly the same key code, so you may have to configure your own joystick buttons according to the actual situation.

How it works...

OSG by default uses platform-specific Windows APIs to handle user interactions, including mouse and keyboard events. For example, under Windows, OSG will internally create a message queue, convert messages into OSG events, and send them to the event handler. The converted events are so called `osgGA::GUIEventAdapter` objects. This works in many situations but does not support some advanced functionalities such as joystick and force feedback. The message mechanism may not work properly when users press multiple keys. DirectInput in contrast returns each key and button's state and lets the developers decide what to do at that time. It also provides complete functions for handling joysticks and gamepads. That is exactly why this recipe makes sense here, and it may also help in your future applications.

There's more...

DirectInput is a great dependency for applications and games requiring joystick support. However, it works only under Windows and your program will thus be platform-specific. Try looking for some other input libraries if you need to port to other platforms. **Object Oriented Input System** (**OIS**) may be a good choice. You may download the library source code at:

`http://sourceforge.net/projects/wgois/`

5

Animating Everything

In this chapter, we will cover:

- Opening and closing doors
- Playing a movie in the 3D world
- Designing scrolling text
- Implementing morph geometry
- Fading in and out
- Animating a flight on fire
- Dynamically lighting within shaders
- Creating a simple Galaxian game
- Building a skeleton system
- Skinning a customized mesh
- Letting the physics engine be

Introduction

Computer animation means to generate moving images and render them on the screen one after another. The animated images make the viewers think that they are seeing smoothly moving objects. There must be at least 12 frames drawn per second to trick the human brain. And over 60 frames per second will create a perfect animating process.

Typical animation types in OSG include path animation (changing position, rotation, and scale factor of an object), texture animation (dynamically updating textures), morph animation (blending shapes and making changes per vertex), state animation (changing rendering states), particle animation, and skeletal animation (representing characters). The **osgAnimation** library provides a complete framework for handling different kinds of animations. And the **osgParticle** library makes use of another flexible framework to design and render particles. We will introduce both in different recipes. You may read the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, for more information.

In this chapter, we will also integrate a famous physics engine into our OSG applications to improve the program's performance and support real-physics features of rigid bodies. And for game developers and lovers, we provide another simple but complete example that imitates the classic Galaxian.

Opening and closing doors

Opening and closing doors is a very common action in both daily life and computer games. You click with your mouse and slide the door open, and then you may meet either a girl or a monster behind it. Doors and entrances are also important in some spatial index algorithms such as the PVS (potentially visible set, refer to `http://en.wikipedia.org/wiki/Potentially_visible_set`). But in this recipe, we will only discuss how to animate the door object by applying actions to it.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/ShapeDrawable>
#include <osg/MatrixTransform>
#include <osgAnimation/BasicAnimationManager>
#include <osgAnimation/UpdateMatrixTransform>
#include <osgAnimation/StackedRotateAxisElement>
#include <osgViewer/Viewer>
#include <algorithm>
```

2. Create the wall geometry which clamps the door:

```
osg::Node* createWall()
{
  osg::ref_ptr<osg::ShapeDrawable> wallLeft =
    new osg::ShapeDrawable( new osg::Box(osg::Vec3(-5.5f,
      0.0f, 0.0f), 10.0f, 0.3f, 10.0f) );
```

```
  osg::ref_ptr<osg::ShapeDrawable> wallRight =
    new osg::ShapeDrawable( new osg::Box(osg::Vec3(10.5f,
      0.0f, 0.0f), 10.0f, 0.3f, 10.0f) );
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( wallLeft.get() );
  geode->addDrawable( wallRight.get() );
  return geode.release();
}
```

3. Create the door geometry with a slightly different color:

```
osg::MatrixTransform* createDoor()
{
  osg::ref_ptr<osg::ShapeDrawable> doorShape =
    new osg::ShapeDrawable( new osg::Box(osg::Vec3(2.5f,
      0.0f, 0.0f), 6.0f, 0.2f, 10.0f) );
  doorShape->setColor( osg::Vec4(1.0f, 1.0f, 0.8f, 1.0f) );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( doorShape.get() );

  osg::ref_ptr<osg::MatrixTransform> trans =
    new osg::MatrixTransform;
  trans->addChild( geode.get() );
  return trans.release();
}
```

4. The next function will compute the door opening or closing animation according to the closed parameter:

```
void generateDoorKeyframes( osgAnimation::FloatLinearChannel*
  ch, bool closed )
{
  osgAnimation::FloatKeyframeContainer* kfs =
    ch->getOrCreateSampler()->getOrCreateKeyframeContainer();
  kfs->clear();
  if ( closed )
  {
    kfs->push_back( osgAnimation::FloatKeyframe(0.0, 0.0f) );
    kfs->push_back( osgAnimation::FloatKeyframe(1.0, osg::PI_2) );
  }
  else
  {
    kfs->push_back( osgAnimation::FloatKeyframe(0.0, osg::PI_2) );
    kfs->push_back( osgAnimation::FloatKeyframe(1.0, 0.0f) );
  }
}
```

5. The `OpenDoorHandler` class receives user-click actions and checks if there is an intersection between the mouse coordinates and the door geometry. If so, it will generate opening/closing door animations and fill them into the keyframe container.

```
class OpenDoorHandler : public osgCookBook::PickHandler
{
public:
  OpenDoorHandler() : _closed(true) {}

  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::Intersection& result )
  {
    osg::NodePath::iterator itr = std::find(
      result.nodePath.begin(), result.nodePath.end(),
        _door.get() );
    if ( itr!=result.nodePath.end() )
    {
      if ( _manager->isPlaying(_animation.get()) )
        return;

      osgAnimation::FloatLinearChannel* ch =
        dynamic_cast<osgAnimation::FloatLinearChannel*>(
          _animation->getChannels().front().get() );
      if ( ch )
      {
        generateDoorKeyframes( ch, _closed );
        _closed = !_closed;
      }
      _manager->playAnimation( _animation.get() );
    }
  }

  osg::observer_ptr<osgAnimation::BasicAnimationManager>
    _manager;
  osg::observer_ptr<osgAnimation::Animation> _animation;
  osg::observer_ptr<osg::MatrixTransform> _door;
  bool _closed;
};
```

6. In the main entry, we will create an animation channel which handles the pivoting animation along one axis. Each of its keyframes requires only one value that indicates the rotation in radians. The door animation will not be repeated.

```
osg::ref_ptr<osgAnimation::FloatLinearChannel> ch =
  new osgAnimation::FloatLinearChannel;
ch->setName( "euler" );
```

```
ch->setTargetName( "DoorAnimCallback" );
generateDoorKeyframes( ch.get(), true );

osg::ref_ptr<osgAnimation::Animation> animation =
  new osgAnimation::Animation;
animation->setPlayMode( osgAnimation::Animation::ONCE );
animation->addChannel( ch.get() );
```

7. We have to also add an updater to the door node to make it recognize the animation channel we set just now. It has the same name as the channel's target name, and has a stacked rotating parameter which records the pivot axis and the initial value.

```
osg::ref_ptr<osgAnimation::UpdateMatrixTransform> updater =
  new osgAnimation::UpdateMatrixTransform(
    "DoorAnimCallback");
updater->getStackedTransforms().push_back(
  new osgAnimation::StackedRotateAxisElement(
    "euler", osg::Z_AXIS, 0.0) );
```

8. Add the animation to the manager.

```
osg::ref_ptr<osgAnimation::BasicAnimationManager> manager =
  new osgAnimation::BasicAnimationManager;
manager->registerAnimation( animation.get() );
```

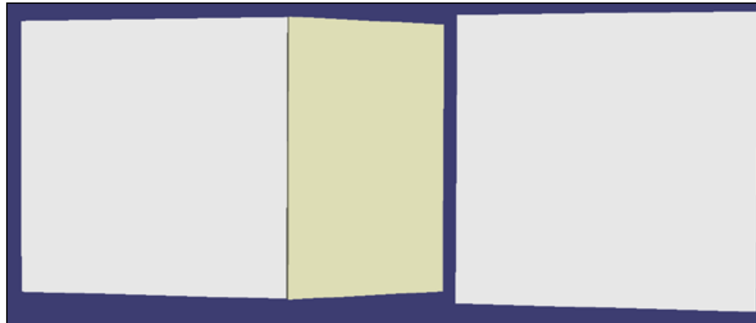
9. Create the scene graph with the updater and the manager set as node callbacks.

```
osg::MatrixTransform* animDoor = createDoor();
animDoor->setUpdateCallback( updater.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( createWall() );
root->addChild( animDoor );
root->setUpdateCallback( manager.get() );
```

10. Configure the handler and start the viewer.

```
osg::ref_ptr<OpenDoorHandler> handler = new OpenDoorHandler;
handler->_manager = manager.get();
handler->_animation = animation.get();
handler->_door = animDoor;

osgViewer::Viewer viewer;
viewer.addEventHandler( handler.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

11. Press *Ctrl* and click on the door with your mouse. Do you seen it opening smoothly? Now press *Ctrl* and click on it again to execute the closing animation. Reopen and reclose it more times if you wish.



How it works...

There are several important roles for an animation system to operate properly. The manager callback (`osgAnimation::BasicAnimationManager` and others) manages animation data including different types of channels and keyframes, it must be set to the root node to handle animations on all child nodes. The updaters (`osgAnimation::UpdateMatrixTransform` and others) must be set as callbacks of animating nodes. It can be connected with one or more channels that set it as the animation target. And its stacked elements, which represent different animating key types, must be matched to channels with the same name to read and use associated keyframes.

Compared with other 3D modelling and animating software like Autodesk 3dsmax, OSG's channels can be treated as tracks with keyframe data and interpolators, and stacked elements are in fact key filters which define animatable key types (position, rotation, and so on). In this example, we only construct one `'euler'` channel which is associated with the updater `'DoorAnimCallback'`. The stacked element `'euler'` is pushed into the updater to enable rotation along one Euler axis.

There's more...

osgAnimation presently supports five types of stacked elements, all of which are transformable elements. Material and morph updaters (the latter will be introduced in the fourth recipe of this chapter) don't need stacked elements at present. The stacked elements and their associatable channel types are listed in the following table:

Stacked element	Channel type	Key type
StackedMatrixElement	MatrixLinearChannel	osg::Matrix
StackedQuaternionElement	QuatSphericalLinearChannel	osg::Quat
StackedRotateAxisElement	FloatLinearChannel	float (along specific axis)
StackedScaleElement	Vec3LinearChannel	osg::Vec3
StackedTranslateElement	Vec3LinearChannel	osg::Vec3

Playing a movie in the 3D world

Have you ever dreamed of watching a movie in the 3D environment? We may consider creating a virtual cinema and put the movie on a big quad, a hemisphere, or some other irregular screens. The movie picture will be treated as the texture and mapped to the geometry mesh. OSG helps us handle the animating of the texture in an effective mode, that is, the `osg::ImageStream` class.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/ImageStream>
#include <osg/Geometry>
#include <osg/Geode>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. We provide two ways to show animated images here: one is to read an image sequence from the disk, another is to load frames from the webcam. The `osgdb_ffmpeg` plugin will do the low-level work for us here. But we have to first ensure that the webcam is the first reachable video-input device under Windows/Linux in this recipe.

```
osg::ArgumentParser arguments( &argc, argv );

osg::ref_ptr<osg::Image> image;
if ( arguments.argc()>1 )
  image = osgDB::readImageFile( arguments[1] );
else
{
  #ifdef WIN32
```

```
  image = osgDB::readImageFile( "0.ffmpeg",
    new osgDB::Options("format=vfwcap frame_rate=25") );
  #else
  image = osgDB::readImageFile( "/dev/video0.ffmpeg" );
  #endif
}
```

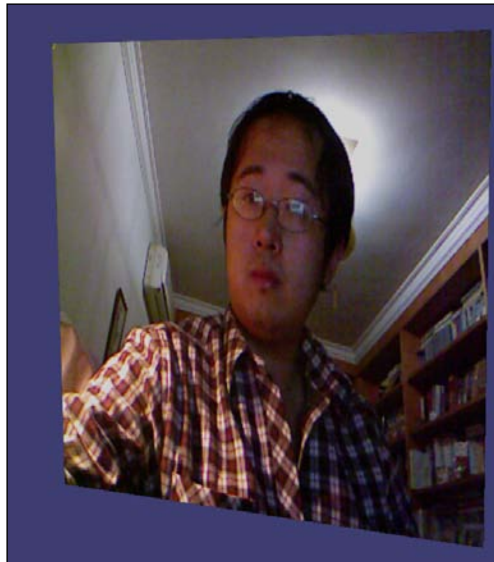
3. Try to convert the loaded image to the image stream and start to play it. If we don't pass a movie filename (for example, AVI and MPG) as the argument, or if the webcam can't be initialized, we will get a NULL pointer here and all the following code may fail due to the same reason.

```
osg::ImageStream* imageStream =
  dynamic_cast<osg::ImageStream*>( image.get() );
if ( imageStream ) imageStream->play();
```

4. Add the image to a 2D texture and apply it as the attribute of a simple quad with texture coordinates.

```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( image.get() );

osg::ref_ptr<osg::Drawable> quad =
  osg::createTexturedQuadGeometry(
osg::Vec3(), osg::Vec3(1.0f, 0.0f, 0.0f), osg::Vec3(
  0.0f, 0.0f, 1.0f) );
quad->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, texture.get() );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( quad.get() );
Start the viewer.
osgViewer::Viewer viewer;
viewer.setSceneData( geode.get() );
return viewer.run();
```

5. If you have already configured the webcam and didn't pass a filename instead, you may see your own figure in the 3D world (as shown in the following screenshot, it is just my room from a low-resolution webcam). It is an exciting functionality if you are developing a virtual chat room, or you want to do some image-recognition work and reflect the result in the 3D scene. **Augmented Reality** (**AR**) is also an interesting topic here with the movie texture implementation in this recipe as a foundation. Look for some extra materials by yourselves as these are already out of the scope of this book.



How it works...

OSG uses the **FFmpeg** library (`http://ffmpeg.org/`) and related OSG plugin to decode and play different kinds of media files. As FFmpeg can hardly be built with Visual Studio under Windows, you may first obtain the SDK packages at `http://ffmpeg.zeranoe.com/builds/`.

And then reset related CMake options to ensure that OSG plugin can be generated.

As FFmpeg supports webcam video under Windows and Linux, we may easily make use of these features by force opening the device through the `osgdb_ffmpeg` plugin. The pseudo-loader mechanism adds a `.ffmpeg` postfix to the real device name (for example, `0` under Windows or `/dev/video0` under Linux) so that it will be automatically transferred to the plugin with the same name.

There's more...

There are some other plugins for reading movie files as textures; each requiring extra dependencies and CMake options before they can be compiled:

- `osgdb_directshow`: Read DirectShow support formats. Requires DirectShow in DirectX SDK (`http://msdn.microsoft.com/en-us/directx`) as dependence.
- `osgdb_gif`: Read static and dynamic GIF pictures. Requires GifLib (`http://sourceforge.net/projects/giflib/`) as dependence.

- `osgdb_quicktime`: Read Apple Quicktime support formats. Requires QuickTime SDK (`http://developer.apple.com/quicktime/`) as dependence.
- `osgdb_QTKit`: Read Apple QuickTime Kit support formats. Requires QTKit framework (`http://developer.apple.com/quicktime/`) as dependence. Only works under Mac OS X.

You may also have a look at the osgART library, which provides a series of APIs for implementing AR functionalities in OSG. It uses its own way to render dynamic images from the webcam devices.

Designing scrolling text

Scrolling text is a classic functionality in many cases. For instance, HTML use the `<marquee>` tag to display texts sliding in and out on the web page, either from left to right or from right to left. They can be used for the purpose of UI design or emphasizing the importance of the contents. In this recipe, we will design simple one line scrolling texts which continuously move on the screen and change its content dynamically.

How to do it...

Let us start.

1. Include necessary headers and define the macro for generating random numbers:

```
#include <osgAnimation/EaseMotion>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <sstream>
#include <iomanip>
#define RAND(min, max) ((min) + (float)rand()/(RAND_MAX+1) *
  ((max)-(min)))
```

2. The `ScrollTextCallback` class will be set to the drawable to change its behaviors during the update traversal. It computes the Y position of a one-line text randomly, and then starts to move it along the X direction. When it reaches the right end of the screen, the callback will compute a new Y position and will restart from the left-hand side again.

```
class ScrollTextCallback : public osg::Drawable::UpdateCallback
{
public:
  ScrollTextCallback()
  {
    _motion = new osgAnimation::LinearMotion;
    computeNewPosition();
```

```
  }

  virtual void update( osg::NodeVisitor* nv, osg::Drawable*
    drawable );

  void computeNewPosition()
  {
    _motion->reset();
    _currentPos.y() = RAND(50.0, 500.0);
  }

protected:
  osg::ref_ptr<osgAnimation::LinearMotion> _motion;
  osg::Vec3 _currentPos;
};
```

3. In the `operator()` method, we process the text-moving animation using an `osgAnimation::LinearMotion` object, which returns the result of a linear interpolation. The actual X and Y values are also put into a string in every frame for dynamically changing the text content.

```
osgText::Text* text = static_cast<osgText::Text*>( drawable );
if ( !text ) return;

_motion->update( 0.002 );
float value = _motion->getValue();
if ( value>=1.0f ) computeNewPosition();
else _currentPos.x() = value * 800.0f;

std::stringstream ss; ss << std::setprecision(3);
ss << "XPos: " << std::setw(5) << std::setfill(' ')
  << _currentPos.x() << ";
YPos: " << std::setw(5) << std::setfill(' ')
  << _currentPos.y();
text->setPosition( _currentPos );
text->setText( ss.str() );
```

4. In the main entry, we simply use the convenient functions to create a text, add the update callback to it, and use an HUD camera to display it.

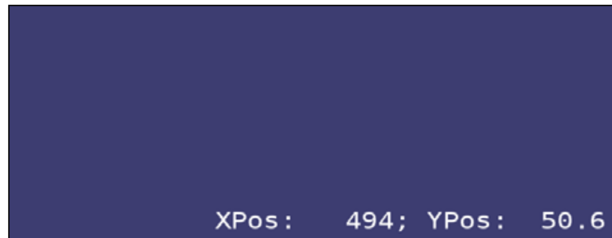
```
osgText::Text* text = osgCookBook::createText(
  osg::Vec3(), "", 20.0f);
text->addUpdateCallback( new ScrollTextCallback );

osg::ref_ptr<osg::Geode> textGeode = new osg::Geode;
textGeode->addDrawable( text );
```

```
osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0, 800, 0, 600);
hudCamera->addChild( textGeode.get() );
```

5. Start the viewer now.

```
osgViewer::Viewer viewer;
viewer.setSceneData( hudCamera.get() );
return viewer.run();
```

6. You will see the text is sliding from left to right, with the content varying all the time. You can easily modify this example to use it in your own applications and screensavers.



How it works...

Here we introduced the `osgAnimation::LinearMotion` class, which belongs to the `EaseMotion` header. Easing means a change in speed. And in the osgAnimation implementation, ease motion means transition between the moving and stopping states. The velocity of a moving object must change when it is going to stop somewhere, and vice versa. Easing actually decides the acceleration values when velocity is changing.

An object starts and speeds up, this is called an 'in' motion. It slows down and finally stops, this is an 'out' motion. If we combine them with a half-and-half ratio, this is so called 'in-out' motion. Thus, an 'in-out-cubic' motion means a cubic equation will be used while computing the velocity values at the beginning and end parts of a motion curve.

Linear motion is special because it doesn't have velocity changes during the entire movement. The object will start and stop suddenly without any easing. For a scrolling text, which will only stop at the screen's right edge and start immediately at another edge, linear motion will be the most appropriate one to use.

There's more...

OSG provides the following types of ease motions besides linear motion, each with 'in', 'out', and 'in-out' forms: `QuadMotion` (quadratic equation), `CubicMotion` (cubic equation), `QuartMotion` (quartic equation), `BounceMotion`, `ElasticMotion`, `SineMotion` (sinusoidal equation), `BackMotion`, `CircMotion` (circular equation), and `ExpoMotion` (exponential equation).

For example, an 'in' type `CubicMotion` class is written as `osgAnimation::InCubicMotion`. You may call `update()` method with a time parameter (between 0 and 1) to update and get the result back with the `getValue()` method (also limited in [0, 1]).

Ease motion classes can be used separately for various purposes.

There is a good website explaining the concepts and implementations of different ease motions. You can read it for more details:

`http://www.robertpenner.com/easing`

Implementing morph geometry

Morphing is a special effect used in image processing and 3D animations. It always morphs the source image or model into another through a seamless transition. Modern morphing techniques require some advanced algorithms and operations and can work in very complex cases. But OSG provides a lightweight solution named `osgAnimation::MorphGeometry`, which can also produce fantastic results in real-time environments.

Although it is great to implement something like face morphing which is already possible in OSG now, we have to simplify the situation here by creating a really easy geometry and change it to another easy one. The emoticon (facial expressions represented by letters) may be simple and interesting enough for demonstration this time.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Point>
#include <osg/Geometry>
#include <osg/Geode>
#include <osgAnimation/MorphGeometry>
#include <osgAnimation/BasicAnimationManager>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The `createEmoticonGeometry()` function here will create an emoticon with two points representing the eyes, and another 13 points representing the mouth. We will pass a function address as the function argument, in which the mouth shape will be described and pushed into the vertex array.

```
typedef void (*VertexFunc)( osg::Vec3Array* );
osg::Geometry* createEmoticonGeometry( VertexFunc func )
{
  osg::ref_ptr<osg::Vec3Array> vertices =
    new osg::Vec3Array(15);
  (*vertices)[0] = osg::Vec3(-0.5f, 0.0f, 1.0f);
  (*vertices)[1] = osg::Vec3(0.5f, 0.0f, 1.0f);
  (*func)( vertices.get() );

  osg::ref_ptr<osg::Vec3Array> normals =
    new osg::Vec3Array(15);
  for ( unsigned int i=0; i<15; ++i )
  (*normals)[i] = osg::Vec3(0.0f, -1.0f, 0.0f);

  osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
  geom->setVertexArray( vertices.get() );
  geom->setNormalArray( normals.get() );
  geom->setNormalBinding( osg::Geometry::BIND_PER_VERTEX );
  geom->addPrimitiveSet( new osg::DrawArrays(
    GL_POINTS, 0, 2) );
  geom->addPrimitiveSet( new osg::DrawArrays(
    GL_LINE_STRIP, 2, 13) );
  return geom.release();
}
```

3. The `emoticonSource()` function creates a normal mouth which is only a straight line (`:|`). It means that the person is annoyed.

```
void emoticonSource( osg::Vec3Array* va )
{
  for ( int i=0; i<13; ++i )
  (*va)[i+2] = osg::Vec3((float)(i-6)*0.15f, 0.0f, 0.0f);
}
```

4. The `emoticonTarget()` function will create a curved mouth instead. It actually looks like the famous emoticon (`:)`), that is, joking or with joy.

```
void emoticonTarget( osg::Vec3Array* va )
{
  float angleStep = osg::PI / 12.0f;
  for ( int i=0; i<13; ++i )
  {
```

```
        float angle = osg::PI - angleStep * (float)i;
        (*va)[i+2] = osg::Vec3(0.9f*cosf(angle), 0.0f,
          -0.2f*sinf(angle));
      }
    }
```

5. The morph keyframes will indicate the next target geometry's index, which will be changed into from the source geometry (at index `0`).

```
void createMorphKeyframes( osgAnimation::FloatLinearChannel*
  ch )
{
  osgAnimation::FloatKeyframeContainer* kfs =
    ch->getOrCreateSampler()->getOrCreateKeyframeContainer();
  kfs->push_back( osgAnimation::FloatKeyframe(0.0, 0.0) );
  kfs->push_back( osgAnimation::FloatKeyframe(2.0, 1.0) );
}
```

6. In the main entry, create the channel and animate the object for morphing. The channel must have a valid name for indicating the starting position of the morphing targets.

```
osg::ref_ptr<osgAnimation::FloatLinearChannel> channel =
  new osgAnimation::FloatLinearChannel;
channel->setName( "0" );
channel->setTargetName( "MorphCallback" );
createMorphKeyframes( channel.get() );

osg::ref_ptr<osgAnimation::Animation> animation =
  new osgAnimation::Animation;
animation->setPlayMode( osgAnimation::Animation::PPONG );
animation->addChannel( channel.get() );
```

7. Add the animation to the manager.

```
osg::ref_ptr<osgAnimation::BasicAnimationManager> manager =
  new osgAnimation::BasicAnimationManager;
manager->registerAnimation( animation.get() );
manager->playAnimation( animation.get() );
```

8. Create the morph geometry by duplicating the source emoticon (you may also create a new one, but you have to then add all vertices and primitives on your own), and add the target emoticon to the morph geometry for use. The morph geometry must be added to an `osg::Geode` node then. And we have to also add an `osgAnimation::UpdateMorph` object as the update callback for connecting the morph with the channel we set just now.

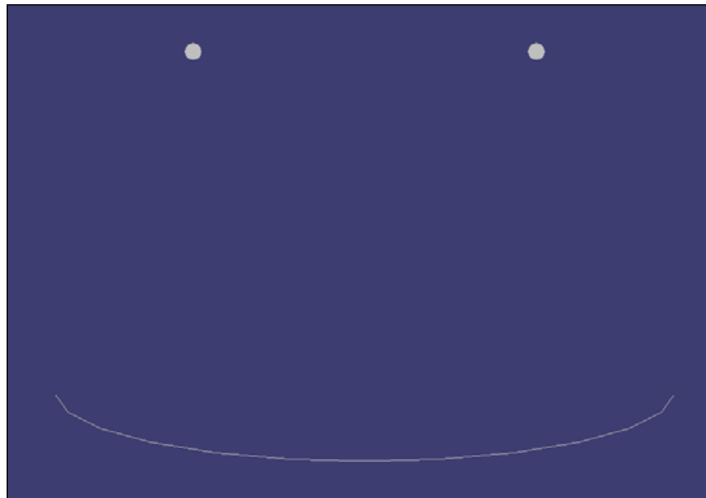
```
osg::ref_ptr<osgAnimation::MorphGeometry> morph =
  new osgAnimation::MorphGeometry(
```

```
    *createEmoticonGeometry(emoticonSource) );
morph->addMorphTarget( createEmoticonGeometry(emoticonTarget) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( morph.get() );
geode->addUpdateCallback(
  new osgAnimation::UpdateMorph("MorphCallback") );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::Point(20.0f) );
```

9. Add the node to the scene graph and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );
root->setUpdateCallback( manager.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

10. You will find that the emoticon is changing from normal state to a smiling form, and then changing back. It is really rough, but still provides some of the basic conditions of implementing morph animations in OSG: the source and target must be both `osg::Geometry` objects, and must have the same number of vertices so that the morphing operation can be performed smoothly all the time.



How it works...

The basic concept of morphing is to construct a source geometry and one or more target geometries. And change the original one into target ones with different weight settings. It requires an `osgAnimation::MorphGeometry` object which includes the source and target geometries, and an updater (`osgAnimation::UpdateMorph`) for connecting with the animation channel. The channel's name must be the same as the index of certain target geometry to make sure they are linked together.

There's more...

The example `osganimationmorph` is another good example for demonstrating the morphing animation. It loads two pre-created models with the same number of vertices and generates a transition from one to the other. You may find the implementation in the examples folder of the source code.

Fading in and out

In this recipe, we will try to implement a practical functionality. When a model is far away from the viewer, it will gradually be transparentized and finally disappear (fade out); and when the viewer moves towards the model and is near enough, the model will appear again. The fade in and out effects can be done by adding a material state to the model surface, or applying a texture with an alpha channel. We will choose the former as shown in the code segments.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/BlendFunc>
#include <osg/Material>
#include <osg/Node>
#include <osgAnimation/EaseMotion>
#include <osgDB/ReadFile>
#include <osgUtil/CullVisitor>
#include <osgViewer/Viewer>
```

2. The fade in/out effects will be done in the node callback. It requires an `osg::Material` object which is also set to the node itself. The alpha component of the diffuse color will be changing during the traversal in every frame, so that the opacity of the node will vary as a result.

```
class FadeInOutCallback : public osg::NodeCallback
{
public:
  FadeInOutCallback( osg::Material* mat )
  :  _material(mat), _lastDistance(-1.0f), _fadingState(0)
  {
    _motion = new osgAnimation::InOutCubicMotion;
  }

  virtual void operator()( osg::Node* node, osg::NodeVisitor*
    nv );

protected:
  osg::ref_ptr<osgAnimation::InOutCubicMotion> _motion;
  osg::observer_ptr<osg::Material> _material;
  float _lastDistance;
  int _fadingState;
};
```

3. In the `operator()` method, which is the actual implementation of the callback, we have two jobs to do: one is to change the diffuse color if the fading animation is running (`_fadingState` is non-zero). A cubic motion object will be used to compute a suitable alpha value (between 0.0 and 1.0, increasing or decreasing) here.

```
if ( _fadingState!=0 )
{
  _motion->update( 0.05 );
  float value = _motion->getValue();
  float alpha = (_fadingState>0 ? value : 1.0f - value);
  _material->setDiffuse( osg::Material::FRONT_AND_BACK,
    osg::Vec4(1.0f, 1.0f, 1.0f, alpha) );

  if ( value>=1.0f ) _fadingState = 0;
  traverse( node, nv ); return;
}
```

4. Another thing we have to check is the distance between the viewer's eye and the node's center. If there is no fade-in/out animation running currently, the check will be processed by calling the `getDistanceFromEyePoint()` method of `osgUtil::CullVisitor` class. Then we will decide if there should be a new fading-in or fading-out effect.

```
osgUtil::CullVisitor* cv = static_cast<osgUtil::CullVisitor*
  >( nv );
if ( cv )
{
  float distance = cv->getDistanceFromEyePoint(
    node->getBound().center(), true );
  if ( _lastDistance>0.0f )
  {
    if ( _lastDistance>200.0f && distance<=200.0f )
    {
      _fadingState = 1; _motion->reset();
    }
    else if ( _lastDistance<200.0f && distance>=200.0f )
    {
      _fadingState =-1; _motion->reset();
    }
    }
  _lastDistance = distance;
}
traverse( node, nv );
```

5. In the main entry, we load the model and apply a new material object to it. Don't forget to enable blending and transparent sorting on this node.

```
osg::Node* loadedModel = osgDB::readNodeFile( "cessna.osg" );
if ( !loadedModel ) return 1;

osg::ref_ptr<osg::Material> material = new osg::Material;
material->setAmbient( osg::Material::FRONT_AND_BACK,
  osg::Vec4(0.0f, 0.0f, 0.0f, 1.0f) );
material->setDiffuse( osg::Material::FRONT_AND_BACK,
  osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  material.get(), osg::StateAttribute::ON|osg::StateAttribute::OVE
RRIDE );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  new osg::BlendFunc );
loadedModel->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
```

6. Add the `FadeInOutCallback` as a cull callback which will be executed while culling scene objects.

```
loadedModel->addCullCallback(
  new FadeInOutCallback(material.get()) );
```

7. Add the node to the scene graph and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( loadedModel );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

8. You can find the node rendered normally at first. Zoom out by pressing and dragging the right mouse button, and the Cessna will lighten and disappear. Zoom in and then you can see it coming out again. This provides a smooth effect when we go near and far from a model, without any bad pop-in effects such as the model suddenly presenting itself to the end users.

How it works...

The `FadeInOutCallback` class in this recipe uses a fixed value as a threshold. And start the fading-in or fading-out animation while the real distance from eye-to-model center comes across the threshold. It must be used as a cull callback because only the cull visitor (`osgUtil::CullVisitor`) can obtain model and view matrices and, thus, compute the position of the model in eye coordinates. You may rewrite this recipe and create a customized node type for such situations. The compass example in *Chapter 2* will be a good reference.

Maybe you have also noticed that the Cessna model is not in good shape while being transparentized (but it requires very good eyesight here). That is because we can't perfectly sort the polygons in the Cessna model for alpha blending. If we can split these polygons up on the fly, sort, and redraw the Cessna in geometry level, the Cessna can be rendered well, but it will cause efficiency losses. And it is hard to design a perfect algorithm too. A possible solution for modern graphic cards is **depth peeling**. We will discuss that in the last chapter of this book.

Animating a flight on fire

OSG provides a complex particle framework, which can design the behaviour of each particle object from its birth to its death. The creation of new particles is realized by emitters, and all post-creation effects will be implemented by programs and its child operators. The basic attributes of each particle will be stored in a template that is managed by the particle system. A system is actually a drawable, and is often updated by a global updater node.

OSG supports multiple particle systems so we can have more than one emitter, program, and particle system. In this example, we will create a fire and a smoke particle system to simulate a Cessna model on fire.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Point>
#include <osg/PointSprite>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgParticle/ModularEmitter>
#include <osgParticle/ParticleSystemUpdater>
#include <osgViewer/Viewer>
```

2. First we use the `createFireParticles()` function to create a fire particle system, including the particle template settings and the emitter which handles the number, initial positions, and velocities of new-born particles.

```
osgParticle::ParticleSystem* createFireParticles(
  osg::Group* parent )
{
  ...
}
```

3. In the function, allocate the particle system and set up a suitable particle color and a smoke-like texture to simulate the tongue of flame.

```
osg::ref_ptr<osgParticle::ParticleSystem> ps =
  new osgParticle::ParticleSystem;
ps->getDefaultParticleTemplate().setLifeTime( 1.5f );
ps->getDefaultParticleTemplate().setShape(
  osgParticle::Particle::QUAD );
ps->getDefaultParticleTemplate().setSizeRange(
  osgParticle::rangef(3.0f, 1.5f) );
ps->getDefaultParticleTemplate().setAlphaRange(
  osgParticle::rangef(1.0f, 0.0f) );
ps->getDefaultParticleTemplate().setColorRange(
  osgParticle::rangev4(osg::Vec4(1.0f,1.0f,0.5f,1.0f),
    osg::Vec4(1.0f,0.5f,0.0f,1.0f)) );
ps->setDefaultAttributes( "Images/smoke.rgb", true, false );
```

4. Generate a random number of particles (between 30 and 50) in every frame. And set up the shooting direction range and initial speed.

```
osg::ref_ptr<osgParticle::RandomRateCounter> rrc =
  new osgParticle::RandomRateCounter;
rrc->setRateRange( 30, 50 );

osg::ref_ptr<osgParticle::RadialShooter> shooter =
  new osgParticle::RadialShooter;
shooter->setThetaRange( -osg::PI_4, osg::PI_4 );
shooter->setPhiRange( -osg::PI_4, osg::PI_4 );
shooter->setInitialSpeedRange( 5.0f, 7.5f );
```

5. Set the counter and shooter to the emitter and add it to a parent node to ensure the emitter is part of the scene graph.

```
osg::ref_ptr<osgParticle::ModularEmitter> emitter =
  new osgParticle::ModularEmitter;
emitter->setParticleSystem( ps.get() );
emitter->setCounter( rrc.get() );
emitter->setShooter( shooter.get() );
parent->addChild( emitter.get() );
return ps.get();
```

6. The creation of a smoke-particle system is very similar to the last function, except for some changes in particle attributes and emitter values. The smoke particle is usually darker, bigger, and moving faster than the flame, so some related attributes will be altered. Please refer to the source code package of this book for details of the implementation.

```
osgParticle::ParticleSystem* createSmokeParticles(
  osg::Group* parent )
{
  ...
}
```

7. In the main entry, we will create a transformation node which will be transformed to one of the wings of the Cessna model. Both emitters will be added to the transformation node to make sure all the particles are emitted under the specified local coordinates.

```
osg::ref_ptr<osg::MatrixTransform> parent =
  new osg::MatrixTransform;
parent->setMatrix( osg::Matrix::rotate(
  -osg::PI_2, osg::X_AXIS) * osg::Matrix::translate(
    8.0f,-10.0f,-3.0f) );

osgParticle::ParticleSystem* fire = createFireParticles(
```

```
  parent.get() );
osgParticle::ParticleSystem* smoke = createSmokeParticles(
  parent.get() );
```

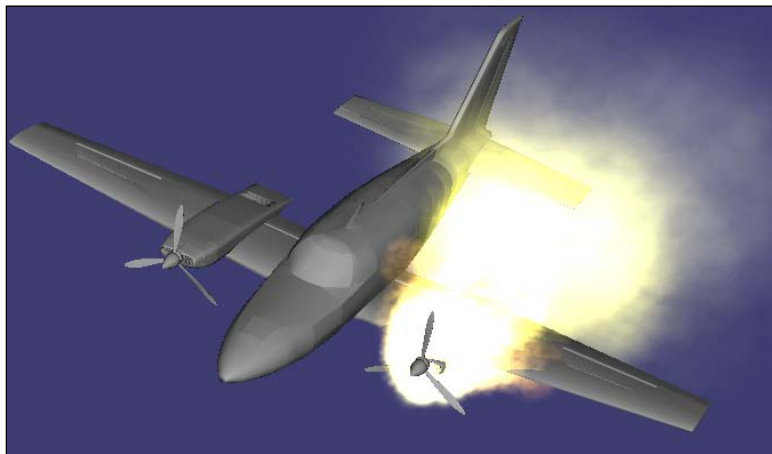
8. A particle system updater will be used to update all kinds of particle systems. And an `osg::Geode` node can display the particles managed by these two systems in the 3D world.

```
osg::ref_ptr<osgParticle::ParticleSystemUpdater> updater =
  new osgParticle::ParticleSystemUpdater;
updater->addParticleSystem( fire );
updater->addParticleSystem( smoke );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( fire );
geode->addDrawable( smoke );
```

9. Add the Cessna model and all above nodes to the root node, and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( osgDB::readNodeFile("cessna.osg") );
root->addChild( parent.get() );
root->addChild( updater.get() );
root->addChild( geode.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

10. Now we successfully made it happen: a Cessna is on fire! Its motor on one of the wings is blazing seriously. Will the Cessna crash after a while? Or someone could save it at the last moment? Now, you can continue writing the story by yourself.



How it works...

The osgParticle framework is as complex and powerful as osgAnimation. It is made up of at least a particle system, an emitter that controls the birth of particles, a program that controls a particle's behaviours after being born, and an updater that manages multiple systems and updates them. As emitters, programs, and updaters are all scene nodes, it is possible to put one or more of them under a transformation node to create different particle effects. Here we only place the emitters to the wing with an `osg::MatrixTransform` node so that all newly-allocated particles will come from the engine which seems to be broken, and the particles will also float in the sky if the flight is moving or falling.

There's more...

This example actually imitates an existing OSG model file named `cessnafire.osg`. You may look into the file content with any text editors and try to analyze the node and particle structures of it.

Dynamically lighting within shaders

Using shaders is a very popular topic now-a-days. As shading language is often the base of many advanced rendering effects, there is no reason not to use it in our OSG applications. In Chapter 2, we have already introduced the integration of OSG and NVIDIA Cg. But in this and the next chapter, we will return to GLSL and try to make use of different shaders with a lively mind. In this example, we are going to implement a simple phone shader and animate the light position so that diffuse and specular lights on the model surface will be animated at runtime.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Program>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. First we will design the vertex shader. It requires a light position, and it computes the direction from eye to the light point, which will be used for per-pixel lighting implementation in the fragment shader.

```
static const char* vertSource =
{
  "uniform vec3 lightPosition;\n"
  "varying vec3 normal, eyeVec, lightDir;\n"
```

```
    "void main()\n"
    "{\n"
    "vec4 vertexInEye = gl_ModelViewMatrix * gl_Vertex;\n"
    "eyeVec = -vertexInEye.xyz;\n"
    "lightDir = vec3(lightPosition - vertexInEye.xyz);\n"
    "normal = gl_NormalMatrix * gl_Normal;\n"
    "gl_Position = ftransform();\n"
    "}\n"
};
```

3. In the fragment shader, we will compute a suitable surface color according to the varying normal and light direction vectors, and other uniform light parameters. The resultant color doesn't contain texture components, but with only a little shader programming experience, you will be able to add some more contents.

```
static const char* fragSource =
{
    "uniform vec4 lightDiffuse;\n"
    "uniform vec4 lightSpecular;\n"
    "uniform float shininess;\n"
    "varying vec3 normal, eyeVec, lightDir;\n"
    "void main (void)\n"
    "{\n"
    "vec4 finalColor = gl_FrontLightModelProduct.sceneColor;\n"
    "vec3 N = normalize(normal);\n"
    "vec3 L = normalize(lightDir);\n"
    "float lambert = dot(N,L);\n"
    "if (lambert > 0.0)\n"
    "{\n"
    "finalColor += lightDiffuse * lambert;\n"
    "vec3 E = normalize(eyeVec);\n"
    "vec3 R = reflect(-L, N);\n"
    "float specular = pow(max(dot(R, E), 0.0), shininess);\n"
    "finalColor += lightSpecular * specular;\n"
    "}\n"
    "gl_FragColor = finalColor;\n"
    "}\n"
};
```

4. The `LightPosCallback` class can handle specific GLSL uniform dynamically, and set a new value to it every frame. In this recipe, we simply set up a new position according to current frame number.

```
class LightPosCallback : public osg::Uniform::Callback
{
public:
  virtual void operator()( osg::Uniform* uniform,
    osg::NodeVisitor* nv )
  {
    const osg::FrameStamp* fs = nv->getFrameStamp();
    if ( !fs ) return;

    float angle = osg::inDegrees( (float)fs->getFrameNumber() );
    uniform->set( osg::Vec3(20.0f * cosf(angle), 20.0f *
      sinf(angle), 1.0f) );
  }
};
```

5. In the main entry, we will first load a model, apply the `osg::Program` object with two shaders to its state set, and add uniforms with initial values.

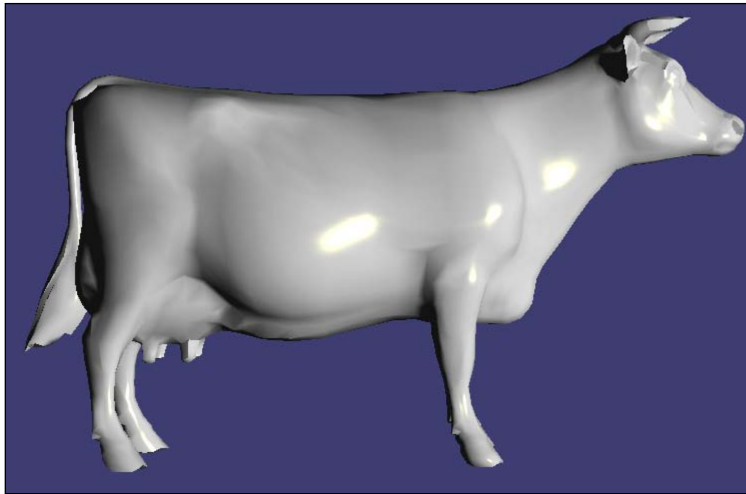
```
osg::ref_ptr<osg::Node> model = osgDB::readNodeFile(
  "cow.osg" );

osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertSource) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  fragSource) );

osg::StateSet* stateset = model->getOrCreateStateSet();
stateset->setAttributeAndModes( program.get() );
stateset->addUniform( new osg::Uniform("lightDiffuse",
  osg::Vec4(0.8f, 0.8f, 0.8f, 1.0f)) );
stateset->addUniform( new osg::Uniform("lightSpecular",
  osg::Vec4(1.0f, 1.0f, 0.4f, 1.0f)) );
stateset->addUniform( new osg::Uniform("shininess", 64.0f) );
```

6. The light position uniform variable will be controlled by an update callback, as shown in the following code segment:

```
osg::ref_ptr<osg::Uniform> lightPos = new osg::Uniform(
"lightPosition", osg::Vec3() );
lightPos->setUpdateCallback( new LightPosCallback );
```

```
stateset->addUniform( lightPos.get() );
Start the viewer.
osgViewer::Viewer viewer;
viewer.setSceneData( model.get() );
return viewer.run();
```

7. Here we actually use one moving light and compute the surface color according to user-defined shader parameters. You may create the same effect in traditional ways, but shaders provide more flexibility and they usually have better rendering results, for instance, per-pixel effects (which is impossible in fixed pipeline).



How it works...

In this example, we defined few uniform light parameters in the shader code, so we can demonstrate the usage of uniform callbacks. It is also possible in GLSL to use the inbuilt uniform array `gl_LightSource[]`. For example, we can use `gl_LightSource[0].position` to represent the first light's position in the scene. And to change the position and make it work in shaders, we can add an `osg::LightSource` node in the scene graph, and use a node callback to execute `osg::Light`'s `setPosition()` method.

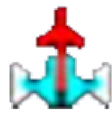
The `osg::Uniform` class has an update callback and an event callback, using the same `osg::Uniform::Callback` structure. You may use either to update uniform variables on the fly.

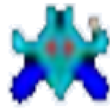
Creating a simple Galaxian game

The Galaxian is a classic 2D game published in Japan in the 1980s. It includes a large number of aliens attacking the player by shooting bullets and making kamikaze-like operations. In this recipe, we will try our best to make such a Galaxian game in OSG and implement most of its features. We don't have enough space in this book to write thousands of lines of source code, so another important task is that we have to limit our code to at most 200-250 lines. But believe me, it is enough for implementing most kinds of functionalities, no matter how simple or complex.

Getting ready

Let us first prepare three RGBA pictures (in PNG format) named `player.png`, `enemy.png`, and `bullet.png`. They are going to be used for describing the roles' shapes in the game. Example images are shown in the following screenshot:


player.png


enemy.png

bullet.png

How to do it...

Let us start.

1. Include necessary headers and define a macro for generating random numbers:

```
#include <osg/Texture2D>
#include <osg/Geometry>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#define RAND(min, max) ((min) + (float)rand()/(RAND_MAX+1) *
  ((max)-(min)))
```

2. Define a `Player` class which is actually a transformation node in the scene graph. The `Player` class not only means the player's role in the game, but also represents enemy aliens and bullets. Its `width()` and `height()` returns the size of the player in a 2D space (which is in fact an HUD camera in OSG, which ignores the Z direction but places all scene objects in the `XOY` plane), `setSpeedVector()` sets the speed of the vector which will make it move every frame, and `setPlayerType()` defines whether the node is a player, a player's bullet, an enemy, or an enemy's bullet.

```
class Player : public osg::MatrixTransform
{
public:
  Player() : _type(INVALID_OBJ) {}
  Player( float width, float height, const std::string&
    texfile );

  float width() const { return _size[0]; }
  float height() const { return _size[1]; }

  void setSpeedVector( const osg::Vec3& sv )
  {
    _speedVec = sv;
  }
  const osg::Vec3& getSpeedVector() const
  {
    return _speedVec;
  }

  enum PlayerType
  {
    INVALID_OBJ=0, PLAYER_OBJ, ENEMY_OBJ,
    PLAYER_BULLET_OBJ, ENEMY_BULLET_OBJ
  };
  void setPlayerType( PlayerType t ) { _type = t; }
  PlayerType getPlayerType() const { return _type; }

  bool isBullet() const
  {
    return _type==PLAYER_BULLET_OBJ ||
    _type==ENEMY_BULLET_OBJ; }

  bool update( const osgGA::GUIEventAdapter& ea,
    osg::Group* root );
  bool intersectWith( Player* player ) const;

protected:
```

```
    osg::Vec2 _size;
    osg::Vec3 _speedVec;
    PlayerType _type;
};
```

3. The constructor of the `Player` class can accept width and height parameters and a texture filename as the input arguments. It will create a textured quad, put it into an `osg::Geode` node, and add the geode as its own child at the end.

```
Player::Player( float width, float height, const std::string&
  texfile )
:  _type(INVALID_OBJ)
{
  _size.set( width, height );
  osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
  texture->setImage( osgDB::readImageFile(texfile) );

  osg::ref_ptr<osg::Drawable> quad =
    osg::createTexturedQuadGeometry(
  osg::Vec3(-width*0.5f, -height*0.5f, 0.0f),
  osg::Vec3(width, 0.0f, 0.0f), osg::Vec3(0.0f, height, 0.0f) );
  quad->getOrCreateStateSet()->setTextureAttributeAndModes(
    0, texture.get() );
  quad->getOrCreateStateSet()->setMode( GL_LIGHTING,
    osg::StateAttribute::OFF );
  quad->getOrCreateStateSet()->setRenderingHint(
    osg::StateSet::TRANSPARENT_BIN );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( quad.get() );
  addChild( geode.get() );
}
```

4. The `update()` method is the core function of the `Player` class. It defines the behaviors of the node when user events (including `FRAME` event) are coming. A player or enemy node may shoot bullets in the `update()` method, so the root node should also be passed to accept newly allocated bullet nodes.

```
bool Player::update( const osgGA::GUIEventAdapter& ea,
  osg::Group* root )
{
  ...
}
```

5. In the `update()` method, we will first check if user presses the left, right, or return key. It will cause a node of the player type to move and shoot. Enemy nodes will shoot randomly regardless of user inputs.

```
bool emitBullet = false;
switch ( _type )
{
case PLAYER_OBJ:
  if ( ea.getEventType()==osgGA::GUIEventAdapter::KEYDOWN )
  {
    switch ( ea.getKey() )
    {
      case osgGA::GUIEventAdapter::KEY_Left:
        _speedVec = osg::Vec3(-0.1f, 0.0f, 0.0f);
      break;
      case osgGA::GUIEventAdapter::KEY_Right:
        _speedVec = osg::Vec3(0.1f, 0.0f, 0.0f);
      break;
      case osgGA::GUIEventAdapter::KEY_Return:
        emitBullet = true;
      break;
      default: break;
    }
  }
  else if ( ea.getEventType()==osgGA::GUIEventAdapter::KEYUP )
  _speedVec = osg::Vec3();
  break;
case ENEMY_OBJ:
  if ( RAND(0, 2000)<1 ) emitBullet = true;
  break;
default: break;
}
```

6. Secondly, we will check if a new bullet should be generated and emitted. A player's bullet will start from the bottom of the screen and speed up to the top; and an enemy's bullet will go from the top to the bottom, trying to destroy the player's role. The bullet node should be added to the root node to make it visible in the space.

```
osg::Vec3 pos = getMatrix().getTrans();
if ( emitBullet )
{
  osg::ref_ptr<Player> bullet = new Player(
    0.4f, 0.8f, "bullet.png");
  if ( _type==PLAYER_OBJ )
  {
    bullet->setPlayerType( PLAYER_BULLET_OBJ );
```

```
    bullet->setMatrix( osg::Matrix::translate(
      pos + osg::Vec3(0.0f, 0.9f, 0.0f)) );
    bullet->setSpeedVector( osg::Vec3(0.0f, 0.2f, 0.0f) );
  }
  else
  {
    bullet->setPlayerType( ENEMY_BULLET_OBJ );
    bullet->setMatrix( osg::Matrix::translate(
      pos - osg::Vec3(0.0f, 0.9f, 0.0f)) );
    bullet->setSpeedVector( osg::Vec3(0.0f,-0.2f, 0.0f) );
  }
  root->addChild( bullet.get() );
}
```

7. In the per-frame event, we actually add the speed vector to the transformation matrix, so that the player/enemy/bullet will actually start to move and fight.

```
if ( ea.getEventType()!=osgGA::GUIEventAdapter::FRAME )
return true;float halfW = width() * 0.5f, halfH = height() *
  0.5f;
pos += _speedVec;
// Don't update the player anymore if it is not in the visible //
area.
if ( pos.x()<halfW || pos.x()>ea.getWindowWidth()-halfW )
  return false;
if ( pos.y()<halfH || pos.y()>ea.getWindowHeight()-halfH )
  return false;
setMatrix( osg::Matrix::translate(pos) );
return true;
```

8. The `intersectWith()` function can quickly check if current node is intersected with another node. This is useful if we want to check whether the bullet hits another node and kills it.

```
bool Player::intersectWith( Player* player ) const
{
  osg::Vec3 pos = getMatrix().getTrans();
  osg::Vec3 pos2 = player->getMatrix().getTrans();
  return fabs(pos[0] - pos2[0]) < (width() + player->width())
    * 0.5f &&
  fabs(pos[1] - pos2[1]) < (height() + player->height())
    * 0.5f;
}
```

9. OK, now we must have a global game controller who will manage all roles in the game and update them one by one. It also removes nodes destroyed by bullets and bullets themselves.

```
class GameControllor : public osgGA::GUIEventHandler
{
public:
  GameControllor( osg::Group* root )
  : _root(root), _direction(0.1f), _distance(0.0f) {}

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

protected:
  osg::observer_ptr<osg::Group> _root;
  float _direction;
  float _distance;
};
```

10. In the `handle()` method, we will manage a `_distance` and a `_direction` variable. They will help the enemy aliens move from left to right. As they will also randomly shoot bullets, it will be not easy for us to beat all of them.

```
_distance += fabs(_direction);
if ( _distance>30.0f )
{
  _direction = -_direction;
  _distance = 0.0f;
}
```

11. The most important part of the game will be done here: All nodes registered in the controller will be updated; bullets out of the screen will be removed; the player and enemies will be destroyed when they knock into a bullet from the opposite roles. Although we don't have a 'welcome' and a 'game over' splash, and the aliens are so silly that they don't have any artificial intelligence, the game is still completed with most Galaxian features added.

```
osg::NodePath toBeRemoved;
for ( unsigned i=0; i<_root->getNumChildren(); ++i )
{
  Player* player = static_cast<Player*>( _root->getChild(i) );
  if ( !player ) continue;

  // Update the player matrix, and remove the player if it is
  //a bullet outside the visible area
  if ( !player->update(ea, _root.get()) )
  {
```

```
    if ( player->isBullet() )
    toBeRemoved.push_back( player );
  }

  // Automatically move the enemies
  if ( player->getPlayerType()==Player::ENEMY_OBJ )
  player->setSpeedVector( osg::Vec3(_direction, 0.0f, 0.0f) );
  if ( !player->isBullet() ) continue;

  // Use a simple loop to check if any two of the players
  // (you, enemies, and bullets) are intersected
  for ( unsigned j=0; j<_root->getNumChildren(); ++j )
  {
    Player* player2 = static_cast<Player*>( _root->getChild(j) );
    if ( !player2 || player==player2 ) continue;

    if ( player->getPlayerType()==Player::ENEMY_BULLET_OBJ &&
    player2->getPlayerType()==Player::ENEMY_OBJ )
    {
      continue;
    }
    else if ( player->intersectWith(player2) )
    {
      // Remove both players if they collide with each other
      toBeRemoved.push_back( player );
      toBeRemoved.push_back( player2 );
    }
  }
}
```

12. At last, remove the nodes that will be destroyed.

```
for ( unsigned i=0; i<toBeRemoved.size(); ++i )
  _root->removeChild( toBeRemoved[i] );
return false;
```

13. We have nearly finished the project. In the main entry, the last step is to add the player node to an HUD camera.

```
osg::ref_ptr<Player> player = new Player(
  1.0f, 1.0f, "player.png");
player->setMatrix( osg::Matrix::translate(
  40.0f, 5.0f, 0.0f) );
player->setPlayerType( Player::PLAYER_OBJ );

osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0, 80, 0, 30);
hudCamera->addChild( player.get() );
```

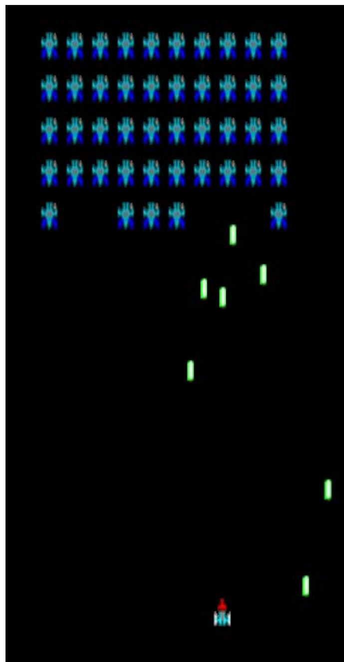
14. And so do the enemies. We will arrange them in a 5 x 10 cavalcade.

```
for ( unsigned int i=0; i<5; ++i )
{
  for ( unsigned int j=0; j<10; ++j )
  {
    osg::ref_ptr<Player> enemy = new Player(1.0f, 1.0f,
      "enemy.png");
    enemy->setMatrix( osg::Matrix::translate(
      20.0f+1.5f*(float)j, 25.0f-1.5f*(float)i, 0.0f) );
    enemy->setPlayerType( Player::ENEMY_OBJ );
    hudCamera->addChild( enemy.get() );
  }
}
```

15. Specify a black background and start the viewer.

```
osgViewer::Viewer viewer;
viewer.getCamera()->setClearColor( osg::Vec4(0.0f, 0.0f,
  0.0f, 1.0f) );
viewer.addEventHandler( new GameControllor(hudCamera.get()) );
viewer.setSceneData( hudCamera.get() );
return viewer.run();
```

16. Good job! Now make sure the PNG files are placed in the executable directory, press *Left arrow* and *Right arrow* buttons to move your fighter to avoid enemies' bullets, and press *Return* to fire. Your goal now is only one: defeat all bad aliens!



How it works...

Let us review the design of this naive game rapidly. A `Player` class is used for representing the player object, the enemies, and the bullets. It can move to a new position, shoot new bullets, and check if it is intersected with others. A global game controller is used to manage all these `Player` nodes and remove unused and destroyed ones. And if you like, you may also add some sentences to show 'you win' or 'you lose' pop ups, and improve the enemies' intelligence. No special functionailities are used except the event handlers and transformation nodes. But these are enough to build a simple game.

Considering more game-related features? Please continue reading the last example in this chapter which introduces integration with physics engines, and the next chapter which discusses rendering effects.

Building a skeleton system

Skeletal animation is important among all kinds of 3D scene animations. It needs a hierarchical set of bones that are connected to each other. Sometimes these bones are also called **rigs**. Animations on one or more bones will finally lead to complex character animations such as walking, running, and even fighting with somebody.

An OSG bone here means a joint at which two parts of the real human can make contact. OSG bones also have hierarchical structures as any of the bone nodes can have one or more children. To represent a bone's shape, it is always suggested to push an additional mesh to the parent. That is because the mesh and the bone are in the same coordinate frame, so they can be actually binded together to perform both the rendering and logic operations of a part of the complete skeleton.

We can directly add a certain model along with a bone so that it will follow the bone's translation and rotation. This can be used to produce some robot-like animations or simulate cartoon characters. A real human or animal has skin and muscles over the bones. The deformation of these muscles, or meshes in 3D developments, can be treated as the basis of real-character animations. To achieve this, we have to bind the mesh vertices to bones, and transform them according to bone's motion and vertex weights. The first implementation which doesn't have a skinning process will be demonstrated in this section. And a simple skinning work will be shown in the next one.

How to do it...

Let us start.

1. Include necessary headers:

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

```
#include <osgAnimation/Bone>
#include <osgAnimation/Skeleton>
#include <osgAnimation/UpdateBone>
#include <osgAnimation/StackedTranslateElement>
#include <osgAnimation/StackedQuaternionElement>
#include <osgAnimation/BasicAnimationManager>
#include <osgViewer/Viewer>
```

2. We create the bone shape by drawing a line from parent bone's original point (which is also the start point of current bone), to the end point of current bone. All will be done in parent bone's local coordinate system, so the generated `osg::Geode` node will be added to that bone too.

```
osg::Geode* createBoneShape( const osg::Vec3& trans,
  const osg::Vec4& color )
{
  osg::ref_ptr<osg::Vec3Array> va = new osg::Vec3Array;
  va->push_back( osg::Vec3() ); va->push_back( trans );
  osg::ref_ptr<osg::Vec4Array> ca = new osg::Vec4Array;
  ca->push_back( color );

  osg::ref_ptr<osg::Geometry> line = new osg::Geometry;
  line->setVertexArray( va.get() );
  line->setColorArray( ca.get() );
  line->setColorBinding( osg::Geometry::BIND_OVERALL );
  line->addPrimitiveSet( new osg::DrawArrays(GL_LINES, 0, 2) );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( line.get() );
  geode->getOrCreateStateSet()->setAttributeAndModes(
    new osg::LineWidth(15.0f) );
  geode->getOrCreateStateSet()->setMode( GL_LIGHTING,
    osg::StateAttribute::OFF );
  return geode.release();
}
```

3. Create the `osgAnimation::Bone` node and add it to the parent bone with an appropriate offset (`trans`). We will create translation and rotation animations for each bone, so there should be an `osgAnimation::UpdateBone` callback which records initial animation values. And to place the bone in its parent's local coordinates, we must consider its current matrix in the skeleton space while using `setMatrixInSkeletonSpace()` to apply the offset.

```
osgAnimation::Bone* createBone( const char* name,
  const osg::Vec3& trans, osg::Group* parent )
{
```

```
    osg::ref_ptr<osgAnimation::Bone> bone =
      new osgAnimation::Bone;
    parent->insertChild( 0, bone.get() );
    parent->addChild( createBoneShape(trans, osg::Vec4(
      1.0f, 1.0f, 1.0f, 1.0f)) );

    osg::ref_ptr<osgAnimation::UpdateBone> updater =
      new osgAnimation::UpdateBone(name);
    updater->getStackedTransforms().push_back( new
      osgAnimation::StackedTranslateElement("translate", trans) );
    updater->getStackedTransforms().push_back( new
      osgAnimation::StackedQuaternionElement("quaternion") );

    bone->setUpdateCallback( updater.get() );
    bone->setMatrixInSkeletonSpace( osg::Matrix::translate(trans)
      * bone->getMatrixInSkeletonSpace() );
    bone->setName( name );
    return bone.get();
  }
```

4. While creating leaf bones of the skeleton, we add an additional shape node as the leaf bone's child. This is in fact not the leaf bone's own shape, but its child shape which will accept the leaf bone's animation and, thus, have transformations in the 3D world. This is the reason we must have an independent function to create these 'end bones' and their child shapes.

```
osgAnimation::Bone* createEndBone( const char* name,
  const osg::Vec3& trans, osg::Group* parent )
{
  osgAnimation::Bone* bone = createBone( name, trans, parent );
  bone->addChild( createBoneShape(trans, osg::Vec4(
    0.4f, 1.0f, 0.4f, 1.0f)) );
  return bone;
}
```

5. The `createChannel()` function will be used later to add rotation animations to bones.

```
osgAnimation::Channel* createChannel( const char* name,
  const osg::Vec3& axis, float rad )
{
  osg::ref_ptr<osgAnimation::QuatSphericalLinearChannel> ch =
    new osgAnimation::QuatSphericalLinearChannel;
  ch->setName( "quaternion" );
  ch->setTargetName( name );
```

```
  osgAnimation::QuatKeyframeContainer* kfs =
    ch->getOrCreateSampler()->getOrCreateKeyframeContainer();
  kfs->push_back( osgAnimation::QuatKeyframe(
    0.0, osg::Quat(0.0, axis)) );
  kfs->push_back( osgAnimation::QuatKeyframe(
    8.0, osg::Quat(rad, axis)) );
  return ch.release();
}
```

6. In the main entry, we first create the skeleton root and child bones. They are both OSG nodes so there is no difference in maintaining bones and normal OSG nodes.

```
osg::ref_ptr<osgAnimation::Skeleton> skelroot =
  new osgAnimation::Skeleton;
skelroot->setDefaultUpdateCallback();

// Here the name 'bone0' means the root.
// And 'bone1*' are bones of the next level.
// The rest (bone2* - bone4*) may be deduced by analogy
// and found in the source code package
osgAnimation::Bone* bone0 = createBone( "bone0",
  osg::Vec3(0.0f,0.0f,0.0f), skelroot.get() );
osgAnimation::Bone* bone11 = createBone( "bone11",
  osg::Vec3(0.5f,0.0f,0.0f), bone0 );
osgAnimation::Bone* bone12 = createEndBone( "bone12",
  osg::Vec3(1.0f,0.0f,0.0f), bone11 );
...
```

These code segments create a mechanical hand with four claws.

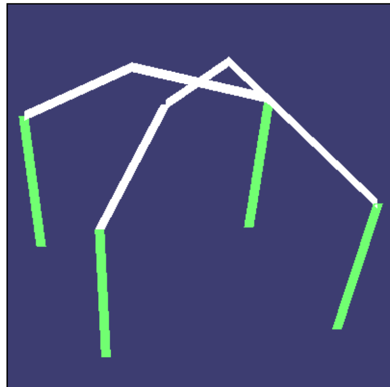
7. Next we will create animations on different claw bones. This gives the hand a clasping action as an animation. Register the animation.

```
osg::ref_ptr<osgAnimation::Animation> anim =
  new osgAnimation::Animation;
anim->setPlayMode( osgAnimation::Animation::PPONG );
anim->addChannel( createChannel("bone11", osg::Y_AXIS,
  osg::PI_4) );
anim->addChannel( createChannel("bone12", osg::Y_AXIS,
  osg::PI_2) );
...// Find rest channel settings in the source code

osg::ref_ptr<osgAnimation::BasicAnimationManager> manager =
  new osgAnimation::BasicAnimationManager;
manager->registerAnimation( anim.get() );
manager->playAnimation( anim.get() );
```

8. Add them to the root node and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( skelroot.get() );
root->setUpdateCallback( manager.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

9. You will see a mechanical hand rendered with lines in the space after starting the example. It will simulate the action of grabbing something with the four claws and then loosening. Bones are bound to their parents, so they won't break while the animation is running in ping-pong mode. Although it is a little too simple for a real application, it is a kind of character animation anyhow, and it can be extended to work with some very complex situations such as the human kinematics.



How it works...

Skeleton animation uses the same osgAnimation framework as we discussed a while ago. A complete application with bone animations still needs to have an animation manager and several channels storing the animating data. It also has a sub-scene graph formed by parent and child bones. The root bone (`bone0` in this recipe) is linked to the skeleton node. All other bones are its direct and indirect children. No orphan bone is allowed except the root one. A bone must have a callback (`osgAnimation::UpdateBone`) who records stacked elements to be associated with channels targeting them.

Be careful of the method `setMatrixInSkeletonSpace()`. It sets the bone in the skeleton space. So if you want to specify the bone in its parent bone's space (which is easier to understand here), you have to convert the offset matrix to skeleton matrix first. This requires you to ensure to first add the new bone into the skeleton scene graph; otherwise you will not be able to get a correct bone matrix for computation. The transformation of the bone can be easily written as the product of the new offset and the previous skeleton space matrix (which equals the parent bone's matrix initially).

```
bone->setMatrixInSkeletonSpace(
  osg::Matrix::translate(trans) *
    bone->getMatrixInSkeletonSpace() );
```

Another thing to note is, in order to add renderables to represent each bone's shape, we have to push the geometry node to the end of the parent bone's child list, and insert the bone node at the beginning of the list. See the following code snippet:

```
osg::ref_ptr<osgAnimation::Bone> bone =
  new osgAnimation::Bone;
parent->insertChild( 0, bone.get() );
parent->addChild( createBoneShape(trans, osg::Vec4(
  1.0f, 1.0f, 1.0f, 1.0f)) );
```

To explain the reason in short: This ensures all bones will be traversed and updated before any geometry is drawn. Just treat this as a rule to follow.

Skinning a customized mesh

Let us continue the last recipe which creates a simple skeleton for representing a mechanical hand with four claws. This time we will do the skinning work, that is, bind vertices of the character geometry with bones. Each bone can be associated with some portion of the vertices, and each vertex can be associated with multiple bones, each with a weight factor which will change the effect of the bone on the vertex. You may find some detailed information in the following link, as well as some shader code:

`http://tech-artists.org/wiki/Vertex_Skinning`

To calculate the final position of one vertex, we must collect all bones associated with it and apply each bone's transformation matrix to the vertex's position, as well as scale the matrix by corresponding weight. Fortunately, OSG does everything we have just described for us. The only thing we have to do is to build a map of the vertex and its associated bone name and weight. We will work on the last example code to add such implementations.

How to do it...

Let us start.

1. First, we don't need the `createBoneShape()` function anymore. Because we are not going to draw the bone directly in the 3D world this time. In the `createBone()` and `createEndBone()` functions, remove the line that adds the return value of `createBoneShape()` to the parent node.

```
osgAnimation::Bone* createBone( const char* name,
  const osg::Vec3& trans, osg::Group* parent )
{
  osg::ref_ptr<osgAnimation::Bone> bone =
    new osgAnimation::Bone;
  parent->insertChild( 0, bone.get() );

  osg::ref_ptr<osgAnimation::UpdateBone> updater =
    new osgAnimation::UpdateBone(name);
  updater->getStackedTransforms().push_back( new
    osgAnimation::StackedTranslateElement("translate", trans) );
  updater->getStackedTransforms().push_back( new
    osgAnimation::StackedQuaternionElement("quaternion") );

  bone->setUpdateCallback( updater.get() );
  bone->setMatrixInSkeletonSpace( osg::Matrix::translate(trans)
    * bone->getMatrixInSkeletonSpace() );
  bone->setName( name );
  return bone.get();
}

osgAnimation::Bone* createEndBone( const char* name,
  const osg::Vec3& trans, osg::Group* parent )
{
  osgAnimation::Bone* bone = createBone( name, trans, parent );
  return bone;
}
```

   And there are no changes to the `createChannel()` function.

2. Now we should create two new functions, and use `osgAnimation::RigGeometry` class to record relations between bones and vertices. The `addVertices()` function adds vertices of one claw to the geometry object, and sets up bones which are bound to these newly allocated vertices.

```
void addVertices( const char* name1, float length1,
  const char* name2, float length2,
    const osg::Vec3& dir,    osg::Geometry* geom,
```

```
        osgAnimation::VertexInfluenceMap* vim )
{
  osg::Vec3Array* va = static_cast<osg::Vec3Array*>(
    geom->getVertexArray() );
  unsigned int start = va->size();
  // The bone shape is supposed to have 5 vertices, the first
  // two of which are unmovable
  va->push_back( dir * 0.0f );
  va->push_back( dir * length1 );
  // The last 3 points will be fully controled by the rig
  // geometry so they should be assoicated with the influence
  // map
  va->push_back( dir * length1 );
  (*vim)[name1].push_back(
    osgAnimation::VertexIndexWeight(start+2, 1.0f) );
  va->push_back( dir * length2 );
  (*vim)[name1].push_back(
    osgAnimation::VertexIndexWeight(start+3, 1.0f) );
  va->push_back( dir * length2 );
  (*vim)[name2].push_back(
    osgAnimation::VertexIndexWeight(start+4, 1.0f) );
  // Push the very, very simple shape definition (actually
  // line strips) to the rig geometry
  geom->addPrimitiveSet( new osg::DrawArrays(
    GL_LINE_STRIP, start, 5) );
}
```

3. The `createBoneShapeAndSkin()` function will simultaneously create the geometry of the mechanical hand and finish the skinning work. It uses the `addVertices()` function we just introduced internally.

```
osg::Geode* createBoneShapeAndSkin()
{
  ...
}
```

4. In this function, first we have to initialize the geometry and the `osgAnimation::VertexInfluenceMap` object, which is the association table of all the vertices and bones. Then we allocate the vertices of each claw geometry, assemble them, and bind them to specific bones.

```
osg::ref_ptr<osg::Geometry> geometry = new osg::Geometry;
geometry->setVertexArray( new osg::Vec3Array );

osg::ref_ptr<osgAnimation::VertexInfluenceMap> vim =
  new osgAnimation::VertexInfluenceMap;
```

```
(*vim)["bone11"].setName( "bone11" );
(*vim)["bone12"].setName( "bone12" );
...  // Please find details in the source code

addVertices( "bone11", 0.5f, "bone12", 1.5f, osg::X_AXIS,
  geometry.get(), vim.get() );
...  // Please find details in the source code
```

5. Then we use the `osgAnimation::RigGeometry` object to accept the geometry and the influence map and add it to a node, which will be added to the scene graph later.

```
osg::ref_ptr<osgAnimation::RigGeometry> rigGeom =
  new osgAnimation::RigGeometry;
rigGeom->setSourceGeometry( geometry.get() );
rigGeom->setInfluenceMap( vim.get() );
rigGeom->setUseDisplayList( false );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( rigGeom.get() );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::LineWidth(15.0f) );
geode->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );
return geode.release();
```

6. The only change in the main entry is to add the rigs and vertex binding data to the skeleton node.

```
skelroot->addChild( createBoneShapeAndSkin() );
```

7. OK, now the claws work as they did in the *Building a skeleton system* recipe. But this time it has a solid mesh and you will see that the mesh is deforming along with the bones' animations. The muscle movement may not be realistic at present, as we have only configured simple weights of each bone on vertices, and the number of changeable vertices is not enough for performing precise movements. Exporting models, skeletons, and animations from some other modeling tools may be a good idea if you need more complex characters in the application. Remember that OSG supports characters in **Collada DAE** and **Autodesk FBX** formats currently.

How it works...

In the *Building a skeleton system* recipe, we added a set of line geometries to the bone structure. Each bone had one geometry to represent its shape and animation states. It required bone nodes and geometry nodes to be mixed in one sub-scene graph. But this time in the skinning example, we keep the bone hierarchy unchanged and directly add a geometry node to the skeleton itself. Then we bind each bone with a number of vertices and set the vertex weight. The character updating and rendering work will be done in OSG backend internally. You may use either software or hardware technique to render the character's rig geometry.

There's more...

OSG provides two examples to explain the usage of skeleton animation clearly: The `osganimationskinning` example shows how to build a simple skeleton and skin it; the `osganimationhardware` example describes how to change the transform technique used in `osgAnimation::RigGeometry` (software or hardware).

Last but not least, don't try to build a complete human model and skeleton and animate it by programming. It is possible but needs heavy work to implement. Consider converting your character from some standard formats and commonly-used modeling software. The DAE and FBX formats are good choices for such purposes.

Letting the physics engine be

It is a great enhancement to your applications to integrate with certain physics engines, and, thus, have the ability to compute collisions between scene objects, simulate rigid, soft body, fluid, and cloth behaviours in a virtual physics world. Especially in games, physics support can make the scenario more realistic and interesting, and provide comfortable user interactions and feedbacks.

In the last recipe of this chapter, we are going to see a simple example of physics integration in OSG. It requires the famous **NVIDIA PhysX** library as dependence, and can finally produce an interactive program demonstrating the most common rigid-collision functionality in an OSG world.

Getting ready

You have to first download the PhysX SDK from the NVIDIA developer website. Remember to download version 2.8, as the latest 3.0 version totally changes the API interface and won't work with this recipe.

Visit the following link and register to download:

`http://developer.nvidia.com/physx-downloads`

Then we can configure the CMake script to add PhysX dependence directories and libraries:

```
FIND_PATH(PHYSX_SDK_DIR Physics/include/NxPhysics.h)
FIND_LIBRARY(PHYSX_LIBRARY PhysXLoader.lib libPhysXLoader.so)

SET(EXTERNAL_INCLUDE_DIR
    "${PHYSX_SDK_DIR}/PhysXLoader/include"
    "${PHYSX_SDK_DIR}/Physics/include"
    "${PHYSX_SDK_DIR}/Foundation/include")
TARGET_LINK_LIBRARIES(${EXAMPLE_NAME} ${PHYSX_LIBRARY})
```

How to do it...

Let us start.

1. Include necessary headers:

```
#include <NxPhysics.h>
#include <osg/ShapeDrawable>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. We will create an independent class to manage all PhysX functions. It is designed to use the singleton design pattern and create and manage physics objects with unique ID numbers. It supports creating a few types of rigid bodies and setting their velocities and matrices later with the ID.

```
class PhysXInterface : public osg::Referenced
{
public:
  static PhysXInterface* instance();

  void createWorld( const osg::Plane& plane, const osg::Vec3&
    gravity );
  void createBox( int id, const osg::Vec3& dim, double mass );
  void createSphere( int id, double radius, double mass );

  void setVelocity( int id, const osg::Vec3& pos );
  void setMatrix( int id, const osg::Matrix& matrix );
  osg::Matrix getMatrix( int id );
```

```
    void simulate( double step );

protected:
    PhysXInterface();
    virtual ~PhysXInterface();

    void createActor( int id, NxShapeDesc* shape,
      NxBodyDesc* body );

    typedef std::map<int, NxActor*> ActorMap;
    ActorMap _actors;
    NxPhysicsSDK* _physicsSDK;
    NxScene* _scene;
};
```

3. The `instance()` function returns the only instance of the `PhysXInterface` class.

```
PhysXInterface* PhysXInterface::instance()
{
  static osg::ref_ptr<PhysXInterface> s_registry =
    new PhysXInterface;
  return s_registry.get();
}
```

4. In the constructor, we initialize the `PhysX SDK` object.

```
PhysXInterface::PhysXInterface() : _scene(NULL)
{
  NxPhysicsSDKDesc desc;
  NxSDKCreateError errorCode = NXCE_NO_ERROR;
  _physicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION,
    NULL, NULL, desc, &errorCode);
  if ( !_physicsSDK )
  {
    OSG_WARN << "Unable to initialize the PhysX SDK, error
    code: " << errorCode << std::endl;
  }
}
```

5. And in the destructor, we will release all registered PhysX actors (which are actually objects in the physics world), the `_scene` variable which describes the physics world, and the SDK variable to make sure all PhysX objects are deleted from the memory.

```
PhysXInterface::~PhysXInterface()
{
  if ( _scene )
  {
```

```
    for ( ActorMap::iterator itr=_actors.begin();
    itr!=_actors.end(); ++itr )
    _scene->releaseActor( *(itr->second) );
    _physicsSDK->releaseScene( *_scene );
  }
  NxReleasePhysicsSDK( _physicsSDK );
}
```

6. The `createWorld()` method will allocate a new physics world with specified gravity and material values, and create a static ground object at the same time. The protected `createActor()` will be always called to create new `NxActor` objects and save it to the actor map.

```
void PhysXInterface::createWorld( const osg::Plane& plane,
  const osg::Vec3& gravity )
{
  NxSceneDesc sceneDesc;
  sceneDesc.gravity = NxVec3(gravity.x(), gravity.y(),
    gravity.z());
  _scene = _physicsSDK->createScene( sceneDesc );

  NxMaterial* defaultMaterial =
    _scene->getMaterialFromIndex(0);
  defaultMaterial->setRestitution( 0.5f );
  defaultMaterial->setStaticFriction( 0.5f );
  defaultMaterial->setDynamicFriction( 0.5f );

  // Create the ground plane
  NxPlaneShapeDesc shapeDesc;
  shapeDesc.normal = NxVec3(plane[0], plane[1], plane[2]);
  shapeDesc.d = plane[3];
  createActor( -1, &shapeDesc, NULL );
}
```

7. The `createBox()` method creates a dynamic box object in the world. To note, it doesn't do anything to the rendering result at present. The function only affects the 'physics' world.

```
void PhysXInterface::createBox( int id, const osg::Vec3&
  dim, double mass )
{
  NxBoxShapeDesc shapeDesc; shapeDesc.dimensions =
    NxVec3(dim.x(), dim.y(), dim.z());
  NxBodyDesc bodyDesc; bodyDesc.mass = mass;
  createActor( id, &shapeDesc, &bodyDesc );
}
```

8. The `createSphere()` method will create a dynamic sphere.

```
void PhysXInterface::createSphere( int id, double radius,
  double mass )
{
  NxSphereShapeDesc shapeDesc; shapeDesc.radius = radius;
  NxBodyDesc bodyDesc; bodyDesc.mass = mass;
  createActor( id, &shapeDesc, &bodyDesc );
}
```

9. After a rigid object is created, we can call `setVelocity()` method with the object ID to set the velocity value.

```
void PhysXInterface::setVelocity( int id, const osg::Vec3&
  vec )
{
  NxActor* actor = _actors[id];
  actor->setLinearVelocity( NxVec3(vec.x(), vec.y(), vec.z()) );
}
```

10. We can also set the matrix (the translation and rotation components) value of a created object.

```
void PhysXInterface::setMatrix( int id, const osg::Matrix&
  matrix )
{
  NxF32 d[16];
  for ( int i=0; i<16; ++i )
  d[i] = *(matrix.ptr() + i);
  NxMat34 nxMat; nxMat.setColumnMajor44( &d[0] );

  NxActor* actor = _actors[id];
  actor->setGlobalPose( nxMat );
}
```

11. The `getMatrix()` method is important because it can obtain the latest matrix value of a physics object, which may be moved or smashed during the simulation. The value can be set to related OSG node to ensure that the changes in the physics world can be reflected to the rendering window too.

```
osg::Matrix PhysXInterface::getMatrix( int id )
{
  float mat[16];
  NxActor* actor = _actors[id];
  actor->getGlobalPose().getColumnMajor44( mat );
  return osg::Matrix(&mat[0]);
}
```

12. The `simulate()` method must be executed in every frame to make sure the physics simulation loop is running.

```
void PhysXInterface::simulate( double step )
{
  _scene->simulate( step );
  _scene->flushStream();
  _scene->fetchResults( NX_RIGID_BODY_FINISHED, true );
}
```

13. The internal `createActor()` function will create an actor according to the shape and body descriptions, and push it into the map for future uses.

```
void PhysXInterface::createActor( int id, NxShapeDesc* shape,
  NxBodyDesc* body )
{
  NxActorDesc actorDesc;
  actorDesc.shapes.pushBack( shape );
  actorDesc.body = body;

  NxActor* actor = _scene->createActor( actorDesc );
  _actors[id] = actor;
}
```

14. After completing the PhysX interface class, we will have to establish relationships between the physics elements and scene graph nodes, and update the physics world in every frame. In the `PhysicsUpdater` class, we use a `NodeMap` data type which records physics ID and OSG node in a key-value map. These IDs also correspond to physics actors in the PhysX layer, so we can actually connect the physics and the rendering layer together here.

```
class PhysicsUpdater : public osgGA::GUIEventHandler
{
public:
  PhysicsUpdater( osg::Group* root ) : _root(root) {}

  void addGround( const osg::Vec3& gravity );
  void addPhysicsBox( osg::Box* shape, const osg::Vec3& pos,
    const osg::Vec3& vel, double mass );
  void addPhysicsSphere( osg::Sphere* shape,
    const osg::Vec3& pos, const osg::Vec3& vel, double mass );
  bool handle( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& aa );

protected:
  void addPhysicsData( int id, osg::Shape* shape,
    const osg::Vec3& pos, const osg::Vec3& vel, double mass );
```

```
typedef std::map<int, osg::observer_ptr<
  osg::MatrixTransform> > NodeMap;
NodeMap _physicsNodes;
osg::observer_ptr<osg::Group> _root;
```

15. In the `addGround()` method, we create a huge box with a very small height to simulate the ground, and create the world element in PhysX as well.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( new osg::ShapeDrawable(
  new osg::Box(osg::Vec3(0.0f, 0.0f,-0.5f), 100.0f,
    100.0f, 1.0f)) );

osg::ref_ptr<osg::MatrixTransform> mt =
  new osg::MatrixTransform;
mt->addChild( geode.get() );
_root->addChild( mt.get() );

PhysXInterface::instance()->createWorld( osg::Plane(
  0.0f, 0.0f, 1.0f, 0.0f), gravity );
```

16. In the `addPhysicsBox()` method, we create a physics box actor and use `addPhysicsData()` to register it and create corresponding OSG node in the scene graph.

```
int id = _physicsNodes.size();
PhysXInterface::instance()->createBox(
  id, shape->getHalfLengths(), mass );
addPhysicsData( id, shape, pos, vel, mass );
```

17. In the `addPhysicsSphere()` method, we have similar work to do as in `addPhysicsBox()`.

```
int id = _physicsNodes.size();
PhysXInterface::instance()->createSphere(
  id, shape->getRadius(), mass );
addPhysicsData( id, shape, pos, vel, mass );
```

18. The `handle()` method has two events to handle. If the `FRAME` event comes, it will update the physics world with a delta time value. Then all IDs registered in the `NodeMap` variable will be traversed to retrieve matrix data from the physics element and apply it to scene graph nodes. And if user presses the *Return* key, the updater will create a new dynamic ball at the eye position with an initial velocity. So we can interactively shoot at any other element in the scene and see how they collapse, roll, and fly off.

```
osgViewer::View* view = static_cast<osgViewer::View*>( &aa );
if ( !view || !_root ) return false;
```

```
switch ( ea.getEventType() )
{
  case osgGA::GUIEventAdapter::KEYUP:
  if ( ea.getKey()==osgGA::GUIEventAdapter::KEY_Return )
  {
    osg::Vec3 eye, center, up, dir;
    view->getCamera()->getViewMatrixAsLookAt( eye, center, up );
    dir = center - eye; dir.normalize();
    addPhysicsSphere( new osg::Sphere(osg::Vec3(), 0.5f),
      eye, dir * 60.0f, 2.0 );
  }
  break;
  case osgGA::GUIEventAdapter::FRAME:
  PhysXInterface::instance()->simulate( 0.02 );
  for ( NodeMap::iterator itr=_physicsNodes.begin();
  itr!=_physicsNodes.end(); ++itr )
  {
    osg::Matrix matrix = PhysXInterface::instance()-
      >getMatrix(itr->first);
    itr->second->setMatrix( matrix );
  }
  break;
default: break;
}
return false;
```

19. The protected `addPhysicsData()` method will add a newly allocated OSG node to the root, and set its ID and physics attributes as well.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( new osg::ShapeDrawable(shape) );

osg::ref_ptr<osg::MatrixTransform> mt =
  new osg::MatrixTransform;
mt->addChild( geode.get() );
_root->addChild( mt.get() );

PhysXInterface::instance()->setMatrix(
  id, osg::Matrix::translate(pos) );
PhysXInterface::instance()->setVelocity( id, vel );
_physicsNodes[id] = mt;
```

20. Now we will get into the main entry, first we will create the root node and the updater object.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
osg::ref_ptr<PhysicsUpdater> updater = new PhysicsUpdater(
  root.get() );
```

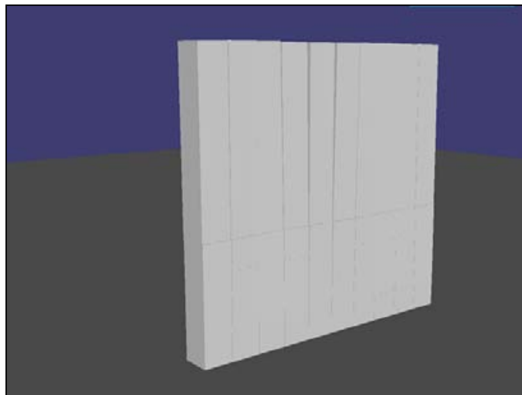
21. And then we create the ground and a wall made up of many small boxes and add them to the updater (and to the `_root` node which is stored in the updater class).

```
updater->addGround( osg::Vec3(0.0f, 0.0f,-9.8f) );
for ( unsigned int i=0; i<10; ++i )
{
  for ( unsigned int j=0; j<10; ++j )
  {
    updater->addPhysicsBox( new osg::Box(osg::Vec3(), 0.99f),
      osg::Vec3((float)i, 0.0f, (float)j+0.5f), osg::Vec3(),
        1.0f );
  }
}
```

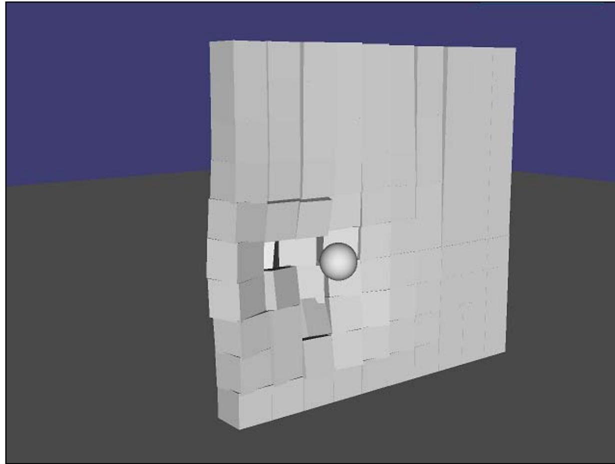
22. Start the viewer at last:

```
osgViewer::Viewer viewer;
viewer.addEventHandler( updater.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

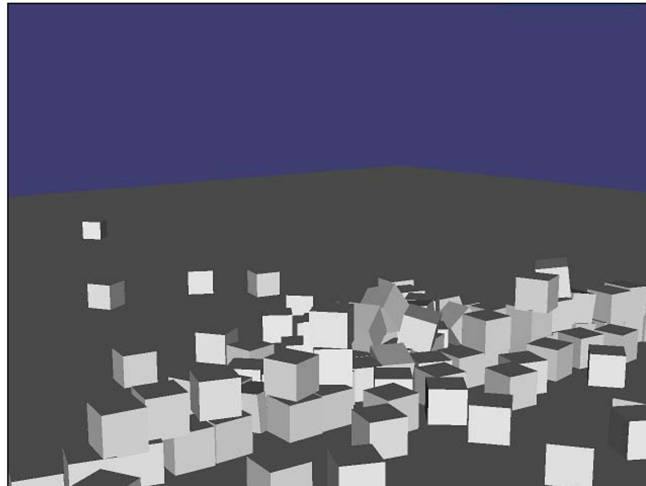
23. Now we can run the application and adjust the view matrix by dragging mouse buttons. First we can see a wall stand on the ground. It is obviously made up of many boxes, and may even shake slightly, but won't fall down without an outside force.



24. Aim with your eye and press the *Return* key to shoot a ball! Can you knock and smash the wall with only one hit?



25. Let us have a look at the ruins we finally created. Are you imagining making an 'Angry Birds' game by yourself?



How it works...

NVIDIA PhysX is a powerful and legible physics library, and so it is easy to embed it into OSG scene. The only data shared by the physics level and the rendering level is the unique object IDs.

The meaning of an ID in the `PhyxXInterface` class is to identify a unique rigid element, for instance, a rigid box, a sphere, or the ground plane. They are not visible on the screen but can be used to carry out physics computation and calculate the resultant positions and rotations.

The same ID in the `PhysicsUpdater` class is used to distinguish scene nodes. They use the same dimensions as the physics elements, but are used for rendering purpose only. This design separates the rendering and physics computing work so that there will be less coupling in the application.

There's more...

You may use the same idea to design your own physics integrations. There are many other choices such as ODE (`http://www.ode.org/`), Bullet Physics (`http://bulletphysics.org/`), Newton (`http://newtondynamics.com/`), and so on. Consider using one or more classes to encapsulate the physics functionalities and don't merge them into the scene graph rashly. A mixure of rendering objects and physics objects in the project may cause confusion for reading and understanding.

6
Designing Creative Effects

In this chapter, we will cover:

- Using the bump mapping technique
- Simulating the view-dependent shadow
- Implementing transparency with multiple passes
- Reading and displaying the depth buffer
- Implementing the night vision effect
- Implementing the depth-of-field effect
- Designing a skybox with the cube map
- Creating a simple water effect
- Creating a piece of cloud
- Customizing the state attribute

Introduction

Now we come to the chapter of effects. In the earlier days of 3D programming based on OpenGL, the final results of an effect were based on the use of state attributes and modes. Even the best developers may have various difficulties to create a realistic enough effect. Ray tracing could be a good choice for light computation but it hardly works in real-time environments. And the limited number of fixed pipeline attributes is also the bottleneck for the development of real-time rendering.

The birth of programmable pipeline changed everything. Modern rendering techniques depend heavily on shaders and can produce infinite types of effects. People can design their own vertex, primitive assemble, tessellation, and pixel-computing solutions freely. Some other advanced methods, such as post processing, deferred shading, deferred lighting, and the latest global illumination techniques, were also developed in a very rapid way.

We can't discuss all these topics in one chapter or even one book. But we will talk more about shaders and some popular post effects here, as they always provide good enough results. Some common landscapes, such as the sky, cloud, reflection of water, and shadows, will also be introduced in the form of recipes. Before that, we will add two new functions to the `osgCookBook` namespace in the common sub-directory. First one is the `createRTTCamera()` function, which creates an `osg::Camera` node for rendering-to-texture operations:

```
osg::Camera* createRTTCamera( osg::Camera::BufferComponent
  buffer, osg::Texture* tex, bool isAbsolute )
{
  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  camera->setClearColor( osg::Vec4() );
  camera->setClearMask(
    GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT );
  camera->setRenderTargetImplementation(
    osg::Camera::FRAME_BUFFER_OBJECT );
  camera->setRenderOrder( osg::Camera::PRE_RENDER );
  if ( tex )
  {
    tex->setFilter( osg::Texture2D::MIN_FILTER,
      osg::Texture2D::LINEAR );
    tex->setFilter( osg::Texture2D::MAG_FILTER,
      osg::Texture2D::LINEAR );
    camera->setViewport( 0, 0, tex->getTextureWidth(),
      tex->getTextureHeight() );
    camera->attach( buffer, tex );
  }

  if ( isAbsolute )
  {
    camera->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
    camera->setProjectionMatrix( osg::Matrix::ortho2D(
      0.0, 1.0, 0.0, 1.0) );
    camera->setViewMatrix( osg::Matrix::identity() );
    camera->addChild( createScreenQuad(1.0f, 1.0f) );
  }
  return camera.release();
}
```

And the `createScreenQuad()` function will be used for post-processing work in some of the recipes.

```
osg::Geode* createScreenQuad( float width, float height,
  float scale )
{
  osg::Geometry* geom = osg::createTexturedQuadGeometry(
    osg::Vec3(), osg::Vec3(width,0.0f,0.0f),
      osg::Vec3(0.0f,height,0.0f),
        0.0f, 0.0f, width*scale, height*scale );
  osg::ref_ptr<osg::Geode> quad = new osg::Geode;
  quad->addDrawable( geom );

  int values = osg::StateAttribute::OFF|
    osg::StateAttribute::PROTECTED;
  quad->getOrCreateStateSet()->setAttribute(
    new osg::PolygonMode(osg::PolygonMode::FRONT_AND_BACK,
      osg::PolygonMode::FILL), values );
  quad->getOrCreateStateSet()->setMode( GL_LIGHTING, values );
  return quad.release();
}
```

Using the bump mapping technique

Bump mapping is a per-pixel lighting technique that simulates bumpy surfaces of 3D objects. Instead of creating many new triangles to actually change the geometry, bump mapping uses the surface normals for calculating the bumps and wrinkles during lighting calculations.

A normal map with all normal vectors in tangent space is always required for a common bump-mapping implementation. It may turn bluish because normals are often directed along the Z axis in tangent space. Two other important vectors here are tangent and binormal, which can be inputted as vertex attributes and used for converting light directions into tangent space.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Program>
#include <osg/Texture2D>
#include <osg/ShapeDrawable>
#include <osg/Geode>
```

```
#include <osgDB/ReadFile>
#include <osgUtil/TangentSpaceGenerator>
#include <osgViewer/Viewer>
```

2. In the vertex shader code, we will try to read some vertex attributes from the user application and set up the light direction for use in the bump-mapping model.

```
static const char* vertSource = {
  "attribute vec3 tangent;\n"
  "attribute vec3 binormal;\n"
  "varying vec3 lightDir;\n"
  "void main()\n"
  "{\n"
    ...  // Please find details in the source code
  "}\n"
};
```

3. In the fragment shader, we will operate on two textures: a color one and a normal one. The second map will provide us the normal vector for computing the correct light value.

```
static const char* fragSource = {
  "uniform sampler2D colorTex;\n"
  "uniform sampler2D normalTex;\n"
  "varying vec3 lightDir;\n"
  "void main (void)\n"
  "{\n"
    ...  // Please find details in the source code
  "}\n"
};
```

4. The vertex attributes, including the tangents and binormals, are often not set in pre-defined models. So we have to traverse the loaded model to apply these attributes to each drawable. The `osgUtil::TangentSpaceGenerator` class is a great tool for automatically computing and generating them.

```
class ComputeTangentVisitor : public osg::NodeVisitor
{
public:
  void apply( osg::Node& node ) { traverse(node); }

  void apply( osg::Geode& node )
  {
    for ( unsigned int i=0; i<node.getNumDrawables(); ++i )
    {
      osg::Geometry* geom = dynamic_cast<osg::Geometry*>(
        node.getDrawable(i) );
```

```
      if ( geom ) generateTangentArray( geom );
    }
    traverse( node );
  }

  void generateTangentArray( osg::Geometry* geom )
  {
    osg::ref_ptr<osgUtil::TangentSpaceGenerator> tsg =
      new osgUtil::TangentSpaceGenerator;
    tsg->generate( geom );
    geom->setVertexAttribArray( 6, tsg->getTangentArray() );
    geom->setVertexAttribBinding(
      6, osg::Geometry::BIND_PER_VERTEX );
    geom->setVertexAttribArray( 7, tsg->getBinormalArray() );
    geom->setVertexAttribBinding(
      7, osg::Geometry::BIND_PER_VERTEX );
  }
};
```

5. In the main entry, we will load a model that already has normals and texture coordinates for texturing (for example, `skydome.osgt`), and add tangents and binormals to it.

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene = osgDB::readNodeFiles(
  arguments );
if ( !scene ) scene = osgDB::readNodeFile("skydome.osgt");

ComputeTangentVisitor ctv;
ctv.setTraversalMode( osg::NodeVisitor::TRAVERSE_ALL_CHILDREN );
scene->accept( ctv );
```

6. Create the new shader state attribute and add attribute variables. The names must fit the ones in the shader code, and the attribute numbers must be the same as we set in the geometries.

```
osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertSource) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  fragSource) );
program->addBindAttribLocation( "tangent", 6 );
program->addBindAttribLocation( "binormal", 7 );
```

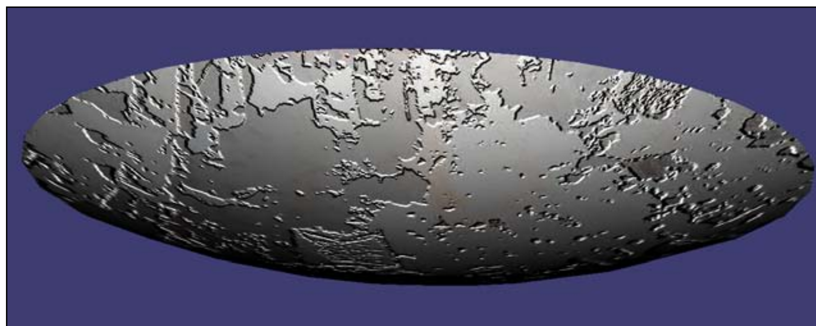
7. Add two images (the color and the normal map) and the program object to the model's state set. We will set an OVERRIDE mask to overwrite the child nodes' texture settings if any.

```
osg::ref_ptr<osg::Texture2D> colorTex = new osg::Texture2D;
colorTex->setImage( osgDB::readImageFile(
  "Images/whitemetal_diffuse.jpg") );
osg::ref_ptr<osg::Texture2D> normalTex =
  new osg::Texture2D;
normalTex->setImage( osgDB::readImageFile(
  "Images/whitemetal_normal.jpg") );

osg::StateSet* stateset = scene->getOrCreateStateSet();
stateset->addUniform( new osg::Uniform("colorTex", 0) );
stateset->addUniform( new osg::Uniform("normalTex", 1) );
stateset->setAttributeAndModes( program.get() );

osg::StateAttribute::GLModeValue value =
  osg::StateAttribute::ON|osg::StateAttribute::OVERRIDE;
stateset->setTextureAttributeAndModes( 0, colorTex.get(),
  value );
stateset->setTextureAttributeAndModes( 1, normalTex.get(),
  value );
```

8. Start the viewer at last.

```
osgViewer::Viewer viewer;
viewer.setSceneData( scene.get() );
return viewer.run();
```

9. Now you will see a dirty and mottled dome instead of the original one. These bumps are not really meant to modify the surface vertices and primitives, but are generated by specifying suitable brightness, color, and shadow values. The dome itself is still as smooth as before.



How it works...

The tangent and binormal arrays can be automatically computed using the `osgUtil::TangentSpaceGenerator` tool, unless you don't have normals specified. These two new vertex attributes are set to index 6 and 7, and then used in the vertex shader for constructing the tangent space's transformation matrix. Here tangents and binormals are actually user-defined attributes as they don't contribute to fixed pipeline rendering, so there are always certain limitations while specifying the attribute index. For example, on most NVIDIA devices, the following indices are already reserved for built-in GLSL attributes:

Type	Index	Built-in attribute name
Position	0	`gl_Vertex`
Normal	2	`gl_Normal`
Color	3	`gl_Color`
Secondary color	4	`gl_SecondaryColor`
Fog coordinate	5	`gl_FogCoord`
Texture coordinate (0-7)	8 - 15	`gl_MultiTexCoord0` to `gl_MultiTexCoord7`

So it becomes clear that we can only use index 1, 6, and 7 for customized attributes. That is exactly what we have done in this example.

Simulating the view-dependent shadow

Shadows are common in modern 3D applications. There are plenty of shadow techniques including shadow map, shadow volume, and many other modifications and improvements based on these basic methods. OSG provide an **osgShadow** library for implementing different kinds of shadow techniques and integrating them with the scene graph.

In the book "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, I have already introduced the common shadow framework. So in this recipe, we will focus on making use of the view-dependent shadows, which depends on the view frustum and will not be too costly to obtain and render shadows, especially when scene data and camera change dynamically.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgShadow/ShadowedScene>
#include <osgShadow/ViewDependentShadowMap>
#include <osgGA/TrackballManipulator>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
```

2. The basic structure of the view-dependent example is nearly the same as the common shadow framework. So the receiver's mask and the caster's mask must be preset properly.

```
unsigned int rcvShadowMask = 0x1;
unsigned int castShadowMask = 0x2;
```

3. Add the ground node which only receives shadow.

```
osg::ref_ptr<osg::MatrixTransform> groundNode =
  new osg::MatrixTransform;
groundNode->addChild( osgDB::readNodeFile("lz.osg") );
groundNode->setMatrix( osg::Matrix::translate(
  200.0f, 200.0f,-200.0f) );
groundNode->setNodeMask( rcvShadowMask );
```

4. Add the cessna node which will cast shadow on the ground.

```
osg::ref_ptr<osg::MatrixTransform> cessnaNode =
  new osg::MatrixTransform;
cessnaNode->addChild( osgDB::readNodeFile(
  "cessna.osg.0,0,90.rot") );
cessnaNode->addUpdateCallback(
  osgCookBook::createAnimationPathCallback(50.0f, 6.0f) );
cessnaNode->setNodeMask( castShadowMask );
```

5. Set up the view-dependent shadow technique, and add the technique as well as all shadowed nodes to the shadow root. We are going to provide as many as possible Cessna instances so that we can immediately benefit from the use of view-dependent technique.

```
osg::ref_ptr<osgShadow::ViewDependentShadowMap> vdsm =
  new osgShadow::ViewDependentShadowMap;
//vdsm->setShadowMapProjectionHint(
  //osgShadow::ViewDependentShadowMap::ORTHOGRAPHIC_SHADOW_MAP );
```

```
//vdsm->setBaseShadowTextureUnit( 1 );

osg::ref_ptr<osgShadow::ShadowedScene> shadowRoot =
  new osgShadow::ShadowedScene;
shadowRoot->setShadowTechnique( vdsm.get() );
shadowRoot->setReceivesShadowTraversalMask( rcvShadowMask );
shadowRoot->setCastsShadowTraversalMask( castShadowMask );

shadowRoot->addChild( groundNode.get() );
for ( unsigned int i=0; i<10; ++i )
{
  for ( unsigned int j=0; j<10; ++j )
  {
    osg::ref_ptr<osg::MatrixTransform> cessnaInstance =
      new osg::MatrixTransform;
    cessnaInstance->setMatrix( osg::Matrix::translate(
      (float)i*50.0f, (float)j*50.0f, 0.0f) );
    cessnaInstance->addChild( cessnaNode.get() );
    shadowRoot->addChild( cessnaInstance.get() );
  }
}
```

6. Compute an appropriate initial camera position and start the viewer.

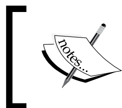
```
const osg::BoundingSphere& bs = groundNode->getBound();
osg::ref_ptr<osgGA::TrackballManipulator> trackball =
  new osgGA::TrackballManipulator;
trackball->setHomePosition( bs.center()+osg::Vec3(
  0.0f, 0.0f, bs.radius()*0.4f), bs.center(), osg::Y_AXIS );

osgViewer::Viewer viewer;
viewer.setCameraManipulator( trackball.get() );
viewer.setSceneData( shadowRoot.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

When the scene is initialized, you will find the camera very near to the ground and the Cessna cluster's shadows are rendered smoothly, as shown in the following screenshot:

7. Zoom out the camera and try to view the whole scene containing all 100 aircraft! Now the frame rate drops down sharply, maybe only 4-5 fps because of the burden of re-computation.

How it works...

The basis of view-dependent shadow rendering has no difference with the osgShadow framework. It requires an `osgShadow::ShadowedScene` node to accept a specific technique and all child scene nodes, including shadow receivers and casters (some receivers may also be casters too). Too many receivers or casters will lead to low performance because of too many computations. But view-dependent techniques can slightly reduce the problem in particular when we are handling shadows in a huge scene. Shadow interactors (eight receivers or casters) will just be ignored if they and their assumed shadows are not visible in the view frustum. You can simply change the recipe to replace `osgShadow::ViewDependentShadowMap` with `osgShadow::ShadowMap`; and you can find that the latter will always work in a very low frame rate, no matter where the viewer is and how many shadows he could see.



The `osgShadow::ViewDependentShadowMap` class is introduced in the latest OSG trunk (version 3.1.0). Please update to a proper version before you try this example.

Implementing transparency with multiple passes

The osgFX library provides a framework for the purpose of multi-pass rendering. Every sub-graph that you want to be rendered through multiple passes should be added to an `osgFX::Effect` node, in which a multi-pass technique is defined and used. You may have already been familiar with some pre-defined effects such as `osgFX::Scribe` and `osgFX::Outline`, but in this recipe, our task is to design a multi-pass technique by ourselves. It is so called multi-pass transparency, which can eliminate the errors while drawing complex objects in transparent mode.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/BlendFunc>
#include <osg/ColorMask>
#include <osg/Depth>
#include <osg/Material>
#include <osgDB/ReadFile>
#include <osgFX/Effect>
#include <osgViewer/Viewer>
```

2. We will first provide a new technique derived from `osgFX::Technique` node. The `validate()` method is used to check if current hardware supports this technique and returns true if everything is OK.

```
class TransparencyTechnique : public osgFX::Technique
{
public:
  TransparencyTechnique() : osgFX::Technique() {}
  virtual bool validate( osg::State& ss ) const
  {
    return true;
  }

protected:
  virtual void define_passes();
};
```

3. Another must-have method in the class is `define_passes()`. It is used for defining multiple passes. In this recipe, we will have two passes: the first one disables the color mask and records the depth buffer value if it is less than the current one; and the second one will enable the usage of the color buffer but only to write to it when the depth value equals to the recorded one.

```
osg::ref_ptr<osg::StateSet> ss = new osg::StateSet;
ss->setAttributeAndModes( new osg::ColorMask(
  false, false, false, false) );
ss->setAttributeAndModes( new osg::Depth(osg::Depth::LESS) );
addPass( ss.get() );

ss = new osg::StateSet;
ss->setAttributeAndModes( new osg::ColorMask(
  true, true, true, true) );
ss->setAttributeAndModes( new osg::Depth(osg::Depth::EQUAL) );
addPass( ss.get() );
```

4. After we complete designing the technique, we can now declare the effect class and add the technique to it in the `define_techniques()` method. The `META_Effect` macro here is used to define basic methods for the effect: the library name, class name, author name, and description.

```
class TransparencyNode : public osgFX::Effect
{
public:
  TransparencyNode() : osgFX::Effect() {}
  TransparencyNode( const TransparencyNode& copy,
    const osg::CopyOp op=osg::CopyOp::SHALLOW_COPY )
  :  osgFX::Effect(copy, op) {}
```

```
    META_Effect( osgFX, TransparencyNode, "TransparencyNode",
      "", "" );

protected:
    virtual bool define_techniques()
    {
      addTechnique(new TransparencyTechnique);
      return true;
    }
};
```

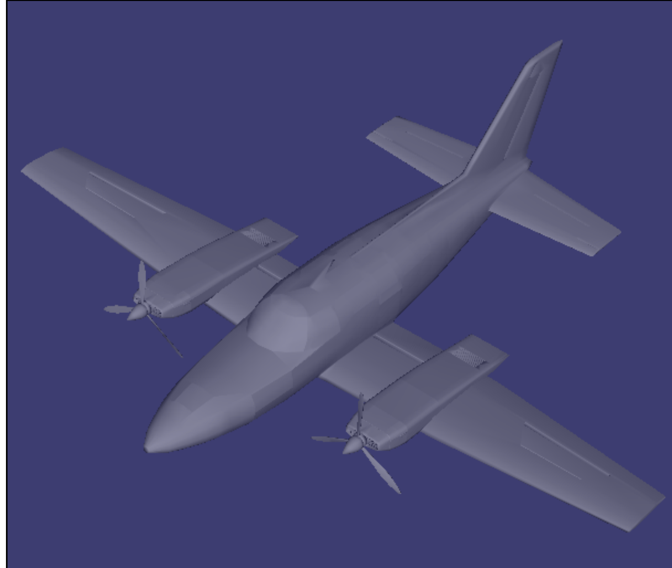
5. In the main entry, we will work on the classic Cessna model. But to make it transparent, we will add a new material to it, set the alpha channel of the diffuse color to a value less than 1, and apply the `TRANSPARENT_BIN` hint to the state set.

```
osg::Node* loadedModel = osgDB::readNodeFile( "cessna.osg" );

osg::ref_ptr<osg::Material> material = new osg::Material;
material->setAmbient( osg::Material::FRONT_AND_BACK,
  osg::Vec4(0.0f, 0.0f, 0.0f, 1.0f) );
material->setDiffuse( osg::Material::FRONT_AND_BACK,
  osg::Vec4(1.0f, 1.0f, 1.0f, 0.5f) );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  material.get(), osg::StateAttribute::ON|osg::StateAttribute::OVE
RRIDE );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  new osg::BlendFunc );
loadedModel->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
```

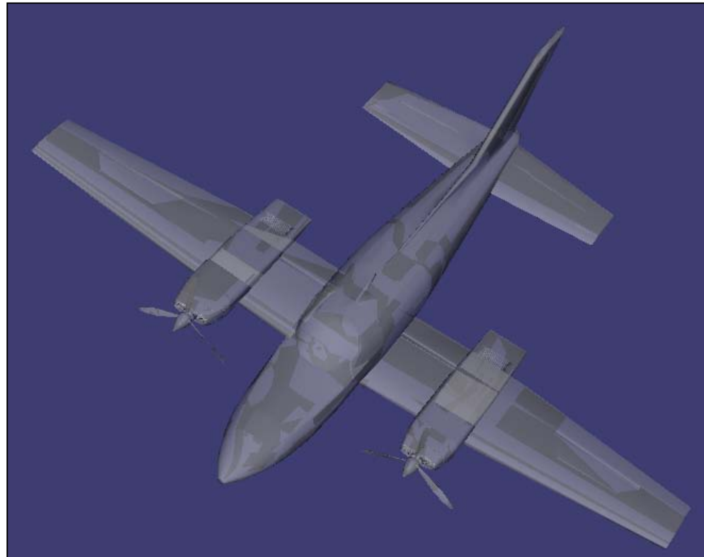
6. Create an instance of the newly-defined class and add the model.

```
osg::ref_ptr<TransparencyNode> fxNode = new TransparencyNode;
fxNode->addChild( loadedModel );
Start the viewer now.
osgViewer::Viewer viewer;
viewer.setSceneData( fxNode.get() );
return viewer.run();
```

7. Now you will see the transparent Cessna rendered properly.



8. If you still remember the fade-in and out example in *Chapter 5*, you will soon realize the advantage of this multi-pass transparency technique: it removes the unnatural mottled blocks on the Cessna without changing the drawing sequence of polygons. Directly set the `loadedModel` node to the viewer and recompile the example. This time you can see errors on the Cessna body, as a result of wrong blending orders.



How it works...

The multi-pass transparency technique will draw the object twice. First pass, as shown in the following block of code, is to update only the depth buffer and find polygons in front of any others:

```
osg::ref_ptr<osg::StateSet> ss = new osg::StateSet;
ss->setAttributeAndModes( new osg::ColorMask(
  false, false, false, false) );
ss->setAttributeAndModes( new osg::Depth(osg::Depth::LESS) );
addPass( ss.get() );
```

The second pass will draw into the color buffer, but because of the depth values set in the first one, only front polygons can pass the depth test and their colors will be drawn and blended with current ones. This avoids the order problem we have discussed before. The code segments are shown here:

```
ss = new osg::StateSet;
ss->setAttributeAndModes( new osg::ColorMask(
  true, true, true, true) );
ss->setAttributeAndModes( new osg::Depth(osg::Depth::EQUAL) );
addPass( ss.get() );
```

There's more...

But is this solution perfect? Of course not, a translucent Cessna should not be rendered in this manner. A translucent model's back faces may also be visible from the viewer, using alpha blending with the front polygons occluding these back faces. In that case, the rendering order of each polygon must be considered before we really put the vertex and primitive data into the OpenGL pipeline. Unfortunately it is nearly impossible for a complex model to do such a sorting work in every frame. **Depth peeling** may solve the problem in another way, and we will talk about it in the last chapter.

Reading and displaying the depth buffer

Rendering-to-texture (**RTT**) is a modern technique that can be used in different fields. It means to draw a scene data to a user-defined target instead of just to the screen. The render target can be a texture or an image object, and even an off-screen render buffer with the **FBO** (**frame buffer object**) extension.

In this chapter, we will use RTT to demonstrate the implementations of post-processing and easily deferred shading techniques within more than one featured example. The common function `osgCookBook::createRTTCamera()` will be heavily used for allocating a new camera and redraw the color or depth of its child scene to a texture. In this beginning recipe, we will first show how to read and display the depth buffer values with a simple post-processing framework.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Texture2D>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. First we load a model to construct the scene.

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene =
  osgDB::readNodeFiles( arguments );
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

3. We must allocate an empty texture by specifying its size for RTT operation. As we are going to handle the depth buffer, we have to set up correct texture parameters here (`GL_DEPTH_COMPONENT24` for internal format and `GL_DEPTH_COMPONENT` for source format).

```
osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
tex2D->setTextureSize( 1024, 1024 );
tex2D->setInternalFormat( GL_DEPTH_COMPONENT24 );
tex2D->setSourceFormat( GL_DEPTH_COMPONENT );
tex2D->setSourceType( GL_FLOAT );
```

4. We will create a new camera node and apply the required buffer value (`DEPTH_BUFFER`) and the texture object to it. The loaded model (or any other scene objects) will be added to the camera to make it rendered to the texture.

```
osg::ref_ptr<osg::Camera> rttCamera =
  osgCookBook::createRTTCamera(osg::Camera::DEPTH_BUFFER, tex2D.
get());
rttCamera->addChild( scene.get() );
```

5. Create an HUD camera over the main scene, and add a quad shown at the left-hand side of the screen. It will be used to represent the pre-rendered texture that contains the sub-graph we loaded before.

```
osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0.0, 1.0, 0.0, 1.0);
hudCamera->addChild( osgCookBook::createScreenQuad(0.5f, 1.0f) );
hudCamera->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, tex2D.get() );
```

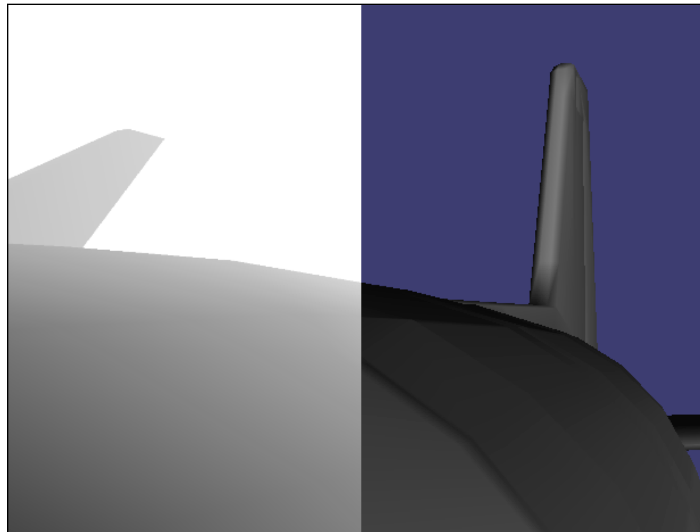
6. Now add all the nodes to the root. To note, we are going to add the model node here too, so that it will be rendered twice: first on the texture, and then on the main camera. Because the HUD quad will only overlay half of the screen, we can now observe the color and depth values of the scene in a comparative view.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( rttCamera.get() );
root->addChild( hudCamera.get() );
root->addChild( scene.get() );
```

7. Before starting the view, we would better disable the automatic near-far computation. That is because the computation changes the near/far plane values all the time and may produce a confusing result when we are trying to retrieve the depth-buffer values.

```
osgViewer::Viewer viewer;
viewer.getCamera()->setComputeNearFarMode(
  osg::CullSettings::DO_NOT_COMPUTE_NEAR_FAR );
viewer.setSceneData( root.get() );
return viewer.run();
```

You will find the depth texture at the left-hand side of the screen. The depth texture looks white at a first glance, but the Cessna part will soon turn dark when you zoom in the scene. It is easy to understand this because the depth always ranges from 0.0 (the nearest) to 1.0 (the farthest). Note that OpenGL always has a non-linear depth buffer so that objects close to the eye will be rendered in greater detail.

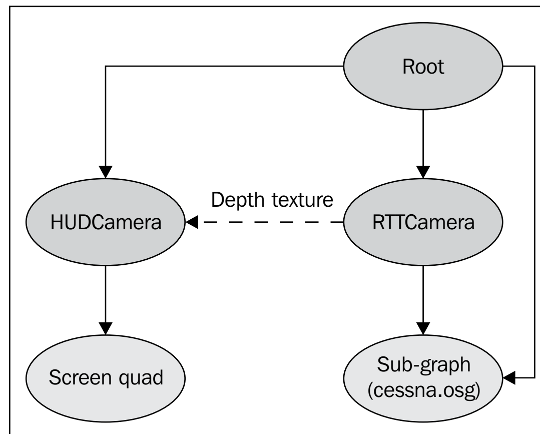


How it works...

In this recipe, we constructed a simple and quick framework for implementing different post-processing effects. The basic concept is listed here:

1. Render the original scene to a texture.
2. Create a quad just filling the entire rendering window and map the texture onto the quad.
3. Now you can enable desired shaders on this quad to make changes to the quad fragments, that is, to make changes to the scene's content.
4. These steps can be repeated if you need more than one shading process; just create another quad to contain the rendering result.
5. Use an HUD camera to render the quad that has the final rendered texture, and show it in full screen to the end users.

The recipe here doesn't show the final texture full screen as it leaves the right half for rendering the original scene (which is linked to the root node as well as the RTT camera node). The scene graph of the reading depth buffer example can be described in the following diagram:



There's more...

In the `osgCookBook::createRTTCamera()` function, we use `FRAME_BUFFER_OBJECT` as the render-target implementation method by default. It means to use OpenGL's FBO extension for the offline rendering work. You can switch to other supported methods if required:

- `FRAME_BUFFER`: Use OpenGL's `glCopySubImage()` function to read buffer values. It can work on early devices that don't support FBO or PBO.

- `PIXEL_BUFFER`: Use OpenGL's **Pixel Buffer Object** (**PBO**) extension for rendering-to-texture support.
- `PIXEL_BUFFER_RTT`: Use the `WGL_ARB_render_texture` extension for rendering-to-texture support.
- `SEPERATE_WINDOW`: Use a separate window for displaying the content in the buffer. It is sometimes useful for debugging.

Implementing the night vision effect

Now we will use the post-processing framework to implement a practical effect called **night vision**. It means to see in very low-light conditions, which is commonly used by military forces. Because humans have poor night vision compared to other animals, they often use special devices with large lenses to gather and concentrate light, for example, night vision goggles.

The reflection of the **Light Interference Filters** (**LIF**) will make the scene in the lenses green. And images from these devices also tend to have some noise and blur. To design such an effect with shaders, we have to consider about these factors and merge the original scene color with a noise map, and then lighten the green parts to simulate the real conditions.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Texture2D>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The vertex shader will be used after the main scene is rendered to texture. It simply transfers the positions and texture coordinates.

```
static const char* vertSource = {
  "void main(void)\n"
  "{\n"
  "gl_Position = ftransform();\n"
  "gl_TexCoord[0] = gl_MultiTexCoord0;\n"
  "}\n"
};
```

3. The fragment shader for post processing will implement the real 'night vision' effect using a noise texture. It reads from the noise texture with random values and uses it to add uncertain changes to the final composition.

```
static const char* fragSource = {
  "uniform sampler2D sceneTex;\n"
  "uniform sampler2D noiseTex;\n"
  "uniform float osg_FrameTime;\n"

  "void main(void)\n"
  "{\n"
  //  Get noise value from texture, changing by frame time
  "float factor = osg_FrameTime * 100.0;\n"
  "vec2 uv = vec2(0.4*sin(factor), 0.4*cos(factor));\n"
  "vec3 n = texture2D(noiseTex, (gl_TexCoord[0].st*3.5) +
    uv).rgb;\n"

  // Get scene and compute greyscale intensity
  "vec3 c = texture2D(sceneTex, gl_TexCoord[0].st +
    (n.xy*0.005)).rgb;\n"
  "float lum = dot(vec3(0.30, 0.59, 0.11), c);\n"
  "if (lum < 0.2) c *= 4.0;\n"

  // Mix a 'night vision' color with the RTT texture
  "vec3 finalColor = (c+(n*0.2)) * vec3(0.1,0.95,0.2);\n"
  "gl_FragColor = vec4(finalColor, 1.0);\n"
  "}\n"
};
```

4. We will make a convenient function for reading an image and applying it to a 2D texture which will automatically repeat at the edge.

```
osg::Texture* createTexture2D( const std::string& fileName )
{
  ...  // Please find details in the source code
}
In the main entry, we load the model first.
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene =
  osgDB::readNodeFiles( arguments );
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

5. The creation of the rendering-to-texture camera is similar to *Reading and displaying the depth buffer* recipe. But this time, it will use `GL_RGBA` as the internal format and read from the color buffer.

```
osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
tex2D->setTextureSize( 1024, 1024 );
tex2D->setInternalFormat( GL_RGBA );

osg::ref_ptr<osg::Camera> rttCamera =
  osgCookBook::createRTTCamera(osg::Camera::COLOR_BUFFER, tex2D.
get());
rttCamera->addChild( scene.get() );
```

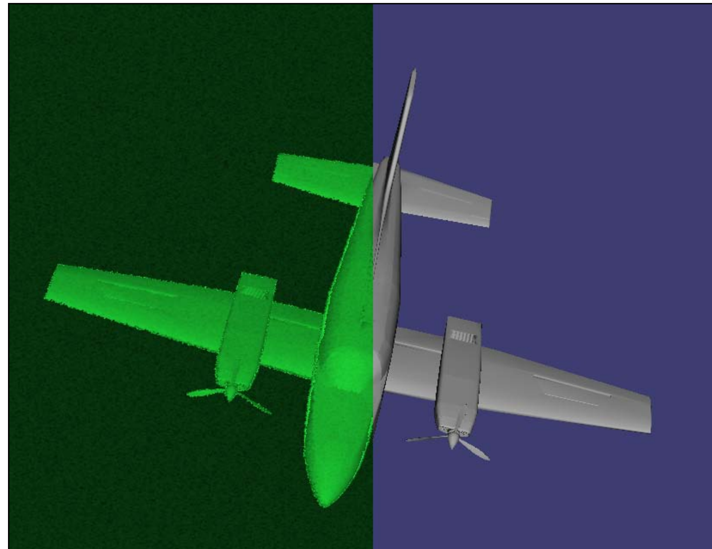
6. Create the HUD quad at the left-hand side of the screen and apply the 'post' shaders, uniforms, and textures to it.

```
osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0.0, 1.0, 0.0, 1.0);
hudCamera->addChild( osgCookBook::createScreenQuad(
  0.5f, 1.0f) );

osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertSource) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  fragSource) );

// Applying the state set to camera will inherit it to the
// child quad, but you may also set the program to the sub-
// graph directly.
osg::StateSet* stateset = hudCamera->getOrCreateStateSet();
stateset->setTextureAttributeAndModes( 0, tex2D.get() );
stateset->setTextureAttributeAndModes(
  1, createTexture2D("noise_tex.jpg") );
stateset->setAttributeAndModes( program.get() );
stateset->addUniform( new osg::Uniform("sceneTex", 0) );
stateset->addUniform( new osg::Uniform("noiseTex", 1) );
```

7. For the last step, we will build the scene graph and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( rttCamera.get() );
root->addChild( hudCamera.get() );
root->addChild( scene.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

Now you can see the scene is applied to the HUD quad with a good night vision effect, as if we are using night glasses to observe the Cessna at night. You can make this recipe look even better by adding a mask uniform that simulates the circular outline of the telescope, and discard all pixels outside the telescope's scope in the fragment shader.



How it works...

To implement the night vision effect, we have to do some tricks in the fragment shader. Every time we read a color value from the scene texture, we compare it with a constant vector `vec3(0.30, 0.59, 0.11)`, which is used to compute the grey scale intensity. If the intensity is too low, that is, if the area is too dark, it will be lightened. And the final color will be multiplied by a green light, to imitate the reflection color on the glass.

Another interesting topic here is how can we apply the main camera manipulator (which is automatically set by `viewer.run()` in this recipe) to the final scene on the HUD full screen quad. The answer can be found in the `osgCookBook::createRTTCamera()` function. It has an `isAbsolute` argument for choosing absolute or relative reference frame of the camera node.

As far as we know from some other OSG tutorials and related books, OSG uses `RELATIVE_RF` to specify that the node belongs to the relative reference frame of its parent. When the RTT camera is set to `RELATIVE_RF`, it will inherit the main camera's view and projection matrices and, thus, allow child scene to be affected by the manipulator. But if it is `ABSOLUTE_RF`, it must use its own matrix settings to project the scene to a desired range, as the view and projection matrices settings are reset before this camera is rendered. The relative frame mode must be applied to the first RTT camera which directly adds the original scene as child so that all navigation operations can work as usual. And the absolute frame is useful when your post-processing framework has more than one middle pass, because you have to make these middle-level RTT cameras view only their child quads to obtain correct rendering results all the time.

Implementing the depth-of-field effect

The classic depth-of-field (DOF) means that when camera lens is focused on an object, only the object and its nearby area will appear sharp in the final image, but subjects outside the area will appear blurred. It is a famous and common post-processing effect used in most 'next generation' computer games, especially the first-person ones, to enhance the reality.

We will use more than one pass to implement a basic DOF effect.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Texture2D>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. The post processing vertex shader is the same as the last one.

```
static const char* vertSource = {
  "void main(void)\n"
  "{\n"
  "gl_Position = ftransform();\n"
  "gl_TexCoord[0] = gl_MultiTexCoord0;\n"
  "}\n"
};
```

3. The post-blurring shader reads from the past color texture and accumulates neighboring texels to obtain the final blurred value.

```
static const char* blurFragSource = {
  "uniform sampler2D inputTex;\n"
  "uniform vec2 blurDir;\n"
  "void main(void)\n"
  "{\n"
    ...  // Please find details in the source code
  "}\n"
};
```

4. After the blurring work, we have to combine the original color texture and the blurred ones according to the depth value of each pixel. Pixels with farther depth values will use a blurred color, and the nearer ones will use the original color to make them clear to viewers. We will use two uniform variables here to decide if a pixel is near or far to the viewer.

```
static const char* combineFragSource = {
  "uniform sampler2D sceneTex;\n"
  "uniform sampler2D blurTex;\n"
  "uniform sampler2D depthTex;\n"
  "uniform float focalDistance;\n"
  "uniform float focalRange;\n"

  "float getBlurFromLinearDepth(vec2 uv)\n"
  "{\n"
  "float z = texture2D(depthTex, uv).x;\n"
  "z = 2.0 * 10001.0 / (10001.0 - z * 9999.0) - 1.0;\n"
  // Considering the default znear/zfar
  "return clamp((z - focalDistance)/focalRange, 0.0, 1.0);\n"
  "}\n"

  "void main(void)\n"
  "{\n"
  "vec2 uv = gl_TexCoord[0].st;\n"
  "vec4 fullColor = texture2D(sceneTex, uv);\n"
  "vec4 blurColor = texture2D(blurTex, uv);\n"
  "float blurValue = getBlurFromLinearDepth(uv);\n"
  "gl_FragColor = fullColor + blurValue * (blurColor -
    fullColor);\n"
  "}\n"
};
```

5. We will use a few independent functions to implement each pass of the depth-of-field effect. Each function will return the created camera node and the texture object at the same time.

```
typedef std::pair<osg::Camera*, osg::Texture*> RTTPair;
```

6. The original scene's color will be processed and outputted to a texture in the `createColorInput()` function.

```
RTTPair createColorInput( osg::Node* scene )
{
  osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
  tex2D->setTextureSize( 1024, 1024 );
  tex2D->setInternalFormat( GL_RGBA );

  osg::ref_ptr<osg::Camera> camera =
    osgCookBook::createRTTCamera(osg::Camera::COLOR_BUFFER,
      tex2D.get());
  camera->addChild( scene );
  return RTTPair(camera.release(), tex2D.get());
}
```

7. The original scene's depth values will be processed and outputted in the `createDepthInput()` function.

```
RTTPair createDepthInput( osg::Node* scene )
{
  osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
  tex2D->setTextureSize( 1024, 1024 );
  tex2D->setInternalFormat( GL_DEPTH_COMPONENT24 );
  tex2D->setSourceFormat( GL_DEPTH_COMPONENT );
  tex2D->setSourceType( GL_FLOAT );

  osg::ref_ptr<osg::Camera> camera =
    osgCookBook::createRTTCamera(osg::Camera::DEPTH_BUFFER,
      tex2D.get());
  camera->addChild( scene );
  return RTTPair(camera.release(), tex2D.get());
}
```

8. The color texture will be blurred in the `createBlurPass()` function. It also has an input argument to change the blur direction (horizontally or vertically).

```
RTTPair createBlurPass( osg::Texture* inputTex,
  const osg::Vec2& dir )
{
  osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
  tex2D->setTextureSize( 1024, 1024 );
```

```
    tex2D->setInternalFormat( GL_RGBA );
    osg::ref_ptr<osg::Camera> camera =
      osgCookBook::createRTTCamera(
        osg::Camera::COLOR_BUFFER, tex2D.get(), true);

    osg::ref_ptr<osg::Program> blurProg = new osg::Program;
    blurProg->addShader( new osg::Shader(osg::Shader::VERTEX,
      vertSource) );
    blurProg->addShader( new osg::Shader(osg::Shader::FRAGMENT,
      blurFragSource) );

    osg::StateSet* ss = camera->getOrCreateStateSet();
    ss->setTextureAttributeAndModes( 0, inputTex );
    ss->setAttributeAndModes( blurProg.get(),
      osg::StateAttribute::ON|osg::StateAttribute::OVERRIDE );
    ss->addUniform( new osg::Uniform("sceneTex", 0) );
    ss->addUniform( new osg::Uniform("blurDir", dir) );
    return RTTPair(camera.release(), tex2D.get());
}
```

9. In the main entry, we load an example terrain first.

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene =
  osgDB::readNodeFiles( arguments );
if ( !scene ) scene = osgDB::readNodeFile("lz.osg");
```

10. There are four passes before the final composition: read the original color and depth values, blur the color horizontally, and then blur the output vertically.

```
RTTPair pass0_color = createColorInput( scene.get() );
RTTPair pass0_depth = createDepthInput( scene.get() );
RTTPair pass1 = createBlurPass( pass0_color.second,
  osg::Vec2(1.0f, 0.0f) );
RTTPair pass2 = createBlurPass( pass2.second, osg::Vec2(
  0.0f, 1.0f) );
```

11. The final pass is to create the HUD quad for compositing past passes and generating the final result. The quad this time will overlay the entire screen.

```
osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0.0, 1.0, 0.0, 1.0);
hudCamera->addChild( osgCookBook::createScreenQuad(
  1.0f, 1.0f) );
```

12. Add the final shaders, textures, and uniforms to the quad's state set.

```
osg::ref_ptr<osg::Program> finalProg = new osg::Program;
finalProg->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertSource) );
finalProg->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  combineFragSource) );

osg::StateSet* stateset = hudCamera->getOrCreateStateSet();
stateset->setTextureAttributeAndModes( 0, pass0_color.second );
stateset->setTextureAttributeAndModes( 1, pass2.second );
stateset->setTextureAttributeAndModes( 2, pass0_depth.second );
stateset->setAttributeAndModes( finalProg.get() );
stateset->addUniform( new osg::Uniform("sceneTex", 0) );
stateset->addUniform( new osg::Uniform("blurTex", 1) );
stateset->addUniform( new osg::Uniform("depthTex", 2) );
stateset->addUniform( new osg::Uniform("focalDistance",
  100.0f) );
stateset->addUniform( new osg::Uniform("focalRange", 200.0f) );
```

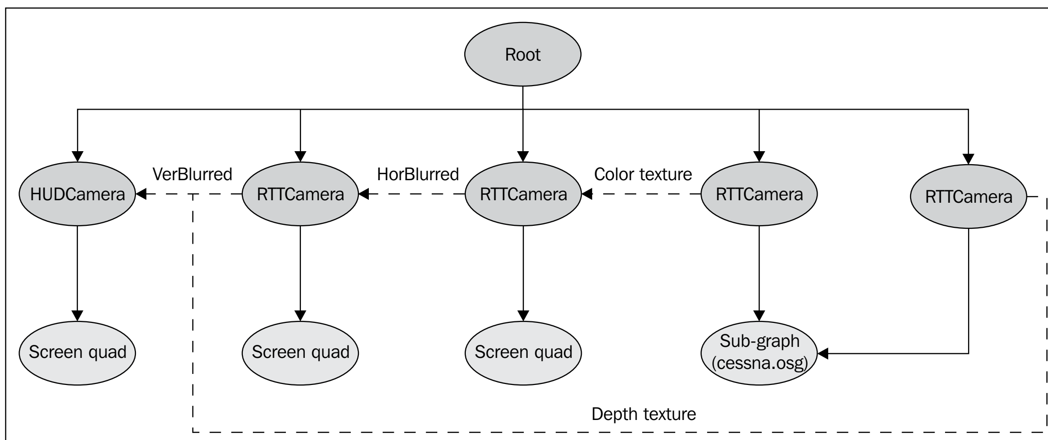
13. Build the scene graph and start the viewer. As we depend on the depth buffer values heavily, we have to remember to disable automatic near/far planes computation.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( pass0_color.first );
root->addChild( pass0_depth.first );
root->addChild( pass1.first );
root->addChild( pass2.first );
root->addChild( hudCamera.get() );

osgViewer::Viewer viewer;
viewer.getCamera()->setComputeNearFarMode(
  osg::CullSettings::DO_NOT_COMPUTE_NEAR_FAR );
viewer.setSceneData( root.get() );
return viewer.run();
```

14. OK, we have finally finished the DOF work. Navigate in the scene and you will find that only ground and trees near enough are clear, and far away objects will turn out of focus. This acts much more like a high-level camera in the real world whose lens can be changed precisely to focus at one distance at a time. There is no doubt that this effect can give us a different feeling while simulating walking or running in a huge scene.

How it works...

The following diagram shows the scene graph of the DOF implementation, and can best explain what we have done in each passes:



The first step is to obtain and store the color and depth values of the original scene into two different textures. The colored one is shown in the following screenshot:

Then we will blur the color texture per pixel (blur it twice—first horizontally, and then vertically) to create a blurred version of the scene image as shown in the following screenshot:

The last pass is to combine the sharp and blurred images based on the depth value of each fragment. The depth buffer values are not linear initially, so we have to create a `getBlurFromLinearDepth()` function in the fragment shader. It recalculates values from the depth texture with a linear method, according to known near and far values, as shown in the following block of code:

```
// Znear = 1.0 & Zfar = 10000.0
z = 2.0 * (10000 + 1.0) / (10000 + 1.0) - z *
    (10000 - 1.0)) - 1.0
  = 2.0 * 10001.0 / 10001.0 - z * 9999.0) - 1.0
```

These values are actually the default projection-matrix settings of OSG's main camera. Because we use `DO_NOT_COMPUTE_NEAR_FAR` to disable automatic near/far computation, they will never be changed unless you set them with the `setProjectionMatrix()` method.

There's more...

Now it's your turn to consider more post-processing effects. Some of them (including DOF) are used heavily in modern 3D games and applications and can make the scene much more realistic or dramatic. Of course, because the original scene should be rendered twice, there will be a remarkable performance loss when we have to handle huge data set. You may consider the multiple render-target solution for optimizing. Have a look at the `osgmultiplerendertargets` example in the core source code.

Here are some materials about other kinds of post-processing effects. You may also search for more introductions and implementations yourselves:

Gaussian Blur:
`http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/`

Motion Blur: `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html`

Bloom: `http://en.wikipedia.org/wiki/Bloom_(shader_effect)`

High Dynamic Range (HDR): `http://transporter-game.googlecode.com/files/HDRRenderingInOpenGL.pdf`

Screen Space Ambient Occlusion (SSAO): `http://en.wikipedia.org/wiki/Screen_Space_Ambient_Occlusion`

Fast Approximate Anti-Aliasing (FXAA): `http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf`

And don't forget the great NVIDIA shader library: `http://developer.download.nvidia.com/shaderlibrary/webpages/shader_library.html`

Designing a skybox with the cube map

A sky box is a method of creating background images. It often uses a cube to display the sky, mountains, and oceans. It can create realistic 3D surroundings for user applications and provide rich features comparing with a single background image, which was introduced in *Chapter 2*.

A standard real-time sky box requires a cube made up of six faces and will project six different textures onto these faces. It always remains stationary with respect to the eye position so that the sky and other distant objects are always very far away no matter how the camera is moved and rotated. The geometry of a sky box can be a cube too; sometimes they can also be constructed of a sphere or a hemisphere instead (so called a sky dome).

How to do it...

Let us start.

1. Include necessary headers:.

```
#include <osg/Depth>
#include <osg/TexGen>
#include <osg/TextureCubeMap>
#include <osg/ShapeDrawable>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgUtil/CullVisitor>
#include <osgViewer/Viewer>
```

2. We will first design a new kind of transformation node for representing sky dome. The most important methods to override will be `computeLocalToWorldMatrix()` and `computeWorldToLocalMatrix()`. The former is often called in the cull traversal for computing the model matrix. So we can do some tricks here to make the node follow the viewer's eyes all the time.

```
class SkyBox : public osg::Transform
{
public:
  SkyBox();

  SkyBox( const SkyBox& copy, osg::CopyOp copyop=
    osg::CopyOp::SHALLOW_COPY )
  : osg::Transform(copy, copyop) {}

  META_Node( osg, SkyBox );
```

```
  void setEnvironmentMap( unsigned int unit, osg::Image* posX,
    osg::Image* negX, osg::Image* posY, osg::Image* negY,
      osg::Image* posZ, osg::Image* negZ );

  virtual bool computeLocalToWorldMatrix( osg::Matrix& matrix,
    osg::NodeVisitor* nv ) const;
  virtual bool computeWorldToLocalMatrix( osg::Matrix& matrix,
    osg::NodeVisitor* nv ) const;

protected:
  virtual ~SkyBox() {}
};
```

3. In the `SkyBox` constructor, we must use `setCullingActive()` method to disable any further culling work on this node. That is because the sky box is following the eye and its actual position and bound (these will be used for view frustum culling) can hardly be calculated. Another interesting matter here is the operation on the state set. If you still remember the background image example in *Chapter 2*, you will easily understand the reason why we set up an `osg::Depth` attribute and make the state set rendered later by calling `setRenderBinDetails()` method.

```
SkyBox::SkyBox()
{
  setReferenceFrame( osg::Transform::ABSOLUTE_RF );
  setCullingActive( false );

  osg::StateSet* ss = getOrCreateStateSet();
  ss->setAttributeAndModes( new osg::Depth(
    osg::Depth::LEQUAL, 1.0f, 1.0f) );
  ss->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
  ss->setMode( GL_CULL_FACE, osg::StateAttribute::OFF );
  ss->setRenderBinDetails( 5, "RenderBin" );
}
```

4. The `setEnvironmentMap()` method will read six images and merge them together as a cube map texture to be applied on the sky geometry.

```
void SkyBox::setEnvironmentMap( unsigned int unit,
  osg::Image* posX, osg::Image* negX,
    osg::Image* posY, osg::Image* negY,
      osg::Image* posZ, osg::Image* negZ )
{
  osg::ref_ptr<osg::TextureCubeMap> cubemap =
    new osg::TextureCubeMap;
  cubemap->setImage( osg::TextureCubeMap::POSITIVE_X, posX );
  cubemap->setImage( osg::TextureCubeMap::NEGATIVE_X, negX );
```

```
    cubemap->setImage( osg::TextureCubeMap::POSITIVE_Y, posY );
    cubemap->setImage( osg::TextureCubeMap::NEGATIVE_Y, negY );
    cubemap->setImage( osg::TextureCubeMap::POSITIVE_Z, posZ );
    cubemap->setImage( osg::TextureCubeMap::NEGATIVE_Z, negZ );
    ...  // Please find details in the source code
    cubemap->setResizeNonPowerOfTwoHint( false );
    getOrCreateStateSet()->setTextureAttributeAndModes(
      unit, cubemap.get() );
}
```

5. In the `computeLocalToWorldMatrix()` method, we will try to obtain current eye position from the `osgUtil::CullVisitor` object and apply it to the sky box's matrix so that the center of the sky box will automatically be set to the eye point in every frame.

```
bool SkyBox::computeLocalToWorldMatrix( osg::Matrix& matrix,
  osg::NodeVisitor* nv ) const
{
  if ( nv && nv->getVisitorType()==
    osg::NodeVisitor::CULL_VISITOR )
  {
    osgUtil::CullVisitor* cv =
      static_cast<osgUtil::CullVisitor*>( nv );
    matrix.preMult( osg::Matrix::translate(cv->getEyeLocal()) );
    return true;
  }
  else
  return osg::Transform::computeLocalToWorldMatrix( matrix, nv );
}
```

6. The `computeLocalToWorldMatrix()` method will work on the world-to-local matrix. So all its operations will be just opposite to the ones in the `computeLocalToWorldMatrix()` method.

```
bool SkyBox::computeWorldToLocalMatrix( osg::Matrix& matrix,
  osg::NodeVisitor* nv ) const
{
  if ( nv && nv->getVisitorType()==
    osg::NodeVisitor::CULL_VISITOR )
  {
    osgUtil::CullVisitor* cv =
      static_cast<osgUtil::CullVisitor*>( nv );
    matrix.postMult( osg::Matrix::translate(
      -cv->getEyeLocal()) );
    return true;
  }
  else
  return osg::Transform::computeWorldToLocalMatrix( matrix, nv );
}
```

7. In the main entry, we load the terrain model again.

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene = osgDB::readNodeFiles(
  arguments );
if ( !scene ) scene =
  osgDB::readNodeFile("lz.osg.90,0,0.rot");
```

8. Create a simple sphere as the sky geometry. Of course a box can fit the class name `SkyBox` better, but a sphere is sometimes easier to handle and it has no seams.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( new osg::ShapeDrawable(
  new osg::Sphere(osg::Vec3(), scene->getBound().radius())) );
```

9. Apply the cube map to the sky node. Note that the sphere geometry needs to have a new texture coordinate array for correct texture mapping. A default `osg::TexGen` is enough in this recipe.

```
osg::ref_ptr<SkyBox> skybox = new SkyBox;
skybox->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, new osg::TexGen );
skybox->setEnvironmentMap( 0,
  osgDB::readImageFile("Cubemap_snow/posx.jpg"),
    osgDB::readImageFile("Cubemap_snow/negx.jpg"),
      osgDB::readImageFile("Cubemap_snow/posy.jpg"),
        osgDB::readImageFile("Cubemap_snow/negy.jpg"),
          osgDB::readImageFile("Cubemap_snow/posz.jpg"),
            osgDB::readImageFile("Cubemap_snow/negz.jpg") );
skybox->addChild( geode.get() );
```

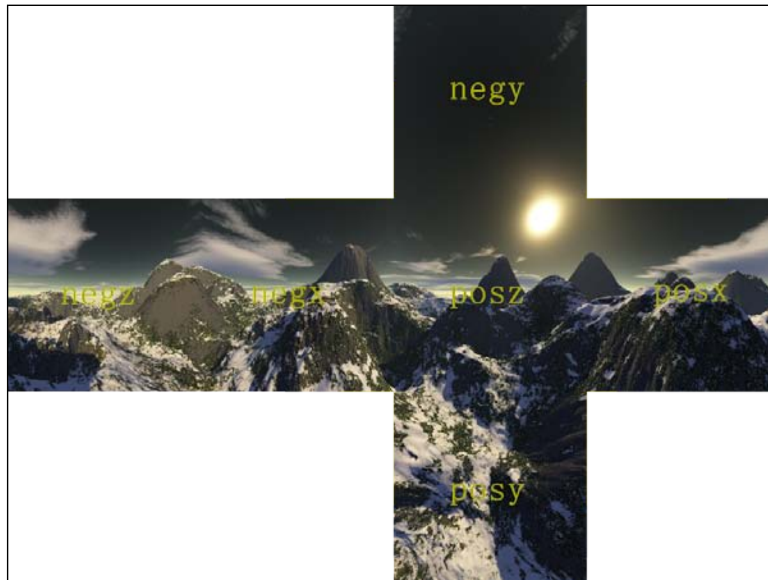
10. Create the scene graph and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( scene.get() );
root->addChild( skybox.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

11. Now manipulate the camera to view the example terrain, and you will see the clouds and snow mountains at an unreachable distance. This makes the entire scene look larger and more vivid, and there is no need to render real sky geometries at all, which may slow down the frame rate sharply.

How it works...

The scenery is projected to the sky-box faces using a technique named **cube mapping**. This is a kind of environment mapping method that uses a six-sided cube as the map shape. You have to provide six separated panoramic sky images and use them to generate such a cube map texture. An unfolded picture of the six faces of the cube is shown in the following screenshot:



OSG uses `osg::TextureCubeMap` class to support cube mapping. It requires the geometries to have 3D texture coordinates (XYZ, or STR in texture coordinates). The `osg::TexGen` class can generate them in object mode quickly, but it is still suggested that you specify the texture coordinates yourselves to provide more flexibility in your own applications.

Creating a simple water effect

Real-time water rendering is one of the most interesting and challenging topics in computer games and virtual reality applications. There are several types of water in the natural world such as the ocean, lake, river, and so on. The basic components of a complete water simulation include water waves, ripples, reflections, refractions, foams, and caustics. Of course it is difficult work to implement all of them in just one example. In fact, we can even write another book to talk about various water rendering techniques. So we would like to make this recipe as an easy start for people who have an interest in this topic.

In *Chapter 2*, we have already discussed about the reflection of a scene. And in the last few recipes, we worked with rendering-to-texture cameras. This time we will merge them together to have a flight flying in the air with its inverted image on the water plane. After that, we are going to add some noise as waves and ripples, and combine them to generate a simple but workable water scene.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/TexGen>
#include <osg/ShapeDrawable>
#include <osg/Geometry>
#include <osg/ClipNode>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
```

2. Although this is so called 'a simple water effect', the vertex shader code is still too long to read through. We will try to explain its main theory later in the *How it works* section. Please find details in the source code package of this book.

```
static const char* waterVert = {
  ...  // Please find details in the source code
};
```

3. The fragment shader is lengthy too. It requires at least two input textures: the default water texture, reflection map, refraction map, and the normal map for lighting.

```
static const char* waterFrag = {
  ...  // Please find details in the source code
};
```

The `createTexture()` function reads an image file and returns a new 2D texture.

```
osg::Texture2D* createTexture( const std::string& filename )
{
   ...  // Please find details in the source code
}
```

4. In the main entry, load a model and use it as the scene reflected on the water surface.

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene = osgDB::readNodeFiles(
  arguments );
if ( !scene ) scene = osgDB::readNodeFile("cessna.osg");
```

5. Use transformation and clip nodes to construct the reversed scene. This is exactly the same as we have done in the reflection example in *Chapter 2*.

```
float z = -20.0f;
osg::ref_ptr<osg::MatrixTransform> reverse =
  new osg::MatrixTransform;
reverse->preMult(osg::Matrix::translate(0.0f, 0.0f, -z) *
  osg::Matrix::scale(1.0f, 1.0f, -1.0f) *
    osg::Matrix::translate(0.0f, 0.0f, z) );
reverse->addChild( scene.get() );

osg::ref_ptr<osg::ClipPlane> clipPlane = new osg::ClipPlane;
clipPlane->setClipPlane( 0.0, 0.0, -1.0, z );
clipPlane->setClipPlaneNum( 0 );

osg::ref_ptr<osg::ClipNode> clipNode = new osg::ClipNode;
clipNode->addClipPlane( clipPlane.get() );
clipNode->addChild( reverse.get() );
```

6. Use a render-to-texture camera to render the reversed scene to a texture.

```
osg::ref_ptr<osg::Texture2D> tex2D = new osg::Texture2D;
tex2D->setTextureSize( 1024, 1024 );
tex2D->setInternalFormat( GL_RGBA );

osg::ref_ptr<osg::Camera> rttCamera =
  osgCookBook::createRTTCamera(osg::Camera::COLOR_BUFFER,
    tex2D.get());
rttCamera->addChild( clipNode.get() );
```

7. Now let us create the water plane and map the reflection map onto it later with the shaders.

```
const osg::Vec3& center = scene->getBound().center();
float planeSize = 20.0f * scene->getBound().radius();
osg::Vec3 planeCorner( center.x()-0.5f*planeSize,
  center.y()-0.5f*planeSize, z );
osg::ref_ptr<osg::Geometry> quad =
  osg::createTexturedQuadGeometry(
    planeCorner, osg::Vec3(planeSize, 0.0f, 0.0f),
      osg::Vec3(0.0f, planeSize, 0.0f) );

osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( quad.get() );
```

8. Add all other textures to the water plane's state set.

```
osg::StateSet* ss = geode->getOrCreateStateSet();
ss->setTextureAttributeAndModes( 0, tex2D.get() );
// the reflection texture (RTT of the cessna)
ss->setTextureAttributeAndModes( 1,
  createTexture("Images/skymap.jpg") ); // the default texture
ss->setTextureAttributeAndModes( 2,
  createTexture("water_DUDV.jpg") ); // the refraction texture
ss->setTextureAttributeAndModes( 3,
  createTexture("water_NM.jpg") ); // the normal texture
```

9. Add create the shader program object and uniforms.

```
osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  waterVert) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  waterFrag) );
ss->setAttributeAndModes( program.get() );
ss->addUniform( new osg::Uniform("reflection", 0) );
ss->addUniform( new osg::Uniform("defaultTex", 1) );
ss->addUniform( new osg::Uniform("refraction", 2) );
ss->addUniform( new osg::Uniform("normalTex", 3) );
```

10. Build the scene graph and start the viewer.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( rttCamera.get() );
root->addChild( geode.get() );
root->addChild( scene.get() );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();
```

11. Use your mouse to manipulate the camera and get near to the Cessna model and its inverted reflection in water. You can see the waves and ripples on the water plane. Although it is still rough for use in real applications, you may find that it contains most basic elements of water simulation and can be improved to provide much better effects in future.

How it works...

Although it will be much better if you construct grid geometry and perform transformations on it to simulate waves, we still select to use noise textures here to simplify the program. The vertex shader helps calculate the light and eye directions of each vertex, and transform them to the tangent space, which is also mentioned in the bump mapping example. We assume that the normal is always `vec3(0.0, 0.0, 1.0)` and, thus, directly have the tangent transformation matrix as shown in the following block of code:

```
vec3 T = vec3(0.0, 1.0, 0.0);
vec3 N = vec3(0.0, 0.0, 1.0);
vec3 B = vec3(1.0, 0.0, 0.0);
...
mat3 TBNmat;
TBNmat[0][0] = T[0]; TBNmat[1][0] = T[1]; TBNmat[2][0] = T[2];
TBNmat[0][1] = B[0]; TBNmat[1][1] = B[1]; TBNmat[2][1] = B[2];
TBNmat[0][2] = N[0]; TBNmat[1][2] = N[1]; TBNmat[2][2] = N[2];
```

Another important step in the vertex shader is to compute two texture coordinates for wave and ripple computing. They come from the built-in OSG uniform `osg_FrameTime`, so they will change all the time when the program is running.

In the fragment shader, we read from the refraction map, which is in fact the derivation of a normal map (a DUDV map), for texture coordinate distortion. The distorted UV values are used for retrieving normal vectors from the normal map for light computation, as well as reading from the base and reflection textures for final output. The wave and ripple effects are generated because of the distortion too.

There's more...

If you have interests in learning more about water simulation, especially realistic ocean simulation, you must not miss the **osgOcean** project, which is developed as part of an EU funded research initiative. You can find more information and the full source code at `http://code.google.com/p/osgocean/`.

Creating a piece of cloud

Clouds are ubiquitous, unique, and beautiful existences of the world. They exist in the atmosphere and can have different shapes and types due to the meteorology factors. A huge aggregation of clouds may lead to precipitation as a result. All of these will be great features in computer games if they can be simulated in a certain way.

As clouds are made up of many small liquid droplets or frozen crystals, they can be treated as a visible mass of these particles, with each particle placed at a certain position with brightness and a color value. This is actually the basic concept of our cloud simulation implementation in this recipe.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/BlendFunc>
#include <osg/Depth>
#include <osg/Texture2D>
#include <osg/Drawable>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <algorithm>
#include <fstream>
#include <iostream>
```

2. Define the `CloudBlock` class. As it is a drawable class, the most important methods to derive will be `computeBound()` and `drawImplementation()`. It also includes two structure definitions—the `CloudCell` records a basic cloud cell's attributes, and the `LessDepthSortFunctor` functor (function object) for sorting cloud cells from back to front (in depth order).

```
class CloudBlock : public osg::Drawable
{
public:
  struct CloudCell
  {
    ...
  };

  struct LessDepthSortFunctor
  {
    ...
  };

public:
  ...  // Please find details in the source code

protected:
  void renderCells( const osg::Matrix& modelview ) const;

  mutable CloudCells _cells;
};
```

3. Define the `CloudCell` structure. A cloud cell in our recipe must have the position, brightness, and density (transparency) parameters.

```
CloudCell() : _brightness(0.0f), _density(0.0f) {}

bool operator==( const CloudCell& copy ) const
{
  return _pos==copy._pos && _brightness==copy._brightness &&
  _density==copy._density;
}

osg::Vec3d _pos;
float _brightness;
float _density;
```

4. The `LessDepthSortFunctor` structure will calculate a hypothetical depth value (as the cells are not really rendered so there are no values in the depth buffer, we have to compute approximate ones here) for each cell according to current view matrix, which is passed as the constructor's argument. It is used as a descending sort functor (depth values from the largest to smallest) while drawing and blending cloud cells.

```
LessDepthSortFunctor( const osg::Matrix& matrix )
{
  _frontVector.set(-matrix(0,2), -matrix(1,2),
  -matrix(2,2), -matrix(3,2));
}

bool operator()( const CloudCell& lhs, const CloudCell& rhs )
  const
{
  return getDepth(lhs._pos) > getDepth(rhs._pos);
}

// The front vector is just the direction from view center to
// the viewer's eye, so we compute the depth by obtaining the
// dot product (in fact projects pos on the vector)
float getDepth( const osg::Vec3d& pos ) const
{
  return (float)pos[0] * _frontVector[0] +
    (float)pos[1] * _frontVector[1] +
      (float)pos[2] * _frontVector[2] + _frontVector[3];
}

osg::Vec4 _frontVector;
```

5. Computing the cloud's bounding box can help cull it correctly.

```
osg::BoundingBox CloudBlock::computeBound() const
{
  osg::BoundingBox bb;
  for ( CloudCells::const_iterator itr=_cells.begin();
  itr!=_cells.end(); ++itr )
  {
    bb.expandBy( itr->_pos );
  }
  return bb;
}
```

6. In the `drawImplementation()` method, we first re-sort all cloud cells using the `LessDepthSortFunctor`, and then call the `renderCells()` method internally.

```
void CloudBlock::drawImplementation( osg::RenderInfo&
  renderInfo ) const
{
  const osg::State* state = renderInfo.getState();
  if ( !state || !_cells.size() ) return;

  const osg::Matrix& modelview = state->getModelViewMatrix();
  std::sort( _cells.begin(), _cells.end(),
    LessDepthSortFunctor(modelview) );

  glPushMatrix();
  renderCells( modelview );
  glPopMatrix();
}
```

7. The `renderCells()` will do the actual rendering work.

```
void CloudBlock::renderCells( const osg::Matrix& modelview ) const
{
  ...
}
```

8. Prepare for the rendering. Here `px` and `py` are calculated from the model-view matrix. They in fact represent the right and up vectors in the eye coordinates. These parameters can help draw cloud cells later because they must face the screen all the time if rendered as a quad.

```
osg::Vec3d px = osg::Matrix::transform3x3( modelview,
  osg::X_AXIS );
osg::Vec3d py = osg::Matrix::transform3x3( modelview,
  osg::Y_AXIS );
px.normalize(); py.normalize();

double size = 1.0f, scale = 1.0f;
osg::Vec3d right, up;
glBegin( GL_QUADS );
```

9. Now for each cloud cell, we will compute a suitable color and alpha value according to the preset variables in the `CloudCell` object and draw a quad directly. This is not efficient but flexible enough if we need more operations on specified cloud cells, and don't depend on any shader code so that it can work on some early machines.

```
unsigned int detail = 1;
unsigned int numOfCells = _cells.size();
for ( unsigned int i=0; i<numOfCells; i+=detail )
{
  const CloudCell& cell = _cells[i];
  osg::Vec3d pos = cell._pos;
  unsigned char alpha = (unsigned char)( cell._density );
  unsigned char color = (unsigned char)( cell._brightness *
    cell._density / 255.0f );
  right.set( px * size * scale );
  up.set( py * size * scale );
  ...  // Use OpenGL calls. Find it in the source code
}
glEnd();
```

10. Don't forget to design a glow image for the cloud cell. This will be done in the `makeGlow()` function, which designs glow images on the fly.

```
osg::Image* makeGlow( int width, int height, float expose,
  float sizeDisc )
{
  ...  // Please find details in the source code
}
```

11. The `readCloudCells()` function reads thousands of cloud cells from a data file and, thus, generates a complete piece of cloud.

```
void readCloudCells( CloudBlock::CloudCells& cells,
  const std::string& file )
{
  ...  // Please find details in the source code
}
```

12. In the main entry, we read the cloud data and construct the `CloudBlock` instance. The `data.txt` file can be found in the source code directory of this book.

```
CloudBlock::CloudCells cells;
readCloudCells( cells, "data.txt" );

osg::ref_ptr<CloudBlock> clouds = new CloudBlock;
clouds->setCloudCells( cells );
```

13. The next three steps: Add the `osg::BlendFunc` attribute to enable transparency; disable writing to the depth buffer so cloud cells are drawn regardless of their distances to eye, and then apply a glow texture on the cloud object.

```
osg::StateSet* ss = clouds->getOrCreateStateSet();
ss->setAttributeAndModes( new osg::BlendFunc(GL_ONE,
  GL_ONE_MINUS_SRC_ALPHA) );
ss->setAttributeAndModes( new osg::Depth(osg::Depth::LESS,
  0.0, 1.0, false) );
ss->setTextureAttributeAndModes(
  0, new osg::Texture2D(makeGlow(32, 32, 2.0f, 0.0f)) );
```

14. Start the viewer at last:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( clouds.get() );

osgViewer::Viewer viewer;
viewer.setLightingMode( osg::View::SKY_LIGHT );
viewer.setSceneData( geode.get() );
return viewer.run();
```

Now a big piece of cloud is created and rendered in the scene! It may cause a low frame rate because of the heavy sorting and rendering work in OpenGL's immediate mode. But at least we can learn to design our own clouds from this recipe, and as there are no shaders in use, the source code can be migrated to some very early devices to support cloud managing and rendering.

How it works...

The key work of this recipe is to sort the cloud particles (called `CloudCell` here) and render them one by one. The first step is really costly as it must compute the depth value of each cell and then arrange them from the largest to the smallest. The depth is the dot product of current cloud cell position and the front vector, which is defined as the direction from the eye to the view point. Cells with larger depth will be placed at the front indices of the list, so they will be rendered at the beginning, and then blended with successor cells to generate approximate cloud effects.

We may speed up the program by replacing the outdated `glBegin()/glEnd()` functions with OpenGL's point sprite extension. But the sorting process can hardly be removed because of the transparency sorting feature. Maybe you can think of some GPU-based algorithms in the future to improve this example for your own use.

Customizing the state attribute

OSG provides over 40 kinds of state attributes that can be applied to the state set of a node or geometry. These attributes work well with OpenGL's state machine mechanism as they are set before the geometries and are rendered to support fixed and shader effects. Some attributes can cooperate with modes, which enable or disable a specific state immediately. All these operations are defined and executed in `osg::StateAttribute` derived classes. So, how can we inherit the `osg::StateAttribute` class? This is actually what we are going to talk about.

You may still remember that we have already integrated NVIDIA Cg and OSG within the camera draw callbacks in *Chapter 2*. Now we would like to rewrite this example to use a custom-state attribute instead. Most Cg-related functions and initialization methods are just the same as before, so we can focus on the implementation of the new attribute class in the following sections.

How to do it...

Let us start.

1. The headers to be included are just the same as the Cg integration example in *Chapter 2*. And we will define a new macro that will be used to identify our Cg state attribute later.

   ```
   #define CGPROGRAM_ID 0x1000
   ```

2. Declare the `CgProgram` class derived from `osg::StateAttribute`. The `META_StateAttribute` macro must be used with the identity macro to make the attribute register in the state set.

   ```
   class CgProgram : public osg::StateAttribute
   {
   public:
     CgProgram() : _initialized(false) {}

     CgProgram( const CgProgram& copy, const osg::CopyOp&
       copyop=osg::CopyOp::SHALLOW_COPY )
     : osg::StateAttribute(copy, copyop),
       _profiles(copy._profiles), _programs(copy._programs),
       _initialized(copy._initialized)
       {}

     META_StateAttribute( osg, CgProgram, (
       osg::StateAttribute::Type)CGPROGRAM_ID );

     void addProfile( CGprofile profile );
   ```

```
    void addCompiledProgram( CGprogram prog )
    {
      _programs.push_back(prog);
    }

    virtual int compare( const osg::StateAttribute& sa ) const;
    virtual void apply(osg::State& state) const;

protected:
    virtual ~CgProgram() {}

    std::vector<CGprofile> _profiles;
    std::vector<CGprogram> _programs;
    mutable bool _initialized;
};
```

3. We have to create a global Cg profile list here. We will see its usage later.

```
static std::vector<CGprofile> g_profiles;
```

4. The `addProfile()` method will add to the `CgProgram` class' member list and the global list together.

```
void CgProgram::addProfile( CGprofile profile )
{
  _profiles.push_back(profile);
  g_profiles.push_back(profile);
}
```

5. The `compare()` (virtual method) is used to check if current attribute is different from another one and, thus, share it or remove the redundancy. It uses kinds of comparison macros for convenience.

```
int CgProgram::compare( const osg::StateAttribute& sa ) const
{
  COMPARE_StateAttribute_Types(CgProgram, sa)
  COMPARE_StateAttribute_Parameter(_profiles)
  COMPARE_StateAttribute_Parameter(_programs)
  COMPARE_StateAttribute_Parameter(_initialized)
  return 0;
}
```

6. The `apply()` method will be called during the drawing process. So we are going to load and enable all Cg programs applied to the state attribute. To note, Cg profiles must be disabled at the end of each frame, but OSG doesn't provide any virtual method such as `applyEnd()`. A solution using the global profile list is shown in the following code segment, and will be explained in the next section:

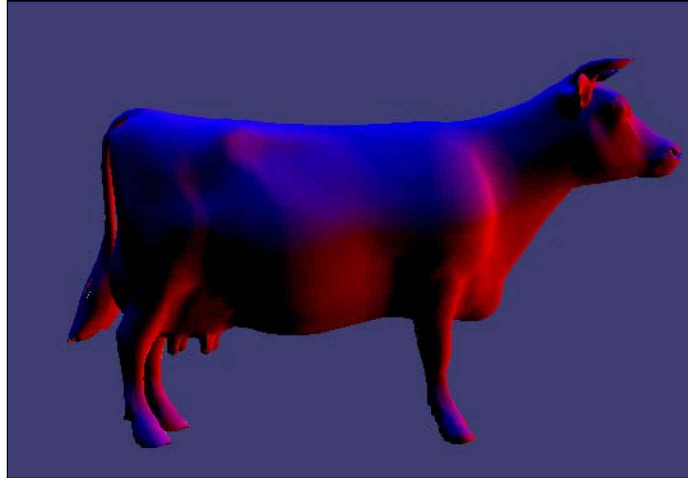
```
void CgProgram::apply(osg::State& state) const
{
  if ( !_profiles.size() )
  {
    // Disable all profiles in the default attribute
    for ( unsigned int i=0; i<g_profiles.size(); ++i )
    cgGLDisableProfile( g_profiles[i] );
    return;
  }

  if ( !_initialized )
  {
    for ( unsigned int i=0; i<_programs.size(); ++i )
    cgGLLoadProgram( _programs[i] );
    _initialized = true;
  }

  for ( unsigned int i=0; i<_programs.size(); ++i )
  cgGLBindProgram( _programs[i] );
  for ( unsigned int i=0; i<_profiles.size(); ++i )
  cgGLEnableProfile( _profiles[i] );
}
```

7. The next work, including creating the NVIDIA Cg shader code and compiling/releasing them in the main entry, is very similar to the code we have already written in *Chapter 2* earlier. The only difference is that we won't use the main camera's post-draw callbacks but apply an instance of the customized `CgProgram` attribute on the model instead.

   The creation of the Cg program attribute is shown as follows:

```
osg::ref_ptr<CgProgram> cgProg = new CgProgram;
cgProg->addProfile( vertProfile );
cgProg->addProfile( fragProfile );
cgProg->addCompiledProgram( vertProg );
cgProg->addCompiledProgram( fragProg );
root->getOrCreateStateSet()->setAttribute( cgProg.get() );
```

8. OK, the result seems completely the same as we had seen in *Chapter 2*. But this time we use a completely different method to achieve the goal. A customized state attribute is used to replace the camera callbacks. This gives us more flexibility as state attributes can be applied to any nodes and geometries in a scene graph, but camera callbacks must be set to a camera and will always affect all its child nodes together.



How it works...

You must be interested about the use of the global profile list, `g_profiles`, and how can OSG provide an 'end' process for attributes that require a finish sign (for example, Cg needs `cgGLDisableProfile()` function to disable a shader after rendering). The following code segments will disable all registered profiles:

```
if ( !_profiles.size() )
{
  for ( unsigned int i=0; i<g_profiles.size(); ++i )
  cgGLDisableProfile( g_profiles[i] );
  return;
}
```

As a global variable, `g_profiles` has no relation to any concrete instance of the `CgProgram` class and can only be called when there is no element in the `_profiles` member variable. So, how can the `_profiles` be empty? The answer is simple, a newly allocated `CgProgram` without calling the `addProfile()` method. But we never created such an empty object in this recipe.

However, OSG does. Every time we add a new attribute to the scene graph, another one using the default constructor and default values will be created at the same time and stored in the internal global state set of OSG's rendering back end.

When a drawable is going to be rendered, it will always apply the global state set first, and the other state sets in its parent node path. That means the `CgProgram` attribute will be reset at the very beginning of each geometry's rendering process. So `cgGLDisableProfile()` function will be executed at the time when a geometry without applying Cg shaders is going to be drawn. That just works as if we have an 'end' process! You can always provide such 'finishing' code in the `apply()` method and make sure it is called when the attribute is allocated in default mode.

We can directly use `osg::StateSet`'s `getAttribute()` and `removeAttribute()` methods to obtain and erase this new attribute type. For example:

```
CgProgram* prog = static_cast<CgProgram*>(
  stateset->getAttribute(CGPROGRAM_ID) );
```

7
Visualizing the World

In this chapter, we will cover:

- Preparing the VirtualPlanetBuilder (VPB) tool
- Generating a small terrain database
- Generating a terrain database on the earth
- Working with multiple imagery and elevation data
- Patching an existing terrain database with newer data
- Building NVTT support for device-independent generation
- Using SSH to implement cluster generation
- Loading and rendering terrain from the Internet

Introduction

It is always exciting to create and view a large area, for example, the earth, in our OSG-based applications. A detailed terrain which can be paged dynamically and rendered smoothly is necessary for geographic information system (GIS). And that is what we are going to discuss in this chapter.

Early OSG developers may have heard of a simple utility named `osgdem` in the core OSG releases at that time. It can build terrain data from original elevation and texture files and makes the results easy to merge into the scene graph. There is even a **BlueMarbleViewer** project showing how to build earth models with NASA's BlueMarble imagery using OpenSceneGraph 1.2 at `http://www.andesengineering.com/BlueMarbleViewer/`.

The `osgdem` utility has grown to a complete terrain generation tool set named **VirtualPlanetBuilder**, which is also managed by Robert Osfield, the OSG team leader. And there are some other very good terrain builders and renderers. We will introduce one of them in this chapter—the **osgEarth** project, which is maintained by developers at Pelican Mapping (`http://pelicanmapping.com/`).

Building terrain requires original data. Some low-resolution data can be downloaded freely from the Internet, but some of them cannot be used directly for commercial purposes. You may have to obtain data from regular map-service providers and get permits first while developing paid software such as GIS systems and earth viewers.

Preparing the VirtualPlanetBuilder (VPB) tool

The **VirtualPlanetBuilder** (**VPB**) is the best known terrain-creation tool based on OSG. It uses the famous GDAL library to read a wide range of geospatial imagery and elevation data formats, and build paged terrain database for real-time viewing and analyzing.

VPB was first designed as a terrain-generation tool in OpenSceneGraph 1.2. As it developed so rapidly, it soon became a separate project focusing only on databases creation. It now supports working under projected and earth coordinates, processing gigabyte- and terabyte-sized data, cluster building using SSH, and different database optimization methods.

At the time the book is being written, VPB is still on its way to the stable 1.0 release. So we will work on the latest trunk version of it while studying the next few recipes in this chapter.

You can read more about VPB at the following website:

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

And for more about the GDAL project and its derivatives, refer to the following link:

`http://www.gdal.org/`

Getting ready

Before we build VPB from the source code and use it for terrain creation later, we should establish some prerequisites. One is to install OSG headers and libraries at a reachable location. Of course every reader of this book should be able to achieve this.

The other requirement is that you must have the GDAL library installed, which will be used heavily in VPB for reading and parsing original raster data. You may download the source code and build it from the source code too. But GDAL has already provided various downloadable binaries for different platforms and versions. Linux, Mac OS X, and Windows users please see the download link for details:

`http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries`

And Debian and Ubuntu developers can also make use of the common `apt-get` command to obtain GDAL binaries and developer files:

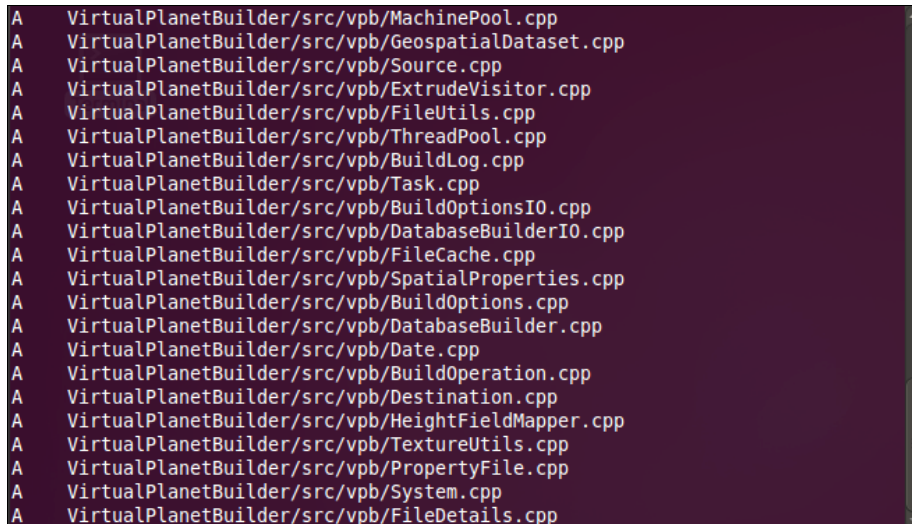
```
# apt-get install gdal-bin
# apt-get install libgdal-dev
```

How to do it...

Let us start.

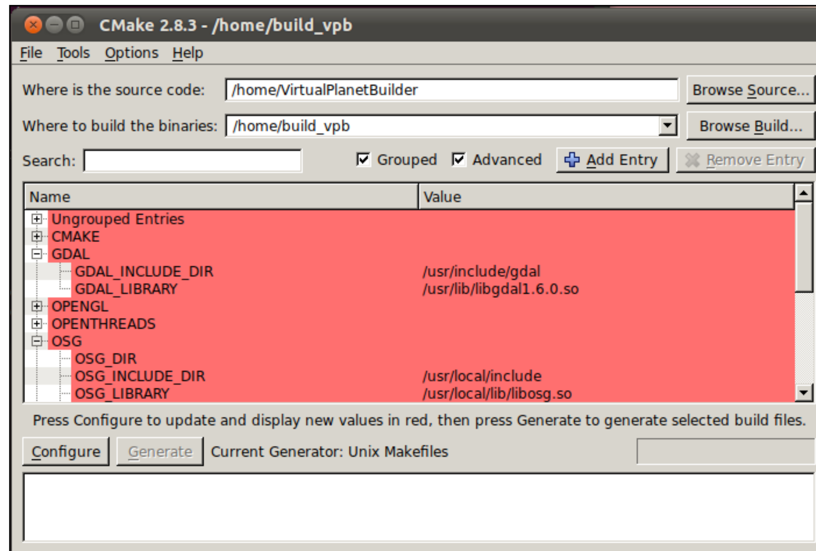
1. You will have to check out the VPB source code with any Subversion tool:

   ```
   # svn checkout http://www.openscenegraph.org/svn/
     VirtualPlanetBuilder/trunk VirtualPlanetBuilder
   ```



2. Start the `cmake-gui` executable and select the VPB source's root directory and a new folder to place the building-related files.
3. Choose a suitable generator and start the configuration. Like the recipes in *Chapter 1*, there will be a few options for you to check and edit before really generating the makefiles or solution.

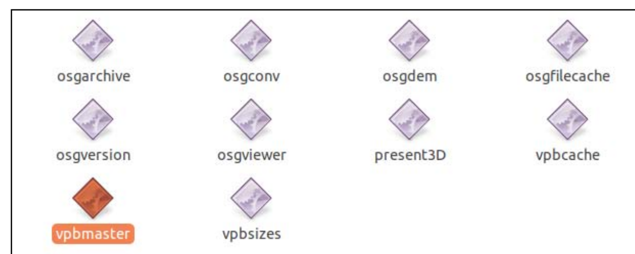
4. The `GDAL` group and the `OSG` group are the most important, without which you will fail to make the VPB system work. Please open these two groups and see if `include` directories and libraries are set. Under Linux, this is always done automatically because most developer files can be located in the `/usr` and `/usr/local` directories. But Windows users may have to specify the folders by themselves.



5. Click on **Generate** to create the makefiles. Note that it is disabled until you choose **Configure** again to set up the options, as shown in the preceding screenshot. Next, you can compile VPB in the build folder immediately using the following command:

```
# sudo make
# sudo make install
```

6. Windows users could choose Visual Studio as the generator. And Mac OS X users may either use XCode project or UNIX makefiles.
7. Now you will find some more executables in the `bin` directory of your installation folder. Among them, `vpbmaster` is the most important one and the only one to be introduced in the remainder part of this chapter.



How it works...

Most OSG-based projects, including VPB and some other projects introduced before (`osgOcean` and so on), use CMake as the building system. So it is important for them to find various OSG libraries as dependencies. CMake provides an automatic searching script which can look for OSG installations under `/usr` and `/usr/bin` directories, as well as the place indicated by the environment variable `OSG_DIR`. The CMake system will then try to find OSG's necessary header files in the `include` subdirectory of each folder specified in `OSG_DIR`, and library files in the `lib` subdirectory. If successful, it presets these locations as the default values before the user-configuration process. This, of course, brings convenience when there are too many options to set for the same dependency.

There is a similar solution for specifying GDAL options in CMake, but it uses another environment variable `GDAL_DIR` instead, which indicates where GDAL binaries and libraries are installed.

Generating a small terrain database

Looking into the installation folder, there are at least four new applications in the `bin` directory:

- `vpbmaster`: The main processor for terrain database generation. It is a command-line tool without any GUI.
- `vpbcache`: A tool for creating cache or building re-projections of original source data.
- `vpbsizes`: A convenient calculator for computing tile sizes of specified terrain width and height.
- `osgdem`: The terrain-creating tool used internally for handling different terrain tiles. There may be multiple `osgdem` applications running parallel when users use `vpbmaster` to build a huge database.

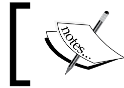
The next step is to create a small terrain database (only a few megabytes) using the `vpbmaster` tool. Of course, the first thing is to look for some adequate original data.

Getting ready

The Large Geometric Models Archive project at Georgia Tech has some excellent terrain data that can be used here to show how VPB works with original geographic data. The project site is managed by Greg Turk and Brendan Mullins. You can visit it at `http://www.cc.gatech.edu/projects/large_models/`.

And we are mainly interested in the Grand Canyon data, which can be found at `http://www.cc.gatech.edu/projects/large_models/gcanyon.html`.

Download the BMP format of the elevation and texture maps. We will use them along with the `vpbmaster` tool soon.



You can't use these data for commercial purposes without their permission.

How to do it...

Let us start.

1. First, we should put the downloaded BMP files in a suitable place. They can't be used for generation yet as GDAL doesn't directly support this format. So we would better convert these raster data into GeoTiff format, which allows geo-referenced information to be integrated within a TIFF file. Now open a new terminal and type the following command:

   ```
   # gdal_translate data/gcanyon_color_4k2k.bmp data/
     gcanyon_color_4k2k.png
   # gdal_translate data/gcanyon_height.bmp data/gcanyon_height.png
   ```

2. Here we assume that all terrain data are stored in the data folder of the working directory, and use relative paths to specify them.
3. Use `vpbmaster` to build our first terrain database now. Here the argument `-d` will specify the digital elevation map to use, and `-t` decides the imagery used as textures. The option `-o` determines the output directory and root filename.

   ```
   # vpbmaster -d data/gcanyon_height.png -t data/
     gcanyon_color_4k2k.png -o output/out.osgb
   ```

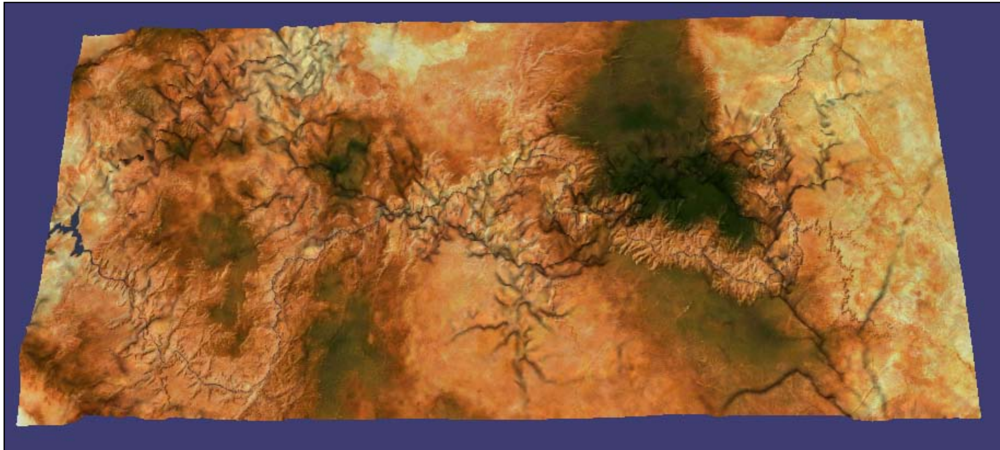
4. The generation process may take a while depending on your system, so you can just serve yourself a cup of tea while VPB is working. The output will be located at the output folder. It will be created automatically if it doesn't exist.

```
Generated tasks file = build_master.tasks
mkpath(output)
   created directory output
Revsion source = output/out.osgb.0.source
Setting up MachinePool to use all 1 cores on this machine.
Begining run
scheduling task : tasks/build_root_L0_X0_Y0.task
scheduling task : tasks/build_subtile_L2_X0_Y0.task
scheduling task : tasks/build_subtile_L2_X0_Y1.task
scheduling task : tasks/build_subtile_L2_X1_Y0.task
scheduling task : tasks/build_subtile_L2_X1_Y1.task
scheduling task : tasks/build_subtile_L2_X2_Y0.task
scheduling task : tasks/build_subtile_L2_X2_Y1.task
scheduling task : tasks/build_subtile_L2_X3_Y0.task
scheduling task : tasks/build_subtile_L2_X3_Y1.task
machine=ray-VirtualBox running task=tasks/build_root_L0_X0_Y0.task
```

5. After the building process is completed, run `osgviewer` to see the terrain model.

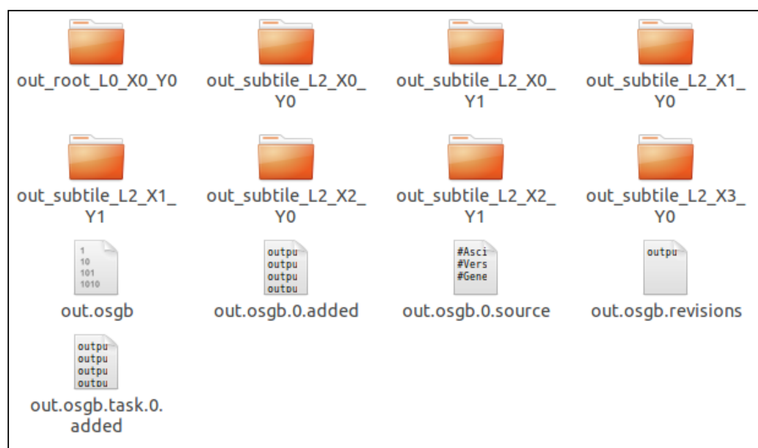
```
# osgviewer output/out.osgb
```

6. The entire output folder's size is over 150 MB, but it can be rendered and displayed smoothly. You can either view the canyon in a global perspective, or press close to one of the hills and valleys.



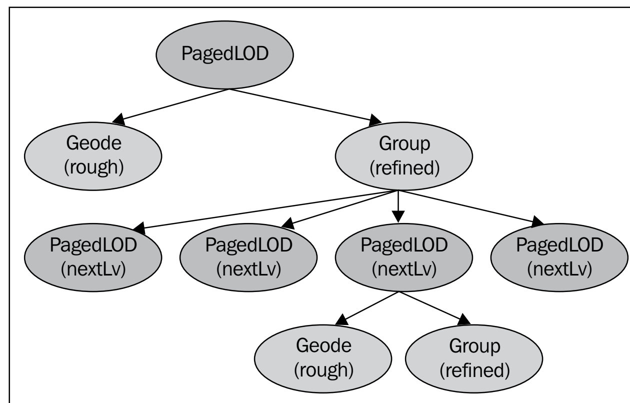
How it works...

Have a look at the generated directory. It includes a large number of files and folders with the same name infix—`L[a]_X[b]_Y[c]`. Here `a` means the level number, and `b` and `c` are range identifiers. Level `0` is the rough level, and level `6` in this example is the most detailed. In the following screenshot, there is only one **L0** subfolder and several **L2** subfolders. So what do they mean here?



VPB will always try to split the input image to some square sections, for instance, 4096 x 2048 (`L0`) will be separated into two 2048 x 2048 tiles. And they are named `L1_X0_Y0` and `L1_X1_Y0`, which lie in the `out_root_L0_X0_Y0` folder. The `L1` sections will then be split again using the quad-tree structure, that is, each level's tile is divided into four pieces of the next level. So we can see in the output folder four `L2` subfolders (`X0_Y0` to `X1_Y1`) for `L1_X0_Y0`, and the other four (`X2_Y0` to `X3_Y1`) for `L1_X1_Y0`. All child levels will be placed in these `L2` subfolders.

Thanks to OSG's paged LOD (level-of-details) mechanism, the tile will only be replaced by its four sub-tiles when the viewer is near enough, and will render data of its range with a higher resolution. The basic structure of a quad-tree LOD in terrain rendering is shown in the following diagram:



And the node structure can be described as follows:

```
osg::Group* nextLvGroup;
...
nextLvGroup->addChild( pagedNextTile1 );
nextLvGroup->addChild( pagedNextTile2 );
nextLvGroup->addChild( pagedNextTile3 );
nextLvGroup->addChild( pagedNextTile4 );
osg::PagedLOD* pagedThisTile;
...
thisTile->addChild( dataOfThisLevel );  // The rough level
thisTile->addChild( nextLvGroup );  // The refined level
```

The nodes `pagedNextTile1` to `pagedNextTile4` are actually files with the prefix `out_La'_Xb'_Yc'`. In this case:

```
a' = a + 1
For tile1: b' = 2 * b, c' = 2 * c
For tile2: b' = 2 * b + 1, c' = 2 * c
For tile3: b' = 2 * b, c' = 2 * c + 1
For tile4: b' = 2 * b + 1, c' = 2 * c + 1
```

There's more...

As .osgb is a binary native format, it is nearly impossible to quickly read and understand the contents of the generated database. We can slightly change the command-line arguments of vpbmaster to support writing to ASCII files (.osg or .osgt), as well as writing out tile images (using --image-ext to set a valid image extension) at the same time:

```
# vpbmaster -d data/gcanyon_height.png -t data/
  gcanyon_color_4k2k.png -o output/out.osg --image-ext bmp
```

By default, each tile of VPB is formed by 64 x 64 vertices and mapped by a 256 x 256 texture. As the original elevation and texture size is 4096 x 2048, VPB must build at least seven levels to get to the highest data resolution. The entire generation time may be too long and the result may not be necessary in some situations. In this case, we can control the levels to generate manually by specifying the -l parameter:

```
# vpbmaster -l 3 -d data/gcanyon_height.png -t data/
  gcanyon_color_4k2k.png -o output/out.osgb
```

Other two useful arguments are --terrain (default) and --polygonal. The option --terrain means to use the osgTerrain::TerrainTile class for generating grid geometries based on height fields, which is used by default. The option --polygonal will treat the data as triangle faces and must tessellate and simplify them while creating tiles, which is much slower and not good for further analyzing work. The two opinions can't co-exist in one terrain generation process.

Generating terrain database on the earth

The terrain we just generated comes from two simple bitmaps and is constructed with height values along the Z axis. We can say that it is computed in a projected coordinate system. It actually means that the terrain is defined on a flat, two-dimensional surface (with height). The two dimensions (x- and y-coordinates) determine the area covered by the terrain. This model can be easily understood and used in a scene, but it doesn't correspond with the geographic coordinate system or with an existing place on the earth. As far as we know, terrain data are often acquired by aircraft and satellites flying over a real region. So could we just build the data in the earth's coordinate? This kind of coordinate system is always called the **geographic coordinate system**.

Getting ready

You may either download the earth's imagery from the TrueMarble or BlueMarble website:

TrueMarble:
`http://www.unearthedoutdoors.net/global_data/true_marble/download`

BlueMarble: `http://earthobservatory.nasa.gov/Features/BlueMarble/`

We are going to make use of the free data from TrueMarble. Here is their copyright information:

"Unearthedoutdoors.net contains graphics, information, data, reviews, and other content accessible by any Internet user. All Content is owned and/or copyrighted by Unearthed Outdoors, LLC (unless otherwise explicitly noted), and may be used only in accordance with this limited use license.

Unearthed Outdoors, LLC is protected by copyright pursuant to U.S. copyright laws, international conventions, and other copyright laws."

You can't use these data for commercial purpose without the permissions from Unearthed Outdoors, LLC.

How to do it...

Let us start.

1. To build databases in geographic coordinate, we can simply use the `--geocentric` option while executing `vbpmaster`. The complete command is:

   ```
   # vpbmaster -t data/TrueMarble.4km.10800x5400.tif --geocentric
     -o output/out.osgb
   ```

   Don't doubt the arguments we used this time. Yes, there is no `-d` option and thus no elevation map specified. As we have already indicated to use the geocentric system to build from the source, VPB will automatically use flat sea-level elevation data and construct the earth geometry according to the given **GeoTiff** imagery.

2. After the generation process, you may view the terrain by calling `osgviewer`.

   ```
   # osgviewer output/out.osgb
   ```

   And you will find that the result is a six-level quad-tree structure, which simulates a realistic earth model with TrueMarble overlays, as shown in the following screenshot:

To note, the `gcanyon` data used in the last recipe is not suitable this time. Those data don't have a valid WKT (well-known text) coordinate system and must be re-projected so that VPB may then recognize them as a piece of ground in the real world.

How it works...

OSG has a special node type `osg::CoordinateSystemNode` for the viewer system to convert data between XYZ and latitude/longitude/height, and it also builds a local transition matrix internally for node transformations in the scene graph. In polygonal mode (set with the option `--polygonal`), VPB will set it as the parent node of the entire terrain sub-graph to guide the creation of all sub-tiles on the earth. But in grid mode (`--terrain`), the `osgTerrain::Terrain`, which is the derivative class, will be used instead. Both classes store the earth dimension and coordinate information. The `--geocentric` option here will indicate VPB to use the center of the earth as the terrain's center and defines units in meters directly. It is actually defined as the ECEF coordinate system, which has an ellipsoid as the measurement of the earth shape, called **World Geodetic System 1984** (WGS-84). Refer to the following link for details:

`http://en.wikipedia.org/wiki/ECEF`

There's more...

We can make use of some other coordinate systems by specifying the `--cs` option. It uses a PROJ4 format string to declare new coordinate systems, for instance:

```
# vpbmaster -cs "+proj=latlong +datum=WGS84" ...
```

More information about the coordinate system string format can be found at the following PROJ4 project website:

http://trac.osgeo.org/proj/

Working with multiple imagery and elevation data

It is impractical to put the whole earth's elevation or texture into only one file. That is because the original data may be terabyte-sized or even larger and, thus, not easy to maintain. Saving multiple images of different areas is a more suitable way, and convenient for outputting data from surveying equipment. This requires VPB to read data from multiple inputs or from a subdirectory including many smaller tiles' images, and merge them into one complete terrain model. Fortunately, this can be done directly with the `-d` and `-t` options.

Getting ready

We will continue working on the earth model and try to add some height fields at a certain longitude and latitude range. Thanks to the SRTM project, we can freely download and use the globa-elevation data along with the textures for non-commercial purposes. The download link is:

http://srtm.csi.cgiar.org/SELECTION/inputCoord.asp

Citation:

"Jarvis A., H.I. Reuter, A. Nelson, E. Guevara, 2008, Hole-filled seamless SRTM data V4, International Centre for Tropical Agriculture (CIAT), available from http://srtm.csi.cgiar.org."

You must read the disclaimer given in the following link before making use of SRTM's elevation data for any purposes:

http://srtm.csi.cgiar.org/SELECTION/SRT_disclaimer.htm

How to do it...

Let us start.

1. Let us first open the website and select a few areas we are interested in.

2. Download the elevation files found by SRTM's data search engine.

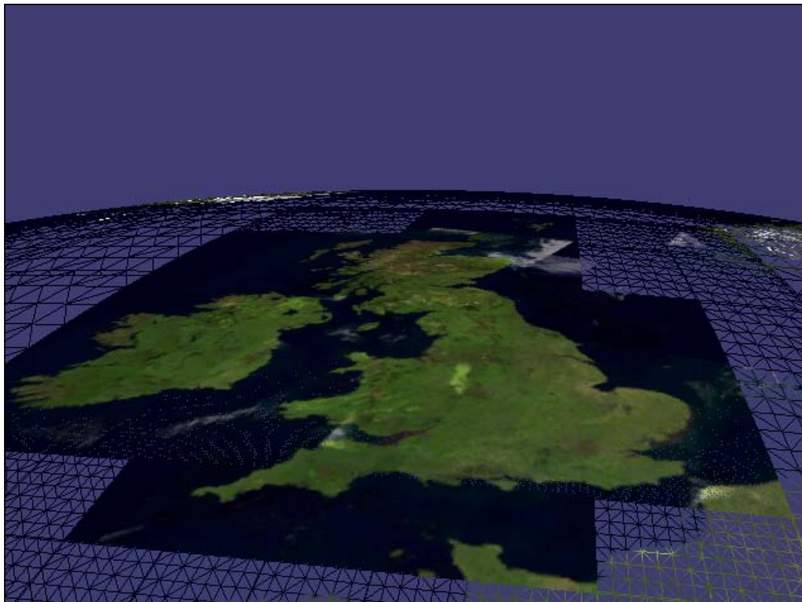
3. Put all TIFF files into a separate folder named `srtm`. Now it's time to start the `vpbmaster` tool again.

```
# vpbmaster -d data/srtm -t data/TrueMarble.4km.10800x5400.tif
  --geocentric -o output/out.osgb
```

4. View the final result with `osgviewer`. As VPB can automatically handle assemblage of multiple files, you may either specify a directory as the parameter of `-d` and `-t`, or use the same option for more than one time to add multiple files to the building process, for example:

```
# vpbmaster -d data/srtm/srtm_54_05.tif -d data/srtm/
  srtm_55_05.tif -d data/srtm/srtm_54_06.tif -d data/srtm/
  srtm_55_06.tif -t data/TrueMarble.4km.10800x5400.tif
  --geocentric -o output/out.osgb
```

5. The final result is shown in the following screenshot. You will find that there are higher-resolution elevation data around the British Isles.



How it works...

You can find in the working directory a series of new files and folders, which are created and managed by VPB. The `build_master.source` is an ASCII file wrapping up all the source data and build options. Open it with any text editor, and you will find that the file looks like a OSG native scene file (`.osgt`) and may even be loaded with the `osgDB::readNodeFile()` function. It has an `osgTerrain::TerrainTile` node to save build options (output name, extents, levels, and others) via the `vpb::DatabaseBuilder` object, and save child layers for different input data.

The `build_master.tasks` file records all the sub-tiles to be generated during the whole process. The status of each sub-tile task can be found in the tasks folder. A standard status file may be automatically written, as shown in the following code block:

```
application : osgdem --run-path /usr/local/bin -s
  build_master.source --record-subtile-on-leaf-tiles -l 8 --
  subtile 3 0 0 --task tasks/build_subtile_L3_X0_Y0.task --
  log logs/build_subtile_L3_X0_Y0.log
date : [building time]
duration : [building duration]
fileListBaseName : output\out_subtile_L3_X0_Y0/
  out_L3_X0_Y0.osgb.task.0
hostname : [host name]
pid : [pid]
source : build_master.source
status : [pending/completed]
```

A pending task indicates that the sub-tile is not created yet; and completed task means there is no need to work on the sub-tile unless the user needs a complete rebuild. If the building process is canceled, or crashes due to some system reason, you can make use of the task files' status and rerun `vpbmaster` with the `--tasks` argument:

```
# vpbmaster --tasks build_master.tasks
```

Tasks that are marked as completed will be skipped this time. Otherwise, when you execute `vpbmaster` again, the finished data will be overwritten instead of a resuming process.

Patching an existing terrain database with newer data

As you may see in Google Earth and some other 3D GIS explorers, sometimes newer and more refined images captured by satellites may be added to the entire earth model, and this helps you take a closer look at the places that are not distinct enough before. It might be important to integrate newly obtained data to the scene and distribute them to end users as soon as possible.

VPB can support patching of existing terrain database too. It requires the source file and all the original data for reference, and will add new raster and elevation data to update the database with higher resolution patches. It is extremely useful if we need to make some changes on a generated terrain model or use higher resolution images to replace the old ones.

How to do it...

To make use of the patching functionality, there are two prerequisites: first, you must have the new data; and second, you should keep the `build_master.source` file and all source files used to produce the old database, as they are needed for handling resolution and boundary problems. Sub-tiles that are not affected by the new patch will not be rebuilt anymore.

Let us start.

1. We will patch the `gcanyon` data used in the first recipe in this chapter. Of course, there are no real patches for use, so we have to create one by ourselves. Open your Photoshop or GIMP and create a new white-colored picture, then save it as a TIFF file (`sub_gcanyon_height.tif`, 1024 x 512 sized in this recipe). If we use this file as an elevation patch, it means that the height field will be set to a very high value, and old values will be totally overwritten.
2. We may have to create a world file (`sub_gcanyon_height.wld`) for specifying the resolution and range of the patch file. The content of this ASCII file can be simply written as shown in the following code block (this will be explained later):

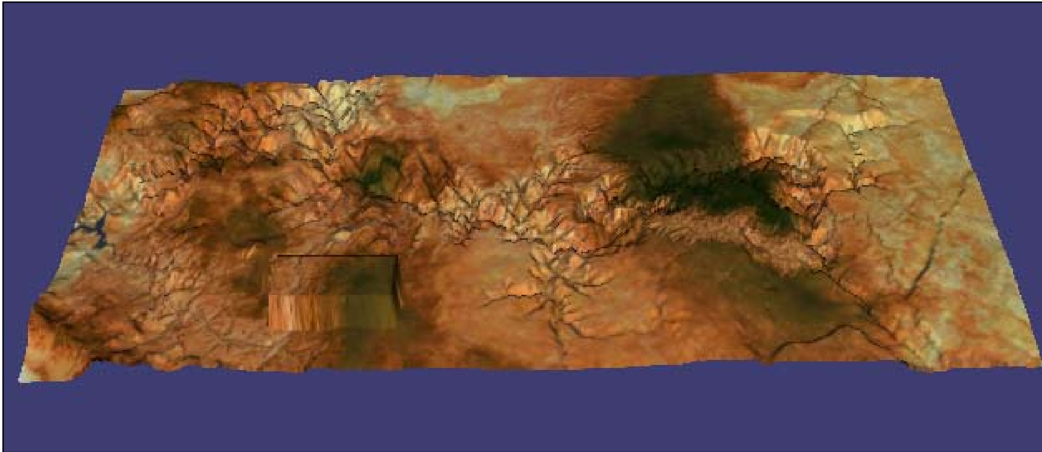
   ```
   0.5
   0.0
   0.0
   -0.5
   1000
   500
   ```

3. Now place the world file and the TIFF file together in the data directory, and start `vpbmaster`:

   ```
   # vpbmaster --patch build_master.source -d data/
     sub_gcanyon_height.tif
   ```

4. The `build_master.source` is the source file generated during the last build. The building process may take less time than creating a complete `gcanyon` terrain. As we specify the `-d` option this time, the height field will be recalculated to merge the effect of the patch.

5. Use `osgviewer` to view the output model. The white-colored elevation map is constructed as a raised cube on the terrain, as shown in the following screenshot:



How it works...

The world file is an ASCII parameter file used for geo-referencing raster map images. It was first introduced by the ESRI. This kind of files (with the extension `.tfw`, `.tifw`, or `.wld`) often describes the location, scale, and rotation of the map with six lines (each with a decimal number). When GDAL is going to read a TIFF file, it will automatically look for a world file with the same name and associate these two files together for gathering necessary information.

The meaning of the six-line parameter is as follows:

- Line 1: pixel size along X (0.5 here).
- Line 2: rotation about X (0 in most cases).
- Line 3: rotation about Y (0 in most cases).
- Line 4: pixel size along Y. It's often a negative number because image data are stored from top to bottom (-0.5 here).
- Line 5: center X of the upper-left pixel (1000 here).
- Line 6: center Y of the upper-left pixel (500 here).

Because the newly-created patch file (`sub_gcanyon_height.tif`) doesn't contain any geographic information inside, we have to provide a world file to place it at an appropriate place and with suitable resolution. If you have GDAL installed, we will find an executable named `gdalinfo`. Let us use it to check the image with the associated `.wld` file:

```
# gdalinfo data/sub_gcanyon_height.tif
```

And you will get some report, as shown in the following code block:

```
Corner Coordinates:
Upper Left  (     999.750,      500.250)
Lower Left  (     999.750,      244.250)
Upper Right (    1511.750,      500.250)
Lower Right (    1511.750,      244.250)
Center      (    1255.750,      372.250)
```

Because the patch is 1024 x 512 but only covers a 500 x 250 area, the maximized level will increase to 7. And you could see some `L7` files in part of the subfolders of output, maybe `L2_X0_Y0` and `L2_X1_Y0` in this recipe, as the new patch has intersections with them.

In fact it is common to obtain geospatial data in the GeoTIFF format instead, which already has such metadata embedded.

Building NVTT support for device-independent generation

By default, VPB uses the `osg::Image` and `osg::Texture` classes to generate compressed data formats for internal use, and create mipmaps if required. They both encapsulate OpenGL functions for implementing such work, and thus must be used on systems where a graphic card, capable of providing an OpenGL rendering context, is available. This may not be a problem in most cases, but because of the development of new compressed texture formats, there are still possibilities that your graphic cards don't afford these features, or you simply don't have a graphics card supporting OpenGL at all (for example, headless cluster or server computers). All these may lead to VPB's functionality missing and a failure to build terrain database on such machines.

Fortunately we could make use of the **NVIDIA Texture Tools** (**NVTT**), which is an open source image processing and texture manipulation project. In this recipe, we will compile and use it to configure a new OSG plugin named `osgdb_nvtt`, and make VPB depend on it to generate device-independent textures and terrain models.

Getting ready

You can download the latest NVTT source code from the following link:

`http://code.google.com/p/nvidia-texture-tools/downloads/list`

Or you can use SVN to check it out:

```
# svn checkout http://nvidia-texture-tools.googlecode.com/
  svn/branches/2.0/ nvtt
```

NVTT also provides CMake scripts for cross-platform building. But for Linux users, you can directly compile it by executing the following commands under the NVTT root directory:

```
# ./configure
# make
# make install
```

An important note for building NVTT with CMake: By default, CMake will generate makefiles for compiling static libraries, but in this way the results will not work for the corresponding OSG plugin. So you must add a definition `NVTT_SHARED=1` to force generating shared libraries while running the `cmake` executable, that is:

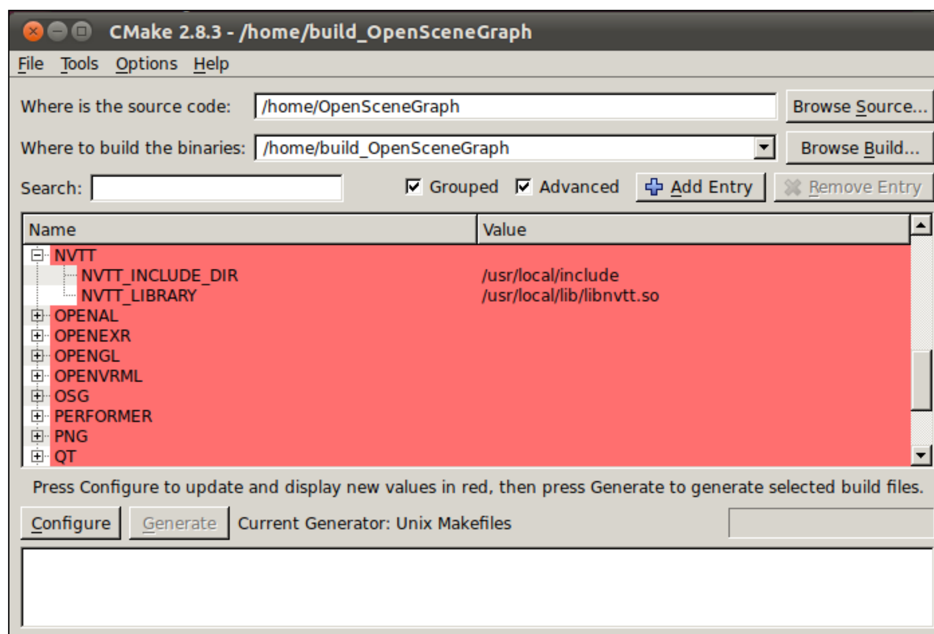
```
# cmake /home/nvtt -DNVTT_SHARED=1
```

The `cmake-gui` tool can't be used here as it doesn't allow macros to be added as arguments.

How to do it...

Let us start.

1. Now start the `cmake-gui` tool and select to configure the OpenSceneGraph directory. Find the group NVTT and set `nvcore` as the `NVTT_LIBRARY`, and the directory containing `nvtt/nvtt.h` as the `NVTT_INCLUDE_DIR` value.



2. Rebuild OpenSceneGraph now. If you have kept all the build files before, the building process will be much faster as only a few projects should be compiled.
3. Make sure to run `'make install'` and see if there is a new `osgdb_nvtt` plugin in the dynamic library's directory (`lib` for UNIX and `bin` for Windows).

   Do we have to rebuild VPB as well? The answer is absolutely not. VPB will automatically look for the NVTT plugin and make use of it for terrain generation regardless of graphics contexts.
4. Now, if you have an old enough computer, turn it on and try to run VPB to generate the `gcanyon` data again. You will see that VPB can work smoothly under such devices too.

How it works...

As we know, OSG uses the `osgDB::ReaderWriter` class as the base interface of all reader/writer plugins. Every file format is parsed within a certain plugin and then returned an `osg::Image` or `osg::Node` pointer as the result (or save to specified filename while writing scene nodes and images). For instance, COLLADA 3D models are handled by `osgdb_dae`. You will see a `ReaderWriterDAE` class defined in the DAE plugin source code which implements the concrete data reading and conversion.

But in this recipe, we meet another kind of OSG plugins—the image processor plugin. It uses a base interface called `osgDB::ImageProcessor` and implements its derived classes in plugins. This class has two important virtual methods to override:

```
virtual void compress(osg::Image&,
  osg::Texture::InternalFormatMode, bool, bool,
  CompressionMethod, CompressionQuality);
virtual void generateMipMap(osg::Image&, bool,
  CompressionMethod);
```

Re-implement them and then the processor will be able to compress images to specific formats and generate mipmaps for them. That is also what the `osgdb_nvtt` plugin does with the external NVTT library.

Using SSH to implement cluster generation

The computer cluster is a more and more common concept in modern development. It means a group of linked computers working together, and often connecting to each other through the **Local Area Network** (**LAN**). It improves the performance and availability compared with just a single computer, but of course it is much more costly.

Could we use VPB on such a cluster system and benefit from the high availability and speed? Of course. As we already know, it is not a short task to build terrain with VPB, especially when the original data are extremely large. So a computer cluster used for computational purposes can be of great help. The original data can be stored using the **Network File System** (**NFS**) technique so that all computers can get access to data stored in the same place. And there should be one primary computer which takes care of the task distribution and keeps in communication with all other slave nodes who build parts of the tiles simultaneously.

In the following section, we will mainly introduce the configuration under Linux. Windows and Mac OS X users may have troubles using the same steps. Please first read the related instructions on the OpenSSH website, and set up your own SSH environment.

How to do it...

Let us start.

1. We can use the **Secure Shell** (**SSH**) protocol to communicate with a remote computer and send commands to it. Please make sure you have had the **OpenSSH** (`http://www.openssh.com/`) service installed and enabled. Type the following command to connect to user `user1` at remote computer `192.168.1.10` (of course, they are both fictional), and execute the `vpbmaster` tool without parameters:

   ```
   # ssh user1@192.168.1.10 vpbmaster
   ```

2. If you have already installed OSG and VPB on the remote computer, the command should work. But you may have to input the password before login. VPB will also try to execute `ssh` internally while working with cluster systems, so it is important to prevent inputting the password all the time. Developers who are familiar with SSH can quickly do this by sending a public key to each remote host:

   ```
   # ssh-keygen -t rsa
   # ssh-copy-id user1@192.168.1.10
   ```

3. Create a new text file named `machinepool.txt` (or any other name) and provide all remote computer names and numbers of CPUs you want to use, as shown in the following code block:

   ```
   Machine {
     hostname user1@192.168.1.10
     processes 1
   }
   Machine {
     hostname user2@192.168.1.11
     processes 1
   }
   ```

```
Machine {
  hostname user3@192.168.1.12
  processes 1
}
...
```

4. Now let us start building the real global data with the `--machines` option (assuming they are stored in `/nfs/data`):

   ```
   # vpbmaster --machines machinepool.txt -d /nfs/data/srtm
     -t /nfs/data/TrueMarble.4km.10800x5400.tif --geocentric
     -o output/out.osgb
   ```

5. Enjoy the process. Is it a much shorter journey this time?

How it works...

Do you remember the sub-tile task files in the tasks folder? Let us open some task files randomly this time and have a look at the hostname line:

```
fileListBaseName : output\out.osgb.task.0
hostname : user1@192.168.1.10
...
fileListBaseName :
  output\out_subtile_L3_X0_Y0/out_L3_X0_Y0.osgb.task.0
hostname : user2@192.168.1.11
...
fileListBaseName :
  output\out_subtile_L3_X0_Y1/out_L3_X0_Y1.osgb.task.0
hostname : user3@192.168.1.12
...
```

You can find that VPB automatically divides the tasks and sends commands to different hosts within the local network. It's really good to see that a series of high-performance computers can cooperate so smoothly on a huge generation work. It really is a time-saving idea if you have such an environment!

There's more...

To build a NFS system, you can try the **GlusterFS** at `http://www.gluster.org/`.

And for SSH implementations under different platforms, see the OpenSSH website for details:

`http://www.openssh.com/`

Loading and rendering terrain from the Internet

VPB generated terrain tiles are so small that they can be easily transferred on the Internet or an intranet. And because of the `osgdb_curl` plugin, which depends on the **cURL** library, OSG can quickly read these files from remote servers through multiple protocols. These will be the foundation for loading and rendering terrain databases from the web.

OSG also provides a simple file-cache mechanism that writes temporary files, reads from the web to local disk, and loads the disk file directly when the same transferring request comes again. At present, it only works for paged nodes that are dynamically managed (loaded or removed due to current view point) by the `osgDB::DatabasePager` class. This solution prevents user applications from visiting remote websites and downloading unchanged data repeatedly and thus saves bandwidth and loading time.

How to do it...

Let us start.

1. It is easy to get a quick taste of rendering-terrain databases on a web server. First you should have a website to store terrain files and provide access authority to anonymous visitors. **AppServ** (`http://www.appservnetwork.com/`) might be a way to create such a site.
2. Copy all the files in the output directory to the site. They are just generated by VPB in the last recipes of this chapter.
3. Make sure the server is enabled and running. Assume the hostname is `127.0.0.1` (`localhost`), and start `osgviewer`:

   ```
   # osgviewer http://127.0.0.1/output/out.osgb
   ```

4. Now you will be able to view the database previously created. Of course it may be too simple to show the powerful web support in OSG. So this time we will try to display a larger earth model from a real remote server:

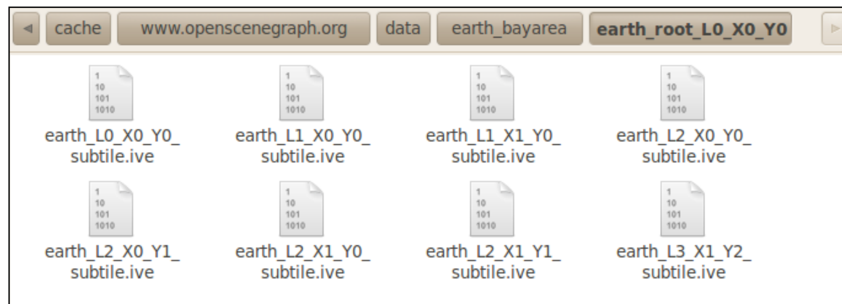
   ```
   # osgviewer http://www.openscenegraph.org/data/
     earth_bayarea/earth.ive
   ```

   This 547 MB paged database is composed of the NASA BlueMarble data and the high-resolution bay area of California, USA. You may navigate to the area to have a look at some very detailed data. And if your Internet service provider doesn't have a good bandwidth, the loading of sub-tiles may be slow, and it will be painful when you zoom in and out multiple times.

5. Now it's time to use the file cache mechanism. Just set a new environment and create a new folder for caching:

   ```
   # export OSG_FILE_CACHE = /home/cache
   # mkdir /home/cache
   ```

6. You may specify any folder as the cache folder. Make sure you have read/write permissions there.
7. Now try step 4 again. Enjoy the picture of the bay area, and exit the viewer program after a while.
8. Go to the cache folder and you will find that it records the sub-tiles you have visited, and thus makes the loading speeds of same files faster.



How it works...

The file cache is checked and reused while the database pager is processing paged nodes in request. First, it determines if a new name is a remote filename (with the hostname at the beginning of the name string). If so, it will try to find the required hostname and filename in the cache folder. If it succeeds, the pager will mark the request as 'high latency' and directly read from the local cached files later.

You can also decide if a file should be cached or not by specifying an `osgDB::FileLocationCallback` object. It has two virtual methods to override:

```
virtual Location fileLocation(const std::string& filename,
  const Options* options);
virtual bool useFileCache() const;
```

The first method will return if the file is local (`LOCAL_FILE`) or remote (`REMOTE_FILE`), and local files will not be added to the cache. The second method can quickly enable or disable the use of caching. You can at any time specify your own location callback by calling the `setFileLocationCallback()` method of the reading option object:

```
osg::ref_ptr<osgDB::Options> options = new osgDB::Options;
options->setFileLocationCallback( ownCallback );
pagedNode->setDatabaseOptions( options.get() );
```

There's more...

OSG also supports a revision mechanism that can provide adding/removing/modifying information of the remote terrain database. The database pager will then decide if the scene graph must be updated due to changes on the remote server. This functionality is not enabled by default at the time of writing this book, but you can try the `osgdatabaserevisions` example in the OSG source code to see how it works.

8 Managing Massive Amounts of Data

In this chapter, we will cover:

- Merging geometry data
- Compressing textures
- Sharing scene objects
- Configuring the database pager
- Designing simple culling strategy
- Using occlusion query to cull objects
- Managing scene objects with an octree algorithm
- Rendering point cloud data with draw instancing
- Speeding up the scene intersections

Introduction

It is more and more common to handle massive scene data in 3D programs. Terrain visualization is one common usage that converts extremely high-resolution images into grid or triangular models. We have already discussed terrain building with VPB in *Chapter 7*. However, this is not enough. Many types of unordered data, such as the geographical and geological information, point cloud from scanners, crowded people and vehicle models, and other scientific data, should also be reconstructed and rendered in 3D applications. It is not surprising if these kinds of data include millions or even billions of points and triangles. Also, we must think of some solutions to cull and render them within the limits of the operating system and hardware abilities.

In this chapter, we will introduce some common methods to optimize nodes, geometries, and textures in OSG, and provide several ways to cull scene objects, in order to reduce the number of objects before they are sent to the rendering pipeline. A spatial indexing algorithm called an **octree** is also introduced here with a very simple example.

We will add some more common functions in the `osgCookBook` namespace. They are used for generating random values for constructing huge scenes. The `randomValue()` function will return a float value between `min` and `max`. The `randomVector()` function returns a 3D vector with one component each between the `min` and `max` values. The `randomMatrix()` function will return a new 4x4 matrix, which is made up of a translation and an euler rotation operation. The translation and the angle values are randomly generated between the input `min` and `max` parameters.

```
Namespace osgCookBook {

float randomValue( float min, float max )
{
  return (min + (float)rand()/(RAND_MAX+1.0f) * (max - min));
}

osg::Vec3 randomVector( float min, float max )
{
  return osg::Vec3( randomValue(min, max),
    randomValue(min, max),
      randomValue(min, max) );
}

osg::Matrix randomMatrix( float min, float max )
{
  osg::Vec3 rot = randomVector(-osg::PI, osg::PI);
  osg::Vec3 pos = randomVector(min, max);
  return osg::Matrix::rotate(rot[0], osg::X_AXIS, rot[1],
    osg::Y_AXIS, rot[2], osg::Z_AXIS)*osg::Matrix::translate(pos);
}

}
```

Merging geometry data

It is common in a complex program to contain hundreds of thousands of geometry objects. For example, a digital city running on the local machine or the Internet may have 10,000 detailed houses, and each house can have doors, windows, fences, and many other components.

It may be sometimes confusing for developers in this situation—should we try to merge all (or majority) of them in to one `osg::Geometry` object or use more geometries to represent the different house elements? The answer depends on different situations that we may face. However, less geometry objects always perform better while rendering the same number of triangles, as the shown in the following section.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
```

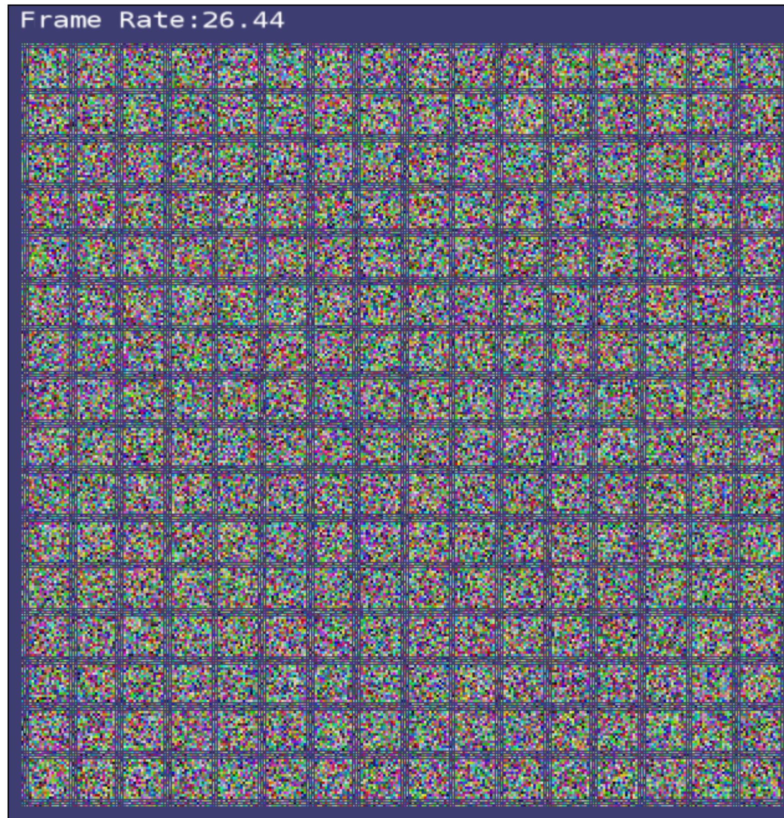
2. We will try to show the importance of merging geometries, so the first thing is to have a `createTilescreateTiles()` function to create as many geometries as possible and force the scene to be slow:

```
osg::Node* createTiles( unsigned int cols, unsigned int
  rows )
{
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  for ( unsigned int y=0; y<rows; ++y )
  {
    for ( unsigned int x=0; x<cols; ++x )
    {
      osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
      ...  // Please see the source code for details
      geode->addDrawable( geom.get() );
    }
  }
  return geode.release();
}
```

3. Now, let us try rendering these geometries:

```
osgViewer::Viewer viewer;
viewer.setSceneData( createTiles(300, 300) );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

4. Press the S key to see the frame rate. I have tested it on an Intel Dual-Core computer with a GTX 460 graphics card, which achieved an underwhelming 20FPS. Absolutely not a good performance!

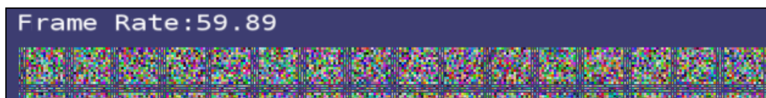


5. Rewrite the `createTiles()` function and use only one `osg::Geometry` instance to contain all the quads. Compile and check the frame rate again:

```
osg::Node* createTiles( unsigned int cols, unsigned int
  rows )
{
  unsigned int totalNum = cols * rows, index = 0;
  osg::ref_ptr<osg::Vec3Array> va = new
    osg::Vec3Array(totalNum * 4);
  osg::ref_ptr<osg::Vec3Array> na = new
    osg::Vec3Array(totalNum);
  osg::ref_ptr<osg::Vec4Array> ca  = new
    osg::Vec4Array(totalNum);

  osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
```

```
    for ( unsigned int y=0; y<rows; ++y )
    {
      for ( unsigned int x=0; x<cols; ++x )
      {
        ...  // Please see the source code for details
      }
    }

    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( geom.get() );
    return geode.release();
  }
```

6. The rendering result is not changed, but the frame rate can rise to at least 60FPS this time (60 is the refreshed frequency for most kinds of displays unless you disable the **vertical sync** feature).



How it works...

Every `osg::Drawable` object by default will generate a **display list** on the GPU side, which can speed up the rendering process of static elements. These display lists, if too many, will decrease the frame rate, and affect the rendering performance. That happens because there is always a fixed cost for the execution of a display list, no matter how much or how little work that list does. Thus, if possible, we have to merge the `osg::Geometry` objects in this recipe to reduce such cost.

Of course, merging of geometries may lose some necessary information. For example, if each of the geometries has a different texture applied, it is nearly impossible to combine them into one `osg::Geometry` object. In that case, some other solutions may have to be considered, such as dividing the scene using quadtree or octrees.

The same problem may occur if you want to use **vertex buffer object** (VBO) on geometries and don't merge them, as VBO has to create buffer data for each of the geometry's vertices and vertex attribute arrays.

Compressing texture

OpenGL provides a series of texture compression methods and new internal texture formats for faster rendering and lower memory requirements. The compression mainly boosts the speed of downloading textures into the texture memory and reduces the file size on the local disk. Also, it massively reduces the GPU memory consumption, which is one of the most important reasons to use it. This technique is used frequently in modern 3D development because it can produce high-quality results with a much smaller size on both CPU and GPU sides.

OSG has already supported a very simple way to make use of different built-in compression algorithms. In this recipe, we will again create a huge number of quads and apply random textures to them, and also show the difference between using and not using compressed textures.

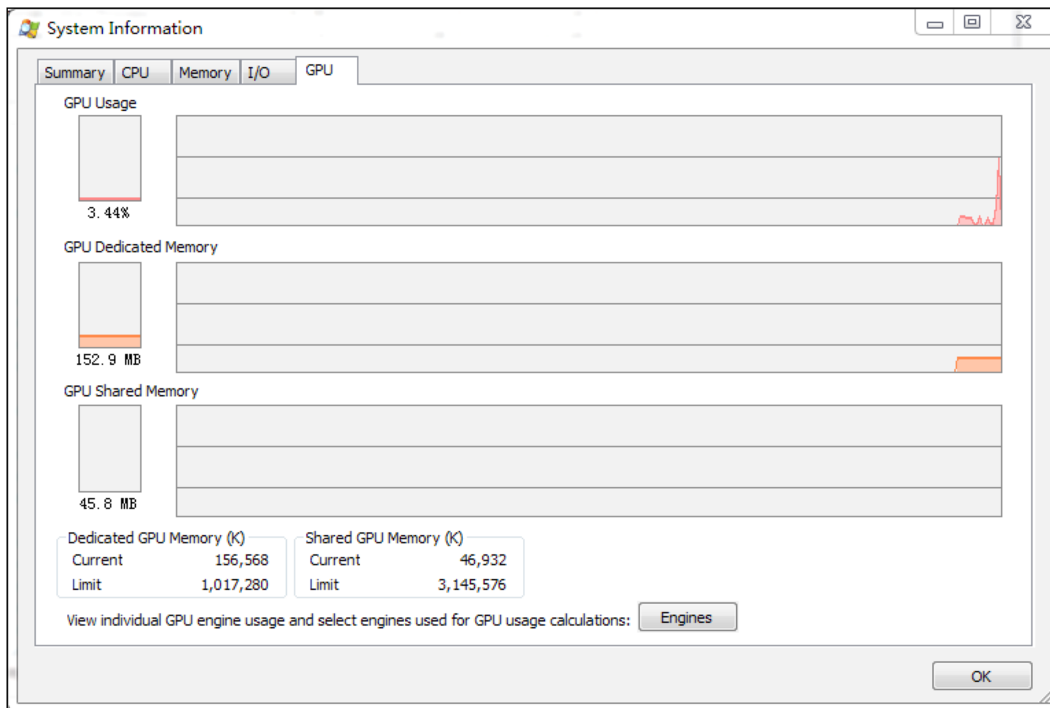
Getting ready

We will introduce the `Process Explorer` utility, which is developed by Mark Russinovich. It is an advanced process management tool that will show detailed information of a particular process and list all the DLLs it has loaded. It can also compute total and available virtual and physical memory, as well as the CPU and GPU usage on the fly. Its ability for tracking GPU usage and memory information is extremely useful here because we cannot read current GPU information directly in common ways. This functionality is not workable under non-Windows systems or versions lower than **Windows 7**.

You can download `Process Explorer` at:

`http://technet.microsoft.com/en-us/sysinternals/bb896653`

In order to view the GPU information, you must first start the `procexp.exe` executable and select **System Information** from **View** on the menu bar. Then you can change to the **GPU** tab to view current usage and available video memory if your operating system supports it, as shown in the following screenshot:



How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Texture2D>
#include <osg/Geometry>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
```

2. We will use a `createRandomImage()` function to make numerous random images for texturing, instead of loading limited numbers of existing images from the local disk.

```
osg::Image* createRandomImage( int width, int height )
{
  osg::ref_ptr<osg::Image> image = new osg::Image;
  image->allocateImage( width, height, 1, GL_RGB,
    GL_UNSIGNED_BYTE );
```

```
  unsigned char* data = image->data();
  for ( int y=0; y<height; ++y )
  {
    for ( int x=0; x<width; ++x )
    {
      *(data++) = osgCookBook::randomValue(0.0f, 255.0f);
      *(data++) = osgCookBook::randomValue(0.0f, 255.0f);
      *(data++) = osgCookBook::randomValue(0.0f, 255.0f);
    }
  }
  return image.release();
}
```

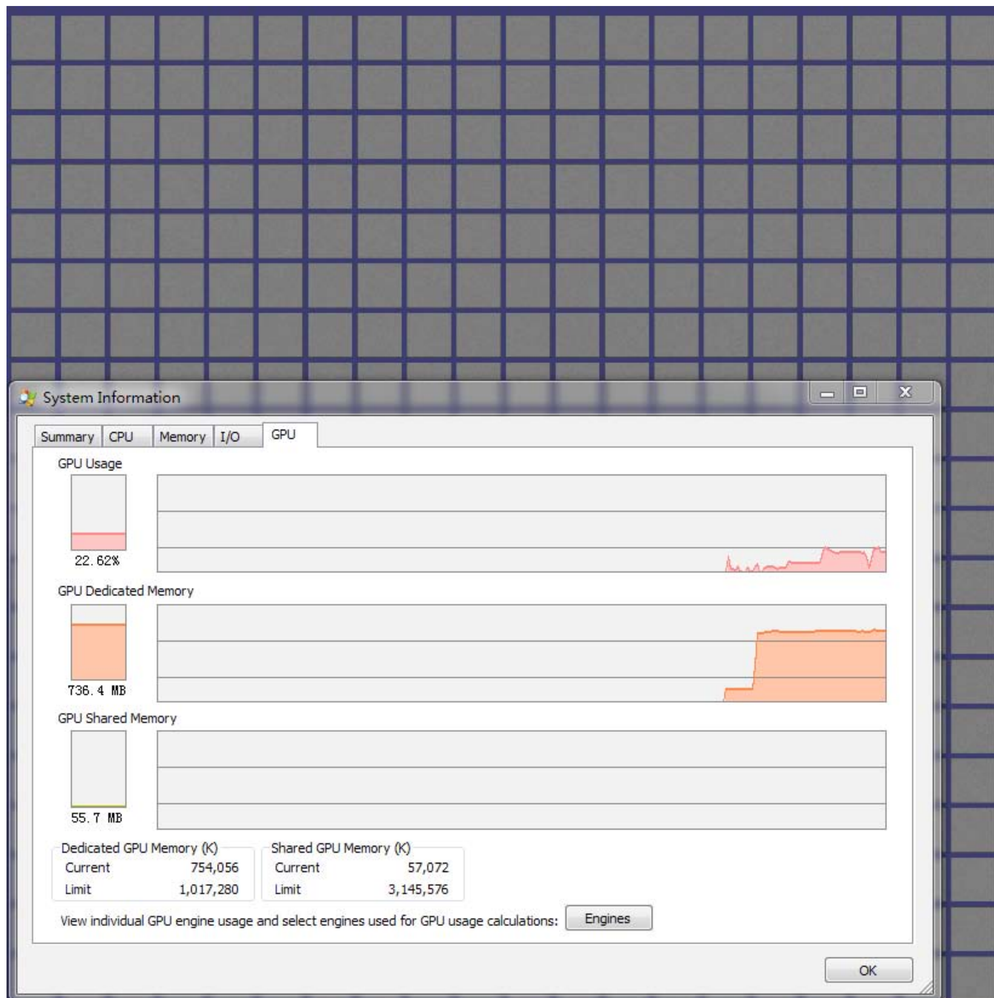
3. First, let us see how much memory is occupied without any compression methods, that is, the default `GL_RGB` format will be used and passed to the OpenGL pipeline.

```
osg::Node* createQuads( unsigned int cols, unsigned int
  rows )
{
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  for ( unsigned int y=0; y<rows; ++y )
  {
    for ( unsigned int x=0; x<cols; ++x )
    {
      osg::ref_ptr<osg::Texture2D> texture = new
        osg::Texture2D;
      texture->setImage( createRandomImage(512, 512) );

      osg::Vec3 center((float)x, 0.0f, (float)y);
      osg::ref_ptr<osg::Drawable> quad =
        osg::createTexturedQuadGeometry(
      center, osg::Vec3(0.9f, 0.0f, 0.0f), osg::Vec3(0.0f,
        0.0f, 0.9f) );
      quad->getOrCreateStateSet()->
        setTextureAttributeAndModes( 0, texture.get() );
      geode->addDrawable( quad.get() );
    }
  }
  return geode.release();
}
```

4. Now start the viewer:

```
osgViewer::Viewer viewer;
viewer.setSceneData( createQuads(20, 20) );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

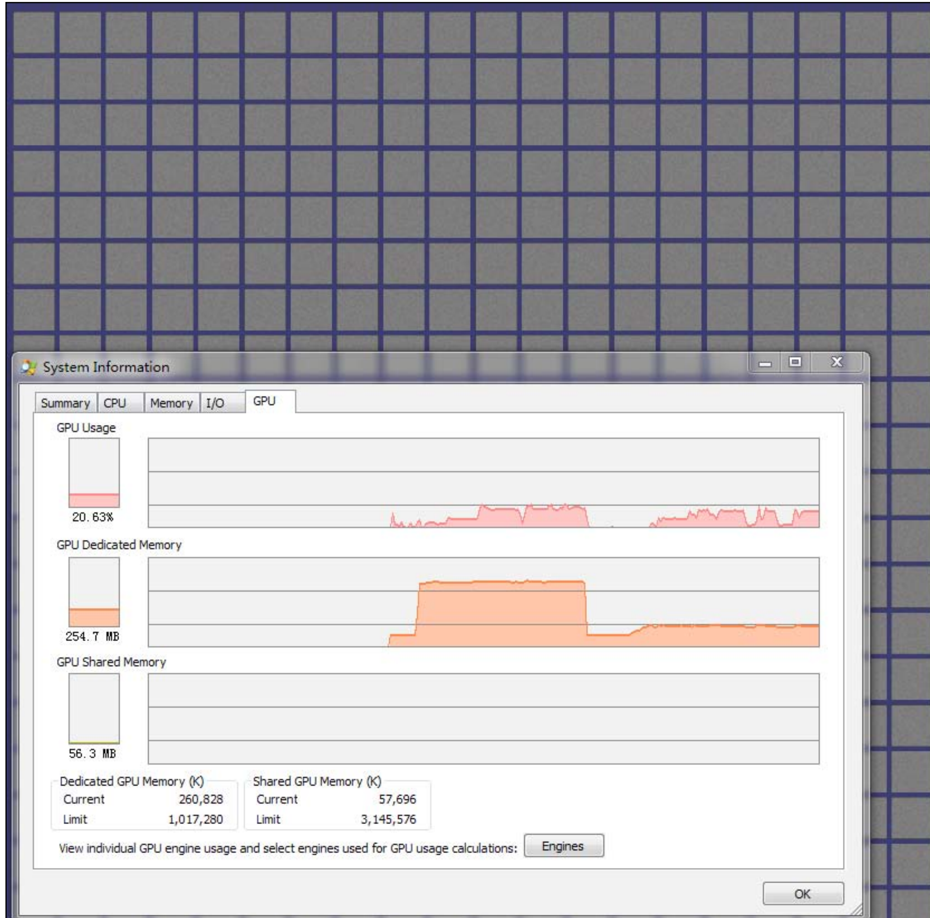
5. Open the `ProcessExplorer` and write down the available memory values on the CPU and GPU sides (~730MB on the GPU side on the author's PC).



6. Now add two lines in the `createQuads()` function, after the texture object is just created:

```
texture->setInternalFormatMode(
  osg::Texture2D::USE_S3TC_DXT1_COMPRESSION );
texture->setUnRefImageDataAfterApply( true );
```

7. Rebuild and re-run the application, and record the values in `ProcessExplorer`. You will find that both the free system memory and used graphics memory are much larger than the last time (~250MB on the GPU side). This of course saves space for further use of resources.



How it works...

It is really simple to use OpenGL-supported compressed textures in OSG applications. The `setInternalFormatMode()` method can be used to quickly specify the compression type, and OpenGL will internally do the work. For instance, in this recipe we indicate OSG to change all textures to **S3TC DXT1** format:

```
texture->setInternalFormatMode(
  osg::Texture2D::USE_S3TC_DXT1_COMPRESSION ););
```

Other supported formats include:

Internal format mode	Supported image type	Description
USE_IMAGE_DATA_FORMAT	Any	
USE_USER_DEFINED_FORMAT	Any	Let the developer decide the texture format by calling setInternalFormat().
USE_ARB_COMPRESSION	Any uncompressed	Use the ARB_texture_compression specification to compress the texture.
USE_S3TC_DXT1_COMPRESSION	RGB and RGBA	Use S3TC DXT1 compression type. http://en.wikipedia.org/wiki/S3_Texture_Compression
USE_S3TC_DXT3_COMPRESSION	RGBA only (RGB is handled by DXT1)	Use S3TC DXT3 compression type.
USE_S3TC_DXT5_COMPRESSION	RGBA only (RGB is handled by DXT1)	Use S3TC DXT5 compression type.
USE_PVRTC_2BPP_COMPRESSION USE_PVRTC_4BPP_COMPRESSION	RGB and RGBA	Use the GLES compression type: http://www.khronos.org/registry/gles/extensions/IMG/IMG_texture_compression_pvrtc.txt
USE_ETC_COMPRESSION	RGB	Use another GLES compression type: http://en.wikipedia.org/wiki/Ericsson_Texture_Compression
USE_RGTC1_COMPRESSION USE_RGTC2_COMPRESSION	RGB and RGBA	Use the OpenGL 2.0 compression type: http://www.opengl.org/registry/specs/EXT/texture_compression_rgtc.txt

This reduces the graphics card storage requirements effectively and accelerates the dynamic loading of textures, as you can see from the Process Explorer utility. However, a more optimal way for using compressed textures is to pre-compress them before rendering. Some utilities, such as **NVTT**, which is introduced in the last chapter, can produce .DDS images using compressed format and save them to disk files. These types of files can be directly recognized and used by OSG and OpenGL in user applications.

Another useful method here is setUnrefImageDataAfterApply(), as it can be set to true to force OSG to release the osg::Image objects (on the CPU side) after they are compiled into the GPU side, and thus release more system memory for other uses.

Note that as the image objects are deleted from the memory after being applied, they can never be read or written after deletion on the CPU side, and can only be retrieved by calling functions such as `glReadPixels()` or loading them from disk files again. It is suggested that you only use this feature on static textures.

Sharing scene objects

You may have already learnt some ways for sharing scene nodes, drawables, and state attributes to improve the storage and rendering performance. For example, you can make multiple mid-layer nodes, share the same child while building the scene graph, and you can also use `osgDB::SharedStateManager` to automatically collect and share texture objects of paged nodes.

In this recipe, we are going to use a share list to manage nodes read from the disk files, and cache the file reading process by comparing the filename with the names stored in this list. If a node in the share list is no longer referred to any other nodes or scene objects (that is, it is only referred by the list), it will be removed from the list to release system memory. This will be done at regular intervals (some seconds) in a 'prune' process.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <fstream>
#include <iostream>
```

2. The `osgDB::ReadFileCallback` will replace the default implementation. Hence, we can make use of this class to check if a new file reading request is already recorded in the sharing list, and use the referenced object stored instead of reading from the file again.

```
class ReadAndShareCallback : public osgDB::ReadFileCallback
{
  public:
  virtual osgDB::ReaderWriter::ReadResult readNode( const
    std::string& filename, const osgDB::Options* options );
  void prune( int second );

  protected:
  osg::Node* getNodeByName( const std::string& filename );
```

```
typedef std::map<std::string, osg::ref_ptr<osg::Node> >
  NodeMap;
NodeMap _nodeMap;
OpenThreads::Mutex _shareMutex;
```

3. The `readNode()` function must be re-implemented to take control of the file reading and sharing process. First, we check simply to see if the filename exists in the sharing list. If not, we will redirect to the default implementation to read the file from a certain plugin, and add the new name to the list; otherwise, we inform the caller that we have found an existing filename and will directly use the stored one as the return value:

```
osgDB::ReaderWriter::ReadResult
  ReadAndShareCallback::readNode( const std::string&
    filename, const osgDB::Options* options )
{
  OpenThreads::ScopedLock<OpenThreads::Mutex> lock(
    _shareMutex );
  osg::Node* node = getNodeByName( filename );
  if ( !node )
  {
    osgDB::ReaderWriter::ReadResult rr =
    osgDB::Registry::instance()->readNodeImplementation(
      filename, options );
    if ( rr.success() ) _nodeMap[filename] = rr.getNode();
    return rr;
  }
  else
  std::cout << "[SHARING] The name " << filename << " is
    already added to the sharing list." << std::endl;
  return node;
}
```

4. The `prune()` function will traverse the sharing list and check if an element has no more reference besides the sharing list itself. It should be executed every few seconds to optimize the list, but not every frame because it takes some time for each call:

```
void ReadAndShareCallback::prune( int second )
{
  if ( !(second%5) )  // Prune the scene every 5 seconds
  return;

  OpenThreads::ScopedLock<OpenThreads::Mutex> lock(
    _shareMutex );
  for ( NodeMap::iterator itr=_nodeMap.begin();
    itr!=_nodeMap.end(); )
```

```
    {
      if ( itr->second.valid() )
      {
        if ( itr->second->referenceCount()<=1 )
        {
          std::cout << "[REMOVING] The name " << itr->first
            << " is removed from the sharing list." <<
              std::endl;
          itr->second = NULL;
        }
      }
      ++itr;
    }
  }
```

5. The checking and getting of the node pointer from a filename is done in the `getNodeByName()` method.

```
osg::Node* ReadAndShareCallback::getNodeByName( const
  std::string& filename )
{
  NodeMap::iterator itr = _nodeMap.find(filename);
  if ( itr!=_nodeMap.end() ) return itr->second.get();
  return NULL;
}
```

6. We can have a `RemoveModelHandler` class to implement a picking and removing handler in the scene. Another important task is to call the pruning method in the `FRAME` event:

```
class RemoveModelHandler : public osgCookBook::PickHandler
{
public:
  RemoveModelHandler( ReadAndShareCallback* cb ) :
    _callback(cb) {}

  virtual bool handle( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& aa )
  {
    if ( ea.getEventType()==osgGA::GUIEventAdapter::FRAME )
    {
      if ( _callback.valid() )
        _callback->prune( (int)ea.getTime() );
    }
  return osgCookBook::PickHandler::handle(ea, aa);
  }
```

```
  virtual void doUserOperations(
    osgUtil::LineSegmentIntersector::Intersection& result )
  {
    ...  // Please see the source code for details
  }

  osg::observer_ptr<ReadAndShareCallback> _callback;
};
```

7. The `addFileList()` function can read model filenames and positions from an ASCII file and add the transformed node to the root. It doesn't handle the case of passing the same filename multiple times in the same file. But the `ReadAndShareCallback` will do this work instead:

```
void addFileList( osg::Group* root, const std::string& file )
{
  ...  // Please see the source code for details
}
```

8. In the main entry, we add the sharing callback to the database registry singleton. Also, read the multi-line data from the `data.txt`.

```
osg::ref_ptr<ReadAndShareCallback> sharer = new
  ReadAndShareCallback;
osgDB::Registry::instance()->setReadFileCallback(
  sharer.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
addFileList( root.get(), "files.txt" );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.addEventHandler( new
  RemoveModelHandler(sharer.get()) );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

9. The application will read multiple lines of filenames and load them. You can use *Ctrl* + left mouse button to remove models from the current scene graph.

10. The result and the terminal output are both shown in the following screenshot. Remove the line of `setReadFileCallback()` and restart the program. Open `Process Explorer` and see if there are any changes to the system memory. The more files read, the clearer result you will see:

```
[SHARING] The name cessna.osg is already added to the sharing list.
[SHARING] The name cessna.osg is already added to the sharing list.
[SHARING] The name cow.osg is already added to the sharing list.
[SHARING] The name cessna.osg is already added to the sharing list.
[REMOVING] The name dumptruck.osg is removed from the sharing list.
```

How it works...

The most important step here is to design the file reading callback, which will be used to replace the standard file reading operation. In the `readNode()` method of `ReadAndShareCallback`, we first check if the input filename is already saved in the `_nodeMap` variable, and then directly return the stored node object if there is a matched result; otherwise, the standard method will be called and the returned value will be cached for further reading requests.

The following line defines a scoped read/write lock for multithreaded developing in both `readNode()` and `prune()` methods.

```
OpenThreads::ScopedLock<OpenThreads::Mutex> lock( _shareMutex );
```

It will work if either `readNode()` or `prune()` is called, and will be automatically disabled when the method ends. While the lock is enabled, all other threads will be blocked if they are trying to execute the same two methods. Therefore, it prevents the same reading process from being called by multiple threads, and avoids possible problems and crashes.

There's more...

In fact, OSG has already provided a much simpler way to implement caching of nodes and images. For instance, we can use the `setObjectCacheHint()` method of `osgDB::Options` class to indicate that the node or image to be read should be recorded in the OSG internal registry and thus avoid replicated loading of files.
An example code segment is given as follows:

```
osg::ref_ptr<osgDB::Options> options = new osgDB::Options;
options->setObjectCacheHint( osgDB::Options::CACHE_NODES );
osg::Node* model = osgDB::readNodeFile( "cow.osg", options.get() );
```

You can use the `CACHE_IMAGES` value instead to cache images while calling the `osgDB::readImageFile()` function.

Configuring the database pager

It is not the first time we come across OSG's powerful database pager. It is actually an internal `osgDB::DatabasePager` object, which manages the loading of external files in a separate thread. It synchronizes the loaded files with the scene graph and makes them render properly in the rendering thread. It can also remove nodes that are out of view from the memory to reduce system resource consumption, and reload them when they are visible to the viewer again.

The `osg::PagedLOD` and `osg::ProxyNode` nodes, both use the database pager for implementing their underlying functionalities. The paged LOD nodes depend heavily on the pager to load and unload child levels dynamically. In this recipe, we will introduce several practical functions, which may help a lot when handling very large landscapes (possibly made up of hundreds of thousands of paged LODs).

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Texture>
#include <osg/Node>
#include <osgDB/DatabasePager>
#include <osgDB/ReadFile>
#include <osgUtil/PrintVisitor>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
```

2. We will get the scene filename to load from the argument parser. Meanwhile, we will set up the maximum texture pool size to 64000 bytes. We will explain the usage of a texture pool later:

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> root =
  osgDB::readNodeFiles(arguments);

osg::Texture::getTextureObjectManager(0)->

  setMaxTexturePoolSize( 64000 );
```

3. Every viewer object will have a default database pager. We can directly obtain it and alter its parameters. The meanings of the two methods (`setDoPreCompile()` and `setTargetMaximumNumberOfPageLOD()`) used here will be discussed later in the *How it works...* section, too:

```
osgViewer::Viewer viewer;
osgDB::DatabasePager* pager = viewer.getDatabasePager();
pager->setDoPreCompile( true);
pager->setTargetMaximumNumberOfPageLOD( 10 );
```

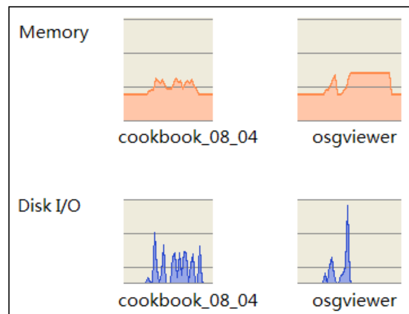
4. Now start the viewer:

```
viewer.setSceneData( root.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

5. We have to pass a filename for the recipe to load and render it. For example, you may work on the gcanyon terrain generated in the last chapter (assuming that the executable is named as `cookbook_08_04`):

`# cookbook_08_04 output/out.osgb`

6. With modern devices and full-fledged computers, you may not be able to figure out the performance difference between using this application and using `osgviewer` directly. In this case, the system memory and IO usage graphs provided in the `Process Explorer` (in the **System Information** dialog) may help you understand something. View the same terrain with this recipe's code and the `osgviewer`. Use your mouse to zoom in and zoom out the scene for more than one time. You may get graphs as shown in the following screenshot:



7. The first line shows the memory utilization of the two executables with the same paged terrain file. You can see that this recipe (left-side) has a more frequent increasing and decreasing alternation of the memory, which surely occurs when the camera is zooming in and out. This means the paged nodes of the terrain are dynamically loaded and unloaded in a more frequent way. In contrast, the result for `osgviewer` holds a constant memory usage, which means there are nearly no unloading processes during the navigation of the scene.

8. The second pair of graphs graphs record the disk I/O requests. It actually indicates how often OSG reads the files from disk, which are the child nodes of the `osg::PagedLOD` nodes. The implementation of this recipe obviously requires more I/O operations, but the other one has a higher number of I/O requests simultaneously, which may lead to frame dropping and low scene performance.

How it works...

The `setMaxTexturePoolSize()` method enables OSG's internal texture object pool, which can be used for recycling orphaned texture objects or reusing textures that are out of date. Without a texture pool, we may have to repeatedly free and allocate memories used by textures while paged nodes are loaded and unloaded with a high frequency. This can lead to memory fragmentation problems and thus slow down the system.

The texture pool partly solves this issue. It preserves a piece of system memory (64000 bytes in this recipe, but this can be customized), which can be reused at any time for texture creation requests. When we allocate space for new textures, the pool will be considered first and its available space will be split and used directly instead of asking for new ones. This feature can efficiently avoid or reduce the frame drops due to un-needed memory allocation and freeing.

The `setDoPreCompile()` is another important method that we should pay attention to, especially when there are too many objects loaded in one frame. However, all these GL objects (buffers, textures, and so on) must be compiled immediately for OpenGL to render them. It may cause terrible frame drops as the system is too busy to handle so many requests and allocations.

However, incremental `setDoPreCompile(true)` enables an incremental pre-compilation mechanism, that is, the objects will be compiled and rendered in succession during several frames instead of just one frame. So, if we have too many scene objects to compile at the same time (mostly, because of improper LOD scales or spatial index strategies), we had better use this feature to avoid excessive stalls because of hanging up the newly loaded nodes until they are compiled. Of course, more balanced scene graphs, compressed and pre-mipmapped textures, and consistent texture sizes (good for the texture pool mechanism) are always very helpful. The performance of the graphics hardware and operating system can often make a big difference too.

In the last part of this section, we will explain why the previous screenshot gives different memory and I/O monitoring results. It is the unloading and reloading of nodes from disk files that makes a difference! Thus, we can conclude that the `cookbook_08_04` executable unloads scene objects (and re-reads them) much more frequently than `osgviewer`. That is happening because of the use of the `setTargetMaximumNumberOfPageLOD()` method, which can set a maintaining target for the database pager. When the number of `osg::PagedLOD` nodes loaded into the scene is greater than the target, the pager will automatically recycle the paged LODs, as they are out-of-date or outside the view frustum. If not, it will just leave these paged nodes in the system memory for caching.

By default, the maximum number of paged nodes is 300. However, we changed it to 10 in this recipe. It means that any invisible paged nodes should be removed as soon as possible in order to fit in a very small target value, and we re-read them when they are in the range of vision again. This leads to drastic changes of memory and I/O as we have seen earlier.

Designing a simple culling strategy

Scene culling is a very important step of the scene rendering operation. In every frame, it checks the visibility of each of the geometries in the current field of vision and reduces the total number of scene objects as much as possible before sending them to the rendering pipeline. A good culling strategy makes the rendering work smooth and does not take much time.

OSG has already provided some efficient culling algorithms that can be used directly. However, sometimes we may have some easier and better solutions for some special cases. In this recipe, we will take a maze game as an example. A maze can be described as a 2D map and an extrusion about the Z axis. Hence, we can cull elements in the maze according to the 2D map. It can be simple but significant if there are massive numbers of small objects to be rendered in the maze.

How to do it...

Let us start.

1. Include necessary headers.

```
#include <osg/Texture2D>
#include <osg/Geometry>
#include <osg/ShapeDrawable>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgGA/FirstPersonManipulator>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <fstream>
#include <sstream>
#include <iostream>
```

2. The maze map is in fact a 2D table with several columns and rows. The index of an element at a certain column and row is defined with a `CellIndex`. Each table element's value can be either 0 (ground) or 1 (wall), which decides the shape of a 1x1 area in the maze:

```
typedef std::pair<int, int> CellIndex;
typedef std::map<CellIndex, int> CellMap;
CellMap g_mazeMap;
```

3. We will use the `getOrCreatePlane()` function to create a 1x1 quad on the XOY plane. It can be transformed and used to construct a ground tile in the maze, which is walkable:

```
osg::Geode* getOrCreatePlane()
{
  ...  // Please see the source code for details
}
```

4. Use the `getOrCreateBox()` function to create a 1x1x1 box. It will be transformed to construct an impassable area surrounded by walls, that is, form one of the maze "blocks". Everything placed in this area is invisible and can be culled before the rendering process:

```
osg::Geode* getOrCreateBox()
{
  ...  // Please see the source code for details
}
```

5. The next step is to create the maze according to the specified maze map information. We are going to design simple mazes using the following ASCII format:

```
1 1 1 1 1 1 0 1
1 0 0 0 0 0 0 1
1 0 1 0 1 1 0 1
1 0 1 0 1 0 0 1
1 0 1 0 1 1 1 1
1 1 1 0 0 0 0 1
0 0 0 0 1 0 1 1
1 1 1 1 1 1 1 1
```

6. The previous step generates a maze with 8x8 cells, each of which may be set to 0 or 1. The player (or the viewer) can only walk on the ground cells set to 0, and can only see objects placed on the ground. The `createMaze()` function will read the map information text from a file:

```
osg::Node* createMaze( const std::string& file )
{
  ...
}
```

7. In the function, we first open the map file and fill the `g_mazeMap` variable with read values. This variable will not only be used to create the maze geometry, but also for checking the visibility of scene objects and helping manipulate the viewer (in first-person mode):

```
std::ifstream is( file.c_str() );
if ( is )
{
```

```
    std::string line;
    int col = 0, row = 0;
    while ( std::getline(is, line) )
    {
      std::stringstream ss(line);
      while ( !ss.eof() )
      {
        int value = 0; ss >> value;
        g_mazeMap[CellIndex(col, row)] = value;
        col++;
      }
      col = 0;
      row++;
    }
  }
```

8. The second part of the `createMaze()` function is to generate maze geometries, create all ground or wall tiles, and place them at the correct columns and rows.

```
osg::ref_ptr<osg::Group> mazeRoot = new osg::Group;
for ( CellMap::iterator itr=g_mazeMap.begin();
  itr!=g_mazeMap.end(); ++itr )
{
  const CellIndex& index = itr->first;
  osg::ref_ptr<osg::MatrixTransform> trans = new
    osg::MatrixTransform;
  trans->setMatrix( osg::Matrix::translate(index.first,
    index.second, 0.0f) );
  mazeRoot->addChild( trans.get() );

  int value = itr->second;
  if ( !value )  // Ground
  trans->addChild( getOrCreatePlane() );
  else  // Wall
  trans->addChild( getOrCreateBox() );
}
return mazeRoot.release();
```

9. OSG already provides us a well-designed `osgGA::FirstPersonManipulator` that is controlled by moving the mouse (look direction) and wheel (move forwards and backwards). However, we have to rewrite it a little to make sure that the viewer can never go into impassable tiles. This is done in the derived `MazeManipulator` class:

```
class MazeManipulator : public osgGA::FirstPersonManipulator
{
  public:
  virtual bool handle( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& aa );
};
```

10. In the `handle()` function, we will first record the manipulator's unhandled matrix, do the default manipulating, and then obtain the viewer's position from the new matrix. The position will be converted to index value and passed to the maze map. If it is outside the maze or in an impassable area, roll back the viewer to the last unhandled matrix variable `lastMatrix`. The method of checking the maze cells is simpler and faster than doing intersections with scene objects directly:

```
osg::Matrix lastMatrix = getMatrix();
bool ok = osgGA::FirstPersonManipulator::handle(ea, aa);

if ( ea.getEventType()==osgGA::GUIEventAdapter::FRAME ||
   ea.getEventType()==osgGA::GUIEventAdapter::SCROLL )
{
  osg::Matrix matrix = getMatrix();
  osg::Vec3 pos = matrix.getTrans();
  if ( pos[2]!=0.5f )  // Fix the player height
  {
    pos[2] = 0.5f;
    matrix.setTrans( pos );
    setByMatrix( matrix );
  }

  CellIndex index(int(pos[0] + 0.5f), int(pos[1] + 0.5f));
  CellMap::iterator itr = g_mazeMap.find(index);
  if ( itr==g_mazeMap.end() )  // Outside the maze
  setByMatrix( lastMatrix );
  else if ( itr->second!=0 )  // Don't intersect with walls
  setByMatrix( lastMatrix );
}
return ok;
```

11. Now, in the main entry, we will create the maze from a file named `maze.txt` (you can find it in the source code directory of this book). However, this is not enough. We will then randomly add a lot of small objects (`dumptruck.osg` in this recipe, which has over 26000 points) to test the performance of the scene:

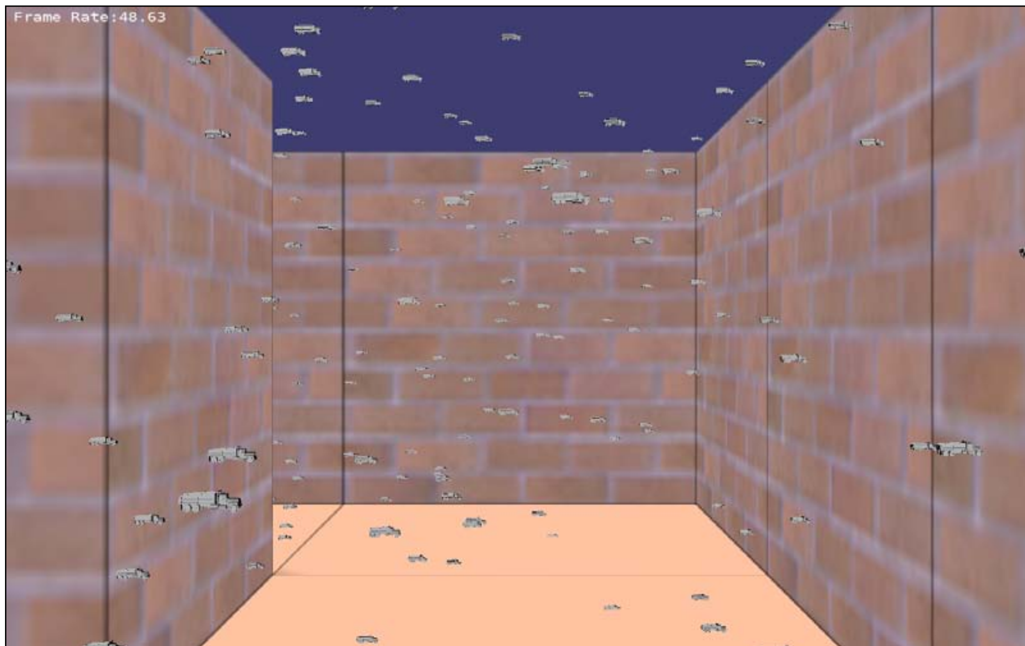
```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->getOrCreateStateSet()->setMode( GL_NORMALIZE,
  osg::StateAttribute::ON );
root->getOrCreateStateSet()->setMode( GL_LIGHTING,
  osg::StateAttribute::OFF );
root->addChild( createMaze("maze.txt") );

osg::Node* loadedModel =
  osgDB::readNodeFile("dumptruck.osg" );
for ( int i=0; i<2000; ++i )
```

```
{
  float x = osgCookBook::randomValue(0.5f, 6.5f);
  float y = osgCookBook::randomValue(0.5f, 6.5f);
  float z = osgCookBook::randomValue(0.0f, 1.0f);

  osg::ref_ptr<osg::MatrixTransform> trans = new
    osg::MatrixTransform;
  trans->setMatrix(osg::Matrix::scale(0.001, 0.001, 0.001) *
  osg::Matrix::translate(x, y, z) );
  trans->addChild( loadedModel );

  osg::ref_ptr<osg::Group> parent = new osg::Group;
  parent->addChild( trans.get() );
  root->addChild( parent.get() );
}
```

12. Create the `MazeManipulator` object and set its home position to the maze entrance. Start the viewer and have a look at the result.

```
osg::ref_ptr<MazeManipulator> manipulator = new MazeManipulator;
manipulator->setHomePosition( osg::Vec3(6.0f, 0.0f, 0.5f),
  osg::Vec3(6.0f, 1.0f, 0.5f), osg::Z_AXIS );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
viewer.setCameraManipulator( manipulator.get() );
return viewer.run();
```

13. Scroll the mouse wheel and make yourself move forward. We can see a huge number of trucks in front of you, as shown in the following screenshot. In order to display them, it will cost us lots of CPU and GPU resources. Press the S key to see the frame rate, which may be already slower than desired rate:



14. Now, it is time for us to design our own culling strategy. We will make use of the node's cull callback instead of deriving a new node type and re-implementing the `traverse()` method. It is non-intrusive and can be easily applied to most built-in types of OSG nodes:

```
class MazeCullCallback : public osg::NodeCallback
{
  public:
  virtual void operator()( osg::Node* node,
    osg::NodeVisitor* nv );

  bool getCellIndex( CellIndex& index, const osg::Vec3& pos );
};
```

15. We must re-implement the `operator()` to cull the node according to the eye position provided by the cull visitor (`nv`). It is not the first time that we derive the `NodeCallback` class to implement cull callbacks; but here we will give up traversing the node's children if the `getCellIndex()` method returns `false`. It means the eye or the node itself is not in the visible area:

```
void MazeCullCallback::operator()( osg::Node* node,
  osg::NodeVisitor* nv )
{
  osg::Vec3 eye = nv->getEyePoint();
  osg::Vec3 center = node->getBound().center();
```

```
    osg::Matrix l2w = osg::computeLocalToWorld( node->
      getParentalNodePaths()[0] );
    eye = eye * l2w; center = center * l2w;

    CellIndex indexEye, indexNode;
    if ( getCellIndex(indexEye, eye) &&
      getCellIndex(indexNode, center) )
    {
      traverse( node, nv );
    }
  }
```

16. The `getCellIndex()` method reads the position value and returns whether its corresponding index is 0 or 1 in the maze map. 0 means current position is not in a wall and can be seen by the viewer:

```
bool MazeCullCallback::getCellIndex( CellIndex& index,
  const osg::Vec3& pos )
{
  index.first = int(pos[0] + 0.5f);
  index.second = int(pos[1] + 0.5f);
  CellMap::iterator itr = g_mazeMap.find(index);
  if ( itr!=g_mazeMap.end() && itr->second==0 )
  return true;
  return false;
}
```

17. Now, when we are creating dump-trucks in the loop. Add the `MazeCullCallback` instance to each truck's `parent` node:

```
parent->setCullCallback( new MazeCullCallback );
```

18. Done! Now, recompile and see the result again, as shown in the following screenshot. You will find that the application runs much smoother than the last time and the cull time is a little longer because of the extra customized culling process:

How it works...

The `osg::NodeCallback` class, when used as cull callbacks, can determine whether a node should be culled or not by calling the `traverse()` method at an appreciated time. If the `traverse()` method is not executed in a certain condition, it means that the node will be ignored and all its children will not be visited by the cull visitor. Thus, we can design our own strategy, as shown in the following code:

```
if ( getCellIndex(indexEye, eye) && getCellIndex(indexNode,
  center) )
{
  traverse( node, nv );
}
// else ignore this node and it's subgraph.
```

In this example, we use the `getCellIndex()` method to check if a position in the maze is a wall or a ground. We can only accept the node to be rendered later when both the eye and the node center are in the ground area, and cull others to improve the performance.

Of course this algorithm can be modified to work even better. We can treat the maze walls as occluders, and check if all the line segments from the eye to one node intersect with these walls. Just figure out a solution by yourselves if you are interested in it.

Using occlusion query to cull objects

In the last recipe, we mentioned about the occluders that can be used to skip rendering objects behind them. You may already know the `osg::OccluderNode` class if you have ever read another OSG book published by Packt Publishing, that is, "*OpenSceneGraph 3.0: Beginner's Guide*", *Rui Wang* and *Xuelei Qian*. In that book, we introduced how to add convex planar occluders to the scene and make them work. This node can create highly efficient results for large scenes where only a small part is visible in each frame (others are hidden behind a few occluders).

However, the occlude node class is a software solution and can cost too much time for culling and unexpectedly decrease the rendering efficiency. Hence, is it possible for the user applications to ask the graphics hardware whether a pixel can be drawn or not? For example, any object hidden by other objects that are closer to the eye can be ignored before it is rendered to the buffer. Can the low-level 3D API check and return the query results efficiently enough?

The answer is yes. There are two possible ways for such kind of occlusion culling: **occlusion query** and **early-Z algorithm**. They both increase the rendering performance simply by not rendering geometries that are covered by other scene objects. We will introduce the first solution here, just because it can be done via the `NV/ARB_occlusion_query` OpenGL extension, and is already encapsulated in the `osg::OcclusionQueryNode` class in the core OSG library.

How to do it...

Let us start.

1. The definition of `g_mazeMap`, the creation of the maze, and the maze manipulator's implementation are just the same as what we did in the last example. Of course, this time we will not use customized callbacks again, but will use `occlusion-query` nodes instead.
2. In the main entry, the addition of 2000 dump-trucks will be done like this:

```
osg::Node* loadedModel =
  osgDB::readNodeFile("dumptruck.osg" );
for ( int i=0; i<2000; ++i )
{
  float x = osgCookBook::randomValue(0.5f, 6.5f);
  float y = osgCookBook::randomValue(0.5f, 6.5f);
  float z = osgCookBook::randomValue(0.0f, 1.0f);

  osg::ref_ptr<osg::MatrixTransform> trans = new
    osg::MatrixTransform;
  trans->setMatrix(osg::Matrix::scale(0.001, 0.001, 0.001) *
    osg::Matrix::translate(x, y, z) );
  trans->addChild( loadedModel );

  osg::ref_ptr<osg::OcclusionQueryNode> parent = new
    osg::OcclusionQueryNode;
  parent->setVisibilityThreshold( 10 );  // Ten pixels
  parent->addChild( trans.get() );
  root->addChild( parent.get() );
}
```

3. The only change is to replace the type of truck parent nodes from `osg::Group` to `osg::OcclusionQueryNode` and set the necessary threshold attributes. Now, let us start the viewer and see if there are any improvements:

How it works...

The OSG implementation of occlusion query is really simple here. You may just create a new `osg::OcclusionQueryNode` node and use it as the to-be-culled geometry's parent node. The remainder of the scene will be automatically used to check if it can completely cover any of the query nodes. A query node, if proved to be 'invisible' after the checking process, will be culled and not rendered in the current frame. In this recipe, we have 2000 dump trucks in the query list. They will be efficiently culled by the maze geometries when the application is running.

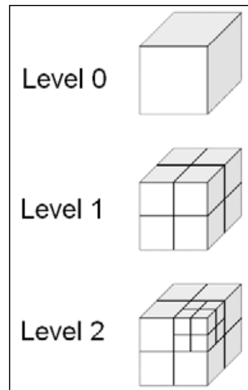
The real OpenGL's occlusion query, which is working under the hood is a little more complex. It requires us to disable writing to depth buffer, and render the query object's bounding boxes to get the computation result. The result is in fact the number of visible pixels of the current query object's bounding box that are visible. If it is greater than a predefined threshold, then we can enable writing the depth and render this object normally; otherwise it can be treated as 'invisible' and ignored. The method `setVisibilityThreshold()` records the threshold value here.

However, occlusion query is not efficient enough at present. These queries need too many additional draw calls, and the returning of query results has latency, too. It still has a long way to go before becoming the most useful culling strategy of high efficiency 3D applications.

Managing scene objects with an octree algorithm

In the last chapter, we took enough time to discuss the structure of **VPB**'s terrain models and are already familiar with its **quad-tree** scene graphs. With the help of LODs and paged LODs, we can quickly manage terrain tiles using the **quad-tree** algorithm and render unlimited size of terrain data. In fact, many other applications also use **quad-tree** to optimize the scene while working with massive data, such as city buildings, crowds of people, kinds of networks, and so on. A **quad-tree**'s internal node has exactly four children, so it is always good at handling objects placed on the XOY plane.

What should we do if we have to partition a 3-dimensional space? For example, if we have a number of balls randomly placed in the 3D world, how can we manage them using an efficient spatial indexing algorithm? One of the solutions is called **octree**. It is another tree structure whose internal node (a 3D region) has exactly eight child regions, as shown in the following diagram:



VPB uses a 2D quad-tree to structure the terrain, similarly we can use a 3D octree to structure volume data or a complex scene (for example, the solar system with massive planets and asteroids) as well. In this recipe, we will use LOD nodes to construct such an **octree** structure for rendering massive numbers of sphere elements. These spheres are located at random positions in the 3D world and can have different sizes.

OSG also implements a KDTree internally, which can speed up the intersection computation. We will introduce it in the following recipe.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/PolygonMode>
#include <osg/ShapeDrawable>
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/LOD>
#include <osgDB/ReadFile>
#include <osgUtil/PrintVisitor>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <iostream>
#include <fstream>
#include <sstream>
```

2. We will first declare an `OctreeBuilder` class for constructing a scene graph using the **octree** algorithm. It uses the `setMaxChildNumber()` method to determine how many geometries can be contained in one leaf node (default is 16), and the `setMaxTreeDepth()` method decides the maximum number of levels that the octree can have (default is 32):

```
class OctreeBuilder
{
  public:
  OctreeBuilder() : _maxChildNumber(16), _maxTreeDepth(32),
    _maxLevel(0) {}
  int getMaxLevel() const { return _maxLevel; }

  void setMaxChildNumber( int max ) { _maxChildNumber= max; }
  int getMaxChildNumber() const { return _maxChildNumber; }

  void setMaxTreeDepth( int max ) { _maxTreeDepth = max; }
  int getMaxTreeDepth() const { return _maxTreeDepth; }

  typedef std::pair<std::string, osg::BoundingBox>
    ElementInfo;
  osg::Group* build( int depth, const osg::BoundingBox&
    total, std::vector<ElementInfo>& elements );

protected:
  osg::LOD* createNewLevel( int level, const osg::Vec3&
    center, float radius );
  osg::Node* createElement( const std::string& id, const
    osg::Vec3& center, float radius );
  osg::Geode* createBoxForDebug( const osg::Vec3& max,
    const osg::Vec3& min );

  int _maxChildNumber;
  int _maxTreeDepth;
  int _maxLevel;
};
```

3. The `build()` method will be recursively called to create each level of the octree structure. User can manually call it with the depth set to 0, and total elements to specify the global boundaries and all the elements of the huge scene:

```
osg::Group* OctreeBuilder::build( int depth, const
  osg::BoundingBox& total, std::vector<ElementInfo>& elements )
{
  ...
}
```

4. We have two 3-dimensional arrays for calculating the basic attributes of a region. The `s[]` array represents all eight cells in any level of an octree. Each value in the array can be 0 or 1 to describe the side (left or right) of the cell on three of the axes (X/Y/Z). The `extentSet[]` array records the minimum, average, and maximum points in a level's region, which will be used later for calculating the region of its children:

```
int s[3];  // axis sides (0 or 1)
osg::Vec3 extentSet[3] = {
  total._min,
  (total._max + total._min) * 0.5f,
  total._max
};
```

5. The `elements` variable contains all the elements in the scene, and hence we have to find out, which of those really intersect with current region `total` and save them to a temporary list `childData`. If elements in current region are few enough to form a leaf node, set `isLeafNode` to `true`; otherwise set it to `false` to go on splitting the space into eight children of the next level:

```
std::vector<ElementInfo> childData;
for ( unsigned int i=0; i<elements.size(); ++i )
{
  const ElementInfo& obj = elements[i];
  if ( total.contains(obj.second._min) &&
    total.contains(obj.second._max) )
      childData.push_back( obj );
  else if ( total.intersects(obj.second) )
  {
    osg::Vec3 center = (obj.second._max + obj.second._min) * 0.5f;
    if ( total.contains(center) ) childData.push_back( obj );
  }
}

bool isLeafNode = false;
if ( (int)childData.size()<=_maxChildNumber ||
  depth>_maxTreeDepth ) isLeafNode = true;

osg::ref_ptr<osg::Group> group = new osg::Group;
if ( !isLeafNode )
{
  ...
}
else
{
  ...
}
```

6. If `isLeafNode` is false, we will have to set up the region box of eight new child regions of the next level. These child regions are created using `osg::Group` and added to a parent group node. The `build()` method will be called recursively with different region parameters to check and build the sub-graphs for them:

```
osg::ref_ptr<osg::Group> childNodes[8];
for ( s[0]=0; s[0]<2; ++s[0] )
{
  for ( s[1]=0; s[1]<2; ++s[1] )
  {
    for ( s[2]=0; s[2]<2; ++s[2] )
    {
      osg::Vec3 min, max;
      for ( int a=0; a<3; ++a )
      {
        min[a] = (extentSet[s[a] + 0])[a];
        max[a] = (extentSet[s[a] + 1])[a];
      }

    int id = s[0] + (2 * s[1]) + (4 * s[2]);
    childNodes[id] = build( depth+1, osg::BoundingBox
      (min, max), childData );
    }
  }
}

for ( unsigned int i=0; i<8; ++i )
{
  if ( childNodes[i] && childNodes[i]->getNumChildren() )
    group->addChild( childNodes[i] );
}
```

7. If the current region is available as a **leaf** of the octree, we can simply call `createElement()` to generate the sphere and set the necessary parameters for rendering it. The renderable element will be added to the group node representing the leaf node of the octree:

```
for ( unsigned int i=0; i<childData.size(); ++i )
{
  const ElementInfo& obj = childData[i];
  osg::Vec3 center = (obj.second._max + obj.second._min) * 0.5;
  float radius = (obj.second._max -
    obj.second._min).length() * 0.5f;
  group->addChild( createElement(obj.first, center, radius) );
}
```

8. The last step of the `build()` method is to use an `osg::LOD` node to finish the construction of the current level. It displays a rough level of details, which only contains a debug box (or may contain nothing) when the viewer's eye is still far away. If the viewer is near enough, it turns to the second child, which may either contain eight child nodes or may be used as a leaf node with a few final spheres (determined by `_maxChildNumber`):

```
osg::Vec3 center = (total._max + total._min) * 0.5;
float radius = (total._max - total._min).length() * 0.5f;
osg::LOD* level = createNewLevel( depth, center, radius );
level->insertChild( 0, createBoxForDebug(total._max,
  total._min) );  // For debug use
level->insertChild( 1, group.get() );
return level;
```

9. The `createNewLevel()` method is used for creating a customized LOD node:

```
osg::LOD* OctreeBuilder::createNewLevel( int level, const
  osg::Vec3& center, float radius )
{
  osg::ref_ptr<osg::LOD> lod = new osg::LOD;
  lod->setCenterMode( osg::LOD::USER_DEFINED_CENTER );
  lod->setCenter( center );
  lod->setRadius( radius );
  lod->setRange( 0, radius * 5.0f, FLT_MAX );
  lod->setRange( 1, 0.0f, radius * 5.0f );

  if ( _maxLevel<level ) _maxLevel = level;
  return lod.release();
}
```

10. The `createElement()` method creates a renderable sphere and returns it.

```
osg::Node* OctreeBuilder::createElement( const std::string&
  id, const osg::Vec3& center, float radius )
{
  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( new osg::ShapeDrawable(new
    osg::Sphere(center, radius)) );
  geode->setName( id );
  return geode.release();
}
```

11. The `createBoxForDebug()` method will create a wire-frame box, which can represent the region's bounding box. It is drawn only for debug purposes here:

```
osg::Geode* OctreeBuilder::createBoxForDebug( const
  osg::Vec3& max, const osg::Vec3& min )
{
  ...  // Please see source code for details
}
```

12. We will also implement a scene graph printing visitor that can write out the scene graph structure and leaf spheres' names to disk files. It is enough to only derive it from the `osgUtil::PrintVisitor` class:

```
class PrintNameVisitor : public osgUtil::PrintVisitor
{
  public:
  PrintNameVisitor( std::ostream& out ) :
    osgUtil::PrintVisitor(out) {}

  void apply( osg::Node& node )
  {
    if ( !node.getName().empty() )
    {
      output() << node.getName() << std::endl;
      enter();
      traverse( node );
      leave();
    }
    else osgUtil::PrintVisitor::apply(node);
  }
};
```

13. We are nearly done. Now, in the main entry, we add 5000 spheres with different positions and radii to the `globalElements` variable. The global bounding box is computed at the same time. After that, we call the `build()` method to create the top-level of the octree graph:

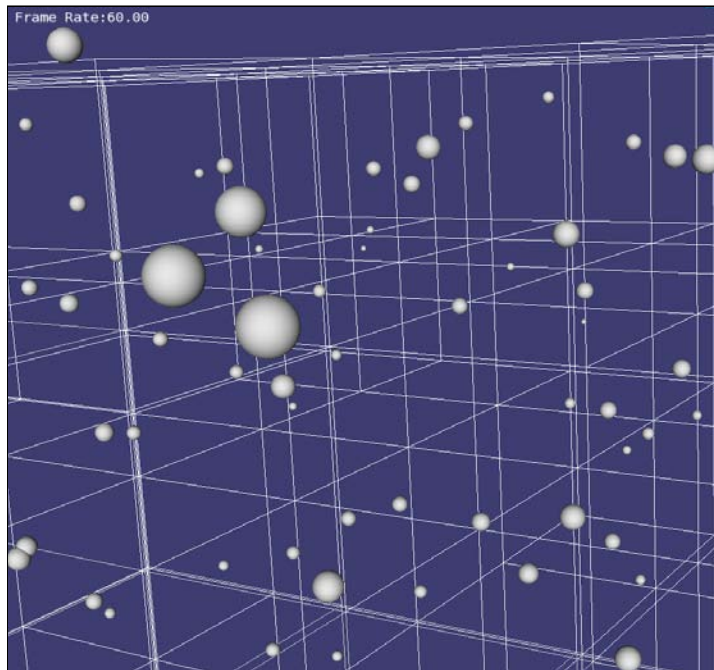
```
osg::BoundingBox globalBound;
std::vector<OctreeBuilder::ElementInfo> globalElements;
for ( unsigned int i=0; i<5000; ++i )
{
  osg::Vec3 pos = osgCookBook::randomVector( -500.0f, 500.0f );
  float radius = osgCookBook::randomValue( 0.5f, 2.0f );
  std::stringstream ss; ss << "Ball-" << i+1;

  osg::Vec3 min = pos - osg::Vec3(radius, radius, radius);
  osg::Vec3 max = pos + osg::Vec3(radius, radius, radius);
```

```
    osg::BoundingBox region(min, max);
    globalBound.expandBy( region );
    globalElements.push_back( OctreeBuilder::ElementInfo(ss.str(),
      region) );
  }

  OctreeBuilder octree;
  osg::ref_ptr<osg::Group> root = octree.build( 0, globalBound,
    globalElements );
```

14. Print the generated scene graph to an ASCII file and start the viewer to render the huge scene:

```
std::ofstream out("octree_output.txt");
PrintNameVisitor printer( out );
root->accept( printer );

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

15. When the application is started, you can see only a box in the view field. Zoom in the camera and you can find the box is divided into smaller ones. Zoom in again, and the spheres in the leaf nodes will be shown when you are close enough to these spheres, as shown in the following screenshot:



How it works...

Let's open the outputted file (which is written out after the application ran once) and paste a part of it here:

```
osg::LOD
  osg::Geode
  osg::Group
    osg::LOD
      osg::Geode
      osg::Group
        Ball-438
        …
    osg::LOD
      osg::Geode
      osg::Group
        Ball-729
        …
    osg::LOD
    …
```

The node named `Ball-*` are random spheres that have to be rendered in the scene. As we can see from the preceding code, ball nodes are stored in group nodes (leaf nodes of the octree), and group nodes are added as the finer level of LOD nodes. The LOD nodes have customized centers and radii, and will decide when the child leaves can be shown according to the distance between the node center and the eye.

Every eight LOD nodes in the same level will be held together in a group node, and used as the finer level of a parent LOD node. This is actually the basic structure of the octree. The rough levels of all LODs are always represented by wire-frame boxes (`osg::Geode`).

There's more...

You may find that integrating the scene graph with an indexing algorithm like a **quad-tree** and **octree** is not a very complex procedure. In this recipe, we only used the `osg::LOD` node to manage different levels of the tree, but it is recommended to replace them with `osg::PagedLOD` to provide paging functionality for huge scene rendering, just like the **VPB** utility has done to terrain database.

You may be interested in some other spatial indexing methods and their introduction links, including:

- **Binary space partitioning** (**BSP**): Navigate to the following URL for more infromation:

  `http://en.wikipedia.org/wiki/Binary_space_partitioning`

- **K-dimensional tree** (**KDTree**): It is used internally by OSG' s intersection visitor. Navigate to the following URL for more infromation:

  `http://en.wikipedia.org/wiki/K-d_tree`

- **R-Tree**: Navigate to the following URL for more infromation:

  `http://en.wikipedia.org/wiki/R-tree`

Try to implement one or more of them along with the scene graph structure. You can use them either for culling objects before rendering, or for speeding up the intersection work between scene objects and a line segment (or some other operators).

Rendering point cloud data with draw instancing

Point cloud data is widely used in scientific fields. It is formed by a huge number of points, including the positions, normals, colors, and other attributes. Point cloud is usually generated using specific scanners (for example, laser, structured light, and so on) and can describe the surface of any complex objects. There are also many solutions for 3D model reconstruction using point cloud as the source. But the first problem we will face is—how to display them?

You might use a geometry to contain all the points and render them in `GL_POINTS` mode, but the frame rate may drop seriously if the number is too large. Fortunately, we have the draw instancing extension, which may help improve performance. In *Chapter 3*, we have already provided a recipe for a drawing instance. In this recipe, we will make use of it again and convert the sample point cloud data to texture and handle them in the shaders.

The cloud rendering example in *Chapter 6*, has a `data.txt` file, which can be directly used as the data source.

How to do it...

Let us start.

1. Include necessary headers.

```
#include <osg/Point>
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <fstream>
#include <iostream>
```

2. We have already learnt that `gl_InstanceID` can be used to indicate a specific instance of the same drawable object. In order to visualize the sample data, we will have to place these instances at different positions for representing every point element in the point cloud. Thus, we will use the `defaultTex` variable to pass the texture object to the shader and use texels to describe point locations:

```
const char* vertCode = {
  "uniform sampler2D defaultTex;\n"
  "uniform int width;\n"
  "uniform int height;\n"
  "varying float brightness;\n"
  "void main()\n"
  "{\n"
  "    float row = float(gl_InstanceID) /
        float(width);\n"
  "    vec2 uv = vec2(fract(row), floor(row) /
        float(height));\n"
  "    vec4 texValue = texture2D(defaultTex, uv);\n"
  // Read and specify the position data from texture
  "    vec4 pos = gl_Vertex + vec4(texValue.xyz, 1.0);\n"
  // Use alpha of the texel as the brightness value
  "    brightness = texValue.a;\n"
  "    gl_Position = gl_ModelViewProjectionMatrix * pos;\n"
  "}\n"
};
```

3. The fragment shader will be used to draw the color of points according to the `brightness` parameter read from the alpha component of the texture:

```
const char* fragCode = {
  "varying float brightness;\n"
  "void main()\n"
  "{\n"
  "    gl_FragColor = vec4(brightness, brightness,
    brightness, 1.0);\n"
  "}\n"
};
```

4. Now, we will create the instanced geometry in the `createInstancedGeometry()` function. It uses an `osg::Image` object to record point cloud data:

```
osg::Geometry* createInstancedGeometry( osg::Image* img,
  unsigned int numInstances )
{
  ...
}
```

5. In the function, we first create a geometry with only one point. It should use the `numInstances` parameter to enable using the OpenGL draw instancing extension, and disable display lists for the purpose of dynamic modification:

```
osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
geom->setUseDisplayList( false );
geom->setUseVertexBufferObjects( true );
geom->setVertexArray( new osg::Vec3Array(1) );
geom->addPrimitiveSet( new osg::DrawArrays(GL_POINTS, 0, 1,

  numInstances) );
```

6. Then we will set the image object as the texture of the geometry. We will add necessary uniforms and the program object to the state set in order to make shaders work properly:

```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( img );
texture->setInternalFormat( GL_RGBA32F_ARB );
texture->setFilter( osg::Texture2D::MIN_FILTER,
  osg::Texture2D::LINEAR );
texture->setFilter( osg::Texture2D::MAG_FILTER,
  osg::Texture2D::LINEAR );
geom->getOrCreateStateSet()->setTextureAttributeAndModes
  ( 0, texture.get() );
geom->getOrCreateStateSet()->addUniform( new
  osg::Uniform("defaultTex", 0) );
geom->getOrCreateStateSet()->addUniform( new
  osg::Uniform("width", (int)img->s()) );
geom->getOrCreateStateSet()->addUniform( new
  osg::Uniform("height", (int)img->t()) );

osg::ref_ptr<osg::Program> program = new osg::Program;
program->addShader( new osg::Shader(osg::Shader::VERTEX,
  vertCode) );
program->addShader( new osg::Shader(osg::Shader::FRAGMENT,
  fragCode) );
geom->getOrCreateStateSet()->setAttributeAndModes
  ( program.get() );
return geom.release();  // end of createInstancedGeometry()
```

7. We will implement one more function that is the `readPointData()` method. It will be used to load point data from an ASCII file and save them in an image with specified size (`w` and `h`):

```
osg::Geometry* readPointData( const std::string& file,
  unsigned int w, unsigned int h )
{
  ...
}
```

8. The image must use the `GL_RGBA` and `GL_FLOAT` format. We will discuss the reason why we should use this format later in the *How it works...* section. The position and brightness values read from the text file will be set to the data pointer of the image:

```
std::ifstream is( file.c_str() );
if ( !is ) return NULL;

osg::ref_ptr<osg::Image> image = new osg::Image;
image->allocateImage( w, h, 1, GL_RGBA, GL_FLOAT );

unsigned int density, brightness;
osg::BoundingBox boundBox;
float* data = (float*)image->data();
while ( !is.eof() )
{
  osg::Vec3 pos;
  is >> pos[0] >> pos[1] >> pos[2] >> density >> brightness;
  boundBox.expandBy( pos );

  *(data++) = pos[0];
  *(data++) = pos[1];
  *(data++) = pos[2];
  *(data++) = brightness / 255.0;
}
```

9. We will create the geometry and accept an initial bounding box to make it visible in the scene; otherwise such a geometry object with only one point will be automatically culled while rendering.

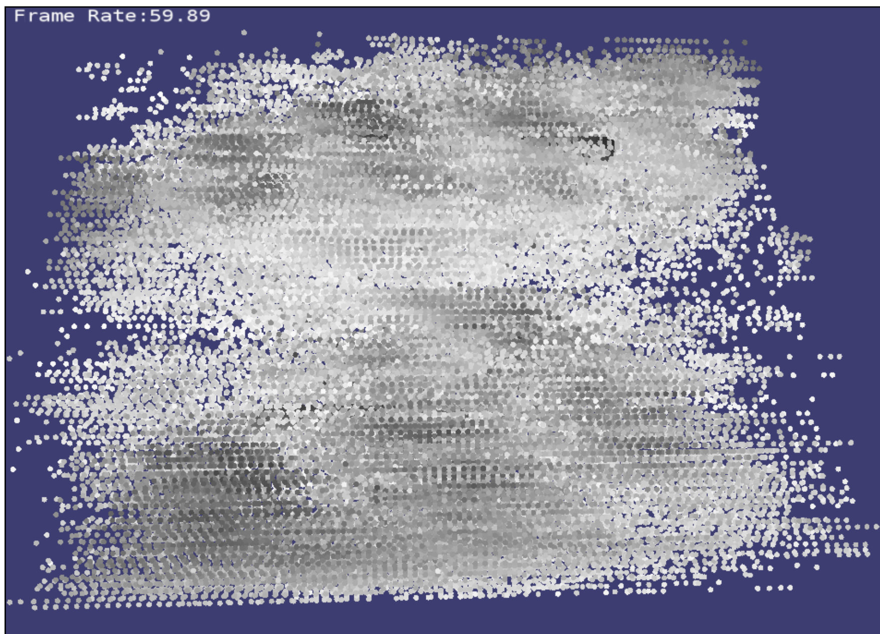
```
osg::ref_ptr<osg::Geometry> geom = createInstancedGeometry
  ( image.get(), w*h );
geom->setInitialBound( boundBox );
geom->getOrCreateStateSet()->setAttributeAndModes
  ( new osg::Point(5.0f) );
return geom.release();
```

10. Now in the main entry, we will add the draw instancing geometry to the scene graph and render it in the viewer. A 512x512 sized image is enough here to contain all the points in the sample `data.txt` file (from the cloud rendering example in *Chapter 6*) which includes about 64000 lines:

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( readPointData("data.txt", 512, 512) );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );
```

```
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

11. The result is shown in the following screenshot. It renders smoothly and doesn't need too much system memory on the CPU side. You can make any changes or improvements to the rendered data by altering the shader code. You can also modify the primitive set of the origin geometry object to change the shapes of all points or apply textures on them:



How it works...

The most important step in this recipe is to add the point cloud data to the image object, which should be used in the shader code for efficiently rendering the points. The `osg::Image` class uses the `allocateImage()` method to create an empty image object, and provides the `data()` method for developers to set up image pixels. We assume that the point cloud requires to record only the position and brightness of each element, so it is enough to have four float components (pixel format `GL_RGBA`, datatype `GL_FLOAT`) for each pixel. The first three components save the XYZ values of the point, and the last one saves the brightness parameter.

The draw instancing extension is not all that powerful. The size of `texture` object for keeping point cloud data cannot be infinite. If we have millions or even billions of points to display, the current solution will not be suitable for it. In that case, some spatial index methods can be used as well. For instance, the octree algorithm, which was introduced in this chapter, will be a good idea here. We can first pre-treat all the point data and divide them into different smaller areas. Each area uses draw instancing to render actual point attributes. Then we can use the octree algorithm along with the paged LOD nodes to manage these 'leaf' areas, guaranteeing that there are not too many points shown at the same time.

Speeding up the scene intersections

There is a common demand while developing huge scene viewing applications, such as when we move the mouse onto the terrain or other scene objects, we hope that we can immediately see the current intersection result of the mouse coordinate and the 3D objects. The intersection result is useful for practical use, for example, describing user's point of interest on the earth or in a digital city.

As OSG uses bounding volumes to manage scene graphs, the computation process can be much quicker than intersecting with all scene objects one by one. However, we still have room for speeding up the process. OSG internally provides the **KDTree** structure for scene intersections with geometries, which can help do the calculations and return results in a faster way.

Another potential problem to be considered here is that the results may not be accurate enough while intersecting with paged scene. That's because paged nodes are loaded due to the current viewer's position and attitude, and the highest level may not be reachable by the intersection visitor at any time. We will discuss a solution for this issue in this recipe.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Group>
#include <osgDB/ReadFile>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <sstream>
#include <iostream>
```

2. The `PagedPickHandler` class is designed to calculate the intersections of current cursor coordinate and the scene graph when the mouse is moving. It refreshes the result intersection point to an HUD text object and can accept a reading callback for obtaining the highest levels of paged nodes:

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

  osg::ref_ptr<osgUtil::IntersectionVisitor::ReadCallback>
    _pagedReader;
  osg::ref_ptr<osgText::Text> _text;
};
```

3. In the `handle()` method, we will do the computations for all user events except the `FRAME` event:

```
if ( ea.getEventType()==osgGA::GUIEventAdapter::FRAME )
  return false;

osgViewer::View* viewer = dynamic_cast<osgViewer::View*>(&aa);
if ( viewer && _text.valid() )
{
  ...
}
```

4. The `osg::Timer` object is used for timekeeping. We will retrieve a time value and then use the intersection visitor to traverse the whole scene. The `_pagedReader` callback, if valid, will be applied to the `setReadCallback()` method here for handling paged nodes. We will show you how to quickly implement such a new callback later, but at present, we will leave this variable to `NULL`.

```
osg::Timer_t t1 = osg::Timer::instance()->tick();
osg::ref_ptr<osgUtil::LineSegmentIntersector> intersector =
  new osgUtil::LineSegmentIntersector
    (osgUtil::Intersector::WINDOW, ea.getX(), ea.getY());
osgUtil::IntersectionVisitor iv( intersector.get() );
iv.setReadCallback( _pagedReader.get() );
viewer->getCamera()->accept( iv );
```

5. If there is any result that is, the mouse has an intersection point with the scene in world space. Then we will obtain the intersection result and record the time again. The difference between the two time variables is the time spent for scene intersections. We will print all these this information on the screen in real-time:

```
if ( intersector->containsIntersections() )
{
  osgUtil::LineSegmentIntersector::Intersection result =
    *(intersector->getIntersections().begin());
  osg::Vec3 point = result.getWorldIntersectPoint();
  osg::Timer_t t2 = osg::Timer::instance()->tick();

  std::stringstream ss;
  ss << "X = " << point.x() << "; ";
  ss << "Y = " << point.y() << "; ";
  ss << "Z = " << point.z() << "; ";
  ss << "Delta time = " << osg::Timer::instance()->
    delta_m(t1, t2) << "ms" << std::endl;
  _text->setText( ss.str() );
}
```

6. In the main entry, we will use the `setBuildKdTreesHint()` method to enable building **KDTree** structure for all scene objects. Then we are going to construct the scene graph with the loaded model and an HUD camera displaying the text:

```
osg::ArgumentParser arguments( &argc, argv );
osgDB::Registry::instance()->setBuildKdTreesHint(
  osgDB::Options::BUILD_KDTREES );
osg::ref_ptr<osg::Node> loadedModel =
  osgDB::readNodeFiles(arguments);

osgText::Text* text =
  osgCookBook::createText(osg::Vec3(50.0f, 50.0f, 0.0f),
    "", 10.0f);
osg::ref_ptr<osg::Geode> textGeode = new osg::Geode;
textGeode->addDrawable( text );

osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0, 800, 0, 600);
hudCamera->addChild( textGeode.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( hudCamera.get() );
root->addChild( loadedModel.get() );
```

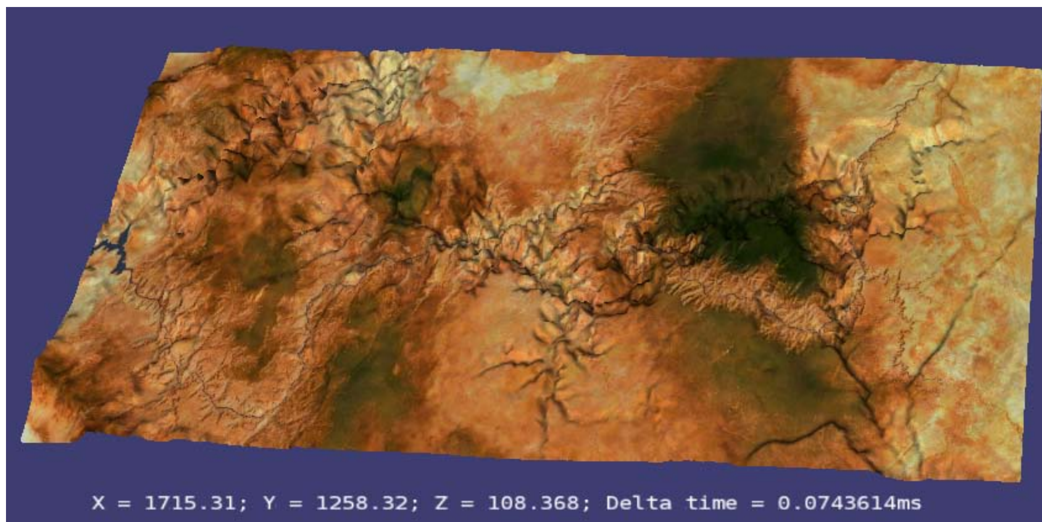
7. The `PagedPickHandler` will be then allocated and added to the viewer for intersection operations:

```
osg::ref_ptr<PagedPickHandler> picker =
  new PagedPickHandler;
picker->_text = text;

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.addEventHandler( picker.get() );
viewer.addEventHandler( new osgViewer::StatsHandler );
return viewer.run();
```

8. We hope you kept the terrain models generated in the last chapter. The gcanyon data is a good sample for testing the picking handler here. We can execute this recipe (the executable is named as `cookbook_08_09`) using the following command:

```
# cookbook_08_09 output/out.osgb
```

9. The computation time is about 0.06-0.09 milliseconds on the author's computer. The output generated is shown in the following screenshot:



10. Now delete the line enabling the **KDTree**. Rebuild the application and run it again. The time increases to 0.2 – 0.5 milliseconds.

11. Now, we start implementing the special reading callback. It is derived from the osgUtil::IntersectionVisitor::ReadCallback class and the only method to be overridden is readNodeFile(). Regardless of file caching and other optimization solutions, we can directly load the filename variable by calling the osgDB::readNodeFile() method. The input filename variable is automatically invoked by the intersector and is actually the filename of the highest level child of each osg::PagedLOD node:

```
struct PagedReaderCallback : public
  osgUtil::IntersectionVisitor::ReadCallback
{
  virtual osg::Node* readNodeFile( const std::string& filename )
  { return osgDB::readNodeFile(filename); }
};
```

12. Set up the _pagedReader variable in the main entry:

```
picker->_pagedReader = new PagedReaderCallback;
```

13. Restart the application again. Oh, you will see the computation speed is much slower than before (even 200-400 milliseconds). This happens because we spent a lot of time loading new files in the paged LOD's subgraph. This is the price of obtaining the most precise intersection results.

How it works...

The **KDTree** is a binary tree for organizing vertices in a k-dimensional space. It is a special case of the famous Binary Space Partitioning (BSP) tree, but very useful for range searching and some other types of multidimensional data searching.

OSG encapsulates the **KDTree** algorithm totally in the OSG core and uses it to manage vertices in any osg::Geometry objects. By default, the intersection visitor can make use of the scene graph's bounding volume hierarchy to quickly find the leaf nodes that may have intersections with the intersector. But on the geometry level, it must traverse all the primitives (for example all the triangles) to find out intersections, no matter whether only a small portion of the node is intersected. Most calculations are useless here and will cost precious time. In this case, **KDTree** is extremely important, as it creates another new spatial index structure inside the geometry and thus makes the traversing of primitives much faster.

The following line will enable building **KDTree** structure for all geometries loaded:

```
osgDB::Registry::instance()->setBuildKdTreesHint(
    osgDB::Options::BUILD_KDTREES );
```

But we can also enable **KDTree** on a few specified nodes before they are ready to be loaded, for instance:

```
osg::ref_ptr<osgDB::Options> options = new osgDB::Options;
options->setBuildKdTreesHint( osgDB::Options::BUILD_KDTREES );
osg::Node* model = osgDB::readNodeFile( "cow.osg", options.get() );
```

9

Integrating with GUI

In this chapter, we will cover:

- Integrating OSG with Qt
- Starting rendering loops in separate threads
- Embedding Qt widgets into the scene
- Embedding CEGUI elements into the scene
- Using the osgWidget library
- Using OSG components in GLUT
- Running OSG examples on Android
- Embedding OSG into web browsers
- Designing the command buffer mechanism

Introduction

This chapter is full of GUIs. As you may already know, OSG can be integrated with windowing systems by specifying the window handle to the `Traits` class before creating the graphics context. But for different GUI toolkits, the situations may be different too.

OSG itself provides a list of examples that demonstrates the solutions of integrating OSG with many kinds of GUIs:

Example name	Description
osgAndroidExampleGLES1 and osgAndroidExampleGLES2	OSG and Android (with **GLES** v1 or v2)
osgQtBrowser and osgQtWidgets	Embed Qt web browsers/widgets to OSG scene
osgviewerCocoa	OSG and **Cocoa**
osgviewerFLTK	OSG and **FLTK** (`http://www.fltk.org/`)
osgviewerFOX	OSG and **FOX** (`http://www.fox-toolkit.org/`)
osgviewerGLUT	OSG and **GLUT** (use **GLUT** as windowing system, but not renderer)
osgviewerGTK	OSG and **GTK** (`http://www.gtk.org/`)
osgviewerIPhone	OSG and iOS (iPhone, iPad, etc.)
osgviewerMFC	OSG and **MFC** (for Windows only)
osgviewerQt	OSG and **Qt**
osgviewerSDL	OSG and **SDL** (`http://www.libsdl.org/`)
osgviewerWX	OSG and **wxWidgets** (`http://www.wxwidgets.org/`)

Some of them may be too difficult for you to understand at once. We will also introduce how to design GUI elements in 3D world, including using the well known third-party library **CEGUI**, and using the native **osgWidget** library. The use of threads in GUI integration is also introduced, as well as an initial example of using the command buffer concept (which is useful for handling OSG events in an external thread).

Integrating OSG with Qt

Qt is one of the most successful cross-platform application and UI frameworks, which can be used under Linux, Windows, Mac OS X, and even mobile systems. So it is more convenient and portable than integrating OSG with X11/Windows window handles. Qt provides a `QGLWidget` class that can render OpenGL elements directly on the window surface, and in older versions of the OSG source code, you may find some examples using this class. But for 3.0 and newer versions, OSG 3.0 has an osgQt library for implementing different functionalities related to Qt. We will first make use of the `osgQt::GraphicsWindowQt` class in this recipe.

Getting ready

You have to download and compile Qt 4.x before working on the following few examples. Under Ubuntu, you can quickly get the full precompiled packages (including the UI designer tool) using the `apt-get` command:

```
# apt-get install qt4-dev-tools qt4-doc qt4-qtconfig qt4-demos
qt4-designer
```

For other Linux and Windows developers, visit the Qt website and use online installers to get the SDK:

`http://qt.nokia.com/downloads`

You can also obtain the source code and compile Qt by yourselves:

`http://download.qt.nokia.com/qt/source/`

If you are using CMake to find dependencies and generate project files, you can ask CMake to look for Qt installations, for example:

```
FIND_PACKAGE(Qt)
INCLUDE_DIRECTORIES(${QT_INCLUDE_DIR})
…
TARGET_LINK_LIBRARIES(your_app ${QT_QTCORE_LIBRARY}
  ${QT_QTGUI_LIBRARY})
```

You must specify CMake's `QT_QMAKE_EXECUTABLE` variable manually to the location of `qmake` executable file if CMake can't find it automatically.

How to do it...

Let us start.

1. Include necessary headers. Note that you may include specific Qt class headers instead of including `<QtCore>` and `<QtGui>`.

    ```
    #include <QtCore/QtCore>
    #include <QtGui/QtGui>>
    #include <osgDB/ReadFile>
    #include <osgGA/TrackballManipulator>
    #include <osgViewer/ViewerEventHandlers>
    #include <osgViewer/Viewer>
    #include <osgQt/GraphicsWindowQt>
    ```

2. The `createCamera()` function will use the `Traits` class to define basic window parameters, and then apply it to a new `osgQt::GraphicsWindowQt` instance. This `osgQt::GraphicsWindowQt` object in fact includes a Qt widget associated with OSG's graphics context for the rendering operations. Once you have set the scene's camera, the widget will automatically render the scene.

```
osg::Camera* createCamera( int x, int y, int w, int h )
{
  osg::DisplaySettings* ds =
    osg::DisplaySettings::instance().get();
  osg::ref_ptr<osg::GraphicsContext::Traits> traits =
    new osg::GraphicsContext::Traits;
  traits->windowDecoration = false;
  traits->x = x;
  traits->y = y;
  traits->width = w;
  traits->height = h;
  traits->doubleBuffer = true;

  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  camera->setGraphicsContext(
    new osgQt::GraphicsWindowQt(traits.get()) );
  camera->setClearColor( osg::Vec4(0.2, 0.2, 0.6, 1.0) );
  camera->setViewport( new osg::Viewport(
    0, 0, traits->width, traits->height) );
  camera->setProjectionMatrixAsPerspective(
    30.0f, static_cast<double>(traits->width)/
      static_cast<double>(traits->height), 1.0f, 10000.0f );
  return camera.release();
}
```

3. We define a `ViewerWidget` class which is derived from `QWidget` as the container of the rendering widget. It also has a timer that can trigger an updating event in order to execute OSG's `frame()` method frequently.

```
class ViewerWidget : public QWidget
{
public:
  ViewerWidget( osg::Camera* camera, osg::Node* scene );

protected:
  virtual void paintEvent( QPaintEvent* event )
  { _viewer.frame(); }

  osgViewer::Viewer _viewer;
  QTimer _timer;
};
```

4. In the constructor of `ViewerWidget` class, we can set up the OSG viewer as usual, including setting the main camera, scene data, and camera manipulator.

```
_viewer.setCamera( camera );
_viewer.setSceneData( scene );
_viewer.addEventHandler( new osgViewer::StatsHandler );
_viewer.setCameraManipulator(
  new osgGA::TrackballManipulator );

// Use single thread here to avoid known issues under Linux
_viewer.setThreadingModel( osgViewer::Viewer::SingleThreaded );
```

5. We will have to obtain the `osgQt::GraphicsWindowQt` object again and add its internal widget to the parent Qt window with the `getGLWidget()` method.

```
osgQt::GraphicsWindowQt* gw =
  dynamic_cast<osgQt::GraphicsWindowQt*>(
    camera->getGraphicsContext() );
if ( gw )
{
  QVBoxLayout* layout = new QVBoxLayout;
  layout->addWidget( gw->getGLWidget() );
  setLayout( layout );
}
```

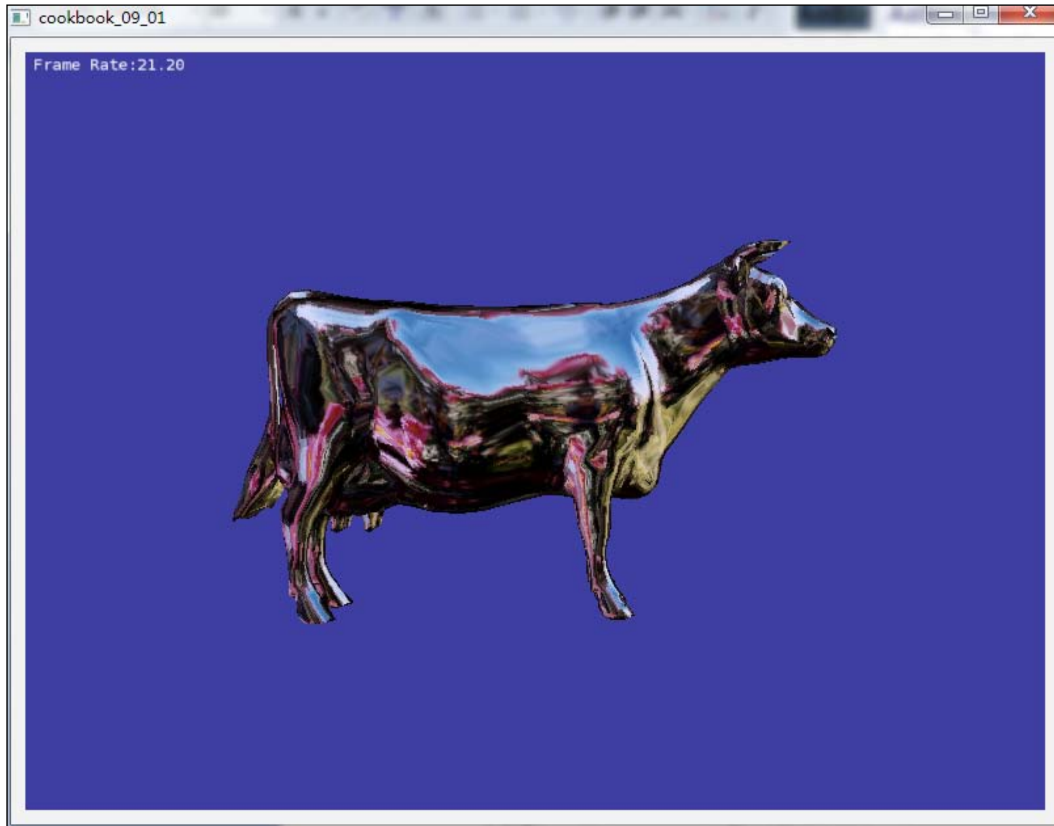
6. The last step in the constructor is to start the `QTimer` and make it start with a constant timeout interval. When the timer times out, a `timeout()` signal will be emitted and the `update()` method will be called, which repaints the window and, thus, executes `_viewer.frame()` in the overridden `paintEvent()`.

```
connect( &_timer, SIGNAL(timeout()), this, SLOT(update()) );
_timer.start( 40 );
```

7. In the main body of the code, we will create the camera and widget object one after another, and start the Qt event loop with the `exec()` method. The OSG simulation loop is already done in the `paintEvent()` method.

```
QApplication app( argc, argv );
osg::Camera* camera = createCamera( 50, 50, 640, 480 );
osg::Node* scene = osgDB::readNodeFile("cow.osg");

ViewerWidget* widget = new ViewerWidget(camera, scene);
widget->setGeometry( 100, 100, 800, 600 );
widget->show();
return app.exec();
```

8. Now you will see the scene with a cow model embedded into a Qt window. Try adding some other common controls like buttons and choice boxes and put them in the widget layout. It is wonderful to have an application with UI elements and a 3D scene.



How it works...

Press *S* and you can see the frame rate is near to 25 fps. That is simply because the `paintEvent()` method will be called when the timer times out every 40 ms, and, thus, causes the `frame()` method of the viewer to be called 25 times per second.

We can also find in the `ViewerWidget` constructor that the viewer uses single threading model here:

```
_viewer.setThreadingModel( osgViewer::Viewer::SingleThreaded );
```

It means the updating, culling, and rendering operations are running in the same thread/process, without benefiting from OSG's multithreading optimization strategy, and, thus, being limited by the timer's interval. Under Windows, we can remove this line directly to improve the rendering performance. But under some Linux distributions, there may be problems when enabling multithreaded rendering with the Qt widget, and the application may crash. Please run the viewer single-threaded in this situation, or report your issues to the `osg-users` if you think it necessary.

Starting rendering loops in separate threads

We may not be delighted with the rendering speed of 25 fps; it means that OSG can use at most 1/25 of one second for the rendering work. And when there are more objects to render or more tasks to finish in the callbacks, the frame rate will continue to drop down and makes the whole process inefficient.

One solution is to use an independent thread for OSG traversals and drawing commands. Of course, the OpenThreads library, which is included in the core OSG source code, is the best choice for implementing such a job. But as we are working with Qt too, we will try the the Qt threading class here. The code will be based on the *Integrating OSG with Qt* recipe and only have a few modifications.

How to do it...

Let us start.

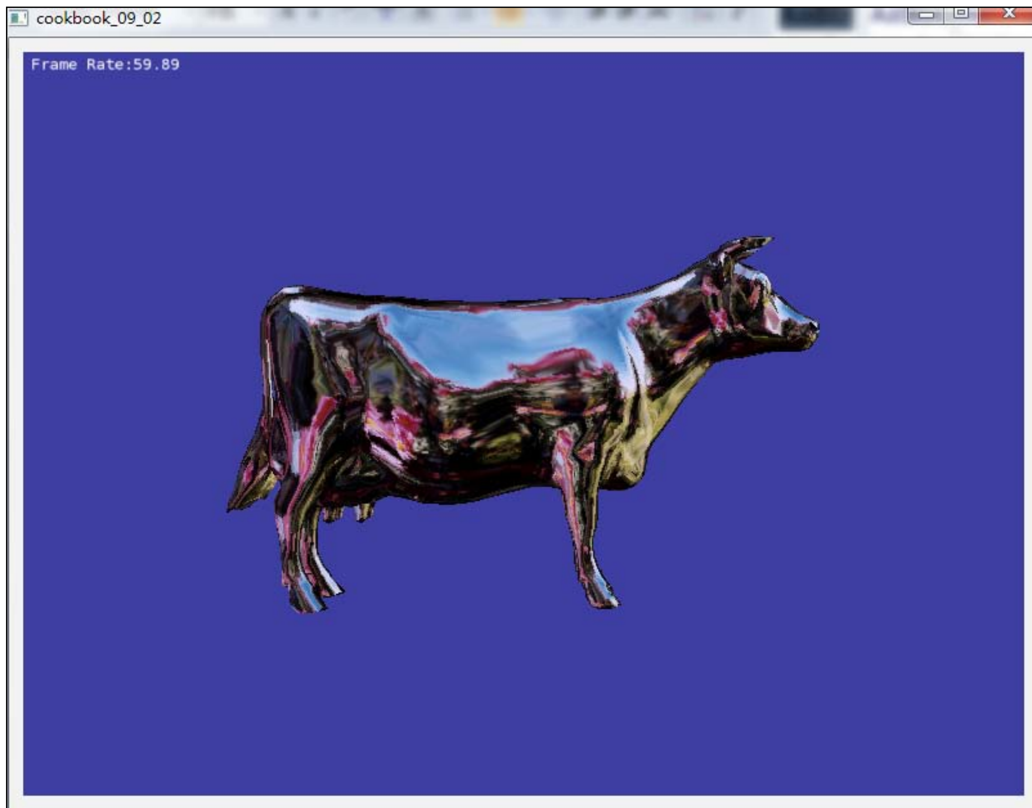
1. The `createCamera()` function doesn't need to be modified in this recipe. We will derive from the `QThread` class to design a multithreaded rendering solution. It in fact overrides only the `run()` method to perform the viewer's simulation loop. And when the instance is destroyed, we must immediately set the exiting flag of the viewer and wait for the loop to be finished. This is a tiny but standard style for writing multithreaded programs.

```
class RenderThread : public QThread
{
public:
  RenderThread() : QThread(), viewerPtr(0) {}

  virtual ~RenderThread()
  { if (viewerPtr) viewerPtr->setDone(true); wait(); }

  osgViewer::Viewer* viewerPtr;

protected:
  virtual void run()
  { if (viewerPtr) viewerPtr->run(); }
};
```

2. In the constructor of `ViewerWidget` class, we will no longer trigger updating events with the timer, but start the thread instead.

```
// 'RenderThread _thread' is the member of ViewerWidget
_thread.viewerPtr = &_viewer;
_thread.start();
```

3. No more changes in the main entry.
4. The result is exactly the same as the Qt integration example before. But when you press S to view the scene and the rendering statistics. You will find that the frame rate will be stable at 60 fps this time (in the last one, it was about 25 fps). This is all because rendering work in a separate thread will run concurrently along with the main process instead of waiting for being called in a timer event.



How it works...

The `QThread` class is easy to understand as it only re-implements the following method:

```
virtual void run()
{ if (viewerPtr) viewerPtr->run(); }
```

Usually a multithreading developer will use a loop to fill the method body so that it will continuously work when a thread is waked up. The `osgViewer::Viewer::run()` method is suitable for this case. But to add more controls of your own, you can create a loop and call the viewer's `frame()` method inside. The update traversal of the scene graph will be done in the thread loop, but the culling and rendering tasks may create and use additional threads due to the threading mode used currently.

There's more...

This time we come across a classic question—how can OSG, or OpenGL work in multithreaded applications? In this recipe, it is a quite simple situation because all rendering-related work is done in one thread and all UI operations in another. It should be safe because there is only one current OpenGL graphics context inside the newly-created thread.

The OpenGL context is thread-specific. It must be released before another context is made as current, and it must always be used from the same thread. Otherwise, the application may fail to continue or even crash.

The preferred way to make OpenGL-based applications benefit from multiple threads is to decouple the user thread from the drawing one. That is, to process the user logic and updating work, as well as scene culling operations in one or more threads, which can never be modified during rendering. The data to be updated or culled must be delayed if it is to be used by the rendering thread at the same time. This is also known as the famous **app-cull-draw** structure of modern rendering frameworks.

Fortunately, all this is handled nicely in the OSG system and we don't have to worry about these 'multi-programming' problems, including multi-threaded, multi-core, multi-context, and multi-display applications.

Embedding Qt widgets into the scene

Besides rendering OpenGL elements in a Qt window, it is also possible to embed Qt content (especially different types of Qt widgets) into the 3D world. A complete example named `WolfenQt` can be found at `http://qt.gitorious.org/qt-labs/wolfenqt`.

It shows an amazing Wolfenstein-like (a famous old computer game) demo program with Qt buttons, edit boxes, movie players, and other widgets embedded on the walls of a maze. All the widgets are derived from the `QGraphicsItem` class with a customized transform and `QGraphicsView` used to handle all user events. This example isn't intended to use Qt as an internal GUI tool in 3D applications, but it is a good demonstration for people who want to try some interesting concepts.

Fortunately, OSG has also encapsulated the `QGraphicsItem/QGraphicsView` structure in the osgQt namespace, which will be shown in the following recipe.

How to do it...

Let us start.

1. Include necessary headers.

```
#include <QtCore/QtCore>
#include <QtGui/QtGui>>
#include <osg/Texture2D>
#include <osg/Geometry>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
#include <osgGA/TrackballManipulator>
#include <osgViewer/ViewerEventHandlers>
#include <osgViewer/Viewer>
#include <osgQt/QWidgetImage>
```

2. The `createDemoWidget()` function is used to create a simple but funny window, composed of a label which can play animated GIF files and two buttons for starting and stopping the movie. We use Qt's preset signals and slots to implement the movie controllers.

```
QWidget* createDemoWidget( const QString& movieFile )
{
  ...  // Please see source code for details
}
```

3. In the main entry, we allocate an `osgQt::QWidgetImage` object with the created Qt widget as the parameter. Note that this must be done after the `QApplication` variable is established; otherwise Qt itself will not be initialized properly.

```
QApplication app( argc, argv );
osg::ref_ptr<osgQt::QWidgetImage> widgetImage =
  new osgQt::QWidgetImage( createDemoWidget("animation.gif") );
```

4. There is no reason to ignore user events on Qt widgets (such as pushing the button or typing in the text-box) in the 3D scene. OSG provides an `osgViewer::InteractiveImageHandler` class to handle such events, that is, to pass the mouse and keyboard events to the `osgQt::QWidgetImage` object.

```
osg::ref_ptr<osgViewer::InteractiveImageHandler> handler =
  new osgViewer::InteractiveImageHandler( widgetImage.get() );
```

5. Now we set the image which displays Qt scene to a new texture object.

```
osg::ref_ptr<osg::Texture2D> texture = new osg::Texture2D;
texture->setImage( widgetImage.get() );
```

6. Allocate a new quad and set the specialized event handler to its callbacks (both event and cull callbacks as the image handler requires).

```
osg::ref_ptr<osg::Geometry> quad =
  osg::createTexturedQuadGeometry(osg::Vec3(), osg::X_AXIS,
    osg::Z_AXIS);
quad->setEventCallback( handler.get() );
quad->setCullCallback( handler.get() );
```

7. Add both the texture and the quad drawable to the scene graph.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( quad.get() );
geode->getOrCreateStateSet()->setTextureAttributeAndModes(
  0, texture.get() );
geode->getOrCreateStateSet()->setMode(
  GL_LIGHTING, osg::StateAttribute::OFF);
geode->getOrCreateStateSet()->setRenderingHint(osg::StateSet::TRAN
SPARENT_BIN);

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( geode.get() );
```

8. Now we can start the viewer. You must still remember that Qt has its own event loop to handle in every frame. We will simply call the `processEvents()` method in the simulation loop body to make it work here. This is essentially the same as running `frame()` in `QWidget`'s painting event in the previous recipe. But later, we will introduce another method to integrate OSG and Qt, but execute them in different threads.

```
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.setCameraManipulator( new osgGA::TrackballManipulator );
viewer.addEventHandler( new osgViewer::StatsHandler );

while ( !viewer.done() )
```

```
{
  QCoreApplication::processEvents( QEventLoop::AllEvents,
    100 );
  viewer.frame();
}
return 0;
```

9. Now it is exciting to see Qt GUI elements embedded in our familiar OSG scene. You can rotate, pan, and zoom the camera as usual. When you move the mouse inside the widget quad and press on either of the buttons, you will find that the movie's state (playing or paused) is changed at the same time, which is exactly as intended.

How it works...

Here the `osgQt::QWidgetImage` class is actually derived from `osg::Image`. It means that the entire Qt widget is drawn to a picture first, and then the picture is attached to a texture and rendered in the 3D world. The drawing process is done in the internal `osgQt::QGraphicsViewAdapter` class which connects OSG and Qt's graphics scene/view mechanism. A similar method is used while we are playing movies in the scene graph, that is, each frame of a movie is encoded and painted to an `osg::Image` object and then updated to the texture for displaying.

The `osgViewer::InteractiveImageHandler` is also special as it won't be applied to the viewer, but to the drawable used for containing the image. It replaces the normal event handling and culling processes of the drawable and handles user events and passes them to Qt widgets. The mapping from OSG key/mouse values to Qt ones is also defined in `osgQt::QGraphicsViewAdapter`. You may take a look at this important class in `src/osgQt` subdirectory of the OSG source code.

Embedding CEGUI elements into the scene

Crazy Eddie's GUI (**CEGUI**) is an open source library providing 2D GUI widgets for graphic APIs such as OpenGL and DirectX. It is one of the most successful 3D GUI projects written in C++ and targeted mainly to game developers. You can read more about this library and download the source code or precompiled SDK at `http://www.cegui.org.uk`.

CEGUI can be used to design fantastic UIs and is already proven to be usable in many commercial and non-commercial projects. It will be great to integrate it into the OSG scene graph and benefit from its powerful functionalities. In this recipe, we will try to provide a solution embedding CEGUI as a customized drawable object in the scene graph. First, you may download the CEGUI library and read its examples to learn how to make use of CEGUI widgets and windowing functions. We won't focus on the usage of CEGUI in the following sections of this recipe.

Getting ready

You can download the CEGUI source code and precompiled libraries at `http://www.cegui.org.uk/wiki/index.php/CEGUI_Downloads_0.7.5`.

Although Ubuntu and other Linux distributions can obtain the CEGUI SDK using `apt-get` and other tools, it still doesn't fit our requirements here. We need version 0.7.5 for the following code to compile and run properly, but there are some vital differences between 0.7.5 and older versions. So you should compile CEGUI from the source code before tasting this recipe if there are no binary CEGUI SDKs provided for your operating systems.

You can add the following lines in the `CMakeLists.txt` file to help specify the CEGUI headers and main libraries (`CEGUIBase` and `CEGUIOpenGLRenderer`), and link them to your project:

```
FIND_PACKAGE(OpenGL)
FIND_PATH(CEGUI_INCLUDE_DIR CEGUI.h)
FIND_LIBRARY(CEGUI_GL_LIBRARY CEGUIOpenGLRenderer)
FIND_LIBRARY(CEGUI_LIBRARY CEGUIBase)
…
INCLUDE_DIRECTORIES(${CEGUI_INCLUDE_DIR} ${OPENGL_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(your_app ${CEGUI_GL_LIBRARY} ${CEGUI_LIBRARY}
${OPENGL_gl_LIBRARY})
```

How to do it...

Let us start.

1. Include necessary headers:

```
#include <CEGUI.h>
#include <RendererModules/OpenGL/CEGUIOpenGLRenderer.h>
#include <osg/BlendFunc>
#include <osg/Drawable>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <osgViewer/ViewerEventHandlers>
#include <iostream>
```

2. We will derive `osg::Drawable` class to execute CEGUI updating and drawing commands when the OpenGL context is current. The CEGUI controls (a window with a single button in this recipe) should also be initialized when the `drawImplementation()` method is called for the first time. The mutable member variables `_activeContextID` and `_initialized` are useful here as they record the working context ID and check if the initialization is finished.

```
class CEGUIDrawable : public osg::Drawable
{
public:
  CEGUIDrawable();
  CEGUIDrawable( const CEGUIDrawable& copy,const osg::CopyOp&
    copyop=osg::CopyOp::SHALLOW_COPY );
  META_Object( osg, CEGUIDrawable );

  virtual void drawImplementation( osg::RenderInfo&
    renderInfo ) const;

  void initializeControls();
  bool handleClose( const CEGUI::EventArgs& e );

protected:
  virtual ~CEGUIDrawable() {}

  // They are mutable for being altered in const methods
  mutable double _lastSimulationTime;
  mutable unsigned int _activeContextID;
  mutable bool _initialized;
};
```

3. Set up suitable member values in the constructor and copy constructor. We should also turn off lighting and depth test here to make sure the CEGUI widgets are always on top of other scene objects.

```
CEGUIDrawable::CEGUIDrawable()
: _lastSimulationTime(0.0), _activeContextID(0),
  _initialized(false)
{
  setSupportsDisplayList( false );
  setDataVariance( osg::Object::DYNAMIC );
  getOrCreateStateSet()->setMode( GL_LIGHTING,
    osg::StateAttribute::OFF );
  getOrCreateStateSet()->setMode( GL_DEPTH_TEST,
    osg::StateAttribute::OFF );
}

CEGUIDrawable::CEGUIDrawable( const CEGUIDrawable&
  copy,const osg::CopyOp& copyop )
: osg::Drawable(copy, copyop),
  _lastSimulationTime(copy._lastSimulationTime),
  _activeContextID(copy._activeContextID),
  _initialized(copy._initialized)
{}
```

4. Let us start to implement the `drawImplementation()` method now, which is the key for integrating CEGUI with OSG.

```
void CEGUIDrawable::drawImplementation( osg::RenderInfo&
renderInfo ) const
{
  ...
}
```

5. Get the current context ID for later use and check if the initialization process is ever called.

```
unsigned int contextID = renderInfo.getContextID();
if ( !_initialized )
{
  ...
}
else
{
  ...
}
```

6. If this is the first time we have entered this method, we have to initialize the CEGUI system as shown in the following block of code. You may read more about the startup of a CEGUI-based application on the CEGUI website. Here we will only create a new OpenGL renderer and set up the search paths and names for different resource types.

    ```
    CEGUI::OpenGLRenderer::bootstrapSystem(
      CEGUI::OpenGLRenderer::TTT_NONE );
    if ( !CEGUI::System::getSingletonPtr() ) return;
    ...  // Please see source code for details
    Run initializeControls() method to create some CEGUI widgets and
    bind callbacks with widgets' specific events.
    const_cast<CEGUIDrawable*>(this)->initializeControls();
    _activeContextID = contextID;
    _initialized = true;
    ```

7. If we have already initialized CEGUI and ensured the context ID is the same as the one we used to create CEGUI elements (if we work with multiple windows, we have to handle multiple IDs too), we can start the rendering work. The first step is to disable any previous vertex arrays and back up current OpenGL states.

    ```
    osg::State* state = renderInfo.getState();
    state->disableAllVertexArrays();
    state->disableTexCoordPointer( 0 );

    glPushMatrix();
    glPushAttrib( GL_ALL_ATTRIB_BITS );
    ```

8. Now we have to check if the viewport is changed or not and call the CEGUI system's resizing method. We will explain why this is not done in event handlers in the next section.

    ```
    CEGUI::OpenGLRenderer* renderer =
      static_cast<CEGUI::OpenGLRenderer*>(
        CEGUI::System::getSingleton().getRenderer() );
    osg::Viewport* viewport =
      renderInfo.getCurrentCamera()->getViewport();
    if ( renderer && viewport )
    {
      const CEGUI::Size& size = renderer->getDisplaySize();
      if ( size.d_width!=viewport->width() ||
        size.d_height!=viewport->height() )
      {
        CEGUI::System::getSingleton().notifyDisplaySizeChanged(
          CEGUI::Size(viewport->width(), viewport->height()) );
      }
    }
    ```

9. Compute delta time from last frame to current frame, use it to update elements requiring timing, and then render the GUI widgets. Don't forget to restore OpenGL attributes at the end of this method.

```
double currentTime = (state->getFrameStamp() ?
  state->getFrameStamp()->getSimulationTime() : 0.0);
CEGUI::System::getSingleton().injectTimePulse( (
  currentTime - _lastSimulationTime)/1000.0 );
CEGUI::System::getSingleton().renderGUI();
_lastSimulationTime = currentTime;

glPopAttrib();
glPopMatrix();
```

10. In the `initializeControls()` method, you can implement any kind of CEGUI-based interface if you have the knowledge of this powerful library. To make this recipe simple, we will only create a demo window with a single button and add it to the root window using the default `TaharezLook` style. The window's close button at the right-top corner is connected with the callback method `handleClose()` to ensure that we can quit this window when desired.

```
void CEGUIDrawable::initializeControls()
{
  CEGUI::SchemeManager::getSingleton().create(
    "TaharezLook.scheme" );
  CEGUI::System::getSingleton().setDefaultMouseCursor(
    "TaharezLook", "MouseArrow" );

  ...  // Please see source code for details

  demoWindow->subscribeEvent(
    CEGUI::FrameWindow::EventCloseClicked,
      CEGUI::Event::Subscriber(&CEGUIDrawable::handleClose,
        this) );
  demoWindow->addChildWindow( demoButtonOK );
  root->addChildWindow( demoWindow );
}
```

11. In the `handleClose()` method, we just hide the window as it is closed by the user.

```
bool CEGUIDrawable::handleClose( const CEGUI::EventArgs& e )
{
  CEGUI::WindowManager::getSingleton().getWindow(
    "DemoWindow")->setVisible( false );
  return true;
}
```

12. Now we have to design an event handler to convert OSG's mouse events to CEGUI-styled ones. This is important as CEGUI can never receive any user inputs without such a conversion. Note that we don't implement keyboard conversion here. If you wish to finish it by yourselves, just remember to use a map to record OSG keys and corresponding CEGUI keys. For example, OSG's `KEY_Return` value must be converted to `CEGUI::Key::Return` and sent to the CEGUI system if it is detected.

```
class CEGUIEventHandler : public osgGA::GUIEventHandler
{
public:
  CEGUIEventHandler( osg::Camera* camera ) : _camera(camera) {}

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

protected:
  CEGUI::MouseButton convertMouseButton( int button );
  osg::observer_ptr<osg::Camera> _camera;
};
```

13. In the `handle()` method, don't forget to recalculate the correct Y value for CEGUI, which increases downwards.

```
int x = ea.getX(), y = ea.getY(), width = ea.getWindowWidth(),
  height = ea.getWindowHeight();
if ( ea.getMouseYOrientation()==
  osgGA::GUIEventAdapter::Y_INCREASING_UPWARDS )
y = ea.getWindowHeight() - y;
if ( !CEGUI::System::getSingletonPtr() )
return false;
```

14. Execute proper CEGUI methods to deliver mouse events here. The only case to note here is `RESIZE`. We must reset the projection matrix and viewport of the HUD camera used by CEGUI (see the code segments in the next few steps for its creation) when the window is resized. CEGUI will be notified about this in the next drawing process in `CEGUIDrawable::drawImplementation()`, as described earlier.

```
switch ( ea.getEventType() )
{
..  // Please see source code for details
  case osgGA::GUIEventAdapter::RESIZE:
  if ( _camera.valid() )
  {
    _camera->setProjectionMatrix( osg::Matrixd::ortho2D(
      0.0, width, 0.0, height) );
    _camera->setViewport( 0.0, 0.0, width, height );
  }
```

```
  break;
  default:
  return false;
}
```

15. The last step in the `handle()` method is to check if any CEGUI window is intersected with the mouse cursor. We will try to return `true` to notify other handlers not to pick up the same event again, because a CEGUI window has already handled the event.

```
CEGUI::Window* rootWindow =
  CEGUI::System::getSingleton().getGUISheet();
if ( rootWindow )
{
  CEGUI::Window* anyWindow = rootWindow->
    getChildAtPosition( CEGUI::Vector2(x, y) );
  if ( anyWindow ) return true;
}
return false;
```

16. The `convertMouseButton()` method is just used for converting OSG mouse event values to CEGUI's.

```
switch ( button )
{
  case osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON:
    return CEGUI::LeftButton;
  case osgGA::GUIEventAdapter::MIDDLE_MOUSE_BUTTON:
    return CEGUI::MiddleButton;
  case osgGA::GUIEventAdapter::RIGHT_MOUSE_BUTTON:
    return CEGUI::RightButton;
  default: break;
}
return static_cast<CEGUI::MouseButton>(button);
```

17. In the main entry, we will add a new `CEGUIDrawable` instance to an `osg::Geode` node. Don't forget to call `setCullingActive(false)` here; otherwise, the drawable will be culled because we didn't implement the `computeBound()` method to define its bound (in fact we don't have such a bound). The node can be transparent to prepare for any translucent widgets.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->setCullingActive( false );
geode->addDrawable( new CEGUIDrawable );
geode->getOrCreateStateSet()->setAttributeAndModes(
  new osg::BlendFunc );
geode->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
```

18. Now we must put the CEGUI widgets to a HUD display and render them on the top of all other models. That means a HUD camera should be placed in the scene graph. We should make sure it can receive user events by calling `setAllowEventFocus(true)` function.

    ```
    osg::ref_ptr<osg::Camera> hudCamera =
      osgCookBook::createHUDCamera(0, 800, 0, 600);
    hudCamera->setAllowEventFocus( true );
    hudCamera->addChild( geode.get() );
    ```

19. OK, now create the root node and start the viewer.

    ```
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->addChild( osgDB::readNodeFile("cow.osg") );
    root->addChild( hudCamera.get() );

    osgViewer::Viewer viewer;
    viewer.setSceneData( root.get() );
    viewer.addEventHandler(
      new CEGUIEventHandler(hudCamera.get()) );
    viewer.addEventHandler(
      new osgViewer::WindowSizeHandler );
    viewer.addEventHandler(
      new osgViewer::StatsHandler );
    return viewer.run();
    ```

20. The last thing to do is to copy the `datafiles` folder in the CEGUI SDK to current working directory. The location of `datafiles` is already specified to CEGUI resource groups in the initialization process of the `drawImplementation()` method. Without the `datfilesdatafiles` folder, CEGUI will miss all the widgets, fonts, and style information, and simply fail to start.

21. Now we can see a game-like dialog with only one button appearing in front of the cow model. Click on the **X** at the top-right and you can close this dialog at any time. If you have time to read the CEGUI documentation and design a complex enough interface, you can easily integrate it with OSG by rewriting `CEGUIDrawable::initializeControls()` method and create beautiful GUI elements from now on.

How it works...

Now we will see why CEGUI's resizing is done in the rendering process instead of in the event handler. CEGUI internally uses plenty of textures to draw different UI elements. When the window size is modified, it rescales all these elements to fit the new size. This requires a recompilation of OpenGL textures and thus needs to be done when the OpenGL context is current. That is why we call `notifyDisplaySizeChanged()` method of the CEGUI system singleton in the drawable's method. The HUD camera's viewport and matrix must be altered as well to ensure that the display is not distorted.

This example successfully detaches the GUI part from other scene operations. CEGUI elements and callbacks are centralized in the customized `CEGUIDrawable` and the `CEGUIEventHandler`. It won't affect the scene graph even if you open or close dialogs, press buttons, or input texts in the text boxes.

The only problem is the threading model. As you can see from the source code, CEGUI uses a singleton system pointer everywhere to handle events and control UI objects. The pointer may be called in more than one thread if we have to take part in the CEGUI system in node callbacks or other places. And due to CEGUI's design, it is already hard to provide any protection solutions here to avoid visiting at the same time. The only suggestion is that you must run these kinds of OSG applications in single-threaded model, or use a command buffer to record operations to the CEGUI system and execute them later. We will discuss the latter in the last recipe of this chapter.

Using the osgWidget library

The osgWidget library is a **NodeKit** that helps OSG support 2D GUI windows and elements in the 3D world. As you have seen from the last few examples, it is not hard to embed Qt and CEGUI controls and layouts into the scene graph, but more external dependencies must be added and maintained along with the program. In addition, the scene graph concept is not used for constructing and updating the graphics interface, and it is really hard to visit, modify, and compute the intersections with these UI elements. To solve this we require the osgWidget, which depends heavily on the OSG node and image derivatives, and can integrate well with other scene objects.

osgWidget accepts three kinds of basic widget (they can be nested, that is, have sub-widgets inside)—box, canvas, and table. A box must line up child widgets horizontally or vertically. A canvas can draw sub-widgets at any coordinates without any constraints. A table, as its name suggests, lays out widgets in the correct cell (specified row and column) in a grid.

The UI example in this recipe will be very simple. We will make a tab widget including three text tabs and sub-windows. It won't have a Windows- or Mac OS X-like style, but can demonstrate how osgWidget works with scene graph and the UI event system simultaneously.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <osgWidget/WindowManager>
#include <osgWidget/Box>
#include <osgWidget/Canvas>
#include <osgWidget/Label>
#include <osgWidget/ViewerEventHandlers>
#include <iostream>
#include <sstream>
```

2. Let us declare the `tabPressed()` callback function which will be executed automatically when one of the tabs is selected.

```
extern bool tabPressed( osgWidget::Event& ev );
```

3. We will have a `createLabel()` function for creating labels quickly. You can find that its methods are very similar to `osgText::Text`. In fact, the osgText library is also used by osgWidget for text rendering.

```
osgWidget::Label* createLabel( const std::string& name,
  const std::string& text, float size,
    const osg::Vec4& color )
{
  osg::ref_ptr<osgWidget::Label> label =
    new osgWidget::Label(name);
  label->setLabel( text );
  label->setFont( "fonts/arial.ttf" );
  label->setFontSize( size );
  label->setFontColor( 1.0f, 1.0f, 1.0f, 1.0f );
  label->setColor( color );
  label->addSize( 10.0f, 10.0f );
  label->setCanFill( true );
  return label.release();
}
```

4. Now we are going to create the entire tab widget.

```
osgWidget::Window* createSimpleTabs( float winX, float winY )
{
  ...
}
```

5. First, we will create a canvas which allows sub-widgets to be placed at any coordinates, and a box window for positioning children uniformly. The canvas is used as the tab contents, and the box will include three tab labels horizontally.

```
osg::ref_ptr<osgWidget::Canvas> contents =
  new osgWidget::Canvas("contents");
osg::ref_ptr<osgWidget::Box> tabs =
  new osgWidget::Box("tabs", osgWidget::Box::HORIZONTAL);
```

6. We create the three tab labels and link them with the same callback function in succession; and to make the recipe easy enough, we will only add three multi-text labels to the contents. The `setLayer()` method decides which label will be placed at the top and, thus, visible to viewers. It actually sets the Z-order of UI window elements.

```
for ( unsigned int i=0; i<3; ++i )
{
  osg::Vec4 color(0.0f, (float)i / 3.0f, 0.0f, 1.0f);
  std::stringstream ss, ss2;
  ss << "Tab-" << i;
  ss2 << "Tab content:" << std::endl <<
```

```
        "Some text for Tab-" << i;

      osgWidget::Label* content = createLabel(ss.str(),
        ss2.str(), 10.0f, color);
      content->setLayer( osgWidget::Widget::LAYER_MIDDLE, i );
      contents->addWidget( content, 0.0f, 0.0f );

      osgWidget::Label* tab = createLabel(ss.str(),
        ss.str(), 20.0f, color);
      tab->setEventMask( osgWidget::EVENT_MOUSE_PUSH );
      tab->addCallback( new osgWidget::Callback(
        &tabPressed, osgWidget::EVENT_MOUSE_PUSH, content) );
      tabs->addWidget( tab );
    }
```

7. After the creation of three tabs and their contents, we use a vertical box to contain them and set its title using another new label object, and return it.

```
osg::ref_ptr<osgWidget::Box> main =
  new osgWidget::Box("main", osgWidget::Box::VERTICAL);
main->setOrigin( winX, winY );
main->attachMoveCallback();
main->addWidget( contents->embed() );
main->addWidget( tabs->embed() );
main->addWidget( createLabel("title", "Tabs Demo",
  15.0f, osg::Vec4(0.0f, 0.4f, 1.0f, 1.0f)) );
return main.release();
```

8. The `tabPressed()` function can handle mouse-clicking events on the tab and change the content shown in the canvas. To achieve this, we must obtain the canvas window first and reset all children's layer numbers. The layer to be displayed will always have bigger values. After the modification, the canvas must be resized to update all its elements. Returning `true` means the callback is captured and handled in a proper way.

```
bool tabPressed( osgWidget::Event& ev )
{
  osgWidget::Label* content = static_cast<
    osgWidget::Label*>( ev.getData() );
  if ( !content ) return false;

  osgWidget::Canvas* canvas = dynamic_cast<
    osgWidget::Canvas*>( content->getParent() );
  if ( canvas )
  {
    osgWidget::Canvas::Vector& objs = canvas->getObjects();
```

```
    for( unsigned int i=0; i<objs.size(); ++i )
    objs[i]->setLayer( osgWidget::Widget::LAYER_MIDDLE, i );

    content->setLayer( osgWidget::Widget::LAYER_TOP, 0 );
    canvas->resize();
  }
  return true;
}
```

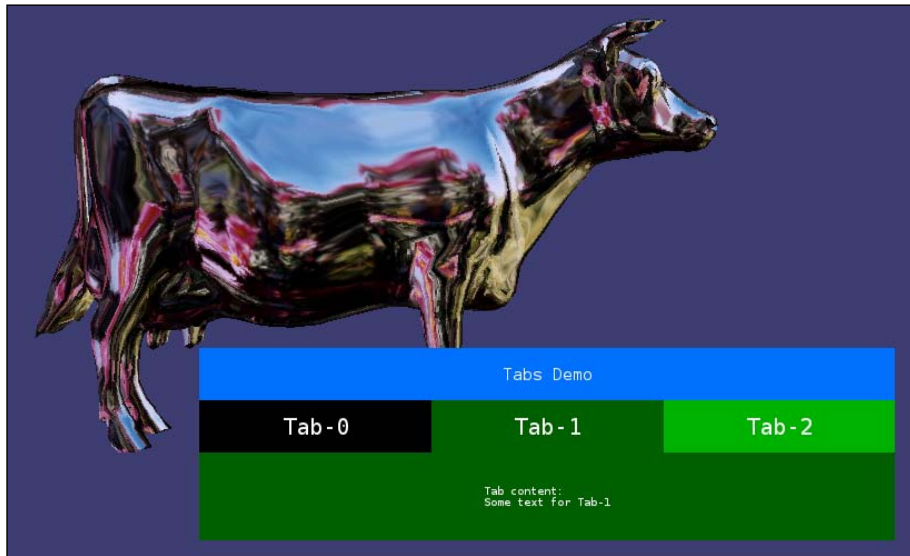
9. In the main entry, we create a window manager with a fixed size and a mask for the intersectors to ignore 2D elements. An **ortho** camera can be retrieved from the window manager for adding to the root node, and the simple tab window should be added to the manager before calling `resizeAllWindows()` method.

```
osgViewer::Viewer viewer;
osg::ref_ptr<osgWidget::WindowManager> wm =
  new osgWidget::WindowManager(&viewer, 1024.0f,
    768.0f, 0xf0000000);
osg::Camera* camera = wm->createParentOrthoCamera();

wm->addChild( createSimpleTabs(100.0f, 100.0f) );
wm->resizeAllWindows();
```

10. Add the UI camera and all other scene nodes to the root and configure the viewer. A few osgWidget-specific handlers are also used here, which will be introduced in the next section of this recipe.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( osgDB::readNodeFile("cow.osg") );
root->addChild( camera );

viewer.setSceneData( root.get() );
viewer.setUpViewInWindow( 50, 50, 1024, 768 );
viewer.addEventHandler(
  new osgWidget::MouseHandler(wm.get()) );
viewer.addEventHandler(
  new osgWidget::KeyboardHandler(wm.get()) );
viewer.addEventHandler( new osgWidget::ResizeHandler(wm.get(),
  camera) );
viewer.addEventHandler(
  new osgWidget::CameraSwitchHandler(wm.get(), camera) );
return viewer.run();
```

11. Run the application and you can see a rough UI window shown in the scene graph, along with the cow model:



12. As osgWidget is still young and weak, it doesn't have any built-in styles and is not as beautiful and professional as the X11/Windows/Mac OS X interfaces we are familiar with. But it is still good to know that OSG has its own UI system which is under construction. If, after reading this book, you choose osgWidget, why not contribute some of your own improvements?

How it works...

The base structure of osgWidget elements is easy to understand—a window manager must be created and maintained separately as it defines the size and basic attributes of the entire UI system. Numbers of window objects, such as `osgWidget::Box` and `osgWidget::Canvas`, can be added to the manager for displaying and receiving user events. They are also added to the ortho camera internally. A window object always has widgets inside, such as labels, buttons, input boxes, and other customized elements. All these objects are in fact group nodes with special functionalities, so the UI can be visited by `osg::NodeVisitor` class and can have updates and cull callbacks besides the event-related ones.

There are four handlers defined in the osgWidget library that are necessary for work:

- `osgWidget::MouseHandler`: It handles mouse events and transfers events to corresponding callbacks defined by `osgWidget::Callback`.
- `osgWidget::KeyboardHandler`: It handles keyboard events and sends them to callbacks.

- `osgWidget::ResizeHandler`: It handles resizing of the window manager and all its children.
- `osgWidget::CameraSwitchHandler`: With this handler, you can press *F12* to change to 3D view of the UI elements, which is convenient and sometimes interesting.

You can also use the global `osgWidget::createExample()` function to manage the camera and all these events internally. The code segment is as follows:

```
osgWidget::createExample( viewer, wm.get() );
```

Using OSG components in GLUT

Now we will try to discuss another interesting topic, that is, how to integrate OSG into other 3D engines based on OpenGL.

This would seem meaningless to developers who use OSG from start to finish in their projects. But for people who are familiar with some other SDKs, such as Ogre3D, Open Inventor, or others, they might have already worked under those SDKs for a long time and had a lot of products and modules in hand, and wouldn't want to change to OSG and rewrite all the existing code. In that case, they can just embed some OSG components into their own applications as modules, and render OSG elements using standard OpenGL contexts (WGL/GLX).

In this recipe, we choose the **GLUT** library (`http://www.opengl.org/resources/libraries/glut/`) as the base system and do the rendering and interacting work in GLUT callbacks. Then we will directly call OSG's rendering back end to update, cull, and render the scene. The scene graph structure and visitor mechanism may not be affected because they have no relations with the renderer.

Getting ready

Under Ubuntu, you can download the **FreeGLUT** library, which is completely compatible with the original GLUT library:

```
# apt-get install freeglut-dev
```

Windows users can obtain the original GLUT headers and libraries pre-built at `http://www.opengl.org/resources/libraries/glut/glut37.zip`.

Then configure this library to be found in CMake scripts:

```
FIND_PACKAGE(GLUT)
...
INCLUDE_DIRECTORIES(${GLUT_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(${GLUT_LIBRARIES})
```

How to do it...

Let us start.

1. Include necessary headers. Note that we don't need to include osgViewer headers here, because GLUT will take the place of it.

```
#include <osgDB/ReadFile>
#include <osgUtil/SceneView>
#include <GL/glut.h>
```

2. Define some global variables for use. The `osgUtil::SceneView` class is often used internally as the rendering back end, but this time it takes the leading role.

```
osg::ref_ptr<osgUtil::SceneView> g_sceneView =
  new osgUtil::SceneView;
osg::Matrix g_viewMatrix, g_projMatrix;
float g_width = 800.0f, g_height = 600.0f;
unsigned int g_frameNumber = 0;
osg::Timer_t g_startTick;
```

3. Create an `initializeFunc()` function to initialize some OSG-related data, including the view and projection matrices, the start time of the timer, and passing the input root node to the `SceneView` object.

```
void initializeFunc( osg::Node* model )
{
  if ( model )
  {
    const osg::BoundingSphere& bs = model->getBound();
    g_viewMatrix.makeLookAt(
      bs.center() - osg::Vec3(0.0f,4.0f*bs.radius(),0.0f),
        bs.center(), osg::Z_AXIS );
  }
  g_projMatrix.makePerspective( 30.0f, g_width/g_height,
    1.0f, 10000.0f );
  g_startTick = osg::Timer::instance()->tick();

  g_sceneView->setDefaults();
  g_sceneView->setSceneData( model );
  g_sceneView->setClearColor( osg::Vec4(0.2f,0.2f,0.6f,1.0f) );
}
```

4. GLUT uses callbacks to handle drawing, resizing, and user events of a render window. The `displayFunc()` function will be called every time when GLUT is going to draw a new frame in the window. We will set the correct reference time, update, and draw the `SceneView` object before the buffer-switching operation.

```
void displayFunc()
{
  osg::Timer_t currTick = osg::Timer::instance()->tick();
  double refTime = osg::Timer::instance()->delta_s(
    g_startTick, currTick);

  osg::ref_ptr<osg::FrameStamp> frameStamp =
    new osg::FrameStamp;
  frameStamp->setReferenceTime( refTime );
  frameStamp->setFrameNumber( g_frameNumber++ );

  g_sceneView->setFrameStamp( frameStamp.get() );
  g_sceneView->setViewport( 0.0f, 0.0f, g_width, g_height );
  g_sceneView->setViewMatrix( g_viewMatrix );
  g_sceneView->setProjectionMatrix( g_projMatrix );

  g_sceneView->update();
  g_sceneView->cull();
  g_sceneView->draw();
  glutSwapBuffers();
}
```

5. When the window is resized, alter the projection matrix used by the `SceneView` object.

```
void reshapeFunc( int width, int height )
{
  g_width = width; g_height = height;
  g_projMatrix.makePerspective( 30.0f, g_width/g_height,
    1.0f, 10000.0f );
}
```

6. In the main entry, we will just put the simplest GLUT code here. There are no OSG-related functions except using `osgDB::readNodeFile()` function to read a scene graph from the disk file.

```
glutInit( &argc, argv );
glutInitWindowSize( 800, 600 );
glutInitDisplayMode( GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH );
glutCreateWindow( "OSG in GLUT" );

glutReshapeFunc( reshapeFunc );
```

```
glutDisplayFunc( displayFunc );
initializeFunc( osgDB::readNodeFile("cow.osg") );

glutMainLoop();
return 0;
```

7. Is that all? Yes, we have just finished the whole program! Compile and run, and we will see a window exactly the same as previous one using osgViewer. But it can't be manipulated, and *Esc* key can't be used to exit the program either. Mouse and keyboard events are not migrated into this recipe yet. But this is already the rudiment of a GLUT application with OSG scene graph rendered in.

How it works...

It is a little too complex to introduce the `osgUtil::SceneView` class here. It is in fact the main renderer of OSG 1.x versions and a **Producer** project that is built upon it for high-level windowing functionalities. But these are out-dated projects.

Today, OSG has a specific osgViewer library for encapsulating windowing APIs, implementing different multithreading mechanisms, and handling user events and camera manipulators. The `osgUtil::SceneView` class is only used internally for sorting and transferring actual rendering commands to the OpenGL pipeline. When we want to merge OSG with other 3D engines, we may have to directly utilize `osgUtil::SceneView` class. But we cannot benefit from OSG's user-level advantages (such as multithreaded rendering) and may have to re-invent the wheel at some point.

Running OSG examples on Android

We have already discussed about compiling OSG on Android in the first chapter of this book. Now it's time to do something more exciting. As Android uses Java as the user-level rendering surface (in 2.3 and higher, Android starts to support C++ programs fully), OSG libraries for Android must also use Java as the front end and there is too much coding work before we can make the simplest OSG program work. So we will directly compile an existing OSG on Android example and try to run it on a true device (a Motorola XOOM).

You can find the example project in `examples/osgAndroidExampleGLES1` and `examples/osgAndroidExampleGLES2` in the OSG source code, but it is not compiled along with the core libraries and other examples. That is because CMake cannot handle Android makefiles properly at present. Copy the folder to a suitable place first. Remember that I assume you have already compiled OSG with GLES v1 support. So `examples/osgAndroidExampleGLES1` should be used here.

Important! If you are using Tegra devices (mentioned in *Chapter 1*), please remember to comment a line in `jni/Android.mk` for both examples.

```
# LOCAL_ARM_NEON := true
```

Otherwise, your application will crash and exit while running on mobile devices.

Getting ready

Be sure to configure Android SDK and NDK properly first. If you are also new to Android developing, see their following official website for instructions:

`http://developer.android.com/sdk/installing.html`

It is OK if you don't use the ADT plugin for Eclipse, we will use **Apache Ant** in this recipe to generate debug/release versions of Android packages. You can read more about it at `http://ant.apache.org/`.

To obtain Ant under Ubuntu, type the following command:

```
# apt-get install ant
```

You must sign your application if you are generating release versions of Android packages (APK). In this recipe, we choose the debug build.

How to do it...

Let us start.

1. First let us make some changes to the `jni/Android.mk` file to specify the OSG installation path. Of course, it should be compiled using GLES configurations.

   ```
   # replace '< type your install directory >' with your OSG
   installation path, for instance, /usr/osg_android/
   OSG_ANDROID_DIR := < type your install directory >
   ```

2. We have two ways to start the building process. First, it is quick and simple to execute `ndk-build` directly in the example's root folder:

   ```
   # ndk-build
   ```

3. Another choice is a little more complex—change to the subfolder `jni` and compile the source code, then copy the generated libraries to the `libs` folder:

   ```
   # cd jni
   # ndk-build NDK_APPLICATION_MK=Application.mk
   # cd ..
   # cp -r obj/local/armeabi/ libs/
   # cp -r obj/local/armeabi-v7a/ libs/
   ```

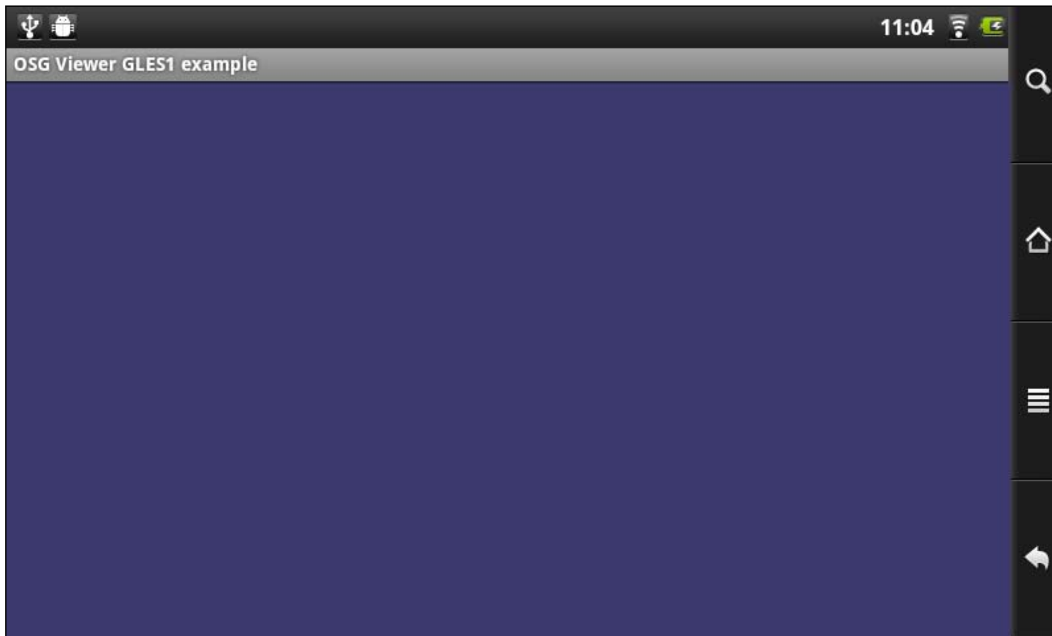
4. Change to the root folder of the example. Use `tools/android` to generate projects for Ant to use.

   ```
   # tools/android update project --name osgAndroidExampleGLES1
   --path . --target "android-14"
   ```

5. Use Ant to generate the package file with `.apk` as the extension. Debug version must be used here as it will automatically generate a public key for signing the application.

   ```
   # ant debug
   ```

6. OK, now find the file `bin/osgAndroidExampleGLES-release-unsigned.apk` (or something with a similar name) and copy it to your Android device.
7. Install and have a look at your result. It is empty except the classic blue screen, isn't it? This just means you got your foot in the door of running OSG applications on Android. Congratulations!



How it works...

We don't have enough space to explain how this example is written. Thanks to the original author of OSG on Android—Jorge Izquierdo Ciges. He contributes these two examples and makes sure they work on most Android platforms.

In short words, the example (either one) is divided into two parts: The files in `src/osg/AndroidExample` are Java source code that manages the UI elements, dialogs, and the interface to middle-level render surface; the `.cpp` files in `jni` are actually using OSG libraries for rendering. The file `osgNativeLib.cpp` uses JNI to wrap some C/C++ functionalities to Java. And the file `osgMainApp.cpp` provides a simple encapsulation of `osgViewer::Viewer` class with some more methods to load models and handle mouse events.

If you want to show something in your first OSG on Android application, try looking into `osgMainApp.cpp` and add some more code to create a triangle or draw elements with shaders. If you need to read model files of other formats, use static macros to declare plugins (refer to `examples/osgstaticviewer`) at the very beginning. You may also build cUrl and a few more libraries for Android and add them to CMake variables for building with the mobile version of OSG.

Embedding OSG into web browsers

It would be cool to create and view OSG-based applications in a web browser such as Firefox, Internet Explorer, or Google Chrome. But to design a plugin for these web browsers is not as easy as one might hope. First of all, you have to be an expert of web plugin programming. Internet Explorer accepts **ActiveX**, but others accept **NPAPI** (hopefully), so you have to do some extra work to support more than one platform. Is there any way to create powerful web plugins in a simpler and portable way?

The answer is **FireBreath**, a web plugin framework that can work either as an NPAPI plugin or as an ActiveX control. It supports all the web browsers we just described, as well as Apple's Safari and the Opera browser (partly). You may read detailed information at `http://www.firebreath.org/display/documentation/FireBreath+Home`.

In this recipe, we will learn to quickly integrate OSG into web browsers using the power of FireBreath. The example code can only work under Windows systems at present as there are some platform-specific parts (a totally platform-independent one is too long for this recipe). But you can certainly rewrite it to support more operating systems after you are familiar with the FireBreath APIs.

Getting ready

We are not going to introduce the full-build instructions of a new FireBreath project. FireBreath itself doesn't need to be compiled. Download it at `https://github.com/firebreath/FireBreath/tarball/firebreath-1.6`.

FireBreath provides a convenient **Python** utility `fbgen.py` for quickly creating new projects from templates. Please follow the link below to create a new plugin project and make sure it can compile before we continue. The **Plugin Name** is **osgWeb**, and so is the name of **Plugin Identifier**. Don't edit the default values of other fields unless you really need to make some changes.

`http://www.firebreath.org/display/documentation/Creating+a+New+Plugin+Project`

The final settings look as shown in the following screenshot:

```
Plugin Name []: osgWeb
Plugin Identifier [osgWeb]:
Plugin Prefix [OWE]:
Plugin MIME type [application/x-osgweb]:
Plugin Description []: OSG webplugin
Plugin has no UI [false]:
Company Name []: OSG
Company Identifier [OSG]:
Company Domain [osg.com]: www.openscenegraph.org
```

And you will find a new directory `osgWeb` in the `projects` folder.

How to do it...

Let us start.

1. When we have the `CMakeLists.txt` file in the project folder (in `projects/osgWeb`), open it with any text editor and modify the last few lines:

   ```
   find_path( OPENSCENEGRAPH_ROOT include/osg/Node PATHS
   $ENV{OSG_ROOT} )
   include_directories( ${OPENSCENEGRAPH_ROOT}/include )
   link_directories( ${Boost_LIBRARY_DIRS}
   ${OPENSCENEGRAPH_ROOT}/lib )
   include_platform()
   target_link_libraries(${PROJECT_NAME} OpenThreads
   osg osgDB osgGA osgViewer)
   ```

2. Follow the pre-making process in the last web link to generate the Visual Studio solution in the `build/` folder. Any other dependencies (such as **Boost**) will be downloaded automatically.
3. If you encounter any errors, use `cmake-gui` to open the `CMakeCache.txt` file and see if you have correctly configured the OSG dependencies for the project. Now it's time to do some coding work in the `osgWeb.h` and `osgWeb.cpp` files, which are originally created by FireBreath.

Ungrouped Entries	
ATLLIB	F:/Microsoft Visual Studio 10.0/VC/atlmfc/lib/atls.lib
ATLWIN	F:/Microsoft Visual Studio 10.0/VC/atlmfc/include/atlwin.h
CURL	CURL-NOTFOUND
GIT	GIT-NOTFOUND
GZIP	GZIP-NOTFOUND
MFCWIN	F:/Microsoft Visual Studio 10.0/VC/atlmfc/include/winres.h
SEVZIP	E:/Writings/OSG_Cookbook/9_INTEGRATING_WITH_GUI/firebreath/cmake/7za.exe
TAR	TAR-NOTFOUND
WGET	E:/Writings/OSG_Cookbook/9_INTEGRATING_WITH_GUI/firebreath/cmake/wget.exe
BUILD_EXAMPLES	
MFC_INCLUDE_DIR	F:/Microsoft Visual Studio 10.0/VC/atlmfc/include
OPENSCENEGRAPH_ROOT	E:/OpenSceneGraph/sdk
VC_DIR	F:/Microsoft Visual Studio 10.0/VC
VS_DIR	F:/Microsoft Visual Studio 10.0
WIX_ROOT_DIR	WIX_ROOT_DIR-NOTFOUND
ATL	
CMAKE	
FB	
WITH	

4. Include necessary headers in `osgWeb.h`.

   ```
   #include "PluginWindowWin.h"
   #include <osg/Group>
   #include <osgDB/ReadFile>
   #include <osgGA/TrackballManipulator>
   #include <osgViewer/Viewer>
   #include <osgViewer/api/Win32/GraphicsWindowWin32>
   ```

5. As described in the `QThread` example, it is possible to refresh OSG scene in a separate thread while the main process is busy with different GUI events. Now we will design such a thread class derived from `OpenThreads::Thread`, which is the base of OSG's threading solutions.

```
class RenderingThread : public OpenThreads::Thread
{
public:
  RenderingThread()
  : OpenThreads::Thread(), viewerPtr(0) {}

  virtual ~RenderingThread();

  virtual void run()

  osgViewer::Viewer* viewerPtr;
};
```

6. We have to cancel the simulation loop when the thread object is destroyed, that is, in the destructor.

```
if ( viewerPtr ) viewerPtr->setDone( true );
while( isRunning() )
OpenThreads::Thread::YieldCurrentThread();
```

7. In the virtual `run()` method, we start the rendering work in a loop and will only quit if the viewer is done or `testCancel()` returns `true`. The latter is implemented and used by OpenThreads to check the state of a running thread.

```
if ( !viewerPtr ) return;
do
{
  viewerPtr->frame();
}
while ( !testCancel() && !viewerPtr->done() );
viewerPtr = NULL;
```

8. Now we have to make some changes to the auto-generated `osgWeb` class. Let us re-implement the `onWindowAttached()` and `onWindowDetached()` methods, and add two member variables to allocate the viewer and the rendering thread.

```
class osgWeb : public FB::PluginCore
{
public:
  ...
  virtual bool onWindowAttached(FB::AttachedEvent *evt,
    FB::PluginWindow *);
  virtual bool onWindowDetached(FB::DetachedEvent *evt,
```

```
    FB::PluginWindow *);
  ...
  osgViewer::Viewer _viewer;
  RenderingThread* _thread;
};
```

9. The `onWindowAttached()` method will be called when the web browser has initialized the plugin and attached a window handle to it. This is the only chance to integrate OSG context to the browser. The following code works for Windows platforms only as it is much easier to implement. You may try writing the X11 version by casting the window parameter to `FB::PluginWindowX11`.

```
bool osgWeb::onWindowAttached(FB::AttachedEvent *evt,
  FB::PluginWindow *win)
{
  FB::PluginWindowWin* window =
    reinterpret_cast<FB::PluginWindowWin*>(win);
  if ( window )
  {
    …
  }
  return false;
}
```

10. If the window exists, we can now set the traits of the OSG context. You should be already familiar with the following code.

```
osg::ref_ptr<osg::Referenced> windata =
  new osgViewer::GraphicsWindowWin32::WindowData(
    window->getHWND() ););
// This works under Windows only
osg::ref_ptr<osg::GraphicsContext::Traits> traits =
  new osg::GraphicsContext::Traits;
traits->x = 0;
traits->y = 0;
traits->width = 800;
traits->height = 600;
traits->windowDecoration = false;
traits->doubleBuffer = true;
traits->inheritedWindowData = windata;
```

11. Create the main camera.

```
osg::ref_ptr<osg::GraphicsContext> gc =
  osg::GraphicsContext::createGraphicsContext(
    traits.get() );
osg::ref_ptr<osg::Camera> camera = new osg::Camera;
camera->setGraphicsContext( gc );
camera->setViewport( new osg::Viewport(0, 0,
  traits->width, traits->height) );
camera->setClearMask( GL_DEPTH_BUFFER_BIT |
  GL_COLOR_BUFFER_BIT );
camera->setClearColor( osg::Vec4f(0.2f, 0.2f,
  0.6f, 1.0f) );
camera->setProjectionMatrixAsPerspective(
  30.0f, (double)traits->width/(double)traits->height,
    1.0, 1000.0 );
```

12. Set the camera and an example scene graph to the viewer now. Don't allow *Esc* key to be triggered, and you may call `setKeyEventSetsDone(0)` to disable this.

```
osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( osgDB::readNodeFile("cessna.osg") );

_viewer.setCamera( camera.get() );
_viewer.setSceneData( root.get() );
_viewer.setKeyEventSetsDone( 0 );
_viewer.setCameraManipulator(
  new osgGA::TrackballManipulator );
```

13. At the end of the method, start the thread to render OSG scene!

```
_thread = new RenderingThread;
_thread->viewerPtr = &_viewer;
_thread->start();
```

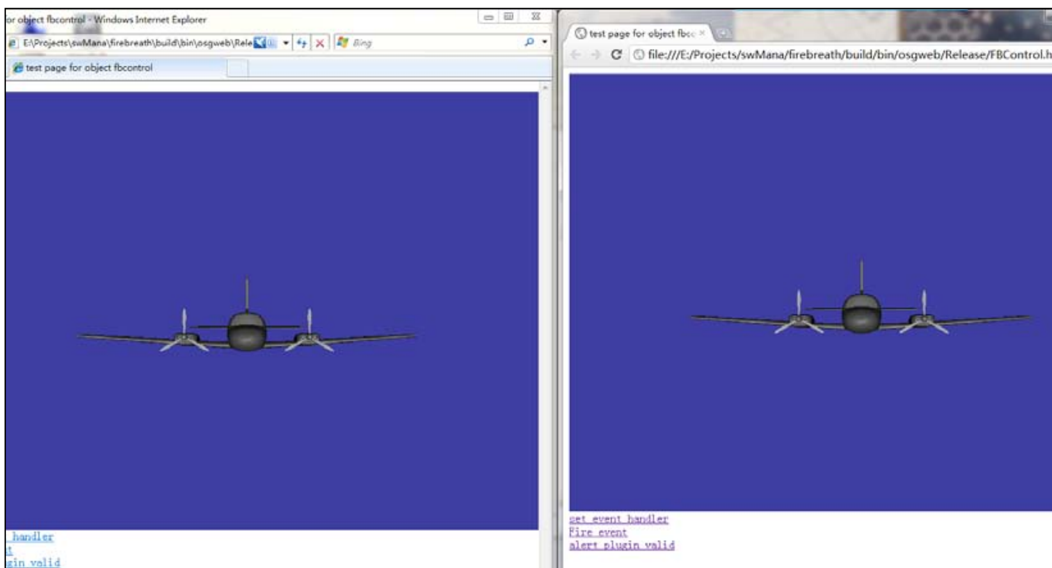
14. When the plugin is going to be reset or released, don't forget to delete the thread to cancel the rendering work.

```
bool osgWeb::onWindowDetached(FB::DetachedEvent *evt,
  FB::PluginWindow *win)
{
  delete _thread;
  return false;
}
```

15. Now compile the `osgWeb` project. It will generate a dynamic plugin named `nposgweb.dll` in `build/bin/osgweb`. Find it and copy it to a suitable position. A web page file `FBControl.htm` including the simplest code for executing the plugin is saved in `build/projects/osgweb/gen`. Don't forget to copy it out for the testing work.
16. Under Windows, you will have to register the plugin before using it. Run the following command in the plugin's directory:

    # regsvr32 nposgweb.dll

17. Now you can open the test page. Can you see the Cessna model displayed in the web browser now? (We have run it on both Chrome and Internet Explorer.)



18. If you want to unregister the plugin, type the following command:

    # regsvr32 -u nposgweb.dll

How it works...

The mechanism used in FireBreath to implement a web plugin is out of the scope of this book. To describe it in short words, it encapsulates two totally different APIs (ActiveX and NPAPI) perfectly and uses one uniform `PluginCore` class to implement all common functionalities during the plugin's lifecycle, including attaching and detaching window handle, and receiving mouse and keyboard events.

The integration of the OSG viewer nearly has no difference with earlier examples (mostly in *Chapter 4*). A window handle is required for specifying the window traits, and then we use the traits to create graphics context and the viewer's main camera. A separate thread is used to render the scene in an efficient way, as discussed in an earlier recipe in this chapter.

We will talk about the safety problem of this integration solution in the next recipe.

There's more...

Web plugin is not the only solution for displaying 3D scene in browsers. Don't forget that we can also make use of the JNI (`http://java.sun.com/docs/books/jni/`) to create Java applets with C++ and OSG-based application.

The latest **webGL** standard is another good choice for Chrome and Firefox users (but it is still blocked in some other browsers due to possible security problems). Fortunately there is already a corresponding OSG project here named `osgjs`. It is actually a WebGL framework based on OpenSceneGraph concepts, but not a wrapper based on the original OSG source code. It can be viewed and downloaded from `http://osgjs.org/`.

Designing the command buffer mechanism

The command buffer is a queue of command strings (tokens) that the application maintains. Each string represents a kind of command for parsing and executing. The application must read the command buffer's tokens sequentially and execute them sequentially too. The command buffer mechanism will be of much help especially in multithreaded applications— the commands are sent in one thread, and received and executed in another. You will have to ensure the buffer area is read/write safe using the thread mutex, because it may be operated by different threads at the same time.

In this recipe, we are going to talk about the simplest implementation of command buffer based on the web plugin example we just discussed. The command can be sent when FireBreath receives mouse events from the outside. And we will only recognize the input command string as filenames to be reloaded into the scene to replace original ones. The work of loading and updating to scene graph will be done in FireBreath's event function first, and in OSG's handler later.

How to do it...

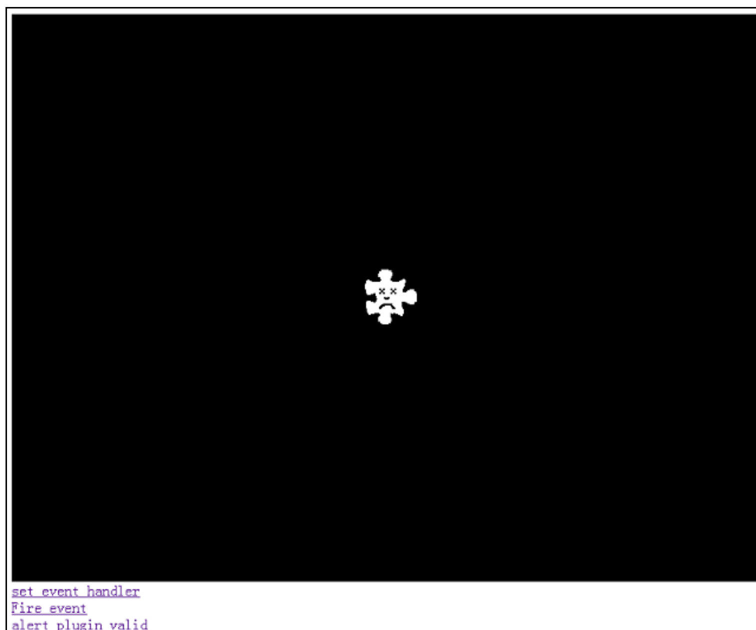
Let us start.

1. We first consider how to add a simple functionality in `osgWeb::onMouseUp()` method. Type the following code:

   ```
   osg::Group* root = dynamic_cast<osg::Group*>(
     _viewer.getSceneData() );
   ```

```
if ( root )
{
  root->removeChildren( 0, root->getNumChildren() );
  switch ( evt->m_Btn )
  {
    case FB::MouseUpEvent::MouseButton_Left:
    root->addChild( osgDB::readNodeFile("cow.osg") );
    break;
    case FB::MouseUpEvent::MouseButton_Middle:
    root->addChild( osgDB::readNodeFile("cessna.osg") );
    break;
    case FB::MouseUpEvent::MouseButton_Right:
    root->addChild( osgDB::readNodeFile("dumptruck.osg") );
    break;
  }
}
```

2. It should clear all existing child nodes under the root node and add a new model according to mouse button clicked. For example, left button up means to add a new Cessna. This is only a naive functionality for test, but it can reproduce some issues and will be solved later with some new mechanism such as the command buffer.
3. Compile and register the plugin. At the beginning, it seems to work like a charm. But after a few attempts, especially when we are clicking mouse very quickly, the plugin may crash, as shown in the following screenshot:



Anything wrong with the program? Yes, this is just because we are trying to alter OSG scene elements outside the rendering thread; neither in callback nor event handlers, but in another thread (probably the UI thread) which may cause data-sharing conflicts.

4. So we need a simple but handy command buffer to solve the problem. First let us derive the class from `osgGA::GUIEventHandler`.

```
class CommandHandler : public osgGA::GUIEventHandler
{
public:
  void addCommand( const std::string& cmd )
  {
    OpenThreads::ScopedLock<OpenThreads::Mutex> lock(mutex);
    commands.push_back( cmd );
  }

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

  std::vector<std::string> commands;
  OpenThreads::Mutex mutex;
};
```

5. In the `handle()` method, we will always swap the member command vector to a local one in the `FRAME` event. To note, a `mutex` variable is always used before the buffer `commands` is used. It is the key to protecting data from being corrupted by more than one thread's operations.

```
if ( ea.getEventType()!=osgGA::GUIEventAdapter::FRAME )
return false;

// Use a local command list as the 'front command buffer'
// Cut and paste the 'back buffer' commands to it for handling
std::vector<std::string> localCommands;
{
  OpenThreads::ScopedLock<OpenThreads::Mutex> lock(mutex);
  localCommands.swap( commands );)
}
Implement the command now, that is, to remove old scene and add
new model from file.
osgViewer::Viewer* viewer =
  static_cast<osgViewer::Viewer*>( &aa );
if ( viewer && viewer->getSceneData() )
{
  osg::Group* root = dynamic_cast<osg::Group*>(
    viewer->getSceneData() );
```

```
  if ( root )
  {
    for ( unsigned int i=0; i<localCommands.size(); ++i )
    {
      root->removeChildren( 0, root->getNumChildren() );
      root->addChild( osgDB::readNodeFile(
        localCommands[i]) );
    }
  }
}
return false;
```

6. We record the command handler variable in the `osgWeb` class.

```
class osgWeb : public FB::PluginCore
{
  ...
  osg::ref_ptr<CommandHandler> _commandHandler;
};
```

Don't forget to apply the handler to the viewer in `osgWeb::onWindowAttached()`.

```
_commandHandler = new CommandHandler;
_viewer.addEventHandler( _commandHandler.get() );
```

7. Now in the implementation of `osgWeb::onMouseUp()` method, we can add string to the command handler instead of directly operating on scene graph nodes.

```
if ( _commandHandler.valid() )
{
  switch ( evt->m_Btn )
  {
    case FB::MouseUpEvent::MouseButton_Left:
    _commandHandler->addCommand("cow.osg"); break;
    case FB::MouseUpEvent::MouseButton_Middle:
    _commandHandler->addCommand("cessna.osg"); break;
    case FB::MouseUpEvent::MouseButton_Right:
    _commandHandler->addCommand("dumptruck.osg"); break;
  }
}
return false;
```

8. Re-run the application, try your best to click on the mouse buttons as command buffer mechanism as fast as possible. You won't see any crashes again, will you? That is because command buffer stores the commands transferred between two threads (UI and OSG) and each thread will only work on these commands when the other one doesn't. The scene graph is also altered in OSG event handlers (will never conflict with the rendering process), rather than in the UI thread function.

How it works...

The `OpenThreads::Mutex` variable is of great use here. It can lock the thread in a scoped time, until all the commands after its birth and before its death are executed. The `CommandHandler` class uses mutex variable in two places: one is the `addCommand()` function, which is only called in `osgWeb::onMouseUp()`; the other is the `handle()` method, which uses mutex to protect the command list and then parses its contents.

You can easily find that `addCommand()` method is in fact only called in the UI thread, and it will lock the data `commands` before writing a new value to it. Meanwhile, `handle()` is only called in OSG, locks `commands` and reads all its elements, and then parses them to execute preset functionalities. These two locks make the data transferring process safe, stable, and clear to understand. Also you may extend the command string to any form, and use them in your own applications.

Index

Symbols

2D camera manipulator
designing 162-166
2D quad-tree 334
2D shape
extruding, to 3D 86-89
3D world
movie, playing 177, 178
_activeContextID member variables 366
_direction variable 203
_distance variable 203
_fadingState 188
_initialized member variables 366
_initialized variable 69
_needleTransform node 76
_nodeMap variable 320
_pagedReader callback 348
_plateTransform node 76
_root node 223
_scene variable 217
_viewer.frame() 357
<marquee> tag 180
-d option 290
-o option 284
-t option 290
.osgb 287
.svn file 14
--polygonal option 287, 289
--geocentric option 289
--image-ext 287
--tasks argument 293
--terrain option 287

A

ABSOLUTE_RF 249
accept() method 80, 95
Acer IconiaTab 26
ACTUAL_3DPARTY_DIR option 21
addCommand() function 396
addCommand() method 396
addFileList() function 319
addGround() method 221
addMatrixManipulator() method 160
addPhysicsBox() method 221
addPhysicsData() method 222
addPhysicsSphere() method 221
addProfile() method 274, 276
addVertices() function 212, 213
allocateImage() method 346
AlphaPixel 8
Android
OSG examples, running 383-385
Android NDK
URL, for downloading 25
Android SDK
URL, for downloading 25
Ant
obtaining under ubuntu, commad for 383
Apache Ant 383
API documentation
generating 30-32
app-cull-draw structure 361
applyEnd() method 275
apply() method 59, 275
AppServ
URL 301

apt-get command 14, 19, 355
ARM OpenGL ES 2.0 Emulator
URL 28
astronomical unit (AU) 139
AuxiliaryViewUpdater class 152, 154

B

background image node
implementing 61-64
BackMotion 183
Ball-* node 341
BFS
BFSVisitor class 60
designing 58
BFSVisitor class 60
Binary space partitioning (BSP)
URL 341
bin directory 282
binormal array 233
Bloom
URL 256
BlueMarble
URL 288
BMP 18
Boost library 43
borderlines
used, for creating polygon 82-85
BounceMotion 183
bounding box
computing 47-51
BoundingBoxCallback class 47
breadth-first-search. *See* **BFS**
brightness parameter 343
build_master.source file 292, 294
build_master.tasks file 293
build() method 335, 337
Bullet Physics
URL 225
bump mapping 119, 229-232

C

CACHE_IMAGES value 320
car
running car, creating 51-56
CEGUI
about 365
precompiled libraries , URL for downloading 365
source code, URL for downloading 365
CEGUIDrawable::drawImplementation() 370
CEGUIDrawable::initializeControls() method 372
CEGUIDrawable instance 371
CEGUI elements
embedding, in scene 365-373
CEGUI library 354
CellIndex 324
CgEndDrawCallback class 69
CgFX 68
cgGLDisableProfile() function 276, 277
Cg language 67
CGprofile object 68
CgProgram attribute 275, 277
CgProgram class 273, 274, 276
CGprogram object 68
CgStartDrawCallback class 69
checkout operation 10, 13, 15
childData 336
CIAT 290
CircMotion (circular equation) 183
clone() method 46
cloud
about 266
bounding box, computing 269, 271
CloudCell object 270
CloudCell structure 268
creating 267
makeGlow() function 270
readCloudCells() function 271
viewer, starting 271
CloudCell
about 267
object 270
structure 268
cluster generation
implementing, SSH used 298-300
CMake
generators 18
options, configuring 14-17
URL 7
URL, for binary packages 14
used, for creating project 33-36
CMAKE_BUILD_TYPE item 17

CMake, generators
about 18
CodeBlocks 18
MinGW Makefiles 18
NMake Makefiles 18
Unix Makefiles 18
Visual Studio 18
XCode 18
cmake-gui tool 297
cmake-gui utility 14, 15
CMAKE_INSTALL_PREFIX item 17
CMake options
configuring 14-17
CMake scripts 7
CodeBlocks 18
Collada DOM 18
command buffer mechanism 392-395
CommandHandler class 396
compare() method 274
compass class 74
compass node
implementing 74-80
computeBound() method 91, 92, 95, 267, 371
computeLocalToWorldMatrix() method 257, 259
computer animation 171
computeTargetToWorldMatrix() 157
computeWorldToLocalMatrix() method 257
convertMouseButton() method 371
CPack packaging system 22
Crazy Eddie's GUI. *See* **CEGUI**
create2DView() function 151, 154
createActor() function 220
createAnimateNode() 145
createAnimationPath() 48
createAnimationPathCallback() function 82
createBlurPass() function 251
createBone() function 212
createBoneShapeAndSkin() function 213
createBoneShape() function 212
createBoxForDebug() method 339
createBox() method 218
createCamera() function 356, 359
createChannel() function 208, 212
createColorInput() function 251
createDemoWidget() function 362
createElement() 337
createElement() method 338
createEmoticonGeometry() function 184
createEndBone() function 212
createFace() function 100
createFaceSelector() method 111
createFireParticles() function 191
createGraphicsContext() method 73
createHUDCamera() function 38
createInstancedGeometry() function 125, 343
createLabel() function 375
createMatrixTransform() function 44
createMaze() function 325, 326
createNewLevel() method 338
createPointSelector() method 115
createQuads() function 313
createRandomImage() function 311
createRibbon() function 102
createRTTCamera() function 228
createScene() function 140
createScreenQuad() function 229
createSimpleGeometry() function 112, 117
createSimpleGeometry() method 107
createSlaveCamera() 134
createSphere() method 219
createStaticNode() function 145
createTexture() function 263
createTilescreateTiles() function 307
createTiles() function 308
createTransformNode() function 52
createView() function 132
createWorld() method 218
cube map
sky box, designing 257-261
cube mapping 261
CubicMotion class 183
CubicMotion (cubic equation) 183
culling algorithms 324
culling strategy
about 324
designing 324-330
working 331
Cygwin 14

D

dangling pointer 42
database pager
configuring 321-323
data.txt file 342
DDS 18
deep copy 44
defaultTex variable 343
define_passes() class 238
define_techniques() method 238
degrees-of-freedom (DOF) 129
DEM 122
depth buffer
displaying 241-244
reading 241-244
depth-first-search. *See* **DFS**
depth-of-field. *See* **DOF**
depth partition
using, to display huge scene 139-143
depth partition range 141
depth partition setting 141
depth peeling 190
derivation of a normal map. *See* **DUDV map**
DFS 58
digital elevation model. *See* **DEM**
DirectInput library 166
DirectX. *See* **HLSL**
dirtyBound() 103
dirtyDisplayList() method 105
dirty() method 105
displacement mapping 119
displayFunc() function 381
DOF
implementing 249-254
door
animation, adding to manager 175
closing 172
door geometry, creating 173
handler, configuring 175
headers, including 172
OpenDoorHandler class 174
opening 172
osgAnimation::UpdateMatrixTransform 176
scene graph, creating with updater 175
StackedMatrixElement 177
StackedQuaternionElement 177
StackedRotateAxisElement 177
StackedScaleElement 177
StackedTranslateElement 177
viewer, starting 175
wall geometry, creating 172, 173
dot utility 31
doUserOperations() method 42, 107-111, 116
Doxygen tool
about 30
URL, for downloading 31
draw callbacks
used, for executing NVIDIA Cg functions 67-74
drawImplementation() method 91, 92, 95, 267, 269, 366, 367, 372
draw instanced extension 124, 125
DUDV map 266
dynamic clock
drawing, on screen 96-101
dynamic libraries
about 29
compiling 29, 30
using 29, 30
DYNAMIC_OPENSCENEGRAPH option 26
DYNAMIC_OPENTHREADS option 26

E

early-Z algorithm 331
earth
terrain database, generating 287-290
EaseMotion header 182
ElasticMotion 183
elements variable 336
elevation data
working with 290-293
emoticonSource() function 184
emoticonTarget() function 184
EnumJoysticksCallback() method 168
exec() method 357
ExpoMotion (exponential equation) 183
export command 13
extentSet[] array 336
extrusion
2D shape to 3D 86-89
about 86-89

F

FadeInOutCallback class 190
fading in effect 187-190
fading out effect 187-190
Fast Approximate Anti-Aliasing. *See* **FXXA**
FBO 241
FFmpeg library
  URL 179
FIFO 59
fire
  flight, animating on 190-193
FireBreath 386
first in, first out. *See* **FIFO**
flight
  animating, on fire 190-194
FRAME_BUFFER method 244
frame buffer object. *See* **FBO**
FRAME_BUFFER_OBJECT 244
FRAME event 318
frame() method 356, 358, 361
FreeGLUT library 379
FreeType 19
front view, model
  manipulating 152-155
  model 148-151
FXXA
  URL 256

G

Galaxian game
  creating 198-206
Gaussian Blur
  URL 256
gcanyon data 289, 294
gcanyon terrain 294
GCC 14
gc variable 151
GDAL group 282
geode1 node 46
geode2 node 45
geode3 node 46
geographic coordinate system 287
Geographic Information Systems. *See* **GIS**
geometry data
  merging 306-309
GeoTiff imagery 288
getBlurFromLinearDepth() function 256
getCellIndex() method 329
getDistanceFromEyePoint() method 189
getGLWidget() method 357
getInverseMatrix() method 163
getMatrix() method 163, 166, 219
getNodeByName() method 318
getNumScreens() method 133
getOrCreateBox() function 325
getOrCreatePlane() function 325
getParentalNodePaths() method 51
getValue() method 183
getViewMatrixAsLookAt() method 155
getVisitorType() method 67
GIS 74, 82
glBegin() function 272
glCopySubImage() function 244
glEnd() function 272
GL_FLOAT format 345
gl_MultiTexCoord* variable 127
gl_Normal variable 127
globalElements variable 339
GL_POINTS mode 342
GL_POLYGON primitive 85, 87
GL_QUAD_STRIP parameter 121
glReadPixels() 316
GL_RGBA format 345
GLSL 67
GlusterFS
  URL 300
GLUT
  OSG components, using 379-383
GLUT library 379
gl_Vertex variable 127
g_mazeMap variable 325, 332
Graph Visualization Software 31

H

handleClose() callback method 369
handleClose() method 369
handle() function 156
handle() method 153, 155, 169, 203, 221, 348, 370, 371, 394, 396
handlers, osgWidget library 378

HDR
URL 256
heads-up display. *See* **HUD**
height() 199
High Dynamic Range. *See* **HDR**
HLSL 67
home() method 164
HUD 38
huge scene
displaying, depth partition used 139-143

I

indexList variable 112
initializeControls() method 368, 369
initializeFunc() function 380
init() method 169
instance() function 217
International Centre for Tropical Agriculture 290
intersectWith() function 202
isAbsolute argument 248
isLeafNode 336

J

JNI
URL 392
joysticks
used, for manipulating view 166-170
JPEG 18

K

KCEGUI::Key::Return 370
K-dimensional tree (KDTree)
about 342, 347, 350, 351
structure, building 352
URL 342
KEY_Return value 370

L

LAN 298
lastMatrix variable 327
leaf node, octree 337
LessDepthSortFunctor functor 267
LessDepthSortFunctor structure 268
libCURL 19
libJPEG 18
libopenscenegraph-dev package 25
libopenscenegraph package 25
libopenthreads-dev package 25
libopenthreads package 25
lib subdirectory 283
LIF 245
lighting
with shaders 194-197
Light Interference Filters. *See* **LIF**
LightPosCallback class 196
LINEAR_MIPMAP_LINEAR parameter 121
LLC 288
loadedModel node 240
Local Area Network. *See* **LAN**
localToWorld matrix 48
local-to-world transformation 47
lock() method 43
LOD (level-of-details) 286
LOD nodes 341

M

MAIN_CAMERA_MASK constant 147
makeGlow() function 270
make package operation 24
manipulator
using, to follow models 159, 160
maze 324
MazeCullCallback instance 330
MazeManipulator class 326
MazeManipulator object 328
maze map 324
member _text variable 41
mesh
customized mesh, skinning 211-215
META_Effect macro 238
META_StateAttribute macro 273
MinGW Makefiles 18
mobile devices
OSG, compiling on 25-27
OSG, using on 25-27
model
following, manipulator used 159, 160
front view, showing 148-151
highlighting 106-109

moving model, following 155-158
point, selecting 114-118
ribbon, drawing from 101-105
selecting 106-109
side view, showing 148-151
top view, showing 148-151
triangle face, selecting 110-113
morph geometry
implementing 183
implementing, steps 183-186
Motion Blur
URL 256
motions 183
Motorola XOOM 26
movie
playing, in 3D world 177, 178
MSDN site 43
multiple imagery
working with 290-293
multiple passes
transparency, implementing 237-241
multiple screens
views, setting up on 130-133
mutex variable 394

N

ndk-build 384
NeHe OpenGL tutorials
URL 58
Network File System. *See* **NFS**
Newton
URL 225
NFS 299
night vision effect
about 245
implementing 246-248
NMake Makefiles 18
node
screen, facing to 64-67
NodeCallback class 329
NodeKit 374
NodeMap variable 221
Non-Uniform Rational B-Splines. *See* **NURBS**
normalColor 106
notifyDisplaySizeChanged() method 373
NPAPI 386
NSIS
about 22
URL, for downloading 22
Nullsoft Scriptable Install System. *See* **NSIS**
numInstances parameter 344
NURBS 89
NURBS surface
drawing 89-95
NurbsSurface class 90
NVIDIA Cg functions
executing, draw callbacks used 67-74
NVIDIA PhysX library 215
NVIDIA shader library
URL 256
NVTT 315
for device-independent generation 296-298

O

objects
cloning 43-46
culling, occlusion query used 331-333
sharing 43-46
ObserveShapeCallback class 41
occlude node class 331
occluders 331
occlusion query
about 331
using, for culling objects 331-333
octree 306
octree algorithm 335
octree structure 334
ODE
URL 225
OIS 170
onWindowAttached() method 388, 389
onWindowDetached() method 388
OpenDoorHandler class 174
OpenGL. *See also* **GLSL**
OpenGL
about 8
URL 127
OPENGL_egl_LIBRARY option 27
OpenGL ES 25
OpenGL for Embedded Systems. *See* **OpenGL ES**
OPENGL_gl_LIBRARY option 27

OPENGL_glu_LIBRARY option 27
OPENGL_INCLUDE option 27
openNURBS
URL 96
OpenSceneGraph. *See also* **OSG**
OpenSceneGraph
about 7
official download link 8
URL, for binaries 8
URL, for online installer 8
URL, for resources 8
OpenSceneGraph 3.0 8
openscenegraph-doc 25
openscenegraph-examples package 25
OpenSceneGraph library 8
openscenegraph package 25
OpenSSH website
URL 300
OpenThreads::Mutex variable 396
openthreads-doc package 25
OpenThreads library 359
operator() implementation 48, 65
operator() method 69, 103, 181
Oriented Input System. *See* **OIS**
ortho camera 377
osg
Light's setPosition() method 197
PolygonOffset attribute 112
OSG. *See also* **OpenSceneGraph**
OSG
about 7
compiling, on different platforms 22-24
compiling, on mobile devices 25-27
culling strategy, designing 324-330
database pager, configuring 321-323
embedding, in web browser 386-392
examples, list 354
geometry data, merging 306-309
integrating, with Qt 354-358
latest version, checking out 8, 9
latest version, checking out for Ubuntu users 9-11
latest version, checking out for Windows users 9-13
massive data, managing 305, 306
objects culling, occlusion query used 331-333
osgQtBrowser 354
osgQtWidgets 354
osgviewerCocoa 354
osgviewerFOX 354
osgviewerGLUT 354
osgviewerGTK 354
osgviewerIPhone 354
osgviewerMFC 354
osgviewerQt 354
osgviewerSDL 354
osgviewerWX 354
packaging, on different platforms 22-24
plugins, building 18-21
point cloud data 342
point cloud data, rendering with draw instancing 342-346
scene intersections, speeding 347-352
scene objects, managing with octree algorithm 333-341
scene objects, sharing 316-320
textures, compressing 310-315
using, on mobile devices 25-27
osg::Billboard class 64
osg::BlendFunc attribute 271
osg::Camera class 74
osg::Camera node 228
osg::CollectOccludersVisitor 67
osg::ComputeBoundsVisitor class 47, 48
osg::computeLocalToWorld() function 51
osg::computeWorldToLocal() function 51
osg::Depth attribute 258
osg::Drawable class 74, 89, 90, 366
osg::Drawable object
display list, generating 309
osg::DrawArrays class 125
osg::DrawElements* class 125
osg::Geode node 42, 44, 97, 185, 193, 207, 371
osg::Geometry class 85
osg::Geometry object 82, 106, 109, 186
osg::GraphicsContext object 73, 131
osg::Group class 80
osg::Group's traverse() method 80
osg::image class 296, 346
osg::Image object 364
osg::ImageStream class 177
osg::Material object 188

osg::Matrix::scale() function 57
osg::Matrix::translate() function 57
osg::MatrixTransform node 103, 113, 194
osg::Node class 59
osg::NodeCallback class 331
osg::NodeVisitor class 58, 61, 378
osg::OccluderNode class 331
osg::OcclusionQueryNode class 331
osg::observer_ptr<> 40
osg::observer_ptr<> template class 40, 41, 42
osg::PagedLOD node 321, 351
osg::PrimitiveSet's sub-classes 85
osg::ProxyNode node 321
osg::ShapeDrawable object 44
osg::StateAttribute class 273
osg::StateAttribute derived class 273
osg::TexGen class 262
osg::Texture class 296
osg::TextureCubeMap class 262
osg::Timer object 348
osg::Uniform::Callback structure 197
osg::Uniform class 197
osgAnimation::Bone node 207
osgAnimation::LinearMotion class 182
osgAnimation::LinearMotion object 181
osgAnimation::MorphGeometry 183
osgAnimation::MorphGeometry object 187
osgAnimation::RigGeometry class 212
osgAnimation::RigGeometry object 214
osgAnimation::UpdateBone callback 207
osgAnimation::UpdateMatrixTransform 176
osgAnimation::UpdateMorph 187
osgAnimation::UpdateMorph object 185
osgAnimation::VertexInfluenceMap object 213
osgAnimation library 172
osganimationmorph 187
OSG_BUILD_PLATFORM_ANDROID option 26
osgCg module 74
OSG components
using, in GLUT 379-383
osgCookBook
createRTTCamera() function 248
osgCookBook::createRTTCamera() function 241, 244
osgCookBook::createText() method 39
osgCookBook::PickHandler auxiliary class 40
osgCookbook namespace 39, 82
osgCookBook namespace 228, 306
OSG_CPP_EXCEPTIONS_AVAILABLE option 26
osgDB::DatabasePager class 301
osgDB::DatabasePager object 321
osgDB::FileLocationCallback object 302
osgDB::Options class 320
osgDB::ReadFileCallback 316
osgDB::readImageFile() function 320
osgDB::readImageFile() method 18
osgDB::readNodeFile() function 292, 381
osgDB::readNodeFile() method 18, 144
osgDB::ReaderWriter class 298
osgDB::SharedStateManager 316
osgdb_curl plugin 20, 301
osgdb_directshow 179
osgdb_ffmpeg plugin 177, 179
osgdb_freetype plugin 21
osgdb_gif plugin 21, 179
osgdb_jpeg plugin 21
osgdb_nvtt plugin 298
osgdb_png plugin 21
osgdb_* prefix 18
osgdb_QTKit 180
osgdb_quicktime 180
osgdem 283
osgdem utility 280
osgdrawinstanced 127
osgEarth 305
osgEarth project 280
OSG examples
running, on Android 383-385
OSG_FILE_PATH variable 13
osgFX::Cartoon node 109
osgFX::Effect node 237
osgFX::Technique node 238
osgGA
FirstPersonManipulator class 159
osgGA::CameraManipulator abstract class 161
osgGA::CameraManipulator class 162
osgGA::EventVisitor 67
osgGA::FirstPersonManipulator 326
osg::Geometry object 307
osgGA::GUIEventAdapter object 170

osgGA::GUIEventHandler class 394
osgGA::KeySwitchMatrixManipulator class 162
osgGA::NodeTrackerManipulator class 159, 162
osgGA::OrbitManipulator object 156
osgGA::StandardManipulator class 162
osgGA::StandardManipulator instance 164
osgGA::TrackballManipulator class 130, 162
OSG_GL1_AVAILABLE option 26
OSG_GL2_AVAILABLE option 26
OSG_GL3_AVAILABLE option 26
OSG_GL_DISPLAYLISTS_AVAILABLE option 26
OSG_GLES1_AVAILABLE option 26, 28
OSG_GLES2_AVAILABLE option 26
OSG_GL_FIXED_FUNCTION_AVAILABLE option 28
OSG_GL_MATRICES_AVAILABLE option 28
OSG_GL_VERTEX_ARRAY_FUNCS_AVAILABLE option 28
OSG_GL_VERTEX_FUNCS_AVAILABLE option 28
OSG group 282
osg::Image class 346
osgmultiplerendertargets 256
osgNativeLib.cpp 385
osgParticle library 172
osgQt::GraphicsWindowQt class 354
osgQt::GraphicsWindowQt instance 356
osgQt::GraphicsWindowQt object 357
osgQt::QGraphicsViewAdapter class 364
osgQt::QWidgetImage class 364
osgQt::QWidgetImage object 362, 363
osgQtBrowser 354
osgQt library 21
osgQtWidgets 354
osgShadow::ShadowedScene node 237
osgShadow::ViewDependentShadowMap class 237
OSG source code 58
osgTerrain::TerrainTile class 287
osgTerrain::TerrainTile node 292
osgText::Text class 64
osgUtil
  Intersector class 118
osgUtil::CullVisitor 67
osgUtil::CullVisitor class 189
osgUtil::CullVisitor object 65, 259
osgUtil::GLObjectsVisitor 67
osgUtil::IntersectionVisitor 67
osgUtil::IntersectionVisitor::ReadCallback class 351
osgUtil::PrintVisitor class 60, 339
osgUtil::SceneView class 380, 382, 383
osgUtil::TangentSpaceGenerator class 230
osgUtil::TangentSpaceGenerator tool 233
osgUtil::Tessellator class 85
osgUtil::UpdateVisitor 67
osgViewer
  about 322
  Viewer class 385
osgViewer::CompositeViewer class 132
osgViewer::InteractiveImageHandler 365
osgViewer::InteractiveImageHandler class 363
osgViewer::Viewer::run() method 361
osgviewerCocoa 354
osgviewerFOX 354
osgviewerGLUT 354
osgviewerGTK 354
osgviewerIPhone 354
osgviewerMFC 354
osgviewerQt 354
osgviewerSDL 354
osgviewerWX 354
osgWeb::onMouseUp() method 392, 395
osgWeb::onWindowAttached() 395
osgWidget::Box 378
osgWidget::CameraSwitchHandler 379
osgWidget::Canvas 378
osgWidget::createExample() function 379
osgWidget::KeyboardHandler 378
osgWidget::MouseHandler 378
osgWidget::ResizeHandler 379
osgWidget library 354
  handlers 378
  using 374-379
OSG_WINDOWING_SYSTEM option 26
OVERRIDE mask 232

P

PagedPickHandler class 348, 350
paintEvent() method 357, 358
parent node 330
PBO 245
performMovementLeftMouseButton() method 165, 169
performMovementRightMouseButton() method 165, 169
PhysicsUpdater class 220, 225
PhysXInterface class 217
PhysX SDK object 217
PhyxXInterface class 225
PIXEL_BUFFER method 245
Pixel Buffer Object. *See* **PBO**
PIXEL_BUFFER_RTT method 245
PluginCore class 391
plugins
  building 18-21
point
  selecting, for model 114-118
point cloud data
  about 342
  rendering, draw instancing used 342-346
pointer
  dangling pointer 42
  smart pointer 40
  strong pointer 40
  weak pointer 43
polygon
  creating, with borderlines 82-85
pos variable 127
power-wall
  slave cameras, using 133-138
processEvents() method 363
Process Explorer
  about 310
  download link 310
project
  creating, CMake used 33-36
prune() function 317
Python utility 386

Q

QApplication variable 362
QGLWidget class 354
QGraphicsItem class 362
QGraphicsItem structure 362
QGraphicsView 362
QGraphicsView structure 362
Qt
  downloading 355
  OSG, integrating with 354-358
  SDK, downloading 355
QThread class 359, 361
QTimer 357
QT_QMAKE_EXECUTABLE variable 355
Qt toolkit 21
Qt widgets
  embedding, in scene 361-365
QuadMotion (quadratic equation) 183
quad-tree internal node 333
quad-tree scene graphs 333
Qualcomm Adreno SDK
  URL 28
QWidget 356

R

radar map
  implementing 143
  implementing, steps 144-148
randomMatrix() function 306
randomValue() function 306
randomVector() function 306
raw pointer 43
ReadAndShareCallback 319
readCloudCells() function 271
ReaderWriterDAE class 298
readNodeFile() 351
readNode() function 317
readNode() method 320
readPointData() method 344
real-time water rendering
  creating 262
RELATIVE_RF 249
removeAttribute() method 277
RemoveModelHandler class 318
RemoveShapeHandler class 40, 42
RemoveShapeHandler instance 42
renderCells() method 269
rendering loop
  starting, in separate threads 359-361

Rendering-to-texture. *See* **RTT**
render-to-texture technique 56
reset() method 59
resizeAllWindows() method 377
ribbon
drawing, from model 101-105
R-Tree
URL 342
RTT 241
run() method 150, 359, 388

S

S3TC DXT1 format 314
scene
CEGUI elements, embedding 365-373
Qt widgets, embedding 361-365
scene culling 324
scene graph
mirrorring 56, 58
scene intersections
speeding 347-352
scene objects
managing, octree algorithm used 333-341
sharing 316-320
SceneView object 380, 381
screen
dynamic clock, drawing 96-101
screenNum value 132
screenNum variable 133
Screen Space Ambient Occlusion. *See* **SSAO**
scrolling text
about 180
designing, steps 180, 181, 182
ScrollTextCallback class 180
SDK 365
SDK package
URL 179
Secure Shell. *See* **SSH**
selectedColor 106
SelectModelHandler class 110, 115
SelectModelHandler object 109
SEPERATE_WINDOW method 245
setAllowEventFocus(true) function 372
setBuildKdTreesHint() method 349
setByInverseMatrix() method 164
setByMatrix() method 164, 166
setColorArray() line 100
setCullingActive() method 258
setCullMaskLeft() method 148
setCullMask() method 148
setCullMaskRight() method 148
setDoPreCompile() 322, 323
setDrawableColor() method 107
setEnvironmentMap() method 258
setFileLocationCallback() method 302
setHomePosition() method 161, 164
setInitialBound() method 122, 123
setInternalFormatMode() method 314
setLayer() method 375
setLightingMode() method 148
setLight() method 148
setMatrixInSkeletonSpace() method 211
setMatrix() method 66
setMaxChildNumber() method 335
setMaxTexturePoolSize() method 323
setMaxTreeDepth() method 335
setNearFarRatio() method 143
setObjectCacheHint() method 320
setProjectionMatrix() method 256
setReadCallback() method 348
setReadFileCallback() 320
setRenderBinDetails() method 258
setRenderOrder() 152
SetShapeColorHandler class 44
setSpeedVector() 199
setTargetMaximumNumberOfPageLOD() method 322, 323
setTessellationType() method 85
setUnrefImageDataAfterApply() 315
setUpDepthPartition() method 143
setUpViewInWindow() method 71, 73
setUpView*() method 73
setVelocity() method 219
setVisibilityThreshold() 333
shaders
dynamically lighting 194-197
vertex-displacement mapping, using 119-123
shadow
view-dependent shadow, simulating 233-237
shallow copy 44
side view, model
manipulating 152-155

model 148-151
simulate() method 220
SineMotion (sinusoidal equation) 183
skeleton system
building 206-210
sky box
designing, with cube map 257-261
SkyBox constructor 258
slave cameras
using, for power-wall 133-138
small feature culling 148
small terrain database
generating 283-286
smart pointer 40
source code control 13
SSAO
URL 256
SSH
about 299
using, to implement cluster generation 298-300
StackedMatrixElement 177
StackedQuaternionElement 177
StackedRotateAxisElement 177
StackedScaleElement 177
StackedTranslateElement 177
state attribute
customizing 273-277
static libraries
about 29
compiling 29, 30
using 29, 30
strong pointer 40
Subversion tool 9, 13
sudo command 10
supports() method 95

T

tabPressed() callback function 374
tabPressed() function 376
TaharezLook style 369
tangent array 233
terrain
loading, from internet 301-303
rendering, from internet 301-303
terrain database
existing database, patching with newer data 293-295
generating, on earth 287-290
TESS_TYPE_GEOMETRY 85, 87
TESS_TYPE_POLYGONS 85, 87
testCancel() 388
textures
compressing 310-315
internal format mode 315
theosgGA::GUIEventHandler class 169
threads
rendering loop, starting 359-361
top view, model
manipulating 152-155
model 148-151
TortoiseSVN 9
TrailerCallback 103
Traits class 133, 353
transformation node 51
transparency
implementing, with multiple passes 237-241
TRANSPARENT_BIN hint 239
traverseBFS() 59
traverse() method 67, 75, 331
triangle face
selecting, for model 110-113
TrueMarble
URL 288
TwoDimManipulator constructor 168

U

Ubuntu users
OSG's latest version, checking out for 9-11
Unix Makefiles 18
update() method 183, 201, 357
update operation 13
USE_DOTOSGWRAPPER_LIBRARY() macro 30
USE_GRAPHICSWINDOW() macro 30
USE_OSGPLUGIN() macro 30
USE_SERIALIZER_WRAPPER_LIBRARY() macro 30

V

validate() method 238
VBO
about 105, 309
features, URL 105
Vertex Buffer Objects. *See* **VBO**
vertex-displacement mapping
using, in shaders 119-123
vertical sync feature 309
view
manipulating, joysticks used 166-170
setting up, on multiple screens 130-133
view-dependent shadow
simulating 233-237
viewer.run() method 150, 248
ViewerWidget class 356, 357, 360
ViewerWidget constructor 358
view-frustum culling 148
VirtualPlanetBuilder. *See* **VPB 13**
Visual Studio 18
VPB
about 280
building 281, 282
requisites 280
URL 280
vpb::DatabaseBuilder object 292
vpbcache 283
vpbmaster 282, 283
vpbsizes 283

W

water effect
creating 262-266
weak pointer 43
weak_ptr implementation 43
web browser
OSG, embedding 386-392
webGL standard 392
WGS-84 289
width() 199
WindowingSystemInterface class 130-133
Windows 7 310
Windows users
OSG's latest version, checking out for 9-13
WinZIP 22
WolfenQt
URL 361
Wolfenstein 362
World Geodetic System 1984. *See* **WGS-84**

X

XCode 14, 18
XOY plane 57

Z

Zlib library 21

Thank you for buying
OpenSceneGraph 3 Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

Ext JS 4 Web Application Development Cookbook

ISBN: 978-1-84951-686-0 Paperback: 450 pages

Over 130 easy to follow recipe backed up with real life examples, walking you through the basic Ext JS features to advanced application design using Sencha Ext JS

1. Learn how to build Rich Internet Applications with the latest version of the Ext JS framework in a cookbook style
2. From creating forms to theming your interface, you will learn the building blocks for developing the perfect web application
3. Easy to follow recipes step through practical and detailed examples which are all fully backed up with code, illustrations, and tips

---

Sencha Touch 1.0 Mobile JavaScript Framework

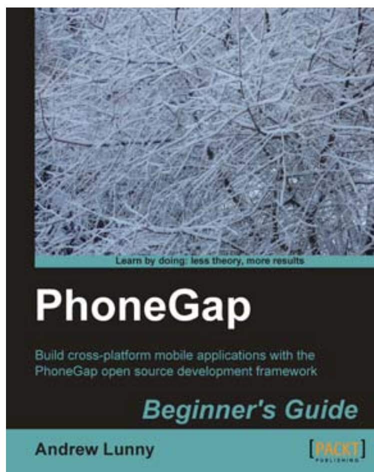
ISBN: 978-1-84951-510-8 Paperback: 300 pages

Build web applications for Apple iOS and Google Android touchscreen devices with this first HTML5 mobile framework

1. Learn to develop web applications that look and feel native on Apple iOS and Google Android touchscreen devices using Sencha Touch through examples
2. Design resolution-independent and graphical representations like buttons, icons, and tabs of unparalleled flexibility
3. Add custom events like tap, double tap, swipe, tap and hold, pinch, and rotate

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING



PhoneGap Beginner's Guide

ISBN: 978-1-84951-536-8 Paperback: 328 pages

Build cross-platform mobile applications with the PhoneGap open source development framework

1. Learn how to use the PhoneGap mobile application framework
2. Develop cross-platform code for iOS, Android, BlackBerry, and more
3. Write robust and extensible JavaScript code
4. Master new HTML5 and CSS3 APIs

HTML5 Mobile Development Cookbook

ISBN: 978-1-84969-196-3 Paperback: 254 pages

Over 60 recipes for building fast, responsive HTML5 mobile websites for iPhone 5, Android, Windows Phone, and Blackberry

1. Solve your cross platform development issues by implementing device and content adaptation recipes
2. Maximum action, minimum theory allowing you to dive straight into HTML5 mobile web development
3. Incorporate HTML5-rich media and geo-location into your mobile websites

Please check **www.PacktPub.com** for information on our titles