# DirectX 11.1 Game Programming

A step-by-step guide to creating 3D applications and interactive games in Windows 8

Pooya Eimandar

# DirectX 11.1 Game Programming

A step-by-step guide to creating 3D applications and interactive games in Windows 8

**Pooya Eimandar**

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

# DirectX 11.1 Game Programming

# Credits

**Author**
Pooya Eimandar

**Reviewers**
Doron Feinstein
Stephan Hodes
Vinjn Zhang

**Acquisition Editors**
Saleem Ahmed
Erol Staveley

**Commissioning Editor**
Yogesh Dalvi

**Technical Editors**
Ruchita Bhansali
Aniruddha Vanage

**Copy Editors**
Gladson Monteiro
Aditya Nair
Alfida Paiva

**Project Coordinator**
Amigya Khurana

**Proofreader**
Paul Hindle

**Indexer**
Mariammal Chettiyar

**Graphics**
Ronak Dhruv

**Production Coordinator**
Adonia Jones

**Cover Work**
Adonia Jones

# About the Author

**Pooya Eimandar** was born on January 07, 1986. He graduated with a degree in Computer Science and Hardware Engineering from Shomal University and has been programming mainly in DirectX and OpenGL since 2002.

His main research interests are GPU-programming, image processing, parallel computing, and game developing.

Since 2010, he has been leading a game engine team for a company Bazipardaz, working on their latest titles for Xbox 360 and PC. You can find more information about this at `http://persianengine.codeplex.com/`.

# About the Reviewers

**Doron Feinstein** is a Senior Graphics Programmer at Rockstar Games and is the author of the book *HLSL Development Cookbook* published by *Packt Publishing*.

After working with simulations for a number of years, he decided to switch to an exciting career in the games industry. Max Payne 3 and All Points Bulletin (APB) are among some of the titles Doron has worked on commercially.

**Stephan Hodes** has been working as a Game Engine programmer for almost 15 years while GPUs made the transition from fixed function pipeline to programmable shader hardware. During this time, he worked on a number of games released for PC as well as for Xbox 360 and PS3.

Since he joined AMD as a Developer Relations Engineer in 2011, he has worked with a number of European developers on optimizing their technology to take full advantage of the processing power that the latest GPU hardware provides.

He is currently living with his wife and son in Berlin, Germany.

**Vinjn Zhang** is an enthusiastic Software Engineer. His main interests in programming include game development, graphics shader writing, human-computer interaction, and computer vision. He has translated two technical books into Chinese, one for the processing language and one for OpenCV.

Vinjn Zhang has worked for several game production companies including Ubisoft and 2K Games. He is currently working as a GPU Architect for NVIDIA, where he gets the chance to see the secrets of GPU. Besides his daily work, he is an active github user. He tries to make every piece of code open source. His website is also an open source repositoryVisit his website `http://vinjn.github.io/`.

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

### Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

### Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

*Dedicated to my mother*

# Table of Contents

# Preface

In the last few years, the number of devices in which complex graphics are embedded has vastly increased. Recently, Microsoft released a new version of Windows called Windows 8. The Direct3D 11.1 API is also included with Windows 8 and provides a significant expansion in capabilities over it's previous version.Microsoft showed that Direct3D 11.1 plays a key role in writing high-performance 3D Metro applications in Windows 8. To ease portability, Windows 8 introduces a new type of application called the Windows Store application, which is a great opportunity for developers to write cross-platform applications to write cross-platform applications over the Microsoft platforms.

This book will help you easily create your own framework and build your first game for Metro Style all by yourself in order to publish it on the Windows Store.

## What this book covers

*Chapter 1*, *Say Hello to DirectX 11.1*, covers the new features of Windows 8, DirectX 11.1, and the new extension of C++ called C++/CX. This chapter also covers how to set up a framework and initialize the Direct3D device.

*Chapter 2*, *Getting Started with HLSL*, provides you with a preliminary knowledge of the new features of HLSL in DirectX 11.1 and explains how to interact with buffers in Direct3D. It also introduces the new additions of Direct2D for Windows 8.

*Chapter 3*, *Rendering a 3D Scene*, presents the details of system usages and how to use the Visual Studio Model Editor to render models. This chapter also covers how to handle inputs, cameras, and finally integrate XAML and Direct3D.

*Chapter 4*, *Tessellation*, introduces the tessellation stages. It also outlines how to use the graphics debugging feature in Visual Studio 2012.

*Chapter 5*, *Multithreading*, introduces the C++ AMP library and the Compute Shader stage and compares the performances of both.

# What you need for this book

You will need a desktop or a tablet running Windows 8 as a test machine. It is assumed that you already have knowledge of C and C++ languages, pointers, object-oriented features, and other modern language concepts. You also need to be familiar with DirectX programming along with the math and the 3D coordinate systems. also You need to use Microsoft Visual Studio Express 2012 for Windows 8 as a developer environment and finally make sure to set up the Windows 8 SDK before starting with this book.

This book is for:

- Those who would like to become a game programmer and who wish to start their game development journey
- Those who wish to be a game developer to create and sell their game on the Windows Store
- A DirectX developer who needs more performance from DirectX 11.1
- A C/C++/C# developer who needs to switch to the new platform of Microsoft
- Those who would like to use XAML and C++/CX for their graphical applications

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Now you can capture your frame by calling the `g_pVsgDbg->CaptureCurrentFrame()` function"

A block of code is set as follows:

```
float4 vertexPosition =
domain.x * patch[0].pos +
domain.y * patch[1].pos +
domain.z * patch[2].pos;
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

# Downloading the example code

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

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

# Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

# Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1
# Say Hello to DirectX 11.1

In the last few years, the range of devices in which complex graphics are embedded has vastly increased. This is great for users, but a demanding challenge for developers. Direct3D 11 had provided a significant expansion in capabilities over the previous version; now, Direct3D 11.1, the new version of DirectX 11 which is shipped in Windows 8, presents some new features.

This chapter is about DirectX 11.1. Before learning about programming with the new component extension (C++/CX) of Microsoft and DirectX 11.1, it seems necessary to take a step backward and commence with the new features of DirectX 11.1 on a Metro Style application to discover why we must move to the newer version. This chapter will cover the following:

- The need for DirectX 11.1
- The prerequisites
- Introduction to C++ component extensions (C++/CX)
- Metro Style apps
- Setting up your first project
- Building your first Metro app
- Working with game time
- Initializing the device

At the end of this chapter, a running DirectX 11.1 project under Windows 8 and a framework will be created that supports game time.

# The need for DirectX 11.1

Microsoft DirectX, first released in 1995, is a set of standard commands that can be used by developers to send instructions to video cards. In 2002, Microsoft released DirectX 9, and later in August 2004, DirectX 9.0*c* was released, which supported shader model 3.0.

DirectX 9 was only supported by the Xbox 360 console and Windows XP and later versions. DirectX 10 changed to a new pipeline and was only available on Windows Vista and later versions. DirectX 10.1, which was shipped with Service Pack 1 of Windows Vista, was the updated version of DirectX 10.

In 2008, Microsoft unveiled DirectX 11, which had some amazing features including a compute shader, dynamic shader linking, tessellation, multithreaded rendering, and shader model 5.0. DirectX 11 can be run on Windows Vista and Windows 7.

Now, Direct3D 11.1 is due to ship with Windows 8. In fact, everything in Windows 8 is based on DirectX 11.1; it comes with several new features, such as the following:

- Device sharing
- HLSL minimum precision
- Support a larger number of UAVs and use them in every pipeline stage
- Force the sample count to create a rasterizer state
- Create larger constant buffers than a shader can access
- Discarding resource and resource views
- Support stereoscopic 3D for gaming and videos

There are many other features, and we will introduce some of them in this book.

> At the moment, Windows 7 Service Pack 1 and Windows Server 2008 R2 Service Pack 1 can be updated via the KnowledgeBase 2670838 platform update to improve and use some features of DirectX 11.1. For more information, check the article at `http://support.microsoft.com/kb/2670838`.

# Why should we target Windows 8?

At the **Game Developers Conference** (**GDC**) 2013, Windows director of communications, *Christopher Flores*, said:

> *"60 million Windows 8 licenses had been sold during the first two months of launch, matching the "record-setting" sales in a similar time period for Windows 7. The Store had seen double digit growth in the number of visiting users week over week since its October launch, with 95 percent of the apps available having been downloaded."*

According to the Steam Hardware and Software Survey of May 2013, Windows 8 (64-bit) surpasses Windows XP in popularity and is almost as popular as Windows 7 (32-bit).

| MOST POPULAR | PERCENTAGE | CHANGE |
| --- | --- | --- |
| Windows 7 64 bit | 55.40% | -0.69% |
| Windows 7 | 13.87% | +0.01% |
| Windows 8 64 bit | 12.25% | +1.00% |
| Windows XP 32 bit | 7.92% | -0.62% |
| Windows Vista 64 bit | 4.81% | +0.20% |
| Windows Vista 32 bit | 2.69% | -0.01% |
| Windows 8 | 0.83% | +0.02% |

Windows Store, which is only available with Windows 8, provides new opportunities for everyone. It's a big chance for developers to write games and sell them in the Store.

For distributing your game on Windows Store, you need to port your code to DirectX 11.1. Windows 8 is the only option for programming with DirectX 11.1; you need to use Visual Studio 2012 along with Windows 8.

Whenever you build your application, you can follow the instructions from the *Windows Dev Center* article for an introduction to publishing your applications on the Store. For more information, please check this article at the following link:

`http://msdn.microsoft.com/en-us/library/windows/apps/br230836.aspx`.

Windows 8 will be cross-platformed, which means your game can be run on Microsoft platforms supported by Windows 8, such as PCs, Microsoft tablets, and Windows Phone 8, but note that each platform has its own limitations.

The new generation of Microsoft's games console, Xbox One, runs on three separate operating systems. The one where games run is NT Core, which is shared between Windows 8, Windows Phone 8, Windows RT, and Windows Server 2012. This means that developers can easily port their code from Windows 8 to other platforms and there is no need to rewrite their code from the very first line.

Finally, the simplified UI programming of Metro Style development should not be forgotten. Microsoft introduces Metro Style development in C++, C#, XAML, HTML, and JavaScript for experienced as well as newbie developers.

> **A good game comes from a good design, not just good graphics!**
>
> If you prefer to work alone, you can try to create a simple game and publish it on Windows Store. However, your games will be more likely to succeed if you build a game with a team that includes a game designer, a concept designer, a sound designer, and an animator.

As you go through this book, it might be useful to start creating your own first game for Metro Style, writing and testing samples all by yourself; if you face any problems, you can check different websites, such as *MSDN Code Gallery* at `http://code.msdn.microsoft.com`, for improving your game code. The samples and articles of Microsoft DirectX SDK (June 2012): *AMD Developer Central* (`http://developer.amd.com`) and *nvidia Developer Zone* (`http://developer.Nvidia.com`) are the best references for getting started with DirectX.

# The prerequisites

The following are some essential prerequisites that should be considered before developing an application using DirectX 11.1:

- The first important requirement is a personal computer or tablet running on Windows 8.

- You need to set up Microsoft Visual Studio Express 2012 for Windows 8 as a developer environment. Furthermore, you should be familiar with DirectX programming and object-oriented programming with C++ and have a basic understanding of the math and the 3D coordinate systems.

- Make sure you set up the Windows **Software Development Kit** (**SDK**) for Windows 8. The Windows SDK for Windows 8 contains headers, libraries, and tools that can be used when creating Windows 8 applications.

- As a further aid to understanding how to run the samples of this chapter, all of the source code is provided for your reference, and they have been used in an open source rendering framework; so, please make sure to get the source code from Packt Publishing's website when reading this book.

> For those who come from a background of C# to C++, it is suggested that for better management and refactoring of C++ code, use Visual Assist X (`http://www.wholetomato.com/`). Please note that it is not free, but a free trial version of 30 days with all functionalities is available.
>
> Check the following link if Visual Studio 2012 is not available. Make sure to get the right product for Windows 8:
>
> `http://www.microsoft.com/visualstudio/eng/products/`
> `visual-studio-express-products`
>
> Use the following link to download Windows Software Development Kit (SDK) for Windows 8:
>
> `http://msdn.microsoft.com/en-us/windows/hardware/`
> `hh852363.aspx`

# Introduction to C++/CX

**Microsoft C++ with Component Extensions** (**C++/CX**) is an update to Microsoft C++, which is the language for developing a Windows Store game running on DirectX 11.1. It is truly mapped out to the new generation of the native C++ programming and is similar to C++/CLI; it is actually native code and not managed code. C++/CX is needed when you want to write C++ code-behind for Windows Store applications which uses the XAML user interface. Alternatively, it creates a **Dynamic Link Library** (**DLL**) to be called out from a Windows Store application, which is built using JavaScript, C#, or Visual Basic. And finally, it writes the Windows Store game using DirectX 11.1.

C++/CX brings the power of C++ along with the ease of programming. *Jim Springfield*, an architect of the Visual C++ team, wrote in his MSDN blog:

> "*However, while the C++/CX syntax is very similar to C++/CLI, the underlying implementation is very different; it does not use the CLR or a garbage collector, and it generates completely native code.*"

Have a look at his blog at the following link:

`http://blogs.msdn.com/b/vcblog/archive/2011/10/20/10228473.aspx`

> If you want more resources for C++/CX, take a look at the *Visual C++ Team blog* at the following URL:
>
> `http://blogs.msdn.com/b/vcblog/archive/2012/08/29/cxxcxpart00anintroduction.aspx`
>
> I'd recommend the Hilo, an example application, as a resource for starting out (`http://hilo.codeplex.com`).

C++/CX is a great place for developing applications for those game programmers who come from a background of C#, such as XNA Platform, SlimDx, and SharpDx. Also, for those who come from the management background of C++ or native C++, C++/CX won't disappoint! As mentioned earlier, you can use C++/CX along with XAML, but it is only available under the Metro UI; this means that C++/CX only runs on Windows 8 platforms.

There is no need to use native C++ while using **Windows Runtime C++ Template Library** (**WRL**); instead, you can use C++/CX code as a high-level language for using WRL code. In other words, it helps you write simpler code.

According to MSDN, the purpose and design of the WRL is inspired by the **Active Template Library** (**ATL**), which is a set of template-based C++ classes that simplifies the programming of **Component Object Model** (**COM**) objects.

# Lifetime management

To have a better overview of C++/CX, let's check its benefits.

WRL includes `ComPtr`, which is an improved version of ATL's `CComPtr` class. It means that most of the functions of `ATL::CComPtr` are implemented in `WRL::ComPtr`. On the other hand, C++/CX has a `hat` (`^`), which is the same as `ComPtr`. If you are familiar with C++ 11, you will realize that both `ComPtr` and `^` have the same behavior as that of `shared_ptr`. Note that only the behavior is similar, not the implementation.

The following is a simple class and two interfaces in C++/CX:

```
interface struct IGet
{
  int Get() = 0;
};
interface struct ISet
```

```
{
    void Set(int value) = 0;
};
ref class CProperty sealed : IGet, ISet
{
public:
    CProperty()
    {
        OutputDebugString(L"CProperty created. \n");
    }
    virtual int Get() { return this->field; }
    virtual void Set(int value) { this->field = value; }
private:
    ~CProperty()
    {
    OutputDebugString(L"CProperty released. \n");
    };
    int field;
};
```

**Downloading the example code**

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

When the `CProperty` object is created or destroyed, the corresponding message is shown on the output debug window of Visual Studio.

# What is a ref class?

All Windows runtime classes are derived from the `IInspectable` interface, and the `IInspectable` interface inherits from the `IUnknown` interface. C++/CX supports `ref` classes and `ref` structs that have got the managed lifetime by the automatic reference counting. It suggests that when you create a new reference to an object, `IUnknown::AddRef` is automatically called in order to increment the reference count, and when the destructor of the object is called, `IUnknown::Release` automatically decrements the reference count.

```
CProperty^ a = ref new CProperty();
```

In order to allocate a `ref` class in the dynamic memory, use the `ref new` keyword; also note that the hat is actually a smart pointer in C++/CX.

# Inheritance

In C++/CX, the object `Platform::Object` is the base class on the platform namespace that provides a common behavior for Windows runtime classes. Unlike native C++, C++/CX does not support multiple inheritance; however, if you still want to use this feature, you need to use the native C++ code within the Metro app. If you are a C++ programmer, you will see that there are a lot of differences between inheritance in C++ and C++/CX. Let's check those differences with reference to the following example. Change the `CProperty` class until the `CObject` class is able to inherit from it:

```
ref class CProperty : IGet, ISet
{
internal:
  CProperty(){};
protected:
  virtual void GetType(){};
protected public:
  virtual void GetName(){};
public:
  virtual int Get() { return this->field; }
  virtual void Set(int value) { this->field = value; }
private:
  int field;
};
ref class CObject sealed : public CProperty
{
public:
  CObject(){};
protected:
  virtual void GetType() override
  {
    CProperty::GetType();
  };
};
```

`CObject` is declared as a `sealed` class, which means this class cannot be used as a base class and its virtual members cannot be overridden. The `sealed` keyword is used for the `ref` classes just as the `final` keyword is used in C++ 11, which is used for the standard classes and functions.

These are the basic rules for inheritance in C++/CX:

- The base `ref` class has got an internal constructor; in other words, an unsealed `ref` class cannot have a `public`, `protected`, or `protected public` constructor.

- The `ref` classes can implement any number of interfaces. As you can see, `CProperty` is implemented from two different interfaces (`ISet` and `IGet`).

- The `ref` classes can inherit from only one base `ref` class. As you can see, the `CObject` `ref` class inherits from only the `CProperty` base `ref` class.

- Only the `sealed` class has got a `public` constructor in order to prevent further derivation.

- Access to a `protected` member declared in the `ref` class is not possible; it is possible only if the member is declared as `protected public`. See the declaration of the `GetName` and `GetType` functions.

- For `ref` classes, only the `public` inheritance is supported. For instance:

```
ref class CObject sealed : public CProperty // OK
ref class CObject sealed : private CProperty // Error
```

# Delegates and events

Delegates are most commonly used in conjunction with events. Let's discuss both of them in this section.

Many components in the Windows runtime expose events, and each event has a delegate type. You can declare an event in a `ref` class or inside an interface. The following example shows how to declare events and delegates:

```
namespace EVENT
{
  //Forward declaration
  ref class CEvent;
  public delegate void CEventHandler(CEvent^ sender, Platform::String^
e);
  void OnEvent(CEvent^ sender, Platform::String^ e)
  {
        //Do something
  }
  public ref class CEvent sealed
  {
```

```
    public:
      CEvent(){};
      event CEventHandler^ Event;
      void RiseEvent()
      {
        Event(this, L"CEvent raised");
      }
    };
  }
```

First, we declare a delegate named `CEventHandler`. This delegate has two arguments, a sender that is a type of `CEvent` class, and a standard string of Windows runtime.

The following code shows how to initialize an event, fire it, and how to use the callback method (`OnEvent`). The `EventRegistrationToken` class helps you store the token of an event in order to remove the event handler that is previously subscribed.

```
  CEvent^ c = ref new CEvent();
  Windows::Foundation::EventRegistrationToken token = c->Event += ref
  new CEventHandler(&OnEvent);
    c->RiseEvent();
    c->Event -= token;
```

In this section, you were introduced to C++/CX. Every application has got unique requirements for programming. Every Windows Store application is built on Windows runtime, and the core of Windows runtime is **Application Binary Interface** (**ABI**), which is built on COM. Although we can use native C++ to directly interact with COM, don't forget that this method is not as simple as when using C++/CX.

# Metro Style apps

Prior to getting started with coding, it is crucial to discover what Metro Style applications are and whether they can be cross-platform.

Windows 8 introduces a new type of application called a Windows Store app. Your application can be written and sold on Windows Store. It is a great opportunity for developers to write applications that can be run on PCs, tablets, and Windows Phones, and it might even be possible to run on the new generation of Xbox; that is, Xbox One!

In Metro Style, you can develop in a variety of languages. If you are a web developer, you may prefer to develop with HTML5, CSS3, and JavaScript; if you are a .NET developer, you can develop with XAML with code-behind in C++, C#, or Visual Basic.
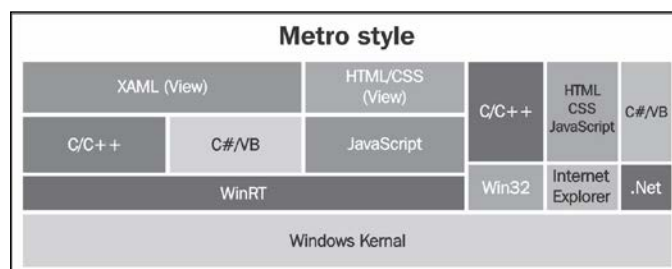


If you are a game programmer, you can use DirectX 11.1 with C++/CX and HLSL to develop your game; if you prefer the .NET platform for game development, it is possible to write a wrapper for C++/CX.

Please note that Windows Store application development is supported only by Windows 8. Visual Studio 2012 supports ARM development, hence all Metro Style applications in the Windows Store work on both ARM and x86/64. The x86/x64 processors are fast and powerful, but they require a lot of electricity; on the other hand, ARM processors are weak and need low power, and they are used for PDAs, smartphones, and tablets. Because of the difference in hardware, a program that is built on an x86/x64 processor cannot run on an ARM processor and vice versa.

# Setting up your first project

For creating your project, the first step is opening Visual Studio 2012 and creating a new Windows Store project from the C++ tab by performing the following steps:

1. Open Visual Studio 2012 and navigate to **File | New | Project** (or Press *Ctrl + Shift + N*). You will see a project dialog box appear.

2. Click on the **Visual C++** tab and select **Windows Store**; then select **Direct3D App** and choose a name for your project, such as Chapter 1 or any other name you prefer. Make sure you browse the right path for your project's location; finally, click on **OK**.

3. In order to develop an application on Windows 8, you will need a developer license. When you see the **Developer License** dialog box, first read Microsoft's terms and then click on **I Agree**.



4. You will need to have a Microsoft account. If you already have one, log in with your own account; if you don't, sign up to get a new account.

5. Press *F5* and Visual Studio will output this: **Would you like to build?**. Click on **Yes**; when the building process completes, you will see a cube that rotates in the center of the screen.

Don't worry if the code seems overwhelming. In the next section, we will guide you through writing your own framework in order to get a better understanding of it.

> Microsoft's slogan is "Windows 8 means apps". It means anyone can create their own app and publish them on the Internet!

# Building your first Metro app

We have introduced C++/CX and ran our first sample; now it is time to build our own framework gradually in order to get familiar with DirectX 11.1 programming:

1. Open the `First Metro App` project from the source code of `Chapter 1`. If you cannot find the **Solution Explorer** pane, press *Ctrl + Alt + L* or navigate to **View** | **Solution Explorer** to enable it.

2. In the **Solution Explorer** pane , enable **Show All Files**. This feature lets us use `Folder` instead of `Filter` in our project. However, you can add `Filter` and just change it to the path of `includes` in your project.

Let's check the `pch.h` and `pch.cpp` files from the source code. The `pch.h` file is shown in the following code:

```
#pragma once
#include <wrl/client.h>
#include <d3d11_1.h>
#include <DirectXMath.h>
#include <memory>
#include <ppl.h>
#include <ppltasks.h>
#include <agile.h>
```

If you are a C++ programmer, you will see unfamiliar header files named `wrl`, `agile.h`, `ppl.h`, and `ppltasks.h`. Also, `DirectXMath.h` is unfamiliar for DirectX programmers. What are they supposed to do?

`wrl` headers are needed to take advantage of the Windows 8 libraries, and `agile.h` is the header needed to enable threading model and marshaling behavior across COM.

In our project, we need an `agile` object type to create `CoreWindow`. If you want to access an instance of a class from both the UI thread and a background thread, define the class as `agile`. If you do not want to define it as `agile`, be aware of the behavior of this class at runtime. Remember that if you define a `ref` class in C++/CX, it is `agile` by default, and do not define a member that is not `agile` inside this type of class. According to MSDN, some classes cannot be `agile` for a variety of reasons.

We added `ppl.h` and `ppltasks.h` to use Windows runtime asynchronous tasks; we will see these headers in action later.

`DirectXMath` is a library designed for C++ developers, and it includes functions for common linear algebra and graphical math operations.

You may be familiar with precompiled headers; it is a performance feature supported by some compilers to reduce the compilation time and store the compiled state of code in a binary file. Because we want to create a precompiled header to compile and build the project more quickly, we need to change some properties. Follow the ensuing steps:

1. Right-click on **Project** and go to the **Properties** page. From **C/C++**, go to the **Precompiled Headers** tab and make sure **Precompiled Header** is set to **Use(/Yu)** and **Precompiled Header File** is set to **pch.h** as shown in the following screenshot:

2. Finally, right-click on **pch.cpp** and make sure **Precompiled Header** is set to **Create [/YC]**:
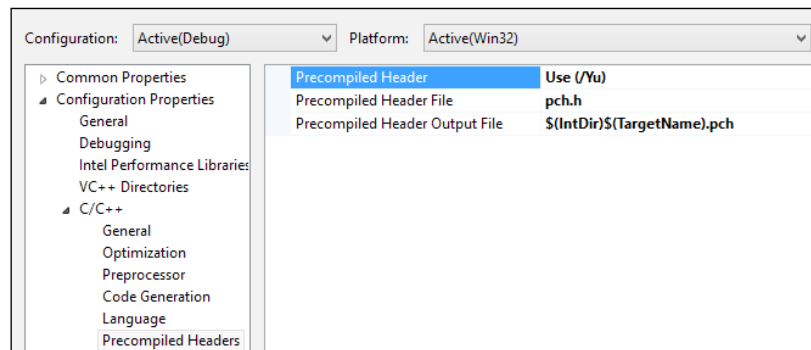


In the `Scenes` folder, there are the `Window.h` and `Window.cpp` files.

The `Window.cpp` file is responsible for creating a window and handling the events, and it must be inherited from `IFrameworkView`. The `IFrameworkView` interface is shown in the following screenshot:
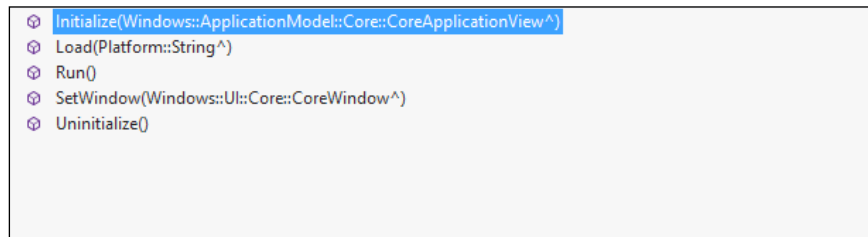


The following are the methods seen in the preceding screenshot:

- The `Initialize` method is for the app view and the initialization of graphics. This method is called when an app is launched.

- The `Load` method loads any external resources used by the app view, such as shaders, textures, or other resources.

- The `Run` method is to start the app view.

- The `SetWindow` method is for setting the current window for the app object's view.

- The `Uninitialize` method is responsible for releasing and disposing the resources.

The following are other events for handling our window more easily:

- The `OnWindowSizeChanged` event is triggered when the window size is changed.

- The `OnLogicalDpiChanged` event occurs when the `LogicalDpi` property changes because the **pixels per inch** (**PPI**) of the display changes.

- The `OnActivated`, `OnSuspending`, and `OnResuming` events are triggered when the state app is activated, suspended, or resumed.

- The `OnWindowClosed` event, as its name implies, is triggered when the window is closed.

- The `OnVisibilityChanged` event is an event handler that is triggered when the visibility of the window is changed.

- The `OnPointerPressed` event is an event handler that our app calls when the user taps on the touch pad or presses the left mouse button.

- The `OnPointerMoved` event is the same as `OnPointerPressed` but is called when the player swipes a finger across the touch pad or moves the mouse.

Now that we have introduced the preceding methods and events, let's see their definition in the `Window.cpp` file as shown in the following screenshot:

```
void Window::Initialize(CoreApplicationView^ appView)
{
 appView->Activated += ref new TypedEventHandler<CoreApplicationView^,
IActivatedEventArgs^>(this, &Window::OnActivated);

  CoreApplication::Suspending += ref new EventHandler<SuspendingEventA
rgs^>(this, &Window::OnSuspending);

  CoreApplication::Resuming += ref new EventHandler<Platform::Object^>
(this, &Window::OnResuming);
}
```

According to the `IFrameworkView` interface, the entry argument of the `Initialize` method is `CoreApplicationView`, which is a pointer. We declare the `Activated`, `Suspending`, and `Resuming` events likewise. Just as in the `Initialize` method, we declared the events of the window, such as `SizeChanged`, `VisibilityChanged`, `Closed`, and `Pointers`, in the `SetWindow` method. Right now, there is no need to load the resource, so leave this method as it is for now.

```
void Window::Run()
{
  while (!isWindowClosed)
  {
```

```
      if (isWindowVisible)
      {
        CoreWindow::GetForCurrentThread()->Dispatcher->ProcessEvents(Cor
   eProcessEventsOption::ProcessAllIfPresent);
      }
      else
      {
        CoreWindow::GetForCurrentThread()->Dispatcher-
   >ProcessEvents(CoreProcessEventsOption::ProcessOneAndAllPending);
      }
    }
  }
```

The `Run` method is the main method of your application. This is an entry point to your game. While your app is not closed, it will check the visibility of your window; if the window is visible, it dispatches all currently available events in the queue. `Co reProcessEventsOption::ProcessAllIfPresent` calls the handlers for each input event that is currently in the message queue. If no events are pending, it returns immediately and waits for the next event.

Do not use `Uninitialize`, as there are no resources to be released right now; the same goes for `OnWindowSizeChanged`. Furthermore, `OnPointerPressed`, `OnPointerPressed`, and `OnPointerMoved` events are going to be used later in *Chapter 3*, *Rendering a 3D Scene*, for handling the input.

```
  void Window::OnActivated(CoreApplicationView^ applicationView,
    IActivatedEventArgs^ args)
  {
    CoreWindow::GetForCurrentThread()->Activate();
  }
```

When the `OnActivated` event triggers, you must activate the current thread. The `Co reWindow::GetForCurrentThread()` method gives you the current thread, and with the `Activate` method, you can activate this thread. Now we just need to create the main method with the array of the string as the entry arguments; so, add the following code to `Window.h`:

```
  ref class App sealed : IFrameworkViewSource
  {
  public:
    virtual IFrameworkView^ CreateView();
  };
```

The `App` class is inherited from `IFrameworkViewSource`; we need to add a public virtual method named `CreateView` to the `App` class in order to create `IFrameworkView`. At last, add the following code to `Window.cpp`:

```
IFrameworkView^ App::CreateView()
{
  return ref new Window();
}
[Platform::MTAThread]
int main(Platform::Array<Platform::String^>^)
{
  auto app = ref new App();
  CoreApplication::Run(app);
  return 0;
}
```

Press *F5* to build your project. The result is just an empty Metro window.

In the next section, we are going to integrate our framework with time.

# Working with game time

Time is the main element of any application; your game cannot be handled without time management. In this section, we are going to create a time manager. Open the `Work with game time` project from `Chapter 1` and then open `Timer.h` from the `FrameWork` folder.

`Timer` is a `ref` class that is declared as `sealed`. `Total` and `Delta` are two properties that are used in the `Timer` class.

The `Total` property is used to measure the duration (in seconds) between the last call to the `Reset` method and the last call to the `Update` method. It returns the total time of the timer. The `Delta` property is used to measure the duration (in seconds) between two calls to the `Update` method. The `Timer` method uses the `QueryPerformanceFrequency` function to retrieve the frequency of the high resolution performance counter.

```
void Update()
{
  if (!QueryPerformanceCounter(&currentTime))
  {
    throw ref new Platform::FailureException();
  }
```

```
    totalTime = static_cast<float>(
      static_cast<double>(currentTime.QuadPart -
        startTime.QuadPart) /
       static_cast<double>(frequency.QuadPart));
    if (lastTime.QuadPart == startTime.QuadPart)
    {
      deltaTime = 1.0f / 60.0f;
    }
    else
    {
      deltaTime = static_cast<float>(
        static_cast<double>(
        currentTime.QuadPart - lastTime.QuadPart) /
        static_cast<double>(frequency.QuadPart));
    }
    lastTime = currentTime;
  }
};
```

Basically, you use `QueryPerformanceFrequency` to get the number of ticks per second and `QueryPerformanceCounter` to get a high resolution timer value.

To calculate `totalTime`, use (`CurrentTime` - `StartTime`) / `frequency`; for calculating `deltaTime`, use (`CurrentTime` - `lastTime`) / `frequency`.

The `QueryPerformanceCounter` method will only reassemble the frequency of the CPU. Please note that according to MSDN, the results of `QueryPerformanceCounter` may vary with different processors.

In the next section, we will create and configure the Direct3D device.

# Initializing the device

In the final section of this chapter, we are going to initialize and configure our device. As the window has been created and the time has been added to our framework, the next step is to obtain a reference to the device. Open the `Initialize the device` project from the source code.

Let's check the new headers and classes of this project step-by-step. First, open
DXHelper.h from the Framework folder. The DXHelper.h file is shown in the
following code:

```
#pragma once
#include "pch.h"
using namespace Platform;
namespace DXHelper
{
  inline void OutputDebug(Platform::String^ Msg)
  {
    auto data = std::wstring(Msg->Data());
    OutputDebugString(data.c_str());
  }
  inline void ThrowIfFailed(HRESULT hr)
  {
    if (FAILED(hr)) throw Exception::CreateException(hr);
  }
}
```

We need two methods to manage our exceptions and to show messages to the output
debug window of Visual Studio. We defined them as inline methods inside the
DXHelper namespace. Game is a class that is designed to be used as the base class of
other scenes. ComPtr is in the Microsoft::WRL namespace, and Platform is vital
for Agile. Open it from Game.h, which is in the Framework folder. You can see the
following code in the Game.cpp file:

```
ref class Game abstract
{
internal:
  Game();
private:
  void CreateDevice();
  void CreateWindowSize();
  void HandleDeviceLost();
protected private:
  //D3D Objects
  ComPtr<ID3D11Device1> d3dDevice;
  ComPtr<ID3D11DeviceContext1> d3dContext;
  ComPtr<IDXGISwapChain1> swapChain;
  ComPtr<ID3D11RenderTargetView> renderTargetView;
  ComPtr<ID3D11DepthStencilView> depthStencilView;
```

You can declare the device as `ID3D11Device1*` or `ComPtr<ID3D11Device1>`; as mentioned earlier, the only difference is that `ComPtr` is a class that is responsible for automatically getting and releasing the object. It is not necessary to use it, but often it is more convenient to use `ComPtr` for the purposes of memory management.

```
D3D_FEATURE_LEVEL featureLevel;
  Size renderTargetSize;
  Rect windowBounds;
  Agile<CoreWindow> coreWindow;
DisplayOrientations displayOrientations;
```

> If you are going to develop a Metro Style application, you need to be concerned with the orientation of the screen. Because you can manage the orientation of the screen, access to `DisplayOrientations` from the `Windows::Graphics::Display` namespace is possible.

```
public:
  virtual void Initialize(CoreWindow^ window);
  virtual void WindowSizeChanged();
  virtual void Render() = 0;
  virtual void Present();
};
```

In the preceding code, all the methods declared are virtual methods except `Render`, which is declared as a pure virtual method. In C++, a pure virtual function (declared using `= 0`) must be implemented by the derived classes. The following code shows the `Game.cpp` class. These three headers are the precompiled header, our framework helper, and the `Game.h` header:

```
#include "pch.h"
#include "..\DXHelper.h"
#include "..\Game.h"

void Game:: CreateDevice ()
{
HRESULT hr =  S_FALSE;
  UINT creationFlags = D3D11_CREATE_DEVICE_BGRA_SUPPORT;
#if defined(_DEBUG)
  creationFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif
```

The `D3D11_CREATE_DEVICE_DEBUG` flag is used for enabling the debug layer when you want to check the errors and warnings of your code; for debugging, you need `D3D11_1SDKLayers.dll`; for getting this DLL, you must install SDK for Windows 8.

In Direct3D 11.1, a device can be created with some configuration flags that can be used in a majority of applications, but there are also several other driver types that must be considered; you can get more info about them from MSDN (`http://msdn.microsoft.com/en-us/library/windows/desktop/ff476107(v=vs.85).aspx`). An example of that is as follows:

```
D3D_FEATURE_LEVEL featureLevels[] =
  {
    D3D_FEATURE_LEVEL_11_1,
    D3D_FEATURE_LEVEL_11_0,
  };
```

The `D3D_FEATURE_LEVEL` driver describes the set of features targeted by a Direct3D device. These features can be set from level `9_1` to level `11_1`.

```
ComPtr<ID3D11Device> device;
ComPtr<ID3D11DeviceContext> context;
hr = D3D11CreateDevice(
    nullptr,
    D3D_DRIVER_TYPE_HARDWARE,
    nullptr,
    creationFlags,
    featureLevels,
    ARRAYSIZE(featureLevels),
    D3D11_SDK_VERSION,
    &device,
    &featureLevel,
    &context);
 DXHelper::ThrowIfFailed(hr);
hr = device.As(&d3dDevice);
hr = context.As(&d3dContext);
}
```

With `D3D11CreateDevice`, we create the Direct3D device and get an immediate context. Just as we do in `ComPtr`, call `QueryInterface` to signify that the interface has been identified. It is always important to check the return value from the API calls to ensure the function has succeeded.

# Connecting to a swap chain

A swap chain is a collection of buffers that are used for displaying frames to the user. When something is displayed on the monitor, the data in the back buffer is being filled while the data in the front buffer is being displayed. Actually, the front buffer is a pointer to a surface and represents the image being displayed on the monitor. The swap chain is responsible for presenting the back buffer by swapping it with the front buffer; this process is called surface flipping.

In the `CreateWindowSize` method, we initialized the swap chain. Firstly, we checked if the swap chain already exists or not. If the answer is yes, just resize it with the same adaptor. But, if the swap chain does not exist, we must create it as follows:

```
if(swapChain != nullptr)
{
  HRESULT hr = swapChain->ResizeBuffers(
    2, // Double-buffered swap chain.
    static_cast<UINT>(renderTargetSize.Width),
    static_cast<UINT>(renderTargetSize.Height),
    DXGI_FORMAT_B8G8R8A8_UNORM,
    0);
}
```

In order to create the swap chain, we first need to define the swap chain description. The properties of the swap chain description are as follows:

- The `width` and `height` properties of the resolution
- The `DXGI_FORMATDXGI_FORMAT` structure, which describes the display format
- The `Stereo flag` property is either true, which means the fullscreen display mode or the swap chain back buffer is stereo, otherwise the false value, which means that the full screen display mode or the swap chain back buffer is not stereo
- The `SampleDesc` property describes the multisampling parameters
- The `BufferUsage` property describes the surface usage and the CPU access options for the back buffer
- The `BufferCount` property indicates the number of buffers in the swap chain
- The `Scaling` property identifies the resize behavior if the size of the back buffer is not equal to the target output
- The `DXGI_SWAP_EFFECT` property typed value uses the options for handling the pixels in a display surface
- The `DXGI_ALPHA_MODEDXGI_ALPHA_MODE` property typed value identifies the transparency behavior of the swap chain back buffer

- A combination of `DXGI_SWAP_CHAIN_FLAGDXGI_SWAP_CHAIN_FLAG` typed values, which are combined by using a bitwise `OR` operation

Now let's get back to the previous code:

```
else
{
  DXGI_SWAP_CHAIN_DESC1 swapChainDesc = {0};
  swapChainDesc.Width =
    static_cast<UINT>(renderTargetSize.Width);
  swapChainDesc.Height =
    static_cast<UINT>(renderTargetSize.Height);
  swapChainDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
  swapChainDesc.Stereo = false;
  swapChainDesc.SampleDesc.Count = 1;
  swapChainDesc.SampleDesc.Quality = 0;
  swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
  swapChainDesc.BufferCount = 2;
  swapChainDesc.Scaling = DXGI_SCALING_NONE;
  swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
  swapChainDesc.Flags = 0;
```

The primary goal of **Microsoft DirectX Graphics Infrastructure** (**DXGI**) is to manage low-level tasks that can be independent of the DirectX graphics runtime. DXGI provides a common framework for future graphics components which is used to present frames to a window, monitor, or read the contents on a monitor.

```
    ComPtr<IDXGIDevice1>  dxgiDevice;
    hr = d3dDevice.As(&dxgiDevice);
    ComPtr<IDXGIAdapter> dxgiAdapter;
    hr = dxgiDevice->GetAdapter(&dxgiAdapter);
    ComPtr<IDXGIFactory2> dxgiFactory;
    hr = dxgiAdapter->GetParent(__uuidof(IDXGIFactory2), &dxgiFactory)

//__uuidof extracts the GUID attached to an object.

    CoreWindow^ window = coreWindow.Get();
    hr = dxgiFactory->CreateSwapChainForCoreWindow(
      d3dDevice.Get(),
      reinterpret_cast<IUnknown*>(window),
      &swapChainDesc, nullptr, &swapChain);
    hr = dxgiDevice->SetMaximumFrameLatency(3);
}
```

`SetMaximumFrameLatency` helps you control the latency of an application loop for the process of updating. Actually, frame latency is the number of frames that are allowed to be stored in a queue while the CPU is responding to user input before submission for rendering. In DirectX 11.1, the value defaults to 3, but it can range from 1 to 16.

# The render target and depth stencil views

The render target view allows you to render the scene to a texture resource. You can then apply a shader to the texture resource before displaying it. When the back buffer is ready, the `CreateRenderTargetView` method is used to create a render target view.

The last part creates the render target view, the depth stencil, and the viewport of Direct3D. Use the back buffer to create the render target view; also, you can define the render target view description or send `nullptr` to use the default value.

The properties of the render target view description are as follows:

- The `Format` property specifies the format type of resource data
- The `ViewDimension` property identifies the type of resource that will be viewed as a render target
- The `Buffer` property specifies how the render target resource will access `Texture1D`, `Texture1DArray`, `Texture2D`, `Texture2DArray`, `Texture2DMS`, `Texture2DMSArray`, and `Texture3D`, which are used to specify the subresources.

First we get the back buffer from the swap chain, and then we will create the render target view from this back buffer as shown in the following code:

```
ComPtr<ID3D11Texture2D> backBuffer;
  hr = swapChain->GetBuffer(0, __uuidof(ID3D11Texture2D),
&backBuffer);
  hr = d3dDevice->CreateRenderTargetView(backBuffer.Get(), nullptr,
&renderTargetView);
```

The depth stencil view is a 2D buffer that controls how the depth and stencil buffers are used; it is similar to a render target view, but the difference is that it represents the depth and stencil buffers. The depth buffer stores z data of each pixel as a floating point while the stencil buffer contains integer data of each pixel. As mentioned before, the depth stencil buffer is a 2D buffer, hence the buffer has a 2D texture.

`CD3D11_TEXTURE2D_DESC` inherits from `D3D11_TEXTURE2D_DESC` and is used to create a depth stencil description (`CD3D11_DEPTH_STENCIL_VIEW_DESC`) which specifies how to access a resource used in a depth stencil view.

```
CD3D11_TEXTURE2D_DESC depthStencilDesc(
  DXGI_FORMAT_D24_UNORM_S8_UINT,
  static_cast<UINT>(renderTargetSize.Width),
  static_cast<UINT>(renderTargetSize.Height),
  1,
  1,
  D3D11_BIND_DEPTH_STENCIL);
ComPtr<ID3D11Texture2D> depthStencil;
```

```
    hr = d3dDevice->CreateTexture2D(&depthStencilDesc, nullptr,
      &depthStencil);
    CD3D11_DEPTH_STENCIL_VIEW_DESC
      depthStencilViewDesc(D3D11_DSV_DIMENSION_TEXTURE2D);
    hr = d3dDevice->CreateDepthStencilView(depthStencil.Get(),
      &depthStencilViewDesc, &depthStencilView );
    CD3D11_VIEWPORT viewport(
      0.0f,
      0.0f,
      renderTargetSize.Width,
      renderTargetSize.Height);
    d3dContext->RSSetViewports(1, &viewport);
  }
```

For creating the viewport, you must specify the top, the left, and the size of the screen.

Finally, the `Present` method is used for presenting `DXGI_ PRESENT_PARAMETERS`. It then discards the render target and depth stencil views. Furthermore, it will be responsible to check whether the video card has been physically removed, a driver upgrade for the video card has been launched, or if there is something wrong with the drivers of the video card. If any of these have happened, it should destroy and recreate the device. As the `HandleDeviceLost` method is responsible for managing it, it is simply a reinitialize device.

So, we have introduced how to initialize the Direct3D device. The `SimpleScene` class shows you how to clear the screen using the functions of the `Game` class. Note that the `SimpleScene` class must inherit from the `Game` class.

# Summary

In this chapter, we have introduced the new features of Windows 8, DirectX 11.1, and the new extension of C++, which is called C++/CX. Also, we have set up our first project, added time to our new framework, and initialized the Direct3D device.

In the next chapter, we will cover the new features of HLSL in DirectX 11.1, buffers, and the compiled shader objects. Finally, we will outline the new additions of Direct3D 1.1.

# 2
# Getting Started with HLSL

This chapter is about HLSL, the GPU programming language for DirectX, which is a high-level shader language that is used to generate shaders.

In this chapter we will cover the following topics:

- An introduction to HLSL
- New features of HLSL
- Compiling and linking to shaders
- Buffers in Direct3D
- Rendering primitives
- Direct2D 1.1

By the end of this chapter, you will have a basic knowledge of what the new features of HLSL in DirectX 11.1 are, and you will learn how to interact with the shaders, the buffers, and the primitives; finally, you will be introduced to the new additions of Direct2D for Windows 8.
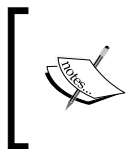
> *Doron Feinstein* explained the power of HLSL in his book, *HLSL Development Cookbook*; you can get more info about this book at `http://www.packtpub.com/high-level-shader-language-development-cookbook/book`.

# An introduction to HLSL

This section contains an overview of **high-level shading language** (**HLSL**), which was developed by Microsoft for programming the Direct3D pipeline.

HLSL supports many different intrinsic data types, such as `Buffer`, `Scalar`, `Vector`, `Matrix`, `Sampler`, `Shader`, `Texture`, `Struct`, and a few user-defined data types. HLSL syntax is similar to C; preprocessor directives are supported and you can use macros definitions, conditional compilation, and `include` statements; also, the basic types such as `float`, `int`, `uint`, and `bool` can be used.

Unlike C, it does not support pointers. This means that the language does not support a dynamic allocation mode, but there are additional data types such as `float2`, `float3`, and `float4` to help maximize the performance of 4-component vectors that use a matrix math to operate on the 3D graphics data. If you want the shader to run efficiently on all hardware, just make sure it is properly vectorized. The current high-end generations of hardware from NVIDIA as well as AMD are based on a scalar architecture, so vectorization for the current hardware is not that important, but all AMD **accelerated processing unit** (**APU**) microprocessors are vector-based.

As you may know, the shader model 5 was introduced in DirectX 11 with some new features, such as the compute shader, multithreading, dynamic shader linking, and tessellation:

- **Compute shaders**: These are also known as **Direct Compute**, and they allow the GPU to be used as a general-purpose parallel processor. This topic will be the outline of *Chapter 5*, *Multithreading*.

- **Multithreading**: The DirectX 11 API has been improved to enable developers to use an efficient interaction between the multiple CPU cores and the GPU.

- **Dynamic shader linking**: The shader model 5 introduced an object-oriented language with dynamic shader linking. It allows you to set which code path must be run at runtime.

- **Tessellation**: This is implemented on the GPU and can be used to increase or decrease the triangle count in order to have a variety of details. This topic is the outline of *Chapter 4*, *Tessellation*.

Now we have introduced these features, let's look at the most important features of HLSL that were released with DirectX 11.1.

# New features of HLSL

Open the first project from the source code of this chapter, which is named `Minimum precision`. The following functionalities have been added to HLSL and help developers use less memory bandwidth in Windows 8, Windows RT, and Windows Server 2012. We can use the `ID3D11Device::CheckFeatureSupport` function to check whether the feature is supported by your graphic drivers or not:

- **HLSL minimum precision**: By using any precision greater than or equal to their specified bit precision, we can save memory bandwidth and system power. You can query for this feature using the `ID3D11Device::CheckFea tureSupport` function; the resultant value is of type `HRESULT`. If it is `S_OK`, it means the graphics hardware can perform the HLSL operations at a lower precision, but does not guarantee that the graphics hardware will actually run at a lower precision. In other words, the graphics hardware can ignore this and run at full 32-bit precision. Please note that according to MSDN, this feature is not supported on a WARP device (`http://msdn.microsoft.com/en-us/library/windows/desktop/hh968108(v=vs.85).aspx`).

  Pay attention to the following `SimpleScene.cpp` file; in the `Load` method, we checked the HLSL minimum precision:

```
D3D11_FEATURE_DATA_SHADER_MIN_PRECISION_SUPPORT
min_precision;
hr = GDevice.d3dDevice->CheckFeatureSupport(
            D3D11_FEATURE_SHADER_MIN_PRECISION_SUPPORT,
&min_precision,
sizeof(min_precision));
if (hr == S_OK){
// minimum precision is supported
}

cbufferObjectInfo : register(b0)
{
min16float4x4WorldViewProjection;
};
```

- **Clip planes**: In DirectX 11.1, you can specify the user clip planes in HLSL on feature level 9 or higher; you can use the `clip planes` function attribute in a HLSL function declaration rather than `SV_ClipDistance`, `SV_ClipDistance`, or `Clip distance data`, which is a signed `float32` distance value to the plane. This feature helps you make the shader work on level 9_x as well as on the feature level 10 or higher. Refer to `http://msdn.microsoft.com/en-us/library/windows/desktop/jj635733(v=vs.85).aspx`.

- **Constant buffers**: DirectX 11.1 lets us create larger constant buffers than a shader can access. As you may know, the constant buffer helps reduce the bandwidth of updating. The maximum size of a constant buffer is 64 KB. When you use `PSSetConstantBuffers1` or `PSSetConstantBuffers` to bind to the constant buffer, it can be defined as a range of buffer that the shader can access and also fits within the 4096 constant limit. You can also bind to the subrange of a constant buffer and specify which one can be used by the shader. The `ID3D11DeviceContext1::VSSetConstantBuffers1` and `ID3D11 DeviceContext1::VSGetConstantBuffers1` functions are used to set or get the subrange of a constant buffer that is bound to the shader.

- **UAVs**: DirectX 11.1 supports a larger number of **Unordered Access Views** (**UAV**s). This feature gives us an efficient way to store and use resources in our shaders. Previously, in shader model 5, a UAV could only be used in the Pixel Shader and the compute shader, but in DirectX 11.1, it can be used in all pipeline stages. We will introduce this feature in *Chapter 5*, *Multithreading*.

# Compiling and linking to shaders

It's time to write our first shader and introduce how we will employ it in the DirectX application. Open the second project of this chapter from the source code (`Shaders`). Open the `VertexShader.hlsl` file from `Assets/Shaders`; note that in Visual Studio, you can add a shader to the project by going to `Add/New Item/HLSL`.

Visual Studio compiles the HLSL files and converts them to `.cso`, which is usually generated in the `Debug` folder (under `ProjectName` in the `Projects` folder) by default. You can change the output directory from the HLSL **Property** page.

In DirectX 11.1, you can use two ways to compile the shaders. The first and traditional way is to compile the FX files at runtime; we use the `D3DCompileFromFile` function to compile and build the shader. You can use this API to develop your Windows Store apps, but you can't use it during the runtime of the Windows Store application. The other way is to use the offline compiled shader objects (`.cso`).

The `FXC.EXE` tool is now integrated into the build environment of Visual Studio 2012 and the Windows 8.0 SDK. It can compile and build them to the HLSL and FX files and convert to the `.cso` format.

Leave the code of the HLSL files for the moment; we will check them later. Because we need to load a type of `.cso` as a stream and convert them to bytes, we need to define an inline function in `pch.h`.

```
inline Array<byte>^ ReadFile(String^ path)
{
Array<byte>^ bytes = nullptr;
FILE* f = nullptr;
_wfopen_s(&f, path->Data(), L"rb");
if (f == nullptr){
//throw Exception
}
else{
fseek(f, 0, SEEK_END);
auto pos = ftell(f);
bytes = ref new Array<byte>(pos);
fseek(f, 0, SEEK_SET);
if (pos > 0)    fread(&bytes[0], 1, pos, f);
fclose(f);
}
return bytes;
}
```

This method gets the target file's path; open the file using the `_wfopen_s` function and then create an array of bytes from the stream of the file. Now open the `shader.h` header from the `Graphics\Shaders` folder of the solution. The functions and the members of the `shader` class are declared here, but the main method that is used for loading a shader is defined as the `static` function. Let's check what is going on in the `LoadShader` method:

```
static void LoadShader(String^ path, ShaderType shaderType,
VertexDeclaration VDeclaration, _Inout_ Shader^ &shader)
{
auto bytes = DX::ReadFile(path);
switch (shaderType)
{
caseShaderType::VertexShader:
shader->CreateVertexShader(bytes->Data,
bytes->Length, VDeclaration);
break;
caseShaderType::PixelShader:
```

```
shader->CreatePixelShader(bytes->Data,
bytes->Length);
break;
default:
throw ref new Exception(0, "Unknown shader type");
}
}
```

The `ShaderType` parameter is an enumeration declaration that is defined for handling what type of shader is going to be loaded. The following diagram shows the programmable runtime that is designed by the 3D real-time applications. In this chapter, we just use the Vertex and Pixel Shaders; we will introduce the other stages of the pipeline in the following chapters:
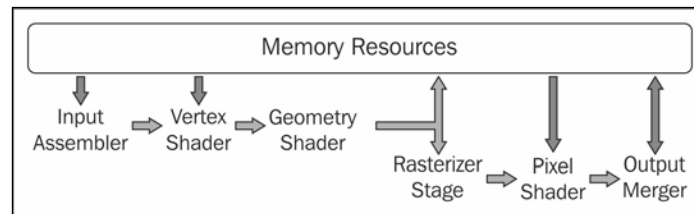


Typically, **Memory Resources** are the buffers, textures, and constant buffers, and all of these types are used as resources in the shader model.

The first stage is the **Input Assembler**, which is responsible for supplying the input data of the Vertex Shader. In our example, the vertex type is defined by a unique structure. For example, in our framework, if the `VertexDeclaration` class is set to `VertexPositionNormalTexture`, it means that each vertex consists of the position, the normal vector, and the texture coordinate of the vertex.

The next stage is the **Vertex Shader**, which is invoked for each vertex. It takes a single vertex and produces the specific output for the inputted vertex.

Take a look at the following code of `VertexShader.hlsl`, which is located in the `Assets/Shaders` folder:

```
#include "Parameters.hlsli"
PS_In main( VS_In IN )
{
PS_In OUT = (PS_In)0;
OUT.pos = mul(float4(IN.pos, 1.0), WorldViewProjection);
OUT.uv = IN.uv;
return OUT;
}
```

A Vertex Shader is a graphics-processing function used to add special effects to objects in a 3D environment by performing mathematical operations on the object's vertices data. It can manipulate properties such as position, color, texture coordinate, and the tangent of the vertex. In our Vertex Shader, we simply transform each vertex from local space to the homogeneous clip space and pass the texture coordinate to the next stage.

The **Geometry Shader** provides us with the ability to create or destroy primitives; it is processed on all primitives, a triangle, or a line. This stage is slow, especially when it runs on a tablet; therefore, we are not going to cover it in this book.

The **Rasterizer Stage** is responsible for clipping and preparing the primitives for the Pixel Shader stage.

The **Pixel Shader** stage enables the rich shading techniques per pixel of data.

The following is the `PixelShader.hlsl` file. This stage samples the 2D texture and returns RGB as the color for each pixel's data:

```
Texture2D Texture : register(t0);
SamplerState Sampler : register(s0);
float4 main(PS_In IN) : SV_TARGET
{
return Texture.Sample(Sampler, IN.uv);
}
```

We will introduce the textures and sampler state in the following section. Now take a look at the `shader.cpp` file, which is located in the same folder. We must use the `ID3D11Device::CreatePixelShader` method to create the Pixel Shader from its bytes. The input arguments of this function are the shader bytes, the length of these bytes, and the class linkage interface. The output parameter of this function is an address to the pointer of the Pixel Shader interface.

```
GDevice.d3dDevice->CreatePixelShader(bytes, Length, nullptr,
&pShader);
```

Creating the Vertex Shader is similar to creating the Pixel Shader; we must call the `ID3D11Device::CreateVertexShader` method to create a Vertex Shader.

As mentioned earlier, we need to specify the input assembler for the Vertex Shader, so first we must define the vertex declaration as shown in the following code:

```
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,
D3D11_APPEND_ALIGNED_ELEMENT,
D3D11_INPUT_PER_VERTEX_DATA, 0
},
{"TEXCOORD", 0,DXGI_FORMAT_R32G32_FLOAT, 0,
D3D11_APPEND_ALIGNED_ELEMENT,
  D3D11_INPUT_PER_VERTEX_DATA, 0
},
};
```

The code indicates that the structure of the inputted vertex which is defined in
`Parameters.hlsli` is like the following code:

```
struct VS_In
{
float3 pos     : POSITION0;
float3 normal : NORMAL0;
float2 uv      : TEXCOORD0;
};
```

Once an input layout description has been set, the
`ID3D11Device::CreateInputLayout` method represents it for the Vertex Shader.
The input parameters of this function are the vertex description, length of vertex
description, bytes of the Vertex Shader, and the length of these bytes; also, the output
parameter is a pointer to the input layout object.

```
void Shader::Apply()
{
GDevice.d3dContext->IASetInputLayout(this->inputLayout.Get());
GDevice.d3dContext->VSSetShader(this->vShader.Get(), nullptr,
0);
GDevice.d3dContext->PSSetShader(this->pShader.Get(), nullptr,
0);
}
```

We used the `Apply` method for setting the input layout, Vertex Shader, and the Pixel
Shader for the device.

Now we can simply load the shader in our framework as shown in the following code:

```
autoshader =ref new Shader();
Shader::LoadShader("VertexShader.cso",
ShaderType::VertexShader,
VertexDeclaration::PositionNormalTexture,
shader);
```

# Buffers in Direct3D

In the previous section, we used the `shader` class in order to load the HLSL file. This section describes what the functionality of each buffer is and how the stored data will be accessed through the rendering pipeline.

## Constant buffers

A constant buffer is just a block of data that allows you to store the shader variables. Each element of a constant buffer stores one to four component vectors. In our example, the preceding code of `Parameters.hlsli` was the following:

```
cbuffer ObjectInfo : register(b0)
{
min16float4x4 WorldViewProjection;
};
```

The code defines a constant buffer in the `b0` bank. The element of `cbuffer` is a 4x4 matrix with minimum 16-bit precision. The `WorldViewProjection` matrix is responsible for transforming each vertex from the local space to the homogeneous clip space.

If we assume that the camera is idle in our scene, we would like to use our shader to update the constant buffer in one frame at least. So, we need a way to bind our `shader` class to our constant buffer.

> Constant buffers help to reduce the bandwidth of updating. For example, assume we have four variables, `view`, `projection`, `light position`, and `light direction`. Because `view` and `projection` might change in each frame and the data of `light position` and `light direction` might not, we declare them in different constant buffers to decide to update the necessary variables.

Open the `CBuffer.h` header from the `Graphics/Shaders/` folder of our solution. Actually, the constant buffer is a pointer to the `ID3D11Buffer` interface. To create constant buffers, call the `ID3D11Device::CreateBuffer` method. The input parameters of this function are the description of the buffer's structure and a pointer to the subresource data, and the output parameter is the address of the pointer to the constant buffer.

```
D3D11_BUFFER_DESC bufferDesc;
ZeroMemory(&bufferDesc, sizeof(D3D11_BUFFER_DESC));
bufferDesc.ByteWidth = sizeof(T);
```

```
bufferDesc.Usage = D3D11_USAGE_DEFAULT;
bufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
hr = GDevice.d3dDevice->CreateBuffer(
&bufferDesc, nullptr, &buffer);
```

The final step is to update this buffer with our shader constants. The `ID3D11Devic eContext::UpdateSubresource` method forces the CPU to copy the data from the memory to our constant buffer.

# Vertex buffers

Prior to drawing primitives, they need to be prepared by copying them into the GPU. The buffer that is used to copy vertices to the GPU is called the vertex buffer. Just like the constant buffer, to create the vertex buffer, we need to call `ID3D11Device::CreateBuffer`, but we must always have the `D3D11_BIND_ VERTEX_ BUFFER` bind flag set. If you would like to stream the vertex data to the buffer, you should add another bind flag (`D3D11_BIND_STREAM_OUTPUT`) to the vertex buffer description; on the other hand, the buffer that will be updated by the GPU streaming vertex data must be created with the `D3D11_USAGE_DEFAULT` usage flag.

If you would like to update the buffer frequently in each frame, use the `D3D11_ USAGE_DYNAMIC` flag; also, the `D3D11_USAGE_IMMUTABLE` usage flag is used for the static vertex buffer, which will not be modified after initializing the vertex buffer. For example, use this flag when you want to create static terrain.

Open the `Quad.cpp` class from the `Graphics/Models` folder of this project and take a look at the code of the `Load` method, which is required to create a vertex buffer and an index buffer.

# Index buffers

The index buffers are the memory data that store the list of indices that are integer offset to the vertices. Typically, the main purpose of index buffers is to provide a significant reduction in the total number of vertices, but you can still draw vertices with only the vertex buffer.

To create the index buffer, you need to create a buffer description with the `D3D11_ BIND_INDEX_BUFFER` bind flag. The following code shows the buffer description of the index buffer:

```
D3D11_BUFFER_DESC indexBufferDesc;
indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
```

```
indexBufferDesc.ByteWidth = sizeof(unsigned short) *
ARRAYSIZE(Indices);
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;
indexBufferDesc.CPUAccessFlags = 0;
indexBufferDesc.MiscFlags = 0;
indexBufferDesc.StructureByteStride = 0;
```

The parameters are as follows:

- The usage flag

- The size of the buffer in bytes

- A flag, which specifies how to use and bind to the buffer

- A flag, which specifies how the CPU can access the buffer

- Miscellaneous, and the size of each element in the buffer

# Textures

Textures are a fundamental part of graphical applications. Actually, they are buffers that can be filtered by the texture samplers and read by the shader units. Also, there are a few types of textures such as 1D, 2D, 3D, cube maps, and the texture resource arrays.

One-dimensional textures (1D) represent textures along one axis. 2D textures are used as the primary texture type in Direct3D since they have two dimensions; they are similar to 2D images.

A 2D texture is created by calling the `ID3D11Device::CreateTexture2D` method, in which the input parameters are the pointer to the `D3D11_TEXTURE2D_DESC` structure and the initial subresource data and the output parameter is addressed to the interface of the 2D texture.

```
struct DXGI_SAMPLE_DESC {
  UINT Count;
  UINT Quality;
};
struct D3D11_TEXTURE2D_DESC {
  UINT Width;
  UINT Height;
  UINT MipLevels;
  UINT ArraySize;
  DXGI_FORMAT Format;
  DXGI_SAMPLE_DESC SampleDesc; D3D11_USAGE Usage;
  UINT BindFlags;
  UINT CPUAccessFlags;
  UINT MiscFlags;
};
```

The structure of a 3D texture (also known as a **volume texture**) is similar to 1D and 2D except for the third dimension, which contains the depth parameters. There is no concept of a 3D texture array, but we can have a 1D texture array and a 2D texture array (the texture cube).

Each texture type can contain mipmap levels. The mipmap improves the quality of the rendered textures by saving the memory bandwidth. It simply represents a lower resolution version of a texture. The lower resolution mipmap image is used when the object appears far away, and a high resolution mipmap image is used for closer objects.

> In this book, we used `DDSTextureLoader` to load the DDS texture type (`http://directxtex.codeplex.com/wikipag e?title=DDSTextureLoader`); there are many existing open source libraries such as *DevIL* (`http://openil.sourceforge. net/`) that can help you load the textures.

In this section, we are going to learn how to load a 2D texture and wrap it in polygons. Open the `Texture2D.h` header file from the `Graphics/Textures` folder of the solution. Like a shader, we have a static method (`LoadTexture`) that is used for loading the texture. In this function, we call the `CreateDDSTextureFromMemory` method of the `DDSTextureLoader` class. The input parameters are the Direct3D device, the bytes of the texture, and the length of bytes. The output parameters are the address of the pointer to the resource interface and the address of the pointer to the shader resource view interface.

```
CreateDDSTextureFromMemory(device,
textureData->Data,
textureData->Length,
nullptr,&rsv);
```

To use this texture resource data in HLSL, we need to sample the texture resource. We must define `sampler state`, which allows us to modify how the pixels are written to the polygon face; in other words, how the texture data is sampled by the addressing modes, filtering, and the level of details.

It must be defined once per application in the `Game.cpp` class file after initializing the Direct3D context. Take a look at the declaration of the `GraphicsDevice` class in `pch.h`; we stored the sampler in the `GDevice.Samplers` variable. Our sampler state uses a linear filtering and applies the wrap texture addresses on boundaries. See the behavior of the `CreateSampler` method that is defined in the `Texture2D.h` header.

> For better compression, make use of the DDS files; download and install *NVIDIA Texture Tools for Adobe Photoshop* from `https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop`. You can also load and create the DDS files from the Microsoft DirectX Texture Tool (SDK, June 2010) or Visual Studio 2012.

Now we can simply load the texture in our framework as shown in the following code:

```
auto texture2D = ref new Texture2D();
Texture2D::LoadTexture(L"Assets/Textures/Persepolis.dds",
texture2D);
```

# Rendering primitives

The final step is to bring all of them together and use the texture and the shader to render the primitives. Open the `Quad.h` header file from the `Graphics/Models` folder of our project.

This class contains a constant buffer, a shader, and a texture2D; also, we declared the vertex buffer and the index buffer for rendering our primitives. As mentioned earlier, to render the primitives, we need to copy the vertex buffer and index buffer (if needed) to the GPU. The `Render` method of the `Quad.cpp` file works in the following way:

```
auto world = XMMatrixIdentity();
auto view = Camera.View;
auto projection = Camera.Projection;
cBuffer.Const.WorldViewProjection =
XMMatrixTranspose(world * view * projection);
cBuffer.Update();
shader->SetConstantBuffer(0, 1, cBuffer.GetBuffer());
```

First, we combine the `world`, `view` and `projection` matrices and assign this matrix to our constant buffer, which is defined as the first slot (zero is based on indexing for the slots).

```
shader->SetTexture2D(0, 1, texture2D);
```

Then assign the texture to the first slot of the textures. Inside this method, we need to set the shader resource view and the sampler to the Pixel Shader.

```
shader->Apply();{
GDevice.d3dContext->IASetVertexBuffers(0, 1,
vertexBuffer.GetAddressOf(),
&stride, &offset );
GDevice.d3dContext->IASetIndexBuffer(
indexBuffer.Get(), DXGI_FORMAT_R16_UINT, 0 );
GDevice.d3dContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
GDevice.d3dContext->DrawIndexed(indicesSize, 0, 0 );
}
```

Call the `Apply` shader and then set the vertex buffer and the index buffer to the GPU. The data enters the graphics pipeline to a stream of primitives; these primitives help us draw some custom shapes on the screen; before drawing shapes, we need to specify the topology of a primitive and set it to the input assembler.

The `ID3D11DeviceContext::IASetPrimitiveTopology` method sets the type of primitive and it's ordering. The following subsection describes different primitives' topologies:

- **Point List**: This topology interprets the vertex data as a list of points.
- **Line List**: This topology interprets that every two vertices make a line. The produced lines might be disconnected from each other.
- **Line Strip**: This topology represents the connected list of lines. The produced lines are connected to each other.
- **Triangle List**: With this topology, every three vertices draw a unique triangle; please note that the individual triangles are disconnected from each other.
- **Triangle Strip**: This topology behaves like the triangle list, assuming that the triangles are connected to each other.

Now build our `Shader` project from the source code of this chapter and you will see a quad with a warped 2D texture as shown in the following screenshot:



# Direct2D 1.1

The Direct2D is an API that provides the ability to perform high-quality 2D graphics rendering. An application that uses Direct2D for graphics can deliver a higher visual quality than what can be achieved by using GDI.

One of the major new features of Windows 8 is Direct2D. In the new version of Direct2D (Version 1.1), the new API does not change fundamentally, but grows up and becomes more compatible; as you may already know, the old version (which was launched with Windows 7) was only compatible with DirectX 10, but now it is one of the members of the DirectX 11.1 family and is built on top of Direct3D 11.1.

Open the final project of this chapter (that is, `Direct2D`); we are going to demonstrate how to initialize and use Direct2D within the framework:

1. First include the Direct2D headers and link the libraries to the framework. These headers and libraries were added at the top of `pch.h`.

2. Call the `D2D1CreateFactory` method to create `ID2D1Factory`. This object is a factory object that is used to create Direct2D resources. The first parameter is `D2D1_FACTORY_TYPE`, which specifies whether our factory and resources are used in a multithread or single-thread state. The other input parameters are a reference to the IID of `ID2D1Factory` and the factory options. The output parameter is the address to a pointer to the factory interface. When the factory is initialized, we can create the Direct2D device and the Direct2D context.

```
GDevice.factory->CreateDevice(dxgiDevice.Get(),
&GDevice.d2dDevice);
GDevice.d2dDevice->CreateDeviceContext(
D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
&GDevice.d2dContext);
```

3. If you would like to use a font loader and cached font data with Direct2D, we will have to create an interface to the `DirectWrite` factory object using the `DWriteCreateFactory` function. According to MSDN, it is recommended that you use a shared factory object because it allows multiple components that use `DirectWrite` to share internal `DirectWrite` state data, and thereby reduces the memory usage (`http://msdn.microsoft.com/en-us/library/windows/desktop/dd368040(v=vs.85).aspx`).

4. If you prefer to show the images with Direct2D, you should encode this image to the `IWICBitmapEncoder` object. Take a look at the `Load` method of `SimpleScene.cpp`; we decode our image file with `IWICImagingFactory::CreateDecoderFromFilename` and then create an interface to the format converter, `IWICFormatConverter`, and convert the bitmap to the **Windows Imaging Component** (**WIC**) object by calling the `Initialize` method of this converter.

5. Before using the Direct2D command, we need to set the target for the Direct2D device context. This target must be a type of Direct2D image. The target can be either a bitmap created with the `D2D1_BITMAP_OPTIONS_TARGET` flag or it can be a command list. You cannot set the output of an effect as the target.

6. You can use Direct2D effects to take advantage of the GPU features for image processing and to create and apply a high-quality effect to the images. When we load a WIC object from the loaded image, we must generate `ID2D1Image` from `IWICBitmapSource`, which will be used as an input in an effect. The effect type that is used in this example is Gaussian blur, which can be specified with `CLSID_D2D1GaussianBlur`. Make sure you use the 2D effect enumeration and link the `dxguid` library to your project. The `Load` and `SetWICBitmapSource` methods of `SpriteBatch.cpp` do this process in our framework. Please note that `SpriteBatch.cpp` is located in the `Graphics` folder of the solution.

```
void SpriteBatch::Load()
{
GDevice.d2dContext->CreateEffect(
 CLSID_D2D1BitmapSource, &bitmapSourceEffect);
GDevice.d2dContext->CreateEffect(CLSID_D2D1GaussianBlur,
&gaussianBlurEffect);
}
voidSpriteBatch::SetWICBitmapSource(
ComPtr<IWICFormatConverter>wicConverter)
{
bitmapSourceEffect->SetValue(
  D2D1_BITMAPSOURCE_PROP_WIC_BITMAP_SOURCE,
wicConverter.Get());
gaussianBlurEffect->SetInputEffect(0,
bitmapSourceEffect.Get());
}
```

7. The main methods of `Spritebatch` are `Begin` and `End`. Before starting to draw, `Spritebatch` stores the state of our drawing; because of this, `Begin` must be called successfully before a drawing method can be called, and also `Begin` cannot be called again until `End` has been successfully called.

```
BEGIN;
{
2D DRAWING;
}
END;
```

8. When `Begin` is called, the `SpriteBatch` class stores the drawing state into a state block and then the 2D context begins drawing; after `End` is called, `SpriteBatch` restores the drawing state. The `ShowString` function uses the `write` factory object to create a text layout and then uses `ID3D11DeviceCont ext1::DrawTextLayout` to draw the text. The `DrawImage` function is used to draw an image using the specific effect.

```
voidSpriteBatch::DrawImage(XMFLOAT2 Position)
{
if (state != STARTED) throw ref new Exception
(0, MISSING_BEGIN_CALL);
GDevice.d2dContext->DrawImage(gaussianBlurEffect.Get(),
Point2F(Position.x, Position.y));
}
```

As mentioned earlier, in DirectX 11.1, we can create a multithreaded Direct2D factory. The resources can be used from multiple threads. In order to prevent resource access conflicts, the `ID2D1Multithread` interface plays the role of the locking mechanism from a Direct2D factory. The `ID2D1Multithread::Enter` method enters the critical section and the `ID2D1Multithread::Leave` method leaves this critical section (if the section exists). In the `Render` method of `SimpleScene.cpp`, we used two threads for rendering asynchronously: the first one draws the primitives with Direct3D and shows some text on the screen with Direct2D; the other one draws images on the screen with Direct2D. To prevent a deadlock, make sure to pair up calls to the `Enter` and `Leave` methods as follows:

```
D2DMultithread->Enter();
{
// Now it is safe to make Direct3D/DXGI calls
D2DMultithread->Leave();
}
```

# Summary

In this chapter, we learned about the new features of HLSL in DirectX 11.1, how to use buffers and textures in DirectX 11.1, and how to render the geometries with the compiled HLSL. We were also introduced to the new features of Direct2D 1.1, which was released with DirectX 11.1.

The next chapter is going to lay out the 3D scene aspects that contain more important features for all game or graphical applications.

# 3
# Rendering a 3D Scene

This may be the most favorable chapter in this book. We are going to lay out the process of developing our framework in order to render 3D scenes. We're going to take you on a tour of some of the advanced concepts of the 3D programming, specifically focusing on system usages, loading and rendering meshes, handling inputs, types of cameras, and finally integrating XAML and Direct3D in order to have an editor for the game.

This chapter covers the following:

- Displaying the performance data
- Asynchronous resource loading
- Getting started with the Model Editor
- Loading a model from the `.cmo` format
- Rendering a model
- The input devices we'll need
  - Keyboard
  - Pointer
  - Xbox 360 controllers
- Turning on the camera
  - Base camera
  - First person camera
  - Third person camera
- Composing XAML and Direct3D

By the end of this chapter, you will have all the knowledge you'll need prior to programming 3D scenes.

# Displaying the performance data

We have studied the fundamentals of Direct3D 11.1 throughout the previous chapters. It is now time to develop our framework supporting more aspects of the 3D programming. Let's start with adding a frame per second.

# A short introduction to FPS

**FPS** is known as **frame rate** or **frames per second**, and it refers to the number of frames being rendered in one second; a higher frame rate is usually better since this results in less latency. The standard value for most games is some value between 30 and 60 FPS, but could be above 100 FPS. To measure FPS, we must simply update our counter in order to count the frames that occur in one second. So, we need a variable for counting frame updates; also, we need a timer to measure the time between each update.

Open the `Display performance data` project from the source code of this chapter and then open the `FPS.cpp` file to see its definition as shown in the following code:

```
Fps::Fps(Graphics2D G2D) :_fps(0), elapsedTime(0)
{
  auto hr = GDevice.d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Red), &this-
>RedSolidBrushColor);
  hr = GDevice.d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Yellow), &this-
>YellowSolidBrushColor);
  hr = GDevice.d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Lime), &this-
>LimeSolidBrushColor);
}
void Fps::Update(float ElapsedTime)
{
  this->elapsedTime += ElapsedTime;
  this->frames++;
  if (this->elapsedTime > 1.0f)
  {
    this->_fps = (float)this->frames / this->elapsedTime;
    this->elapsedTime = 0;
    this->frames = 0;
    if (this->_fps < 20.0f) {  state = State::Error; }
    else if (this->_fps < 35.0f) { state = State::Warning;  }
    else {  state = State::Normal; }
  }
}
```

The `frames` variable is responsible for storing a number of frames. When the elapsed time reaches one second, we must store the number of frames in `fps`, initialize the variables, and make them ready to start counting in the next period. By default, we draw FPS with a lime color but if the FPS is less than 20, we use the red color to show it on the screen, which means your graphic card cannot reach the standard time in each period.

The `CpuInfo` class is used to get information of the CPU as shown in the following code:

```
CpuInfo:: CpuInfo ():totalCores(0)
{
    this->totalCores = 0;
  auto filter = "System.Devices.InterfaceClassGuid:=\"" +
GUID_DEVICE_PROCESSOR + "\"";

Concurrency::task<DeviceInformationCollection^>(DeviceInformation:
:FindAllAsync(filter, nullptr))
  then([this](DeviceInformationCollection^ interfaces)
  {
    std::for_each(begin(interfaces), end(interfaces),
      [this](DeviceInformation^ DeviceInfo)
    {
      this->totalCores++;
      if (this->name->IsEmpty())
      {
        this->name = DeviceInfo->Name;
      }
    });
  });
}
```

`DeviceInformation::FindAllAsync` allows us to access information that pertains to the whole device hardware product. To get information of the CPU, we must pass `guid` (`97FADB10-4E33-40AE-359C-8BEF029DBDD0`) as an input parameter in this method.

In the final step, we use the `Perf.cpp` class that manages our drawing classes as shown in the following code:

```
void Perf::ShowStatus(SpriteBatch^ spriteBatch, SpriteFont^
spriteFont, DirectX::XMFLOAT2* Position)
{
  const float YPlus = 20;
```

```
   spriteBatch->ShowString("FPS : " + _FPS->fps.ToString(), Position,
_FPS->FpsColor, spriteFont, Matrix3x2F::Identity());
   Position->y += YPlus;
   spriteBatch->ShowString("Elapsed Time: " + deltaTime.ToString(),
Position, spriteFont);
   Position->y += YPlus;
   spriteBatch->ShowString("Total Time: " + totalTime.ToString(),
Position, spriteFont);
   Position->y += YPlus;
   spriteBatch->ShowString("CPU Core(s) : " + _CPU->TotalCores.
ToString(), Position, spriteFont);
   Position->y += YPlus;
}
```

The `Perf` class contains the specific functions and properties of `FPS` and `CpuInfo`. It uses `spritebatch` to show this information.

# Asynchronous loading

In a few years, most computers and game consoles will support multiple CPU cores. Programming on multiple cores makes a lot of sense, as each part of the game can work separately. This section will demonstrate how to move from a single-thread load system approach to a multitask load system. It will cover how to load resources efficiently with tasks and at the same time provide you with a basic guideline on how to consume asynchronous methods using `ppltasks.h`, which is defined in the concurrency namespace.

# Introduction to tasks

A task is a unit of work that can execute asynchronously. When a task is created at runtime, it can execute on any available threads, and it is also able to use all the CPU cores of the machine. A set of tasks that are needed to work is called a taskset or a task group. A task cannot synchronize data between itself and other tasks, but it is possible to have a chain of tasks that can perform continuation. In the continuation tasks, each task is connected to another task so the task runs asynchronously after another task completes; also, the task and its related types provide us with the capability for cancellation.

If you use a synchronous method on a UI thread, the application is blocked and will not respond until the method returns successfully. To avoid this, it is advisable to use asynchronous tasks where the application continues in the main UI thread while another task waits to complete.

The following code shows a simple task:

```
auto t = create_task([&message]
{
  const int Total = 20;
  for (int i=0; i< Total; ++i)
  {
    message = L"task t0 is working : " +
 i.ToString() + L"/" + Total;
    wait(700);
  }
}).then([&message]()
{
  message = Continuation of the t0";
  wait(2000);
});
message = L"task t0 created";
t.wait();
message = L"task t0 finished";
```

Considering the example, the `create_task` function created a simple task that is named `t`.

This task performs a simple job. The `Wait(700)` function blocks the current thread for 700 milliseconds. When `t` is completed, the `concurrency::task::then` function invokes a second operation. The `t.wait()` function will run the task and then it will block the current thread until `t` performs its function.

Note that tasks typically run on the background threads. When you pass variables to a lambda expression by reference (such as `message` in the previous example), the lifetime of a variable must be guaranteed until the task finishes.

> The user interface of a Windows Store application runs entirely in a **single-threaded apartment** (**STA**).

To get a deeper understanding of the task types, please refer to the example of this section provided in the source codes. It will be a good point to start if you are not familiar with the tasks.

# Asynchronous resource loading

This section goes through the process of implementing asynchronous resource loaders to make use of them in the framework. We will compare the load times of synchronous and asynchronous ways of loading resources. Loading resources and animation is probably the best way of using tasks. Assume that we have a lot of models which have a lot of animated frames. With a task-based system, the programmers need to specify the updates of the set of frames in each task function.

For loading resources, usually we should avoid loading huge resources as a stream in the gameplay thread because this might have a lot of latency. With tasks, we can separate resources on each task so that they can be loaded on another thread along with the gameplay thread.

Let's look at a simple sample; open `Texture2D.h` from the `Asynchronous resources` project of this chapter.

In the previous chapter, we were introduced to the process of loading a texture, and now we will implement this process with a task as shown in the following code:

```
static task<void> LoadTextureAsync(String^ path, _Inout_ Texture2D^
&texture)
  {
    return DX::ReadFileAsync(path)
      then([texture](const Array<byte>^ textureData)
    {
      ID3D11ShaderResourceView* srv = nullptr;
      auto device = DX::GDevice.d3dDevice.Get();
      CreateDDSTextureFromMemory(
        device, textureData->Data, textureData->Length,
        nullptr, &srv);
      texture->Load(srv);
    });
  }
```

The task is responsible for creating a texture from a file. The `Asynchronous resources` project will load multiple assets. Take a look at the `Update` method of `SimpleScene.cpp` file in the source code of this project. In this method, we are going to load an asset several times. There are six states, which are the `NotStarted`, `LoadSync`, `LoadSync`, `LoadAsync`, `LoadTaskGroup`, and `Finished` states. These states help us to load the resources in different situations. In the `LoadSync` state, the application will load the assets sequentially.

In this situation, the main thread will be blocked since these assets are being loaded:

```
else if(state == State::LoadSync)
{
  timer->Reset();
  {
    for (int i=0; i <50; ++i){//Load asset}
    timer->Update();
  }
}
```

When the application is in the `LoadAsync` state, it will load each asset on a separate task. In this way, dependencies on each task will be very important, and it is necessary to break the task into pieces that are guaranteed to be independent of each other.

```
else if(state == State::LoadAsync)
{
  vector<task<void>> tasks;
  timer->Reset();
  for (int i=0; i <50; ++i)
  {
    tasks.push_back(/* Load asset which returns task<void> */);
  }
  when_all(tasks.begin(), tasks.end()).then([this] ()
  {
    timer->Update();
  });
}
```

All of the tasks which are run by `when_all` must return the same type. In the final state, the application will load the assets on a task group. The `parallel_invoke` function executes each task in parallel as shown in the following code:

```
else if(state == State::LoadTaskGroup)
{
   task_group tasks;
  timer->Reset();
  parallel_invoke([&]()
  {
    tasks.run([]
    {
      for (int i=0; i <10; ++i){ //Load assets }
    });
```

```
      tasks.run([]()
      {
        for (int i=0; i <15; ++i){ //Load assets }
      });
    },[&]()
    {
      tasks.run([]()
      {
        for (int i=0; i <25; ++i) { //Load assets }
      });
    });
    tasks.wait();
  timer->Update();
    }
  }
```

The example code is available in the source code; make sure to build the asynchronous resource project from the source code of this chapter and compare the result times

In this section, we introduced tasks and learned how to load resources using these tasks. Remember, moving the entire real-game engine to tasking is complicated. In the following section, we are going to demonstrate how to interact with the Model Editor of Visual Studio 2012.

# Getting started with the Model Editor

This section describes how to work with the Visual Studio Model Editor to view, create, and modify graphical assets. Visual Studio 2012 was released with a lot of amazing features that have provided game developers with great benefits. One of these new features is the Model Editor. 3D programmers always used to have problems with integrating their game engine with the 3D tools such as Softimage, Maya, 3DMax, and others. There are several standard formats that are used to export the geometric data from these tools, such as Collada, `.fbx`, `.obj`, and `.X` (no longer supported) files.

You can write your parser to each format, but there is a better way and also an easier way for smaller teams to do this, which is using the Model Editor. It gives you the ability to load the most favorable formats in order to convert them to a standard type which can be loaded easily.

These favorable input formats of the visual editor are Collada (`.dae`), `.fbx`, and `.obj`, and the output format is `.cmo`. Furthermore, you can use the Shader Designer, another new feature of Visual Studio 2012, which is used to create custom visual effects in your application and to edit dependency graphs without affecting the underlying codes.

The Shader Designer creates **Directed Graph Shader Language** (**DGSL**) files and also exports the HLSL shader source code (`.hlsl`), the HLSL shader bytecode (`.cso`), and the C++ header which contains an HLSL bytecode array.

Finally, with the Image Editor, you can work with a variety of rich textures and image formats which are being used in DirectX app development.

> Note that at this time, the graphic asset tools are not supported by Visual Studio 2012 Express.

Let's try using the Model Editor; to do so, follow the given steps:

1. Add an existing model to the project (in the format of `.obj`, `.fbx`, or `.dae`) or create a new **3D Scene (.fbx)** from the **Graphics** tab as shown in the following screenshot:
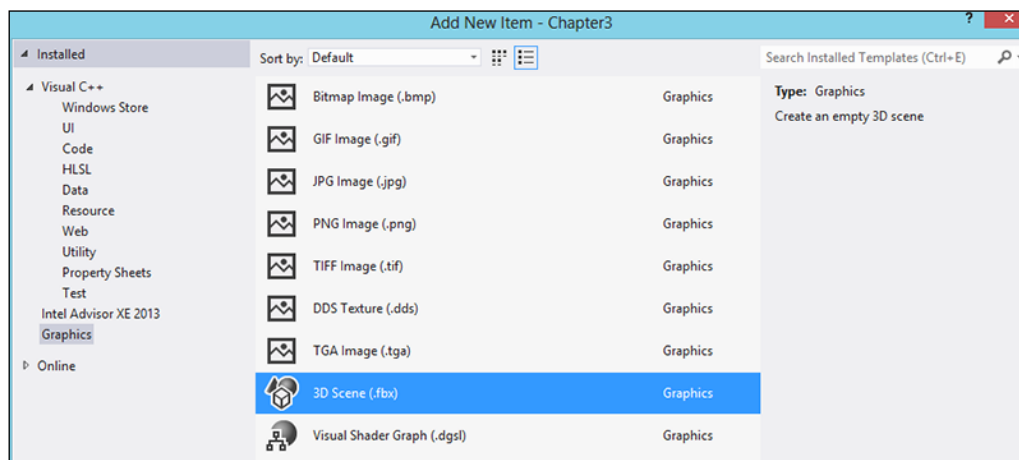
2. From the **Toolbox**, you can add different basic models to your editor; let's add a **Teapot** model to the scene. By selecting the teapot, you can change the properties of the model from the **Properties** window. For example, change the scale of the teapot from one to another value:
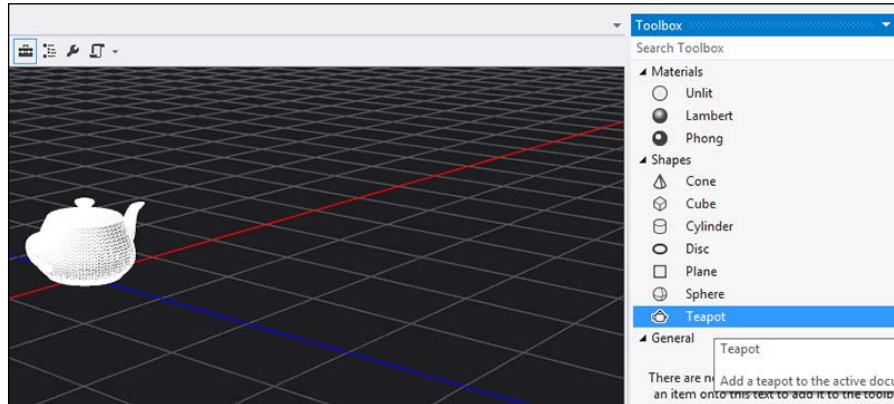


3. Go to the **Properties** window of the model. The default `phong.dgsl` is applied to the model from the following path:

```
~\Microsoft Visual Studio 11.0\Common7\IDE\Extensions\
Microsoft\VsGraphics\Assets\Effects
```

4. Right-click on **Teapot** and select **view material phong**; then a new window will appear as shown in the following screenshot, which shows the diagram of your shader. By default, the shader graph looks like the following screenshot:
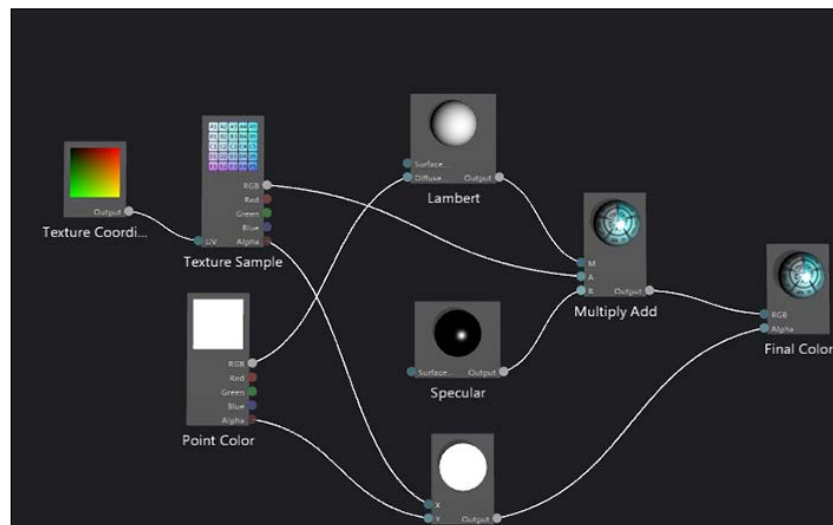
5. Assume that we want to change the color of this material; to do so, select **Color Constant** from the **Toolbox** dropdown and drag-and-drop it onto the editor.

6. Remove **Point Color** and make a connection between **RGB** of **Color Constant** and **Diffuse** of **Lambert** and also link **Alpha** of **Color Constant** to the **Y** input of **Multiply**. Then from the **Properties** window of **Color Constant**, change the output color to any other color you may prefer.

7. Now select **Texture Sample** and set an image to Texture1 from the **Properties** window.
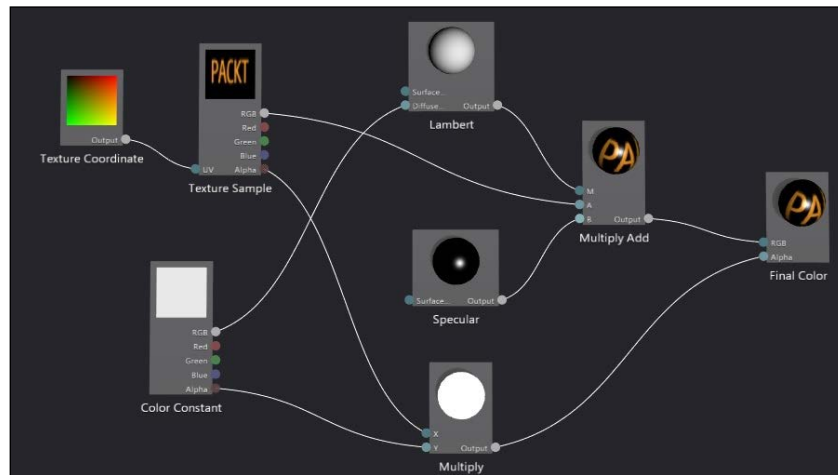


8. The previous screenshot shows the complete diagram of a material for which we have used **Texture Sample**, **Specular**, and **Lambert** to create the final color of the material.

9. We have created the Phong shader, so now let's add an edge detection to our shader. First, add **Edge Detection** from the **Toolbox** to the editor. Then make a connection between the **Output** of **Texture Coordinate** and the **UV** of **Edge Detection**. We need to apply **One Minus** before adding this result to the output of Phong. The following screenshot shows the complete diagram:
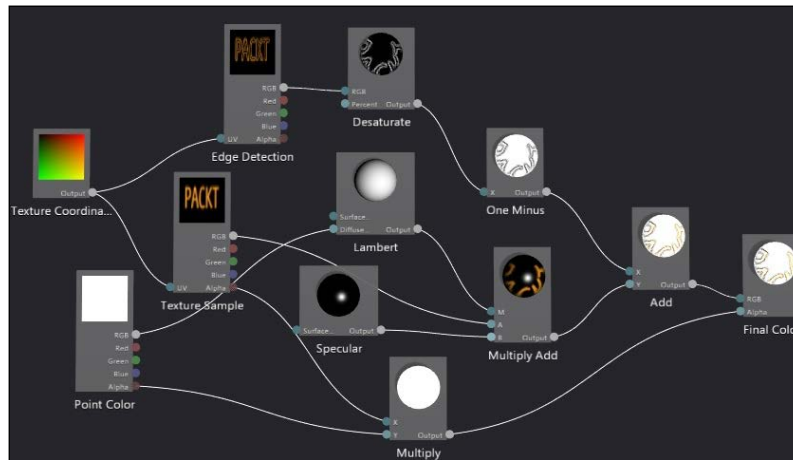


10. The following screenshot shows the result of each effect on the teapot:



11. Now you can export your material as an HLSL file or a compiled Pixel Shader file (`.cso`). You can also convert your model to `.obj`, `.dae`, or `.fbx`.

12. Before you can deploy your 3D assets as a part of your project, Visual Studio has to know about the kinds of assets that you want to deploy.

13. In **Solution Explorer**, right-click on the project; then, from its shortcut menu, select **Build Customizations**. After the **Visual C++ Build Customizations Files:** dialog box is displayed, click on the **Find Existing...** button and go to the Visual Studio installation directory, which is located at `~\Common7\ IDE\Extensions\Microsoft\VSGraphics\`, and add `ImageContentTask`, `MeshContentTask`, and `ShaderGraphContentTask`:
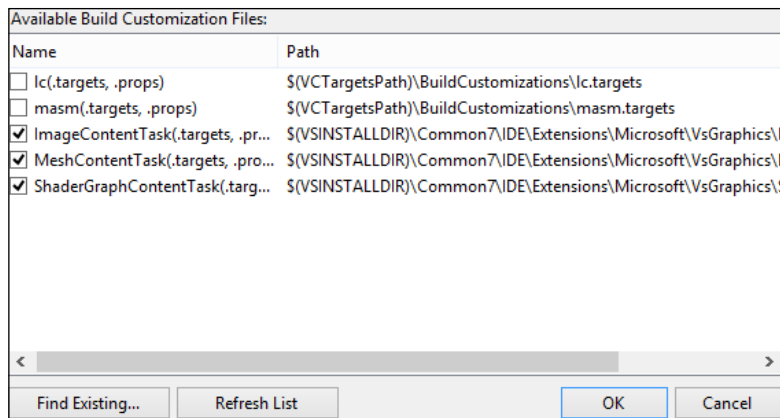
14. Finally, change the properties of your model as shown in the following screenshot:
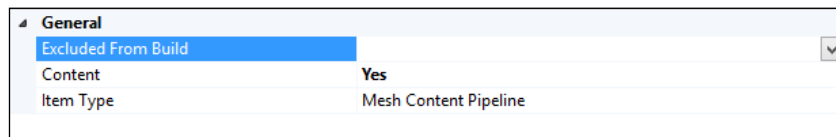


In this section, you have learned how to use the Model Editor and how to create and apply various graphical effects on the model.

# Loading a model from the .cmo format

In this section, we are going to learn about the process of loading meshes from the `.cmo` format step-by-step. The purpose of this section is to show you how to integrate your framework with an output object of the Visual Studio Model Editor. It will not rely on any particular file formats; after reading this section, you will be able to create your own content loader and encapsulate any meshes generated by the 3D tools.

As mentioned in the previous section, there are several formats that are used by the 3D tools and game engines, and choosing the best one is really up to you. For example, if you want to use the Visual Studio Model Editor and Shader Designer parsingly, then `.cmo` is the best choice; however, if you want to load extra data such as physics, rigid bodies, dynamics, or other formats from the 3D tools to your game engine, you should use another format or even create your own format. It is more efficient to create compressed assets because this will make the application smaller and it usually reduces the loading times.

Open the `Render CMO` project from the source code; then, open `Phong.dgsl` as shown in the following screenshot and export `Phong.dgsl` as the HLSL file:
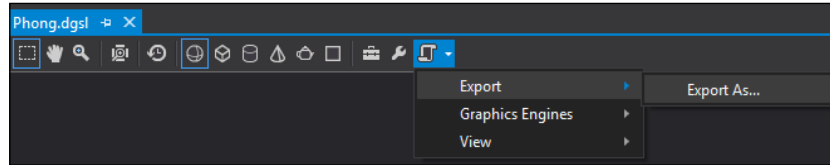


Open the HLSL file which was exported by the Model Editor. Here you can see the definition of this Pixel Shader file; we need to get familiar with its structure because we want to load and render the model with this shader.

Prior to loading `.cmo` files, make sure to check whether the item type of each `dgsl` file is set to `Shader Graph Content Pipeline`, the item type of `scene.fbx` is set to `Mesh Content Pipeline`, and finally whether all the textures that are used within `scene.fbx` are built with `Image Content Pipeline`.

Here we are going to introduce the structure of the `.cmo` format; you can find the example code in `Model.cpp` of the `Render CMO` project:

```
UINT => How many meshes
for each mesh
{
  UINT             => Length of mesh name
  Array of wchar_t => Read name of mesh
  UINT             => The count of materials
  for each material
  {
    UINT             => Length of material name
    Array of wchar_t => Read the name of material
      Read the structure of material
    {
      XMFLOAT4     => Ambient
      XMFLOAT4     => Diffuse
      XMFLOAT4     => Specular
      float        => Specular Power
      XMFLOAT4     => Emissive
      XMFLOAT4X4   => Transform of UV
    }//end of structure
```

These properties of the material can be set either in code or from the **Properties** window of the selected model in `Scene.fbx`.

In this project, we just use one shader and one texture for each material, but sometimes a material might have multiple textures or even shaders. It is more efficient to put everything you want in one shader and then apply the multiple materials to the submesh:

```
UINT            => Length of pixel shader name
Array of wchar_t => Read the name of pixel shader
```

In the Visual Studio Model Editor, you can assign eight textures to your material. In the following code, we just use the first texture, which is the color texture:

```
for I = 0 to I = 8
{
  UINT             => Length of texture name
  Array of wchar_t => Read the name of texture
}//end of textures
}//end of materials
BYTE => if 1 means the mesh contains skeletal animation
UINT => The counts of sub mesh
for each sub mesh
{
   UINT  => The index of material
   UINT  => The index of index buffer
   UINT  => The index of vertex buffer
   UINT  => The start index of sub mesh
   UINT  => The count of primitives
}
```

Each `mesh` has got one or more `sub mesh`. Each `sub mesh` includes the following information:

- The material that has been used for it
- The start index of the `vertex buffer` and `index buffer` of the submesh
- How many primitives (simple geometric objects) it has

Our model has got a `vertex buffer` and an `index buffer` to store the vertices and indices and also a list of submeshes, triangles, and materials for rendering primitives. Some parameters such as rotation, position, world, and bounding sphere are used to control the model in a coordinate system.

Read the data of `vertex buffers` and `index buffers`. Note that the size of each vertex is 52 bytes (this refers to the size of `VertexPositionNormalTangent ColorTexture`) as shown in the following code:

```
UINT => Count of index buffers
for each index buffer
{
    UINT             => Count of USHORTs in index buffer
    Array of USHORT => read array of indices
}
UINT => Count of vertex buffers
for each vertex buffer
{
  UINT             => Count of vertex in vertex buffer
  Array of USHORT => read array of vertices
}
UINT => Count of skinning vertex buffers
for each skinning vertex buffer
{
  UINT             => Count of vertex in skinning vertex buffer
  Array of USHORT => read array of skinning vertices
}// end of for each skinning
Read the bounding sphere of mesh
{
  XMFLOAT3  => Center of bounding sphere;
  float     => Radius of bounding sphere;
  XMFLOAT3  => Min point from vertices
  XMFLOAT3  => Max point from vertices;
}// end of bounding sphere structure
If mesh has the skeleton animation data
UINT => Count of Bones
for each Bone
{
  UINT             => Length of Bone name
  Array of wchar_t => Read the Bone name
}// end of for each Bone
UINT => The count of animation clips
for each animation clip
{
  UINT             => Length of Clip name
  Array of wchar_t => Read the Clip name
  float            => Start time
```

```
    float            => End time
    UINT             => Count of Keyframes
    for each key frame
      {
        Read key frame structure
          {
            UINT       => index of Bone
            float      => time of key frame
            XMFLOAT4X4 => Transform
          }// end of key frame structure
      }// end of for each key frame
  }// end of for each animation clip
}// end of for each mesh
```

In this section, you have successfully learned how to read a `.cmo` file. In the next section, we are going to learn how to render a model.

# Rendering a model

This section will discuss how to render the runtime assets that are produced by the `MSBUILD` task from the Model Editor. We will use the Pixel Shader which is created by the Shader Designer and a default Vertex Shader for rendering the model. In order to render the model, you need to get familiar with the constant buffers of `phong.hlsl`. There are four constant buffers that are declared in this shader.

The first one is called the `MaterialVars` buffer, and it contains the properties of the material and its structure, which were introduced to you in the previous section; the second is the `LightVars` buffer which contains the properties of the light; the third one, which is named `ObjectVars`, contains the transformation matrices; and the fourth one, `MiscVars`, presents the time and the viewport size. For example, following is the structure of `LightVars` in our framework:

```
struct LightVars
{
  XMFLOAT4   AmbientLight;
  XMFLOAT4   LightColor[4];
  XMFLOAT4   LightAttenuation[4];
  XMFLOAT4   LightDirection[4];
  XMFLOAT4   LightSpecularIntensity[4];
  UINT       IsPointLight[16];
  UINT       ActiveLights;
```

```
   XMFLOAT3    Paddings;
   LightVars()
   {
     ZeroMemory(this, sizeof(LightVars));
     AmbientLight = XMFLOAT4(1.0f,1.0f,1.0f,1.0f);
   }
};
```

This structure represents the properties of the light, but what is `Paddings` which is being used for this structure? The answer is that the HLSL packs do not cross a 16-byte boundary, so the variables are packed into a given four-component vector until the variable will straddle a 4-vector boundary. As mentioned in the previous chapter, if you would like the shader to run efficiently on all hardware, you should make sure that the shader is properly vectorized.

Open the `Model.cpp` file from the `Graphics/Models` folder. In the `Load` method, when all the tasks are finished, it means that the model has loaded successfully; then, the constant buffer of `LightVars` will get initialized:

```
when_all(tasks.begin(), tasks.end()).then([this]()
  {
    loadingComplete = true;
    CObjectVars.Load();
    CMaterialVars.Load();
    CLightVars.Load();
    CLightVars.Const.ActiveLights = 1;
    CLightVars.Const.AmbientLight =  XMFLOAT4(0.3f, 0.3f, 0.3f,
      1.0f);
    CLightVars.Const.IsPointLight[0] = false;
    CLightVars.Const.LightColor[0] = XMFLOAT4(0.8f, 0.8f, 0.8f,
      1.0f);
    CLightVars.Const.LightDirection[0].x = 1;
    CLightVars.Const.LightDirection[0].y = 1;
    CLightVars.Const.LightDirection[0].z = 1;
    CLightVars.Const.LightDirection[0].w = 0;
    CLightVars.Const.LightSpecularIntensity[0].x = 2;
  });
```

Check the `Update` method of this class. First rotate the model and then calculate the world matrix of the model. Each model can have at least one transformation applied to it; if the model needs to move in the world, we need to update the translation transform of the world matrix. If you need to rotate the model or scale it either downwards or upwards, you need to apply the rotation and scaling transformation to it.

Apply these transformations to the world matrix as follows:

World of mesh = (Scale) * (Rotation) * (Position)

For calculating rotation, we have $R_x$, $R_y$ ,and $R_z$, which are the rotation matrices around the world matrix for the X, Y, and Z axes:

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}$$

$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix}$$

$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If we multiply these three matrices, the result is as follows:

$$R_f = R_x(\theta_x)\, R_y(\theta_y)\, R_z(\theta_z)$$

$$R_f = \begin{bmatrix} \cos\theta_y\,\cos\theta_z & -\cos\theta_y\,\sin\theta_z & \sin\theta_y \\ \sin\theta_x\,\sin\theta_y\,\cos\theta_z + \cos\theta_x\,\sin\theta_z & -\sin\theta_x\,\sin\theta_y\,\sin\theta_z + \cos\theta_x\,\cos\theta_z & -\sin\theta_x\,\cos\theta_y \\ -\cos\theta_x\,\sin\theta_y\,\cos\theta_z + \sin\theta_x\,\sin\theta_z & \cos\theta_x\,\sin\theta_y\,\sin\theta_z + \sin\theta_x\,\cos\theta_z & \cos\theta_x\,\cos\theta_y \end{bmatrix}$$

Assuming that the scale is equal 1, the world matrix creates the following elements:

$$World = \begin{bmatrix} Rf_{11} & Rf_{12} & Rf_{13} & Position_X \\ Rf_{21} & Rf_{22} & Rf_{23} & Position_Y \\ Rf_{31} & Rf_{32} & Rf_{33} & Position_Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Refer to the following code to see how the world matrix works:

```
void Model::Update(float time)
{
  Rotation.x += time;
  Rotation.y += time;
  Rotation.z += time;
  //Update world
  world =
    XMMatrixScaling(1, 1, 1) *
XMMatrixRotationX(Rotation.x) *
XMMatrixRotationY(Rotation.y) *
XMMatrixRotationZ(Rotation.z) *
XMMatrixTranslation(Position.x, Position.y, Position.z);
}
```

And here is the `Render` method; it simply renders each `submesh` with the corresponding material.

> In this example, we use a camera to get the view and projection matrices. We will introduce different types of cameras later in this chapter.

```
void Model::Render()
{
if (!loadingComplete) return;
  auto view = Camera.GetView();
  auto projection = Camera.GetProjection();
  GDevice.d3dContext>IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
  for (UINT i = 0; i < subMeshes.size(); i++)
  {
    auto submesh = this->subMeshes[i];
```

```
if (submesh.IndexBufferIndex < indexBuffers.size() &&
  submesh.VertexBufferIndex < vertexBuffers.size())
{
  UINT offset = 0;
  GDevice.d3dContext->IASetVertexBuffers(0, 1,
    &vertexBuffers[submesh.VertexBufferIndex],
    &stride,
    &offset);
  GDevice.d3dContext>IASetIndexBuffer(
    indexBuffers[submesh.IndexBufferIndex],
    DXGI_FORMAT_R16_UINT, 0);
}
auto _material = materials[submesh.MaterialIndex];
auto _shader = materials[submesh.MaterialIndex].shader;
memcpy(&CMaterialVars.Const.Ambient,
  &_material.Const.Ambient, sizeof(_material.Const.Ambient));
memcpy(&CMaterialVars.Const.Diffuse,
  &_material.Const.Diffuse, sizeof(_material.Const.Diffuse));
memcpy(&CMaterialVars.Const.Specular,
  &_material.Const.Specular,
  sizeof(_material.Const.Specular));
memcpy(&CMaterialVars.Const.Emissive,
  &_material.Const.Emissive,
  sizeof(_material.Const.Emissive));
CMaterialVars.Const.SpecularPower =
  _material.Const.SpecularPower;
CMaterialVars.Update();
```

First, specify the primitive topology as a triangle list by setting `D3D11_PRIMITIVE_`
`TOPOLOGY_TRIANGLELIST` to the device context; then, for each `submesh`, set the vertex
buffer and index buffer and update the constant buffer of the related material. The
`memcpy` function is required to copy each material of `submesh` to the constant buffer:

```
XMMATRIX localToView = world * view;
CObjectVars.Const.LocalToWorld4x4 =
  XMMatrixTranspose(this->world);
CObjectVars.Const.LocalToProjected4x4 =
  XMMatrixTranspose(world * view * projection);
CObjectVars.Const.WorldToLocal4x4 =
  XMMatrixTranspose(XMMatrixInverse(nullptr, world));
CObjectVars.Const.WorldToView4x4 = XMMatrixTranspose(view);
CObjectVars.Const.UvTransform4x4 = XMMatrixIdentity();
CObjectVars.Const.EyePosition = Camera.GetPosition();
```

```
CObjectVars.Update();
CLightVars.Update();
_shader->SetConstantBuffer(0, 1, CMaterialVars.GetBuffer());
_shader->SetConstantBuffer(1, 1, CLightVars.GetBuffer());
_shader->SetConstantBuffer(2, 1, CObjectVars.GetBuffer());
_shader->SetTexture2D(0, 1, _material.texture);
```

As you might see in `phong.hlsl`, the first constant buffer that is defined in the register as `b0` is `MaterialVars`, so the starting slot of this constant buffer is zero and the number of buffers is one. Similarly, `MaterialVars`, `b1`, `b2`, and `b3` registers are being used for `LightVars`, `ObjectVars`, and `MiscVars`:

```
GDevice.d3dContext->OMSetBlendState(nullptr, nullptr,
  0xFFFFFFFF);
_material.shader->Apply();
{
  GDevice.d3dContext->DrawIndexed(
    submesh.PrimCount * 3, submesh.StartIndex, 0);
}
}
```

Finally, render the primitives of each `submesh` with the shader of the material. In this section, we learned how to render models that have been built with the Model Editor of Visual Studio 2012. The Model Editor and the Shader Designer are two great features of Visual Studio 2012 (not express!). They have provided the basic functions for small developer teams to create the levels of the game within the Visual Studio without using or even creating the game editor. In the next section, we are going to learn how to use inputs in the Metro applications.

# The input devices we'll need

In this section, we are going to learn how to handle input devices such as a keyboard, mouse or touch, and Xbox controllers. Let's start the integration of our framework with the keyboard, the old friend of gamers.

# Keyboard

Open the `Inputs` project from the source code and then open the `KeyboardState.cpp` class from the `FrameWork/Input` folder. The `KeyboardState.cpp` class has the following code:

```
bool KeyboardState::IsKeyDown(VirtualKey key)
{
  return keys[key] == true;
}
bool KeyboardState::IsKeyUp(VirtualKey key)
{
  return keys[key] == false;
}
void KeyboardState::SaveKeyState(VirtualKey key, bool IsPressed)
{
  keys[key] = IsPressed;
}
void KeyboardState::ClearBuffer()
{
  for (auto k : this->keys) { k.second = false; }
}
void KeyboardState::Unload()
{
  keys.clear();
}
```

The preceding methods simply store the states of the keys. When the keys are pressed or released, the `SaveKeyState` method stores the keys' states as Boolean values. The `IsKeyDown` and `IsKeyUp` methods return the keys' states; the `ClearBuffer` method is responsible for resetting all the stored states.

We need a class to handle and manage all the inputs, so let's call it the `InputManager` class; open the `InputManager.h` class and you will see the following code:

```
#pragma once
#include "KeyboardState.h"
namespace InputManager
{
  extern KeyboardState keyboardState;
  extern void Update();
  extern void ClearBuffers();
  extern void Unload();
}
```

All the fields and methods of the `InputManager` class are declared as `extern`. We have them in the C++ class as shown in the following code:

```
#include "pch.h"
#include "InputManager.h"
KeyboardState InputManager::keyboardState;
void InputManager::ClearBuffers()
{
  keyboardState.ClearBuffer();
}
void InputManager::Unload()
{
  keyboardState.Unload();
}
```

We need to relate the `InputManager` class and the events of the `Window.` function. In the `windows.cpp` class, we must include `inputManager.h` and add the following changes to this class:

```
void Window::SetWindow(CoreWindow^ window)
{
  //Keyboard's events
  window->KeyDown += ref new
TypedEventHandler<CoreWindow^, KeyEventArgs^>
(this, &Window::OnKeyDown);
  window->KeyUp += ref new
    TypedEventHandler<CoreWindow^, KeyEventArgs^>
    (this, &Window::OnKeyUp);
}
```

First create an event for `KeyDown` and `KeyUp` of the `window`, and then in the `Run` method of the `window`, call the `Update` method of the `InputManager` class before updating the scene and call `ClearBuffers` after presenting the scene:

```
void Window::Run()
{
  InputManager::Update();
  Scene->Update(Total, Delta);
  Scene->Present();
  InputManager::ClearBuffers();
}
void Window::Uninitialize()
{
  SafeUnload(this->Scene);
  InputManager::Unload();
}
void Window::OnKeyDown(CoreWindow^ sender, KeyEventArgs^ args)
{
  InputManager::keyboardState.SaveKeyState(args->VirtualKey,
    true);
}
void Window::OnKeyUp(CoreWindow^ sender, KeyEventArgs^ args)
{
  InputManager::keyboardState.SaveKeyState(args->VirtualKey,
    false);
}
```

These events simply store the keys' states in the `keyboardState` class. Now we can use the keyboard's inputs in our scene, for example:

```
if (InputManager::keyboardState.IsKeyDown(VirtualKey::W))
{
  //Write your code
}
```

# Pointer

For personal computers, the pointer indicates the mouse, but in tablets, touch screens, or mobile devices, the pointer's events mean the touch screen's events. Fortunately, DirectX 11.1 will support all types of these inputs. Touch and mouse controls have got a very similar core implementation. In a Windows Store application, a pointer is simply a point on the screen or a moving mouse on the screen. You can handle both of them in a single event. Open the `PointerState.cpp` class from the same folder as illustrated in the following code:

```
void PointerState::SavePosition( Point Position )
{
  position.x = Position.X;
  position.y = Position.Y;
}
void PointerState::SaveState( PointerPointProperties^ Properties )
{
  state = Properties;
}
bool PointerState::IsLeftButtonPressed()
{
  if(state == nullptr) return false;
  return state->IsLeftButtonPressed;
}
bool PointerState::IsRightButtonPressed()
{
  if(state == nullptr) return false;
  return state->IsRightButtonPressed;
}
```

The `PointerPointProperties` class gives us the properties of the pointer. These properties are `IsLeftButtonPressed`, `IsRightButtonPressed`, `IsXButton1Pressed`, `MouseWheelDelta`, and many other useful properties that help us control the pointer.

Similar to `KeyboardState`, we need to add `PointerState` to the `InputManager` class. In the final step, we will integrate `windows.cpp` and `windows.h` with the `InputManager` class as shown in the following code:

```
void Window::OnPointerPressed(CoreWindow^ sender, PointerEventArgs^
args)
{
  InputManager::pointerState.SaveState(args->CurrentPoint-
    >Properties);
}
```

```
void Window::OnPointerReleased(CoreWindow^ sender,
PointerEventArgs^ args)
{
  InputManager::pointerState.SaveState(args->CurrentPoint-
>Properties);
}
void Window::OnPointerMoved(CoreWindow^ sender, PointerEventArgs^
args)
{
  InputManager::pointerState.SavePosition(args->CurrentPoint-
>Position);
}
```

These events just update `pointerState`. When the user presses or releases the pointer, they store the properties of the pointer into `pointerState`. Also, `OnPointerMoved` is responsible for receiving and storing the position of the pointer. Now if you want to use the pointer in your scene, you can write the following code:

```
if (InputManager::pointerState.IsLeftButtonPressed())
{
  //Write your code
}
```

# Xbox 360 controllers

Playing games with a gamepad controller is more enjoyable than with a mouse and keyboard. If you have a wired Xbox 360 gamepad, it has to be plugged into a USB port on your PC. `XInput` is an application programming interface that enables us to receive the inputs from the Xbox 360 controllers; these inputs can be buttons, voice input, and the controlling of the rumble packs, which are compatible with the Windows application.

Open the `GamePadState.cpp` class from the project; also, make sure that in order to use `xinput`, you need to link the header and the library of `xinput` to the project as shown in the following code:

```
void GamePadState::Update()
{
  for(UINT i=0; i< MAXCONTROLLERS; i++)
  {
    auto hr = XInputGetState( i, & states[i].XState );
    if( hr == ERROR_SUCCESS )
```

```
      {
        states[i].isConnected = true;
      }
      else
      {
        states[i].isConnected = false;
      }
    }
  }
```

The XInputGetState method retrieves the state of the controller; this state is typed as the XINPUT_STATE structure. This structure contains a packet number, which is a type of DWORD, and another structure, which is called the XINPUT_GAMEPAD structure. When the state of the controller changes, the packet number also changes and you need to get the new controller states. If the packet number is not different from the previous packet number, this means that there are no changes to the states of the controller.

The controller index can be a value between 0 and 3. If the function fails to execute, the return value is an error, which is defined in Winerror.h; however, if it succeeds, the return value can be either ERROR_SUCCESS when the controller is connected to the USB or ERROR_DEVICE_NOT_CONNECTED when it is not connected, as shown in the following code:

```
bool GamePadState::IsLeftThumbToLeft(UINT index)
{
  auto state = this->states[index];
  if(!state.isConnected) return false;
  if (state.XState.Gamepad.sThumbLX <
    -XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE)
  {
    return true;
  }
  return false;
}
```

Many of these methods behave in a similar manner; for example, if you want to check the left thumbstick, assume that it is moved to the left-hand side, and then we just need to compare it with XINPUT_GAMEPAD_LEFT_THUMB_DEADZONE.

The ClearBuffers function resets all states to their initial values and the Unload function is responsible for clearing them up. Now we can use a gamepad in our scene as shown in the following code:

```
if (InputManager::gamePadState.IsRightThumbToLeft(0))
{
  //Write your code
}
```

In this section, we presented the types of inputs for Windows Store games. We also learned how to use these inputs in our framework. In the next section, we are going to add some different types of cameras to the framework.

# Turn on the camera

The camera is one of the most elementary components of each 3D application, so in this section, we are going to turn it on.

The camera is responsible for showing the viewport of the user, and a 3D scene might have many cameras; there are many types of cameras that are introduced depending on the genre of each game, such as a first person, third person, and the other types. In this section, we are going to discuss three types of cameras–the base camera which is known as the fixed camera, the first person camera, and third person cameras, which can be inherited from the base camera.

# Base camera

Prior to getting your hands dirty with the code, it is more important to understand what the basic properties of a camera are. The view and projection matrices are the most important matrices that are needed to make a camera. The view matrix holds information about the position of the camera, what direction it is pointing to, and its orientation.

The projection matrix holds information about what the angle of the camera is and how far or near the camera can see. After defining a 3D area by a projection matrix, all the objects that are located in this area will be viewable from the camera. If the object is not shown in the scene, check the object's position to make sure that it is not located in front of the near clip or behind the far clip, and also check the pointing direction of the camera as shown in the following screenshot:
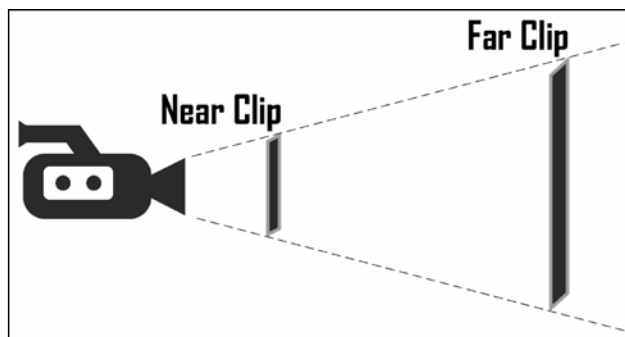
> Try to avoid setting a large value for the far clip even when you are creating a simple game. When an unexpected large value is set to the far clip, the model that is located near the far clip will be shown small and also with hard edges, but they will still use the same graphical process to render their primitives.

That is enough of theory, so now let's see how to implement a base camera. Open the `First Person Camera` project and then look at the `BaseCamera.cpp` class which is located in the `FrameWork/Cameras` folder:

```
void BaseCamera::UpdateView()
{
  auto _position = XMLoadFloat3(&position);
  auto _lookAt = XMLoadFloat3(&lookAt);
  auto _up = XMLoadFloat3(&up);
  view = XMMatrixLookAtRH(_position, _lookAt, _up);
}
```

By calling `XMMatrixLookAtRH`, we create a view matrix for the right-hand coordinate system from the camera position by looking at the focal point and the upward direction of the camera:

```
void BaseCamera::UpdateProjection(float fieldOfView, float
aspectRatio, float nearPlane, float farPlane )
{
  float fovY = fieldOfView * XM_PI / 180.0f;
  if (aspectRatio < 1.0f)
  {
    up = XMFLOAT3(1.0f, 0.0f, 0.0f);
    fovY = 120.0f * XM_PI / 180.0f;
  }
  else
  {
    up = XMFLOAT3(0.0f, 1.0f, 0.0f);
  }
  projection = XMMatrixPerspectiveFovRH(fovY, aspectRatio,
    nearPlane, farPlane);
}
```

When the screen orientation changes, we need to recreate the projection matrix. If the width of the screen is greater than its height, it means that we are using a landscape view, so the upward direction is the same as before; however, if we use a portrait or a snap view, we need to use different values for handling this situation.

By calling the XMMatrixPerspectiveFovRH method, the right-handed perspective projection matrix is created based on the field of view, the proportional relationship between the width and height of the screen, and finally the far and near clips of the camera.

# First person camera

A very common camera type is the first person camera, which is used extensively in first person shooter games.

Now that we had an introduction to the base camera, we are going to inherit a camera from this base camera to create a first person camera. You can use the keyboard and mouse to rotate and move in the 3D scene. Before everything else, make sure to open the FirstCamera.cpp class from the same project and you'll see the following code:

```
void FirstCamera::UpdateView()
{
  auto forward = XMVectorSet(0.0f, 0.0f, -1.0f, 1.0f);
  auto _lookAt = XMLoadFloat3(&position)
    + XMVector3Transform(forward, rotationMatrix);
  this->lookAt.x = XMVectorGetX(_lookAt);
  this->lookAt.y = XMVectorGetY(_lookAt);
  this->lookAt.z = XMVectorGetZ(_lookAt);
  BaseCamera::UpdateView();
}
```

In the UpdateView method, we need to change the focal point of the camera. Because we need to move and rotate the first person camera, the transformation of the focal point (lookAt) is done by the summation of the current position and the rotation matrix, which is transformed by the forward vector.

After calculating the lookAt vector, call the UpdateView method of the base camera in order to apply these changes to the view matrix as shown in the following code:

```
void FirstCamera::UpdateWorld(XMFLOAT3 MoveVector)
{
  auto vect = XMVector3Transform(XMLoadFloat3(&MoveVector),
rotationMatrix);

  position.x += XMVectorGetX(vect);
  position.y += XMVectorGetY(vect);
  position.z += XMVectorGetZ(vect);
}
```

The `UpdateWorld` function is called when the angle or position of a camera changes. We use `XMVector3Transform` to transform `MoveVector` by the rotation matrix. The rotation matrix will be calculated by the angle's value as shown in the following code snippet:

```
rotationMatrix = XMMatrixTranspose(
  XMMatrixRotationX(angle.x) * XMMatrixRotationY(angle.y));
```

Usually in the first person camera, the player is free to look around the world, but rotation around the forward vector is not allowed; because of this, we use a 2D rotation parameter for the first person camera, which is named `angle` in this example. The `angle` parameter has the rotation's value around the right and the upward directions.

Let's check the `Update` method. First, as mentioned before, the rotation matrix is created, and then we need to check the user inputs to change the position and the rotation of the camera. If the right button of the mouse or pointer is pressed, the rotation of the camera must be updated upon the movements of the mouse. With the *A*, *D*, *S*, *Q*, and *Z* keys, you can change the position of the camera.

Finally, at the end of the update routine, we need to call the `UpdateView` method to update the view matrix.

The next section will discuss the most interesting camera in 3D games.

# Third person camera

The third person camera is one of the most interesting cameras in video games. In order to implement this type of camera, you must have an object that is chased by the camera with a specific distance and direction. As you can see in the following diagram, the chase position is the position of the object, the chase direction is the direction where the camera must follow the target, and the desired distance is the distance between the camera and the target:
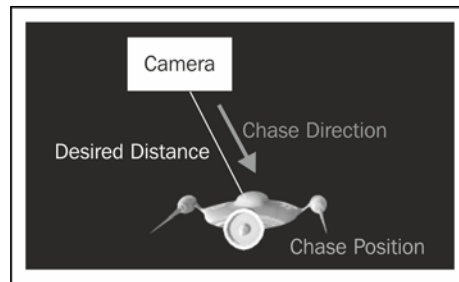
Like the `FirstCamera` class, the `ChaseCamera` class must be inherited from the `BaseCamera` class. Open the `Third Person Camera` project and let's look at the definition of `ChaseCamera.cpp`. This class is located in the `FrameWork/Cameras` folder.

`ChaseCamera.cpp`:

```
void ChaseCamera::Update(XMFLOAT3 BindPos, float Yaw)
{
  const float Step = 0.016f;
  yaw = Yaw;
  auto rotationMatrix = XMMatrixRotationRollPitchYaw(
    pitch, yaw, 0);
  desiredPosition = BindPos +
    TransformNormal(desiredPositionOffset, rotationMatrix);
```

In the chase camera, the orientation must be over the orientation of the chased model. After calculating the rotation matrix, we need to set the desired position of the camera.

The `desiredPositionOffset` variable stores an offset vector between the camera and the chased model. Note that `desiredPositionOffset` is a constant value, so we need to manually set it for each model.

Because the camera must chase the center of a model, the desired position of the camera is calculated by the position of the model (`BindPos`) and the `TransformNormal` of the offset vector by `rotationMatrix`:

```
    auto stretch = position - desiredPosition;
    auto force = ( - stiffness * stretch ) - (damping * velocity );
    auto acceleration = force / mass;
    velocity = velocity + acceleration * Step;
    position = position + velocity * Step;
```

You can change the variables of the chase camera such as the velocity, damping, and mass to change how fast or smooth the camera should be while chasing the model.

Finally, when the position is calculated, we must update the view matrix as mentioned in the previous section:

```
lookAt = position + XMMatrixForward(rotationMatrix);
BaseCamera::UpdateView();
```

Now take a look at the `SimpleScene.cpp` file in the source code of this project to see how to add Chase camera and how to update it  within the `Update` method of our scene.

The following screenshot shows the output result of our sample:
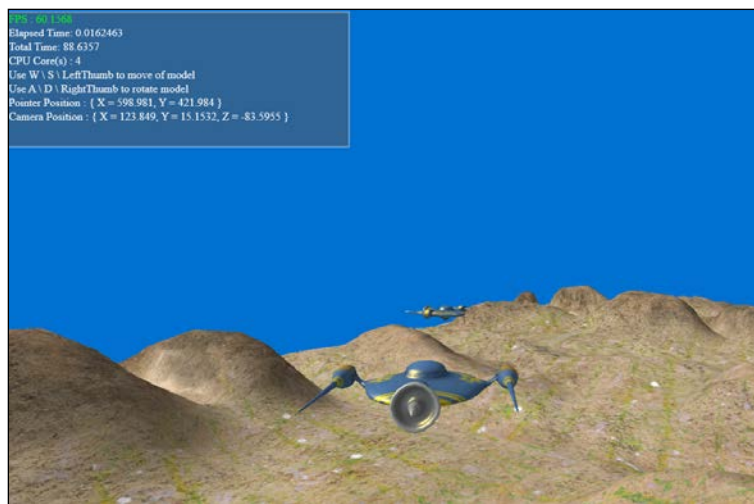


In the next section, we are going to integrate XAML with our framework in order to have a simple editor.

# Composing XAML and Direct3D

Welcome to one of the most amazing features of C++/CX. Most game programmers have the same problem when it comes to developing an editor for their game engine because it requires a lot of effort.

In order to develop an editor for a game engine, you can start wrapping a native DLL in C# and use a WPF or Windows Form application as the GUI of your game engine; this requires a lot of calls for unmanaged DLLs and involves a lot of pointers which are only available in the unsafe mode of C#. If you don't prefer this way, instead of using .NET, you can go deep into the native code and then use libraries such as `Nokia QT`, `wxWidgets`, or `cegui`. These libraries are open source and use the standard C++ language, and they also give you standard user controls.

You can also write in your editor using MFC or buy the license of Autodesk Scaleform, which enables developers to increase the power of the Adobe Flash tool that is used to create powerful and immersive user interface environments for video games and beyond.

Before the C++/CX language extensions were released, I wondered why there was no way to have C++ as a code-behind XAML. You could not decide whether you were happy or not when C++/CX was released with Microsoft Windows 8 because it gave you great features just for Metro Style applications, but unfortunately runs only on the Windows 8 runtime libraries. C++/CX consists of both the power of UI designing with XAML and native programming for developers.

The following image shows the diagram of building XAML and C++ code:
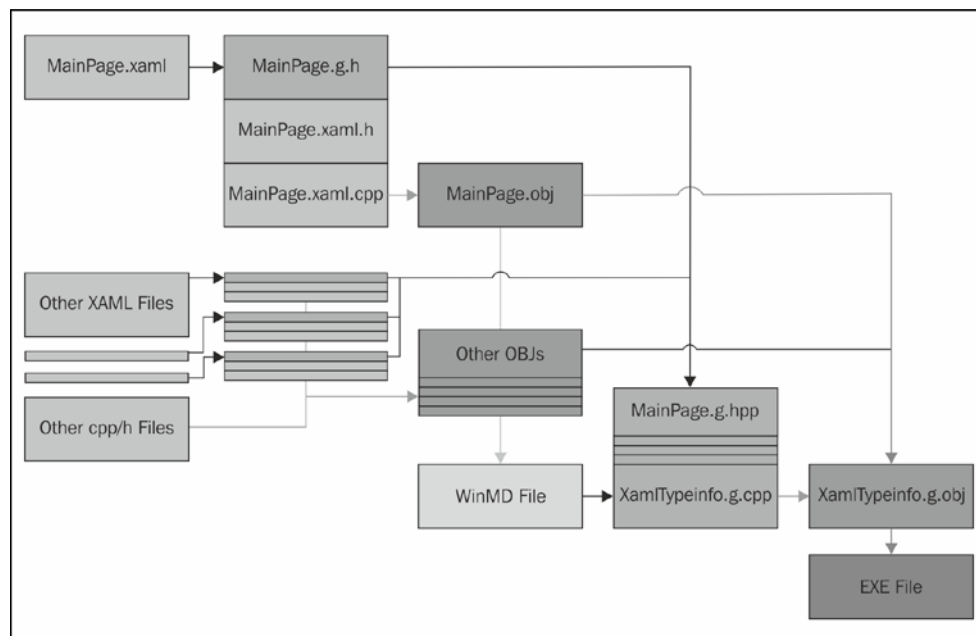


This diagram was obtained from `http://blogs.msdn.com/b/vcblog/ archive/2012/08/24/10343401.aspx`.

Let's open the final project of this chapter (XAML). We are going to use the combination of XAML and DirectX, which helps us to build a flexible user interface for DirectX-rendered content.
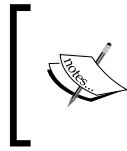
> If you are unfamiliar with XAML, read the *Windows Presentation Foundation 4.5 Cookbook*. You can get more information about this book from `http://www.packtpub.com/windows-presentation-foundation-4-5-cookbook/book`.

The following is the structure of `MainPage.xaml`:

```
<SwapChainBackgroundPanel x:Name="SwapChainPanel"
  Loaded="OnLoaded"
  Unloaded="OnUnloaded">
</SwapChainBackgroundPanel>
```

WPF developers are familiar with these controls, but what is `SwapChainBackgroundPanel`?

The `SwapChainBackgroundPanel` panel is the Windows runtime type designed to manage the swap chain directly in XAML. For each application, only one instance of this object will be created.

According to MSDN, this object is inherited from the `Windows::UI::Xaml::Cont rols::Grid` element and supports the similar layout behavior; hence, you can add XAML elements and UI overlays to this object, but note that all the elements are drawn over the swap chain.

For composing your XAML and Direct3D, first create `SwapChainBackgroundPanel`. The UI controls are added as children of `Page.BottomAppBar`. We need two checkboxes to handle the scene; also, we need two text blocks for showing static texts, and finally we need a button to exit from the application. The style of checkboxes is defined as a static resource in `App.xaml`:

```
<Application.Resources>
  <Style x:Key="TextBlockStyle" TargetType="TextBlock">
    <Setter Property="VerticalAlignment" Value="Center">
    </Setter>
    <Setter Property="HorizontalAlignment" Value="Center">
    </Setter>
    <Setter Property="Margin" Value="10">
    </Setter>
    <Setter Property="FontSize" Value="20">
    </Setter>
  </Style>
</Application.Resources>
```

Open the `MainPage.xaml.cpp` class from the root of the project. If you notice, the code is familiar; we saw it in the previous chapter. When the user clicks on the **Exit** button, `OnExitBtnClicked` is called, which unloads our scene and then exits from the application:

```
void MainPage::OnExitBtnClicked(Platform::Object^ sender, Windows::UI:
:Xaml::RoutedEventArgs^ e)
{
  OnUnloaded(sender, e);
  Application::Current->Exit();
}
```

In the `Checked` and `UnChecked` events, we just change the properties of a scene in order to control whether HUD or models are being rendered or not.

```
void Chapter3::MainPage::Checked(Platform::Object^ sender, Windows::UI
::Xaml::RoutedEventArgs^ e)
{
  if (Scene == nullptr) return;
  auto CheckBox = safe_cast<CheckBox^>(sender);
  if (CheckBox->Tag->ToString() == "0")
  {
    this->Scene->RenderHUD = true;
  }
  else
  {
    this->Scene->RenderModels = true;
  }
}
```

There is only one step to compose XAML and Direct3D. As mentioned earlier, we created a swap chain with `SwapChainBackgroundPanel`. So, in `game.cpp`, we do not need to create the swap chain from `corewindow`; instead, we used `CreateSwapChainForComposition` to get it from `swapChainPanel`. We need to add the `windows.ui.xaml.media.dxinterop.h` header to `pch.h` and finally change the scaling property of the swap chain description from `DXGI_SCALING_NONE` to `DXGI_SCALING_STRETCH`; this flag specifies stretched scaling for the swap chain.

Controlling our framework is much easier than before with the advance in the integration of XAML and native C++.

# Summary

In this chapter, we learned what frame rate is and how to measure it. We also learned what tasks are and how to implement asynchronous loading. We saw how to use the Visual Studio Model Editor and learned what the `.cmo` format is and how to parse and render it.

We also learned how to get inputs from different types of input devices. We learned how to use a base camera, first person camera, and third person camera for viewing the models. Also, we learned how to use XAML along with the native C++ in order to compose GUI and Direct3D.

In the next chapter, we will cover tessellation and debugging.

# 4
# Tessellation

You've already probably heard a lot about tessellation with the new features that were available on DirectX 11. In this chapter, we are going to kill two birds with one stone. First, we are going to look at some examples on which to perform tessellation step-by-step and we will take a look at the significant changes made to 3D applications by tessellation. Then, we will outline how to use the Graphics Debugging feature in Visual Studio 2012.

In this chapter, we will cover the following topics:

- What is hardware tessellation?
- Basic tessellation
- Displacement mapping with tessellation
- DirectX graphics diagnostics

By the end of this chapter, you will have gained a basic knowledge of tessellation, which is one of the most amazing features of DirectX 11 and its later versions.

## Hardware tessellation

Before DirectX 11 was introduced, you needed a software to dice polygons into smaller pieces. In order to implement this method, we needed to change the data on the CPU side and use the dynamic vertex buffers, as they require more memory compared to the static vertex buffers. This process was quite slow, but tessellation solved this problem. Now you can simply increase the triangle count in order to achieve a high-poly mesh.

# The most popular usage of hardware tessellation

Some of the popular uses of hardware tessellation are listed as follows:

- **Displacement mapping**: A displacement map is a texture that stores information on height. This technique, contrary to bump mapping, normal mapping, and parallax mapping, applies a texture input to manipulate the position of vertices on a rendered geometry. For flat surfaces or camera-facing objects, the parallax occlusion mapping is a good technique to achieve the same effect even faster. In this chapter, we will implement displacement mapping with tessellation.

- **Smoothing the model surface**: The other partner of tessellation is the refinement algorithm. A refinement algorithm, with the assistance of tessellation, creates a smoother-looking model. Furthermore, the **LOD** (**level of details**) needs tessellation in order to become dynamic. LOD amounts to the details of your environments without tessellation noticeably switching between a high-quality mesh and a low-quality mesh. With dynamic tessellation, it is easy to show a mesh to be far in distance with low details and without any jagged edges, and then when it comes closer, it is rendered with more triangles, which makes it look a lot smoother. Smoothing the jagged edges of a mesh drawn in the far clip is the main difference with the non-tessellated method.

- **Memory usage:** Do not forget that we can load a low-poly mesh and tessellate it to a high-poly mesh in order to reduce the memory usage. This might be an issue for the new game consoles released in the markets!

- **Blend shapes**: Tessellation is used in blend shapes or morph to make realistic animation, and they can even be used in the physical calculations of the high-level collision detection in patch-based, displacement mapped objects that run entirely on the GPU.

We usually write a Vertex Shader and Pixel Shader to render our scenes. In order to perform tessellation, we must add the Hull Shader and Domain Shader after the Vertex Shader.

The following image shows a schematic overview of the pipeline. According to the pipeline, the first step is the Vertex Shader. The first difference is that the triangle lists are replaced with patch lists. A patch is a primitive made up of control points. For example, for quad shapes, we have a single patch with four control points, whereas there are three control points for triangles.
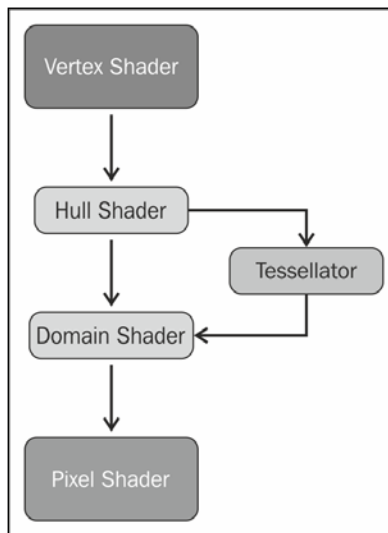


The Hull Shader is called for each output control point in order to calculate and transform the input control points into the control points that make up a patch. Also, the tessellation factors are set in this stage in order to specify how much we need to tessellate a patch.

The next stage is the Domain Shader, which is the final stage of the tessellation system. This stage is responsible for calculating the final vertex's position.

Finally, we have the Pixel Shader, which is unchanged and enables the rich shading techniques per pixel data. Note that if we would like to use the Geometry Shader, this stage is between the Domain Shader and the Pixel Shader.

Now let's start looking at how to implement the basic hardware tessellation.

# Basic tessellation

We have introduced the functionality of each stage of tessellation; now it is time to check out the first example of this chapter. Open the `Basic Tessellation` project; this example will implement the most basic tessellation patterns.

## The Hull Shader stage

Open the `TriHullShader.hlsl` file by navigating to `Assets/Shaders/Triangle`; we are going to draw our attention to the Hull Shader. The following function is a patch constant function which called by the Hull Shader once per patch.

```
HS_Const CalcHSPatchConstants(InputPatch<HS_Input,
NUM_CONTROL_POINTS> ip, uint PatchID : SV_PrimitiveID)
{
  HS_Const OUT = (HS_Const)0;
  OUT.EdgeTessFactor[0] =
    OUT.EdgeTessFactor[1] =
    OUT.EdgeTessFactor[2] = TessEdge;
    OUT.InsideTessFactor = TessInside;
  return OUT;
}
```

The `CalcHSPatchConstants` is responsible for passing the same tessellation factor to all edges and inner parts of the triangle.

The inputs of the constant Hull Shader function are the control points that are defined by `InputPatch<HS_Input, NUM_CONTROL_POINTS>`, and the patch ID value, which is an unsigned integer value defined through the `SV_PrimitiveID` semantic. In the triangle pattern, we have used three control points, so the number of `NUM_CONTROL_POINTS` is equal to three.

Inside this function, we simply pass the tessellation factors; these factors specify how finely to tessellate the current primitive. The `TessEdge` factor defines the amount of tessellation on each edge of the polygon, and the `TessInside` factor specifies the amount of tessellation for the inside of the polygon. If all tessellation factors are one, this means a primitive is not really tessellating; in this case, do not waste the GPU's time and disable tessellation. Set the tessellation factors to zero for the patch's frustum culling and Backfacing patches. When the factor is zero, the patch will not process in further tessellation stages. The following code represents the main function of the Hull Shader which outputs the control points

```
[domain("tri")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(NUM_CONTROL_POINTS)]
```

```
[patchconstantfunc("CalcHSPatchConstants")]
[maxtessfactor(64.0f)]
HS_Output main(InputPatch<HS_Input, NUM_CONTROL_POINTS> IN, uint i
: SV_OutputControlPointID, uint PatchID : SV_PrimitiveID )
{
  HS_Output OUT = (HS_Output)0;
  OUT.pos = IN[i].pos;
  OUT.color = IN[i].color;
  return OUT;
}
```

The input parameters of the Hull Shader's function are the patch of the control points, the ID of the control points, and the ID of the patch. This function outputs the control points; we can add the additional points to the input patch; hence, the output control points do not need to match the input control points. Inside the function, we simply pass the control points and the color to the Domain Shader.

There are different attributes that are used to control each tessellation stage. The following table represents the attributes that are used in this chapter:

| Name | Value | Description |
|------|-------|-------------|
| domain | • tri<br>• quad<br>• isoline | These attributes specify the patch type of the primitive being tessellated. Three primitive types can be tessellated: the triangle, the quad, and the line. |
| partitioning | • integer<br>• fractional_ even<br>• fractional_ odd<br>• pow2 | These attributes specify the tessellation factor's range.<br><br>For example, when the integer value is selected, the new vertices can be added or removed only at the integer tessellation factor value range, and the fractional part of tessellation is ignored.<br><br>The value range of integer and pow2 is from 1 to 64, that of fractional_even is from 2 to 64, and that of fractional_ odd is from 1 to 63. |

| Name | Value | Description |
|---|---|---|
| `outputtopology` | • `triangle_cw`<br>• `triangle_ccw`<br>• `line` | These attributes specify what kind of primitives we would like to deal with after tessellation.<br><br>The `triangle_cw` value means clockwise-wound triangles, the `triangle_ccw` value means a counter-clockwise winding order, and the `line` value is used for line tessellation. |
| `outputcontrolpoints` | An integer value between 0 to 32. | This attribute defines the number of output control points (per thread) that will be created in the Hull Shader. |
| `patchconstantfunc` | A constant string value. | This attribute specifies the name of the patch constant function. |
| `maxtessfactor` | An integer value between 0 to 64. | This attribute specifies the largest tessellation factor that the patch constant function will be able to produce. |

# The Domain Shader stage

The Domain Shader is like the Vertex Shader, but it is invoked once for each point generated by the fixed function tessellator. The Domain Shader receives a domain and a tessellated patch and then puts the vertices in the world space. In the body of this function, we need to calculate the final vertex position as follows:
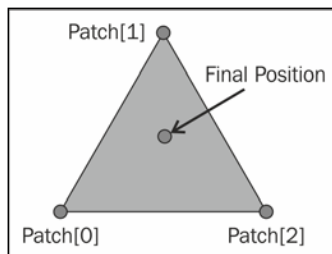


```
float4 vertexPosition =
domain.x * patch[0].pos +
domain.y * patch[1].pos +
domain.z * patch[2].pos;
```

Like with the Vertex Shader, transform this vector to the 3D space after calculating the vertex position.

The Pixel Shader returns the input color. Make sure that when you want to draw primitives, you set D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST as the primitives' topology:

```
d3dContext->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_
POINT_PATCHLIST);
```

The following figure shows the output result of the basic tessellation for triangle:
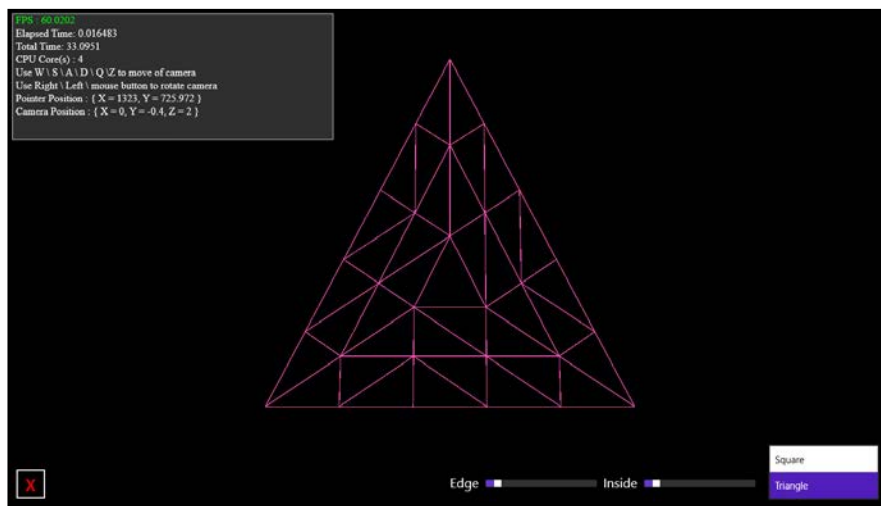


# Tessellating a quad

Tessellating a quad  in the world is similar to tessellating a triangle which we introduced in previous section, except for some changes that need to be considered.

In the Domain Shader, we should calculate the final vertex position as shown in the following figure:
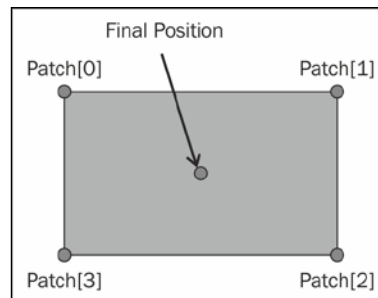
The following code shows how to calculate the vertex position of the quad:

```
float3 top = lerp(patch[0], patch[1], domain.x );
float3 bottom = lerp(patch[3], patch[2], domain.x );
float3 result = lerp( top, bottom, domain.y );
float4 vertexPosition = float4(result, 1.0f);
```

The major differences between tessellating a triangle and tessellating a quad are:

- The triangle has three points, so we assigned the value 3 to the `NUM_CONTROL_POINTS` input parameter, while in quad, we would assign the value 4, because each quad has four vertices

- The size of tessellation factors is  different for each of them.

- In a triangle, the `domain` attribute of the Hull Shader and the Domain Shader has the `tri` value, while in quad, we must use the `quad` value for this attribute.

- In a triangle, we must use `D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST`,  while in quad, we must use `D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST` as the topology for drawing primitives.

In this section, we learned how to use basic tessellation to subdivide polygons using the GPU.

We presented both the methods that are used in tessellation; the first was the triangle method and the second was the quad method. Also, you learned how to change the amount of **Edge** and **Inside** from the user interface.

This section helped us to get started by using the basic tessellation technique on a single polygon. In the next section, we are going to implement the displacement mapping using the tessellation technique.
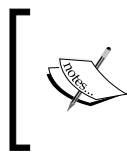
> For more information about tessellation, make sure to check the `DetailTessellation11` and `PNTriangles11` samples in the Microsoft DirectX SDK June 2010; also, take a look at the `terrain tessellation` and `PN-Patches` samples of NVIDIA SDK 11.

# Displacement mapping using tessellation

Displacement mapping is a technique for adding details to the surface of geometries. As mentioned before, it is contrary to the bump mapping technique. In bump mapping (which is known as normal mapping), we just change the normal surface to improve the light quality. In other words, each pixel in a bump map indicates the light direction of the corresponding pixel on the color map, but in displacement mapping, we are dealing with the geometries.

# The normal mapping technique

Prior to implementing the displacement mapping, we need to get familiar with normal mapping. In normal mapping, we need to create a map that is of type texture2D and store the normal vector of each pixel in RGB channels, but in displacement mapping, we need to use a height map that contains a single channel for storing the height of each pixel. The following image shows the normal vector of each vertex for our model:
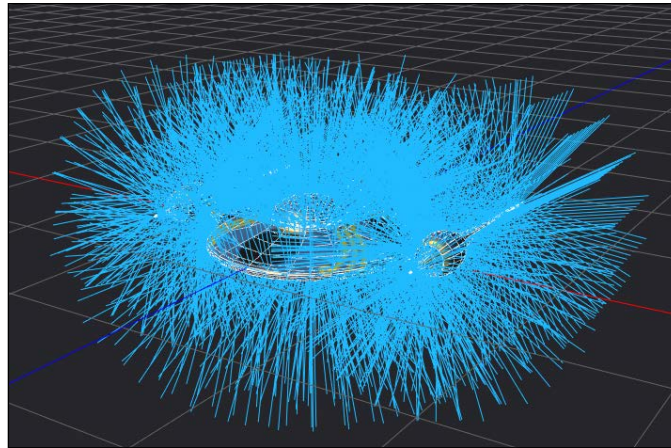


Open the `Displacement mapping` project. The first part of this project implements the normal mapping technique; the `NormalmapVertexShader.hlsl` file transforms the vertex position, vertex normal, and vertex tangent to the world space; also, the texture coordinate and the world position pass the Vertex Shader.

Open the `NormalMapPixelShader.hlsl` file by navigating to `Assets/Shaders/NormalMapping`. The main part of this shader is responsible for building the tangent space, sampling the normal map, and transforming these vectors from the tangent space to the world space:

```
IN.Normal = normalize(IN.Normal);
float3 toEye = eyePos - IN.WorldPos;
toEye = toEye / length(toEye);

float3 normalMapSample = gNormalMap.Sample(Sampler, IN.UV).rgb;

float3 normalRange = 2.0f * (normalMapSample - 0.5f);

float3 N = IN.Normal;
float3 T = normalize(IN.Tangent - dot(IN.Tangent, N) * N);
float3 B = cross(N, T);
float3 bump = mul(normalRange, float3x3(T, B, N));
```

In the first few lines of our code, the eye vector is calculated in order to be used in lighting later on; sample the normal vector and store it in the `normalMapSample` variable.

We need to bias and double a bump value to change it from the [0...1] to [-1...1] range, and then we store it in the `normalRange` variable. Finally, we build the tangent space from the input tangent and normal vectors and transform the tangent space to the world space.

# The displacement mapping technique

After an introduction to normal mapping, we can combine this effect with the displacement mapping and the tessellation technique. We are going to displace the geometries inside the tessellation stages according to the tessellation factors and the height-scale parameters.

First of all, open the `LandNORM.dds` file by navigating to `Assets\Models` in the same project folder. This is the normal map file, which is created in Photoshop:
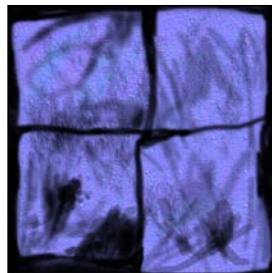


This file contains the normal vector of each pixel, but some parts of this image are shown in black. The black values represent the lowest height. In other words, the alpha channel of this pixel shows the height value.

Now open the `DispMapVertexShader.hlsl` file by navigating to `Assets\Shaders\DisplacementMap`. The main change to be made starts from here; we do not use a static value for the tessellation factor. Instead, we are going to calculate it based on the minimum and maximum factors and the distance between the camera position and the vertex position in the world, because closer triangles need more of tessellating.

```
HS_In main( VS_In IN )
{
```

```
    HS_In OUT = (HS_In)0;
    OUT.Pos     = mul(float4(IN.Pos, 1.0f), World);
    OUT.Normal  = mul(IN.Normal, (float3x3)WorldInvTranspose);
    OUT.Tangent = mul(IN.Tangent, (float3x3)World);
    OUT.UV = IN.UV;
    float Lenght = MinTessDistance - MaxTessDistance;
    float tess = saturate( (MinTessDistance –
distance(OUT.Pos, eyePos)) / Lenght );
    OUT.TessFactor = MinTessFactor + tess *
( MaxTessFactor - MinTessFactor);
    return OUT;
}
```

An important part of the displacement mapping occurs in the Domain Shader. When we pass from the tessellation factors and the patch of the control points to the Domain Shader, we need to get the height of each pixel from the sampled vectors of the normal map. As the following code shows, actually, we just need the alpha channels. As mentioned before, an alpha channel is the height value, so we can calculate the new position of the vertex in the world matrix from this height value:

```
    float height = NormalMap.SampleLevel(Sampler, OUT.UV, mipLevel).a;
    OUT.WorldPos += (height - 1.0f) * Height * OUT.Normal;
    OUT.Pos = mul(float4(OUT.WorldPos, 1.0f), ViewProjection);
```

# DirectX graphics diagnostics

Debugging a captured frame is usually a real challenge in comparison with debugging C++ code. We are dealing with hundreds of thousands of more, pixels that are produced, and in addition, there might be several functions being processed by the GPU. Typically, in modern games, there are different passes on a frame constructing it; also, there are many post-process renderings that will be applied on the final result to increase the quality of the frame. All these processes make it quite difficult to find why a specific pixel is drawn with an unexpected color during debugging! As mentioned in the previous chapter, Visual Studio 2012 comes with a series of tools that intend to assist game developers. The new DirectX graphics diagnostics tools are a set of development tools integrated with Microsoft Visual Studio 2012, which can help us to analyze and debug the frames captured from a Direct3D application. Some of the functionalities of these tools come from the PIX for a Windows tool, which is a part of DirectX SDK.

> Please note that the DirectX graphics diagnostics tools are not supported by Visual Studio 2012 Express at the time of writing this book.

In this section, we are going to explain a complete example that shows how to use graphics diagnostics to capture and analyze the graphics information of a frame. Open the final project of this chapter, `DirectX Graphics Diagnostics`, and let's see what is going on with the GPU.

> **Intel Graphics Performance Analyzer** (**Intel GPA**) is another suite of graphics analysis and optimization tools that are supported by Windows Store applications. At the time of writing this book, the final release of this suite (Intel GPA 2013 R2) is able to analyze Windows 8 Store applications, but tracing the captured frames is not supported yet. Also, Nvidia Nsight™ Visual Studio Edition 3.1 is another options which supports Visual Studio 2012 and Direct3D 11.1 for debugging, profiling, and tracing heterogeneous compute and graphics applications

# Capturing the frame

To start debugging the application, press *Alt + F5* or select the `Start Diagnostics` command from **Debug** | **Graphics** | **Start Diagnostics**, as shown in the following screenshot:
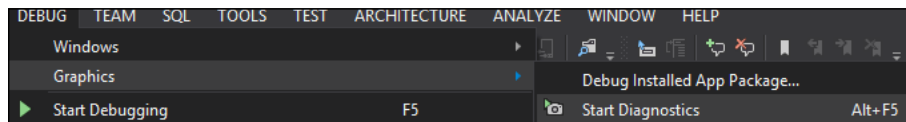


You can capture graphics information by using the application in two ways. The first way is to use Visual Studio to manually capture the frame while it is running, and the second way is to use the programmatic capture API. The latter is useful when the application is about to run on a computer that does not have Visual Studio installed or when you would like to capture the graphics information from the Windows RT devices.

For in the first way, when the application starts, press the Print Screen key (*Prt Scr*).

For in the second way, for preparing the application to use the programmatic capture, you need to use the CaptureCurrentFrame API. So, make sure to add the following header to the `pch.h` file:

```
#include <vsgcapture.h>
```

For Windows Store applications, the location of the `temp` directory is specific to each user and application, and can be found at `C:\users\username\AppData\Local\Packages\package family name\TempState\`.

Now you can capture your frame by calling the `g_pVsgDbg->CaptureCurrentFrame()` function. By default, the name of the captured file is `default.vsglog`.

Remember, do not start the graphics diagnostics when using the programmatic capture API, just run or debug your application.

# The Graphics Experiment window

After a frame is captured, it is displayed in Visual Studio as `Graphics Experiment.vsglog`. Each captured frame will be added to the Frame List and is presented as a thumbnail image at the bottom of the **Graphics Experiment** tab. This logfile contains all the information needed for debugging and tracing. As you can see in the following screenshot, there are three subwindows: the left one displays the captured frames, the right one, which is named **Graphics Events List**, demonstrates the list of all DirectX events, and finally, the **Graphics Pixel History** subwindow in the middle is responsible for displaying the activities of the selected pixel in the running frame:
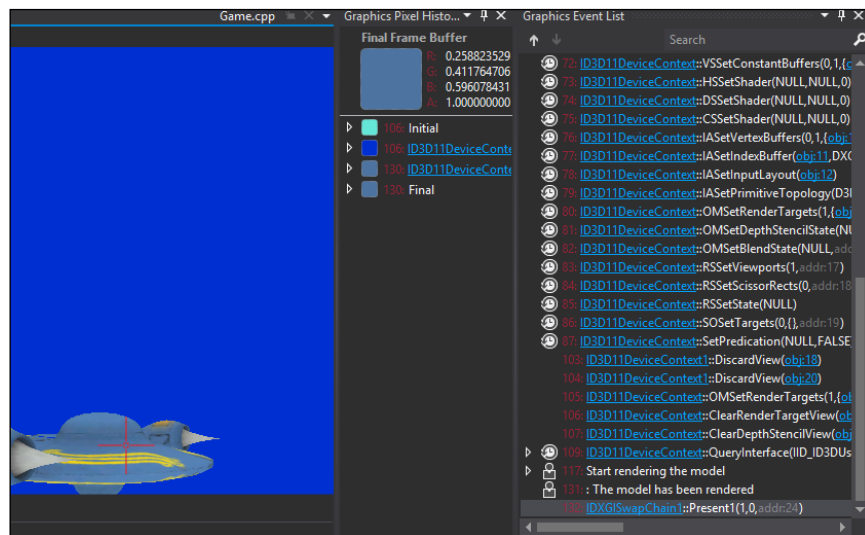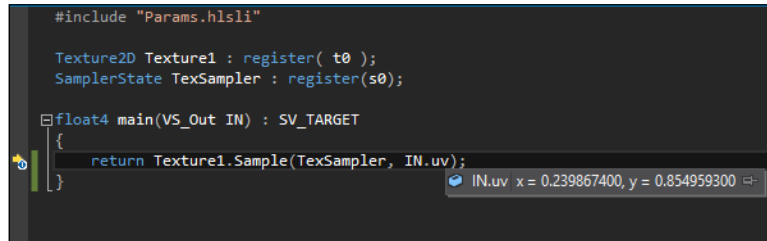
Let's start with the **Graphics Pixel History** subwindow. As you can see in the preceding screenshot, we selected one of the pixels on the spaceship model. Now let us take a look at the **Graphics Pixel History** subwindow of that pixel, as shown in the following screenshot:
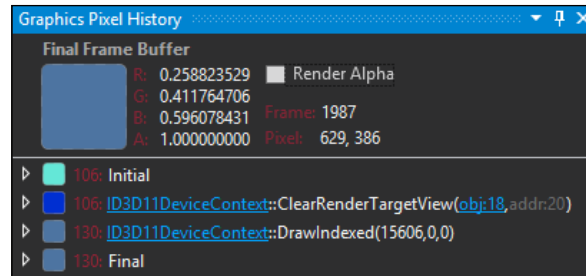


The preceding screenshot shows how this pixel has been modified by each DirectX event; first it is initialized with a specific color, then it is changed to blue by the `ClearRenderTargetView` function, and after this, it is changed to the color of our model by drawing indexed primitives. Open the collapsed `DrawIndexed` function to see what really happens in the Vertex Shader and Pixel Shader pipelines. The following screenshot shows the information about each of the vertices:
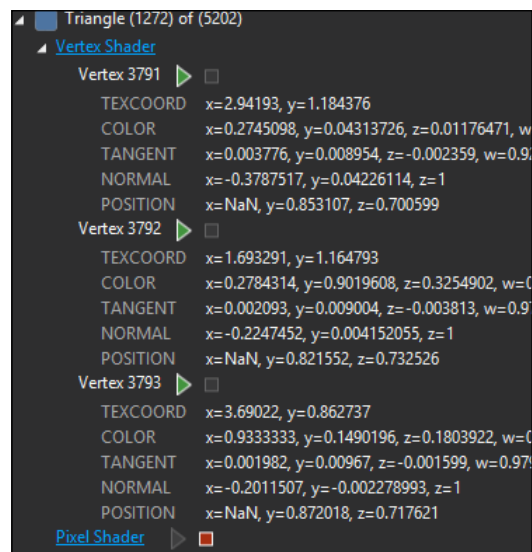
As mentioned in the previous chapter, the input layout of the vertex buffer is VertexPositionNormalTangentColorTexture. In this section, you can see each vertex's value of the model's triangle. Now, we would like to debug the Pixel Shader of this pixel, so just press the green triangular icon to start debugging. As you can see in the following screenshot, when the debug process is started, Visual Studio will navigate to the source code of the Pixel Shader:
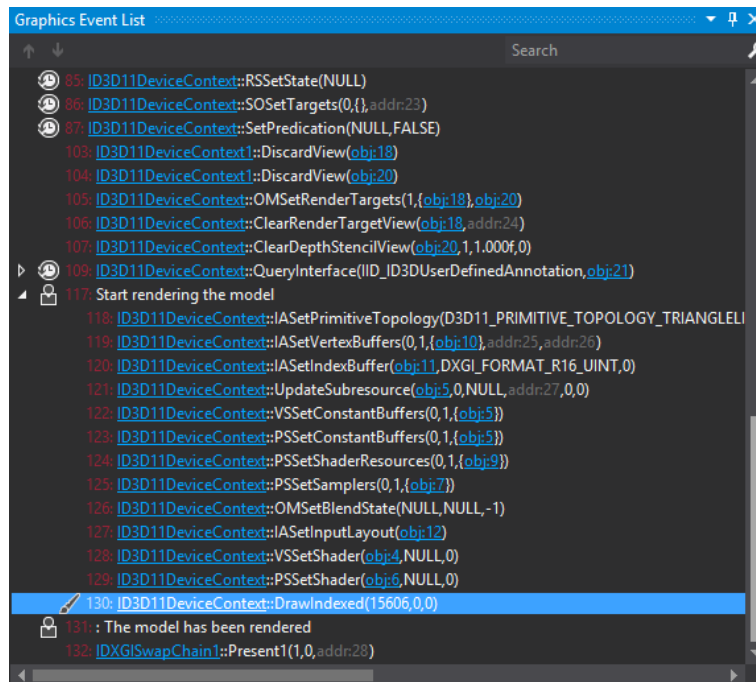


Now you can easily debug the Pixel Shader code of the specific pixel in the DrawIndexed stage. You can also right-click on each pixel of the captured frame and select **Graphics Object Table** to check the Direct3D object's data.

The following screenshot shows the **Graphics Event List** subwindow. Draw calls in this list are typically more important events:

The event that is displayed with the ✎ icon marks a draw event, the one that is displayed with the ⏱ icon marks an event that occurs before the captured frame, and the user-defined event marker or the group shown with the 👤 icon can be defined inside the application code. In this example, we mark an event (`Start rendering the model`) before rendering the model and mark another event (`The model has been rendered`) after the model is rendered. You can create these events by using the `ID3DUserDefinedAnnotation:: BeginEvent`, `ID3DUserDefinedAnnotation:: EndEvent`, and `ID3DUserDefinedAnnotation:: SetMarker` interfaces. At the end of this chapter, we will be familiar with the `ID3DUserDefinedAnnotation` interface.

# Investigating a missing object

This section will demonstrate how to investigate an error that occurs in the Vertex Shader stage so we can find out why an object is missing from our scene. First we need to find the draw call for the missing geometry. Select the Draw Event from the event list then select Graphics Pipeline Stage by navigating to **Debug** | **Graphics** | **Pipeline Stages**.
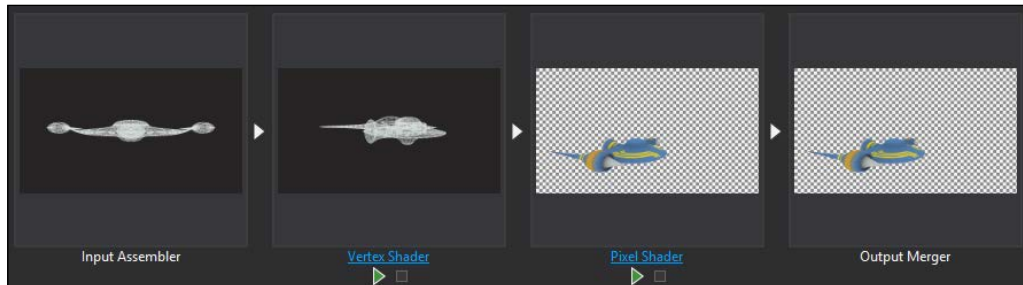


First **select Input Assembler** and view the object's geometry inside the Model Editor. If we make sure that it is the missing geometry, we must find out why it is not shown in our scene. The following are some of the common reasons:

- **Incompatible input layout:** When you click on the `DrawIndexed` event, you will see a window that shows the device context information. Make sure the input element description of the vertex buffer is the same as the Vertex Shader input structure.

- **Error in the Vertex Shader code:** The input assembler provides the right data for the Vertex Shader, so we need to see what is going on inside the Vertex Shader. Use the HLSL debugger to examine the state of variables in the Vertex Shader, especially for the output position of the Vertex Shader.

- **Error in the application source code:** If neither of the previous two solutions solve the rendering issue, then we must find out where the constant buffer is being set in the source code of the program. Open the **Graphics Event Call Stack** window by navigating to **Debug | Graphics | Event Call Stack** and check the source code.

- **Let's check the device state**: Open the Graphics Object Table tool and examine the device state, especially the depth stencil view, to see whether the primitives have failed the depth test or not.

# Disabling graphics diagnostics

Before publishing your game, make sure to disable the graphics diagnostics in the application. It is obvious that you strongly need to prevent malicious users from accessing your game assets before your game is released. Fortunately, in DirectX 11.1, there are two simple ways to do this. The first way is to use the `ID3DUserDefinedAnnotation::GetStatus` function, which returns true if the application is being observed by a tool such as graphics diagnostics. Then, you can easily perform any action that you prefer and then call the `exit` function to shut down the application.

The second way is to set the `D3D11_CREATE_DEVICE_PREVENT_ALTERING_LAYER_SETTINGS_FROM_REGISTRY` flag when you create your Direct3D device.

In the `game.cpp` file, when the device context is created, we can get the `ID3DUserDefinedAnnotation` interface and call `GetState` to check whether the application is being observed by another tool or not:

```
hr = context.As(&GDevice.d3dContext);
hr = GDevice.d3dContext.Get()->QueryInterface(
__uuidof(ID3DUserDefinedAnnotation),
(void**)(&gPerf));
if (this->gPerf->GetStatus())
{
OutputDebug(L"The app is being observed by another tool");
}
```

In this final section of the chapter, we were introduced to the DirectX graphics diagnostics tools. These tools help us to determine exactly where the application bottleneck is; they also allow us to watch the workflows of GPU processes and perform a complete system trace on these processes.

# Summary

In this chapter, we learned what tessellation is and why we must use it. We then focused on how to perform basic tessellation and implemented the displacement mapping with the tessellation technique. Also, we learned the fundamentals of the DirectX graphics diagnostics to debug and carry out a diagnosis of the 3D Metro Style application.

In the next chapter, we will outline how to have a multithreaded framework with the C++AMP library and the Compute Shader stage.

# 5

# Multithreading

Today, most computers possess multiple cores within their processors. The CPU and GPU core counts will continue to increase, and in a few years, many applications and tools will be developed to utilize these hardware improvements efficiently. In this chapter, we are going to demonstrate how to improve the framework for a parallel game engine using the new technology of Microsoft, which is called C++ Amp. We are also going to integrate our engine to use Compute Shaders and then compare their performances.

In this chapter, we will cover the following topics:

- C++ AMP
- Compute Shaders
- Compute Shader versus C++ AMP
- Post-processing

By the end of this chapter, we are going to have a multithreaded game engine. We will also learn when it is necessary to use C++ AMP and when we must use the Compute Shader.

## C++ AMP

**C++ AMP** (**C++ Accelerated Massive Parallelism**) is a small extension library that enables heterogeneous computing, which provides developers with the **GPGPU** (**General Purpose GPU**) programming for making use of the GPU.

C++ AMP fairly looks like an extension. It is C++, not C, and is used to implement data parallelism directly in C++. It also allocates interoperability with Direct3D for accelerating your code. If the code cannot be run on the GPUs, it will fall back onto the CPUs. This means that C++ AMP is a cross-platform library.

Taking over the runtime errors, you need a debugger to visualize the threads and the value of each buffer; Visual Studio 2012 fully supports C++ AMP. You can run and debug your code on Windows 7, Windows 8, Windows Server 2008 R2, and Windows Server 2012. Debugging on the software emulator is only supported on Windows 8 and Windows Server 2012.

In order to have a multithreaded game engine, our framework must be redesigned to use as many processors that are available within a GPU and CPU. Some parts of our framework that will take this approach are post-processing and the real-time collision detection.

The best way to begin learning C++ AMP is to start from the beginning by creating a simple example application. Now we'll take you step by step through the process of your first C++ AMP application.

> To run the C++ AMP code, a DirectX 11 feature level 11.0 or a later version is needed. Also, drivers of your graphics card must be installed for debugging.

We are going to compare the performance of C++ AMP with more traditional methods. Our example will demonstrate how to use C++ AMP to accelerate the execution of vector multiplication. Open the first project of this chapter, which is named `Array`, then take a look at the `Test.h` header, which is located in the `Scenes` folder of the solution.

The algorithm is simple; in the first function, we are going to execute a sequence for the process for these vectors. It does not use any parallel or threaded algorithms to reduce the computation time:

```
for (UINT i = 0; i < size; i++)
{
  for (UINT j = 0; j < size; j++)
  {
    C[i] += A[i] * B[j];
  }
}
```

This way will take a long time to execute for large size vectors; let's write this function in another way using the `std` library:

```
std::for_each(A.begin(), A.end(), [&](float a)
{
  std::for_each(B.begin(), B.end(), [&](float b)
  {
```

```
    C[i] += a * b;
  });
  i++;
});
```

The `std::for_each` loop is very flexible and is faster than other traditional ways; comparing the results will prove this. Now let's implement this algorithm with the `concurrency::parallel_for` loop. It iterates over a range of indices and executes a user-supplied function per iteration in parallel.

```
parallel_for(0, size, [&](int i)
{
  parallel_for(0, size, [&](int j)
  {
    C[i] += A[i] * B[j];
  });
});
```

OpenMP is available on Windows Store applications and on Visual Studio 2012. OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express the shared memory parallelism. At the time of writing this book, Visual Studio 2012 supports OpenMP 2.0. If you would like to use this feature for your C++ compiler, go to the **Properties** window of the project, navigate to **C/C++** | **Language**, and from the **Language** tab, choose the **Yes** option for OpenMP support. The following screenshot shows the way you can enable OpenMP:
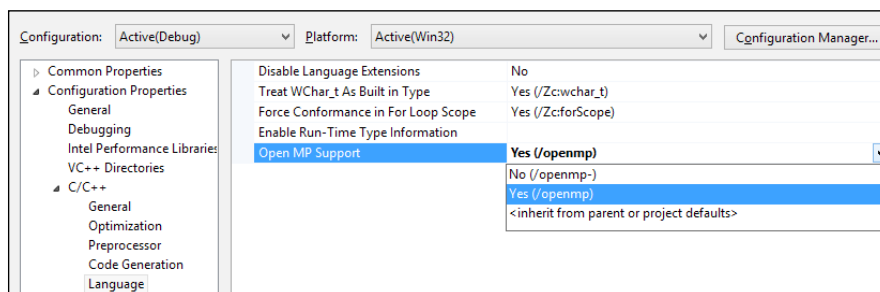


OpenMP makes the multithreading process much easier. In OpenMP, `#pragma omp parallel for` divides the loop iterations between the spawned threads:

```
#pragma omp parallel for
for (int i = 0; i < size; i++)
{
  for (int j = 0; j < size; j++)
  {
```

```
      C[i] += A[i] * B[j];
   }
}
```

As you can see, all those functions do the same job, but the first one does it sequentially while the others do it in parallel modes. Now let's implement this function by applying the C++ AMP:

```
array_view<const float, 1> _A(size, A);
array_view<const float, 1> _B(size, B);
array_view<float, 1> _C(size, C);
parallel_for_each(_A.extent, [=](index<1> idx) restrict(amp)
{
  for (int i = 0; i < size; i++)
  {
    _C[idx] += _A[idx] * _B[i];
  }
});
_C.synchronize();
```
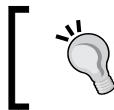
> Prior to coding with C++ AMP, make sure to use the `concurrency` namespace, which is included within the `amp.h` header.

As you can see, the code was not changed much. `array view<const float, 1>` is a dimension float array that is responsible for copying the data from a vector to an accelerator. The `parallel_for_each` method provides the mechanism for iterating through the data elements and runs a function across the compute domain.

We used this function to multiply to each one of the elements in `array_view` variables, and we used `restrict(amp)` keyword in order to use an accelerator-compatible code.

Let's see what really happens in the background. First, the CPU transfers the data from RAM to **VRAM** (**Video Random Access Memory**) by initializing the `array_view` variable. Then the GPU starts to process the data by calling `parallel_for_each` in a parallel mode.

The result is stored in the `_C` variable, so we need to call `_C.synchronize()` to transfer the data back to the CPU. Actually, in this way, the running thread is blocked until the data transfer is completed. If you would like to prevent the blocking, you must call `_C.synchronize_async()`.

In the next section, we will discuss the Compute Shader concept and execute the vector multiplication on it to compare the result times.

# Compute Shader

This section will be devoted to the new general purpose of the computation stage. We are going to introduce this interesting new processing concept and study the threading architecture of this stage.

Compute Shader is a programmable shader stage that can read and write to the GPU resources. As you can see in the following figure, Compute Shader is fundamentally different from the other pipeline stages since it works beside the other stages. Therefore, it does not explicitly have an input or output parameter for the previous or next stage.

This technology is commonly useful for the GPGPU programming; however, there are many graphical effect techniques that can be implemented on it. Also, it is not limited to graphical applications, since some algorithms such as animation, physic, AI, and compression can be implemented with Compute Shaders.
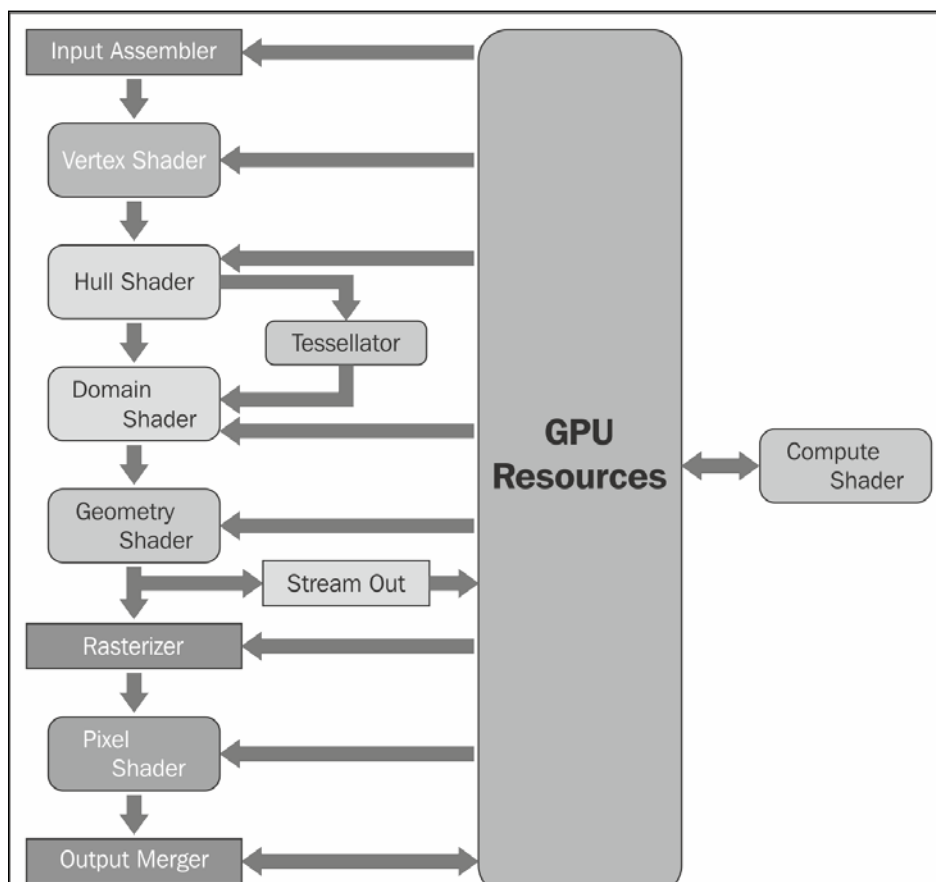
In the GPGPU programming, the computation results must be transferred back from the GPU memory (VRAM) to the CPU memory (RAM). Like the preceding example, the array that was stored in RAM must wait to be synchronized with the GPU array that was stored in VRAM. Assuming that the amount of data is huge or the time of calculation might be larger than a frame time, the CPU must wait for the GPU since it has been observed that this might cause a bottleneck for your graphical application. In the Compute Shader, we typically use the output results of the computation as the input of the rendering pipeline; in short, the transformation will be done just inside the GPU.

In the next section, we will take you step-by-step through the process of your first Compute Shader application example.

# C++ AMP versus Compute Shader

In many technologies, it seems that Compute Shader outperforms C++ AMP, but this is not a sufficient reason to throw C++ AMP away and just focus on Compute Shader to accelerate the computation . Let's start comparing the performances of these technologies; it is an efficient way to find out when it is required to use C++ AMP and when we must switch to Compute Shader:

- Never use C++ AMP when your result's data must be an input of another pipeline stage. In this case, switch to Compute Shader; this is an efficient way when you need to use the results of Compute Shader's process just inside the pipeline stages. On the other hand, C++ AMP needs to transfer the data back to the CPU; however, it cannot access the pipeline stages. Assuming that the relative memory bandwidth speed between the CPU and the GPU is 1 GB/s, the difference between the bandwidths of the GPU and VRAM and that of the CPU and RAM is quite large. In this case, the bandwidth speed is 10 GB/s for the CPU and RAM in comparison to 100 GB/s for the GPU and VRAM.

- Never use the Compute Shader if your application has to run on a machine that might not have hardware-detected DirectX 11-compatible GPU. In this case, switch to C++ AMP; AMP always supports the use of the accelerator class to check whether the target machine has the hardware with a DirectX 11 driver or not. You can also decide to use a WARP accelerator, which will run your code on the CPU, taking advantage of multiple cores. The WARP accelerator is only a Windows 8 solution and also supports the DirectCompute API.

In this section, we are going to demonstrate how to use the Compute Shader to accelerate the execution of vector multiplication and finally compare the result times. Open the same project (`Array`) and then take a look at the `ComputeShader.hlsl` file that is located in the `Assets/Shaders` folder. The Compute Shader code is as follows:

```
StructuredBuffer<float> A;
StructuredBuffer<float> B;
RWStructuredBuffer<float> C;

[numthreads(32, 1, 1)]
void main( uint3 DTid : SV_DispatchThreadID )
{
  for(int i=0; i < 5024; ++i)
  {
    C[DTid.x] += A[DTid.x] * B[i];
  }
}
```

A basic element of the Compute Shader is a thread. The Compute Shader declares the number of threads to operate on them; these threads are divided up into thread groups. Each thread group is executed on a single multiprocessor of the GPU. The `numthreads(x, y, z)` attribute defines that each thread group has a total of `x*y*z` threads. The following image shows the visualization of threads within one thread group:
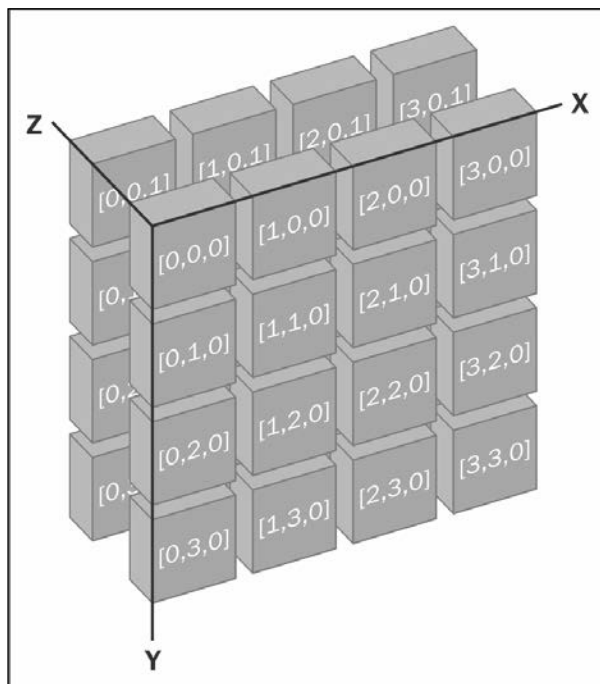
The preceding image shows a single thread group. This thread group has 32 threads (4 * 4 * 2 = 32), which is specified by `numthreads(4, 4, 2)`.

We can set the number of thread groups by calling `D3D11DeviceContext::Dispatch`. The input parameters of this function are three unsigned integer numbers that provide the three-dimensional array of the group threads. If any of the values of the input parameters is 0, the driver's `Dispatch` function does nothing.

The following diagram shows the visualization of thread groups within the `Dispatch` call:
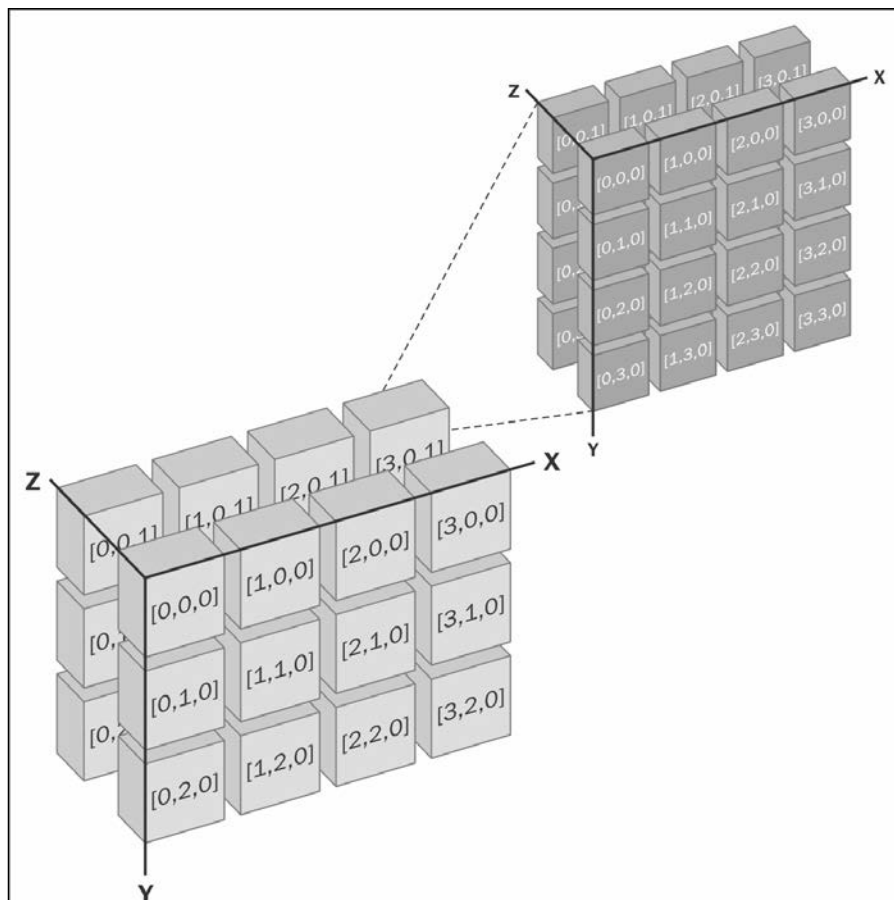


According to the preceding image, we have `Dispatch(4,3,2)`; actually, 24 thread groups (4*3*2 = 24) will be created and each thread group has 32 threads specified by (`numthreads(4,4,2)`), so the total number of threads is 768 (that is, 24 * 32 =768).

> In the Shader Model 5, the maximum number of threads can be 1024 (x*y*z < = 1024). The z component of the `numthreads` attribute must be a value between 1 to 64, and the x and y components must be greater than or equal to one.

The semantics that are attached to a Compute Shader input or output convey the following information:

- `SV_DispatchThreadID`: This semantic refers to the 3D identifier within the entire Compute Shader thread stage. This ID is uniquely an integer and indices to the specific thread (indexing by x, y, and z components).

- `SV_GroupID`: This semantic refers to the ID of a thread group within a dispatch.

- `SV_GroupThreadID`: This semantic indicates to the ID of the thread within its own thread group.

- `SV_GroupIndex`: This semantic refers to a flattened index of a thread within a thread group.

As you can see, the algorithm of our Compute Shader code is the same as the algorithm that was used in the C++ AMP code. A and B are the input arrays, and the results of the calculation will be stored in the C array. The input array is defined as a structured buffer, which is simply an array of elements; this buffer can be implemented in the same manner as a simple buffer, except for some properties of its description. Navigate to `Graphics/Shaders` of the project, open the `StructuredBuffer.h` header file, and see the definition of the `Load` method:

```
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.ByteWidth = (size of element) * (number of elements);
bufferDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;
bufferDesc.StructureByteStride = (size of element);
bufferDesc.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
```

The C array behaves like the structured array, except that it also provides the ability of writing. The only difference in creating this buffer is `BindFlags`, which must be set to `D3D11_BIND_UNORDERED_ACCESS`.

The structured buffer can be bound as an **SRV** (**shader resource view**); it is needed to create a shader resource view object by calling the `ID3D11Device::CreateShaderResourceView` function.

```
D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc;
srvDesc.Format = DXGI_FORMAT_UNKNOWN;
srvDesc.ViewDimension = D3D11_SRV_DIMENSION_BUFFEREX;
```

```
srvDesc.BufferEx.FirstElement = 0;
srvDesc.BufferEx.Flags = 0;
srvDesc.BufferEx.NumElements = number of elements;
```

By calling the `ID3D11Device::CreateUnorderedAccessView` function, we can create a UAV in order to bind it to `RWStructuredBuffer`. In DirectX 11.1, unlike the previous version, a large number of **UAVs** (**Unordered Access Views**) can be created, and these UAVs can be used in all pipeline stages:

```
D3D11_UNORDERED_ACCESS_VIEW_DESC uavDesc;
uavDesc.Format = DXGI_FORMAT_UNKNOWN;
uavDesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
uavDesc.Buffer.FirstElement = 0;
uavDesc.Buffer.Flags = 0;
uavDesc.Buffer.NumElements = number of elements;
```

As mentioned earlier, in the GPGPU computing, we might often get the result from VRAM back to RAM. We need to create a system buffer that can be accessed from the CPU with the `D3D11_USAGE_STAGING` usage flag:

```
D3D11_BUFFER_DESC bufferDesc;
bufferDesc.Usage = D3D11_USAGE_STAGING;
bufferDesc.BindFlags = 0;
bufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
```

Within the `Synchronize` method, the `ID3D11DeviceContext::Map` function is used to copy the results of the Compute Shader from the GPU memory to the system memory; also, the `ID3D11DeviceContext::Unmap` function must be called when the system memory is done updating.

> In order to optimize the performance of your application, avoid copying the resources from the GPU to the CPU for each frame; this might decrease the frames per second of your graphical application.
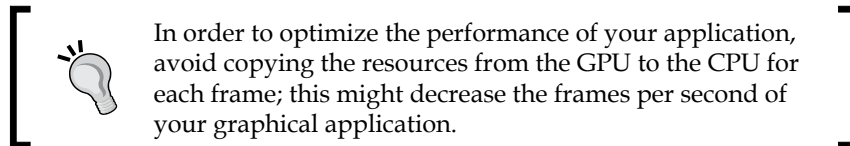
The following screenshot shows the results for the array with size of 5024 float numbers for each array on the target machine with the following features:

- CPU: Intel® Core™ i5-2450M CPU @ 2.50 GHz 2.49GHz
- RAM: 6.00 GB
- System type: 64-bit operating system
- Graphics card: GeForce GT 525M
- Dedicated video memory: 1024 MB DDR3

```
FPS : 60.0692
Elapsed Time: 0.0169545
Total Time: 15.8711
CPU Core(s) : 4
Waiting for the sequential...
Load time in the sequential is 4.36315 (s).
Waiting for the C++AMP...
Load time in the C++AMP is 0.0984551 (s).
Waiting for the standard parallel...
Load time in the standard parallel is 2.15103 (s).
Waiting for the ppl parallel...
Load time in the ppl parallel is 5.39204 (s).
Waiting for the OpenMP...
Load time in the OpenMP is 2.08069 (s).
Waiting for the ComputeShader...
Load time in the ComputeShader is 0.0506632 (s).
```

The total time of execution with C++ AMP is less than 99 milliseconds, while the total time of the Compute Shader is around 50 milliseconds, which seems amazing. Try it on your own machine; build and run the project and compare the result time of the Compute Shader with the result time of C++ AMP.

# Post-processing

A post-processor takes in an image and runs some techniques on that image, such as blurring, blooming, negative, and so on. Performing the post-processing for rendering a scene is required to increase the quality of the output presented to the user.

In this section, we are going to introduce an approach that will show you where to use both C++ AMP and Compute Shader to implement some post-processing techniques in our framework.

# Implementing post-processing using C++ AMP

Open the `Post Processing` project from the source code. Open the `Quad.cpp` file from the `Graphics/Models` folder and find the definition of the `Load` method of this class; we will try to apply a post-process on the texture of the quad.

We need to create an empty texture on which we can write our processed result pixels. This resulting texture will have the same height and width as the original texture, and also it is required to create an SRV from this texture. Let's call this SRV `processedResourceView` The following code shows the texture description of our texture:

```
D3D11_TEXTURE2D_DESC desc = {0};
desc.Height = height;
desc.Width = width;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_UNORDERED_ACCESS |
    D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;
```

Set one to `mipLevels`, because C++ AMP can only access the first MIP map level of the texture. The `D3D11_BIND_SHADER_RESOURCE` bind flag is set to allow reading to the texture, and the `D3D11_BIND_UNORDERED_ ACCESS` flag is set to allow writing to the texture.

C++ AMP supports `texture<T, N>` in the `concurrency::graphics::direct3d` namespace, which is defined in the `amp_graphics.h` file, to map the texture in C++ AMP to an exact DXGI format.

It is still impossible to both read and write to this type of `texture<T, N>`; therefore, C++ AMP provides the `write_only_texture_view<T, N>` type to allow writing to the texture.

By calling the `make_texture()` method, we can create a texture associated with our texture2D resource; the return value is stored in `amp_textureView`:

```
auto writeTexture = make_texture<unorm4, 2>(
accViewObj, texture.Get());
amp_textureView =
unique_ptr<writeonly_texture_view<unorm4, 2>>(
new writeonly_texture_view<unorm4, 2>(writeTexture));
```

The main section of the post-processing in C++ AMP occurs in the `AMPPostProcess` method. In this method, first we specify which post-processing technique is required, then we create an input texture associated with our texture2D resource:

```
auto input_tex = make_texture<unorm4, 2>(
    accViewObj, texture2D->Texture);
auto processedView = *this->amp_textureView.get();
```

The `accViewObj` is an `extern` object that is a type of `accelerator_view`. We have created an accelerator view from the D3D device interface pointer by the `concurrency::direct3d::create_accelerator_view` function right after creating the Direct3D device in the `Game.cpp` file.

Now send the input texture captured by the value to the `parallel_for_each` function. The following is the body of your C++ AMP code. First add the `fast_math` namespace to save some typing and implement the negative, which simplifies inverting each pixel:

```
parallel_for_each(input_tex.accelerator_view,
 processedView.extent,
 [ = , &input_tex ](index<2> idx) restrict(amp) {
  using namespace fast_math;

  auto Negative = [=] (unorm4 _in) restrict(amp) -> unorm4
  {
    auto rgb = static_cast<unorm3>(1) - _in.rgb;
    return unorm4( rgb.r, rgb.g, rgb.b, 1.0f);
  };
  auto color = input_tex[idx];
  if (hasNegative)
  {
    color = Negative(input_tex[idx]);
  }
```

In the following technique, we are going to make each pixel lighter using the array of `weights`:

```
  if (hasLighter)
  {
      const float weights[3] = { 0.05f, 0.1f, 0.2f };
      for(int i = 0; i < 3; i++)
      {
      color.rgb += color.rgb * unorm3(weights[i]);
      }
  }
```

Finally, we have a technique that applies a separate mathematical calculation on the color of odd and even pixels:

```
if (hasOddEven)
{
  const float _CONST0 = 0.5f;
  const float _CONST1 = 10.0f;

  if (idx[0] % 2 != 0)//if is Odd
  {
    color.rgb += unorm3(sin(color.r * _CONST1) *
                  _CONST0, sin(color.g * _CONST1) * _CONST0,
                   sin(color.b * _CONST1) * _CONST0);
  }
  else
  {
    color.rgb += unorm3(cos(color.r * _CONST1) *
                  _CONST0, cos(color.g * _CONST1) * _CONST0,
                   cos(color.b * _CONST1) * _CONST0);
  }
}
```

After modifying the pixels, we need to save the result to the processed texture view:

```
    processedView.set(idx, color);
  });
```

Finally, in the `Render` method, we set the processed SRV to the Pixel Shader:

```
    shader->BindSRV(ShaderType::PixelShader ,0, 1,
          processedResourceView.GetAddressOf());
```

We need to unbind the SRV from the Pixel Shader and clean the pipeline; this will happen when `EndApply` is called:

```
  ID3D11ShaderResourceView * NullSRV = nullptr;
  GDevice.d3dContext->PSSetShaderResources(0, 1, &NullSRV);
```

Within this section, we implemented some post-processing techniques using the C++ AMP library. In the next section, we are going to implement post-processing using the Compute Shader.

# Implementing post-processing using Compute Shader

In this section, we are going to implement the post-processing techniques described in the previous section with the help of the Compute Shader.

Open the `ComputeShader.hlsl` file from the `Assets/Shaders` folder. The code is given as follows:

```
#define N 256
[numthreads(N, 1, 1)]
void main(uint3 DTid  : SV_DispatchThreadID)
{
  float4 color = T[DTid .xy];
  if (PixelState.x == 1)
  {
    //Apply negative
  }
  if (PixelState.y == 1)
  {
    //Apply High Lighter
  }
  if (PixelState.z == 1)
  {
    //Change odd and even pixels
  }
  RWT[DTid.xy] = color;
}
```

The implementation of post-processing techniques using the Compute Shader is similar to the C++ AMP implementation, so we will leave them and focus on the main parts of this shader code. In the first line of the code, we accessed the current pixel of the `T` texture with an integer index to the specific running thread. The `T` variable was declared as a `Texture2D` type.

When the resultant color is modified, we need to store it to a UAV type object. We declared this object as a `RWTexture2D<float4>` type, which is a readable and writeable resource unlike the `T` variable, which is declared as a `Texture2D` type, and can only be a read-only texture.

Now look at the `CSPostProcess` method from the `Qaud.cpp` file. First, we set which post-process is required to be applied by setting `PixelState`:

```
ObjectInforVars.Const.PixelState.x = negative ? 1.0f : 0.0f;
ObjectInforVars.Const.PixelState.y = highLighter ? 1.0f : 0.0f;
ObjectInforVars.Const.PixelState.z = oddEven  ? 1.0f : 0.0f;
ObjectInforVars.Update();
shader->SetConstantBuffer(0, 1, ObjectInforVars.Buffer);
```

Actually, to bind `RWTexture2D<float4>`, an SRV and a UAV are needed. As mentioned before, the resource can be accessed by the Compute Shader for the input by creating an SRV to the texture; we bind it to the Compute Shader by calling the `BindSRV` method of the `shader` class:

```
shader->BindSRV(ShaderType::ComputeShader, 1, 1, rwTexture->SRV);
```

The resource can be accessed by the Compute Shader for the output parameter. By creating a UAV to the texture, we can also bind it by calling `BindUAV`:

```
shader->BindUAV(0, 1, rwTexture->UAV);
```

> In each draw frame, it is more efficient to perform the Compute Shader before performing the render.

Now it's time to set the number of group threads and call the `Dispatch` method to execute the commands in the Compute Shader. Make sure to unbind the SRV and UAV when the Compute Shader is done updating, because a resource cannot be both an output and input at the same time:

```
UINT groupThreadsX = static_cast<UINT>(
        ceilf(texture2D->Width / 256.0f));
shader->Dispatch(groupThreadsX, texture2D->Height, 1);
shader->UnbindSRV(ShaderType::ComputeShader, 1, 1);
shader->UnbindUAV(0, 1);
```

Finally, the result must be set to the Pixel Shader:

```
shader->BindSRV(ShaderType::PixelShader, 0, 1, rwTexture->SRV);
```

# Summary

In this chapter, we learned what C++ AMP is and how to use it for accelerating the application. We also introduced the Compute Shader, a programmable shader stage that can read and write to the GPU resources. Both technologies provide a high-speed and general-purpose computing that takes advantage of a large number of parallel processors on the GPU.

We also compared the performances of both technologies in the vector multiplications and image-processing techniques and found out when we must use either of these technologies in our applications.

# Index

## HLSL Development Cookbook

ISBN: 978-1-84969-420-9          Paperback: 224 pages

Implement stunning 3D rendering techniques using the power of HLSL and DirectX 11

1.  Discover powerful rendering techniques utilizing HLSL and DirectX 11

2.  Augment the visual impact of your projects whilst taking full advantage of the latest GPUs

3.  Experience the practical side of 3D rendering with a comprehensive set of excellent demonstrations
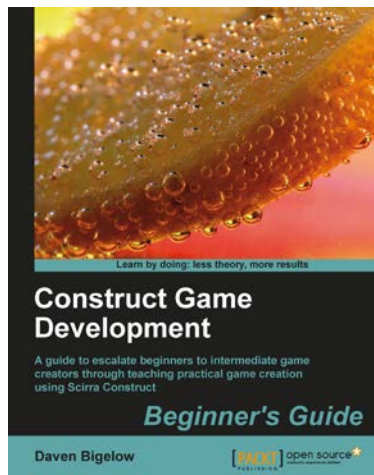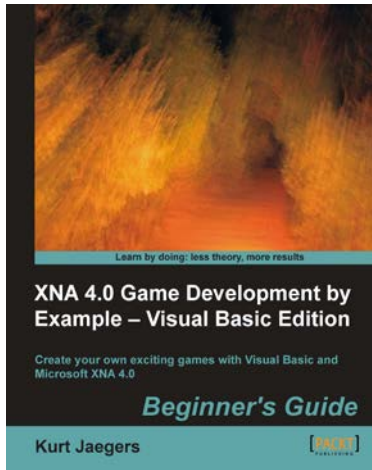
## Construct Game Development Beginner's Guide

ISBN: 978-1-84951-660-0          Paperback: 298 pages

A guide to escalate beginners to intermediate game creators through teaching practical game creation using Scirra Construct

1.  Learn the skills necessary to make your own games through the creation of three very different sample games

2.  Create animated sprites, use built-in physics and shadow engines of Construct Classic

3.  A wealth of step-by-step instructions and images to lead the way

Please check **www.PacktPub.com** for information on our titles

[PACKT]
PUBLISHING

## XNA 4.0 Game Development by Example– Visual Basic Edition: Beginner's Guide

ISBN: 978-1-84969-240-3        Paperback: 424 pages

Create your own exciting games with Visual basic and Microsoft XNA 4.0

1. Visual Basic edition of Kurt Jaegers' XNA 4.0 Game Development by Example. The first book to target Visual Basic developers who want to develop games with the XNA framework

2. Dive headfirst into game creation with Visual Basic and the XNA Framework

3. Four different styles of games comprising a puzzler, space shooter, multi-axis shoot 'em up, and a jump-and-run platformer
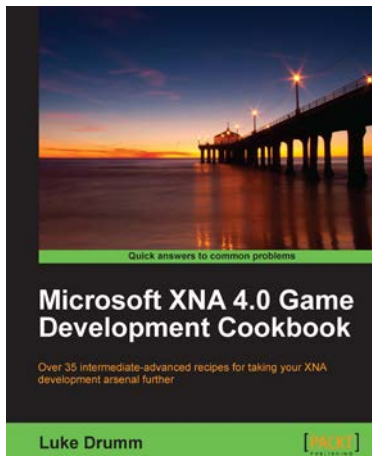
## Microsoft XNA 4.0 Game Development Cookbook

ISBN: 978-1-84969-198-7        Paperback: 356 pages

Over 35 intermediate-advanced recipes for taking your XNA development arsenal further

1. Accelerate your XNA learning with a myriad of tips and tricks to solve your everyday problems

2. Get to grips with adding special effects, virtual atmospheres and computer controlled characters with this book and e-book

3. A fast-paced cookbook packed with screenshots to illustrate each advanced step by step task

Please check **www.PacktPub.com** for information on our titles