OpenGL Data Visualization Cookbook

This book provides a series of easy-to-follow, hands-on tutorials to create appealing OpenGL-based visualization tools with minimal development time. We will first illustrate how to quickly set up the development environment in Windows, Mac OS X, and Linux. Next, we will demonstrate how to visualize data for a wide range of applications using OpenGL, starting from simple 2D datasets to increasingly complex 3D datasets with more advanced techniques. Each chapter addresses different visualization problems encountered in real life and introduces the relevant OpenGL features and libraries in a modular fashion.

By the end of this book, 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 this book will do for you...

- Install, compile, and integrate the OpenGL pipeline into your own project
- Create interactive applications using GLFW to handle user inputs and the Android Sensor framework to detect gestures and motions on mobile devices
- Use OpenGL primitives to plot 2-D datasets such as time series dynamically
- Render complex 3D volumetric datasets with techniques such as data slicers and multiple viewpoint projection
- Render images, videos, and point cloud data from 3D range-sensing cameras using the OpenGL Shading Language (GLSL)
- Develop video see-through augmented reality applications on mobile devices with OpenGL ES 3.0 and OpenCV
- Visualize 3D models with meshes and surfaces using stereoscopic 3D technology

$ 49.99 US
£ 31.99 UK

Prices do not include local sales tax or VAT where applicable

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Quick answers to common problems

Over 35 hands-on recipes to create impressive, stunning visuals for a wide range of real-time, interactive applications using OpenGL

In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 4 'Rendering 2D Images and Videos with Texture Mapping'
- A synopsis of the book’s content
- More information on OpenGL Data Visualization Cookbook
About the Authors

**Raymond C. H. Lo** is currently the CTO and cofounder of Meta (http://www.getameta.com), a company in Silicon Valley that is creating the world's first augmented reality eyeglasses with 3D gesture input and 3D stereoscopic display. This next-generation wearable computing technology, which is the result of his PhD research, has been featured extensively in news media, including CNN, MIT News, CNET, and Forbes magazine. During his PhD, Raymond worked with Professor Steve Mann, who is widely recognized as the father of wearable computing. Together, they published and presented papers at leading conferences, including the SIGGRAPH and IEEE conferences, on real-time high-dynamic-range (HDR) imaging, augmented reality, and digital eyeglasses, which involve high-performance computation using CUDA and visualization using OpenGL.

**William C. Y. Lo** is currently an MD-PhD candidate at Harvard Medical School. He is pursuing his PhD degree in the joint Harvard-MIT Medical Engineering and Medical Physics program under the guidance of Professor Brett Bouma (and co-advisor Professor Benjamin Vakoc) at Massachusetts General Hospital, who founded the NIH-funded Center for Biomedical OCT Research and Translation. He obtained his bachelor of applied science degree in computer engineering and his MSc degree in medical biophysics from the University of Toronto, where he worked with Professor Lothar Lilge and Professor Jonathan Rose on high-performance computing for photodynamic therapy planning using custom FPGA hardware and graphics processors with CUDA.

He, along with J. Rose and L. Lilge, worked on *Computational Acceleration for Medical Treatment Planning: Monte Carlo Simulation of Light Therapies Accelerated using GPUs and FPGAs*, VDM Verlag, 2010.
OpenGL is a multiplatform, cross-language, and hardware-accelerated application programming interface for the high-performance rendering of 2D and 3D graphics. An emerging use of OpenGL is the development of real-time, high-performance data visualization applications in fields ranging from medical imaging, simulation or modeling in architecture and engineering, to cutting-edge mobile/wearable computing. Indeed, data visualization has become increasingly challenging using conventional approaches without graphics hardware acceleration as datasets become larger and more complex, especially with the evolution of big data. From a mobile device to a sophisticated high-performance computing cluster, the 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 book contains a series of hands-on recipes that are tailored to both beginners who have very little experience with OpenGL and more advanced users who would like to explore state-of-the-art techniques. We begin with a basic introduction to OpenGL in chapters 1 to 3 by demonstrating how to set up the environment in Windows, Mac OS X, and Linux and learning how to render basic 2D datasets with primitives, as well as more complex 3D volumetric datasets interactively. This part requires only OpenGL 2.0 or higher so that even readers with older graphics hardware can experiment with the code. In chapters 4 to 6, we transition to more advanced techniques (which requires OpenGL 3.2 or higher), such as texture mapping for image/video processing, point cloud rendering of depth sensor data from 3D range-sensing cameras, and stereoscopic 3D rendering. Finally, in chapters 7 to 9, we conclude this book by introducing the use of OpenGL ES 3.0 on the increasingly powerful mobile (Android-based) computing platform and the development of highly interactive, augmented reality applications on mobile devices.

Each recipe in this book gives readers a set of standard functions that can be imported to an existing project and can form the basis for the creation of a diverse array of real-time, interactive data visualization applications. This book also utilizes a set of popular open-source libraries, such as GLFW, GLM, Assimp, and OpenCV, to simplify application development and extend the capabilities of OpenGL by enabling OpenGL context management and 3D model loading, as well as image/video processing using state-of-the-art computer vision algorithms.
Preface

What this book covers

Chapter 1, **Getting Started with OpenGL**, introduces the essential development tools required to create OpenGL-based data visualization applications and provides a step-by-step tutorial on how to set up the environment for our first OpenGL demo application in Windows, Mac OS X, and Linux.

Chapter 2, **OpenGL Primitives and 2D Data Visualization**, focuses on the use of OpenGL 2.0 primitives, such as points, lines, and triangles, to enable the basic 2D visualization of data, including time series such as an electrocardiogram (ECG).

Chapter 3, **Interactive 3D Data Visualization**, builds upon the fundamental concepts discussed previously and extends the demos to incorporate more sophisticated OpenGL features for 3D rendering.

Chapter 4, **Rendering 2D Images and Videos with Texture Mapping**, introduces 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.

Chapter 5, **Rendering of Point Cloud Data for 3D Range-sensing Cameras**, introduces the techniques used to visualize another interesting and emerging class of data—depth information from 3D range sensing cameras.

Chapter 6, **Rendering Stereoscopic 3D Models using OpenGL**, demonstrates how to visualize data with stunning stereoscopic 3D technology using OpenGL. OpenGL does not provide any mechanism to load, save, or manipulate 3D models. Thus, to support this, we will integrate a new library named Assimp into our code.

Chapter 7, **An Introduction to Real-time Graphics Rendering on a Mobile Platform using OpenGL ES 3.0** transitions to an increasingly powerful and ubiquitous computing platform by demonstrating how to set up the Android development environment and create the first Android-based application on the latest mobile devices, from smartphones to tablets, using OpenGL for Embedded Systems (OpenGL ES).

Chapter 8, **Interactive Real-time Data Visualization on Mobile Devices**, demonstrates how to visualize data interactively by using built-in motion sensors called Inertial Measurement Units (IMUs) and the multitouch interface found on mobile devices.

Chapter 9, **Augmented Reality-based Visualization on Mobile or Wearable Platforms**, introduces the fundamental building blocks required to create 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.
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:

```c
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:

```cpp
#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
cmake_minimum_required (VERSION 2.8)
set(CMAKE_CONFIGURATION_TYPES Debug Release)
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*:

![CMake 3.2.1 - C:/Users/raymond/Desktop/ch4/code/code_s...](image)

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](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](http://sourceforge.net/projects/ogl-math/files(glm-0.9.5.4/glm-0.9.5.4.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 `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 `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:
   
   ```bash
   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:
      
      ```bash
      unzip opencv-2.4.9.zip
      ```

   2. Build the OpenCV library using CMake:
      
      ```bash
      cd opencv-2.4.9/
      mkdir build
      cd build
      cmake ../
      make && sudo make install
      ```

   3. Configure the library path:
      
      ```bash
      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:
      
      ```bash
      CFFILES = ../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: $(CFFILES) $(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:

- **GLM**: [http://glm.g-truc.net/0.9.5/index.html](http://glm.g-truc.net/0.9.5/index.html)
- **SOIL**: [http://www.lonesock.net/soil.html](http://www.lonesock.net/soil.html)
- **OpenCV**: [http://opencv.org/](http://opencv.org/)
- **MacPorts**: [http://www.macports.org/](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:

![Processing Pipeline Diagram](image)

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:

```glsl
#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:

```cpp
#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`:

```cpp
#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:

   ```cpp
   #include <iostream>
   #include <fstream>
   #include <algorithm>
   #include <vector>
   #include "shader.hpp"
   ```

2. Implement the `readSourceFile` function as follows:

   ```cpp
   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:

```cpp
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
", &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`:

```cpp
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:

```c
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:

```c
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" );
    }
```
glfTerminate();
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, "Failed 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");
gUseProgram(program);

8. Set up Vertex Buffer Object (and color buffer) and copy the vertex data to it:

GLuint vertex_buffer;
GLuint color_buffer;
gGenBuffers(1, &vertex_buffer);
gGenBuffers(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);
gBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);
9. Specify the layout of the vertex data:

```c
GLint position_attrib = glGetAttribLocation(program, "position");
glEnableVertexAttribArray(position_attrib);
BindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
VertexAttribPointer(position_attrib, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

GLint color_attrib = glGetAttribLocation(program, "color_in");
glEnableVertexAttribArray(color_attrib);
BindBuffer(GL_ARRAY_BUFFER, color_buffer);
VertexAttribPointer(color_attrib, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

10. Run the drawing program:

```c
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:

```c
//clean up the memories
glDisableVertexAttribArray(position_attrib);
glDisableVertexAttribArray(color_attrib);
deleteBuffers(1, &vertex_buffer);
deleteBuffers(1, &color_buffer);
deleteVertexArrays(1, &vertex_array);
deleteProgram(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`).

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.cpp` and `controls.hpp`). 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`:

```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:

```cpp
#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:

```cpp
#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:
   ```cpp
   #include "texture.hpp"
   #include <SOIL.h>
   ```

2. Define the initialization of texture objects and set up all parameters:
   ```cpp
   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:

```c
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 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);
}
```

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:

```c
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:

   ```c
   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:

```cpp
#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:

```c
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:

```c
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:

```c
int main(int argc, char **argv)
{
    //Initialize the GLFW
    if(!glfwInit()){
        fprintf( stderr, "Failed to initialize GLFW
"");
        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, "Failed 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:

```c
//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):

```c
GLuint vertex_array_id;
glGenVertexArrays(1, &vertex_array_id);
glBindVertexArray(vertex_array_id);
```

11. Define our VAO for vertices and UV mapping:

```c
//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:

```c
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
    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 recomputed 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:

![Screenshot of Image Viewed with Different Perspective](image.png)

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:

\[
G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \ast I(x, y), \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \ast I(x, y)
\]
$G_x$ 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:

```cpp
#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:
   ```cpp
   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 = 0.0f;
        g = 4.0f * (vmin + 0.25f * dv - v) / dv;
        b = 1.0f + 4.0f * (vmin + 0.25f * dv - v) / dv;
    } else if (v < (vmin + 0.75f * dv)) {
\[
\begin{align*}
    r &= 4.0f \times (v - vmin - 0.5f \times dv) / dv; \\
    b &= 0.0f; \\
\end{align*}
\] else {
    \[
\begin{align*}
    g &= 1.0f + 4.0f \times (vmin + 0.75f \times dv - v) / dv; \\
    b &= 0.0f; \\
    \end{align*}
\]}
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:

```cpp
void main()
{
    // compute the results of Sobel filter
    float graylevel = sobel_filter();
    color = vec4(graylevel, graylevel, graylevel, 1.0);
}
```

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:
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):

```cpp
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:

![Image of volumetric dataset]
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.
Where to buy this book

You can buy OpenGL Data Visualization Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.