OpenGL is one of the most popular rendering SDKs used to develop games. OpenGL has been used to create everything from 3D masterpieces running on desktop computers to 2D puzzles running on mobile devices.

This book is a complete tutorial on designing and coding games from the setup of the development environment to the final credits screen, through the creation of a 2D and 3D game.

The book starts off by showing you how to set up a development environment using Visual Studio and create a code framework for your game. It then walks you through the creation of two games. You'll create sprite classes, render sprites and animation, and navigate and control the characters. You will also learn how to implement input, use audio, and code basic collision and physics systems. The book will take you through the complete journey of creating a game engine that you can extend to create your own games.

Who this book is written for
If you are a prospective game developer with some experience using C++, then this book is for you. Both prospective and experienced game programmers will find nuggets of wisdom and practical advice as they learn to code two full games using OpenGL, C++, and a host of related tools.

What you will learn from this book
- Set up your development environment in Visual Studio using OpenGL
- Use 2D and 3D coordinate systems
- Implement an input system to handle the mouse and the keyboard
- Create a state machine to handle complex changes in the game
- Load, display, and manipulate both 2D and 3D graphics
- Implement collision detection and basic physics
- Discover the key components needed to complete a polished game
- Handle audio files and implement sound effects and music

In this package, you will find:

- The author’s biography
- A preview chapter from the book, Chapter 2 'Your point of view'
- A synopsis of the book’s content
- More information on OpenGL Game Development By Example
Robert Madsen is an accomplished game programmer, with dozens of published games to his credit. He started programming in 1979, and he has been a programmer for all of his professional life. He entered the game industry in 2004, and he founded SynapticSwitch, LLC in 2010. As studio director, he continues to code while also managing the broader needs of an independent game development studio.

Stephen Madsen completed his degree in game development from Full Sail Real World Education in 2007, beginning his first job as a game programmer in 2008. He then joined SynapticSwitch, LLC as the lead software engineer in 2012. He has developed and published many titles on the mobile, console, and personal computer platforms with OpenGL being the foundational rendering technology for most of these platforms.
Welcome to *OpenGL Game Development Blueprints*! We are excited that you chose this book as your guide to both OpenGL and game development. This section will provide you with a brief preview of each chapter, followed by the technologies that are required to complete the work that is presented in the book. Finally, we will discuss the target audience for this book so that you will know whether this book is right for you.

**What this book covers**

*Chapter 1, Building the Foundation*, guides you through creating the code framework for the game. Games use a particular structure that is known as the *game loop*. By the end of this chapter, you will understand and have created the game loop for the game as well as initialized the required OpenGL elements.

*Chapter 2, Your Point of View*, introduces you to the first project in the book—creating a 2D platform game. The first step in this project will be to define the type of view that is required by OpenGL, and render the background of the game.

*Chapter 3, A Matter of Character*, covers the creation of sprites that move on the screen. 2D frame-based animations are the core of any 2D game, and you will learn how to create simple graphics and render them to the screen.

*Chapter 4, Control Freak*, teaches you how to build an input system that will allow you to control the main character and other aspects of the game. You will also create a basic user interface that allows you to start the game and navigate to various options.

*Chapter 5, Hit and Run*, covers collision detection. You will learn how to stop the character from falling through the ground, how to land on objects, and how to detect whether enemies have hit you or have been hit by player weapons. By the end of this chapter, you will be able to play the game for the first time.
Preface

Chapter 6, *Polishing the Silver*, covers the topics that make a game presentable (but are often overlooked by novice developers). You will learn how to implement a scoring system, game over and game won scenarios, and simple level progression. This chapter will conclude the 2D project of the book.

Chapter 7, *Audio Adrenaline*, guides you through implementing sound effects and music in the game. We will provide links to some audio files that you can use in your game.

Chapter 8, *Expanding Your Horizons*, will start the second project of the book—a 3D first-person space shooter. At the end of this chapter you will have created a new project, starting the framework for a 3D game.

Chapter 9, *Super Models*, introduces you to the concepts of 3D art and modeling, and then guides you through the process of loading 3D models into the game environment. Although you will be able try your hand at creating a 3D model, the resources that are required for the game will be provided online.

Chapter 10, *Expanding Space*, expands on many of the concepts that were covered in the 2D segment of the book and applies them to a 3D world. Movement and collision detection are revamped to take this new dimension into consideration. An input scheme to move in 3D space is implemented. By the end of this chapter, you will be able to control a 3D model in 3D space.

Chapter 11, *Heads Up*, guides you through creating a 2D user interface on top of the 3D world. You will create a menu system to start and end the game, as well as a heads-up-display (HUD) that shows the score and stats in game. By the end of this chapter, you will have created a playable 3D shooter game.

Chapter 12, *Conquer the Universe*, introduces you to some of the more advanced concepts that were beyond the scope of the book, and it gives you some direction to advance your skills.
Imagine that you are making a video. You've got your cell phone out, and you point it at the area that you want to shoot and press record. You're taking a video of the Grand Canyon, so you have to pan the camera around to get the whole scene in. Suddenly, a bird flies past the field of view, and you've captured the whole scene.

The preceding scenario is pretty much how games work as well. The game has a virtual camera that can be positioned and even moved around. Similarly to the video camera on your cell phone, the game camera can only see a part of the game world, so sometimes you have to move it around. Any game objects that move in front of the camera will be seen by the player.

This chapter will explain how things are rendered in the game. Rendering is the process of actually displaying images on the screen. In order to get your game onto the screen, you will need to have a solid understanding of the following terms:

- **Coordinate systems**: The coordinate system is the reference that allows you to position objects in the game
- **Primitives**: Primitives are the fundamental building blocks of the images that you see on screen, and OpenGL was designed to work with them
- **Textures**: Textures are image files that are used to give the objects in your game a realistic appearance

By the time you have read this chapter, you will understand how to use images to build your game world and display it on the screen.
Plotting your revenge

Okay, so you're not really plotting your revenge. But you are plotting everything in your game as if you were putting it all down on a piece of graph paper. Remember high-school geometry? You got out your graph paper, drew a couple of lines for the X and Y axis, and the plotted points on the graph. OpenGL works in pretty much the same way.

The OpenGL coordinate system

The OpenGL coordinate system is a standard X and Y axis system that you have most likely learned all your life. You can conceptualize (0, 0) as being the center of the screen.

Let's say that we want to display a moving car on the screen. We could start by plotting our car at position (5, 5) in the coordinate plane. If we then moved the car from (5, 5) to (6, 5), then (7, 5), and so forth, the car would move to the right (and eventually leave the screen), as illustrated in the following figure:

We haven't been completely honest with you. Since OpenGL is a 3D rendering engine, there is actually one more axis called the Z-axis that we haven't discussed. As this part of the book focuses on 2D game programming, we will ignore the Z axis for now.
Making your point

As we learn each concept, we will actually write code to demonstrate each point. Speaking of points, we will write code to plot points using OpenGL.

We are going to set this project up as a separate project from the actual game. We will use this project to demonstrate how to code basic OpenGL tasks. To keep this thing as simple as possible, this project will be created as a console project in Visual Studio. A console project doesn't have many of the features of a full-blown Windows project and therefore, the setup code is much smaller.

Start Visual Studio and create a new project. For the project template, choose Win32 Console Application from the Visual C++ group of templates. Name the project OpenGLFun, and click OK. Click Finish to complete the project wizard.

You should notice that the code is much simpler than the code that was created in the previous chapter for a full-blown Windows application. We will return to using the more complicated code as we continue building the game.
Once you have the project created, type following the code into the code window:

```cpp
#include "stdafx.h"
#include <windows.h>
#include "glut.h"

void initGL() {
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

drawPoints() 
{
  glBegin(GL_POINTS);
  glColor3f(1.0f, 1.0f, 1.0f);
  glVertex2f(0.1f, -0.6f);
  glVertex2f(0.7f, -0.6f);
  glVertex2f(0.4f, -0.1f);
  glEnd();
}

void update()
{
  glClear(GL_COLOR_BUFFER_BIT);
  drawPoints();
  glFlush();
}

int _tmain(int argc, _TCHAR* argv[])
{
  glutCreateWindow("GL Fun");
  glutInitWindowSize(320, 320);
  glutInitWindowPosition(50, 50);
  glutDisplayFunc(update);
  initGL();
  glutMainLoop();
  return 0;
}
Chapter 2

Understanding the code
As we will be using the code to demonstrate the fundamentals of OpenGL, we will look at it in detail so that you understand what the code is doing.

Header files
This code uses three header files:

- \texttt{stdafx.h}: This header file loads the precompiled header that was created by Visual Studio when we created the project
- \texttt{windows.h}: This header file allows the window that renders the OpenGL content to be created
- \texttt{glut.h}: This header file allows us to use the OpenGL Utility Toolkit, which simplifies the setup and use of OpenGL

You will need to download the GLUT files and place them in your project folder. Download the files from \url{http://user.xmission.com/~nate/glut/glut-3.7.6-bin.zip}. Open the zipped file and copy \texttt{glut.h}, \texttt{glut32.dll}, and \texttt{glut32.lib} into the folder that contains your source code. You may have to add glut.h to your project (right-click on \texttt{Header files} | \texttt{Add} | \texttt{Existing item}).

Initializing OpenGL
You will notice a function called \texttt{initGL}. This function currently contains a single line of code whose sole purpose is to set the background color of the screen at the start of each frame. This is often referred to as the \textit{clear color} because it is the default that OpenGL clears the background to before it begins to render additional items:

\begin{verbatim}
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
\end{verbatim}

The four numbers inside the parenthesis define the color, and the opacity of the color. The first three numbers represent the amount of red, green, and blue (RGB) that will be used to create the color. The fourth number represents the opacity (or seen another way, the transparency) of the color. This is also referred to as the alpha channel (RGBA). The values above create a black background that is 100 percent opaque.

All values in OpenGL have a range from 0 to 1. This means that there will be many decimal values, known in C++ as floats. Thus, the range in C++ lingo is from \texttt{0.0f} to \texttt{1.0f}. 

\[ \text{[23]} \]
C++ is different from many languages, which use integers or even hexadecimal numbers to express their ranges. For example, many other languages use a range of 0 to 255 for each color component. In these cases, integer 0 corresponds to 0.0f, and integer 255 corresponds to 1.0f.

To convert an integer of range 0 to 255 to OpenGL's system, use the formula \((1/255) \times \text{value}\), where value is the integer value you are trying to convert. Thus, to convert the number 50, you would calculate \((1/255) \times 50\), which results in 0.1096.

The main entry point

Every program has to have a starting point, known as the entry point. In our program, this is the \_tmain function. We put this at the very end because C++ expects the functions that are being used to have been defined before the function that calls them. There are various tricks around this, but we'll keep our examples simple and just always define \_tmain as the last function in the code.

When we start the program, there are a few things that have to be done to set up the environment to render OpenGL. Here is the anatomy of the \_tmain function:

- \glutCreateWindow("GL Fun"): This function creates the window that will render the OpenGL content. We include the name of the program as a parameter.
- \glutInitWindowSize(320, 320): This function initializes the size of the window. We have specified 320 pixels by 320 pixels. Feel free to try larger (or smaller) window sizes.
- \glutInitWindowPosition(50, 50): This function sets the position of the window's upper-left corner in relation to the device's screen. In this case, the window will start drawing 50 pixels from the left and 50 pixels from the top of the screen. Feel free to try other positions.
- \glutDisplayFunc(update): Remember the previous chapter where we talked about the game loop? The game loop is the part of the program that runs over and over again (that is, every frame). We need to tell GLUT the name of the function that we want to run every frame. In this case, we are telling GLUT to use a function named update (described in the next section).
- \initGL(): This simply calls the \initGL function that we described earlier.
- \glutMainLoop(): This function starts the main game loop, which in turn will call our update function every frame. This essentially starts our program, which will run in an infinite loop until we close the program.
• return 0: This line is required by the _tmain function. It basically tells our system that the program has exited and everything is okay. This line of code won't run until we exit the program.

The update function
The update function is called every frame. Any work that we want to do will have to be coded in this function. The update function currently has three lines of code:

• glClear(GL_COLOR_BUFFER_BIT): The glClear function resets the render buffer to the color that was specified earlier by the glClearColor function. The render buffer is a separate location in the memory where OpenGL renders objects before they are displayed on the screen. Later, when all of the render operations are completed, the contents of the buffer are displayed on the screen in one fast transfer.

• drawPoints(): This is a function that we wrote to display three points on the screen. Later, we will replace this line of code to draw other objects. This function is described in the next section.

• glFlush(): This function flushes the OpenGL buffer, including the back buffer that currently holds our render. As a result, the rendering buffer is flushed, and all of the contents are rendered to the device screen.

OpenGL uses two buffers to draw. One is the screen buffer, which is what the player currently sees on the computer display. The other is the back buffer, which is where we create the objects that we intend to render in the next frame. Once we are done creating the render in the back buffer, we quickly swap the contents of the back buffer onto the current screen. This occurs so quickly that the player cannot detect the swap.

Drawing the points
The drawPoints function does the actual work of determining what to draw, and where to draw it. Here is what each line of code does:

• glBegin(GL_POINTS): The call to glBegin tells OpenGL to prepare to render items to the screen. We also tell OpenGL what we want to render. In our example, we are directing OpenGL to interpret the data that we send it as individual points. Later, we will learn to render other objects, such as triangles using GL_TRIANGLES, or rectangles using GL_QUADS.

• glColor3f(1.0f, 1.0f, 1.0f): As the name suggests, glColor sets the color of the item that is going to be rendered. Remember, OpenGL uses the RGB color system, so the color will be white (0, 0, 0 specified black).
• glVertex2f(0.1f, -0.6f): Each point in OpenGL is known as a vertex. This code tells OpenGL to render a single point at the coordinates (0.1, -0.6). In this case, zero means the center of the screen, and one means one unit from the center. The settings for the camera determine exactly how far one unit from the center actually is on the screen. There are three glVertex calls in our example code, one for each of the points that we want to render to the screen.

The names of OpenGL functions give you a clue as to how to use the function. For example, glVertex2f means that this function takes 2 parameters and they will be of type float. In comparison, the glVertex3f function takes three parameters of type float.

• glEnd(): Just like all good things must come to an end, we have to tell OpenGL when we are done rendering. That is the purpose of the call to glEnd.

You have probably noticed a lot of the use of the lower case letter f; this stands for float, meaning that a number that may contain a part after the decimal point (as opposed to an integer, which is always a whole number). So, a number, such as 0.0f, is telling C++ to treat the number zero as a floating point number. OpenGL uses a similar naming convention for its functions. For example, the function glVertex2f indicates that the function requires two floating point numbers (in this case, the x and y coordinates of the point to render).

Running the program
Now that you have entered your code, it's time to see it in action. When you run the program (Debug | Start Debugging), here is what you will see:
You’ll have to look at it closely, but if all went well, you should see three white points in the lower-right area of the screen. Congratulations! You have rendered your first OpenGL objects!

Hopefully, you have been able to follow the code. Think of _tmain as a manager that controls the program by setting everything up and then calling the main loop (just like we will do in our game). Then GLUT takes over and calls the update function every frame. The update function initializes the render buffer, draws objects to the render buffer, and then transfers the contents of the render buffer to the screen. In a game running at 60 frames per second, this entire operation will happen 60 times a second!

**Stretching your point**

Let's see how easy it will be to modify GLFun to draw other objects. This time we will draw two lines. Add the following function to your code just under the drawPoints function:

```c
void drawLines()
{
    glBegin(GL_LINES);

    glColor3f(1.0f, 1.0f, 1.0f);
    glVertex2f(0.1f, -0.6f);
    glVertex2f(0.7f, -0.6f);

    glEnd();
}
```
Your Point of View

```c
    glVertex2f(0.7f, -0.6f);
    glVertex2f(0.4f, -0.1f);
    glEnd();
```

Next, go to the update function and replace `drawPoints` with a call to `drawLines`. The new `update` function will look like this:

```c
void update()
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawLines();
    glFlush();
}
```

You will notice that there are four `glVertex` calls. Each pair of vertices sets the beginning and ending points of a line. As there are four points defined, the result is that two lines are drawn.
Getting primitive
Basic objects, such as points and lines, are called primitives. It would be pretty
difficult to create everything out of points and lines, so OpenGL defines other
primitive shapes that you can use to create more complicated objects.

In this section, we will dig a little under the hood and find out how OpenGL actually
creates more realistic images on your screen. It may surprise you that a single,
geometric figure is used to create everything from the simplest to the most complex
graphics. So, roll up your sleeves and get ready to get a little greasy.

A triangle by any other name
Have you ever seen a geodesic dome? Although the dome appears to be spherical, it
is actually built out of a combination of triangles. It turns out that triangles are very
easy to put together in such a way that you can add a slight amount of curvature
to the object. Each triangle can be attached at a slight angle to the others, allowing
you to create a dome made out of flat triangles. Also, consider this: the smaller the
triangle, the more convincing the end result!
The basic unit that is used to draw all modern graphics is the humble triangle. Graphic cards have been specifically engineered to be able to draw triangles—really small triangles—really fast. A typical graphics card can draw millions of triangles every second. Higher end cards reach billions of triangles per second.

Remember when we drew points and lines earlier? Each point had one vertex, and each line had two vertices. Of course, each triangle has three vertices.

**A primitive example**

It's time to take a look at some code in action. Add the following code after the `drawLines` function in the GLFun project:

```c
void drawSolidTriangle()
{
    glBegin(GL_TRIANGLES);

    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(0.1f, -0.6f);
    glVertex2f(0.7f, -0.6f);
    glVertex2f(0.4f, -0.1f);

    glEnd();
}
```

Then change the middle line of the `update` function to call `drawSolidTriangle`:

```c
void update()
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawSolidTriangle();
    glFlush();
}
```
Run the program, and you will see the following output:

![Triangle](image)

You may notice a similarity between the code for `drawSolidTriangle` and `drawPoints`. Look closely at the code, and you will see that the three `glVertex` functions define the same three points. However, in this case we told OpenGL to draw triangles, not points. You should also take a look at the code and make sure you understand why the triangle is rendered blue.

Let's take one more example. Add the following code below the `drawSolidTriangle` function:

```c
void drawGradientTriangle()
{
    glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);
    glVertex2f(0.3f, -0.4f);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(0.9f, -0.4f);
    glColor3f(0.0f, 0.0f, 1.0f);
    glVertex2f(0.6f, -0.9f);
    glEnd();
}
```
Be sure to change the middle line in update to call `drawGradientTriangle`:

```java
void update()
{
    glClear(GL_COLOR_BUFFER_BIT);
    drawGradientTriangle();
    glFlush();
}
```

Run the program, and this is what you will see:

You will immediately notice that this triangle is filled with a gradient instead of a solid color. If you look closely at the code, you will see that a different color is being set for each vertex. OpenGL then takes care of interpolating the colors between each vertex.

### From triangles to models

Triangles can be put together in an infinite number of ways to form almost any shape imaginable. It is important to understand that triangles are just geometry. Triangles are used to build the shape of your object. We call these shapes models.
Building a model using a single triangle at a time would be very time consuming, so 3D graphics programs, such as Maya and Blender, allow you to create models out more complex shapes (which are themselves built out of triangles). These models can then be loaded into your game and rendered by OpenGL. OpenGL literally sends a the list of points to form these triangles directly to the video card, which then creates and image out of them on the screen. We will see this process in action when we begin to deal with 3D game design.

Introducing textures

Images in games are called textures. Textures allow us to use real world images to paint our world. Think about what it would take to create a dirt road. You could either color the triangles in exactly the right way to make the overall scene look like dirt, or you could apply an actual image (that is, a texture) of dirt to the triangles. Which of these do you think would look more realistic?

Using textures to fill the triangles

Let's say that you are going to paint your bedroom. You can either use paint to color the walls, or you could buy some wallpaper and put that on your walls. Using images to add color to our triangles is pretty much like using wallpaper to color our bedroom walls. The image is applied to the triangle, giving it a more complex appearance than what could be created by color alone:

When we want to get really tricky, we use textures to fill the inside of our triangles instead of colors. A marble texture has been applied to the triangle in the preceding image. You could imagine using this technique to create a marble floor.
Remember the car we were working with before? It didn't look much like a triangle, did it? In fact, many real-world objects look more like rectangles than triangles:

![Car and Tree](image)

It turns out that all the textures that we use in games are actually rectangles. Imagine that the car that we have been dealing with is actually embedded inside an invisible rectangle, depicted in the following image as light gray:

![Car](image)

Most graphic programs use a checkerboard background to indicate the areas of the image that are transparent.

![Checkerboard Car](image)

Using rectangles for all of our shapes solves one big problem that you might not have thought of earlier. If you recall, it was very important to position the car at exactly (5, 5). To do so, we decided to place the bottom-left corner of the car at point (5, 5).
Looking at the car, it is actually a little difficult to figure out exactly where the bottom-left corner would be. Is it the lower left corner of the bumper, the tire, or somewhere else?

By embedding the car inside of a rectangle, as we just discussed, the problem is immediately solved.

**A matter of reference**

When working with a texture, it is very important to know what point is being used as a reference, usually known as the pivot point. In the following images, a black dot is used to represent the pivot point. The pivot point affects two critical issues. First, the pivot point determines exactly where the image will be placed on the screen. Second, the pivot point is the point on which the image will pivot when rotated.

Compare the two scenarios depicted in the following images:
The pivot point for the car in the preceding image has been set to the bottom-left corner of the image. The car has been rotated 90 degrees counter-clockwise.

The pivot point for the car in the preceding image has been set to the center of the image. The car has been rotated 90 degrees counter-clockwise. Notice how the pivot point affects not only how the car is rotated but also its final position in relation to its original position after the rotation is completed.

**Hanging out in the quad**

So, are you confused yet? First, I tell you that the most basic shape used to create images is a triangle, and then I tell you that all textures are actually rectangles. Which one is it?

Just then, your high-school geometry teacher silently walks into the room, goes up to the chalkboard that just magically appeared on your wall, and draws something like the following diagram:

Of course! You suddenly realize that two triangles can be fit together to form a rectangle. In fact, this arrangement is so useful that we have given it a name: **quad**.

When it comes to 2D graphics, the quad is the king.
Coding the quad

It's time to take a look at some code. Add the following code beneath the drawGradientTriangle function in GLFun:

```c
void drawQuad()
{
    glBegin(GL_QUADS);
    
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex2f(0.1f, -0.1f);
    glVertex2f(0.1f, -0.6f);
    glVertex2f(0.6f, -0.6f);
    glVertex2f(0.6f, -0.1f);
    
    glEnd();
}
```

As usual, change the middle line in update to call drawQuad. Run the program, and you will get a pretty green square, er quad! It's important to note that the points are defined in order starting from the upper-left corner and then moving counter-clockwise in order.
The order that the points are defined in is known as *winding*. By default, a counter-clockwise winding tells OpenGL that the side facing out is the side that is considered the front. This helps determine, among other things, whether this face should be lit, and it becomes even more significant when we begin working in 3D. As it turns out, GLUT simplifies our life so that it doesn't matter if we use clockwise or counter-clockwise winding when using GLUT.

**Rendering a texture**

Rendering a texture consist of two steps: loading the image and rendering the image using an OpenGL primitive. Our final achievement in this chapter will be to modify GLFun so that it will render a texture using a quad.

**Loading the texture**

Our first step is to create a function to load a texture. As it turns out, this isn't all that easy. So, I'm going to give you the code for a function that loads a 24-bit BMP file, and we'll treat it like a black box that you can use in your own code.

Add this code to the top of your existing **GLFun** code:

```c
GLuint texture;
#pragma warning(disable: 4996)
bool loadTexture(const char* filename)
{
    unsigned char header[54];
    unsigned char* data;
    int dataPos;
    int width;
    int height;
    int imageSize;

    FILE * file = fopen(filename, "rb");
    if (!file) return false;
    if (fread(header, 1, 54, file) != 54) return false;
    if (header[0] != 'B' || header[1] != 'M') return false;
    dataPos = *(int*)&(header[0x0A]);
    imageSize = *(int*)&(header[0x22]);
    width = *(int*)&(header[0x12]);
    height = *(int*)&(header[0x16]);
```
Chapter 2

if (imageSize == 0) imageSize = width*height * 3;
if (dataPos == 0) dataPos = 54;

data = new unsigned char[imageSize];
fread(data, 1, imageSize, file);
fclose(file);

glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
return true;
}

Add these lines of code to initGL:

glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

We are not going to dissect this piece of code line by line. In brief, it opens the image file, extracts the first 54 bytes of the file (the bmp header data), and stores the rest of the file as image data. A few OpenGL calls are made to assign this data to an OpenGL texture and that’s it.

You need to have a call that loads the texture in, so add this line of code to _tmain just after the call to initGL:

loadTexture("car.bmp");

Of course, replace car.bmp with the file that you want to load in. Ensure that you have placed the appropriate graphic files in the source code folder.

Texture wrapping
In order to display a texture on the screen, OpenGL maps the texture onto another primitive. This process is known as texture wrapping. As textures are rectangular, it makes sense to map the texture onto a quad.
The following image shows a texture the way that OpenGL sees it: a rectangle with four texture coordinates:

![Texture Coordinates Diagram]

The upper-left is texture coordinate 0, 0. The lower-right is texture coordinate 1, 1. You should be able to identify the texture coordinates of the other corners.

It might make it easier to conceptualize OpenGL numbers if you convert them to percentage, where 0 is zero percent and 1 is 100 percent. For example, you can think of the lower-left corner as being zero percent of the width of the texture and one-hundred percent of the height of the texture.

In order to render a texture, we overlay it (or wrap it) onto a quad. So, let's say we have the following quad defined:

![Quad Diagram]
We could map the texture coordinates to the quad coordinates:

<table>
<thead>
<tr>
<th>Texture Coordinate</th>
<th>Maps to</th>
<th>Quad Coordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 0</td>
<td></td>
<td>0, 0</td>
</tr>
<tr>
<td>1, 0</td>
<td></td>
<td>1, 0</td>
</tr>
<tr>
<td>1, 0</td>
<td></td>
<td>1, 0</td>
</tr>
<tr>
<td>0, 1</td>
<td></td>
<td>0, 1</td>
</tr>
</tbody>
</table>

The following figure shows this graphically:

In its simplest form, texture wrapping is the process of mapping the corners of a texture to the corners of a quad.

You will see texture wrapping also referred to as uv wrapping. I always tried to figure out what uv meant! Here’s the real story: x and y were already used to refer to the quad coordinates, and we had to have something else to call the texture coordinates, so some bright person said, "Let’s use u and v!"
Creating a textured quad

Now, we will write the code to render a textured quad. Add the following function to the code:

```c
void drawTexture()
{
    glBindTexture(GL_TEXTURE_2D, texture);
    glBegin(GL_QUADS);
    glTexCoord2d(0.0, 0.0); glVertex2d(0.0, 0.0);
    glTexCoord2d(1.0, 0.0); glVertex2d(0.5, 0.0);
    glTexCoord2d(1.0, 1.0); glVertex2d(0.5, 0.5);
    glTexCoord2d(0.0, 1.0); glVertex2d(0.0, 0.5);
    glEnd();
}
```

Here is what this code does:

- `glBindTexture(GL_TEXTURE_2D, texture)`: Even if we have thousands of textures in a game, OpenGL can only work with one texture at a time. The call to `glBindTexture` tells OpenGL which texture we are working with right now. Each time a texture is created, OpenGL assigns a number to that texture, called the texture handle. When we loaded our bitmap, we used the `glGenTextures(1, &texture)` command, which instructed OpenGL to generate one texture and save the handle into the variable called `texture`. We then pass this value into the `glBindTexture` function, along with a flag that tells OpenGL that we are working with a 2D texture.

- `glTexCoord2d(0.0, 0.0); glVertex2d(0.0, 0.0)`: We put these two lines together because they work together. You should recognize the call to `glVertex2d`. This function tells OpenGL how to wrap the texture onto the quad (you should also recognize that we are drawing a quad because we set that up in the previous line of code).

- Each call to `glTexCoord2d` defines a texture coordinate. The very next line of code maps the texture coordinate to a quad coordinate. The order is essential: first define a texture coordinate, then define the corresponding quad coordinate.

By the way, don't forget to replace the middle line of code in update with the following line of code:

```c
drawTexture();
```
Now, run the program!

Putting the pieces together
The following image is a composite that illustrates most of the concepts we have covered so far. See if you can you identify the following:

- The transparent areas
- The triangles
- The vertices
- The pivot point
- The texture
- The quad
Summary
This chapter has covered the core concepts that are required to display images on your screen. We started by discussing the OpenGL coordinate system for a 2D game. The coordinate system allows you to place objects on the screen. This was followed by a discussion about the camera, OpenGL’s way of viewing objects that appear on your screen.

Next, you learned how triangles and quads are used to create simple graphics, and how textures can be applied to these primitives to render 2D images to the screen.

You could finally see an image on your screen that has been rendered by OpenGL. As they say, a picture is worth a thousand lines of code!

In the next chapter, you will learn how to turn your still photography into moving pictures through the wonder of animation!
Where to buy this book

You can buy OpenGL Game Development By Example 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.