Direct3D Succinctly

by Chris Rose

Syncfusion | Technology Resource Portal

Direct3D Succinctly

By

Chris Rose

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Jeff Boenig
Copy Editor: Ben Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Chris Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Pomona Majestic Theatre in Queensland.

Chapter 1 Introduction

DirectX is an application programming interface (API) developed by Microsoft to enable programmers to leverage the power of many different types of hardware with a uniform programming interface. It contains components that deal with all aspects of multimedia including graphics, sound, and input. In this book, we will look at techniques for programming three-dimensional (3-D) graphics using DirectX 11 and Visual Studio 2012. The version of Visual Studio used throughout the book is the Windows 8 version of Visual Studio Express 2012.

A background of C++ is assumed, and this book is designed as a follow up to the previous book in the series (*Direct2D Succinctly)*, which mostly looked at two-dimensional (2-D) graphics. We will look at the basics of DirectX and 3-D graphics, communicating with the GPU and loading 3-D model files. We will look at texture mapping, high-level shading language (HLSL), and lighting. We will also look at how to read and respond to user input via a mouse, keyboard, and touchscreen.

We will put it all together, including information on Direct2D from the previous book, and create the beginnings of a simple 3-D game.

Chapter 2 Introduction to 3-D Graphics

Before we dive into DirectX, it is important to look at some of the terms and concepts behind 3-D graphics. In this chapter, we will examine some fundamental concepts of 3-D graphics that are applicable to all graphics APIs.

3-D graphics is an optical illusion, or a collection of techniques for creating optical illusions. Colored pixels are lit up on a 2-D screen in such a way that the image on the screen resembles objects with perspective. Nearer objects overlap and block those farther away, just as they would in the real world.

Coordinate Systems

A coordinate system is a method for describing points in a geometric space. We will be using a standard Cartesian coordinate system for our 3-D graphics. In 2-D graphics, points are specified using two coordinates, one for each of the X and Y dimensions. The X coordinate usually specifies the horizontal location of a point, and the Y coordinate specifies the vertical location of a point. We will see later, when using 2-D textures, that it is also common to use the signifiers U and V to describe 2-D texture coordinates.

In 3-D space, points are specified using three coordinates (X, Y, and Z). Any two coordinates define a plane perpendicular to any other two. The positive and negative directions of each axis, with respect to the monitor, can be arbitrarily chosen by placing a virtual camera in the 3-D scene. For instance, the Y-axis can point upwards, the X-axis can point rightwards, and the Z-axis can point into the screen. If you rotate the camera, the Y-axis can point out of the screen, the X-axis can point downwards, and the Z-axis can point rightwards.

When working with a 3-D Cartesian coordinate system there is a choice to make as to which direction each of the axes point with respect to one another. Any two axes define a 2-D plane. For instance, the X- and Y-axes define a plane, and the Z- and X-axis define another. If you imagine a camera oriented in such a way that the X- and Y-axes define a plane parallel to the monitor, with the Y-axis pointing up and the X-axis pointing to the right, then there is a choice for which direction the Z-axis points. It can point into or out of the screen. A commonly used mnemonic for remembering these two coordinate systems is handedness, or right-handed coordinates and left-handed coordinates. When you hold your hands in the same manner as depicted in *Figure 2.1*, the fingers point in the positive directions of the axis.

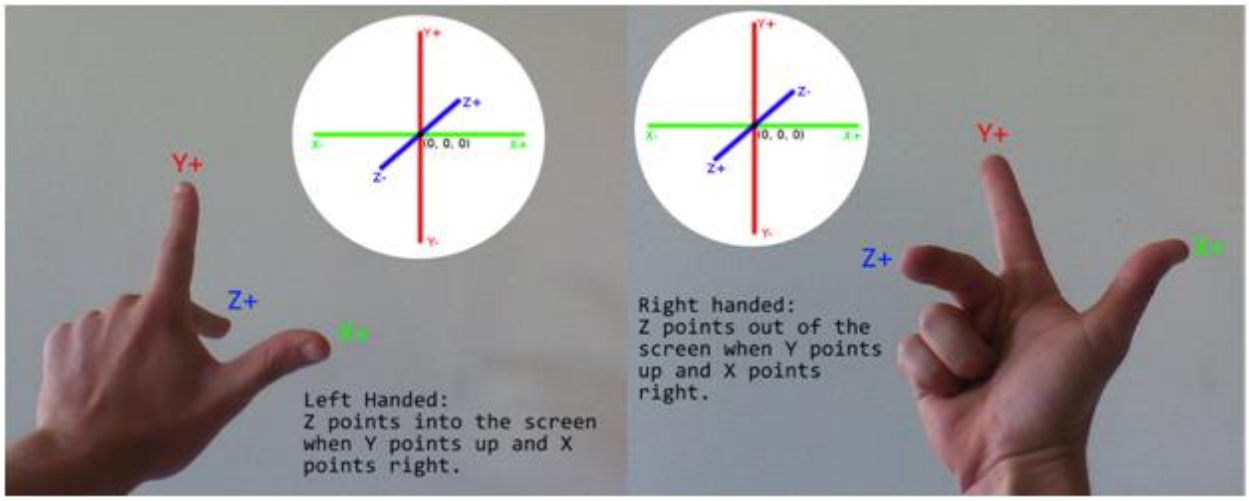

Figure 1: Figure 2.1: Left-handed and Right-handed Coordinates

When using a left-handed coordinate system, the positive Z-axis points into the screen, the Y-axis points up, and the X-axis points to the right. When using a right-handed coordinate system, the positive Z-axis points out of the screen, the Y points up, and the X points to the right. We will be using right-handed coordinates in the code, but DirectX is able to use either.

It is very important to know that the positive directions for the axes are only partially defined by the handedness of the coordinates. The positive directions for the axes can point in any direction with respect to the monitor, because the virtual camera or viewer is able to rotate upside down, backwards, or any direction.

Model, World, and View Space

Models are usually created as separate assets using a 3-D modeling application. I have used Blender for the examples in this book; Blender is available for download from http://www.blender.org/. Models can be exported from the modeling application to files and loaded into our programs. When the models are designed in the 3-D modeler, they are designed with their own local origin. For instance, if you designed a table model, it might look like *Figure 2.2* in the modeling application.

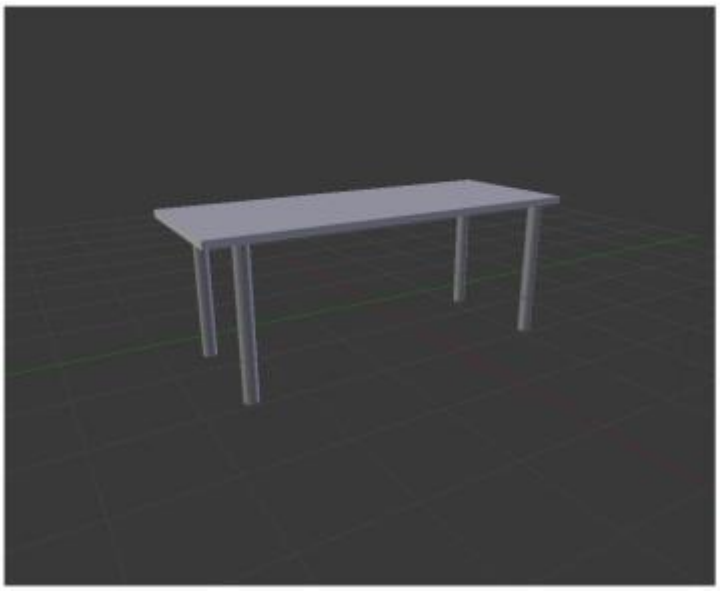

Figure 2: Figure 2.2: Table in the Blender Modeler

Figure 2.2 is a cropped screen shot of the Blender workspace. The red and green lines intersect at the local origin for the object. In Blender, the red line is the X-axis and the green line is the Y-axis. The Z-axis is not pictured, but it would point upwards and intersect the same point that the X and Y intersect. The point where they meet is the location (0, 0, 0) in Blender's coordinates, it is the origin in model coordinates. When we export the object to a file that we can read into our application, the coordinates in the file will be specified with respect to the local origin.

Figure 2.3 shows another screen shot of the same model, but now it has been placed into a room.

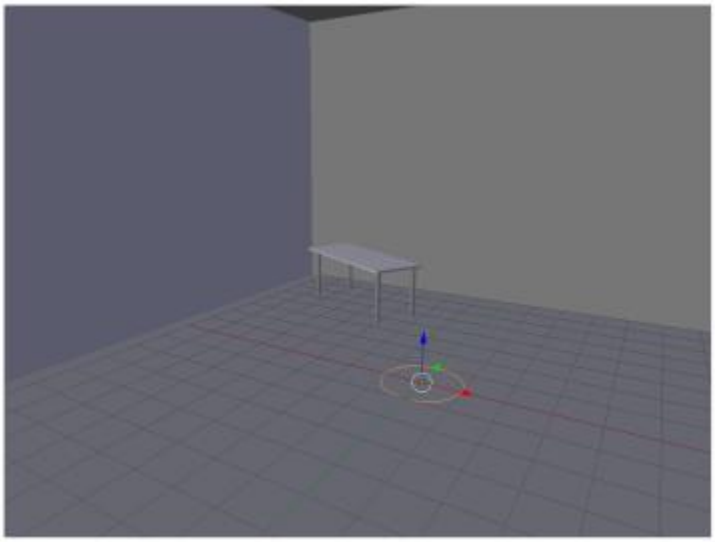

Figure 3: Figure 2.3: Table in World Space

Once we load a model file into our application, we can place the object at any position in our 3-D world. It was modeled using its own local coordinates, but when we place it into the world, we do so by specifying its position relative to the origin of the world coordinates. The origin of the world coordinates can be seen in the image above. This is actually another screen shot from Blender, and usually the axis will not be visible. The table has been placed in a simple room with a floor, ceiling, and a few walls. This translation of the table's coordinates from its local coordinates to the world is achieved in DirectX using a matrix multiplication. We multiply the coordinates of the table by a matrix that positions the table in our 3-D world space. I will refer to this matrix as the model matrix, since it is used to position individual models.

Once the objects are positioned relative to the world origin, the final step in representing the world coordinate space is to place a camera or eye at some point in the virtual world. In 3-D graphics, cameras are positioned and given a direction to face. The camera sees an area in the virtual world that has a very particular shape. The shape is called a frustum. A frustum is the portion of a geometric shape, usually a pyramid or cone that lies between two parallel planes cutting the shape. The frustum in 3-D graphics is a square base pyramid shape with its apex at the camera. The pyramid is cut at the near and far clipping planes (*Figure 2.4*).

Figure 4:

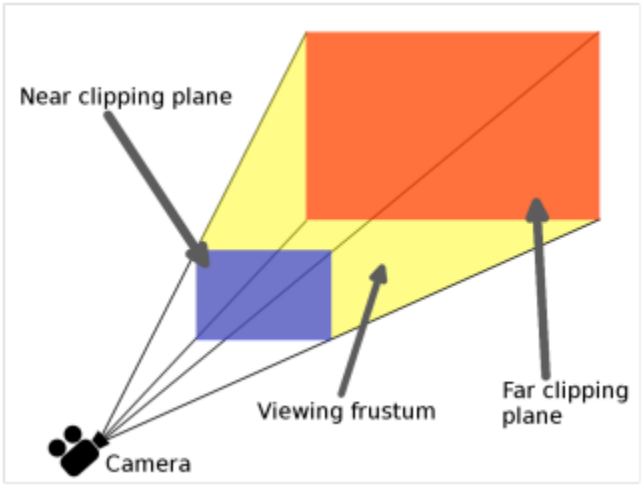

Figure 5: Figure 2.4: Frustum

Figure 2.4 depicts the viewing frustum. The camera is able to view objects within the yellow shaded frustum, but it cannot see objects outside this area. Objects that are closer to the camera than the blue shaded plane (called the near clipping plane) are not rendered, because they are too close to the camera. Likewise, objects that are beyond the orange shaded plane (called the far clipping plane) are also not rendered, because they are too far from the camera. The camera moves around the 3-D world, and any objects that fall in the viewing frustum are rendered. The objects that fall inside the viewing frustum are projected to the 2-D screen by multiplying by another matrix that is commonly called the projection matrix.

Figure 2.5 shows a representation of projecting a 3-D shape onto a 2-D plane. In DirectX, the actual process of projection is nothing more than a handful of matrix multiplications, but the illustration may help to conceptualize the operation.

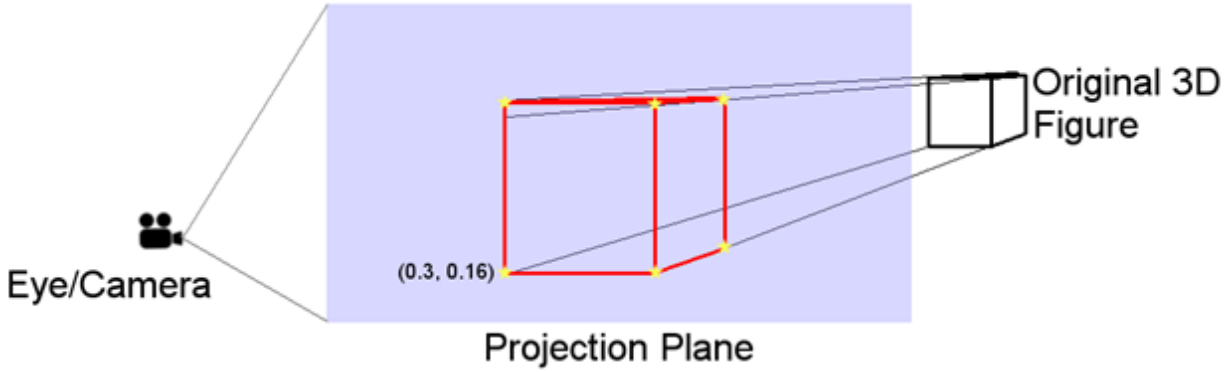

Figure 6: Figure 2.5: 3-D Projection

Figure 2.5 illustrates 3-D projection onto a 2-D plane. The viewer of the scene, depicted as a camera, is on the left side of the image. The middle area, shaded in blue, is the projection plane. It is a plane, which means it is 2-D and flat. It represents the area that the viewer can see. On the far right side, we can see a 3-D cube. This is the object that the camera is looking at. The cube on the right is meant to be a real 3-D object, and the cube projected onto the plane is meant to be 2-D.

Colors

Each pixel on a monitor or screen has three tiny lights very close together. Every pixel has a red, green, and blue light, one beside the other. Each of these three lights can shine at different levels of intensity, and our eyes see a mixture of these three intensities as the pixel's color. Humans see colors as a mixture of three primary colors: red, green, and blue.

Colors are described in Direct3D using normalized RGB or RGBA components. Each pixel has a red, green, and blue variable that specifies the intensity of each of the three primary colors. The components are normalized, so they should range from 0.0f to 1.0f inclusive. 0.0f means 0% of a particular component and 1.0f means 100%.

Colors are specified using three (RGB) or four (RGBA) floating point values with the red first, green second, and blue third. If there is an alpha component, it is last.

To create a red color with 100% red, 13% green and 25% blue, we can use (`1.0f, 0.13f, 0.25f`).

If present, the alpha component is normally used for transparency. In this book, we will not be using the alpha channel, and its value is irrelevant, but I will set it to 100% or 1.0f.

Graphics Pipeline

The graphics pipeline is a set of steps that take some representation of objects, usually a collection of 3-D coordinates, colors, and textures, and transform them into pixels to be displayed on the screen. Every graphics API has its own pipeline. For instance, the OpenGL pipeline is quite different from the DirectX graphics pipeline. The pipelines are always being updated, and new features are added with each new generation of the DirectX API.

In early versions of DirectX, the pipeline was fixed, and it was a predesigned set of stages that the programmers of the API designed. Programmers could select several options that altered the way the GPU rendered the final graphics, but the entire process was largely set in stone. Today's graphics pipeline is extremely flexible and it features many stages that are directly programmable. This means that the pipeline is vastly more complex, but it is also much more flexible. *Figure 2.6* is a general outline of the stages of the current DirectX 11 graphics pipeline.

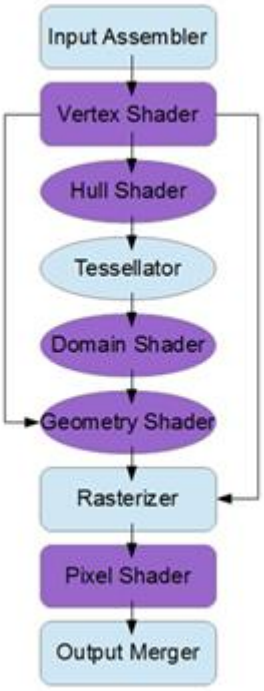

Figure 7: Figure 2.6: DirectX 11 Graphics Pipeline

The rectangular boxes in *Figure 2.6* indicate stages that are necessary, and the ellipses indicate the stages that are optional. Purple stages are programmable using the HLSL language and blue boxes are fixed or nonprogrammable stages. The black arrows indicate the execution flow of the pipeline. For instance, the domain shader leads to the geometry shader, and the vertex shader has three possible subsequent stages. Following the vertex shader can be the hull shader, the geometry shader, or the pixel shader.

Each pipeline stage is designed to allow some specific functionality. In this book, we will concentrate on the two most important stages: the vertex shader stage and the pixel shader stage. The following is a general description of all the stages.

Input Assembler:

This stage of the pipeline reads data from the GPU's buffers and passes it to the vertex shader. It assembles the input for the vertex shader based on descriptions of the data and its layout.

Vertex Shader:

This stage processes vertices. It can lead to the hull, geometry, or pixel shader, depending on what the programmer needs to do. We will examine this stage in detail in later chapters. The vertex shader is a required stage, and it is also completely programmable using the HLSL language in DirectX.

Hull Shader:

This stage and the next two are all used for tessellation, and they are optional. Tessellation can be used to approximate complex shapes from simpler ones. The Hull shader creates geometry patches or control points for the tessellator stage.

Tessellator:

The tessellator takes the geometry patches from the hull shader and divides the primitives into smaller sections.

Domain Shader:

The domain shader takes the output from the tessellator and generates vertices from it.

Geometry Shader:

The geometry shader is a programmable part of the pipeline that works with entire primitives. These could be triangles, points, or lines. The geometry shader stage can follow the vertex shader if you are not using tessellation.

Rasterizer:

The rasterizer takes the output from the previous stages, which consists of vertices, and decides which are visible and which should be passed onto the pixel shaders. Any pixels that are not visible do not need to be processed by the subsequent pixel shader stage. A nonvisible pixel could be outside the screen or located on the back faces of objects that are not facing the camera.

Pixel Shader:

The pixel shader is another programmable part of the pipeline. It is executed once for every visible pixel in a scene. This stage is required, and we will examine pixel shaders in more detail in later chapters.

Output Merger:

This stage takes the output from the other stages and creates the final graphics.

Render Targets, Swap Chain, and the Back Buffer

The GPU writes pixel data to an array in its memory that is sent to the monitor for display. The memory buffer that the GPU writes pixels to is called a render target. There are usually two or more buffers; one is being shown on the screen, while the GPU writes the next frame to another that cannot be seen. The buffer the user can see is called the front buffer. The render target to which the GPU writes is called the back buffer. When the GPU has finished rendering a frame to the back buffer, the buffers swap. The back buffer becomes the front buffer and is displayed on the screen, and the front buffer becomes the back buffer. The GPU renders the next frame to the new back buffer, which was previously the front buffer. This repeated writing of data to the back buffer and swapping of buffers enables smooth graphics. These buffers are all 2-D arrays of RGB pixel data.

The buffers are rendered and swapped many times in sequence by an object called the swap chain. It is called a swap chain because there need not be only two buffers; there could be a chain of many buffers each rendered to and flipped to the screen in sequence.

Depth Buffer

When the GPU renders many objects, it must render those closer to the viewer and not the objects behind or obscured by these closer objects. It may seem that, if there are two objects one in front of the other, the viewer will see the front object and the hidden object does not need to be rendered. In graphics programming, the vertices and pixels are all rendered independently of each other using shaders. The GPU does not know when it is rendering a vertex if this particular vertex is in front of or behind all the other vertices in the scene.

We use a z-buffer to solve this problem. A z-buffer is a 2-D array usually consisting of floating point values. The values indicate the distance to the viewer from each of the pixels currently rasterized in the rasterizer stage of the pipeline. When the GPU renders a pixel from an object at some distance (Z) from the viewer, it first checks that the Z of the current pixel is closer than the Z that it previously rendered. If the pixel has already been rendered and the object was closer last time, the new pixel does not need to be rendered; otherwise the pixel should be updated.

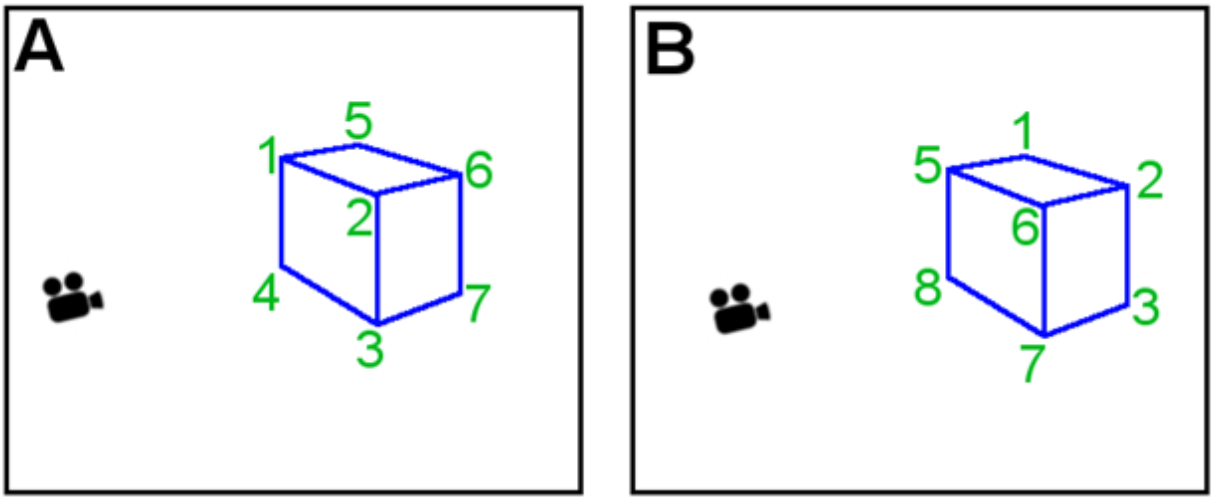

Figure 8: Figure 2.7: Depth Buffer and Faces

Figure 2.7 illustrates two examples of a box being rasterized, or turned into pixels. There is a camera looking at the box on the left. In this example, we will step through rasterizing two faces of the boxes: the one nearest the camera and the one farthest away. In reality, a box has six faces, and this process should be easy to extrapolate to the remaining faces.

Imagine that, in example A, the first face of the box that is rasterized is described by the corners marked 1, 2, 3, and 4. This is the face nearest to the camera. The GPU will rasterize all the points on this face. It will record the distance from each point to the camera in the depth buffer as it writes the rasterized pixels to a pixel buffer.

Eventually, the farthest face from the camera will also be read. This face is described by the corners 5, 6, 7, and 8. Corner 8 is not visible in the diagram. Once again, the GPU will look at the points that comprise the face, and determine how far each is from the camera. It will look to the depth buffer and note that these points have already been rasterized. The distance that it previously recorded in the depth buffer is nearer to the camera than those from the far face. The points from the far face cannot be seen by the camera, because they are blocked by the front face. The pixels written while rasterizing the front face will not be overwritten.

Contrast this with example B on the right-hand side of *Figure 2.7*. Imagine that the face that is rasterized first is the one described by corners 1, 2, 3, and 4. Corner 4 is not depicted in the diagram. This time, it is the far face from the camera that is rasterized first. The GPU will determine the distance from each point on this face to the camera. It will write these distances to the depth buffer while writing the rasterized pixels to a pixel buffer. After a while, it will come to the nearer face, described by corners 5, 6, 7, and 8. The GPU will calculate the distance of each of the points on the face, and it will compare this with the distance it wrote to the depth buffer. It will note that the present points, those comprising the nearer face, are closer to the camera than the ones it rasterized before. It will therefore overwrite the previously rasterized pixels with the new ones and record the nearer depths in the depth buffer.

The above description is a simplified version of the use of depth buffers in the rasterizer stage of the pipeline. As you can imagine, it is easy to rasterize a simple box in this manner, but usually 3-D scenes are composed of thousands or millions of faces, not two as in the previous example. Extensive and ongoing research is constantly finding new ways to improve operations like this, and reduce the number of reads and writes to the depth and pixel buffers. In DirectX, the faces farthest from the camera in the diagrams will actually be ignored by the GPU, simply because they are facing away from the camera. They are back faces and will be culled in the process called back face culling.

Device and Device Context

Device and device context are both software abstractions of the graphics card or Direct3D capable hardware in the machine. They are both classes with many important methods for creating and using resources on the GPU. The device tends to be lower level than the device context. The device creates the context and many other resources. The device context is responsible for rendering the scene, and creating and managing resources that are higher level than the device.

Chapter 3 Setting up the Visual Studio Template

The code in this book is based on the Direct2D App (XAML) template. Most of the functionality of this template should be removed before we begin, and I will spend some time explaining what to remove to get a basic Direct2D/Direct3D framework from this template. The code changes in this chapter are designed to create the starting point for any Direct2D or Direct3D application.

Creating the Project

Open Visual Studio 2012 and create a new **Direct2D App (XAML)** project. I have named my project DXGameProgramming in the screen shot (*Figure 3.1*). Keep in mind that if you use a different name for your project you should rename all the references to the `DXGameProgramming` namespace in your code.

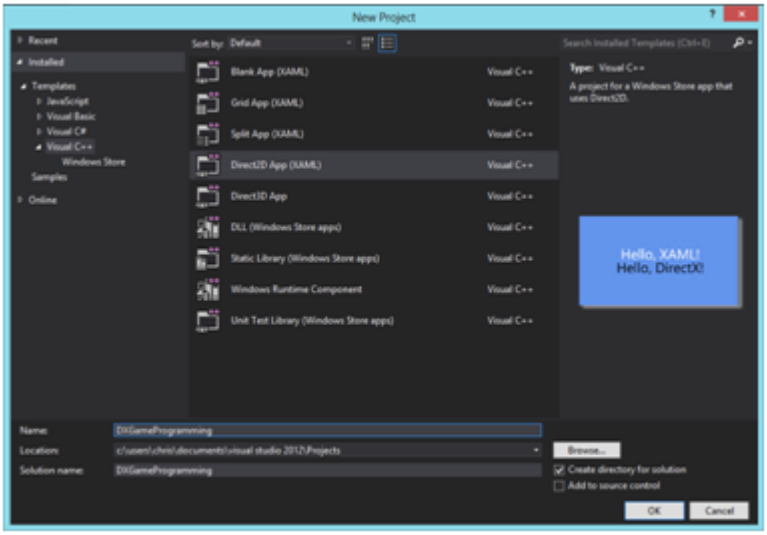

Figure 9: Figure 3.1: Starting a new Direct2D App (XAML) project

Note: I have based all of the code throughout this book on the Direct2D App (XAML) template. This template sets up an application to use both 2-D and 3-D. We will be concentrating mainly on Direct3D, but Direct2D is also very important in creating 3-D applications. Direct2D is used to render things like the heads up display (HUD), player scores, various other sprites, and possibly the backgrounds.

Changes to DirectXPage.xaml

The main XAML page for the application has some controls that we do not need, and these can be removed. Double-click the DirectXPage.XAML file in the solution explorer. This should open the page in Visual Studio's XAML page designer. Delete the text control that says "Hello, XAML" by right-clicking the object and selecting Delete from the context menu (see Figure 3.2).

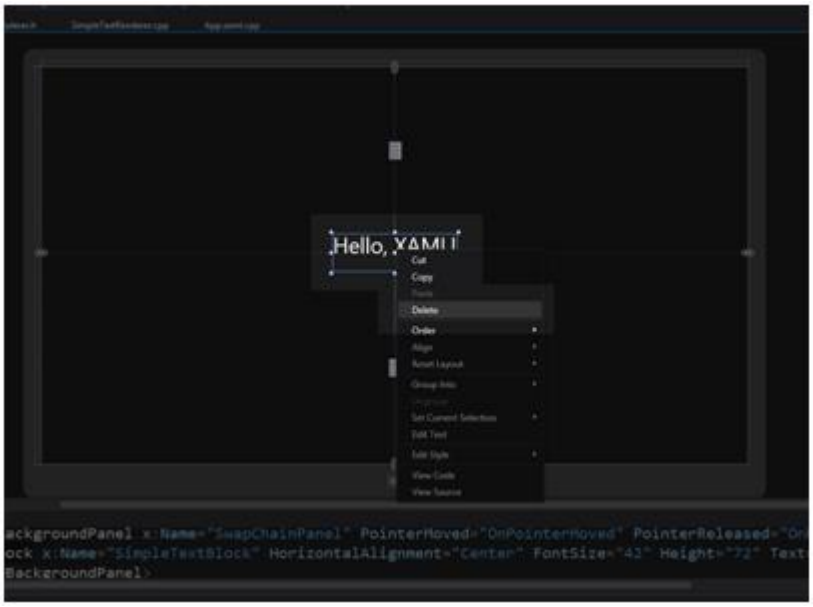

Figure 10: Figure 3.2: Deleting Hello, XAML

Select the XAML code for the `Page.BottomAppBar` and delete it. The code for the DirectXPage.xaml file is presented below. The following code table shows the XAML code after the `Page.BottomAppBar` has been removed.

```
<Page
    x:Class="DXGameProgramming.DirectXPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:DXGameProgramming"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    mc:Ignorable="d">
    <SwapChainBackgroundPanel x:Name="SwapChainPanel"
PointerMoved="OnPointerMoved" PointerReleased="OnPointerReleased"/>
</Page>
```

The DirectXPage.xaml.cpp contains functionality to change the background color and move some text around the screen; this can all be removed. There are several changes to the DirectXPage.xaml.cpp to make. For convenience, the entire modified code is presented in the following code table. In the listing, the four methods `OnPreviousColorPressed`, `OnNextColorPressed`, `SaveInternalState` and `LoadInternalState` have been removed. All of the lines that reference the `m_renderNeeded` variable, the `m_lastPointValid` bool, and the `m_lastPoint` point have also been removed. These variables are used to prevent rendering until the user interacts with the application. This is not useful for a real-time game, since the nonplayer characters and physics of a real-time game continue even when the player does nothing. These changes will make our application update at 60 frames per second, instead of waiting for the user to move the pointer. I have also removed the code in the `OnPointerMoved` event. After making these changes, the project will not compile, since we removed methods that are referenced in other files.

```
//
// DirectXPage.xaml.cpp
// Implementation of the DirectXPage.xaml class.
//

#include "pch.h"
#include "DirectXPage.xaml.h"

using namespace DXGameProgramming;

using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::Graphics::Display;
using namespace Windows::UI::Input;
using namespace Windows::UI::Core;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;

DirectXPage::DirectXPage()
{
	InitializeComponent();

	m_renderer = ref new SimpleTextRenderer();

	m_renderer->Initialize(
```

```
            Window::Current->CoreWindow,
            SwapChainPanel,
            DisplayProperties::LogicalDpi
            );

      Window::Current->CoreWindow->SizeChanged +=
            ref new TypedEventHandler<CoreWindow^,
WindowSizeChangedEventArgs^>(this, &DirectXPage::OnWindowSizeChanged);

      DisplayProperties::LogicalDpiChanged +=
            ref new DisplayPropertiesEventHandler(this,
&DirectXPage::OnLogicalDpiChanged);

      DisplayProperties::OrientationChanged +=
         ref new DisplayPropertiesEventHandler(this,
&DirectXPage::OnOrientationChanged);

      DisplayProperties::DisplayContentsInvalidated +=
            ref new DisplayPropertiesEventHandler(this,
&DirectXPage::OnDisplayContentsInvalidated);

      m_eventToken = CompositionTarget::Rendering::add(ref new
EventHandler<Object^>(this, &DirectXPage::OnRendering));

      m_timer = ref new BasicTimer();
}


void DirectXPage::OnPointerMoved(Object^ sender,
PointerRoutedEventArgs^ args)
{
}

void DirectXPage::OnPointerReleased(Object^ sender,
PointerRoutedEventArgs^ args)
{
}

void DirectXPage::OnWindowSizeChanged(CoreWindow^ sender,
WindowSizeChangedEventArgs^ args)
{
      m_renderer->UpdateForWindowSizeChange();
}
```

```
void DirectXPage::OnLogicalDpiChanged(Object^ sender)
{
	m_renderer->SetDpi(DisplayProperties::LogicalDpi);
}

void DirectXPage::OnOrientationChanged(Object^ sender)
{
	m_renderer->UpdateForWindowSizeChange();
}

void DirectXPage::OnDisplayContentsInvalidated(Object^ sender)
{
	m_renderer->ValidateDevice();
}

void DirectXPage::OnRendering(Object^ sender, Object^ args)
{
	m_timer->Update();
	m_renderer->Update(m_timer->Total, m_timer->Delta);
	m_renderer->Render();
	m_renderer->Present();
}
```

The following code table shows the updated code to the DirectXPage.xaml.h file. The prototypes to the `OnPreviousColorPressed`, `OnNextColorPressed`, `SaveInternalState`, and `LoadInternalState` methods have been removed. The declarations of `m_renderNeeded`, `m_lastPointValid`, and the `m_lastPoint` point have also been removed. The project will still not compile at this point.

```
//
// BlankPage.xaml.h
// Declaration of the BlankPage.xaml class.
//

#pragma once

#include "DirectXPage.g.h"
#include "SimpleTextRenderer.h"
#include "BasicTimer.h"

namespace DXGameProgramming
{
	/// <summary>
	/// A DirectX page that can be used on its own.  Note that it may
not be used within a Frame.
```

```
      /// </summary>
    [Windows::Foundation::Metadata::WebHostHidden]
      public ref class DirectXPage sealed
      {
      public:
            DirectXPage();

      private:
            void OnPointerMoved(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
            void OnPointerReleased(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
            void OnWindowSizeChanged(Windows::UI::Core::CoreWindow^
sender, Windows::UI::Core::WindowSizeChangedEventArgs^ args);
            void OnLogicalDpiChanged(Platform::Object^ sender);
            void OnOrientationChanged(Platform::Object^ sender);
            void OnDisplayContentsInvalidated(Platform::Object^ sender);
            void OnRendering(Object^ sender, Object^ args);

            Windows::Foundation::EventRegistrationToken m_eventToken;

            SimpleTextRenderer^ m_renderer;

            BasicTimer^ m_timer;
      };
}
```

Changes to App.XAML

Open the **App.xaml.cpp** file. Remove the references to the `LoadInternalState` method and the `OnSuspending` event. The code for this file is presented in the following code table.

```
//
// App.xaml.cpp
// Implementation of the App class.
//

#include "pch.h"
#include "DirectXPage.xaml.h"

using namespace DXGameProgramming;
```

```
using namespace Platform;
using namespace Windows::ApplicationModel;
using namespace Windows::ApplicationModel::Activation;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::Storage;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Interop;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;

/// <summary>
/// Initializes the singleton application object.  This is the first
line of authored code
/// executed, and as such is the logical equivalent of main() or
WinMain().
/// </summary>
App::App()
{
     InitializeComponent();
}

/// <summary>
/// Invoked when the application is launched normally by the end user.
Other entry points
/// will be used when the application is launched to open a specific
file, to display
/// search results, and so forth.
/// </summary>
/// <param name="args">Details about the launch request and
process.</param>
void App::OnLaunched(LaunchActivatedEventArgs^ args)
{
     m_directXPage = ref new DirectXPage();

     // Place the page in the current window and ensure that it is
active.
     Window::Current->Content = m_directXPage;
     Window::Current->Activate();
}
```

Open the **App.xaml.h** file and remove the prototype to the `OnSuspending` event. The updated code for this file is presented in the following code table.

```
//
// App.xaml.h
// Declaration of the App class.
//

#pragma once

#include "App.g.h"
#include "DirectXPage.xaml.h"

namespace DXGameProgramming
{
	/// <summary>
	/// Provides application-specific behavior to supplement the
default
	/// Application class.
	/// </summary>
	ref class App sealed
	{
	public:
		App();
		virtual void
OnLaunched(Windows::ApplicationModel::Activation::LaunchActivatedEvent
Args^ args) override;

	private:
		DirectXPage^ m_directXPage;
	};
}
```

At this point, you should be able to compile and run your application. When you run your program, you should see the screen cleared to a light blue color and text saying “Hello, DirectX”. This text is no longer moveable like it was when you first opened the template.

Changes to SimpleTextRenderer

The `SimpleTextRenderer` class will be the main renderer for our application. It will no longer render text, and the name could be changed to something else. I have left it as `SimpleTextRenderer` in the code for simplicity, but usually either the name of this class would be changed or we would write a new class from scratch to do the rendering.

Open the **SimpleTextRenderer.cpp** file. The modified file is presented in the following code table. I have removed all the lines that reference `BackgroundColors`, `m_backgroundColorIndex`, `m_renderNeeded`, `m_textPosition`, `m_textFormat`, `m_blackBrush`, and `m_textLayout`. I have also removed the definitions of the `UpdateTextPosition`, `BackgroundColorNext`, `BackgroundPrevious`, `SaveInteralState`, and `LoadInternalState` methods. In the code below, the screen is still cleared to blue, but it no longer references the BackgroundColors array. Instead, I have used "m_d2dContext->Clear(ColorF(ColorF::CornflowerBlue));".

```
// SimpleTextRenderer.cpp
#include "pch.h"
#include "SimpleTextRenderer.h"

using namespace D2D1;
using namespace DirectX;
using namespace Microsoft::WRL;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::UI::Core;

SimpleTextRenderer::SimpleTextRenderer() { }

void SimpleTextRenderer::CreateDeviceIndependentResources()
{
      DirectXBase::CreateDeviceIndependentResources();
}

void SimpleTextRenderer::CreateDeviceResources()
{
      DirectXBase::CreateDeviceResources();
}

void SimpleTextRenderer::CreateWindowSizeDependentResources()
{
      DirectXBase::CreateWindowSizeDependentResources();
}

void SimpleTextRenderer::Update(float timeTotal, float timeDelta)
```

```
{
	(void) timeTotal; // Unused parameter.
	(void) timeDelta; // Unused parameter.
}

void SimpleTextRenderer::Render()
{
	m_d2dContext->BeginDraw();

	m_d2dContext->Clear(ColorF(ColorF::CornflowerBlue));

	// Ignore D2DERR_RECREATE_TARGET. This error indicates that the
device
	// is lost. It will be handled during the next call to Present.
	HRESULT hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET)
	{
		DX::ThrowIfFailed(hr);
	}
}
```

Open the **SimpleTextRenderer.h** file. The modified code to this file is presented in the following code table. I have removed the declarations for the methods we just deleted (`UpdateTextPosition`, `BackgroundColorNext`, `BackgroundPrevious`, `SaveInteralState`, and `LoadInternalState`). I have also removed the member variables `m_renderNeeded`, `m_textPosition`, `m_textFormat`, `m_blackBrush`, and `m_textLayout`.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"

// This class renders simple text with a colored background.
ref class SimpleTextRenderer sealed : public DirectXBase
{
public:
	SimpleTextRenderer();

	// DirectXBase methods.
	virtual void CreateDeviceIndependentResources() override;
	virtual void CreateDeviceResources() override;
	virtual void CreateWindowSizeDependentResources() override;
	virtual void Render() override;
```

```
        // Method for updating time-dependent objects.
        void Update(float timeTotal, float timeDelta);

};
```

At this point, you should be able to compile and run your application. The application should now clear the screen to `CornFlowerBlue` without printing the text saying “Hello, DirectX”.

This project is now a very basic Direct2D and Direct3D framework with no functionality other than clearing the screen. This is a very good place to begin a project if you are building a graphics engine. We will develop future code samples to add to this project in the following chapters.

Chapter 4 Basic Direct3D

Clearing the Screen using Direct3D

We will begin our exploration of Direct3D by clearing the screen to `CornflowerBlue`. This exact functionality is presently being done by Direct2D in our framework with the call to `m_d2dContext->Clear` in the `SimpleTextRenderer::Render` method. To use Direct3D instead of Direct2D, we can call the `m_d3dContext->ClearRenderTargetView` method. This method takes two parameters; the first parameter is a pointer to an `ID3D11RenderTargetView` and the second parameter is a color specified by normalized RGB floating point values. The altered version of the code to the `Render` method is listed in the following code table.

```
void SimpleTextRenderer::Render()
{
	m_d3dContext->ClearRenderTargetView(m_d3dRenderTargetView.Get(),
		(float*) &XMFLOAT3(0.39f, 0.58f, 0.93f)); // Clear to
cornflower blue

	m_d2dContext->BeginDraw();

	HRESULT hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET)
	{
		DX::ThrowIfFailed(hr);
	}
}
```

You can change the floating point values in the code, in the call to `ClearRenderTargetView`, to any color you like, but it is not recommended that you change it to black (`0.0f, 0.0f, 0.0f`). Always choose something easily recognizable, with bright color, and always clear the screen as the first thing in the `Render` method, whether all pixels are being overwritten or not. The clear to cornflower blue tells a programmer a lot when debugging. For instance, if the screen seems to be flickering random colors instead of showing cornflower blue, it means that it is not presenting the buffers properly; either the buffer being written is not being presented or nothing at all is being written to the buffers, including the clear to cornflower blue. If the program runs and clears to cornflower blue, but does not seem to render any other objects, it may mean that camera is not facing the objects or that the objects are not being rendered to the render target at all. If your objects appear on a background of cornflower blue when you have another background that should be overwriting the cornflower blue, it means that objects are being rendered but the background is not.

Rendering a Triangle

Following on from the previous chapter, we will now render a 3-D triangle. This chapter will introduce the use of buffers. It is extremely important to note the flow of DirectX programming presented in this chapter. Data is often represented in main memory then created on the GPU according to the representation.

Microsoft decided to use data structures instead of long parameter lists in many of the DirectX function calls. This decision makes for lengthy code, but it is not complicated. The same basic steps occur when we create many other resources for the GPU.

Basic Model Class

We will encapsulate our models in a new class called `Model`. Initially, this will be a very basic class. Add two files to your project, **Model.h** and **Model.cpp**. The Model.h code is presented as the following code table, and the code for the Model.cpp file is presented in the second code table.

```
// Model.h
#pragma once

#include "pch.h"

// Constant buffer which will hold the matrices
struct ModelViewProjectionConstantBuffer
{
	DirectX::XMFLOAT4X4 model;
	DirectX::XMFLOAT4X4 view;
	DirectX::XMFLOAT4X4 projection;
};

// Definition of our vertex types
struct Vertex
{
	DirectX::XMFLOAT3 position;
	DirectX::XMFLOAT3 color;
};

class Model
{
	// GPU buffer which will hold the vertices
	Microsoft::WRL::ComPtr<ID3D11Buffer> m_vertexBuffer;

	// Record of the vertex count
```

```
      uint32 m_vertexCount;

public:
      // Constructor creates the vertices for the model
      Model(ID3D11Device* device, Vertex* vertices, int vertexCount);

      // Getters
      ID3D11Buffer** GetAddressOfVertexBuffer() { return
m_vertexBuffer.GetAddressOf(); }
      uint32 GetVertexCount() { return m_vertexCount; }
};
```

In this file you will see two structures defined: the `ModelViewProjectionConstantBuffer` and the `Vertex` structure. The first structure holds matrices to position objects and the camera, as well as projects our 3-D scene onto the 2-D monitor. The second structure describes the types of points we will be rendering our model with. In this chapter, we will use position coordinates and colors to render a triangle so each `vertex` structure consists of a `position` and a `color` element. We will see later that this structure must be described exactly as it appears here for the GPU as well. The version we are describing here is the one stored in main memory.

```
// Model.cpp
#include "pch.h"
#include "Model.h"

Model::Model(ID3D11Device* device, Vertex* vertices, int vertexCount)
{
// Save the vertex count
this->m_vertexCount = vertexCount;

// Create a subresource which points to the data to be copied
D3D11_SUBRESOURCE_DATA vertexBufferData = {0};
vertexBufferData.pSysMem = vertices;
vertexBufferData.SysMemPitch = sizeof(Vertex);
vertexBufferData.SysMemSlicePitch = 0;

// Create a description of the buffer we're making on the GPU
CD3D11_BUFFER_DESC vertexBufferDesc(sizeof(Vertex)*vertexCount,
D3D11_BIND_VERTEX_BUFFER);

// Copy the data from *vertices in system RAM to the GPU RAM:
DX::ThrowIfFailed(device->CreateBuffer(&vertexBufferDesc,
&vertexBufferData, &m_vertexBuffer));
}
```

The body of the constructor in the previous code table illustrates a very common pattern. It describes a data structure and some array of data for the GPU, and it can copy or create the data on the GPU.

In the previous code table, the first thing we need to do is create a `D3D11_SUBRESOURCE_DATA` structure. This is used to point to the data that must be copied to the GPU or to the `vertices` pointer in this instance. Most of the time, the CPU loads data from the disk or creates it, as we are about to do. The CPU uses system RAM, and the GPU does not have access to system RAM, so the data that the CPU loads or creates must be copied to GPU RAM.

The `D3D11_SUBRESOURCE_DATA` structure is required to point to data being copied. It points to the vertices, and the `sysMemPitch` is the size of each element being copied.

The description of the buffer being created must provide the type of buffer being created and the size of the data to copy from the pointer specified in the `D3D11_SUBRESOURCE_DATA` structure.

Creating a Triangle

We will create a model triangle in the `SimpleTextRenderer::CreateDeviceResources` method, since vertex buffers are device dependent resources. Open the **SimpleTextRenderer.h** file and add a reference to include the “Model.h” header at the top. See the following code table with the additional reference highlighted in blue.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "Model.h"
```

Add a new member variable, a pointer, to a model that we will create. I have marked it as private and declared it at the end of the code for the SimpleTextRenderer class in the following code table.

```
      // Method for updating time-dependent objects.
      void Update(float timeTotal, float timeDelta);

private:
      Model *m_model;
};
```

The next step is to define the vertices of the triangle. Open the **SimpleTextRenderer.cpp** file and define a triangle in the `CreateDeviceResources` method. The changes are highlighted in the following code table.

```
void SimpleTextRenderer::CreateDeviceResources()
{
	DirectXBase::CreateDeviceResources();

	// Define the vertices with the CPU in system RAM
	Vertex triangleVertices[] =
		{
		{ XMFLOAT3(-1.0f, 0.0f, 0.0f), XMFLOAT3(1.0f, 0.0f, 0.0f) },
		{ XMFLOAT3(0.0f, 1.0f, 0.0f), XMFLOAT3(0.0f, 1.0f, 0.0f) },
		{ XMFLOAT3(1.0f, 0.0f, 0.0f), XMFLOAT3(0.0f, 0.0f, 1.0f) }
		};

	// Create the model instance from the vertices:
	m_model = new Model(m_d3dDevice.Get(), triangleVertices, 3 );
}
```

In the previous code table, the vertices are created using the CPU in system RAM. Remember that the constructor for the `Model` class will create a copy of this buffer on the GPU. The temporary `triangleVertices` array will fall out of scope at the end of this method, but the vertex buffer will remain intact on the GPU.

You should be able to run your application at this point. It won't look any different but it is creating a rainbow colored triangle on the GPU.

Creating a Constant Buffer

We need to create a constant buffer on the GPU to hold the transformation matrices for the object's position, the camera's position, and the projection matrix. A constant buffer is only constant with respect to the GPU. The CPU is able to change the values by updating the buffers.

The idea behind the constant buffer is that the CPU needs to pass information to the GPU frequently. Instead of passing many individual small variables, variables are collected together into a structure and all passed at once.

Open the **SimpleTextRenderer.h** file and add two new variables: `m_constantBufferCPU` and `m_constantBufferGPU`. These changes are highlighted in the following code table.

```
private:
	Model *m_model;
	Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
	ModelViewProjectionConstantBuffer m_constantBufferCPU;
};
```

Open the **SimpleTextRenderer.cpp** file and create the `m_constantBufferGPU` on the device in the `CreateDeviceResources` method. The code to create this buffer is highlighted in the following code table.

```
	// Create the model instance from the vertices:
	m_model = new Model(m_d3dDevice.Get(), triangleVertices, 3);

	// Create the constant buffer on the device
	CD3D11_BUFFER_DESC
constantBufferDesc(sizeof(ModelViewProjectionConstantBuffer),
D3D11_BIND_CONSTANT_BUFFER);
	DX::ThrowIfFailed(m_d3dDevice->CreateBuffer(&constantBufferDesc,
nullptr, &m_constantBufferGPU));
}
```

The code in the previous table is used to reserve space on the GPU for a buffer exactly the size of the `ModelViewProjectionConstantBuffer` structure. It sets the `m_constantBufferGPU` to point to this space.

Next, we need to set the values for the CPU's version of the constant buffer, the version that is stored in system memory. The projection matrix will not change throughout our application, so we can set the CPU's value for this matrix once. The values for the projection matrix depend on the size and resolution of the screen, so it is best to do this in the `SimpleTextRenderer::CreateWindowSizeDependentResources` method. The code for setting the projection matrix is highlighted in the following code table.

```
void SimpleTextRenderer::CreateWindowSizeDependentResources()
{
	DirectXBase::CreateWindowSizeDependentResources();

	// Store the projection matrix
	float aspectRatio = m_windowBounds.Width / m_windowBounds.Height;
	float fovAngleY = 70.0f * XM_PI / 180.0f;
	XMStoreFloat4x4(&m_constantBufferCPU.projection,
		XMMatrixTranspose(XMMatrixPerspectiveFovRH
		(fovAngleY,aspectRatio,0.01f,500.0f)));
}
```

The aspect ratio of the screen is the width divided by the height. The `fovAngleY` is the angle that will be visible to our camera in the Y-axis. The calculation here means that 70 degrees will be visible; this is 35 degrees left and right of the center of the camera. The `0.01f` parameter sets the near clipping plane to 0.01 units in front of the camera. The parameter passed as `500.0f` sets the far clipping plane to 500 units in front of the camera. This means anything outside of the 70 degree field of view (FOV), closer than 0.01f, or farther than 500.0f from the camera will not be rendered. Note also that we are updating the CPU's version of the constant buffer (`m_constantBufferCPU`), not the GPU's version.

Next, we can position our camera. In many games, the camera is able to move, but our camera will be static. We will position it in the `SimpleTextRenderer::Update` method. The code for positioning the camera is highlighted in the following code table.

```
void SimpleTextRenderer::Update(float timeTotal, float timeDelta)
{
      (void) timeTotal; // Unused parameter.
      (void) timeDelta; // Unused parameter.

// View matrix defines where the camera is and what direction it looks
in
      XMStoreFloat4x4(&m_constantBufferCPU.view, XMMatrixTranspose(
            XMMatrixLookAtRH(
                  XMVectorSet(0.0f, 0.0f, 2.0f, 0.0f),// Position
                  XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f),// Look at
                  XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f)   // Up vector
                  )));
}
```

The above code sets the view matrix for the CPU's constant buffer. We will use an XMMatrixLookAtRH so our coordinate system will be right-handed. The parameters define where the camera is located, to what point is it looking, and the up vector for the camera. Figure 4.1 illustrates the meaning of these three vectors.

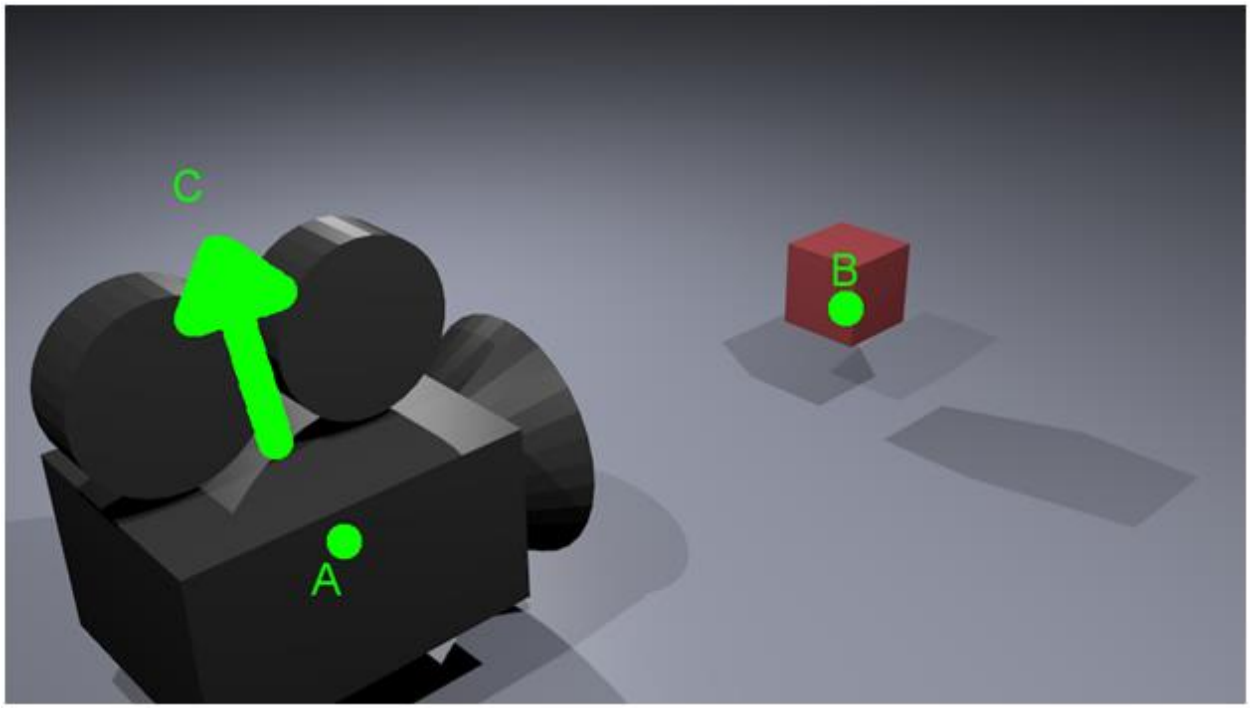

Figure 11: Figure 4.1: LookAtMatrix

Figure 4.1 depicts a camera looking at a cube. Two points are highlighted in bright green, and there is also a prominent green arrow. Point A corresponds to the position of the camera; it defines the location in world coordinates that the camera is positioned. This is the first vector of the three vectors in the call to `XMMatrixLookAtRH` from the previous code table.

Point B defines the point that the camera is looking towards; it determines the direction the camera is facing. This corresponds to the second vector in the call to `XMMatrixLookAtRH` in the previous code table. There happens to be a box in the diagram at this point, but a camera can look toward a point whether there is an object there or not.

Point C corresponds to the up vector; it is the direction that the top of the camera is pointing toward. Notice that without specifying an up vector, the camera is free to roll left or right and still look at the little box from the same position. The up vector is the third and final vector in the call to `XMMatrixLookAtRH` from the previous code table. By specifying all three of these vectors, we have defined exactly where a camera is positioned, what it is looking towards, and its orientation. Notice that the up vector is a direction, not a point, and it is relative to the camera.

Vertex and Pixel Shaders

And now we come to the most powerful and flexible part of the DirectX API: shaders. We have a buffer on the GPU and we wish to tell the GPU to render its contents. We do this by writing small programs. The first is a vertex shader. A vertex shader's code executes once for every vertex in a vertex buffer. A pixel shader's code executes once for every pixel in a scene.

Vertex Shader

Before we dive into vertex shaders, we should take some time to look at what vertex shaders are used for. In the chapter titled *Introduction to 3-D Graphics*, under the section “Model, World, and View Space,” we looked at transforming the coordinates of a model from its original local coordinates to world coordinates. We can place the model anywhere we like in a scene. We can reuse the same model and place multiple copies at different locations. We could also change the size of the objects, turn them upside-down, etc.

The next step, once all of our models are all placed into the world somewhere, is to place a viewer: a camera or eye, something to look at the scene. When we place a viewer into a DirectX scene, what we actually do is rotate and move the whole world such that we can determine the objects that are visible to the camera and render them.

The vertex shader is responsible for the above mentioned transformations. It converts models from local coordinates to view space. After the models are placed and camera is oriented, a vertex shader usually transforms all the points again so that the projection of the scene is correct. It does this using the projection matrix. When the vertex shader is finished placing all the models, the viewer and the projection is applied, and the resulting vertices are sent to the next shader. In our case, the next stage is the pixel shader, although a rasterizer pipeline stage will be called implicitly behind the scenes to turn the vectors from the vertex shader into pixels for the pixel shader.

The best way to see how a vertex shader works is to create one. To add a new vertex shader to your project, right-click on the project’s name in the solution explorer and select **Add new item** from the context menu (see *Figure 4.2*).

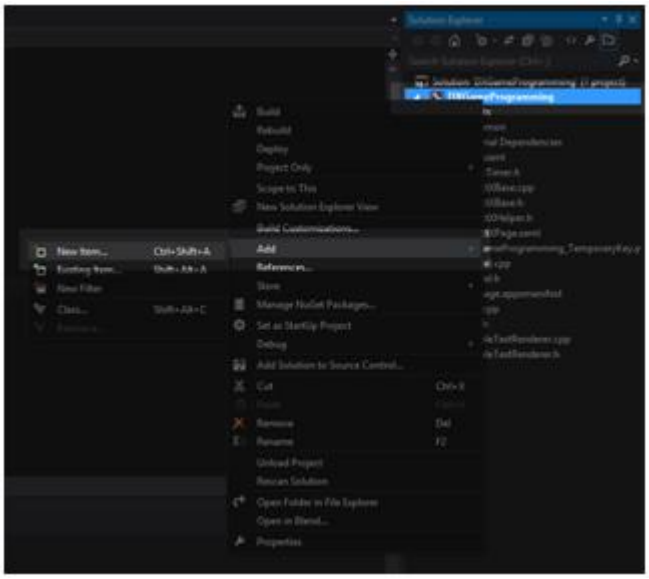

Figure 4.2: Adding a New Item

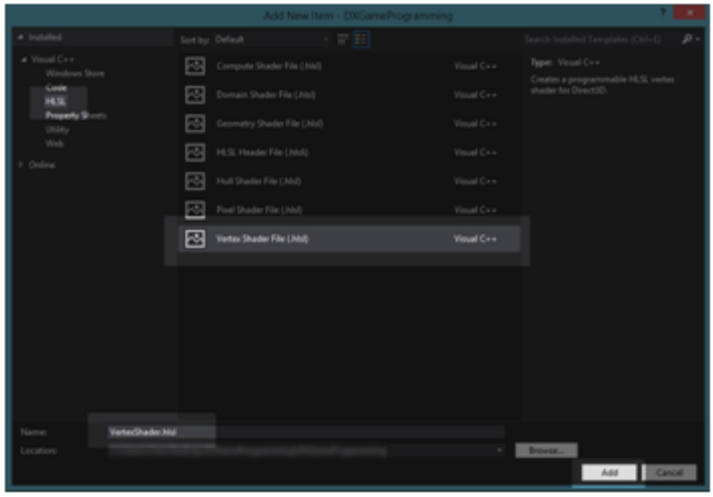

Figure 4.3: Adding a Vertex Shader

Select HLSL from the side panel and **Vertex Shader File (.hlsl)** from the middle panel (see *Figure 4.3*). I have left the filename as **VertexShader.hlsl**. Once you are done, click **Add**. Visual Studio will write a small example vertex shader for us and add it to our project. Replace the code with the code from the following code table.

```
// VertexShader.hlsl

// The GPU version of the constant buffer
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
	matrix model;
	matrix view;
	matrix projection;
};

// The input vertices
struct VertexShaderInput
{
	float3 position : POSITION;
	float3 color : COLOR0;
};

// The output vertices as the pixel shader will get them
struct VertexShaderOutput
{
	float4 position : SV_POSITION;
	float3 color : COLOR0;
};
```

```
// This is the main entry point to the shader:
VertexShaderOutput main(VertexShaderInput input)
{
      VertexShaderOutput output;
      float4 pos = float4(input.position, 1.0f);

      // Use constant buffer matrices to position the vertices:
      pos = mul(pos, model);      // Position the model in the world
      pos = mul(pos, view); // Position the world with respect to a
camera
      pos = mul(pos, projection);// Project the vertices
      output.position = pos;

      // Pass the color of the vertices to the pixel shader
      output.color = input.color;

      return output;
}
```

This file is written in the HLSL language. This language is very similar to C++. We will examine the language in a little more detail later. Briefly, you will see a constant buffer (`cbuffer`) defined at the top of the file. This is the shader's version of the constant buffer we created earlier. The `m_constantBufferGPU` points to this buffer.

There are two structures defined: `VertexShaderInput` and `VertexShaderOutput`. The input to a vertex shader is a vertex buffer. We have already created and copied our triangle to the GPU as a vertex buffer. It is extremely important to note that this structure (`VertexShaderInput`) is exactly the same as the `Vertex` structure we defined in the `Model` class. Coordination of the same data types between the CPU and the GPU can be very confusing. `XMFLOAT3` for the CPU is exactly the same thing as `FLOAT3` for the GPU.

The second structure is the output of the vertex shader; this will pass to the pixel shader that we are about to write. The real magic in this shader lies in the three calls to the `mul` intrinsic. This is the intrinsic for a standard matrix multiplication. This method multiplies the positions of the vertices, first by the `model` matrix, then by the `view` matrix. Finally, they are multiplied by the `projection` matrix, which will result in the final position of the 3-D vertices that are visible in the viewing frustum. After the vertices are positioned, the color of each is copied to the output. Whatever the vertex shader returns will be given to the pixel shader. Note that the pixel shader has a float 4 instead of a float 3 for its position element.

Pixel Shader

A pixel shader executes its code once for every pixel that the rasterizer stage of the pipeline passes to it. These pixels depend completely on the vertex shader, since it is the vertex shader that determines where the models are and the position of the camera. The vertex shader also uses the projection matrix to specify how wide the viewing angle is, how near and how far the camera can see, etc. The visible pixels will be determined by the rasterizer stage and passed to the pixel shader. The rasterizer stage of the pipeline is passed the output from the vertex shader. It determines which pixels are going to be visible on the user's screen, and it calls the pixel shader for every visible pixel. The rasterizer passes the output from the vertex shader to the pixel shaders. The pixel shaders can color pixels with different hues; they can apply textures and lighting effects to the pixels as well.

To add a pixel shader, follow the same steps as *Figure 4.2* and *Figure 4.3*, only select a **Pixel Shader file (.hlsl)** from the window instead of a vertex shader. Once again, I have left the file's name as the default, so mine will be called PixelShader.hlsl. Once again, Visual Studio will create the new shader and write some example HLSL code. Replace the code with the following code table.

```
// PixelShader.hlsl

// Input is exactly the same as
// vertex shader output!
struct PixelShaderInput
{
      float4 position : SV_POSITION;
      float3 color : COLOR0;
};

// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
      // Return the color unchanged
      return float4(input.color,1.0f);
}
```

The pixel shader is very simple; it begins with a structure definition of the type of input. This must match the VertexShaderOutput exactly, because they are one and the same thing. The output from the vertex shader is the input to the pixel shader.

The main body of the pixel shader does almost nothing other than pass on the colors of the pixels. It also adds a fourth component to the RGB, which is the Alpha channel (`1.0f`).

You should be able run your application at this point. It will not look any different, but if you navigate to the output directory, you will see two new files have been compiled: VertexShader.cso and PixelShader.cso (see *Figure 4.4*).

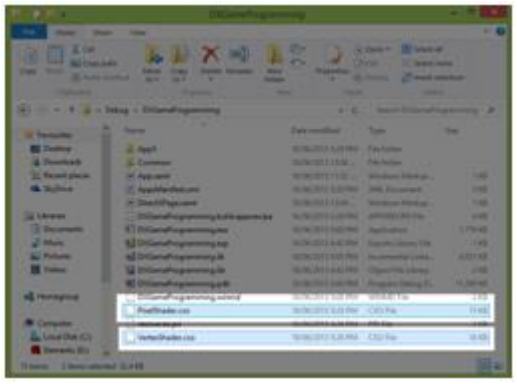

Figure 12: Figure 4.4: CSO Files

These files are an intermediate version of our shaders. Visual Studio has compiled our HLSL source code files to a binary format using the FXC.exe compiler. CSO stands for Compiled Shader Object file. From here we need our graphics card's driver to compile the CSO files to a binary format that the particular hardware inside the computer understands. This is good because all GPUs understand different machine code, but we can create shaders with the HLSL language and the graphics card drivers compile them to specific machine code. In this way, they will run on AMD graphics cards as well as NVidia or integrated graphics cards.

Adding Shader Classes

The next step towards rendering our triangle is compiling the shaders from the CSO files. We need a method that reads the files; this could be placed anywhere, but I have selected to place it in the DirectXHelper.h file. The following code table highlights the changes to the DirectXHelper.h file.

```
// DirectXHelper.h
#pragma once

using namespace Microsoft::WRL;
using namespace Windows::ApplicationModel;
using namespace Windows::Graphics::Display;

namespace DX
{
	inline void ThrowIfFailed(HRESULT hr)
	{
		if (FAILED(hr))
		{
			// Set a breakpoint on this line to catch Win32 API
```

```
errors.
                throw Platform::Exception::CreateException(hr);
            }
        }

// Reads bytes from the specified file in the current folder
inline Platform::Array<byte>^ ReadData(_In_ Platform::String^
filename)
        {
        CREATEFILE2_EXTENDED_PARAMETERS extendedParams = {0};
        extendedParams.dwSize = sizeof(CREATEFILE2_EXTENDED_PARAMETERS);
        extendedParams.dwFileAttributes = FILE_ATTRIBUTE_NORMAL;
        extendedParams.dwFileFlags = FILE_FLAG_SEQUENTIAL_SCAN;
        extendedParams.dwSecurityQosFlags = SECURITY_ANONYMOUS;
        extendedParams.lpSecurityAttributes = nullptr;
        extendedParams.hTemplateFile = nullptr;

        Platform::String ^path =
Platform::String::Concat(Package::Current->InstalledLocation->Path,
"\\");

        Wrappers::FileHandle
file(CreateFile2(Platform::String::Concat(path, filename)->Data(),
            GENERIC_READ, FILE_SHARE_READ, OPEN_EXISTING,
&extendedParams));

        if (file.Get() == INVALID_HANDLE_VALUE)
            throw ref new Platform::FailureException();

        FILE_STANDARD_INFO fileInfo = {0};
        if (!GetFileInformationByHandleEx(file.Get(), FileStandardInfo,
&fileInfo, sizeof(fileInfo)))
            throw ref new Platform::FailureException();

        if (fileInfo.EndOfFile.HighPart != 0)
            throw ref new Platform::OutOfMemoryException();

        Platform::Array<byte>^ fileData = ref new
Platform::Array<byte>(fileInfo.EndOfFile.LowPart);

        if (!ReadFile(file.Get(), fileData->Data, fileData->Length,
nullptr, nullptr))
            throw ref new Platform::FailureException();
```

```
        return fileData;
        }
}
```

The method previously used reads binary data from the file specified as a parameter and returns it. The file must reside in the application's directory, and the method does not accept full file paths. This code will cause a subtle problem. Currently, in the pch.h file we are likely referencing the wrl/client.h header. We need to reference the wrl.h header instead, because this header defines some of the structures used in the `ReadData` method like `file` and the `Wrappers` namespace. Change the header in the pch.h file as per the following code table.

```
// pch.h
#pragma once

// #include <wrl/client.h>
#include <wrl.h>
#include <d3d11_1.h>
#include <d2d1_1.h>
#include <d2d1effects.h>
#include <dwrite_1.h>
#include <wincodec.h>
#include <agile.h>
#include <DirectXMath.h>
#include "App.xaml.h"
```

Next we can add two classes: one to wrap up the vertex shaders and another to wrap up the pixel shaders. First we will add the class called `VertexShader`. Add two new files to your project, **VertexShader.h** and **VertexShader.cpp**. The code for these files is specified in the two following code tables.

```
// VertexShader.h
#pragma once

#include "DirectXHelper.h"

class VertexShader {
private:
      ID3D11VertexShader*m_vertexShader;
      Microsoft::WRL::ComPtr<ID3D11InputLayout> m_inputLayout;

public:
      // Loads a compiled vertex shader from a CSO file
      void LoadFromFile(ID3D11Device *device,
```

```
            _In_ Platform::String^ filename);

      // Returns pointer to vertex shader
      ID3D11VertexShader* GetVertexShader() { return m_vertexShader; }

      // Returns pointer to input layout
      ID3D11InputLayout* GetInputLayout() { return m_inputLayout.Get();
}
};
```

```
// VertexShader.cpp
#include "pch.h"
#include "VertexShader.h"

void VertexShader::LoadFromFile(ID3D11Device *device,
      _In_ Platform::String^ filename)
{
// Read the file
Platform::Array<unsigned char, 1U>^ fileDataVS =
DX::ReadData(filename);

// Crreate the vertex shader from the file's data
DX::ThrowIfFailed(device->CreateVertexShader(fileDataVS->Data,
      fileDataVS->Length, nullptr, &m_vertexShader));

// Describe the layout of the data
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
      {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
      { "COLOR",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
                 D3D11_INPUT_PER_VERTEX_DATA, 0 },
      };

DX::ThrowIfFailed(device->CreateInputLayout
      (vertexDesc,ARRAYSIZE(vertexDesc), fileDataVS->Data,
      fileDataVS->Length, &m_inputLayout));
}
```

The code above reads a CSO file and creates a vertex shader from it called `m_vertexShader`. The `D3D_INPUT_ELEMENT_DESCRIPTION` is used to specify the exact layout of the vertices this shader will accept. Each element is described using the following.

- LPCSTR SemanticName;
- UINT SemanticIndex;
- DXGI_FORMAT Format;
- UINT InputSlot;
- UINT AlignedByteOffset;
- D3D11_INPUT_CLASSIFICATION InputSlotClass;
- UINT InstanceDataStepRate;

SemanticName: This is a string representing the intended use of the element. This should match the `VertexShaderInput` structure from the VertexShader.hlsl file.

SemanticIndex: This is used when there is more than one element with the same semantic name; we have only a `POSITION` and `COLOR`, so we do not use this index.

Format: This must be the exact format of the data for each element. We used `XMFLOAT3` to create our vertex buffer, so the format for both the color and the position elements is `DXGI_FORMAT_R32G32B32_FLOAT`. For a complete list of all the possible formats, right-click on this constant and select **Go to Definition** from the context menu. Once again, this needs to match the `Vertex` structure on the CPU, which we defined in the Model.h file as well as the `VertexShaderInput` structure in the VertexShader.hlsl file.

Aligned Byte Offset: This is the offset within the structure that the element falls. The `POSITION` element is first in our structure, so its offset is 0. The `COLOR` element is after the position element, which is 3 floats or 12 bytes wide. The offset of the `COLOR` element within the structure is 12. For convenience, you can use `D3D11_APPEND_ALIGNED_ELEMENT` for this value and the data will be packed automatically. Be aware that `D3D11_APPEND_ALIGNED_ELEMENT` could not possibly know the correct way to pack all data, and it relies on the CPU to use standard aligned packing.

Input Slot Class: This is the class of the data. Each vertex has a position and color so we use `D3D11_INPUT_PER_VERTEX_DATA` here.

The `InputSlot` and `InstanceDataStep` are used when there is more than one assembler or for doing instancing; these can be ignored for the time being.

Next, we will add a class to wrap up our pixel shader. Add two files, **PixelShader.h** and **PixelShader.cpp**. The code for these files is specified in the following two code tables.

```
// PixelShader.h
#pragma once

#include "DirectXHelper.h"
```

```
class PixelShader
{
private:
      ID3D11PixelShader* m_pixelShader;

public:
      // Loads a compiled pixel shader from a CSO file
      void LoadFromFile(ID3D11Device *device,
      _In_ Platform::String^ filename);

      // Returns pointer to pixel shader
      ID3D11PixelShader* GetPixelShader() { return m_pixelShader; }
};
```

```
// PixelShader.cpp
#include "pch.h"
#include "PixelShader.h"

void PixelShader::LoadFromFile(ID3D11Device *device,
      _In_ Platform::String^ filename)
{
// Read the file
Platform::Array<unsigned char,
      1U>^  fileDataPS = DX::ReadData(filename);

// Create a pixel shader from the data in the file:
DX::ThrowIfFailed(device->CreatePixelShader(fileDataPS->Data,
      fileDataPS->Length, nullptr, &m_pixelShader));
}
```

The `PixelShader` class is a little simpler than the `VertexShader` class, since we do not need to describe the layout of the data. It just reads the file and creates a shader from it.

Next, we need to add references to the shader headers (PixelShader.h and VertexShader.h) to the SimpleTextRenderer.h file. These changes are highlighted in the following code table.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "Model.h"
#include "VertexShader.h"
#include "PixelShader.h"
```

Add two new member variables to this class to hold our shaders. I have called them `m_vertexShader` and `m_pixelShader`, and I have added them to the end of the class declaration.

```
private:
	Model *m_model;
	Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
	ModelViewProjectionConstantBuffer m_constantBufferCPU;

	// Shaders
	VertexShader m_vertexShader;
	PixelShader m_pixelShader;
};
```

Now that we have declared our shaders, we need to have them load the appropriate files. The shaders are a device dependent resource, so this should be done in the `CreateDeviceResources` method of the `SimpleTextRenderer` class, in the SimpleTextRenderer.cpp file. I have placed this after the creation of the constant buffer and highlighted the changes in the following code table.

```
	// Create the constant buffer on the device
	CD3D11_BUFFER_DESC constantBufferDesc(sizeof
		(ModelViewProjectionConstantBuffer),
D3D11_BIND_CONSTANT_BUFFER);
	DX::ThrowIfFailed(m_d3dDevice->CreateBuffer(&constantBufferDesc,
		nullptr, &m_constantBufferGPU));

	// Load the shaders from files (note the CSO extension, not
hlsl!):
	m_vertexShader.LoadFromFile(m_d3dDevice.Get(),
"VertexShader.cso");
	m_pixelShader.LoadFromFile(m_d3dDevice.Get(), "PixelShader.cso");
}
```

We should position our triangle in world space by setting the `m_constantBufferCPU.model` matrix. I will place the triangle at the origin (`0.0f, 0.0f, 0.0f`). The position of the triangle can be set in the `SimpleTextRenderer::Update` method. The following code table highlights the code to position the model.

```
		XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f),// Look at
		XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f)   // Up vector
		)));

// Position the triangle model
```

```
XMStoreFloat4x4(&m_constantBufferCPU.model,
      XMMatrixTranspose(
      XMMatrixTranslation(0.0f, 0.0f, 0.0f)));
}
```

Rendering the Model

Finally, we can now render the triangle with the shaders. There are a few steps to rendering, but most of them are called once per call to `Render`. The steps to rendering are as follows:

- Clear the Depth Stencil Buffer: We have begun a new round of rendering and the depth data in the depth stencil must be reset. For the current round of rendering, the GPU hasn't examined any pixels yet, and there are presently no pixels in front of or behind any others.
- Set the Render Target: The GPU needs to know the render target; this is where it is rendering to. In our present example, we are rendering to a target that will shortly be flipped to the screen and shown to the user.
- Reset the index of the current vertex being rendered: The GPU will maintain the index of the last vertex it rendered from the previous round of rendering. We want it to start rendering our model again from vertex number 0.
- Set resources to point to our triangle and shaders: We need to set all the resources (the input layout, the subresoucres, the current vertex and pixel shaders, the constant buffer) to point to our triangle, accompanying buffers, and shaders.

The final `Render` method, after adding all of these changes to the `SimpleTextRenderer` class, is presented as the following code table.

```
void SimpleTextRenderer::Render() {
      m_d3dContext->ClearRenderTargetView(m_d3dRenderTargetView.Get(),
            (float*) &XMFLOAT3(0.39f, 0.58f, 0.93f)); // Clear to
cornflower blue

      // Clear the depth stencil
      m_d3dContext->ClearDepthStencilView(m_d3dDepthStencilView.Get(),
D3D11_CLEAR_DEPTH, 1.0f, 0);

      // Set the render target
      m_d3dContext->OMSetRenderTargets(1,
m_d3dRenderTargetView.GetAddressOf(), m_d3dDepthStencilView.Get());

      // Set to render triangles
      UINT stride = sizeof(Vertex);    // Reset to the frist vertices
in the buffer
      UINT offset = 0;
      m_d3dContext-
```

```
>IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
	
	// Set the input layout
	m_d3dContext->IASetInputLayout(m_vertexShader.GetInputLayout());

	// Set the vertex shader
	m_d3dContext->VSSetShader(m_vertexShader.GetVertexShader(),
nullptr, 0);

	// Set the vertex shader's constant buffer
	m_d3dContext->VSSetConstantBuffers(0, 1,
m_constantBufferGPU.GetAddressOf());

	// Set the pixel shader
	m_d3dContext->PSSetShader(m_pixelShader.GetPixelShader(),
nullptr, 0);

	// Load the data from the CPU into the GPU's constant buffer
	m_d3dContext->UpdateSubresource(m_constantBufferGPU.Get(), 0,
NULL, &m_constantBufferCPU, 0, 0);

	// Set the vertex buffer
	m_d3dContext->IASetVertexBuffers(0, 1, m_model-
>GetAddressOfVertexBuffer(), &stride, &offset);

	// Render the vertices
	m_d3dContext->Draw(m_model->GetVertexCount(), 0);

	m_d2dContext->BeginDraw();

	HRESULT hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET) {
		DX::ThrowIfFailed(hr);
	}
}
```

At this point, you should be able to compile and run the application and see a rather fetching colored triangle in the top half of the screen.

Chapter 5 Loading a Model

Hand coding models is not difficult if the models are very simple, like the triangle from the previous chapter. Using this technique to create complex models is not practical. It is far better to create models using a 3-D modeling application. The examples in this book were created using Blender 2.66, which is an open source 3-D modeling application available from http://www.blender.org/. Models can be created using Blender, or many other 3-D modelers, and exported to various 3-D file formats. The files can then be loaded into our applications. This allows for much more complex models since visually creating models is far simpler than working with the vertices themselves.

Object Model File Format

There are many 3-D file formats, each having advantages and disadvantages over the others. In this chapter, we will examine a simple but flexible format that is fairly universal and standardized. It is called the object file format, and it was originally developed by Wavefront Technologies. Object files have an “obj” file extension. We will write a class that parses object files and creates meshes from the vertex information in them.

Note: A mesh is a collection of triangles that forms a 3-D model. Mesh and model are interchangeable terms in the current context. In past versions of DirectX there was a mesh interface, but DirectX 11 does not contain this interface, so meshes must be loaded and manipulated manually. This is a little extra work, but it is a lot more flexible.

There are several advantages to the object file format. It is plain text and easy to read and parse. The files are human readable and writable; they are not binary. Human readable plaint text requires more storage space and is generally slower for the computer to parse than binary formats. For instance, the value 3.14159f takes 8 bytes in plaint text, but only 4 bytes in binary as a 32 bit float. The size of the data is negligible for the small models we will be working with, as is the time the CPU needs to parse the text into variables. For larger projects where the models are more complicated or there are a lot of them, you might think about either using a binary file format or writing your own format.

The file format we will be using is also a simplified version of the entire object file specification. We will write a class that loads only object files specifically designed for it. Object file parsers can be written to ignore any of the unknown syntaxes within the file. We can write an object file with texture coordinates and normal information, along with the vertex positions, and our program can ignore whatever specifications it does not need until we examine texturing and lighting.

Another downside to the format is that the vertex buffers are not indexed in an efficient manner. In a previous book, we looked at using index buffers to reference vertices multiple times on the GPU and it saved us some GPU memory. The method we will be examining is not recommended for complex geometry, because without using an indexed topology, the GPU must store many times more vertices than is actually necessary. For instance, we need only 8 vertices to store a cube on the GPU. We can efficiently index the vertices using an index buffer. However, the object files we will be loading store a cube as 12 vertices: 3 vertices per triangle, 2 triangles per face, and 6 faces.

Note: When you feel comfortable with this simple text based OBJ format, you could develop your own format specifically for storing the models in a manner best suiting the unique circumstance. We are examining the OBJ file format as an example of reading basic models. You can easily write your own custom 3-D file format which stores the information exactly as you need it, and which is far smaller and faster to parse than the OBJ format.

Adding a Model to the Project

Object files are stored using a small scripting language with a very simple syntax. It is a plain text file with lists or arrays of vertex positions and other information. Each element of the arrays is specified on its own line. There are arrays of vertex positions, texture coordinates, vertex normal, and faces. A face in this context is a flat surface. Faces are defined by a collection of indices that reference values in each of the arrays (the vertex position, texture coordinate, or normal arrays). A face definition can reference any of the values in each of the arrays, allowing for great flexibility. There are many other keywords that can be used in object files. The following is only a brief description of the ones we will be using in our program.

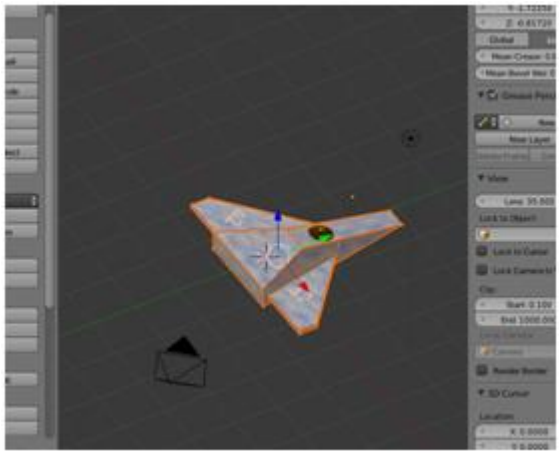

Figure 13: Figure 5.1: Spaceship in Blender

Figure 5.1 is a cropped screen shot of the spaceship model we will load as it appears in Blender. You can see a light source and a camera as well as the model. The light source and camera will not be exported to the object file, only the spaceship itself.

The following code is the object file for the spaceship in *Figure 5.1* as exported by Blender 2.66. The spaceship model is very simple, but even this simple mode creates a rather lengthy object file. The following code table shows the code for the spaceship.

```
# Blender v2.66 (sub 0) OBJ File: 'spaceship.blend'
# www.blender.org
v 0.245036 -0.277313 -3.008238
v 1.000000 -0.277313 0.638722
v -1.000000 -0.277313 0.638721
v -0.245036 -0.277313 -3.008238
v 0.332394 0.200748 -0.653899
v 0.999999 0.200748 0.638722
v -1.000000 0.200748 0.638721
v -0.332394 0.200748 -0.653899
v 0.924301 -0.171645 0.476444
v 0.294592 -0.171645 -1.523556
v 2.173671 -0.171645 0.066711
v 2.173671 -0.171645 0.476444
v 0.924301 -0.071908 0.476444
v 0.294592 -0.071908 -1.523556
v 2.173671 -0.071908 0.066711
v 2.173671 -0.071908 0.476444
v -0.921304 -0.073625 0.467630
v -0.288946 -0.073625 -1.532370
v -2.170674 -0.073627 0.057897
v -2.170674 -0.073627 0.467630
v -0.921304 -0.173363 0.467630
v -0.288946 -0.173363 -1.532370
v -2.170674 -0.173364 0.057897
v -2.170674 -0.173364 0.467630
vt 0.585247 0.203295
vt 0.000325 0.320701
vt 0.000323 0.000323
vt 0.800338 0.208366
vt 0.693343 0.208366
vt 0.585894 0.000324
vt 0.606542 0.475453
vt 0.219632 0.475452
vt 0.027282 0.321348
vt 0.929796 0.869160
vt 0.929795 0.946102
```

```
vt 0.607903 0.869160
vt 0.007848 0.968632
vt 0.000323 0.892059
vt 0.607257 0.968632
vt 0.585894 0.213707
vt 0.972548 0.227766
vt 0.972548 0.306642
vt 0.607189 0.747112
vt 0.810796 0.747112
vt 0.608702 0.773718
vt 0.577718 0.698455
vt 0.577718 0.477807
vt 0.606542 0.476099
vt 0.465133 0.534562
vt 0.399188 0.534562
vt 0.399188 0.518510
vt 0.557042 0.517863
vt 0.400036 0.517863
vt 0.399188 0.497305
vt 0.819439 0.606656
vt 0.999676 0.321348
vt 0.999677 0.717546
vt 0.626993 0.719207
vt 0.607189 0.656306
vt 0.818792 0.658818
vt 0.607189 0.719854
vt 0.810834 0.719854
vt 0.812349 0.746465
vt 0.830351 0.868513
vt 0.609613 0.868513
vt 0.828641 0.839677
vt 0.531725 0.534562
vt 0.465780 0.534562
vt 0.531725 0.518510
vt 0.399188 0.476099
vt 0.556194 0.476099
vt 0.557042 0.496658
vt 0.153905 0.688352
vt 0.439413 0.868513
vt 0.000323 0.818146
vt 0.000323 0.495980
vt 0.063200 0.476099
vt 0.060943 0.687706
vt 0.584527 0.121258
```

```
vt 0.907786 0.000323
vt 0.000323 0.393413
vt 0.607903 0.946102
vt 0.233352 0.869160
vt 0.585894 0.320701
vt 0.812309 0.773718
vt 0.606542 0.696746
vt 0.465133 0.518510
vt 0.556194 0.497305
vt 0.949339 0.760147
vt 0.818792 0.321348
vt 0.608703 0.746465
vt 0.607903 0.839677
vt 0.465780 0.518510
vt 0.400036 0.496658
vt 0.042889 0.868513
vt 0.398542 0.687706
vn 0.000000 -1.000000 -0.000000
vn -0.000000 1.000000 0.000000
vn 0.757444 0.633792 -0.156800
vn -0.000000 -0.000001 1.000000
vn -0.979238 -0.000001 -0.202714
vn 0.000000 0.980001 -0.198995
vn -0.953839 0.000000 0.300320
vn 0.646008 0.000000 -0.763331
vn 1.000000 -0.000000 0.000000
vn 0.000000 0.000000 1.000000
vn 0.953476 0.000000 0.301469
vn -0.645477 0.000000 -0.763779
vn -1.000000 -0.000000 -0.000000
vn -0.000001 1.000000 -0.000000
vn 0.000001 -1.000000 0.000000
vn 0.888496 0.000002 -0.458885
vn -0.382358 0.902665 -0.197479
s off
f 1/1/1 2/2/1 3/3/1
f 5/4/2 8/5/2 7/6/2
f 1/7/3 5/8/3 2/9/3
f 2/10/4 6/11/4 3/12/4
f 3/13/5 7/14/5 4/15/5
f 5/16/6 1/17/6 4/18/6
f 13/19/7 14/20/7 9/21/7
f 14/22/8 15/23/8 11/24/8
f 15/25/9 16/26/9 12/27/9
```

```
f 16/28/10 13/29/10 9/30/10
f 9/31/1 10/32/1 11/33/1
f 16/34/2 15/35/2 13/36/2
f 21/37/11 22/38/11 18/39/11
f 22/40/12 23/41/12 18/42/12
f 23/43/13 24/44/13 19/45/13
f 24/46/10 21/47/10 17/48/10
f 17/49/14 18/50/14 20/51/14
f 24/52/15 23/53/15 21/54/15
f 4/55/1 1/1/1 3/3/1
f 6/56/2 5/4/2 7/6/2
f 5/8/16 6/57/16 2/9/16
f 6/11/10 7/58/10 3/12/10
f 7/14/17 8/59/17 4/15/17
f 8/60/6 5/16/6 4/18/6
f 14/20/7 10/61/7 9/21/7
f 10/62/8 14/22/8 11/24/8
f 11/63/9 15/25/9 12/27/9
f 12/64/10 16/28/10 9/30/10
f 12/65/1 9/31/1 11/33/1
f 15/35/2 14/66/2 13/36/2
f 17/67/11 21/37/11 18/39/11
f 23/41/12 19/68/12 18/42/12
f 24/44/13 20/69/13 19/45/13
f 20/70/10 24/46/10 17/48/10
f 18/50/14 19/71/14 20/51/14
f 23/53/15 22/72/15 21/54/15
```

There are several methods for adding this code to your project. You could add a text file to your solution and copy the above code, but usually models are added as references to external files. Copy the above code and save it to a text file called **spaceship.obj**. Right-click on the project name in the solution explorer and select **Add existing item**. Find the spaceship.obj file and add it to the project.

Visual Studio will assume the file is a regular relocatable machine code object file, because of the obj extension. This is no good, our file is a model, and it is not a binary code file. Right-click on the **spaceship.obj** file in your solution explorer and select **Properties** from the context menu. You will see the file properties form where you can change the Item **Type value** from **Object** to **Text** (see *Figure 5.2*).

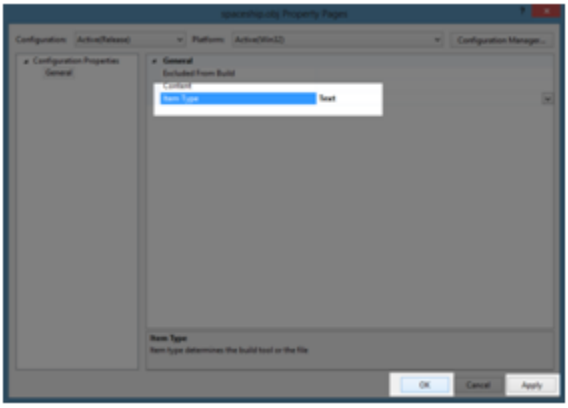

Figure 14: Figure 5.2: Change Object File Item Type

If you do not change the file type, Visual Studio will think the file is corrupted, as it clearly does not contain the information that a standard binary object file should contain. Click **Apply** after you change the item type setting, then click **OK**.

OBJ File Syntax

Comments: Any line that begins with the cross hatch or hash symbol (#) is a comment. The first line in the spaceship.obj file is a comment that Blender 2.66 wrote to identify the modelling application and the second line is a comment with Blender's website.

Vertices: Lines beginning with the "v" keyword specify vertex positions. The spaceship.obj file above has the vertices specified directly after the initial comments, but the actual order is irrelevant. Vertices could be mixed in with faces, normal, and texture coordinates. The "v" is followed by a space delimited series of three or four floating point values. The values specify the location of the vertex in 3-D space using the order X, Y, Z with an optional W component. For instance, the following specification is a vertex positioned at X=0.245036; Y=-0.277313; and Z=-3.008238:

```
v 0.245036 -0.277313 -3.008238
```

If there is a W component, our model reader will ignore it. The W is usually included to assist in matrix multiplication and other manipulations. Modern computers process four floating point values as easily as three, since they do so in a SIMD manner. If the W component is included, it will almost always be set to either all 1.0f or all 0.0f.

Texture Coordinates: Texture coordinates are specified using the "vt" keyword. The "vt" is followed by two or three values that represent U, V, and the optional W coordinate of the texture. For our purposes, textures use only the (U, V) coordinates, and they will be standard normalized UV texture space. These values will range from 0.0f to 1.0f. We will look in detail at UV coordinates and texturing later. The texture coordinates have been included in this file, even though we will not wrap a texture around our model in this chapter. The following line specifies a texture coordinate with U=0.585247 and V=0.203295:

```
vt 0.585247 0.203295
```

Vertex Normals: Vertex normals are specified using the "vn" keyword. The "vn" is followed by three floating point values delimited with spaces. We will examine what normals are and a very common use of them when we look into lighting a little later. The values are in the order (X, Y, Z) so the following line specifies a normal which points in the direction of X=0.757444; Y=0.633792; and Z=-0.156800:

```
vn 0.757444 0.633792 -0.156800
```

The values from a normal definition in the object file format specify a vertex normal, or the direction in which a vertex is pointing. They are not face or surface normals. The value is normalized and will define a normal with length 1.0f. A common strategy for extracting a face normal from a collection of vertex normals is to average the normals of the vertices that define the face. The normalized average of the vertex normals is used as the face normal.

Faces: Faces are defined on lines beginning with the "f" keyword. Lines beginning with the "f" keyword can be broken into space delimited vertices, each described with a slash delimited list of references into the above mentioned arrays. The order that attributes are defined is position, texture, and normal. A face can be described using three or more of these specifications, but for our purposes we will assume all faces are triangles. That means there will be three specifications per "f" keyword:

```
f 16/34/2 15/35/2 13/36/2
```

The above line specifies three vertices, each with its own position, texture coordinate, and normal. These vertices create a single 3-D triangle in our model. The first element, which I have colored blue, 16/34/2, means the 16th vertex position, the 34th texture coordinate, and the 2nd normal specified.

Note: The indices reference the arrays using a "1" based counting system. The first element in each array has a subscript of 1. C++ references arrays use a "0" based system, so when our program reads these lines, we must can subtract one from each index to convert between the two systems. Alternatively, we could add a dummy value to the C++ arrays at position 0, which is never referenced by the faces.

Note: Object files can be accompanied by a material file with an MTL extension. This file describes the material used by each of the objects described in the OBJ file. For our purposes, the materials will be hardcoded except for the UV texture coordinates.

Blender Export Settings

If you are using Blender 2.66, or a similar version with the same OBJ exporter, you might like to export your own models with the same settings as I have used to export this spaceship. Selecting similar options for an OBJ file export will create models that can be loaded using our `ModelReader` class. *Figure 5.3* shows a screen shot of the Blender OBJ exporter. The left panel is where these settings are changed.

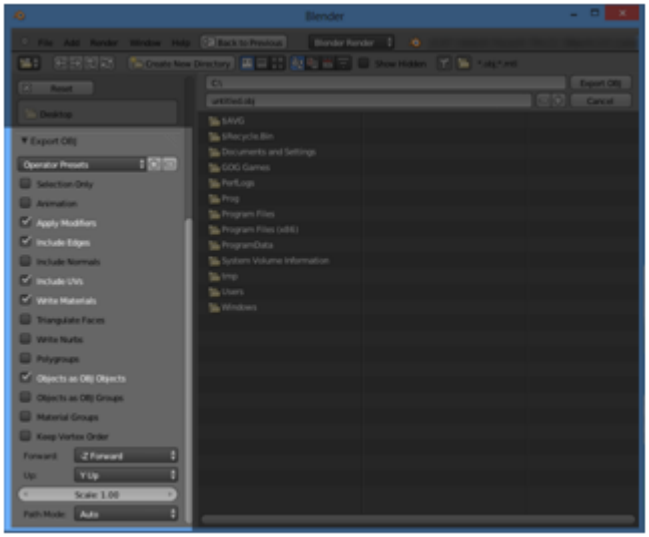

Figure 15: Figure 5.3: Setting OBJ Model Export Options

The following table (*Table 5.1*) shows the settings I chose to export the spaceship model.

Table 5.1: *Blender OBJ Export Settings*

Setting	Value
Selection Only	False
Animation	False
Apply Modifiers	False
Include Edges	False
Include Normals	True
Include UVs	True
Write Materials	False
Triangulate Faces	True
Write Nurbs	False
Polygroups	False

Setting	Value
Objects as OBJ Objects	False
Objects as OBJ Groups	False
Material Groups	False
Keep Vertex Order	False
Scale	1.00
Forward	-Z Forward
Up	Y Up
Path Mode	Auto

Note: Object files can contain more than one object. In our example, there is only the spaceship model, but the format allows for many models to be defined in a single file. Each object in a file is given a name, and each object has its own list of vertices, normals, texture coordinates, and faces. If there is more than one object, each is specified after the "o" keyword. The "o" keyword is followed by the particular object's name. All vertex information until the next "o" keyword correspond to the last named object.

Note: The OBJ file syntax allows for nontriangular faces to be specified. For instance, square faces can be constructed using four vertices, or pentagonal faces can be made from five. This format allows for more succinct face definitions. When a model is loaded using this capability, the faces must be converted to a collection of triangles, since DirectX renders meshes using triangles lists. In the following examples, I have assumed the faces are all triangles and I have exported models using the "Triangulate Faces" option that will be available in many 3-D modeling applications.

Model Class

Now that we have examined the basics of the object file format, we can write a class that loads and parses these files to create the mesh for a `Model` object in our application. I have called the new class `ModelReader`, and it can be added to the project as we left it at the end of the previous chapter. Add two files to your project, **ModelReader.h** and **ModelReader.cpp**. The following two code tables show the code for this new class.

```
// ModelReader.h
```

```
#pragma once

#include <Windows.h>
#include <string>
#include "Model.h"

using namespace DirectX;

class ModelReader
{
	// Reads and returns the data from a text file in the current
folder
	// this method also returns the size of the file in bytes
	static char* ReadFile(char* filename, int& filesize);

	// Methods for reading an int or float from a string
	static float ReadFloat(std::string str, int &start);
	static int ReadInt(std::string str, int &start);

public:
	static Model* ReadModel(ID3D11Device* device, char* filename);

	static std::string ReadLine(char* data, int length, int &start);
};
```

```
// ModelReader.cpp

#include "pch.h"
#include "ModelReader.h"
#include <fstream>
#include <vector>

using namespace DirectX;

// Reads a 32 bit float from a substring starting from character index
&start
float ModelReader::ReadFloat(std::string str, int &start)
{
std::string t;
while(str.data()[start] != ' ' && str.data()[start] != '/' &&
	str.data()[start] != '\n' && start < (int) str.length())
	t.push_back(str.data()[start++]);
```

```
start++;
// Parse to float and return
return (float)atof(t.data());
}

// Reads an int from a substring starting from the character index
&start
int ModelReader::ReadInt(std::string str, int &start)
{
std::string t;
while(str.data()[start] != ' ' && str.data()[start] != '/' &&
      str.data()[start] != '\n' && start < (int) str.length())
      t.push_back(str.data()[start++]);

start++;
// Parse to int and return
return atoi(t.data());
}

char* ModelReader::ReadFile(char* filename, int &filesize)
{
filesize = 0;
std::ifstream filestream;

// Open the file for reading
filestream.open(filename, std::ifstream::in);

// If the file could not be opened, return NULL
if(!filestream.is_open()) return NULL; // The file could not be opened

// Find the file's length
filestream.seekg(0, std::ios::end);
filesize = (int) filestream.tellg();

// Allocate space for the file's data in RAM
char* filedata = new char[filesize]; // Throws bad_alloc if there's
problems

// Read the data from the file into the array
filestream.seekg(0, std::ios::beg);   // Reset the file back to the
start
filestream.read(filedata, filesize); // Read the whole file
filestream.close();
```

```
return filedata;
}

Model* ModelReader::ReadModel(ID3D11Device* device, char* filename)
{
// Read the file
int filesize = 0;
char* filedata = ReadFile(filename, filesize);

// Parse the data into vertices and indices
int startPos = 0;
std::string line;

std::vector<float> vertices;
std::vector<int> vertexIndices;

int index; // The index within the line we're reading

while(startPos < filesize) {
      line = ReadLine(filedata, filesize, startPos);
      if(line.data()[0] == 'v' && line.data()[1] == ' ')
            {
            index = 2;
            // Add to vertex buffer
            vertices.push_back(ReadFloat(line, index)); // Read X
            vertices.push_back(ReadFloat(line, index)); // Read Y
            vertices.push_back(ReadFloat(line, index)); // Read Z
            // If there's a "W" it will be ignored
            }
      else if(line.data()[0] == 'f' && line.data()[1] == ' ')
            {
            index = 2;
            // Add triangle to index buffer
            for(int i = 0; i < 3; i++)
                  {
                  // Read position of vertex
                  vertexIndices.push_back(ReadInt(line, index));

                  // Read and ignore the texture and normal indices:
                  ReadInt(line, index);
                  ReadInt(line, index);
                  }
            }
```

```
      }

delete[] filedata;    // Deallocate the file data

// Subtract one from the vertex indices to change from base 1
// indexing to base 0:
for(int i = 0; i < (int) vertexIndices.size(); i++)
      {
      vertexIndices[i]--;
      }

// Create a collection of Vertex structures from the faces
std::vector<Vertex> verts;
int j = vertexIndices.size();
int qq = vertices.size();
for(int i = 0; i < (int) vertexIndices.size(); i++)
      {
      Vertex v;

      // Create a vertex from the referenced positions
      v.position = XMFLOAT3(
            vertices[vertexIndices[i]*3+0],
            vertices[vertexIndices[i]*3+1],
            vertices[vertexIndices[i]*3+2]);
      // Specify random colors for our vertices:
      v.color = XMFLOAT3(
            (float)(rand()%10) / 10.0f,
            (float)(rand()%10) / 10.0f,
            (float)(rand()%10) / 10.0f
            );

      verts.push_back(v); // Push to the verts vector
      }

// Create a an array from the verts vector.
// While we're running through the array reverse
// the winding order of the vertices.
Vertex* vertexArray = new Vertex[verts.size()];
for(int i = 0; i < (int) verts.size(); i+=3)
      {
      vertexArray[i] = verts[i+1];
      vertexArray[i+1] = verts[i];
      vertexArray[i+2] = verts[i+2];
      }
```

```
// Construct the model
Model* model = new Model(device, vertexArray, verts.size());

// Clear the vectors
vertices.clear();
vertexIndices.clear();
verts.clear();

// Delete the array/s
delete[] vertexArray;

return model; // Return the model
}

// This method reads and returns a single '\n' delimited line
// from the *data array beginning at the index start. It reads
// the line and advances the start index.
std::string ModelReader::ReadLine(char* data, int length, int &start)
{
std::string str;
int index = 0;
while(data[start] != '\n')
      {
      // Ignore whitespace at the start of the string:
      if((data[start] == ' ' || data[start] == '\t') && index == 0)
            {
            start++;
            if(start == length) break;
            else continue;
            }

      index = 1;
      str.push_back(data[start++]);
      if(start == length) break;
      }
start++;

return str;
}
```

The ModelReader class is designed to be used statically; models are read from files using the ReadModel method. It is a helper class that loads text from a file and parses it into a simple array of Vertex structures, which are then used to generate a Model object. The spaceship.obj file contains texture coordinates and normal, but these are presently ignored by ModelLoader,

and only the vertex positions are read from the file. The ReadModel method returns a new Model object that has the vertices specified in the file. At present, our vertex structure (defined in the Model class) contains positions as well as colors. We will look at texturing shortly, but for the moment I have used the rand() method to generate random colors and ignored the UV coordinates. This will allow us to see our object's shape without reading and mapping the texture.

There are a few interesting things to note about the code for the `ModelReader::ReadModel` method that may help you write a custom model reading class. It reads the model's vertices into a `std::vector` because we do not know how many vertices are in the file and they are dynamically sized. After the vertices and indices are read from the file, all the indices must be reduced by 1, because OBJ files use a base 1 counting system to reference elements in the arrays, and C++ uses a base 0 counting system as mentioned.

After this, I have made an additional temporary vector called `verts`, which is used to convert the separate arrays of positions, UV's, and normal into instances of our `Vertex` structure. This step and the next could be combined into a single `for` loop. I have kept them separate for clarity in the example code.

Finally, something to be very careful of is the winding order of the 3-D program and the 3-D file format. DirectX uses a clockwise winding order to know which face is the front and which face is the back for each of the triangles in a mesh. The OBJ file format uses the reverse, a counter-clockwise winding order. For this reason, if we wish to render the OBJ file properly, we must reverse the winding order of the triangles. This is a simple matter of swapping any two of the points that create each triangle. I chose to swap them during the `for` loop of the final copy from the `verts std::vector` to the `Vector` structure array. If we do not do this, DirectX will think the backs of the triangles are facing the viewer, and it will cull the triangles and render black for the model.

To load a model, first include the ModelReader.h header in the **SimpleTextRenderer.h** file, see the following code table.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "Model.h"
#include "VertexShader.h"
#include "PixelShader.h"
#include "ModelReader.h"

// This class renders simple text with a colored background.
ref class SimpleTextRenderer sealed : public DirectXBase
```

In the `SimpleTextRenderer::LoadDeviceResources` method of the SimpleTextRenderer.cpp file, we can create our model. I have removed the code that we used to hard code our triangle and replaced it with new code that calls the `ModelReader::ReadModel` method. The following code table shows the changes to the code. I have commented out the code to be replaced, but this code can actually be deleted.

```
void SimpleTextRenderer::CreateDeviceResources()
{
	DirectXBase::CreateDeviceResources();

	// Define the vertices with the CPU in system RAM
	//Vertex triangleVertices[] =
	//	{
	//		{ XMFLOAT3(-1.0f,0.0f, 0.0f), XMFLOAT3(1.0f, 0.0f, 0.0f) },
	//		{ XMFLOAT3(0.0f, 1.0f, 0.0f), XMFLOAT3(0.0f, 1.0f, 0.0f) },
	//		{ XMFLOAT3(1.0f, 0.0f, 0.0f), XMFLOAT3(0.0f, 0.0f, 1.0f) }
	//	};

	// Create the model instance from the vertices:
	//m_model = new Model(m_d3dDevice.Get(), triangleVertices, 3);

	// Read the spaceship model
	m_model = ModelReader::ReadModel(m_d3dDevice.Get(), "spaceship.obj ");

	// Create the constant buffer on the device
```

You should be able to run the application at this point and see a colored spaceship model; it will not look like a spaceship, because the camera is rather close and looking directly at the back of the model. Let's raise the camera a little so we can get a better view of the mesh. The camera's position is set in the `SimpleTextRenderer::Update` method; change the Y value to `5.0f` so the camera is above and behind the mesh looking down. This change is highlighted in the following code table.

```
	// View matrix defines where the camera is and what direction it is looking in
	XMStoreFloat4x4(&m_constantBufferCPU.view, XMMatrixTranspose(
		XMMatrixLookAtRH(
			XMVectorSet(0.0f, 5.0f, 2.0f, 0.0f),// Position
			XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f),// Look at
			XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f)   // Up vector
```

```
                    )));
```

Now when you run the application you should see the colored spaceship in *Figure 5.4*. It looks a little strange because it's being rendered with random colored triangles. Only the vertices from the original model are actually being applied.

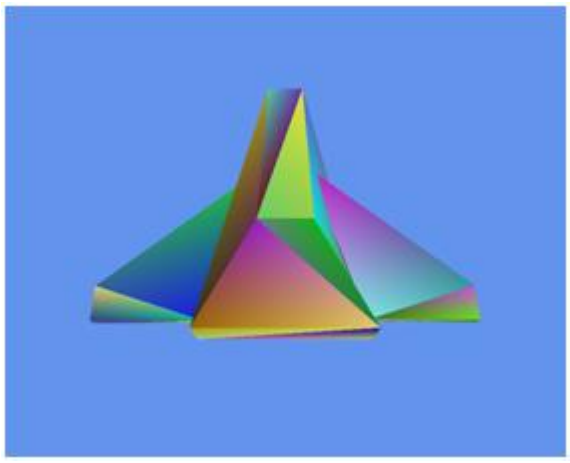

Figure 16: Figure 5.4: Spaceship Model with Vertices

Chapter 6 Texture Mapping

Models can be made to look much more realistic by applying a texture. A texture is a flat image file which is wrapped around a polygon. For instance, a model of a table could have a wooden texture applied to it, characters might have a skin and facial features textures applied, and the ground in a 3-D scene might have a grass texture.

The process of wrapping a 3-D model in a texture is called texture mapping. The mapping part of the term comes about because the 2-D image's pixels must be mapped to their corresponding vertices in the 3-D model. Most models are more complex than basic rectangles, and 3-D modeling applications are usually used to generate the texture coordinates which map to the vertices.

Texel or UV Coordinates

A 2-D image is wrapped around a mesh to create the illusion of a complex colored object. The pixels in the 2-D image are usually referenced using what is called a texel, or UV coordinates. Instead of using X and Y, texel coordinates are described with U and V components. The U component is the same as X, in it represents the horizontal axis, and the V component is analogous to the Y-axis, representing the vertical axis. The main difference between UV coordinates and standard Cartesian XY coordinates is that UV coordinates reference positions in an image as normalized values between 0.0 and 1.0. The point (0.0, 0.0) references the top left corner of the image and (1.0, 1.0) references the lower right corner. Every point in the image has a UV coordinate between (0.0, 0.0) and (1.0, 1.0). *Figure 6.1* is a grassy texture that could be applied to the ground of a video game. Several points in the texture have been highlighted to show what the UV values coordinates.

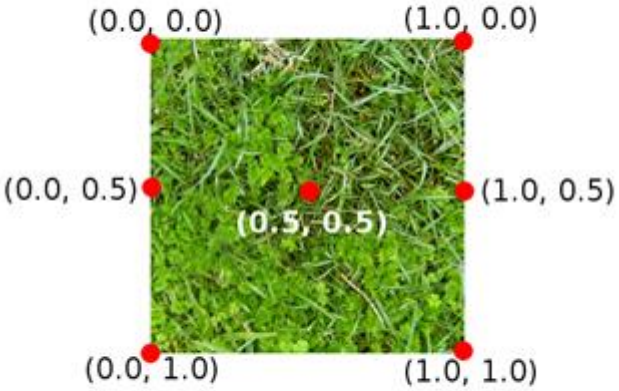

Figure 17: Figure 6.1: Texel/UV Coordinates

Texel coordinates do not reference particular pixels in the image; they are normalized positions. The GPU is very efficient at sampling pixels or collections of pixels referenced in this way. If a texture is very detailed, the texel coordinates may be read for multiple pixels in the texture and averaged for the final color on screen. Vice versa is also common; the texture may be small and lack separate colors for each pixel in the final image. In this case, the colors of the texture will be stretched over the faces to which the texture is mapped.

UV Layouts

It is easy to map a rectangular or square texture to a 3-D rectangle object, but mapping a 2-D texture to a mesh, like our spaceship from the previous chapter, is a little trickier. It is best to use the 3-D modeling application to export a UV layout from the mesh. The 3-D modeling software is also able to generate and export the UV coordinates that reference the layout. We saw these coordinates in the previous chapter's object file code. I've used Blender to export the UV layout for our spaceship model pictured in *Figure 6.2*; this was the spaceship.png UV layout. The UV layout is a collection of the outlines of the faces from a model. We can paint these shapes using a standard 2-D image editor. These shapes are referenced in the spaceship.obj file as the VT coordinates.

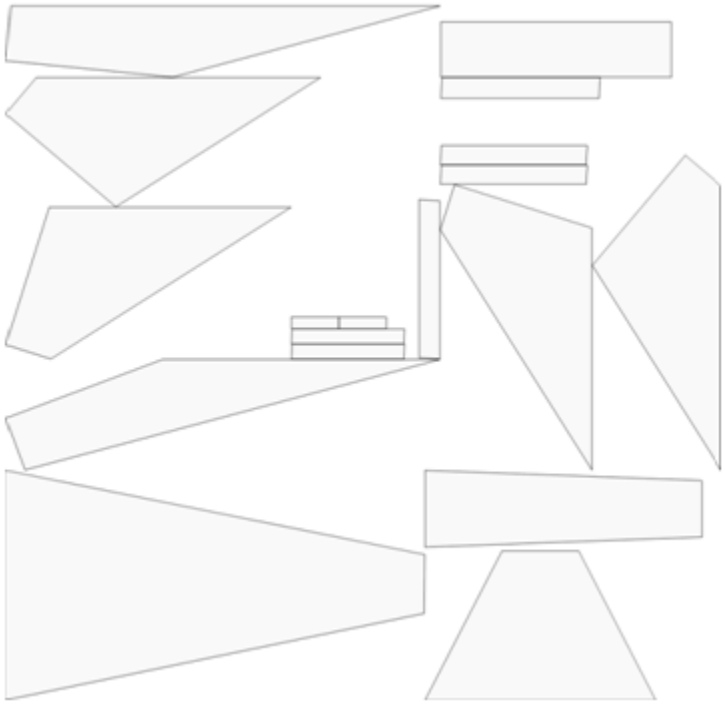

Figure 18: Figure 6.2: Spaceship UV Layout

Each of the shapes in *Figure 6.2* corresponds to a face in the spaceship. I have used Gimp 2.8.4 to create the textures in this book. Gimp is available from (http://www.gimp.org/). If you want to create your own texture for the spaceship, copy the above image into a 2-D image application such as Gimp and save it as spaceship.png. In *Figure 6.3,* I have colored the shapes using a blue metal texture.

Note: The textures in this book were exported as PNG image files. PNG is good because it uses lossless compression, so the textures will not be damaged by compression artifacts like a JPEG, but they are smaller than a standard Bitmap file. PNG also supports transparency. When you copy the images from the book you may find that the colors differ. For instance, the background may be black instead of transparent. The background is not mapped or referenced by the UV coordinates in the OBJ file, and it will not be visible.

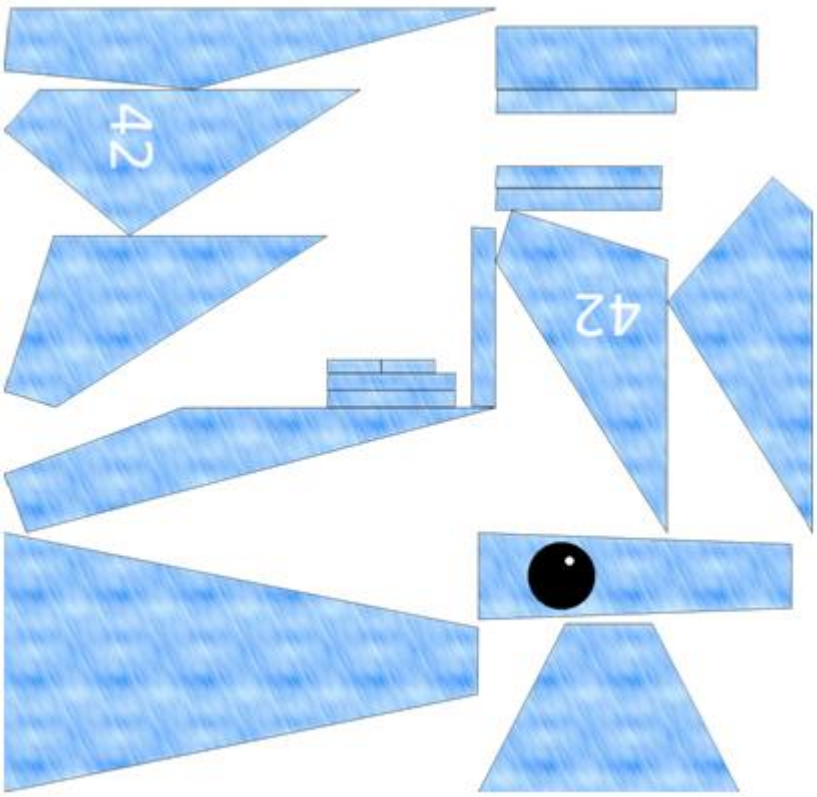

Figure 19: Figure 6.3: Textured UV Layout

In *Figure 6.3*, I have painted the UV with a blue steel texture. Copy and save the new textured UV layout from *Figure 6.3* or create your own based on the UV layout in *Figure 6.2*. Once you have designed or saved the texture as **spaceship.png**, import the image into our Visual Studio project by right-clicking on the project name in the solution explorer, selecting **Add Existing Item**, and locating the PNG file.

Note: The original texture of Figure 6.3 was exported from Blender to a 1024x1024 PNG image. 2-D textures consume width*height*3 bytes of memory or width*height*4 with the alpha channel. A modest 128x128 or 256x256 PNG should be large enough to maintain detail in the texture and yet small enough so as not to consume all the memory of a portable device that is running the application. UV coordinates use percentages from 0.0 to 1.0 so textures can be scaled to suit the memory restrictions of the target device, and the UV coordinates referenced in the model's mesh do not need to be changed.

Reading a Texture from a File

Now that we have added the PNG file to our project, we need to read the texture into our application and create an ID3D11Texture2D from it. This is the Direct3D object that represents a 2-D texture. We are using a standard PNG file, which Windows 8 has decoders for, so we can use the Windows Imaging Component (WIC) to read the file and turn it into an array of RGBA color values. We can then create the texture on the device from this array. We will wrap this functionality in a new class called Texture. Add two new files to the project, **Texture.h** and **Texture.cpp**. The code listings for this class are presented the following two code tables.

```
// Texture.h
#pragma once

#include <Windows.h>
#include <string>
#include "Model.h"

using namespace DirectX;

// This class reads an image from a file and creates an
// ID3D11Texure2D and a ID3D11ShaderResourceView from it
class Texture
{
	Microsoft::WRL::ComPtr<ID3D11Texture2D> m_texture;
	Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> m_resourceView;
public:
	// Read a texture and create device resources
	void ReadTexture(ID3D11Device* device, IWICImagingFactory2*
wicFactory, LPCWSTR filename);

	// Getters
	Microsoft::WRL::ComPtr<ID3D11Texture2D> GetTexure() { return
m_texture; }
```

```
    Microsoft::WRL::ComPtr<ID3D11ShaderResourceView>
GetResourceView() { return m_resourceView; }
};
```

```
// Texture.cpp.cpp
#include "pch.h"
#include "Texture.h"

void Texture::ReadTexture(ID3D11Device* device, IWICImagingFactory2*
wicFactory, LPCWSTR filename)
{
// Create a WIC decoder from the file
IWICBitmapDecoder *pDecoder;
DX::ThrowIfFailed(wicFactory->CreateDecoderFromFilename(filename,
    nullptr, GENERIC_READ, WICDecodeMetadataCacheOnDemand,
&pDecoder));

// Create a frame, this will always be 0, PNG have only 1 frame
IWICBitmapFrameDecode *pFrame = nullptr;
DX::ThrowIfFailed(pDecoder->GetFrame(0, &pFrame));

// Convert the format to ensure it's 32bpp RGBA
IWICFormatConverter *m_pConvertedSourceBitmap;
DX::ThrowIfFailed(wicFactory-
>CreateFormatConverter(&m_pConvertedSourceBitmap));
DX::ThrowIfFailed(m_pConvertedSourceBitmap->Initialize(
    pFrame, GUID_WICPixelFormat32bppPRGBA, // Pre-multiplied RGBA
    WICBitmapDitherTypeNone, nullptr,
    0.0f, WICBitmapPaletteTypeCustom));

// Create a texture2D from the decoded pixel data
UINT width = 0;
UINT height = 0;
m_pConvertedSourceBitmap->GetSize(&width, &height);
int totalBytes = width * height * 4; // Total bytes in the pixel data

// Set up a rectangle which represents the size of the entire image:
WICRect rect;
rect.X = 0; rect.Y = 0; rect.Width = width; rect.Height = height;

// Copy the entire decoded bitmap image to a buffer of bytes:
BYTE *buffer = new BYTE[totalBytes];
```

```
DX::ThrowIfFailed(m_pConvertedSourceBitmap->CopyPixels(&rect, width *
4, totalBytes, buffer));

// Describe the texture we will create:
D3D11_TEXTURE2D_DESC desc;
ZeroMemory(&desc, sizeof(D3D11_TEXTURE2D_DESC));
desc.Width = width;
desc.Height = height;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_IMMUTABLE;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

// Create the sub resource data which points to our BYTE *buffer
D3D11_SUBRESOURCE_DATA subresourceData;
ZeroMemory(&subresourceData, sizeof(D3D11_SUBRESOURCE_DATA));
subresourceData.pSysMem = buffer;
subresourceData.SysMemPitch = (width * 4);
subresourceData.SysMemSlicePitch = (width * height * 4);

// Create the texture2d
DX::ThrowIfFailed(device->CreateTexture2D(&desc, &subresourceData,
m_texture.GetAddressOf()));

// Create a resource view for the texture:
D3D11_SHADER_RESOURCE_VIEW_DESC rvDesc;
rvDesc.Format = desc.Format; // Use format from the texture
rvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D; // Resource is a
2D texture
rvDesc.Texture2D.MostDetailedMip = 0;
rvDesc.Texture2D.MipLevels = 1;

// Create resource view:
DX::ThrowIfFailed(device->CreateShaderResourceView(
     m_texture.Get(), &rvDesc,
     m_resourceView.GetAddressOf()));

// Delete the WIC decoder
pDecoder->Release();
```

```
pFrame->Release();
m_pConvertedSourceBitmap->Release();

// Delete the pixel buffer
delete[] buffer;
}
```

The previous class uses the WIC decoders to read an image from a file. WIC decoders are able to read several common image formats. Once the decoder reads and decompresses the PNG image, the pixel data is copied to `BYTE* buffer`. We use a `D3D11_TEXTURE2D_DESC` structure to describe the texture being created, a `D3D11_SUBRESOURCE_DATA` structure to point to the texture's pixel data in system memory, and then we create the texture using the device's `CreateTexture2D` method. This will make a copy of the pixel data on the GPU. Once the pixel data is on the GPU, we do not need to maintain a copy in system RAM.

We also need to create an `ID3D11ShaderResourceView`. A resource can be made from multiple sub resources; an `ID3D11ShaderResourceView` is used to specify which sub resources can be used by the shaders and how the data in the resources is to be read.

Note: You will see references to MIP maps in the above previous code table. An MIP map, derived from the Latin "multum in parvo" or much in little, is a texture that is composed of the same texture at several levels of detail. MIP levels are used to improve performance. When the viewer of a 3-D object is very close to the object the most detailed texture can be applied, and when the viewer is far from the object a less detailed texture can be used. MIP maps are more advanced than what we are doing here, but they have a much better performance when rendering complex scenes.

Applying the Texture2D

We have loaded a model and we have a class to load the texture onto the GPU. We need to make some changes to the `Model` class and the `Vertex` structure to incorporate the new `Vertex` type with texture coordinates instead of colors. The `Vertex` structure in the Model.h file does not include texture coordinates; at the moment it's just positions and colors. We do not need colors anymore since our new `Vertex` structure will include UV coordinates instead. The following code table highlights the changes in the `Vertex` structure. The old `color` element has been commented out, but this line can be replaced by the new one which defines the `uv` element for the structure.

```
// Definition of our vertex types
struct Vertex
{
```

```
	DirectX::XMFLOAT3 position;
	// DirectX::XMFLOAT3 color;
	DirectX::XMFLOAT2 uv;
};
```

At the moment, our `ModelReader` is ignoring the `vt` texture coordinate specifications in the OBJ files it reads. Change the ModelReader.cpp file to read the texture coordinates. These changes have been highlighted in the following code table.

```
Model* ModelReader::ReadModel(ID3D11Device* device, char* filename)
{
// Read the file
int filesize = 0;
char* filedata = ReadFile(filename, filesize);

// Parse the data into vertices and indices
int startPos = 0;
std::string line;

// Vectors for vertex positions
std::vector<float> vertices;
std::vector<int> vertexIndices;
// Vectors for texture coordinates
std::vector<float> textureCoords;
std::vector<int> textureIndices;

int index; // The index within the line we're reading

while(startPos < filesize) {
	line = ReadLine(filedata, filesize, startPos);
	if(line.data()[0] == 'v' && line.data()[1] == ' ') {
		index = 2;
		// Add to vertex buffer
		vertices.push_back(ReadFloat(line, index)); // Read X
		vertices.push_back(ReadFloat(line, index)); // Read Y
		vertices.push_back(ReadFloat(line, index)); // Read Z
		// If there's a "W" it will be ignored
		}
	else if(line.data()[0] == 'f' && line.data()[1] == ' ') {
		index = 2;
		// Add triangle to index buffer
		for(int i = 0; i < 3; i++) {
			// Read position of vertex
```

```
                vertexIndices.push_back(ReadInt(line, index));
                // Read the texture coordinate
                textureIndices.push_back(ReadInt(line, index));

                // Ignore the normals
                ReadInt(line, index );
                }
          }
     else if(line.data()[0]=='v'&& line.data()[1] == 't' &&
line.data()[2] == ' ')
          {
          index = 3;
          // Add to texture
          textureCoords.push_back(ReadFloat(line, index)); // Read U
          textureCoords.push_back(ReadFloat(line, index)); // Read V
          }
     }

// Deallocate the file data
delete[] filedata;    // Deallocate the file data

// Subtract one from the vertex indices to change from base 1
// indexing to base 0:
for(int i = 0; i < (int) vertexIndices.size(); i++) {
     vertexIndices[i]--;
     textureIndices[i]--;
     }

// Create a collection of Vertex structures from the faces
std::vector<Vertex> verts;
int j = vertexIndices.size();
int qq = vertices.size();
for(int i = 0; i < (int) vertexIndices.size(); i++) {
     Vertex v;

     // Create a vertex from the referenced positions
     v.position = XMFLOAT3(
          vertices[vertexIndices[i]*3+0],
          vertices[vertexIndices[i]*3+1],
          vertices[vertexIndices[i]*3+2]);

     // Set the vertex's texture coordinates
     v.uv = XMFLOAT2(
          textureCoords[textureIndices[i]*2+0],
```

```
            1.0f-textureCoords[textureIndices[i]*2+1] // Negate V
            );

      verts.push_back(v); // Push to the verts vector
      }

// Create a an array from the verts vector.
// While we're running through the array reverse
// the winding order of the vertices.
Vertex* vertexArray = new Vertex[verts.size()];
for(int i = 0; i < (int) verts.size(); i+=3) {
      vertexArray[i] = verts[i+1];
      vertexArray[i+1] = verts[i];
      vertexArray[i+2] = verts[i+2];
      }

// Construct the model
Model* model = new Model(device, vertexArray, verts.size());

// Clear the vectors
vertices.clear();
vertexIndices.clear();
verts.clear();
textureCoords.clear();
textureIndices.clear();
// Delete the array/s
delete[] vertexArray;

return model; // Return the model
}
```

Next we can change the vertex and pixel shader HLSL files. The structure defined in these files should match the structure of new `Vertex` type with texture coordinates instead of colors. The altered code for these shaders is presented in the following code table for the vertex shader, and the second code table for the pixel shader.

```
// VertexShader.hlsl

// The GPU version of the constant buffer
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
      matrix model;
      matrix view;
      matrix projection;
```

```
};

// The input vertices
struct VertexShaderInput
{
      float3 position : POSITION;
      float2 tex : TEXCOORD0;
};

// The output vertices as the pixel shader will get them
struct VertexShaderOutput
{
      float4 position : SV_POSITION;
      float2 tex : TEXCOORD0;
};

// This is the main entry point to the shader:
VertexShaderOutput main(VertexShaderInput input)
{
      VertexShaderOutput output;
      float4 pos = float4(input.position, 1.0f);

      // Use constant buffer matrices to position the vertices:
      pos = mul(pos, model);      // Position the model in the world
      pos = mul(pos, view); // Position the world with respect to a
camera
      pos = mul(pos, projection);// Project the vertices
      output.position = pos;

      // Pass the texture coordinates unchanged to pixel shader
      output.tex = input.tex;

      return output;
}
```

```
// PixelShader.hlsl

Texture2D shaderTexture; // This is the texture
SamplerState samplerState;

// Input is exactly the same as
// vertex shader output!
```

```
struct PixelShaderInput
{
	float4 position : SV_POSITION;
	float2 tex: TEXCOORD0;
};

// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
	float4 textureColor =
		shaderTexture.Sample(samplerState, input.tex);

	// Return the color unchanged
	return textureColor;
}
```

The vertex shader just passes the UV coordinates on to the pixel shader. The pixel shader requires some additional resources; it needs the texture (`shaderTexture`) and it needs a texture sampler state. I've called it `samplerState`. The pixel shader looks up the pixels in the texture using the `Sample` method. It passes the `samplerState` and the coordinate that is `input.tex` as parameters to this method. This method returns the color of the pixel, which the shader can then return. The pixel shader is still responsible for providing a color for each pixel. It now does so by sampling the texture in order to determine the color for each pixel.

Next, we need to load our texture from file and assign it to our `Model`, `m_model`. Add an include for the Texture.h header to the `SimpleTextRenderer` class in the SimpleTextRenderer.h file. The following code table shows the newly included file.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "Model.h"
#include "VertexShader.h"
#include "PixelShader.h"
#include "ModelReader.h"
#include "Texture.h"
```

Add a member variable to this class; I have called mine `m_texture` in the code. The following code table highlights this modification in the SimpleTextRenderer.h file.

```
private:
	Model *m_model;
```

```
        Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
        ModelViewProjectionConstantBuffer m_constantBufferCPU;
        Texture m_texture;

        // Shaders
        VertexShader m_vertexShader;
        PixelShader m_pixelShader;
```

Textures are a device resource so they should be created in the `SimpleTextRenderer::CreateDeviceResources` method. Open the **SimpleTextRenderer.cpp** file and load the texture. I have placed this code directly after the `m_model` member variable is initialized. The following code table highlights the changes.

```
        DirectXBase::CreateDeviceResources();

        // Read the spaceship model
        m_model = ModelReader::ReadModel(m_d3dDevice.Get(),
"spaceship.obj");

        // Read the texture:
        m_texture.ReadTexture(m_d3dDevice.Get(), m_wicFactory.Get(),
L"spaceship.png ");

        // Create the constant buffer on the device
```

We also need to change our `VertexShader` class, since at the moment it is still expecting the data to contain `COLOR` values instead of `TEXCOORD`. Open the **VertexShader.cpp** file and alter the `vertexDesc`. These changes are highlighted in the following code table.

```
// Describe the layout of the data
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
        {
        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
                    D3D11_INPUT_PER_VERTEX_DATA, 0 },
        { "TEXCOORD",    0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
                   D3D11_INPUT_PER_VERTEX_DATA, 0 },
        };
```

Make sure you make two changes here; the semantic name is `TEXCOORD`, but equally important is the fact that type is `DXGI_FORMAT_R32G32_FLOAT`, and there are now only two elements, not three.

Next we can create an `ID3D11SamplerState` member variable, `m_samplerState`. The sampler state is an object that holds information about the current sampler. First, add an `m_samplerState` member variable to the **SimpleTextRenderer.h** file. The declaration of the new member variable is highlighted in the code in the following code table.

```
private:
      Model *m_model;
      Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
      ModelViewProjectionConstantBuffer m_constantBufferCPU;
      Texture m_texture;
      ID3D11SamplerState *m_samplerState;
```

Note: According to Microsoft, you do not need to use a sampler state. There is a default state that will be assumed should the code not include its own. The problem is that some machines, including Windows RT Surface, do not seem to include any sampler state by default and if you do not specify a sampler state yourself, you might end up looking at a black, untextured model when debugging on some devices.

Sampler states are device resources. Open the **SimpleTextRenderer.cpp** file and create the sampler state at the end of the `CreateDeviceResources` method. The additional code is highlighted in the following code table.

```
      // Load the shaders from files (note the CSO extension, not
hlsl!):
      m_vertexShader.LoadFromFile(m_d3dDevice.Get(),
"VertexShader.cso");
      m_pixelShader.LoadFromFile(m_d3dDevice.Get(), "PixelShader.cso");

      // Create the sampler state
      D3D11_SAMPLER_DESC samplerDesc;
      ZeroMemory(&samplerDesc, sizeof(D3D11_SAMPLER_DESC));
      samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
      samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
      samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
      samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
      samplerDesc.MipLODBias = 0.0f;
      samplerDesc.MaxAnisotropy = 1;
      samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
      samplerDesc.BorderColor[0] = 0;
      samplerDesc.BorderColor[1] = 0;
```

```
        samplerDesc.BorderColor[2] = 0;
        samplerDesc.BorderColor[3] = 0;
        samplerDesc.MinLOD = 0;
        samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

        DX::ThrowIfFailed(m_d3dDevice->CreateSamplerState
              (&samplerDesc, &m_samplerState ));
}
```

Examining the previous code table should give a good idea of what a sampler state is. It is a way of specifying how a texture is to be read. The above sampler state specifies that we want linear interpolation (`D3D11_FILTER_MIN_MAG_MIP_LINEAR`); this will smooth the texture and make it appear less pixelated when viewing close up. We have also specified that the texture wraps round in every axis with the `Addressxx` settings. This means that the sampler sees the texture as an infinite, repeating, tiled pattern.

The next thing we need to do is set the resources for the shaders as active. Open the `Render` method in the **SimpleTextRenderer.cpp** file and call the `m_d3dContext`'s set methods with the resources we have just created. This tells the GPU which shaders and other resources are active. Once the GPU knows which vertices to render, which shaders to use, and which sampler state to use, we can render the model using the `m_d3dContext`'s `Draw` method. The following code table shows these changes to the `Render` method.

```
        // Set the vertex buffer
        m_d3dContext->IASetVertexBuffers(0, 1, m_model-
>GetAddressOfVertexBuffer(), &stride, &offset);

        // Set the sampler state for the pixel shader
        m_d3dContext->PSSetSamplers(0, 1, &m_samplerState);

        // Set the resource view which points to the texture
        m_d3dContext->PSSetShaderResources(0, 1,
m_texture.GetResourceView().GetAddressOf());

        // Render the vertices
        m_d3dContext->Draw(m_model->GetVertexCount(), 0);
```

At this point you should be able to run the application and see an improved model spaceship, complete with the blue texture we painted earlier (*Figure 6.4*).

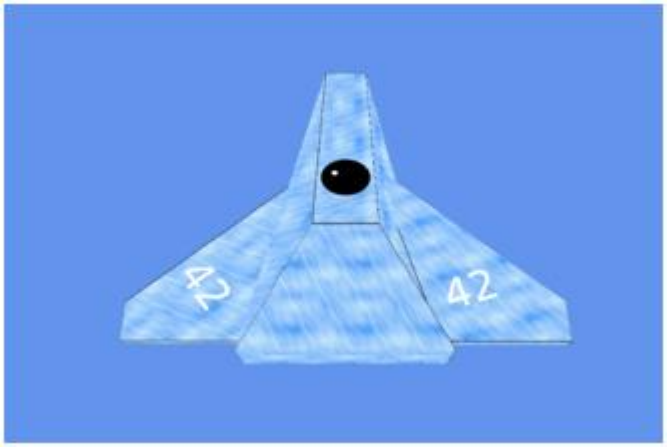

Figure 20: Figure 6.4: Textured Spaceship

Chapter 7 HLSL Overview

We've used the high level shader language a little in our code so far, and now we'll look in more detail at the language. It is a C based language, but it is designed specifically for the parallel architecture of the GPU. Code written in HLSL usually runs many times at once. For instance, a vertex shader's code might execute thousands of times, once for every vertex in a 3-D scene. The GPU does not do this sequentially, instead it executes thousands in parallel.

HLSL also has special data types and functions that are designed to make 3-D programming easier. The GPU has its own machine code that is completely different from the CPU's, and its instructions set is full of fast methods for vector and matrix operations.

Data Types

Most of the regular fundamental data types from C are available in HLSL. Depending on the hardware you are targeting, some may not be available. For instance, the `double` is only available on newer hardware.

The majority of HLSL variables are structures, vectors, or matrices. HLSL is far better at dealing with large amounts of similarly structured data than it is at dealing with single values. This is the opposite of the CPU. A CPU would tend to treat a 4x4 floating point matrix as 16 distinct floating point variables, whereas the GPU tends to operate on the 16 floating point at once in SIMD.

Scalar Types

The scalar types are single values like `int` or `bool`. They are used either as single values or in small collections as matrices, vectors, and structures:

- bool: Standard Boolean value, true of false.
- int: 32-bit signed integer.
- uint/dword: 32-bit unsigned integer.
- half: 16-bit IEEE floating point value. Do not use these, they are deprecated.
- float: 32-bit IEEE floating point value.
- double: 64-bit IEEE floating point value.

Note: The 16 bit IEEE half floating type was a way to compress 32 bit floats. With a loss in precision, 32 bit floats could be compressed to 16 bits. The data type is excellent for storing large amounts of data but it is now deprecated and it is recommended that you do not use it.

Semantic Names

We have used HLSL semantic names already; these names describe what a particular element of a structure will be used for. We describe the semantic names when we specify the layout of data in the C++ code, and we also describe it in the structure definitions in the shader's HLSL code. For instance, the following code is used in our `VertexShader` class. By the end of the last chapter we'd not yet included the NORMAL element shown in the following code table. We will add this element in *Chapter 8 Lighting* when we look at lighting.

```
// Describe the layout of the data
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
      {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
      { "NORMAL",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
                 D3D11_INPUT_PER_VERTEX_DATA, 0 },
      { "TEXCOORD",    0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
                 D3D11_INPUT_PER_VERTEX_DATA, 0 },
      };
```

The semantic names here are `POSITION`, `NORMAL` and `TEXCOORD`. These names are fairly self-explanatory. Elements with a `POSITION` semantic are generally going to be used to specify the positions of vertices and I'm sure you can guess the use of the other two semantics. For a complete list of the semantic names, visit the MSDN website:

http://msdn.microsoft.com/en-us/library/windows/desktop/bb509647%28v=vs.85%29.aspx.

The second parameter for each element in the previous code is a number that can be used to differentiate the values where the same semantic is used more than once. For instance, if you have a vertex type which uses more than one texture, like for bump mapping or various other tricks, you might include two `TEXCOORD` elements in the layout description. In this case, the first one could have a number 0 and the second could be number 1. These numbers are referenced in the shaders after the semantic names. Note the zeros after the semantic names in the code below. This code is the HLSL structure corresponding to the C++ description above as it might appear in a vertex shader, as shown in the following code table.

```
// The input vertices
struct VertexShaderInput
{
      float3 position : POSITION0;
      float3 normal : NORMAL0;
      float2 tex : TEXCOORD0;
};
```

You may have noticed that the vertex shader input specifies `POSITION` as the semantic and the output specifies `SV_POSITION`. This `SV_POSITION` is also used as the semantic in the pixel shader's HLSL code. The SV stands for system value. The output of a vertex shader, or the input of a pixel shader, must specify all four components of an `SV_POSITION` at a very minimum. The pipeline inherently knows the meaning of an `SV_POSITION` output from the vertex shader. An `SV_POSITION`, when it is the output from a vertex shader, has been transformed by the vertex shader to its final state; it is ready to be rasterized by the rasterizer stage of the pipeline.

Vector Types

Vectors are small arrays, between one and four components, of the same scalar data type. The term vector here has absolutely nothing to do with the C++ STL vector class that we used to read our model data earlier. Operations on vectors by the GPU are executed in a similar style to an SIMD, and they are very efficient. We have seen and used vectors in our code so far to store and manipulate the positions of vertices, the colors of the vertices, and the texture coordinates. The term vector, as used here, is more flexible than the standard definition in mathematics. In mathematics, a vector often specifies a magnitude or length and a direction. Vectors in DirectX are simply small collections of data. We can use a vector of floats in any way we please, and they need not specify a direction and magnitude. We have used DirectX vectors already in code to specify positions, texture coordinates, and colors.

Any of the scalar types mentioned above can be used to create a vector. There are two syntaxes, and the following is the first.

```
typeCount variableName;
```

Where `type` is one of the scalar types, count is 1, 2, 3, or 4, and represents the number of components in the vector; and `variableName` is the name of the variable being declared. You can initialize the values for a vector using curly braces. Here are some examples.

```
 int4 myIntVector;

float2 fv = { 0.5f, 0.5f };

double1 d = 0.6;
```

The other syntax uses the vector keyword. Other than the syntax, this does exactly the same thing.

```
Vector <type, count> variableName;
```

The following examples are the same as those above, only the syntax uses the `vector` keyword.

```
vector<int, 4> myIntVector;

vector<float, 2> fv = { 0.5f, 0.5f };
```

```
vector<double, 1> d = 0.6;
```

You can also initialize vectors using the following syntax.

```
int4 myVector = int4(1, 2, 3, 4);
```

Accessing Vector Elements

The elements can be accessed using the standard array notation, or you can use the color space notation (RGBA) or the coordinate notation (XYZW). These notations are sometimes called color component namespace or coordinate component namespace.

```
Vector <int, 4> myVector;
// Referencing the first element:
myVector[0] = 50;     // These three lines all do the same thing.
myVector.r = 50; // They set the first element of the vector to 50.
myVector.x = 50;


myVector[1] = 10;     // These three lines all set the second element
to 10.
myVector.g = 10;
myVector.y = 10;


myVector[2] = 20;     // These three lines set the third element to 20.
myVector.b = 20;
myVector.z = 20;


myVector[3] = 42;     // These lines set the final element of the
vector to 42.
myVector.a = 42;
myVector.z = 42;
```

Vector Swizzles

A swizzle is a method of accessing the components of a vector or matrix as a collection.

```
int4 v1 = { 1, 3, 9, 27 };

int4 v2 = { 2, 4, 6, 8 ;}

v1.rb = v2.gr;
```

In the above code there are two vectors being declared. The third line contains the swizzle. It assigns the g and r components of the v2 vector to the r and b components of the v1 vector. The v1 vector will contain { 4, 3, 2, 27 } after this operation; the vector v2 is not changed.

You cannot mix the color and coordinate component namespaces in a swizzle. The following is illegal because it uses both r and a from RGBA and X and Y from XYZW.

```
v1.rx = v2.ya;
```

You can reuse the same element more than once as the source.

```
v1.rgba = v2.rrrr;      // Broadcast the r element of v2 across v1
```

You cannot reuse the same element more than once as the destination (v1 in the following) because it is nonsense.

```
v1.rr = v2.gb;          // Illegal, cannot set v1.r to multiple values
```

Matrix Types

Matrix types are similar to vectors, only they can store more values and they can be 2-D. Each dimension of the matrices can range from one to four elements, so matrices can be anything from one to sixteen elements wide (which is a 1x1 or a 4x4 matrix). Like vectors, matrices are created as a collection of scalar data types, and every element in a matrix must have the same type. Also like vectors, there are two syntaxes for declaring a matrix, with and without the matrix keyword.

```
typeRowsxCols variableName;
```

Where `type` is the data type, `Rows` is the number of rows in the matrix, and `Cols` is the number of columns.

```
int3x2 neo;
```

```
float4x4 trinity;
float1x2 morpheus;
```

The other syntax using the `matrix` keyword achieves exactly the same thing.

```
matrix <type, rows, cols> variableName;
```

For example:

```
matrix <int, 3, 2> neo;
matrix <float, 4, 4> trinity;
matrix <float,1, 2> morpheus;
```

You can also initialize the values in a matrix using the curly braces, { and }.

```
matrix <int, 3, 2> neo = {
     1, 2, // First row
     3, 4, // Second row
     5, 6  // Third row
     };
```

Accessing Matrix Elements

Matrix elements can be accessed using either a zero based notation or a one based notation.

```
matrix<float, 4, 4> myMatrix = {
     1.0f, 2.0f, 3.0f, 4.0f,
     2.0f, 3.0f, 5.0f, 7.0f,
     1.0f, 2.0f, 6.0f, 24.0f
     1.0f, 1.0f, 2.0f, 3.0f
     };
myMatrix._m00 = 0.0f; // Change first element to 0, 0 based notation
myMatrix._11 = 12.9f; // Change first element to 12.9, 1 based notation
```

The element indexes begin with an underscore. For zero based notation, the indexes are prefixed with '_m'. For one based notation, the indexes are prefixed with '_' only. Note that depending on your display, some of the underscores '_' may appear as spaces in this text; be very careful not to mix the two. In both cases, the prefixes are followed by the index of the row and then column of the element to access.

To set the value of the element in the previous matrix that presently has the 7.0 to 100.0, we could do either of the following.

```
myMatrix._m23 = 100.0f;
myMatrix._34 = 100.0f;
```

Matrix elements can also be addressed using the standard C array notation. This is always zero based.

```
myMatrix[2][3] = 100.0f;
```

Matrix Swizzles

You can use swizzles to access and set the elements of matrices.

```
someMatrix._32_12 = someOtherMatrix._11_33;
```

This will copy elements `[1][1]` and `[3][3]` from `someOtherMatrix` to elements `[3][2]` and `[1][2]` of `someMatrix`.

Once again, you cannot mix the namespaces, so the one based indexing cannot be mixed with the zero based indexing in a single swizzle. This is illegal.

```
someMatrix._32_m12 = someOtherMatrix._11_33;
```

Also, it does not make sense to set the same element in a matrix to two or more different values. This is also illegal.

```
someMatrix._32_32 = someOtherMatrix._11_33;
```

Other Data Types

There are other data types, some of which we have seen. The `cbuffer` is used to hold data that is constant (from the GPU's perspective). The `Texture2D` is used to hold the texture for a pixel shader, and the `SamplerState` is used to hold data governing how a texture is sampled.

You can create structures with a syntax similar to C structures; the `struct` keyword is followed by the name and then the elements of the structure in curly braces.

```
struct someStructure {
     float x;
     float y;
```

```
    };
```

Operators

Scalar, vector, and matrix data types can all be manipulated using the standard operators (+, -, *, and /). With scalar data these operations perform as expected, but it is important to know that with vector and matrix types, the operators are element by element. This means the standard matrix product is not calculated using the multiplication operator "*", instead this operator will multiply corresponding elements from the two source operands. To calculate a standard matrix product, we must use the `mul` intrinsic. We have seen this when multiplying by the model, world, and projection matrices in our code. For a detailed explanation of matrix multiplication, have a look at the following websites.

From Wolfram:

http://mathworld.wolfram.com/MatrixMultiplication.html

From Wikipedia:

http://en.wikipedia.org/wiki/Matrix_multiplication

From Khan Academy:

https://www.khanacademy.org/math/algebra/algebra-matrices/matrix_multiplication/v/matrix-multiplication--part-1

Intrinsics

The GPU has its own processing unit, memory, and architecture. It even has its own machine code with a completely different set of instructions to the CPU. Many of the machine code instructions the GPU understands are designed specifically to assist in 3-D graphics. It is very efficient at matrix operations and operations on vector data types. Many of the instructions the GPU is capable of performing have no operators (like + or * can be used for addition or multiplication). The instructions are invoked using intrinsics that resemble regular function calls. These are special functions designed to closely match the machine code of the hardware. Not all GPUs are capable of the same instructions, and each new generation of GPU is usually capable of more instructions than the previous one.

The following is a small reference of some useful intrinsic instructions available in HLSL. The intrinsics are organized in alphabetical order based on the mnemonic or method name. There are around 140 intrinsics available in the HLSL language in total. Many of the intrinsics not mentioned can save a lot of time. For instance, the “lit” intrinsic calculates the lighting coefficient with a single, extremely fast instruction. There are hyperbolic trigonometric functions, as well as arcsine, arccosine, and arctangent. There are also functions that calculate both sine and the cosine at once. In short, once you are familiar with some of these fundamental intrinsics, you might like to look at the complete list and see some of the more advanced capabilities of the GPU. The entire list of intrinsics can be found at Microsoft’s MSDN.

http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376%28v=vs.85%29.aspx

Short HLSL Intrinsic Reference

Absolute

Mnemonic: ret abs(x) **Shader Model**: 1.1

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: Returns the absolute value of X. The absolute value is the positive version of the scalar or scalars contains in x. If you use a vector or matrix, the absolute values of each element will be calculated.

Ceiling

Mnemonic: ret ceil(x) **Shader Model**: 1.1

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: This function returns the smallest integer greater than or equal to the value or values of x. Ceiling rounds numbers up to the nearest integer. If x is a vector or matrix, each element will be rounded up to the nearest integer.

Clamp

Mnemonic: ret clamp(x, min, max) **Shader Model**: 1.1

Parameters: x, min and max can be scalar, vector, or matrix, but they must be the same type; the calculated ret value will have the same type as the inputs.

Description: This function clamps the value or values of the elements in x to between the corresponding values specified by the min and max parameters. Any element in x that is less than the corresponding element in min will be set to min. Any elements in x that are greater than the corresponding element in max will be set the value in max.

Cosine

Mnemonic: ret cos(x) **Shader Model**: 1.1

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function cosine in radians of the value or values in x.

Cross Product

Mnemonic: ret cross(x, y) **Shader Model**: 1.1

Parameters: x and y must be 3-D float vectors; ret is also a 3-D float vector.

Description: This function calculates and returns the cross product of two 3-D vectors. The return value is a vector that is perpendicular to both the inputs; it returns the normal to a plane that contains the input vectors.

Radians to Degrees

Mnemonic: ret degrees(x) **Shader Model**: 1.1

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: This function converts the angles in x, which are read as radians to degrees. If x is a vector or matrix, all elements in x have the conversion performed. This is the same as multiplying the values in x by 180/Pi.

Distance

Mnemonic: ret distance(x, y) **Shader Model**: 1.1

Parameters: x and y are vectors of any size; ret is a scalar float.

Description: This function calculates the distance between the two points x and y. x and y must have the same number of elements.

Dot Product

Mnemonic: ret dot(x, y) **Shader Model**: 1

Parameters: x and y are vectors of any size; ret is a scalar float.

Description: This function calculates the dot product between the two vectors x and y. This is the sum of the products of each of the corresponding elements in x and y. x can be any vector size, but y must match it.

Floor

Mnemonic: ret floor(x) **Shader Model**: 1.1

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: This function returns the largest integer less than or equal to the value or values of x. Floor rounds numbers down to the nearest integer. If x is a vector or matrix, each element will be rounded down to the nearest integer.

Length

Mnemonic: ret length(x) **Shader Model**: 1.1

Parameters: x is a float vector; ret will be a scalar float.

Description: This function calculates the length or magnitude of the vector x.

Maximum

Mnemonic: ret max(x, y) **Shader Model**: 1.1

Parameters: x, y, and ret must all be the same type; they can be scalar, vector, or matrix.

Description: This function selects the maximums or larger of each of the two corresponding elements in x and y and returns them.

Minimum

Mnemonic: ret min(x, y) **Shader Model**: 1.1

Parameters: x, y, and ret must all be the same type; they can be scalar, vector, or matrix.

Description: This function selects the minimums or smaller of each of the two corresponding elements in x and y and returns them.

Multiply

Mnemonic: ret mul(x, y) **Shader Model**: 1.0

Parameters: There are many overloaded versions of mul for different input types.

Description: This function multiplies matrices, scalars, or vectors. x and y need not be the same data type. Depending on their data types, different operations are performed by the function. For instance, if x is a scalar and y is matrix, then each component in y will be multiplied by the x, thus the matrix will be scaled.

This function is used to perform a standard matrix multiplication. If both inputs are matrices, then the resulting output will be the standard matrix product of the two inputs. Note that the multiplication operator (*) with two matrices as operands will perform an element by element multiplication; it does not calculate the matrix product.

Normalize

Mnemonic: ret normalize(x) **Shader Model**: 1.1

Parameters: x is a vector; ret will have the same size and type as x.

Description: This function calculates the normalized vector of x. This vector has the same angle as the input vector, but it has a length of exactly 1.0. This is very useful, as many algorithms and other functions expect normalized vectors.

Power

Mnemonic: ret pow(x, y) **Shader Model**: 1.1

Parameters: Returns x to the power of y; x and y can be scalars, vectors, or matrices.

Description: Raises x to the power of y; x, y, and ret must all be the same type and size.

Degrees to Radians

Mnemonic: ret radians(x) **Shader Model**: 1.0

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: This function converts the angles in x, which are read as degrees to radians. If x is a vector or matrix, all elements in x have the conversion performed. This is the same as multiplying the values in x by Pi/180.

Approximate Reciprocal

Mnemonic: ret rcp(x) **Shader Model**: 5.0

Parameters: x can be scalar, vector, or matrix; ret will have the same type as x.

Description: This function calculates an approximation of the reciprocal of the element or elements in x; the reciprocal is 1.0 divided by the elements in the parameter x.

Reflection Vector

Mnemonic: ret reflect(x, y) **Shader Model**: 1.0

Parameters: x and y are float vectors of the same size; x is the vector of incidence and y is the normal of the surface that x is striking.

Description: This function calculates and returns the reflection vector given a ray of incidence (the x value) and a surface normal (the y value). It calculates the result from the following formula: ret = x-2*y*dot(x, y).

Y is a surface normal vector and should be normalized, or have a length of exactly 1.0.

Refraction Vector

Mnemonic: ret refract(x, y, z)**Shader Model**: 1.1

Parameters: x and y are float vectors of the same size; x is the vector of the entering ray, and y is the normal of the surface the ray is entering. z is a scalar that is the refraction index.

Description: This function calculates and returns the refraction vector given the vector of the entering ray, the surface normal that the ray is entering, and the z is the refractive index of the substance which the ray is entering. For instance, water has a refraction index of about 1.333f, air is around 1.000f, and diamond is around 2.419f.

Round

Mnemonic: ret round(x) **Shader Model**: 1.1

Parameters: x can be floating scalar, vector, or matrix; ret will have the same type as x.

Description: This function rounds the value or values in x to the nearest integers and returns the resulting values.

Reciprocal of Square Root

Mnemonic: ret rsqrt(x) **Shader Model**: 1.1

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function calculates the reciprocal of the square root of the value or values in x. That is, 1.0/sqrt(x).

Saturate

Mnemonic: ret saturate(x) **Shader Model**: 1.0

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function saturates and returns the value or values in x. To saturate is to clamp between 0.0 and 1.0. Any values less than 0.0 will become 0.0, and any values greater than 1.0 will become 1.0.

Sine

Mnemonic: ret sin(x) **Shader Model**: 1.1

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function sine in radians of the value or values in x.

Tangent

Mnemonic: ret tan(x) **Shader Model**: 1.1

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function tangent in radians of the value or values in x.

Square Root

Mnemonic: ret sqrt(x) **Shader Model**: 1.1

Parameters: x can be floating point scalar, vector, or matrix; ret will have the same type as x.

Description: This function calculates and returns the square root of the value or values in x.

Truncate

Mnemonic: ret trunc(x) **Shader Model**: 1.0

Parameters: x can be floating scalar, vector, or matrix; ret will have the same type as x.

Description: This function rounds floating point values to integers by truncating or chopping off any digits right of the radix point; in other words, it performs a float to int cast then back to float. It rounds number towards 0.0.

Chapter 8 Lighting

Light is very complicated, like the texture of our model. In 3-D graphics programming, we use techniques to emulate light without actually calculating the countless light rays that bounce around in the real world. In this chapter, we will examine some simple but effective techniques for lighting our models.

The lighting equations in this chapter are based on Chapter 5 of The CG Tutorial, which is available online from the NVidia website:
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html

Normals

Before we look at lighting, we need to look at normals. A normal is a vector that is perpendicular to another vector or perpendicular to a surface. The image in *Figure 8.1* depicts a cube with the normals rendered as arrows. The cube has eight vectors, one of which is not visible in the image (the lower far left vector). These vectors collectively describe the six faces or surfaces of the cube, only three of which are visible. The white arrows are vector normal; there is one for every vector in the cube, and they point in the direction of the corners from the center of the cube. The black arrows are surface normal; they represent the direction that each of the cube's faces is pointing.

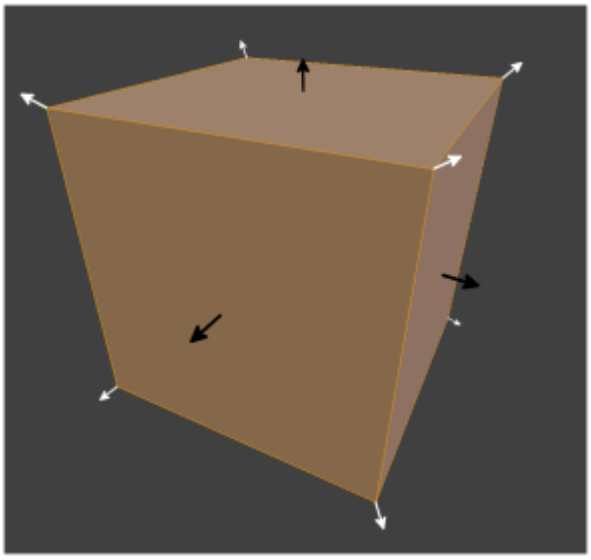

Figure 8.1: Cube with Vector and Surface Normals

Reading Normals

Before we start programming lighting, we should read the normals from the object file into our `Model`. Presently, these are being ignored by the `ModelReader` class. This involves changing the Vertex structure to include the vector normals we read from the file. Open the **Model.h** file and add a `float3` to the vertex structure that will hold the vertex normal. This change is highlighted in the following code table.

```
// Definition of our vertex types
struct Vertex
{
      DirectX::XMFLOAT3 position;
      DirectX::XMFLOAT3 normal;
      DirectX::XMFLOAT2 uv;
};
```

Next, the VertexShader class's D3D11_INPUT_ELEMENT_DESC must be changed to include the new normal. Open the VertexShader.cpp file and add the normal to the description. This change is highlighted in the following code table.

```
// VertexShader.cpp
#include "pch.h"
#include "VertexShader.h"

void VertexShader::LoadFromFile(ID3D11Device *device,
      _In_ Platform::String^ filename)
{
// Read the file
Platform::Array<unsigned char, 1U>^ fileDataVS =
      DX::ReadData(filename);

// Crreate the vertex shader from the file's data
DX::ThrowIfFailed(device->CreateVertexShader(fileDataVS->Data,
      fileDataVS->Length, nullptr, &m_vertexShader));

// Describe the layout of the data
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
      {
      { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
      { "NORMAL",  0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
      { "TEXCOORD",    0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
                  D3D11_INPUT_PER_VERTEX_DATA, 0 },
```

```
	};

DX::ThrowIfFailed(device->CreateInputLayout(vertexDesc,
	ARRAYSIZE(vertexDesc), fileDataVS->Data, fileDataVS->Length,
	&m_inputLayout));
}
```

In the previous code table, I have placed the normals between the `POSITION` and `TEXCOORD`. The 24 in the `TEXCOORD`'s specifications must be changed as well; it was previously 12.

Next we can change the `Model` class to load the normals from file, just as we did for the vertex positions and texture coordinates. Open the **ModelReader.cpp** file and add code to read the normals in the `ReadModel` method. These changes are highlighted in the following code table.

```
Model* ModelReader::ReadModel(ID3D11Device* device, char* filename)
{
// Read the file
int filesize = 0;
char* filedata = ReadFile(filename, filesize);

// Parse the data into vertices and indices
int startPos = 0;
std::string line;

// Vectors for vertex positions
std::vector<float> vertices;
std::vector<int> vertexIndices;
// Vectors for texture coordinates
std::vector<float> textureCoords;
std::vector<int> textureIndices;
// Vectors for normals
std::vector<float> normals;
std::vector<int> normalIndices;

int index; // The index within the line we're reading

while(startPos < filesize) {
	line = ReadLine(filedata, filesize, startPos);
	if(line.data()[0] == 'v' && line.data()[1] == ' ') {
		index = 2;
		// Add to vertex buffer
		vertices.push_back(ReadFloat(line, index)); // Read X
		vertices.push_back(ReadFloat(line, index)); // Read Y
		vertices.push_back(ReadFloat(line, index)); // Read Z
		// If there's a "W" it will be ignored
```

```
            }
      else if(line.data()[0] == 'f' && line.data()[1] == ' ') {
            index = 2;
            // Add triangle to index buffer
            for(int i = 0; i < 3; i++) {
                  // Read position of vertex
                  vertexIndices.push_back(ReadInt(line, index));
                  // Read the texture coordinate
                  textureIndices.push_back(ReadInt(line, index));

                  // Ignore the normals
                  //ReadInt(line, index );
                  // Read the normal indices
                  normalIndices.push_back(ReadInt(line, index));
                  }
            }
      else if(line.data()[0]=='v'&& line.data()[1] == 't' &&
line.data()[2] == ' ')
            {
            index = 3;
            // Add to texture
            textureCoords.push_back(ReadFloat(line, index)); // Read U
            textureCoords.push_back(ReadFloat(line, index)); // Read V
            }
      else if(line.data()[0]=='v' && line.data()[1] == 'n' &&
line.data()[2] == ' ')
            {
            index = 3;
            // Add to normals
            normals.push_back(ReadFloat(line, index)); // Read X
            normals.push_back(ReadFloat(line, index)); // Read Y
            normals.push_back(ReadFloat(line, index)); // Read Z
            }
      }

// Deallocate the file data
delete[] filedata;    // Deallocate the file data

// Subtract one from the vertex indices to change from base 1
// indexing to base 0:
for(int i = 0; i < (int) vertexIndices.size(); i++) {
      vertexIndices[i]--;
      textureIndices[i]--;
      normalIndices[i]--;
```

```
	}

// Create a collection of Vertex structures from the faces
std::vector<Vertex> verts;
int j = vertexIndices.size();
int qq = vertices.size();
for(int i = 0; i < (int) vertexIndices.size(); i++) {
	Vertex v;

	// Create a vertex from the referenced positions
	v.position = XMFLOAT3(
		vertices[vertexIndices[i]*3+0],
		vertices[vertexIndices[i]*3+1],
		vertices[vertexIndices[i]*3+2]);

	// Set the vertex's normals
	v.normal = XMFLOAT3(
		normals[normalIndices[i]*3+0],
		normals[normalIndices[i]*3+1],
		normals[normalIndices[i]*3+2]);

	// Set the vertex's texture coordinates
	v.uv = XMFLOAT2(
		textureCoords[textureIndices[i]*2+0],
		1.0f-textureCoords[textureIndices[i]*2+1] // Negate V
		);

	verts.push_back(v); // Push to the verts vector
	}

// Create a an array from the verts vector.
// While we're running through the array reverse
// the winding order of the vertices.
Vertex* vertexArray = new Vertex[verts.size()];
for(int i = 0; i < (int) verts.size(); i+=3) {
	vertexArray[i] = verts[i+1];
	vertexArray[i+1] = verts[i];
	vertexArray[i+2] = verts[i+2];
	}

// Construct the model
Model* model = new Model(device, vertexArray, verts.size());

// Clear the vectors
```

```
vertices.clear();
vertexIndices.clear();
verts.clear();
textureCoords.clear();
textureIndices.clear();
normalIndices.clear();
normals.clear();

// Delete the array/s
delete[] vertexArray;

return model; // Return the model
}
```

Now we can change the vertex specification in both the HLSL files for our shaders. Open the **VertexShader.hlsl** file and add normals to both the `VertexShaderInput` and the `VertexShaderOutput` structures. I have passed the normals unchanged to the pixel shader in the code presented in the following code table.

```
// VertexShader.hlsl

// The GPU version of the constant buffer
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
	matrix model;
	matrix view;
	matrix projection;
};

// The input vertices
struct VertexShaderInput
{
	float3 position : POSITION0;
	float3 normal : NORMAL0;
	float2 tex : TEXCOORD0;
};

// The output vertices as the pixel shader will get them
struct VertexShaderOutput
{
	float4 position : SV_POSITION0;
	float3 normal : NORMAL0;
	float2 tex : TEXCOORD0;
```

```
};

// This is the main entry point to the shader:
VertexShaderOutput main(VertexShaderInput input)
{
      VertexShaderOutput output;
      float4 pos = float4(input.position, 1.0f);

      // Use constant buffer matrices to position the vertices:
      pos = mul(pos, model);      // Position the model in the world
      pos = mul(pos, view); // Position the world with respect to a
camera
      pos = mul(pos, projection);// Project the vertices
      output.position = pos;

      // Pass the texture coordinates unchanged to pixel shader
      output.tex = input.tex;

      // Pass the normals unchanged to the pixel shader
      output.normal = input.normal;

      return output;
}
```

Change the structure in the PixelShader.hlsl file as well. The altered code for the pixel shader is presented as the following code table.

```
// PixelShader.hlsl

Texture2D shaderTexture; // This is the texture
SamplerState samplerState;

// Input is exactly the same as
// vertex shader output!
struct PixelShaderInput
{
      float4 pos : SV_POSITION0;
      float3 normal : NORMAL0;
      float2 tex : TEXCOORD0;
};


// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
```

```
{
      float4 textureColor =
            shaderTexture.Sample(samplerState, input.tex);

      // Return the color unchanged
      return textureColor;
}
```

At this point, you should be able to run the application and it will appear the same as before, only now we have the vertex normals being passed to the pixel shader.

Emissive Lighting

We will add several simple lighting techniques together; this will give us some flexibility. The first type of lighting we will implement will seem like a big step backwards, since it will remove the texture from our model. Emissive lighting is the glow that an object has itself. In this simple model, our emissive lighting will not light other objects in the scene. Emissive lighting needs only a color. Open the **PixelShader.hlsl** file and alter the main method. The following code table highlights these changes.

```
// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
      //float4 textureColor =
      //      shaderTexture.Sample(samplerState, input.tex);

      // Return the color unchanged
      // return textureColor;

      float4 emissive = float4(0.2f, 0.2f, 0.2f, 1.0f);

      float4 finalColor = emissive;

      finalColor = normalize(finalColor);

      return finalColor;
}
```

In the previous code table, we have said that all pixels in the model are dark grey, `(0.2f, 0.2f, 0.2f, 1.0f)`. This means that even without any lighting at all, our model is glowing a dim grey color. The line I have highlighted beginning with `float4 finalColor` is fairly pointless at present, but we will add more lights to it as we create them. Also notice the call to `normalize`, the RGBA values in our color should each be from 0.0f, to 1.0f.

Ambient Lighting

The next type of lighting we will add is called ambient lighting. When objects are not directly lit by a light source in the real world, they are often still visible because of light rays are bouncing off other objects. For instance, if you look under a desk in a well lit room, you will clearly see the underside despite the fact that the light source is not apparently shining under the desk. In reality, ambient lighting is extremely complicated. When programming we can summarize the effect by giving our model an ambient reflection, which is the amount of ambient light the material of our model reflects, and by giving the ambient light in our scene a color. The following code table highlights these changes, and I have also removed the lines commented out in the previous code table.

```
// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
	float4 emissive = float4(0.2f, 0.2f, 0.2f, 1.0f);

	float materialReflection = 1.0f;
	float4 ambientLightColor = float4(0.1f, 0.1f, 0.1f, 1.0f);
	float4 ambient = ambientLightColor * materialReflection;

	float4 finalColor = emissive + ambient;

	finalColor = normalize(finalColor);

	return finalColor;
}
```

In the previous code table, I have set the `materialReflection` to 100% and the `ambientLightColor` to a dark grey (`0.1f, 0.1f, 0.1f, 1.0f`). The two values (`materialReflection` and `ambientLightColor`) are multiplied together to calculate the `ambient` component in our light equation, which is then added to the `emissive` component to form the final color.

Diffuse Lighting

Diffuse lighting is the light that hits our material and bounces off equally in all directions. It requires a light source with a position and color, a material color, and the normal of the surface. It does not account for the highlight, reflection, or shine of a material, only the general color of it. Diffuse lighting is useful for matte materials which do not have a glossy finish: unpolished wood, carpet, and plaster, for instance.

The values we read from our texture file are not necessarily exactly the same colors as the material when it is lit, since shading must be applied to indicate the lighting. If green light shines on our model, it will look more green; likewise, a red light will make the material look more red. The point is that the values in our texture are the colors that should appear when white light is shining directly at the material.

The following code table highlights the changes to the pixel shader's main method to read the texture, as well as add our lighting effects.

```
// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
	float4 emissive =	float4(0.1f, 0.1f, 0.1f, 1.0f);

	float materialReflection = 1.0f;
	float4 ambientLightColor = float4(0.1f, 0.1f, 0.1f, 1.0f);
	float4 ambient = ambientLightColor * materialReflection;

	float diffuseIntensity = 1.0f;
	float4 diffuseLightColor = float4(1.0f, 1.0f, 1.0f, 1.0f);
	float4 diffuseLightDirection = float4(1.0f, -1.0f, 1.0f, 1.0f);
	float4 materialColor = shaderTexture.Sample(samplerState,
input.tex);
	float4 diffuse = diffuseIntensity * diffuseLightColor *
		saturate(dot(-diffuseLightDirection, input.normal));
	diffuse = diffuse * materialColor;

	float4 finalColor = emissive + ambient + diffuse;

	finalColor = normalize(finalColor);

	return finalColor;
}
```

When running the application, you should see the spaceship again, only now it has a simple diffuse lighting shading effect (*Figure 8.2*).

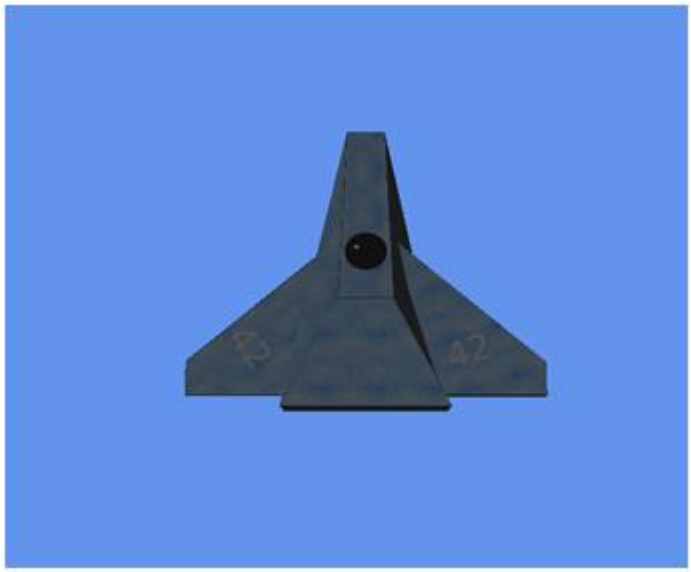

Figure 21: Figure 8.2: Spaceship with Diffuse Lighting

Note: I have hard coded the variables for the lights. It is much more common to place these into a `cbuffer` so the CPU can update and change the values. It is common to group items together into different `cbuffers` based on how often they are accessed. For instance, you might have `perobject`, `perframe`, `pergame` `cbuffers` which hold data that must be updated per object in the game, per frame, or just once respectively. I have used a single cbuffer because what we are doing is not processor intensive.

The basis of the diffuse lighting equation presented above is fairly straightforward; if the normal of the face is facing the light, the surface is depicted as being the color from the texture. If the normal of the face is not facing the diffuse light source, then the surface is rendered as the emissive and ambient colors.

The equation and code for the above diffuse lighting was adapted from the RasterTek Tutorial 6, available online from http://www.rastertek.com/dx11tut06.html.

Chapter 9 User Input

Processing user input is important for games. In the past there was a component included with DirectX called DirectInput, but this is no longer recommended. As a rule of thumb, try to create user inputs with well established design methodologies. Use Windows controls where possible or controls that have the same functionality. Users are generally familiar with the way Windows works. Its user interface was not developed by a single person in a short amount of time. They were designed and developed by possibly thousands of people over the years, and the current state of operating system (OS) controls is very reliable, flexible, and user friendly.

When designing an interface, we often want a gamer to be able to play the game with absolutely no instruction. They should simply know how to play just by looking at the game. Sometimes this is not possible or desirable, but usually this is what a user interface should aim for.

Control Types

There are several different methods users commonly use to interact with computers. Keyboards, computer mice, touch screens, and game controllers are the most common. In this chapter, we will examine processing all of these except for the game controller.

From our point of view, the mouse and the touchscreen will be the same input method. There are many differences between the two input methods. The mouse has several buttons and the touchscreen has other metrics such as how hard the pointer is being depressed. I will refer to both of these methods as the pointer.

It is important to know that your games should fully support both a keyboard and mouse or touch screen input methods. If applications do not fully support both of these user controls, Microsoft may not allow the application to be uploaded to the Windows Store. The mouse control and touchscreen can be handled in almost the same way, and throughout this chapter we will refer to both simply as the pointer.

Keyboard

Normally in video games we are interested in knowing if a key on the keyboard is down or up. This is different from a word processing application where the keystroke is important. When a key is depressed, a key press event is raised. The key press is repeated over and over as long as the key remains pressed. This means that if we have a word processor open and we hold the “a” key down, the first “a” will appear immediately, then after a short period many more will appear in rapid succession. The repeat delay is settable from the Windows control panel. This is not a desirable method for controlling video games; the delay between the initial key press event and the repeated ones makes reading key press events impractical for controlling video games.

Instead of key press events, we read key down events and key up events. The simplest method for recording which keys are down is to create an array of boolean values, one for each key on the keyboard. This array is the key state array, or the state of each of the keys on the keyboard. Initially, every element in the array is set to false, meaning all the keys are up. When the user depresses a key, the corresponding array element is set to true. When the user releases a key, the corresponding array element is set to false. We can read the key state of any particular key from our update method and respond to the keys the user has depressed.

The first thing to do is add a class with the key state array. Add two files, **Keyboard.h** and **Keyboard.cpp**. The code for these new files is presented in the two following code tables.

```
// Keyboard.h
#pragma once
// This class records the keystates
class Keyboard
{
private:
	bool m_keystates[256];

public:
	// Public constructor
	Keyboard();

	// Set all keys to up
	void Reset();

	// Register keydown event
	void KeyDown(Windows::System::VirtualKey key);

	// Register keyup event
	void KeyUp(Windows::System::VirtualKey key);

	// Query a keystate
	bool IsKeyDown(Windows::System::VirtualKey key);
};
```

```
// Keyboard.cpp
#include "pch.h"
#include "Keyboard.h"

Keyboard::Keyboard(){
Reset();
}
```

```
void Keyboard::Reset(){
for(int i = 0; i < 256; i++) m_keystates[i] = false;
}

void Keyboard::KeyDown(Windows::System::VirtualKey key){
m_keystates[(unsigned char) key] = true;
}

void Keyboard::KeyUp(Windows::System::VirtualKey key){
m_keystates[(unsigned char) key] = false;
}

bool Keyboard::IsKeyDown(Windows::System::VirtualKey key){
return m_keystates[(unsigned char) key];
}
```

The class is fairly simple; it consists of a `bool` array called `m_keystates` and several getters and setters for when the keys are pressed and released.

To add a reference to the **Keyboard.h** file to the **SimpleTextRenderer.h** file, see the following code table.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "Model.h"
#include "VertexShader.h"
#include "PixelShader.h"
#include "ModelReader.h"
#include "Texture.h"
#include "Keyboard.h"

// This class renders simple text with a colored background.
```

Add the definitions for `KeyUp` and `KeyDown` member methods to the `SimpleTextRenderer`, and a keyboard member variable called `m_keyboard` to the class as well. These changes are highlighted in the following code table. When a key is pressed or released on the keyboard, we will route it to the `KeyUp` and `KeyDown` methods so the `SimpleTextRenderer` can respond to the user input.

```
        // Method for updating time-dependent objects.
        void Update(float timeTotal, float timeDelta);
```

```
        // Keyboard methods:
        void KeyDown(Windows::System::VirtualKey key);
        void KeyUp(Windows::System::VirtualKey key);


private:
        Model *m_model;
        Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
        ModelViewProjectionConstantBuffer m_constantBufferCPU;
        Texture m_texture;
        ID3D11SamplerState *m_samplerState;

        // Shaders
        VertexShader m_vertexShader;
        PixelShader m_pixelShader;

        // Keyboard member variable
        Keyboard m_keyboard;
};
```

Next, we need to record when a key is pressed or released. This can be done by attaching events to the `DirectXPage` class. Open the **DirectXPage.xaml.h** file and add method prototypes to capture this event. These changes are highlighted in the following code table.

```
void OnOrientationChanged(Platform::Object^ sender);
void OnDisplayContentsInvalidated(Platform::Object^ sender);
void OnRendering(Object^ sender, Object^ args);

// Keyboard events
void OnKeyDown(Object^ sender,
        Windows::UI::Xaml::Input::KeyRoutedEventArgs^ e);
void OnKeyUp(Object^ sender,
        Windows::UI::Xaml::Input::KeyRoutedEventArgs^ e);

Windows::Foundation::EventRegistrationToken m_eventToken;
```

Add the body of these events to the **DirectXPage.xaml.cpp** file. These events route the key presses to the `m_renderer` member variable we just created. These changes are presented in the following code table. I have added these event handlers to the bottom of the **DirectXPage.xaml.cpp** file.

```
void DirectXPage::OnRendering(Object^ sender, Object^ args)
{
        m_timer->Update();
```

```
	m_renderer->Update(m_timer->Total, m_timer->Delta);
	m_renderer->Render();
	m_renderer->Present();
}

void DirectXPage::OnKeyDown(Object^ sender,
	Windows::UI::Xaml::Input::KeyRoutedEventArgs^ e) {
m_renderer->KeyDown(e->Key);
}

void DirectXPage::OnKeyUp(Object^ sender,
	Windows::UI::Xaml::Input::KeyRoutedEventArgs^ e) {
m_renderer->KeyUp(e->Key );
}
```

Attach the events in the **DirectXPage.xaml.cpp** file in the constructor. The following code table illustrates these changes.

```
	m_eventToken = CompositionTarget::Rendering::add(ref new
EventHandler<Object^>(this, &DirectXPage::OnRendering));

	m_timer = ref new BasicTimer();

	// Attach the keyboard events
	this->KeyUp += ref new KeyEventHandler(this,
		&DirectXPage::OnKeyUp);
	this->KeyDown += ref new KeyEventHandler(this,
		&DirectXPage::OnKeyDown);
}
```

The body of the methods (`KeyDown` and `KeyUp`) in the SimpleTextRenderer.cpp file is very simple, it just routes the keys to the `m_keyboard` member variable. The code for these functions is highlighted in the following code table. I have placed these functions at the end of the SimpleTextRenderer.cpp file, after the `Render` method.

```
	HRESULT hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET) {
		DX::ThrowIfFailed(hr);
	}
}

void SimpleTextRenderer::KeyDown(Windows::System::VirtualKey key) {
m_keyboard.KeyDown(key);
```

```
}

void SimpleTextRenderer::KeyUp(Windows::System::VirtualKey key) {
m_keyboard.KeyUp(key);
}
```

You should be able to run the application at this point. The keys will not do anything yet, but the application is reading and routing all of the key up and down events through the `m_keyboard` variable.

Mouse Touchscreen Pointer

Many mobile devices have a touchscreen and desktop computers have a mouse. From our point of view, these input methods will be the same. We can program touchscreen or mouse controls using the standard Windows XAML controls. In this section, we will add a directional pad (dpad) for up, down, left, and right. We will move the spaceship in the next chapter. For now, we will just capture the controls.

Double-click on the **DirectXPage.XAML** and add four `Ellipse` controls to the XAML page (see *Figure 9.1*). These will be our dpad; it does not matter where you add the ellipses or how big they are, since we will change these settings in the XAML code view.

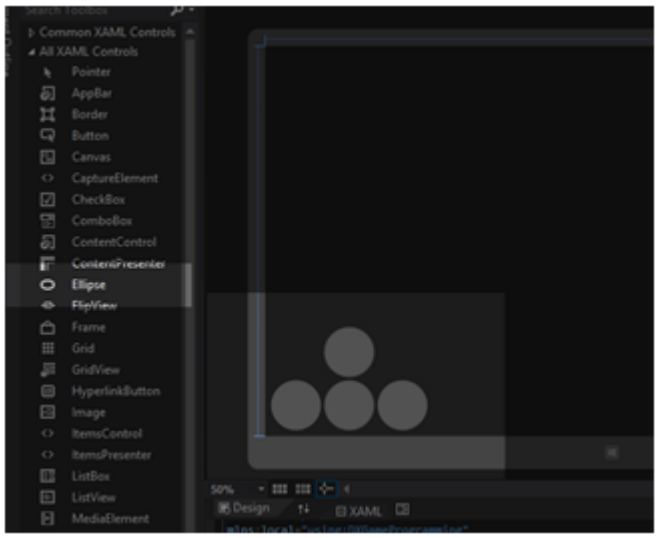

Figure 22: Figure 9.1: Adding Ellipses for a DPad

Note: I have used the Ellipse control rather than XAML buttons for our DPad. XAML buttons have a lot of code already written in the background. For instance, they have functionality to handle the pointer being pressed and released. These events are handled automatically and they will not be passed on to our custom event handlers. It is easy to capture pointer presses and releases with the Ellipse controls.

Now that we have added the ellipses, the settings can be changed in the XAML code view. These changes are highlighted in the following code table.

```
    <SwapChainBackgroundPanel x:Name="SwapChainPanel"
PointerMoved="OnPointerMoved" PointerReleased="OnPointerReleased">
        <Ellipse Fill="#FFF4F4F5" HorizontalAlignment="Left"
Height="100" Margin="115,0,0,10" Stroke="Black"
VerticalAlignment="Bottom" Width="100" Opacity="0.25"
PointerPressed="OnDownButtonDown"
PointerEntered="OnDownButtonPointerEntered"
PointerReleased="OnDownButtonUp" PointerExited="OnDownButtonUp"/>
        <Ellipse Fill="#FFF4F4F5" HorizontalAlignment="Left"
Height="100" Margin="220,0,0,10" Stroke="Black"
VerticalAlignment="Bottom" Width="100" Opacity="0.25"
PointerPressed="OnRightButtonDown"
PointerEntered="OnRightButtonPointerEntered"
PointerReleased="OnRightButtonUp" PointerExited="OnRightButtonUp"/>
        <Ellipse Fill="#FFF4F4F5" HorizontalAlignment="Left"
Height="100" Margin="115,0,0,115" Stroke="Black"
VerticalAlignment="Bottom" Width="100" Opacity="0.25"
PointerPressed="OnUpButtonDown"
PointerEntered="OnUpButtonPointerEntered"
PointerReleased="OnUpButtonUp" PointerExited="OnUpButtonUp"/>
        <Ellipse Fill="#FFF4F4F5" HorizontalAlignment="Left"
Height="100" Margin="10,0,0,10" Stroke="Black"
VerticalAlignment="Bottom" Width="100" Opacity="0.25"
PointerPressed="OnLeftButtonDown"
PointerEntered="OnLeftButtonPointerEntered"
PointerReleased="OnLeftButtonUp" PointerExited="OnLeftButtonUp"/>
    </SwapChainBackgroundPanel>
```

This XAML code specifies that the ellipses are to be anchored to the lower left corner of the screen, which is the most common position for a dpad on touchscreens. They are circular with a diameter of 100 pixels. They are almost transparent; this is so they do not obscure the playfield.

The events I have captured are `PointerPressed` and `PointerReleased`. If you are programming for mouse, the other two events I have captured, `PointerEntered` and `PointerExited`, do not make much sense. I have captured these events so that if the user holds down the pointer with a touchscreen and slides their finger to another ellipse, the `PointerPressed` event will not fire. This is no good for touchscreen. The other side of this coin is, if the mouse cursor is not pressed but enters or exits the ellipses, the `PointerEntered` and `PointerExited` events should not fire. I have captured the events and distinguished between the pointer types in the event handlers. I have routed the `PointerExited` event through the same handler as `PointerReleased`, as they do exactly the same thing. To create the event handlers, right-click the method's name in the XAML code and select **Navigate to Event Handler** from the context menu. Visual Studio will write the event handler and take us to it in the code (see *Figure 8.2*).

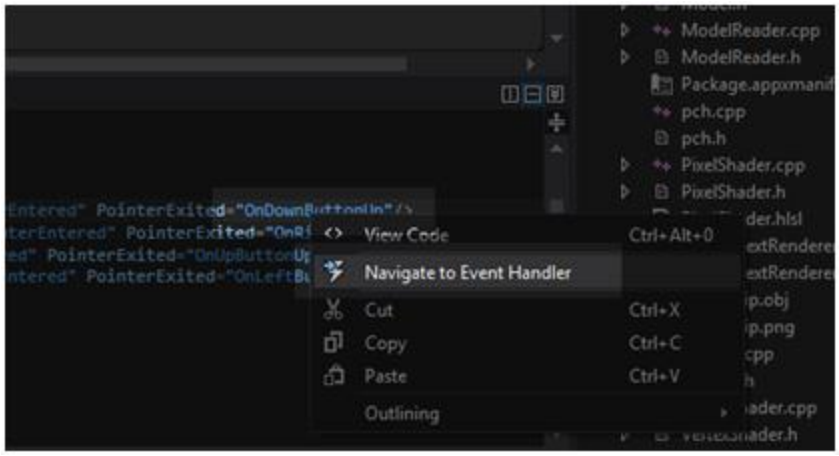

Figure 23: Figure 8.2: Adding Event Handlers

Next, we will fill in the bodies for the event handlers. The changes to the DirectXPage.xaml.cpp file are highlighted in the following code table.

```
void DirectXPage::OnDownButtonDown(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyDown(Windows::System::VirtualKey::Down);
      }

void DirectXPage::OnRightButtonDown(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyDown(Windows::System::VirtualKey::Right);
      }

void DirectXPage::OnUpButtonDown(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyDown(Windows::System::VirtualKey::Up);
      }
```

```
void DirectXPage::OnLeftButtonDown(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyDown(Windows::System::VirtualKey::Left);
      }

void DirectXPage::OnDownButtonUp(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyUp(Windows::System::VirtualKey::Down);
      }

void DirectXPage::OnRightButtonUp(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyUp(Windows::System::VirtualKey::Right);
      }

void DirectXPage::OnUpButtonUp(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyUp(Windows::System::VirtualKey::Up);
      }

void DirectXPage::OnLeftButtonUp(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      m_renderer->KeyUp(Windows::System::VirtualKey::Left);
      }
void DirectXPage::OnDownButtonPointerEntered(Platform::Object^ sender,
      Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      if(e->Pointer->PointerDeviceType ==
Windows::Devices::Input::PointerDeviceType::Touch)
            m_renderer->KeyDown(Windows::System::VirtualKey::Down);
      }

void DirectXPage::OnRightButtonPointerEntered(Platform::Object^
sender,
      Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      if(e->Pointer->PointerDeviceType ==
Windows::Devices::Input::PointerDeviceType::Touch)
            m_renderer->KeyDown(Windows::System::VirtualKey::Right);
      }

void DirectXPage::OnUpButtonPointerEntered(Platform::Object^ sender,
      Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      if(e->Pointer->PointerDeviceType ==
Windows::Devices::Input::PointerDeviceType::Touch)
```

```
            m_renderer->KeyDown(Windows::System::VirtualKey::Up);
        }

void DirectXPage::OnLeftButtonPointerEntered(Platform::Object^ sender,
        Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
        if(e->Pointer->PointerDeviceType ==
Windows::Devices::Input::PointerDeviceType::Touch)
            m_renderer->KeyDown(Windows::System::VirtualKey::Left);
        }
```

The above methods send `KeyUp` and `KeyDown` signals to our `Keyboard` class. Note the test for the `PointerDeviceType`; this distinguishes between mouse and touchscreen. It is actually very cumbersome to control an on screen dpad with the mouse, but being able to distinguish the input type may be useful, so I have kept it in the code.

At this point, you should be able to run the application. Once again, it will look exactly as before, only now we can cause `KeyUp` and `KeyDown` events with a keyboard, mouse, or a touch screen.

Chapter 10 Putting it all Together

In the final chapter, we will look at putting many of the concepts we have seen in both this book and the previous book (*Direct2D Succinctly*). At the moment we are seemingly very far away from having anything even remotely resembling a game, but actually we are quite close to having an engine that can create many simple 3-D games. The aim of this book and the previous one has been to introduce general techniques for graphics programming. I have, so far, ignored almost all aspects that would lead us down a path of programming some specific game genre. At this point, I think it would be best to illustrate a game and show that we have examined enough concepts to create 3-D games. We will make something that resembles a space shooter like Space Invaders.

In the following code tables, I've neglected to go into any details because it is all either basic C++ programming or DirectX code, which we have been through in this book and the previous book.

Baddies and Bullets

The first thing we need to do is load some more models, one for a baddie and one for a bullet. These are extremely simple models only to save space in the text of this book. The following code table is the object file for the baddie.obj.

```
# Blender v2.66 (sub 0) OBJ File: 'baddie.blend'
# www.blender.org
o Cube
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -0.231813 1.000000 0.231813
v -0.231813 1.000000 -0.231813
v 0.231813 1.000000 -0.231813
v 0.231813 1.000000 0.231813
vt 0.976658 0.674214
vt 0.651396 0.674214
vt 0.651396 0.348953
vt 0.075574 0.424353
vt 0.000175 0.424353
vt 0.000175 0.348953
vt 0.325436 0.000175
vt 0.200505 0.348604
vt 0.000175 0.000175
```

```
vt 0.651047 0.000175
vt 0.526116 0.348604
vt 0.450716 0.348604
vt 0.325786 0.697382
vt 0.450716 0.348953
vt 0.526116 0.348953
vt 0.999825 0.200505
vt 0.651396 0.325436
vt 0.999825 0.125106
vt 0.976658 0.348953
vt 0.075574 0.348953
vt 0.125106 0.348604
vt 0.325786 0.000175
vt 0.651047 0.697382
vt 0.651396 0.000175
vn -0.000000 -1.000000 0.000000
vn 0.000000 1.000000 0.000000
vn -0.933509 0.358555 0.000000
vn 0.000000 0.358554 -0.933509
vn 0.933509 0.358555 0.000000
vn -0.000000 0.358555 0.933509
vn 0.000000 0.358555 -0.933509
s off
f 1/1/1 2/2/1 3/3/1
f 5/4/2 8/5/2 7/6/2
f 1/7/3 5/8/3 2/9/3
f 2/10/4 6/11/4 7/12/4
f 3/13/5 7/14/5 8/15/5
f 5/16/6 1/17/6 8/18/6
f 4/19/1 1/1/1 3/3/1
f 6/20/2 5/4/2 7/6/2
f 5/8/3 6/21/3 2/9/3
f 3/22/7 2/10/7 7/12/7
f 4/23/5 3/13/5 8/15/5
f 1/17/6 4/24/6 8/18/6
```

The texture to this object, called baddie.png, is the following UV map (*Figure 10.1*).

Figure 24: Figure 10.1: baddie.png

The object file and texture should be saved as baddie.obj and baddie.png, and imported into the Visual Studio project the same way we added the spacehip.obj and spaceship.png files. We are about to change the way these models are loaded by the application, but the import of the files is the same. Don't forget to change the file type of the obj file from object to text.

The bullet model object file is presented as the following code table.

```
# Blender v2.67 (sub 0) OBJ File: ''
# www.blender.org
v 0.086050 -0.086050 -0.086050
v 0.086050 -0.086050 0.086050
v -0.086050 -0.086050 0.086050
v -0.086050 -0.086050 -0.086050
v 0.086050 0.086050 -0.086050
v 0.086050 0.086050 0.086050
v -0.086050 0.086050 0.086050
v -0.086050 0.086050 -0.086050
vt 0.666467 0.666467
vt 0.333533 0.666467
vt 0.333533 0.333533
vt 0.333134 0.333133
vt 0.000200 0.333134
vt 0.000200 0.000200
vt 0.666467 0.000200
vt 0.666467 0.333133
vt 0.333533 0.333134
vt 0.999800 0.333533
vt 0.999800 0.666467
vt 0.666867 0.666467
vt 0.000200 0.999800
vt 0.000200 0.666867
vt 0.333134 0.999800
```

```
vt 0.333134 0.333533
vt 0.333134 0.666467
vt 0.000200 0.666467
vt 0.666467 0.333533
vt 0.333133 0.000200
vt 0.333533 0.000200
vt 0.666867 0.333533
vt 0.333134 0.666867
vt 0.000200 0.333533
vn 0.000000 -1.000000 0.000000
vn -0.000000 1.000000 0.000000
vn 1.000000 -0.000000 0.000001
vn -0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
vn 1.000000 0.000000 -0.000000
s off
f 1/1/1 2/2/1 3/3/1
f 5/4/2 8/5/2 7/6/2
f 1/7/3 5/8/3 6/9/3
f 2/10/4 6/11/4 7/12/4
f 3/13/5 7/14/5 4/15/5
f 5/16/6 1/17/6 4/18/6
f 4/19/1 1/1/1 3/3/1
f 6/20/2 5/4/2 7/6/2
f 2/21/7 1/7/7 6/9/7
f 3/22/4 2/10/4 7/12/4
f 7/14/5 8/23/5 4/15/5
f 8/24/6 5/16/6 4/18/6
```

The bullet's UV layout is the following image (*Figure 10.2*).

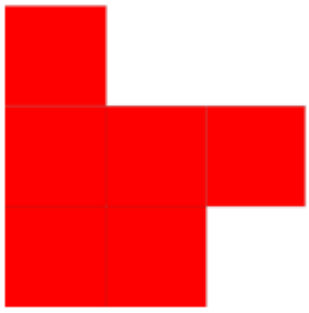

Figure 25: Figure 10.2: bullet.png

These two items can be saved and imported into the project in the same way as the spaceship and baddie. I called these two files bullet.obj and bullet.png; if you name them something different, be sure to change the filename references in the code.

GameObject Class

The `GameObject` class is a simple wrapper class. It holds the model and position of the objects (the player, the bullet, and the baddies) in our game. It is not required for a game engine, but it really helps to create a `GameObject` class or something similar to encapsulate all the complexities of the physical objects in our virtual world. The following class is specifically designed for exactly what the game will need, but usually the `GameObject` class should be flexible and contain a more uniform and complete set of methods for manipulating objects. Add two files to the project, **GameObject.h** and **GameObject.cpp**. The two following code tables show the code for these files.

```
// GameObject.h
#pragma once

#include "pch.h"
#include "Model.h"
#include "ModelReader.h"
#include "Texture.h"

using namespace DirectX;

class GameObject {
	XMFLOAT3 m_objectPosition; // Object's position
	XMFLOAT3 m_objectMovement; // Object's movement
	Model *m_model; // Object's model
	Texture m_texture; // Object's texture
	float m_objectSize;   // Object's size in units
	bool m_isActive;// Is the object active?

public:
	GameObject();

	// For use when multiple objects use the sme texture and model
	void SetModelAndTexture(Model* model, Texture* texture)
		{
		m_model = model;
		m_texture = *texture;
		}

	void SetObjectSize(float size) { this->m_objectSize = size; }
```

```
	void SetActive( bool active) { m_isActive = active; }

	bool IsActive() { return m_isActive; }

	void LoadModelFileAndTexture(ID3D11Device* device,
		IWICImagingFactory2* wicFactory, char* modelFilename,
		LPCWSTR texturefilename, float objectSize);

	XMMATRIX GetTranslationMatrix() {
		return XMMatrixTranslation(m_objectPosition.x,
			m_objectPosition.y, m_objectPosition.z);
		}

	// Reverse the direction of the object's movement
	void ReverseDirection() {
		m_objectMovement.x = -m_objectMovement.x;
		m_objectMovement.y = -m_objectMovement.y;
		m_objectMovement.z = -m_objectMovement.z;
		}

	void SetPosition(float x, float y, float z) {
		m_objectPosition.x = x;
		m_objectPosition.y = y;
		m_objectPosition.z = z;
		}

	XMFLOAT3 GetPosition() { return m_objectPosition; }

	XMFLOAT3 GetSpeed() { return m_objectMovement; }

	void SetSpeed(float x, float y, float z) {
		m_objectMovement.x = x; m_objectMovement.y = y;
		m_objectMovement.z = z; }

	void SetPosition(XMFLOAT3 pos) { m_objectPosition = pos; }

	void SetYPosition(float y) { m_objectPosition.y = y; }

	void Move();

	void Accelerate(float amountX, float amountY, float amountZ,
float max);
```

```
	void Render(ID3D11DeviceContext1* context);

	bool Overlapping(GameObject *obj);
};
```

```
// GameObject.cpp

#include "pch.h"
#include "GameObject.h"

GameObject::GameObject()
{
m_objectPosition = XMFLOAT3(0.0f, 0.0f, 0.0f);
m_objectMovement = XMFLOAT3(0.0f, 0.0f, 0.0f);
}

void GameObject::LoadModelFileAndTexture(ID3D11Device* device,
IWICImagingFactory2* wicFactory, char* modelFilename, LPCWSTR
texturefilename, float objectSize)
{
// Read the spaceship model
m_model = ModelReader::ReadModel(device, modelFilename);

// Read the texture:
m_texture.ReadTexture(device, wicFactory, texturefilename);

// Record the radius of the the object's bounding sphere
m_objectSize = objectSize;
}

void GameObject::Render(ID3D11DeviceContext1* context)
{
	if(!m_isActive) return;	// If the object's not active, return

	// Set to render triangles
	UINT stride = sizeof(Vertex);	// Reset to the frist vertices
in the buffer
	UINT offset = 0;
	// Set the vertex buffer
	context->IASetVertexBuffers(0, 1, m_model-
>GetAddressOfVertexBuffer(), &stride, &offset);
	// Set the resource view which points to the texture
	context->PSSetShaderResources(0, 1,
```

```
m_texture.GetResourceView().GetAddressOf());
      // Render the vertices
      context->Draw(m_model->GetVertexCount(), 0);
}

void GameObject::Move()
{
m_objectPosition.x += m_objectMovement.x;
m_objectPosition.y += m_objectMovement.y;
m_objectPosition.z += m_objectMovement.z;
}

void GameObject::Accelerate(float amountX, float amountY, float
amountZ, float max)
{
m_objectMovement.x += amountX;
m_objectMovement.y += amountY;
m_objectMovement.z += amountZ;

if(m_objectMovement.x > max) m_objectMovement.x = max;
if(m_objectMovement.y > max) m_objectMovement.y = max;
if(m_objectMovement.z > max) m_objectMovement.z = max;
if(m_objectMovement.x < -max) m_objectMovement.x = -max;
if(m_objectMovement.y < -max) m_objectMovement.y = -max;
if(m_objectMovement.z < -max) m_objectMovement.z = -max;
}

bool GameObject::Overlapping(GameObject *obj)
{
// If either of the two objects are not active, return false
if(!m_isActive) return false;
if(!obj->m_isActive) return false;

// Find distance in each axis
float distX = m_objectPosition.x - obj->m_objectPosition.x;
float distY = m_objectPosition.y - obj->m_objectPosition.y;
float distZ = m_objectPosition.z - obj->m_objectPosition.z;

// Find total distance from axis distances
float dist = sqrt(distX*distX + distY*distY + distZ * distZ);

// The models overlap if theie distance is less or equal
// to either of the object's sizes
return dist <= (m_objectSize + obj->m_objectSize);
```

```
}
```

The previous class holds the position, movement, active or inactive state, texture, and model for each object in the game. The collection of methods I have written for this class is very specific to the way this game will operate. They are methods for moving the objects around, assigning the texture and model, and testing if two objects collide.

Background

We will place our spaceship in the middle of space by rendering a 2-D image of a star field behind the 3-D objects. The following image (*Figure 10.3*) should be saved as **starfieldbackground.png** and imported to your project just as we did with the UV textures earlier. It will form the background of our game.

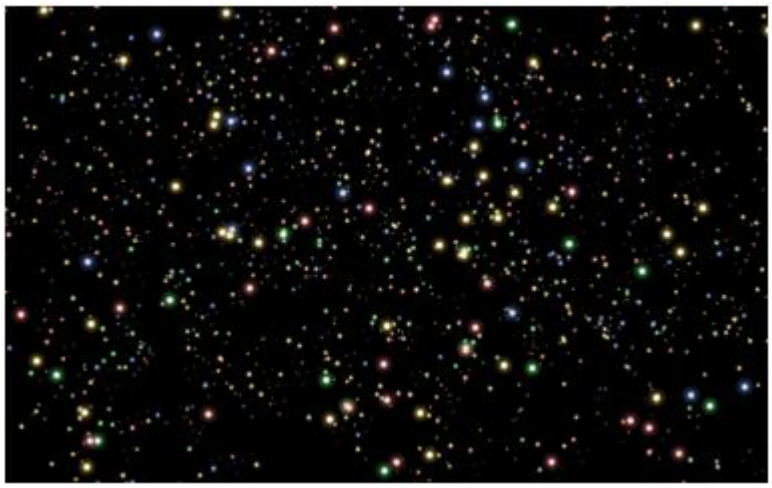

Figure 26: Figure 10.3: starfieldbackground.png

The following class, which loads and renders the background, is taken from the previous book, *Direct2D Succinctly*. It is included here for convenience, but for a description of the class please see the other book. Add two files for the `BitmapBackground` class, **BitmapBackground.h** and **BitmapBackground.cpp**. The two following code tables show the code for these files.

```
// BitmapBackground.h
#pragma once
#include "DirectXBase.h"

// Defines a background consisting of a bitmap image
class BitmapBackground {
private:
	ID2D1Bitmap * m_bmp; // The image to draw
```

```
      D2D1_RECT_F m_screenRectangle; // Destination rectangle

public:
      // Constructor for bitmap backgrounds
      BitmapBackground();

      // Release dynamic memory
      ~BitmapBackground();

      void CreateDeviceDependentResources
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context,
            IWICImagingFactory2 *wicFactory, LPCWSTR filename);
      void CreateWindowSizeDependentResources(
Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);
      void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);
};
```

```
// BitmapBackground.cpp
#include "pch.h"
#include "BitmapBackground.h"

// This constructor must be called at some point after the
// WIC factory is initialized!
BitmapBackground::BitmapBackground() { }

BitmapBackground::~BitmapBackground(){
m_bmp->Release();
}

void BitmapBackground::CreateDeviceDependentResources
      (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context,
IWICImagingFactory2 *wicFactory, LPCWSTR filename) {
// Create a WIC decoder
IWICBitmapDecoder *pDecoder;

// Decode a file, make sure you've added the file to the project
first:
DX::ThrowIfFailed(wicFactory->CreateDecoderFromFilename(filename,
      nullptr, GENERIC_READ, WICDecodeMetadataCacheOnDemand,
&pDecoder));

// Read a frame from the file (png, jpg, bmp etc. images only have one
frame so
// the index is always 0):
```

```
IWICBitmapFrameDecode *pFrame = nullptr;
DX::ThrowIfFailed(pDecoder->GetFrame(0, &pFrame));

// Create format converter to ensure data is the correct format
despite the
// file's format.
// It's likely the format is already perfect but we can't be sure:
IWICFormatConverter *m_pConvertedSourceBitmap;
DX::ThrowIfFailed(wicFactory-
>CreateFormatConverter(&m_pConvertedSourceBitmap));
DX::ThrowIfFailed(m_pConvertedSourceBitmap->Initialize(
      pFrame, GUID_WICPixelFormat32bppPRGBA,
      WICBitmapDitherTypeNone, nullptr,
      0.0f, WICBitmapPaletteTypeCustom));

// Create a Direct2D bitmap from the converted source
DX::ThrowIfFailed(context->CreateBitmapFromWicBitmap(
m_pConvertedSourceBitmap, &m_bmp));

// Release the dx objects we used to create the bmp
pDecoder->Release();
pFrame->Release();
m_pConvertedSourceBitmap->Release();
}

void BitmapBackground::CreateWindowSizeDependentResources(
Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// Save a rectangle the same size as the area to draw the background
m_screenRectangle = D2D1::RectF(0, 0, context->GetSize().width,
context->GetSize().height);
}

void
BitmapBackground::Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext>
context) {
context->DrawBitmap(m_bmp, &m_screenRectangle);
}
```

Pixel Shader

The vertex shader can stay the same, but the lighting in the pixel shader should be a little darker to help the models look more like they are in the dark star field. The following code table is the main method from the pixelshader.hlsl file.

```
// Main entry point to the shader
float4 main(PixelShaderInput input) : SV_TARGET
{
	float4 emissive = float4(0.0f, 0.0f, 0.0f, 1.0f);

	float materialReflection = 0.9f;
	float4 ambientLightColor = float4(0.1f, 0.1f, 0.1f, 1.0f);
	float4 ambient = ambientLightColor * materialReflection;

	float diffuseIntensity = 0.9f;
	float4 diffuseLightColor = float4(1.0f, 1.0f, 1.0f, 1.0f);
	float4 diffuseLightDirection = float4(-1.0f, -1.0f, -1.0f, 1.0f);
	float4 materialColor = shaderTexture.Sample(samplerState,
input.tex);
	float4 diffuse = diffuseIntensity * diffuseLightColor *
		saturate(dot(-diffuseLightDirection, input.normal));
	diffuse = diffuse * materialColor;

	float4 finalColor = emissive + ambient + diffuse;

	finalColor = normalize(finalColor);

	return finalColor;
}
```

SimpleTextRenderer

Most of the changes to the game are in the `SimpleTextRenderer` class. The following code table shows the updated SimpleTextRenderer.h file. It now includes the GameObject.h file, the BitmapBackground.h file, and the string header. The member variables have also changed, since we no longer want a single spaceship model but several different models.

```
// SimpleTextRenderer.h
#pragma once

#include "DirectXBase.h"
#include "VertexShader.h"
#include "PixelShader.h"
#include "Keyboard.h"
#include "GameObject.h"
#include "BitmapBackground.h"
#include "Model.h"
#include "ModelReader.h"
```

```
#include "Texture.h"
#include <string>

// This class renders simple text with a colored background.
ref class SimpleTextRenderer sealed : public DirectXBase
{
public:
	SimpleTextRenderer();

	// DirectXBase methods.
	virtual void CreateDeviceIndependentResources() override;
	virtual void CreateDeviceResources() override;
	virtual void CreateWindowSizeDependentResources() override;
	virtual void Render() override;

	// Method for updating time-dependent objects.
	void Update(float timeTotal, float timeDelta);

	// Keyboard methods:
	void KeyDown(Windows::System::VirtualKey key);
	void KeyUp(Windows::System::VirtualKey key);

private:
	Microsoft::WRL::ComPtr<ID3D11Buffer> m_constantBufferGPU;
	ModelViewProjectionConstantBuffer m_constantBufferCPU;

	ID3D11SamplerState *m_samplerState;

	// Shaders
	VertexShader m_vertexShader;
	PixelShader m_pixelShader;

	Keyboard m_keyboard;

	// Game objects and background
	GameObject m_spaceship;
	GameObject m_baddies[5];
	GameObject m_bullet;
	BitmapBackground  m_bitmapbackground;
	Texture m_baddieTexture;

	// Variables for printing the user's score
	int m_playerScore; // The player's score
	Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;
```

```
	Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_scoreBrush;
};
```

The remaining changes are to be made to the SimpleTextRenderer.cpp file. The `CreateDeviceIndependent` resources method can be used to initialize the `m_textFormat` object to store aspects of the text that will render the player's score. We can also set the player's score to 0 in this method. The following code table shows these changes.

```
void SimpleTextRenderer::CreateDeviceIndependentResources()
{
	DirectXBase::CreateDeviceIndependentResources();

	m_playerScore = 0;

	DX::ThrowIfFailed(
	m_dwriteFactory->CreateTextFormat(L"Segoe UI",
		nullptr, DWRITE_FONT_WEIGHT_NORMAL,
		DWRITE_FONT_STYLE_NORMAL,
		DWRITE_FONT_STRETCH_NORMAL,
		42.0f,L"en-US",
		&m_textFormat));
}
```

The `CreateDeviceResources` method loads all the models and textures into the `GameObject` member variables and sets their initial values. It also creates the brush (`m_scoreBrush`) with which we will render the player's score. The altered code for this method is presented as the following code table.

```
void SimpleTextRenderer::CreateDeviceResources()
{
	DirectXBase::CreateDeviceResources();

	// Load the spaceship model and texture
	m_spaceship.LoadModelFileAndTexture(m_d3dDevice.Get(),
m_wicFactory.Get(), "spaceship.obj", L"spaceship.png", 3.0f);
	m_spaceship.SetPosition(0.0f, 0.0f, 5.0f);
	m_spaceship.SetActive(true);

	// Load the bullet
	m_bullet.LoadModelFileAndTexture(m_d3dDevice.Get(),
m_wicFactory.Get(), "bullet.obj", L"bullet.png", 0.09f);
	m_bullet.SetActive(false);

	// Read the baddie model and texture
```

```
	Model *baddieModel = ModelReader::ReadModel(m_d3dDevice.Get(),
"baddie.obj");
	m_baddieTexture.ReadTexture(m_d3dDevice.Get(),
m_wicFactory.Get(), L"baddie.png");

	for(int i = 0; i < 5; i++)
		{
		m_baddies[i].SetModelAndTexture(baddieModel,
&m_baddieTexture);
		m_baddies[i].SetPosition((float)(i-2.5f)* 3.0f, 0.0f, -
5.0f);
		m_baddies[i].SetObjectSize(1.0f);
		m_baddies[i].SetActive(true);
		}

	// Load the background file
	m_bitmapbackground.CreateDeviceDependentResources(m_d2dContext,
m_wicFactory.Get(), L"starfieldbackground.png");

	// Create the constant buffer on the device
	CD3D11_BUFFER_DESC
constantBufferDesc(sizeof(ModelViewProjectionConstantBuffer),
D3D11_BIND_CONSTANT_BUFFER);
	DX::ThrowIfFailed(m_d3dDevice->CreateBuffer(&constantBufferDesc,
nullptr, &m_constantBufferGPU));

	// Load the vertex and pixel shaders from their files (note the
CSO extension, not hlsl!):
	m_vertexShader.LoadFromFile(m_d3dDevice.Get(),
"VertexShader.cso");
	m_pixelShader.LoadFromFile(m_d3dDevice.Get(), "PixelShader.cso");

	// Create the sampler state
	D3D11_SAMPLER_DESC samplerDesc;
	ZeroMemory(&samplerDesc, sizeof(D3D11_SAMPLER_DESC));
	samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
	samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.MipLODBias = 0.0f;
	samplerDesc.MaxAnisotropy = 1;
	samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
	samplerDesc.BorderColor[0] = 0;
	samplerDesc.BorderColor[1] = 0;
```

```
	samplerDesc.BorderColor[2] = 0;
	samplerDesc.BorderColor[3] = 0;
	samplerDesc.MinLOD = 0;
	samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

	DX::ThrowIfFailed(m_d3dDevice->CreateSamplerState(&samplerDesc,
&m_samplerState));

	// Create the solid brush for the text
	DX::ThrowIfFailed(m_d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Yellow),&m_scoreBrush));
}
```

We should call `CreateWindowSizeDependentResources` for the `m_bitmapbackground` object in the `SimpleTextRenderer::CreateWindowSizeDependentResources` method. The bitmap background class relies on the size of the window, because it must stretch the image across the whole window. The following code table highlights this extra call.

```
void SimpleTextRenderer::CreateWindowSizeDependentResources()
{
	DirectXBase::CreateWindowSizeDependentResources();

	m_bitmapbackground.CreateWindowSizeDependentResources(m_d2dContex
t);

	// Store the projection matrix
	float aspectRatio = m_windowBounds.Width / m_windowBounds.Height;
	float fovAngleY = 70.0f * XM_PI / 180.0f;
	XMStoreFloat4x4(&m_constantBufferCPU.projection,
XMMatrixTranspose(XMMatrixPerspectiveFovRH(fovAngleY,aspectRatio,0.01f
,500.0f)));
}
```

There are many changes to make to the `Update` method of the `SimpleTextRenderer` to allow our models to move and interact with each other. It is this method that responds to the user input and moves all the objects around. It determines if the bullet has hit a baddie and it increases the player's score. I have also moved the camera a little higher. The altered version of the update method is presented as the following code table. Note that I have removed the static positioning of our model from the end of this method.

```
void SimpleTextRenderer::Update(float timeTotal, float timeDelta) {
// View matrix defines where the camera is and what direction it looks
in
XMStoreFloat4x4(&m_constantBufferCPU.view, XMMatrixTranspose(
```

```
      XMMatrixLookAtRH(
            XMVectorSet(0.0f, 10.0f, 0.01f, 0.0f),// Position
            XMVectorSet(0.0f, 0.0f, 0.0f, 0.0f),  // Look at
            XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f)   // Up vector
            )));

// Accelerate player left and right when the keys are down
if(m_keyboard.IsKeyDown(Windows::System::VirtualKey::Left))
      m_spaceship.Accelerate(-2.0f * timeDelta, 0.0f, 0.0f, 0.25f);
else if(m_keyboard.IsKeyDown(Windows::System::VirtualKey::Right))
      m_spaceship.Accelerate(2.0f * timeDelta, 0.0f, 0.0f, 0.25f);

// Check if the ship hit the side of the screen:
if(m_spaceship.GetPosition().x < -10.0f && m_spaceship.GetSpeed().x <
0.0f)
      m_spaceship.ReverseDirection();
if(m_spaceship.GetPosition().x > 10.0f && m_spaceship.GetSpeed().x >
0.0f)
      m_spaceship.ReverseDirection();

// Move the spaceship
m_spaceship.Move();
// Store the ship's model buffer
XMStoreFloat4x4(&m_constantBufferCPU.model,
      XMMatrixTranspose(m_spaceship.GetTranslationMatrix()));

// Check if the player is shooting:
if(   m_keyboard.IsKeyDown(Windows::System::VirtualKey::Up)) {
      // If the bullet's not active, fire it
      if(!m_bullet.IsActive()) {
            m_bullet.SetPosition(m_spaceship.GetPosition());
            m_bullet.SetSpeed(0.0f, 0.0f, -20.0f * timeDelta);
            m_bullet.SetActive(true);
            }
      }

// If the bullet's active, move it
if(m_bullet.IsActive()) {
      m_bullet.SetSpeed(0.0f, 0.0f, -20.0f * timeDelta);
      m_bullet.Move();

      // If the bullet goes out of the screen, deactivate it
      if(m_bullet.GetPosition().z < -10.0f)
            m_bullet.SetActive(false);
```

```
	}

// Move the baddies
for(int i = 0; i < 5; i++) {
	// Make the baddies bob up and down like real aliens
	m_baddies[i].SetYPosition(1.0f * sin(timeTotal*5 + i));

	if(m_baddies[i].GetSpeed().x == 0.0f) // Set the initial speed if
the baddie is not moving
		m_baddies[i].SetSpeed(4.0f*timeDelta, 0.0f, 0.0f);

	// If they hit the side of the screen, reverse their direction
	if(m_baddies[i].GetSpeed().x < 0.0f &&
m_baddies[i].GetPosition().x < -10.0f)
		m_baddies[i].ReverseDirection();
	if(m_baddies[i].GetPosition().x > 10.0f &&
m_baddies[i].GetSpeed().x > 0.0f)
		m_baddies[i].ReverseDirection();

	// Move this baddie
	m_baddies[i].Move();

	// If the bullet hits a baddie, set it as inactive
	if(m_baddies[i].Overlapping(&m_bullet)) {
		m_baddies[i].SetActive(false);
		m_bullet.SetActive(false);
		m_playerScore += 100;
		}
	}
}
```

The previous code table sets the baddies' speed relative to the first `timeDelta` value. This is not a uniform way to move models, since the first `timeDelta` value will probably be slower than subsequent ones; it will do for this demo code, but you would never normally do this.

The changes to the `Render` method are fairly extensive. Basically, we render the 2-D background, then the spaceship, the baddies, and the bullet. We then render the player's score so it appears above the objects and background. I have not highlighted the changes in the following code table because almost all the code is changed from the last time we saw this method.

```
void SimpleTextRenderer::Render() {
	m_d3dContext->ClearRenderTargetView(m_d3dRenderTargetView.Get(),
	(float*) &XMFLOAT3(0.39f, 0.58f, 0.93f)); // Clear to cornflower
blue
```

```
	// Render 2-D background so it appears behind everything
	m_d2dContext->BeginDraw();
	m_bitmapbackground.Render(m_d2dContext);
	HRESULT hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

	// Clear the depth stencil
	m_d3dContext->ClearDepthStencilView(m_d3dDepthStencilView.Get(),
D3D11_CLEAR_DEPTH, 1.0f, 0);

	// Set the render target
	m_d3dContext->OMSetRenderTargets(1,
m_d3dRenderTargetView.GetAddressOf(), m_d3dDepthStencilView.Get());

	// Set to render a triangle list
	m_d3dContext-
>IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

	// Set the input layout
	m_d3dContext->IASetInputLayout(m_vertexShader.GetInputLayout());

	// Set the vertex shader
	m_d3dContext->VSSetShader(m_vertexShader.GetVertexShader(),
nullptr, 0);

	// Set the vertex shader's constant buffer
	m_d3dContext->VSSetConstantBuffers(0, 1,
m_constantBufferGPU.GetAddressOf());

	// Set the pixel shader
	m_d3dContext->PSSetShader(m_pixelShader.GetPixelShader(),
nullptr, 0);

	// Set the sampler state for the pixel shader
	m_d3dContext->PSSetSamplers(0, 1, &m_samplerState);

	// Load the data from the CPU into the GPU's constant buffer
	m_d3dContext->UpdateSubresource(m_constantBufferGPU.Get(), 0,
NULL, &m_constantBufferCPU, 0, 0);

	// Render the spaceship
	m_spaceship.Render(m_d3dContext.Get());
```

```
	// Render the baddies
	for(int i = 0; i < 5; i++) {
		XMStoreFloat4x4(&m_constantBufferCPU.model,
			XMMatrixTranspose(
				m_baddies[i].GetTranslationMatrix()
				));
		m_d3dContext->UpdateSubresource(m_constantBufferGPU.Get(),
0, NULL, &m_constantBufferCPU, 0, 0);
		m_baddies[i].Render(m_d3dContext.Get());
		}

	// Render the bullet
	XMStoreFloat4x4(&m_constantBufferCPU.model,
		XMMatrixTranspose(
			m_bullet.GetTranslationMatrix()
			));
	m_d3dContext->UpdateSubresource(m_constantBufferGPU.Get(), 0,
NULL, &m_constantBufferCPU, 0, 0);
	m_bullet.Render(m_d3dContext.Get());

	// Render 2-D background so it appears behind everything
	m_d2dContext->BeginDraw();

	// Print the player's score:
	m_d2dContext->SetTransform(m_orientationTransform2D);
	// Set up the string to print:
	std::wstring s = std::wstring(
		L"Score: ") + std::to_wstring(m_playerScore);
	// Render the string in the top left corner
	m_d2dContext->DrawText(s.c_str(), s.length(), m_textFormat.Get(),
		D2D1::RectF(0, 0, 600, 32), m_scoreBrush.Get());

	hr = m_d2dContext->EndDraw();
	if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);
}
```

And finally, the most important step, hit F5 to debug. You should see a 3-D space shooter (*Figure 10.3*).

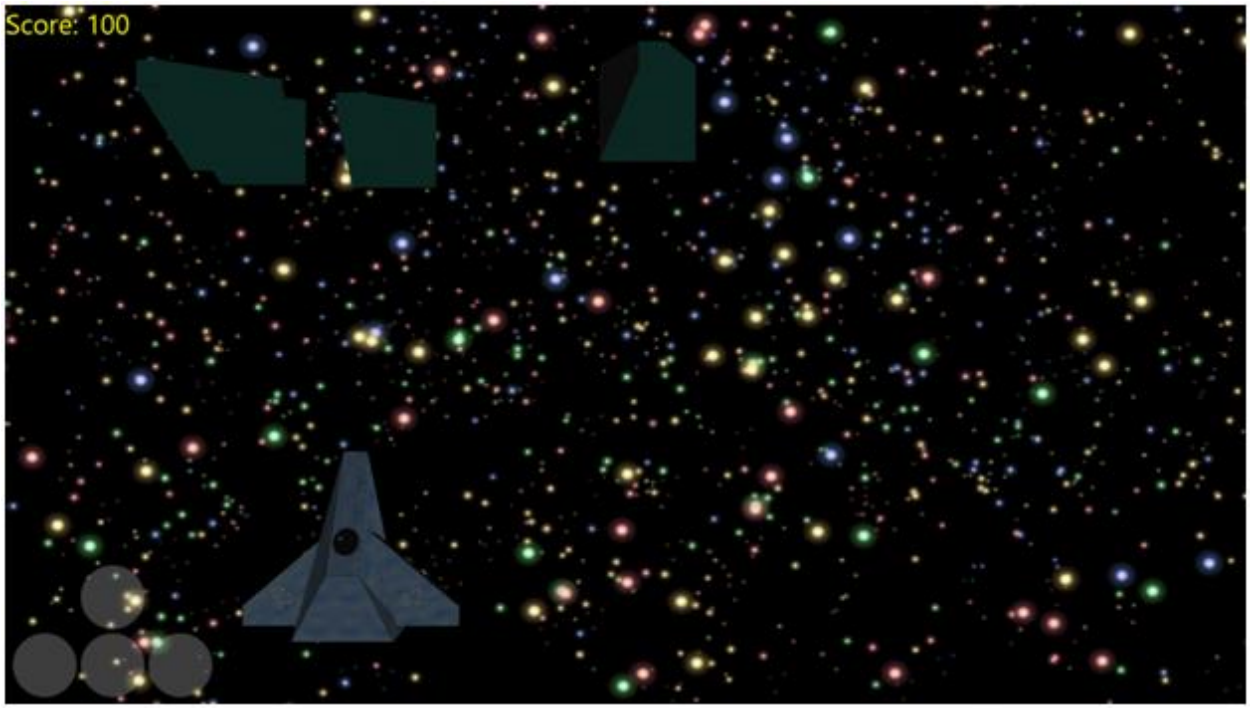

Figure 27: Figure 10.3: Space Shooter

Space shooter may have been an exaggeration, but we certainly have the rudiments of a 3-D shooter. Use the left and right keys to move the spaceship, and the up key to shoot. You can also use the mouse and click on the on-screen dpad or use the touchscreen if you have a Windows Surface or other WinRT device.

Chapter 11 Further Reading

MSDN: The first port of call for almost all DirectX and Visual Studio related enquiries. Microsoft's online MSDN reference contains a massive collection of information on DirectX and many other things. There is also an extensive set of samples called *Windows 8 app samples* that are well worth downloading and studying. The entire website is gigantic, but you might like to start by visiting: http://msdn.microsoft.com/en-us/library/windows/apps/hh452744.aspx

Beginning DirectX11: This book by Allen Sherrod and Wendy Jones presents similar information to what we have been through in this book. It is only the basics of DirectX11, but it is very well written and informative. It is available online from Amazon:

http://www.amazon.com/Beginning-DirectX-11-Game-Programming/dp/1435458958

RasterTek DirectX Tutorials: The website http://www.rastertek.com/ has incredibly useful tutorials on many generations of DirectX, including DirectX 11. It has tutorials organized by topic, for things such as bump mapping, reflection, etc. The code and text is clear and well written. Many online tutorial series do not closely conform to Microsoft's coding practices, which can quickly lead to horrid debugging episodes. The RasterTek tutorial's code, on the other hand, is very robust.

Graphics Gems series: This is a series of four free eBooks available online as PDF documents. The Graphics Gems series covers a large collection of algorithms and short papers on programming graphics. Much of the information is quite old and rather low level, but these algorithms and ideas still constitute the foundations of modern graphics. This stuff is what DirectX is built from. I have yet to find a stable site with downloads for the Graphics Gems series, so I suggest searching for "graphics gems filetype:pdf" using your favorite search engine to find the PDFs.

GPU Gems: This is a series of books from Nvidia. It is similar to the Graphics Gems series, but it is based on more modern hardware and DirectX. The series is available to read online as HTML or you can purchase a printed copy from Nvidia:

https://developer.nvidia.com/content/gpu-gems

https://developer.nvidia.com/content/gpu-gems-2

https://developer.nvidia.com/content/gpu-gems-3

Graphics Programming Black Book: Long before DirectX, 3-D games were being created for hardware that was barely capable of rendering smooth 2-D graphics. Michael Abrash worked with John Carmack on DooM II and the original Quake engine. He also wrote this book that he called *Graphics Programming Black Book*. The information it contains is very low level, but this book is easily one of the most interesting and informative books ever written on the topic of graphics programming and optimization. It is available for free from many sites including:

http://www.drdobbs.com/parallel/graphics-programming-black-book/184404919

Game Design Secrets of the Sages: Compiled by Marc Saltzman. This is a book on game design more so than programming. The author interviewed hundreds of people in the field of game developers, including some of the biggest names in the industry. The book is the result of these interviews; it contains an incredible amount of rare and useful information. The latest edition is available from Amazon:

http://www.amazon.com/Game-Design-Secrets-Sages-Guide/dp/1566869048

Multimedia Fusion: This is not a book; it is a game making application by a company called the ClickTeam. I have included it as a reference because I think that it is the best way to illustrate the structure of an object-oriented game engine. It requires no programming skills and it can be used to create a variety of different game types. It is not traditionally 3-D. The friendly user interface, the way objects interact, and the event handling system is exactly how an object-oriented game engine should be. There is a free demo and a registered version available from the ClickTeam website:

http://www.clickteam.com/website/world/