2
Rendering and User Input

In this chapter, we will begin our journey into the world of OpenGL by performing some basic initialization steps, and rendering an object onto our window. Then we will learn how to gather user inputs, and how to manipulate the camera using that input in order to view our 3D scene from any angle.

Rendering the scene

In its most basic form, 3D rendering involves four essential tasks:

- Creating a blank canvas on which to draw
- Painting every object in the world onto the canvas, based on the direction it is being viewed from (by the camera)
- Copying the canvas image to the screen
- Clearing the canvas and repeating the process

However, there is much more nuance and complication involved in this process than it might first appear. In this section, we will explore some of the complications of 3D rendering and how they are typically worked around. In-depth explanations of these topics are beyond the scope of this book, and could fill entire chapters by themselves. But, we'll give each of them a cursory examination so that we're not left completely in the dark.

Continue from here using the `Chapter2.1_RenderingTheScene` project files.

Introducing double-buffering

One complete cycle of the previous tasks is often called a single frame, and when the cycle is repeated multiple times per second, this gives us the frame rate, or how many frames per second are being drawn. As long as the cycle is repeated often enough, and there are gradual differences in the position of the camera and the world's objects, then our brain interprets this information as an animated scene — much like an animated cartoon on TV. Other common words to describe these cycles are refresh, iteration, or render-call.

When we perform these tasks, the graphics system spends most of its time handling the second task: drawing the world's objects onto the canvas. When an object is rendered the graphics system picks the corresponding pixel in the canvas and sets the color of the object there.

This canvas is typically referred to as a buffer. Whenever lots of unique values of a common data type are stored together (in this case a unique color value for each pixel), we usually refer to it as a buffer.

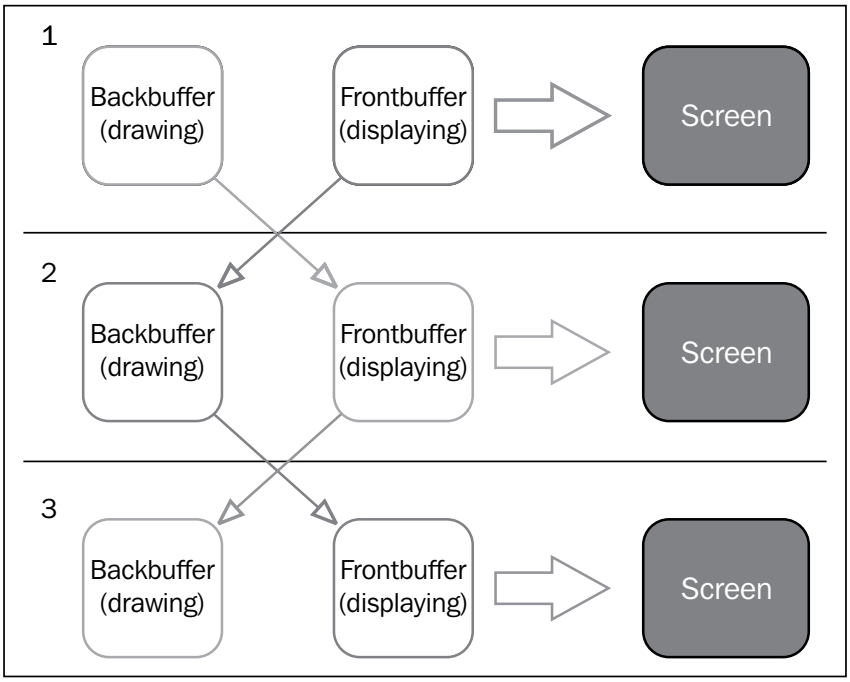
When the display system is ready to draw the next frame, it grabs the current buffer from the video memory (which could be in a dedicated GPU or elsewhere) and copies it for us to be seen on the screen. The buffer is then cleared and the process repeats.

But, what happens if the graphics card has not finished rendering all of the objects before the screen grabs the buffer? If the two processes are not synchronized, it would result in rendering of partial frames which would look very obvious to the human eye and ruin the illusion we're trying to create.

To solve this problem, we will use two buffers instead of one; this is called **double-buffering**. At any moment, one buffer is being displayed (known as the **frontbuffer**) while the other is being drawn to (known as the **backbuffer**). When we've finished drawing onto the backbuffer, we swap the buffers around so that the second is being displayed, while we draw onto the first. We repeat this process over and over again to ensure that we never draw onto the same buffer that we're displaying. This results in a more consistent scene without the graphical glitches.

Note that we have already enabled this feature back in *Chapter 1*, *Building a Game Application*, when we called the `glutInitDisplayMode()` function.

The following diagram shows the working of double-buffering:

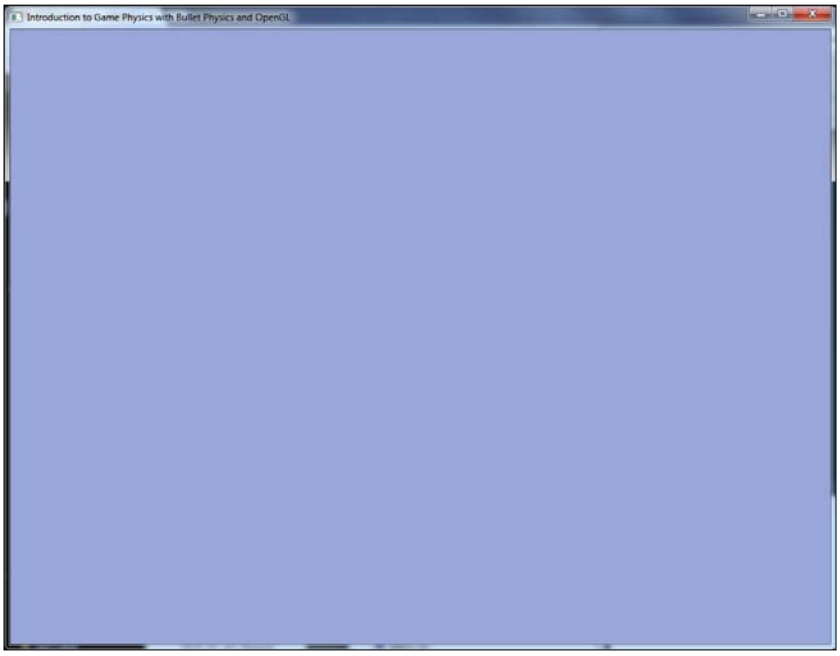


The command to swap these buffers in FreeGLUT is `glutSwapBuffers()`, and the command to clear the backbuffer is `glClear()`. In order to clear the color value of every pixel in the current buffer, we pass the value `GL_COLOR_BUFFER_BIT` into `glClear()`. Both of these functions must be called inside the `Idle()` function, which is automatically and repeatedly called by FreeGLUT whenever it's not busy processing its own events. This is the moment where we should process our own background tasks such as physics processing, object management, responding to game events, and even rendering itself.

In order to set the clearing color, we call the `glClearColor`() function as follows:

```
glClearColor(0.6, 0.65, 0.85, 0);
```

The given values should result in a light-blue color that is 60 percent red, 65 percent green, and 85 percent blue. The fourth parameter is the alpha (transparency) value, and is typically set to `0` in this situation. The following screenshot shows our application window, and now that `glClear`() is being called every iteration:



Understanding the basics of a camera

In order to visualize objects in our scene, a **camera** is required. The mathematics involved in camera control and movement can be quite confusing, so we'll explore it more in-depth towards the end of this chapter. For now, we will simply discuss a stationary camera.

An essential concept in 3D rendering is the transformation matrix, and the most important of which, that are used time and time again, are the **view** and **projection** matrices. The view matrix represents the camera's position/rotation in space, and where it's facing, while the projection matrix represents the camera's aspect ratio and bounds (also known as the camera's **frustum**), and how the scene is stretched/warped to give an appearance of depth (which we call perspective).

One of the most important properties of matrices is being able to combine two matrices together, through a simple matrix multiplication, and resulting in a transformation matrix that represents both. This property massively cuts down the amount of mathematics that needs to be performed every time we render the scene.

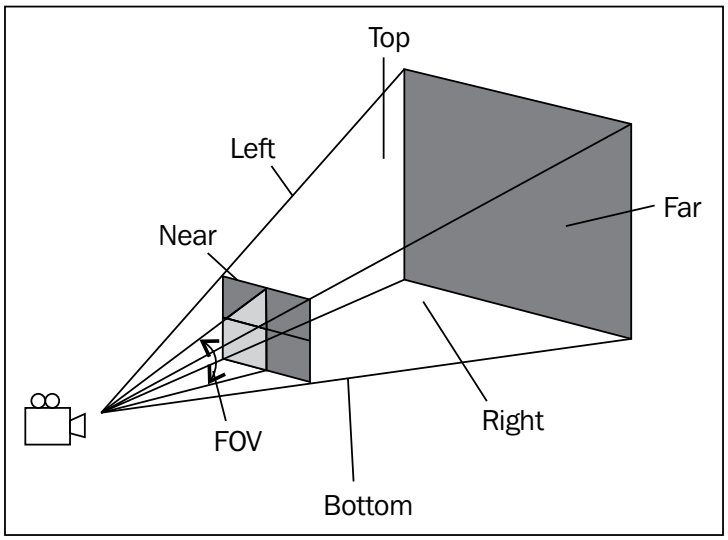
In OpenGL, we must select the matrix we wish to modify with `glMatrixMode()`. From that point onwards, or until `glMatrixMode()` is called again, any matrix-modifying commands will affect the selected matrix. We will be using this command to select the projection (`GL_PROJECTION`) and view (`GL_MODELVIEW`) matrices.

glIdentity

The `glIdentity` function sets the currently selected matrix to the **identity** matrix, which is effectively the matrix equivalent of the number one. The identity matrix is most often used to initialize a matrix to a default value before calling future functions described in the following sections.

glFrustum

The `glFrustum` function multiplies the currently selected matrix by a projection matrix defined by the parameters fed into it. This generates our perspective effect (mentioned previously), and when applied, creates the illusion of depth. It accepts six values describing the left, right, bottom, top, near, and far clipping planes of the camera's frustum: essentially the six sides of a 3D trapezoid (or trapezoidal prism in technical terms). The following diagram is an example of a camera frustum, where **FOV** stands for **field of view**:



gluLookAt

The `gluLookAt` function multiplies the currently selected matrix by a view matrix generated from nine `doubles` (essentially three vectors), which represents the eye position, the point at which the camera is looking at, and a vector that represents which direction is up. The up vector is used to assist in defining the camera's rotation. To use common angular rotation vernacular, if we only define a position and target, that gives us the pitch and yaw we need, but there's still the question of the roll, so we use the up vector to help us calculate it.

glViewport

Finally, `glViewport`() is used to describe the current `Viewport`, or where we should draw the current camera's view of the scene in the application window. Typically, this would stretch to the bounds of the window from `0, 0` to `w, h` (where `w` and `h` are the screen width/height respectively), but this can be used to define whatever viewport is required.

The `glViewport`() function should be called each time when FreeGLUT calls the `Reshape()` function, which is called every time when the window size changes, passing us the new height and width. It's also called once when the application is first launched.

In order to maintain data for our camera, we will keep the following member variables in our application class so that we can refer to them as needed:

```
btVector3 m_cameraPosition;
btVector3 m_cameraTarget;
btVector3 m_upVector;
float m_nearPlane;
float m_farPlane;
```

Throughout this book we will be using Bullet's built-in vector, quaternion, and matrix classes for our 3D mathematics to spare us from having to build our own from scratch.

Meanwhile, the code to update our camera is called within the `Idle()` function. The comments in the chapter's source code will explain the details of this function. If any of the commands in the `UpdateCamera()` function don't make much sense, then go back to the start of this section and refamiliarize yourself with the purpose of the various `gl-` commands, when the FreeGLUT callback functions are triggered, and how they are used.

Basic rendering and lighting

We will now construct a simple object from the primitive shapes (triangles), and explore how OpenGL's built-in lighting system can help us to visualize our object in three dimensions.

Continue from here using the `Chapter2.2_BasicRenderingAndLighting` project files.

Creating a simple box

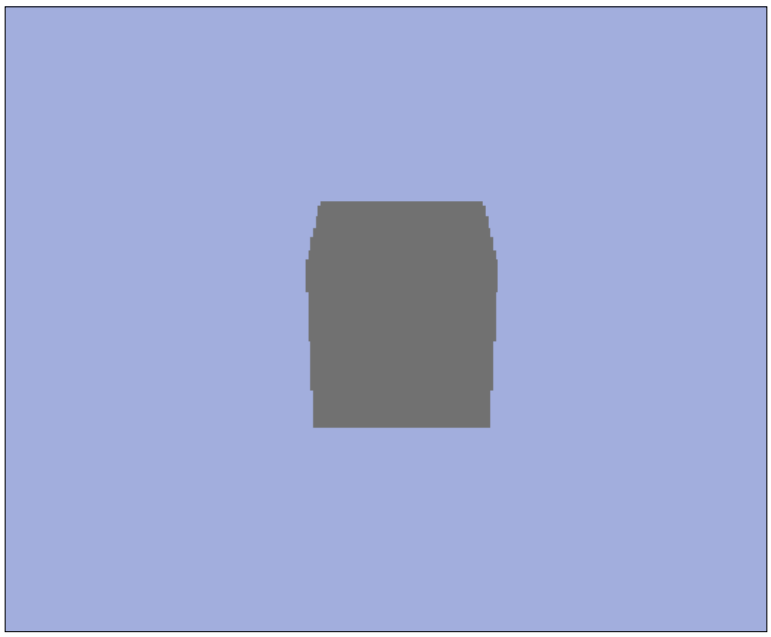
The `glBegin()` and `glEnd()` functions are the two important OpenGL commands that work together to define the starting and ending points (known as **delimiters**) for the construction of a primitive shape. The `glBegin()` function requires a single argument that specifies the type of primitive to render. This determines whether the vertices we input represent points, lines, triangles, quads, or whatever the renderer supports. We'll be using `GL_TRIANGLES` for our box, each of which requires three unique vertices in space in order to render.

There are a variety of commands that can be called between `glBegin()` and `glEnd()` to build the primitive, but the two commands that we will be using are `glVertex3f()`, which defines the position of a vertex in space, and `glColor3f()` which sets the color of subsequent vertices using the same RGB system that we saw in the previous chapter (note that it does not have an alpha value).

The actual task of rendering the box happens in the `DrawBox()` function of the chapter's source code. The most important part is as follows:

```
static int indices[36] = {
  0,1,2, 3,2,1, 4,0,6, 6,0,2, 5,1,4, 4,1,0, 7,3,1, 7,1,5, 5,4,7,
  7,4,6, 7,2,3, 7,6,2};
glBegin (GL_TRIANGLES);
for (int i = 0; i < 36; i += 3) {
  const btVector3 &vert1 = vertices[indices[i]];
  const btVector3 &vert2 = vertices[indices[i+1]];
  const btVector3 &vert3 = vertices[indices[i+2]];
  glVertex3f (vert1.x(), vert1.y(), vert1.z());
  glVertex3f (vert2.x(), vert2.y(), vert2.z());
  glVertex3f (vert3.x(), vert3.y(), vert3.z());
}
glEnd();
```

`DrawBox()` creates a closed box object based on the size of the dimensions we wish to build it from. The input parameter is `btVector3`, providing the three dimensions of the box. `DrawBox()` then uses the concept of indices to iterate through the number of vertices we want, without having to repeat the data. We could create the box from 36 different points, but really there are only eight unique points on a box. Indexes work by labelling each of these eight points with a unique number (index) from `0` to `7`, and use those to define the triangles, instead. Here is a screenshot of our box with no lighting applied:



Let there be light!

At this stage, we can see our box, but all of its faces have exactly the same coloring, which makes it a little difficult to determine the exact shape as it moves around in space. OpenGL has some basic built-in lighting functionality, which we will make use of.

Normals

Normals represent the direction pointing away from a given surface or point. They are used in a variety of useful techniques (and not just lighting!), and the most basic of which is simple diffuse lighting or lighting an object based on the angle between the light source and the surface. Our lighting system will use each point's normal to decide in which direction the incoming light should reflect away from that surface helping it calculate the overall color of the polygon.

Setting the normal for a given vertex can be accomplished by calling the `glNormal3f()` function. This function sets the normal for the subsequent vertices, which could be more than one in the case where they all share the same normal value until `glNormal3f()` is called again. For the record, `glColor3f()` functions in the same way. The renderer assumes that you're using the same color and normal for each new vertex until you specify otherwise.

The normal can be calculated fairly easily by performing a cross-product on the three vertices that make up the triangle. If we remember our 3D mathematics, this gives us a vector perpendicular to the vectors of all three vertices. The cross product is noncommutative, so the output vector could either point inwards or outwards from the surface, depending on what order we performed the cross product, but fixing it is simply a matter of multiplying it by `-1`.

Creating ambient, diffuse, and specular lighting

There are three basic lighting effects that were some of the earliest lighting effects produced in 3D graphics and are still used to today to simulate basic and cheap lighting effects in a 3D environment.

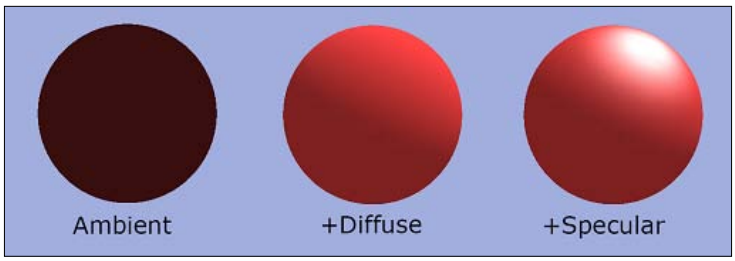
Ambient lighting is used to simulate the base background lighting of a scene. It is essentially the minimum color value of every pixel in the scene in the absence of any other light sources; so if we had an ambient lighting value of (0.3,0.3,0.3), and there were no other light sources present, everything we render would be colored dark grey. Computationally, this effect is cheap.

Diffuse lighting, as mentioned earlier, depends on the direction of the light and simulates the effect of light radiating from a source and rebounding off the surfaces. The shallower the angle between the direction of the light and the surface, the weaker the effect that the light will have on that surface. This effect requires additional mathematics compared to ambient lighting (essentially one dot-product per vertex per light) to determine the output.

Finally, **specular** lighting represents the shininess of an object by highlighting certain areas with a brighter color depending on the angle of the camera with the light source. Because the camera also gets involved, the effect itself changes as the camera moves, and requires a greater amount of mathematics to produce.

However, despite of the difference in mathematical requirements, these three effects are almost trivialized by modern GPUs, and there are far more advanced and realistic visual effects such as global illumination, refraction, depth of field, HDR lighting, and so on, making these simple lighting effects a drop in the ocean by comparison.

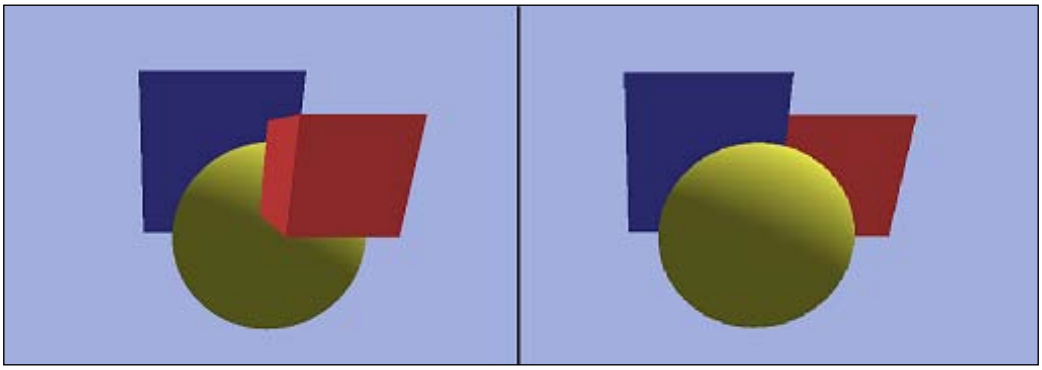
The following diagram shows the same object rendered with ambient, ambient plus diffuse, and ambient plus diffuse plus specular lighting, respectively.



Understanding depth testing

Depth testing is an important part of graphical rendering that specifies how objects should be redrawn over others. To draw a painting in the real world, we must layer our objects on top of others in the correct order, as we start with the sky, then we add a hill on top of the sky, and then add a tree on top of the hill. But, if we draw our tree first, then overdraw with the hill, and then overdraw again with the sky, we would be left with just the sky on our canvas, and an incorrect representation of what we wanted.

The following diagram shows three objects rendered with and without depth testing enabled, respectively. The order of rendering is the small box, the large box, and then the sphere. The small box is closer to the camera, but without depth testing, the small box will be overdrawn by the remaining two. When depth testing is enabled, the renderer understands not to overdraw an object that is closer to the camera.



We need to store this depth information each time we render a new object so we know the depth of the object currently drawn there; but we can't use the original backbuffer to store this information since, there's just not enough information stored in a single RGBA value to do so. So, to keep a track of this information, we add another buffer called the **depth buffer**. Each time we attempt to render a pixel, we check the depth value of the pixel from the depth buffer (also known as the **z-buffer**, because it keeps track of the z-value of each pixel away from the camera). If the pixel is closer, then we render the object's color pixel to the backbuffer, and write the new z-value into the depth buffer. The next time we try to render at that pixel location, we will have the updated value to compare with.

Earlier in this chapter, we mentioned how we can set multiple flags in `glClear()`, to clear certain buffers. The `GL_DEPTH_BUFFER_BIT` flag is used to clear the depth buffer each render call.

Let's go over some of the important OpenGL functions used for a basic lighting and depth testing system. In each case, there are more options available in the OpenGL documentation, which can be examined at your leisure.

glLightfv

The `glLightfv()` function is used to specify the properties of a given light. The first parameter is used to select which light to edit, the second is used to determine which property to edit, and the third is used to specify the new value. The first two parameters must be an enumerator (or enum) corresponding to a specific value. For instance, the options for the first parameter can be `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2`, and so on. Meanwhile, the second parameter could be `GL_AMBIENT`, `GL_DIFFUSE`, or `GL_SPECULAR` to define which lighting property of the given light to modify, or even `GL_POSITION` to define its position. As an example, the following call sets the ambient lighting value of the first (zeroth) light to the value of `ambient`, where `ambient` is `btVector3` representing the ambient color we want:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
```

glEnable

The `glEnable()` function is a very generic function used to enable certain features in the OpenGL library. Every feature we enable typically consumes more processing power, so OpenGL gives us the freedom to enable only what we need. We will use this function to enable lighting in general (`GL_LIGHTING`), create a single light (`GL_LIGHT0`), enable the coloring of our primitives (`GL_COLOR_MATERIAL`), and enable depth testing (`GL_DEPTH_TEST`).

glMaterialfv/glMateriali

The `glMaterialfv` and `glMateriali` functions specify material parameters for the lighting system. Specifically, we will be using them to define the strength of our specular lighting effect. The first parameter for both functions can either be `GL_FRONT`, `GL_BACK`, or both combined to define if the setting should affect front or back faces (faces pointing towards or away from the camera — which is determined by using the normal).

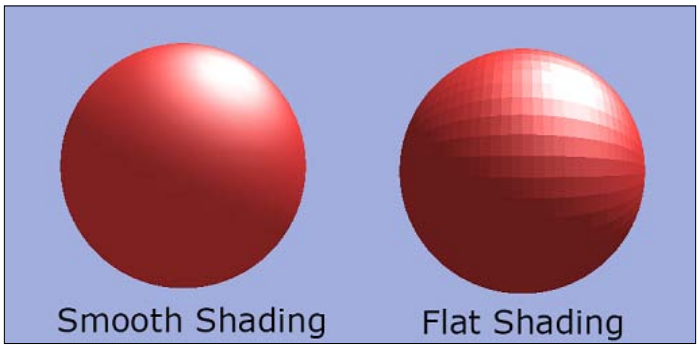
We will be using them as follows:

```
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
glMateriali(GL_FRONT, GL_SHININESS, 15);
```

The first call sets the color of the specular effect through `GL_SPECULAR` (we re-use the specular variable for convenience since it already defines a white color). The second sets the shininess (`GL_SHININESS`) to a value of `15`. Larger values produce weaker shine effects, and vice versa.

glShadeModel

The `glShadeModel()` function sets the current style of lighting to either `GL_FLAT` or `GL_SMOOTH`. This is most noticeable on objects such as spheres. Flat shading is less computationally expensive, but provides a less believable lighting model. Even though the performance hit is barely noticeable on modern GPUs, it can be used for a particular artistic style, as shown in the following screenshot:



This effect is achieved by using the previously mentioned normals. With a flat shading model, the face is colored based on the direction of the light and the normal of the first vertex on the face, and it assumes that the normal value for the rest is the same. Hence, the mathematics used to calculate the diffuse lighting color of each pixel on the face will result in the exact same color. Smooth shading, on the other hand, calculates a separate result for each vertex independently, and then blends the computed color for each pixel between them. It is a very primitive lighting effect, but gives our objects a more believable appearance.

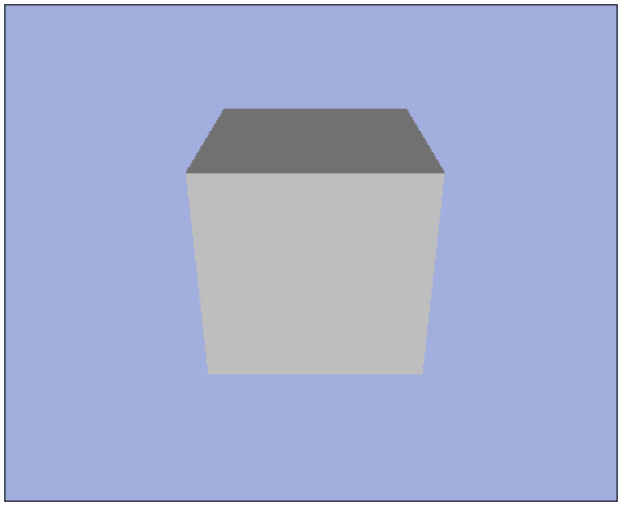
glDepthFunc

The `glDepthFunc` function sets the depth testing function the renderer will use, which could be one of the following options:

- `GL_ALWAYS`
- `GL_NEVER`
- `GL_GREATER`
- `GL_LESS`

These options specify whether the pixel should be overwritten with new information in one of four ways: always, never, if the z-value is greater than the current value (further away), or less than the current value (closer). The most common approach is to choose `GL_LESS`, but never let it to be said that OpenGL didn't give us the power to create whatever weird and wacky graphical effects we want, since choosing the other values will result in some interesting scenes (just use your imagination and the tree/hill/sky example from earlier).

So with this knowledge, inside the `Initialize()` function, we create a light at position (`5,10,1`) with ambient, diffuse, and specular properties, set the shading model to smooth lighting (`GL_SMOOTH`), and set the depth testing function to only overwrite pixels if their depth value is less than the existing value (`GL_LESS`). And voila! We have lighting enabled in our scene. Here is our box with basic lighting applied:



Coloring your box

If you recall, we called `glColor3f()` in our `DrawBox()` function, and set the default color parameter in the function's declaration to an RGB value of (`1,1,1`). This represents the white color: 100 percent red, green, and blue in additive coloring. Since we're not passing a value for this parameter in our `DrawBox()` call, it is still defaulting to white. Therefore, in order to change the color of our box, we simply add a color value to our `DrawBox()` call in the form of a `btVector3`.

```
DrawBox(btVector3(1, 1, 1), btVector3(1.0f, 0.2f, 0.2f));
```

Feel free to tweak the highlighted values until you find a color you prefer.

Note that even though color itself is not a vector, a `btVector3` is a convenient object to use to store three unique floats.

Understanding rendering pipelines

So far (and throughout the remainder of this book), we have performed rendering through a technique called **immediate mode**, where the graphics system essentially forgets the information of each render-call, and must be reminded of the data every time we draw. We have witnessed this in our `DrawBox()` function, where we literally rewrite the information to define and draw our box every single time the function is called, and it is called every single render-call. This is obviously not the most efficient way of doing things!

Immediate mode is opposite to **retained mode** where vertex data is stored (retained) within memory, and recalled by the graphics system when requested. The retained mode method consumes more memory during runtime, but it is much faster and more efficient. However, it can be difficult to understand and use when you're just starting with the 3D graphics.

In addition, there are two types of high-level rendering process called the **fixed-function pipeline** and the **programmable pipeline**. All of the code in this book performs its rendering using the fixed-function pipeline. It is called so because it follows a fixed series of function calls to draw the graphical elements to the screen, such as processing the primitives, performing transforms and lighting, coloring, depth testing, and so on. The important point to make here is that the order of these steps cannot be changed and you have a limited amount of control over each step.

The more modern form of rendering is the programmable pipeline. It is much more fluid, allowing the graphics programmer to have an immense amount of control over the rendering steps through the use of custom scripts called **shaders**. Shaders can vary from very simple scripts that render objects with ambient shading, to complex procedures that render effects such as motion blur, HDR lighting, produce millions of pixel-sized particles, depth of field, and many more. There are even several languages that can be used to write shaders (common ones are Cg, GLSL, and HLSL), and they can be used to produce any graphical effect you want to render (depending on performance limitations, of course). Entire volumes can, and have been written on the nuances of writing shaders with all the various languages, algorithms, and styles of programming they support, making their understanding a highly skilled and highly valued area of 3D graphics programming.

But, don't feel that you've wasted time learning the fixed-function pipeline, because the programmable pipeline can be utterly bewildering if you haven't established a firm grasp of the fundamentals through the fixed-function pipeline first.

User input and camera control

We will now learn how to gather keyboard input from FreeGLUT and use it to interact with our world. This will take the form of rotating our camera around the center of our world (and hence, our box).

Continue from here using the `Chapter2.3_UserInputAndCameraControl` project files.

Implementing camera control

We created a basic camera earlier in this chapter, but it is currently stationary. In order to verify that our world is truly three-dimensional, it would help if we could rotate our camera around a central pivot point. There's a lot of 3D mathematics that goes into controlling camera movement (particularly when it comes to rotation), which is beyond the scope of this book. So, if some of what is covered here seems confusing, know that pretty much any book or blog that covers 3D mathematics and rendering as a whole should cover these topics.

To begin, we will need three new variables for our camera. These will store the current values for the camera's rotation and zoom:

```
float m_cameraDistance; // distance from the camera to its target
float m_cameraPitch; // pitch of the camera
float m_cameraYaw; // yaw of the camera
```

Significant changes are required in our `UpdateCamera()` code to calculate the new camera position based on the previous variables. The additions to this function are liberally commented to explain the process, so we won't consume space explaining them here.

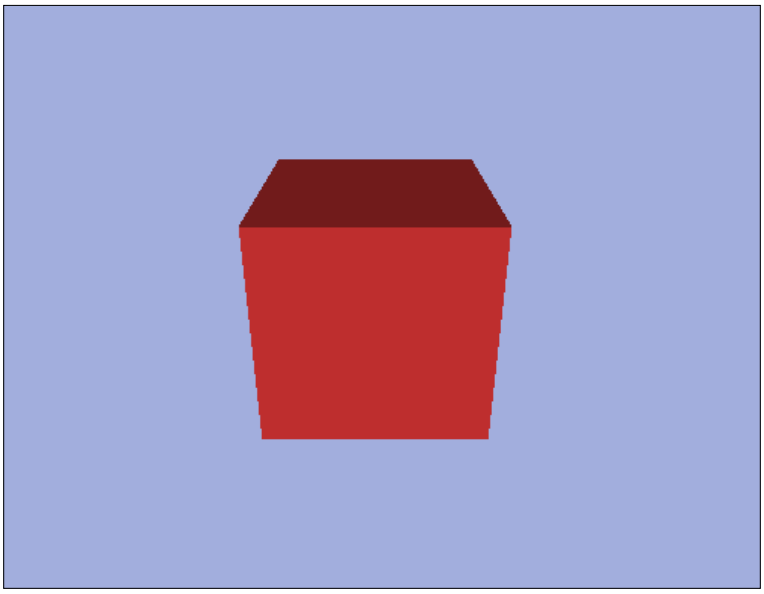
With our camera control code in place, we can manipulate its position by simply modifying the three new variables and calling `UpdateCamera()` to perform the required math for us. However, this isn't useful without some way of gathering input and making the necessary changes to the aforementioned variables.

Gathering user input

FreeGLUT allows us to gather input using the `Keyboard()` and `Special()` callback functions. Remember that `Keyboard()` is called whenever FreeGLUT detects that a generic keyboard key has been pressed, such as letters, numbers, and so on. Meanwhile, the `Special()` function is called for special keys such as the arrow keys, *Home*, *End*, *Page Up*/*Page Down*, and so on.

In this chapter's source code, we've used both of these functions to grab different types of user input, and employed them to modify our camera's pitch, yaw, and zoom distance (recall that the roll is always calculated using an up vector that never changes). Each time the values are modified by a key press, we perform the mathematical calculations necessary to reposition the camera based on this updated information. Note that these functions are repeatedly called while a key is still held down, allowing the camera to continue moving as long as the key is pressed.

Within our application, we can now zoom and rotate the camera with the arrow keys, *Z*, and *X*. Here is our box viewed from a different angle and distance:



Summary

We have delved into some OpenGL code and taken a crash course in 3D rendering, by exploring the basics of double-buffering, and camera initialization through the important view and projection matrices. We've used this knowledge to build a basic scene using primitive shapes, vertices, normals, depth testing, lighting, shading types, and finally, color. As confusing as they might seem, they all contribute to the building blocks from which all modern graphical techniques are born.

Finally, we created a simple and colored box, complete with a moveable camera and basic lighting effects. This will be our essential OpenGL application template going forward, and hopefully we learned a lot about the fundamentals of 3D graphics, OpenGL, and FreeGLUT from building it.

In the next chapter, we will begin to integrate our physics engine by creating and initializing Bullet's core components, and turn our cube into a rigid body which will fall under the effects of gravity!