

**Muhammad Mobeen Movania, David Wolff,
Raymond C. H. Lo, William C. Y. Lo**

OpenGL – Build high performance graphics

Learning Path

Assimilate the ideas shared in the course to utilize the power of OpenGL to perform a wide variety of tasks



Packt

OpenGL – Build high performance graphics

Assimilate the ideas shared in the course to utilize the power of OpenGL for performing a wide variety of tasks

A course in three modules

Packt

BIRMINGHAM - MUMBAI

OpenGL – Build high performance graphics

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course 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 course.

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

Published on: May 2017

Production reference: 1050517

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

ISBN 978-1-78829-672-4

www.packtpub.com

Credits

Authors

Muhammad Mobeen Movania
David Wolff
Raymond C. H. Lo
William C. Y. Lo

Content Development Editor

Vikas Tiwari

Reviewers

Bastien Berthe
Dimitrios Christopoulos
Oscar Ripples Mateu
Bartłomiej Filipek
Thomas Le Guerroué-Drévillon
Muhammad Mobeen Movania
Dario Scarpa
Javed Rabbani Shah
Samar Agrawal
Sebastian Eichelbaum
Oscar Ripples
Qing Zhang

Production Coordinator

Shantanu N. Zagade

Preface

OpenGL is a fully functional, cross-platform API widely adopted across the industry for 2D and 3D graphics development. It is mainly used for game development and applications, but is equally popular in a vast variety of additional sectors. This practical course will help you gain proficiency with OpenGL and build compelling graphics for your games and applications.

What this learning path covers

Module 1, OpenGL Development Cookbook - This is your go to guide to learn graphical programming techniques and implement 3D animations with OpenGL. This straight-talking cookbook is perfect for intermediate C++ programmers who want to exploit the full potential of OpenGL. Full of practical techniques for implementing amazing computer graphics and visualizations, using OpenGL. Crammed full of useful recipes, OpenGL Development Cookbook will help you exploit OpenGL to its full potential.

Module 2, OpenGL 4.0 Shading Language Cookbook, Second Edition – With Version 4, the language has been further refined to provide programmers with greater power and flexibility, with new stages such as tessellation and compute. OpenGL Shading Language 4 Cookbook is a practical guide that takes you from the fundamentals of programming with modern GLSL and OpenGL, through to advanced techniques. The recipes build upon each other and take you quickly from novice to advanced level code.

Module 3, OpenGL Data Visualization Cookbook – From a mobile device to a sophisticated high-performance computing cluster, OpenGL libraries provide developers with an easy-to-use interface to create stunning visuals in 3D in real time for a wide range of interactive applications. This is an easy-to-follow, comprehensive Cookbook showing readers how to create a variety of real-time, interactive data visualization tools. Each topic is explained in a step-by-step format. A range of hot topics is included, including stereoscopic 3D rendering and data visualization on mobile/wearable platforms. By the end of this guide, you will be equipped with the essential skills to develop a wide range of impressive OpenGL-based applications for your unique data visualization needs, on platforms ranging from conventional computers to the latest mobile/wearable devices.

What you need for this learning path

Module 1:

The module assumes that the reader has basic knowledge of using the OpenGL API. The example code distributed with this module contains Visual Studio 2010 Professional version project files. In order to build the source code, you will need freeglut, GLEW, GLM, and SOIL libraries. The code has been tested on a Windows 7 platform with an NVIDIA graphics card and the following versions of libraries:

- freeglut v2.8.0 (latest version available from:
<http://freeglut.sourceforge.net>)
- GLEW v1.9.0 (latest version available from:
<http://glew.sourceforge.net>)
- GLM v0.9.4.0 (latest version available from: <http://glm.g-truc.net>)
- SOIL (latest version available from: <http://www.lonesock.net/soil.html>)

We recommend using the latest version of these libraries. The code should compile and build fine with the latest libraries.

Module 2:

The recipes in this module use some of the latest and greatest features in OpenGL 4.x. Therefore, in order to implement them, you'll need graphics hardware (graphics card or onboard GPU) and drivers that support at least OpenGL 4.3. If you're unsure about what version of OpenGL your setup can support, there are a number of utilities available for determining this information. One option is GLview from Realtech VR, available at: <http://www.realtech-vr.com/glview/>. If you're running Windows or Linux, drivers are readily available for most modern hardware. However, if you're using MacOS X, unfortunately, you may need to wait. As of this writing, the latest version of MacOS X (10.9 Mavericks) only supports OpenGL 4.1.

Once you've verified that you have the required OpenGL drivers, you'll also need the following:

- A C++ compiler. On Linux, the GNU Compiler Collection (gcc, g++, and so on) may already be available, and if not, it should be available through your distribution's package manager. On Windows, Microsoft Visual Studio will work fine, but if you don't have a copy, then the MinGW compiler (available from <http://mingw.org/>) is a good option.
- The GLFW library Version 3.0 or later, available from <http://www.glfw.org/>. This library provides OpenGL context creation, window support, and support for user input events.
- The GLM library Version 0.9.4 or later, available from <http://glm.g-truc.net/>. This provides mathematics support with classes for matrices, vectors, common transformations, noise functions, and much more.

Module 3:

This module supports a wide range of platforms and open source libraries, ranging from Windows, Mac OS X, or Linux-based desktop applications to portable Android-based mobile applications. You will need a basic understanding of C/C++ programming and background in basic linear algebra for geometric models.

The following are the requirements for chapters 1 to 3:

- **OpenGL version:** 2.0 or higher (easy to test on legacy graphics hardware).
- **Platforms:** Windows, Mac OS X, or Linux.
- **Libraries:** GLFW for OpenGL Windows/context management and handling user inputs. No additional libraries are needed, which makes it very easy to integrate into existing projects.
- **Development tools:** Windows Visual Studio or Xcode, CMake, and gcc.
- The following are the requirements for chapters 4 to 6:
 - **OpenGL version:** 3.2 or higher.
 - **Platforms:** Windows, Mac OS X, or Linux.
 - **Libraries:** Assimp for 3D model loading, SOIL for image and texture loading, GLEW for runtime OpenGL extension support, GLM for matrix operations, and OpenCV for image processing.
 - **Development tools:** Windows Visual Studio or Xcode, CMake, and gcc.
- The following are the requirements for chapters 7 to 9:
 - **OpenGL version:** OpenGL ES 3.0.

- **Platforms:** Linux or Mac OS X for development, and Android OS 4.3 and higher (API 18 and higher) for deployment.
- **Libraries:** OpenCV for Android and GLM.
- **Development tools:** Android SDK, Android NDK, and Apache Ant in Mac OS X or Linux.
- For more information, keep in mind that the code in this module was built and tested with the following libraries and development tools in all supported platforms:
 - OpenCV 2.4.9 (<http://opencv.org/downloads.html>)
 - OpenCV 3.0.0 for Android (<http://opencv.org/downloads.html>) SOIL (<http://www.lonesock.net/soil.html>)
 - GLEW 1.12.0 (<http://glew.sourceforge.net/>)
 - GLFW 3.0.4 (<http://www.glfw.org/download.html>)
 - GLM 0.9.5.4 (<http://glm.g-truc.net/0.9.5/index.html>)
 - Assimp 3.0 (http://assimp.sourceforge.net/main_downloads.html)
 - Android SDK r24.3.3 (<https://developer.android.com/sdk/index.html>)
 - Android NDK r10e (<https://developer.android.com/ndk/downloads/index.html>)
 - Windows Visual Studio 2013 (<https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx>)
 - CMake 3.2.1 (<http://www.cmake.org/download/>)

Who this learning path is for

The course is appropriate for anyone who wants to develop the skills and techniques essential for working with OpenGL to develop compelling 2D and 3D graphics.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in 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 at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, 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 this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/OpenGL-Build-High-Performance-Graphics>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

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 courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1

OpenGL Development Cookbook

*Over 40 recipes to help you learn, understand, and implement
modern OpenGL in your applications*

Module 1: OpenGL Development Cookbook

Chapter 1: Introduction to Modern OpenGL	7
Introduction	7
Setting up the OpenGL v3.3 core profile on Visual Studio 2010 using the GLEW and freeglut libraries	8
Designing a GLSL shader class	16
Rendering a simple colored triangle using shaders	19
Doing a ripple mesh deformer using the vertex shader	28
Dynamically subdividing a plane using the geometry shader	37
Dynamically subdividing a plane using the geometry shader with instanced rendering	45
Drawing a 2D image in a window using the fragment shader and the SOIL image loading library	48
Chapter 2: 3D Viewing and Object Picking	55
Introduction	55
Implementing a vector-based camera with FPS style input support	56
Implementing the free camera	59
Implementing the target camera	63
Implementing view frustum culling	66
Implementing object picking using the depth buffer	72
Implementing object picking using color	74
Implementing object picking using scene intersection queries	76
Chapter 3: Offscreen Rendering and Environment Mapping	81
Introduction	81
Implementing the twirl filter using fragment shader	82
Rendering a skybox using the static cube mapping	85
Implementing a mirror with render-to-texture using FBO	89

Table of Contents

Rendering a reflective object using dynamic cube mapping	93
Implementing area filtering (sharpening/blurring/embossing) on an image using convolution	98
Implementing the glow effect	101
Chapter 4: Lights and Shadows	107
Introduction	107
Implementing per-vertex and per-fragment point lighting	108
Implementing per-fragment directional light	114
Implementing per-fragment point light with attenuation	117
Implementing per-fragment spot light	120
Implementing shadow mapping with FBO	122
Implementing shadow mapping with percentage closer filtering (PCF)	128
Implementing variance shadow mapping	132
Chapter 5: Mesh Model Formats and Particle Systems	141
Introduction	141
Implementing terrains using the height map	142
Implementing 3ds model loading using separate buffers	146
Implementing OBJ model loading using interleaved buffers	157
Implementing EZMesh model loading	163
Implementing simple particle system	171
Chapter 6: GPU-based Alpha Blending and Global Illumination	181
Introduction	181
Implementing order-independent transparency using front-to-back peeling	182
Implementing order-independent transparency using dual depth peeling	189
Implementing screen space ambient occlusion (SSAO)	195
Implementing global illumination using spherical harmonics lighting	202
Implementing GPU-based ray tracing	207
Implementing GPU-based path tracing	213
Chapter 7: GPU-based Volume Rendering Techniques	219
Introduction	219
Implementing volume rendering using 3D texture slicing	220
Implementing volume rendering using single-pass GPU ray casting	228
Implementing pseudo-isosurface rendering in single-pass GPU ray casting	232
Implementing volume rendering using splatting	237
Implementing transfer function for volume classification	244
Implementing polygonal isosurface extraction using the Marching Tetrahedra algorithm	248
Implementing volumetric lighting using the half-angle slicing	254

Table of Contents

Chapter 8: Skeletal and Physically-based Simulation on the GPU	261
Introduction	261
Implementing skeletal animation using matrix palette skinning	262
Implementing skeletal animation using dual quaternion skinning	273
Modeling cloth using transform feedback	279
Implementing collision detection and response on a transform feedback-based cloth model	290
Implementing a particle system using transform feedback	296
Index	307

Module 1

OpenGL Development Cookbook

*Over 40 recipes to help you learn, understand, and implement
modern OpenGL in your applications*

1

Introduction to Modern OpenGL

In this chapter, we will cover:

- ▶ Setting up the OpenGL v3.3 core profile on Visual Studio 2010 using the GLEW and freeglut libraries
- ▶ Designing a GLSL shader class
- ▶ Rendering a simple colored triangle using shaders
- ▶ Doing a ripple mesh deformer using the vertex shader
- ▶ Dynamically subdividing a plane using the geometry shader
- ▶ Dynamically subdividing a plane using the geometry shader with instanced rendering
- ▶ Drawing a 2D image in a window using the fragment shader and SOIL image loading library

Introduction

The OpenGL API has seen various changes since its creation in 1992. With every new version, new features were added and additional functionality was exposed on supporting hardware through extensions. Until OpenGL v2.0 (which was introduced in 2004), the functionality in the graphics pipeline was fixed, that is, there were fixed set of operations hardwired in the graphics hardware and it was impossible to modify the graphics pipeline. With OpenGL v2.0, the shader objects were introduced for the first time. That enabled programmers to modify the graphics pipeline through special programs called shaders, which were written in a special language called OpenGL shading language (GLSL).

After OpenGL v2.0, the next major version was v3.0. This version introduced two profiles for working with OpenGL; the core profile and the compatibility profile. The core profile basically contains all of the non-deprecated functionality whereas the compatibility profile retains deprecated functionality for backwards compatibility. As of 2012, the latest version of OpenGL available is OpenGL v4.3. Beyond OpenGL v3.0, the changes introduced in the application code are not as drastic as compared to those required for moving from OpenGL v2.0 to OpenGL v3.0 and above.

In this chapter, we will introduce the three shader stages accessible in the OpenGL v3.3 core profile, that is, vertex, geometry, and fragment shaders. Note that OpenGL v4.0 introduced two additional shader stages that is tessellation control and tessellation evaluation shaders between the vertex and geometry shader.

Setting up the OpenGL v3.3 core profile on Visual Studio 2010 using the GLEW and freeglut libraries

We will start with a very basic example in which we will set up the modern OpenGL v3.3 core profile. This example will simply create a blank window and clear the window with red color.

OpenGL or any other graphics API for that matter requires a window to display graphics in. This is carried out through platform specific codes. Previously, the GLUT library was invented to provide windowing functionality in a platform independent manner. However, this library was not maintained with each new OpenGL release. Fortunately, another independent project, freeglut, followed in the GLUT footsteps by providing similar (and in some cases better) windowing support in a platform independent way. In addition, it also helps with the creation of the OpenGL core/compatibility profile contexts. The latest version of freeglut may be downloaded from <http://freeglut.sourceforge.net>. The version used in the source code accompanying this book is v2.8.0. After downloading the freeglut library, you will have to compile it to generate the libs/dlls.

The extension mechanism provided by OpenGL still exists. To aid with getting the appropriate function pointers, the GLEW library is used. The latest version can be downloaded from <http://glew.sourceforge.net>. The version of GLEW used in the source code accompanying this book is v1.9.0. If the source release is downloaded, you will have to build GLEW first to generate the libs and dlls on your platform. You may also download the pre-built binaries.

Prior to OpenGL v3.0, the OpenGL API provided support for matrices by providing specific matrix stacks such as the modelview, projection, and texture matrix stacks. In addition, transformation functions such as translate, rotate, and scale, as well as projection functions were also provided. Moreover, immediate mode rendering was supported, allowing application programmers to directly push the vertex information to the hardware.

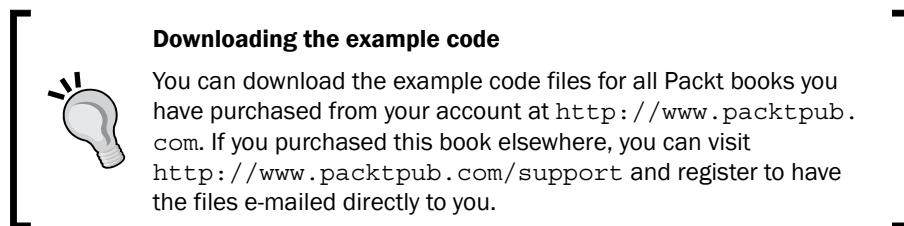
In OpenGL v3.0 and above, all of these functionalities are removed from the core profile, whereas for backward compatibility they are retained in the compatibility profile. If we use the core profile (which is the recommended approach), it is our responsibility to implement all of these functionalities including all matrix handling and transformations. Fortunately, a library called `glm` exists that provides math related classes such as vectors and matrices. It also provides additional convenience functions and classes. For all of the demos in this book, we will use the `glm` library. Since this is a headers only library, there are no linker libraries for `glm`. The latest version of `glm` can be downloaded from <http://glm.g-truc.net>. The version used for the source code in this book is v0.9.4.0.

There are several image formats available. It is not a trivial task to write an image loader for such a large number of image formats. Fortunately, there are several image loading libraries that make image loading a trivial task. In addition, they provide support for both loading as well as saving of images into various formats. One such library is the `SOIL` image loading library. The latest version of `SOIL` can be downloaded from <http://www.lonesock.net/soil.html>.

Once we have downloaded the `SOIL` library, we extract the file to a location on the hard disk. Next, we set up the include and library paths in the Visual Studio environment. The include path on my development machine is `D:\Libraries\soil\Simple OpenGL Image Library\src` whereas, the library path is set to `D:\Libraries\soil\Simple OpenGL Image Library\lib\VC10_Debug`. Of course, the path for your system will be different than mine but these are the folders that the directories should point to.

These steps will help us to set up our development environment. For all of the recipes in this book, Visual Studio 2010 Professional version is used. Readers may also use the free express edition or any other version of Visual Studio (for example, Ultimate/Enterprise). Since there are a myriad of development environments, to make it easier for users on other platforms, we have provided premake script files as well.

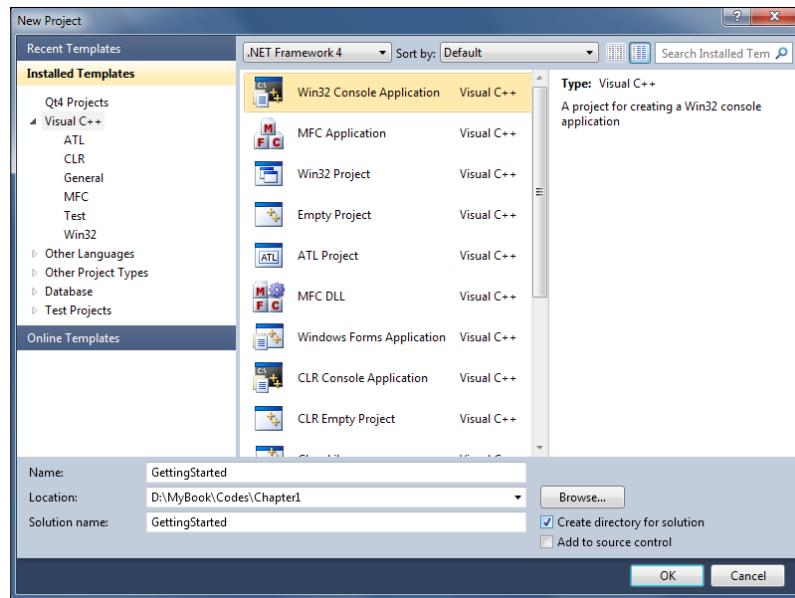
The code for this recipe is in the `Chapter1/GettingStarted` directory.



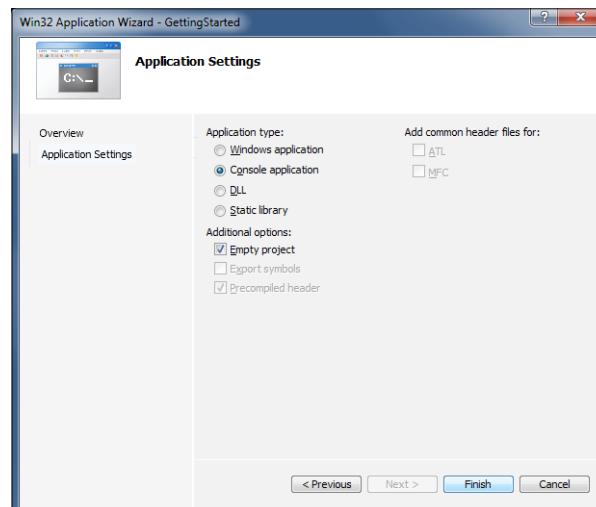
How to do it...

Let us setup the development environment using the following steps:

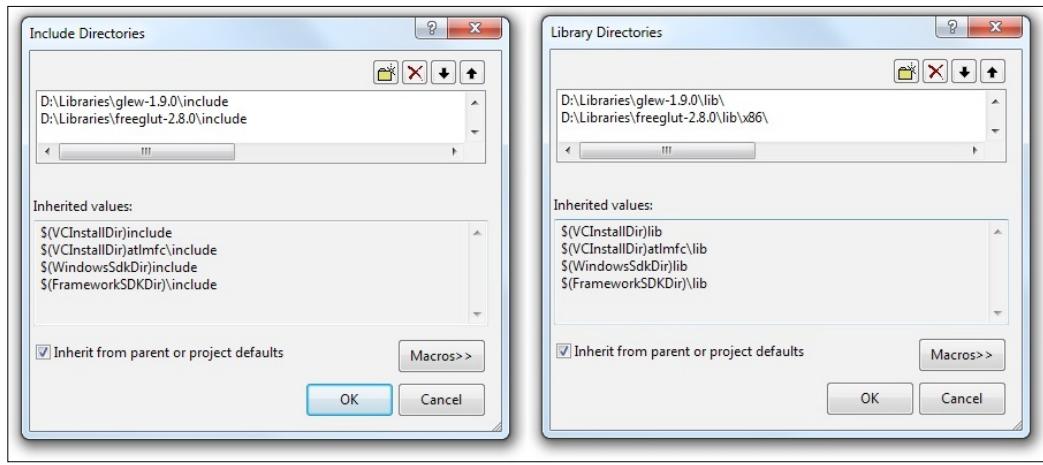
1. After downloading the required libraries, we set up the Visual Studio 2010 environment settings.



2. We first create a new **Win32 Console Application** project as shown in the preceding screenshot. We set up an empty Win32 project as shown in the following screenshot:



3. Next, we set up the include and library paths for the project by going into the **Project** menu and selecting project **Properties**. This opens a new dialog box. In the left pane, click on the **Configuration Properties** option and then on **VC++ Directories**.
4. In the right pane, in the **Include Directories** field, add the GLEW and freeglut subfolder paths.
5. Similarly, in the **Library Directories**, add the path to the lib subfolder of GLEW and freeglut libraries as shown in the following screenshot:

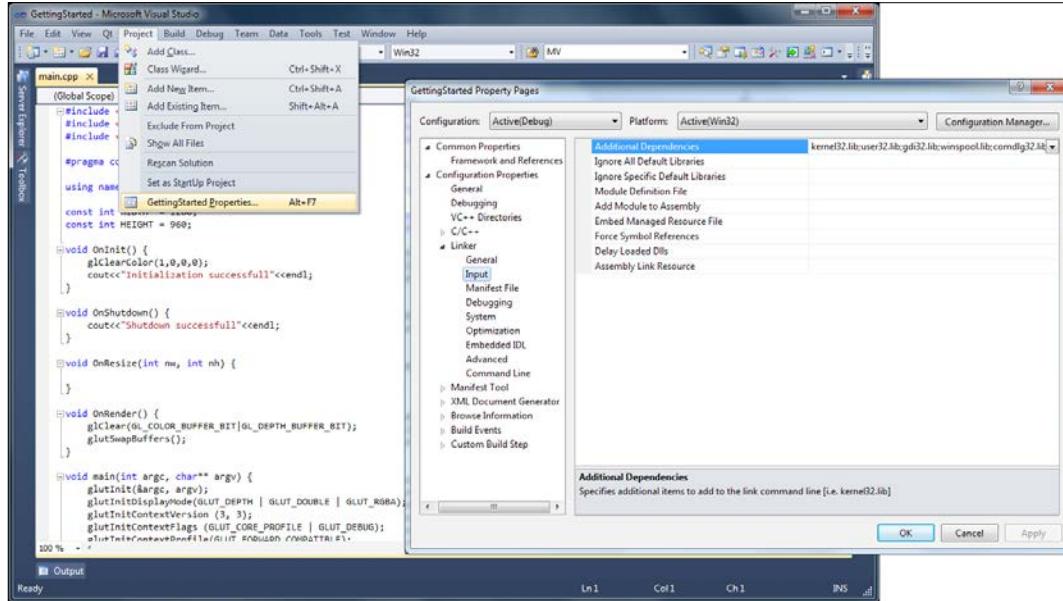


6. Next, we add a new .cpp file to the project and name it `main.cpp`. This is the main source file of our project. You may also browse through `Chapter1/GettingStarted/GettingStarted/main.cpp` which does all this setup already.
7. Let us skim through the `Chapter1/ GettingStarted/GettingStarted/main.cpp` file piece by piece.

```
#include <GL/glew.h>
#include <GL/freeglut.h>
#include <iostream>
```

These lines are the include files that we will add to all of our projects. The first is the GLEW header, the second is the freeglut header, and the final include is the standard input/output header.

8. In Visual Studio, we can add the required linker libraries in two ways. The first way is through the Visual Studio environment (by going to the **Properties** menu item in the **Project** menu). This opens the project's property pages. In the configuration properties tree, we collapse the **Linker** subtree and click on the **Input** item. The first field in the right pane is **Additional Dependencies**. We can add the linker library in this field as shown in the following screenshot:



9. The second way is to add the `glew32.lib` file to the linker settings programmatically. This can be achieved by adding the following pragma:

```
#pragma comment(lib, "glew32.lib")
```
10. The next line is the using directive to enable access to the functions in the `std` namespace. This is not mandatory but we include this here so that we do not have to prefix `std::` to any standard library function from the `iostream` header file.

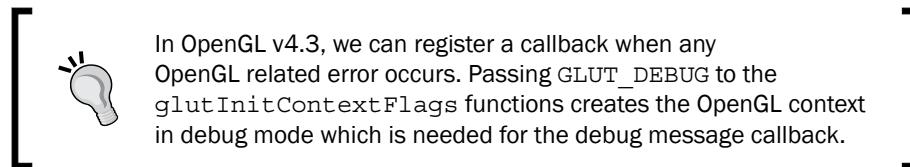
```
using namespace std;
```
11. The next lines define the width and height constants which will be the screen resolution for the window. After these declarations, there are five function definitions. The `OnInit()` function is used for initializing any OpenGL state or object, `OnShutdown()` is used to delete an OpenGL object, `OnResize()` is used to handle the resize event, `OnRender()` helps to handle the paint event, and `main()` is the entry point of the application. We start with the definition of the `main()` function.

```
const int WIDTH = 1280;
const int HEIGHT = 960;
```

```
int main(int argc, char** argv) {  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE |  
    GLUT_RGBA);  
    glutInitContextVersion (3, 3);  
    glutInitContextFlags (GLUT_CORE_PROFILE | GLUT_DEBUG);  
    glutInitContextProfile(GLUT_FORWARD_COMPATIBLE);  
    glutInitWindowSize(WIDTH, HEIGHT);
```

12. The first line `glutInit` initializes the GLUT environment. We pass the command line arguments to this function from our entry point. Next, we set up the display mode for our application. In this case, we request the GLUT framework to provide support for a depth buffer, double buffering (that is a front and a back buffer for smooth, flicker-free rendering), and the format of the frame buffer to be RGBA (that is with red, green, blue, and alpha channels). Next, we set the required OpenGL context version we desire by using the `glutInitContextVersion`. The first parameter is the major version of OpenGL and the second parameter is the minor version of OpenGL. For example, if we want to create an OpenGL v4.3 context, we will call `glutInitContextVersion (4, 3)`. Next, the context flags are specified:

```
glutInitContextFlags (GLUT_CORE_PROFILE | GLUT_DEBUG);  
glutInitContextProfile(GLUT_FORWARD_COMPATIBLE);
```



13. For any version of OpenGL including OpenGL v3.3 and above, there are two profiles available: the core profile (which is a pure shader based profile without support for OpenGL fixed functionality) and the compatibility profile (which supports the OpenGL fixed functionality). All of the matrix stack functionality `glMatrixMode (*)`, `glTranslate*`, `glRotate*`, `glScale*`, and so on, and immediate mode calls such as `glVertex*`, `glTexCoord*`, and `glNormal*` of legacy OpenGL, are retained in the compatibility profile. However, they are removed from the core profile. In our case, we will request a forward compatible core profile which means that we will not have any fixed function OpenGL functionality available.

14. Next, we set the screen size and create the window:

```
glutInitWindowSize(WIDTH, HEIGHT);  
glutCreateWindow("Getting started with OpenGL 3.3");
```

15. Next, we initialize the GLEW library. It is important to initialize the GLEW library after the OpenGL context has been created. If the function returns `GLEW_OK` the function succeeds, otherwise the GLEW initialization fails.

```
glewExperimental = GL_TRUE;
GLenum err = glewInit();
if (GLEW_OK != err){
    cerr<<"Error: "<<glewGetErrorString(err)<<endl;
} else {
    if (GLEW_VERSION_3_3)
    {
        cout<<"Driver supports OpenGL 3.3\nDetails:"<<endl;
    }
}
cout<<"\tUsing glew "<<glewGetString(GLEW_VERSION)<<endl;
cout<<"\tVendor: "<<glGetString(GL_VENDOR)<<endl;
cout<<"\tRenderer: "<<glGetString(GL_RENDERER)<<endl;
cout<<"\tVersion: "<<glGetString(GL_VERSION)<<endl;
cout<<"\tGLSL:
"<<glGetString(GL_SHADING_LANGUAGE_VERSION)<<endl;
```

The `glewExperimental` global switch allows the GLEW library to report an extension if it is supported by the hardware but is unsupported by the experimental or pre-release drivers. After the function is initialized, the GLEW diagnostic information such as the GLEW version, the graphics vendor, the OpenGL renderer, and the shader language version are printed to the standard output.

16. Finally, we call our initialization function `OnInit()` and then attach our uninitialization function `OnShutdown()` as the `glutCloseFunc` method—the close callback function which will be called when the window is about to close. Next, we attach our display and reshape function to their corresponding callbacks. The main function is terminated with a call to the `glutMainLoop()` function which starts the application's main loop.

```
OnInit();
glutCloseFunc(OnShutdown);
glutDisplayFunc(OnRender);
glutReshapeFunc(OnResize);
glutMainLoop();
return 0;
}
```

There's more...

The remaining functions are defined as follows:

```
void OnInit() {
    glClearColor(1,0,0,0);
```

```
        cout<<"Initialization successfull"<<endl;
    }
void OnShutdown() {
    cout<<"Shutdown successfull"<<endl;
}
void OnResize(int nw, int nh) {
}
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glutSwapBuffers();
}
```

For this simple example, we set the clear color to red (R:1, G:0, B:0, A:0). The first three are the red, green, and blue channels and the last is the alpha channel which is used in alpha blending. The only other function defined in this simple example is the `OnRender()` function, which is our display callback function that is called on the paint event. This function first clears the color and depth buffers to the clear color and clear depth values respectively.



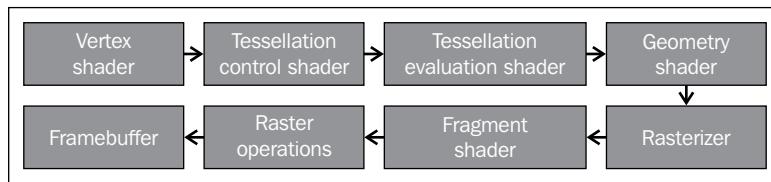
Similar to the color buffer, there is another buffer called the depth buffer. Its clear value can be set using the `glClearDepth` function. It is used for hardware based hidden surface removal. It simply stores the depth of the nearest fragment encountered so far. The incoming fragment's depth value overwrites the depth buffer value based on the depth clear function specified for the depth test using the `glDepthFunc` function. By default the depth value gets overwritten if the current fragment's depth is lower than the existing depth in the depth buffer.

The `glutSwapBuffers` function is then called to set the current back buffer as the current front buffer that is shown on screen. This call is required in a double buffered OpenGL application. Running the code gives us the output shown in the following screenshot.



Designing a GLSL shader class

We will now have a look at how to set up shaders. Shaders are special programs that are run on the GPU. There are different shaders for controlling different stages of the programmable graphics pipeline. In the modern GPU, these include the **vertex shader** (which is responsible for calculating the clip-space position of a vertex), the **tessellation control shader** (which is responsible for determining the amount of tessellation of a given patch), the **tessellation evaluation shader** (which computes the interpolated positions and other attributes on the tessellation result), the **geometry shader** (which processes primitives and can add additional primitives and vertices if needed), and the **fragment shader** (which converts a rasterized fragment into a colored pixel and a depth). The modern GPU pipeline highlighting the different shader stages is shown in the following figure.



Note that the tessellation control/evaluation shaders are only available in the hardware supporting OpenGL v4.0 and above. Since the steps involved in shader handling as well as compiling and attaching shaders for use in OpenGL applications are similar, we wrap these steps in a simple class we call `GLSLShader`.

Getting ready

The `GLSLShader` class is defined in the `GLSLShader.h/cpp` files. We first declare the constructor and destructor which initialize the member variables. The next three functions, `LoadFromString`, `LoadFromFile`, and `CreateAndLinkProgram` handle the shader compilation, linking, and program creation. The next two functions, `Use` and `UnUse` functions bind and unbind the program. Two `std::map` datastructures are used. They store the attribute's/uniform's name as the key and its location as the value. This is done to remove the redundant call to get the attribute's/uniform's location each frame or when the location is required to access the attribute/uniform. The next two functions, `AddAttribute` and `AddUniform` add the locations of the attribute and uniforms into their respective `std::map` (`_attributeList` and `_uniformLocationList`).

```
class GLSLShader
{
public:
    GLSLShader(void);
    ~GLSLShader(void);
```

```
void LoadFromString(GLenum whichShader, const string& source);
void LoadFromFile(GLenum whichShader, const string& filename);
void CreateAndLinkProgram();
void Use();
void UnUse();
void AddAttribute(const string& attribute);
void AddUniform(const string& uniform);
GLuint operator[] (const string& attribute);
GLuint operator() (const string& uniform);
void DeleteShaderProgram();

private:
    enum ShaderType{VERTEX_SHADER,FRAGMENT_SHADER,GEOMETRY_SHADER};
    GLuint _program;
    int _totalShaders;
    GLuint _shaders[3];
    map<string,GLuint> _attributeList;
    map<string,GLuint> _uniformLocationList;
};
```

To make it convenient to access the attribute and uniform locations from their maps , we declare the two indexers. For attributes, we overload the square brackets ([]) whereas for uniforms, we overload the parenthesis operation () . Finally, we define a function DeleteShaderProgram for deletion of the shader program object. Following the function declarations are the member fields.

How to do it...

In a typical shader application, the usage of the GLSLShader object is as follows:

1. Create the GLSLShader object either on stack (for example, GLSLShader shader;) or on the heap (for example, GLSLShader* shader=new GLSLShader();)
2. Call LoadFromFile on the GLSLShader object reference
3. Call CreateAndLinkProgram on the GLSLShader object reference
4. Call Use on the GLSLShader object reference to bind the shader object
5. Call AddAttribute/AddUniform to store locations of all of the shader's attributes and uniforms respectively
6. Call UnUse on the GLSLShader object reference to unbind the shader object

Note that the above steps are required at initialization only. We can set the values of the uniforms that remain constant throughout the execution of the application in the Use/UnUse block given above.

At the rendering step, we access uniform(s), if we have uniforms that change each frame (for example, the modelview matrices). We first bind the shader by calling the `GLSLShader::Use` function. We then set the uniform by calling the `glUniform{ * }` function, invoke the rendering by calling the `glDraw{ * }` function, and then unbind the shader (`GLSLShader::UnUse`). Note that the `glDraw{ * }` call passes the attributes to the GPU.

How it works...

In a typical OpenGL shader application, the shader specific functions and their sequence of execution are as follows:

```
glCreateShader  
glShaderSource  
glCompileShader  
glGetShaderInfoLog
```

Execution of the above four functions creates a shader object. After the shader object is created, a shader program object is created using the following set of functions in the following sequence:

```
glCreateProgram  
glAttachShader  
glLinkProgram  
glGetProgramInfoLog
```



Note that after the shader program has been linked, we can safely delete the shader object.

There's more...

In the `GLSLShader` class, the first four steps are handled in the `LoadFromString` function and the later four steps are handled by the `CreateAndLinkProgram` member function. After the shader program object has been created, we can set the program for execution on the GPU. This process is called **shader binding**. This is carried out by the `glUseProgram` function which is called through the `Use/UnUse` functions in the `GLSLShader` class.

To enable communication between the application and the shader, there are two different kinds of fields available in the shader. The first are the attributes which may change during shader execution across different shader stages. All per-vertex attributes fall in this category. The second are the uniforms which remain constant throughout the shader execution. Typical examples include the modelview matrix and the texture samplers.

In order to communicate with the shader program, the application must obtain the location of an attribute/uniform after the shader program is bound. The location identifies the attribute/uniform. In the `GLSLShader` class, for convenience, we store the locations of attributes and uniforms in two separate `std::map` objects.

For accessing any attribute/uniform location, we provide an indexer in the `GLSLShader` class. In cases where there is an error in the compilation or linking stage, the shader log is printed to the console. Say for example, our `GLSLShader` object is called `shader` and our `shader` contains a uniform called `MVP`. We can first add it to the map of `GLSLShader` by calling `shader.AddUniform("MVP")`. This function adds the uniform's location to the map. Then when we want to access the uniform, we directly call `shader("MVP")` and it returns the location of our uniform.

Rendering a simple colored triangle using shaders

We will now put the `GLSLShader` class to use by implementing an application to render a simple colored triangle on screen.

Getting ready

For this recipe, we assume that the reader has created a new empty Win32 project with OpenGL 3.3 core profile as shown in the first recipe. The code for this recipe is in the `Chapter1/SimpleTriangle` directory.



In all of the code samples in this book, you will see a macro `GL_CHECK_ERRORS` dispersed throughout. This macro checks the current error bit for any error which might be raised by passing invalid arguments to an OpenGL function, or when there is some problem with the OpenGL state machine. For any such error, this macro traps it and generates a debug assertion signifying that the OpenGL state machine has some error. In normal cases, no assertion should be raised, so adding this macro helps to identify errors. Since this macro calls `glGetError` inside a debug assert, it is stripped in the release build.

Now we will look at the different transformation stages through which a vertex goes, before it is finally rendered on screen. Initially, the vertex position is specified in what is called the object space. This space is the one in which the vertex location is specified for an object. We apply modeling transformation to the object space vertex position by multiplying it with an affine matrix (for example, a matrix for scaling, rotating, translating, and so on). This brings the object space vertex position into world space. Next, the world space positions are multiplied by the camera/viewing matrix which brings the position into view/eye/camera space. OpenGL stores the modeling and viewing transformations in a single (modelview) matrix.

The view space positions are then projected by using a projection transformation which brings the position into clip space. The clip space positions are then normalized to get the normalized device coordinates which have a canonical viewing volume (coordinates are [-1, -1, 1] to [1, 1, 1] in x, y, and z coordinates respectively). Finally, the viewport transformation is applied which brings the vertex into window/screen space.

How to do it...

Let us start this recipe using the following steps:

1. Define a vertex shader (`shaders/shader.vert`) to transform the object space vertex position to clip space.

```
#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vColor;
smooth out vec4 vSmoothColor;
uniform mat4 MVP;
void main()
{
    vSmoothColor = vec4(vColor,1);
    gl_Position = MVP*vec4(vVertex,1);
}
```

2. Define a fragment shader (`shaders/shader.frag`) to output a smoothly interpolated color from the vertex shader to the frame buffer.

```
#version 330 core
smooth in vec4 vSmoothColor;
layout(location=0) out vec4 vFragColor;
void main()
{
    vFragColor = vSmoothColor;
}
```

3. Load the two shaders using the `GLSLShader` class in the `OnInit()` function.

```
shader.LoadFromFile(GL_VERTEX_SHADER,
"shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
shader.AddAttribute("vVertex");
shader.AddAttribute("vColor");
shader.AddUniform("MVP");
shader.UnUse();
```

4. Create the geometry and topology. We will store the attributes together in an interleaved vertex format, that is, we will store the vertex attributes in a struct containing two attributes, position and color.

```
vertices[0].color=glm::vec3(1,0,0);
vertices[1].color=glm::vec3(0,1,0);
vertices[2].color=glm::vec3(0,0,1);

vertices[0].position=glm::vec3(-1,-1,0);
vertices[1].position=glm::vec3(0,1,0);
vertices[2].position=glm::vec3(1,-1,0);

indices[0] = 0;
indices[1] = 1;
indices[2] = 2;
```

5. Store the geometry and topology in the buffer object(s). The stride parameter controls the number of bytes to jump to reach the next element of the same attribute. For the interleaved format, it is typically the size of our vertex struct in bytes, that is, `sizeof(Vertex)`.

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
 glBindVertexArray(vaoID);
 glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
 &vertices[0], GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);
 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
 GL_FALSE, stride, 0);
 glEnableVertexAttribArray(shader["vColor"]);
 glVertexAttribPointer(shader["vColor"], 3, GL_FLOAT,
 GL_FALSE, stride, (const GLvoid*)offsetof(Vertex, color));

 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
 &indices[0], GL_STATIC_DRAW);
```

6. Set up the resize handler to set up the viewport and projection matrix.

```
void OnResize(int w, int h) {
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    P = glm::ortho(-1,1,-1,1);
}
```

7. Set up the rendering code to bind the `GLSLShader` shader, pass the uniforms, and then draw the geometry.

```
void OnRender() {  
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  
    shader.Use();  
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,  
    glm::value_ptr(P*MV));  
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT,  
    0);  
    shader.UnUse();  
    glutSwapBuffers();  
}
```

8. Delete the shader and other OpenGL objects.

```
void OnShutdown() {  
    shader.DeleteShaderProgram();  
    glDeleteBuffers(1, &vboVerticesID);  
    glDeleteBuffers(1, &vboIndicesID);  
    glDeleteVertexArrays(1, &vaoID);  
}
```

How it works...

For this simple example, we will only use a vertex shader (`shaders/shader.vert`) and a fragment shader (`shaders/shader.frag`). The first line in the shader signifies the GLSL version of the shader. Starting from OpenGL v3.0, the version specifiers correspond to the OpenGL version used. So for OpenGL v3.3, the GLSL version is 330. In addition, since we are interested in the core profile, we add another keyword following the version number to signify that we have a core profile shader.

Another important thing to note is the layout qualifier. This is used to bind a specific integral attribute index to a given per-vertex attribute. While we can give the attribute locations in any order, for all of the recipes in this book the attribute locations are specified starting from 0 for position, 1 for normals, 2 for texture coordinates, and so on. The layout location qualifier makes the `glBindAttribLocation` call redundant as the location index specified in the shader overrides any `glBindAttribLocation` call.

The vertex shader simply outputs the input per-vertex color to the output (`vSmoothColor`). Such attributes that are interpolated across shader stages are called **varying attributes**. It also calculates the clip space position by multiplying the per-vertex position (`vVertex`) with the combined modelview projection (MVP) matrix.

```
vSmoothColor = vec4(vColor,1);  
gl_Position = MVP*vec4(vVertex,1);
```



By prefixing `smooth` to the output attribute, we tell the GLSL shader to do smooth perspective-correct interpolation for the attribute to the next stage of the pipeline. The other qualifiers usable are `flat` and `noperspective`. When no qualifier is specified the default interpolation qualifier is `smooth`.

The fragment shader writes the input color (`vSmoothColor`) to the frame buffer output (`vFragColor`).

```
vFragColor = vSmoothColor;
```

There's more...

In the simple triangle demo application code, we store the `GLSLShader` object reference in the global scope so that we can access it in any function we desire. We modify the `OnInit()` function by adding the following lines:

```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
shader.AddAttribute("vVertex");
shader.AddAttribute("vColor");
shader.AddUniform("MVP");
shader.UnUse();
```

The first two lines create the GLSL shader of the given type by reading the contents of the file with the given filename. In all of the recipes in this book, the vertex shader files are stored with a `.vert` extension, the geometry shader files with a `.geom` extension, and the fragment shader files with a `.frag` extension. Next, the `GLSLShader::CreateAndLinkProgram` function is called to create the shader program from the shader object. Next, the program is bound and then the locations of attributes and uniforms are stored.

We pass two attributes per-vertex, that is vertex position and vertex color. In order to facilitate the data transfer to the GPU, we create a simple `Vertex` struct as follows:

```
struct Vertex {
    glm::vec3 position;
    glm::vec3 color;
};

Vertex vertices[3];
GLushort indices[3];
```

Next, we create an array of three vertices in the global scope. In addition, we store the triangle's vertex indices in the indices global array. Later we initialize these two arrays in the `OnInit()` function. The first vertex is assigned the red color, the second vertex is assigned the green color, and the third vertex is assigned the blue color.

```
vertices[0].color=glm::vec3(1,0,0);
vertices[1].color=glm::vec3(0,1,0);
vertices[2].color=glm::vec3(0,0,1);

vertices[0].position=glm::vec3(-1,-1,0);
vertices[1].position=glm::vec3(0,1,0);
vertices[2].position=glm::vec3(1,-1,0);

indices[0] = 0;
indices[1] = 1;
indices[2] = 2;
```

Next, the vertex positions are given. The first vertex is assigned an object space position of (-1,-1,0), the second vertex is assigned (0,1,0), and the third vertex is assigned (1,-1,0). For this simple demo, we use an orthographic projection for a view volume of (-1,1,-1,1). Finally, the three indices are given in a linear order.

In OpenGL v3.3 and above, we typically store the geometry information in buffer objects, which is a linear array of memory managed by the GPU. In order to facilitate the handling of buffer object(s) during rendering, we use a **vertex array object (VAO)**. This object stores references to buffer objects that are bound after the VAO is bound. The advantage we get from using a VAO is that after the VAO is bound, we do not have to bind the buffer object(s).

In this demo, we declare three variables in global scope; `vaoID` for VAO handling, and `vboVerticesID` and `vboIndicesID` for buffer object handling. The VAO object is created by calling the `glGenVertexArrays` function. The buffer objects are generated using the `glGenBuffers` function. The first parameter for both of these functions is the total number of objects required, and the second parameter is the reference to where the object handle is stored. These functions are called in the `OnInit()` function.

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
 glBindVertexArray(vaoID);
```

After the VAO object is generated, we bind it to the current OpenGL context so that all successive calls affect the attached VAO object. After the VAO binding, we bind the buffer object storing vertices (`vboVerticesID`) using the `glBindBuffer` function to the `GL_ARRAY_BUFFER` binding. Next, we pass the data to the buffer object by using the `glBufferData` function. This function also needs the binding point, which is again `GL_ARRAY_BUFFER`. The second parameter is the size of the vertex array we will push to the GPU memory. The third parameter is the pointer to the start of the CPU memory. We pass the address of the vertices global array. The last parameter is the usage hint which tells the GPU that we are not going to modify the data often.

```
glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);
glBufferData (GL_ARRAY_BUFFER, sizeof(vertices), &vertices[0],
GL_STATIC_DRAW);
```

The usage hints have two parts; the first part tells how frequently the data in the buffer object is modified. These can be `STATIC` (modified once only), `DYNAMIC` (modified occasionally), or `STREAM` (modified at every use). The second part is the way this data will be used. The possible values are `DRAW` (the data will be written but not read), `READ` (the data will be read only), and `COPY` (the data will be neither read nor written). Based on the two hints a qualifier is generated. For example, `GL_STATIC_DRAW` if the data will never be modified and `GL_DYNAMIC_DRAW` if the data will be modified occasionally. These hints allow the GPU and the driver to optimize the read/write access to this memory.

In the next few calls, we enable the vertex attributes. This function needs the location of the attribute, which we obtain by the `GLSLShader::operator[]`, passing it the name of the attribute whose location we require. We then call `glVertexAttribPointer` to tell the GPU how many elements there are and what is their type, whether the attribute is normalized, the stride (which means the total number of bytes to skip to reach the next element; for our case since the attributes are stored in a `Vertex` struct, the next element's stride is the size of our `Vertex` struct), and finally, the pointer to the attribute in the given array. The last parameter requires explanation in case we have interleaved attributes (as we have). The `offsetof` operator returns the offset in bytes, to the attribute in the given struct. Hence, the GPU knows how many bytes it needs to skip in order to access the next attribute of the given type. For the `vVertex` attribute, the last parameter is `0` since the next element is accessed immediately after the stride. For the second attribute `vColor`, it needs to hop 12 bytes before the next `vColor` attribute is obtained from the given vertices array.

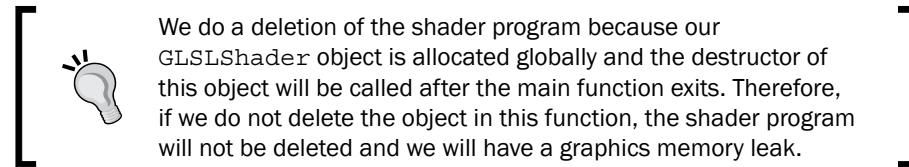
```
glEnableVertexAttribArray(shader["vVertex"]);
glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
GL_FALSE,stride,0);
glEnableVertexAttribArray(shader["vColor"]);
glVertexAttribPointer(shader["vColor"], 3, GL_FLOAT,
GL_FALSE,stride, (const GLvoid*)offsetof(Vertex, color));
```

The indices are pushed similarly using `glBindBuffer` and `glBufferData` but to a different binding point, that is, `GL_ELEMENT_ARRAY_BUFFER`. Apart from this change, the rest of the parameters are exactly the same as for the vertices data. The only difference being the buffer object, which for this case is `vboIndicesID`. In addition, the passed array to the `glBufferData` function is the indices array.

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
&indices[0], GL_STATIC_DRAW);
```

To complement the object generation in the `OnInit()` function, we must provide the object deletion code. This is handled in the `OnShutdown()` function. We first delete the shader program by calling the `GLSLShader::DeleteShaderProgram` function. Next, we delete the two buffer objects (`vboVerticesID` and `vboIndicesID`) and finally we delete the vertex array object (`vaoID`).

```
void OnShutdown() {
    shader.DeleteShaderProgram();
    glDeleteBuffers(1, &vboVerticesID);
    glDeleteBuffers(1, &vboIndicesID);
    glDeleteVertexArrays(1, &vaoID);
}
```

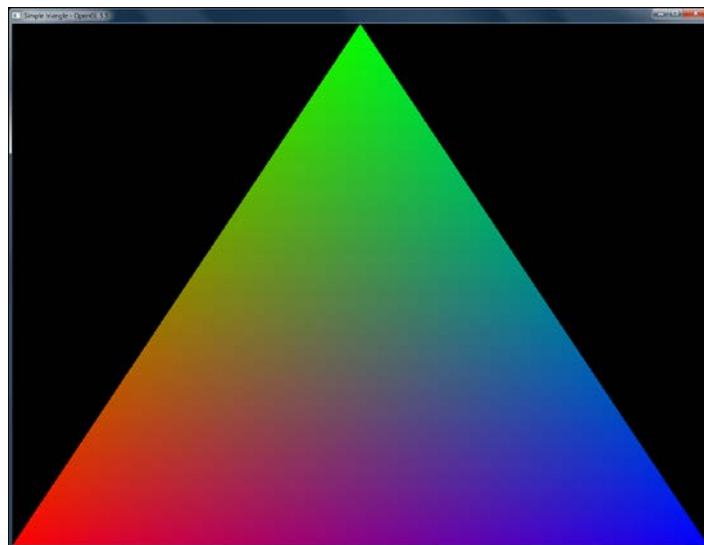


The rendering code of the simple triangle demo is as follows:

```
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    shader.Use();
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(P*MV));
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, 0);
    shader.UnUse();
    glutSwapBuffers();
}
```

The rendering code first clears the color and depth buffer and binds the shader program by calling the `GLSLShader::Use()` function. It then passes the combined modelview and projection matrix to the GPU by invoking the `glUniformMatrix4fv` function. The first parameter is the location of the uniform which we obtain from the `GLSLShader::operator()` function, by passing it the name of the uniform whose location we need. The second parameter is the total number of matrices we wish to pass. The third parameter is a Boolean signifying if the matrix needs to be transposed, and the final parameter is the float pointer to the matrix object. Here we use the `glm::value_ptr` function to get the float pointer from the matrix object. Note that the OpenGL matrices are concatenated right to left since it follows a right handed coordinate system in a column major layout. Hence we keep the projection matrix on the left and the modelview matrix on the right. For this simple example, the modelview matrix (MV) is set as the identity matrix.

After this function, the `glDrawElements` call is made. Since we have left our VAO object (`vaoID`) bound, we pass `0` to the final parameter of this function. This tells the GPU to use the references of the `GL_ELEMENT_ARRAY_BUFFER` and `GL_ARRAY_BUFFER` binding points of the bound VAO. Thus we do not need to explicitly bind the `vboVerticesID` and `vboIndicesID` buffer objects again. After this call, we unbind the shader program by calling the `GLSLShader::UnUse()` function. Finally, we call the `glutSwapBuffer` function to show the back buffer on screen. After compiling and running, we get the output as shown in the following figure:



See also

Learn modern 3D graphics programming by Jason L. McKesson at <http://www.arcsynthesis.org/gltut/Basics/Basics.html>.

Doing a ripple mesh deformer using the vertex shader

In this recipe, we will deform a planar mesh using the vertex shader. We know that the vertex shader is responsible for outputting the clip space position of the given object space vertex. In between this conversion, we can apply the modeling transformation to transform the given object space vertex to **world space position**.

Getting ready

For this recipe, we assume that the reader knows how to set up a simple triangle on screen using a vertex and fragment shader as detailed in the previous recipe. The code for this recipe is in the `Chapter1\RippleDeformer` directory.

How to do it...

We can implement a ripple shader using the following steps:

1. Define the vertex shader that deforms the object space vertex position.

```
#version 330 core
layout(location=0) in vec3 vVertex;
uniform mat4 MVP;
uniform float time;
const float amplitude = 0.125;
const float frequency = 4;
const float PI = 3.14159;
void main()
{
    float distance = length(vVertex);
    float y = amplitude*sin(-PI*distance*frequency+time);
    gl_Position = MVP*vec4(vVertex.x, y, vVertex.z, 1);
}
```

2. Define a fragment shader that simply outputs a constant color.

```
#version 330 core
layout(location=0) out vec4 vFragColor;
void main()
{
    vFragColor = vec4(1,1,1,1);
}
```

3. Load the two shaders using the `GLSLShader` class in the `OnInit()` function.

```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("MVP");
    shader.AddUniform("time");
shader.UnUse();
```

4. Create the geometry and topology.

```
int count = 0;
int i=0, j=0;
for( j=0;j<=NUM_Z;j++) {
    for( i=0;i<=NUM_X;i++) {
        vertices [count++] = glm::vec3(
            ((float(i)/(NUM_X-1)) *2-1)* HALF_SIZE_X, 0,
            ((float(j)/(NUM_Z-1))*2-1)*HALF_SIZE_Z);
    }
}
GLushort* id=&indices[0];
for (i = 0; i < NUM_Z; i++) {
    for (j = 0; j < NUM_X; j++) {
        int i0 = i * (NUM_X+1) + j;
        int i1 = i0 + 1;
        int i2 = i0 + (NUM_X+1);
        int i3 = i2 + 1;
        if ((j+i)%2) {
            *id++ = i0; *id++ = i2; *id++ = i1;
            *id++ = i1; *id++ = i2; *id++ = i3;
        } else {
            *id++ = i0; *id++ = i2; *id++ = i3;
            *id++ = i0; *id++ = i3; *id++ = i1;
        }
    }
}
```

5. Store the geometry and topology in the buffer object(s).

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
glBindVertexArray(vaoID);
glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),  
&vertices[0], GL_STATIC_DRAW);  
glEnableVertexAttribArray(shader["vVertex"]);  
glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,  
GL_FALSE, 0, 0);  
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);  
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),  
&indices[0], GL_STATIC_DRAW);
```

6. Set up the perspective projection matrix in the resize handler.

```
P = glm::perspective(45.0f, (GLfloat)w/h, 1.f, 1000.f);
```

7. Set up the rendering code to bind the `GLSLShader` shader, pass the uniforms and then draw the geometry.

```
void OnRender() {  
    time = glutGet(GLUT_ELAPSED_TIME)/1000.0f * SPEED;  
    glm::mat4 T=glm::translate(glm::mat4(1.0f),  
    glm::vec3(0.0f, 0.0f, dist));  
    glm::mat4 Rx= glm::rotate(T, rX, glm::vec3(1.0f, 0.0f,  
    0.0f));  
    glm::mat4 MV= glm::rotate(Rx, rY, glm::vec3(0.0f, 1.0f,  
    0.0f));  
    glm::mat4 MVP= P*MV;  
    shader.Use();  
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,  
    glm::value_ptr(MVP));  
    glUniform1f(shader("time"), time);  
    glDrawElements(GL_TRIANGLES, TOTAL_INDICES,  
    GL_UNSIGNED_SHORT, 0);  
    shader.UnUse();  
    glutSwapBuffers();  
}
```

8. Delete the shader and other OpenGL objects.

```
void OnShutdown() {  
    shader.DeleteShaderProgram();  
    glDeleteBuffers(1, &vboVerticesID);  
    glDeleteBuffers(1, &vboIndicesID);  
    glDeleteVertexArrays(1, &vaoID);  
}
```

How it works...

In this recipe, the only attribute passed in is the per-vertex position (`vVertex`). There are two uniforms: the combined modelview projection matrix (`MVP`) and the current time (`time`). We will use the `time` uniform to allow progression of the deformer so we can observe the ripple movement. After these declarations are three constants, namely `amplitude` (which controls how much the ripple moves up and down from the zero base line), `frequency` (which controls the total number of waves), and `PI` (a constant used in the wave formula). Note that we could have replaced the constants with uniforms and had them modified from the application code.

Now the real work is carried out in the main function. We first find the distance of the given vertex from the origin. Here we use the `length` built-in GLSL function. We then create a simple sinusoid. We know that a general sine wave can be given using the following function:

$$y = A \cdot \sin(2\pi f t + \phi)$$

Here, `A` is the wave amplitude, `f` is the frequency, `t` is the time, and `\phi` is the phase. In order to get our ripple to start from the origin, we modify the function to the following:

$$\boxed{d(x,y,z) = \sqrt{x^2 + y^2 + z^2}} \\ F(x,y,z) = A \cdot \sin(-\pi f d(x,y,z) + \phi)}$$

In our formula, we first find the distance (`d`) of the vertex from the origin by using the Euclidean distance formula. This is given to us by the `length` built-in GLSL function. Next, we input the distance into the `sin` function multiplying the distance by the frequency (`f`) and (π). In our vertex shader, we replace the phase (`\phi`) with time.

```
#version 330 core
layout(location=0) in vec3 vVertex;
uniform mat4 MVP;
uniform float time;
const float amplitude = 0.125;
const float frequency = 4;
const float PI = 3.14159;
void main()
{
    float distance = length(vVertex);
    float y = amplitude * sin(-PI * distance * frequency + time);
    gl_Position = MVP * vec4(vVertex.x, y, vVertex.z, 1);
}
```

After calculating the new y value, we multiply the new vertex position with the combined modelview projection matrix (MVP). The fragment shader simply outputs a constant color (in this case white color, `vec4(1,1,1,1)`).

```
#version 330 core
layout(location=0) out vec4 vFragColor;
void main()
{
    vFragColor = vec4(1,1,1,1);
}
```

There's more

Similar to the previous recipe, we declare the `GLSLShader` object in the global scope to allow maximum visibility. Next, we initialize the `GLSLShader` object in the `OnInit()` function.

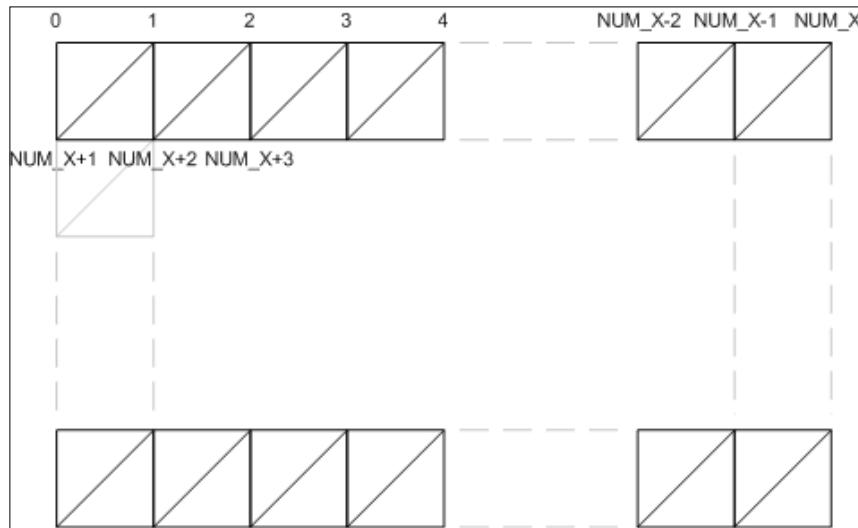
```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("MVP");
    shader.AddUniform("time");
shader.UnUse();
```

The only difference in this recipe is the addition of an additional uniform (`time`).

We generate a simple 3D planar grid in the XZ plane. The geometry is stored in the vertices global array. The total number of vertices on the X axis is stored in a global constant `NUM_X`, whereas the total number of vertices on the Z axis is stored in another global constant `NUM_Z`. The size of the planar grid in world space is stored in two global constants, `SIZE_X` and `SIZE_Z`, and half of these values are stored in the `HALF_SIZE_X` and `HALF_SIZE_Z` global constants. Using these constants, we can change the mesh resolution and world space size.

The loop simply iterates $(\text{NUM_X}+1) * (\text{NUM_Z}+1)$ times and remaps the current vertex index first into the 0 to 1 range and then into the -1 to 1 range, and finally multiplies it by the `HALF_SIZE_X` and `HALF_SIZE_Z` constants to get the range from `-HALF_SIZE_X` to `HALF_SIZE_X` and `-HALF_SIZE_Z` to `HALF_SIZE_Z`.

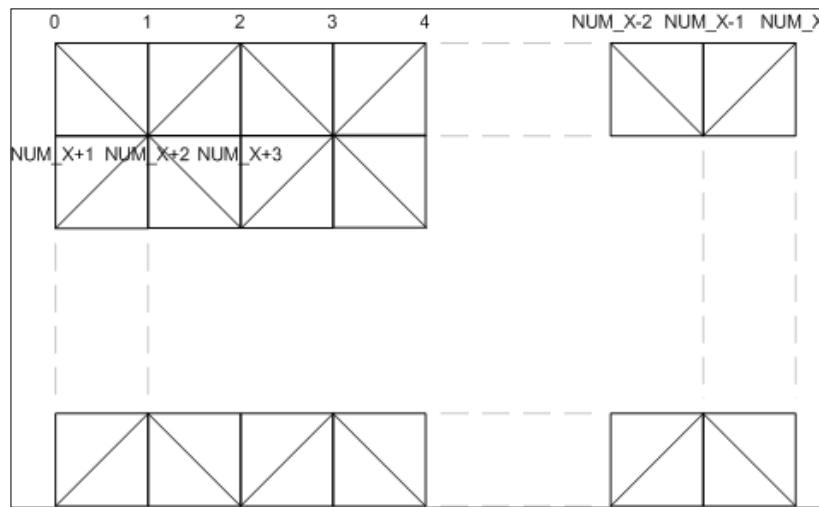
The topology of the mesh is stored in the indices global array. While there are several ways to generate the mesh topology, we will look at two common ways. The first method keeps the same triangulation for all of the mesh quads as shown in the following screenshot:



This sort of topology can be generated using the following code:

```
GLushort* id=&indices[0];
for (i = 0; i < NUM_Z; i++) {
    for (j = 0; j < NUM_X; j++) {
        int i0 = i * (NUM_X+1) + j;
        int i1 = i0 + 1;
        int i2 = i0 + (NUM_X+1);
        int i3 = i2 + 1;
        *id++ = i0; *id++ = i2; *id++ = i1;
        *id++ = i1; *id++ = i2; *id++ = i3;
    }
}
```

The second method alternates the triangulation at even and odd iterations resulting in a better looking mesh as shown in the following screenshot:



In order to alternate the triangle directions and maintain their winding order, we take two different combinations, one for an even iteration and second for an odd iteration. This can be achieved using the following code:

```

GLushort* id=&indices[0];
for (i = 0; i < NUM_Z; i++) {
    for (j = 0; j < NUM_X; j++) {
        int i0 = i * (NUM_X+1) + j;
        int i1 = i0 + 1;
        int i2 = i0 + (NUM_X+1);
        int i3 = i2 + 1;
        if ((j+i)%2) {
            *id++ = i0; *id++ = i2; *id++ = i1;
            *id++ = i1; *id++ = i2; *id++ = i3;
        } else {
            *id++ = i0; *id++ = i2; *id++ = i3;
            *id++ = i0; *id++ = i3; *id++ = i1;
        }
    }
}

```

After filling the vertices and indices arrays, we push this data to the GPU memory. We first create a vertex array object (`vaoID`) and two buffer objects, the `GL_ARRAY_BUFFER` binding for vertices and the `GL_ELEMENT_ARRAY_BUFFER` binding for the indices array. These calls are exactly the same as in the previous recipe. The only difference is that now we only have a single per-vertex attribute, that is, the vertex position (`vVertex`). The `OnShutdown()` function is also unchanged as in the previous recipe.

The rendering code is slightly changed. We first get the current elapsed time from freeglut so that we can move the ripple deformer in time. Next, we clear the color and depth buffers. After this, we set up the modelview matrix. This is carried out by using the matrix transformation functions provided by the `glm` library.

```
glm::mat4 T=glm::translate(glm::mat4(1.0f),
glm::vec3(0.0f, 0.0f, dist));
glm::mat4 Rx= glm::rotate(T, rX, glm::vec3(1.0f, 0.0f, 0.0f));
glm::mat4 MV= glm::rotate(Rx, rY, glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 MVP= P*MV;
```

Note that the matrix multiplication in `glm` follows from right to left. So the order in which we generate the transformations will be applied in the reverse order. In our case the combined modelview matrix will be calculated as `MV = (T * (Rx * Ry))`. The translation amount, `dist`, and the rotation values, `rX` and `rY`, are calculated in the mouse input functions based on the user's input.

After calculating the modelview matrix, the combined modelview projection matrix (`MVP`) is calculated. The projection matrix (`P`) is calculated in the `OnResize()` handler. In this case, the perspective projection matrix is used with four parameters, the vertical fov, the aspect ratio, and the near and far clip plane distances. The `GLSLShader` object is bound and then the two uniforms, `MVP` and `time` are passed to the shader program. The attributes are then transferred using the `glDrawElements` call as we saw in the previous recipe. The `GLSLShader` object is then unbound and finally, the back buffer is swapped.

In the ripple deformer main function, we attach two new callbacks; `glutMouseFunc` handled by the `OnMouseDown` function and `glutMotionFunc` handled by the `OnMouseMove` function. These functions are defined as follows:

```
void OnMouseDown(int button, int s, int x, int y) {
    if (s == GLUT_DOWN) {
        oldX = x;
        oldY = y;
    }
    if(button == GLUT_MIDDLE_BUTTON)
        state = 0;
    else
        state = 1;
}
```

This function is called whenever the mouse is clicked in our application window. The first parameter is for the button which was pressed (`GLUT_LEFT_BUTTON` for the left mouse button, `GLUT_MIDDLE_BUTTON` for the middle mouse button, and `GLUT_RIGHT_BUTTON` for the right mouse button). The second parameter is the state which can be either `GLUT_DOWN` or `GLUT_UP`. The last two parameters are the `x` and `y` screen location of the mouse click. In this simple example, we store the mouse click location and then set a state variable when the middle mouse button is pressed.

The `OnMouseMove` function is defined as follows:

```
void OnMouseMove(int x, int y) {
    if (state == 0)
        dist *= (1 + (y - oldY)/60.0f);
    else {
        rY += (x - oldX)/5.0f;
        rX += (y - oldY)/5.0f;
    }
    oldX = x; oldY = y;
    glutPostRedisplay();
}
```

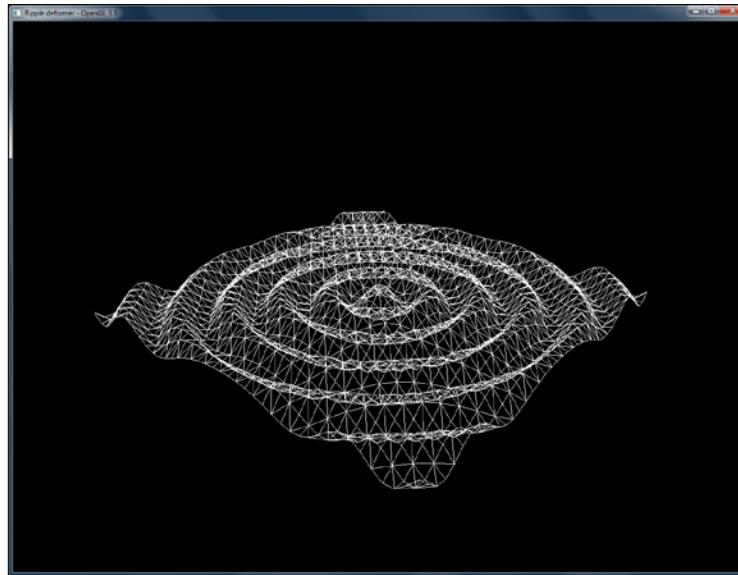
The `OnMouseMove` function has only two parameters, the `x` and `y` screen location where the mouse currently is. The mouse move event is raised whenever the mouse enters and moves in the application window. Based on the state set in the `OnMouseDown` function, we calculate the zoom amount (`dist`) if the middle mouse button is pressed. Otherwise, we calculate the two rotation amounts (`rX` and `rY`). Next, we update the `oldX` and `oldY` positions for the next event. Finally we request the freenglut framework to repaint our application window by calling `glutPostRedisplay()` function. This call sends the repaint event which re-renders our scene.

In order to make it easy for us to see the deformation, we enable wireframe rendering by calling the `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)` function in the `OnInit()` function.



There are two things to be careful about with the `glPolygonMode` function. Firstly, the first parameter can only be `GL_FRONT_AND_BACK` in the core profile. Secondly, make sure that the second parameter is named `GL_LINE` instead of `GL_LINES` which is used with the `glDraw*` functions. To disable the wireframe rendering and return to the default fill rendering, change the second parameter from `GL_LINE` to `GL_FILL`.

Running the demo code shows a ripple deformer propagating the deformation in a mesh grid as shown in the following screenshot. Hopefully, this recipe should have cleared how to use vertex shaders, especially for doing per-vertex transformations.



Dynamically subdividing a plane using the geometry shader

After the vertex shader, the next programmable stage in the OpenGL v3.3 graphics pipeline is the geometry shader. This shader contains inputs from the vertex shader stage. We can either feed these unmodified to the next shader stage or we can add/omit/modify vertices and primitives as desired. One thing that the vertex shaders lack is the availability of the other vertices of the primitive. Geometry shaders have information of all on the vertices of a single primitive.

The advantage with geometry shaders is that we can add/remove primitives on the fly. Moreover it is easier to get all vertices of a single primitive, unlike in the vertex shader, which has information on a single vertex only. The main drawback of geometry shaders is the limit on the number of new vertices we can generate, which is dependent on the hardware. Another disadvantage is the limited availability of the surrounding primitives.

In this recipe, we will dynamically subdivide a planar mesh using the geometry shader.

Getting ready

This recipe assumes that the reader knows how to render a simple triangle using vertex and fragment shaders using the OpenGL v3.3 core profile. We render four planar meshes in this recipe which are placed next to each other to create a bigger planar mesh. Each of these meshes is subdivided using the same geometry shader. The code for this recipe is located in the Chapter1\SubdivisionGeometryShader directory.

How to do it...

We can implement the geometry shader using the following steps:

1. Define a vertex shader (`shaders/shader.vert`) which outputs object space vertex positions directly.

```
#version 330 core
layout(location=0) in vec3 vVertex;
void main() {
    gl_Position = vec4(vVertex, 1);
}
```

2. Define a geometry shader (`shaders/shader.geom`) which performs the subdivision of the quad. The shader is explained in the next section.

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=256) out;
uniform int sub_divisions;
uniform mat4 MVP;
void main() {
    vec4 v0 = gl_in[0].gl_Position;
    vec4 v1 = gl_in[1].gl_Position;
    vec4 v2 = gl_in[2].gl_Position;
    float dx = abs(v0.x-v2.x)/sub_divisions;
    float dz = abs(v0.z-v1.z)/sub_divisions;
    float x=v0.x;
    float z=v0.z;
    for(int j=0;j<sub_divisions*sub_divisions;j++) {
        gl_Position = MVP * vec4(x,0,z,1);
        EmitVertex();
        gl_Position = MVP * vec4(x,0,z+dz,1);
        EmitVertex();
        gl_Position = MVP * vec4(x+dx,0,z,1);
        EmitVertex();
        gl_Position = MVP * vec4(x+dx,0,z+dz,1);
```

```
    EmitVertex();
    EndPrimitive();
    x+=dx;
    if((j+1) %sub_divisions == 0) {
        x=v0.x;
        z+=dz;
    }
}
}
```

3. Define a fragment shader (`shaders/shader.frag`) that simply outputs a constant color.

```
#version 330 core
layout(location=0) out vec4 vFragColor;
void main() {
    vFragColor = vec4(1,1,1,1);
}
```

4. Load the shaders using the `GLSLShader` class in the `OnInit()` function.

```
shader.LoadFromFile(GL_VERTEX_SHADER,
"shaders/shader.vert");
shader.LoadFromFile(GL_GEOMETRY_SHADER, "shaders/shader.geom");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("MVP");
    shader.AddUniform("sub_divisions");
    glUniform1i(shader("sub_divisions"), sub_divisions);
shader.UnUse();
```

5. Create the geometry and topology.

```
vertices[0] = glm::vec3(-5,0,-5);
vertices[1] = glm::vec3(-5,0,5);
vertices[2] = glm::vec3(5,0,5);
vertices[3] = glm::vec3(5,0,-5);
GLushort* id=&indices[0];

*id++ = 0;
*id++ = 1;
*id++ = 2;
*id++ = 0;
*id++ = 2;
*id++ = 3;
```

6. Store the geometry and topology in the buffer object(s). Also enable the line display mode.

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
 glBindVertexArray(vaoID);
 glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
 &vertices[0], GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);
 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
 GL_FALSE, 0, 0);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
 &indices[0], GL_STATIC_DRAW);
 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

7. Set up the rendering code to bind the GLSLShader shader, pass the uniforms and then draw the geometry.

```
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glm::mat4 T = glm::translate( glm::mat4(1.0f),
    glm::vec3(0.0f,0.0f, dist));
    glm::mat4 Rx=glm::rotate(T,rX,glm::vec3(1.0f, 0.0f,
    0.0f));
    glm::mat4 MV=glm::rotate(Rx,rY,
    glm::vec3(0.0f,1.0f,0.0f));
    MV=glm::translate(MV, glm::vec3(-5,0,-5));
    shader.Use();
    glUniform1i(shader("sub_divisions"), sub_divisions);
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(P*MV));
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT,
    0);

    MV=glm::translate(MV, glm::vec3(10,0,0));
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(P*MV));
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT,
    0);

    MV=glm::translate(MV, glm::vec3(0,0,10));
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(P*MV));
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT,
    0);
```

```
MV=glm::translate(MV, glm::vec3(-10,0,0));
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
glm::value_ptr(P*MV));
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT,
0);
shader.UnUse();
glutSwapBuffers();
}
```

8. Delete the shader and other OpenGL objects.

```
void OnShutdown() {
    shader.DeleteShaderProgram();
    glDeleteBuffers(1, &vboVerticesID);
    glDeleteBuffers(1, &vboIndicesID);
    glDeleteVertexArrays(1, &vaoID);
    cout<<"Shutdown successfull"<<endl;
}
```

How it works...

Let's dissect the geometry shader.

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=256) out;
```

The first line signifies the GLSL version of the shader. The next two lines are important as they tell the shader processor about the input and output primitives of our geometry shader. In this case, the input will be `triangles` and the output will be a `triangle_strip`.

In addition, we also need to give the maximum number of output vertices from this geometry shader. This is a hardware specific number. For the hardware used in this development, the `max_vertices` value is found to be 256. This information can be obtained by querying the `GL_MAX_GEOMETRY_OUTPUT_VERTICES` field and it is dependent on the primitive type used and the number of attributes stored per-vertex.

```
uniform int sub_divisions;
uniform mat4 MVP;
```

Next, we declare two uniforms, the total number of subdivisions desired (`sub_divisions`) and the combined modelview projection matrix (`MVP`).

```
void main() {
    vec4 v0 = gl_in[0].gl_Position;
    vec4 v1 = gl_in[1].gl_Position;
    vec4 v2 = gl_in[2].gl_Position;
```

The bulk of the work takes place in the main entry point function. For each triangle pushed from the application, the geometry shader is run once. Thus, for each triangle, the positions of its vertices are obtained from the `gl_Position` attribute which is stored in the built-in `gl_in` array. All other attributes are input as an array in the geometry shader. We store the input positions in local variable `v0`, `v1`, and `v2`.

Next, we calculate the size of the smallest quad for the given subdivision based on the size of the given base triangle and the total number of subdivisions required.

```
float dx = abs(v0.x-v2.x)/sub_divisions;
float dz = abs(v0.z-v1.z)/sub_divisions;
float x=v0.x;
float z=v0.z;
for(int j=0;j<sub_divisions*sub_divisions;j++) {
    gl_Position = MVP * vec4(x, 0, z,1); EmitVertex();
    gl_Position = MVP * vec4(x, 0,z+dz,1); EmitVertex();
    gl_Position = MVP * vec4(x+dx,0, z,1); EmitVertex();
    gl_Position = MVP * vec4(x+dx,0,z+dz,1); EmitVertex();
    EndPrimitive();
    x+=dx;
    if((j+1) % sub_divisions == 0) {
        x=v0.x;
        z+=dz;
    }
}
}
```

We start from the first vertex. We store the `x` and `z` values of this vertex in local variables. Next, we iterate $N \times N$ times, where N is the total number of subdivisions required. For example, if we need to subdivide the mesh three times on both axes, the loop will run nine times, which is the total number of quads. After calculating the positions of the four vertices, they are emitted by calling `EmitVertex()`. This function emits the current values of output variables to the current output primitive on the primitive stream. Next, the `EndPrimitive()` call is issued to signify that we have emitted the four vertices of `triangle_strip`.

After these calculations, the local variable `x` is incremented by `dx` amount. If we are at an iteration that is a multiple of `sub_divisions`, we reset variable `x` to the `x` value of the first vertex while incrementing the local variable `z`.

The fragment shader outputs a constant color (white: `vec4(1,1,1,1)`).

There's more...

The application code is similar to the last recipes. We have an additional shader (`shaders/shader.geom`), which is our geometry shader that is loaded from file.

```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/shader.vert");
shader.LoadFromFile(GL_GEOMETRY_SHADER, "shaders/shader.geom");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();

    shader.AddAttribute("vVertex");
    shader.AddUniform("MVP");
    shader.AddUniform("sub_divisions");
    glUniform1i(shader("sub_divisions"), sub_divisions);
shader.UnUse();
```

The notable additions are highlighted, which include the new geometry shader and an additional uniform for the total subdivisions desired (`sub_divisions`). We initialize this uniform at initialization. The buffer object handling is similar to the simple triangle recipe. The other difference is in the rendering function where there are some additional modeling transformations (translations) after the viewing transformation.

The `OnRender()` function starts by clearing the color and depth buffers. It then calculates the viewing transformation as in the previous recipe.

```
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glm::mat4 T = glm::translate( glm::mat4(1.0f),
        glm::vec3(0.0f,0.0f, dist));
    glm::mat4 Rx=glm::rotate(T,rX,glm::vec3(1.0f, 0.0f, 0.0f));
    glm::mat4 MV=glm::rotate(Rx,rY, glm::vec3(0.0f,1.0f,0.0f));
    MV=glm::translate(MV, glm::vec3(-5,0,-5));
```

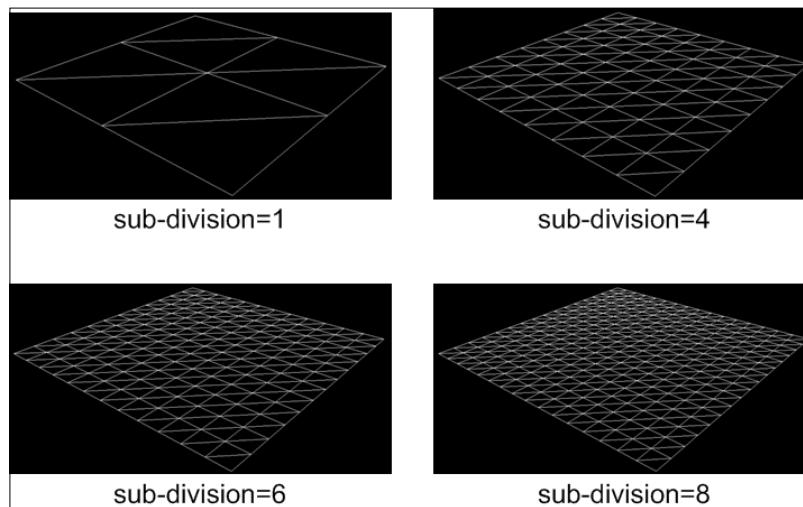
Since our planer mesh geometry is positioned at origin going from -5 to 5 on the X and Z axes, we have to place them in the appropriate place by translating them, otherwise they would overlay each other.

Next, we first bind the shader program. Then we pass the shader uniforms which include the `sub_divisions` uniform and the combined modelview projection matrix (`MVP`) uniform. Then we pass the attributes by issuing a call to the `glDrawElements` function. We then add the relative translation for each instance to get a new modelview matrix for the next draw call. This is repeated three times to get all four planar meshes placed properly in the world space.

In this recipe, we handle keyboard input to allow the user to change the subdivision level dynamically. We first attach our keyboard event handler (`OnKey`) to `glutKeyboardFunc`. The keyboard event handler is defined as follows:

```
void OnKey(unsigned char key, int x, int y) {
    switch(key) {
        case ',': sub_divisions--; break;
        case '.': sub_divisions++; break;
    }
    sub_divisions = max(1,min(8, sub_divisions));
    glutPostRedisplay();
}
```

We can change the subdivision levels by pressing the , and . keys. We then check to make sure that the subdivisions are within the allowed limit. Finally, we request the freeglut function, `glutPostRedisplay()`, to repaint the window to show the new mesh. Compiling and running the demo code displays four planar meshes. Pressing the , key decreases the subdivision level and the . key increases the subdivision level. The output from the subdivision geometry shader showing multiple subdivision levels is displayed in the following screenshot:



See also

You can view the Geometry shader tutorial part 1 and 2 at Geeks3D:

<http://www.geeks3d.com/20111111/simple-introduction-to-geometry-shaders-glsl-opengl-tutorial-part1/>

<http://www.geeks3d.com/20111117/simple-introduction-to-geometry-shader-in-glsl-part-2/>

Dynamically subdividing a plane using the geometry shader with instanced rendering

In order to avoid pushing the same data multiple times, we can exploit the instanced rendering functions. We will now see how we can omit the multiple `glDrawElements` calls in the previous recipe with a single `glDrawElementsInstanced` call.

Getting ready

Before doing this, we assume that the reader knows how to use the geometry shader in the OpenGL 3.3 core profile. The code for this recipe is in the `Chapter1\SubdivisionGeometryShader_Instanced` directory.

How to do it...

Converting the previous recipe to use instanced rendering requires the following steps:

1. Change the vertex shader to handle the instance modeling matrix and output world space positions (`shaders/shader.vert`).

```
#version 330 core
layout(location=0) in vec3 vVertex;
uniform mat4 M[4];
void main()
{
    gl_Position = M[gl_InstanceID]*vec4(vVertex, 1);
}
```

2. Change the geometry shader to replace the MVP matrix with the PV matrix (`shaders/shader.geom`).

```
#version 330 core
layout(triangles) in;
layout(triangle_strip, max_vertices=256) out;
uniform int sub_divisions;
uniform mat4 PV;

void main()
{
    vec4 v0 = gl_in[0].gl_Position;
    vec4 v1 = gl_in[1].gl_Position;
    vec4 v2 = gl_in[2].gl_Position;
    float dx = abs(v0.x-v2.x)/sub_divisions;
    float dz = abs(v0.z-v1.z)/sub_divisions;
```

```
float x=v0.x;
float z=v0.z;
for(int j=0;j<sub_divisions*sub_divisions;j++) {
    gl_Position = PV * vec4(x,0,z,1);           EmitVertex();
    gl_Position = PV * vec4(x,0,z+dz,1);           EmitVertex();
    gl_Position = PV * vec4(x+dx,0,z,1);           EmitVertex();
    gl_Position = PV * vec4(x+dx,0,z+dz,1);   EmitVertex();
    EndPrimitive();
    x+=dx;
    if((j+1) %sub_divisions == 0) {
        x=v0.x;
        z+=dz;
    }
}
```

3. Initialize the per-instance model matrices (M).

```
void OnInit() {
    //set the instance modeling matrix
    M[0] = glm::translate(glm::mat4(1), glm::vec3(-5,0,-5));
    M[1] = glm::translate(M[0], glm::vec3(10,0,0));
    M[2] = glm::translate(M[1], glm::vec3(0,0,10));
    M[3] = glm::translate(M[2], glm::vec3(-10,0,0));
    ..
    shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("PV");
    shader.AddUniform("M");
    shader.AddUniform("sub_divisions");
    glUniform1i(shader("sub_divisions"), sub_divisions);
    glUniformMatrix4fv(shader("M"), 4, GL_FALSE,
    glm::value_ptr(M[0]));
    shader.UnUse();
```

4. Render instances using the glDrawElementInstanced call.

```
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glm::mat4 T =glm::translate(glm::mat4(1.0f),
    glm::vec3(0.0f, 0.0f, dist));
    glm::mat4 Rx=glm::rotate(T,rX,glm::vec3(1.0f, 0.0f,
    0.0f));
    glm::mat4 V =glm::rotate(Rx,rY,glm::vec3(0.0f,
    1.0f,0.0f));
    glm::mat4 PV = P*V;
```

```

        shader.Use();
        glUniformMatrix4fv(shader("PV"), 1, GL_FALSE,
        glm::value_ptr(PV));
        glUniform1i(shader("sub_divisions"), sub_divisions);
        glDrawElementsInstanced(GL_TRIANGLES,
        6, GL_UNSIGNED_SHORT, 0, 4);
        shader.UnUse();
        glutSwapBuffers();
    }
}

```

How it works...

First, we need to store the model matrix for each instance separately. Since we have four instances, we store a uniform array of four elements (`M[4]`). Second, we multiply the per-vertex position (`vVertex`) with the model matrix for the current instance (`M[gl_InstanceID]`).



Note that the `gl_InstanceID` built-in attribute will be filled with the index of each instance automatically at the time of the `glDrawElementsInstanced` call. Also note that this built-in attribute is only accessible in the vertex shader.

The MVP matrix is omitted from the geometry shader since now the input vertex positions are in world space. So we only need to multiply them with the combined view projection (`PV`) matrix. On the application side, the MV matrix is removed. Instead, we store the model matrix array for all four instances (`glm::mat4 M[4]`). The values of these matrices are initialized in the `OnInit()` function as follows:

```

M[0] = glm::translate(glm::mat4(1), glm::vec3(-5, 0, -5));
M[1] = glm::translate(M[0], glm::vec3(10, 0, 0));
M[2] = glm::translate(M[1], glm::vec3(0, 0, 10));
M[3] = glm::translate(M[2], glm::vec3(-10, 0, 0));

```

The rendering function, `OnRender()`, creates the combined view projection matrix (`PV`) and then calls `glDrawElementsInstanced`. The first four parameters are similar to the `glDrawElements` function. The final parameter is the total number of instances desired. Instanced rendering is an efficient mechanism for rendering identical geometry whereby the `GL_ARRAY_BUFFER` and `GL_ELEMENT_ARRAY_BUFFER` bindings are shared between instances allowing the GPU to do efficient resource access and sharing.

```

void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f,
    0.0f, dist));
    glm::mat4 Rx = glm::rotate(T, rX, glm::vec3(1.0f, 0.0f,
    0.0f));

```

```
glm::mat4 V = glm::rotate(Rx, rY, glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 PV = P*V;
shader.Use();
    glUniformMatrix4fv(shader("PV"), 1, GL_FALSE,
    glm::value_ptr(PV));
    glUniform1i(shader("sub_divisions"), sub_divisions);
    glDrawElementsInstanced(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0,
    4);
shader.UnUse();
glutSwapBuffers();
}
```

There is always a limit on the maximum number of matrices one can output from the vertex shader and this has some performance implications as well. Some performance improvements can be obtained by replacing the matrix storage with translation and scaling vectors, and an orientation quaternion which can then be converted on the fly into a matrix in the shader.

See also

The official OpenGL wiki can be found at http://www.opengl.org/wiki/Built-in_Variable_%28GLSL%29.

An instance rendering tutorial from OGLDev can be found at <http://ogldev.atspace.co.uk/www/tutorial33/tutorial33.html>.

Drawing a 2D image in a window using the fragment shader and the SOIL image loading library

We will wrap up this chapter with a recipe for creating a simple image viewer in the OpenGL v3.3 core profile using the SOIL image loading library.

Getting ready

After setting up the Visual Studio environment, we can now work with the SOIL library. The code for this recipe is in the `Chapter1/ImageLoader` directory.

How to do it...

Let us now implement the image loader by following these steps:

1. Load the image using the SOIL library. Since the loaded image from SOIL is inverted vertically, we flip the image on the Y axis.

```
int texture_width = 0, texture_height = 0, channels=0;
GLubyte* pData = SOIL_load_image(filename.c_str(),
&texture_width, &texture_height, &channels,
SOIL_LOAD_AUTO);
if(pData == NULL) {
    cerr<<"Cannot load image: "<<filename.c_str()<<endl;
    exit(EXIT_FAILURE);
}
int i,j;
for( j = 0; j*2 < texture_height; ++j )
{
    int index1 = j * texture_width * channels;
    int index2 = (texture_height - 1 - j) * texture_width *
    channels;
    for( i = texture_width * channels; i > 0; --i )
    {
        GLubyte temp = pData[index1];
        pData[index1] = pData[index2];
        pData[index2] = temp;
        ++index1;
        ++index2;
    }
}
```

2. Set up the OpenGL texture object and free the data allocated by the SOIL library.

```
glGenTextures(1, &textureID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textureID);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
 GL_CLAMP);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
 GL_CLAMP);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture_width,
 texture_height, 0, GL_RGB, GL_UNSIGNED_BYTE, pData);
SOIL_free_image_data(pData);
```

3. Set up the vertex shader to output the clip space position (`shaders/shader.vert`).

```
#version 330 core
layout(location=0) in vec2 vVertex;
smooth out vec2 vUV;
void main()
{
    gl_Position = vec4(vVertex*2.0-1,0,1);
    vUV = vVertex;
}
```

4. Set up the fragment shader that samples our image texture (`shaders/shader.frag`).

```
#version 330 core
layout (location=0) out vec4 vFragColor;
smooth in vec2 vUV;
uniform sampler2D textureMap;
void main()
{
    vFragColor = texture(textureMap, vUV);
}
```

5. Set up the application code using the `GLSLShader` shader class.

```
shader.LoadFromFile(GL_VERTEX_SHADER,
"shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.
frag");
shader.CreateAndLinkProgram();
shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("textureMap");
    glUniform1i(shader("textureMap"), 0);
shader.UnUse();
```

6. Set up the geometry and topology and pass data to the GPU using buffer objects.

```
vertices[0] = glm::vec2(0.0,0.0);
vertices[1] = glm::vec2(1.0,0.0);
vertices[2] = glm::vec2(1.0,1.0);
vertices[3] = glm::vec2(0.0,1.0);
GLushort* id=&indices[0];
*id++ =0;
*id++ =1;
*id++ =2;
*id++ =0;
*id++ =2;
```

```
*id++ =3;

glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
glBindVertexArray(vaoID);
glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
&vertices[0], GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);
 glVertexAttribPointer(shader["vVertex"], 2, GL_FLOAT,
GL_FALSE, 0, 0);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices),
&indices[0], GL_STATIC_DRAW);
```

7. Set the shader and render the geometry.

```
void OnRender() {
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    shader.Use();
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
    shader.UnUse();
    glutSwapBuffers();
}
```

8. Release the allocated resources.

```
void OnShutdown() {
    shader.DeleteShaderProgram();
    glDeleteBuffers(1, &vboVerticesID);
    glDeleteBuffers(1, &vboIndicesID);
    glDeleteVertexArrays(1, &vaoID);
    glDeleteTextures(1, &textureID);
}
```

How it works...

The SOIL library provides a lot of functions but for now we are only interested in the `SOIL_load_image` function.

```
int texture_width = 0, texture_height = 0, channels=0;
GLubyte* pData = SOIL_load_image(filename.c_str(), &texture_width,
&texture_height, &channels, SOIL_LOAD_AUTO);
if(pData == NULL) {
    cerr<<"Cannot load image: "<<filename.c_str()<<endl;
    exit(EXIT_FAILURE);
}
```

The first parameter is the image file name. The next three parameters return the texture width, texture height, and total color channels in the image. These are used when generating the OpenGL texture object. The final parameter is the flag which is used to control further processing on the image. For this simple example, we will use the `SOIL_LOAD_AUTO` flag which keeps all of the loading settings set to default. If the function succeeds, it returns `unsigned char*` to the image data. If it fails, the return value is `NULL` (0). Since the image data loaded by `SOIL` is vertically flipped, we then use two nested loops to flip the image data on the Y axis.

```
int i,j;
for( j = 0; j*2 < texture_height; ++j )
{
    int index1 = j * texture_width * channels;
    int index2 = (texture_height - 1 - j) * texture_width *
    channels;
    for( i = texture_width * channels; i > 0; --i )
    {
        GLubyte temp = pData[index1];
        pData[index1] = pData[index2];
        pData[index2] = temp;
        ++index1;
        ++index2;
    }
}
```

After the image data is loaded, we generate an OpenGL texture object and pass this data to the texture memory.

```
glGenTextures(1, &textureID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textureID);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture_width,
 texture_height, 0, GL_RGB, GL_UNSIGNED_BYTE, pData);
 SOIL_free_image_data(pData);
```

As with every other OpenGL object, we have to first call `glGenTextures`. The first parameter is the total number of texture objects we need and the second parameter holds the ID of the texture object generated. After generation of the texture object, we set the active texture unit by calling `glActiveTexture(GL_TEXTURE0)` and then bind the texture to the active texture unit by calling `glBindTextures(GL_TEXTURE_2D, &textureID)`. Next, we adjust the texture parameters like the texture filtering for minification and magnification, as well as the texture wrapping modes for S and T texture coordinates. After these calls, we pass the loaded image data to the `glTexImage2D` function.

The `glTexImage2D` function is where the actual allocation of the texture object takes place. The first parameter is the texture target (in our case this is `GL_TEXTURE_2D`). The second parameter is the mipmap level which we keep to 0. The third parameter is the internal format. We can determine this by looking at the image properties. The fourth and fifth parameters store the texture width and height respectively. The sixth parameter is 0 for no border and 1 for border. The seventh parameter is the image format. The eighth parameter is the type of the image data pointer, and the final parameter is the pointer to the raw image data. After this function, we can safely release the image data allocated by SOIL by calling `SOIL_free_image_data(pData)`.

There's more...

In this recipe, we use two shaders, the vertex shader and the fragment shader. The vertex shader outputs the clip space position from the input vertex position (`vVertex`) by simple arithmetic. Using the vertex positions, it also generates the texture coordinates (`vUV`) for sampling of the texture in the fragment shader.

```
gl_Position = vec4(vVertex*2.0-1,0,1);
vUV = vVertex;
```

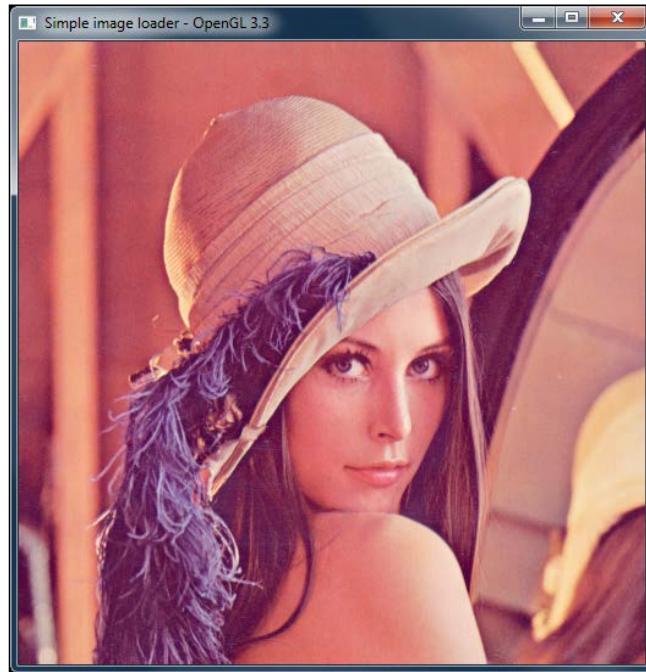
The fragment shader has the texture coordinates smoothly interpolated from the vertex shader stage through the rasterizer. The image that we loaded using SOIL is passed to a texture sampler (`uniform sampler2D textureMap`) which is then sampled using the input texture coordinates (`vFragColor = texture(textureMap, vUV)`). So in the end, we get the image displayed on the screen.

The application side code is similar to the previous recipe. The changes include an addition of the `textureMap` sampler uniform.

```
shader.Use();
shader.AddAttribute("vVertex");
shader.AddUniform("textureMap");
glUniform1i(shader("textureMap"), 0);
shader.UnUse();
```

Since this uniform will not change throughout the lifetime of the application, we initialize it once only. The first parameter of `glUniform1i` is the location of the uniform. We set the value of the sampler uniform to the active texture unit where the texture is bound. In our case, the texture is bound to texture unit 0, that is, `GL_TEXTURE0`. Therefore we pass 0 to the uniform. If it was bound to `GL_TEXTURE1`, we would pass 1 to the uniform.

The `OnShutdown()` function is similar to the earlier recipes. In addition, this code adds deletion of the OpenGL texture object. The rendering code first clears the color and depth buffers. Next, it binds the shader program and then invokes the `glDrawElements` call to render the triangles. Finally the shader is unbound and then the `glutSwapBuffers` function is called to display the current back buffer as the next front buffer. Compiling and running this code displays the image in a window as shown in the following screenshot:



Using image loading libraries like `SOIL` and a fragment shader, we can make a simple image viewer with basic GLSL functionality. More elaborate effects may be achieved by using techniques detailed in the later recipes of this book.

2

3D Viewing and Object Picking

The recipes covered in this chapter include:

- ▶ Implementing a vector-based camera model with FPS style input support
- ▶ Implementing the free camera
- ▶ Implementing target camera
- ▶ Implementing the view frustum culling
- ▶ Implementing object picking using the depth buffer
- ▶ Implementing object picking using color based picking
- ▶ Implementing object picking using scene intersection queries

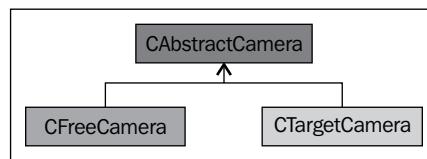
Introduction

In this chapter, we will look at the recipes for handling 3D viewing tasks and object picking in OpenGL v3.3 and above. All of the real-time simulations, games, and other graphics applications require a virtual camera or a virtual viewer from the point of view of which the 3D scene is rendered. The virtual camera is itself placed in the 3D world and has a specific direction called the camera look direction. Internally, the virtual camera is itself a collection of translations and rotations, which is stored inside the viewing matrix.

Moreover, projection settings for the virtual camera control how big or small the objects appear on screen. This is the kind of functionality which is controlled through the real world camera lens. These are controlled through the projection matrix. In addition to specifying the viewing and projection matrices, the virtual camera may also help with reducing the amount of geometry pushed to the GPU. This is through a process called view frustum culling. Rather than rendering all of the objects in the scene, only those that are visible to the virtual camera are rendered, thus improving the runtime performance of the application.

Implementing a vector-based camera with FPS style input support

We will begin this chapter by designing a simple class to handle the camera. In a typical OpenGL application, the viewing operations are carried out to place a virtual object on screen. We leave the details of the transformations required in between to a typical graduate text on computer graphics like the one given in the *See also* section of this recipe. This recipe will focus on designing a simple and efficient camera class. We create a simple inheritance from a base class called `CAbstractCamera`. We will inherit two classes from this parent class, `CFreeCamera` and `CTargetCamera`, as shown in the following figure:



Getting ready

The code for this recipe is in the `Chapter2/src` directory. The `CAbstractCamera` class is defined in the `AbstractCamera.h/cpp` files.

```

class CAbstractCamera
{
public:
    CAbstractCamera(void);
    ~CAbstractCamera(void);
    void SetupProjection(const float fovy, const float aspectRatio,
    const float near=0.1f, const float far=1000.0f);
    virtual void Update() = 0;
    virtual void Rotate(const float yaw, const float pitch, const
    float roll);
    const glm::mat4 GetViewMatrix() const;
    const glm::mat4 GetProjectionMatrix() const;
    void SetPosition(const glm::vec3& v);
    const glm::vec3 GetPosition() const;
    void SetFOV(const float fov);
    const float GetFOV() const;
    const float GetAspectRatio() const;
    void CalcFrustumPlanes();
    bool IsPointInFrustum(const glm::vec3& point);
    bool IsSphereInFrustum(const glm::vec3& center, const float
    radius);
    bool IsBoxInFrustum(const glm::vec3& min, const glm::vec3& max);
  
```

```

void GetFrustumPlanes(glm::vec4 planes[6]);
glm::vec3 farPts[4];
glm::vec3 nearPts[4];
protected:
    float yaw, pitch, roll, fov, aspect_ratio, Znear, Zfar;
    static glm::vec3 UP;
    glm::vec3 look;
    glm::vec3 up;
    glm::vec3 right;
    glm::vec3 position;
    glm::mat4 V;           //view matrix
    glm::mat4 P;           //projection matrix
    CPlane planes[6];     //Frustum planes
};

```

We first declare the constructor/destructor pair. Next, the function for setting the projection for the camera is specified. Then some functions for updating the camera matrices based on rotation values are declared. Following these, the accessors and mutators are defined.

The class declaration is concluded with the view frustum culling-specific functions. Finally, the member fields are declared. The inheriting class needs to provide the implementation of one pure virtual function—`Update` (to recalculate the matrices and orientation vectors). The movement of the camera is based on three orientation vectors, namely, `look`, `up`, and `right`.

How to do it...

In a typical application, we will not use the `CAbstractCamera` class. Instead, we will use either the `CFreeCamera` class or the `CTargetCamera` class, as detailed in the following recipes. In this recipe, we will see how to handle input using the mouse and keyboard.

In order to handle the keyboard events, we perform the following processing in the idle callback function:

1. Check for the keyboard key press event.
2. If the `W` or `S` key is pressed, move the camera in the `look` vector direction:

```

if( GetAsyncKeyState(VK_W) & 0x8000)
    cam.Walk(dt);
if( GetAsyncKeyState(VK_S) & 0x8000)
    cam.Walk(-dt);

```

3. If the `A` or `D` key is pressed, move the camera in the `right` vector direction:

```

if( GetAsyncKeyState(VK_A) & 0x8000)
    cam.Strafe(-dt);
if( GetAsyncKeyState(VK_D) & 0x8000)
    cam.Strafe(dt);

```

4. If the Q or Z key is pressed, move the camera in the up vector direction:

```
if( GetAsyncKeyState(VK_Q) & 0x8000)
    cam.Lift(dt);
if( GetAsyncKeyState(VK_Z) & 0x8000)
    cam.Lift(-dt);
```

For handling mouse events, we attach two callbacks. One for mouse movement and the other for the mouse click event handling:

1. Define the mouse down and mouse move event handlers.
2. Determine the mouse input choice (the zoom or rotate state) in the mouse down event handler based on the mouse button clicked:

```
if(button == GLUT_MIDDLE_BUTTON)
    state = 0;
else
    state = 1;
```

3. If zoom state is chosen, calculate the `fov` value based on the drag amount and then set up the camera projection matrix:

```
if (state == 0) {
    fov += (y - oldY)/5.0f;
    cam.SetupProjection(fov, cam.GetAspectRatio());
}
```

4. If the rotate state is chosen, calculate the rotation amount (pitch and yaw). If mouse filtering is enabled, use the filtered mouse input, otherwise use the raw rotation amount:

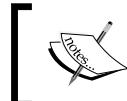
```
else {
    rY += (y - oldY)/5.0f;
    rX += (oldX-x)/5.0f;
    if(useFiltering)
        filterMouseMoves(rX, rY);
    else {
        mouseX = rX;
        mouseY = rY;
    }
    cam.Rotate(mouseX, mouseY, 0);
}
```

There's more...

It is always better to use filtered mouse input, which gives smoother movement. In the recipes, we use a simple average filter of the last 10 inputs weighted based on their temporal distance. So the previous input is given more weight and the 5th latest input is given less weight. The filtered result is used as shown in the following code snippet:

```
void filterMouseMoves(float dx, float dy) {
    for (int i = MOUSE_HISTORY_BUFFER_SIZE - 1; i > 0; --i) {
        mouseHistory[i] = mouseHistory[i - 1];
    }
    mouseHistory[0] = glm::vec2(dx, dy);
    float averageX = 0.0f, averageY = 0.0f, averageTotal = 0.0f,
    currentWeight = 1.0f;

    for (int i = 0; i < MOUSE_HISTORY_BUFFER_SIZE; ++i) {
        glm::vec2 tmp=mouseHistory[i];
        averageX += tmp.x * currentWeight;
        averageY += tmp.y * currentWeight;
        averageTotal += 1.0f * currentWeight;
        currentWeight *= MOUSE_FILTER_WEIGHT;
    }
    mouseX = averageX / averageTotal;
    mouseY = averageY / averageTotal;
}
```



When using filtered mouse input, make sure that the history buffer is filled with the appropriate initial value; otherwise you will see a sudden jerk in the first few frames.

See also

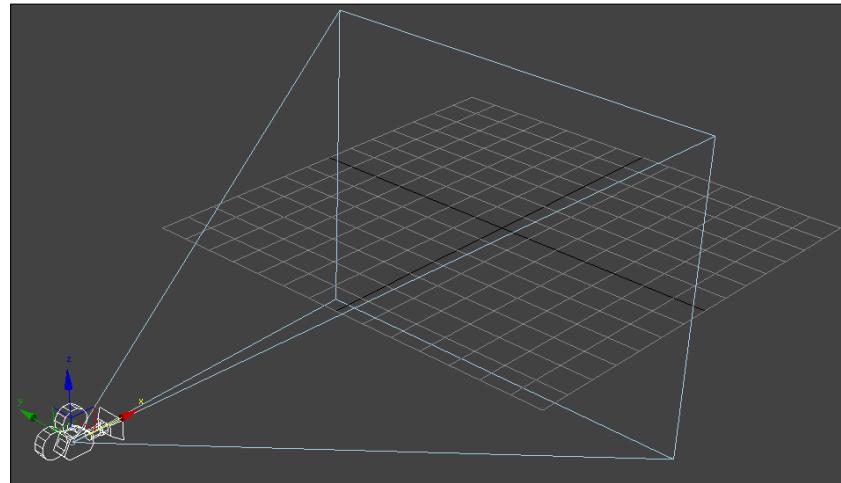
- ▶ Smooth mouse filtering FAQ by Paul Nettle (http://www.flipcode.com/archives/Smooth_Mouse_Filtering.shtml)
- ▶ Real-time Rendering 3rd Edition by Tomas Akenine-Moller, Eric Haines, and Naty Hoffman, AK Peters/CRC Press, 2008

Implementing the free camera

Free camera is the first camera type which we will implement in this recipe. A free camera does not have a fixed target. However it does have a fixed position from which it can look in any direction.

Getting ready

The following figure shows a free viewing camera. When we rotate the camera, it rotates at its position. When we move the camera, it keeps looking in the same direction.



The source code for this recipe is in the Chapter2/FreeCamera directory. The CFreeCamera class is defined in the Chapter2/src/FreeCamera.[h/cpp] files. The class interface is as follows:

```
class CFreeCamera : public CAbstractCamera
{
public:
    CFreeCamera(void);
    ~CFreeCamera(void);
    void Update();
    void Walk(const float dt);
    void Strafe(const float dt);
    void Lift(const float dt);
    void SetTranslation(const glm::vec3& t);
    glm::vec3 GetTranslation() const;
    void SetSpeed(const float speed);
    const float GetSpeed() const;
protected:
    float speed; //move speed of camera in m/s
    glm::vec3 translation;
};
```

How to do it...

The steps needed to implement the free camera are as follows:

1. Define the `CFreeCamera` class and add a vector to store the current translation.
2. In the `Update` method, calculate the new orientation (rotation) matrix, using the current camera orientations (that is, yaw, pitch, and roll amount):

```
glm::mat4 R = glm::yawPitchRoll(yaw,pitch,roll);
```



Make sure that the yaw, pitch, and roll angles are in radians.



3. Translate the camera position by the translation amount:

```
position+=translation;
```

If we need to implement a free camera which gradually comes to a halt, we should gradually decay the translation vector by adding the following code after the key events are handled:

```
glm::vec3 t = cam.GetTranslation();
if(glm::dot(t,t)>EPSILON2) {
    cam.SetTranslation(t*0.95f);
}
```

If no decay is needed, then we should clear the translation vector to 0 in the `CFreeCamera::Update` function after translating the position:

```
translation = glm::vec3(0);
```

4. Transform the `look` vector by the current rotation matrix, and determine the `right` and `up` vectors to calculate the orthonormal basis:

```
look = glm::vec3(R*glm::vec4(0,0,1,0));
up = glm::vec3(R*glm::vec4(0,1,0,0));
right = glm::cross(look, up);
```

5. Determine the camera target point:

```
glm::vec3 tgt = position+look;
```

6. Use the `glm::lookat` function to calculate the new view matrix using the camera position, target, and the `up` vector:

```
V = glm::lookAt(position, tgt, up);
```

There's more...

The Walk function simply translates the camera in the look direction:

```
void CFreeCamera::Walk(const float dt) {  
    translation += (look*dt);  
}
```

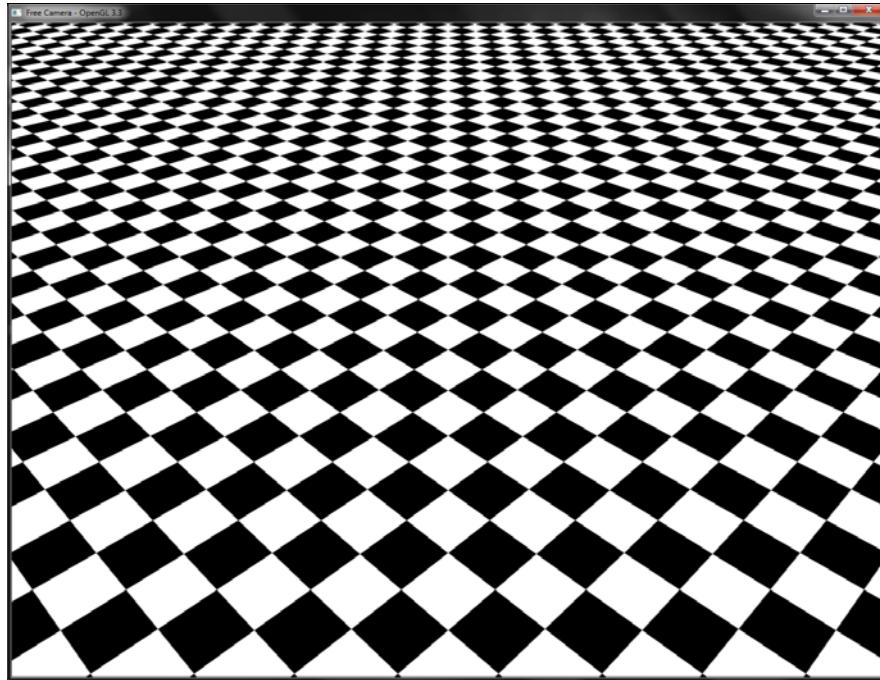
The Strafe function translates the camera in the right direction:

```
void CFreeCamera::Strafe(const float dt) {  
    translation += (right*dt);  
}
```

The Lift function translates the camera in the up direction:

```
void CFreeCamera::Lift(const float dt) {  
    translation += (up*dt);  
}
```

Running the demo application renders an infinite checkered plane as shown in the following figure. The free camera can be moved around by pressing the keys W, S, A, D, Q, and Z. Left-clicking the mouse rotates the camera at the current position to change the look direction. Middle-click zooms the camera in the look direction.



See also

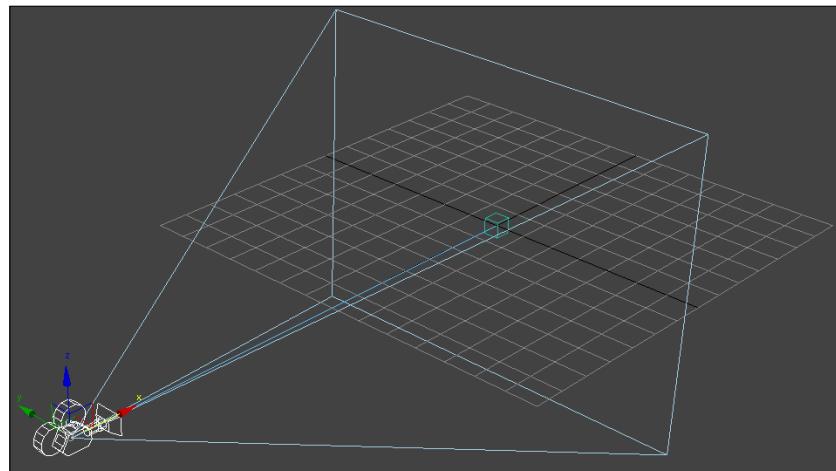
- ▶ DHPOWare OpenGL camera demo – Part 1 (<http://www.dhpoware.com/demos/glCamera1.html>)
- ▶ DHPOWare OpenGL camera demo – Part 2 (<http://www.dhpoware.com/demos/glCamera2.html>)
- ▶ DHPOWare OpenGL camera demo – Part 3 (<http://www.dhpoware.com/demos/glCamera3.html>)

Implementing the target camera

The target camera works the opposite way. Rather than the position, the target remains fixed, while the camera moves or rotates around the target. Some operations like panning, move both the target and the camera position together.

Getting ready

The following figure shows an illustration of a target camera. Note that the small box is the target position for the camera.



The code for this recipe resides in the Chapter2/TargetCamera directory. The CTargetCamera class is defined in the Chapter2/src/TargetCamera.[h/cpp] files. The class declaration is as follows:

```
class CTargetCamera : public CAbstractCamera
{
public:
```

```

CTargetCamera(void);
~CTargetCamera(void);
void Update();
void Rotate(const float yaw, const float pitch, const float
    roll);
void SetTarget(const glm::vec3 tgt);
const glm::vec3 GetTarget() const;
void Pan(const float dx, const float dy);
void Zoom(const float amount );
void Move(const float dx, const float dz);
protected:
    glm::vec3 target;
    float minRy, maxRy;
    float distance;
    float minDistance, maxDistance;
};

```

How to do it...

We implement the target camera as follows:

1. Define the CTargetCamera class with a target position (`target`), the rotation limits (`minRy` and `maxRy`), the distance between the target and the camera position (`distance`), and the distance limits (`minDistance` and `maxDistance`).
2. In the `Update` method, calculate the new orientation (rotation) matrix using the current camera orientations (that is, yaw, pitch, and roll amount):

```
glm::mat4 R = glm::yawPitchRoll(yaw,pitch,roll);
```

3. Use the distance to get a vector and then translate this vector by the current rotation matrix:

```
glm::vec3 T = glm::vec3(0,0,distance);
T = glm::vec3(R*glm::vec4(T,0.0f));
```

4. Get the new camera position by adding the translation vector to the target position:

```
position = target + T;
```

5. Recalculate the orthonormal basis and then the view matrix:

```
look = glm::normalize(target-position);
up = glm::vec3(R*glm::vec4(UP,0.0f));
right = glm::cross(look, up);
V = glm::lookAt(position, target, up);
```

There's more...

The Move function moves both the position and target by the same amount in both look and right vector directions.

```
void CTargetCamera::Move(const float dx, const float dy) {  
    glm::vec3 X = right*dx;  
    glm::vec3 Y = look*dy;  
    position += X + Y;  
    target += X + Y;  
    Update();  
}
```

The Pan function moves in the xy plane only, hence the up vector is used instead of the look vector:

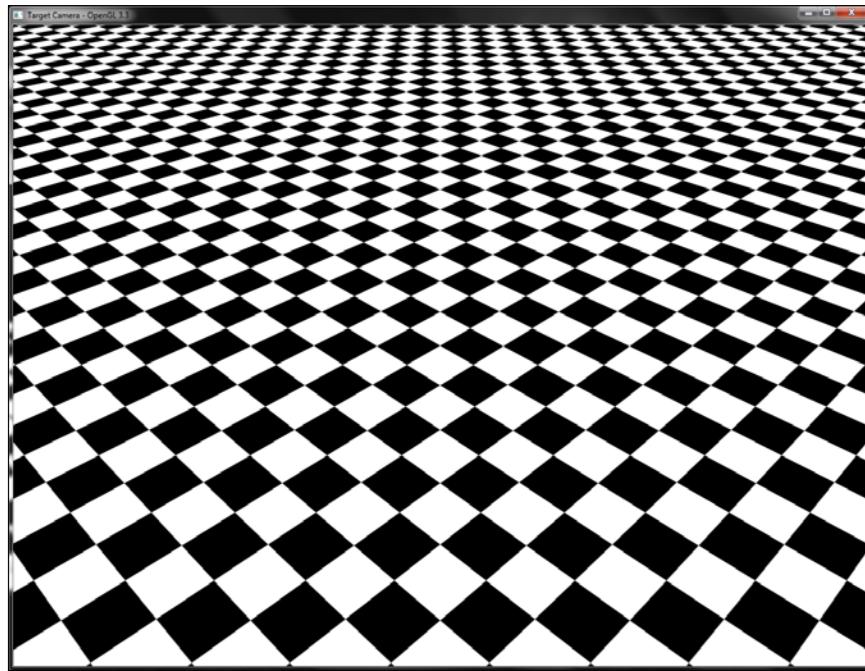
```
void CTargetCamera::Pan(const float dx, const float dy) {  
    glm::vec3 X = right*dx;  
    glm::vec3 Y = up*dy;  
    position += X + Y;  
    target += X + Y;  
    Update();  
}
```

The Zoom function moves the position in the look direction:

```
void CTargetCamera::Zoom(const float amount) {  
    position += look * amount;  
    distance = glm::distance(position, target);  
    Distance = std::max(minDistance,  
        std::min(distance, maxDistance));  
    Update();  
}
```

3D Viewing and Object Picking

The demonstration for this recipe renders an infinite checkered plane, as in the previous recipe, and is shown in the following figure:



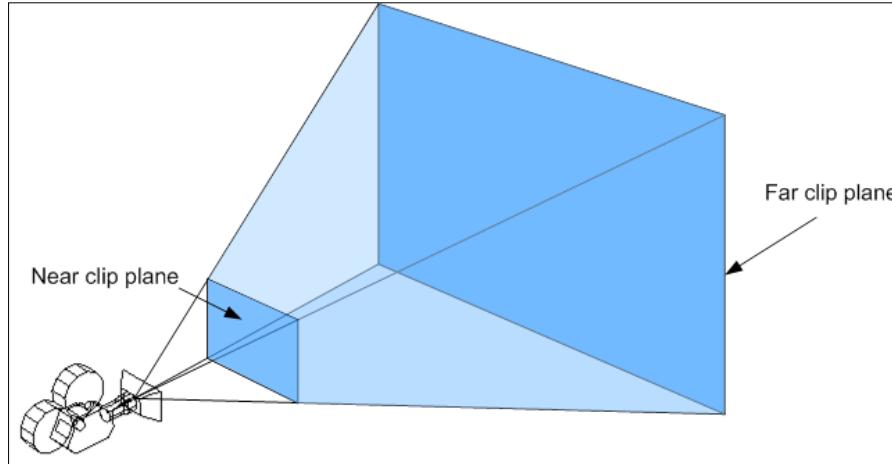
See also

- ▶ DHPoWare OpenGL camera demo – Part 1 (<http://www.dhpoware.com/demos/glCamera1.html>)
- ▶ DHPoWare OpenGL camera demo – Part 2 (<http://www.dhpoware.com/demos/glCamera2.html>)
- ▶ DHPoWare OpenGL camera demo – Part 3 (<http://www.dhpoware.com/demos/glCamera3.html>)

Implementing view frustum culling

When working with a lot of polygonal data, there is a need to reduce the amount of geometry pushed to the GPU for processing. There are several techniques for scene management, such as quadtrees, octrees, and bsp trees. These techniques help in sorting the geometry in visibility order, so that the objects are sorted (and some of these even culled from the display). This helps in reducing the work load on the GPU.

Even before such techniques can be used, there is an additional step which most graphics applications do and that is view frustum culling. This process removes the geometry if it is not in the current camera's view frustum. The idea is that if the object is not viewable, it should not be processed. A frustum is a chopped pyramid with its tip at the camera position and the base is at the far clip plane. The near clip plane is where the pyramid is chopped, as shown in the following figure. Any geometry inside the viewing frustum is displayed.



Getting ready

For this recipe, we will create a grid of points that are moved in a sine wave using a simple vertex shader. The geometry shader does the view frustum culling by only emitting vertices that are inside the viewing frustum. The calculation of the viewing frustum is carried out on the CPU, based on the camera projection parameters. We will follow the geometric approach in this tutorial. The code implementing this recipe is in the `Chapter2/ViewFrustumCulling` directory.

How to do it...

We will implement view frustum culling by taking the following steps:

1. Define a vertex shader that displaces the object-space vertex position using a sine wave in the y axis:

```
#version 330 core
layout(location = 0) in vec3 vVertex;
uniform float t;
const float PI = 3.141562;
void main()
```

```
{
    gl_Position=vec4(vVertex,1)+vec4(0,sin(vVertex.x*2*PI+t),0,0);
}
```

2. Define a geometry shader that performs the view frustum culling calculation on each vertex passed in from the vertex shader:

```
#version 330 core
layout (points) in;
layout (points, max_vertices=3) out;
uniform mat4 MVP;
uniform vec4 FrustumPlanes[6];
bool PointInFrustum(in vec3 p) {
    for(int i=0; i < 6; i++)
    {
        vec4 plane=FrustumPlanes[i];
        if ((dot(plane.xyz, p)+plane.w) < 0)
            return false;
    }
    return true;
}
void main()
{
    //get the basic vertices
    for(int i=0;i<gl_in.length(); i++) {
        vec4 vInPos = gl_in[i].gl_Position;
        vec2 tmp = (vInPos.xz*2-1.0)*5;
        vec3 V = vec3(tmp.x, vInPos.y, tmp.y);
        gl_Position = MVP*vec4(V,1);
        if(PointInFrustum(V)) {
            EmitVertex();
        }
    }
    EndPrimitive();
}
```

3. To render particles as rounded points, we do a simple trigonometric calculation by discarding all fragments that fall outside the radius of the circle:

```
#version 330 core
layout(location = 0) out vec4 vFragColor;
void main() {
    vec2 pos = (gl_PointCoord.xy-0.5);
    if(0.25<dot(pos,pos))    discard;
    vFragColor = vec4(0,0,1,1);
}
```

4. On the CPU side, call the `CAbstractCamera::CalcFrustumPlanes()` function to calculate the viewing frustum planes. Get the calculated frustum planes as a `glm::vec4` array by calling `CAbstractCamera::GetFrustumPlanes()`, and then pass these to the shader. The `xyz` components store the plane's normal, and the `w` coordinate stores the distance of the plane. After these calls we draw the points:

```
pCurrentCam->CalcFrustumPlanes();
glm::vec4 p[6];
pCurrentCam->GetFrustumPlanes(p);
pointShader.Use();
    glUniform1f(pointShader("t"), current_time);
    glUniformMatrix4fv(pointShader("MVP"), 1, GL_FALSE,
    glm::value_ptr(MVP));
    glUniform4fv(pointShader("FrustumPlanes"), 6,
    glm::value_ptr(p[0]));
    glBindVertexArray(pointVAOID);
    glDrawArrays(GL_POINTS, 0, MAX_POINTS);
pointShader.UnUse();
```

How it works...

There are two main parts of this recipe: calculation of the viewing frustum planes and checking if a given point is in the viewing frustum. The first calculation is carried out in the `CAbstractCamera::CalcFrustumPlanes()` function. Refer to the `Chapter2/src/AbstractCamera.cpp` files for details.

In this function, we follow the geometric approach, whereby we first calculate the eight points of the frustum at the near and far clip planes. Theoretical details about this method are well explained in the reference given in the `See also` section. Once we have the eight frustum points, we use three of these points successively to get the bounding planes of the frustum. Here, we call the `CPlane::FromPoints` function, which generates a `CPlane` object from the given three points. This is repeated to get all six planes.

Testing whether a point is in the viewing frustum is carried out in the geometry shader's `PointInFrustum` function, which is defined as follows:

```
bool PointInFrustum(in vec3 p) {
    for(int i=0; i < 6; i++) {
        vec4 plane=FrustumPlanes[i];
        if ((dot(plane.xyz, p)+plane.w) < 0)
            return false;
    }
    return true;
}
```

This function iterates through all of the six frustum planes. In each iteration, it checks the signed distance of the given point p with respect to the i th frustum plane. This is a simple dot product of the plane normal with the given point and adding the plane distance. If the signed distance is negative for any of the planes, the point is outside the viewing frustum so we can safely reject the point. If the point has a positive signed distance for all of the six frustum planes, it is inside the viewing frustum. Note that the frustum planes are oriented in such a way that their normals point inside the viewing frustum.

There's more...

The demonstration implementing this recipe shows two cameras, the local camera (camera 1) which shows the sine wave and a world camera (camera 2) which shows the whole world, including the first camera frustum. We can toggle the current camera by pressing 1 for camera 1 and 2 for camera 2. When in camera 1 view, dragging the left mouse button rotates the scene, and the information about the total number of points in the viewing frustum are displayed in the title bar. In the camera 2 view, left-clicking rotates camera 1, and the displayed viewing frustum is updated so we can see what the camera view should contain.

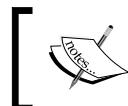
In order to see the total number of visible vertices emitted from the geometry shader, we use a hardware query. The whole shader and the rendering code are bracketed in the begin/end query call as shown in the following code:

```
glBeginQuery(GL_PRIMITIVES_GENERATED, query);
pointShader.Use();
    glUniform1f(pointShader("t"), current_time);
    glUniformMatrix4fv(pointShader("MVP"), 1, GL_FALSE,
        glm::value_ptr(MVP));
    glUniform4fv(pointShader("FrustumPlanes"), 6,
        glm::value_ptr(p[0]));
    glBindVertexArray(pointVAOID);
    glDrawArrays(GL_POINTS, 0, MAX_POINTS);
pointShader.UnUse();
glEndQuery(GL_PRIMITIVES_GENERATED);
```

After these calls, the query result is retrieved by calling:

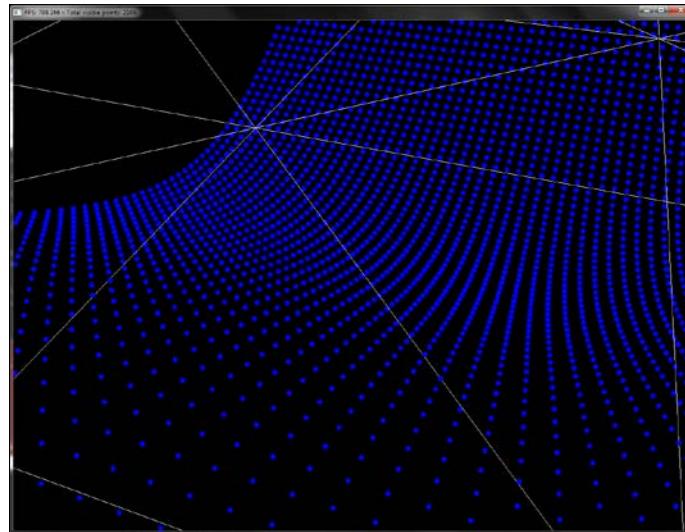
```
GLuint res;
glGetQueryObjectuiv(query, GL_QUERY_RESULT, &res);
```

If successful, this call returns the total number of vertices emitted from the geometry shader, and that is the total number of vertices in the viewing frustum.

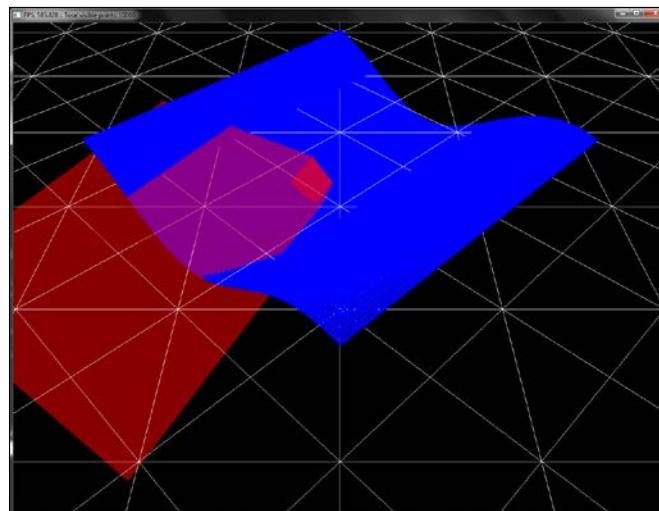


Note that for the camera 2 view, all points are emitted. Hence, the total number of points is displayed in the title bar.

When in the camera 1 view (see the following figure), we see the close-up of the wave as it displaces the points in the Y direction. In this view, the points are rendered in blue color. Moreover, the total number of visible points is written in the title bar. The frame rate is also written to show the performance benefit from view frustum culling.



When in the camera 2 view (see the following figure), we can click-and-drag the left mouse button to rotate camera 1. This allows us to see the updated viewing frustum and the visible points. In the camera 2 view, visible points in the camera 1 view frustum are rendered in magenta color, the viewing frustum planes are in red color, and the invisible points (in camera 1 viewing frustum) are in blue color.



See also

Lighthouse 3D view frustum culling tutorial (<http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/>).

Implementing object picking using the depth buffer

Often when working on projects, we need the ability to pick graphical objects on screen. While in OpenGL versions before OpenGL 3.0, the selection buffer was used for this purpose, this buffer is removed in the modern OpenGL 3.3 core profile. However, this leaves us with some alternate methods. We will implement a simple picking technique using the depth buffer in this recipe.

Getting ready

The code for this recipe is in the `Chapter2/Picking_DepthBuffer` folder. Relevant source files are in the `Chapter2/src` folder.

How to do it...

Picking using depth buffer can be implemented as follows:

1. Enable depth testing:

```
glEnable(GL_DEPTH_TEST);
```

2. In the mouse down event handler, read the depth value from the depth buffer using the `glReadPixels` function at the clicked point:

```
glReadPixels( x, HEIGHT-y, 1, 1, GL_DEPTH_COMPONENT,
GL_FLOAT, &winZ);
```

3. Unproject the 3D point, `vec3(x,HEIGHT-y,winZ)`, to obtain the object-space point from the clicked screen-space point `x,y` and the depth value `winZ`. Make sure to invert the `y` value by subtracting `HEIGHT` from the screen-space `y` value:

```
glm::vec3 objPt = glm::unProject(glm::vec3
(x,HEIGHT-y,winZ), MV, P, glm::vec4(0,0,WIDTH, HEIGHT));
```

4. Check the distances of all of the scene objects from the object-space point `objPt`. If the distance is within the bounds of the object and the distance of the object is the nearest to the camera, store the index of the object:

```
size_t i=0;
float minDist = 1000;
selected_box=-1;
```

```
for(i=0;i<3;i++) {  
    float dist = glm::distance(box_positions[i], objPt);  
    if( dist<1 && dist<minDist) {  
        selected_box = i;  
        minDist = dist;  
    }  
}
```

5. Based on the selected index, color the object as selected:

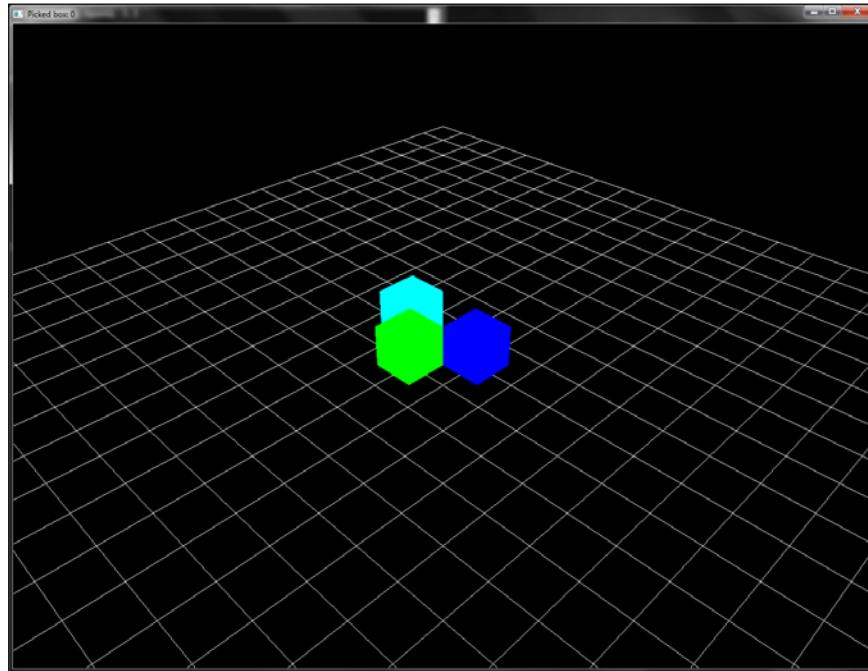
```
glm::mat4 T = glm::translate(glm::mat4(1),  
    box_positions[0]);  
cube->color =  
    (selected_box==0)?glm::vec3(0,1,1):glm::vec3(1,0,0);  
cube->Render(glm::value_ptr(MVP*T));  
  
T = glm::translate(glm::mat4(1), box_positions[1]);  
cube->color =  
    (selected_box==1)?glm::vec3(0,1,1):glm::vec3(0,1,0);  
cube->Render(glm::value_ptr(MVP*T));  
  
T = glm::translate(glm::mat4(1), box_positions[2]);  
cube->color =  
    (selected_box==2)?glm::vec3(0,1,1):glm::vec3(0,0,1);  
cube->Render(glm::value_ptr(MVP*T));
```

How it works...

This recipe renders three cubes in red, green, and blue on the screen. When the user clicks on any of these cubes, the depth buffer is read to find the depth value at the clicked point. The object-space point is then obtained by unprojecting (`glm::unProject`) the clicked point (`x, HEIGHT-y, winZ`). A loop is then iterated over all objects in the scene to find the nearest object to the object-space point. The index of the nearest intersected object is then stored.

There's more...

In the demonstration application for this recipe, when the user clicks on any cube, the currently selected box changes color to cyan to signify selection, as shown in the following figure:



See also

Picking tutorial at OGLDEV (<http://ogldev.atspace.co.uk/www/tutorial29/tutorial29.html>).

Implementing object picking using color

Another method which is used for picking objects in a 3D world is color-based picking. In this recipe, we will use the same scene as in the last recipe.

Getting ready

The code for this recipe is in the Chapter2/Picking_ColorBuffer folder. Relevant source files are in the Chapter2/src folder.

How to do it...

To enable picking with the color buffer, the following steps are needed:

1. Disable dithering. This is done to prevent any color mismatch during the query:

```
glDisable(GL_DITHER);
```

2. In the mouse down event handler, read the color value at the clicked position from the color buffer using the `glReadPixels` function:

```
GLubyte pixel[4];
glReadPixels(x, HEIGHT-y, 1, 1, GL_RGBA, GL_UNSIGNED_BYTE,
pixel);
```

3. Compare the color value at the clicked point to the color values of all objects to find the intersection:

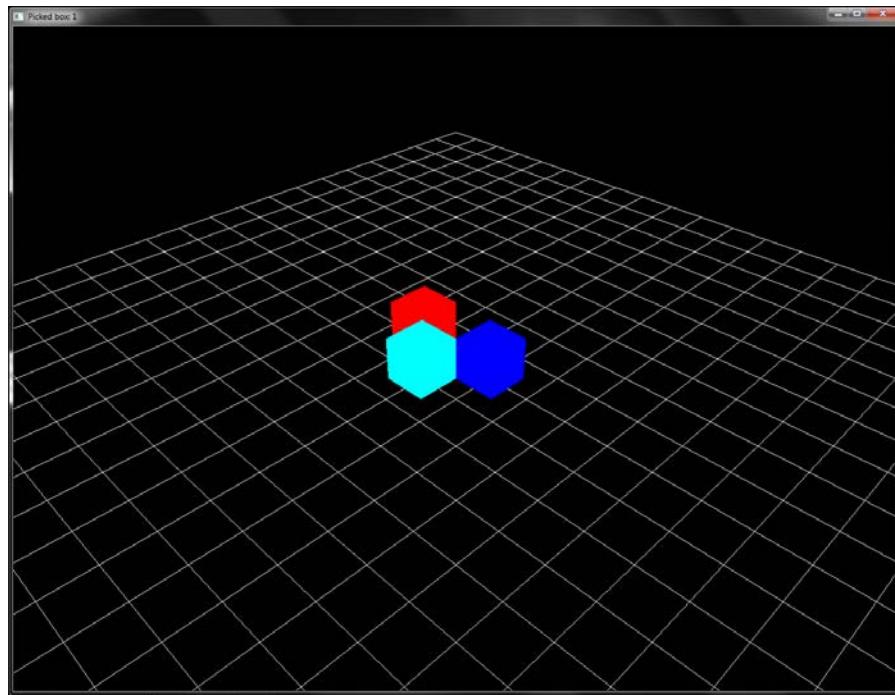
```
selected_box=-1;
if(pixel[0]==255 && pixel[1]==0 && pixel[2]==0) {
    cout<<"picked box 1"<<endl;
    selected_box = 0;
}
if(pixel[0]==0 && pixel[1]==255 && pixel[2]==0) {
    cout<<"picked box 2"<<endl;
    selected_box = 1;
}
if(pixel[0]==0 && pixel[1]==0 && pixel[2]==255) {
    cout<<"picked box 3"<<endl;
    selected_box = 2;
}
```

How it works...

This method is simple to implement. We simply check the color of the pixel where the mouse is clicked. Since dithering might generate a different color value, we disable dithering. The pixel's r, g, and b values are then checked against all of the scene objects and the appropriate object is selected. We could also have used the float data type, `GL_FLOAT`, when reading and comparing the pixel value. However, due to floating point imprecision, we might not have an accurate test. Therefore, we use the integral data type `GL_UNSIGNED_BYTE`.

3D Viewing and Object Picking

The demonstration application for this recipe uses the scene from the previous recipe. In this demonstration also, the user left-clicks on a box and the selection is highlighted in cyan, as shown in the following figure:



See also

Lighthouse3d color coded picking tutorial (<http://www.lighthouse3d.com/opengl/picking/index.php3?color1>).

Implementing object picking using scene intersection queries

The final method we will cover for picking involves casting rays in the scene to determine the nearest object to the viewer. We will use the same scene as in the last two recipes, three cubes (red, green, and blue colored) placed near the origin.

Getting ready

The code for this recipe is in the `Chapter2/Picking_SceneIntersection` folder. Relevant source files are in the `Chapter2/src` folder.

How to do it...

For picking with scene intersection queries, take the following steps:

1. Get two object-space points by unprojecting the screen-space point (x , $HEIGHT-y$), with different depth value, one at $z=0$ and the other at $z=1$:

```
glm::vec3 start = glm::unProject(glm::vec3(x,HEIGHT-y,0),
    MV, P, glm::vec4(0,0,WIDTH,HEIGHT));
glm::vec3 end = glm::unProject(glm::vec3(x,HEIGHT-y,1),
    MV, P, glm::vec4(0,0,WIDTH,HEIGHT));
```

2. Get the current camera position as `eyeRay.origin` and get `eyeRay.direction` by subtracting and normalizing the difference of the two object-space points, `end` and `start`, as follows:

```
eyeRay.origin      = cam.GetPosition();
eyeRay.direction  = glm::normalize(end-start);
```

3. For all of the objects in the scene, find the intersection of the eye ray with the **Axially Aligned Bounding Box (AABB)** of the object. Store the nearest intersected object index:

```
float tMin = numeric_limits<float>::max();
selected_box = -1;
for(int i=0;i<3;i++) {
    glm::vec2 tMinMax = intersectBox(eyeRay, boxes[i]);
    if(tMinMax.x<tMinMax.y && tMinMax.x<tMin) {
        selected_box=i;
        tMin = tMinMax.x;
    }
}
if(selected_box==-1)
    cout<<"No box picked"=<<endl;
else
    cout<<"Selected box: "<<selected_box<<endl;
```

How it works...

The method discussed in this recipe first casts a ray from the camera origin in the clicked direction, and then checks all of the scene objects' bounding boxes for intersection. There are two sub parts: estimation of the ray direction from the clicked point and the ray AABB intersection. We first focus on the estimation of the ray direction from the clicked point.

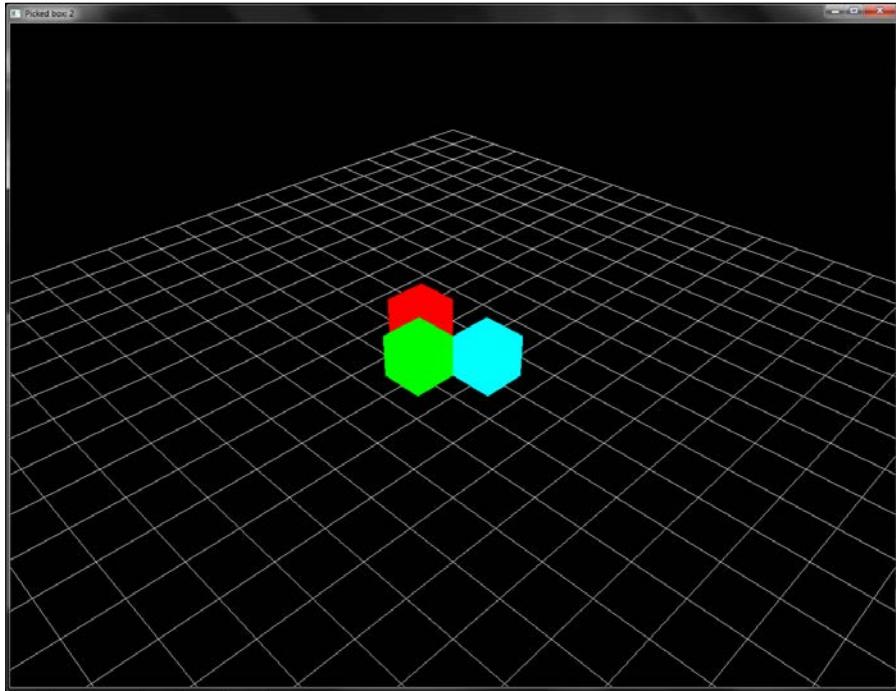
We know that after projection, the x and y values are in the -1 to 1 range. The z or depth values are in the 0 to 1 range, with 0 at the near clip plane and 1 at the far clip plane. We first take the screen-space point and unproject it taking the near clip plane z value of 0. This gives us the object-space point at the near clip plane. Next, we pass the screen-space point and unproject it with the z value of 1. This gives us the object-space point at the far clip plane. Subtracting the two unprojected object-space points gives us the ray direction. We store the camera position as `eyeRay.origin` and normalize the ray direction as `eyeRay.direction`.

After calculating the eye ray, we check it for intersection with all of the scene geometries. If the object-bounding box intersects the eye ray and it is the nearest intersection, we store the index of the object. The `intersectBox` function is defined as follows:

```
glm::vec2 intersectBox(const Ray& ray, const Box& cube) {  
    glm::vec3 inv_dir = 1.0f/ray.direction;  
    glm::vec3 tMin = (cube.min - ray.origin) * inv_dir;  
    glm::vec3 tMax = (cube.max - ray.origin) * inv_dir;  
    glm::vec3 t1 = glm::min(tMin, tMax);  
    glm::vec3 t2 = glm::max(tMin, tMax);  
    float tNear = max(max(t1.x, t1.y), t1.z);  
    float tFar = min(min(t2.x, t2.y), t2.z);  
    return glm::vec2(tNear, tFar);  
}
```

There's more...

The `intersectBox` function works by finding the intersection of the ray with a pair of slabs for each of the three axes individually. Next it finds the `tNear` and `tFar` values. The box can only intersect with the ray if `tNear` is less than `tFar` for all of the three axes. So the code finds the smallest `tFar` value and the largest `tMin` value. If the smallest `tFar` value is less than the largest `tNear` value, the ray misses the box. For further details, refer to the See also section. The output result from the demonstration application for this recipe uses the same scene as in the last two recipes. In this case also, left-clicking the mouse selects the box, which is highlighted in cyan, as shown in the following figure:



See also

[http://www.siggraph.org/education/materials/HyperGraph/raytrace/
rtinter3.htm](http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm).

3

Offscreen Rendering and Environment Mapping

In this chapter, we will cover:

- ▶ Implementing the twirl filter using fragment shader
- ▶ Rendering a skybox using static cube mapping
- ▶ Implementing a mirror with render-to-texture using FBO
- ▶ Rendering a reflective object using dynamic cube mapping
- ▶ Implementing area filtering (sharpening/blurring/embossing) on an image using convolution
- ▶ Implementing the glow effect

Introduction

Offscreen rendering functionality is a powerful feature of modern graphics API. In modern OpenGL, this is implemented by using the **Framebuffer objects (FBOs)**. Some of the applications of the offscreen rendering include post processing effects such as glows, dynamic cubemaps, mirror effect, deferred rendering techniques, image processing techniques, and so on. Nowadays almost all games use this feature to carry out stunning visual effects with high rendering quality and detail. With the FBOs, the offscreen rendering is greatly simplified, as the programmer uses FBO the way he would use any other OpenGL object. This chapter will focus on using FBO to carry out image processing effects for implementing digital convolution and glow. In addition, we will also elaborate on how to use the FBO for mirror effect and dynamic cube mapping.

Implementing the twirl filter using the fragment shader

We will use a simple image manipulation operator in the fragment shader by implementing the twirl filter on the GPU.

Getting ready

This recipe builds up on the image loading recipe from *Chapter 1, Introduction to Modern OpenGL*. The code for this recipe is contained in the `Chapter3/TwirlFilter` directory.

How to do it...

Let us get started with the recipe as follows:

1. Load the image as in the `ImageLoader` recipe from *Chapter 1, Introduction to Modern OpenGL*. Set the texture wrap mode to `GL_CLAMP_TO_BORDER`.

```
int texture_width = 0, texture_height = 0, channels=0;
GLubyte* pData = SOIL_load_image(filename.c_str(),
&texture_width, &texture_height, &channels,
SOIL_LOAD_AUTO);
int i,j;
for( j = 0; j*2 < texture_height; ++j )
{
    int index1 = j * texture_width * channels;
    int index2 = (texture_height - 1 - j) * texture_width *
    channels;
    for( i = texture_width * channels; i > 0; --i )
    {
        GLubyte temp = pData[index1];
        pData[index1] = pData[index2];
        pData[index2] = temp;
        ++index1;
        ++index2;
    }
}
glGenTextures(1, &textureID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textureID);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  
GL_CLAMP_TO_BORDER);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  
GL_CLAMP_TO_BORDER);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texture_width,  
texture_height, 0, GL_RGB, GL_UNSIGNED_BYTE, pData);  
SOIL_free_image_data(pData);
```

2. Set up a simple pass through vertex shader that outputs the texture coordinates for texture lookup in the fragment shader, as given in the `ImageLoader` recipe of *Chapter 1*.

```
void main()  
{  
    gl_Position = vec4(vVertex*2.0-1, 0, 1);  
    vUV = vVertex;  
}
```

3. Set up the fragment shader that first shifts the texture coordinates, performs the twirl transformation, and then converts the shifted texture coordinates back for texture lookup.

```
void main()  
{  
    vec2 uv = vUV-0.5;  
    float angle = atan(uv.y, uv.x);  
    float radius = length(uv);  
    angle+= radius*twirl_amount;  
    vec2 shifted = radius* vec2(cos(angle), sin(angle));  
    vFragColor = texture(textureMap, (shifted+0.5));  
}
```

4. Render a 2D screen space quad and apply the two shaders as was done in the `ImageLoader` recipe in *Chapter 1*.

```
void OnRender() {  
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  
    shader.Use();  
    glUniform1f(shader("twirl_amount"), twirl_amount);  
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);  
    shader.UnUse();  
    glutSwapBuffers();  
}
```

How it works...

Twirl is a simple 2D transformation which deforms the image. In polar coordinates, this transformation is given simply as follows:

$$g(r, \theta) = f(r, \theta + r * t)$$

In this equation, t is the amount of twirl applied on the input image f . In practice, our images are a 2D function $f(x,y)$ of Cartesian coordinates. We first convert the Cartesian coordinates to polar coordinates (r,θ) by using the following transformation:

$$\begin{aligned} \theta &= \arctan(y, x) \\ r &= \sqrt{x^*x + y^*y} \end{aligned}$$

Here, x and y are the two Cartesian coordinates. In the fragment shader, we first offset the texture coordinates so that the origin is at the center of the image. Next, we get the angle θ and radius r .

```
void main() {
    vec2 uv = vUV - 0.5;
    float angle = atan(uv.y, uv.x);
    float radius = length(uv);
```

We then increment the angle by the given amount, multiplied by the radius. Next, we convert the polar coordinates back to Cartesian coordinates.

```
angle += radius*twirl_amount;
vec2 shifted = radius* vec2(cos(angle), sin(angle));
```

Finally, we offset the texture coordinates back to the original position. The transformed texture coordinates are then used for texture lookup.

```
vFragColor = texture(textureMap, (shifted+0.5));
}
```

There's more...

The demo application implementing this recipe shows a rendered image. Using the - and + keys, we can adjust the twirl amount as shown in the following figure:



Since the texture clamping mode was set to `GL_CLAMP_TO_BORDER`, the out of image pixels get the black color. In this recipe, we applied the twirl effect to the whole image. As an exercise, we invite the reader to limit the twirl to a specific zone within the image; for example, within a radius of, say, 150 pixels from the center of image. Hint: You can constrain the radius using the given pixel distance.

Rendering a skybox using static cube mapping

This recipe will show how to render a skybox object using static cube mapping. Cube mapping is a simple technique for generating a surrounding environment. There are several methods, such as sky dome, which uses a spherical geometry; skybox, which uses a cubical geometry; and skyplane, which uses a planar geometry. For this recipe, we will focus on skyboxes using the static cube mapping approach. The cube mapping process needs six images that are placed on each face of a cube. The skybox is a very large cube that moves with the camera but does not rotate with it.

Getting ready

The code for this recipe is contained in the `Chapter3/Skybox` directory.

How to do it...

Let us get started with the recipe as follows:

1. Set up the vertex array and vertex buffer objects to store a unit cube geometry.
2. Load the skybox images using an image loading library, such as SOIL.

```
int texture_widths[6];
int texture_heights[6];
int channels[6];
GLubyte* pData[6];
cout<<"Loading skybox images: ... "<<endl;
for(int i=0;i<6;i++) {
    cout<<"\tLoading: "<<texture_names[i]<< " ... ";
    pData[i] = SOIL_load_image(texture_names[i],
        &texture_widths[i], &texture_heights[i], &channels[i],
        SOIL_LOAD_AUTO);
    cout<<"done."<<endl;
}
```

3. Generate a cubemap OpenGL texture object and bind the six loaded images to the GL_TEXTURE_CUBE_MAP texture targets. Also make sure that the image data loaded by the SOIL library is deleted after the texture data has been stored into the OpenGL texture.

```
glGenTextures(1, &skyboxTextureID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_CUBE_MAP, skyboxTextureID);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
    GL_CLAMP_TO_EDGE);
 GLint format = (channels[0]==4)?GL_RGBA:GL_RGB;

for(int i=0;i<6;i++) {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
        format, texture_widths[i], texture_heights[i], 0, format,
        GL_UNSIGNED_BYTE, pData[i]);
    SOIL_free_image_data(pData[i]);
}
```

4. Set up a vertex shader (see Chapter3/Skybox/shaders/skybox.vert) that outputs the vertex's object space position as the texture coordinate.

```
smooth out vec3 uv;
void main()
{
    gl_Position = MVP*vec4(vVertex, 1);
    uv = vVertex;
}
```

5. Add a cubemap sampler to the fragment shader. Use the texture coordinates output from the vertex shader to sample the cubemap sampler object in the fragment shader (see Chapter3/Skybox/shaders/skybox.frag).

```
layout(location=0) out vec4 vFragColor;
uniform samplerCube cubeMap;
smooth in vec3 uv;
void main()
{
    vFragColor = texture(cubeMap, uv);
}
```

How it works...

There are two parts of this recipe. The first part, which loads an OpenGL cubemap texture, is self explanatory. We load the six images and bind these to an OpenGL cubemap texture target. There are six cubemap texture targets corresponding to the six sides of a cube. These targets are `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Z`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`, and `GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`. Since their identifiers are linearly generated, we offset the target by the loop variable to move to the next cubemap texture target in the following code:

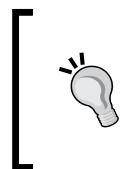
```
for(int i=0;i<6;i++) {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
    format, texture_widths[i], texture_heights[i], 0, format,
    GL_UNSIGNED_BYTE, pData[i]);
    SOIL_free_image_data(pData[i]);
}
```

The second part is the shader responsible for sampling the cubemap texture. This work is carried out in the fragment shader (Chapter3/Skybox/shaders/skybox.frag). In the rendering code, we set the skybox shader and then render the skybox, passing it the MVP matrix, which is obtained as follows:

```
glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f,
dist));
glm::mat4 Rx = glm::rotate(glm::mat4(1), rX, glm::vec3(1.0f,
0.0f, 0.0f));
```

```
glm::mat4 MV = glm::rotate(Rx, rY, glm::vec3(0.0f, 1.0f, 0.0f));
glm::mat4 S      = glm::scale(glm::mat4(1),glm::vec3(1000.0));
glm::mat4 MVP   = P*MV*S;
skybox->Render( glm::value_ptr(MVP) );
```

To sample the correct location in the cubemap texture we need a vector. This vector can be obtained from the object space vertex positions that are passed to the vertex shader. These are passed through the `uv` output attribute to the fragment shader.



In this recipe, we scaled a unit cube. While it is not necessary to have a unit cube, one thing that we have to be careful with is that the size of the cube after scaling should not be greater than the far clip plane distance. Otherwise, our skybox will be clipped.

There's more...

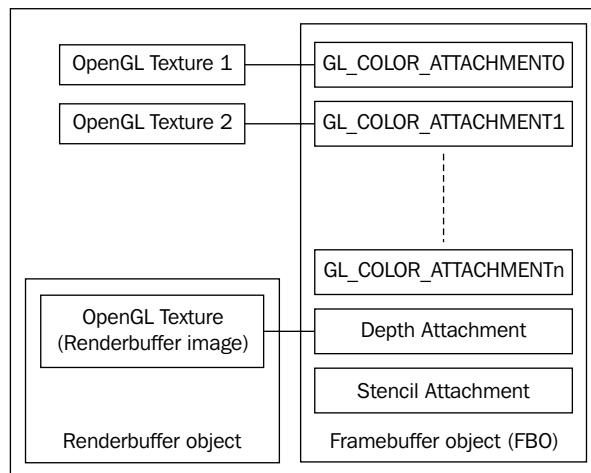
The demo application implementing this recipe shows a statically cube mapped skybox which can be looked around by dragging the left mouse button. This gives a surrounded environment feeling to the user as shown in the following figure:



Implementing a mirror with render-to-texture using FBO

We will now use the FBO to render a mirror object on the screen. In a typical offscreen rendering OpenGL application, we set up the FBO first, by calling the `glGenFramebuffers` function and passing it the number of FBOs desired. The second parameter stores the returned identifier. After the FBO object is generated, it has to be bound to the `GL_FRAMEBUFFER`, `GL_DRAW_FRAMEBUFFER`, or `GL_READ_FRAMEBUFFER` target. Following this call, the texture to be bound to the FBOs color attachment is attached by calling the `glFramebufferTexture2D` function.

There can be more than one color attachment on an FBO. The maximum number of color attachments supported on any GPU can be queried using the `GL_MAX_COLOR_ATTACHMENTS` field. The type and dimension of the texture has to be specified and it is not necessary to have the same size as the screen. However, all color attachments on the FBO must have the same dimensions. At any time, only a single FBO can be bound for a drawing operation and similarly, only one can be bound for a reading operation. In addition to the color attachment, there are also depth and stencil attachments on an FBO. The following image shows the different attachment points on an FBO:



If depth testing is required, a render buffer is also generated and bound by calling `glGenRenderbuffers` followed by the `glBindRenderbuffer` function. For render buffers, the depth buffer's data type and its dimensions have to be specified. After all these steps, the render buffer is attached to the frame buffer by calling the `glFramebufferRenderbuffer` function.

After the setup of the frame buffer and render buffer objects, the frame buffer completeness status has to be checked by calling `glCheckFramebufferStatus` by passing it the framebuffer target. This ensures that the FBO setup is correct. The function returns the status as an identifier. If this returned value is anything other than `GL_FRAMEBUFFER_COMPLETE`, the FBO setup is unsuccessful.



Make sure to check the `Framebuffer` status after the `Framebuffer` is bound.



Similar to other OpenGL objects, we must delete the `framebuffer` and the `renderbuffer` objects and any texture objects used for offscreen rendering after they are no more needed, by calling the `glDeleteFramebuffers` and `glDeleteRenderbuffers` functions. These are the typical steps needed to enable offscreen rendering using FBO objects in modern OpenGL.

Getting ready

The code for this recipe is contained in the `Chapter3/MirrorUsingFBO` directory.

How to do it...

Let us get started with the recipe as follows:

1. Initialize the `framebuffer` and `renderbuffer` objects' color and depth attachments respectively. The render buffer is required if we need depth testing for the offscreen rendering, and the depth precision is specified using the `glRenderbufferStorage` function.

```
glGenFramebuffers(1, &fboID);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboID);
glGenRenderbuffers(1, &rbID);
glBindRenderbuffer(GL_RENDERBUFFER, rbID);
glRenderbufferStorage(GL_RENDERBUFFER,
GL_DEPTH_COMPONENT32, WIDTH, HEIGHT);
```

2. Generate the offscreen texture on which FBO will render to. The last parameter of `glTexImage2D` is `NULL`, which tells OpenGL that we do not have any content yet, please provide a new block of GPU memory which gets filled when the FBO is used as a render target.

```
glGenTextures(1, &renderTextureID);
glBindTexture(GL_TEXTURE_2D, renderTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
L_REPEAT);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
GL_NEAREST);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, WIDTH, HEIGHT, 0,  
GL_BGRA, GL_UNSIGNED_BYTE, NULL);
```

3. Attach Renderbuffer to the bound Framebuffer object and check for Framebuffer completeness.

```
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,  
GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D, renderTextureID, 0);  
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,  
GL_DEPTH_ATTACHMENT,GL_RENDERBUFFER, rbID);  
GLuint status = glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);  
if(status==GL_FRAMEBUFFER_COMPLETE) {  
printf("FBO setup succeeded.");  
} else {  
printf("Error in FBO setup.");  
}
```

4. Unbind the Framebuffer object as follows:

```
glBindTexture(GL_TEXTURE_2D, 0);  
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

5. Create a quad geometry to act as a mirror:

```
mirror = new CQuad(-2);
```

6. Render the scene normally from the point of view of camera. Since the unit color cube is rendered at origin, we translate it on the Y axis to shift it up in Y axis which effectively moves the unit color cube in Y direction so that the unit color cube's image can be viewed completely in the mirror.

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);  
grid->Render(glm::value_ptr(MVP));  
localR[3][1] = 0.5;  
cube->Render(glm::value_ptr(P*MV*localR));
```

7. Store the current modelview matrix and then change the modelview matrix such that the camera is placed at the mirror object position. Also make sure to laterally invert this modelview matrix by scaling by -1 on the X axis.

```
glm::mat4 oldMV = MV;  
glm::vec3 target;  
glm::vec3 V = glm::vec3(-MV[2][0], -MV[2][1], -MV[2][2]);  
glm::vec3 R = glm::reflect(V, mirror->normal);  
MV = glm::lookAt(mirror->position, mirror->position + R,  
glm::vec3(0,1,0));  
MV = glm::scale(MV, glm::vec3(-1,1,1));
```

- Bind the FBO, set up the FBO color attachment for `Drawbuffer (GL_COLOR_ATTACHMENT0)` or any other attachment to which texture is attached, and clear the FBO. The `glDrawBuffer` function enables the code to draw to a specific color attachment on the FBO. In our case, there is a single color attachment so we set it as the draw buffer.

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

- Set the modified modelview matrix and render the scene again. Also make sure to only render from the shiny side of the mirror.

```
if(glm::dot(V,mirror->normal)<0) {
    grid->Render(glm::value_ptr(P*MV));
    cube->Render(glm::value_ptr(P*MV*localR));
}
```

- Unbind the FBO and restore the default `Drawbuffer (GL_BACK_LEFT)`.

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
```



Note that there are several aliases for the back buffer. The real back buffer is `GL_BACK_LEFT`, which is also referred by the `GL_BACK` alias. The default `Framebuffer` has up to four color buffers, namely `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, and `GL_BACK_RIGHT`. If stereo rendering is not active, then only the left buffers are active, that is, `GL_FRONT_LEFT` (the active front color buffer) and `GL_BACK_LEFT` (the active back color buffer).

- Finally render the mirror quad at the saved modelview matrix.

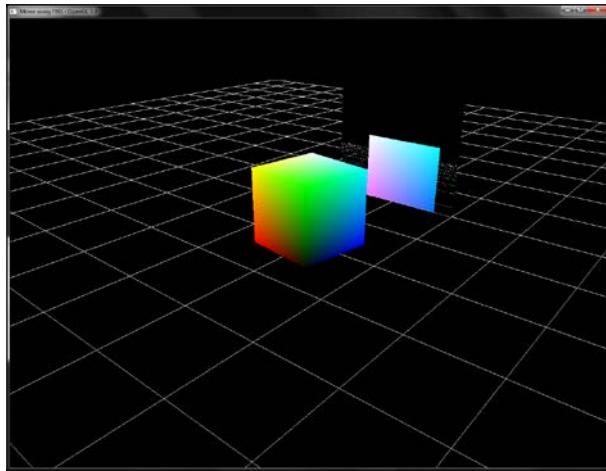
```
MV = oldMV;
glBindTexture(GL_TEXTURE_2D, renderTextureID);
mirror->Render(glm::value_ptr(P*MV));
```

How it works...

The mirror algorithm used in the recipe is very simple. We first get the view direction vector (V) from the viewing matrix. We reflect this vector on the normal of the mirror (N). Next, the camera position is moved to the place behind the mirror. Finally, the mirror is scaled by -1 on the X axis. This ensures that the image is laterally inverted as in a mirror. Details of the method are covered in the reference in the See also section.

There's more...

Details of the `Framebuffer` object can be obtained from the `Framebuffer` object specifications (see the *See also* section). The output from the demo application implementing this recipe is as follows:



See also

- ▶ The Official OpenGL registry-`Framebuffer` object specifications can be found at http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt.
- ▶ *OpenGL Superbible, Fifth Edition, Chapter 8*, pages 354-358, Richard S. Wright, Addison-Wesley Professional
- ▶ FBO tutorial by Song Ho Ahn: http://www.songho.ca/opengl/gl_fbo.html

Rendering a reflective object using dynamic cube mapping

Now we will see how to use dynamic cube mapping to render a real-time scene to a cubemap render target. This allows us to create reflective surfaces. In modern OpenGL, offscreen rendering (also called render-to-texture) functionality is exposed through FBOs.

Getting ready

In this recipe, we will render a box with encircling particles. The code is contained in the `Chapter3/DynamicCubemap` directory.

How to do it...

Let us get started with the recipe as follows:

1. Create a cubemap texture object.

```
glGenTextures(1, &dynamicCubeMapID);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_CUBE_MAP, dynamicCubeMapID);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
 GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
 GL_CLAMP_TO_EDGE);
 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
 GL_CLAMP_TO_EDGE);
for (int face = 0; face < 6; face++) {
    glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + face, 0,
    GL_RGBA, CUBEMAP_SIZE, CUBEMAP_SIZE, 0, GL_RGBA, GL_FLOAT,
    NULL);
}
```

2. Set up an FBO with the cubemap texture as an attachment.

```
glGenFramebuffers(1, &fboID);
 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboID);
 glGenRenderbuffers(1, &rboID);
 glBindRenderbuffer(GL_RENDERBUFFER, rboID);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
 CUBEMAP_SIZE, CUBEMAP_SIZE);
 glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
 GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, fboID);
 glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
 GL_COLOR_ATTACHMENT0, GL_TEXTURE_CUBE_MAP_POSITIVE_X,
 dynamicCubeMapID, 0);
GLenum status =
 glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE) {
    cerr<<"Frame buffer object setup error."<<endl;
    exit(EXIT_FAILURE);
} else {
    cerr<<"FBO setup successfully."<<endl;
}
```

3. Set the viewport to the size of the offscreen texture and render the scene six times without the reflective object to the six sides of the cubemap using FBO.

```
glViewport(0,0,CUBEMAP_SIZE,CUBEMAP_SIZE);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboID);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0,GL_TEXTURE_CUBE_MAP_POSITIVE_X,
dynamicCubeMapID, 0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glm::mat4 MV1 = glm::lookAt(glm::vec3(0),glm::vec3(1,0,0),glm::v
ec3(0,-
1,0));
DrawScene( MV1*T, PcubeMap);

glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
dynamicCubeMapID, 0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glm::mat4 MV2 = glm::lookAt(glm::vec3(0),glm::vec3(-1,0,0),
glm::vec3(0,-1,0));
DrawScene( MV2*T, PcubeMap);

...//similar for rest of the faces
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

4. Restore the viewport and the modelview matrix, and render the scene normally.

```
glViewport(0,0,WIDTH,HEIGHT);
DrawScene(MV, P);
```

5. Set the cubemap shader and then render the reflective object.

```
glBindVertexArray(sphereVAOID);
cubemapShader.Use();
T = glm::translate(glm::mat4(1), p);
glUniformMatrix4fv(cubemapShader("MVP"), 1, GL_FALSE,
glm::value_ptr(P*(MV*T)));
glUniform3fv(cubemapShader("eyePosition"), 1,
glm::value_ptr(eyePos));
glDrawElements(GL_TRIANGLES,indices.size(),
GL_UNSIGNED_SHORT,0);
cubemapShader.UnUse();
```

How it works...

Dynamic cube mapping renders the scene six times from the reflective object using six cameras at the reflective object's position. For rendering to the cubemap texture, an FBO is used with a cubemap texture attachment. The cubemap texture's `GL_TEXTURE_CUBE_MAP_POSITIVE_X` target is bound to the `GL_COLOR_ATTACHMENT0` color attachment of the FBO. The last parameter of `glTexImage2D` is `NULL` since this call just allocates the memory for offscreen rendering and the real data will be populated when the FBO is set as the render target.

The scene is then rendered to the cubemap texture without the reflective object by placing six cameras at the reflective object's position in the six directions. The cubemap projection matrix (`Pcubemap`) is given a 90 degree fov.

```
Pcubemap = glm::perspective(90.0f, 1.0f, 0.1f, 1000.0f);
```

This renders the scene into the cubemap texture. For each side, a new MVP matrix is obtained by multiplying the new MV matrix (obtained by using `glm::lookAt` function). This is repeated for all six sides of the cube. Next, the scene is rendered normally and the reflective object is finally rendered using the generated cubemap to render the reflective environment. Rendering each frame six times into an offscreen target hinders performance, especially if there are complex objects in the world. Therefore this technique should be used with caution.

The cubemap vertex shader outputs the object space vertex positions and normals.

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;
uniform mat4 MVP;
smooth out vec3 position;
smooth out vec3 normal;
void main() {
    position = vVertex;
    normal = vNormal;
    gl_Position = MVP*vec4(vVertex, 1);
}
```

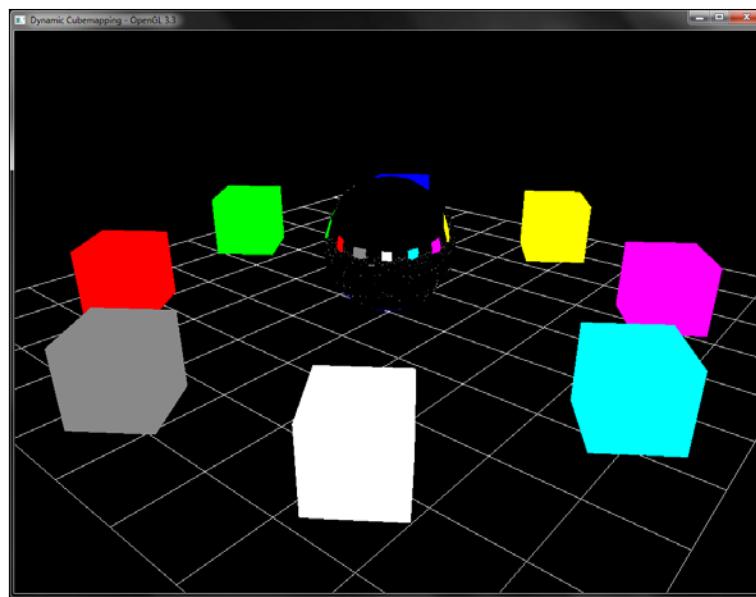
The cubemap fragment shader uses the object space vertex positions to determine the view vector. The reflection vector is then obtained by reflecting the view vector at the object space normal.

```
#version 330 core
layout(location=0) out vec4 vFragColor;
uniform samplerCube cubeMap;
smooth in vec3 position;
smooth in vec3 normal;
uniform vec3 eyePosition;
```

```
void main() {  
    vec3 N = normalize(normal);  
    vec3 V = normalize(position-eyePosition);  
    vFragColor = texture(cubeMap, reflect(V,N));  
}
```

There's more...

The demo application implementing this recipe renders a reflective sphere with eight cubes pulsating around it, as shown in the following figure:



In this recipe, we could also use layered rendering by using the geometry shader to output to a different `Framebuffer` object layer. This can be achieved by outputting to the appropriate `gl_Layer` attribute from the geometry shader and setting the appropriate viewing transformation. This is left as an exercise for the reader.

See also

- ▶ Check the OpenGL wiki page at http://www.opengl.org/wiki/Geometry_Shader#Layered_rendering
- ▶ FBO tutorial by Song Ho Ahn: http://www.songho.ca/opengl/gl_fbo.html

Implementing area filtering (sharpening/blurring/embossing) on an image using convolution

We will now see how to do area filtering, that is, 2D image convolution to implement effects like sharpening, blurring, and embossing. There are several ways to achieve image convolution in the spatial domain. The simplest approach is to use a loop that iterates through a given image window and computes the sum of products of the image intensities with the convolution kernel. The more efficient method, as far as the implementation is concerned, is separable convolution which breaks up the 2D convolution into two 1D convolutions. However, this approach requires an additional pass.

Getting ready

This recipe is built on top of the image loading recipe discussed in the first chapter. If you feel a bit lost, we suggest skimming through it to be on page with us. The code for this recipe is contained in the `Chapter3/Convolution` directory. For this recipe, most of the work takes place in the fragment shader.

How to do it...

Let us get started with the recipe as follows:

1. Create a simple pass-through vertex shader that outputs the clip space position and the texture coordinates which are to be passed into the fragment shader for texture lookup.

```
#version 330 core
in vec2 vVertex;
out vec2 vUV;
void main()
{
    gl_Position = vec4(vVertex*2.0-1,0,1);
    vUV = vVertex;
}
```

2. In the fragment shader, we declare a constant array called `kernel` which stores our convolution kernel. Changing the convolution kernel values dictates the output of convolution. The default kernel sets up a sharpening convolution filter. Refer to `Chapter3/Convolution/shaders/shader_convolution.frag` for details.

```
const float kernel[] = float[9] (-1,-1,-1,
                                 -1, 8,-1,
                                 -1,-1,-1);
```

3. In the fragment shader, we run a nested loop that loops through the current pixel's neighborhood and multiplies the `kernel` value with the current pixel's value. This is continued in an $n \times n$ neighborhood, where n is the width/height of the `kernel`.

```
for(int j=-1;j<=1;j++) {
    for(int i=-1;i<=1;i++) {
        color += kernel[index--] *
            texture(textureMap, vUV+(vec2(i,j)*delta));
    }
}
```

4. After the nested loops, we divide the color value with the total number of values in the `kernel`. For a 3×3 `kernel`, we have nine values. Finally, we add the convolved color value to the current pixel's value.

```
color/=9.0;
vFragColor = color + texture(textureMap, vUV);
```

How it works...

For a 2D digital image $f(x,y)$, the processed image $g(x,y)$, after the convolution operation with a kernel $h(x,y)$, is defined mathematically as follows:

$$g(x,y) = \sum_{j=y-w}^{y+w} \sum_{i=x-w}^{x+w} f(i,j) * h(x-i, y-j)$$

For each pixel, we simply sum the product of the current image pixel value with the corresponding coefficient in the kernel in the given neighborhood. For details about the kernel coefficients, we refer the reader to any standard text on digital image processing, like the one given in the *See also* section.

The overall algorithm works like this. We set up our FBO for offscreen rendering. We render our image on the offscreen render target of the FBO, instead of the back buffer. Now the FBO attachment stores our image. Next, we set the output from the first step (that is, the rendered image on the FBO attachment) as input to the convolution shader in the second pass. We render a full-screen quad on the back buffer and apply our convolution shader to it. This performs convolution on the input image. Finally, we swap the back buffer to show the result on the screen.

After the image is loaded and an OpenGL texture has been generated, we render a screen-aligned quad. This allows the fragment shader to run for the whole screen. In the fragment shader, for the current fragment, we iterate through its neighborhood and sum the product of the corresponding entry in the kernel with the look-up value. After the loop is terminated, the sum is divided by the total number of kernel coefficients. Finally, the convolution sum is added to the current pixel's value. There are several different kinds of kernels. We list the ones we will use in this recipe in the following table.



Based on the wrapping mode set for the texture, for example, `GL_CLAMP` or `GL_REPEAT`, the convolution result will be different. In case of the `GL_CLAMP` wrapping mode, the pixels out of the image are not considered, whereas, in case of the `GL_REPEAT` wrapping mode, the out of the image pixel information is obtained from the pixel at the wrapping position.

Effect	Kernel matrix
Sharpening	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
Blurring / Unweighted Smoothing	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
3 x 3 Gaussian blur	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 5 & 1 \\ 0 & 1 & 0 \end{bmatrix}$
Emboss north-west direction	$\begin{bmatrix} -4 & -4 & 0 \\ -4 & 12 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
Emboss north-east direction	$\begin{bmatrix} 0 & -4 & -4 \\ 0 & 12 & -4 \\ 0 & 0 & 0 \end{bmatrix}$

Effect	Kernel matrix
Emboss south-east direction	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 12 & -4 \\ 0 & -4 & -4 \end{bmatrix}$
Emboss south-west direction	$\begin{bmatrix} 0 & 0 & 0 \\ -4 & 12 & 0 \\ -4 & -4 & 0 \end{bmatrix}$

There's more...

We just touched the topic of digital image convolution. For details, we refer the reader to the See also section. In the demo application, the user can set the required kernel and then press the Space bar key to see the filtered image output. Pressing the Space bar key once again shows the normal unfiltered image.

See also

- ▶ *Digital Image Processing, Third Edition, Rafael C. Gonzales and Richard E. Woods, Prentice Hall*
- ▶ FBO tutorial by Song Ho Ahn: http://www.songho.ca/opengl/gl_fbo.html

Implementing the glow effect

Now that we know how to perform offscreen rendering and blurring, we will put this knowledge to use by implementing the glow effect. The code for this recipe is in the Chapter3/Glow directory. In this recipe, we will render a set of points encircling a cube. Every 50 frames, four alternate points glow.

How to do it...

Let us get started with the recipe as follows:

1. Render the scene normally by rendering the points and the cube. The particle shader renders the `GL_POINTS` value (which by default, renders as quads) as circles.

```
grid->Render(glm::value_ptr(MVP)) ;
cube->Render(glm::value_ptr(MVP)) ;
glBindVertexArray(particlesVAO) ;
particleShader.Use();
```

```
glUniformMatrix4fv(particleShader("MVP"), 1, GL_FALSE,
glm::value_ptr(MVP*Rot));
glDrawArrays(GL_POINTS, 0, 8);
```

The particle vertex shader is as follows:

```
#version 330 core
layout(location=0) in vec3 vVertex;
uniform mat4 MVP;
smooth out vec4 color;
const vec4 colors[8]=vec4[8](vec4(1,0,0,1), vec4(0,1,0,1),
vec4(0,0,1,1),vec4(1,1,0,1), vec4(0,1,1,1), vec4(1,0,1,1),
vec4(0.5,0.5,0.5,1), vec4(1,1,1,1)) ;

void main() {
    gl_Position = MVP*vec4(vVertex,1);
    color = colors[gl_VertexID/4];
}
```

The particle fragment shader is as follows:

```
#version 330 core
layout(location=0) out vec4 vFragColor;

smooth in vec4 color;

void main() {
    vec2 pos = gl_PointCoord-0.5;
    if(dot(pos,pos)>0.25)
        discard;
    else
        vFragColor = color;
}
```

2. Set up a single FBO with two color attachments. The first attachment is for rendering of scene elements requiring glow and the second attachment is for blurring.

```
glGenFramebuffers(1, &fb0ID);
 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fb0ID);
 glGenTextures(2, texID);
 glBindTexture(GL_TEXTURE0);
 for(int i=0;i<2;i++) {
    glBindTexture(GL_TEXTURE_2D, texID[i]);
    glTexParameterf(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER,GL_LINEAR)
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  
GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  
GL_CLAMP_TO_EDGE);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,  
RENDER_TARGET_WIDTH, RENDER_TARGET_HEIGHT, 0,  
GL_RGBA,GL_UNSIGNED_BYTE, NULL);  
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,  
GL_COLOR_ATTACHMENT0+i,GL_TEXTURE_2D, texID[i], 0);  
}  
GLenum status =  
glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER);  
if(status != GL_FRAMEBUFFER_COMPLETE) {  
    cerr<<"Frame buffer object setup error."<<endl;  
    exit(EXIT_FAILURE);  
} else {  
    cerr<<"FBO set up successfully."<<endl;  
}  
 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
```

- Bind FBO, set the viewport to the size of the attachment texture, set Drawbuffer to render to the first color attachment (GL_COLOR_ATTACHMENT0), and render the part of the scene which needs glow.

```
 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fboID);  
glViewport(0,0,RENDER_TARGET_WIDTH,RENDER_TARGET_HEIGHT);  
glDrawBuffer(GL_COLOR_ATTACHMENT0);  
glClear(GL_COLOR_BUFFER_BIT);  
glDrawArrays(GL_POINTS, offset, 4);  
particleShader.UnUse();
```

- Set Drawbuffer to render to the second color attachment (GL_COLOR_ATTACHMENT1) and bind the FBO texture attached to the first color attachment. Set the blur shader by convolving with a simple unweighted smoothing filter.

```
 glDrawBuffer(GL_COLOR_ATTACHMENT1);  
 glBindTexture(GL_TEXTURE_2D, texID[0]);
```

- Render a screen-aligned quad and apply the blur shader to the rendering result from the first color attachment of the FBO. This output is written to the second color attachment.

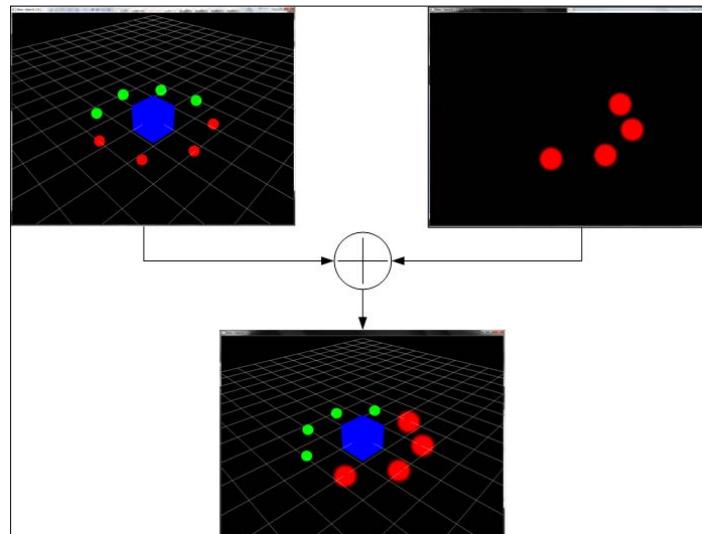
```
blurShader.Use();  
	glBindVertexArray(quadVAOID);  
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

6. Disable FBO rendering, reset the default drawbuffer (GL_BACK_LEFT) and viewport, bind the texture attached to the FBO's second color attachment, draw a screen-aligned quad, and blend the blur output to the existing scene using additive blending.

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
glBindTexture(GL_TEXTURE_2D, texID[1]);
glViewport(0, 0, WIDTH, HEIGHT);
 glEnable(GL_BLEND);
 glBlendFunc(GL_ONE, GL_ONE);
 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
 glBindVertexArray(0);
blurShader.UnUse();
glDisable(GL_BLEND);
```

How it works...

The glow effect works by first rendering the candidate elements of the scene for glow into a separate render target. After rendering, a smoothing filter is applied on the rendered image containing the elements requiring glow. The smoothed output is then additively blended with the current rendering on the frame buffer, as shown in the following figure:



Note that we could also enable blending in the fragment shader. Assuming that the two images to be blended are bound to their texture units and their shader samplers are `texture1` and `texture2`, the additive blending shader code will be like this:

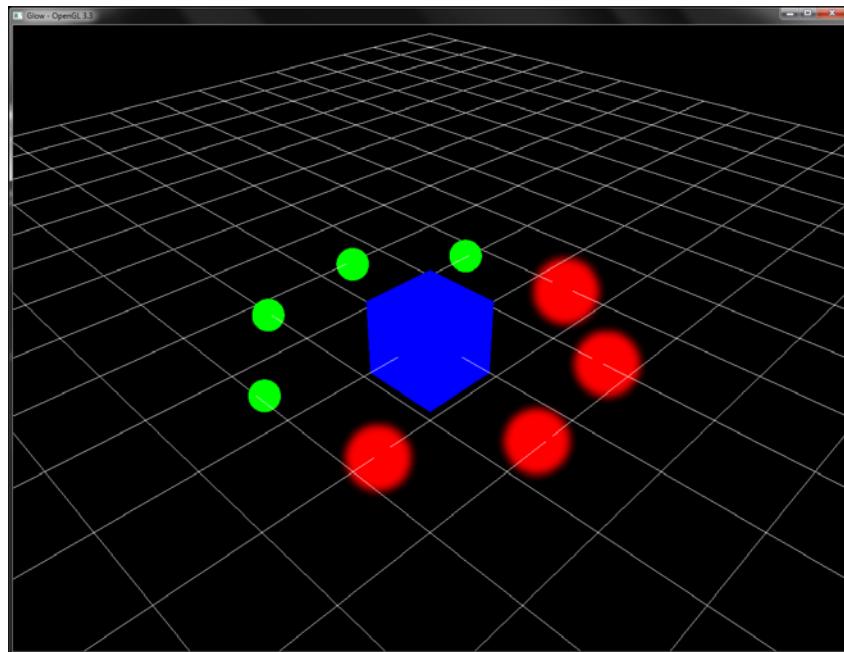
```
#version 330 core
uniform sampler2D texture1;
uniform sampler2D texture2;
layout(location=0) out vec4 vFragColor;
smooth in vec2 vUV;
void main() {
    vec4 color1 = texture(texture1, vUV);
    vec4 color2 = texture(texture2, vUV);
    vFragColor = color1+color2;
}
```

Additionally, we can also apply separable convolution, but that requires two passes. The process requires three color attachments. We first render the scene normally on the first color attachment while the glow effect objects are rendered on the second color attachment. The third color attachment is then set as the render target while the second color attachment acts as input. A full-screen quad is then rendered with the vertical smoothing shader which simply iterates through a row of pixels. This vertically smoothed result is written to the third color attachment.

The second color attachment is then set as output while the output results from the vertical smoothing pass (which was written to the third color attachment) is set as input. The horizontal smoothing shader is then applied on a column of pixels which smoothes the entire image. The image is then rendered to the second color attachment. Finally, the blend shader combines the result from the first color attachment with the result from the second color attachment. Note that the same effect could be carried out by using two separate FBOs: a rendering FBO and a filtering FBO, which gives us more flexibility as we can down sample the filtering result to take advantage of hardware linear filtering. This technique has been used in the *Implementing variance shadow mapping* recipe in Chapter 4, Lights and Shadows.

There's more...

The demo application for this recipe shows a simple unit cube encircled by eight points. The first four points are rendered in red and the latter four are rendered in green. The application applies glow to the first four points. After every 50 frames, the glow shifts to the latter four points and so on for the lifetime of the application. The output result from the application is shown in the following figure:



See also

- ▶ Glow sample in NVIDIA OpenGL SDK v10
- ▶ FBO tutorial by Song Ho Ahn: http://www.songho.ca/opengl/gl_fbo.html

4

Lights and Shadows

In this chapter, we will cover:

- ▶ Implementing per-vertex and per-fragment point lighting
- ▶ Implementing per-fragment directional light
- ▶ Implementing per-fragment point light with attenuation
- ▶ Implementing per-fragment spot light
- ▶ Implementing shadow mapping with FBO
- ▶ Implementing shadow mapping with percentage closer filtering (PCF)
- ▶ Implementing variance shadow mapping

Introduction

Similar to how the real world would be dark without lights, we require simulated lights to see in our virtual worlds. Visual applications will be incomplete without the presence of lights. There are several kinds of lights; for example, point lights, directional lights, spot lights, and so on. Each of these have some common properties, for example, light position. In addition, they have some specific properties, such as spot direction and spot exponent for spot lights. We will cover all of these light types as well as how to implement them in the vertex shader stage or the fragment shader stage.

Although we can leave the lights to just light the environment, our visual system will start to find problems with such a setting. This is because our eyes are not used to seeing objects lit but casting no shadows. In addition, without shadows, it is very difficult to judge how near or far an object is to the other. Therefore, we detail several shadow generation techniques varying from classic depth shadow mapping to more advanced variance shadow mapping. All of these will be implemented in OpenGL v3.3 and all implementation details will be given to enable the reader to implement the technique on their own.

Implementing per-vertex and per-fragment point lighting

To give more realism to 3D graphic scenes, we add lighting. In OpenGL's fixed function pipeline, per-vertex lighting is provided (which is deprecated in OpenGL v3.3 and above). Using shaders, we can not only replicate the per-vertex lighting of fixed function pipeline but also go a step further by implementing per-fragment lighting. The per-vertex lighting is also known as **Gouraud shading** and the per-fragment shading is known as **Phong shading**. So, without further ado, let's get started.

Getting started

In this recipe, we will render many cubes and a sphere. All of these objects are generated and stored in the buffer objects. For details, refer to the `CreateSphere` and `CreateCube` functions in `Chapter4/PerVertexLighting/main.cpp`. These functions generate both vertex positions as well as per-vertex normals, which are needed for the lighting calculations. All of the lighting calculations take place in the vertex shader of the per-vertex lighting recipe (`Chapter4/PerVertexLighting/`), whereas, for the per-fragment lighting recipe (`Chapter4/PerFragmentLighting/`) they take place in the fragment shader.

How to do it...

Let us start our recipe by following these simple steps:

1. Set up the vertex shader that performs the lighting calculation in the view/eye space. This generates the color after the lighting calculation.

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;
uniform mat4 MVP;
uniform mat4 MV;
uniform mat3 N;
uniform vec3 light_position; //light position in object
space
uniform vec3 diffuse_color;
uniform vec3 specular_color;
uniform float shininess;
smooth out vec4 color;
const vec3 vEyeSpaceCameraPosition = vec3(0,0,0);
void main()
{
    vec4 vEyeSpaceLightPosition = MV*vec4(light_position,1);
```

```

vec4 vEyeSpacePosition = MV*vec4(vVertex,1);
vec3 vEyeSpaceNormal = normalize(N*vNormal);
vec3 L = normalize(vEyeSpaceLightPosition.xyz -
vEyeSpacePosition.xyz);
vec3 V = normalize(vEyeSpaceCameraPosition.xyz-
vEyeSpacePosition.xyz);
vec3 H = normalize(L+V);
float diffuse = max(0, dot(vEyeSpaceNormal, L));
float specular = max(0, pow(dot(vEyeSpaceNormal, H),
shininess));
color = diffuse*vec4(diffuse_color,1) +
specular*vec4(specular_color, 1);
gl_Position = MVP*vec4(vVertex,1);
}

```

2. Set up a fragment shader which, inputs the shaded color from the vertex shader interpolated by the rasterizer, and set it as the current output color.

```

#version 330 core
layout(location=0) out vec4 vFragColor;
smooth in vec4 color;
void main() {
    vFragColor = color;
}

```

3. In the rendering code, set the shader and render the objects by passing their modelview/projection matrices to the shader as shader uniforms.

```

shader.Use();
glBindVertexArray(cubeVAOID);
for(int i=0;i<8;i++)
{
    float theta = (float)(i/8.0f*2*M_PI);
    glm::mat4 T = glm::translate(glm::mat4(1),
    glm::vec3(radius*cos(theta), 0.5,radius*sin(theta)));
    glm::mat4 M = T;
    glm::mat4 MV = View*M;
    glm::mat4 MVP = Proj*MV;
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(MVP));
    glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
    glm::value_ptr(MV));
    glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
    glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
    glUniform3fv(shader("diffuse_color"),1, &(colors[i].x));
    glUniform3fv(shader("light_position"),1,&(lightPosOS.x));
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
}

```

```
    }
    glBindVertexArray(sphereVAOID);
    glm::mat4 T = glm::translate(glm::mat4(1),
    glm::vec3(0,1,0));
    glm::mat4 M = T;
    glm::mat4 MV = View*M;
    glm::mat4 MVP = Proj*MV;
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(MVP));
    glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
    glm::value_ptr(MV));
    glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
    glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
    glUniform3f(shader("diffuse_color"), 0.9f, 0.9f, 1.0f);
    glUniform3fv(shader("light_position"), 1, &(lightPosOS.x));
    glDrawElements(GL_TRIANGLES, totalSphereTriangles,
    GL_UNSIGNED_SHORT, 0);
    shader.UnUse();
    glBindVertexArray(0);
    grid->Render(glm::value_ptr(Proj*View));
```

How it works...

We can perform the lighting calculations in any coordinate space we wish, that is, object space, world space, or eye/view space. Similar to the lighting in the fixed function OpenGL pipeline, in this recipe we also do our calculations in the eye space. The first step in the vertex shader is to obtain the vertex position and light position in the eye space. This is done by multiplying the current vertex and light position with the modelview (MV) matrix.

```
vec4 vEyeSpaceLightPosition = MV*vec4(light_position,1);
vec4 vEyeSpacePosition = MV*vec4(vVertex,1);
```

Similarly, we transform the per-vertex normals to eye space, but this time we transform them with the inverse transpose of the modelview matrix, which is stored in the normal matrix (N).

```
vec3 vEyeSpaceNormal = normalize(N*vNormal);
```



In the OpenGL versions prior to v3.0, the normal matrix was stored in the `gl_NormalMatrix` shader uniform, which is the inverse transpose of the modelview matrix. Compared to positions, normals are transformed differently since the scaling transformation may modify the normals in such a way that the normals are not normalized anymore. Multiplying the normals with the inverse transpose of the modelview matrix ensures that the normals are only rotated based on the given matrix, maintaining their unit length.

Next, we obtain the vector from the position of the light in eye space to the position of the vertex in eye space, and do a dot product of this vector with the eye space normal. This gives us the diffuse component.

```
vec3 L = normalize(vEyeSpaceLightPosition.xyz-
vEyeSpacePosition.xyz);
float diffuse = max(0, dot(vEyeSpaceNormal, L));
```

We also calculate two additional vectors, the view vector (V) and the half-way vector (H) between the light and the view vector.

```
vec3 V = normalize(vEyeSpaceCameraPosition.xyz-
vEyeSpacePosition.xyz);
vec3 H = normalize(L+V);
```

These are used for specular component calculation in the **Blinn Phong lighting model**. The specular component is then obtained using $\text{pow}(\text{dot}(N, H), \sigma)$, where σ is the shininess value; the larger the shininess, the more focused the specular.

```
float specular = max(0, pow(dot(vEyeSpaceNormal, H), shininess));
```

The final color is then obtained by multiplying the diffuse value with the diffuse color and the specular value with the specular color.

```
color = diffuse*vec4(diffuse_color, 1) +
specular*vec4(specular_color, 1);
```

The fragment shader in the per-vertex lighting simply outputs the per-vertex color interpolated by the rasterizer as the current fragment color.

```
smooth in vec4 color;
void main() {
    vFragColor = color;
}
```

Alternatively, if we move the lighting calculations to the fragment shader, we get a more pleasing rendering result at the expense of increased processing overhead. Specifically, we transform the per-vertex position, light position, and normals to eye space in the vertex shader, shown as follows:

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;
uniform mat4 MVP;
uniform mat4 MV;
uniform mat3 N;
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;
```

```
void main()
{
    vEyeSpacePosition = (MV*vec4(vVertex,1)).xyz;
    vEyeSpaceNormal = N*vNormal;
    gl_Position = MVP*vec4(vVertex,1);
}
```

In the fragment shader, the rest of the calculation, including the diffuse and specular component contributions, is carried out.

```
#version 330 core
layout(location=0) out vec4 vFragColor;
uniform vec3 light_position; //light position in object space
uniform vec3 diffuse_color;
uniform vec3 specular_color;
uniform float shininess;
uniform mat4 MV;
smooth in vec3 vEyeSpaceNormal;
smooth in vec3 vEyeSpacePosition;
const vec3 vEyeSpaceCameraPosition = vec3(0,0,0);

void main() {
    vec3 vEyeSpaceLightPosition=(MV*vec4(light_position,1)).xyz;
    vec3 N = normalize(vEyeSpaceNormal);
    vec3 L = normalize(vEyeSpaceLightPosition-vEyeSpacePosition);
    vec3 V = normalize(vEyeSpaceCameraPosition.xyz-
                        vEyeSpacePosition.xyz);
    vec3 H = normalize(L+V);
    float diffuse = max(0, dot(N, L));
    float specular = max(0, pow(dot(N, H), shininess));
    vFragColor = diffuse*vec4(diffuse_color,1) +
                 specular*vec4(specular_color, 1);
}
```

We will now dissect the per-fragment lighting fragment shader line-by-line. We first calculate the light position in eye space. Then we calculate the vector from the light to the vertex in eye space. We also calculate the view vector (V) and the half way vector (H).

```
vec3 vEyeSpaceLightPosition = (MV * vec4(light_position,1)).xyz;
vec3 N = normalize(vEyeSpaceNormal);
vec3 L = normalize(vEyeSpaceLightPosition-vEyeSpacePosition);
vec3 V = normalize(vEyeSpaceCameraPosition.xyz-
                    vEyeSpacePosition.xyz);
vec3 H = normalize(L+V);
```

Next, the diffuse component is calculated using the dot product with the eye space normal.

```
float diffuse = max(0, dot(vEyeSpaceNormal, L));
```

The specular component is calculated as in the per-vertex case.

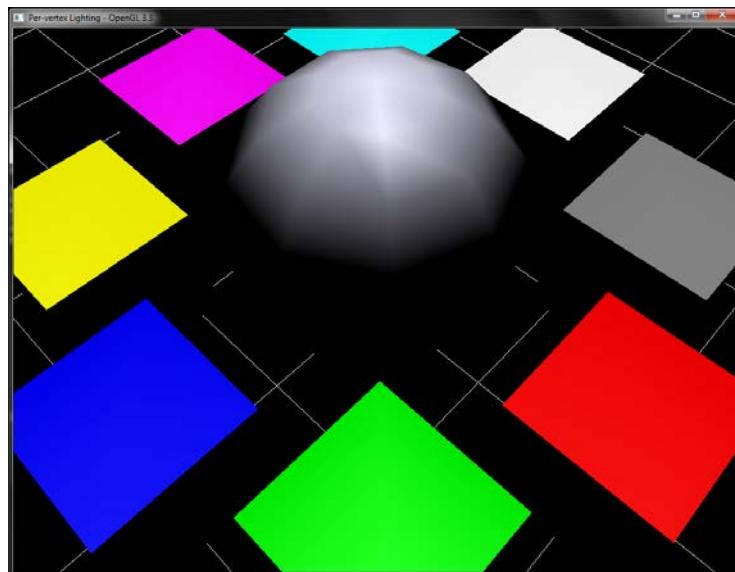
```
float specular = max(0, pow(dot(N, H), shininess));
```

Finally, the combined color is obtained by summing the diffuse and specular contributions. The diffuse contribution is obtained by multiplying the diffuse color with the diffuse component and the specular contribution is obtained by multiplying the specular component with the specular color.

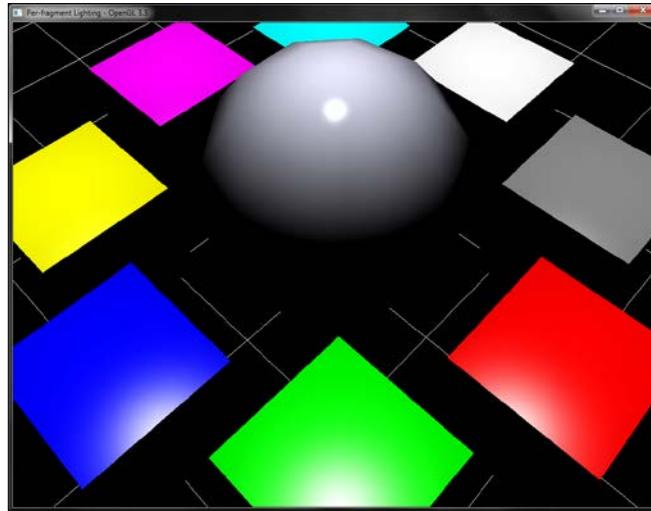
```
vFragColor = diffuse*vec4(diffuse_color, 1) +  
            specular*vec4(specular_color, 1);
```

There's more...

The output from the demo application for this recipe renders a sphere with eight cubes moving in and out, as shown in the following screenshot. The following figure shows the result of the per-vertex lighting. Note the ridge lines clearly visible on the middle sphere, which represents the vertices where the lighting calculations are carried out. Also note the appearance of the specular, which is predominantly visible at vertex positions only.



Now, let us see the result of the same demo application implementing per-fragment lighting:



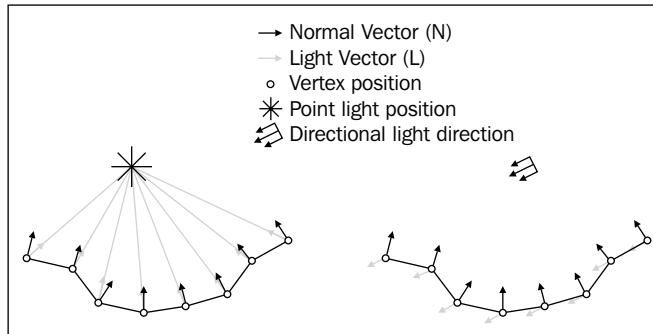
Note how the per-fragment lighting gives a smoother result compared to the per-vertex lighting. In addition, the specular component is clearly visible.

See also

Learning Modern 3D Graphics Programming, Section III, Jason L. McKesson: <http://www.arc synthesis.org/gltut/Illumination/Illumination.html>

Implementing per-fragment directional light

In this recipe, we will now implement directional light. The only difference between a point light and a directional light is that in the case of the directional light source, there is no position, however, there is direction, as shown in the following figure.



The figure compares directional and point light sources. For a point light source (left-hand side image), the light vector at each vertex is variable, depending on the relative positioning of the vertex with respect to the point light source. For directional light source (right-hand side image), all of the light vectors at vertices are the same and they all point in the direction of the directional light source.

Getting started

We will build on the geometry handling code from the per-fragment lighting recipe, but, instead of the pulsating cubes, we will now render a single cube with a sphere. The code for this recipe is contained in the Chapter4/DirectionalLight folder. The same code also works for per-vertex directional light.

How to do it...

Let us start the recipe by following these simple steps:

1. Calculate the light direction in eye space and pass it as shader uniform. Note that the last component is 0 since now we have a light direction vector.

```
lightDirectionES = glm::vec3(MV*  
                           glm::vec4(lightDirectionOS, 0));
```

2. In the vertex shader, output the eye space normal.

```
#version 330 core  
layout(location=0) in vec3 vVertex;  
layout(location=1) in vec3 vNormal;  
uniform mat4 MVP;  
uniform mat3 N;  
smooth out vec3 vEyeSpaceNormal;  
void main()  
{  
    vEyeSpaceNormal = N*vNormal;  
    gl_Position = MVP*vec4(vVertex, 1);  
}
```

3. In the fragment shader, compute the diffuse component by calculating the dot product between the light direction vector in eye space with the eye space normal, and multiply with the diffuse color to get the fragment color. Note that here, the light vector is independent of the eye space vertex position.

```
#version 330 core  
layout(location=0) out vec4 vFragColor;  
uniform vec3 light_direction;  
uniform vec3 diffuse_color;  
smooth in vec3 vEyeSpaceNormal;
```

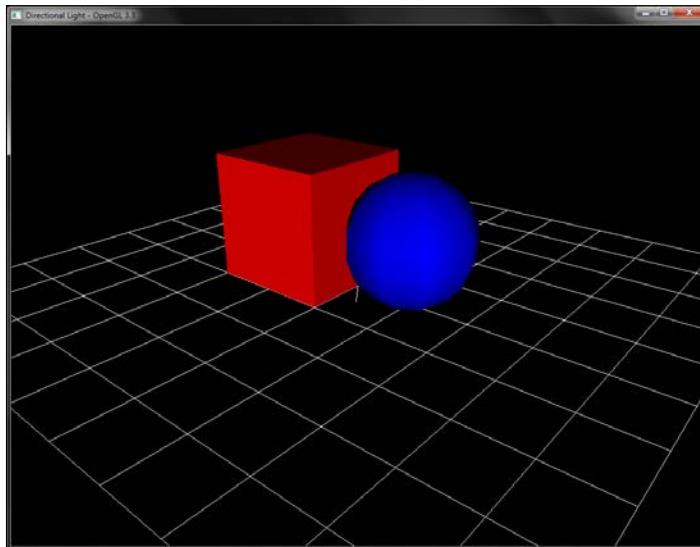
```
void main() {
    vec3 L = (light_direction);
    float diffuse = max(0, dot(vEyeSpaceNormal, L));
    vFragColor = diffuse*vec4(diffuse_color,1);
}
```

How it works...

The only difference between this recipe and the previous one is that we now pass the light direction instead of the position to the fragment shader. The rest of the calculation remains unchanged. If we want to apply attenuation, we can add the relevant shader snippets from the previous recipe.

There's more...

The demo application implementing this recipe shows a sphere and a cube object. In this demo, the direction of the light is shown by using a line segment at origin. The direction of the light can be changed using the right mouse button. The output from this demo application is shown in the following screenshot:



See also

- ▶ The *Implementing per-vertex and per-fragment point lighting* recipe
- ▶ *Learning Modern 3D Graphics Programming, Chapter 9, Lights On, Jason L. McKesson*: <http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2009.html>

Implementing per-fragment point light with attenuation

The previous recipe handled a directional light source but without attenuation. The relevant changes to enable per-fragment point light with attenuation will be given in this recipe. We start by implementing per-fragment point light, as in the *Implementing per-vertex and per-fragment point lighting* recipe.

Getting started

The code for this recipe is contained in the Chapter4/PointLight folder.

How to do it...

Implementing per-fragment point light is demonstrated by following these steps:

1. From the vertex shader, output the eye space vertex position and normal.

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;
uniform mat4 MVP;
uniform mat4 MV;
uniform mat3 N;
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;

void main() {
    vEyeSpacePosition = (MV*vec4(vVertex,1)).xyz;
    vEyeSpaceNormal = N*vNormal;
    gl_Position = MVP*vec4(vVertex,1);
}
```

2. In the fragment shader, calculate the light position in eye space, and then calculate the vector from the eye space vertex position to the eye space light position. Store the light distance before normalizing the light vector.

```
#version 330 core
layout(location=0) out vec4 vFragColor;
uniform vec3 light_position; //light position in object space
uniform vec3 diffuse_color;
uniform mat4 MV;
smooth in vec3 vEyeSpaceNormal;
```

```

smooth in vec3 vEyeSpacePosition;
const float k0 = 1.0; //constant attenuation
const float k1 = 0.0; //linear attenuation
const float k2 = 0.0; //quadratic attenuation

void main() {
    vec3 vEyeSpaceLightPosition =
        (MV*vec4(light_position,1)).xyz;
    vec3 L = (vEyeSpaceLightPosition-vEyeSpacePosition);
    float d = length(L);
    L = normalize(L);
    float diffuse = max(0, dot(vEyeSpaceNormal, L));
    float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
    diffuse *= attenuationAmount;
    vFragColor = diffuse*vec4(diffuse_color,1);
}

```

3. Apply attenuation based on the distance from the light source to the diffuse component.

```

float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
diffuse *= attenuationAmount;

```

4. Multiply the diffuse component to the diffuse color and set it as the fragment color.

```

vFragColor = diffuse*vec4(diffuse_color,1);

```

How it works...

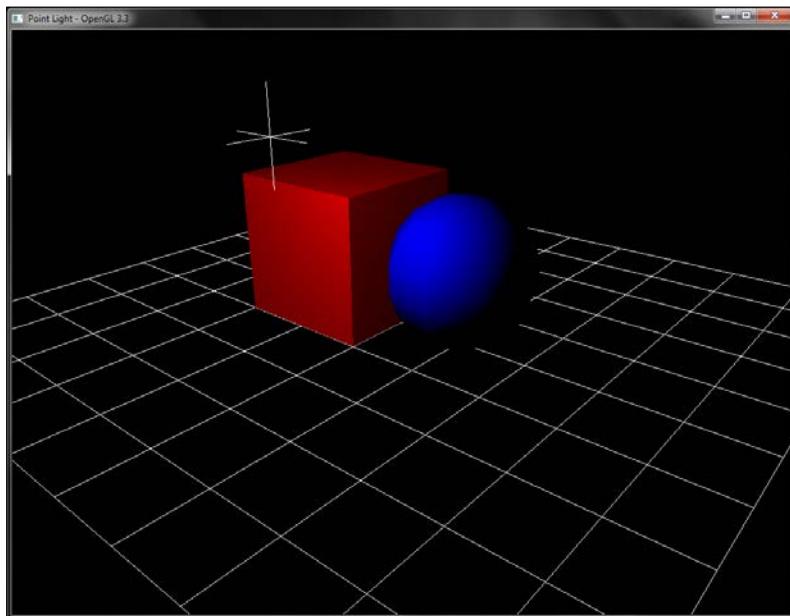
The recipe follows the *Implementing per-fragment directional light* recipe. In addition, it performs the attenuation calculation. The attenuation of light is calculated by using the following formula:

$$Attn(d) = \frac{1}{k1 + k2 * d + k3 * d^2}$$

Here, d is the distance from the current position to the light source and $k1$, $k2$, and $k3$ are the constant, linear, and quadratic attenuation coefficients respectively. For details about the values and their effect on lighting, we recommend the references in the See also section.

There's more...

The output from the demo application implementing this recipe is given in the following screenshot. In this recipe, we render a cube and a sphere. The position of light is shown using a crosshair on the screen. The camera position can be changed using the left mouse button and the light position can be changed by using the right mouse button. The light distance can be changed by using the mouse wheel.

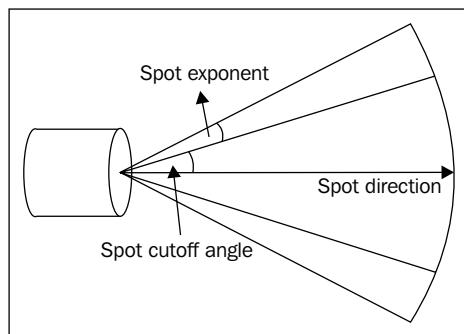


See also

- ▶ *Real-time Rendering, Third Edition, Tomas Akenine-Moller, Eric Haines, Naty Hoffman, A K Peters/CRC Press*
- ▶ *Learning Modern 3D Graphics Programming, Chapter 10, Plane Lights, Jason L. McKesson: <http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2010.html>*

Implementing per-fragment spot light

We will now implement per-fragment spot light. Spot light is a special point light that emits light in a directional cone. The size of this cone is determined by the spot cutoff amount, which is given in angles, as shown in the following figure. In addition, the sharpness of the spot is controlled by the parameter spot exponent. A higher value of the exponent gives a sharper falloff and vice versa.



Getting started

The code for this recipe is contained in the `Chapter4/SpotLight` directory. The vertex shader is the same as in the point light recipe. The fragment shader calculates the diffuse component, as in the *Implementing per-vertex and per-fragment point lighting* recipe.

How to do it...

Let us start this recipe by following these simple steps:

- From the light's object space position and spot light target's position, calculate the spot light direction vector in eye space.

```
spotDirectionES = glm::normalize(glm::vec3(MV *
    glm::vec4(spotPositionOS - lightPosOS, 0)))
```

- In the fragment shader, calculate the diffuse component as in point light. In addition, calculate the spot effect by finding the angle between the light direction and the spot direction vector.

```
vec3 L = (light_position.xyz - vEyeSpacePosition);
float d = length(L);
L = normalize(L);
vec3 D = normalize(spot_direction);
vec3 V = -L;
float diffuse = 1;
float spotEffect = dot(V, D);
```

3. If the angle is greater than the spot cutoff, apply the spot exponent and then use the diffuse shader on the fragment.

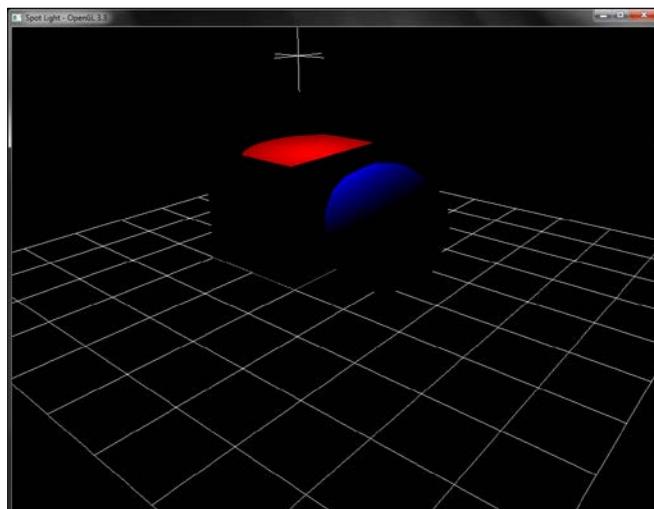
```
if(spotEffect > spot_cutoff) {  
    spotEffect = pow(spotEffect, spot_exponent);  
    diffuse = max(0, dot(vEyeSpaceNormal, L));  
    float attenuationAmount = spotEffect/(k0 + (k1*d) +  
        (k2*d*d));  
    diffuse *= attenuationAmount;  
    vFragColor = diffuse*vec4(diffuse_color,1);  
}
```

How it works...

The spot light is a special point light source that illuminates in a certain cone of direction. The amount of cone and the sharpness is controlled using the spot cutoff and spot exponent parameters respectively. Similar to the point light source, we first calculate the diffuse component. Instead of using the vector to light source (L) we use the opposite vector, which points in the direction of light ($v=-L$). Then we find out if the angle between the spot direction and the light direction vector is within the cutoff angle range. If it is, we apply the diffuse shading calculation. In addition, the sharpness of the spot light is controlled using the spot exponent parameter. This reduces the light in a falloff, giving a more pleasing spot light effect.

There's more...

The demo application implementing this recipe renders the same scene as in the point light demo. We can change the spot light direction using the right mouse button. The output result is shown in the following figure:



See also

- ▶ *Real-time Rendering, Third Edition, Tomas Akenine-Moller, Eric Haines, Naty Hoffman, A K Peters/CRC Press*
- ▶ Spot Light in GLSL tutorial at Ozone3D: http://www.ozone3d.net/tutorials/glsl_lighting_phong_p3.php

Implementing shadow mapping with FBO

Shadows give important cues about the relative positioning of graphical objects. There are myriads of shadow generation techniques, including shadow volumes, shadow maps, cascaded shadow maps, and so on. An excellent reference on several shadow generation techniques is given in the *See also* section. We will now see how to carry out basic shadow mapping using FBO.

Getting started

For this recipe, we will use the previous scene but instead of a grid object, we will use a plane object so that the generated shadows can be seen. The code for this recipe is contained in the Chapter4/ShadowMapping directory.

How to do it...

Let us start with this recipe by following these simple steps:

1. Create an OpenGL texture object which will be our shadow map texture. Make sure to set the clamp mode to `GL_CLAMP_TO_BORDER`, set the border color to `{1, 0, 0, 0}`, give the texture comparison mode to `GL_COMPARE_REF_TO_TEXTURE`, and set the compare function to `GL_EQUAL`. Set the texture internal format to `GL_DEPTH_COMPONENT24`.

```
glGenTextures(1, &shadowMapTexID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, shadowMapTexID);
 GLfloat border[4]={1,0,0,0};
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
 GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
 GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
 GL_CLAMP_TO_BORDER);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
 GL_CLAMP_TO_BORDER);
```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
    GL_LEQUAL);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
    border);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24,
    SHADOMAP_WIDTH, SHADOMAP_HEIGHT, 0, GL_DEPTH_COMPONENT,
    GL_UNSIGNED_BYTE, NULL);

```

2. Set up an FBO and use the shadow map texture as a single depth attachment. This will store the scene's depth from the point of view of light.

```

 glGenFramebuffers(1, &fboID);
 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D, shadowMapTexID, 0);
 GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
 if(status == GL_FRAMEBUFFER_COMPLETE) {
    cout<<"FBO setup successful."<<endl;
 } else {
    cout<<"Problem in FBO setup."<<endl;
 }
 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

3. Using the position and the direction of the light, set up the shadow matrix (S) by combining the light modelview matrix (MV_L), projection matrix (P_L), and bias matrix (B). For reducing runtime calculation, we store the combined projection and bias matrix (BP) at initialization.

```

MV_L = glm::lookAt(lightPosOS, glm::vec3(0,0,0),
    glm::vec3(0,1,0));
P_L = glm::perspective(50.0f, 1.0f, 1.0f, 25.0f);
B = glm::scale(glm::translate(glm::mat4(1),
    glm::vec3(0.5,0.5,0.5)), glm::vec3(0.5,0.5,0.5));
BP = B * P_L;
S = BP * MV_L;

```

4. Bind the FBO and render the scene from the point of view of the light. Make sure to enable front-face culling (`glEnable(GL_CULL_FACE)` and `glCullFace(GL_FRONT)`) so that the back-face depth values are rendered. Otherwise our objects will suffer from shadow acne.



Normally, a simple shader could be used for rendering of a scene in the depth texture. This may also be achieved by disabling writing to the color buffer (`glDrawBuffer(GL_NONE)`) and then enabling it for normal rendering. In addition, an offset bias can also be added in the shader code to reduce shadow acne.

```
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glClear(GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, SHADOWMAP_WIDTH, SHADOWMAP_HEIGHT);
glCullFace(GL_FRONT);
DrawScene(MV_L, P_L);
glCullFace(GL_BACK);
```

5. Disable FBO, restore default viewport, and render the scene normally from the point of view of the camera.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0, 0, WIDTH, HEIGHT);
DrawScene(MV, P, 0);
```

6. In the vertex shader, multiply the world space vertex positions (`M*vec4(vVertex, 1)`) with the shadow matrix (`S`) to obtain the shadow coordinates. These will be used for lookup of the depth values from the shadowmap texture in the fragment shader.

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;

uniform mat4 MVP;           //modelview projection matrix
uniform mat4 MV;            //modelview matrix
uniform mat4 M;             //model matrix
uniform mat3 N;             //normal matrix
uniform mat4 S;             //shadow matrix
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;
smooth out vec4 vShadowCoords;
void main()
{
    vEyeSpacePosition = (MV*vec4(vVertex, 1)).xyz;
    vEyeSpaceNormal = N*vNormal;
    vShadowCoords = S*(M*vec4(vVertex, 1));
    gl_Position = MVP*vec4(vVertex, 1);
}
```

7. In the fragment shader, use the shadow coordinates to lookup the depth value in the shadow map sampler which is of the `sampler2DShadow` type. This sampler can be used with the `textureProj` function to return a comparison outcome. We then use the comparison result to darken the diffuse component, simulating shadows.

```

version 330 core
layout(location=0) out vec4 vFragColor;
uniform sampler2DShadow shadowMap;
uniform vec3 light_position; //light position in eye space
uniform vec3 diffuse_color;
smooth in vec3 vEyeSpaceNormal;
smooth in vec3 vEyeSpacePosition;
smooth in vec4 vShadowCoords;
const float k0 = 1.0; //constant attenuation
const float k1 = 0.0; //linear attenuation
const float k2 = 0.0; //quadratic attenuation
uniform bool bIsLightPass; //no shadows in light pass
void main() {
    if(bIsLightPass)
        return;
    vec3 L = (light_position.xyz-vEyeSpacePosition);
    float d = length(L);
    L = normalize(L);
    float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
    float diffuse = max(0, dot(vEyeSpaceNormal, L)) *
                    attenuationAmount;
    if(vShadowCoords.w>1) {
        float shadow = textureProj(shadowMap, vShadowCoords);
        diffuse = mix(diffuse, diffuse*shadow, 0.5);
    }
    vFragColor = diffuse*vec4(diffuse_color, 1);
}

```

How it works...

The shadow mapping algorithm works in two passes. In the first pass, the scene is rendered from the point of view of light, and the depth buffer is stored into a texture called `shadowmap`. We use a single FBO with a depth attachment for this purpose. Apart from the conventional minification/magnification texture filtering, we set the texture wrapping mode to `GL_CLAMP_TO_BORDER`, which ensures that the values are clamped to the specified border color. Had we set this as `GL_CLAMP` or `GL_CLAMP_TO_EDGE`, the border pixels forming the shadow map would produce visible artefacts.

The shadowmap texture has some additional parameters. The first is the `GL_TEXTURE_COMPARE_MODE` parameter, which is set as the `GL_COMPARE_REF_TO_TEXTURE` value. This enables the texture to be used for depth comparison in the shader. Next, we specify the `GL_TEXTURE_COMPARE_FUNC` parameter, which is set as `GL_LEQUAL`. This compares the currently interpolated texture coordinate value (r) with the depth texture's sample value (D). It returns 1 if $r \leq D$, otherwise it returns 0. This means that if the depth of the current sample is less than or equal to the depth from the shadowmap texture, the sample is not in shadow; otherwise, it is in shadow. The `textureProj` GLSL shader function performs this comparison for us and returns 0 or 1 based on whether the point is in shadow or not. These are the texture parameters required for the shadowmap texture.

To ensure that we do not have any shadow acne, we enable front-face culling (`glEnable(GL_CULL_FACE)` and `glCullFace(GL_FRONT)`) so that the back-face depth values get written to the shadowmap texture. In the second pass, the scene is rendered normally from the point of view of the camera and the shadow map is projected on the scene geometry using shaders.

To render the scene from the point of view of light, the modelview matrix of the light (MV_L), the projection matrix (P_L), and the bias matrix (B) are calculated. After multiplying with the projection matrix, the coordinates are in clip space (that is, they range from [-1,-1,-1]) to [1,1,1]. The bias matrix rescales this range to bring the coordinates from [0,0,0] to [1,1,1] range so that the shadow lookup can be carried out.

If we have the object's vertex position in the object space given as V_{obj} , the shadow coordinates (UV_{proj}) for the lookup in the shadow map can be given by multiplying the shadow matrix (S) with the world space position of the object ($M * V_{obj}$). The whole series of transformations is given as follows:

$$\begin{aligned} UV_{proj} &= S * MV * V_{obj} \\ S &= B * P_L * MV_L \end{aligned}$$

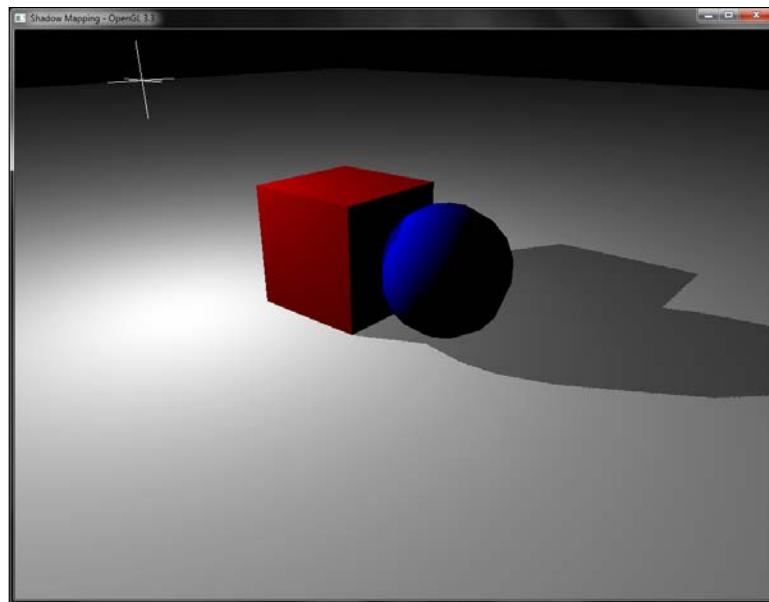
Here, B is the bias matrix, P_L is the projection matrix of light, and MV_L is the modelview matrix of light. For efficiency, we precompute the bias matrix of the light and the projection matrix, since they are unchanged for the lifetime of the application. Based on the user input, the light's modelview is modified and then the shadow matrix is recalculated. This is then passed to the shader.

In the vertex shader, the shadowmap texture coordinates are obtained by multiplying the world space vertex position ($M * V_{obj}$) with the shadow matrix (S). In the fragment shader, the shadow map is looked up using the projected texture coordinate to find if the current fragment is in shadow. Before the texture lookup, we check the value of the w coordinate of the projected texture coordinate. We only do our calculations if the w coordinate is greater than 1. This ensures that we only accept the forward projection and reject the back projection. Try removing this condition to see what we mean.

The shadow map lookup computation is facilitated by the `textureProj` GLSL function. The result from the shadow map lookup returns 1 or 0. This result is multiplied with the shading computation. As it happens in the real world, we never have coal black shadows. Therefore, we combine the shadow outcome with the shading computation by using the `mix` GLSL function.

There's more...

The demo application for this recipe shows a plane, a cube, and a sphere. A point light source, which can be rotated using the right mouse button, is placed. The distance of the light source can be altered using the mouse wheel. The output result from the demo is displayed in the following figure:



This recipe detailed the shadow mapping technique for a single light source. With each additional light source, the processing, as well as storage requirements, increase.

See also

- ▶ *Real-time Shadows, Elmar Eisemann, Michael Schwarz, Ulf Assarsson, Michael Wimmer, A K Peters/CRC Press*
- ▶ *OpenGL 4.0 Shading Language Cookbook, Chapter 7, Shadows, David Wolff, Packt Publishing*
- ▶ *ShadowMapping with GLSL* by Fabien Sanglard: <http://www.fabiensanglard.net/shadowmapping/index.php>

Implementing shadow mapping with percentage closer filtering (PCF)

The shadow mapping algorithm, though simple to implement, suffers from aliasing artefacts, which are due to the shadowmap resolution. In addition, the shadows produced using this approach are hard. These can be minimized either by increasing the shadowmap resolution or taking more samples. The latter approach is called **percentage closer filtering (PCF)**, where more samples are taken for the shadowmap lookup and the percentage of the samples is used to estimate if a fragment is in shadow. Thus, in PCF, instead of a single lookup, we sample an $n \times n$ neighborhood of shadowmap and then average the values.

Getting started

The code for this recipe is contained in the `Chapter4/ShadowMappingPCF` directory. It builds on top of the previous recipe, *Implementing shadow mapping with FBO*. We use the same scene but augment it with PCF.

How to do it...

Let us see how to extend the basic shadow mapping with PCF.

1. Change the shadowmap texture minification/magnification filtering modes to `GL_LINEAR`. Here, we exploit the texture filtering capabilities of the GPU to reduce aliasing artefacts during sampling of the shadow map. Even with the linear filtering support, we have to take additional samples to reduce the artefacts.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
```

2. In the fragment shader, instead of a single texture lookup as in the shadow map recipe, we use a number of samples. GLSL provides a convenient function, `textureProjOffset`, to allow calculation of samples using an offset. For this recipe, we look at a 3×3 neighborhood around the current shadow map point. Hence, we use a large offset of 2. This helps to reduce sampling artefacts.

```
if(vShadowCoords.w>1) {
    float sum = 0;
    sum += textureProjOffset(shadowMap, vShadowCoords,
    ivec2(-2, -2));
    sum += textureProjOffset(shadowMap, vShadowCoords,
    ivec2(-2, 0));
    sum += textureProjOffset(shadowMap, vShadowCoords,
    ivec2(-2, 2));
```

```

sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 0, -2));
sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 0, 0));
sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 0, 2));
sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 2, -2));
sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 2, 0));
sum += textureProjOffset(shadowMap, vShadowCoords,
ivec2( 2, 2));
float shadow = sum/9.0;
diffuse = mix(diffuse, diffuse*shadow, 0.5);
}

```

How it works...

In order to implement PCF, the first change we need is to set the texture filtering mode to linear filtering. This change enabled the GPU to bilinearly interpolate the shadow value. This gives smoother edges since the hardware does PCF filtering underneath. However it is not enough for our purpose. Therefore, we have to take additional samples to improve the result.

Fortunately, we can use a convenient function, `textureProjOffset`, which accepts an offset that is added to the given shadow map texture coordinate. Note that the offset given to this function must be a constant literal. Thus, we cannot use a loop variable for dynamic sampling of the shadow map sampler. We, therefore, have to unroll the loop to sample the neighborhood.

We use an offset of 2 units because we wanted to sample at a value of 1.5. However, since the `textureProjOffset` function does not accept a floating point value, we round it to the nearest integer. The offset is then modified to move to the next sample point until the entire 3×3 neighborhood is sampled. We then average the sampling result for the entire neighborhood. The obtained sampling result is then multiplied to the lighting contribution, thus, producing shadows if the current sample happens to be in an occluded region.

Even with adding additional samples, we get sampling artefacts. These can be reduced by shifting the sampling points randomly. To achieve this, we first implement a pseudo-random function in GLSL as follows:

```

float random(vec4 seed) {
    float dot_product = dot(seed, vec4(12.9898, 78.233, 45.164,
                                      94.673));
    return fract(sin(dot_product) * 43758.5453);
}

```

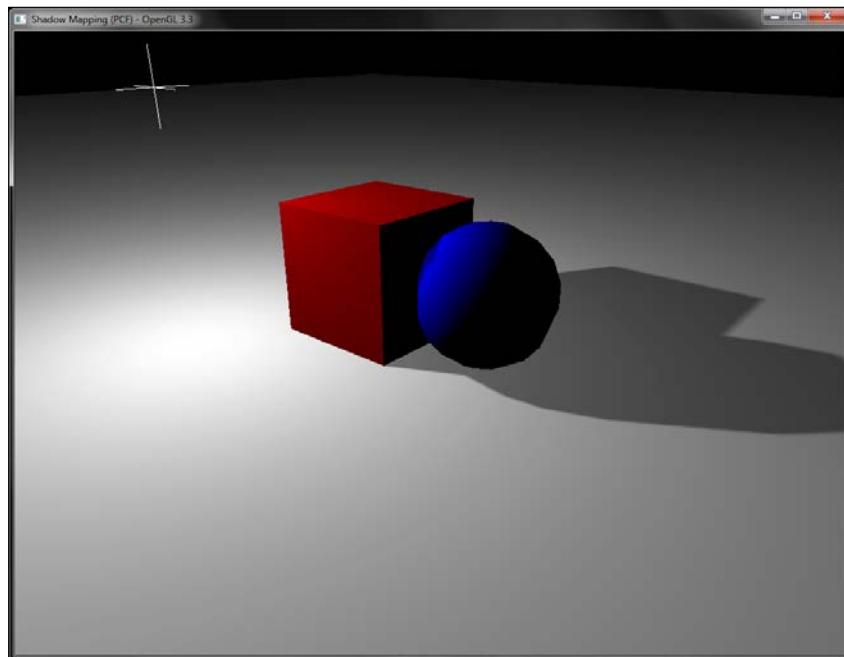
Then, the sampling for PCF uses the noise function to shift the shadow offset, as shown in the following shader code:

```
for(int i=0;i<16;i++) {  
    float indexA = (random(vec4(gl_FragCoord.xyx, i))*0.25);  
    float indexB = (random(vec4(gl_FragCoord.yxy, i))*0.25);  
    sum += textureProj(shadowMap, vShadowCoords +  
        vec4(indexA, indexB, 0, 0));  
}  
shadow = sum/16.0;
```

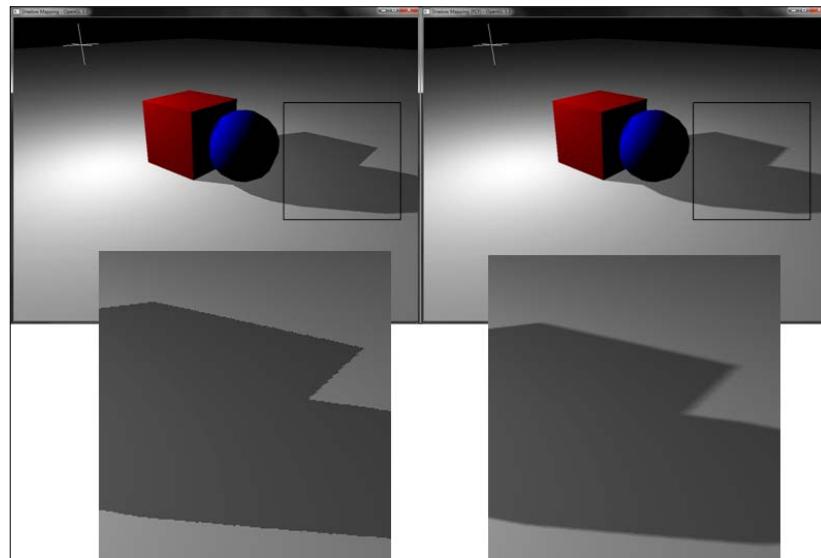
In the given code, three macros are defined, `STRATIFIED_3x3` (for 3x3 stratified sampling), `STRATIFIED_5x5` (for 5x5 stratified sampling), and `RANDOM_SAMPLING` (for 4x4 random sampling).

There's more...

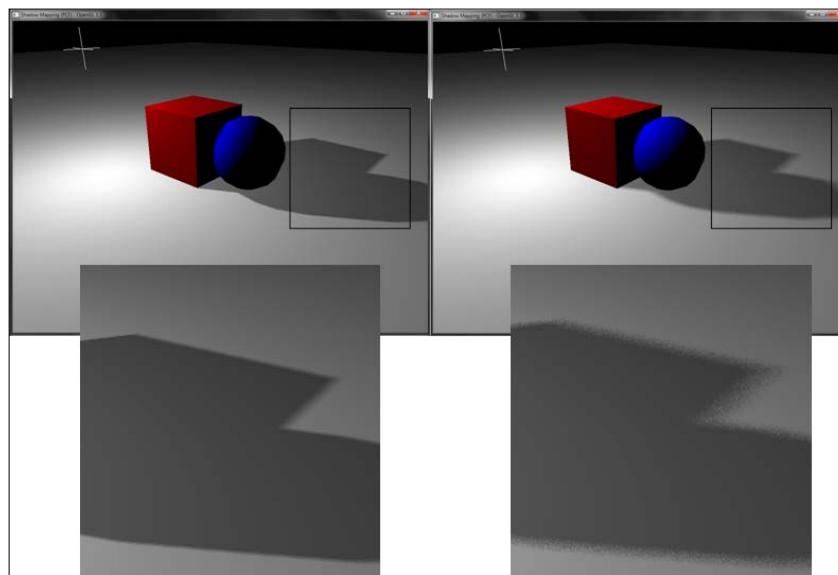
Making these changes, we get a much better result, as shown in the following figure. If we take a bigger neighborhood, we get a better result. However, the computational requirements also increase.



The following figure compares this result of the PCF-filtered shadow map (right) with a normal shadow map (left). We can see that the PCF-filtered result gives softer shadows with reduced aliasing artefacts.



The following figure compares the result of the stratified PCF-filtered image (left) against the random PCF-filtered image (right). As can be seen, the noise-filtered image gives a much better result.



See also

- ▶ GPU Gems, Chapter 11, *Shadow Map Antialiasing*, Michael Bunnell, Fabio Pellacini, available online at: http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html
- ▶ Shadow mapping, Tutorial 16: <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>

Implementing variance shadow mapping

In this recipe, we will cover a technique which gives a much better result, has better performance, and at the same time is easier to calculate. The technique is called **variance shadow mapping**. In conventional PCF-filtered shadow mapping, we compare the depth value of the current fragment to the mean depth value in the shadow map, and based on the outcome, we shadow the fragment.

In case of variance shadow mapping, the mean depth value (also called first moment) and the mean squared depth value (also called second moment) are calculated and stored. Then, rather than directly using the mean depth, the variance is used. The variance calculation requires both the mean depth as well as the mean of the squared depth. Using the variance, the probability of whether the given sample is shadowed is estimated. This probability is then compared to the maximum probability to determine if the current sample is shadowed.

Getting started

For this recipe, we will build on top of the shadow mapping recipe, *Implementing shadow mapping with FBO*. The code for this recipe is contained in the `Chapter4/VarianceShadowMapping` folder.

How to do it...

Let us start our recipe by following these simple steps:

1. Set up the `shadowmap` texture as in the shadow map recipe, but this time remove the depth compare mode (`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE)` and `glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL)`). Also set the format of the texture to the `GL_RGBA32F` format. Also enable the mipmap generation for this texture. The mipmaps provide filtered textures across different scales and produces better alias-free shadows. We request five mipmap levels (by specifying the max level as 4).

```
glGenTextures(1, &shadowMapTexID);  
glActiveTexture(GL_TEXTURE0);
```

```

glBindTexture(GL_TEXTURE_2D, shadowMapTexID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR;
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
border;
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, SHADOWMAP_WIDTH,
SHADOWMAP_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
glGenerateMipmap(GL_TEXTURE_2D);

```

2. Set up two FBOs: one for shadowmap generation and another for shadowmap filtering. The shadowmap FBO has a renderbuffer attached to it for depth testing. The filtering FBO does not have a renderbuffer attached to it but it has two texture attachments.

```

 glGenFramebuffers(1, &fboID);
 glGenRenderbuffers(1, &rboID);
 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glBindRenderbuffer(GL_RENDERBUFFER, rboID);
 glRenderbufferStorage(GL_RENDERBUFFER,
 GL_DEPTH_COMPONENT32, SHADOWMAP_WIDTH,
 SHADOWMAP_HEIGHT);

 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 GL_TEXTURE_2D, shadowMapTexID, 0);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER,
 GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboID);

 GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
 if(status == GL_FRAMEBUFFER_COMPLETE) {
    cout<<"FBO setup successful."<<endl;
 } else {
    cout<<"Problem in FBO setup."<<endl;
 }
 glBindFramebuffer(GL_FRAMEBUFFER, 0);

 glGenFramebuffers(1, &filterFBOID);
 glBindFramebuffer(GL_FRAMEBUFFER, filterFBOID);
 glGenTextures(2, blurTexID);
 for(int i=0;i<2;i++) {
    glActiveTexture(GL_TEXTURE1+i);
}

```

```

glBindTexture(GL_TEXTURE_2D, blurTexID[i]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
border);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, SHADOWMAP_WIDTH,
SHADOWMAP_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0+i, GL_TEXTURE_2D, blurTexID[i], 0);
}
status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE) {
    cout<<"Filtering FBO setup successful."<<endl;
} else {
    cout<<"Problem in Filtering FBO setup."<<endl;
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

- Bind the shadowmap FBO, set the viewport to the size of the shadowmap texture, and render the scene from the point of view of the light, as in the *Implementing shadow mapping with FBO* recipe. In this pass, instead of storing the depth as in the shadow mapping recipe, we use a custom fragment shader (Chapter4/VarianceShadowmapping/shaders/firststep.frag) to output the *depth* and *depth*depth* values in the **red** and **green** channels of the fragment output color.

```

glBindFramebuffer(GL_FRAMEBUFFER, fbOID);
glViewport(0, 0, SHADOWMAP_WIDTH, SHADOWMAP_HEIGHT);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
DrawSceneFirstPass(MV_L, P_L);

```

The shader code is as follows:

```

#version 330 core
layout(location=0) out vec4 vFragColor;
smooth in vec4 clipSpacePos;
void main()
{
    vec3 pos = clipSpacePos.xyz/clipSpacePos.w; // -1 to 1
    pos.z += 0.001; // add some offset to remove the shadow
    acne

```

```

    float depth = (pos.z +1)*0.5; // 0 to 1
    float moment1 = depth;
    float moment2 = depth * depth;
    vFragColor = vec4(moment1,moment2,0,0);
}

```

4. Bind the filtering **FBO** to filter the shadowmap texture generated in the first pass using separable Gaussian smoothing filters, which are more efficient and offer better performance. We first attach the vertical smoothing fragment shader (Chapter4/VarianceShadowmapping/shaders/GaussV.frag) to filter the shadowmap texture and then the horizontal smoothing fragment shader (Chapter4/VarianceShadowmapping/shaders/GaussH.frag) to smooth the output from the vertical Gaussian smoothing filter.

```

glBindFramebuffer(GL_FRAMEBUFFER,filterFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glBindVertexArray(quadVAOID);
gaussianV_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
glDrawBuffer(GL_COLOR_ATTACHMENT1);
gaussianH_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
glBindFramebuffer(GL_FRAMEBUFFER,0);

```

The horizontal Gaussian blur shader is as follows:

```

#version 330 core
layout(location=0) out vec4 vFragColor;
smooth in vec2 vUV;
uniform sampler2D textureMap;

const float kernel[] = float[21] (0.000272337, 0.00089296,
0.002583865, 0.00659813, 0.014869116, 0.029570767,
0.051898313, 0.080381679, 0.109868729, 0.132526984,
0.14107424, 0.132526984, 0.109868729, 0.080381679,
0.051898313, 0.029570767, 0.014869116, 0.00659813,
0.002583865, 0.00089296, 0.000272337);

void main()
{
    vec2 delta = 1.0/textureSize(textureMap,0);
    vec4 color = vec4(0);
    int index = 20;

    for(int i=-10;i<=10;i++) {
        color += kernel[index--]*texture(textureMap, vUV +
        (vec2(i*delta.x,0)));
    }
}

```

```
    }  
  
    vFragColor = vec4(color.xy, 0, 0);  
}
```

In the vertical Gaussian shader, the loop statement is modified, whereas the rest of the shader is the same.

```
color += kernel[index--]*texture(textureMap, vUV +  
(vec2(0,i*delta.y)));
```

5. Unbind the FBO, reset the default viewport, and then render the scene normally, as in the shadow mapping recipe.

```
glDrawBuffer(GL_BACK_LEFT);  
glViewport(0, 0, WIDTH, HEIGHT);  
DrawScene(MV, P);
```

How it works...

The variance shadowmap technique tries to represent the depth data such that it can be filtered linearly. Instead of storing the depth, it stores the depth and depth*depth value in a floating point texture, which is then filtered to reconstruct the first and second moments of the depth distribution. Using the moments, it estimates the variance in the filtering neighborhood. This helps in finding the probability of a fragment at a specific depth to be occluded using Chebyshev's inequality. For more mathematical details, we refer the reader to the *See also* section of this recipe.

From the implementation point of view, similar to the shadow mapping recipe, the method works in two passes. In the first pass, we render the scene from the point of view of light. Instead of storing the depth, we store the depth and the depth*depth values in a floating point texture using the custom fragment shader (see [Chapter4/VarianceShadowmapping/shaders/firststep.frag](#)).

The vertex shader outputs the clip space position to the fragment shader using which the fragment depth value is calculated. To reduce self-shadowing, a small bias is added to the z value.

```
vec3 pos = clipSpacePos.xyz/clipSpacePos.w;  
pos.z += 0.001;  
float depth = (pos.z +1)*0.5;  
float moment1 = depth;  
float moment2 = depth * depth;  
vFragColor = vec4(moment1,moment2,0,0);
```

After the first pass, the `shadowmap` texture is blurred using a separable Gaussian smoothing filter. First the vertical and then the horizontal filter is applied to the `shadowmap` texture by applying the `shadowmap` texture to a full-screen quad and alternating the filter FBO's color attachment. Note that the `shadowmap` texture is bound to texture unit 0 whereas the textures used for filtering are bound to texture unit 1 (attached to `GL_COLOR_ATTACHMENT0` on the filtering FBO) and texture unit 2 (attached to `GL_COLOR_ATTACHMENT1` on the filtering FBO).

```
glBindFramebuffer(GL_FRAMEBUFFER,fboID);
glViewport(0,0,SHADOWMAP_WIDTH, SHADOWMAP_HEIGHT);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
DrawSceneFirstPass(MV_L, P_L);

glBindFramebuffer(GL_FRAMEBUFFER,filterFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glBindVertexArray(quadVAOID);
gaussianV_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);

glDrawBuffer(GL_COLOR_ATTACHMENT1);
gaussianH_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
glBindFramebuffer(GL_FRAMEBUFFER,0);
glDrawBuffer(GL_BACK_LEFT);
glViewport(0,0,WIDTH, HEIGHT);
```

In the second pass, the scene is rendered from the point of view of the camera. The blurred shadowmap is used in the second pass as a texture to lookup the sample value (see `Chapter4/VarianceShadowmapping/shaders/VarianceShadowMap.{vert, frag}`). The variance shadow mapping vertex shader outputs the shadow texture coordinates, as in the shadow mapping recipe.

```
#version 330 core
layout(location=0) in vec3 vVertex;
layout(location=1) in vec3 vNormal;
uniform mat4 MVP; //modelview projection matrix
uniform mat4 MV; //modelview matrix
uniform mat4 M; //model matrix
uniform mat3 N; //normal matrix
uniform mat4 S; //shadow matrix
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;
smooth out vec4 vShadowCoords;
void main()
```

```
{
    vEyeSpacePosition = (MV*vec4 (vVertex,1)).xyz;
    vEyeSpaceNormal = N*vNormal;
    vShadowCoords = S*(M*vec4 (vVertex,1));
    gl_Position = MVP*vec4 (vVertex,1);
}
```

The variance shadow mapping fragment shader operates differently. We first make sure that the shadow coordinates are in front of the light (to prevent back projection), that is, `shadowCoord.w>1`. Next, the `shadowCoords.xyz` values are divided by the homogeneous coordinate, `shadowCoord.w`, to get the depth value.

```
if(vShadowCoords.w>1) {
    vec3 uv = vShadowCoords.xyz/vShadowCoords.w;
    float depth = uv.z;
```

The texture coordinates after homogeneous division are used to lookup the shadow map storing the two moments. The two moments are used to estimate the variance. The variance is clamped and then the occlusion probability is estimated. The diffuse component is then modulated based on the obtained occlusion probability.

```
vec4 moments = texture(shadowMap, uv.xy);
float E_x2 = moments.y;
float Ex_2 = moments.x*moments.x;
float var = E_x2-Ex_2;
var = max(var, 0.00002);
float mD = depth-moments.x;
float mD_2 = mD*mD;
float p_max = var/(var+ mD_2);
diffuse *= max(p_max, (depth<=moments.x)?1.0:0.2);
}
```

To recap, here is the complete variance shadow mapping fragment shader:

```
#version 330 core
layout(location=0) out vec4 vFragColor;
uniform sampler2D shadowMap;
uniform vec3 light_position; //light position in object space
uniform vec3 diffuse_color;
uniform mat4 MV;
smooth in vec3 vEyeSpaceNormal;
smooth in vec3 vEyeSpacePosition;
smooth in vec4 vShadowCoords;
const float k0 = 1.0; //constant attenuation
const float k1 = 0.0; //linear attenuation
```

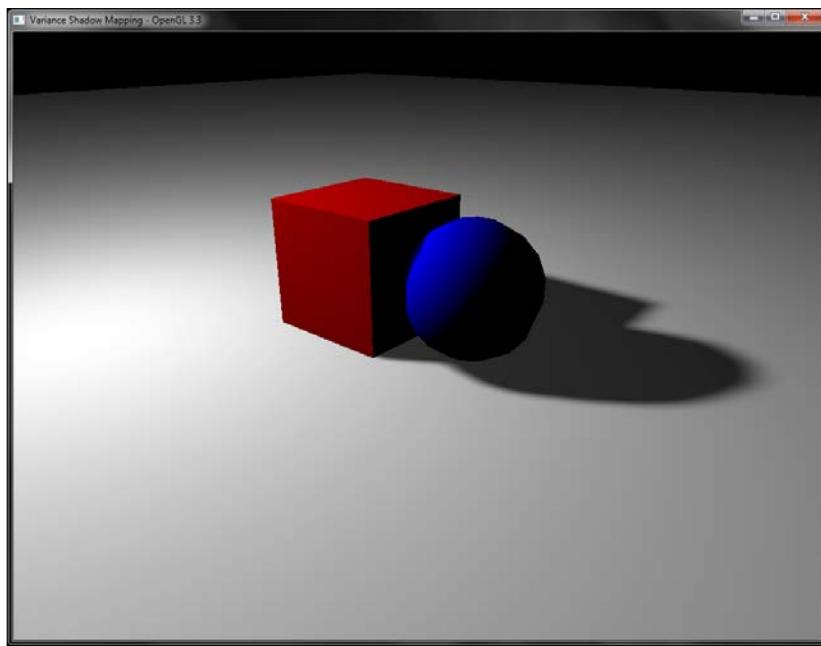
```
const float k2 = 0.0; //quadratic attenuation
void main() {
    vec4 vEyeSpaceLightPosition = (MV*vec4(light_position,1));
    vec3 L = (vEyeSpaceLightPosition.xyz-vEyeSpacePosition);
    float d = length(L);
    L = normalize(L);
    float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
    float diffuse = max(0, dot(vEyeSpaceNormal, L)) *
        attenuationAmount;
    if(vShadowCoords.w>1) {
        vec3 uv = vShadowCoords.xyz/vShadowCoords.w;
        float depth = uv.z;
        vec4 moments = texture(shadowMap, uv.xy);
        float E_x2 = moments.y;
        float Ex_2 = moments.x*moments.x;
        float var = E_x2-Ex_2;
        var = max(var, 0.00002);
        float mD = depth-moments.x;
        float mD_2 = mD*mD;
        float p_max = var/(var+ mD_2);
        diffuse *= max(p_max, (depth<=moments.x)?1.0:0.2);
    }
    vFragColor = diffuse*vec4(diffuse_color, 1);
}
```

There's more...

Variance shadow mapping is an interesting idea. However, it does suffer from light bleeding artefacts. There have been several improvements to the basic technique, such as summed area variance shadow maps, layered variance shadow maps, and more recently, sample distribution shadow maps, that are referred to in the *See also* section of this recipe. After getting a practical insight into the basic variance shadow mapping idea, we invite the reader to try and implement the different variants of this algorithm, as detailed in the references in the *See also* section.

Lights and Shadows

The demo application for this recipe shows the same scene (a cube and a sphere on a plane) lit by a point light source. Right-clicking the mouse button rotates the point light around the objects. The output result is shown in the following figure:



Comparing this output to the previous shadow mapping recipes, we can see that the output quality is much better if compared to the conventional shadow mapping and the PCF-based technique. When comparing the outputs, variance shadow mapping gives a better output with a significantly less number of samples. Obtaining the same output using PCF or any other technique would require a very large neighborhood lookup with more samples. This makes this technique well-suited for real-time applications such as games.

See also

- ▶ *Proceedings of the 2006 symposium on Interactive 3D graphics and games, Variance Shadow Maps, pages 161-165 William Donnelly, Andrew Lauritzen*
- ▶ *GPU Gems 3, Chapter 8, Summed-Area Variance Shadow Maps, Andrew Lauritzen:* http://http.developer.nvidia.com/GPUGems3/gpugems3_ch08.html
- ▶ *Proceedings of the Graphics Interface 2008, Layered variance shadow maps, pages 139-146, Andrew Lauritzen, Michael McCool*
- ▶ *Sample Distribution Shadow Maps, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D) 2011, February, Andrew Lauritzen, Marco Salvi, and Aaron Lefohn*

5

Mesh Model Formats and Particle Systems

In this chapter, we will focus on:

- ▶ Implementing terrains using height map
- ▶ Implementing 3ds model loading using separate buffers
- ▶ Implementing OBJ model loading using interleaved buffers
- ▶ Implementing EZMesh model loading
- ▶ Implementing a simple particle system

Introduction

While simple demos and applications can get along with basic primitives like cubes and spheres, most real-world applications and games use 3D mesh models which are modelled in 3D modeling software such as 3ds Max and Maya. For games, the models are then exported into the proprietary game format and then the models are loaded into the game.

While there are many formats available, some formats such as Autodesk® 3ds and Wavefront® OBJ are common formats. In this chapter, we will look at recipes for loading these model formats. We will look at how to load the geometry information, stored in the external files, into the vertex buffer object memory of the GPU. In addition, we will also load material and texture information which is required to improve the fidelity of the model so that it appears more realistic. We will also work on loading terrains which are often used to model outdoor environments. Finally, we will implement a basic particle system for simulating fuzzy phenomena such as fire and smoke. All of the discussed techniques will be implemented in the OpenGL v3.3 and above core profile.

Implementing terrains using the height map

Several demos and applications require rendering of terrains. This recipe will show how to implement terrain generation in modern OpenGL. The height map is loaded using the SOIL image loading library which contains displacement information. A 2D grid is then generated depending on the required terrain resolution. Then, the displacement information contained in the height map is used to displace the 2D grid in the vertex shader. Usually, the obtained displacement value is scaled to increase or decrease the displacement scale as desired.

Getting started

For the terrain, first the 2D grid geometry is generated depending on the terrain resolution. The steps to generate such geometry were previously covered in the *Doing a ripple mesh deformer using vertex shader* recipe in Chapter 1, *Introduction to Modern OpenGL*. The code for this recipe is contained in the Chapter5/TerrainLoading directory.

How to do it...

Let us start our recipe by following these simple steps:

1. Load the height map texture using the SOIL image loading library and generate an OpenGL texture from it. The texture filtering is set to GL_NEAREST as we want to obtain the exact values from the height map. If we had changed this to GL_LINEAR, we would get interpolated values. Since the terrain height map is not tiled, we set the texture wrap mode to GL_CLAMP.

```
int texture_width = 0, texture_height = 0, channels=0;
GLubyte* pData = SOIL_load_image(filename.c_str(),
&texture_width, &texture_height, &channels, SOIL_LOAD_L);
//vertically flip the image data
for( j = 0; j*2 < texture_height; ++j )
{
    int index1 = j * texture_width ;
    int index2 = (texture_height - 1 - j) * texture_width ;
    for( i = texture_width ; i > 0; --i )
    {
        GLubyte temp = pData[index1];
        pData[index1] = pData[index2];
        pData[index2] = temp;
        ++index1;
        ++index2;
    }
}
```

```

glGenTextures(1, &heightMapTextureID);
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, heightMapTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, texture_width,
texture_height, 0, GL_RED, GL_UNSIGNED_BYTE, pData);
SOIL_free_image_data(pData);

```

2. Set up the terrain geometry by generating a set of points in the XZ plane. The TERRAIN_WIDTH parameter controls the total number of vertices in the X axis whereas the TERRAIN_DEPTH parameter controls the total number of vertices in the Z axis.

```

for( j=0;j<TERRAIN_DEPTH;j++) {
    for( i=0;i<TERRAIN_WIDTH;i++) {
        vertices [count]=glm::vec3((float(i)/(TERRAIN_WIDTH-
1)), 0, (float(j)/(TERRAIN_DEPTH-1)));
        count++;
    }
}

```

3. Set up the vertex shader that displaces the 2D terrain mesh. Refer to Chapter5/TerrainLoading/shaders/shader.vert for details. The height value is obtained from the height map. This value is then added to the current vertex position and finally multiplied with the combined modelview projection (MVP) matrix to get the clip space position. The HALF_TERRAIN_SIZE uniform contains half of the total number of vertices in both the X and Z axes, that is, HALF_TERRAIN_SIZE = ivec2(TERRAIN_WIDTH/2, TERRAIN_DEPTH/2). Similarly the scale uniform is used to scale the height read from the height map. The half_scale and HALF_TERRAIN_SIZE uniforms are used to position the mesh at origin.

```

#version 330 core
layout (location=0) in vec3 vVertex;
uniform mat4 MVP;
uniform ivec2 HALF_TERRAIN_SIZE;
uniform sampler2D heightMapTexture;
uniform float scale;
uniform float half_scale;
void main()
{

```

```
    float height = texture(heightMapTexture,
    vVertex.xz).r*scale - half_scale;
    vec2 pos   = (vVertex.xz*2.0-1)*HALF_TERRAIN_SIZE;
    gl_Position = MVP*vec4(pos.x, height, pos.y, 1);
}
```

4. Load the shaders and the corresponding uniform and attribute locations. Also, set the values of the uniforms that never change during the lifetime of the application, at initialization.

```
shader.LoadFromFile(GL_VERTEX_SHADER,
"shaders/shader.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER,
"shaders/shader.frag");
shader.CreateAndLinkProgram();
shader.Use();
    shader.AddAttribute("vVertex");
    shader.AddUniform("heightMapTexture");
    shader.AddUniform("scale");
    shader.AddUniform("half_scale");
    shader.AddUniform("HALF_TERRAIN_SIZE");
    shader.AddUniform("MVP");
    glUniform1i(shader("heightMapTexture"), 0);
    glUniform2i(shader("HALF_TERRAIN_SIZE"),
TERRAIN_WIDTH>>1, TERRAIN_DEPTH>>1);
    glUniform1f(shader("scale"), scale);
    glUniform1f(shader("half_scale"), half_scale);
shader.UnUse();
```

5. In the rendering code, set the shader and render the terrain by passing the modelview/projection matrices to the shader as shader uniforms.

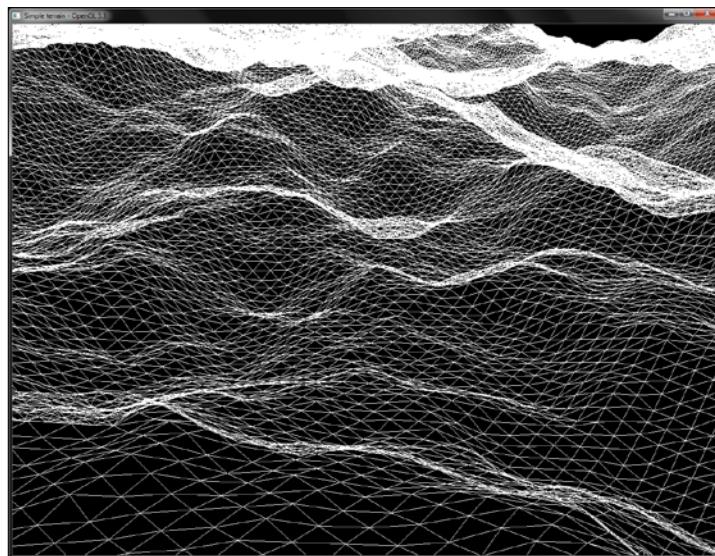
```
shader.Use();
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
glm::value_ptr(MVP));
    glDrawElements(GL_TRIANGLES, TOTAL_INDICES,
GL_UNSIGNED_INT, 0);
shader.UnUse();
```

How it works...

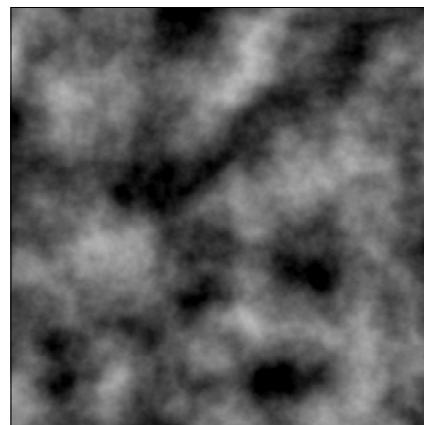
Terrain rendering is relatively straight forward to implement. The geometry is first generated on the CPU and is then stored in the GPU buffer objects. Next, the height map is loaded from an image which is then transferred to the vertex shader as a texture sampler uniform.

In the vertex shader, the height of the vertex is obtained from the height map by texture lookup using the position of the vertex. The final vertex position is obtained by combining the height with the input vertex position. The resulting vector is multiplied with the modelview projection matrix to obtain the clip space position. The vertex displacement technique can also be used to give realistic surface detail to a low resolution 3D model.

The output from the demo application for this recipe renders a wireframe terrain as shown in the following screenshot:



The height map used to generate this terrain is shown in the following screenshot:



There's more...

The method we have presented in this recipe uses the vertex displacement to generate a terrain from a height map. There are several tools available that can help with the terrain height map generation. One of them is Terragen (planetside.co.uk). Another useful tool is World Machine (<http://world-machine.com/>). A general source of information for terrains is available at the virtual terrain project (<http://vterrain.org/>).

We can also use procedural methods to generate terrains such as fractal terrain generation. Noise methods can also be helpful in the generation of the terrains.

See also

To know more about implementing terrains, you can check the following:

- ▶ *Focus on 3D Terrain Programming*, by Trent Polack, Premier Press, 2002
- ▶ *Chapter 7, Terrain Level of Detail in Level of Detail for 3D Graphics* by David Luebke, Morgan Kaufmann Publishers, 2003.

Implementing 3ds model loading using separate buffers

We will now create model loader and renderer for Autodesk® 3ds model format which is a simple yet efficient binary model format for storing digital assets.

Getting started

The code for this recipe is contained in the Chapter5/3DsViewer folder. This recipe will be using the *Drawing a 2D image in a window using a fragment shader and the SOIL image loading library* recipe from *Chapter 1, Introduction to Modern OpenGL*, for loading the 3ds mesh file's textures using the SOIL image loading library.

How to do it...

The steps required to implement a 3ds file viewer are as follows:

1. Create an instance of the C3dsLoader class. Then call the C3dsLoader::Load3DS function passing it the name of the mesh file and a set of vectors to store the submeshes, vertices, normals, uvs, indices, and materials.

```
if(!loader.Load3DS(mesh_filename.c_str(), meshes,  
vertices, normals, uvs, faces, indices, materials)) {
```

```
    cout<<"Cannot load the 3ds mesh"<<endl;
    exit(EXIT_FAILURE);
}
```

2. After the mesh is loaded, use the mesh's material list to load the material textures into the OpenGL texture object.

```
for(size_t k=0;k<materials.size();k++) {
    for(size_t m=0;m< materials[k]->textureMaps.size();m++)
    {
        GLuint id = 0;
        glGenTextures(1, &id);
        glBindTexture(GL_TEXTURE_2D, id);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
        GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
        GL_REPEAT);
        int texture_width = 0, texture_height = 0,
        channels=0;
        const string& filename = materials[k]->
        textureMaps[m]->filename;
        std::string full_filename = mesh_path;
        full_filename.append(filename);
        GLubyte* pData = SOIL_load_image
        (full_filename.c_str(), &texture_width,
        &texture_height, &channels, SOIL_LOAD_AUTO);
        if(pData == NULL) {
            cerr<<"Cannot load image: "<<
            full_filename.c_str()<<endl;
            exit(EXIT_FAILURE);
        }
        //Flip the image on Y axis
        int i,j;
        for( j = 0; j*2 < texture_height; ++j ) {
            int index1 = j * texture_width * channels;
            int index2 = (texture_height - 1 - j) *
            texture_width * channels;
            for( i = texture_width * channels; i > 0; --i ){
                GLubyte temp = pData[index1];
                pData[index1] = pData[index2];
                pData[index2] = temp;
                ++index1;
            }
        }
    }
}
```

```

        ++index2;
    }
}
GLenum format = GL_RGBA;
switch(channels) {
    case 2: format = GL_RG32UI; break;
    case 3: format = GL_RGB; break;
    case 4: format = GL_RGBA; break;
}
glTexImage2D(GL_TEXTURE_2D, 0, format, texture_width,
texture_height, 0, format, GL_UNSIGNED_BYTE, pData);
SOIL_free_image_data(pData);
textureMaps[filename]=id;
}
}
}

```

3. Pass the loaded per-vertex attributes; that is, positions (`vertices`), texture coordinates (`uvs`), per-vertex normals (`normals`), and triangle indices (`indices`) to GPU memory by allocating separate buffer objects for each attribute. Note that for easier handling of buffer objects, we bind a single vertex array object (`vaoID`) first.

```

glBindVertexArray(vaoID);
glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);
glBufferData (GL_ARRAY_BUFFER, sizeof(glm::vec3)*
vertices.size(), &(vertices[0].x), GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);
 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
GL_FALSE,0,0);
glBindBuffer (GL_ARRAY_BUFFER, vboUVsID);
glBufferData (GL_ARRAY_BUFFER,
sizeof(glm::vec2)*uvs.size(), &(uvs[0].x),
GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vUV"]);
 glVertexAttribPointer(shader["vUV"], 2,GL_FLOAT,
GL_FALSE,0, 0);
glBindBuffer (GL_ARRAY_BUFFER, vboNormalsID);
glBufferData (GL_ARRAY_BUFFER, sizeof(glm::vec3)*
normals.size(), &(normals[0].x), GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vNormal"]);
 glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT,
GL_FALSE, 0, 0);

```

4. If we have only a single material in the 3ds file, we store the face indices into `GL_ELEMENT_ARRAY_BUFFER` so that we can render the whole mesh in a single call. However, if we have more than one material, we bind the appropriate submeshes separately. The `glBufferData` call allocates the GPU memory, however, it is not initialized. In order to initialize the buffer object memory, we can use the `glMapBuffer` function to obtain a direct pointer to the GPU memory. Using this pointer, we can then write to the GPU memory. An alternative to using `glMapBuffer` is `glBufferSubData` which can modify the GPU memory by copying contents from a CPU buffer.

```
if(materials.size()==1) {  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);  
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,  
    sizeof(GLushort)*  
    3*faces.size(), 0, GL_STATIC_DRAW);  
    GLushort* pIndices = static_cast<GLushort*>(br/>    glMapBuffer(GL_ELEMENT_ARRAY_BUFFER, GL_WRITE_ONLY));  
    for(size_t i=0;i<faces.size();i++) {  
        *(pIndices++)=faces[i].a;  
        *(pIndices++)=faces[i].b;  
        *(pIndices++)=faces[i].c;  
    }  
    glUnmapBuffer(GL_ELEMENT_ARRAY_BUFFER);  
}
```

5. Set up the vertex shader to output the clip space position as well as the per-vertex texture coordinates. The texture coordinates are then interpolated by the rasterizer to the fragment shader using an output attribute `vUVout`.

```
#version 330 core  
  
layout(location = 0) in vec3 vVertex;  
layout(location = 1) in vec3 vNormal;  
layout(location = 2) in vec2 vUV;  
  
smooth out vec2 vUVout;  
  
uniform mat4 P;  
uniform mat4 MV;  
uniform mat3 N;  
  
smooth out vec3 vEyeSpaceNormal;  
smooth out vec3 vEyeSpacePosition;  
  
void main()  
{
```

```

vUVout=vUV;
vEyeSpacePosition = (MV*vec4(vVertex,1)).xyz;
vEyeSpaceNormal = N*vNormal;
gl_Position = P*vec4(vEyeSpacePosition,1);
}

```

- Set up the fragment shader, which looks up the texture map sampler with the interpolated texture coordinates from the rasterizer. Depending on whether the submesh has a texture, we linearly interpolate between the texture map color and the diffused color of the material, using the GLSL mix function.

```

#version 330 core
uniform sampler2D textureMap;
uniform float hasTexture;
uniform vec3 light_position;//light position in object
space
uniform mat4 MV;
smooth in vec3 vEyeSpaceNormal;
smooth in vec3 vEyeSpacePosition;
smooth in vec2 vUVout;

layout(location=0) out vec4 vFragColor;

const float k0 = 1.0;//constant attenuation
const float k1 = 0.0;//linear attenuation
const float k2 = 0.0;//quadratic attenuation

void main()
{
    vec4 vEyeSpaceLightPosition =
(MV*vec4(light_position,1));
    vec3 L = (vEyeSpaceLightPosition.xyz-vEyeSpacePosition);
    float d = length(L);
    L = normalize(L);
    float diffuse = max(0, dot(vEyeSpaceNormal, L));
    float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
    diffuse *= attenuationAmount;

    vFragColor = diffuse*mix(vec4(1),
texture(textureMap, vUVout), hasTexture);
}

```

- The rendering code binds the shader program, sets the shader uniforms, and then renders the mesh, depending on how many materials the 3ds mesh has. If the mesh has only a single material, it is drawn in a single call to glDrawElement by using the indices attached to the GL_ELEMENT_ARRAY_BUFFER binding point.

```

glBindVertexArray(vaoID); {
    shader.Use();
}

```

```

glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
glm::value_ptr(MV));
glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
glUniformMatrix4fv(shader("P"), 1, GL_FALSE,
glm::value_ptr(P));
glUniform3fv(shader("light_position"), 1,
&(lightPosOS.x));
if(materials.size()==1) {
    GLint whichID[1];
    glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);
    if(textureMaps.size()>0) {
        if(whichID[0] != textureMaps[
materials[0]->textureMaps[0]->filename]) {
            glBindTexture(GL_TEXTURE_2D,
textureMaps[materials[0]->textureMaps[0]
->filename]);
            glUniform1f(shader("hasTexture"), 1.0);
        }
    } else {
        glUniform1f(shader("hasTexture"), 0.0);
        glUniform3fv(shader("diffuse_color"), 1,
materials[0]->diffuse);
    }
    glDrawElements(GL_TRIANGLES, meshes[0]->faces.size()*3,
GL_UNSIGNED_SHORT, 0);
}

```

8. If the mesh contains more than one material, we iterate through the material list, and bind the texture map (if the material has one), otherwise we use the diffuse color stored in the material for the submesh. Finally, we pass the `sub_indices` array stored in the material to the `glDrawElements` function to load those indices only.

```

else {
    for(size_t i=0;i<materials.size();i++) {
        GLint whichID[1];
        glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);
        if(materials[i]->textureMaps.size()>0) {
            if(whichID[0] != textureMaps[materials[i]
->textureMaps[0]->filename]) {
                glBindTexture(GL_TEXTURE_2D, textureMaps
[materials[i]->textureMaps[0]->filename]);
            }
            glUniform1f(shader("hasTexture"), 1.0);
        } else {
            glUniform1f(shader("hasTexture"), 0.0);
        }
    }
}

```

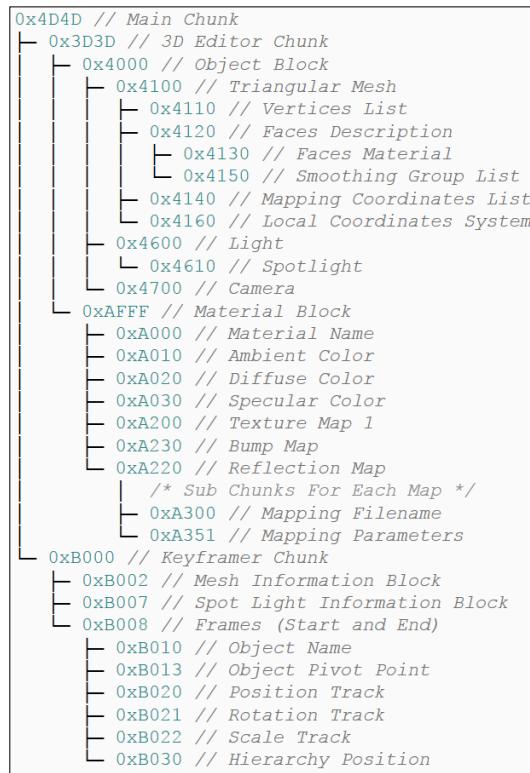
```

        glUniform3fv(shader("diffuse_color"),1,
materials[i]->diffuse);
glDrawElements(GL_TRIANGLES,
materials[i]->sub_indices.size(), GL_UNSIGNED_SHORT,
&(materials[i]->sub_indices[0]));
}
}
shader.UnUse();

```

How it works...

The main component of this recipe is the C3dsLoader::Load3DS function. The 3ds file is a binary file which is organized into a collection of chunks. Typically, a reader reads the first two bytes from the file which are stored in the chunk ID. The next four bytes store the chunk length in bytes. We continue reading chunks, and their lengths, and then store data appropriately into our vectors/variables until there are no more chunks and we pass reading the end of file. The 3ds specifications detail all of the chunks and their lengths as well as subchunks, as shown in the following figure:



Note that if there is a subchunk that we are interested in, we need to read the parent chunk as well, to move the file pointer to the appropriate offset in the file, for our required chunk. The loader first finds the total size of the 3ds mesh file in bytes. Then, it runs a while loop that checks to see if the current file pointer is within the file's size. If it is, it continues to read the first two bytes (the chunk's ID) and the next four bytes (the chunk's length).

```
while(infile.tellg() < fileSize) {  
    infile.read(reinterpret_cast<char*>(&chunk_id), 2);  
    infile.read(reinterpret_cast<char*>(&chunk_length), 4);
```

Then we start a big switch case with all of the required chunk IDs and then read the bytes from the respective chunks as desired.

```
switch(chunk_id) {  
    case 0x4d4d: break;  
    case 0x3d3d: break;  
    case 0x4000: {  
        std::string name = "";  
        char c = ' ';  
        while(c != '\0') {  
            infile.read(&c, 1);  
            name.push_back(c);  
        }  
        pMesh = new C3dsMesh(name);  
        meshes.push_back(pMesh);  
    } break;  
    ...//rest of the chunks  
}
```

All names (object name, material name, or texture map name) have to be read byte-by-byte until the null terminator character (\0) is found. For reading vertices, we first read two bytes that store the total number of vertices (N). Two bytes means that the maximum number of vertices one mesh can store is 65536. Then, we read the whole chunk of bytes, that is, `sizeof(glm::vec3) * N`, directly into our mesh's vertices, shown as follows:

```
case 0x4110: {  
    unsigned short total_vertices=0;  
    infile.read(reinterpret_cast<char*>(&total_vertices), 2);  
    pMesh->vertices.resize(total_vertices);  
    infile.read(reinterpret_cast<char*>(&pMesh->vertices[0].x),  
    sizeof(glm::vec3) * total_vertices);  
}break;
```

Similar to how the vertex information is stored, the face information stores the three unsigned short indices of the triangle and another unsigned short index containing the face flags. Therefore, for a mesh with M triangles, we have to read $4 \times M$ unsigned shorts from the file. We store the four unsigned shorts into a Face struct for convenience and then read the contents, as shown in the following code snippet:

```
case 0x4120: {
    unsigned short total_tris=0;
    infile.read(reinterpret_cast<char*>(&total_tris), 2);
    pMesh->faces.resize(total_tris);
    infile.read(reinterpret_cast<char*>(&pMesh->faces[0].a),
    sizeof(Face)*total_tris);
}break;
```

The code for reading the material face IDs and texture coordinates follows in the same way as the total entries are first read and then the appropriate number of bytes are read from the file. Note that, if a chunk has a color chunk (as for chunk IDs: 0xa010 to 0xa030), the color information is contained in a subchunk (IDs: 0x0010 to 0x0013) depending on the data type used to store the color information in the parent chunk.

After the mesh and material information is loaded, we generate global `vertices`, `uv`s, and `indices` vectors. This makes it easy for us to render the submeshes in the render function.

```
size_t total = materials.size();
for(size_t i=0;i<total;i++) {
    if(materials[i]->face_ids.size()==0)
        materials.erase(materials.begin()+i);
}

for(size_t i=0;i<meshes.size();i++) {
    for(size_t j=0;j<meshes[i]->vertices.size();j++)
        vertices.push_back(meshes[i]->vertices[j]);

    for(size_t j=0;j<meshes[i]->uv.size();j++)
        uv.push_back(meshes[i]->uv[j]);

    for(size_t j=0;j<meshes[i]->faces.size();j++) {
        faces.push_back(meshes[i]->faces[j]);
    }
}
```

Note that the 3ds format does not store the per-vertex normal explicitly. It only stores smoothing groups which tell us which faces have shared normals. After we have the vertex positions and face information, we can generate the per-vertex normals by averaging the per-face normals. This is carried out by using the following code snippet in the `3ds.cpp` file. We first allocate space for the per-vertex normals. Then we estimate the face's normal by using the cross product of the two edges. Finally, we add the face normal to the appropriate vertex index and then normalize the normal.

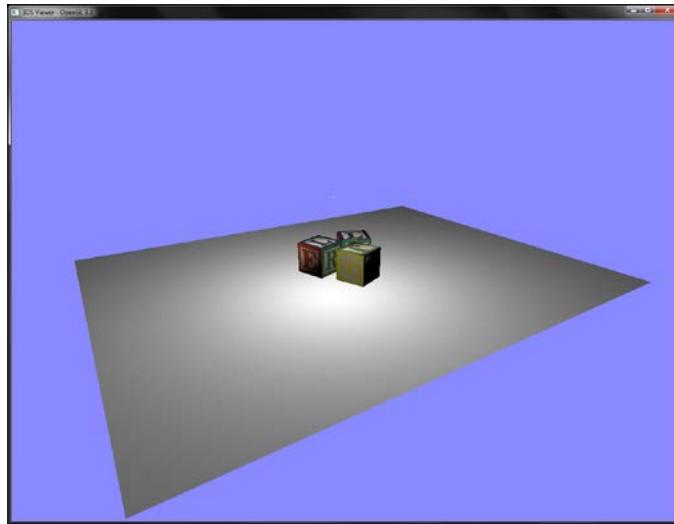
```
normals.resize(vertices.size());
for(size_t j=0;j<faces.size();j++) {
    Face f = faces[j];
    glm::vec3 v0 = vertices[f.a];
    glm::vec3 v1 = vertices[f.b];
    glm::vec3 v2 = vertices[f.c];
    glm::vec3 e1 = v1 - v0;
    glm::vec3 e2 = v2 - v0;
    glm::vec3 N = glm::cross(e1,e2);
    normals[f.a] += N;
    normals[f.b] += N;
    normals[f.c] += N;
}
for(size_t i=0;i<normals.size();i++) {
    normals[i]=glm::normalize(normals[i]);
}
```

Once we have all the per-vertex attributes and faces information, we use this to group the triangles by material. We loop through all of the materials and expand their face IDs to include the three vertex IDs and make the face.

```
for(size_t i=0;i<materials.size();i++) {
    Material* pMat = materials[i];
    for(int j=0;j<pMat->face_ids.size();j++) {
        pMat->sub_indices.push_back(faces[pMat->face_ids[j]].a);
        pMat->sub_indices.push_back(faces[pMat->face_ids[j]].b);
        pMat->sub_indices.push_back(faces[pMat->face_ids[j]].c);
    }
}
```

There's more...

The output from the demo application implementing this recipe is given in the following figure. In this recipe, we render three blocks on a quad plane. The camera position can be changed using the left mouse button. The point light source position can be changed using the right mouse button. Each block has six textures attached to it, whereas the plane has no texture, hence it uses the diffuse color value.



Note that the 3ds loader shown in this recipe does not take smoothing groups into consideration. For a more robust loader, we recommend the lib3ds library which provides a more elaborate 3ds file loader with support for smoothing groups, animation tracks, cameras, lights, keyframes, and so on.

See also

For more information on implementing 3ds model loading, you can refer to the following links:

- ▶ Lib3ds: <http://code.google.com/p/lib3ds/>
- ▶ 3ds file loader by Damiano Vitulli: http://www.spacesimulator.net/wiki/index.php?title=Tutorials:3ds_Loader
- ▶ 3ds file format details on Wikipedia.org: <http://en.wikipedia.org/wiki/.3ds>

Implementing OBJ model loading using interleaved buffers

In this recipe we will implement the Wavefront® OBJ model. Instead of using separate buffer objects for storing positions, normals, and texture coordinates as in the previous recipe, we will use a single buffer object with interleaved data. This ensures that we have more chances of a cache hit since related attributes are stored next to each other in the buffer object memory.

Getting started

The code for this recipe is contained in the Chapter5/ObjViewer folder.

How to do it...

Let us start the recipe by following these simple steps:

1. Create a global reference of the `ObjLoader` object. Call the `ObjLoader::Load` function, passing it the name of the OBJ file. Pass vectors to store the meshes, vertices, indices, and materials contained in the OBJ file.

```
ObjLoader obj;
if(!obj.Load(mesh_filename.c_str(), meshes, vertices,
indices, materials)) {
    cout<<"Cannot load the 3ds mesh"<<endl;
    exit(EXIT_FAILURE);
}
```

2. Generate OpenGL texture objects for each material using the SOIL library if the material has a texture map.

```
for(size_t k=0;k<materials.size();k++) {
    if(materials[k]->map_Kd != "") {
        GLuint id = 0;
        glGenTextures(1, &id);
        glBindTexture(GL_TEXTURE_2D, id);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
        GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
        GL_REPEAT);
```

```
    int texture_width = 0, texture_height = 0,
channels=0;
const string& filename = materials[k]->map_Kd;
std::string full_filename = mesh_path;
full_filename.append(filename);

GLubyte* pData =
SOIL_load_image(full_filename.c_str(),
&texture_width, &texture_height, &channels,
SOIL_LOAD_AUTO);
if(pData == NULL) {
    cerr<<"Cannot load image:
"<<full_filename.c_str()<<endl;
    exit(EXIT_FAILURE);
}
//... image flipping code
GLenum format = GL_RGBA;
switch(channels) {
    case 2: format = GL_RG32UI; break;
    case 3: format = GL_RGB; break;
    case 4: format = GL_RGBA; break;
}
glTexImage2D(GL_TEXTURE_2D, 0, format, texture_width,
texture_height, 0, format, GL_UNSIGNED_BYTE, pData);
SOIL_free_image_data(pData);
textures.push_back(id);
}
```

3. Set up shaders and generate buffer objects to store the mesh file data in the GPU memory. The shader setup is similar to the previous recipes.

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);
glBindVertexArray(vaoID);
glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
glBufferData(GL_ARRAY_BUFFER,
sizeof(Vertex)*vertices.size(),
&(vertices[0].pos.x), GL_STATIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);
glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
GL_FALSE,sizeof(Vertex),0);

 glEnableVertexAttribArray(shader["vNormal"]);
```

```

glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT,
GL_FALSE, sizeof(Vertex), (const GLvoid*)(offsetof( Vertex,
normal)) ) ;

glEnableVertexAttribArray(shader["vUV"]);
glVertexAttribPointer(shader["vUV"], 2, GL_FLOAT,
GL_FALSE, sizeof(Vertex), (const
GLvoid*)(offsetof(Vertex, uv)) );
if(materials.size()==1) {
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
    sizeof(GLushort)*indices.size(), &(indices[0]),
    GL_STATIC_DRAW);
}

```

4. Bind the vertex array object associated with the mesh, use the shader and pass the shader uniforms, that is, the modelview (MV), projection (P), normal matrices (N) and light position, and so on.

```

glBindVertexArray(vaoID) {
    shader.Use();
    glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
    glm::value_ptr(MV));
    glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
    glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
    glUniformMatrix4fv(shader("P"), 1, GL_FALSE,
    glm::value_ptr(P));
    glUniform3fv(shader("light_position"),1,
    &(lightPosOS.x));
}

```

5. To draw the mesh/submesh, loop through all of the materials in the mesh and then bind the texture to the GL_TEXTURE_2D target if the material contains a texture map. Otherwise, use a default color for the mesh. Finally, call the `glDrawElements` function to render the mesh/submesh.

```

for(size_t i=0;i<materials.size();i++) {
    Material* pMat = materials[i];
    if(pMat->map_Kd != "") {
        glUniform1f(shader("useDefault"), 0.0);
        GLint whichID[1];
        glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);
        if(whichID[0] != textures[i])
            glBindTexture(GL_TEXTURE_2D, textures[i]);
    }
    else
        glUniform1f(shader("useDefault"), 1.0);
}

```

```
if(materials.size()==1)
    glDrawElements(GL_TRIANGLES, indices.size(),
    GL_UNSIGNED_SHORT, 0);
else
    glDrawElements(GL_TRIANGLES, pMat->count,
    GL_UNSIGNED_SHORT, (const GLvoid*)(& indices
    [pMat->offset]));
}
shader.UnUse();
```

How it works...

The main component of this recipe is the `ObjLoader::Load` function defined in the `Obj.cpp` file. The Wavefront® OBJ file is a text file which has different text descriptors for different mesh components. Usually, the mesh starts with the geometry definition, that is, vertices that begin with the letter `v` followed by three floating point values. If there are normals, their definitions begin with `vn` followed by three floating point values. If there are texture coordinates, their definitions begin with `vt`, followed by two floating point values. Comments start with the `#` character, so whenever a line with this character is encountered, it is ignored.

Following the geometry definition, the topology is defined. In this case, the line is prefixed with `f` followed by the indices for the polygon vertices. In case of a triangle, three indices sections are given such that the vertex position indices are given first, followed by texture coordinates indices (if any), and finally the normal indices (if any). Note that the indices start from 1, not 0.

So, for example, say that we have a quad geometry having four position indices (1,2,3,4) having four texture coordinate indices (5,6,7,8), and four normal indices (1,1,1,1) then the topology would be stored as follows:

```
f 1/5/1 2/6/1 3/7/1 4/8/1
```

If the mesh is a triangular mesh with position vertices (1,2,3), texture coordinates (7,8,9), and normals (4,5,6) then the topology would be stored as follows:

```
f 1/7/4 2/8/5 3/9/6
```

Now, if the texture coordinates are omitted from the first example, then the topology would be stored as follows:

```
f 1//1 2//1 3//1 4//1
```

The OBJ file stores material information in a separate material (.mtl) file. This file contains similar text descriptors that define different materials with their ambient, diffuse, and specular color values, texture maps, and so on. The details of the defined elements are given in the OBJ format specifications. The material file for the current OBJ file is declared using the `mtllib` keyword followed by the name of the .mtl file. Usually, the .mtl file is stored in the same folder as the OBJ file. A polygon definition is preceded with a `usemtl` keyword followed by the name of the material to use for the upcoming polygon definition. Several polygonal definitions can be grouped using the `g` or `o` prefix followed by the name of the group/object respectively.

The `ObjLoader::Load` function first finds the current prefix. Then, the code branches to the appropriate section depending on the prefix. The suffix strings are then parsed and the extracted data is stored in the corresponding vectors. For efficiency, rather than storing the indices directly, we store them by material so that we can then sort and render the mesh by material. The associated material library file (.mtl) is loaded using the `ReadMaterialLibrary` function. Refer to the `Obj.cpp` file for details.

The file parsing is the first piece of the puzzle. The second piece is the transfer of this data to the GPU memory. In this recipe, we use an interleaved buffer, that is, instead of storing each per-vertex attribute separately in its own vertex buffer object, we store them interleaved one after the other in a single buffer object. First positions are followed by normals and then texture coordinates. We achieve this by first defining our vertex format using a custom `Vertex` struct. Our vertices are a vector of this struct.

```
struct Vertex {  
    glm::vec3 pos, normal;  
    glm::vec2 uv;  
};
```

We generate the vertex array object and then the vertex buffer object. Next, we bind the buffer object passing it our vertices. In this case, we specify the stride of each attribute in the data stream separately as follows:

```
glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);  
glBufferData (GL_ARRAY_BUFFER, sizeof(Vertex)*vertices.size(),  
&(vertices[0].pos.x), GL_STATIC_DRAW);  
 glEnableVertexAttribArray(shader["vVertex"]);  
 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,  
 GL_FALSE,sizeof(Vertex),0);  
 glEnableVertexAttribArray(shader["vNormal"]);  
 glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT, GL_FALSE,  
 sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, normal)) );  
 glEnableVertexAttribArray(shader["vUV"]);  
 glVertexAttribPointer(shader["vUV"], 2, GL_FLOAT, GL_FALSE,  
 sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, uv)) );
```

If the mesh has a single material, we store the mesh indices into a `GL_ELEMENT_ARRAY_BUFFER` target. Otherwise, we render the submeshes by material.

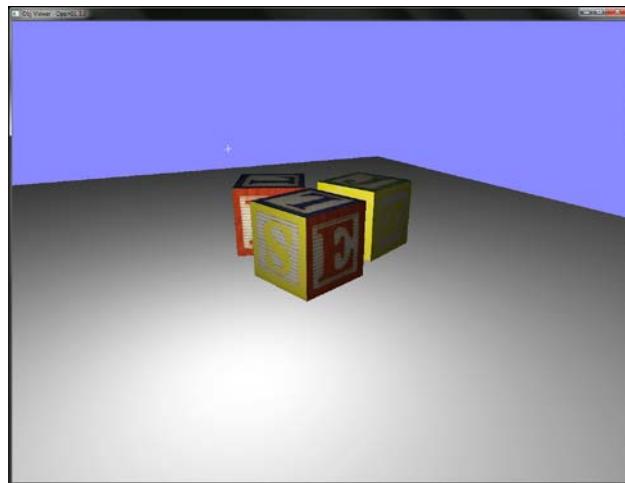
```
if(materials.size() == 1) {  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndicesID);  
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort) *  
        indices.size(), &(indices[0]), GL_STATIC_DRAW);  
}
```

At the time of rendering, if we have a single material, we render the whole mesh, otherwise we render the subset stored with the material.

```
if(materials.size() == 1)  
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_SHORT, 0);  
else  
    glDrawElements(GL_TRIANGLES, pMat->count, GL_UNSIGNED_SHORT,  
        (const GLvoid*)(&indices[pMat->offset]));
```

There's more...

The demo application implementing this recipe shows a scene with three blocks on a planar quad. The camera view can be rotated with the left mouse button. The light source's position is shown by a 3D crosshair that can be moved by dragging the right mouse button. The output from this demo application is shown in the following figure:



See also

You can see the OBJ file specification on Wikipedia at http://en.wikipedia.org/wiki/Wavefront_.obj_file.

Implementing EZMesh model loading

In this recipe, we will learn how to load and render an EZMesh model. There are several skeletal animation formats such as Quake's md2 (.md2), Autodesk® FBX (.fbx), and Collada (.dae). The conventional model formats such as Collada are overly complicated for doing simple skeletal animation. Therefore, in this recipe, we will learn how to load and render an EZMesh (.ezm) skeletal model.

Getting started

The code for this recipe is contained in the Chapter5/EZMeshViewer directory. For this recipe, we will be using two external libraries to aid with the EZMesh (.ezm) mesh file parsing. The first library is called MeshImport and it can be downloaded from <http://code.google.com/p/meshimport/>. Make sure to get the latest svn trunk of the code. After downloading, change directory to the compiler subdirectory which contains the visual studio solution files. Double-click to open the solution and build the project dlls. After the library is built successfully, copy MeshImport_[x86/x64].dll and MeshImportEZM_[x86/x64].dll (subject to your machine configuration) into your current project directory. In addition, also copy the MeshImport.[h/cpp] files which contain some useful library loading routines.

In addition, since EZMesh is an XML format to support loading of textures, we parse the EZMesh XML manually with the help of the pugixml library. You can download it from <http://pugixml.org/downloads/>. As pugixml is tiny, we can directly include the source files with the project.

How to do it...

Let us start this recipe by following these simple steps:

1. Create a global reference to an EzmLoader object. Call the EzmLoader::Load function passing it the name of the EZMesh (.ezm) file. Pass the vectors to store the submeshes, vertices, indices, and materials-to-image map. The Load function also accepts the min and max vectors to store the EZMesh bounding box.

```
if (!ezm.Load(mesh_filename.c_str(), submeshes, vertices,
    indices, material2ImageMap, min, max)) {
    cout<<"Cannot load the EZMesh mesh"<<endl;
    exit(EXIT_FAILURE);
}
```

-
2. Using the material information, generate the OpenGL textures for the EZMesh geometry.

```
for(size_t k=0;k<materialNames.size();k++) {  
    GLuint id = 0;  
    glGenTextures(1, &id);  
    glBindTexture(GL_TEXTURE_2D, id);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                    GL_LINEAR);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
                    GL_LINEAR);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  
                    GL_REPEAT);  
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  
                    GL_REPEAT);  
    int texture_width = 0, texture_height = 0, channels=0;  
    const string& filename = materialNames[k];  
  
    std::string full_filename = mesh_path;  
    full_filename.append(filename);  
  
    //Image loading using SOIL and vertical image flipping  
    //...  
    GLenum format = GL_RGBA;  
    switch(channels) {  
        case 2: format = GL_RG32UI; break;  
        case 3: format = GL_RGB; break;  
        case 4: format = GL_RGBA; break;  
    }  
    glTexImage2D(GL_TEXTURE_2D, 0, format, texture_width,  
    texture_height, 0, format, GL_UNSIGNED_BYTE, pData);  
    SOIL_free_image_data(pData);  
    materialMap[filename] = id ;  
}
```

3. Set up the interleaved buffer object as in the previous recipe, *Implementing OBJ model loading using interleaved buffers*.

```
glBindVertexArray(vaoID);  
glBindBuffer (GL_ARRAY_BUFFER, vboVerticesID);  
glBufferData (GL_ARRAY_BUFFER,  
sizeof(Vertex)*vertices.size(),  
&(vertices[0].pos.x), GL_DYNAMIC_DRAW);  
  
glEnableVertexAttribArray(shader["vVertex"]);  
glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,  
GL_FALSE,sizeof(Vertex),0);
```

```
glEnableVertexAttribArray(shader["vNormal"]);
glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT,
GL_FALSE, sizeof(Vertex), (const GLvoid*)
(offsetof(Vertex, normal)) );

glEnableVertexAttribArray(shader["vUV"]);
glVertexAttribPointer(shader["vUV"], 2, GL_FLOAT,
GL_FALSE, sizeof(Vertex), (const GLvoid*)
(offsetof(Vertex, uv)) );
```

4. To render the EZMesh, bind the mesh's vertex array object, set up the shader, and pass the shader uniforms.

```
glBindVertexArray(vaoID) {
    shader.Use();
    glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
    glm::value_ptr(MV));
    glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
    glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
    glUniformMatrix4fv(shader("P"), 1, GL_FALSE,
    glm::value_ptr(P));
    glUniform3fv(shader("light_position"), 1,
    &(lightPosES.x));
```

5. Loop through all submeshes, bind the submesh texture, and then issue the `glDrawElements` call, passing it the submesh indices. If the submesh has no materials, a default solid color material is assigned to the submesh.

```
for(size_t i=0;i<submeshes.size();i++) {
    if(strlen(submeshes[i].materialName)>0) {
        GLuint id = materialMap[material2ImageMap[
        submeshes[i].materialName]];

        GLint whichID[1];
        glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);

        if(whichID[0] != id)
            glBindTexture(GL_TEXTURE_2D, id);
        glUniform1f(shader("useDefault"), 0.0);
    } else {
        glUniform1f(shader("useDefault"), 1.0);
    }
    glDrawElements(GL_TRIANGLES,
    submeshes[i].indices.size(),
    GL_UNSIGNED_INT, &submeshes[i].indices[0]);
}
```

How it works...

EZMesh is an XML based skeletal animation format. There are two parts to this recipe: parsing of the EZMesh file using the MeshImport/pugixml libraries and handling of the data using OpenGL buffer objects. The first part is handled by the EzmLoader::Load function. Along with the filename, this function accepts vectors to store the submeshes, vertices, indices, and material names map contained in the mesh file.

If we open an EZMesh file, it contains a collection of XML elements. The first element is MeshSystem. This element contains four child elements: Skeletons, Animations, Materials, and Meshes. Each of these subelements has a count attribute that stores the total number of corresponding items in the EZMesh file. Note that we can remove the element as desired. So the hierarchy is typically as follows:

```
<MeshSystem>
    <Skeletons count="N">
    <Animations count="N">
    <Materials count="N">
    <Meshes count="N">
</MeshSystem>
```

For this recipe, we are interested in the last two subelements: Materials and Meshes. We will be using the first two subelements in the skeletal animation recipe in a later chapter of this book. Each Materials element has a counted number of Material elements. Each Material element stores the material's name in the name attribute and the material's details. For example, the texture map file name in the meta_data attribute. In the EzmLoader::Load function, we use pugi_xml to parse the Materials element and its subelements into a material map. This map stores the material's name and its texture file name. Note that the MeshImport library does provide functions for reading material information, but they are broken.

```
pugi::xml_node mats = doc.child("MeshSystem").child("Materials");
int totalMaterials = atoi(mats.attribute("count").value());
pugi::xml_node material = mats.child("Material");
for(int i=0;i<totalMaterials;i++) {
    std::string name = material.attribute("name").value();
    std::string metadata = material.attribute("meta_data").value();
    //clean up metadata
    int len = metadata.length();
    if(len>0) {
        string fullName="";
        int index = metadata.find_last_of("\\\\");
        if(index == string::npos) {
            fullName.append(metadata);
        } else {
```

```

        std::string fileNameOnly = metadata.substr(index+1,
            metadata.length());
        fullName.append(fileNameOnly);
    }
    bool exists = true;
    if(materialNames.find(name) == materialNames.end())
        exists = false;
    if(!exists)
        materialNames[name] = (fullName);
    material = material.next_sibling("Material");
}
}
}

```

After the material information is loaded in, we initialize the MeshImport library by calling the NVSHARE::loadMeshImporters function and passing it the directory where MeshImport dlls (MeshImport_[x86,x64].dll and MeshImportEJM_[x86,x64].dll) are placed. Upon success, this function returns the NVSHARE::MeshImport library object. Using the MeshImport library object, we first create the mesh system container by calling the NVSHARE::MeshImport::createMeshSystemContainer function. This function accepts the object name and the EZMesh file contents. If successful, this function returns the MeshSystemContainer object which is then passed to the NVSHARE::MeshImport::getMeshSystem function which returns the NVSHARE::MeshSystem object. This represents the MeshSystem node in the EZMesh XML file.

Once we have the MeshSystem object, we can query all of the subelements. These reside in the MeshSystem object as member variables. So let's say we want to traverse through all of the meshes in the current EZMesh file and copy the per-vertex attributes to our own vector (`vertices`), we would simply do the following:

```

for(size_t i=0;i<ms->mMeshCount;i++) {
    NVSHARE::Mesh* pMesh = ms->mMeshes[i];
    vertices.resize(pMesh->mVertexCount);
    for(size_t j=0;j<pMesh->mVertexCount;j++) {
        vertices[j].pos.x = pMesh->mVertices[j].mPos[0];
        vertices[j].pos.y = pMesh->mVertices[j].mPos[1];
        vertices[j].pos.z = pMesh->mVertices[j].mPos[2];

        vertices[j].normal.x = pMesh->mVertices[j].mNormal[0];
        vertices[j].normal.y = pMesh->mVertices[j].mNormal[1];
        vertices[j].normal.z = pMesh->mVertices[j].mNormal[2];

        vertices[j].uv.x = pMesh->mVertices[j].mTexel1[0];
        vertices[j].uv.y = pMesh->mVertices[j].mTexel1[1];
    }
}
}

```

In an EZMesh file, the indices are sorted by materials into submeshes. We iterate through all of the submeshes and then store their material name and indices into our container.

```
submeshes.resize(pMesh->mSubMeshCount);
for(size_t j=0;j<pMesh->mSubMeshCount;j++) {
    NVSHARE::SubMesh* pSubMesh = pMesh->mSubMeshes[j];
    submeshes[j].materialName = pSubMesh->mMaterialName;
    submeshes[j].indices.resize(pSubMesh->mTriCount * 3);
    memcpy(&(submeshes[j].indices[0]), pSubMesh->mIndices,
           sizeof(unsigned int) * pSubMesh->mTriCount * 3);
}
```

After the EZMesh file is parsed and we have the per-vertex data stored, we first generate the OpenGL textures from the EZMesh materials list. Then we store the texture IDs into a material map so that we can refer to the textures by material name.

```
for(size_t k=0;k<materialNames.size();k++) {
    GLuint id = 0;
    glGenTextures(1, &id);
    glBindTexture(GL_TEXTURE_2D, id);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    int texture_width = 0, texture_height = 0, channels=0;
    const string& filename = materialNames[k];
    std::string full_filename = mesh_path;
    full_filename.append(filename);
    GLubyte* pData = SOIL_load_image(full_filename.c_str(),
                                     &texture_width, &texture_height, &channels, SOIL_LOAD_AUTO);
    if(pData == NULL) {
        cerr<<"Cannot load image: "<<full_filename.c_str()<<endl;
        exit(EXIT_FAILURE);
    }
    //... Flip the image on Y axis and determine the image format
    glTexImage2D(GL_TEXTURE_2D, 0, format, texture_width,
                 texture_height, 0, format, GL_UNSIGNED_BYTE, pData);
    SOIL_free_image_data(pData);
    materialMap[filename] = id ;
}
```

After the materials, the shaders are loaded as in the previous recipes. The per-vertex data is then transferred to the GPU using vertex array and vertex buffer objects. In this case, we use the interleaved vertex buffer format.

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);

 glBindVertexArray(vaoID);
 glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
 glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*vertices.size(),
 &(vertices[0].pos.x), GL_DYNAMIC_DRAW);
 glEnableVertexAttribArray(shader["vVertex"]);

 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT,
 GL_FALSE, sizeof(Vertex), 0);
 glEnableVertexAttribArray(shader["vNormal"]);
 glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT, GL_FALSE,
 sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, normal)));
 glEnableVertexAttribArray(shader["vUV"]);
 glVertexAttribPointer(shader["vUV"], 2, GL_FLOAT, GL_FALSE,
 sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, uv)) );
```

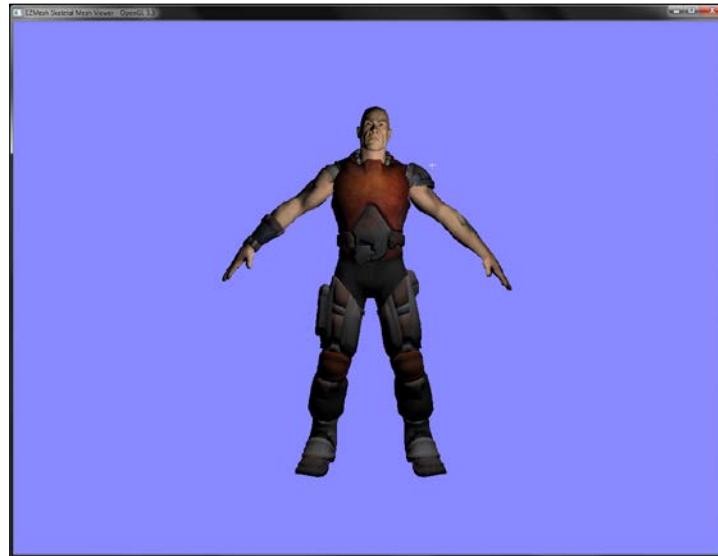
For rendering of the mesh, we first bind the vertex array object of the mesh, attach our shader and pass the shader uniforms. Then we loop over all of the submeshes and bind the appropriate texture (if the submesh has texture). Otherwise, a default color is used. Finally, the indices of the submesh are used to draw the mesh using the `glDrawElements` function.

```
glBindVertexArray(vaoID);
 shader.Use();
 glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
 glm::value_ptr(MV));
 glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
 glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
 glUniformMatrix4fv(shader("P"), 1, GL_FALSE,
 glm::value_ptr(P));
 glUniform3fv(shader("light_position"), 1, &(lightPosES.x));
 for(size_t i=0;i<submeshes.size();i++) {
 if(strlen(submeshes[i].materialName)>0) {
 GLuint id =
 materialMap[material2ImageMap[submeshes[i].materialName]];
 GLint whichID[1];
 glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);
 if(whichID[0] != id)
 glBindTexture(GL_TEXTURE_2D, id);
```

```
    glUniform1f(shader("useDefault"), 0.0);
} else {
    glUniform1f(shader("useDefault"), 1.0);
}
glDrawElements(GL_TRIANGLES, submeshes[i].indices.size(),
GL_UNSIGNED_INT, &submeshes[i].indices[0]);
}
shader.UnUse();
}
```

There's more...

The demo application implementing this recipe renders a skeletal model with textures. The point light source can be moved by dragging the right mouse button. The output result is shown in the following figure:



See also

You can also see John Ratcliff's code repository: A test application for MeshImport library and showcasing EZMesh at <http://codesuppository.blogspot.sg/2009/11/test-application-for-meshimport-library.html>.

Implementing simple particle system

In this recipe, we will implement a simple particle system. Particle systems are a special category of objects that enable us to simulate fuzzy effects in computer graphics; for example, fire or smoke. In this recipe, we will implement a simple particle system that emits particles at the specified rate from an oriented emitter. In this recipe, we will assign particles with a basic fire color map without texture, to give the effect of fire.

Getting started

The code for this recipe is contained in the `Chapter5/SimpleParticles` directory. All of the work for particle simulation is carried out in the vertex shader.

How to do it...

Let us start this recipe by following these simple steps:

1. Create a vertex shader without any per-vertex attribute. The vertex shader generates the current particle position and outputs a smooth color to the fragment shader for use as the current fragment color.

```
#version 330 core
smooth out vec4 vSmoothColor;
uniform mat4 MVP;
uniform float time;

const vec3 a = vec3(0,2,0);      //acceleration of particles
//vec3 g = vec3(0,-9.8,0);    // acceleration due to gravity

const float rate = 1/500.0;     //rate of emission
const float life = 2;           //life of particle

//constants
const float PI = 3.14159;
const float TWO_PI = 2*PI;

//colormap colours
const vec3 RED = vec3(1,0,0);
const vec3 GREEN = vec3(0,1,0);
const vec3 YELLOW = vec3(1,1,0);

//pseudorandom number generator
float rand(vec2 co){
```

```

        return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) *
43758.5453);
    }

//pseudorandom direction on a sphere
vec3 uniformRadomDir(vec2 v, out vec2 r) {
    r.x = rand(v.xy);
    r.y = rand(v.yx);
    float theta = mix(0.0, PI / 6.0, r.x);
    float phi = mix(0.0, TWO_PI, r.y);
    return vec3(sin(theta) * cos(phi), cos(theta), sin(theta)
* sin(phi));
}

void main() {
    vec3 pos=vec3(0);
    float t = gl_VertexID*rate;
    float alpha = 1;
    if(time>t) {
        float dt = mod((time-t), life);
        vec2 xy = vec2(gl_VertexID,t);
        vec2 rdm=vec2(0);
        pos = ((uniformRadomDir(xy, rdm) + 0.5*a*dt)*dt);
        alpha = 1.0 - (dt/life);
    }
    vSmoothColor = vec4(mix(RED,YELLOW,alpha),alpha);
    gl_Position = MVP*vec4(pos,1);
}

```

2. The fragment shader outputs the smooth color as the current fragment output color.

```

#version 330 core
smooth in vec4 vSmoothColor;

layout(location=0) out vec4 vFragColor;

void main() {
    vFragColor = vSmoothColor;
}

```

3. Set up a single vertex array object and bind it.

```

glGenVertexArrays(1, &vaoID);
 glBindVertexArray(vaoID);

```

4. In the rendering code, set up the shader and pass the shader uniforms. For example, pass the current time to the `time` shader uniform and the combined modelview projection matrix (MVP). Here we add an emitter transform matrix (`emitterXForm`) to the combined MVP matrix that controls the orientation of our particle emitter.

```
shader.Use();
glUniform1f(shader("time"), time);
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
glm::value_ptr(P*MV*emitterXForm));
```

5. Finally, we render the total number of particles (`MAX_PARTICLES`) with a call to the `glDrawArrays` function and unbind our shader.

```
glDrawArrays(GL_POINTS, 0, MAX_PARTICLES);
shader.UnUse();
```



Versions of OpenGL prior to OpenGL 3 provided a special particle type called `GL_POINT_SPRITE`. In OpenGL 3.3 and above core profiles, the `GL_POINT_SPRITE` enum has been deprecated. Hence, now `GL_POINTS` acts as point sprites by default.

How it works...

The entire code from generation of particle positions to assignment of colors and forces is carried out in the vertex shader. In this recipe, we do not store any per-vertex attribute as in the previous recipes. Instead, we simply invoke the `glDrawArrays` call with the number of particles (`MAX_PARTICLES`) we need to render. This calls our vertex shader for each particle in turn.

We have two uniforms in the vertex shader, the combined modelview projection matrix (MVP) and the current simulation time (`time`). The other variables required for particle simulation are stored as shader constants.

```
#version 330
smooth out vec4 vSmoothColor;
uniform mat4 MVP;
uniform float time;
const vec3 a = vec3(0,2,0);           //acceleration of particles
//vec3 g = vec3(0,-9.8,0);           //acceleration due to gravity
const float rate = 1/500.0;          //rate of emission of particles
const float life = 2;                //particle life
const float PI = 3.14159;
const float TWO_PI = 2*PI;
const vec3 RED = vec3(1,0,0);
const vec3 GREEN = vec3(0,1,0);
const vec3 YELLOW = vec3(1,1,0);
```

In the main function, we calculate the current particle time (t) by multiplying its vertex ID (`gl_VertexID`) with the emission rate (`rate`). The `gl_VertexID` attribute is a unique integer identifier associated with each vertex. We then check the current time (`time`) against the particle's time (t). If it is greater, we calculate the time step amount (`dt`) and then calculate the particle's position using a simple kinematics formula.

```
void main() {
    vec3 pos=vec3(0);
    float t = gl_VertexID*rate;
    float alpha = 1;
    if(time>t) {
```

To generate the particle, we need to have its initial velocity. This is generated on the fly by using a pseudorandom generator with the vertex ID and time as the seeds using the function `uniformRandomDir` which is defined as follows:

```
//pseudorandom number generator
float rand(vec2 co){
    return fract(sin(dot(co.xy ,vec2(12.9898,78.233))) *
    43758.5453);
}
//pseudorandom direction on a sphere
vec3 uniformRadomDir(vec2 v, out vec2 r) {
    r.x = rand(v.xy);
    r.y = rand(v.yx);
    float theta = mix(0.0, PI / 6.0, r.x);
    float phi = mix(0.0, TWO_PI, r.y);
    return vec3(sin(theta) * cos(phi), cos(theta), sin(theta) *
    sin(phi));
}
```

The particle's position is then calculated using the current time and the random initial velocity. To enable respawning, we use the modulus operator (`mod`) of the difference between the particle's time and the current time (`time-t`) with the life of particle (`life`). After calculation of the position, we calculate the particle's alpha to gently fade it when its life is consumed.

```
float dt = mod((time-t), life);
vec2 xy = vec2(gl_VertexID,t);
vec2 rdm;
pos = ((uniformRadomDir(xy, rdm) + 0.5*a*dt)*dt);
alpha = 1.0 - (dt/life);
}
```

The alpha value is used to linearly interpolate between red and yellow colors by calling the GLSL `mix` function to give the fire effect. Finally, the generated position is multiplied with the combined modelview projection (MVP) matrix to get the clip space position of the particle.

```
vSmoothColor = vec4(mix(RED,YELLOW,alpha),alpha);  
gl_Position = MVP*vec4(pos,1);  
}
```

The fragment shader simply uses the `vSmoothColor` output variable from the vertex shader as the current fragment color.

```
#version 330 core  
smooth in vec4 vSmoothColor;  
layout(location=0) out vec4 vFragColor;  
void main()  
{  
    vFragColor = vSmoothColor;  
}
```

Extending to textured billboarded particles requires us to change only the fragment shader. The point sprites provide a varying `gl_PointCoord` that can be used to sample a texture in the fragment shader as shown in the textured particle fragment shader (Chapter5/SimpleParticles/shaders/textured.frag).

```
#version 330 core  
smooth in vec4 vSmoothColor;  
layout(location=0) out vec4 vFragColor;  
uniform sampler2D textureMap;  
void main()  
{  
    vFragColor = texture(textureMap, gl_PointCoord) *  
    vSmoothColor.a;  
}
```

The application loads a particle texture and generates an OpenGL texture object from it.

```
GLubyte* pData = SOIL_load_image(texture_filename.c_str(),  
&texture_width, &texture_height, &channels, SOIL_LOAD_AUTO);  
if(pData == NULL) {  
    cerr<<"Cannot load image: "<<texture_filename.c_str()<<endl;  
    exit(EXIT_FAILURE);  
}  
//Flip the image on Y axis  
int i,j;  
for( j = 0; j*2 < texture_height; ++j )  
{  
    int index1 = j * texture_width * channels;
```

```
int index2 = (texture_height - 1 - j)*texture_width* channels;
for( i = texture_width * channels; i > 0; --i )
{
    GLubyte temp = pData[index1];
    pData[index1] = pData[index2];
    pData[index2] = temp;
    ++index1;
    ++index2;
}
GLenum format = GL_RGBA;
switch(channels) {
    case 2: format = GL_RG32UI; break;
    case 3: format = GL_RGB;   break;
    case 4: format = GL_RGBA;  break;
}

 glGenTextures(1, &textureID);
 glBindTexture(GL_TEXTURE_2D, textureID);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
 glTexImage2D(GL_TEXTURE_2D, 0, format, texture_width,
 texture_height, 0, format, GL_UNSIGNED_BYTE, pData);
 SOIL_free_image_data(pData);
```

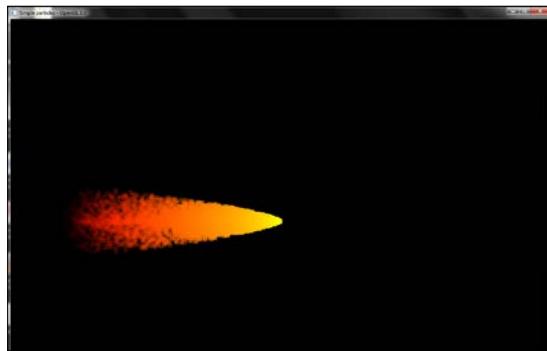
Next, the texture unit to which the texture is bound is passed to the shader.

```
texturedShader.LoadFromFile(GL_VERTEX_SHADER,
"shaders/shader.vert");
texturedShader.LoadFromFile(GL_FRAGMENT_SHADER,
"shaders/textured.frag");
texturedShader.CreateAndLinkProgram();
texturedShader.Use();
texturedShader.AddUniform("MVP");
texturedShader.AddUniform("time");
texturedShader.AddUniform("textureMap");
glUniform1i(texturedShader("textureMap"), 0);
texturedShader.UnUse();
```

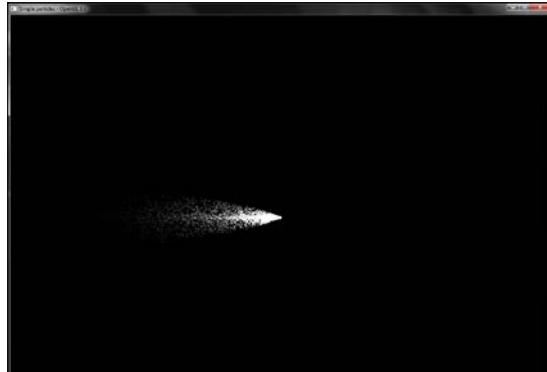
Finally, the particles are rendered using the `glDrawArrays` call as shown earlier.

There's more...

The demo application for this recipe renders a particle system to simulate fire emitting from a point emitter as would typically come out from a rocket's exhaust. We can press the space bar key to toggle display of textured particles. The current view can be rotated and zoomed by dragging the left and middle mouse buttons respectively. The output result from the demo is displayed in the following figure:



If the textured particles shader is used, we get the following output:

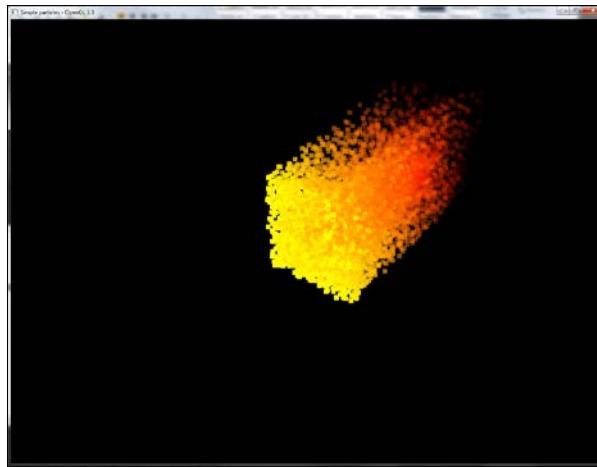


The orientation and position of the emitter is controlled using the emitter transformation matrix (`emitterXForm`). We can change this matrix to reorient/reposition the particle system in the 3D space.

The shader code given in the previous subsection generates a particle system from a point emitter source. If we want to change the source to a rectangular emitter, we can replace the position calculation with the following shader code snippet:

```
pos = ( uniformRandomDir(xy, rdm) + 0.5*a*dt)*dt;
vec2 rect = (rdm*2.0 - 1.0);
pos += vec3(rect.x, 0, rect.y) ;
```

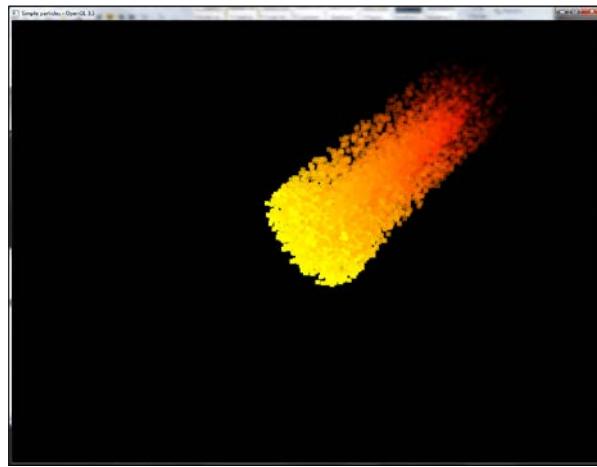
This gives the following output:



Changing the emitter to a disc shape further filters the points spawned in the rectangle emitter by only accepting those which lie inside the circle of a given radius, as given in the following code snippet:

```
pos = ( uniformRandomDir(xy, rdm) + 0.5*a*dt)*dt;
vec2 rect = (rdm*2.0 - 1.0);
float dotP = dot(rect, rect);
if(dotP<1)
    pos += vec3(rect.x, 0, rect.y);
```

Using this position calculation gives a disc emitter as shown in the following output:



We can also add additional forces such as air drag, wind, vortex, and so on, by simply adding to the acceleration or velocity component of the particle system. Another option could be to direct the emitter to a specific path such as a b-spline. We could also add deflectors to deflect the generated particles or create particles that spawn other particles as is typically used in a fireworks particle system. Particle systems are an extremely interesting area in computer graphics which help us obtain wonderful effects easily.

The recipe detailed here shows how to do a very simple particle system entirely on the GPU. While such a particle system might be useful for basic effects, more detailed effects would need more elaborate treatment as detailed in the references in the See also section.

See also

To know more about detailed effects you can refer to the following links:

- ▶ Real-time particle systems on the GPU in Dynamic Environment SIGGRAPH 2007 Talk: http://developer.amd.com.wordpress/media/2012/10/Drone-Real-Time_Particles_Systems_on_the_GPU_in_Dynamic_Environments%28Siggraph07%29.pdf
- ▶ GPU Gems 3 Chapter 23-High speed offscreen particles: http://http://developer.nvidia.com/GPUGems3/gpugems3_ch23.html
- ▶ Building a million particle system by Lutz Latta: http://www.gamasutra.com/view/feature/130535/building_a_millionparticle_system.php?print=1
- ▶ CG Tutorial chapter 6: http://http://developer.nvidia.com/CgTutorial/cg_tutorial_chapter06.html

6

GPU-based Alpha Blending and Global Illumination

In this chapter, we will focus on:

- ▶ Implementing order-independent transparency using front-to-back peeling
- ▶ Implementing order-independent transparency with dual depth peeling
- ▶ Implementing screen space ambient occlusion (SSAO)
- ▶ Implementing global illumination using spherical harmonics lighting
- ▶ Implementing GPU-based ray tracing
- ▶ Implementing GPU-based path tracing

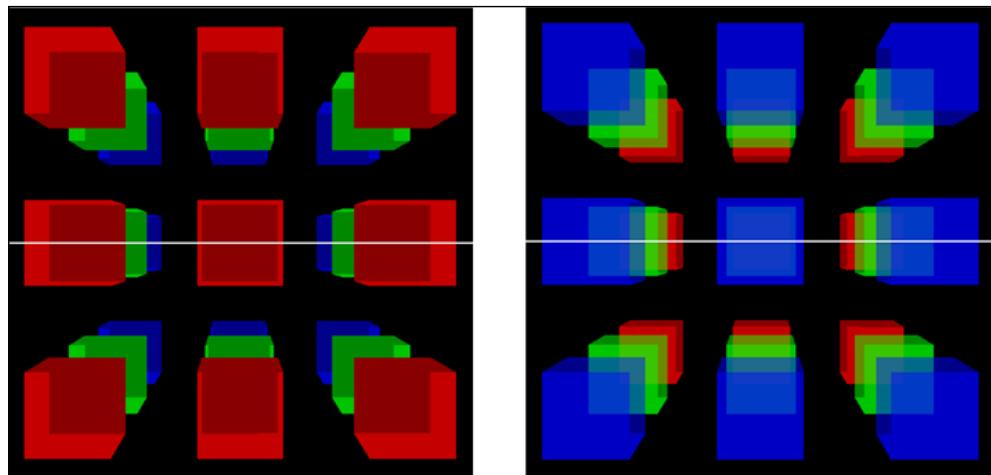
Introduction

Even with the introduction of lighting, our virtual objects don't look and feel real. This is because our lights are a simple approximation of the reflection behavior of the surface. There is a specific category of algorithms that help bridge the gap between the real-world lighting and the virtual-world lighting. These are called **global illumination methods**. Although these methods had been proven to be expensive to evaluate in real time, new methods have been proposed that fake the global illumination using clever techniques. One such technique is spherical harmonics lighting that uses HDR light probes to light a virtual scene having no light source. The idea is to extract the lighting information from the light probe and give a feeling that the virtual objects are in the same environment.

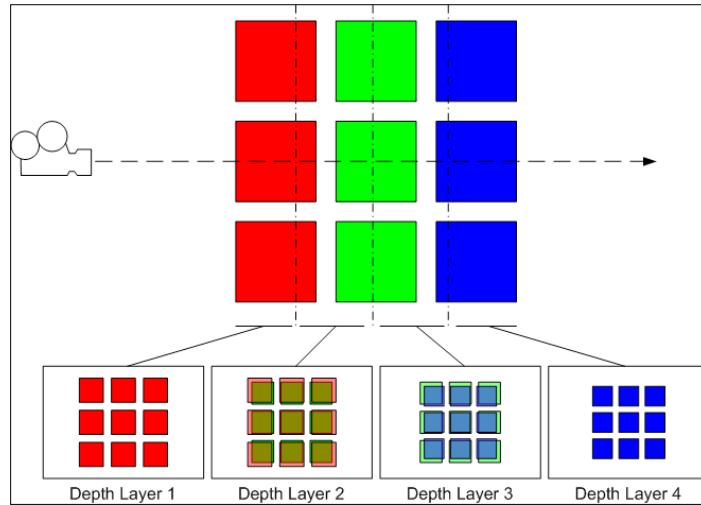
In addition, rendering of transparent geometry is also problematic since this requires sorting of geometry in the depth order. If the scene complexity increases, it becomes not only difficult to maintain the depth order, but the processing overhead also increases. To circumvent these scenarios and handle the alpha blending for order-independent transparency of the 3D geometry efficiently, we implement depth peeling and the more efficient dual depth peeling, on the modern GPU. All of these techniques will be implemented in OpenGL 3.3 core profile.

Implementing order-independent transparency using front-to-back peeling

When we have to render translucent geometry, for example, a glass window in a graphics application, care has to be taken to make sure that the geometry is properly rendered in the depth order such that the opaque objects in the scene are rendered first and the transparent objects are rendered last. This unfortunately incurs additional overhead where the CPU is busy sorting objects. In addition, the blending result will be correct only from a specific viewing direction, as shown in the following figure. Note that the image on the left is the result if we view from the direction of the Z axis. There is no blending at all in the left image. If the same scene is viewed from the opposite side, we can see the correct alpha blending result.



Depth peeling (also called front-to-back peeling) is one technique that helps in this process. In this technique, the scene is rendered in slices in such a way that slices are rendered one after another from front to back until the whole object is processed, as shown in the following figure, which is a 2D side view of the same scene as in the previous figure.



The number of layers to use for peeling is dependent on the depth complexity of the scene. This recipe will show how to implement this technique in modern OpenGL.

Getting ready

The code for this recipe is contained in the `Chapter6/FrontToBackPeeling` directory.

How to do it...

Let us start our recipe by following these simple steps:

1. Set up two frame buffer objects (FBOs) with two color and depth attachments. For this recipe, we will use rectangle textures (`GL_TEXTURE_RECTANGLE`) since they enable easier handling of images (samplers) in the fragment shader. With rectangle textures we can access texture values using pixel positions directly. In case of normal texture (`GL_TEXTURE_2D`), we have to normalize the texture coordinates.

```
glGenFramebuffers(2, fbo);
glGenTextures (2, texID);
glGenTextures (2, depthTexID);
for(int i=0;i<2;i++) {
    glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[i]);
    //set texture parameters like minification etc.
```

```

glTexImage2D(GL_TEXTURE_RECTANGLE, 0,
GL_DEPTH_COMPONENT32F, WIDTH, HEIGHT, 0,
GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glBindTexture(GL_TEXTURE_RECTANGLE, texID[i]);
//set texture parameters like minification etc.
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, WIDTH,
HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glBindFramebuffer(GL_FRAMEBUFFER, fbo[i]);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_TEXTURE_RECTANGLE,
depthTexID[i], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_RECTANGLE,
texID[i], 0);
}
glGenTextures(1, &colorBlenderTexID);
glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID);
//set texture parameters like minification etc.
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, WIDTH,
HEIGHT, 0, GL_RGBA, GL_FLOAT, 0);

```

- Set another FBO for color blending and check the FBO for completeness. The color blending FBO uses the depth texture from the first FBO as a depth attachment, as it uses the depth output from the first step during blending.

```

glGenFramebuffers(1, &colorBlenderFBOID);
glBindFramebuffer(GL_FRAMEBUFFER, colorBlenderFBOID);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_TEXTURE_RECTANGLE,
depthTexID[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_RECTANGLE,
colorBlenderTexID, 0);
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE)
    printf("FBO setup successful !!! \n");
else
    printf("Problem with FBO setup");

glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

- In the rendering function, set the color blending FBO as the current render target and then render the scene normally with depth testing enabled.

```

glBindFramebuffer(GL_FRAMEBUFFER, colorBlenderFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
 glEnable(GL_DEPTH_TEST);
DrawScene(MVP, cubeShader);

```

4. Next, bind the other FBO pair alternatively, clear the render target, and enable depth testing, but disable alpha blending. This is to render the nearest surface in the offscreen render target. The number of passes dictate the number of layers the given geometry is peeled into. The more the number of passes, the more continuous the depth peeling result. For the demo in this recipe, the number of passes is set as 6. The number of passes is dependent on the depth complexity of the scene. If the user wants to check the number of samples output from the depth peeling step, then based on the value of the flag (`bUseOQ`) an occlusion query is used to find the number of samples output from the depth peeling step.

```

int numLayers = (NUM_PASSES - 1) * 2;
for (int layer = 1; bUseOQ || layer < numLayers; layer++) {
    int currId = layer % 2;
    int prevId = 1 - currId;
    glBindFramebuffer(GL_FRAMEBUFFER, fbo[currId]);
    glDrawBuffer(GL_COLOR_ATTACHMENT0);
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    if (bUseOQ) {
        glBeginQuery(GL_SAMPLES_PASSED_ARB, queryId);
    }
}

```

5. Bind the depth texture from the first step so that the nearest fragment can be used with the attached shaders and then render the scene with the front peeling shaders. Refer to Chapter6/FrontToBackPeeling/shaders/front_peel.{vert,frag} for details. We then end the hardware query if the query was initiated.

```

glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[prevId]);
DrawScene(MVP, frontPeelShader);
if (bUseOQ) {
    glEndQuery(GL_SAMPLES_PASSED_ARB);
}

```

6. Bind the color blender FBO again, disable depth testing, and enable additive blending; however, specify separate blending so that the color and alpha can be blended separately. Finally, bind the rendered output from step 5 and then using a full-screen quad and the blend shader (Chapter6/FrontToBackPeeling/shaders/blend.{vert,frag}), blend the whole scene.

```

glBindFramebuffer(GL_FRAMEBUFFER, colorBlenderFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glDisable(GL_DEPTH_TEST);
 glEnable(GL_BLEND);
 glBlendEquation(GL_FUNC_ADD);

```

```

glBlendFuncSeparate(GL_DST_ALPHA, GL_ONE, GL_ZERO,
GL_ONE_MINUS_SRC_ALPHA);
glBindTexture(GL_TEXTURE_RECTANGLE, texID[currId]);
blendShader.Use();
DrawFullScreenQuad();
blendShader.UnUse();
glDisable(GL_BLEND);

```

7. In the final step, restore the default draw buffer (GL_BACK_LEFT) and disable alpha blending and depth testing. Use a full-screen quad and a final shader (Chapter6/FrontToBackPeeling/shaders/final.frag) to blend the output from the color blending FBO.

```

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
glDisable(GL_DEPTH_TEST);
glDisable(GL_BLEND);

glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID);
finalShader.Use();
glUniform4fv(finalShader("vBackgroundColor"), 1,
&bg.x);
DrawFullScreenQuad();
finalShader.UnUse();

```

How it works...

The front-to-back depth peeling works in three steps. First, the scene is rendered normally on a depth FBO with depth testing enabled. This ensures that the scene depth values are stored in the depth attachment of the FBO. In the second pass, we bind the depth FBO, bind the depth texture from the first step, and then iteratively clip parts of the geometry by using a fragment shader (see Chapter6/FrontToBackPeeling/shaders/front_peel.frag) as shown in the following code snippet:

```

#version 330 core
layout(location = 0) out vec4 vFragColor;
uniform vec4 vColor;
uniform sampler2DRect depthTexture;
void main() {
    float frontDepth = texture(depthTexture, gl_FragCoord.xy).r;
    if(gl_FragCoord.z <= frontDepth)
        discard;
    vFragColor = vColor;
}

```

This shader simply compares the incoming fragment's depth against the depth value stored in the depth texture. If the current fragment's depth is less than or equal to the depth in the depth texture, the fragment is discarded. Otherwise, the fragment color is output.

```
float frontDepth = texture(depthTexture, gl_FragCoord.xy).r;
if(gl_FragCoord.z <= frontDepth)
    discard;
```

After this step, we bind the color blend FBO, disable depth test, and then enable alpha blending with separate blending of colors and alpha values. The `glBlendFunctionSeparate` function is used here as it enables us to handle color and alpha channels for source and destination separately. The first parameter is the source RGB, which is assigned the alpha value of the pixel in the frame buffer. This blends the incoming fragment with the existing color in the frame buffer. The second parameter, that is, the destination RGB, is set as `GL_ONE`, which keeps the value in the destination as is. The third parameter is set as `GL_ZERO`, which removes the source alpha component as we already applied the alpha from the destination using the first parameter. The final parameter, that is, the destination alpha is set as the conventional over-compositing alpha value (`GL_ONE_MINUS_SRC_ALPHA`).

We then bind the texture from the previous step output and then use the blend shader (see `Chapter6/FrontToBackPeeling/shaders/blend.frag`) on a full-screen quad to alpha blend the current fragments with the existing fragments on the frame buffer. The blend shader is defined as follows:

```
#version 330 core
uniform sampler2DRect tempTexture;
layout(location = 0) out vec4 vFragColor;
void main()
{
    vFragColor = texture(tempTexture, gl_FragCoord.xy);
}
```

The `tempTexture` sampler contains the output from the depth peeling step stored in the `colorBlenderFBO` attachment. After this step, the alpha blending is disabled, as shown in the code snippet in step 6 of the *How to do it...* section.

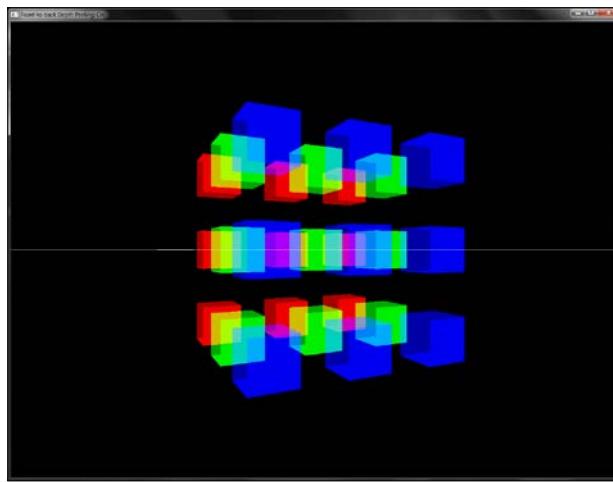
In the final step, the default draw buffer is restored, depth testing and alpha blending is disabled, and the final output from the color blend FBO is blended with the background color using a simple fragment shader. The code snippet is as shown in step 7 of the *How to do it...* section. The final fragment shader is defined as follows:

```
#version 330 core
uniform sampler2DRect colorTexture;
uniform vec4 vBackgroundColor;
layout(location = 0) out vec4 vFragColor;
void main()
{
    vec4 color = texture(colorTexture, gl_FragCoord.xy);
    vFragColor = color + vBackgroundColor*color.a;
}
```

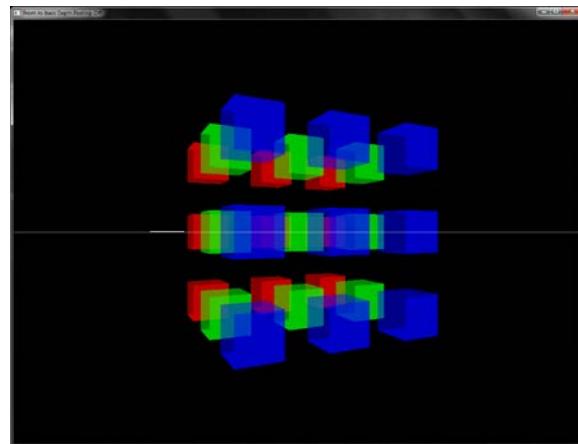
The final shader takes the front peeled result and blends it with the background color using the alpha value from the front peeled result. This way rather than taking the nearest depth fragment all fragments are taken into consideration showing a correctly blended result.

There's more...

The output from the demo application for this recipe renders 27 translucent cubes at the origin. The camera position can be changed using the left mouse button. The front-to-back depth peeling gives the following output. Note the blended color, for example, the yellow color where the green boxes overlay the red ones.



Pressing the Space bar disables front-to-back peeling so that we can see the normal alpha blending without back-to-front sorting which gives the following output. Note that we do not see the yellow blended color where the green and red boxes overlap.



Even though the output produced by front-to-back peeling is correct, it requires multiple passes through the geometry that incur additional processing overhead. The next recipe details the more robust method called **dual depth peeling** which tackles this problem.

See also

- ▶ *Interactive Order-Independent Transparency*, Cass Everitt: <http://gamedevs.org/uploads/interactive-order-independent-transparency.pdf>

Implementing order-independent transparency using dual depth peeling

In this recipe, we will implement dual depth peeling. The main idea behind this method is to peel two depth layers at the same time. This results in a much better performance with the same output, as dual depth peeling peels two layers at a time; one from the front and one from the back.

Getting ready

The code for this recipe is contained in the Chapter6/DualDepthPeeling folder.

How to do it...

The steps required to implement dual depth peeling are as follows:

1. Create an FBO and attach six textures in all: two for storing the front buffer, two for storing the back buffer, and two for storing the depth buffer values.

```
glGenFramebuffers(1, &dualDepthFBOID);
glGenTextures (2, texID);
glGenTextures (2, backTexID);
glGenTextures (2, depthTexID);
for(int i=0;i<2;i++) {
    glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[i]);
    //set texture parameters
    glTexImage2D(GL_TEXTURE_RECTANGLE , 0, GL_FLOAT_RG32_NV,
    WIDTH, HEIGHT, 0, GL_RGB, GL_FLOAT, NULL);
    glBindTexture(GL_TEXTURE_RECTANGLE,texID[i]);
    //set texture parameters
    glTexImage2D(GL_TEXTURE_RECTANGLE , 0, GL_RGBA, WIDTH,
    HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
    glBindTexture(GL_TEXTURE_RECTANGLE,backTexID[i]);
    //set texture parameters
```

```

glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, WIDTH,
HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
}

```

2. Bind the six textures to the appropriate attachment points on the FBO.

```

glBindFramebuffer(GL_FRAMEBUFFER, dualDepthFBOID);
for(int i=0;i<2;i++) {
    glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i],
GL_TEXTURE_RECTANGLE, depthTexID[i], 0);
    glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i]+1,
GL_TEXTURE_RECTANGLE, texID[i], 0);
    glFramebufferTexture2D(GL_FRAMEBUFFER, attachID[i]+2,
GL_TEXTURE_RECTANGLE, backTexID[i], 0);
}

```

3. Create another FBO for color blending and attach a new texture to it. Also attach this texture to the first FBO and check the FBO completeness.

```

glGenTextures(1, &colorBlenderTexID);
glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID);
//set texture parameters
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, WIDTH,
HEIGHT, 0, GL_RGBA, GL_FLOAT, 0);
glGenFramebuffers(1, &colorBlenderFBOID);
glBindFramebuffer(GL_FRAMEBUFFER, colorBlenderFBOID);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_RECTANGLE,
colorBlenderTexID, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT6, GL_TEXTURE_RECTANGLE,
colorBlenderTexID, 0);
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE )
    printf("FBO setup successful !!! \n");
else
    printf("Problem with FBO setup");
glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

4. In the render function, first disable depth testing and enable blending and then bind the depth FBO. Initialize and clear DrawBuffer to write on the render target attached to GL_COLOR_ATTACHMENT1 and GL_COLOR_ATTACHMENT2.

```

glDisable(GL_DEPTH_TEST);
 glEnable(GL_BLEND);
 glBindFramebuffer(GL_FRAMEBUFFER, dualDepthFBOID);
 glDrawBuffers(2, &drawBuffers[1]);
 glClearColor(0, 0, 0, 0);
 glClear(GL_COLOR_BUFFER_BIT);

```

5. Next, set `GL_COLOR_ATTACHMENT0` as the draw buffer, enable min/max blending (`glBlendEquation(GL_MAX)`), and initialize the color attachment using fragment shader (see `Chapter6/DualDepthPeeling/shaders/dual_init.frag`). This completes the first step of dual depth peeling, that is, initialization of the buffers.

```
glDrawBuffer(drawBuffers[0]);
glClearColor(-MAX_DEPTH, -MAX_DEPTH, 0, 0);
glClear(GL_COLOR_BUFFER_BIT);
glBlendEquation(GL_MAX);
DrawScene(MVP, initShader);
```

6. Next, set `GL_COLOR_ATTACHMENT6` as the draw buffer and clear it with background color. Then, run a loop that alternates two draw buffers and then uses min/max blending. Then draw the scene again.

```
glDrawBuffer(drawBuffers[6]);
glClearColor(bg.x, bg.y, bg.z, bg.w);
glClear(GL_COLOR_BUFFER_BIT);
int numLayers = (NUM_PASSES - 1) * 2;
int currId = 0;
for (int layer = 1; bUseOQ || layer < numLayers; layer++) {
    currId = layer % 2;
    int prevId = 1 - currId;
    int bufId = currId * 3;
    glDrawBuffers(2, &drawBuffers[bufId+1]);
    glClearColor(0, 0, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawBuffer(drawBuffers[bufId+0]);
    glClearColor(-MAX_DEPTH, -MAX_DEPTH, 0, 0);
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawBuffers(3, &drawBuffers[bufId+0]);
    glBlendEquation(GL_MAX);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[prevId]);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_RECTANGLE, texID[prevId]);
    DrawScene(MVP, dualPeelShader, true, true);
```

7. Finally, enable additive blending (`glBlendFunc(GL_FUNC_ADD)`) and then draw a full screen quad with the blend shader. This peels away fragments from the front as well as the back layer of the rendered geometry and blends the result on the current draw buffer.

```
glDrawBuffer(drawBuffers[6]);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
if (bUseOQ) {
```

```

    glBeginQuery(GL_SAMPLES_PASSED_ARB, queryId) ;
}
glActiveTexture(GL_TEXTURE0) ;
 glBindTexture(GL_TEXTURE_RECTANGLE, backTexID[currId]) ;
blendShader.Use() ;
 DrawFullScreenQuad() ;
blendShader.UnUse() ;
}

```

8. In the final step, we unbind the FBO and enable rendering on the default back buffer (GL_BACK_LEFT). Next, we bind the outputs from the depth peeling and blending steps to their appropriate texture location. Finally, we use a final blending shader to combine the two peeled and blended fragments.

```

glBindFramebuffer(GL_FRAMEBUFFER, 0) ;
glDrawBuffer(GL_BACK_LEFT) ;
 glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID) ;
glActiveTexture(GL_TEXTURE0) ;
 glBindTexture(GL_TEXTURE_RECTANGLE, depthTexID[currId]) ;
glActiveTexture(GL_TEXTURE1) ;
 glBindTexture(GL_TEXTURE_RECTANGLE, texID[currId]) ;
glActiveTexture(GL_TEXTURE2) ;
 glBindTexture(GL_TEXTURE_RECTANGLE, colorBlenderTexID) ;
finalShader.Use() ;
 DrawFullScreenQuad() ;
finalShader.UnUse() ;

```

How it works...

Dual depth peeling works in a similar fashion as the front-to-back peeling. However, the difference is in the way it operates. It peels away depths from both the front and the back layer at the same time using min/max blending. First, we initialize the fragment depth values using the fragment shader (Chapter6/DualDepthPeeling/shaders/dual_init.frag) and min/max blending.

```
vFragColor.xy = vec2(-gl_FragCoord.z, gl_FragCoord.z);
```

This initializes the blending buffers. Next, a loop is run but instead of peeling depth layers front-to-back, we first peel back depths and then the front depths. This is carried out in the fragment shader (Chapter6/DualDepthPeeling/shaders/dual_peel.frag) along with max blending.

```

float fragDepth = gl_FragCoord.z;
vec2 depthBlender = texture(depthBlenderTex, gl_FragCoord.xy).xy;
vec4 forwardTemp = texture(frontBlenderTex, gl_FragCoord.xy);
//initialize variables ...

```

```
if (fragDepth < nearestDepth || fragDepth > farthestDepth) {
    vFragColor0.xy = vec2(-MAX_DEPTH);
    return;
}
if(fragDepth > nearestDepth && fragDepth < farthestDepth) {
    vFragColor0.xy = vec2(-fragDepth, fragDepth);
    return;
}
vFragColor0.xy = vec2(-MAX_DEPTH);

if (fragDepth == nearestDepth) {
    vFragColor1.xyz += vColor.rgb * alpha * alphaMultiplier;
    vFragColor1.w = 1.0 - alphaMultiplier * (1.0 - alpha);
} else {
    vFragColor2 += vec4(vColor.rgb, alpha);
}
```

The blend shader (Chapter6/DualDepthPeeling/shaders/blend.frag) simply discards fragments whose alpha values are zero. This ensures that the occlusion query is not incremented, which would give a wrong number of samples than the actual fragment used in the depth blending.

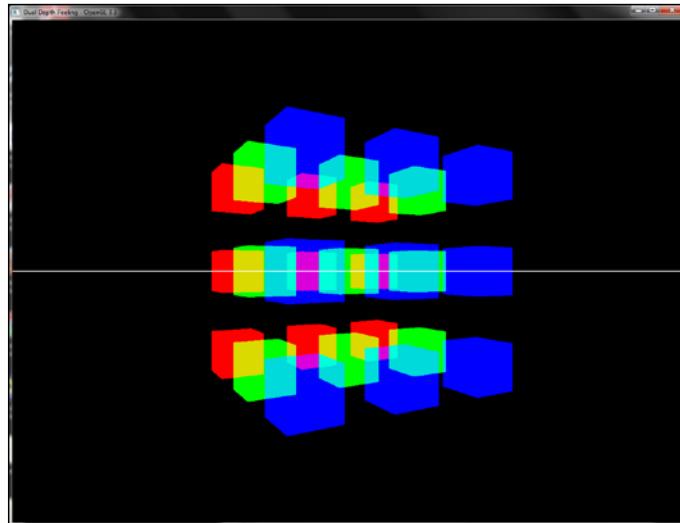
```
vFragColor = texture(tempTexture, gl_FragCoord.xy);
if(vFragColor.a == 0)
    discard;
```

Finally, the last blend shader (Chapter6/DualDepthPeeling/shaders/final.frag) takes the blended fragments from the front and back blend textures and blends the results to get the final fragment color.

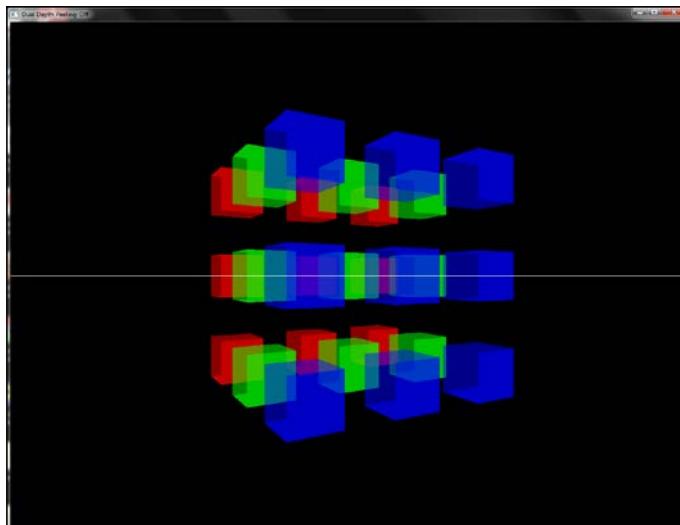
```
vec4 frontColor = texture(frontBlenderTex, gl_FragCoord.xy);
vec3 backColor = texture(backBlenderTex, gl_FragCoord.xy).rgb;
vFragColor.rgb = frontColor.rgb + backColor * frontColor.a;
```

There's more...

The demo application for this demo is similar to the one shown in the previous recipe. If dual depth peeling is enabled, we get the result as shown in the following figure:



Pressing the Space bar enables/disables dual depth peeling. If dual peeling is disabled, the result is as follows:



See also

- ▶ Louis Bavoil and Kevin Myers, *Order Independent Transparency with Dual Depth Peeling*
demo in NVIDIA OpenGL 10 sdk: http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf

Implementing screen space ambient occlusion (SSAO)

We have implemented simple lighting recipes in previous chapters. These unfortunately approximate some aspects of lighting. However, effects such as global illumination are not handled by the basic lights, as discussed earlier. In this respect, several techniques have been developed over the years which fake the global illumination effects. One such technique is **Screen Space Ambient Occlusion (SSAO)** which we will implement in this recipe.

As the name suggests, this method works in screen space. For any given pixel onscreen, the amount of occlusion due to its neighboring pixels can be obtained by looking at the difference in their depth value. In order to reduce the sampling artefacts, the neighbor coordinates are randomly offset. For a pixel whose depth values are close to one another, they belong to the geometry which is spatially lying close. Based on the difference of the depth values, an occlusion value is determined. Given in pseudocode, the algorithm may be given as follows:

```

Get the position (p), normal (n) and depth (d) value at current pixel
position
For each pixel in the neighborhood of current pixel
    Get the position (p0) of the neighborhood pixel
    Call proc. CalcAO(p, p0, n)
End for
Return the ambient occlusion amount as color

```

The ambient occlusion procedure is defined as follows:

```

const float DEPTH_TOLERANCE = 0.00001;
proc CalcAO(p,p0,n)
    diff = p0-p-DEPTH_TOLERANCE;
    v = normalize(diff);
    d = length(diff)*scale;
    return max(0.1, dot(n,v)-bias)*(1.0/(1.0+d))*intensity;
end proc

```

Note that we have three artist control parameters: scale, bias, and intensity. The scale parameter controls the size of the occlusion area, bias shifts the occlusion, and intensity controls the strength of the occlusion. The DEPTH_TOLERANCE constant is added to remove depth-fighting artefacts.

The whole recipe proceeds as follows. We load our 3D model and render it into an offscreen texture using FBO. We use two FBOs: one for storing the eye space normals and depth, and another FBO is for filtering of intermediate results. For both the color attachment and the depth attachment of first FBO, floating point texture formats are used. For the color attachment, `GL_RGBA32F` is used, whereas for depth texture, the `GL_DEPTH_COMPONENT32F` floating point format is used. Floating point texture formats are used as we require more precision, otherwise truncation errors will show up in the rendering result. The second FBO is used for separable Gaussian smoothing as was carried out in the *Implementing variance shadow mapping* recipe in *Chapter 4, Lights and Shadows*. This FBO has two color attachments with the floating point texture format `GL_RGBA32F`.

In the rendering function, the scene is first rendered normally. Then, the first shader is used to output the eye space normals. This is stored in the color attachment and the depth values are stored in the depth attachment of the first FBO. After this step, the filtering FBO is bound and the second shader is used, which uses the depth and normal textures from the first FBO to calculate the ambient occlusion result. Since the neighbor points are randomly offset, noise is introduced. The noisy result is then smoothed by applying separable gaussian smoothing. Finally, the filtered result is blended with the existing rendering by using conventional alpha blending.

Getting ready

The code for this recipe is contained in the `Chapter6/SSAO` folder. We will be using the Obj model viewer from *Chapter 5, Mesh Model Formats and and Particle Systems*. We will add SSAO to the Obj model.

How to do it...

Let us start the recipe by following these simple steps:

1. Create a global reference of the `ObjLoader` object. Call the `ObjLoader::Load` function passing it the name of the OBJ file. Pass vectors to store the `meshes`, `vertices`, `indices`, and `materials` contained in the OBJ file.
2. Create a framebuffer object (FBO) with two attachments: first to store the scene normals and second to store the depth. We will use a floating point texture format (`GL_RGBA32F`) for both of these. In addition, we create a second FBO for Gaussian smoothing of the SSAO output. We are using multiple texture units here as the second shader expects normal and depth textures to be bound to texture units 1 and 3 respectively.

```
glGenFramebuffers(1, &fboID);
 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glGenTextures(1, &normalTextureID);
 glGenTextures(1, &depthTextureID);
 glActiveTexture(GL_TEXTURE1);
```

```

glBindTexture(GL_TEXTURE_2D, normalTextureID);
//set texture parameters
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, WIDTH, HEIGHT,
0, GL_BGRA, GL_FLOAT, NULL);
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_2D, depthTextureID);
//set texture parameters
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32F,
WIDTH, HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, normalTextureID, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT,GL_TEXTURE_2D, depthTextureID, 0);
glGenFramebuffers(1,&filterFBOID);
glBindFramebuffer(GL_FRAMEBUFFER,filterFBOID);
glGenTextures(2, blurTexID);
glGenTextures(2, blurTexID);
for(int i=0;i<2;i++) {
    glActiveTexture(GL_TEXTURE4+i);
    glBindTexture(GL_TEXTURE_2D, blurTexID[i]);
    //set texture parameters
    glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA32F,RTT_WIDTH,
RTT_HEIGHT,0,GL_RGBA,GL_FLOAT,NULL);
    glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0+i,GL_TEXTURE_2D,blurTexID[i],0);
}

```

3. In the render function, render the scene meshes normally. After this step, bind the first FBO and then use the first shader program. This program takes the per-vertex positions/normals of the mesh and outputs the view space normals from the fragment shader.

```

glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glViewport(0,0,RTT_WIDTH, RTT_HEIGHT);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBindVertexArray(vaoID);
ssaoFirstShader.Use();
glUniformMatrix4fv(ssaoFirstShader("MVP"), 1, GL_FALSE,
glm::value_ptr(P*MV));
glUniformMatrix3fv(ssaoFirstShader("N"), 1, GL_FALSE,
glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
for(size_t i=0;i<materials.size();i++) {
Material* pMat = materials[i];
if(materials.size()==1)
glDrawElements(GL_TRIANGLES, indices.size(),
GL_UNSIGNED_SHORT, 0);
}

```

```

    else
        glDrawElements(GL_TRIANGLES, pMat->count,
                      GL_UNSIGNED_SHORT, (const GLvoid*)(&indices
                      [pMat->offset]));
    }
    ssaoFirstShader.UnUse();
}

```

The first vertex shader (`Chapter6/SSAO/shaders/SSAO_FirstStep.vert`) outputs the eye space normal as shown in the following code snippet:

```

#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;
uniform mat4 MVP;
uniform mat3 N;
smooth out vec3 vEyeSpaceNormal;
void main() {
    vEyeSpaceNormal = N*vNormal;
    gl_Position = MVP*vec4(vVertex,1);
}

```

The fragment shader (`Chapter6/SSAO/shaders/SSAO_FirstStep.frag`) returns the interpolated normal, as the fragment color, shown as follows:

```

#version 330 core
smooth in vec3 vEyeSpaceNormal;
layout(location=0) out vec4 vFragColor;
void main() {
    vFragColor = vec4(normalize(vEyeSpaceNormal)*0.5 + 0.5,
                      1);
}

```

4. Bind the filtering FBO and use the second shader (`Chapter6/SSAO/shaders/SSAO_SecondStep.frag`). This shader does the actual SSAO calculation. The input to the shader is the normals texture from step 3. This shader is invoked on a full screen quad.

```

glBindFramebuffer(GL_FRAMEBUFFER, filterFBOID);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glBindVertexArray(quadVAOID);
ssaoSecondShader.Use();
glUniform1f(ssaoSecondShader("radius"), sampling_radius);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
ssaoSecondShader.UnUse();

```

5. Filter the output from step 4 by using separable Gaussian convolution using two fragment shaders (Chapter6/SSAO/shaders/GaussH.frag and Chapter6/SSAO/shaders/GaussV.frag). The separable Gaussian smoothing is added in to smooth out the ambient occlusion result.

```
glDrawBuffer(GL_COLOR_ATTACHMENT1);
glBindVertexArray(quadVAOID);
gaussianV_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
gaussianH_shader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
```

6. Unbind the filtering FBO, reset the default viewport, and then the default draw buffer. Enable alpha blending and then use the final shader (Chapter6/SSAO/shaders/final.frag) to blend the output from steps 3 and 5. This shader simply renders the final output from the filtering stage using a full-screen quad.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0, 0, WIDTH, HEIGHT);
glDrawBuffer(GL_BACK_LEFT);
 glEnable(GL_BLEND);
 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
finalShader.Use();
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0);
finalShader.UnUse();
glDisable(GL_BLEND);
```

How it works...

There are three steps in the SSAO calculation. The first step is the preparation of inputs, that is, the view space normals and depth. The normals are stored using the first step vertex shader (Chapter6/SSAO/shaders/SSAO_FirstStep.vert).

```
vEyeSpaceNormal_Depth = N*vNormal;
vec4 esPos = MV*vec4(vVertex,1);
gl_Position = P*esPos;
```

The fragment shader (Chapter6/SSAO/shaders/SSAO_FirstStep.frag) then outputs these values. The depth is extracted from the depth attachment of the FBO.

The second step is the actual SSAO calculation. We use a fragment shader (Chapter6/SSAO/shaders/SSAO_SecondStep.frag) to perform this by first rendering a screen-aligned quad. Then, for each fragment, the corresponding normal and depth values are obtained from the render target, from the first step. Next, a loop is run to compare the depth values of the neighboring fragments and then an occlusion value is estimated.

```
float depth = texture(depthTex, vUV).r;
if(depth<1.0)
{
    vec3 n = normalize(texture(normalTex, vUV).xyz*2.0 - 1.0);
    vec4 p = invP*vec4(vUV,depth,1);
    p.xyz /= p.w;

    vec2 random = normalize(texture(noiseTex,
        viewportSize/random_size * vUV).rg * 2.0 - 1.0);
    float ao = 0.0;

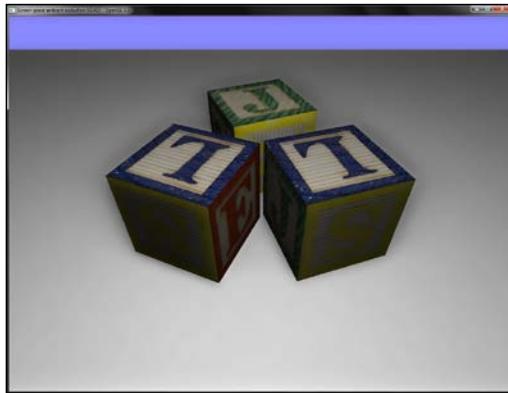
    for(int i = 0; i < NUM_SAMPLES; i++)
    {
        float npw = (pw + radius * samples[i].x * random.x);
        float nph = (ph + radius * samples[i].y * random.y);

        vec2 uv = vUV + vec2(npw, nph);
        vec4 p0 = invP * vec4(vUV,texture2D(depthTex, uv).r, 1.0);
        p0.xyz /= p0.w;
        ao += calcAO(p0, p, n);
        //calculate similar depth points from the neighborhood
        //and calculate ambient occlusion amount
    }
    ao *= INV_NUM_SAMPLES/8.0;
    vFragColor = vec4(vec3(0), ao);
}
```

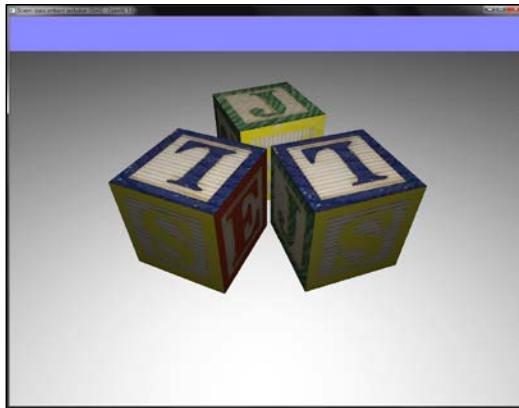
After the second shader, we filter the SSAO output using separable Gaussian convolution. The default draw buffer is then restored and then the Gaussian filtered SSAO output is alpha blended with the normal rendering.

There's more...

The demo application implementing this recipe shows the scene with three blocks on a planar quad. When run, the output is as shown in the following screenshot:



Pressing the Space bar disables SSAO to produce the following output. As can be seen, ambient occlusion helps in giving shaded cues that approximate how near or far objects are. We can also change the sampling radius by using the + and - keys.



See also

- ▶ A Simple and Practical Approach to SSAO by Jose Maria Mendez: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753
- ▶ SSAO Article at GameRendering.com: <http://www.gamerendering.com/category/lighting/ssao-lighting/>

Implementing global illumination using spherical harmonics lighting

In this recipe, we will learn about implementing simple global illumination using spherical harmonics. Spherical harmonics is a class of methods that enable approximation of functions as a product of a set of coefficients with a set of basis functions. Rather than calculating the lighting contribution by evaluating the **bi-directional reflectance distribution function (BRDF)**, this method uses special HDR/RGBE images that store the lighting information. The only attribute required for this method is the per-vertex normal. These are multiplied with the spherical harmonics coefficients that are extracted from the HDR/RGBE images.

The RGBE image format was invented by *Greg Ward*. These images store three bytes for the RGB value (that is, the red, green, and blue channel) and an additional byte which stores a shared exponent. This enables these files to have an extended range and precision of floating point values. For details about the theory behind the spherical harmonics method and the RGBE format, refer to the references in the *See also* section of this recipe.

To give an overview of the recipe, using the probe image, the SH coefficients (C1 to C5) are estimated by projection. Details of the projection method are given in the references in the *See also* section. For most of the common lighting HDR probes, the spherical harmonic coefficients are documented. We use these values as constants in our vertex shader.

Getting ready

The code for this recipe is contained in the `Chapter6/SphericalHarmonics` directory. For this recipe, we will be using the Obj mesh loader discussed in the previous chapter.

How to do it...

Let us start this recipe by following these simple steps:

1. Load an obj mesh using the `ObjLoader` class and fill the OpenGL buffer objects and the OpenGL textures, using the material information loaded from the file, as in the previous recipes.
2. In the vertex shader that is used for the mesh, perform the lighting calculation using spherical harmonics. The vertex shader is detailed as follows:

```
#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;
layout(location = 2) in vec2 vUV;

smooth out vec2 vUVout;
smooth out vec4 diffuse;
```

```
uniform mat4 P;
uniform mat4 MV;
uniform mat3 N;

const float C1 = 0.429043;
const float C2 = 0.511664;
const float C3 = 0.743125;
const float C4 = 0.886227;
const float C5 = 0.247708;
const float PI = 3.1415926535897932384626433832795;

//Old town square probe
const vec3 L00 = vec3( 0.871297, 0.875222, 0.864470);
const vec3 L1m1 = vec3( 0.175058, 0.245335, 0.312891);
const vec3 L10 = vec3( 0.034675, 0.036107, 0.037362);
const vec3 L11 = vec3(-0.004629, -0.029448, -0.048028);
const vec3 L2m2 = vec3(-0.120535, -0.121160, -0.117507);
const vec3 L2m1 = vec3( 0.003242, 0.003624, 0.007511);
const vec3 L20 = vec3(-0.028667, -0.024926, -0.020998);
const vec3 L21 = vec3(-0.077539, -0.086325, -0.091591);
const vec3 L22 = vec3(-0.161784, -0.191783, -0.219152);
const vec3 scaleFactor = vec3(0.161784/
(0.871297+0.161784), 0.191783/(0.875222+0.191783),
0.219152/(0.864470+0.219152));

void main()
{
    vUVout=vUV;
    vec3 tmpN = normalize(N*vNormal);
    vec3 diff = C1 * L22 * (tmpN.x*tmpN.x -
    tmpN.y*tmpN.y) +
        C3 * L20 * tmpN.z*tmpN.z +
        C4 * L00 -
        C5 * L20 +
        2.0 * C1 * L2m2*tmpN.x*tmpN.y +
        2.0 * C1 * L21*tmpN.x*tmpN.z +
        2.0 * C1 * L2m1*tmpN.y*tmpN.z +
        2.0 * C2 * L11*tmpN.x +
        2.0 * C2 * L1m1*tmpN.y +
        2.0 * C2 * L10*tmpN.z;
    diff *= scaleFactor;
    diffuse = vec4(diff, 1);
    gl_Position = P*(MV*vec4(vVertex,1));
}
```

3. The per-vertex color calculated by the vertex shader is interpolated by the rasterizer and then the fragment shader sets the color as the current fragment color.

```
#version 330 core
uniform sampler2D textureMap;
uniform float useDefault;
smooth in vec4 diffuse;
smooth in vec2 vUVout;
layout(location=0) out vec4 vFragColor;
void main() {
    vFragColor = mix(texture(textureMap, vUVout)*diffuse,
                    diffuse, useDefault);
}
```

How it works...

Spherical harmonics is a technique that approximates the lighting, using coefficients and spherical harmonics basis. The coefficients are obtained at initialization from an HDR/RGEBE image file that contains information about lighting. This allows us to approximate the same light so the graphical scene feels more immersive.

The method reproduces accurate diffuse reflection using information extracted from an HDR/RGEBE light probe. The light probe itself is not accessed in the code. The spherical harmonics basis and coefficients are extracted from the original light probe using projection. Since this is a mathematically involved process, we refer the interested readers to the references in the See also section. The code for generating the spherical harmonics coefficients is available online. We used this code to generate the spherical harmonics coefficients for the shader.

The spherical harmonics is a frequency space representation of an image on a sphere. As was shown by Ramamoorthi and Hanrahan, only the first nine spherical harmonic coefficients are enough to give a reasonable approximation of the diffuse reflection component of a surface. These coefficients are obtained by constant, linear, and quadratic polynomial interpolation of the surface normal. The interpolation result gives us the diffuse component which has to be normalized by a scale factor which is obtained by summing all of the coefficients as shown in the following code snippet:

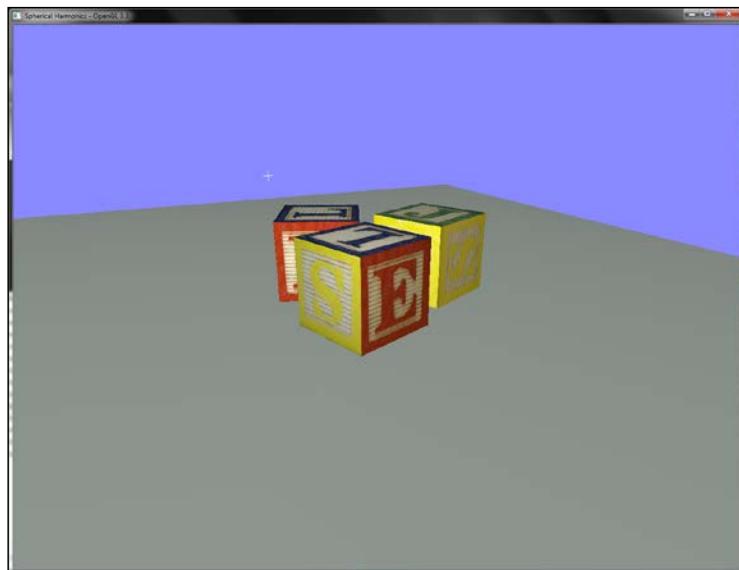
```
vec3 tmpN = normalize(N*vNormal);
vec3 diff = C1 * L22 * (tmpN.x*tmpN.x - tmpN.y*tmpN.y) +
            C3 * L20 * tmpN.z*tmpN.z +
            C4 * L00 -
            C5 * L20 +
            2.0 * C1 * L2m2*tmpN.x*tmpN.y +
            2.0 * C1 * L21*tmpN.x*tmpN.z +
            2.0 * C1 * L2m1*tmpN.y*tmpN.z +
            2.0 * C2 * L11*tmpN.x +
            2.0 * C2 * L1m1*tmpN.y +
            2.0 * C2 * L10*tmpN.z;
diff *= scaleFactor;
```

The obtained per-vertex diffuse component is then forwarded through the rasterizer to the fragment shader where it is directly multiplied by the texture of the surface.

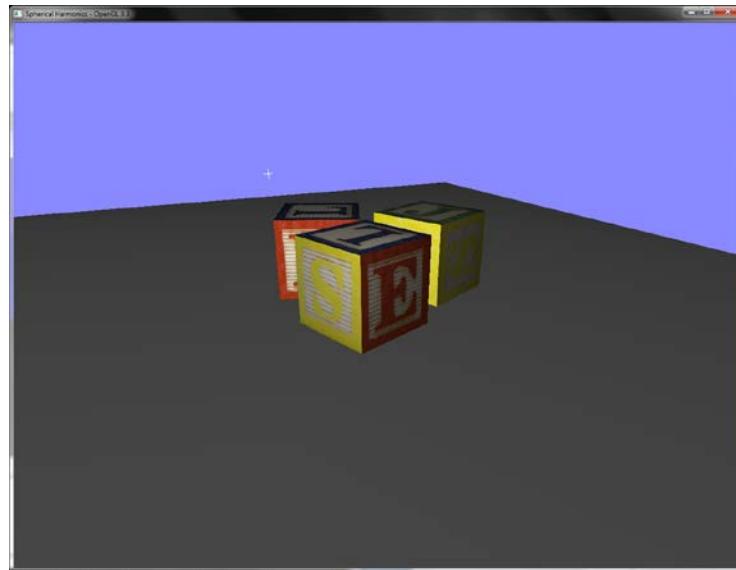
```
vFragColor = mix(texture(textureMap, vUVout)*diffuse,  
diffuse, useDefault);
```

There's more...

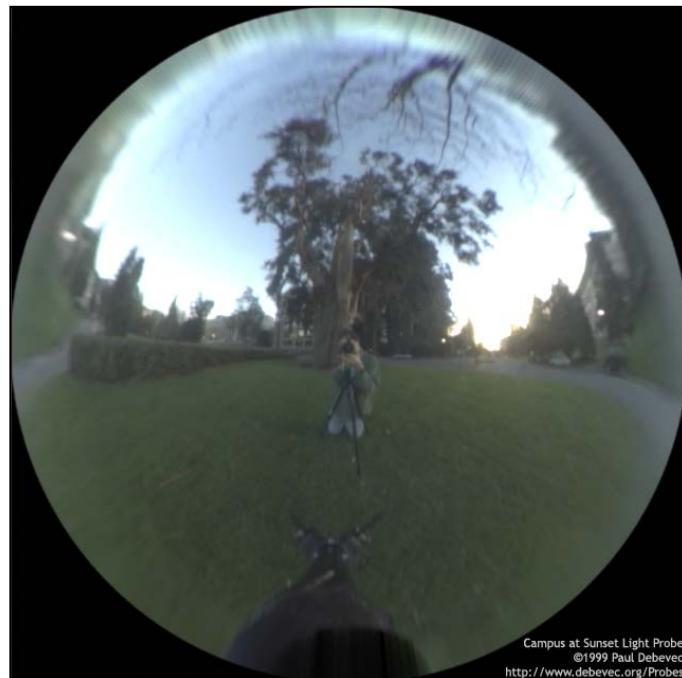
The demo application implementing this recipe renders the same scene as in the previous recipes, as shown in the following figure. We can rotate the camera view using the left mouse button, whereas, the point light source can be rotated using the right mouse button. Pressing the Space bar toggles the use of spherical harmonics. When spherical harmonics lighting is on, we get the following result:



Without the spherical harmonics lighting, the result is as follows:



The probe image used for this image is shown in the following figure:



Note that this method approximates global illumination by modifying the diffuse component using the spherical harmonics coefficients. We can also add the conventional Blinn Phong lighting model as we did in the earlier recipes. For that we would only need to evaluate the Blinn Phong lighting model using the normal and light position, as we did in the previous recipe.

See also

- ▶ *Ravi Ramamoorthi and Pat Hanrahan, An Efficient Representation for Irradiance Environment Maps:* <http://www1.cs.columbia.edu/~ravir/papers/envmap/index.html>
- ▶ *Randi J. Rost, Mill M. Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, Mike Weiblen, OpenGL Shading Language, Third Edition, Section 12.3, Lighting and Spherical Harmonics,* Addison-Wesley Professional
- ▶ *Kelly Dempski and Emmanuel Viale, Advanced Lighting and Materials with Shaders, Chapter 8, Spherical Harmonic Lighting,* Jones & Bartlett Publishers
- ▶ The RGBE image format specifications: <http://www.graphics.cornell.edu/online/formats/rbge/>
- ▶ Paul Debevec HDR light probes: <http://www.pauldebevec.com/Probes/>
- ▶ Spherical harmonics lighting tutorial: <http://www.paulsprojects.net/opengl/sh/sh.html>

Implementing GPU-based ray tracing

To this point, all of the recipes rendered 3D geometry using rasterization. In this recipe, we will implement another method for rendering geometry, which is called **ray tracing**. Simply put, ray tracing uses a probing ray from the camera position into the graphical scene. The intersections of this ray are obtained for each geometry. The good thing with this method is that only the visible objects are rendered.

The ray tracing algorithm can be given in pseudocode as follows:

```
For each pixel on screen
    Get the eye ray origin and direction using camera position
    For the amount of traces required
        Cast the ray into scene
        For each object in the scene
            Check eye ray for intersection
            If intersection found
                Determine the hit point and surface normal
                For each light source
                    Calculate diffuse and specular comp. at hit point
                    Cast shadow ray from hit point to light
            End For
```

```

    Darken diffuse component based on shadow result
    Set the hit point as the new ray origin
    Reflect the eye ray direction at surface normal
End If
End For
End For
End For

```

Getting ready

The code for this recipe is contained in the Chapter6/GPURaytracing directory.

How to do it...

Let us start with this recipe by following these simple steps:

1. Load the Obj mesh model using the Obj loader and store mesh geometry in vectors. Note that for the GPU ray tracer we use the original vertices and indices lists stored in the OBJ file.

```

vector<unsigned short> indices2;
vector<glm::vec3> vertices2;
if(!obj.Load(mesh_filename.c_str(), meshes, vertices,
indices, materials, aabb, vertices2, indices2)) {
    cout<<"Cannot load the 3ds mesh"<<endl;
    exit(EXIT_FAILURE);
}

```

2. Load the material texture maps into an OpenGL texture array instead of loading each texture separately, as in previous recipes. We opted for texture arrays because this helps in simplifying the shader code and we would have no way in determining the total samplers we would require, as that is dependent on the material textures we have in the model. In previous recipes, there was a single texture sampler which was modified for each sub-mesh.

```

for(size_t k=0;k<materials.size();k++) {
if(materials[k]->map_Kd != "") {
if(k==0) {
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D_ARRAY, textureID);
    glTexParameteri(GL_TEXTURE_2D_ARRAY,
    GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    //set other texture parameters
}
//set image name
}

```

```

GLubyte* pData = SOIL_load_image(full_filename.c_str(),
&texture_width, &texture_height, &channels,
SOIL_LOAD_AUTO);
if(pData == NULL) {
    cerr<<"Cannot load image: "<<full_filename.c_str()<<endl;
    exit(EXIT_FAILURE);
}
//flip the image and set the image format
if(k==0) {
    glTexImage3D(GL_TEXTURE_2D_ARRAY, 0, format,
    texture_width, texture_height, total, 0, format,
    GL_UNSIGNED_BYTE, NULL);
}
glTexSubImage3D(GL_TEXTURE_2D_ARRAY, 0, 0, 0, k,
    texture_width, texture_height, 1, format,
    GL_UNSIGNED_BYTE, pData);
SOIL_free_image_data(pData);
}
}

```

3. Store the vertex positions into a texture for the ray tracing shader. We use a floating point texture with the GL_RGBA32F internal format.

```

glGenTextures(1, &texVerticesID);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, texVerticesID);
//set the texture formats
GLfloat* pData = new GLfloat[vertices2.size()*4];
int count = 0;
for(size_t i=0;i<vertices2.size();i++) {
pData[count++] = vertices2[i].x;
pData[count++] = vertices2[i].y;
pData[count++] = vertices2[i].z;
pData[count++] = 0;
}
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
vertices2.size(), 1, 0, GL_RGBA, GL_FLOAT, pData);
delete [] pData;

```

4. Store the list of indices into an integral texture for the ray tracing shader. Note that for this texture, the internal format is GL_RGBA16I and the format is GL_RGBA_INTEGER.

```

glGenTextures(1, &texTrianglesID);
glActiveTexture(GL_TEXTURE2);
 glBindTexture(GL_TEXTURE_2D, texTrianglesID);
//set the texture formats

```

```

GLushort* pData2 = new GLushort[indices2.size()];
count = 0;
for(size_t i=0;i<indices2.size();i+=4) {
    pData2[count++] = (indices2[i]);
    pData2[count++] = (indices2[i+1]);
    pData2[count++] = (indices2[i+2]);
    pData2[count++] = (indices2[i+3]);
}
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16I,
indices2.size()/4, 1, 0, GL_RGBA_INTEGER,
GL_UNSIGNED_SHORT, pData2);
delete [] pData2;

```

5. In the render function, bind the ray tracing shader and then draw a full-screen quad to invoke the fragment shader for the entire screen.

How it works...

The main code for ray tracing is the ray tracing fragment shader (Chapter6/GPURaytracing/shaders/raytracer.frag). We first set up the camera ray origin and direction using the parameters passed to the shader as shader uniforms.

```

eyeRay.origin = eyePos;
cam.U = (invMVP*vec4(1,0,0,0)).xyz;
cam.V = (invMVP*vec4(0,1,0,0)).xyz;
cam.W = (invMVP*vec4(0,0,1,0)).xyz;
cam.d = 1;
eyeRay.dir = get_direction(uv, cam);
eyeRay.dir += cam.U*uv.x;
eyeRay.dir += cam.V*uv.y;

```

After the eye ray is set up, we check the ray against the axially aligned bounding box of the scene. If there is an intersection, we continue further. For this simple example, we use a brute force method of looping through all of the triangles and testing each of them in turn for ray intersection.

In ray tracing, we try to find the nearest intersection of a parametric ray with the given triangle. Any point along the ray is obtained by using a parameter t . We are looking for the nearest intersection (smallest t value). If there is an intersection and it is the closest so far, we store the collision information and the normal at the intersection point. The t parameter gives us the exact position where the intersection occurs.

```

vec4 val=vec4(t,0,0,0);
vec3 N;
for(int i=0;i<int(TRIANGLE_TEXTURE_SIZE);i++)
{

```

```
vec3 normal;
vec4 res = intersectTriangle(eyeRay.origin, eyeRay.dir, i,
normal);
if(res.x>0 && res.x <= val.x) {
    val = res;
    N = normal;
}
}
```

When we plug its value into the parametric equation of a ray, we get the hit point. Then, we calculate a vector to light from the hit point. This vector is then used to estimate the diffuse component and the attenuation amount.

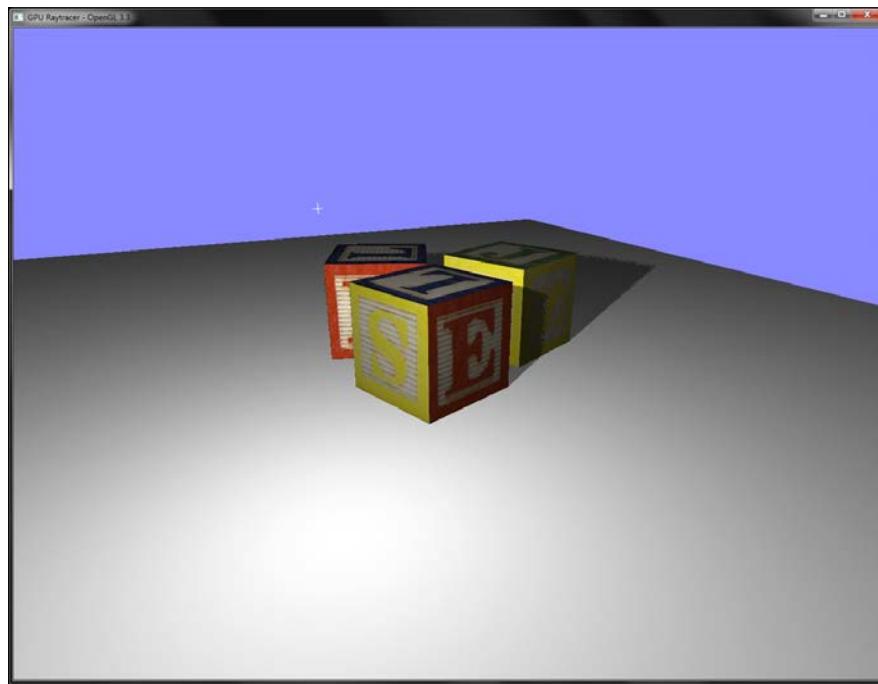
```
if(val.x != t) {
    vec3 hit = eyeRay.origin + eyeRay.dir*val.x;
    vec3 jitteredLight = light_position +
uniformlyRandomVector(gl_FragCoord.x);
    vec3 L = (jitteredLight.xyz-hit);
    float d = length(L);
    L = normalize(L);
    float diffuse = max(0, dot(N, L));
    float attenuationAmount = 1.0/(k0 + (k1*d) + (k2*d*d));
    diffuse *= attenuationAmount;
```

With ray tracing, shadows are very easy to calculate. We simply cast another ray, but this time, just look at if the ray intersects any object on its way to the light source. If it does, we darken the final color, otherwise we leave the color as is. Note that to prevent the shadow acne, we add a slight offset to the ray start position.

```
float inShadow = shadow(hit+ N*0.0001, L);
vFragColor = inShadow*diffuse*mix(texture(textureMaps,
val.yzw), vec4(1), (val.w==255) );
return;
}
```

There's more...

The demo application for this recipe renders the same scene as in previous recipes. The scene can be toggled between rasterization and GPU ray tracing by pressing the Space bar. We can see that the shadows are clearly visible in the ray tracing scene. Note that the performance of GPU ray tracing is directly related to how close or far the object is from the camera, as well as how many triangles are there in the rendered mesh. For better performance, some acceleration structure, such as uniform grid or kd-tree should be employed. Also note, soft shadows require us to cast more shadow rays, which also add additional strain on the ray tracing fragment shader.



See also

- ▶ *Timothy Purcell, Ian Buck, William R. Mark, and Pat Hanrahan*, ACM Transactions on Graphics 21 (3), Ray Tracing on Programmable Graphics Hardware, pages 703-712: <http://graphics.stanford.edu/papers/rtongfx/>
- ▶ *Real-time GPU Ray-Tracer at Icare3D*: http://www.icare3d.org/codes-and-projects/codes/raytracer_gpu_full_1-0.html

Implementing GPU-based path tracing

In this recipe, we will implement another method, called path tracing, for rendering geometry. Similar to ray tracing, path tracing casts rays but these rays are shot randomly from the light position(s). Since it is usually difficult to approximate real lighting, we can approximate it using Monte Carlo-based integration schemes. These methods use random sampling and if there are enough samples, the integration result converges to the true solution.

We can give the path tracing pseudocode as follows:

```
For each pixel on screen
    Create a light ray from light position in a random direction
    For the amount of traces required
        For each object in the scene
            Check light ray for intersection
            If intersection found
                Determine the hit point and surface normal
                Calculate diffuse and specular comp. at hit point
                Cast shadow ray in random direction from hit point
                Darken diffuse component based on shadow result
                Set the randomly jittered hit point as new ray origin
                Reflect the light ray direction at surface normal
            End If
        End For
    End For
End For
```

Getting ready

The code for this recipe is contained in the `Chapter6/GPUPathtracing` directory.

How to do it...

Let us start with this recipe by following these simple steps:

1. Load the Obj mesh model using the Obj loader and store the mesh geometry in vectors. Note that for the GPU path tracer we use the original vertices and indices lists stored in the OBJ file, as in the previous recipe.
2. Load the material texture maps into an OpenGL texture array instead of loading each texture separately as in the previous recipe.

-
3. Store the vertex positions into a texture for the path tracing shader, similar to how we stored them for ray tracing in the previous recipe. We use a floating point texture with the GL_RGBA32F internal format.

```
glGenTextures(1, &texVerticesID);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, texVerticesID);
//set the texture formats
GLfloat* pData = new GLfloat[vertices2.size()*4];
int count = 0;
for(size_t i=0;i<vertices2.size();i++) {
    pData[count++] = vertices2[i].x;
    pData[count++] = vertices2[i].y;
    pData[count++] = vertices2[i].z;
    pData[count++] = 0;
}
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
vertices2.size(),1, 0, GL_RGBA, GL_FLOAT, pData);
delete [] pData;
```

4. Store the indices list into an integral texture for the path tracing shader, as was done for the ray tracing recipe. Note that for this texture, the internal format is GL_RGBA16I and format is GL_RGBA_INTEGER.

```
glGenTextures(1, &texTrianglesID);
glActiveTexture(GL_TEXTURE2);
 glBindTexture(GL_TEXTURE_2D, texTrianglesID);
//set the texture formats
GLushort* pData2 = new GLushort[indices2.size()];
count = 0;
for(size_t i=0;i<indices2.size();i+=4) {
    pData2[count++] = (indices2[i]);
    pData2[count++] = (indices2[i+1]);
    pData2[count++] = (indices2[i+2]);
    pData2[count++] = (indices2[i+3]);
}
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16I,
indices2.size()/4, 1, 0, GL_RGBA_INTEGER,
GL_UNSIGNED_SHORT, pData2);
delete [] pData2;
```

5. In the render function, bind the path tracing shader and then draw a full-screen quad to invoke the fragment shader for the entire screen.

```
pathtraceShader.Use();
 glUniform3fv(pathtraceShader("eyePos"), 1,
glm::value_ptr(eyePos));
```

```
glUniform1f(pathtraceShader("time"), current);
glUniform3fv(pathtraceShader("light_position"), 1,
&(lightPosOS.x));
glUniformMatrix4fv(pathtraceShader("invMVP"), 1,
GL_FALSE, glm::value_ptr(invMVP));
DrawFullScreenQuad();
pathtraceShader.UnUse();
```

How it works...

The main code for path tracing is carried out in the path tracing fragment shader (Chapter6/ GPUPathtracing/shaders/pathtracer.frag). We first set up the camera ray origin and direction using the parameters passed to the shader as shader uniforms.

```
eyeRay.origin = eyePos;
cam.U = (invMVP*vec4(1,0,0,0)).xyz;
cam.V = (invMVP*vec4(0,1,0,0)).xyz;
cam.W = (invMVP*vec4(0,0,1,0)).xyz;
cam.d = 1;
eyeRay.dir = get_direction(uv, cam);
eyeRay.dir += cam.U*uv.x;
eyeRay.dir += cam.V*uv.y;
```

After the eye ray is set up, we check the ray against the scene's axially aligned bounding box. If there is an intersection, we call our `path_trace` function.

```
vec2 tNearFar = intersectCube(eyeRay.origin, eyeRay.dir, aabb);
if(tNearFar.x<tNearFar.y) {
    t = tNearFar.y+1;
    vec3 light = light_position + uniformlyRandomVector(time) *
    0.1;
    vFragColor = vec4(pathtrace(eyeRay.origin, eyeRay.dir, light,
    t),1);
}
```

In the `path_trace` function, we run a loop that iterates for a number of passes. In each pass, we check the scene geometry for an intersection with the ray. We use a brute force method of looping through all of the triangles and testing each of them in turn for collision. If we have an intersection, we check to see if this is the nearest intersection. If it is, we store the normal and the texture coordinates at the intersection point.

```
for(int bounce = 0; bounce < MAX_BOUNCES; bounce++) {
    vec2 tNearFar = intersectCube(origin, ray, aabb);
    if( tNearFar.x > tNearFar.y)
        continue;
    if(tNearFar.y<t)
```

```

    t = tNearFar.y+1;
    vec3 N;
    vec4 val=vec4(t,0,0,0);
    for(int i=0;i<int(TRIANGLE_TEXTURE_SIZE);i++)
    {
        vec3 normal;
        vec4 res = intersectTriangle(origin, ray, i, normal);
        if(res.x>0.001 && res.x < val.x) {
            val = res;
            N = normal;
        }
    }
}

```

We then check the t parameter value to find the nearest intersection and then use the texture array to sample the appropriate texture for the output color value for the current fragment. We then change the current ray origin to the current hit point and then change the current ray direction to a uniform random direction in the hemisphere above the intersection point.

```

if(val.x < t) {
    surfaceColor = mix(texture(textureMaps, val.yzw), vec4(1),
    (val.w==255) ).xyz;
    vec3 hit = origin + ray * val.x;
    origin = hit;
    ray = uniformlyRandomDirection(time + float(bounce));
}

```

The diffuse component is then estimated and then the color is accumulated. At the end of the loop, the final accumulated color is returned.

```

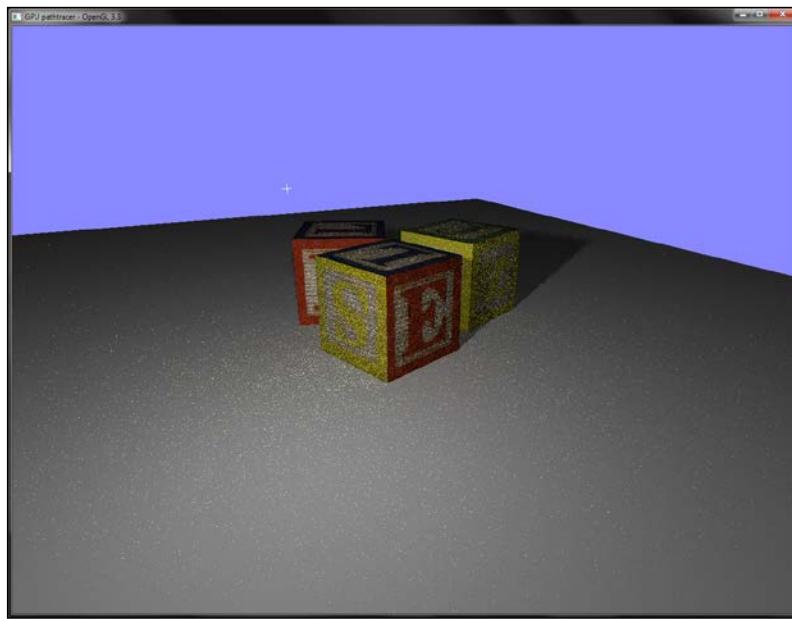
vec3 jitteredLight = light + ray;
vec3 L = normalize(jitteredLight - hit);
diffuse = max(0.0, dot(L, N));
colorMask *= surfaceColor;
float inShadow = shadow(hit+ N*0.0001, L);
accumulatedColor += colorMask * diffuse * inShadow;
t = val.x;
}
}
if(accumulatedColor == vec3(0))
    return surfaceColor*diffuse;
else
    return accumulatedColor/float(MAX_BOUNCES-1);
}

```

Note that the path tracing output is noisy and a large number of samples are needed to get a less noisy result.

There's more...

The demo application for this recipe renders the same scene as in previous recipes. The scene can be toggled between rasterization and GPU path tracing by pressing the Space bar, as shown below:



Note that the performance of GPU path tracing is directly related to how close or far the object is from camera, as well as how many triangles are there in the rendered mesh. In order to reduce the amount of testing, some acceleration structure, such as uniform grid or kd-tree should be employed. In addition, since the results obtained from path tracing are generally noisier as compared to the ray tracing results, noise removal filters, such as Gaussian smoothing could be carried out on the path traced result.

Ray tracing is poor at approximating global illumination and soft shadows. Path tracing, on the other hand, handles global illumination and soft shadows well, but it suffers from noise. To get a good result, it requires a large number of random sampling points. There are other techniques, such as Metropolis light transport, which uses heuristics to only accept good sample points and reject bad sampling points. As a result, it converges to a less noisy result faster as compared to naïve path tracing.

See also

- ▶ *Tim Purcell, Ian Buck, William Mark, and Pat Hanrahan*, Ray Tracing on Programmable Graphics Hardware, ACM Transactions on Graphics 21(3), pp: 703-712, 2002.
Available online: <http://graphics.stanford.edu/papers/rtongfx/>
- ▶ Peter and Karl's GPU Path Tracer: <http://gpupathtracer.blogspot.sg/>
- ▶ Real-time path traced Brigade demo at Siggraph 2012: <http://raytracey.blogspot.co.nz/2012/08/real-time-path-traced-brigade-demo-at.html>

7

GPU-based Volume Rendering Techniques

In this chapter, we will focus on:

- ▶ Implementing volume rendering using 3D texture slicing
- ▶ Implementing volume rendering using single-pass GPU ray casting
- ▶ Implementing pseudo isosurface rendering in single-pass GPU ray casting
- ▶ Implementing volume rendering using splatting
- ▶ Implementing the transfer function for volume classification
- ▶ Implementing polygonal isosurface extraction using the Marching Tetrahedra algorithm
- ▶ Implementing volumetric lighting using half-angle slicing

Introduction

Volume rendering techniques are used in various domains in biomedical and engineering disciplines. They are often used in biomedical imaging to visualize the CT/MRI datasets. In mechanical engineering, they are used to visualize intermediate results from FEM simulations, flow, and structural analysis. With the advent of GPU, all of the existing models and methods of visualization were ported to GPU to harness their computational power. This chapter will detail several algorithms that are used for volume visualization on the GPU in OpenGL Version 3.3 and above. Specifically, we will look at three widely used methods including 3D texture slicing, single-pass ray casting with alpha compositing as well as isosurface rendering, and splatting.

After looking at the volume rendering methods, we will look at volume classification by implementing transfer functions. Polygonal isosurfaces are also often generated to extract out classified regions, for example, cellular boundaries. We, therefore, implement the Marching Tetrahedra algorithm. Finally, volume lighting is another area that is actively researched in the volume rendering community. As there are very few implementations of volume lighting, and especially half-angle slicing, we detail how to implement volume lighting through the half-angle slicing technique in modern OpenGL.

Implementing volume rendering using 3D texture slicing

Volume rendering is a special class of rendering algorithms that allows us to portray fuzzy phenomena, such as smoke. There are numerous algorithms for volume rendering. To start our quest, we will focus on the simplest method called 3D texture slicing. This method approximates the volume-density function by slicing the dataset in front-to-back or back-to-front order and then blends the proxy slices using hardware-supported blending. Since it relies on the rasterization hardware, this method is very fast on the modern GPU.

The pseudo code for view-aligned 3D texture slicing is as follows:

1. Get the current view direction vector.
2. Calculate the min/max distance of unit cube vertices by doing a dot product of each unit cube vertex with the view direction vector.
3. Calculate all possible intersections parameter (λ) of the plane perpendicular to the view direction with all edges of the unit cube going from the nearest to farthest vertex, using min/max distances from step 1.
4. Use the intersection parameter λ (from step 3) to move in the viewing direction and find the intersection points. Three to six intersection vertices will be generated.
5. Store the intersection points in the specified order to generate triangular primitives, which are the proxy geometries.
6. Update the buffer object memory with the new vertices.

Getting ready

The code for this recipe is in the `Chapter7/3DTextureSlicing` directory.

How to do it...

Let us start our recipe by following these simple steps:

1. Load the volume dataset by reading the external volume datafile and passing the data into an OpenGL texture. Also enable hardware mipmap generation. Typically, the volume datafiles store densities that are obtained from using a cross-sectional imaging modality such as CT or MRI scans. Each CT/MRI scan is a 2D slice. We accumulate these slices in Z direction to obtain a 3D texture, which is simply an array of 2D textures. The densities store different material types, for example, values ranging from 0 to 20 are typically occupied by air. As we have an 8-bit unsigned dataset, we store the dataset into a local array of `GLubyte` type. If we had an unsigned 16-bit dataset, we would have stored it into a local array of `GLushort` type. In case of 3D textures, in addition to the S and T parameters, we have an additional parameter R that controls the slice we are at in the 3D texture.

```
std::ifstream infile(volume_file.c_str(), std::ios_base::binary);
if(infile.good()) {
    GLubyte* pData = new GLubyte[XDIM*YDIM*ZDIM];
    infile.read(reinterpret_cast<char*>(pData),
    XDIM*YDIM*ZDIM*sizeof(GLubyte));
    infile.close();
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_3D, textureID);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S,
    GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T,
    GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R,
    GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_BASE_LEVEL, 0);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAX_LEVEL, 4);
    glTexImage3D(GL_TEXTURE_3D, 0, GL_RED, XDIM, YDIM, ZDIM, 0, GL_RED, GL_
UNSIGNED_BYTE, pData);
    glGenerateMipmap(GL_TEXTURE_3D);
    return true;
} else {
    return false;
}
```

The filtering parameters for 3D textures are similar to the 2D texture parameters that we have seen before. Mipmaps are collections of down-sampled versions of a texture that are used for **level of detail (LOD)** functionality. That is, they help to use a down-sampled version of the texture if the viewer is very far from the object on which the texture is applied. This helps improve the performance of the application. We have to specify the max number of levels (`GL_TEXTURE_MAX_LEVEL`), which is the maximum number of mipmaps generated from the given texture. In addition, the base level (`GL_TEXTURE_BASE_LEVEL`) denotes the first level for the mipmap that is used when the object is closest.

The `glGenerateMipMap` function works by generating derived arrays by repeated filtered reduction operation on the previous level. So let's say that we have three mipmap levels and our 3D texture has a resolution of $256 \times 256 \times 256$ at level 0. For level 1 mipmap, the level 0 data will be reduced to half the size by filtered reduction to $128 \times 128 \times 128$. For level 2 mipmap, the level 1 data will be filtered and reduced to $64 \times 64 \times 64$. Finally, for level 3 mipmap, the level 2 data will be filtered and reduced to $32 \times 32 \times 32$.

2. Setup a vertex array object and a vertex buffer object to store the geometry of the proxy slices. Make sure that the buffer object usage is specified as `GL_DYNAMIC_DRAW`. The initial `glBufferData` call allocates GPU memory for the maximum number of slices. The `vTextureSlices` array is defined globally and it stores the vertices produced by texture slicing operation for triangulation. The `glBufferData` is initialized with 0 as the data will be filled at runtime dynamically.

```
const int MAX_SLICES = 512;
glm::vec3 vTextureSlices[MAX_SLICES*12];

glGenVertexArrays(1, &volumeVAO);
glGenBuffers(1, &volumeVBO);
 glBindVertexArray(volumeVAO);
 glBindBuffer (GL_ARRAY_BUFFER, volumeVBO);
 glBufferData (GL_ARRAY_BUFFER, sizeof(vTextureSlices), 0, GL_DYNAMIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
 glBindVertexArray(0);
```

3. Implement slicing of volume by finding intersections of a unit cube with proxy slices perpendicular to the viewing direction. This is carried out by the `sliceVolume` function. We use a unit cube since our data has equal size in all three axes that is, $256 \times 256 \times 256$. If we have a non-equal sized dataset, we can scale the unit cube appropriately.

```
//determine max and min distances
glm::vec3 vecStart[12];
glm::vec3 vecDir[12];
```

```
float lambda[12];
float lambda_inc[12];
float denom = 0;
float plane_dist = min_dist;
float plane_dist_inc = (max_dist-min_dist)/float(num_slices);

//determine vecStart and vecDir values
glm::vec3 intersection[6];
float dL[12];

for(int i=num_slices-1;i>=0;i--) {
    for(int e = 0; e < 12; e++)
    {
        dL[e] = lambda[e] + i*lambda_inc[e];
    }

    if ((dL[0] >= 0.0) && (dL[0] < 1.0)) {
        intersection[0] = vecStart[0] +
        dL[0]*vecDir[0];
    }
    //like wise for all intersection points
    int indices[]={0,1,2, 0,2,3, 0,3,4, 0,4,5};
    for(int i=0;i<12;i++)
        vTextureSlices[count++]=intersection[indices[i]];
}
//update buffer object
glBindBuffer(GL_ARRAY_BUFFER, volumeVBO);
glBufferSubData(GL_ARRAY_BUFFER, 0,
sizeof(vTextureSlices), &(vTextureSlices[0].x));
```

4. In the render function, set the over blending, bind the volume vertex array object, bind the shader, and then call the `glDrawArrays` function.

```
 glEnable(GL_BLEND);
 glEnable(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
 glBindVertexArray(volumeVAO);
 shader.Use();
 glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_
ptr(MVP));
 glDrawArrays(GL_TRIANGLES, 0, sizeof(vTextureSlices)/
sizeof(vTextureSlices[0]));
 shader.UnUse();
 glDisable(GL_BLEND);
```

How it works...

Volume rendering using 3D texture slicing approximates the volume rendering integral by alpha-blending textured slices. The first step is loading and generating a 3D texture from the volume data. After loading the volume dataset, the slicing of the volume is carried out using proxy slices. These are oriented perpendicular to the viewing direction. Moreover, we have to find the intersection of the proxy polygons with the unit cube boundaries. This is carried out by the `sliceVolume` function. Note that slicing is carried out only when the view is rotated.

We first obtain the view direction vector (`viewDir`), which is the third column in the model-view matrix. The first column of the model-view matrix stores the right vector and the second column stores the up vector. We will now detail how the `sliceVolume` function works internally. We find the minimum and maximum vertex in the current viewing direction by calculating the maximum and minimum distance of the 8 unit vertices in the viewing direction. These distances are obtained using the dot product of each unit cube vertex with the view direction vector:

```
float max_dist = glm::dot(viewDir, vertexList[0]);
float min_dist = max_dist;
int max_index = 0;
int count = 0;
for(int i=1;i<8;i++) {
    float dist = glm::dot(viewDir, vertexList[i]);
    if(dist > max_dist) {
        max_dist = dist;
        max_index = i;
    }
    if(dist<min_dist)
        min_dist = dist;
}
int max_dim = FindAbsMax(viewDir);
min_dist -= EPSILON;
max_dist += EPSILON;
```

There are only three unique paths when going from the nearest vertex to the farthest vertex from the camera. We store all possible paths for each vertex into an edge table, which is defined as follows:

```
int edgeList[8][12]={{0,1,5,6, 4,8,11,9, 3,7,2,10 }, //v0 is front
{0,4,3,11, 1,2,6,7, 5,9,8,10 }, //v1 is front
{1,5,0,8, 2,3,7,4, 6,10,9,11}, //v2 is front
{ 7,11,10,8, 2,6,1,9, 3,0,4,5 }, // v3 is front
{ 8,5,9,1, 11,10,7,6, 4,3,0,2 }, // v4 is front
{ 9,6,10,2, 8,11,4,7, 5,0,1,3 }, // v5 is front
{ 9,8,5,4, 6,1,2,0, 10,7,11,3}, // v6 is front
{ 10,9,6,5, 7,2,3,1, 11,4,8,0 } // v7 is front
```

Next, plane intersection distances are estimated for the 12 edge indices of the unit cube:

```

glm::vec3 vecStart[12];
glm::vec3 vecDir[12];
float lambda[12];
float lambda_inc[12];
float denom = 0;
float plane_dist = min_dist;
float plane_dist_inc = (max_dist-min_dist)/float(num_slices);
for(int i=0;i<12;i++) {
    vecStart[i]=vertexList[edges[edgeList[max_index][i]][0]];
    vecDir[i]=vertexList[edges[edgeList[max_index][i]][1]]-
        vecStart[i];
    denom = glm::dot(vecDir[i], viewDir);
    if (1.0 + denom != 1.0) {
        lambda_inc[i] = plane_dist_inc/denom;
        lambda[i]=(plane_dist-glm::dot(vecStart[i],viewDir))/denom;
    } else {
        lambda[i]      = -1.0;
        lambda_inc[i] = 0.0;
    }
}
    
```

Finally, the interpolated intersections with the unit cube edges are carried out by moving back-to-front in the viewing direction. After proxy slices have been generated, the vertex buffer object is updated with the new data.

```

for(int i=num_slices-1;i>=0;i--) {
    for(int e = 0; e < 12; e++) {
        dL[e] = lambda[e] + i*lambda_inc[e];
    }
    if ((dL[0] >= 0.0) && (dL[0] < 1.0)) {
        intersection[0] = vecStart[0] + dL[0]*vecDir[0];
    } else if ((dL[1] >= 0.0) && (dL[1] < 1.0)) {
        intersection[0] = vecStart[1] + dL[1]*vecDir[1];
    } else if ((dL[3] >= 0.0) && (dL[3] < 1.0)) {
        intersection[0] = vecStart[3] + dL[3]*vecDir[3];
    } else continue;

    if ((dL[2] >= 0.0) && (dL[2] < 1.0)){
        intersection[1] = vecStart[2] + dL[2]*vecDir[2];
    } else if ((dL[0] >= 0.0) && (dL[0] < 1.0)){
        intersection[1] = vecStart[0] + dL[0]*vecDir[0];
    } else if ((dL[1] >= 0.0) && (dL[1] < 1.0)){
        intersection[1] = vecStart[1] + dL[1]*vecDir[1];
    } else {
    }
}
    
```

```

        intersection[1] = vecStart[3] + dL[3]*vecDir[3];
    }
    //similarly for others edges until intersection[5]
    int indices[]={0,1,2, 0,2,3, 0,3,4, 0,4,5};
    for(int i=0;i<12;i++)
        vTextureSlices [count++]=intersection[indices[i]];
}
glBindBuffer(GL_ARRAY_BUFFER, volumeVBO);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vTextureSlices),
&(vTextureSlices[0].x));

```

In the rendering function, the appropriate shader is bound. The vertex shader calculates the clip space position by multiplying the object space vertex position (`vPosition`) with the combined model view projection (`MVP`) matrix. It also calculates the 3D texture coordinates (`vUV`) for the volume data. Since we render a unit cube, the minimum vertex position will be (-0.5,-0.5,-0.5) and the maximum vertex position will be (0.5,0.5,0.5). Since our 3D texture lookup requires coordinates from (0,0,0) to (1,1,1), we add (0.5,0.5,0.5) to the object space vertex position to obtain the correct 3D texture coordinates.

```

smooth out vec3 vUV;
void main() {
    gl_Position = MVP*vec4(vVertex.xyz,1);
    vUV = vVertex + vec3(0.5);
}

```

The fragment shader then uses the 3D texture coordinates to sample the volume data (which is now accessed through a new sampler type `sampler3D` for 3D textures) to display the density. At the time of creation of the 3D texture, we specified the internal format as `GL_RED` (the third parameter of the `glTexImage3D` function). Therefore, we can now access our densities through the red channel of the texture sampler. To get a shader of grey, we set the same value for green, blue, and alpha channels as well.

```

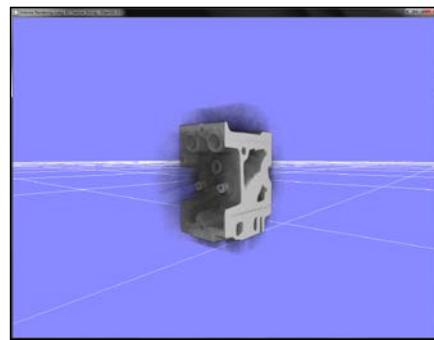
smooth in vec3 vUV;
uniform sampler3D volume;
void main(void) {
    vFragColor = texture(volume, vUV).rrrr;
}

```

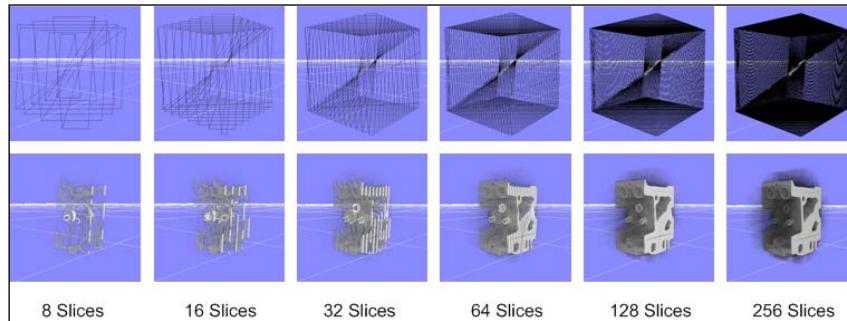
In previous OpenGL versions, we would store the volume densities in a special internal format `GL_INTENSITY`. This is deprecated in the OpenGL3.3 core profile. So now we have to use `GL_RED`, `GL_GREEN`, `GL_BLUE`, or `GL_ALPHA` internal formats.

There's more...

The output from the demo application for this recipe volume renders the engine dataset using 3D texture slicing. In the demo code, we can change the number of slices by pressing the + and - keys.



We now show how we obtain the result by showing an image containing successive 3D texture slicing images in the same viewing direction from **8 slices** all the way to **256 slices**. The results are given in the following screenshot. The wireframe view is shown in the top row, whereas the alpha-blended result is shown in the bottom row.



As can be seen, increasing the number of slices improves the volume rendering result. When the total number of slices goes beyond 256 slices, we do not see a significant difference in the rendering result. However, we begin to see a sharp decrease in performance as we increase the total number of slices beyond 350. This is because more geometry is transferred to the GPU and that reduces performance.

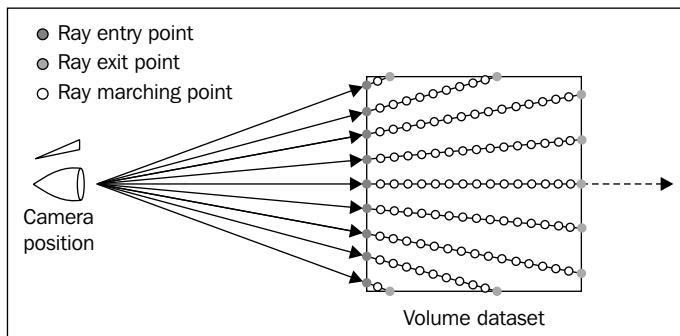
Note that we can see the black halo around the volume dataset. This is due to acquisition artifacts, for example, noise or air that was stored during scanning of the engine dataset. These kinds of artifacts can be removed by either applying a transfer function to remove the unwanted densities or simply removing the unwanted densities in the fragment shader as we will do in the *Implementing volumetric lighting using half-angle slicing* recipe later.

See also

- ▶ The 3.5.2 Viewport-Aligned Slices section in *Chapter 3, GPU-based Volume Rendering, Real-time Volume Graphics, AK Peters/CRC Press*, page numbers 73 to 79

Implementing volume rendering using single-pass GPU ray casting

In this recipe, we will implement volume rendering using single-pass GPU ray casting. There are two basic approaches for doing GPU ray casting: the multi-pass approach and the single-pass approach. Both of these approaches differ in how they estimate the ray marching direction vectors. The single-pass approach uses a single fragment shader. The steps described here can be understood easily from the following diagram:



First, the camera ray direction is calculated by subtracting the vertex positions from the camera position. This gives the ray marching direction. The initial ray position (that is, the ray entry position) is the vertex position. Then based on the ray step size, the initial ray position is advanced in the ray direction using a loop. The volume dataset is then sampled at this position to obtain the density value. This process is continued forward advancing the current ray position until either the ray exits the volume dataset or the alpha value of the color is completely saturated.

The obtained samples during the ray traversal are composited using the current ray function. If the average ray function is used, all of the sample densities are added and then divided by the total number of samples. Similarly, in case of front-to-back alpha compositing, the alpha value of the current sample is multiplied by the accumulated color alpha value and the product is subtracted from the current density. This gives the alpha for the previous densities. This alpha value is then added to the accumulated color alpha. In addition, it is multiplied by the current density and then the obtained color is added to the accumulated color. The accumulated color is then returned as the final fragment color.

Getting ready

The code for this recipe is contained in the Chapter7/GPURaycasting folder.

How to do it...

The steps required to implement single-pass GPU ray casting are as follows:

1. Load the volume data from the file into a 3D OpenGL texture as in the previous recipe. Refer to the `LoadVolume` function in `Chapter7/GPURaycasting/main.cpp` for details.
2. Set up a vertex array object and a vertex buffer object to render a unit cube as follows:

```
glGenVertexArrays(1, &cubeVAOID);
glGenBuffers(1, &cubeVBOID);
glGenBuffers(1, &cubeIndicesID);
glm::vec3 vertices[8]={    glm::vec3(-0.5f,-0.5f,-0.5f), glm::vec3(
0.5f,-0.5f,-0.5f),glm::vec3( 0.5f, 0.5f,-0.5f), glm::vec3(-0.5f,
0.5f,-0.5f),glm::vec3(-0.5f,-0.5f, 0.5f), glm::vec3( 0.5f,-0.5f,
0.5f),glm::vec3( 0.5f, 0.5f, 0.5f), glm::vec3(-0.5f, 0.5f, 0.5f)};
```

```
GLushort cubeIndices[36]={0,5,4,5,0,1,3,7,6,3,6,2,7,4,6,6,4,5,2,1,
3,3,1,0,3,0,7,7,0,4,6,5,2,2,5,1};
```

```
glBindVertexArray(cubeVAOID);
glBindBuffer (GL_ARRAY_BUFFER, cubeVBOID);
glBufferData (GL_ARRAY_BUFFER, sizeof(vertices), &(vertices[0].x),
GL_STATIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

 glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, cubeIndicesID);
 glBufferData (GL_ELEMENT_ARRAY_BUFFER, sizeof(cubeIndices),
&cubeIndices[0], GL_STATIC_DRAW);
 glBindVertexArray(0);
```

3. In the render function, set the ray casting vertex and fragment shaders (`Chapter7/GPURaycasting/shaders/raycaster.vert,frag`) and then render the unit cube.

```
 glEnable(GL_BLEND);
 glBindVertexArray(cubeVAOID);
 shader.Use();
 glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_
ptr(MVP));
```

```

glUniform3fv(shader("camPos"), 1, &(camPos.x));
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
shader.UnUse();
glDisable(GL_BLEND);

```

4. From the vertex shader, in addition to the clip space position, output the 3D texture coordinates for lookup in the fragment shader. We simply offset the object space vertex positions.

```

smooth out vec3 vUV;
void main()
{
    gl_Position = MVP*vec4(vVertex.xyz,1);
    vUV = vVertex + vec3(0.5);
}

```

5. In the fragment shader, use the camera position and the 3D texture coordinates to run a loop in the current viewing direction. Terminate the loop if the current sample position is outside the volume or the alpha value of the accumulated color is saturated.

```

vec3 dataPos = vUV;
vec3 geomDir = normalize((vUV-vec3(0.5)) - camPos);
vec3 dirStep = geomDir * step_size;
bool stop = false;
for (int i = 0; i < MAX_SAMPLES; i++) {
    // advance ray by step
    dataPos = dataPos + dirStep;
    // stop condition
    stop=dot(sign(dataPos-texMin),sign(texMax-dataPos)) < 3.0;
    if (stop)
        break;
}

```

6. Composite the current sample value obtained from the volume using an appropriate operator and finally return the composited color.

```

float sample = texture(volume, dataPos).r;

float prev_alpha = sample - (sample * vFragColor.a);
vFragColor.rgb = prev_alpha * vec3(sample) + vFragColor.rgb;
vFragColor.a += prev_alpha;
//early ray termination
if( vFragColor.a>0.99)
break;
}

```

How it works...

There are two parts of this recipe. The first step is the generation and rendering of the cube geometry for invoking the fragment shader. Note that we could also use a full-screen quad for doing this as we did for the GPU ray tracing recipe but for volumetric datasets it is more efficient to just render the unit cube. The second step is carried out in the shaders.

In the vertex shader (Chapter7/GPURaycasting/shaders/raycast.vert), the 3D texture coordinates are estimated using the per-vertex position of the unit cube. Since the unit cube is at origin, we add `vec(0.5)` to the position to bring the 3D texture coordinates to the 0 to 1 range.

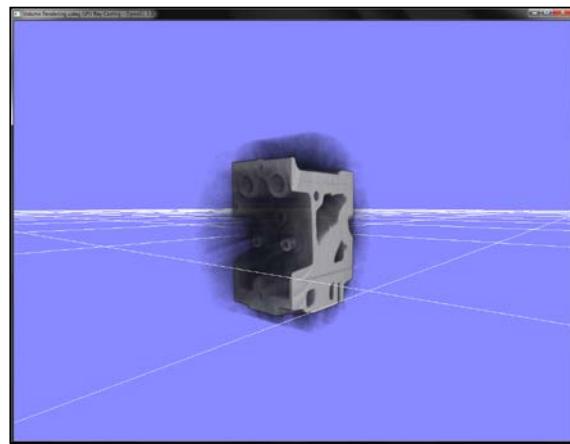
```
#version 330 core
layout(location = 0) in vec3 vVertex;
uniform mat4 MVP;
smooth out vec3 vUV;
void main() {
    gl_Position = MVP*vec4(vVertex.xyz, 1);
    vUV = vVertex + vec3(0.5);
}
```

Next, the fragment shader uses the 3D texture coordinates and the eye position to estimate the ray marching directions. A loop is then run in the fragment shader (as shown in step 5) that marches through the volume dataset and composites the obtained sample values using the current compositing scheme. This process is continued until the ray exits the volume or the alpha value of the accumulated color is fully saturated.

The `texMin` and `texMax` constants have a value of `vec3(-1, -1, -1)` and `vec3(1, 1, 1)` respectively. To determine if the data value is outside the volume data, we use the `sign` function. The `sign` function returns -1 if the value is less than 0, 0 if the value is equal to 0, and 1 if value is greater than 0. Hence, the `sign(dataPos-texMin)` and `sign (texMax-dataPos)` calculation will give us `vec3(1, 1, 1)` at the possible minimum and maximum position. When we do a dot product between two `vec3(1, 1, 1)`, we get the answer 3. So to be within the dataset limits, the dot product will return a value less than 3. If it is greater than 3, we are already out of the volume dataset.

There's more...

The demo application for this demo shows the engine dataset rendered using single-pass GPU ray casting. The camera position can be changed using the left-mouse button and the view can be zoomed in/out by using the middle-mouse button.



See also

- ▶ Chapter 7, *GPU-based Ray Casting, Real-time Volume Graphics*, AK Peters/CRC Press, page numbers 163 to 184
- ▶ Single pass Raycasting at The Little Grasshopper, <http://prideout.net/blog/?p=64>

Implementing pseudo-isosurface rendering in single-pass GPU ray casting

We will now implement pseudo-isosurface rendering in single-pass GPU ray casting. While much of the setup is the same as for the single-pass GPU ray casting, the difference will be in the compositing step in the ray casting fragment shader. In this shader, we will try to find the given isosurface. If it is actually found, we estimate the normal at the sampling point to carry out the lighting calculation for the isosurface.

In the pseudocode, the pseudo-isosurface rendering in single-pass ray casting can be elaborated as follows:

```
Get camera ray direction and ray position  
Get the ray step amount  
For each sample along the ray direction
```

```
Get sample value at current ray position as sample1
Get another sample value at (ray position+step) as sample2
If (sample1-isoValue) < 0 and (sample2-isoValue)>0
    Refine the intersection position using Bisection method
    Get the gradient at the refined position
    Apply Phong illumination at the refined position
    Assign current colour as fragment colour
    Break
End If
End For
```

Getting ready

The code for this recipe is in the Chapter7/GPURaycastingIsosurface folder. We will be starting from the single-pass GPU ray casting recipe using the exact same application side code.

How to do it...

Let us start the recipe by following these simple steps:

1. Load the volume data from file into a 3D OpenGL texture as in the previous recipe.
Refer to the `LoadVolume` function in `Chapter7/GPURaycasting/main.cpp` for details.
2. Set up a vertex array object and a vertex buffer object to render a unit cube as in the previous recipe.
3. In the render function, set the ray casting vertex and fragment shaders (`Chapter7/GPURaycasting/shaders/raycasting.vert,frag`) and then render the unit cube.

```
glEnable(GL_BLEND);
 glBindVertexArray(cubeVAOID);
 shader.Use();
 glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE, glm::value_
 ptr(MVP));
 glUniform3fv(shader("camPos"), 1, &(camPos.x));
 glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
 shader.UnUse();
 glDisable(GL_BLEND);
```

4. From the vertex shader, in addition to the clip-space position, output the 3D texture coordinates for lookup in the fragment shader. We simply offset the object space vertex positions as follows:

```
smooth out vec3 vUV;
void main()
{
    gl_Position = MVP*vec4(vVertex.xyz,1);
    vUV = vVertex + vec3(0.5);
}
```

5. In the fragment shader, use the camera position and the 3D texture coordinates to run a loop in the current viewing direction. The loop is terminated if the current sample position is outside the volume or the alpha value of the accumulated color is saturated.

```
vec3 dataPos = vUV;
vec3 geomDir = normalize((vUV-vec3(0.5)) - camPos);
vec3 dirStep = geomDir * step_size;
bool stop = false;
for (int i = 0; i < MAX_SAMPLES; i++) {
    // advance ray by step
    dataPos = dataPos + dirStep;
    // stop condition
    stop=dot(sign(dataPos-texMin),sign(texMax-dataPos)) < 3.0;
    if (stop)
        break;
```

6. For isosurface estimation, we take two sample values to find the zero crossing of the isofunction inside the volume dataset. If there is a zero crossing, we find the exact intersection point using bisection based refinement. Finally, we use the Phong illumination model to shade the isosurface assuming that the light is located at the camera position.

```
float sample=texture(volume, dataPos).r;
float sample2=texture(volume, dataPos+dirStep).r;
if( (sample -isoValue) < 0  && (sample2-isoValue) >= 0.0)
{
    vec3 xN = dataPos;
    vec3 xF = dataPos+dirStep;
    vec3 tc = Bisection(xN, xF, isoValue);
    vec3 N = GetGradient(tc);
    vec3 V = -geomDir;
    vec3 L = V;
    vFragColor = PhongLighting(L,N,V,250, vec3(0.5));
    break;
}
```

The Bisection function is defined as follows:

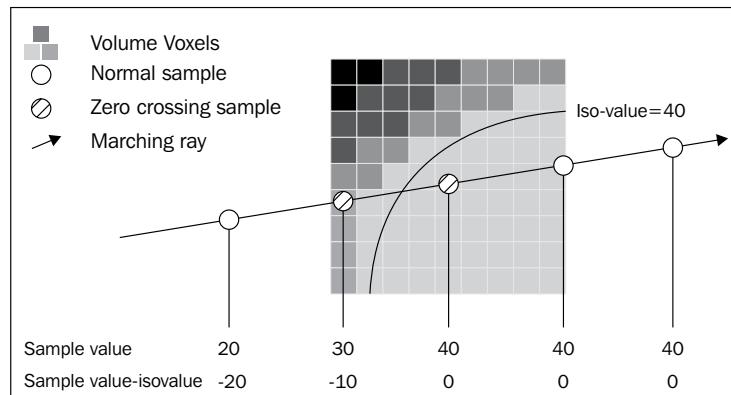
```
vec3 Bisection(vec3 left, vec3 right , float iso) {  
    for(int i=0;i<4;i++) {  
        vec3 midpoint = (right + left) * 0.5;  
        float cM = texture(volume, midpoint).x ;  
        if(cM < iso)  
            left = midpoint;  
        else  
            right = midpoint;  
    }  
    return vec3(right + left) * 0.5;  
}
```

The Bisection function takes the two samples between which the given sample value lies. It then runs a loop. In each step, it calculates the midpoint of the two sample points and checks the density value at the midpoint to the given isovalue. If it is less, the left sample point is swapped with the mid position otherwise, the right sample point is swapped. This helps to reduce the search space quickly. The process is repeated and finally, the midpoint between the left sample point and right sample point is returned. The Gradient function estimates the gradient of the volume density using center finite difference approximation.

```
vec3 GetGradient (vec3 uvw)  
{  
    vec3 s1, s2;  
    //Using center finite difference  
    s1.x = texture(volume, uvw-vec3(DELTA,0.0,0.0)).x ;  
    s2.x = texture(volume, uvw+vec3(DELTA,0.0,0.0)).x ;  
  
    s1.y = texture(volume, uvw-vec3(0.0,DELTA,0.0)).x ;  
    s2.y = texture(volume, uvw+vec3(0.0,DELTA,0.0)).x ;  
  
    s1.z = texture(volume, uvw-vec3(0.0,0.0,DELTA)).x ;  
    s2.z = texture(volume, uvw+vec3(0.0,0.0,DELTA)).x ;  
  
    return normalize((s1-s2)/2.0);  
}
```

How it works...

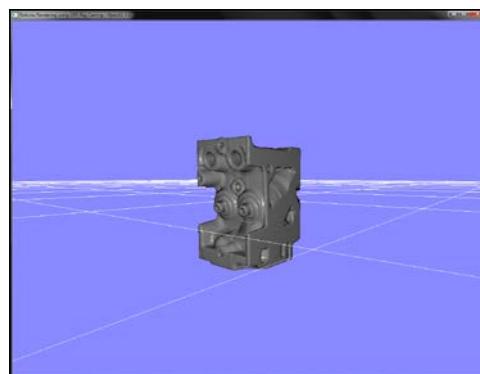
While bulk of the code is similar to the single-pass GPU ray casting recipe. There is a major difference in the ray marching loop. In case of isosurface rendering, we do not use compositing. Instead, we find the zero crossing of the volume dataset isofunction by sampling two consecutive samples. This is well illustrated with the following diagram. If there is a zero crossing, we refine the detected isosurface by using bisection-based refinement.



Next, we use the Phong illumination model to render the shaded isosurface and break out from the ray marching loop. Note that the method shown here renders the nearest isosurface. If we want to render all the surfaces with the given isovalue, we should remove this break statement.

There's more...

The demo application implementing this recipe shows the engine dataset rendered using the pseudo-isosurface rendering mode. When run, the output is as shown in the following screenshot:



See also

- ▶ Advanced Illumination Techniques for GPU-based Volume Rendering, SIGGRAPH 2008 course notes, Available online at http://www.voreen.org/files/sa08-coursesnotes_1.pdf

Implementing volume rendering using splatting

In this recipe, we will implement splatting on the GPU. The splatting algorithm converts the voxel representation into splats by convolving them with a Gaussian kernel. The Gaussian smoothing kernel reduces high frequencies and smoothes out edges giving a smoothed rendered output.

Getting ready

The code for this recipe is in the Chapter7/Splatting directory.

How to do it...

Let us start this recipe by following these simple steps:

1. Load the 3D volume data and store it into an array.

```
std::ifstream infile(filename.c_str(), std::ios_base::binary);
if(infile.good()) {
    pVolume = new GLubyte[XDIM*YDIM*ZDIM];
    infile.read(reinterpret_cast<char*>(pVolume), XDIM*YDIM*ZDIM*sizeof(GLubyte));
    infile.close();
    return true;
} else {
    return false;
}
```

2. Depending on the sampling box size, run three loops to iterate through the entire volume voxel by voxel.

```
vertices.clear();
int dx = XDIM/X_SAMPLING_DIST;
int dy = YDIM/Y_SAMPLING_DIST;
int dz = ZDIM/Z_SAMPLING_DIST;
scale = glm::vec3(dx,dy,dz);
for(int z=0;z<ZDIM;z+=dz) {
```

```

for(int y=0;y<YDIM;y+=dy) {
    for(int x=0;x<XDIM;x+=dx) {
        SampleVoxel(x,y,z);
    }
}
}
}

```

The `SampleVoxel` function is defined in the `VolumeSplatter` class as follows:

```

void VolumeSplatter::SampleVoxel(const int x, const int y,
                                  const int z) {
    GLubyte data = SampleVolume(x, y, z);
    if(data>isoValue) {
        Vertex v;
        v.pos.x = x;
        v.pos.y = y;
        v.pos.z = z;
        v.normal = GetNormal(x, y, z);
        v.pos *= invDim;
        vertices.push_back(v);
    }
}

```

3. In each sampling step, estimate the volume density values at the current voxel. If the value is greater than the given isovalue, store the voxel position and normal into a vertex array.

```

GLubyte data = SampleVolume(x, y, z);
if(data>isoValue) {
    Vertex v;
    v.pos.x = x;
    v.pos.y = y;
    v.pos.z = z;
    v.normal = GetNormal(x, y, z);
    v.pos *= invDim;
    vertices.push_back(v);
}

```

The `SampleVolume` function takes the given sampling point and returns the nearest voxel density. It is defined in the `VolumeSplatter` class as follows:

```

GLubyte VolumeSplatter::SampleVolume(const int x, const int y,
                                      const int z) {
    int index = (x+(y*XDIM)) + z*(XDIM*YDIM);
    if(index<0)
        index = 0;
    if(index >= XDIM*YDIM*ZDIM)
        index = (XDIM*YDIM*ZDIM)-1;
    return pVolume[index];
}

```

4. After the sampling step, pass the generated vertices to a **vertex array object (VAO)** containing a **vertex buffer object (VBO)**.

```
glGenVertexArrays(1, &volumeSplatterVAO);
glGenBuffers(1, &volumeSplatterVBO);
 glBindVertexArray(volumeSplatterVAO);
 glBindBuffer(GL_ARRAY_BUFFER, volumeSplatterVBO);
 glBufferData(GL_ARRAY_BUFFER, splatter->GetTotalVertices()
 *sizeof(Vertex), splatter->GetVertexPointer(), GL_STATIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
 0);
 glEnableVertexAttribArray(1);
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
 (const GLvoid*) offsetof(Vertex, normal));
```

5. Set up two FBOs for offscreen rendering. The first FBO (filterFBOID) is used for Gaussian smoothing.

```
glGenFramebuffers(1, &filterFBOID);
 glBindFramebuffer(GL_FRAMEBUFFER, filterFBOID);
 glGenTextures(2, blurTexID);
 for(int i=0; i<2; i++) {
    glBindTexture(GL_TEXTURE_2D, blurTexID[i]);
    //set texture parameters
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, IMAGE_WIDTH,
 IMAGE_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
    glFramebufferTexture2D(GL_FRAMEBUFFER,
 GL_COLOR_ATTACHMENT0+i, GL_TEXTURE_2D, blurTexID[i], 0);
 }
 GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
 if(status == GL_FRAMEBUFFER_COMPLETE) {
    cout<<"Filtering FBO setup successful."<<endl;
 } else {
    cout<<"Problem in Filtering FBO setup."<<endl;
 }
```

6. The second FBO (fboID) is used to render the scene so that the smoothing operation can be applied on the rendered output from the first pass. Add a render buffer object to this FBO to enable depth testing.

```
glGenFramebuffers(1, &fboID);
 glGenRenderbuffers(1, &rboID);
 glGenTextures(1, &texID);
 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glBindRenderbuffer(GL_RENDERBUFFER, rboID);
```

```

glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texID);
 //set texture parameters
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, IMAGE_WIDTH,
 IMAGE_HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 GL_TEXTURE_2D, texID, 0);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
 GL_RENDERBUFFER, rboID);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT32,
 IMAGE_WIDTH, IMAGE_HEIGHT);
 status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
 if(status == GL_FRAMEBUFFER_COMPLETE) {
    cout<<"Offscreen rendering FBO setup successful."<<endl;
 } else {
    cout<<"Problem in offscreen rendering FBO setup."<<endl;
 }

```

7. In the render function, first render the point splats to a texture using the first FBO (fboID).

```

 glBindFramebuffer(GL_FRAMEBUFFER, fboID);
 glViewport(0,0, IMAGE_WIDTH, IMAGE_HEIGHT);
 glDrawBuffer(GL_COLOR_ATTACHMENT0);
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 glm::mat4 T = glm::translate(glm::mat4(1),
 glm::vec3(-0.5,-0.5,-0.5));
 glBindVertexArray(volumeSplatterVAO);
 shader.Use();
 glUniformMatrix4fv(shader("MV"), 1, GL_FALSE,
 glm::value_ptr(MV*T));
 glUniformMatrix3fv(shader("N"), 1, GL_FALSE,
 glm::value_ptr(glm::inverseTranspose(glm::mat3(MV*T))));
 glUniformMatrix4fv(shader("P"), 1, GL_FALSE,
 glm::value_ptr(P));
 glDrawArrays(GL_POINTS, 0, splatter->GetTotalVertices());
 shader.UnUse();

```

The splatting vertex shader (Chapter7/Splatting/shaders/splatShader.vert) is defined as follows. It calculates the eye space normal. The splat size is calculated using the volume dimension and the sampling voxel size. This is then written to the `gl_PointSize` variable in the vertex shader.

```

#version 330 core
layout(location = 0) in vec3 vVertex;

```

```

layout(location = 1) in vec3 vNormal;
uniform mat4 MV;
uniform mat3 N;
uniform mat4 P;
smooth out vec3 outNormal;
uniform float splatSize;
void main() {
    vec4 eyeSpaceVertex = MV*vec4(vVertex,1);
    gl_PointSize = 2*splatSize/-eyeSpaceVertex.z;
    gl_Position = P * eyeSpaceVertex;
    outNormal = N*vNormal;
}

```

The splatting fragment shader (`Chapter7/Splatting/shaders/splatShader.frag`) is defined as follows:

```

#version 330 core
layout(location = 0) out vec4 vFragColor;
smooth in vec3 outNormal;
const vec3 L = vec3(0,0,1);
const vec3 V = L;
const vec4 diffuse_color = vec4(0.75,0.5,0.5,1);
const vec4 specular_color = vec4(1);
void main() {
    vec3 N;
    N = normalize(outNormal);
    vec2 P = gl_PointCoord*2.0 - vec2(1.0);
    float mag = dot(P.xy,P.xy);
    if (mag > 1)
        discard;

    float diffuse = max(0, dot(N,L));
    vec3 halfVec = normalize(L+V);
    float specular=pow(max(0, dot(halfVec,N)),400);
    vFragColor = (specular*specular_color) +
                 (diffuse*diffuse_color);
}

```

8. Next, set the filtering FBO and first apply the vertical and then the horizontal Gaussian smoothing pass by drawing a full-screen quad as was done in the Variance shadow mapping recipe in *Chapter 4, Lights and Shadows*.

```

glBindVertexArray(quadVAOID);
glBindFramebuffer(GL_FRAMEBUFFER, filterFBOID);

```

```

glDrawBuffer(GL_COLOR_ATTACHMENT0) ;
gaussianV_shader.Use() ;
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0) ;
glDrawBuffer(GL_COLOR_ATTACHMENT1) ;
gaussianH_shader.Use() ;
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0) ;

```

9. Unbind the filtering FBO, restore the default draw buffer and render the filtered output on the screen.

```

glBindFramebuffer(GL_FRAMEBUFFER, 0) ;
glDrawBuffer(GL_BACK_LEFT) ;
glViewport(0, 0, WIDTH, HEIGHT) ;
quadShader.Use() ;
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, 0) ;
quadShader.UnUse() ;
glBindVertexArray(0) ;

```

How it works...

Splatting algorithm works by rendering the voxels of the volume data as Gaussian blobs and projecting them on the screen. To achieve this, we first estimate the candidate voxels from the volume dataset by traversing through the entire volume dataset voxel by voxel for the given isovalue. If we have the appropriate voxel, we store its normal and position into a vertex array. For convenience, we wrap all of this functionality into the `VolumeSplatter` class.

We first create a new instance of the `VolumeSplatter` class. Next, we set the volume dimensions and then load the volume data. Next, we specify the target isovalue and the number of sampling voxels to use. Finally, we call the `VolumeSplatter::SplatVolume` function that traverses the whole volume voxel by voxel.

```

splatter = new VolumeSplatter();
splatter->SetVolumeDimensions(256, 256, 256);
splatter->LoadVolume(volume_file);
splatter->SetIsosurfaceValue(40);
splatter->SetNumSamplingVoxels(64, 64, 64);
std::cout<<"Generating point splats ...";
splatter->SplatVolume();
std::cout<<"Done."<<std::endl;

```

The splatter stores the vertices and normals into a vertex array. We then generate the vertex buffer object from this array. In the rendering function, we first draw the entire splat dataset in a single-pass into an offscreen render target. This is done so that we can filter it using separable Gaussian convolution filters. Finally, the filtered output is displayed on a full-screen quad.

The splatting vertex shader (`Chapter7/Splatting/shaders/splatShader.vert`) calculates the point size on screen based on the depth of the splat. In order to achieve this in the vertex shader, we have to enable the `GL_VERTEX_PROGRAM_POINT_SIZE` state that is, `glEnable(GL_VERTEX_PROGRAM_POINT_SIZE)`. The vertex shader also outputs the splat normals in eye space.

```
vec4 eyeSpaceVertex = MV*vec4(vVertex,1);
gl_PointSize = 2*splatSize/-eyeSpaceVertex.z;
gl_Position = P * eyeSpaceVertex;
outNormal = N*vNormal;
```

Since the default point sprite renders as a screen-aligned quad, in the fragment shader (`Chapter7/Splatting/shaders/splatShader.frag`), we discard all fragments that are outside the radius of the splat at the current splat position.

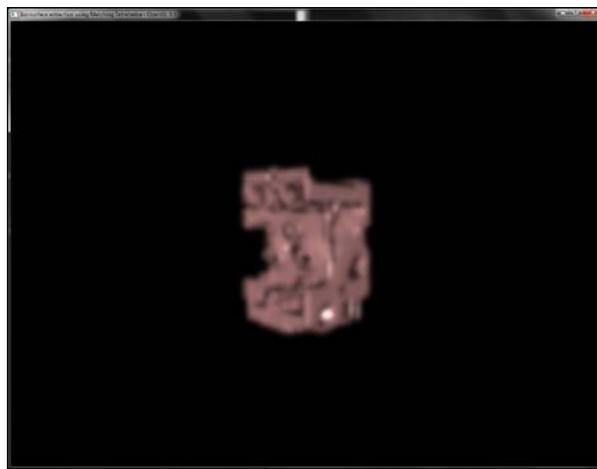
```
vec3 N;
N = normalize(outNormal);
vec2 P = gl_PointCoord*2.0 - vec2(1.0);
float mag = dot(P.xy,P.xy);
if (mag > 1) discard;
```

Finally, we estimate the diffuse and specular components and output the current fragment color using the eye space normal of the splat.

```
float diffuse = max(0, dot(N,L));
vec3 halfVec = normalize(L+V);
float specular = pow(max(0, dot(halfVec,N)),400);
vFragColor = (specular*specular_color) + (diffuse*diffuse_color);
```

There's more...

The demo application implementing this recipe renders the engine dataset as in the previous recipes, as shown in the following screenshot. Note the output appears blurred due to Gaussian smoothing of the splats.



This recipe gave us an overview on the splatting algorithm. Our brute force approach in this recipe was to iterate through all of the voxels. For large datasets, we have to employ an acceleration structure, like an octree, to quickly identify voxels with densities and cull unnecessary voxels.

See also

- ▶ The Qsplat project: <http://graphics.stanford.edu/software/qsplat/>
- ▶ Splatting research at ETH Zurich: (http://graphics.ethz.ch/research/past_projects/surfels/surfacesplatting/)

Implementing transfer function for volume classification

In this recipe, we will implement classification to the 3D texture slicing presented before. We will generate a lookup table to add specific colors to specific densities. This is accomplished by generating a 1D texture that is looked up in the fragment shader with the current volume density. The returned color is then used as the color of the current fragment. Apart from the setup of the transfer function data, all the other content remains the same as in the 3D texture slicing recipe. Note that the classification method is not limited to 3D texture slicing, it can be applied to any volume rendering algorithm.

Getting ready

The code for this recipe is in the `Chapter7/3DTextureSlicingClassification` directory.

How to do it...

Let us start this recipe by following these simple steps:

1. Load the volume data and setup the texture slicing as in the *Implementing volume rendering using 3D texture slicing* recipe.
2. Create a 1D texture that will be our transfer function texture for color lookup. We create a set of color values and then interpolate them on the fly. Refer to `LoadTransferFunction` in `Chapter7/3DTextureSlicingClassification/main.cpp`.

```
float pData[256][4];
int indices[9];
for(int i=0;i<9;i++) {
    int index = i*28;
    pData[index][0] = jet_values[i].x;
    pData[index][1] = jet_values[i].y;
    pData[index][2] = jet_values[i].z;
    pData[index][3] = jet_values[i].w;
    indices[i] = index;
}
for(int j=0;j<9-1;j++)
{
    float dDataR = (pData[indices[j+1]][0] - pData[indices[j]][0]);
    float dDataG = (pData[indices[j+1]][1] - pData[indices[j]][1]);
    float dDataB = (pData[indices[j+1]][2] - pData[indices[j]][2]);
    float dDataA = (pData[indices[j+1]][3] - pData[indices[j]][3]);
    int dIndex = indices[j+1]-indices[j];
    float dDataIncR = dDataR/float(dIndex);
    float dDataIncG = dDataG/float(dIndex);
    float dDataIncB = dDataB/float(dIndex);
    float dDataIncA = dDataA/float(dIndex);
    for(int i=indices[j]+1;i<indices[j+1];i++)
    {
        pData[i][0] = (pData[i-1][0] + dDataIncR);
        pData[i][1] = (pData[i-1][1] + dDataIncG);
        pData[i][2] = (pData[i-1][2] + dDataIncB);
        pData[i][3] = (pData[i-1][3] + dDataIncA);
    }
}
```

3. Generate a 1D OpenGL texture from the interpolated lookup data from step 1. We bind this texture to texture unit 1 (GL_TEXTURE1);

```
glGenTextures(1, &tfTexID);
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_1D, tfTexID);
 glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
 glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexImage1D(GL_TEXTURE_1D, 0, GL_RGBA, 256, 0, GL_RGBA, GL_FLOAT, pData);
```

4. In the fragment shader, add a new sampler for the transfer function lookup table. Since we now have two textures, we bind the volume data to texture unit 0 (GL_TEXTURE0) and the transfer function texture to texture unit 1 (GL_TEXTURE1).

```
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/textureSlicer.vert");
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/textureSlicer.frag");
shader.CreateAndLinkProgram();
shader.Use();
 shader.AddAttribute("vVertex");
 shader.AddUniform("MVP");
 shader.AddUniform("volume");
 shader.AddUniform("lut");
 glUniform1i(shader("volume"), 0);
 glUniform1i(shader("lut"), 1);
 shader.UnUse();
```

5. Finally, in the fragment shader, instead of directly returning the current volume density value, we lookup the density value in the transfer function and return the appropriate color value. Refer to Chapter7/3DTextureSlicingClassification/shaders/textureSlicer.frag for details.

```
uniform sampler3D volume;
uniform sampler1D lut;
void main(void) {
    vFragColor = texture(lut, texture(volume, vUV).r);
}
```

How it works...

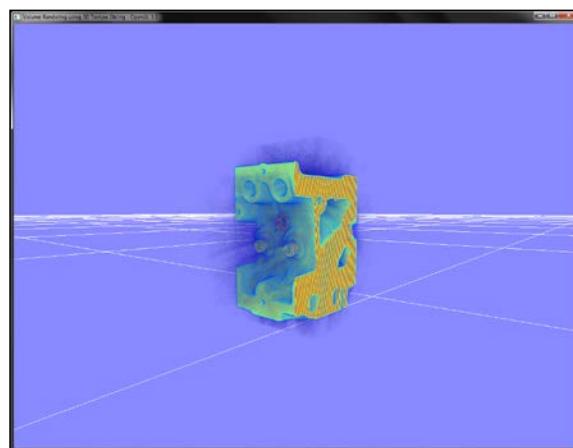
There are two parts of this recipe: the generation of the transfer function texture and the lookup of this texture in the fragment shader. Both of these steps are relatively straightforward to understand. For generation of the transfer function texture, we first create a simple array of possible colors called `jet_values`, which is defined globally as follows:

```
const glm::vec4 jet_values[9]={glm::vec4(0,0,0.5,0),  
    glm::vec4(0,0,1,0.1),  
    glm::vec4(0,0.5,1,0.3),  
    glm::vec4(0,1,1,0.5),  
    glm::vec4(0.5,1,0.5,0.75),  
    glm::vec4(1,1,0,0.8),  
    glm::vec4(1,0.5,0,0.6),  
    glm::vec4(1,0,0,0.5),  
    glm::vec4(0.5,0,0,0.0)};
```

At the time of texture creation, we first reorganize this data into a 256 element array by interpolation. Then, we find the differences among adjacent values and then increment the current value using the difference. This is carried out for all items in the `jet_values` array. Once the data is ready, it is stored in a 1D texture. This is then passed to the fragment shader using another sampler object. In the fragment shader, the density value of the sample that is processed is used as an index into the transfer function texture. Finally, the color obtained from the transfer function texture is stored as the current fragment color.

There's more...

The demo application for this recipe renders the engine dataset as in the 3D texture slicing recipe but now the rendered output is colored using a transfer function. The output from the demo application is displayed in the following screenshot:



See also

- ▶ Chapter 4, Transfer Functions, and Chapter 10, Transfer Functions Reloaded, in *Real-time Volume Graphics*, AK Peters/CRC Press.

Implementing polygonal isosurface extraction using the Marching Tetrahedra algorithm

In the *Implementing pseudo-isosurface rendering in single-pass GPU ray casting* recipe, we implemented pseudo-isosurface rendering in single-pass GPU ray casting. However, these isosurfaces are not composed of triangles; so it is not possible for us to uniquely address individual isosurface regions easily to mark different areas in the volume dataset. This can be achieved by doing an isosurface extraction process for a specific isovalue by traversing the entire volume dataset. This method is known as the **Marching Tetrahedra (MT)** algorithm. This algorithm traverses the whole volume dataset and tries to fit a specific polygon based on the intersection criterion. This process is repeated for the whole volume and finally, we obtain the polygonal mesh from the volume dataset.

Getting ready

The code for this recipe is in the `Chapter7/MarchingTetrahedra` directory. For convenience, we will wrap the Marching Tetrahedra algorithm in a simple class called `TetrahedraMarcher`.

How to do it...

Let us start this recipe by following these simple steps:

1. Load the 3D volume data and store it into an array:

```
std::ifstream infile(filename.c_str(), std::ios_base::binary);
if(infile.good()) {
    pVolume = new GLubyte[XDIM*YDIM*ZDIM];
    infile.read(reinterpret_cast<char*>(pVolume), XDIM*YDIM*ZDIM*sizeof(GLubyte));
    infile.close();
    return true;
} else {
    return false;
}
```

2. Depending on the sampling box size, run three loops to iterate through the entire volume voxel by voxel:

```
vertices.clear();
int dx = XDIM/X_SAMPLING_DIST;
int dy = YDIM/Y_SAMPLING_DIST;
int dz = ZDIM/Z_SAMPLING_DIST;
glm::vec3 scale = glm::vec3(dx,dy,dz);
for(int z=0;z<ZDIM;z+=dz) {
    for(int y=0;y<YDIM;y+=dy) {
        for(int x=0;x<XDIM;x+=dx) {
            SampleVoxel(x,y,z, scale);
        }
    }
}
```

3. In each sampling step, estimate the volume density values at the eight corners of the sampling box:

```
GLubyte cubeCornerValues[8];
for( i = 0; i < 8; i++) {
    cubeCornerValues[i] = SampleVolume(
        x + (int)(a2fVertexOffset[i][0] *scale.x),
        y + (int)(a2fVertexOffset[i][1]*scale.y),
        z + (int)(a2fVertexOffset[i][2]*scale.z));
}
```

4. Estimate an edge flag value to identify the matching tetrahedra case based on the given isovalue:

```
int flagIndex = 0;
for( i= 0; i<8; i++) {
    if(cubeCornerValues[i]<= isoValue)
        flagIndex |= 1<<i;
}
edgeFlags = aiCubeEdgeFlags[flagIndex];
```

5. Use the lookup tables (a2iEdgeConnection) to find the correct edges for the case and then use the offset table (a2fVertexOffset) to find the edge vertices and normals. These tables are defined in the Tables.h header in the Chapter7/MarchingTetrahedra/ directory.

```
for(i = 0; i < 12; i++)
{
    if(edgeFlags & (1<<i))
    {
        float offset = GetOffset(cubeCornerValues[
            a2iEdgeConnection[i][0] ],
```

```

        cubeCornerValues[ a2iEdgeConnection[i][1] ] );
edgeVertices[i].x = x + (a2fVertexOffset[
a2iEdgeConnection[i][0]][0] + offset *
a2fEdgeDirection[i][0])*scale.x ;
edgeVertices[i].y = y + (a2fVertexOffset[
a2iEdgeConnection[i][0]][1] + offset *
a2fEdgeDirection[i][1])*scale.y ;
edgeVertices[i].z = z + (a2fVertexOffset[
a2iEdgeConnection[i][0]][2] + offset *
a2fEdgeDirection[i][2])*scale.z ;
edgeNormals[i] = GetNormal( (int)edgeVertices[i].x ,
(int)edgeVertices[i].y , (int)edgeVertices[i].z ) ;
}
}
}

```

- Finally, loop through the triangle connectivity table to connect the correct vertices and normals for the given case.

```

for(i = 0; i< 5; i++) {
if(a2iTriangleConnectionTable[flagIndex][3*i] < 0)
break;
for(int j= 0; j< 3; j++) {
int vertex = a2iTriangleConnectionTable
[flagIndex][3*i+j];
Vertex v;
v.normal = (edgeNormals[vertex]);
v.pos = (edgeVertices[vertex])*invDim;
vertices.push_back(v);
}
}

```

- After the marcher is finished, we pass the generated vertices to a vertex array object containing a vertex buffer object:

```

glGenVertexArrays(1, &volumeMarcherVAO);
glGenBuffers(1, &volumeMarcherVBO);
 glBindVertexArray(volumeMarcherVAO);
 glBindBuffer(GL_ARRAY_BUFFER, volumeMarcherVBO);
 glBufferData(GL_ARRAY_BUFFER, marcher->GetTotalVertices()*sizeof
(Vertex), marcher->GetVertexPointer(), GL_STATIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,sizeof(Vertex),0);
 glEnableVertexAttribArray(1);
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_
FALSE,sizeof(Vertex),(const GLvoid*)offsetof(Vertex, normal));

```

- For rendering of the generated geometry, we bind the Marching Tetrahedra VAO, bind our shader and then render the triangles. For this recipe, we output the per-vertex normals as color.

```
glBindVertexArray(volumeMarcherVAO);
shader.Use();
glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
glm::value_ptr(MVP*T));
glDrawArrays(GL_TRIANGLES, 0, marcher->GetTotalVertices());
shader.UnUse();
```

How it works...

For convenience, we wrap the entire recipe in a reusable class called `TetrahedraMarcher`. `Marching Tetrahedra`, as the name suggests, marches a sampling box throughout the whole volume dataset. To give a bird's eye view there are several cases to consider based on the distribution of density values at the vertices of the sampling cube. Based on the sampling values at the eight corners and the given isovalue, a flag index is generated. This flag index gives us the edge flag by a lookup in a table. This edge flag is then used in an edge lookup table, which is predefined for all possible edge configurations of the marching tetrahedron. The edge connection table is then used to find the appropriate offset for the corner values of the sampling box. These offsets are then used to obtain the edge vertices and normals for the given tetrahedral case. Once the list of edge vertices and normals are estimated, the triangle connectivity is obtained based on the given flag index.

Now we will detail the steps in the `Marching Tetrahedra` algorithm. First, the flag index is obtained by iterating through all eight sampling cube vertices and comparing the density value at the vertex location with the given isovalue as shown in the following code. The flag index is then used to retrieve the edge flags from the lookup table (`aiCubeEdgeFlags`).

```
flagIndex = 0;
for( i= 0; i<8; i++) {
    if(cubeCornerValues[i] <= isoValue)
        flagIndex |= 1<<i;
}
edgeFlags = aiCubeEdgeFlags[flagIndex];
```

The vertices and normals for the given index are stored in a local array by looking up the edge connection table (`a2iEdgeConnection`).

```
for(i = 0; i < 12; i++) {
    if(edgeFlags & (1<<i)) {
        float offset = GetOffset(cubeCornerValues[
            a2iEdgeConnection[i][0]], cubeCornerValues[
            a2iEdgeConnection[i][1]]);
        edgeVertices[i].x = x + (a2fVertexOffset[
            a2iEdgeConnection[i][0]][0] + offset *
```

```

        a2fEdgeDirection[i][0])*scale.x ;

    edgeVertices[i].y = y + (a2fVertexOffset[
        a2iEdgeConnection[i][0][1] + offset *
        a2fEdgeDirection[i][1])*scale.y ;
    edgeVertices[i].z = z + (a2fVertexOffset[
        a2iEdgeConnection[i][0][2] + offset *
        a2fEdgeDirection[i][2])*scale.z ;
    edgeNormals[i] = GetNormal( (int)edgeVertices[i].x ,
                                (int)edgeVertices[i].y ,
                                (int)edgeVertices[i].z ) ;
}
}

```

Finally, the triangle connectivity table (`a2iTriangleConnectionTable`) is used to obtain the proper vertex and normal ordering and these attributes are then stored into a vectors.

```

for(i = 0; i< 5; i++) {
    if(a2iTriangleConnectionTable[flagIndex][3*i] < 0)
        break;
    for(int j= 0; j< 3; j++) {
        int vertex = a2iTriangleConnectionTable[flagIndex][3*i+j];
        Vertex v;
        v.normal = (edgeNormals[vertex]);
        v.pos = (edgeVertices[vertex])*invDim;
        vertices.push_back(v);
    }
}

```

After the Marching Tetrahedra code is processed, we store the generated vertices and normals in a buffer object. In the rendering code, we bind the appropriate vertex array object, bind our shader and then draw the triangles. The fragment shader for this recipe outputs the per-vertex normals as colors.

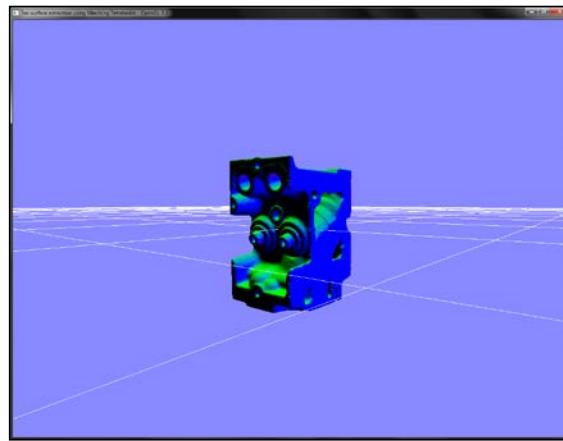
```

#version 330 core
layout(location = 0) out vec4 vFragColor;
smooth in vec3 outNormal;
void main() {
    vFragColor = vec4(outNormal,1);
}

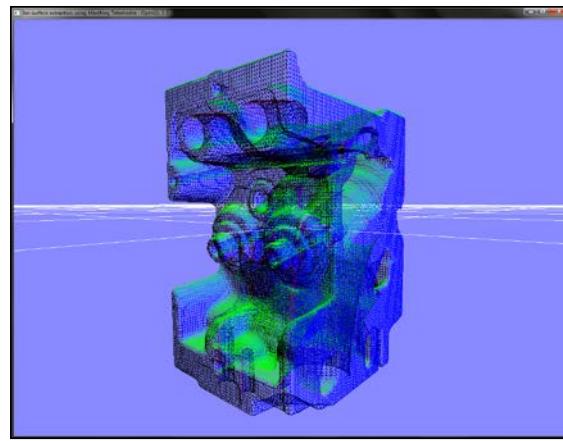
```

There's more...

The demo application for this recipe renders the engine dataset as shown in the following screenshot. The fragment shader renders the isosurface normals as color.



Pressing the **W** key toggles the wireframe display, which shows the underlying isosurface polygons for isovalue of 40 as shown in the following screenshot:



While in this recipe, we focused on the Marching Tetrahedra algorithm. There is another, more robust method of triangulation called **Marching Cubes**, which gives a more robust polygonisation as compared to the Marching Tetrahedra algorithm.

See also

- ▶ *Polygonising a scalar field*, Paul Bourke: <http://paulbourke.net/geometry/polygonise/>
- ▶ *Volume Rendering: Marching Cubes Algorithm*, <http://cns-alumni.bu.edu/~lavanya/Graphics/cs580/p5/web-page/p5.html>
- ▶ *An implementation of Marching Cubes and Marching Tetrahedra Algorithms*, <http://www.siafoo.net/snippet/100>

Implementing volumetric lighting using the half-angle slicing

In this recipe, we will implement volumetric lighting using the half-angle slicing technique. Instead of slicing the volume perpendicular to the viewing direction, the slicing direction is set between the light and the view direction vectors. This enables us to simulate light absorption slice by slice.

Getting ready

The code for this recipe is in the `Chapter7/HalfAngleSlicing` directory. As the name suggests, this recipe will build up on the 3D texture slicing code.

How to do it...

Let us start this recipe by following these simple steps:

1. Setup offscreen rendering using one FBO with two attachments: one for offscreen rendering of the light buffer and the other for offscreen rendering of the eye buffer.

```
glGenFramebuffers(1, &lightFBOID);
glGenTextures (1, &lightBufferID);
glGenTextures (1, &eyeBufferID);
glActiveTexture(GL_TEXTURE2);
lightBufferID = CreateTexture(IMAGE_WIDTH, IMAGE_HEIGHT, GL_RGBA16F, GL_RGBA);
eyeBufferID = CreateTexture(IMAGE_WIDTH, IMAGE_HEIGHT, GL_RGBA16F, GL_RGBA);
glBindFramebuffer(GL_FRAMEBUFFER, lightFBOID);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, lightBufferID, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, eyeBufferID, 0);
```

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE )
    printf("Light FBO setup successful !!! \n");
else
    printf("Problem with Light FBO setup");
```

The CreateTexture function performs the texture creation and texture format specification into a single function for convenience. This function is defined as follows:

```
GLuint CreateTexture(const int w,const int h,
                     GLenum internalFormat, GLenum format) {
    GLuint texid;
    glGenTextures(1, &texid);
    glBindTexture(GL_TEXTURE_2D, texid);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_CLAMP_TO_BORDER);
    glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, w, h, 0,
                format, GL_FLOAT, 0);
    return texid;
}
```

2. Load the volume data, as in the 3D texture slicing recipe:

```
std::ifstream infile(volume_file.c_str(),
                     std::ios_base::binary);
if(infile.good()) {
    GLubyte* pData = new GLubyte[XDIM*YDIM*ZDIM];
    infile.read(reinterpret_cast<char*>(pData),
                XDIM*YDIM*ZDIM*sizeof(GLubyte));
    infile.close();
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_3D, textureID);
    // set the texture parameters
    glTexImage3D(GL_TEXTURE_3D, 0, GL_RED, XDIM, YDIM, ZDIM, 0,
                 GL_RED, GL_UNSIGNED_BYTE, pData);
    GL_CHECK_ERRORS
    glGenerateMipmap(GL_TEXTURE_3D);
    return true;
} else {
```

```
    return false;
}
```

3. Similar to the shadow mapping technique, calculate the shadow matrix by multiplying the model-view and projection matrices of the light with the bias matrix:

```
MV_L=glm::lookAt(lightPosOS,glm::vec3(0,0,0),
                  glm::vec3(0,1,0));
P_L=glm::perspective(45.0f,1.0f,1.0f, 200.0f);
B=glm::scale(glm::translate(glm::mat4(1),
                            glm::vec3(0.5,0.5,0.5)), glm::vec3(0.5,0.5,0.5));
BP    = B*P_L;
S     = BP*MV_L;
```

4. In the rendering code, calculate the half vector by using the view direction vector and the light direction vector:

```
viewVec = -glm::vec3(MV[0][2], MV[1][2], MV[2][2]);
lightVec = glm::normalize(lightPosOS);
bIsViewInverted = glm::dot(viewVec, lightVec)<0;
halfVec = glm::normalize( (bIsViewInverted?-viewVec:viewVec) +
                           lightVec);
```

5. Slice the volume data as in the 3D texture slicing recipe. The only difference here is that instead of slicing the volume data in the direction perpendicular to the view, we slice it in the direction which is halfway between the view and the light vectors.

```
float max_dist = glm::dot(halfVec, vertexList[0]);
float min_dist = max_dist;
int max_index = 0;
int count = 0;
for(int i=1;i<8;i++) {
    float dist = glm::dot(halfVec, vertexList[i]);
    if(dist > max_dist) {
        max_dist = dist;
        max_index = i;
    }
    if(dist<min_dist)
        min_dist = dist;
}
//rest of the SliceVolume function as in 3D texture slicing but
//viewVec is changed to halfVec
```

6. In the rendering code, bind the FBO and then first clear the light buffer with the white color (1,1,1,1) and the eye buffer with the color (0,0,0,0):

```
glBindFramebuffer(GL_FRAMEBUFFER, lightFBOID);
glDrawBuffer(attachIDs[0]);
glClearColor(1,1,1,1);
glClear(GL_COLOR_BUFFER_BIT );
```

```
glDrawBuffer(attachIDs[1]);
glClearColor(0,0,0,0);
glClear(GL_COLOR_BUFFER_BIT );
```

7. Bind the volume VAO and then run a loop for the total number of slices. In each iteration, first render the slice in the eye buffer but bind the light buffer as the texture. Next, render the slice in the light buffer:

```
glBindVertexArray(volumeVAO);
for(int i = 0;i<num_slices;i++) {
    shaderShadow.Use();
    glUniformMatrix4fv(shaderShadow("MVP"), 1, GL_FALSE,
    glm::value_ptr(MVP));
    glUniformMatrix4fv(shaderShadow("S"), 1, GL_FALSE,
    glm::value_ptr(S));
    glBindTexture(GL_TEXTURE_2D, lightBuffer);
    DrawSliceFromEyePointOfView(i);

    shader.Use();
    glUniformMatrix4fv(shader("MVP"), 1, GL_FALSE,
    glm::value_ptr(P_L*MV_L));
    DrawSliceFromLightPointOfView(i);
}
```

8. For the eye buffer rendering step, swap the blend function based on whether the viewer is viewing in the direction of the light or opposite to it:

```
void DrawSliceFromEyePointOfView(const int i) {
    glDrawBuffer(attachIDs[1]);
    glViewport(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
    if(bIsViewInverted) {
        glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
    } else {
        glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
    }
    glDrawArrays(GL_TRIANGLES, 12*i, 12);
}
```

9. For the light buffer, we simply blend the slices using the conventional "over" blending:

```
void DrawSliceFromLightPointOfView(const int i) {
    glDrawBuffer(attachIDs[0]);
    glViewport(0, 0, IMAGE_WIDTH, IMAGE_HEIGHT);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDrawArrays(GL_TRIANGLES, 12*i, 12);
}
```

10. Finally, unbind the FBO and restore the default draw buffer. Next, set the viewport to the entire screen and then render the eye buffer on screen using a shader:

```
glBindVertexArray(0);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glDrawBuffer(GL_BACK_LEFT);
glViewport(0,0,WIDTH, HEIGHT);
glBindTexture(GL_TEXTURE_2D, eyeBufferID);
glBindVertexArray(quadVAOID);
quadShader.Use();
glDrawArrays(GL_TRIANGLES, 0, 6);
quadShader.UnUse();
glBindVertexArray(0);
```

How it works...

As the name suggests, this technique renders the volume by accumulating the intermediate results into two separate buffers by slicing the volume halfway between the light and the view vectors. When the scene is rendered from the point of view of the eye, the light buffer is used as texture to find out whether the current fragment is in shadow or not. This is carried out in the fragment shader by looking up the light buffer by using the shadow matrix as in the shadow mapping algorithm. In this step, the blending equation is swapped based on the direction of view with respect to the light direction vector. If the view is inverted, the blending direction is swapped from front-to-back to back-to-front using `glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE)`. On the other hand, if the view direction is not inverted, the blend function is set as `glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA)`. Note that here we have not used the over compositing since we premultiply the color with its alpha in the fragment shader (see Chapter7/HalfAngleSlicing/shaders/slicerShadow.frag), as shown in the following code:

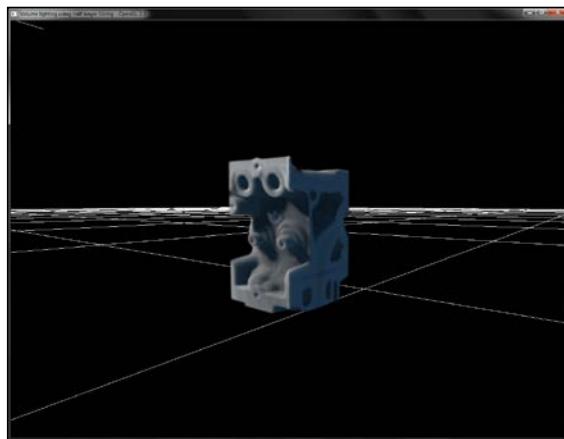
```
vec3 lightIntensity = textureProj(shadowTex, vLightUVW.xyw).xyz;
float density = texture(volume, vUV).r;
if(density > 0.1) {
    float alpha = clamp(density, 0.0, 1.0);
    alpha *= color.a;
    vFragColor = vec4(color.xyz*lightIntensity*alpha, alpha);
}
```

In the next step, the scene is rendered from the point of view of the light. This time, the normal over compositing is used. This ensures that the light contributions accumulate with each other similar to how light behaves in normal circumstances. In this case, we use the same fragment shader as was used in the 3D texture slicing recipe (see Chapter7/HalfAngleSlicing/shaders/textureSlicer.frag).

```
vFragColor = texture(volume, vUV).rrrr * color ;
```

There's more...

The demo application implementing this recipe renders the scene, as shown in the following screenshot, similar to the previous recipes. The light source position can be changed using the right mouse button. We can see the shadow changing dynamically for the scene. Attenuation of light is also controlled by setting a shader uniform. This is the reason why we can observe a bluish tinge in the output image.



Note that we cannot see the black halo around the volume dataset as was evident in earlier recipes. The reason for this is the if condition used in the fragment shader. We only perform these calculations if the current density value is greater than 0.1. This essentially removed air and other low intensity artifacts, producing a much better result.

See also

- ▶ *Chapter 39, Volume Rendering Techniques*, in *GPU Gems 1*. Available online at http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html
- ▶ *Chapter 6, Global Volume Illumination*, in *Real-time Volume Graphics*, AK Peters/CRC Press.

8

Skeletal and Physically-based Simulation on the GPU

In this chapter we will focus on the following topics:

- ▶ Implementing skeletal animation using matrix palette skinning
- ▶ Implementing skeletal animation using dual quaternion skinning
- ▶ Modeling cloth using transform feedback
- ▶ Implementing collision detection and response on a transform feedback-based cloth model
- ▶ Implementing a particle system using transform feedback

Introduction

Most of the real-time graphics applications have interactive elements. We have automated bots that move and animate in an interactive application. These elements include objects that are animated using preset sequences of frames. These are called **frame-by-frame animations**. There are other scene elements that have motion, which is derived using physical simulation. These are called **physically-based animations**. In addition, humanoid or character models have a special category of animations called **skeletal animation**. In this chapter, we will look at recipes for doing skeletal and physically-based simulation on the GPU in modern OpenGL.

Implementing skeletal animation using matrix palette skinning

When working with games and simulation systems, virtual characters are often used to give a detailed depiction of scenarios. Such characters are typically represented using a combination of bones and skin. The vertices of the 3D model are assigned influence weights (called **blend weights**) that control how much a bone influences that vertex. Up to four bones can influence a vertex. The process whereby bone weights are assigned to the vertices of a 3D model is called **skinning**. Each bone stores its transformation. These stored sequences of transformations are applied to every frame and every bone in the model and in the end, we get an animated character on the screen. This representation of animation is called **skeletal animation**. There are several methods for skeletal animation. One popular method is **matrix palette skinning**, which is also known as **linear blend skinning (LBS)**. This method will be implemented in this recipe.

Getting ready

The code for this recipe is contained in the Chapter8/MatrixPaletteSkinning directory. This recipe will be using the *Implementing EZMesh model loading* recipe from *Chapter 5, Mesh Model Formats and Particle Systems* and it will augment it with skeletal animation. The EZMesh format was developed by John Ratcliff and it is an easy-to-understand format for storing skeletal animation. Typical skeletal animation formats like COLLADA and FBX are needlessly complicated, where dozens of segments have to be parsed before the real content can be loaded. On the other hand, the EZMesh format stores all of the information in an XML-based format, which is easier to parse. It is the default skeletal animation format used in the NVIDIA PhysX sdk. More information about the EZMesh model format and loaders can be obtained from the references in the *See also* section of this recipe.

How to do it...

Let us start our recipe by following these simple steps:

1. Load the EZMesh model as we did in the *Implementing EZMesh loader* recipe from *Chapter 5, Mesh Model Formats and Particle System*. In addition to the model submeshes, vertices, normals, texture coordinates, and materials, we also load the skeleton information from the EZMesh file.

```
EzmLoader ezm;
if(!ezm.Load(mesh_filename.c_str(), skeleton, animations,
             submeshes, vertices, indices, material2ImageMap,
             min, max)) {
    cout<<"Cannot load the EZMesh file"<<endl;
    exit(EXIT_FAILURE);
}
```

2. Get the `MeshSystem` object from the `meshImportLibrary` object. Then load the bone transformations contained in the `EZMesh` file using the `MeshSystem::mSkeletons` array. This is carried out in the `EzmLoader::Load` function. Also generate absolute bone transforms from the relative transforms. This is done so that the transform of the child bone is influenced by the transform of the parent bone. This is continued up the hierarchy until the root bone. If the mesh is modeled in a positive Z axis system, we need to modify the orientation, positions, and scale by swapping Y and Z axes and changing the sign of one of them. This is done because we are using a positive Y axis system in OpenGL; otherwise, our mesh will be lying in the XZ plane rather than the XY plane. We obtain a combined matrix from the position orientation and scale of the bone. This is stored in the `xform` field, which is the relative transform of the bone.

```

if(ms->mSkeletonCount>0) {
    NVSHARE::MeshSkeleton* pSkel = ms->mSkeletons[0];
    Bone b;
    for(int i=0;i<pSkel->GetBoneCount();i++) {
        const NVSHARE::MeshBone pBone = pSkel->mBones[i];
        const int s = strlen(pBone.mName);
        b.name = new char[s+1];
        memset(b.name, 0, sizeof(char)*(s+1));
        strncpy_s(b.name,sizeof(char)*(s+1), pBone.mName, s);
        b.orientation = glm::quat(
            pBone.mOrientation[3],pBone.mOrientation[0],
            pBone.mOrientation[1],pBone.mOrientation[2]);
        b.position = glm::vec3( pBone.mPosition[0],
            pBone.mPosition[1],pBone.mPosition[2]);
        b.scale = glm::vec3(pBone.mScale[0], pBone.mScale[1],
            pBone.mScale[2]);

        if(!bYup) {
            float tmp = b.position.y;
            b.position.y = b.position.z;
            b.position.z = -tmp;
            tmp = b.orientation.y;
            b.orientation.y = b.orientation.z;
            b.orientation.z = -tmp;
            tmp = b.scale.y;
            b.scale.y = b.scale.z;
            b.scale.z = -tmp;
        }
    }

    glm::mat4 S = glm::scale(glm::mat4(1), b.scale);
    glm::mat4 R = glm::toMat4(b.orientation);
}

```

```
glm::mat4 T = glm::translate(glm::mat4(1), b.position);

b.xform = T*R*S;
b.parent = pBone.mParentIndex;
skeleton.push_back(b);
}

UpdateCombinedMatrices();
bindPose.resize(skeleton.size());
invBindPose.resize(skeleton.size());
animatedXform.resize(skeleton.size());
```

3. Generate the bind pose and inverse bind pose arrays from the stored bone transformations:

```
for(size_t i=0;i<skeleton.size();i++) {
    bindPose[i] = (skeleton[i].comb);
    invBindPose[i] = glm::inverse(bindPose[i]);
}
```

4. Store the blend weights and blend indices of each vertex in the mesh:

```
mesh.vertices[j].blendWeights.x = pMesh->mVertices[j].mWeight[0];
mesh.vertices[j].blendWeights.y = pMesh->mVertices[j].mWeight[1];
mesh.vertices[j].blendWeights.z = pMesh->mVertices[j].mWeight[2];
mesh.vertices[j].blendWeights.w = pMesh->mVertices[j].mWeight[3];
mesh.vertices[j].blendIndices[0] = pMesh->mVertices[j].mBone[0];
mesh.vertices[j].blendIndices[1] = pMesh->mVertices[j].mBone[1];
mesh.vertices[j].blendIndices[2] = pMesh->mVertices[j].mBone[2];
mesh.vertices[j].blendIndices[3] = pMesh->mVertices[j].mBone[3];
```

5. In the idle callback function, calculate the amount of time to spend on the current frame. If the amount has elapsed, move to the next frame and reset the time. After this, calculate the new bone transformations as well as new skinning matrices, and pass them to the shader:

```
QueryPerformanceCounter(&current);
dt = (double)(current.QuadPart - last.QuadPart) /
(double)freq.QuadPart;
last = current;
static double t = 0;
t+=dt;
NVSHARE::MeshAnimation* pAnim = &animations[0];
float framesPerSecond = pAnim->GetFrameCount()/
pAnim->GetDuration();
if( t > 1.0f/ framesPerSecond) {
    currentFrame++;
```

```
t=0;
}
if(bLoop) {
    currentFrame = currentFrame%pAnim->mFrameCount;
} else {
    currentFrame=max(-1,min(currentFrame,pAnim->mFrameCount-1));
}
if(currentFrame == -1) {
    for(size_t i=0;i<skeleton.size();i++) {
        skeleton[i].comb = bindPose[i];
        animatedXform[i] = skeleton[i].comb*invBindPose[i];
    }
}
else {
    for(int j=0;j<pAnim->mTrackCount;j++) {
        NVSHARE::MeshAnimTrack* pTrack = pAnim->mTracks[j];
        NVSHARE::MeshAnimPose* pPose =
        pTrack->GetPose(currentFrame);
        skeleton[j].position.x = pPose->mPos[0];
        skeleton[j].position.y = pPose->mPos[1];
        skeleton[j].position.z = pPose->mPos[2];

        glm::quat q;
        q.x = pPose->mQuat[0];
        q.y = pPose->mQuat[1];
        q.z = pPose->mQuat[2];
        q.w = pPose->mQuat[3];

        skeleton[j].scale = glm::vec3(pPose->mScale[0],
                                       pPose->mScale[1],
                                       pPose->mScale[2]);
        if(!bYup) {
            skeleton[j].position.y = pPose->mPos[2];
            skeleton[j].position.z = -pPose->mPos[1];
            q.y = pPose->mQuat[2];
            q.z = -pPose->mQuat[1];
            skeleton[j].scale.y = pPose->mScale[2];
            skeleton[j].scale.z = -pPose->mScale[1];
        }
        skeleton[j].orientation = q;
        glm::mat4 S =glm::scale(glm::mat4(1),skeleton[j].scale);
        glm::mat4 R = glm::toMat4(q);
        glm::mat4 T = glm::translate(glm::mat4(1), skeleton[j].
position);
    }
}
```

```
skeleton[j].xform = T*R*S;
Bone& b = skeleton[j];

if(b.parent== -1)
    b.comb = b.xform;
else
    b.comb = skeleton[b.parent].comb * b.xform;

animatedXform[j] = b.comb * invBindPose[j];
}

}

shader.Use();
glUniformMatrix4fv(shader("Bones"), animatedXform.size(),
GL_FALSE, glm::value_ptr(animatedXform[0]));
shader.UnUse();
shader.UnUse();
```

How it works...

There are two parts of this recipe: generation of skinning matrices and the calculation of GPU skinning in the vertex shader. To understand the first step, we will start with the different transforms that will be used in skinning. Typically, in a simulation or game, a transform is represented as a 4×4 matrix. For skeletal animation, we have a collection of bones. Each bone has a **local transform** (also called **relative transform**), which tells how the bone is positioned and oriented with respect to its parent bone. If the bone's local transform is multiplied to the global transform of its parent, we get the **global transform** (also called **absolute transform**) of the bone. Typically, the animation formats store the local transforms of the bones in the file. The user application uses this information to generate the global transforms.

We define our bone structure as follows:

```
struct Bone {
    glm::quat orientation;
    glm::vec3 position;
    glm::mat4 xform, comb;
    glm::vec3 scale;
    char* name;
    int parent;
};
```

The first field is `orientation`, which is a quaternion storing the orientation of bone in space relative to its parent. The `position` field stores its position relative to its parent. The `xform` field is the local (relative) transform, and the `comb` field is the global (absolute) transform. The `scale` field contains the scaling transformation of the bone. In the big picture, the `scale` field gives the scaling matrix (S), the `orientation` field gives the rotation matrix (R), and the `position` field gives the translation matrix (T). The combined matrix $T \cdot R \cdot S$ gives us the relative transform that is calculated when we load the bone information from the EZMesh file in the second step.

The `name` field is the unique name of the bone in the skeleton. Finally, the `parent` field stores the index of the parent of the current bone in the skeleton array. For the root bone, the parent is -1. For all of the other bones, it will be a number starting from 0 to $N-1$, where N is the total number of bones in the skeleton.

After we have loaded and stored the relative transforms of each bone in the skeleton, we iterate through each bone to obtain its absolute transform. This is carried out in the `UpdateCombinedMatrices` function in `Chapter8/MatrixPaletteSkinning/main.cpp`.

```
for(size_t i=0;i<skeleton.size();i++) {  
    Bone& b = skeleton[i];  
    if(b.parent==-1)  
        b.comb = b.xform;  
    else  
        b.comb = skeleton[b.parent].comb * b.xform;  
}
```

After generating the absolute transforms of each bone, we store the bind pose and inverse bind pose matrices of the skeleton. Simply put, bind pose is the absolute transforms of the bones in the non-animated state (that is, when no animation is applied). This is usually when the skeleton is attached (skinned) to the geometry. In other words, it is the default pose of the skeletal animated mesh. Typically, bones can be in any bind pose (usually, for humanoid characters, the character may be in A pose, T pose, and so on based on the convention used). Typically, the inverse bind pose is stored at the time of initialization. So, continuing to the previous skeleton example, we can get the bind pose and inverse bind pose matrices as follows:

```
for(size_t i=0;i < skeleton.size(); i++) {  
    bindPose[i] = skeleton[i].comb;  
    invBindPose[i] = glm::inverse(bindPose[i]);  
}
```

Note that we do this once at initialization, so that we do not have to calculate the bind pose's inverse every frame, as it is required during animation.

When we apply any new transformation (an animation sequence for example) to the skeleton, we have to first undo the bind pose transformation. This is done by multiplying the animated transformation with the inverse of the bind pose transformation. This is required because the given relative transformations will add to the existing transformations of the bone and, if the bind pose transformation is not undone, the animation output will be wrong.

The final matrix that we get from this process is called the **skinning matrix** (also called the **final bone matrix**). Continuing from the example given in the previous paragraph, let's say we have modified the relative transforms of bone using the animation sequence. We can then generate the skinning matrix as follows:

```
for(size_t i=0;i < skeleton.size(); i++) {  
    Bone& b = skeleton[i];  
    if(b.parent== -1)  
        b.comb = b.xform;  
    else  
        b.comb = skeleton[b.parent].comb * b.xform;  
    animatedXForm[i] = b.comb*invBindPose[i];  
}
```

One thing to note here is the order of the different matrices. As you can see, we right multiply the inverse bind pose matrix with the combined bone transform. We put it this way because OpenGL and glm matrices work right to left. So the inverse of bind pose matrix will be multiplied by the given vertex first. Then it will be multiplied with the local transform (`xform`) of the current bone. Finally, it will be multiplied with the global transform (`comb`) of its parent to get the final transformation matrix.

After we have calculated the skinning matrices, we pass these to the GPU in a single call:

```
shader.Use();  
glUniformMatrix4fv(shader("Bones"), animatedXForm.size(),  
GL_FALSE, glm::value_ptr(animatedXForm[0]));  
shader.UnUse();
```

To make sure that the size of the bones array is correct in the vertex shader, we append text to the shader dynamically by using the overloaded `GLSLShader::LoadFromFile` function.

```
stringstream str( ios_base::app | ios_base::out);  
str<<"\nconst int NUM_BONES="<<skeleton.size()<<";"<<endl;  
str<<"uniform mat4 Bones [NUM_BONES];"\n;  
shader.LoadFromFile(GL_VERTEX_SHADER, "shaders/shader.vert", str.  
str());  
shader.LoadFromFile(GL_FRAGMENT_SHADER, "shaders/shader.frag");
```

This ensures that our vertex shader has the same number of bones as our mesh file.



For this simple recipe we have modified the shader code at loading time before compilation. Such shader modification must not be done at runtime as this will require a recompilation of the shader and will likely hamper application performance.

The `Vertex` struct storing all of our per-vertex attributes is defined as follows:

```
struct Vertex {
    glm::vec3 pos,
        normal;
    glm::vec2 uv;
    glm::vec4 blendWeights;
    glm::ivec4 blendIndices;
};
```

The vertices array is filled in by the `EzmLoader::Load` function. We generate a vertex array object with a vertex buffer object to store our interleaved per-vertex attributes:

```
glGenVertexArrays(1, &vaoID);
glGenBuffers(1, &vboVerticesID);
glGenBuffers(1, &vboIndicesID);

glBindVertexArray(vaoID);
glBindBuffer(GL_ARRAY_BUFFER, vboVerticesID);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*vertices.size(),
&(vertices[0].pos.x), GL_DYNAMIC_DRAW);

 glEnableVertexAttribArray(shader["vVertex"]);
 glVertexAttribPointer(shader["vVertex"], 3, GL_FLOAT, GL_
FALSE, sizeof(Vertex), 0);

 glEnableVertexAttribArray(shader["vNormal"]);
 glVertexAttribPointer(shader["vNormal"], 3, GL_FLOAT, GL_FALSE,
sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, normal)) );

 glEnableVertexAttribArray(shader["vUV"]);
 glVertexAttribPointer(shader["vUV"], 2, GL_FLOAT, GL_FALSE,
sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, uv)) );

 glEnableVertexAttribArray(shader["vBlendWeights"]);
 glVertexAttribPointer(shader["vBlendWeights"], 4, GL_FLOAT,
GL_FALSE, sizeof(Vertex), (const GLvoid*)(offsetof(Vertex,
blendWeights)) );

 glEnableVertexAttribArray(shader["viBlendIndices"]);
 glVertexAttribIPointer(shader["viBlendIndices"], 4, GL_INT,
sizeof(Vertex), (const GLvoid*)(offsetof(Vertex, blendIndices)) );
```



Note that for blend indices we use the `glVertexAttribIPointer` function, as the attribute (`vBlendIndices`) is defined as `ivec4` in the vertex shader.

Finally, in the rendering code, we set the vertex array object and use the shader program. Then we iterate through all of the submeshes. We then set the material texture for the current submesh and set the shader uniforms. Finally, we issue a `glDrawElements` call:

```
glBindVertexArray(vaoID);
shader.Use();
glUniformMatrix4fv(shader("MV"), 1, GL_FALSE, glm::value_ptr(MV));
glUniformMatrix3fv(shader("N"), 1, GL_FALSE, glm::value_ptr(glm::inverseTranspose(glm::mat3(MV))));
glUniformMatrix4fv(shader("P"), 1, GL_FALSE, glm::value_ptr(P));
glUniform3fv(shader("light_position"), 1, &(lightPosOS.x));
for(size_t i=0;i<submeshes.size();i++) {
    if(strlen(submeshes[i].materialName)>0) {
        GLuint id = materialMap[ material2ImageMap[
            submeshes[i].materialName]];
        GLint whichID[1];
        glGetIntegerv(GL_TEXTURE_BINDING_2D, whichID);
        if(whichID[0] != id)
            glBindTexture(GL_TEXTURE_2D, id);
        glUniform1f(shader("useDefault"), 0.0);
    } else {
        glUniform1f(shader("useDefault"), 1.0);
    }
    glDrawElements(GL_TRIANGLES, submeshes[i].indices.size(),
                  GL_UNSIGNED_INT, &submeshes[i].indices[0]);
} //end for
shader.UnUse();
}
```

The matrix palette skinning is carried out on the GPU using the vertex shader (Chapter8/MatrixPaletteSkinning/shaders/shader.vert). We simply use the blend indices and blend weights to calculate the correct vertex position and normal based on the combined influence of all of the effecting bones. The `Bones` array contains the skinning matrices that we generated earlier. The complete vertex shader is as follows:



Note that the `Bones` uniform array is not declared in the shader, as it is filled in the shader code dynamically as was shown earlier.

```
#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;
layout(location = 2) in vec2 vUV;
layout(location = 3) in vec4 vBlendWeights;
layout(location = 4) in ivec4 viBlendIndices;
smooth out vec2 vUVout;
uniform mat4 P;
uniform mat4 MV;
uniform mat3 N;
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;
void main() {
    vec4 blendVertex=vec4(0);
    vec3 blendNormal=vec3(0);
    vec4 vVertex4 = vec4(vVertex,1);

    int index = viBlendIndices.x;
    blendVertex = (Bones[index] * vVertex4) * vBlendWeights.x;
    blendNormal = (Bones[index] * vec4(vNormal, 0.0)).xyz *
        vBlendWeights.x;

    index = viBlendIndices.y;
    blendVertex = ((Bones[index] * vVertex4) * vBlendWeights.y) +
        blendVertex;
    blendNormal = (Bones[index] * vec4(vNormal, 0.0)).xyz *
        vBlendWeights.y + blendNormal;

    index = viBlendIndices.z;
    blendVertex = ((Bones[index] * vVertex4) * vBlendWeights.z)
        + blendVertex;
    blendNormal = (Bones[index] * vec4(vNormal, 0.0)).xyz *
        vBlendWeights.z + blendNormal;

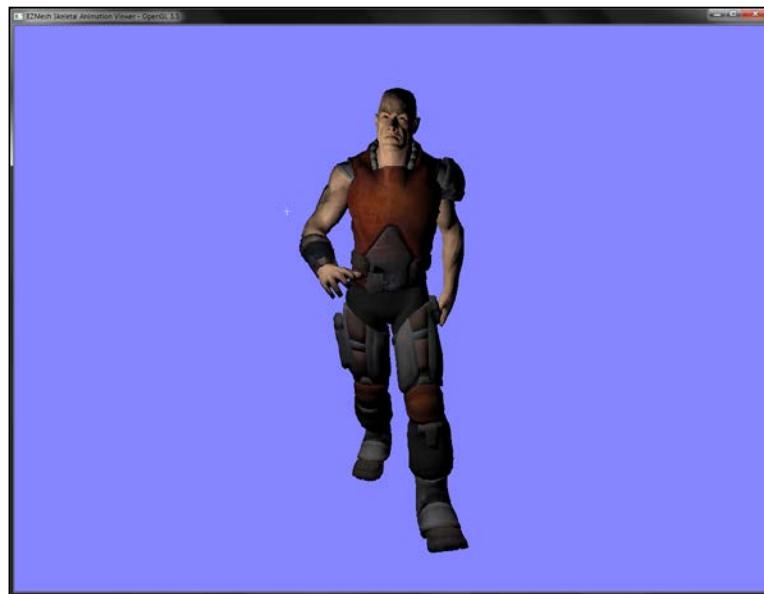
    index = viBlendIndices.w;
    blendVertex = ((Bones[index] * vVertex4) * vBlendWeights.w)
        + blendVertex;
    blendNormal = (Bones[index] * vec4(vNormal, 0.0)).xyz *
        vBlendWeights.w + blendNormal;

    vEyeSpacePosition = (MV*blendVertex).xyz;
    vEyeSpaceNormal = normalize(N*blendNormal);
    vUVout=vUV;
    gl_Position = P*vec4(vEyeSpacePosition,1);
}
```

The fragment shader uses the attenuated point light source for illumination as we have seen in the *Implementing per-fragment point light with attenuation* recipe in *Chapter 4, Lights and Shadows*.

There's more...

The output from the demo application for this recipe shows the `dude.ezm` model animating using the matrix palette skinning technique as shown in the following figure. The light source can be rotated by right-clicking on it and dragging. Pressing the `/` key stops the loop playback.



See also

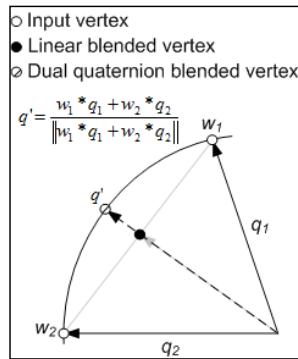
- ▶ NVIDIA DirectX SDK 9.0 Matrix Skinning demo at http://http.download.nvidia.com/developer/SDK/Individual_Samples/DEMOS/Direct3D9/src/HLSL_PaletteSkin/docs/HLSL_PaletteSkin.pdf
- ▶ John Ratcliff code suppository containing a lot of useful tools, including the EZMesh format specifications and loaders available online at <http://codesuppository.blogspot.sg/2009/11/test-application-for-meshimport-library.html>
- ▶ Improved Skinning demo in the NVIDIA sdk at http://http.download.nvidia.com/developer/SDK/Individual_Samples/samples.html

Implementing skeletal animation using dual quaternion skinning

Matrix palette skinning suffers from candy wrapping artefacts, especially in regions like shoulder and elbow, where there are several rotations across various axes. If dual quaternion skinning is employed, these artefacts are minimized. In this recipe we will implement skeletal animation using dual quaternion skinning.

Before understanding dual quaternions, let us first see what quaternions are. **Quaternions** are a mathematical entity containing three imaginary dimensions (which specify the axis of rotation) and a real dimension (which specifies the angle of rotation). Quaternions are used in 3D graphics to represent rotation, since they do not suffer from gimbal lock, as Euler angles do. In order to store translation with rotation simultaneously, dual quaternions are used to store dual number coefficients instead of real ones. Instead of four components, as in a quaternion, dual quaternions have eight components.

Even in dual quaternion skinning, the linear blend method is used. However, due to the nature of transformation in dual quaternion, spherical blending is preferred. After linear blending, the resulting dual quaternion is renormalized, which generates a spherical blending result, which is a better approximation as compared to the linear blend skinning. This whole process is illustrated by the following figure:



Getting ready

The code for this recipe is contained in the Chapter8/DualQuaternionSkinning folder. We will be building on top of the previous recipe and replace the skinning matrices with dual quaternions.

How to do it...

Converting linear blend skinning to dual quaternion skinning requires the following steps:

1. Load the EZMesh model as we did in the *Implementing EZMesh loader* recipe from *Chapter 5, Mesh Model Formats and Particle System*:

```
if (!ezm.Load(mesh_filename.c_str(), skeleton, animations,
    submeshes, vertices, indices, material2ImageMap, min, max)) {
    cout<<"Cannot load the EZMesh file"<<endl;
    exit(EXIT_FAILURE); }
```

2. After loading up the mesh, materials, and textures, load the bone transformations contained in the EZMesh file using the `MeshSystem::mSkeletons` array as we did in the previous recipe. In addition to the bone matrices, also store the bind pose and inverse bind pose matrices as we did in the previous recipe. Instead of storing the skinning matrices, we initialize a vector of dual quaternions. Dual quaternions are a different representation of the skinning matrices.

```
UpdateCombinedMatrices();
bindPose.resize(skeleton.size());
invBindPose.resize(skeleton.size());
animatedXform.resize(skeleton.size());
dualQuaternions.resize(skeleton.size());
for(size_t i=0;i<skeleton.size();i++) {
    bindPose[i] = (skeleton[i].comb);
    invBindPose[i] = glm::inverse(bindPose[i]);
}
```

3. Implement the idle callback function as in the previous recipe. Here, in addition to calculating the skinning matrix, also calculate the dual quaternion for the given skinning matrix. After all of the joints are done, pass the dual quaternion to the shader:

```
glm::mat4 S = glm::scale(glm::mat4(1), skeleton[j].scale);
glm::mat4 R = glm::toMat4(q);
glm::mat4 T = glm::translate(glm::mat4(1),
    skeleton[j].position);
skeleton[j].xform = T*R*S;
Bone& b = skeleton[j];
if (b.parent == -1)
    b.comb = b.xform;
else
    b.comb = skeleton[b.parent].comb * b.xform;
animatedXform[j] = b.comb * invBindPose[j];
glm::vec3 t = glm::vec3( animatedXform[j][3][0],
    animatedXform[j][3][1], animatedXform[j][3][2] );
dualQuaternions[j].QuatTrans2UDQ(
```

```

glm::toQuat(animatedXform[j]), t);
...
shader.Use();
glUniform4fv(shader("Bones"), skeleton.size()*2,
&(dualQuaternions[0].ordinary.x));
shader.UnUse();

```

4. In the vertex shader (Chapter8/DualQuaternionSkinning/shaders/shader.vert), calculate the skinning matrix from the passed dual quaternion and blend weights of the given vertices. Then proceed with the skinning matrix as we did in the previous recipe:

```

#version 330 core
layout(location = 0) in vec3 vVertex;
layout(location = 1) in vec3 vNormal;
layout(location = 2) in vec2 vUV;
layout(location = 3) in vec4 vBlendWeights;
layout(location = 4) in ivec4 viBlendIndices;
smooth out vec2 vUVout;
uniform mat4 P;
uniform mat4 MV;
uniform mat3 N;
smooth out vec3 vEyeSpaceNormal;
smooth out vec3 vEyeSpacePosition;

void main() {
    vec4 blendVertex=vec4(0);
    vec3 blendNormal=vec3(0);
    vec4 blendDQ[2];
    float yc = 1.0, zc = 1.0, wc = 1.0;
    if (dot(Bones[viBlendIndices.x * 2],
            Bones[viBlendIndices.y * 2]) < 0.0)
        yc = -1.0;
    if (dot(Bones[viBlendIndices.x * 2],
            Bones[viBlendIndices.z * 2]) < 0.0)
        zc = -1.0;
    if (dot(Bones[viBlendIndices.x * 2],
            Bones[viBlendIndices.w * 2]) < 0.0)
        wc = -1.0;
    blendDQ[0] = Bones[viBlendIndices.x * 2] * vBlendWeights.x;
    blendDQ[1] = Bones[viBlendIndices.x * 2 + 1] *
                 vBlendWeights.x;
    blendDQ[0] += yc*Bones[viBlendIndices.y * 2] *
                  vBlendWeights.y;

```

```

blendDQ[1] += yc*Bones[viBlendIndices.y * 2 + 1] *
    vBlendWeights.y;
blendDQ[0] += zc*Bones[viBlendIndices.z * 2] *
    vBlendWeights.z;
blendDQ[1] += zc*Bones[viBlendIndices.z * 2 + 1] *
    vBlendWeights.z;
blendDQ[0] += wc*Bones[viBlendIndices.w * 2] *
    vBlendWeights.w;
blendDQ[1] += wc*Bones[viBlendIndices.w * 2 + 1] *
    vBlendWeights.w;
mat4 skinTransform = dualQuatToMatrix(blendDQ[0],
    blendDQ[1]);
blendVertex = skinTransform*vec4(vVertex,1);
blendNormal = (skinTransform*vec4(vNormal,0)).xyz;
vEyeSpacePosition = (MV*blendVertex).xyz;
vEyeSpaceNormal = N*blendNormal;
vUVout=vUV;
gl_Position = P*vec4(vEyeSpacePosition,1);
}

```

To convert the given dual quaternion to a matrix, we define a function `dualQuatToMatrix`. This gives us a matrix, which we can then multiply with the vertex to obtain the transformed result.

How it works...

The only difference in this recipe and the previous recipe is the creation of a dual quaternion from the skinning matrix on the CPU, and its conversion back to a matrix in the vertex shader. After we have obtained the skinning matrices, we convert them into a dual quaternion array by using the `dual_quat::QuatTrans2UDQ` function that gets a dual quaternion from a rotation quaternion and a translation vector. This function is defined as follows in the `dual_quat` class (in Chapter8/DualQuaternionSkinning/main.cpp):

```

void QuatTrans2UDQ(const glm::quat& q0, const glm::vec3& t) {
    ordinary = q0;
    dual.w = -0.5f * (t.x * q0.x + t.y * q0.y + t.z * q0.z);
    dual.x = 0.5f * (t.x * q0.w + t.y * q0.z - t.z * q0.y);
    dual.y = 0.5f * (-t.x * q0.z + t.y * q0.w + t.z * q0.x);
    dual.z = 0.5f * (t.x * q0.y - t.y * q0.x + t.z * q0.w);
}

```

The dual quaternion array is then passed to the shader instead of the bone matrices. In the vertex shader, we first do a dot product of the ordinary quaternion with the dual quaternion. If the dot product of the two quaternions is less than zero, it means they are both facing in opposite direction. We thus subtract the quaternion from the blended dual quaternion, otherwise we add it to the blended dual quaternion:

```

float yc = 1.0, zc = 1.0, wc = 1.0;

if (dot(Bones[viBlendIndices.x * 2],
         Bones[viBlendIndices.y * 2]) < 0.0)
    yc = -1.0;

if (dot(Bones[viBlendIndices.x * 2],
         Bones[viBlendIndices.z * 2]) < 0.0)
    zc = -1.0;

if (dot(Bones[viBlendIndices.x * 2],
         Bones[viBlendIndices.w * 2]) < 0.0)
    wc = -1.0;

blendDQ[0] = Bones[viBlendIndices.x * 2] * vBlendWeights.x;
blendDQ[1] = Bones[viBlendIndices.x * 2 + 1] * vBlendWeights.x;

blendDQ[0] += yc*Bones[viBlendIndices.y * 2] * vBlendWeights.y;
blendDQ[1] += yc*Bones[viBlendIndices.y * 2 + 1] * vBlendWeights.y;

blendDQ[0] += zc*Bones[viBlendIndices.z * 2] * vBlendWeights.z;
blendDQ[1] += zc*Bones[viBlendIndices.z * 2 + 1] * vBlendWeights.z;

blendDQ[0] += wc*Bones[viBlendIndices.w * 2] * vBlendWeights.w;
blendDQ[1] += wc*Bones[viBlendIndices.w * 2 + 1] * vBlendWeights.w;

```

The blended dual quaternion (blendDQ) is then converted to a matrix by the `dualQuatToMatrix` function, which is defined as follows:

```

mat4 dualQuatToMatrix(vec4 Qn, vec4 Qd) {
    mat4 M;
    float len2 = dot(Qn, Qn);
    float w = Qn.w, x = Qn.x, y = Qn.y, z = Qn.z;
    float t0 = Qd.w, t1 = Qd.x, t2 = Qd.y, t3 = Qd.z;

    M[0][0] = w*w + x*x - y*y - z*z;
    M[0][1] = 2 * x * y + 2 * w * z;
    M[0][2] = 2 * x * z - 2 * w * y;
    M[0][3] = 0;

    M[1][0] = 2 * x * y - 2 * w * z;
    M[1][1] = w * w + y * y - x * x - z * z;
    M[1][2] = 2 * y * z + 2 * w * x;
    M[1][3] = 0;
}

```

```
M[2][0] = 2 * x * z + 2 * w * y;
M[2][1] = 2 * y * z - 2 * w * x;
M[2][2] = w * w + z * z - x * x - y * y;
M[2][3] = 0;

M[3][0] = -2 * t0 * x + 2 * w * t1 - 2 * t2 * z + 2 * y * t3;
M[3][1] = -2 * t0 * y + 2 * t1 * z - 2 * x * t3 + 2 * w * t2;
M[3][2] = -2 * t0 * z + 2 * x * t2 + 2 * w * t3 - 2 * t1 * y;
M[3][3] = len2;

M /= len2;

return M;
}
```

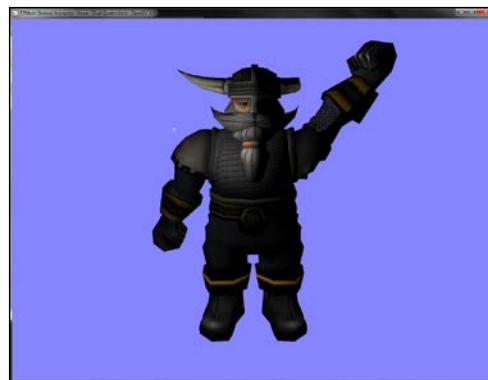
The returned matrix is then multiplied with the given vertex/normal, and then the eye space position/normal and texture coordinates are obtained. Finally, the clip space position is calculated:

```
mat4 skinTransform = dualQuatToMatrix(blendDQ[0], blendDQ[1]);
blendVertex = skinTransform*vec4(vVertex,1);
blendNormal = (skinTransform*vec4(vNormal,0)).xyz;
vEyeSpacePosition = (MV*blendVertex).xyz;
vEyeSpaceNormal = N*blendNormal;
vUVout=vUV;
gl_Position = P*vec4(vEyeSpacePosition,1);
```

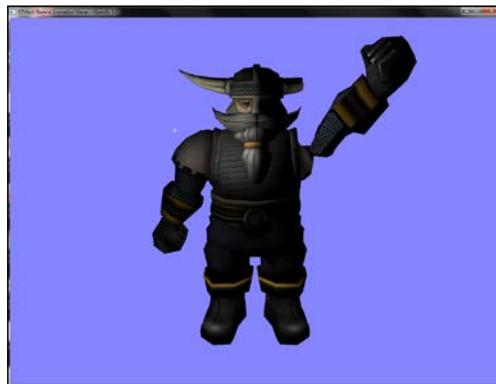
The fragment shader follows the similar steps as it did in the previous recipe to output the lit textured fragments.

There's more...

The demo application for this recipe renders the `dwarf_anim.ezm` skeletal model. Even with extreme rotation at the shoulder joint, the output does not suffer from candy wrapper artefacts as shown in the following figure:



On the other hand, if we use the matrix palette skinning, we get the following output, which clearly shows the candy wrapper artefacts:



See also

- ▶ Skinning with Dual Quaternions at <http://isg.cs.tcd.ie/projects/DualQuaternions/>
- ▶ Skinning with Dual Quaternions demo in NVIDIA DirectX sdk 10.5 at <http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html>
- ▶ Dual Quaternion Google Summer of Code 2011 implementation in OGRE at <http://www.ogre3d.org/tikiwiki/tiki-index.php?page=SoC2011%20Dual%20Quaternion%20Skinning>

Modeling cloth using transform feedback

In this recipe we will use the transform feedback mechanism of the modern GPU to model cloth. **Transform feedback** is a special mode of modern GPU in which the vertex shader can directly output to a buffer object. This allows developers to do complex computations without affecting the rest of the rendering pipeline. We will elaborate how to use this mechanism to simulate cloth entirely on the GPU.

From the implementation point of view in modern OpenGL, transform feedback exists as an OpenGL object similar to textures. Working with transform feedback object requires two steps: first, generation of transform feedback with specification of shader outputs, and second, usage of the transform feedback for simulation and rendering. We generate it by calling the `glGetTransformFeedbacks` function and passing it the number of objects and the variable to store the returned IDs. After the object is created, it is bound to the current OpenGL context by calling `glBindTransformFeedback`, and its only parameter is the ID of the transform feedback object we are interested to bind.

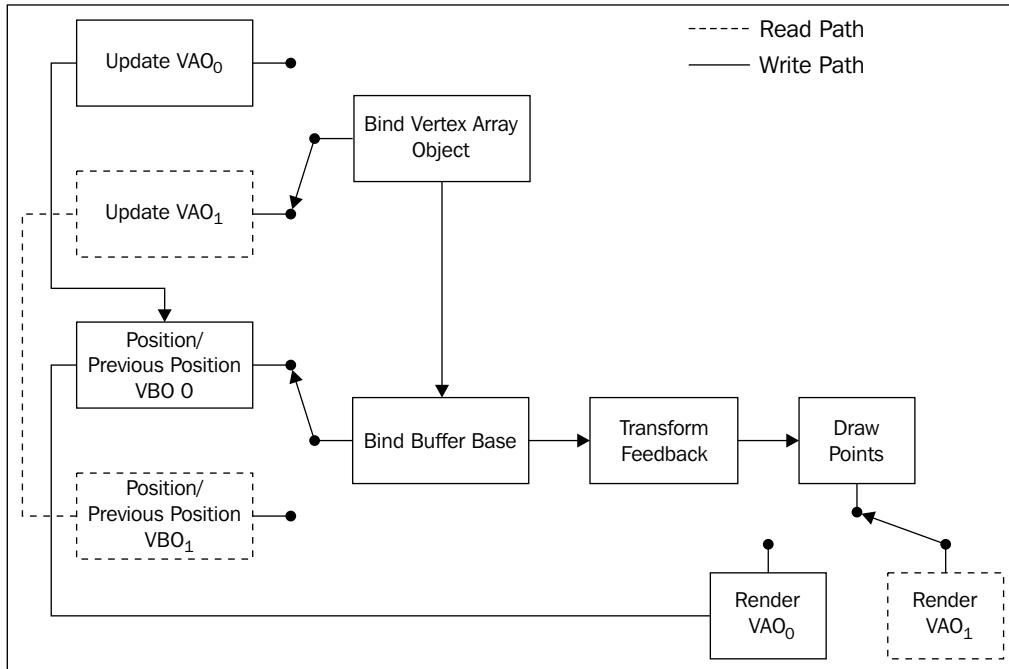
Next, we need to register the vertex attributes that we want to record in a transform feedback buffer. This is done through the `glTransformFeedbackVaryings` function. The parameters this function requires are in the following order: the shader program object, the number of outputs from the shader, the names of the attributes, and the recording mode. Recording mode can be either `GL_INTERLEAVED_ATTRIBS` (which means that the attributes will all be stored in a single interleaved buffer object) or `GL_SEPARATE_ATTRIBS` (which means each attribute will be stored in its own buffer object). Note that the shader program has to be relinked after the shader output varyings are specified.

We also have to set up our buffer objects that are going to store the attributes' output through transform feedback. At the rendering stage, we first set up our shader and the required uniforms. Then, we bind out vertex array objects storing out buffer object binding. Next, we bind the buffer object for transform feedback by calling the `glBindBufferBase` function. The first parameter is the index and the second parameter is the buffer object ID, which will store the shader output attribute. We can bind as many objects as we need, but the total calls to this function must be at least equal to the total output attributes from the vertex shader. Once the buffers are bound, we can initiate transform feedback by issuing a call to `glBeginTransformFeedback` and the parameter to this function is the output primitive type. We then issue our `glDraw*` call and then call `glEndTransformFeedback`.

[ OpenGL 4.0 and above provide a very convenient function, `glDrawTransformFeedback`. We just give it out primitive type and it automatically renders our primitives based on the total number of outputs from the vertex shader. In addition, OpenGL 4.0 provides the ability to pause/resume the transform feedback object as well as outputting to multiple transform feedback streams.]

For the cloth simulation implementation using transform feedback, this is how we proceed. We store the current and previous position of the cloth vertices into a pair of buffer objects. To have convenient access to the buffer objects, we store these into a pair of vertex array objects. Then in order to deform the cloth, we run a vertex shader that inputs the current and previous positions from the buffer objects. In the vertex shader, the internal and external forces are calculated for each pair of cloth vertices and then acceleration is calculated. Using Verlet integration, the new vertex position is obtained. The new and previous positions are output from the vertex shader, so they are written out to the attached transform feedback buffers. Since we have a pair of vertex array objects, we ping pong between the two. This process is continued and the simulation proceeds forward.

The whole process is well summarized by the following figure:



More details of the inner workings of this method are detailed in the reference in the See also section.

Getting ready

The code for this recipe is contained in the Chapter8/TransformfeedbackCloth folder.

How to do it...

Let us start the recipe by following these simple steps:

1. Generate the geometry and topology for a piece of cloth by creating a set of points and their connectivity. Bind this data to a buffer object. The vectors x and x_last store the current and last position respectively, and the vector F stores the force for each vertex:

```

vector<GLushort> indices;
vector<glm::vec4> X;
vector<glm::vec4> X_last;
vector<glm::vec3> F;
  
```

```

indices.resize( numX*numY*2*3 );
X.resize(total_points);
X_last.resize(total_points);
F.resize(total_points);
for(int j=0;j<=numY;j++) {
    for(int i=0;i<=numX;i++) {
        X[count] = glm::vec4( ((float(i)/(u-1)) *2-1)* hsize,
        sizeX+1, ((float(j)/(v-1) )* sizeY),1);
        X_last [count] = X[count];
        count++;
    }
}
GLushort* id=&indices[0];
for (int i = 0; i < numY; i++) {
    for (int j = 0; j < numX; j++) {
        int i0 = i * (numX+1) + j;
        int i1 = i0 + 1;
        int i2 = i0 + (numX+1);
        int i3 = i2 + 1;
        if ((j+i)%2) {
            *id++ = i0; *id++ = i2; *id++ = i1;
            *id++ = i1; *id++ = i2; *id++ = i3;
        } else {
            *id++ = i0; *id++ = i2; *id++ = i3;
            *id++ = i0; *id++ = i3; *id++ = i1;
        }
    }
}
glGenVertexArrays(1, &clothVAOID);
glGenBuffers (1, &clothVBOVerticesID);
glGenBuffers (1, &clothVBOIndicesID);
 glBindVertexArray(clothVAOID);
 glBindBuffer (GL_ARRAY_BUFFER, clothVBOVerticesID);
 glBufferData (GL_ARRAY_BUFFER, sizeof(float)*4*X.size(),
 &X[0].x, GL_STATIC_DRAW);
 glEnableVertexAttribArray(0);
 glVertexAttribPointer (0, 4, GL_FLOAT, GL_FALSE,0,0);
 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, clothVBOIndicesID);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER,
 sizeof(GLushort)*indices.size(), &indices[0], GL_STATIC_DRAW);
 glBindVertexArray(0);

```

2. Create two pairs of **vertex array objects (VAO)**, one pair for rendering and another pair for update of points. Bind two buffer objects (containing current positions and previous positions) to the update VAO, and one buffer object (containing current positions) to the render VAO. Also attach an element array buffer for geometry indices. Set the buffer object usage parameter as `GL_DYNAMIC_COPY`. This usage parameter hints to the GPU that the contents of the buffer object will be frequently changed, and it will be read in OpenGL or used as a source for GL commands:

```
glGenVertexArrays(2, vaoUpdateID);
glGenVertexArrays(2, vaoRenderID);
glGenBuffers( 2, vboID_Pos);
glGenBuffers( 2, vboID_PrePos);
for(int i=0;i<2;i++) {
    glBindVertexArray(vaoUpdateID[i]);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Pos[i]);
    glBufferData( GL_ARRAY_BUFFER, X.size()* sizeof(glm::vec4),
                  &(X[0].x), GL_DYNAMIC_COPY);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_PrePos[i]);
    glBufferData( GL_ARRAY_BUFFER,
                  X_last.size()*sizeof(glm::vec4), &(X_last[0].x),
                  GL_DYNAMIC_COPY);
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,0);
}
//set render vao
for(int i=0;i<2;i++) {
    glBindVertexArray(vaoRenderID[i]);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Pos[i]);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIndices);
    if(i==0)
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                     indices.size()*sizeof(GLushort), &indices[0],
                     GL_STATIC_DRAW);
}
```

3. For ease of access in the vertex shader, bind the current and previous position buffer objects to a set of buffer textures. The buffer textures are one dimensional textures that are created like normal OpenGL textures using the glGenTextures call, but they are bound to the GL_TEXTURE_BUFFER target. They provide read access to the entire buffer object memory in the vertex shader. The data is accessed in the vertex shader using the texelFetchBuffer function:

```
for(int i=0;i<2;i++) {
    glBindTexture( GL_TEXTURE_BUFFER, texPosID[i] );
    glTexBuffer( GL_TEXTURE_BUFFER, GL_RGBA32F, vboID_Pos[i] );
    glBindTexture( GL_TEXTURE_BUFFER, texPrePosID[i] );
    glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, vboID_PrePos[i]);
}
```

4. Generate a transform feedback object and pass the attribute names that will be output from our deformation vertex shader. Make sure to relink the program.

```
glGenTransformFeedbacks(1, &tfID);
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, tfID);
const char* varying_names[]={ "out_position_mass",
    "out_prev_position"};
glTransformFeedbackVaryings(massSpringShader.GetProgram(), 2,
    varying_names, GL_SEPARATE_ATTRIBS);
glLinkProgram(massSpringShader.GetProgram());
```

5. In the rendering function, bind the cloth deformation shader (Chapter8/TransformFeedbackCloth/shaders/Spring.vert) and then run a loop. In each loop iteration, bind the texture buffers, and then bind the update vertex array object. At the same time, bind the previous buffer objects as the transform feedback buffers. These will store the output from the vertex shader. Disable the rasterizer, begin the transform feedback mode, and then draw the entire set of cloth vertices. Use the ping pong approach to swap the read/write pathways:

```
massSpringShader.Use();
glUniformMatrix4fv(massSpringShader("MVP"), 1, GL_FALSE,
glm::value_ptr(mMVP));
for(int i=0;i<NUM_ITER;i++) {
    glBindTexture(GL_TEXTURE0, texPosID[writeID]);
    glBindTexture(GL_TEXTURE1, texPrePosID[writeID]);
    glBindVertexArray(vaoUpdateID[writeID]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0,
        vboID_Pos[readID]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1,
        vboID_PrePos[readID]);
    glEnable(GL_RASTERIZER_DISCARD); // disable rasterization
```

```
    glBeginQuery(GL_TIME_ELAPSED, t_query);
    glBeginTransformFeedback(GL_POINTS);
    glDrawArrays(GL_POINTS, 0, total_points);
    glEndTransformFeedback();
    glEndQuery(GL_TIME_ELAPSED);
    glFlush();
    glDisable(GL_RASTERIZER_DISCARD);
    int tmp = readID;
    readID=writeID;
    writeID = tmp;
}
glGetQueryObjectui64v(t_query, GL_QUERY_RESULT,
    &elapsed_time);
delta_time = elapsed_time / 1000000.0f;
massSpringShader.UnUse();
```

6. After the loop is terminated, bind the render VAO that renders the cloth geometry and vertices:

```
glBindVertexArray(vaoRenderID[writeID]);
glDisable(GL_DEPTH_TEST);
renderShader.Use();
glUniformMatrix4fv(renderShader("MVP"), 1, GL_FALSE,
    glm::value_ptr(mMVP));
glDrawElements(GL_TRIANGLES, indices.size(),
    GL_UNSIGNED_SHORT, 0);
renderShader.UnUse();
 glEnable(GL_DEPTH_TEST);
if(bDisplayMasses) {
    particleShader.Use();
    glUniform1i(particleShader("selected_index"),
        selected_index);
    glUniformMatrix4fv(particleShader("MV"), 1, GL_FALSE,
        glm::value_ptr(mMV));
    glUniformMatrix4fv(particleShader("MVP"), 1, GL_FALSE,
        glm::value_ptr(mMVP));
    glDrawArrays(GL_POINTS, 0, total_points);
    particleShader.UnUse();
}
glBindVertexArray( 0);
```

7. In the vertex shader, obtain the current and previous position of the cloth vertex. If the vertex is a pinned vertex, set its mass to 0 so it would not be simulated; otherwise, add an external force based on gravity. Next loop through all neighbors of the current vertex by looking up the texture buffer and estimate the internal force:

```

float m = position_mass.w;
vec3 pos = position_mass.xyz;
vec3 pos_old = prev_position.xyz;
vec3 vel = (pos - pos_old) / dt;
float ks=0, kd=0;
int index = gl_VertexID;
int ix = index % texsize_x;
int iy = index / texsize_x;
if(index ==0 || index == (texsize_x-1))
    m = 0;
vec3 F = gravity*m + (DEFAULT_DAMPING*vel);
for(int k=0;k<12;k++) {
    ivec2 coord = getNextNeighbor(k, ks, kd);
    int j = coord.x;
    int i = coord.y;
    if (((iy + i) < 0) || ((iy + i) > (texsize_y-1)))
        continue;
    if (((ix + j) < 0) || ((ix + j) > (texsize_x-1)))
        continue;
    int index_neigh = (iy + i) * texsize_x + ix + j;
    vec3 p2 = texelFetchBuffer(tex_position_mass,
    index_neigh).xyz;
    vec3 p2_last = texelFetchBuffer(tex_prev_position_mass,
    index_neigh).xyz;
    vec2 coord_neigh = vec2(ix + j, iy + i)*step;
    float rest_length = length(coord*inv_cloth_size);
    vec3 v2 = (p2- p2_last)/dt;
    vec3 deltaP = pos - p2;
    vec3 deltaV = vel - v2;
    float dist = length(deltaP);
    float leftTerm = -ks * (dist-rest_length);
    float rightTerm = kd * (dot(deltaV, deltaP)/dist);
    vec3 springForce = (leftTerm + rightTerm) *
    normalize(deltaP);
    F += springForce;
}

```

8. Using the combined force, calculate the acceleration and then estimate the new position using Verlet integration. Output the appropriate attribute from the shader:

```
vec3 acc = vec3(0);
if(m!=0)
    acc = F/m;
vec3 tmp = pos;
pos = pos * 2.0 - pos_old + acc* dt * dt;
pos_old = tmp;
pos.y=max(0, pos.y);
out_position_mass = vec4(pos, m);
out_prev_position = vec4(pos_old,m);
gl_Position = MVP*vec4(pos, 1);
```

How it works...

There are two parts of this recipe, the generation of geometry and identifying output attributes for transform feedback buffers. We first generate the cloth geometry and then associate our buffer objects. To enable easier access of current and previous positions, we bind the position buffer objects as texture buffers.

To enable deformation, we first bind our deformation shader and the update VAO. Next, we specify the transform feedback buffers that receive the output from the vertex shader. We disable the rasterizer to prevent the execution of the rest of the pipeline. Next, we begin the transform feedback mode, render our vertices, and then end the transform feedback mode. This invokes one step of the integration. To enable more steps, we use a ping pong strategy by binding the currently written buffer object as the read point for the next iteration.

The actual deformation is carried out in the vertex shader ([Chapter8/TransformFeedbackCloth/shaders/Spring.vert](#)). We first determine the current and previous positions. The velocity is then determined. The current vertex ID (`gl_VertexID`) is used to determine the linear index of the current vertex. This is a unique index of each vertex and can be used by a vertex shader. We use it here to determine if the current vertex is a pinned vertex. If so, the mass of 0 is assigned to it which makes this vertex immovable:

```
float m = position_mass.w;
vec3 pos = position_mass.xyz;
vec3 pos_old = prev_position.xyz;
vec3 vel = (pos - pos_old) / dt;
float ks=0, kd=0;
int index = gl_VertexID;
int ix = index % texsize_x;
int iy = index / texsize_x;
if(index ==0 || index == (texsize_x-1))
    m = 0;
```

Next, the acceleration due to gravity and velocity damping force is applied. After this, a loop is run which basically loops through all of the neighbors of the current vertex and estimates the net internal (spring) force. This force is then added to the combined force for the current vertex:

```

vec3 F = gravity*m + (DEFAULT_DAMPING*vel);

for(int k=0;k<12;k++) {
    ivec2 coord = getNextNeighbor(k, ks, kd);
    int j = coord.x;
    int i = coord.y;
    if (((iy + i) < 0) || ((iy + i) > (texsize_y-1)))
        continue;
    if (((ix + j) < 0) || ((ix + j) > (texsize_x-1)))
        continue;
    int index_neigh = (iy + i) * texsize_x + ix + j;
    vec3 p2 = texelFetchBuffer(tex_position_mass,
    index_neigh).xyz;
    vec3 p2_last = texelFetchBuffer(tex_prev_position_mass,
    index_neigh).xyz;
    vec2 coord_neigh = vec2(ix + j, iy + i)*step;
    float rest_length = length(coord*inv_cloth_size);
    vec3 v2 = (p2- p2_last)/dt;
    vec3 deltaP = pos - p2;
    vec3 deltaV = vel - v2;
    float dist = length(deltaP);
    float leftTerm = -ks * (dist-rest_length);
    float rightTerm = kd * (dot(deltaV, deltaP)/dist);
    vec3 springForce = (leftTerm + rightTerm)* normalize(deltaP);
    F += springForce;
}

```

From the net force, the acceleration is first obtained and then the new position is obtained using Verlet integration. Finally, the collision with the ground plane is determined by looking at the Y value. We end the shader by outputting the output attributes (`out_position` and `out_prev_position`), which are then stored into the buffer objects bound as the transform feedback buffers:

```

vec3 acc = vec3(0);
if(m!=0)
    acc = F/m;
vec3 tmp = pos;
pos = pos * 2.0 - pos_old + acc* dt * dt;
pos_old = tmp;
pos.y=max(0, pos.y);

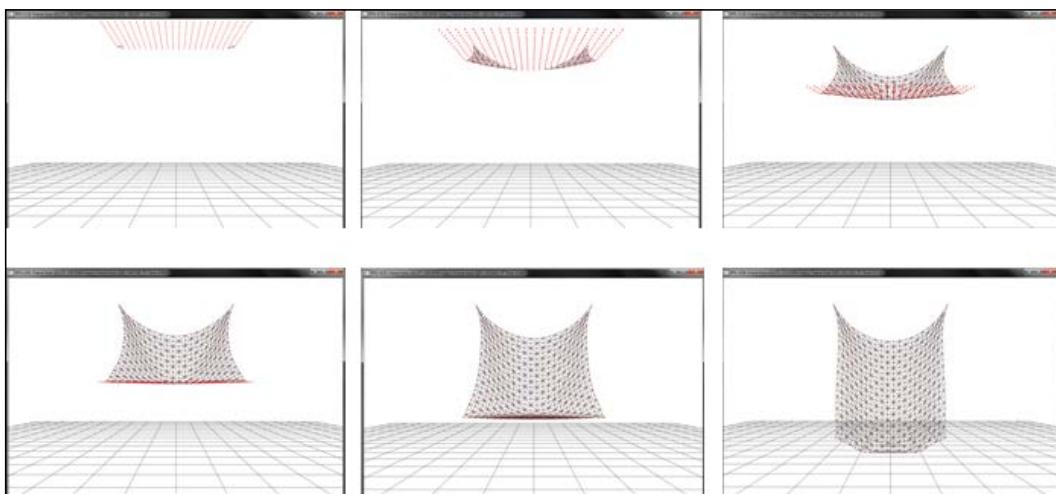
```

```
out_position_mass = vec4(pos, m);
out_prev_position = vec4(pos_old,m);
gl_Position = MVP*vec4(pos, 1);
```

The shader, along with the transform feedback mechanism, proceeds to deform all of the cloth vertices and in the end, we get the cloth vertices deformed.

There's more...

The demo application implementing this recipe shows the piece of cloth falling under gravity. Several frames from the deformation are shown in the following figure. Using the left mouse button, we can pick the cloth vertices and move them around.



In this recipe we only output to a single stream. We can attach more than one stream and store results in separate buffer objects. In addition, we can have several transform feedback objects and we can pause/resume them as required.

See also

- ▶ *Chapter 17, Real-Time Physically Based Deformation Using Transform Feedback, in OpenGL Insights, AK Peters CRC press*

Implementing collision detection and response on a transform feedback-based cloth model

In this recipe, we will build on top of the previous recipe and add collision detection and response to the cloth model.

Getting ready

The code for this recipe is contained in the `Chapter8/TransformFeedbackClothCollision` directory. For this recipe, the setup code and rendering code remains the same as in the previous recipe. The only change is the addition of the ellipsoid/sphere collision code.

How to do it...

Let us start this recipe by following these simple steps:

1. Generate the geometry and topology for a piece of cloth by creating a set of points and their connectivity. Bind this data to a buffer object as in the previous recipe.
2. Set up a pair of vertex array objects and buffer objects as in the previous recipe. Also attach buffer textures for easier access to the buffer object memory in the vertex shader.
3. Generate a transform feedback object and pass the attribute names that will be output from our deformation vertex shader. Make sure to relink the program again:

```
glGenTransformFeedbacks(1, &tfID);
 glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, tfID);
 const char* varying_names[] = {"out_position_mass", "out_prev_
position"};
 glTransformFeedbackVaryings(massSpringShader.GetProgram(), 2,
varying_names, GL_SEPARATE_ATTRIBS);
 glLinkProgram(massSpringShader.GetProgram());
```

4. Generate an ellipsoid object by using a simple 4×4 matrix. Also store the inverse of the ellipsoid's transform. The location of the ellipsoid is stored by the translate matrix, the orientation by the rotate matrix, and the non-uniform scaling by the scale matrix as follows. When applied, the matrices work in the opposite order. The non-uniform scaling causes the sphere to compress in the Z direction first. Then, the rotation orients the ellipsoid such that it is rotated by 45 degrees on the X axis. Finally, the ellipsoid is shifted by 2 units on the Y axis:

```
ellipsoid = glm::translate(glm::mat4(1), glm::vec3(0, 2, 0));
ellipsoid = glm::rotate(ellipsoid, 45.0f, glm::vec3(1, 0, 0));
```

```
ellipsoid = glm::scale(ellipsoid,
    glm::vec3(fRadius,fRadius,fRadius/2));
inverse_ellipsoid = glm::inverse(ellipsoid);
```

5. In the rendering function, bind the cloth deformation shader (`Chapter8/TransformFeedbackClothCollision/shaders/Spring.vert`) and then run a loop. In each iteration, bind the texture buffers, and then bind the update vertex array object. At the same time, bind the previous buffer objects as the transform feedback buffers. Do the ping pong strategy as in the previous recipe.
6. After the loop is terminated, bind the render VAO and render the cloth:

```
glBindVertexArray(vaoRenderID[writeID]);
glDisable(GL_DEPTH_TEST);
renderShader.Use();
glUniformMatrix4fv(renderShader("MVP"), 1, GL_FALSE,
glm::value_ptr(mMVP));
glDrawElements(GL_TRIANGLES, indices.size(),
GL_UNSIGNED_SHORT, 0);
renderShader.UnUse();
 glEnable(GL_DEPTH_TEST);
if(bDisplayMasses) {
    particleShader.Use();
    glUniform1i(particleShader("selected_index"),
selected_index);
    glUniformMatrix4fv(particleShader("MV"), 1, GL_FALSE,
glm::value_ptr(mMV));
    glUniformMatrix4fv(particleShader("MVP"), 1, GL_FALSE,
glm::value_ptr(mMVP));
    glDrawArrays(GL_POINTS, 0, total_points);
    particleShader.UnUse(); }
glBindVertexArray( 0 );
```

7. In the vertex shader, obtain the current and previous position of the cloth vertex. If the vertex is a pinned vertex, set its mass to 0 so it would not be simulated, otherwise, add an external force based on gravity. Next, loop through all neighbors of the current vertex by looking up the texture buffer and estimate the internal force:

```
float m = position_mass.w;
vec3 pos = position_mass.xyz;
vec3 pos_old = prev_position.xyz;
vec3 vel = (pos - pos_old) / dt;
float ks=0, kd=0;
int index = gl_VertexID;
int ix = index % texsize_x;
int iy = index / texsize_x;
```

```

if(index ==0 || index == (texsize_x-1))
    m = 0;
vec3 F = gravity*m + (DEFAULT_DAMPING*vel);
for(int k=0;k<12;k++) {
    ivec2 coord = getNextNeighbor(k, ks, kd);
    int j = coord.x;
    int i = coord.y;
    if (((iy + i) < 0) || ((iy + i) > (texsize_y-1)))
        continue;
    if (((ix + j) < 0) || ((ix + j) > (texsize_x-1)))
        continue;
    int index_neigh = (iy + i) * texsize_x + ix + j;
    vec3 p2 = texelFetchBuffer(tex_position_mass,
    index_neigh).xyz;
    vec3 p2_last = texelFetchBuffer(tex_prev_position_mass,
    index_neigh).xyz;
    vec2 coord_neigh = vec2(ix + j, iy + i)*step;
    float rest_length = length(coord*inv_cloth_size);
    vec3 v2 = (p2- p2_last)/dt;
    vec3 deltaP = pos - p2;
    vec3 deltaV = vel - v2;
    float dist = length(deltaP);
    float leftTerm = -ks * (dist-rest_length);
    float rightTerm = kd * (dot(deltaV, deltaP)/dist);
    vec3 springForce = (leftTerm + rightTerm)*
    normalize(deltaP);
    F += springForce;
}

```

8. Using the combined force, calculate the acceleration and then estimate the new position using Verlet integration. Output the appropriate attribute from the shader:

```

vec3 acc = vec3(0);
if(m!=0)
    acc = F/m;
vec3 tmp = pos;
pos = pos * 2.0 - pos_old + acc* dt * dt;
pos_old = tmp;
pos.y=max(0, pos.y);

```

9. After applying the floor collision, check for collision with an ellipsoid. If there is a collision, modify the position such that the collision is resolved. Finally, output the appropriate attributes from the vertex shader.

```
vec4 x0 = inv_ellipsoid*vec4(pos, 1);
vec3 delta0 = x0.xyz-ellipsoid.xyz;
float dist2 = dot(delta0, delta0);
if(dist2<1) {
    delta0 = (ellipsoid.w - dist2) * delta0 / dist2;
    vec3 delta;
    vec3 transformInv = vec3(ellipsoid_xform[0].x,
    ellipsoid_xform[1].x,
    ellipsoid_xform[2].x);
    transformInv /= dot(transformInv, transformInv);
    delta.x = dot(delta0, transformInv);
    transformInv = vec3(ellipsoid_xform[0].y,
    ellipsoid_xform[1].y,
    ellipsoid_xform[2].y);
    transformInv /= dot(transformInv, transformInv);
    delta.y = dot(delta0, transformInv);
    transformInv = vec3(ellipsoid_xform[0].z,
    ellipsoid_xform[1].z,
    ellipsoid_xform[2].z);
    transformInv /= dot(transformInv, transformInv);
    delta.z = dot(delta0, transformInv);
    pos += delta ;
    pos_old = pos;
}
out_position_mass = vec4(pos, m);
out_prev_position = vec4(pos_old,m);
gl_Position = MVP*vec4(pos, 1);
```

How it works...

The cloth deformation vertex shader has some additional lines of code to enable collision detection and response. For detection of collision with a plane, we can simply put the current position in the plane equation to find the distance of the current vertex from the plane. If it is less than 0, we have passed through the plane, in which case, we can move the vertex back in the plane's normal direction.

```
void planeCollision(inout vec3 x,  vec4 plane) {
    float dist = dot(plane.xyz,x)+ plane.w;
    if(dist<0) {
        x += plane.xyz*-dist;
    }
}
```

Simple geometric primitive, like spheres and ellipsoids, are trivial to handle. In case of collision with the sphere, we check the distance of the current position from the center of the sphere. If this distance is less than the sphere's radius, we have a collision. Once we have a collision, we push the position in the normal direction based on the amount of penetration.

```
void sphereCollision(inout vec3 x, vec4 sphere)
{
    vec3 delta = x - sphere.xyz;
    float dist = length(delta);
    if (dist < sphere.w) {
        x = sphere.xyz + delta*(sphere.w / dist);
    }
}
```



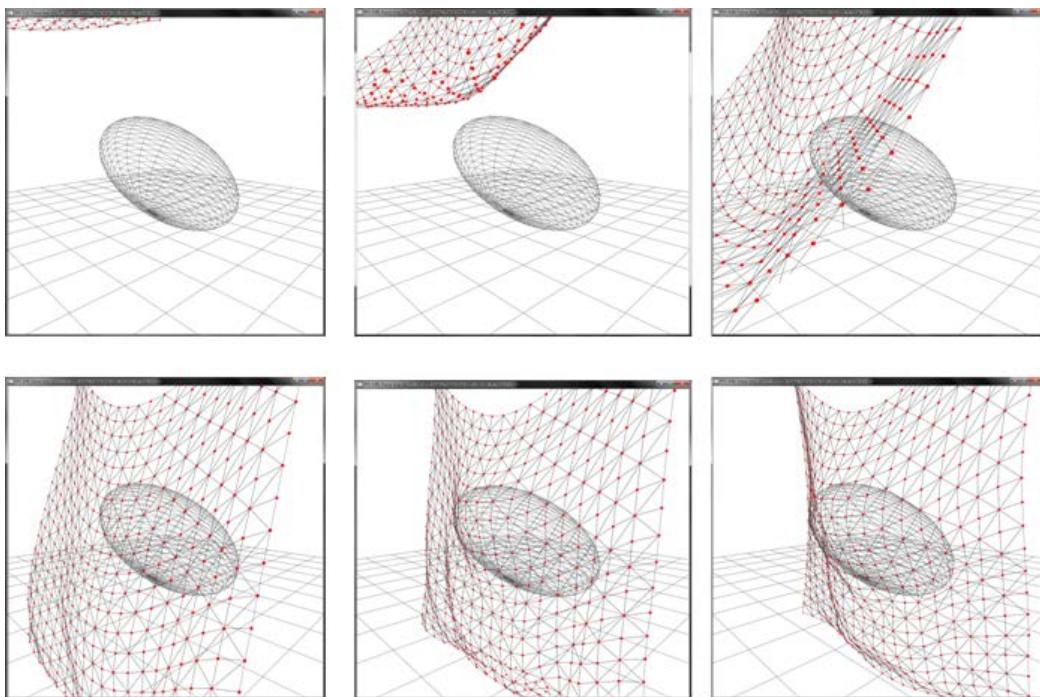
Note that in the preceding calculation, we can avoid the square root altogether by comparing against the squared distance. This can provide significant performance gain when a large number of vertices are there.

For an arbitrarily oriented ellipsoid, we first move the point into the ellipsoid's object space by multiplying with the inverse of the ellipsoid's transform. In this space, the ellipsoid is a unit sphere, hence we can then determine collision by simply looking at the distance between the current vertex and the ellipsoid. If it is less than 1, we have a collision. In this case, we then transform the point to the ellipsoid's world space to find the penetration depth. This is then used to displace the current position out in the normal direction.

```
vec4 x0 = inv_ellipsoid*vec4(pos,1);
vec3 delta0 = x0.xyz-ellipsoid.xyz;
float dist2 = dot(delta0, delta0);
if(dist2<1) {
    delta0 = (ellipsoid.w - dist2) * delta0 / dist2;
    vec3 delta;
    vec3 transformInv = vec3(ellipsoid_xform[0].x, ellipsoid_xform[1].x,
    ellipsoid_xform[2].x);
    transformInv /= dot(transformInv, transformInv);
    delta.x = dot(delta0, transformInv);
    transformInv = vec3(ellipsoid_xform[0].y, ellipsoid_xform[1].y,
    ellipsoid_xform[2].y);
    transformInv /= dot(transformInv, transformInv);
    delta.y = dot(delta0, transformInv);
    transformInv = vec3(ellipsoid_xform[0].z, ellipsoid_xform[1].z,
    ellipsoid_xform[2].z);
    transformInv /= dot(transformInv, transformInv);
    delta.z = dot(delta0, transformInv);
    pos += delta ;
    pos_old = pos;
}
```

There's more...

The demo application implementing this recipe renders a piece of cloth fixed at two points and is allowed to fall under gravity. In addition, there is an oriented ellipsoid with which the cloth collides as shown in the following figure:



Although we have touched upon basic collision primitives, like spheres, oriented ellipsoids, and plane, more complex primitives can be implemented with the combination of these basic primitives. In addition, polygonal primitives can also be implemented. We leave that as an exercise for the reader.

See also

- ▶ MOVANIA Muhammad Mobeen and Lin Feng, "A Novel GPU-based Deformation Pipeline" in ISRN Computer Graphics, Volume 2012(2012), Article ID 936315, available online at <http://downloads.hindawi.com/isrn/cg/2012/936315.pdf>

Implementing a particle system using transform feedback

In this recipe, we will implement a simple particle system using the transform feedback mechanism. In this mode, the GPU bypasses the rasterizer and, later, the programmable graphics pipeline stages to feedback result to the vertex shader. The benefit from this mode is that using this feature, we can implement a physically-based simulation entirely on the GPU.

Getting ready

The code for this recipe is contained in the Chapter8/TransformFeedbackParticles directory.

How to do it...

Let us start this recipe by following these simple steps:

1. Set up two vertex array pairs: one for update and another for rendering. Bind two vertex buffer objects to each of the pairs, as was done in the previous two recipes. Here, the buffer objects will store the per-particle properties. Also, enable the corresponding vertex attributes:

```
glGenVertexArrays(2, vaoUpdateID);
glGenVertexArrays(2, vaoRenderID);
glGenBuffers( 2, vboID_Pos);
glGenBuffers( 2, vboID_PrePos);
glGenBuffers( 2, vboID_Direction);
for(int i=0;i<2;i++) {
    glBindVertexArray(vaoUpdateID[i]);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Pos[i] );
    glBufferData( GL_ARRAY_BUFFER, TOTAL_PARTICLES*
        sizeof(glm::vec4), 0, GL_DYNAMIC_COPY );
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_PrePos[i] );
    glBufferData( GL_ARRAY_BUFFER, TOTAL_PARTICLES*
        sizeof(glm::vec4), 0, GL_DYNAMIC_COPY );
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0,0);
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Direction[i] );
    glBufferData( GL_ARRAY_BUFFER, TOTAL_PARTICLES*
        sizeof(glm::vec4), 0, GL_DYNAMIC_COPY );
    glEnableVertexAttribArray(2);
```

```
    glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0, 0);
}
for(int i=0;i<2;i++) {
    glBindVertexArray(vaoRenderID[i]);
    glBindBuffer(GL_ARRAY_BUFFER, vboID_Pos[i]);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
}
```

2. Generate a transform feedback object and bind it. Next, specify the output attributes from the shader that would be stored in the transform feedback buffer. After this step, relink the shader program:

```
glGenTransformFeedbacks(1, &tfID);
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, tfID);
const char* varying_names []={"out_position",
"out_prev_position", "out_direction"};
glTransformFeedbackVaryings(particleShader.GetProgram(), 3,
varying_names, GL_SEPARATE_ATTRIBS);
glLinkProgram(particleShader.GetProgram());
```

3. In the update function, bind the particle vertex shader that will output to the transform feedback buffer and set the appropriate uniforms and update vertex array object. Note that to enable read/write access, we use a pair of vertex array objects such that we can read from one and write to another:

```
particleShader.Use();
glUniformMatrix4fv(particleShader("MVP"), 1, GL_FALSE,
glm::value_ptr(mMVP));
glUniform1f(particleShader("time"), t);
for(int i=0;i<NUM_ITER;i++) {
    glBindVertexArray( vaoUpdateID[readID]);
```

4. Bind the vertex buffer objects that will store the outputs from the transform feedback step using the output attributes from the vertex shader:

```
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vboID_Pos[writeID]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, vboID_PrePos[writeID]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, vboID_Direction[writeID]);
```

5. Disable the rasterizer to prevent the execution of the later stages of the pipeline and then begin the transform feedback. Next, issue a call to the `glDrawArrays` function to allow the vertices to be passed to the graphics pipeline. After this step, end the transform feedback and then enable the rasterizer. Note that to correctly determine the amount of execution time needed, we issue a hardware query. Next, we alternate the read/write paths by swapping the read and write IDs:

```
glEnable(GL_RASTERIZER_DISCARD);
glBeginQuery(GL_TIME_ELAPSED,t_query);
glBeginTransformFeedback(GL_POINTS);
glDrawArrays(GL_POINTS, 0, TOTAL_PARTICLES);
glEndTransformFeedback();
glEndQuery(GL_TIME_ELAPSED);
glFlush();
glDisable(GL_RASTERIZER_DISCARD);
int tmp = readID;
readID=writeID;
writeID = tmp;
```

6. Render the particles using the render shader. First, bind the render vertex array object and then draw the points using the `glDrawArrays` function:

```
glBindVertexArray(vaoRenderID[readID]);
renderShader.Use();
glm::value_ptr(mMVP);
glDrawArrays(GL_POINTS, 0, TOTAL_PARTICLES);
renderShader.UnUse();
glBindVertexArray(0);
```

7. In the particle vertex shader, check if the particle's life is greater than 0. If so, move the particle and reduce the life. Otherwise, calculate a new random direction and reset the life to spawn the particle from the origin. Refer to [Chapter8/TransformFeedbackParticles/shaders/Particle.vert](#) for details. After this step, output the appropriate values to the output attributes. The particle vertex shader is defined as follows:

```
#version 330 core
precision highp float;
#extension EXT_gpu_shader4 : require
layout( location = 0 ) in vec4 position;
layout( location = 1 ) in vec4 prev_position;
layout( location = 2 ) in vec4 direction;
uniform mat4 MVP;
uniform float time;
const float PI = 3.14159;
```

```
const float TWO_PI = 2*PI;
const float PI_BY_2 = PI*0.5;
const float PI_BY_4 = PI_BY_2*0.5;

//shader outputs
out vec4 out_position;
out vec4 out_prev_position;
out vec4 out_direction;

const float DAMPING_COEFFICIENT = 0.9995;
const vec3 emitterForce = vec3(0.0f,-0.001f, 0.0f);
const vec4 collidor = vec4(0,1,0,0);
const vec3 emitterPos = vec3(0);

float emitterYaw = (0.0f);
float emitterYawVar = TWO_PI;
float emitterPitch = PI_BY_2;
float emitterPitchVar = PI_BY_4;
float emitterSpeed = 0.05f;
float emitterSpeedVar = 0.01f;

int emitterLife = 60;
int emitterLifeVar = 15;

const float UINT_MAX = 4294967295.0;

void main() {
    vec3 prevPos = prev_position.xyz;
    int life = int(prev_position.w);
    vec3 pos = position.xyz;
    float speed = position.w;
    vec3 dir = direction.xyz;
    if(life > 0) {
        prevPos = pos;
        pos += dir*speed;
        if(dot(pos+emitterPos, collidor.xyz)+ collidor.w <0) {
            dir = reflect(dir, collidor.xyz);
            speed *= DAMPING_COEFFICIENT;
        }
        dir += emitterForce;
        life--;
    } else {
        uint seed = uint(time + gl_VertexID);
```

```
    life = emitterLife + int(randhashf(seed++,
emitterLifeVar));
    float yaw = emitterYaw + (randhashf(seed++,
emitterYawVar));
    float pitch = emitterPitch + randhashf(seed++,
emitterPitchVar);
    RotationToDirection(pitch, yaw, dir);
    float nspeed = emitterSpeed + (randhashf(seed++,
emitterSpeedVar));
    dir *= nspeed;
    pos = emitterPos;
    prevPos = emitterPos;
    speed = 1;
}
out_position = vec4(pos, speed);
out_prev_position = vec4(prevPos, life);
out_direction = vec4(dir, 0);
gl_Position = MVP*vec4(pos, 1);
}
```

The three helper functions `randhash`, `randhashf`, and `RotationToDirection` are defined as follows:

```
uint randhash(uint seed) {
    uint i=(seed^12345391u)*2654435769u;
    i^=(i<<6u)^ (i>>26u);
    i*=2654435769u;
    i+=(i<<5u)^ (i>>12u);
    return i;
}

float randhashf(uint seed, float b) {
    return float(b * randhash(seed)) / UINT_MAX;
}

void RotationToDirection(float pitch, float yaw,
                           out vec3 direction) {
    direction.x = -sin(yaw) * cos(pitch);
    direction.y = sin(pitch);
    direction.z = cos(pitch) * cos(yaw);
}
```

How it works...

The transform feedback mechanism allows us to feedback one or more attributes from the vertex shader or geometry shader back to a buffer object. This feedback path could be used for implementing a physically-based simulation. This recipe uses this mechanism to output the particle position after each iteration. After each step, the buffers are swapped and, therefore, it can simulate the particle motion.

To make the particle system, we first set up three pairs of vertex buffer objects that store the per-particle attributes that we input to the vertex shader. These include the particle's position, previous position, life, direction, and speed. These are stored into separate buffer objects for convenience. We could have stored all of these attributes into a single interleaved buffer object. Since we output to the buffer object from our shader, we specify the buffer object usage as `GL_DYNAMIC_COPY`. Similarly, we set up a separate vertex array object for rendering the particles:

```
for(int i=0;i<2;i++) {  
    glBindVertexArray(vaoUpdateID[i]);  
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Pos[i]);  
    glBufferData( GL_ARRAY_BUFFER, TOTAL_PARTICLES*  
        sizeof(glm::vec4), 0, GL_DYNAMIC_COPY);  
    glEnableVertexAttribArray(0);  
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);  
    glBindBuffer( GL_ARRAY_BUFFER, vboID_PrePos[i]);  
    glBufferData( GL_ARRAY_BUFFER,  
        TOTAL_PARTICLES*sizeof(glm::vec4), 0, GL_DYNAMIC_COPY);  
    glEnableVertexAttribArray(1);  
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 0, 0);  
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Direction[i]);  
    glBufferData( GL_ARRAY_BUFFER,  
        TOTAL_PARTICLES*sizeof(glm::vec4), 0, GL_DYNAMIC_COPY);  
    glEnableVertexAttribArray(2);  
    glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, 0, 0);  
}  
for(int i=0;i<2;i++) {  
    glBindVertexArray(vaoRenderID[i]);  
    glBindBuffer( GL_ARRAY_BUFFER, vboID_Pos[i]);  
    glEnableVertexAttribArray(0);  
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);  
}
```

Next, we specify the shader output attributes that we would like to connect to the transform feedback buffers. We use three outputs, namely `out_position`, `out_prev_position`, and `out_direction`, which output the particle's current position, particle's previous position, and the particle's direction along with the particle's speed, current, and initial life, respectively. We specify that we would connect these to separate buffer objects:

```
glGenTransformFeedbacks(1, &tfID);
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, tfID);
const char* varying_names [] = {"out_position", "out_prev_position",
"out_direction"};
glTransformFeedbackVaryings(particleShader.GetProgram(), 3, varying_
names, GL_SEPARATE_ATTRIBS);
glLinkProgram(particleShader.GetProgram());
```

One last step is the actual initialization of the transform feedback. We do so by first binding the particle vertex shader. Then, we pass the appropriate uniforms to the shader, which includes the combined modelview projection (MVP) matrix and the time (`t`):

```
particleShader.Use();
glUniformMatrix4fv(particleShader("MVP"), 1, GL_FALSE,
glm::value_ptr(mMVP));
glUniform1f(particleShader("time"), t);
```

We then run a loop for the number of iterations desired. In the loop, we first bind the update vertex array object and assign the appropriate transform feedback buffer base indices:

```
for(int i=0;i<NUM_ITER;i++) {
    glBindVertexArray(vaoUpdateID[readID]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0,
vboID_Pos[writeID]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1,
vboID_PrePos[writeID]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2,
vboID_Direction[writeID]);
```

We then disable the rasterizer and begin the transform feedback. During this, we issue a `glDrawArrays` call to pass the vertices to the vertex shader. Next, we end the transform feedback and then restore the rasterizer. Finally, we swap the read/write pathways. In between, to estimate the amount of time needed for this process, we issue an OpenGL hardware timer query (`GL_TIME_ELAPSED`). This returns the total time in nanoseconds:

```
 glEnable(GL_RASTERIZER_DISCARD); // disable rasterization
 glBeginQuery(GL_TIME_ELAPSED, t_query);
 glBeginTransformFeedback(GL_POINTS);
 glDrawArrays(GL_POINTS, 0, TOTAL_PARTICLES);
 glEndTransformFeedback();
 glEndQuery(GL_TIME_ELAPSED);
```

```
glFlush();
glDisable(GL_RASTERIZER_DISCARD);
int tmp = readID;
readID=writeID;
writeID = tmp;
}
// get the query result
glGetQueryObjectui64v(t_query, GL_QUERY_RESULT, &elapsed_time);
delta_time = elapsed_time / 1000000.0f;
particleShader.UnUse();
```

The main work of particle simulation takes place in the vertex shader (Chapter8/TransformFeedbackParticles/shaders/Particle.vert). After storing the initial attributes, the particle's life is checked. If the value is greater than 0, we update the position of the particle using the current direction of the particle and its speed. Next, we then check the particle for collision with the colliding plane. If there is a collision, we deflect the particle using the reflect GLSL function, passing it the particle's current direction of motion and the normal of the corridor. We also reduce the speed on collision. We then increase the direction of the particle using the emitter's force. We then reduce the life of the particle:

```
if(life > 0) {
    prevPos = pos;
    pos += dir*speed;
    if(dot(pos+emitterPos, corridor.xyz)+ corridor.w <0) {
        dir = reflect(dir, corridor.xyz);
        speed *= DAMPING_COEFFICIENT;
    }
    dir += emitterForce;
    life--;
}
```

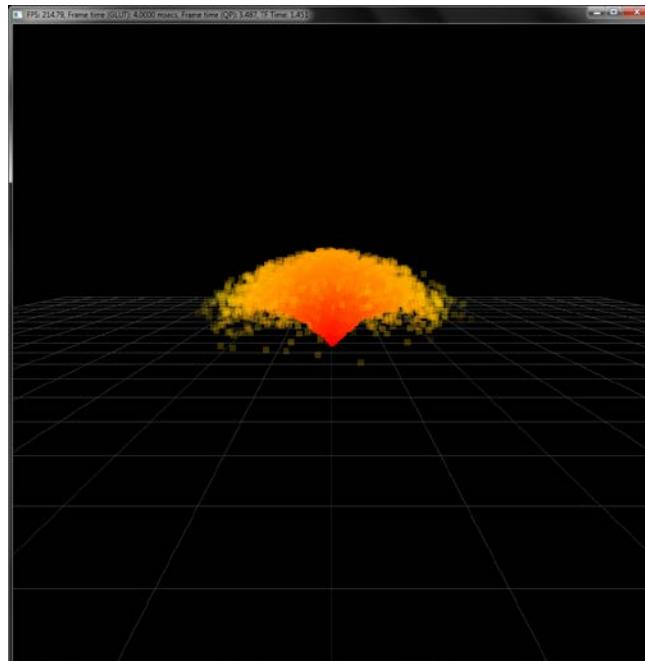
If the life is less than 0, we reset the particle's direction of motion to a new random direction. We reset the life to a random value based on the maximum allowed value. The current and previous positions of the particle are reset to the emitter origin and finally, the speed is reset to the default value. We then output the output attributes:

```
else {
    uint seed = uint(time + gl_VertexID);
    life = emitterLife + int(randhashf(seed++, emitterLifeVar));
    float yaw = emitterYaw + (randhashf(seed++, emitterYawVar));
    float pitch=emitterPitch+randhashf(seed++, emitterPitchVar);
    RotationToDirection(pitch, yaw, dir);
    float nspeed = emitterSpeed + (randhashf(seed++,
    emitterSpeedVar));
    dir *= nspeed;
```

```
pos = emitterPos;
prevPos = emitterPos;
speed = 1;
}
out_position = vec4(pos, speed);
out_prev_position = vec4(prevPos, life);
out_direction = vec4(dir, 0);
gl_Position = MVP*vec4(pos, 1);
```

There's more...

The demo application for this recipe generates a simple particle system running entirely on the GPU using the transform feedback mechanism coupled with a vertex shader that writes to output attributes bound as transform feedback buffers. Running the demo application gives us the output as shown in the following figure:



Note that for this demo, we render the particles as points of size 10 units. We could easily change the rendering mode to point sprites with size modified in the vertex shader to give the particles a different look. Also, we can also change the colors and blending modes to achieve various effects. In addition, we could achieve the same result by using one vertex buffer pair with interleaved attributes or two separate transform feedback objects. All of these variants should be straightforward to implement by following the guidelines laid out in this recipe.

We had already looked at a simple approach of simulating GPU-based particle systems using vertex shader in *Chapter 5, Mesh Model Formats and Particle Systems*, we will now detail pros and cons of each. In *Chapter 5* we presented a stateless particle system since all of the attributes (that is, position and velocity) were generated on the fly using the vertex ID, time, and basic kinematic equations on each particle vertex.

As the state of the particle is not stored, we cannot reproduce the same simulation every frame. Hence, collision detection and response are problematic, as we do not have any information of the previous state of the particle, which is often required for collision response. On the contrary, the particle simulation technique presented in this recipe uses a state-preserving particle system. We stored current and previous positions of each particle in buffer objects. In addition, we used transform feedback and a vertex shader for particle simulation on the GPU. As the state of the particle is stored, we can carry out collision detection and response easily.

See also

- ▶ OGLDev Tutorial on particle system using transform feedback at <http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html>
- ▶ *OpenGL 4.0 Shading Language Cookbook, Chapter 9, Animation and Particles*, the *Creating a particle system using transform feedback* section, Packt Publishing, 2011.
- ▶ *Noise based Particles, Part II* at *The Little Grasshopper*, <http://prideout.net/blog/?p=67>

Index

Symbols

2D image

 drawing, fragment shader used 48-54
 drawing, SOIL image loading library used
 48-54

3D graphics programming

 URL 27, 114

3ds file format

 URL 156

3ds file loader

 URL 156

3ds Max 141

3DS model loading

 implementing, separate buffers used
 146-155

3D texture slicing

 about 220
 used, for implementing volume rendering
 220-227

3 x 3 Gaussian blur effect 100

A

absolute transform. *See* **global transform**

area filtering

 implementing, on image 98-100

attenuation

 per-fragment point light, implementing with
 117, 118

Autodesk® 3ds 141

Autodesk® FBX (.fbx) 163

Axially Aligned Bounding Box (AABB) 77

B

bi-directional reflectance distribution function (BRDF) 202

Bisection function 235

blend weights 262

Blinn Phong lighting model 111

blurring effect 100

C

C3dsLoader class 146

C3dsLoader::Load3DS function 146, 152

CAbstractCamera class 56

CFreeCamera class 56

cloth

 modeling, transform feedback used 279-289

COLLADA 262

Collada (.dae) 163

collision detection

 implementing 290-295

color

 used, for implementing object picking 74, 75

colored triangle

 rendering, shaders used 19-27

convolution

 area filtering, applying on image 98-100

CreateTexture function 255

CTargetCamera class 56

D

depth buffer

 used, for implementing object picking 72, 73

- dual depth peeling**
 about 189
 used, for implementing order independent transparency 189-193
- dual quaternion skinning**
 used, for implementing skeletal animation 273-279
- dynamic cube mapping**
 used, for rendering reflective object 93-96
- E**
- emboss north-east direction effect** 100
emboss north-west direction effect 100
emboss south-east direction effect 101
emboss south-west direction effect 101
EmitVertex() function 42
EndPrimitive() function 42
EZMesh (.ezm) 163
EZMesh format 262
EZMesh model
 implementing 163-169
EzmLoader::Load function 263
- F**
- FBO**
 shadow mapping, implementing with 122-127
 used, for implementing mirror object 89-92
- FBX** 262
- final bone matrix.** *See* **skinning matrix**
- FPS style input support**
 used, for implementing vector-based camera 56-59
- fragment shader**
 about 16
 used, for drawing 2D image 48-54
 used, for implementing twirl filter 82-85
- Framebuffer objects (FBOs)** 81
- frame-by-frame animations** 261
- free camera**
 about 59
 implementing 60-62
- freeglut libraries**
 OpenGL v3.3 core profile, setting up on Visual Studio 2010 8-14
 URL, for downloading 8
- front-to-back peeling**
 used, for implementing order independent transparency 182-188
- G**
- geometry shader**
 about 16
 plane, subdividing with instanced rendering 45-48
 URL, for tutorial 44
 used, for subdividing plane 37-44
- glBindBufferBase function** 280
glBufferData function 149
glClearDepth function 15
glDepthFunc function 15
glDrawElements function 159
GLEW
 OpenGL v3.3 core profile, setting up on Visual Studio 2010 8-14
- GLEW library**
 URL, for downloading 8
- glFramebufferRenderbuffer function** 89
glFramebufferTexture2D function 89
glGetTransformFeedbacks function 279
glMapBuffer function 149
glm library
 URL, for downloading 9
- global illumination**
 implementing, spherical harmonics lighting used 202-206
- global illumination methods** 181
- global transform** 266
- glow effect**
 implementing 101-105
- GLSL shader class**
 designing 16-19
- GLSLShader::CreateAndLinkProgram function** 23
- GLSLShader::operator() function** 27
- glTexImage3D function** 226
- glTransformFeedbackVaryings function** 280
- glutCloseFunc method** 14
glutMainLoop() function 14
glutSwapBuffer function 27
glutSwapBuffers function 15
glVertexAttribIPointer function 270

Gouraud shading [108](#)

GPU-based path tracing

implementing [213-217](#)

GPU-based ray tracing

implementing [207-211](#)

H

half angle slicing

used, for implementing volumetric lighting

[254-259](#)

height map

used, for implementing terrains [142-145](#)

I

image

area filtering, applying on [98-100](#)

interleaved buffers

used, for implementing Wavefront ® Obj model [157-161](#)

intersectBox function [78](#)

L

level of detail (LOD) [222](#)

Lib3ds

[URL 156](#)

Lift function [62](#)

Lighthouse 3D view frustum culling

[URL, for tutorial 72](#)

linear blend skinning (LBS) [262](#)

local transform [266](#)

M

main() function [12](#)

Marching Cubes [253](#)

Marching Tetrahedra (MT) algorithm

about [248](#)

used, for implementing polygonal isosurface extraction [248-253](#)

matrix palette skinning

about [262](#)

used, for implementing skeletal animation

[262-271](#)

Maya [141](#)

MeshImport library

[URL 170](#)

MeshSystem::mSkeletons array [263](#)

mirror, with render-to-texture

implementing, FBO used [89-92](#)

modelview projection (MVP) [143](#)

Move function [65](#)

N

NVIDIA PhysX sdk [262](#)

O

object picking

implementing, color used [74, 75](#)

implementing, depth buffer used [72, 73](#)

implementing, scene intersection queries used [76-78](#)

OBJ file specification

[URL 162](#)

ObjLoader::Load function [157](#)

ObjLoader object [157](#)

offscreen rendering functionality [81](#)

OnInit() function [12](#)

OnMouseMove function [36](#)

OnRender() function [12, 15](#)

OnResize() function [12](#)

OnShutdown() function [12, 26, 35, 54](#)

OpenGL 4.3 [8](#)

OpenGL API [7](#)

OpenGL shading language (GLSL) [7](#)

OpenGL v2.0 [7](#)

OpenGL v3.0 [8](#)

OpenGL v3.3 [55, 141](#)

OpenGL v3.3 core profile

setting up, on Visual Studio 2010 [8-14](#)

order independent transparency

implementing, dual depth peeling used [189-193](#)

implementing, front-to-back peeling used [182-188](#)

P

Pan function [65](#)

particle system

about [171](#)

- implementing 171-179
- implementing, transform feedback used 296-305
- percentage closer filtering (PCF)**
 - about 128
 - shadow mapping, implementing with 128-131
- per-fragment directional light**
 - about 114
 - implementing 115, 116
- per-fragment point light**
 - implementing 108-113
 - implementing, with attenuation 117, 118
- per-fragment spot light**
 - about 120
 - implementing 120, 121
- per-vertex point lighting**
 - implementing 108-113
- Phong shading 108**
- physically based animations 261**
- plane**
 - subdividing, geometry shader used 37-44
- polygonal isosurface extraction**
 - implementing, Marching Tetrahedra algorithm used 248-253
- pseudo isosurface rendering**
 - implementing, in single-pass GPU ray casting 232-236
- Q**
- Quake's md2 (.md2) 163**
- Quaternions 273**
- R**
- reflective object**
 - rendering, dynamic cube mapping used 93-96
- relative transform. *See local transform***
- response on cloth model**
 - implementing 290-295
- ripple mesh deformer**
 - implementing, vertex shader used 28-36
- S**
- SampleVoxel function 238**
- scene intersection queries**
 - used, for implementing object picking 76-78
- screen space ambient occlusion (SSAO)**
 - about 195
 - implementing 196-201
- separate buffers**
 - used, for implementing 3DS model loading 146-155
- shader binding 18**
- shaders**
 - used, for rendering colored triangle 19-27
- shadow mapping**
 - implementing, with FBO 122-127
 - implementing, with percentage closer filtering (PCF) 128-131
- sharpening effect 100**
- single-pass GPU ray casting**
 - pseudo isosurface rendering, implementing 232-236
 - used, for implementing volume rendering 228-232
- skeletal animation**
 - about 261, 262
 - implementing, dual quaternion skinning used 273-279
 - implementing, matrix palette skinning used 262-271
- skinning 262**
- skinning matrix 268**
- skybox**
 - rendering, static cube mapping used 85-88
- SliceVolume function 224**
- smooth mouse filtering**
 - URL 59
- SOIL image loading library 142**
 - URL, for downloading 9
 - used, for drawing 2D image 48-54
- spherical harmonics lighting**
 - used, for implementing global illumination 202-206
- splatting**
 - used, for implementing volume rendering 237-244
- static cube mapping**
 - used, for rendering skybox 85-88
- Strafe function 62**

T

target camera

about 63
implementing 63-66

Terragen 146

TERRAIN_DEPTH parameter 143

terrain height map generation

tools 146

terrain rendering 144

terrains

implementing, height map used 142-145

TERRAIN_WIDTH parameter 143

tessellation control shader 16

tessellation evaluation shader 16

texelFetchBuffer function 284

transfer function

implementing, for volume classification
244-247

transform 266

transform feedback

about 279
used, for implementing particle system
296-304
used, for modeling cloth 279-289

twirl filter

implementing, fragment shader used 82-85

U

unweighted smoothing effect 100

V

variance shadow mapping

about 132
implementing 132-139

varying attributes 22

vector-based camera

implementing, with FPS style input support
56-59

vertex array object (VAO) 24, 239, 283

vertex buffer object (VBO) 239

vertex shader

about 16
used, for implementing ripple mesh deformer
28-36

view frustum culling

about 66
implementing 67-70

virtual terrain project

about 146
URL 146

volume classification

transfer function, implementing for 244-247

volume rendering

about 219, 220
implementing, 3D texture slicing used
220-227
implementing, single-pass GPU ray casting
used 228-232
implementing, splatting used 237-244

volumetric lighting

implementing, half angle slicing used
254-259

W

Walk function 62

Wavefront® OBJ 141

Wavefront ® Obj model

implementing, interleaved buffers used
157-161

world machine

about 146
URL 146

world space position 28

Z

Zoom function 65

Table of Contents

Module 2: OpenGL 4.0 Shading Language Cookbook, Second Edition

Chapter 1: Getting Started with GLSL	3
Introduction	3
Using a function loader to access the latest OpenGL functionality	6
Using GLM for mathematics	9
Determining the GLSL and OpenGL version	11
Compiling a shader	13
Linking a shader program	17
Sending data to a shader using vertex attributes and vertex buffer objects	21
Getting a list of active vertex input attributes and locations	30
Sending data to a shader using uniform variables	33
Getting a list of active uniform variables	37
Using uniform blocks and uniform buffer objects	39
Getting debug messages	45
Building a C++ shader program class	48
Chapter 2: The Basics of GLSL Shaders	53
Introduction	53
Implementing diffuse, per-vertex shading with a single point light source	56
Implementing per-vertex ambient, diffuse, and specular (ADS) shading	61
Using functions in shaders	67
Implementing two-sided shading	71
Implementing flat shading	75
Using subroutines to select shader functionality	77
Discarding fragments to create a perforated look	82

Table of Contents

Chapter 3: Lighting, Shading, and Optimization	87
Introduction	87
Shading with multiple positional lights	88
Shading with a directional light source	91
Using per-fragment shading for improved realism	94
Using the halfway vector for improved performance	97
Simulating a spotlight	99
Creating a cartoon shading effect	103
Simulating fog	106
Configuring the depth test	109
Chapter 4: Using Textures	113
Introduction	113
Applying a 2D texture	114
Applying multiple textures	120
Using alpha maps to discard pixels	123
Using normal maps	126
Simulating reflection with cube maps	132
Simulating refraction with cube maps	139
Applying a projected texture	145
Rendering to a texture	150
Using sampler objects	155
Chapter 5: Image Processing and Screen Space Techniques	159
Introduction	159
Applying an edge detection filter	160
Applying a Gaussian blur filter	167
Implementing HDR lighting with tone mapping	174
Creating a bloom effect	180
Using gamma correction to improve image quality	184
Using multisample anti-aliasing	187
Using deferred shading	192
Implementing order-independent transparency	198
Chapter 6: Using Geometry and Tessellation Shaders	211
Introduction	211
Point sprites with the geometry shader	216
Drawing a wireframe on top of a shaded mesh	221
Drawing silhouette lines using the geometry shader	229
Tessellating a curve	239
Tessellating a 2D quad	244
Tessellating a 3D surface	249
Tessellating based on depth	255

Table of Contents

Chapter 7: Shadows	259
Introduction	259
Rendering shadows with shadow maps	260
Anti-aliasing shadow edges with PCF	270
Creating soft shadow edges with random sampling	274
Creating shadows using shadow volumes and the geometry shader	280
Chapter 8: Using Noise in Shaders	289
Introduction	289
Creating a noise texture using GLM	291
Creating a seamless noise texture	294
Creating a cloud-like effect	296
Creating a wood-grain effect	298
Creating a disintegration effect	302
Creating a paint-spatter effect	305
Creating a night-vision effect	307
Chapter 9: Particle Systems and Animation	311
Introduction	311
Animating a surface with vertex displacement	312
Creating a particle fountain	316
Creating a particle system using transform feedback	322
Creating a particle system using instanced particles	331
Simulating fire with particles	335
Simulating smoke with particles	337
Chapter 10: Using Compute Shaders	341
Introduction	341
Implementing a particle simulation with the compute shader	345
Using the compute shader for cloth simulation	350
Implementing an edge detection filter with the compute shader	357
Creating a fractal texture using the compute shader	362

Module 3: OpenGL Data Visualization Cookbook

Chapter 1: Getting Started with OpenGL	369
Introduction	369
Setting up a Windows-based development platform	370
Setting up a Mac-based development platform	373
Setting up a Linux-based development platform	377
Installing the GLFW library in Windows	378
Installing the GLFW library in Mac OS X and Linux	381

Table of Contents

Creating your first OpenGL application with GLFW	384
Compiling and running your first OpenGL application in Windows	386
Compiling and running your first OpenGL application in Mac OS X or Linux	391
Chapter 2: OpenGL Primitives and 2D Data Visualization	395
Introduction	395
OpenGL primitives	396
Creating a 2D plot using primitives	406
Real-time visualization of time series	409
2D visualization of 3D/4D datasets	412
Chapter 3: Interactive 3D Data Visualization	417
Introduction	417
Setting up a virtual camera for 3D rendering	418
Creating a 3D plot with perspective rendering	421
Creating an interactive environment with GLFW	428
Rendering a volumetric dataset – MCML simulation	436
Chapter 4: Rendering 2D Images and Videos with Texture Mapping	447
Introduction	447
Getting started with modern OpenGL (3.2 or higher)	448
Setting up the GLEW, GLM, SOIL, and OpenCV libraries in Windows	449
Setting up the GLEW, GLM, SOIL, and OpenCV libraries in Mac OS X/Linux	454
Creating your first vertex and fragment shader using GLSL	457
Rendering 2D images with texture mapping	466
Real-time video rendering with filters	479
Chapter 5: Rendering of Point Cloud Data for 3D Range-sensing Cameras	489
Introduction	489
Getting started with the Microsoft Kinect (PrimeSense)	490
3D range-sensing camera	491
Capturing raw data from depth-sensing cameras	494
Chapter 6: Rendering Stereoscopic 3D Models using OpenGL	509
Introduction	509
Installing the Open Asset Import Library (Assimp)	510
Loading the first 3D model in the Wavefront Object (.obj) format	512
Rendering 3D models with points, lines, and triangles	518
Stereoscopic 3D rendering	529

Table of Contents

Chapter 7: An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0	537
Introduction	537
Setting up the Android SDK	538
Setting up the Android Native Development Kit (NDK)	541
Developing a basic framework to integrate the Android NDK	542
Creating your first Android application with OpenGL ES 3.0	548
Chapter 8: Interactive Real-time Data Visualization on Mobile Devices	561
Introduction	561
Visualizing real-time data from built-in Inertial Measurement Units (IMUs)	562
Part I – handling multi-touch interface and motion sensor inputs	581
Part II – interactive, real-time data visualization with mobile GPUs	587
Chapter 9: Augmented Reality-based Visualization on Mobile or Wearable Platforms	601
Introduction	601
Getting started I: Setting up OpenCV on Android	602
Getting started II: Accessing the camera live feed using OpenCV	604
Displaying real-time video using texture mapping	612
Augmented reality-based data visualization over real-world scenes	624
Bibliography	639
Index	641

Module 2

OpenGL 4.0 Shading Language Cookbook, Second Edition

Over 70 recipes demonstrating simple and advanced techniques for producing high-quality, real-time 3D graphics using OpenGL and GLSL 4.x

1

Getting Started with GLSL

In this chapter, we will cover the following recipes:

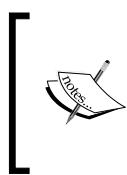
- ▶ Using a function loader to access the latest OpenGL functionality
- ▶ Using GLM for mathematics
- ▶ Determining the GLSL and OpenGL version
- ▶ Compiling a shader
- ▶ Linking a shader program
- ▶ Sending data to a shader using vertex attributes and vertex buffer objects
- ▶ Getting a list of active vertex input attributes and locations
- ▶ Sending data to a shader using uniform variables
- ▶ Getting a list of active uniform variables
- ▶ Using uniform blocks and uniform buffer objects
- ▶ Getting debug messages
- ▶ Building a C++ shader program class

Introduction

The **OpenGL Shading Language (GLSL)** Version 4 brings unprecedented power and flexibility to programmers interested in creating modern, interactive, and graphical programs. It allows us to harness the power of modern **Graphics Processing Units (GPUs)** in a straightforward way by providing a simple yet powerful language and API. Of course, the first step towards using GLSL is to create a program that utilizes the latest version of the OpenGL API. GLSL programs don't stand on their own; they must be a part of a larger OpenGL program. In this chapter, we will provide some tips and techniques for getting a basic program up and running. First, let's start with some background.

The OpenGL Shading Language

The GLSL is now a fundamental and integral part of the OpenGL API. Going forward, every program written using the OpenGL API will internally utilize one or several GLSL programs. These "mini-programs" are often referred to as **shader programs**. A shader program usually consists of several components called **shaders**. Each shader executes within a different section of the OpenGL pipeline. Each shader runs on the GPU, and as the name implies, (typically) implement the algorithms related to the lighting and shading effects of an image. However, shaders are capable of doing much more than just implementing a shading algorithm. They are also capable of performing animation, tessellation, or even generalized computation.



The field of study dubbed **GPGPU (General Purpose Computing on Graphics Processing Units)** is concerned with utilization of GPUs (often using specialized APIs such as CUDA or OpenCL) to perform general purpose computations such as fluid dynamics, molecular dynamics, cryptography, and so on. With compute shaders, introduced in OpenGL 4.3, we can now do GPGPU within OpenGL.

Shader programs are designed for direct execution on the GPU and are executed in parallel. For example, a fragment shader might be executed once for every pixel, with each execution running simultaneously on a separate GPU thread. The number of processors on the graphics card determines how many can be executed at one time. This makes shader programs incredibly efficient, and provides the programmer with a simple API for implementing highly parallel computation.

The computing power available in modern graphics cards is impressive. The following table shows the number of shader processors available for several models in the NVIDIA GeForce series cards (source: http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units).

Model	Unified Shader Processors
GeForce GTS 450	192
GeForce GTX 480	480
GeForce GTX 780	2304

Shader programs are intended to replace parts of the OpenGL architecture referred to as the **fixed-function pipeline**. Prior to OpenGL Version 2.0, the shading algorithm was "hard-coded" into the pipeline and had only limited configurability. This default lighting/shading algorithm was a core part of the fixed-function pipeline. When we, as programmers, wanted to implement more advanced or realistic effects, we used various tricks to force the fixed-function pipeline into being more flexible than it really was. The advent of GLSL will help by providing us with the ability to replace this "hard-coded" functionality with our own programs written in GLSL, thus giving us a great deal of additional flexibility and power. For more details on the programmable pipeline, see the introduction to *Chapter 2, The Basics of GLSL Shaders*.

In fact, recent (core) versions of OpenGL not only provide this capability, but they require shader programs as part of every OpenGL program. The old fixed-function pipeline has been deprecated in favor of a new programmable pipeline, a key part of which is the shader program written in GLSL.

Profiles – Core vs. Compatibility

OpenGL Version 3.0 introduced a **deprecation model**, which allowed for the gradual removal of functions from the OpenGL specification. Functions or features can be marked as deprecated, meaning that they are expected to be removed from a future version of OpenGL. For example, immediate mode rendering using `glBegin/glEnd` was marked deprecated in version 3.0 and removed in version 3.1.

In order to maintain backwards compatibility, the concept of **compatibility profiles** was introduced with OpenGL 3.2. A programmer that is writing code intended to be used with a particular version of OpenGL (with older features removed) would use the so-called **core profile**. Someone who also wanted to maintain compatibility with older functionality could use the compatibility profile.



It may be somewhat confusing that there is also the concept of a **forward compatible** context, which is distinguished slightly from the concept of a core/compatibility profile. A context that is considered forward compatible basically indicates that all deprecated functionality has been removed. In other words, if a context is forward compatible, it only includes functions that are in the core, but not those that were marked as deprecated. Some window APIs provide the ability to select forward compatible status along with the profile.

The steps for selecting a core or compatibility profile are window system API dependent. For example, in GLFW, one can select a forward compatible, 4.3 core profile using the following code:

```
glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 4 );
glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 3 );
glfwWindowHint( GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE );
glfwWindowHint( GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE );

GLFWwindow *window = glfwCreateWindow(640, 480, "Title",
                                     NULL, NULL);
```

All programs in this book are designed to be compatible with a forward compatible OpenGL 4.3 core profile.

Using a function loader to access the latest OpenGL functionality

The OpenGL **ABI (application binary interface)** is frozen to OpenGL version 1.1 on Windows. Unfortunately for Windows developers, that means that it is not possible to link directly to functions that are provided in newer versions of OpenGL. Instead, one must get access to these functions by acquiring a function pointer at runtime. Getting access to the function pointers is not difficult, but requires somewhat tedious work, and has a tendency to clutter your code. Additionally, Windows typically comes with a standard OpenGL `g1.h` file that also conforms to OpenGL 1.1. The OpenGL wiki states that Microsoft has no plans to ever update the `g1.h` and `opengl32.lib` that come with their compilers. Thankfully, others have provided libraries that manage all of this for us by transparently providing the needed function pointers, while also exposing the needed functionality in header files. There are several libraries available that provide this kind of support. One of the oldest and most common is **GLEW (OpenGL Extension Wrangler)**. However, there are a few serious issues with GLEW that might make it less desirable, and insufficient for my purposes when writing this book. First, at time of writing, it doesn't yet support core profiles properly, and for this book, I want to focus only on the latest non-deprecated functionality. Second, it provides one large header file that includes everything from all versions of OpenGL. It might be preferable to have a more streamlined header file that only includes functions that we might use. Finally, GLEW is distributed as a library that needs to be compiled separately and linked into our project. It is often preferable to have a loader that can be included into a project simply by adding the source files and compiling them directly into our executable, avoiding the need to support another link-time dependency.

In this recipe, we'll use the **OpenGL Loader Generator** (GLLoadGen), available from <https://bitbucket.org/alfONSE/glloadgen/wiki/Home>. This very flexible and efficient library solves all three of the issues described in the previous paragraph. It supports core profiles and it can generate a header that includes only the needed functionality, and also generates just a couple of files (a source file and a header) that we can add directly into our project.

Getting ready

To use GLLoadGen, you'll need **Lua**. Lua is a lightweight embeddable scripting language that is available for nearly all platforms. Binaries are available at <http://luabinaries.sourceforge.net>, and a fully packaged install for Windows (LuaForWindows) is available at:

<https://code.google.com/p/luaforwindows>

Download the GLLoadGen distribution from: <https://bitbucket.org/alfONSE/glloadgen/downloads>. The distribution is compressed using 7zip, which is not widely installed, so you may need to install a 7zip utility, available at <http://7-zip.org/>. Extract the distribution to a convenient location on your hard drive. Since GLLoadGen is written in Lua, there's nothing to compile, once the distribution is uncompressed, you're ready to go.

How to do it...

The first step is to generate the header and source files for the OpenGL version and profile of choice. For this example, we'll generate files for an OpenGL 4.3 core profile. We can then copy the files into our project and compile them directly alongside our code:

1. To generate the header and source files, navigate to the GLLoadGen distribution directory, and run GLLoadGen with the following arguments:
`lua LoadGen.lua -style=pointer_c -spec=gl -version=4.3 \
-profile=core core_4_3`
2. The previous step should generate two files: `gl_core_4_3.c` and `gl_core_4_3.h`. Move these files into your project and include `gl_core_4_3.c` in your build. Within your program code, you can include the `gl_core_4_3.h` file whenever you need access to the OpenGL functions. However, in order to initialize the function pointers, you need to make sure to call a function to do so. The needed function is called `ogl_LoadFunctions`. Somewhere just after the GL context is created (typically in an initialization function), and before any OpenGL functions are called, use the following code:

```
int loaded = ogl_LoadFunctions();
if(loaded == ogl_LOAD_FAILED) {
    //Destroy the context and abort
    return;
}

int num_failed = loaded - ogl_LOAD_SUCCEEDED;
printf("Number of functions that failed to load: %i.\n",
    num_failed);
```

That's all there is to it!

How it works...

The `lua` command in step 1 generates a pair of files, that is; a header and a source file. The header provides prototypes for all of the selected OpenGL functions and redefines them as function pointers, and defines all of the OpenGL constants as well. The source file provides initialization code for the function pointers as well as some other utility functions. We can include the `gl_core_4_3.h` header file wherever we need prototypes for OpenGL functions, so all function entry points are available at compile time. At run time, the `ogl_LoadFunctions()` function will initialize all available function pointers. If some functions fail to load, the number of failures can be determined by the subtraction operation shown in step 2. If a function is not available in the selected OpenGL version, the code may not compile, because only function prototypes for the selected OpenGL version and profile are available in the header (depending on how it was generated).

The command line arguments available to GLLoadGen are fully documented here:
https://bitbucket.org/alfonse/glloadgen/wiki/Command_Line_Options.
The previous example shows the most commonly used setup, but there's a good amount of flexibility built into this tool.

Now that we have generated this source/header pair, we no longer have any dependency on GLLoadGen and our program can be compiled without it. This is a significant advantage over tools such as GLEW.

There's more...

GLLoadGen includes a few additional features that are quite useful. We can generate more C++ friendly code, manage extensions, and generate files that work without the need to call an initialization function.

Generating a C++ loader

GLLoadGen supports generation of C++ header/source files as well. This can be selected via the `-style` parameter. For example, to generate C++ files, use `-style=pointer_cpp` as in the following example:

```
lua LoadGen.lua -style=pointer_cpp -spec=gl -version=4.3 \
-profile=core core_4_3
```

This will generate `gl_core_4_3.cpp` and `gl_core_4_3.hpp`. This places all OpenGL functions and constants within the `gl::` namespace, and removes their `gl` (or `GL`) prefix. For example, to call the function `glBufferData`, you might use the following syntax.

```
gl::BufferData(gl::ARRAY_BUFFER, size, data, gl::STATIC_DRAW);
```

Loading the function pointers is also slightly different. The return value is an object rather than just a simple integer and `LoadFunctions` is in the `gl::sys` namespace.

```
gl::exts::LoadTest didLoad = gl::sys::LoadFunctions();

if(!didLoad) {
    // Clean up (destroy the context) and abort.
    return;
}

printf("Number of functions that failed to load: %i.\n",
    didLoad.GetNumMissing());
```

No-load styles

GLLoadGen supports the automatic initialization of function pointers. This can be selected using the `noload_c` or `noload_cpp` options for the `style` parameter. With these styles, there is no need to call the initialization function `ogl_LoadFunctions`. The pointers are loaded automatically, the first time a function is called. This can be convenient, but there's very little overhead to loading them all at initialization.

Using Extensions

GLLoadGen does not automatically support extensions. Instead, you need to ask for them with command line parameters. For example, to request `ARB_texture_view` and `ARB_vertex_attrib_binding` extensions, you might use the following command.

```
lua LoadGen.lua -style=pointer_c -spec=gl -version=3.3 \
-profile=core core_3_3 \
-exts ARB_texture_view ARB_vertex_attrib_binding
```

The `-exts` parameter is a space-separated list of extensions. GLLoadGen also provides the ability to load a list of extensions from a file (via the `-extfile` parameter) and provides some common extension files on the website.

You can also use GLLoadGen to check for the existence of an extension at run-time. For details, see the GLLoadGen wiki.

See also

- ▶ GLEW, an older, and more common loader and extension manager, available from glew.sourceforge.net.

Using GLM for mathematics

Mathematics is core to all of computer graphics. In earlier versions, OpenGL provided support for managing coordinate transformations and projections using the standard matrix stacks (`GL_MODELVIEW` and `GL_PROJECTION`). In recent versions of core OpenGL however, all of the functionality supporting the matrix stacks has been removed. Therefore, it is up to us to provide our own support for the usual transformation and projection matrices, and then to pass them into our shaders. Of course, we could write our own matrix and vector classes to manage this, but some might prefer to use a ready-made, robust library.

One such library is **GLM (OpenGL Mathematics)** written by *Christophe Riccio*. Its design is based on the GLSL specification, so the syntax is very similar to the mathematical support in GLSL. For experienced GLSL programmers, this makes GLM very easy to use and familiar. Additionally, it provides extensions that include functionality similar to some of the much-missed OpenGL functions such as `glOrtho`, `glRotate`, or `gluLookAt`.

Getting ready

Since GLM is a header-only library, installation is simple. Download the latest GLM distribution from <http://glm.g-truc.net>. Then, unzip the archive file, and copy the `glm` directory contained inside to anywhere in your compiler's include path.

How to do it...

To use the GLM libraries, it is simply a matter of including the core header file, and headers for any extensions. For this example, we'll include the matrix transform extension as follows:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

Then the GLM classes are available in the `glm` namespace. The following is an example of how you might go about making use of some of them:

```
glm::vec4 position = glm::vec4( 1.0f, 0.0f, 0.0f, 1.0f );
glm::mat4 view = glm::lookAt( glm::vec3(0.0,0.0,5.0),
    glm::vec3(0.0,0.0,0.0),
    glm::vec3(0.0,1.0,0.0) );
glm::mat4 model(1.0f); // The identity matrix
model = glm::rotate( model, 90.0f, glm::vec3(0.0f,1.0f,0.0) );
glm::mat4 mv = view * model;
glm::vec4 transformed = mv * position;
```

How it works...

The GLM library is a header-only library. All of the implementation is included within the header files. It doesn't require separate compilation and you don't need to link your program to it. Just placing the header files in your include path is all that's required!

The previous example first creates a `vec4` (four coordinate vector) representing a position. Then it creates a 4×4 view matrix by using the `glm::lookAt` function. This works in a similar fashion to the old `gluLookAt` function. Here, we set the camera's location at $(0, 0, 5)$, looking towards the origin, with the "up" direction in the direction of the y-axis. We then go on to create the model matrix by first storing the identity matrix in the variable `model` (via the single argument constructor), and multiplying by a rotation matrix using the `glm::rotate` function. The multiplication here is implicitly done by the `glm::rotate` function. It multiplies its first parameter by the rotation matrix (on the right) that is generated by the function. The second parameter is the angle of rotation (in degrees), and the third parameter is the axis of rotation. Since before this statement, `model` is the identity matrix, the net result is that `model` becomes a rotation matrix of 90 degrees around the y-axis.

Finally, we create our modelview matrix (`mv`) by multiplying the `view` and `model` variables, and then using the combined matrix to transform the position. Note that the multiplication operator has been overloaded to behave in the expected way.

There's more...

It is not recommended to import all of the GLM namespace by using the following command:

```
using namespace glm;
```

This will most likely cause a number of namespace clashes. Instead, it is preferable to import symbols one at a time, as needed. For example:

```
#include <glm/glm.hpp>
using glm::vec3;
using glm::mat4;
```

Using the GLM types as input to OpenGL

GLM supports directly passing a GLM type to OpenGL using one of the OpenGL vector functions (with the suffix `v`). For example, to pass a `mat4` named `proj` to OpenGL we can use the following code:

```
glm::mat4 proj = glm::perspective( viewAngle, aspect, nearDist,
                                    farDist );
glUniformMatrix4fv(location, 1, GL_FALSE, &proj[0][0]);
```

See also

- ▶ The Qt SDK includes many classes for vector/matrix mathematics, and is another good option if you're already using Qt
- ▶ The GLM website <http://glm.g-truc.net> has additional documentation and examples

Determining the GLSL and OpenGL version

In order to support a wide range of systems, it is essential to be able to query for the supported OpenGL and GLSL version of the current driver. It is quite simple to do so, and there are two main functions involved: `glGetString` and `glGetIntegerv`.

How to do it...

The code shown as follows will print the version information to `stdout`:

```
const GLubyte *renderer = glGetString( GL_RENDERER );
const GLubyte *vendor = glGetString( GL_VENDOR );
const GLubyte *version = glGetString( GL_VERSION );
const GLubyte *glslVersion =
    glGetString( GL_SHADING_LANGUAGE_VERSION );

GLint major, minor;
glGetIntegerv(GL_MAJOR_VERSION, &major);
glGetIntegerv(GL_MINOR_VERSION, &minor);

printf("GL Vendor           : %s\n", vendor);
printf("GL Renderer         : %s\n", renderer);
printf("GL Version (string) : %s\n", version);
printf("GL Version (integer) : %d.%d\n", major, minor);
printf("GLSL Version        : %s\n", glslVersion);
```

How it works...

Note that there are two different ways to retrieve the OpenGL version: using `glGetString` and `glGetIntegerv`. The former can be useful for providing readable output, but may not be as convenient for programmatically checking the version because of the need to parse the string. The string provided by `glGetString(GL_VERSION)` should always begin with the major and minor versions separated by a dot, however, the minor version could be followed with a vendor-specific build number. Additionally, the rest of the string can contain additional vendor-specific information and may also include information about the selected profile (see the *Introduction* section of this chapter). It is important to note that the use of `glGetIntegerv` to query for version information requires OpenGL 3.0 or greater.

The queries for `GL_VENDOR` and `GL_RENDERER` provide additional information about the OpenGL driver. The call `glGetString(GL_VENDOR)` returns the company responsible for the OpenGL implementation. The call to `glGetString(GL_RENDERER)` provides the name of the renderer which is specific to a particular hardware platform (such as the ATI Radeon HD 5600 Series). Note that both of these do not vary from release to release, so can be used to determine the current platform.

Of more importance to us in the context of this book is the call to `glGetString(GL_SHADING_LANGUAGE_VERSION)` which provides the supported GLSL version number. This string should begin with the major and minor version numbers separated by a period, but similar to the `GL_VERSION` query, may include other vendor-specific information.

There's more...

It is often useful to query for the supported extensions of the current OpenGL implementation. In versions prior to OpenGL 3.0, one could retrieve a full, space separated list of extension names with the following code:

```
GLubyte *extensions = glGetString(GL_EXTENSIONS) ;
```

The string that is returned can be extremely long and parsing it can be susceptible to error if not done carefully.

In OpenGL 3.0, a new technique was introduced, and the previous functionality was deprecated (and finally removed in 3.1). Extension names are now indexed and can be individually queried by index. We use the `glGetStringi` variant for this. For example, to get the name of the extension stored at index `i`, we use: `glGetStringi(GL_EXTENSIONS, i)`. To print a list of all extensions, we could use the following code:

```
GLint nExtensions;
glGetIntegerv(GL_NUM_EXTENSIONS, &nExtensions);

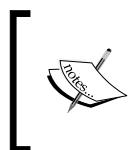
for( int i = 0; i < nExtensions; i++ )
    printf("%s\n", glGetStringi(GL_EXTENSIONS, i) );
```

See also

- ▶ The `GLLoadGen` tool has additional support for querying version and extension information. Refer to the *Using a function loader to access the latest OpenGL functionality* recipe and the `GLLoadGen` website.

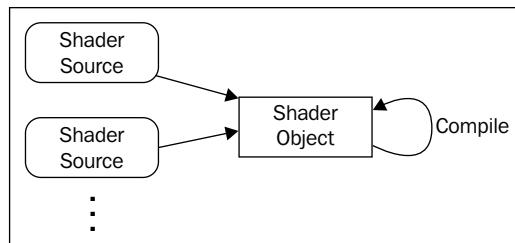
Compiling a shader

To get started, we need to know how to compile our GLSL shaders. The GLSL compiler is built right into the OpenGL library, and shaders can only be compiled within the context of a running OpenGL program. There is currently no external tool for precompiling GLSL shaders and/or shader programs.



Recently, OpenGL 4.1 added the ability to save compiled shader programs to a file, enabling OpenGL programs to avoid the overhead of shader compilation by loading pre-compiled shader programs.

Compiling a shader involves creating a shader object, providing the source code (as a string or set of strings) to the shader object, and asking the shader object to compile the code. The process is roughly represented by the following diagram:



Getting ready

To compile a shader, we'll need a basic example to work with. Let's start with the following simple vertex shader. Save it in a file named `basic.vert`.

```
#version 430

in vec3 VertexPosition;
in vec3 VertexColor;

out vec3 Color;

void main()
{
    Color = VertexColor;
    gl_Position = vec4( VertexPosition, 1.0 );
}
```

In case you're curious about what this code does, it works as a "pass-through" shader. It takes the input attributes `VertexPosition` and `VertexColor` and passes them along to the fragment shader via the output variables `gl_Position` and `Color`.

Next, we'll need to build a basic shell for an OpenGL program using a window toolkit that supports OpenGL. Examples of cross-platform toolkits include GLFW, GLUT, FLTK, Qt, or wxWidgets. Throughout this text, I'll make the assumption that you can create a basic OpenGL program with your favorite toolkit. Virtually all toolkits have a hook for an initialization function, a resize callback (called upon resizing of the window), and a drawing callback (called for each window refresh). For the purposes of this recipe, we need a program that creates and initializes an OpenGL context; it need not do anything other than display an empty OpenGL window. Note that you'll also need to load the OpenGL function pointers (refer to the *Using a function loader to access the latest OpenGL functionality* recipe).

Finally, load the shader source code into a character array named `shaderCode`. Don't forget to add the null character at the end! This example assumes that the variable `shaderCode` points to an array of `GLchar` that is properly terminated by a null character.

How to do it...

To compile a shader, use the following steps:

1. Create the shader object as follows:

```
GLuint vertShader = glCreateShader( GL_VERTEX_SHADER );
if( 0 == vertShader )
{
    fprintf(stderr, "Error creating vertex shader.\n");
    exit(EXIT_FAILURE);
}
```

2. Copy the source code (perhaps from multiple locations) into the shader object:

```
const GLchar * shaderCode = loadShaderAsString("basic.vert");
const GLchar* codeArray[] = { shaderCode };
glShaderSource( vertShader, 1, codeArray, NULL );
```

3. Compile the shader:

```
glCompileShader( vertShader );
```

4. Verify the compilation status:

```
GLint result;
glGetShaderiv( vertShader, GL_COMPILE_STATUS, &result );
if( GL_FALSE == result )
{
    fprintf(stderr, "Vertex shader compilation failed!\n");

    GLint logLen;
    glGetShaderiv(vertShader, GL_INFO_LOG_LENGTH, &logLen);

    if( logLen > 0 )
    {
        char * log = new char[logLen];

        GLsizei written;
        glGetShaderInfoLog(vertShader, logLen, &written, log);

        fprintf(stderr, "Shader log:\n%s", log);
        delete [] log;
    }
}
```

How it works...

The first step is to create the shader object using the `glCreateShader` function. The argument is the type of shader, and can be one of the following: `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, `GL_TESS_EVALUATION_SHADER`, `GL_TESS_CONTROL_SHADER`, or (as of version 4.3) `GL_COMPUTE_SHADER`. In this case, since we are compiling a vertex shader, we use `GL_VERTEX_SHADER`. This function returns the value used for referencing the vertex shader object, sometimes called the object "handle". We store that value in the variable `vertShader`. If an error occurs while creating the shader object, this function will return 0, so we check for that and if it occurs, we print an appropriate message and terminate.

Following the creation of the shader object, we load the source code into the shader object using the function `glShaderSource`. This function is designed to accept an array of strings (as opposed to just a single one) in order to support the option of compiling multiple sources (files, strings) at once. So before we call `glShaderSource`, we place a pointer to our source code into an array named `codeArray`. The first argument to `glShaderSource` is the handle to the shader object. The second is the number of source code strings that are contained in the array. The third argument is a pointer to an array of source code strings. The final argument is an array of `GLint` values that contains the length of each source code string in the previous argument. In the previous code, we pass a value of `NULL`, which indicates that each source code string is terminated by a null character. If our source code strings were not null terminated then this argument must be a valid array. Note that once this function returns, the source code has been copied into OpenGL internal memory, so the memory used to store the source code can be freed.

The next step is to compile the source code for the shader. We do this by simply calling `glCompileShader`, and passing the handle to the shader that is to be compiled. Of course, depending on the correctness of the source code, the compilation may fail, so the next step is to check whether or not the compilation was successful.

We can query for the compilation status by calling `glGetShaderiv`, which is a function for querying the attributes of a shader object. In this case we are interested in the compilation status, so we use `GL_COMPILE_STATUS` as the second argument. The first argument is of course the handle to the shader object, and the third argument is a pointer to an integer where the status will be stored. The function provides a value of either `GL_TRUE` or `GL_FALSE` in the third argument indicating whether or not the compilation was successful.

If the compile status is `GL_FALSE`, then we can query for the shader log, which will provide additional details about the failure. We do so by first querying for the length of the log by calling `glGetShaderiv` again with a value of `GL_INFO_LOG_LENGTH`. This provides the length of the log in the variable `logLen`. Note that this includes the null termination character. We then allocate space for the log, and retrieve the log by calling `glGetShaderInfoLog`. The first parameter is the handle to the shader object, the second is the size of the character buffer for storing the log, the third argument is a pointer to an integer where the number of characters actually written (excluding the null terminator character) will be stored, and the fourth argument is a pointer to the character buffer for storing the log itself. Once the log is retrieved, we print it to `stderr` and free its memory space.

There's more...

The previous example only demonstrated compiling a vertex shader. There are several other types of shaders including fragment, geometry, and tessellation shaders. The technique for compiling is nearly identical for each shader type. The only significant difference is the argument to `glCreateShader`.

It is also important to note that shader compilation is only the first step. To create a working shader program, we often have at least two shaders to compile, and then the shaders must be linked together into a shader program object. We'll see the steps involved in linking in the next recipe.

Deleting a Shader Object

Shader objects can be deleted when no longer needed by calling `glDeleteShader`. This frees the memory used by the shader and invalidates its handle. Note that if a shader object is already attached to a program object (refer to the *Linking a shader program* recipe), it will not be immediately deleted, but flagged for deletion when it is detached from the program object.

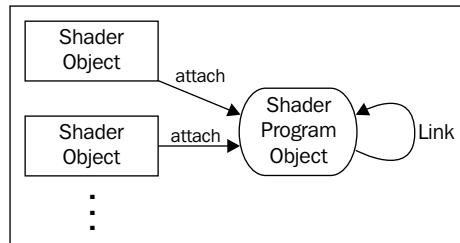
See also

- ▶ The *Linking a shader program* recipe

Linking a shader program

Once we have compiled our shaders and before we can actually install them into the OpenGL pipeline, we need to link them together into a shader program. Among other things, the linking step involves making the connections between input variables from one shader to output variables of another, and making the connections between the input/output variables of a shader to appropriate locations in the OpenGL environment.

Linking involves steps that are similar to those involved in compiling a shader. We attach each shader object to a new shader program object and then tell the shader program object to link (making sure that the shader objects are compiled before linking).



Getting ready

For this recipe, we'll assume that you've already compiled two shader objects whose handles are stored in the variables `vertShader` and `fragShader`.

For this and a few other recipes in this Chapter, we'll use the following source code for the fragment shader:

```
#version 430

in vec3 Color;

out vec4 FragColor;

void main() {
    FragColor = vec4(Color, 1.0);
}
```

For the vertex shader, we'll use the source code from the previous recipe, *Compiling a shader*.

How to do it...

In our OpenGL initialization function, and after the compilation of shader objects referred to by `vertShader` and `fragShader`, use the following steps:

1. Create the program object using the following code:

```
GLuint programHandle = glCreateProgram();
if( 0 == programHandle )
{
    fprintf(stderr, "Error creating program object.\n");
    exit(1);
}
```

2. Attach the shaders to the program object as follows:

```
glAttachShader( programHandle, vertShader );
glAttachShader( programHandle, fragShader );
```

3. Link the program:

```
glLinkProgram( programHandle );
```

4. Verify the link status:

```
GLint status;
glGetProgramiv( programHandle, GL_LINK_STATUS, &status );
if( GL_FALSE == status ) {

    fprintf( stderr, "Failed to link shader program!\n" );
    GLint logLen;
```

```
glGetProgramiv(programHandle, GL_INFO_LOG_LENGTH,
               &logLen);
if( logLen > 0 )
{
    char * log = new char[logLen];
    GLsizei written;
    glGetProgramInfoLog(programHandle, logLen, &written, log);
    fprintf(stderr, "Program log: \n%s", log);
    delete [] log;
}
```

5. If linking is successful, install the program into the OpenGL pipeline:

```
else
{
    glUseProgram( programHandle );
}
```

How it works...

We start by calling `glCreateProgram` to create an empty program object. This function returns a handle to the program object, which we store in a variable named `programHandle`. If an error occurs with program creation, the function will return 0. We check for that, and if it occurs, we print an error message and exit.

Next, we attach each shader to the program object using `glAttachShader`. The first argument is the handle to the program object, and the second is the handle to the shader object to be attached.

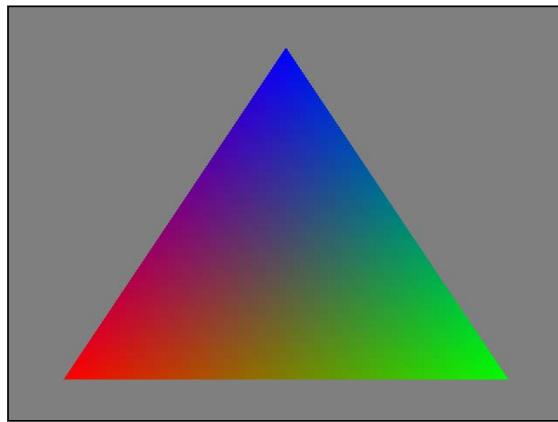
Then, we link the program by calling `glLinkProgram`, providing the handle to the program object as the only argument. As with compilation, we check for the success or failure of the link, with the subsequent query.

We check the status of the link by calling `glGetProgramiv`. Similar to `glGetShaderiv`, `glGetProgramiv` allows us to query various attributes of the shader program. In this case, we ask for the status of the link by providing `GL_LINK_STATUS` as the second argument. The status is returned in the location pointed to by the third argument, in this case named `status`.

The link status is either `GL_TRUE` or `GL_FALSE` indicating the success or failure of the link. If the value of `status` is `GL_FALSE`, we retrieve and display the program information log, which should contain additional information and error messages. The program log is retrieved by the call to `glGetProgramInfoLog`. The first argument is the handle to the program object, the second is the size of the buffer to contain the log, the third is a pointer to a `GLsizei` variable where the number of bytes written to the buffer will be stored (excluding the null terminator), and the fourth is a pointer to the buffer that will store the log. The buffer can be allocated based on the size returned by the call to `glGetProgramiv` with the parameter `GL_INFO_LOG_LENGTH`. The string that is provided in `log` will be properly null terminated.

Finally, if the link is successful, we install the program into the OpenGL pipeline by calling `glUseProgram`, providing the handle to the program as the argument.

With the simple fragment shader from this recipe and the vertex shader from the previous recipe compiled, linked, and installed into the OpenGL pipeline, we have a complete OpenGL pipeline and are ready to begin rendering. Drawing a triangle and supplying different values for the `Color` attribute yields an image of a multi-colored triangle where the vertices are red, green, and blue, and inside the triangle, the three colors are interpolated, causing a blending of colors throughout.



There's more...

You can use multiple shader programs within a single OpenGL program. They can be swapped in and out of the OpenGL pipeline by calling `glUseProgram` to select the desired program.

Deleting a Shader program

If a program is no longer needed, it can be deleted from OpenGL memory by calling `glDeleteProgram`, providing the program handle as the only argument. This invalidates the handle and frees the memory used by the program. Note that if the program object is currently in use, it will not be immediately deleted, but will be flagged for deletion when it is no longer in use.

Also, the deletion of a shader program detaches the shader objects that were attached to the program but does not delete them unless those shader objects have already been flagged for deletion by a previous call to `glDeleteShader`.

See also

- ▶ The [Compiling a shader](#) recipe

Sending data to a shader using vertex attributes and vertex buffer objects

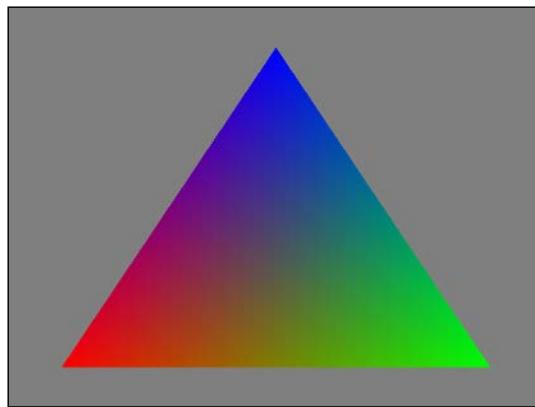
The vertex shader is invoked once per vertex. Its main job is to process the data associated with the vertex, and pass it (and possibly other information) along to the next stage of the pipeline. In order to give our vertex shader something to work with, we must have some way of providing (per-vertex) input to the shader. Typically, this includes the vertex position, normal vector, and texture coordinates (among other things). In earlier versions of OpenGL (prior to 3.0), each piece of vertex information had a specific "channel" in the pipeline. It was provided to the shaders using functions such as `glVertex`, `glTexCoord`, and `glNormal` (or within client vertex arrays using `glVertexPointer`, `glTexCoordPointer`, or `glNormalPointer`). The shader would then access these values via built-in variables such as `gl_Vertex` and `gl_Normal`. This functionality was deprecated in OpenGL 3.0 and later removed. Instead, vertex information must now be provided using *generic vertex attributes*, usually in conjunction with (*vertex*) *buffer objects*. The programmer is now free to define an arbitrary set of per-vertex attributes to provide as input to the vertex shader. For example, in order to implement normal mapping, the programmer might decide that position, normal vector and tangent vector should be provided along with each vertex. With OpenGL 4, it's easy to define this as the set of input attributes. This gives us a great deal of flexibility to define our vertex information in any way that is appropriate for our application, but may require a bit of getting used to for those of us who are used to the old way of doing things.

In the vertex shader, the per-vertex input attributes are defined by using the GLSL qualifier `in`. For example, to define a 3-component vector input attribute named `VertexColor`, we use the following code:

```
in vec3 VertexColor;
```

Of course, the data for this attribute must be supplied by the OpenGL program. To do so, we make use of vertex buffer objects. The buffer object contains the values for the input attribute. In the main OpenGL program we make the connection between the buffer and the input attribute and define how to "step through" the data. Then, when rendering, OpenGL pulls data for the input attribute from the buffer for each invocation of the vertex shader.

For this recipe, we'll draw a single triangle. Our vertex attributes will be position and color. We'll use a fragment shader to blend the colors of each vertex across the triangle to produce an image similar to the one shown as follows. The vertices of the triangle are red, green, and blue, and the interior of the triangle has those three colors blended together. The colors may not be visible in the printed text, but the variation in the shade should indicate the blending.



Getting ready

We'll start with an empty OpenGL program, and the following shaders:

The vertex shader (`basic.vert`):

```
#version 430

layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexColor;

out vec3 Color;

void main()
{
    Color = VertexColor;

    gl_Position = vec4(VertexPosition, 1.0);
}
```

Attributes are the input variables to a vertex shader. In the previous code, there are two input attributes: `VertexPosition` and `VertexColor`. They are specified using the GLSL keyword `in`. Don't worry about the `layout` prefix, we'll discuss that later. Our main OpenGL program needs to supply the data for these two attributes for each vertex. We will do so by mapping our polygon data to these variables.

It also has one output variable named `Color`, which is sent to the fragment shader. In this case, `Color` is just an unchanged copy of `VertexColor`. Also, note that the attribute `VertexPosition` is simply expanded and passed along to the built-in output variable `gl_Position` for further processing.

The fragment shader (`basic.frag`):

```
#version 430

in vec3 Color;

out vec4 FragColor;

void main() {
    FragColor = vec4(Color, 1.0);
}
```

There is just one input variable for this shader, `Color`. This links to the corresponding output variable in the vertex shader, and will contain a value that has been interpolated across the triangle based on the values at the vertices. We simply expand and copy this color to the output variable `FragColor` (more about fragment shader output variables in later recipes).

Write code to compile and link these shaders into a shader program (see "Compiling a shader" and "Linking a shader program"). In the following code, I'll assume that the handle to the shader program is `programHandle`.

How to do it...

Use the following steps to set up your buffer objects and render the triangle:

1. Create a global (or private instance) variable to hold our handle to the vertex array object:

```
GLuint vaoHandle;
```

2. Within the initialization function, we create and populate the vertex buffer objects for each attribute:

```
float positionData[] = {
    -0.8f, -0.8f, 0.0f,
    0.8f, -0.8f, 0.0f,
    0.0f, 0.8f, 0.0f };
float colorData[] = {
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f };
```

```
// Create and populate the buffer objects
GLuint vboHandles[2];
glGenBuffers(2, vboHandles);
GLuint positionBufferHandle = vboHandles[0];
GLuint colorBufferHandle = vboHandles[1];

// Populate the position buffer
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float),
             positionData, GL_STATIC_DRAW);

// Populate the color buffer
glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(float), colorData,
             GL_STATIC_DRAW);
```

3. Create and define a vertex array object, which defines the relationship between the buffers and the input attributes. (See "There's more..." for an alternate way to do this that is valid for OpenGL 4.3 and later.)

```
// Create and set-up the vertex array object
 glGenVertexArrays( 1, &vaoHandle );
 glBindVertexArray(vaoHandle);

// Enable the vertex attribute arrays
 glEnableVertexAttribArray(0); // Vertex position
 glEnableVertexAttribArray(1); // Vertex color

// Map index 0 to the position buffer
 glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);

// Map index 1 to the color buffer
 glBindBuffer(GL_ARRAY_BUFFER, colorBufferHandle);
 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
```

4. In the render function, we bind to the vertex array object and call `glDrawArrays` to initiate rendering:

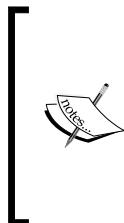
```
 glBindVertexArray(vaoHandle);
 glDrawArrays(GL_TRIANGLES, 0, 3 );
```

How it works...

Vertex attributes are the input variables to our vertex shader. In the given vertex shader, our two attributes are `VertexPosition` and `VertexColor`. The main OpenGL program refers to vertex attributes by associating each (active) input variable with a generic attribute index. These generic indices are simply integers between 0 and `GL_MAX_VERTEX_ATTRIBS` – 1. We can specify the relationship between these indices and the attributes using the `layout` qualifier. For example, in our vertex shader, we use the layout qualifier to assign `VertexPosition` to attribute index 0 and `VertexColor` to attribute index 1.

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;
```

We refer to the vertex attributes in our OpenGL code, by referring to the corresponding generic vertex attribute index.



It is not strictly necessary to explicitly specify the mappings between attribute variables and generic attribute indexes, because OpenGL will automatically map active vertex attributes to generic indexes when the program is linked. We could then query for the mappings and determine the indexes that correspond to the shader's input variables. It may be somewhat more clear however, to explicitly specify the mapping as we do in this example.



The first step involves setting up a pair of buffer objects to store our position and color data. As with most OpenGL objects, we start by creating the objects and acquiring handles to the two buffers by calling `glGenBuffers`. We then assign each handle to a separate descriptive variable to make the following code more clear.

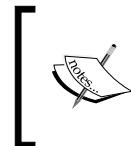
For each buffer object, we first bind the buffer to the `GL_ARRAY_BUFFER` binding point by calling `glBindBuffer`. The first argument to `glBindBuffer` is the target binding point. In this case, since the data is essentially a generic array, we use `GL_ARRAY_BUFFER`. Examples of other kinds of targets (such as `GL_UNIFORM_BUFFER`, or `GL_ELEMENT_ARRAY_BUFFER`) will be seen in later examples. Once our buffer object is bound, we can populate the buffer with our vertex/color data by calling `glBufferData`. The second and third arguments to this function are the size of the array and a pointer to the array containing the data. Let's focus on the first and last arguments. The first argument indicates the target buffer object. The data provided in the third argument is copied into the buffer that is bound to this binding point. The last argument is one that gives OpenGL a hint about how the data will be used so that it can determine how best to manage the buffer internally. For full details about this argument, take a look into the OpenGL documentation. In our case, the data is specified once, will not be modified, and will be used many times for drawing operations, so this usage pattern best corresponds to the value `GL_STATIC_DRAW`.

Now that we have set up our buffer objects, we tie them together into a **Vertex Array Object (VAO)**. The VAO contains information about the connections between the data in our buffers and the input vertex attributes. We create a VAO using the function `glGenVertexArrays`. This gives us a handle to our new object, which we store in the (global) variable `vaoHandle`. Then we enable the generic vertex attribute indexes 0 and 1 by calling `glEnableVertexAttribArray`. Doing so indicates that the values for the attributes will be accessed and used for rendering.

The next step makes the connection between the buffer objects and the generic vertex attribute indexes.

```
// Map index 0 to the position buffer
glBindBuffer(GL_ARRAY_BUFFER, positionBufferHandle);
glVertexAttribPointer( 0, 3, GL_FLOAT, GL_FALSE, 0, NULL );
```

First we bind the buffer object to the `GL_ARRAY_BUFFER` binding point, then we call `glVertexAttribPointer`, which tells OpenGL which generic index that the data should be used with, the format of the data stored in the buffer object, and where it is located within the buffer object that is bound to the `GL_ARRAY_BUFFER` binding point. The first argument is the generic attribute index. The second is the number of components per vertex attribute (1, 2, 3, or 4). In this case, we are providing 3-dimensional data, so we want 3 components per vertex. The third argument is the data type of each component in the buffer. The fourth is a Boolean which specifies whether or not the data should be automatically normalized (mapped to a range of [-1, 1] for signed integral values or [0, 1] for unsigned integral values). The fifth argument is the stride, which indicates the byte offset between consecutive attributes. Since our data is tightly packed, we can use a value of zero. The last argument is a pointer, which is not treated as a pointer! Instead, its value is interpreted as a byte offset from the beginning of the buffer to the first attribute in the buffer. In this case, there is no additional data in either buffer before the first element, so we use a value of zero.



The `glVertexAttribPointer` function stores (in the VAO's state) a pointer to the buffer currently bound to the `GL_ARRAY_BUFFER` binding point. When another buffer is bound to that binding point, it does not change the value of the pointer.

The VAO stores all of the OpenGL state related to the relationship between buffer objects and the generic vertex attributes, as well as the information about the format of the data in the buffer objects. This allows us to quickly return all of this state when rendering. The VAO is an extremely important concept, but can be tricky to understand. It's important to remember that the VAO's state is primarily associated with the enabled attributes and their connection to buffer objects. It doesn't necessarily keep track of buffer bindings. For example, it doesn't remember what is bound to the `GL_ARRAY_BUFFER` binding point. We only bind to this point in order to set up the pointers via `glVertexAttribPointer`.

Once we have the VAO set up (a one-time operation), we can issue a draw command to render our image. In our render function, we clear the color buffer using `glClear`, bind to the vertex array object, and call `glDrawArrays` to draw our triangle. The function `glDrawArrays` initiates rendering of primitives by stepping through the buffers for each enabled attribute array, and passing the data down the pipeline to our vertex shader. The first argument is the render mode (in this case we are drawing triangles), the second is the starting index in the enabled arrays, and the third argument is the number of indices to be rendered (3 vertxes for a single triangle).

To summarize, we followed these steps:

1. Make sure to specify the generic vertex attribute indexes for each attribute in the vertex shader using the `layout` qualifier.
2. Create and populate the buffer objects for each attribute.
3. Create and define the vertex array object by calling `glVertexAttribPointer` while the appropriate buffer is bound.
4. When rendering, bind to the vertex array object and call `glDrawArrays`, or other appropriate rendering function (e.g. `glDrawElements`).

There's more...

In the following, we'll discuss some details, extensions, and alternatives to the previous technique.

Separate attribute format

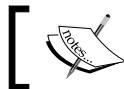
With OpenGL 4.3, we have an alternate (arguably better) way of specifying the vertex array object state (attribute format, enabled attributes, and buffers). In the previous example, the `glVertexAttribPointer` function does two important things. First, it indirectly specifies which buffer contains the data for the attribute, which is the buffer currently bound (at the time of the call) to `GL_ARRAY_BUFFER`. Secondly, it specifies the format of that data (type, offset, stride, and so on). It is arguably clearer to separate these two concerns into their own functions. This is exactly what has been implemented in OpenGL 4.3. For example, to implement the same functionality as in step 3 of the previous *How to do it...* section, we would use the following code:

```
glGenVertexArrays(1, &vaoHandle);
 glBindVertexArray(vaoHandle);
 glEnableVertexAttribArray(0);
 glEnableVertexAttribArray(1);

 glBindVertexBuffer(0, positionBufferHandle, 0, sizeof(GLfloat)*3);
 glBindVertexBuffer(1, colorBufferHandle, 0, sizeof(GLfloat)*3);
```

```
glVertexAttribFormat(0, 3, GL_FLOAT, GL_FALSE, 0);  
glVertexAttribBinding(0, 0);  
glVertexAttribFormat(1, 3, GL_FLOAT, GL_FALSE, 0);  
glVertexAttribBinding(1, 1);
```

The first four lines of the previous code are exactly the same as in the first example. We create and bind to the VAO, then enable attributes 0 and 1. Next, we bind our two buffers to two different indexes within the vertex buffer binding point using `glBindVertexBuffer`. Note that we're no longer using `GL_ARRAY_BUFFER` binding point. Instead, we now have a new binding point specifically for vertex buffers. This binding point has several indexes (usually from 0 - 15), so we can bind multiple buffers to this point. The first argument to `glBindVertexBuffer` specifies the index within the vertex buffer binding point. Here, we bind our position buffer to index 0 and our color buffer to index 1.



Note that the indexes within the vertex buffer binding point need not be the same as the attribute locations.



The other arguments to `glBindVertexBuffer` are as follows. The second argument is the buffer to be bound, the third is the offset from the beginning of the buffer to where the data begins, and the fourth is the stride, which is the distance between successive elements within the buffer. Unlike `glVertexAttribPointer`, we can't use a 0 value here for tightly packed data, because OpenGL can't determine the size of the data without more information, so we need to specify it explicitly here.

Next, we call `glVertexAttribFormat` to specify the format of the data for the attribute. Note that this time, this is decoupled from the buffer that stores the data. Instead, we're just specifying the format to expect for this attribute. The arguments are the same as the first four arguments to `glVertexAttribPointer`.

The function `glVertexAttribBinding` specifies the relationship between buffers that are bound to the vertex buffer binding point and attributes. The first argument is the attribute location, and the second is the index within the vertex buffer binding point. In this example, they are the same, but they need not be.

Also note that the buffer bindings of the vertex buffer binding point (specified by `glBindVertexBuffer`) are part of the VAO state, unlike the binding to `GL_ARRAY_BUFFER`, which is not.

This version is arguably more clear and easy to understand. It removes the confusing aspects of the "invisible" pointers that are managed in the VAO, and makes the relationship between attributes and buffers much more clear with `glVertexAttribBinding`. Additionally, it separates concerns that really need not be combined.

Fragment shader output

You may have noticed that I've neglected to say anything about the output variable `FragColor` in the fragment shader. This variable receives the final output color for each fragment (pixel). Like vertex input variables, this variable also needs to be associated with a proper location. Of course, we typically would like this to be linked to the back color buffer, which by default (in double buffered systems) is "color number" zero. (The relationship of the color numbers to render buffers can be changed by using `glDrawBuffers`). In this program, we are relying on the fact that the linker will automatically link our only fragment output variable to color number zero. To explicitly do so, we could (and probably should) have used a layout qualifier in the fragment shader:

```
layout (location = 0) out vec4 FragColor;
```

We are free to define multiple output variables for a fragment shader, thereby enabling us to render to multiple output buffers. This can be quite useful for specialized algorithms such as deferred rendering (see *Chapter 5, Image Processing and Screen Space Techniques*).

Specifying attribute indexes without using layout qualifiers

If you'd rather not clutter up your vertex shader code with the `layout` qualifiers (or you're using a version of OpenGL that doesn't support them), you can define the attribute indexes within the OpenGL program. We can do so by calling `glBindAttribLocation` just prior to linking the shader program. For example, we'd add the following code to the main OpenGL program just before the link step:

```
glBindAttribLocation(programHandle, 0, "VertexPosition");
glBindAttribLocation(programHandle, 1, "VertexColor");
```

This would indicate to the linker that `VertexPosition` should correspond to generic attribute index 0 and `VertexColor` to index 1.

Similarly, we can specify the color number for fragment shader output variables without using the layout qualifier. We do so by calling `glBindFragDataLocation` prior to linking the shader program:

```
glBindFragDataLocation(programHandle, 0, "FragColor");
```

This would tell the linker to bind the output variable `FragColor` to color number 0.

Using element arrays

It is often the case that we need to step through our vertex arrays in a non-linear fashion. In other words, we may want to "jump around" the data rather than just moving through it from beginning to end as we did in this example. For example, we might want to draw a cube where the vertex data consists of only eight positions (the corners of the cube). In order to draw the cube, we would need to draw 12 triangles (2 for each face), each of which consists of 3 vertices. All of the needed position data is in the original 8 positions, but to draw all the triangles, we'll need to jump around and use each position for at least three different triangles.

To jump around in our vertex arrays, we can make use of element arrays. The element array is another buffer that defines the indices used when stepping through the vertex arrays. For details on using element arrays, take a look at the function `glDrawElements` in the OpenGL documentation (<http://www.opengl.org/sdk/docs/man>).

Interleaved arrays

In this example, we used two buffers (one for color and one for position). Instead, we could have used just a single buffer and combined all of the data. In general, it is possible to combine the data for multiple attributes into a single buffer. The data for multiple attributes can be interleaved within an array, such that all of the data for a given vertex is grouped together within the buffer. Doing so just requires careful use of the `stride` argument to `glVertexAttribPointer` or `glBindVertexBuffer`. Take a look at the documentation for full details (<http://www.opengl.org/sdk/docs/man>).

The decision about when to use interleaved arrays and when to use separate arrays, is highly dependent on the situation. Interleaved arrays may bring better results due to the fact that data is accessed together and resides closer in memory (so-called locality of reference), resulting in better caching performance.

Getting a list of active vertex input attributes and locations

As covered in the previous recipe, the input variables within a vertex shader are linked to generic vertex attribute indices at the time the program is linked. If we need to specify the relationship, we can either use layout qualifiers within the shader, or we could call `glBindAttribLocation` before linking.

However, it may be preferable to let the linker create the mappings automatically and query for them after program linking is complete. In this recipe, we'll see a simple example that prints all the active attributes and their indices.

Getting ready

Start with an OpenGL program that compiles and links a shader pair. You could use the shaders from the previous recipe.

As in previous recipes, we'll assume that the handle to the shader program is stored in a variable named `programHandle`.

How to do it...

After linking and enabling the shader program, use the following code to display the list of active attributes:

1. Start by querying for the number of active attributes:

```
GLint numAttrs;  
glGetProgramInterfaceiv(programHandle, GL_PROGRAM_INPUT,  
    GL_ACTIVE_RESOURCES, &numAttrs);
```

2. Loop through each attribute and query for the length of the name, the type and the attribute location, and print the results to standard out:

```
GLenum properties[] = {GL_NAME_LENGTH, GL_TYPE,  
    GL_LOCATION};  
  
printf("Active attributes:\n");  
for( int i = 0; i < numAttrs; ++i ) {  
    GLint results[3];  
    glGetProgramResourceiv(programHandle, GL_PROGRAM_INPUT,  
        i, 3, properties, 3, NULL, results);  
  
    GLint nameBufSize = results[0] + 1;  
    char * name = new char[nameBufSize];  
    glGetProgramResourceName(programHandle,  
        GL_PROGRAM_INPUT, i, nameBufSize, NULL, name);  
    printf("%-5d %s (%s)\n", results[2],  
        name, getTypeString(results[1]));  
    delete [] name;  
}
```

How it works...

In step 1, we query for the number of active attributes, by calling `glGetProgramInterfaceiv`. The first argument is the handle to the program object, and the second (`GL_PROGRAM_INPUT`) indicates that we are querying for information about the program input variables (the vertex attributes). The third argument (`GL_ACTIVE_RESOURCES`) indicates that we want the number of active resources. The result is stored in the location pointed to by the last argument `numAttrs`.

Now that we have the number of attributes, we query for information about each one. The indices of the attributes run from 0 to `numAttribs - 1`. We loop over those indices and for each we call `glGetProgramResourceiv` to get the length of the name, the type and the location. We specify what information we would like to receive by means of an array of `GLenum` values called `properties`. The first argument is the handle to the program object, the second is the resource that we are querying (`GL_PROGRAM_INPUT`). The third is the index of the attribute, the fourth is the number of values in the `properties` array, which is the fifth argument. The `properties` array contains `GLenum`s, which specify the specific properties we would like to receive. In this example, the array contains: `GL_NAME_LENGTH`, `GL_TYPE`, and `GL_LOCATION`, which indicates that we want the length of the attribute's name, the data type of the attribute and its location. The sixth argument is the size of the buffer that will receive the results; the seventh argument is a pointer to an integer that would receive the number of results that were written. If that argument is `NULL`, then no information is provided. Finally, the last argument is a pointer to a `GLint` array that will receive the results. Each item in the `properties` array corresponds to the same index in the `results` array.

Next, we retrieve the name of the attribute by allocating a buffer to store the name and calling `glGetProgramResourceName`. The `results` array contains the length of the name in the first element, so we allocate an array of that size with an extra character just for good measure. The OpenGL documentation says that the size returned from `glGetProgramResourceiv` includes the null terminator, but it doesn't hurt to make sure by making a bit of additional space. In my tests, I've found this to be necessary on the latest NVIDIA drivers.

Finally, we get the name by calling `glGetProgramResourceName`, and then print the information to the screen. We print the attribute's location, name and type. The location is available in the third element of the `results` array, and the type is in the second. Note the use of the function `getTypeString`. This is a simple custom function that just returns a string representation of the data type. The data type is represented by one of the OpenGL defined constants `GL_FLOAT`, `GL_FLOAT_VEC2`, `GL_FLOAT_VEC3`, and so on. The `getTypeString` function consists of just one big switch statement returning a human-readable string corresponding to the value of the parameter (see the source code for `glslprogram.cpp` in the example code for this book).

The output of the previous code looks like this when it is run on the shaders from the previous recipes:

```
Active attributes:  
1  VertexColor (vec3)  
0  VertexPosition (vec3)
```

There's more...

It should be noted that in order for a vertex shader input variable to be considered active, it must be used within the vertex shader. In other words, a variable is considered active if it is determined by the GLSL linker that it may be accessed during program execution. If a variable is declared within a shader, but not used, the previous code will not display the variable because it is not considered active and effectively ignored by OpenGL.

The previous code is only valid for OpenGL 4.3 and later. Alternatively, you can achieve similar results with the functions `glGetProgramiv`, `glGetActiveAttrib` and `glGetAttribLocation`.

See also

- ▶ The [Compiling a shader](#) recipe
- ▶ The [Linking a shader program](#) recipe
- ▶ The [Sending data to a shader using vertex attributes and vertex buffer objects](#) recipe

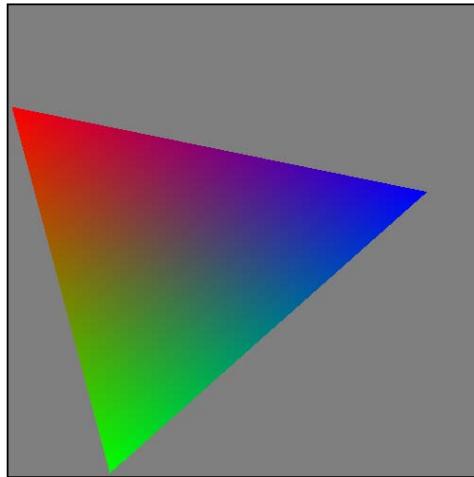
Sending data to a shader using uniform variables

Vertex attributes provide one avenue for providing input to shaders; a second technique is uniform variables. Uniform variables are intended to be used for data that may change relatively infrequently compared to per-vertex attributes. In fact, it is simply not possible to set per-vertex attributes with uniform variables. For example, uniform variables are well suited for the matrices used for modeling, viewing, and projective transformations.

Within a shader, uniform variables are read-only. Their values can only be changed from outside the shader, via the OpenGL API. However, they can be initialized within the shader by assigning to a constant value along with the declaration.

Uniform variables can appear in any shader within a shader program, and are always used as input variables. They can be declared in one or more shaders within a program, but if a variable with a given name is declared in more than one shader, its type must be the same in all shaders. In other words, the uniform variables are held in a shared uniform namespace for the entire shader program.

In this recipe, we'll draw the same triangle as in previous recipes in this chapter, however, this time, we'll rotate the triangle using a uniform matrix variable.



Getting ready

We'll use the following vertex shader:

```
#version 430

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;

out vec3 Color;

uniform mat4 RotationMatrix;

void main()
{
    Color = VertexColor;
    gl_Position = RotationMatrix * vec4(VertexPosition, 1.0);
}
```

Note the variable `RotationMatrix` is declared using the `uniform` qualifier. We'll provide the data for this variable via the OpenGL program. The `RotationMatrix` is also used to transform `VertexPosition` before assigning it to the default output position variable `gl_Position`.

We'll use the same fragment shader as in previous recipes:

```
#version 430

in vec3 Color;

layout (location = 0) out vec4 FragColor;

void main() {
    FragColor = vec4(Color, 1.0);
}
```

Within the main OpenGL code, we determine the rotation matrix and send it to the shader's uniform variable. To create our rotation matrix, we'll use the GLM library (see the *Using the GLM for mathematics* recipe in this chapter). Within the main OpenGL code, add the following include statements:

```
#include <glm/glm.hpp>
using glm::mat4;
using glm::vec3;

#include <glm/gtc/matrix_transform.hpp>
```

We'll also assume that code has been written to compile and link the shaders, and to create the vertex array object for the color triangle. We'll assume that the handle to the vertex array object is `vaoHandle`, and the handle to the program object is `programHandle`.

How to do it...

Within the render method, use the following code:

```
glClear(GL_COLOR_BUFFER_BIT);

mat4 rotationMatrix = glm::rotate(mat4(1.0f), angle,
    vec3(0.0f,0.0f,1.0f));

GLint location = glGetUniformLocation(programHandle, "RotationMatrix");

if( location >= 0 )
{
    glUniformMatrix4fv(location, 1, GL_FALSE,
        &rotationMatrix[0][0]);
}

 glBindVertexArray(vaoHandle);
 glDrawArrays(GL_TRIANGLES, 0, 3 );
```

How it works...

The steps involved with setting the value of a uniform variable include finding the location of the variable, then assigning a value to that location using one of the `glUniform` functions.

In this example, we start by clearing the color buffer, then creating a rotation matrix using GLM. Next, we query for the location of the uniform variable by calling `glGetUniformLocation`. This function takes the handle to the shader program object, and the name of the uniform variable and returns its location. If the uniform variable is not an active uniform variable, the function returns -1.

We then assign a value to the uniform variable's location using `glUniformMatrix4fv`. The first argument is the uniform variable's location. The second is the number of matrices that are being assigned (note that the uniform variable could be an array). The third is a Boolean value indicating whether or not the matrix should be transposed when loaded into the uniform variable. With GLM matrices, a transpose is not required, so we use `GL_FALSE` here. If you were implementing the matrix using an array, and the data was in row-major order, you might need to use `GL_TRUE` for this argument. The last argument is a pointer to the data for the uniform variable.

There's more...

Of course uniform variables can be any valid GLSL type including complex types such as arrays or structures. OpenGL provides a `glUniform` function with the usual suffixes, appropriate for each type. For example, to assign to a variable of type `vec3`, one would use `glUniform3f` or `glUniform3fv`.

For arrays, one can use the functions ending in "v" to initialize multiple values within the array. Note that if it is desired, one can query for the location of a particular element of the uniform array using the `[]` operator. For example, to query for the location of the second element of `MyArray`:

```
GLuint location =
    glGetUniformLocation( programHandle, "MyArray[1]" );
```

For structures, the members of the structure must be initialized individually. As with arrays, one can query for the location of a member of a structure using something like the following:

```
GLuint location =
    glGetUniformLocation( programHandle, "MyMatrices.Rotation" );
```

Where the structure variable is `MyMatrices` and the member of the structure is `Rotation`.

See also

- ▶ The [Compiling a shader](#) recipe
- ▶ The [Linking a shader program](#) recipe
- ▶ The [Sending data to a shader using vertex attributes and vertex buffer objects](#) recipe

Getting a list of active uniform variables

While it is a simple process to query for the location of an individual uniform variable, there may be instances where it can be useful to generate a list of all active uniform variables. For example, one might choose to create a set of variables to store the location of each uniform and assign their values after the program is linked. This would avoid the need to query for uniform locations when setting the value of the uniform variables, creating slightly more efficient code.

The process for listing uniform variables is very similar to the process for listing attributes (see the [Getting a list of active vertex input attributes and locations](#) recipe), so this recipe will refer the reader back to the previous recipe for detailed explanation.

Getting ready

Start with a basic OpenGL program that compiles and links a shader program. In the following, we'll assume that the handle to the program is in a variable named `programHandle`.

How to do it...

After linking and enabling the shader program, use the following code to display the list of active uniforms:

1. Start by querying for the number of active uniform variables:

```
GLint numUniforms = 0;
glGetProgramInterfaceiv( handle, GL_UNIFORM,
    GL_ACTIVE_RESOURCES, &numUniforms);
```

2. Loop through each uniform index and query for the length of the name, the type, the location and the block index:

```
GLenum properties[] = {GL_NAME_LENGTH, GL_TYPE,
    GL_LOCATION, GL_BLOCK_INDEX};

printf("Active uniforms:\n");
for( int i = 0; i < numUniforms; ++i ) {
    GLint results[4];
    glGetProgramResourceiv(handle, GL_UNIFORM, i, 4,
        properties, 4, NULL, results);
```

```

if( results[3] != -1 )
    continue;           // Skip uniforms in blocks
GLint nameBufSize = results[0] + 1;
char * name = new char[nameBufSize];
glGetProgramResourceName(handle, GL_UNIFORM, i,
    nameBufSize, NULL, name);
printf("%-5d %s (%s)\n", results[2], name,
    getTypeString(results[1]));
delete [] name;
}

```

How it works...

The process is very similar to the process shown in the recipe *Getting a list of active vertex input attributes and locations*. I will focus on the main differences.

First and most obvious is that we use `GL_UNIFORM` instead of `GL_PROGRAM_INPUT` as the interface that we are querying in `glGetProgramResourceiv` and `glGetProgramInterfaceiv`. Second, we query for the block index (using `GL_BLOCK_INDEX` in the `properties` array). The reason for this is that some uniform variables are contained within a uniform block (see the recipe *Using uniform blocks and uniform buffer objects*). For this example, we only want information about uniforms that are not within blocks. The block index will be `-1` if the uniform variable is not within a block, so we skip any uniform variables that do not have a block index of `-1`.

Again, we use the `getTypeString` function to convert the type value into a human-readable string (see example code).

When this is run on the shader program from the previous recipe, we see the following output:

```

Active uniforms:
0      RotationMatrix (mat4)

```

There's more...

As with vertex attributes, a uniform variable is not considered active unless it is determined by the GLSL linker that it will be used within the shader.

The previous code is only valid for OpenGL 4.3 and later. Alternatively, you can achieve similar results using the functions `glGetProgramiv`, `glGetActiveUniform`, `glGetUniformLocation`, and `glGetActiveUniformName`.

See also

- ▶ The *Sending data to a shader using uniform variables* recipe

Using uniform blocks and uniform buffer objects

If your program involves multiple shader programs that use the same uniform variables, one has to manage the variables separately for each program. Uniform locations are generated when a program is linked, so the locations of the uniforms may change from one program to the next. The data for those uniforms may have to be regenerated and applied to the new locations.

Uniform blocks were designed to ease the sharing of uniform data between programs. With uniform blocks, one can create a buffer object for storing the values of all the uniform variables, and bind the buffer to the uniform block. When changing programs, the same buffer object need only be re-bound to the corresponding block in the new program.

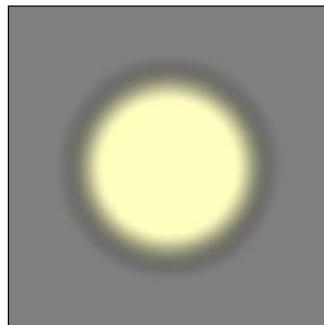
A uniform block is simply a group of uniform variables defined within a syntactical structure known as a uniform block. For example, in this recipe, we'll use the following uniform block:

```
uniform BlobSettings {  
    vec4 InnerColor;  
    vec4 OuterColor;  
    float RadiusInner;  
    float RadiusOuter;  
};
```

This defines a block with the name `BlobSettings` that contains four uniform variables. With this type of block definition, the variables within the block are still part of the global scope and do not need to be qualified with the block name.

The buffer object used to store the data for the uniforms is often referred to as a *uniform buffer object*. We'll see that a uniform buffer object is simply just a buffer object that is bound to a certain location.

For this recipe, we'll use a simple example to demonstrate the use of uniform buffer objects and uniform blocks. We'll draw a quad (two triangles) with texture coordinates, and use our fragment shader to fill the quad with a fuzzy circle. The circle is a solid color in the center, but at its edge, it gradually fades to the background color, as shown in the following image:



Getting ready

Start with an OpenGL program that draws two triangles to form a quad. Provide the position at vertex attribute location 0, and the texture coordinate (0 to 1 in each direction) at vertex attribute location 1 (see the [Sending data to a shader using vertex attributes and vertex buffer objects recipe](#)).

We'll use the following vertex shader:

```
#version 430

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexTexCoord;

out vec3 TexCoord;

void main()
{
    TexCoord = VertexTexCoord;
    gl_Position = vec4(VertexPosition, 1.0);
}
```

The fragment shader contains the uniform block, and is responsible for drawing our fuzzy circle:

```
#version 430

in vec3 TexCoord;
layout (location = 0) out vec4 FragColor;

layout (binding = 0) uniform BlobSettings {
    vec4 InnerColor;
    vec4 OuterColor;
    float RadiusInner;
    float RadiusOuter;
};

void main() {
    float dx = TexCoord.x - 0.5;
    float dy = TexCoord.y - 0.5;
    float dist = sqrt(dx * dx + dy * dy);
    FragColor =
        mix(InnerColor, OuterColor,
            smoothstep(RadiusInner, RadiusOuter, dist));
}
```

Note the uniform block named `BlobSettings`. The variables within this block define the parameters of our fuzzy circle. The variable `OuterColor` defines the color outside of the circle. `InnerColor` is the color inside of the circle. `RadiusInner` is the radius defining the part of the circle that is a solid color (inside the fuzzy edge), and the distance from the center of the circle to the inner edge of the fuzzy boundary. `RadiusOuter` is the outer edge of the fuzzy boundary of the circle (when the color is equal to `OuterColor`).

The code within the main function computes the distance of the texture coordinate to the center of the quad located at (0.5, 0.5). It then uses that distance to compute the color by using the `smoothstep` function. This function provides a value that smoothly varies between 0.0 and 1.0 when the value of the third argument is between the values of the first two arguments. Otherwise it returns 0.0 or 1.0 depending on whether `dist` is less than the first or greater than the second, respectively. The `mix` function is then used to linearly interpolate between `InnerColor` and `OuterColor` based on the value returned by the `smoothstep` function.

How to do it...

In the OpenGL program, after linking the shader program, use the following steps to assign data to the uniform block in the fragment shader:

1. Get the index of the uniform block using `glGetUniformBlockIndex`.

```
GLuint blockIndex = glGetUniformBlockIndex(programHandle,  
    "BlobSettings");
```
2. Allocate space for the buffer to contain the data for the uniform block. We get the size using `glGetActiveUniformBlockiv`:

```
GLint blockSize;  
glGetActiveUniformBlockiv(programHandle, blockIndex,  
    GL_UNIFORM_BLOCK_DATA_SIZE, &blockSize);
```
3. Query for the offset of each variable within the block. To do so, we first find the index of each variable within the block:

```
const GLchar *names[] = { "InnerColor", "OuterColor",  
    "RadiusInner", "RadiusOuter" };  
GLuint indices[4];  
glGetUniformIndices(programHandle, 4, names, indices);
```

```
GLint offset[4];  
glGetActiveUniformsiv(programHandle, 4, indices,  
    GL_UNIFORM_OFFSET, offset);
```

-
4. Place the data into the buffer at the appropriate offsets:

```
// Store data within the buffer at the appropriate offsets
GLfloat outerColor[] = {0.0f, 0.0f, 0.0f, 0.0f};
GLfloat innerColor[] = {1.0f, 1.0f, 0.75f, 1.0f};
GLfloat innerRadius = 0.25f, outerRadius = 0.45f;

memcpy(blockBuffer + offset[0], innerColor,
       4 * sizeof(GLfloat));
memcpy(blockBuffer + offset[1], outerColor,
       4 * sizeof(GLfloat));
memcpy(blockBuffer + offset[2], &innerRadius,
       sizeof(GLfloat));
memcpy(blockBuffer + offset[3], &outerRadius,
       sizeof(GLfloat));
```

5. Create the buffer object and copy the data into it:

```
GLuint uboHandle;
 glGenBuffers( 1, &uboHandle );
 glBindBuffer( GL_UNIFORM_BUFFER, uboHandle );
 glBufferData( GL_UNIFORM_BUFFER, blockSize, blockBuffer,
               GL_DYNAMIC_DRAW );
```

6. Bind the buffer object to the uniform buffer binding point at the index specified by the binding layout qualifier in the fragment shader (0):

```
glBindBufferBase(GL_UNIFORM_BUFFER, 0, uboHandle);
```

How it works...

Phew! This seems like a lot of work! However, the real advantage comes when using multiple programs where the same buffer object can be used for each program. Let's take a look at each step individually.

First we get the index of the uniform block by calling `glGetUniformBlockIndex`, then we query for the size of the block by calling `glGetActiveUniformBlockiv`. After getting the size, we allocate a temporary buffer named `blockBuffer` to hold the data for our block.

The layout of data within a uniform block is implementation dependent, and implementations may use different padding and/or byte alignment. So in order to accurately layout our data, we need to query for the offset of each variable within the block. This is done in two steps. First, we query for the index of each variable within the block by calling `glGetUniformIndices`. This accepts an array of variable names (third argument) and returns the indices of the variables in the array `indices` (fourth argument). Then we use the indices to query for the offsets by calling `glGetActiveUniformsiv`. When the fourth argument is `GL_UNIFORM_OFFSET`, this returns the offset of each variable in the array pointed to by the fifth argument. This function can also be used to query for the size and type, however, in this case we choose not to do so, to keep the code simple (albeit less general).

The next step involves filling our temporary buffer `blockBuffer` with the data for the uniforms at the appropriate offsets. Here we use the standard library function `memcpy` to accomplish this.

Now that the temporary buffer is populated with the data with the appropriate layout, we can create our buffer object and copy the data into the buffer object. We call `glGenBuffers` to generate a buffer handle, and then bind that buffer to the `GL_UNIFORM_BUFFER` binding point by calling `glBindBuffer`. The space is allocated within the buffer object and the data is copied when `glBufferData` is called. We use `GL_DYNAMIC_DRAW` as the usage hint here because uniform data may be changed somewhat often during rendering. Of course, this is entirely dependent on the situation.

Finally, we associate the buffer object with the uniform block by calling `glBindBufferBase`. This function binds to an index within a buffer binding point. Certain binding points are also so-called "indexed buffer targets". This means that the target is actually an array of targets, and `glBindBufferBase` allows us to bind to one index within the array. In this case, we bind it to the index that we specified in the layout qualifier in the fragment shader: `layout(binding = 0)` (see the "Getting ready..." section). These two indices must match.



You might be wondering why we use `glBindBuffer` and `glBindBufferBase` with `GL_UNIFORM_BUFFER`. Aren't these the same binding points used in two different contexts? The answer is that the `GL_UNIFORM_BUFFER` point can be used in each function with a slightly different meaning. With `glBindBuffer`, we bind to a point that can be used for filling or modifying a buffer, but can't be used as a source of data for the shader. When we use `glBindBufferBase`, we are binding to an index within a location that can be directly sourced by the shader. Granted, that's a bit confusing.

There's more...

If the data for a uniform block needs to be changed at some later time, one can call `glBufferSubData` to replace all or part of the data within the buffer. If you do so, don't forget to first bind the buffer to the generic binding point `GL_UNIFORM_BUFFER`.

Using an instance name with a uniform block

A uniform block can have an optional instance name. For example, with our `BlobSettings` block we could have used the instance name `Blob`, as shown here:

```
uniform BlobSettings {  
    vec4 InnerColor;  
    vec4 OuterColor;  
    float RadiusInner;  
    float RadiusOuter;  
} Blob;
```

In this case, the variables within the block are placed within a namespace qualified by the instance name. Therefore our shader code needs to refer to them prefixed with the instance name. For example:

```
FragColor =  
    mix( Blob.InnerColor, Blob.OuterColor,  
        smoothstep( Blob.RadiusInner, Blob.RadiusOuter, dist )  
    );
```

Additionally, we need to qualify the variable names (with the block name: `BlobSettings`) within the OpenGL code when querying for variable indices:

```
const GLchar *names[] = { "BlobSettings.InnerColor",  
    "BlobSettings.OuterColor", "BlobSettings.RadiusInner",  
    "BlobSettings.RadiusOuter" };  
GLuint indices[4];  
glGetUniformIndices(programHandle, 4, names, indices);
```

Using layout qualifiers with uniform blocks

Since the layout of the data within a uniform buffer object is implementation dependent, it required us to query for the variable offsets. However, one can avoid this by asking OpenGL to use the standard layout `std140`. This is accomplished by using a layout qualifier when declaring the uniform block. For example:

```
layout( std140 ) uniform BlobSettings {  
};
```

The `std140` layout is described in detail within the OpenGL specification document (available at <http://www.opengl.org>).

Other options for the layout qualifier that apply to uniform block layouts include `packed` and `shared`. The `packed` qualifier simply states that the implementation is free to optimize memory in whatever way it finds necessary (based on variable usage or other criteria). With the `packed` qualifier, we still need to query for the offsets of each variable. The `shared` qualifier guarantees that the layout will be consistent between multiple programs and program stages provided that the uniform block declaration does not change. If you are planning to use the same buffer object between multiple programs and/or program stages, it is a good idea to use the `shared` option.

There are two other layout qualifiers that are worth mentioning: `row_major` and `column_major`. These define the ordering of data within the matrix type variables within the uniform block.

One can use multiple (non-conflicting) qualifiers for a block. For example, to define a block with both the `row_major` and `shared` qualifiers, we would use the following syntax:

```
layout( row_major, shared ) uniform BlobSettings {  
};
```

See also

- ▶ The [Sending data to a shader using uniform variables](#) recipe

Getting debug messages

Prior to recent versions of OpenGL, the traditional way to get debug information was to call `glGetError`. Unfortunately, that is an exceedingly tedious method for debugging a program. The `glGetError` function returns an error code if an error has occurred at some point previous to the time the function was called. This means that if we're chasing down a bug, we essentially need to call `glGetError` after every function call to an OpenGL function, or do a binary search-like process where we call it before and after a block of code, and then move the two calls closer to each other until we determine the source of the error. What a pain!

Thankfully, as of OpenGL 4.3, we now have support for a more modern method for debugging. Now we can register a debug callback function that will be executed whenever an error occurs, or other informational message is generated. Not only that, but we can send our own custom messages to be handled by the same callback, and we can filter the messages using a variety of criteria.

Getting ready

Create an OpenGL program with a debug context. While it is not strictly necessary to acquire a debug context, we might not get messages that are as informative as when we are using a debug context. To create an OpenGL context using GLFW with debugging enabled, use the following function call prior to creating the window.

```
glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
```

An OpenGL debug context will have debug messages enabled by default. If, however, you need to enable debug messages explicitly, use the following call.

```
 glEnable(GL_DEBUG_OUTPUT);
```

How to do it...

Use the following steps:

1. Create a callback function to receive the debug messages. The function must conform to a specific prototype described in the OpenGL documentation. For this example, we'll use the following one:

```
void debugCallback(GLenum source, GLenum type, GLuint id,
                  GLenum severity, GLsizei length,
                  const GLchar * message, void * param) {

    // Convert GLenum parameters to strings

    printf("%s:%s[%s] (%d): %s\n", sourceStr, typeStr,
           severityStr, id, message);
}
```

2. Register our callback with OpenGL using `glDebugMessageCallback`:

```
glDebugMessageCallback( debugCallback, NULL );
```

3. Enable all messages, all sources, all levels, and all IDs:

```
glDebugMessageControl(GL_DONT_CARE, GL_DONT_CARE,
                      GL_DONT_CARE, 0, NULL, GL_TRUE);
```

How it works...

The callback function `debugCallback` has several parameters, the most important of which is the debug message itself (the sixth parameter, `message`). For this example, we simply print the message to standard output, but we could send it to a log file or some other destination.

The first four parameters to `debugCallback` describe the source, type, id number, and severity of the message. The id number is an unsigned integer specific to the message. The possible values for the source, type and severity parameters are described in the following tables.

The source parameter can have any of the following values:

Source	Generated By
<code>GL_DEBUG_SOURCE_API</code>	Calls to the OpenGL API
<code>GL_DEBUG_SOURCE_WINDOW_SYSTEM</code>	Calls to a window system API
<code>GL_DEBUG_SOURCE_THIRD_PARTY</code>	An application associated with OpenGL
<code>GL_DEBUG_SOURCE_APPLICATION</code>	This application itself.
<code>GL_DEBUG_SOURCE_OTHER</code>	Some other source

The type parameter can have any of the following values:

Type	Description
GL_DEBUG_TYPE_ERROR	An error from the OpenGL API.
GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR	Behavior that has been deprecated
GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR	Undefined behaviour
GL_DEBUG_TYPE_PORTABILITY	Some functionality is not portable.
GL_DEBUG_TYPE_PERFORMANCE	Possible performance issues
GL_DEBUG_TYPE_MARKER	An annotation
GL_DEBUG_TYPE_PUSH_GROUP	Messages related to debug group push.
GL_DEBUG_TYPE_POP_GROUP	Messages related to debug group pop.
GL_DEBUG_TYPE_OTHER	Other messages

The severity parameter can have the following values:

Severity	Meaning
GL_DEBUG_SEVERITY_HIGH	Errors or dangerous behaviour
GL_DEBUG_SEVERITY_MEDIUM	Major performance warnings, other warnings or use of deprecated functionality.
GL_DEBUG_SEVERITY_LOW	Redundant state changes, unimportant undefined behaviour.
GL_DEBUG_SEVERITY_NOTIFICATION	A notification, not an error or performance issue.

The length parameter is the length of the message string, excluding the null terminator. The last parameter `param` is a user-defined pointer. We can use this to point to some custom object that might be helpful to the callback function. For example, if we were logging the messages to a file, this could point to an object containing file I/O capabilities. This parameter can be set using the second parameter to `glDebugMessageCallback` (more on that in the following content).

Within `debugCallback` we convert each `GLenum` parameter into a string. Due to space constraints, I don't show all of that code here, but it can be found in the example code for this book. We then print all of the information to standard output.

The call to `glDebugMessageCallback` registers our callback function with the OpenGL debug system. The first parameter is a pointer to our callback function, and the second parameter (`NULL` in this example) can be a pointer to any object that we would like to pass into the callback. This pointer is passed as the last parameter with every call to `debugCallback`.

Finally, the call to `glDebugMessageControl` determines our message filters. This function can be used to selectively turn on or off any combination of message source, type, id, or severity. In this example, we turn everything on.

There's more...

OpenGL also provides support for stacks of named debug groups. Essentially what this means is that we can remember all of our debug message filter settings on a stack and return to them later after some changes have been made. This might be useful, for example, if there are sections of code where we have needs for filtering some kinds of messages and other sections where we want a different set of messages.

The functions involved are `glPushDebugGroup` and `glPopDebugGroup`. A call to `glPushDebugGroup` generates a debug message with type `GL_DEBUG_TYPE_PUSH_GROUP`, and retains the current state of our debug filters on a stack. We can then change our filters using `glDebugMessageControl`, and later return to the original state using `glPopDebugGroup`. Similarly, the function `glPopDebugGroup` generates a debug message with type `GL_DEBUG_TYPE_POP_GROUP`.

Building a C++ shader program class

If you are using C++, it can be very convenient to create classes to encapsulate some of the OpenGL objects. A prime example is the shader program object. In this recipe, we'll look at a design for a C++ class that can be used to manage a shader program.

Getting ready

There's not much to prepare for with this one, you just need a build environment that supports C++. Also, I'll assume that you are using GLM for matrix and vector support, if not just leave out the functions involving the GLM classes.

How to do it...

First, we'll use a custom exception class for errors that might occur during compilation or linking:

```
class GLSLProgramException : public std::runtime_error {
public:
    GLSLProgramException( const string & msg ) :
        std::runtime_error(msg) { }
};
```

We'll use an `enum` for the various shader types:

```
namespace GLSLShader {
    enum GLSLShaderType {
        VERTEX = GL_VERTEX_SHADER,
        FRAGMENT = GL_FRAGMENT_SHADER,
        GEOMETRY = GL_GEOMETRY_SHADER,
        TESS_CONTROL = GL_TESS_CONTROL_SHADER,
        TESS_EVALUATION = GL_TESS_EVALUATION_SHADER,
        COMPUTE = GL_COMPUTE_SHADER
    };
}
```

The program class itself has the following interface:

```
class GLSLProgram
{
private:
    int handle;
    bool linked;
    std::map<string, int> uniformLocations;

    int getUniformLocation(const char * name);

    // A few other helper functions

public:
    GLSLProgram();
    ~GLSLProgram();

    void compileShader( const char * filename )
        throw(GLSLProgramException);
    void compileShader( const char * filename,
        GLSLShader::GLSLShaderType type )
        throw(GLSLProgramException);
    void compileShader( const string & source,
        GLSLShader::GLSLShaderType type,
        const char * filename = NULL )
        throw(GLSLProgramException);
    void link()         throw(GLSLProgramException);
    void use()          throw(GLSLProgramException);
    void validate()    throw(GLSLProgramException);

    int     getHandle();
    bool    isLinked();
```

```
void    bindAttribLocation( GLuint location,
                           const char * name);
void    bindFragDataLocation( GLuint location,
                           const char * name );
void    setUniform(const char *name, float x, float y,
                  float z);
void    setUniform(const char *name, const vec3 & v);
void    setUniform(const char *name, const vec4 & v);
void    setUniform(const char *name, const mat4 & m);
void    setUniform(const char *name, const mat3 & m);
void    setUniform(const char *name, float val );
void    setUniform(const char *name, int val );
void    setUniform(const char *name, bool val );

void    printActiveUniforms();
void    printActiveAttribs();
void    printActiveUniformBlocks();
};
```

Code Download Tip



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.

Full source code for all of the recipes in this text is also available on GitHub at: <https://github.com/daw42/glslcookbook>.

The techniques involved in the implementation of these functions are covered in previous recipes in this chapter. Due to space limitations, I won't include the code here (it's available from this book's GitHub repository), but we'll discuss some of the design decisions in the next section.

How it works...

The state stored within a `GLSLProgram` object includes the handle to the OpenGL shader program object (`handle`), a Boolean variable indicating whether or not the program has been successfully linked (`linked`), and a map used to store `uniform` locations as they are discovered (`uniformLocations`).

The `compileShader` overloads will throw a `GLSLProgramException` if the compilation fails. The first version determines the type of shader based on the filename extension. In the second version, the caller provides the shader type, and the third version is used to compile a shader, taking the shader's source code from a `string`. The file name can be provided as a third argument in the case that the `string` was taken from a file, which is helpful for providing better error messages.

The `GLSLProgramException`'s error message will contain the contents of the shader log or program log when an error occurs.

The private function `getUniformLocation` is used by the `setUniform` functions to find the location of a uniform variable. It checks the map `uniformLocations` first, and if the location is not found, queries OpenGL for the location, and stores the result in the map before returning. The `fileExists` function is used by `compileShaderFromFile` to check for file existence.

The constructor simply initializes `linked` to `false` and `handle` to zero. The variable `handle` will be initialized by calling `glCreateProgram` when the first shader is compiled.

The `link` function simply attempts to link the program by calling `glLinkProgram`. It then checks the link status, and if successful, sets the variable `linked` to `true` and returns `true`. Otherwise, it gets the program log (by calling `glGetProgramInfoLog`), stores the result in a `GLSLProgramException` and throws it.

The `use` function simply calls `glUseProgram` if the program has already been successfully linked, otherwise it does nothing.

The functions `getHandle` and `isLinked` are simply "getter" functions that return the handle to the OpenGL program object and the value of the `linked` variable.

The functions `bindAttribLocation` and `bindFragDataLocation` are wrappers around `glBindAttribLocation` and `glBindFragDataLocation`. Note that these functions should only be called prior to linking the program.

The `setUniform` overloaded functions are straightforward wrappers around the appropriate `glUniform` functions. Each of them calls `getUniformLocation` to query for the variable's location before calling the `glUniform` function.

Finally, the `printActiveUniforms`, `printActiveUniformBlocks`, and `printActiveAttribs` functions are useful for debugging purposes. They simply display a list of the active uniforms/attributes to standard output.

The following is a simple example of the use of the `GLSLProgram` class:

```
GLSLProgram prog;

try {
    prog.compileShader("myshader.vert");
    prog.compileShader("myshader.frag");
    prog.link();
    prog.validate();
    prog.use();
} catch( GLSLProgramException &e ) {
    cerr << e.what() << endl;
    exit(EXIT_FAILURE);
}

prog.printActiveUniforms();
prog.printActiveAttribs();

prog.setUniform("ModelViewMatrix", matrix);
prog.setUniform("LightPosition", 1.0f, 1.0f, 1.0f);
```

See also

- ▶ For full source code, check out the GitHub site for this book: <http://github.com/daw42/glslcookbook>
- ▶ All of the recipes in this chapter!

2

The Basics of GLSL Shaders

In this chapter, we will cover:

- ▶ Implementing diffuse, per-vertex shading with a single point light source
- ▶ Implementing per-vertex ambient, diffuse, and specular (ADS) shading
- ▶ Using functions in shaders
- ▶ Implementing two-sided shading
- ▶ Implementing flat shading
- ▶ Using subroutines to select shader functionality
- ▶ Discarding fragments to create a perforated look

Introduction

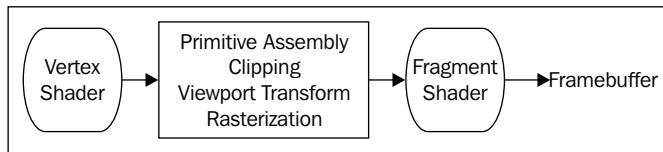
Shaders were first added into OpenGL in Version 2.0, introducing programmability into the formerly fixed-function OpenGL pipeline. Shaders give us the power to implement alternative rendering algorithms and a greater degree of flexibility in the implementation of those techniques. With shaders, we can run custom code directly on the GPU, providing us with the opportunity to leverage the high degree of parallelism available with modern GPUs.

Shaders are implemented using the **OpenGL Shading Language (GLSL)**. The GLSL is syntactically similar to C, which should make it easier for experienced OpenGL programmers to learn. Due to the nature of this text, I won't present a thorough introduction to GLSL here. Instead, if you're new to GLSL, reading through these recipes should help you to learn the language by example. If you are already comfortable with GLSL, but don't have experience with Version 4.x, you'll see how to implement these techniques utilizing the newer API. However, before we jump into GLSL programming, let's take a quick look at how vertex and fragment shaders fit within the OpenGL pipeline.

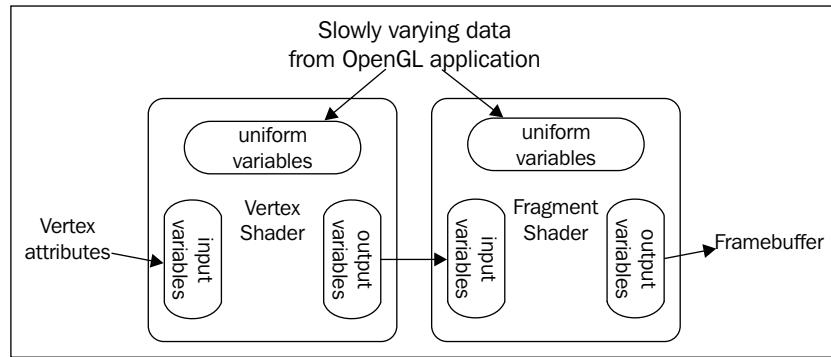
Vertex and fragment shaders

In OpenGL Version 4.3, there are six shader stages/types: vertex, geometry, tessellation control, tessellation evaluation, fragment, and compute. In this chapter we'll focus only on the vertex and fragment stages. In *Chapter 6, Using Geometry and Tessellation Shaders*, I'll provide some recipes for working with the geometry and tessellation shaders, and in *Chapter 10, Using Compute Shaders*, I'll focus specifically on compute shaders.

Shaders replace parts of the OpenGL pipeline. More specifically, they make those parts of the pipeline programmable. The following block diagram shows a simplified view of the OpenGL pipeline with only the vertex and fragment shaders installed:



Vertex data is sent down the pipeline and arrives at the vertex shader via shader input variables. The vertex shader's input variables correspond to the vertex attributes (refer to the *Sending data to a shader using vertex attributes and vertex buffer objects* recipe in *Chapter 1, Getting Started with GLSL*). In general, a shader receives its input via programmer-defined input variables, and the data for those variables comes either from the main OpenGL application or previous pipeline stages (other shaders). For example, a fragment shader's input variables might be fed from the output variables of the vertex shader. Data can also be provided to any shader stage using uniform variables (refer to the *Sending data to a shader using uniform variables* recipe, in *Chapter 1, Getting Started with GLSL*). These are used for information that changes less often than vertex attributes (for example, matrices, light position, and other settings). The following figure shows a simplified view of the relationships between input and output variables when there are two shaders active (vertex and fragment):



The vertex shader is executed once for each vertex, usually in parallel. The data corresponding to the position of the vertex must be transformed into clip coordinates and assigned to the output variable `gl_Position` before the vertex shader finishes execution. The vertex shader can send other information down the pipeline using shader output variables. For example, the vertex shader might also compute the color associated with the vertex. That color would be passed to later stages via an appropriate output variable.

Between the vertex and fragment shader, the vertices are assembled into primitives, clipping takes place, and the viewport transformation is applied (among other operations). The rasterization process then takes place and the polygon is filled (if necessary). The fragment shader is executed once for each fragment (pixel) of the polygon being rendered (typically in parallel). Data provided from the vertex shader is (by default) interpolated in a perspective correct manner, and provided to the fragment shader via shader input variables. The fragment shader determines the appropriate color for the pixel and sends it to the frame buffer using output variables. The depth information is handled automatically.

Replicating the old fixed functionality

Programmable shaders give us tremendous power and flexibility. However, in some cases we might just want to re-implement the basic shading techniques that were used in the default fixed-function pipeline, or perhaps use them as a basis for other shading techniques. Studying the basic shading algorithm of the old fixed-function pipeline can also be a good way to get started when learning about shader programming.

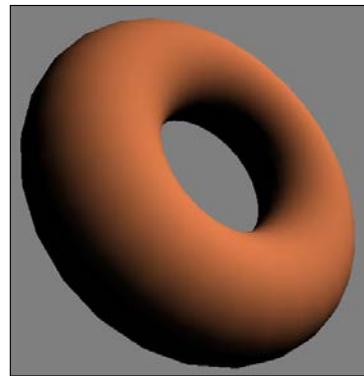
In this chapter, we'll look at the basic techniques for implementing shading similar to that of the old fixed-function pipeline. We'll cover the standard ambient, diffuse, and specular (ADS) shading algorithm, the implementation of two-sided rendering, and flat shading. Along the way, we'll also see some examples of other GLSL features such as functions, subroutines, and the `discard` keyword.

The algorithms presented within this chapter are largely unoptimized. I present them this way to avoid additional confusion for someone who is learning the techniques for the first time. We'll look at a few optimization techniques at the end of some recipes, and some more in the next chapter.

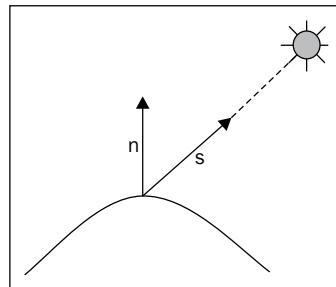
Implementing diffuse, per-vertex shading with a single point light source

One of the simplest shading techniques is to assume that the surface exhibits purely diffuse reflection. That is to say that the surface is one that appears to scatter light in all directions equally, regardless of direction. Incoming light strikes the surface and penetrates slightly before being re-radiated in all directions. Of course, the incoming light interacts with the surface before it is scattered, causing some wavelengths to be fully or partially absorbed and others to be scattered. A typical example of a diffuse surface is a surface that has been painted with a matte paint. The surface has a dull look with no shine at all.

The following screenshot shows a torus rendered with diffuse shading:



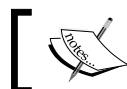
The mathematical model for diffuse reflection involves two vectors: the direction from the surface point to the light source (\mathbf{s}), and the normal vector at the surface point (\mathbf{n}). The vectors are represented in the following diagram:



The amount of incoming light (or radiance) that reaches the surface is partially dependent on the orientation of the surface with respect to the light source. The physics of the situation tells us that the amount of radiation that reaches a point on a surface is maximal when the light arrives along the direction of the normal vector, and zero when the light is perpendicular to the normal. In between, it is proportional to the cosine of the angle between the direction towards the light source and the normal vector. So, since the dot product is proportional to the cosine of the angle between two vectors, we can express the amount of radiation striking the surface as the product of the light intensity and the dot product of \mathbf{s} and \mathbf{n} .

$$L_d \mathbf{s} \cdot \mathbf{n}$$

Where L_d is the intensity of the light source, and the vectors \mathbf{s} and \mathbf{n} are assumed to be normalized.



The dot product of two unit vectors is equal to the cosine of the angle between them.



As stated previously, some of the incoming light is absorbed before it is re-emitted. We can model this interaction by using a reflection coefficient (K_d), which represents the fraction of the incoming light that is scattered. This is sometimes referred to as the **diffuse reflectivity**, or the diffuse reflection coefficient. The diffuse reflectivity becomes a scaling factor for the incoming radiation, so the intensity of the outgoing light can be expressed as follows:

$$L = K_d L_d \mathbf{s} \cdot \mathbf{n}$$

Because this model depends only on the direction towards the light source and the normal to the surface, not on the direction towards the viewer, we have a model that represents uniform (omnidirectional) scattering.

In this recipe, we'll evaluate this equation at each vertex in the vertex shader and interpolate the resulting color across the face.



In this and the following recipes, light intensities and material reflectivity coefficients are represented by 3-component (RGB) vectors. Therefore, the equations should be treated as component-wise operations, applied to each of the three components separately. Luckily, the GLSL will make this nearly transparent because the needed operators operate component-wise on vector variables.

Getting ready

Start with an OpenGL application that provides the vertex position in attribute location 0, and the vertex normal in attribute location 1 (refer to the *Sending data to a shader using vertex attributes and vertex buffer objects* recipe in *Chapter 1, Getting Started with GLSL*). The OpenGL application also should provide the standard transformation matrices (projection, modelview, and normal) via uniform variables.

The light position (in eye coordinates), K_d , and L_d should also be provided by the OpenGL application via uniform variables. Note that K_d and L_d are of type `vec3`. We can use `vec3` to store an RGB color as well as a vector or point.

How to do it...

To create a shader pair that implements diffuse shading, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

uniform vec4 LightPosition; // Light position in eye coords.
uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ld;           // Light source intensity

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;          // Projection * ModelView

void main()
{
    // Convert normal and position to eye coords
    vec3 tnorm = normalize( NormalMatrix * VertexNormal );
    vec4 eyeCoords = ModelViewMatrix *
                      vec4(VertexPosition,1.0));
    vec3 s = normalize(vec3(LightPosition - eyeCoords));

    // The diffuse shading equation
    LightIntensity = Ld * Kd * max( dot( s, tnorm ), 0.0 );

    // Convert position to clip coordinates and pass along
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering. See *Chapter 1, Getting Started with GLSL*, for details about compiling, linking, and installing shaders.

How it works...

The vertex shader does all of the work in this example. The diffuse reflection is computed in eye coordinates by first transforming the normal vector using the normal matrix, normalizing, and storing the result in `tnorm`. Note that the normalization here may not be necessary if your normal vectors are already normalized and the normal matrix does not do any scaling.

 The normal matrix is typically the inverse transpose of the upper-left 3×3 portion of the model-view matrix. We use the inverse transpose because normal vectors transform differently than the vertex position. For a more thorough discussion of the normal matrix, and the reasons why, see any introductory computer graphics textbook (A good choice would be *Computer Graphics with OpenGL* by Hearn and Baker). If your model-view matrix does not include any non-uniform scalings, then one can use the upper-left 3×3 of the model-view matrix in place of the normal matrix to transform your normal vectors. However, if your model-view matrix does include (uniform) scalings, you'll still need to (re)normalize your normal vectors after transforming them.

The next step converts the vertex position to eye (camera) coordinates by transforming it via the model-view matrix. Then we compute the direction towards the light source by subtracting the vertex position from the light position and storing the result in `s`.

Next, we compute the scattered light intensity using the equation described previously and store the result in the output variable `LightIntensity`. Note the use of the `max` function here. If the dot product is less than zero, then the angle between the normal vector and the light direction is greater than 90 degrees. This means that the incoming light is coming from inside the surface. Since such a situation is not physically possible (for a closed mesh), we use a value of 0.0. However, you may decide that you want to properly light both sides of your surface, in which case the normal vector needs to be reversed for those situations where the light is striking the back side of the surface (refer to the *Implementing two-sided shading* recipe in this chapter).

Finally, we convert the vertex position to clip coordinates by multiplying with the model-view projection matrix, (which is: `projection * view * model`) and store the result in the built-in output variable `gl_Position`.

```
gl_Position = MVP * vec4(VertexPosition, 1.0);
```

 The subsequent stage of the OpenGL pipeline expects that the vertex position will be provided in clip coordinates in the output variable `gl_Position`. This variable does not directly correspond to any input variable in the fragment shader, but is used by the OpenGL pipeline in the primitive assembly, clipping, and rasterization stages that follow the vertex shader. It is important that we always provide a valid value for this variable.

Since `LightIntensity` is an output variable from the vertex shader, its value is interpolated across the face and passed into the fragment shader. The fragment shader then simply assigns the value to the output fragment.

There's more...

Diffuse shading is a technique that models only a very limited range of surfaces. It is best used for surfaces that have a "matte" appearance. Additionally, with the technique used previously, the dark areas may look a bit too dark. In fact, those areas that are not directly illuminated are completely black. In real scenes, there is typically some light that has been reflected about the room that brightens these surfaces. In the following recipes, we'll look at ways to model more surface types, as well as provide some light for those dark parts of the surface.

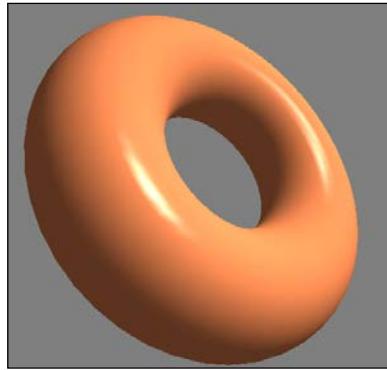
See also

- ▶ The [Sending data to a shader using uniform variables](#) recipe in *Chapter 1, Getting Started with GLSL*
- ▶ The [Compiling a shader](#) recipe in *Chapter 1, Getting Started with GLSL*
- ▶ The [Linking a shader program](#) recipe in *Chapter 1, Getting Started with GLSL*
- ▶ The [Sending data to a shader using vertex attributes and vertex buffer objects](#) recipe in *Chapter 1, Getting Started with GLSL*

Implementing per-vertex ambient, diffuse, and specular (ADS) shading

The OpenGL fixed function pipeline implemented a default shading technique which is very similar to the one presented here. It models the light-surface interaction as a combination of three components: ambient, diffuse, and specular. The **ambient** component is intended to model light that has been reflected so many times that it appears to be emanating uniformly from all directions. The **diffuse** component was discussed in the previous recipe, and represents omnidirectional reflection. The **specular** component models the shininess of the surface and represents reflection around a preferred direction. Combining these three components together can model a nice (but limited) variety of surface types. This shading model is also sometimes called the **Phong reflection model** (or **Phong shading model**), after *Bui Tuong Phong*.

An example of a torus rendered with the ADS shading model is shown in the following screenshot:



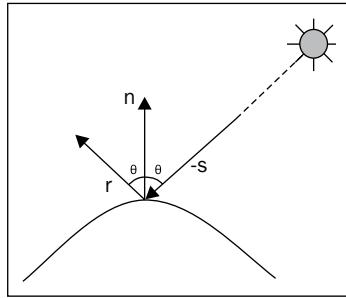
The ADS model is implemented as the sum of the three components: ambient, diffuse, and specular. The ambient component represents light that illuminates all surfaces equally and reflects equally in all directions. It is often used to help brighten some of the darker areas within a scene. Since it does not depend on the incoming or outgoing directions of the light, it can be modeled simply by multiplying the light source intensity (L_a) by the surface reflectivity (K_a).

$$I_a = L_a K_a$$

The diffuse component models a rough surface that scatters light in all directions (refer to the *Implementing diffuse, per-vertex shading with a single point light source* recipe in this chapter). The intensity of the outgoing light depends on the angle between the surface normal and the vector towards the light source.

$$I_d = L_d K_d (\mathbf{s} \cdot \mathbf{n})$$

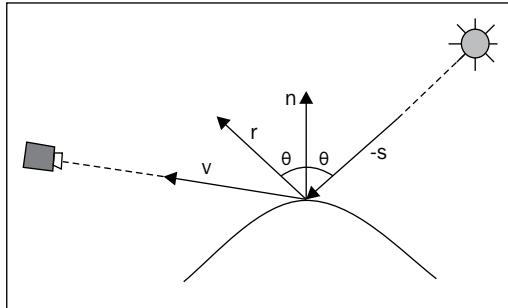
The specular component is used for modeling the shininess of a surface. When a surface has a glossy shine to it, the light is reflected off of the surface in a mirror-like fashion. The reflected light is strongest in the direction of perfect (mirror-like) reflection. The physics of the situation tells us that for perfect reflection, the angle of incidence is the same as the angle of reflection and that the vectors are coplanar with the surface normal, as shown in the following diagram:



In the preceding diagram, \mathbf{r} represents the vector of pure-reflection corresponding to the incoming light vector ($-\mathbf{s}$), and \mathbf{n} is the surface normal. We can compute \mathbf{r} by using the following equation:

$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$$

To model specular reflection, we need to compute the following (normalized) vectors: the direction towards the light source (\mathbf{s}), the vector of perfect reflection (\mathbf{r}), the vector towards the viewer (\mathbf{v}), and the surface normal (\mathbf{n}). These vectors are represented in the following diagram:



We would like the reflection to be maximal when the viewer is aligned with the vector \mathbf{r} , and to fall off quickly as the viewer moves further away from alignment with \mathbf{r} . This can be modeled using the cosine of the angle between \mathbf{v} and \mathbf{r} raised to some power (f).

$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$$

(Recall that the dot product is proportional to the cosine of the angle between the vectors involved.) The larger the power, the faster the value drops towards zero as the angle between \mathbf{v} and \mathbf{r} increases. Again, similar to the other components, we also introduce a specular light intensity term (\mathbf{L}_s) and reflectivity term (\mathbf{K}_s).

The specular component creates **specular highlights** (bright spots) that are typical of glossy surfaces. The larger the power of f in the equation, the smaller the specular highlight and the shinier the surface appears. The value for f is typically chosen to be somewhere between 1 and 200.

Putting all of this together, we have the following shading equation:

$$\begin{aligned} I &= I_a + I_d + I_s \\ &= L_a K_a + L_d K_d (\mathbf{s} \cdot \mathbf{n}) + L_s K_s (\mathbf{r} \cdot \mathbf{v})^f \end{aligned}$$

For more details about how this shading model was implemented in the fixed function pipeline, take a look at *Chapter 5, Image Processing and Screen Space Techniques*.

In the following code, we'll evaluate this equation in the vertex shader, and interpolate the color across the polygon.

Getting ready

In the OpenGL application, provide the vertex position in location 0 and the vertex normal in location 1. The light position and the other configurable terms for our lighting equation are uniform variables in the vertex shader and their values must be set from the OpenGL application.

How to do it...

To create a shader pair that implements ADS shading, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;        // Ambient light intensity
    vec3 Ld;        // Diffuse light intensity
    vec3 Ls;        // Specular light intensity
};
```

```
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};

uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyeCoords = ModelViewMatrix *
                      vec4(VertexPosition,1.0);
    vec3 s = normalize(vec3(Light.Position - eyeCoords));
    vec3 v = normalize(-eyeCoords.xyz);
    vec3 r = reflect( -s, tnorm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,tnorm) , 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Ls * Material.Ks *
               pow(max( dot(r,v), 0.0 ), Material.Shininess);

    LightIntensity = ambient + diffuse + spec;
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

The vertex shader computes the shading equation in eye coordinates. It begins by transforming the vertex normal into eye coordinates and normalizing, then storing the result in `tNorm`. The vertex position is then transformed into eye coordinates and stored in `eyeCoords`.

Next, we compute the normalized direction towards the light source (`s`). This is done by subtracting the vertex position in eye coordinates from the light position and normalizing the result.

The direction towards the viewer (`v`) is the negation of the position (normalized) because in eye coordinates the viewer is at the origin.

We compute the direction of pure reflection by calling the GLSL built-in function `reflect`, which reflects the first argument about the second. We don't need to normalize the result because the two vectors involved are already normalized.

The ambient component is computed and stored in the variable `ambient`. The dot product of `s` and `n` is computed next. As in the preceding recipe, we use the built-in function `max` to limit the range of values to between one and zero. The result is stored in the variable named `sDotN`, and is used to compute the diffuse component. The resulting value for the diffuse component is stored in the variable `diffuse`. Before computing the specular component, we check the value of `sDotN`. If `sDotN` is zero, then there is no light reaching the surface, so there is no point in computing the specular component, as its value must be zero. Otherwise, if `sDotN` is greater than zero, we compute the specular component using the equation presented earlier. Again, we use the built-in function `max` to limit the range of values of the dot product to between one and zero, and the function `pow` raises the dot product to the power of the Shininess exponent (corresponding to `f` in our lighting equation).



If we did not check `sDotN` before computing the specular component, it is possible that some specular highlights could appear on faces that are facing away from the light source. This is clearly a non-realistic and undesirable result. Some people solve this problem by multiplying the specular component by the diffuse component, which would decrease the specular component substantially and alter its color. The solution presented here avoids this, at the cost of a branch statement (the `if` statement). (Branch statements can have a significant impact on performance.)

The sum of the three components is then stored in the output variable `LightIntensity`. This value will be associated with the vertex and passed down the pipeline. Before reaching the fragment shader, its value will be interpolated in a perspective correct manner across the face of the polygon.

Finally, the vertex shader transforms the position into clip coordinates, and assigns the result to the built-in output variable `gl_Position` (refer to the *Implementing diffuse, per-vertex shading with a single point light source* recipe in this chapter).

The fragment shader simply applies the interpolated value of `LightIntensity` to the output fragment by storing it in the shader output variable `FragColor`.

There's more...

This version of the ADS (Ambient, Diffuse, and Specular) reflection model is by no means optimal. There are several improvements that could be made. For example, the computation of the vector of pure reflection can be avoided via the use of the so-called "halfway vector". This is discussed in the *Using the halfway vector for improved performance* recipe in Chapter 3, *Lighting, Shading, and Optimization*.

Using a non-local viewer

We can avoid the extra normalization needed to compute the vector towards the viewer (v), by using a so-called **non-local viewer**. Instead of computing the direction towards the origin, we simply use the constant vector $(0, 0, 1)$ for all vertices. This is similar to assuming that the viewer is located infinitely far away in the z direction. Of course, it is not accurate, but in practice the visual results are very similar, often visually indistinguishable, saving us normalization.

In the old fixed-function pipeline, the non-local viewer was the default, and could be adjusted (turned on or off) using the function `glLightModel`.

Per-vertex versus per-fragment

Since the shading equation is computed within the vertex shader, we refer to this as **per-vertex shading**. One of the disadvantages of this is that specular highlights can be warped or lost, due to the fact that the shading equation is not evaluated at each point across the face. For example, a specular highlight that should appear in the middle of a polygon might not appear at all when per-vertex shading is used, because of the fact that the shading equation is only computed at the vertices where the specular component is near zero. In the *Using per-fragment shading for improved realism* recipe of Chapter 3, *Lighting, Shading, and Optimization*, we'll look at the changes needed to move the shading computation into the fragment shader, producing more realistic results.

Directional lights

We can also avoid the need to compute a light direction (s), for each vertex if we assume a directional light. A **directional light source** is one that can be thought of as located infinitely far away in a given direction. Instead of computing the direction towards the source for each vertex, a constant vector is used, which represents the direction towards the remote light source. We'll look at an example of this in the *Shading with a directional light source* recipe of Chapter 3, *Lighting, Shading, and Optimization*.

Light attenuation with distance

You might think that this shading model is missing one important component. It doesn't take into account the effect of the distance to the light source. In fact, it is known that the intensity of radiation from a source falls off in proportion to the inverse square of the distance from the source. So why not include this in our model?

It would be fairly simple to do so, however, the visual results are often less than appealing. It tends to exaggerate the distance effects and create unrealistic looking images. Remember, our equation is just an approximation of the physics involved and is not a truly realistic model, so it is not surprising that adding a term based on a strict physical law produces unrealistic results.

In the OpenGL fixed-function pipeline, it was possible to turn on distance attenuation using the `glLight` function. If desired, it would be straightforward to add a few uniform variables to our shader to produce the same effect.

See also

- ▶ The *Shading with a directional light source* recipe in Chapter 3, *Lighting, Shading, and Optimization*
- ▶ The *Using per-fragment shading for improved realism* recipe in Chapter 3, *Lighting, Shading, and Optimization*
- ▶ The *Using the halfway vector for improved performance* recipe in Chapter 3, *Lighting, Shading, and Optimization*

Using functions in shaders

The GLSL supports functions that are syntactically similar to C functions. However, the calling conventions are somewhat different. In the following example, we'll revisit the ADS shader using functions to help provide abstractions for the major steps.

Getting ready

As with previous recipes, provide the vertex position at attribute location 0 and the vertex normal at attribute location 1. Uniform variables for all of the ADS coefficients should be set from the OpenGL side, as well as the light position and the standard matrices.

How to do it...

To implement ADS shading using functions, use the following code:

1. Use the following vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;        // Ambient light intensity
    vec3 Ld;        // Diffuse light intensity
    vec3 Ls;        // Specular light intensity
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    vec3 s = normalize(vec3(Light.Position - position));
    vec3 v = normalize(-position.xyz);
    vec3 r = reflect( -s, norm );
    vec3 ambient = Light.La * Material.Ka;
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Ld * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
```

```
if( sDotN > 0.0 )
    spec = Light.Ls * Material.Ks *
        pow( max( dot(r,v), 0.0 ), Material.Shininess );

    return ambient + diffuse + spec;
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);

    // Evaluate the lighting equation.
    LightIntensity = phongModel( eyePosition, eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following fragment shader:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

In GLSL functions, the evaluation strategy is "call by value-return" (also called "call by copy-restore" or "call by value-result"). Parameter variables can be qualified with `in`, `out`, or `inout`. Arguments corresponding to input parameters (those qualified with `in` or `inout`) are copied into the parameter variable at call time, and output parameters (those qualified with `out` or `inout`) are copied back to the corresponding argument before the function returns. If a parameter variable does not have any of the three qualifiers, the default qualifier is `in`.

We've created two functions in the vertex shader. The first, named `getEyeSpace`, transforms the vertex position and vertex normal into eye space, and returns them via output parameters. In the `main` function, we create two uninitialized variables (`eyeNorm` and `eyePosition`) to store the results, and then call the function with the variables as the function's arguments. The function stores the results into the parameter variables (`norm` and `position`) which are copied into the arguments before the function returns.

The second function, `phongModel`, uses only input parameters. The function receives the eye-space position and normal, and computes the result of the ADS shading equation. The result is returned by the function and stored in the shader output variable `LightIntensity`.

There's more...

Since it makes no sense to read from an output parameter variable, output parameters should only be written to within the function. Their value is undefined.

Within a function, writing to an input-only parameter (qualified with `in`) is allowed. The function's copy of the argument is modified, and changes are not reflected in the argument.

The `const` qualifier

The additional qualifier `const` can be used with input-only parameters (not with `out` or `inout`). This qualifier makes the input parameter read-only, so it cannot be written to within the function.

Function overloading

Functions can be overloaded by creating multiple functions with the same name, but with different number and/or type of parameters. As with many languages, two overloaded functions may not differ in return type only.

Passing arrays or structures to a function

It should be noted that when passing arrays or structures to functions, they are passed by value. If a large array or structure is passed, it can incur a large copy operation which may not be desired. It would be a better choice to declare these variables in the global scope.

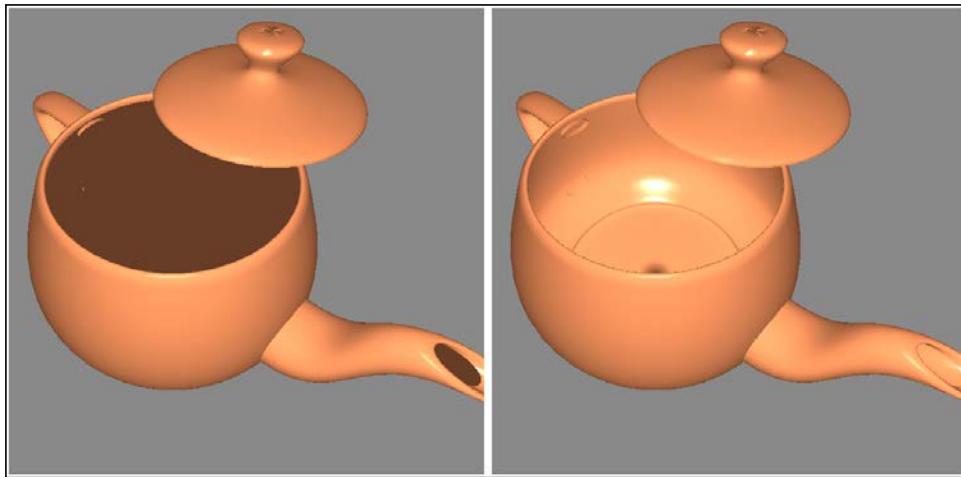
See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe

Implementing two-sided shading

When rendering a mesh that is completely closed, the back faces of polygons are hidden. However, if a mesh contains holes, it might be the case that the back faces would become visible. In this case, the polygons may be shaded incorrectly due to the fact that the normal vector is pointing in the wrong direction. To properly shade those back faces, one needs to invert the normal vector and compute the lighting equations based on the inverted normal.

The following screenshot shows a teapot with the lid removed. On the left, the ADS lighting model is used. On the right, the ADS model is augmented with the two-sided rendering technique discussed in this recipe.



In this recipe, we'll look at an example that uses the ADS model discussed in the previous recipes, augmented with the ability to correctly shade back faces.

Getting ready

The vertex position should be provided in attribute location 0 and the vertex normal in attribute location 1. As in previous examples, the lighting parameters must be provided to the shader via uniform variables.

How to do it...

To implement a shader pair that uses the ADS shading model with two-sided lighting, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 FrontColor;
out vec3 BackColor;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;        // Ambient light intensity
    vec3 Ld;        // Diffuse light intensity
    vec3 Ls;        // Specular light intensity
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 phongModel( vec4 position, vec3 normal ) {
    // The ADS shading calculations go here ("Implementing
    // per-vertex ambient, diffuse, and specular (ADS)
    // shading")
    ...
}

void main()
{
    vec3 tnorm = normalize( NormalMatrix * VertexNormal );
    vec4 eyeCoords = ModelViewMatrix *
                     vec4(VertexPosition,1.0);
```

```

FrontColor = phongModel( eyeCoords, tnorm );
BackColor = phongModel( eyeCoords, -tnorm );

gl_Position = MVP * vec4(VertexPosition, 1.0);
}

```

2. Use the following for the fragment shader:

```

in vec3 FrontColor;
in vec3 BackColor;

layout( location = 0 ) out vec4 FragColor;

void main() {
    if( gl_FrontFacing ) {
        FragColor = vec4(FrontColor, 1.0);
    } else {
        FragColor = vec4(BackColor, 1.0);
    }
}

```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

In the vertex shader, we compute the lighting equation using both the vertex normal and the inverted version, and pass each resultant color to the fragment shader. The fragment shader chooses and applies the appropriate color depending on the orientation of the face.

The vertex shader is a slightly modified version of the vertex shader presented in the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe of this chapter. The evaluation of the shading model is placed within a function named `phongModel`. The function is called twice, first using the normal vector (transformed into eye coordinates), and second using the inverted normal vector. The combined results are stored in `FrontColor` and `BackColor`, respectively.



Note that there are a few aspects of the shading model that are independent of the orientation of the normal vector (such as the ambient component). One could optimize this code by rewriting it so that the redundant calculations are only done once. However, in this recipe we compute the entire shading model twice in the interest of making things clear and readable.

In the fragment shader, we determine which color to apply based on the value of the built-in variable `gl_FrontFacing`. This is a `bool` value that indicates whether the fragment is part of a front or back facing polygon. Note that this determination is based on the **winding** of the polygon, and not the normal vector. (A polygon is said to have counter-clockwise winding if the vertices are specified in counter-clockwise order as viewed from the front side of the polygon.) By default when rendering, if the order of the vertices appear on the screen in a counter-clockwise order, it indicates a front facing polygon, however, we can change this by calling `glFrontFace` from the OpenGL program.

There's more...

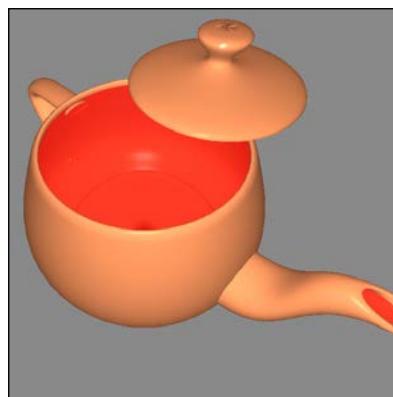
In the vertex shader we determine the front side of the polygon by the direction of the normal vector, and in the fragment shader, the determination is based on the polygon's winding. For this to work properly, the normal vector must be defined appropriately for the face determined by the setting of `glFrontFace`.

Using two-sided rendering for debugging

It can sometimes be useful to visually determine which faces are front facing and which are back facing. For example, when working with arbitrary meshes, polygons may not be specified using the appropriate winding. As another example, when developing a mesh procedurally, it can sometimes be helpful to determine which faces are oriented in the proper direction in order to help with debugging. We can easily tweak our fragment shader to help us solve these kinds of problems by mixing a solid color with all back (or front) faces. For example, we could change the `else` clause within our fragment shader to the following:

```
FragColor = mix( vec4(BackColor,1.0),  
                 vec4(1.0,0.0,0.0,1.0), 0.7 );
```

This would mix a solid red color with all back faces, helping them to stand out, as shown in the following screenshot. In the screenshot, back faces are mixed with 70 percent red as shown in the preceding code.



See also

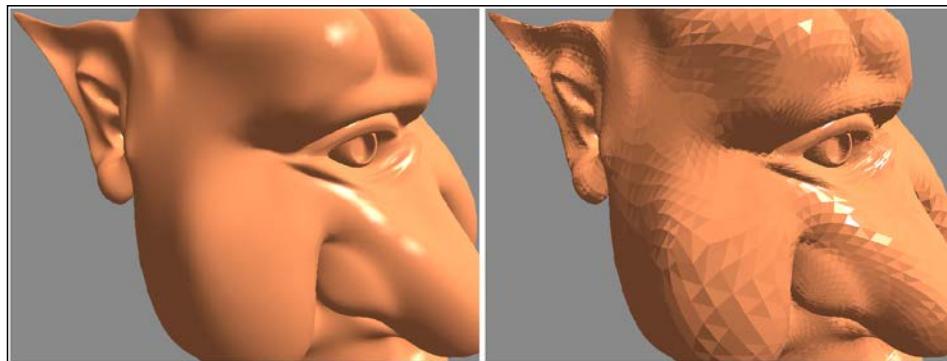
- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe

Implementing flat shading

Per-vertex shading involves computation of the shading model at each vertex and associating the result (a color) with that vertex. The colors are then interpolated across the face of the polygon to produce a smooth shading effect. This is also referred to as **Gouraud shading**. In earlier versions of OpenGL, this per-vertex shading with color interpolation was the default shading technique.

It is sometimes desirable to use a single color for each polygon so that there is no variation of color across the face of the polygon, causing each polygon to have a flat appearance. This can be useful in situations where the shape of the object warrants such a technique, perhaps because the faces really are intended to look flat, or to help visualize the locations of the polygons in a complex mesh. Using a single color for each polygon is commonly called **flat shading**.

The following screenshot shows a mesh rendered with the ADS shading model. On the left, Gouraud shading is used. On the right, flat shading is used.



In earlier versions of OpenGL, flat shading was enabled by calling the function `glShadeModel` with the argument `GL_FLAT`. In which case, the computed color of the last vertex of each polygon was used across the entire face.

In OpenGL 4, flat shading is facilitated by the interpolation qualifiers available for shader input/output variables.

How to do it...

To modify the ADS shading model to implement flat shading, use the following steps:

1. Use the same vertex shader as in the ADS example provided earlier. Change the output variable LightIntensity as follows:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

flat out vec3 LightIntensity;

// the rest is identical to the ADS shader...
```

2. Use the following code for the fragment shader:

```
flat in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

Flat shading is enabled by qualifying the vertex output variable (and its corresponding fragment input variable) with the `flat` qualifier. This qualifier indicates that no interpolation of the value is to be done before it reaches the fragment shader. The value presented to the fragment shader will be the one corresponding to the result of the invocation of the vertex shader for either the first or last vertex of the polygon. This vertex is called the **provoking vertex**, and can be configured using the OpenGL function `glProvokingVertex`. For example, the call:

```
glProvokingVertex(GL_FIRST_VERTEX_CONVENTION);
```

This indicates that the first vertex should be used as the value for the flat shaded variable. The argument `GL_LAST_VERTEX_CONVENTION` indicates that the last vertex should be used.

See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe

Using subroutines to select shader functionality

In GLSL, a subroutine is a mechanism for binding a function call to one of a set of possible function definitions based on the value of a variable. In many ways it is similar to function pointers in C. A uniform variable serves as the pointer and is used to invoke the function. The value of this variable can be set from the OpenGL side, thereby binding it to one of a few possible definitions. The subroutine's function definitions need not have the same name, but must have the same number and type of parameters and the same return type.

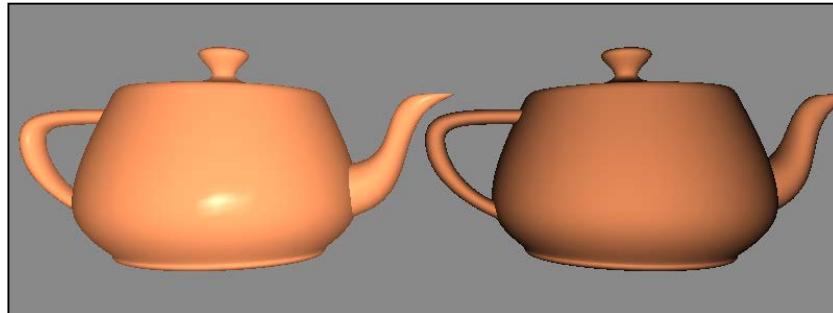
Subroutines therefore provide a way to select alternate implementations at runtime without swapping shader programs and/or recompiling, or using the `if` statements along with a uniform variable. For example, a single shader could be written to provide several shading algorithms intended for use on different objects within the scene. When rendering the scene, rather than swapping shader programs (or using a conditional statement), we can simply change the subroutine's uniform variable to choose the appropriate shading algorithm as each object is rendered.



Since performance is crucial in shader programs, avoiding a conditional statement or a shader swap can be very valuable. With subroutines, we can implement the functionality of a conditional statement or shader swap without the computational overhead.

In this example, we'll demonstrate the use of subroutines by rendering a teapot twice. The first teapot will be rendered with the full ADS shading model described earlier. The second teapot will be rendered with diffuse shading only. A subroutine uniform will be used to choose between the two shading techniques.

In the following screenshot, we see an example of a rendering that was created using subroutines. The teapot on the left is rendered with the full ADS shading model, and the teapot on the right is rendered with diffuse shading only. A subroutine is used to switch between shader functionality.



Getting ready

As with previous recipes, provide the vertex position at attribute location 0 and the vertex normal at attribute location 1. Uniform variables for all of the ADS coefficients should be set from the OpenGL side, as well as the light position and the standard matrices.

We'll assume that, in the OpenGL application, the variable `programHandle` contains the handle to the shader program object.

How to do it...

To create a shader program that uses a subroutine to switch between pure-diffuse and ADS shading, use the following steps:

1. Use the following code for the vertex shader:

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
subroutine uniform shadeModelType shadeModel;

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;        // Ambient light intensity
    vec3 Ld;        // Diffuse light intensity
    vec3 Ls;        // Specular light intensity
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};
uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
```

```
void getEyeSpace( out vec3 norm, out vec4 position )
{
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    // The ADS shading calculations go here (see: "Using
    // functions in shaders," and "Implementing
    // per-vertex ambient, diffuse, and specular (ADS)
    // shading")
    ...
}

subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(Light.Position - position) );
    return
        Light.Ld * Material.Kd * max( dot(s, norm), 0.0 );
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);

    // Evaluate the shading equation, calling one of
    // the functions: diffuseOnly or phongModel.
    LightIntensity = shadeModel(eyePosition, eyeNorm);

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 LightIntensity;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(LightIntensity, 1.0);
}
```

3. In the OpenGL application, compile and link the previous shaders into a shader program, and install the program into the OpenGL pipeline.
4. Within the render function of the OpenGL application, use the following code:

```
GLuint adsIndex =
    glGetSubroutineIndex(programHandle,
        GL_VERTEX_SHADER, "phongModel") ;

GLuint diffuseIndex =
    glGetSubroutineIndex(programHandle,
        GL_VERTEX_SHADER, "diffuseOnly") ;

glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &adsIndex) ;
... // Render the left teapot

glUniformSubroutinesuiv( GL_VERTEX_SHADER, 1, &diffuseIndex) ;
... // Render the right teapot
```

How it works...

In this example, the subroutine is defined within the vertex shader. The first step involves declaring the subroutine type:

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
```

This defines a new subroutine type with the name `shadeModelType`. The syntax is very similar to a function prototype, in that it defines a name, a parameter list, and a return type. As with function prototypes, the parameter names are optional.

After creating the new subroutine type, we declare a uniform variable of that type named `shadeModel`:

```
subroutine uniform shadeModelType shadeModel;
```

This variable serves as our function pointer and will be assigned to one of the two possible functions in the OpenGL application.

We declare two functions to be part of the subroutine by prefixing their definition with the subroutine qualifier:

```
subroutine ( shadeModelType )
```

This indicates that the function matches the subroutine type, and therefore its header must match the one in the subroutine type definition. We use this prefix for the definition of the functions `phongModel` and `diffuseOnly`. The `diffuseOnly` function computes the diffuse shading equation, and the `phongModel` function computes the complete ADS shading equation.

We call one of the two subroutine functions by utilizing the subroutine uniform `shadeModel` within the main function:

```
LightIntensity = shadeModel( eyePosition, eyeNorm );
```

Again, this call will be bound to one of the two functions depending on the value of the subroutine uniform `shadeModel`, which we will set within the OpenGL application.

Within the render function of the OpenGL application, we assign a value to the subroutine uniform with the following two steps. First, we query for the index of each subroutine function using `glGetSubroutineIndex`. The first argument is the program handle. The second is the shader stage. In this case, the subroutine is defined within the vertex shader, so we use `GL_VERTEX_SHADER` here. The third argument is the name of the subroutine. We query for each function individually and store the indexes in the variables `adsIndex` and `diffuseIndex`.

Second, we select the appropriate subroutine function. To do so we need to set the value of the subroutine uniform `shadeModel` by calling `glUniformSubroutinesuiv`. This function is designed for setting multiple subroutine uniforms at once. In our case, of course, we are setting only a single uniform. The first argument is the shader stage (`GL_VERTEX_SHADER`), the second is the number of uniforms being set, and the third is a pointer to an array of subroutine function indexes. Since we are setting a single uniform, we simply provide the address of the `GLuint` variable containing the index, rather than a true array of values. Of course, we would use an array if multiple uniforms were being set. In general, the array of values provided as the third argument is assigned to subroutine uniform variables in the following way. The *i*th element of the array is assigned to the subroutine uniform variable with index *i*. Since we have provided only a single value, we are setting the subroutine uniform at index zero.

You may be wondering, "How do we know that our subroutine uniform is located at index zero? We didn't query for the index before calling `glUniformSubroutinesuiv`!" The reason that this code works is that we are relying on the fact that OpenGL will always number the indexes of the subroutines consecutively starting at zero. If we had multiple subroutine uniforms, we could (and should) query for their indexes using `glGetSubroutineUniformLocation`, and then order our array appropriately.



`glUniformSubroutinesuiv` requires us to set all subroutine uniform variables at once, in a single call. This is so that they can be validated by OpenGL in a single burst.

There's more...

Unfortunately, subroutine bindings get reset when a shader program is unbound (switched out) from the pipeline, by calling `glUseProgram` or other technique. This requires us to call `glUniformSubroutinesuiv` each time that we activate a shader program.

A subroutine function defined in a shader can match more than one subroutine type. The subroutine qualifier can contain a comma-separated list of subroutine types. For example, if a subroutine matched the types `type1` and `type2`, we could use the following qualifier:

```
subroutine( type1, type2 )
```

This would allow us to use subroutine uniforms of differing types to refer to the same subroutine function.

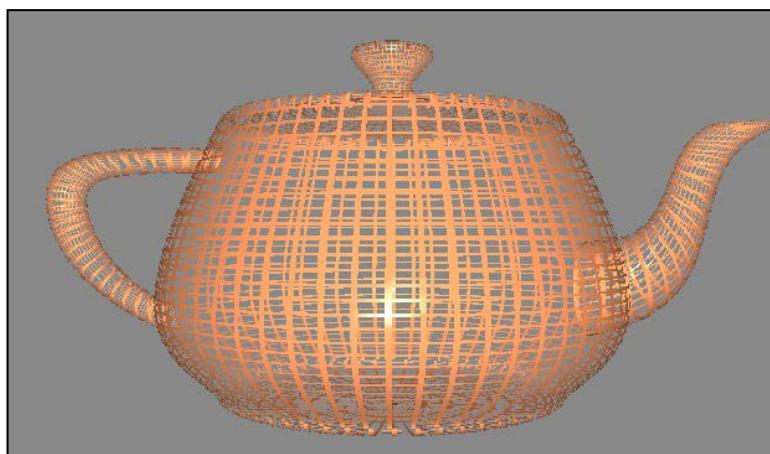
See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe
- ▶ The *Implementing diffuse, per-vertex shading with a single point light source* recipe

Discarding fragments to create a perforated look

Fragment shaders can make use of the `discard` keyword to "throw away" fragments. Use of this keyword causes the fragment shader to stop execution, without writing anything (including depth) to the output buffer. This provides a way to create holes in polygons without using blending. In fact, since fragments are completely discarded, there is no dependence on the order in which objects are drawn, saving us the trouble of doing any depth sorting that might have been necessary if blending was used.

In this recipe, we'll draw a teapot, and use the `discard` keyword to remove fragments selectively, based on texture coordinates. The result will look like the following diagram:



Getting ready

The vertex position, normal, and texture coordinates must be provided to the vertex shader from the OpenGL application. The position should be provided at location 0, the normal at location 1, and the texture coordinates at location 2. As in the previous examples, the lighting parameters must be set from the OpenGL application via the appropriate uniform variables.

How to do it...

To create a shader program that discards fragments based on a square lattice (as in the preceding screenshot), use the following code:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 FrontColor;
out vec3 BackColor;
out vec2 TexCoord;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 La;        // Ambient light intensity
    vec3 Ld;        // Diffuse light intensity
    vec3 Ls;        // Specular light intensity
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};

uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position )
{
```

```
    norm = normalize( NormalMatrix * VertexNormal);
    position = ModelViewMatrix * vec4(VertexPosition,1.0);
}

vec3 phongModel( vec4 position, vec3 norm )
{
    // The ADS shading calculation (see: "Implementing
    // per-vertex ambient, diffuse, and specular (ADS)
    // shading")
    ...
}

void main()
{
    vec3 eyeNorm;
    vec4 eyePosition;

    TexCoord = VertexTexCoord;

    // Get the position and normal in eye space
    getEyeSpace(eyeNorm, eyePosition);

    FrontColor = phongModel( eyePosition, eyeNorm );
    BackColor = phongModel( eyePosition, -eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 FrontColor;
in vec3 BackColor;
in vec2 TexCoord;

layout( location = 0 ) out vec4 FragColor;

void main() {
    const float scale = 15.0;

    bvec2 toDiscard = greaterThan( fract(TexCoord * scale),
                                   vec2(0.2,0.2) );

    if( all(toDiscard) )
        discard;

    if( gl_FrontFacing )
```

```
    FragColor = vec4(FrontColor, 1.0);
else
    FragColor = vec4(BackColor, 1.0);
}
```

3. Compile and link both shaders within the OpenGL application, and install the shader program prior to rendering.

How it works...

Since we will be discarding some parts of the teapot, we will be able to see through the teapot to the other side. This will cause the back sides of some polygons to become visible. Therefore, we need to compute the lighting equation appropriately for both sides of each face. We'll use the same technique presented earlier in the two-sided shading recipe.

The vertex shader is essentially the same as in the two-sided shading recipe, with the main difference being the addition of the texture coordinate. The differences are highlighted in the previous listing. To manage the texture coordinate, we have an additional input variable, `VertexTexCoord`, that corresponds to attribute location 2. The value of this input variable is passed directly on to the fragment shader unchanged via the output variable `TexCoord`. The ADS shading model is calculated twice, once using the given normal vector, storing the result in `FrontColor`, and again using the reversed normal, storing that result in `BackColor`.

In the fragment shader, we calculate whether or not the fragment should be discarded based on a simple technique designed to produce the lattice-like pattern shown in the preceding screenshot. We first scale the texture coordinate by the arbitrary scaling factor `scale`. This corresponds to the number of lattice rectangles per unit (scaled) texture coordinate. We then compute the fractional part of each component of the scaled texture coordinate using the built-in function `fract`. Each component is compared to 0.2 using the built-in function `greaterThan`, and the result is stored in the `bool` vector `toDiscard`. The `greaterThan` function compares the two vectors component-wise, and stores the Boolean results in the corresponding components of the return value.

If both components of the vector `toDiscard` are true, then the fragment lies within the inside of each lattice frame, and therefore we wish to discard this fragment. We can use the built-in function `all` to help with this check. The function `all` will return true if all of the components of the parameter vector are true. If the function returns true, we execute the `discard` statement to reject the fragment.

In the `else` branch, we color the fragment based on the orientation of the polygon, as in the *Implementing two-sided shading* recipe presented earlier.

See also

- ▶ The *Implementing two-sided shading* recipe

3

Lighting, Shading, and Optimization

In this chapter, we will cover:

- ▶ Shading with multiple positional lights
- ▶ Shading with a directional light source
- ▶ Using per-fragment shading for improved realism
- ▶ Using the halfway vector for improved performance
- ▶ Simulating a spotlight
- ▶ Creating a cartoon shading effect
- ▶ Simulating fog
- ▶ Configuring the depth test

Introduction

In *Chapter 2, The Basics of GLSL Shaders*, we covered a number of techniques for implementing some of the shading effects that were produced by the former fixed-function pipeline. We also looked at some basic features of GLSL such as functions and subroutines. In this chapter, we'll move beyond the shading model introduced in *Chapter 2, The Basics of GLSL Shaders*, and see how to produce shading effects such as spotlights, fog, and cartoon style shading. We'll cover how to use multiple light sources, and how to improve the realism of the results with a technique called per-fragment shading.

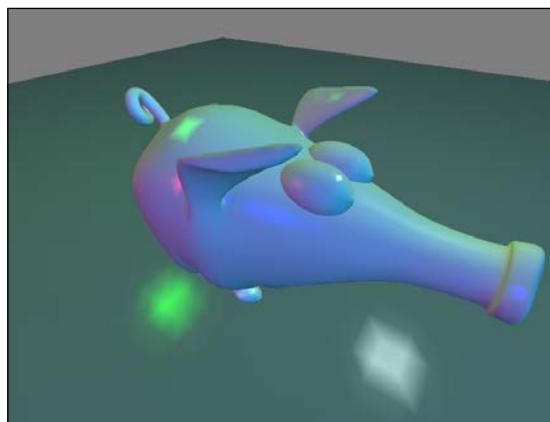
We'll also see techniques for improving the efficiency of the shading calculations by using the so-called "halfway vector" and directional light sources.

Finally, we'll cover how to fine-tune the depth test by configuring the early depth test optimization.

Shading with multiple positional lights

When shading with multiple light sources, we need to evaluate the shading equation for each light and sum the results to determine the total light intensity reflected by a surface location. The natural choice is to create uniform arrays to store the position and intensity of each light. We'll use an array of structures so that we can store the values for multiple lights within a single uniform variable.

The following figure shows a "pig" mesh rendered with five light sources of different colors. Note the multiple specular highlights.



Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the normal in location one.

How to do it...

To create a shader program that renders using the ADS (Phong) shading model with multiple light sources, use the following steps:

1. Use the following vertex shader:

```
layout (location = 0) in vec3 VertexPosition;  
layout (location = 1) in vec3 VertexNormal;
```

```
out vec3 Color;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // Light intensity
};
uniform LightInfo lights[5];

// Material parameters
uniform vec3 Kd; // Diffuse reflectivity
uniform vec3 Ka; // Ambient reflectivity
uniform vec3 Ks; // Specular reflectivity
uniform float Shininess; // Specular shininess factor

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

vec3 ads( int lightIndex, vec4 position, vec3 norm )
{
    vec3 s = normalize( vec3(lights[lightIndex].Position -
                           position) );
    vec3 v = normalize(vec3(-position));
    vec3 r = reflect( -s, norm );
    vec3 I = lights[lightIndex].Intensity;
    return
        I * ( Ka +
               Kd * max( dot(s, norm), 0.0 ) +
               Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}
void main()
{
    vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyePosition = ModelViewMatrix *
                       vec4(VertexPosition,1.0);

    // Evaluate the lighting equation for each light
    Color = vec3(0.0);
    for( int i = 0; i < 5; i++ )
        Color += ads( i, eyePosition, eyeNorm );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following simple fragment shader:

```
in vec3 Color;  
  
layout( location = 0 ) out vec4 FragColor;  
  
void main() {  
    FragColor = vec4(Color, 1.0);  
}
```

3. In the OpenGL application, set the values for the `lights` array in the vertex shader. For each light, use something similar to the following code. This example uses the C++ shader program class (`prog` is a `GLSLProgram` object).

```
prog.setUniform("lights[0].Intensity",  
                vec3(0.0f, 0.8f, 0.8f));  
prog.setUniform("lights[0].Position", position);
```

Update the array index as appropriate for each light.

How it works...

Within the vertex shader, the lighting parameters are stored in the uniform array `lights`. Each element of the array is a struct of type `LightInfo`. This example uses five lights. The light intensity is stored in the `Intensity` field, and the position in eye coordinates is stored in the `Position` field.

The rest of the uniform variables are essentially the same as in the ADS (ambient, diffuse, and specular) shader presented in *Chapter 2, The Basics of GLSL Shaders*.

The `ads` function is responsible for computing the shading equation for a given light source. The index of the light is provided as the first parameter `lightIndex`. The equation is computed based on the values in the `lights` array at that index.

In the `main` function, a `for` loop is used to compute the shading equation for each light, and the results are summed into the shader output variable `Color`.

The fragment shader simply applies the interpolated color to the fragment.

See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2, The Basics of GLSL shaders*
- ▶ The *Shading with a directional light source* recipe

Shading with a directional light source

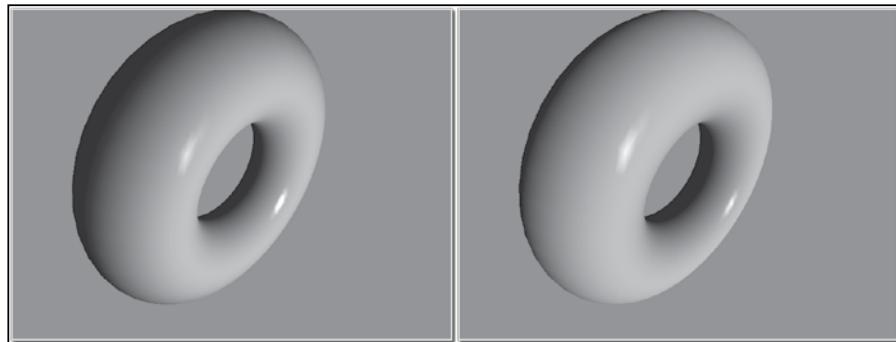
A core component of a shading equation is the vector that points from the surface location towards the light source (s in previous examples). For lights that are extremely far away, there is very little variation in this vector over the surface of an object. In fact, for very distant light sources, the vector is essentially the same for all points on a surface. (Another way of thinking about this is that the light rays are nearly parallel.) Such a model would be appropriate for a distant, but powerful, light source such as the sun. Such a light source is commonly called a **directional light source** because it does not have a specific position, only a direction.



Of course, we are ignoring the fact that, in reality, the intensity of the light decreases with the square of the distance from the source. However, it is not uncommon to ignore this aspect for directional light sources.

If we are using a directional light source, the direction towards the source is the same for all points in the scene. Therefore, we can increase the efficiency of our shading calculations because we no longer need to recompute the direction towards the light source for each location on the surface.

Of course, there is a visual difference between a positional light source and a directional one. The following figures show a torus rendered with a positional light (left) and a directional light (right). In the left figure, the light is located somewhat close to the torus. The directional light covers more of the surface of the torus due to the fact that all of the rays are parallel.



In previous versions of OpenGL, the fourth component of the light position was used to determine whether or not a light was considered directional. A zero in the fourth component indicated that the light source was directional and the position was to be treated as a direction towards the source (a vector). Otherwise, the position was treated as the actual location of the light source. In this example, we'll emulate the same functionality.

Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the vertex normal in location one.

How to do it...

To create a shader program that implements ADS shading using a directional light source, use the following code:

1. Use the following vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Color;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;

uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ka;           // Ambient reflectivity
uniform vec3 Ks;           // Specular reflectivity
uniform float Shininess;   // Specular shininess factor

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

vec3 ads( vec4 position, vec3 norm )
{
    vec3 s;
    if( LightPosition.w == 0.0 )
        s = normalize(vec3(LightPosition));
    else
        s = normalize(vec3(LightPosition - position));
    vec3 v = normalize(vec3(-position));
    vec3 r = reflect( -s, norm );
    return
        LightIntensity * ( Ka +
                           Kd * max( dot(s, norm), 0.0 ) +
                           Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}
```

```
void main()
{
    vec3 eyeNorm = normalize( NormalMatrix * VertexNormal);
    vec4 eyePosition = ModelViewMatrix *
        vec4(VertexPosition,1.0);

    // Evaluate the lighting equation
    Color = ads( eyePosition, eyeNorm );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the same simple fragment shader from the previous recipe:

```
in vec3 Color;

layout( location = 0 ) out vec4 FragColor;

void main() {
    FragColor = vec4(Color, 1.0);
}
```

How it works...

Within the vertex shader, the fourth coordinate of the uniform variable LightPosition is used to determine whether or not the light is to be treated as a directional light. Inside the ads function, which is responsible for computing the shading equation, the value of the vector s is determined based on whether or not the fourth coordinate of LightPosition is zero. If the value is zero, LightPosition is normalized and used as the direction towards the light source. Otherwise, LightPosition is treated as a location in eye coordinates, and we compute the direction towards the light source by subtracting the vertex position from LightPosition and normalizing the result.

There's more...

There is a slight efficiency gain when using directional lights due to the fact that there is no need to recompute the light direction for each vertex. This saves a subtraction operation, which is a small gain, but could accumulate when there are several lights, or when the lighting is computed per-fragment.

See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*.
- ▶ The *Using per-fragment shading for improved realism* recipe

Using per-fragment shading for improved realism

When the shading equation is evaluated within the vertex shader (as we have done in previous recipes), we end up with a color associated with each vertex. That color is then interpolated across the face, and the fragment shader assigns that interpolated color to the output fragment. As mentioned previously (the *Implementing flat shading* recipe in *Chapter 2, The Basics of GLSL Shaders*), this technique is often called **Gouraud shading**. Gouraud shading (like all shading techniques) is an approximation, and can lead to some less than desirable results when; for example, the reflection characteristics at the vertices have little resemblance to those in the center of the polygon. For example, a bright specular highlight may reside in the center of a polygon, but not at its vertices. Simply evaluating the shading equation at the vertices would prevent the specular highlight from appearing in the rendered result. Other undesirable artifacts, such as edges of polygons, may also appear when Gouraud shading is used, due to the fact that color interpolation is less physically accurate.

To improve the accuracy of our results, we can move the computation of the shading equation from the vertex shader to the fragment shader. Instead of interpolating color across the polygon, we interpolate the position and normal vector, and use these values to evaluate the shading equation at each fragment. This technique is often called **Phong shading** or **Phong interpolation**. The results from Phong shading are much more accurate and provide more pleasing results, but some undesirable artifacts may still appear.

The following figure shows the difference between Gouraud and Phong shading. The scene on the left is rendered with Gouraud (per-vertex) shading, and on the right is the same scene rendered using Phong (per-fragment) shading. Underneath the teapot is a partial plane, drawn with a single quad. Note the difference in the specular highlight on the teapot, as well as the variation in the color of the plane beneath the teapot.



In this example, we'll implement Phong shading by passing the position and normal from the vertex shader to the fragment shader, and then evaluate the ADS shading model within the fragment shader.

Getting ready

Set up your OpenGL program with the vertex position in attribute location zero, and the normal in location one. Your OpenGL application must also provide the values for the uniform variables `Ka`, `Kd`, `Ks`, `Shininess`, `LightPosition`, and `LightIntensity`, the first four of which are the standard material properties (reflectivities) of the ADS shading model. The latter two are the position of the light in eye coordinates, and the intensity of the light source, respectively. Finally, the OpenGL application must also provide the values for the uniforms `ModelViewMatrix`, `NormalMatrix`, `ProjectionMatrix`, and `MVP`.

How to do it...

To create a shader program that can be used for implementing per-fragment (or Phong) shading using the ADS shading model, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix *
                    vec4(VertexPosition,1.0) );
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;
uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ka;           // Ambient reflectivity
uniform vec3 Ks;           // Specular reflectivity
uniform float Shininess;   // Specular shininess factor
```

```
layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
    vec3 n = normalize( Normal );
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 r = reflect( -s, n );
    return
        LightIntensity *
        ( Ka +
        Kd * max( dot(s, n), 0.0 ) +
        Ks * pow( max( dot(r,v), 0.0 ), Shininess ) );
}

void main()
{
    FragColor = vec4(ads(), 1.0);
}
```

How it works...

The vertex shader has two output variables: `Position` and `Normal`. In the `main` function, we convert the vertex normal to eye coordinates by transforming with the normal matrix, and then store the converted value in `Normal`. Similarly, the vertex position is converted to eye coordinates by transforming it by the model-view matrix, and the converted value is stored in `Position`.

The values of `Position` and `Normal` are automatically interpolated and provided to the fragment shader via the corresponding input variables. The fragment shader then computes the standard ADS shading equation using the values provided. The result is then stored in the output variable, `FragColor`.

There's more...

Evaluating the shading equation within the fragment shader produces more accurate renderings. However, the price we pay is in the evaluation of the shading model for each pixel of the polygon, rather than at each vertex. The good news is that with modern graphics cards, there may be enough processing power to evaluate all of the fragments for a polygon in parallel. This can essentially provide nearly equivalent performance for either per-fragment or per-vertex shading.

See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*

Using the halfway vector for improved performance

As covered in the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*, the specular term in the ADS shading equation involves the dot product of the vector of pure reflection (\mathbf{r}), and the direction towards the viewer (\mathbf{v}).

$$I_s = L_s K_s (\mathbf{r} \cdot \mathbf{v})^f$$

In order to evaluate the above equation, we need to find the vector of pure reflection (\mathbf{r}), which is the reflection of the vector towards the light source (\mathbf{s}) about the normal vector (\mathbf{n}).

$$\mathbf{r} = -\mathbf{s} + 2(\mathbf{s} \cdot \mathbf{n})\mathbf{n}$$

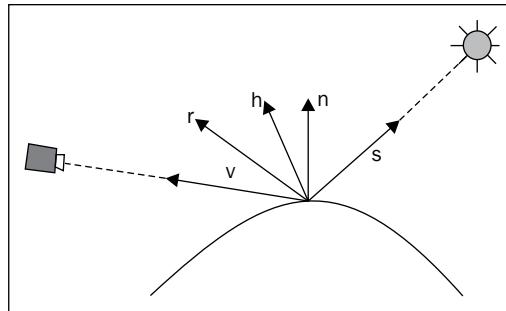
[ This equation is implemented by the GLSL function: `reflect`.]

The above equation requires a dot product, an addition, and a couple of multiplication operations. We can gain a slight improvement in the efficiency of the specular calculation by making use of the following observation. When \mathbf{v} is aligned with \mathbf{r} , the normal vector (\mathbf{n}) must be halfway between \mathbf{v} and \mathbf{s} .

Let's define the halfway vector (\mathbf{h}) as the vector that is halfway between \mathbf{v} and \mathbf{s} , where \mathbf{h} is normalized after the addition:

$$\mathbf{h} = \mathbf{v} + \mathbf{s}$$

The following diagram shows the relative positions of the halfway vector and the others:



We can then replace the dot product in the equation for the specular component, with the dot product of **h** and **n**.

$$I_s = L_s K_s (\mathbf{h} \cdot \mathbf{n})^f$$

Computing **h** requires fewer operations than it takes to compute **r**, so we should expect some efficiency gain by using the halfway vector. The angle between the halfway vector and the normal vector is proportional to the angle between the vector of pure reflection (**r**) and the vector towards the viewer (**v**) when all vectors are coplanar. Therefore, we expect that the visual results will be similar, although not exactly the same.

Getting ready

Start by utilizing the same shader program that was presented in the recipe, *Using per-fragment shading for improved realism*, and set up your OpenGL program as described there.

How to do it...

Using the same shader pair as in the recipe, *Using per-fragment shading for improved realism*, replace the `ads` function in the fragment shader with the following code:

```
vec3 ads( )
{
    vec3 n = normalize( Normal );
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 h = normalize( v + s );

    return
        LightIntensity *
        (Ka +
        Kd * max( dot(s, Normal), 0.0 ) +
        Ks * pow(max(dot(h,n),0.0), Shininess ) );
}
```

How it works...

We compute the halfway vector by summing the direction towards the viewer (**v**), and the direction towards the light source (**s**), and normalizing the result. The value for the halfway vector is then stored in **h**.

The specular calculation is then modified to use the dot product between h and the normal vector (`Normal`). The rest of the calculation is unchanged.

There's more...

The halfway vector provides a slight improvement in the efficiency of our specular calculation, and the visual results are quite similar. The following figure shows the teapot rendered using the halfway vector (right), versus the same rendering using the equation provided in the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders* (left). The halfway vector produces a larger specular highlight, but the visual impact is not substantially different. If desired, we could compensate for the difference in the size of the specular highlight by increasing the value of the exponent `Shininess`.



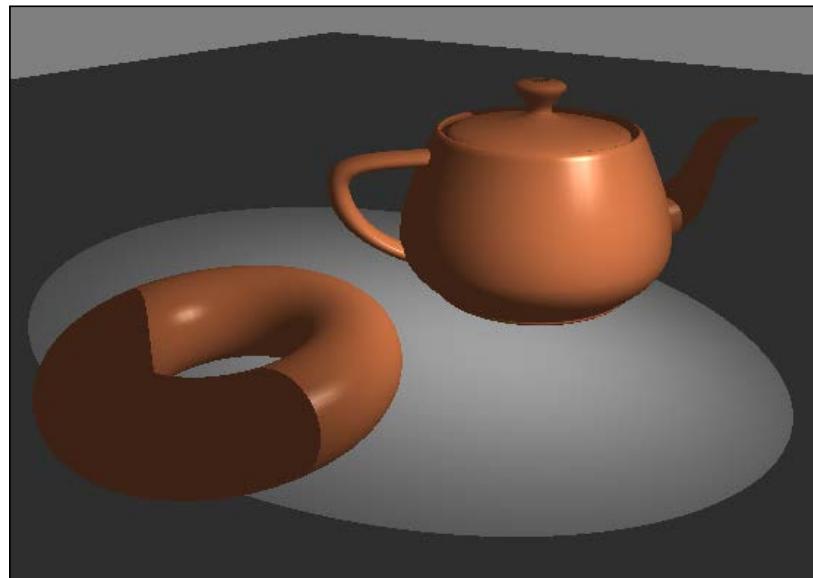
See also

- ▶ The *Using per-fragment shading for improved realism* recipe

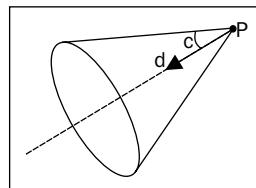
Simulating a spotlight

The fixed function pipeline had the ability to define light sources as spotlights. In such a configuration, the light source was considered to be one that only radiated light within a cone, the apex of which was located at the light source. Additionally, the light was attenuated so that it was maximal along the axis of the cone and decreased towards the outside edges. This allowed us to create light sources that had a similar visual effect to a real spotlight.

The following figure shows a teapot and a torus rendered with a single spotlight. Note the slight decrease in the intensity of the spotlight from the center towards the outside edge.



In this recipe, we'll use a shader to implement a spotlight effect similar to that produced by the fixed-function pipeline.



The spotlight's cone is defined by a spotlight direction (**d** in the preceding figure), a cutoff angle (**c** in the preceding figure), and a position (**P** in the preceding figure). The intensity of the spotlight is considered to be strongest along the axis of the cone, and decreases as you move towards the edges.

Getting ready

Start with the same vertex shader from the recipe, *Using per-fragment shading for improved realism*. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader shown below.

How to do it...

To create a shader program that uses the ADS shading model with a spotlight, use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

struct SpotLightInfo {
    vec4 position; // Position in eye coords.
    vec3 intensity; // Amb., Diff., and Specular intensity
    vec3 direction; // Normalized direction of the spotlight
    float exponent; // Angular attenuation exponent
    float cutoff; // Cutoff angle (between 0 and 90)
};
uniform SpotLightInfo Spot;

uniform vec3 Kd; // Diffuse reflectivity
uniform vec3 Ka; // Ambient reflectivity
uniform vec3 Ks; // Specular reflectivity
uniform float Shininess; // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 adsWithSpotlight( )
{
    vec3 s = normalize( vec3( Spot.position ) - Position );
    float angle = acos( dot( -s, Spot.direction ) );
    float cutoff = radians( clamp( Spot.cutoff, 0.0, 90.0 ) );
    vec3 ambient = Spot.intensity * Ka;

    if( angle < cutoff ) {
        float spotFactor = pow( dot( -s, Spot.direction ),
                               Spot.exponent );
        vec3 v = normalize( vec3( -Position ) );
        vec3 h = normalize( v + s );
        return
            ambient +
            spotFactor * Spot.intensity * (
                Kd * max( dot( s, Normal ), 0.0 ) +
                Ks * pow( max( dot( h, Normal ), 0.0 ), Shininess ) );
    } else {
        return ambient;
    }
}

void main() {
    FragColor = vec4( adsWithSpotlight(), 1.0 );
}
```

How it works...

The structure `SpotLightInfo` defines all of the configuration options for the spotlight. We declare a single uniform variable named `Spot` to store the data for our spotlight. The `position` field defines the location of the spotlight in eye coordinates. The `intensity` field is the intensity (ambient, diffuse, and specular) of the spotlight. If desired, you could break this into three variables. The `direction` field will contain the direction that the spotlight is pointing, which defines the center axis of the spotlight's cone. This vector should be specified in eye coordinates. Within the OpenGL program it should be transformed by the normal matrix in the same way that normal vectors would be transformed. We could do so within the shader; however, within the shader, the normal matrix would be specified for the object being rendered. This may not be the appropriate transform for the spotlight's direction.

The `exponent` field defines the exponent that is used when calculating the angular attenuation of the spotlight. The intensity of the spotlight is decreased in proportion to the cosine of the angle between the vector from the light to the surface location (the negation of the variable `s`) and the direction of the spotlight. That cosine term is then raised to the power of the variable `exponent`. The larger the value of this variable, the faster the intensity of the spotlight is decreased. This is quite similar to the exponent in the specular shading term.

The `cutoff` field defines the angle between the central axis and the outer edge of the spotlight's cone of light. We specify this angle in degrees, and clamp its value between 0 and 90.

The function, `adsWithSpotlight`, computes the standard ambient, diffuse, and specular (ADS) shading equation, using a spotlight as the light source. The first line computes the vector from the surface location to the spotlight's position (`s`). Next, the spotlight's direction is normalized and stored within `spotDir`. The angle between `spotDir` and the negation of `s` is then computed and stored in the variable `angle`. The `cutoff` variable stores the value of `Spot.cutoff` after it has been clamped between 0 and 90, and converted from degrees to radians. Next, the ambient lighting component is computed and stored in the `ambient` variable.

We then compare the value of the `angle` variable with that of the `cutoff` variable. If `angle` is less than `cutoff`, then the surface point is within the spotlight's cone. Otherwise the surface point only receives ambient light, so we return only the ambient component.

If `angle` is less than `cutoff`, we compute the `spotFactor` variable by raising the dot product of `-s` and `spotDir` to the power of `Spot.exponent`. The value of `spotFactor` is used to scale the intensity of the light so that the light is maximal in the center of the cone, and decreases as you move towards the edges. Finally, the ADS shading equation is computed as usual, but the diffuse and specular terms are scaled by `spotFactor`.

See also

- ▶ The *Using per-fragment shading for improved realism* recipe
- ▶ The *Implementing per-vertex ambient, diffuse, and specular shading (ADS)* recipe in Chapter 2, *The Basics of GLSL Shaders*

Creating a cartoon shading effect

Toon shading (also called **Celshading**) is a non-photorealistic technique that is intended to mimic the style of shading often used in hand-drawn animation. There are many different techniques that are used to produce this effect. In this recipe, we'll use a very simple technique that involves a slight modification to the ambient and diffuse shading model.

The basic effect is to have large areas of constant color with sharp transitions between them. This simulates the way that an artist might shade an object using strokes of a pen or brush. The following figure shows an example of a teapot and torus rendered with toon shading.



The technique presented here involves computing only the ambient and diffuse components of the typical ADS shading model, and quantizing the cosine term of the diffuse component. In other words, the value of the dot product normally used in the diffuse term is restricted to a fixed number of possible values. The following table illustrates the concept for four levels:

Cosine of the Angle between s and n	Value used
Between 1 and 0.75	0.75
Between 0.75 and 0.5	0.5
Between 0.5 and 0.25	0.25
Between 0.25 and 0.0	0.0

In the preceding table, **s** is the vector towards the light source and **n** is the normal vector at the surface. By restricting the value of the cosine term in this way, the shading displays strong discontinuities from one level to another (see the preceding figure), simulating the pen strokes of hand-drawn cel animation.

Getting ready

Start with the same vertex shader from the *Using per-fragment shading for improved realism* recipe. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader code described below.

How to do it...

To create a shader program that produces a toon shading effect, use the following fragment shader:

```
in vec3 Position;
in vec3 Normal;

struct LightInfo {
    vec4 position;
    vec3 intensity;
};
uniform LightInfo Light;

uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ka;           // Ambient reflectivity

const int levels = 3;
const float scaleFactor = 1.0 / levels;
layout( location = 0 ) out vec4 FragColor;
vec3 toonShade( )
{
    vec3 s = normalize( Light.position.xyz - Position.xyz );
    float cosine = max( 0.0, dot( s, Normal ) );
    vec3 diffuse = Kd * floor( cosine * levels ) *
                  scaleFactor;

    return Light.intensity * (Ka + diffuse);
}

void main()
{
    FragColor = vec4(toonShade(), 1.0);
}
```

How it works...

The constant variable, `levels`, defines how many distinct values will be used in the diffuse calculation. This could also be defined as a uniform variable to allow for configuration from the main OpenGL application. We will use this variable to quantize the value of the cosine term in the diffuse calculation.

The `toonShade` function is the most significant part of this shader. We start by computing `s`, the vector towards the light source. Next, we compute the cosine term of the diffuse component by evaluating the dot product of `s` and `Normal`. The next line quantizes that value in the following way. Since the two vectors are normalized, and we have removed negative values with the `max` function, we are sure that the value of cosine is between zero and one. By multiplying this value by `levels` and taking the floor, the result will be an integer between 0 and `levels - 1`. When we divide that value by `levels` (by multiplying by `scaleFactor`), we scale these integral values to be between zero and one again. The result is a value that can be one of `levels` possible values spaced between zero and one. This result is then multiplied by `Kd`, the diffuse reflectivity term.

Finally, we combine the diffuse and ambient components together to get the final color for the fragment.

There's more...

When quantizing the cosine term, we could have used `ceil` instead of `floor`. Doing so would have simply shifted each of the possible values up by one level. This would make the levels of shading slightly brighter.

The typical cartoon style seen in most cel animation includes black outlines around the silhouettes and along other edges of a shape. The shading model presented here does not produce those black outlines. There are several techniques for producing them, and we'll look at one later on in this book.

See also

- ▶ The *Using per-fragment shading for improved realism* recipe
- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*
- ▶ The *Drawing silhouette lines using the geometry shader* recipe in Chapter 6, *Using Geometry and Tessellation Shaders*

Simulating fog

A simple fog effect can be achieved by mixing the color of each fragment with a constant fog color. The amount of influence of the fog color is determined by the distance from the camera. We could use either a linear relationship between the distance and the amount of fog color, or we could use a non-linear relationship such as an exponential one.

The following figure shows four teapots rendered with a fog effect produced by mixing the fog color in a linear relationship with distance.



To define this linear relationship we can use the following equation:

$$f = \frac{d_{\max} - |z|}{d_{\max} - d_{\min}}$$

In the preceding equation, d_{\min} is the distance from the eye where the fog is minimal (no fog contribution), and d_{\max} is the distance where the fog color obscures all other colors in the scene. The variable z represents the distance from the eye. The value f is the fog factor. A fog factor of zero represents 100 percent fog, and a factor of one represents no fog. Since fog typically looks thickest at large distances, the fog factor is minimal when $|z|$ is equal to d_{\max} , and maximal when $|z|$ is equal to d_{\min} .



Since the fog is applied by the fragment shader, the effect will only be visible on the objects that are rendered. It will not appear on any "empty" space in the scene (the background). To help make the fog effect consistent, you should use a background color that matches the maximum fog color.

Getting ready

Start with the same vertex shader from the *Using per-fragment shading for improved realism* recipe. Your OpenGL program must set the values for all uniform variables defined in that vertex shader as well as the fragment shader shown in the following section.

How to do it...

To create a shader that produces a fog-like effect, use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

struct tLightInfo {
    vec4 position;
    vec3 intensity;
};
uniform LightInfo Light;

struct FogInfo {
    float maxDist;
    float minDist;
    vec3 color;
};
uniform FogInfo Fog;

uniform vec3 Kd;           // Diffuse reflectivity
uniform vec3 Ka;           // Ambient reflectivity
uniform vec3 Ks;           // Specular reflectivity
uniform float Shininess;   // Specular shininess factor

layout( location = 0 ) out vec4 FragColor;

vec3 ads( )
{
    // ... The ADS shading algorithm
}

void main() {
    float dist = abs( Position.z );
    float fogFactor = (Fog.maxDist - dist) /
                      (Fog.maxDist - Fog.minDist);
    fogFactor = clamp( fogFactor, 0.0, 1.0 );
    vec3 shadeColor = ads();
    vec3 color = mix( Fog.color, shadeColor, fogFactor );

    FragColor = vec4( color, 1.0 );
}
```

How it works...

In this shader, the `ads` function is exactly the same as the one used in the recipe *Using the halfway vector for improved performance*. The part of this shader that deals with the fog effect lies within the `main` function.

The uniform variable `Fog` contains the parameters that define the extent and color of the fog. The `minDist` field is the distance from the eye to the fog's starting point, and `maxDist` is the distance to the point where the fog is maximal. The `color` field is the color of the fog.

The `dist` variable is used to store the distance from the surface point to the eye position. The `z` coordinate of the position is used as an estimate of the actual distance. The `fogFactor` variable is computed using the preceding equation. Since `dist` may not be between `minDist` and `maxDist`, we clamp the value of `fogFactor` to be between zero and one.

We then call the `ads` function to evaluate the basic ADS shading model. The result of this is stored in the `shadeColor` variable.

Finally, we mix `shadeColor` and `Fog.color` together based on the value of `fogFactor`, and the result is used as the fragment color.

There's more...

In this recipe, we used a linear relationship between the amount of fog color and the distance from the eye. Another choice would be to use an exponential relationship. For example, the following equation could be used:

$$f = e^{-d|z|}$$

In the above equation, `d` represents the density of the fog. Larger values would create "thicker" fog. We could also square the exponent to create a slightly different relationship (a faster increase in the fog with distance).

$$f = e^{-(dz)^2}$$

Computing distance from the eye

In the above code, we used the absolute value of the `z` coordinate as the distance from the camera. This may cause the fog to look a bit unrealistic in certain situations. To compute a more precise distance, we could replace the line:

```
float dist = abs( Position.z );
```

with the following:

```
float dist = length( Position.xyz );
```

Of course, the latter version requires a square root, and therefore would be a bit slower in practice.

See also

- ▶ The *Using per-fragment shading for improved realism* recipe
- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*

Configuring the depth test

GLSL 4 provides the ability to configure how the depth test is performed. This gives us additional control over how and when fragments are tested against the depth buffer.

Many OpenGL implementations automatically provide an optimization known as the early depth test or early fragment test. With this optimization, the depth test is performed before the fragment shader is executed. Since fragments that fail the depth test will not appear on the screen (or the framebuffer), there is no point in executing the fragment shader at all for those fragments and we can save some time by avoiding the execution.

The OpenGL specification, however, states that the depth test is performed *after* the fragment shader. This means that if an implementation wishes to use the early depth test optimization, it must be careful. The implementation must make sure that if anything within the fragment shader might change the results of the depth test, then it should avoid using the early depth test.

For example, a fragment shader can change the depth of a fragment by writing to the output variable, `gl_FragDepth`. If it does so, then the early depth test cannot be performed because, of course, the final depth of the fragment is not known prior to the execution of the fragment shader. However, the GLSL provides ways to notify the pipeline roughly how the depth will be modified, so that the implementation may determine when it might be ok to use the early depth test.

Another possibility is that the fragment shader might conditionally discard the fragment using the `discard` keyword. If there is any possibility that the fragment may be discarded, some implementations may not perform the early depth test.

There are also certain situations where we want to rely on the early depth test. For example, if the fragment shader writes to memory other than the framebuffer (with image load/store, shader storage buffers, or other incoherent memory writing), we might not want the fragment shader to execute for fragments that fail the depth test. This would help us to avoid writing data for fragments that fail. The GLSL provides a technique for forcing the early depth test optimization.

How to do it...

To ask the OpenGL pipeline to always perform the early depth test optimization, use the following layout qualifier in your fragment shader:

```
layout(early_fragment_tests) in;
```

If your fragment shader will modify the fragment's depth, but you still would like to take advantage of the early depth test when possible, use the following layout qualifier in a declaration of `gl_FragDepth` within your fragment shader:

```
layout(depth_*) out float gl_FragDepth;
```

Where, `depth_*` is one of the following: `depth_any`, `depth_greater`, `depth_less`, or `depth_unchanged`.

How it works...

The following statement forces the OpenGL implementation to always perform the early depth test:

```
layout(early_fragment_tests) in;
```

We must keep in mind that if we attempt to modify the depth anywhere within the shader by writing to `gl_FragDepth`, the value that is written will be ignored.

If your fragment shader needs to modify the depth value, then we can't force early fragment tests. However, we can help the pipeline to determine when it can still apply the early test. We do so by using one of the layout qualifiers for `gl_FragDepth` as shown above. This places some limits on how the value will be modified. The OpenGL implementation can then determine if the fragment shader can be skipped. If it can be determined that the depth will not be changed in such a way that it would cause the result of the test to change, the implementation can still use the optimization.

The layout qualifier for the output variable `gl_FragDepth` tells the OpenGL implementation specifically how the depth might change within the fragment shader. The qualifier `depth_any` indicates that it could change in any way. This is the default.

The other qualifiers describe how the value may change with respect to `gl_FragCoord.z`.

- ▶ `depth_greater`: This fragment shader promises to only increase the depth.
- ▶ `depth_less`: This fragment shader promises to only decrease the depth.
- ▶ `depth_unchanged`: This fragment shader promises not to change the depth. If it writes to `gl_FragDepth`, the value will be equal to `gl_FragCoord.z`.

If you use one of these qualifiers, but then go on to modify the depth in an incompatible way, the results are undefined. For example, if you declare `gl_FragDepth` with `depth_greater`, but decrease the depth of the fragment, the code will compile and execute, but you shouldn't expect to see accurate results.



If your fragment shader writes to `gl_FragDepth`, then it must be sure to write a value in all circumstances. In other words, it must write a value no matter which branches are taken within the code.



See also

- ▶ The *Implementing order-independent transparency* recipe in *Chapter 5, Image Processing and Screen Space Techniques*

4

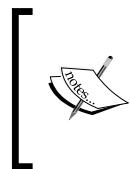
Using Textures

In this chapter, we will cover:

- ▶ Applying a 2D texture
- ▶ Applying multiple textures
- ▶ Using alpha maps to discard pixels
- ▶ Using normal maps
- ▶ Simulating reflection with cube maps
- ▶ Simulating refraction with cube maps
- ▶ Applying a projected texture
- ▶ Rendering to a texture
- ▶ Using sampler objects

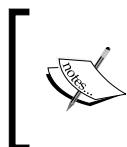
Introduction

Textures are an important and fundamental aspect of real-time rendering in general, and OpenGL in particular. The use of textures within a shader opens up a huge range of possibilities. Beyond just using textures as sources of color information, they can be used for things like depth information, shading parameters, displacement maps, normal vectors, or other vertex data. The list is virtually endless. Textures are among the most widely used tools for advanced effects in OpenGL programs, and that isn't likely to change anytime soon.



In OpenGL 4, we now have the ability to read and write to memory via buffer textures, shader storage buffer objects, and image textures (image load/store). This further muddies the waters of what exactly defines a texture. In general, we might just think of it as a buffer of data that may or may not contain an image.

OpenGL 4.2 introduced **immutable storage textures**. Despite what the term may imply, immutable storage textures are not textures that can't change. Instead, the term *immutable* refers to the fact that, once the texture is allocated, the storage cannot be changed. That is, the size, format, and number of layers are fixed, but the texture content itself can be modified. The word *immutable* refers to the allocation of the memory, not the contents of the memory. Immutable storage textures are preferable in the vast majority of cases because of the fact that many run-time (draw-time) consistency checks can be avoided, and you include a certain degree of "type safety," since we can't accidentally change the allocation of a texture. Throughout this book, we'll use immutable storage textures exclusively.



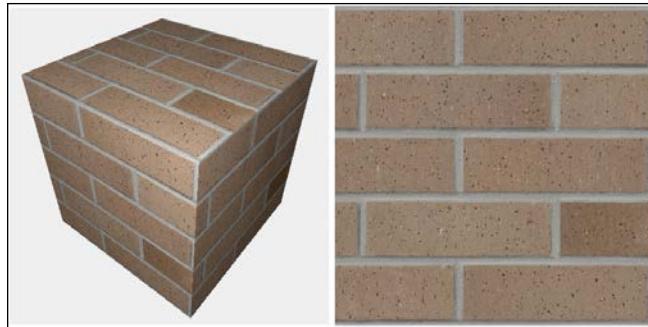
Immutable storage textures are allocated using the `glTexStorage*` functions. If you're experienced with textures, you might be accustomed to using `glTexImage*` functions, which are still supported, but create mutable storage textures.

In this chapter, we'll look at some basic and advanced texturing techniques. We'll start with the basics, just applying color textures, and move on to using textures as normal maps and environment maps. With environment maps, we can simulate things like reflection and refraction. We'll see an example of projecting a texture onto objects in a scene similar to the way that a slide projector projects an image. Finally, we'll wrap up with an example of rendering directly to a texture (using **framebuffer objects (FBOs)**) and then applying that texture to an object.

Applying a 2D texture

In GLSL, applying a texture to a surface involves accessing texture memory to retrieve a color associated with a texture coordinate, and then applying that color to the output fragment. The application of the color to the output fragment could involve mixing the color with the color produced by a shading model, simply applying the color directly, using the color in the reflection model, or some other mixing process. In GLSL, textures are accessed via **sampler** variables. A sampler variable is a "handle" to a texture unit. It is typically declared as a uniform variable within the shader and initialized within the main OpenGL application to point to the appropriate texture unit.

In this recipe, we'll look at a simple example involving the application of a 2D texture to a surface as shown in the following image. We'll use the texture color to scale the color provided by the **ambient, diffuse, and specular (ADS)** reflection model. The following image shows the results of a brick texture applied to a cube. The texture is shown on the right and the rendered result is on the left.



Getting ready

Set up your OpenGL application to provide the vertex position in attribute location 0, the vertex normal in attribute location 1, and the texture coordinate in attribute location 2. The parameters for the ADS reflection model are declared again as uniform variables within the shader, and must be initialized from the OpenGL program. Make the handle to the shader available in a variable named `programHandle`.

How to do it...

To render a simple shape with a 2D texture, use the following steps:

1. In your initialization of the OpenGL application, use the following code to load the texture. (The following makes use of a simple TGA image loader, provided with the sample code.)

```
GLint width, height;
GLubyte * data = TGAIO::read("brick1.tga", width, height);

// Copy file to OpenGL
glActiveTexture(GL_TEXTURE0);
GLuint tid;
 glGenTextures(1, &tid);
 glBindTexture(GL_TEXTURE_2D, tid);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,
                 GL_RGBA, GL_UNSIGNED_BYTE, data);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR);
```

```
delete [] data;
```

```
// Set the Tex1 sampler uniform to refer to texture unit 0
int loc = glGetUniformLocation(programHandle, "Tex1");
if( loc >= 0 )
    glUniform1i(loc, 0);
```

2. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    TexCoord = VertexTexCoord;
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix *
                    vec4(VertexPosition,1.0) );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform sampler2D Tex1;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // A,D,S intensity
};
```

```
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka;           // Ambient reflectivity
    vec3 Kd;           // Diffuse reflectivity
    vec3 Ks;           // Specular reflectivity
    float Shininess;   // Specular shininess factor
};

uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;

void phongModel( vec3 pos, vec3 norm,
    out vec3 ambAndDiff, out vec3 spec ) {
    // Compute the ADS shading model here, return ambient
    // and diffuse color in ambAndDiff, and return specular
    // color in spec
}

void main() {
    vec3 ambAndDiff, spec;
    vec4 texColor = texture( Tex1, TexCoord );
    phongModel( Position, Normal, ambAndDiff, spec );
    FragColor = vec4(ambAndDiff, 1.0) * texColor +
        vec4(spec, 1.0);
}
```

How it works...

The first code segment demonstrates the steps needed to load the texture from a file, copy the texture data to OpenGL memory, and initialize the sampler variable within the GLSL program. The first step, loading the texture image file, is accomplished via a simple TGA image loader that is provided along with the example code (`TGAI0::read()`). It reads the image data from a file in the TGA format, and stores the data into an array of unsigned bytes in RGBA order. The width and height of the image are returned via the last two parameters. We keep a pointer to the image data, simply named `data`.



The TGA format is simple and easy to understand, it is free of any encumbering patents, and supports true color images with an alpha channel. This makes it a very convenient format for texture reading/writing. If your images are not in that format, just grab a copy of ImageMagick and convert. However, the TGA format is not very memory efficient. If you want to load images stored in other formats, there are a variety of other options. For example, check out ResIL (<http://resil.sourceforge.net/>), or Freeimage (<http://freeimage.sourceforge.net/>).

Experienced OpenGL programmers should be familiar with the next part of the code. First, we call `glActiveTexture` to set the current active texture unit to `GL_TEXTURE0` (the first texture unit, also called a texture *channel*). The subsequent texture state calls will be effective on texture unit zero. The next two lines involve creating a new texture object by calling `glGenTextures`. The handle for the new texture object is stored in the variable `tid`. Then, we call `glBindTexture` to bind the new texture object to the `GL_TEXTURE_2D` target. Once the texture is bound to that target, we allocate immutable storage for the texture with `glTexStorage2D`. After that, we copy the data for that texture into the texture object using `glTexSubImage2D`. The last argument to this function is a pointer to the raw data for the image.

The next steps involve setting the magnification and minimization filters for the texture object using `glTexParameter`. For this example, we'll use `GL_LINEAR`.



The texture filter setting determines whether any interpolation will be done prior to returning the color from the texture. This setting can have a strong effect on the quality of the results. In this example, `GL_LINEAR` indicates that it will return a weighted average of the four texels that are nearest to the texture coordinates. For details on the other filtering options, see the OpenGL documentation for `glTexParameter`: <http://www.opengl.org/wiki/GLAPI/glTexParameter>.

Next, we delete the texture data pointed to by `data`. There's no need to hang on to this, because it was copied into texture memory via `glTexSubImage2D`.

Finally, we set the uniform variable `Tex1` in the GLSL program to zero. This is our sampler variable. Note that it is declared within the fragment shader with type `sampler2D`. Setting its value to zero indicates to the OpenGL system that the variable should refer to texture unit zero (the same one selected previously with `glActiveTexture`).

The vertex shader is very similar to the one used in previous examples except for the addition of the texture coordinate input variable `VertexTexCoord`, which is bound to attribute location 2. Its value is simply passed along to the fragment shader by assigning it to the shader output variable `TexCoord`.

The fragment shader is also very similar to those used in the recipes of previous chapters. The important parts for the purpose of this recipe involve the variable `Tex1`. `Tex1` is a `sampler2D` variable that was assigned by the OpenGL program to refer to texture unit zero. In the main function, we use that variable along with the texture coordinate (`TexCoord`) to access the texture. We do so by calling the built-in function `texture`. This is a general purpose function, used to access a texture. The first parameter is a sampler variable indicating which texture unit is to be accessed, and the second parameter is the texture coordinate used to access the texture. The return value is a `vec4` containing the color obtained by the texture access (stored in `texColor`), which in this case is an interpolated value with the four nearest texture values (texels).

Next, the shading model is evaluated by calling `phongModel` and the results are returned in the parameters `ambAndDiff` and `spec`. The variable `ambAndDiff` contains only the ambient and diffuse components of the shading model. A color texture is often only intended to affect the diffuse component of the shading model and not the specular. So we multiply the texture color by the ambient and diffuse components and then add the specular. The final sum is then applied to the output fragment `FragColor`.

There's more...

There are several choices that could be made when deciding how to combine the texture color with other colors associated with the fragment. In this example, we decided to multiply the colors, but one could have chosen to use the texture color directly, or to mix them in some way based on the alpha value.

Another choice would be to use the texture value as the value of the diffuse and/or specular reflectivity coefficient(s) in the Phong reflection model. The choice is up to you!

Specifying the sampler binding within GLSL

As of OpenGL 4.2, we now have the ability to specify the default value of the sampler's binding (the value of the sampler uniform) within GLSL. In the previous example, we of set the value of the uniform variable from the OpenGL side using the following code:

```
int loc = glGetUniformLocation(programHandle, "Tex1");
if( loc >= 0 )
    glUniform1i(loc, 0);
```

Instead, if we're using OpenGL 4.2, we can specify the default value within the shader, using the layout qualifier as shown in the following statement:

```
layout (binding=0) uniform sampler2D Tex1;
```

Thus simplifying the code on the OpenGL side, and making one less thing we need to worry about. The example code that accompanies this book uses this technique to specify the value of `Tex1`, so take a look there for a more complete example. We'll also use this layout qualifier in the following recipes.

See also

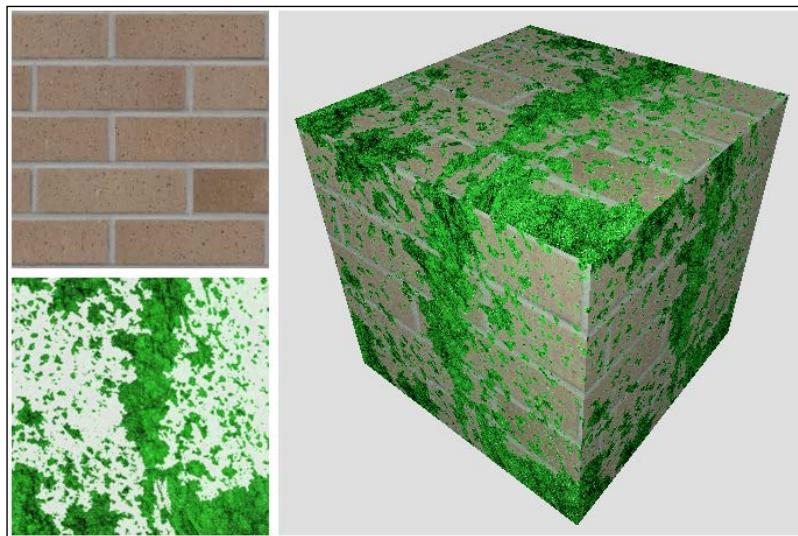
- ▶ For more information about sending data to a shader via vertex attributes refer the *Sending data to a shader using vertex attributes and vertex buffer objects* recipe in *Chapter 1, Getting Started with GLSL*
- ▶ The *Using per-fragment shading for improved realism* recipe in *Chapter 3, Lighting, Shading and Optimization*

Applying multiple textures

The application of multiple textures to a surface can be used to create a wide variety of effects. The base layer texture might represent the "clean" surface and the second layer could provide additional detail such as shadow, blemishes, roughness, or damage. In many games, so-called light maps are applied as an additional texture layer to provide the information about light exposure, effectively producing shadows and shading without the need to explicitly calculate the reflection model. These kinds of textures are sometimes referred to as "prebaked" lighting.

In this recipe, we'll demonstrate this multiple texture technique by applying two layers of texture. The base layer will be a fully opaque brick image, and the second layer will be one that is partially transparent. The non-transparent parts look like moss that has grown on the bricks beneath.

The following image shows an example of multiple textures. The textures on the left are applied to the cube on the right. The base layer is the brick texture, and the moss texture is applied on top. The transparent parts of the moss texture reveal the brick texture underneath.



Getting ready

Set up your OpenGL application to provide the vertex position in attribute location 0, the vertex normal in attribute location 1, and the texture coordinate in attribute location 2. The parameters for the Phong reflection model are declared as uniform variables within the shader and must be initialized from the OpenGL program.

How to do it...

To render objects with multiple textures, use the following steps:

1. In the initialization section of your OpenGL program, load the two images into texture memory in the same way as indicated in the previous recipe *Applying a 2D texture*. Make sure that the brick texture is loaded into texture unit 0 and the moss texture is in texture unit 1. Use the following code to do this:

```
GLuint texIDs[2];
GLint w, h;
 glGenTextures(2, texIDs);

// Load brick texture file
GLubyte * brickImg = TGAIO::read("brick1.tga", w, h);

// Copy brick texture to OpenGL
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texIDs[0]);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, w, h, GL_RGBA,
                 GL_UNSIGNED_BYTE, brickImg);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR);
 delete [] brickImg;

// Load moss texture file
GLubyte * mossImg = TGAIO::read("moss.tga", w, h);

// Copy moss texture to OpenGL
glActiveTexture(GL_TEXTURE1);
 glBindTexture(GL_TEXTURE_2D, texIDs[1]);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, w, h, GL_RGBA,
                 GL_UNSIGNED_BYTE, mossImg);
```

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
                GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                GL_LINEAR);  
  
delete [] mossImg;  
  
2. Use the vertex shader from the previous recipe Applying a 2D texture.  
3. Starting with the fragment shader from the recipe Applying a 2D texture, replace the declaration of the sampler variable Tex1 with the following code:  
layout(binding=0) uniform sampler2D BrickTex;  
layout(binding=1) uniform sampler2D MossTex;  
  
4. Replace the main function in the fragment shader with the following code:  
  
void main() {  
    vec3 ambAndDiff, spec;  
    vec4 brickTextColor = texture( BrickTex, TexCoord );  
    vec4 mossTextColor = texture( MossTex, TexCoord );  
    phongModel(Position, Normal, ambAndDiff, spec);  
    vec3 texColor = mix(brickTextColor, mossTextColor,  
                        mossTextColor.a);  
    FragColor = vec4(ambAndDiff, 1.0) * texColor +  
                vec4(spec,1.0);  
}
```

How it works...

The preceding code that loads the two textures into the OpenGL program is very similar to the code from the previous recipe *Applying a 2D texture*. The main difference is that we load each texture into a different texture unit. When loading the brick texture, we set the OpenGL state such that the active texture unit is unit zero.

```
glActiveTexture(GL_TEXTURE0);
```

And when loading the second texture, we set the OpenGL state to texture unit one.

```
glActiveTexture(GL_TEXTURE1);
```

In step 3, we specify the texture binding for each sampler variable using the layout qualifier, corresponding to the appropriate texture unit.

Within the fragment shader, we access the two textures using the corresponding uniform variables, and store the results in `brickTexColor` and `mossTexColor`. The two colors are blended together using the built-in function `mix`. The third parameter to the `mix` function is the percentage used when mixing the two colors. The alpha value of the moss texture is used for that parameter. This causes the result to be a linear interpolation of the two colors based on the value of the alpha in the moss texture. For those familiar with OpenGL blending functions, this is the same as the following blending function:

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

In this case, the moss color would be the source color, and the brick color would be the destination color.

Finally, we multiply the result of the `mix` function by the ambient and diffuse components of the Phong reflection model, add the specular component, and apply the result to the fragment.

There's more...

In this example, we mixed the two texture colors together using the alpha value of the second texture. This is just one of many options for mixing the texture colors. There are a number of different choices here, and your choice will be dependent on the kind of texture data available and the desired effect.

A popular technique is to use an additional vertex attribute to augment the amount of blending between the textures. This additional vertex attribute would allow us to vary the blending factor throughout a model. For example, we could vary the amount of moss that grows on a surface by defining another vertex attribute, which would control the amount of blending between the moss texture and the base texture. A value of zero might correspond to zero moss, up to a value of one that would enable blending based on the texture's alpha value alone.

See also

- ▶ [The Applying a 2D texture recipe](#)

Using alpha maps to discard pixels

To create the effect of an object that has holes, we could use a texture with an appropriate alpha channel that contains information about the transparent parts of the object. However, that requires us to make sure to make the depth buffer read-only, and render all of our polygons from back to front in order to avoid blending problems. We would need to sort our polygons based on the camera position and then render them in the correct order. What a pain!

With GLSL shaders, we can avoid all of this by using the `discard` keyword to completely discard fragments when the alpha value of the texture map is below a certain value. By completely discarding the fragments, there's no need to modify the depth buffer because when discarded, they aren't evaluated against the depth buffer at all. We don't need to depth-sort our polygons because there is no blending.

The following image on the right shows the teapot with fragments discarded based upon the texture on the left. The fragment shader discards fragments that correspond to texels that have an alpha value below a certain threshold.



If we create a texture map that has an alpha channel, we can use the value of the alpha channel to determine whether or not the fragment should be discarded. If the alpha value is below a certain value, then the pixel is discarded.

As this will allow the viewer to see within the object, possibly making some back faces visible, we'll need to use two-sided lighting when rendering the object.

Getting ready

1. Start with the same shader pair and set up from the previous recipe, *Applying multiple textures*.
2. Load the base texture for the object into texture unit 0, and your alpha map into texture unit 1.

How to do it...

To discard fragments based on alpha data from a texture, use the following steps:

1. Use the same vertex and fragment shaders from the recipe *Applying multiple textures*. However, make the following modifications to the fragment shader.
2. Replace the `sampler2D` uniform variables with the following:

```
layout(binding=0) uniform sampler2D BaseTex;  
layout(binding=1) uniform sampler2D AlphaTex;
```

3. Replace the contents of the `main` function with the following code:

```
void main() {  
    vec4 baseColor = texture( BaseTex, TexCoord );  
    vec4 alphaMap = texture( AlphaTex, TexCoord );  
  
    if(alphaMap.a < 0.15 )  
        discard;  
    else {  
        if( gl_FrontFacing ) {  
            FragColor = vec4(phongModel(Position,Normal),1.0) *  
                baseColor;  
        } else {  
            FragColor = vec4(phongModel(Position,-Normal),1.0) *  
                baseColor;  
        }  
    }  
}
```

How it works...

Within the `main` function of the fragment shader, we access the base color texture, and store the result in `baseColor`. We access the alpha map texture and store the result in `alphaMap`. If the alpha component of `alphaMap` is less than a certain value (0.15 in this example), then we discard the fragment using the `discard` keyword.

Otherwise, we compute the Phong lighting model using the normal vector oriented appropriately, depending on whether or not the fragment is a front facing fragment. The result of the Phong model is multiplied by the base color from `BaseTex`.

There's more...

This technique is fairly simple and straightforward, and is a nice alternative to traditional blending techniques. It is a great way to make holes in objects or to present the appearance of decay. If your alpha map has a gradual change in the alpha throughout the map, (for example, an alpha map where the alpha values make a smoothly varying height field) then it can be used to animate the decay of an object. We could vary the alpha threshold (0.15 in the preceding example) from 0.0 to 1.0 to create an animated effect of the object gradually decaying away to nothing.

See also

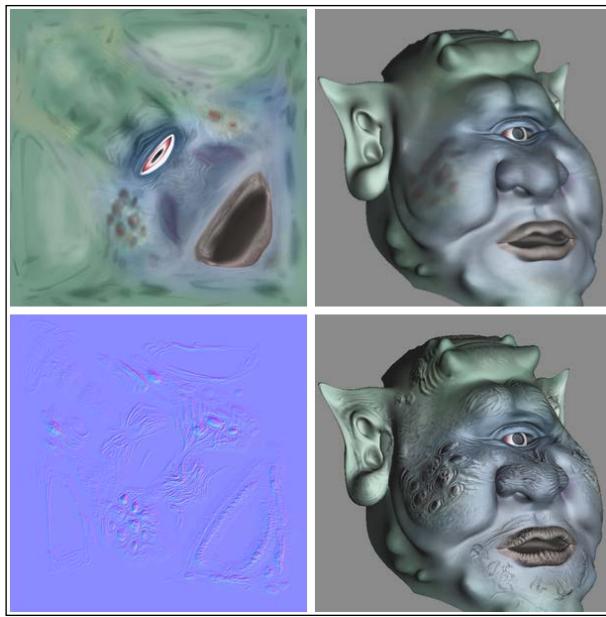
- ▶ The *Applying multiple textures* recipe

Using normal maps

Normal mapping is a technique for "faking" variations in a surface that doesn't really exist in the geometry of the surface. It is useful for producing surfaces that have bumps, dents, roughness, or wrinkles without actually providing enough position information (vertices) to fully define those deformations. The underlying surface is actually smooth, but is made to appear rough by varying the normal vectors using a texture (the normal map). The technique is closely related to bump mapping or displacement mapping. With normal maps, we modify the normal vectors based on information that is stored in a texture. This creates the appearance of a bumpy surface without actually providing the geometry of the bumps.

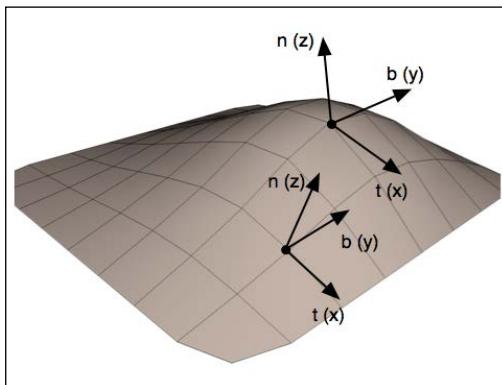
A normal map is a texture in which the data stored within the texture is interpreted as normal vectors instead of colors. The normal vectors are typically encoded into the RGB information of the normal map such that the red channel contains the x coordinate, the green channel contains the y, and the blue channel contains the z coordinate. The normal map can then be used as a "texture" in the sense that the texture values affect the normal vector used in the reflection model rather than the color of the surface. This can be used to make a surface look like it contains variations (bumps or wrinkles) that do not actually exist in the geometry of the mesh.

The following images show an ogre mesh (courtesy of Keenan Crane) with and without a normal map. The upper-left corner shows the base color texture for the ogre. In this example, we use this texture as the diffuse reflectivity in the Phong reflection model. The upper right shows the ogre with the color texture and default normal vectors. The bottom left is the normal map texture. The bottom right shows the ogre with the color texture and normal map. Note the additional detail in the wrinkles provided by the normal map.



A normal map can be produced in a number of ways. Many 3D modeling programs such as Maya, Blender, or 3D Studio Max can generate normal maps. Normal maps can also be generated directly from grayscale heightmap textures. There is a NVIDIA plugin for Adobe Photoshop that provides this functionality (see http://developer.nvidia.com/object/photoshop_dds_plugins.html).

Normal maps are interpreted as vectors in a **tangent space** (also called the **object local coordinate system**). In the tangent coordinate system, the origin is located at the surface point and the normal to the surface is aligned with the z axis (0, 0, 1). Therefore, the x and y axes are at a tangent to the surface. The following image shows an example of the tangent frames at two different positions on a surface.



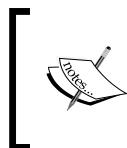
The advantage of using such a coordinate system lies in the fact that the normal vectors stored within the normal map can be treated as perturbations to the true normal, and are independent of the object coordinate system. This saves us the need to transform the normals, add the perturbed normal, and renormalize. Instead, we can use the value in the normal map directly in the reflection model without any modification.

To make all of this work, we need to evaluate the reflection model in tangent space. In order to do so, we transform the vectors used in our reflection model into tangent space in the vertex shader, and then pass them along to the fragment shader where the reflection model will be evaluated. To define a transformation from the camera (eye) coordinate system to the tangent space coordinate system, we need three normalized, co-orthogonal vectors (defined in eye coordinates) that define the tangent space system. The z axis is defined by the normal vector (n), the x axis is defined by a vector called the *tangent vector* (t), and the y axis is often called the *binormal vector* (b). A point P , defined in eye coordinates, could then be transformed into tangent space by multiplying by the following matrix:

$$\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

In the preceding equation, S is the point in tangent space and P is the point in eye coordinates. In order to apply this transformation within the vertex shader, the OpenGL program must provide at least two of the three vectors that define the object local system along with the vertex position. The usual situation is to provide the normal vector (n) and the tangent vector (t). If the tangent vector is provided, the binormal vector can be computed as the cross product of the tangent and normal vectors.

Tangent vectors are sometimes included as additional data in mesh data structures. If the tangent data is not available, we can approximate the tangent vectors by deriving them from the variation of the texture coordinates across the surface (see *Computing Tangent Space Basis Vectors for an Arbitrary Mesh*, Eric Lengyel, Terathon Software 3D Graphics Library, 2001, at <http://www.terathon.com/code/tangent.html>).



One must take care that the tangent vectors are consistently defined across the surface. In other words, the direction of the tangent vectors should not vary greatly from one vertex to its neighboring vertex. Otherwise, it can lead to ugly shading artifacts.

In the following example, we'll read the vertex position, normal vector, tangent vector, and texture coordinate in the vertex shader. We'll transform the position, normal, and tangent to eye space, and then compute the binormal vector (in eye space). Next, we'll compute the viewing direction (v) and the direction towards the light source (s) in eye space, and then transform them to tangent space. We'll pass the tangent space v and s vectors and the (unchanged) texture coordinate to the fragment shader, where we'll evaluate the Phong reflection model, using the tangent space vectors and the normal vector retrieved from the normal map.

Getting ready

Set up your OpenGL program to provide the position in attribute location 0, the normal in attribute location 1, the texture coordinate in location 2, and the tangent vector in location 3. For this example, the fourth coordinate of the tangent vector should contain the "handedness" of the tangent coordinate system (either -1 or +1). This value will be multiplied by the result of the cross product.

Load the normal map into texture unit one and the color texture into texture unit zero.

How to do it...

To render an image using normal mapping, use the following shaders:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
layout (location = 3) in vec4 VertexTangent;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // A,D,S intensity
};
uniform LightInfo Light;

out vec3 LightDir;
out vec2 TexCoord;
out vec3 ViewDir;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    // Transform normal and tangent to eye space
    vec3 norm = normalize(NormalMatrix * VertexNormal);
    vec3 tang = normalize(NormalMatrix *
                          vec3(VertexTangent));
    // Compute the binormal
    vec3 binormal = normalize( cross( norm, tang ) ) *
```

```
VertexTangent.w;
// Matrix for transformation to tangent space
mat3 toObjectLocal = mat3(
    tang.x, binormal.x, norm.x,
    tang.y, binormal.y, norm.y,
    tang.z, binormal.z, norm.z ) ;
// Get the position in eye coordinates
vec3 pos = vec3( ModelViewMatrix *
                 vec4(VertexPosition,1.0) ) ;

// Transform light dir. and view dir. to tangent space
LightDir = normalize( toObjectLocal *
                       (Light.Position.xyz - pos) );
ViewDir = toObjectLocal * normalize(-pos);

// Pass along the texture coordinate
TexCoord = VertexTexCoord;

gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 LightDir;
in vec2 TexCoord;
in vec3 ViewDir;

layout(binding=0) uniform sampler2D ColorTex;
layout(binding=1) uniform sampler2D NormalMapTex;

struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // A,D,S intensity
};
uniform LightInfo Light;

struct MaterialInfo {
    vec3 Ka; // Ambient reflectivity
    vec3 Ks; // Specular reflectivity
    float Shininess; // Specular shininess factor
};
uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;
```

```
vec3 phongModel( vec3 norm, vec3 diffR ) {
    vec3 r = reflect( -LightDir, norm );
    vec3 ambient = Light.Intensity * Material.Ka;
    float sDotN = max( dot(LightDir, norm), 0.0 );
    vec3 diffuse = Light.Intensity * diffR * sDotN;

    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Intensity * Material.Ks *
            pow( max( dot(r,ViewDir), 0.0 ),
                Material.Shininess );

    return ambient + diffuse + spec;
}

void main() {
    // Lookup the normal from the normal map
    vec4 normal = 2.0 * texture( NormalMapTex, TexCoord ) -
        1.0;

    // The color texture is used as the diff. reflectivity
    vec4 texColor = texture( ColorTex, TexCoord );

    FragColor = vec4( phongModel(normal.xyz, texColor.rgb),
        1.0 );
}
```

How it works...

The vertex shader starts by transforming the vertex normal and the tangent vectors into eye coordinates by multiplying by the normal matrix (and renormalizing). The binormal vector is then computed as the cross product of the normal and tangent vectors. The result is multiplied by the w coordinate of the vertex tangent vector, which determines the handedness of the tangent space coordinate system. Its value will be either -1 or +1.

Next, we create the transformation matrix used to convert from eye coordinates to tangent space and store the matrix in `toObjectLocal`. The position is converted to eye space and stored in `pos`, and we compute the light direction by subtracting `pos` from the light position. The result is multiplied by `toObjectLocal` to convert it into tangent space, and the final result is normalized and stored in the output variable `LightDir`. This value is the direction to the light source in tangent space, and will be used by the fragment shader in the Phong reflection model.

Similarly, the view direction is computed and converted to tangent space by normalizing `-pos` and multiplying by `toObjectLocal`. The result is stored in the output variable `ViewDir`.

The texture coordinate is passed to the fragment shader unchanged by just assigning it to the output variable `TexCoord`.

In the fragment shader, the tangent space values for the light direction and view direction are received in the variables `LightDir` and `ViewDir`. The `phongModel` function is slightly modified from what has been used in previous recipes. The first parameter is the normal vector, and the second is the diffuse reflectivity coefficient. The value for this will be taken from the color texture. The function computes the Phong reflection model with the parameter `diffR`, used as the diffuse reflectivity, and uses `LightDir` and `ViewDir` for the light and view directions rather than computing them.

In the main function, the normal vector is retrieved from the normal map texture and stored in the variable `normal`. Since textures store values that range from zero to one, and normal vectors should have components that range from -1 to +1, we need to re-scale the value to that range. We do so by multiplying the value by 2.0, and then subtracting 1.0.

The color texture is then accessed to retrieve the color to be used as the diffuse reflectivity coefficient, and the result is stored in `texColor`. Finally, the `phongModel` function is called, and is provided `normal` and `texColor`. The `phongModel` function evaluates the Phong reflection model using `LightDir`, `ViewDir`, and `norm`, all of which are defined in tangent space. The result is applied to the output fragment by assigning it to `FragColor`.

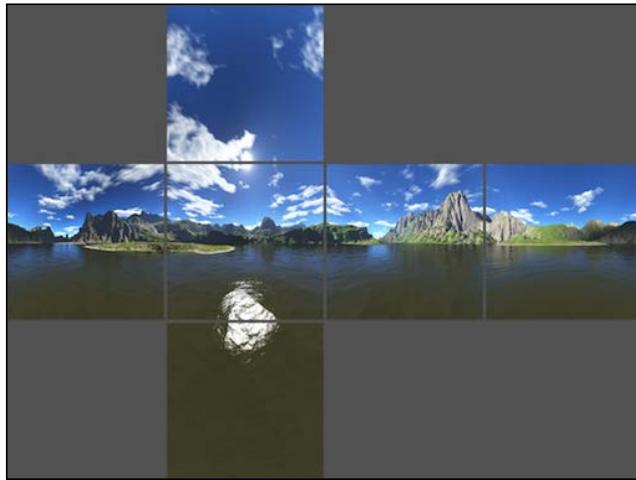
See also

- ▶ The [Applying multiple textures](#) recipe
- ▶ The [Implementing per-vertex ambient, diffuse, and specular \(ADS\) shading](#) recipe in Chapter 2, *The Basics of GLSL Shaders*

Simulating reflection with cube maps

Textures can be used to simulate a surface that has a component which is purely reflective (a mirror-like surface such as chrome). In order to do so, we need a texture that is representative of the environment surrounding the reflective object. This texture could then be mapped onto the surface of the object in a way that represents how it would look when reflected off of the surface. This general technique is known as **environment mapping**. In general, environment mapping involves creating a texture that is representative of the environment and mapping it onto the surface of an object. It is typically used to simulate the effects of reflection or refraction.

A **cube map** is one of the more common varieties of textures used in environment mapping. A cube map is a set of six separate images that represent the environment projected onto each of the six faces of a cube. The six images represent a view of the environment from the point of view of a viewer located at the center of the cube. An example of a cube map is shown in the following image. The images are laid out as if the cube was "unfolded" and laid flat. The four images across the middle would make up the sides of the cube, and the top and bottom images correspond to the top and bottom of the cube.



OpenGL provides built-in support for cube map textures (using the `GL_TEXTURE_CUBE_MAP` target). The texture is accessed using a 3-dimensional texture coordinate (s, t, r). The texture coordinate is interpreted as a direction vector from the center of the cube. The line defined by the vector and the center of the cube is extended to intersect one of the faces of the cube. The image that corresponds to that face is then accessed at the location of the intersection.



Truth be told, the conversion between the 3-dimensional texture coordinate used to access the cube map, and the 2-dimensional texture coordinate used to access the individual face image is somewhat complicated. It can be non-intuitive and confusing. A very good explanation can be found on NVIDIA's developer website: <http://developer.nvidia.com/content/cube-map-ogl-tutorial>. However, the good news is that if you are careful to orient your textures correctly within the cube map, the details of the conversion can be ignored, and the texture coordinate can be visualized as a 3-dimensional vector as described previously.

In this example, we'll demonstrate using a cube map to simulate a reflective surface. We'll also use the cube map to draw the environment around the reflective object (sometimes called a **skybox**).

Getting ready

Prepare the six images of the cube map. In this example, the images will have the following naming convention. There is a base name (stored in variable `baseFileName`) followed by an underscore, followed by one of the six possible suffixes (`posx`, `negx`, `posy`, `negy`, `posz`, or `negz`), followed by the file extension (`.tga`). The suffixes `posx`, `posy`, and so on, indicate the axis that goes through the center of the face (positive `x`, positive `y`, and so on).

Make sure that they are all square images (preferably with dimensions that are a power of 2), and that they are all the same size. You will need to orient them appropriately for the way that OpenGL accesses them. As mentioned previously, this can be a bit tricky. One way to do this is to load the textures in their default orientation and draw the sky box (more on how to do that follows). Then re-orient the textures (by trial and error) until they line up correctly. Alternatively, take a close look at the conversion described in the NVIDIA link mentioned in the previous tip and determine the proper orientation based on the texture coordinate conversions.

Set up your OpenGL program to provide the vertex position in attribute location 0, and the vertex normal in attribute location 1.

This vertex shader requires the modeling matrix (the matrix that converts from object coordinates to world coordinates) to be separated from the model-view matrix and provided to the shader as a separate uniform. Your OpenGL program should provide the modeling matrix in the uniform variable `ModelMatrix`.

The vertex shader also requires the location of the camera in world coordinates. Make sure that your OpenGL program sets the uniform `WorldCameraPosition` to the appropriate value.

How to do it...

To render an image with reflection based on a cube map, and also render the cube map itself, carry out the following steps:

1. Load the six images of the cube map into a single texture target using the following code within the main OpenGL program:

```
glActiveTexture(GL_TEXTURE0);

GLuint texID;
 glGenTextures(1, &texID);
 glBindTexture(GL_TEXTURE_CUBE_MAP, texID);

const char * suffixes[] = { "posx", "negx", "posy",
                           "negy", "posz", "negz" };

GLuint targets[] = {
    GL_TEXTURE_CUBE_MAP_POSITIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
```

```
GL_TEXTURE_CUBE_MAP_POSITIVE_Y,  
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,  
GL_TEXTURE_CUBE_MAP_POSITIVE_Z,  
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z  
};  
GLint w,h;  
glTexStorage2D(GL_TEXTURE_CUBE_MAP, 1, GL_RGBA8, 256, 256);  
for( int i = 0; i < 6; i++ ) {  
    string texName = string(baseFileName) +  
        "_" + suffixes[i] + ".tga";  
    GLubyte *data = TGAIO::read(texName.c_str(), w, h);  
    glTexSubImage2D(targets[i], 0, 0, 0, w, h,  
                    GL_RGBA, GL_UNSIGNED_BYTE, data);  
    delete [] data;  
}  
  
// Typical cube map settings  
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,  
                 GL_LINEAR);  
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,  
                 GL_LINEAR);  
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,  
                 GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,  
                 GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,  
                 GL_CLAMP_TO_EDGE);
```

2. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;  
layout (location = 1) in vec3 VertexNormal;  
layout (location = 2) in vec2 VertexTexCoord;  
  
out vec3 ReflectDir; // The direction of the reflected ray  
uniform bool DrawSkyBox; // Are we drawing the sky box?  
uniform vec3 WorldCameraPosition;  
uniform mat4 ModelViewMatrix;  
uniform mat4 ModelMatrix;  
uniform mat3 NormalMatrix;  
uniform mat4 ProjectionMatrix;  
uniform mat4 MVP;
```

```
void main()
{
    if( DrawSkyBox ) {
        ReflectDir = VertexPosition;
    } else {

        // Compute the reflected direction in world coords.
        vec3 worldPos = vec3( ModelMatrix *
                            vec4(VertexPosition,1.0) );
        vec3 worldNorm = vec3(ModelMatrix *
                            vec4(VertexNormal, 0.0));
        vec3 worldView = normalize( WorldCameraPosition -
                                    worldPos );

        ReflectDir = reflect(-worldView, worldNorm );
    }

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Use the following code for the fragment shader:

```
in vec3 ReflectDir;    // The direction of the reflected ray

// The cube map
layout(binding=0) uniform samplerCube CubeMapTex;

uniform bool DrawSkyBox;    // Are we drawing the sky box?
uniform float ReflectFactor;// Amount of reflection
uniform vec4 MaterialColor; // Color of the object's "Tint"

layout( location = 0 ) out vec4 FragColor;

void main() {
    // Access the cube map texture
    vec4 cubeMapColor = texture(CubeMapTex, ReflectDir);
    if( DrawSkyBox )
        FragColor = cubeMapColor;
    else
        FragColor = mix(MaterialColor, CubeMapColor, ReflectFactor);

}
```

4. In the render portion of the OpenGL program, set the uniform `DrawSkyBox` to true, and then draw a cube surrounding the entire scene, centered at the origin. This will become the sky box. Following that, set `DrawSkyBox` to false, and draw the object(s) within the scene.

How it works...

In OpenGL, a cube map texture is actually six separate images. To fully initialize a cube map texture, we need to bind to the cube map texture, and then load each image individually into the six "slots" within that texture. In the preceding code (within the main OpenGL application), we start by binding to texture unit zero with `glActiveTexture`. Then we create a new texture object by calling `glGenTextures`, and store its handle within the variable `texID`, and then bind that texture object to the `GL_TEXTURE_CUBE_MAP` target using `glBindTexture`. The following loop loads each texture file, and copies the texture data into OpenGL memory using `glTexSubImage2D`. Note that the first argument to this function is the texture target, which corresponds to `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, and so on. After the loop is finished, the cube map texture should be fully initialized with the six images.

Following this, we set up the cube map texture environment. We use linear filtering, and we also set the texture wrap mode to `GL_CLAMP_TO_EDGE` for all three of the texture coordinate's components. This tends to work the best, avoiding the possibility of a border color appearing between the cube edges.

Within the vertex shader, the main goal is to compute the direction of reflection and pass that to the fragment shader to be used to access the cube map. The output variable `ReflectDir` will store this result. If we are not drawing the sky box (the value of `DrawSkyBox` is false), then we can compute the reflected direction (in world coordinates) by reflecting the vector towards the viewer about the normal vector.



We choose to compute the reflection direction in world coordinates because, if we were to use eye coordinates, the reflection would not change as the camera moved within the scene.

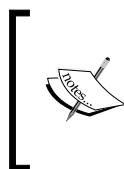


In the `else` branch within the main function, we start by converting the position to world coordinates and storing in `worldPos`. We then do the same for the normal, storing the result in `worldNorm`. Note that the `ModelMatrix` is used to transform the vertex normal. It is important when doing this to use a value of 0.0 for the fourth coordinate of the normal, to avoid the translation component of the model matrix affecting the normal. Also, the model matrix must not contain any non-uniform scaling component; otherwise the normal vector will be transformed incorrectly.

The direction towards the viewer is computed in world coordinates and stored in `worldView`.

Finally, we reflect `worldView` about the normal and store the result in the output variable `ReflectDir`. The fragment shader will use this direction to access the cube map texture and apply the corresponding color to the fragment. One can think of this as a light ray that begins at the viewer's eye, strikes the surface, reflects off of the surface, and hits the cube map. The color that the ray "sees" when it strikes the cube map is the color that we need for the object.

If we are drawing the sky box, (`DrawSkyBox` is `true`), then we use the vertex position as the reflection direction. Why? Well, when the sky box is rendered, we want the location on the sky box to correspond to the equivalent location in the cube map (the sky box is really just a rendering of the cube map). In the fragment shader, `ReflectDir` will be used as the texture coordinate to access the cube map. Therefore, if we want to access a position on the cube map corresponding to a location on a cube centered at the origin, we need a vector that points at that location. The vector we need is the position of that point minus the origin (which is $(0,0,0)$). Hence, we just need the position of the vertex.



Sky boxes are often rendered with the viewer at the center of the sky box and the sky box moving along with the viewer (so the viewer is always at the center of the sky box). We have not done so in this example; however, we could do so by transforming the sky box using the rotational component of the view matrix (not the translational).

Within the fragment shader, we simply use the value of `ReflectDir` to access the cube map texture.

```
vec4 cubeMapColor = texture(CubeMapTex, ReflectDir)
```

If we are drawing the sky box, we simply use the color unchanged. However, if we are not drawing the sky box, then we'll mix the sky box color with some material color. This allows us to provide some slight "tint" to the object. The amount of tint is adjusted by the variable `ReflectFactor`. A value of 1.0 would correspond to zero tint (all reflection), and a value of 0.0 corresponds to no reflection. The following images show the teapot rendered with different values of `ReflectFactor`. The teapot on the left uses a reflection factor of 0.5, the one on the right uses a value of 0.85. The base material color is grey. (Cube map used is an image of St. Peter's Basilica, Rome. ©Paul Debevec.)



There's more...

There are two important points to keep in mind about this technique. First, the objects will only reflect the environment map. They will not reflect the image of any other objects within the scene. In order to do so, we would need to generate an environment map from the point of view of each object by rendering the scene six times with the view point located at the center of the object and the view direction in each of the six coordinate directions. Then we could use the appropriate environment map for the appropriate object's reflections. Of course, if any of the objects were to move relative to one another, we'd need to regenerate the environment maps. All of this effort may be prohibitive in an interactive application.

The second point involves the reflections that appear on moving objects. In these shaders, we compute the reflection direction and treat it as a vector emanating from the center of the environment map. This means that regardless of where the object is located, the reflections will appear as if the object is in the center of the environment. In other words, the environment is treated as if it were "infinitely" far away. *Chapter 19* of the book *GPU Gems*, by Randima Fernando, Addison-Wesley Professional, 2009 has an excellent discussion of this issue and provides some possible solutions for localizing the reflections.

See also

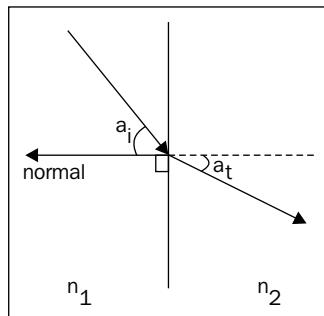
- ▶ The Applying a 2D texture recipe

Simulating refraction with cube maps

Objects that are transparent cause the light rays that pass through them to bend slightly at the interface between the object and the surrounding environment. This effect is called **refraction**. When rendering transparent objects, we simulate that effect by using an environment map, and mapping the environment onto the object in such a way as to mimic the way that light would pass through the object. In other words, we can trace the rays from the viewer, through the object (bending in the process), and along to the environment. Then we can use that ray intersection as the color for the object.

As in the previous recipe, we'll do this using a cube map for the environment. We'll trace rays from the viewer position, through the object, and finally intersect with the cube map.

The process of refraction is described by **Snell's law**, which defines the relationship between the angle of incidence and the angle of refraction.



Snell's law describes the angle of incidence (a_i) as the angle between the incoming light ray and the normal to the surface, and the angle of refraction (a_t) as the angle between the transmitted ray and the extended normal. The material through which the incident light ray travels and the material containing the transmitted light ray are each described by an index of refraction (n_1 and n_2 in the figure). The ratio between the two indices of refraction defines the amount that the light ray will be bent at the interface.

Starting with Snell's law, and with a bit of mathematical effort, we can derive a formula for the transmitted vector, given the ratio of the indices of refraction, the normal vector, and the incoming vector.

$$\frac{\sin a_i}{\sin a_t} = \frac{n_2}{n_1}$$

However, there's no real need to do so, because GLSL provides a built-in function for computing this transmitted vector called `refract`. We'll make use of that function within this example.

It is usually the case that for transparent objects, not all of the light is transmitted through the surface. Some of the light is reflected. In this example, we'll model that in a very simple way, and at the end of this recipe we'll discuss a more accurate representation.

Getting ready

Set up your OpenGL program to provide the vertex position in attribute location 0 and the vertex normal in attribute location 1. As with the previous recipe, we'll need to provide the model matrix in the uniform variable `ModelMatrix`.

Load the cube map using the technique shown in the previous recipe. Place it in texture unit zero.

Set the uniform variable `WorldCameraPosition` to the location of your viewer in world coordinates. Set the value of the uniform variable `Material.Eta` to the ratio between the index of refraction of the environment `n1` and the index of refraction of the material `n2` ($n1/n2$). Set the value of the uniform `Material.ReflectionFactor` to the fraction of light that is reflected at the interface (a small value is probably what you want).

As with the preceding example, if you want to draw the environment, set the uniform variable `DrawSkyBox` to `true`, then draw a large cube surrounding the scene, and then set `DrawSkyBox` to `false`.

How to do it...

To render an object with reflection and refraction as well as the cube map itself, carry out the following steps:

1. Use the following code within the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 ReflectDir; // Reflected direction
out vec3 RefractDir; // Transmitted direction

struct MaterialInfo {
    float Eta;           // Ratio of indices of refraction
    float ReflectionFactor; // Percentage of reflected light
};

uniform MaterialInfo Material;

uniform bool DrawSkyBox;

uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    if( DrawSkyBox ) {
        ReflectDir = VertexPosition;
    } else {
        vec3 worldPos = vec3( ModelMatrix *
```

```
        vec4(VertexPosition,1.0) );
vec3 worldNorm = vec3(ModelMatrix *
                      vec4(VertexNormal, 0.0));
vec3 worldView = normalize( WorldCameraPosition -
                            worldPos );

ReflectDir = reflect(-worldView, worldNorm );
RefractDir = refract(-worldView, worldNorm,
                     Material.Eta );
}
gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code within the fragment shader:

```
in vec3 ReflectDir;
in vec3 RefractDir;

layout(binding=0) uniform samplerCube CubeMapTex;
uniform bool DrawSkyBox;
struct MaterialInfo {
    float Eta; // Ratio of indices of refraction
    float ReflectionFactor; // Percentage of reflected light
};
uniform MaterialInfo Material;

layout( location = 0 ) out vec4 FragColor;

void main() {
    // Access the cube map texture
    vec4 reflectColor = texture(CubeMapTex, ReflectDir);
    vec4 refractColor = texture(CubeMapTex, RefractDir);

    if( DrawSkyBox )
        FragColor = reflectColor;
    else
        FragColor = mix(refractColor, reflectColor,
                        Material.ReflectionFactor);
}
```

3. In the render portion of the OpenGL program, set the uniform `DrawSkyBox` to true, and then draw a cube surrounding the entire scene, centered at the origin. This will become the sky box. Following that, set `DrawSkyBox` to false, and draw the object(s) within the scene.

How it works...

Both shaders are quite similar to the shaders in the previous recipe.

The vertex shader computes the position, normal, and view direction in world coordinates (`worldPos`, `worldNorm`, and `worldView`). They are then used to compute the reflected direction using the `reflect` function, and the result is stored in the output variable `ReflectDir`. The transmitted direction is computed using the built-in function `refract` (which requires the ratio of the indices of refraction `Material.Eta`). This function makes use of Snell's law to compute the direction of the transmitted vector which is then stored in the output variable `RefractDir`.

In the fragment shader, we use the two vectors `ReflectDir` and `RefractDir` to access the cube map texture. The color retrieved by the reflected ray is stored in `reflectColor` and the color retrieved by the transmitted ray is stored in `refractColor`. We then mix those two colors together based on the value of `Material.ReflectionFactor`. The result is a mixture between the color of the reflected ray and the color of the transmitted ray.

The following image shows the teapot rendered with 10% reflection and 90% refraction.
(Cubemap © Paul Debevec.)



There's more...

This technique has the same drawbacks that were discussed in the *There's more...* section of the preceding recipe, *Simulating reflection with cube maps*.

Like most real-time techniques, this is a simplification of the real physics of the situation. There are a number of things about the technique that could be improved to provide more realistic looking results.

The Fresnel equations

The amount of reflected light actually depends on the angle of incidence of the incoming light. For example, when looking at the surface of a lake from the shore, much of the light is reflected and it is easy to see reflections of the surrounding environment on the surface. However, when floating on a boat on the surface of the lake and looking straight down, there is less reflection and it is easier to see what lies below the surface. This effect is described by the Fresnel equations (after Augustin-Jean Fresnel).

The Fresnel equations describe the amount of light that is reflected as a function of the angle of incidence, the polarization of the light, and the ratio of the indices of refraction. If we ignore the polarization, it is easy to incorporate the Fresnel equations into the preceding shaders. A very good explanation of this can be found in the book *The OpenGL Shading Language, 3rd Edition*, Randi J Rost, Addison-Wesley Professional, 2009.

Chromatic aberration

White light is of course composed of many different individual wavelengths (or colors). The amount that a light ray is refracted is actually wavelength dependent. This causes the effect where a spectrum of colors can be observed at the interface between materials. The most well-known example of this is the rainbow that is produced by a prism.

We can model this effect by using slightly different values of Eta for the red, green, and blue components of the light ray. We would store three different values for Eta , compute three different reflection directions (red, green, and blue), and use those three directions to look up colors in the cube map. We take the red component from the first color, the green component from the second, and the blue component for the third, and combine the three components together to create the final color for the fragment.

Refracting through both sides of the object

It is important to note that we have simplified things by only modeling the interaction of the light with one of the boundaries of the object. In reality the light would be bent once when entering the transparent object, and again when leaving the other side. However, this simplification generally does not result in unrealistic looking results. As is often the case in real-time graphics, we are more interested in a result that looks good than one that models the physics accurately.

See also

- ▶ The *Simulating reflection with cube maps* recipe

Applying a projected texture

We can apply a texture to the objects in a scene as if the texture was a projection from a hypothetical "slide projector" located somewhere within the scene. This technique is often called **projective texture mapping** and produces a very nice effect.

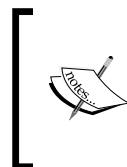
The following images show an example of projective texture mapping. The flower texture on the left (Stan Shebs via Wikimedia Commons) is projected onto the teapot and plane beneath.



To project a texture onto a surface, all we need to do is determine the texture coordinates based on the relative position of the surface location and the source of the projection (the "slide projector"). An easy way to do this is to think of the projector as a camera located somewhere within the scene. In the same way that we would define an OpenGL camera, we define a coordinate system centered at the projector's location, and a **view matrix (V)** that converts coordinates to the projector's coordinate system. Next, we'll define a perspective **projection matrix (P)** that converts the view frustum (in the projector's coordinate system) into a cubic volume of size 2, centered at the origin. Putting these two things together, and adding an additional matrix for rescaling and translating the volume to a volume of size one (shifted so that the volume is centered at (0.5, 0.5, 0.5), we have the following transformation matrix:

$$\mathbf{M} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{PV}$$

The goal here is basically to convert the view frustum to a range between 0 and 1 in x and y. The preceding matrix can be used to do just that! It will convert world coordinates that lie within the view frustum of the projector to a range between 0 and 1 (homogeneous), which can then be used to access the texture. Note that the coordinates are homogeneous and need to be divided by the w coordinate before they can be used as a real position.



For more details on the mathematics of this technique, take a look at the following white paper, written by Cass Everitt from NVIDIA:
<http://developer.nvidia.com/content/projective-texture-mapping>

In this example, we'll apply a single texture to a scene using projective texture mapping.

Getting ready

Set up your OpenGL application to provide the vertex position in attribute location 0 and the normal in attribute location 1. The OpenGL application must also provide the material and lighting properties for the Phong reflection model (see the fragment shader given in the following section). Make sure to provide the model matrix (for converting to world coordinates) in the uniform variable `ModelMatrix`.

How to do it...

To apply a projected texture to a scene, use the following steps:

1. In the OpenGL application, load the texture into texture unit zero. While the texture object is bound to the `GL_TEXTURE_2D` target, use the following code to set the texture's settings:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                 GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                 GL_CLAMP_TO_BORDER);
```

2. Also within the OpenGL application, set up your transformation matrix for the "slide projector", and assign it to the uniform `ProjectorMatrix`. Use the following code to do this. Note that this code makes use of the GLM libraries discussed in *Chapter 1, Getting Started with GLSL*.

```
vec3 projPos = vec3(2.0f, 5.0f, 5.0f);
vec3 projAt = vec3(-2.0f, -4.0f, 0.0f);
vec3 projUp = vec3(0.0f, 1.0f, 0.0f);

mat4 projView = glm::lookAt(projPos, projAt, projUp);
mat4 projProj = glm::perspective(30.0f, 1.0f, 0.2f,
                                 1000.0f);
```

```
mat4 projScaleTrans = glm::translate(vec3(0.5f)) *
    glm::scale(vec3(0.5f));

mat4 m = projScaleTrans * projProj * projView;

// Set the uniform variable
int loc =
    glGetUniformLocation(progHandle, "ProjectorMatrix");
 glUniformMatrix4fv(loc, 1, GL_FALSE, &m[0][0]);

3. Use the following code for the vertex shader:

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 EyeNormal;           // Normal in eye coordinates
out vec4 EyePosition;         // Position in eye coordinates
out vec4 ProjTexCoord;

uniform mat4 ProjectorMatrix;
uniform vec3 WorldCameraPosition;
uniform mat4 ModelViewMatrix;
uniform mat4 ModelMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    vec4 pos4 = vec4(VertexPosition, 1.0);

    EyeNormal = normalize(NormalMatrix * VertexNormal);
    EyePosition = ModelViewMatrix * pos4;
    ProjTexCoord = ProjectorMatrix * (ModelMatrix * pos4);
    gl_Position = MVP * pos4;
}

4. Use the following code for the fragment shader:

in vec3 EyeNormal;           // Normal in eye coordinates
in vec4 EyePosition;          // Position in eye coordinates
in vec4 ProjTexCoord;

layout(binding=0) uniform sampler2D ProjectorTex;
```

```
struct MaterialInfo {
    vec3 Kd;
    vec3 Ks;
    vec3 Ka;
    float Shininess;
};

uniform MaterialInfo Material;

struct LightInfo {
    vec3 Intensity;
    vec4 Position; // Light position in eye coordinates
};
uniform LightInfo Light;

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 pos, vec3 norm ) {
    vec3 s = normalize(vec3(Light.Position) - pos);
    vec3 v = normalize(-pos.xyz);
    vec3 r = reflect( -s, norm );
    vec3 ambient = Light.Intensity * Material.Ka;
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Intensity * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Intensity * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess);

    return ambient + diffuse + spec;
}

void main() {
    vec3 color = phongModel(vec3(EyePosition), EyeNormal);

    vec4 projTexColor = vec4(0.0);
    if( ProjTexCoord.z > 0.0 )
        projTexColor = textureProj(ProjectorTex,ProjTexCoord);

    FragColor = vec4(color,1.0) + projTexColor * 0.5;
}
```

How it works...

When loading the texture into the OpenGL application, we make sure to set the wrap mode for the s and t directions to `GL_CLAMP_TO_BORDER`. We do this because if the texture coordinates are outside of the range of zero to one, we do not want any contribution from the projected texture. With this mode, using the default border color, the texture will return (0,0,0,0) when the texture coordinates are outside of the range between 0 and 1 inclusive.

The transformation matrix for the slide projector is set up in the OpenGL application. We start by using the GLM function `glm::lookAt` to produce a view matrix for the projector. In this example, we locate the projector at (5, 5, 5), looking towards the point (-2, -4, 0), with an "up vector" of (0, 1, 0). This function works in a similar way to the `gluLookAt` function. It returns a matrix for converting to the coordinate system located at (5, 5, 5), and oriented based on the second and third arguments.

Next, we create the projection matrix using `glm::perspective`, and the scale/translate matrix M (shown in the introduction to this recipe). These two matrices are stored in `projProj` and `projScaleTrans` respectively. The final matrix is the product of `projScaleTrans`, `projProj`, and `projView`, which is stored in `m` and assigned to the uniform variable `ProjectorTex`.

In the vertex shader, we have three output variables `EyeNormal`, `EyePosition`, and `ProjTexCoord`. The first two are the vertex normal and vertex position in eye coordinates. We transform the input variables appropriately, and assign the results to the output variables within the `main` function.

We compute `ProjTexCoord` by first transforming the position to world coordinates (by multiplying by `ModelMatrix`), and then applying the projector's transformation.

In the fragment shader, within the `main` function, we start by computing the Phong reflection model and storing the result in the variable `color`. The next step is to look up the color from the texture. First, however, we check the z coordinate of `ProjTexCoord`. If this is negative then the location is behind the projector, so we avoid doing the texture lookup. Otherwise we use `textureProj` to look up the texture value and store it in `projTextColor`.

The function `textureProj` is designed for accessing textures with coordinates that have been projected. It will divide the coordinates of the second argument by its last coordinate before accessing the texture. In our case, that is exactly what we want. We mentioned earlier that after transforming by the projector's matrix we will be left with homogeneous coordinates, so we need to divide by the w coordinate before accessing the texture. The `textureProj` function will do exactly that for us.

Finally, we add the projected texture's color to the base color from the Phong model. We scale the projected texture color slightly so that it is not overwhelming.

There's more...

There's one big drawback to the technique presented here. There is no support for shadows yet, so the projected texture will shine right through any objects in the scene and appear on objects that are behind them (with respect to the projector). In later recipes, we will look at some examples of techniques for handling shadows that could help to solve this problem.

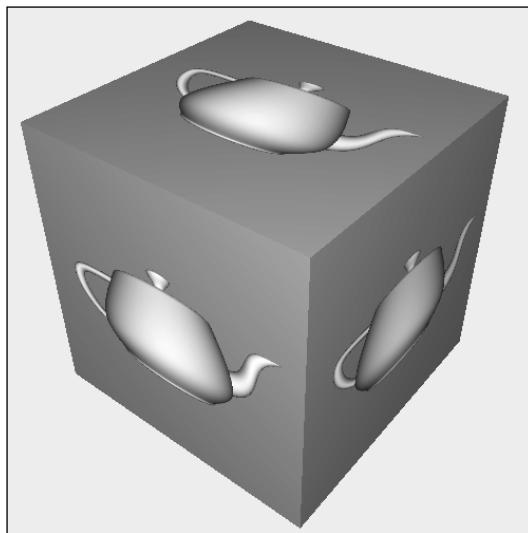
See also

- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*
- ▶ The *Applying a 2D texture* recipe

Rendering to a texture

Sometimes it makes sense to generate textures "on the fly" during the execution of the program. The texture could be a pattern that is generated from some internal algorithm (a so-called **procedural texture**), or it could be that the texture is meant to represent another portion of the scene. An example of the latter case might be a video screen where one can see another part of the "world", perhaps via a security camera in another room. The video screen could be constantly updated as objects move around in the other room, by re-rendering the view from the security camera to the texture that is applied to the video screen!

In the following image, the texture appearing on the cube was generated by rendering a teapot to an internal texture and then applying that texture to the faces of the cube.



In recent versions of OpenGL, rendering directly to textures has been greatly simplified with the introduction of **framebuffer objects (FBOs)**. We can create a separate rendering target buffer (the FBO), attach our texture to that FBO, and render to the FBO in exactly the same way that we would render to the default framebuffer. All that is required is to swap in the FBO, and swap it out when we are done.

Basically, the process involves the following steps when rendering:

1. Bind to the FBO.
2. Render the texture.
3. Unbind from the FBO (back to the default framebuffer).
4. Render the scene using the texture.

There's actually not much that we need to do on the GLSL side in order to use this kind of texture. In fact, the shaders will see it as any other texture. However, there are some important points that we'll talk about regarding fragment output variables.

In this example, we'll cover the steps needed to create the FBO and its backing texture, and how to set up a shader to work with the texture.

Getting ready

For this example, we'll use the shaders from the previous recipe *Applying a 2D texture*, with some minor changes. Set up your OpenGL program as described in that recipe. The only change that we'll make to the shaders is changing the name of the `sampler2D` variable from `Tex1` to `Texture`.

How to do it...

To render to a texture and then apply that texture to a scene in a second pass, use the following steps:

1. Within the main OpenGL program, use the following code to set up the framebuffer object:

```
GLuint fboHandle; // The handle to the FBO

// Generate and bind the framebuffer
glGenFramebuffers(1, &fboHandle);
 glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

// Create the texture object
GLuint renderTex;
 glGenTextures(1, &renderTex);
 glBindTexture(GL_TEXTURE_2D, renderTex);
 glActiveTexture(GL_TEXTURE0); // Use texture unit 0
```

```
glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 512, 512);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);

// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, renderTex, 0);

// Create the depth buffer
GLuint depthBuf;
 glGenRenderbuffers(1, &depthBuf);
 glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      512, 512);

// Bind the depth buffer to the FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);

// Set the target for the fragment shader outputs
GLenum drawBufs[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBufs);

// Unbind the framebuffer, and revert to default
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

2. Use the following code to create a simple 1×1 texture that can be used as a "non-texture texture". Note that we place this one in texture unit 1:

```
// One pixel white texture
GLuint whiteTexHandle;
GLubyte whiteTex[] = { 255, 255, 255, 255 };
glActiveTexture(GL_TEXTURE1);
 glGenTextures(1, &whiteTexHandle);
 glBindTexture(GL_TEXTURE_2D, whiteTexHandle);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, 1, 1);
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 1, 1, GL_RGBA,
                 GL_UNSIGNED_BYTE, whiteTex);
```

3. In your render function within the OpenGL program, use the following code, or something similar:

```
// Bind to texture's FBO
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
glViewport(0, 0, 512, 512); // Viewport for the texture
```

```
// Use the "white" texture here
int loc = glGetUniformLocation(programHandle, "Texture");
glUniform1i(loc, 1);

// Setup the projection matrix and view matrix
// for the scene to be rendered to the texture here.
// (Don't forget to match aspect ratio of the viewport.)

renderTextureScene();

// Unbind texture's FBO (back to default FB)
 glBindFramebuffer(GL_FRAMEBUFFER, 0);
 glViewport(0,0,width,height); // Viewport for main window

// Use the texture that is associated with the FBO
int loc = glGetUniformLocation(programHandle, "Texture");
glUniform1i(loc, 0);

// Reset projection and view matrices here

renderScene();
```

How it works...

Let's start by looking at the code for creating the framebuffer object (the preceding step 1). Our FBO will be 512 pixels square because we intend to use it as a texture. We begin by generating the FBO using `glGenFramebuffers` and binding the framebuffer to the `GL_FRAMEBUFFER` target with `glBindFramebuffer`. Next, we create the texture object to which we will be rendering, and use `glActiveTexture` to select texture unit zero. The rest is very similar to creating any other texture. We allocate space for the texture using `glTexStorage2D`. We don't need to copy any data into that space (using `glTexSubImage2D`), because we'll be writing to that memory later when rendering to the FBO.

Next, we link the texture to the FBO by calling the function `glFramebufferTexture2D`. This function attaches a texture object to an attachment point in the currently bound framebuffer object. The first argument (`GL_FRAMEBUFFER`) indicates that the texture is to be attached to the FBO currently bound to the `GL_FRAMEBUFFER` target. The second argument is the attachment point. Framebuffer objects have several attachment points for color buffers, one for the depth buffer, and a few others. This allows us to have several color buffers to target from our fragment shaders. We'll see more about this later. We use `GL_COLOR_ATTACHMENT0` to indicate that this texture is linked to color attachment 0 of the FBO. The third argument (`GL_TEXTURE_2D`) is the texture target, and the fourth (`renderTex`) is the handle to our texture. The last argument (0) is the mip-map level of the texture that is being attached to the FBO. In this case, we only have a single level, so we use a value of zero.

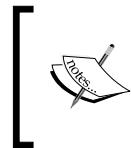
As we want to render to the FBO with depth testing, we need to also attach a depth buffer. The next few lines of code create the depth buffer. The function `glGenRenderbuffer` creates a `renderbuffer` object, and `glRenderbufferStorage` allocates space for the `renderbuffer`. The second argument to `glRenderbufferStorage` indicates the internal format for the buffer, and as we are using this as a depth buffer, we use the special format `GL_DEPTH_COMPONENT`.

Next, the depth buffer is attached to the `GL_DEPTH_ATTACHMENT` attachment point of the FBO using `glFramebufferRenderbuffer`.

The shader's output variables are assigned to the attachments of the FBO using `glDrawBuffers`. The second argument to `glDrawBuffers` is an array indicating the FBO buffers to be associated with the output variables. The *i*th element of the array corresponds to the fragment shader output variable at location *i*. In our case, we only have one shader output variable (`FragColor`) at location zero. This statement associates that output variable with `GL_COLOR_ATTACHMENT0`.

The last statement in step 1 unbinds the FBO to revert back to the default framebuffer.

Step 2 creates a 1×1 white texture in texture unit one. We use this texture when rendering the texture so that we don't need to change anything about our shader. As our shader multiplies the texture color by the result of the Phong reflection model, this texture will effectively work as a "non-texture" because multiplying will not change the color. When rendering the texture, we want to use this "non-texture", but when rendering the scene, we'll use the texture attached to the FBO.



This use of a 1×1 texture is certainly not necessary in general. We use it here just so that we can draw to the FBO without a texture being applied to the scene. If you have a texture that should be applied, then that would be more appropriate here.

In step 3 (within the render function), we bind to the FBO, use the "non-texture" in unit one, and render the texture. Note that we need to be careful to set up the viewport (`glViewport`), and the view and projection matrices appropriately for our FBO. As our FBO is 512×512 , we use `glViewport(0, 0, 512, 512)`. Similar changes should be made to the view and projection matrices to match the aspect ratio of the viewport and set up the scene to be rendered to the FBO.

Once we've rendered to the texture, we unbind from the FBO, reset the viewport, and the view and projection matrices, use the FBO's texture (texture unit 0), and draw the scene!

There's more...

As FBOs have multiple color attachment points, we can have several output targets from our fragment shaders. Note that so far, all of our fragment shaders have only had a single output variable assigned to location zero. Hence, we set up our FBO so that its texture corresponds to color attachment zero. In later chapters, we'll look at examples where we use more than one of these attachments for things like deferred shading.

See also

- ▶ The [Applying a 2D texture recipe](#)

Using sampler objects

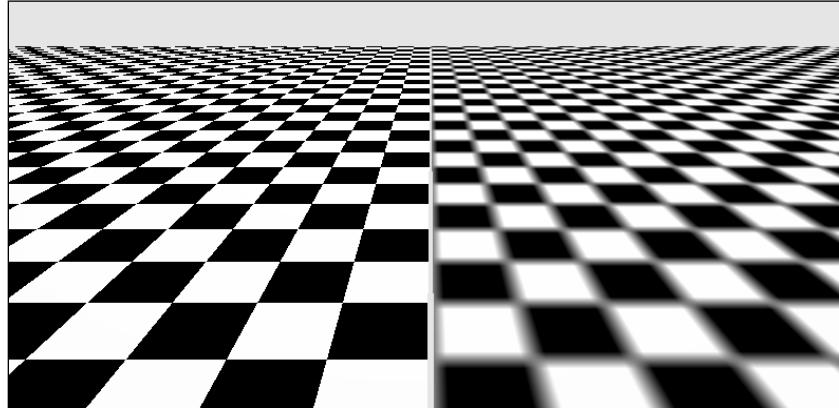
Sampler objects were introduced in OpenGL 3.3, and provide a convenient way to specify the sampling parameters for a GLSL sampler variable. The traditional way to specify the parameters for a texture is to specify them using `glTexParameter`, typically at the time that the texture is defined. The parameters define the sampling state (sampling mode, wrapping and clamping rules, and so on.) for the associated texture. This essentially combines the texture and its sampling state into a single object. If we wanted to sample from a single texture in more than one way (with and without linear filtering for example), we'd have two choices. We would either need to modify the texture's sampling state, or use two copies of the same texture.

In addition, we might want to use the same set of texture sampling parameters for multiple textures. With what we've seen up until now, there's no easy way to do that. With sampler objects we can specify the parameters once, and share them among several texture objects.

Sampler objects separate the sampling state from the texture object. We can create sampler objects that define a particular sampling state and apply that to multiple textures or bind different sampler objects to the same texture. A single sampler object can be bound to multiple textures, which allows us to define a particular sampling state once and share it among several texture objects.

Sampler objects are defined on the OpenGL side (not in GLSL), which makes it effectively transparent to the GLSL.

In this recipe, we'll define two sampler objects and apply them to a single texture. The following image shows the result. The same texture is applied to the two planes. On the left, we use a sampler object set up for nearest-neighbor filtering, and on the right we use the same texture with a sampler object set up for linear filtering.



Getting ready

Will start with the same shaders used in the recipe *Applying a 2D texture*. The shader code will not change at all, but we'll use sampler objects to change the state of the sampler variable `Tex1`.

How to do it...

To set up the texture object and the sampler objects, use the following steps.

1. Create and fill the texture object in the usual way, but this time, we won't set any sampling state using `glTexParameter`.

```
GLuint texID;  
glGenTextures(1, &texID);  
 glBindTexture(GL_TEXTURE_2D, texID);  
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, w, h);  
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGBA,  
 GL_UNSIGNED_BYTE, data);
```

2. Bind the texture to texture unit 0, which is the unit that is used by the shader.

```
glActiveTexture(GL_TEXTURE0);  
 glBindTexture(GL_TEXTURE_2D, texID);
```

3. Next, we create two sampler objects and assign their IDs to separate variables for clarity:

```
GLuint samplers[2];
 glGenSamplers(2, samplers);
 linearSampler = samplers[0];
 nearestSampler = samplers[1];
```

4. Set up `linearSampler` for linear interpolation:

```
glSamplerParameteri(linearSampler, GL_TEXTURE_MAG_FILTER,
                      GL_LINEAR);
glSamplerParameteri(linearSampler, GL_TEXTURE_MIN_FILTER,
                      GL_LINEAR);
```

5. Set up `nearestSampler` for nearest-neighbor sampling:

```
glSamplerParameteri(nearestSampler, GL_TEXTURE_MAG_FILTER,
                      GL_NEAREST);
glSamplerParameteri(nearestSampler, GL_TEXTURE_MIN_FILTER,
                      GL_NEAREST);
```

6. When rendering, we bind to each sampler object when needed:

```
glBindSampler(0, nearestSampler);
// Render objects that use nearest-neighbor sampling
glBindSampler(0, linearSampler);
// Render objects that use linear sampling
```

How it works...

Sampler objects are simple to use, and make it easy to switch between different sampling parameters for the same texture, or use the same sampling parameters for different textures. In steps 1 and 2, we create a texture and bind it to texture unit 0. Normally, we would set the sampling parameters here using `glTexParameter`, but in this case, we'll set them in the sampler objects using `glSamplerParameter`. In step 3, we create the sampler objects and assign their IDs to some variables. In steps 4 and 5, we set up the appropriate sampling parameters using `glSamplerParameter`. This function is almost exactly the same as `glTexParameter` except the first argument is the ID of the sampler object instead of the texture target. This defines the sampling state for each of the two sampler objects (linear for `linearSampler` and nearest for `nearestSampler`).

Finally, we use the sampler objects by binding them to the appropriate texture unit using `glBindSampler` just prior to rendering. In step 6 we bind `nearestSampler` to texture unit 0 first, render some objects, bind `linearSampler` to texture unit 0, and render some more objects. The result here is that the same texture uses different sampling parameters by binding different sampler objects to the texture unit during rendering.

Using Textures _____

See also

- ▶ The *Applying a 2D texture* recipe

5

Image Processing and Screen Space Techniques

In this chapter, we will cover:

- ▶ Applying an edge detection filter
- ▶ Applying a Gaussian blur filter
- ▶ Implementing HDR shading with tone mapping
- ▶ Creating a bloom effect
- ▶ Using gamma correction to improve image quality
- ▶ Using multisample anti-aliasing
- ▶ Using deferred shading
- ▶ Implementing order-independent transparency

Introduction

In this chapter, we will focus on techniques that work directly with the pixels in a framebuffer. These techniques typically involve multiple passes. An initial pass produces the pixel data and subsequent passes apply effects or further process those pixels. To implement this we make use of the ability provided in OpenGL for rendering directly to a texture or set of textures (refer to the *Rendering to a texture* recipe in *Chapter 4, Using Textures*).

The ability to render to a texture, combined with the power of the fragment shader, opens up a huge range of possibilities. We can implement image processing techniques such as brightness, contrast, saturation, and sharpness by applying an additional process in the fragment shader prior to output. We can apply **convolution** filters such as edge detection, smoothing (blur), or sharpening. We'll take a closer look at convolution filters in the recipe on edge detection.

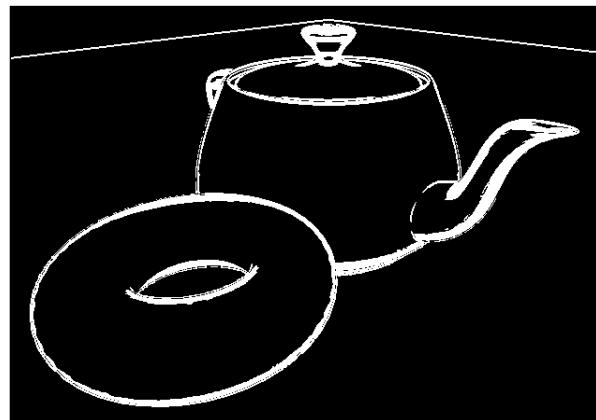
A related set of techniques involves rendering additional information to textures beyond the traditional color information and then, in a subsequent pass, further processing that information to produce the final rendered image. These techniques fall under the general category that is often called **deferred shading**.

In this chapter, we'll look at some examples of each of the preceding techniques. We'll start off with examples of convolution filters for edge detection, blur, and bloom. Then we'll move on to the important topics of gamma correction and multisample anti-aliasing. Finally, we'll finish with a full example of deferred shading.

Most of the recipes in this chapter involve multiple passes. In order to apply a filter that operates on the pixels of the final rendered image, we start by rendering the scene to an intermediate buffer (a texture). Then, in a final pass, we render the texture to the screen by drawing a single full-screen quad, applying the filter in the process. You'll see several variations on this theme in the following recipes.

Applying an edge detection filter

Edge detection is an image processing technique that identifies regions where there is a significant change in the brightness of the image. It provides a way to detect the boundaries of objects and changes in the topology of the surface. It has applications in the field of computer vision, image processing, image analysis, and image pattern recognition. It can also be used to create some visually interesting effects. For example, it can make a 3D scene look similar to a 2D pencil sketch as shown in the following image. To create this image, a teapot, and torus were rendered normally, and then an edge detection filter was applied in a second pass.



The edge detection filter that we'll use here involves the use of a convolution filter, or convolution kernel (also called a filter kernel). A convolution filter is a matrix that defines how to transform a pixel by replacing it with the sum of the products between the values of nearby pixels and a set of pre-determined weights. As a simple example, consider the following convolution filter:

10	11	12	13	14
1 17	0 18	1 19	20	21
0 24	2 25	0 26	27	28
1 31	0 32	1 33	34	35
38	39	40	41	42

The 3×3 filter is shaded in gray superimposed over a hypothetical grid of pixels. The bold faced numbers represent the values of the filter kernel (weights), and the non-bold faced values are the pixel values. The values of the pixels could represent gray-scale intensity or the value of one of the RGB components. Applying the filter to the center pixel in the gray area involves multiplying the corresponding cells together and summing the results. The result would be the new value for the center pixel (25). In this case, the value would be $(17 + 19 + 2 * 25 + 31 + 33)$ or 150.

Of course, in order to apply a convolution filter, we need access to the pixels of the original image and a separate buffer to store the results of the filter. We'll achieve this here by using a two-pass algorithm. In the first pass, we'll render the image to a texture; and then in the second pass, we'll apply the filter by reading from the texture and send the filtered results to the screen.

One of the simplest, convolution-based techniques for edge detection is the so-called **Sobel operator**. The Sobel operator is designed to approximate the gradient of the image intensity at each pixel. It does so by applying two 3×3 filters. The results of the two are the vertical and horizontal components of the gradient. We can then use the magnitude of the gradient as our edge trigger. When the magnitude of the gradient is above a certain threshold, then we assume that the pixel is on an edge.

The 3 x 3 filter kernels used by the Sobel operator are shown in the following equation:

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \mathbf{S}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

If the result of applying Sx is sx and the result of applying Sy is sy, then an approximation of the magnitude of the gradient is given by the following equation:

$$g = \sqrt{s_x^2 + s_y^2}$$

If the value of g is above a certain threshold, we consider the pixel to be an edge pixel, and we highlight it in the resulting image.

In this example, we'll implement this filter as the second pass of a two-pass algorithm. In the first pass, we'll render the scene using an appropriate lighting model, but we'll send the result to a texture. In the second pass, we'll render the entire texture as a screen-filling quad, and apply the filter to the texture.

Getting ready

Set up a framebuffer object (refer to the *Rendering to a texture* recipe in *Chapter 4, Using Textures*) that has the same dimensions as the main window. Connect the first color attachment of the FBO to a texture object in texture unit zero. During the first pass, we'll render directly to this texture. Make sure that the mag and min filters for this texture are set to `GL_NEAREST`. We don't want any interpolation for this algorithm.

Provide vertex information in vertex attribute zero, normals in vertex attribute one, and texture coordinates in vertex attribute two.

The following uniform variables need to be set from the OpenGL application:

- ▶ `Width`: This is used to set the width of the screen window in pixels
- ▶ `Height`: This is used to set the height of the screen window in pixels
- ▶ `EdgeThreshold`: This is the minimum value of g squared required to be considered "on an edge"
- ▶ `RenderTex`: This is the texture associated with the FBO

Any other uniforms associated with the shading model should also be set from the OpenGL application.

How to do it...

To create a shader program that applies the Sobel edge detection filter, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix *
                    vec4(VertexPosition,1.0) );

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;

// The texture containing the results of the first pass
layout( binding=0 ) uniform sampler2D RenderTex;

uniform float EdgeThreshold; // The squared threshold

// This subroutine is used for selecting the functionality
// of pass1 and pass2.
subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Other uniform variables for the Phong reflection model
// can be placed here...

layout( location = 0 ) out vec4 FragColor;
const vec3 lum = vec3(0.2126, 0.7152, 0.0722);
```

```
vec3 phongModel( vec3 pos, vec3 norm )
{
    // The code for the basic ADS shading model goes here...
}

// Approximates the brightness of a RGB value.
float luminance( vec3 color ) {
    return dot(lum, color);

}

subroutine (RenderPassType)
vec4 pass1()
{
    return vec4(phongModel( Position, Normal ),1.0);
}
subroutine( RenderPassType )
vec4 pass2()
{
    ivec2 pix = ivec2(gl_FragCoord.xy);
    float s00 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(-1,1)).rgb);
    float s10 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(-1,0)).rgb);
    float s20 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(-1,-1)).rgb);
    float s01 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(0,1)).rgb);
    float s21 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(0,-1)).rgb);
    float s02 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(1,1)).rgb);
    float s12 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(1,0)).rgb);
    float s22 = luminance(
        texelFetchOffset(RenderTex, pix, 0,
                          ivec2(1,-1)).rgb);
```

```

float sx = s00 + 2 * s10 + s20 - (s02 + 2 * s12 + s22);
float sy = s00 + 2 * s01 + s02 - (s20 + 2 * s21 + s22);

float g = sx * sx + sy * sy;

if( g > EdgeThreshold ) return vec4(1.0);
else return vec4(0.0,0.0,0.0,1.0);
}

void main()
{
    // This will call either pass1() or pass2()
    FragColor = RenderPass();
}

```

In the render function of your OpenGL application, follow these steps for pass #1:

1. Select the framebuffer object (FBO), and clear the color/depth buffers.
2. Select the `pass1` subroutine function (refer to the *Using subroutines to select shader functionality* recipe in *Chapter 2, The Basics of GLSL Shaders*).
3. Set up the model, view, and projection matrices, and draw the scene.

For pass #2, carry out the following steps:

1. Deselect the FBO (revert to the default framebuffer), and clear the color/depth buffers.
2. Select the `pass2` subroutine function.
3. Set the model, view, and projection matrices to the identity matrix.
4. Draw a single quad (or two triangles) that fills the screen (-1 to +1 in x and y), with texture coordinates that range from 0 to 1 in each dimension.

How it works...

The first pass renders all of the scene's geometry sending the output to a texture. We select the subroutine function `pass1`, which simply computes and applies the Phong reflection model (refer to the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2, The Basics of GLSL Shaders*).

In the second pass, we select the subroutine function `pass2`, and render only a single quad that covers the entire screen. The purpose of this is to invoke the fragment shader once for every pixel in the image. In the `pass2` function, we retrieve the values of the eight neighboring pixels of the texture containing the results from the first pass, and compute their brightness by calling the `luminance` function. The horizontal and vertical Sobel filters are then applied and the results are stored in `sx` and `sy`.



The luminance function determines the brightness of an RGB value by computing a weighted sum of the intensities. The weights are from the ITU-R Recommendation Rec. 709. For more details on this, see the Wikipedia entry for "luma".

We then compute the squared value of the magnitude of the gradient (in order to avoid the square root) and store the result in `g`. If the value of `g` is greater than `EdgeThreshold`, we consider the pixel to be on an edge, and we output a white pixel. Otherwise, we output a solid black pixel.

There's more...

The Sobel operator is somewhat crude, and tends to be sensitive to high frequency variations in the intensity. A quick look at Wikipedia will guide you to a number of other edge detection techniques that may be more accurate. It is also possible to reduce the amount of high frequency variation by adding a "blur pass" between the render and edge detection passes. The "blur pass" will smooth out the high frequency fluctuations and may improve the results of the edge detection pass.

Optimization techniques

The technique discussed here requires eight texture fetches. Texture accesses can be somewhat slow, and reducing the number of accesses can result in substantial speed improvements. *Chapter 24 of GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, edited by Randima Fernando (Addison-Wesley Professional 2004), has an excellent discussion of ways to reduce the number of texture fetches in a filter operation by making use of so-called "helper" textures.

See also

- ▶ D. Ziou and S. Tabbone (1998), *Edge detection techniques: An overview*, *International Journal of Computer Vision*, Vol 24, Issue 3
- ▶ *Frei-Chen edge detector*: <http://rastergrid.com/blog/2011/01/frei-chen-edge-detector/>
- ▶ The *Using subroutines to select shader functionality* recipe in *Chapter 2, The Basics of GLSL Shaders*
- ▶ The *Rendering to a texture* recipe in *Chapter 4, Using Textures*
- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in *Chapter 2, The Basics of GLSL Shaders*

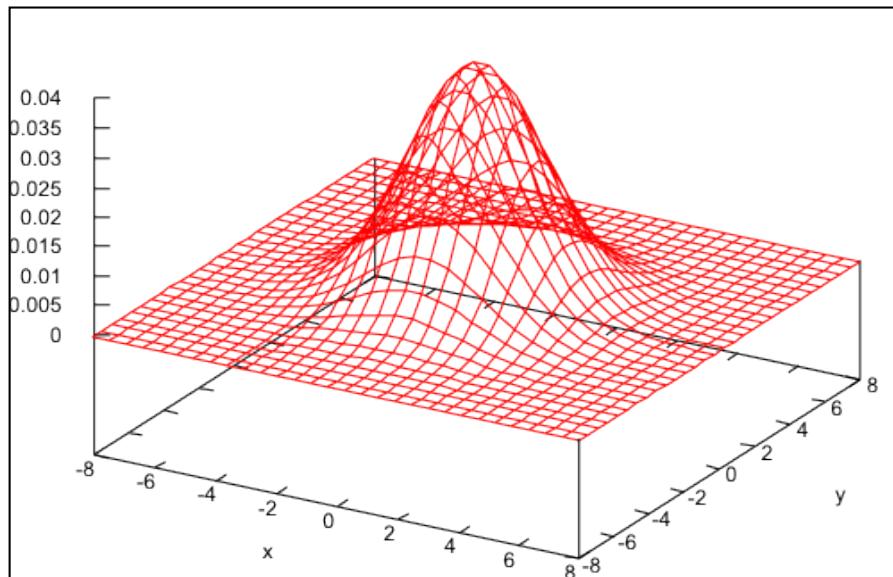
Applying a Gaussian blur filter

A blur filter can be useful in many different situations where the goal is to reduce the amount of noise in the image. As mentioned in the previous recipe, applying a blur filter prior to the edge detection pass may improve the results by reducing the amount of high frequency fluctuation across the image. The basic idea of any blur filter is to mix the color of a pixel with that of nearby pixels using a weighted sum. The weights typically decrease with the distance from the pixel (in 2D screen space) so that pixels that are far away contribute less than those closer to the pixel being blurred.

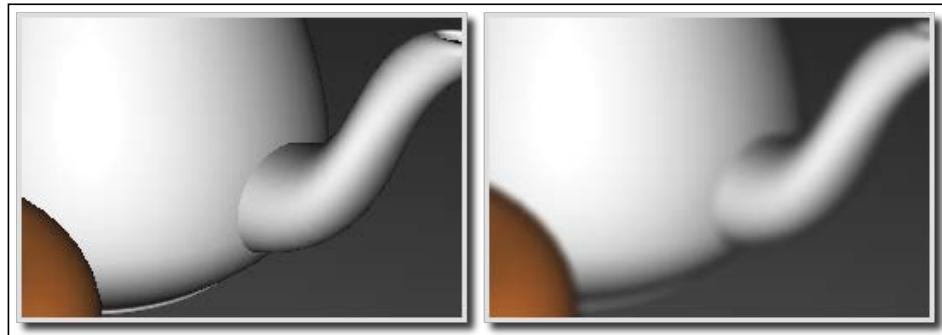
A **Gaussian blur** uses the 2-dimensional Gaussian function to weight the contributions of the nearby pixels.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The sigma squared term is the **variance** of the Gaussian, and determines the width of the Gaussian curve. The Gaussian function is maximum at (0,0), which corresponds to the location of the pixel being blurred and its value decreases as x or y increases. The following graph shows the two-dimensional Gaussian function with a sigma squared value of 4.0:



The following images show a portion of an image before (left) and after (right) the Gaussian blur operation:



To apply a Gaussian blur, for each pixel, we need to compute the weighted sum of all pixels in the image scaled by the value of the Gaussian function at that pixel (where the x and y coordinates of each pixel are based on an origin located at the pixel being blurred). The result of that sum is the new value for the pixel. However, there are two problems with the algorithm so far:

- ▶ As this is a $O(n^2)$ process (where n is the number of pixels in the image), it is likely to be too slow for real-time use
- ▶ The weights must sum to one in order to avoid changing the overall brightness of the image

As we sampled the Gaussian function at discrete locations, and didn't sum over the entire (infinite) bounds of the function, the weights almost certainly do not sum to one.

We can deal with both of the preceding problems by limiting the number of pixels that we blur with a given pixel (instead of the entire image), and by normalizing the values of the Gaussian function. In this example, we'll use a 9×9 Gaussian blur filter. That is, we'll only compute the contributions of the 81 pixels in the neighborhood of the pixel being blurred.

Such a technique would require 81 texture fetches in the fragment shader, which is executed once for each pixel. The total number of texture fetches for an image of size 800×600 would be $800 * 600 * 81 = 38,880,000$. This seems like a lot, doesn't it? The good news is that we can substantially reduce the number of texture fetches by doing the Gaussian blur in two passes.

The two-dimensional Gaussian function is actually just the product of two one-dimensional Gaussians:

$$G(x, y) = G(x)G(y)$$

Where the one-dimensional Gaussian function is given by the following equation:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

So if C_{ij} is the color of the pixel at pixel location (i, j) , the sum that we need to compute is given by the following equation:

$$C_{lm} \leftarrow \sum_{i=-4}^{4} \sum_{j=-4}^{4} G(i, j) C_{l+i \ m+j}$$

This can be re-written using the fact that the two-dimensional Gaussian is a product of two one-dimensional Gaussians:

$$C_{lm} \leftarrow \sum_{i=-4}^{4} G(i) \sum_{j=-4}^{4} G(j) C_{l+i \ m+j}$$

This implies that we can compute the Gaussian blur in two passes. In the first pass, we can compute the sum over j (the vertical sum) in the preceding equation and store the results in a temporary texture. In the second pass, we compute the sum over i (the horizontal sum) using the results from the previous pass.

Now, before we look at the code, there is one important point that has to be addressed. As we mentioned previously, the Gaussian weights must sum to one in order to be a true weighted average. Therefore, we need to normalize our Gaussian weights as in the following equation:

$$C_{lm} \leftarrow \sum_{i=-4}^{4} \frac{G(i)}{k} \sum_{j=-4}^{4} \frac{G(j)}{k} C_{l+i \ m+j}$$

The value of k in the preceding equation is just the sum of the raw Gaussian weights.

$$k = \sum_{i=-4}^{4} G(i)$$

Phew! We've reduced the $O(n_2)$ problem to one that is $O(n)$. OK, with that, let's move on to the code.

We'll implement this technique using three passes and two textures. In the first pass, we'll render the entire scene to a texture. Then, in the second pass, we'll apply the first (vertical) sum to the texture from the first pass and store the results in another texture. Finally, in the third pass, we'll apply the horizontal sum to the texture from the second pass, and send the results to the default framebuffer.

Getting ready

Set up two framebuffer objects (refer to the *Rendering to a texture* recipe in *Chapter 4, Using Textures*), and two corresponding textures. The first FBO should have a depth buffer because it will be used for the first pass. The second FBO need not have a depth buffer because, in the second and third passes, we'll only render a single screen-filling quad in order to execute the fragment shader once for each pixel.

As with the previous recipe, we'll use a subroutine to select the functionality of each pass. The OpenGL program should also set the following uniform variables:

- ▶ **Width**: This is used to set the width of the screen in pixels
- ▶ **Height**: This is used to set the height of the screen in pixels
- ▶ **Weight []**: This is the array of normalized Gaussian weights
- ▶ **Texture0**: This is to set this to texture unit zero
- ▶ **PixOffset []**: This is the array of offsets from the pixel being blurred

How to do it...

To create a shader program that implements Gaussian blur, use the following code:

1. Use the same vertex shader as was used in the previous recipe *Applying an edge detection filter*.
2. Use the following code for the fragment shader:

```
in vec3 Position; // Vertex position
in vec3 Normal; // Vertex normal
layout(binding=0) uniform sampler2D Texture0;

subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Other uniform variables for the Phong reflection model
// can be placed here...
layout( location = 0 ) out vec4 FragColor;

uniform int PixOffset[5] = int[] (0,1,2,3,4);
uniform float Weight[5];
```

```
vec3 phongModel( vec3 pos, vec3 norm )
{
    // The code for the Phong reflection model goes here...
}

subroutine (RenderPassType)
vec4 pass1()
{
    return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2()
{
    ivec2 pix = ivec2(gl_FragCoord.xy);
    vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];
    for( int i = 1; i < 5; i++ )
    {
        sum += texelFetchOffset( Texture0, pix, 0,
                                ivec2(0,PixOffset[i])) * Weight[i];
        sum += texelFetchOffset( Texture0, pix, 0,
                                ivec2(0,-PixOffset[i])) * Weight[i];
    }
    return sum;
}

subroutine( RenderPassType )
vec4 pass3()
{
    ivec2 pix = ivec2(gl_FragCoord.xy);
    vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];
    for( int i = 1; i < 5; i++ )
    {
        sum += texelFetchOffset( Texture0, pix, 0,
                                ivec2(PixOffset[i],0)) * Weight[i];
        sum += texelFetchOffset( Texture0, pix, 0,
                                ivec2(-PixOffset[i],0)) * Weight[i];
    }
    return sum;
}

void main()
{
    // This will call either pass1(), pass2(), or pass3()
    FragColor = RenderPass();
}
```

3. In the OpenGL application, compute the Gaussian weights for the offsets found in the uniform variable `PixOffset`, and store the results in the array `Weight`. You could use the following code to do so:

```
char uniName[20];
float weights[5], sum, sigma2 = 4.0f;

// Compute and sum the weights
weights[0] = gauss(0,sigma2); // The 1-D Gaussian function
sum = weights[0];
for( int i = 1; i < 5; i++ ) {
    weights[i] = gauss(i, sigma2);
    sum += 2 * weights[i];
}

// Normalize the weights and set the uniform
for( int i = 0; i < 5; i++ ) {
    snprintf(uniName, 20, "Weight[%d]", i);
    prog.setUniform(uniName, weights[i] / sum);
}
```

In the main render function, implement the following steps for pass #1:

1. Select the render framebuffer, enable the depth test, and clear the color/depth buffers.
2. Select the `pass1` subroutine function.
3. Draw the scene.

Use the following steps for pass #2:

1. Select the intermediate framebuffer, disable the depth test, and clear the color buffer.
2. Select the `pass2` subroutine function.
3. Set the view, projection, and model matrices to the identity matrix.
4. Bind the texture from pass #1 to texture unit zero.
5. Draw a full-screen quad.

Use the following steps for pass #3:

1. Deselect the framebuffer (revert to the default), and clear the color buffer.
2. Select the `pass3` subroutine function.
3. Bind the texture from pass #2 to texture unit zero.
4. Draw a full-screen quad.

How it works...

In the preceding code for computing the Gaussian weights (code segment 3), the function named `gauss` computes the one-dimensional Gaussian function where the first argument is the value for `x` and the second argument is `sigma squared`. Note that we only need to compute the positive offsets because the Gaussian is symmetric about zero. As we are only computing the positive offsets, we need to carefully compute the sum of the weights. We double all of the non-zero values because they will be used twice (for the positive and negative offset).

The first pass (subroutine function `pass1`) renders the scene to a texture using the Phong reflection model (refer to the *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, *The Basics of GLSL Shaders*).

The second pass (subroutine function `pass2`) applies the weighted vertical sum of the Gaussian blur operation, and stores the results in yet another texture. We read pixels from the texture created in the first pass, offset in the vertical direction by the amounts in the `PixOffset` array. We sum using weights from the `Weight` array. (The `dy` term is the height of a texel in texture coordinates.) We sum in both directions at the same time, a distance of four pixels in each vertical direction.

The third pass (subroutine `pass3`) is very similar to the second pass. We accumulate the weighted, horizontal sum using the texture from the second pass. By doing so, we are incorporating the sums produced in the second pass into our overall weighted sum as described earlier. Thereby, we are creating a sum over a 9×9 pixel area around the destination pixel. For this pass, the output color goes to the default framebuffer to make up the final result.

There's more...

We can further optimize the preceding technique to reduce the number of texture accesses by half. If we make clever use of the automatic linear interpolation that takes place when accessing a texture (when `GL_LINEAR` is the mag/min mode), we can actually get information about two texels with one texture access! A great blog post by Daniel Rákos describes the technique in detail (visit <http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>).

Of course, we can also adapt the preceding technique to blur a larger range of texels by increasing the size of the arrays `Weight` and `PixOffset` and re-computing the weights, and/or we could use different values of `sigma2` to vary the shape of the Gaussian.

See also

- ▶ Bilateral filtering: http://people.csail.mit.edu/sparis/bf_course/
- ▶ The *Rendering to a texture* recipe in *Chapter 4, Using Textures*
- ▶ The *Applying an edge detection filter* recipe
- ▶ The *Using subroutines to select shader functionality* recipe in *Chapter 2, The Basics of GLSL Shaders*

Implementing HDR lighting with tone mapping

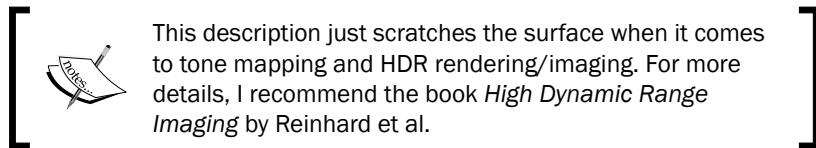
When rendering for most output devices (monitors or televisions), the device only supports a typical color precision of 8 bits per color component, or 24 bits per pixel. Therefore, for a given color component, we're limited to a range of intensities between 0 and 255. Internally, OpenGL uses floating-point values for color intensities, providing a wide range of both values and precision. These are eventually converted to 8 bit values by mapping the floating-point range [0.0, 1.0] to the range of an unsigned byte [0, 255] before rendering.

Real scenes, however, have a much wider range of luminance. For example, light sources that are visible in a scene, or direct reflections of them, can be hundreds to thousands of times brighter than the objects that are illuminated by the source. When we're working with 8 bits per channel, or the floating-point range [0.0, -1.0], we can't represent this range of intensities. If we decide to use a larger range of floating point values, we can do a better job of internally representing these intensities, but in the end, we still need to compress down to the 8-bit range.

The process of computing the lighting/shading using a larger dynamic range is often referred to as **High Dynamic Range rendering (HDR rendering)**. Photographers are very familiar with this concept. When a photographer wants to capture a larger range of intensities than would normally be possible in a single exposure, he/she might take several images with different exposures to capture a wider range of values. This concept, called **High Dynamic Range imaging (HDR imaging)**, is very similar in nature to the concept of HDR rendering. A post-processing pipeline that includes HDR is now considered a fundamentally essential part of any game engine.

Tone mapping is the process of taking a wide dynamic range of values and compressing them into a smaller range that is appropriate for the output device. In computer graphics, generally, tone mapping is about mapping to the 8-bit range from some arbitrary range of values. The goal is to maintain the dark and light parts of the image so that both are visible, and neither is completely "washed out".

For example, a scene that includes a bright light source might cause our shading model to produce intensities that are greater than 1.0. If we were to simply send that to the output device, anything greater than 1.0 would be clamped to 255, and would appear white. The result might be an image that is mostly white, similar to a photograph that is over exposed. Or, if we were to linearly compress the intensities to the [0, 255] range, the darker parts might be too dark, or completely invisible. With tone mapping, we want to maintain the brightness of the light source, and also maintain detail in the darker areas.



The mathematical function used to map from one dynamic range to a smaller range is called the **Tone Mapping Operator (TMO)**. These generally come in two "flavors", local operators and global operators. A local operator determines the new value for a given pixel by using its current value and perhaps the value of some nearby pixels. A global operator needs some information about the entire image, in order to do its work. For example, it might need to have the overall average luminance of all pixels in the image. Other global operators use a histogram of luminance values over the entire image to help fine-tune the mapping.

In this recipe, we'll use a simple global operator that is described in the book *Real Time Rendering*. This operator uses the log-average luminance of all pixels in the image. The log-average is determined by taking the logarithm of the luminance and averaging those values, then converting back, as shown in the following equation:

$$\bar{L}_w = \exp \left(\frac{1}{N} \sum_{x,y} \ln(0.0001 + L_w(x,y)) \right)$$

$L_w(x, y)$ is the luminance of the pixel at (x, y) . The 0.0001 term is included in order to avoid taking the logarithm of zero for black pixels. This log-average is then used as part of the tone mapping operator shown as follows::

$$L(x, y) = \frac{a}{\bar{L}_w} L_w(x, y)$$

The a term in this equation is the key. It acts in a similar way to the exposure level in a camera. The typical values for a range from 0.18 to 0.72. Since this tone mapping operator compresses the dark and light values a bit too much, we'll use a modification of the previous equation that doesn't compress the dark values as much, and includes a maximum luminance (L_{white}), a configurable value that helps to reduce some of the extremely bright pixels.

$$L_d(x, y) = \frac{L(x, y) \left(1 + \frac{L(x, y)}{L_{\text{white}}^2}\right)}{1 + L(x, y)}$$

This is the tone mapping operator that we'll use in this example. We'll render the scene to a high-resolution buffer, compute the log-average luminance, and then apply the previous tone-mapping operator in a second pass.

However, there's one more detail that we need to deal with before we can start implementing. The previous equations all deal with luminance. Starting with an RGB value, we can compute its luminance, but once we modify the luminance, how do we modify the RGB components to reflect the new luminance, but without changing the hue (or **chromaticity**)?

 The chromaticity is the perceived color, independent of the brightness of that color. For example, grey and white are two brightness levels for the same color.

The solution involves switching color spaces. If we convert the scene to a color space that separates out the luminance from the chromaticity, then we can change the luminance value independently. The **CIE XYZ** color space has just what we need. The CIE XYZ color space was designed so that the Y component describes the luminance of the color, and the chromaticity can be determined by two derived parameters (x and y). The derived color space is called the **CIE xyY** space, and is exactly what we're looking for. The Y component contains the luminance and the x and y components contain the chromaticity. By converting to the CIE xyY space, we've factored out the luminance from the chromaticity allowing us to change the luminance without affecting the perceived color.

So the process involves converting from RGB to CIE XYZ, then converting to CIE xyY, modifying the luminance and reversing the process to get back to RGB. To convert from RGB to CIE XYZ (and vice-versa) can be described as a transformation matrix (refer to the code or the See also section for the matrix).

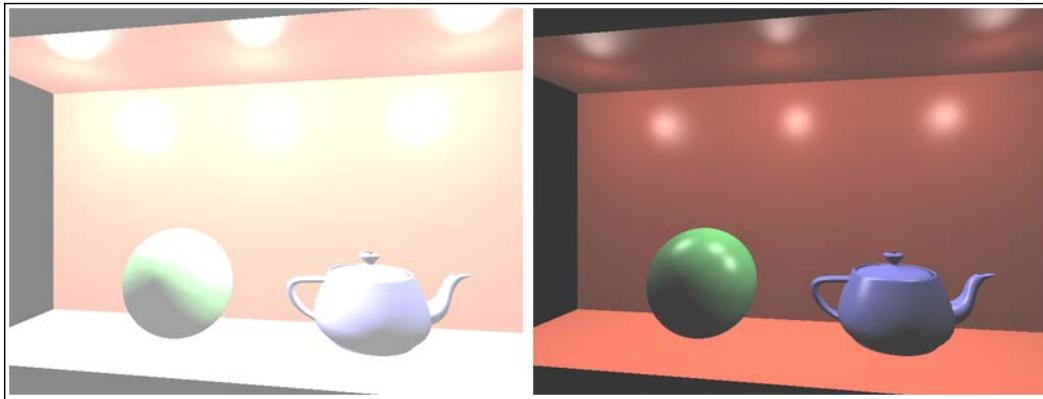
The conversion from XYZ to xyY involves the following:

$$x = \frac{X}{X + Y + Z} \quad y = \frac{Y}{X + Y + Z}$$

Finally, converting from xyY back to XYZ is done using the following equations:

$$X = \frac{Y}{y}x \quad Z = \frac{Y}{y}(1 - x - y)$$

The following images show an example of the results of this tone mapping operator. The left image shows the scene rendered without any tone mapping. The shading was deliberately calculated with a wide dynamic range using three strong light sources. The scene appears "blown out" because any values that are greater than 1.0 simply get clamped to the maximum intensity. The image on the right uses the same scene and the same shading, but with the previous tone mapping operator applied. Note the recovery of the specular highlights from the "blown-out" areas on the sphere and teapot.



Getting ready

The steps involved are the following:

1. Render the scene to a high-resolution texture.
2. Compute the log-average luminance (on the CPU).
3. Render a screen-filling quad to execute the fragment shader for each screen pixel. In the fragment shader, read from the texture created in step 1, apply the tone mapping operator, and send the results to the screen.

To get set up, create a high-res texture (using `GL_RGB32F` or similar format) attached to a framebuffer with a depth attachment. Set up your fragment shader with a subroutine for each pass. The vertex shader can simply pass through the position and normal in eye coordinates.

How to do it...

To implement HDR tone mapping, we'll use the following steps:

1. In the first pass we want to just render the scene to the high-resolution texture. Bind to the framebuffer that has the texture attached and render the scene normally. Apply whatever shading equation strikes your fancy.
2. Compute the log average luminance of the pixels in the texture. To do so, we'll pull the data from the texture and loop through the pixels on the CPU side. We do this on the CPU for simplicity, a GPU implementation, perhaps with a compute shader, would be faster.

```
GLfloat *texData = new GLfloat [width*height*3];
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, hdrTex);
 glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_FLOAT, texData);
 float sum = 0.0f;
 int size = width*height;
 for( int i = 0; i < size; i++ ) {
    float lum = computeLuminance(
        texData[i*3+0], texData[i*3+1], texData[i*3+2]));
    sum += logf( lum + 0.00001f );
 }
 delete [] texData;
 float logAve = expf( sum / size );
```

3. Set the AveLum uniform variable using logAve. Switch back to the default framebuffer, and draw a screen-filling quad. In the fragment shader, apply the tone mapping operator to the values from the texture produced in step 1.

```
// Retrieve high-res color from texture
vec4 color = texture( HdrTex, TexCoord );

// Convert to XYZ
vec3 xyzCol = rgb2xyz * vec3(color);

// Convert to xyY
float xyzSum = xyzCol.x + xyzCol.y + xyzCol.z;
vec3 xyYCol = vec3(0.0);
if( xyzSum > 0.0 ) // Avoid divide by zero
    xyYCol = vec3( xyzCol.x / xyzSum,
                    xyzCol.y / xyzSum, xyzCol.z);

// Apply the tone mapping operation to the luminance
// (xyYCol.z or xyzCol.y)
float L = (Exposure * xyYCol.z) / AveLum;
```

```
L = (L * (1 + L / (White * White))) / (1 + L);  
  
// Using the new luminance, convert back to XYZ  
if( xyYCol.y > 0.0 ) {  
    xyzCol.x = (L * xyYCol.x) / (xyYCol.y);  
    xyzCol.y = L;  
    xyzCol.z = (L * (1 - xyYCol.x - xyYCol.y)) / xyYCol.y;  
}  
  
// Convert back to RGB and send to output buffer  
FragColor = vec4( xyz2rgb * xyzCol, 1.0);
```

How it works...

In the first step, we render the scene to an HDR texture. In step 2, we compute the log-average luminance by retrieving the pixels from the texture and doing the computation on the CPU (OpenGL side).

In step 3, we render a single screen-filling quad to execute the fragment shader for each screen pixel. In the fragment shader, we retrieve the HDR value from the texture and apply the tone-mapping operator. There are two "tunable" variables in this calculation. The variable `Exposure` corresponds to the a term in the tone mapping operator, and the variable `White` corresponds to L_{white} . For the previous image, we used values of 0.35 and 0.928 respectively.

There's more...

Tone mapping is not an exact science. Often, it is a process of experimenting with the parameters until you find something that works well and looks good.

We could improve the efficiency of the previous technique by implementing step 2 on the GPU using compute shaders (refer to *Chapter 10, Using Compute Shaders*) or some other clever technique. For example, we could write the logarithms to a texture, then iteratively downsample the full frame to a 1×1 texture. The final result would be available in that single pixel. However, with the flexibility of the compute shader, we could optimize this process even more.

See also

- ▶ Bruce Justin Lindbloom has provided a useful web resource for conversion between color spaces. It includes among other things the transformation matrices needed to convert from RGB to XYZ. Visit: http://www.brucelindbloom.com/index.html?Eqn_XYZ_to_RGB.html.
- ▶ The *Rendering to a texture* recipe in *Chapter 4, Using Textures*.

Creating a bloom effect

A **bloom** is a visual effect where the bright parts of an image seem to have fringes that extend beyond the boundaries into the darker parts of the image. This effect has its basis in the way that cameras and the human visual system perceive areas of high contrast. Sources of bright light "bleed" into other areas of the image due to the so-called **Airy disc**, which is a diffraction pattern produced by light that passes through an aperture.

The following image shows a bloom effect in the animated film Elephant's Dream (© 2006, Blender Foundation / Netherlands Media Art Institute / www.elephantsdream.org). The bright white color from the light behind the door "bleeds" into the darker parts of the image.



Producing such an effect within an artificial CG rendering requires determining which parts of the image are bright enough, extracting those parts, blurring, and re-combining with the original image. Typically, the bloom effect is associated with HDR (High Dynamic Range) rendering. With HDR rendering, we can represent a larger range of intensities for each pixel (without quantizing artifacts). The bloom effect is more accurate when used in conjunction with HDR rendering due to the fact that a wider range of brightness values can be represented.

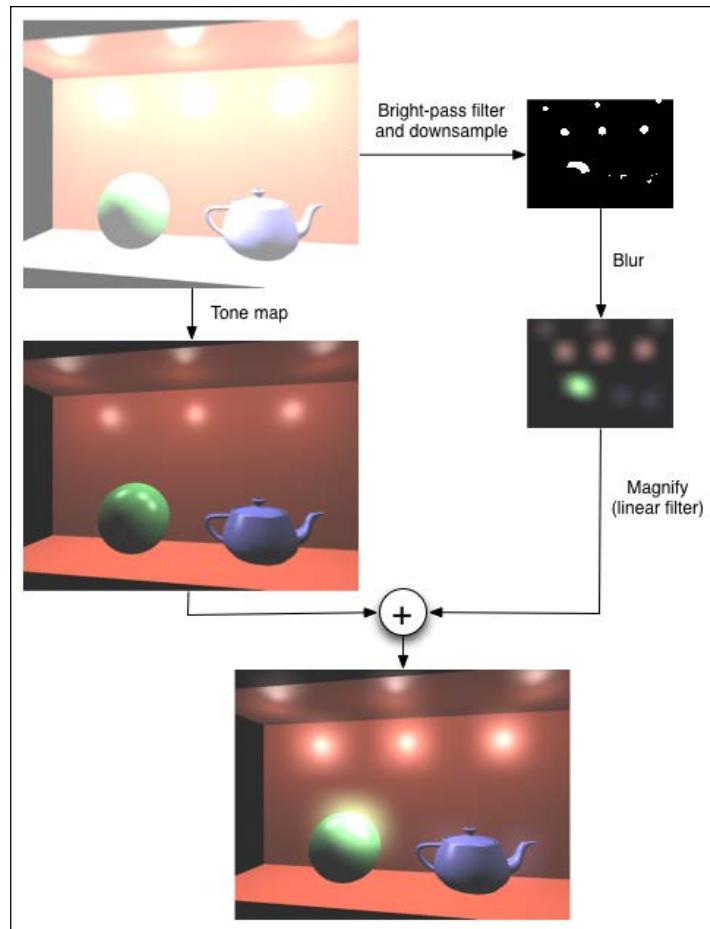
Despite the fact that HDR produces higher quality results, it is still possible to produce a bloom effect when using standard (non-HDR) color values. The result may not be as effective, but the principles involved are similar for either situation.

In the following example, we'll implement a bloom effect using five passes, consisting of four major steps:

1. In the first pass, we will render the scene to an HDR texture.
2. The second pass will extract the parts of the image that are brighter than a certain threshold value. We'll refer to this as the **bright-pass filter**. We'll also downsample to a lower resolution buffer when applying this filter. We do so because we will gain additional blurring of the image when we read back from this buffer using a linear sampler.

3. The third and fourth passes will apply the Gaussian blur to the bright parts (refer to the *Applying a Gaussian blur filter* recipe in this chapter).
4. In the fifth pass, we'll apply tone mapping and add the tone-mapped result to the blurred bright-pass filter results.

The following diagram summarizes the process. The upper-left shows the scene rendered to an HDR buffer, with some of the colors out of gamut, causing much of the image to be "blown-out". The bright-pass filter produces a smaller (about a quarter or an eighth of the original size) image with only pixels that correspond to a luminance that is above a threshold. The pixels are shown as white because they have values that are greater than one in this example. A two-pass Gaussian blur is applied to the downsampled image, and tone mapping is applied to the original image. The final image is produced by combining the tone-mapped image with the blurred bright-pass filter image. When sampling the latter, we use a linear filter to get additional blurring. The final result is shown at the bottom. Note the bloom on the bright highlights on the sphere and the back wall.



Getting ready

For this recipe, we'll need two framebuffer objects, each associated with a texture. The first will be used for the original HDR render, the second will be used for the two passes of the Gaussian blur operation. In the fragment shader, we'll access the original render via the variable `HdrTex`, and the two stages of the Gaussian blur will be accessed via `BlurTex`.

The uniform variable `LumThresh` is the minimum luminance value used in the second pass. Any pixels greater than that value will be extracted and blurred in the following passes.

Use a vertex shader that passes through the position and normal in eye coordinates.

How to do it...

To generate a bloom effect, use the following steps:

1. In the first pass, render the scene to the framebuffer with a high-res backing texture.
2. In the second pass, switch to a framebuffer containing a high-res texture that is smaller than the size of the full render. In the example code, we use a texture that is one-eighth the size. Draw a full screen quad to initiate the fragment shader for each pixel, and in the fragment shader sample from the high-res texture, and write only those values that are larger than `LumThresh`. Otherwise, color the pixel black.

```
vec4 val = texture(HdrTex, TexCoord);
if( luminance(val.rgb) > LumThresh )
    FragColor = val;
else
    FragColor = vec4(0.0);
```

3. In the third and fourth passes, apply the Gaussian blur to the results of the second pass. This can be done with a single framebuffer and two textures. "Ping-pong" between them, reading from one and writing to the other. For details, refer to the *Applying a Gaussian blur filter* recipe in this chapter.
4. In the fifth and final pass, switch to linear filtering from the texture that was produced in the fourth pass. Switch to the default frame buffer (the screen). Apply the tone-mapping operator from the *Implementing HDR lighting with tone mapping* recipe to the original image texture (`HdrTex`), and combine the results with the blurred texture from step 3. The linear filtering and magnification should provide an additional blur.

```
// Retrieve high-res color from texture
vec4 color = texture( HdrTex, TexCoord );

// Apply tone mapping to color, result is toneMapColor
...
```

```
////////// Combine with blurred texture //////////
vec4 blurTex = texture(BlurTex1, TexCoord);

FragColor = toneMapColor + blurTex;
```

How it works...

Due to space constraints, I haven't show the entire fragment shader code here. The code is available from the GitHub repository. The fragment shader is implemented with five subroutine methods, one for each pass. The first pass renders the scene normally to the HDR texture. During this pass, the active framebuffer object (FBO) is the one associated with the texture corresponding to `HdrTex`, so output is sent directly to that texture.

The second pass reads from `HdrTex`, and writes out only pixels that have a luminance above the threshold value `LumThresh`. The value is `(0,0,0,0)` for pixels that have a brightness (luma) value below `LumThresh`. The output goes to the second framebuffer, which contains a much smaller texture (one-eighth the size of the original).

The third and fourth passes apply the basic Gaussian blur operation (refer to the *Applying a Gaussian blur filter* recipe in this chapter). In these passes, we "ping-pong" between `BlurTex1` and `BlurTex2`, so we must be careful to swap the appropriate texture into the framebuffer.

In the fifth pass, we switch back to the default framebuffer, and read from `HdrTex` and `BlurTex1`. `BlurTex1` contains the final blurred result from step four, and `HdrTex` contains the original render. We apply tone mapping to the results of `HdrTex` and add to `BlurTex1`. When pulling from `BlurTex1`, we are applying a linear filter, gaining additional blurring.

There's more...

Note that we applied the tone-mapping operator to the original rendered image, but not to the blurred bright-pass filter image. One could choose apply the TMO to the blurred image as well, but in practice, it is often not necessary.

We should keep in mind that the bloom effect can also be visually distracting if it is overused. A little goes a long way.

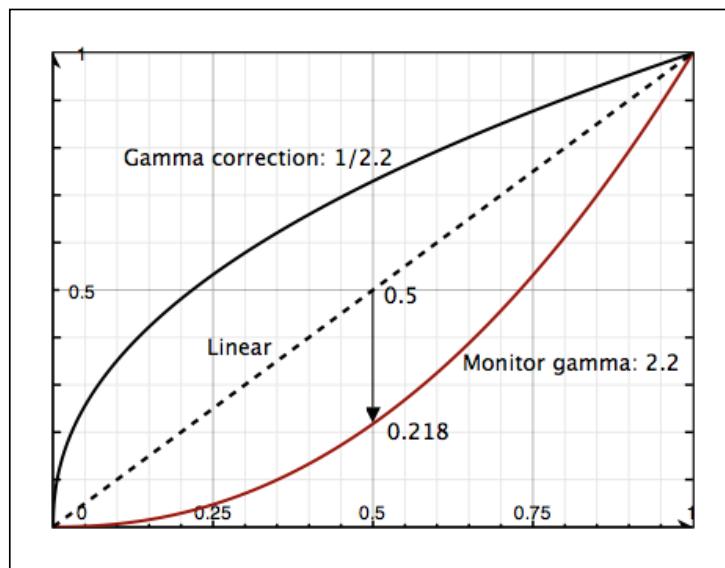
See also

- ▶ *HDR meets Black&White 2* by Francesco Caruzzi in *Shader X6*
- ▶ The *Rendering to a texture* in recipe in *Chapter 4, Using Textures*
- ▶ The *Applying an edge detection filter* recipe
- ▶ The *Using subroutines to select shader functionality* recipe in *Chapter 2, The Basics of GLSL Shaders*

Using gamma correction to improve image quality

It is common for many books about OpenGL and 3D graphics to somewhat neglect the subject of gamma correction. Lighting and shading calculations are performed, and the results are sent directly to the output buffer without modification. However, when we do this, we may produce results that don't quite end up looking the way we might expect they should. This may be due to the fact that computer monitors (both the old CRT and the newer LCD) have a non-linear response to pixel intensity. For example, without gamma correction, a grayscale value of 0.5 will not appear half as bright as a value of 1.0. Instead, it will appear to be darker than it should.

The lower curve in the following graph shows the response curves of a typical monitor (gamma of 2.2). The x axis is the intensity, and the y axis is the perceived intensity. The dashed line represents a linear set of intensities. The upper curve represents gamma correction applied to linear values. The lower curve represents the response of a typical monitor. A grayscale value of 0.5 would appear to have a value of 0.218 on a screen that had a similar response curve.



The non-linear response of a typical monitor can usually be modeled using a simple power function. The perceived intensity (P) is proportional to the pixel intensity (I) raised to a power that is usually called "gamma".

$$P = I^\gamma$$

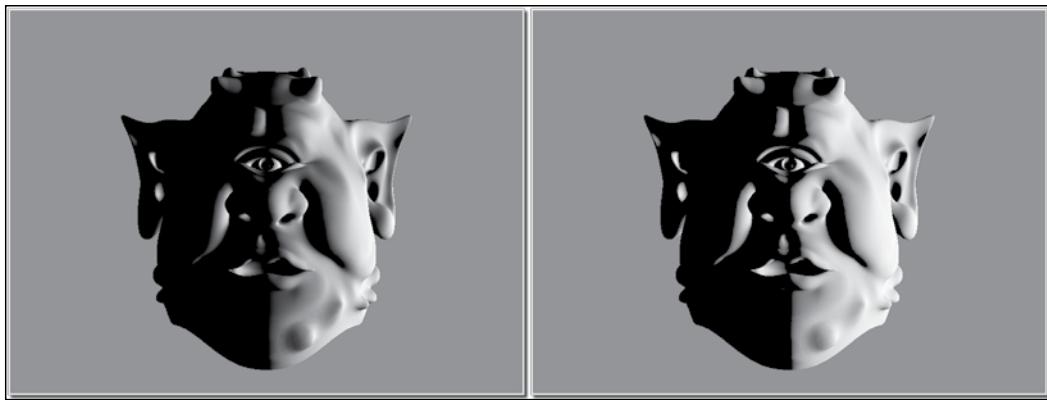
Depending on the display device, the value of gamma is usually somewhere between 2.0 and 2.4. Some kind of monitor calibration is often needed to determine a precise value.

In order to compensate for this non-linear response, we can apply **gamma correction** before sending our results to the output framebuffer. Gamma correction involves raising the pixel intensities to a power that will compensate for the monitor's non-linear response to achieve a perceived result that appears linear. Raising the linear-space values to the power of $1/\text{gamma}$ will do the trick.

$$I = \left(I^{\frac{1}{\gamma}} \right)^{\gamma}$$

When rendering, we can do all of our lighting and shading computations ignoring the fact that the monitor's response curve is non-linear. This is sometimes referred to as "working in linear space". When the final result is to be written to the output framebuffer, we can apply the gamma correction by raising the pixel to the power of $1/\text{gamma}$ just before writing. This is an important step that will help to improve the look of the rendered result.

As an example, consider the following images. The image on the left is the mesh rendered without any consideration of gamma at all. The reflection model is computed and the results are directly sent to the framebuffer. On the right is the same mesh with gamma correction applied to the color just prior to output.



The obvious difference is that the left image appears much darker than the image on the right. However, the more important distinction is the variations from light to dark across the face. While the transition at the shadow terminator seems stronger than before, the variations within the lighted areas are less extreme.

Applying gamma correction is an important technique, and can be effective in improving the results of a lighting model.

How to do it...

Adding gamma correction to an OpenGL program can be as simple as carrying out the following steps:

1. Set up a uniform variable named `Gamma` and set it to an appropriate value for your system.
2. Use the following code or something similar in a fragment shader:

```
vec3 color = lightingModel( ... );
FragColor = vec4( pow( color, vec3(1.0/Gamma) ), 1.0 );
```

If your shader involves texture data, care must be taken to make sure that the texture data is not already gamma-corrected so that you don't apply gamma correction twice (refer to the *There's more...* section of this recipe).

How it works...

The color determined by the lighting/shading model is computed and stored in the variable `color`. We think of this as computing the color in "linear space". There is no consideration of the monitor's response during the calculation of the shading model (assuming that we don't access any texture data that might already be gamma-corrected).

To apply the correction, in the fragment shader, we raise the color of the pixel to the power of $1.0 / \text{Gamma}$, and apply the result to the output variable `FragColor`. Of course, the inverse of `Gamma` could be computed outside the fragment shader to avoid the division operation.

We do not apply the gamma correction to the alpha component because it is typically not desired.

There's more...

The application of gamma correction is a good idea in general; however, some care must be taken to make sure that computations are done within the correct "space". For example, textures could be photographs or images produced by other imaging applications that apply gamma correction before storing the data within the image file. Therefore, if we use a texture in our application as a part of the lighting model and then apply gamma correction, we will be effectively applying gamma correction twice to the data from the texture. Instead, we need to be careful to "decode" the texture data, by raising to the power of gamma prior to using the texture data in our lighting model.

There is a very detailed discussion about these and other issues surrounding gamma correction in *Chapter 24, The Importance of Being Linear* in the book *GPU Gems 3*, edited by Hubert Nguyen (Addison-Wesley Professional 2007), and this is highly recommended supplemental reading.

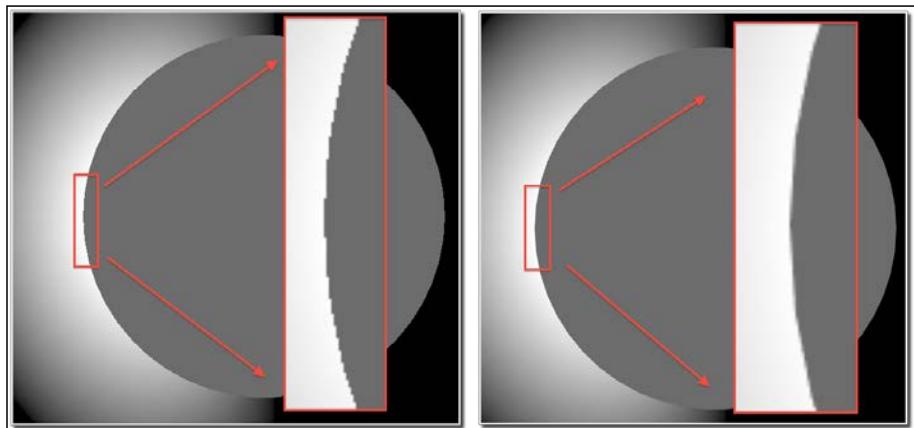
Using multisample anti-aliasing

Anti-aliasing is the technique of removing or reducing the visual impact of **aliasing artifacts** that are present whenever high-resolution or continuous information, is presented at a lower resolution. In real-time graphics, aliasing often reveals itself in the jagged appearance of polygon edges, or the visual distortion of textures that have a high degree of variation.

The following images show an example of aliasing artifacts at the edge of an object. On the left, we see that the edge appears jagged. This occurs because each pixel is determined to lie either completely inside the polygon, or completely outside it. If the pixel is determined to be inside, it is shaded, otherwise it is not. Of course, this is not entirely accurate. Some pixels lie directly on the edge of the polygon. Some of the screen area that the pixel encompasses actually lies within the polygon and some lies outside. Better results could be achieved if we were to modify the shading of a pixel based upon the amount of the pixel's area that lies within the polygon. The result could be a mixture of the shaded surface's color with the color outside the polygon, where the area that is covered by the pixel determines the proportions. You might be thinking that this sounds like it would be prohibitively expensive to do. That may be true; however, we can approximate the results by using multiple **samples** per pixel.

Multisample anti-aliasing involves evaluating multiple samples per pixel and combining the results of those samples to determine the final value for the pixel. The samples are located at various points within the pixel's extent. Most of these samples will fall inside the polygon; but for pixels near a polygon's edge, some will fall outside. The fragment shader will typically execute only once for each pixel as usual. For example, with 4x multisample anti-aliasing (MSAA), rasterization happens at four times the frequency. For each pixel, the fragment shader is executed once and the result is scaled based on how many of the four samples fall within the polygon.

The following image on the right shows the results when multisample anti-aliasing is used. The inset image is a zoomed portion of the inside edge of a torus. On the left, the torus is rendered without MSAA. The right-hand image shows the results with MSAA enabled:



OpenGL has supported multisampling for some time now, and it is nearly transparent to use. It is simply a matter of turning it on or off. It works by using additional buffers to store the subpixel samples as they are processed. Then the samples are combined together to produce a final color for the fragment. Nearly all of this is automatic, and there is little that a programmer can do to fine-tune the results. However, at the end of this recipe, we'll discuss the interpolation qualifiers that can affect the results.

In this recipe, we'll see the code needed to enable multisample anti-aliasing in an OpenGL application.

Getting ready

The technique for enabling multisampling is unfortunately dependent on the window system API. In this example, we'll demonstrate how it is done using GLFW. The steps will be similar in GLUT or other APIs that support OpenGL.

How to do it...

To make sure that the multisample buffers are created and available, use the following steps:

1. When creating your OpenGL window, you need to select an OpenGL context that supports MSAA. The following is how one would do so in GLFW:

```
glfwWindowHint(GLFW_SAMPLES, 8);
... // Other settings
window = glfwCreateWindow( WIN_WIDTH, WIN_HEIGHT,
                           "Window title", NULL, NULL );
```

2. To determine whether multisample buffers are available and how many samples per-pixel are actually being used, you can use the following code (or something similar):

```
GLint bufs, samples;
glGetIntegerv(GL_SAMPLE_BUFFERS, &bufs);
glGetIntegerv(GL_SAMPLES, &samples);
printf("MSAA: buffers = %d samples = %d\n", bufs, samples);
```

3. To enable multisampling, use the following:

```
 glEnable(GL_MULTISAMPLE);
```

4. To disable multisampling, use the following:

```
 glDisable(GL_MULTISAMPLE);
```

How it works...

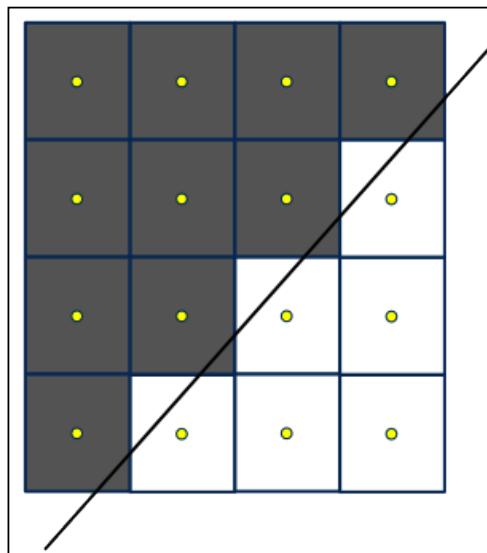
As just mentioned, the technique for creating an OpenGL context with multisample buffers is dependent on the API used for interacting with the window system. The preceding example demonstrates how it might be done using GLFW. Once the OpenGL context is created, it is easy to enable multisampling by simply using the `glEnable` call shown in the preceding example.

Stay tuned, because in the next section, I'll discuss a subtle issue surrounding interpolation of shader variables when multisample anti-aliasing is enabled.

There's more...

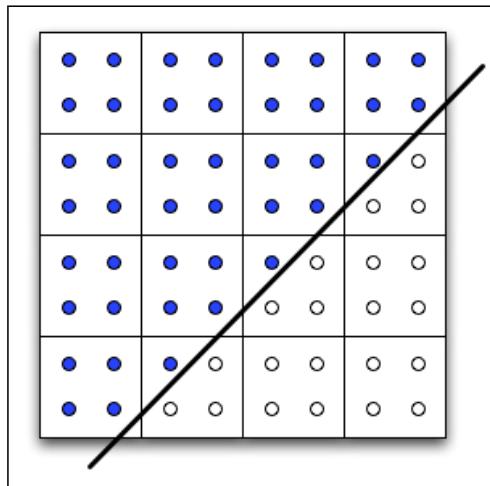
There are two interpolation qualifiers within the GLSL that allow the programmer to fine-tune some aspects of multisampling. They are: `sample` and `centroid`.

Before we can get into how `sample` and `centroid` work, we need a bit of background. Let's consider the way that polygon edges are handled without multisampling. A fragment is determined to be inside or outside of a polygon by determining where the center of that pixel lies. If the center is within the polygon, the pixel is shaded, otherwise it is not. The following image represents this behavior. It shows pixels near a polygon edge without MSAA. The line represents the edge of the polygon. Gray pixels are considered to be inside the polygon. White pixels are outside and are not shaded. The dots represent the pixel centers.



The values for the interpolated variables (the fragment shader's input variables) are interpolated with respect to the center of each fragment, which will always be inside the polygon.

When multisample anti-aliasing is enabled, multiple samples are computed per fragment at various locations within the fragment's extent. If any of those samples lie within the polygon, then the shader is executed at least once for that pixel (but not necessarily for each sample). As a visual example, the following image represents pixels near a polygon's edge. The dots represent the samples. The dark samples lie within the polygon and the white samples lie outside the polygon. If any sample lies within the polygon, the fragment shader is executed (usually only once) for that pixel. Note that for some pixels, the pixel centers lie outside the polygon. So with MSAA, the fragment shader may execute slightly more often near the edges of polygons.



Now, here's the important point. The values of the fragment shader's input variables are normally interpolated to the center of the pixel rather than to the location of any particular sample. In other words, the value that is used by the fragment shader is determined by interpolating to the location of the fragment's center, which may lie outside the polygon! If we are relying on the fact that the fragment shader's input variables are interpolated strictly between their values at the vertices (and not outside that range) then this might lead to unexpected results.

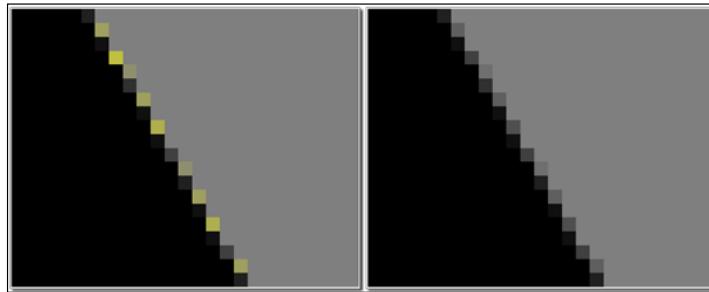
As an example, consider the following portion of a fragment shader:

```
in vec2 TexCoord;

layout( location = 0 ) out vec4 FragColor;

void main()
{
    vec3 yellow = vec3(1.0,1.0,0.0);
    vec3 color = vec3(0.0); // black
    if( TexCoord.s > 1.0 )
        color = yellow;
    FragColor = vec4( color , 1.0 );
}
```

This shader is designed to color the polygon black unless the s component of the texture coordinate is greater than one. In that case, the fragment gets a yellow color. If we render a square with texture coordinates that range from zero to one in each direction, we may get the results shown in the following image on the left. The images show the enlarged edge of a polygon where the s texture coordinate is about 1.0. Both images were rendered using the preceding shader. The right-hand image was created using the `centroid` qualifier (more on this later in this chapter).



The left image shows that some pixels along the edge have a lighter color (yellow if the image is in full color). This is due to the fact that the texture coordinate is interpolated to the pixel's center, rather than to any particular sample's location. Some of the fragments along the edge have a center that lies outside of the polygon, and therefore end up with a texture coordinate that is greater than one!

We can ask OpenGL to instead compute the value for the input variable by interpolating to some location that is not only within the pixel, but also within the polygon. We can do so by using the `centroid` qualifier as shown in the following code:

```
centroid in vec2 TexCoord;
```

(The qualifier needs to also be included with the corresponding output variable in the vertex shader.) When `centroid` is used with the preceding shader, we get the preceding image shown on the right.



In general, we should use `centroid` or `sample` when we know that the interpolation of the input variables should not extend beyond the values of those variables at the vertices.



The `sample` qualifier forces OpenGL to interpolate the shader's input variables to the actual location of the sample itself.

```
sample in vec2 TexCoord;
```

This, of course, requires that the fragment shader be executed once for each sample. This will produce the most accurate results, but the performance hit may not be worthwhile, especially if the visual results produced by `centroid` (or without the default) are good enough.

Using deferred shading

Deferred shading is a technique that involves postponing (or "deferring") the lighting/shading step to a second pass. We do this (among other reasons) in order to avoid shading a pixel more than once. The basic idea is as follows:

1. In the first pass, we render the scene, but instead of evaluating the reflection model to determine a fragment color, we simply store all of the geometry information (position, normal, texture coordinate, reflectivity, and so on) in an intermediate set of buffers, collectively called the **g-buffer** (g for geometry).
2. In the second pass, we simply read from the g-buffer, evaluate the reflection model, and produce a final color for each pixel.

When deferred shading is used, we avoid evaluating the reflection model for a fragment that will not end up being visible. For example, consider a pixel located in an area where two polygons overlap. The fragment shader may be executed once for each polygon that covers that pixel; however, the resulting color of only one of the two executions will end up being the final color for that pixel (assuming that blending is not enabled). The cycles spent in evaluating the reflection model for one of the two fragments are effectively wasted. With deferred shading, the evaluation of the reflection model is postponed until all the geometry has been processed, and the visible geometry is known at each pixel location. Hence, the reflection model is evaluated only once for each pixel on the screen. This allows us to do lighting in a more efficient fashion. For example, we could use even hundreds of light sources because we are only evaluating the lighting once per screen pixel.

Deferred shading is fairly simple to understand and work with. It can therefore help with the implementation of complex lighting/reflection models.

In this recipe, we'll go through a simple example of deferred shading. We'll store the following information in our g-buffer: the position, normal, and diffuse color (the diffuse reflectivity). In the second pass, we'll simply evaluate the diffuse lighting model using the data stored in the g-buffer.



This recipe is meant to be a starting point for deferred shading. If we were to use deferred shading in a more substantial (real-world) application, we'd probably need more components in our g-buffer. It should be straightforward to extend this example to use more complex lighting/shading models.

Getting ready

The g-buffer will contain three textures for storing the position, normal, and diffuse color. There are three uniform variables that correspond to these three textures: `PositionTex`, `NormalTex`, and `ColorTex`; these textures should be assigned to texture units 0, 1, and 2, respectively. Likewise, the vertex shader assumes that position information is provided in vertex attribute 0, the normal is provided in attribute 1, and the texture coordinate in attribute 2.

The fragment shader has several uniform variables related to light and material properties that must be set from the OpenGL program. Specifically, the structures `Light` and `Material` apply to the shading model used here.

You'll need a variable named `deferredFBO` (type `GLuint`) to store the handle to the FBO.

How to do it...

To create a shader program that implements deferred shading (with diffuse shading only), use the following code:

1. To create the framebuffer object that contains our g-buffer use the following code (or something similar):

```
void createGBufTex(GLenum texUnit, GLenum format,
                   GLuint &texid) {
    glBindTexture(GL_TEXTURE_2D, texid);
    glTexStorage2D(GL_TEXTURE_2D, 1, format, width, height);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
}

...
GLuint depthBuf, posTex, normTex, colorTex;

// Create and bind the FBO
 glGenFramebuffers(1, &deferredFBO);
 glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

// The depth buffer
 glGenRenderbuffers(1, &depthBuf);
 glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
 glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      width, height);
```

```
// The position, normal and color buffers
createGBufTex(GL_TEXTURE0, GL_RGB32F, posTex); // Position
createGBufTex(GL_TEXTURE1, GL_RGB32F, normTex); // Normal
createGBufTex(GL_TEXTURE2, GL_RGB8, colorTex); // Color

// Attach the images to the framebuffer
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
    GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuf);
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, posTex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, normTex, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
    GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, colorTex, 0);

GLenum drawBuffers [] = {GL_NONE, GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2};
glDrawBuffers(4, drawBuffers);
```

2. Use the following code for the vertex shader:

```
layout( location = 0 ) in vec3 VertexPosition;
layout( location = 1 ) in vec3 VertexNormal;
layout( location = 2 ) in vec2 VertexTexCoord;

out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
    Normal = normalize( NormalMatrix * VertexNormal );
    Position = vec3( ModelViewMatrix *
                    vec4(VertexPosition,1.0) );
    TexCoord = VertexTexCoord;
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Use the following code for the fragment shader:

```
struct LightInfo {
    vec4 Position; // Light position in eye coords.
    vec3 Intensity; // Diffuse intensity
};
```

```
uniform LightInfo Light;
struct MaterialInfo {
    vec3 Kd;           // Diffuse reflectivity
};
uniform MaterialInfo Material;

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

// The g-buffer textures
layout(binding = 0) uniform sampler2D PositionTex;
layout(binding = 1) uniform sampler2D NormalTex;
layout(binding = 2) uniform sampler2D ColorTex;
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec3 PositionData;
layout (location = 2) out vec3 NormalData;
layout (location = 3) out vec3 ColorData;

vec3 diffuseModel( vec3 pos, vec3 norm, vec3 diff )
{
    vec3 s = normalize(vec3(Light.Position) - pos);
    float sDotN = max( dot(s,norm), 0.0 );
    vec3 diffuse = Light.Intensity * diff * sDotN;

    return diffuse;
}

subroutine (RenderPassType)
void pass1()
{
    // Store position, norm, and diffuse color in g-buffer
    PositionData = Position;
    NormalData = Normal;
    ColorData = Material.Kd;
}

subroutine(RenderPassType)
void pass2()
{
    // Retrieve position, normal and color information from
    // the g-buffer textures
    vec3 pos = vec3( texture( PositionTex, TexCoord ) );
```

```
    vec3 norm = vec3( texture( NormalTex, TexCoord ) );
    vec3 diffColor = vec3( texture(ColorTex, TexCoord) );

    FragColor=vec4(diffuseModel(pos,norm,diffColor), 1.0);
}

void main() {
    // This will call either pass1 or pass2
    RenderPass();
}
```

In the render function of the OpenGL application, use the following steps for pass #1:

1. Bind to the framebuffer object `deferredFBO`.
2. Clear the color/depth buffers, select the `pass1` subroutine function, and enable the depth test (if necessary).
3. Render the scene normally.

Use the following steps for pass #2:

1. Revert to the default FBO (bind to framebuffer 0).
2. Clear the color buffer, select the `pass2` subroutine function, and disable the depth test (if desired).
3. Render a screen-filling quad (or two triangles) with texture coordinates that range from zero to one in each direction.

How it works...

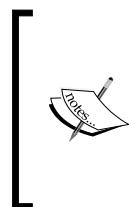
When setting up the framebuffer object (FBO) for the g-buffer, we use textures with internal format `GL_RGB32F` for the position and normal components. As we are storing geometry information, rather than simply color information, there is a need to use a higher resolution (that is more bits per pixel). The buffer for the diffuse reflectivity just uses `GL_RGB8` since we don't need the extra resolution for these values.

The three textures are then attached to the framebuffer at color attachments 0, 1, and 2 using `glFramebufferTexture2D`. They are then connected to the fragment shader's output variables with the call to `glDrawBuffers`.

```
glDrawBuffers(4, drawBuffers);
```

The array `drawBuffers` indicates the relationship between the framebuffer's components and the fragment shader's output variable locations. The *i*th item in the array corresponds to the *i*th output variable location. This call sets color attachments 0, 1, and 2 to output variable locations 1, 2, and 3, respectively. (Note that the fragment shader's corresponding variables are `PositionData`, `NormalData`, and `ColorData`.)

The vertex shader is a basic "pass-through" shader. It just converts the position and normal to eye (camera) coordinates and passes them along to the fragment shader. The texture coordinate is passed through unchanged.



During pass 2, it is not strictly necessary to convert and pass through the normal and position, as they will not be used in the fragment shader at all. However, to keep things simple, I did not include this optimization. It would be a simple matter to add a subroutine to the vertex shader in order to "switch off" the conversion during pass 2. (Of course, we need to set `gl_Position` regardless.)

In the fragment shader, the functionality depends on the value of the subroutine variable `RenderPass`. It will either call `pass1` or `pass2`, depending on its value. In the `pass1` function, we store the values of `Position`, `Normal`, and `Material.Kd` in the appropriate output variables, effectively storing them in the textures that we just talked about.

In the `pass2` function, the values of the position, normal, and color are retrieved from the textures, and used to evaluate the diffuse lighting model. The result is then stored in the output variable `FragColor`. In this pass, `FragColor` should be bound to the default framebuffer, so the results of this pass will appear on the screen.

There's more...

In the graphics community, the relative advantages and disadvantages of deferred shading are a source of much debate. Deferred shading is not ideal for all situations. It depends greatly on the specific requirements of your application, and one needs to carefully evaluate the benefits and drawbacks before deciding whether or not to use deferred shading.

Multi-sample anti-aliasing with deferred shading is possible in recent versions of OpenGL by making use of `GL_TEXTURE_2D_MULTISAMPLE`.

Another consideration is that deferred shading can't do blending/transparency very well. In fact, blending is impossible with the basic implementation we saw some time ago. Additional buffers with depth-peeling can help by storing additional layered geometry information in the g-buffer.

One notable advantage of deferred shading is that one can retain the depth information from the first pass and access it as a texture during the shading pass. Having access to the entire depth buffer as a texture can enable algorithms such as depth of field (depth blur), screen space ambient occlusion, volumetric particles, and other similar techniques.

For much more information about deferred shading, refer to *Chapter 9* in *GPU Gems 2* edited by Matt Pharr and Randima Fernando (Addison-Wesley Professional 2005) and *Chapter 19* of *GPU Gems 3* edited by Hubert Nguyen (Addison-Wesley Professional 2007). Both combined, provide an excellent discussion of the benefits and drawbacks of deferred shading, and how to make the decision of whether or not to use it in your application.

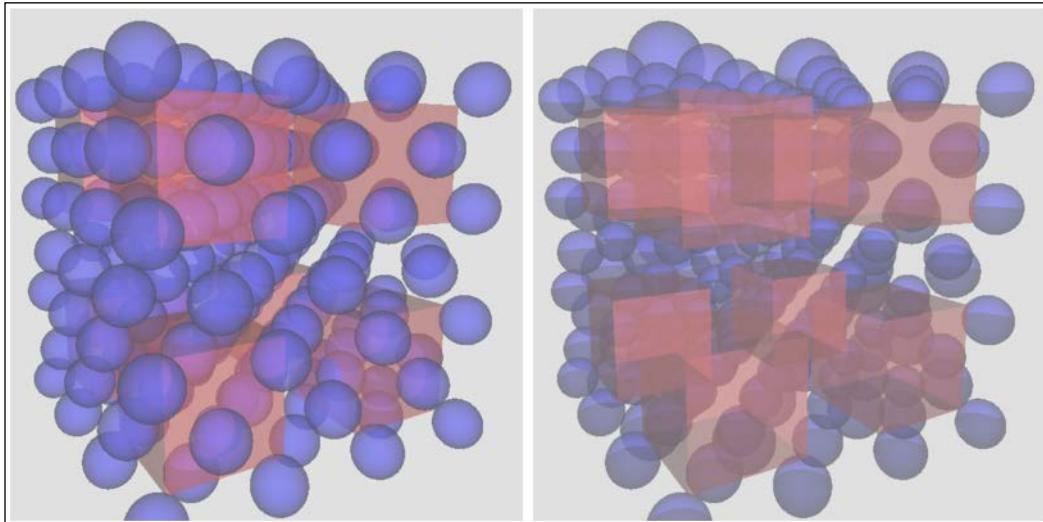
See also

- ▶ The *Rendering to a texture* recipe in Chapter 4, *Using Textures*

Implementing order-independent transparency

Transparency can be a difficult effect to do accurately in pipeline architectures like OpenGL. The general technique is to draw opaque objects first, with the depth buffer enabled, then to make the depth buffer read-only (using `glDepthMask`), disable the depth test, and draw the transparent geometry. However, care must be taken to ensure that the transparent geometry is drawn from "back to front". That is, objects farther from the viewer should be drawn before the objects that are closer. This requires some sort of depth-sorting to take place prior to rendering.

The following images show an example of a block of small, semi-transparent spheres with some semi-transparent cubes placed evenly within them. On the right-hand side, the objects are rendered in an arbitrary order, using standard OpenGL blending. The result looks incorrect because objects are blended in an improper order. The cubes, which were drawn last, appear to be on top of the spheres, and the spheres look "jumbled", especially in the middle of the block. On the left, the scene is drawn using proper ordering, so objects appear to be oriented correctly with respect to depth, and the overall look is more realistic looking.



Order Independent Transparency (OIT) means that we can draw objects in any order, and still get accurate results. Depth sorting is done at some other level, perhaps within the fragment shader, so that the programmer need not sort objects before rendering. There are a variety of techniques for doing this; one of the most common technique is to keep a list of colors for each pixel, sort them by depth, and then blend them together in the fragment shader. In this recipe we'll use this technique to implement OIT making use of some of the newest features in OpenGL 4.3.

Shader storage buffer objects (SSBO) and **image load/store** are some of the newest features in OpenGL, introduced in 4.3 and 4.2, respectively. They allow arbitrary read/write access to data from within a shader. Prior to this, shaders were very limited in terms of what data they could access. They could read from a variety of locations (textures, uniforms, and so on), but writing was very limited. Shaders could only write to controlled, isolated locations such as fragment shader outputs and transform feedback buffers. This was for very good reason. Since shaders can execute in parallel and in a seemingly arbitrary order, it is very difficult to ensure that data is consistent between instantiations of a shader. Data written by one shader instance might not be visible to another shader instance whether or not that instance is executed after the other. Despite this, there are good reasons for wanting to read and write to shared locations. With the advent of SSBOs and image load/store, that capability is now available to us. We can create buffers and textures (called images) with read/write access to any shader instance. This is especially important for compute shaders, the subject of *Chapter 10, Using Compute Shaders*. However, this power comes at a price. The programmer must now be very careful to avoid the types of memory consistency errors that come along with writing to memory that is shared among parallel threads. Additionally, the programmer must be aware of the performance issues that come with synchronization between shader invocations.



For a more thorough discussion of the issues involved with memory consistency and shaders, refer to *Chapter 11*, of The OpenGL Programming Guide, 8th Edition. That chapter also includes another similar implementation of OIT.

In this recipe, we'll use SSBOs and image load/store to implement order-independent transparency. We'll use two passes. In the first pass, we'll render the scene geometry and store a linked list of fragments for each pixel. After the first pass, each pixel will have a corresponding linked list containing all fragments that were written to that pixel including their depth and color. In the second pass, we'll draw a full-screen quad to invoke the fragment shader for each pixel. In the fragment shader, we'll extract the linked list for the pixel, sort the fragments by depth (largest to smallest), and blend the colors in that order. The final color will then be sent to the output device.

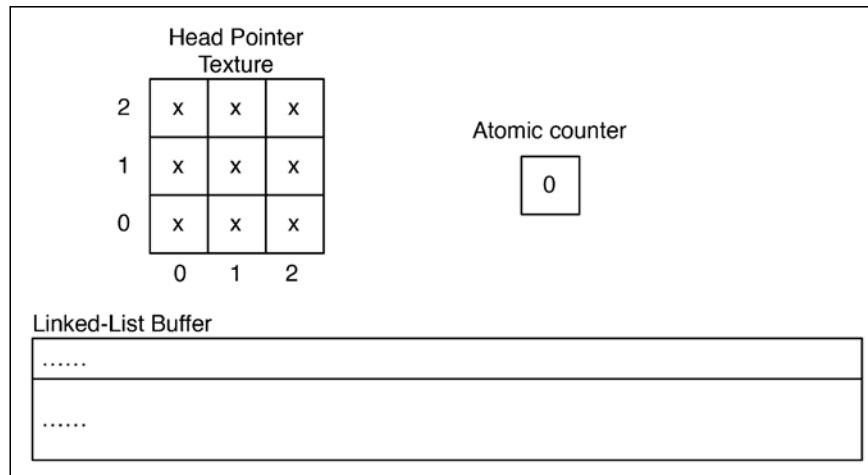
That's the basic idea, so let's dig into the details. We'll need three memory objects that are shared among the fragment shader instances.

1. **An atomic counter:** This is just an unsigned integer that we'll use to keep track of the size of our linked list buffer. Think of this as the index of the first unused slot in the buffer.

2. **A head-pointer texture that corresponds to the size of the screen:** The texture will store a single unsigned integer in each texel. The value is the index of the head of the linked list for the corresponding pixel.
3. **A buffer containing all of our linked lists:** Each item in the buffer will correspond to a fragment, and contains a struct with the color and depth of the fragment as well as an integer, which is the index of the next fragment in the linked list.

In order to understand how all of this works together, let's consider a simple example.

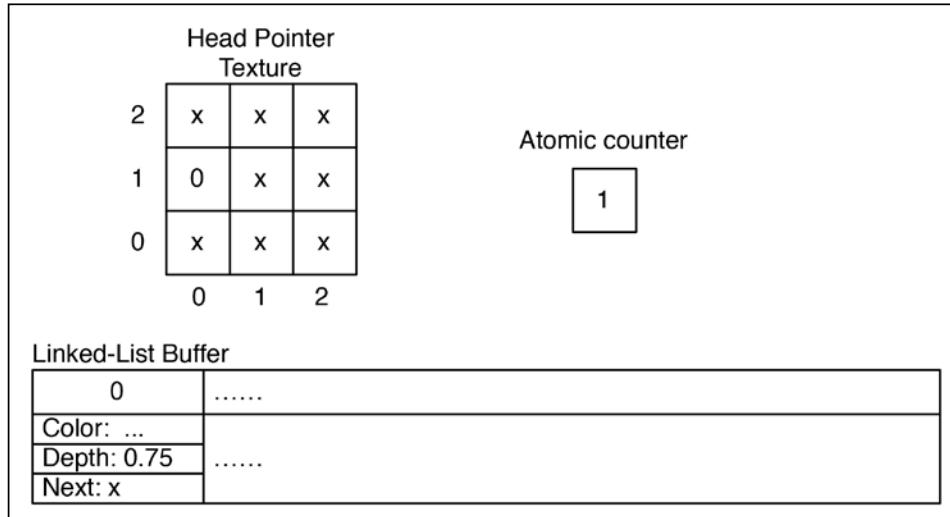
Suppose that our screen is 3 pixels wide and 3 pixels high. We'll have a head pointer texture that is the same dimensions, and we'll initialize all of the texels to a special value that indicates the end of the linked list (an empty list). In the following diagram, that value is shown as an 'x', but in practice, we'll use `0xffffffff`. The initial value of the counter is zero, and the linked list buffer is allocated to a certain size, but treated as empty initially. The initial state of our memory looks like the following diagram:



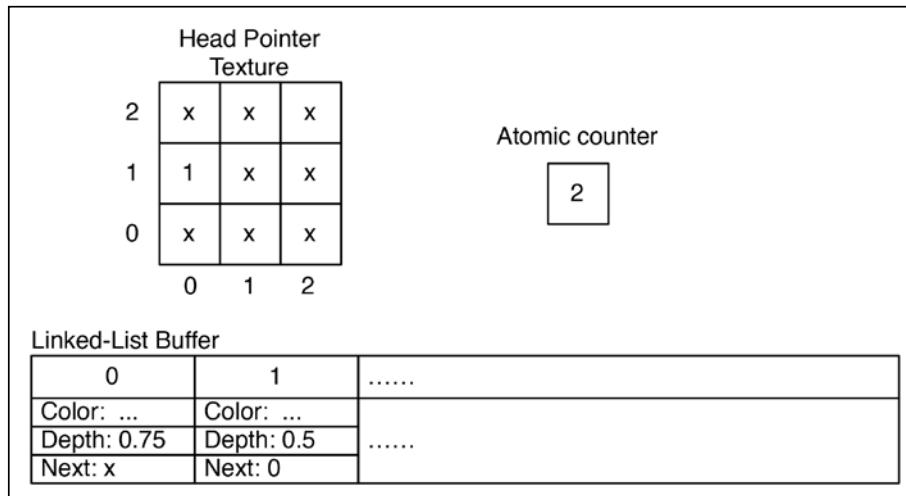
Now suppose that a fragment is rendered at the position (0,1) with a depth of 0.75. The fragment shader will take the following steps:

1. Increment the atomic counter. The new value will be 1, but we'll use the previous value (0) as the index for our new node in the linked list.
2. Update the head pointer texture at (0,1) with the previous value of the counter (0). This is the index of the new head of the linked list at that pixel. Hold on to the previous value that was stored there (x), we'll need that in the next step.
3. Add a new value into the linked list buffer at the location corresponding to the previous value of the counter (0). Store here the color of the fragment and its depth. Store in the "next" component the previous value of the head pointer texture at (0,1) that we held on to in step 2. In this case, it is the special value indicating the end of the list.

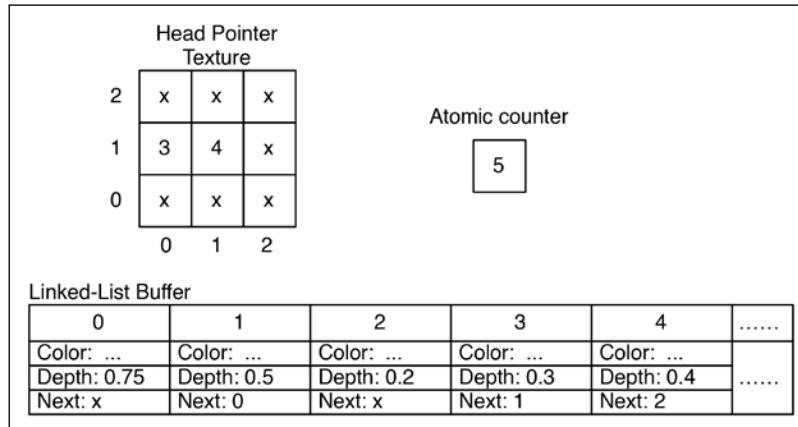
After processing this fragment, the memory layout looks like the following:



Now, suppose another fragment is rendered at (0,1), with a depth of 0.5. The fragment shader will execute the same steps as the previous ones, resulting in the following memory layout:



We now have a 2-element linked list starting at index 1 and ending at index 0. Suppose, now that we have three more fragments in the following order: a fragment at (1,1) with a depth of 0.2, a fragment at (0,1) with a depth of 0.3, and a fragment at (1,1) with a depth of 0.4. Following the same steps for each fragment, we get the following result:



The linked list at (0,1) consists of fragments {3, 1, 0} and the linked list at (1,1) contains fragments {4, 2}.

Now, we must keep in mind that due to the highly parallel nature of GPUs, fragments can be rendered in virtually any order. For example, fragments from two different polygons might proceed through the pipeline in the opposite order as to when the draw instructions for polygons were issued. As a programmer, we must not expect any specific ordering of fragments. Indeed, instructions from separate instances of the fragment shader may interleave in arbitrary ways. The only thing that we can be sure of is that the statements within a particular instance of the shader will execute in order. Therefore, we need to convince ourselves that any interleaving of the previous three steps will still result in a consistent state. For example, suppose instance one executes steps 1 and 2, then another instance (another fragment, perhaps at the same fragment coordinates) executes steps 1, 2, and 3, before the first instance executes step 3. Will the result still be consistent? I think you can convince yourself that it will be, even though the linked list will be broken for a short time during the process. Try working through other interleavings and convince yourself that we're OK.

Not only can statements within separate instances of a shader interleave with each other, but the subinstructions that make up the statements can interleave. (For example, the subinstructions for an increment operation consist of a load, increment, and a store.) What's more, they could actually execute at exactly the same time. Consequently, if we aren't careful, nasty memory consistency issues can crop up. To help avoid this, we need to make careful use of the GLSL support for atomic operations.

Recent versions of OpenGL (4.2 and 4.3) have introduced the tools that we need to make this algorithm possible. OpenGL 4.2 introduced atomic counters and the ability to read and write to arbitrary locations within a texture (called image load/store). OpenGL 4.3 introduced shader storage buffer objects. We'll make use of all three of these features in this example, as well as the various atomic operations and memory barriers that go along with them.

Getting ready

There's a bunch of setup needed here, so I'll go into a bit of detail with some code segments. First, we'll set up a buffer for our atomic counter:

```
GLuint counterBuffer;
 glGenBuffers(1, &counterBuffer);
 glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, counterBuffer);
 glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), NULL,
              GL_DYNAMIC_DRAW);
```

Next, we create a buffer for our linked list storage:

```
GLuint llBuf;
 glGenBuffers(1, &llBuf);
 glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, llBuf);
 glBufferData(GL_SHADER_STORAGE_BUFFER, maxNodes * nodeSize, NULL,
              GL_DYNAMIC_DRAW);
```

[nodeSize in the previous code is the size of a struct `NodeType` used in the fragment shader (in the later part of code). This is computed based on the `std430` layout. For details on the `std430` layout, see the OpenGL specification document. For this example, nodeSize is `5 * sizeof(GLfloat) + sizeof(GLuint)`.]

We also need to create a texture to hold the list head pointers. We'll use 32-bit unsigned integers, and bind it to image unit 0:

```
glGenTextures(1, &headPtrTex);
 glBindTexture(GL_TEXTURE_2D, headPtrTex);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_R32UI, width, height);
 glBindImageTexture(0, headPtrTex, 0, GL_FALSE, 0, GL_READ_WRITE,
                    GL_R32UI);
```

After we render each frame, we need to clear the texture by setting all texels to a value of `0xffffffff`. To help with that, we'll create a buffer of the same size as the texture, with each value set to our "clear value":

```
vector<GLuint> headPtrClear(width * height, 0xffffffff);
 GLuint clearBuf;
```

```
glGenBuffers(1, &clearBuf);
 glBindBuffer(GL_PIXEL_UNPACK_BUFFER, clearBuf);
 glBufferData(GL_PIXEL_UNPACK_BUFFER,
              headPtrClear.size()*sizeof(GLuint),
              &headPtrClear[0], GL_STATIC_COPY);
```

That's all the buffers we'll need. Note the fact that we've bound the head pointer texture to image unit 0, the atomic counter buffer to index 0 of the `GL_ATOMIC_COUNTER_BUFFER` binding point (`glBindBufferBase`), and the linked list storage buffer to index 0 of the `GL_SHADER_STORAGE_BUFFER` binding point. We'll refer back to that later.

Use a pass-through vertex shader that sends the position and normal along in eye coordinates.

How to do it...

With all of the buffers set up, we need two render passes. Before the first pass, we want to clear our buffers to default values (that is, empty lists), and to reset our atomic counter buffer to zero.

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, clearBuf);
 glBindTexture(GL_TEXTURE_2D, headPtrTex);
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,
                 GL_RED_INTEGER, GL_UNSIGNED_INT, NULL);
 GLuint zero = 0;
 glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, counterBuffer);
 glBufferSubData(GL_ATOMIC_COUNTER_BUFFER, sizeof(GLuint), &zero);
```

In the first pass, we'll render the full scene geometry. Generally, we should render all the opaque geometry first, and store the results in a texture. However, we'll skip that step for this example to keep things simple and focused. Instead, we'll render only transparent geometry. When rendering the transparent geometry, we need to make sure to put the depth buffer in read-only mode (use `glDepthMask`). In the fragment shader, we add each fragment to the appropriate linked list.

```
layout (early_fragment_tests) in;

#define MAX_FRAGMENTS 75

in vec3 Position;
in vec3 Normal;

struct NodeType {
    vec4 color;
    float depth;
    uint next;
};


```

```
layout(binding=0, r32ui) uniform uimage2D headPointers;
layout(binding=0, offset=0) uniform atomic_uint
                           nextNodeCounter;
layout(binding=0, std430) buffer linkedLists {
    NodeType nodes[];
};
uniform uint MaxNodes;

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

...

subroutine(RenderPassType)
void pass1()
{
    // Get the index of the next empty slot in the buffer
    uint nodeIdx = atomicCounterIncrement(nextNodeCounter);

    // Is there space left in the buffer?
    if( nodeIdx < MaxNodes ) {
        // Update the head pointer image
        uint prevHead = imageAtomicExchange(headPointers,
                                             ivec2(gl_FragCoord.xy), nodeIdx);

        // Set the color and depth of this new node to the color
        // and depth of the fragment. The next pointer points to the
        // previous head of the list.
        nodes[nodeIdx].color = vec4(shadeFragment(), Kd.a);
        nodes[nodeIdx].depth = gl_FragCoord.z;
        nodes[nodeIdx].next = prevHead;
    }
}
```

Before rendering the second pass, we need to be sure that all of the data has been written to our buffers. In order to ensure that is indeed the case, we can use a memory barrier.

```
glMemoryBarrier( GL_ALL_BARRIER_BITS );
```

In the second pass, we don't render the scene geometry, just a single, screen-filling quad in order to invoke the fragment shader for each screen pixel. In the fragment shader, we start by copying the linked list for the fragment into a temporary array.

```
struct NodeType frags[MAX_FRAGMENTS];
int count = 0;
```

```
// Get the index of the head of the list
uint n = imageLoad(headPointers, ivec2(gl_FragCoord.xy)).r;

// Copy the linked list for this fragment into an array
while( n != 0xffffffff && count < MAX_FRAGMENTS) {
    frags[count] = nodes[n];
    n = frags[count].next;
    count++;
}
```

Then, we sort the fragments using insertion sort:

```
// Sort the array by depth (largest to smallest).
for( uint i = 1; i < count; i++ )
{
    struct NodeType toInsert = frags[i];
    uint j = i;
    while( j > 0 && toInsert.depth > frags[j-1].depth ) {
        frags[j] = frags[j-1];
        j--;
    }
    frags[j] = toInsert;
}
```

Finally, we blend the fragments "manually", and send the result to the output variable:

```
// Traverse the array, and blend the colors.
vec4 color = vec4(0.5, 0.5, 0.5, 1.0); // Background color
for( int i = 0; i < count; i++ ) {
    color = mix( color, frags[i].color, frags[i].color.a);
}

// Output the final color
FragColor = color;
```

How it works...

To clear our buffers, prior to the first pass, we bind `clearBuf` to the `GL_PIXEL_UNPACK_BUFFER` binding point, and call `glTexSubImage2D` to copy data from `clearBuf` to the the head pointer texture. Note that when a non-zero buffer is bound to `GL_PIXEL_UNPACK_BUFFER`, `glTexSubImage2D` treats the last parameter as an offset into the buffer that is bound there. Therefore, this will initiate a copy from `clearBuf` into `headPtrTex`. Clearing the atomic counter is straightforward, but the use of `glBindBufferBase` may be a bit confusing. If there can be several buffers bound to the binding point (at different indices), then how does `glBufferSubData` know which buffer to target? It turns out that when we bind a buffer using `glBindBufferBase`, it is also bound to the "generic" binding point as well.

In the fragment shader during the first pass, we start with the layout specification enabling the early fragment test optimization.

```
layout (early_fragment_tests) in;
```

This is important because if any fragments are obscured by the opaque geometry, we don't want to add them to a linked list. If the early fragment test optimization is not enabled, the fragment shader may be executed for fragments that will fail the depth test, and hence will get added to the linked list. The previous statement ensures that the fragment shader will not execute for those fragments.

The definition of `struct NodeType` specifies the type of data that is stored in our linked list buffer. We need to store color, depth, and a pointer to the next node in the linked list.

The next three statements declare the objects related to our linked list storage. The first, `headPointers`, is the image object that stores the locations of the heads of each linked list. The layout qualifier indicates that it is located at image unit 0 (refer to the *Getting ready* section of this recipe), and the data type is `r32ui` (red, 32-bit, unsigned integer). The second object is our atomic counter `nextNodeCounter`. The layout qualifier indicates the index within the `GL_ATOMIC_COUTER_BUFFER` binding point (refer to the *Getting ready* section of this recipe), and the offset within the buffer at that location. Since we only have a single value in the buffer, the offset is 0, but in general, you might have several atomic counters located within a single buffer. Third is our linked-list storage buffer `linkedLists`. This is a shader storage buffer object. The organization of the data within the object is defined within the curly braces here. In this case, we just have an array of `NodeType` structures. The bounds of the array can be left undefined, the size being limited by the underlying buffer object that we created. The layout qualifiers define the binding and memory layout. The first, `binding`, indicates that the buffer is located at index 0 within the `GL_SHADER_STORAGE_BUFFER` binding point. The second, `std430`, indicates how memory is organized within the buffer. This is mainly important when we want to read the data back from the OpenGL side. As mentioned previously, this is documented in the OpenGL specification document.

The first step in the fragment shader during the first pass is to increment our atomic counter using `atomicCounterIncrement`. This will increment the counter in such a way that there is no possibility of memory consistency issues if another shader instance is attempting to increment the counter at the same time.



An atomic operation is one that is isolated from other threads and can be considered to be a single, uninterruptable operation. Other threads cannot interleave with an atomic operation. It is always a good idea to use atomic operations when writing to shared data within a shader.

The return value of `atomicCounterIncrement` is the previous value of the counter. It is the next unused location in our linked list buffer. We'll use this value as the location where we'll store this fragment, so we store it in a variable named `nodeIdx`. It will also become the new head of the linked list, so the next step is to update the value in the `headPointers` image at this pixel's location `gl_FragCoord.xy`. We do so using another atomic operation: `imageAtomicExchange`. This replaces the value within the image at the location specified by the second parameter with the value of the third parameter. The return value is the previous value of the image at that location. This is the previous head of our linked list. We hold on to this value in `prevHead`, because we want to link our new head to that node, thereby restoring the consistency of the linked list with our new node at the head.

Finally, we update the node at `nodeIdx` with the color and depth of the fragment, and set the next value to the previous head of the list (`prevHead`). This completes the insertion of this fragment into the linked list at the head of the list.

After the first pass is complete, we need to make sure that all changes are written to our shader storage buffer and image object before proceeding. The only way to guarantee this is to use a memory barrier. The call to `glMemoryBarrier` will take care of this for us. The parameter to `glMemoryBarrier` is the type of barrier. We can "fine tune" the type of barrier to specifically target the kind of data that we want to read. However, just to be safe, and for simplicity, we'll use `GL_ALL_BARRIER_BITS`, which ensures that all possible data has been written.

In the second pass, we start by copying the linked list for the fragment into a temporary array. We start by getting the location of the head of the list from the `headPointers` image using `imageLoad`. Then we traverse the linked list with the `while` loop, copying the data into the array `frags`.

Next, we sort the array by depth from largest to smallest, using the insertion sort algorithm. Insertion sort works well on small arrays, so should be a fairly efficient choice here.

Finally, we combine all the fragments in order, using the `mix` function to blend them together based on the value of the alpha channel. The final result is stored in the output variable `FragColor`.

There's more...

As mentioned previously, we've skipped anything that deals with opaque geometry. In general, one would probably want to render any opaque geometry first, with the depth buffer enabled, and store the rendered fragments in a texture. Then, when rendering the transparent geometry, one would disable writing to the depth buffer, and build the linked list as shown previously. Finally, you could use the value of the opaque texture as the background color when blending the linked lists.

This is the first example in this book that makes use of reading and writing from/to arbitrary (shared) storage from a shader. This capability, only recently introduced, has given us much more flexibility, but that comes at a price. As indicated previously, we have to be very careful to avoid memory consistency and coherence issues. The tools to do so include atomic operations and memory barriers, and this example has just scratched the surface. There's much more to come in *Chapter 10, Using Compute Shaders* when we look at compute shaders, and I recommend you read through the memory chapter in the *OpenGL Programming Guide* for much more detail than I can provide here.

See also

- ▶ *Chapter 10, Using Compute Shaders*
- ▶ *OpenGL Development Cookbook* by Muhammad Moeen Movania has several recipes in *Chapter 6, GPU-based Alpha Blending and Global Illumination*.

6

Using Geometry and Tessellation Shaders

In this chapter, we will cover:

- ▶ Point sprites with the geometry shader
- ▶ Drawing a wireframe on top of a shaded mesh
- ▶ Drawing silhouette lines using the geometry shader
- ▶ Tessellating a curve
- ▶ Tessellating a 2D quad
- ▶ Tessellating a 3D surface
- ▶ Tessellating based on depth

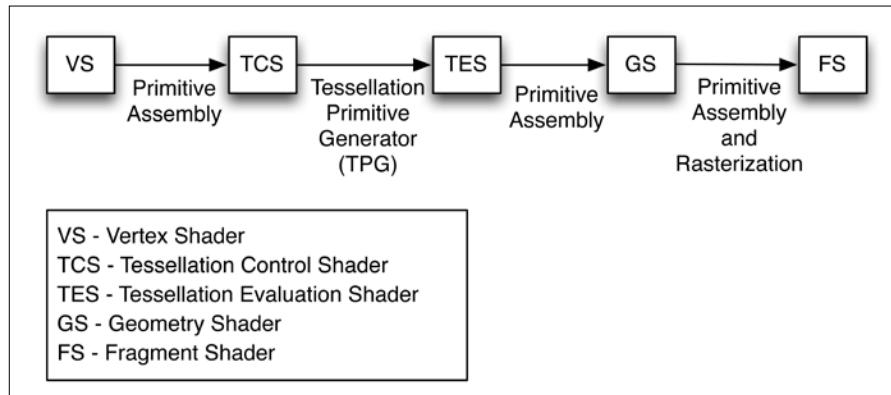
Introduction

Tessellation and geometry shaders are relatively new additions to the OpenGL pipeline, and provide programmers with additional ways to modify geometry as it progresses through the shader pipeline. Geometry shaders can be used to add, modify, or delete geometry, and tessellation shaders can be configured to automatically generate geometry at various levels of detail and to facilitate interpolation based on arbitrary input (patches).

In this chapter, we'll look at several examples of geometry and tessellation shaders in various contexts. However, before we get into the recipes, let's investigate how all of this fits together.

The shader pipeline extended

The following diagram shows a simplified view of the shader pipeline when the shader program includes geometry and tessellation shaders:



The tessellation portion of the shader pipeline includes two stages: the **tessellation control shader (TCS)**, and the **tessellation evaluation shader (TES)**. The geometry shader follows the tessellation stages and precedes the fragment shader. The tessellation shader and geometry shader are optional; however, when a shader program includes a tessellation or geometry shader, a vertex shader must be included.

 Other than the preceding requirement, all shaders are optional. However, when a shader program does not include a vertex or fragment shader, the results are undefined. When using a geometry shader, there is no requirement that you also include a tessellation shader and vice versa. It is rare to have a shader program that does not include at least a fragment shader and a vertex shader.

The geometry shader

The **geometry shader (GS)** is designed to execute once for each primitive. It has access to all of the vertices of the primitive, as well as the values of any input variables associated with each vertex. In other words, if a previous stage (such as the vertex shader) provides an output variable, the geometry shader has access to the value of that variable for all vertices in the primitive. As a result, the input variables within the geometry shader are always arrays.

The geometry shader can output zero, one, or more primitives. Those primitives need not be of the same kind that were received by the geometry shader. However, the GS can only output one primitive type. For example, a GS could receive a triangle, and output several line segments as a line strip. Or a GS could receive a triangle and output zero or many triangles as a triangle strip.

This enables the GS to act in many different ways. A GS could be responsible for culling (removing) geometry based on some criteria, such as visibility based on occlusions. It could generate additional geometry to augment the shape of the object being rendered. The GS could simply compute additional information about the primitive and pass the primitive along unchanged. Or the GS could produce primitives that are entirely different from the input geometry.

The functionality of the GS is centered around the two built-in functions, `EmitVertex` and `EndPrimitive`. These two functions allow the GS to send multiple vertices and primitives down the pipeline. The GS defines the output variables for a particular vertex, and then calls `EmitVertex`. After that, the GS can proceed to re-define the output variables for the next vertex, call `EmitVertex` again, and so on. After emitting all of the vertices for the primitive, the GS can call `EndPrimitive` to let the OpenGL system know that all the vertices of the primitive have been emitted. The `EndPrimitive` function is implicitly called when the GS finishes execution. If a GS does not call `EmitVertex` at all, then the input primitive is effectively dropped (it is not rendered).

In the following recipes, we'll examine a few examples of the geometry shader. In the *Point sprites with the geometry shader* recipe, we'll see an example where the input primitive type is entirely different than the output type. In the *Drawing a wireframe on top of a shaded mesh* recipe, we'll pass the geometry along unchanged, but also produce some additional information about the primitive to help in drawing wireframe lines. In the *Drawing silhouette lines using the geometry shader* recipe, we'll see an example where the GS passes along the input primitive, but generates additional primitives as well.

The tessellation shaders

When the tessellation shaders are active, we can only render one kind of primitive: the patch (`GL_PATCHES`). Rendering any other kind of primitive (such as triangles, or lines) while a tessellation shader is active is an error. The **patch primitive** is an arbitrary "chunk" of geometry (or any information) that is completely defined by the programmer. It has no geometrical interpretation beyond how it is interpreted within the TCS and TES. The number of vertices within the patch primitive is also configurable. The maximum number of vertices per patch is implementation dependent, and can be queried via the following command:

```
glGetIntegerv(GL_MAX_PATCH_VERTICES, &maxVerts);
```

We can define the number of vertices per patch with the following function:

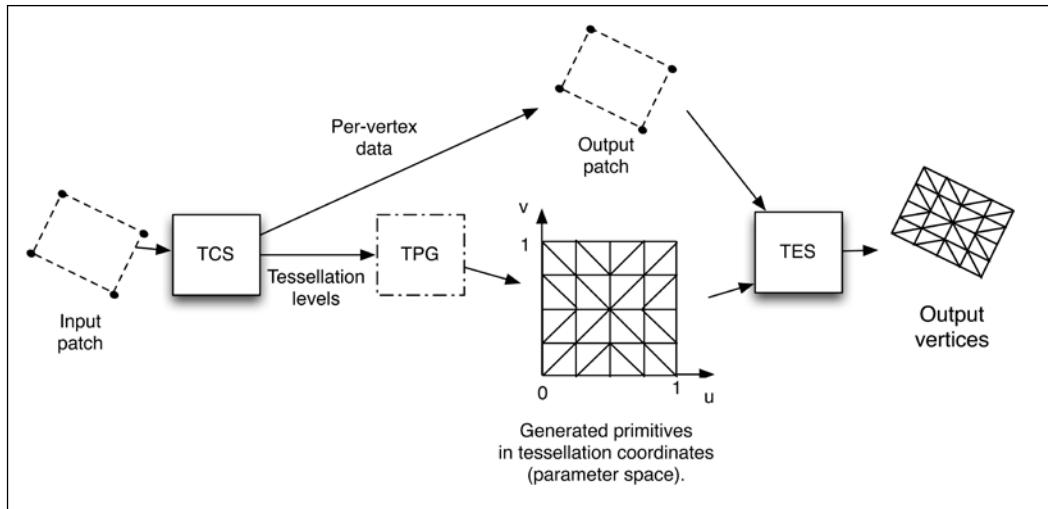
```
glPatchParameteri(GL_PATCH_VERTICES, numPatchVerts);
```

A very common application of this is when the patch primitive consists of a set of control points that define an interpolated surface or curve (such as a Bezier curve or surface). However, there is no reason why the information within the patch primitive couldn't be used for other purposes.

The patch primitive is never actually rendered, instead, it is used as additional information for the TCS and TES. The primitives that actually make their way further down the pipeline are created by the **tessellation primitive generator (TPG)**, which lies between the TCS and the TES. Think of the tessellation primitive generator as a configurable engine that produces primitives based on a set of standard tessellation algorithms. The TCS and the TES have access to the entire input patch, but have fundamentally different responsibilities. The TCS is responsible for setting up the TPG, defining how the primitives should be generated by the TPG (how many and what algorithm to use), and producing per-vertex output attributes. The TES has the job of determining the position (and any other information) of each vertex of the primitives that are produced by the TPG. For example, the TCS might tell the TPG to generate a line strip consisting of 100 line segments, and the TES is responsible for determining the position of each vertex of those 100 line segments. The TES would likely make use of the information within the entire patch primitive in order to do so.

The TCS is executed once for each vertex in the output patch (specified in the TCS code). It can compute additional information about the patch and pass it along to the TES using output variables. However, the most important task of the TCS is to tell the TPG how many primitives it should produce. It does this by defining tessellation levels via the `gl_TessLevelInner` and `gl_TessLevelOuter` arrays. These arrays define the granularity of the tessellation produced by the TPG.

The TPG generates primitives based on a particular algorithm (quads, isolines, or triangles). Each algorithm produces primitives in a slightly different fashion, and we will see examples of isolines and quads in the recipes in this chapter. Each vertex of the generated primitives is associated with a position in parameter space (u , v , w). Each coordinate of this position is a number that can range from zero to one. This coordinate can be used for evaluating the location of the vertex, often by interpolation of the patch primitive's vertices. The primitive generation algorithms produce vertices (and the associated parametric coordinates) in a slightly different fashion. The tessellation algorithms for quads and isolines make use of only the first two parametric coordinates: u and v . The following diagram illustrates the process for an input and output patch consisting of four vertices. In the diagram, the TPG uses the quad tessellation algorithm with inner and outer tessellation levels set at four.



The number of vertices in the input patch need not be the same as the number of vertices in the output patch, although that will be the case in all of the examples in this chapter.

The TES is executed once for each parameter-space vertex that is generated by the TPG. Somewhat strangely, the TES is actually the shader that defines the algorithm used by the TPG. It does so via its input layout qualifier. As stated above, its main responsibility is to determine the position of the vertex (possibly along with other information, such as normal vector and texture coordinate). Typically, the TES uses the parametric coordinate (u, v) provided by the TPG along with the positions of all of the input patch vertices to do so. For example, when drawing a curve, the patch might consist of four vertices, which are the control points for the curve. The TPG would then generate 101 vertices to create a line strip (if the tessellation level was set to 100), and each vertex might have a u coordinate that ranged appropriately between zero and one. The TES would then use that u coordinate along with the positions of the four patch vertices to determine the position of the vertex associated with the shader's execution.

If all of this seems confusing, start with the *Tessellating a curve* recipe, and work your way through the following recipes.

In the *Tessellating a curve* recipe, we'll go through a basic example where we use tessellation shaders to draw a Bezier curve with four control points. In the *Tessellating a 2D quad* recipe, we'll try to understand how the quad tessellation algorithm works by rendering a simple quad and visualizing the triangles produced by the TPG. In the *Tessellating a 3D surface* recipe, we'll use quad tessellation to render a 3D Bezier surface. Finally, in the *Tessellating based on depth* recipe, we'll see how the tessellation shaders make it easy to implement **level-of-detail (LOD)** algorithms.

Point sprites with the geometry shader

Point sprites are simple quads (usually texture mapped) that are aligned such that they are always facing the camera. They are very useful for particle systems in 3D (refer to *Chapter 9, Particles Systems and Animation*) or 2D games. The point sprites are specified by the OpenGL application as single point primitives, via the `GL_POINTS` rendering mode. This simplifies the process, because the quad itself and the texture coordinates for the quad are determined automatically. The OpenGL side of the application can effectively treat them as point primitives, avoiding the need to compute the positions of the quad vertices.

The following screenshot shows a group of point sprites. Each sprite is rendered as a point primitive. The quad and texture coordinates are generated automatically (within the geometry shader) and aligned to face the camera.



OpenGL already has built-in support for point sprites in the `GL_POINTS` rendering mode. When rendering point primitives using this mode, the points are rendered as screen-space squares that have a diameter (side length) as defined by the `glPointSize` function. In addition, OpenGL will automatically generate texture coordinates for the fragments of the square. These coordinates run from zero to one in each direction (left-to-right for `s`, bottom-to-top for `t`), and are accessible in the fragment shader via the `gl_PointCoord` built-in variable.

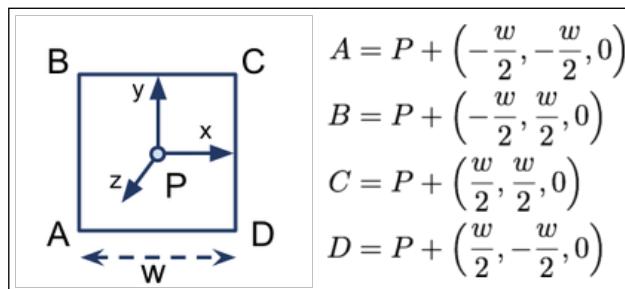
There are various ways to fine-tune the rendering of point sprites within OpenGL. One can define the origin of the automatically generated texture coordinates using the `glPointParameter` functions. The same set of functions also can be used to tweak the way that OpenGL defines the alpha value for points when multisampling is enabled.

The built-in support for point sprites does not allow the programmer to rotate the screen-space squares, or define them as different shapes such as rectangles or triangles. However, one can achieve similar effects with creative use of textures and transformations of the texture coordinates. For example, we could transform the texture coordinates using a rotation matrix to create the look of a rotating object even though the geometry itself is not actually rotating. In addition, the size of the point sprite is a screen-space size. In other words, the point size must be adjusted with the depth of the point sprite if we want to get a perspective effect (sprites get smaller with distance).

If these (and possibly other) issues make the default support for point sprites too limiting, we can use the geometry shader to generate our point sprites. In fact, this technique is a good example of using the geometry shader to generate different kinds of primitives than it receives. The basic idea here is that the geometry shader will receive point primitives (in camera coordinates) and will output a quad centered at the point and aligned so that it is facing the camera. The geometry shader will also automatically generate texture coordinates for the quad.

If desired, we could generate other shapes such as hexagons, or we could rotate the quads before they are output from the geometry shader. The possibilities are endless. Implementing the primitive generation within the geometry shader gives us a great deal of flexibility, but possibly at the cost of some efficiency. The default OpenGL support for point sprites is highly optimized and is likely to be faster in general.

Before jumping directly into the code, let's take a look at some of the mathematics. In the geometry shader, we'll need to generate the vertices of a quad that is centered at a point and aligned with the camera's coordinate system (eye coordinates). Given the point location (P) in camera coordinates, we can generate the vertices of the corners of the quad by simply translating P in a plane parallel to the x-y plane of the camera's coordinate system as shown in the following figure:



The geometry shader will receive the point location in camera coordinates, and output the quad as a triangle strip with texture coordinates. The fragment shader will then just apply the texture to the quad.

Getting ready

For this example, we'll need to render a number of point primitives. The positions can be sent via attribute location 0. There's no need to provide normal vectors or texture coordinates for this one.

The following uniform variables are defined within the shaders, and need to be set within the OpenGL program:

- ▶ `Size2`: This should be half the width of the sprite's square
- ▶ `SpriteTex`: This is the texture unit containing the point sprite texture

As usual, uniforms for the standard transformation matrices are also defined within the shaders, and need to be set within the OpenGL program.

How to do it...

To create a shader program that can be used to render point primitives as quads, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;

void main()
{
    gl_Position = ModelViewMatrix *
                  vec4(VertexPosition, 1.0);
}
```

2. Use the following code for the geometry shader:

```
layout( points ) in;
layout( triangle_strip, max_vertices = 4 ) out;

uniform float Size2; // Half the width of the quad
```

```
uniform mat4 ProjectionMatrix;

out vec2 TexCoord;

void main()
{
    mat4 m = ProjectionMatrix; // Reassign for brevity

    gl_Position = m * (vec4(-Size2,-Size2,0.0,0.0) +
                       gl_in[0].gl_Position);
    TexCoord = vec2(0.0,0.0);
    EmitVertex();

    gl_Position = m * (vec4(Size2,-Size2,0.0,0.0) +
                       gl_in[0].gl_Position);
    TexCoord = vec2(1.0,0.0);
    EmitVertex();

    gl_Position = m * (vec4(-Size2,Size2,0.0,0.0) +
                       gl_in[0].gl_Position);
    TexCoord = vec2(0.0,1.0);
    EmitVertex();

    gl_Position = m * (vec4(Size2,Size2,0.0,0.0) +
                       gl_in[0].gl_Position);
    TexCoord = vec2(1.0,1.0);
    EmitVertex();

    EndPrimitive();
}
```

3. Use the following code for the fragment shader:

```
in vec2 TexCoord; // From the geometry shader

uniform sampler2D SpriteTex;

layout( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = texture(SpriteTex, TexCoord);
```

4. Within the OpenGL render function, render a set of point primitives.

How it works...

The vertex shader is almost as simple as it can get. It converts the point's position to camera coordinates by multiplying by the model-view matrix, and assigns the result to the built-in output variable `gl_Position`.

In the geometry shader, we start by defining the kind of primitive that this geometry shader expects to receive. The first layout statement indicates that this geometry shader will receive point primitives.

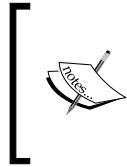
```
layout( points ) in;
```

The next layout statement indicates the kind of primitives produced by this geometry shader, and the maximum number of vertices that will be output.

```
layout( triangle_strip, max_vertices = 4 ) out;
```

In this case, we want to produce a single quad for each point received, so we indicate that the output will be a triangle strip with a maximum of four vertices.

The input primitive is available to the geometry shader via the built-in input variable `gl_in`. Note that it is an array of structures. You might be wondering why this is an array since a point primitive is only defined by a single position. Well, in general the geometry shader can receive triangles, lines, or points (and possibly adjacency information). So, the number of values available may be more than one. If the input were triangles, the geometry shader would have access to three input values (associated with each vertex). In fact, it could have access to as many as six values when `triangles_adjacency` is used (more on that in a later recipe).



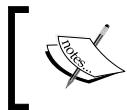
The `gl_in` variable is an array of structs. Each struct contains the following fields: `gl_Position`, `gl_PointSize`, and `gl_ClipDistance []`. In this example, we are only interested in `gl_Position`. However, the others can be set in the vertex shader to provide additional information to the geometry shader.



Within the `main` function of the geometry shader, we produce the quad (as a triangle strip) in the following way. For each vertex of the triangle strip we execute the following steps:

1. Compute the attributes for the vertex (in this case the position and texture coordinate), and assign their values to the appropriate output variables (`gl_Position` and `TexCoord`). Note that the position is also transformed by the projection matrix. We do this because the variable `gl_Position` must be provided in clip coordinates to later stages of the pipeline.
2. Emit the vertex (send it down the pipeline) by calling the built-in function `EmitVertex()`.

Once we have emitted all vertices for the output primitive, we call `EndPrimitive()` to finalize the primitive and send it along.



It is not strictly necessary to call `EndPrimitive()` in this case because it is implicitly called when the geometry shader finishes. However, like closing files, it is good practice to do so anyway.

The fragment shader is also very simple. It just applies the texture to the fragment using the (interpolated) texture coordinate provided by the geometry shader.

There's more...

This example is fairly straightforward and is intended as a gentle introduction to geometry shaders. We could expand on this by allowing the quad to rotate or to be oriented in different directions. We could also use the texture to discard fragments (in the fragment shader) in order to create point sprites of arbitrary shapes. The power of the geometry shader opens up plenty of possibilities!

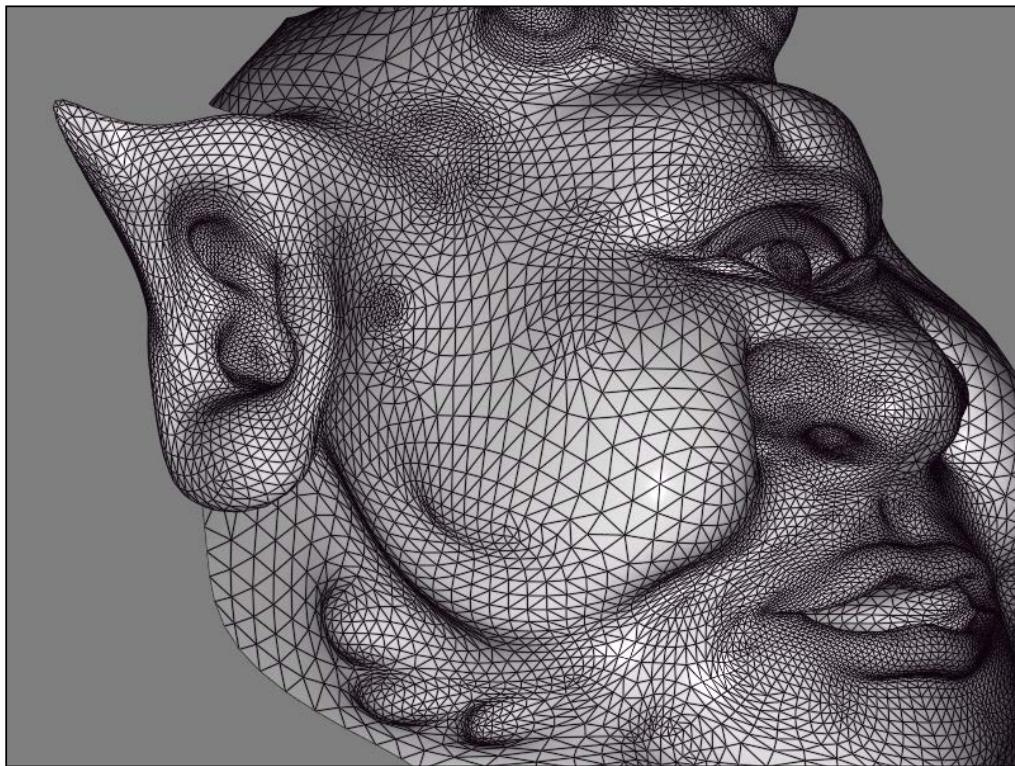
Drawing a wireframe on top of a shaded mesh

The preceding recipe demonstrated the use of a geometry shader to produce a different variety of primitive than it received. Geometry shaders can also be used to provide additional information to later stages. They are quite well suited to do so because they have access to all of the vertices of the primitive at once, and can do computations based on the entire primitive rather than a single vertex.

This example involves a geometry shader that does not modify the triangle at all. It essentially passes the primitive along unchanged. However, it computes additional information about the triangle that will be used by the fragment shader to highlight the edges of the polygon. The basic idea here is to draw the edges of each polygon directly on top of the shaded mesh.

Using Geometry and Tessellation Shaders

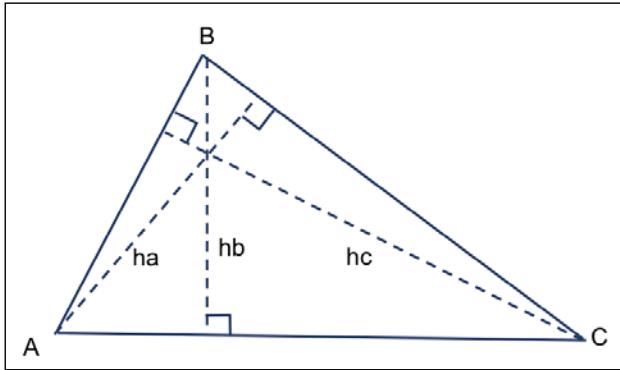
The following figure shows an example of this technique. The mesh edges are drawn on top of the shaded surface by using information computed within the geometry shader.



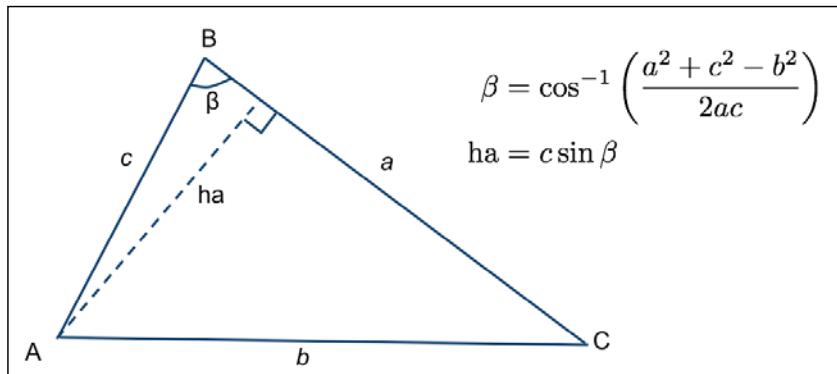
There are many techniques for producing wireframe structures on top of shaded surfaces. This technique comes from an NVIDIA whitepaper published in 2007. We make use of the geometry shader to produce the wireframe and shaded surface in a single pass. We also provide some simple anti-aliasing of the mesh lines that are produced, and the results are quite nice (refer to the preceding figure).

To render the wireframe on top of the shaded mesh, we'll compute the distance from each fragment to the nearest triangle edge. When the fragment is within a certain distance from the edge, it will be shaded and mixed with the edge color. Otherwise, the fragment will be shaded normally.

To compute the distance from a fragment to the edge, we use the following technique. In the geometry shader, we compute the minimum distance from each vertex to the opposite edge (also called the **triangle altitude**). In the following figure, the desired distances are **ha**, **hb**, and **hc**.



We can compute these altitudes using the interior angles of the triangle, which can be determined using the law of cosines. For example, to find **ha**, we use the interior angle at vertex **C** (β).



The other altitudes can be computed in a similar way. (Note that β could be greater than 90 degrees, in which case, we would want the sine of $180-\beta$. However, the sine of $180-\beta$ is the same as the sine of β .)

Once we have computed these triangle altitudes, we can create an output vector (an "edge-distance" vector) within the geometry shader for interpolation across the triangle. The components of this vector represent the distances from the fragment to each edge of the triangle. The x component represents the distance from edge **a**, the y component is the distance from edge **b**, and the z component is the distance from edge **c**. If we assign the correct values to these components at the vertices, the hardware will automatically interpolate them for us to provide the appropriate distances at each fragment. At vertex **A** the value of this vector should be (**ha**, 0, 0) because the vertex **A** is at a distance of **ha** from edge **a** and directly on edges **b** and **c**. Similarly, the value for vertex **B** is (0, **hb**, 0) and for vertex **C** is (0, 0, **hc**). When these three values are interpolated across the triangle, we should have the distance from the fragment to each of the three edges.

We will calculate all of this in screen space. That is, we'll transform the vertices to screen space within the geometry shader before computing the altitudes. Since we are working in screen space, there's no need (and it would be incorrect) to interpolate the values in a perspective correct manner. So we need to be careful to tell the hardware to interpolate linearly.

Within the fragment shader, all we need to do is find the minimum of the three distances, and if that distance is less than the line width, we mix the fragment color with the line color. However, we'd also like to apply a bit of anti-aliasing while we're at it. To do so, we'll fade the edge of the line using the GLSL `smoothstep` function.

We'll scale the intensity of the line in a two-pixel range around the edge of the line. Pixels that are at a distance of one or less from the true edge of the line get 100 percent of the line color, and pixels that are at a distance of one or more from the edge of the line get zero percent of the line color. In between, we'll use the `smoothstep` function to create a smooth transition. Of course, the edge of the line itself is a configurable distance (we'll call it `Line.Width`) from the edge of the polygon.

Getting ready

The typical setup is needed for this example. The vertex position and normal should be provided in attributes zero and one respectively, and you need to provide the appropriate parameters for your shading model. As usual, the standard matrices are defined as uniform variables and should be set within the OpenGL application. However, note that this time we also need the viewport matrix (uniform variable `ViewportMatrix`) in order to transform into screen space.

There are a few uniforms related to the mesh lines that need to be set:

- ▶ `Line.Width`: This should be half the width of the mesh lines
- ▶ `Line.Color`: This is the color of the mesh lines

How to do it...

To create a shader program that utilizes the geometry shader to produce a wireframe on top of a shaded surface, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0 ) in vec3 VertexPosition;
layout (location = 1 ) in vec3 VertexNormal;

out vec3 VNormal;
out vec3 VPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
    VNormal = normalize( NormalMatrix * VertexNormal);
    VPosition = vec3(ModelViewMatrix *
                      vec4(VertexPosition,1.0));
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

2. Use the following code for the geometry shader:

```
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;

out vec3 GNormal;
out vec3 GPosition;
noperspective out vec3 GEdgeDistance;

in vec3 VNormal[];
in vec3 VPosition[];

uniform mat4 ViewportMatrix; // Viewport matrix

void main()
{
    // Transform each vertex into viewport space
    vec3 p0 = vec3(ViewportMatrix * (gl_in[0].gl_Position /
                                      gl_in[0].gl_Position.w));
    vec3 p1 = vec3(ViewportMatrix * (gl_in[1].gl_Position /
                                      gl_in[1].gl_Position.w));
```

```

vec3 p2 = vec3(ViewportMatrix * (gl_in[2].gl_Position /
                                gl_in[2].gl_Position.w));

// Find the altitudes (ha, hb and hc)
float a = length(p1 - p2);
float b = length(p2 - p0);
float c = length(p1 - p0);
float alpha = acos( (b*b + c*c - a*a) / (2.0*b*c) );
float beta = acos( (a*a + c*c - b*b) / (2.0*a*c) );
float ha = abs( c * sin( beta ) );
float hb = abs( c * sin( alpha ) );
float hc = abs( b * sin( alpha ) );

// Send the triangle along with the edge distances
GEdgeDistance = vec3( ha, 0, 0 );
GNormal = VNormal[0];
GPosition = VPosition[0];
gl_Position = gl_in[0].gl_Position;
EmitVertex();

GEdgeDistance = vec3( 0, hb, 0 );
GNormal = VNormal[1];
GPosition = VPosition[1];
gl_Position = gl_in[1].gl_Position;
EmitVertex();

GEdgeDistance = vec3( 0, 0, hc );
GNormal = VNormal[2];
GPosition = VPosition[2];
gl_Position = gl_in[2].gl_Position;
EmitVertex();

EndPrimitive();
}

```

3. Use the following code for the fragment shader:

```

// *** Insert appropriate uniforms for the Phong model ***

// The mesh line settings
uniform struct LineInfo {
    float Width;
    vec4 Color;
} Line;

in vec3 GPosition;

```

```
in vec3 GNormal;
noperspective in vec3 GEdgeDistance;

layout( location = 0 ) out vec4 FragColor;
vec3 phongModel( vec3 pos, vec3 norm )
{
    // *** Phong model evaluation code goes here ***
}

void main() {

    // The shaded surface color.
    vec4 color=vec4(phongModel(GPosition, GNormal), 1.0);

    // Find the smallest distance
    float d = min( GEdgeDistance.x, GEdgeDistance.y );
    d = min( d, GEdgeDistance.z );

    // Determine the mix factor with the line color
    float mixVal = smoothstep( Line.Width - 1,
                               Line.Width + 1, d );

    // Mix the surface color with the line color
    FragColor = mix( Line.Color, color, mixVal );
}
```

How it works...

The vertex shader is pretty simple. It passes the normal and position along to the geometry shader after converting them into camera coordinates. The built-in variable `gl_Position` gets the position in clip coordinates. We'll use this value in the geometry shader to determine the screen space coordinates.

In the geometry shader, we begin by defining the input and output primitive types for this shader.

```
layout( triangles ) in;
layout( triangle_strip, max_vertices = 3 ) out;
```

We don't actually change anything about the geometry of the triangle, so the input and output types are essentially the same. We will output exactly the same triangle that was received as input.

The output variables for the geometry shader are `GNormal`, `GPosition`, and `GEdgeDistance`. The first two are simply the values of the normal and position in camera coordinates, passed through unchanged. The third is the vector that will store the distance to each edge of the triangle (described previously). Note that it is defined with the `noperspective` qualifier.

```
noperspective out vec3 GEdgeDistance;
```

The `noperspective` qualifier indicates that the values are to be interpolated linearly, instead of the default perspective correct interpolation. As mentioned previously, these distances are in screen space, so it would be incorrect to interpolate them in a non-linear fashion.

Within the `main` function, we start by transforming the position of each of the three vertices of the triangle from clip coordinates to screen space coordinates by multiplying with the viewport matrix. (Note that it is also necessary to divide by the `w` coordinate as the clip coordinates are homogeneous and may need to be converted back to true Cartesian coordinates.)

Next, we compute the three altitudes `ha`, `hb`, and `hc` using the law of cosines as described earlier.

Once we have the three altitudes, we set `GEdgeDistance` appropriately for the first vertex; pass along `GNormal`, `GPosition`, and `gl_Position` unchanged; and emit the first vertex by calling `EmitVertex()`. This finishes the vertex and emits the vertex position and all of the per-vertex output variables. We then proceed similarly for the other two vertices of the triangle, finishing the polygon by calling `EndPrimitive()`.

In the fragment shader, we start by evaluating the basic shading model and storing the resulting color in `color`. At this stage in the pipeline, the three components of the `GEdgeDistance` variable should contain the distance from this fragment to each of the three edges of the triangle. We are interested in the minimum distance, so we find the minimum of the three components and store that in the `d` variable. The `smoothstep` function is then used to determine how much to mix the line color with the shaded color (`mixVal`).

```
float mixVal = smoothstep( Line.Width - 1,  
                           Line.Width + 1, d );
```

If the distance is less than `Line.Width - 1`, then `smoothstep` will return a value of 0, and if it is greater than `Line.Width + 1`, it will return 1. For values of `d` that are in between the two, we'll get a smooth transition. This gives us a value of 0 when inside the line, a value of 1 when outside the line, and in a two pixel area around the edge, we'll get a smooth variation between 0 and 1. Therefore, we can use the result directly to mix the color with the line color.

Finally, the fragment color is determined by mixing the shaded color with the line color using `mixVal` as the interpolation parameter.

There's more...

This technique produces very nice looking results and has relatively few drawbacks. It is a good example of how geometry shaders can be useful for tasks other than modification of the actual geometry. In this case, we used the geometry shader simply to compute additional information about the primitive as it was being sent down the pipeline.

This shader can be dropped in and applied to any mesh without any modification to the OpenGL side of the application. It can be useful when debugging mesh issues or when implementing a mesh modeling program.

Other common techniques for accomplishing this effect typically involve rendering the shaded object and wireframe in two passes with a polygon offset (via the `glPolygonOffset` function) applied to avoid the "z-fighting", which takes place between the wireframe and the shaded surface beneath. This technique is not always effective because the modified depth values might not always be correct, or as desired, and it can be difficult to find the "sweet-spot" for the polygon offset value. For a good survey of techniques, refer to *Section 11.4.2 in Real Time Rendering, third edition*, by T Akenine-Moller, E Haines, and N Hoffman, AK Peters, 2008.

See also...

- ▶ This technique was originally published in an NVIDIA whitepaper in 2007 (*Solid Wireframe, NVIDIA Whitepaper WP-03014-001_v01* available at developer.nvidia.com). The whitepaper was listed as a Direct3D example, but of course our implementation here is provided in OpenGL.
- ▶ The *Creating shadows using shadow volumes and the geometry shader* recipe in Chapter 7, Shadows.
- ▶ The *Implementing per-vertex ambient, diffuse, and specular (ADS) shading* recipe in Chapter 2, The Basics of GLSL Shaders.

Drawing silhouette lines using the geometry shader

When a cartoon or hand-drawn effect is desired, we often want to draw black outlines around the edges of a model and along ridges or creases (silhouette lines). In this recipe, we'll discuss one technique for doing this using the geometry shader, to produce the additional geometry for the silhouette lines. The geometry shader will approximate these lines by generating small, skinny quads aligned with the edges that make up the silhouette of the object.

The following figure shows the ogre mesh with black silhouette lines generated by the geometry shader. The lines are made up of small quads that are aligned with certain mesh edges.

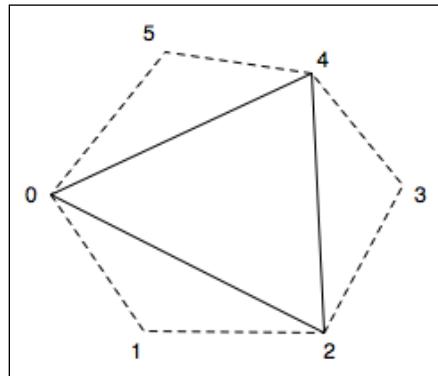


The technique shown in this recipe is based on a technique published in a recent blog post by *Philip Rideout* (prideout.net/blog/?p=54). His implementation uses two passes (base geometry and silhouette), and includes many optimizations, such as anti-aliasing and custom depth testing (with g-buffers). To keep things simple, as our main goal is to demonstrate the features of the geometry shader, we'll implement the technique using a single pass without anti-aliasing or custom depth testing. If you are interested in adding these additional features, refer to Philip's excellent blog posting.

One of the most important features of the geometry shader is that it allows us to provide additional vertex information beyond just the primitive being rendered. When geometry shaders were introduced into OpenGL, several additional primitive rendering modes were also introduced. These "adjacency" modes allow additional vertex data to be associated with each primitive. Typically, this additional information is related to the nearby primitives within a mesh, but there is no requirement that this be the case (we could actually use the additional information for other purposes if desired). The following list includes the adjacency modes along with a short description:

- ▶ **GL_LINES_ADJACENCY:** This mode defines lines with adjacent vertices (four vertices per line segment)
- ▶ **GL_LINE_STRIP_ADJACENCY:** This mode defines a line strip with adjacent vertices (for n lines, there are n+3 vertices)
- ▶ **GL_TRIANGLES_ADJACENCY:** This mode defines triangles along with vertices of adjacent triangles (six vertices per primitive)
- ▶ **GL_TRIANGLE_STRIP_ADJACENCY:** This mode defines a triangle strip along with vertices of adjacent triangles (for n triangles, there are 2(n+2) vertices provided)

For full details on each of these modes, check out the official OpenGL documentation. In this recipe, we'll use the `GL_TRIANGLES_ADJACENCY` mode to provide information about adjacent triangles in our mesh. With this mode, we provide six vertices per primitive. The following diagram illustrates the locations of these vertices:



In the preceding diagram, the solid line represents the triangle itself, and the dotted lines represent adjacent triangles. The first, third, and fifth vertices (**0**, **2**, and **4**) make up the triangle itself. The second, fourth, and sixth are vertices that make up the adjacent triangles.

Mesh data is not usually provided in this form, so we need to preprocess our mesh to include the additional vertex information. Typically, this only means expanding the element index array by a factor of two. The position, normal, and texture coordinate arrays can remain unchanged.

When a mesh is rendered with adjacency information, the geometry shader has access to all six vertices associated with a particular triangle. We can then use the adjacent triangles to determine whether or not a triangle edge is part of the silhouette of the object. The basic assumption is that an edge is a silhouette edge if the triangle is front facing and the corresponding adjacent triangle is not front facing.

We can determine whether or not a triangle is front facing within the geometry shader by computing the triangle's normal vector (using a cross product). If we are working within eye coordinates (or clip coordinates), the z coordinate of the normal vector will be positive for front facing triangles. Therefore, we only need to compute the z coordinate of the normal vector, which should save a few cycles. For a triangle with vertices A, B, and C, the z coordinate of the normal vector is given by the following equation:

$$n_z = (A_x B_y - B_x A_y) + (B_x C_y - C_x B_y) + (C_x A_y - A_x C_y)$$

Once we determine which edges are silhouette edges, the geometry shader will produce additional skinny quads aligned with the silhouette edge. These quads, taken together, will make up the desired dark lines (refer to the preceding figure). After generating all the silhouette quads, the geometry shader will output the original triangle.

In order to render the mesh in a single pass with appropriate shading for the base mesh, and no shading for the silhouette lines, we'll use an additional output variable. This variable will let the fragment shader know when we are rendering the base mesh and when we are rendering the silhouette edge.

Getting ready

Set up your mesh data so that adjacency information is included. As just mentioned, this probably requires expanding the element index array to include the additional information. This can be done by passing through your mesh and looking for shared edges. Due to space limitations, we won't go through the details here, but the blog post mentioned some time back has some information about how this might be done. Also, the source code for this example contains a simple (albeit not very efficient) technique.

The important uniform variables for this example are as follows:

- ▶ **EdgeWidth**: This is the width of the silhouette edge in clip (normalized device) coordinates
- ▶ **PctExtend**: This is a percentage to extend the quads beyond the edge
- ▶ **LineColor**: This is the color of the silhouette edge lines

As usual, there are also the appropriate uniforms for the shading model, and the standard matrices.

How to do it...

To create a shader program that utilizes the geometry shader to render silhouette edges, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0 ) in vec3 VertexPosition;
layout (location = 1 ) in vec3 VertexNormal;

out vec3 VNormal;
out vec3 VPosition;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;
void main()
{
    VNormal = normalize( NormalMatrix * VertexNormal);
```

```
VPosition = vec3(ModelViewMatrix *  
                  vec4(VertexPosition,1.0));  
gl_Position = MVP * vec4(VertexPosition,1.0);  
}
```

2. Use the following code for the geometry shader:

```
layout( triangles_adjacency ) in;  
layout( triangle_strip, max_vertices = 15 ) out;  
  
out vec3 GNormal;  
out vec3 GPosition;  
  
// Which output primitives are silhouette edges  
flat out bool GIIsEdge;  
  
in vec3 VNormal[]; // Normal in camera coords.  
in vec3 VPosition[]; // Position in camera coords.  
  
uniform float EdgeWidth; // Width of sil. edge in clip cds.  
uniform float PctExtend; // Percentage to extend quad  
  
bool isFrontFacing( vec3 a, vec3 b, vec3 c )  
{  
    return ((a.x * b.y - b.x * a.y) +  
            (b.x * c.y - c.x * b.y) +  
            (c.x * a.y - a.x * c.y)) > 0;  
}  
void emitEdgeQuad( vec3 e0, vec3 e1 )  
{  
    vec2 ext = PctExtend * (e1.xy - e0.xy);  
    vec2 v = normalize(e1.xy - e0.xy);  
    vec2 n = vec2(-v.y, v.x) * EdgeWidth;  
  
    // Emit the quad  
    GIIsEdge = true; // This is part of the sil. edge  
  
    gl_Position = vec4( e0.xy - ext, e0.z, 1.0 );  
    EmitVertex();  
    gl_Position = vec4( e0.xy - n - ext, e0.z, 1.0 );  
    EmitVertex();  
    gl_Position = vec4( e1.xy + ext, e1.z, 1.0 );  
    EmitVertex();  
    gl_Position = vec4( e1.xy - n + ext, e1.z, 1.0 );  
    EmitVertex();
```

```
    EndPrimitive();
}

void main()
{
    vec3 p0 = gl_in[0].gl_Position.xyz /
              gl_in[0].gl_Position.w;
    vec3 p1 = gl_in[1].gl_Position.xyz /
              gl_in[1].gl_Position.w;
    vec3 p2 = gl_in[2].gl_Position.xyz /
              gl_in[2].gl_Position.w;
    vec3 p3 = gl_in[3].gl_Position.xyz /
              gl_in[3].gl_Position.w;
    vec3 p4 = gl_in[4].gl_Position.xyz /
              gl_in[4].gl_Position.w;
    vec3 p5 = gl_in[5].gl_Position.xyz /
              gl_in[5].gl_Position.w;

    if( isFrontFacing(p0, p2, p4) ) {
        if( ! isFrontFacing(p0,p1,p2) )
            emitEdgeQuad(p0,p2);
        if( ! isFrontFacing(p2,p3,p4) )
            emitEdgeQuad(p2,p4);
        if( ! isFrontFacing(p4,p5,p0) )
            emitEdgeQuad(p4,p0);
    }

    // Output the original triangle
    GIIsEdge = false; // Triangle is not part of an edge.

    GNormal = VNormal[0];
    GPosition = VPosition[0];
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
    GNormal = VNormal[2];
    GPosition = VPosition[2];
    gl_Position = gl_in[2].gl_Position;
    EmitVertex();

    GNormal = VNormal[4];
    GPosition = VPosition[4];
    gl_Position = gl_in[4].gl_Position;
    EmitVertex();

    EndPrimitive();
}
```

3. Use the following code for the fragment shader:

```
/** Light and material uniforms go here ***/  
  
uniform vec4 LineColor; // The sil. edge color  
  
in vec3 GPosition; // Position in camera coords  
in vec3 GNormal; // Normal in camera coords.  
  
flat in bool GIIsEdge; // Whether or not we're drawing an edge  
  
layout( location = 0 ) out vec4 FragColor;  
  
vec3 toonShade( )  
{  
    // *** toon shading algorithm from Chapter 3 ***  
}  
  
void main()  
{  
    // If we're drawing an edge, use constant color,  
    // otherwise, shade the poly.  
    if( GIIsEdge ) {  
        FragColor = LineColor;  
    } else {  
        FragColor = vec4( toonShade(), 1.0 );  
    }  
}
```

How it works...

The vertex shader is a simple "pass-through" shader. It converts the vertex position and normal to camera coordinates and sends them along, via `VPosition` and `VNormal`. These will be used for shading within the fragment shader and will be passed along (or ignored) by the geometry shader. The position is also converted to clip coordinates (or normalized device coordinates) by transforming with the model-view projection matrix, and it is then assigned to the built-in `gl_Position`.

The geometry shader begins by defining the input and output primitive types using the `layout` directive.

```
layout( triangles_adjacency ) in;  
layout( triangle_strip, max_vertices = 15 ) out;
```

This indicates that the input primitive type is triangles with adjacency information, and the output type is triangle strips. This geometry shader will produce a single triangle (the original triangle) and at most one quad for each edge. This corresponds to a maximum of 15 vertices that could be produced, and we indicate that maximum within the output layout directive.

The output variable `GIsEdge` is used to indicate to the fragment shader whether or not the polygon is an edge quad. The fragment shader will use this value to determine whether or not to shade the polygon. There is no need to interpolate the value and since it is a Boolean, interpolation doesn't quite make sense, so we use the `flat` qualifier.

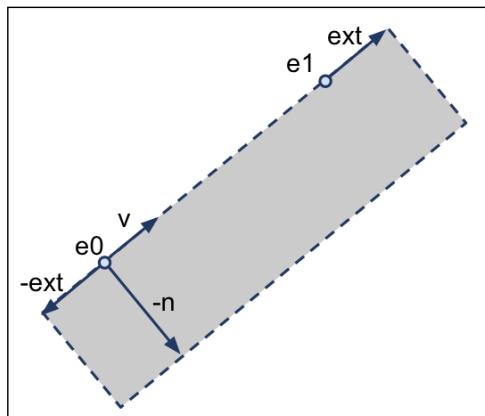
The first few lines within the `main` function take the position for each of the six vertices (in clip coordinates) and divides it by the fourth coordinate in order to convert from its homogeneous representation to the true Cartesian value. This is necessary if we are using a perspective projection, but is not necessary for orthographic projections.

Next, we determine whether the main triangle (defined by points 0, 2, and 4) is front facing. The function `isFrontFacing`, returns whether or not the triangle defined by its three parameters is front facing using the equation described previously. If the main triangle is front facing, then we will emit a silhouette edge quad only if the adjacent triangle is not front facing.

The function `emitEdgeQuad` produces a quad that is aligned with an edge defined by the points `e0` and `e1`. It begins by computing `ext`, which is the vector from `e0` to `e1`, scaled by `PctExtend` (in order to slightly lengthen the edge quad). We lengthen the edge quad in order to cover gaps that may appear between quads (we'll discuss this further in *There's more...*).

Note also that we drop the z coordinate here. As the points are defined in clip coordinates, and we are going to produce a quad that is aligned with the x-y plane (facing the camera), we want to compute the positions of the vertices by translating within the x-y plane. Therefore we can ignore the z coordinate for now. We'll use its value unchanged in the final position of each vertex.

Next, the variable `v` is assigned to the normalized vector from `e0` to `e1`. The variable `n` gets a vector that is perpendicular to `v` (in 2D this can be achieved by swapping the x and y coordinates and negating the new x coordinate). This is just a counter-clockwise 90 degree rotation in 2D. We scale the vector `n` by `EdgeWidth` because we want the length of the vector to be the same as the width of the quad. The two vectors `ext` and `n` will be used to determine the vertices of the quad as shown in the following figure:



The four corners of the quad are given by: **e0 - ext**, **e0 - n - ext**, **e1 + ext**, and **e1 - n + ext**. The z coordinate for the lower two vertices is the same as the z coordinate for **e0**, and the z coordinate for the upper two vertices is the z coordinate for **e1**.

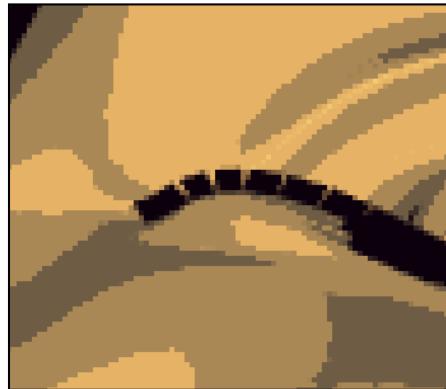
We then finish up the `emitEdgeQuad` function by setting `GIsEdge` to `true` in order to let the fragment shader know that we are rendering a silhouette edge, and then emitting the four vertices of the quad. The function ends with a call to `EndPrimitive` to terminate the processing of the triangle strip for the quad.

Back within the `main` function, after producing the silhouette edges, we proceed by emitting the original triangle unchanged. `vNormal`, `vPosition`, and `gl_Position` for vertices 0, 2, and 4 are passed along without any modification to the fragment shader. Each vertex is emitted with a call to `EmitVertex`, and the primitive is completed with `EndPrimitive`.

Within the fragment shader we either shade the fragment (using the toon shading algorithm), or simply give the fragment a constant color. The `GIsEdge` input variable will indicate which option to choose. If `GIsEdge` is true, then we are rendering a silhouette edge so the fragment is given the line color. Otherwise, we are rendering a mesh polygon, so we shade the fragment using the toon shading technique from *Chapter 3, Lighting, Shading Effects, and Optimizations*.

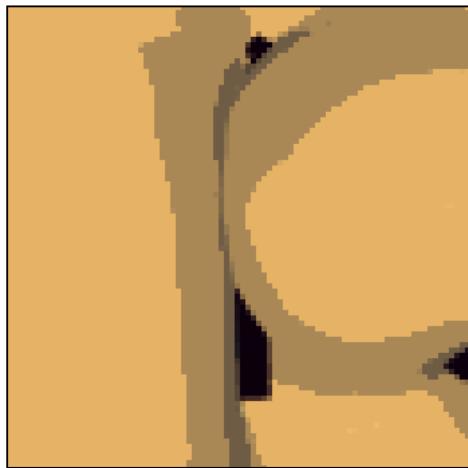
There's more...

One of the problems with the preceding technique is that "feathering" can occur due to the gaps between consecutive edge quads.



The preceding figure shows the feathering of a silhouette edge. The gaps between the polygons can be filled with triangles, but in our example, we simply extend the length of each quad to fill in the gap. This can, of course, cause artifacts if the quads are extended too far, but in practice they haven't been very distracting in my experience.

A second issue is related to depth testing. If an edge polygon extends into another area of the mesh, it can be clipped due to the depth test. The following is an example:



The edge polygon should extend vertically throughout the middle of the preceding figure, but is clipped because it falls behind the part of the mesh that is nearby. This issue can be solved by using custom depth testing when rendering the silhouette edges. Refer to the blog post mentioned earlier for details on this technique. It may also be possible to turn depth testing off when rendering the edges, being careful not to render any edges from the opposite side of the model.

See also

- ▶ A whitepaper on using the geometry shader for fur and fins: <http://developer.download.nvidia.com/whitepapers/2007/SDK10/FurShellsAndFins.pdf>
- ▶ The *Creating shadows using shadow volumes and the geometry shader* recipe in *Chapter 7, Shadows*
- ▶ The *Creating a cartoon shading effect* recipe in *Chapter 3, Lighting, Shading, and Optimization*

Tessellating a curve

In this recipe, we'll take a look at the basics of tessellation shaders by drawing a **cubic Bezier curve**. A Bezier curve is a parametric curve defined by four control points. The control points define the overall shape of the curve. The first and last of the four points define the start and end of the curve, and the middle points guide the shape of the curve, but do not necessarily lie directly on the curve itself. The curve is defined by interpolating the four control points using a set of **blending functions**. The blending functions define how much each control point contributes to the curve for a given position along the curve. For Bezier curves, the blending functions are known as the **Bernstein polynomials**.

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

In the preceding equation, the first term is the binomial coefficient function (shown in the following equation), **n** is the degree of the polynomial, **i** is the polynomial number, and **t** is the parametric parameter.

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

The general parametric form for the Bezier curve is then given as a sum of the products of the Bernstein polynomials with the control points (\mathbf{P}_i).

$$P(t) = \sum_{i=0}^n B_i^n(t) P_i$$

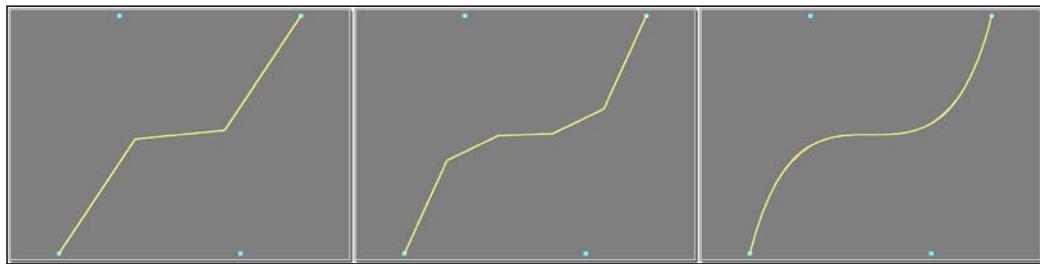
In this example, we will draw a cubic Bezier curve, which involves four control points ($n = 3$).

$$P(t) = B_0^3(t)P_0 + B_1^3(t)P_1 + B_2^3(t)P_2 + B_3^3(t)P_3$$

And the cubic Bernstein polynomials are:

$$\begin{aligned} B_0^3(t) &= (1-t)^3 \\ B_1^3(t) &= 3(1-t)^2t \\ B_2^3(t) &= 3(1-t)t^2 \\ B_3^3(t) &= t^3 \end{aligned}$$

As stated in the introduction of this chapter, the tessellation functionality within OpenGL involves two shader stages. They are the tessellation control shader (TCS) and the tessellation evaluation shader (TES). In this example, we'll define the number of line segments for our Bezier curve within the TCS (by defining the outer tessellation levels), and evaluate the Bezier curve at each particular vertex location within the TES. The following screenshot shows the output of this example for three different tessellation levels. The left figure uses three line segments (level 3), the middle uses level 5, and the right-hand figure is created with tessellation level 30. The small squares are the control points.



The control points for the Bezier curve are sent down the pipeline as a patch primitive consisting of four vertices. A patch primitive is a programmer-defined primitive type. Basically, it is a set of vertices that can be used for anything that the programmer chooses. The TCS is executed once for each vertex within the patch, and the TES is executed, a variable number of times, depending on the number of vertices produced by the TPG. The final output of the tessellation stages is a set of primitives. In our case, it will be a line strip.

Part of the job for the TCS is to define the tessellation level. In very rough terms, the tessellation level is related to the number of vertices that will be generated. In our case, the TCS will be generating a line strip, so the tessellation level is the number of line segments in the line strip. Each vertex that is generated for this line strip will be associated with a tessellation coordinate that will vary between zero and one. We'll refer to this as the *u* coordinate, and it will correspond to the parametric parameter *t* in the preceding Bezier curve equation.

What we've looked at so far is not, in fact, the whole story. Actually, the TCS will trigger a generation of a set of line strips called isolines. Each vertex in this set of isolines will have a *u* and a *v* coordinate. The *u* coordinate will vary from zero to one along a given isoline, and *v* will be constant for each isoline. The number of distinct values of *u* and *v* is associated with two separate tessellation levels, the so-called "outer" levels. For this example, however, we'll only generate a single line strip, so the second tessellation level (for *v*) will always be one.



Within the TES, the main task is to determine the position of the vertex associated with this execution of the shader. We have access to the u and v coordinates associated with the vertex, and we also have (read-only) access to all of the vertices of the patch. We can then determine the appropriate position for the vertex by using the parametric equation described above, with u as the parametric coordinate (t in the preceding equation).

Getting ready

The following are the important uniform variables for this example:

- ▶ `NumSegments`: This is the number of line segments to be produced.
- ▶ `NumStrips`: This is the number of isolines to be produced. For this example, this should be set to one.
- ▶ `LineColor`: This is the color for the resulting line strip.

Set the uniform variables within the main OpenGL application. There are a total of four shaders to be compiled and linked. They are the vertex, fragment, tessellation control, and tessellation evaluation shaders.

How to do it...

To create a shader program that will generate a Bezier curve from a patch of four control points, use the following steps:

1. Use the following code for the simple vertex shader:

```
layout (location = 0 ) in vec2 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Use the following code as the tessellation control shader:

```
layout( vertices=4 ) out;

uniform int NumSegments;
uniform int NumStrips;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
    // Define the tessellation levels
    gl_TessLevelOuter[0] = float(NumStrips);
```

```
    gl_TessLevelOuter[1] = float(NumSegments);  
}
```

3. Use the following code as the tessellation evaluation shader:

```
layout( isolines ) in;  
uniform mat4 MVP; // projection * view * model  
  
void main()  
{  
    // The tessellation u coordinate  
    float u = gl_TessCoord.x;  
  
    // The patch vertices (control points)  
    vec3 p0 = gl_in[0].gl_Position.xyz;  
    vec3 p1 = gl_in[1].gl_Position.xyz;  
    vec3 p2 = gl_in[2].gl_Position.xyz;  
    vec3 p3 = gl_in[3].gl_Position.xyz;  
  
    float u1 = (1.0 - u);  
    float u2 = u * u;  
  
    // Bernstein polynomials evaluated at u  
    float b3 = u2 * u;  
    float b2 = 3.0 * u2 * u1;  
    float b1 = 3.0 * u * u1 * u1;  
    float b0 = u1 * u1 * u1;  
  
    // Cubic Bezier interpolation  
    vec3 p = p0 * b0 + p1 * b1 + p2 * b2 + p3 * b3;  
  
    gl_Position = MVP * vec4(p, 1.0);  
}
```

4. Use the following code for the fragment shader:

```
uniform vec4 LineColor;  
  
layout ( location = 0 ) out vec4 FragColor;  
  
void main()  
{  
    FragColor = LineColor;  
}
```

5. It is important to define the number of vertices per patch within the OpenGL application. You can do so using the `glPatchParameter` function:

```
glPatchParameteri( GL_PATCH_VERTICES, 4);
```

6. Render the four control points as a patch primitive within the OpenGL application's render function:

```
glDrawArrays(GL_PATCHES, 0, 4);
```

How it works...

The vertex shader is just a "pass-through" shader. It sends the vertex position along to the next stage without any modification.

The tessellation control shader begins by defining the number of vertices in the output patch:

```
layout (vertices = 4) out;
```

Note that this is not the same as the number of vertices that will be produced by the tessellation process. In this case, the patch is our four control points, so we use a value of four.

The main method within the TCS passes the input position (of the patch vertex) to the output position without modification. The arrays `gl_out` and `gl_in` contain the input and output information associated with each vertex in the patch. Note that we assign and read from location `gl_InvocationID` in these arrays. The `gl_InvocationID` variable defines the output patch vertex for which this invocation of the TCS is responsible. The TCS can access all of the array `gl_in`, but should only write to the location in `gl_out` corresponding to `gl_InvocationID`.

Next, the TCS sets the tessellation levels by assigning to the `gl_TessLevelOuter` array. Note that the values for `gl_TessLevelOuter` are floating point numbers rather than integers. They will be rounded up to the nearest integer and clamped automatically by the OpenGL system.

The first element in the array defines the number of isolines that will be generated. Each isoline will have a constant value for `v`. In this example, the value of `gl_TessLevelOuter[0]` should be one. The second defines the number of line segments that will be produced in the line strip. Each vertex in the strip will have a value for the parametric `u` coordinate that will vary from zero to one.

In the TES, we start by defining the input primitive type using a layout declaration:

```
layout (isolines) in;
```

This indicates the type of subdivision that is performed by the tessellation primitive generator. Other possibilities here include `quads` and `triangles`.

Within the `main` function of the TES, the variable `gl_TessCoord` contains the tessellation `u` and `v` coordinates for this invocation. As we are only tessellating in one dimension, we only need the `u` coordinate, which corresponds to the `x` coordinate of `gl_TessCoord`.

The next step accesses the positions of the four control points (all the points in our patch primitive). These are available in the `gl_in` array.

The cubic Bernstein polynomials are then evaluated at `u` and stored in `b0`, `b1`, `b2`, and `b3`. Next, we compute the interpolated position using the Bezier curve equation described some time back. The final position is converted to clip coordinates and assigned to the output variable `gl_Position`.

The fragment shader simply applies `LineColor` to the fragment.

There's more...

There's a lot more to be said about tessellation shaders, but this example is intended to be a simple introduction so we'll leave that for the following recipes. Next, we'll look at tessellation across surfaces in two dimensions.

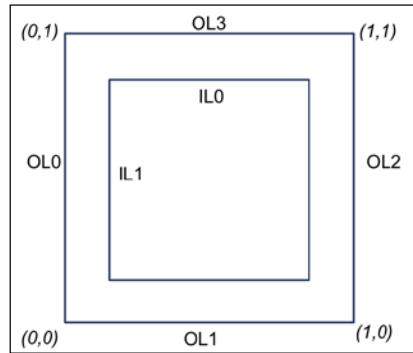
Tessellating a 2D quad

One of the best ways to understand OpenGL's hardware tessellation is to visualize the tessellation of a 2D quad. When linear interpolation is used, the triangles that are produced are directly related to the tessellation coordinates (u, v) that are produced by the tessellation primitive generator. It can be extremely helpful to draw a few quads with different inner and outer tessellation levels, and study the triangles produced. We will do exactly that in this recipe.

When using quad tessellation, the tessellation primitive generator subdivides (u, v) parameter space into a number of subdivisions based on six parameters. These are the inner tessellation levels for u and v (inner level 0 and inner level 1), and the outer tessellation levels for u and v along both edges (outer levels 0 to 3). These determine the number of subdivisions along the edges of parameter space and internally. Let's look at each of these individually:

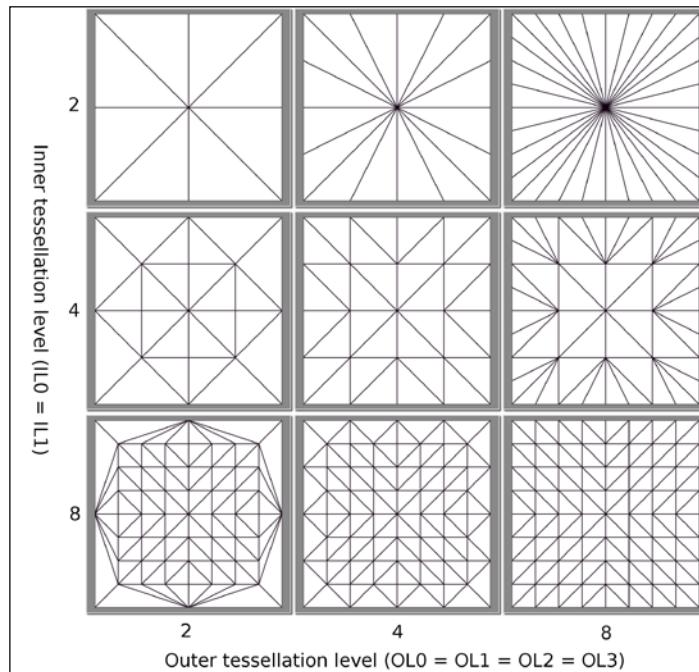
- ▶ **Outer level 0 (OL0):** This is the number of subdivisions along the v direction where $u = 0$
- ▶ **Outer level 1 (OL1):** This is the number of subdivisions along the u direction where $v = 0$
- ▶ **Outer level 2 (OL2):** This is the number of subdivisions along the v direction where $u = 1$
- ▶ **Outer level 3 (OL3):** This is the number of subdivisions along the u direction where $v = 1$
- ▶ **Inner level 0 (IL0):** This is the number of subdivisions along the u direction for all internal values of v
- ▶ **Inner level 1 (IL1):** This is the number of subdivisions along the v direction for all internal values of u

The following diagram represents the relationship between the tessellation levels and the areas of parameter space that are affected by each. The outer levels defines the number of subdivisions along the edges, and the inner levels define the number of subdivisions internally.



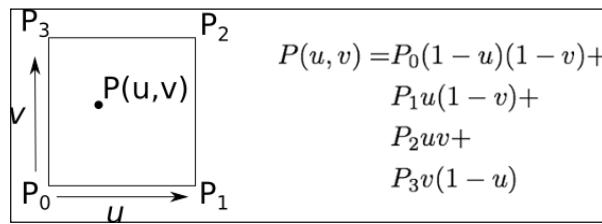
The six tessellation levels described some time back can be configured via the arrays `gl_TessLevelOuter` and `gl_TessLevelInner`. For example, `gl_TessLevelInner[0]` corresponds to **IL0**, `gl_TessLevelOuter[2]` corresponds to **OL2**, and so on.

If we draw a patch primitive that consists of a single quad (four vertices), and use linear interpolation, the triangles that result can help us to understand how OpenGL does quad tessellation. The following diagram shows the results for various tessellation levels:



When we use linear interpolation, the triangles that are produced represent a visual representation of parameter (u, v) space. The x axis corresponds to the u coordinate and the y axis corresponds to the v coordinate. The vertices of the triangles are the (u, v) coordinates generated by the tessellation primitive generator. The number of subdivisions can be clearly seen in the mesh of triangles. For example, when the outer levels are set to 2 and the inner levels are set to 8, you can see that the outer edges have two subdivisions, but within the quad, u and v are subdivided into 8 intervals.

Before jumping into the code, let's discuss linear interpolation. If the four corners of the quad are as shown in the following figure, then any point within the quad can be determined by linearly interpolating the four corners with respect to parameters u and v .



We'll let the tessellation primitive generator create a set of vertices with appropriate parametric coordinates, and we'll determine the corresponding positions by interpolating the corners of the quad using the preceding equation.

Getting ready

The outer and inner tessellation levels will be determined by the uniform variables `Inner` and `Outer`. In order to display the triangles, we will use the geometry shader described earlier in this chapter.

Set up your OpenGL application to render a patch primitive consisting of four vertices in counter clockwise order as shown in the preceding figure.

How to do it...

To create a shader program that will generate a set of triangles using quad tessellation from a patch of four vertices, use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0 ) in vec2 VertexPosition;

void main()
{
    gl_Position = vec4(VertexPosition, 0.0, 1.0);
}
```

2. Use the following code as the tessellation control shader:

```
layout( vertices=4 ) out;

uniform int Outer;
uniform int Inner;
void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(Outer);
    gl_TessLevelOuter[1] = float(Outer);
    gl_TessLevelOuter[2] = float(Outer);
    gl_TessLevelOuter[3] = float(Outer);

    gl_TessLevelInner[0] = float(Inner);
    gl_TessLevelInner[1] = float(Inner);
}
```

3. Use the following code as the tessellation evaluation shader:

```
layout( quads, equal_spacing, ccw ) in;

uniform mat4 MVP;

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    vec4 p0 = gl_in[0].gl_Position;
    vec4 p1 = gl_in[1].gl_Position;
    vec4 p2 = gl_in[2].gl_Position;
    vec4 p3 = gl_in[3].gl_Position;

    // Linear interpolation
    gl_Position =
        p0 * (1-u) * (1-v) +
        p1 * u * (1-v) +
        p3 * v * (1-u) +
        p2 * u * v;

    // Transform to clip coordinates
    gl_Position = MVP * gl_Position;
}
```

4. Use the geometry shader from the recipe, *Drawing a wireframe on top of a shaded mesh*.

5. Use the following code as the fragment shader:

```
uniform float LineWidth;
uniform vec4 LineColor;
uniform vec4 QuadColor;

noperspective in vec3 EdgeDistance; // From geom. shader

layout ( location = 0 ) out vec4 FragColor;

float edgeMix()
{
    // ** insert code here to determine how much of the edge
    // color to include (see recipe "Drawing a wireframe on
    // top of a shaded mesh"). **
}

void main()
{
    float mixVal = edgeMix();

    FragColor = mix( QuadColor, LineColor, mixVal );
}
```

6. Within the render function of your main OpenGL program, define the number of vertices within a patch:

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
```

7. Render the patch as four 2D vertices in counter clockwise order.

How it works...

The vertex shader passes the position along to the TCS unchanged.

The TCS defines the number of vertices in the patch using the layout directive:

```
layout (vertices=4) out;
```

In the `main` function, it passes along the position of the vertex without modification, and sets the inner and outer tessellation levels. All four of the outer tessellation levels are set to the value of `Outer`, and both of the inner tessellation levels are set to `Inner`.

In the tessellation evaluation shader, we define the tessellation mode and other tessellation parameters with the input layout directive:

```
layout ( quads, equal_spacing, ccw ) in;
```

The parameter `quads` indicates that the tessellation primitive generator should tessellate the parameter space using quad tessellation as described some time back. The parameter `equal_spacing` says that the tessellation should be performed such that all subdivisions have equal length. The last parameter, `ccw`, indicates that the primitives should be generated with counter clockwise winding.

The `main` function in the TES starts by retrieving the parametric coordinates for this vertex by accessing the variable `gl_TessCoord`. Then we move on to read the positions of the four vertices in the patch from the `gl_in` array. We store them in temporary variables to be used in the interpolation calculation.

The built-in output variable `gl_Position` then gets the value of the interpolated point using the preceding equation. Finally, we convert the position into clip coordinates by multiplying by the model-view projection matrix.

Within the fragment shader, we give all fragments a color that is possibly mixed with a line color in order to highlight the edges.

See also

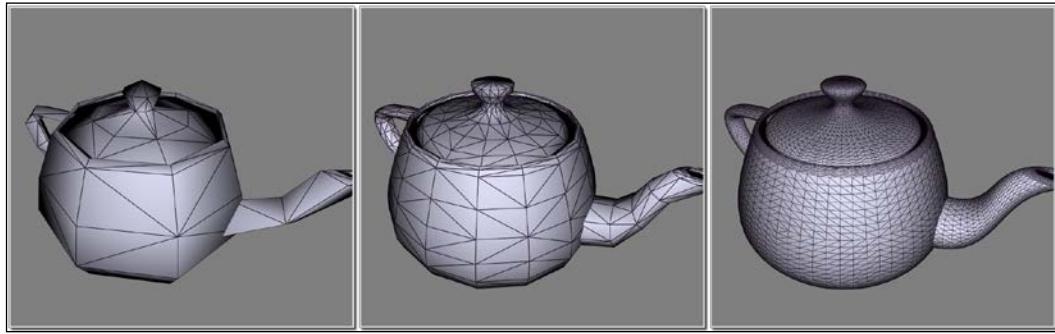
- ▶ The *Drawing a wireframe on top of a shaded mesh* recipe

Tessellating a 3D surface

As an example of tessellating a 3D surface, let's render (yet again) the "teapotahedron". It turns out that the teapot's data set is actually defined as a set of 4×4 patches of control points, suitable for cubic Bezier interpolation. Therefore, drawing the teapot really boils down to drawing a set of cubic Bezier surfaces.

Of course, this sounds like a perfect job for tessellation shaders! We'll render each patch of 16 vertices as a patch primitive, use quad tessellation to subdivide the parameter space, and implement the Bezier interpolation within the tessellation evaluation shader.

The following figure shows an example of the desired output. The left teapot is rendered with inner and outer tessellation level 2, the middle uses level 4 and the right-hand teapot uses tessellation level 16. The tessellation evaluation shader computes the Bezier surface interpolation.



First, let's take a look at how cubic Bezier surface interpolation works. If our surface is defined by a set of 16 control points (laid out in a 4×4 grid) P_{ij} , with i and j ranging from 0 to 3, then the parametric Bezier surface is given by the following equation:

$$P(u, v) = \sum_{i=0}^3 \sum_{j=0}^3 B_i^3(u) B_j^3(v) P_{ij}$$

The instances of **B** in the preceding equation are the cubic Bernstein polynomials (refer to the previous recipe, *Tessellating a 2D quad*).

We also need to compute the normal vector at each interpolated location. To do so, we have to compute the cross product of the partial derivatives of the preceding equation:

$$\mathbf{n}(u, v) = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

The partial derivatives of the Bezier surface boil down to the partial derivatives of the Bernstein polynomials:

$$\begin{aligned} \frac{\partial P}{\partial u} &= \sum_{i=0}^3 \sum_{j=0}^3 \frac{\partial B_i^3(u)}{\partial u} B_j^3(v) P_{ij} \\ \frac{\partial P}{\partial v} &= \sum_{i=0}^3 \sum_{j=0}^3 B_i^3(u) \frac{\partial B_j^3(v)}{\partial v} P_{ij} \end{aligned}$$

We'll compute the partials within the TES and compute the cross product to determine the normal to the surface at each tessellated vertex.

Getting ready

Set up your shaders with a vertex shader that simply passes the vertex position along without any modification (you can use the same vertex shader as was used in the *Tessellating a 2D quad* recipe). Create a fragment shader that implements whatever shading model you choose. The fragment shader should receive the input variables `TENormal` and `TEPosition`, which will be the normal and position in camera coordinates.

The uniform variable `TessLevel` should be given the value of the tessellation level desired. All of the inner and outer levels will be set to this value.

How to do it...

To create a shader program that creates Bezier patches from input patches of 16 control points, use the following steps:

1. Use the vertex shader from the *Tessellating a 2D quad* recipe.
2. Use the following code for the tessellation control shader:

```
layout( vertices=16 ) out;

uniform int TessLevel;

void main()
{
    // Pass along the vertex position unmodified
    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;

    gl_TessLevelOuter[0] = float(TessLevel);
    gl_TessLevelOuter[1] = float(TessLevel);
    gl_TessLevelOuter[2] = float(TessLevel);
    gl_TessLevelOuter[3] = float(TessLevel);

    gl_TessLevelInner[0] = float(TessLevel);
    gl_TessLevelInner[1] = float(TessLevel);
}
```

3. Use the following code for the tessellation evaluation shader:

```
layout( quads ) in;
out vec3 TENormal;    // Vertex normal in camera coords.
out vec4 TEPosition; // Vertex position in camera coords
```

```
uniform mat4 MVP;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;

void basisFunctions(out float[4] b, out float[4] db, float t)
{
    float t1 = (1.0 - t);
    float t12 = t1 * t1;

    // Bernstein polynomials
    b[0] = t12 * t1;
    b[1] = 3.0 * t12 * t;
    b[2] = 3.0 * t1 * t * t;
    b[3] = t * t * t;

    // Derivatives
    db[0] = -3.0 * t1 * t1;
    db[1] = -6.0 * t * t1 + 3.0 * t12;
    db[2] = -3.0 * t * t + 6.0 * t * t1;
    db[3] = 3.0 * t * t;
}

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;

    // The sixteen control points
    vec4 p00 = gl_in[0].gl_Position;
    vec4 p01 = gl_in[1].gl_Position;
    vec4 p02 = gl_in[2].gl_Position;
    vec4 p03 = gl_in[3].gl_Position;
    vec4 p10 = gl_in[4].gl_Position;
    vec4 p11 = gl_in[5].gl_Position;
    vec4 p12 = gl_in[6].gl_Position;
    vec4 p13 = gl_in[7].gl_Position;
    vec4 p20 = gl_in[8].gl_Position;
    vec4 p21 = gl_in[9].gl_Position;
    vec4 p22 = gl_in[10].gl_Position;
    vec4 p23 = gl_in[11].gl_Position;
    vec4 p30 = gl_in[12].gl_Position;
    vec4 p31 = gl_in[13].gl_Position;
    vec4 p32 = gl_in[14].gl_Position;
    vec4 p33 = gl_in[15].gl_Position;
    // Compute basis functions
    float bu[4], bv[4];    // Basis functions for u and v
    float dbu[4], dbv[4]; // Derivitives for u and v
```

```
basisFunctions(bu, dbu, u);
basisFunctions(bv, dbv, v);

// Bezier interpolation
TEPosition =
    p00*bu[0]*bv[0] + p01*bu[0]*bv[1] + p02*bu[0]*bv[2] +
    p03*bu[0]*bv[3] +
    p10*bu[1]*bv[0] + p11*bu[1]*bv[1] + p12*bu[1]*bv[2] +
    p13*bu[1]*bv[3] +
    p20*bu[2]*bv[0] + p21*bu[2]*bv[1] + p22*bu[2]*bv[2] +
    p23*bu[2]*bv[3] +
    p30*bu[3]*bv[0] + p31*bu[3]*bv[1] + p32*bu[3]*bv[2] +
    p33*bu[3]*bv[3];

// The partial derivatives
vec4 du =
    p00*dbu[0]*bv[0]+p01*dbu[0]*bv[1]+p02*dbu[0]*bv[2] +
    p03*dbu[0]*bv[3] +
    p10*dbu[1]*bv[0]+p11*dbu[1]*bv[1]+p12*dbu[1]*bv[2] +
    p13*dbu[1]*bv[3] +
    p20*dbu[2]*bv[0]+p21*dbu[2]*bv[1]+p22*dbu[2]*bv[2] +
    p23*dbu[2]*bv[3] +
    p30*dbu[3]*bv[0]+p31*dbu[3]*bv[1]+p32*dbu[3]*bv[2] +
    p33*dbu[3]*bv[3];

vec4 dv =
    p00*bu[0]*dbv[0]+p01*bu[0]*dbv[1]+p02*bu[0]*dbv[2] +
    p03*bu[0]*dbv[3] +
    p10*bu[1]*dbv[0]+p11*bu[1]*dbv[1]+p12*bu[1]*dbv[2] +
    p13*bu[1]*dbv[3] +
    p20*bu[2]*dbv[0]+p21*bu[2]*dbv[1]+p22*bu[2]*dbv[2] +
    p23*bu[2]*dbv[3] +
    p30*bu[3]*dbv[0]+p31*bu[3]*dbv[1]+p32*bu[3]*dbv[2] +
    p33*bu[3]*dbv[3];

// The normal is the cross product of the partials
vec3 n = normalize( cross(du.xyz, dv.xyz) );

// Transform to clip coordinates
gl_Position = MVP * TEPosition;

// Convert to camera coordinates
TEPosition = ModelViewMatrix * TEPosition;
TENormal = normalize(NormalMatrix * n);
}
```

4. Implement your favorite shading model within the fragment shader utilizing the output variables from the TES.
5. Render the Bezier control points as a 16-vertex patch primitive. Don't forget to set the number of vertices per patch within the OpenGL application:
`glPatchParameteri(GL_PATCH_VERTICES, 16);`

How it works...

The tessellation control shader starts by defining the number of vertices in the patch using the layout directive:

```
layout( vertices=16 ) out;
```

It then simply sets the tessellation levels to the value of `TessLevel`. It passes the vertex position along without any modification.

The tessellation evaluation shader starts by using a layout directive to indicate the type of tessellation to be used. As we are tessellating a 4 x 4 Bezier surface patch, quad tessellation makes the most sense.

The `basisFunctions` function evaluates the Bernstein polynomials and their derivatives for a given value of the parameter `t`. The results are returned in the output parameters `b` and `db`.

Within the `main` function, we start by assigning the tessellation coordinates to variables `u` and `v`, and reassigning all 16 of the patch vertices to variables with shorter names (to shorten the code that appears later).

We then call `basisFunctions` to compute the Bernstein polynomials and their derivatives at `u` and at `v`, storing the results in `bu`, `dbu`, `bv`, and `dbv`.

The next step is the evaluation of the sums from the preceding equations for the position (`TEPosition`), the partial derivative with respect to `u` (`du`), and the partial derivative with respect to `v` (`dv`).

We compute the normal vector as the cross product of `du` and `dv`.

Finally, we convert the position (`TEPosition`) to clip coordinates and assign the result to `gl_Position`. We also convert it to camera coordinates before it is passed along to the fragment shader.

The normal vector is converted to camera coordinates by multiplying with the `NormalMatrix`, and the result is normalized and passed along to the fragment shader via `TENormal`.

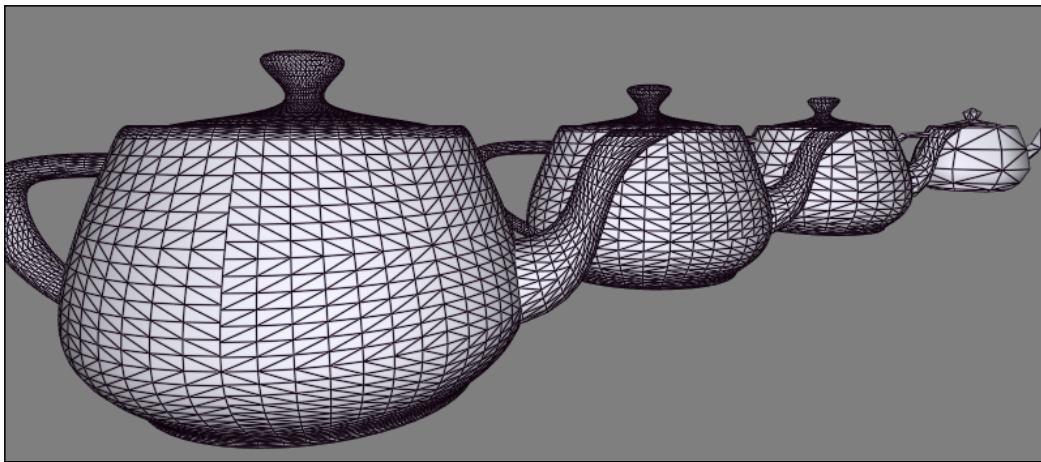
See also

- ▶ [The Tessellating a 2D quad recipe](#)

Tessellating based on depth

One of the greatest things about tessellation shaders is how easy it is to implement **level-of-detail (LOD)** algorithms. LOD is a general term in computer graphics that refers to the process of increasing/decreasing the complexity of an object's geometry with respect to the distance from the viewer (or other factors). As an object moves farther away from the camera, less geometric detail is needed to represent the shape because the overall size of the object becomes smaller. However, as the object moves closer to the camera, the object fills more and more of the screen, and more geometric detail is needed to maintain the desired appearance (smoothness or lack of other geometric artifacts).

The following figure shows a few teapots rendered with tessellation levels that depend on distance from the camera. Each teapot is rendered using exactly the same code on the OpenGL side. The TCS automatically varies the tessellation levels based on depth.



When tessellation shaders are used, the tessellation level is what determines the geometric complexity of the object. As the tessellation levels can be set within the tessellation control shader, it is a simple matter to vary the tessellation levels with respect to the distance from the camera.

In this example, we'll vary the tessellation levels linearly (with respect to distance) between a minimum level and a maximum level. We'll compute the "distance from the camera" as the absolute value of the z coordinate in camera coordinates, (of course, this is not the true distance, but should work fine for the purposes of this example). The tessellation level will then be computed based on that value. We'll also define two additional values (as uniform variables) `MinDepth` and `MaxDepth`. Objects that are closer to the camera than `MinDepth` get the maximum tessellation level, and any objects that are further from the camera than `MaxDepth` will get the minimum tessellation level. The tessellation level for objects in between will be linearly interpolated.

Getting ready

This program is nearly identical to the one in the *Tessellating a 3D surface* recipe. The only difference lies within the TCS. We'll remove the uniform variable `TessLevel`, and add a few new ones that are described as follows:

- ▶ `MinTessLevel`: This is the lowest desired tessellation level
- ▶ `MaxTessLevel`: This is the highest desired tessellation level
- ▶ `MinDepth`: This is the minimum "distance" from the camera, where the tessellation level is maximal
- ▶ `MaxDepth`: This is the maximum "distance" from the camera, where the tessellation level is at a minimum

Render your objects as 16-vertex patch primitives as indicated in the recipe, *Tessellating a 3D surface*.

How to do it...

To create a shader program that varies the tessellation level based on the depth, use the following steps:

1. Use the vertex shader and tessellation evaluation shader from the recipe, *Tessellating a 3D surface*.
2. Use the following code for the tessellation control shader:

```
layout( vertices=16 ) out;

uniform int MinTessLevel;
uniform int MaxTessLevel;
uniform float MaxDepth;
uniform float MinDepth;

uniform mat4 ModelViewMatrix;

void main()
{
    // Position in camera coordinates
    vec4 p = ModelViewMatrix *
        gl_in[gl_InvocationID].gl_Position;

    // "Distance" from camera scaled between 0 and 1
    float depth = clamp( (abs(p.z) - MinDepth) /
        (MaxDepth - MinDepth), 0.0, 1.0 );

    // Interpolate between min/max tess levels
```

```
float tessLevel =
    mix(MaxTessLevel, MinTessLevel, depth);

gl_TessLevelOuter[0] = float(tessLevel);
gl_TessLevelOuter[1] = float(tessLevel);
gl_TessLevelOuter[2] = float(tessLevel);
gl_TessLevelOuter[3] = float(tessLevel);

gl_TessLevelInner[0] = float(tessLevel);
gl_TessLevelInner[1] = float(tessLevel);

gl_out[gl_InvocationID].gl_Position =
    gl_in[gl_InvocationID].gl_Position;
}
```

3. As with the previous recipe, implement your favorite shading model within the fragment shader.

How it works...

The TCS takes the position and converts it to camera coordinates and stores the result in the variable `p`. The absolute value of the z coordinate is then scaled and clamped so that the result is between zero and one. If the z coordinate is equal to `MaxDepth`, the value of `depth` will be `1.0`, if it is equal to `MinDepth`, then `depth` will be `0.0`. If `z` is between `MinDepth` and `MaxDepth`, then `depth` will get a value between zero and one. If `z` is outside that range, it will be clamped to `0.0` or `1.0` by the `clamp` function.

The value of `depth` is then used to linearly interpolate between `MaxTessLevel` and `MinTessLevel` using the `mix` function. The result (`tessLevel`) is used to set the inner and outer tessellation levels.

There's more...

There is a somewhat subtle aspect to this example. Recall that the TCS is executed once for each output vertex in the patch. Therefore, assuming that we are rendering cubic Bezier surfaces, this TCS will be executed 16 times for each patch. Each time it is executed, the value of `depth` will be slightly different because it is evaluated based on the z coordinate of the vertex. You might be wondering, which of the 16 possible different tessellation levels will be the one that is used? It doesn't make sense for the tessellation level to be interpolated across the parameter space. What's going on?

The output arrays `gl_TessLevelInner` and `gl_TessLevelOuter` are per-patch output variables. This means that only a single value will be used per-patch, similar to the way that the flat qualifier works for fragment shader input variables. The OpenGL specification seems to indicate that any of the values from each of the invocations of the TCS could be the value that ends up being used.

See also

- ▶ DirectX 11 Terrain Tessellation at: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/TerrainTessellation_WhitePaper.pdf
- ▶ The *Tessellating a 3D surface* recipe

7

Shadows

In this chapter, we will cover:

- ▶ Rendering shadows with shadow maps
- ▶ Anti-aliasing shadow edges with PCF
- ▶ Creating soft shadow edges with random sampling
- ▶ Creating shadows using shadow volumes and the geometry shader

Introduction

Shadows add a great deal of realism to a scene. Without shadows, it can be easy to misjudge the relative location of objects, and the lighting can appear unrealistic, as light rays seem to pass right through objects.

Shadows are important visual cues for realistic scenes, but can be challenging to produce in an efficient manner in interactive applications. One of the most popular techniques for creating shadows in real-time graphics is the **shadow mapping** algorithm (also called depth shadows). In this chapter, we'll look at several recipes surrounding the shadow mapping algorithm. We'll start with the basic algorithm, and discuss it in detail in the first recipe. Then we'll look at a couple of techniques for improving the look of the shadows produced by the basic algorithm.

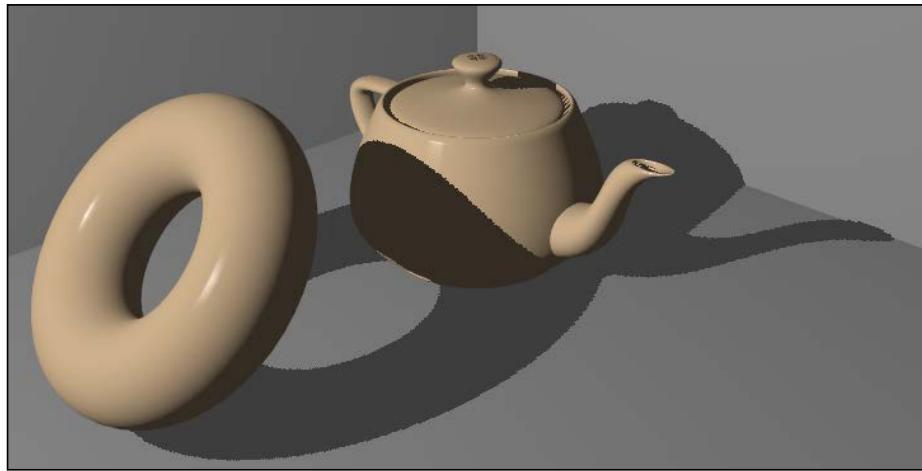
We'll also look at an alternative technique for shadows called shadow volumes. Shadow volumes produce near perfect hard-edged shadows, but are not well suited for creating shadows with soft edges.

Rendering shadows with shadow maps

One of the most common and popular techniques for producing shadows is called shadow mapping. In its basic form, the algorithm involves two passes. In the first pass, the scene is rendered from the point of view of the light source. The depth information from this pass is saved into a texture called the shadow map. This map will help provide information about the visibility of objects from the light's perspective. In other words, the shadow map stores the distance (actually the pseudo-depth) from the light to whatever the light can "see". Anything that is closer to the light than the corresponding depth stored in the map is lit; anything else must be in shadow.

In the second pass, the scene is rendered normally, but each fragment's depth (from the light's perspective) is first tested against the shadow map to determine whether or not the fragment is in shadow. The fragment is then shaded differently depending on the result of this test. If the fragment is in shadow, it is shaded with ambient lighting only; otherwise, it is shaded normally.

The following figure shows an example of shadows produced by the basic shadow mapping technique:



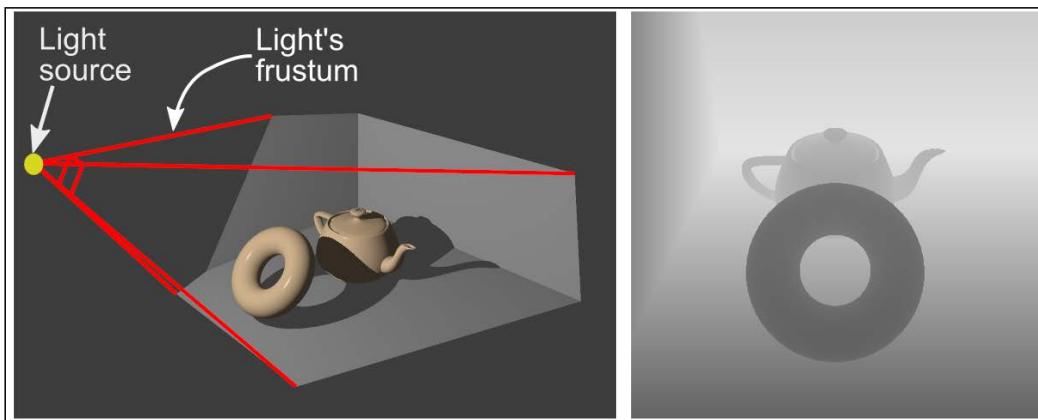
Let's look at each step of the algorithm in detail.

The first step is the creation of the shadow map. We set up our view matrix so that we are rendering the scene as if the camera is located at the position of the light source, and is oriented towards the shadow-casting objects. We set up a projection matrix such that the view frustum encloses all objects that may cast shadows as well as the area where the shadows will appear. We then render the scene normally and store the information from the depth buffer in a texture. This texture is called the shadow map (or simply depth map). We can think of it (roughly) as a set of distances from the light source to various surface locations.



Technically, these are depth values, not distances. A depth value is not a true distance (from the origin), but can be roughly treated as such for the purposes of depth testing.

The following figures represent an example of the basic shadow mapping setup. The left figure shows the light's position and its associated perspective frustum. The right-hand figure shows the corresponding shadow map. The grey scale intensities in the shadow map correspond to the depth values (darker is closer).



Once we have created the shadow map and stored the map to a texture, we render the scene again from the point of view of the camera. This time, we use a fragment shader that shades each fragment based on the result of a depth test with the shadow map. The position of the fragment is first converted into the coordinate system of the light source and projected using the light source's projection matrix. The result is then biased (in order to get valid texture coordinates) and tested against the shadow map. If the depth of the fragment is greater than the depth stored in the shadow map, then there must be some surface that is between the fragment and the light source. Therefore, the fragment is in shadow and is shaded using ambient lighting only. Otherwise, the fragment must have a clear view to the light source, and so it is shaded normally.

The key aspect here is the conversion of the fragment's 3D coordinates to the coordinates appropriate for a lookup into the shadow map. As the shadow map is just a 2D texture, we need coordinates that range from zero to one for points that lie within the light's frustum. The light's view matrix will transform points in world coordinates to points within the light's coordinate system. The light's projection matrix will transform points that are within the light's frustum to **homogeneous clip coordinates**.



These are called clip coordinates because the built-in clipping functionality takes place when the position is defined in these coordinates. Points within the perspective (or orthographic) frustum are transformed by the projection matrix to the (homogeneous) space that is contained within a cube centered at the origin, with each side of length two. This space is called the **canonical viewing volume**. The term "homogeneous" means that these coordinates should not necessarily be considered to be true Cartesian positions until they are divided by their fourth coordinate. For full details about homogeneous coordinates, refer to your favorite textbook on computer graphics.

The x and y components of the position in clip coordinates are roughly what we need to access the shadow map. The z coordinate contains the depth information that we can use to compare with the shadow map. However, before we can use these values we need to do two things. First, we need to bias them so that they range from zero to one (instead of -1 to 1), and second, we need to apply **perspective division** (more on this later).

To convert the value from clip coordinates to a range appropriate for use with a shadow map, we need the x, y, and z coordinates to range from zero to one (for points within the light's view frustum). The depth that is stored in an OpenGL depth buffer (and also our shadow map) is simply a fixed or floating-point value between zero and one (typically). A value of zero corresponds to the near plane of the perspective frustum, and a value of one corresponds to points on the far plane. Therefore, if we are to use our z coordinate to accurately compare with this depth buffer, we need to scale and translate it appropriately.



In clip coordinates (after perspective division) the z coordinate ranges from -1 to 1. It is the viewport transformation that (among other things) converts the depth to a range between zero and one. Incidentally, if so desired, we can configure the viewport transformation to use some other range for the depth values (say between 0 and 100) via the `glDepthRange` function.

Of course, the x and y components also need to be biased between zero and one because that is the appropriate range for texture access.

We can use the following "bias" matrix to alter our clip coordinates.

$$\mathbf{B} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix will scale and translate our coordinates such that the x, y, and z components range from 0 to 1 (after perspective division) for points within the light's frustum. Now, combining the bias matrix with the light's view (V_l) and projection (P_l) matrices, we have the following equation for converting positions in world coordinates (\mathbf{W}) to homogeneous positions that can be used for shadow map access (\mathbf{Q}).

$$\mathbf{Q} = \mathbf{B}\mathbf{P}_l\mathbf{V}_l\mathbf{W}$$

Finally, before we can use the value of \mathbf{Q} directly, we need to divide by the fourth (w) component. This step is sometimes called "perspective division". This converts the position from a homogeneous value to a true Cartesian position, and is always required when using a perspective projection matrix.

In the following equation, we'll define a shadow matrix (\mathbf{S}) that also includes the model matrix (\mathbf{M}), so that we can convert directly from the modeling coordinates (\mathbf{C}). (Note that $\mathbf{W} = \mathbf{MC}$, because the model matrix takes modeling coordinates to world coordinates.)

$$\mathbf{Q} = \mathbf{SC}$$

Here, \mathbf{S} is the shadow matrix, the product of the model matrix with all of the preceding matrices.

$$\mathbf{S} = \mathbf{BP}_l\mathbf{V}_l\mathbf{M}$$

In this recipe, in order to keep things simple and clear, we'll cover only the basic shadow mapping algorithm, without any of the usual improvements. We'll build upon this basic algorithm in the following recipes. Before we get into the code, we should note that the results will likely be less than satisfying. This is because the basic shadow mapping algorithm suffers from significant aliasing artifacts. Nevertheless, it is still an effective technique when combined with one of many techniques for anti-aliasing. We'll look at some of those techniques in the recipes that follow.

Getting ready

The position should be supplied in vertex attribute zero and the normal in vertex attribute one. Uniform variables for the ADS shading model should be declared and assigned, as well as uniforms for the standard transformation matrices. The `ShadowMatrix` variable should be set to the matrix for converting from modeling coordinates to shadow map coordinates (\mathbf{S} in the preceding equation).

The uniform variable `ShadowMap` is a handle to the shadow map texture, and should be assigned to texture unit zero.

How to do it...

To create an OpenGL application that creates shadows using the shadow mapping technique, use the following steps. We'll start by setting up a **Framebuffer Object (FBO)** to contain the shadow map texture, and then move on to the required shader code:

1. In the main OpenGL program, set up a FBO with a depth buffer only. Declare a GLuint variable named `shadowFBO` to store the handle to this framebuffer. The depth buffer storage should be a texture object. You can use something similar to the following code to accomplish this:

```
GLfloat border[]={1.0f,0.0f,0.0f,0.0f};

//The shadowmap texture
GLuint depthTex;
 glGenTextures(1,&depthTex);
 glBindTexture(GL_TEXTURE_2D,depthTex);
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT24,
                shadowMapWidth, shadowMapHeight);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                 GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                 GL_NEAREST);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
                 GL_CLAMP_TO_BORDER);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,
                 GL_CLAMP_TO_BORDER);
 glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_BORDER_COLOR,
                  border);

 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_MODE,
                 GL_COMPARE_REF_TO_TEXTURE);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_COMPARE_FUNC,
                 GL_LESS);

//Assign the shadow map to texture unit 0
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D,depthTex);

//Create and set up the FBO
 glGenFramebuffers(1,&shadowFBO);
 glBindFramebuffer(GL_FRAMEBUFFER,shadowFBO);
 glFramebufferTexture2D(GL_FRAMEBUFFER,GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D,depthTex,0);
 GLenum drawBuffers[]={GL_NONE};
 glDrawBuffers(1,drawBuffers);
 // Revert to the default framebuffer for now
 glBindFramebuffer(GL_FRAMEBUFFER,0);
```

2. Use the following code for the vertex shader:

```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexNormal;

out vec3 Normal;
out vec3 Position;

// Coordinate to be used for shadow map lookup
out vec4 ShadowCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;
uniform mat4 ShadowMatrix;

void main()
{
    Position = (ModelViewMatrix *
                vec4(VertexPosition,1.0)).xyz;
    Normal = normalize( NormalMatrix * VertexNormal );

    // ShadowMatrix converts from modeling coordinates
    // to shadow map coordinates.
    ShadowCoord =ShadowMatrix * vec4(VertexPosition,1.0);

    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

3. Use the following code for the fragment shader:

```
// Declare any uniforms needed for your shading model
uniform sampler2DShadow ShadowMap;

in vec3 Position;
in vec3 Normal;
in vec4 ShadowCoord;

layout (location = 0) out vec4 FragColor;

vec3 diffAndSpec()
{
    // Compute only the diffuse and specular components of
    // the shading model.
}
```

Shadows

```
subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;

subroutine (RenderPassType)
void shadeWithShadow()
{
    vec3 ambient = ...; // compute ambient component here
    vec3 diffSpec = diffAndSpec();

    // Do the shadow-map look-up
    float shadow = textureProj(ShadowMap, ShadowCoord);

    // If the fragment is in shadow, use ambient light only.
    FragColor = vec4(diffSpec * shadow + ambient, 1.0);
}

subroutine (RenderPassType)
void recordDepth()
{
    // Do nothing, depth will be written automatically
}

void main() {
    // This will call either shadeWithShadow or recordDepth
    RenderPass();
}
```

Within the main OpenGL program, perform the following steps when rendering:

Pass 1

1. Set the viewport, view, and projection matrices to those that are appropriate for the light source.
2. Bind to the framebuffer containing the shadow map (shadowFBO).
3. Clear the depth buffer.
4. Select the subroutine function `recordDepth`.
5. Enable front-face culling.
6. Draw the scene.

Pass 2

1. Select the viewport, view, and projection matrices appropriate for the scene.
2. Bind to the default framebuffer.
3. Disable culling (or switch to back-face culling).
4. Select the subroutine function `shadeWithShadow`.
5. Draw the scene.

How it works...

The first block of the preceding code demonstrates how to create a FBO for our shadow map texture. The FBO contains only a single texture connected to its depth buffer attachment. The first few lines of code create the shadow map texture. The texture is allocated using the `glTexStorage2D` function with an internal format of `GL_DEPTH_COMPONENT24`.

We use `GL_NEAREST` for `GL_TEXTURE_MAG_FILTER` and `GL_TEXTURE_MIN_FILTER` here, although `GL_LINEAR` could also be used, and might provide slightly better-looking results. We use `GL_NEAREST` here so that we can see the aliasing artifacts clearly, and the performance will be slightly better.

Next, the `GL_TEXTURE_WRAP_*` modes are set to `GL_CLAMP_TO_BORDER`. When a fragment is found to lie completely outside of the shadow map (outside of the light's frustum), then the texture coordinates for that fragment will be greater than one or less than zero. When that happens, we need to make sure that those points are not treated as being in shadow. When `GL_CLAMP_TO_BORDER` is used, the value that is returned from a texture lookup (for coordinates outside the $0..1$ range) will be the border value. The default border value is $(0, 0, 0, 0)$. When the texture contains depth components, the first component is treated as the depth value. A value of zero will not work for us here because a depth of zero corresponds to points on the near plane. Therefore all points outside of the light's frustum will be treated as being in shadow! Instead, we set the border color to $(1, 0, 0, 0)$ using the `glTexParameterfv` function, which corresponds to the maximum possible depth.

The next two calls to `glTexParameteri` affect settings that are specific to depth textures. The first call sets `GL_TEXTURE_COMPARE_MODE` to `GL_COMPARE_REF_TO_TEXTURE`. When this setting is enabled, the result of a texture access is the result of a comparison, rather than a color value retrieved from the texture. The third component of the texture coordinate (the `p` component) is compared against the value in the texture at location (s, t) . The result of the comparison is returned as a single floating-point value. The comparison function that is used is determined by the value of `GL_TEXTURE_COMPARE_FUNC`, which is set on the next line. In this case, we set it to `GL_LESS`, which means that the result will be 1.0 if the `p` value of the texture coordinate is less than the value stored at (s, t) . (Other options include `GL_EQUAL`, `GL_ALWAYS`, `GL_GEQUAL`, and so on.)

The next few lines create and set up the FBO. The shadow map texture is attached to the FBO as the depth attachment with the `glFramebufferTexture2D` function. For more details about FBOs, check out the *Rendering to a texture* recipe in Chapter 4, *Using Textures*.

The vertex shader is fairly simple. It converts the vertex position and normal to camera coordinates and passes them along to the fragment shader via the output variables `Position` and `Normal`. The vertex position is also converted into shadow map coordinates using `ShadowMatrix`. This is the matrix `S` that we referred to in the previous section. It converts a position from modeling coordinates to shadow coordinates. The result is sent to the fragment shader via the output variable `ShadowCoord`.

Shadows

As usual, the position is also converted to clip coordinates and assigned to the built-in output variable `gl_Position`.

In the fragment shader, we provide different functionality for each pass. In the main function, we call `RenderPass`, which is a subroutine uniform that will call either `recordDepth` or `shadeWithShadow`. For the first pass (shadow map generation), the subroutine function `recordDepth` is executed. This function does nothing at all! This is because we only need to write the depth to the depth buffer. OpenGL will do this automatically (assuming that `gl_Position` was set correctly by the vertex shader), so there is nothing for the fragment shader to do.

During the second pass, the `shadeWithShadow` function is executed. We compute the ambient component of the shading model and store the result in the `ambient` variable. We then compute the diffuse and specular components and store those in the `diffuseAndSpec` variable.

The next step is the key to the shadow mapping algorithm. We use the built-in texture access function `textureProj`, to access the shadow map texture `ShadowMap`. Before using the texture coordinate to access the texture, the `textureProj` function will divide the first three components of the texture coordinate by the fourth component. Remember that this is exactly what is needed to convert the homogeneous position (`ShadowCoord`) to a true Cartesian position.

After this perspective division, the `textureProj` function will use the result to access the texture. As this texture's sampler type is `sampler2DShadow`, it is treated as texture containing depth values, and rather than returning a value from the texture, it returns the result of a comparison. The first two components of `ShadowCoord` are used to access a depth value within the texture. That value is then compared against the value of the third component of `ShadowCoord`. When `GL_NEAREST` is the interpolation mode (as it is in our case) the result will be `1.0` or `0.0`. As we set the comparison function to `GL_LESS`, this will return `1.0`, if the value of the third component of `ShadowCoord` is less than the value within the depth texture at the sampled location. This result is then stored in the variable `shadow`. Finally, we assign a value to the output variable `FragColor`. The result of the shadow map comparison (`shadow`) is multiplied by the diffuse and specular components, and the result is added to the ambient component. If `shadow` is `0.0`, that means that the comparison failed, meaning that there is something between the fragment and the light source. Therefore, the fragment is only shaded with ambient light. Otherwise, `shadow` is `1.0`, and the fragment is shaded with all three shading components.

When rendering the shadow map, note that we culled the front faces. This is to avoid the z-fighting that can occur when front faces are included in the shadow map. Note that this only works if our mesh is completely closed. If back faces are exposed, you may need to use another technique (that uses `glPolygonOffset`) to avoid this. I'll talk a bit more about this in the next section.

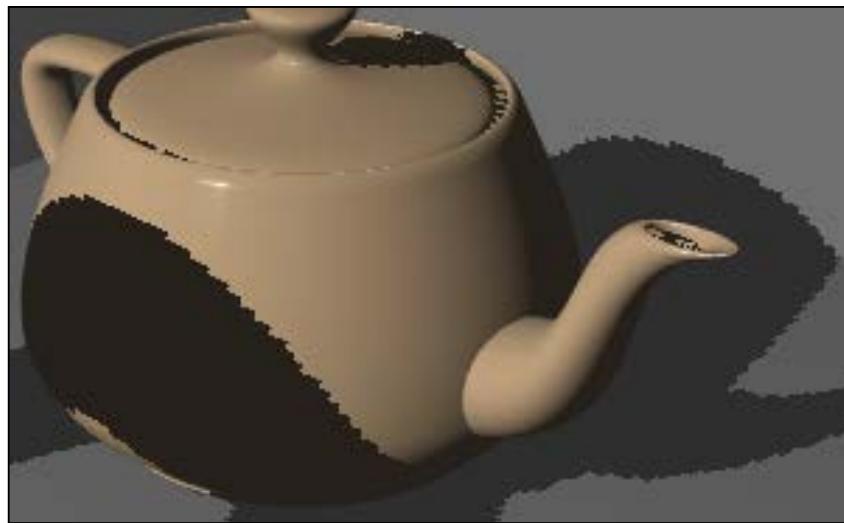
There's more...

There's a number of challenging issues with the shadow mapping technique. Let's look at just a few of the most immediate ones.

Aliasing

As mentioned earlier, this algorithm often suffers from severe aliasing artifacts at the shadow's edges. This is due to the fact that the shadow map is essentially projected onto the scene when the depth comparison is made. If the projection causes the map to be magnified, aliasing artifacts appear.

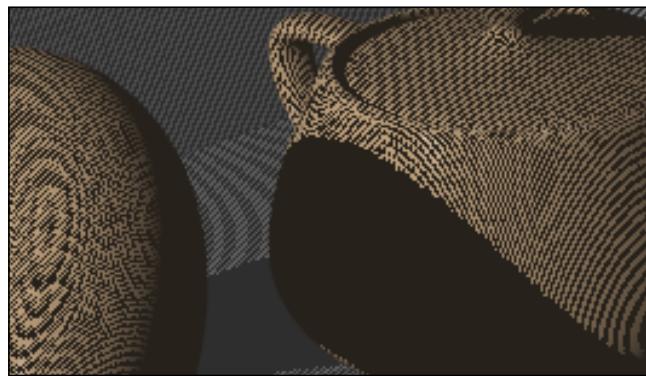
The following figure shows the aliasing of the shadow's edges:



The easiest solution is to simply increase the size of the shadow map. However, that may not be possible due to memory, CPU speed, or other constraints. There is a large number of techniques for improving the quality of the shadows produced by the shadow mapping algorithm such as resolution-matched shadow maps, cascaded shadow maps, variance shadow maps, perspective shadow maps and many others. In the following recipes, we'll look at some ways to help soften and anti-alias the edges of the shadows.

Rendering back faces only for the shadow map

When creating the shadow map, we only rendered back faces. This is because of the fact that if we were to render front faces, points on certain faces would have nearly the same depth as the shadow map's depth, which can cause fluctuations between light and shadow across faces that should be completely lit. The following figure shows an example of this effect:



Since the majority of faces that cause this issue are those that are facing the light source, we avoid much of the problem by only rendering back faces during the shadow map pass. This of course will only work correctly if your meshes are completely closed. If that is not the case, `glPolygonOffset` can be used to help the situation by offsetting the depth of the geometry from that in the shadow map. In fact, even when back faces are only rendered when generating the shadow map, similar artifacts can appear on faces that are facing away from the light (back faces in the shadow map, but front from the camera's perspective). Therefore, it is quite often the case that a combination of front-face culling and `glPolygonOffset` is used when generating the shadow map.

See also

- ▶ The *Rendering to a texture* recipe in Chapter 4, *Using Textures*
- ▶ The *Anti-aliasing shadow edges with PCF* recipe
- ▶ The *Creating soft shadow edges with random sampling* recipe

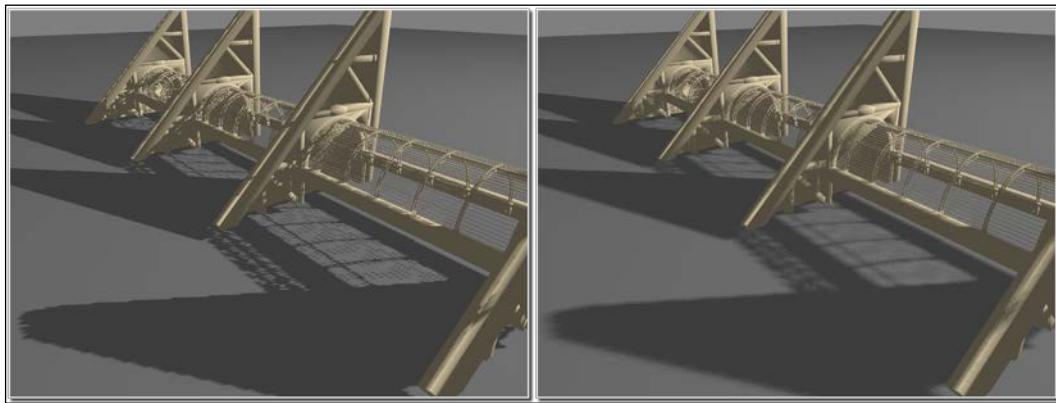
Anti-aliasing shadow edges with PCF

One of the simplest and most common techniques for dealing with the aliasing of shadow edges is called **percentage-closer filtering (PCF)**. The name comes from the concept of sampling the area around the fragment and determining the percentage of the area that is closer to the light source (in shadow). The percentage is then used to scale the amount of (diffuse and specular) shading that the fragment receives. The overall effect is a blurring of the shadow's edges.

The basic technique was first published by Reeves et al in a 1987 paper (*SIGGRAPH Proceedings, Volume 21, Number 4, July 1987*). The concept involved transforming the fragment's extents into shadow space, sampling several locations within that region, and computing the percent that is closer than the depth of the fragment. The result is then used to attenuate the shading. If the size of this filter region is increased, it can have the effect of blurring the shadow's edges.

A common variant of the PCF algorithm involves just sampling a constant number of nearby texels within the shadow map. The percent of those texels that are closer to the light is used to attenuate the shading. This has the effect of blurring the shadow's edges. While the result may not be physically accurate, the result is not objectionable to the eye.

The following figures show shadows rendered with PCF (right) and without PCF (left). Note that the shadows in the right-hand image have fuzzier edges and the aliasing is less visible.



In this recipe, we'll use the latter technique, and sample a constant number of texels around the fragment's position in the shadow map. We'll calculate an average of the resulting comparisons and use that result to scale the diffuse and specular components.

We'll make use of OpenGL's built-in support for PCF, by using linear filtering on the depth texture. When linear filtering is used with this kind of texture, the hardware can automatically sample four nearby texels (execute four depth comparisons) and average the results (the details of this are implementation dependent). Therefore, when linear filtering is enabled, the result of the `textureProj` function can be somewhere between 0.0 and 1.0.

We'll also make use of the built-in functions for texture accesses with offsets. OpenGL provides the texture access function `textureProjOffset`, which has a third parameter (the offset) that is added to the texel coordinates before the lookup/comparison.

Getting ready

Start with the shaders and FBO presented in the previous recipe, *Rendering shadows with shadow maps*. We'll just make a few minor changes to the code presented there.

How to do it...

To add the PCF technique to the shadow mapping algorithm, use the following steps:

1. When setting up the FBO for the shadow map, make sure to use linear filtering on the depth texture. Replace the corresponding lines with the following code:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  
                GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  
                GL_LINEAR);
```

2. Use the following code for the `shadeWithShadow` function within the fragment shader:

```
subroutine (RenderPassType)  
void shadeWithShadow()  
{  
    vec3 ambient = vec3(0.2);  
    vec3 diffSpec = diffAndSpec();  
  
    // The sum of the comparisons with nearby texels  
    float sum = 0;  
  
    // Sum contributions from texels around ShadowCoord  
    sum += textureProjOffset(ShadowMap, ShadowCoord,  
                            ivec2(-1,-1));  
    sum += textureProjOffset(ShadowMap, ShadowCoord,  
                            ivec2(-1,1));  
    sum += textureProjOffset(ShadowMap, ShadowCoord,  
                            ivec2(1,1));  
    sum += textureProjOffset(ShadowMap, ShadowCoord,  
                            ivec2(1,-1));  
    float shadow = sum * 0.25;  
  
    FragColor = vec4(ambient + diffSpec * shadow,1.0);  
}
```

How it works...

The first step enables linear filtering on the shadow map texture. When this is enabled, the OpenGL driver can repeat the depth comparison on the four nearby texels within the texture. The results of the four comparisons will be averaged and returned.

Within the fragment shader, we use the `textureProjOffset` function to sample the four texels (diagonally) surrounding the texel nearest to `ShadowCoord`. The third argument is the offset. It is added to the texel's coordinates (not the texture coordinates) before the lookup takes place.

As linear filtering is enabled, each lookup will sample an additional four texels, for a total of 16 texels. The results are then averaged together and stored within the variable `shadow`.

As before, the value of `shadow` is used to attenuate the diffuse and specular components of the lighting model.

There's more...

An excellent survey of the PCF technique was written by *Fabio Pellacini* of Pixar, and can be found in *Chapter 11, Shadow Map Anti-aliasing* of *GPU Gems*, edited by *Randima Fernando*, Addison-Wesley Professional, 2004. If more details are desired, I highly recommend reading this short, but informative, chapter.

Because of its simplicity and efficiency, the PCF technique is an extremely common method for anti-aliasing the edges of shadows produced by shadow mapping. Since it has the effect of blurring the edges, it can also be used to simulate soft shadows. However, the number of samples must be increased with the size of the blurred edge (the penumbra) to avoid certain artifacts. This can, of course, be a computational roadblock. In the next recipe, we'll look at a technique for producing soft shadows by randomly sampling a larger region.



The penumbra is the region of a shadow where only a portion of the light source is obscured.



See also

- ▶ The *Rendering shadows with shadow maps* recipe

Creating soft shadow edges with random sampling

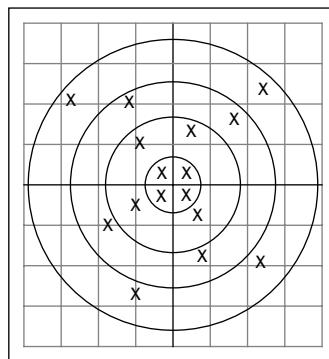
The basic shadow mapping algorithm combined with PCF can produce shadows with soft edges. However, if we desire blurred edges that are substantially wide (to approximate true soft shadows) then a large number of samples is required. Additionally, there is a good deal of wasted effort when shading fragments that are in the center of large shadows, or completely outside of the shadow. For those fragments, all of the nearby shadow map texels will evaluate to the same value. Therefore, the work of accessing and averaging those texels is essentially a wasted effort.

The technique presented in this recipe is based on a chapter published in *GPU Gems 2*, edited by Matt Pharr and Randima Fernando, Addison-Wesley Professional, 2005. (Chapter 17 by Yury Uralsky). It provides an approach that can address both of the preceding issues to create shadows with soft edges of various widths, while avoiding unnecessary texture accesses in areas inside and outside of the shadow.

The basic idea is as follows:

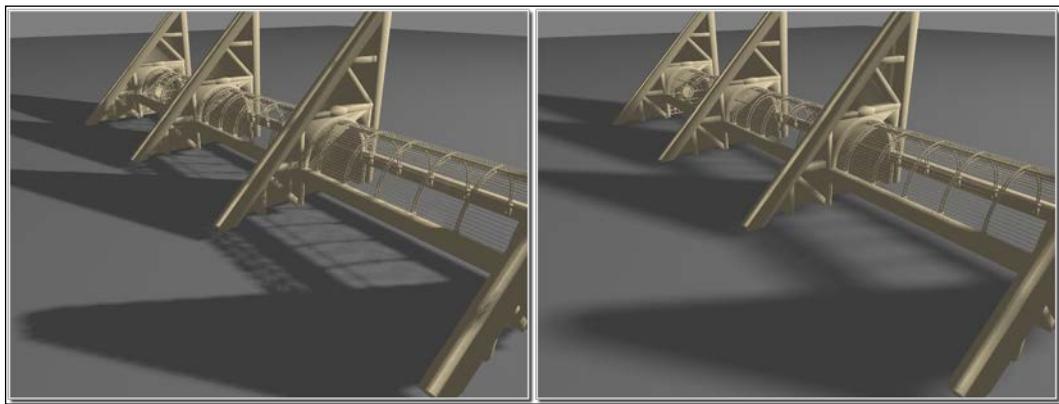
- ▶ Instead of sampling texels around the fragment's position (in shadow map space) using a constant set of offsets, we use a random, circular pattern of offsets
- ▶ In addition, we sample only the outer edges of the circle first in order to determine whether or not the fragment is in an area that is completely inside or outside of the shadow

The following figure is a visualization of a possible set of shadow map samples. The center of the cross-hairs is the fragment's location in the shadow map, and each **x** is a sample. The samples are distributed randomly within a circular grid around the fragment's location (one sample per grid cell).



Additionally, we vary the sample locations through a set of precomputed sample patterns. We compute random sample offsets and store them in a texture prior to rendering. Then, in the fragment shader, the samples are determined by first accessing the offset texture to grab a set of offsets and use them to vary the fragment's position in the shadow map. The results are then averaged together in a similar manner to the basic PCF algorithm.

The following figures show the difference between shadows using the PCF algorithm (left), and the random sampling technique described in this recipe (right).



We'll store the offsets in a three-dimensional texture ($n \times n \times d$). The first two dimensions are of arbitrary size, and the third dimension contains the offsets. Each (s,t) location contains a list (size d) of random offsets packed into an RGBA color. Each RGBA color in the texture contains two 2D offsets. The R and G channels contain the first offset, and the B and A channels contain the second. Therefore, each (s,t) location contains a total of $2*d$ offsets. For example, location $(1, 1, 3)$ contains the sixth and seventh offset at location $(1,1)$. The entire set of values at a given (s,t) comprise a full set of offsets.

We'll rotate through the texture based on the fragment's screen coordinates. The location within the offset texture will be determined by taking the remainder of the screen coordinates divided by the texture's size. For example, if the fragment's coordinates are $(10.0, 10.0)$ and the texture's size is $(4,4)$, then we use the set of offsets located in the offset texture at location $(2,2)$.

Getting ready

Start with the code presented in the *Rendering shadows with shadow maps* recipe.

There are three additional uniforms that need to be set. They are as follows:

- ▶ **OffsetTexSize**: This gives the width, height, and depth of the offset texture. Note that the depth is same as the number of samples per fragment divided by two.
- ▶ **OffsetTex**: This is a handle to the texture unit containing the offset texture.

- Radius: This is the blur radius in pixels divided by the size of the shadow map texture (assuming a square shadow map). This could be considered as the softness of the shadow.

How to do it...

To modify the shadow mapping algorithm and to use this random sampling technique, use the following steps. We'll build the offset texture within the main OpenGL program, and make use of it within the fragment shader:

1. Use the following code within the main OpenGL program to create the offset texture. This only needs to be executed once during the program initialization:

```
void buildOffsetTex(int size, int samplesU, int samplesV)
{
    int samples = samplesU * samplesV;
    int bufSize = size * size * samples * 2;
    float *data = new float[bufSize];

    for( int i = 0; i < size; i++ ) {
        for( int j = 0; j < size; j++ ) {
            for( int k = 0; k < samples; k += 2 ) {
                int x1,y1,x2,y2;
                x1 = k % (samplesU);
                y1 = (samples - 1 - k) / samplesU;
                x2 = (k+1) % samplesU;
                y2 = (samples - 1 - k - 1) / samplesU;

                vec4 v;
                // Center on grid and jitter
                v.x = (x1 + 0.5f) + jitter();
                v.y = (y1 + 0.5f) + jitter();
                v.z = (x2 + 0.5f) + jitter();
                v.w = (y2 + 0.5f) + jitter();
                // Scale between 0 and 1
                v.x /= samplesU;
                v.y /= samplesV;
                v.z /= samplesU;
                v.w /= samplesV;
                // Warp to disk
                int cell = ((k/2) * size * size + j *
                           size + i) * 4;
                data[cell+0] = sqrtf(v.y) * cosf(TWOPI*v.x);
                data[cell+1] = sqrtf(v.y) * sinf(TWOPI*v.x);
                data[cell+2] = sqrtf(v.w) * cosf(TWOPI*v.z);
                data[cell+3] = sqrtf(v.w) * sinf(TWOPI*v.z);
            }
        }
    }
}
```

```

        }
    }

    glActiveTexture(GL_TEXTURE1);
    GLuint texID;
    glGenTextures(1, &texID);

    glBindTexture(GL_TEXTURE_3D, texID);
    glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA32F, size, size,
                   samples/2);
    glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, size, size,
                   samples/2, GL_RGBA, GL_FLOAT, data);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);

    delete [] data;
}

// Return random float between -0.5 and 0.5
float jitter() {
    return ((float)rand() / RAND_MAX) - 0.5f;
}

```

2. Add the following uniform variables to the fragment shader:

```

uniform sampler3D OffsetTex;
uniform vec3 OffsetTexSize; // (width, height, depth)
uniform float Radius;

```

3. Use the following code for the shadeWithShadow function in the fragment shader:

```

subroutine (RenderPassType)
void shadeWithShadow()
{
    vec3 ambient = vec3(0.2);
    vec3 diffSpec = diffAndSpec();

    ivec3 offsetCoord;
    offsetCoord.xy = ivec2( mod( gl_FragCoord.xy,
                                OffsetTexSize.xy ) );

    float sum = 0.0;
    int samplesDiv2 = int(OffsetTexSize.z);
    vec4 sc = ShadowCoord;

    for( int i = 0 ; i < 4; i++ ) {
        offsetCoord.z = i;

```

```

vec4 offsets = texelFetch(OffsetTex, offsetCoord, 0) *
    Radius * ShadowCoord.w;

sc.xy = ShadowCoord.xy + offsets.xy;
sum += textureProj(ShadowMap, sc);
sc.xy = ShadowCoord.xy + offsets.zw;
sum += textureProj(ShadowMap, sc);
}
float shadow = sum / 8.0;

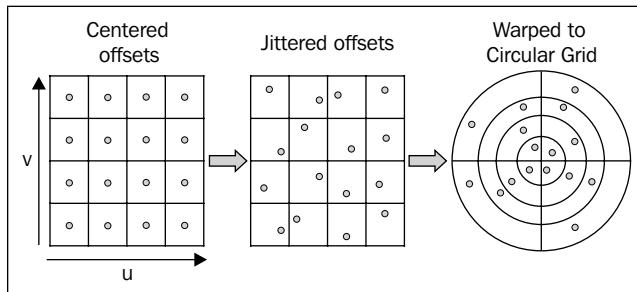
if( shadow != 1.0 && shadow != 0.0 ) {
    for( int i = 4; i < samplesDiv2; i++ ) {
        offsetCoord.z = i;
        vec4 offsets =
            texelFetch(OffsetTex, offsetCoord, 0) *
            Radius * ShadowCoord.w;

        sc.xy = ShadowCoord.xy + offsets.xy;
        sum += textureProj(ShadowMap, sc);
        sc.xy = ShadowCoord.xy + offsets.zw;
        sum += textureProj(ShadowMap, sc);
    }
    shadow = sum / float(samplesDiv2 * 2.0);
}
FragColor = vec4(diffSpec * shadow + ambient, 1.0);
}

```

How it works...

The `buildOffsetTex` function creates our three-dimensional texture of random offsets. The first parameter, `texSize`, defines the width and height of the texture. To create the preceding images, I used a value of 8. The second and third parameters, `samplesU` and `samplesV`, define the number of samples in the `u` and `v` directions. I used a value of 4 and 8, respectively, for a total of 32 samples. The `u` and `v` directions are arbitrary axes that are used to define a grid of offsets. To help understand this, take a look at the following figure:



The offsets are initially defined to be centered on a grid of size `samplesU x samplesV` (4×4 in the preceding figure). The coordinates of the offsets are scaled such that the entire grid fits in the unit cube (side length 1) with the origin in the lower left corner. Then each sample is randomly jittered from its position to a random location inside the grid cell. Finally, the jittered offsets are warped such that they surround the origin and lie within the circular grid shown on the right.

The last step can be accomplished by using the `v` coordinate as the distance from the origin and the `u` coordinate as the angle scaled from 0 to 360. The following equations should do the trick:

$$w_x = \sqrt{v} \cos(2\pi u)$$
$$w_y = \sqrt{v} \sin(2\pi u)$$

Here, `w` is the warped coordinate. What we are left with is a set of offsets around the origin that are a maximum distance of 1.0 from the origin. Additionally, we generate the data such that the first samples are the ones around the outer edge of the circle, moving inside towards the center. This will help us avoid taking too many samples when we are working completely inside or outside of the shadow.

Of course, we also pack the samples in such a way that a single texel contains two samples. This is not strictly necessary, but is done to conserve memory space. However, it does make the code a bit more complex.

Within the fragment shader, we start by computing the ambient component of the shading model separately from the diffuse and specular components. We access the offset texture at a location based on the fragment's screen coordinates (`gl_FragCoord`). We do so by taking the modulus of the fragment's position and the size of the offset texture. The result is stored in the first two components of `offsetCoord`. This will give us a different set of offsets for each nearby pixel. The third components of `offsetCoord` will be used to access a pair of samples. The number of samples is the depth of the texture divided by two. This is stored in `samplesDiv2`. We access the sample using the `texelFetch` function. This function allows us to access a texel using the integer texel coordinates rather than the usual normalized texture coordinates in the range 0...1.

The offset is retrieved and multiplied by `Radius` and the `w` component of `ShadowCoord`. Multiplying by `Radius` simply scales the offsets so that they range from 0.0 to `Radius`. We multiply by the `w` component because `ShadowCoord` is still a homogeneous coordinate, and our goal is to use offsets to translate the `ShadowCoord`. In order to do so properly, we need to multiply the offset by the `w` component. Another way of thinking of this is that the `w` component will be cancelled when perspective division takes place.

Next, we use offsets to translate `ShadowCoord` and access the shadow map to do the depth comparison using `textureProj`. We do so for each of the two samples stored in the texel, once for the first two components of offsets and again for the last two. The result is added to `sum`.

The first loop repeats this for the first eight samples. If the first eight samples are all 0.0 or 1.0, then we assume that all of the samples will be the same (the sample area is completely in or out of the shadow). In that case, we skip the evaluation of the rest of the samples. Otherwise, we evaluate the following samples and compute the overall average.

Finally the resulting average (shadow) is used to attenuate the diffuse and specular components of the lighting model.

There's more...

The use of a small texture containing a set of random offsets helps to blur the edges of the shadow better than what we might achieve with the standard PCF technique that uses a constant set of offsets. However, artifacts can still appear as repeated patterns within the shadow edges because the texture is finite and offsets are repeated every few pixels. We could improve this by also using a random rotation of the offsets within the fragment shader, or simply compute the offsets randomly within the shader.

It should also be noted that this blurring of the edges may not be desired for all shadow edges. For example, edges that are directly adjacent to the occluder, that is, creating the shadow, should not be blurred. These may not always be visible, but can become so in certain situations, such as when the occluder is a narrow object. The effect is to make the object appear as if it is hovering above the surface. Unfortunately, there isn't an easy fix for this one.

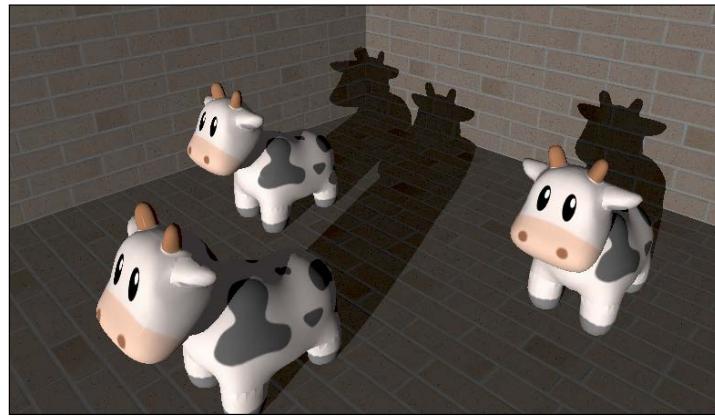
See also

- ▶ The *Rendering shadows with shadow maps* recipe

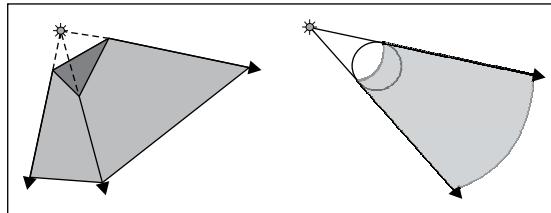
Creating shadows using shadow volumes and the geometry shader

As we discovered in the previous recipes, one of the main problems with shadow maps is aliasing. The problem essentially boils down to the fact that we are sampling the shadow map(s) at a different frequency (resolution) than we are using when rendering the scene. To minimize the aliasing we can blur the shadow edges (as in the previous recipes), or try to sample the shadow map at a frequency that is closer to the corresponding resolution in projected screen space. There are many techniques that help with the latter; for more details, I recommend the book *Real-Time Shadows*.

An alternate technique for shadow generation is called *shadow volumes*. The shadow volume method completely avoids the aliasing problem that plagues shadow maps. With shadow volumes, you get pixel-perfect hard shadows, without the aliasing artifacts of shadow maps. The following figure shows a scene with shadows that are produced using the shadow volume technique.

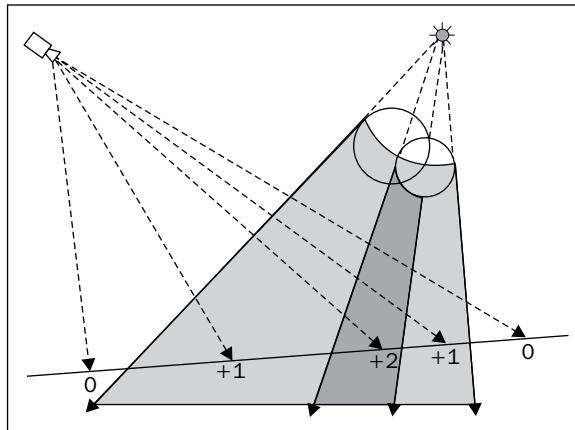


The shadow volume technique works by making use of the stencil buffer to mask out areas that are in shadow. We do this by drawing the boundaries of the actual shadow volumes (more on this below). A shadow volume is the region of space where the light source is occluded by an object. For example, the following figures show a representation of the shadow volumes of a triangle (left) and a sphere (right).



The boundaries of a shadow volume are made up of quads formed by extending the edges of the object away from the light source. For a single triangle, the boundaries would consist of three quads, extended from each edge, and triangular caps on each end. One cap is the triangle itself and the other is placed at some distance from the light source. For an object that consists of many triangles, such as the sphere above, the volume can be defined by the so-called silhouette edges. These are edges that are on or near the boundary between the shadow volume and the portion of the object that is lit. In general, a silhouette edge borders a triangle that faces the light and another triangle that faces away from the light. To draw the shadow volume, one would find all of the silhouette edges and draw extended quads for each edge. The caps of the volume could be determined by making a closed polygon (or triangle fan) that includes all the points on the silhouette edges, and similarly on the far end of the volume.

The shadow volume technique works in the following way. Imagine a ray that originates at the camera position and extends through a pixel on the near plane. Suppose that we follow that ray and keep track of a counter that is incremented every time that it enters a shadow volume and decremented each time that it exits a shadow volume. If we stop counting when we hit a surface, that point on the surface is occluded (in shadow) if our count is non-zero, otherwise, the surface is lit by the light source. The following figure shows an example of this idea:



The roughly horizontal line represents a surface that is receiving a shadow. The numbers represent the counter for each camera ray. For example, the rightmost ray with value **+1** has that value because the ray entered two volumes and exited one along the way from the camera to the surface: $1 + 1 - 1 = 1$. The rightmost ray has a value of zero at the surface because it entered and exited both shadow volumes: $1 + 1 - 1 - 1 = 0$.

This all sounds fine in theory, but how can we trace rays in OpenGL? The good news is that we don't have to. The stencil buffer provides just what we need. With the stencil buffer, we can increment/decrement a counter for each pixel based on whether a front or back face is rendered into that pixel. So we can draw the boundaries of all of the shadow volumes, then for each pixel, increment the stencil buffer's counter when a front face is rendered to that pixel and decrement when it is a back face.

The key here is to realize that each pixel in the rendered figure represents an eye-ray (as in the above diagram). So for a given pixel, the value in the stencil buffer is the value that we would get if we actually traced a ray through that pixel. The depth test helps to stop tracing when we reach a surface.



The above is just a quick introduction to shadow volumes, a full discussion is beyond the scope of this book. For more detail, a great resource is *Real Time Shadows* by Eisemann et al.

In this recipe, we'll draw our shadow volumes with the help of the geometry shader. Rather than computing the shadow volumes on the CPU side, we'll render the geometry normally, and have the geometry shader produce the shadow volumes. In the *Drawing silhouette lines using the geometry shader* recipe in Chapter 6, *Using Geometry and Tessellation Shaders*, we saw how the geometry shader can be provided with adjacency information for each triangle. With adjacency information, we can determine whether a triangle has a silhouette edge. If the triangle faces the light, and a neighboring triangle faces away from the light, then the shared edge can be considered a silhouette edge, and used to create a polygon for the shadow volume.

The entire process is done in three passes. They are as follows:

- ▶ Render the scene normally, but write the shaded color to two separate buffers. We'll store the ambient component in one and the diffuse and specular components in another.
- ▶ Set up the stencil buffer so that the stencil test always passes, and front faces cause an increment and back faces cause a decrement. Make the depth buffer read-only, and render only the shadow-casting objects. In this pass, the geometry shader will produce the shadow volumes, and only the shadow volumes will be rendered to the fragment shader.
- ▶ Set up the stencil buffer so the test succeeds when the value is equal to zero. Draw a screen-filling quad, and combine the values of the two buffers from step one when the stencil test succeeds.

That's the high-level view, and there are many details. Let's go through them in the next sections.

Getting ready

We'll start by creating our buffers. We'll use a framebuffer object with a depth attachment and two color attachments. The ambient component can be stored in a renderbuffer (as opposed to a texture) because we'll blit (a fast copy) it over to the default framebuffer rather than reading from it as a texture. The diffuse + specular component will be stored in a texture. Create the ambient buffer (`ambBuf`), a depth buffer (`depthBuf`), and a texture (`diffSpecTex`), then set up the FBO.

```
glGenFramebuffers(1, &colorDepthFBO);  
 glBindFramebuffer(GL_FRAMEBUFFER, colorDepthFBO);  
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  
                         GL_RENDERBUFFER, depthBuf);  
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,  
                         GL_RENDERBUFFER, ambBuf);  
 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,  
                      GL_TEXTURE_2D, diffSpecTex, 0);
```

Set up the draw buffers so that we can write to the color attachments.

```
GLenum drawBuffers[] = {GL_COLOR_ATTACHMENT0,
                       GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, drawBuffers);
```

How to do it...

For the first pass, enable the framebuffer object that we set up above, and render the scene normally. In the fragment shader, send the ambient component and the diffuse + specular component to separate outputs.

```
layout( location = 0 ) out vec4 Ambient;
layout( location = 1 ) out vec4 DiffSpec;

void shade()
{
    // Compute the shading model, and separate out the ambient
    // component.
    Ambient = ...;    // Ambient
    DiffSpec = ...;   // Diffuse + specular
}
void main() { shade(); }
```

In the second pass, we'll render our shadow volumes. We want to set up the stencil buffer so that the test always succeeds, and that front faces cause an increment, and back faces cause a decrement.

```
glClear(GL_STENCIL_BUFFER_BIT);
 glEnable(GL_STENCIL_TEST);
 glStencilFunc(GL_ALWAYS, 0, 0xffff);
 glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
 glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR_WRAP);
```

Also in this pass, we want to use the depth buffer from the first pass, but we want to use the default frame buffer, so we need to copy the depth buffer over from the FBO used in the first pass. We'll also copy over the color data, which should contain the ambient component.

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, colorDepthFBO);
 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
 glBlitFramebuffer(0,0,width,height,0,0,width,height,
                  GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

We don't want to write to the depth buffer or the color buffer in this pass, since our only goal is to update the stencil buffer, so we'll disable writing for those buffers.

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
 glDepthMask(GL_FALSE);
```

Next, we render the shadow-casting objects with adjacency information. In the geometry shader, we determine the silhouette edges and output only quads that define the shadow volume boundaries.

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 18 ) out;

in vec3 VPosition[];
in vec3 VNormal[];

uniform vec4 LightPosition; // Light position (eye coords)
uniform mat4 ProjMatrix; // Proj. matrix (infinite far plane)

bool facesLight( vec3 a, vec3 b, vec3 c )
{
    vec3 n = cross( b - a, c - a );
    vec3 da = LightPosition.xyz - a;
    vec3 db = LightPosition.xyz - b;
    vec3 dc = LightPosition.xyz - c;
    return dot(n, da) > 0 || dot(n, db) > 0 || dot(n, dc) > 0;
}

void emitEdgeQuad( vec3 a, vec3 b ) {
    gl_Position = ProjMatrix * vec4(a, 1);
    EmitVertex();
    gl_Position = ProjMatrix * vec4(a - LightPosition.xyz, 0);
    EmitVertex();
    gl_Position = ProjMatrix * vec4(b, 1);
    EmitVertex();
    gl_Position = ProjMatrix * vec4(b - LightPosition.xyz, 0);
    EmitVertex();
    EndPrimitive();
}

void main()
{
    if( facesLight(VPosition[0], VPosition[2], VPosition[4]) ) {
        if( ! facesLight(VPosition[0],VPosition[1],VPosition[2]) )
            emitEdgeQuad(VPosition[0],VPosition[2]);
        if( ! facesLight(VPosition[2],VPosition[3],VPosition[4]) )
            emitEdgeQuad(VPosition[2],VPosition[4]);
        if( ! facesLight(VPosition[4],VPosition[5],VPosition[0]) )
            emitEdgeQuad(VPosition[4],VPosition[0]);
    }
}
```

Shadows

In the third pass, we'll set up our stencil buffer so that the test passes only when the value in the buffer is equal to zero.

```
glStencilFunc(GL_EQUAL, 0, 0xffff);  
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

We want to enable blending so that our ambient component is combined with the diffuse + specular when the stencil test succeeds.

```
	glEnable(GL_BLEND);  
	glBlendFunc(GL_ONE,GL_ONE);
```

In this pass, we just draw a screen-filling quad, and output the diffuse + specular value. If the stencil test succeeds, the value will be combined with the ambient component, which is already in the buffer (we copied it over earlier using `glBlitFramebuffer`).

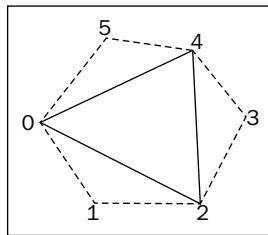
```
layout(binding = 0) uniform sampler2D DiffSpecTex;  
layout(location = 0) out vec4 FragColor;  
  
void main() {  
    vec4 diffSpec = texelFetch(DiffSpecTex, ivec2(gl_FragCoord), 0);  
    FragColor = vec4(diffSpec.xyz, 1);  
}
```

How it works...

The first pass is fairly straightforward. We draw the entire scene normally, except we separate the ambient color from the diffuse and specular color, and send the results to different buffers.

The second pass is the core of the algorithm. Here we render only the objects that cast shadows and let the geometry shader produce the shadow volumes. Thanks to the geometry shader, we don't actually end up rendering the shadow-casting objects at all, only the shadow volumes. However, before this pass, we need to do a bit of setup. We set up the stencil test so that it increments when a front face is rendered and decrements for back faces using `glStencilOpSeparate`, and the stencil test is configured to always succeed using `glStencilFunc`. We also use `glBlitFramebuffer` to copy over the depth buffer and (ambient) color buffer from the FBO used in the first pass. Since we want to only render shadow volumes that are not obscured by geometry, we make the depth buffer read-only using `glDepthMask`. Lastly, we disable writing to the color buffer using `glColorMask` because we don't want to mistakenly overwrite anything in this pass.

The geometry shader does the work of producing the silhouette shadow volumes. Since we are rendering using adjacency information (see the *Drawing silhouette lines using the geometry shader* recipe in Chapter 6, *Using Geometry and Tessellation Shaders*), the geometry shader has access to six vertices that define the current triangle being rendered and the three neighboring triangles. The vertices are numbered from 0 to 5, and are available via the input array named `vPosition` in this example. Vertices 0, 2, and 4 define the current triangle and the others define the adjacent triangles as shown in the following figure:



The geometry shader starts by testing the main triangle (0, 2, 4) to see if it faces the light source. We do so by computing the normal to the triangle (n) and the vector from each vertex to the light source. Then we compute the dot product of n and each of the three light source direction vectors (da , db , and dc). If any of the three are positive, then the triangle faces the light source. If we find that triangle (0, 2, 4) faces the light, then we test each neighboring triangle in the same way. If a neighboring triangle does not face the light source, then the edge between them is a silhouette edge and can be used as an edge of a face of the shadow volume.

We create a shadow volume face in the `emitEdgeQuad` function. The points `a` and `b` define the silhouette edge, one edge of the shadow volume face. The other two vertices of the face are determined by extending `a` and `b` away from the light source. Here, we use a mathematical trick that is enabled by homogeneous coordinates. We extend the face out to infinity by using a zero in the `w` coordinate of the extended vertices. This effectively defines a homogeneous vector, sometimes called a point at infinity. The `x`, `y` and `z` coordinates define a vector in the direction away from the light source, and the `w` value is set to zero. The end result is that we get a quad that extends out to infinity, away from the light source.



Note that this will only work properly if we use a modified projection matrix that can take into account points defined in this way. Essentially, we want a projection matrix with a far plane set at infinity. GLM provides just such a projection matrix via the function `infinitePerspective`.

We don't worry about drawing the caps of the shadow volume here. We don't need a cap at the far end, because we've used the homogeneous trick described above, and the object itself will serve as the cap on the near end.

In the third and final pass, we reset our stencil test to pass when the value in the stencil buffer is equal to zero using `glStencilFunc`. Here we want to sum the ambient with the diffuse + specular color when the stencil test succeeds, so we enable blending, and set the source and destination blend functions to `GL_ONE`. We render just a single screen-filling quad, and output the value from the texture that contains our diffuse + specular color. The stencil test will take care of discarding fragments that are in shadow, and OpenGL's blending support will blend the output with the ambient color for fragments that pass the test. (Remember that we copied over the ambient color using `glBlitFramebuffer` earlier.)

There's more...

The technique described above is often referred to as the **z-pass** technique. It has one fatal flaw. If the camera is located within a shadow volume, this technique breaks down because the counts in the stencil buffer will be off by at least one. The common solution is to basically invert the problem and trace a ray from infinity towards the view point. This is called the z-fail technique or Carmack's reverse.



The "fail" and "pass" here refers to whether or not we are counting when the depth test passes or fails.



Care must be taken when using z-fail because it is important to draw the caps of the shadow volumes. However, the technique is very similar to z-pass. Instead of incrementing/decrementing when the depth test passes, we do so when the depth test fails. This effectively "traces" a ray from infinity back towards the view point.

I should also note that the preceding code is not robust to degenerate triangles (triangles that have sides that are nearly parallel), or non-closed meshes. One might need to take care in such situations. For example, to better deal with degenerate triangles we could use another technique for determining the normal to the triangle. We could also add additional code to handle edges of meshes, or simply always use closed meshes.

See also

- ▶ The Drawing silhouette lines using the geometry shader recipe in Chapter 6, *Using Geometry and Tessellation Shaders*

8

Using Noise in Shaders

In this chapter, we will cover:

- ▶ Creating a noise texture using GLM
- ▶ Creating a seamless noise texture
- ▶ Creating a cloud-like effect
- ▶ Creating a wood-grain effect
- ▶ Creating a disintegration effect
- ▶ Creating a paint-spatter effect
- ▶ Creating a night-vision effect

Introduction

It's easy to use shaders to create a smooth-looking surface, but that is not always the desired goal. If we want to create realistic-looking objects, we need to simulate the imperfections of real surfaces. That includes things such as scratches, rust, dents, and erosion. It is somewhat surprising how challenging it can be to make surfaces look like they have really been subjected to these natural processes. Similarly, we sometimes want to represent natural surfaces such as wood grain or natural phenomena such as clouds to be as realistic as possible without giving the impression of being synthetic or exhibiting a repetitive pattern or structure.

Most effects or patterns in nature exhibit a certain degree of randomness and non-linearity. Therefore, you might imagine that we could generate them by simply using random data. However, random data such as the kind that is generated from a pseudorandom-number generator is not very useful in computer graphics. There are two main reasons:

- ▶ First, we need data that is repeatable, so that the object will render in the same way during each frame of the animation. (We could achieve this by using an appropriate seed value for each frame, but that only solves half of the problem.)

- ▶ Second, in order to model most of these natural phenomena, we actually need data that is continuous, but still gives the appearance of randomness. Continuous data more accurately represents many of these natural materials and phenomena. Purely random data does not have this continuity property. Each value has no dependence on the previous value.

Thanks to the groundbreaking work of Ken Perlin, we have the concept of **noise** (as it applies to computer graphics). His work defined noise as a function that has certain qualities such as the following:

- ▶ It is a continuous function
- ▶ It is repeatable (generates the same output from the same input)
- ▶ It can be defined for any number of dimensions
- ▶ It does not have any regular patterns and gives the appearance of randomness

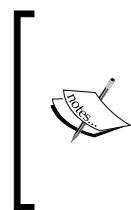
Such a noise function is a valuable tool for computer graphics and it can be used to create an endless array of interesting effects. For instance, in this chapter, we'll use noise to create clouds, wood, disintegration, and other effects.

Perlin noise is the noise function originally defined by Ken Perlin (see <http://mrl.nyu.edu/~perlin/doc/oscar.html>). A full discussion of the details behind Perlin noise is outside the scope of this book.

To use Perlin noise within a shader, we have the following three main choices:

1. We can use the built-in GLSL noise functions.
2. We can create our own GLSL noise functions.
3. We can use a texture map to store pre-computed noise data.

At the time of writing, the GLSL noise functions are not implemented in some of the commercial OpenGL drivers, and therefore cannot be relied upon to be available, so I have decided not to use them in this chapter. As creating our own noise functions is a bit beyond the scope of this book, and because choice 3 in the preceding list gives the best performance on modern hardware, the recipes in this chapter will use the third approach (using a pre-computed noise texture).



Many books use a 3D noise texture rather than a 2D one, to provide another dimension of noise that is available to the shaders. To keep things simple, and to focus on using surface texture coordinates, I've chosen to use a 2D noise texture in the recipes within this chapter. If desired, it should be straightforward to extend these recipes to use a 3D source of noise.

We'll start out with two recipes that demonstrate how to generate a noise texture using GLM. Then we'll move on to several examples that use noise textures to produce natural and artificial effects such as wood grain, clouds, electrical interference, splattering, and erosion.

The recipes in this chapter are meant to be a starting point for you to experiment with. They are certainly not intended to be the definitive way of implementing any of these effects. One of the best things about computer graphics is the element of creativity. Try tweaking the shaders in these recipes to produce similar results and then try creating your own effects. Most of all, have fun!

See Also...

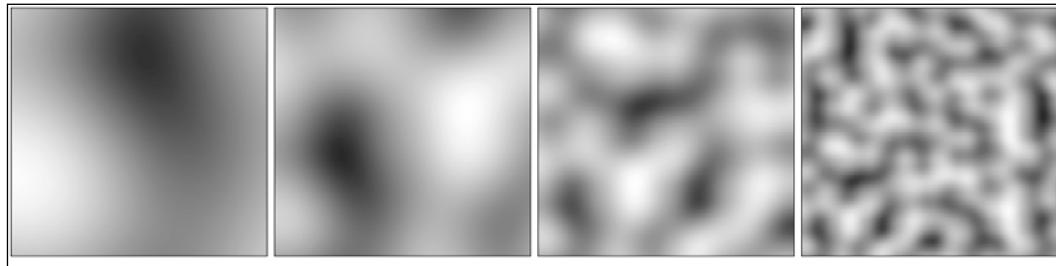
- ▶ The book *Texturing and Modeling: A Procedural Approach*, by Ken Musgrave et al

Creating a noise texture using GLM

To create a texture for use as a source of noise, we need some way to generate noise values. Implementing a proper noise generator from scratch can be a fairly daunting task. Luckily, GLM provides some functions for noise generation that are straightforward and easy to use.

In this recipe, we'll use GLM to generate a 2D texture of noise values created using a **Perlin noise** generator. GLM can generate 2D, 3D, and 4D Perlin noise via the function `glm::perlin`.

It is common practice to use Perlin noise by summing the values of the noise function with increasing frequencies and decreasing amplitudes. Each frequency is commonly referred to as an **octave** (double the frequency). For example, in the following image, we show the results of the 2D Perlin noise function sampled at four different octaves. The sampling frequencies increase from left to right. The leftmost image is the function sampled at our base frequency, and each image to the right shows the function sampled at twice the frequency of the one to its left.



In mathematical terms, if our coherent 2D Perlin noise function is $P(x, y)$, then each previous image represents the following equation.

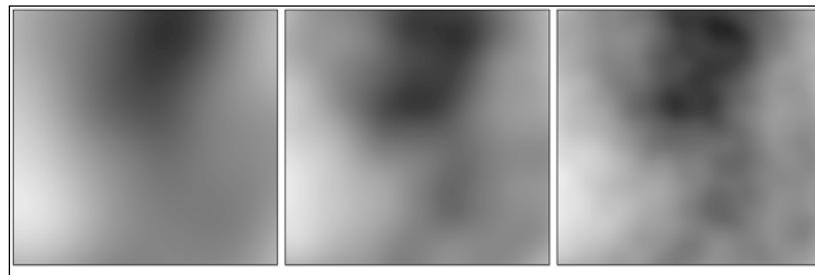
$$n_i(x, y) = P(2^i x, 2^i y)$$

Here $i = 0, 1, 2$, and 3 from left to right.

As mentioned previously, the common practice is to sum octaves together to get the final result. We add each octave to the previous, scaling the amplitude down by some factor. So for N octaves, we have the following sum.

$$n(x, y) = \sum_{i=0}^{N-1} \frac{P(2^i ax, 2^i ay)}{b^i}$$

Where a and b are tunable constants. The following images show the sum of 2, 3, and 4 octaves (left to right) with $a = 1$ and $b = 2$.



Summed noise involving higher octaves will have more high-frequency variation than noise involving only lower octaves. However, it is possible to quickly reach frequencies that exceed the resolution of the buffer used to store the noise data, so care must be taken not to do unnecessary computation. In practice, it is both an art and a science. The previous equation can be used as a starting point; feel free to tweak until you get the desired effect.

We'll store four noise values in a single 2D texture. We'll store Perlin noise with one octave in the first component (red channel), two octaves in the green channel, three octaves in the blue, and four octaves in the alpha channel.

Getting ready

Make sure you have the GLM library installed and placed in the include path.

How to do it...

To create a 2D noise texture with GLM, use the following steps:

1. Include the GLM header that includes the noise functions.

```
#include <glm/gtc/noise.hpp>
```

2. Generate the noise data, using the previous equation:

```
GLubyte *data = new GLubyte[ width * height * 4 ];  
  
float xFactor = 1.0f / (width - 1);  
float yFactor = 1.0f / (height - 1);  
  
for( int row = 0; row < height; row++ ) {  
    for( int col = 0 ; col < width; col++ ) {  
        float x = xFactor * col;  
        float y = yFactor * row;  
        float sum = 0.0f;  
        float freq = a;  
        float scale = b;  
  
        // Compute the sum for each octave  
        for( int oct = 0; oct < 4; oct++ ) {  
            glm::vec2 p(x * freq, y * freq);  
            float val = glm::perlin(p) / scale;  
            sum += val;  
            float result = (sum + 1.0f)/ 2.0f;  
  
            // Store in texture buffer  
            data[((row * width + col) * 4) + oct] =  
                (GLubyte) ( result * 255.0f );  
            freq *= 2.0f;    // Double the frequency  
            scale *= b;      // Next power of b  
        }  
    }  
}
```

3. Load the data into an OpenGL texture.

```
GLuint texID;  
 glGenTextures(1, &texID);  
  
 glBindTexture(GL_TEXTURE_2D, texID);  
 glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width, height);  
 glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,  
                 GL_RGBA, GL_UNSIGNED_BYTE, data);  
  
 delete [] data;
```

How it works...

The GLM library provides 2D, 3D, and 4D coherent noise via the `glm::perlin` function. It returns a float roughly between -1 and 1. We start by allocating a buffer named `data` to hold the generated noise values.

Next, we loop over each texel and compute the `x` and `y` coordinates (normalized). Then we loop over octaves. Here we compute the sum of the previous equation, storing the first term in the first component, the first two terms in the second, and so on. The value is scaled into the range from 0 to 1, then multiplied by 255 and cast to a byte.

The next few lines of code should be familiar. Texture memory is allocated with `glTexStorage2D` and the data is loaded in to GPU memory using `glTexSubImage2D`.

Finally, the array named `data` is deleted, as it is no longer needed.

There's more...

Rather than using unsigned byte values, we could get more resolution in our noise data by using a floating-point texture. This might provide better results if the effect needs a high degree of fine detail. The preceding code needs relatively few changes to achieve this. Just use an internal format of `GL_RGBA32F` instead of `GL_RGBA`, and don't multiply by 255 when storing the noise values in the array.

GLM also provides periodic Perlin noise via an overload of the `glm::perlin` function. This makes it easy to create noise textures that tile without seams. We'll see how to use this in the next recipe.

See also

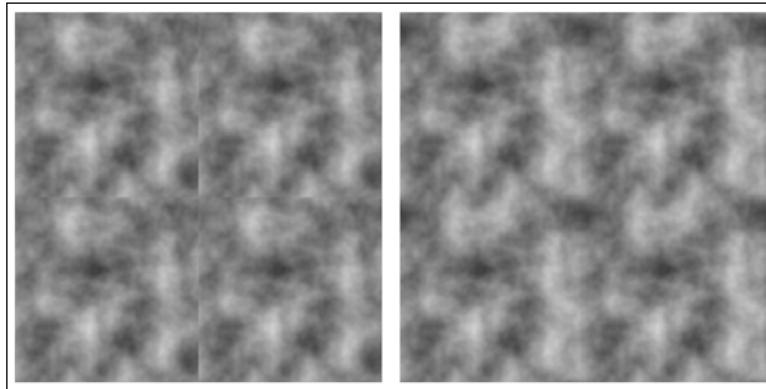
- ▶ For general information about coherent noise, take a look at the book *Graphics Shaders*, by Mike Bailey and Steve Cunningham
- ▶ The *Applying a 2D texture* recipe in *Chapter 4, Using Textures*
- ▶ The *Creating a seamless noise texture* recipe

Creating a seamless noise texture

It can be particularly useful to have a noise texture that tiles well. If we simply create a noise texture as a finite slice of noise values, then the values will not wrap smoothly across the boundaries of the texture. This can cause hard edges (seams) to appear in the rendered surface if the texture coordinates extend outside of the range of zero to one.

Fortunately, GLM provides a periodic variant of Perlin noise that can be used to create a seamless noise texture.

The following image shows an example of regular (left) and periodic (right) 4-octave Perlin noise. Note that in the left image, the seams are clearly visible, while they are hidden in the right image.



In this example, we'll modify the code from the previous recipe to produce a seamless noise texture.

Getting ready

For this recipe, we'll start with the code from the previous recipe, *Creating a noise texture using GLM*.

How to do it...

Modify the code from the previous recipe in the following way.

Within the innermost loop, instead of calling `glm::perlin`, we'll instead call the overload that provides periodic Perlin noise. Replace the statement

```
float val = glm::perlin(p) / scale;
```

with the following:

```
float val = 0.0f;
if( periodic ) {
    val = glm::perlin(p, glm::vec2(freq)) / scale;
} else {
    val = glm::perlin(p) / scale;
}
```

How it works...

The second parameter to `glm::perlin` determines the period in x and y of the noise values. We use `freq` as the period because we are sampling the noise in the range from 0 to `freq` for each octave.

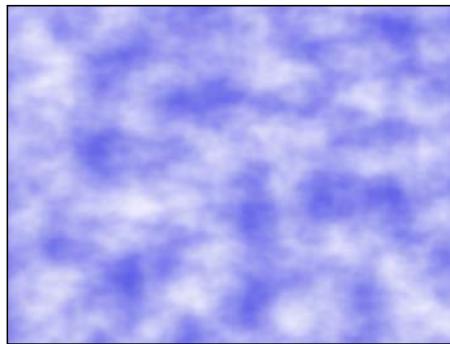
See also

- ▶ The [Creating a noise texture using GLM](#) recipe

Creating a cloud-like effect

To create a texture that resembles a sky with clouds, we can use the noise values as a blending factor between the sky color and the cloud color. As clouds usually have large scale structure, it makes sense to use low octave noise. However, the large scale structure often has higher frequency variations, so some contribution from higher octave noise may be desired.

The following image shows an example of clouds generated by the technique in this recipe:



To create this effect, we take the cosine of the noise value and use the result as the blending factor between the cloud color.

Getting ready

Set up your program to generate a seamless noise texture and make it available to the shaders through the uniform sampler variable `NoiseTex`.

There are two uniforms in the fragment shader that can be assigned from the OpenGL program:

- ▶ `SkyColor`: The background sky color
- ▶ `CloudColor`: The color of the clouds

How to do it...

To create a shader program that uses a noise texture to create a cloud-like effect, use the following steps:

1. Set up your vertex shader to pass the texture coordinates to the fragment shader via the variable TexCoord.
2. Use the following code for the fragment shader:

```
#define PI 3.14159265

layout( binding=0 ) uniform sampler2D NoiseTex;

uniform vec4 SkyColor = vec4( 0.3, 0.3, 0.9, 1.0 );
uniform vec4 CloudColor = vec4( 1.0, 1.0, 1.0, 1.0 );

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    vec4 noise = texture(NoiseTex, TexCoord);
    float t = (cos( noise.g * PI ) + 1.0) / 2.0;
    vec4 color = mix( SkyColor, CloudColor, t );
    FragColor = vec4( color.rgb , 1.0 );
}
```

How it works...

We start by retrieving the noise value from the noise texture (variable `noise`). The green channel contains two octave noises, so we use the value stored in that channel (`noise.g`). Feel free to try out other channels and determine what looks right to you.

We use a cosine function to make a sharper transition between the cloud and sky color. The noise value will be between zero and one, and the cosine of that value will range between -1 and 1, so we add 1.0 and divide by 2.0. The result that is stored in `t` should again range between zero and one. Without this cosine transformation, the clouds look a bit too spread out over the sky. However, if that is the desired effect, one could remove the cosine and just use the noise value directly.

Next, we mix the sky color and the cloud color using the value of `t`. The result is used as the final output fragment color.

There's more...

If you desire less clouds and more sky, you could translate and clamp the value of t prior to using it to mix the cloud and sky colors. For example, you could use the following code:

```
float t = (cos( noise.g * PI ) + 1.0 ) / 2.0;  
t = clamp( t - 0.25, 0.0, 1.0 );
```

This causes the cosine term to shift down (toward negative values), and the `clamp` function sets all negative values to zero. This has the effect of increasing the amount of sky and decreasing the size and intensity of the clouds.

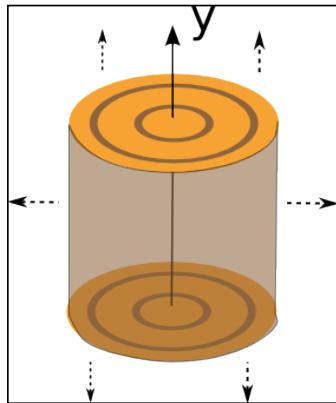
See also

- ▶ Further reading on cloud generation: <http://vterrain.org/Atmosphere/Clouds/>
- ▶ The *Creating a seamless noise texture* recipe

Creating a wood-grain effect

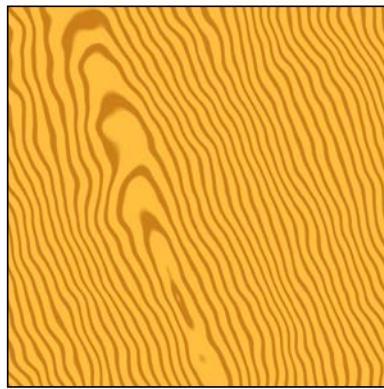
To create the look of wood, we can start by creating a virtual "log", with perfectly cylindrical growth rings. Then we'll take a slice of the log, and perturb the growth rings using noise from our noise texture.

The following image illustrates our virtual "log". It is aligned with the y-axis, and extends infinitely in all directions. The growth rings are aligned with integer distances from the y-axis. Each ring is given a darker color with lighter color in between rings. Each growth ring spans a narrow distance around the integer distances.



To take a "slice", we'll simply define a 2D region of the log's space based on the texture coordinates. Initially, the texture coordinates define a square region, with coordinates ranging from zero to one. We'll assume that the region is aligned with the x-y plane, so that the s coordinate corresponds to x, the t coordinate corresponds to y, and the value of z is zero. We can then transform this region in any way that suits our fancy, to create an arbitrary 2D slice.

After defining the slice, we'll determine the color based on the distance from the y-axis. However, before doing so, we'll perturb that distance based on a value from the noise texture. The result has a general look that is similar to real wood. The following image shows an example:



Getting ready

Set up your program to generate a noise texture and make it available to the shaders through the uniform variable `NoiseTex`.

There are three uniforms in the fragment shader that can be assigned from the OpenGL program. They are as follows:

- ▶ `LightWoodColor`: The lightest wood color
- ▶ `DarkWoodColor`: The darkest wood color
- ▶ `Slice`: A matrix that defines the slice of the virtual "log" and transforms the default region defined by the texture coordinates to some other arbitrary rectangular region

How to do it...

To create a shader program that generates a wood-grain effect using a noise texture, use the following steps:

1. Set up your vertex shader to pass the texture coordinate to the fragment shader via the variable `TexCoord`.

2. Use the following code for the fragment shader:

```
layout(binding=0) uniform sampler2D NoiseTex;

uniform vec4 DarkWoodColor = vec4( 0.8, 0.5, 0.1, 1.0 );
uniform vec4 LightWoodColor = vec4( 1.0, 0.75, 0.25, 1.0 );
uniform mat4 Slice;

in vec2 TexCoord;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    // Transform the texture coordinates to define the
    // "slice" of the log.
    vec4 cyl = Slice * vec4( TexCoord.st, 0.0, 1.0 );

    // The distance from the log's y axis.
    float dist = length(cyl.xz);

    // Perturb the distance using the noise texture
    vec4 noise = texture(NoiseTex, TexCoord);
    dist += noise.b;

    // Determine the color as a mixture of the light and
    // dark wood colors.
    float t = 1.0 - abs( fract( dist ) * 2.0 - 1.0 );
    t = smoothstep( 0.2, 0.5, t );
    vec4 color = mix( DarkWoodColor, LightWoodColor, t );

    FragColor = vec4( color.rgb , 1.0 );
}
```

How it works...

The first line of the `main` function within the fragment shader expands the texture coordinates to a 3D (homogeneous) value with a z coordinate of zero (s, t, 0, 1), and then transforms the value by the matrix `Slice`. This matrix can scale, translate, and/or rotate the texture coordinates to define the 2D region of the virtual "log".



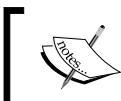
One way to visualize this is to think of the slice as a 2D unit square embedded in the "log" with its lower-left corner at the origin. The matrix is then used to transform that square within the log to define a slice through the log. For example, I might just translate the square by (-0.5, -0.5, -0.5) and scale by 20 in x and y to get a slice through the middle of the log.

Next, the distance from the y-axis is determined by using the built-in `length` function (`length(cyl.xz)`). This will be used to determine how close we are to a growth ring. The color will be a light wood color if we are between growth rings, and a dark color when we are close to a growth ring. However, before determining the color, we perturb the distance slightly using a value from our noise texture by using the following line of code:

```
dist += noise.b;
```

The next step is just a bit of numerical trickery to determine the color based on how close we are to a whole number. We start by taking the fractional part of the distance (`fract(dist)`), multiplying by two, subtracting one, and taking the absolute value. As `fract(dist)` is a value between zero and one, multiplying by two, subtracting one, and taking the absolute value will result in a value that is also between zero and one. However, the value will range from 1.0 when `dist` is 0.0, to 0.0 when `dist` is 0.5, and back to 1.0 when `dist` is 1.0 (a "v" shape).

We then invert the "v" by subtracting from one, and storing the result in `t`. Next, we use the `smoothstep` function to create a somewhat sharp transition between the light and dark colors. In other words, we want a dark color when `t` is less than 0.2, a light color when it is greater than 0.5, and a smooth transition in between. The result is used to mix the light and dark colors via the GLSL `mix` function.



The `smoothstep(a, b, x)` function works in the following way. It returns 0.0 when $x \leq a$, 1.0 when $x \geq b$ and uses Hermite interpolation between 0 and 1 when x is between a and b .



The result of all of this is a narrow band of the dark color around integer distances, and a light color in between, with a rapid, but smooth transition.

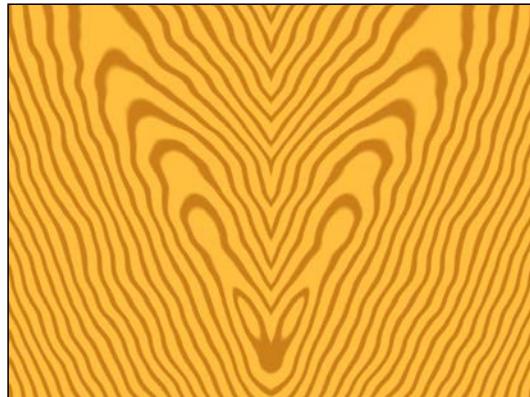
Finally, we simply apply the final color to the fragment.

There's more...

A book-matched pair of boards is a pair that is cut from the same log and then glued together. The result is a larger board that has symmetry in the grain from one side to the other. We can approximate this effect by mirroring the texture coordinate. For example, we could use the following in place of the first line of the preceding `main` function:

```
vec2 tc = TexCoord;  
if( tc.s > 0.5 ) tc.s = 1.0 - tc.s;  
vec4 cyl = Slice * vec4( tc, 0.0, 1.0 );
```

The following image shows an example of the results:



See also

- ▶ The *Creating a noise texture using GLM* recipe

Creating a disintegration effect

It is straightforward to use the GLSL `discard` keyword in combination with noise to simulate erosion or decay. We can simply discard fragments that correspond to a noise value that is above or below a certain threshold. The following image shows a teapot with this effect. Fragments are discarded when the noise value corresponding to the texture coordinate is outside a certain threshold range.



Getting ready

Set up your OpenGL program to provide position, normal, and texture coordinates to the shader. Be sure to pass the texture coordinate along to the fragment shader. Set up any uniforms needed to implement the shading model of your choice.

Create a seamless noise texture (see *Creating a seamless noise texture*), and place it in the appropriate texture channel.

The following uniforms are defined in the fragment shader, and should be set via the OpenGL program:

- ▶ `NoiseTex`: The noise texture.
- ▶ `LowThreshold`: Fragments are discarded if the noise value is below this value.
- ▶ `HighThreshold`: Fragments are discarded if the noise value is above this value.

How to do it...

To create a shader program that provides a disintegration effect, use the following steps:

1. Create a vertex shader that sends the texture coordinate to the fragment shader via the output variable `TexCoord`. It should also pass the position and normal to the fragment shader through the variables `Position` and `Normal`.
2. Use the following code for the fragment shader:

```
// Insert uniforms needed for the Phong shading model

layout(binding=0) uniform sampler2D NoiseTex;

in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform float LowThreshold;
uniform float HighThreshold;

layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel() {
    // Compute Phong shading model...
}
void main()
{
```

```
// Get the noise value at TexCoord  
vec4 noise = texture( NoiseTex, TexCoord );  
  
// If the value is outside the threshold, discard  
if( noise.a < LowThreshold || noise.a > HighThreshold)  
    discard;  
  
// Color the fragment using the shading model  
vec3 color = phongModel();  
FragColor = vec4( color , 1.0 );  
}
```

How it works...

The fragment shader starts by retrieving a noise value from the noise texture (`NoiseTex`), and storing the result in the variable `noise`. We want noise that has a large amount of high-frequency fluctuation, so we choose four-octave noise, which is stored in the alpha channel (`noise.a`).

We then discard the fragment if the noise value is below `LowThreshold` or above `HighThreshold`. As the `discard` keyword causes the execution of the shader to stop, the following statements will not execute if the fragment is discarded.



The discard operation can have a performance impact due to how it might affect early depth tests.



Finally, we compute the shading model and apply the result to the fragment.

See also

- ▶ The *Creating a seamless noise texture* recipe

Creating a paint-spatter effect

Using high-frequency noise, it is easy to create the effect of random spatters of paint on the surface of an object. The following image shows an example:



We use the noise texture to vary the color of the object, with a sharp transition between the base color and the paint color. We'll use either the base color or paint color as the diffuse reflectivity of the shading model. If the noise value is above a certain threshold, we'll use the paint color; otherwise, we'll use the base color of the object.

Getting ready

Start with a basic setup for rendering using the Phong shading model (or whatever model you prefer). Include texture coordinates and pass them along to the fragment shader.

There are a couple of uniform variables that define the parameters of the paint spatters:

- ▶ `PaintColor`: The color of the paint spatters
- ▶ `Threshold`: The minimum noise value where a spatter will appear

Create a noise texture with high-frequency noise.

Make your noise texture available to the fragment shader via the uniform sampler variable `NoiseTex`.

How to do it...

To create a shader program that generates a paint-spatter effect, use the following steps:

1. Create a vertex shader that sends the texture coordinates to the fragment shader via the output variable TexCoord. It should also pass the position and normal to the fragment shader through the variables Position and Normal.
2. Use the following code for the fragment shader:

```
// Uniforms for the Phong shading model
uniform struct LightInfo {
    vec4 Position;
    vec3 Intensity;
} Light;

uniform struct MaterialInfo {
    vec3 Ka;
    vec3 Kd;
    vec3 Ks;
    float Shininess;
} Material;

// The noise texture
layout(binding=0) uniform sampler2D NoiseTex;
// Input from the vertex shader
in vec4 Position;
in vec3 Normal;
in vec2 TexCoord;

// The paint-spatter uniforms
uniform vec3 PaintColor = vec3(1.0);
uniform float Threshold = 0.65;

layout ( location = 0 ) out vec4 FragColor;

vec3 phongModel(vec3 kd) {
    // Evaluate the Phong shading model
}

void main()
{
    vec4 noise = texture( NoiseTex, TexCoord );
    vec3 color = Material.Kd;
    if( noise.g > Threshold ) color = PaintColor;
    FragColor = vec4( phongModel(color) , 1.0 );
}
```

How it works...

The main function of the fragment shader retrieves a noise value from `NoiseTex`, and stores it in the variable `noise`. The next two lines set the variable `color` to either the base diffuse reflectivity (`Material.Kd`) or `PaintColor`, depending on whether or not the noise value is greater than the threshold value (`Threshold`). This will cause a sharp transition between the two colors and the size of the spatters will be related to the frequency of the noise.

Finally, the Phong shading model is evaluated using `color` as the diffuse reflectivity. The result is applied to the fragment.

There's more...

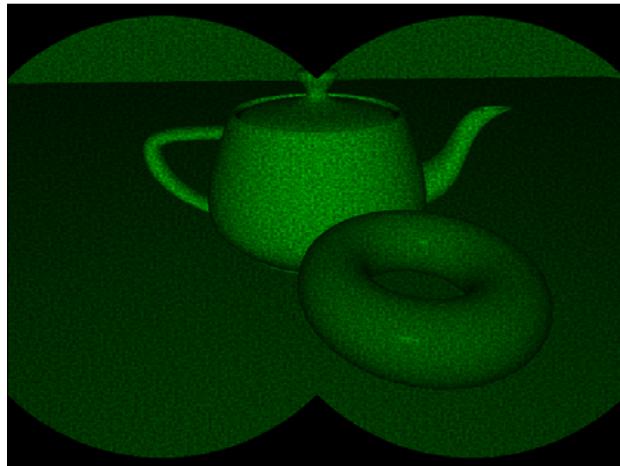
As indicated in the *Creating a noise texture using GLM* recipe using lower-frequency noise will cause the spatters to be larger in size and more spread out. A lower threshold will also increase the size without spreading over the surface, but as the threshold gets lower, it starts to look more uniform and less like random spattering.

See also

- ▶ The *Creating a seamless noise texture* recipe

Creating a night-vision effect

Noise can be useful to simulate static or other kinds of electronic interference effects. This recipe is a fun example of that. We'll create the look of night-vision goggles with some noise thrown in to simulate some random static in the signal. Just for fun, we'll also outline the scene in the classic "binocular" view. The following image shows an example:



We'll apply the night-vision effect as a second pass to the rendered scene. The first pass will render the scene to a texture (see *Chapter 4, Using Textures*), and the second pass will apply the night-vision effect.

Getting ready

Create a Framebuffer Object (FBO) for the first pass. Attach a texture to the first color attachment of the FBO. For more information on how to do this, see *Chapter 4, Using Textures*.

Create and assign any uniform variables needed for the shading model. Set the following uniforms defined in the fragment shader:

- ▶ **Width:** The width of the viewport in pixels
- ▶ **Height:** The height of the viewport in pixels
- ▶ **Radius:** The radius of each circle in the "binocular" effect (in pixels)
- ▶ **RenderTex:** The texture containing the render from the first pass
- ▶ **NoiseTex:** The noise texture
- ▶ **RenderPass:** The subroutine uniform used to select the functionality for each pass

Create a noise texture with high-frequency noise, and make it available to the shader via `NoiseTex`. Associate the texture with the FBO available via `RenderTex`.

How to do it...

To create a shader program that generates a night-vision effect, use the following steps:

1. Set up your vertex shader to pass along the position, normal, and texture coordinates via the variables `Position`, `Normal`, and `TexCoord` respectively.
2. Use the following code for the fragment shader:

```
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform int Width;
uniform int Height;
uniform float Radius;
layout(binding=0) uniform sampler2D RenderTex;
layout(binding=1) uniform sampler2D NoiseTex;

subroutine vec4 RenderPassType();
subroutine uniform RenderPassType RenderPass;

// Define any uniforms needed for the shading model.
```

```

layout( location = 0 ) out vec4 FragColor;

vec3 phongModel( vec3 pos, vec3 norm )
{
    // Compute the Phong shading model
}

// Returns the relative luminance of the color value
float luminance( vec3 color ) {
    return dot( color.rgb, vec3(0.2126, 0.7152, 0.0722) );
}

subroutine (RenderPassType)
vec4 pass1()
{
    return vec4(phongModel( Position, Normal ),1.0);
}

subroutine( RenderPassType )
vec4 pass2()
{
    vec4 noise = texture(NoiseTex, TexCoord);
    vec4 color = texture(RenderTex, TexCoord);
    float green = luminance( color.rgb );

    float dist1 = length(gl_FragCoord.xy -
        vec2(Width*0.25, Height*0.5));
    float dist2 = length(gl_FragCoord.xy -
        vec2(3.0*Width*0.25, Height*0.5));
    if( dist1 > Radius && dist2 > Radius ) green = 0.0;

    return vec4(0.0, green * clamp(noise.a + 0.25, 0.0, 1.0),
        0.0 ,1.0);
}

void main()
{
    // This will call either pass1() or pass2()
    FragColor = RenderPass();
}

```

3. In the render function of your OpenGL program, use the following steps:
 1. Bind to the FBO that you set up for rendering the scene to a texture.
 2. Select the `pass1` subroutine function in the fragment shader via `RenderPass`.
 3. Render the scene.

4. Bind to the default FBO.
5. Select the `pass2` subroutine function in the fragment shader via `RenderPass`.
6. Draw a single quad that fills the viewport using texture coordinates that range from 0 to 1 in each direction.

How it works...

The fragment shader is broken into two subroutine functions, one for each pass. Within the `pass1` function, we simply apply the Phong shading model to the fragment. The result is written to the FBO, which contains a texture to be used in the second pass.

In the second pass, the `pass2` function is executed. We start by retrieving a noise value (`noise`), and the color from the render texture from the first pass (`color`). Then we compute the luminance value for the color and store that result in the variable `green`. This will eventually be used as the green component of the final color.

The next step involves determining whether or not the fragment is inside the "binocular" lenses. We compute the distance to the center of the left lens (`dist1`), which is located in the viewport halfway from top to bottom and one-quarter of the way from left to right. The right lens is located at the same vertical location, but three-quarters of the way from left to right. The distance from the center of the right-hand lens is stored in `dist2`. If both `dist1` and `dist2` are greater than the radius of the virtual lenses, then we set `green` to zero.

Finally, we return the final color, which has only a green component; the other two are set to zero. The value of `green` is multiplied by the noise value in order to add some noise to the image to simulate random interference in the signal. We add 0.25 to the noise value and clamp it between zero and one, in order to brighten the overall image. I have found that it appeared a bit too dark if the noise value wasn't biased in this way.

There's more...

It would make this shader even more effective if the noise varied in each frame during animation to simulate interference that is constantly changing. We can accomplish this roughly by modifying the texture coordinates used to access the noise texture in a time-dependent way. See the blog post mentioned in *See also* section for an example.

See also

- ▶ The *Rendering to a texture* recipe in Chapter 4, *Using Textures*
- ▶ The *Creating a noise texture using GLM* recipe
- ▶ This recipe was inspired by a blog post by Wojciech Toman: (wtomandev.blogspot.com/2009/09/night-vision-effect.html)

9

Particle Systems and Animation

In this chapter, we will cover:

- ▶ Animating a surface with vertex displacement
- ▶ Creating a particle fountain
- ▶ Creating a particle system using transform feedback
- ▶ Creating a particle system using instanced particles
- ▶ Simulating fire with particles
- ▶ Simulating smoke with particles

Introduction

Shaders provide us with the ability to leverage the massive parallelism offered by modern graphics processors. Since they have the ability to transform the vertex positions, they can be used to implement animation directly within the shaders themselves. This can provide a bump in efficiency if the animation algorithm can be parallelized appropriately for execution within the shader.

If a shader is to help with animation, it must not only compute the positions, but often we need to write the updated positions for use in the next frame. Shaders were not originally designed to write to arbitrary buffers (except of course the framebuffer). However, with recent versions, OpenGL provides the ability to do so via shader storage buffer objects and image load/store. As of OpenGL 3.0, we can also send the values of the vertex or geometry shader's output variables to an arbitrary buffer (or buffers). This feature is called **Transform Feedback**, and is particularly useful for particle systems.

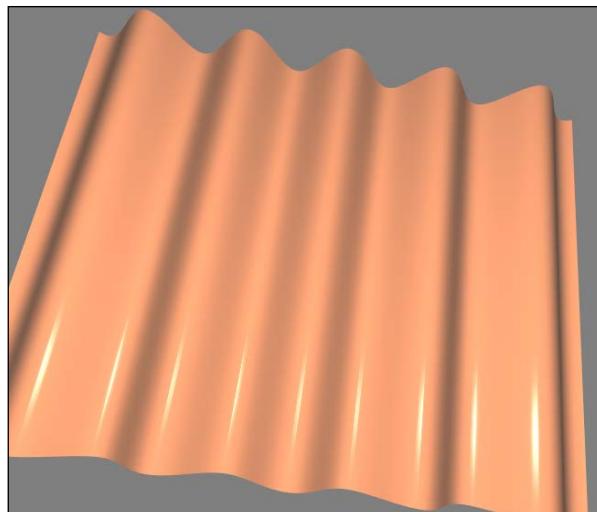
In this chapter, we'll look at several examples of animation within shaders, focusing mostly on particle systems. The first example, animating with vertex displacement, demonstrates animation by transforming the vertex positions of an object based on a time-dependent function. In the *Creating a particle fountain* recipe, we create a simple particle system under constant acceleration. In the *Creating a particle system using transform feedback* recipe there is an example illustrating how to use OpenGL's transform feedback functionality within a particle system. The *Creating a particle system using instanced particles* recipe shows you how to animate many complex objects using instanced rendering.

The last two recipes demonstrate some particle systems for simulating complex real phenomena such as smoke and fire.

Animating a surface with vertex displacement

A straightforward way to leverage shaders for animation is to simply transform the vertices within the vertex shader based on some time-dependent function. The OpenGL application supplies static geometry, and the vertex shader modifies the geometry using the current time (supplied as a uniform variable). This moves the computation of the vertex position from the CPU to the GPU, and leverages whatever parallelism the graphics driver makes available.

In this example, we'll create a waving surface by transforming the vertices of a tessellated quad based on a sine wave. We'll send down the pipeline a set of triangles that make up a flat surface in the x-z plane. In the vertex shader we'll transform the y-coordinate of each vertex based on a time-dependent sine function, and compute the normal vector of the transformed vertex. The following image shows the desired result. (You'll have to imagine that the waves are travelling across the surface from left to right).

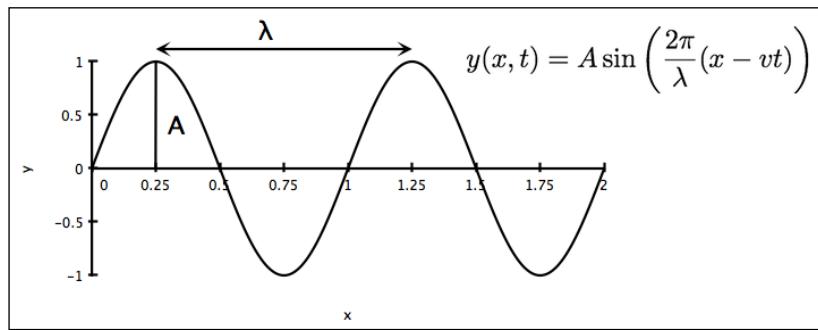




Alternatively, we could use a noise texture to animate the vertices (that make up the surface) based on a random function. (See *Chapter 8, Using Noise in Shaders*, for details on noise textures).

Before we jump into the code, let's take a look at the mathematics that we'll need.

We'll transform the y-coordinate of the surface as a function of the current time and the modeling x-coordinate. To do so, we'll use the basic plane wave equation as shown in the following diagram:



Where A is the wave's amplitude (the height of the peaks), λ is the wavelength (the distance between successive peaks), and v is the wave's velocity. The previous image shows an example of the wave when $t = 0$ and the wavelength is equal to one. We'll configure these coefficients through uniform variables.

In order to render the surface with proper shading, we also need the normal vector at the transformed location. We can compute the normal vector using the (partial) derivative of the previous function. The result is the following equation:

$$\mathbf{n}(x, t) = \left(-A \frac{2\pi}{\lambda} \cos\left(\frac{2\pi}{\lambda}(x - vt)\right), 1 \right)$$

Of course, the previous vector should be normalized before using it in our shading model.

Getting ready

Set up your OpenGL application to render a flat, tessellated surface in the x-z plane. The results will look better if you use a large number of triangles. Also, keep track of the animation time using whatever method you prefer. Provide the current time to the vertex shader via the uniform variable `Time`.

The other important uniform variables are the coefficients of the previous wave equation.

- ▶ **K:** It is the wavenumber ($2\pi/\lambda$).
- ▶ **Velocity:** It is the wave's velocity.
- ▶ **Amp:** It is the wave's amplitude.

Set up your program to provide appropriate uniform variables for your chosen shading model.

How to do it...

Use the following steps:

1. Use the following code for the vertex shader:

```
layout (location = 0) in vec3 VertexPosition;

out vec4 Position;
out vec3 Normal;

uniform float Time; // The animation time

// Wave parameters
uniform float K; // Wavenumber
uniform float Velocity; // Wave's velocity
uniform float Amp; // Wave's amplitude

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;

void main()
{
    vec4 pos = vec4(VertexPosition, 1.0);

    // Translate the y coordinate
    float u = K * (pos.x - Velocity * Time);
    pos.y = Amp * sin(u);

    // Compute the normal vector
    vec3 n = vec3(0.0);
    n.xy = normalize(vec2(-K * Amp * cos(u), 1.0));

    // Send position and normal (in camera cords) to frag.
    Position = ModelViewMatrix * pos;
    Normal = NormalMatrix * n;

    // The position in clip coordinates
    gl_Position = MVP * pos;
}
```

2. Create a fragment shader that computes the fragment color based on the variables `Position` and `Normal` using whatever shading model you choose (see the *Using per-fragment shading for improved realism* recipe in *Chapter 3, Lighting, Shading, and Optimizations*).

How it works...

The vertex shader takes the position of the vertex and updates the y-coordinate using the wave equation discussed previously. After the first three statements, the variable `pos` is just a copy of the input variable `VertexPosition` with the modified y-coordinate.

We then compute the normal vector using the previous equation, normalize the result and store it in the variable `n`. Since the wave is really just a two-dimensional wave (doesn't depend on z), the z component of the normal vector will be zero.

Finally we pass along the new position and normal to the fragment shader after converting to camera coordinates. As usual, we also pass the position in clip coordinates to the built-in variable `gl_Position`.

There's more...

Modifying the vertex position within the vertex shader is a straightforward way to offload some computation from the CPU to the GPU. It also eliminates the possible need to transfer vertex buffers between the GPU memory and main memory in order to modify the positions.

The main disadvantage is that the updated positions are not available on the CPU side. For example, they might be needed for additional processing (such as collision detection). However, there are a number of ways to provide this data back to the CPU. One technique might be clever use of FBOs to receive the updated positions from the fragment shader. In a following recipe, we'll look at another technique that makes use of a newer OpenGL feature called transform feedback.

See also

- ▶ The *Using per-fragment shading for improved realism* recipe in *Chapter 3, Lighting, Shading, and Optimization*

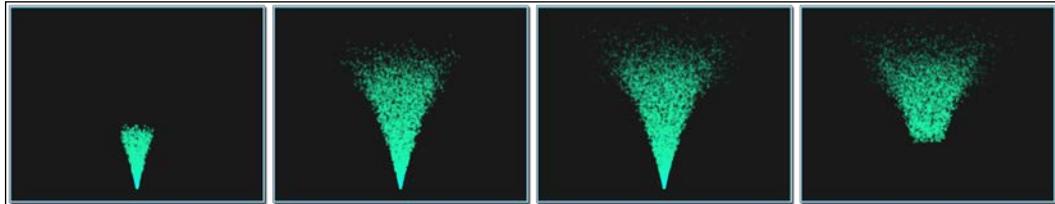
Creating a particle fountain

In computer graphics, a particle system is a group of objects that are used to simulate a variety of "fuzzy" systems like smoke, liquid spray, fire, explosions, or other similar phenomena. Each particle is considered to be a point object with a position, but no size. Often, they are rendered as point sprites (using the `GL_POINTS` primitive mode), but could also be rendered as camera aligned quads or triangles. Each particle has a lifetime: it is born, animates according to a set of rules, and then dies. The particle can then be resurrected and go through the entire process again. Generally particles do not interact with other particles, or reflect light. The particle is often rendered as a single, textured, camera-facing quad with transparency.

During the lifetime of a particle, it is animated according to a set of rules. These rules often include the basic kinematic equations that define the movement of a particle that is subjected to constant acceleration (such as a gravitational field). In addition, we might take into account things like wind, friction or other factors. The particle may also change shape or transparency during its lifetime. Once the particle has reached a certain age (or position), it is considered to be "dead" and can be "recycled" and used again.

In this example, we'll implement a relatively simple particle system that has the look of a fountain of water. For simplicity, the particles in this example will not be "recycled." Once they have reached the end of their lifetime, we'll draw them as fully transparent so that they are effectively invisible. This gives the fountain a finite lifetime, as if it only has a limited supply of material. In later recipes, we'll see some ways to improve this system by recycling particles.

The following sequence of images shows several successive frames from the output of this simple particle system.



To animate the particles, we'll use the standard kinematics equation for objects under constant acceleration.

$$P(t) = P_0 + \mathbf{v}_0 t + \frac{1}{2} \mathbf{a} t^2$$

The previous equation describes the position of a particle at time t . \mathbf{P}_0 is the initial position, \mathbf{v}_0 is the initial velocity, and \mathbf{a} is the acceleration.

We'll define the initial position of all particles to be the origin (0,0,0). The initial velocity will be determined randomly within a range of values. Each particle will be created at a slightly different time, so the time that we use in the previous equation will be relative to the start time for the particle.

Since the initial position is the same for all particles, we won't need to provide it as an input attribute to the shader. Instead, we'll just provide two other vertex attributes: the initial velocity and the start time (the particle's time of "birth"). Prior to the particle's birth time, we'll render it completely transparent. During its lifetime, the particle's position will be determined using the previous equation with a value for t that is relative to the particle's start time (Time - StartTime).

We'll render each particle as a textured point sprite (using `GL_POINTS`). It is easy to apply a texture to a point sprite because OpenGL will automatically generate texture coordinates and make them available to the fragment shader via the built-in variable `gl_PointCoord`. We'll also increase the transparency of the point sprite linearly with the age of the particle, to make the particle appear to fade out as it animates.

Getting ready

We'll create two buffers (or a single interleaved buffer) to store the input to the vertex shader. The first buffer will store the initial velocity for each particle. We'll choose the values randomly from a limited range of possible vectors. To create the vertical "cone" of particles in the previous image, we'll choose randomly from a set of vectors within the cone. The following code is one way to do this:

```
vec3 v(0.0f);
float velocity, theta, phi;
GLfloat *data = new GLfloat[nParticles * 3];
for( unsigned int i = 0; i < nParticles; i++ ) {
    // Pick the direction of the velocity
    theta = glm::mix(0.0f, (float)PI / 6.0f, randFloat());
    phi = glm::mix(0.0f, (float)TWOPI, randFloat());

    v.x = sinf(theta) * cosf(phi);
    v.y = cosf(theta);
    v.z = sinf(theta) * sinf(phi);

    // Scale to set the magnitude of the velocity (speed)
    velocity = glm::mix(1.25f, 1.5f, randFloat());
    v = v * velocity;

    data[3*i]    = v.x;
    data[3*i+1]  = v.y;
    data[3*i+2]  = v.z;
```

```
        }
        glBindBuffer(GL_ARRAY_BUFFER, initVel);
        glBufferSubData(GL_ARRAY_BUFFER, 0,
                        nParticles * 3 * sizeof(float), data);
```

In the previous code the `randFloat` function returns a random value between zero and one. We pick random numbers within a range of possible values by using the GLM `mix` function, (the GLM `mix` function works the same as the corresponding GLSL function. It performs a linear interpolation between the values of the first two arguments). Here, we choose a random float between zero and one and use that value to interpolate between the endpoints of our range.

To pick vectors from within our cone, we utilize spherical coordinates. The value of `theta` determines the angle between the center of the cone and the outer edge. The value of `phi` defines the possible directions around the y-axis for a given value of `theta`. For more on spherical coordinates, grab your favorite math book.

Once a direction is chosen, the vector is scaled to have a magnitude between 1.25 and 1.5. This is a range that seems to work well for the desired effect. The magnitude of the velocity vector is the overall speed of the particle, and we can tweak this range to get a wider variety of speeds or faster/slower particles.

The last three lines in the loop assign the vector to the appropriate location in the array `data`. After the loop, we copy the data into the buffer referred to by `initVel`. Set up this buffer to provide data for vertex attribute zero.

In the second buffer, we'll store the start time for each particle. This will provide only a single float per vertex (particle). For this example, we'll just create each particle in succession at a fixed rate. The following code will set up a buffer with each particle created 0.00075 seconds after the previous one.

```
float * data = new GLfloat[nParticles];
float time = 0.0f, rate = 0.00075f;

for( unsigned int i = 0; i < nParticles; i++ ) {
    data[i] = time;
    time += rate;
}
glBindBuffer(GL_ARRAY_BUFFER, startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), data);
```

This code simply creates an array of floats that starts at zero and gets incremented by `rate`. The array is then copied into the buffer referred to by `startTime`. Set this buffer to be the input for vertex attribute one.

Set the following uniform variables from within the OpenGL program:

- ▶ **ParticleTex**: It is the particle's texture.
- ▶ **Time**: It is the amount of time that has elapsed since the animation began.
- ▶ **Gravity**: It is the vector representing one half of the acceleration in the previous equation.
- ▶ **ParticleLifetime**: It defines how long a particle survives after it is created.

Make sure that the depth test is off, and enable alpha blending using the following statements:

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

You will also want to choose a reasonable size for each point sprite. For example, the following line sets it to 10 pixels:

```
glPointSize(10.0f);
```

How to do it...

Use the following code for the vertex shader:

```
// Initial velocity and start time
layout (location = 0) in vec3 VertexInitVel;
layout (location = 1) in float StartTime;

out float Transp; // Transparency of the particle

uniform float Time; // Animation time
uniform vec3 Gravity = vec3(0.0,-0.05,0.0); // world coords
uniform float ParticleLifetime; // Max particle lifetime

uniform mat4 MVP;

void main()
{
    // Assume the initial position is (0,0,0).
    vec3 pos = vec3(0.0);
    Transp = 0.0;

    // Particle doesn't exist until the start time
    if( Time > StartTime ) {
        float t = Time - StartTime;
```

```
if( t < ParticleLifetime ) {  
    pos = VertexInitVel * t + Gravity * t * t;  
    Transp = 1.0 - t / ParticleLifetime;  
}  
}  
  
gl_Position = MVP * vec4(pos, 1.0);  
}
```

Use the following code for the fragment shader:

```
in float Transp;  
uniform sampler2D ParticleTex;  
  
layout ( location = 0 ) out vec4 FragColor;  
  
void main()  
{  
    FragColor = texture(ParticleTex, gl_PointCoord);  
    FragColor.a *= Transp;  
}
```

How it works...

The vertex shader receives the particle's initial velocity (`VertexInitVel`) and start time (`StartTime`) in its two input attributes. The variable `Time` stores the amount of time that has elapsed since the beginning of the animation. The output variable `Transp` is the overall transparency of the particle.

In the main function of the vertex shader, we start by setting the initial position to the modeling origin (0,0,0), and the transparency to 0.0 (fully transparent). The following `if` statement determines whether the particle is alive yet. If the current time is greater than the start time for the particle, the particle is alive, otherwise, the particle has yet to be "born". In the latter case, the position remains at the origin and the particle is rendered fully transparent.

If the particle is alive, we determine the "age" of the particle by subtracting the start time from the current time, and storing the result in the variable `t`. If `t` is greater than or equal to the lifetime for a particle (`ParticleLifetime`), the particle has already fully evolved through its animation and is rendered fully transparent. Otherwise, the particle is still active and the body of the inner `if` statement executes, which is responsible for animating the particle.

If the particle is alive, the position (`pos`) is determined using the kinematic equation described previously. The transparency is determined by linearly interpolating based on the particle's age.

```
Transp = 1.0 - t / ParticleLifetime;
```

When the particle is born it is fully opaque, and linearly becomes transparent as it ages. The value of `Transp` is 1.0 at birth and 0.0 at the end of the particle's lifetime.

In the fragment shader, we color the fragment with the result of value of a texture lookup. Since we are rendering `GL_POINT` primitives, the texture coordinate is determined automatically by OpenGL and is available in the built-in variable `gl_PointCoord`. Before finishing, we multiply the alpha value of the final color by the variable `Transp`, in order to scale the overall transparency of the particle based on the particle's age (as determined in the vertex shader).

There's more...

This example is meant to be a fairly gentle introduction to GPU-based particle systems. There are many things that could be done to improve the power and flexibility of this system. For example, we could vary the size or rotation of the particles as they progress through their lifetime to produce different effects.

We could also create a better indication of distance by varying the size of the particles with the distance from the camera. This could be accomplished by defining the point size within the vertex shader using the built-in variable `gl_PointSize`, or drawing quads or triangles instead of points.



While point sprites (via `GL_POINTS`) are an obvious choice for particles, they also have several drawbacks. First, hardware often has fairly low limits on point size. If particles need to be large, or change size over time, the limits might be a severe restriction. Second, points are usually clipped when their center moves outside of the view volume. This can cause "popping" artefacts when the particles have a significant size. A common solution is to draw quads or triangles instead of points, using a technique like the one outlined in *Chapter 6, Using Geometry and Tessellation Shaders*.

One of the most significant drawbacks of the technique of this recipe is that the particles can't be recycled easily. When a particle dies, it is simply rendered as transparent. It would be nice to be able to re-use each dead particle to create an apparently continuous stream of particles. Additionally, it would be useful to be able to have the particles respond appropriately to changing accelerations or modifications of the system (for example wind or movement of the source). With the system described here, we couldn't do so because we are working with a single equation that defines the movement of the particle for all time. What would be needed is to incrementally update the positions based on the current forces involved (a simulation).

In order to accomplish the previous objective, we need some way to feed the output of the vertex shader (the particle's updated positions) back into the input of the vertex shader during the next frame. This would of course be simple if we weren't doing the simulation within the shader because we could simply update the positions of the primitives directly before rendering. However, since we are doing the work within the vertex shader we are limited in the ways that we can write to memory.

In the following recipe, we'll see an example of how to use a new OpenGL feature called **transform feedback** to accomplish exactly what was just described. We can designate certain output variables to be sent to buffers that can be read as input in subsequent rendering passes.

See also

- ▶ The *Animating a surface with vertex displacement* recipe
- ▶ The *Creating a particle system using transform feedback* recipe

Creating a particle system using transform feedback

Transform Feedback provides a way to capture the output of the vertex (or geometry) shader to a buffer for use in subsequent passes. Originally introduced into OpenGL with version 3.0, this feature is particularly well suited for particle systems because among other things, it enables us to do discrete simulations. We can update a particle's position within the vertex shader and render that updated position in a subsequent pass (or the same pass). Then the updated positions can be used in the same way as input to the next frame of animation.

In this example, we'll implement the same particle system from the previous recipe (*Creating a particle fountain*), this time making use of transform feedback. Instead of using an equation that describes the particle's motion for all time, we'll update the particle positions incrementally, solving the equations of motion based on the forces involved at the time each frame is rendered.

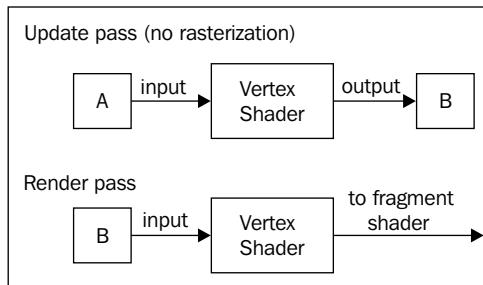
A common technique is to make use of the **Euler method**, which approximates the position and velocity at time t based on the position, velocity, and acceleration at an earlier time.

$$\begin{aligned}P_{n+1} &= P_n + \mathbf{v}_n h \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \mathbf{a}_n h\end{aligned}$$

In the previous equation the subscripts represent the time step (or animation frame), P is the particle position, and v is the particle velocity. The equations describe the position and velocity at frame $n + 1$ as a function of the position and velocity during the previous frame (n). The variable h represents the time step size, or the amount of time that has elapsed between frames. The term a_n represents the instantaneous acceleration that is computed based on the positions of the particles. For our simulation, this will be a constant value, but in general it might be a value that changes depending on the environment (wind, collisions, inter-particle interactions, and so on).

 The Euler method is actually numerically integrating the Newtonian equation of motion. It is one of the simplest techniques for doing so. However, it is a first-order technique, which means that it can introduce a significant amount of error. More accurate techniques include **Verlet integration**, and **Runge-Kutta integration**. Since our particle simulation is designed to look good and physical accuracy is not of high importance, the Euler method should suffice.

To make our simulation work, we'll use a technique sometimes called buffer "ping-ponging." We maintain two sets of vertex buffers and swap their uses each frame. For example, we use buffer A to provide the positions and velocities as input to the vertex shader. The vertex shader updates the positions and velocities using the Euler method and sends the results to buffer B using transform feedback. Then in a second pass, we render the particles using buffer B.



In the next frame of animation, we repeat the same process, swapping the two buffers.

In general, transform feedback allows us to define a set of shader output variables that are to be written to a designated buffer (or set of buffers). There are several steps involved that will be demonstrated, but the basic idea is as follows. Just before the shader program is linked, we define the relationship between buffers and shader output variables using the function `glTransformFeedbackVaryings`. During rendering, we initiate a transform feedback pass. We bind the appropriate buffers to the transform feedback binding points. (If desired, we can disable rasterization so that the particles are not rendered.) We enable transform feedback using the function `glBeginTransformFeedback` and then draw the point primitives. The output from the vertex shader will be stored in the appropriate buffers. Then we disable transform feedback by calling `glEndTransformFeedback`.

Getting ready

Create and allocate three pairs of buffers. The first pair will be for the particle positions, the second for the particle velocities, and the third for the "start time" for each particle (the time when the particle comes to life). For clarity, we'll refer to the first buffer in each pair as the A buffer, and the second as the B buffer. Also, we'll need a single buffer to contain the initial velocity for each particle.

Create two vertex arrays. The first vertex array should link the A position buffer with the first vertex attribute (attribute index 0), the A velocity buffer with vertex attribute one, the A start time buffer with vertex attribute two, and the initial velocity buffer with vertex attribute three. The second vertex array should be set up in the same way using the B buffers and the same initial velocity buffer. In the following code, the handles to the two vertex arrays will be accessed via the `GLuint` array named `particleArray`.

Initialize the A buffers with appropriate initial values. For example, all of the positions could be set to the origin, and the velocities and start times could be initialized in the same way as described in the previous recipe *Creating a particle fountain*. The initial velocity buffer could simply be a copy of the velocity buffer.

When using transform feedback, we define the buffers that will receive the output data from the vertex shader by binding the buffers to the indexed binding points under the `GL_TRANSFORM_FEEDBACK_BUFFER` target. The index corresponds to the index of the vertex shader's output variable as defined by `glTransformFeedbackVaryings`.

To help simplify things, we'll make use of transform feedback objects. Use the following code to set up two transform feedback objects for each set of buffers:

```
GLuint feedback[2]; // Transform feedback objects
GLuint posBuf[2]; // Position buffers (A and B)
GLuint velBuf[2]; // Velocity buffers (A and B)
GLuint startTime[2]; // Start time buffers (A and B)

// Create and allocate buffers A and B for posBuf, velBuf
// and startTime
```

```
// Setup the feedback objects
glGenTransformFeedbacks(2, feedback);

// Transform feedback 0
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[0]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, startTime[0]);

// Transform feedback 1
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, posBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 1, velBuf[1]);
glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 2, startTime[1]);
```

Similar to vertex arrays, transform feedback objects store the buffer bindings for the `GL_TRANSFORM_FEEDBACK_BUFFER` binding point so that they can be reset quickly at a later time. In the previous code, we create two transform feedback objects, and store their handles in the array named `feedback`. For the first object, we bind `posBuf[0]` to index 0, `velBuf[0]` to index 1 and `startTime[0]` to index 2 of the binding point (buffer set A). These bindings are connected to the shader output variables with `glTransformFeedbackVaryings` (or via a layout qualifier, see the *There's More* following section). The last argument for each is the buffer's handle. For the second object, we do the same thing using the buffer set B.

Once this is set up, we can define the set of buffers to receive the vertex shader's output, by binding to one or the other transform feedback object.

The uniform variables that need to be set are the following:

- ▶ `ParticleTex`: It is the texture to apply to the point sprites.
- ▶ `Time`: It defines the simulation time.
- ▶ `H`: It defines the elapsed time between animation frames.
- ▶ `Accel`: It is used to define the acceleration.
- ▶ `ParticleLifetime`: It defines the length of time that a particle exists before it is recycled.

How to do it...

Use the following steps:

1. Use the following code for your vertex shader:

```
subroutine void RenderPassType();
subroutine uniform RenderPassTypeRenderPass;

layout (location = 0) in vec3 VertexPosition;
```

```
layout (location = 1) in vec3 VertexVelocity;
layout (location = 2) in float VertexStartTime;
layout (location = 3) in vec3 VertexInitialVelocity;

out vec3 Position;    // To transform feedback
out vec3 Velocity;   // To transform feedback
out float StartTime; // To transform feedback
out float Transp;    // To fragment shader

uniform float Time;   // Simulation time
uniform float H;      // Elapsed time between frames
uniform vec3 Accel;   // Particle acceleration
uniform float ParticleLifetime; // Particle lifespan

uniform mat4 MVP;

subroutine (RenderPassType)
void update() {

    Position = VertexPosition;
    Velocity = VertexVelocity;
    StartTime = VertexStartTime;

    if( Time >= StartTime ) {
        float age = Time - StartTime;
        if( age >ParticleLifetime ) {
            // The particle is past its lifetime, recycle.
            Position = vec3(0.0);
            Velocity = VertexInitialVelocity;
            StartTime = Time;
        } else {
            // The particle is alive, update.
            Position += Velocity * H;
            Velocity += Accel * H;
        }
    }
}

subroutine (RenderPassType)
void render() {
    float age = Time - VertexStartTime;
    Transp = 1.0 - age / ParticleLifetime;
    gl_Position = MVP * vec4(VertexPosition, 1.0);
}

void main()
{
    // This will call either render() or update()
    RenderPass();
}
```

2. Use the following code for the fragment shader:

```
uniform sampler2D ParticleTex;
in float Transp;
layout ( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = texture(ParticleTex, gl_PointCoord);
    FragColor.a *= Transp;
}
```

3. After compiling the shader program, but before linking, use the following code to set up the connection between vertex shader output variables and output buffers:

```
const char * outputNames[] = { "Position", "Velocity",
    "StartTime" };
glTransformFeedbackVaryings(progHandle, 3, outputNames,
    GL_SEPARATE_ATTRIBS);
```

4. In the OpenGL render function, send the particle positions to the vertex shader for updating, and capture the results using transform feedback. The input to the vertex shader will come from buffer A, and the output will be stored in buffer B. During this pass we enable `GL_RASTERIZER_DISCARD` so that nothing is actually rendered to the framebuffer:

```
// Select the subroutine for particle updating
glUniformSubroutinesiv(GL_VERTEX_SHADER, 1, &updateSub);

// Set the uniforms: H and Time
...

// Disable rendering
 glEnable(GL_RASTERIZER_DISCARD);

// Bind the feedback obj. for the buffers to be drawn
 glBindTransformFeedback(GL_TRANSFORM_FEEDBACK,
    feedback[drawBuf]);

// Draw points from input buffer with transform feedback
 glBeginTransformFeedback(GL_POINTS);
 glBindVertexArray(particleArray[1-drawBuf]);
 glDrawArrays(GL_POINTS, 0, nParticles);
 glEndTransformFeedback();
```

5. Render the particles at their updated positions using buffer B as input to the vertex shader:

```
// Enable rendering  
glDisable(GL_RASTERIZER_DISCARD);  
  
glUniformSubroutinesuiv(GL_VERTEX_SHADER, 1, &renderSub);  
glClear( GL_COLOR_BUFFER_BIT );  
  
// Initialize uniforms for transform matrices if needed  
...  
  
// Draw the sprites from the feedback buffer  
glBindVertexArray(particleArray[drawBuf]);  
glDrawTransformFeedback(GL_POINTS, feedback[drawBuf]);
```

6. Swap the purposes of the buffers:

```
// Swap buffers  
drawBuf = 1 - drawBuf;
```

How it works...

There's quite a bit here to sort through. Let's start with the vertex shader.

The vertex shader is broken up into two subroutine functions. The `update` function is used during the first pass, and uses Euler's method to update the position and velocity of the particle. The `render` function is used during the second pass. It computes the transparency based on the age of the particle and sends the position and transparency along to the fragment shader.

The vertex shader has four output variables. The first three: `Position`, `Velocity`, and `StartTime` are used in the first pass to write to the feedback buffers. The fourth (`Transp`) is used during the second pass as input to the fragment shader.

The `update` function just updates the particle position and velocity using Euler's method unless the particle is not alive yet, or has passed its lifetime. If its age is greater than the lifetime of a particle, we recycle the particle by resetting its position to the origin, updating the particle's start time to the current time (`Time`), and setting its velocity to its original initial velocity (provided via input attribute `VertexInitialVelocity`).

The `render` function computes the particle's age and uses it to determine the transparency of the particle, assigning the result to the output variable `Transp`. It transforms the particle's position into clip coordinates and places the result in the built-in output variable `gl_Position`.

The fragment shader (step 2) is only utilized during the second pass. It colors the fragment based on the texture `ParticleTex` and the transparency delivered from the vertex shader (`Transp`).

The code segment in step 3 is placed prior to linking the shader program is responsible for setting up the correspondence between shader output variables and feedback buffers (buffers that are bound to indices of the `GL_TRANSFORM_FEEDBACK_BUFFER` binding point). The function `glTransformFeedbackVaryings` takes three arguments. The first is the handle to the shader program object. The second is the number of output variable names that will be provided. The third is an array of output variable names. The order of the names in this list corresponds to the indices of the feedback buffers. In this case, `Position` corresponds to index zero, `Velocity` to index one, and `StartTime` to index two. Check back to the previous code that creates our feedback buffer objects (the `glBindBufferBase` calls) to verify that this is indeed the case.



`glTransformFeedbackVaryings` can be used to send data into an interleaved buffer instead (rather than separate buffers for each variable). Take a look at the OpenGL documentation for details.



The previous steps 4 through 6 describes how you might implement the render function within the main OpenGL program. In this example, there two important `GLuint` arrays: `feedback` and `particleArray`. They are each of size two and contain the handles to the two feedback buffer objects, and the two vertex array objects respectively. The variable `drawBuf` is just an integer used to alternate between the two sets of buffers. At any given frame, `drawBuf` will be either zero or one.

The code begins in step 4 by selecting the `update` subroutine to enable the update functionality within the vertex shader, and then setting the uniforms `Time` and `H`. The next call, `glEnable(GL_RASTERIZER_DISCARD)`, turns rasterization off so that nothing is rendered during this pass. The call to `glBindTransformFeedback` selects the set of buffers corresponding to the variable `drawBuf`, as the target for the transform feedback output.

Before drawing the points (and thereby triggering our vertex shader), we call `glBeginTransformFeedback` to enable transform feedback. The argument is the kind of primitives that will be sent down the pipeline (in our case `GL_POINTS`). Output from the vertex (or geometry) shader will go to the buffers that are bound to the `GL_TRANSFORM_FEEDBACK_BUFFER` binding point until `glEndTransformFeedback` is called. In this case, we bind the vertex array corresponding to `1 - drawBuf` (if `drawBuf` is 0, we use 1 and vice versa) and draw the particles.

At the end of the update pass (step 5), we re-enable rasterization with `glEnable(GL_RASTERIZER_DISCARD)`, and move on to the render pass.

The render pass is straightforward; we just select the render subroutine, and draw the particles from the vertex array corresponding to `drawBuf`. However, instead of using `glDrawArrays`, we use the function `glDrawTransformFeedback`. The latter is used here because it is designed for use with transform feedback. A transform feedback object keeps track of the number of vertices that were written. A call to `glDrawTransformFeedback` takes the feedback object as the third parameter. It uses the number of vertices that were written to that object as the number of vertices to draw. In essence it is equivalent to calling `glDrawArrays` with a value of zero for the second parameter and the count taken from the transform feedback.

Finally, at the end of the render pass (step 6), we swap our buffers by setting `drawBuf` to `1 - drawBuf`.

There's more...

You might be wondering why it was necessary to do this in two passes. Why couldn't we just keep the fragment shader active and do the render and update in the same pass? This is certainly possible for this example, and would be more efficient. However, I've chosen to demonstrate it this way because it is probably the more common way of doing this in general. Particles are usually just one part of a larger scene, and the particle update logic is not needed for most of the scene. Therefore, in most real-world situations it will make sense to do the particle update in a pass prior to the rendering pass so that the particle update logic can be decoupled from the rendering logic.

Using layout qualifiers

OpenGL 4.4 introduced layout qualifiers that make it possible to specify the relationship between the shader output variables and feedback buffers directly within the shader instead of using `glTransformFeedbackVaryings`. The layout qualifiers `xfb_buffer`, `xfb_stride`, and `xfb_offset` can be specified for each output variable that is to be used with transform feedback.

Querying transform feedback results

It is often useful to determine how many primitives were written during transform feedback pass. For example, if a geometry shader was active, the number of primitives written could be different than the number of primitives that were sent down the pipeline.

OpenGL provides a way to query for this information using query objects. To do so, start by creating a query object:

```
GLuint query;  
 glGenQueries(1, &query);
```

Then, prior to starting the transform feedback pass, start the counting process using the following command:

```
glBeginQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, query);
```

After the end of the transform feedback pass, call `glEndQuery` to stop counting:

```
glEndQuery(GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN);
```

Then we can get the number of primitives using the following code:

```
GLuint primWritten;
glGetQueryObjectui(query, GL_QUERY_RESULT, &primWritten);
printf("Primitives written: %d\n", primWritten);
```

Recycling particles

In this example, we recycled particles by resetting their position and initial velocity. This can cause the particles to begin to "clump" together over time. It would produce better results to generate a new random velocity and perhaps a random position (depending on the desired results). Unfortunately, there is currently no support for random number generation within shader programs. The solution might be to create your own random number generator function, use a texture with random values, or use a noise texture (see *Chapter 8, Using Noise in Shaders*).

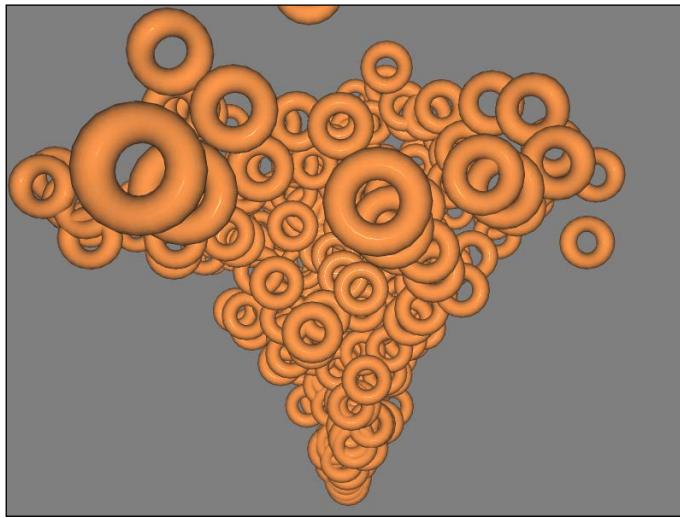
See also

- ▶ The *Creating a particle fountain* recipe

Creating a particle system using instanced particles

To give more geometric detail to each particle in a particle system, we can make use of OpenGL's support for **instanced rendering**. Instanced rendering is a convenient and efficient way to draw several copies of a particular object. OpenGL provides support for instanced rendering through the functions `glDrawArraysInstanced` and `glDrawElementsInstanced`.

In this example, we'll modify the particle system introduced in the previous recipes. Rather than using point sprites, we'll render a more complex object in the place of each particle. The following image shows an example where each particle is rendered as a shaded torus.



Using instanced rendering is simply a matter of calling one of the instanced draw functions, providing the number of instances to draw. However, there is some subtlety to the way that we provide vertex attributes to the shader. If all particles were drawn with exactly the same attributes, it would be simple to draw, but would hardly be an interesting result because all particles would appear at the same location and in the same orientation. Since we'd like to draw each copy in a different position, we need some way of providing the needed information (in our case, the particle's start time) to the vertex shader separately for each particle.

The key to this is the function `glVertexAttribDivisor`. This function specifies the rate at which vertex attributes are advanced during instanced rendering. For example, consider the following setting.

```
glVertexAttribDivisor(1, 1);
```

The first argument is the vertex attribute index, and the second is the number of instances that will pass between updates of the attribute. In other words, the previous command specifies that all vertices of the first instance will receive the first value in the buffer corresponding to attribute one. The second instance will receive the second value, and so on. If the second argument was 2, then the first two instances would receive the first value, the next two would receive the second, and so on in the same way.

The default divisor for each attribute is zero, which means that vertex attributes are processed normally (the attribute advances once per vertex rather than some number per instance). An attribute is called an **instanced attribute** if its divisor is non-zero.

Getting ready

Start with a particle system as described in *Creating a particle fountain*. We'll just make a few modifications to that basic system. Note that you can also use this with transform feedback if desired, but to keep things simple, we'll use the more basic particle system. It should be straightforward to adapt this example to the transform feedback based system.

When setting up the vertex array object for your particle shape, add two new instanced attributes for the initial velocity and start time. Something similar to the following code should do the trick:

```
glBindVertexArray(myVAO);

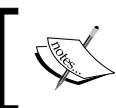
// Set up the pointers for attributes 0, 1, and 2 (position,
// normal, and texture coord.)
...

// Initial velocity (attribute 3)
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(3);
 glVertexAttribDivisor(3, 1);

// Start time (attribute 4)
glBindBuffer(GL_ARRAY_BUFFER, startTime);
glVertexAttribPointer(4, 1, GL_FLOAT, GL_FALSE, 0, NULL);
 glEnableVertexAttribArray(4);
 glVertexAttribDivisor(4, 1);

// Bind to the element array buffer if necessary
```

Note the use of `glVertexAttribDivisor` in the previous code. This indicates that attributes 3 and 4 are instanced attributes (the values in the arrays are to be advanced only once per instance, rather than once per vertex). Therefore the size of the buffers must be proportional to the number of instances rather than the number of vertices in an instance. The buffers for attributes 0, 1 and 2 should (as usual) be sized in relation to the number of vertices.



The value of the vertex attribute divisor becomes part of the vertex array object's state, so that just like the other elements of the VAO's state, we can reset it at a later point by binding to the VAO.

How to do it...

Use the following steps:

1. The vertex shader code is nearly identical to the code shown in the previous recipe *Creating a particle fountain*. The difference lies in the input and output variables. Use something similar to the following:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec3 VertexTexCoord;
layout (location = 3) in vec3 VertexInitialVelocity;
layout (location = 4) in float StartTime;

out vec3 Position;
out vec3 Normal;
```

2. Within the `main` function, update the position of the vertex by translating it using the equation of motion:

```
Position = VertexPosition + VertexInitialVelocity * t +
           Gravity * t * t;
```

3. Be sure to pass along the normal, and updated position (in camera coordinates) to the fragment shader.
4. In the fragment shader, implement your favorite shading model.
5. In the main OpenGL program, within the render function, render the instances using the following code:

```
glBindVertexArray(myVArray);
glDrawElementsInstanced(GL_TRIANGLES, nEls,
                       GL_UNSIGNED_INT, 0, nParticles);
```

How it works...

Recall that the first three input attributes to the vertex shader are not-instanced, meaning that they are advanced every vertex (and repeated every instance). The last two (attributes 3 and 4) are instanced attributes and only update every instance. Therefore, the effect is that each instance is translated by the result of the equation of motion.

The `glDrawElementsInstanced` function (step 5) will draw `nParticles` instances of the object. Of course `nEls` is the number of vertices in each instance.

There's more...

OpenGL provides a built-in variable to the vertex shader named `gl_InstanceID`. This is simply a counter and takes on a different value for each instance that is rendered. The first instance will have an ID of zero, the second will have an ID of one, and so on. This can be useful as a way to index to texture data appropriate for each instance. Another possibility is to use the instance's ID as a way to generate some random data for that instance. For example, we could use the instance ID (or some hash) as a seed to a pseudo-random number generation routine to get a unique random stream for each instance.

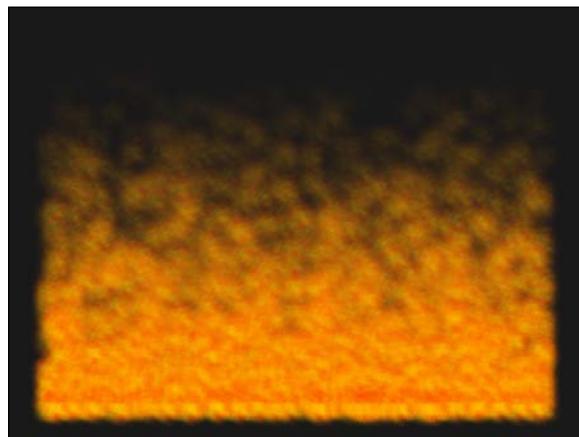
See also

- ▶ The *Creating a particle fountain* recipe
- ▶ The *Creating a particle system using transform feedback* recipe

Simulating fire with particles

To create an effect that roughly simulates fire, we only need to make a few changes to our basic particle system. Since fire is a substance that is only slightly affected by gravity, we don't worry about a downward gravitational acceleration. In fact, we'll actually use a slight upwards acceleration to make the particles spread out near the top of the flame. We'll also spread out the initial positions of the particles so that the base of the flame is not just a single point. Of course, we'll need to use a particle texture that has the red and orange colors associated with flame.

The following image shows an example of the running particle system:



The texture that was used for the particles looks like a light "smudge" of the flame's colors. It is not shown here because it would not be very visible in print.

Getting ready

Start with the basic particle system presented in the recipe *Creating a particle system using transform feedback* earlier in this chapter.

Set the uniform variable `Accel` to a small upward value like (0.0, 0.1, 0.0).

Set the uniform variable `ParticleLifetime` to about four seconds.

Create and load a texture for the particles that has fire-like colors. Bind it to the first texture channel, and set the uniform `ParticleTex` to zero.

Use a point size of about 50.0. This is a good size for the texture that is used in this recipe, but you might use a different size depending on the number of particles and the texture.

How to do it...

Use the following steps:

1. When setting up the initial positions for your particles, instead of using the origin for all particles, use a random x location. The following code could be used:

```
GLfloat *data = new GLfloat[nParticles * 3];
for( int i = 0; i < nParticles * 3; i += 3 ) {
    data[i] = glm::mix(-2.0f, 2.0f, randFloat());
    data[i+1] = 0.0f;
    data[i+2] = 0.0f;
}
glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
```

2. When setting up the initial velocities, we'll make sure that the x and z components are zero and the y component contains a random speed. This, combined with the chosen acceleration (see the previous code) makes each particle move in only the y (vertical) direction:

```
// Fill the first velocity buffer with random velocities
for( unsigned int i = 0; i < nParticles; i++ ) {
    data[3*i]      = 0.0f;
    data[3*i+1]    = glm::mix(0.1f, 0.5f, randFloat());
    data[3*i+2]    = 0.0f;
}
glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
```

3. In the vertex shader, when recycling particles, reset the y and z coordinates, but don't change the x coordinate:

```
if( age > ParticleLifetime ) {  
    // The particle is past its lifetime, recycle.  
    Position = vec3(VertexPosition.x, 0.0, 0.0);  
    Velocity = VertexInitialVelocity;  
    StartTime = Time;  
}
```

How it works...

We randomly distribute the x-coordinate of the initial positions between -2.0 and 2.0 for all of the particles, and set the initial velocities to have a y-coordinate between 0.1 and 0.5. Since the acceleration has only a y-component, the particles will move only along a straight, vertical line in the y direction. The x or z component of the position should always remain at zero. This way, when recycling the particles, we can simply just reset the y coordinate to zero, to restart the particle at its initial position.

There's more...

Of course, if you want a flame that moves in different directions, perhaps blown in the wind, you'd need to use a different value for the acceleration. In which case, our little trick for resetting particles to their initial position will no longer work. However, we only need to add another buffer to our particle system (similar to the initial velocity buffer) to maintain the initial position and re-use it when recycling particles.

See also

- ▶ The *Creating a particle system using transform feedback* recipe

Simulating smoke with particles

Smoke is characterized by many small particles that float away from the source, and spread out as they move through the air. We can simulate the floatation effect with particles by using a small upwards acceleration (or constant velocity), but simulating the diffusion of each small smoke particle would be too expensive. Instead, we can simulate the diffusion of many small particles by making our simulated particles change their size (grow) over time.

The following image shows an example of the results:



The texture for each particle is a very light "smudge" of grey or black color.

To make the particles grow over time, we'll make use of the `GL_PROGRAM_POINT_SIZE` functionality in OpenGL, which allows us to modify the point size within the vertex shader.



Alternatively, we could draw quads, or use the geometry shader to generate the quads using the technique demonstrated in *Chapter 6, Using Geometry and Tessellation Shaders*.

Getting ready

Start with the basic particle system presented in the recipe *Creating a particle system using transform feedback*.

Set the uniform variable `Accel` to a small upward value like (0.0, 0.1, 0.0).

Set the uniform variable `ParticleLifetime` to about six seconds.

Create and load a texture for the particles that looks like just light grey smudge. Bind it to texture unit zero, and set the uniform `ParticleTex` to zero.

Set the uniform variables `MinParticleSize` and `MaxParticleSize` to 10 and 200 respectively.

How to do it...

Use the following steps:

1. Set the initial positions to the origin. Define the initial velocities in the same way as described in the recipe *Creating a particle system using transform feedback*. However, it looks best when you use a large variance in theta.

2. Within the vertex shader, add the following uniforms:

```
uniform float MinParticleSize;  
uniform float MaxParticleSize;
```

3. Also within the vertex shader, use the following code for the `render` function:

```
subroutine (RenderPassType)  
void render() {  
    float age = Time - VertexStartTime;  
    Transp = 0.0;  
    if( Time >= VertexStartTime ) {  
        float agePct = age/ParticleLifetime;  
        Transp = 1.0 - agePct;  
        gl_PointSize =  
            mix(MinParticleSize,MaxParticleSize,agePct);  
    }  
    gl_Position = MVP * vec4(VertexPosition, 1.0);  
}
```

4. In the main OpenGL application, before rendering your particles, make sure to enable `GL_PROGRAM_POINT_SIZE`:

```
glEnable(GL_PROGRAM_POINT_SIZE);
```

How it works...

The `render` subroutine function sets the built-in variable `gl_PointSize` to a value between `MinParticleSize` and `MaxParticleSize`, determined by the age of the particle. This causes the size of the particles to grow as they evolve through the system.

Note that the variable `gl_PointSize` is ignored by OpenGL unless `GL_PROGRAM_POINT_SIZE` is enabled.

See also

- ▶ The *Creating a particle system using transform feedback* recipe

10

Using Compute Shaders

In this chapter, we will cover the following recipes:

- ▶ Implementing a particle simulation with the compute shader
- ▶ Using the compute shader for cloth simulation
- ▶ Implementing an edge detection filter with the compute shader
- ▶ Creating a fractal texture using the compute shader

Introduction

Compute shaders were introduced into OpenGL with Version 4.3. A compute shader is a shader stage that can be used for arbitrary computation. It provides the ability to leverage the GPU and its inherent parallelism for general computing tasks that might have previously been implemented in serial on the CPU. The compute shader is most useful for tasks that are not directly related to rendering, such as physical simulation.



Although APIs such as OpenCL and CUDA are already available for general purpose computation on the GPU, they are completely separate from OpenGL. Compute shaders are integrated directly within OpenGL, and therefore are more suitable for general computing tasks that are more closely related to graphics rendering.

The compute shader is not a traditional shader stage in the same sense as the fragment or vertex shader. It is not executed in response to rendering commands. In fact, when a compute shader is linked with vertex, fragment, or other shader stages, it is effectively inert when drawing commands are executed. The only way to execute the compute shader is via the OpenGL commands `glDispatchCompute` or `glDispatchComputeIndirect`.

Compute shaders do not have any direct user-defined inputs and no outputs at all. It gets its work by fetching data directly from memory using image access functions such as the image load/store operations, or via shader storage buffer objects. Similarly, it provides its results by writing to the same or other objects. The only non-user-defined inputs to a compute shader are a set of variables that determine where the shader invocation is within its "space" of execution.

The number of invocations of the compute shader is completely user defined. It is not tied in any way to the number of vertices or fragments being rendered. We specify the number of invocations by defining the number of work groups, and the number of invocations within each work group.

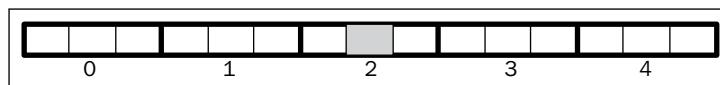
Compute space and work groups

The number of invocations of a compute shader is governed by the user-defined compute space. This space is divided into a number of work groups. Each work group is then broken down into a number of invocations. We think of this in terms of the global compute space (all shader invocations) and the local work group space (the invocations within a particular work group). The compute space can be defined as a 1, 2 or 3 dimensional space.



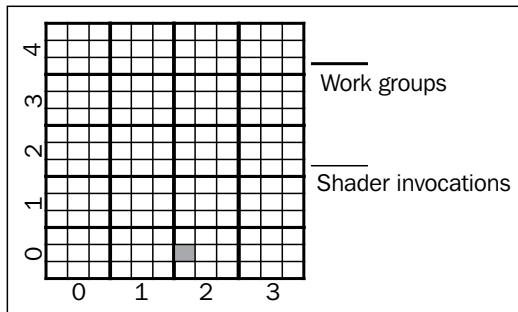
Technically, it is always defined as a three-dimensional space, but any of the three dimensions can be defined with a size of one (1), which effectively removes that dimension.

For example, a one-dimensional compute space with five work groups and three invocations per work group could be represented as the following figure. The thicker lines represent the work groups, and the thinner lines represent the invocations within each work group.



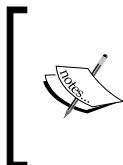
In this case, we have $5 * 3 = 15$ shader invocations. The grey shaded invocation is in work group 2, and within that work group, it is invocation 1 (the invocations are indexed starting at zero). We can also refer to that invocation with a global index of 7, by indexing the total number of invocations starting at zero. The global index determines an invocation's location within the global compute space, rather than just within the work group. It is determined by taking the product of work group (2) and index number of invocations per work group (3), plus the local invocation index (1) that is $2 * 3 + 1 = 7$. The global index is simply the index of each invocation in the global compute space, starting at zero on the left and counting from there.

The following figure shows a representation of a two-dimensional compute space where the space is divided into 20 work groups, four in the x direction and five in the y direction. Each work group is then divided into nine invocations, three in the x direction and three in the y direction.



The cell that is shaded in gray represents invocation (0, 1) within work group (2, 0). The total number of compute shader invocations in this example is then $20 * 9 = 180$. The global index of this shaded invocation is (6, 1). As with the one-dimensional case, this index is determined by thinking of it as a global compute space (without the work groups), and can be computed (for each dimension) by the number of invocations per work group times the work group index, plus the local invocation index. For the x dimension, this would be $3 * 2 + 0 = 6$, and for the y dimension it is $3 * 0 + 1 = 1$.

The same idea can extend in a straightforward manner to a three-dimensional compute space. In general, we choose the dimensionality based on the data to be processed. For example, if I'm working on the physics of a particle simulation, I would just have a list of particles to process, so a one-dimensional compute space might make sense. On the other hand, if I'm processing a cloth simulation, the data has a grid structure, so a two-dimensional compute space would be appropriate.



There are limits on the total number of work groups and local shader invocations. These can be queried (via `glGetInteger*`) using the parameters `GL_MAX_COMPUTE_WORK_GROUP_COUNT`, `GL_MAX_COMPUTE_WORK_GROUP_SIZE`, and `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`.

The order of execution of the work groups and thereby the individual shader invocations is unspecified and the system can execute them in any order. Therefore, we shouldn't rely on any particular ordering of the work groups. Local invocations within a particular work group will be executed in parallel (if possible). Therefore, any communication between invocations should be done with great care. Invocations within a work group can communicate via shared local data, but invocations should not (in general) communicate with invocations in other work groups without consideration of the various pitfalls involved such as deadlock and data races. In fact, those can also be issues for local shared data within a work group as well, and care must be taken to avoid these problems. In general, for reasons of efficiency, it is best to only attempt communication within a work group. As with any kind of parallel programming, "there be dragons here".

OpenGL provides a number of atomic operations and memory barriers that can help with the communication between invocations. We'll see some examples in the recipes that follow.

Executing the Compute Shader

When we execute the compute shader, we define the compute space. The number of work groups are determined by the parameters to `glDispatchCompute`. For example, to execute the compute shader with a two-dimensional compute space with 4 work groups in the x dimension and 5 work groups in the y dimension (matching the preceding figure), we'd use the following call:

```
glDispatchCompute( 4, 5, 1 );
```

The number of local invocations within each work group is not specified on the OpenGL side. Instead, it is specified within the compute shader itself with a layout specifier. For example, here we specify nine local invocations per work group, 3 in the x direction and 3 in the y direction.

```
layout (local_size_x = 3, local_size_y = 3) in;
```

The size in the z dimension can be left out (the default is one).

When a particular invocation of the compute shader is executing, it usually needs to determine where it is within the global compute space. GLSL provides a number of built-in input variables that help with this. Most of them are listed in the following table:

Variable	Type	Meaning
<code>gl_WorkGroupSize</code>	<code>uvec3</code>	The number of invocations per work group in each dimension. Same as what is defined in the layout specifier.
<code>gl_NumWorkGroups</code>	<code>uvec3</code>	The total number of work groups in each dimension.
<code>gl_WorkGroupID</code>	<code>uvec3</code>	The index of the current work group for this shader invocation.
<code>gl_LocalInvocationID</code>	<code>uvec3</code>	The index of the current invocation within the current work group.
<code>gl_GlobalInvocationID</code>	<code>uvec3</code>	The index of the current invocation within the global compute space.

The last one in the preceding table, `gl_GlobalInvocationID` is computed in the following way (each operation is component-wise):

```
gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID
```

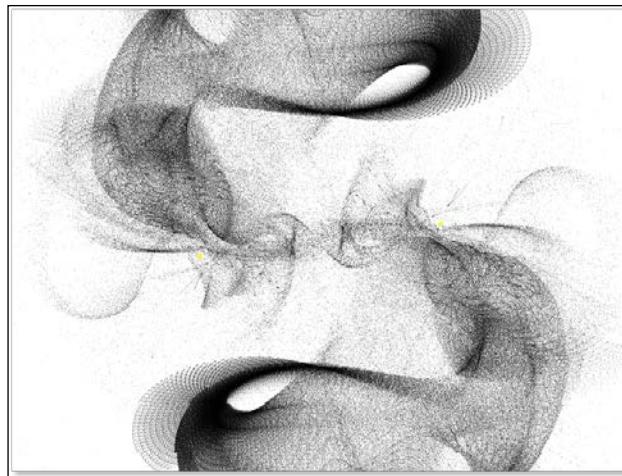
This helps us to locate the current invocation within the global compute space (Refer to the preceding examples).

GLSL also defines `gl_LocalInvocationIndex`, which is a flattened form of `gl_LocalInvocationID`. It can help when multidimensional data is provided in a linear buffer, but is not used in any of the examples that follow.

Implementing a particle simulation with the compute shader

In this recipe, we'll implement a simple particle simulation. We'll have the compute shader handle the physics computations and update the particle positions directly. Then, we'll just render the particles as points. Without the compute shader, we'd need to update the positions on the CPU by stepping through the array of particles and updating each position in a serial fashion, or by making use of transform feedback as shown in the *Creating a particle system using transform feedback* recipe in *Chapter 9, Particle Systems and Animation*. Doing such animations with vertex shaders is sometimes counterintuitive and requires some additional work (such as transform feedback setup). With the compute shader, we can do the particle physics in parallel on the GPU, and customize our compute space to get the most "bang for the buck" out of our GPU.

The following figure shows our particle simulation running with one million particles. Each particle is rendered as a 1×1 point. The particles are partially transparent, and the particle attractors are rendered as small 5×5 squares (barely visible).



These simulations can create beautiful, abstract figures, and are a lot of fun to produce.

For our simulation, we'll define a set of attractors (two in this case, but you can create more), which I'll call the **black holes**. They will be the only objects that affect our particles and they'll apply a force on each particle that is inversely proportional to the distance between the particle and the black hole. More formally, the force on each particle will be determined by the following equation:

$$\mathbf{F} = \sum_{i=1}^N \frac{G_i}{|\mathbf{r}_i|} \frac{\mathbf{r}_i}{|\mathbf{r}_i|}$$

Where N is the number of black holes (attractors), \mathbf{r}_i is the vector between the i th attractor and the particle (determined by the position of the attractor minus the particle position), and G_i is the strength of the i th attractor.

To implement the simulation, we compute the force on each particle and then update the position by integrating the Newtonian equations of motion. There are a number of well studied numerical techniques for integrating the equations of motion. For this simulation, the simple Euler method is sufficient. With the Euler method, the position of the particle at time $t + \Delta t$ is given by the following equation:

$$\mathbf{P}(t + \Delta t) = \mathbf{P}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2$$

Where \mathbf{P} is the position of the particle, \mathbf{v} is the velocity, and \mathbf{a} is the acceleration. Similarly, the updated velocity is determined by the following equation:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

These equations are derived from a Taylor expansion of the position function about time t . The result is dependent upon the size of the time step (Δt), and is more accurate when the time step is very small.

The acceleration is directly proportional to the force on the particle, so by calculating the force on the particle (using the preceding equation), we essentially have a value for the acceleration. To simulate the particle's motion, we track its position and velocity, determine the force on the particle due to the black holes, and then update the position and velocity using the equations.

We'll use the compute shader to implement the physics here. Since we're just working with a list of particles, we'll use a one-dimensional compute space, and work groups of about 1000 particles each. Each invocation of the compute shader will be responsible for updating the position of a single particle.

We'll use shader storage buffer objects to track the positions and velocities, and when rendering the particles themselves, we can just render directly from the position buffer.

Getting ready

In the OpenGL side, we need a buffer for the position of the particles and a buffer for the velocity. Create a buffer containing the initial positions of the particles and a buffer with zeroes for the initial velocities. We'll use four-component positions and velocities for this example in order to avoid issues with data layouts. For example, to create the buffer for the positions, we might do something as follows:

```
vector<GLfloat> initPos;  
  
... // Set initial positions  
  
GLuint bufSize = totalParticles * 4 * sizeof(GLfloat);  
  
GLuint posBuf;  
glGenBuffers(1, &posBuf);  
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, posBuf);  
glBufferData(GL_SHADER_STORAGE_BUFFER, bufSize, &initPos[0],  
             GL_DYNAMIC_DRAW);
```

Use a similar process for the velocity data, but bind it to index one of the `GL_SHADER_STORAGE_BUFFER` binding location.

```
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, velBuf);
```

Set up a vertex array object that uses the same position buffer as its data source for the vertex position.

To render the points, set up a vertex and fragment shader pair that just produces a solid color. Enable blending and set up a standard blending function.

How to do it...

Use the following steps:

1. We'll use the compute shader for updating the positions of the particles.

```
layout( local_size_x = 1000 ) in;  
  
uniform float Gravity1 = 1000.0;  
uniform vec3 BlackHolePos1;  
uniform float Gravity2 = 1000.0;  
uniform vec3 BlackHolePos2;  
  
uniform float ParticleInvMass = 1.0 / 0.1;  
uniform float DeltaT = 0.0005;
```

```
layout(std430, binding=0) buffer Pos {
    vec4 Position[];
};

layout(std430, binding=1) buffer Vel {
    vec4 Velocity[];
};

void main() {
    uint idx = gl_GlobalInvocationID.x;

    vec3 p = Position[idx].xyz;
    vec3 v = Velocity[idx].xyz;

    // Force from black hole #1
    vec3 d = BlackHolePos1 - p;
    vec3 force = (Gravity1 / length(d)) * normalize(d);

    // Force from black hole #2
    d = BlackHolePos2 - p;
    force += (Gravity2 / length(d)) * normalize(d);

    // Apply simple Euler integrator
    vec3 a = force * ParticleInvMass;
    Position[idx] = vec4(
        p + v * DeltaT + 0.5 * a * DeltaT * DeltaT, 1.0);
    Velocity[idx] = vec4( v + a * DeltaT, 0.0);
}
```

2. In the render routine, invoke the compute shader to update the particle positions.

```
glDispatchCompute(totalParticles / 1000, 1, 1);
```

3. Then make sure that all data has been written out to the buffer, by invoking a memory barrier.

```
glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );
```

4. Finally, render the particles using data in the position buffer.

How it works...

The compute shader starts by defining the number of invocations per work group using the layout specifier.

```
layout( local_size_x = 1000 ) in;
```

This specifies 1000 invocations per work group in the x dimension. You can choose a value for this that makes the most sense for the hardware on which you're running. Just make sure to adjust the number of work groups appropriately. The default size for each dimension is one so we don't need to specify the size of the y and z directions.

Then, we have a set of uniform variables that define the simulation parameters. `Gravity1` and `Gravity2` are the strengths of the two black holes (G in the above equation), and `BlackHolePos1` and `BlackHolePos2` are their positions. `ParticleInvMass` is the inverse of the mass of each particle, which is used to convert force to acceleration. Finally, `DeltaT` is the time-step size, used in the Euler method for integration of the equations of motion.

The buffers for position and velocity are declared next. Note that the binding values here match those that we used on the OpenGL side when initializing the buffers.

Within the main function, we start by determining the index of the particle for which this invocation is responsible. Since we're working with a linear list of particles, and the number of particles is the same as the number of shader invocations, what we want is the index within the global range of invocations. This index is available via the built-in input variable `gl_GlobalInvocationID.x`. We use the global index here because it is the index within the entire buffer that we need, not the index within our work group, which would only reference a portion of the entire array.

Next we retrieve the position and velocity from their buffers, and compute the force due to each black hole, storing the sum in the variable `force`. Then we convert the force to acceleration and update the particle's position and velocity using the Euler method. We write to the same location from which we read previously. Since invocations do not share data, this is safe.

In the render routine, we invoke the compute shader (step 2 in *How to do it...*), defining the number of work groups per dimension. In the compute shader, we specified a work group size of 1000. Since we want one invocation per particle, we divide the total number of particles by 1000 to determine the number of work groups.

Finally, in step 3, before rendering the particles, we need to invoke a memory barrier to ensure that all compute shader writes have fully executed.

See also

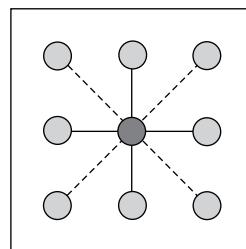
- ▶ Refer to *Chapter 9, Particle Systems and Animation*, for other particle simulations. Most of these have been implemented using transform feedback, but could instead be implemented using the compute shader.

Using the compute shader for cloth simulation

The compute shader is well suited for harnessing the GPU for physical simulation. Cloth simulation is a prime example. In this recipe, we'll implement a simple particle-spring based cloth simulation using the compute shader. The following is a screenshot of the simulation of a cloth hanging by five pins. (You'll have to imagine it animating.)



A common way to represent cloth is with a particle-spring lattice. The cloth is composed of a 2D grid of point masses, each connected to its eight neighboring masses with idealized springs. The following figure represents one of the point masses (center) connected to its neighboring masses. The lines represent the springs. The dark lines are the horizontal/vertical springs and the dashed lines are the diagonal springs.



The total force on a particle is the sum of the forces produced by the eight springs to which it is connected. The force for a single spring is given by the following equation:

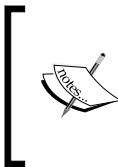
$$\mathbf{F} = K(|\mathbf{r}| - R) \frac{\mathbf{r}}{|\mathbf{r}|}$$

Where K is the stiffness of the spring, R is the rest-length of the spring (the length where the spring applies zero force), and \mathbf{r} is the vector between the neighboring particle and the particle (the neighbor's position minus the particle's position).

Similar to the previous recipe, the process is simply to compute the total force on each particle and then integrate Newton's equations of motion using our favorite integration. Again, we'll use the Euler method for this example. For details on the Euler method, refer to the previous recipe, *Implementing a particle simulation with the compute shader*.

This particle-spring lattice is obviously a two-dimensional structure, so it makes sense to map it to a two-dimensional compute space. We'll define rectangular work groups and use one shader invocation per particle. Each invocation needs to read the positions of its eight neighbors, compute the force on the particle, and update the particle's position and velocity.

Note that in this case, each invocation needs to read the positions of the neighboring particles. Those neighboring particles will be updated by other shader invocations. Since we can't rely on any execution order for the shader invocations, we can't read and write directly to the same buffer. If we were to do so, we wouldn't know for sure whether we are reading the original positions of the neighbors or their updated positions. To avoid this problem, we'll use pairs of buffers. For each simulation step, one buffer will be designated for reading and the other for writing, then we'll swap them for the next step, and repeat.



It might be possible to read/write to the same buffer with careful use of local shared memory; however, there is still the issue of the particles along the edges of the work group. Their neighbor's positions are managed by another work group, and again, we have the same problem.

This simulation tends to be quite sensitive to numerical noise, so we need to use a very small integration time step. A value of around 0.000005 works well. Additionally, the simulation looks better when we apply a damping force to simulate air resistance. A good way to simulate air resistance is to add a force that is proportional to and in the opposite direction to the velocity, as in the following equation:

$$\mathbf{F} = -D\mathbf{v}$$

Where D is the strength of the damping force and \mathbf{v} is the velocity of the particle.

Getting ready

Start by setting up two buffers for the particle position and two for the particle velocity. We'll bind them to the `GL_SHADER_STORAGE_BUFFER` indexed binding point at indices 0 and 1 for the position buffers and 2 and 3 for the velocity buffers. The data layout in these buffers is important. We'll lay out the particle positions/velocities in row-major order starting at the lower left and proceeding to the upper right of the lattice.

We'll also set up a vertex array object for drawing the cloth using the particle positions as triangle vertices. We may also need buffers for normal vectors and texture coordinates. For brevity, I'll omit them from this discussion, but the example code for this book includes them.

How to do it...

Use the following steps:

1. In the compute shader, we start by defining the number of invocations per work group.

```
layout( local_size_x = 10, local_size_y = 10 ) in;
```

2. Then, define a set of uniform variables for the simulation parameters.

```
uniform vec3 Gravity = vec3(0, -10, 0);
uniform float ParticleMass = 0.1;
uniform float ParticleInvMass = 1.0 / 0.1;
uniform float SpringK = 2000.0;
uniform float RestLengthHoriz;
uniform float RestLengthVert;
uniform float RestLengthDiag;
uniform float DeltaT = 0.000005;
uniform float DampingConst = 0.1;
```

3. Next, declare the shader storage buffer pairs for the position and velocity.

```
layout(std430, binding=0) buffer PosIn {
    vec4 PositionIn[];
};

layout(std430, binding=1) buffer PosOut {
    vec4 PositionOut[];
};

layout(std430, binding=2) buffer VelIn {
    vec4 VelocityIn[];
};

layout(std430, binding=3) buffer VelOut {
    vec4 VelocityOut[];
};
```

4. In the main function, we get the position of the particle for which this invocation is responsible.

```
void main() {
    uvec3 nParticles = gl_NumWorkGroups * gl_WorkGroupSize;
    uint idx = gl_GlobalInvocationID.y * nParticles.x +
               gl_GlobalInvocationID.x;

    vec3 p = vec3(PositionIn[idx]);
    vec3 v = vec3(VelocityIn[idx]), r;
```

5. Initialize our force with the force due to gravity.

```
vec3 force = Gravity * ParticleMass;
```

6. Add the force due to the particle above this one.

```
if( gl_GlobalInvocationID.y < nParticles.y - 1 ) {
    r = PositionIn[idx + nParticles.x].xyz - p;
    force += normalize(r)*SpringK*(length(r) -
                                    RestLengthVert);
}
```

7. Repeat the preceding steps for the particles below and to the left and right. Then add the force due to the particle that is diagonally above and to the left.

```
if( gl_GlobalInvocationID.x > 0 &&
    gl_GlobalInvocationID.y < nParticles.y - 1 ) {
    r = PositionIn[idx + nParticles.x - 1].xyz - p;
    force += normalize(r)*SpringK*(length(r) -
                                    RestLengthDiag);
}
```

8. Repeat the above for the other three diagonally connected particles. Then add the damping force.

```
force += -DampingConst * v;
```

9. Next, we integrate the equations of motion using the Euler method.

```
vec3 a = force * ParticleInvMass;
PositionOut[idx] = vec4(
    p + v * DeltaT + 0.5 * a * DeltaT * DeltaT, 1.0);
VelocityOut[idx] = vec4( v + a * DeltaT, 0.0);
```

10. Finally, we pin some of the top verts so that they do not move.

```
if( gl_GlobalInvocationID.y == nParticles.y - 1 &&
    (gl_GlobalInvocationID.x == 0 ||
     gl_GlobalInvocationID.x == nParticles.x / 4 ||
     gl_GlobalInvocationID.x == nParticles.x * 2 / 4 ||
```

```
    gl_GlobalInvocationID.x == nParticles.x * 3 / 4 ||  
    gl_GlobalInvocationID.x == nParticles.x - 1)) {  
    PositionOut[idx] = vec4(p, 1.0);  
    VelocityOut[idx] = vec4(0,0,0,0);  
}  
}
```

11. Within the OpenGL render function, we invoke the compute shader such that each work group is responsible for 100 particles. Since the time step size is so small, we need to execute the process many times (1000), each time swapping the input and output buffers.

```
for( int i = 0; i < 1000; i++ ) {  
    glDispatchCompute(nParticles.x/10, nParticles.y/10, 1);  
    glMemoryBarrier( GL_SHADER_STORAGE_BARRIER_BIT );  
  
    // Swap buffers  
    readBuf = 1 - readBuf;  
  
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0,  
                    posBufs[readBuf]);  
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1,  
                    posBufs[1-readBuf]);  
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2,  
                    velBufs[readBuf]);  
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3,  
                    velBufs[1-readBuf]);  
}
```

12. Finally, we render the cloth using the position data from the position buffer.

How it works...

We use 100 invocations per work group, 10 in each dimension. The first statement in the compute shader defines the number of invocations per work group.

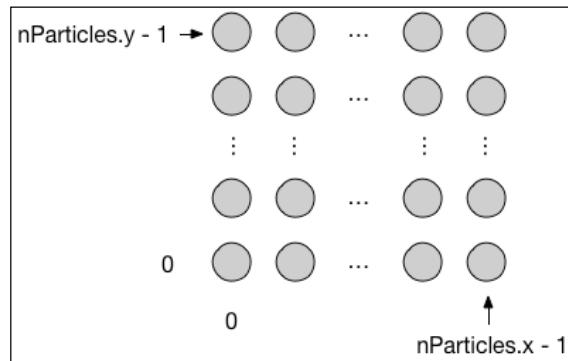
```
layout( local_size_x = 10, local_size_y = 10 ) in;
```

The uniform variables that follow define the constants in the force equations and the rest lengths for each of the horizontal, vertical, and diagonal springs. The time step size is DeltaT. The position and velocity buffers are declared next. We define the position buffers at binding indexes 0 and 1, and the velocity buffers at indexes 2 and 3.

In the main function (step 4), we start by determining the number of particles in each dimension. This is going to be the same as the number of work groups times the work group size. Next, we determine the index of the particle for which this invocation is responsible. Since the particles are organized in the buffers in row-major order, we compute the index by the global invocation ID in the y direction times the number of particles in the x dimension plus the global invocation ID in the x direction.

In step 5, we initialize our force with the gravitational force, `Gravity` times the mass of a particle (`ParticleMass`). Note that it's not really necessary here to multiply by the mass since all particles have the same mass. We could just pre-multiply the mass into the gravitational constant.

In steps 6 and 7, we add the force on this particle due to each of the eight neighboring particles connected by virtual springs. For each spring, we add the force due to that spring. However, we first need to check to see if we are on the edge of the lattice. If we are, there may not be a neighboring particle (see the following figure). For example, in the preceding code, when computing the force due to the spring/particle above, we verify that `g1_GlobalInvocationID.y` is less than the number of particles in the y dimension minus one. If that is true, there must be a particle above this one. Otherwise, the current particle is on the top edge of the lattice and there is no neighboring particle above. (Essentially, `g1_GlobalInvocationID` contains the particle's location in the overall lattice.) We can do a similar test for the other three horizontal/vertical directions. When computing the force for the diagonally connected particles we need to check that we're not on a horizontal and a vertical edge. For example, in the preceding code, we're looking for the particle that is above and to the left, so we check that `g1_GlobalInvocationID.x` is greater than zero (we're not on the left edge), and that `g1_GlobalInvocationID.y` is less than the number of particles in the y direction minus one (we're not on the top edge).



Once we verify that the neighboring particle exists, we compute the force due to the spring connected to that particle and add it to the total force. We organized our particles in row-major order in the buffer. Therefore, to access the position of the neighboring particle we take the index of the current particle and add/subtract the number of particles in the x direction to move vertically, and/or add/subtract one to move horizontally.

In step 8, we apply the damping force that simulates air resistance, by adding to the total force `DampingConst` times the velocity. The minus sign here assures that the force is in the opposite direction of the velocity.

In step 9 we apply the Euler method to update the position and velocity based on the force. We multiply the force by the inverse of the particle mass to get the acceleration, then store the results of the Euler integration into the corresponding positions in the output buffers.

Finally, in step 10, we reset the position of the particle if it is located at one of the 5 pin positions at the top of the cloth.

Within the OpenGL render function (step 11), we invoke the compute shader multiple times, switching the input/output buffers after each invocation. After calling `glDispatchCompute`, we issue a `glMemoryBarrier` call to make sure that all shader writes have completed before swapping the buffers. Once that is complete, we go ahead and render the cloth using the positions from the shader storage buffer.

There's more...

For rendering, it is useful to have normal vectors. One option is to create another compute shader to recalculate the normal vectors after the positions are updated. For example, we might execute the preceding compute shader 1000 times, dispatch the other compute shader once to update the normals, and then render the cloth.

Additionally, we may be able to achieve better performance with the use of local shared data within the work group. In the above implementation the position of each particle is read a maximum of eight times. Each read can be costly in terms of execution time. It is faster to read from memory that is closer to the GPU. One way to achieve this is to read data into local shared memory once, and then read from the shared memory for subsequent reads. In the next recipe, we'll see an example of how this is done. It would be straightforward to update this recipe in a similar way.

See also

- ▶ The *Implementing an edge detection filter with the compute shader* recipe

Implementing an edge detection filter with the compute shader

In the *Applying an edge detection filter* recipe in *Chapter 5, Image Processing and Screen Space Techniques*, we saw an example of how to implement edge detection using the fragment shader. The fragment shader is well suited for many image processing operations, because we can trigger execution of the fragment shader for each pixel by rendering a screen-filling quad. Since image processing filters are often applied to the result of a render, we can render to a texture, then invoke the fragment shader for each screen pixel (by rendering a quad), and each fragment shader invocation is then responsible for processing a single pixel. Each invocation might need to read from several locations in the (rendered) image texture, and a texel might be read multiple times from different invocations.

This works well for many situations, but the fragment shader was not designed for image processing. With the compute shader, we can have more fine grained control over the distribution of shader invocations, and we can make use of local shared memory to gain a bit more efficiency with data reads.

In this example, we'll re-implement the edge detection filter using the compute shader. We'll make use of local (work group) shared memory to gain additional speed. Since this local memory is closer to the GPU, memory accesses are faster than they would be when reading directly from the shader storage buffers (or textures).

As with the previous recipe, we'll implement this using the Sobel operator, which is made up of two 3×3 filter kernels shown as follows:

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \mathbf{S}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

For details on the Sobel operator, please refer to *Chapter 5, Image Processing and Screen Space Techniques*. The key point here is that in order to compute the result for a given pixel, we need to read the values of the eight neighboring pixels. This means that the value of each pixel needs to be fetched up to eight times (when processing the neighbors of that pixel). To gain some additional speed, we'll copy the needed data into local shared memory so that within a work group, we can read from the shared memory rather than fetching from the shader storage buffer.



Work group shared memory is generally faster to access than texture or shader storage memory.

In this example, we'll use one compute shader invocation per pixel, and a 2D work group size of 25 x 25. Before computing the Sobel operator, we'll copy the corresponding pixel values into local shared memory for the work group. For each pixel, in order to compute the filter, we need to read the values of the eight neighboring pixels. In order to do so for the pixels on the edge of the work group, we need to include in our local memory an extra strip of pixels outside the edges of the work group. Therefore, for a work group size of 25 x 25, we'll need storage of size 27 x 27.

Getting ready

Start by setting up for rendering to a **Framebuffer Object (FBO)** with a color texture attached, we'll render the raw pre-filtered image to this texture. Create a second texture to receive the output from the edge detection filter. Bind this latter texture to unit 0. We'll use this as the output from the compute shader. Bind the FBO texture to image texture unit 0, and the second texture to image texture unit 1 using `glBindImageTexture`.

Next, set up a vertex/fragment shader pair for rendering directly to the FBO, and for rendering a full-screen texture.

How to do it...

Use the following steps:

1. In the compute shader, as usual, we start by defining the number of shader invocations per work group.

```
layout (local_size_x = 25, local_size_y = 25) in;
```

2. Next, we declare uniform variables for our input and output images and for the edge detection threshold. The input image is the rendered image from the FBO, and the output image will be the result of the edge detection filter.

```
uniform float EdgeThreshold = 0.1;
layout(binding=0, rgba8) uniform image2D InputImg;
layout(binding=1, rgba8) uniform image2D OutputImg;
```

3. Then we declare our work group's shared memory, which is an array of size 27 x 27.

```
shared float
localData[gl_WorkGroupSize.x+2][gl_WorkGroupSize.y+2];
```

4. We also define a function for computing the luminance of a pixel called `luminance`. Since the same function was used in several previous recipes, this need not be repeated here.

5. Next, we define a function that applies the Sobel filter to the pixel that corresponds to this shader invocation. It reads directly from the local shared data.

```
void applyFilter()
{
```

```

uvec2 p = gl_LocalInvocationID.xy + uvec2(1,1);

float sx = localData[p.x-1][p.y-1] +
            2*localData[p.x-1][p.y] +
            localData[p.x-1][p.y+1] -
            (localData[p.x+1][p.y-1] +
            2 * localData[p.x+1][p.y] +
            localData[p.x+1][p.y+1]);
float sy = localData[p.x-1][p.y+1] +
            2*localData[p.x][p.y+1] +
            localData[p.x+1][p.y+1] -
            (localData[p.x-1][p.y-1] +
            2 * localData[p.x][p.y-1] +
            localData[p.x+1][p.y-1]);
float g = sx * sx + sy * sy;

if( g > EdgeThreshold )
    imageStore(OutputImg,
        ivec2(gl_GlobalInvocationID.xy), vec4(1.0));
else
    imageStore(OutputImg,
        ivec2(gl_GlobalInvocationID.xy), vec4(0,0,0,1));
}

```

6. In the main function, we start by copying the luminance for this pixel into the shared memory array.

```

void main()
{
    localData
    [gl_LocalInvocationID.x+1][gl_LocalInvocationID.y+1] =
        luminance(imageLoad(InputImg,
            ivec2(gl_GlobalInvocationID.xy)).rgb);
}

```

7. If we're on the edge of the work group, we need to copy one or more additional pixels into the shared memory array in order to fill out the pixels around the edge. So we need to determine whether or not we're on the edge of the work group (by examining `gl_LocalInvocationID`), and then determine which pixels we're responsible for copying. This is not complex, but is fairly involved and lengthy, due to the fact that we also must determine whether or not that external pixel actually exists. For example, if this work group is on the edge of the global image, then some of the edge pixels don't exist (are outside of the image). Due to its length, I won't include that code here. For full details, grab the code for this book from the GitHub site.

8. Once we've copied the data for which this shader invocation is responsible, we need to wait for other invocations to do the same, so here we invoke a barrier. Then we call our `applyFilter` function to compute the filter and write the results to the output image.

```
    barrier();

    // Apply the filter using local memory
    applyFilter();
}
```

9. In the OpenGL render function, we start by rendering the scene to the FBO, then dispatch the compute shader, and wait for it to finish all of its writes to the output image.

```
glDispatchCompute(width/25, height/25, 1);
glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
```

10. Finally, we render the output image to the screen via a full-screen quad.

How it works...

In step 1, we specify 625 shader invocations per work group, 25 in each dimension. Depending on the system on which the code is running, this could be changed to better match the hardware available.

The uniform `image2D` variables (step 2) are the input and output images. Note the binding locations indicated in the layout qualifier. These correspond to the image units specified in the `glBindImageTexture` call within the main OpenGL program. The input image should contain the rendered scene, and corresponds to the image texture bound to the FBO. The output image will receive the result of the filter. Also note the use of `rgb8` as the format. This must be the same as the format used when creating the image using `glTexStorage2D`.

The array `localData` is declared in step 3 with the `shared` qualifier. This is our work group's local shared memory. The size is 27×27 in order to include an extra strip, one pixel wide along the edges. We store the luminance of all of the pixels in the work group here, plus the luminance for a strip of surrounding pixels of width one.

The `applyFilter` function (step 5) is where the Sobel operator is computed using the data in `localData`. It is fairly straightforward, except for an offset that needs to be applied due to the extra strip around the edges. The luminance of the pixel that this invocation is responsible for is located at:

```
p = gl_LocalInvocationID.xy + uvec2(1,1);
```

Without the extra strip of pixels, we could just use `gl_LocalInvocationID`, but here we need to add an offset of one in each dimension.

The next few statements just compute the Sobel operator, and determine the magnitude of the gradient, stored in `g`. This is done by reading the luminance of the eight nearby pixels, reading from the shared array `localData`.

At the end of the `applyFilter` function, we write to `OutputImg` the result of the filter. This is either `(1,1,1,1)` or `(0,0,0,1)` depending on whether `g` is above the threshold or not. Note that here, we use `gl_GlobalInvocationID` as the location in the output image. The global ID is appropriate for determining the location within the global image, while the local ID tells us where we are within the local work group, and is more appropriate for access to the local shared array.

In the main function (step 6), we compute the luminance of the pixel corresponding to this invocation (at `gl_GlobalInvocationID`) and store it in the local shared memory (`localData`) at `gl_LocalInvocationID + 1`. Again, the `+ 1` is due to the additional space for the edge pixels.

The next step (step 7) is to copy the edge pixels. We only do so if this invocation is on the edge of the work group. Additionally, we need to determine if the edge pixels actually exist or not. For details on this, please refer to the code that accompanies the book.

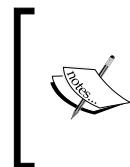
In step 8, we call the GLSL barrier function. This synchronizes all shader invocations within the work group to this point in the code, assuring that all writes to the local shared data have completed. Without calling the barrier function, there's no guarantee that all shader invocations will have finished writing to `localData`, and therefore the data might be incomplete. It is interesting (and instructive) to remove this call and observe the results.

Finally, we call `applyFilter` to compute the Sobel operator and write to the output image.

Within the OpenGL render function, we dispatch the compute shader such that there are enough work groups to cover the image. Since the work group size is 25×25 , we invoke $\text{width}/25$ work groups in the x dimension and $\text{height}/25$ in the y. The result is one shader invocation per pixel in the input/output image.

There's more...

This is a straightforward example of the use of local shared memory. It is only slightly complicated by the fact that we need to deal with the extra row/column of pixels. In general, however, local shared data can be used for any type of communication between invocations within a work group. In this case, the data is not used for communication, but is instead used to increase efficiency by decreasing the global number of reads from the image.



Note that there are (sometimes stringent) limits on the size of shared memory. We can use `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE` (via `glGetInteger*`) to query the maximum size available on the current hardware. The minimum required by the OpenGL specification is 32 KB.

See also

- ▶ The Applying an edge detection filter recipe in Chapter 5, *Image Processing and Screen Space Techniques*

Creating a fractal texture using the compute shader

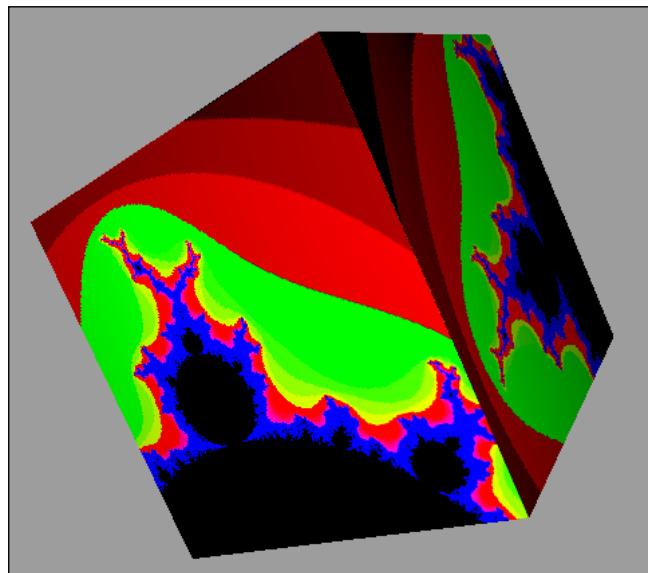
We'll wrap up this chapter with an example that makes use of the compute shader to produce an image of a fractal. We'll use the classic Mandelbrot set.

The Mandelbrot set is based on iterations of the following complex polynomial:

$$z_{n+1} = z_n^2 + c$$

Where **z** and **c** are complex numbers. Starting with the value **z** = 0 + 0*i*, we apply the iteration repeatedly until a maximum number of iterations is reached or the value of **z** exceeds a specified maximum. For a given value of **c**, if the iteration remains stable (**z** doesn't increase above the maximum) the point is inside the Mandelbrot set and we color the position corresponding to **c** black. Otherwise, we color the point based on the number of iterations it took for the value to exceed the maximum.

In the following figure, the image of the Mandelbrot set is applied as a texture to a cube:



We'll use the compute shader to evaluate the Mandelbrot set. Since this is another image-based technique, we'll use a two-dimensional compute space with one compute shader invocation per pixel. Each invocation can work independently, and doesn't need to share any data with other invocations.

Getting ready

Create a texture to store the results of our fractal calculation. The image should be bound to the image texture unit 0 using `glBindImageTexture`.

```
GLuint imgTex;
 glGenTextures(1, &imgTex);
 glBindTexture(GL_TEXTURE_2D, imgTex);
 glBindImageTexture(0, imgTex, 0, GL_FALSE, 0, GL_READ_WRITE,
 GL_RGBA8);
```

How to do it...

Use the following steps:

1. In the compute shader, we start by defining the number of shader invocations per work group.

```
layout( local_size_x = 32, local_size_y = 32 ) in;
```

2. Next, we declare the output image as well as some other uniform variables.

```
layout( binding = 0, rgba8 ) uniform image2D ColorImg;
#define MAX_ITERATIONS 100
uniform vec4 CompWindow;
uniform uint Width = 256;
uniform uint Height = 256;
```

3. We define a function to compute the number of iterations for a given position on the complex plane.

```
int mandelbrot( vec2 c ) {
    vec2 z = vec2(0.0,0.0);
    uint i = 0;
    while(i < MAX_ITERATIONS && (z.x*z.x + z.y*z.y) < 4.0) {
        z = vec2( z.x*z.x-z.y*z.y+c.x, 2 * z.x*z.y + c.y );
        i++;
    }
    return i;
}
```

4. In the main function, we start by computing the size of a pixel in the complex space.

```
void main() {
    float dx = (CompWindow.z - CompWindow.x) / Width;
    float dy = (CompWindow.w - CompWindow.y) / Height;
```

5. Then we determine the value of c for this invocation.

```
vec2 c = vec2(
    dx * gl_GlobalInvocationID.x + CompWindow.x,
    dy * gl_GlobalInvocationID.y + CompWindow.y);
```

6. Next we call the `mandelbrot` function and determine the color based on the number of iterations.

```
uint i = mandelbrot(c);
vec4 color = vec4(0.0,0.5,0.5,1);
if( i < MAX_ITERATIONS ) {
    if( i < 5 )
        color = vec4(float(i)/5.0,0,0,1);
    else if( i < 10 )
        color = vec4((float(i)-5.0)/5.0,1,0,1);
    else if( i < 15 )
        color = vec4(1,0,(float(i)-10.0)/5.0,1);
    else color = vec4(0,0,1,0);
}
else
    color = vec4(0,0,0,1);
```

7. Finally, we write the color to the output image.

```
imageStore(ColorImg,
    ivec2(gl_GlobalInvocationID.xy), color);
}
```

8. Within the render function of the OpenGL program, we execute the compute shader with one invocation per texel, and call `glMemoryBarrier`.

```
glDispatchCompute(256/32, 256/32, 1);
glMemoryBarrier( GL_SHADER_IMAGE_ACCESS_BARRIER_BIT );
```

9. Then we render the scene, applying the texture to the appropriate objects.

How it works...

In step 2, the uniform variable `ColorImg` is the output image. It is defined to be located at image texture unit 0 (via the binding layout option). Also note that the format is `rgb8`, which must be the same as what is used in the `glTexStorage2D` call when creating the texture.

`MAX_ITERATIONS` is the maximum number of iterations of the complex polynomial mentioned above. `CompWindow` is the region of complex space with which we are working. The first two components `CompWindow.xy` are the real and imaginary parts of the lower left corner of the window, and `CompWindow.zw` is the upper right corner. `Width` and `Height` define the size of the texture image.

The `mandelbrot` function (step 3) takes a value for `c` as the parameter, and repeatedly iterates the complex function until either a maximum number of iterations is reached, or the absolute value of `z` becomes greater than 2. Note that here, we avoid computing the square root and just compare the absolute value squared with 4. The function returns the total number of iterations.

Within the main function (step 4), we start by computing the size of a pixel within the complex window (`dx`, `dy`). This is just the size of the window divided by the number of texels in each dimension.

The compute shader invocation is responsible for the texel located at `gl_GlobalInvocationID.xy`. We compute the point on the complex plane that corresponds to this texel next. For the x position (real axis), we take the size of the texel in that direction (`dx`) times `gl_GlobalInvocationID.x` (which gives the distance from the left edge of the window), plus the position of the left edge of the window (`CompWindow.x`). A similar calculation is done for the y position (imaginary axis).

In step 6, we call the `mandelbrot` function with the value for `c` that was just determined, and determine a color based on the number of iterations returned.

In step 7, we apply the color to the output image at `gl_GlobalInvocationID.xy` using `imageStore`.

In the OpenGL render function (step 8), we dispatch the compute shader with enough invocations so that there is one invocation per texel. The `glMemoryBarrier` call assures that all writes to the output image are complete before continuing.

There's more...

Prior to the advent of the compute shader, we might have chosen to do this using the fragment shader. However, the compute shader gives us a bit more flexibility in defining how the work is allocated on the GPU. We can also gain memory efficiency by avoiding the overhead of a complete FBO for the purposes of a single texture.

Module 3

OpenGL Data Visualization Cookbook

*Over 35 hands-on recipes to create impressive, stunning visuals for a wide range
of real-time, interactive applications using OpenGL*

1

Getting Started with OpenGL

In this chapter, we will cover the following topics:

- ▶ Setting up a Windows-based development platform
- ▶ Setting up a Mac-based development platform
- ▶ Setting up a Linux-based development platform
- ▶ Installing the GLFW library in Windows
- ▶ Installing the GLFW library in Mac OS X and Linux
- ▶ Creating your first OpenGL application with GLFW
- ▶ Compiling and running your first OpenGL application in Windows
- ▶ Compiling and running your first OpenGL application in Mac OS X or Linux

Introduction

OpenGL is an ideal multiplatform, cross-language, and hardware-accelerated graphics rendering interface that is well suited to visualize large 2D and 3D datasets in many fields. In fact, OpenGL has become the industry standard to create stunning graphics, most notably in gaming applications and numerous professional tools for 3D modeling. As we collect more and more data in fields ranging from biomedical imaging to wearable computing (especially with the evolution of Big Data), a high-performance platform for data visualization is becoming an essential component of many future applications. Indeed, the visualization of massive datasets is becoming an increasingly challenging problem for developers, scientists, and engineers in many fields. Therefore, OpenGL can provide a unified solution for the creation of impressive, stunning visuals in many real-time applications.

The APIs of OpenGL encapsulate the complexity of hardware interactions while allowing users to have low-level control over the process. From a sophisticated multiserver setup to a mobile device, OpenGL libraries provide developers with an easy-to-use interface for high-performance graphics rendering. The increasing availability and capability of graphics hardware and mass storage devices, coupled with their decreasing cost, further motivate the development of interactive OpenGL-based data visualization tools.

Modern computers come with dedicated **Graphics Processing Units (GPUs)**, highly customized pieces of hardware designed to accelerate graphics rendering. GPUs can also be used to accelerate general-purpose, highly parallelizable computational tasks. By leveraging hardware and OpenGL, we can produce highly interactive and aesthetically pleasing results.

This chapter introduces the essential tools to develop OpenGL-based data visualization applications and provides a step-by-step tutorial on how to set up the environment for our first demo application. In addition, this chapter outlines the steps to set up a popular tool called CMake, which is a cross-platform software that automates the process of generating standard build files (for example, makefiles in Linux that define the compilation parameters and commands) with simple configuration files. The CMake tool will be used to compile additional libraries in the future, including the GLFW (OpenGL FrameWork) library introduced later in this chapter. Briefly, the GLFW library is an open source, multiplatform library that allows users to create and manage windows with OpenGL contexts as well as handle inputs from peripheral devices such as the mouse and keyboard. By default, OpenGL itself does not support other peripherals; thus, the GLFW library is used to fill in the gap. We hope that this detailed tutorial will be especially useful for beginners who are interested in exploring OpenGL for data visualization but have little or no prior experience. However, we will assume that you are familiar with the C/C++ programming language.

Setting up a Windows-based development platform

There are various development tools available to create applications in the Windows environment. In this book, we will focus on creating OpenGL applications using Visual C++ from Microsoft Visual Studio 2013, given its extensive documentation and support.

Installing Visual Studio 2013

In this section, we outline the steps to install Visual Studio 2013.

Getting ready

We assume that you have already installed Windows 7.0 or higher. For optimal performance, we recommend that you get a dedicated graphics card, such as NVIDIA GeForce graphics cards, and have at least 10 GB of free disk space as well as 4 GB of RAM on your computer. Download and install the latest driver for your graphics card.

How to do it...

To install Microsoft Visual Studio 2013 for free, download the Express 2013 version for Windows Desktop from Microsoft's official website (refer to <https://www.visualstudio.com/en-us/downloads/>). Once you have downloaded the installer executable, we can start the process. By default, we will assume that programs are installed in the following path:



To verify the installation, click on the **Launch** button at the end of the installation, and it will execute the VS Express 2013 for Desktop application for the first time.

Installing CMake in Windows

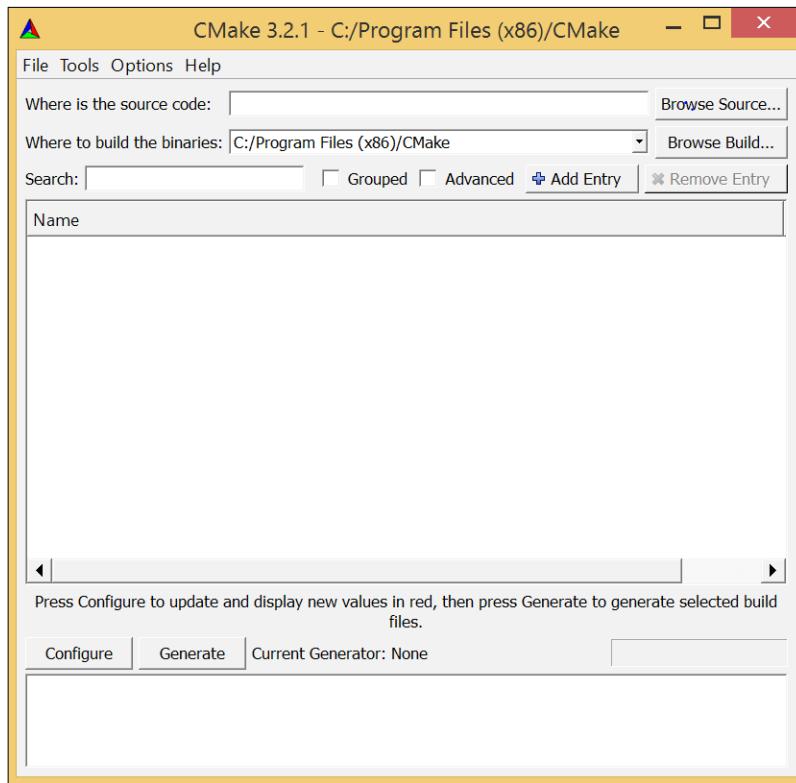
In this section, we outline the steps to install CMake, which is a popular tool that automates the process of creating standard build files for Visual Studio (among other tools).

Getting ready

To obtain the CMake tool (CMake 3.2.1), you can download the executable (cmake-3.2.1-win32-x86.exe) from <http://www.cmake.org/download/>.

How to do it...

The installation wizard will guide you through the process (select **Add CMake to the system PATH for all users** when prompted for installation options). To verify the installation, run CMake(cmake-gui).



At this point, you should have both Visual Studio 2013 and CMake successfully installed on your machine and be ready to compile/install the GLFW library to create your first OpenGL application.

Setting up a Mac-based development platform

One important advantage of using OpenGL is the possibility of cross-compiling the same source code on different platforms. If you are planning to develop your application on a Mac platform, you can easily set up your machine for development using the upcoming steps. We assume that you have either Mac OS X 10.9 or higher installed. OpenGL updates are integrated into the system updates for Mac OS X through the graphics driver.

Installing Xcode and command-line tools

The Xcode development software from Apple provides developers with a comprehensive set of tools, which include an IDE, OpenGL headers, compilers, and debugging tools, to create native Mac applications. To simplify the process, we will compile our code using the command-line interface that shares most of the common features in Linux.

Getting ready

If you are using Mac OS X 10.9 or higher, you can download Xcode through the App Store shipped with Mac OS. Full installation support and instructions are available on the Apple Developer website (<https://developer.apple.com/xcode/>).

How to do it...

We can install the command-line tools in Xcode through the following steps:

1. Search for the keyword Terminal in **Spotlight** and run **Terminal**.



2. Execute the following command in the terminal:

```
xcode-select --install
```

Note that if you have previously installed the command-line tools, an error stating "command-line are already installed" will appear. In this case, simply skip to step 4 to verify the installation.

3. Click on the **Install** button to directly install the command-line tools. This will install basic compiling tools such as **gcc** and **make** for application development purposes (note that CMake needs to be installed separately).
4. Finally, enter `gcc --version` to verify the installation.

```
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx
-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 6.0 (clang-600.0.57) (based on LLVM 3.5svn)
Target: x86_64-apple-darwin14.1.0
Thread model: posix
```

See also

If you encounter the **command not found** error or other similar issues, make sure that the command-line tools are installed successfully. Apple provides an extensive set of documentation, and more information on installing Xcode can be found at <https://developer.apple.com/xcode>.

Installing MacPorts and CMake

In this section, we outline the steps to install MacPorts, which greatly simplifies the subsequent setup steps, and CMake for Mac.

Getting ready

Similar to the Windows installation, you can download the binary distribution of **CMake** from <http://www.cmake.org/cmake/resources/software.html> and manually configure the command-line options. However, to simplify the installation and automate the configuration process, we highly recommend that you use MacPorts.

How to do it...

To install MacPorts, follow these steps:

1. Download the MacPorts package installer for the corresponding version of Mac OS X (<https://guide.macports.org/#installing.macports>):
 - ❑ Mac OS X 10.10 Yosemite: <https://distfiles.macports.org/MacPorts/MacPorts-2.3.3-10.10-Yosemite.pkg>
 - ❑ Mac OS X 10.9 Mavericks: <https://distfiles.macports.org/MacPorts/MacPorts-2.3.3-10.9-Mavericks.pkg>
2. Double-click on the package installer and follow the onscreen instructions.



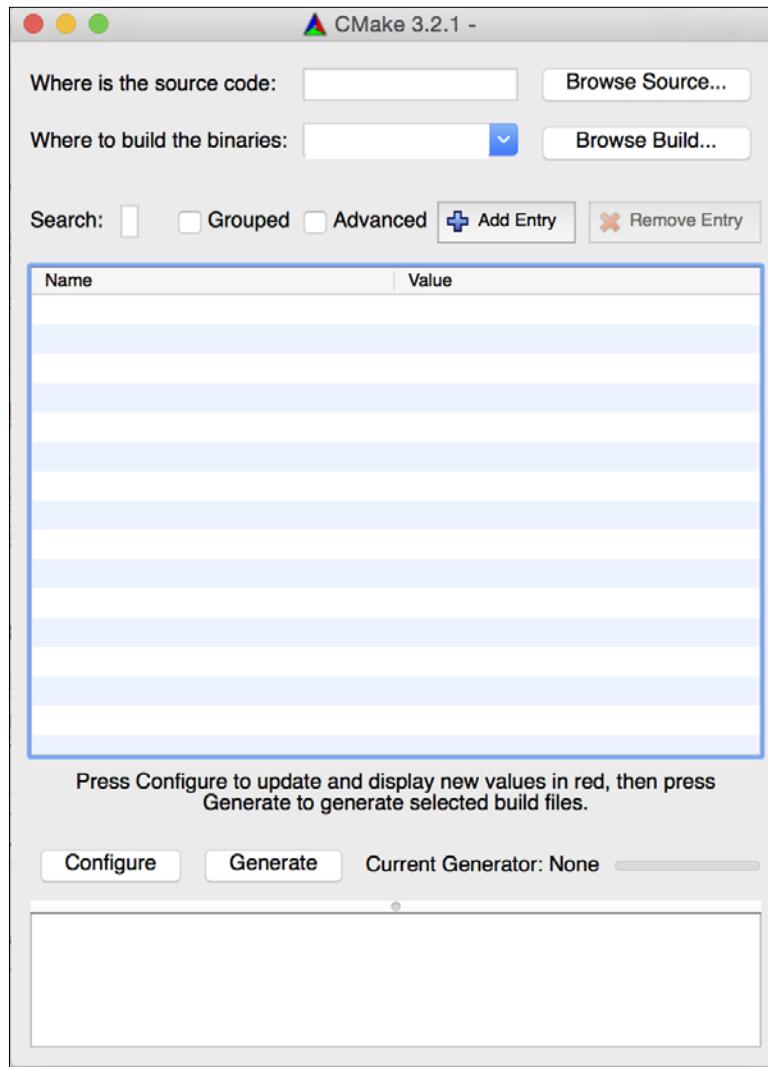
3. Verify the installation in the terminal by typing in `port version`, which returns the version of MacPorts currently installed (Version: 2.3.3 in the preceding package).

To install **CMake** on Mac, follow these steps:

1. Open the **Terminal** application.
2. Execute the following command:

```
sudo port install cmake +gui
```

To verify the installation, enter `cmake --version` to show the current version installed and enter `cmake-gui` to explore the GUI.



At this point, your Mac is configured for OpenGL development and is ready to compile your first OpenGL application. For those who have been more accustomed to GUIs, using the command-line interface in Mac can initially be an overwhelming experience. However, in the long run, it is a rewarding learning experience due to its overall simplicity. Command-line tools and interfaces are often more time-invariant compared to constantly evolving GUIs. At the end of the day, you can just copy and paste the same command lines, thereby saving precious time needed to consult new documentation every time a GUI changes.

Setting up a Linux-based development platform

To prepare your development environment on the Linux platform, we can utilize the powerful Debian Package Management system. The `apt-get` or `aptitude` program automatically retrieves the precompiled packages from the server and also resolves and installs all dependent packages that are required. If you are using non-Debian based platform, such as Fedora, you can find the equivalents by searching for the keywords of each packages listed in this recipe.

Getting ready

We assume that you have successfully installed all updates and latest graphics drivers associated with your graphics hardware. Ubuntu 12.04 or higher has support for third-party proprietary NVIDIA and AMD graphics drivers, and more information can be found at <https://help.ubuntu.com/community/BinaryDriverHowto>.

How to do it...

Use the following steps to install all development tools and the associated dependencies:

1. Open a terminal.
2. Enter the update command:

```
sudo apt-get update
```

3. Enter the install command and enter `y` for all prompts:

```
sudo apt-get install build-essential cmake-gui xorg-dev  
libglu1-mesa-dev mesa-utils
```

4. Verify the results:

```
gcc --version
```

If successful, this command should return the current version of `gcc` installed.

How it works...

In summary, the `apt-get update` command automatically updates the local database in the Debian Package Management system. This ensures that the latest packages are retrieved and installed in the process. The `apt-get` system also provides other package management features, such as package removal (uninstall), dependency retrieval, as well as package upgrades. These advanced functions are outside the scope of this book, but more information can be found at <https://wiki.debian.org/apt-get>.

The preceding commands install a number of packages to your machine. Here, we will briefly explain the purpose of each package.

The `build-essential` package, as the name itself suggests, encapsulates the essential packages, namely `gcc` and `g++`, that are required to compile C and C++ source code in Linux. Additionally, it will download header files and resolve all dependencies in the process.

The `cmake-gui` package is the CMake program described earlier in the chapter. Instead of downloading CMake directly from the website and compiling from the source, it retrieves the latest supported version that had been compiled, tested, and released by the Ubuntu community. One advantage of using the Debian Package Management system is the stability and ease of updating in the future. However, for users who are looking for the cutting-edge version, `apt-get` based systems would be a few versions behind.

The `xorg-dev` and `libglu1-mesa-dev` packages are the development files required to compile the GLFW library. These packages include header files and libraries required by other programs. If you choose to use the precompiled binary version of GLFW, you may be able to skip some of the packages. However, we highly recommend that you follow the steps for the purpose of this tutorial.

See also

For more information, most of the steps described are documented and explained in depth in this online documentation: <https://help.ubuntu.com/community/UsingTheTerminal>.

Installing the GLFW library in Windows

There are two ways to install the GLFW library in Windows, both of which will be discussed in this section. The first approach involves compiling the GLFW source code directly with CMake for full control. However, to simplify the process, we suggest that you download the precompiled binary distribution.

Getting ready

We assume that you have successfully installed both Visual Studio 2013 and CMake, as described in the earlier section. For completeness, we will demonstrate how to install GLFW using CMake.

How to do it...

To use the precompiled binary package for GLFW, follow these steps:

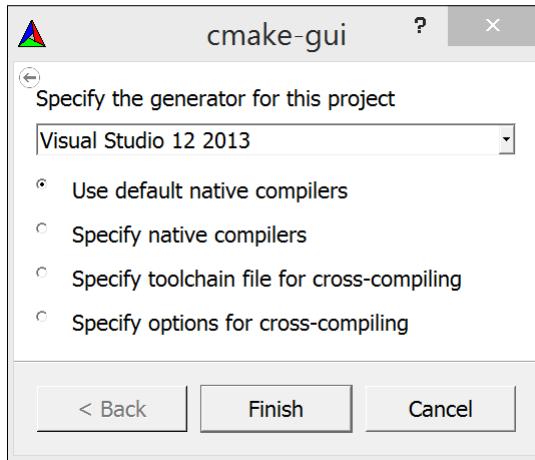
1. Create the `C:/Program Files (x86)/glfw-3.0.4` directory. Grant the necessary permissions when prompted.
2. Download the `glfw-3.0.4.bin.WIN32.zip` package from <http://sourceforge.net/projects glfw/files glfw/3.0.4/glfw-3.0.4.bin.WIN32.zip> and unzip the package.
3. Copy all the extracted content inside the `glfw-3.0.4.bin.WIN32` folder (for example, include `lib-msvc2012`) into the `C:/Program Files (x86)/glfw-3.0.4` directory. Grant permissions when prompted.
4. Rename the `lib-msvc2012` folder to `lib` inside the `C:/Program Files (x86)/glfw-3.0.4` directory. Grant permissions when prompted.

Alternatively, to compile the source files directly, follow these procedures:

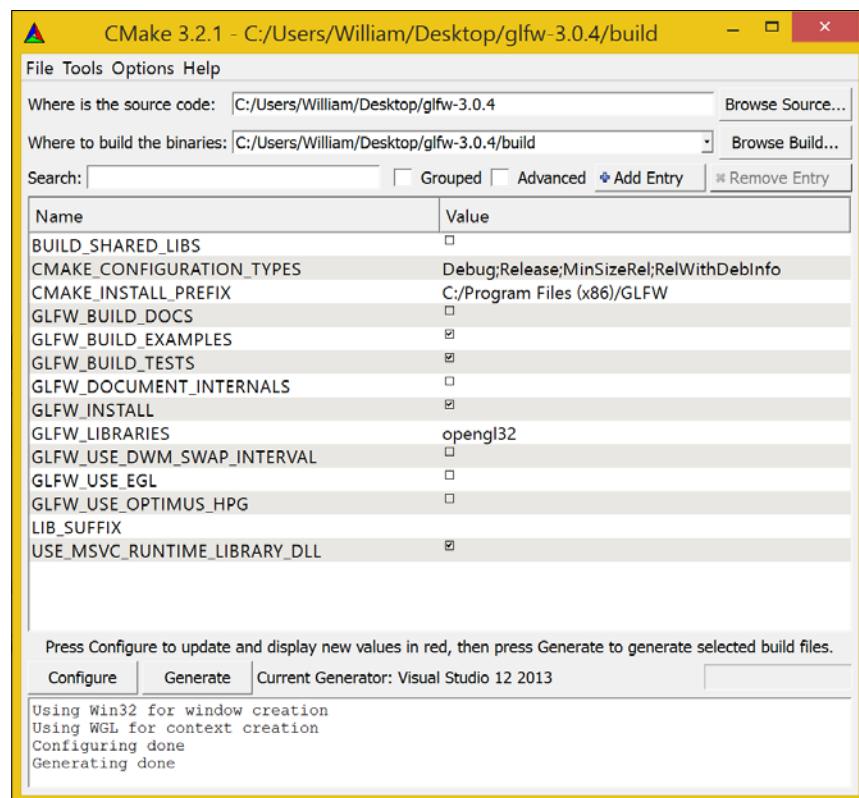
1. Download the source package from <http://sourceforge.net/projects glfw/files glfw/3.0.4/glfw-3.0.4.zip> and unzip the package on the desktop. Create a new folder called `build` inside the extracted `glfw-3.0.4` folder to store the binaries and open `cmake-gui`.
2. Select `glfw-3.0.4` (from the desktop) as the source directory and `glfw-3.0.4/build` as the build directory. The screenshot is shown as follows:



3. Click on **Generate** and select **Visual Studio 12 2013** in the prompt.



4. Click on **Generate** again.



5. Open the build directory and double-click on **GLFW.sln** to open Visual Studio.
6. In Visual Studio, click Build Solution (press *F7*).
7. Copy **build/src/Debug/glfw3.lib** to **C:/Program Files (x86)/glfw-3.0.4/lib**.
8. Copy the **include** directory (inside **glfw-3.0.4/include**) to **C:/Program Files (x86)/glfw-3.0.4/**.

After this step, we should have the **include** (**glfw3.h**) and library (**glfw3.lib**) files inside the **C:/Program Files (x86)/glfw-3.0.4** directory, as shown in the setup procedure using precompiled binaries.

Installing the GLFW library in Mac OS X and Linux

The installation procedures for Mac and Linux are essentially identical using the command-line interface. To simplify the process, we recommend that you use MacPorts for Mac users.

Getting ready

We assume that you have successfully installed the basic development tools, including CMake, as described in the earlier section. For maximum flexibility, we can compile the library directly from the source code (refer to <http://www.glfw.org/docs/latest/compile.html> and <http://www.glfw.org/download.html>).

How to do it...

For Mac users, enter the following command in a terminal to install GLFW using MacPorts:

```
sudo port install glfw
```

For Linux users (or Mac users who would like to practice using the command-line tools), here are the steps to compile and install the GLFW source package directly with the command-line interface:

1. Create a new folder called **opengl_dev** and change the current directory to the new path:

```
mkdir ~/opengl_dev  
cd ~/opengl_dev
```
2. Obtain a copy of the GLFW source package (**glfw-3.0.4**) from the official repository: <http://sourceforge.net/projects/glfw/files/glfw/3.0.4/glfw-3.0.4.tar.gz>.

3. Extract the package.

```
tar xzvf glfw-3.0.4.tar.gz
```

4. Perform the compilation and installation:

```
cd glfw-3.0.4
mkdir build
cd build
cmake ../
make && sudo make install
```

How it works...

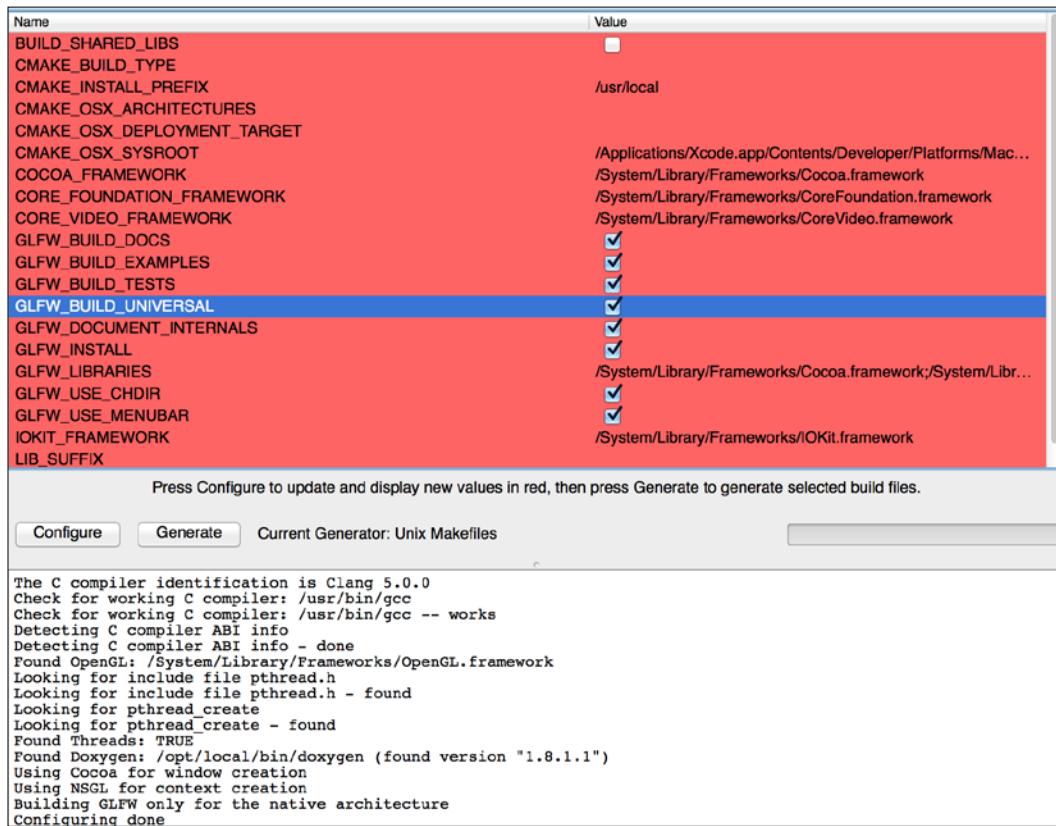
The first set of commands create a new working directory to store the new files retrieved using the `wget` command, which downloads a copy of the GLFW library to the current directory. The `tar xzvf` command extracts the compressed packages and creates a new folder with all the contents.

Then, the `cmake` command automatically generates the necessary build files that are needed for the compilation process to the current `build` directory. This process also checks for missing dependencies and verifies the versioning of the applications.

The `make` command then takes all instructions from the Makefile script that is generated automatically and compiles the source code into libraries.

The `sudo make install` command installs the library header files as well as the static or shared libraries onto your machine. As this command requires writing to the root directory, the `sudo` command is needed to grant such permissions. By default, the files will be copied to the `/usr/local` directory. In the rest of the book, we will assume that the installations follow these default paths.

For advanced users, we can optimize the compilation by configuring the packages with the CMake GUI (`cmake-gui`).



For example, you can enable the `BUILD_SHARED_LIBS` option if you are planning to compile the GLFW library as a shared library. In this book, we will not explore the full functionality of the GLFW library, but these options can be useful to developers who are looking for further customizations. Additionally, you can customize the installation prefix (`CMAKE_INSTALL_PREFIX`) if you would like to install the library files at a separate location.

Creating your first OpenGL application with GLFW

Now that you have successfully configured your development platform and installed the GLFW library, we will provide a tutorial on how to create your first OpenGL-based application.

Getting ready

At this point, you should already have all the pre requisite tools ready regardless of which operating system you may have, so we will immediately jump into building your first OpenGL application using these tools.

How to do it...

The following code outlines the basic steps to create a simple OpenGL program that utilizes the GLFW library and draws a rotating triangle:

1. Create an empty file, and then include the header file for the GLFW library and standard C++ libraries:

```
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>
```

2. Initialize GLFW and create a GLFW window object (640 x 480):

```
int main(void)
{
    GLFWwindow* window;
    if (!glfwInit())
        exit(EXIT_FAILURE);
    window = glfwCreateWindow(640, 480, "Chapter 1: Simple
        GLFW Example", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    glfwMakeContextCurrent(window);
```

3. Define a loop that terminates when the window is closed:

```
while (!glfwWindowShouldClose(window))
{
```

4. Set up the viewport (using the width and height of the window) and clear the screen color buffer:

```
float ratio;
int width, height;

glfwGetFramebufferSize(window, &width, &height);
ratio = (float) width / (float) height;

glViewport(0, 0, width, height);
glClear(GL_COLOR_BUFFER_BIT);
```

5. Set up the camera matrix. Note that further details on the camera model will be discussed in *Chapter 3, Interactive 3D Data Visualization*:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-ratio, ratio, -1.f, 1.f, 1.f, -1.f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

6. Draw a rotating triangle and set a different color (red, green, and blue channels) for each vertex (x , y , and z) of the triangle. The first line rotates the triangle over time:

```
glRotatef((float)glfwGetTime() * 50.f, 0.f, 0.f, 1.f);
 glBegin(GL_TRIANGLES);
 glColor3f(1.f, 0.f, 0.f);
 glVertex3f(-0.6f, -0.4f, 0.f);
 glColor3f(0.f, 1.f, 0.f);
 glVertex3f(0.6f, -0.4f, 0.f);
 glColor3f(0.f, 0.f, 1.f);
 glVertex3f(0.f, 0.6f, 0.f);
 glEnd();
```

7. Swap the front and back buffers (GLFW uses double buffering) to update the screen and process all pending events:

```
glfwSwapBuffers(window);
glfwPollEvents();
}
```

8. Release the memory and terminate the GLFW library. Then, exit the application:

```
glfwDestroyWindow(window);
glfwTerminate();
exit(EXIT_SUCCESS);
}
```

9. Save the file as `main.cpp` using the text editor of your choice.

How it works...

By including the GLFW library header, `glfw3.h`, we automatically import all necessary files from the OpenGL library. Most importantly, GLFW automatically determines the platform and thus allows you to write portable source code seamlessly.

In the main function, we must first initialize the GLFW library with the `glfwInit` function in the main thread. This is required before any GLFW functions can be used. Before a program exits, GLFW should be terminated to release any allocated resources.

Then, the `glfwCreateWindow` function creates a window and its associated context, and it also returns a pointer to the `GLFWwindow` object. Here, we can define the width, height, title, and other properties for the window. After the window is created, we then call the `glfwMakeContextCurrent` function to switch the context and make sure that the context of the specified window is current on the calling thread.

At this point, we are ready to render our graphics element on the window. The `while` loop provides a mechanism to redraw our graphics as long as the window remains open. OpenGL requires an explicit setup on the camera parameters; further details will be discussed in the upcoming chapters. In the future, we can provide different parameters to simulate perspective and also handle more complicated issues (such as anti-aliasing). For now, we have set up a simple scene to render a basic primitive shape (namely a triangle) and fixed the color for the vertices. Users can modify the parameters in the `glColor3f` and `glVertex3f` functions to change the color as well as the position of the vertices.

This example demonstrates the basics required to create graphics using OpenGL. Despite the simplicity of the sample code, it provides a nice introductory framework on how you can create high-performance graphics rendering applications with graphics hardware using OpenGL and GLFW.

Compiling and running your first OpenGL application in Windows

There are several ways to set up an OpenGL project. Here, we create a sample project using Visual Studio 2013 or higher and provide a complete walkthrough for the first-time configuration of the OpenGL and GLFW libraries. These same steps can be incorporated into your own projects in the future.

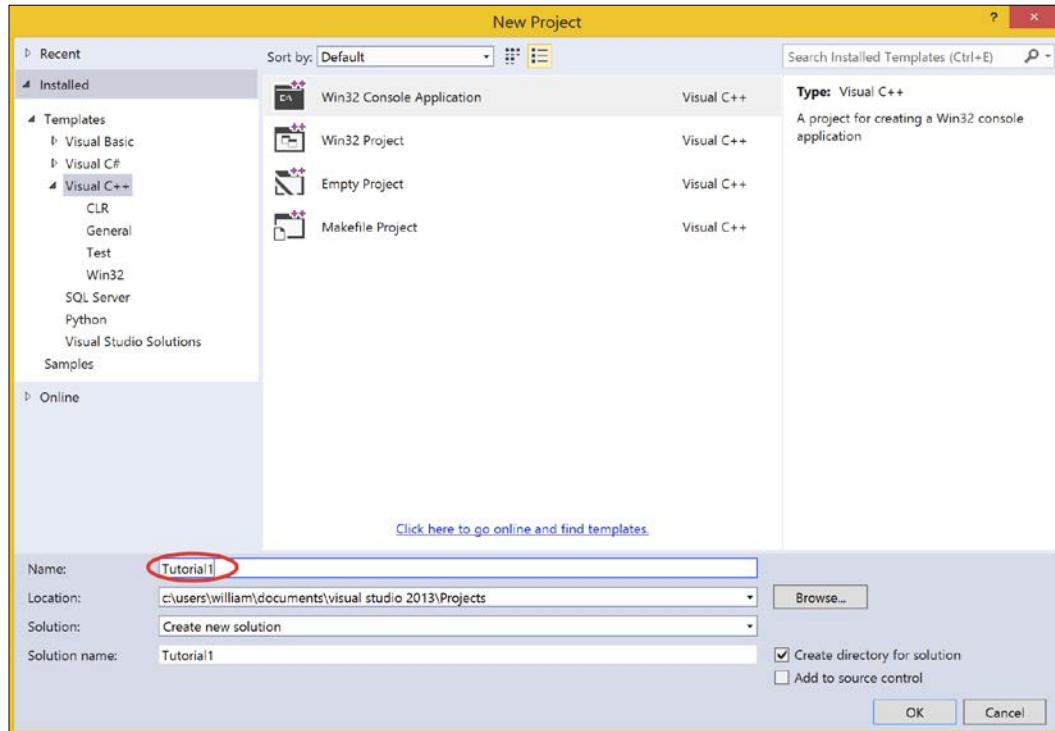
Getting ready

Assuming that you have both Visual Studio 2013 and GLFW (version 3.0.4) installed successfully on your environment, we will start our project from scratch.

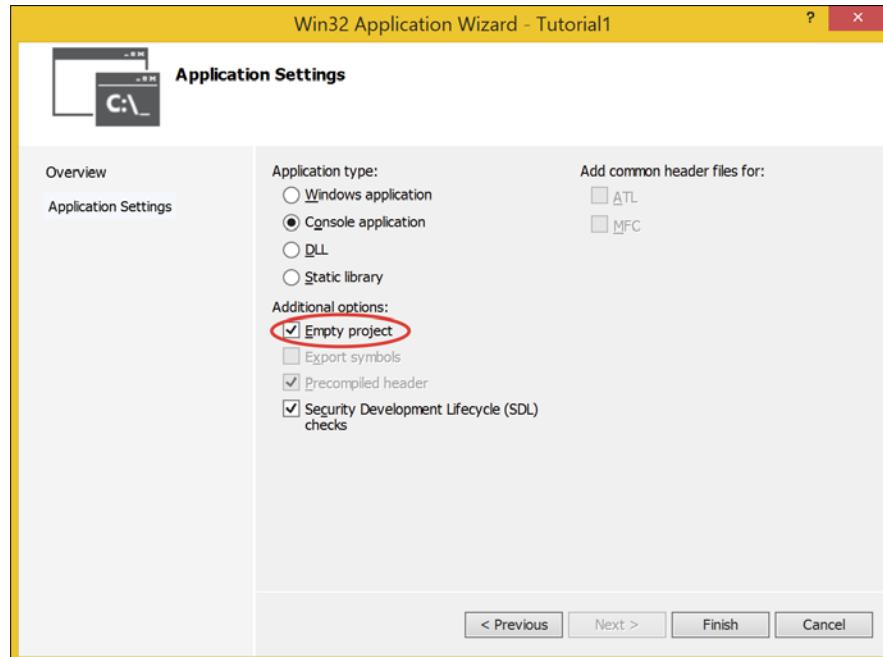
How to do it...

In Visual Studio 2013, use the following steps to create a new project and compile the source code:

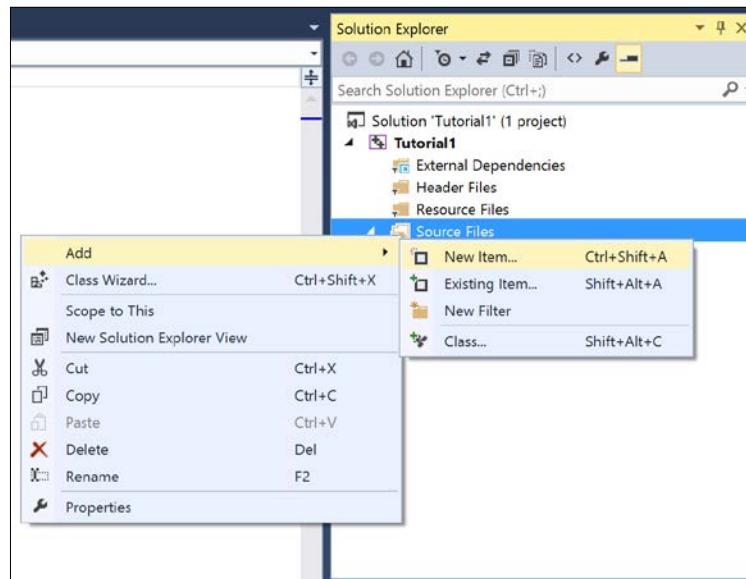
1. Open Visual Studio 2013 (VS Express 2013 for desktop).
2. Create a new Win32 Console Application and name it as Tutorial1.



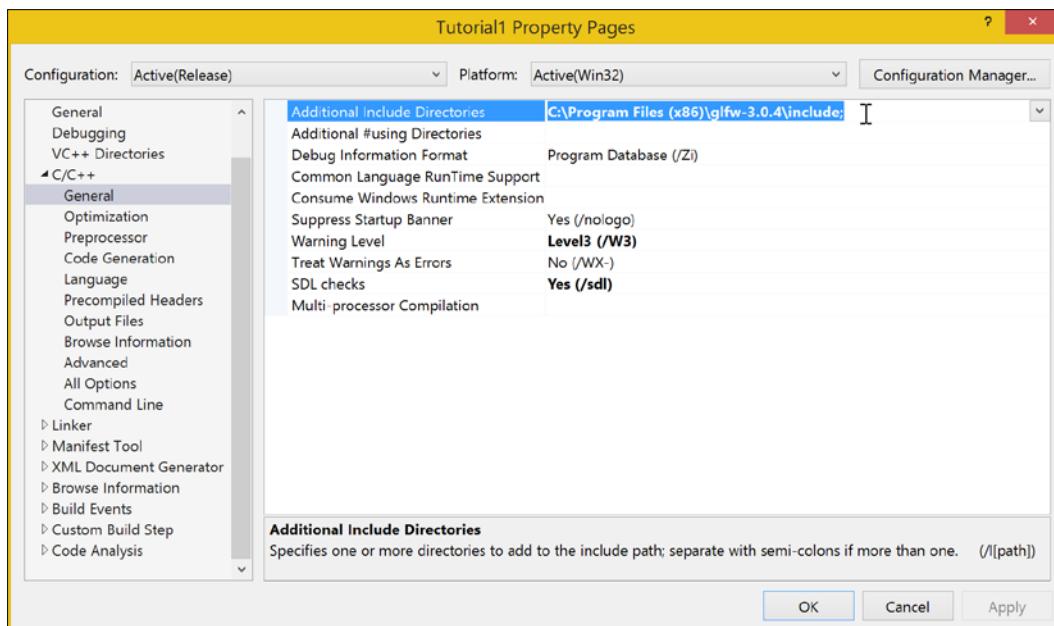
3. Check the **Empty project** option, and click on **Finish**.



4. Right-click on **Source Files**, and add a new C++ source file (**Add | New Item**) called **main.cpp**.



5. Copy and paste the source code from the previous section into the `main.cpp` and save it.
6. Open **Project Properties** (*Alt + F7*).
7. Add the include path of the GLFW library, `C:\Program Files (x86)\glfw-3.0.4\include`, by navigating to **Configuration Properties | C/C++ | General | Additional Include Directories**.

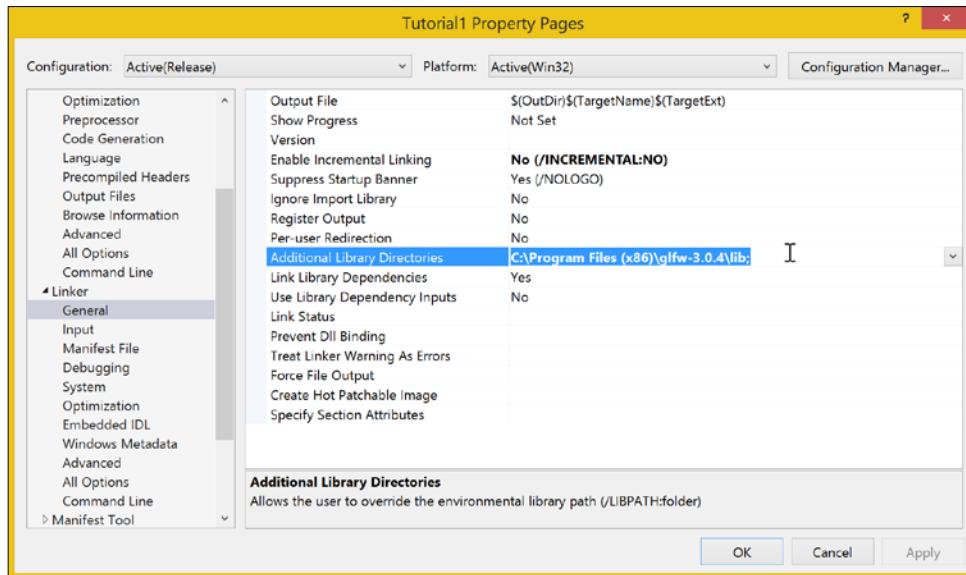


Downloading the example code

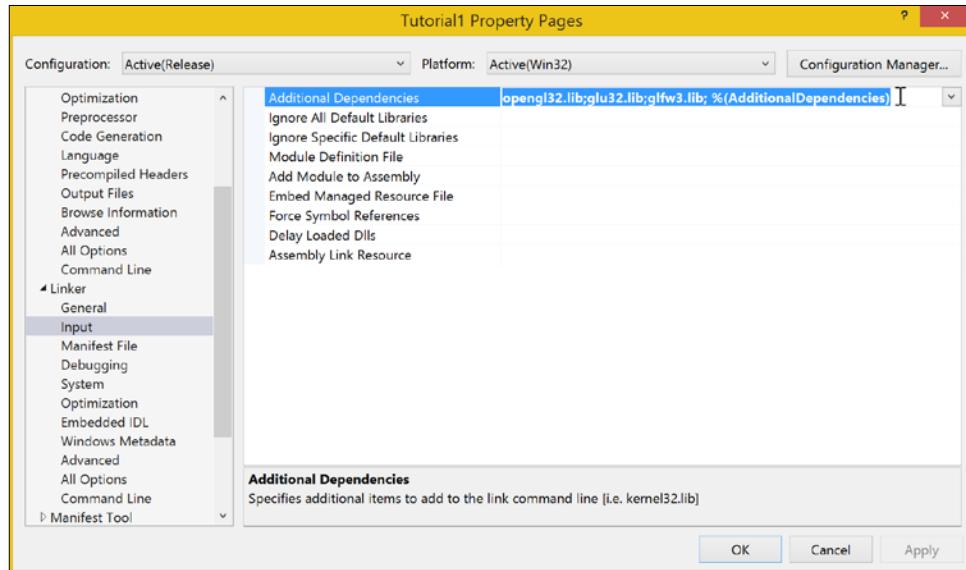
You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Getting Started with OpenGL

8. Add the GLFW library path, `C:\Program Files (x86)\glfw-3.0.4\lib`, by navigating to **Configuration Properties | Linker | General | Additional Library Directories**.

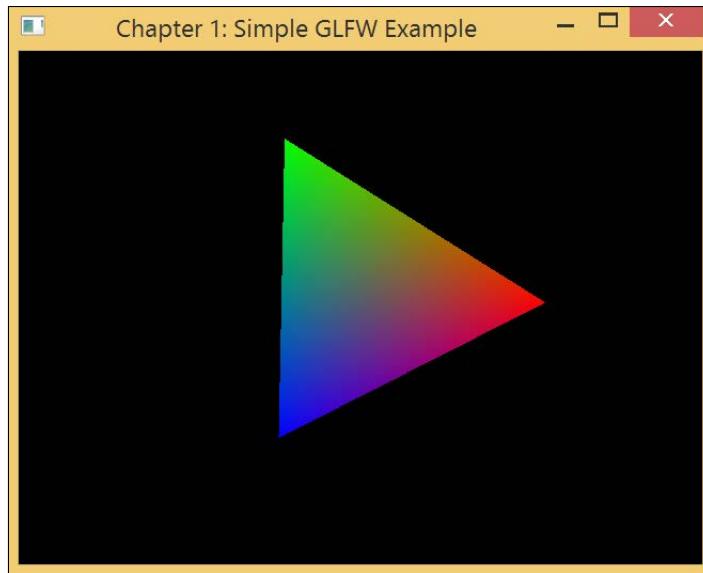


9. Add the GLFW and OpenGL libraries (`glu32.lib`, `glfw3.lib` and `opengl32.lib`) by navigating to **Configuration Properties | Linker | Input | Additional Dependencies**.



10. Build **Solution** (press *F7*).
11. Run the program (press *F5*).

Here is your first OpenGL application showing a rotating triangle that is running natively on your graphics hardware. Although we have only defined the color of the vertices to be red, green, and blue, the graphics engine interpolates the intermediate results and all calculations are performed using the graphics hardware. The screenshot is shown as follows:



Compiling and running your first OpenGL application in Mac OS X or Linux

Setting up a Linux or Mac machine is made much simpler with the command-line interface. We assume that you have all the components that were discussed earlier ready, and all default paths are used as recommended.

Getting ready

We will start by compiling the sample code described previously. You can download the complete code package from the official website of Packt Publishing <https://www.packtpub.com>. We assume that all files are saved to a top-level directory called `code` and the `main.cpp` file is saved inside the `/code/Tutorial1` subdirectory.

How to do it...

1. Open a terminal or an equivalent command-line interface.
2. Change the current directory to the working directory:

```
cd ~/code
```

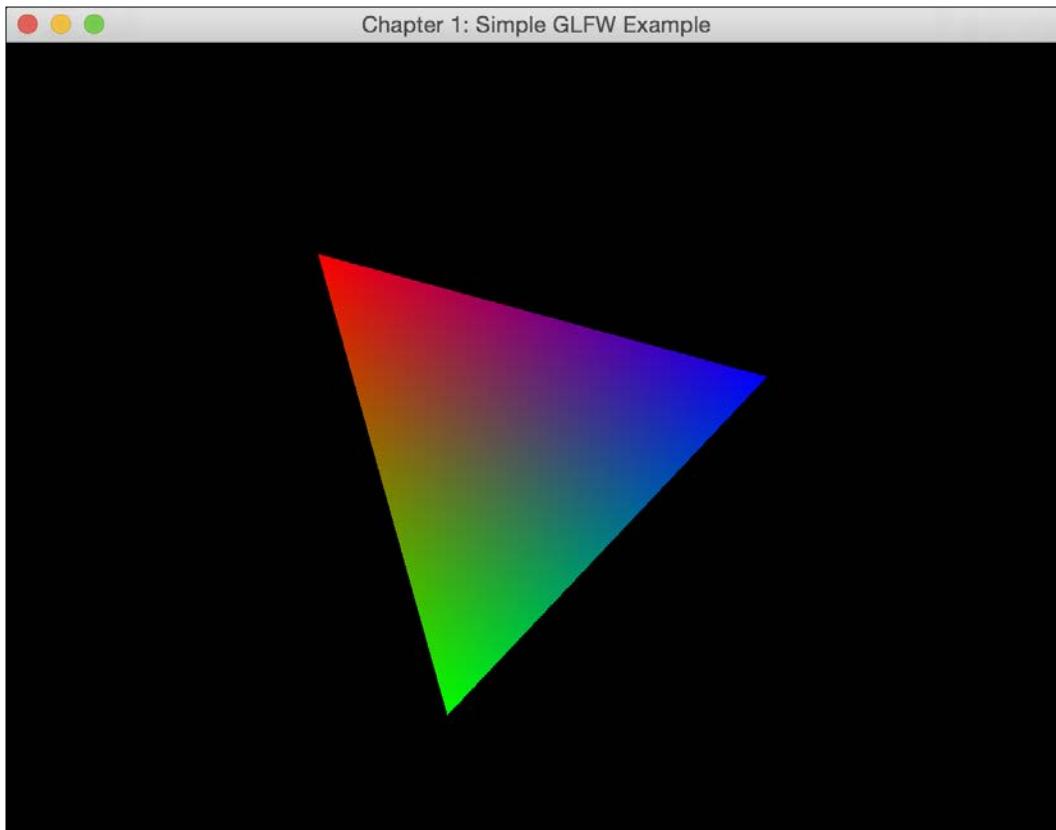
3. Enter the following command to compile the program:

```
gcc -Wall `pkg-config --cflags glfw3` -o main Tutorial1/main.cpp  
`pkg-config --static --libs glfw3`
```

4. Run the program:

```
./main
```

Here is your first OpenGL application that runs natively on your graphics hardware and displays a rotating triangle. Although we have defined the color of only three vertices to be red, green, and blue, the graphics engine interpolates the intermediate results and all calculations are performed using the graphics hardware.



To further simplify the process, we have provided a compile script in the sample code. You can execute the script by simply typing the following commands in a terminal:

```
chmod +x compile.sh  
./compile.sh
```

You may notice that the OpenGL code is platform-independent. One of the most powerful features of the GLFW library is that it handles the windows management and other platform-dependent functions behind the scene. Therefore, the same source code (`main.cpp`) can be shared and compiled on multiple platforms without the need for any changes.

2

OpenGL Primitives and 2D Data Visualization

In this chapter, we will cover the following topics:

- ▶ OpenGL primitives
- ▶ Creating a 2D plot using primitives
- ▶ Real-time visualization of time series
- ▶ 2D visualization of 3D/4D datasets

Introduction

In the previous chapter, we provided a sample code to render a triangle on the screen using OpenGL and the GLFW library. In this chapter, we will focus on the use of OpenGL primitives, such as points, lines, and triangles, to enable the basic 2D visualization of data, including time series such as an **electrocardiogram (ECG)**. We will begin with an introduction to each primitive, along with sample code to allow readers to experiment with the OpenGL primitives with a minimal learning curve.

One can think of primitives as the fundamental building blocks to create graphics using OpenGL. These building blocks can be easily reused in many applications and are highly portable among different platforms. Frequently, programmers struggle with displaying their results in a visually appealing manner, and an enormous amount of time may be spent on performing simple drawing tasks on screen. In this chapter, we will introduce a rapid prototyping approach to 2D data visualization using OpenGL so that impressive graphics can be created with minimal efforts. Most importantly, the proposed framework is highly intuitive and reusable, and it can be extended to be used in more sophisticated applications. Once you have mastered the basics of the OpenGL language, you will be equipped with the skills to create impressive applications that harness the true potential of OpenGL for data visualization using modern graphics hardware.

OpenGL primitives

In the simplest terms, primitives are just basic shapes that are drawn in OpenGL. In this section, we will provide a brief overview of the main geometric primitives that are supported by OpenGL and focus specifically on three commonly used primitives (which will also appear in our demo applications): points, lines, and triangles.

Drawing points

We begin with a simple, yet very useful, building block for many visualization problems: a point primitive. A point can be in the form of ordered pairs in 2D, or it can be visualized in the 3D space.

Getting ready

To simplify the workflow and improve the readability of the code, we first define a structure called `Vertex`, which encapsulates the fundamental elements such as the position and color of a vertex.

```
typedef struct
{
    GLfloat x, y, z; //position
    GLfloat r, g, b, a; //color and alpha channels
} Vertex;
```

Now, we can treat every object and shape in terms of a set of vertices (with a specific color) in space. In this chapter, as our focus is on 2D visualization, the z positions of vertices are often manually set to 0.0f.

We can create a vertex at the center of the screen (0, 0, 0) with a white color as an example:

```
Vertex v = {0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
```

Note that the color element consists of the red, green, blue, and alpha channels. These values range from 0.0 to 1.0. The alpha channel allows us to create transparency (0: fully transparent; 1: fully opaque) so that objects can be blended together.

How to do it...

We can first define a function called `drawPoint` to encapsulate the complexity of OpenGL primitive functions, illustrated as follows:

1. Create a function called `drawPoint` to draw points which takes in two parameters (the vertex and size of the point):

```
void drawPoint(Vertex v1, GLfloat size){
```

2. Specify the size of the point:

```
glPointSize(size);
```

3. Set the beginning of the list of vertices to be specified and indicate the primitive type associated with the vertices (`GL_POINTS` in this case):

```
glBegin(GL_POINTS);
```

4. Set the color and the vertex position using the fields from the `Vertex` structure:

```
glColor4f(v1.r, v1.g, v1.b, v1.a);  
glVertex3f(v1.x, v1.y, v1.z);
```

5. Set the end of the list:

```
glEnd();  
}
```

6. In addition, we can define a function called `drawPointsDemo` to encapsulate the complexity further. This function draws a series of points with an increasing size:

```
void drawPointsDemo(int width, int height){  
    GLfloat size=5.0f;  
    for(GLfloat x = 0.0f; x<=1.0f; x+=0.2f, size+=5)  
    {  
        Vertex v1 = {x, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
        drawPoint(v1, size);  
    }  
}
```

Finally, let's integrate these two functions into a complete OpenGL demo program (refer to identical steps in *Chapter 1, Getting Started withOpenGL*):

1. Create a source file called `main_point.cpp`, and then include the header file for the GLFW library and standard C++ libraries:

```
#include <GLFW/glfw3.h>  
#include <stdlib.h>  
#include <stdio.h>
```

2. Define the size of the window for display:

```
const int WINDOWS_WIDTH = 640*2;  
const int WINDOWS_HEIGHT = 480;
```

3. Define the `Vertex` structure and function prototypes:

```
typedef struct  
{  
    GLfloat x, y, z;  
    GLfloat r, g, b, a;
```

```
    } Vertex;
    void drawPoint(Vertex v1, GLfloat size);
    void drawPointsDemo(int width, int height);
```

4. Implement the `drawPoint` and `drawPointsDemo` functions, as shown previously.
5. Initialize GLFW and create a GLFW window object:

```
int main(void)
{
    GLFWwindow* window;
    if (!glfwInit())
        exit(EXIT_FAILURE);
    window = glfwCreateWindow(WINDOWS_WIDTH, WINDOWS_HEIGHT,
        "Chapter 2: Primitive drawings", NULL, NULL);
    if (!window){
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    glfwMakeContextCurrent(window);
```

6. Enable anti-aliasing and smoothing:

```
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

7. Define a loop that terminates when the window is closed. Set up the viewport (using the size of the window) and clear the color buffer at the beginning of each iteration to update with new content:

```
while (!glfwWindowShouldClose(window))
{
    float ratio;
    int width, height;
    glfwGetFramebufferSize(window, &width, &height);
    ratio = (float) width / (float)height;
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT);
```

8. Set up the camera matrix for orthographic projection:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
//Orthographic Projection
glOrtho(-ratio, ratio, -1.f, 1.f, 1.f, -1.f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

9. Call the `drawPointsDemo` function:

```
drawPointsDemo(width, height);
```

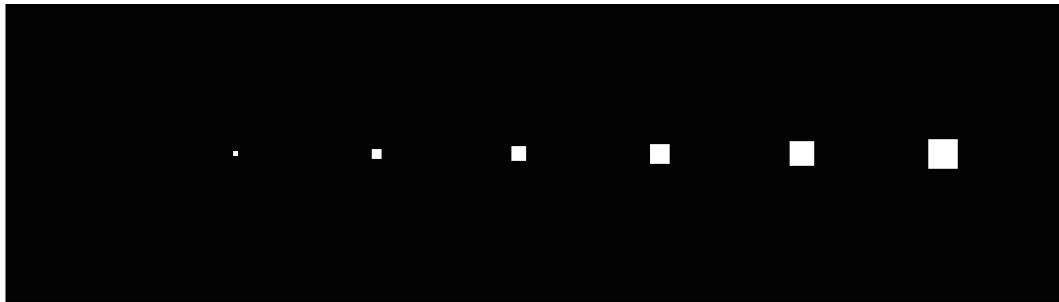
10. Swap the front and back buffers of the window and process the event queue (such as keyboard inputs) to avoid lock-up:

```
glfwSwapBuffers(window);  
glfwPollEvents();  
}
```

11. Release the memory and terminate the GLFW library. Then, exit the application:

```
glfwDestroyWindow(window);  
glfwTerminate();  
exit(EXIT_SUCCESS);  
}
```

Here is the result (with anti-aliasing disabled) showing a series of points with an increasing size (that is, the diameter of each point as specified by `glPointSize`):



How it works...

The `glBegin` and `glEnd` functions delimit the list of vertices corresponding to a desired primitive (`GL_POINTS` in this demo). The `glBegin` function accepts a set of symbolic constants that represent different drawing methods, including `GL_POINTS`, `GL_LINES`, and `GL_TRIANGLES`, as discussed in this chapter.

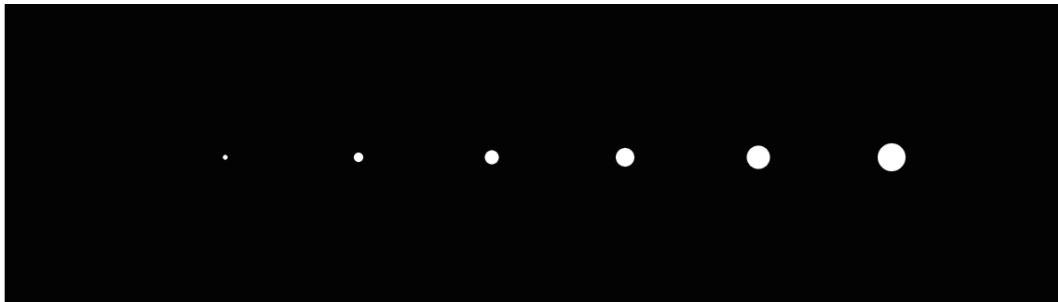
There are several ways to control the process of drawing points. First, we can set the diameter of each point (in pixels) with the `glPointSize` function. By default, a point has a diameter of 1 without anti-aliasing (a method to smooth sampling artifacts) enabled. Also, we can define the color of each point as well as the alpha channel (transparency) using the `glColor4f` function. The alpha channel allows us to overlay points and blend graphics elements. This is a powerful, yet very simple, technique used in graphics design and user interface design. Lastly, we define the position of the point in space with the `glVertex3f` function.

In OpenGL, we can define the projection transformation in two different ways: orthographic projection or perspective projection. In 2D drawing, we often use orthographic projection which involves no perspective correction (for example, the object on screen will remain the same size regardless of its distance from the camera). In 3D drawing, we use perspective projection to create more realistic-looking scenes similar to how the human eye sees. In the code, we set up an orthographic projection with the `glOrtho` function. The `glOrtho` function takes these parameters: the coordinates of the vertical clipping plane, the coordinates of the horizontal clipping plane, and the distance of the nearer and farther depth clipping planes. These parameters determine the projection matrix, and the detailed documentation can be found in <https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/glOrtho.3.html>.

Anti-aliasing and smoothing are necessary to produce the polished look seen in modern graphics. Most graphics cards support native smoothing and in OpenGL, it can be enabled as follows:

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

Here is the final result with anti-aliasing enabled, showing a series of circular points with an increasing size:



Note that in the preceding diagram, the points are now rendered as circles instead of squares with the anti-aliasing feature enabled. Readers are encouraged to disable and enable the features of the preceding diagram to see the effects of the operation.

See also

In this tutorial, we have focused on the C programming style due to its simplicity. In the upcoming chapters, we will migrate to an object-oriented programming style using C++. In addition, in this chapter, we focus on three basic primitives (and discuss the derivatives of these primitives where appropriate): `GL_POINTS`, `GL_LINES`, and `GL_TRIANGLES`. Here is a more extensive list of primitives supported by OpenGL (refer to <https://www.opengl.org/wiki/Primitive> for more information):

```
GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES,  
GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, and GL_  
POLYGON
```

Drawing line segments

One natural extension now is to connect a line between data points and then to connect the lines together to form a grid for plotting. In fact, OpenGL natively supports drawing line segments, and the process is very similar to that of drawing a point.

Getting ready

In OpenGL, we can simply define a line segment with a set of 2 vertices, and a line will be automatically formed between them by choosing `GL_LINES` as the symbolic constant in the `glBegin` statement.

How to do it...

Here, we define a new line drawing function called `drawLineSegment` which users can test by simply replacing the `drawPointsDemo` function in the previous section:

1. Define the `drawLineSegment` function which takes in two vertices and the width of the line as inputs:

```
void drawLineSegment(Vertex v1, Vertex v2, GLfloat width) {
```

2. Set the width of the line:

```
    glLineWidth(width);
```

3. Set the primitive type for line drawing:

```
    glBegin(GL_LINES);
```

4. Set the vertices and the color of the line:

```
glColor4f(v1.r, v1.g, v1.b, v1.a);  
glVertex3f(v1.x, v1.y, v1.z);  
glColor4f(v2.r, v2.g, v2.b, v2.a);  
glVertex3f(v2.x, v2.y, v2.z);  
glEnd();  
}
```

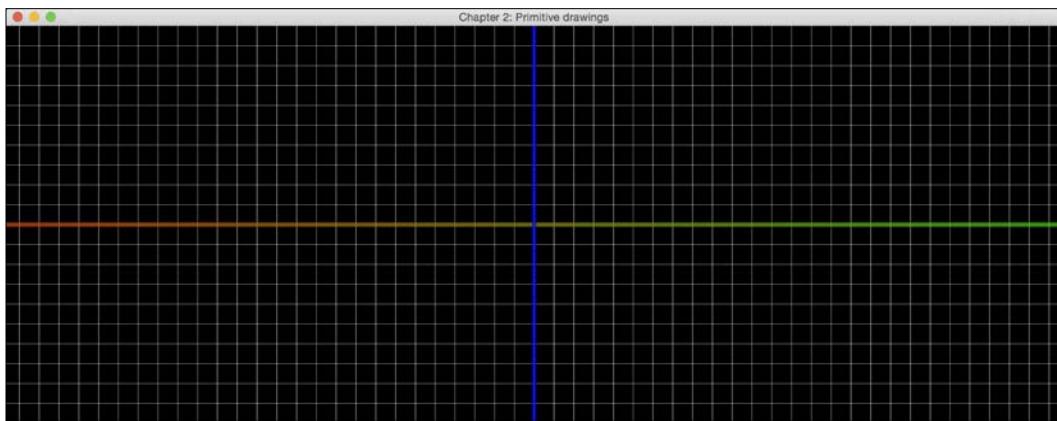
In addition, we define a new grid drawing function called `drawGrid`, built on top of the `drawLineSegment` function as follows:

```
void drawGrid(GLfloat width, GLfloat height, GLfloat grid_width){  
    //horizontal lines  
    for(float i=-height; i<height; i+=grid_width){  
        Vertex v1 = {-width, i, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
        Vertex v2 = {width, i, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
        drawLineSegment(v1, v2);  
    }  
    //vertical lines  
    for(float i=-width; i<width; i+=grid_width){  
        Vertex v1 = {i, -height, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
        Vertex v2 = {i, height, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
        drawLineSegment(v1, v2);  
    }  
}
```

Finally, we can execute the full demo by replacing the call for the `drawPointsDemo` function in the previous section with the following `drawLineDemo` function:

```
void drawLineDemo(){  
    //draw a simple grid  
    drawGrid(5.0f, 1.0f, 0.1f);  
    //define the vertices and colors of the line segments  
    Vertex v1 = {-5.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.7f};  
    Vertex v2 = {5.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.7f};  
    Vertex v3 = {0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.7f};  
    Vertex v4 = {0.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.7f};  
    //draw the line segments  
    drawLineSegment(v1, v2, 10.0f);  
    drawLineSegment(v3, v4, 10.0f);  
}
```

Here is a screenshot of the demo showing a grid with equal spacing and the x and y axes drawn with the line primitives:



How it works...

There are multiple ways of drawing line segments in OpenGL. We have demonstrated the use of `GL_LINES` which takes every consecutive pair of vertices in the list to form an independent line segment for each pair. On the other hand, if you would like to draw a line without gaps, you can use the `GL_LINE_STRIP` option, which connects all the vertices in a consecutive fashion. Finally, to form a closed loop sequence in which the endpoints of the lines are connected, you would use the `GL_LINE_LOOP` option.

In addition, we can modify the width and the color of a line with the `glLineWidth` and `glColor4f` functions for each vertex, respectively.

Drawing triangles

We will now move on to another very commonly used primitive, namely a triangle, which forms the basis for drawing all possible polygons.

Getting ready

Similar to drawing a line segment, we can simply define a triangle with a set of 3 vertices, and line segments will be automatically formed by choosing `GL_TRIANGLES` as the symbolic constant in the `glBegin` statement.

How to do it...

Finally, we define a new function called `drawTriangle`, which users can test by simply replacing the `drawPointsDemo` function. We will also reuse the `drawGrid` function from the previous section:

1. Define the `drawTriangle` function, which takes in three vertices as the input:

```
void drawTriangle(Vertex v1, Vertex v2, Vertex v3) {
```

2. Set the primitive type to draw triangles:

```
    glBegin(GL_TRIANGLES);
```

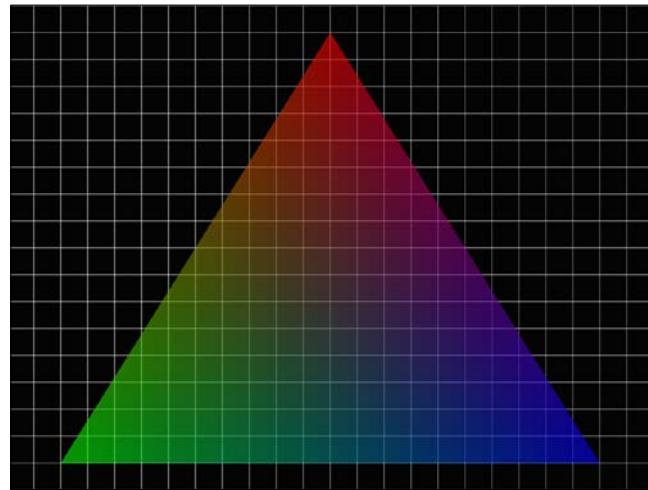
3. Set the vertices and the color of the triangle:

```
    glColor4f(v1.r, v1.g, v1.b, v1.a);
    glVertex3f(v1.x, v1.y, v1.z);
    glColor4f(v2.r, v2.g, v2.b, v2.a);
    glVertex3f(v2.x, v2.y, v2.z);
    glColor4f(v3.r, v3.g, v3.b, v3.a);
    glVertex3f(v3.x, v3.y, v3.z);
    glEnd(),
}
```

4. Execute the demo by replacing the call for the `drawPointsDemo` function in the full demo code with the following `drawTriangleDemo` function:

```
void drawTriangleDemo() {
    //Triangle Demo
    Vertex v1 = {0.0f, 0.8f, 0.0f, 1.0f, 0.0f, 0.0f, 0.6f};
    Vertex v2 = {-1.0f, -0.8f, 0.0f, 0.0f, 1.0f, 0.0f, 0.6f};
    Vertex v3 = {1.0f, -0.8f, 0.0f, 0.0f, 0.0f, 1.0f, 0.6f};
    drawTriangle(v1, v2, v3);
}
```

Here is the final result with a triangle rendered with 60 percent transparency overlaid on top of the grid lines:



How it works...

While the process of drawing a triangle in OpenGL appears similar to previous examples, there are some subtle differences and further complexities that can be incorporated. There are three different modes in this primitive (`GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN`), and each handles the vertices in a different manner. First, `GL_TRIANGLES` takes three vertices from a given list to create a triangle. The triangles are independently formed from each triplet of the vertices (that is, every three vertices are turned into a different triangle). On the other hand, `GL_TRIANGLE_STRIP` forms a triangle with the first three vertices, and each subsequent vertex forms a new triangle using the previous two vertices. Lastly, `GL_TRIANGLE_FAN` creates an arbitrarily complex convex polygon by creating triangles that have a common vertex in the center specified by the first vertex v_1 , which forms a fan-shaped structure consisting of triangles. In other words, triangles will be generated in the grouping order specified as follows:

$(v_1, v_2, v_3), (v_1, v_3, v_4), \dots, (v_1, v_{n-1}, v_n)$
for n vertices

Although a different color is set for each vertex, OpenGL handles color transition (linear interpolation) automatically, as shown in the triangle drawing in the previous example. The vertices are set to red, green, and blue, but the spectrum of colors can be clearly seen. Additionally, transparency can be set using the alpha channel, which enables us to clearly see the grid behind the triangle. With OpenGL, we can also add other elements, such as the advanced handling of color and shading, which will be discussed in the upcoming chapters.

Creating a 2D plot using primitives

Creating a 2D plot is a common way of visualizing trends in datasets in many applications. With OpenGL, we can render such plots in a much more dynamic way compared to conventional approaches (such as basic MATLAB plots) as we can gain full control over the graphics shader for color manipulation and we can also provide real-time feedback to the system. These unique features allow users to create highly interactive systems, so that, for example, time series such as an electrocardiogram can be visualized with minimal effort.

Here, we first demonstrate the visualization of a simple 2D dataset, namely a sinusoidal function in discrete time.

Getting ready

This demo requires a number of functions (including the `drawPoint`, `drawLineSegment`, and `drawGrid` functions) implemented earlier. In addition, we will reuse the code structure introduced in the *Chapter 1, Getting Started with OpenGL* to execute the demo.

How to do it...

We begin by generating a simulated data stream for a sinusoidal function over a time interval. In fact, the data stream can be any arbitrary signal or relationship:

1. Let's define an additional structure called `Data` to simplify the interface:

```
typedef struct
{
    GLfloat x, y, z;
} Data;
```

2. Define a generic 2D data plotting function called `draw2DscatterPlot` with the input data stream and number of points as the input:

```
void draw2DscatterPlot (const Data *data, int num_points){
```

3. Draw the x and y axes using the `drawLineSegment` function described earlier:

```
Vertex v1 = {-10.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
Vertex v2 = {10.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
drawLineSegment(v1, v2, 2.0f);
v1.x = 0.0f;
v2.x = 0.0f;
v1.y = -1.0f;
v2.y = 1.0f;
drawLineSegment(v1, v2, 2.0f);
```

4. Draw the data points one by one with the `drawPoint` function:

```
for(int i=0; i<num_points; i++) {
    GLfloat x=data[i].x;
    GLfloat y=data[i].y;
    Vertex v={x, y, 0.0f, 1.0f, 1.0f, 1.0f, 0.5f};
    drawPoint(v, 8.0f);
}
```

5. Create a similar function called `draw2DLineSegments` to connect the dots together with the line segments so that both the curve and the data points can be shown simultaneously:

```
void draw2DLineSegments(const Data *data, int num_points){
    for(int i=0; i<num_points-1; i++) {
        GLfloat x1=data[i].x;
        GLfloat y1=data[i].y;
        GLfloat x2=data[i+1].x;
        GLfloat y2=data[i+1].y;
        Vertex v1={x1, y1, 0.0f, 0.0f, 1.0f, 1.0f, 0.5f};
        Vertex v2={x2, y2, 0.0f, 0.0f, 1.0f, 0.0f, 0.5f};
        drawLineSegment(v1, v2, 4.0f);
    }
}
```

6. Integrate everything into a full demo by creating the grid, generating the simulated data points using a cosine function and plotting the data points:

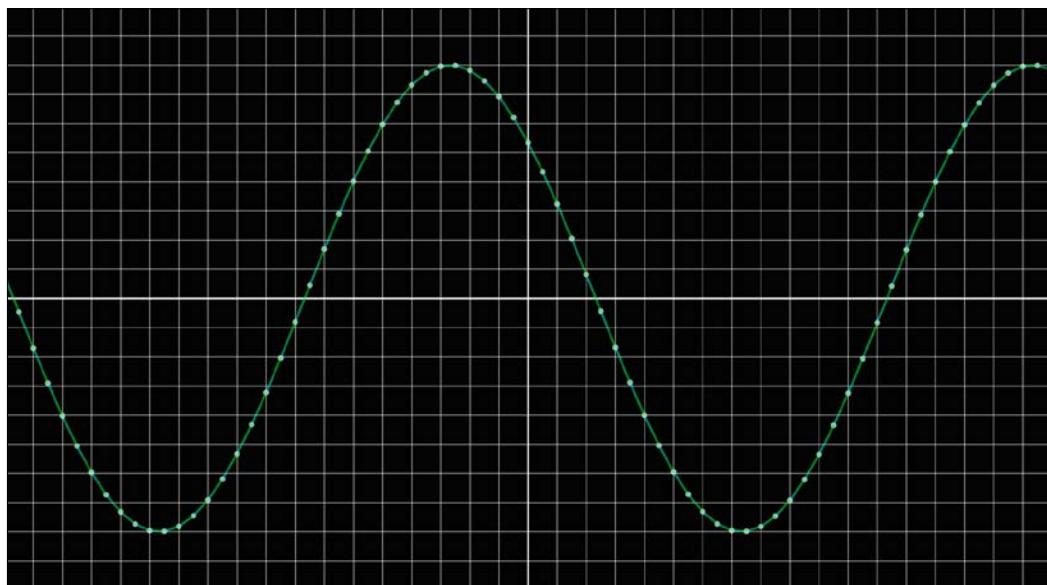
```
void linePlotDemo(float phase_shift) {
    drawGrid(5.0f, 1.0f, 0.1f);
    GLfloat range = 10.0f;
    const int num_points = 200;
    Data *data=(Data*)malloc(sizeof(Data)*num_points);
    for(int i=0; i<num_points; i++){
        data[i].x=((GLfloat)i/num_points)*range-range/2.0f;
        data[i].y= 0.8f*cosf(data[i].x*3.14f+phase_shift);
    }
    draw2DScatterPlot(data, num_points);
    draw2DLineSegments(data, num_points);
    free(data);
}
```

7. Finally, in the main program, include the `math.h` header file for the cosine function and add a new variable called `phase_shift` outside the loop to execute this demo. You can download the code package from Packt Publishing website for the complete demo code:

```
#include <math.h>
...
int main(void) {
    ...
    float phase_shift=0.0f;
    while (!glfwWindowShouldClose(window)) {
        ...
        phase_shift+=0.02f;
        linePlotDemo(phase_shift);
        ...

        //finished all demo calls
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    ...
}
```

The final result simulating a real-time input data stream with a sinusoidal shape is plotted on top of grid lines using a combination of basic primitives discussed in previous sections.



How it works...

Using the simple toolkit we created earlier using basic OpenGL primitives, we plotted a sinusoidal function with the data points (sampled at a constant time interval) overlaid on top of the curve. The smooth curve consists of many individual line segments drawn using the `draw2DlineSegments` function, while the samples were plotted using the `drawPoint` function. This intuitive interface serves as the basis for the visualization of more interesting time series for real-world applications in the next section.

Real-time visualization of time series

In this section, we further demonstrate the versatility of our framework to plot general time series data for biomedical applications. In particular, we will display an ECG in real time. As a brief introduction, an ECG is a very commonly used diagnostic and monitoring tool to detect abnormalities in the heart. ECG surface recording essentially probes the electrical activities of the heart. For example, the biggest spike (called a QRS complex) typically corresponds to the depolarization of the ventricles of the heart (the highly muscular chambers of the heart that pump blood). A careful analysis of the ECG can be a very powerful, noninvasive method for distinguishing many heart diseases clinically, including many forms of arrhythmia and heart attacks.

Getting ready

We begin by importing a computer-generated ECG data stream. The ECG data stream is stored in `data_ecg.h` (only a small portion of the data stream is provided here):

```
float data_ecg[]={0.396568808f, 0.372911844f, 0.311059085f, 0.220346775f,  
0.113525529f, 0.002200333f, -0.103284775f, -0.194218528f, -0.266285973f,  
-0.318075979f, -0.349670132f, -0.362640042f, -0.360047348f,  
-0.346207663f, -0.325440887f, -0.302062532f, -0.279400804f, -0.259695686f  
... };
```

How to do it...

1. Use the following code to plot the ECG data by drawing line segments:

```
void plotECGData(int offset, int size, float offset_y,  
    float scale){  
    //space between samples  
    const float space = 2.0f/size*ratio;  
    //initial position of the first vertex to render  
    float pos = -size*space/2.0f;  
    //set the width of the line  
    glLineWidth(5.0f);
```

```
glBegin(GL_LINE_STRIP);
//set the color of the line to green
	glColor4f(0.1f, 1.0f, 0.1f, 0.8f);
for (int i=offset; i<size+offset; i++) {
    const float data = scale*data_ecg[i]+offset_y;
    glVertex3f(pos, data, 0.0f);
    pos += space;
}
glEnd();
}
```

2. Display multiple ECG data streams (simulating recording from different leads):

```
void ecg_demo(int counter){
    const int data_size=ECG_DATA_BUFFER_SIZE;
    //Emulate the presence of multiple ECG leads (just for demo/
    //display purposes)
    plotECGData(counter, data_size*0.5, -0.5f, 0.1f);
    plotECGData(counter+data_size, data_size*0.5, 0.0f, 0.5f);
    plotECGData(counter+data_size*2, data_size*0.5, 0.5f, -0.25f);
}
```

3. Finally, in the main program, include the `data_ecg.h` header file and add the following lines of code to the loop. You can download the code package from the Packt Publishing website for the complete demo code:

```
#include "data_ecg.h"
...
int main(void) {
    ...
    while (!glfwWindowShouldClose(window)) {
        ...
        drawGrid(5.0f, 1.0f, 0.1f);
        //reset counter to 0 after reaching the end of the
        //sample data
        if(counter>5000) {
            counter=0;
        }
        counter+=5;
        //run the demo visualizer
        ecg_demo(counter);
        ...
    }
}
```

Here are two snapshots of the real-time display across multiple ECG leads simulated at two different time points. If you execute the demo, you will see the ECG recording from multiple leads move across the screen as the data stream is processed for display.



Here is the second snapshot at a later time point:



How it works...

This demo shows the use of the `GL_LINE_STRIP` option, described previously, to plot an ECG time series. Instead of drawing individual and independent line segments (using the `GL_LINE` option), we draw a continuous stream of data by calling the `glVertex3f` function for each data point. Additionally, the time series animates through the screen and provides dynamic updates on an interactive frame with minimal impact on the CPU computation cycles.

2D visualization of 3D/4D datasets

We have now learned multiple methods to generate plots on screen using points and lines. In the last section, we will demonstrate how to visualize a million data points in a 3D dataset using OpenGL in real time. A common strategy to visualize a complex 3D dataset is to encode the third dimension (for example, the z dimension) in the form of a heat map with a desirable color scheme. As an example, we show a heat map of a 2D Gaussian function with its height z , encoded using a simple color scheme. In general, a 2-D Gaussian function, $f(x,y)$, is defined as follows:

$$z = f(x, y) = Ae^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)}$$

Here, A is the amplitude ($1/(2\pi\sigma_x\sigma_y)$) of the distribution centered at (x_0, y_0) and σ_x, σ_y are the standard deviations (spread) of the distribution in the x and y directions. To make this demo more interesting and more visually appealing, we vary the standard deviation or sigma term (equally in the x and y directions) over time. Indeed, we can apply the same method to visualize very complex 3D datasets.

Getting ready

By now, you should be very familiar with the basic primitives described in previous sections. Here, we employ the `GL_POINTS` option to generate a dense grid of data points with different colors encoding the z dimension.

How to do it...

1. Generate a million data points (1,000 x 1,000 grid) with a 2-D Gaussian function:

```
void gaussianDemo(float sigma) {
    //construct a 1000x1000 grid
    const int grid_x = 1000;
    const int grid_y = 1000;
```

```
const int num_points = grid_x*grid_y;
Data *data=(Data*)malloc(sizeof(Data)*num_points);
int data_counter=0;
for(int x = -grid_x/2; x<grid_x/2; x+=1){
    for(int y = -grid_y/2; y<grid_y/2; y+=1){
        float x_data = 2.0f*x/grid_x;
        float y_data = 2.0f*y/grid_y;
        //compute the height z based on a
        //2D Gaussian function.
        float z_data = exp(-0.5f*(x_data*x_data)/(sigma*sigma)
            -0.5f*(y_data*y_data)/(sigma*sigma))/(
            (sigma*sigma*2.0f*M_PI));
        data[data_counter].x = x_data;
        data[data_counter].y = y_data;
        data[data_counter].z = z_data;
        data_counter++;
    }
}
//visualize the result using a 2D heat map
draw2DHeatMap(data, num_points);
free(data);
}
```

2. Draw the data points using a heat map function for color visualization:

```
void draw2DHeatMap(const Data *data, int num_points) {
    //locate the maximum and minimum values in the dataset
    float max_value=-999.9f;
    float min_value=999.9f;
    for(int i=0; i<num_points; i++){
        const Data d = data[i];
        if(d.z > max_value){
            max_value = d.z;
        }
        if(d.z < min_value){
            min_value = d.z;
        }
    }
    const float halfmax = (max_value + min_value) / 2;

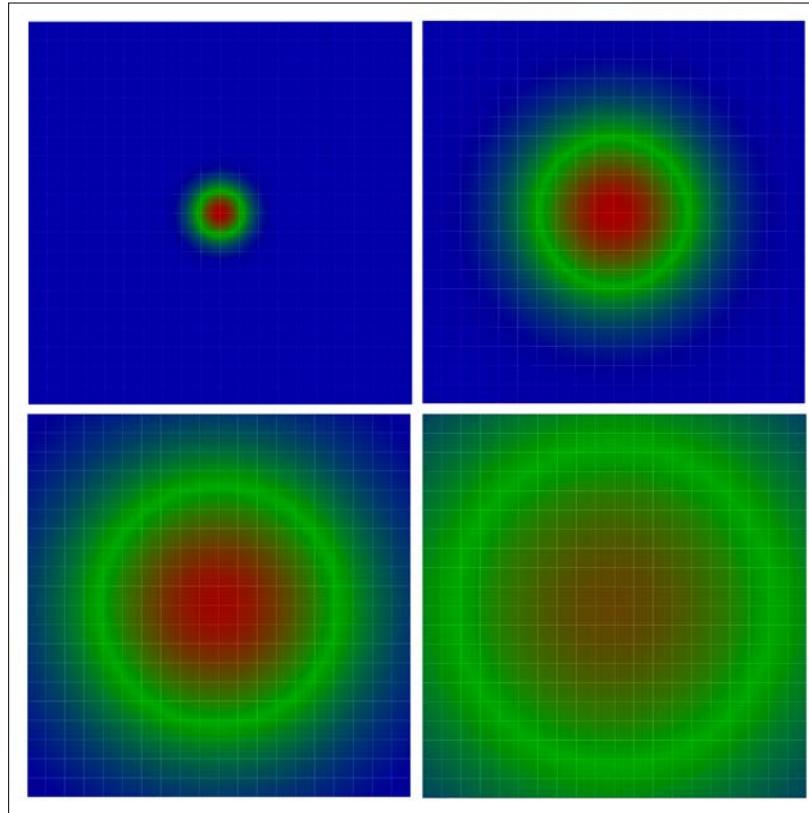
    //display the result
    glPointSize(2.0f);
    glBegin(GL_POINTS);
    for(int i = 0; i<num_points; i++) {
```

```
    const Data d = data[i];
    float value = d.z;
    float b = 1.0f - value/halfmax;
    float r = value/halfmax - 1.0f;
    if(b < 0) {
        b=0;
    }
    if(r < 0) {
        r=0;
    }
    float g = 1.0f - b - r;
    glColor4f(r, g, b, 0.5f);
    glVertex3f(d.x, d.y, 0.0f);
}
glEnd();
}
```

3. Finally, in the main program, include the `math.h` header file and add the following lines of code to the loop to vary the sigma term over time. You can download the example code from the Packt Publishing website for the complete demo code:

```
#define _USE_MATH_DEFINES // M_PI constant
#include <math.h>
...
int main(void){
    ...
    float sigma = 0.01f;
    while (!glfwWindowShouldClose(window)) {
        ...
        drawGrid(5.0f, 1.0f, 0.1f);
        sigma+=0.01f;
        if(sigma>1.0f)
            sigma=0.01;
        gaussianDemo(sigma);
        ...
    }
}
```

Here are four figures illustrating the effect of varying the sigma term of the 2-D Gaussian function over time (from 0.01 to 1):



How it works...

We have demonstrated how to visualize a Gaussian function using a simple heat map in which the maximum value is represented by red, while the minimum value is represented by blue. In total, a million data points ($1,000 \times 1,000$) were plotted using vertices for each Gaussian function with a specific sigma term. This sigma term was varied from 0.01 to 1 to show a time-varying Gaussian distribution. To reduce the overhead, vertex buffers can be implemented in the future (we can perform the memory copy operation all at once and remove the `glVertex3f` calls). Similar techniques can be applied to the color channel as well.

There's more...

The heat map we have described here provides a powerful visualization tool for complex 3D datasets seen in many scientific and biomedical applications. Indeed, we have actually extended our demo to the visualization of a 4D dataset, to be precise, since a time-varying 3D function; with the height encoded using a color map was displayed. This demo shows the many possibilities for displaying data in an interesting, dynamic way using just 2D techniques based on OpenGL primitives. In the next chapter, we will demonstrate the potential of OpenGL further by incorporating 3D rendering and adding user inputs to enable the 3D, interactive visualization of more complex datasets.

3

Interactive 3D Data Visualization

In this chapter, we will cover the following topics:

- ▶ Setting up a virtual camera for 3D rendering
- ▶ Creating a 3D plot with perspective rendering
- ▶ Creating an interactive environment with GLFW
- ▶ Rendering a volumetric dataset – MCML simulation

Introduction

OpenGL is a very attractive platform for creating dynamic, highly interactive tools for visualizing data in 3D. In this chapter, we will build upon the fundamental concepts discussed in the previous chapter and extend our demos to incorporate more sophisticated OpenGL features for 3D rendering. To enable 3D visualization, we will first introduce the basic steps of setting up a virtual camera in OpenGL. In addition, to create more interactive demos, we will introduce the use of GLFW callback functions for handling user inputs. Using these concepts, we will illustrate how to create an interactive 3D plot with perspective rendering using OpenGL. Finally, we will demonstrate how to render a 3D volumetric dataset generated from a Monte Carlo simulation of light transport in biological tissue. By the end of this chapter, readers will be able to visualize data in 3D with perspective rendering and interact with the environment dynamically through user inputs for a wide range of applications.

Setting up a virtual camera for 3D rendering

Rendering a 3D scene is similar to taking a photograph with a digital camera in the real world. The steps that are taken to create a photograph can also be applied in OpenGL.

For example, you can move the camera from one position to another and adjust the viewpoint freely in space, which is known as **viewing transformation**. You can also adjust the position and orientation of the object of interest in the scene. However, unlike in the real world, in the virtual world you can position the object at any orientation freely without any physical constraints, termed as **modeling transformation**. Finally, we can exchange camera lenses to adjust the zoom and create different perspectives the process is called **projection transformation**.

When you take a photo applying the viewing and modeling transformation, the digital camera takes the information and creates an image on your screen. This process is called **rasterization**.

These sets of matrices—encompassing the viewing transformation, modeling transformation, and projection transformation—are the fundamental elements we can adjust at run-time, which allows us to create an interactive and dynamic rendering of the scene. To get started, we will first look into the setup of the camera matrix, and how we can create a scene with different perspectives.

Getting ready

The source code in this chapter is based on the final demo from the previous chapter. Basically, we will be modifying the previous implementation by setting up a camera model using a perspective matrix. In the upcoming chapters, we will explore the use of the **OpenGL Shading Language (GLSL)** to enable even more complex rendering techniques and higher performance.

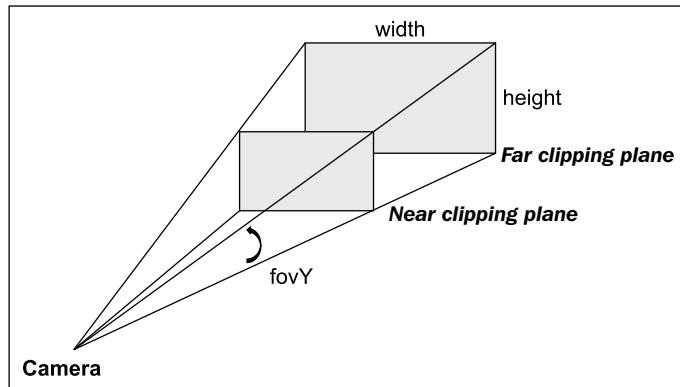
How to do it...

Let's get started on the first new requirement for handling perspective transformation in OpenGL. Since the camera parameters depend on the window size, we need to first implement a callback function that handles a window resize event and updates the matrices accordingly:

1. Define the function prototype for the callback function:

```
void framebuffer_size_callback(GLFWwindow* window, int width,  
                             int height)  
{
```

2. Preset the camera parameters: the vertical **field of view angle (fovY)**, the distance to the **Near clipping plane** (front), the distance to **Far clipping plane** (back), and the screen aspect ratio (**width/height**):



```
const float fovY = 45.0f;
const float front = 0.1f;
const float back = 128.0f;
float ratio = 1.0f;
if (height > 0)
    ratio = (float) width / (float) height;
```

3. Set up the viewport of the virtual camera (using the window size):

```
glViewport(0, 0, width, height);
```

4. Specify the matrix mode as **GL_PROJECTION** and allow subsequent matrix operations to be applied to the projection matrix stack:

```
glMatrixMode(GL_PROJECTION);
```

5. Load the identity matrix to the current matrix (that is, reset the matrix to its default state):

```
glLoadIdentity();
```

6. Set up the perspective projection matrix for the virtual camera:

```
    gluPerspective(fovY, ratio, front, back);
}
```

How it works...

The purpose of the `framebuffer_size_callback` function is to handle callback events from the GLFW library. Upon resizing the window, an event will be captured and the callback function provides a mechanism to update the virtual camera parameters accordingly. One important problem is that changing the aspect ratio of the screen can introduce distortion if we do not adjust our virtual camera rendering parameters appropriately. Therefore, the `update` function also calls the `glViewport` function to ensure that the graphic is rendered onto the new viewable area.

Furthermore, imagine we are taking a photo of a scene with a camera physically in the real world. The `gluPerspective` function basically controls the camera lens' zoom (that is, the field of view angle) as well as the camera sensor (that is, the image plane) aspect ratio. One major difference between the virtual and real camera is the concept of a near clipping and far clipping plane (front and back variables) that limits the viewable area of the rendered image. These constraints are related to more advanced topics (the depth buffer and depth testing) and how the graphical engine works with a virtual 3D scene. One rule of thumb is, we should never set an unnecessarily large value as it will affect the precision of the depth testing result, which can lead to z-fighting issue. **Z-fighting** is a phenomenon that occurs when objects share very similar depth values and the precision of the depth value is not sufficient to resolve the ambiguity (due to precision loss in the floating-point representation during the 3D rendering process). Setting a higher resolution depth buffer, or reducing the distance between the clipping planes, is often the simplest way to mitigate such problems.

The sample code provides perspective rendering of a scene that resembles how the human eye sees the world. For example, an object will appear larger if it is closer to the camera and smaller if it is farther away. This allows for a more realistic view of a scene. On the other hand, by controlling the field of view angle, we can exaggerate perspective distortion, similar to capturing a scene with an ultra-wide angle lens.

There's more...

Alternatively, we can set up the camera with the `glFrustum()` function by replacing the `gluPerspective()` function with the following code:

```
const double DEG2RAD = 3.14159265 / 180;
// tangent of half fovY
double tangent = tan(fovY/2 * DEG2RAD);
// half height of near plane
double height_f = front * tangent;
// half width of near plane
double width_f = height_f * ratio;
```

```
//Create the projection matrix based on the near clipping
//plane and the location of the corners
glFrustum(-width_f, width_f, -height_f, height_f, front, back);
}
```

The `glFrustum` function takes the corners of the near clipping and far clipping planes to construct the projective matrix. Fundamentally, there is no difference between the `gluPerspective` and `glFrustum` functions, so they are interchangeable.

As we can see, the virtual camera in OpenGL can be updated upon changes to the screen frame buffer (window size) and these event updates are captured with the callback mechanism of the GLFW library. Of course, we can also handle other events such as keyboard and mouse inputs. Further details on how to handle additional events will be discussed later. In the next section, let's implement the rest of the demo to create our first 3D plot with perspective rendering.

Creating a 3D plot with perspective rendering

In the previous chapter, we showed a heat map of a 2D Gaussian distribution with varying standard deviation over time. Now, we will continue with more advanced rendering of the same dataset in 3D and demonstrate the effectiveness of visualizing multi-dimensional data with OpenGL. The code base from the previous chapter will be modified to enable 3D rendering.

Instead of rendering the 2D Gaussian distribution function on a plane, we take the output of the Gaussian function $f(x, y)$ as the z (height) value as follows:

$$z = f(x, y) = Ae^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)}$$

Here A is the amplitude of the distribution centered at (x_0, y_0) , and σ_x, σ_y are the standard deviations (spread) of the distribution in the x and y directions. In our example, we will vary the spread of the distribution over time to change its shape in 3D. Additionally, we will apply a heat map to each vertex based on the height for better visualization effect.

Getting ready

With the camera set up using the projection model, we can render our graph in 3D with the desired effects by changing some of the virtual camera parameters such as the field of view angle for perspective distortion as well as the rotation angles for different viewing angles. To reduce coding complexity, we will re-use the `draw2DHeatMap` and `gaussianDemo` functions implemented in *Chapter 2, OpenGL Primitives and 2D Data Visualization* with minor modifications. The rendering techniques will be based on the OpenGL primitives described in the previous chapter.

How to do it...

Let's modify the final demo in *Chapter 2, OpenGL Primitives and 2D Data Visualization* (`main_gaussian_demo.cpp` in the code package) to enable perspective rendering in 3D. The overall code structure is provided here to orient readers first and major changes will be discussed in smaller blocks sequentially:

```
#include <GLFW/glfw3.h>
...
// Window size
const int WINDOWS_WIDTH = 1280;
const int WINDOWS_HEIGHT = 720;

// NEW: Callback functions and helper functions for 3D plot
void framebuffer_size_callback(GLFWwindow* window, int width,
    int height);
void draw2DHeatMap(const Data *data, int num_points);
void gaussianDemo(float sigma);
...

int main(void)
{
    GLFWwindow* window;
    int width, height;
    if (!glfwInit()){
        exit(EXIT_FAILURE);
    }
    window = glfwCreateWindow(WINDOWS_WIDTH, WINDOWS_HEIGHT,
        "Chapter 3: 3D Data Plotting", NULL, NULL);
    if (!window){
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
```

```
glfwMakeContextCurrent(window) ;
glfwSwapInterval(1) ;
// NEW: Callback functions
...

//enable anti-aliasing
glEnable(GL_BLEND) ;
//smooth the points
glEnable(GL_LINE_SMOOTH) ;
//smooth the lines
glEnable(GL_POINT_SMOOTH) ;
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST) ;
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST) ;
//needed for alpha blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) ;
glEnable(GL_ALPHA_TEST) ;
// NEW: Initialize parameters for perspective rendering
...
while (!glfwWindowShouldClose(window))
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) ;
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f) ;
    // NEW: Perspective rendering
    ...
}
glfwDestroyWindow(window) ;
glfwTerminate() ;
exit(EXIT_SUCCESS) ;
}
```

With the preceding framework in mind, inside the main function let's add the new callback function for handling window resizing implemented in the previous section:

```
glfwGetFramebufferSize(window, &width, &height) ;
framebuffer_size_callback(window, width, height) ;
```

Let's define several global variables and initialize them for perspective rendering, including the zoom level (zoom) and rotation angles around the x (beta) and z (alpha) axes, respectively:

```
GLfloat alpha=210.0f, beta=-70.0f, zoom=2.0f;
```

In addition, outside the `main` loop, let's initialize some parameters for rendering the Gaussian distribution, including the standard deviation (`sigma`), sign, and step size for dynamically changing the function over time:

```
float sigma = 0.1f;  
float sign = 1.0f;  
float step_size = 0.01f;
```

In the `while` loop, we perform the following transformations to render the Gaussian function in 3D:

1. Specify the matrix mode as `GL_MODELVIEW` to allow subsequent matrix operations to be applied to the `MODELVIEW` matrix stack:

```
glMatrixMode(GL_MODELVIEW);
```

2. Perform the translation and rotation of the object:

```
glLoadIdentity();  
glTranslatef(0.0, 0.0, -2.0);  
// rotate by beta degrees around the x-axis  
glRotatef(beta, 1.0, 0.0, 0.0);  
// rotate by alpha degrees around the z-axis  
glRotatef(alpha, 0.0, 0.0, 1.0);
```

3. Draw the origin (with the `x`, `y`, and `z` axes) and the Gaussian function in 3D. Dynamically plot a series of Gaussian functions with varying `sigma` values over time and reverse the sign once a certain threshold is reached:

```
drawOrigin();  
sigma=sigma+sign*step_size;  
if(sigma>1.0f){  
    sign = -1.0f;  
}  
if(sigma<0.1){  
    sign = 1.0f;  
}  
gaussianDemo(sigma);
```

For handling each of the preceding drawing tasks, we implement the origin visualizer, Gaussian function generator, and 3D point visualizer in separate functions.

To visualize the origin, implement the following drawing function:

1. Define the function prototype:

```
void drawOrigin(){
```

2. Draw the x, y, and z axes in red, green, and blue, respectively:

```
glLineWidth(4.0f);
glBegin(GL_LINES);
float transparency = 0.5f;

//draw a red line for the x-axis
glColor4f(1.0f, 0.0f, 0.0f, transparency);
glVertex3f(0.0f, 0.0f, 0.0f);
glColor4f(1.0f, 0.0f, 0.0f, transparency);
glVertex3f(0.3f, 0.0f, 0.0f);

//draw a green line for the y-axis
glColor4f(0.0f, 1.0f, 0.0f, transparency);
glVertex3f(0.0f, 0.0f, 0.0f);
glColor4f(0.0f, 1.0f, 0.0f, transparency);
glVertex3f(0.0f, 0.0f, 0.3f);

//draw a blue line for the z-axis
glColor4f(0.0f, 0.0f, 1.0f, transparency);
glVertex3f(0.0f, 0.0f, 0.0f);
glColor4f(0.0f, 0.0f, 1.0f, transparency);
glVertex3f(0.0f, 0.3f, 0.0f);
glEnd();
}
```

For the implementation of the Gaussian function demo, we have broken down the problem into two parts: a Gaussian data generator and a heat map visualizer function with point drawing. Together with 3D rendering and the heat map, we can now clearly see the shape of the Gaussian distribution and how the samples animate and move in space over time:

1. Generate the Gaussian distribution:

```
void gaussianDemo(float sigma) {
    const int grid_x = 400;
    const int grid_y = 400;
    const int num_points = grid_x*grid_y;
    Data *data=(Data*)malloc(sizeof(Data)*num_points);
    int data_counter=0;

    //standard deviation
    const float sigma2=sigma*sigma;
    //amplitude
    const float sigma_const = 10.0f*(sigma2*2.0f*(float)M_PI);

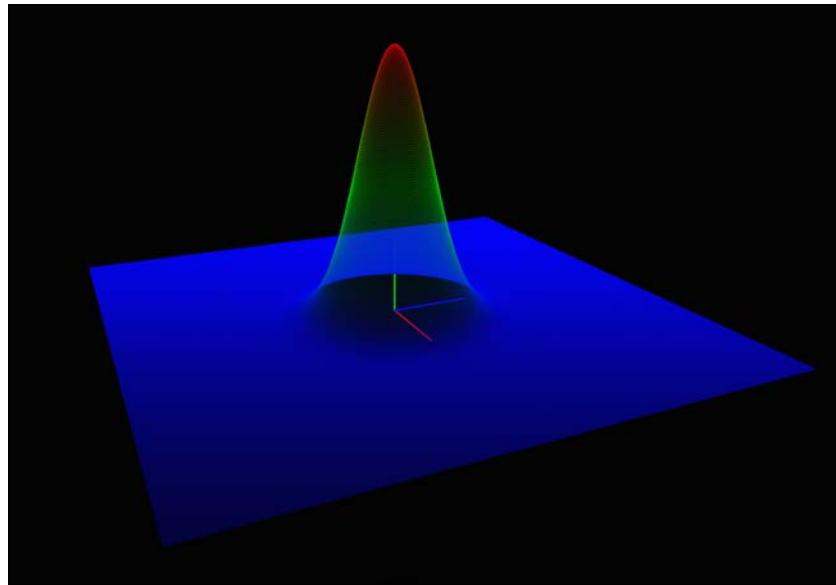
    for(float x = -grid_x/2.0f; x<grid_x/2.0f; x+=1.0f){
        for(float y = -grid_y/2.0f; y<grid_y/2.0f; y+=1.0f){
            float x_data = 2.0f*x/grid_x;
            float y_data = 2.0f*y/grid_y;
            //Set the mean to 0
```

```
        float z_data = exp(-0.5f*(x_data*x_data)/(sigma2)
            - 0.5f*(y_data*y_data)/(sigma2)) /sigma_const;
        data[data_counter].x = x_data;
        data[data_counter].y = y_data;
        data[data_counter].z = z_data;
        data_counter++;
    }
}
draw2DHeatMap(data, num_points);
free(data);
}
```

2. Next, implement the `draw2DHeatMap` function to visualize the result. Note that, unlike in *Chapter 2, OpenGL Primitives and 2D Data Visualization*, we use the `z` value inside the `glVertex3f` function:

```
void draw2DHeatMap(const Data *data, int num_points) {
    glPointSize(3.0f);
    glBegin(GL_POINTS);
    float transparency = 0.25f;
    //locate the maximum and minimum values in the dataset
    float max_value=-999.9f;
    float min_value=999.9f;
    for(int i=0; i<num_points; i++){
        Data d = data[i];
        if(d.z > max_value)
            max_value = d.z;
        if(d.z < min_value)
            min_value = d.z;
    }
    float halfmax = (max_value + min_value) / 2;
    //display the result
    for(int i = 0; i<num_points; i++){
        Data d = data[i];
        float value = d.z;
        float b = 1.0f - value/halfmax;
        float r = value/halfmax - 1.0f;
        if(b < 0)
            b=0;
        if(r < 0)
            r=0;
        float g = 1.0f - b - r;
        glColor4f(r, g, b, transparency);
        glVertex3f(d.x, d.y, d.z);
    }
    glEnd();
}
```

The rendered result is shown in the following screenshot. We can see that the transparency (alpha blending) allows us to see through the data points and provides a visually appealing result:



How it works...

This simple example demonstrates the use of perspective rendering as well as OpenGL transformation functions to rotate and translate the rendered objects in virtual space. As you can see, the overall code structure remains the same as in *Chapter 2, OpenGL Primitives and 2D Data Visualization* and the major changes primarily include setting up the camera parameters for perspective rendering (inside the `framebuffer_size_callback` function) and performing the required transformations to render the Gaussian function in 3D (after setting the matrix mode to `GL_MODELVIEW`). Two very commonly used transformation functions to manipulate virtual objects include `glRotatef` and `glTranslatef`, which allow us to position objects at any orientation and position. These functions can significantly improve the dynamics of your own application, with very minimal cost in development and computation time since they are heavily optimized.

The `glRotatef` function takes four parameters: the rotation angle and three components of the direction vector (x, y, z), which define the axis of rotation. The function also replaces the current matrix with the product of the rotation matrix and the current matrix:

$$\begin{bmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here $c = \cos(\text{angle})$, $s = \sin(\text{angle})$ and $\|(x, y, z)\|=1$.

There's more...

One may ask, what if we would like to position two objects at different orientations and positions? What if we would like to position many parts in space relative to one another? The answer to these is to use the `glPushMatrix` and `glPopMatrix` functions to control the stack of transformation matrices. The concept behind this can get relatively complex for a model with a large number of parts and keeping a history of the state machine with many components can be tedious. To address this issue, we will look into newer versions of GLSL support (OpenGL 3.x and higher).

Creating an interactive environment with GLFW

In the previous two sections, we focused on the creation of 3D objects and on utilizing basic OpenGL rendering techniques with a virtual camera. Now, we are ready to incorporate user inputs, such as mouse and keyboard inputs, to enable more dynamic interactions using camera control features such as zoom and rotate. These features will be the fundamental building blocks for the upcoming applications and the code will be reused in later chapters.

Getting ready

The GLFW library provides a mechanism to handle user inputs from different environments. The event handlers are implemented as callback functions in C/C++, and, in the previous tutorials, we bypassed these options for the sake of simplicity. To get started, we first need to enable these callback functions and implement basic features to control the rendering parameters.

How to do it...

To handle keyboard inputs, we attach our own implementation of the callback functions back to the event handler of GLFW. We will perform the following operations in the callback function:

1. Define the following global variables (including a new variable called `locked` to track whether the mouse button is pressed down, as well as the angles of rotation and zoom level) that will be updated by the callback functions:

```
GLboolean locked = GL_FALSE;  
GLfloat alpha=210.0f, beta=-70.0f, zoom=2.0f;
```

2. Define the keyboard callback function prototype:

```
void key_callback(GLFWwindow* window, int key, int scancode,  
                  int action, int mods)  
{
```

3. If we receive any event other than the key press event, ignore it:

```
if (action != GLFW_PRESS)  
    return;
```

4. Create a switch statement to handle each key press case:

```
switch (key)  
{
```

5. If the *Esc* key is pressed, exit the program:

```
case GLFW_KEY_ESCAPE:  
    glfwSetWindowShouldClose(window, GL_TRUE);  
    break;
```

6. If the space bar is pressed, start or stop the animation by toggling the variable:

```
case GLFW_KEY_SPACE:  
    freeze=!freeze;  
    break;
```

7. If the direction keys (up, down, left, and right) are pressed, update the variables that control the angles of rotation for the rendered object:

```
case GLFW_KEY_LEFT:  
    alpha += 5.0f;  
    break;  
case GLFW_KEY_RIGHT:  
    alpha -= 5.0f;  
    break;  
case GLFW_KEY_UP:
```

```
        beta -= 5.0f;
        break;
    case GLFW_KEY_DOWN:
        beta += 5.0f;
        break;
```

8. Lastly, if the *Page Up* or *Page Down* keys are pressed, zoom in and out from the object by updating the `zoom` variable:

```
case GLFW_KEY_PAGE_UP:
    zoom -= 0.25f;
    if (zoom < 0.0f)
        zoom = 0.0f;
    break;
case GLFW_KEY_PAGE_DOWN:
    zoom += 0.25f;
    break;
default:
    break;
}
}
```

To handle mouse click events, we implement another `callback` function similar to the one for the keyboard. The mouse click event is rather simple as there is only a limited set of buttons available:

1. Define the mouse press `callback` function prototype:

```
void mouse_button_callback(GLFWwindow* window, int button,
                           int action, int mods)
{
```

2. Ignore all inputs except for the left click event for simplicity:

```
if (button != GLFW_MOUSE_BUTTON_LEFT)
    return;
```

3. Toggle the `lock` variable to store the mouse hold event. The `lock` variable will be used to determine whether the mouse movement is used for rotating the object:

```
if (action == GLFW_PRESS)
{
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
    locked = GL_TRUE;
}
else
{
```

```
    locked = GL_FALSE;
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
}
}
```

For handling mouse movement events, we need to create another callback function. The callback function for mouse movement takes the x and y coordinates from the window instead of unique key inputs:

1. Define the callback function prototype that takes in the mouse coordinates:

```
void cursor_position_callback(GLFWwindow* window, double x,
    double y)
{
```

2. Upon mouse press and mouse movement, we update the rotation angles of the object with the x and y coordinates of the mouse:

```
//if the mouse button is pressed
if (locked)
{
    alpha += (GLfloat) (x - cursorX) / 10.0f;
    beta += (GLfloat) (y - cursorY) / 10.0f;
}
//update the cursor position
cursorX = (int) x;
cursorY = (int) y;
}
```

Finally, we will implement the mouse scroll callback function, which allows users to scroll up and down to zoom in and zoom out of the object.

1. Define the callback function prototype that captures the x and y scroll variables:

```
void scroll_callback(GLFWwindow* window, double x, double y)
{
```

2. Take the y parameter (up and down scroll) and update the zoom variable:

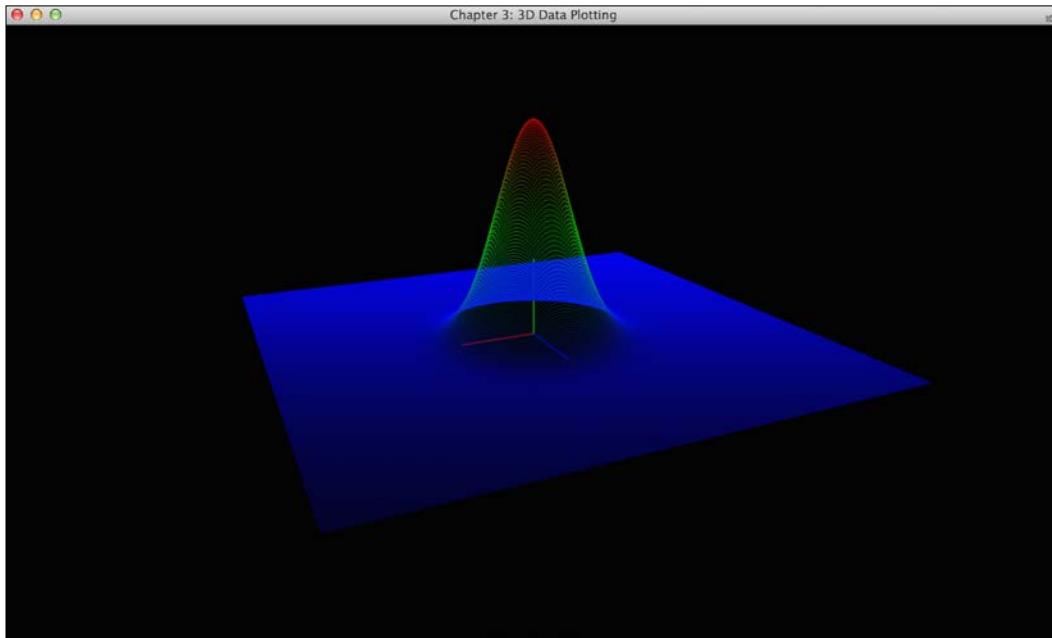
```
    zoom += (float) y / 4.0f;
    if (zoom < 0.0f)
        zoom = 0.0f;
}
```

With all of the callback functions implemented, we are now ready to link these functions to the GLFW library event handlers. The GLFW library provides a platform-independent API for handling each of these events, so the same code will run in Windows, Linux, and Mac OS X seamlessly.

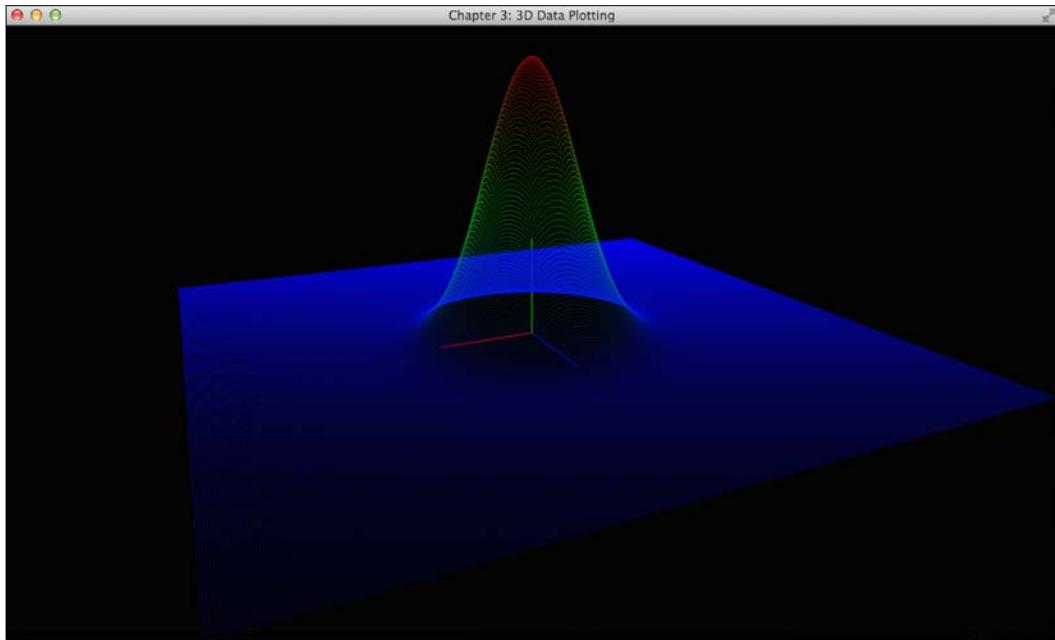
To integrate the callbacks with the GLFW library, call the following functions in the main function:

```
//keyboard input callback  
glfwSetKeyCallback(window, key_callback);  
  
//framebuffer size callback  
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);  
  
//mouse button callback  
glfwSetMouseButtonCallback(window, mouse_button_callback);  
  
//mouse movement callback  
glfwSetCursorPosCallback(window, cursor_position_callback);  
  
//mouse scroll callback  
glfwSetScrollCallback(window, scroll_callback);
```

The end result is an interactive interface that allows the user to control the rendering object freely in space. First, when the user scrolls the mouse (see the following screenshots), we translate the object forward or backward. This creates the visual perception that the object is zoomed in or zoomed out of the camera:

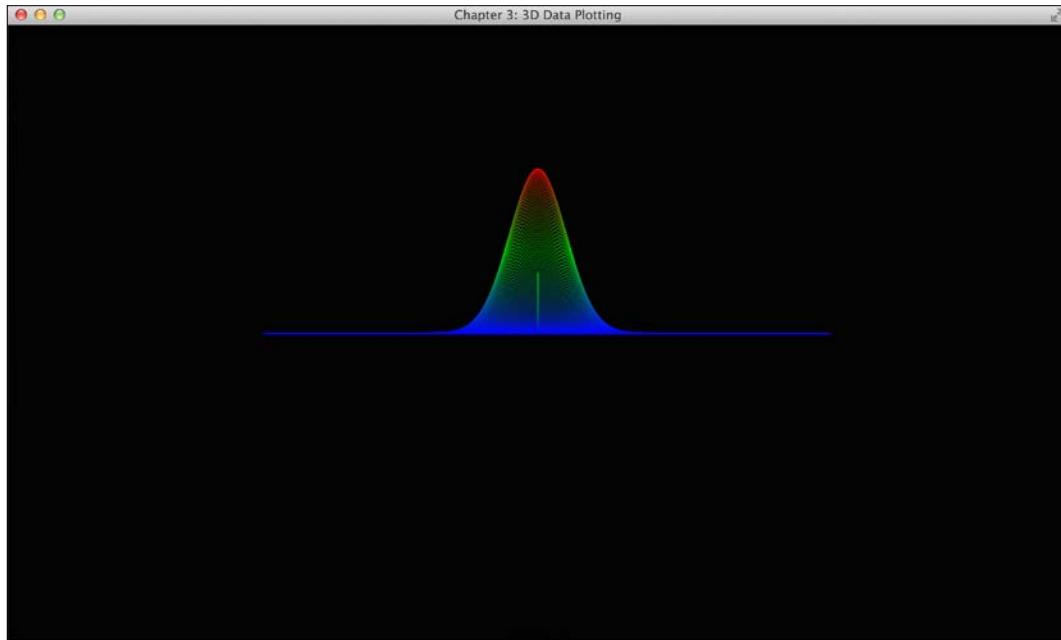


Here is another screenshot at a different zoom level:

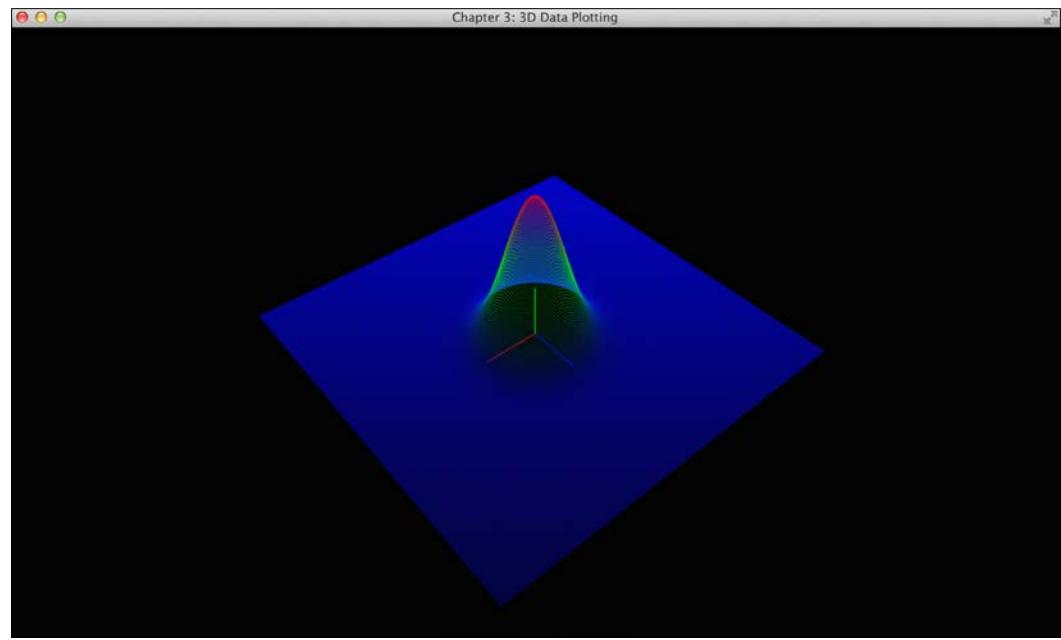


These simple yet powerful techniques allow users to manipulate virtual objects in real-time and can be extremely useful when visualizing complex datasets. Additionally, we can rotate the object at different angles by holding the mouse button and dragging the object in various directions. The screenshots below show how we can render the graph at any arbitrary angle to better understand the data distribution.

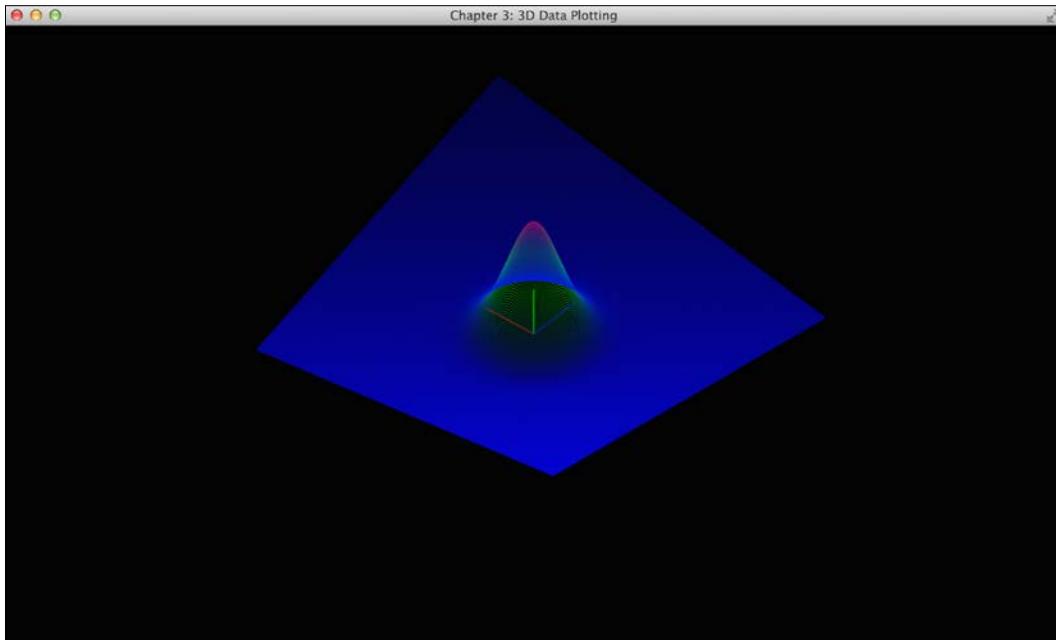
Here is a screenshot showing the side view of the Gaussian function:



Here is a screenshot showing the Gaussian function from the top:



Finally, here is a screenshot showing the Gaussian function from the bottom:



How it works...

This sample code illustrates the basic interface needed to build interactive applications that are highly portable across multiple platforms using OpenGL and the GLFW library. The use of callback functions in the GLFW library allows non-blocking calls that run in parallel with the rendering engine. This concept is particularly useful since input devices such as the mouse, keyboard, and joysticks all have different input rates and latency. These callback functions allow for asynchronous execution without blocking the main rendering loop.

The `glfwSetKeyCallback`, `glfwSetFramebufferSizeCallback`, `glfwSetScrollCallback`, `glfwSetMouseButtonCallback`, and `glfwSetCursorPosCallback` functions provide controls over the mouse buttons and scrolling wheel, keyboard inputs, and window resizing events. These are only some of the many handlers we can implement with the GLFW library support. For example, we can further extend the error handling capabilities by adding additional callback functions. Also, we can handle window closing and opening events, thereby enabling even more sophisticated interfaces with multiple windows. With the examples provided thus far, we have introduced the basics of how to create interactive interfaces with relatively simple API calls.

See also

For complete coverage of GLFW library function calls, this website provides a comprehensive set of examples and documentation for all callback functions as well as the handling of inputs and other events: <http://www.glfw.org/docs/latest/>.

Rendering a volumetric dataset – MCML simulation

In this section, we will demonstrate the rendering of a 3D volumetric dataset generated from a Monte Carlo simulation of light transport in biological tissue, called **Monte Carlo for multi-layered media (MCML)**. For simplicity, the simulation output file is included with the code bundle for this chapter so that readers can directly run the demo without setting up the simulation code. The source code for the Monte Carlo simulation is described in detail in a series of publications listed in the See also section and the GPU implementation is available online for interested readers (<https://code.google.com/p/gpumcml/>).

Light transport in biological tissue can be modeled with the **radiative transport equation (RTE)**, which has proven difficult to solve analytically for complex geometry. The time-dependent RTE can be expressed as:

$$\frac{1}{v} \frac{\partial}{\partial t} L(\mathbf{r}, \Omega, t) + \Omega \cdot \nabla L(\mathbf{r}, \Omega, t) + [\mu_a(\mathbf{r}) + \mu_s(\mathbf{r})] L(\mathbf{r}, \Omega, t) = \int_{4\pi} L(\mathbf{r}, \Omega', t) \mu_s(\mathbf{r}, \Omega' \rightarrow \Omega) d\Omega' + S(\mathbf{r}, \Omega, t)$$

Here $L(r, \Omega, t)$ is the radiance [$W m^{-2}sr^{-1}$] defined as the radiant power [W] crossing an infinitesimal area at location r perpendicular to the direction Ω per unit solid angle, μ_s is the scattering coefficient, μ_a is the absorption coefficient, v is the speed of light, and $S(r, \Omega, t)$ is the source term. To solve the RTE numerically, Wilson and Adam introduced the **Monte Carlo (MC)** method, which is widely accepted as a gold-standard approach for photon migration modeling due to its accuracy and versatility (especially for complex tissue geometry).

The MC method is a statistical sampling technique that has been applied to a number of important problems in many different fields, ranging from radiation therapy planning in medicine to option pricing in finance. The name Monte Carlo is derived from the resort city in Monaco that is known for its casinos, among other attractions. As its name implies, the key feature of the MC method involves the exploitation of random chance (through the generation of random numbers with a particular probability distribution) to model the physical process in question.

In our case, we are interested in modeling photon propagation in biological tissue. The MCML algorithm provides an MC model of steady-state light transport in multi-layered media. In particular, we will simulate photon propagation in a homogeneous medium with a circular light source incident on the tissue surface in order to compute the light dose (absorbed energy) distribution. Such computations have a wide range of applications, including treatment planning for light therapies such as photodynamic therapy (this can be considered a light-activated chemotherapy for cancer).

Here, we demonstrate how to integrate our code base for displaying volumetric data with OpenGL rendering functions. We will take advantage of techniques such as alpha blending, perspective rendering, and heat map rendering. Together with the GLFW interface for capturing user inputs, we can create an interactive visualizer that can display a large volumetric dataset in real-time and control a slicer that magnifies a plane of data points within the volumetric dataset using a few simple key inputs.

Getting ready

The simulation result is stored in an ASCII text file that contains a 3D matrix. Each value in the matrix represents the absorbed photon energy density at some fixed position within the voxelized geometry. Here, we will provide a simple parser that extracts the simulation output matrix from the file and stores it in the local memory.

How to do it...

Let's get started by implementing the MCML data parser, the jet color scheme heat map generator, as well as the slicer in OpenGL:

1. Take the data from the simulation output text file and store it in floating-point arrays:

```
#define MCML_SIZE_X 50
#define MCML_SIZE_Y 50
#define MCML_SIZE_Z 200
float mcml_data[MCML_SIZE_X][MCML_SIZE_Y][MCML_SIZE_Z];
Vertex mcml_vertices[MCML_SIZE_X][MCML_SIZE_Y][MCML_SIZE_Z];
float max_data, min_data;
int slice_x = 0, slice_z = 0, slice_y = 0;
float point_size=5.0f;

//load the data from a text file
void loadMCML(){
    FILE *ifp;
    //open the file for reading
    ifp = fopen("MCML_output.txt", "r");
    if (ifp == NULL) {
```

```
    fprintf(stderr, "ERROR: Can't open MCML Data file!\n");
    exit(1);
}
float data;
float max=0, min=9999999;
for(int x=0; x<MCML_SIZE_X; x++) {
    for(int z=0; z<MCML_SIZE_Z; z++) {
        for(int y=0; y<MCML_SIZE_Y; y++) {
            if (fscanf(ifp, "%f\n", &data) == EOF) {
                fprintf(stderr, "ERROR: Missing MCML Data file!\n");
                exit(1);
            }
            //store the log compressed data point
            data = log(data+1);
            mcml_data[x][y][z]=data;
            //find the max and min from the data set for heatmap
            if (data>max) {
                max=data;
            }
            if (data<min) {
                min=data;
            }
            //normalize the coordinates
            mcml_vertices[x][y][z].x=(float)(x-MCML_SIZE_X/2.0f)/
                MCML_SIZE_X;
            mcml_vertices[x][y][z].y=(float)(y-MCML_SIZE_Y/2.0f)/
                MCML_SIZE_Y;
            mcml_vertices[x][y][z].z=(float)(z-MCML_SIZE_Z/2.0f)/
                MCML_SIZE_Z*2.0f;
        }
    }
}
fclose(ifp);
max_data = max;
min_data = min;
halfmax= (max+min)/2.0f;
```

2. Encode the simulation output values using a custom color map for display:

```
//store the heat map representation of the data
for(int z=0; z<MCML_SIZE_Z; z++) {
    for(int x=0; x<MCML_SIZE_X; x++) {
        for(int y=0; y<MCML_SIZE_Y; y++) {
            float value = mcml_data[x][y][z];
            COLOUR c = GetColour(value, min_data,max_data);
            mcml_vertices[x][y][z].r=c.r;
```

```
    mcml_vertices[x][y][z].g=c.g;
    mcml_vertices[x][y][z].b=c.b;
}
}
}
}
```

3. Implement the heat map generator with the jet color scheme:

```
Color getHeatMapColor(float value, float min, float max)
{
    //remapping the value to the JET color scheme
    Color c = {1.0f, 1.0f, 1.0f}; // default value
    float dv;
    //clamp the data
    if (value < min)
        value = min;
    if (value > max)
        value = max;
    range = max - min;
    //the first region (0%-25%)
    if (value < (min + 0.25f * range)) {
        c.r = 0.0f;
        c.g = 4.0f * (value - min) / range;
    }
    //the second region of value (25%-50%)
    else if (value < (min + 0.5f * range)) {
        c.r = 0.0f;
        c.b = 1.0f + 4.0f * (min + 0.25f * range - value) / range;
    }
    //the third region of value (50%-75%)
    else if (value < (min + 0.75f * range)) {
        c.r = 4.0f * (value - min - 0.5f * range) / range;
        c.b = 0.0f;
    }
    //the fourth region (75%-100%)
    else {
        c.g = 1.0f + 4.0f * (min + 0.75f * range - value) / range;
        c.b = 0.0f;
    }
    return(c);
}
```

-
4. Draw all data points on screen with transparency enabled:

```
void drawMCMLPoints(){
    glPointSize(point_size);
    glBegin(GL_POINTS);
    for(int z=0; z<MCML_SIZE_Z; z++) {
        for(int x=0; x<MCML_SIZE_X; x++) {
            for(int y=0; y<MCML_SIZE_Y; y++) {
                glColor4f(mcml_vertices[x][y][z].r,mcml_vertices[x][y]
                          [z].g,mcml_vertices[x][y][z].b, 0.15f);
                glVertex3f(mcml_vertices[x][y][z].x,mcml_vertices[x][y]
                           [z].y,mcml_vertices[x][y][z].z);
            }
        }
    glEnd();
}
```

5. Draw three slices of data points for cross-sectional visualization:

```
void drawMCMLSlices(){
    glPointSize(10.0f);
    glBegin(GL_POINTS);

    //display data on xy plane
    for(int x=0; x<MCML_SIZE_X; x++) {
        for(int y=0; y<MCML_SIZE_Y; y++) {
            int z = slice_z;
            glColor4f(mcml_vertices[x][y][z].r,mcml_vertices[x][y]
                      [z].g,mcml_vertices[x][y][z].b, 0.9f);
            glVertex3f(mcml_vertices[x][y][z].x,mcml_vertices[x][y]
                       [z].y,mcml_vertices[x][y][z].z);
        }
    }

    //display data on yz plane
    for(int z=0; z<MCML_SIZE_Z; z++) {
        for(int y=0; y<MCML_SIZE_Y; y++) {
            int x = slice_x;
            glColor4f(mcml_vertices[x][y][z].r,mcml_vertices[x][y]
                      [z].g,mcml_vertices[x][y][z].b, 0.9f);
            glVertex3f(mcml_vertices[x][y][z].x,mcml_vertices[x][y]
                       [z].y,mcml_vertices[x][y][z].z);
        }
    }

    //display data on xz plane
    for(int z=0; z<MCML_SIZE_Z; z++) {
        for(int x=0; x<MCML_SIZE_X; x++) {
            int y = slice_y;
```

```
    glColor4f(mcml_vertices[x][y][z].r,mcml_vertices[x][y]
              [z].g,mcml_vertices[x][y][z].b, 0.9f);
    glVertex3f(mcml_vertices[x][y][z].x,mcml_vertices[x][y]
              [z].y,mcml_vertices[x][y][z].z);
}
}
glEnd();
}
```

6. In addition, we need to update the `key_callback` function for moving the slices:

```
void key_callback(GLFWwindow* window, int key, int scancode,
                  int action, int mods)
{
    if (action != GLFW_PRESS)
        return;
    switch (key)
    {
        case GLFW_KEY_ESCAPE:
            glfwSetWindowShouldClose(window, GL_TRUE);
            break;
        case GLFW_KEY_P:
            point_size+=0.5;
            break;
        case GLFW_KEY_O:
            point_size-=0.5;
            break;
        case GLFW_KEY_A:
            slice_y -=1;
            if(slice_y < 0)
                slice_y = 0;
            break;
        case GLFW_KEY_D:
            slice_y +=1;
            if(slice_y >= MCML_SIZE_Y-1)
                slice_y = MCML_SIZE_Y-1;
            break;
        case GLFW_KEY_W:
            slice_z +=1;
            if(slice_z >= MCML_SIZE_Z-1)
                slice_z = MCML_SIZE_Z-1;
            break;
        case GLFW_KEY_S:
            slice_z -= 1;
            if (slice_z < 0)
                slice_z = 0;
```

```
        break;
    case GLFW_KEY_E:
        slice_x -=1;
        if(slice_x < 0)
            slice_x = 0;
        break;
    case GLFW_KEY_Q:
        slice_x +=1;
        if(slice_x >= MCML_SIZE_X-1)
            slice_x = MCML_SIZE_X-1;
        break;
    case GLFW_KEY_PAGE_UP:
        zoom -= 0.25f;
        if (zoom < 0.f)
            zoom = 0.f;
        break;
    case GLFW_KEY_PAGE_DOWN:
        zoom += 0.25f;
        break;
    default:
        break;
    }
}
```

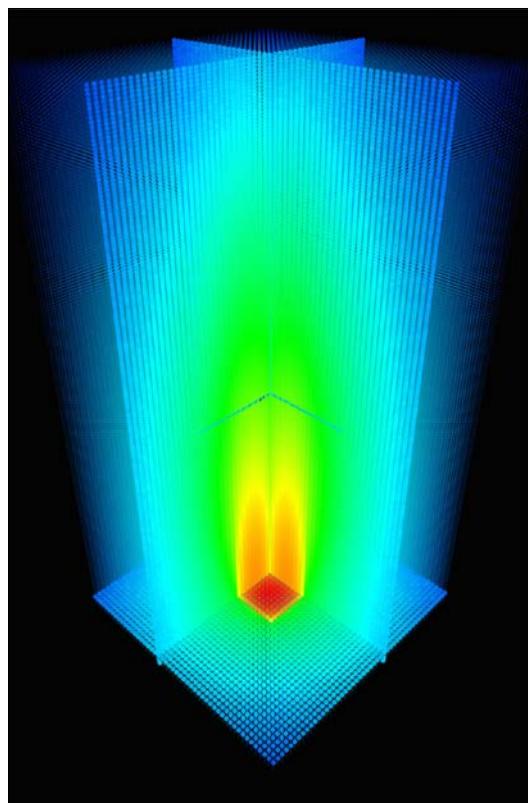
7. Finally, to complete the demo, simply call the `drawMCMLPoints` and `drawMCMLSlices` functions inside the `main` loop using the same code structure for perspective rendering introduced in the previous demo for plotting a Gaussian function:

```
while (!glfwWindowShouldClose(window))
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -zoom);
    glRotatef(beta, 1.0, 0.0, 0.0);
    glRotatef(alpha, 0.0, 0.0, 1.0);
    //disable depth test so we can render the points with blending
    glDisable(GL_DEPTH_TEST);
    drawMCMLPoints();
    //must enable this to ensure the slides are rendered in the
    right order
```

```
glEnable(GL_DEPTH_TEST);  
drawMCMLSlices();  
  
//draw the origin with the x,y,z axes for visualization  
drawOrigin();  
glfwSwapBuffers(window);  
glfwPollEvents();  
}
```

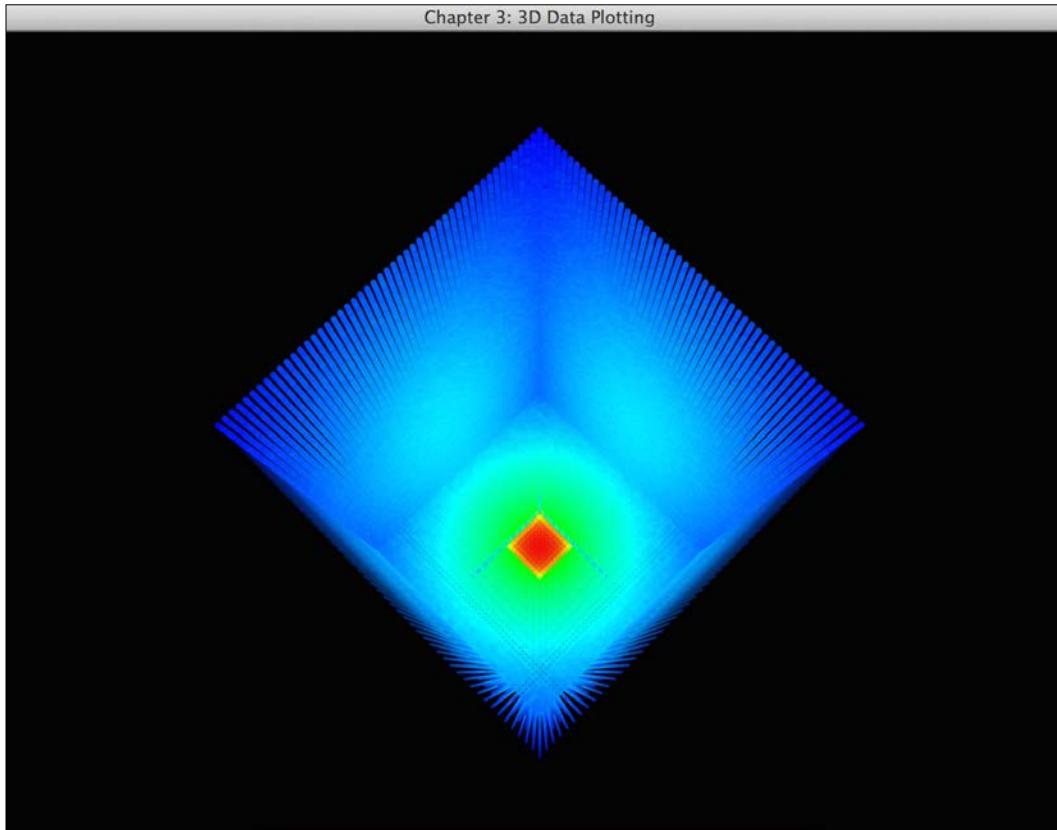
The simulation results, representing the photon absorption distribution in a voxelized geometry, are displayed in 3D in the following screenshot. The light source illuminates the tissue surface ($z=0$ at the bottom) and propagates through the tissue (positive z direction) that is modeled as an infinitely wide homogeneous medium. The photon absorption distribution follows the expected shape for a finite-sized, flat, and circular beam:



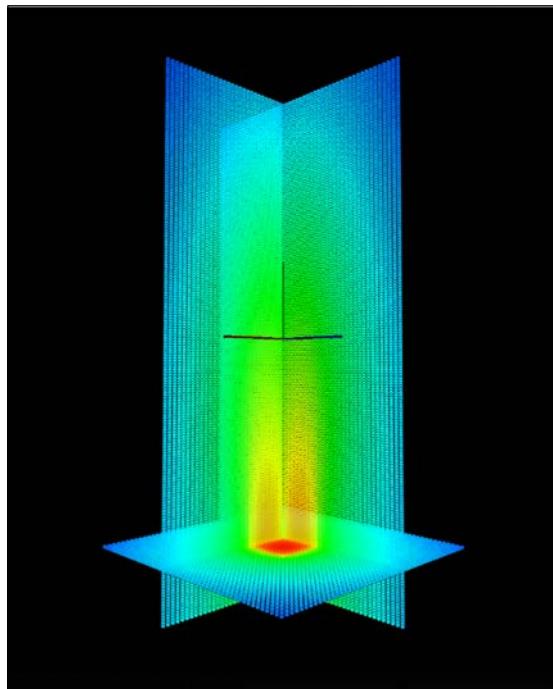
How it works...

This demo illustrates how we can take a volumetric dataset generated from a Monte Carlo simulation (and, more generally, a volumetric dataset from any application) and render it with a highly interactive interface using OpenGL. The data parser takes an ASCII text file as input. Then, we turn the floating-point data into individual vertices that can fit into our rendering pipeline. Upon initialization, the variables `mcmc_vertices` and `mcmc_data` store the pre-computed heat map data as well as the position of each data point. The `parser` function also computes the maximum and minimum value in the dataset for heat map visualization. The `getHeatMapColor` function takes the simulation output value and maps it to a color in the jet color scheme. The algorithm basically defines a color spectrum and we remap the value based on its range.

In the following screenshot, we show a top view of the simulation result, which allows us to visualize the symmetry of the light distribution:



The `drawMCMLSlices` function takes a slice (that is, a plane) of data and renders the data points at the full opacity and a larger point size. This provides a useful and very common visualization method (especially in medical imaging) that allows users to examine the volumetric data in detail by moving the cross-sectional slices. As we can see in the following screenshot, we can shift the slicer in the x , y , and z directions to visualize the desired regions of interest:



There's more...

This demo provides an example of real-time volumetric data visualization for rendering simulation data in an interactive 3D environment. The current implementation can be easily modified for a wide range of applications that require volumetric dataset visualization. Our approach provides an intuitive way to render complex 3D datasets with a heat map generator and a slicer as well as 3D perspective rendering techniques using OpenGL.

One important observation is that this demo required a significant number of `glVertex3f` calls, which can become a major performance bottleneck. To address this, in the upcoming chapters, we will explore more sophisticated ways to handle memory transfer and draw even more complex models with **Vertex Buffer Objects (VBOs)**, a memory buffer in your graphics card designed to store information about vertices. This will lead us towards fragment programs and custom vertex shader programs (that is, moving from OpenGL 2.0 to OpenGL 3.2 or higher). However, the simplicity of using classical OpenGL 2.0 calls is an important consideration if we are aiming for a short development cycle, minimal overhead, and backward compatibility with older hardware.

See also

For further information, please consult the following references:

- ▶ E. Alerstam & W. C. Y. Lo, T. Han, J. Rose, S. Andersson-Engels, and L. Lilge, "Next-generation acceleration and code optimization for light transport in turbid media using GPUs," *Biomed. Opt. Express* 1, 658-675 (2010).
- ▶ W. C. Y. Lo, K. Redmond, J. Luu, P. Chow, J. Rose, and L. Lilge, "Hardware acceleration of a Monte Carlo simulation for photodynamic therapy treatment planning," *J. Biomed. Opt.* 14, 014019 (2009).
- ▶ L. Wang, S. Jacques, and L. Zheng, "MCML - Monte Carlo modeling of light transport in multi-layered tissues," *Comput. Meth. Prog. Biol.* 47, 131–146 (1995).
- ▶ B. Wilson and G. Adam, "A Monte Carlo model for the absorption and flux distributions of light in tissue," *Med. Phys.* 10, 824 (1983).

4

Rendering 2D Images and Videos with Texture Mapping

In this chapter, we will cover the following topics:

- ▶ Getting started with modern OpenGL (3.2 or higher)
- ▶ Setting up the GLEW, GLM, SOIL, and OpenCV libraries in Windows
- ▶ Setting up the GLEW, GLM, SOIL, and OpenCV libraries in Mac OS X/Linux
- ▶ Creating your first vertex and fragment shader using GLSL
- ▶ Rendering 2D images with texture mapping
- ▶ Real-time video rendering with filters

Introduction

In this chapter, we will introduce OpenGL techniques to visualize another important class of datasets: those involving images or videos. Such datasets are commonly encountered in many fields, including medical imaging applications. To enable the rendering of images, we will discuss fundamental OpenGL concepts for texture mapping and transition to more advanced techniques that require newer versions of OpenGL (OpenGL 3.2 or higher). To simplify our tasks, we will also employ several additional libraries, including **OpenGL Extension Wrangler Library (GLEW)** for runtime OpenGL extension support, **Simple OpenGL Image Loader (SOIL)** to load different image formats, **OpenGL Mathematics (GLM)** for vector and matrix manipulation, as well as **OpenCV** for image/video processing. To get started, we will first introduce the features of modern OpenGL 3.2 and higher.

Getting started with modern OpenGL (3.2 or higher)

Continuous evolution of OpenGL APIs has led to the emergence of a modern standard. One of the biggest changes happened in 2008 with OpenGL version 3.0, in which a new context creation mechanism was introduced and most of the older functions, such as Begin/End primitive specifications, were marked as deprecated. The removal of these older standard features also implies a more flexible yet more powerful way of handling the graphics pipeline. In OpenGL 3.2 or higher, a core and a compatible profile were defined to differentiate the deprecated APIs from the current features. These profiles provide clear definitions for various features (core profile) while enabling backward compatibility (compatibility profile). In OpenGL 4.x, support for the latest graphics hardware that runs Direct3D 11 is provided, and a detailed comparison between OpenGL 3.x and OpenGL 4.x is available at <http://www.g-truc.net/post-0269.html>.

Getting ready

Starting from this chapter, we need compatible graphics cards with OpenGL 3.2 (or higher) support. Most graphics cards released before 2008 will most likely not be supported. For example, NVIDIA GeForce 100, 200, 300 series and higher support the OpenGL 3 standard. You are encouraged to consult the technical specifications of your graphics cards to confirm the compatibility (refer to <https://developer.nvidia.com/opengl-driver>).

How to do it...

To enable OpenGL 3.2 support, we need to incorporate the following lines of code at the beginning of every program for initialization:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

How it works...

The `glfwWindowHint` function defines a set of constraints for the creation of the GLFW windows context (refer to *Chapter 1, Getting Started with OpenGL*). The first two lines of code here define the current version of OpenGL that will be used (3.2 in this case). The third line enables forward compatibility, while the last line specifies that the core profile will be used.

See also

Detailed explanation of the differences between various OpenGL versions can be found at http://www.opengl.org/wiki/History_of_OpenGL.

Setting up the GLEW, GLM, SOIL, and OpenCV libraries in Windows

In this section, we will provide step-by-step instructions to set up several popular libraries that will be used extensively in this chapter (and in subsequent chapters), including the GLEW, GLM, SOIL, and OpenCV libraries:

- ▶ The GLEW library is an open-source OpenGL extension library.
- ▶ The GLM library is a header-only C++ library that provides an easy-to-use set of common mathematical operations. It is built on the GLSL specifications and as it is a header-only library, it does not require tedious compilation steps.
- ▶ The SOIL library is a simple C library that is used to load images in a variety of common formats (such as BMP, PNG, JPG, TGA, TIFF, and HDR) in OpenGL textures.
- ▶ The OpenCV library is a very powerful open source computer vision library that we will use to simplify image and video processing in this chapter.

Getting ready

We will first need to download the prerequisite libraries from the following websites:

- ▶ **GLEW** (glew-1.10.0): <http://sourceforge.net/projects/glew/files/glew/1.10.0/glew-1.10.0-win32.zip>
- ▶ **GLM** (glm-0.9.5.4): <http://sourceforge.net/projects/ogl-math/files/glm-0.9.5.4/glm-0.9.5.4.zip>
- ▶ **SOIL**: <http://www.lonesock.net/files/soil.zip>
- ▶ **OpenCV** (opencv-2.4.9): <http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.9/opencv-2.4.9.exe>

How to do it...

To use the precompiled package from GLEW, follow these steps:

1. Unzip the package.
2. Copy the directory to C:/Program Files (x86).

3. Ensure that the `glew32.dll` file (`C:\Program Files (x86)\glew-1.10.0\bin\Release\Win32`) can be found at runtime by placing it either in the same folder as the executable or including the directory in the Windows system PATH environment variable (Navigate to **Control Panel | System and Security | System | Advanced Systems Settings | Environment Variables**).

To use the header-only GLM library, follow these steps:

1. Unzip the package.
2. Copy the directory to `C:/Program Files (x86)`.
3. Include the desired header files in your source code. Here is an example:

```
#include <glm/glm.hpp>
```

To use the SOIL library, follow these steps:

1. Unzip the package.
2. Copy the directory to `C:/Program Files (x86)`.
3. Generate the `SOIL.lib` file by opening the Visual Studio solution file (`C:\Program Files (x86)\Simple OpenGL Image Library\projects\VC9\SOIL.sln`) and compiling the project files. Copy this file from `C:\Program Files (x86)\Simple OpenGL Image Library\projects\VC9\Debug` to `C:\Program Files (x86)\Simple OpenGL Image Library\lib`.
4. Include the header file in your source code:

```
#include <SOIL.h>
```

Finally, to install OpenCV, we recommend that you use prebuilt binaries to simplify the process:

1. Download the prebuilt binaries from <http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.9/opencv-2.4.9.exe> and extract the package.
2. Copy the directory (the `opencv` folder) to `C:\Program Files (x86)`.
3. Add this to the system PATH environment variable (Navigate to **Control Panel | System and Security | System | Advanced Systems Settings | Environment Variables**) - `C:\Program Files (x86)\opencv\build\x86\vc12\bin`.

4. Include the desired header files in your source code:

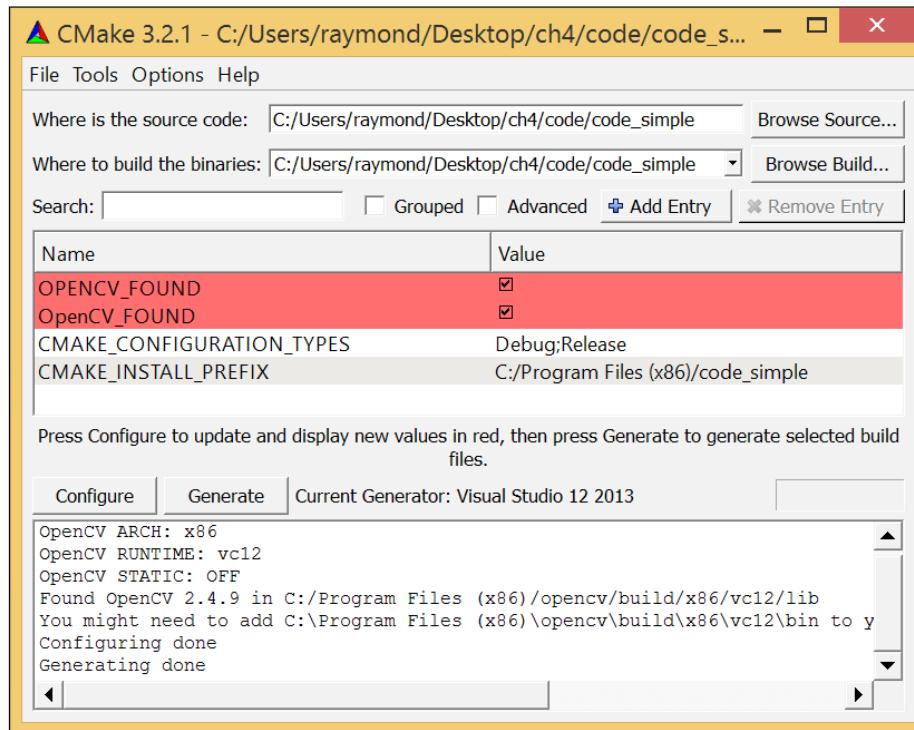
```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

Now, we generate our Microsoft Visual Studio Solution files (the build environment) using CMake. We create the `CMakeList.txt` file within each project directory, which lists all the libraries and dependencies for the project. The following is a sample `CMakeList.txt` file for our first demo application:

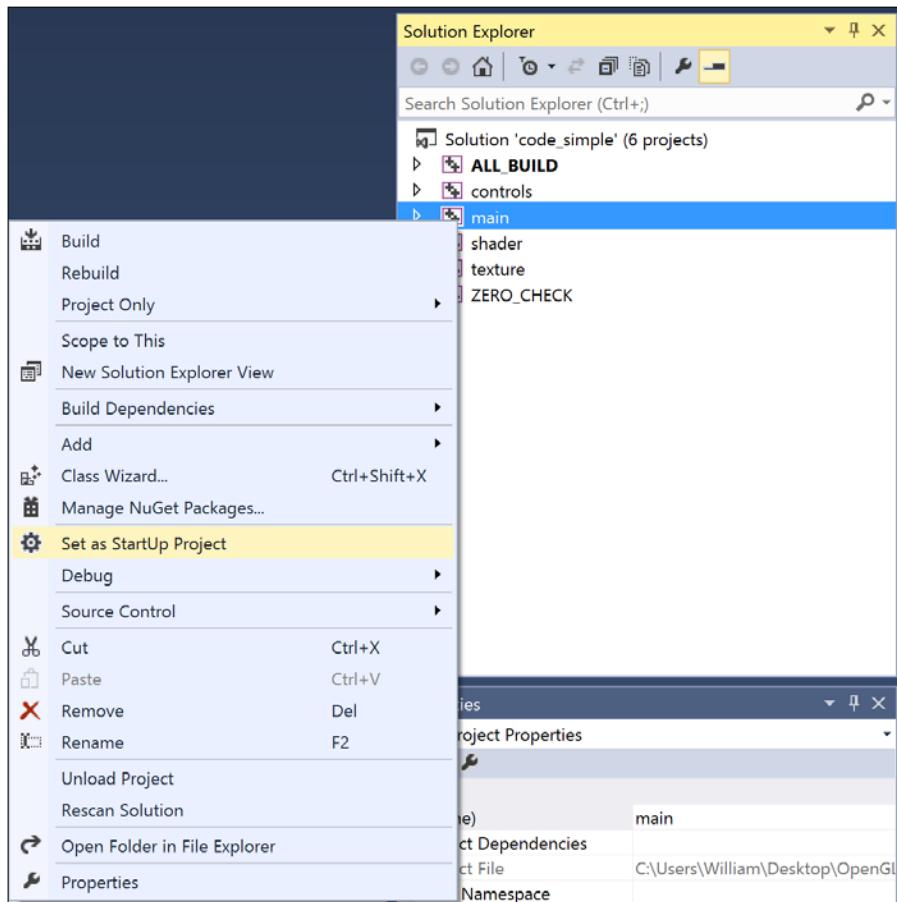
```
cmake_minimum_required (VERSION 2.8)
set(CMAKE_CONFIGURATION_TYPES Debug Release)
set(PROGRAM_PATH "C:/Program Files \x86")
set(OpenCV_DIR ${PROGRAM_PATH}/opencv/build)
project (code_simple)
#modify these path based on your configuration
#OpenCV
find_package(OpenCV REQUIRED )
INCLUDE_DIRECTORIES(${OpenCV_INCLUDE_DIRS})
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glm)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glew-1.10.0/include)
LINK_DIRECTORIES(${PROGRAM_PATH}/glew-1.10.0/lib/Release)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glfw-3.0.4/include)
LINK_DIRECTORIES(${PROGRAM_PATH}/glfw-3.0.4/lib)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/Simple\ OpenGL\ Image\ Library/
src)
LINK_DIRECTORIES(${PROGRAM_PATH}/Simple\ OpenGL\ Image\ Library/lib)
add_subdirectory (../common common)
add_executable (main main.cpp)
target_link_libraries (main LINK_PUBLIC shader controls texture
glew32s glfw3 opengl32 ${OpenCV_LIBS} SOIL)
```

As you can see in the `CMakeList.txt` file, the various dependencies, including the OpenCV, SOIL, GLFW, and GLEW libraries, are all included.

Finally, we run the CMake program to generate the Microsoft Visual Studio Solution for the project (refer to *Chapter 1, Getting Started with OpenGL* for details). Note that the output path for the binary must match the project folder due to dependencies of the shader programs. The following is a screenshot of the CMake window after generating the first sample project called `code_simple`:



We will repeat this step for each project we create, and the corresponding Microsoft Visual Studio Solution file will be generated accordingly (for example, `code_simple.sln` in this case). To compile the code, open `code_simple.sln` with Microsoft Visual Studio 2013 and build the project using the Build (press *F7*) function as usual. Make sure that you set main as the start up project (by right-clicking on the *main* project in the **Solution Explorer** and left-clicking on the **Set as StartUp Project** option) before running the program, as shown follows:



See also

Further documentation on each of the libraries installed can be found here:

- ▶ **GLEW:** <http://glew.sourceforge.net/>
- ▶ **GLM:** <http://glm.g-truc.net/0.9.5/index.html>
- ▶ **SOIL:** <http://www.lonesock.net/soil.html>
- ▶ **OpenCV:** <http://opencv.org/>

Setting up the **GLEW**, **GLM**, **SOIL**, and **OpenCV** libraries in Mac OS X/Linux

In this section, we will outline the steps required to set up the same libraries in Mac OS X and Linux.

Getting ready

We will first need to download the prerequisite libraries from the following websites:

1. **GLEW** (`glew-1.10.0`): <https://sourceforge.net/projects/glew/files/glew/1.10.0/glew-1.10.0.tgz>
2. **GLM** (`glm-0.9.5.4`): <http://sourceforge.net/projects/ogl-math/files/glm-0.9.5.4/glm-0.9.5.4.zip>
3. **SOIL**: <http://www.lonesock.net/files/soil.zip>
4. **OpenCV** (`opencv-2.4.9`): <http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.9/opencv-2.4.9.zip>

To simplify the installation process for Mac OS X or Ubuntu users, the use of MacPorts in Mac OS X or the `apt-get` command in Linux (as described in *Chapter 1, Getting Started with OpenGL*) is highly recommended.

The following section assumes that the download directory is `~/opengl_dev` (refer to *Chapter 1, Getting Started with OpenGL*).

How to do it...

There are two methods to install the prerequisite libraries. The first method uses precompiled binaries. These binary files are fetched from remote repository servers and the version updates of the library are controlled externally. An important advantage of this method is that it simplifies the installation, especially in terms of resolving dependencies. However, in a release environment, it is recommended that you disable the automatic updates and thus protect the binary from version skewing. The second method requires users to download and compile the source code directly with various customizations. This method is recommended for users who would like to control the installation process (such as the paths), and it also provides more flexibility in terms of tracking and fixing bugs.

For beginners or developers who are looking for rapid prototyping, we recommend that you use the first method as it will simplify the workflow and have short-term maintenance. On an Ubuntu or Debian system, we can install the various libraries using the `apt-get` command. To install all the prerequisite libraries and dependencies on Ubuntu, simply run the following commands in the terminal:

```
sudo apt-get install libglm-dev libglew1.6-dev libsoil-dev libopencv
```

Similarly, on Mac OS X, we can install GLEW, OpenCV, and GLM with MacPorts through command lines in the terminal:

```
sudo port install opencv glm glew
```

However, the SOIL library is not currently supported by MacPorts, and thus, the installation has to be completed manually, as described in the following section.

For advanced users, we can install the latest packages by directly compiling from the source, and the upcoming steps are common among Mac OS as well as other Linux OS.

To compile the GLEW package, follow these steps:

1. Extract the `glew-1.10.0.tgz` package:

```
tar xzvf glew-1.10.0.tgz
```

2. Install GLEW in `/usr/include/GL` and `/usr/lib`:

```
cd glew-1.10.0  
make && sudo make install
```

To set up the header-only GLM library, follow these steps:

1. Extract the `unzip glm-0.9.5.4.zip` package:

```
unzip glm-0.9.5.4.zip
```

2. Copy the header-only GLM library directory (`~/opengl_dev/glm/glm`) to `/usr/include/glm`:

```
sudo cp -r glm/glm/ /usr/include/glm
```

To set up the SOIL library, follow these steps:

1. Extract the `unzip soil.zip` package:

```
unzip soil.zip
```

2. Edit `makefile` (inside the `projects/makefile` directory) and add `-arch x86_64` and `-arch i386` to `CXXFLAGS` to ensure proper support:

```
CXXFLAGS =-arch x86_64 -arch i386 -O2 -s -Wall
```

3. Compile the source code library:

```
cd Simple\ OpenGL\ Image\ Library/projects/makefile
mkdir obj
make && sudo make install
```

To set up the OpenCV library, follow these steps:

1. Extract the opencv-2.4.9.zip package:

```
unzip opencv-2.4.9.zip
```

2. Build the OpenCV library using CMake:

```
cd opencv-2.4.9/
mkdir build
cd build
cmake ../
make && sudo make install
```

3. Configure the library path:

```
sudo sh -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
sudo ldconfig -v
```

4. With the development environment fully configured, we can now create the compilation script (Makefile) within each project folder:

```
CFILES = ../common/shader.cpp ../common/texture.cpp ../common/
controls.cpp main.cpp
CFLAGS = -O3 -c -Wall
INCLUDES = -I/usr/include -I/usr/include/SOIL -I../common `pkg-
config --cflags glfw3` `pkg-config --cflags opencv`
LIBS = -lm -L/usr/local/lib -lGLEW -lSOIL `pkg-config --static
--libs glfw3` `pkg-config --libs opencv`
CC = g++
OBJECTS=$(CFILES:.cpp=.o)
EXECUTABLE=main
all: $(CFILES) $(EXECUTABLE)
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(INCLUDES) $(OBJECTS) -o $@ $(LIBS)
.cpp.o:
    $(CC) $(CFLAGS) $(INCLUDES) $< -o $@

clean:
    rm -v -f *~ ../common/*.o *.o *.obj $(EXECUTABLE)
```

To compile the code, we simply run the `make` command in the project directory and it generates the executable (`main`) automatically.

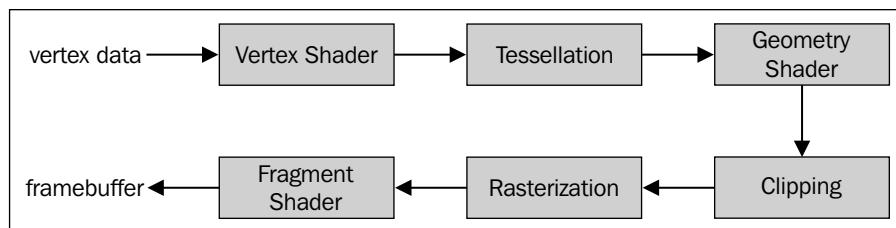
See also

Further documentation on each of the libraries installed can be found here:

- ▶ **GLEW:** <http://glew.sourceforge.net/>
- ▶ **GLM:** <http://glm.g-truc.net/0.9.5/index.html>
- ▶ **SOIL:** <http://www.lonesock.net/soil.html>
- ▶ **OpenCV:** <http://opencv.org/>
- ▶ **MacPorts:** <http://www.macports.org/>

Creating your first vertex and fragment shader using GLSL

Before we can render images using OpenGL, we need to first understand the basics of the GLSL. In particular, the concept of shader programs is essential in GLSL. Shaders are simply programs that run on graphics processors (GPUs), and a set of shaders is compiled and linked to form a program. This concept emerges as a result of the increasing complexity of various common processing tasks in modern graphics hardware, such as vertex and fragment processing, which necessitates greater programmability of specialized processors. Accordingly, the vertex and fragment shader are two important types of shaders that we will cover here, and they run on the vertex processor and fragment processor, respectively. A simplified diagram illustrating the overall processing pipeline is shown as follows:



The main purpose of the vertex shader is to perform the processing of a stream of vertex data. An important processing task involves the transformation of the position of each vertex from the 3D virtual space to a 2D coordinate for display on the screen. Vertex shaders can also manipulate the color and texture coordinates. Therefore, vertex shaders serve as an important component of the OpenGL pipeline to control movement, lighting, and color.

A fragment shader is primarily designed to compute the final color of an individual pixel (fragment). Oftentimes, we implement various image post-processing techniques, such as blurring or sharpening, at this stage; the end results are stored in the framebuffer, which will be displayed on screen.

For readers interested in understanding the rest of the pipeline, a detailed summary of these stages, such as the clipping, rasterization, and tessellation, can be found at https://www.opengl.org/wiki/Rendering_Pipeline_Overview. Additionally, a detailed documentation of GLSL can be found at <https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf>.

Getting ready

At this point, we should have all the prerequisite libraries, such as GLEW, GLM, and SOIL. With GLFW configured for the OpenGL core profile, we are now ready to implement the first simple example code, which takes advantage of the modern OpenGL pipeline.

How to do it...

To keep the code simple, we will divide the program into two components: the main program (`main.cpp`) and shader programs (`shader.cpp`, `shader.hpp`, `simple.vert`, and `simple.frag`). The main program performs the essential tasks to set up the simple demo, while the shader programs perform the specialized processing in the modern OpenGL pipeline. The complete sample code can be found in the `code_simple` folder.

First, let's take a look at the shader programs. We will create two extremely simple vertex and fragment shader programs (specified inside the `simple.vert` and `simple.frag` files) that are compiled and loaded by the program at runtime.

For the `simple.vert` file, enter the following lines of code:

```
#version 150
in vec3 position;
in vec3 color_in;
out vec3 color;
void main() {
    color = color_in;
    gl_Position = vec4(position, 1.0);
}
```

For the `simple.frag` file, enter the following lines of code:

```
#version 150
in vec3 color;
out vec4 color_out;
void main() {
    color_out = vec4(Color, 1.0);
}
```

First, let's define a function to compile and load the shader programs (`simple.frag` and `simple.vert`) called `LoadShaders` inside `shader.hpp`:

```
#ifndef SHADER_HPP
#define SHADER_HPP
GLuint LoadShaders(const char * vertex_file_path,const char *
                    fragment_file_path);
#endif
```

Next, we will create the `shader.cpp` file to implement the `LoadShaders` function and two helper functions to handle file I/O (`readSourceFile`) and the compilation of the shaders (`CompileShader`):

1. Include prerequisite libraries and the `shader.hpp` header file:

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include "shader.hpp"
```

2. Implement the `readSourceFile` function as follows:

```
std::string readSourceFile(const char *path) {
    std::string code;
    std::ifstream file_stream(path, std::ios::in);
    if(file_stream.is_open()){
        std::string line = "";
        while(getline(file_stream, line))
            code += "\n" + line;
        file_stream.close();
        return code;
    }else{
        printf("Failed to open \"%s\".\n", path);
        return "";
    }
}
```

3. Implement the `CompileShader` function as follows:

```
void CompileShader(std::string program_code, GLuint
    shader_id) {
    GLint result = GL_FALSE;
    int infolog_length;
    char const * program_code_pointer = program_code.c_str();
    glShaderSource(shader_id, 1, &program_code_pointer ,
        NULL);
    glCompileShader(shader_id);
    //check the shader for successful compile
    glGetShaderiv(shader_id, GL_COMPILE_STATUS, &result);
    glGetShaderiv(shader_id, GL_INFO_LOG_LENGTH,
        &infolog_length);
    if ( infolog_length > 0 ){
        std::vector<char> error_msg(infolog_length+1);
        glGetShaderInfoLog(shader_id, infolog_length, NULL,
            &error_msg[0]);
        printf("%s\n", &error_msg[0]);
    }
}
```

4. Now, let's implement the `LoadShaders` function. First, create the shader ID and read the shader code from two files specified by `vertex_file_path` and `fragment_file_path`:

```
GLuint LoadShaders(const char * vertex_file_path,const char
    * fragment_file_path){
    GLuint vertex_shader_id =
        glCreateShader(GL_VERTEX_SHADER);
    GLuint fragment_shader_id =
        glCreateShader(GL_FRAGMENT_SHADER);
    std::string vertex_shader_code =
        readSourceFile(vertex_file_path);
    if(vertex_shader_code == ""){
        return 0;
    }
    std::string fragment_shader_code =
        readSourceFile(fragment_file_path);
    if(fragment_shader_code == ""){
        return 0;
    }
```

5. Compile the vertex shader and fragment shader programs:

```
printf("Compiling Vertex shader : %s\n",
       vertex_file_path);
CompileShader(vertex_shader_code, vertex_shader_id);
printf("Compiling Fragment shader :
       %s\n",fragment_file_path);
CompileShader(fragment_shader_code, fragment_shader_id);
```

6. Link the programs together, check for errors, and clean up:

```
GLint result = GL_FALSE;
int infolog_length;
printf("Linking program\n");
GLuint program_id = glCreateProgram();
glAttachShader(program_id, vertex_shader_id);
glAttachShader(program_id, fragment_shader_id);
glLinkProgram(program_id);
//check the program and ensure that the program is linked properly
glGetProgramiv(program_id, GL_LINK_STATUS, &result);
glGetProgramiv(program_id, GL_INFO_LOG_LENGTH,
               &infolog_length);
if ( infolog_length > 0 ){
    std::vector<char> program_error_msg(infolog_length+1);
    glGetProgramInfoLog(program_id, infolog_length, NULL,
                         &program_error_msg[0]);
    printf("%s\n", &program_error_msg[0]);
} else{
    printf("Linked Successfully\n");
}

//flag for delete, and will free all memories
//when the attached program is deleted
glDeleteShader(vertex_shader_id);
glDeleteShader(fragment_shader_id);
return program_id;
}
```

Finally, let's put everything together with the `main.cpp` file:

1. Include prerequisite libraries and the shader program header file inside the common folder:

```
#include <stdio.h>
#include <stdlib.h>
//GLFW and GLEW libraries
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "common/shader.hpp"
```

2. Create a global variable for the GLFW window:

```
//Global variables
GLFWwindow* window;
```

3. Start the main program with the initialization of the GLFW library:

```
int main(int argc, char **argv)
{
    //Initialize GLFW
    if(!glfwInit()){
        fprintf( stderr, "Failed to initialize GLFW\n" );
        exit(EXIT_FAILURE);
    }
}
```

4. Set up the GLFW window:

```
//enable anti-aliasing 4x with GLFW
glfwWindowHint(GLFW_SAMPLES, 4);
/* specify the client API version that the created context
   must be compatible with. */
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
//make the GLFW forward compatible
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
//use the OpenGL Core
glfwWindowHint(GLFW_OPENGL_PROFILE,
    GLFW_OPENGL_CORE_PROFILE);
```

5. Create the GLFW window object and make the context of the specified window current on the calling thread:

```
window = glfwCreateWindow(640, 480, "Chapter 4 - GLSL",
    NULL, NULL);
if(!window){
    fprintf( stderr, "Failed to open GLFW window. If you
        have an Intel GPU, they are not 3.3 compatible. Try
        the 2.1 version of the tutorials.\n" );
```

```
    glfwTerminate();
    exit(EXIT_FAILURE);
}
glfwMakeContextCurrent(window);
glfwSwapInterval(1);
```

6. Initialize the GLEW library and include support for experimental drivers:

```
glewExperimental = true;
if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Final to Initialize GLEW\n");
    glfwTerminate();
    exit(EXIT_FAILURE);
}
```

7. Set up the shader programs:

```
GLuint program = LoadShaders("simple.vert",
    "simple.frag");
glBindFragDataLocation(program, 0, "color_out");
glUseProgram(program);
```

8. Set up Vertex Buffer Object (and color buffer) and copy the vertex data to it:

```
GLuint vertex_buffer;
GLuint color_buffer;
 glGenBuffers(1, &vertex_buffer);
 glGenBuffers(1, &color_buffer);
const GLfloat vertices[] = {
    -1.0f, -1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    1.0f, 1.0f, 0.0f,
    -1.0f, -1.0f, 0.0f,
    1.0f, 1.0f, 0.0f,
    -1.0f, 1.0f, 0.0f
};
const GLfloat colors[] = {
    0.0f, 0.0f, 1.0f,
    0.0f, 1.0f, 0.0f,
    1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f
};

 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
 glBindBuffer(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
 GL_STATIC_DRAW);
```

```
glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors,
    GL_STATIC_DRAW);
```

9. Specify the layout of the vertex data:

```
GLint position_attrib = glGetAttribLocation(program,
    "position");
 glEnableVertexAttribArray(position_attrib);
 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
 glVertexAttribPointer(position_attrib, 3, GL_FLOAT,
    GL_FALSE, 0, (void*)0);

GLint color_attrib = glGetAttribLocation(program,
    "color_in");
 glEnableVertexAttribArray(color_attrib);
 glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
 glVertexAttribPointer(color_attrib, 3, GL_FLOAT,
    GL_FALSE, 0, (void*)0);
```

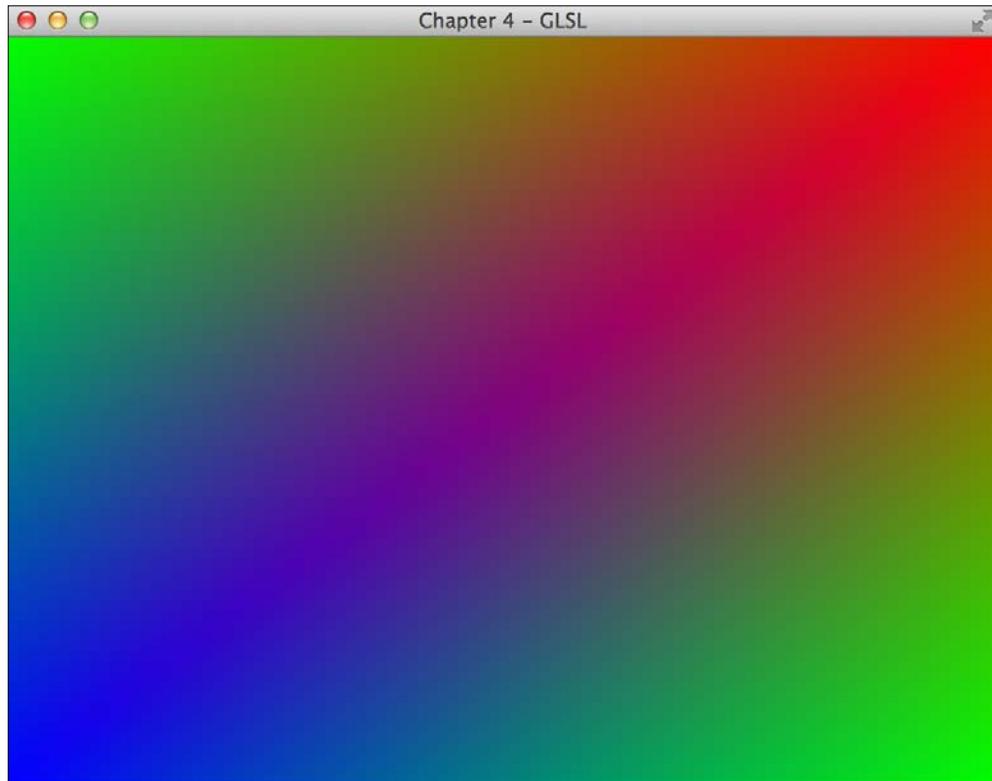
10. Run the drawing program:

```
while (!glfwWindowShouldClose(window)) {
    // Clear the screen to black
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);
    // Draw a rectangle from the 2 triangles using 6
    // vertices
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

11. Clean up and exit the program:

```
//clean up the memories
glDisableVertexAttribArray(position_attrib);
glDisableVertexAttribArray(color_attrib);
glDeleteBuffers(1, &vertex_buffer);
glDeleteBuffers(1, &color_buffer);
glDeleteVertexArrays(1, &vertex_array);
glDeleteProgram(program);
// Close OpenGL window and terminate GLFW
glfwDestroyWindow(window);
glfwTerminate();
exit(EXIT_SUCCESS);
}
```

Now we have created the first GLSL program by defining custom shaders:



How it works...

As there are multiple components in this implementation, we will highlight the key features inside each component separately, organized in the same order as the previous section using the same file name for simplicity.

Inside `simple.vert`, we defined a simple vertex shader. In the first simple implementation, the vertex shader simply passes information forward to the rest of the rendering pipeline. First, we need to define the GLSL version that corresponds to the OpenGL 3.2 support, which is 1.50 (`#version 150`). The vertex shader takes two parameters: the position of the vertex (`in vec3 position`) and the color (`in vec3 color_in`). Note that only the color is defined explicitly in an output variable (`out vec3 color`) as `gl_Position` is a built-in variable. In general, variable names with the prefix `gl` should not be used inside shader programs in OpenGL as these are reserved for built-in variables. Notice that the final position, `gl_Position`, is expressed in homogeneous coordinates.

Inside `simple.frag`, we defined the fragment shader, which again passes the color information forward to the output framebuffer. Notice that the final output (`color_out`) is expressed in the RGBA format, where A is the alpha value (transparency).

Next, in `shader.cpp`, we created a framework to compile and link shader programs. The workflow shares some similarity with conventional code compilation in C/C++. Briefly, there are six major steps:

1. Create a shader object (`glCreateShader`).
2. Read and set the shader source code (`glShaderSource`).
3. Compile (`glCompileShader`).
4. Create the final program ID (`glCreateProgram`).
5. Attach a shader to the program ID (`glAttachShader`).
6. Link everything together (`glLinkProgram`).

Finally, in `main.cpp`, we set up a demo to illustrate the use of the compiled shader program. As described in the *Getting Started with Modern OpenGL* section of this chapter, we need to use the `glfwWindowHint` function to properly create the GLFW window context in OpenGL 3.2. An interesting aspect to point out about this demo is that even though we defined only six vertices (three vertices for each of the two triangles drawn using the `glDrawArrays` function) and their corresponding colors, the final result is an interpolated color gradient.

Rendering 2D images with texture mapping

Now that we have introduced the basics of GLSL using a simple example, we will incorporate further complexity to provide a complete framework that enables users to modify any part of the rendering pipeline in the future.

The code in this framework is divided into smaller modules to handle the shader programs (`shader.cpp` and `shader.hpp`), texture mapping (`texture.cpp` and `texture.hpp`), and user inputs (`controls.hpp` and `controls.cpp`). First, we will reuse the mechanism to load shader programs in OpenGL introduced previously and incorporate new shader programs for our purpose. Next, we will introduce the steps required for texture mapping. Finally, we will describe the main program, which integrates all the logical pieces and prepares the final demo. In this section, we will show how we can load an image and convert it into a texture object to be rendered in OpenGL. With this framework in mind, we will further demonstrate how to render a video in the next section.

Getting ready

To avoid redundancy here, we will refer readers to the previous section for part of this demo (in particular, `shader.cpp` and `shader.hpp`).

How to do it...

First, we aggregate all the common libraries used in our program into the `common.h` header file. The `common.h` file is then included in `shader.hpp`, `controls.hpp`, `texture.hpp`, and `main.cpp`:

```
#ifndef _COMMON_H
#define _COMMON_H
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <string>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
using namespace std;
#endif
```

We previously implemented a mechanism to load a fragment and vertex shader program from files, and we will reuse the code here (`shader.cpp` and `shader.hpp`). However, we will modify the actual vertex and shader programs as follows.

For the vertex shader (`transform.vert`), we will implement the following:

```
#version 150
in vec2 UV;
out vec4 color;
uniform sampler2D textureSampler;
void main(){
    color = texture(textureSampler, UV).rgba;
}
```

For the fragment shader (`texture.frag`), we will implement the following:

```
#version 150
in vec3 vertexPosition_modelspace;
in vec2 vertexUV;
out vec2 UV;
uniform mat4 MVP;
void main(){
    //position of the vertex in clip space
    gl_Position = MVP * vec4(vertexPosition_modelspace,1);
    UV = vertexUV;
}
```

For the texture objects, in `texture.cpp`, we provide a mechanism to load images or video stream into the texture memory. We also take advantage of the SOIL library for simple image loading and the OpenCV library for more advanced video stream handling and filtering (refer to the next section).

In `texture.cpp`, we will implement the following:

1. Include the `texture.hpp` header and SOIL library header for simple image loading:

```
#include "texture.hpp"
#include <SOIL.h>
```

2. Define the initialization of texture objects and set up all parameters:

```
GLuint initializeTexture(const unsigned char *image_data,
    int width, int height, GLenum format){
    GLuint textureID=0;
    //create and bind one texture element
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    /* Specify target texture. The parameters describe the
       format and type of the image data */
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
        format, GL_UNSIGNED_BYTE, image_data);
    /* Set the wrap parameter for texture coordinate s & t to
       GL_CLAMP, which clamps the coordinates within [0, 1] */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
        GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
        GL_CLAMP_TO_EDGE);
    /* Set the magnification method to linear and return
       weighted average of four texture elements closest to
       the center of the pixel */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_LINEAR);
    /* Choose the mipmap that most closely matches the size of
       the pixel being textured and use the GL_NEAREST
       criterion (the texture element nearest to the center
       of the pixel) to produce a texture value. */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR_MIPMAP_LINEAR);
    glGenerateMipmap(GL_TEXTURE_2D);
    return textureID;
}
```

3. Define the routine to update the texture memory:

```
void updateTexture(const unsigned char *image_data, int width, int
height, GLenum format){
    // Update Texture
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height,
    format, GL_UNSIGNED_BYTE, image_data);
    /* Sets the wrap parameter for texture coordinates s & t to
       GL_CLAMP, which clamps the coordinates within [0, 1]. */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
    /* Set the magnification method to linear and return
       weighted average of four texture elements closest to
       the center of the pixel */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
    /* Choose the mipmap that most closely matches the size of
       the pixel being textured and use the GL_NEAREST
       criterion (the texture element nearest to the center
       of the pixel) to produce a texture value. */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR_MIPMAP_LINEAR);
    glGenerateMipmap(GL_TEXTURE_2D);
}
```

4. Finally, implement the texture-loading mechanism for images. The function takes the image path and automatically converts the image into various compatible formats for the texture object:

```
GLuint loadImageToTexture(const char * imagepath) {
    int width, height, channels;
    GLuint textureID=0;
    //Load the images and convert them to RGBA format
    unsigned char* image = SOIL_load_image(imagepath, &width,
    &height, &channels, SOIL_LOAD_RGBA);
    if(!image){
        printf("Failed to load image %s\n", imagepath);
        return textureID;
    }
    printf("Loaded Image: %d x %d - %d channels\n", width,
    height, channels);
    textureID=initializeTexture(image, width, height,
    GL_RGBA);
    SOIL_free_image_data(image);
    return textureID;
}
```

On the controller front, we capture the arrow keys and modify the camera model parameter in real time. This allows us to change the position and orientation of the camera as well as the angle of view. In `controls.cpp`, we implement the following:

1. Include the GLM library header and the `controls.hpp` header for the projection matrix and view matrix computations:

```
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "controls.hpp"
```

2. Define global variables (camera parameters as well as view and projection matrices) to be updated after each frame:

```
//initial position of the camera
glm::vec3 g_position = glm::vec3( 0, 0, 2 );
const float speed = 3.0f; // 3 units / second
float g_initial_fov = glm::pi<float>()*0.4f;
//the view matrix and projection matrix
glm::mat4 g_view_matrix;
glm::mat4 g_projection_matrix;
```

3. Create helper functions to return the most updated view matrix and projection matrix:

```
glm::mat4 getViewMatrix() {
    return g_view_matrix;
}
glm::mat4 getProjectionMatrix() {
    return g_projection_matrix;
}
```

4. Compute the view matrix and projection matrix based on the user input:

```
void computeViewProjectionMatrices(GLFWwindow* window) {
    static double last_time = glfwGetTime();
    // Compute time difference between current and last frame
    double current_time = glfwGetTime();
    float delta_time = float(current_time - last_time);
    int width, height;
    glfwGetWindowSize(window, &width, &height);
    //direction vector for movement
    glm::vec3 direction(0, 0, -1);
    //up vector
    glm::vec3 up = glm::vec3(0, -1, 0);
    if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
        g_position += direction * delta_time * speed;
    }
}
```

```
else if (glfwGetKey(window, GLFW_KEY_DOWN) ==  
        GLFW_PRESS){  
    g_position -= direction * delta_time * speed;  
}  
else if (glfwGetKey(window, GLFW_KEY_RIGHT) ==  
        GLFW_PRESS){  
    g_initial_fov -= 0.1 * delta_time * speed;  
}  
else if (glfwGetKey(window, GLFW_KEY_LEFT) ==  
        GLFW_PRESS){  
    g_initial_fov += 0.1 * delta_time * speed;  
}  
/* update projection matrix: Field of View, aspect ratio,  
   display range : 0.1 unit <-> 100 units */  
g_projection_matrix = glm::perspective(g_initial_fov,  
                                         (float)width/(float)height, 0.1f, 100.0f);  
  
// update the view matrix  
g_view_matrix = glm::lookAt(  
    g_position,           // camera position  
    g_position+direction, // viewing direction  
    up                  // up direction  
);  
last_time = current_time;  
}
```

In `main.cpp`, we will use the various previously defined functions to complete the implementation:

1. Include the GLFW and GLM libraries as well as our helper functions, which are stored in separate files inside a folder called the `common` folder:

```
#define GLM_FORCE_RADIANS  
#include <stdio.h>  
#include <stdlib.h>  
#include <GL/glew.h>  
#include <GLFW/glfw3.h>  
#include <glm/glm.hpp>  
#include <glm/gtc/matrix_transform.hpp>  
using namespace glm;  
#include <common/shader.hpp>  
#include <common/texture.hpp>  
#include <common/controls.hpp>  
#include <common/common.h>
```

2. Define all global variables for the setup:

```
GLFWwindow* g_window;
const int WINDOWS_WIDTH = 1280;
const int WINDOWS_HEIGHT = 720;
float aspect_ratio = 3.0f/2.0f;
float z_offset = 2.0f;
float rotateY = 0.0f;
float rotateX = 0.0f;
//Our vertices
static const GLfloat g_vertex_buffer_data[] = {
    -aspect_ratio,-1.0f,z_offset,
    aspect_ratio,-1.0f,z_offset,
    aspect_ratio,1.0f,z_offset,
    -aspect_ratio,-1.0f,z_offset,
    aspect_ratio,1.0f,z_offset,
    -aspect_ratio,1.0f,z_offset
};
//UV map for the vertices
static const GLfloat g_uv_buffer_data[] = {
    1.0f, 0.0f,
    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f
};
```

3. Define the keyboard callback function:

```
static void key_callback(GLFWwindow* window, int key, int
    scancode, int action, int mods)
{
    if (action != GLFW_PRESS && action != GLFW_REPEAT)
        return;
    switch (key)
    {
        case GLFW_KEY_ESCAPE:
            glfwSetWindowShouldClose(window, GL_TRUE);
            break;
        case GLFW_KEY_SPACE:
            rotateX=0;
            rotateY=0;
            break;
        case GLFW_KEY_Z:
            rotateX+=0.01;
```

```
        break;
    case GLFW_KEY_X:
        rotateX-=0.01;
        break;
    case GLFW_KEY_A:
        rotateY+=0.01;
        break;
    case GLFW_KEY_S:
        rotateY-=0.01;
        break;
    default:
        break;
    }
}
```

4. Initialize the GLFW library with the OpenGL core profile enabled:

```
int main(int argc, char **argv)
{
    //Initialize the GLFW
    if(!glfwInit()){
        fprintf( stderr, "Failed to initialize GLFW\n" );
        exit(EXIT_FAILURE);
    }

    //enable anti-alising 4x with GLFW
    glfwWindowHint(GLFW_SAMPLES, 4);
    //specify the client API version
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
    //make the GLFW forward compatible
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
    //enable the OpenGL core profile for GLFW
    glfwWindowHint(GLFW_OPENGL_PROFILE,
        GLFW_OPENGL_CORE_PROFILE);
```

5. Set up the GLFW windows and keyboard input handlers:

```
//create a GLFW windows object
window = glfwCreateWindow(WINDOWS_WIDTH, WINDOWS_HEIGHT,
    "Chapter 4 - Texture Mapping", NULL, NULL);
if(!window){
    fprintf( stderr, "Failed to open GLFW window. If you
        have an Intel GPU, they are not 3.3 compatible. Try
        the 2.1 version of the tutorials.\n" );
    glfwTerminate();
    exit(EXIT_FAILURE);
```

```
    }
    /* make the context of the specified window current for
       the calling thread */
    glfwMakeContextCurrent(window);
    glfwSwapInterval(1);
    glewExperimental = true; // Needed for core profile
    if (glewInit() != GLEW_OK) {
        fprintf(stderr, "Final to Initialize GLEW\n");
        glfwTerminate();
        exit(EXIT_FAILURE);
    }
    //keyboard input callback
    glfwSetInputMode(window,GLFW_STICKY_KEYS,GL_TRUE);
    glfwSetKeyCallback(window, key_callback);
```

6. Set a black background and enable alpha blending for various visual effects:

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
 glEnable(GL_BLEND);
 glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

7. Load the vertex shader and fragment shader:

```
GLuint program_id = LoadShaders( "transform.vert",
                                  "texture.frag" );
```

8. Load an image file into the texture object using the SOIL library:

```
char *filepath;
//load the texture from image with SOIL
if(argc<2){
    filepath = (char*)malloc(sizeof(char)*512);
    sprintf(filepath, "texture.png");
}
else{
    filepath = argv[1];
}

int width;
int height;
GLuint texture_id = loadImageToTexture(filepath, &width,
                                         &height);

aspect_ratio = (float)width/(float)height;
if(!texture_id){
    //if we get 0 with no texture
    glfwTerminate();
    exit(EXIT_FAILURE);
}
```

9. Get the locations of the specific variables in the shader programs:

```
//get the location for our "MVP" uniform variable  
GLuint matrix_id = glGetUniformLocation(program_id,  
    "MVP");  
//get a handler for our "myTextureSampler" uniform  
GLuint texture_sampler_id =  
    glGetUniformLocation(program_id, "textureSampler");  
//attribute ID for the variables  
GLint attribute_vertex, attribute_uv;  
attribute_vertex = glGetAttribLocation(program_id,  
    "vertexPosition_modelspace");  
attribute_uv = glGetAttribLocation(program_id,  
    "vertexUV");
```

10. Define our **Vertex Array Objects (VAO)**:

```
GLuint vertex_array_id;  
 glGenVertexArrays(1, &vertex_array_id);  
 glBindVertexArray(vertex_array_id);
```

11. Define our VAO for vertices and UV mapping:

```
//initialize the vertex buffer memory.  
GLuint vertex_buffer;  
 glGenBuffers(1, &vertex_buffer);  
 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);  
 glBufferData(GL_ARRAY_BUFFER,  
     sizeof(g_vertex_buffer_data), g_vertex_buffer_data,  
     GL_STATIC_DRAW);  
//initialize the UV buffer memory  
GLuint uv_buffer;  
 glGenBuffers(1, &uv_buffer);  
 glBindBuffer(GL_ARRAY_BUFFER, uv_buffer);  
 glBufferData(GL_ARRAY_BUFFER, sizeof(g_uv_buffer_data),  
     g_uv_buffer_data, GL_STATIC_DRAW);
```

12. Use the shader program and bind all texture units and attribute buffers:

```
glUseProgram(program_id);  
//binds our texture in Texture Unit 0  
glActiveTexture(GL_TEXTURE0);  
 glBindTexture(GL_TEXTURE_2D, texture_id);  
 glUniform1i(texture_sampler_id, 0);  
//1st attribute buffer: vertices for position  
 glEnableVertexAttribArray(attribute_vertex);  
 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);  
 glVertexAttribPointer(attribute_vertex, 3, GL_FLOAT,  
     GL_FALSE, 0, (void*)0);
```

```
//2nd attribute buffer: UVs mapping  
glEnableVertexAttribArray(attribute_uv);  
glBindBuffer(GL_ARRAY_BUFFER, uv_buffer);  
glVertexAttribPointer(attribute_uv, 2, GL_FLOAT,  
GL_FALSE, 0, (void*)0);
```

13. In the main loop, clear the screen and depth buffers:

```
//time-stamping for performance measurement  
double previous_time = glfwGetTime();  
do{  
    //clear the screen  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

14. Compute the transforms and store the information in the shader variables:

```
//compute the MVP matrix from keyboard and mouse input  
computeMatricesFromInputs(g_window);  
//obtain the View and Model Matrix for rendering  
glm::mat4 projection_matrix = getProjectionMatrix();  
glm::mat4 view_matrix = getViewMatrix();  
glm::mat4 model_matrix = glm::mat4(1.0);  
model_matrix = glm::rotate(model_matrix,  
    glm::pi<float>() * rotateY, glm::vec3(0.0f, 1.0f,  
    0.0f));  
model_matrix = glm::rotate(model_matrix,  
    glm::pi<float>() * rotateX, glm::vec3(1.0f, 0.0f,  
    0.0f));  
glm::mat4 mvp = projection_matrix * view_matrix *  
    model_matrix;  
//send our transformation to the currently bound shader  
//in the "MVP" uniform variable  
glUniformMatrix4fv(matrix_id, 1, GL_FALSE, &mvp[0][0]);
```

15. Draw the elements and flush the screen:

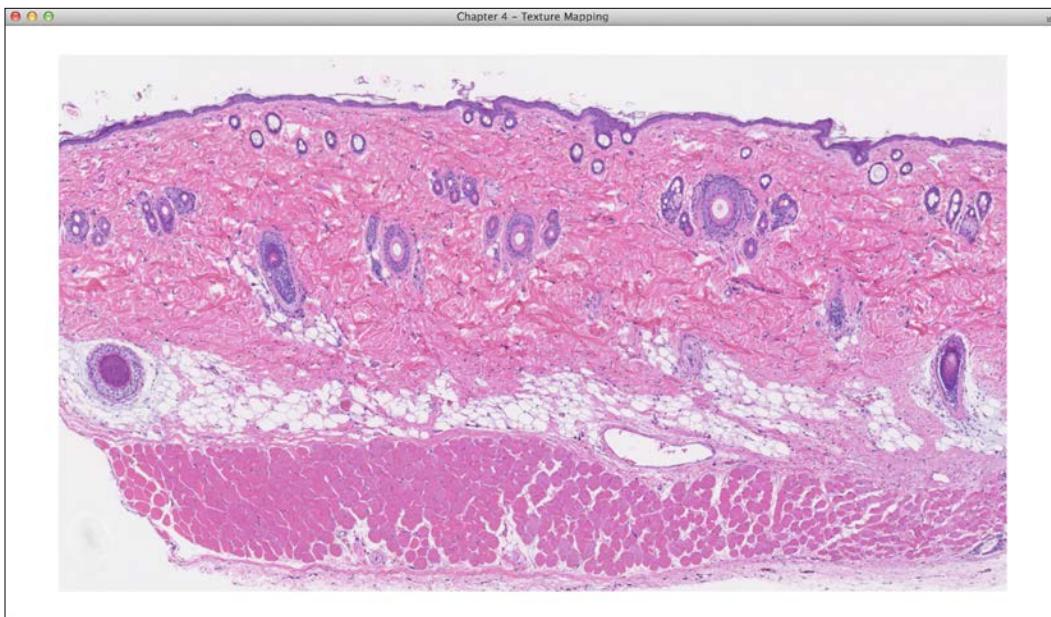
```
glDrawArrays(GL_TRIANGLES, 0, 6); //draw a square  
//swap buffers  
glfwSwapBuffers(window);  
glfwPollEvents();
```

16. Finally, define the conditions to exit the main loop and clear all the memory to exit the program gracefully:

```
    } // Check if the ESC key was pressed or the window was closed
    while(!glfwWindowShouldClose(window) &&
        glfwGetKey(window, GLFW_KEY_ESCAPE ) !=GLFW_PRESS );
    glDisableVertexAttribArray(attribute_vertex);
    glDisableVertexAttribArray(attribute_uv);
    // Clean up VBO and shader
    glDeleteBuffers(1, &vertex_buffer);
    glDeleteBuffers(1, &uv_buffer);
    glDeleteProgram(program_id);
    glDeleteTextures(1, &texture_id);
    glDeleteVertexArrays(1, &vertex_array_id);
    // Close OpenGL window and terminate GLFW
    glfwDestroyWindow(g_window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
}
```

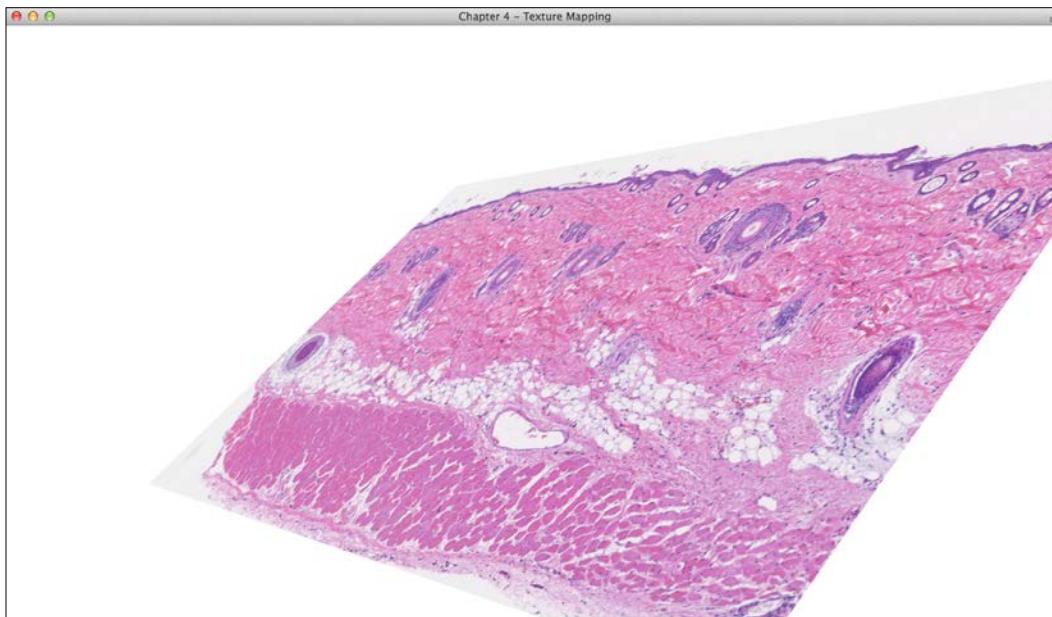
How it works...

To demonstrate the use of the framework for data visualization, we will apply it to the visualization of a histology slide (an H&E cross-section of a skin sample), as shown in the following screenshot:

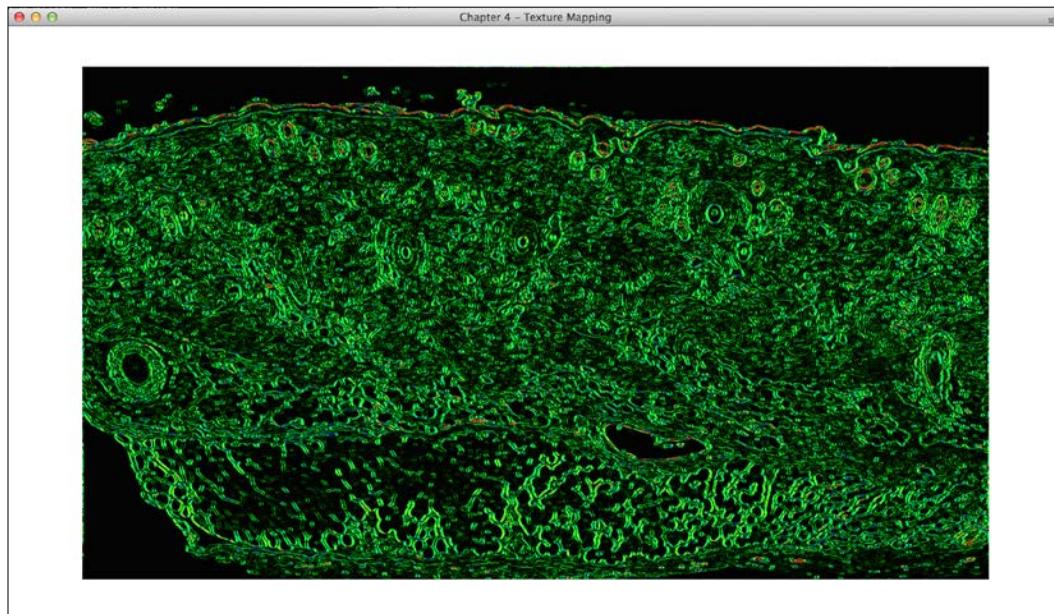


An important difference between this demo and the previous one is that here, we actually load an image into the texture memory (`texture.cpp`). To facilitate this task, we use the SOIL library call (`SOIL_load_image`) to load the histology image in the RGBA format (`GL_RGBA`) and the `glTexImage2D` function call to generate a texture image that can be read by shaders.

Another important difference is that we can now dynamically recompute the view (`g_view_matrix`) and projection (`g_projection_matrix`) matrices to enable an interactive and interesting visualization of an image in the 3D space. Note that the GLM library header is included to facilitate the matrix computations. Using the keyboard inputs (up, down, left, and right) defined in `controls.cpp` with the GLFW library calls, we can zoom in and out of the slide as well as adjust the view angle, which gives an interesting perspective of the histology image in the 3D virtual space. Here is a screenshot of the image viewed with a different perspective:



Yet another unique feature of the current OpenGL-based framework is illustrated by the following screenshot, which is generated with a new image filter implemented into the fragment shader that highlights the edges in the image. This shows the endless possibilities for the real-time interactive visualization and processing of 2D images using OpenGL rendering pipeline without compromising on CPU performance. The filter implemented here will be discussed in the next section.



Real-time video rendering with filters

The GLSL shader provides a simple way to perform highly parallelized processing. On top of the texture mapping shown previously, we will demonstrate how to implement a simple video filter that postprocesses the end results of the buffer frame using the fragment shader. To illustrate this technique, we implement the Sobel Filter along with a heat map rendered using the OpenGL pipeline. The heat map function that was previously implemented in *Chapter 3, Interactive 3D Data Visualization*, will now be directly ported to GLSL with very minor changes.

The Sobel operator is a simple image processing technique frequently used in computer vision algorithms such as edge detection. This operator can be defined as a convolution operation with a 3 x 3 kernel, shown as follows:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I(x, y), \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I(x, y)$$

G_x and G_y are results of the horizontal and vertical derivatives of an image, respectively, from the convolution operation of image I at the pixel location (x, y) .

We can also perform a sum of squares operation to approximate the gradient magnitude of the image:

$$G^2 = G_x^2 + G_y^2$$

Getting ready

This demo builds on top of the previous section, where an image was rendered. In this section, we will demonstrate the rendering of an image sequence or a video with the use of OpenCV library calls to handle videos. Inside `common.h`, we will add the following lines to include the OpenCV libraries:

```
#include <opencv2/opencv.hpp>
using namespace cv;
```

How to do it...

Now, let's complete the implementation as follows:

1. First, modify `main.cpp` to enable video processing using OpenCV. Essentially, instead of loading an image, feed the individual frames of a video into the same pipeline:

```
char *filepath;
if(argc<2){
    filepath = (char*)malloc(sizeof(char)*512);
    sprintf(filepath, "video.mov");
}
else{
    filepath = argv[1];
}
//Handling Video input with OpenCV
VideoCapture cap(filepath); // open the default camera
Mat frame;
if (!cap.isOpened()) { // check if we succeeded
    printf("Cannot open files\n");
    glfwTerminate();
    exit(EXIT_FAILURE);
} else{
    cap >> frame; // get a new frame from camera
```

```
    printf("Got Video, %d x %d\n", frame.size().width,
           frame.size().height);
}
cap >> frame; // get a new frame from camera
GLuint texture_id = initializeTexture(frame.data,
                                         frame.size().width, frame.size().height, GL_BGR);
aspect_ratio = (float)frame.size().width/
               (float)frame.size().height;
```

2. Then, add the update function in the main loop to update the texture in every frame:

```
/* get the video feed, reset to beginning if it reaches
   the end of the video */
if(!cap.grab()){
    printf("End of Video, Resetting\n");
    cap.release();
    cap.open(filepath); // open the default camera
}
cap >> frame; // get a new frame from camera
//update the texture with the new frame
updateTexture(frame.data, frame.size().width,
               frame.size().height, GL_BGR);
```

3. Next, modify the fragment shader and rename it `texture_sobel.frag` (from `texture.frag`). In the `main` function, we will outline the overall processing (process the texture buffers with the Sobel filter and heat map renderer):

```
void main() {
    //compute the results of Sobel filter
    float graylevel = sobel_filter();
    color = heatMap(graylevel, 0.1, 3.0);
}
```

4. Now, implement the Sobel filter algorithm that takes the neighboring pixels to compute the result:

```
float sobel_filter()
{
    float dx = 1.0 / float(1280);
    float dy = 1.0 / float(720);

    float s00 = pixel_operator(-dx, dy);
    float s10 = pixel_operator(-dx, 0);
    float s20 = pixel_operator(-dx, -dy);
    float s01 = pixel_operator(0.0, dy);
    float s21 = pixel_operator(0.0, -dy);
    float s02 = pixel_operator(dx, dy);
```

```

        float s12 = pixel_operator(dx, 0.0);
        float s22 = pixel_operator(dx, -dy);
        float sx = s00 + 2 * s10 + s20 - (s02 + 2 * s12 + s22);
        float sy = s00 + 2 * s01 + s02 - (s20 + 2 * s21 + s22);
        float dist = sx * sx + sy * sy;
        return dist;
    }
}

```

5. Define the helper function that computes the brightness value:

```

float rgb2gray(vec3 color) {
    return 0.2126 * color.r + 0.7152 * color.g + 0.0722 *
        color.b;
}

```

6. Create a helper function for the per-pixel operator operations:

```

float pixel_operator(float dx, float dy) {
    return rgb2gray(texture( textureSampler, UV +
        vec2(dx,dy)).rgb);
}

```

7. Lastly, define the heat map renderer prototype and implement the algorithm for better visualization of the range of values:

```

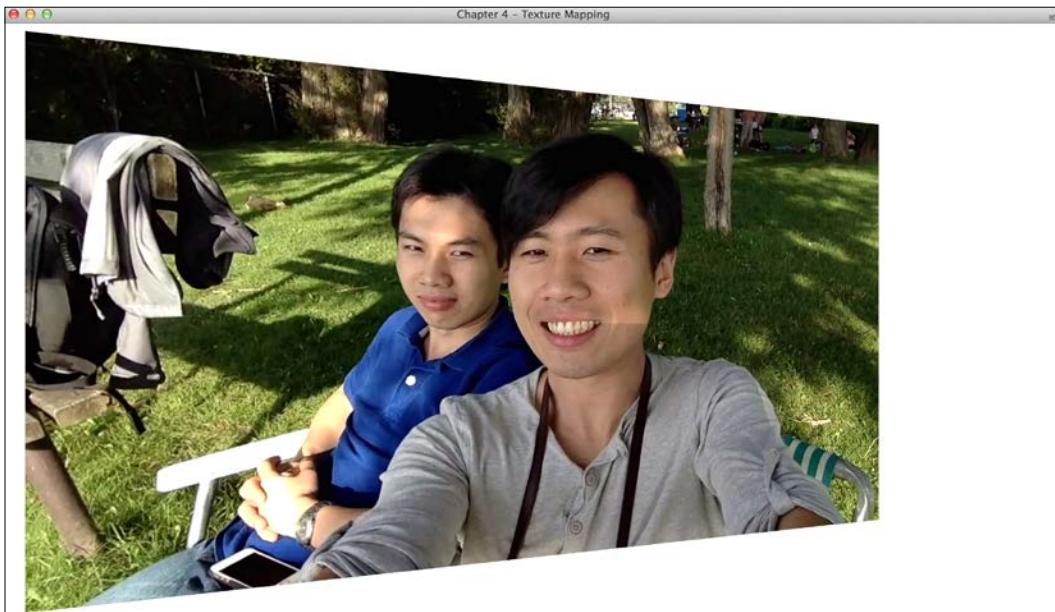
vec4 heatMap(float v, float vmin, float vmax) {
    float dv;
    float r, g, b;
    if (v < vmin)
        v = vmin;
    if (v > vmax)
        v = vmax;
    dv = vmax - vmin;
    if(v == 0){
        return vec4(0.0, 0.0, 0.0, 1.0);
    }
    if (v < (vmin + 0.25f * dv)) {
        r = 0.0f;
        g = 4.0f * (v - vmin) / dv;
    } else if (v < (vmin + 0.5f * dv)) {
        r = 0.0f;
        b = 1.0f + 4.0f * (vmin + 0.25f * dv - v) / dv;
    } else if (v < (vmin + 0.75f * dv)) {

```

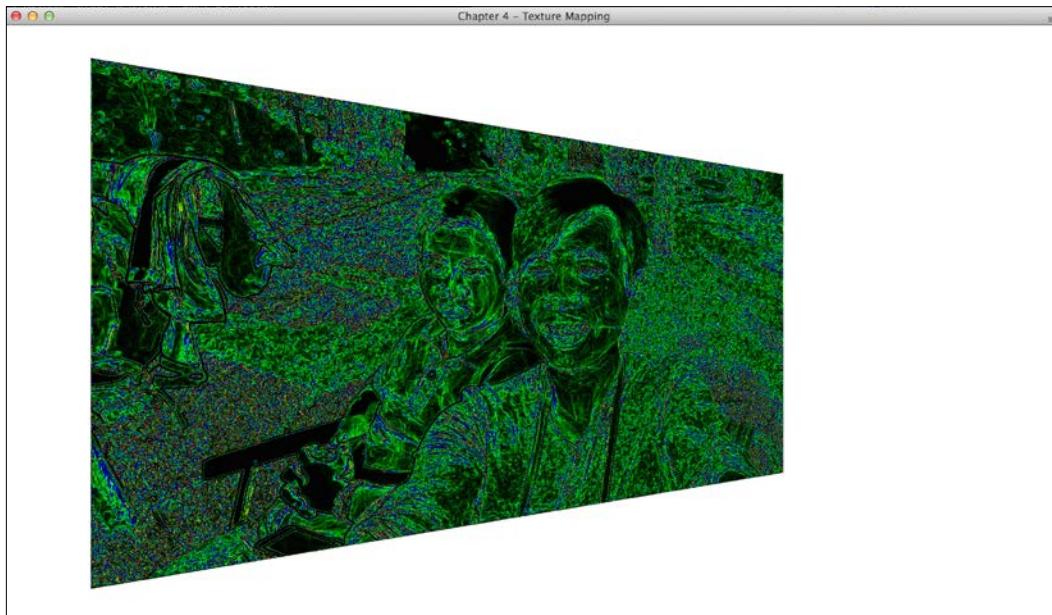
```
r = 4.0f * (v - vmin - 0.5f * dv) / dv;
b = 0.0f;
} else {
    g = 1.0f + 4.0f * (vmin + 0.75f * dv - v) / dv;
    b = 0.0f;
}
return vec4(r, g, b, 1.0);
}
```

How it works...

This demo effectively opens up the possibility of rendering any image sequence with real-time processing using the OpenGL pipeline at the fragment shading stage. The following screenshot is an example that illustrates the use of this powerful OpenGL framework to display one frame of a video (showing the authors of the book) without the Sobel filter enabled:



Now, with the Sobel filter and heat map rendering enabled, we see an interesting way to visualize the world using real-time OpenGL texture mapping and processing using custom shaders:



Further fine-tuning of the threshold parameters and converting the result into grayscale (in the `texture_sobel.frag` file) leads to an aesthetically interesting output:

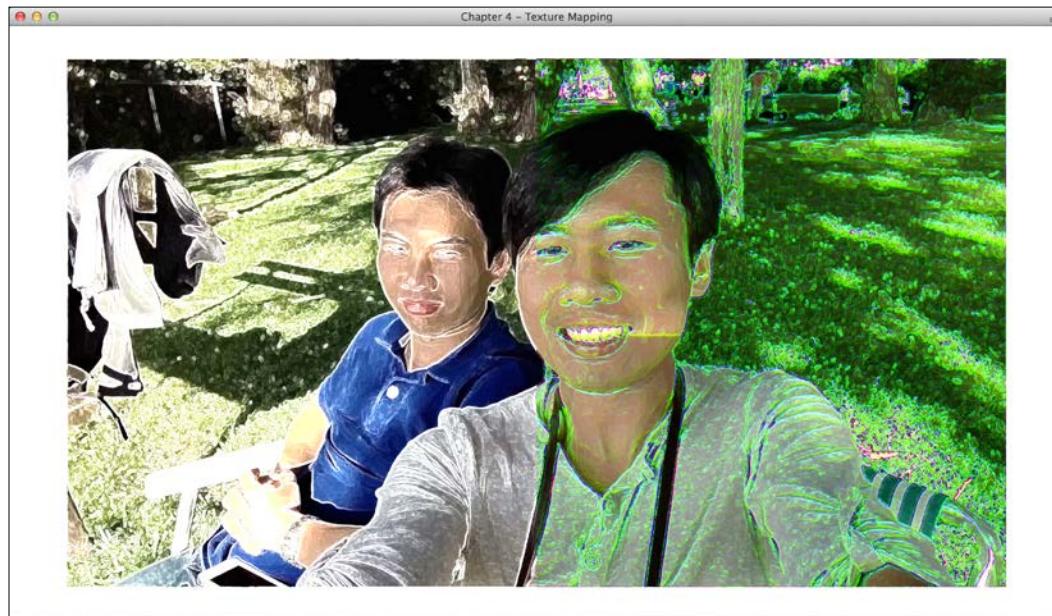
```
void main() {
    //compute the results of Sobel filter
    float graylevel = sobel_filter();
    color = vec4(graylevel, graylevel, graylevel, 1.0);
}
```



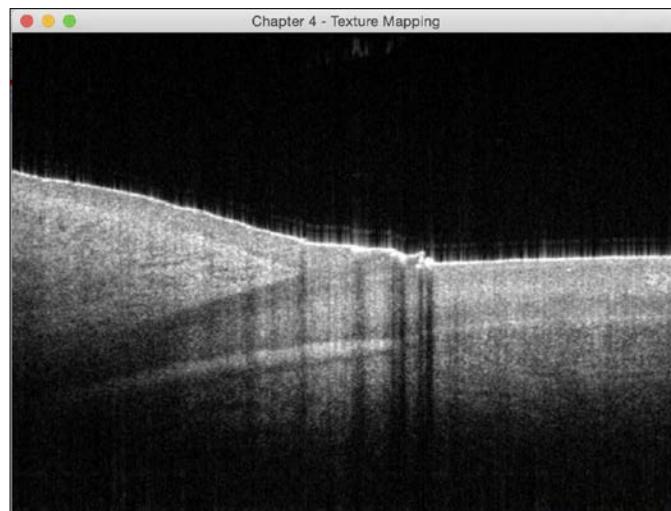
In addition, we can blend these results with the original video feed to create filtered effects in real time by modifying the main function in the shader program (`texture_sobel.frag`):

```
void main(){
    //compute the results of Sobel filter
    float graylevel = sobel_filter();
    //process the right side of the image
    if(UV.x > 0.5)
        color = heatMap(graylevel, 0.0, 3.0) + texture
            (textureSampler, UV);
```

```
else
color = vec4(graylevel, graylevel, graylevel, 1.0) + texture
(textureSampler, UV);
}
```

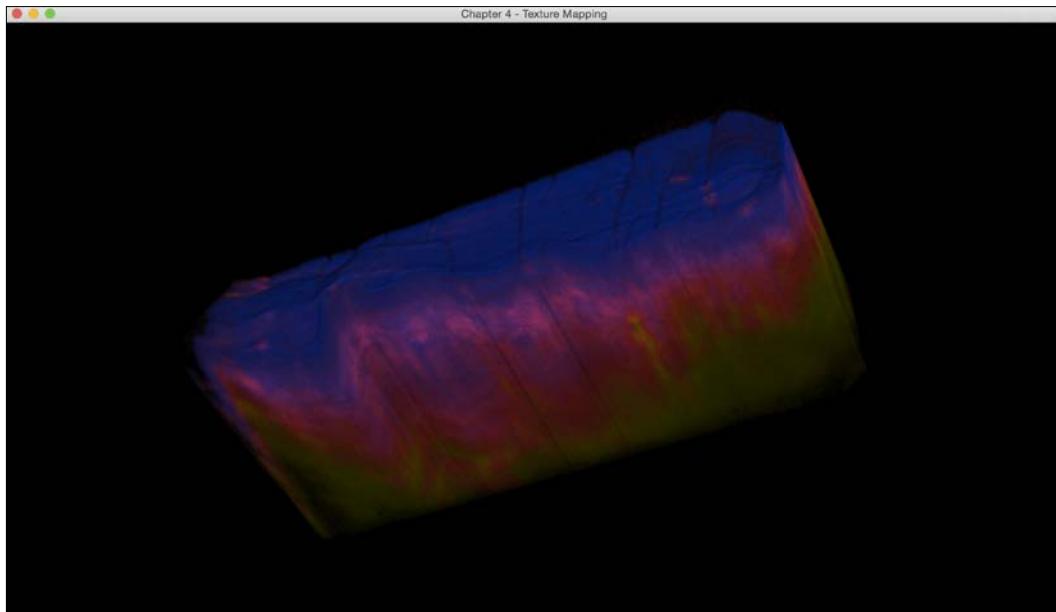


To illustrate the use of the exact same program to visualize imaging datasets, here is an example that shows a volumetric dataset of a human finger imaged with **Optical Coherence Tomography (OCT)**, simply by changing the input video's filename:



This screenshot represents one of 256 cross-sectional images of the nail bed in this volumetric OCT dataset (which is exported in a movie file format).

Here is another example that shows a volumetric dataset of a scar specimen imaged with **Polarization-Sensitive Optical Coherence Tomography (PS-OCT)**, which provides label-free, intrinsic contrast to the scar region:



In this case, the volumetric PS-OCT dataset was rendered with the ImageJ 3D Viewer and converted into a movie file. The colors denote the **Degree of Polarization (DOP)**, which is a measure of the randomness of the polarization states of light (a low DOP in yellow/green and a high DOP in blue), in the skin. The scar region is characterized by a high DOP compared to the normal skin.

As we have demonstrated here, this program can be easily adopted (by changing the input video source) to display many types of datasets, such as endoscopy videos or other volumetric imaging datasets. The utility of OpenGL becomes apparent in demanding applications that require real-time processing of very large datasets.

5

Rendering of Point Cloud Data for 3D Range-sensing Cameras

In this chapter, we will cover the following topics:

- ▶ Getting started with the Microsoft Kinect (PrimeSense) 3D range-sensing camera
- ▶ Capturing raw data from depth-sensing cameras
- ▶ OpenGL point cloud rendering with texture mapping and overlays

Introduction

The purpose of this chapter is to introduce the techniques to visualize another interesting and emerging class of data: depth information from 3D range-sensing cameras. Devices with 3D depth sensors are hitting the market everyday, and companies such as Intel, Microsoft, SoftKinetic, PMD, Structure Sensor, and Meta (wearable Augmented Reality eyeglasses) are all using these novel 3D sensing devices to track user inputs, such as hand gestures for interaction and/or tracking a user's environment. An interesting integration of 3D sensors with OpenGL is the ability to look at a scene in 3D from different perspectives, thereby enabling a virtual 3D fly-through of a scene captured with the depth sensors. In our case, for data visualization, being able to walk through a massive 3D dataset could be particularly powerful in scientific computing, urban planning, and many other applications that involve the visualization of 3D structures of a scene.

In this chapter, we propose a simplified pipeline that takes any 3D point data (X, Y, Z) with color (r, g, b) and renders these point clouds on the screen in real time. The point clouds will be obtained directly from real-world data using a 3D range-sensing camera. We will also provide ways to fly around the point cloud and have dynamic ways to adjust the camera's parameters. This chapter will build on the OpenGL graphics rendering pipeline discussed in the previous chapter, and we will show you a few additional tricks to filter the data with GLSL. We will display our depth information using our heat map generator to see the depth in 2D and remap this data to a 3D point cloud using texture mapping and perspective projection. This will allow us to see the real-life depth-based rendering of a scene and navigate around the scene from any perspective.

Getting started with the Microsoft Kinect (PrimeSense) 3D range-sensing camera

The Microsoft Kinect 3D range-sensing camera based on the PrimeSense technology is an interesting piece of equipment that enables the estimation of the 3D geometry of a scene through depth-sensing using light patterns. The 3D sensor has an active infrared laser projector, which emits encoded speckle light patterns. The sensors allow users to capture color images and provide a 3D depth map at a resolution of 640 x 480. Since the Kinect sensor is an active sensor, it is invariant to indoor lighting condition (that is, it even works in the dark) and enables many applications, such as gesture and pose tracking as well as 3D scanning and reconstruction.

In this section, we will demonstrate how to set up this type of range-sensing camera, as an example. While we do not require readers to purchase a 3D range-sensing camera for this chapter (since we will provide the raw data captured on this device for the purpose of running our demos), we will demonstrate how one can set up the device to capture data directly, primarily for those who are interested in further experimenting with real-time 3D data.

How to do it...

Windows users can download the OpenNI 2 SDK and driver from <http://structure.io/openni> (or using the direct download link: <http://com.occipital.openni.s3.amazonaws.com/OpenNI-Windows-x64-2.2.0.33.zip>) and follow the on-screen instructions. Linux users can download the OpenNI 2 SDK from the same website at <http://structure.io/openni>.

Mac users can install the OpenNI2 driver as follows:

1. Install libraries with Macport:

```
sudo port install libtool  
sudo port install libusb +universal
```

2. Download OpenNI2 from <https://github.com/occipital/openni2>.
3. Compile the source code with the following commands:

```
cd OpenNI2-master  
make  
cd Bin/x64-Release/
```

4. Run the SimpleViewer executable:

```
./SimpleViewer
```

If you are using a computer with a USB 3.0 interface, it is important that you first upgrade the firmware for the PrimeSense sensor to version 1.0.9 (http://das1.mem.drexel.edu/wiki/images/5/51/FWUpdate_RD109-112_5.9.2.zip). This upgrade requires a Windows platform. Note that the Windows driver for the PrimeSense sensor must be installed (<http://structure.io/openni>) for you to proceed. Execute the FWUpdate_RD109-112_5.9.2.exe file, and the firmware will be automatically upgraded. Further details on the firmware can be found at http://das1.mem.drexel.edu/wiki/index.php/4._Updating_Firmware_for_Primesense.

See also

Detailed technical specifications of the Microsoft Kinect 3D system can be obtained from <http://msdn.microsoft.com/en-us/library/jj131033.aspx>, and further installation instructions and prerequisites to build OpenNI2 drivers can be found at <https://github.com/occipital/openni2>.

In addition, Microsoft Kinect V2 is also available and is compatible with Windows. The new sensor provides higher resolution images and better depth fidelity. More information about the sensor, as well as the Microsoft Kinect SDK, can be found at <https://www.microsoft.com/en-us/kinectforwindows>.

Capturing raw data from depth-sensing cameras

Now that you have installed the prerequisite libraries and drivers, we will demonstrate how to capture raw data from your depth-sensing camera.

How to do it...

To capture sensor data directly in a binary format, implement the following function:

```
void writeDepthBuffer(openni::VideoFrameRef depthFrame) {
    static int depth_buffer_counter=0;
    char file_name [512];
    sprintf(file_name, "%s%d.bin", "depth_frame",
            depth_buffer_counter);
    openni::DepthPixel *depthPixels = new
        openni::DepthPixel[depthFrame.getHeight()*depthFrame.getWidth()];
    memcpy(depthPixels, depthFrame.getData(),
           depthFrame.getHeight()*depthFrame.getWidth()*sizeof(uint16_t));
    std::fstream myFile (file_name, std::ios::out
        | std::ios::binary);
    myFile.write ((char*)depthPixels,
                  depthFrame.getHeight()*depthFrame.getWidth()*sizeof(uint16_t));
    depth_buffer_counter++;
    printf("Dumped Depth Buffer %d\n", depth_buffer_counter);
    myFile.close();
    delete depthPixels;
}
```

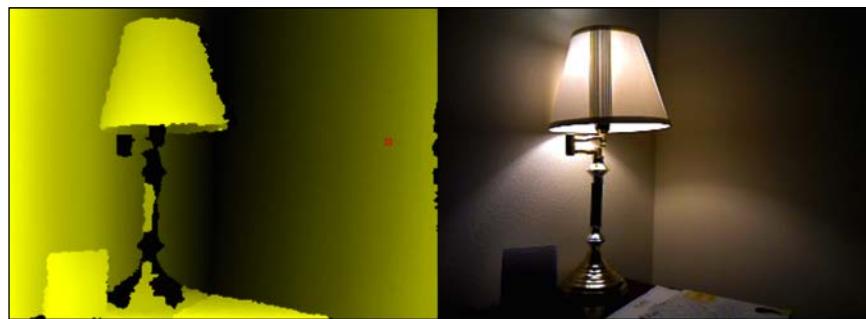
Similarly, we also capture the raw RGB color data with the following implementation:

```
void writeColorBuffer(openni::VideoFrameRef colorFrame) {
    static int color_buffer_counter=0;
    char file_name [512];
    sprintf(file_name, "%s%d.bin", "color_frame",
            color_buffer_counter);
    //basically unsigned char*
    const openni::RGB888Pixel* imageBuffer = (const
        openni::RGB888Pixel*)colorFrame.getData();
    std::fstream myFile (file_name, std::ios::out |
        std::ios::binary);
    myFile.write ((char*)imageBuffer,
                  colorFrame.getHeight()*colorFrame.getWidth()*sizeof(uint8_t)*3);
    color_buffer_counter++;
    printf("Dumped Color Buffer %d, %d, %d\n",
           colorFrame.getHeight(), colorFrame.getWidth(),
           color_buffer_counter);
    myFile.close();
}
```

The preceding code snippet can be added to any sample code within the OpenNI2 SDK that provides depth and color data visualization (to enable raw data capture). We recommend that you modify the `Viewer.cpp` file in the `OpenNI2-master/Samples/SimpleViewer` folder. The modified sample code is included in our code package. To capture raw data, press `R` and the data will be stored in the `depth_frame0.bin` and `color_frame0.bin` files.

How it works...

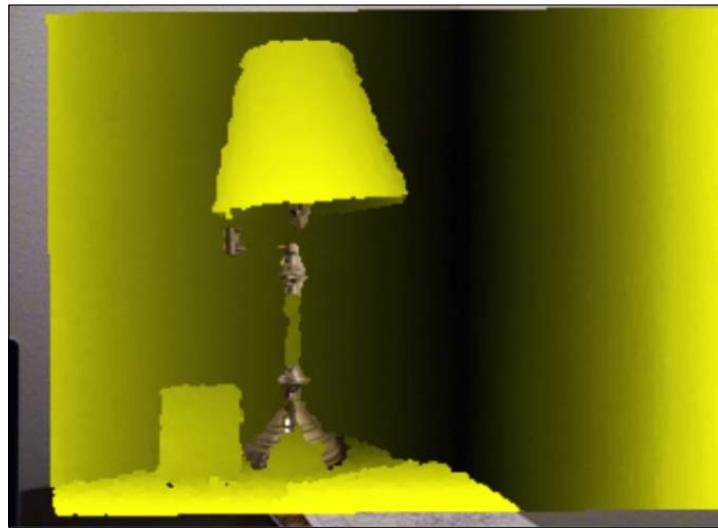
The depth sensor returns two streams of data in real time. One data stream is a 3D depth map, which is stored in 16-bits unsigned short data type (see the following figure on the left-hand side). Another data stream is a color image (see the following figure on the right-hand side), which is stored in a 24 bits per pixel, RGB888 format (that is, the memory is aligned in the R, G, and B order, and $8 \text{ bits} * 3 \text{ channels} = 24 \text{ bits}$ are used per pixel).



The binary data is written directly to the hard disk without compression or modification to the data format. On the client side, we read the binary files as if there is a continuous stream of data and color data pairs arriving synchronously from the hardware device. The OpenNI2 driver provides the mechanism to interface with the PrimeSense-based sensors (Microsoft Kinect or PS1080).

The `openni::VideoFrameRef depthFrame` variable, for example, stores the reference to the depth data buffer. By calling the `depthFrame.getData()` function, we obtain a pointer to the buffer in the `DepthPixel` format, which is equivalent to the unsigned short data type. Then, we write the binary data to a file using the `write()` function in the `fstream` library. Similarly, we perform the same task with the color image, but the data is stored in the `RGB888` format.

Additionally, we can enable the `setImageRegistrationMode (openni::IMAGE_REGISTRATION_DEPTH_TO_COLOR)` depth map registration function in OpenNI2 to automatically compute and map a depth value onto the color image. The depth map is overlaid onto the color image and is shown in the following figure:



In the next section, we will assume that the raw depth map is precalibrated with image registration by OpenNI2 and can be used to compute the real-world coordinates and UV mapping indices directly.

OpenGL point cloud rendering with texture mapping and overlays

We will build on the OpenGL framework discussed in the previous chapter for point cloud rendering in this section. The texture mapping technique introduced in the previous chapter can also be applied in the point cloud format. Basically, the depth sensor provides a set of vertices in real-world space (the depth map), and the color camera provides us with the color information of the vertices. UV mapping is a simple lookup table once the depth map and color camera are calibrated.

Getting ready

Readers should use the raw data provided for the subsequent demo or obtain their own raw data from a 3D range-sensing camera. In either case, we assume these filenames will be used to denote the raw data files: `depth_frame0.bin` and `color_frame0.bin`.

How to do it...

Similar to the previous chapter, we will divide the program into three major components: the main program (`main.cpp`), shader programs (`shader.cpp`, `shader.hpp`, `pointcloud.vert`, `pointcloud.frag`), and texture-mapping functions (`texture.cpp`, `texture.hpp`). The main program performs the essential tasks to set up the demo, while the shader programs perform the specialized processing. The texture-mapping functions provide a mechanism to load and map the color information onto the vertices. Finally, we modify the `control.cpp` file to provide more refined controls over the **fly-through** experience through various additional keyboard inputs (using the up, down, left, and right arrow keys to zoom in and out in addition to adjusting the rotation angles using the `a`, `s`, `x`, and `z` keys).

First, let's take a look at the shader programs. We will create two vertex and fragment shader programs inside the `pointcloud.vert` and `pointcloud.frag` files that are compiled and loaded by the program at runtime by using the `LoadShaders` function in the `shader.cpp` file.

For the `pointcloud.vert` file, we implement the following:

```
#version 150 core
// Input vertex data
in vec3 vertexPosition_modelspace;
in vec2 vertexUV;
// Output data: interpolated for each fragment.
out vec2 UV;
out vec4 color_based_on_position;
// Values that stay constant for the whole mesh
uniform mat4 MVP;
//heat map generator
vec4 heatMap(float v, float vmin, float vmax) {
    float dv;
    float r=1.0f, g=1.0f, b=1.0f;
    if (v < vmin)
        v = vmin;
    if (v > vmax)
        v = vmax;
    dv = vmax - vmin;
    if (v < (vmin + 0.25f * dv)) {
        r = 0.0f;
        g = 4.0f * (v - vmin) / dv;
    } else if (v < (vmin + 0.5f * dv)) {
        r = 0.0f;
        b = 1.0f+4.0f*(vmin+0.25f*dv-v)/dv;
    } else if (v < (vmin + 0.75f * dv)) {
        r = 4.0f*(v-vmin-0.5f*dv)/dv;
```

```
b = 0.0f;
} else {
    g = 1.0f+4.0f*(vmin+0.75f*dv-v)/dv;
    b = 0.0f;
}
return vec4(r, g, b, 1.0);
}
void main(){
    // Output position of the vertex, in clip space: MVP * position
    gl_Position = MVP * vec4(vertexPosition_modelspace,1);
    color_based_on_position = heatMap(vertexPosition_modelspace.z, -
        3.0, 0.0f);
    UV = vertexUV;
}
```

For the pointcloud.frag file, we implement the following:

```
#version 150 core
in vec2 UV;
out vec4 color;
uniform sampler2D textureSampler;
in vec4 color_based_on_position;
void main(){
    //blend the depth map color with RGB
    color = 0.5f*texture(textureSampler,
        UV).rgba+0.5f*color_based_on_position;
}
```

Finally, let's put everything together with the main.cpp file:

1. Include prerequisite libraries and the shader program header files inside the common folder:

```
#include <stdio.h>
#include <stdlib.h>
//GLFW and GLEW libraries
#include <GL/glew.h>
#include <GLFW/glfw3.h>
//GLM library
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "../common/shader.hpp"
#include "../common/texture.hpp"
#include "../common/controls.hpp"
#include "../common/common.h"
#include <iostream>
```

2. Create a global variable for the GLFW window:

```
GLFWwindow* window;
```

3. Define the width and height of the input depth dataset as well as other window/camera properties for rendering:

```
const int WINDOWS_WIDTH = 640;
const int WINDOWS_HEIGHT = 480;
const int IMAGE_WIDTH = 320;
const int IMAGE_HEIGHT = 240;
float z_offset = 0.0f;
float rotateY = 0.0f;
float rotateX = 0.0f;
```

4. Define helper functions to parse the raw depth and color data:

```
unsigned short *readDepthFrame(const char *file_path) {
    int depth_buffer_size =
        DEPTH_WIDTH*DEPTH_HEIGHT*sizeof(unsigned short);
    unsigned short *depth_frame = (unsigned
        short*)malloc(depth_buffer_size);
    char *depth_frame_pointer = (char*)depth_frame;
    //read the binary file
    ifstream myfile;
    myfile.open (file_path, ios::binary | ios::in);
    myfile.read(depth_frame_pointer, depth_buffer_size);
    return depth_frame;
}
unsigned char *readColorFrame(const char *file_path) {
    int color_buffer_size =
        DEPTH_WIDTH*DEPTH_HEIGHT*sizeof(unsigned char)*3;
    unsigned char *color_frame = (unsigned
        char*)malloc(color_buffer_size);
    //read the binary file
    ifstream myfile;
    myfile.open (file_path, ios::binary | ios::in);
    myfile.read((char *)color_frame, color_buffer_size);
    return color_frame;
}
```

5. Create callback functions to handle key strokes:

```
static void key_callback(GLFWwindow* window, int key, int
    scancode, int action, int mods)
{
    if (action != GLFW_PRESS && action != GLFW_REPEAT)
        return;
```

```
switch (key)
{
    case GLFW_KEY_ESCAPE:
        glfwSetWindowShouldClose(window, GL_TRUE);
        break;
    case GLFW_KEY_SPACE:
        rotateX=0;
        rotateY=0;
        break;
    case GLFW_KEY_Z:
        rotateX+=0.01;
        break;
    case GLFW_KEY_X:
        rotateX-=0.01;
        break;
    case GLFW_KEY_A:
        rotateY+=0.01;
        break;
    case GLFW_KEY_S:
        rotateY-=0.01;
        break;
    default:
        break;
}
```

6. Start the main program with the initialization of the GLFW library:

```
int main(int argc, char **argv)
{
    if(!glfwInit()){
        fprintf( stderr, "Failed to initialize GLFW\n" );
        exit(EXIT_FAILURE);
    }
}
```

7. Set up the GLFW window:

```
glfwWindowHint(GLFW_SAMPLES, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE,
    GLFW_OPENGL_CORE_PROFILE);
```

8. Create the GLFW window object and make it current for the calling thread:

```
g_window = glfwCreateWindow(WINDOWS_WIDTH,
    WINDOWS_HEIGHT, "Chapter 5 - 3D Point Cloud
    Rendering", NULL, NULL);
if(!g_window){
    fprintf( stderr, "Failed to open GLFW window. If you
        have an Intel GPU, they are not 3.3 compatible. Try
        the 2.1 version of the tutorials.\n" );
    glfwTerminate();
    exit(EXIT_FAILURE);
}
glfwMakeContextCurrent(g_window);
glfwSwapInterval(1);
```

9. Initialize the GLEW library and include support for experimental drivers:

```
glewExperimental = true;
if (glewInit() != GLEW_OK) {
    fprintf(stderr, "Final to Initialize GLEW\n");
    glfwTerminate();
    exit(EXIT_FAILURE);
}
```

10. Set up keyboard callback:

```
glfwSetInputMode(g_window, GLFW_STICKY_KEYS, GL_TRUE);
glfwSetKeyCallback(g_window, key_callback);
```

11. Set up the shader programs:

```
GLuint program_id = LoadShaders("pointcloud.vert",
    "pointcloud.frag");
```

12. Create the vertex (x, y, z) for all depth pixels:

```
GLfloat *g_vertex_buffer_data =
(GLfloat*)malloc(IMAGE_WIDTH*IMAGE_HEIGHT *
3*sizeof(GLfloat));
GLfloat *g_uv_buffer_data =
(GLfloat*)malloc(IMAGE_WIDTH*IMAGE_HEIGHT *
2*sizeof(GLfloat));
```

13. Read the raw data using the helper functions defined previously:

```
unsigned short *depth_frame =
readDepthFrame("depth_frame0.bin");
unsigned char *color_frame =
readColorFrame("color_frame0.bin");
```

14. Load the color information into a texture object:

```
GLuint texture_id = loadRGBImageToTexture(color_frame,  
    IMAGE_WIDTH, IMAGE_HEIGHT);
```

15. Create a set of vertices in a real-world space based on the depth map and also define the UV mapping for the color mapping:

```
//divided by two due to 320x240 instead of 640x480 resolution  
float cx = 320.0f/2.0f;  
float cy = 240.0f/2.0f;  
float fx = 574.0f/2.0f;  
float fy = 574.0f/2.0f;  
for(int y=0; y<IMAGE_HEIGHT; y++) {  
    for(int x=0; x<IMAGE_WIDTH; x++) {  
        int index = y*IMAGE_WIDTH+x;  
        float depth_value =  
            (float)depth_frame[index]/1000.0f; //in meter  
        int ver_index = index*3;  
        int uv_index = index*2;  
        if(depth_value != 0){  
            g_vertex_buffer_data[ver_index+0] = ((float)x-  
                cx)*depth_value/fx;  
            g_vertex_buffer_data[ver_index+1] = ((float)y-  
                cy)*depth_value/fy;  
            g_vertex_buffer_data[ver_index+2] = -depth_value;  
            g_uv_buffer_data[uv_index+0] =  
                (float)x/IMAGE_WIDTH;  
            g_uv_buffer_data[uv_index+1] =  
                (float)y/IMAGE_HEIGHT;  
        }  
    }  
}  
//Enable depth test to ensure occlusion:  
//uncommented glEnable(GL_DEPTH_TEST);
```

16. Get the location for various uniform and attribute variables:

```
GLuint matrix_id = glGetUniformLocation(program_id,  
    "MVP");  
GLuint texture_sampler_id =  
    glGetUniformLocation(program_id, "textureSampler");  
GLint attribute_vertex, attribute_uv;  
attribute_vertex = glGetAttribLocation(program_id,  
    "vertexPosition_modelspace");  
attribute_uv = glGetAttribLocation(program_id,  
    "vertexUV");
```

17. Generate the vertex array object:

```
GLuint vertex_array_id;  
 glGenVertexArrays(1, &vertex_array_id);  
 glBindVertexArray(vertex_array_id);
```

18. Initialize the vertex buffer memory:

```
GLuint vertex_buffer;  
 glGenBuffers(1, &vertex_buffer);  
 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);  
 glBufferData(GL_ARRAY_BUFFER,  
 IMAGE_WIDTH*IMAGE_HEIGHT*2* sizeof(GLfloat),  
 g_uv_buffer_data, GL_STATIC_DRAW);
```

19. Create and bind the UV buffer memory:

```
GLuint uv_buffer;  
 glGenBuffers(1, &uv_buffer);  
 glBindBuffer(GL_ARRAY_BUFFER, uv_buffer);  
 glBufferData(GL_ARRAY_BUFFER,  
 IMAGE_WIDTH*IMAGE_HEIGHT*3* sizeof(GLfloat),  
 g_vertex_buffer_data, GL_STATIC_DRAW);
```

20. Use our shader program:

```
glUseProgram(program_id);
```

21. Bind the texture in Texture Unit 0:

```
glActiveTexture(GL_TEXTURE0);  
 glBindTexture(GL_TEXTURE_2D, texture_id);  
 glUniform1i(texture_sampler_id, 0);
```

22. Set up attribute buffers for vertices and UV mapping:

```
 glEnableVertexAttribArray(attribute_vertex);  
 glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);  
 glVertexAttribPointer(attribute_vertex, 3, GL_FLOAT,  
 GL_FALSE, 0, (void*)0);  
 glEnableVertexAttribArray(attribute_uv);  
 glBindBuffer(GL_ARRAY_BUFFER, uv_buffer);  
 glVertexAttribPointer(attribute_uv, 2, GL_FLOAT,  
 GL_FALSE, 0, (void*)0);
```

23. Run the draw functions and loop:

```
do{  
 //clear the screen  
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
 glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
```

```
//compute the MVP matrix from keyboard and mouse input
computeViewProjectionMatrices(g_window);
//get the View and Model Matrix and apply to the rendering
glm::mat4 projection_matrix = getProjectionMatrix();
glm::mat4 view_matrix = getViewMatrix();
glm::mat4 model_matrix = glm::mat4(1.0);
model_matrix = glm::rotate(model_matrix,
    glm::pi<float>() * rotateY, glm::vec3(0.0f, 1.0f, 0.0f));
model_matrix = glm::rotate(model_matrix,
    glm::pi<float>() * rotateX, glm::vec3(1.0f, 0.0f, 0.0f));
glm::mat4 mvp = projection_matrix * view_matrix *
    model_matrix;
//send our transformation to the currently bound
//shader in the "MVP" uniform variable
glUniformMatrix4fv(matrix_id, 1, GL_FALSE,
    &mvp[0][0]);
glPointSize(2.0f);
//draw all points in space
glDrawArrays(GL_POINTS, 0, IMAGE_WIDTH*IMAGE_HEIGHT);
//swap buffers
glfwSwapBuffers(g_window);
glfwPollEvents();
}
// Check if the ESC key was pressed or the window was closed
while(!glfwWindowShouldClose(g_window) &&
    glfwGetKey(g_window, GLFW_KEY_ESCAPE ) !=GLFW_PRESS);
```

24. Clean up and exit the program:

```
glDisableVertexAttribArray(attribute_vertex);
glDisableVertexAttribArray(attribute_uv);
glDeleteBuffers(1, &vertex_buffer);
glDeleteBuffers(1, &uv_buffer);
glDeleteProgram(program_id);
glDeleteTextures(1, &texture_id);
glDeleteVertexArrays(1, &vertex_array_id);
glfwDestroyWindow(g_window);
glfwTerminate();
exit(EXIT_SUCCESS);
}
```

25. In `texture.cpp`, we implement the additional image-loading functions based on the previous chapter:

```
/* Handle loading images to texture memory and setting
   up the parameters */
GLuint loadRGBImageToTexture(const unsigned char *
    image_buffer, int width, int height){
    int channels;
    GLuint textureID=0;
    textureID=initializeTexture(image_buffer, width,
        height, GL_RGB);
    return textureID;
}
GLuint initializeTexture(const unsigned char *image_data,
    int width, int height, GLenum input_format){
    GLuint textureID=0;
    //for the first time we create the image,
    //create one texture element
    glGenTextures(1, &textureID);
    //bind the one element
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPixelStorei(GL_UNPACK_ALIGNMENT,1);
    /* Specify the target texture. Parameters describe the
       format and type of image data */
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
        input_format, GL_UNSIGNED_BYTE, image_data);
    /* Set the magnification method to linear, which returns
       an weighted average of 4 texture elements */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
        GL_CLAMP);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
        GL_CLAMP);
    /* Set the magnification method to linear, which //returns
       an weighted average of 4 texture elements */
    //closest to the center of the pixel
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
        GL_LINEAR);
    /* Choose the mipmap that most closely matches the size
       of the pixel being textured and use the GL_NEAREST
       criterion (texture element nearest to the center of
       the pixel) to produce texture value. */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
        GL_LINEAR_MIPMAP_LINEAR);
    glGenerateMipmap(GL_TEXTURE_2D);
    return textureID;
}
```

26. In `texture.hpp`, we simply define the function prototypes:

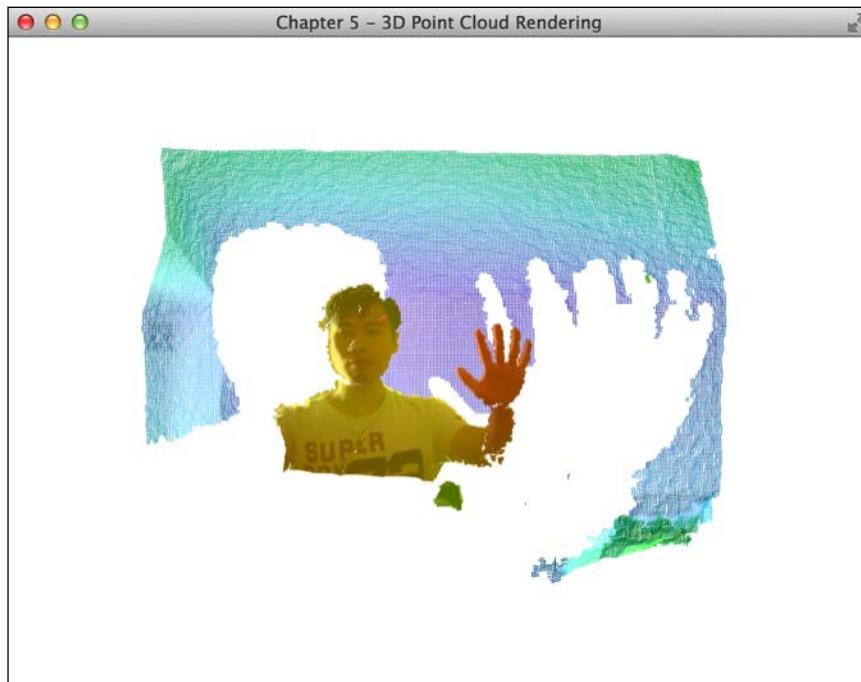
```
GLuint loadRGBImageToTexture(const unsigned char
    *image_data, int width, int height);
GLuint initializeTexture(const unsigned char *image_data,
    int width, int height, GLenum input_format = GL_RGBA);
```

27. In `control.cpp`, we modify the `computeViewProjectionMatrices` function with the following code to support additional translation controls:

```
//initial position of the camera
glm::vec3 g_position = glm::vec3( 0, 0, 3.0 );
const float speed = 3.0f; // 3 units / second
float g_initial_fov = glm::pi<float>()*0.25f;
//compute the view matrix and projection matrix based on
//user input
void computeViewProjectionMatrices(GLFWwindow* window) {
    static double last_time = glfwGetTime();
    // Compute time difference between current and last frame
    double current_time = glfwGetTime();
    float delta_time = float(current_time - last_time);
    int width, height;
    glfwGetWindowSize(window, &width, &height);
    //direction vector for movement
    glm::vec3 direction_z(0, 0, -0.5);
    glm::vec3 direction_y(0, 0.5, 0);
    glm::vec3 direction_x(0.5, 0, 0);
    //up vector
    glm::vec3 up = glm::vec3(0,-1,0);
    if (glfwGetKey( window, GLFW_KEY_UP ) == GLFW_PRESS){
        g_position += direction_y * delta_time * speed;
    }
    else if (glfwGetKey( window, GLFW_KEY_DOWN ) ==
        GLFW_PRESS){
        g_position -= direction_y * delta_time * speed;
    }
    else if (glfwGetKey( window, GLFW_KEY_RIGHT ) ==
        GLFW_PRESS){
        g_position += direction_z * delta_time * speed;
    }
    else if (glfwGetKey( window, GLFW_KEY_LEFT ) ==
        GLFW_PRESS){
        g_position -= direction_z * delta_time * speed;
    }
    else if (glfwGetKey( window, GLFW_KEY_PERIOD ) ==
        GLFW_PRESS){
        g_position -= direction_x * delta_time * speed;
    }
}
```

```
else if (glfwGetKey( window, GLFW_KEY_COMMA ) ==  
        GLFW_PRESS){  
    g_position += direction_x * delta_time * speed;  
}  
/* update projection matrix: Field of View, aspect ratio,  
   display range : 0.1 unit <-> 100 units */  
g_projection_matrix = glm::perspective(g_initial_fov,  
                                       (float)width/(float)height, 0.01f, 100.0f);  
  
// update the view matrix  
g_view_matrix = glm::lookAt(  
    g_position,           // camera position  
    g_position+direction_z, //viewing direction  
    up                  // up direction  
);  
last_time = current_time;  
}
```

Now we have created a way to visualize the depth sensor information in a 3D fly-through style; the following figure shows the rendering of the point cloud with a virtual camera at the central position of the frame:

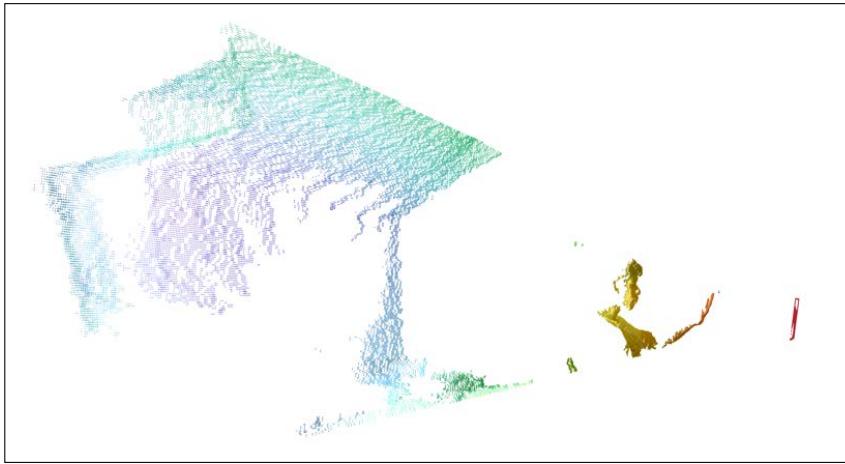


Rendering of Point Cloud Data for 3D Range-sensing Cameras

By rotating and translating the virtual camera, we can create various representations of the scene from different perspectives. With a bird's eye view or side view of the scene, we can see the contour of the face and hand more apparently from these two angles, respectively:



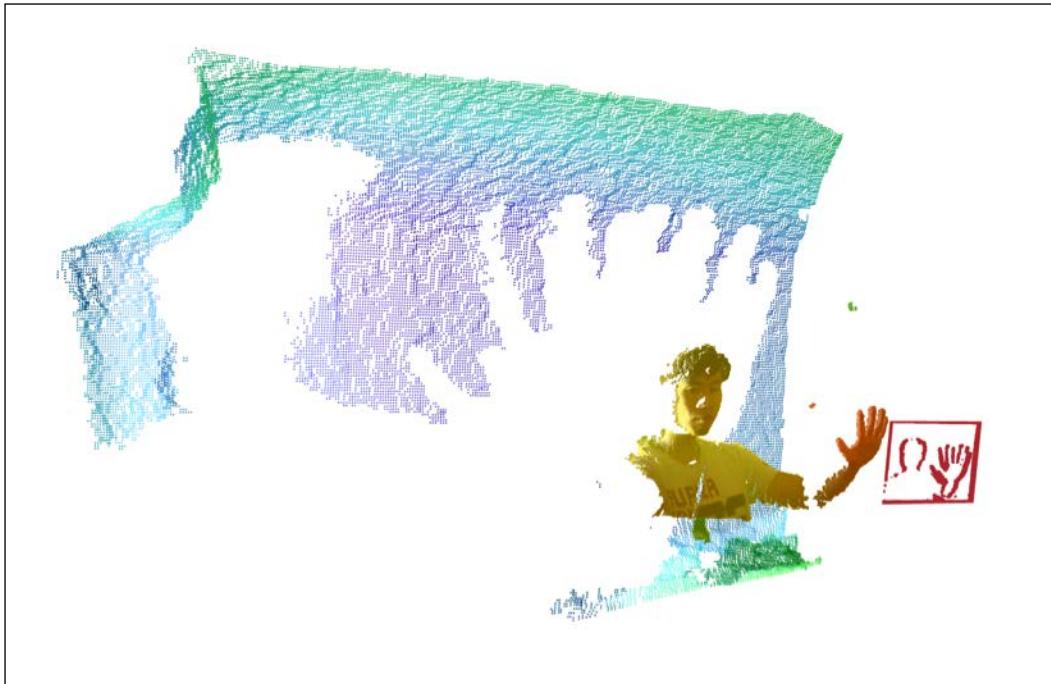
This is the side view of the same scene:



By adding an additional condition to the remapping loop, we can render the unknown regions (holes) from the scene where the depth camera fails to reconstruct due to occlusion, field of view limitation, range limitation, and/or surface properties such as reflectance:

```
if(depth_value != 0){  
    g_vertex_buffer_data[ver_index+0] = ((float)x-cx)*depth_value/fx;  
    g_vertex_buffer_data[ver_index+1] = ((float)y-cy)*depth_value/fy;  
    g_vertex_buffer_data[ver_index+2] = -depth_value;  
    g_uv_buffer_data[uv_index+0] = (float)x/IMAGE_WIDTH;  
    g_uv_buffer_data[uv_index+1] = (float)y/IMAGE_HEIGHT;  
}  
else{  
    g_vertex_buffer_data[ver_index+0] = ((float)x-cx)*0.2f/fx;  
    g_vertex_buffer_data[ver_index+1] = ((float)y-cy)*0.2f/fy;  
    g_vertex_buffer_data[ver_index+2] = 0;  
}
```

This condition allows us to segment the region and project the regions with depth values of 0 onto a plane that is 0.2 meters away from the virtual camera, as shown in the following figure:



How it works...

In this chapter, we exploited the GLSL pipeline and texture-mapping technique to create an interactive point cloud visualization tool that enables the 3D navigation of a scene captured with a 3D range-sensing camera. The shader program also combines the result with the color image to produce our desired effect. The program reads two binary images: the calibrated depth map image and the RGB color image. The color is loaded into a texture object directly using the new `loadRGBImageToTexture()` function, which converts the data from `GL_RGB` to `GL_RGBA`. Then, the depth map data is converted into point cloud data in real-world coordinates based on the intrinsic value of the cameras as well as the depth value at each pixel, as follows:

$$(X, Y, Z) = d(x - c_x)/f_x, d(y - c_y)/f_y, d/1000)$$

Here, d is the depth value in millimeter, x and y are the positions of the depth value in pixel (projective) space, c_x and c_y are the principle axes of the depth camera, f_x and f_y are the focal lengths of the camera, and (X, Y, Z) is the position of the point cloud in the real-world coordinate.

In our example, we do not require fine alignment or registration as our visualizer uses a primitive estimation of the intrinsic parameters:

$$\begin{aligned}c_x &= 320, c_y = 240, \\f_x &= 574, f_y = 574\end{aligned}$$

These numbers could be estimated with the camera calibration tools in OpenCV. The details of these tools are beyond the scope of this chapter.

For our application, we are provided a set of 3D points (x, y, z) as well as the corresponding color information (r, g, b) to compute the point cloud representation. However, the point visualization does not support dynamic lighting and other more advanced rendering techniques. To address this, we can extend the point cloud further into a mesh (that is, a set of triangles to represent surfaces), which will be discussed in the next chapter.

6

Rendering Stereoscopic 3D Models using OpenGL

In this chapter, we will cover the following topics:

- ▶ Installing the Open Asset Import Library (Assimp)
- ▶ Loading the first 3D model in the Wavefront Object (.obj) format
- ▶ Rendering 3D models with points, lines, and triangles
- ▶ Stereoscopic 3D rendering

Introduction

In this chapter, we will demonstrate how to visualize data with stunning stereoscopic 3D technology using OpenGL. Stereoscopic 3D devices are becoming increasingly popular, and the latest generation's wearable computing devices (such as the 3D vision glasses from NVIDIA, Epson, and more recently, the augmented reality 3D glasses from Meta) can now support this feature natively.

The ability to visualize data in a stereoscopic 3D environment provides a powerful and highly intuitive platform for the interactive display of data in many applications. For example, we may acquire data from the 3D scan of a model (such as in architecture, engineering, and dentistry or medicine) and would like to visualize or manipulate 3D objects in real time.

Unfortunately, OpenGL does not provide any mechanism to load, save, or manipulate 3D models. Thus, to support this, we will integrate a new library named **Open Asset Import Library (Assimp)** into our code. The source code in this chapter is built on top of the *OpenGL point cloud rendering with texture mapping and overlays* recipe in *Chapter 5, Rendering of Point Cloud Data for 3D Range-sensing Cameras*. The main dependencies include the GLFW library that requires OpenGL version 3.2 and higher. We will assume that you have all the prerequisite packages installed from earlier chapters.

Installing the Open Asset Import Library (Assimp)

Assimp is an open source library that loads and processes 3D geometric scenes from various 3D model data formats. The library provides a unified interface to load many different data formats, such as **Wavefront Object (.obj)**, **3ds Max 3DS (.3ds)**, and **Stereolithography (.stl)**. Moreover, this library is written in portable, ISO-compliant C++, and thus, it allows further customization and long-term support. Since the library is cross-platform, we can easily install it in Mac OS X, Linux, as well as Windows with the instructions given in the next section.

How to do it...

To obtain the library source files or binary library for Assimp 3.0, download them directly from Assimp's official website at <http://sourceforge.net/projects/assimp/files/assimp-3.0/>. Alternatively, for Linux and Mac OS X users, use the command-line interface to simplify the installation steps described next.

In Mac OS X, install Assimp using the MacPort's command-line interface. It automatically resolves all dependencies, so this is recommended:

```
sudo port install assimp
```

In Linux, install Assimp using the apt-get command interface:

```
sudo apt-get install libassimp-dev
```

After the installation, modify the Makefile to ensure the libraries are linked to the source files by appending the following to the LIBS variable:

```
`pkg-config --static --libs assimp`
```

and the INCLUDES path variable, respectively:

```
`pkg-config --cflags assimp`
```

The final Makefile is shown here for your reference:

```
PKG_CONFIG_PATH=/usr/local/lib/pkgconfig/
CFILES = ./common/shader.cpp ./common/controls.cpp ./common/ObjLoader.
cpp main.cpp
CFLAGS = -c
OPT = -O3
INCLUDES = -I../common -I/usr/include/SOIL -I. `pkg-
config --cflags glfw3` `pkg-config --cflags assimp`
LIBS = -lm -L/usr/local/lib -lGLEW `pkg-config --static --libs glfw3` `pkg-
config --static --libs assimp`
CC = g++
OBJECTS=$(CFILES:.cpp=.o)
EXECUTABLE=main
all: $(CFILES) $(EXECUTABLE)
$(EXECUTABLE): $(OBJECTS)
$(CC) $(OPT) $(INCLUDES) $(OBJECTS) -o $@ $(LIBS)
.cpp.o:
$(CC) $(OPT) $(CFLAGS) $(INCLUDES) $< -o $@
clean:
rm -v -f *~ ./common/*.o *.o $(EXECUTABLE)
```

To install Assimp in Windows, first, download the binary library from this link:
<http://sourceforge.net/projects/assimp/files/assimp-3.0/assimp--3.0.1270-full.zip/download>.

Then, we configure the environment with the following steps:

1. Unpack assimp--3.0.1270-full.zip and save it in C:/Program Files (x86)/.
2. Add the DLL path, C:/Program Files (x86)/assimp--3.0.1270-sdk/bin/assimp_release-dll_win32, to the PATH environment variable.
3. Include the CMakeLists.txt file to the project:

```
cmake_minimum_required (VERSION 2.8)
set(CMAKE_CONFIGURATION_TYPES Debug Release)
set(PROGRAM_PATH "C:/Program Files (x86)\")
set(OpenCV_DIR ${PROGRAM_PATH}/opencv/build)
project (code)
#modify these path based on your configuration
#OpenCV
find_package(OpenCV REQUIRED )
INCLUDE_DIRECTORIES(${OpenCV_INCLUDE_DIRS})
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glm)
```

```
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glew-1.10.0/include)
LINK_DIRECTORIES(${PROGRAM_PATH}/glew-
1.10.0/lib/Release/Win32)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/glfw-3.0.4/include)
LINK_DIRECTORIES(${PROGRAM_PATH}/glfw-3.0.4/lib)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/Simple\ OpenGL\ Image\
Library/src)
INCLUDE_DIRECTORIES(${PROGRAM_PATH}/assimp--3.0.1270-
sdk/include/assimp)
LINK_DIRECTORIES(${PROGRAM_PATH}/assimp--3.0.1270-
sdk/lib/assimp_release-dll_win32)
add_subdirectory(../common common)
add_executable(main main.cpp)
target_link_libraries(main LINK_PUBLIC shader controls
texture glew32s glfw3 opengl32 assimp ObjLoader)
```

Finally, generate the build files with the same steps as described in *Chapter 4, Rendering 2D Images and Videos with Texture Mapping* and *Chapter 5, Rendering of Point Cloud Data for 3D Range-sensing Cameras*.

See also

In addition to importing 3D model objects, Assimp also supports the exporting of 3D models in .obj, .stl, and .ply formats. By combining this library with the OpenGL graphics rendering engine, we have created a simple yet powerful mechanism to visualize and exchange 3D models collaboratively or remotely. The Assimp library can also handle some postprocessing tasks of 3D scenes after importing the model (for example, splitting large meshes to overcome certain GPU limitations on vertex count). These additional features are documented on the official website and may be of interest to advanced users (http://assimp.sourceforge.net/lib_html/index.html).

Loading the first 3D model in the Wavefront Object (.obj) format

Now, we are ready to integrate a 3D object loader into our code. The first step is to create an empty class called `ObjLoader` along with the source (.cpp) and header (.h) files. This class handles all the functions related to 3D object loading, parsing, and drawing using the OpenGL and Assimp libraries. The headers of the class will include the Assimp core functions for the handling of the data structures and all I/O mechanisms of the 3D data format:

```
#include <cimport.h>
#include <scene.h>
#include <postprocess.h>
```

In the `ObjLoader.h` file, we provide interfaces for the main program to create, destroy, load, and display the 3D data. In the `ObjLoader.cpp` file, we implement a set of functions to parse the scene (a hierarchical representation of the 3D objects in terms of meshes and faces) using the built-in functions from Assimp.

The Assimp library can support various 3D model data formats; however, in our example, we will focus on the Wavefront Object (`.obj`) format due to its simplicity. The `.obj` file is a simple geometric definition file that was first developed by Wavefront Technologies. The file contains the core elements of graphics, such as vertex, vertex position, normal face and so on, and is stored in a simple text format. Since the files are stored in ASCII text, we can easily open and examine the files without any parsers. For example, the following is the `.obj` file of a front-facing square:

```
# This is a comment.  
# Front facing square.  
# vertices [x, y, z]  
v 0 0 0    # Bottom left.  
v 1 0 0    # Bottom right.  
v 1 1 0    # Top      right.  
v 0 1 0    # Top      left.  
# List of faces:  
f 1 2 3 4    # Square.
```

As we can see from the preceding example, the representation is quite simple and intuitive for beginners. The vertices can be read and extracted one line at a time, and then they can be modified.

In the next section, we will show the full implementation, which allows users to load the `.obj` file, store the scene in a vertex buffer object, and display the scene.

How to do it...

First, we create the `ObjLoader.h` file in the common folder and append the class function definitions and variables that will be used in our implementation:

```
#ifndef OBJLOADER_H_  
#define OBJLOADER_H_  
/* Assimp include files. These three are usually needed. */  
#include <cimport.h>  
#include <scene.h>  
#include <postprocess.h>  
#include <common.h>  
#define aisgl_min(x,y) (x<y?x:y)  
#define aisgl_max(x,y) (y>x?y:x)  
class ObjLoader {
```

```
public:  
ObjLoader();  
virtual ~ObjLoader();  
int loadAsset(const char* path);  
void setScale(float scale);  
unsigned int getNumVertices();  
void draw(const GLenum draw_mode);  
void loadVertices(GLfloat *g_vertex_buffer_data);  
  
private:  
//helper functions and variables  
const struct aiScene* scene;  
GLuint scene_list;  
aiVector3D scene_min, scene_max, scene_center;  
float g_scale;  
unsigned int num_vertices;  
unsigned int recursiveDrawing(const struct aiNode* nd,  
    unsigned int v_count, const GLenum);  
unsigned int recursiveVertexLoading(const struct aiNode *nd,  
    GLfloat *g_vertex_buffer_data, unsigned int v_counter);  
unsigned int recursiveGetNumVertices(const struct aiNode *nd);  
void get_bounding_box (aiVector3D* min, aiVector3D* max);  
void get_bounding_box_for_node (const struct aiNode* nd,  
    aiVector3D* min, aiVector3D* max, aiMatrix4x4* trafo);  
};  
#endif
```

The names of classes from the Assimp library are preceded by the prefix ai - (for example, aiScene and aiVector3D). The ObjLoader file provides ways to dynamically load and draw the object loaded into the memory. It also handles simple dynamic scaling so that the object will fit on the screen.

In the source file, ObjLoader.cpp, we start by adding the constructor for the class:

```
#include <ObjLoader.h>  
ObjLoader::ObjLoader() {  
    g_scale=1.0f;  
    scene = NULL; //empty scene  
    scene_list = 0;  
    num_vertices = 0;  
}
```

Then, we implement the file-loading mechanism with the `aiImportFile` function. The scene is processed to extract the bounding box size for proper scaling to fit the screen. The number of vertices of the scene is then used to allow dynamic vertex buffer creation in later steps:

```
int ObjLoader::loadAsset(const char *path) {
    scene = aiImportFile(path,
        aiProcessPreset_TargetRealtime_MaxQuality);
    if (scene) {
        get_bounding_box(&scene_min,&scene_max);
        scene_center.x = (scene_min.x + scene_max.x) / 2.0f;
        scene_center.y = (scene_min.y + scene_max.y) / 2.0f;
        scene_center.z = (scene_min.z + scene_max.z) / 2.0f;
        printf("Loaded file %s\n", path);
        g_scale = 4.0/(scene_max.x-scene_min.x);

        printf("Scaling: %lf", g_scale);
        num_vertices = recursiveGetNumVertices(scene->mRootNode);
        printf("This Scene has %d vertices.\n", num_vertices);
        return 0;
    }
    return 1;
}
```

To extract the total number of vertices required to draw the scene, we recursively walk through every node in the tree hierarchy. The implementation requires a simple recursive function that returns the number of vertices in each node, and then the total is calculated based on the summation of all nodes upon the return of the function:

```
unsigned int ObjLoader::recursiveGetNumVertices(const struct
    aiNode *nd) {
    unsigned int counter=0;
    unsigned int i;
    unsigned int n = 0, t;
    // draw all meshes assigned to this node
    for ( ; n < nd->mNumMeshes; ++n) {
        const struct aiMesh* mesh = scene->mMeshes[nd->mMeshes[n]];
        for (t = 0; t < mesh->mNumFaces; ++t) {
            const struct aiFace* face = &mesh->mFaces[t];
            counter+=3*face->mNumIndices;
        }
        printf("recursiveGetNumVertices: mNumFaces      %d\n",
            mesh->mNumFaces);
    }
}
```

```

//traverse all children nodes
for (n = 0; n < nd->mNumChildren; ++n) {
    counter+=recursiveGetNumVertices(nd-> mChildren[n]);
}
printf("recursiveGetNumVertices: counter %d\n", counter);
return counter;
}

```

Similarly, to calculate the size of the bounding box (that is, the minimum volume that is required to contain the scene), we recursively examine each node and extract the points that are farthest away from the center of the object:

```

void ObjLoader::get_bounding_box (aiVector3D* min,
aiVector3D* max)
{
    aiMatrix4x4 trafo;
    aiIdentityMatrix4(&trafo);
    min->x = min->y = min->z =  1e10f;
    max->x = max->y = max->z = -1e10f;
    get_bounding_box_for_node(scene-> mRootNode,min,max,&trafo);
}
void ObjLoader::get_bounding_box_for_node (const struct aiNode*
nd, aiVector3D* min, aiVector3D* max, aiMatrix4x4* trafo)
{
    aiMatrix4x4 prev;
    unsigned int n = 0, t;
    prev = *trafo;
    aiMultiplyMatrix4(trafo,&nd->mTransformation);
    for ( ; n < nd->mNumMeshes; ++n) {
        const struct aiMesh* mesh = scene-> mMeshes[nd->mMeshes[n]];
        for (t = 0; t < mesh->mNumVertices; ++t) {
            aiVector3D tmp = mesh->mVertices[t];
            aiTransformVecByMatrix4 (&tmp,trafo);
            min->x = aisgl_min(min->x,tmp.x);
            min->y = aisgl_min(min->y,tmp.y);
            min->z = aisgl_min(min->z,tmp.z);
            max->x = aisgl_max(max->x,tmp.x);
            max->y = aisgl_max(max->y,tmp.y);
            max->z = aisgl_max(max->z,tmp.z);
        }
    }
}

```

```
for (n = 0; n < nd->mNumChildren; ++n) {
    get_bounding_box_for_node(nd-> mChildren[n],min,max,trafo);
}
*trafo = prev;
}
```

The resulting bounding box allows us to calculate the scaling factor and recenter the object coordinate to fit within the viewable screen.

In the `main.cpp` file, we integrate the code by first inserting the header file:

```
#include <ObjLoader.h>
```

Then, we create the `ObjLoader` object and load the model with the given filename in the main function:

```
ObjLoader *obj_loader = new ObjLoader();
int result = 0;
if(argc > 1){
    result = obj_loader->loadAsset(argv[1]);
}
else{
    result = obj_loader-> loadAsset("dragon.obj");
}
if(result){
    fprintf(stderr, "Final to Load the 3D file\n");
    glfwTerminate();
    exit(EXIT_FAILURE);
}
```

The `ObjLoader` contains an algorithm that recursively examines each mesh and computes the bounding box and the number of vertices in the scene. Then, we dynamically allocate the vertex buffer based on the number of vertices and load the vertices into the buffer:

```
GLfloat *g_vertex_buffer_data = (GLfloat*)
malloc (obj_loader->getNumVertices()*sizeof(GLfloat));
//load the scene data to the vertex buffer
obj_loader->loadVertices(g_vertex_buffer_data);
```

Now, we have all the necessary vertex information for display with our custom shader program written in OpenGL.

How it works...

Assimp provides the mechanism to load and parse the 3D data format efficiently. The key feature we utilized is the hierarchical way to import 3D objects, which allows us to unify our rendering pipeline regardless of the 3D format. The `aiImportFile` function reads the given file and returns its content in the `aiScene` structure. The second parameter of this function specifies the optional postprocessing steps to be executed after a successful import. The `aiProcessPreset_TargetRealtime_MaxQuality` flag is a predefined variable, which combines the following set of parameters:

```
( \
    aiProcessPreset_TargetRealtime_Quality | \
    aiProcess_FindInstances | \
    aiProcess_ValidateDataStructure | \
    aiProcess_OptimizeMeshes | \
    aiProcess_Debone | \
0 )
```

These postprocessing options are described in further detail at http://assimp.sourceforge.net/lib_html/postprocess_8h.html#a64795260b95f5a4b3f3dc1be4f52e410. Advanced users can look into each option and understand whether these functions need to be enabled or disabled based on the content.

At this point, we have a simple mechanism to load graphics into the Assimp `aiScene` object, present the bounding box size, as well as extract the number of vertices required to render the scene. Next, we will create a simple shader program as well as various drawing functions to visualize the content with different styles. In short, by integrating this with the OpenGL graphics rendering engine, we now have a flexible way to visualize 3D models using the various tools we developed in the previous chapters.

Rendering 3D models with points, lines, and triangles

The next step after importing the 3D model is to display the content on the screen using an intuitive and aesthetically pleasing way. Many complex scenes consist of multiple surfaces (meshes) and many vertices. In the previous chapter, we implemented a simple shader program to visualize the point cloud at various depth values based on a heat map. In this section, we will utilize very simple primitives (points, lines, and triangles) with transparency to create skeleton-like rendering effects.

How to do it...

We will continue the implementation of the `ObjLoader` class to support loading vertices and draw the graphics for each mesh in the scene.

In the source file of `ObjLoader.cpp`, we add a recursive function to extract all vertices from the scene and store them in a single vertex buffer array. This allows us to reduce the number of vertex buffers to be managed, thus reducing the complexity of the code:

```
void ObjLoader::loadVertices(GLfloat *g_vertex_buffer_data)
{
    recursiveVertexLoading(scene->mRootNode, g_vertex_buffer_data,
                           0);
}
unsigned int ObjLoader::recursiveVertexLoading (const struct
                                              aiNode *nd, GLfloat *g_vertex_buffer_data, unsigned int
                                              v_counter)
{
    unsigned int i;
    unsigned int n = 0, t;
    /* save all data to the vertex array, perform offset and scaling
       to reduce the computation */
    for (; n < nd->mNumMeshes; ++n) {
        const struct aiMesh* mesh = scene->mMeshes[nd->mMeshes[n]];
        for (t = 0; t < mesh->mNumFaces; ++t) {
            const struct aiFace* face = &mesh->mFaces[t];
            for(i = 0; i < face->mNumIndices; i++) {
                int index = face->mIndices[i];
                g_vertex_buffer_data[v_counter] =
                    (mesh->mVertices[index].x-scene_center.x)*g_scale;
                g_vertex_buffer_data[v_counter+1] =
                    (mesh->mVertices[index].y-scene_center.y)*g_scale;
                g_vertex_buffer_data[v_counter+2] =
                    (mesh->mVertices[index].z-scene_center.z)*g_scale;
                v_counter+=3;
            }
        }
    }
    //traverse all children nodes
    for (n = 0; n < nd->mNumChildren; ++n) {
        v_counter = recursiveVertexLoading(nd->mChildren[n],
                                           g_vertex_buffer_data, v_counter);
    }
    return v_counter;
}
```

To draw the graphics, we traverse the `aiScene` object from the root node and draw the meshes one piece at a time:

```

void ObjLoader::draw(const GLenum draw_mode) {
    recursiveDrawing(scene->mRootNode, 0, draw_mode);
}
unsigned int ObjLoader::recursiveDrawing(const struct aiNode* nd,
    unsigned int v_counter, const GLenum draw_mode) {
    /* break up the drawing, and shift the pointer to draw different
       parts of the scene */
    unsigned int i;
    unsigned int n = 0, t;
    unsigned int total_count = v_counter;
    // draw all meshes assigned to this node
    for (; n < nd->mNumMeshes; ++n) {
        unsigned int count=0;
        const struct aiMesh* mesh = scene-> mMeshes[nd->mMeshes[n]];
        for (t = 0; t < mesh->mNumFaces; ++t) {
            const struct aiFace* face = &mesh-> mFaces[t];
            count+=3*face->mNumIndices;
        }
        glDrawArrays(draw_mode, total_count, count);
        total_count+=count;
    }
    v_counter = total_count;
    // draw all children nodes recursively
    for (n = 0; n < nd->mNumChildren; ++n) {
        v_counter = recursiveDrawing(nd-> mChildren[n], v_counter,
            draw_mode);
    }
    return v_counter;
}

```

In the vertex shader, `pointcloud.vert`, we compute the color of vertices based on their positions in space. The remapping algorithm creates a heat map representation of the object in space, and it serves as an important depth cue for the human eye (depth perception):

```

#version 150 core
// Input
in vec3 vertexPosition_modelspace;
// Output
out vec4 color_based_on_position;
// Uniform/constant variable.
uniform mat4 MVP;
//heat map generator

```

```
vec4 heatMap(float v, float vmin, float vmax) {
    float dv;
    float r=1.0f, g=1.0f, b=1.0f;
    if (v < vmin)
        v = vmin;
    if (v > vmax)
        v = vmax;
    dv = vmax - vmin;
    if (v < (vmin + 0.25f * dv)) {
        r = 0.0f;
        g = 4.0f * (v - vmin) / dv;
    } else if (v < (vmin + 0.5f * dv)) {
        r = 0.0f;
        b = 1.0f + 4.0f * (vmin + 0.25f * dv - v) / dv;
    } else if (v < (vmin + 0.75f * dv)) {
        r = 4.0f * (v - vmin - 0.5f * dv) / dv;
        b = 0.0f;
    } else {
        g = 1.0f + 4.0f * (vmin + 0.75f * dv - v) / dv;
        b = 0.0f;
    }
    //with 0.2 transparency - can be dynamic if we pass in variables
    return vec4(r, g, b, 0.2f);
}

void main () {
    // Output position of the vertex, in clip space : MVP * position
    gl_Position = MVP * vec4(vertexPosition_modelspace, 1.0f);
    // remapping the color based on the depth (z) value.
    color_based_on_position = heatMap(vertexPosition_modelspace.z,
        -1.0f, 1.0f);
}
```

The vertex shader passes the heat-mapped color information along to the fragment shader through the `color_based_on_position` variable. Then, the final color is returned through the fragment shader (`pointcloud.frag`) directly without further processing. The implementation of such a simple pipeline is shown as follows:

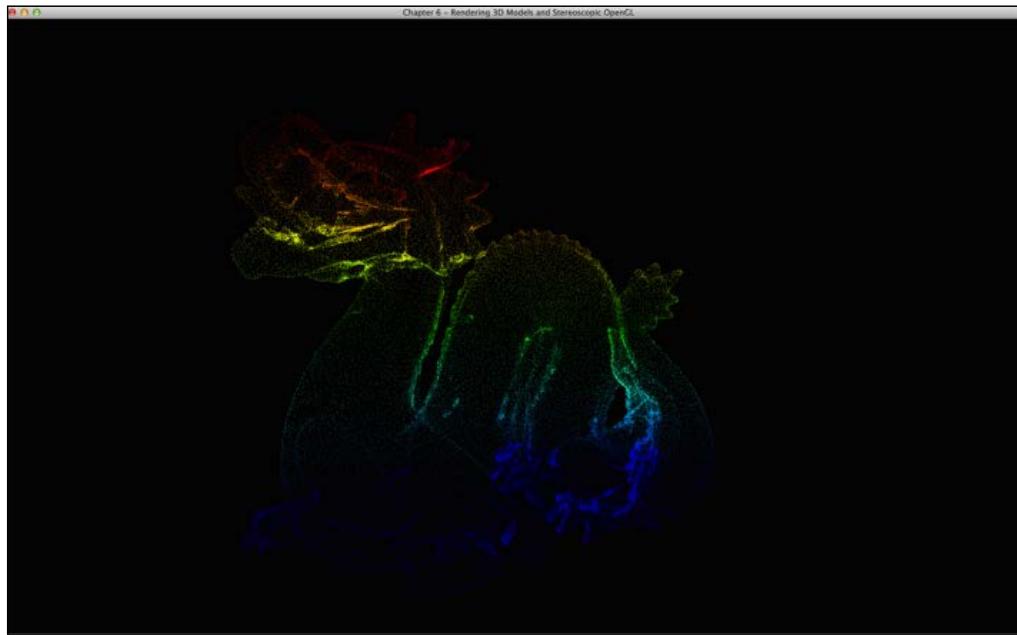
```
#version 150 core
out vec4 color;
in vec4 color_based_on_position;
void main(){
    color = color_based_on_position;
}
```

Finally, we draw the scene with various styles: lines, points, and triangles with transparency. The following is the code snippet inside the drawing loop:

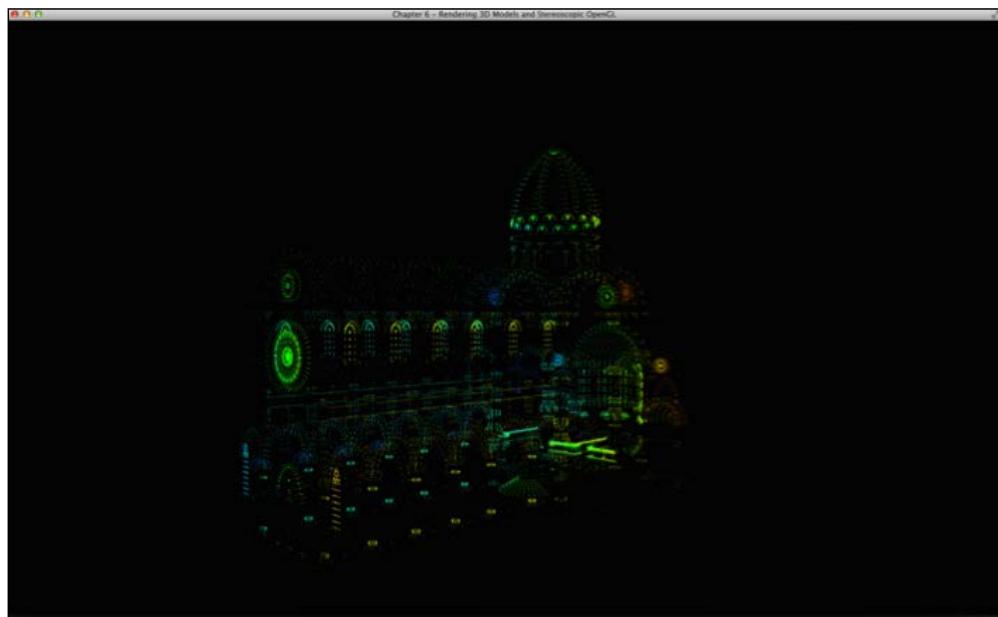
```
//draw the left eye (but full screen)
glViewport(0, 0, width, height);
//compute the MVP matrix from the IOD and virtual image plane distance
computeStereoViewProjectionMatrices(g_window, IOD, depthZ, true);
//get the View and Model Matrix and apply to the rendering
glm::mat4 projection_matrix = getProjectionMatrix();
glm::mat4 view_matrix = getViewMatrix();
glm::mat4 model_matrix = glm::mat4(1.0);
model_matrix = glm::translate(model_matrix, glm::vec3(0.0f, 0.0f,
-depthZ));
model_matrix = glm::rotate(model_matrix,
glm::pi<float>()*rotateY, glm::vec3(0.0f, 1.0f, 0.0f));
model_matrix = glm::rotate(model_matrix,
glm::pi<float>()*rotateX, glm::vec3(1.0f, 0.0f, 0.0f));
glm::mat4 mvp = projection_matrix * view_matrix * model_matrix;
//send our transformation to the currently bound shader,
//in the "MVP" uniform variable
glUniformMatrix4fv(matrix_id, 1, GL_FALSE, &mvp[0][0]);
/* render scene with different modes that can be enabled separately
to get different effects */
obj_loader->draw(GL_TRIANGLES);
if(drawPoints)
    obj_loader->draw(GL_POINTS);
if(drawLines)
    obj_loader->draw(GL_LINES);
```

The series of screenshots that follow illustrate the aesthetically pleasing results we can achieve with our custom shader. The color mapping based on the depth position using the heat map shader provides a strong depth perception that helps us understand the 3D structure of the objects more easily. Furthermore, we can enable and disable various rendering options separately to achieve various effects. For example, the same object can be rendered with different styles: points, lines, and triangles (surfaces) with transparency.

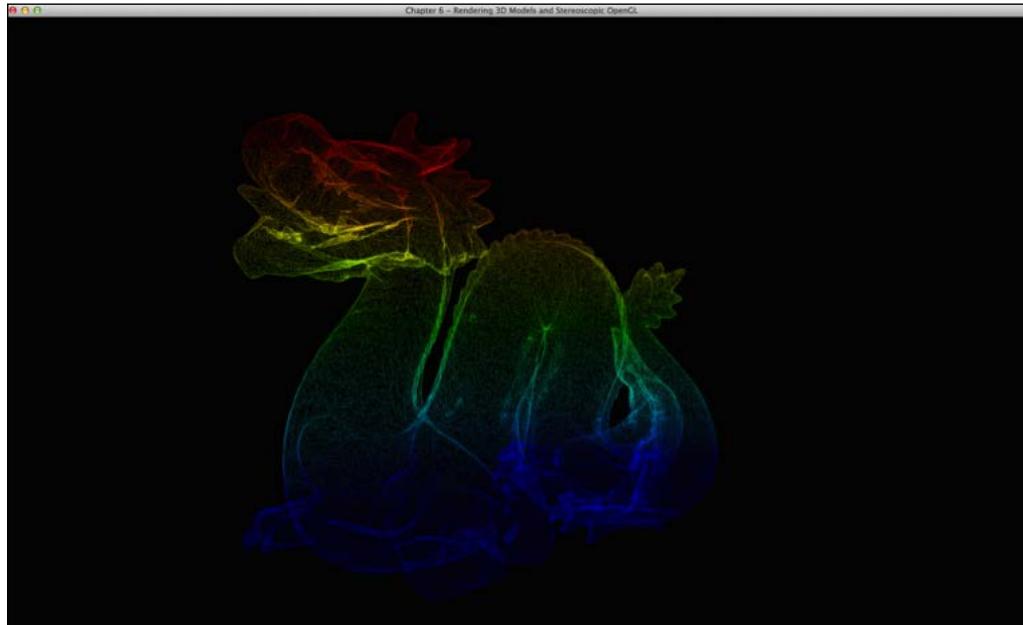
To demonstrate the effects, we will first render two objects with points only. The first example is a dragon model:



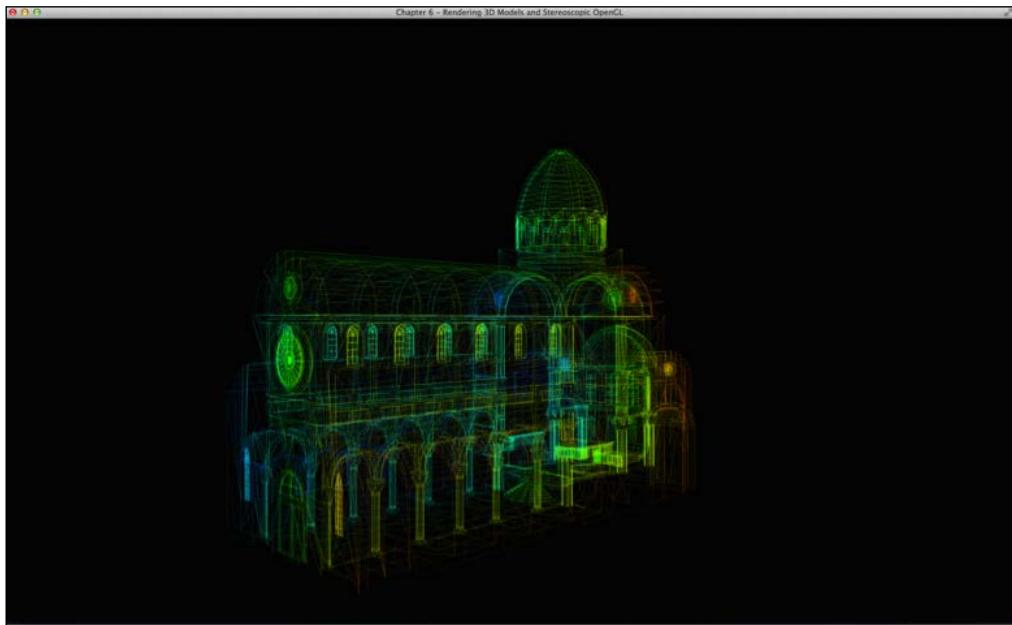
The second example is an architectural model:



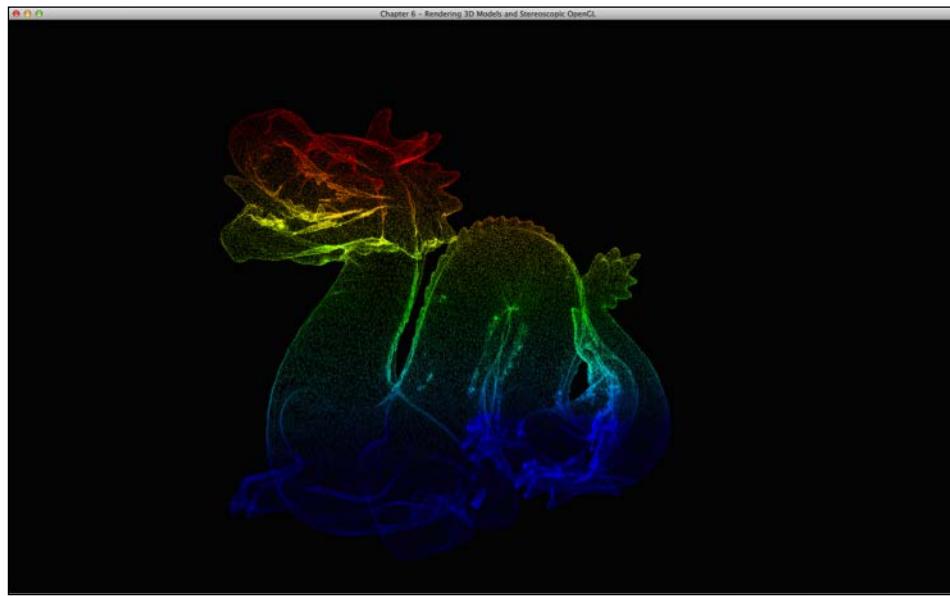
The point-based rendering style is great for visualizing a large dataset with unknown relations or distribution. Next, we will render the same objects with lines only:



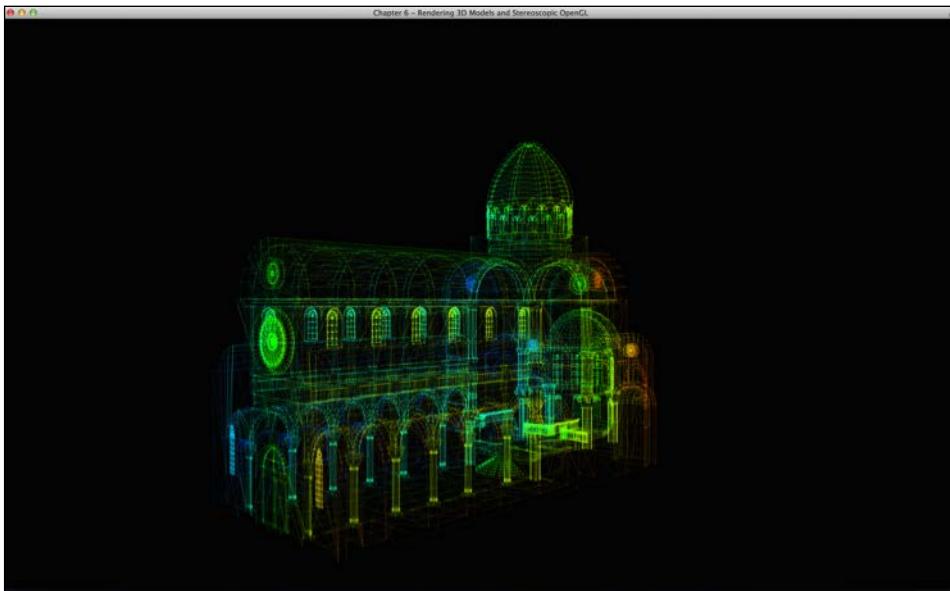
Here's the architectural model rendered with lines only:



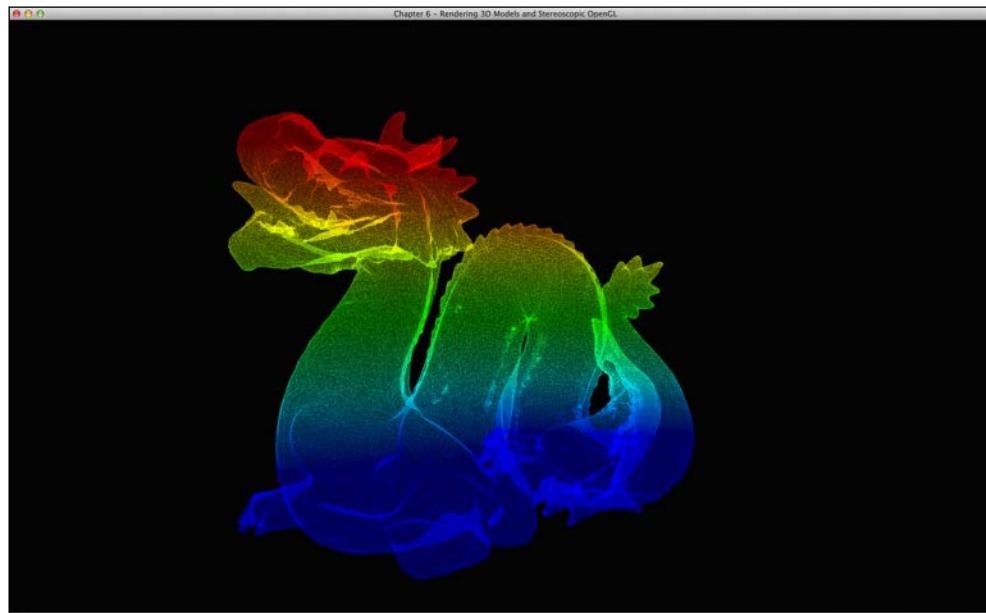
With the lines, now we can see the structure of the object more easily. This rendering technique is great for simple structures, such as architectural models and other well-defined models. In addition, we can render the scene with both points and lines enabled, as shown here:



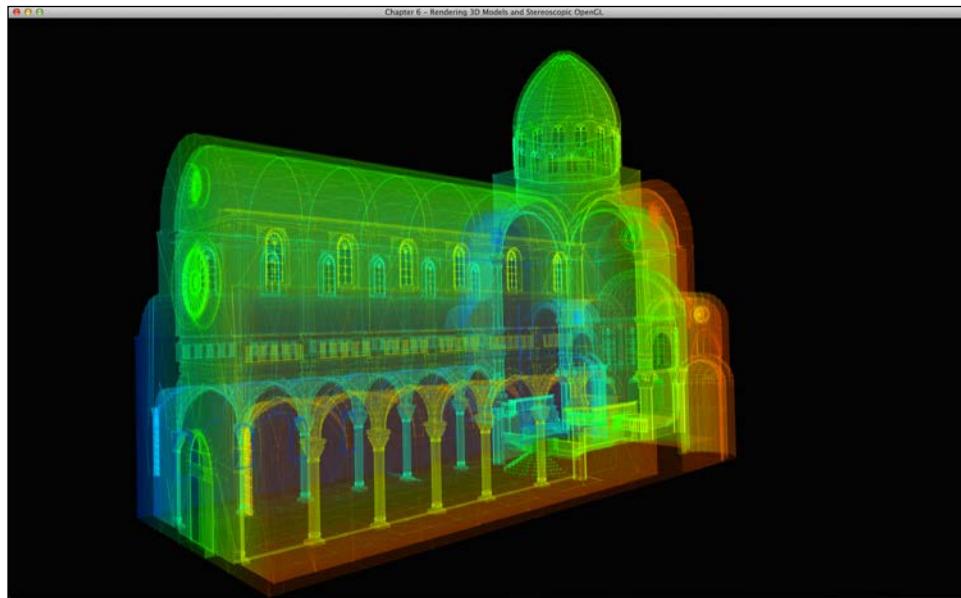
Here's the architectural model rendered with points and lines enabled:



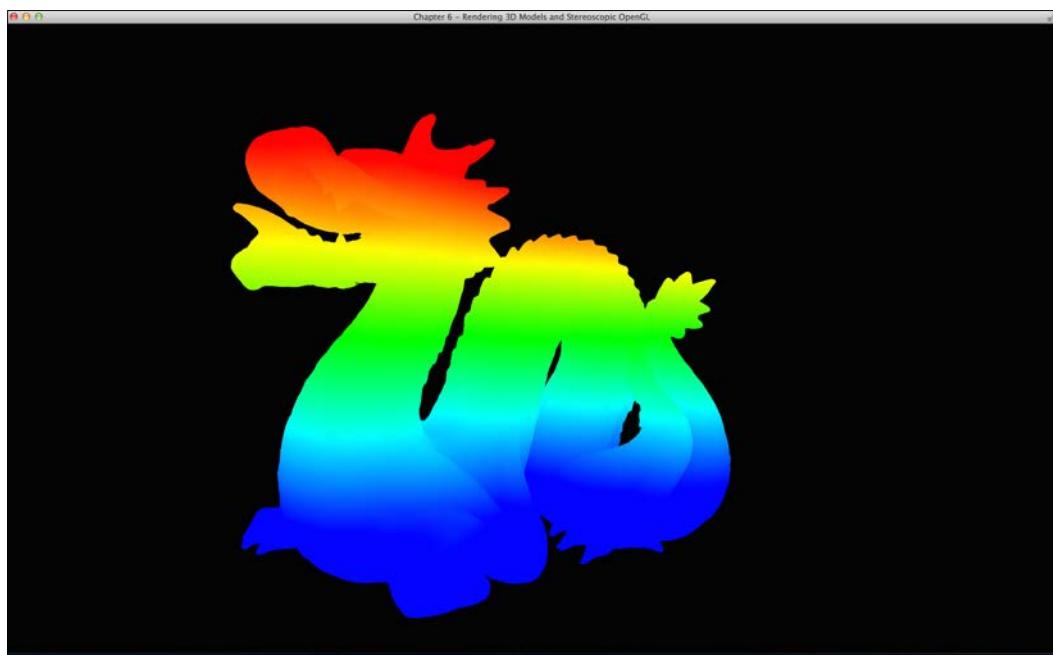
The combination of both points and lines provides additional visual cue to the structure of the object (that is, emphasis on the intersection points). Finally, we render the scene with all options enabled: points, lines, and triangles (surfaces) with transparency:



Here's the architectural model rendered using points, lines and triangles with transparency:



The final combination with all the options enabled provides an even more intuitive visualization of the volume of the object as well as the overall 3D structure. Alternatively, we can also enable the depth test and render the solid model with no transparency:



Instructions on how to enable/disable these options at runtime are documented in the source code.

How it works...

By combining the Assimp library and OpenGL, we can now dynamically load 3D models on the screen and create visually appealing 3D effects through an OpenGL-based interactive visualization tool.

In `ObjLoader.cpp`, the `loadVertices` function converts the scene into a single vertex buffer array to reduce the complexity of memory management. In particular, this approach reduces the number of OpenGL memory copies and the number of memory buffers on the rendering side (that is, `glBufferData` and `glGenBuffers`). In addition, the loading function handles the scaling and centering of vertices based on the bounding box. This step is critical as most 3D formats do not normalize their coordinate system.

Next, the `draw` function in `ObjLoader.cpp` traverses the `aiScene` object and draws each part of the scene with the vertex buffer. In the case of point-based rendering, we can skip this step and directly draw the entire array using `glDrawArray` because there is no dependency among the neighboring vertices.

The vertex shader (`pointcloud.vert`) contains the implementation of the heat map color generator. The `heatmap` function takes in three parameters: the input value (that is, the depth or z value), the minimum value, and maximum value. It returns the heat map color representation in the RGBA format.

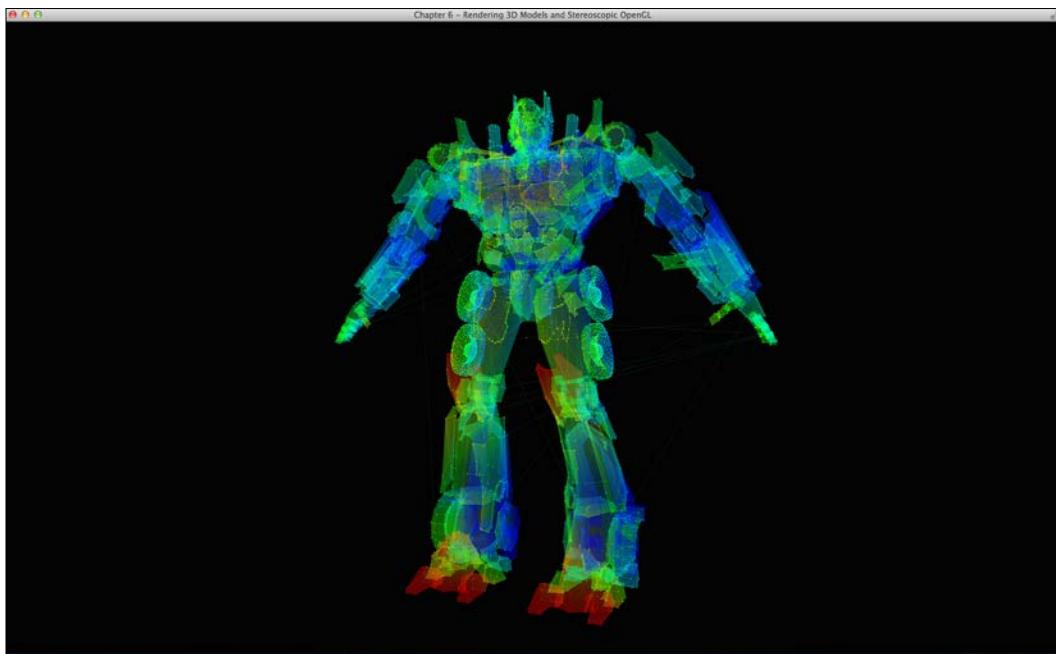
Inside the drawing loop, the `computeStereoViewProjectionMatrices` function constructs the view and projection matrices. The details are explained in the next section.

Finally, we can mix and match various rendering techniques; for example, by enabling both points and lines only for skeleton-based rendering. Various depth visual cues, such as occlusion and motion parallax, can be easily added by supporting rotation or translation of the object. To further improve the result, other rendering techniques such as lighting or shading can be added based on the application requirements.

See also

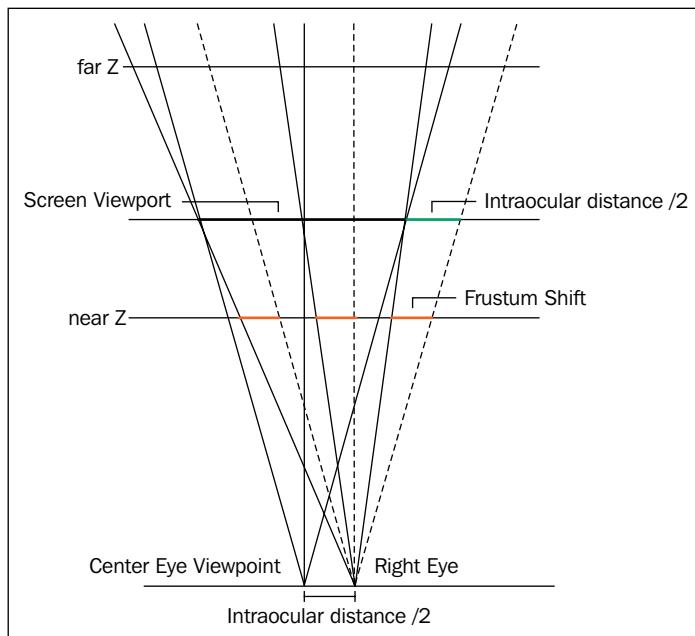
The Assimp library also supports many file formats in addition to `.obj` files. For example, we can load `.stl` files into our system without changing the source code at all.

To download more 3D models, visit various 3D model-sharing websites such as *Makerbot Thingiverse* (<http://www.thingiverse.com/>) or *Turbosquid* (<http://www.turbosquid.com/>):



Stereoscopic 3D rendering

3D television and 3D glasses are becoming much more prevalent with the latest trends in consumer electronics and technological advances in wearable computing. In the market, there are currently many hardware options that allow us to visualize information with stereoscopic 3D technology. One common format is side-by-side 3D, which is supported by many 3D glasses as each eye sees an image of the same scene from a different perspective. In OpenGL, creating side-by-side 3D rendering requires asymmetric adjustment as well as viewport adjustment (that is, the area to be rendered) – asymmetric frustum parallel projection or equivalently to lens-shift in photography. This technique introduces no vertical parallax and widely adopted in the stereoscopic rendering. To illustrate this concept, the following diagram shows the geometry of the scene that a user sees from the right eye:



The **intraocular distance (IOD)** is the distance between two eyes. As we can see from the diagram, the **Frustum Shift** represents the amount of skew/shift for asymmetric frustum adjustment. Similarly, for the left eye image, we perform the transformation with a mirrored setting. The implementation of this setup is described in the next section.

How to do it...

The following code illustrates the steps to construct the projection and view matrices for stereoscopic 3D visualization. The code uses the intraocular distance, the distance of the image plane, and the distance of the near clipping plane to compute the appropriate frustum shifts value. In the source file, `common/controls.cpp`, we add the implementation for the stereo 3D matrix setup:

```
void computeStereoViewProjectionMatrices(GLFWwindow* window,
    float IOD, float depthZ, bool left_eye){
    int width, height;
    glfwGetWindowSize(window, &width, &height);
    //up vector
    glm::vec3 up = glm::vec3(0, -1, 0);
    glm::vec3 direction_z(0, 0, -1);
    //mirror the parameters with the right eye
    float left_right_direction = -1.0f;
    if(left_eye)
        left_right_direction = 1.0f;
    float aspect_ratio = (float)width/(float)height;
    float nearZ = 1.0f;
    float farZ = 100.0f;
    double frustumshift = (IOD/2)*nearZ/depthZ;
    float top = tan(g_initial_fov/2)*nearZ;
    float right =
        aspect_ratio*top+frustumshift*left_right_direction;
    //half screen
    float left =
        -aspect_ratio*top+frustumshift*left_right_direction;
    float bottom = -top;
    g_projection_matrix = glm::frustum(left, right, bottom, top,
        nearZ, farZ);
    // update the view matrix
    g_view_matrix =
        glm::lookAt(
            g_position-direction_z+
            glm::vec3(left_right_direction*IOD/2, 0, 0),
            //eye position
            g_position+
            glm::vec3(left_right_direction*IOD/2, 0, 0),
            //centre position
            up //up direction
        );
}
```

In the rendering loop in `main.cpp`, we define the viewports for each eye (*left* and *right*) and set up the projection and view matrices accordingly. For each eye, we translate our camera position by half of the intraocular distance, as illustrated in the previous figure:

```
if(stereo) {
    //draw the LEFT eye, left half of the screen
    glViewport(0, 0, width/2, height);
    //computes the MVP matrix from the IOD and virtual image plane distance
    computeStereoViewProjectionMatrices(g_window, IOD, depthZ, true);
    //gets the View and Model Matrix and apply to the rendering
    glm::mat4 projection_matrix = getProjectionMatrix();
    glm::mat4 view_matrix = getViewMatrix();
    glm::mat4 model_matrix = glm::mat4(1.0);
    model_matrix = glm::translate(model_matrix, glm::vec3(0.0f,
        0.0f, -depthZ));
    model_matrix = glm::rotate(model_matrix, glm::pi<float>() *
        rotateY, glm::vec3(0.0f, 1.0f, 0.0f));
    model_matrix = glm::rotate(model_matrix, glm::pi<float>() *
        rotateX, glm::vec3(1.0f, 0.0f, 0.0f));
    glm::mat4 mvp = projection_matrix * view_matrix * model_matrix;
    //sends our transformation to the currently bound shader,
    //in the "MVP" uniform variable
    glUniformMatrix4fv(matrix_id, 1, GL_FALSE, &mvp[0][0]);
    //render scene, with different drawing modes

    if(drawTriangles)
        obj_loader->draw(GL_TRIANGLES);

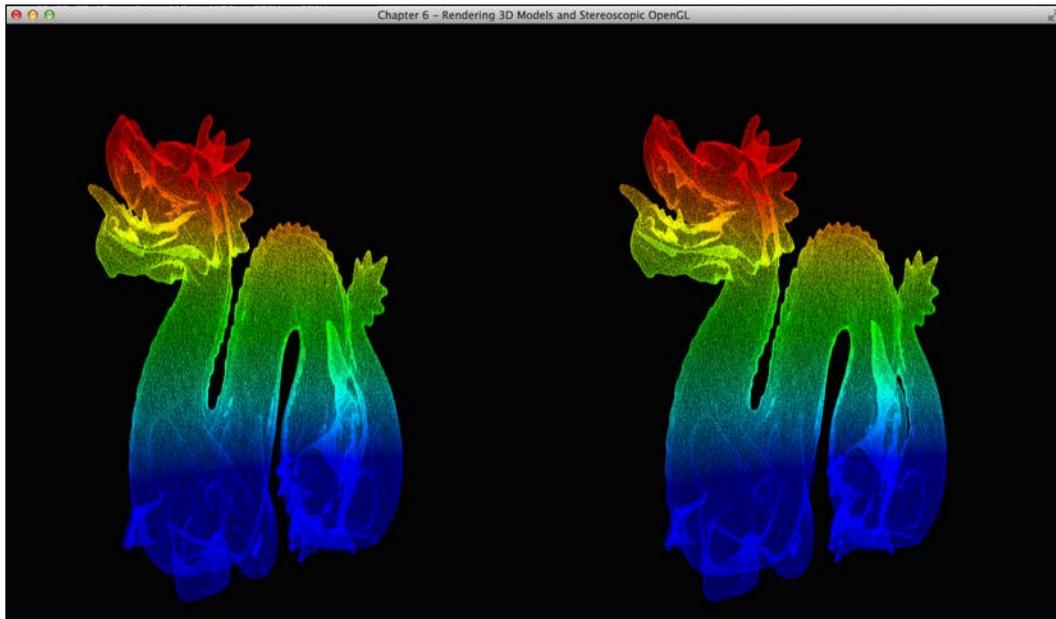
    if(drawPoints)
        obj_loader->draw(GL_POINTS);

    if(drawLines)
        obj_loader->draw(GL_LINES);
    //Draw the RIGHT eye, right half of the screen
    glViewport(width/2, 0, width/2, height);
    computeStereoViewProjectionMatrices(g_window, IOD, depthZ,
        false);
    projection_matrix = getProjectionMatrix();
    view_matrix = getViewMatrix();
    model_matrix = glm::mat4(1.0);
    model_matrix = glm::translate(model_matrix, glm::vec3(0.0f,
        0.0f, -depthZ));
    model_matrix = glm::rotate(model_matrix, glm::pi<float>() *
        rotateY, glm::vec3(0.0f, 1.0f, 0.0f));
```

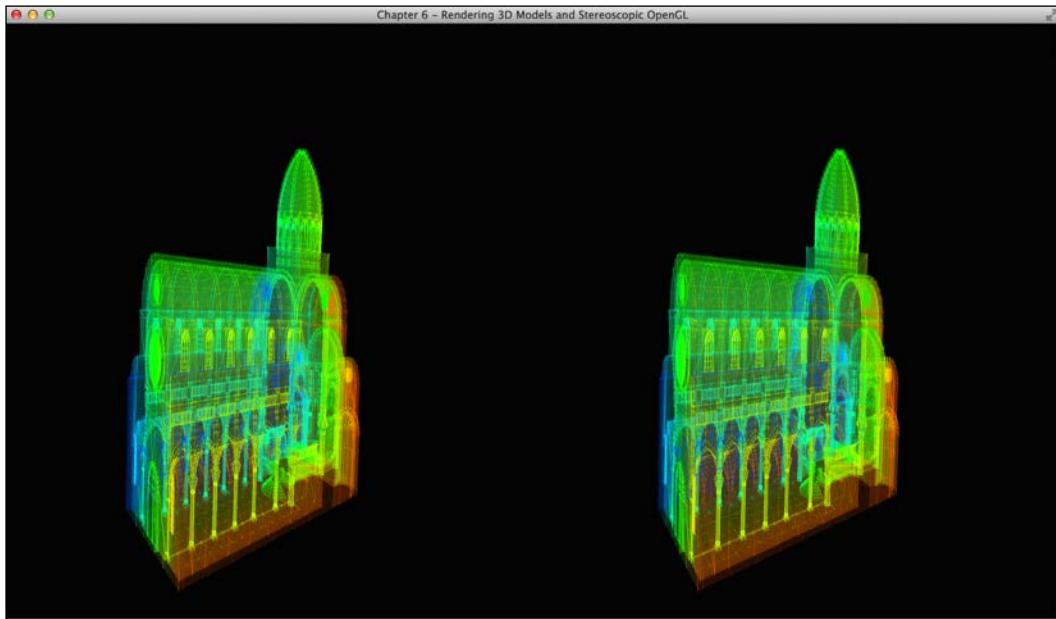
```
model_matrix = glm::rotate(model_matrix, glm::pi<float>() *  
    rotateX, glm::vec3(1.0f, 0.0f, 0.0f));  
mvp = projection_matrix * view_matrix * model_matrix;  
glUniformMatrix4fv(matrix_id, 1, GL_FALSE, &mvp[0][0]);  
if(drawTriangles)  
    obj_loader->draw(GL_TRIANGLES);  
if(drawPoints)  
    obj_loader->draw(GL_POINTS);  
if(drawLines)  
    obj_loader->draw(GL_LINES);  
}
```

The final rendering result consists of two separate images on each side of the display, and note that each image is compressed horizontally by a scaling factor of two. For some display systems, each side of the display is required to preserve the same aspect ratio depending on the specifications of the display.

Here are the final screenshots of the same models in true 3D using stereoscopic 3D rendering:



Here's the rendering of the architectural model in stereoscopic 3D:



How it works...

The stereoscopic 3D rendering technique is based on the parallel axis and asymmetric frustum perspective projection principle. In simpler terms, we rendered a separate image for each eye as if the object was seen at a different eye position but viewed on the same plane. Parameters such as the intraocular distance and frustum shift can be dynamically adjusted to provide the desired 3D stereo effects.

For example, by increasing or decreasing the frustum asymmetry parameter, the object will appear to be moved in front or behind the plane of the screen. By default, the zero parallax plane is set to the middle of the view volume. That is, the object is set up so that the center position of the object is positioned at the screen level, and some parts of the object will appear in front of or behind the screen. By increasing the frustum asymmetry (that is, positive parallax), the scene will appear to be pushed behind the screen. Likewise, by decreasing the frustum asymmetry (that is, negative parallax), the scene will appear to be pulled in front of the screen.

The `glm::frustum` function sets up the projection matrix, and we implemented the asymmetric frustum projection concept illustrated in the drawing. Then, we use the `glm::lookAt` function to adjust the eye position based on the IOP value we have selected.

To project the images side by side, we use the `glViewport` function to constrain the area within which the graphics can be rendered. The function basically performs an affine transformation (that is, scale and translation) which maps the normalized device coordinate to the window coordinate. Note that the final result is a side-by-side image in which the graphic is scaled by a factor of two vertically (or compressed horizontally). Depending on the hardware configuration, we may need to adjust the aspect ratio.

The current implementation supports side-by-side 3D, which is commonly used in most wearable **Augmented Reality (AR)** or **Virtual Reality (VR)** glasses. Fundamentally, the rendering technique, namely the asymmetric frustum perspective projection described in our chapter, is platform-independent. For example, we have successfully tested our implementation on the Meta 1 Developer Kit (<https://www.getmeta.com/products>) and rendered the final results on the optical see-through stereoscopic 3D display:



Here is the front view of the Meta 1 Developer Kit, showing the optical see-through stereoscopic 3D display and 3D range-sensing camera (introduced in *Chapter 5, Rendering of Point Cloud Data for 3D Range-sensing Cameras*):



The result is shown as follows, with the stereoscopic 3D graphics rendered onto the real world (which forms the basis of augmented reality):





In the upcoming chapters, we will transition to the increasingly powerful and ubiquitous mobile platform and introduce how to use OpenGL to visualize data in interesting ways using built-in motion sensors on mobile devices. Further details on implementing augmented reality applications will be covered in *Chapter 9, Augmented reality-based visualization on mobile or wearable platforms*.

See also

In addition, we can easily extend our code to support shutter glasses-based 3D monitors by utilizing the Quad Buffered OpenGL APIs (refer to the `GL_BACK_RIGHT` and `GL_BACK_LEFT` flags in the `glDrawBuffer` function). Unfortunately, such 3D formats require specific hardware synchronization and often require higher frame rate display (for example, 120Hz) as well as a professional graphics card. Further information on how to implement stereoscopic 3D in your application can be found at http://www.nvidia.com/content/GTC-2010/pdfs/2010_GTC2010.pdf.

7

An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0

In this chapter, we will cover the following topics:

- ▶ Setting up the Android SDK
- ▶ Setting up the **Android Native Development Kit (NDK)**
- ▶ Developing a basic framework to integrate the Android NDK
- ▶ Creating your first Android application with OpenGL ES 3.0

Introduction

In this chapter, we will transition to an increasingly powerful and ubiquitous computing platform by demonstrating how to visualize data on the latest mobile devices, from smart phones to tablets, using **OpenGL for Embedded Systems (OpenGL ES)**. As mobile devices become more ubiquitous and with their increasing computing capability, we now have an unprecedented opportunity to develop novel interactive data visualization tools using high-performance graphics hardware directly integrated into modern mobile devices.

OpenGL ES plays an important role in standardizing the 2D and 3D graphics APIs to allow the large-scale deployment of mobile applications on embedded systems with various hardware settings. Among the various mobile platforms (predominantly Google Android, Apple iOS, and Microsoft Windows Phone), the Android mobile operating system is currently one of the most popular ones. Therefore, in this chapter, we will focus primarily on the development of an Android-based application (API 18 and higher) using OpenGL ES 3.0, which provides a newer version of GLSL support (including full support for integer and 32-bit floating point operations) and enhanced texture rendering support. Nevertheless, OpenGL ES 3.0 is also supported on other mobile platforms, such as Apple iOS and Microsoft Phone.

Here, we will first introduce how to set up the Android development platform, including the SDK that provides the essential tools to build mobile applications, and the NDK, which enables the use of native-code languages (C/C++) for high-performance scientific computing and simulations by exploiting direct hardware acceleration. We will provide a script to simplify the process of deploying your first Android-based application on your mobile device.

Setting up the Android SDK

The Google Android OS website provides a standalone package for Android application development called the **Android SDK**. It contains all the necessary compilation and debugging tools to develop an Android application (except native code support, which is provided by the Android NDK). The upcoming steps explain the installation procedure in Mac OS X or, similarly, in Linux, with minor modifications to the script and binary packages required.

How to do it...

To install the Android SDK, follow these steps:

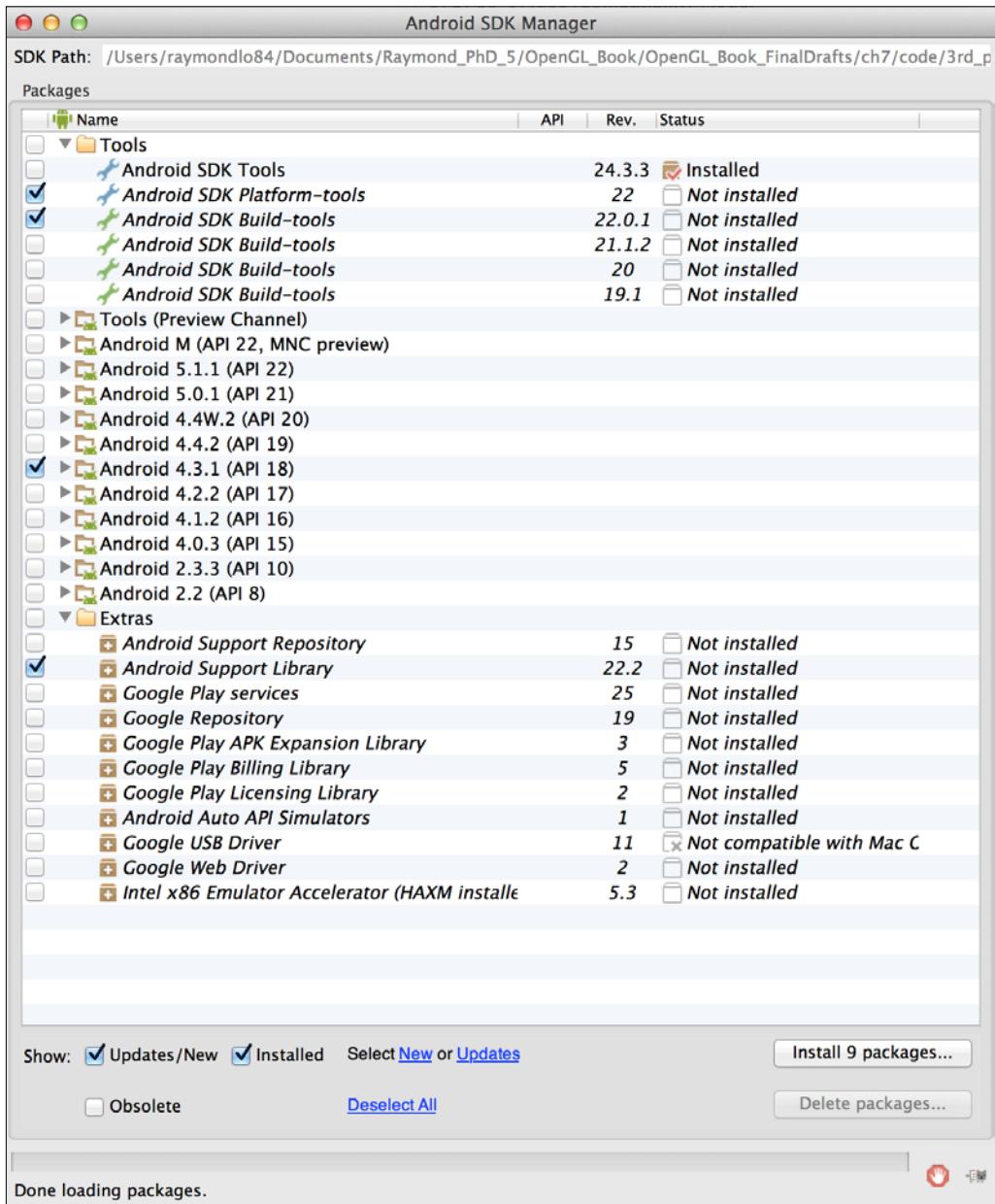
1. Download the standalone package from the Android Developers website at
http://dl.google.com/android/android-sdk_r24.3.3-macosx.zip.
2. Create a new directory called `3rd_party/android` and move the setup file into this folder:

```
mkdir 3rd_party/android
mv android-sdk_r24.3.3-macosx.zip 3rd_party/android
```
3. Unzip the package:

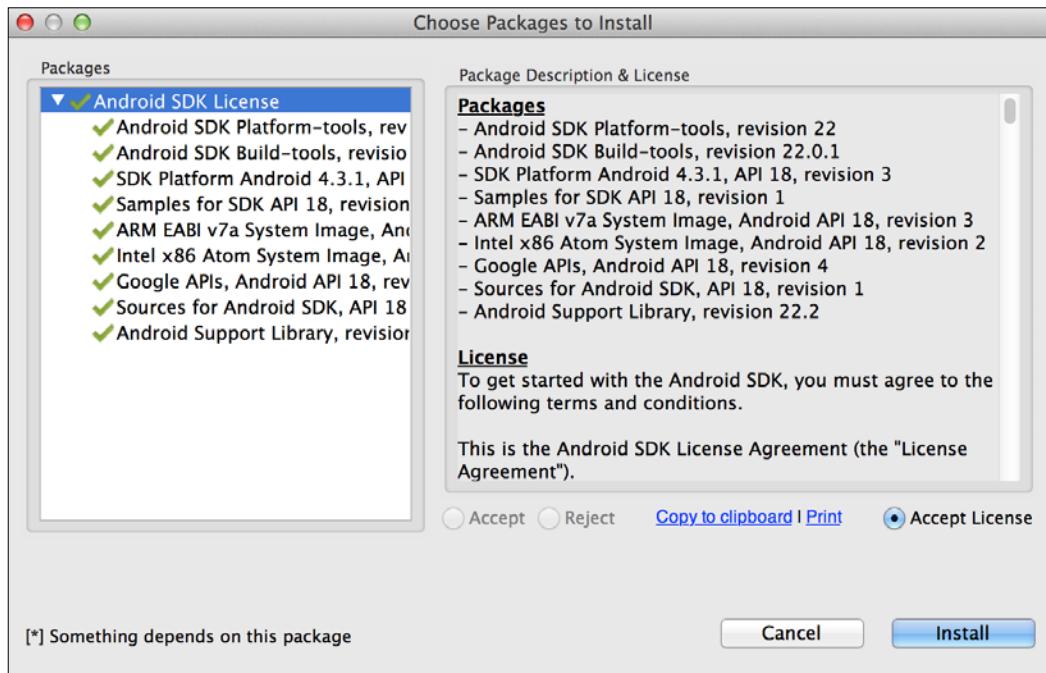
```
cd 3rd_party/android && unzip android-sdk_r24.3.3-macosx.zip
```
4. Execute the Android SDK Manager:

```
./android-sdk-macosx/tools/android
```

5. Select **Android 4.3.1 (API 18)** from the list of packages in addition to the default options. Deselect **Android M (API22, MNC preview)** and **Android 5.1.1 (API 22)**. Press the **Install 9 packages...** button on the **Android SDK Manager** screen, as shown here:



6. Select **Accept License** and click on the **Install** button:



7. To verify the installation, type the following command into the terminal:

```
./android-sdk-macosx/tools/android list
```

8. This is an example that illustrates the successful installation of the Android 4.3.1 platform:

```
Available Android targets:  
-----  
id: 1 or "android-18"  
Name: Android 4.3.1  
Type: Platform  
API level: 18  
Revision: 3  
Skins: HVGA, QVGA, WQVGA400, WQVGA432, WSVGA, WVGA800  
      (default), WVGA854, WXGA720, WXGA800, WXGA800-7in  
Tag/ABIs : default/armeabi-v7a, default/x86  
...
```

9. Finally, we will install Apache Ant to automate the software build process for Android application development. We can easily obtain the Apache Ant package by using MacPort with the command line or from its official website at <http://ant.apache.org/>:

```
sudo port install apache-ant
```

See also

To install the Android SDK in Linux or Windows, download the corresponding installation files and follow the instructions on the Android developer website at <https://developer.android.com/sdk/index.html>.

The setup procedures to set up the Android SDK in Linux are essentially identical using the command-line interface, except that a different standalone package should be downloaded using this link: http://dl.google.com/android/android-sdk_r24.3.3-linux.tgz.

In addition, for Windows users, the standalone package can be obtained using this link: http://dl.google.com/android/installer_r24.3.3-windows.exe.

To verify that your mobile phone has proper OpenGL ES 3.0 support, consult the Android documentation on how to check the OpenGL ES version at runtime: <https://developer.android.com/guide/topics/graphics/opengl.html#version-check>.

Setting up the Android Native Development Kit (NDK)

The Android NDK environment is essential for native-code language development. Here, we will outline the setup steps for the Mac OS X platform again.

How to do it...

To install the Android NDK, follow these steps:

1. Download the NDK installation package from the Android developer website at http://dl.google.com/android/ndk/android-ndk-r10e-darwin-x86_64.bin.
2. Move the setup file into the same installation folder:

```
mv android-ndk-r10e-darwin-x86_64.bin 3rd_party/android
```
3. Set the permission of the file to be an executable:

```
cd 3rd_party/android && chmod +x android-ndk-r10e-darwin-x86_64.bin
```

4. Run the NDK installation package:

```
./android-ndk-r10e-darwin-x86_64.bin
```

5. The installation process is fully automated and the following output confirms the successful installation of the Android NDK:

```
...
Extracting android-ndk-r10e/build/tools
Extracting android-ndk-r10e/build/gmsl
Extracting android-ndk-r10e/build/core
Extracting android-ndk-r10e/build/awk
Extracting android-ndk-r10e/build
Extracting android-ndk-r10e
```

```
Everything is Ok
```

See also

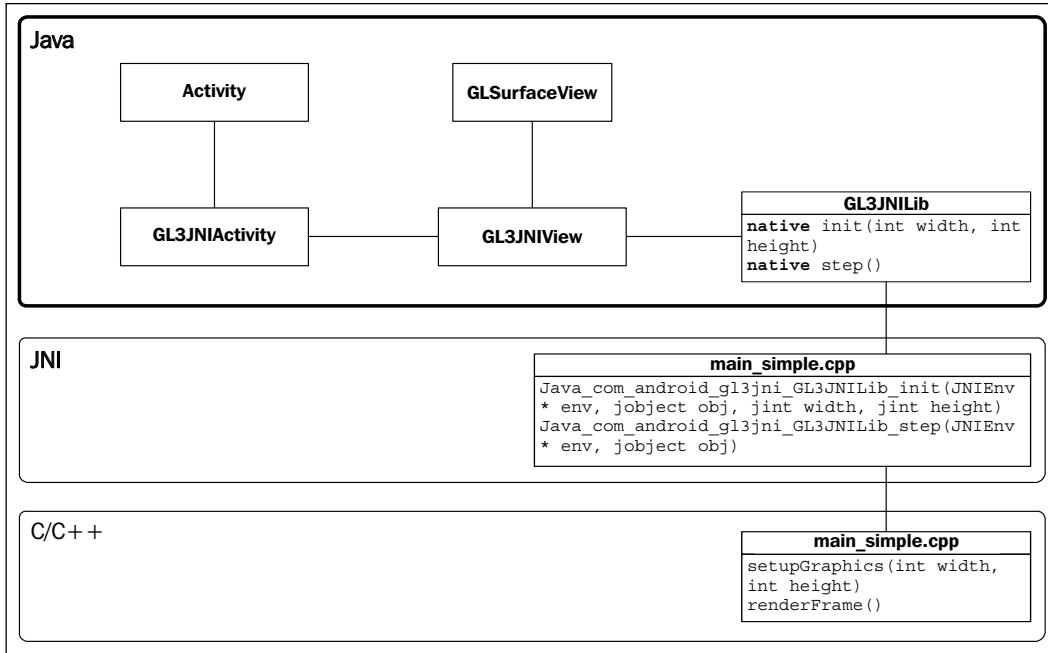
To install the Android NDK on Linux or Windows, download the corresponding installation files and follow the instructions on the Android developer website at <https://developer.android.com/tools/sdk/ndk/index.html>.

Developing a basic framework to integrate the Android NDK

Now that we have successfully installed the Android SDK and NDK, we will demonstrate how to develop a basic framework to integrate native C/C++ code into a Java-based Android application. Here, we describe the general mechanism to create high-performance code for deployment on mobile devices using OpenGL ES 3.0.

OpenGL ES 3.0 supports both Java and C/C++ interfaces. Depending on the specific requirements of the application, you may choose to implement the solution in Java due to its flexibility and portability. For high-performance computing and applications that require a high memory bandwidth, it is preferable that you use the NDK for fine-grain optimization and memory management. In addition, we can port our existing libraries, such as OpenCV with Android NDK, using static library linking. The cross-platform compilation capability opens up many possibilities for real-time image and signal processing on a mobile platform with minimal development effort.

Here, we introduce a basic framework that consists of three classes: `GL3JNIActivity`, `GL3JNIView`, and `GL3JNIActivity`. We show a simplified class diagram in the following figure, illustrating the relationship between the classes. The native code (C/C++) is implemented separately and will be described in detail in the next section:



How to do it...

First, we will create the core Java source files that are essential to an Android application. These files serve as a wrapper for our OpenGL ES 3.0 native code:

1. In the project directory, create a folder named `src/com/android/gl3jni` with the following command:

```
mkdir src/com/android/gl3jni
```

2. Create the first class, `GL3JNIActivity`, in the Java source file, `GL3JNIActivity.java`, within the new folder, `src/com/android/gl3jni/`:

```
package com.android.gl3jni;

import android.app.Activity;
import android.os.Bundle;
/**
 * Main application for Android
 */
public class GL3JNIActivity extends Activity {

    GL3JNIView mView;
```

```
@Override protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    mView = new GL3JNIView(getApplicationContext());
    setContentView(mView);
}

@Override protected void onPause() {
    super.onPause();
    mView.onPause();
}

@Override protected void onResume() {
    super.onResume();
    mView.onResume();
}
```

3. Next, implement the `GL3JNIView` class, which handles the OpenGL rendering setup in the `GL3JNIView.java` source file inside `src/com/android/gl3jni/`:

```
package com.android.gl3jni;

import android.content.Context;
import android.opengl.GLSurfaceView;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

/**
 * A simple application that uses OpenGL ES3 and GLSurface
 */
class GL3JNIView extends GLSurfaceView {
    public GL3JNIView(Context context) {
        super(context);
        /* Pick an EGLConfig with RGB8 color, 16-bit depth,
        no stencil, supporting OpenGL ES 3.0 or later */
        setEGLConfigChooser(8, 8, 8, 0, 16, 0);
        setEGLContextClientVersion(3);
        setRenderer(new Renderer());
    }
    private static class Renderer implements
        GLSurfaceView.Renderer {
        public void onDrawFrame(GL10 gl) {
```

```
        GL3JNILib.step();
    }

    public void onSurfaceChanged(GL10 gl, int width,
        int height) {
        GL3JNILib.init(width, height);
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig
        config) {
    }
}
}
```

4. Finally, create the `GL3JNILib` class to handle native library loading and calling in `GL3JNILib.java` inside `src/com/android/gl3jni`:

```
package com.android.gl3jni;

public class GL3JNILib {
    static {
        System.loadLibrary("gl3jni");
    }

    public static native void init(int width, int height);
    public static native void step();
}
```

5. Now, in the project directory of the project, add the `AndroidManifest.xml` file, which contains all the essential information about your application on the Android system:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
"http://schemas.android.com/apk/res/android"
    package="com.android.gl3jni">
    <application android:label=
"@string/gl3jni_activity">
        <activity android:name="GL3JNIActivity"
            android:theme=
"@android:style/Theme.NoTitleBar.Fullscreen"
            android:launchMode="singleTask"
            android:configChanges=
"orientation|keyboardHidden">
```

```

<intent-filter>
    <action android:name=
        "android.intent.action.MAIN" />
    <category android:name=
        "android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-feature android:glEsVersion="0x00030000"/>
<uses-sdk android:minSdkVersion="18"/>
</manifest>

```

- In the res/values/ directory, add the strings.xml file, which saves our application's name:

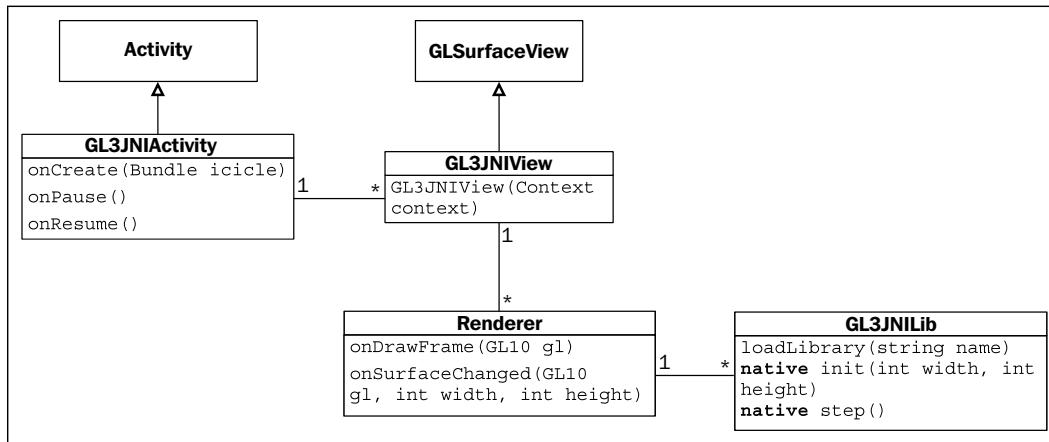
```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="gl3jni_activity">OpenGL ES Demo</string>
</resources>

```

How it works...

The following class diagram illustrates the core functions and relationships between the classes. Similar to all other Android applications with a user interface, we define the **Activity** class, which handles the core interactions. The implementation of GL3JNIActivity is straightforward. It captures the events from the Android application (for example, onPause and onResume) and also creates an instance of the GL3JNIView class, which handles graphics rendering. Instead of adding UI elements, such as textboxes or labels, we create a surface based on GLSurfaceView, which handles hardware-accelerated OpenGL rendering:



The `GL3JNIView` class is a subclass of the `GLSurfaceView` class, which provides a dedicated surface for OpenGL rendering. We choose the `RGB8` color mode, a 16-bit depth buffer, and no stencil with the `setEGLConfigChooser` function and ensure that the environment is set up for OpenGL ES 3.0 by using the `setEGLContextClientVersion` function. The `setRenderer` function then registers the custom `Renderer` class, which is responsible for the actual OpenGL rendering.

The `Renderer` class implements the key event functions—`onDrawFrame`, `onSurfaceChanged`, and `onSurfaceCreated`—in the rendering loop. These functions connect to the native implementation (C/C++) portion of the code that is handled by the `GL3JNILib` class.

Finally, the `GL3JNILib` class creates the interface to communicate with the native code functions. First, it loads the native library named `gl3jni`, which contains the actual OpenGL ES 3.0 implementation. The function prototypes, `step` and `init`, are used to interface with the native code, which will be defined separately in the next section. Note that we can also pass in the canvas width and height values to the native functions as parameters.

The `AndroidManifest.xml` and `strings.xml` files are the configuration files required by the Android application, and they must be stored in the root directory of the project in the XML format. The `AndroidManifest.xml` file defines all the essential information including the name of the Java package and the declaration of permission requirements (for example, file read/write access), as well as the minimum version of the Android API that the application requires.

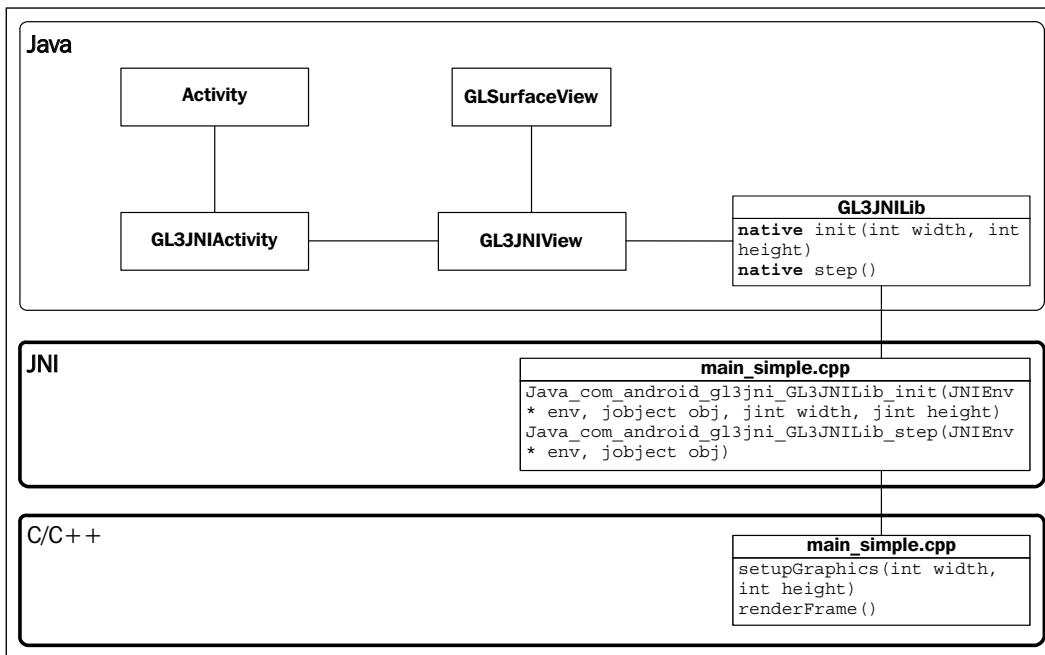
See also

For further information on Android application development, the Android Developers website provides detailed documentation on the API at <http://developer.android.com/guide/index.html>.

For further information on using OpenGL ES within an Android application, the Android programming guide describes the programming workflow in detail and provides useful examples at <http://developer.android.com/training/graphics/opengl/environment.html>.

Creating your first Android application with OpenGL ES 3.0

In this section, we will complete our implementation with native code in C/C++ to create the first Android application with OpenGL ES 3.0. As illustrated in the simplified class diagram, the Java code only provides the basic interface on the mobile device. Now, on the C/C++ side, we implement all the functionalities previously defined on the Java side and also include all the required libraries from OpenGL ES 3.0 (inside the `main_simple.cpp` file). The `main_simple.cpp` file also defines the key interface between the C/C++ and Java side by using the **Java Native Interface (JNI)**:



Getting ready

We assume that you have installed all the prerequisite tools from the Android SDK and NDK in addition to setting up the basic framework introduced in the previous section. Also, you should review the basics of shader programming, introduced in earlier chapters, before you proceed.

How to do it...

Here, we describe the implementation of the OpenGL ES 3.0 native code to complete the demo application:

1. In the project directory, create a folder named `jni` by using the following command:

```
mkdir jni
```

2. Create a file named `main_simple.cpp` and store it inside the `jni` directory.

3. Include all necessary header files for JNI and OpenGL ES 3.0:

```
//header for JNI  
#include <jni.h>
```

```
//header for the OpenGL ES3 library  
#include <GLES3/gl3.h>
```

4. Include the logging header and define the macros to show the debug messages:

```
#include <android/log.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
  
//android error log interface  
#define LOG_TAG "libgl3jni"  
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_  
TAG, __VA_ARGS__)  
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_  
TAG, __VA_ARGS__)
```

5. Declare the shader program variables for our demo application:

```
GLuint gProgram;  
GLuint gvPositionHandle;  
GLuint gvColorHandle;
```

```
int width = 1280;  
int height = 720;
```

6. Define the shader program code for the vertex shader and the fragment shader:

```
// Vertex shader source code  
static const char g_vshader_code[] =  
"#version 300 es\n"  
"in vec4 vPosition;\n"
```

```
"in vec4 vColor;\n"
"out vec4 color;\n"
"void main() {\n"
    " gl_Position = vPosition;\n"
    " color = vColor;\n"
}\n";

// fragment shader source code
static const char g_fshader_code[] =
"#version 300 es\n"
"precision mediump float;\n"
"in vec4 color;\n"
"out vec4 color_out;\n"
"void main() {\n"
    " color_out = color;\n"
}\n";
```

7. Implement the error call handlers for OpenGL ES, using the Android log:

```
/**
 * Print out the error string from OpenGL
 */
static void printGLString(const char *name, GLenum s) {
    const char *v = (const char *) glGetString(s);
    LOGI("GL %s = %s\n", name, v);
}

/**
 * Error checking with OpenGL calls
 */
static void checkGlError(const char* op) {
    for (GLint error = glGetError(); error; error
        = glGetError()) {
        LOGI("After %s() glError (0x%x)\n", op, error);
    }
}
```

8. Implement the vertex or fragment program-loading mechanisms. The warning and error messages are redirected to the Android log output:

```
GLuint loadShader(GLenum shader_type, const char* p_source) {
    GLuint shader = glCreateShader(shader_type);
    if (shader) {
        glShaderSource(shader, 1, &p_source, 0);
        glCompileShader(shader);
```

```
GLint compiled = 0;
glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);

//Report error and delete the shader
if (!compiled) {
    GLint infoLen = 0;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);
    if (infoLen) {
        char* buf = (char*) malloc(infoLen);
        if (buf) {
            glGetShaderInfoLog(shader, infoLen, 0, buf);
            LOGE("Could not compile shader %d:\n%s\n",
                 shader_type, buf);
            free(buf);
        }
        glDeleteShader(shader);
        shader = 0;
    }
}
return shader;
}
```

9. Implement the shader program creation mechanism. The function also attaches and links the shader program:

```
GLuint createShaderProgram(const char *vertex_shader_code,
                           const char *fragment_shader_code) {
    //create the vertex and fragment shaders
    GLuint vertex_shader_id = loadShader(GL_VERTEX_SHADER,
                                          vertex_shader_code);
    if (!vertex_shader_id) {
        return 0;
    }

    GLuint fragment_shader_id =
        loadShader(GL_FRAGMENT_SHADER, fragment_shader_code);
    if (!fragment_shader_id) {
        return 0;
    }

    GLint result = GL_FALSE;
    //link the program
    GLuint program_id = glCreateProgram();
    glAttachShader(program_id, vertex_shader_id);
    checkGLError("glAttachShader");
```

```
glAttachShader(program_id, fragment_shader_id);
checkGlError("glAttachShader");
glLinkProgram(program_id);

//check the program and ensure that the program is linked properly
glGetProgramiv(program_id, GL_LINK_STATUS, &result);
if ( result != GL_TRUE ){
    //error handling with Android
    GLint bufLength = 0;
    glGetProgramiv(program_id, GL_INFO_LOG_LENGTH,
        &bufLength);
    if (bufLength) {
        char* buf = (char*) malloc(bufLength);
        if (buf) {
            glGetProgramInfoLog(program_id, bufLength, 0, buf);
            LOGE("Could not link program:\n%s\n",
                buf);
            free(buf);
        }
    }
    glDeleteProgram(program_id);
    program_id = 0;
}
else {
    LOGI("Linked program Successfully\n");
}

glDeleteShader(vertex_shader_id);
glDeleteShader(fragment_shader_id);

return program_id;
}
```

10. Create a function to handle the initialization. This function is a helper function that handles requests from the Java side:

```
bool setupGraphics(int w, int h) {
    printGLString("Version", GL_VERSION);
    printGLString("Vendor", GL_VENDOR);
    printGLString("Renderer", GL_RENDERER);
    printGLString("Extensions", GL_EXTENSIONS);

    LOGI("setupGraphics(%d, %d)", w, h);
    gProgram = createShaderProgram(g_vshader_code,
        g_fshader_code);
    if (!gProgram) {
        LOGE("Could not create program.");
        return false;
    }
}
```

```
    gvPositionHandle = glGetAttribLocation(gProgram,
        "vPosition");
    checkGlError("glGetAttribLocation");
    LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
        gvPositionHandle);

    gvColorHandle = glGetAttribLocation(gProgram,
        "vColor");
    checkGlError("glGetAttribLocation");
    LOGI("glGetAttribLocation(\"vColor\") = %d\n",
        gvColorHandle);

    glViewport(0, 0, w, h);
    width = w;
    height = h;

    checkGlError("glViewport");

    return true;
}
```

11. Set up the rendering function that draws a triangle on the screen with red, green, and blue vertices:

```
//vertices
GLfloat gTriangle[9]={-1.0f, -1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    0.0f, 1.0f, 0.0f};
GLfloat gColor[9]={1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f};

void renderFrame() {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    checkGlError("glClearColor");

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    checkGlError("glClear");

    glUseProgram(gProgram);
    checkGlError("glUseProgram");

    glVertexAttribPointer(gvPositionHandle, 3, GL_FLOAT,
        GL_FALSE, 0, gTriangle);
    checkGlError("glVertexAttribPointer");

    glVertexAttribPointer(gvColorHandle, 3, GL_FLOAT,
        GL_FALSE, 0, gColor);
    checkGlError("glVertexAttribPointer");
```

```
glEnableVertexAttribArray(gvPositionHandle);  
checkGlError("glEnableVertexAttribArray");  
  
glEnableVertexAttribArray(gvColorHandle);  
checkGlError("glEnableVertexAttribArray");  
  
glDrawArrays(GL_TRIANGLES, 0, 9);  
checkGlError("glDrawArrays");  
}
```

12. Define the JNI prototypes that connect to the Java side. These calls are the interfaces to communicate between the Java code and the C/C++ native code:

```
//external calls for Java  
extern "C" {  
    JNIEXPORT void JNICALL  
        Java_com_android_gl3jni_GL3JNILib_init(JNIEnv * env,  
            jobject obj, jint width, jint height);  
    JNIEXPORT void JNICALL  
        Java_com_android_gl3jni_GL3JNILib_step(JNIEnv * env,  
            jobject obj);  
};
```

13. Set up the internal function calls with the helper functions:

```
//link to internal calls  
JNIEXPORT void JNICALL  
Java_com_android_gl3jni_GL3JNILib_init(JNIEnv * env,  
    jobject obj, jint width, jint height)  
{  
    setupGraphics(width, height);  
}  
  
JNIEXPORT void JNICALL  
Java_com_android_gl3jni_GL3JNILib_step(JNIEnv * env,  
    jobject obj)  
{  
    renderFrame();  
}  
//end of file
```

14. Now that we have completed the implementation of the native code, we must compile the code and link it to the Android application. To compile the code, create a build file that is similar to a Makefile, called `Android.mk`, in the `jni` folder:

```
LOCAL_PATH:= $(call my-dir)  
  
include $(CLEAR_VARS)
```

```
LOCAL_MODULE      := libgl3jni
LOCAL_CFLAGS      := -Werror
#for simplified demo
LOCAL_SRC_FILES   := main_simple.cpp
LOCAL_LDLIBS       := -llog -lGLESv3

include $(BUILD_SHARED_LIBRARY)
```

15. In addition, we must create an `Application.mk` file that provides information about the build type, such as the **Application Binary Interface (ABI)**. The `Application.mk` file must be stored inside the `jni` directory:

```
APP_ABI := armeabi-v7a
#required for GLM and other static libraries
APP_STL := gnustl_static
```

16. At this point, we should have the following list of files in the root directory:

```
src/com/android/gl3jni/GL3JNIActivity.java
src/com/android/gl3jni/GL3JNILib.java
src/com/android/gl3jni/GL3JNIView.java
AndroidManifest.xml
res/value/strings.xml
jni/Android.mk
jni/Application.mk
jni/main_simple.cpp
```

To compile the native source code and deploy our application on a mobile phone, run the following `build` script in the terminal, which is shown as follows:

1. Set up our environment variables for the SDK and the NDK. (Note that the following relative paths assume that the SDK and NDK are installed 3 levels outside the current directory, where the `compile.sh` and `install.sh` scripts are executed in the code package. These paths should be modified to match your code directory structure as necessary.):

```
export ANDROID_SDK_PATH="../../../../3rd_party/android/android-sdk-macosx"
export ANDROID_NDK_PATH="../../../../3rd_party/android/android-ndk-r10e"
```

2. Initialize the project with the `android update` command for the first-time compilation. This will generate all the necessary files (such as the `build.xml` file) for later steps:

```
$ANDROID_SDK_PATH/tools/android update project -p . -s
--target "android-18"
```

3. Compile the JNI native code with the `build` command:

```
$ANDROID_NDK_PATH/ndk-build
```

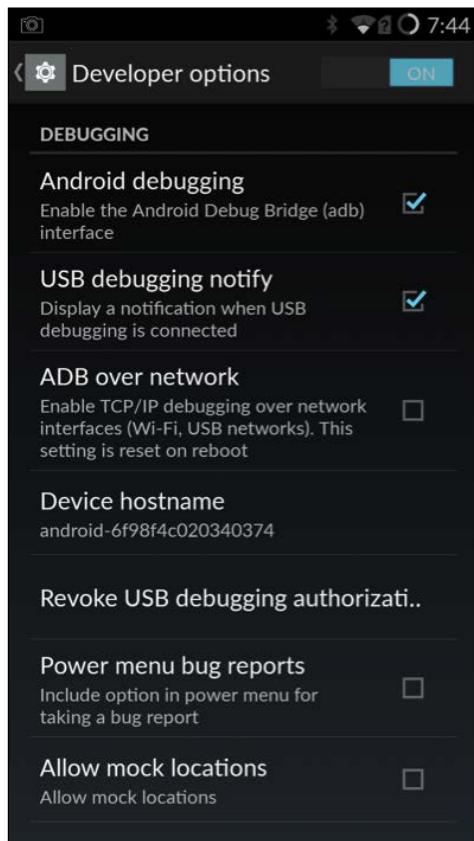
4. Run the build command. Apache Ant takes the `build.xml` script and builds the **Android Application Package (APK)** file that is ready for deployment:

```
ant debug
```

5. Install the Android application by using the **Android Debug Bridge (adb)** command:

```
$ANDROID_SDK_PATH/platform-tools/adb install -r  
bin/GL3JNIActivity-debug.apk
```

For this command to work, before connecting the mobile device through the USB port, ensure that the USB Debugging mode is enabled and accept any prompts for security-related warnings. On most devices, you can find this option by navigating to **Settings | Applications | Development** or **Settings | Developer**. However, on Android 4.2 or higher, this option is hidden by default and must be enabled by navigating to **Settings | About Phone** (or **About Tablet**) and tapping **Build Number** multiple times. For further details, follow the instructions provided on the official Android Developer website at <http://developer.android.com/tools/device.html>. Here is a sample screenshot of an Android phone with the USB debugging mode successfully configured:

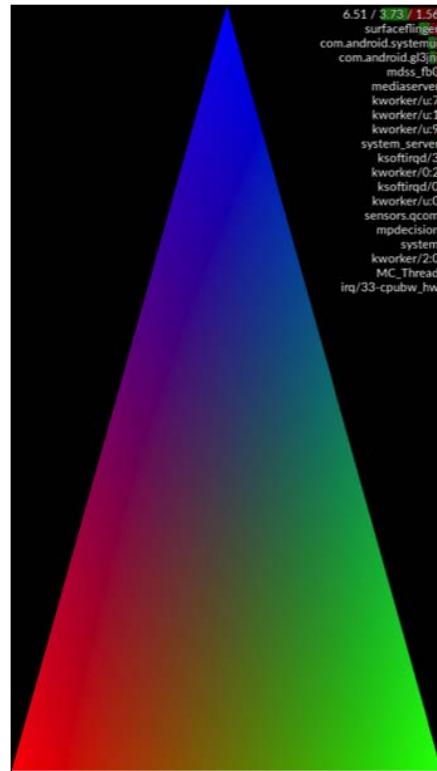


After the application is installed, we can execute the application as we normally do with any other Android application by opening it directly using the application icon on the phone, as shown here:

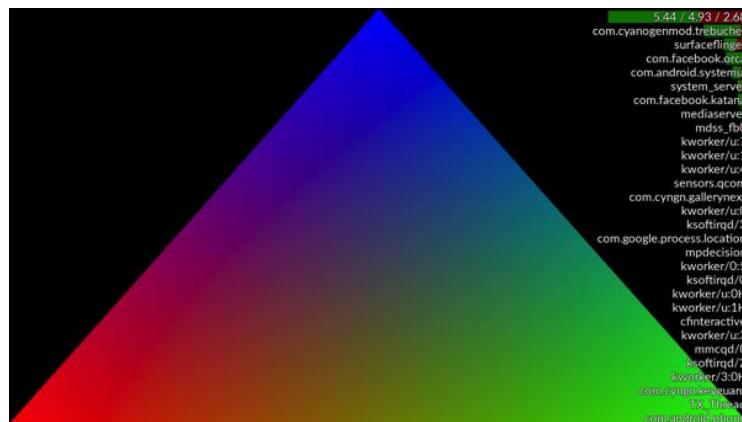


An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0 —

A screenshot after launching the application is shown next. Note that the CPU monitor has been enabled to show the CPU utilization. This is not enabled by default but can be found in **Developer Options**. The application supports both the portrait and landscape modes and the graphics automatically scale to the window size upon changing the frame buffer size:



Here is another screenshot of the landscape mode:



How it works...

This chapter demonstrates the portability of our approach in previous chapters. Essentially, the native code developed in this chapter resembles what we covered in previous chapters. In particular, the shader program's creation and loading mechanism is virtually identical, except that we have used a predefined string (`static char []`) to simplify the complexity of loading files in Android. However, there are some subtle differences. Here, we will list the differences and new features.

In the fragment program and vertex program, we need to add the `#version 300 es` directive to ensure that the shader code can access the new features, such as uniform blocks and the full support of integer and floating point operations. For example, OpenGL ES 3.0 replaces the attribute and varying qualifiers with the `in` and `out` keywords. This standardization allows much faster code development of OpenGL on various platforms.

The other notable difference is that we have replaced the GLFW library completely with the EGL library, which comes as a standard library in Android, for context management. All event handling, such as Windows management and user inputs, are now handled through the Android API and the native code is only responsible for graphics rendering.

The Android log and error reporting system is now accessible through the Android `adb` program. The interaction is similar to a terminal output, and we can see the log in real time with the following command:

```
adb logcat
```

For example, our application reports the OpenGL ES version, as well as the extensions supported by the mobile device in the log. With the preceding command, we can extract the following information:

```
I/libgl3jni( 6681): GL Version = OpenGL ES 3.0 V@66.0
AU@04.04.02.048.042_LNXBUILD_AU_LINUX_ANDROID_LNX.LA.3.5.1_
RB1.04.04.02.048.042+PATCH[ES]_msm8974_LNX.LA.3.5.1_RB1__release_ENGG
(CL@)
I/libgl3jni( 6681): GL Vendor = Qualcomm
I/libgl3jni( 6681): GL Renderer = Adreno (TM) 330
I/libgl3jni( 6681): GL Extensions = GL_AMD_compressed_ATC_texture
GL_AMD_performance_monitor GL_AMD_program_binary_Z400 GL_EXT_debug_
label GL_EXT_debug_marker GL_EXT_discard_framebuffer GL_EXT_robustness
GL_EXT_texture_format_BGRA8888 GL_EXT_texture_type_2_10_10_10_REV
GL_NV_fence GL_OES_compressed_ETC1_RGB8_texture GL_OES_depth_texture
GL_OES_depth24 GL_OES_EGL_image GL_OES_EGL_image_external GL_OES_
element_index_uint GL_OES_fbo_render_mipmap GL_OES_fragment_precision_
high GL_OES_get_program_binary GL_OES_packed_depth_stencil GL_OES_
depth_texture_cube_map GL_OES_rgb8_rgba8 GL_OES_standard_derivatives
GL_OES_texture_3D GL_OES_texture_float GL_OES_texture_half_float
GL_OES_texture_half_float_linear GL_OES_texture_npot GL_OES_vertex_
half_float GL_OES_vertex_type_10_10_10_2 GL_OES_vertex_array_object
```

An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0 —

```
GL_QCOM_alpha_test GL_QCOM_binning_control GL_QCOM_driver_control  
GL_QCOM_perfmon_global_mode GL_QCOM_extended_get GL_QCOM_extended_get2  
GL_QCOM_tiled_rendering GL_QCOM_writeonly_rendering GL_EXT_sRGB GL_  
EXT_sRGB_write_control GL_EXT_  
I/libgl3jni( 6681): setupGraphics(1440, 2560)
```

The real-time log data is very useful for debugging and can allow developers to quickly analyze the problem.

One common question is how the Java and C/C++ elements communicate with each other. The JNI syntax is rather puzzling to understand in the first place, but we can decode it by carefully analyzing the following code snippet:

```
JNIEEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_init  
(JNIEnv *env, jobject obj, jint width, jint height)
```

The JNIEEXPORT and JNICALL tags allow the functions to be located in the shared library at runtime. The class name is specified by com_android_gl3jni_GL3JNILib (com.android.gl3jni.GL3JNILib), and init is the method name of the Java native function. As we can see, the period in the class name is replaced by an underscore. In addition, we have two additional parameters, namely the width and height of the frame buffer. More parameters can be simply appended to the end of the parameters' list in the function, as required.

In terms of backward compatibility, we can see that OpenGL 4.3 is a complete superset of OpenGL ES 3.0. In OpenGL 3.1 and higher, we can see that the embedded system version of OpenGL and the standard Desktop version of OpenGL are slowly converging, which reduces the underlying complexity in maintaining various versions of OpenGL in the application life cycle.

See also

A detailed description of the Android OS architecture is beyond the scope of this book. However, you are encouraged to consult the official developer workflow guide at <http://developer.android.com/tools/workflow/index.html>.

Further information on the OpenGL ES Shading Language can be found at https://www.khronos.org/registry/gles/specs/3.0/GLSL_ES_Specification_3.0.0.3.pdf.

8

Interactive Real-time Data Visualization on Mobile Devices

In this chapter, we will cover the following topics:

- ▶ Visualizing real-time data from built-in Inertial Measurement Units (IMUs)
- ▶ Part I – handling multi-touch interface and motion sensor inputs
- ▶ Part II – interactive, real-time data visualization with mobile GPUs

Introduction

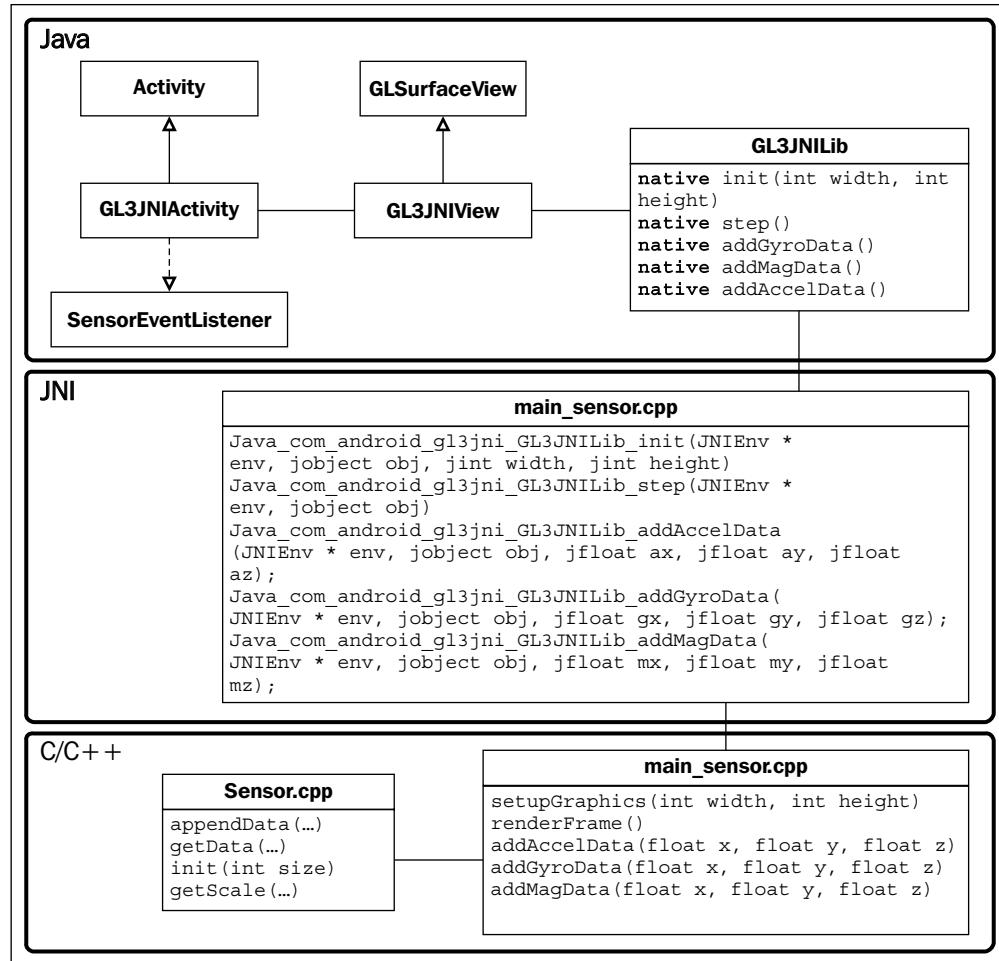
In this chapter, we will demonstrate how to visualize data interactively using built-in motion sensors called **Inertial Measurement Units (IMUs)** and the multi-touch interface on mobile devices. We will further explore the use of shader programs to accelerate computationally intensive operations to enable real-time visualization of 3D data with mobile graphics hardware. We will assume familiarity with the basic framework for building an Android-based OpenGL ES 3.0 application introduced in the previous chapter and add significantly more complexity in the implementation in this chapter to achieve interactive, real-time 3D visualization of a Gaussian function using both motion sensors and the multi-touch gesture interface. The final demo is designed to work on any Android-based mobile device with proper sensor hardware support.

Here, we will first introduce how to extract data directly from the IMUs and plot the real-time data stream acquired on an Android device. We will divide the final demo into two parts given its complexity. In part I, we will demonstrate how to handle the multi-touch interface and motion sensor inputs on the Java side. In part II, we will demonstrate how to implement the shader program in OpenGL ES 3.0 and other components of the native code to finish our interactive demo.

Visualizing real-time data from built-in Inertial Measurement Units (IMUs)

Many modern mobile devices now integrate a plethora of built-in sensors including various motion and position sensors (such as an accelerometer, gyroscope, and magnetometer/digital compass) to enable novel forms of user interaction (such as complex gesture and motion control) as well as other environmental sensors, which can measure environmental conditions (such as an ambient light sensor and proximity sensor) to enable smart wearable applications. The Android Sensor Framework provides a comprehensive interface to access many types of sensors, which can be either hardware-based (physical sensors) or software-based (virtual sensors that derive inputs from hardware sensors). In general, there are three major categories of sensors—motion sensors, position sensors, and environmental sensors.

In this section, we will demonstrate how to utilize the Android Sensor Framework to communicate with the sensors available on your device, register sensor event listeners to monitor changes in the sensors, and acquire raw sensor data for display on your mobile device. To create this demo, we will implement the Java code and native code using the same framework design introduced in the previous chapter. The following block diagram illustrates the core functions and the relationship among the classes that will be implemented in this demo:



Getting ready

This demo requires an Android device with OpenGL ES 3.0 support as well as physical sensor hardware support. Unfortunately, at the moment these functions cannot be simulated with an emulator shipped with the Android SDK. Specifically, an Android mobile device with the following set of sensors, which are now commonly available, would be required to run this demo: an accelerometer, gyroscope, and magnetometer (digital compass).

In addition, we assume that the Android SDK and Android NDK are configured as discussed in *Chapter 7, An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0*.

How to do it...

First, we will create the core Java source files similar to the previous chapter. Since the majority of the code is similar, we will only discuss the new and significant elements that are introduced in the current code. The rest of the code is abbreviated with the "..." notation. Please download the complete source code from the official Packt Publishing website.

In the `GL3JNIActivity.java` file, we first integrate Android Sensor Manager, which allows us to read and parse sensor data. The following steps are required to complete the integration:

1. Import the classes for the Android Sensor Manager:

```
package com.android.gl3jni;  
...  
import android.hardware.Sensor;  
import android.hardware.SensorEvent;  
import android.hardware.SensorEventListener;  
import android.hardware.SensorManager;  
...
```

2. Add the `SensorEventListener` interface to interact with the sensors:

```
public class GL3JNIActivity extends Activity implements  
SensorEventListener{
```

3. Define the `SensorManager` and the `Sensor` variables to handle the data from the accelerometer, gyroscope, and magnetometer:

```
...  
private SensorManager mSensorManager;  
private Sensor mAccelerometer;  
private Sensor mGyro;  
private Sensor mMag;
```

4. Initialize the `SensorManager` as well as all other sensor services:

```
@Override protected void onCreate(Bundle icicle) {  
    super.onCreate(icicle);  
    setRequestedOrientation(  
        ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);  
  
    mSensorManager =  
        (SensorManager) getSystemService(SENSOR_SERVICE);
```

```
mAccelerometer = mSensorManager.getDefaultSensor(  
    Sensor.TYPE_ACCELEROMETER);  
mGyro = mSensorManager.getDefaultSensor(  
    Sensor.TYPE_GYROSCOPE);  
mMag = mSensorManager.getDefaultSensor(  
    Sensor.TYPE_MAGNETIC_FIELD);  
mView = new GL3JNIView(getApplicationContext());  
setContentView(mView);  
}
```

5. Register the callback functions and start listening to these events:

```
@Override protected void onPause() {  
    super.onPause();  
    mView.onPause();  
    //unregister accelerometer and other sensors  
    mSensorManager.unregisterListener(this, mAccelerometer);  
    mSensorManager.unregisterListener(this, mGyro);  
    mSensorManager.unregisterListener(this, mMag);  
}  
  
@Override protected void onResume() {  
    super.onResume();  
    mView.onResume();  
    /* register and activate the sensors. Start streaming  
       data and handle with callback functions */  
    mSensorManager.registerListener(this,  
        mAccelerometer, SensorManager.SENSOR_DELAY_GAME);  
    mSensorManager.registerListener(this,  
        mGyro, SensorManager.SENSOR_DELAY_GAME);  
    mSensorManager.registerListener(this,  
        mMag, SensorManager.SENSOR_DELAY_GAME);  
}
```

6. Handle the sensor events. The `onSensorChanged` and `onAccuracyChanged` functions capture any changes detected and the `SensorEvent` variable holds all the information about the sensor type, time-stamp, accuracy, and so on:

```
@Override  
public void onAccuracyChanged(Sensor sensor, int  
    accuracy) {  
    //included for completeness  
}  
@Override  
public void onSensorChanged(SensorEvent event) {  
    //handle the accelerometer data  
    //All values are in SI units (m/s^2)
```

```
if (event.sensor.getType() ==  
    Sensor.TYPE_ACCELEROMETER) {  
    float ax, ay, az;  
    ax = event.values[0];  
    ay = event.values[1];  
    az = event.values[2];  
    GL3JNILib.addAccelData(ax, ay, az);  
}  
/* All values are in radians/second and measure the  
   rate of rotation around the device's local X, Y,  
   and Z axes */  
if (event.sensor.getType() ==  
    Sensor.TYPE_GYROSCOPE) {  
    float gx, gy, gz;  
    //angular speed  
    gx = event.values[0];  
    gy = event.values[1];  
    gz = event.values[2];  
    GL3JNILib.addGyroData(gx, gy, gz);  
}  
//All values are in micro-Tesla (uT) and measure  
//the ambient magnetic field in the X, Y and Z axes.  
if (event.sensor.getType() ==  
    Sensor.TYPE_MAGNETIC_FIELD) {  
    float mx, my, mz;  
    mx = event.values[0];  
    my = event.values[1];  
    mz = event.values[2];  
    GL3JNILib.addMagData(mx, my, mz);  
}  
}
```

Next implement the `GL3JNIView` class, which handles OpenGL rendering, in the `GL3JNIView.java` source file inside the `src/com/android/gl3jni/` directory. Since this implementation is identical to content in the *Chapter 7, An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0*, we will not discuss it again here.

Finally, integrate all the new features in the `GL3JNILib` class, which handles native library loading and calling, in the `GL3JNILib.java` file inside the `src/com/android/gl3jni` directory:

```
package com.android.gl3jni;  
  
public class GL3JNILib {  
    static {
```

```
System.loadLibrary("gl3jni");
}

public static native void init(int width, int height);
public static native void step();

public static native void addAccelData(float ax,
    float ay, float az);
public static native void addGyroData(float gx,
    float gy, float gz);
public static native void addMagData(float mx,
    float my, float mz);
}
```

Now, on the JNI/C++ side, create a class called `Sensor` for managing the data buffer for each sensor, including the accelerometer, gyroscope, and magnetometer (digital compass). First, create a header file for the `Sensor` class called `Sensor.h`:

```
#ifndef SENSOR_H_
#define SENSOR_H_
#include <stdlib.h>
#include <jni.h>
#include <GLES3/gl3.h>
#include <math.h>

class Sensor {
public:
    Sensor();
    Sensor(unsigned int size);
    virtual ~Sensor();

    //Resize buffer size dynamically with this function
    void init(unsigned int size);
    //Append new data to the buffer
    void appendAccelData(GLfloat x, GLfloat y, GLfloat z);
    void appendGyroData(GLfloat x, GLfloat y, GLfloat z);
    void appendMagData(GLfloat x, GLfloat y, GLfloat z);

    //Get sensor data buffer
    GLfloat *getAccelDataPtr(int channel);
    GLfloat *getGyroDataPtr(int channel);
    GLfloat *getMagDataPtr(int channel);
    GLfloat *getAxisPtr();
```

```
//Auto rescale factors based on max and min
GLfloat getAccScale();
GLfloat getGyroScale();
GLfloat getMagScale();

unsigned int getBufferSize();

private:
    unsigned int buffer_size;

    GLfloat **accel_data;
    GLfloat **gyro_data;
    GLfloat **mag_data;
    GLfloat *x_axis;

    GLfloat abs_max_acc;
    GLfloat abs_max_mag;
    GLfloat abs_max_gyro;

    void createBuffers(unsigned int size);
    void free_all();

    void findAbsMax(GLfloat *src, GLfloat *max);
    void appendData(GLfloat *src, GLfloat data);
    void setNormalizedAxis(GLfloat *data,
                           unsigned int size, float min, float max);
};

#endif /* SENSOR_H_ */
```

Then, implement the Sensor class in the `Sensor.cpp` file with the following steps:

1. Implement the constructor and destructor for the Sensor class. Set the default size of the buffer to 256:

```
#include "Sensor.h"
Sensor::Sensor() {
    //use default size
    init(256);
}
// Initialize with different buffer size
Sensor::Sensor(unsigned int size) {
    init(size);
}
Sensor::~Sensor() {
    free_all();
}
```

2. Add the initialization function, which sets all default parameters, and allocate and deallocate memory at runtime:

```
void Sensor::init(unsigned int size){  
    buffer_size = size;  
    //delete the old memory if already exist  
    free_all();  
    //allocate the memory for the buffer  
    createBuffers(size);  
    setNormalizedAxis(x_axis, size, -1.0f, 1.0f);  
    abs_max_acc = 0;  
    abs_max_gyro = 0;  
    abs_max_mag = 0;  
}
```

3. Implement the `createBuffers` function for memory allocation:

```
// Allocate memory for all sensor data buffers  
void Sensor::createBuffers(unsigned int buffer_size){  
    accel_data = (GLfloat**)malloc(3*sizeof(GLfloat*));  
    gyro_data = (GLfloat**)malloc(3*sizeof(GLfloat*));  
    mag_data = (GLfloat**)malloc(3*sizeof(GLfloat*));  
  
    //3 channels for accelerometer  
    accel_data[0] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    accel_data[1] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    accel_data[2] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
  
    //3 channels for gyroscope  
    gyro_data[0] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    gyro_data[1] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    gyro_data[2] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
  
    //3 channels for digital compass  
    mag_data[0] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    mag_data[1] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
    mag_data[2] =  
        (GLfloat*)calloc(buffer_size,sizeof(GLfloat));
```

```
//x-axis precomputed  
x_axis = (GLfloat*)calloc(buffer_size,sizeof(GLfloat));  
}
```

4. Implement the `free_all` function for deallocating memory:

```
// Dealloacate all memory  
void Sensor::free_all(){  
    if(accel_data){  
        free(accel_data[0]);  
        free(accel_data[1]);  
        free(accel_data[2]);  
        free(accel_data);  
    }  
    if(gyro_data){  
        free(gyro_data[0]);  
        free(gyro_data[1]);  
        free(gyro_data[2]);  
        free(gyro_data);  
    }  
    if(mag_data){  
        free(mag_data[0]);  
        free(mag_data[1]);  
        free(mag_data[2]);  
        free(mag_data);  
    }  
    if(x_axis){  
        free(x_axis);  
    }  
}
```

5. Create routines for appending data to the data buffer of each sensor:

```
// Append acceleration data to the buffer  
void Sensor::appendAccelData(GLfloat x, GLfloat y, GLfloat z){  
    abs_max_acc = 0;  
    float data[3] = {x, y, z};  
    for(int i=0; i<3; i++){  
        appendData(accel_data[i], data[i]);  
        findAbsMax(accel_data[i], &abs_max_acc);  
    }  
}  
  
// Append the gyroscope data to the buffer  
void Sensor::appendGyroData(GLfloat x, GLfloat y, GLfloat z){
```

```
abs_max_gyro = 0;
float data[3] = {x, y, z};
for(int i=0; i<3; i++){
    appendData(gyro_data[i], data[i]);
    findAbsMax(gyro_data[i], &abs_max_gyro);
}
}

// Append the magnetic field data to the buffer
void Sensor::appendMagData(GLfloat x, GLfloat y, GLfloat z){
    abs_max_mag = 0;
    float data[3] = {x, y, z};
    for(int i=0; i<3; i++){
        appendData(mag_data[i], data[i]);
        findAbsMax(mag_data[i], &abs_max_mag);
    }
}

// Append Data to the end of the buffer
void Sensor::appendData(GLfloat *src, GLfloat data){
    //shift the data by one
    int i;
    for(i=0; i<buffer_size-1; i++){
        src[i]=src[i+1];
    }
    //set the last element with the new data
    src[buffer_size-1]=data;
}
```

6. Create routines for returning the pointer to the memory buffer of each sensor:

```
// Return the x-axis buffer
GLfloat* Sensor::getAxisPtr() {
    return x_axis;
}

// Get the acceleration data buffer
GLfloat* Sensor::getAccelDataPtr(int channel) {
    return accel_data[channel];
}

// Get the Gyroscope data buffer
GLfloat* Sensor::getGyroDataPtr(int channel) {
    return gyro_data[channel];
}
```

```
// Get the Magnetic field data buffer
GLfloat* Sensor::getMagDataPtr(int channel) {
    return mag_data[channel];
}

7. Implement methods for displaying/plotting the data stream properly from each
sensor (for example, determining the maximum value of the data stream from
each sensor to scale the data properly):

// Return buffer size
unsigned int Sensor::getBufferSize() {
    return buffer_size;
}

/* Return the global max for the acceleration data
   buffer (for rescaling and fitting purpose) */
GLfloat Sensor::getAccScale() {
    return abs_max_acc;
}

/* Return the global max for the gyroscope data
   buffer (for rescaling and fitting purpose) */
GLfloat Sensor::getGyroScale() {
    return abs_max_gyro;
}

/* Return the global max for the magnetic field data
   buffer (for rescaling and fitting purpose) */
GLfloat Sensor::getMagScale() {
    return abs_max_mag;
}

// Pre-compute the x-axis for the plot
void Sensor::setNormalizedAxis(GLfloat *data,
    unsigned int size, float min, float max){
    float step_size = (max - min)/(float)size;
    for(int i=0; i<size; i++){
        data[i]=min+step_size*i;
    }
}

// Find the absolute maximum from the buffer
void Sensor::findAbsMax(GLfloat *src, GLfloat *max) {
    int i=0;
    for(i=0; i<buffer_size; i++) {
        if(*max < fabs(src[i])){
```

```
    *max= fabs(src[i]);  
}  
}  
}
```

Finally, we describe the implementation of the OpenGL ES 3.0 native code to complete the demo application (`main_sensor.cpp`). The code is built upon the structure introduced in the previous chapter, so only new changes and modifications will be described in the following steps:

1. In the project directory, create a file named `main_sensor.cpp` and store it inside the `jni` directory.
2. Include all necessary header files, including `sensor.h` at the beginning of the file:

```
#include <Sensor.h>  
...
```

3. Declare shader program handlers and variables for handling sensor data:

```
GLuint gProgram;  
GLuint gxPositionHandle;  
GLuint gyPositionHandle;  
GLuint gColorHandle;  
GLuint gOffsetHandle;  
GLuint gScaleHandle;  
static Sensor g_sensor_data;
```

4. Define the shader program code for both the vertex shader and fragment shader to render points and lines:

```
// Vertex shader source code  
static const char g_vshader_code[] =  
    "#version 300 es\n"  
    "in float yPosition;\n"  
    "in float xPosition;\n"  
    "uniform float scale;\n"  
    "uniform float offset;\n"  
    "void main() {\n"  
        "    vec4 position = vec4(xPosition,  
        "        yPosition*scale+offset, 0.0, 1.0);\n"  
        "    gl_Position = position;\n"  
    }\n";
```

```
// fragment shader source code  
static const char g_fshader_code[] =  
    "#version 300 es\n"  
    "precision mediump float;\n"  
    "uniform vec4 color;\n"
```

```
"out vec4 color_out;\n"
"void main() {\n"
    "    color_out = color;\n"
} \n";
```

5. Set up all attribute variables in the `setupGraphics` function. These variables will be used to communicate with the shader programs:

```
bool setupGraphics(int w, int h) {

    ...

    gxPositionHandle = glGetUniformLocation(gProgram,
        "xPosition");
    checkGLError("glGetAttribLocation");
    LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
        gxPositionHandle);

    gyPositionHandle = glGetUniformLocation(gProgram, "yPosition");
    checkGLError("glGetAttribLocation");
    LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
        gyPositionHandle);

    gColorHandle = glGetUniformLocation(gProgram,
        "color");
    checkGLError("glGetUniformLocation");
    LOGI("glGetUniformLocation(\"color\") = %d\n",
        gColorHandle);

    gOffsetHandle = glGetUniformLocation(gProgram,
        "offset");
    checkGLError("glGetUniformLocation");
    LOGI("glGetUniformLocation(\"offset\") = %d\n",
        gOffsetHandle);

    gScaleHandle = glGetUniformLocation(gProgram,
        "scale");
    checkGLError("glGetUniformLocation");
    LOGI("glGetUniformLocation(\"scale\") = %d\n",
        gScaleHandle);

    glViewport(0, 0, w, h);
    width = w;
    height = h;
```

```
    checkGlError("glViewport");

    return true;
}
```

6. Create a function for drawing 2D plots to display real-time sensor data:

```
void draw2DPlot(GLfloat *data, unsigned int size, GLfloat scale,
GLfloat offset){
    glVertexAttribPointer(gyPositionHandle, 1, GL_FLOAT,
    GL_FALSE, 0, data);
    checkGlError("glVertexAttribPointer");

    glEnableVertexAttribArray(gyPositionHandle);
    checkGlError("glEnableVertexAttribArray");

    glUniform1f(gOffsetHandle, offset);
    checkGlError("glUniform1f");

    glUniform1f(gScaleHandle, scale);
    checkGlError("glUniform1f");

    glDrawArrays(GL_LINE_STRIP, 0,
    g_sensor_data.getBufferSize());
    checkGlError("glDrawArrays");
}
```

7. Set up the rendering function which draws the various 2D time series with the data stream from the sensors:

```
void renderFrame() {
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    checkGlError("glClearColor");

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    checkGlError("glClear");

    glUseProgram(gProgram);
    checkGlError("glUseProgram");

    glVertexAttribPointer(gxPositionHandle, 1, GL_FLOAT,
    GL_FALSE, 0, g_sensor_data.getAxisPtr());
    checkGlError("glVertexAttribPointer");

    glEnableVertexAttribArray(gxPositionHandle);
    checkGlError("glEnableVertexAttribArray");
```

```
//Obtain the scaling factor based on the dataset
//0.33f for 1/3 of the screen for each graph
float acc_scale = 0.33f/g_sensor_data.getAccScale();
float gyro_scale =
    0.33f/g_sensor_data.getGyroScale();
float mag_scale = 0.33f/g_sensor_data.getMagScale();

glLineWidth(4.0f);

//set the rendering color
glUniform4f(gColorHandle, 1.0f, 0.0f, 0.0f, 1.0f);
checkGlError("glUniform1f");
/* Render the accelerometer, gyro, and digital compass data.
As the vertex shader does not use any projection matrix, every
visible vertex has to be in the range of [-1, 1]. 0.67f, 0.0f,
and -0.67f define the vertical positions of each graph */
draw2DPlot(g_sensor_data.getAccelDataPtr(0),
    g_sensor_data.getBufferSize(), acc_scale, 0.67f);
draw2DPlot(g_sensor_data.getGyroDataPtr(0),
    g_sensor_data.getBufferSize(), gyro_scale, 0.0f);
draw2DPlot(g_sensor_data.getMagDataPtr(0),
    g_sensor_data.getBufferSize(), mag_scale, -0.67f);

glUniform4f(gColorHandle, 0.0f, 1.0f, 0.0f, 1.0f);
checkGlError("glUniform1f");
draw2DPlot(g_sensor_data.getAccelDataPtr(1),
    g_sensor_data.getBufferSize(), acc_scale, 0.67f);
draw2DPlot(g_sensor_data.getGyroDataPtr(1),
    g_sensor_data.getBufferSize(), gyro_scale, 0.0f);
draw2DPlot(g_sensor_data.getMagDataPtr(1),
    g_sensor_data.getBufferSize(), mag_scale, -0.67f);

glUniform4f(gColorHandle, 0.0f, 0.0f, 1.0f, 1.0f);
checkGlError("glUniform1f");
draw2DPlot(g_sensor_data.getAccelDataPtr(2),
    g_sensor_data.getBufferSize(), acc_scale, 0.67f);
draw2DPlot(g_sensor_data.getGyroDataPtr(2),
    g_sensor_data.getBufferSize(), gyro_scale, 0.0f);
draw2DPlot(g_sensor_data.getMagDataPtr(2),
    g_sensor_data.getBufferSize(), mag_scale, -0.67f);
}
```

8. Define the JNI prototypes that connect to the Java side. These calls are the interfaces for communicating between the Java code and C/C++ native code:

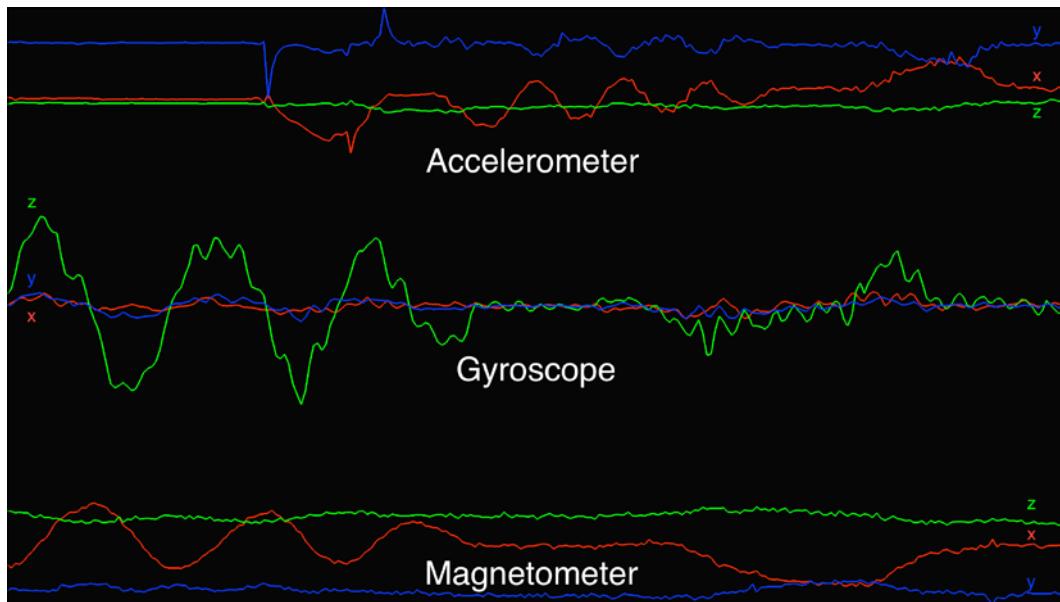
```
//external calls for Java
extern "C" {
    JNIEEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_init(JNIEnv *env, jobject obj, jint width, jint height);
    JNIEEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_step(JNIEnv *env, jobject obj);
    JNIEEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_addAccelData
    (JNIEnv * env, jobject obj, jfloat ax, jfloat ay, jfloat az);
    JNIEEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_addGyroData
    (JNIEnv * env, jobject obj, jfloat gx, jfloat gy, jfloat gz);
    JNIEEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_addMagData
    (JNIEnv * env, jobject obj, jfloat mx, jfloat my, jfloat mz)
    {
        g_sensor_data.appendMagData(mx, my, mz);
    }
};

//link to internal calls
JNIEEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_
init(JNIEnv * env, jobject obj, jint width, jint height)
{
    setupGraphics(width, height);
}
JNIEEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_
step(JNIEnv * env, jobject obj)
{
    renderFrame();
}
JNIEEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_
addAccelData(JNIEnv * env, jobject obj, jfloat ax, jfloat ay,
jfloat az){
    g_sensor_data.appendAccelData(ax, ay, az);
}
JNIEEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_
addGyroData(JNIEnv * env, jobject obj, jfloat gx, jfloat gy,
jfloat gz){
    g_sensor_data.appendGyroData(gx, gy, gz);
}
```

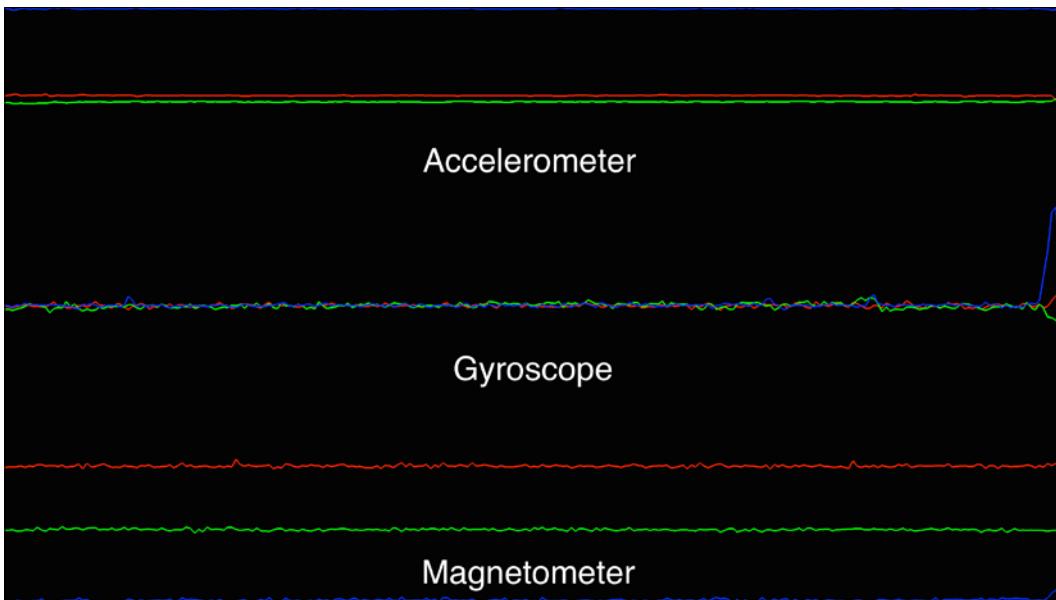
```
JNIEXPORT void JNICALL Java_com_android_gl3jni_GL3JNILib_
addMagData(JNIEnv * env, jobject obj, jfloat mx, jfloat my,
jfloat mz) {
    g_sensor_data.appendMagData(mx, my, mz);
}
```

Finally, we need to compile and install the Android application with the same instructions as outlined in the previous chapter.

The following screenshots show the real-time sensor data stream from the accelerometer, gyroscope, and digital compass (top panel, middle panel, and bottom panel, respectively) on our Android device. Red, green, and blue are used to differentiate the channels from each sensor data stream. For example, the red plot in the top panel represents the acceleration value of the device along the x axis (the blue plot for the y axis and the green plot for the z axis). In the first example, we rotated the phone freely at various orientations and the plots show the corresponding changes in the sensor values. The visualizer also provides an auto-scale function, which automatically computes the maximum values to rescale the plots accordingly:



Next, we positioned the phone on a stationary surface and we plotted the values of the sensors. Instead of observing constant values over time, the time series plots show that there are some very small changes (jittering) in the sensor values due to sensor noise. Depending on the application, you will often need to apply filtering techniques to ensure that the user experience is jitter-free. One simple solution is to apply a low-pass filter to smooth out any high-frequency noise. More details on the implementation of such filters can be found at http://developer.android.com/guide/topics/sensors/sensors_motion.html.



How it works...

The Android Sensor Framework allows users to access the raw data from various types of sensors on a mobile device. This framework is part of the `android.hardware` package and the `Sensor` package includes a set of classes and interfaces for sensor-specific features.

The `SensorManager` class provides an interface and methods for accessing and listing the available sensors from the device. Some common hardware sensors include the accelerometer, gyroscope, proximity sensor, and the magnetometer (digital compass). These sensors are represented by constant variables (such as `TYPE_ACCELEROMETER` for the accelerometer, `TYPE_MAGNETIC_FIELD` for the magnetometer, and `TYPE_GYROSCOPE` for the gyroscope) and the `getDefaultSensor` function returns an instance of the `Sensor` object based on the type requested.

To enable data streaming, we must register the sensor to the `SensorEventListener` class such that the raw data is reported back to the application upon updates. The `registerListener` function then creates the callback to handle updates to the sensor value or sensor accuracy. The `SensorEvent` variable stores the name of the sensor, the timestamp and accuracy of the event, as well as the raw data.

The raw data stream from each sensor is reported back with the `onSensorChange` function. Since sensor data may be acquired and streamed at a high rate, it is important that we do not block callback function calls or perform any computationally intensive processes within the `onSensorChange` function. In addition, it is a good practice to reduce the data rate of the sensor based on your application requirements. In our case, we set the sensor to run at the optimal rate for gaming purposes by passing the constant preset variable `SENSOR_DELAY_GAME` to the `registerListener` function.

The `GL3JNILib` class then handles all the data passing to the native code using the new functions. For simplicity, we have created separate functions for each sensor type, which makes it easier for the reader to understand the data flow for each sensor.

At this point, we have created the interfaces that redirect data to the native side. However, to plot the sensor data on the screen, we need to create a simple buffering mechanism that stores the data points over some period of time. We have created a custom `Sensor` class in C++ to handle data creation, updates, and processing needed to manage these interactions. The implementation of the class is straightforward, and we preset the buffer size to store 256 data points by default.

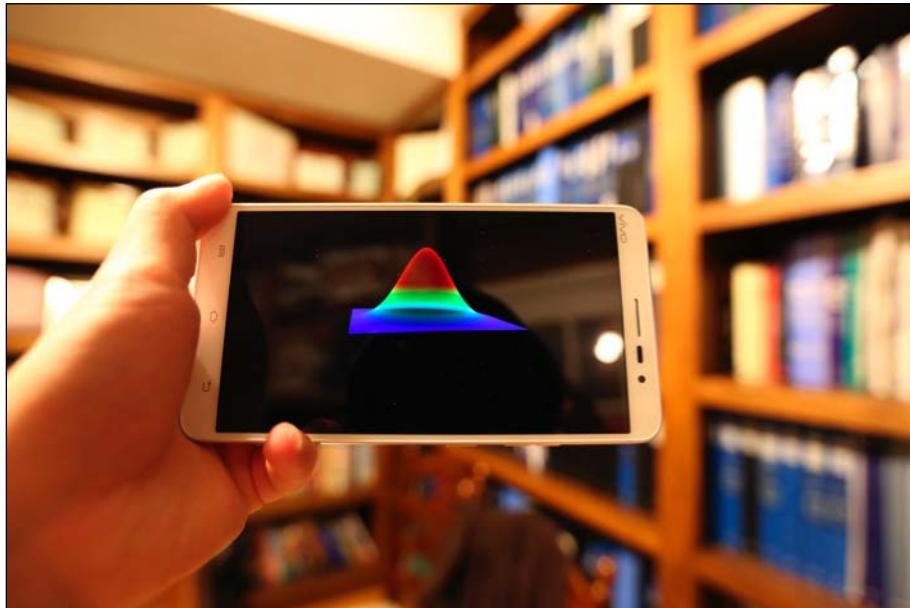
On the OpenGL ES side, we create the 2D plot by appending the data stream to our vertex buffer. The scale of the data stream is adjusted dynamically based on the current values to ensure that the values fit on the screen. Notice that we have also performed all data scaling and translation on the vertex shader to reduce any overhead in the CPU computation.

See also

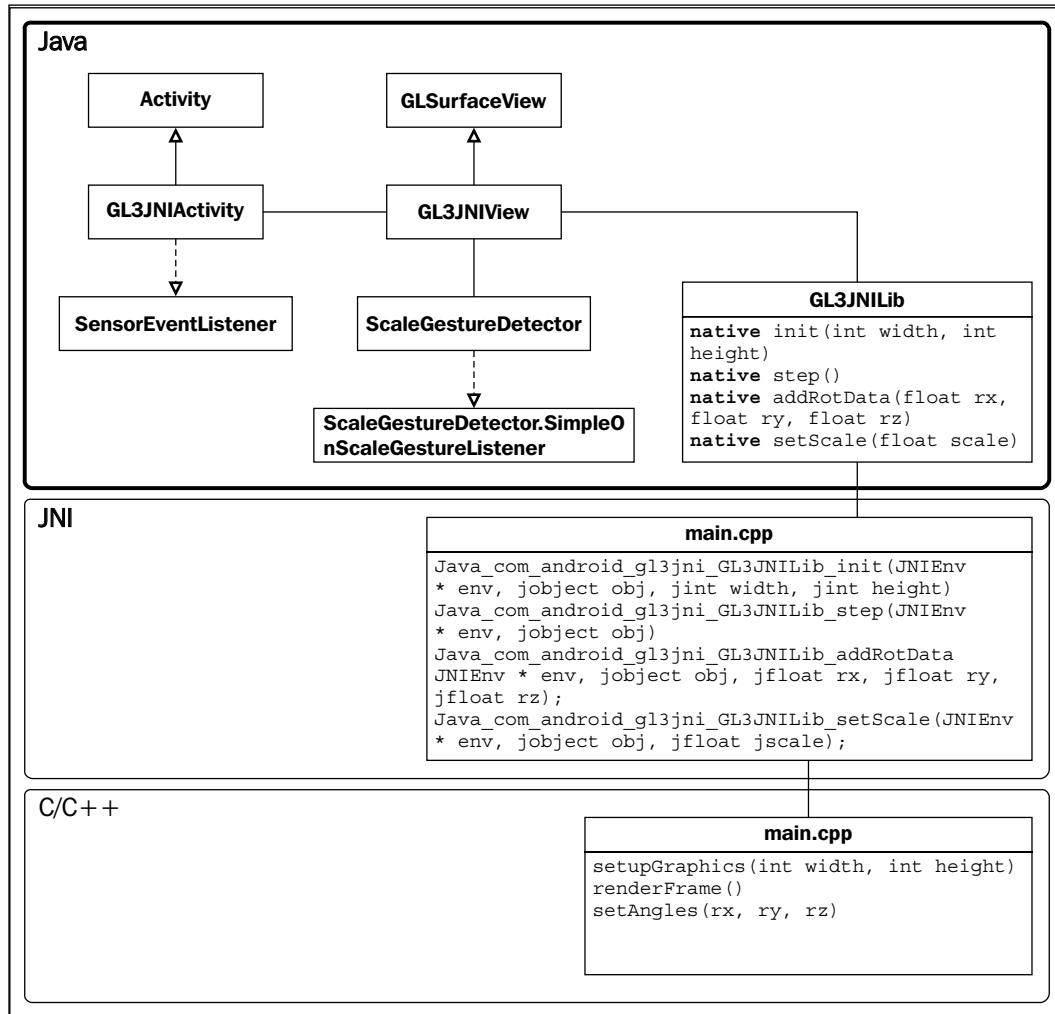
- ▶ For more information on the Android Sensor Framework, consult the documentation online at http://developer.android.com/guide/topics/sensors/sensors_overview.html.

Part I – handling multi-touch interface and motion sensor inputs

Now that we have introduced the basics of handling sensor inputs, we will develop an interactive, sensor-based data visualization tool. In addition to using motion sensors, we will introduce a multi-touch interface for user interaction. The following is a preview of the final application, integrating all the elements in this chapter:



In this section, we will focus solely on the Java side of the implementation and the native code will be described in part II. The following class diagram illustrates the various components of the Java code (part I) that provide the basic interface for user interaction on the mobile device and demonstrates how the native code (part II) completes the entire implementation:



How to do it...

First, we will create the core Java source files that are essential to an Android application. These files serve as a wrapper for our OpenGL ES 3.0 native code. The code structure is based on the `gl3jni` package described in the previous section. Here we will highlight the major changes made to the code and discuss the interaction of these new components.

In the project directory, modify the `GL3JNIActivity` class in the `GL3JNIActivity.java` file within the `src/com/android/gl3jni` directory. Instead of using the raw sensor data, we will utilize the Android sensor fusion algorithm, which intelligently combines all sensor data to recover the orientation of the device as a rotation vector. The steps to enable this feature are described as follows:

1. In the `GL3JNIActivity` class, add the new variables for handling the rotation matrix and vector:

```
public class GL3JNIActivity extends Activity implements  
SensorEventListener{  
    GL3JNIView mView;  
    private SensorManager mSensorManager;  
    private Sensor mRotate;  
    private float[] mRotationMatrix=new float[16];  
    private float[] orientationVals=new float[3];
```

2. Initialize the `Sensor` variable with the `TYPE_ROTATION_VECTOR` type, which returns the device orientation as a rotation vector/matrix:

```
@Override protected void onCreate(Bundle icicle) {  
    super.onCreate(icicle);  
    //lock the screen orientation for this demo  
    //otherwise the canvas will rotate  
    setRequestedOrientation  
        (ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);  
  
    mSensorManager = (SensorManager) getSystemService  
        (SENSOR_SERVICE);  
    //TYPE_ROTATION_VECTOR for device orientation  
    mRotate = mSensorManager.getDefaultSensor  
        (Sensor.TYPE_ROTATION_VECTOR);  
  
    mView = new GL3JNIView(getApplicationContext());  
    setContentView(mView);  
}
```

3. Register the `Sensor Manager` object and set the sensor response rate to `SENSOR_DELAY_GAME`, which is used for gaming or real-time applications:

```
@Override protected void onResume() {  
    super.onResume();  
    mView.onResume();  
    mSensorManager.registerListener(this, mRotate,  
        SensorManager.SENSOR_DELAY_GAME);  
}
```

4. Retrieve the device orientation and save the event data as a rotation matrix. Then convert the rotation matrix into Euler angles that are passed to the native code:

```
@Override  
public void onSensorChanged(SensorEvent event) {  
    if (event.sensor.getType() ==  
        Sensor.TYPE_ROTATION_VECTOR) {  
        SensorManager.getRotationMatrixFromVector  
            (mRotationMatrix, event.values);  
        SensorManager.getOrientation (mRotationMatrix,  
            orientationVals);  
        GL3JNILib.addRotData(orientationVals[0],  
            orientationVals[1],orientationVals[2]);  
    }  
}
```

Next, modify the `GL3JNIView` class, which handles OpenGL rendering, in the `GL3JNIView.java` file inside the `src/com/android/gl3jni/` directory. To make the application interactive, we also integrate the touch-based gesture detector that handles multi-touch events. Particularly, we add the `ScaleGestureDetector` class that enables the pinch gesture for scaling the 3D plot. To implement this feature, we make the following modifications to the `GL3JNIView.java` file:

1. Import the `MotionEvent` and `ScaleGestureDetector` classes:

```
package com.android.gl3jni;  
...  
import android.view.MotionEvent;  
import android.view.ScaleGestureDetector;  
...
```

2. Create a `ScaleGestureDetector` variable and initialize with `ScaleListener`:

```
class GL3JNIView extends GLSurfaceView {  
    private ScaleGestureDetector mScaleDetector;  
    ...  
  
    public GL3JNIView(Context context) {  
        super(context);  
        ...  
        //handle gesture input  
        mScaleDetector = new ScaleGestureDetector  
            (context, new ScaleListener());  
    }
```

3. Pass the motion event to the gesture detector when a touch screen event occurs (`onTouchEvent`):

```
@Override  
public boolean onTouchEvent(MotionEvent ev) {  
    // Let ScaleGestureDetector inspect all events.  
    mScaleDetector.onTouchEvent(ev);  
    return true;  
}
```

4. Implement `SimpleOnScaleGestureListener` and handle the callback (`onScale`) on pinch gesture events:

```
private class ScaleListener extends  
    ScaleGestureDetector.SimpleOnScaleGestureListener {  
    private float mScaleFactor = 1.f;  
    @Override  
    public boolean onScale(ScaleGestureDetector  
        detector)  
    {  
        //scaling factor  
        mScaleFactor *= detector.getScaleFactor();  
        //Don't let the object get too small/too large.  
        mScaleFactor = Math.max(0.1f,  
            Math.min(mScaleFactor, 5.0f));  
        invalidate();  
        GL3JNILib.setScale(mScaleFactor);  
        return true;  
    }  
}
```

Finally, in the `GL3JNILib` class, we implement the functions to handle native library loading and calling in the `GL3JNILib.java` file inside the `src/com/android/gl3jni` directory:

```
package com.android.gl3jni;  
  
public class GL3JNILib {  
    static {  
        System.loadLibrary("gl3jni");  
    }  
  
    public static native void init(int width, int height);  
    public static native void step();
```

```
/* pass the rotation angles and scaling factor to the
native code */
public static native void addRotData(float rx, float
ry, float rz);
public static native void setScale(float scale);
}
```

How it works...

Similar to the previous demo, we will use the Android Sensor Framework to handle the sensor inputs. Notice that, in this demo, we specify `TYPE_ROTATION_VECTOR` for the sensor type inside the `getDefaultSensor` function in `GL3JNIAActivity.java`, which allows us to detect the device orientation. This is a software type sensor in which all IMUs data (from the accelerometer, gyroscope, and magnetometer) are fused together to create the rotation vector. The device orientation data is first stored in the rotation matrix `mRotationMatrix` using the `getRotationMatrixFromVector` function and the azimuth, pitch, and roll angles (rotation around the x, y, and z axes, respectively) are retrieved using the `getOrientation` function. Finally, we pass the three orientation angles to the native code portion of the implementation using the `GL3JNILib.addRotData` call. This allows us to control 3D graphics based on the device's orientation.

Next we will explain how the multi-touch interface works. Inside the `GL3JNIView` class, you will notice that we have created an instance (`mScaleDetector`) of a new class called `ScaleGestureDetector`. The `ScaleGestureDetector` class detects scaling transformation gestures (pinching with two fingers) using the `MotionEvent` class from the multi-touch screen. The algorithm returns the scale factor that can be redirected to the OpenGL pipeline to update the graphics in real time. The `SimpleOnScaleGestureListener` class provides a callback function for the `onScale` event and we pass the scale factor (`mScaleFactor`) to the native code using the `GL3JNILib.setScale` call.

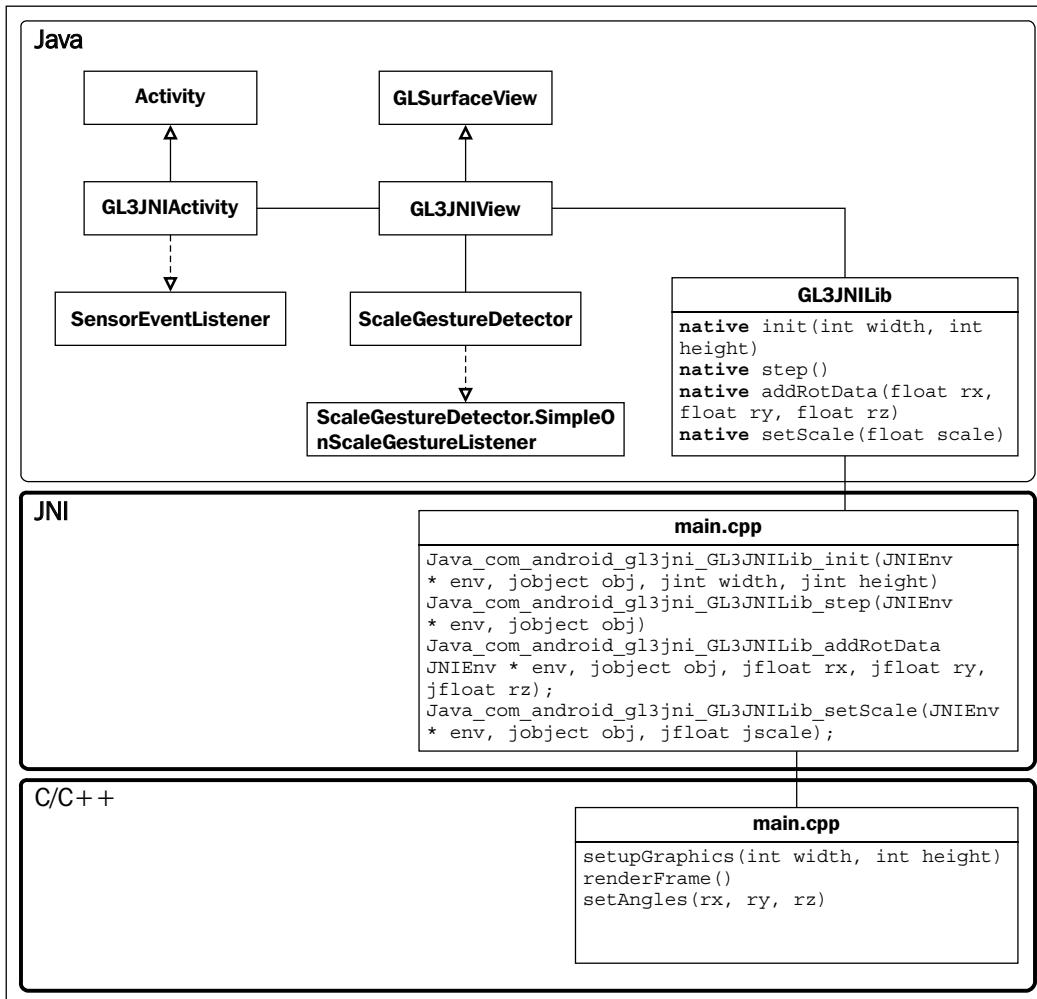
See also

- ▶ For further information on the Android multi-touch interface, see the detailed documentation at <http://developer.android.com/training/gestures/index.html>.

Part II – interactive, real-time data visualization with mobile GPUs

Now we will complete our demo with the native code implementation to create our highly interactive Android-based data visualization application with OpenGL ES 3.0 as well as the Android sensor and gesture control interface.

The following class diagram highlights what remains to be implemented on the C/C++ side:



How to do it...

Here, we describe the implementation of the OpenGL ES 3.0 native code to complete the demo application. We will preserve the same code structure from *Chapter 7, An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0*. In the following steps, only the new codes are highlighted, and all changes are implemented in the `main.cpp` file inside the `jni` folder:

1. Include all necessary header files, including `JNI`, OpenGL ES 3.0, and the `GLM` library:

```
#define GLM_FORCE_RADIANS

//header for JNI
#include <jni.h>
...

//header for GLM library
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

2. Declare the shader program variables:

```
//shader program handlers
GLuint gProgram;
GLuint gvPositionHandle;
GLuint matrixHandle;
GLuint sigmaHandle;
GLuint scaleHandle;
```

3. Declare variables for setting up the camera as well as other relevant variables such as the rotation angles and grid:

```
//the view matrix and projection matrix
glm::mat4 g_view_matrix;
glm::mat4 g_projection_matrix;

//initial position of the camera
glm::vec3 g_position = glm::vec3( 0, 0, 4 );

//FOV of the camera
float g_initial_fov = glm::pi<float>()*0.25f;
//rotation angles, set by sensors or by touch screen
float rx, ry, rz;
float scale=1.0f;
//vertices for the grid
const unsigned int GRID_SIZE=400;
GLfloat gGrid[GRID_SIZE*GRID_SIZE*3]={ 0 };
```

4. Define the shader program code for both the vertex shader and fragment shader. Note the similarity in the heat map generation code between this implementation in OpenGL ES 3.0 and an earlier implementation in standard OpenGL (see chapters 4-6):

```
// Vertex shader source code
static const char g_vshader_code[] =
    "#version 300 es\n"
    "in vec4 vPosition;\n"
    "uniform mat4 MVP;\n"
    "uniform float sigma;\n"
    "uniform float scale;\n"
    "out vec4 color_based_on_position;\n"
    "/* Heat map generator */\n"
    "vec4 heatMap(float v, float vmin, float vmax) {\n"
    "    float dv;\n"
    "    float r=1.0, g=1.0, b=1.0;\n"
    "    if (v < vmin){\n"
    "        v = vmin;\n"
    "    }\n    if (v > vmax){\n"
    "        v = vmax;\n"
    "    }\n    dv = vmax - vmin;\n"
    "    if (v < (vmin + 0.25 * dv)) {\n"
    "        r = 0.0;\n"
    "        g = 4.0 * (v - vmin) / dv;\n"
    "    } else if (v < (vmin + 0.5 * dv)) {\n"
    "        r = 0.0;\n"
    "        b = 1.0 + 4.0 * (vmin + 0.25 * dv - v) / dv;\n"
    "    } else if (v < (vmin + 0.75 * dv)) {\n"
    "        r = 4.0 * (v - vmin - 0.5 * dv) / dv;\n"
    "        b = 0.0;\n"
    "    } else {\n"
    "        g = 1.0 + 4.0 * (vmin + 0.75 * dv - v) / dv;\n"
    "        b = 0.0;\n"
    "    }\n    return vec4(r, g, b, 0.1);\n"
    "}\nvoid main() {\n    /*Simulation on GPU */\n    float x_data = vPosition.x;\n    float y_data = vPosition.y;\n    float sigma2 = sigma*sigma;\n    float z = exp(-0.5*(x_data*x_data)/(sigma2)\n        -0.5*(y_data*y_data)/(sigma2));\n    vec4 position = vPosition;\n}
```

```
// scale the graphics based on user gesture input
" position.z = z*scale;\n"
" position.x = position.x*scale;\n"
" position.y = position.y*scale;\n"
" gl_Position = MVP*position;\n"
" color_based_on_position = heatMap(position.z, 0.0, 0.5);\n"
" gl_PointSize = 5.0*scale;\n"
"}\n";

// fragment shader source code
static const char g_fshader_code[] =
"#version 300 es\n"
"precision mediump float;\n"
"in vec4 color_based_on_position;\n"
"out vec4 color;\n"
"void main() {\n"
" color = color_based_on_position;\n"
"}\n";
```

5. Initialize the grid pattern for data visualization:

```
void computeGrid(){
    float grid_x = GRID_SIZE;
    float grid_y = GRID_SIZE;
    unsigned int data_counter = 0;
    //define a grid ranging from -1 to +1
    for(float x = -grid_x/2.0f; x<grid_x/2.0f; x+=1.0f) {
        for(float y = -grid_y/2.0f; y<grid_y/2.0f; y+=1.0f) {
            float x_data = 2.0f*x/grid_x;
            float y_data = 2.0f*y/grid_y;
            gGrid[data_counter] = x_data;
            gGrid[data_counter+1] = y_data;
            gGrid[data_counter+2] = 0;
            data_counter+=3;
        }
    }
}
```

6. Set the rotation angles that are used to control the model viewing angles.
These angles (device orientation) are passed from the Java side:

```
void setAngles(float irx, float iry, float irz){
    rx = irx;
    ry = iry;
    rz = irz;
}
```

7. Compute the projection and view matrices based on camera parameters:

```
void computeProjectionMatrices() {
    //direction vector for z
    glm::vec3 direction_z(0, 0, -1.0);
    //up vector
    glm::vec3 up = glm::vec3(0,-1,0);

    float aspect_ratio = (float)width/(float)height;
    float nearZ = 0.1f;
    float farZ = 100.0f;
    float top = tan(g_initial_fov/2*nearZ);
    float right = aspect_ratio*top;
    float left = -right;
    float bottom = -top;
    g_projection_matrix = glm::frustum(left, right,
                                         bottom, top, nearZ, farZ);

    // update the view matrix
    g_view_matrix = glm::lookAt(
        g_position,           // camera position
        g_position+direction_z, // view direction
        up                   // up direction
    );
}
```

8. Create a function for handling the initialization of all attribute variables for the shader program and other one-time setups, such as the memory allocation and initialization for the grid:

```
bool setupGraphics(int w, int h) {
    ...
    gvPositionHandle = glGetUniformLocation(gProgram,
                                           "vPosition");
    checkGlError("glGetAttribLocation");
    LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
         gvPositionHandle);

    matrixHandle = glGetUniformLocation(gProgram, "MVP");
    checkGlError("glGetUniformLocation");
    LOGI("glGetUniformLocation(\"MVP\") = %d\n",
         matrixHandle);

    sigmaHandle = glGetUniformLocation(gProgram, "sigma");
    checkGlError("glGetUniformLocation");
```

```
LOGI("glGetUniformLocation(\"sigma\") = %d\n",
     sigmaHandle);

scaleHandle = glGetUniformLocation(gProgram,
    "scale");
checkGlError("glGetUniformLocation");
LOGI("glGetUniformLocation(\"scale\") = %d\n",
     scaleHandle);

...

computeGrid();
return true;
}
```

9. Set up the rendering function for the 3D plot of the Gaussian function:

```
void renderFrame() {
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    static float sigma;

    //update the variables for animations
    sigma+=0.002f;
    if(sigma>0.5f){
        sigma = 0.002f;
    }

    /* gets the View and Model Matrix and apply to the
       rendering */
    computeProjectionMatrices();
    glm::mat4 projection_matrix = g_projection_matrix;
    glm::mat4 view_matrix = g_view_matrix;
    glm::mat4 model_matrix = glm::mat4(1.0);
    model_matrix = glm::rotate(model_matrix, rz,
        glm::vec3(-1.0f, 0.0f, 0.0f));
    model_matrix = glm::rotate(model_matrix, ry,
        glm::vec3(0.0f, -1.0f, 0.0f));
    model_matrix = glm::rotate(model_matrix, rx,
        glm::vec3(0.0f, 0.0f, 1.0f));
    glm::mat4 mvp = projection_matrix * view_matrix *
        model_matrix;
```

```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
checkGlError("glClearColor");

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
checkGlError("glClear");

glUseProgram(gProgram);
checkGlError("glUseProgram");

glUniformMatrix4fv(matrixHandle, 1, GL_FALSE,
&mvp[0][0]);
checkGlError("glUniformMatrix4fv");

glUniform1f(sigmaHandle, sigma);
checkGlError("glUniform1f");

glUniform1f(scaleHandle, scale);
checkGlError("glUniform1f");

glVertexAttribPointer(gvPositionHandle, 3,
GL_FLOAT, GL_FALSE, 0, gGrid);
checkGlError("glVertexAttribPointer");

 glEnableVertexAttribArray(gvPositionHandle);
checkGlError("	glEnableVertexAttribArray");

 glDrawArrays(GL_POINTS, 0, GRID_SIZE*GRID_SIZE);
checkGlError("glDrawArrays");
}
```

10. Define the JNI prototypes that connect to the Java side. These calls are the interfaces for communicating between the Java code and C/C++ native code:

```
extern "C" {
    JNIEXPORT void JNICALL
        Java_com_android_gl3jni_GL3JNILib_init(JNIEnv*
        * env, jobject obj, jint width, jint height);
    JNIEXPORT void JNICALL
        Java_com_android_gl3jni_GL3JNILib_step(JNIEnv*
        * env, jobject obj);
    JNIEXPORT void JNICALL
        Java_com_android_gl3jni_GL3JNILib_addRotData(JNIEnv*
        * env, jobject obj, jfloat rx,
        jfloat ry, jfloat rz);
```

```
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setScale(JNIEnv
* env, jobject obj, jfloat jscale);
};
```

11. Set up the internal function calls with the helper functions:

```
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_init(JNIEnv
* env, jobject obj, jint width, jint height)
{
    setupGraphics(width, height);
}
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_step(JNIEnv
* env, jobject obj)
{
    renderFrame();
}
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_addRotData(JNIEnv
* env, jobject obj, jfloat rx, jfloat ry, jfloat rz)
{
    setAngles(rx, ry, rz);
}
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setScale(JNIEnv
* env, jobject obj, jfloat jscale)
{
    scale = jscale;
    LOGI("Scale is %lf", scale);
}
```

Finally, in terms of the compilation steps, modify the build files `Android.mk` and `Application.mk` accordingly as follows:

1. Add in the GLM path to the `LOCAL_C_INCLUDES` variable in `Android.mk`:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE      := libgl3jni
```

```
LOCAL_CFLAGS      := -Werror
LOCAL_SRC_FILES  := main.cpp
LOCAL_LDLIBS      := -llog -lGLESv3
#The GLM library is installed in one of these two folders by
default
LOCAL_C_INCLUDES := /opt/local/include /usr/local/include

include $(BUILD_SHARED_LIBRARY)
```

2. Add in `gnustl_static` to the `APP_STL` variable to use GNU STL as a static library. This allows for all runtime supports from C++, which is needed by the GLM library. See more at <https://developer.android.com/ndk/guides/cpp-support.html>:

```
APP_ABI := armeabi-v7a
#required for GLM and other static libraries
APP_STL := gnustl_static
```

3. Run the compilation script (this is similar to what we did in the previous chapter). Please note that the `ANDROID_SDK_PATH` and `ANDROID_NDK_PATH` variables should be changed to the correct directories based on the local environment setup:

```
#!/bin/bash

$ANDROID_SDK_PATH="../../../../3rd_party/android/android-sdk-macosx"
$ANDROID_NDK_PATH="../../../../3rd_party/android/android-ndk-r10e"

$ANDROID_SDK_PATH/tools/android update project -p . -s --target
"android-18"
$ANDROID_NDK_PATH/ndk-build
ant debug
```

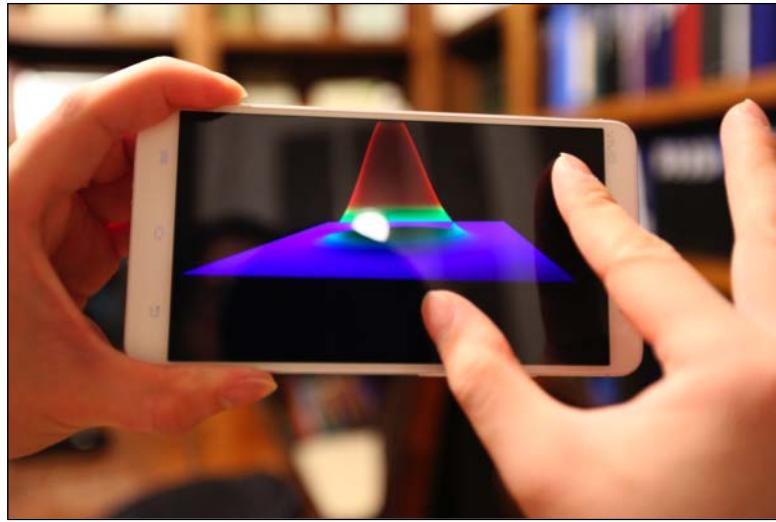
4. Install the **Android Application Package (APK)** on the Android phone, using the following commands in the terminal:

```
ANDROID_SDK_PATH="../../../../3rd_party/android/android-sdk-macosx"
$ANDROID_SDK_PATH/platform-tools/adb install -r bin/
GL3JNIActivity-debug.apk
```

The final results of our implementation are shown next. By changing the orientation of the phone, the Gaussian function can be viewed from different angles. This provides a very intuitive way to visualize 3D datasets. Here is a photo showing the Gaussian function when the device is oriented parallel to the ground:



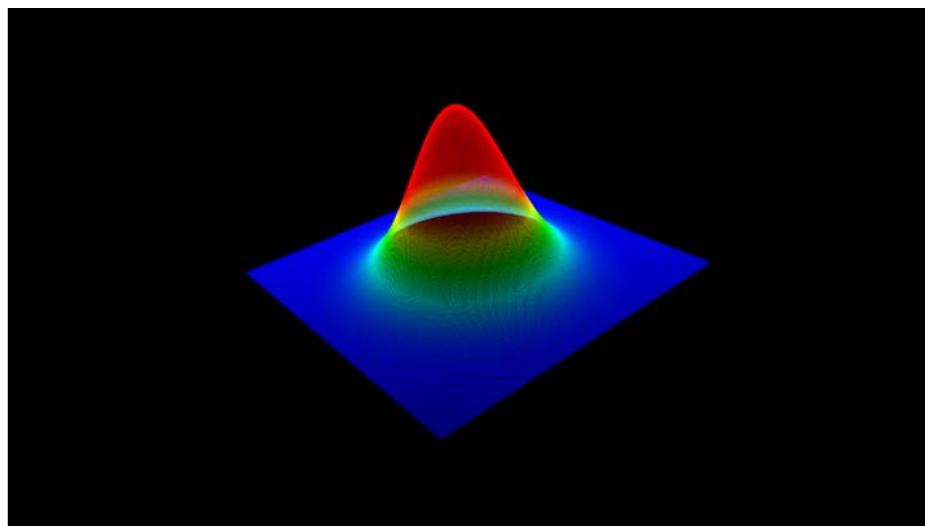
Finally, we test our multi-touch gesture interface by pinching on the touch screen with 2 fingers. This provides an intuitive way to zoom into and out of the 3D data. Here is the first photo that shows the close-up view after zooming into the data:



Here is another photo that shows what the data looks like when you zoom out by pinching your fingers:



Finally, here is a screenshot of the demo application that shows a Gaussian distribution in 3D rendered in real-time with our OpenGL ES 3.0 shader program:



How it works...

In the second part of the demo, we demonstrated the use of a shader program written in OpenGL ES 3.0 to perform all the simulation and heat map-based 3D rendering steps to visualize a Gaussian distribution on a mobile GPU. Importantly, the shader code in OpenGL ES 3.0 is very similar to the code written in standard OpenGL 3.2 and above (see chapters 4 to 6). However, we recommend that you consult the specification to ensure that a particular feature of interest co-exists in both versions. More details on the OpenGL ES 3.0 specifications can be found at https://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.0.pdf.

The hardware-accelerated portion of the code is programmed within the vertex shader program and stored inside the `g_vshader_code` variable; then the fragment shade program passes the processed color information onto the screen's color buffer. The vertex program handles the computation related to the simulation (in our case, we have a Gaussian function with a time-varying sigma value as demonstrated in *Chapter 3, Interactive 3D Data Visualization*) in the graphics hardware. We pass in the sigma value as a uniform variable and it is used to compute the surface height. In addition, we also compute the heat map color value within the shader program based on the height value. With this approach, we have significantly improved the speed of the graphic rendering step by completely eliminating the use of the CPU cycles on these numerous floating point operations.

In addition, we have included the GLM library used in previous chapters into the Android platform by adding the headers as well as the GLM path in the build script `Android.mk`. The GLM library handles the view and projection matrix computation and also allows us to migrate most of our previous work, such as setting up 3D rendering, to the Android platform.

Finally, our Android-based application also utilizes the inputs from the multi-touch screen interface and the device orientation derived from the motion sensor data. These values are passed through the JNI directly to the shader program as uniform variables.

9

Augmented Reality-based Visualization on Mobile or Wearable Platforms

In this chapter, we will cover the following topics:

- ▶ Getting started I: Setting up OpenCV on Android
- ▶ Getting started II: Accessing the camera live feed using OpenCV
- ▶ Displaying real-time video processing with texture mapping
- ▶ Augmented reality-based data visualization over real-world scenes

Introduction

The field of digital graphics has traditionally been living within its own virtual world since computers were invented. Often, computer-generated content has no awareness of the user and how the information is relevant to the user in the real world. The application is always simply waiting for a user command such as the mouse or keyboard input. One major limiting factor in the early design of computer applications is that computers are typically sitting on a desk in an office or in a home environment. The lack of mobility and the inability to interact with its environment or user ultimately limited the development of real-world interactive visualization applications.

Today, with the evolution of mobile computing, we have redefined many of our daily interactions with the world—for example, through applications that enable navigation with GPS using a mobile phone. However, instead of enabling users to seamlessly interact with the world, mobile devices still draw users away from the real world. In particular, as in previous generations of desktop computing, users are still required to look away from the real world into a virtual world (in many cases, just a tiny mobile screen).

The notion of **Augmented Reality (AR)** is a step towards reconnecting the user with the real world through the fusion of the virtual world (generated by the computer) with the real world. This is distinctly different from virtual reality, in which the user is immersed into the virtual world and detached from the real world. For example, a typical embodiment of AR involves the use of a video see-through display in which virtual content (such as a computer-generated map) is combined with a real-world scene (captured continuously with a built-in camera). Now, the user is engaged with the real world—a step closer to a truly human-centric application.

Ultimately, the emergence of AR-enabled wearable computing devices (such as Meta's AR eyeglasses, which features the world's first holographic interface with 3D gesture detection and 3D stereoscopic display) will create a new era of computing that will greatly revolutionize the way humans interact with computers. Developers interested in data visualization now have another set of tools that are significantly more human-centric and intuitive. Such a design, needless to say, truly connects human, machine, and the real world together. Having information directly overlaid onto the real world (for example, by overlaying a virtual guidance map for navigation) is so much more powerful and meaningful.

This final chapter introduces the fundamental building blocks for creating your first AR-based application on a commodity Android-based mobile device: OpenCV for computer vision, OpenGL for graphics rendering, as well as Android's sensor framework for interaction. With these tools, the graphics rendering capability that used to only exist in Hollywood movie production can now be made available at everyone's fingertips. While we will only focus on the use of an Android-based mobile device in this chapter, the conceptual framework for AR-based data visualization introduced in this chapter can be similarly extended to state-of-the-art wearable computing platforms, such as Meta's AR eyeglasses.

Getting started I: Setting up OpenCV on Android

In this section, we will outline the steps to set up the OpenCV library on the Android platform, which is needed to enable access to the live camera stream central to any Augmented Reality applications.

Getting ready

We assume that the Android SDK and NDK are configured exactly as discussed in *Chapter 7, An Introduction to Real-time Graphics Rendering on a Mobile Platform Using OpenGL ES 3.0*. Here, we add in the support of OpenCV for Android. We will import and integrate the OpenCV library into our existing code structure from the previous chapter.

How to do it...

Here, we describe the major steps for setting up the OpenCV library, mainly path setup and pre-configuration of the Java SDK project setup:

1. Download the OpenCV for Android SDK package, Version 3.0.0 (OpenCV-3.0.0-android-sdk-1.zip) at <http://sourceforge.net/projects/opencvlibrary/files/opencv-android/3.0.0/OpenCV-3.0.0-android-sdk-1.zip>.
2. Move the package (OpenCV-3.0.0-android-sdk-1.zip) to the 3rd_party/android folder created in *Chapter 7, An Introduction to Real-time Graphics Rendering on a Mobile Platform Using OpenGL ES 3.0*.
3. Unzip the package with the following commands

```
cd 3rd_party/android && unzip OpenCV-3.0.0-android-sdk-1.zip
```

4. Then in the project folder (for example ch9/code/opencv_demo_1), run the following script to initialize the project for Android. Note that the 3rd_party folder is assumed to be in the same top-level directory as in previous chapters:

```
#!/bin/bash
ANDROID_SDK_PATH="../../../../3rd_party/android/android-sdk-macosx"
OPENCV_SDK_PATH="../../../../3rd_party/android/opencv-android-sdk"

#initialize the SDK Java library
$ANDROID_SDK_PATH/tools/android update project -p $OPENCV_SDK_PATH/sdk/java -s --target "android-18"
$ANDROID_SDK_PATH/tools/android update project -p . -s --target
"android-18" --library $OPENCV_SDK_PATH/sdk/java
```

5. Finally, include the OpenCV path in the build script `jni/Android.mk`.

```
LOCAL_PATH:= $(call my-dir)
#build the OpenGL + OpenCV code in JNI
include $(CLEAR_VARS)
#include OpenCV SDK
include ../../../../3rd_party/android/OpenCV-android-sdk/sdk/native/
jni/OpenCV.mk
```

Now, the project is linked to the OpenCV library, both from the Java side as well as from the native side.

Next we must install the OpenCV Manager on the mobile phone. The OpenCV Manager allows us to create applications without statically linking all the required libraries, and it is recommended. To install the package, we can execute the following `adb` command from the same project folder (`ch9/code/opencv_demo_1`). Again, note the relative location of the `3rd_party` folder. You can also execute this command within the Android SDK folder and modify the relative path of the `3rd_party` folder accordingly.

```
$ANDROID_SDK_PATH/platform-tools/adb install ../../../../../../3rd_party/
android/OpenCV-android-sdk/apk/OpenCV_3.0.0_Manager_3.0.0_armeabi-v7a.
apk
```

After we have successfully completed the setup, we are ready to create our first OpenCV Android application on the phone.

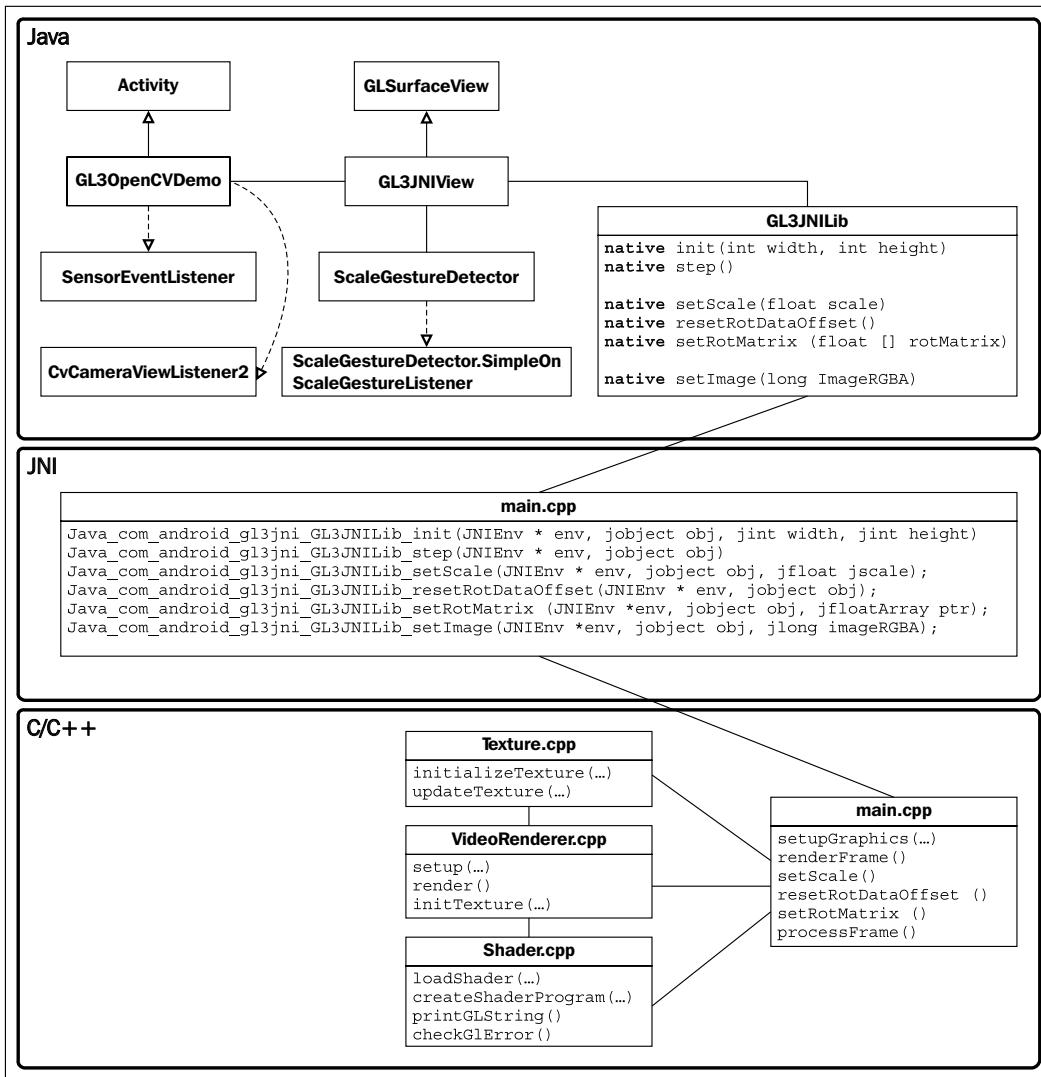
See also

Windows users should consult the following tutorials on Android development with OpenCV for setup instructions: http://docs.opencv.org/doc/tutorials/introduction/android_binary_package/android_dev_intro.html and http://docs.opencv.org/doc/tutorials/introduction/android_binary_package/dev_with_OCV_on_Android.html#native-c.

For further information on using OpenCV in an Android application, consult the online documentation at <http://opencv.org/platforms/android.html>.

Getting started II: Accessing the camera live feed using OpenCV

Next we need to demonstrate how to integrate OpenCV into our Android-based development framework. The following block diagram illustrates the core functions and relationship among the classes that will be implemented in this chapter (only the functions or classes relevant to the introduction of OpenCV will be discussed in this section):



In particular, we will demonstrate how to extract an image frame from the camera video stream for further image processing steps. The OpenCV library provides camera support for accessing the live camera feed (the raw data buffer of the video data stream) as well as controlling the camera parameters. This feature allows us to get the raw frame data from the live preview camera with optimal resolution, frame rate, and image format.

Getting ready

The demos in this chapter build upon the basic structure introduced in the sample code of *Chapter 8, Interactive Real-time Data Visualization on Mobile Devices* which utilizes the multi-touch interface and motion sensor inputs to enable interactive real-time data visualization on mobile devices. The major changes that are made to support OpenCV will be highlighted. For the complete code, download the code package from the Packt Publishing website.

How to do it...

First, we will highlight the changes to the Java source files required to enable the use of OpenCV and the OpenCV camera module. Rename `GL3JNIActivity.java` (`src/com/android/g13jni/`) as `GL3OpenCVDemo.java` and modify the code as follows:

1. Include the packages for the OpenCV library:

```
package com.android.g13jni;
...
import org.opencv.android.BaseLoaderCallback;
import org.opencv.android.LoaderCallbackInterface;
import org.opencv.android.OpenCVLoader;
import org.opencv.android.CameraBridgeViewBase;
import org.opencv.android.CameraBridgeViewBase.CvCameraViewFrame;
import org.opencv.android.CameraBridgeViewBase.CvCameraViewListener2;
import org.opencv.core.CvType;
import org.opencv.core.Mat;

import android.widget.RelativeLayout;
import android.view.SurfaceView;
```

2. Add the `CvCameraViewListener2` interface to the `GL3OpenCVDemo` class:

```
public class GL3OpenCVDemo extends Activity implements
    SensorEventListener, CvCameraViewListener2{
```

3. Create the variables to handle the camera view:

```
private GL3JNIView mView=null;
...
private boolean g13_loaded = false;
private CameraBridgeViewBase mOpenCvCameraView;
private RelativeLayout l_layout;
```

4. Implement the `BaseLoaderCallback` function for `OpenCVLoader`:

```
private BaseLoaderCallback mLoaderCallback = new
    BaseLoaderCallback(this) {
@Override
public void onManagerConnected(int status) {
    switch (status) {
        case LoaderCallbackInterface.SUCCESS: {
            Log.i("OpenCVDemo", "OpenCV loaded successfully");
            // load the library *AFTER* we have OpenCV lib ready!
            System.loadLibrary("gl3jni");
            gl3_loaded = true;

            //load the view as we have all JNI loaded
            mView = new GL3JNIView(getApplicationContext());
            l_layout.addView(mView);
            setContentView(l_layout);

            /* enable the camera, and push the images to the
               OpenGL layer */
            mOpenCvCameraView.enableView();
        } break;
        default: {
            super.onManagerConnected(status);
        } break;
    }
}
};
```

5. Implement the OpenCV camera callback functions and pass the image data to the JNI C/C++ side for processing and rendering:

```
public void onCameraViewStarted(int width, int height) {
}
public void onCameraViewStopped() {
}
public Mat onCameraFrame(CvCameraViewFrame inputFrame) {
    //Log.i("OpenCVDemo", "Got Frame\n");
    Mat input = inputFrame.rgba();
    if(gl3_loaded){
        GL3JNILib.setImage(input.nativeObj);
    }
    //don't show on the java side
    return null;
}
```

6. Initialize the camera in the `onCreate` function, upon starting the application:

```
@Override protected void onCreate(Bundle icicle) {  
    super.onCreate(icicle);  
    ...  
    //setup the Java Camera with OpenCV  
    setContentView(R.layout.ar);  
    l_layout =  
        (RelativeLayout)findViewById(R.id.linearLayoutRest);  
    mOpenCvCameraView =  
        (CameraBridgeViewBase)findViewById(R.id.opencv_camera_  
            surface_view);  
    mOpenCvCameraView.setVisibility( SurfaceView.VISIBLE );  
    mOpenCvCameraView.setMaxFrameSize(1280, 720); /* cap it at  
        720 for performance issue */  
    mOpenCvCameraView.setCvCameraViewListener(this);  
    mOpenCvCameraView.disableView();  
}
```

7. Load the OpenCV library using the synchronized initialization function called `initAsync` from the `OpenCVLoader` class. This event is captured by the `BaseLoaderCallback mLoaderCallback` function defined earlier:

```
@Override  
protected void onResume() {  
    super.onResume();  
    OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_3_0_0,  
        this, mLoaderCallback);  
    ...  
}
```

8. Finally, handle the `onPause` event, which pauses the camera preview when the application is no longer running in the foreground:

```
@Override  
protected void onPause() {  
    super.onPause();  
    mSensorManager.unregisterListener(this);  
    //stop the camera  
    if(mView!=null){  
        mView.onPause();  
    }  
    if (mOpenCvCameraView != null)  
        mOpenCvCameraView.disableView();  
    gl3_loaded = false;  
}
```

9. Now inside `GL3JNILib.java` (`src/com/android/gl3jni/`), add the native `setImage` function to pass the camera raw data. The entire source file is shown here, given its simplicity:

```
package com.android.gl3jni;

public class GL3JNILib {
    public static native void init(int width, int height);
    public static native void step();

    //pass the image to JNI C++ side
    public static native void setImage(long imageRGBA);

    //pass the device rotation angles and the scaling factor
    public static native void resetRotDataOffset();
    public static native void setRotMatrix(float []
        rotMatrix);
    public static native void setScale(float scale);
}
```

10. Finally, the source code inside `GL3JNIView.java` is virtually identical except that we offer the option to reset the rotation data and call the `setZOrderOnTop` function to ensure that the OpenGL layer is on top of the Java layer:

```
class GL3JNIView extends GLSurfaceView {
    ...
    public GL3JNIView(Context context) {
        super(context);
        // Pick an EGLConfig with RGB8 color, 16-bit depth, no stencil
        setZOrderOnTop(true);
        setEGLConfigChooser(8, 8, 8, 8, 16, 0);
        setEGLContextClientVersion(3);
        getHolder().setFormat(PixelFormat.TRANSLUCENT);
        renderer = new Renderer();
        setRenderer(renderer);
        //handle gesture input
        mScaleDetector = new ScaleGestureDetector(context, new
            ScaleListener());
    }
    ...
    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        mScaleDetector.onTouchEvent(ev);
        int action = ev.getActionMasked();
        switch (action) {
```

```
        case MotionEvent.ACTION_DOWN:
            GL3JNILib.resetRotDataOffset();
            break;
        }
        return true;
    }
    ...
}
```

11. Finally, define the JNI prototypes to interface with the Java side in the `main.cpp` file that connects all components.

```
//external calls for Java
extern "C" {
    JNIEXPORT void JNICALL
    Java_com_android_gl3jni_GL3JNILib_setImage(JNIEnv * jenv,
        jobject, jlong imageRGBA);
};

JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setImage(
    JNIEnv * jenv, jobject, jlong imageRGBA) {
    cv::Mat* image = (cv::Mat*) imageRGBA;
    /* use mutex lock to ensure the write/read operations
       are synced (to avoid corrupting the frame) */
    pthread_mutex_lock(&count_mutex);
    frame = image->clone();
    pthread_mutex_unlock(&count_mutex);
    //LOGI("Got Image: %dx%d\n", frame.rows, frame.cols);
}
```

12. To access the device camera, the following elements must be declared in the `AndroidManifest.xml` file to ensure we have the permission to control the camera. In our current example, we request access to the front and back cameras with autofocus support.

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-feature android:name="android.hardware.camera"
    android:required="false"/>
<uses-feature
    android:name="android.hardware.camera.autofocus"
    android:required="false"/>
<uses-feature android:name="android.hardware.camera.front"
    android:required="false"/>
<uses-feature
    android:name="android.hardware.camera.front.autofocus"
    android:required="false"/>
```

At this point, we have developed a full demo application that supports OpenCV and real-time camera feed. In the next section, we will connect the camera raw data stream to the OpenGL layer and perform real-time feature extraction with OpenCV in C/C++.

How it works...

On the Java side, we have integrated the OpenCV Manager (installed previously) to handle the dynamic loading of all libraries at runtime. Upon starting the application, we must call the `OpenCVLoader.initAsync` function; all OpenCV-related JNI libraries must only be called after the OpenCV libraries are successfully loaded. To synchronize these actions in our case, the callback function (`BaseLoaderCallback`) checks the status of the initialization of OpenCV, and we proceed with the `System.loadLibrary` function to initialize OpenGL and other components only if the OpenCV loader returns success (`LoaderCallbackInterface.SUCCESS`). For simplicity, we did not include the implementation to handle library loading exceptions in this demo.

On the sensor side, we have also changed the implementation for the `SensorManager` function to return the rotation matrix instead of the Euler angles to avoid the issue of Gimbal lock (refer to http://en.wikipedia.org/wiki/Gimbal_lock). We also remapped the coordinates (from device orientation to OpenGL camera orientation) using the `SensorManager.remapCoordinateSystem` function. Then the rotation matrix is directed to the OpenGL side with the native calls `GL3JNILib.setRotMatrix`. Also, we can allow the user to reset the default orientation by touching the screen. This is achieved by calling the `GL3JNILib.resetRotDataOffset` function, which resets the rotation matrix with the touch event.

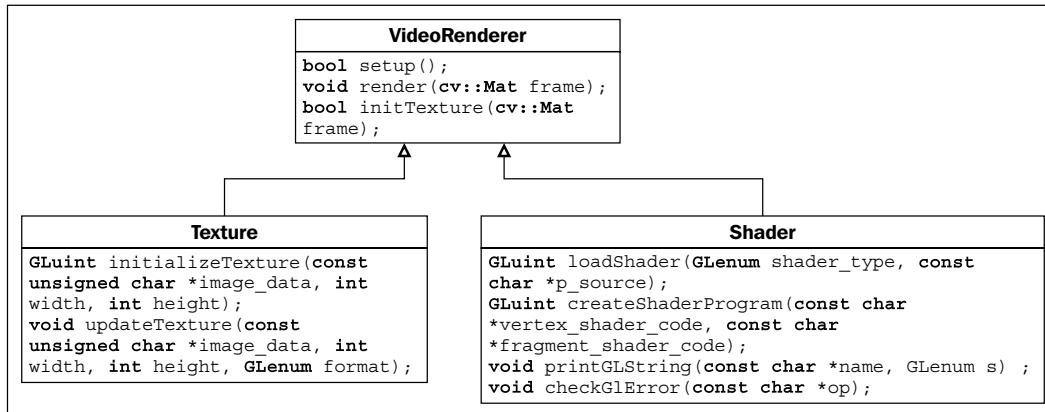
Additionally, we have added the OpenCV `CvCameraViewListener2` interface and `CameraBridgeViewBase` class to enable native camera access. The `CameraBridgeViewBase` class is a basic class that handles the interaction with the Android Camera class and OpenCV library. It is responsible for controlling the camera, such as resolution, and processing the frame, such as changing the image format. The client implements `CvCameraViewListener` to receive callback events. In the current implementation, we manually set the resolution as 1280 x 720. However, we can increase or decrease the resolution based on the application needs. Finally, the color frame buffers are returned in RGBA format, and the data stream will be transferred to the JNI C/C++ side and rendered using texture mapping.

Displaying real-time video using texture mapping

Today, most mobile phones are equipped with cameras that are capable of capturing high-quality photos as well as videos. For example, the Samsung Galaxy Note 4 is equipped with a 16MP back-facing camera as well as a 3.7MP front-facing camera for video conferencing applications. With these built-in cameras, we can record high-definition videos with exceptional image quality in both outdoor and indoor environments. The ubiquity of these imaging sensors, as well as the increasing computational capability of mobile processors, now enable us to develop much more interactive applications such as real-time tracking of objects or faces.

By combining OpenGL with the OpenCV library, we can create interactive applications that perform real-time video processing of the real world to register and augment 3D virtual information onto real-world objects. Since both libraries are hardware-accelerated (GPU and CPU optimized), it is important that we explore the use of these libraries to obtain real-time performance.

In the previous section, we introduced the framework that provides access to the live camera feed. Here, we will create a full demo that displays real-time video using OpenGL-based texture mapping techniques (similar to those introduced in *Chapter 4, Rendering 2D Images and Videos with Texture Mapping* to *Chapter 6, Rendering Stereoscopic 3D Models using OpenGL*, except we will deploy OpenGL ES for mobile platforms), and processes the video stream to perform corner detection using OpenCV. To help readers understand the additional code needed to finalize the demo, here is an overview diagram of the implementation:



Getting ready

This demo requires the completion of all the *Getting ready* steps to enable the capture of the real-time video stream using OpenCV on an Android device. The implementation of the shader program and texture mapping code is based on the demos from *Chapter 8, Interactive Real-time Data Visualization on Mobile Devices*.

How to do it...

On the native code side, create two new files called `VideoRenderer.hpp` and `VideoRenderer.cpp`. These files contain the implementation to render the video using texture mapping. Also, we will import the `Texture.cpp` and `Texture.hpp` files from the previous chapter to handle texture creation.

Inside the `VideoRenderer.hpp` file, define the `VideoRenderer` class as follows (the details of each function will be discussed next):

```
#ifndef VIDEORENDERER_H_
#define VIDEORENDERER_H_
//The shader program and basic OpenGL calls
#include <Shader.hpp>
//for texture support
#include <Texture.hpp>
//opencv support
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

class VideoRenderer {
public:
    VideoRenderer();
    virtual ~VideoRenderer();
    //setup all shader program and texture mapping variables
    bool setup();
    bool initTexture(cv::Mat frame);
    //render the frame on screen
    void render(cv::Mat frame);

private:
    //this handles the generic camera feed view
    GLuint gProgram;
    GLuint gvPositionHandle;
    GLuint vertexUVHandle;
    GLuint textureSamplerID;
```

```

        GLuint texture_id;
        Shader shader;
    };

#endif /* VIDEORENDERER_H_ */

```

Inside the `VideoRenderer.cpp` file, we implement each of the three key member functions (`setup`, `initTexture`, and `render`). Here is the complete implementation:

1. Include the `VideoRenderer.hpp` header file, define functions to print debug messages, and define the constructor and destructor:

```

#include "VideoRenderer.hpp"

#define LOG_TAG      "VideoRenderer"
#define LOGI(...)     __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
#define LOGE(...)     __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

VideoRenderer::VideoRenderer() {
}

VideoRenderer::~VideoRenderer() {
}

```

2. Define the vertex and fragment shaders as well as associated configuration steps (similar to *Chapter 8, Interactive Real-time Data Visualization on Mobile Devices*):

```

bool VideoRenderer::setup() {
    // Vertex shader source code
    const char g_vshader_code[] =
        "#version 300 es\n"
        "layout(location = 1) in vec4 vPosition;\n"
        "layout(location = 2) in vec2 vertexUV;\n"
        "out vec2 UV;\n"
        "void main() {\n"
            "    gl_Position = vPosition;\n"
            "    UV=vertexUV;\n"
        "}\n";
    // fragment shader source code
    const char g_fshader_code[] =
        "#version 300 es\n"
        "precision mediump float;\n"
        "out vec4 color;\n"
        "uniform sampler2D textureSampler;\n"
        "in vec2 UV;\n"

```

```
"void main() { \n"
    "    color = vec4(texture(textureSampler, UV).rgb,
    1.0); \n"
} \n";

LOGI("setupVideoRenderer");
gProgram = shader.createShaderProgram(g_vshader_code,
    g_fshader_code);
if (!gProgram) {
    LOGE("Could not create program.");
    return false;
}

gvPositionHandle = glGetAttribLocation(gProgram,
    "vPosition");
shader.checkGlError("glGetAttribLocation");
LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
gvPositionHandle);

vertexUVHandle = glGetAttribLocation(gProgram,
    "vertexUV");
shader.checkGlError("glGetAttribLocation");
LOGI("glGetAttribLocation(\"vertexUV\") = %d\n",
vertexUVHandle);

textureSamplerID = glGetUniformLocation(gProgram,
    "textureSampler");
shader.checkGlError("glGetUniformLocation");
LOGI("glGetUniformLocation(\"textureSampler\") = %d\n",
textureSamplerID);

return true;
}
```

3. Initialize and bind the texture:

```
bool VideoRenderer::initTexture(cv::Mat frame) {
    texture_id = initializeTexture(frame.data,
        frame.size().width, frame.size().height);
    //binds our texture in Texture Unit 0
    glBindTexture(GL_TEXTURE_2D, texture_id);
    glUniform1i(textureSamplerID, 0);

    return true;
}
```

4. Render the camera feed on the screen with texture mapping:

```
void VideoRenderer::render(cv::Mat frame){  
    //our vertices  
    const GLfloat g_vertex_buffer_data[] = {  
        1.0f,1.0f,0.0f,  
        -1.0f,1.0f,0.0f,  
        -1.0f,-1.0f,0.0f,  
        1.0f,1.0f  
        ,0.0f,  
        -1.0f,-1.0f,0.0f,  
        1.0f,-1.0f,0.0f  
    };  
    //UV map for the vertices  
    const GLfloat g_uv_buffer_data[] = {  
        1.0f, 0.0f,  
        0.0f, 0.0f,  
        0.0f, 1.0f,  
        1.0f, 0.0f,  
        0.0f, 1.0f,  
        1.0f, 1.0f  
    };  
  
    glUseProgram(gProgram);  
    shader.checkGlError("glUseProgram");  
  
    glEnableVertexAttribArray(gvPositionHandle);  
    shader.checkGlError("	glEnableVertexAttribArray");  
  
    glEnableVertexAttribArray(vertexUVHandle);  
    shader.checkGlError("	glEnableVertexAttribArray");  
  
    glVertexAttribPointer(gvPositionHandle, 3, GL_FLOAT,  
        GL_FALSE, 0, g_vertex_buffer_data);  
    shader.checkGlError("glVertexAttribPointer");  
  
    glVertexAttribPointer(vertexUVHandle, 2, GL_FLOAT,  
        GL_FALSE, 0, g_uv_buffer_data);  
    shader.checkGlError("glVertexAttribPointer");  
  
    updateTexture(frame.data, frame.size().width,  
        frame.size().height, GL_RGBA);
```

```
//draw the camera feed on the screen
glDrawArrays(GL_TRIANGLES, 0, 6);
shader.checkGlError("glDrawArrays");

glDisableVertexAttribArray(gvPositionHandle);
glDisableVertexAttribArray(vertexUVHandle);
}
```

To further enhance the readability of the code, we encapsulate the handling of the shader program and texture mapping inside `Shader.hpp` (`Shader.cpp`) and `Texture.hpp` (`Texture.cpp`), respectively. We will only show the header files here for completeness and refer readers to the code package on the Packt Publishing website for the detailed implementation of each function.

Here is the `Shader.hpp` file:

```
#ifndef SHADER_H_
#define SHADER_H_

#define GLM_FORCE_RADIANS
#include <jni.h>
#include <android/log.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <GLES3/gl3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

class Shader {
public:
    Shader();
    virtual ~Shader();
    GLuint loadShader(GLenum shader_type, const char*p_source);
    GLuint createShaderProgram(const char*vertex_shader_code,
        const char*fragment_shader_code);
    void printGLString(const char *name, GLenum s) ;
    void checkGlError(const char* op);
};

#endif /* SHADER_H_ */
```

The Texture.hpp file should read:

```
#ifndef TEXTURE_HPP
#define TEXTURE_HPP

#include <GLES3/gl3.h>

class Texture {
public:
    Texture();
    virtual ~Texture();
    GLuint initializeTexture(const unsigned char *image_data,
                             int width, int height);
    void updateTexture(const unsigned char *image_data, int width,
                       int height, GLenum format);
};

#endif
```

Finally, we integrate everything inside the main.cpp file with the following steps:

1. Include all headers. In particular, include pthread.h to handle synchronization and OpenCV libraries for image processing.

```
...
#include <pthread.h>
#include <Texture.hpp>
#include <Shader.hpp>
#include <VideoRenderer.hpp>

//including opencv headers
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
...
```

2. Define the VideoRenderer and Shader objects, as well as the pthread_mutex_t lock variable to handle synchronization for data copying using a mutex lock.

```
//mutex lock for data copying
pthread_mutex_t count_mutex;
...
//pre-set image size.
const int IMAGE_WIDTH = 1280;
const int IMAGE_HEIGHT = 720;
```

```
bool enable_process = true;
//main camera feed from the Java side
cv::Mat frame;
//all shader related code
Shader shader;
//for video rendering
VideoRenderer videorenderer;
```

3. Set up the VideoRenderer object in the setupGraphics function and initialize the texture.

```
bool setupGraphics(int w, int h) {
    ...
    videorenderer.setup();
    //template for the first texture
    cv::Mat frameM(IMAGE_HEIGHT, IMAGE_WIDTH, CV_8UC4,
        cv::Scalar(0,0,0,255));
    videorenderer.initTexture(frameM);
    frame = frameM;
    ...
    return true;
}
```

4. Create a processFrame helper function to handle feature extraction with the OpenCV goodFeaturesToTrack function. The function also draws the result directly on the frame for visualization.

```
void processFrame(cv::Mat *frame_local){
    int maxCorners = 1000;
    if( maxCorners < 1 ) { maxCorners = 1; }
    cv::RNG rng(12345);
    // Parameters for Shi-Tomasi algorithm
    std::vector<cv::Point2f> corners;
    double qualityLevel = 0.05;
    double minDistance = 10;
    int blockSize = 3;
    bool useHarrisDetector = false;
    double k = 0.04;

    // Copy the source image
    cv::Mat src_gray;
    cv::Mat frame_small;
    cv::resize(*frame_local, frame_small, cv::Size(), 0.5,
        0.5, CV_INTER_AREA);
    cv::cvtColor(frame_small, src_gray, CV_RGB2GRAY );
```

```
// Apply feature extraction
cv::goodFeaturesToTrack( src_gray, corners, maxCorners,
    qualityLevel, minDistance, cv::Mat(), blockSize,
    useHarrisDetector, k );

// Draw corners detected on the image
int r = 10;
for( int i = 0; i < corners.size(); i++ )
{
    cv::circle(*frame_local, 2*corners[i], r,
        cv::Scalar(rng.uniform(0,255),
        rng.uniform(0,255), rng.uniform(0,255), 255), -1, 8, 0
    );
}
//LOGI("Found %d features", corners.size());
}
```

5. Implement frame copying with mutex lock synchronization (to avoid frame corruption due to shared memory and race condition) in the `renderFrame` function. Process the frame with the OpenCV library and render the result using OpenGL texture-mapping techniques.

```
void renderFrame() {
    shader.checkGlError("glClearColor");
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    shader.checkGlError("glClear");

    pthread_mutex_lock(&count_mutex);
    cv::Mat frame_local = frame.clone();
    pthread_mutex_unlock(&count_mutex);

    if(enable_process)
        processFrame(&frame_local);
```

- ```
//render the video feed on screen
videorenderer.render(frame_local);
//LOGI("Rendering OpenGL Graphics");
}

6. Define the JNI prototypes and implement the setImage function, which receives the raw camera image data from the Java side using a mutex lock to ensure data copying is protected. Also, implement the toggleFeatures function to turn feature tracking on and off upon touching the screen.
```

```
extern "C" {
..
 JNIEEXPORT void JNICALL
 Java_com_android_gl3jni_GL3JNILib_setImage(JNIEnv *
 jenv, jobject, jlong imageRGBA);
 //toggle features
 JNIEEXPORT void JNICALL
 Java_com_android_gl3jni_GL3JNILib_toggleFeatures(JNIEnv
 * jenv, jobject);
};

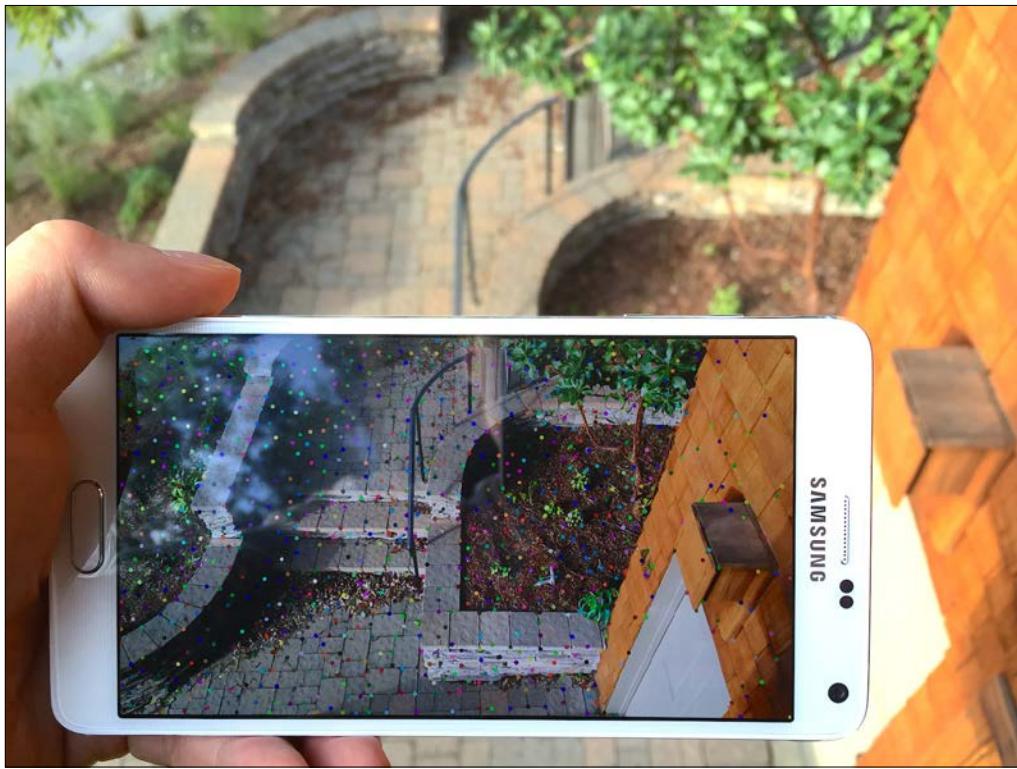
JNIEEXPORT void JNICALL
 Java_com_android_gl3jni_GL3JNILib_toggleFeatures(JNIEnv *
 env, jobject obj){
 //toggle the processing on/off
 enable_process = !enable_process;
}
JNIEEXPORT void JNICALL
 Java_com_android_gl3jni_GL3JNILib_setImage(
 JNIEnv * jenv, jobject, jlong imageRGBA) {
 cv::Mat* image = (cv::Mat*) imageRGBA;
 /* use mutex lock to ensure the write/read operations
 are synced (to avoid corrupting the frame) */
 pthread_mutex_lock(&count_mutex);
```

```
frame = image->clone();
pthread_mutex_unlock(&count_mutex);
//LOGI("Got Image: %dx%d\n", frame.rows, frame.cols);
}
```



The resulting image is a post-processed frame from OpenCV. In addition to displaying the raw video frame, we demonstrate that our implementation can easily be extended to support real-time video processing with OpenCV. The `processFrame` function uses the OpenCV `goodFeaturesToTrack` corner detection function and we overlay all corners extracted from the scene on the image.

Image features are the fundamental elements for many tracking algorithms such as **Simultaneous localization and Mapping (SLAM)** as well as recognition algorithms such as image-based matching. For example, with the SLAM algorithm, we can construct a map of the environment and, at the same time, keep track of the position of the device in space. Such techniques are particularly useful in AR applications as we always need to align the virtual world with the real world. Next, we can see a feature extraction algorithm (corner detection) running in real-time on a mobile phone.



## How it works...

The `VideoRenderer` class has two primary functions:

- ▶ Creating the shader program that handles texture mapping (`Shader.cpp` and `Texture.cpp`).
- ▶ Updating the texture memory with the OpenCV raw camera frame. Each time a new frame is retrieved from OpenCV, we call the `render` function, which updates the texture memory and also draws the frame on the screen.

The `main.cpp` file connects all the components of the implementation, and encapsulates all the logics for the interaction. It interfaces with the Java side (for example, `setImage`) and we offload all computationally intensive tasks to the C++ native side. For example, the `processFrame` function handles the OpenCV video processing pipeline, and we can efficiently handle memory I/O and parallelization. On the other hand, the `videoRenderer` class accelerates rendering with OpenGL for real-time performance on the mobile platform.

One may notice that the implementations of OpenGL and OpenCV on Android are mostly identical to the desktop version. That's the key reason why we employ such cross-platform languages as we can easily extend our code to any future platform with minimal effort.

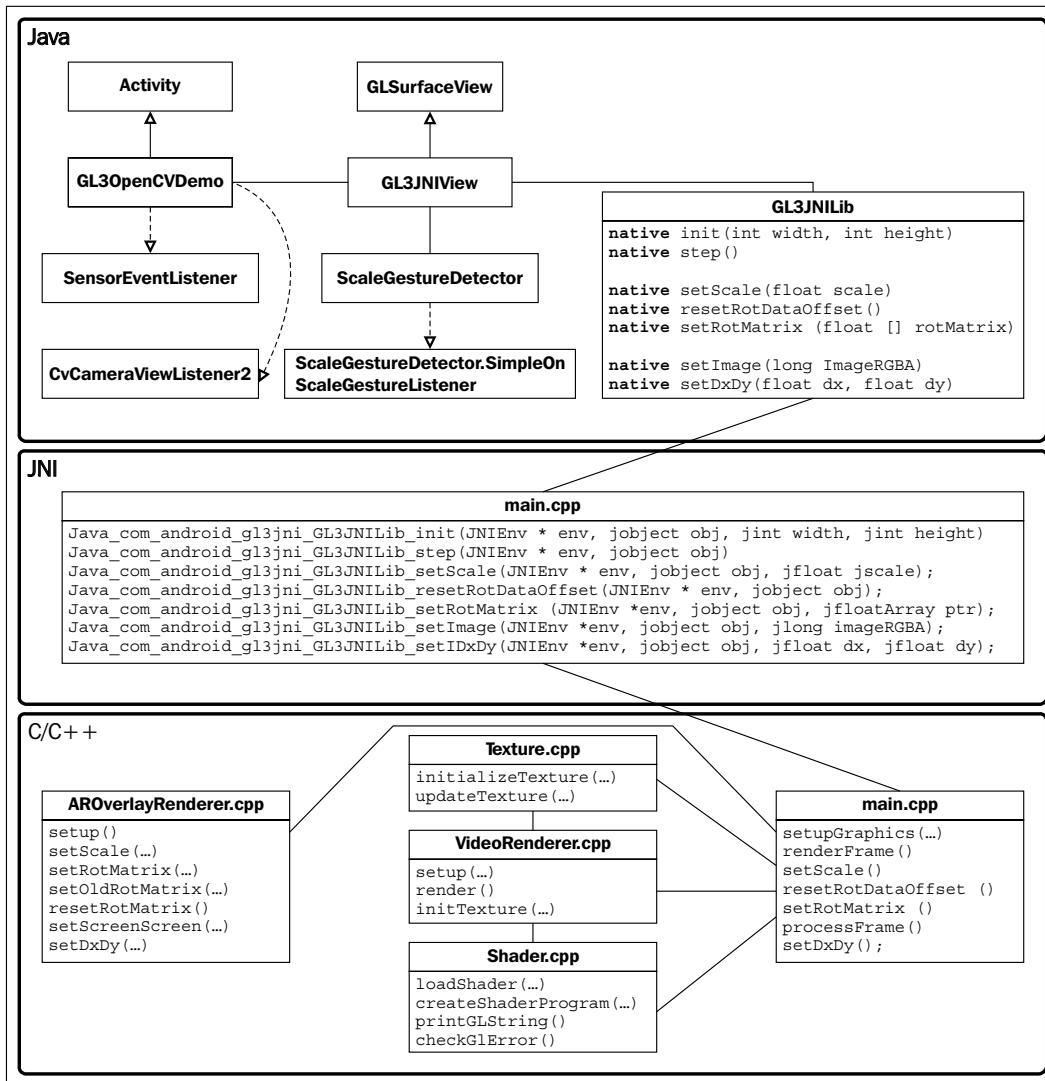
### See also

On a mobile platform, computational resources are particularly limited and thus it is important to optimize the use of all available hardware resources. With OpenGL-based hardware acceleration, we can reduce most of our overhead in rendering graphics in 2D and 3D on the graphics processor. In the near future, especially with the emergence of mobile processors supporting GPGPU (for example, Nvidia's K1 mobile processor), we will enable more parallelized processing for computer vision algorithms and offer real-time performance for many applications on a mobile device. For example, Nvidia now officially supports CUDA for all its upcoming mobile processors, so we will see many more real-time image processing, machine learning (such as deep learning algorithms), and high-performance graphics emerging on the mobile platform. See the following website for more information:

<https://developer.nvidia.com/embedded-computing>.

## Augmented reality-based data visualization over real-world scenes

In our ultimate demo, we will introduce the basic framework for AR-based data visualization by overlaying 3D data on real-world objects and scenes. We apply the same GPU-accelerated simulation model and register it to the world with a sensor-based tracking approach. The following diagram illustrates the final architecture of the implementation in this chapter:



## Getting ready

This final demo integrates together all the concepts previously introduced in this chapter and requires the capture (and possibly processing) of a real-time video stream using OpenCV on an Android-based phone. To reduce the complexity of the code, we have created the Augmented Reality layer (AROverlayRenderer) and we can improve the registration, alignment, and calibration of the layer with more advanced algorithms in the future.

## How to do it...

Let's define a new class called `AROverlayRenderer` inside the `AROverlayRenderer.hpp` file:

```
#ifndef AROVERLAYRENDERER_H_
#define AROVERLAYRENDERER_H_

#include<Shader.hpp>

class AROverlayRenderer {
public:
 AROverlayRenderer();
 virtual ~AROverlayRenderer();
 void render();
 bool setup();
 void setScale(float s);

 void setOldRotMatrix(glm::mat4 r_matrix);
 void setRotMatrix(glm::mat4 r_matrix);
 void resetRotMatrix();
 void setScreenSize(int width, int height);
 void setDxDy (float dx, float dy);
private:
 //this renders the overlay view
 GLuint gProgramOverlay;
 GLuint gvOverlayPositionHandle;
 GLuint gvOverlayColorHandle;
 GLuint matrixHandle;
 GLuint sigmaHandle;
 GLuint scaleHandle;

 //vertices for the grid
 int grid_size;
 GLfloat *gGrid;
 GLfloat sigma;

 //for handling the object rotation from user
 GLfloat dx, dy;
 GLfloat rotX, rotY;

 //the view matrix and projection matrix
 glm::mat4 g_view_matrix;
```

```
glm::mat4 g_projection_matrix;

//initial position of the camera
glm::vec3 g_position;
//FOV of the virtual camera in OpenGL
float g_initial_fov;

glm::mat4 rotMatrix;
glm::mat4 old_rotMatrix;

float scale;
int width;
int height;

Shader shader;
void computeProjectionMatrices();
void computeGrid();
};

#endif /* AROVERLAYRENDERER_H_ */
```

Now implement the `AROverlayRenderer` member functions inside the `AROverlayRenderer.cpp` file:

1. Include the `AROverlayRenderer.hpp` header file and define functions to print messages as well as the constructor and destructor:

```
#include "AROverlayRenderer.hpp"

#define LOG_TAG "AROverlayRenderer"
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, LOG_TAG, __VA_ARGS__)

AROverlayRenderer::AROverlayRenderer() {
 //initial position of the camera
 g_position = glm::vec3(0.0f, 0.0f, 0.0f);

 //FOV of the virtual camera in OpenGL
 //45 degree FOV
 g_initial_fov = 45.0f*glm::pi<float>()/180.0f;

 /* scale for the panel and other objects, allow for
 zooming in with pinch. */
 scale = 1.0f;
```

```
 dx=0.0f; dy=0.0f;
 rotX=0.0f, rotY=0.0f;
 sigma = 0;

 grid_size = 400;
 //allocate memory for the grid
 gGrid = (GLfloat*)
 malloc(sizeof(GLfloat)*grid_size*grid_size*3);
}

AROverlayRenderer::~AROverlayRenderer() {
 //delete all dynamically allocated objects here
 free(gGrid);
}
```

2. Initialize the grid pattern for the simulation:

```
void AROverlayRenderer::computeGrid() {
 float grid_x = grid_size;
 float grid_y = grid_size;
 unsigned int data_counter = 0;
 //define a grid ranging from -1 to +1
 for(float x = -grid_x/2.0f; x<grid_x/2.0f; x+=1.0f){
 for(float y = -grid_y/2.0f; y<grid_y/2.0f; y+=1.0f){
 float x_data = x/grid_x;
 float y_data = y/grid_y;
 gGrid[data_counter] = x_data;
 gGrid[data_counter+1] = y_data;
 gGrid[data_counter+2] = 0;
 data_counter+=3;
 }
 }
}
```

3. Set up the shader program to overlay graphics:

```
bool AROverlayRenderer::setup() {
 // Vertex shader source code
 static const char g_vshader_code_overlay[] =
 "#version 300 es\n"
 "in vec4 vPosition;\n"
 "uniform mat4 MVP;\n"
 "uniform float sigma;\n"
 "uniform float scale;\n"
 "out vec4 color_based_on_position;\n"
 "// Heat map generator\n"
```

```

"vec4 heatMap(float v, float vmin, float vmax) {\n"
" float dv;\n"
" float r=1.0, g=1.0, b=1.0;\n"
" if (v < vmin){\n"
" v = vmin;}\n"
" if (v > vmax){\n"
" v = vmax;}\n"
" dv = vmax - vmin;\n"
" if (v < (vmin + 0.25 * dv)) {\n"
" r = 0.0;\n"
" g = 4.0 * (v - vmin) / dv;\n"
" } else if (v < (vmin + 0.5 * dv)) {\n"
" r = 0.0;\n"
" b = 1.0 + 4.0 * (vmin + 0.25 * dv - v) / dv;\n"
" } else if (v < (vmin + 0.75 * dv)) {\n"
" r = 4.0 * (v - vmin - 0.5 * dv) / dv;\n"
" b = 0.0;\n"
" } else {\n"
" g = 1.0 + 4.0 * (vmin + 0.75 * dv - v) / dv;\n"
" b = 0.0;\n"
" }\n"
" return vec4(r, g, b, 0.1);\n"
"}\n"
"void main() {\n"
" //Simulation on GPU\n"
" float x_data = vPosition.x;\n"
" float y_data = vPosition.y;\n"
" float sigma2 = sigma*sigma;\n"
" float z = exp(-0.5*(x_data*x_data)/(sigma2) -\n"
"0.5*(y_data*y_data)/(sigma2));\n"
" vec4 position = vPosition;\n"
" position.z = z*scale;\n"
" position.x = position.x*scale;\n"
" position.y = position.y*scale;\n"
" gl_Position = MVP*position;\n"
" color_based_on_position = heatMap(position.z, 0.0,\n"
"0.5);\n"
" gl_PointSize = 5.0*scale;\n"
"}\n";

// fragment shader source code
static const char g_fshader_code_overlay[] =
"#version 300 es\n"
"precision mediump float;\n"

```

```
"in vec4 color_based_on_position;\n"
"out vec4 color;\n"
"void main() {\n"
 " color = color_based_on_position;\n"
}\n";\n\n//setup the shader for the overlay
gProgramOverlay =
 shader.createShaderProgram(g_vshader_code_overlay,
 g_fshader_code_overlay);
if (!gProgramOverlay) {
 LOGE("Could not create program for overlay.");
 return false;
}
//get handlers for the overlay side
matrixHandle = glGetUniformLocation(gProgramOverlay, "MVP");
shader.checkGLError("glGetUniformLocation");
LOGI("glGetUniformLocation(\"MVP\") = %d\n",
 matrixHandle);\n\n
gvOverlayPositionHandle = glGetAttribLocation(gProgramOverlay,
"vPosition");
shader.checkGLError("glGetAttribLocation");
LOGI("glGetAttribLocation(\"vPosition\") = %d\n",
 gvOverlayPositionHandle);\n\n
sigmaHandle = glGetUniformLocation(gProgramOverlay,
 "sigma");
shader.checkGLError("glGetUniformLocation");
LOGI("glGetUniformLocation(\"sigma\") = %d\n",
 sigmaHandle);\n\n
scaleHandle = glGetUniformLocation(gProgramOverlay,
 "scale");
shader.checkGLError("glGetUniformLocation");
LOGI("glGetUniformLocation(\"scale\") = %d\n",
 scaleHandle);\n\n
computeGrid();
}
```

4. Create helper functions to set the scale, screen size, and rotation variables from the touch interface:

```
void AROverlayRenderer::setScale(float s) {
 scale = s;
}

void AROverlayRenderer::setScreenSize(int w, int h) {
 width = w;
 height = h;
}

void AROverlayRenderer::setRotMatrix(glm::mat4 r_matrix){
 rotMatrix= r_matrix;
}

void AROverlayRenderer::setOldRotMatrix(glm::mat4
r_matrix){
 old_rotMatrix = r_matrix;
}

void AROverlayRenderer::resetRotMatrix(){
 old_rotMatrix = rotMatrix;
}

void AROverlayRenderer::setDxDy(float dx, float dy){
 //update the angle of rotation for each
 rotX += dx/width;
 rotY += dy/height;
}
```

5. Compute the projection and view matrices based on the camera parameters:

```
void AROverlayRenderer::computeProjectionMatrices(){
 //direction vector for z
 glm::vec3 direction_z(0.0, 0.0, -1.0);
 //up vector
 glm::vec3 up = glm::vec3(0.0, -1.0, 0.0);

 float aspect_ratio = (float)width/(float)height;
 float nearZ = 0.01f;
 float farZ = 50.0f;
 float top = tan(g_initial_fov/2*nearZ);
 float right = aspect_ratio*top;
 float left = -right;
 float bottom = -top;
```

```
g_projection_matrix = glm::frustum(left, right, bottom, top,
 nearZ, farZ);

g_view_matrix = glm::lookAt(
 g_position, // camera position
 g_position+direction_z, //viewing direction
 up // up direction
);
}
```

6. Render the graphics on the screen:

```
void AROverlayRenderer::render(){
 //update the variables for animations
 sigma+=0.002f;
 if(sigma>0.5f){
 sigma = 0.002f;
 }
 glUseProgram(gProgramOverlay);
 /* Retrieve the View and Model matrices and apply them to
 the rendering */
 computeProjectionMatrices();
 glm::mat4 projection_matrix = g_projection_matrix;
 glm::mat4 view_matrix = g_view_matrix;
 glm::mat4 model_matrix = glm::mat4(1.0);

 model_matrix = glm::translate(model_matrix,
 glm::vec3(0.0f, 0.0f, scale-5.0f));
 //X,Y reversed for the screen orientation
 model_matrix = glm::rotate(model_matrix,
 rotY*glm::pi<float>(), glm::vec3(-1.0f, 0.0f, 0.0f));
 model_matrix = glm::rotate(model_matrix,
 rotX*glm::pi<float>(), glm::vec3(0.0f, -1.0f, 0.0f));
 model_matrix = glm::rotate(model_matrix,
 90.0f*glm::pi<float>()/180.0f, glm::vec3(0.0f, 0.0f,
 1.0f));
 /* the inverse of rotational matrix is to counter-
 rotate
 the graphics to the center. This allows us to reset the
 camera orientation since R*inv(R) = I. */
 view_matrix =
 rotMatrix*glm::inverse(old_rotMatrix)*view_matrix;

 //create the MVP (model view projection) matrix
 glm::mat4 mvp = projection_matrix * view_matrix *
 model_matrix;
 glUniformMatrix4fv(matrixHandle, 1, GL_FALSE,
 &mvp[0][0]);
```

```
shader.checkGlError("glUniformMatrix4fv");
glEnableVertexAttribArray(gvOverlayPositionHandle);
shader.checkGlError("glEnableVertexAttribArray");
glVertexAttribPointer(gvOverlayPositionHandle, 3,
 GL_FLOAT, GL_FALSE, 0, gGrid);
shader.checkGlError("glVertexAttribPointer");
glUniform1f(sigmaHandle, sigma);
shader.checkGlError("glUniform1f");

glUniform1f(scaleHandle, 1.0f);
shader.checkGlError("glUniform1f");

//draw the overlay graphics
glDrawArrays(GL_POINTS, 0, grid_size*grid_size);
shader.checkGlError("glDrawArrays");
glDisableVertexAttribArray(gvOverlayPositionHandle);
}
```

7. Finally, we only need to make minor modifications to the `main.cpp` file used in the previous demo to enable the AR overlay on top of the real-time video stream (real-world scene). Only the relevant code snippets that highlight the required modifications are shown here (download the complete code from the Packt Publishing website):

```
...
#include <AROverlayRenderer.hpp>
...
AROverlayRenderer aroverlayrenderer;
...
bool setupGraphics(int w, int h) {
 ...
 videorenderer.setup();
 aroverlayrenderer.setup();
 ...
 videorenderer.initTexture(frame);
 aroverlayrenderer.setScreenSize(width, height);
}

void renderFrame() {
 ...
 videorenderer.render(frame);
 aroverlayrenderer.render();
}
...
extern "C" {
 ...
}
```

```
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setScale(JNIEnv *
env, jobject obj, jfloat jscale);
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_resetRotDataOffset(JNIEnv
* env, jobject obj);
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setRotMatrix (JNIEnv
*env, jobject obj, jfloatArray ptr);
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setDxDy(JNIEnv *env,
jobject obj, jfloat dx, jfloat dy);
};

...
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_resetRotDataOffset
(JNIEnv * env, jobject obj){
 aroverlayrenderer.resetRotMatrix();
}

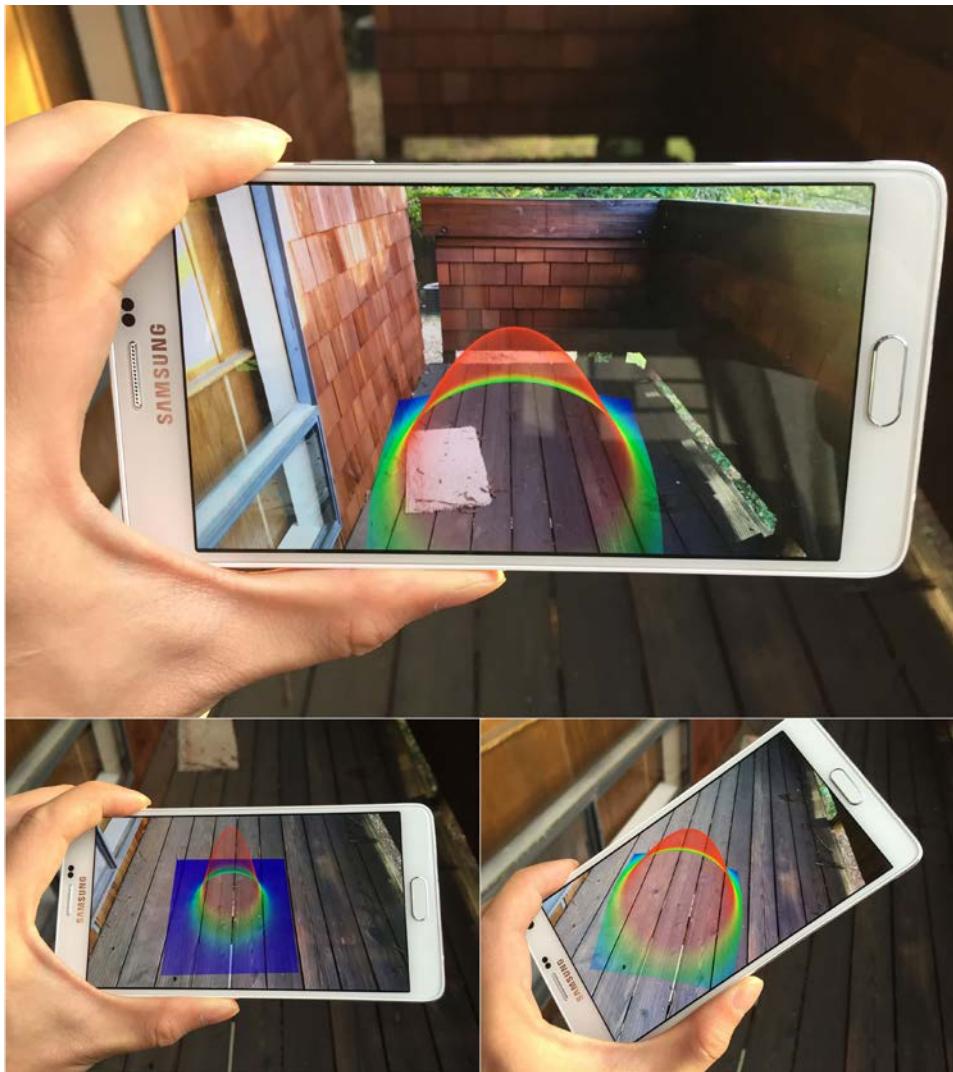
JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setScale (JNIEnv * env,
jobject obj, jfloat jscale)
{
 aroverlayrenderer.setScale(jscale);
 LOGI("Scale is %lf", scale);
}

JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_resetRotDataOffset
(JNIEnv * env, jobject obj){
 aroverlayrenderer.resetRotMatrix();
}

JNIEXPORT void JNICALL
Java_com_android_gl3jni_GL3JNILib_setRotMatrix
(JNIEnv *env, jobject obj, jfloatArray ptr) {
 jsize len = env->GetArrayLength(ptr);
 jfloat *body = env->GetFloatArrayElements(ptr, 0);
 //should be 16 elements from the rotation matrix
 glm::mat4 rotMatrix(1.0f);
 int count = 0;
 for(int i = 0; i<4; i++){
 for(int j=0; j<4; j++){
 rotMatrix[i][j] = body[count];
 count++;
 }
 }
 env->ReleaseFloatArrayElements(ptr, body, 0);
 aroverlayrenderer.setRotMatrix(rotMatrix);
}
```

```
JNIEXPORT void JNICALL
Java_com_android_g13jni_GL3JNILib_setDxDy(JNIEnv * env,
 jobject obj, jfloat dx, jfloat dy){
 aroverlayrenderer.setDxDy(dx, dy);
}
```

With this framework, one can overlay virtually any dataset on different real-world objects or surfaces and enable truly interactive applications, using the built-in sensors and gesture interface on mobile devices and emerging state-of-the-art wearable AR eyeglasses. Following are the results demonstrating a real-time, interactive, AR-based visualization of a 3-D dataset (in this case, a Gaussian distribution) overlaid on real-world scenes:



## How it works...

The key element for enabling an AR application is the ability to overlay information onto the real world. The `AROverlayRenderer` class implements the core functions essential to all AR applications. First, we create a virtual camera that matches the parameters of the actual camera on the mobile phone. Parameters such as the **field of view (FOV)** and aspect ratio of the camera are currently hard-coded, but we can easily modify them in the `computeProjectionMatrices` function. Then, to perform the registration between the real world and virtual world, we control the orientation of the virtual camera based on the orientation of the device. The orientation values are fed through the rotation matrix passed from the Java side (the `setRotMatrix` function) and we apply this directly to the OpenGL camera view matrix (`view_matrix`). Also, we use the multi-touch interface of the mobile phone to reset the default orientation of the rotation matrix. This is achieved by storing the rotational matrix value upon the touch event (the `resetRotDataOffset` function) and we apply the inverse to the rotational matrix to the view matrix (this is equivalent to rotating the camera in the opposite direction).

In terms of user interaction, we have enabled the pinch and drag option to support dynamic interaction with the virtual object. Upon the pinch event, we take the scale factor and we position the rendered object at a farther distance by applying the `glm::translate` function on the `model_matrix` variable. In addition, we rotate the virtual object by capturing the dragging action from the Java side (the `setDxDy` function). The user can control the orientation of the virtual object by dragging a finger across the screen. Together, these multi-touch gestures enable a highly interactive application interface that allows users to change the perspective of the rendered object intuitively.

Due to the underlying complexity of the calibration process, we will not cover these details here. However, advanced users may consult the following website for a more in-depth discussion: [http://docs.opencv.org/doc/tutorials/calib3d/camera\\_calibration/camera\\_calibration.html](http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html).

Also, the current registration process is purely based on the IMU, and it does not support translation (that is, the virtual object does not move exactly with the real world). To address this, we can apply various image-processing techniques such as mean shift tracking, feature-based tracking, and marker-based tracking to recover the full 6 DOF (degree of freedom) model of the camera. SLAM, for example, is a great candidate to recover the 6 DOF camera model, but its detailed implementation is beyond the scope of this chapter.

## See also

Indeed, in this chapter, we have only covered the fundamentals of AR. The field of AR is becoming an increasingly hot topic in both academia and industry. If you are interested in implementing AR data visualization applications on the latest wearable computing platforms (such as the one provided by Meta that features 3D gesture input and 3D stereoscopic output), visit the following websites:

- ▶ <https://www.getameta.com/>
- ▶ <http://www.eyetap.org/publications/>

For further technical details on AR eyeglasses, please consult the following publications:

- ▶ Raymond Lo, Alexander Chen, Valmiki Rampersad, Jason Huang, Han Wu, Steve Mann (2013). "Augmediated reality system based on 3D camera selfgesture sensing," IEEE International Symposium on Technology and Society (ISTAS) 2013, pp. 20-31.
- ▶ Raymond Lo, Valmiki Rampersad, Jason Huang, Steve Mann (2013). "Three Dimensional High Dynamic Range Veillance for 3D Range-Sensing Cameras," IEEE International Symposium on Technology and Society (ISTAS) 2013, pp. 255-265.
- ▶ Raymond Chun Hing Lo, Steve Mann, Jason Huang, Valmiki Rampersad, and Tao Ai. 2012. "High Dynamic Range (HDR) Video Image Processing For Digital Glass." In Proceedings of the 20th ACM international conference on Multimedia (MM '12). ACM, New York, NY, USA, pp. 1477-1480.
- ▶ Steve Mann, Raymond Lo, Jason Huang, Valmiki Rampersad, Ryan Janzen, Tao Ai (2012). "HDRchitecture: Real-Time stereoscopic HDR Imaging for Extreme Dynamic Range," In ACM SIGGRAPH 2012 Emerging Technologies (SIGGRAPH '12).



# Bibliography

This course is packaged keeping your journey in mind. It includes content from the following Packt products:

- *OpenGL Development Cookbook, Muhammad Mobeen Movania*
- *OpenGL 4.0 Shading Language Cookbook, Second Edition, David Wolff*
- *OpenGL Data Visualization Cookbook, Raymond C. H. Lo & William C. Y. Lo*



# Index

## Symbols

### 2D images

rendering, with texture mapping 466-478

### 2D plot

creating, primitives used 406-409

### 2D quad

drawing, tessellation shader used 244-249

### 2D texture

applying 114-119

### 2D visualization, of 3D/4D datasets 412-416

### 3D model

loading, in Wavefront Object (.obj)

format 512-518

rendering, with lines 518-528

rendering, with points 518-528

rendering, with triangles 518-528

### 3D model-sharing websites

references 528

### 3D plot

creating, with perspective

rendering 421-428

### 3ds Max 3DS (.3ds) 510

### 3D surface

drawing, tessellation shader  
used 249-254

### 7zip

URL 6

### -exts parameter 9

## A

### active vertex attributes

listing 30-33

### active vertex locations

listing 30-33

### Activity class 546

### Adobe Photoshop

URL 127

### ads function 98

### ADS shading

halfway vector, used 97-99

### ADS shading model

about 61, 114

implementing 61-65

### Airy disc 180

### aliasing shadow edges 269

### alpha maps

used, for discarding pixels 123-125

### ambient component 61

### Android

OpenCV, setting up on 602-604

### Android application

creating, with OpenGL ES 3.0 548-560

### Android Application

Package (APK) 556, 595

### Android Debug Bridge (adb) command 556

### Android Developers

references 538

### Android development, with OpenCV

URL, for tutorials 604

### Android NDK

about 537

integrating, by developing

basic framework 542-547

references 547

setting up 541, 542

### Android SDK

about 538

for Linux users, URL 541

for Window users, URL 541

setting up 538-541

**Android sensor framework**  
URL, for documentation 581  
**animation** 312  
**anti-aliasing** 187  
**anti-aliasing shadow edges**  
with PCF 270-273  
**Apache Ant**  
URL 541  
**Application Binary Interface (ABI)** 555  
**applyFilter function** 360  
**AR-based data visualization**  
over real-world scenes 624-636  
**AR eyeglasses**  
references 637  
**arrays**  
passing, to functions 70  
**artifacts** 187  
**Assimp**  
reference 512  
**Assimp 3.0**  
URL 510  
**attribute indexes**  
specifying, without layout qualifiers 29  
**Augmented Reality (AR)** 534, 602

## B

**back faces**  
rendering, for shadow mapping 270  
**basic framework**  
developing, for Android NDK  
integration 542-547  
**Bernstein polynomials** 239  
**binary library**  
URL 511  
**black holes** 346  
**blending functions** 239  
**bloom effect**  
about 180-183  
creating 180-183  
**bright-pass filter** 180  
**buildOffsetTex function** 278  
**built-in Inertial Measurement Units (IMUs)**  
real-time data, visualizing from 562-580

## C

**camera calibration, with OpenCV**  
URL 636  
**camera live feed**  
accessing, OpenCV used 604-611  
**canonical viewing volume** 262  
**centroid qualifier** 191  
**chromatic aberration** 144  
**chromaticity** 176  
**CIE XYZ color space** 176  
**cloth simulation**  
compute shader, using for 350-356  
**cloud-like effect**  
creating 296-298  
**command line argument, GLLoadGen**  
URL 8  
**compatibility profile**  
about 5  
versus core profile 5  
**computational resources** 624  
**compute shader**  
about 341, 342  
compute space 342, 343  
edge detection filter,  
implementing with 357-361  
executing 344  
particle simulation,  
implementing with 345-349  
used, for creating fractal texture 362-365  
using, for cloth simulation 350-356  
work groups 342, 343  
**compute space** 342, 343  
**const qualifier** 70  
**convolution filter** 160, 161  
**core profile**  
about 5  
versus compatibility profile 5  
**C++ shader program class**  
building 48-52  
**cube map**  
about 133  
used, for simulating reflection 132-139  
used, for simulating refraction 139-143

**cubic Bezier curve**  
     about 239  
     drawing, tessellation shader used 239-244

**CUDA** 341, 624

## D

**data**  
     sending, to shaders using uniform variables 33-37  
     sending, to shaders using vertex attributes 21-26  
     sending, to shader using vertex buffer objects 21-27

**debug messages**  
     obtaining 45-48

**deferred shading**  
     about 160, 192  
     using 192-197

**Degree of Polarization (DOP)** 487

**deprecation model** 5

**depth-sensing cameras**  
     raw data, capturing 492-494

**depth test**  
     configuring 109-111

**diffuse component** 61

**diffuse reflectivity** 57

**diffuse shading**  
     implementing, with single point light source 56-60

**directional light source** 66  
     about 91  
     shading with 91-93

**DirectX 11 Terrain Tessellation**  
     URL 258

**discard keyword** 55, 82

**disintegration effect**  
     creating 302-304

## E

**edge detection filter**  
     about 160  
     applying 160-166

implementing, with compute shader 357-361

**electrocardiogram (ECG)** 395

**element arrays**  
     using 29, 30

**emitEdgeQuad function** 287

**environment mapping** 132

**Euler method** 322

## F

**field of view angle (fovY)** 419

**field of view (FOV)** 636

**filters**  
     reference link 579

**fire**  
     simulating, with particle system 335-337

**fixed-function pipeline** 4

**flat shading**  
     about 75  
     implementing 75, 76

**fly-through experience** 495

**fog effect**  
     camera distance, computing 108  
     simulating 106-108

**forward compatible** 5

**fractal texture**  
     creating, compute shader used 362-365

**fragment shader**  
     about 54, 55  
     creating, GLSL used 457-466  
     discarding, to create perforated look 82-85

**fragment shader output** 29

**Framebuffer Object (FBO)** 114, 151, 264, 358

**Freeimage**  
     URL 118

**Fresnel equations** 144

**Frustum Shift** 529

**functions**  
     arrays, passing to 70  
     overloading 70  
     structures, passing to 70  
     using, in shaders 67-70

## G

**gamma correction**  
about 185  
used, for improving image quality 184-186

**Gaussian blur filter**  
about 167  
applying 167-173

**g-buffer** 192

**geometry shader (GS)**  
about 212, 213  
point sprites, drawing with 216-220  
used, for creating shadows 280-288  
used, for drawing silhouette lines 229-238

**Gimbal lock**  
URL 611

**GitHub**  
URL 52

**glBegin function** 399

**glCompileShader function** 16

**glCreateShader function** 17

**glDeleteShader function** 17

**glEnd function** 399

**GLEW**  
about 6  
URL 9

**GLFW**  
about 428  
interactive environment,  
creating with 428-435

**glfwWindowHint function** 448

**glGetError function** 45

**glGetIntegerv function** 12

**glGetString function** 11

**gl\_GlobalInvocationID variable** 344

**glLight function** 67

**GLLoadGen**  
about 6  
C++ loader, generating 8  
extensions, using 9  
no-load styles 9  
URL 6  
URL, for Downloading 6  
used, for accessing OpenGL  
functionality 6, 7

**gl\_LocalInvocationID variable** 344

## GLM

about 9  
URL 10  
used, for creating noise texture 291-294  
using 9, 10

## GLM types

using, as input to OpenGL 11

**gl\_NumWorkGroups variable** 344

## glOrtho function

about 400  
reference link 400

**glShaderSource function** 16

## GLSL

about 3, 4, 54  
GLSLanimation 311, 312  
GLSLparticle system 311, 312  
sampler binding, specifying 119  
shaders, compiling 13-17  
texture 113, 114

## GLSL version

determining 11-13

**GLSL version 4** 3

## glTexParameteri

URL 118

**glUniform function** 51

**gl\_WorkGroupID variable** 344

**gl\_WorkGroupSize variable** 344

## Gouraud shading

about 94  
differentiating, with Phong shading 94

**GPGPU (General Purpose Computing  
on Graphics Processing Units)** 4, 624

**Graphics Processing Units (GPUs)** 3

## H

## halfway vector

used, for ADS shading 97-99

## High Dynamic Range

**imaging (HDR imaging)** 174

## High Dynamic Range

**rendering (HDR rendering)** 174

## HDR lighting

implementing, with tone mapping 174-179

**homogeneous clip coordinates** 261

## I

**image load/store** 199  
**image quality**  
    improving, gamma correction used 184-186  
**immutable storage texture** 114  
**Inertial Measurement Units (IMUs)** 561  
**infinitePerspective function** 287  
**Inner level 0 (IL0)** 244  
**Inner level 1 (IL1)** 244  
**instanced attribute** 332  
**instanced particle**  
    used, for creating particle system 331-334  
**instanced rendering** 331  
**instance name**  
    using, with uniform blocks 43  
**interactive Android-based data visualization application**  
    creating, with mobile GPUs 587-599  
**interactive environment**  
    creating, with GLFW 428-435  
**interleaved arrays** 30  
**intraocular distance (IOD)** 529

## J

**Java Native Interface (JNI)** 548

## L

**layout qualifiers**  
    using 330  
    using, with uniform blocks 44, 45  
**level-of-detail (LOD) algorithm**  
    about 215  
    implementing, with tessellation  
        shader 255-257  
**light attenuation** 67  
**line segments**  
    drawing 401-403  
**link function** 51  
**Lua**  
    about 6  
    URL 6  
**luminance function** 166, 358

## M

**mandelbrot function** 364  
**MCML simulation** 436  
**Meta 1 Developer Kit**  
    URL 534  
**Microsoft Kinect 3D range-sensing camera** 490, 491  
**mobile GPUs**  
    interactive Android-based  
        data visualization application,  
        creating with 587-599  
**modeling transformation** 418  
**modern OpenGL**  
    about 448  
    working 448  
**Monte Carlo for multi-layered media (MCML)** 436  
**Monte Carlo (MC) method** 436  
**motion sensor inputs**  
    handling 581-586  
**multiple light sources**  
    shading with 88-90  
**multiple texture**  
    applying 120-123  
**multisample anti-aliasing**  
    about 187  
    using 187-191  
**multi-touch interface**  
    handling 581-586  
    reference link 586

## N

**night-vision effect**  
    creating 307-310  
**noise concept** 290  
**noise texture**  
    creating, using GLM 291-294  
**non-local viewer**  
    about 66  
    using 66  
**normal mapping** 126  
**normal maps**  
    using 126-132

**Nvidia**  
about 624  
references 146, 229

**O**

**object local coordinate system** 127

**Open Asset Import Library (Assimp)**  
about 510  
installing 510-512

**OpenCL** 341

**OpenCV**  
setting up, on Android 602-604  
used, for accessing camera  
live feed 604-611

**OpenCV, for Android SDK package**  
URL, for downloading 603

**OpenCV, in Android application**  
URL 604

**OpenCV libraries**  
setting up, in Mac OS X/Linux 454-456  
setting up, in Windows 449-452

**OpenGL**  
about 417  
GLM types, using 11  
shaders 53  
URL, for documentation 30

**OpenGL 3.2**  
compatibility profile 5  
core profile 5

**OpenGL 4** 113

**OpenGL 4.2** 114

**OpenGL 4.3** 45  
compute shader 341, 342

**OpenGL application binary interface (ABI)** 6

**OpenGL ES 3.0**  
URLs 560  
used, for creating Android application 548-560

**OpenGL ES 3.0 specifications**  
reference link 598

**OpenGL Extension Wrangler Library (GLEW)**  
about 447  
setting up, in Mac OS X/Linux 454-456

setting up, in Windows 449-452

**OpenGL for Embedded Systems (OpenGL ES)**  
about 537  
URL 541

**OpenGL functionality**  
accessing, GLLoadGen used 6, 7

**OpenGL Mathematics (GLM)**  
about 447  
setting up, in Windows 449-452  
setting up, Mac OS X/Linux 454-456

**OpenGL point cloud**  
rendering, with texture mapping and overlays 494-508

**OpenGL primitives**  
about 396  
line segments, drawing 401-403  
points, drawing 396-400  
triangles, drawing 403-405  
URL 401

**OpenGL Shading Languages (GLSL)** 418

**OpenGL version**  
determining 11-13

**OpenGL version 1.1** 6

**OpenGL Version 4.3**  
about 54  
fragment shader 54, 55  
vertex shader 54, 55

**OpenNI2**  
references 490

**Optical Coherence Tomography (OCT)** 486

**optimization technique** 166

**Order Independent Transparency (OIT)**  
about 199  
implementing 198-208

**P**

**paint-spatter effect**  
creating 305, 307

**particle simulation**  
implementing, with compute shader 345-349

**particle system**  
about 312  
creating 316-322

creating, instanced particle used 331-334  
creating, transform  
    feedback used 322-329  
fire, simulating with 335-337  
recycling 331  
smoke, simulating with 337-339  
**patch primitive** 213  
**percentage-closer filtering (PCF)**  
    about 270  
    used, for anti-aliasing  
        shadow edges 270-273  
**per-fragment shading**  
    for improved realism 94-96  
    vs, per-vertex shading 66  
**Perlin noise**  
    about 290, 291  
    URL 290, 291  
**perspective division** 262  
**perspective rendering**  
    3D plot, creating with 421-427  
**per-vertex shading**  
    about 75  
    implementing, with single  
        point light source 56-60  
    vs, per-fragment shading 66  
**Phong shading**  
    about 94  
    differentiating, with Gouraud shading 94  
**pixels**  
    discarding, alpha maps used 123-125  
**points**  
    drawing 396-400  
**point sprites**  
    drawing, with GS 216-220  
**Polarization-Sensitive Optical Coherence Tomography (PS-OCT)** 487  
**postprocessing options**  
    reference 518  
**primitives**  
    used, for creating 2D plot 406-409  
**printActiveAttribs function** 51  
**printActiveUniformBlocks function** 51  
**printActiveUniforms function** 51  
**procedural texture** 150  
**projected texture**  
    applying 145-150

**projection matrix (P)** 145  
**projection transformation** 418  
**projective texture mapping** 145  
**provoking vertex** 76

**R**

**radiative transport equation (RTE)** 436  
**random sampling**  
    soft shadow edges, creating with 274-280  
**rasterization** 418  
**raw data**  
    capturing, from depth-sensing  
        cameras 491-494  
**real-time data**  
    visualizing, from built-in Inertial Measurement Units (IMUs) 562-580  
**real-time video**  
    displaying, texture mapping used 612-624  
    rendering, with filters 479-487  
**real-time visualization, time series** 409, 410-412

**reflection**  
    simulating, with cube map 132-139  
**refraction**  
    about 139  
    on both sides of object 144  
    simulating, with cube map 139-143  
**render function** 328  
**ResIL**  
    URL 118  
**Runge-Kutta integration** 323

**S**

**sample qualifier** 191  
**sampler binding**  
    specifying, within GLSL 119  
**sampler objects**  
    using 155-157  
**sampler variable** 114  
**seamless noise texture**  
    creating 294-296  
**setUniform overloaded function** 51  
**shaded mesh**  
    wireframe, drawing on 221-229

**shader functionality**  
selecting, subroutine used 77-81

**shader object**  
deleting 17

**shader pipeline**  
about 212  
tessellation control shader (TCS) 212  
tessellation evaluation shader (TES) 212

**shader program**  
about 4  
deleting 20  
linking 17-19

**shaders**  
about 4, 53  
cloud-like effect, creating 296-298  
compiling 13-17  
data sending, uniform variables used 33-37  
data sending, vertex attributes used 21-26  
data sending, vertex buffer objects used 21-26  
disintegration effect, creating 302-304  
functions, using 67-70  
night-vision effect, creating 307-310  
noise texture, creating 291-294  
paint-splat effect, creating 305-307  
seamless noise texture, creating 294-296  
wood grain effect, creating 298-301

**shader storage buffer objects (SSBO)** 199

**shadeWithShadow function** 277

**shading**  
with directional light source 91-93  
with multiple light source 88-90

**shadow mapping**  
about 259  
back faces, rendering for 270  
shadows, rendering with 260-268

**shadows**  
about 259  
creating, geometry shader used 280-288  
creating, shadow volumes used 280-288  
rendering, with shadow mapping 260-268

**shadow volumes**  
about 281  
used, for creating shadows 280-288

**silhouette lines**  
drawing, GS used 229-238

**Simple OpenGL Image Loader (SOIL)**  
about 447  
setting up, in Windows 449-454

**Simultaneous localization and Mapping (SLAM)** 622

**skybox** 133

**smoke**  
simulating, with particle system 337-339

**Snells law** 140

**Sobel operator** 161

**soft shadow edges**  
creating, with random sampling 274-280

**specular component** 61

**specular highlights** 63

**spotlight** 99-102

**Stereolithography (.stl)** 510

**stereoscopic 3D rendering**  
about 529-532  
working 533-536

**structures**  
passing, to functions 70

**subroutine**  
about 77  
using, to select shader functionality 77-81

**surface**  
animating, with vertex displacement 312-315

## T

**tangent space** 127

**Terathon**  
URL 128

**tessellation control shader (TCS)** 212

**tessellation evaluation shader (TES)** 212

**tessellation primitive generator (TPG)** 214

**tessellation shader**  
about 213-215  
level-of-detail (LOD) algorithm, implementing with 255-257  
used, for drawing 2D quad 244-249  
used, for drawing 3D surface 249-254  
used, for drawing cubic Bezier curve 239-244

**texture**  
2D texture, applying 114-119  
about 113, 114

projected texture, applying 145-150  
rendering 150-154

**texture mapping**  
used, for displaying real-time video 612-624

**time series**  
real-time visualization 409-412

**tone mapping**  
about 174  
HDR lighting, implementing with 174-179

**Tone Mapping Operator (TMO)** 175

**toonShade function** 105

**toon shading**  
about 103  
creating 103-105

**transform feedback**  
about 311, 322  
results, querying 330  
used, for creating particle system 322-329

**triangle altitude** 223

**triangles**  
drawing 403-405

**two-sided rendering**  
using, for debugging 74

**two-sided shading**  
implementing 71-74

## U

**uniform blocks**  
instance name, using 43  
layout qualifiers, using 44, 45  
using 39-43

**uniform buffer objects**  
using 39-43

**uniform variables**  
listing 37, 38  
used, for sending data to shaders 33-37

**update function** 328

**use function** 51

## V

**Verlet integration** 323

**vertex**  
creating, GLSL used 457-465

**vertex array object (VAO)** 26, 475

**vertex attributes**  
format 27, 28  
used, for sending data to shaders 21-26

**vertex buffer objects**  
used, for sending data to shaders 21-26

**Vertex Buffer Objects (VBOs)** 446

**vertex displacement**  
surface, animating with 312-315

**vertex shader** 54, 55

**VideoRenderer class**  
functions 623

**viewing transformation** 418

**view matrix (V)** 145

**virtual camera**  
setting up, for 3D rendering 418-421

**Virtual Reality (VR)** 534

**volumetric dataset**  
rendering 436-446

## W

**Wavefront Object (.obj)**  
3D model, loading in 512-517  
about 510

**wireframe**  
drawing, on shaded mesh 221-229

**wood grain effect**  
creating 298-301

**work groups** 342, 343

## Z

**Z-fighting** 420

**z-pass technique** 288





Thank you for buying  
**OpenGL – Build high performance graphics**

## 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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto: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, then please 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.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles