1
Building a Game Application

In this chapter we will set up our Visual Studio project and build a basic OpenGL application from scratch. We will be using this application throughout the book by extending its capabilities and introducing more features in the later chapters.

We are not going to build anything as complex as the latest multimillion dollar budget First-person shooter or Real-time strategy games in a scant 100 pages, but we are going to learn as much as we can about using OpenGL graphics and Bullet physics by writing small 3D demos. These demos will teach you the foundations necessary to build customized physics and graphical effects in other game projects. Sounds fun? Then let's get started!

Application components

In order to create the simple 3D game demo applications of this book, we will need the following four essential components:

- Application layer
- Physics
- Graphics
- Input handling

The reason for the application layer should be pretty obvious; it provides a starting point to work with, even if it's just a blank window. Meanwhile, we need the remaining components to provide two important elements of any game: visuals and interactivity. If you can't see anything, and you can't interact with it, it would be quite a stretch to claim that what you have is a game!

These are the essential building blocks or components of most games and game engines, and it's important to note that each of them is independent of the rest. When we write code to implement or change the visualization of our objects, we don't want to have to worry about changing anything in the physics system at the same time. This decoupling makes it easy to make these components as simple or complex as we desire.

Of course, a modern game or game engine will have many more components than this, such as networking, animation, resource management, and even audio; but these won't be necessary for the applications in this book since we are focussed on learning about physics and graphics with two specific libraries: Bullet and OpenGL respectively. However, the beauty of component-based design is that there's nothing that stops us from grabbing an audio library such as `FMOD` and giving the demos some much needed sound effects and background music, thus bringing them one step closer to being real games.

Bullet is a physics engine and it is important to realize that Bullet is only a physics simulation solution. It does not provide a means for visualizing its objects and it never promises to. The authors of the library assume that we will provide an independent means of rendering, so that they can focus on making the library as feature-rich in physics as possible. Therefore, in order to visualize Bullet's objects, we will be using OpenGL. But, OpenGL itself is a very low-level library that is as close to the graphics-card hardware as you can get. This makes it very unwieldy, complicated, and frustrating to work with, unless you really want to get into the nuts and bolts of 3D graphics.

To spare us from such hair-pulling frustration, we will be using **FreeGLUT**. This is a library which encapsulates and simplifies OpenGL instructions (such libraries are often called **wrappers**) and, as a bonus, takes care of application bootup, control, and input handling as well. So, with just Bullet and FreeGLUT, we have everything that we need to begin building our first game application.

Exploring the Bullet and FreeGLUT projects

Packaged versions of the Bullet and FreeGLUT projects can be found with this book's source code, which can be downloaded from the PACKT website at: `http://www.packtpub.com/learning-game-physics-with-bullet-physics-and-opengl/book`

Note that this book uses Bullet Version 2.81. As of the time of writing, Bullet is undergoing an overhaul in Version 3.x to make use of multiprocessor environments and push physics processing onto GPUs. Check this github repository for more information:

`http://github.com/erwincoumans/bullet3`

Bullet and FreeGLUT can also be downloaded from their respective project websites:

- `http://bulletphysics.org`
- `http://freeglut.sourceforge.net`

Bullet and FreeGLUT are both open source libraries, licensed under the `zlib` and `X-Consortium/MIT` licenses, respectively. The details can be found at:

`http://opensource.org/licenses/zlib-license.php`

`http://opensource.org/licenses/MIT`

Also, the main website for OpenGL itself is: `http://www.opengl.org`

Exploring Bullet's built-in demo applications

A lot of the designing and coding throughout this book is based upon, and very closely mimics the design of Bullet's own demo applications. This was intentional for good reason; if you can understand everything in this book, you can dig through all of Bullet's demo applications without having to absorb hundreds of lines of code at once. You will also have an understanding of how to use the API from top to bottom.

One significant difference between this book and Bullet's demos is that Bullet uses **GLUT** (**OpenGL Utility Toolkit**) for rendering, while this book uses **FreeGLUT**. This library was chosen partly because FreeGLUT is open source, allowing you to browse through its internals if you wish to, and partly because GLUT has not received an update since 1998 (the main reason why FreeGLUT was built to replace it). But, for our purposes, GLUT and FreeGLUT are essentially identical, even down to the function names they use, so it should be intuitive to compare and find similarities between Bullet's demo applications and the applications we will be building throughout this book.

You can examine the Bullet application demos by opening the following project file in Visual Studio:

```
<Bullet installation folder>\build\vs2010\0BulletSolution.sln
```

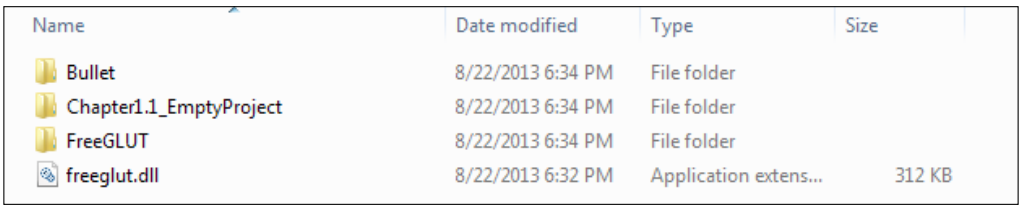
This would be a good time to open this project, compile, and launch some demos. This will help us to get a feel for the kinds of applications we will be building.

To run a different project, right-click on one of the projects, select **Set as StartUp Project**, and hit *F5*.

Starting a new project

Linking the library and header files into a new project can be an exhausting process, but it is essential for building a new standalone project. However, to keep things simple, the `Chapter1.1_EmptyProject` project in the book's source code has all of the headers and library files included with an empty `main()` function ready for future development. If you wish to examine how these projects are pieced together, take the time to explore their project properties in Visual Studio.

Here is a screenshot of the files extracted from the book's source code, and made ready for use:



Name	Date modified	Type	Size
Bullet	8/22/2013 6:34 PM	File folder	
Chapter1.1_EmptyProject	8/22/2013 6:34 PM	File folder	
FreeGLUT	8/22/2013 6:34 PM	File folder	
freeglut.dll	8/22/2013 6:32 PM	Application extens...	312 KB

Note that FreeGLUT also relies on `freeglut.dll` being placed in the project's working folder. Normally this requires the FreeGLUT project to be compiled first, but since it's packaged with the book's source code, this is unnecessary.

Building the application layer

Now we can begin to build an application layer. The purpose of this layer is to separate essential communication with the Windows operating system from our custom application logic. This allows our future demo applications to be more focused, and keep our codebase clean and re-usable.

Continue from here using the `Chapter1.2_TheApplicationLayer` project files.

Configuring FreeGLUT

Handling low-level operating system commands, particularly for a graphical application, can be a tedious and painful task, but the FreeGLUT library was created to help people like us to create OpenGL-based applications and avoid such burdens. The trade-off is that when we launch our application, we effectively hand the majority of control over to the FreeGLUT library.

We can still control our application, but only through a series of callback functions. Each callback has a unique purpose, so that one might be used when its time to render the scene, and another is used when keyboard input is detected. This is a common design for utility toolkits such as FreeGLUT. We will be keeping all of our application layer code within a single class called `BulletOpenGLApplication`.

Downloading the example code

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

Here is a code snippet of the basic class declaration for `BulletOpenGLApplication`:

```
class BulletOpenGLApplication {
public:
  BulletOpenGLApplication();
  ~BulletOpenGLApplication();
  void Initialize();
  virtual void Keyboard(unsigned char key, int x, int y);
  virtual void KeyboardUp(unsigned char key, int x, int y);
  virtual void Special(int key, int x, int y);
```

```
    virtual void SpecialUp(int key, int x, int y);
    virtual void Reshape(int w, int h);
    virtual void Idle();
    virtual void Mouse(int button, int state, int x, int y);
    virtual void PassiveMotion(int x, int y);
    virtual void Motion(int x, int y);
    virtual void Display();
};
```

These essential functions make up the important hooks of our application layer class. The functions have been made virtual to enable us to extend or override them in future projects.

As mentioned previously, FreeGLUT has different functions for different purposes, such as when we press a key, or resize the application window. In order for FreeGLUT to know which function to call at what moment, we make a series of calls that map specific actions to a custom list of callback functions. Since these calls will only accept function pointers that follow specific criteria in return value and input parameters, we are restricted to using the arguments listed in the previous functions.

Meanwhile, by their nature, callback functions must call to a known, constant place in memory; hence a static function fits the bill. But, static functions cannot perform actions on nonstatic or nonlocal objects. So, we either have to turn the functions in `BulletOpenGLApplication` static, which would be incredibly ugly from a programming perspective, or we have to find a way to give it a local reference by passing it as a parameter. However, we just determined that the arguments have already been decided by FreeGLUT and we cannot change them.

The workaround for this is to store our application in a global static pointer during initialization.

```
static BulletOpenGLApplication* g_pApp;
```

With this pointer our callback functions can reach an instance of our application object to work with at any time. Meanwhile an example declaration of one of our callbacks is written as follows:

```
static void KeyboardCallback(unsigned char key, int x, int y);
```

The only purpose of each of these callback functions is to call the equivalent function in our application class through the global static pointer, as follows:

```
static void KeyboardCallback(unsigned char key, int x, int y) {
  g_pApp->Keyboard(key, x, y);
}
```

Next, we need to hook these functions into FreeGLUT. This can be accomplished using the following code:

```
glutKeyboardFunc(KeyboardCallback);
```

The previous command tells FreeGLUT to map our `KeyboardCallback()` function to any *key-down* events. The following section lists FreeGLUT functions which accomplish a similar task for other types of events.

glutKeyboardFunc/glutKeyboardUpFunc

The `glutKeyboardFunc` and `glutKeyboardUpFunc` functions are called when FreeGLUT detects that a keyboard key has been pressed down or up, respectively. These functions only work for keyboard characters that can be represented by a `char` data type (`glutSpecialFunc` and `glutSpecialUpFunc` handle other types).

Some applications and game engines may only call the input function once the key is pressed down, and only sends another signal when the key is released, but nothing in-between. Meanwhile, others may buffer the inputs allowing you to poll it at later times to check the current state of any key or input control, while others may provide a combination of both methods allowing you to choose which method works best for you.

By default, FreeGLUT calls this function repeatedly while a key is held down, but this behavior can be toggled globally with the `glutSetKeyRepeat()` and `glutIgnoreKeyRepeat()` commands.

glutSpecialFunc/glutSpecialUpFunc

The `glutSpecialFunc` and `glutSpecialUpFunc` functions are similar to the previous keyboard commands, but called for special keys such as *Home*, *Insert*, the arrow keys, and so on.

glutMouseFunc

The `glutMouseFunc` function is called when mouse button input is detected. This applies to both button up and button down events, which can be distinguished from the `state` parameter it sends.

glutMotionFunc/glutPassiveMotionFunc

The `glutMotionFunc` and `glutPassiveMotionFunc` functions are called when mouse movement is detected. The `glutMotionFunc()` function is used when any mouse button is currently held down, while the `glutPassiveMotionFunc()` function is used when no mouse buttons are pressed.

glutReshapeFunc

The `glutReshapeFunc` function is called when FreeGLUT detects that the application window has changed its shape. This is necessary for the graphics system (and sometimes game logic) to know the new screen size and it's up to us to make important changes to the scene to handle all possibilities.

glutDisplayFunc

If FreeGLUT determines that the current window needs to be redrawn, the `glutDisplayFunc` function is called. Sometimes Windows detects that an application window is in a damaged state, such as when another window has been partially obscuring it, and this is where this function might be called. We would typically just re-render the scene here.

glutIdleFunc

The `glutIdleFunc` function fills the role of the typical update of game applications. It is called when FreeGLUT is not busy processing its own events, giving us time to perform our own game logic instructions.

More information about these functions can be found in the FreeGLUT documentation at: `http://freeglut.sourceforge.net/docs/api.php`

Initializing FreeGLUT

Finally, we need to configure our application window before FreeGLUT can launch it for us. This is done through the following function calls:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glutInitWindowPosition(0, 0);
glutInitWindowSize(width, height);
glutCreateWindow(title);
glutSetOption (GLUT_ACTION_ON_WINDOW_CLOSE,
  GLUT_ACTION_GLUTMAINLOOP_RETURNS);
```

The following section provides a brief description of each of the previous function calls.

glutInit

The `glutInit` function performs first-step initialization of the FreeGLUT library, passing in the application's parameters. There are several low-level options one can play with here (such as enabling debugging in FreeGLUT itself), but we're not interested in them for our demos. Check the documentation for more information about the available options.

glutInitDisplayMode

The `glutInitDisplayMode` function sets the initial display mode of the window, mostly in terms of what kind of buffers are available. It uses a bitmask to set the values and the call shown previously enables a double-buffered window (`GLUT_DOUBLE`), make these buffers include an alpha channel (`GLUT_RGBA`), and also include a separate depth buffer (`GLUT_DEPTH`). We'll explain these concepts more throughout the book. There are many more options available, so those who are curious can check the online documentation.

Note that RGBA is a short form for the three primary colors; red, green, and blue, and A is short form for alpha, or transparency. This is a common form of describing a single color value in computer graphics.

glutInitWindowPosition/glutInitWindowSize

The `glutInitWindowPosition` and `glutInitWindowSize` functions set the initial position and size of the window in pixels. The position is set relative to the top-left of the main screen.

glutCreateWindow

The `glutCreateWindow` function spawns a top-level window for the Windows OS to manage, and sets the title we want it to display in the title bar.

glutSetOption

The `glutSetOption` function is used to configure a number of options in the window, even the values that we've already edited such as the display mode and the window size. The two options passed in the previous example ensure that when the main window is closed, the main loop will return, exiting our game logic. The main loop itself will be explained in the following section.

Launching FreeGLUT

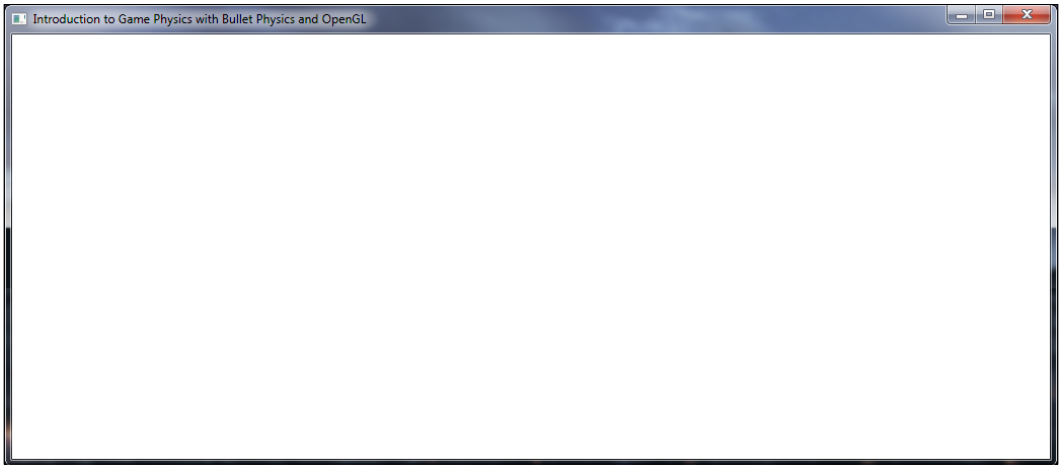
The final and possibly most important function in FreeGLUT is `glutMainloop()`. The moment this function is called, we hand the responsibility of application management over to the FreeGLUT library. From that point forward, we only have control when FreeGLUT calls the callback functions we mapped previously.

In our project code, all of the listed functions are encapsulated with a global function called `glutmain()`, which accepts an instance of our application class as a parameter, stores it in our global pointer, calls its own `Initialize()` function (because even our application class will want to know when the application is powering up), and then calls the `glutMainloop()` function.

And so, finally, we have everything in place to write the all-powerful `main()` function. In this chapter's source code, the `main()` function looks as follows:

```
int main(int argc, char** argv)
{
  BulletOpenGLApplication demo;
  return glutmain(argc, argv, 1024, 768, "Introduction to Game
 Physics with Bullet Physics and OpenGL", &demo);
}
```

Before proceeding, try to compile and run the application from this chapter's source code (*F5* in Visual Studio). A new window should launch with either a plain-white or garbled background (depending on various low-level Windows configuration settings) as shown in the following screenshot. Do not worry if you see a garbled background for now as this will be resolved later.



It is also worth checking that the callback functions are working properly by adding breakpoints to them and verifying that they trigger each frame, and/or when you press a key or click on a mouse button.

Summary

Building a standalone project that hooks into other libraries is the first step towards building an application. We skipped most of this grunt work by using a prebuilt template; but if you're just starting out with the game development, it is important to understand and practice this process for the future, since this will not be the last time you have to tinker with Visual Studio project properties!

The most interesting lesson we learned is how to keep our application layer code in a separate class, and how to get hooks into the FreeGLUT library, thus giving it control over our application.

In the next chapter, we will introduce two of the most important parts of any game: graphics and user input!