



Quick answers to common problems

Direct3D Rendering Cookbook

50 practical recipes to guide you through the advanced rendering techniques in Direct3D to help bring your 3D graphics project to life

Justin Stenning

[PACKT]
PUBLISHING

Direct3D Rendering Cookbook

50 practical recipes to guide you through the advanced rendering techniques in Direct3D to help bring your 3D graphics project to life

Justin Stenning



BIRMINGHAM - MUMBAI

Direct3D Rendering Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2014

Production Reference: 1130114

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-710-1

www.packtpub.com

Cover Image by Justin Stenning (justin.stenning@gmail.com)

Credits

Author

Justin Stenning

Project Coordinator

Wendell Palmer

Reviewers

Julian Amann

Stephan Hodes

Brian Klamik

Todd J. Seiler

Chuck Walbourn

Vinjn Zhang

Proofreaders

Amy Johnson

Lindsey Thomas

Mario Cecere

Indexers

Hemangini Bari

Monica Ajmera Mehta

Rekha Nair

Acquisition Editor

James Jones

Graphics

Ronak Dhruv

Abhinash Sahu

Lead Technical Editor

Priya Singh

Production Coordinator

Nitesh Thakur

Technical Editors

Iram Malik

Shali Sasidharan

Anand Singh

Cover Work

Nitesh Thakur

Copy Editors

Roshni Banerjee

Gladson Monteiro

Adithi Shetty

About the Author

Justin Stenning, a software enthusiast since DOS was king, has been working as a software engineer since he was 20. He has been the technical lead on a range of projects, from enterprise content management and software integrations to mobile apps, mapping, and biosecurity management systems. Justin has been involved in a number of open source projects, including capturing images from fullscreen Direct3D games and displaying in-game overlays, and enjoys giving a portion of his spare time to the open source community. Justin completed his Bachelor of Information Technology at Central Queensland University, Rockhampton. When not coding or gaming, he thinks about coding or gaming, or rides his motorbike. Justin lives with his awesome wife, and his cheeky and quirky children in Central Victoria, Australia.

To Lee, thanks for keeping things running smoothly using your special skill of getting stuff done and of course for your awesomeness. To the kids, yes, I will now be able to play more Minecraft and Terraria with you.

I would like to thank Michael for taking a punt on me all those years ago and mentoring me in the art of coding.

I would also like to thank the SharpDX open source project for producing a great interface to Direct3D from the managed code, and Blendswap.com and its contributors for providing such a great service to the Blender community.

Thank you to the reviewers who provided great feedback and suggestions throughout.

Lastly, a big thank you to James, Priya, Wendell, and all the folks at Packt Publishing who have made this book possible.

About the Reviewers

Julian Amann started working with DirectX 13 years ago, as a teenager. He received his master's degree in Computer Science from the Technische Universität München (Germany) in 2011. He has worked as a research assistant at the Chair of Computer Graphics at Bauhaus-Universität Weimar, where he did his research on image quality algorithms and has also been involved in teaching computer graphics. Currently, Julian works at the **Chair of Computational Modeling and Simulation (CMS)** at the Technische Universität München. He is writing his PhD thesis about product data models for infrastructure projects in the field of Civil Engineering. In his spare time, Julian enjoys programming computer-graphics-related applications and blogging at vertexwahn.de.

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

Since he joined AMD as a Developer Relations Engineer in 2011, he has been working 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.

Brian Klamik has worked as a software design engineer at Microsoft Corporation for 15 years. Nearly all of this time was spent evolving the Direct3D API in Windows by working together with the graphics hardware partners and industry's leading application developers. He enjoys educating developers about using Direct3D optimally, as well as enjoying the results of their labor.

Todd J. Seiler works in the CAD/CAM dental industry as a Graphics Software Engineer at E4D Technologies in Dallas, TX. He has worked as a Software Development Engineer in Test on Games for Windows LIVE at Microsoft, and he has also worked in the mobile game development industry. He has a B.S. in Computer Graphics and Interactive Media from the University of Dubuque in Dubuque, IA with a minor in Computer Information Systems. He also has a B.S. in Real-time Interactive Simulations from DigiPen Institute of Technology in Redmond, WA, with minors in Mathematics and Physics.

In his spare time, he plays video games, studies Catholic apologetics and theology, writes books and articles, and toys with new technology when he can. He periodically blogs about random things at <http://www.toddseiler.com>.

Chuck Walbourn, a software design engineer at Microsoft Corporation, has been working on games for the Windows platform since the early days of DirectX and Windows 95. He entered the gaming industry by starting his own development house during the mid-90s in Austin. He shipped several Windows titles for Interactive Magic and Electronic Arts, and he developed the content tools pipeline for Microsoft Game Studios Xbox titled as Voodoo Vince. Chuck worked for many years in the game developer relations groups at Microsoft, presenting at GDC, Gamefest, X-Fest, and other events. He was the lead developer on the DirectX SDK (June 2010) release. He currently works in the Xbox platform group at Microsoft, where he supports game developers working on the Microsoft platforms through the *Games for Windows and the DirectX SDK* blog, the *DirectX Tool Kit* and *DirectXTex* libraries on CodePlex, and other projects. Chuck holds a bachelor's degree and a master's degree in Computer Science from the University of Texas, Austin.

Vinjn Zhang is an enthusiastic software engineer. His interest in programming includes 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 other for OpenCV.

Vinjn Zhang has worked for several game production companies, including Ubisoft and 2K Games. He currently works as a GPU architect in NVIDIA, where he gets the chance to see the secrets of GPU. Besides his daily work, he is an active GitHub user who turns projects into open source; even his blog is an open source available at <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.

Table of Contents

Preface	1
Chapter 1: Getting Started with Direct3D	7
Introduction	7
Introducing Direct3D 11.1 and 11.2	22
Building a Direct3D 11 application with C# and SharpDX	24
Initializing a Direct3D 11.1/11.2 device and swap chain	32
Debugging your Direct3D application	38
Chapter 2: Rendering with Direct3D	45
Introduction	45
Using the sample rendering framework	46
Creating device-dependent resources	51
Creating size-dependent resources	53
Creating a Direct3D renderer class	59
Rendering primitives	61
Applying multisample anti-aliasing	82
Implementing texture sampling	83
Chapter 3: Rendering Meshes	91
Introduction	91
Rendering a cube and sphere	92
Preparing the vertex and constant buffers for materials and lighting	99
Adding material and lighting	109
Using a right-handed coordinate system	119
Loading a static mesh from a file	121

Chapter 4: Animating Meshes with Vertex Skinning	131
Introduction	131
Preparing the vertex shader and buffers for vertex skinning	131
Loading bones in the mesh renderer	139
Animating bones	147
Chapter 5: Applying Hardware Tessellation	155
Introduction	155
Preparing the vertex shader and buffers for tessellation	156
Tessellating a triangle and quad	158
Tessellating bicubic Bezier surfaces	171
Refining meshes with Phong tessellation	179
Optimizing tessellation through back-face culling and dynamic Level-of-Detail	185
Chapter 6: Adding Surface Detail with Normal and Displacement Mapping	191
Introduction	191
Referencing multiple textures in a material	192
Adding surface detail with normal mapping	194
Adding surface detail with displacement mapping	204
Implementing displacement decals	212
Optimizing tessellation based on displacement decal (displacement adaptive tessellation)	220
Chapter 7: Performing Image Processing Techniques	223
Introduction	223
Running a compute shader – desaturation (grayscale)	224
Adjusting the contrast and brightness	231
Implementing box blur using separable convolution filters	234
Implementing a Gaussian blur filter	243
Detecting edges with the Sobel edge-detection filter	246
Calculating an image's luminance histogram	250
Chapter 8: Incorporating Physics and Simulations	257
Introduction	257
Using a physics engine	257
Simulating ocean waves	266
Rendering particles	274

Chapter 9: Rendering on Multiple Threads and Deferred Contexts	295
Introduction	295
Benchmarking multithreaded rendering	296
Implementing multithreaded dynamic cubic environment mapping	305
Implementing dual paraboloid environment mapping	322
Chapter 10: Implementing Deferred Rendering	333
Introduction	333
Filling the G-Buffer	334
Implementing a screen-aligned quad renderer	346
Reading the G-Buffer	352
Adding multiple lights	357
Incorporating multisample anti-aliasing	373
Chapter 11: Integrating Direct3D with XAML and Windows 8.1	379
Introduction	379
Preparing the swap chain for a Windows Store app	380
Rendering to a CoreWindow	384
Rendering to an XAML SwapChainPanel	390
Loading and compiling resources asynchronously	397
Appendix: Further Reading	403
Index	407

Preface

The latest 3D graphics cards bring us amazing visuals in the latest games, from Indie to AAA titles. This is made possible on Microsoft platforms including PC, Xbox consoles, and mobile devices thanks to Direct3D—a component of the DirectX API dedicated to exposing 3D graphics hardware to programmers. Microsoft DirectX is the graphics technology powering today's hottest games on Microsoft platforms. DirectX 11 features hardware tessellation for rich geometric detail, compute shaders for custom graphics effects, and improved multithreading for better hardware utilization. With it comes a number of fundamental game changing improvements to the way in which we render 3D graphics.

The last decade has also seen the rise of **General-Purpose computation on Graphics Processing Units (GPGPU)**, exposing the massively parallel computing power of **Graphics Processing Units (GPUs)** to programmers for scientific or technical computing. Some uses include implementing **Artificial Intelligence (AI)**, advanced postprocessing and physics within games, powering complex scientific modeling, or contributing to large scale distributed computing projects.

Direct3D and related DirectX graphics APIs continue to be an important part of the Microsoft technology stack. Remaining an integral part of their graphics strategy on all platforms, the library advances in leaps and bounds with each new release, opening further opportunities for developers to exploit. With the release of the third generation Xbox console—the Xbox One—and the latest games embracing the recent DirectX 11 changes in 11.1 and 11.2, we will continue to see Direct3D be a leading 3D graphics API.

Direct3D Rendering Cookbook is a practical, example-driven, technical cookbook with numerous Direct3D 11.1 and 11.2 rendering techniques supported by illustrations, example images, strong sample code, and concise explanations.

What this book covers

Chapter 1, Getting Started with Direct3D, reviews the components of Direct3D and the graphics pipeline, explores the latest features in DirectX 11.1 and 11.2, and looks at how to build and debug Direct3D applications with C# and SharpDX.

Chapter 2, Rendering with Direct3D, introduces a simple rendering framework, teaches how to render primitive shapes, and compiles HLSL shaders and use textures.

Chapter 3, Rendering Meshes, explores rendering more complex objects and demonstrates how to use the Visual Studio graphics content pipeline to compile and render 3D assets.

Chapter 4, Animating Meshes with Vertex Skinning, teaches how to implement vertex skinning for the animation of 3D models.

Chapter 5, Applying Hardware Tessellation, covers tessellating primitive shapes, parametric surfaces, mesh subdivision/refinement, and techniques for optimizing tessellation performance.

Chapter 6, Adding Surface Detail with Normal and Displacement Mapping, teaches how to combine tessellation with normal and displacement mapping to increase surface detail. Displacement decals are explored and then optimized for performance with displacement adaptive tessellation.

Chapter 7, Performing Image Processing Techniques, describes how to use compute shaders to implement a number of image-processing techniques often used within postprocessing.

Chapter 8, Incorporating Physics and Simulations, explores implementing physics, simulating ocean waves, and rendering particles.

Chapter 9, Rendering on Multiple Threads and Deferred Contexts, benchmarks multithreaded rendering and explores the impact of multithreading on two common environment-mapping techniques.

Chapter 10, Implementing Deferred Rendering, provides insight into the techniques necessary to implement deferred rendering solutions.

Chapter 11, Integrating Direct3D with XAML and Windows 8.1, covers how to implement Direct3D Windows Store apps and optionally integrate with XAML based UIs and effects. Loading and compiling resources within Windows 8.1 is also explored.

Appendix, Further Reading, includes all the references and papers that can be referred for gathering more details and information related to the topics covered in the book.

What you need for this book

To complete the recipes in this book, it is necessary that you have a graphics card that supports a minimum of DirectX 11.1.

It is recommended that you have the following software:

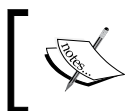
- ▶ Windows 8.1
- ▶ Microsoft Visual Studio 2013 Express (or higher edition)
- ▶ Microsoft .NET Framework 4.5
- ▶ Windows Software Development Kit (SDK) for Windows 8.1
- ▶ SharpDX 2.5.1 or higher—<http://sharpdx.org/news/>

Other resources and libraries are indicated in individual recipes.

For those running Windows 7 or Windows 8, you will require a minimum of the following software. Please note that although some portions of Chapter 11, *Integrating Direct3D with XAML and Windows 8.1*, can be adapted to Windows 8, you will not be able to complete the final chapter in its entirety as it is specific to Windows 8.1.

- ▶ Microsoft Visual Studio 2012 or 2013 Express (or higher edition)
- ▶ Microsoft .NET Framework 4.5
- ▶ Windows 8 or Windows 7 with Platform Update for SP1*
- ▶ Windows Software development Kit (SDK) for Windows 8
- ▶ SharpDX 2.5.1 or higher—<http://sharpdx.org/news/>

Other resources and libraries as indicated in individual recipes.



Chapter 11, *Integrating Direct3D with XAML and Windows 8.1*, is not compatible with Windows 7, and the *Rendering to a XAML SwapChainPanel* recipe requires a minimum of Windows 8.1.

Who this book is for

Direct3D Rendering Cookbook is for C# .NET developers who want to learn the advanced rendering techniques made possible with DirectX 11.1 and 11.2. It is expected that the reader has at least a cursory knowledge of graphics programming, and although some knowledge of Direct3D 10+ is helpful, it is not necessary. An understanding of vector and matrix algebra is recommended.

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: A command list is represented by the `ID3D11CommandList` interface in unmanaged C++ and the `Direct3D11.CommandList` class in managed C# with SharpDX.


A block of code is set as follows:


```
SharpDX.Direct3D.FeatureLevel.Level_11_1,  
SharpDX.Direct3D.FeatureLevel.Level_11_0,  
SharpDX.Direct3D.FeatureLevel.Level_10_1,  
SharpDX.Direct3D.FeatureLevel.Level_10_0,
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
// Create the device and swapchain  
Device.CreateWithSwapChain(  
    SharpDX.Direct3D.DriverType.Hardware,  
    DeviceCreationFlags.None,
```

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: "These are accessible by navigating to the **DEBUG/Graphics** menu".

[ 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.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/71010T_ColoredImages.pdf

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

Getting Started with Direct3D

In this chapter, we will cover the following topics:

- ▶ Components of Direct3D
- ▶ Stages of the programmable pipeline
- ▶ Introducing Direct3D 11.1 and 11.2
- ▶ Building a Direct3D 11 application with C# and SharpDX
- ▶ Initializing a Direct3D 11.1/11.2 device and swap chain
- ▶ Debugging your Direct3D application

Introduction

Direct3D is the component of the DirectX API dedicated to exposing 3D graphics hardware to programmers on Microsoft platforms including PC, console, and mobile devices. It is a native API allowing you to create not only 3D graphics for games, scientific and general applications, but also to utilize the underlying hardware for **General-purpose computing on graphics processing units (GPGPU)**.

Programming with Direct3D can be a daunting task, and although the differences between the unmanaged C++ API and the managed .NET SharpDX API (from now on referred to as the unmanaged and managed APIs respectively) are subtle, we will briefly highlight some of these while also gaining an understanding of the graphics pipeline.

We will then learn how to get started with programming for Direct3D using C# and SharpDX along with some useful debugging techniques.

Components of Direct3D

Direct3D is a part of the larger DirectX API comprised of many components that sits between applications and the graphics hardware drivers. Everything in Direct3D begins with the device and you create resources and interact with the graphics pipeline through various **Component Object Model (COM)** interfaces from there.

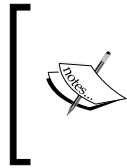
Device

The main role of the device is to enumerate the capabilities of the display adapter(s) and to create resources. Applications will typically only have a single device instantiated and must have at least one device to use the features of Direct3D.

Unlike previous versions of Direct3D, in Direct3D 11 the device is thread-safe. This means that resources can be created from any thread.

The device is accessed through the following interfaces/classes:

- ▶ Managed: `Direct3D11.Device` (Direct3D 11), `Direct3D11.Device1` (Direct3D 11.1), and `Direct3D11.Device2` (Direct3D 11.2)
- ▶ Unmanaged: `ID3D11Device`, `ID3D11Device1`, and `ID3D11Device2`



Each subsequent version of the COM interface descends from the previous version; therefore, if you start with a Direct3D 11 device instance and query the interface for the Direct3D 11.2 implementation, you will still have access to the Direct3D 11 methods with the resulting device reference.

One important difference between the unmanaged and managed version of the APIs used throughout this book is that when creating resources on a device with the managed API, the appropriate class constructor is used with the first parameter passed in being a device instance, whereas the unmanaged API uses a `Create` method on the device interface.

For example, creating a new blend state would look like the following for the managed C# API:

```
var blendState = new BlendState(device, desc);
```

And like this for the unmanaged C++ API:

```
ID3D11BlendState* blendState;  
HRESULT r = device->CreateBlendState(&desc, &blendState);
```

**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.

Further, a number of the managed classes use overloaded constructors and methods that only support valid parameter combinations, relying less on a programmer's deep understanding of the Direct3D API.

With Direct3D 11, Microsoft introduced Direct3D feature levels to manage the differences between video cards. The feature levels define a matrix of Direct3D features that are mandatory or optional for hardware devices to implement in order to meet the requirements for a specific feature level. The minimum feature level required for an application can be specified when creating a device instance, and the maximum feature level supported by the hardware device is available on the `Device.FeatureLevel` property. More information on feature levels and the features available at each level can be found at [http://msdn.microsoft.com/en-us/library/windows/desktop/ff476876\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff476876(v=vs.85).aspx).

Device context

The device context encapsulates all rendering functions. These include setting the pipeline state and generating rendering commands with resources created on the device.

Two types of device context exist in Direct3D 11, the immediate context and deferred context. These implement immediate rendering and deferred rendering respectively.

The interfaces/classes for both context types are:

- ▶ **Managed:** `Direct3D11.DeviceContext`, `Direct3D11.DeviceContext1`, and `Direct3D11.DeviceContext2`
- ▶ **Unmanaged:** `ID3D11DeviceContext`, `ID3D11DeviceContext1`, and `ID3D11DeviceContext2`

Immediate context

The immediate context provides access to data on the GPU and the ability to execute/playback command lists immediately against the device. Each device has a single immediate context and only one thread may access the context at the same time; however, multiple threads can interact with the immediate context provided appropriate thread synchronization is in place.

All commands to the underlying device eventually must pass through the immediate context if they are to be executed.

The immediate context is available on the device through the following methods/properties:

- ▶ **Managed:** `Device.ImmediateContext`, `Device1.ImmediateContext1`, and `Device2.ImmediateContext2`
- ▶ **Unmanaged:** `ID3D11Device::GetImmediateContext`, `ID3D11Device1::GetImmediateContext1`, and `ID3D11Device2::GetImmediateContext2`

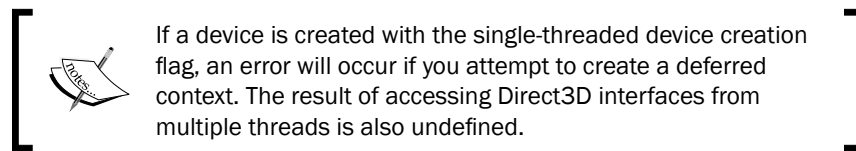
Deferred context

The same rendering methods are available on a deferred context as for an immediate context; however, the commands are added to a queue called a command list for later execution upon the immediate context.

Using deferred contexts results in some additional overhead, and only begins to see benefits when parallelizing CPU-intensive tasks. For example, rendering the same simple scene for the six sides of a cubic environment map will not immediately see any performance benefits, and in fact will increase the time it takes to render a frame as compared to using the immediate context directly. However, render the same scene again with enough CPU load and it is possible to see some improvements over rendering directly on the immediate context. The usage of deferred contexts is no substitute for a well written engine and needs to be carefully evaluated to be correctly taken advantage of.

Multiple deferred context instances can be created and accessed from multiple threads; however, each may only be accessed by one thread at a time. For example, with the deferred contexts A and B, we can access both at the exact same time from threads 1 and 2 provided that thread 1 is only accessing deferred context A and thread 2 is only accessing deferred context B (or vice versa). Any sharing of contexts between threads requires thread synchronization.

The resulting command lists are not executed against the device until they are played back by an immediate context.



A deferred context is created with:

- ▶ **Managed:** `new DeviceContext(device)`
- ▶ **Unmanaged:** `ID3D11Device::CreateDeferredContext`

Command lists


A command list stores a queue of Direct3D API commands for deferred execution or merging into another deferred context. They facilitate the efficient playback of a number of API commands queued from a device context.

A command list is represented by the `ID3D11CommandList` interface in unmanaged C++ and the `Direct3D11.CommandList` class in managed C# with SharpDX. They are created using:

- ▶ Managed: `DeviceContext.FinishCommandList`
- ▶ Unmanaged: `ID3D11DeviceContext::FinishCommandList`

Command lists are played back on the immediate context using:

- ▶ Managed: `DeviceContext.ExecuteCommandList`
- ▶ Unmanaged: `ID3D11DeviceContext::ExecuteCommandList`

 Trying to execute a command list on a deferred context or trying to create a command list from an immediate context will result in an error.

Swap chains

A swap chain facilitates the creation of one or more back buffers. These buffers are used to store rendered data before being presented to an output display device. The swap chain takes care of the low-level presentation of this data and with Direct3D 11.1, supports stereoscopic 3D display behavior (left and right eye for 3D glasses/displays).

If the output of rendering is to be sent to an output connected to the current adapter, a swap chain is required.

Swap chains are part of the **DirectX Graphics Infrastructure (DXGI)** API, which is responsible for enumerating graphics adapters, display modes, defining buffer formats, sharing resources between processes, and finally (via the swap chain) presenting rendered frames to a window or output device for display.

A swap chain is represented by the following types:

- ▶ Managed: `SharpDX.DXGI.SwapChain` and `SharpDX.DXGI.SwapChain1`
- ▶ Unmanaged: `IDXGISwapChain` and `IDXGISwapChain1`

States

A number of state types exist to control the behavior of some fixed function stages of the pipeline and how samplers behave for shaders.

All shaders can accept several sampler states. The output merger can accept both, a blend state and depth-stencil state, and the rasterizer accepts a rasterizer state. The types used are shown in the following table.

Managed type (SharpDX.Direct3D11)	Unmanaged type
BlendState	ID3D11BlendState
BlendState1	ID3D11BlendState1
DepthStencilState	ID3D11DepthStencilState
RasterizerState	ID3D11RasterizerState
RasterizerState1	ID3D11RasterizerState1
SamplerState	ID3D11SamplerState

Resources

A resource is any buffer or texture that is used as an input and/or output from the Direct3D pipeline. A resource is consumed by creating one or more views to the resource and then binding them to stages of the pipeline.

Textures

A texture resource is a collection of elements known as texture pixels or **texels**—which represent the smallest unit of a texture that can be read or written to by the pipeline. A texel is generally comprised of between one and four components depending on which format is being used for the texture; for example, a format of `Format.R32G32B32_Float` is used to store three 32-bit floating point numbers in each texel whereas a format of `Format.R8G8_UInt` represents two 8-bit unsigned integers per texel. There is a special case when dealing with compressed formats (`Format.BC`) where the smallest unit consists of a block of 4 x 4 texels.

A texture resource can be created in a number of different formats as defined by the DXGI format enumeration (`SharpDX.DXGI.Format` and `DXGI_FORMAT` for managed/unmanaged, respectively). The format can be either applied at the time of creation, or specified when it is bound by a resource view to the pipeline.

Hardware device drivers may support different combinations of formats for different purposes, although there is a list of mandatory formats that the hardware must support depending on the version of Direct3D. The device's `CheckFormatSupport` method can be used to determine what resource type and usage a particular format supports on the current hardware.



Textures do not just store image data. They are used for information, such as height-maps, displacement-maps, or for any data structure that needs to be read or written within a shader that can benefit from the speed benefits of hardware support for textures and texture sampling.

Types of texture resources include:

- ▶ 1D Textures and 1D Texture Arrays
- ▶ 2D Textures and 2D Texture Arrays
- ▶ 3D Textures (or volume textures)
- ▶ Unordered access textures
- ▶ Read/Write textures

The following table maps the managed to unmanaged types for the different textures.

Managed type (SharpDX.Direct3D11)	Unmanaged type
Texture1D	ID3D11Texture1D
Texture2D	ID3D11Texture2D
Texture3D	ID3D11Texture3D

Arrays of 1D and 2D textures are configured with the subresource data associated with the description of the texture passed into the appropriate constructor. A common use for texture arrays is supporting **Multiple Render Targets (MRT)**.

Resource views

Before a resource can be used within a stage of the pipeline it must first have a view. This view describes to the pipeline stages what format to expect the resource in and what region of the resource to access. The same resource can be bound to multiple stages of the pipeline using the same view, or by creating multiple resource views.

It is important to note that although a resource can be bound to multiple stages of the pipeline, there may be restrictions on whether the same resource can be bound for input and output at the same time. As an example, a **Render Target View (RTV)** and **Shader Resource View (SRV)** for the same resource both cannot be bound to the pipeline at the same time. When a conflict arises the read-only resource view will be automatically unbound from the pipeline, and if the debug layer is enabled, a warning message will be output to the debug output.

Using resources created with a typeless format, allows the same underlying resource to be represented by multiple resource views, where the compatible resolved format is defined by the view. For example, using a resource with both a **Depth Stencil View (DSV)** and SRV requires that the underlying resource be created with a format like `Format.R32G8X24_Typeless`. The SRV then specifies a format of `Format.R32_Float_X8X24_Typeless`, and finally the DSV is created with a format of `Format.D32_Float_S8X24_UInt`.

Some types of buffers can be provided to certain stages of the pipeline without a resource view, generally when the structure and format of the buffer is defined in some other way, for example, using state objects or structures within shader files.

Types of resource views include:

- ▶ Depth Stencil View (DSV),
- ▶ Render Target View (RTV),
- ▶ Shader Resource View (SRV)
- ▶ Unordered Access View (UAV)
- ▶ Video decoder output view
- ▶ Video processor input view
- ▶ Video processor output view

The following table shows the managed and unmanaged types for the different resource views.

Managed type (SharpDX.Direct3D11)	Unmanaged type
DepthStencilView	ID3D11DepthStencilView
RenderTargetView	ID3D11RenderTargetView
ShaderResourceView	ID3D11ShaderResourceView
UnorderedAccessView	ID3D11UnorderedAccessView
VideoDecoderOutputView	ID3D11VideoDecoderOutputView
VideoProcessorInputView	ID3D11VideoProcessorInputView
VideoProcessorOutputView	ID3D11VideoProcessorOutputView

Buffers

A buffer resource is used to provide structured and unstructured data to stages of in the graphics pipeline.

Types of buffer resources include:

- ▶ Vertex buffer
- ▶ Index buffer
- ▶ Constant buffer
- ▶ Unordered access buffers
 - ❑ Byte address buffer
 - ❑ Structured buffer
 - ❑ Read/Write buffers
 - ❑ Append/Consume structured buffers

All buffers are represented by the `SharpDX.Direct3D11.Buffer` class (`ID3D11Buffer` for the unmanaged API). The usage is defined by how and where it is bound to the pipeline. The following table shows the binding flags for different buffers:

Buffer type	Managed BindFlags flags	Unmanaged D3D11_BIND_FLAG flags
Vertex buffer	<code>VertexBuffer</code>	<code>D3D11_BIND_VERTEX_BUFFER</code>
Index buffer	<code>IndexBuffer</code>	<code>D3D11_BIND_INDEX_BUFFER</code>
Constant buffer	<code>ConstantBuffer</code>	<code>D3D11_BIND_CONSTANT_BUFFER</code>
Unordered access buffers	<code>UnorderedAccess</code>	<code>D3D11_BIND_UNORDERED_ACCESS</code>

Unordered access buffers are further categorized into the following types using an additional option/miscellaneous flag within the buffer description as shown in the following table:

Buffer type	Managed ResourceOptionFlags flags	Unmanaged D3D11_RESOURCE_MISC_FLAG flags
Byte address buffer	<code>BufferAllowRawViews</code>	<code>D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS</code>
Structured buffer	<code>BufferStructured</code>	<code>D3D11_RESOURCE_MISC_BUFFER_STRUCTURED</code>
Read/Write buffers	Either use Byte address buffer / Structured buffer and then use <code>RWBuffer</code> or <code>RWStructuredBuffer<MyStruct></code> instead of <code>Buffer</code> and <code>StructuredBuffer<MyStruct></code> in HLSL.	
Append/Consume buffers	A structured buffer and then use <code>AppendStructuredBuffer</code> or <code>ConsumeStructuredBuffer</code> in HLSL. Use <code>UnorderedAccessViewBufferFlags.Append</code> when creating the UAV.	

Shaders and High Level Shader Language

The graphics pipeline is made up of fixed function and programmable stages. The programmable stages are referred to as shaders, and are programmed using small **High Level Shader Language (HLSL)** programs. The HLSL is implemented with a series of shader models, each building upon the previous version. Each shader model version supports a set of shader profiles, which represent the target pipeline stage to compile a shader. Direct3D 11 introduces **Shader Model 5 (SM5)**, a superset of **Shader Model 4 (SM4)**.

An example shader profile is `ps_5_0`, which indicates a shader program is for use in the pixel shader stage and requires SM5.

Stages of the programmable pipeline

All Direct3D operations take place via one of the two pipelines, known as pipelines for the fact that information flows in one direction from one stage to the next. For all drawing operations, the graphics pipeline is used (also known as drawing pipeline or rendering pipeline). To run compute shaders, the dispatch pipeline is used (aka DirectCompute pipeline or compute shader pipeline).

Although these two pipelines are conceptually separate. They cannot be active at the same time. Context switching between the two pipelines also incurs additional overhead so each pipeline should be used in blocks—for example, run any compute shaders to prepare data, perform all rendering, and finally post processing.

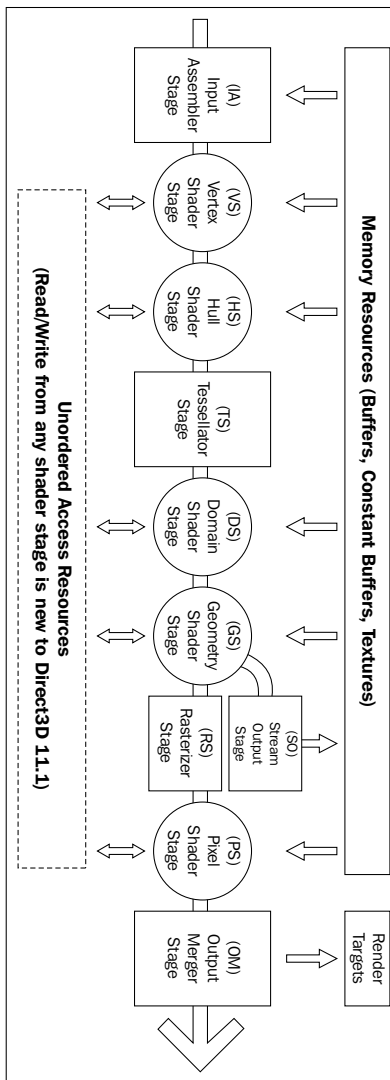
Methods related to stages of the pipeline are found on the device context. For the managed API, each stage is grouped into a property named after the pipeline stage. For example, for the vertex shader stage, `deviceContext.VertexShader.SetShaderResources`, whereas the unmanaged API groups the methods by a stage acronym directly on the device context, for example, `deviceContext->VSSetShaderResources`, where VS represents the vertex shader stage.

The graphics pipeline

The graphics pipeline is comprised of nine distinct stages that are generally used to create 2D raster representations of 3D scenes, that is, take our 3D model and turn it into what we see on the display. Four of these stages are fixed function and the remaining five programmable stages are called shaders (the following diagram shows the programmable stages as a circle). The output of each stage is taken as input into the next along with bound resources or in the case of the last stage, **Output Merger (OM)**, the output is sent to one or more render targets. Not all of the stages are mandatory and keeping the number of stages involved to a minimum will generally result in faster rendering.

Optional tessellation support is provided by the three tessellation stages (two programmable and one fixed function): the hull shader, tessellator, and domain shader. The tessellation stages require a Direct3D feature level of 11.0 or later.

As of Direct3D 11.1, each programmable stage is able to read/write to an **Unordered Access View (UAV)**. A UAV is a view of a buffer or texture resource that has been created with the `BindFlags.UnorderedAccess` flag (`D3D11_BIND_UNORDERED_ACCESS` from the `D3D11_BIND_FLAG` enumeration).



Direct3D Graphics Pipeline

Input Assembler (IA) stage

The IA stage reads primitive data (points, lines, and/or triangles) from buffers and assembles them into primitives for use in subsequent stages.

Usually one or more vertex buffers, and optionally an index buffer, are provided as input. An input layout tells the input assembler what structure to expect the vertex buffer in.

The vertex buffer itself is also optional, where a vertex shader only has a vertex ID as input (using the `SV_VertexID` shader system value input semantic) and then can either generate the vertex data procedurally or retrieve it from a resource using the vertex ID as an index. In this instance, the input assembler is not provided with an input layout or vertex buffer, and simply receives the number of vertices that will be drawn. For more information, see [http://msdn.microsoft.com/en-us/library/windows/desktop/bb232912\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb232912(v=vs.85).aspx).

Device context commands that act upon the input assembler directly are found on the `DeviceContext.InputAssembler` property, for example, `DeviceContext.InputAssembler.SetVertexBuffers`, or for unmanaged begin with IA, for example, `ID3D11DeviceContext::IASetVertexBuffers`.

Vertex Shader (VS) stage

The vertex shader allows per-vertex operations to be performed upon the vertices provided by the input assembler. Operations include manipulating per-vertex properties such as position, color, texture coordinate, and a vertex's normal.

A vertex can be comprised of up to sixteen 32-bit vectors (up to four components each). A minimal vertex usually consists of position, color, and the normal vector. In order to support larger sets of data or as an alternative to using a vertex buffer, the vertex shader can also retrieve data from a texture or UAV.

A vertex shader is required; even if no transform is needed, a shader must be provided that simply returns vertices without modifications.

Device context commands that are used to control the vertex shader stage are grouped within the `DeviceContext.VertexShader` property or for unmanaged begin with VS, for example, `DeviceContext.VertexShader.SetShaderResources` and `ID3D11DeviceContext::VSSetShaderResources`, respectively.

Hull Shader (HS) stage

The hull shader is the first stage of the three optional stages that together support hardware accelerated tessellation. The hull shader outputs control points and patches constant data that controls the fixed function tessellator stage. The shader also performs culling by excluding patches that do not require tessellation (by applying a tessellation factor of zero).

Unlike other shaders, the hull shader consists of two HLSL functions: the patch constant function, and hull shader function.

This shader stage requires that the IA stage has one of the patch list topologies set as its active primitive topology (for example, `SharpDX.Direct3D.PrimitiveTopology.PatchListWith3ControlPoints` for managed and `D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST` for unmanaged).

Device context commands that control the hull shader stage are grouped within the `DeviceContext.HullShader` property or for unmanaged device begin with HS.

Tessellator stage

The tessellator stage is the second stage of the optional tessellation stages. This fixed function stage subdivides a quad, triangle, or line into smaller objects. The tessellation factor and type of division is controlled by the output of the hull shader stage.

Unlike all other fixed function stages the tessellator stage does not include any direct method of controlling its state. All required information is provided within the output of the hull shader stage and implied through the choice of primitive topology and configuration of the hull and domain shaders.

Domain Shader (DS) stage

The domain shader is the third and final stage of the optional tessellation stages. This programmable stage calculates the final vertex position of a subdivided point generated during tessellation.

The types of operations that take place within this shader stage are often fairly similar to a vertex shader when not using the tessellation stages.

Device context commands that control the domain shader stage are grouped by the `DeviceContext.DomainShader` property, or for unmanaged begin with DS.

Geometry Shader (GS) stage

The optional geometry shader stage runs shader code that takes an entire primitive or primitive with adjacency as input. The shader is able to generate new vertices on output (triangle strip, line strip, or point list).



The geometry shader stage is unique in that its output can go to the rasterizer stage and/or be sent to a vertex buffer via the stream output stage (SO).

It is critical for performance that the amount of data sent into and out of the geometry shader is kept to a minimum. The geometry shader stage has the potential to slow down the rendering performance quite significantly.

Uses of the geometry shader might include rendering multiple faces of environment maps in a single pass (refer to *Chapter 9, Rendering on Multiple Threads and Deferred Contexts*), and point sprites/billboarding (commonly used in particle systems). Prior to Direct3D 11, the geometry shader could be used to implement tessellation.

Device context commands that control the geometry shader stages are grouped in the `GeometryShader` property, or for unmanaged begin with `GS`.

Stream Output (SO) stage

The stream output stage is an optional fixed function stage that is used to output geometry from the geometry shader into vertex buffers for re-use or further processing in another pass through the pipeline.

There are only two commands on the device context that control the stream output stage found on the `StreamOutput` property of the device context: `GetTargets` and `SetTargets` (unmanaged `SOGetTargets` and `SOSetTargets`).

Rasterizer stage (RS)

The rasterizer stage is a fixed function stage that converts the vector graphics (points, lines, and triangles) into raster graphics (pixels). This stage performs view frustum clipping, back-face culling, early depth/stencil tests, perspective divide (to convert our vertices from clip-space coordinates to normalized device coordinates), and maps vertices to the viewport. If a pixel shader is specified, this will be called by the rasterizer for each pixel, with the result of interpolating per-vertex values across each primitive passed as the pixel shader input.

There are additional interpolation modifiers that can be applied to the pixel shader input structure that tell the rasterizer stage the method of interpolation that should be used for each property (for more information see Interpolation Modifiers introduced in Shader Model 4 on MSDN at [http://msdn.microsoft.com/en-us/library/windows/desktop/bb509668\(v=vs.85\).aspx#Remarks](http://msdn.microsoft.com/en-us/library/windows/desktop/bb509668(v=vs.85).aspx#Remarks)).

When using multisampling, the rasterizer stage can provide an additional coverage mask to the pixel shader that indicates which samples are covered by the pixel. This is provided within the `SV_Coverage` system-value input semantic. If the pixel shader specifies the `SV_SampleIndex` input semantic, instead of being called once per pixel by the rasterizer, it will be called once per sample per pixel (that is, a 4xMSAA render target would result in four calls to the pixel shader for each pixel).

Device context commands that control the rasterizer stage state are grouped in the `Rasterizer` property of the device context or for unmanaged begin with `RS`.

Pixel Shader (PS) stage

The final programmable stage is the pixel shader. This stage executes a shader program that performs per-pixel operations to determine the final color of each pixel. Operations that take place here include per-pixel lighting and post processing.

Device context commands that control the pixel shader stage are grouped by the `PixelShader` property or begin with PS for the unmanaged API.

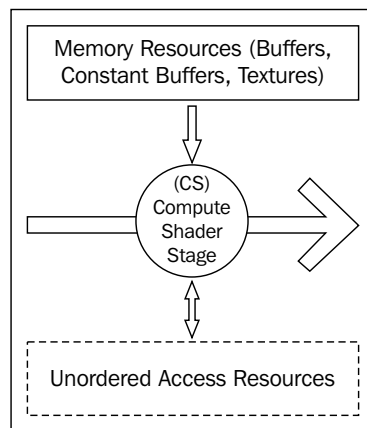
Output Merger (OM) stage

The final stage of the graphics pipeline is the output merger stage. This fixed function stage generates the final rendered pixel color. You can bind a depth-stencil state to control z-buffering, and bind a blend state to control blending of pixel shader output with the render target.

Device context commands that control the state of the output merger stage are grouped by the `OutputMerger` property or for unmanaged begin with OM.

The dispatch pipeline

The dispatch pipeline is where compute shaders are executed. There is only one stage in this pipeline, the compute shader stage. The dispatch pipeline and graphics pipeline cannot run at the same time and there is an additional context change cost when switching between the two, therefore calls to the dispatch pipeline should be grouped together where possible.



Direct3D Dispatch/DirectCompute Pipeline

Compute Shader (CS) stage

The compute shader (also known as DirectCompute) is an optional programmable stage that executes a shader program upon multiple threads, optionally passing in a dispatch thread identifier (`SV_DispatchThreadID`) and up to three thread group identifier values as input (`SV_GroupIndex`, `SV_GroupID`, and `SV_GroupThreadID`). This shader supports a whole range of uses including post processing, physics simulation, AI, and GPGPU tasks.

Compute shader support is mandatory for hardware devices from feature level 11_0 onwards, and optionally available on hardware for feature levels 10_0 and 10_1.

The thread identifier is generally used as an index into a resource to perform an operation. The same shader program is run upon many thousands of threads at the same time, usually with each reading and/or writing to an element of a UAV resource.

Device context commands that control the compute shader stage are grouped in the `ComputeShader` property or begin with CS in the unmanaged API.

After the compute shader is prepared, it is executed by calling the `Dispatch` command on the device context, passing in the number of thread groups to use.

Introducing Direct3D 11.1 and 11.2

With the release of Windows 8 came a minor release of Direct3D, Version 11.1 and the DXGI API, Version 1.2. A number of features that do not require **Windows Display Driver Model (WDDM) 1.2** were later made available for Windows 7 and Windows Server 2008 R2 with the Platform Update for Windows 7 SP1 and Windows Server 2008 R2 SP1.

Now with the release of Windows 8.1 in October 2013 and the arrival of the Xbox One not long after, Microsoft has provided another minor release of Direct3D, Version 11.2 and DXGI Version 1.3. These further updates are not available on previous versions of Windows 7 or Windows 8.

Direct3D 11.1 and DXGI 1.2 features

Direct3D 11.1 introduces a number of enhancements and additional features, including:

- ▶ **Unordered Access Views (UAVs)** can now be used in any shader stage, not just the pixel and compute shaders
- ▶ A larger number of UAVs can be used when you bind resources to the output merger stage
- ▶ Support for reducing memory bandwidth and power consumption (HLSL minimum precision and swap chain dirty regions and scroll present parameters)

- ▶ Shader tracing and compiler enhancements
- ▶ Direct3D device sharing
- ▶ Create larger constant buffers than a shader can access (by binding a subset of a constant buffer)
- ▶ Support logical operations in a render target with new blend state options
- ▶ Create SRV/RTV and UAVs to video resources so that Direct3D shaders can process video resources
- ▶ Ability to use Direct3D in Session 0 processes (from background services)
- ▶ Extended resource sharing for shared Texture2D resources

DXGI 1.2 enhancements include:

- ▶ A new flip-model swap chain
- ▶ Support for stereoscopic 3D displays
- ▶ Restricting output to a specific display
- ▶ Support for dirty rectangles and scrolled areas that can reduce memory bandwidth and power consumption
- ▶ Events for notification of application occlusion status (that is, knowing when rendering is not necessary)
- ▶ A new desktop duplication API that replaces the previous mirror drivers
- ▶ Improved event-based synchronization to share resources
- ▶ Additional debugging APIs

Direct3D 11.2 and DXGI 1.3 features

Direct3D 11.2 is a smaller incremental update by comparison and includes the following enhancements:

- ▶ HLSL compilation within Windows Store apps under Windows 8.1. This feature was missing from Windows 8 Windows Store apps and now allows applications to compile shaders at runtime for Windows Store apps.
- ▶ HLSL shader linking, adding support for precompiled HLSL functions that can be packaged into libraries and linked into shaders at runtime.
- ▶ Support for tiled resources, large resources that use small amounts of physical memory—suitable for large terrains.
- ▶ Ability to annotate graphics commands, sending strings and an integer value to **Event Tracing for Windows (ETW)**.

DXGI 1.3 enhancements include:

- ▶ Overlapping swap chains and scaling, for example, presenting a swap chain that is rendered at a lower resolution, then up-scaling and overlapping with a UI swap chain at the displays native resolution.
- ▶ Trim device command, allowing memory to be released temporarily. Suitable for when an application is being suspended and to reduce the chances that it will be terminated to reclaim resources for other apps.
- ▶ Ability to set the source size of the back buffer allowing the swap chain to be resized (smaller) without recreating the swap chain resources.
- ▶ Ability to implement more flexible and lower frame latencies by specifying the maximum frame latency (number of frames that can be queued at one time) and retrieving a wait handle to use with `WaitForSingleObjectEx` before commencing the next frame's drawing commands.

Building a Direct3D 11 application with C# and SharpDX

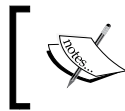
In this recipe we will prepare a blank project that contains the appropriate SharpDX references and a minimal rendering loop. The project will initialize necessary Direct3D objects and then provide a basic rendering loop that sets the color of the rendering surface to `Color.LightBlue`.

Getting ready

Make sure you have Visual Studio 2012 Express for Windows Desktop or Professional and higher installed. Download the SharpDX binary package and have it at hand.

To simplify the recipes in this book, lets put all our projects in a single solution:

1. Create a new Blank Solution in Visual Studio by navigating to **File | New | Project...** (*Ctrl + Shift + N*), search for and select **Blank Solution** by typing that in the search box at the top right of the **New Project** form (*Ctrl + E*).
2. Enter a solution name and location and click on **Ok**.



The recipes in this book will assume that the solution has been named `D3DRendering.sln` and that it is located in `C:\Projects\D3DRendering`.

3. You should now have a new Blank Solution at C:\Projects\D3DRendering\D3DRendering.sln.
4. Extract the contents of the SharpDX package into C:\Projects\D3DRendering\External. The C:\Projects\D3DRendering\External\Bin folder should now exist among others.

How to do it...

With the solution open, let's create a new project:

1. Add a new Windows Form Application project to the solution with .NET Framework 4.5 selected.
2. We will name the project Ch01_01EmptyProject.
3. Add the SharpDX references to the project by selecting the project in the solution explorer and then navigate to **PROJECT | Add Reference** from the main menu. Now click on the **Browse** option on the left and click on the **Browse...** button in **Reference Manager**.
4. For a Direct3D 11.1 project compatible with Windows 7, Windows 8, and Windows 8.1, navigate to C:\Projects\D3DRendering\External\Bin\DirectX11_1-net40 and select **SharpDX.dll**, **SharpDX.DXGI.dll**, and **SharpDX.Direct3D11.dll**.
5. For a Direct3D 11.2 project compatible only with Windows 8.1, navigate to C:\Projects\D3DRendering\External\Bin\DirectX11_2-net40 and add the same references located there.



SharpDX.dll, SharpDX.DXGI.dll, and SharpDX.Direct3D11.dll are the minimum references required to create Direct3D 11 applications with SharpDX.

6. Click on **Ok** in **Reference Manager** to accept the changes.
7. Add the following using directives to Program.cs:

```
using SharpDX;
using SharpDX.Windows;
using SharpDX.DXGI;
using SharpDX.Direct3D11;
// Resolve name conflicts by explicitly stating the class to use:
using Device = SharpDX.Direct3D11.Device;
```

8. In the same source file, replace the `Main()` function with the following code to initialize our Direct3D device and swap chain.

```
[STAThread]
static void Main()
{
    #region Direct3D Initialization
    // Create the window to render to
    Form1 form = new Form1();
    form.Text = "D3DRendering - EmptyProject";
    form.Width = 640;
    form.Height = 480;

    // Declare the device and swapChain vars
    Device device;
    SwapChain swapChain;

    // Create the device and swapchain
    Device.CreateWithSwapChain(
        SharpDX.Direct3D.DriverType.Hardware,
        DeviceCreationFlags.None,
        new [] {
            SharpDX.Direct3D.FeatureLevel.Level_11_1,
            SharpDX.Direct3D.FeatureLevel.Level_11_0,
            SharpDX.Direct3D.FeatureLevel.Level_10_1,
            SharpDX.Direct3D.FeatureLevel.Level_10_0,
        },
        new SwapChainDescription()
        {
            ModeDescription =
                new ModeDescription(
                    form.ClientSize.Width,
                    form.ClientSize.Height,
                    new Rational(60, 1),
                    Format.R8G8B8A8_UNorm
                ),
            SampleDescription = new SampleDescription(1, 0),
            Usage = SharpDX.DXGI.Usage.BackBuffer | Usage.
RenderTargetOutput,
            BufferCount = 1,
            Flags = SwapChainFlags.None,
            IsWindowed = true,
```

```

        OutputHandle = form.Handle,
        SwapEffect = SwapEffect.Discard,
    },
    out device, out swapChain
);

// Create references for backBuffer and renderTargetView
var backBuffer = Texture2D.FromSwapChain<Texture2D>(swapChain,
0);
var renderTargetView = new RenderTargetView(device,
backBuffer);

#endregion

...
}

```

9. Within the same `Main()` function, we now create a simple render loop using a SharpDX utility class `SharpDX.Windows.RenderLoop` that clears the render target with a light blue color.

```

#region Render loop
// Create and run the render loop
RenderLoop.Run(form, () =>
{
    // Clear the render target with light blue
    device.ImmediateContext.ClearRenderTargetView(
        renderTargetView,
        Color.LightBlue);
    // Execute rendering commands here...

    // Present the frame
    swapChain.Present(0, PresentFlags.None);
});
#endregion

```

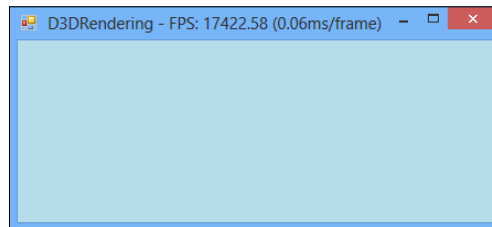
10. And finally, after the render loop we have our code to clean up the Direct3D COM references.

```

#region Direct3D Cleanup
// Release the device and any other resources created
renderTargetView.Dispose();
backBuffer.Dispose();
device.Dispose();
swapChain.Dispose();
#endregion

```


11. Start debugging the project (F5). If all is well, the application will run and show a window like the following screenshot. Nothing very exciting yet but we now have a working device and swap chain.



Output from the empty project

How it works...

We've created a standard Windows Forms Application to simplify the example so that the project can be built on Windows 7, Windows 8, and Windows 8.1.

Adding the `SharpDX.dll` reference to your project provides access to all the common enumerations and structures that have been generated in SharpDX from the Direct3D SDK header files, along with a number of base classes and helpers such as a matrix implementation and the `RenderLoop` we have used. Adding the `SharpDX.DXGI.dll` reference provides access to the DXGI API (where we get our `SwapChain` from), and finally `SharpDX.Direct3D11.dll` provides us with access to the Direct3D 11 types.

The `using` directives added are fairly self-explanatory except perhaps the `SharpDX.Windows` namespace. This contains the implementation for `RenderLoop` and also a `System.Windows.Form` descendant that provides some helpful events for Direct3D applications (for example, when to pause/resume rendering).

When adding the `using` directives, there are sometimes conflicts in type names between namespaces. In this instance there is a definition for the `Device` class in the namespaces `SharpDX.DXGI` and `SharpDX.Direct3D11`. Rather than having to always use fully qualified type names, we can instead explicitly state which type should be used with a device using an alias directive as we have done with:

```
using Device = SharpDX.Direct3D11.Device;
```

Our Direct3D recipes will typically be split into three stages:

- ▶ **Initialization:** This is where we will create the Direct3D device and resources
- ▶ **Render loop:** This is where we will execute our rendering commands and logic
- ▶ **Finalization:** This is where we will cleanup and free any resources

The previous code listing has each of the key lines of code highlighted so that you can easily follow along.

Initialization

First is the creation of a window so that we have a valid handle to provide while creating the `SwapChain` object. We then declare the `device` and `swapChain` variables that will store the output of our call to the static method `Device.CreateDeviceAndSwapChain`.

The creation of the device and swap chain takes place next. This is the first highlighted line in the code listing.

Here we are telling the API to create a Direct3D 11 device using the hardware driver, with no specific flags (the native enumeration for `DeviceCreationFlags` is `D3D11_CREATE_DEVICE_FLAG`) and to use the feature levels available between 11.1 and 10.0. Because we have not used the `Device.CreateDeviceAndSwapChain` override that accepts a `SharpDX.DXGI.Adapter` object instance, the device will be constructed using the first adapter found.

This is a common theme with the `SharpDX` constructors and method overrides, often implementing default behavior or excluding invalid combinations of parameters to simplify their usage, while still providing the option of more detailed control that is necessary with such a complex API.

`SwapChainDescription` (natively `DXGI_SWAP_CHAIN_DESC`) is describing a back buffer that is the same size as the window with a fullscreen refresh rate of 60 Hz. We have specified a format of `SharpDX.DXGI.Format.R8G8B8A8_UNorm`, meaning each pixel will be made up of 32-bits consisting of four 8-bit unsigned normalized values (for example, values between 0.0-1.0 represent the range 0-255) representing Red, Green, Blue, and Alpha respectively. `UNorm` refers to the fact that each of the values stored are normalized to 8-bit values between 0.0 and 1.0, for example, a red component stored in an unsigned byte of 255 is 1 and 127 becomes 0.5. A texture format ending in `_UInt` on the other hand is storing unsigned integer values, and `_Float` is using floating point values. Formats ending in `_SRgb` store gamma-corrected values, the hardware will linearize these values when reading and convert back to the `sRGB` format when writing out pixels.

The back buffer can only be created using a limited number of the available resource formats. The feature level also impacts the formats that can be used. Supported back buffer formats for feature level ≥ 11.0 are:

```
SharpDX.DXGI.Format.R8G8B8A8_UNorm
SharpDX.DXGI.Format.R8G8B8A8_UNorm_SRgb
SharpDX.DXGI.Format.B8G8R8A8_UNorm
SharpDX.DXGI.Format.B8G8R8A8_UNorm_SRgb
SharpDX.DXGI.Format.R16G16B16A16_Float
SharpDX.DXGI.Format.R10G10B10A2_UNorm
SharpDX.DXGI.Format.R10G10B10_Xr_Bias_A2_UNorm
```


We do not want to implement any multisampling of pixels at this time, so we have provided the default sampler mode for no anti-aliasing, that is, one sample and a quality of zero: `new SampleDescription(1, 0)`.

The buffer usage flag is set to indicate that the buffer will be used as a back buffer and as a render-target output resource. The bitwise OR operator can be applied to all flags in Direct3D.

The number of back buffers for the swap chain is set to one and there are no flags that we need to add to modify the swap chain behavior.

With `IsWindowed = true`, we have indicated that the output will be windowed to begin with and we have passed the handle of the form we created earlier for the output window.

The swap effect used is `SwapEffect.Discard`, which will result in the back buffer contents being discarded after each `swapChain.Present`.



Windows Store apps must use a swap effect of `SwapEffect.FlipSequential`, which in turn limits the valid resource formats for the back buffer to one of the following:

- `SharpDX.DXGI.Format.R8G8B8A8_UNorm`
- `SharpDX.DXGI.Format.B8G8R8A8_UNorm`
- `SharpDX.DXGI.Format.R16G16B16A16_Float`

With the device and swap chain initialized, we now retrieve a reference to the back buffer so that we can create `RenderTargetView`. You can see here that we are not creating any new objects. We are simply querying the existing objects for a reference to the applicable Direct3D interfaces. We do still have to dispose of these correctly as the underlying COM reference counters will have been incremented.

Render loop

The next highlighted piece of code is the `SharpDX.Windows.RenderLoop.Run` helper function. This takes our form and `delegate` or `Action` as input, with `delegate` executed within a loop. The loop takes care of all application messages, and will listen for any application close events and exit the loop automatically, for example, if the form is closed. The render loop blocks the thread so that any code located after the call to `RenderLoop.Run` will not be executed until the loop has exited.