Direct2D

Succinctly

by Chris Rose

Syncfusion | Technology Resource Portal

Direct2D Succinctly

By
Chris Rose

Foreword by Daniel Jebaraj

Important licensing information. Please read.

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

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

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

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

Information is plentiful but harder to digest

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

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

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

The *Succinctly* series

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

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

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

The best authors, the best content

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

Free forever

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

Free? What is the catch?

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

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

Let us know what you think

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

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

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

About the Author

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

Introduction

This book is an introduction to some of the capabilities of Direct2D and Direct3D. Direct2D and Direct3D are the graphics rendering components of DirectX. It is about leveraging the graphics card and DirectX to efficiently represent data. It is aimed at programmers already familiar with C++ (both managed and unmanaged) and Visual Studio 2012 Express. We will be using the version of Visual Studio designed for Windows 8 application development, not the desktop version. The desktop version is designed for building standard Windows Forms applications, and the version for Windows 8 is designed for Windows Store applications. This book presents methods for rendering vector graphics and visualizing different types of data on Windows 8 and Windows RT platforms using Direct2D and Direct3D. It is not an in-depth discussion of these topics; for further information, consult the appropriate MSDN library pages from Microsoft along with the specification of the graphics hardware for which you are programming.

This book provides a general introduction to Direct2D and Direct3D. It is written from the perspective of rendering data as nodes and lines, but the information presented is useful for any applications that require efficient rendering using DirectX. In the initial chapters of this book we will develop a small but scalable charting system that can be adapted to suit other projects or incorporated into an existing project. We will examine some common requirements of charting applications, such as detecting if the pointer is near a node, as well as printing Direct2D.

In the interest of keeping things as general as possible, I have generated random data in the examples. In a real situation this data would be loaded from some data source. I will also build on the standard project templates provided by Visual Studio 2012, rather than concentrate on the boilerplate code. The verbose DirectX boilerplate code is a barrier for any programmers hoping to become familiar with the API. Thankfully, the templates supplied with Visual Studio 2012 write all of the boilerplate code for us. We will largely take it for granted, and examine options in the boilerplate code as they arise.

The code in this book is designed for desktop PCs running Windows 8 and tablet PCs running Windows RT. It has been formatted to suit the page of this document. This means it is very difficult to read, and should be reformatted if it is copied and pasted for testing purposes.

Part 1 Direct2D

Direct2D is a graphics API (Application Programming Interface) designed to render 2-D vector and raster graphics. It is built on top of the Direct3D API, which in turn is built on the DXGI (DirectX Graphics Infrastructure). It can be used in conjunction with Direct3D to render any 2-D portions of a scene. It is high performance, leveraging the GPU for efficient, complex 2-D graphics.

Note: Throughout this book I will refer to GPU many times (short for Graphics Processing Unit). The term GPU usually refers to a dedicated graphics card, however, I will use the term more generally to refer to the hardware which performs the majority of graphics processing in a computer. This includes a dedicated graphics card, an onboard graphics card, or the execution units in the NVidia Tegra chips in the WinRT devices.

The API consists of a number of interfaces (COM objects), which are used to communicate with the graphics hardware. It can render vector primitives, like lines and ellipses, and can also fill shapes with solid colors or gradients, as well as display raster images. Raster graphics are composed of pixels, one for each point on a screen (or image). The pixels each have values which determine their colors, and collectively they are arranged in a large grid.

Direct2D is important for visualizing data because many chart types (line charts, scatter plots, etc.) are fundamentally 2-D in design. The most important difference between using Direct2D and using Direct3D to render 2-D graphics is simplicity. Direct3D is orders of magnitude faster than Direct2D but it is more complicated to program. In addition to this, the Direct2D project template is a perfect combination of standard Windows 8 XAML and Direct2D. This allows programmers to use standard Windows 8 controls and XAML pages to deal with user input, while Direct2D handles all the graphics processing. This combination of DirectX and XAML is a feature only available in Windows 8 applications.

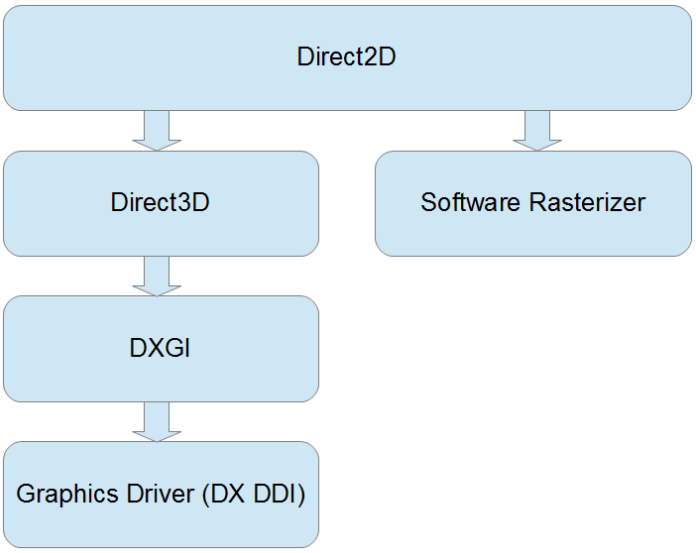


Figure 1: Relationship between Major DirectX Components

The graphics driver is the lowest level depicted; it controls the hardware directly. Above this is the DirectX Graphics Infrastructure (DXGI), then Direct3D and finally Direct2D. The software rasterizer is used in place of graphics hardware, it uses the CPU to render graphics when a dedicated GPU is not available.

Chapter 1: Direct2D (XAML) Template

We will begin by creating a standard Direct2D (XAML) template project and becoming familiar with its structure. Open Visual Studio 2012 and on the **File Menu**, click **New Project**.

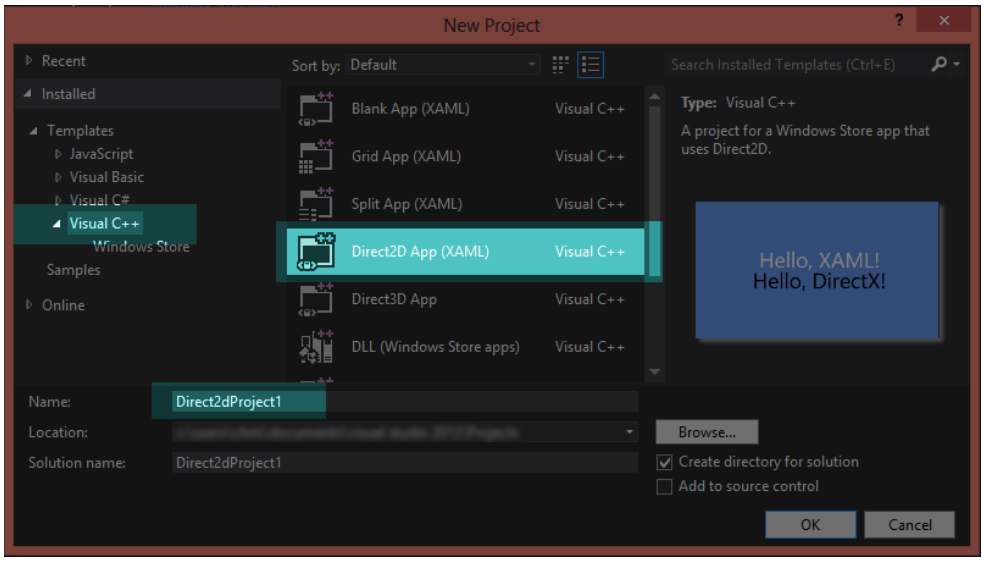


Figure 2: Creating a new Direct2D App (XAML)

Click **Visual C++** on the left panel, and then select **Direct2D App (XAML)** from the project templates in the center panel. Type a name for your project in the **Name** box, and then click **OK**.

Visual Studio will create many files for the new project which contain the boilerplate code and some other useful helper methods. The Solution Explorer should look like Figure 3.

To run the application in debug mode press F5, or on the **File** menu click **Debug** > **Start Debugging**. After Visual Studio builds and links your project files, it will execute the application.

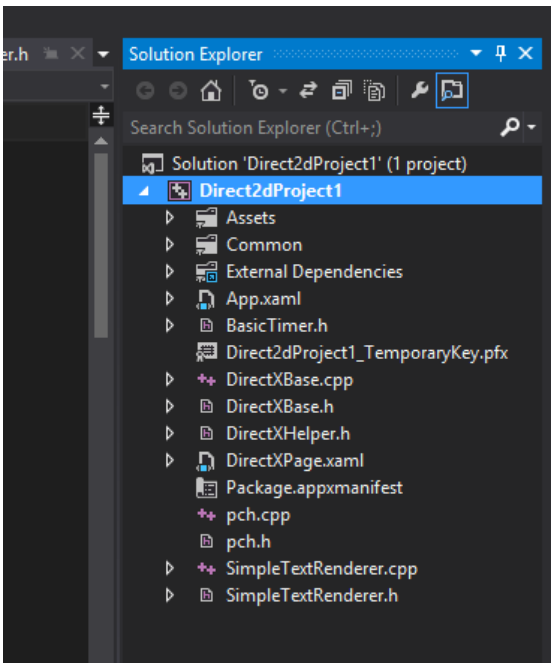


Figure 3: Direct2D App (XAML) Solution Explorer

Figure 4: Output of Direct2D App (XAML) Template

Assets Folder

This folder contains several PNG images for the new application:

- Logo.png: This image appears as the tile on the Windows 8 Start page. It is similar to the desktop icon from previous versions of Windows.
- SmallLogo.png: This is the icon image used when a smaller icon should be displayed, such as when the user is searching 'All Apps' in Windows 8.
- SplashScreen.png: The splash screen appears briefly when your application is executed.
- StoreLogo.png: This is the logo for your app as it appears in the Windows Store.

Common

This folder contains a single XAML file that describes common settings between the XAML files.

External Dependencies:

This folder contains a very large list of external files that your project may depend on. Some are generated per project, and others are the standard Windows C++ header files. You should not change the files in this list, especially the standard Windows headers.

App

The App.xaml, App.cpp, and App.h files define your application. The XAML file contains some global settings across your entire app. The CPP and H files define a class with the starting point for executing the program. This class owns a member variable called m_directXPage which is the main Direct2D rendering class. It also controls some important system-level operations, like saving and restoring the state of the application when the program is suspended.

BasicTimer.h

The basic timer header defines a class that can be used for any time-based tasks such as physics or animation.

xxx_TemporaryKey.pfx

This is the ClickOnce digital certificate for your application. It is used to help ensure that the application is not malicious software. If the application is not signed, Windows will warn users that the application "comes from an unknown publisher," and it will ask them if they are sure they wish to run the program.

DirectXBase

The DirectXBase class is defined in two files: DirectXBase.h and DirectXBase.cpp. This class contains most of the boilerplate code to get Direct2D up and running. It contains code to initialize the device, the factories, device context, and many other things. It can be used for both 2-D and 3-D graphics. It has many helper functions to enable us to quickly begin DirectX programming without typing the extremely verbose boilerplate code. The reader is encouraged to investigate this file thoroughly, as it shows exactly how DirectX should be initialized.

DirectXHelper

This file consists of a single function, DX::ThrowIfFailed. This is a helper function that converts an HRESULT to a managed C++ exception. DirectX function calls return an HRESULT. Many of the codes we will examine surround the DirectX function calls with a call to this method, such that the programmer has an opportunity to examine any errors that are thrown by DirectX. If you set a break point on this line, Visual Studio will break when an exception is thrown, and allow you to examine what went wrong. The errors will give you an error number and you can research the meaning of this, or look it up using the error look up application that comes with the DirectX SDK.

DirectXPage

This is the main XAML page of your application. The Direct2D (XAML) template application contains a simple page with two sentences written on a XAML form. The top sentence is written using XAML and the lower one is written by DirectX. This is the class that renders the top line of code.

Package.appxmanifest

This is the main manifest of your application. It contains all the information about your app, including who the publisher is, and what capabilities the app requires (Internet access, access to the webcam, etc.).

PCH

The precompiled header file (pch.h) contains headers that are compiled to an intermediate format to save time when recompiling the entire project. Most of the classes you add to your project will include this file in order to work correctly.

SimpleTextRenderer

This is the core class of this DirectX application. This class renders the lower sentence on the screen. Because the SimpleTextRenderer class is the main class controlling what DirectX displays on the screen, we will examine it in detail.

SimpleTextRenderer Class

This class uses Direct2D to render a line of text to the screen. In this section, it is not the class itself we are examining, but rather the way that it operates. The Graph Renderer class we will build in future chapters will be heavily based on this class. Open the SimpleTextRenderer.h file.

The class derives from the DirectXBase class. It contains a default constructor and several methods, which are called during resource allocation (**`CreateDeviceIndependentResources`**, **`CreateDeviceResources`** and **`CreateWindowSizeDependentResources`**).

Note: Resources is a general term referring to many different types of objects and structures that are stored in memory (either system memory or in the GPU's dedicated memory) and used by DirectX. Resources must be created and initialized prior to their use. Most of the resources we will examine are created shortly after the main DirectX objects. These resources are destroyed when the application closes. Resources can be created and destroyed at any time after the main DirectX objects are initialized, since the resource creation methods belong to these objects.

The Render method is where DirectX does all of its rendering. This class also defines an update method which can be used to perform calculations to determine where objects should be moved to in the scene. The **`UpdateTextPosition`**, **`BackgroundNextColor`**, and **`BackgroundPreviousColor`** functions are specific to this template, and not required when you develop your own. They allow the user to manipulate the position of the DirectX drawn text, as well as cycle through some predefined background colors.

The **`SaveInternalState`** and **`LoadInternalState`** methods are used to save and restore the state of the application; for example, when a WinRT tablet goes to sleep, and then is woken.

These methods are followed by several member variables that are used to maintain and manipulate the position of the text. Apart from the **`m_renderNeeded`** variable, most of these variables are application specific and most likely not required for your application. The **`m_renderNeeded`** variable is used by the application to determine if the Render method should be called. If nothing has changed in the scene, there is no point in rendering it again. The following diagram depicts the relationships between the most important classes of this application. Lines ending in diamond shapes indicate ownership (the AppXAML class owns a member of the type DirectXPage), and the lines ending with a triangle indicate inheritance (the SimpleTextRenderer inherits from the DirectXBase class).

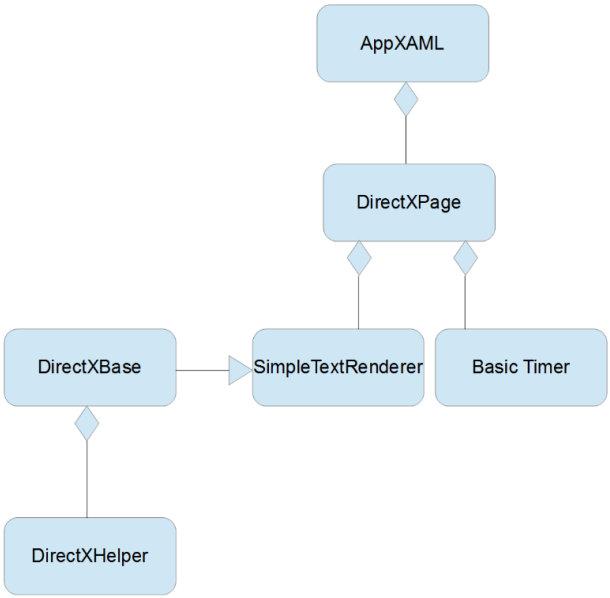


Figure 5: Class Relationships of Direct2D App (XAML) Template

Real-time graphics applications often render frames at some predefined interval described with the metric of the number frames displayed per second (FPS). A frame is a single still image of a game or movie. In order to create the illusion of smooth animation, slightly different frames are displayed to the viewer in succession. DirectX applications often render frames at a fixed refresh rate, such as 60fps or even 100fps. It is not likely that the frames of a charting application need to be re-rendered every 60th or 100th of a second. They usually stay exactly the same for long periods of time. The user may pan or zoom into the chart which would require a re-rendering of the scene, but this action is not as time critical as updating the frames of a real-time game.

Note: The member variables for this and other classes in this template have an "m_" prefix. This signifies that they are member variables as opposed to local variables to a function. It is not necessary but it is a good idea to name all member variables with this prefix.

Next, open the SimpleTextRenderer.cpp file in the Solution Explorer. At the top of the file you will see an **#include** directive for the Precompiled Header (**pch.h**). Below this is the **#include** for the **SimpleTextRenderer.h**, and the list of namespaces the class uses. Under the using directives you will see the predefined order of the background colors that the user can cycle while running the application.

```
static const ColorF BackgroundColors[] = { …  }
```

The user can cycle through the colors and change the background of the application by right-clicking the mouse in the screen, or swiping the pointer if you are using a touch screen device, and selecting **Next** or **Previous**. This is an application-specific array, and it is unlikely that other applications will use it. Below this we see the default constructor for the class.

```
SimpleTextRenderer::SimpleTextRenderer():m_renderNeeded(true),
      m_backgroundColorIndex(0),m_textPosition(0.0f, 0.0f) { }
```

The default constructor initializes several variables; it sets the **backcolor** to **CornflowerBlue** by selecting index 0 (this is a reference to the **BackgroundColors** array defined in the previous code sample). It also initializes the text position and sets the m_**renderNeeded** Boolean to true, such that the first frame will be drawn to the screen. Resources are not created or allocated at this point; the DirectX factories and context do not yet exist either.

Following this are three resource allocation methods. The first of which is the **CreateDeviceIndependentResources** method.

```
void SimpleTextRenderer::CreateDeviceIndependentResources() {
      DirectXBase::CreateDeviceIndependentResources();
      DX::ThrowIfFailed(
            m_dwriteFactory->CreateTextFormat(
                  L"Segoe UI", nullptr, DWRITE_FONT_WEIGHT_NORMAL,
                  DWRITE_FONT_STYLE_NORMAL, DWRITE_FONT_STRETCH_NORMAL,
                  42.0f, L"en-US", &m_textFormat));
      DX::ThrowIfFailed(
            m_textFormat->SetTextAlignment(DWRITE_TEXT_ALIGNMENT_LEADING)
            );
}
```

The **CreateDeviceIndependentResources** method is used to create and initialize any Direct2D objects that are device independent. This method begins by calling the base class's method of the same name. The base class method creates the DirectX factories, such as the **m_dwriteFactory** used on the next line, which can be used by the application to create more DirectX objects.

Note: Resources in DirectX are all from one of two broad categories: device resources or device independent resources. The device is the graphics card, and the device resources reside on the graphics card itself. Device independent resources reside in system RAM, and tend to render slower because they require CPU cycles to transfer to the video card.

```
void SimpleTextRenderer::CreateDeviceResources() {
      DirectXBase::CreateDeviceResources();
      DX::ThrowIfFailed(
            m_d2dContext->CreateSolidColorBrush(
                  ColorF(ColorF::Black), &m_blackBrush));
      Platform::String^ text = "Hello, DirectX!";
      DX::ThrowIfFailed(m_dwriteFactory->CreateTextLayout(
                  text->Data(), text->Length(),
                  m_textFormat.Get(),
                  700, // maxWidth.
                  1000, // maxHeight.
                  &m_textLayout));
      DX::ThrowIfFailed(m_textLayout->GetMetrics(&m_textMetrics));
}
```

The **CreateDeviceResources** method creates and initializes the device dependent resources. The method calls the base class method with the same name, which creates the instance of the Direct3D device and the context used by the application (**m_d3dcontext** and **m_d3dDevice**).

Note: Device and context are two important terms in DirectX. The device can be thought of as the graphics card itself; this class is used to initialize the hardware, query its capabilities, and create resources such as textures and shaders. A context is a particular use of the device; it renders things to the screen using the resources on the device. There is normally one device, but there may be more than one context. For instance, the printing sample uses three contexts: one for rendering, another for the print preview, and a third for the printing itself. Figure 6 shows some of the tasks each of these classes is responsible for.

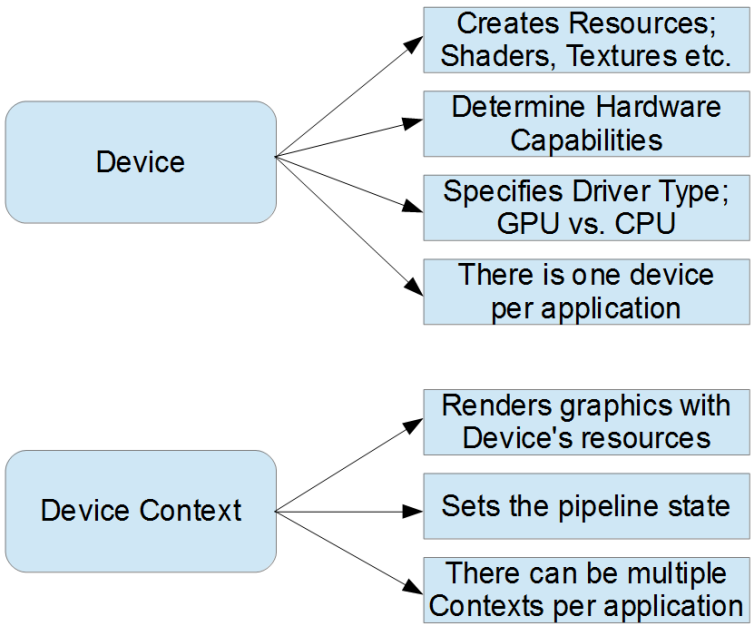


Figure 6: Device versus Context

Brushes are device resources; this method creates a black brush for painting the text. The actual string to be written to the screen is created on the device as a **`TextLayout`** object using the **`CreateTextLayout`** method. After this, the measurements and proportions of the string are saved to **`m_textMetrics`** using the **`GetMetrics`** method.

Note: The CreateTextLayout method creates the IDWriteTextLayout device resource. This resource contains information about the string to be printed, the bounding box within which it is printed and its location. The CreateTextFormat method (in the CreateDeviceIndependentResources method) creates an IDWriteTextFormat object, which is used to specify the font, size, and attributes of the text to render.

```
void SimpleTextRenderer::CreateWindowSizeDependentResources() {

      DirectXBase::CreateWindowSizeDependentResources();

      // Add code to create window size dependent objects here.

}


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

      (void) timeTotal; // Unused parameter.

      (void) timeDelta; // Unused parameter.

      // Add code to update time dependent objects here.

}
```

The previous two methods are empty in the template. The **CreateWindowSizeDependentResources** method is used to create any objects (device or device independent) whose settings are dependent on the size or orientation of the screen. The **Update** method is also empty; it controls the physics or other logic of the application, usually things that are time dependent. The following code is an example of the template's **Render** method.

```
void SimpleTextRenderer::Render() {

      m_d2dContext->BeginDraw();

      m_d2dContext->Clear(ColorF(BackgroundColors[m_backgroundColorIndex]));

      // Position the rendered text.

      Matrix3x2F translation = Matrix3x2F::Translation(

            m_windowBounds.Width / 2.0f –

            m_textMetrics.widthIncludingTrailingWhitespace / 2.0f +

            m_textPosition.X,

            m_windowBounds.Height / 2.0f –

            m_textMetrics.height / 2.0f + m_textPosition.Y

            );


      // Note that the m_orientationTransform2D matrix is post-multiplied
here

      // in order to correctly orient the text to match the display
orientation.

      // This post-multiplication step is required for any draw calls that
are

      // made to the swap chain's target bitmap. For draw calls to other
targets,

      // this transform should not be applied.

      m_d2dContext->SetTransform(translation * m_orientationTransform2D);

      m_d2dContext->DrawTextLayout(Point2F(0.0f, 0.0f),

            m_textLayout.Get(),     m_blackBrush.Get(),

            D2D1_DRAW_TEXT_OPTIONS_NO_SNAP);
```

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

It is here in the Render method that actual drawing of the scene takes place. Most of the drawing of the scene is performed by the m_d2dContext object. The Render method begins by stating **m_d2dcontext->BeginDraw**; this line is coupled to the call to **m_d2dContext->EndDraw** method call near the bottom. You should place all of your Direct2D drawing between these two function calls. BeginDraw is used to specify the start of some code which builds a batch of rendering commands for a render target. EndDraw specifies that the batch of commands is finished and they can be rendered.

The next line calls the **Clear** method, passing the color the user currently has selected. This results in clearing the screen to a solid color, one that the **BackgroundColors** array defined previously, which the user can cycle through.

Tip: It is a good idea to clear the screen to some color other than black in a render method, even if your subsequent drawing will completely overwrite the cleared screen. If you do not do this and there is a problem with the program, you might be left staring at a black screen (or random flashing colors or pixels) with no way of telling whether the render method is being called at all.

Following the call to **Clear**, a matrix is set up. Transforms such as scaling, rotation, and translation (or panning, which is what we are doing here) are all controlled by matrices. This particular matrix is a translation matrix; it moves the text such that the user can drag it around the screen. The calculation in the definition of this matrix places the text in the middle of the screen with some offset when the user drags it around. It uses the **TextMetrics** object and the **WindowBounds** object to find where the text should go.

Once defined, the translation matrix must be applied to the context. This occurs on the next line with the call to **SetTransform**. After the appropriate transformations have been applied, the text itself can be rendered. This happens on the next line with the call to **DrawTextLayout**. Then the drawing is ended with the call to **EndDraw**, and the image is presented to the user.

Tip: The actual screen refresh of the rendered scene occurs in the DirectXPage.xaml.cpp file when the m_renderer object calls Present() in its OnRendering event handler method. It is very important to note that the DirectXPage class presents the scene. When you add more sophisticated rendering classes that call Present() themselves, it is important that you remove this Present() call from the DirectXPage class. Otherwise you might Present() twice which will result in first flipping the actual scene to the screen but immediately overwriting it with some other image.

The remaining methods are event handlers and other things which are specific to this. I recommend that programmers new to DirectX with Visual Studio 2012 spend some time altering the workings of this template before continuing on to the next section. A good familiarity with this template is essential to understanding the remaining chapters of the Direct2D portion of this book.

Tip: Direct2D is designed to use multiple cores of the CPU automatically when rendering geometry. If you use the D2D1_DEVICE_CONTEXT_OPTIONS_ENABLE_MULTITHREADED_OPTIMIZATIONS option when creating the device context in the DirectXBase.cpp file, automatic multithreading may provide a good speed boost to your code at the cost of utilizing more of the system's cores.

VSync, Swap Chain, and Buffering

Computer monitors update their display at a fixed speed. 60 times per second is common, referred to as 60 Hz, but there are others like 75 Hz and 100 Hz. The pixel data is stored in a buffer on the GPU, which is called the front buffer. The image on the monitor is refreshed with the data from this buffer 60 times per second. At the same time the monitor is refreshing its display, the GPU is busy rendering the frames to be displayed. The GPU writes the pixel data to the buffer.

There is a problem with this system which leads to unpleasant artifacts. The trouble is that the GPU and the monitor are not necessarily updating frames at the same speed. This leads to an artifact called tearing (see Figure 7). The monitor draws half of one frame to the screen and half of the previous frame, because the GPU updates the frame in the front buffer when the monitor is partially through updating its display.

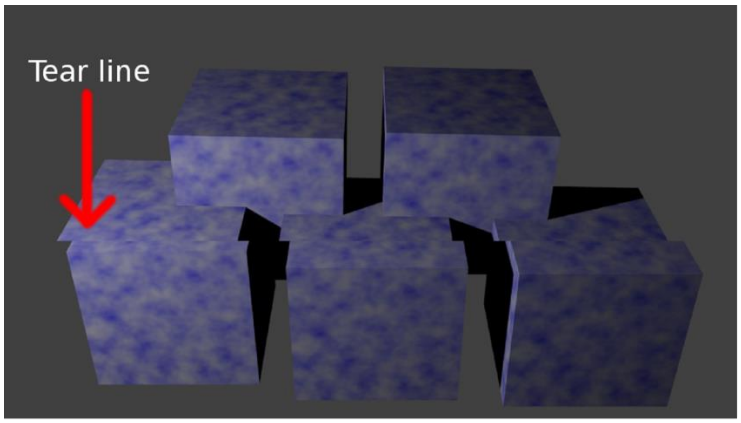


Figure 7: Tearing

To get around this, the GPU does not render to the front buffer. Instead, it renders to a back buffer, which is identical to the front buffer in every aspect, except that it is not rendered to the screen.

The monitor refreshes its display by rendering pixels from the upper left corner of the screen to the lower right corner, then it resets and repeats the operation. The time period in which it resets itself from the lower right corner back to the top is called the vertical retrace. To avoid the tearing artifact, the GPU waits for the monitor to be in this vertical retrace phase, then it flips the buffers (swaps the back and front buffers either by copying the pixel data or swapping pointers). This is called vertical synchronization or V-sync for short. By the time the monitor has finished resetting itself, the GPU can copy an entire frame to the front buffer. This way there is no tearing and the monitor will never display half of one frame and half of another.

The buffers are coordinated using a swap chain object. This is a class dedicated to controlling the swapping of the buffers. In our applications there are two buffers: the front buffer and the back buffer. Sometimes it is beneficial to use more than one back buffer and render frames to each, one after the other, queuing the frames to be presented.

Chapter 2: Debugging with a WinRT Device

All of the code in this book works for Windows 8 PCs as well as WinRT devices. If you are authoring software for a WinRT tablet and have a real device, it is very beneficial to use it for debugging and testing your application instead of an emulator (which is usually the default). Most of the code in C++ and DirectX works fine on a Windows 8 PC, as well as a WinRT device (compiled for the ARM target). The emulators are good but can never match the exact characteristics of a real device.

Install the Remote Tools

Install the remote tools for Visual Studio 2012 onto the device. This is available from the Microsoft website (available from http://www.microsoft.com/visualstudio/eng/downloads#d-additional-software). It is a service that connects with the Visual Studio development machine to run and debug the app on the device. All the regular debugging mechanisms are available from Visual Studio such as break points, examining the ARM registers, and Memory windows. You need to know the name of the device in order to deploy an application onto it. You also need to have the device run the Remove Debugging Monitor that comes with the previously mentioned installation. Each build configuration (Release x86, Debug x86, Release ARM, etc.) you want the device to run must have the device's name in its project settings.

Change the application to ARM

If the WinRT device that you are deploying to is ARM based, such as a Microsoft Surface, you can change the configuration for the project from the main menu by selecting **ARM**.

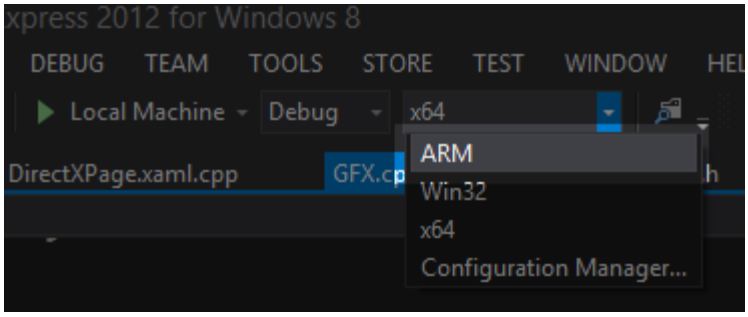


Figure 8: Configurations

Change debugging to Remote Machine

If it is not set already, you should change the debugging to Remote Machine.

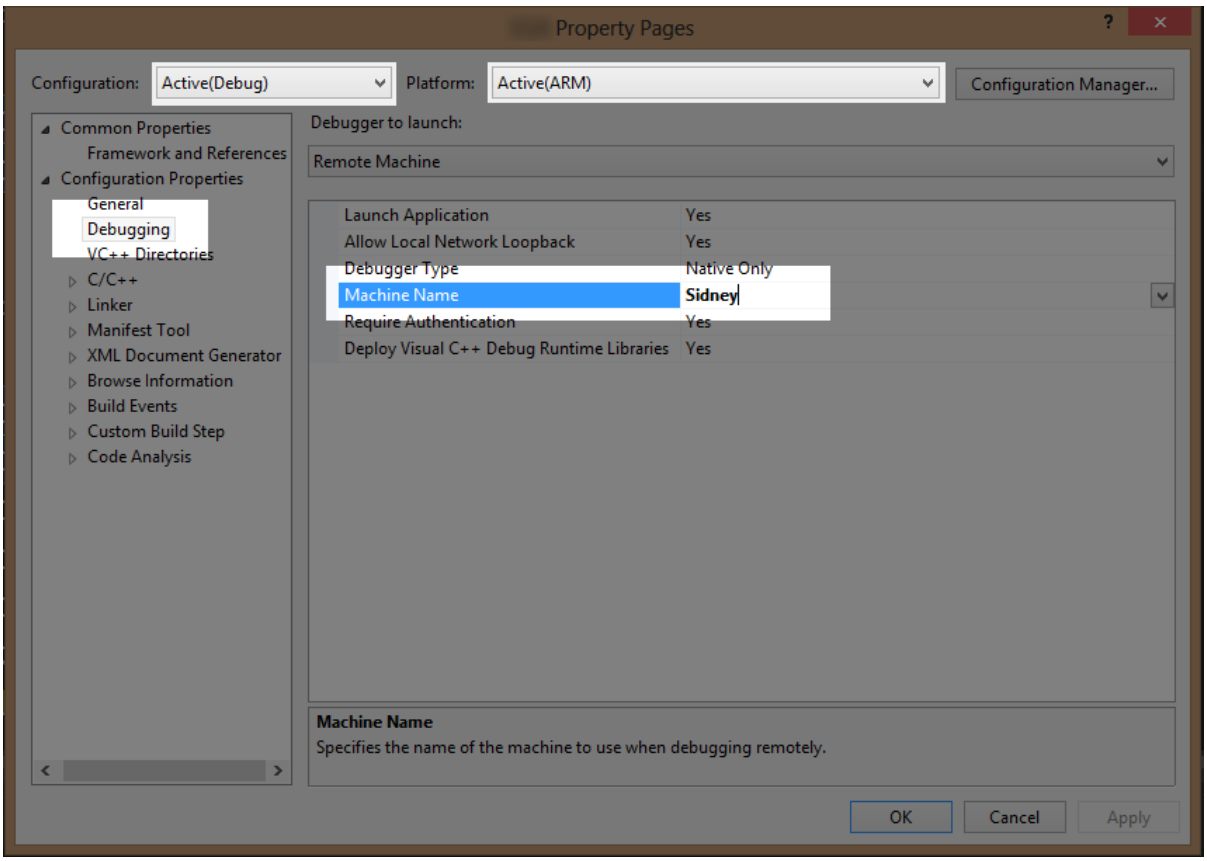


Figure 9: Machine Name

Specify the Name of the Remote Machine

Open the **Project > [Name] Properties** page from the main menu of Visual Studio or right-click on your project's name in the Solution Explorer and click **Properties** on the context menu. This will open the properties page for the project. Click **Debugging** on the left panel and type the name of your remote machine into the space labeled **Machine Name**.

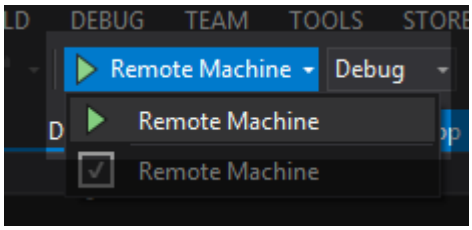


Figure 10: Remote Machine

Run the Remote Debugger

Run the remote debugging service on the device and you should be able to start debugging from Visual Studio 2012 as usual (press F5 or click the start debugging button). The first thing you will see on the device (Visual Studio Remote debugger's window) says it is connected to the development computer with a message like the following:

```
3/01/2013 2:48:40 PM [MachineName]\[ComputerName] connected
```

Shortly after this you will see a message in the output window of Visual Studio saying it is uploading the program to the device. This takes some time, but once the upload is complete the application should run.

Here are some ideas if you are unable to debug the application from the device, or it does not run as expected:

- Make sure you have the correct remote debugging tools installed on the device. Install the tools for Visual Studio 2012. Always download this directly from the Microsoft website and download any available updates to ensure the current remote debugging supports your particular device.
- Make sure you have spelled the name of the remote machine correctly in the project properties. The remote machine name was chosen when Windows RT was first installed on the machine. You can see the name of the remote machine in the Remote Debugger window if you have forgotten or are unsure what the remote machine is called. At present, the case of the name in the properties of the project is irrelevant, but the machine uses all uppercase so you might try to match the exact case the machine is using.
- Make sure the current configuration has the name of the remote machine specified in its debugging field in the properties page. You need to put the name of the device in each configuration. For instance, if you use Debug and Release, you need to specify the remote machine's name in both.

Finally, if the application is not executing as expected but is running, ensure the code you have used is completely portable to WinRT. Be aware that these devices do not have a dedicated graphics card. They rely on a scaled down, portable, and energy efficient CPU/GPU combination. The version of DirectX 11 which runs on these devices is also scaled down. It does not contain the full capabilities of the DirectX 11 standard. The operating system itself (Windows RT) is a scaled down version of the full Windows 8, and many features are missing (free access to the file structure, for instance).

Chapter 3: Beginning a Graph Rendering App

Figure 11 is a basic bar chart. This particular one was generated using Open Office Calc with random data. It consists of a title, background, axis labels, grid, key, and bars representing the data.

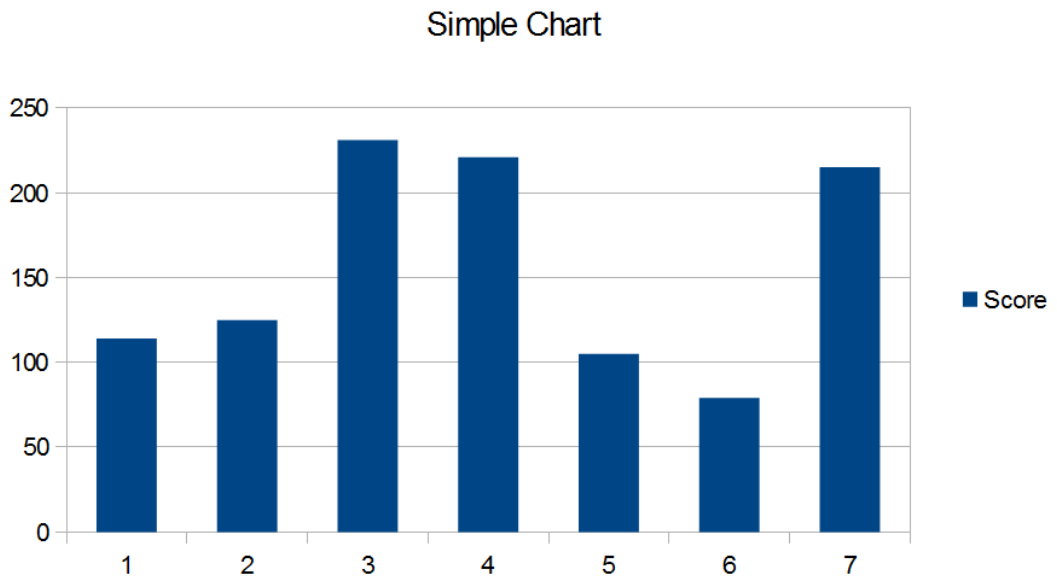


Figure 11: Bar Chart

Each part of the chart can be thought of as being a distinct object. Each object is rendered one after the other, starting with the background followed by the grid, the data, and then the labels. The graph itself is composed of several objects which it draws one after the other to build a complete graphical representation of the data. Many things about the previous chart are generic and applicable to different chart types. The grid, for example, could be used for a scatter plot, line chart, or histogram exactly as it is used here.

Our charting application will work in the same way. We will develop a collection of chart objects that can be added and removed from charts at will. The objects will be very basic to maintain a generic and usable foundation for a Direct2D charting application. The graph itself will be a class called GraphRenderer, which will be based on the SimpleTextRenderer class that we just examined. Each of the objects comprising the GraphRenderer will be a scaled down version of the SimpleTextRenderer.

Create a new Direct2D (XAML) application in Visual Studio 2012 for Windows 8. This will form the starting point for our application. I have called my application GraphPlotting. You will need to change any references to this namespace to the name of your application if you copy the code for testing.

First, we can delete the XAML text on the form. Open the DirectXPage.xaml file by double-clicking its name in the Solution Explorer. This should display the page in a visual designer. Select **Hello, XAML!** and right-click. Click **Delete** on the context menu.

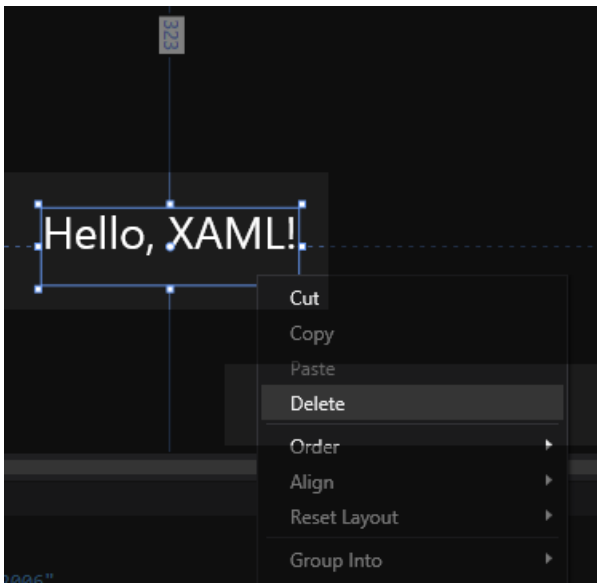


Figure 12: Deleting Text

There is also a hidden bar at the lower side of this panel which can be deleted. It is the bar that appears when the user right-clicks on the screen, allowing the background colors to change. This bar is not visible from the designer, so it is easiest to remove it from the XAML code. I have highlighted the lines to remove.

```
<Page

    x:Class="GraphPlotting.DirectXPage"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

    xmlns:local="using:GraphPlotting"

    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

    mc:Ignorable="d">

    <SwapChainBackgroundPanel x:Name="SwapChainPanel"
PointerMoved="OnPointerMoved" PointerReleased="OnPointerReleased"/>

    <Page.BottomAppBar>

        <AppBar Padding="10,0,10,0">

            <Grid>

            … Lots of XAML code here…!!!

            </Grid>
```

```
        </AppBar>
    </Page.BottomAppBar>
</Page>
```

Open the **DirectXPage.xaml.cpp** file. Delete the **OnPreviousColorPressed** and **OnNextColorPressed** event handlers. In the constructor of this class, a SimpleTextRenderer object is created. We need to change this to a GraphRenderer constructor call. You can also delete the state saving methods, **SaveInternalState** and **LoadInternalState**. The GraphRenderer class uses a method called **PointerMoved** instead of **UpdateTextPosition**. This method has been renamed in the code listing. The entire DirectXPage.xaml.cpp file should look like the following:

```
// DirectXPage.xaml.cpp
#include "pch.h"
#include "DirectXPage.xaml.h"
using namespace GraphPlotting;
using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::Graphics::Display;
using namespace Windows::UI::Input;
using namespace Windows::UI::Core;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;

DirectXPage::DirectXPage() : m_renderNeeded(true), m_lastPointValid(false) {
      InitializeComponent();
```

```
	m_renderer = ref new GraphRenderer();
	m_renderer->Initialize(Window::Current->CoreWindow,  SwapChainPanel,
		DisplayProperties::LogicalDpi);

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

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

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

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

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

void DirectXPage::OnPointerMoved(Object^ sender, PointerRoutedEventArgs^
args) {
	auto currentPoint = args->GetCurrentPoint(nullptr);
```

```
      if (currentPoint->IsInContact) {
            if (m_lastPointValid) {
                  Windows::Foundation::Point delta(
                        currentPoint->Position.X - m_lastPoint.X,
                        currentPoint->Position.Y - m_lastPoint.Y
                        );
                  m_renderer->PointerMoved(delta);
                  m_renderNeeded = true;
            }
            m_lastPoint = currentPoint->Position;
            m_lastPointValid = true;
      }
      else {
            m_lastPointValid = false;
      }
}

void DirectXPage::OnPointerReleased(Object^ sender, PointerRoutedEventArgs^
args) {
      m_lastPointValid = false;
}

void DirectXPage::OnWindowSizeChanged(CoreWindow^ sender,
            WindowSizeChangedEventArgs^ args) {
      m_renderer->UpdateForWindowSizeChange();
      m_renderNeeded = true;
}
void DirectXPage::OnLogicalDpiChanged(Object^ sender) {
      m_renderer->SetDpi(DisplayProperties::LogicalDpi);
```

```
	m_renderNeeded = true;
}
void DirectXPage::OnOrientationChanged(Object^ sender) {
	m_renderer->UpdateForWindowSizeChange();
	m_renderNeeded = true;
}
void DirectXPage::OnDisplayContentsInvalidated(Object^ sender) {
	m_renderer->ValidateDevice();
	m_renderNeeded = true;
}
void DirectXPage::OnRendering(Object^ sender, Object^ args) {
	if (m_renderNeeded)    // Comment out this line to make real-time
updating
	{
		m_timer->Update();
		m_renderer->Update(m_timer->Total, m_timer->Delta);
		m_renderer->Render();
		m_renderer->Present();
		m_renderNeeded = false;
	}
}
```

Open the DirectXPage.xaml.h file and delete the declarations of the **OnPreviousColorPressed** and **OnNextColorPressed** event handlers that we removed from the CPP file, and change the include from SimpleTextRenderer.h to GraphRenderer.h, and the member variable declaration from SimpleTextRenderer^ to GraphRenderer^. Delete the declarations for the SaveInternalState and LoadInternalState methods as well. The file should look like the following (Visual Studio will underline references in red as we are yet to declare the GraphRenderer class).

```
// DirectXPage.xaml.h
#pragma once
#include "DirectXPage.g.h"
```

```
#include "GraphRenderer.h"
#include "BasicTimer.h"
namespace GraphPlotting{
    [Windows::Foundation::Metadata::WebHostHidden]
       public ref class DirectXPage sealed {
       public:
              DirectXPage();
       private:
              void OnPointerMoved(Platform::Object^ sender,
                     Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
              void OnPointerReleased(Platform::Object^ sender,
                     Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
              void OnWindowSizeChanged(Windows::UI::Core::CoreWindow^ sender,
                     Windows::UI::Core::WindowSizeChangedEventArgs^ args);
              void OnLogicalDpiChanged(Platform::Object^ sender);
              void OnOrientationChanged(Platform::Object^ sender);
              void OnDisplayContentsInvalidated(Platform::Object^ sender);
              void OnRendering(Object^ sender, Object^ args);
              Windows::Foundation::EventRegistrationToken m_eventToken;
              GraphRenderer^ m_renderer;
              bool m_renderNeeded;
              Windows::Foundation::Point m_lastPoint;
              bool m_lastPointValid;
              BasicTimer^ m_timer;
       };
}
```

Open the **App.xaml.cpp** file, and remove the two references to the **`LoadInternalState`** and **`SaveInternalState`** methods:

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

```
#include "pch.h"
#include "DirectXPage.xaml.h"
using namespace GraphPlotting;
using namespace Platform;
using namespace Windows::ApplicationModel;
using namespace Windows::ApplicationModel::Activation;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::Storage;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Interop;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;


App::App() {
	InitializeComponent();
	Suspending += ref new SuspendingEventHandler(this, &App::OnSuspending);
}


void App::OnLaunched(LaunchActivatedEventArgs^ args) {
	m_directXPage = ref new DirectXPage();
	// Place the page in the current window and ensure that it is active.
	Window::Current->Content = m_directXPage;
	Window::Current->Activate();
}
```

```
void App::OnSuspending(Object^ sender, SuspendingEventArgs^ args) {

	(void) sender; // Unused parameter.

	(void) args; // Unused parameter.

}
```

The following GraphRenderer class will take the place of the SimpleTextRenderer supplied in the Direct2D (XAML) template, so that we can delete the SimpleTextRenderer from our project. Select the two files that define the SimpleTextRenderer (SimpleTextRenderer.h and SimpleTextRenderer.cpp) in the Solution Explorer. To delete them, right-click and select **Remove** from the context menu.

Add two files to the project, GraphRenderer.h and GraphRenderer.cpp. These files will define our graph renderer class. These files will change often as our charts evolve, but the following is their initial listing.

```
// GraphRenderer.h

#pragma once

#include "DirectXBase.h"

//

// Additional headers for graph objects here

//


// This class represents a graph

ref class GraphRenderer sealed : public DirectXBase{

public:

	// Public constructor

	GraphRenderer();


	// DirectXBase methods.

	virtual void CreateDeviceIndependentResources() override;

	virtual void CreateDeviceResources() override;

	virtual void CreateWindowSizeDependentResources() override;

	virtual void Render() override;


	// Capture the pointer movements so the user can pan the chart
```

```
        void PointerMoved(Windows::Foundation::Point point);

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

private:
        // Global pan value for moving the chart with the mouse
        Windows::Foundation::Point m_pan;
};
// GraphRenderer.cpp
#include "pch.h"
#include "GraphRenderer.h"

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

GraphRenderer::GraphRenderer() {
}

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

void GraphRenderer::CreateDeviceResources() {
DirectXBase::CreateDeviceResources();
}
```

```
void GraphRenderer::CreateWindowSizeDependentResources() {
DirectXBase::CreateWindowSizeDependentResources();
}

void GraphRenderer::Update(float timeTotal, float timeDelta) {
}

void GraphRenderer::PointerMoved(Windows::Foundation::Point point)
{
// Allow the user to set the current pan value with the mouse or pointer
m_pan.X += point.X;
m_pan.Y += point.Y;
}
```

Compile and test your application at this point. You should see the entire screen cleared to a light blue color.

Chapter 4: Graph Backgrounds

The first graph objects we define will be backgrounds. The background of a chart acts as the canvas upon which the other objects are rendered. It can be a simple single color, a gradient, or even an image. Charts are usually meant to clearly portray information, and the background should not obscure the data.

Solid Color Background

The simplest chart background is a single solid color, usually white or some other unsaturated pigment. These are common because they do not tend to draw the attention of the viewer away from the data being represented, and they are quick and easy to render.

Note: We could change the color in the call to Clear from `CornflowerBlue` to something else. Instead, we will encapsulate the rendering of the background in a separate class. Once our chart is clearing the screen, you can remove the clear to CornflowerBlue.

The following code defines a class that renders a solid color background.

```
// SolidBackground.h

#pragma once

#include "DirectXBase.h"

// Defines a background consisting of a solid color

class SolidBackground {

private:

	D2D1::ColorF color; // The color of this background

public:

	// Creates a new SolidBackground set to the specified color

	SolidBackground(D2D1::ColorF color);

	// Draw the background

	void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);

};
```

```
// SolidBackground.cpp

#include "pch.h"

#include "SolidBackground.h"


SolidBackground::SolidBackground(D2D1::ColorF col):   color(col) { }


void SolidBackground::Render(

       Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {

context->Clear(color); // Clear the screen to the color

}
```

This class takes a color parameter in its constructor, saves it to a member variable, and uses this to clear the screen in its render method. To create an instance of our solid background, we need to add it to the GraphRenderer class. Open GraphRenderer.h and add an #include for the SolidBackground.h file (I have highlighted the lines which have been added or changed in blue).

```
// GraphRenderer.h

#pragma once

#include "DirectXBase.h"

//

// Additional headers for graph objects here

//

#include "SolidBackground.h"
```

Declare a member variable at the bottom of the GraphRenderer.h file.

```
private:

       // Global pan value for moving the chart with the mouse

       Windows::Foundation::Point m_pan;

       SolidBackground* m_solidBackground;

};
```

Open the GraphRenderer.cpp file and use “new” to create the instance of **m_solidBackground** in the GraphRenderer class constructor.

```
GraphRenderer::GraphRenderer() {

m_solidBackground = new SolidBackground(D2D1::ColorF::Bisque);

}
```

And finally, the GraphRenderer’s render method must be changed to call the new m_solidBackground’s render method. The call to the m_d2dContext’s Clear method is no longer needed and can be removed.

```
void GraphRenderer::Render(){

m_d2dContext->BeginDraw();

// Clear to some color other than blank

// m_d2dContext->Clear(D2D1::ColorF(ColorF::CornflowerBlue));

// Pan the chart

Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y);

m_d2dContext->SetTransform(panMatrix*m_orientationTransform2D);

//

// Draw objects here

//

m_solidBackground->Render(m_d2dContext);

// Ignore D2DERR_RECREATE_TARGET error

HRESULT hr = m_d2dContext->EndDraw();

if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

}
```

You can remove the call to **m_d2dContext::Clear()** in the render method, since it is no longer needed. Now is a good time to compile and run your application.

DirectX Colors

Colors can be specified using the **D2D1::ColorF** enumeration, which defines a list of around 140 predefined colors.

```
D2D_COLOR_F copyOfPredefinedColor = D2D1::ColorF(D2D1::ColorF::AliceBlue);

// Alternative syntax using the derived helper class would be:

D2D1::ColorF copyOfPredefinedColor2 = D2D1::ColorF(D2D1::ColorF::AliceBlue);
```

Note: For the complete list of predefined colors available in the D2D1::ColorF enumeration, right-click on AliceBlue or another color identifier and select Go To Definition from the context menu. This will open the Direct2DHelper.h file where the list of predefined colors is defined.

Note: The D2D1::ColorF class inherits from the D2D_COLOR_F class. It is the same but it defines some useful functions and an enumeration of predefined colors.

You can also create your own colors by specifying the amount of red, green, and blue the color has as floating point values:

```
D2D1::ColorF brightMagenta = D2D1::ColorF(1.0f, 0.0f, 1.0f);
```

The constructor for the ColorF class takes three parameters with an optional fourth (which defaults to 1.0f and represents the opacity or alpha channel). The first three arguments are the amount of red, green, and blue in the color. Here I have defined 100% red, 0% green, and 100% blue. This combines to create a bright magenta color. In this color model, the range for the components is from 0.0f to 1.0f, where 0.0f means none at all and 1.0f means full saturation or 100%.

You will often see colors initialized with something like the following:

`D2D1::ColorF myColor = D2D1::ColorF(D2D1::ColorF::PredefinedColor);`

The PredefinedColor is one of the colors from the ColorF enum defined in the D2D1Helper.h file.This is a call to the copy constructor of the ColorF class. The nested reference to the predefined color is the value to copy.

You can also define colors in a style similar to HTML colors using their hexadecimal representation.

```
D2D1::ColorF coffee = 0xCEAA7A;
```

Here, the value is an unsigned integer usually written as six hexadecimal digits, which represent three unsigned bytes ranging from 0 to 255 in decimal each. The lowest two digits represent the amount of blue (the 7A in the example), and they can range from 00 (none) to FF (255 or 100% saturation). The next two digits (the AA in the example) represent the amount of green in a similar fashion, and the highest two digits (the CE in the example) are the amount of red.

The example color model is called RGB for red, green, and blue. Sometimes there is an additional channel called the alpha channel, which is usually used to represent the opacity of the color. An alpha value of 0% means completely transparent, and 100% means completely opaque. The RGB color system with the additional alpha channel is called the ARGB color system, because the topmost bits of a 32-bit unsigned integer are used to store the alpha channel.

Tip: The RGB color model on little-endian systems (like x86 and ARM) results in the byte order for the color of a pixel actually being BGR, the reverse, when stored in memory. The blue byte is the lowest in memory and the red byte is the highest. When using the ARGB model, the byte order is BGRA.

Gradient Background

The solid background introduced clearing the screen; the next background will introduce Direct2D's Gradient Brush. Almost everything that we draw in Direct2D we do so using a brush. There are several different types of brush. We saw previously the use of a solid color brush to render text. Gradient backgrounds can be created by coloring the whole render target with a linear gradient brush prior to rendering the data. To create a GradientBackground class, add two files to the project, **`GradientBackground.h`** and **`GradientBackground.cpp`**.

```
// GradientBackground.h

#pragma once

#include "DirectXBase.h"

// Gradient background

class GradientBackground {

private:

	D2D1_COLOR_F *colors;	// The colors in the gradient

	float *stops;// Positions of the colors

	int count;	// The number of different colors used

	D2D1_RECT_F m_ScreenRectangle;	// The size of the rectangle we're filling


	// The linear gradient brush performs the painting

	Microsoft::WRL::ComPtr<ID2D1LinearGradientBrush> m_linearGradientBrush;


public:

	// Creates a new gradient background

	GradientBackground(D2D1_COLOR_F colors[], float stops[], int count);

	// Release dynamic memory
```

```
        ~GradientBackground();
        void CreateWindowSizeDependentResources
                (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);
        void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);
};
```

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

GradientBackground::GradientBackground(D2D1_COLOR_F colors[],
        float stops[], int count) {
// The constructor just saves the colors and stops
this->count = count;
this->colors = new D2D1_COLOR_F[count];
this->stops = new float[count];

for(int i = 0; i < count; i++) {
        this->colors[i] = D2D1_COLOR_F(colors[i]);
        this->stops[i] = stops[i];
        }
}
void GradientBackground::CreateWindowSizeDependentResources
        (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
{
// Create a gradient stops array from the colors and stops
D2D1_GRADIENT_STOP *gradientStops = new D2D1_GRADIENT_STOP[count];
for(int i = 0; i < count; i++) {
        gradientStops[i].color = colors[i];
```

```
        gradientStops[i].position = stops[i];
        }
// Create a Stop Collection from this using the
// context's create method:
ID2D1GradientStopCollection *gradientStopsCollection;
DX::ThrowIfFailed(
        context->CreateGradientStopCollection (
                gradientStops,      // Stops
                count,              // How many?
                &gradientStopsCollection   // Output object
                ));
// Create a linear gradient brush from this:
DX::ThrowIfFailed(
        context->CreateLinearGradientBrush(
        D2D1::LinearGradientBrushProperties (
                D2D1::Point2F(0, 0),// Start point of gradient
                D2D1::Point2F(      // Finish point of gradient
                        context->GetSize().width,
                        context->GetSize().height
                        )),
        gradientStopsCollection,
        &m_linearGradientBrush));


// Also save the rectangle we're filling
m_ScreenRectangle = D2D1::RectF(0, 0, context->GetSize().width,
        context->GetSize().height);
}


void GradientBackground::Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// The fill the whole screen with the gradient
```

```
context->FillRectangle(&m_ScreenRectangle, m_linearGradientBrush.Get());

}



GradientBackground::~GradientBackground() {

delete[] colors;

delete[] stops;

}
```

The constructor of this class does nothing more than save the values passed as parameters to a member variable, such that they can be used in the resources allocation methods to create a brush. The most important method is the creation of window size dependent resources. We want our gradient to fill the whole render target so we should place it in the window size dependent resources method. In the code sample, you will see two points specified in the method call to CreateLinearGradientBrush.

```
            D2D1::Point2F(0, 0),    // Start point of gradient

            D2D1::Point2F(          // Finish point of gradient

                  context->GetSize().width,

                  context->GetSize().height

                  )),
```

When we create a gradient brush, we first create a collection of the stops, positions at which the colors are blended, and from this we create the gradient brush itself. I have also saved the render target size to a member variable, so that it isn't calculated in the Render method.

I have used the screen coordinates (0, 0) and (width, height) such that the gradient stretches from the top left corner to the lower right. If you want a straight vertical gradient blending from the top of the screen to the bottom, you could use 0 instead of the screen's width as the first parameter of the second point. Likewise, if you want a gradient that stretches horizontally, you could replace the **`context->GetSize().height`** with 0.

Creating a linear gradient brush requires specifying a list of colors and a list of positions where the colors change. If the colors were red, green, and blue, and the stops were 0.0f, 0.2f, and 1.0f, the gradient's colors would be blended like the following:

At 0% across the gradient it is red, and at 20% across it is green, and at 100% it is blue. You can add as many stops and colors as needed. The stops can be higher than 100% (1.0f). This means the gradient's blend extends further than the visible area. If the stops do not begin at 0.0f, the first color will be assumed for the start of the gradient. Likewise, if the stops do not end with 1.0f, the remainder of the gradient brush will use the last color.

To make an instance and render the gradient background we follow a similar pattern to that of the solid background. However, the solid background did not need any resources, whereas the gtradient background creates a brush on the device, so we need to call the **`CreateWindowSizeDependentResources`** method to have the gradient background create this brush.

Note: We are examining the linear gradient, but there are also radial gradient brushes available in Direct2D. These radiate the gradient outwards from a central point in concentric circles.

Replace the reference to the SolidBackground.h header with a reference to the new GradientBackground.h header in the GraphRenderer.h file.

```
//
// Additional headers for graph objects here
//
#include "GradientBackground.h"
```

Replace the declaration of the SolidBackground member variable with a declaration of the new GradientBackground.

```
private:
	// Global pan value for moving the chart with the mouse
	Windows::Foundation::Point m_pan;


	GradientBackground* m_gradientBackground;
};
```

Replace the call to the SolidBackground constructor with a call to the new GradientBackground constructor in the GraphRenderer's constructor.

```
GraphRenderer::GraphRenderer() {
D2D1_COLOR_F colors[] = {
	D2D1::ColorF(ColorF::PaleGoldenrod),
	D2D1::ColorF(ColorF::PaleTurquoise),
	D2D1::ColorF(0.7f, 0.7f, 1.0f, 1.0f)
	};
```

```
float stops[] = {
        0.0f,
        0.5f,
        1.0f
        };

m_gradientBackground = new GradientBackground(colors, stops, 3);
}
```

Note: Standard colors, such as ColorF::Red, are declared as integer RGBA values in an enum. Our GradientBackground class expects an array of D2D1_COLOR_F structures, each of which represents a color as four distinct floating point values. This is the reason for wrapping the standard color inside a call to D2D1::ColorF().

Call the method to create the background's window size dependent resources in the GraphRenderer's **`CreateWindowSizeDependent`** resources method.

```
void GraphRenderer::CreateWindowSizeDependentResources() {
DirectXBase::CreateWindowSizeDependentResources();
m_gradientBackground->CreateWindowSizeDependentResources(m_d2dContext);
}
```

And finally, replace the call to render the SolidBackground with the call to render our new GradientBackground.

```
void GraphRenderer::Render() {
m_d2dContext->BeginDraw();

// Reset the transform matrix so our gradient does not pan
m_d2dContext->SetTransform(m_orientationTransform2D);
m_gradientBackground->Render(m_d2dContext);
// Pan the chart
Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y);
```

```
m_d2dContext->SetTransform(panMatrix*m_orientationTransform2D);

//

// Draw objects here

//

// Ignore D2DERR_RECREATE_TARGET error

HRESULT hr = m_d2dContext->EndDraw();

if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

}
```

I have assumed that the gradient background is not to be affected by the panning (translation) of the chart. This is only meant for panning the data. For this reason, I have added the call to render just after resetting the transformation matrix.

The number of gradients in the collection can be very large and generated rather than stored, or hard programmed into the code. The following creation of a gradient background produces a random pastel rainbow, and this could replace the code we placed into the GraphRenderer's constructor:

```
const int count = 500;

D2D1_COLOR_F *cols = new D2D1_COLOR_F[count];

float* stops = new float[count];


for(int i = 0; i < count; i++) {

      cols[i] = D2D1::ColorF(

            0.75f+(float)(rand()%192)/192.0f, // Random pastels

            0.75f+(float)(rand()%192)/192.0f,

            0.75f+(float)(rand()%192)/192.0f);

      stops[i] = (float)i / (count - 1);

      }


m_gradientBackground = new GradientBackground(cols,  stops, count);
```

The example code produces a rather pleasing gradient which should look something like Figure 13.

Figure 13: Rainbow Gradient

Gradient backgrounds are excellent for charting, especially on devices with limited resources, because they are more appealing to look at than the standard solid background. They are faster to render and initialize than a bitmap background, and they do not take up any disk space or bloat the application by storing an image.

Bitmap Backgrounds

Next we will examine loading and displaying an image by creating a bitmap background. You can use the WIC (Windows Imaging Component) to load a bitmap image (or several other standard image formats) and display it as a background. Bitmap backgrounds provide the most flexibility, but are more costly in terms of rendering performance.

Note: Thanks to the flexibility of the WIC decoders, this class will be able to load many standard image file formats. Windows 8 ships with decoders for JPEG, TIFF, PNG, BMP, and others. With no change to the code, our charts should be able to load any of these image formats.

The first thing to do is add an image file to your project. Right-click the project's name in the Solution Explorer and click **Add Existing item...** as per Figure 14.

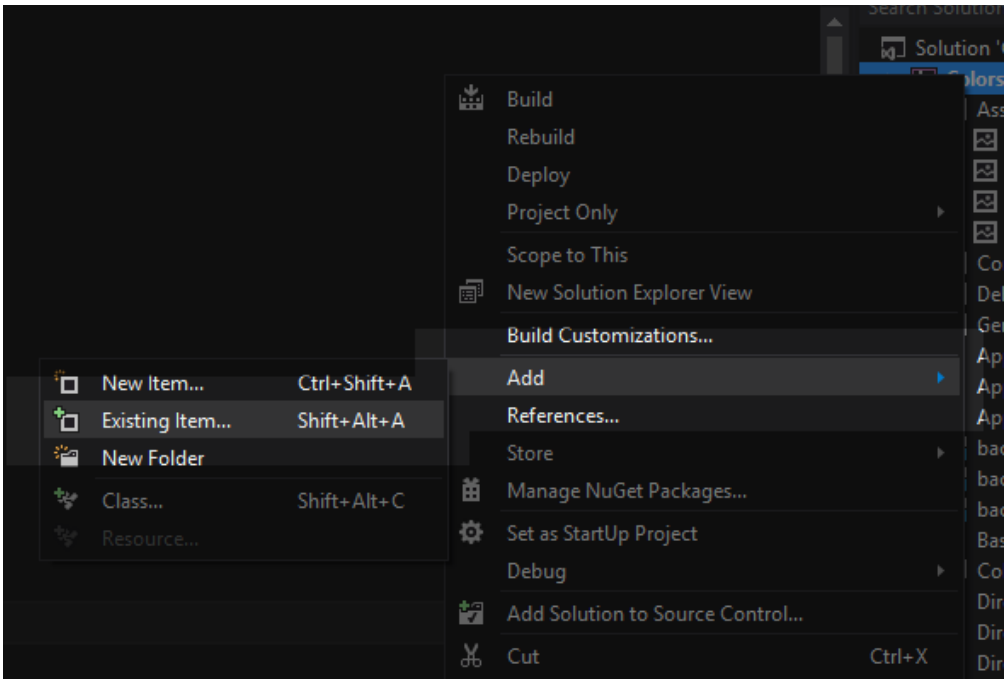


Figure 14: Adding an Existing Item

Locate the image file you wish to use in the **Add Existing Item** box that appears. I will use an image file called background5.jpg in this example. Once you have selected your file, click **Add** as per Figure 15.

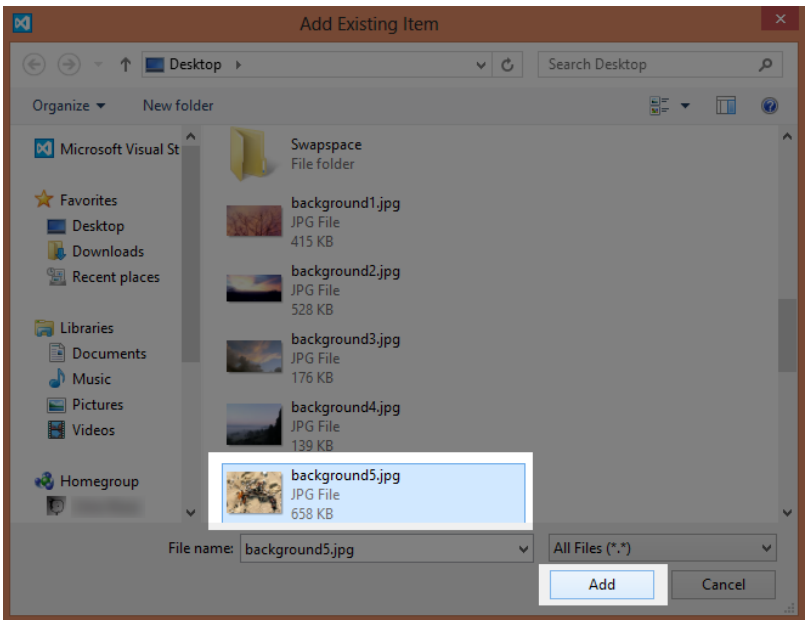


Figure 15: Adding the Image

Now that we have added a bitmap to our project, we can create the BitmapBackground class by adding the BitmapBackground.h and BitmapBackground.cpp files.

```
// BitmapBackground.h

#pragma once

#include "DirectXBase.h"


// Defines a background consisting of a bitmap image

class BitmapBackground {

private:

	ID2D1Bitmap * m_bmp; // The image to draw

	D2D1_RECT_F m_screenRectangle; // Destination rectangle


public:

	// Constructor for bitmap backgrounds

	BitmapBackground();


	// Release dynamic memory

	~BitmapBackground();
```

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

Tip: The ID2D1Bitmap is a device dependent resource. Many WinRT devices do not have dedicated GPU RAM, so the bitmap will be stored in the limited system memory. To reduce strain on system resources, it is best to load only relatively small bitmaps or load few of them if you intend your application to function smoothly on these devices.

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

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

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

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

// Decode a file, make sure you've added the file to the project first:
```

```
DX::ThrowIfFailed(wicFactory->CreateDecoderFromFilename(filename,
       nullptr, GENERIC_READ, WICDecodeMetadataCacheOnDemand, &pDecoder));

// Read a frame from the file (png, jpg, bmp etc. images only have one frame so
// the index is always 0):
IWICBitmapFrameDecode *pFrame = nullptr;
DX::ThrowIfFailed(pDecoder->GetFrame(0, &pFrame));

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

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

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

void BitmapBackground::CreateWindowSizeDependentResources(
       Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
```

```
// Save a rectangle the same size as the area to draw the background

m_screenRectangle = D2D1::RectF(0, 0, context->GetSize().width, context-
>GetSize().height);

}



void BitmapBackground::Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {

context->DrawBitmap(m_bmp, &m_screenRectangle);

}
```

The constructor is empty and the destructor just releases the bitmap. Most of the code for this background revolves around loading and decoding the image with the WIC factory and related components. This is done in the **`BitmapBackground::CreateDeviceDependentResources` method**. We first create a decoder using the wicFactory's **`CreateDecoderFromFile`** method. Then we use that to read a frame from the file. There will only be one frame in a JPEG, PNG, or bitmap image, so the frame index we pass is 0 (in the call to pDecoder's **`GetFrame`** method). There is a chance that the format of the frame we just read was something other than the standard RGB pixels we want to use. Use a WIC Format Converter to convert the data, and finally, create an **`ID2D1Bitmap`** object from this. We can then render the resulting converted frame to the screen.

The constructor for this class is empty, and the creation of the bitmap is delegated to the **`CreateDeviceDependentResources`** method because we require the use of the WIC decoder, but the WIC decoder is not initialized when the constructor of the graph renderer is called.

Note: The order of the calls to the constructor and methods to create the resources in the GraphRenderer class is Constructor, CreateDeviceIndependentResources, CreateDeviceResources, then CreateWindowSizeDependentResources. This order is specified in the DirectXBase.cpp file. DirectXPage also plays a role in this sequencing by calling the constructor.

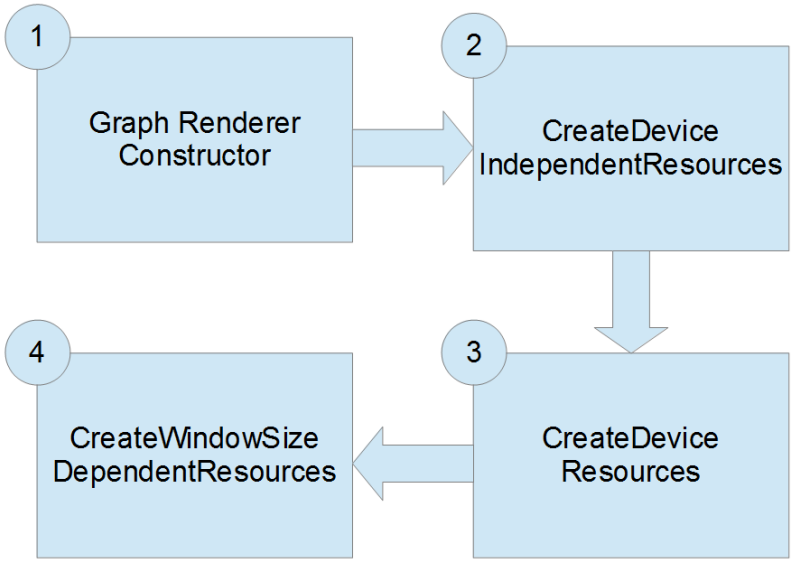


Figure 16: Sequence of Resource Creation Methods

To make a new instance of our class and render a bitmap background, replace the #include of the GradientBackground.h header with an #include for the new BitmapBackground.h header in the GraphRenderer.h file.

```
//

// Additional headers for graph objects here

//

#include "BitmapBackground.h"
```

Replace the definition of the GradientBackground member variable with a definition for the new BitmapBackground member variable called **m_bitmapBackground** at the bottom of the GraphRenderer.h file.

```
private:

	// Global pan value for moving the chart with the mouse

	Windows::Foundation::Point m_pan;



	BitmapBackground* m_bitmapBackground;

};
```

Replace the code in the GraphRenderer's constructor, which was used to construct the gradient background, with a call to the constructor for our new class.

```
GraphRenderer::GraphRenderer() {

m_bitmapBackground = new BitmapBackground();

}
```

Call the new bitmap background's **CreateDeviceResources** method in the Graph Renderer's method of the same name (remember to change the file name to the actual image you have used).

```
void GraphRenderer::CreateDeviceResources() {

DirectXBase::CreateDeviceResources();



// Load the bitmap for our background

m_bitmapBackground->CreateDeviceDependentResources(
```

```
        m_d2dContext,
        m_wicFactory.Get(),
        L"Background5.jpg");
}
```

Replace the call to the **m_gradientBackground->CreateWindowSizeDependentResources** method in the GraphRenderer's **CreateWindowSizeDependentResources** method with a call to the new class's **CreateWindowSizeDependentResources**.

```
void GraphRenderer::CreateWindowSizeDependentResources() {
DirectXBase::CreateWindowSizeDependentResources();
m_bitmapBackground->CreateWindowSizeDependentResources(m_d2dContext);
}
```

And finally, replace the call to the GradientBackground's render method with a call to the new BitmapBackground's render method in the graph renderer's render method before the panning matrix is applied.

```
void GraphRenderer::Render() {
m_d2dContext->BeginDraw();



// Reset the transform matrix so our background does not pan
m_d2dContext->SetTransform(m_orientationTransform2D);
m_bitmapBackground->Render(m_d2dContext);


// Pan the chart
Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y);
m_d2dContext->SetTransform(panMatrix*m_orientationTransform2D);


//
// Draw objects here
//
```

Note: The downside to using a bitmap background is that it increases the footprint of the application. An image takes space on the hard drive, but depending on the format this may be negligible. However, once the image is decoded and loaded into our application as an ID2D1Bitmap, it will no longer use the compression algorithm of the file format. For instance, a 2247 × 1345 pixel JPEG consumes less than 1 MB of disk space, but when loaded into the application it will add almost 9MBs to the memory usage. This 9 MB is due to there being 2247 × 1345 pixels in the image, each stored as red, green, and blue bytes. So the whole image is 2247 × 1345 × 3 bytes or about 8.6 MB.

Chapter 5: 2-D Data Plots

We will now move on to rendering some shapes in Direct2D by examining how to plot some data. This will be a small introduction to rendering vector graphics in Direct2D. In the interest of keeping the sample code simple, I will use arrays of randomly generated values for the data that will be plotted. When used in a real application, this data will probably come from a database or some other real data source.

Graph Variable Base Class

We will create scatter plots and line charts in the following sections. Both the scatter plot and the line chart share many characteristics, so we will implement a small class hierarchy. Both the scatter plot and the line chart will inherit from the following base class, which is called the GraphVariable class. Add two files to your project to define this base class, GraphVariable.h and GraphVariable.cpp.

```
// GraphVariable.h
#pragma once

#include "DirectXBase.h"

// This class represents a generic plottable variable
// It is the base class for the ScatterPlot and the LineChart
// classes.
class GraphVariable abstract
{
protected:
	D2D1_POINT_2F* m_points; // These are the x and y values of each node
	int m_nodeCount;  // This is a record of the total number of nodes

	// Record of smallest point
	float m_minX, m_minY; // Used to auto pan
public:
	// Getters for min values
	float GetMinX() { return m_minX; }
	float GetMinY() { return m_minY; }

	GraphVariable(float* x, float* y, int count);

	virtual void CreateDeviceDependentResources
(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) = 0;
	virtual void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
 = 0;

};
```

The class consists of the nodes as an array of D2D1_POINT_2F structures, as well as a count and a record of the smallest x and y values in the node collection. The reason we are taking note of the smallest x and y values is to automatically pan the data later on, so that some data is visible in the initial view of our charts.

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

GraphVariable::GraphVariable(float* x, float* y, int count)
{
this->m_nodeCount = count;

// Assume the minimum is the first value
m_minX = x[0]; m_minY = y[0];

// Create an array of points from the *x and *y.
// We can't assume that the *x and *y are permanent so
// allocate seperate heap space for a copy of the data
// as D2D1::Point2F's:
m_points = new D2D1_POINT_2F[count];
for(int i = 0; i < count; i++)
      {
      m_points[i] = D2D1::Point2F(x[i], y[i]);

      // Check if the point is lower than the current minimum
      if(x[i] < m_minX) m_minX = x[i];
      if(y[i] < m_minY) m_minY = y[i];
      }

}
```

The only method defined for this base class is a constructor that copies the two float arrays (*x and *y) to the member variable m_points. It also finds and records the smallest values in both the *x and the *y arrays, to be used for automatically panning later.

Scatter Plot

The scatter plot is one of the most common and useful depictions of 2-D data. It is usually represented on a chart as a collection of nodes, each drawn as a primitive shape such as a circle, square, or a triangle. Each node represents some point in 2-D space and each has various properties including its position, color, and size. Scatter plots are usually employed to represent data with two dimensions where each of the dimensions is parametric (such as weight and height). The values are not ordered by the x-axis and they do not show continuation in the same way that a line chart does. Two variables are needed for a scatter plot: the values in one variable determine how far left or right a node is, while the values in the other variable determine how far up or down a node is. Scatter plots are excellent for displaying data that may be correlated since correlated data appears to collect around a straight line or as a spray of nodes.

If you do not need to maintain a steady 60 frames per second as dictated by the V-sync, you can change the SwapChain's sync interval to 2, 3, or 4. By default, the swap chain will try to present after the V-sync at 60 fps. Change the value of the first parameter of the call to the following:

```
HRESULT hr = m_swapChain->Present1(1, 0, &parameters);
```

Changing the value of the first parameter of the call in the DirectXBase.cpp file to 2, 3, or 4 will cause the swap chain to sleep the application for up to 4 V-syncs, instead of the usual 1. This will result in far better power consumption at the cost of a smooth frame rate. The input 1 means 60 fps, 2 means 30 fps, 3 means 20 fps, and 4 means 15 fps. The frames per second required to adequately graph data are not usually 60 fps. 30 fps or 20 fps should be fine for most applications. Even 15 fps will look fairly smooth in a charting application.

The following scatter plot inherits from the previously defined GraphVariable base class. Add two files to your project, ScatterPlot.h and ScatterPlot.cpp.

```
// ScatterPlot.h
#pragma once

#include "DirectXBase.h"
#include "GraphVariable.h"

// Two different example shapes
enum NodeShape { Circle, Square };

// This class represents data to be drawn as a scatter plot
class ScatterPlot: public GraphVariable {
      float m_HalfNodeSize;// Half size of the nodes
      D2D1::ColorF m_NodeColor;    // The color of the nodes
      ID2D1SolidColorBrush* m_brush;     // Solid brush for painting nodes
      NodeShape m_NodeShape; // The shape of the nodes

public:
      // Public constructor
      ScatterPlot(float* x, float* y, float nodeSize,
            D2D1::ColorF nodeColor, NodeShape nodeShape, int count);

      virtual void
CreateDeviceDependentResources(Microsoft::WRL::ComPtr<ID2D1DeviceContext>
context) override;
      virtual void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
override;
};
```

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

using namespace D2D1;
using namespace DirectX;

ScatterPlot::ScatterPlot(float* x, float* y, float nodeSize,
      D2D1::ColorF nodeColor, NodeShape nodeShape, int count):
            m_NodeColor(0), GraphVariable(x, y, count) {
// Save half the node size. The nodes are drawn with
// the point they're representing at the middle of the shape.
this->m_HalfNodeSize = nodeSize / 2;

this->m_NodeShape = nodeShape;
```

```
this->m_NodeColor = nodeColor;
}

void ScatterPlot::Render(
      Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
switch(m_NodeShape) {
      // Draw as circle nodes
      case NodeShape::Circle:
            for(int i = 0; i < m_nodeCount; i++) {
                  context->FillEllipse(D2D1::Ellipse(m_points[i],
m_HalfNodeSize,
                        m_HalfNodeSize), m_brush);
                  }
            break;

      // Draw as square nodes
      case NodeShape::Square:
            for(int i = 0; i < m_nodeCount; i++) {
                  context->FillRectangle(D2D1::RectF(m_points[i].x -
                        m_HalfNodeSize,
                  m_points[i].y - m_HalfNodeSize, m_points[i].x +
m_HalfNodeSize,
                  m_points[i].y + m_HalfNodeSize), m_brush);
                  }
            break;

      // Additional shapes could follow

      default:
            break;
      }
}

void ScatterPlot::CreateDeviceDependentResources
      (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// Create a brush of the specified color for painting the nodes
DX::ThrowIfFailed(context->CreateSolidColorBrush(ColorF(m_NodeColor),
      &m_brush));
}
```

The constructor saves the color and node shape settings to member variables. The Render method of this class shows how to render two basic shapes, the ellipse and the rectangle. The shapes are drawn with the context's FillXXX methods, where the XXX is some primitive shape. These methods require a brush (created in the **`CreateDeviceDependentResources`** method), as well as the shape to draw.

The FillEllipse method takes an ellipse as its first parameter. This could be generated previously or calculated on the fly. Ellipses have a position and x and y radius. For instance, to create an ellipse whose center is at pixel (100, 150), is 60 DIPs wide, and twice as tall, you could use D2D1::Ellipse(100, 150, 60, 120). I have created circles by specifying the x and y radius as the same value.

The FillRectangle method takes a Rectangle as its first parameter. This requires specifying the x-coordinate and y-coordinate of the upper left corner of the rectangle, as well as the width and height.

The ID2D1DeviceContext interface has two versions of each of the primitive drawing methods; one fills shapes (FillEllipse, FillRectangle, etc.) and the other renders only the outline of an empty shape (the context's DrawEllipse or DrawRectangle methods).

We will now add a scatter plot to our GraphRenderer, but there are some big problems with our charting application that will become apparent as a result and will lead nicely to the next section.

Add the #include to the top of the GraphRenderer.h file (I have also included a gradient background).

```
//

// Additional headers for graph objects here

//

#include "GradientBackground.h"

#include "ScatterPlot.h"
```

Add a **GraphVariable** pointer member to the graph renderer class. This will be used as the **ScatterPlot** in this chapter, but it will also be used as the **LineChart** later.

```
private:

	// Global pan value for moving the chart with the mouse

	Windows::Foundation::Point m_pan;



	// Background

	GradientBackground *m_gradientBackground;



	// Plottable data

GraphVariable* m_graphVariable;
```

Call the constructors for the **GradientBackground** and the new **ScatterPlot** in the **GraphRenderer's** constructor in the GraphRenderer.cpp file.

```
GraphRenderer::GraphRenderer()
{
// Create the gradient background:
D2D1_COLOR_F colors[] = {
	D2D1::ColorF(ColorF::PaleGoldenrod),
	D2D1::ColorF(ColorF::PaleTurquoise),
	D2D1::ColorF(0.7f, 0.7f, 1.0f, 1.0f)
	};
float stops[] = { 0.0f, 0.5f, 1.0f };
```

```
m_gradientBackground = new GradientBackground(colors, stops, 3);

// Create the scatter plot:
const int count = 500; // Create 500 nodes
float* x = new float[count];
float* y = new float[count];

// Create random points to plot, these
// would usually be read from some data source:
for(int i = 0; i < count; i++) {
      x[i] = (float)(rand() % 2000);
      y[i] = (float)(rand() % 1000);
      }

m_graphVariable = new ScatterPlot(x, y, 10.0f,
      D2D1::ColorF::Chocolate,
      NodeShape::Circle, count);

delete[] x;
delete[] y;

}
```

Call the **ScatterPlot**'s **CreateDeviceDependentResources** method and the **GradientBackground**'s **CreateWindowSizeDependentResources** methods. In the following code I have also set the initial values for the **m_pan** member variable so the data will be visible when the application starts. Otherwise the data would be plotted off the screen.

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

// Call the create device resources for our graph variable
m_graphVariable->CreateDeviceDependentResources(m_d2dContext);
}

void GraphRenderer::CreateWindowSizeDependentResources() {
DirectXBase::CreateWindowSizeDependentResources();

// Create window size resources for gradient background
m_gradientBackground->CreateWindowSizeDependentResources(m_d2dContext);

// Set the initial pan value so the lowest node is visible in the corner
m_pan.X = -m_graphVariable->GetMinX();
m_pan.Y = -m_d2dContext->GetSize().height - m_graphVariable->GetMinY();

}
```

And finally, render the **GradientBackground** and the **ScatterPlot** in the **GraphRenderer::Render** method.

```
void GraphRenderer::Render()
{
m_d2dContext->BeginDraw();

// Reset the transform matrix so our background does not pan
m_d2dContext->SetTransform(m_orientationTransform2D);
// Render the background
m_gradientBackground->Render(m_d2dContext);

// Pan the chart
Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y +
m_d2dContext->GetSize().height);
m_d2dContext->SetTransform(panMatrix*m_orientationTransform2D);

//
// Draw objects here
//
// Render the graph variable
m_graphVariable->Render(m_d2dContext);

// Ignore D2DERR_RECREATE_TARGET error
HRESULT hr = m_d2dContext->EndDraw();
if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

}
```

Upon running the application, you should see something similar to Figure 17.

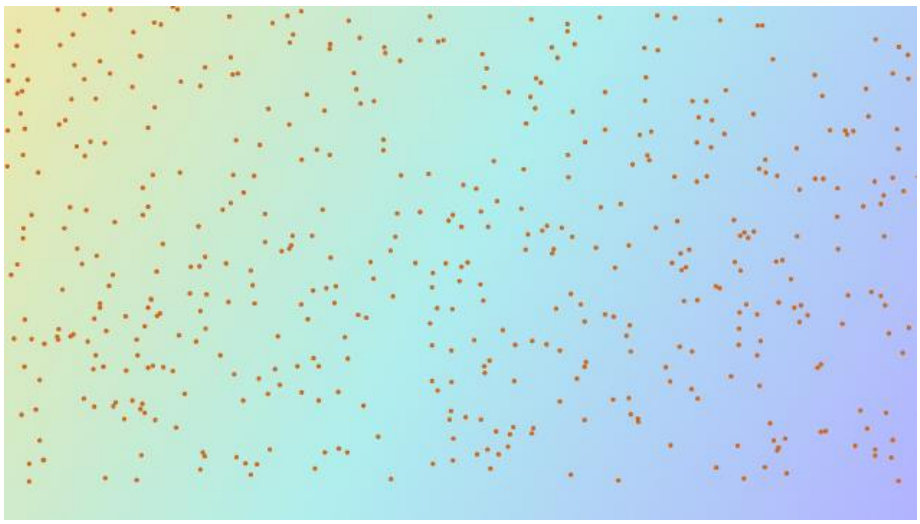


Figure 17: Scatter Plot Screenshot

The problem with our chart is that this looks like a scatter plot, but actually the y-axis is reversed. In computer graphics, the point (0, 0) refers to the pixel at the top left corner of the screen. Y values increase down the screen, while x values increase across the screen to the right. This behavior in the y-axis is the exact opposite of the way charts are normally rendered. Usually the point at the lower left corner is used to represent the origin (0, 0). If the data is changed from being random to increasing coordinates in both the x-axis and y-axis, you will see a diagonal line of nodes running from the top left toward the lower right.

```
// Create the scatter plot:
const int count = 500; // Create 500 nodes
float* x = new float[count];
float* y = new float[count];

// Create random points to plot, these
// would usually be read from some data source:
for(int i = 0; i < count; i++) {
      x[i] = i; //(float)(rand() % 2000);
      y[i] = i; //(float)(rand() % 1000);
      }

m_graphVariable = new ScatterPlot(x, y, 10.0f,
      D2D1::ColorF::Chocolate,

      NodeShape::Circle, count);
```

Upon running the application, you should see something like Figure 18.

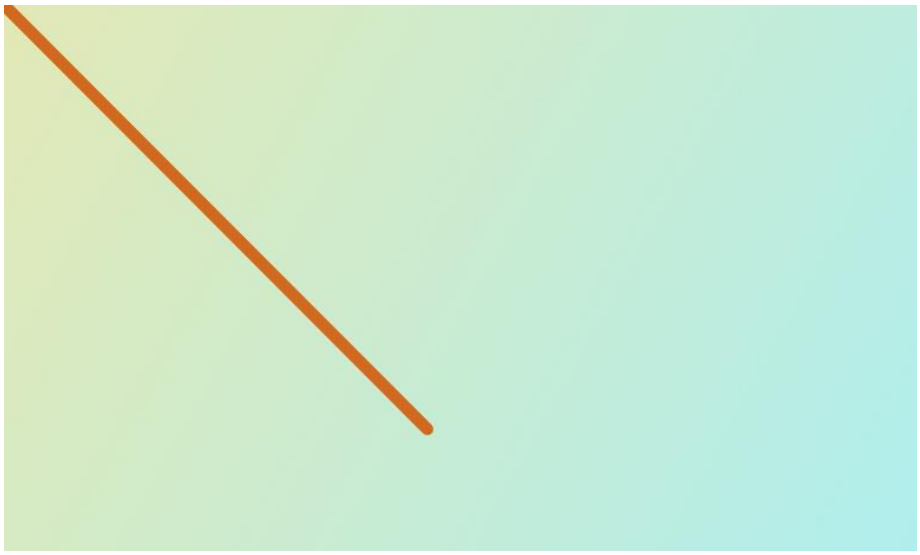


Figure 18: Scatter Plots Rendered in a Line

This is an image of 500 nodes all in a line, so close together that there are no gaps at all. The problem is that the whole y-axis needs to be flipped. We could do this fairly easily by multiplying each y value for the scatter plot by -1 and accounting for the screen's height. We could do this either when we create the scatter plot's data or when we render the nodes. However, there is a far better way. The GPU can easily flip the y-axis for us with just a few lines of code by applying a transform.

2-D Transformations

The movement, rotation, zooming, and many other aspects of DirectX scenes can be implemented using transformation matrices. Transformations are applied by creating matrices that can control almost all aspects of rendering: from an object's size and rotation settings, to its position and eventual pixels on the screen.

Note: The graphics card is specifically built to perform this type of operation. It applies matrix transforms to a collection of points or vertices far more efficiently than a CPU. It is an inherently parallel device, having many low powered execution units (perhaps hundreds of cores), as opposed to the multicore CPU, which has only a few high powered cores.

Matrices make graphics programming much easier. Matrix multiplication is cumulative, such that if you multiply a translation matrix by a rotation matrix, it will result in a single matrix that both translates and rotates. This is exactly the way the various transforms are applied to our objects when we render with DirectX.

Graphics programming is an excellent way to introduce matrix operations. It is very easy to see the effect of multiplying matrices when a scene is drawn using the results. The matrices used in 2-D and 3-D graphics are usually very small, consisting of perhaps 3 to 16 components. The steps to multiplying matrices can be found in many places on the Internet, and will not be examined in this book.

I will not describe matrix multiplication in detail, but one of the most important differences between matrix multiplication and regular arithmetic is that matrix multiplication is not commutative. If we have two matrices, one for translation and another for rotation, the order that they are multiplied is very important:

- Translation*Rotation ← Translate first then rotate
- Rotation*Translation ← Rotate first then translate

The top product translates first, and then rotates about the translated points. The lower product will rotate about the origin, and then translate the points in the direction of the rotation. Also, when two matrices are multiplied together, the result is a single matrix which does what both of the original two matrices did.

There are static helper functions in the **`Matrix3x2F`** class that can be used to create common transformations. The functions are **`Translation`**, **`Rotation`**, **`Scale`**, **`Skew`**, and **`Identity`**. To apply a collection of transformation matrices, create the matrices with the appropriate helper functions, multiply them together, and supply the resulting matrix as a parameter to the **`SetTransform`** method. **`SetTransform`** is a method belonging to the **`ID2D1DeviceContext`**, which is used to set the context's current transformation matrix. For example, to pan and translate, or scale and rotate, create three matrices for each transformation. Multiply the three matrices together in the call to the **`SetTranform`** method. The multiplication results in a single matrix that does all three jobs, panning, scaling, and rotating.

```
// Define each of the transformations
Matrix3x2F pan = Matrix3x2F::Translation(10.0f, 15.0f); // Pan 10 × 15 pixels
Matrix3x2F scale = Matrix3x2F::Scale(10.0f, 10.0f); // Scale 10 times the
size
Matrix3x2F rotate = Matrix3x2F::Rotation(25.0f);// Rotate 25 degrees

// Apply them all by multiplying together

m_d2dContext->SetTransform(pan * scale * rotate);
```

For the following discussion I will use a new Direct2D (XAML) template project for illustration. We will apply transformation to our charting application afterward. The code being altered is in the **`SimpleTextRenderer::Render`** method. The lines we are interested in have been highlighted in the following code:

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

	m_d2dContext->Clear(ColorF(BackgroundColors[m_backgroundColorIndex]));

	// Position the rendered text.
	Matrix3x2F translation = Matrix3x2F::Translation(
		m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
		m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y
		);

	// Note that the m_orientationTransform2D matrix is post-multiplied
here
	// in order to correctly orient the text to match the display
orientation.
	// This post-multiplication step is required for any draw calls that
are
	// made to the swap chain's target bitmap. For draw calls to other
targets,
	// this transform should not be applied.
	m_d2dContext->SetTransform(translation * m_orientationTransform2D);

	m_d2dContext->DrawTextLayout(
		Point2F(0.0f, 0.0f),
		m_textLayout.Get(),
		m_blackBrush.Get(),
		D2D1_DRAW_TEXT_OPTIONS_NO_SNAP
		);

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

	m_renderNeeded = false;

}
```

Translation Transform

The translation transform is applied to a matrix, and the matrix is used to transform a set of points. This is a movement transform that can be used to pan our chart. Open a new Direct2D (XAML) project, and open the SimpleTextRenderer.cpp file. Scroll down to the **Render** method, and at line 103 you will see a translation matrix being created. This particular matrix calculates the x and y values for the text, such that it begins centered and can be dragged around with the pointer. The DirectX XAML page captures the pointer movements, recording the x and y position in the **m_textPosition** member variable.

```
// Position the rendered text.

Matrix3x2F translation = Matrix3x2F::Translation(

      m_windowBounds.Width / 2.0f -

            m_textMetrics.widthIncludingTrailingWhitespace / 2.0f +
m_textPosition.X,

      m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y

      );
```

The matrix is comprised of two components, the amount to translate in the x or horizontal axis, and the amount to translate in the y or vertical axis.

```
Matrix3x2F translation = Matrix3x2F::Translation (

      amountToMoveInHorizontal,    // X-axis 0 is far left

      amountToMoveInVertical       // Y-axis 0 is top of screen

      );
```

To place the text at the top left corner of the screen, you could set both these values to **`0.0f`**. This would mean do not translate the axis at all.

```
// Position the rendered text.

Matrix3x2F translation = Matrix3x2F::Translation(

      0.0f, 0.0f

      );
```

This will produce something like Figure 19, and the text will no longer move around with the pointer.

Figure 19: No Translation

In this particular program we also see that the matrix defined previously is multiplied by a matrix called **m_orientationTransform2D** (in the context's call to **SetTransform)**. The orientation matrix is updated when the program is run on a WinRT device, such that if the user turns the screen, the text can correct itself and always be displayed upright. The values for the orientation matrix are set up in the DirectXBase class. Remember that two or more matrices can be multiplied together to produce a single matrix that contains all the transformations of the original matrices.

Rotation Transform

The rotation transform in Direct2D allows objects to rotate clockwise or counterclockwise around some defined point. If you define a rotation transform just after the translation matrix, and then multiply the translation and orientation matrices by this new matrix, you will see the text has been rotated by 45 degrees in a clockwise direction.

```
      // Position the rendered text.
      Matrix3x2F translation = Matrix3x2F::Translation(
            m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
            m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y

            );

// Rotate text about the middle

Matrix3x2F rotation = Matrix3x2F::Rotation (

      45.0f,      // Angle to rotate in degrees, clockwise is positive

      D2D1::Point2F (

      m_textMetrics.widthIncludingTrailingWhitespace / 2.0f, // X position of
origin

            m_textMetrics.height / 2.0f  // Y position of origin

            )
```

```
      );


// Apply the rotation, then the translation, and then the orientation matrix

m_d2dContext->SetTransform(rotation * translation *
m_orientationTransform2D);
```

Figure 20: Rotation

The text in Figure 20 has been rotated about its center point since this was the origin specified in the rotation matrix. The origin specified in the matrix is the point about which the rotation is to occur; it is the point that will remain static. If instead we specify the origin to rotate as the point (0, 0), the text will be rotated about its top left corner (the output of this will look like Figure 21).

```
// Position the rendered text.
Matrix3x2F translation = Matrix3x2F::Translation(
      m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
      m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y

      );



      // Rotate text about the middle

      Matrix3x2F rotation = Matrix3x2F::Rotation (

            45.0f,      // Angle to rotate in degrees

            D2D1::Point2F (
```

```
                0.0f, // X position of origin
                0.0f  // Y position of origin
                )
            );
      // Apply the rotation, then the translation, and then the orientation
matrix
      m_d2dContext->SetTransform(rotation * translation *
       m_orientationTransform2D);
```

Figure 21: Rotation

The other extremely important point to reiterate is that matrix multiplication is not commutative. If we apply the rotation after the translation by placing it as the second operand in the string of matrix multiplications in the call to **`SetTransform`**, we see that the effect is very different.

```
      m_d2dContext->SetTransform(translation * rotation *
m_orientationTransform2D);
```

Figure 22: Rotation

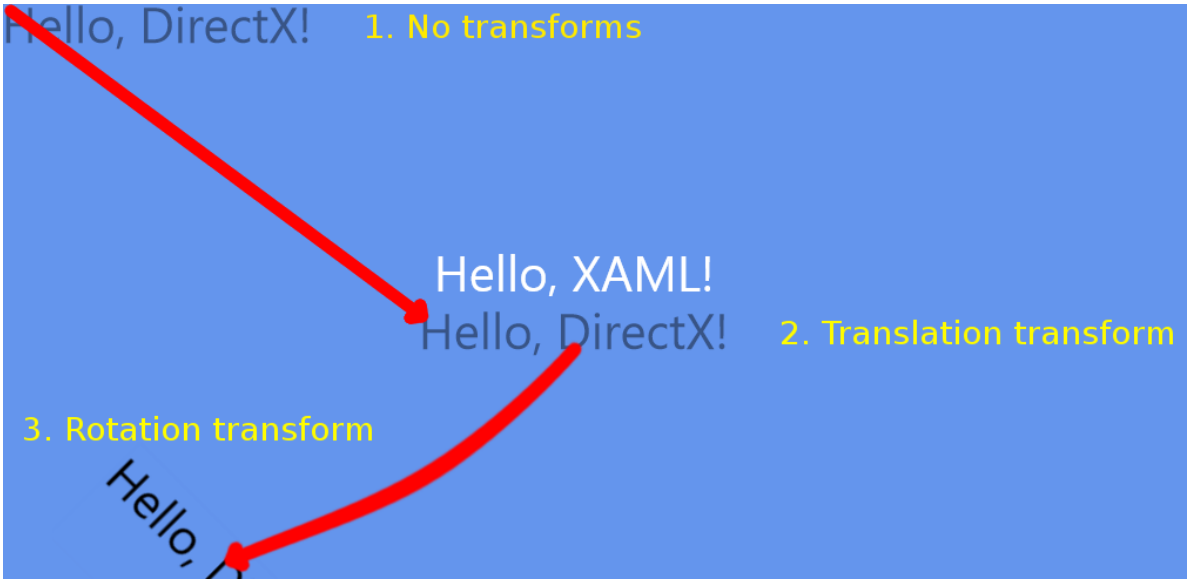


Figure 23: Order of Transforms

In Figure 22 the text is almost rotated off the screen altogether. The origin of the rotation is (0, 0), the top left corner of the screen (the second parameter in the call to create the matrix). Figure 23 is an illustration of the order of transformations that lead to the text appearing half rotated off the screen:

Scale Transform

Scaling applies a multiplier to one or both of the axes in order to shrink or enlarge the rendered shapes. The scale transform requires three parameters, an X multiplier, a Y multiplier and an origin about which to scale. The output from this code should look like Figure 24.

```
// Position the rendered text.
Matrix3x2F translation = Matrix3x2F::Translation(
      m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
      m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y
```

```
        );
// This will make the text 3 times wider and half as high!
Matrix3x2F scale = Matrix3x2F::Scale (
        3.0f, // Multiply x by 3
        0.5f, // Halve the y values
        D2D1::Point2F(    // Make the origin the middle of the text
                m_textMetrics.widthIncludingTrailingWhitespace / 2.0f,
                m_textMetrics.height / 2.0f)
        );
// Apply the scale, then the translation, and then the orientation
m_d2dContext->SetTransform(scale * translation * m_orientationTransform2D);
```

Figure 24: Scaling

The scale's origin was calculated such that the text grows and shrinks from its center point, the letter D. I have marked the code which calculates the origin in blue. Moving the text first, and then applying the scale produces very different results, similar to rotation and translation:

```
// Position the rendered text.
Matrix3x2F translation = Matrix3x2F::Translation(
        m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
        m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y
        );
```

```
      // This will make the text 3 times wider and half as high!
Matrix3x2F scale = Matrix3x2F::Scale (
      3.0f, // Multiply x by 3
      0.5f, // Halve the y values
      D2D1::Point2F(    // Make the origin the middle of the text
            m_textMetrics.widthIncludingTrailingWhitespace / 2.0f,
            m_textMetrics.height / 2.0f)
      );


      m_d2dContext->SetTransform(translation * scale *
m_orientationTransform2D);
```

The text is first translated to the center of the screen using the **translation** matrix. When the **scale** matrix is applied, the current position of the text is multiplied by 3.0f in the x-axis and halved (multiplied by 0.5f) in the y-axis. This leads to the text being off the screen altogether. I have depicted the area outside the right side of the screen as a dark blue color in Figure 25, so we can see where our text has gone.

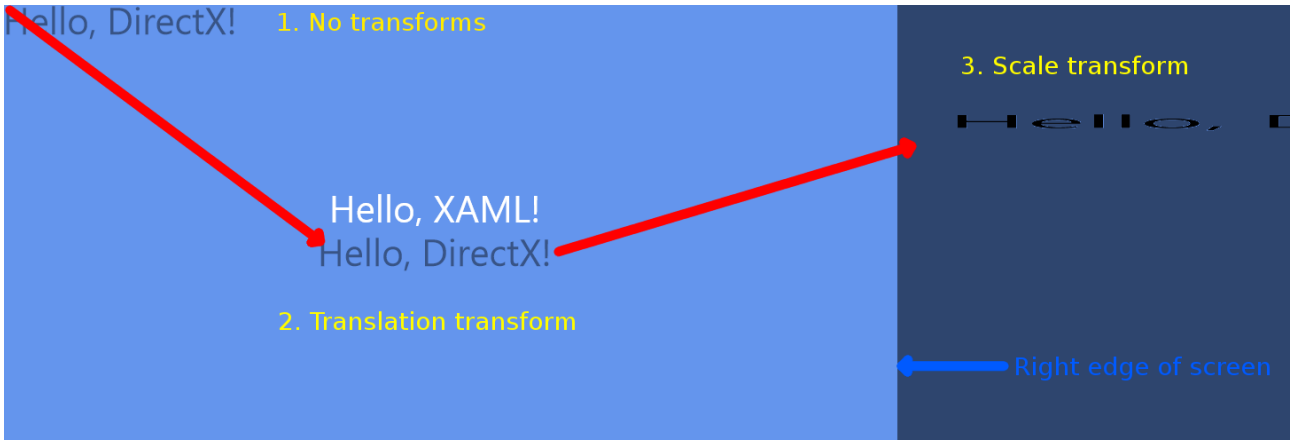


Figure 25: Order of Transforms 2

Like the rotation matrix, the scale matrix requires the specification of an origin point. The origin point is the point that will remain static throughout the growing and shrinking. In the previous example it was the center of the text, but it can be any point, even points well outside the bounds of the text. For instance, if we change the origin to the top right corner of the text, we can see that it no longer grows and shrinks from the center, but leftwards and downwards instead.

```
// Position the rendered text.
Matrix3x2F translation = Matrix3x2F::Translation(
      m_windowBounds.Width / 2.0f -
m_textMetrics.widthIncludingTrailingWhitespace / 2.0f + m_textPosition.X,
      m_windowBounds.Height / 2.0f - m_textMetrics.height / 2.0f +
m_textPosition.Y

      );

      // This will make the text 3 times wider and half as high!

      Matrix3x2F scale = Matrix3x2F::Scale (
```

```
            3.0f, // Multiply x by 3

            0.5f, // Halve the y values

            D2D1::Point2F(    // Make the origin the top right corner

                  m_textMetrics.widthIncludingTrailingWhitespace,

                  0.0f)

            );


      // Apply the scale, then the transform, and then the orientation

      m_d2dContext->SetTransform(scale * translation *
m_orientationTransform2D);
```

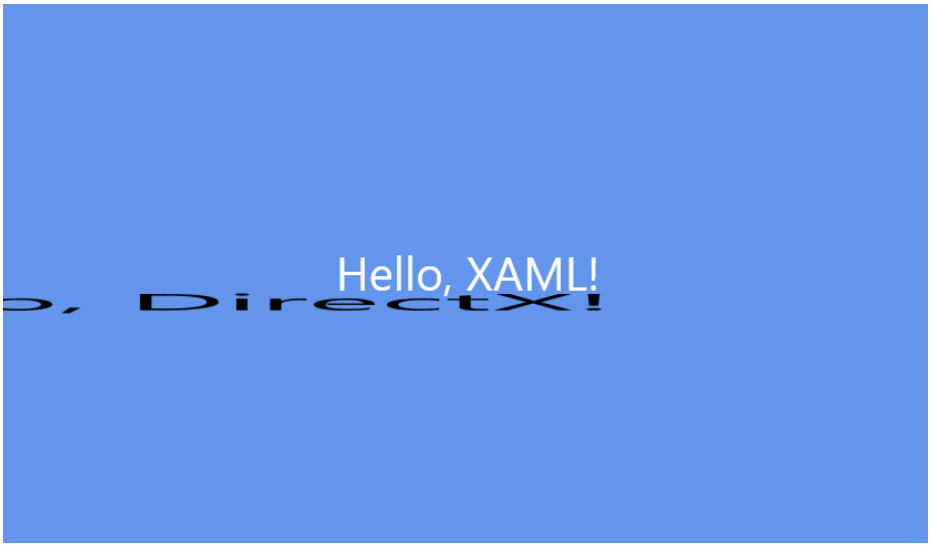


Figure 26: Scaling

The origin remains fixed in Figure 26. The text is squashed along the y-axis and stretched along the x-axis. If you apply the scale after the translation, then the translation will also be scaled.

Note: There are other transformations in Direct2D that can be used to produce a variety of effects. Matrix3x2F::Identity() is a matrix with the original or default values. It can be used to reset the transformations. The Skew matrix is also interesting; it can be used to produce some simple perspective transformations on 2-D data.

Translating the Scatter Plot

Now that we have looked at how to transform the objects we are drawing, we can fix our scatter plot. By default, the computer will assume the top left corner is the origin, and the y-axis increases downwards and to the right. We want our origin to be in the lower left corner of the screen, and we want the y-axis values to increase up the screen.

To fix our scatter plot, we can apply two matrices, a scale matrix and a translation. The scale matrix multiplies all the y values by -1.0f, thereby inverting the y-axis. In flipping the y-axis, we have flipped all the data upwards above the screen. The translation matrix can be used to add the screen's height, so we are once again looking at the data.

```
void GraphRenderer::Render()
{
m_d2dContext->BeginDraw();

// Reset the transform matrix so our background does not pan
m_d2dContext->SetTransform(m_orientationTransform2D);
// Render the background
m_gradientBackground->Render(m_d2dContext);

// The scale matrix inverts the y-axis
Matrix3x2F scale = Matrix3x2F::Scale(1.0f, -1.0f, D2D1::Point2F(0.0f, 0.0f));

// The pan matrix still pans but it also adds the height of the screen
Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y +
      m_d2dContext->GetSize().height);

// Apply the scale first
m_d2dContext->SetTransform(scale*panMatrix*m_orientationTransform2D);


//
// Draw objects here
//
// Render the graph variable
m_graphVariable->Render(m_d2dContext);

// Ignore D2DERR_RECREATE_TARGET error
HRESULT hr = m_d2dContext->EndDraw();
if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

}
```

Before we run the application, we also need to adjust the initial starting position for the **m_pan** member variable. This is specified in the **GraphRenderer:CreateWindowSizeDependentResources** method, the x value can stay the same but the y value must be changed since we flipped the axis.

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

// Create window size resources for gradient background
m_gradientBackground->CreateWindowSizeDependentResources(m_d2dContext);

// Set the initial pan value so the lowest node is visible in the corner
m_pan.X = -m_graphVariable->GetMinX();
m_pan.Y = m_graphVariable->GetMinY();

}
```

Upon running the application, you should now see that the origin is in the lower left corner and the y-axis increases upwards, exactly the same as a regular graph.

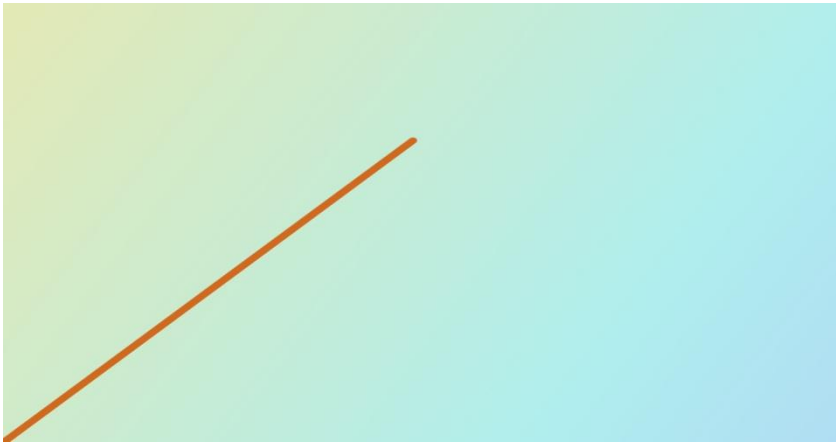


Figure 27: Scatter Plot with Flipped Y-Axis

Chapter 6: Infinite Lines and the Axes

In this section, we will introduce one method of rendering infinite lines. I will use the example of rendering the chart's axes that intersect at the origin. Scatter plots and line charts often have the concept of an origin which is the point (0, 0) in the chart's coordinates. Sometimes this point is very important, and we will display it on the chart as two intersecting lines: one representing the 0 point for the x-axis and another representing the 0 point for the y-axis.

Add two files to your project, Axes.h and Axes.cpp.

```
// Axes.h
#pragma once
#include "DirectXBase.h"
// This class represents a graph's axes as two perpendicular lines
class Axes {
      ID2D1SolidColorBrush* m_solidBrush; // Brush to draw with
      float m_lineThickness; // Thickness in pixels
      float m_opacity;       // Opacity, 0.0f is invisible 1.0f is solid
      D2D1::ColorF m_color;  // The color of the lines

public:
      // Public constructor
      Axes(D2D1::ColorF col, float thickness, float opacity);

      // Create the solid brush to draw with
      void CreateDeviceDependentResources
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);

      // The render method needs to know the panning and scaling
      void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context,
            float panX, float panY, float scaleX, float scaleY);
};
```

```
// Axes.cpp
#include "pch.h"
#include "Axes.h"
using namespace D2D1;
using namespace DirectX;
Axes::Axes(D2D1::ColorF col, float thickness = 3.0f, float opacity = 1.0f):
m_color(0)
{
// Save these settings to member variables so
// they can create the brush later:
this->m_color = col;
this->m_lineThickness = thickness;
this->m_opacity = opacity;
}

void Axes::CreateDeviceDependentResources(
Microsoft::WRL::ComPtr<ID2D1DeviceContext> context){
// Create the solid color brush
```

```
DX::ThrowIfFailed(context->CreateSolidColorBrush(
      m_color, D2D1::BrushProperties(m_opacity), &m_solidBrush));
}

void Axes::Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context, float
panX, float panY, float scaleX, float scaleY) {
// Draw infinite vertical line with 0.0f as the x-coordinate
context->DrawLine(
      D2D1::Point2F(
            0.0f,            // Horizontal axis
            (-context->GetSize().height - panY) / scaleY // Top of the screen
            ),
      D2D1::Point2F(
            0.0f,            // Horizontal axis
            (-panY) / scaleY  // Bottom of the screen
            ),
      m_solidBrush,
      m_lineThickness/scaleX);

// Draw infinite horizontal line with 0.0f as the y-coordinate
context->DrawLine(
      D2D1::Point2F(
            -(panX)/scaleX,   // Left side of screen
            0.0f                        // Vertical axis
            ),
      D2D1::Point2F(
            (context->GetSize().width - panX)/scaleX, // Right side of screen
            0.0f                        // Vertical axis
            ),
      m_solidBrush,
      m_lineThickness/scaleY);
}
```

The constructor saves some settings to member variables. The **`CreateDeviceDependentResources`** method creates a brush to paint the lines.

The axis lines are theoretically infinite in length. It does not matter how far left, right, up, or down the user pans around the graph's plane, and the ends of these lines should never be visible. To achieve this effect we draw two lines, one for the x-axis and the other for the y-axis. The horizontal line (which marks the 0 point for the y-axis) is drawn the same width as the screen, and the vertical line (which marks the 0 point for the x-axis) has the same length as the screen's height. In this way, regardless of how far the chart is panned, the axis lines will always be drawn across the whole screen if it is visible at all. This will appear as infinite when actually these lines are quite short.

The actual drawing of the lines is achieved through the use of Context's **`DrawLine`** method. This method takes two points, a brush and a line thickness. The line is drawn to connect the two points.

The thickness of the lines is static. I have assumed that even if the user zooms out thousands of units, the origin line should still be visible. Likewise, if the user zooms right into tiny points around the origin, it should not scale to the zoom and take up the entire screen. I have made our origin a standard thickness in pixels no matter the zoom factor by dividing the thickness by the current scale or zoom. In the code I have undone the panning and zooming manually to achieve this.

Note: When drawing lines with some thickness other than 1.0f, it is the center (lengthwise) of the line that will be at the specified coordinate. This is different from bitmaps whose top left corner is drawn at the specified coordinate. This means that if you draw a line from (0, 0) to (100, 0) with a thickness of 30, the top edge of the line will be drawn at (0, (-30/2)) and the lower edge will be drawn at (0, (30/2)).

I have also included a margin. This is the amount, in pixels, short of the screen's edge that the lines will be drawn. It can be used to produce a crosshair origin instead of infinite lines for the axes. To instantiate an object of our new **Axes** class, add the header to the GraphRenderer.h file. I have included a gradient background and scatter plot in the following code and I have highlighted the code that deals with the origin in blue.

```
//

// Additional headers for graph objects here

//

#include "GradientBackground.h"

#include "ScatterPlot.h"

#include "Axes.h"
```

Also add an Axes member variable to this file.

```
private:

	// Global pan value for moving the chart with the mouse

	Windows::Foundation::Point m_pan;

	// Background

	GradientBackground *m_gradientBackground;

	// Axes

	Axes* m_axes;

	// Data

	ScatterPlot* m_scatterPlot;
```

Create the chart objects in the GraphRenderer's constructor. These can be created in any order. I have created the Axes last.

```
GraphRenderer::GraphRenderer() {

// Create the gradient background:
```

```
D2D1_COLOR_F colors[] = {

       D2D1::ColorF(ColorF::PaleGoldenrod), D2D1::ColorF(ColorF::PaleTurquoise),

       D2D1::ColorF(0.7f, 0.7f, 1.0f, 1.0f) };



float stops[] = { 0.0f, 0.5f, 1.0f };

m_gradientBackground = new GradientBackground(colors, stops, 3);



// Create the scatter plot:

const int count = 25;



float* x = new float[count];

float* y = new float[count];



// Create random points to plot, these

// would usually be read from some data source:

for(int i = 0; i < count; i++) {

       x[i] = (float)((rand() % 2000) - 1000);

       y[i] = (float)((rand() % 1000) - 500);

       }



m_scatterPlot = new ScatterPlot(x, y, 10.0f, D2D1::ColorF::Chocolate, NodeShape::Circle,
count);



delete[] x;

delete[] y;



// Create the Axes

m_axes = new Axes(D2D1::ColorF::Black, 5.0f, 0.75f);

}
```

Call the Axes' **`CreateDeviceResources`** method so it can initialize its solid color brush.

```
void GraphRenderer::CreateDeviceResources() {

DirectXBase::CreateDeviceResources();


// Create the brush for the scatter plot:

m_scatterPlot->CreateDeviceDependentResources(m_d2dContext);


// Create the brush for the Axes

m_axes->CreateDeviceDependentResources(m_d2dContext);
}
```

And finally, call the origin's render method in the GraphRenderer's **`Render`** method.

```
void GraphRenderer::Render() {

m_d2dContext->BeginDraw();

// Clear to some color other than blank

m_d2dContext->Clear(D2D1::ColorF(ColorF::CornflowerBlue));

// Reset the transform matrix so our background does not pan

m_d2dContext->SetTransform(m_orientationTransform2D);

// Draw the background

m_gradientBackground->Render(m_d2dContext);

// The scale matrix inverts the y-axis

Matrix3x2F scale = Matrix3x2F::Scale(1.0f, -1.0f, D2D1::Point2F(0.0f, 0.0f));

// The pans added to the screen height so origin is at lower left

Matrix3x2F panMatrix = Matrix3x2F::Translation

       (m_pan.X, m_pan.Y + m_d2dContext->GetSize().height);

// Apply the scale and the pan

m_d2dContext->SetTransform(scale*panMatrix*m_orientationTransform2D);

// Draw the axes

m_axes->Render(m_d2dContext, m_pan.X, m_pan.Y, 1.0f, -1.0f);

//
```

```
// Draw objects here

//

m_scatterPlot->Render(m_d2dContext);

// Ignore D2DERR_RECREATE_TARGET error

HRESULT hr = m_d2dContext->EndDraw();

if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);

}
```

Upon running the application and panning a little to the right and upwards, you will see the origin and the scatter plot. This is the (0, 0) point in our chart's world coordinates as shown in Figure 28.

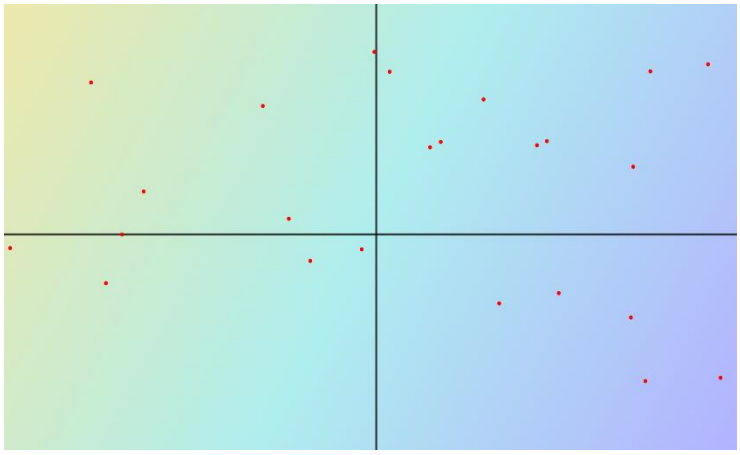


Figure 28: Axis Lines

Chapter 7: Displaying FPS (Frames per Second)

Frames per second (FPS) is the rate at which DirectX refreshes or renders a scene. In this section, we will examine how to display the FPS in the top left corner of our chart. This will provide an example of calculating the FPS, but also of rendering text. Text is very important in a charting application and can be used to render a title for the cart, node positions, axis labels, and many other things. Text is very slow to render, so aim to minimize the amount of text to less than 200 strings or so. The labels of the axis, chart title, node values, and many other things can all be rendered with text, but the FPS will very quickly drop if you render thousands of strings.

We will build on the chart from the last chapter, which displayed the axis lines. Add two member variable floats to the **GraphRenderer** class for recording the time in the **BasicTimer** when the **GraphRenderer::Update** method is run. Also, add an **IDWriteTextFormat** object which will hold the format of our FPS output, and a black brush which will be used to draw the text.

```
private:
	// Global pan value for moving the chart with the mouse
	Windows::Foundation::Point m_pan;


	// Member variables for displaying FPS
	float m_timeDelta;  // Time since last update call
	float m_timeTotal;  // Total time of application
	Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;
	Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;
```

Create the Text Format instance in the **GraphRenderer::CreateDeviceIndependentResources** resources method. The text format is used to specify the font, size, and several other text formatting options.

```
void GraphRenderer::CreateDeviceIndependentResources() {
DirectXBase::CreateDeviceIndependentResources();


DX::ThrowIfFailed(
	m_dwriteFactory->CreateTextFormat(
		L"Segoe UI",
		nullptr,
		DWRITE_FONT_WEIGHT_NORMAL,
```

```
            DWRITE_FONT_STYLE_NORMAL,

            DWRITE_FONT_STRETCH_NORMAL,

            42.0f,

            L"en-US",

            &m_textFormat

            )

      );
}
```

Create the brush for drawing the text in the **CreateDeviceDependentResources** method.

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

// Call the create device resources for our graph variable
m_graphVariable->CreateDeviceDependentResources(m_d2dContext);

// Create the brush for the origin
m_axes->CreateDeviceDependentResources(m_d2dContext);

// Create the solid brush for the text
DX::ThrowIfFailed(
      m_d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Black),&m_blackBrush));

}
```

Add an **#include <string>** to the GraphRenderer's code file. This gives us functions to append the floats to strings for displaying the frames per second.

```
// GraphRenderer.cpp

#include "pch.h"

#include <string>

#include "GraphRenderer.h"
```

Record the times (**m_timeTotal** and **m_timeDelta**) passed as parameters to the **Update** method in the GraphRenderer's code file. **m_timeTotal** is the total number of milliseconds that have elapsed since the program started, and **timeDelta** is the amount of time that has elapsed since the last call to **Update**.

```
void GraphRenderer::Update(float timeTotal, float timeDelta) {

	// Record the times for displaying:

	m_timeDelta = timeDelta;

	m_timeTotal = timeTotal;

}
```

In the **GraphRenderer::Render** method, create the string by appending the times to labels (Total Time and FPS). The inverse of m_timeDelta (**1.0f/m_timeDelta**) is the speed of the last update. I have rounded this value to an integer. It is a good idea to render our new FPS string after all of the graph objects are drawn so it is not obscured.

```
//
// Draw objects here
//
// Render the graph variable
m_graphVariable->Render(m_d2dContext);

// Reset the transform matrix so the time and FPS does not pan or zoom
m_d2dContext->SetTransform(m_orientationTransform2D);
// Set up the string to print:
std::wstring s = std::wstring(
	L"Total Time: ") + std::to_wstring(m_timeTotal) +
	std::wstring(L" FPS: ") + std::to_wstring(
		(int)(0.5f+1.0f/m_timeDelta));	// FPS rounded to nearest int
// Render the string in the top left corner
m_d2dContext->DrawText(s.c_str(), s.length(), m_textFormat.Get(),
	D2D1::RectF(0, 0, 600, 32), m_blackBrush.Get());


// Ignore D2DERR_RECREATE_TARGET error
```

Upon running the application, you will note that the timers are only updated when the graph is panned. This is a very good thing for saving power on WinRT devices, but for testing the performance of our chart rendering, we want to switch the application to real time. To switch to real time (updating repeatedly), open the DirectXPage.cpp file and comment out the “if” condition that causes the program to update based on the **m_renderNeeded** member variable. The **OnRendering** method of the DirectXPage is called when the **CompositionTarget::Rendering** event is fired. The **CompositionTarget** is the surface to which the XAML controls are rendered.

```
void DirectXPage::OnRendering(Object^ sender, Object^ args) {

	// if (m_renderNeeded)

	{

		m_timer->Update();

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

```
            m_renderer->Render();

            m_renderer->Present();

            m_renderNeeded = false;

        }

}
```

Now when you run the application it should update continuously. The FPS is a little difficult to read when it updates many times per second. We can slow it down and update the counter only every 16 frames by adding a static counter and "if" condition to the GraphRenderer's update.

```
void GraphRenderer::Update(float timeTotal, float timeDelta) {

static int fpsCounter = -1;// Start at -1 so frame 0 updates timers

fpsCounter++;

if((fpsCounter & 15) == 0) { // Update every 16 frames

        // Record the times for display in the render method:

        m_timeDelta = timeDelta;

        m_timeTotal = timeTotal;

        }

}
```

Tip: Use Boolean operations and bit shifting instead of integer division wherever you can. The CPU needs to run the update method very quickly. It is best to minimize division, modulus, square roots, trigonometry, and all other complex functions in the update and render methods. Some of these complex functions are hundreds of times slower to execute than simple Boolean instructions. The optimizing compiler is most likely smart enough to realize that "x%16" is the same as "x&15" but every compiler is different and it may be best not to trust the compiler when there is a simple optimization such as this.

Chapter 8: Line Charts

We have looked a little at rendering lines in our origin, but now we will examine rendering many lines, and we will use a line chart as an example. Line charts display data as a series of connected lines. The lines connect consecutive nodes or points, and depicts continuity or a chronological order to the data in a way which the scatter plot does not. Frequently, the x-axis is seen as time and progresses from early points on the left to later points on the right, as shown in Figure 29.

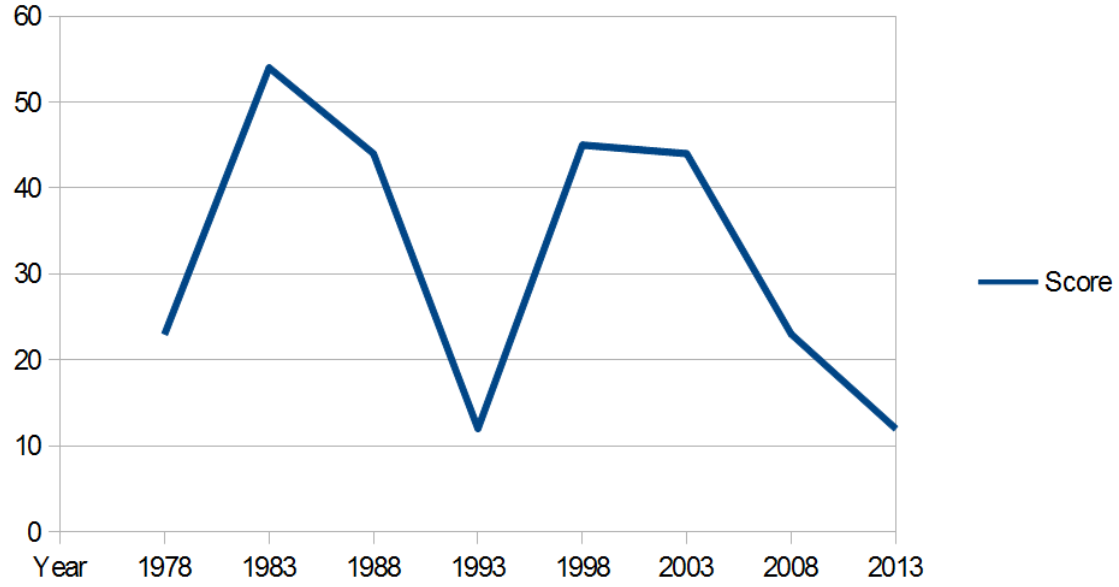


Figure 29: Line Chart

Figure 29 is a simple line chart that uses the year as the x-axis. It progresses from 1978 to 2013, and at each year a score variable is plotted and connected with the previous point as a straight line. The important difference is that the data must be sorted by the x-axis prior to rendering the lines; otherwise, something like Figure 30 may happen.

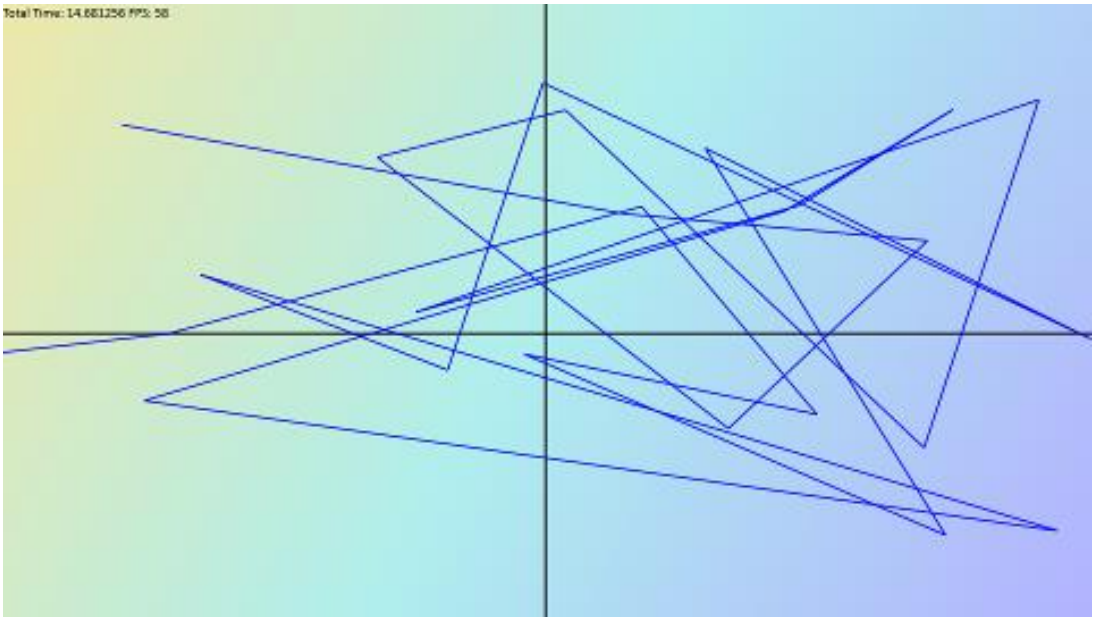


Figure 30: Line Chart with Unordered X-Axis

Figure 30 is certainly a line chart, but the line is drawn backwards and forwards with regards to the x-axis values, since the nodes were created in a random order. To properly render a line chart from data with an unordered x-axis, the data must be sorted. I have used the STL (Standard Template Library) stable sort in the following code.

Tip: Always sort the data outside of the Update and Render methods. If the data is sorted outside these critical methods, the speed of the sorting algorithm is largely negligible. Modern hardware will easily sort 1,000,000 nodes in a few seconds, but we can't afford a few seconds in our Update or Render methods.

Add two files to your project, LineChart.h and LineChart.cpp.

```
// LineChart.h
#pragma once
#include "DirectXBase.h"
#include "GraphVariable.h"

// This class represents a variable rendered as a line
class LineChart: public GraphVariable {
      ID2D1SolidColorBrush* m_solidBrush; // Brush to draw with
      float m_lineThickness; // Thickness in pixels
      D2D1::ColorF m_color;  // The color of the line

      // Method to stable-sort the data by the x-axis
      void SortData();

public:
      // Public constructor
      LineChart(float* x, float *y, int count, D2D1::ColorF col, float
thickness);

      // Create the solid brush to draw with
      virtual void CreateDeviceDependentResources
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) override;

      // The main render method of the line class
      virtual void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
override;

};
```

```
// LineChart.cpp
#include "pch.h"
#include "LineChart.h"
#include <vector>
#include <algorithm>
using namespace D2D1;
using namespace DirectX;
// Comparison method used by the stable-sort below
bool ComparePoints(D2D1_POINT_2F a, D2D1_POINT_2F b) {
return a.x < b.x; // Sort on x values
```

```
}

LineChart::LineChart(float* x, float *y, int count,
      D2D1::ColorF col, float thickness = 3.0f): m_color(0), GraphVariable(x,
y, count) {
// Save these settings to member variables to
// create the brush later:
this->m_color = col;
this->m_lineThickness = thickness;

// Sort the data by the x-axis
SortData();
}

void LineChart::SortData()
{
// Sort the data by the x-axis
std::vector<D2D1_POINT_2F> sortedNodes(m_points, m_points + m_nodeCount);

// Note the use of the stable sort, using an unstable sort
// like Quicksort will produce unexpected results!
std::stable_sort(sortedNodes.begin(), sortedNodes.end(), ComparePoints);

// Copy the sorted points back to the m_pointsArray
int counter = 0;
for(std::vector<D2D1_POINT_2F>::iterator nodeIterator = sortedNodes.begin();
      nodeIterator != sortedNodes.end(); nodeIterator++, counter++) {
      m_points[counter].x = (*nodeIterator).x;
      m_points[counter].y = (*nodeIterator).y;
      }
}

void LineChart::CreateDeviceDependentResources(
      Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// Create the solid color brush
DX::ThrowIfFailed(context->CreateSolidColorBrush(
      m_color, D2D1::BrushProperties(1.0f), &m_solidBrush));
}
```

To create and render a LineChart, we can replace the code we used to create the ScatterPlot. It is also common to render ScatterPlot nodes over the top of a line chart, so I will include the ScatterPlot object as well. The ScatterPlot nodes will be used to accent the LineChart. Add the header to the GraphRenderer.h file.

```
//
// Additional headers for graph objects here
//
#include "GradientBackground.h"
#include "ScatterPlot.h"
#include "LineChart.h"

#include "Axes.h"
```

Add a member variable called m_lineChart to the GraphRenderer class.

```
	// Member variables for displaying FPS
	float m_timeDelta;	// Time since last update call
	float m_timeTotal;	// Total time of application
	Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;
	Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;

	// Solid background
	GradientBackground* m_gradientBackground;

	// Plottable data
	GraphVariable* m_graphVariable;
	GraphVariable* m_lineChart;

	// Axes

	Axes* m_axes;
```

Call the constructor for the LineChart object in the GraphRenderer constructor. I am plotting the LineChart and the ScatterPlot using the same data and color.

```
GraphRenderer::GraphRenderer()
{
// Create the gradient background:
D2D1_COLOR_F colors[] = {
	D2D1::ColorF(ColorF::PaleGoldenrod),
	D2D1::ColorF(ColorF::PaleTurquoise),
	D2D1::ColorF(0.7f, 0.7f, 1.0f, 1.0f)
	};
float stops[] = { 0.0f, 0.5f, 1.0f };
m_gradientBackground = new GradientBackground(colors, stops, 3);

// Create the scatter plot:
const int count = 25;  // Create 25 nodes
float* x = new float[count];
float* y = new float[count];

// Create random points to plot, these
// would usually be read from some data source:
for(int i = 0; i < count; i++) {
	x[i] = (float)(rand() % 2000) - 1000;
	y[i] = (float)(rand() % 1000) - 500;
	}

m_graphVariable = new ScatterPlot(x, y, 10.0f,
	D2D1::ColorF::Chocolate,
	NodeShape::Circle, count);

// Create the line chart
m_lineChart = new LineChart(x, y, count, D2D1::ColorF::Chocolate, 5.0f);

delete[] x;
delete[] y;
```

```
// Create the axes lines
m_axes = new Axes(D2D1::ColorF::Black, 5.0f, 0.75f);

}
```

Call the line chart's **CreateDeviceDependentResources** method to create the brush to use when drawing the lines.

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

// Call the create device resources for our graph variable
m_graphVariable->CreateDeviceDependentResources(m_d2dContext);

// Create device resources for the line chart
m_lineChart->CreateDeviceDependentResources(m_d2dContext);

// Create the brush for the axes
m_axes->CreateDeviceDependentResources(m_d2dContext);

// Create the solid brush for the text
DX::ThrowIfFailed(
      m_d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Black),&m_blackBrush));

}
```

And finally, call the **m_lineChart::Render** method in the **GraphRenderer::Render** method just prior to plotting the ScatterPlot nodes.

```
//
// Draw objects here
//
// Render the graph variable(s)
m_lineChart->Render(m_d2dContext);

m_graphVariable->Render(m_d2dContext);
```

Upon running the application, you should see something like Figure 31.

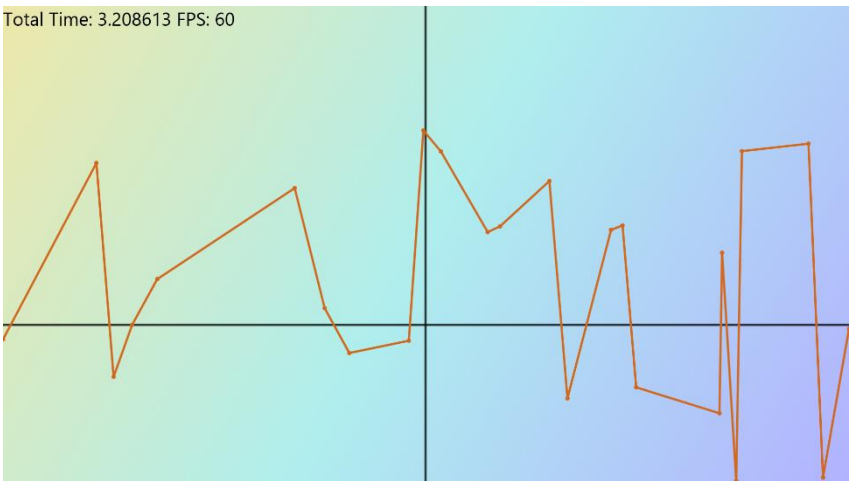


Figure 31: Line Chart

The data in the x-axis may already be ordered (since it may have been collected in chronological order), in which case it need not be sorted at all. The **`LineChart`** class should be fairly self-explanatory. We have created a list of points from the data passed to the constructor and in the render method, and we join the points with a collection of lines.

Chapter 9: Navigating between Multiple XAML Pages

Up until now, our charts have consisted of a single XAML page, which is being drawn upon by Direct2D. It is often necessary to include more than one XAML page. I will use the example of a settings page where the user can select some options for the line chart. This will be a completely separate page from the main XAML page, and each of the chart's objects could have their own settings XAML page, which can be made in the same way.

In a plain XAML project, the programmer can use the simple **`Frame->Navigate(destination)`** syntax. This is not possible in an application with Direct2D, as the Frame member variable will be null (it will not be set up at all by the framework). Instead of using frames, the programmer can navigate manually using the following syntax.

```
Window::Current->Content = ref new SomeOtherXAMLPage();// Reference another
page

Window::Current->Activate();
```

The first line sets the content of the current window to another XAML page, and the second line activates it. To activate a XAML page means brings it to the front, and gives it focus so it receives the input events.

This new XAML subpage will most likely need to return to the original page. For instance, if it is altering some settings in the chart, we will need to display the chart again after the user has changed the settings. We could create another copy of the original page using the previous code (to be executed when the user closes the subpage), but this would be wasteful and likely lead to a crash (since we would be re-creating the main DirectX pages repeatedly and never closing them). We want to reinstate the original page and give it focus rather than create more than one copy. There are many ways to do this. One of the simplest is to give the constructor of the new page a handle to the parent XAML page.

```
Window::Current->Content=ref new SomeOtherXAMLPage(this);//Reference parent
page
```

The second page saves the "this" pointer passed in its constructor as **`m_myParent`**, and when the second page closes, it won't re-create the parent, but reinstate it from this parent handle.

```
Window::Current->Content = m_myParent; // Give the parent the focus

Window::Current->Activate();
```

The "this" pointer passed to the subpage can have as generic a type as possible to allow almost any window to open any subwindow. The **`Windows::UI::Xaml::UIElement^`** is a good choice.

We will now examine adding a second page by creating a XAML page that allows users to edit a setting for our line chart, and then returns control back to the main **`DirectXPage`** class to draw the updated chart. To add a new blank XAML page to your solution, right-click the solution and click **Add**, and then click **New Item** on the context menu.

Click **Windows Store** on the left panel, then click **Blank Page** on the middle panel and type a name for the new page. I have used EditLine in this example, as per Figure 32.

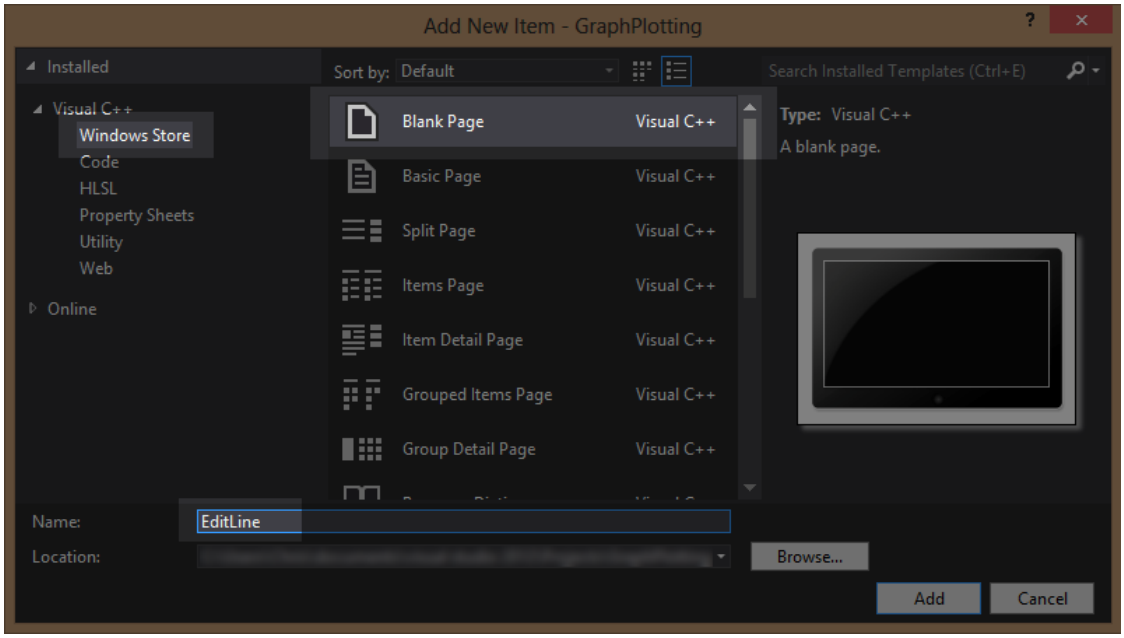


Figure 32: Add a Blank Page

Once Visual Studio has created the new XAML page, open the header (**`EditLine.xaml.h`** in this example) and add a parameter to the constructor, specifying that it requires a **`Windows::UI::Xaml::UIElement^`**. Also, add a member variable with the same type called **`m_myParent`**.

```
//

// EditLine.xaml.h

// Declaration of the EditLine class

//

#pragma once

#include "EditLine.g.h"

namespace GraphPlotting

{

      /// <summary>

      /// Empty page that can be used on its own or navigated to within a
Frame.

      /// </summary>

      [Windows::Foundation::Metadata::WebHostHidden]
```

```
	public ref class EditLine sealed{

	public:

		EditLine(Windows::UI::Xaml::UIElement^ myParent);

	protected:

		virtual void OnNavigatedTo

		(Windows::UI::Xaml::Navigation::NavigationEventArgs^ e) override;

	private:

		Windows::UI::Xaml::UIElement^ m_myParent;

	};

}
```

Double-click the new XAML page (EditLine.xaml file in this example) in the Solution Explorer, and Visual Studio will open the XAML designer with our blank page. Add a button to the top left corner. This will be our back button, and will allow users to navigate back to the main DirectX page, see Figure 33.

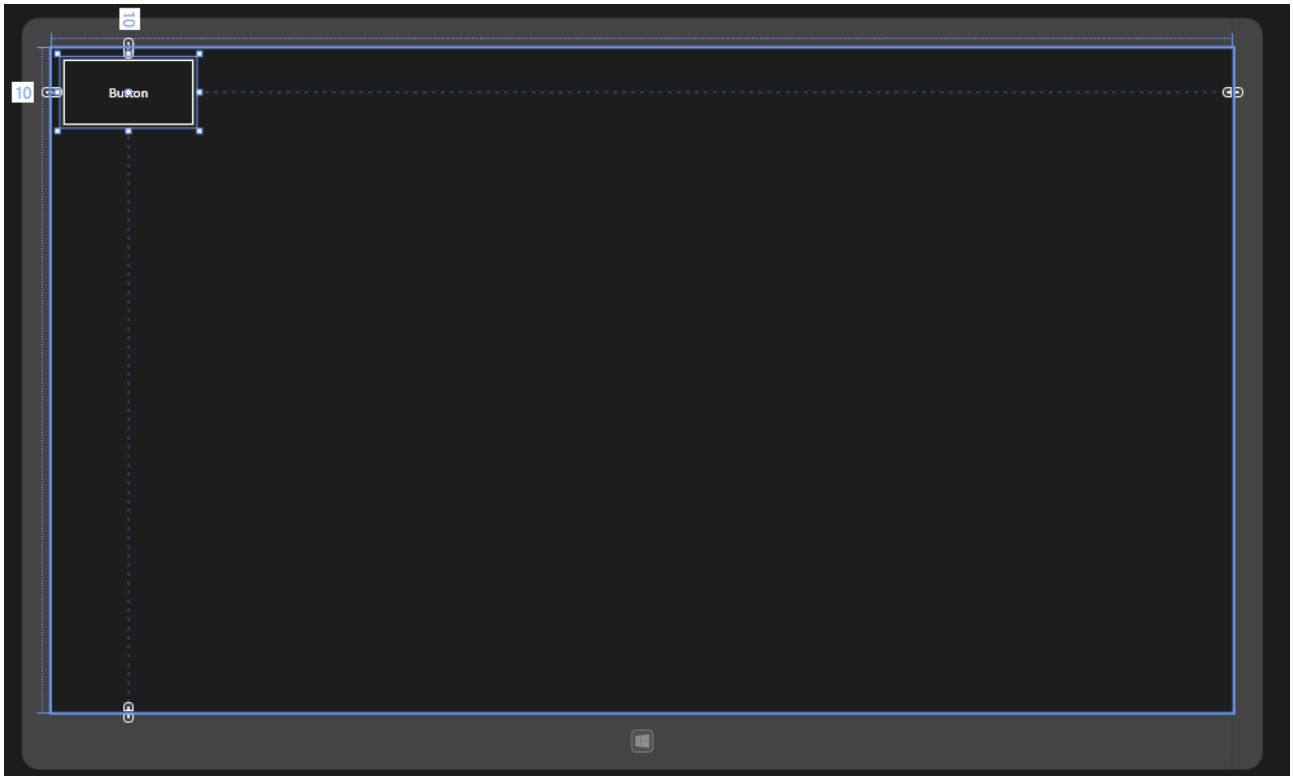


Figure 33: Adding a Button

In the XAML code section you can set the name, content, and font size of the new button.

```
<Page
    x:Class="GraphPlotting.EditLine"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GraphPlotting"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Button Name="btnBack" Content="Back" FontSize="32"
            HorizontalAlignment="Left" Height="82" Margin="10,10,0,0"
            VerticalAlignment="Top" Width="157"/>

    </Grid>
</Page>
```

Double-click the new button in the designer, and Visual Studio will write the **OnClicked** event and take us to the code. We want the parent to become activated when the user clicks the back button.

```
void GraphPlotting::EditLine::btnBack_Click(Platform::Object^ sender,
            Windows::UI::Xaml::RoutedEventArgs^ e){
      Window::Current->Content = m_myParent; // Give the parent the focus
      Window::Current->Activate();
      }
```

Change the constructor of the page so it takes the new parent UIElement as a parameter, which we specified in the header. Save the handle as the class's **m_myParent** member variable.

```
EditLine::EditLine(Windows::UI::Xaml::UIElement^ myParent){
      InitializeComponent();
      this->m_myParent = myParent;
}
```

Now that we have our subpage set up to open and close, we can design the method by which the user should request to edit the LineChart (a button in an AppBar in this example). Open the DirectXPage.xaml file by double-clicking it in the Solution Explorer. This will take you to the XAML designer for the main DirectXPage. Add an AppBar with a button to the DirectXPage XAML file.

```
<Page
    x:Class="GraphPlotting.DirectXPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:GraphPlotting"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <SwapChainBackgroundPanel x:Name="SwapChainPanel"
PointerMoved="OnPointerMoved"
            PointerReleased="OnPointerReleased">
    </SwapChainBackgroundPanel>
    <Page.BottomAppBar>
        <AppBar Padding="10, 0, 10, 0">
            <Button Name="btnEditLine" Content="Edit Line" FontSize="24"
                HorizontalAlignment="Center" Width="240"/>
        </AppBar>
    </Page.BottomAppBar>
</Page>
```

Tip: Working with AppBar and several other object types is far easier in the XAML code window than it is in the main designer window, since these controls are often invisible in the designer. To have the designer window show an invisible control, you can select the control's opening tag in the XAML code.

Double-click the new button in the designer (btnEditLine) and Visual Studio will add the **OnClicked** event and take us to the code. The first thing to do is #include a reference to the EditLine.xaml.h header at the top of this file.

```
// DirectXPage.xaml.cpp

#include "pch.h"

#include "DirectXPage.xaml.h"

#include "EditLine.xaml.h"
```

Next, scroll back down to the button clicked event, so we can add the code to create and activate the edit line window.

```
void GraphPlotting::DirectXPage::btnEditLine_Click(Platform::Object^ sender,

            Windows::UI::Xaml::RoutedEventArgs^ e){

      Window::Current->Content = ref new EditLine(this);

      Window::Current->Activate();

      }
```

Upon running the application, you should be able to bring up the app bar by right-clicking (or swiping with the pointer on a touchscreen device). Upon clicking the button, you will be presented with the Edit Line page, and from there you can click the Back button to return to the graph renderer.

As an example of changing the value of a graph object with the new XAML page, we will allow the user to set the thickness of our line chart. Add a public getter and setter to the **LineChart** class for the line's thickness in the LineChart.h file.

```
// Public constructor
      LineChart(float* x, float *y, int count, D2D1::ColorF col, float
thickness);

      // Create the solid brush to draw with
      virtual void CreateDeviceDependentResources
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) override;

      // The main render method of the line class
      virtual void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
```

```
override;

	// Getters and setters for line thickness
	void SetLineThickness(float newThickness) {
		m_lineThickness = newThickness;
		}

	float GetLineThickness() {
		return m_lineThickness;

		}
```

Add a public internal getter to the GraphRenderer.h file that returns the class's line chart member variable.

```
public:
	// Public constructor
	GraphRenderer();

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

	// Capture the pointer movements so the user can pan the chart
	void PointerMoved(Windows::Foundation::Point point);

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

internal:
	LineChart* GetLine() {
		return (LineChart*) m_lineChart;

		}
```

Add an #include to the LineChart header in the EditLine.xaml.h file. Add a new argument to the constructor that is a pointer to a LineChart, mark the constructor as internal, and then add a new LineChart pointer member variable in which to store that argument.

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

#pragma once
#include "EditLine.g.h"
#include "LineChart.h"
namespace GraphPlotting{
	/// <summary>
	/// An empty page that can be used on its own or navigated to within a
frame.
```

```
      /// </summary>
      [Windows::Foundation::Metadata::WebHostHidden]
      public ref class EditLine sealed
      {
      public:
      internal:
            EditLine(Windows::UI::Xaml::UIElement^ myParent, LineChart*
myLine);
      protected:
            virtual void OnNavigatedTo
                  (Windows::UI::Xaml::Navigation::NavigationEventArgs^ e)
override;
      private:
            Windows::UI::Xaml::UIElement^ m_myParent;
            LineChart* m_myLine;
            void btnBack_Click(Platform::Object^ sender,
                  Windows::UI::Xaml::RoutedEventArgs^ e);
            };
}
```

Add a call to the m_renderer::GetLine method and pass this as a parameter in the DirectXPage::btnEditLine_Click event of the DirectXPage.xaml.cpp file.

```
void GraphPlotting::DirectXPage::btnEditLine_Click(Platform::Object^ sender,
Windows::UI::Xaml::RoutedEventArgs^ e) {
      Window::Current->Content = ref new EditLine(this, m_renderer-
>GetLine());
      Window::Current->Activate();

      }
```

Add a slider control to the EditLine.xaml file. This will be used to set the thickness of the line. I have called it **sldLineThickness**. I have also added a text block to the grid in the following code, which is used to label the slider for users. The complete code for the grid of the EditLine.xaml file follows:

```
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Button Name="btnBack" Content="Back" FontSize="32"
HorizontalAlignment="Left" Margin="10,10,0,0" VerticalAlignment="Top"
Click="btnBack_Click"/>

    <Slider Name="sldLineThickness" HorizontalAlignment="Left"
Margin="420,302,0,0"
            VerticalAlignment="Top" Width="573" Height="47" Minimum="1"
Maximum="12"/>

    <TextBlock FontSize="32" HorizontalAlignment="Left" Margin="185,302,0,0"
                TextWrapping="Wrap" Text="Line Thickness"
                VerticalAlignment="Top"/>

    </Grid>
```

Alter the constructor's code in the EditLine.xaml.cpp file to take a **LineChart** parameter and save it to the **m_myLine** member variable. I have also set the initial value for the slider to be the same as the current thickness of the **myLine** parameter.

```
EditLine::EditLine(Windows::UI::Xaml::UIElement^ myParent, LineChart* myLine)

{

	InitializeComponent();

	this->m_myParent = myParent;

	this->m_myLine = myLine;

	this->sldLineThickness->Value = (int)myLine->GetLineThickness();

}
```

Finally, in the **btnBack_Click** event method in the EditLine.xaml.cpp file, just prior to handing control back to the parent window, we can call the line chart's set line thickness method and set the thickness of the line to **sldLineThickness->Value**.

```
void GraphPlotting::EditLine::btnBack_Click(Platform::Object^ sender,

		Windows::UI::Xaml::RoutedEventArgs^ e)   {

	m_myLine->SetLineThickness((float)sldLineThickness->Value);

	Window::Current->Content = m_myParent; // Give the parent the focus

	Window::Current->Activate();

	}
```

Upon running the application, you should be able to set the thickness of the line chart by moving the slider. This example only allowed changing the thickness of the line, but other controls could be added to the page to allow editing other aspects of the line. New XAML pages could be added to change any aspect of the chart.

Chapter 10: Printing Direct2D

In this chapter we will walk through how to add printing to a standard Direct2D (XAML) application, which users can activate from the charms bar by selecting the Devices icon. I have described how to add printing to a new Direct2D (XAML) project, as opposed to adding the functionality to our existing application, because printing Direct2D is applicable beyond graphing and charting data.

Note: Most of the code I have presented here is either based heavily on the Direct2Dapp printing sample from Microsoft, or taken directly from this sample. The Microsoft sample is designed as a standalone application. In this walk-through we will examine how to add printing to a Direct2D (XAML) application, rather than writing a completely new application. If you are starting a new project from scratch, it may be beneficial to more closely model the structure of your application after the Microsoft sample called "Direct2Dapp printing sample." See Appendix A for Microsoft's license for the use of the code from its samples.

Create a new Direct2D XAML Project

Create a new Direct2D (XAML) app. Even if you are adding printing functionality to an existing project, it is recommended that you step through this process with a new Direct2D (XAML) app in order to familiarize yourself with the structure of a Direct2D printing application.

Open the DirectXPage.xaml file and delete the text that says "Hello XAML." This example will not print out the contents of the XAML controls.

Note: XAML controls are not related to the Direct2D graphics we will be printing in this example. If you wish to print XAML controls, Microsoft has a separate sample application titled "Quickstart: Printing from your app (Windows Store apps using C#/VB/C++ and XAML)."

Make your application Multithreaded

Direct2D printing occurs on a separate thread with another device context. There is also a separate thread for rendering the print preview. The screen, the preview, and the printing can all occur at once without interfering with each other. The main Direct2D factory must be created multithreaded. The option to create a multithreaded application is set in the DirectXBase class in the **`CreateDeviceIndependentResources`** method of the DirectXBase.cpp file.

```
// These are the resources required independent of the device.

void DirectXBase::CreateDeviceIndependentResources(){

      D2D1_FACTORY_OPTIONS options;

      ZeroMemory(&options, sizeof(D2D1_FACTORY_OPTIONS));

#if defined(_DEBUG)

       // If the project is in a debug build, enable Direct2D debugging via
SDK Layers.
```

```
	options.debugLevel = D2D1_DEBUG_LEVEL_INFORMATION;
#endif
	DX::ThrowIfFailed(
		D2D1CreateFactory(
			D2D1_FACTORY_TYPE_MULTI_THREADED,
			__uuidof(ID2D1Factory1),&options,&m_d2dFactory
			)
		);
```

Add the Print Manager

Add a "using namespace" directive to the top of the DirectXPage class in the DirectXPage.cpp file to use the Printing namespace. There are several other changes that must be made to this class. See Appendix B for a complete listing of the DirectXPage class.

```
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;
using namespace Windows::Graphics::Printing;


DirectXPage::DirectXPage() :
```

Add a print manager member variable to your DirectXPage class in the DirectXPage.h file.

```
BasicTimer^ m_timer;
	Windows::Graphics::Printing::PrintManager^ m_printManager;
};
```

Initialize the Print Manager

Initialize the print manager in the constructor of the DirectXPage class in the DirectXPage.cpp file. This can occur after the other code. I have placed it directly under the initialization of the m_timer. To initialize it we need to do two things: grab the print manager for the current program, and register an event handler so we know when the user requests to print something.

```
m_timer = ref new BasicTimer();
```

```
// Grab the print manager for the current view

m_printManager =
Windows::Graphics::Printing::PrintManager::GetForCurrentView();

// Add an event handler to capture when the user requests a print task

m_printManager->PrintTaskRequested +=

      ref new TypedEventHandler<PrintManager^,
PrintTaskRequestedEventArgs^>(this,

      &DirectXPage::SetPrintTask);
```

In this code we have specified that there is a handler method called SetPrintTask, which the print manager will use to make new printing tasks.

Create the SetPrintTask Method

This method is very important, as it determines when the user has requested to print something from the charms bar. It will be called when the user selects Devices from the charms bar, and depending on what the event does, it will either enable the application to print, or the charms bar will say "this app can't send to other devices at the moment." Add the SetPrintTask prototype to the DirectXPage.h header.

```
      BasicTimer^ m_timer;

      Windows::Graphics::Printing::PrintManager^ m_printManager;

internal:

      // Print task requested event handler method

      void SetPrintTask(_In_ Windows::Graphics::Printing::PrintManager^
sender,

            _In_ Windows::Graphics::Printing::PrintTaskRequestedEventArgs^
args);

      };
```

Note: The declaration of the SetPrintTask (as well as many other methods in this sample) must be marked as internal, since it contains native data types in its prototype, and the class is marked as ref, meaning it is a .NET Framework Reference class. Reference classes cannot contain public methods or data based on native types.

The following listing is the method body for the SetPrintTask method. I have placed it after the LoadInternalState method as the final method in the DirectXPage.cpp file.

```
void DirectXPage::LoadInternalState(IPropertySet^ state) {
```

```
        m_renderer->LoadInternalState(state);

}
void DirectXPage::SetPrintTask(_In_ PrintManager^ sender,

      _In_ PrintTaskRequestedEventArgs^ args){

// Create a new source requested handler

PrintTaskSourceRequestedHandler^ sourceRequestedHandler = ref new
PrintTaskSourceRequestedHandler(

      [this](PrintTaskSourceRequestedArgs^ args)-> void {

            Microsoft::WRL::ComPtr<CDocumentSource> documentSource;

            DX::ThrowIfFailed (

                  Microsoft::WRL::MakeAndInitialize<CDocumentSource>(


      &documentSource,reinterpret_cast<IUnknown*>(m_renderer)

                        )

                  );



                  // Cast the document to an object

                  IPrintDocumentSource^ objSource(
      reinterpret_cast<IPrintDocumentSource^>(documentSource.Get())

                  );

            args->SetSource(objSource);

            });

// Create the print task

PrintTask^ printTask = args->Request->CreatePrintTask(L"Direct 2D Printing
Example",

            sourceRequestedHandler);

}
```

Note: This method uses the new (C++11) syntax for lambda expressions since the new event handler needs access to the data members of the outer DirectXPage class. These expressions are similar in functionality to anonymous inner types in Java. The code is from Microsoft. Please refer to Appendix A for the license and reuse of this code.

The DirectX (XAML) template has several references to methods in the **SimpleTextRenderer**, which will be removed when we rewrite the class. It also references the **CDocumentClass** from Microsoft, which I have included as Appendix C. It creates and registers a **PrintTaskSourceRequestedHandler** event handler. This event is fired when the print task requests a document to be printed. It must be called to finish the initialization of the **PrintTask** object. As mentioned, for a full listing of the final altered DirectXPage class without these methods, refer to Appendix B.

Add DocSource class

Next, we will create the document source class referenced in the previous code. Add an #include to the top of the DirectXPage.cpp file to include the header for a new class called DocSource.h.

```
#include "pch.h"

#include "DirectXPage.xaml.h"

#include "DocSource.h"
```

Add a header and a code file for the new Document Source class called DocSource.h and DocSource.cpp. The complete code listing (from Microsoft's printing sample) for the Document Source class is included as Appendix C.

Split the SimpleTextRenderer

The next step is to enable the SimpleTextRenderer class to be created more than once to create multiple contexts. The multiple device contexts will run in parallel. The rendering method itself must also be updated to consider the format of the output. The dimensions of a computer monitor are usually different from those of a standard A4 piece of paper. The Rendering method of this class must now consider the differences in these dimensions, and alter the layout of the image if required. There is no standard way to format output from a monitor such that it perfectly suits a physical piece of paper. Some suggestions would be that the background not be colored when printing, since printing a solid colored background is very slow and wastes a lot of ink.

Tip: The standard orientation of the printed page is portrait. It may match the image on the monitor better if this orientation of the printed page is changed to landscape.

The SimpleTextRenderer class contains a lot of code that is specific to this Visual Studio template. The background changing colors, loading and saving of the internal state, and the member variables for displaying the "Hello DirectX!" can be removed, since they have nothing to do with printing. The final listing for the two files is included as Appendix D. Much of the following code comes from Microsoft; Appendix A is the license agreement for use of the Microsoft code.

The actual drawing is exactly the same as we have previously examined; it is made of perfectly normal Direct2D context method calls. The method is called Draw in the example code. I have highlighted it in green. The App class must also be updated and the Visual Studio template methods dealing with loading and saving the internal state can be removed.

The final listing for App.xaml.cpp is as follows.

```
//
// App.xaml.cpp
// Implementation of the App class.
//
#include "pch.h"
#include "DirectXPage.xaml.h"
using namespace DXPrinting;
using namespace Platform;
using namespace Windows::ApplicationModel;
using namespace Windows::ApplicationModel::Activation;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
using namespace Windows::Storage;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Interop;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;
App::App(){
	InitializeComponent();
	Suspending += ref new SuspendingEventHandler(this, &App::OnSuspending);
}

void App::OnLaunched(LaunchActivatedEventArgs^ args){
	m_directXPage = ref new DirectXPage();
```

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

void App::OnSuspending(Object^ sender, SuspendingEventArgs^ args){
	(void) sender; // Unused parameter.
	(void) args; // Unused parameter.
}
```

After implementing the code in this chapter and the appendices, you should be able to run the application and print a pattern of circles (rendered in the Draw method). The same pattern of colored circles can be drawn to the screen, the print preview, or printed to a page using the charms bar.

The code for this chapter was lengthy and mostly taken from the Microsoft Direct2D printing sample. Interacting with printers is a complex task, and all the boilerplate code for this type of activity should be taken from the Microsoft samples. Printing DirectX is really a job for the operating system, and has very little to do with DirectX. Minor deviations from the standard code as presented in the Microsoft samples will not work.

Chapter 11: Margins

We now return to our charting application, where we will add a margin around the edge. I am going to add to the code as it was after the "Navigating between Multiple XAML Pages" chapter. Adding a margin to a chart is important, as it can be used to separate the title, axis labels, and grid figures from the data being plotted. The following margin class consists of a collection of four rectangles and a solid color brush. The rectangles form a border around the visible chart area. Add a Margin.h and Margin.cpp file to your project.

```
// Margin.h

#pragma once

#include "DirectXBase.h"

enum MarginStyle { Absolute, WindowSizeDependent };


class Margin {

      ID2D1SolidColorBrush* m_solidBrush; // Brush to draw margin

      D2D1::ColorF m_color;  // The color of the margin

      D2D1_RECT_F m_leftRect;// Rectangles which make the margin

      D2D1_RECT_F m_rightRect;

      D2D1_RECT_F m_topRect;

      D2D1_RECT_F m_bottomRect;

      MarginStyle m_style;   // Style is Absolute or Window size dependent

      // The size of the margin

      float m_left, m_right, m_top, m_bottom;

public:

      // Public constructor

      Margin(float left, float right, float top, float bottom,

      D2D1::ColorF color, MarginStyle style);


      // Create the rectangles to draw the margin

      void CreateWindowSizeDependentResources
```

```
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);

      // Create the solid brush to draw with
      void CreateDeviceDependentResources
            (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);

      // The render method needs to know the panning and scaling
      void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context);
      };
```

```
// Margin.cpp
#include "pch.h"
#include "Margin.h"
Margin::Margin(float left, float right, float top, float bottom,
      D2D1::ColorF color, MarginStyle style)   : m_color(0) {
// Save the parameters for the create resources methods
this->m_color = color; m_style = style; this->m_left = left; this->m_right =
right;
this->m_top = top; this->m_bottom = bottom;
}

void Margin::CreateWindowSizeDependentResources(
      Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// If the sizes of the margin passed were percentages of the screen
// we have to multiply the values before creating the rectangles:
if(m_style == MarginStyle::WindowSizeDependent) {
      m_left = context->GetSize().width * m_left;
```

```
      m_right = context->GetSize().width * m_right;
      m_top = context->GetSize().height * m_top;
      m_bottom = context->GetSize().height * m_bottom;
      }

// Left rectangle
m_leftRect.left = 0; m_leftRect.right = m_left;
m_leftRect.top = 0; m_leftRect.bottom = context->GetSize().height;

// Right rectangle
m_rightRect.left = context->GetSize().width - m_right;
m_rightRect.right = context->GetSize().width;
m_rightRect.top = 0; m_rightRect.bottom = context->GetSize().height;

// Top margin
m_topRect.left = m_left;
m_topRect.right = context->GetSize().width - m_right;
m_topRect.top = 0; m_topRect.bottom = m_top;

// Bottom margin
m_bottomRect.left = m_left;
m_bottomRect.right = context->GetSize().width - m_right;
m_bottomRect.top = context->GetSize().height - m_bottom;
m_bottomRect.bottom = context->GetSize().height;
}

void Margin::CreateDeviceDependentResources
      (Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
// Create the brush
```

```
DX::ThrowIfFailed(context->CreateSolidColorBrush(m_color, &m_solidBrush));

}


void Margin::Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {

// Draw the left margin

context->FillRectangle(m_leftRect, m_solidBrush);

// Draw the right margin

context->FillRectangle(m_rightRect, m_solidBrush);

// Draw the top margin

context->FillRectangle(m_topRect, m_solidBrush);

// Draw the bottom margin

context->FillRectangle(m_bottomRect, m_solidBrush);

}
```

To add a margin object, we need to add a member variable to the graph renderer. Load its resources when appropriate and render it.

Note: The margin class takes a MarginStyle enum as one of the parameters to the constructor. I have added this to enable margins to be created dependent on the size and/or resolution of the device running the application. Using Absolute as this parameter results in the margin's thickness parameters being interpreted as pixels. Using WindowSizeDependent means that the parameters passed in are interpreted as a percentage of the context width and height.

Add the Margin.h file to the Graph Renderer.h headers section:

```
//
// Additional headers for graph objects here
//
#include "GradientBackground.h"
#include "ScatterPlot.h"
#include "LineChart.h"
#include "Axes.h"

#include "Margin.h"
```

Add the member variables to the GraphRenderer.h file.

```
// Plottable data
      GraphVariable* m_graphVariable;
      GraphVariable* m_lineChart;
```

```
	// Axes
	Axes* m_axes;

	// Margin

	Margin* m_margin;
```

Call the constructor(s) for the margin(s) in the **GraphRenderer** constructor; this can be done at any time. I have placed it after constructing the **m_axes** object.

```
// Create the line chart
m_lineChart = new LineChart(x, y, count, D2D1::ColorF::Chocolate, 5.0f);

delete[] x;
delete[] y;

// Create the Axes
m_axes = new Axes(D2D1::ColorF::Black, 5.0f, 0.75f);

// Create the margin
m_margin = new Margin(0.1f, 0.1f, 0.1f, 0.1f,  // 10% of window size
	D2D1::ColorF(0.38f, 0.66f, 0.74f, 1.0f),
MarginStyle::WindowSizeDependent);

}
```

Call the **CreateDeviceDependentResources** method for the new m_margin object in the **GraphRenderer::CreateDeviceResources** method (this will create the margin's brush).

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

// Call the create device resources for our graph variable
m_graphVariable->CreateDeviceDependentResources(m_d2dContext);

// Create device resources for the line chart
m_lineChart->CreateDeviceDependentResources(m_d2dContext);

// Create the brush for the Axes
m_axes->CreateDeviceDependentResources(m_d2dContext);

// Create the solid brush for the text
DX::ThrowIfFailed(
	m_d2dContext-
>CreateSolidColorBrush(ColorF(ColorF::Black),&m_blackBrush));

// Create the margin's device dependent resources
m_margin->CreateDeviceDependentResources(m_d2dContext);

}
```

Call the **CreateWindowSizeDependentResources** method for the margin(s) in the **GraphRenderer::CreateWindowSizeDependentResources** method (this will create the rectangles to fill for the margin).

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

// Create window size resources for gradient background
m_gradientBackground->CreateWindowSizeDependentResources(m_d2dContext);

// Set the initial pan value so the lowest node is visible in the corner
m_pan.X = -m_graphVariable->GetMinX();
m_pan.Y = m_graphVariable->GetMinY();

// Create window size dependent resources for the margin
m_margin->CreateWindowSizeDependentResources(m_d2dContext);

}
```

And finally, call Render on the margin(s) in the **GraphRenderer::Render** method. I have rendered the margins immediately after the transformation matrix is reset, and just prior to rendering the FPS counter. We originally added this reset to make sure the FPS counter was not affected by the pan and zoom, but we can also use it to ensure the margin is not transformed.

```
//
// Draw objects here
//
// Render the graph variable(s)
m_lineChart->Render(m_d2dContext);
m_graphVariable->Render(m_d2dContext);

// Reset the transform matrix so the time and FPS do not pan or zoom
m_d2dContext->SetTransform(m_orientationTransform2D);

// Render the margin
m_margin->Render(m_d2dContext);

// Set up the string to print:
std::wstring s = std::wstring(
      L"Total Time: ") + std::to_wstring(m_timeTotal) +
      std::wstring(L" FPS: ") + std::to_wstring(

            (int)(0.5f+1.0f/m_timeDelta)); // FPS rounded to nearest int
```

Upon debugging the application, you should see a stormy cyan colored margin surrounding the window, as shown in Figure 34.

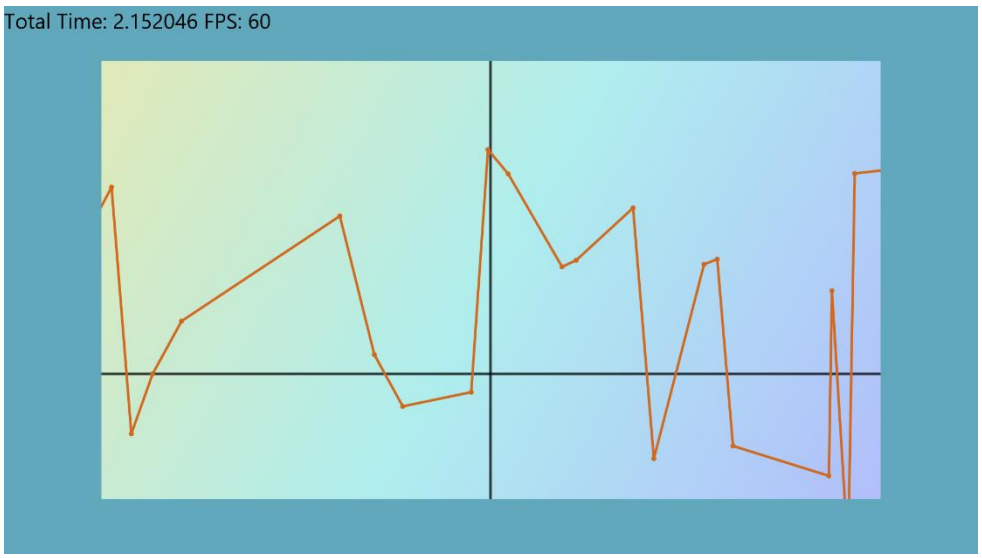


Figure 34: Margins

Chapter 12: Zooming

As mentioned in the section on transformations, the entire view can be zoomed in or out using a simple Scale matrix. We will attach an event to the DirectXRenderer page that captures the mouse wheel (if the user is using a Windows 8 desktop), and allow the wheel to alter the zoom of our chart.

We already have a scale matrix being applied to our data, but its task is to flip the y-axis, so we are not drawing our chart upside down. To create a zoom capability, we can add an additional multiplication to the values in the parameters of this matrix. I have used two member floats to hold the zoom coefficients, and I have used #define to specify some limits, avoid zooming in too far, and to avoid division by zero errors that can occur when zooming out so far that it underflows the 32-bit floats. I've made the changes to the GraphRenderer.h file.

```
// Member variables for displaying FPS
      float m_timeDelta;     // Time since last update call
      float m_timeTotal;     // Total time of application
      // Member variables and constants for zooming
      #define MIN_ZOOM (0.01f)     // Smallest zoom value is 1%
      #define MAX_ZOOM (100.0f)    // Largest zoom value is 10,000%
      float m_zoomX;    // The amount the x-axis is scaled by
      float m_zoomY;    // The amount the y-axis is scaled by
      Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;
      Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;
```

Initialize the floats we just defined in the constructor of the **GraphRenderer** in the GraphRenderer.cpp file.

```
GraphRenderer::GraphRenderer():
      m_zoomX(1.0f),
      m_zoomY(1.0f)
{
```

Add a public, internal Zoom method prototype to the GraphRenderer class so we can call it from the DirectXPage class. I have placed this prototype directly after the GetLine method prototype.

```
internal:
      LineChart* GetLine() {
            return (LineChart*) m_lineChart;
            }

      // Zooming method

      void Zoom(float amount);
```

The body for the function in the GraphRenderer.cpp file is very basic. We multiply the axis zooms by the parameter passes, and make sure they are within the limits MAX_ZOOM and MIN_ZOOM. Place the following code in the GraphRenderer.cpp file. It can be placed at the end after the Render method.

```
void GraphRenderer::Zoom(float amount) {

// Multiply the zooms

m_zoomX *= amount;

m_zoomY *= amount;



// Make sure the new zooms are still within the limits:

if(m_zoomX < MIN_ZOOM) m_zoomX = MIN_ZOOM;

else if(m_zoomX > MAX_ZOOM) m_zoomX = MAX_ZOOM;



if(m_zoomY < MIN_ZOOM) m_zoomY = MIN_ZOOM;

else if(m_zoomY > MAX_ZOOM) m_zoomY = MAX_ZOOM;

}
```

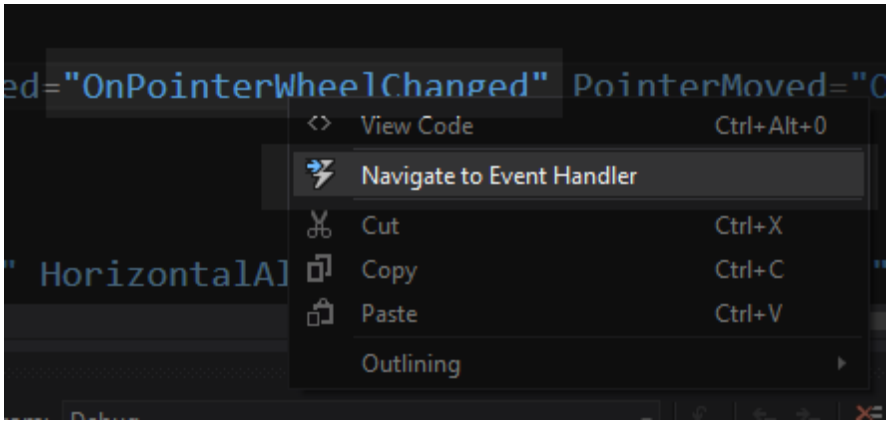
Next, we need to add an event handler to capture when the mouse wheel is changed. Open the DirectXPage.xaml file, find the line describing the SwapChainBackground panel, and add a **PointerWheelChanged** event to the XAML code.

```
<SwapChainBackgroundPanel x:Name="SwapChainPanel"

      PointerWheelChanged="OnPointerWheelChanged"
PointerMoved="OnPointerMoved"

      PointerReleased="OnPointerReleased">
```

Right-click on the event name in the XAML code (I've used **`OnPointerWheelChanged`**) and click **Navigate to Event Handler** on the context menu which appears.



Visual Studio will write the event handler code for us, and take us directly there so we can specify what is to happen when the mouse wheel changes. All we need to do is check which way the wheel was rotated, and call the **`GraphRenderer::Zoom`** method with appropriate values. I have used **`1.2f`** to zoom in and **`0.8f`** to zoom out. These values mean the zooming will be fairly smooth at around 20%. Here is the code for the **`DirectXPage::OnPointerWheelChangedEvent`**.

```
void DirectXPage::OnPointerWheelChanged(Platform::Object^ sender,
            Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e) {
      Windows::UI::Input::PointerPoint ^p = e->GetCurrentPoint(this);
      if(p->Properties->MouseWheelDelta > 0)
            m_renderer->Zoom(1.2f);
      else
            m_renderer->Zoom(0.8f);
      }
```

Add the additional multiplication of our new axis zoom values to the **`scale`** matrix in the **`GraphRenderer::Render`** method.

```
// The scale matrix inverts the y-axis
Matrix3x2F scale = Matrix3x2F::Scale(
      1.0f * m_zoomX,   // Multiply by x-axis zoom
      -1.0f * m_zoomY,  // Flip and multiply by y-axis zoom
      D2D1::Point2F(0.0f, 0.0f));
```

Upon running the application, you should find that you can now zoom in and out of the chart using the mouse wheel. The Axes class is no longer rendering properly. First, the thickness of the axis lines is being altered by the scale matrix. When the user zooms in, the axis lines become thicker. When the user zooms out, they become thinner and disappear altogether. Figure 35 is a screenshot zoomed into the meeting point of the axes, the origin.

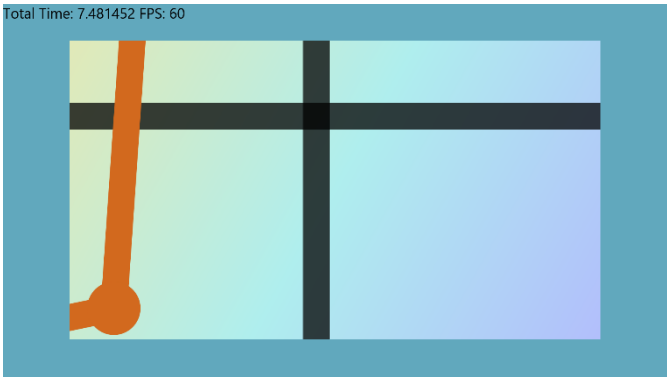


Figure 35: Zooming into Origin

The other problem is that when the chart is zoomed out, the axes' lines no longer span the entire window. This destroys the illusion that they are infinite since the ends are clearly visible. See Figure 36.

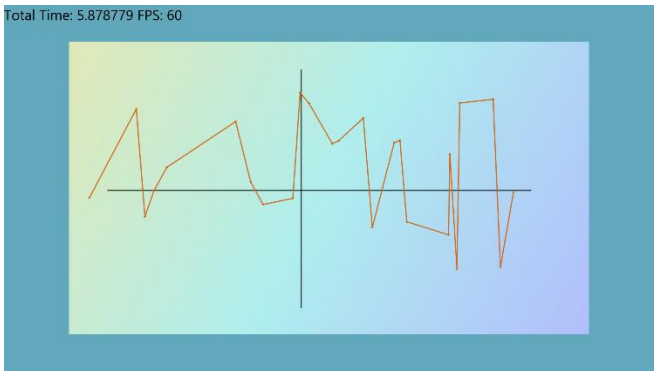


Figure 36: Zooming Out

We can fix both of these problems by multiplying by the m_zoomX and m_zoomY in the **`m_axes::Render`** method call.

```
// The pan matrix still pans but it also adds the height of the screen
Matrix3x2F panMatrix = Matrix3x2F::Translation(m_pan.X, m_pan.Y +
      m_d2dContext->GetSize().height);

// Apply the scale first
m_d2dContext->SetTransform(scale*panMatrix*m_orientationTransform2D);

// Draw the axes
m_axes->Render(m_d2dContext, m_pan.X, m_pan.Y, m_zoomX*1.0f, m_zoomY*-1.0f);

//
// Draw objects here

//
```

Upon making this change, you should be able to run the application and zoom in and out without altering the thickness of the axis lines, and without making their ends visible. See Figure 37.

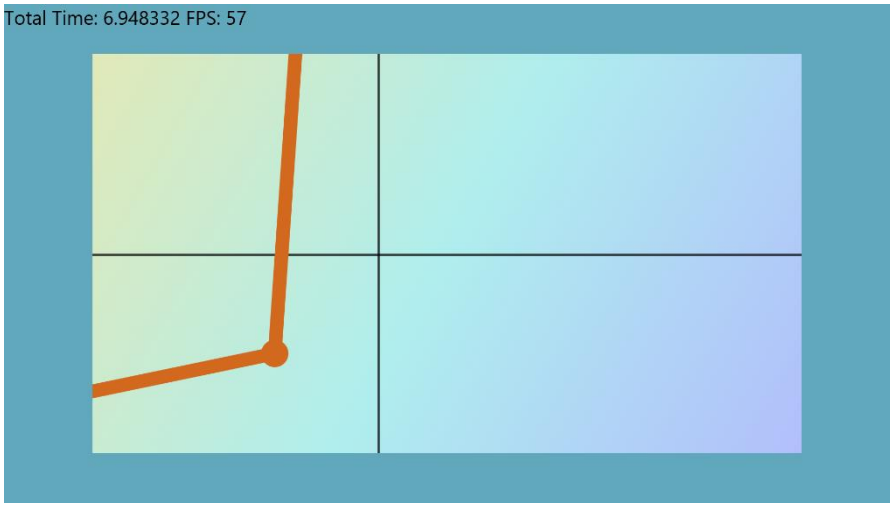


Figure 37: Zooming with Scaling Axes

Chapter 13: Hit Testing or Picking

It is often useful in a charting application to determine if the user has clicked the mouse on a node or other object in the chart. This is called hit testing or picking.

Tip: The Direct2D geometries have a method called FillContainsPoint, which returns a BOOL indicating whether the filled geometry contains a specified point. This method is not useful for hit testing a collection of simple primitives, because the geometries are device independent and are very slow at this type of task. Geometries will determine if a point lies within a complex shape fairly quickly, but they are not good at telling which of a large collection of nodes is closest to a given point.

We will examine actual hit testing (determining if a point lies inside a shape) using Direct2D in the Geometries section that follows. This section will concentrate on a more efficient way to select a node with the pointer. The particular mechanism we will create is common in graphing software; it allows the user of the chart a little freedom when they are selecting a node. The pointer doesn't need to be exactly on top of a tiny node, but just close enough. This mechanism makes selecting a single small node from a collection of many much easier for the user.

All we have to do is figure out which node is closest to the pointer, and then whether the pointer is close enough to the node. For example, if the pointer is distant from all nodes, then no nodes should be picked. If we wish to know which node the pointer is closest to, we can very quickly examine all the nodes and calculate the distance to each, determining which is the smallest. We will edit the **ScatterPlot** class and have it show the user which node is being selected by enlarging the selected node. The ellipses and points that comprise the scatter plot are all device independent resources, so the CPU is in control of them. Open the ScatterPlot.h file and add a public method prototype called PickPoint. This method will take the x and y position position of the cursor as parameters. I have added this prototype at the end of the class.

```
virtual void
CreateDeviceDependentResources(Microsoft::WRL::ComPtr<ID2D1DeviceContext>
context) override;
      virtual void Render(Microsoft::WRL::ComPtr<ID2D1DeviceContext> context)
override;

      // Method to select a node
      void PickPoint(float x, float y);
};
```

Add a private member variable to the same class. This will be used to keep track of the index of the node that is selected.

```
NodeShape m_NodeShape; // The shape of the nodes

      // Selected node index
      int m_selectedNode;


public:
```

Initialize the m_selectedNode to -1 in the **ScatterPlot** constructor.

```
ScatterPlot::ScatterPlot(float* x, float* y, float nodeSize,
```

```
      D2D1::ColorF nodeColor, NodeShape nodeShape, int count):
         m_NodeColor(0), GraphVariable(x, y, count), m_selectedNode(-1)
{
// Save half the node size. The nodes are drawn with
// the point they're representing at the middle of the shape.
this->m_HalfNodeSize = nodeSize / 2;

this->m_NodeShape = nodeShape;
this->m_NodeColor = nodeColor;

}
```

Add the body of the **PickPoint** method to the end of the ScatterPlot.cpp file.

```
void ScatterPlot::PickPoint(float x, float y) {

float smallestDistance = (x - m_points[0].x) * (x - m_points[0].x) +

      (y - m_points[0].y) * (y - m_points[0].y);


int indexOfSmallest = 0;     // Assume closest node is the first one


// Run through all nodes and see if any are closer

for(int i = 1; i < m_nodeCount; i++) {

      // Approximate the distance, don't take the sqrt()!

      float thisDistance = ((x - m_points[i].x) * (x - m_points[i].x) +

            (y - m_points[i].y) * (y - m_points[i].y));


      // If this one's closer, update the index and dist

      if(thisDistance < smallestDistance) {

            smallestDistance = thisDistance;

            indexOfSmallest = i;

            }

      }


// Calculate the sqrt to get the real Euclidean distance
```

```
smallestDistance = sqrt(smallestDistance);



// If the distance is greater than a threshhold (50.0f), assume

// no points are selected at all:

if(smallestDistance > 50.0f)

      m_selectedNode = -1; // Nothing is selected

// Otherwise, select the node that's closest to the pointer

else

      m_selectedNode = indexOfSmallest;

}
```

To work out which node is closest to our pointer, we must calculate the distance (I have used the Euclidean distance) from each of the nodes to the pointer, and decide which node is closest. Initially, I assumed the pointer is closest to the first node. Then I ran through every other node checking the distance to the pointer. If the pointer is closer to a node than our current shortest distance, update the shortest distance and the index of the node. In this way, by the time we reach the final node, we will have the index of the node that is closest to the pointer.

Finally, alter the render method of the **`ScatterPlot`** class (in the ScatterPlot.cpp file) such that it renders the selected node a different color and size. I have only included example code for circle nodes, but the idea could be extended to the square shape.

```
void ScatterPlot::Render(
      Microsoft::WRL::ComPtr<ID2D1DeviceContext> context) {
switch(m_NodeShape) {
      // Draw as circle nodes
      case NodeShape::Circle:
            for(int i = 0; i < m_nodeCount; i++) {
                  context->FillEllipse(D2D1::Ellipse(m_points[i],
                        m_HalfNodeSize, m_HalfNodeSize), m_brush);
               }
            if(m_selectedNode != -1) // If a node is selected, render it
larger
                  context->FillEllipse(
                        D2D1::Ellipse(m_points[m_selectedNode],
                        m_HalfNodeSize*2.0f, m_HalfNodeSize*2.0f),
m_brush);
            break;

      // Draw as square nodes
      case NodeShape::Square:
            for(int i = 0; i < m_nodeCount; i++) {
                  context->FillRectangle(D2D1::RectF(m_points[i].x -
                        m_HalfNodeSize,
                  m_points[i].y - m_HalfNodeSize, m_points[i].x +
m_HalfNodeSize,
```

```
                    m_points[i].y + m_HalfNodeSize), m_brush);
                    }
            break;

      // Additional shapes could follow

      default:
            break;
      }

}
```

We want the PickPoint method to execute when the pointer moves at the moment the GraphRenderer class has a PointerMoved method. However, it is only called when the pointer is depressed, either the mouse button is down or the user is sliding his or her finger.

Add a new public member method to the GraphRenderer class called UpdatePointerPosition. I have placed the prototype after the PointerMoved method in the GraphRenderer.h file.

```
      // Capture the pointer movements so the user can pan the chart
      void PointerMoved(Windows::Foundation::Point point);

      // Record pointers x and t value
      void UpdatePointerPosition(Windows::Foundation::Point point);

      // Method for updating time-dependent objects.

      void Update(float timeTotal, float timeDelta);
```

Add two private member variables to the **GraphRenderer** class for recording the pointer's position.

```
      float m_zoomX;     // The amount the x-axis is scaled by
      float m_zoomY;     // The amount the y-axis is scaled by
      float m_pointerX; // X position of pointer in pixels
      float m_pointerY; // Y position of pointer in pixels
      Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;

      Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;
```

Add the body of **UpdatePointerPosition** method to the GraphRenderer.cpp file. I have added it at the end.

```
void GraphRenderer::UpdatePointerPosition(Windows::Foundation::Point point)
{
m_pointerX = point.X;
m_pointerY = point.Y;

}
```

And finally, we need to call the **m_graphVariable->PickPoint** method. We could call this in the **UpdatePosition** method every time the pointer moves, but **PickPoint** is a slow method so we should call it less frequently. I have placed the call in the **GraphRenderer::Update** method inside a new condition, which will cause it to execute once every 32 frames.

```
void GraphRenderer::Update(float timeTotal, float timeDelta) {
static int fpsCounter = -1;  // Start at -1 so frame 0 updates timers
fpsCounter++;
if((fpsCounter & 15) == 0) { // Update every 16 frames
      // Record the times for display in the render method:
      m_timeDelta = timeDelta;
      m_timeTotal = timeTotal;
      }

if((fpsCounter & 31) == 0) {
      ((ScatterPlot*)m_graphVariable)->PickPoint(
            (m_pointerX-m_pan.X)/m_zoomX,
            (-m_pointerY + m_d2dContext->GetSize().height + m_pan.Y)/m_zoomY
            );
      }

}
```

The member variables **m_pointerX** and **m_pointerY** are screen coordinates of the pointer so we have to convert them to the graph's coordinates when we use them as parameters to the **PickPoint** method.

Tip: Avoid using sqrt in tight loops. I have removed the sqrt function (used to properly calculate the Euclidean distances between the pointer and the nodes) from the body of my loop in the previoud codesamplee. We do not need to know the actual distance in the middle of this loop. All we need to know is the index of the node that has the shortest distance. I have calculated the actual distance using the square root only once after the loop. Removing the square root from the loop allows it to execute around 100 times faster.

I have included a threshold in the previous code (50.0f). If the cursor is more than 50.0f units from all the nodes we conclude that it is too far and no node is selected, setting the selected node to -1. If the cursor is within 50.0f of one or more nodes, this function will set the index of the nearest node. If the threshold value is equal to the radius of a collection of circular nodes (the **m_halfNodeSize** member variable in our ScatterPlot class), this method becomes a hit test, and the cursor must be exactly on a node to select it instead of being near.

The next section contains another interesting and flexible method for performing hit tests, this time using geometries. The geometries are capable of real hit testing on a pixel level, not the simple nearest neighbor search I presented previously. The use of geometries is potentially far slower than the manual hit testing we have just examined. The reason is that geometries themselves are complex, device independent structures; they are very flexible but they are also heavy weight.

We have now reached the end of developing our graphing application. The remainder of this book is a discussion of some extra topics and tools which will not be applied to the current application.

Chapter 14: Direct2D Geometry

Geometries allow us to specify shapes for clipping regions, hit testing, and paths for animations. They are device independent resources, and should be created outside of any tight loops. The most basic geometries are those for the primitive shapes, ellipses, rectangles, and rounded rectangles. The information in this chapter is not designed to extend the graphing application we have been working on up to this point. It is an introduction to the syntax and concepts of some more advanced topics. The following code samples could easily be added to our graphing application, but I assumed the project is a new DirectX (XAML) project and the rendering class is called **SimpleTextRenderer**.

Simple Geometries

To render geometries in your project, add these three member variables to your rendering class.

```
	ID2D1RectangleGeometry *m_rectangle;

	ID2D1EllipseGeometry *m_ellipse;

	ID2D1RoundedRectangleGeometry *m_roundedRectangle;
```

Call the **m_d2dFactory** create geometry methods in the device independent resource creation method (**CreateDeviceIndependentResources**() in the SimpleTextRenderer class).

```
m_d2dFactory->CreateRectangleGeometry(
	D2D1::RectF(0, 0, 100, 100),
	&m_rectangle); // 100 × 100 rectangle


m_d2dFactory->CreateEllipseGeometry(
	D2D1::Ellipse(D2D1::Point2F(100, 10), 150, 100),
	&m_ellipse); // 150 × 100 ellipse


m_d2dFactory->CreateRoundedRectangleGeometry(
	D2D1::RoundedRect(
		D2D1::RectF(-100, -100, 100, 100),
		25, 25), &m_roundedRectangle); // 200 × 200 rectangle w/ rounded
corners
```

The creation methods for making the geometry types is almost the same as those creating the corresponding shapes. For instance, the **CreateEllipseGeometry** syntax is very similar to the **m_d2dContext::DrawEllipse** method. To render previously created geometry, call the Render Geometry method of the Context in your render method.

```
// I've assumed the m_blackBrush brush exists!
m_d2dContext->DrawGeometry(
      m_rectangle, m_blackBrush.Get());
m_d2dContext->DrawGeometry(
      m_ellipse, m_blackBrush.Get());
m_d2dContext->DrawGeometry(
      m_roundedRectangle, m_blackBrush.Get());
```

As mentioned previously, geometries are very slow to render (compared to primitives). However, geometries are very flexible and can be used for more than just rendering shapes. For instance, they have many interesting methods like **ComputeLength** and **ComputeArea,** which return the length and area of the shape, and consider the transform matrices as part of the calculation. That is, they consider scaling and skew. They can also be used to build geometry sinks and hit testing with the **FillContainsPoint** method. All of these functions and others make the geometry a very flexible class, but one that, unless you require these particular abilities, is slower for the computer to manipulate and render.

Once you have created geometries, you can render the filled shapes with a solid color brush using the context's **FillGeometry** method. The context's **DrawGeometry** method can be used to render only the outline of the geometry.

Transformed Geometry

Transformed geometry allows us to attach a matrix transformation to a geometry. This is useful for creating shapes that scale and translate, but the stroke thickness (the thickness of the pen) remains the same. A transformed geometry is basically just a regular geometry, but it carries its own transformation matrix. Geometrical shapes rendered with DrawGeometry are transformed using the transformation matrix in the context.

```
m_d2dFactory->CreateTransformedGeometry(
      m_rectangle,      // Pointer to original geometry
      Matrix3x2F::Scale(1.0f, 12.0f), // Tranformation Matrix
      &m_transformed);  // Pointer to our transformed geometry
```

The code example uses the factory to create a transformed geometry with the **m_rectangle** from the start of the chapter. The transformation matrix supplied is a scale of 1.0f along the x-axis and 12.0f along the y-axis, so the shape will be stretched by 12 units. The important thing to note is that stroke will not be stretched when the shape is rendered, only the shape itself.

To render the transformed geometry you can use the context's regular **DrawGeometry** method.

```
m_d2dContext->DrawGeometry(
      m_transformed, m_blackBrush.Get());
```

Tip: In our original line chart, the line was scaled in a strange way when the x-axis and y-axis were scaled by different values. Depending on the angles of the lines, the stroke thickness was also being stretched. This produces an effect that is like a calligraphy pen, and is undesirable for a line chart. It may be better to use a TransformedGeometry for our line chart, since the line can be scaled with a static stroke width.

Complex Geometries

One of the powerful features of geometries is the fact that simple shapes can be combined together to form more complex geometries. The resulting geometries have all the abilities of the original ones, calculating the area, length, and hit testing. To combine several geometries together, use the **CombineWithGeometry** method of the geometries.

To create a custom geometry from a collection of lines, you can use the **PathGeometry** class. This allows us to specify a set of points, connect them with lines, and render the final shape. An ID2D1GeometrySink is an object which is used to describe a path built from lines, arcs, and other geometric line primitives. The path geometry uses an ID2D1GeometrySink to build the shape, and once it is built you can use the Context's **DrawGeometry** or **FillGeometry** methods to render the shape. In the following example, I have set up a collection of random points and have specified that the points create a closed shape. I have indicated that the geometry shape is filled alternatively. First, add a Path Geometry member variable to your rendering class (SimpleTextRenderer.h file).

```
Microsoft::WRL::ComPtr<ID2D1PathGeometry> m_pathGeometry;
```

The path geometry is a device independent resource like the other geometries, so you can specify the shape in the **CreateDeviceIndependentResources** method of your renderer.

```
      // Create or load some data into a points array.
      int count = 25;
      D2D1_POINT_2F * points = new D2D1_POINT_2F[count*2];
      for(int i = 0; i < count; i++) {
```

```
            points[i].x = 50;
            points[i].y = i * 10;
            points[i+count].x = (count * 10)-(i * 10);
            points[i+count].y = 50;
            }
      // Create the path geometry:
      DX::ThrowIfFailed(m_d2dFactory->CreatePathGeometry(&m_pathGeometry));
      // Use the path geometry to create a geometry sink:
      ID2D1GeometrySink *geometrySink;
      DX::ThrowIfFailed(m_pathGeometry->Open(&geometrySink));
      // Set the fill mode:
      geometrySink->SetFillMode(D2D1_FILL_MODE::D2D1_FILL_MODE_ALTERNATE);
      // Set the start point and specify the figure is to be filled
      geometrySink->BeginFigure(points[0], D2D1_FIGURE_BEGIN_FILLED);
      // Add the other points
      for(int i = 1; i < count; i++) {
            // Add a line to the sink connecting the current point to the
last:
            geometrySink->AddLine(points[i+count]);
            geometrySink->AddLine(points[i]);
            }
      // End the figure, connect the final point to first
      geometrySink->EndFigure(D2D1_FIGURE_END_CLOSED);
      // Release the geometry sink from RAM
      geometrySink->Close();
      geometrySink->Release();
      delete[] points;
```

Note: A geometry sink is a temporary geometry building tool. It is not needed once the geometries are built, so I have used a local pointer and released it once I have finished creating the geometry with it.

Finally, the geometry can be rendered in your render method with a call to either **`FillGeometry`** or **`DrawGeometry`**.

```
m_d2dContext->DrawGeometry(m_pathGeometry.Get(), m_blackBrush.Get());
```

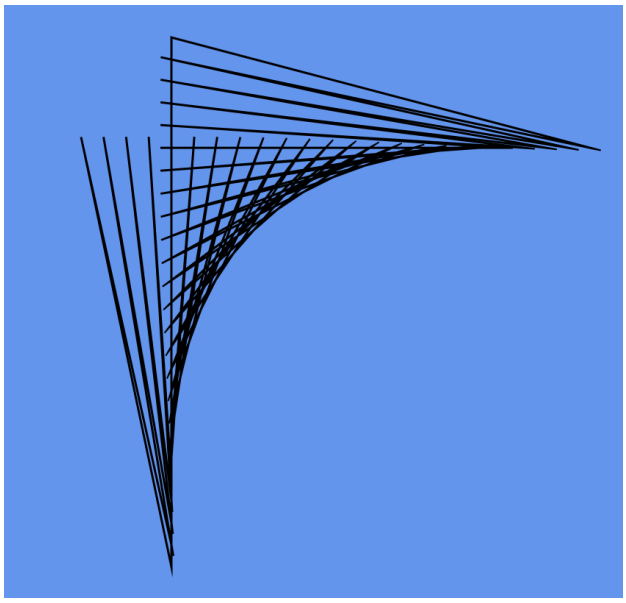


Figure 38: Unfilled Geometry

Figure 38 is the output of the drawn (not filled) geometry. If you render the geometry using the context's **`FillGeometry`** method you will get something like the following.

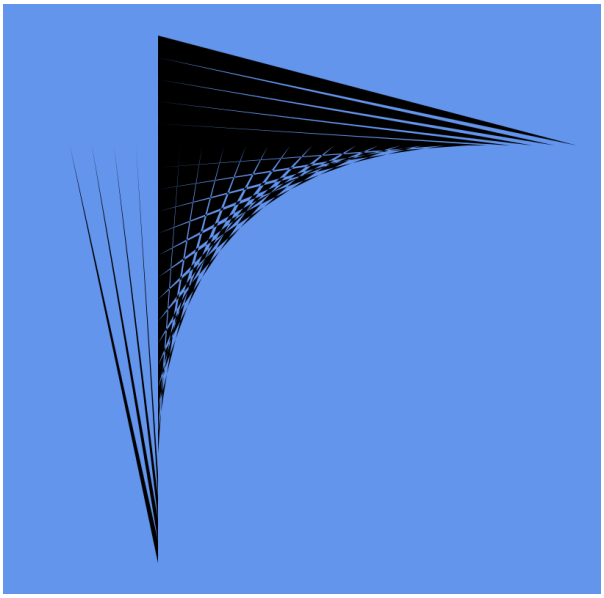


Figure 39: Filled Geometry

Figure 39 is the same geometric figure as the previous one, only every second shape created by the lines in the geometry has been filled in. This is due to the alternate **FILL_MODE** specified in the geometry sink. You can also set the fill mode to **D2D1_FILL_MODE_WINDING**.

The implementation of the fill mode is as follows. For the case of alternate fill mode: for each pixel in the shape, an imaginary line is drawn in an arbitrary direction (it does not matter what the direction is, the result is the same so long as the line is infinite). If this line crosses an odd number of lines in the geometry, the pixel is filled. Otherwise, it is transparent.

In the case of the Winding Fill Mode, for each pixel, an infinite line is drawn in an arbitrary direction. If there are as many lines from the geometry that cross this imaginary line from left to right (from the perspective of the imaginary line), as those that cross from right to left, the shape is not filled. Otherwise, it is.

As mentioned previously, geometries are able to do a lot more than render shapes. For instance, you could calculate the length and area, or perform a hit test with ease with the following code. This code is best called outside the **Render()** method because it is very slow. The **scale** and **translation** matrices used in this code could be member variables of the **SimpleTextRenderer** class.

```
// Compute the area of the geometry
float area;
m_pathGeometry->ComputeArea(scale * translation *
      m_orientationTransform2D, &area);
// Compute the length of the geometry
float length;
m_pathGeometry->ComputeLength(scale * translation *
      m_orientationTransform2D, &length);
// Specify a point for hit test:
D2D1_POINT_2F point = D2D1::Point2F(120.0f, 150.0f);
BOOL result = false;
m_pathGeometry->FillContainsPoint(point, scale *
      translation * m_orientationTransform2D, &result);
```

Tip: In these examples, I have assumed there are translation, scale, and orientation matrices which have transformed the geometry, and these are considered in the calculations. For instance, if the current scale matrix is 3.0f in the x-axis and y-axis, then the length reported by the calculate length method will be three times the original length of the shape. If you want to get the area or length of the shape without the scaling or other transformations, you can use the Matrix3x2F::Identity() as the transform matrix.

Note: The way Direct2D uses points and a geometry sink to create a collection of lines is very similar to the way that Direct3D uses vertices and index buffers. The points are specified in an array, and then lines can be created between any two points using their indices within the array. Make sure you have a good grasp of this concept before you examine the vertex and index buffers; it is simpler in 2-D than it is in 3-D, but it is exactly the same concept.

In the previous example we looked at rendering straight lines. Although we used a geometry sink, we could have used a simplified geometry sink since there are no curves in the path we rendered. If you require adding Bézier curves or arc segments to your path, you can use the **AddBezier**, **AddQuadraticBezier**, and **AddArc** methods of the geometry sink.

In the following example, a set of marching shapes is rendered from a collection of points loaded into the geometry using the **AddBezier** method. This example assumes the code from the last example. For example, there is a member variable called **m_pathGeometry** and a black brush from the SimpleTextRenderer sample, and so on.

```
      // Create or load some data into a points array.
      int count = 50;
      D2D1_POINT_2F * points = new D2D1_POINT_2F[count];
      for(int i = 0; i < count; i++) {
            points[i].x = (i*i);
            points[i].y = (i%2)*(i*10+i*10);
            }
```

Add Bézier curves to the geometry using the points.

```
// Set the fill mode:
geometrySink->SetFillMode(D2D1_FILL_MODE::D2D1_FILL_MODE_ALTERNATE);
// Set the start point and specify the figure is to be filled
geometrySink->BeginFigure(points[0], D2D1_FIGURE_BEGIN_FILLED);
// Add the other points
for(int i = 1; i < count-1; i++) {
      // Create Beziers from the points:
      geometrySink->AddBezier(D2D1::BezierSegment(
            points[i-1],points[i], points[i+1]));
```

```
    }
```

And render the geometry in the render method.

```
    m_d2dContext->FillGeometry(m_pathGeometry.Get(), m_blackBrush.Get());
```

Upon running the application you should see something like Figure 40.

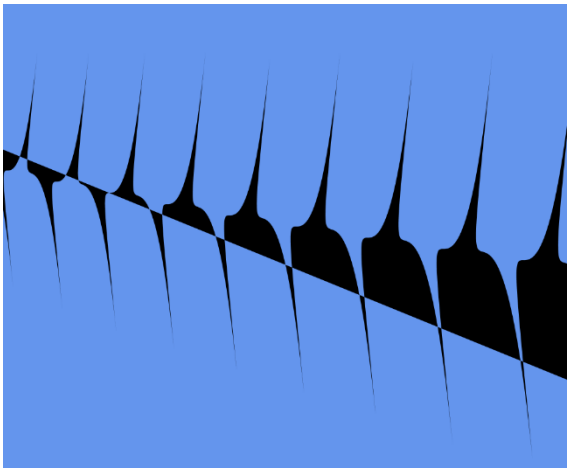


Figure 40: Curved Geometry

Tip: Curves can be a simple way to approximate intermediate values in a line chart. Instead of connecting the nodes in a line chart with straight lines, they can be rendered as curves. Any point between nodes could be approximated by the curves, and in addition, a curved line looks much more appealing than the jagged straight lines that are normally employed to render charts. The actual intermediate values may be no closer to a curve than the straight lines, but (particularly with highly correlated or otherwise well-behaved data sets) the curve may be a closer approximation.

Part 2 Direct3D

The remainder of this book is dedicated to exploring some of the basics of Direct3D. We will only examine some fundamentals of Direct3D here. We will look into the topic with much more depth in the *Direct3D Succinctly* e-book. The Direct3D API is much more powerful than Direct2D. It can obviously render 3-D graphics, but it is also capable of faster 2-D graphics than Direct2D. The Direct2D API is built on top of the Direct3D API. The Direct3D API is closer to the hardware, and the instructions and methods for programming Direct3D give programmers more control over what the GPU does and how it does it. Direct3D is an immense topic and the vast majority of its capabilities have little application to rendering charts.

Chapter 15: Rendering Pipeline

The rendering pipeline, sometimes called graphics pipeline, or simply pipeline, is the set of steps to transform vertices and other structures from a collection of floating point values into 3-D graphics. In order to create graphics, the programmer uses code and data to create a collection of points, triangles, vertices, and so on. They must be transformed and manipulated by the hardware such that a 3-D scene can be displayed on the user's 2-D screen.

Each API and each generation of each API has its own rendering pipeline. The Direct3D rendering Pipeline (although related in some ways) is different from the OpenGL one. The Direct3D 9 pipeline is very different from the Direct3D 11 pipeline. Things have gradually become more flexible as the power of the hardware has scaled. The present pipeline has several stages that are programmable. This means the programmer is able to code instructions for the graphics card directly. Other steps in the pipeline are automated, or at least semiautomated. The programmer can select some settings but cannot dictate what the GPU is to do directly. Each stage of the pipeline takes input from the previous stage and gives input to the next stage. Several of the stages are optional. These are the steps to the current (DirectX 11) rendering pipeline:

1. Input Assembler
2. Vertex Shader
3. Hull Shader
4. Tessellator
5. Domain Shader
6. Geometry Shader
7. Rasterizer
8. Pixel Shader
9. Output Merger

Input Assembler

This stage of the pipeline is where we set up the resources, vertices, colors, and so on—anything we need the GPU to render.

Vertex Shader

This stage is completely programmable. A vertex shader is a small chunk of code that is to be run for every vertex from the previous stage.

Hull Shader, Tesselator, Domain Shader

The next three shaders are new to DirectX 11, and they are for tessellation. Tessellation can increase and decrease the level of detail in a polygon to scale with the hardware running the application. This book will not cover the topic of tessellation.

Geometry Shader

The geometry shader was new to DirectX 10. It is a programmable shader that works on whole shapes, whereas the vertex shader just worked on a single vertex. It is optional and we will not cover geometry shaders in this book.

Rasterizer

The rasterizer takes the vertices from the previous stage and determines which pixels are visible. This is very important because if a pixel is not visible, there is no point in computing its eventual color. The rasterizer clips the scene and performs backface culling. Backface culling is the removal of polygons that are facing away from the camera and are therefore not visible.

Pixel Shader

The pixel shader is another programmable stage in the pipeline. Pixel shaders are executed once for every visible pixel in the scene. There can be significantly more than a million pixels visible in a scene, so it is important to keep pixel shaders very efficient.

Output Merger

This stage puts all the information together and displays the image.

Note: Since this is only a very short introduction to Direct3D, the only shaders we will be using are the vertex shader and the pixel shader. In the follow-up book, Direct3D Succinctly, we will examine the API in greater detail.

Chapter 16: Starting a Direct3D Project

To begin a new Direct3D app, click **File > New Project** in the main menu of Visual Studio. You will be presented with the New Project screen. Click **Visual C++** on the left panel, and then click **Direct3D App** on the middle panel. Give your new project a name. Mine is called Direct3DTesting, as shown in Figure 41.

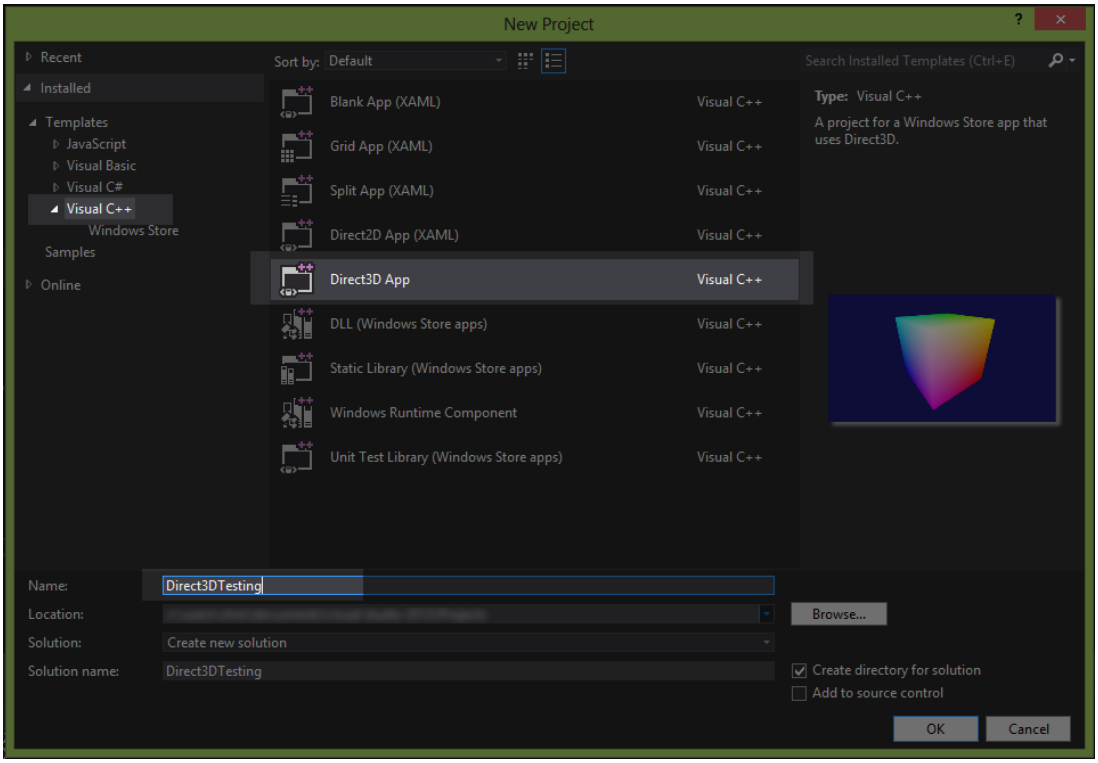


Figure 41: Creating a Direct3D App

Once Visual Studio has created the project you can click **Start Debugging** and you should see a spinning colored cube (as depicted in the preview pane of Figure 41).

Terms and Concepts

3-D Coordinates

We will describe points in our 3-D examples using a standard Cartesian x, y, and z system. Each of the x, y, and z values refers to an point in 3-D space. Each axis can be thought of as an infinite plane perpendicular to the other two axes. They are often summarized as three lines, as shown in Figure 42.

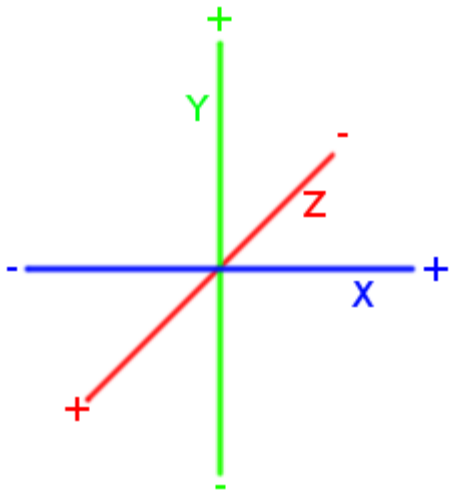


Figure 42: 3-D Axes

We will use the x value of a coordinate to represent how far left or right a point is, the y value to represent how high or low the coordinate is, and the z value to represent how far into or out of the screen the coordinate is. In addition, as an object's x value increases, the object moves rightward. As the object's y value increases, it moves upward. As the object's z value increases, the object moves closer to the camera (or out of the screen).

Vertices

A vertex is a point in 3-D space used to represent the corner of an object, shape, or the end of a line. To specify a 3-D vertex, we need to supply the three coordinates mentioned previously. The coordinates are almost always 32-bit floating point values. Each element (x, y, or z) specifies a position along the dimension, and collectively they describe an exact and unique point in 3-D space.

Note: I will be using x, y, then z as the order for my elements when describing vertices. So something like (9.0f, 8.0f, 5.0f) means 9 along the x-axis, 8 along the y-axis and 5 along the z-axis.

In Figure 43 the yellow ball represents a vertex. In reality, the vertex does not have a physical form; it represents an infinitely small point. The vertex is at position (1.2f, 0.4f, 0.7f). This means it is 1.2 units right of the blue x-axis origin, 0.4 units above the origin of the green y-axis, and 0.7 units away from the origin of the red z-axis.

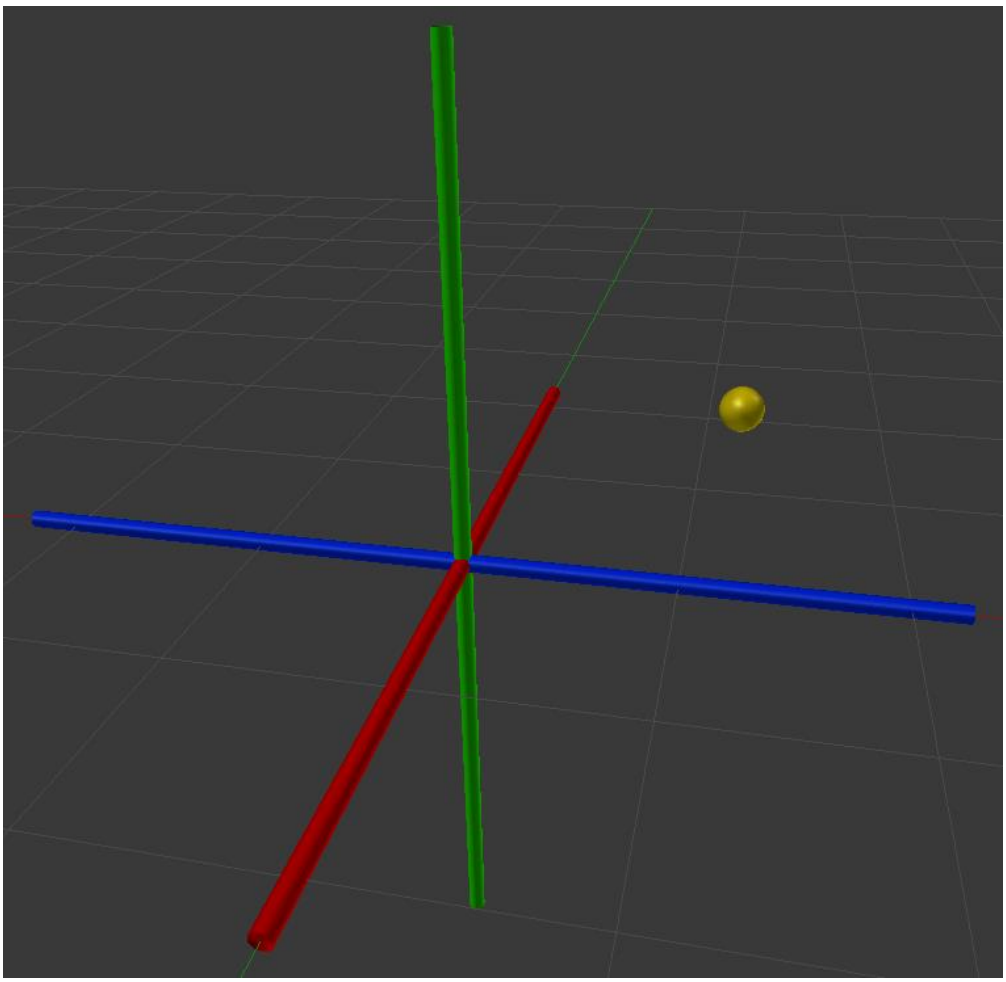


Figure 43: A Point

In Direct3D, vertices are far more flexible than simple points in space. A vertex can carry lighting and coloring information about a point, as well as any other information required. In their most basic form (and the way we will be using them, they consist of a position and color, each described with three or four floating point values.

Lines

A 3-D line can be formed from any two vertices. They are exactly the same as lines in 2-D space, only the points which define the ends each have 3 dimensions. Lines are important in DirectX, because three of them makes a triangle (and as we shall see, 3-D graphics is almost nothing but rendering massive numbers of triangles). Lines also allow us to render 3-D objects as wire frames, so we can easily see the triangles from which the objects are made.

Triangle

Instead of describing 3-D objects with billions of points, objects are usually summarized and are described as a collection of small triangles. The triangles collectively form a mesh, which is a net-like structure. Each triangle is made up of three vertices and three lines. Any three distinct vertices can be used to form a triangle, and if there are enough triangles almost any imaginable shape can be approximated. Modern graphics cards are staggeringly efficient at rendering triangles.

Matrices

Everything from scaling, placement, and the rotation of objects is controlled by matrices in Direct3D. Displaying 3-D graphics consists of multiplying large data sets of vertices and triangles by transformation matrices. There are some matrices which are almost always used; they have a special purpose in 3-D graphics and they have come to be known as the world matrix, the view matrix, and the projection matrix.

- **World Matrix:** When we define objects in 3-D, we usually define them individually with their own origins, scale, and rotation. For instance, when we create a model using a 3-D modeling program, we will probably use the origin and define the model with respect to its own local coordinates. When the model is added to a virtual 3-D world, it will probably not be placed at the origin. Maybe it is moving around in the virtual world. It has its own coordinate system, but when we place it into a scene these coordinates must be translated to world space. That is, the position in the world that the object resides. The world matrix performs this operation.
- **View Matrix:** Once the world matrix has positioned all the objects that may (or may not) need to be rendered, we have to position an eye or camera in the scene. By placing a camera into the scene, we are defining another origin of sorts. The world matrix with all of its objects must be rotated, scaled, and translated to appear as though the camera has been placed at some point among the 3-D objects, and is looking at the virtual world. The view matrix accomplishes this operation.
- **Projection Matrix:** Once the world and its 3-D objects are positioned with respect to some camera, the whole scene can be translated from 3-D coordinates and color vectors to pixels to be displayed on a 2-D monitor. This involves working out which objects appear in front of other objects with respect to the camera, which objects are too near or behind the camera to see, and which objects are too far away. The projection matrix is the final matrix involved in turning a scene into pixels.

Chapter 17: Rendering a Triangle with Direct3D

Getting back to Visual Studio's Direct3D template, we will now look at how to define a triangle. Open the CubeRenderer.cpp file and examine its contents. This file creates the cube, the colors, rotates it, positions the eye, and does almost all of the runtime work. Scroll down to the definition of the **CreateDeviceResources** method (this should around line 15 of the CubeRenderer.cpp file). Firstly, you will see the creation of a couple of shaders (more on these in a moment), but then at line 69 you will see an array of **VertexPositionColor** structures called **cubeVertices**. It is here that the program sets the positions and the colors of each of the cube's eight corners.

```
      auto createCubeTask = (createPSTask &&createVSTask).then([this]()
{
            VertexPositionColor cubeVertices[] = {
                  {XMFLOAT3(-0.5f, -0.5f, -0.5f), XMFLOAT3(0.0f, 0.0f,
0.0f)},
                  {XMFLOAT3(-0.5f, -0.5f,  0.5f), XMFLOAT3(0.0f, 0.0f,
1.0f)},
                  {XMFLOAT3(-0.5f,  0.5f, -0.5f), XMFLOAT3(0.0f, 1.0f,
0.0f)},
```

The **VertexPositionColor** structure type is declared at the top of the CubeRenderer.h file as containing two **XMFLOAT3** types, one for the position and the other for the color of the vertices. The first element of each item in this array is the position of the vertex, and the second element is a normalized RGB color specification.

To render a single triangle instead of the cube, change the positions to the following.

```
auto createCubeTask = (createPSTask && createVSTask).then([this] () {
            VertexPositionColor cubeVertices[] = {
                  {XMFLOAT3(-0.5f, -0.5f, -0.5f), XMFLOAT3(1.0f, 0.0f,
0.0f)},
                  {XMFLOAT3(0.0f, 0.5f,  -0.5f), XMFLOAT3(0.0f, 1.0f, 0.0f)},
                  {XMFLOAT3(0.5f,  -0.5f, -0.5f), XMFLOAT3(0.0f, 0.0f,
1.0f)},
            };
```

These three vertices describe a triangle with red, green, and blue corners. The triangle is 2.0f units away from the camera. The camera is presently being positioned at 1.5f in the z-axis in the **Update** method. Scroll the code down to around line 89, and you will see another local array, this one called **cubeIndices**. This array is composed of unsigned short integers. It is used to initialize an index buffer. Change the values in this array to match the indices of our new triangle.

```
unsigned short cubeIndices[] =      {

      0,1, 2, // A triangle from 3 points

 };
```

You should now be able to run the application and view a beautiful rainbow colored triangle, as shown in Figure 44.

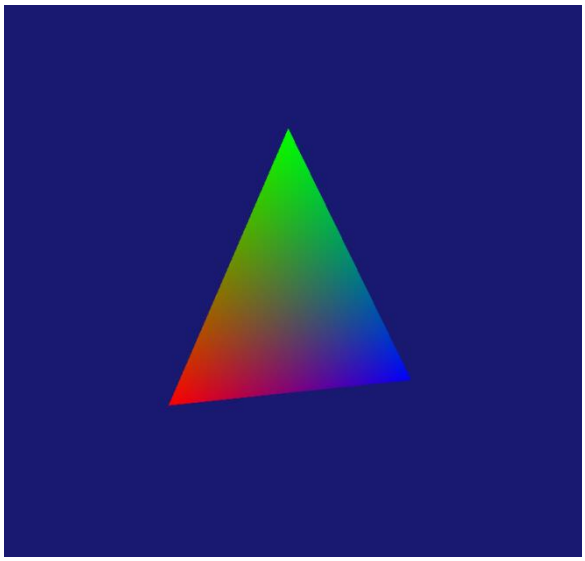


Figure 44: Triangle

You will note that, like the cube, our triangle is spinning about the y-axis. You will also see that when it turns around and the back is facing our camera, Direct3D draws nothing at all. This is the result of backface culling.

Vertex and Index Buffers

In the example code, we described three colored vertices. After the definition of the vertex array, there is a call to the d3dDevice's **CreateBuffer** method. This method takes our array and copies the data to the device (GPU) RAM. Thus, a vertex buffer is a Device Resource.

```
D3D11_SUBRESOURCE_DATA vertexBufferData = {0};

 vertexBufferData.pSysMem = cubeVertices;

 vertexBufferData.SysMemPitch = 0;
```

```
vertexBufferData.SysMemSlicePitch = 0;

CD3D11_BUFFER_DESC vertexBufferDesc(sizeof(cubeVertices),

      D3D11_BIND_VERTEX_BUFFER);

DX::ThrowIfFailed(

      m_d3dDevice->CreateBuffer(

            &vertexBufferDesc,&vertexBufferData,

            &m_vertexBuffer));
```

We do not have direct access to the GPU's RAM. For example, we cannot create a pointer in C++ to an address in GPU RAM, and change the value at will. This is why we first create the array in system RAM, then use the **`CreateBuffer`** method to copy the data to the GPU.

A vertex buffer is an area of memory on the graphics card that is used to store vertices. Vertices are usually created in system RAM by the CPU, and then copied to the graphics card's on board RAM, since the card's on board RAM is usually much faster than system RAM. It is also common to load vertices from a model file created with 3-D modelling software. Once the buffer is copied to the graphics card, it is no longer needed in system RAM, unless it is to be changed by the CPU, reloaded at some point onto the GPU, or both.

An index buffer references the vertices in a vertex buffer. Each of the triangles is made up of three points from the vertex buffer, but the graphics card does not just guess that nearby points are from the same triangle. We have to tell it exactly which points create every one of the triangles we wish to render. We do this by creating an index buffer. Index buffers are integer arrays whose elements specify the triangles by referencing the points in the vertex buffer. Every three integers (unsigned short integers are often used) creates a triangle from any of the three vertices in the vertex buffer. The purpose of separating the vertex buffer and the index buffer is that it avoids duplicate points. The same point can be used by multiple vertices.

Backface Culling

The order in which vertices are defined for a triangle determines which side of the triangle is the front and which is the back. The index buffer describes this order by specifying the sequence of points from the vertex buffer to be used to construct our shapes (this is very similar to the way we used a geometry sink in the Direct2D geometries). We said the first point of the triangle was 0, then 1, then 2 (these are the indices of the vertices in the vertex buffer we just described). It is very important to see that this describes the triangle in a clockwise order of points when viewed from one side only. When viewed from the front, the points are arranged in a clockwise order; when viewed from the back, they are arranged in a counter-clockwise order.

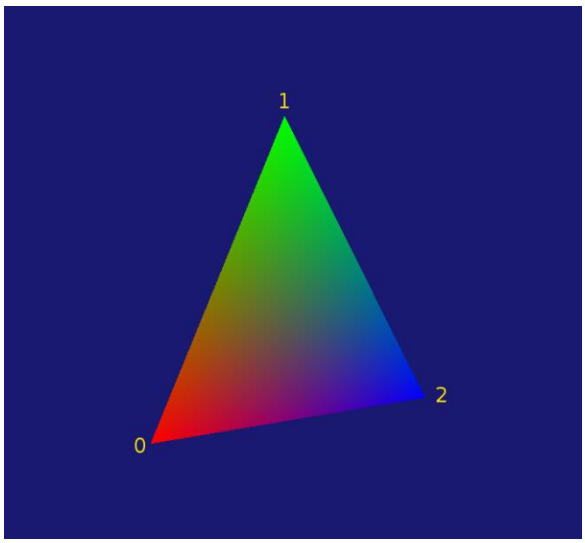


Figure 45: Indices

The triangle spins around the y-axis and from the back we can see straight through it. This is the result of a process called backface culling. Normally a triangle in a 3-D scene will only be viewed from one side. Consider the cube we had originally: it was made up of 12 triangles (two for each side) and the camera was viewing the outside of it. If the camera went into the cube, we would see that DirectX is actually not bothering to draw the other sides of the triangles; it is assuming our cube is solid and we will only ever look at the outside of it. This assumption saves the GPU a lot of processing. You could imagine that roughly half the triangles in a complex 3-D scene (consisting of perhaps hundreds of thousands of triangles) will be facing the insides of 3-D objects, and the GPU doesn't need to render them at all.

DirectX determines the front face of any particular triangle based on the order of the points in the index buffer being clockwise or counterclockwise. The face which is described with the points in a clockwise direction is the front face and is rendered by the GPU. The counterclockwise side is the back face and it is culled, or not rendered.

If we want our triangle to be visible from both sides, we can tell DirectX to render two triangles with the same vertex buffer. One is the front face which uses the three points in the order 0, 1, then 2, and the other is the back face which uses the points in the opposite order, 0, 2, then 1. Note that we need not change the vertex buffer, only the index buffer.

```
unsigned short cubeIndices[] = {
      0,1,2,      // The front face
      0,2,1       // The back face
};
```

Upon running the application you will see the triangle now spins and is visible from both sides. It should be clear at this point that the vertices in the vertex buffer can be used more than once. The indices in the index buffer reference the elements in the vertex buffer and describe the order which they are to be used to create triangles.

Positioning the Eye

The eye (or the viewer of the scene) is positioned in the **Update** method of the **CubeRenderer** class.

```
void CubeRenderer::Update(float timeTotal, float timeDelta) {

      (void) timeDelta; // Unused parameter.

      XMVECTOR eye = XMVectorSet(0.0f, 0.7f, 1.5f, 0.0f);

      XMVECTOR at = XMVectorSet(0.0f, -0.1f, 0.0f, 0.0f);

      XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

      XMStoreFloat4x4(&m_constantBufferData.view,

            XMMatrixTranspose(XMMatrixLookAtRH(eye, at, up)));

      XMStoreFloat4x4(&m_constantBufferData.model,

           XMMatrixTranspose(XMMatrixRotationY(timeTotal *XM_PIDIV4)));

}
```

The three XMVECTORs described here are the position, rotation, and up vector of the eye. The first vector is the eye's position. It is on the x-axis, 0.7 units above the y-axis, and 1.5 units away from the z-axis origin.

The second XMVECTOR describes the point the eye is looking at; it is looking at a spot -0.1 below the y-axis (this happens to be directly at the cube or triangle we described earlier).

The final of the three XMVECTORs is the up vector for the eye; this specifies which direction is to be used as up in relation to the eye. Here, it is 1.0 in the y-axis. That is to say, the direction that our eye considers as upwards is the same as positive values in the y-axis.

Note: The up vector is important, because although we have positioned the eye and described the point that it is looking at, the eye is still free to roll. It can stay in the same place and look at the same pixel, but be upside down. The up vector can be used to specify that the eye is upside down, lying on its side, or the right way up. For instance, an up vector of (0.0f, -1.0f, 0.0f, 0.0f) would mean that the eye is upside down, its top is pointing towards negative values in the y-axis. Setting the up vector to (0.0f, 0.0f, 0.0f) for all elements is meaningless, and will cause a crash. In the specification of the up vector, all negative numbers are read as -1.0, all positive numbers are read as 1.0, and 0.0 is read as 0.0.

Two matrices are stored in the **constantBufferData**, the eye (which is the view member) we have just looked at, and a second model matrix. The model matrix contains the rotation about the y-axis. If you want your triangle to stop rotating, you can replace the multiplication by the rotation matrix with the **Identity** matrix.

```
   XMStoreFloat4x4(&m_constantBufferData.model, XMMatrixIdentity());
```

```
        //XMMatrixTranspose(XMMatrixRotationY(timeTotal * XM_PIDIV4)));
```

This will cause our model to be positioned in the world space exactly as it is described in its own model space. You could also change the y rotation to another type of rotation and examine the effects of animated rotations about the different axes. Remember also that matrix multiplication is cumulative, so you could rotate the triangle about all axes by multiplying the rotation matrices together.

```
    XMStoreFloat4x4(&m_constantBufferData.model,
          XMMatrixTranspose(
                XMMatrixRotationX(timeTotal * XM_PIDIV4)
                *XMMatrixRotationY(timeTotal * XM_PIDIV4)
                *XMMatrixRotationZ(timeTotal * XM_PIDIV4)
                ));
```

Note: The angles in the previous rotation (and elsewhere in Direct3D) are in radians. Radians are a measure of angle linked to the constant pi (π). One radian is equal to (180/π) degrees. 2π radians is the same as a full 360 degrees. Arbitrary angles of rotation can be specified in radians by multiplying 2π radians (the full 360 degrees) by the amount to rotate. For instance, to rotate by 50% (180 degrees) we could use 0.5*(2π), and to rotate by 67.58% we could use 0.6758*(2π). DirectXMath.h has some useful constants (one of which we can see in the following table, XM_PIDIV4).

Constant	Value	Meaning	Degrees
XM_PI	3.141592654	PI	180
XM_2PI	6.283185307	2*PI	360
XM_1DIVPI	0.318309886	1/PI	18.24
XM_1DIV2PI	0.159154943	1/(2*PI)	9.12
XM_PIDIV2	1.570796327	PI/2	90
XM_PIDIV4	0.785398163	PI/4	45

Note: The XMVector is a vector type from the DirectXMath library (the header is DirectXMath.h). This library consists of a collection of helper functions to achieve many standard Math operations. The XMVector is a simple structure with four floating point values or integers, aligned to 16 bytes. The DirectXMath library has optimized methods for dealing with this data type (depending on the hardware, the library uses SIMD extensions to perform operations in parallel). The XMVector is used for many different things, including storing and specifying the positions of vertices and their colors.

Primitive Topologies

A primitive's topology is the basic type of primitive that the GPU renders with a collection of vertices. A collection of vertices can be rendered as points, lines, or triangles by using the **`POINTLIST`**, **`LINELIST`** or **`TRIANGLELIST`** topologies, respectively. The GPU can also connect the adjacent primitives together (so the first is connected to the second, and the second to the third, and so on) by using the **`LINESTRIP`** or the **`TRIANGLESTRIP`**. The following code sample lists some common primitive topologies. This is not a complete list; the complete list is described in the direct3dcommon.h file.

```
D3D11_PRIMITIVE_TOPOLOGY_POINTLIST = D3D_PRIMITIVE_TOPOLOGY_POINTLIST,

 D3D11_PRIMITIVE_TOPOLOGY_LINELIST = D3D_PRIMITIVE_TOPOLOGY_LINELIST,

 D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP= D3D_PRIMITIVE_TOPOLOGY_LINESTRIP,

 D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST,

 D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP =
D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP,
```

The topology is set using the **`IASetPrimitiveTopology`** method of the D3D context. This offers a very quick and easy way to switch between rendering solid shapes (triangles and triangle strips) and rendering wire frames (lines and line strips).

When vertices are rendered as a point list, every vertex will be rendered as a single point or a pixel. The line list renders the points as lines. Every pair of points is used to render a line. The lines are not necessarily connected (depending on the index buffer, they may or may not be connected but they will not be connected automatically).

The line strip topology renders a line list, but it also connects adjacent the lines together. This will create a single long continuous line connecting all the points in the vertex buffer. It is useful for rendering wire frame meshes.

The triangle list takes every group of three points from the points list and renders them as a solid triangle. The triangles are not necessarily connected (again they may be connected using the index buffer but they will not be connected automatically).

The triangle strip is the same as the triangle list, but all of the triangles are connected. Vertices are shared among adjacent triangles.

Chapter 18: Rendering a Height Map

Now we will examine one very common way to render 3-D data known as a height map. Height maps are an important visualization tool for 3-D data sets (data with at least three parametric variables). They are often used in games for terrain generation, as they can be made to look like mountainous terrain. They can be thought of as the 3-D equivalent to the line chart; instead of a single line being rendered to represent the data, a mountainous surface is rendered.

The height map we will render will be a collection of points connected with solid triangles. The x and z values will form a grid or surface, and the y values will be rendered as the heights of elements in the grid. This type of height map could be used if a researcher wants to visualize the effect of two variables (x and z) on a third variable (y).

To create a height map, we can alter the vertices of the cube in the standard Direct3D template. Create a new Direct3D application and open the CubeRenderer.cpp file. Find the line with **`VertexPositionColor cubeVertices[] =`** and replace the definition of the vertices with the following. I have commented out the lines to replace in the following code listing to show where the new code goes.

```
      /*VertexPositionColor cubeVertices[] =
            {
                  {XMFLOAT3(-0.5f, -0.5f, -0.5f), XMFLOAT3(0.0f, 0.0f,
0.0f)},
                  {XMFLOAT3(-0.5f, -0.5f,  0.5f), XMFLOAT3(0.0f, 0.0f,
1.0f)},
                  {XMFLOAT3(-0.5f,  0.5f, -0.5f), XMFLOAT3(0.0f, 1.0f,
0.0f)},
                  {XMFLOAT3(-0.5f,  0.5f,  0.5f), XMFLOAT3(0.0f, 1.0f,
1.0f)},
                  {XMFLOAT3( 0.5f, -0.5f, -0.5f), XMFLOAT3(1.0f, 0.0f,
0.0f)},
                  {XMFLOAT3( 0.5f, -0.5f,  0.5f), XMFLOAT3(1.0f, 0.0f,
1.0f)},
                  {XMFLOAT3( 0.5f,  0.5f, -0.5f), XMFLOAT3(1.0f, 1.0f,
0.0f)},
                  {XMFLOAT3( 0.5f,  0.5f,  0.5f), XMFLOAT3(1.0f, 1.0f,
1.0f)},

            };*/
      const int mapSize = 10;
      VertexPositionColor cubeVertices[mapSize*mapSize];
      float height = 0.0f;
      for(int z = 0; z < mapSize; z++)   {
            for(int x = 0; x < mapSize; x++)   {
                  height = (float)(rand()%100) / 100.0f;
```

```
                cubeVertices[x+(z * mapSize)].pos = XMFLOAT3(x, height, z);
                cubeVertices[x+(z * mapSize)].color = XMFLOAT3(0.0f,
height,
                      0.0f);
                }
           }
```

This code defines a 2-D grid running left to right and into the screen. The y values are the data points we will be plotting. In this example they are random values from 0.0f to 1.0f, but you would usually load these values from a data source. This is our vertex buffer in system RAM.

Now that we have stored the points to render for our height map in the vertex buffer, we need to construct a list of triangles that describes adjacent points. The vertex buffer does not describe any shapes, it is just a list of points. We wish to specify a collection of flat triangles connecting adjacent points, such that when the data is rendered, each of the y values from the nested for loops in the previous code becomes little mountains and crevices. The following code creates an index buffer for this purpose (again, I have commented out the original **cubeIndices** definition, which we are replacing):

```
      /*unsigned short cubeIndices[] =
            {
                  0,2,1, // -x
                  1,2,3,

                  4,5,6, // +x
                  5,7,6,

                  0,1,5, // -y
                  0,5,4,

                  2,6,7, // +y
                  2,7,3,

                  0,4,6, // -z
                  0,6,2,

                  1,3,7, // +z
                  1,7,5,

            };*/

// There's (mapSize-1)*(mapSize-1) squares and 2 triangles per square,

// so 6 points each.

unsigned short cubeIndices[(6*(mapSize-1))*(mapSize - 1)];

int idx = 0; // Index counter to keep track of which square we're defining
```

```
// Step through the points and define the indices that create adjacent
squares

// from them.

for(int z = 0; z < (mapSize-1); z++){

      for(int x = 0; x < (mapSize-1); x++){

            unsigned short farLeftPoint = x + z * mapSize; // Far left point

            unsigned short farRightPoint = farLeftPoint+1; // Far right point

            unsigned short nearLeftPoint = x + (z+1)*mapSize;// Near left
point

            unsigned short nearRightPoint = nearLeftPoint+1; // Near right
point



            // Define 6 points that create 2 triangles that are the square

            cubeIndices[idx++] = farLeftPoint;

            cubeIndices[idx++] = nearRightPoint;

            cubeIndices[idx++] = nearLeftPoint;



            cubeIndices[idx++] = farLeftPoint;

            cubeIndices[idx++] = farRightPoint;

            cubeIndices[idx++] = nearRightPoint;

            }

      }
```

As mentioned previously, it is normal for an index buffer to reference the same vertices more than once to describe adjacent triangles. If two triangles are directly beside each other and they share a line, we need not store six vertices (one for each point of each of the two triangles). We can store four vertices and describe the two adjacent triangles with our index buffer, reusing two of the vertices:

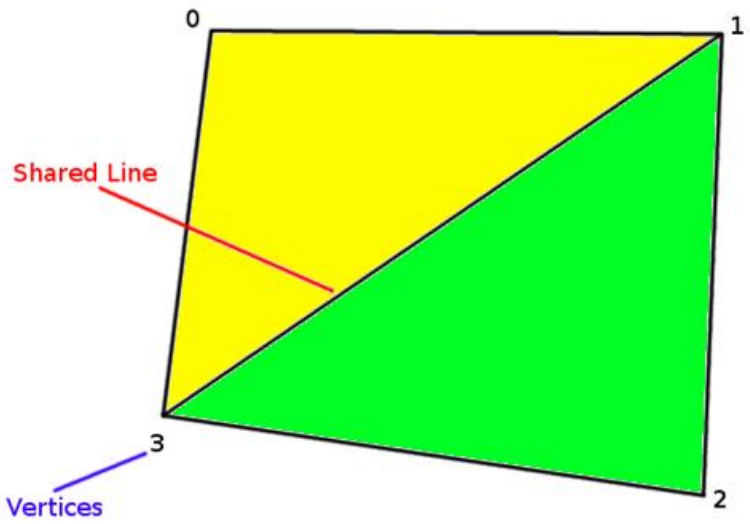


Figure 46: Triangles Forming a Square

In Figure 46, there are two triangles formed by drawing lines from four points. Our vertex buffer holds the four points and our index buffer holds the indices that create the triangles: { 0, 1, 3, 1, 2, 3}. The first three integers in the index buffer describe the lines that create the yellow colored triangle { 0, 1, 3 }. The next three indices describe the lines that make the green colored triangle { 1, 2, 3 }. In this way, vertices 1 and 3 are reused, and memory use and processing time is improved on the GPU.

In this height map code, we are stepping through the points one at a time, defining triangles with each. Two triangles make a square, and we step through to mapSize-1 because the final points on the edges of the map are the right-hand sides of the squares. In other words, there is one less square than the number of points. When you execute the program, you should see a spinning, green height map. Higher values are rendered more green than lower ones.

In the **`Render`** method, there is a call to **`IASetPrimitiveTopology`** which asks the GPU to render the points as triangles. Another good way to render a height map is to use lines. You can change the parameter of this method from **`D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST`** to **`D3D11_PRIMITIVE_TOPOLOGY_LINELIST`** and your height map will be rendered as a collection of colored lines.

Tip: In the previous example, I used local arrays to store the indices and vertices in system memory; with even a moderately sized height map this will very quickly lead to a stack overflow. Parameters local to functions are stored on the stack such that memory allocated for them is automatically cleared when the function returns. If you require a larger height map, you should consider using the heap with the "new" operator.

Chapter 19: Projection Options

The projection of a scene is the method by which the scene is transformed from 3-D data points in RAM to a 2-D projection for display on the monitor. Desktop computers have 2-D monitors for displaying graphics; 3-D graphics are an illusion simulated by projecting a set of 3-D coordinates onto a 2-D surface (the computer monitor).

Perspective Projection

The projection matrix is defined in the **CreateWindowSizeDependentResources** method of the CubeRenderer.cpp file. The default definition is as follows.

```
float aspectRatio = m_windowBounds.Width / m_windowBounds.Height;

      float fovAngleY = 70.0f * XM_PI / 180.0f;

// Note that the m_orientationTransform3D matrix is post-multiplied here

// …

// this transform should not be applied.

      XMStoreFloat4x4(

            &m_constantBufferData.projection,

            XMMatrixTranspose(

                  XMMatrixMultiply(

                        XMMatrixPerspectiveFovRH(

                              fovAngleY,aspectRatio,

                              0.01f,      100.0f      ),

                        XMLoadFloat4x4(&m_orientationTransform3D)

                        )));
```

XMMatrixPerspectiveFovRH

In the previous code we have defined a perspective field of view (FOV). The farther objects are away from the camera, the smaller they appear. Perspective is good for mimicking reality since our own visual system describes distance using a similar algorithm (coupled with binocular depth perception). The matrix used here is the FovRH (Right-Handed Field of View). A detailed discussion of right versus left handed coordinates is provided in the followup book *Direct3D Succinctly*.

Aspect Ratio

This parameter is the aspect ratio of the projection. This will usually be the same as the aspect ratio of the screen as we see here. This is the screen's width divided by its height. If this value is greater than the aspect ratio of the screen, objects will appear much taller when projected; if it is less than the screen, objects will appear squashed.

Field of View (FOV) Angle

This is the angle of the field of view. The parameter expects radians, so the 70 degrees in the definition of this variable is converted to radians by multiplying it by pi/180. This is the angle of visible objects from the camera. For humans it is around 180 degrees, but most of the sides of what we can see is very blurry. In projection matrices, it is conventional to use a value less than 180 degrees. The value 70 means the camera can see things 35 degrees left of its center and 35 degrees right.

Near Clipping Plane

It is conventional to clip objects too near the camera, because if they are rendered they block the rest of the scene. The closer an object is to the camera, the larger it will appear when rendered with perspective. The value given as a near clipping plane is the closest an object can be before it is no longer rendered. Here the value 0.01f is used to mean that anything closer to the camera than 0.01 units should be clipped or not rendered.

Far Clipping Plane

To save processing time, objects which are far from the camera may not need to be rendered. The value given for the far clipping plane is the farthest an object can be from the camera before it is no longer rendered. It is the viewing distance in units. It should be some value greater than the near clipping plane, otherwise the camera would not be able to see anything. For charting applications, the far clipping plane will usually be far enough that all the data is visible at all times.

Orthographic Projection

By default, the Direct3D template presents a cube rendered with perspective. Sometimes it is easier to understand data when it is not rendered in this way. After all, perspective is a distortion effect, and it may obscure the meaning of the data. An orthographic projection is the same as a perspective projection except that objects maintain their apparent size despite their distance from the camera. This looks a little unnatural, but it can be a very clear way to render data. Distant data points will not appear smaller than closer data points, and they can be compared more easily. Distant data is as clear as nearer data, whereas with a perspective projection, distant data is all squashed together approaching a point on the horizon line. See Figure 47 for a depiction of the difference between these projections.

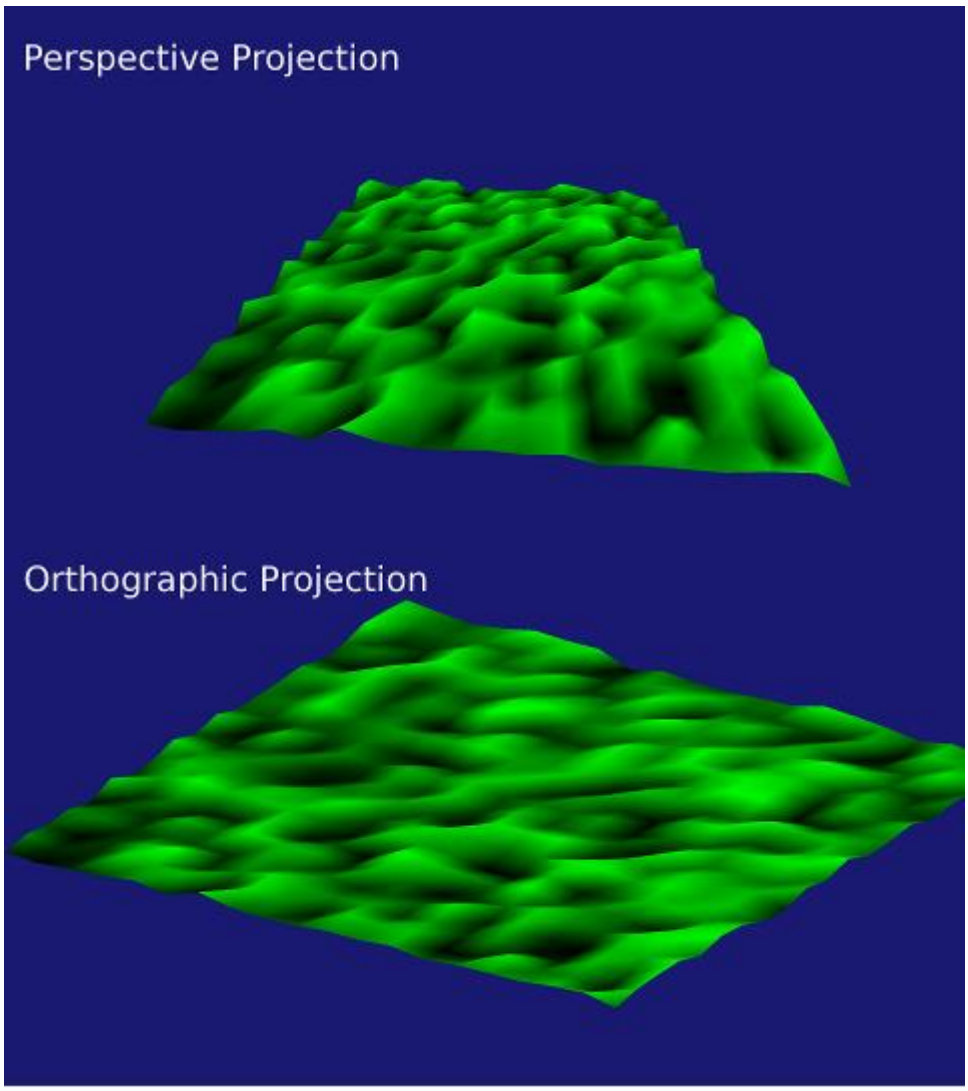


Figure 47: Projection Options

To use an orthographic projection, replace the **`XMMatrixPerspectiveFovRH`** (in the CubeRenderer's **`CreateWindowSizeDependentResources`** method) with the following.

```
    XMMatrixOrthographicRH(
          25.0f,      // Horizontal units visible
          25.0f,      // Vertical units visible
          0.01f,      // Distance to closest renderable point
          500.0f),    // Distance to farthest renderable point
```

ViewWidth

This is number of units visible in the horizontal axis. If, for instance, you have rendered a 20 × 20 height map, you might set this value to 25.0f to make all the data points visible with an additional 2.5 units (half of 5.0) of extra padding on the side.

ViewHeight

This is the number of units visible in the vertical axis. For instance, to view a height map with data from 0.0f to 20.0f, you might set this value to 25.0f to allow all the data to be visible with a little extra for padding.

Near and Far Clipping Planes

These values have an identical meaning in an orthographic projection to the perspective projection. They represent the nearest and farthest points an object can be from the camera and still be rendered.

Direct3D Scatter Plot

As a final example of rendering data using Direct3D, we will examine rendering a collection of unconnected triangles (like the nodes in a scatter plot). The basic scatter plot we examined earlier was fairly good at rendering thousands of nodes, but when the count reaches tens or hundreds of thousands of nodes, this technique is no longer useful; it is simply too slow. The problem is the rendering loop. This loop is performed by the CPU. One node is drawn to the render target per iteration and this is a severe bottleneck. The CPU is running through the loop and relaying the drawing instructions to the GPU. Remember that Direct2D was an extra layer of abstraction on top of Direct3D. It is far more efficient to exclude the CPU and loop altogether, and render nodes in parallel using only the GPU.

Drawing many thousands of triangular nodes is extremely efficient in Direct3D. Instead of rendering the nodes as circles using a “for loop,” we could simply store them as a collection of triangles on the GPU and render them directly with Direct3D. The CPU does not need to perform any calculations once the nodes are loaded onto the graphics card. The performance gained by using the GPU to render triangles, instead of rendering circles with Direct2D, is vast. For instance, the original Direct2D scatter plot could render around 10,000 nodes comfortably on the machine I am writing with. This new method can easily render 1,000,000 at a steady 60 frames per second. This machine has only a moderately powerful GPU (nVidia GT 430) and it has no trouble. It becomes very choppy only at around 10,000,000 nodes.

For this, we will use a new Direct3D cube template. Open a new Direct3D application, and change the DeviceResources loading method to the following.

```
void CubeRenderer::CreateDeviceResources() {

	Direct3DBase::CreateDeviceResources();



	auto loadVSTask = DX::ReadDataAsync("SimpleVertexShader.cso");

	auto loadPSTask = DX::ReadDataAsync("SimplePixelShader.cso");

	auto createVSTask = loadVSTask.then([this](Platform::Array<byte>^ fileData) {

		DX::ThrowIfFailed(

			m_d3dDevice->CreateVertexShader(

				fileData->Data,fileData->Length,  nullptr,
```

```
                &m_vertexShader));

        const D3D11_INPUT_ELEMENT_DESC vertexDesc[] = {
                { "POSITION", 0,
                DXGI_FORMAT_R32G32B32_FLOAT,0,0,D3D11_INPUT_PER_VERTEX_DATA, 0 },
                { "COLOR",    0,
        DXGI_FORMAT_R32G32B32_FLOAT,0,12,D3D11_INPUT_PER_VERTEX_DATA, 0 },
        };

        DX::ThrowIfFailed(
                m_d3dDevice->CreateInputLayout(
                        vertexDesc,ARRAYSIZE(vertexDesc),
                        fileData->Data,fileData->Length,&m_inputLayout
                        ));
});

auto createPSTask = loadPSTask.then([this](Platform::Array<byte>^ fileData) {
        DX::ThrowIfFailed(
                m_d3dDevice->CreatePixelShader(
                        fileData->Data,fileData->Length,
                        nullptr,&m_pixelShader
                        ));

        CD3D11_BUFFER_DESC
                constantBufferDesc(sizeof(ModelViewProjectionConstantBuffer),
                D3D11_BIND_CONSTANT_BUFFER);
        DX::ThrowIfFailed(
                m_d3dDevice->CreateBuffer(
                        &constantBufferDesc,nullptr,
                        &m_constantBuffer)
```

```
                );
        });


        auto createCubeTask = (createPSTask && createVSTask).then([this] () {
                int count = 100;
                float x, y, r, g, b;
                m_CubeVertices = new VertexPositionColor[count * 3];
                float sze = 0.01f;
                float z = 0.0f;
                for(int i = 0; i < count; i++){
                        x = (float(rand() % 5000)/5000.0f)-0.5f;
                        y = (float(rand() % 5000)/5000.0f)-0.5f;


                        r = (float(rand() % 10)/10.0f);
                        g = (float(rand() % 10)/10.0f);
                        b = (float(rand() % 10)/10.0f);


                        m_CubeVertices[(i*3)+0].pos = XMFLOAT3(x, y, z);
                        m_CubeVertices[(i*3)+0].color = XMFLOAT3(r, g, b);


                        m_CubeVertices[(i*3)+1].pos = XMFLOAT3(x-sze, y+sze, z);
                        m_CubeVertices[(i*3)+1].color = XMFLOAT3(r, g, b);


                        m_CubeVertices[(i*3)+2].pos = XMFLOAT3(x+sze, y+sze, z);
                        m_CubeVertices[(i*3)+2].color = XMFLOAT3(r, g, b);


                        z+= 0.0000001f;
                        }


                D3D11_SUBRESOURCE_DATA vertexBufferData = {0};
                vertexBufferData.pSysMem = m_CubeVertices;
```

```
        vertexBufferData.SysMemPitch = 0;
        vertexBufferData.SysMemSlicePitch = 0;
        CD3D11_BUFFER_DESC vertexBufferDesc(
              count*2*3*sizeof(XMFLOAT3), D3D11_BIND_VERTEX_BUFFER);
        DX::ThrowIfFailed(
              m_d3dDevice->CreateBuffer(
                    &vertexBufferDesc,&vertexBufferData,
                    &m_vertexBuffer)
              );

        m_CubeIndices = new unsigned short[count * 3];
        for(int i = 0; i < count * 3; i++){
              m_CubeIndices[i] = i;
              }

        m_indexCount = count * 3;

        D3D11_SUBRESOURCE_DATA indexBufferData = {0};
        indexBufferData.pSysMem = m_CubeIndices;
        indexBufferData.SysMemPitch = 0;
        indexBufferData.SysMemSlicePitch = 0;
        CD3D11_BUFFER_DESC indexBufferDesc(2*count * 3, D3D11_BIND_INDEX_BUFFER);
        DX::ThrowIfFailed(
              m_d3dDevice->CreateBuffer( &indexBufferDesc,
                    &indexBufferData,&m_indexBuffer
                    ));
  });

  createCubeTask.then([this] () {
        m_loadingComplete = true;
```

```
        });

}
```

Add the m_CubeVertices and m_CubeIndices arrays to the CubeRenderer.h file.

```
        VertexPositionColor *m_CubeVertices;

        unsigned short *m_CubeIndices;
```

Most of this code should be familiar. I have created a vertex buffer, an index buffer, and rendered the data. This is a demonstration of rendering 100 colored triangles. The number can be increased by changing the line that contains count = 100. You should find that you can increase the number of triangles far in excess of the number of triangles you could possibly plot with Direct2D geometry.

Tip: I have added a slowly incrementing z value for each of the triangles in the code sample. This was added so that no two triangles would lie exactly in the same position. If two triangles lie in exactly the same position, the GPU will sometimes render one first, and at other times it will render the other triangle first. This leads to an unpleasant flickering artifact. By creating each triangle on a slightly different z plane, we eliminate this flickering.

Conclusion

We have looked at printing, programming for the new WinRT devices, hit testing, rendering 2-D and 3-D primitives, and many other aspects of basic DirectX. The adventure has just begun. This book will shortly be followed by another, which will be aimed at DirectX graphics for computer and video games, and will pick up where this one left off.

Direct2D is a simple and efficient API for 2-D graphics. It is capable of rendering almost any data set, even those consisting of thousands of nodes. There are many aspects of Direct2D which we did not cover such as sprites, animation, bitmap atlases, and so on. These items will be covered in the next book.

The final part of this book was a small dip into the fascinating waters of Direct3D. The follow-up book will describe much more about Direct3D. It is an extremely powerful API. We have glanced over, and taken for granted, almost all the boilerplate code. I have completely and purposely neglected shaders and the High Level Shader Language, which is arguably the most powerful aspect of Direct3D. Even with our tiny exploration of Direct3D, we have seen that the GPU is capable of rendering data extremely efficiently, especially if it is rendered as points, lines, or triangles. In this introduction we have examined the API on a rather low level; we built our models (the height map and our triangles) manually.

I hope you have enjoyed reading this book, and seeing some of the aspects of this gigantic and very powerful API in action. I hope you will enjoy the follow-up book also. I have enjoyed writing, and DirectX only gets better the more we explore. This is not the end of using the API for representing data. The next book, although it will concentrate on graphics for games, will introduce many concepts which can easily be used to render data with much more flexibility than what we have seen so far.

Appendix A: Microsoft Limited Public License

MICROSOFT LIMITED PUBLIC LICENSE version 1.1

This license governs use of code marked as “sample” or “example” available on this web site without a license agreement, as provided under the section above titled “NOTICE SPECIFIC TO SOFTWARE AVAILABLE ON THIS WEB SITE.” If you use such code (the “software”), you accept this license. If you do not accept the license, do not use the software.

1. Definitions

The terms “reproduce,” “reproduction,” “derivative works,” and “distribution” have the same meaning here as under U.S. copyright law.

A “contribution” is the original software, or any additions or changes to the software.

A “contributor” is any person that distributes its contribution under this license.

“Licensed patents” are a contributor’s patent claims that read directly on its contribution.

2. Grant of Rights

(A) Copyright Grant - Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free copyright license to reproduce its contribution, prepare derivative works of its contribution, and distribute its contribution or any derivative works that you create.

(B) Patent Grant - Subject to the terms of this license, including the license conditions and limitations in section 3, each contributor grants you a non-exclusive, worldwide, royalty-free license under its licensed patents to make, have made, use, sell, offer for sale, import, and/or otherwise dispose of its contribution in the software or derivative works of the contribution in the software.

3. Conditions and Limitations

(A) No Trademark License- This license does not grant you rights to use any contributors’ name, logo, or trademarks.

(B) If you bring a patent claim against any contributor over patents that you claim are infringed by the software, your patent license from such contributor to the software ends automatically.

(C) If you distribute any portion of the software, you must retain all copyright, patent, trademark, and attribution notices that are present in the software.

(D) If you distribute any portion of the software in source code form, you may do so only under this license by including a complete copy of this license with your distribution. If you distribute any portion of the software in compiled or object code form, you may only do so under a license that complies with this license.

(E) The software is licensed “as-is.” You bear the risk of using it. The contributors give no express warranties, guarantees or conditions. You may have additional consumer rights under your local laws which this license cannot change. To the extent permitted under your local laws, the contributors exclude the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

(F) Platform Limitation - The licenses granted in sections 2(A) and 2(B) extend only to the software or derivative works that you create that run directly on a Microsoft Windows operating system product, Microsoft run-time technology (such as the .NET Framework or Silverlight), or Microsoft application platform (such as Microsoft Office or Microsoft Dynamics).

Appendix B: DirectXPage.xaml Class Listing

The events and methods to handle the background color changing and other aspects of the Direct2D (XAML) app should be removed from this class, since we are going to remove these methods from the SimpleTextRenderer class. At the top of these class files is a reference to the namespace PrinterApplication; this must be changed to match your application's namespace (I have highlighted the line in green).

DirectXPage.xaml.h

```
//
// DirectXPage.xaml.h
// Declaration of the DirectXPage.xaml class.
//
#pragma once
#include "DirectXPage.g.h"
#include "SimpleTextRenderer.h"
#include "BasicTimer.h"
namespace PrinterApplication {
    [Windows::Foundation::Metadata::WebHostHidden]
      public ref class DirectXPage sealed{
      public:
            DirectXPage();
      private:
            void OnPointerMoved(Platform::Object^ sender,
                  Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
            void OnPointerReleased(Platform::Object^ sender,
                  Windows::UI::Xaml::Input::PointerRoutedEventArgs^ args);
            void OnWindowSizeChanged(Windows::UI::Core::CoreWindow^ sender,
                  Windows::UI::Core::WindowSizeChangedEventArgs^ args);
            void OnLogicalDpiChanged(Platform::Object^ sender);
            void OnOrientationChanged(Platform::Object^ sender);
```

```
            void OnDisplayContentsInvalidated(Platform::Object^ sender);
            void OnRendering(Object^ sender, Object^ args);
            Windows::Foundation::EventRegistrationToken m_eventToken;
            SimpleTextRenderer^ m_renderer;
            bool m_renderNeeded;
            BasicTimer^ m_timer;
            Windows::Graphics::Printing::PrintManager^ m_printManager;
      internal:
            // Print task requested event handler method
            void SetPrintTask(_In_
      Windows::Graphics::Printing::PrintManager^ sender,
            _In_
      Windows::Graphics::Printing::PrintTaskRequestedEventArgs^ args);
      };
}
```

```
//
// DirectXPage.xaml.cpp
// Implementation of the DirectXPage.xaml class.
//
#include "pch.h"
#include "DirectXPage.xaml.h"
#include "DocSource.h"
using namespace PrinterApplication;
using namespace Platform;
using namespace Windows::Foundation;
using namespace Windows::Foundation::Collections;
```

```
using namespace Windows::Graphics::Display;
using namespace Windows::UI::Input;
using namespace Windows::UI::Core;
using namespace Windows::UI::Xaml;
using namespace Windows::UI::Xaml::Controls;
using namespace Windows::UI::Xaml::Controls::Primitives;
using namespace Windows::UI::Xaml::Data;
using namespace Windows::UI::Xaml::Input;
using namespace Windows::UI::Xaml::Media;
using namespace Windows::UI::Xaml::Navigation;
using namespace Windows::Graphics::Printing;
DirectXPage::DirectXPage() :m_renderNeeded(true) {
      InitializeComponent();
      m_renderer = ref new SimpleTextRenderer();
      m_renderer->Initialize(
            Window::Current->CoreWindow,
            SwapChainPanel,
            DisplayProperties::LogicalDpi
            );
      Window::Current->CoreWindow->SizeChanged +=
            ref new TypedEventHandler<CoreWindow^,
            WindowSizeChangedEventArgs^>(this,
&DirectXPage::OnWindowSizeChanged);
      DisplayProperties::LogicalDpiChanged +=
            ref new DisplayPropertiesEventHandler(this,
                  &DirectXPage::OnLogicalDpiChanged);
      DisplayProperties::OrientationChanged +=
            ref new DisplayPropertiesEventHandler(this,
                  &DirectXPage::OnOrientationChanged);
```

```
      DisplayProperties::DisplayContentsInvalidated +=
            ref new DisplayPropertiesEventHandler(this,
                  &DirectXPage::OnDisplayContentsInvalidated);
      m_eventToken = CompositionTarget::Rendering::add(
            ref new EventHandler<Object^>(this, &DirectXPage::OnRendering));
      m_timer = ref new BasicTimer();
      // Grab the print manager for the current view
      m_printManager =
            Windows::Graphics::Printing::PrintManager::GetForCurrentView();
      // Add an event handler to capture when the user requests a print task
      m_printManager->PrintTaskRequested +=
            ref new TypedEventHandler<PrintManager^,
                  PrintTaskRequestedEventArgs^>(this,
                        &DirectXPage::SetPrintTask);
}


void DirectXPage::OnPointerMoved(Object^ sender, PointerRoutedEventArgs^
args) {
      m_renderNeeded = true;
}


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


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

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

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

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

void DirectXPage::OnRendering(Object^ sender, Object^ args) {
	if (m_renderNeeded) {
		m_timer->Update();
		m_renderer->Update(m_timer->Total, m_timer->Delta);
		m_renderer->Render();
		m_renderNeeded = false;
	}
}

void DirectXPage::SetPrintTask(_In_ PrintManager^ sender,
	_In_ PrintTaskRequestedEventArgs^ args) {
// Create a new source requested handler
PrintTaskSourceRequestedHandler^ sourceRequestedHandler = ref new
```

```
      PrintTaskSourceRequestedHandler(
      [this](PrintTaskSourceRequestedArgs^ args)-> void {
            Microsoft::WRL::ComPtr<CDocumentSource> documentSource;
            DX::ThrowIfFailed (
                  Microsoft::WRL::MakeAndInitialize<CDocumentSource>(
                        &documentSource,
                        reinterpret_cast<IUnknown*>(m_renderer)));
                  // Cast the document to an object
                  IPrintDocumentSource^ objSource(

      reinterpret_cast<IPrintDocumentSource^>(documentSource.Get())
                  );
            args->SetSource(objSource);
            });
// Create the print task
PrintTask^ printTask = args->Request->CreatePrintTask(L"Direct 2D Printing
Example", sourceRequestedHandler);
}
```

Appendix C: CDocSource Class Code Listing

The following code is from the "Direct2Dapp printing sample (Windows 8.1)" available from Microsoft. I have altered the formatting to better fit the pages of this book; this has led to far more difficult to read code. This code is included here for convenience, and the original (which you can download with the Microsoft Windows 8 samples) is far more readable. All comments are Microsoft's. I have only altered the formatting as mentioned. They refer to the sample from which the code was downloaded.

The class name in the sample code as provided by Microsoft is D2DPageRenderer. For our printing application this should be changed to "SimpleTextRenderer". I have highlighted the line in green in the code that follows. The main renderer class in our code is called SimpleTextRenderer so you must also change all references to the PageRenderer class to SimpleTextRenderer. I have highlighted these lines in red.

DocSource.h listing

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF

//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO

//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A

//// PARTICULAR PURPOSE.

////

//// Copyright (c) Microsoft Corporation. All rights reserved

#pragma once

#include <windows.graphics.printing.h>

#include <printpreview.h>

#include <documentsource.h>

#include "D2DPageRenderer.h"

class CDocumentSource : public
Microsoft::WRL::RuntimeClass<Microsoft::WRL::RuntimeClassFlags<Microsoft::WRL::
WinRtClassicComMix>,

      ABI::Windows::Graphics::Printing::IPrintDocumentSource,

      IPrintDocumentPageSource,

      IPrintPreviewPageCollection>

{

private:
```

```
    InspectableClass(L"Windows.Graphics.Printing.IPrintDocumentSource",
        BaseTrust);


public:
    HRESULT RuntimeClassInitialize(
        _In_ IUnknown* pageRenderer) {
        HRESULT hr = (pageRenderer != nullptr) ? S_OK : E_INVALIDARG;
        if (SUCCEEDED(hr)){
            m_paginateCalled = false;
            m_totalPages     = 1;
            m_height         = 0.f;
            m_width          = 0.f;


            // Cast d2dRender back to PageRenderer object.
            m_renderer = reinterpret_cast<PageRenderer^>(pageRenderer);
        }


        return hr;
    }


    //
    // classic COM interface IDocumentPageSource methods
    //
    IFACEMETHODIMP
    GetPreviewPageCollection(
        _In_  IPrintDocumentPackageTarget*  docPackageTarget,
        _Out_ IPrintPreviewPageCollection** docPageCollection
        );
```

```
    IFACEMETHODIMP
    MakeDocument(
        _In_ IInspectable*                docOptions,
        _In_ IPrintDocumentPackageTarget*   docPackageTarget
        );
    //
    // classic COM interface IPrintPreviewPageCollection methods
    //
    IFACEMETHODIMP
    Paginate(
        _In_   uint32           currentJobPage,
        _In_   IInspectable*    docOptions);
    IFACEMETHODIMP
    MakePage(
        _In_ uint32 desiredJobPage,
        _In_ float  width,
        _In_ float  height);
private:
    float TransformedPageSize(
        _In_  float                       desiredWidth,
        _In_  float                       desiredHeight,
        _Out_ Windows::Foundation::Size*    previewSize);

    uint32                                          m_totalPages;
    bool                                            m_paginateCalled;
    float                                           m_height;
    float                                           m_width;
    D2D1_RECT_F                                     m_imageableRect;
```

```
    PageRenderer^                                           m_renderer;
    Microsoft::WRL::ComPtr<IPrintPreviewDxgiPackageTarget>
m_dxgiPreviewTarget;

};
```

DocSource.cpp listing

```
//// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF

//// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO

//// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A

//// PARTICULAR PURPOSE.

////

//// Copyright (c) Microsoft Corporation. All rights reserved

#include "pch.h"

#include "docsource.h"

using namespace Microsoft::WRL;

using namespace Windows::Graphics::Printing;

#pragma region IDocumentPageSource Methods

IFACEMETHODIMP

CDocumentSource::GetPreviewPageCollection(

    _In_  IPrintDocumentPackageTarget*  docPackageTarget,

    _Out_ IPrintPreviewPageCollection** docPageCollection

    )

{

    HRESULT hr = (docPackageTarget != nullptr) ? S_OK : E_INVALIDARG;

    // Get for IPrintPreviewDxgiPackageTarget interface.

    if (SUCCEEDED(hr)){

        hr = docPackageTarget->GetPackageTarget(

            ID_PREVIEWPACKAGETARGET_DXGI,
```

```
            IID_PPV_ARGS(&m_dxgiPreviewTarget));
    }
    ComPtr<IPrintPreviewPageCollection> pageCollection;
    if (SUCCEEDED(hr)){
        ComPtr<CDocumentSource> docSource(this);
        hr = docSource.As<IPrintPreviewPageCollection>(&pageCollection);
    }

    if (SUCCEEDED(hr)){
        hr = pageCollection.CopyTo(docPageCollection);
    }

    return hr;
}
IFACEMETHODIMP
CDocumentSource::MakeDocument(
    _In_ IInspectable*                docOptions,
    _In_ IPrintDocumentPackageTarget* docPackageTarget)
{
    if (docOptions == nullptr || docPackageTarget == nullptr){
        return E_INVALIDARG;
    }
   // Get print settings from PrintTaskOptions for printing,
   // such as page description,
    // which contains page size, imageable area, DPI.
    // User can obtain other print settings in the same way, such as
ColorMode,
    // NumberOfCopies, etc., which are not shown in this sample.
    PrintTaskOptions^ option =
reinterpret_cast<PrintTaskOptions^>(docOptions);
```

```
    // Get the description of the first page.
    PrintPageDescription pageDesc = option->GetPageDescription(1);
    // Create a print control properties and set DPI from
PrintPageDescription.
    D2D1_PRINT_CONTROL_PROPERTIES printControlProperties;
    // DPI for rasterization of all unsupported D2D commands or options
   printControlProperties.rasterDPI  = (float)(min(pageDesc.DpiX,
            pageDesc.DpiY));
    // Color space for vector graphics in D2D print control.
    printControlProperties.colorSpace = D2D1_COLOR_SPACE_SRGB;
   // Subset for used glyphs, send and discard font resource after every
   // five pages
    printControlProperties.fontSubset = D2D1_PRINT_FONT_SUBSET_MODE_DEFAULT;
    HRESULT hr = S_OK;
    try {
        // Create a new print control linked to the package target.
        m_renderer-
>CreatePrintControl(docPackageTarget,&printControlProperties);


        // Calculate imageable area and page size from PrintPageDescription.
        D2D1_RECT_F imageableRect = D2D1::RectF(
            pageDesc.ImageableRect.X, pageDesc.ImageableRect.Y,
            pageDesc.ImageableRect.X + pageDesc.ImageableRect.Width,
            pageDesc.ImageableRect.Y + pageDesc.ImageableRect.Height
            );
        D2D1_SIZE_F pageSize = D2D1::SizeF(pageDesc.PageSize.Width,
       pageDesc.PageSize.Height);
        // Loop to add page command list to d2d print control.
        for (uint32 pageNum = 1; pageNum <= m_totalPages; ++pageNum){
```

```
            m_renderer->PrintPage(pageNum, imageableRect, pageSize,
                nullptr // If a page-level print ticket is not specified
here, the
                        //package print ticket is applied for each page.
                );
        }
    }
    catch (Platform::Exception^ e){
        hr = e->HResult;
        if (hr == D2DERR_RECREATE_TARGET){
        // In case of device lost, the whole print job will be aborted,
      // and we should recover
        // so that the device is ready when used again. At the same time,
      // we should propagate this error to the Modern Print Dialog.
        m_renderer->HandleDeviceLost();
        }
    }
    // Make sure to close d2d print control even if AddPage fails.
    HRESULT hrClose = m_renderer->ClosePrintControl();
    if (SUCCEEDED(hr)){
        hr = hrClose;
    }
    return hr;
}

#pragma endregion IDocumentPageSource Methods
#pragma region IPrintPreviewPageCollection Methods
IFACEMETHODIMP CDocumentSource::Paginate(
    _In_    uint32          currentJobPage,
```

```
    _In_   IInspectable*    docOptions
    )
{
    HRESULT hr = (docOptions != nullptr) ? S_OK : E_INVALIDARG;
    if (SUCCEEDED(hr)) {
        // Get print settings from PrintTaskOptions for preview, such as page
// description, which contains page size, imageable area, DPI.
        // User can obtain other print settings in the same way, such
      // as ColorMode,
// NumberOfCopies, etc., which are not shown in this sample.
        PrintTaskOptions^ option =
            reinterpret_cast<PrintTaskOptions^>(docOptions);
        PrintPageDescription pageDesc =
            option->GetPageDescription(currentJobPage);
        hr = m_dxgiPreviewTarget->InvalidatePreview();
        // Set the total page number.
        if (SUCCEEDED(hr)){
            hr = m_dxgiPreviewTarget->SetJobPageCount(
                  PageCountType::FinalPageCount, m_totalPages);
        }
        if (SUCCEEDED(hr)) {
            m_width = pageDesc.PageSize.Width;
            m_height = pageDesc.PageSize.Height;
            m_imageableRect = D2D1::RectF(
                pageDesc.ImageableRect.X,
                pageDesc.ImageableRect.Y,
                pageDesc.ImageableRect.X + pageDesc.ImageableRect.Width,
                pageDesc.ImageableRect.Y + pageDesc.ImageableRect.Height
```

```
                );
            // Now we are ready to let MakePage to be called.
            m_paginateCalled = true;
        }
    }
    return hr;
}


// Here, desiredWidth/desiredHeight is the desired size of preview surface
// by print mananger in system. The final size of the preview surface must
// have the same proportion as that of the desired width/height.
// In this sample, we just use it as preview size and return the scale
variant
// for surface drawing.
// The size here is in DIPs.
Float CDocumentSource::TransformedPageSize(
    _In_  float                         desiredWidth,
    _In_  float                         desiredHeight,
    _Out_ Windows::Foundation::Size*    previewSize
    ){
    float scale = 1.0f;
    if (desiredWidth > 0 && desiredHeight > 0) {
        previewSize->Width  = desiredWidth;
        previewSize->Height = desiredHeight;
        scale = m_width / desiredWidth;
    }
    else {
        previewSize->Width = 0;
        previewSize->Height = 0;
```

```
    }
    return scale;
}

// This sample only acts upon orientation setting for an example.
// The orientation is read from the user selection in the Print Experience
// and is then used to reflow the content in a different way.
IFACEMETHODIMP CDocumentSource::MakePage(
    _In_ uint32 desiredJobPage,
    _In_ float  width,
    _In_ float  height){
    HRESULT hr = (width > 0 && height > 0) ? S_OK : E_INVALIDARG;
   // When desiredJobPage is JOB_PAGE_APPLICATION_DEFINED, it means a new
   // preview begins. If the implementation here is by an async way,
   // for example, queue MakePage calls for preview, app needs to clean
   // resources for previous preview before next.
    // In this sample, we will reset page number if Paginate() has been
called.
    if (desiredJobPage == JOB_PAGE_APPLICATION_DEFINED && m_paginateCalled){
        desiredJobPage = 1;
    }
    if (SUCCEEDED(hr) && m_paginateCalled){
        // Calculate the size of preview surface, according to desired width
and
// height.
        Windows::Foundation::Size previewSize;
        float scale = TransformedPageSize(width, height, &previewSize);
        try {
            m_renderer->DrawPreviewSurface(
```

```
                previewSize.Width, previewSize.Height, scale,
m_imageableRect,

                desiredJobPage,m_dxgiPreviewTarget.Get());

        }

        catch (Platform::Exception^ e) {

            hr = e->HResult;

            if (hr == D2DERR_RECREATE_TARGET){

// In case of device lost, we should recover so that the device is

      // ready to render the next preview page when requested. At the same
time,

      // we should propagate this error to the Modern Print Dialog.

            m_renderer->HandleDeviceLost();

            }

        }

    }

    return hr;

}


#pragma region IPrintPreviewPageCollection Methods
```

Appendix D: Code Listing for SimpleTextRenderer Printing

Following is a listing for the altered simple text renderer class based on building an application that requires printing. Most of this code comes from the Microsoft Windows 8 Samples. I have changed the formatting to better fit the pages of this book, but it is far less readable than the original, available from Microsoft.

SimpleTextRenderer.h

```
#pragma once

#include "DirectXBase.h"

#include <PrintPreview.h>

enum class DrawTypes { Rendering, Preview,    Printing    };


// RAII (Resource Acquisition Is Initialization) class for manually

// acquiring/releasing the D2D lock.

class D2DFactoryLock {

public:

D2DFactoryLock(_In_ ID2D1Factory* d2dFactory) {

      DX::ThrowIfFailed(

            d2dFactory->QueryInterface(IID_PPV_ARGS(&m_d2dMultithread))

            );

      m_d2dMultithread->Enter();

      }

~D2DFactoryLock() {

      m_d2dMultithread->Leave();

      }

private:

      Microsoft::WRL::ComPtr<ID2D1Multithread> m_d2dMultithread;

};
```

```
// Renders or prints drawings using Direct2D
ref class SimpleTextRenderer sealed : public DirectXBase {
public:
      SimpleTextRenderer();
      // DirectXBase methods.
      virtual void CreateDeviceIndependentResources() override;
      virtual void CreateDeviceResources() override;
      virtual void CreateWindowSizeDependentResources() override;
      virtual void Render() override;
      void Update(float timeTotal, float timeDelta);
internal:
      // These two methods return immutable resources shared amongst contexts
      SimpleTextRenderer(
            _In_ D2D1_RECT_F targetBox,
            _In_ ID2D1DeviceContext* d2dContext,
            _In_ DrawTypes type,
            _In_ SimpleTextRenderer^ myParent
            );
    void UpdateTargetBox(_In_ D2D1_RECT_F& targetBox);
    void Draw(_In_ float scale);
      void CreatePrintControl(_In_  IPrintDocumentPackageTarget*
docPackageTarget,
                  _In_  D2D1_PRINT_CONTROL_PROPERTIES*
printControlProperties);
      HRESULT ClosePrintControl();

      void DrawPreviewSurface(_In_  float width, _In_  float height, _In_
float scale,
            _In_  D2D1_RECT_F contentBox, _In_  uint32 desiredJobPage,
```

```
            _In_  IPrintPreviewDxgiPackageTarget* previewTarget);

      void PrintPage(_In_ uint32 pageNumber, _In_ D2D1_RECT_F imageableArea,

            _In_ D2D1_SIZE_F pageSize, _In_opt_ IStream*
pagePrintTicketStream);

private:

      Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;

      bool m_renderNeeded;

      // The main print control

      Microsoft::WRL::ComPtr<ID2D1PrintControl> m_d2dPrintControl;

    float m_margin;      // The margin size is in DIPs.

    D2D1_RECT_F m_targetBox; // Region to format for.

    DrawTypes m_type; // Record of context type, screen/preview/printing

};
```

SimpleTextRenderer.cpp

```
#include "pch.h"

#include "SimpleTextRenderer.h"


using namespace D2D1;

using namespace DirectX;

using namespace Microsoft::WRL;

using namespace Windows::Foundation;

using namespace Windows::Foundation::Collections;

using namespace Windows::UI::Core;

SimpleTextRenderer::SimpleTextRenderer() :

      m_renderNeeded(true),

      m_type(DrawTypes::Rendering),// By default the context is rendering to
screen
```

```
	m_margin(96.0f)				// Default margin size
{}

SimpleTextRenderer::SimpleTextRenderer(_In_ D2D1_RECT_F targetBox,
	_In_ ID2D1DeviceContext* d2dContext, _In_ DrawTypes type,
	_In_ SimpleTextRenderer^ myParent) {
m_margin = 96.0f;
m_d2dContext = d2dContext;
m_type = type;
UpdateTargetBox(targetBox);
DX::ThrowIfFailed(
	d2dContext->CreateSolidColorBrush(
	D2D1::ColorF(D2D1::ColorF::Black),
	&m_blackBrush));
}

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

void SimpleTextRenderer::CreateDeviceResources() {
	DirectXBase::CreateDeviceResources();
	DX::ThrowIfFailed(
		m_d2dContext->CreateSolidColorBrush(
			ColorF(ColorF::Black),
			&m_blackBrush
			)
		);
}
```

```
void SimpleTextRenderer::CreateWindowSizeDependentResources() {
	DirectXBase::CreateWindowSizeDependentResources();
	// Make the target box the whole screen
	D2D1_SIZE_F size = m_d2dContext->GetSize();
	m_targetBox = D2D1::RectF(0, 0, size.width, size.height);
}

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

void SimpleTextRenderer::Render() {
m_d2dContext->BeginDraw();
// Microsoft check that the current screen is not snapped here but we will
// assume it is not.
// Render page context.
Draw(1.0f);
// We ignore D2DERR_RECREATE_TARGET here. This error indicates that the
device
// is lost. It will be handled during the next call to Present.
HRESULT hr = m_d2dContext->EndDraw();
if (hr != D2DERR_RECREATE_TARGET) DX::ThrowIfFailed(hr);
// We are accessing D3D resources directly in Present() without D2D's
knowledge,
// so we must manually acquire the D2D factory lock.
//
// Note: it's absolutely critical that the factory lock be released upon
```

```
// exiting this function, or else the entire app will deadlock. This is

// ensured via the following RAII class.

D2DFactoryLock factoryLock(m_d2dFactory.Get());

Present();

}

void SimpleTextRenderer::UpdateTargetBox(_In_ D2D1_RECT_F& targetBox) {

    m_targetBox = targetBox;

}

// Draws the scene to a rendering device context or a printing device
context.

void SimpleTextRenderer::Draw(_In_ float scale)

{

if (m_type == DrawTypes::Rendering) { // Clear to CornFlowerBlue if rendering
to screen

      m_d2dContext->Clear(D2D1::ColorF(D2D1::ColorF::CornflowerBlue));    }

else if (m_type == DrawTypes::Preview) { // Do not clear when printing

      m_d2dContext->Clear(D2D1::ColorF(D2D1::ColorF::White));    }


// We use scale matrix to shrink the image when previewing.

// On-screen rendering or printing scale is 1.f.

m_d2dContext->SetTransform(D2D1::Matrix3x2F(1/scale, 0, 0, 1/scale, 0, 0));

// This is where the drawing of the chart takes place. Below is a pattern

// of colored circles as an example:

D2D1_ELLIPSE ell;

for(float y = 0; y < 16; y++) {

      for(float x = 0; x < 16; x++) {

            ell = D2D1::Ellipse(D2D1::Point2F(x * 50.0f, y * 50.0f), 100.0f,
100.0f);

            m_blackBrush->SetColor(D2D1::ColorF(

                  (x * 16.0f) / 256.0f,
```

```
                (y * 16.0f) / 256.0f,

                ((x + y) * 8.0f) / 256.0f));

            m_d2dContext->DrawEllipse(ell, m_blackBrush.Get());

            }

      }

}



HRESULT SimpleTextRenderer::ClosePrintControl() {

return (m_d2dPrintControl == nullptr) ? S_OK : m_d2dPrintControl->Close();

}



void SimpleTextRenderer::CreatePrintControl(_In_
IPrintDocumentPackageTarget* docPackageTarget, _In_
D2D1_PRINT_CONTROL_PROPERTIES* printControlProperties){

// Explicitly release existing D2D print control.

m_d2dPrintControl = nullptr;



DX::ThrowIfFailed(

      m_d2dDevice->CreatePrintControl(m_wicFactory.Get(),

            docPackageTarget, printControlProperties, &m_d2dPrintControl));

}

void SimpleTextRenderer::DrawPreviewSurface(_In_  float width, _In_  float
height,

      _In_  float scale, _In_  D2D1_RECT_F contentBox, _In_  uint32
desiredJobPage,

      _In_  IPrintPreviewDxgiPackageTarget* previewTarget)

{

D2DFactoryLock factoryLock(m_d2dFactory.Get());

CD3D11_TEXTURE2D_DESC textureDesc(DXGI_FORMAT_B8G8R8A8_UNORM,
static_cast<uint32>(ceil(width  * m_dpi / 96)),
```

```
      static_cast<uint32>(ceil(height * m_dpi / 96)), 1, 1,

            D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE);

ComPtr<ID3D11Texture2D> texture;

DX::ThrowIfFailed(m_d3dDevice->CreateTexture2D(&textureDesc, nullptr,
&texture));



// Create a preview DXGI surface with given size.

ComPtr<IDXGISurface> dxgiSurface;

DX::ThrowIfFailed(texture.As<IDXGISurface>(&dxgiSurface));



// Create a new D2D device context for rendering the preview surface. D2D

// device contexts are stateful, and hence a unique device context must be

// used on each thread.

ComPtr<ID2D1DeviceContext> d2dContext;

DX::ThrowIfFailed(m_d2dDevice->CreateDeviceContext(

      D2D1_DEVICE_CONTEXT_OPTIONS_NONE, &d2dContext));

// Update DPI for preview surface as well.

d2dContext->SetDpi(m_dpi, m_dpi);

// Recommend using the screen DPI for better fidelity and better performance
in the print preview.

D2D1_BITMAP_PROPERTIES1 bitmapProperties = D2D1::BitmapProperties1(

      D2D1_BITMAP_OPTIONS_TARGET | D2D1_BITMAP_OPTIONS_CANNOT_DRAW,

      D2D1::PixelFormat(DXGI_FORMAT_B8G8R8A8_UNORM,
D2D1_ALPHA_MODE_PREMULTIPLIED));



// Create surface bitmap on which page content is drawn.

ComPtr<ID2D1Bitmap1> d2dSurfaceBitmap;

DX::ThrowIfFailed(

      d2dContext->CreateBitmapFromDxgiSurface(dxgiSurface.Get(),

      &bitmapProperties, &d2dSurfaceBitmap));
```

```
d2dContext->SetTarget(d2dSurfaceBitmap.Get());

// Create and initialize the page renderer context for preview.

SimpleTextRenderer^ previewTextRenderer = ref new
SimpleTextRenderer(contentBox, d2dContext.Get(), DrawTypes::Preview, this);

d2dContext->BeginDraw();

// Draw page content on the preview surface.

previewTextRenderer->Draw(scale);

// The document source handles D2DERR_RECREATETARGET, so it's okay to throw
here.

DX::ThrowIfFailed(d2dContext->EndDraw());

// Must pass the same DPI used to create the DXGI surface for the correct
print

// preview.

DX::ThrowIfFailed(

      previewTarget->DrawPage(desiredJobPage,dxgiSurface.Get(), m_dpi, m_dpi)

      );

}


void SimpleTextRenderer::PrintPage(_In_ uint32 pageNumber,

      _In_ D2D1_RECT_F imageableArea, _In_ D2D1_SIZE_F pageSize,

      _In_opt_ IStream* pagePrintTicketStream)

{

// Create a new D2D device context for generating the print command list.

// D2D device contexts are stateful, and hence a unique device context must

// be used on each thread.

ComPtr<ID2D1DeviceContext> d2dContext;

DX::ThrowIfFailed(

      m_d2dDevice->CreateDeviceContext(D2D1_DEVICE_CONTEXT_OPTIONS_NONE,
&d2dContext)

      );
```

```
ComPtr<ID2D1CommandList> printCommandList;

DX::ThrowIfFailed(

      d2dContext->CreateCommandList(&printCommandList)

      );

d2dContext->SetTarget(printCommandList.Get());

// Create and initialize the page renderer context for print.

// In this case, we want to use the bitmap source that already has

// the color context embedded in it. Thus, we pass NULL for the

// color context parameter.

SimpleTextRenderer^ printPageRendererContext = ref new
SimpleTextRenderer(imageableArea, d2dContext.Get(),

      DrawTypes::Printing, this);

d2dContext->BeginDraw();

// Draw page content on a command list.

// 1.0f below indicates that the printing content does not scale.

// "DrawTypes::Printing" below indicates it is a printing case.

printPageRendererContext->Draw(1.0f);

// The document source handles D2DERR_RECREATETARGET, so it's okay to throw
here.

DX::ThrowIfFailed(d2dContext->EndDraw());



DX::ThrowIfFailed(printCommandList->Close());



DX::ThrowIfFailed(m_d2dPrintControl->AddPage(printCommandList.Get(),
pageSize,

            pagePrintTicketStream));

}
```