The Starter Code Documentation

Overview

The following pages describe the starter code and some of the architectural decisions I've made.

Where possible, it will also discuss other options you may have as well. There are plenty of places where I could have made different decisions. In general, I wanted to keep the overall codebase as straightforward as possible rather than building a sophisticated but convoluted architecture.

Why do we need starter code?

If you've done any OpenGL programming, you've probably used a library like GLUT or GLFW to handle creating a window and initializing OpenGL itself. While there are a few similar libraries for Direct3D, I decided a while ago to just do it myself and show you the code. I think it's important to understand what goes into initializing our graphics API and creating the window (the guts of many of our programs).

Much of what you'll see in the starter code is simply what libraries like GLUT or GLFW do for you. In fact, you could take the non-Direct3D-specific parts of this starter code and use them to create an OpenGL application instead.

An advanced game engine might even initialize either OpenGL or Direct3D based on a config file or command line parameters. However, this is *outside the scope of this course*, as it usually requires wrapping your entire graphics system in an abstraction layer or, worse, doing everything twice.

Basic Architecture

The core architecture consists of a few code files and a bit of inheritance:

- **Main.cpp** – Our program's entry point
- **DXCore class** – Handles window creation, OS-level messages and DX initialization & shut-down
- **Game class** – Inherits from DXCore. Contains application-specific code.

The idea is that you, a student in this class, shouldn't have to edit DXCore at all (unless you're fundamentally altering how it works, which you're absolutely allowed to do as the course goes on). DXCore will take care of the basics that just about any DirectX application would need, and the Game class (plus any custom classes) will be where you actually build cool, custom stuff.

Win32 Programming

A lot of the "windows" stuff you'll see in the code might look a little foreign. It's the Windows API (often referred to as Win32), and it hasn't changed all that much in the last few decades, hence its archaic feel. Most of it is confined to DXCore. I don't expect you to interact with it directly or even understand it all.

The Project Itself

I started by creating an empty C++ project in Visual Studio 2022 (using the template called "Empty Project"). I went into the project properties and changed the Linker > System > Subsystem option from "Console" to "Windows". Since I already had the Windows SDK installed, I didn't have to alter anything else or include any other libraries. Note that the project is, by default, set to use Unicode characters, which will impact the way we use strings, especially in regards to any Windows API function calls.

Wide Character Strings

Throughout the starter code, you'll see strings used for things like the text of our window's title bar, file paths for loading external files (precompiled shaders for now), etc. In all of these cases, the project is using "wide character" strings instead of standard strings. So, what exactly is a "wide character"?

The "char" data type is 8 bits, meaning it can store 256 unique values. This is fine for standard ASCII characters, but for extended character sets like Unicode that have more than 256 possibilities, a larger data type is necessary. This is where "wide characters" come in. "Wide" in this case refers to the data type literally taking up more room in memory to support larger ranges of possible characters.

In C++, the data type for a single wide character is `wchar_t`. It holds a single character but takes up more memory than a regular "char". Taking this one step further, a C-style wide-character string would then be a null-terminated array of `wchar_t` values. Any functions that exist for standard characters generally also have a "wide" version.

C++ also contains a special string class for wide-character strings – `std::wstring` – which can be found in the standard <string> header. It's virtually identical to `std::string`, but uses wide characters. You'll see `wstring` used in the starter code and in many of my demos and sample code.

*Note that there is **no** implicit or explicit cast between regular and wide character strings!*

Lastly, to denote to the compiler that a string literal is meant to be interpreted as a wide character string instead of a standard character string, you must prepend the string data with a capital L character.

```
// Regular and wide literal examples
char    letter     = 'a';
wchar_t letterWide = L'a';

char*    name     = "Chris";
wchar_t* nameWide = L"Chris";

std::string  path     = "C:\Data\file.txt";
std::wstring pathWide = L"C:\Data\file.txt";
```

Do we need to use wide characters? Generally, when programming for Windows, especially with the Windows API, the answer is *yes*. It isn't necessarily difficult, but is easy to forget if you haven't used wide characters before. The compiler will let you know if you mess it up.

Main.cpp

This is where our program entry point resides. It turns on memory leak detection (in debug builds), creates our one & only Game class object, initializes it and enters our game loop. That's about it.

However, since we're creating a graphical Windows application, there's a bunch of Operating System (OS) stuff that our application must handle. It mostly gets dealt with by DXCore, but it's worth talking about it a little here.

Entry point: Main vs. WinMain

The first thing that may be a little different from what you're used to is that we don't have a main() function. Instead, we use WinMain(). This is a convention used by Win32 graphical applications that don't require a console window, although we can (and do) allocate our own console later if we want. Behind the scenes, the compiler replaces WinMain() with a main() containing some necessary code.

One advantage of using WinMain() is that it comes with some useful OS-level parameters that we're going to need anyway when setting up a window and Direct3D. Of course, we could just use main() and get that information another way, but WinMain() is the standard way of starting this kind of app.

If you're interested, you can find more information about WinMain on MSDN.

Application Handles

In Windows, every application (and window, and UI element, etc.) has a unique identifier assigned by the operating system. These are called *handles*, and we use them when we want to reference or create a particular object controlled by the OS, such as creating a window for our application to use. Under the hood, handles are just integers, though the C++ data type we use is HINSTANCE.

Since we're using WinMain(), it's implied that we're going to be creating a window for our program, so the first parameter is our application's OS-level handle. Super handy!

S_OK and HRESULTs

You'll see the constant **S_OK** in WinMain(). The "S" stands for *success*, and the "OK" stands for, you guessed it, *okay*. This is a predefined constant often seen with Win32 programming that means "Everything is fine, no errors occurred!" Really, it's just the value zero under the hood.

Most Win32 functions have a return value of type HRESULT, which is a 32-bit value whose bits are divided into three sections: severity, facility and error codes. There are some predefined constants and macros in the windows API that simplify checking these return values. S_OK is one of these constants.

DXCore Header

Before we get into the guts of DXCore, it's worth looking at a few novel pieces of the header file.

Including Library (.lib) Files

You'll notice the following lines of code near the top of DXCore.h:

```
#pragma comment(lib, "d3d11.lib")
#pragma comment(lib, "dxgi.lib")
```

What's that **pragma comment** thing? Often, we need to denote that we'll be using a library that isn't normally included in our project. This can be done through Visual Studio's project settings, or by including this particular preprocessor directive directly in our code.

One advantage to putting it directly into the code is that it's obvious: you can tell which file is required without having to search through settings. If you're more comfortable including it in your project settings, you could do that instead.

Windows Message Handling

You'll also notice two static members in the public section of DXCore.h:

- DXCoreInstance
- WindowProc()

WindowProc

WindowProc() is a necessary part of creating a graphical windows application, regardless of whether we're making a game or just a regular application.

Under the hood, every graphical Windows application is sent messages by the operating system. These messages tell the program things like "the mouse just moved", "your window is being resized" or "the user clicked the close button". Our program *must* deal with these, by either capturing them for our own use or by simply telling the operating system "Okay, thanks for letting me know".

If an OS-level message isn't dealt with properly, Windows thinks our program is frozen. This is how Windows detects that a program "isn't responding" – it's literally not responding to the messages being sent to it. This can happen when a program is stuck inside an infinite loop, for example.

So what makes WindowProc() special? It's the function within *our* program that the operating system will call when it wants to send us a message. However, it **cannot** be a member function! This means it has to be a global or static function, which makes it harder to talk to our non-global Game object.

DXCoreInstance

A quick and dirty solution to the problem above is to create a static pointer to our game object, called DXCoreInstance, and set it during the object's constructor. This will let WindowProc() talk to our object.

If this makes you cringe a little, that's probably good. Like I said, it's a quick fix without adding a bunch of extra complexity to the starter code. It might be a little better to use a singleton or similar design pattern, but for our purposes this will work fine.

ComPtr

Direct3D resources, such as the **device**, **context** and **swapChain** members of DXCore, are a special kind of object. These objects abide by the COM, or Component Object Model, interface. They can't be instantiated, nor deleted, directly. Instead, they're created for us by the Direct3D API. They also have a Release() method to notify the API when the object is no longer needed (and can be safely deleted).

While it's fine to use raw C++ pointers with Direct3D objects, I suggest using smart pointers instead. Because of the aforementioned restrictions, however, Direct3D objects can't be used with standard smart pointers like `shared_ptr`. Luckily, Microsoft provides a smart pointer specifically for COM objects: the **ComPtr**. You'll see these used in several places throughout the starter code.

DXCore Class

This class is the core of our little engine. It's going to create a window for our application, initialize Direct3D, handle incoming Windows messages and perform a few other odds & ends. Descriptions of major areas and methods can be found below.

Includes

Aside from the obvious DXCore.h, we're including the following two headers:

- **dxgi1_5.h:** For querying graphics device feature support
- **WindowsX.h**: Needed for a few of the Windows messages we plan on handling
- **sstream**: Used for quick and dirty debug string manipulation in UpdateTitleBarStats()

Static Members

Here we *define* DXCoreInstance, since we've only *declared* it over in the header file.

Directly below that, you'll find the WindowProc() function. It forwards its parameters to our object's ProcessMessage() function, which will actually check the message and do something useful if necessary.

DXCore Constructor

The constructor for DXCore takes a few parameters related to the window we'll be creating:

- **hInstance**: The applications handle, which is needed to create a window
- **titleBarText**: The text you want in the title bar (note: This is a *wstring*, not a string)
- **windowWidth**: The width of the window's *client* area (not including title bar & borders)
- **windowHeight**: The height of the window's *client* area
- **vsync**: Should the framerate be synced to the monitor's refresh rate?
- **debugTitleBarStats**: Should extra stats be shown in the title bar, such as FPS and DX version?

Most member variables are set using an initializer list before the constructor body rather than assignment statements within the constructor itself. This was a coding style choice on my part. Within the constructor, the global DXCoreInstance pointer it set.

Lastly, it queries the *performance counter* to find out its frequency. The *performance counter* is a way of retrieving very high-resolution time stamps (less than 1 microsecond), allowing us to calculate very precise timing. The frequency may be different on each computer, but it is fixed at system boot time, so we only need to grab it once at start up.

DXCore Destructor

If we *were* using raw C++ pointers to Direct3D objects, we'd need to call Release() on them instead of deleting them directly, since they are created by the Direct3D API itself. (Each has an internal reference count, and once the last reference is released the object is automatically cleaned up.) However, since the starter code is using ComPtr's for Direct3D objects, they'll be cleaned up properly automatically. We do, however, need to clean up the Input Manager's singleton here.

DXCore::InitWindow

This method creates the window for our application, which must exist before we can initialize DirectX.

The first thing you'll notice is a big struct of type WNDCLASS being filled out. This will hold the basic settings for the window class we're trying to register with the operating system, and is passed to RegisterClass(). We can't create a window on the screen unless we register one of these special "window classes" that describes it. Some options at this step include which function to call when messages come in, the icon for the application and default cursor to show.

If that succeeds, we then create a rectangle of the appropriate window size and adjust it to include borders & title bar. We also calculate the centered X and Y location to be used in the next step.

Next up, we actually call CreateWindow(), passing in the name of the window class that was registered, some information about the size of the window and its title bar text.

As long as the call to CreateWindow() succeeded, we show the window (finally making it visible), initialize the Input Manager (which needs the window handle) and return S_OK to indicate success.

DXCore::InitDirect3D

This method initializes Direct3D, which requires a window to already exist. In addition to simply "activating" Direct3D, this method creates the basic resources we'll need to do anything useful with Direct3D, including creating the *back buffer* (where we draw) and a *depth buffer* (for basic occlusion).

First though, a preprocessor directive determines if we're in Debug mode (rather than Release mode), and sets the D3D11_CREATE_DEVICE_DEBUG flag. This allows Direct3D to do extra error checking and provide (somewhat) useful error messages at run time. Since this can slow things down, we only want this to happen while in debug mode.

Next, we check to see if the graphics drivers support an unlocked framerate. While most do by default, certain laptops and displays with variable refresh rates require specific parameters for this.

It then creates the description of the *Swap Chain*, which is an object that automatically handles *double buffering* (sometimes called *page flipping*) for us. Options here include the size of the buffer (the image to be presented to the user) which should match the window size, as well as the color format and whether or not the application should be windowed.

Now that we have that description, we call D3D11CreateDeviceAndSwapChain(), which initializes the three objects we use to communicate with the Direct3D API:

- Device – Creates resources on the GPU
- DeviceContext – Issues drawing commands and state changes (called *context* for short)
- SwapChain – Handles presenting our final frames to the user

One useful feature of Direct3D 11 is that it can automatically fall back to older versions if it detects incompatible hardware. One of the parameters to D3D11CreateDeviceAndSwapChain() will hold the actual feature level being used once the function returns.

The creation of the swap chain automatically creates the back buffer resource, but we need a specific wrapper (called a *view*) around it so we can use it correctly throughout the lifetime of our game. This means we need to get a reference to the buffer from the swap chain and build a *Render Target View* for it. This is essentially a wrapper that denotes that we plan on using this particular resource as a *render target* (putting new colors in), rather than as a *texture* (pulling colors out).

Next up, we create a *depth buffer*, which handles basic depth-based occlusion for us (*not* sorting! A depth buffer doesn't sort anything). Again, we fill out a large struct with the options we want, and then pass it to a method that creates the resource. This is a very common patterns with Direct3D and Win32 programming. A few of the bits of each pixel in a depth buffer are sometimes used for a feature called stenciling, so the full name of the view is the *Depth Stencil View*.

Lastly, we set the current render target and depth buffer, and create a viewport that defines how much of the window we actually want to render into. Our viewport will be the entire window, although it's possible to create multiple, smaller viewports and swap between them while drawing to create split-screen games.

DXCore::OnResize

The resolution of our buffers (like the back buffer and the depth buffer) are tied to the size of the window. So, what happens when we resize the window? If we do nothing, the window's resolution and our rendering resolution won't match.

OnResize() is called by another part of DXCore when we get a message from Windows telling us the window size has changed. It releases any existing views into GPU resources, resizes and/or recreates those resources, and wraps them in new views. It also updates the viewport to match, so we always fill the whole window, and determines if we're in fullscreen mode.

DXCore::Run

Run() is the last thing called from our WinMain() function, after the window and DirectX are ready. It's where our actual game takes place, and where OS-level messages are first processed.

The first thing that happens in Run() is we get a timestamp that represents the "start" of the game, so we can measure total time during the game loop. Next, the pure virtual Init() method is called, allowing a subclass the chance to do any initialization logic.

Then we enter the overall game and message loop, which is a while loop that continues until it finds a WM_QUIT message. This message could come from several sources: closing the window, pressing Alt+F4 or by adding our own quit message to the message queue.

Inside this loop, we first determine if a message is waiting to be processed by peeking at the queue. If one is there, we remove it from the queue and handle it. It is first *translated* (this alters some keyboard input to be more appropriate), and then *dispatched* to our static WindowProc() function.

> *How does it know to call WindowProc() during dispatch? Remember: A function pointer to WindowProc() was required as part of the creation of the window, back in InitWindow().*

If there are no messages waiting to be handled, our game loop proceeds: We update the timer and input manager, potentially update the window's title bar, and then call Update() & Draw(). The last step is to let the input manager know that the frame is over so it can properly reset for next frame.

Once we get a quit message, the loop ends and we return the result from that message. Since Run() ends, so does our WinMain() function and our overall program.

DXCore::Quit

Sometimes we want to be able to quit through code, like when the user clicks "exit" on a menu or we detect an unrecoverable error. Since our game loop ends when we get a quit message, we can simply post our own quit message and our program will end gracefully.

DXCore::UpdateTimer

Each time through our game loop, we want to know how much time has passed, both since last frame and since the beginning of the game. UpdateTimer() queries the performance counter to get the latest timestamp, and then calculates those pieces of information.

Both deltaTime and totalTime are stored as floats, since they're most often used with our math library's vectors and/or matrices, all of which use floats instead of doubles. This saves us from having to cast from double to float over and over later on. Since they're fully recalculated each frame, we shouldn't run into any round-off errors.

DXCore::UpdateTitleBarStats

This method counts frames until 1 second has gone by, and then updates the title bar with:

- Width and height of the current window
- Current FPS
- Approximately how long each frame took (ms/frame)
- The version of DirectX that is currently in use (usually 11)

This is not the most performant function, but it's very useful to gauge *relative* performance. If you add a feature and see your framerate drop drastically, you know where to start optimizing.

One thing to remember: **Don't judge your overall performance while in debug mode!** Try running your game in release mode and check the FPS. Chances are it will be vastly higher, as Direct3D won't be in debug mode and functions marked as inline will actually be inlined.

Seriously, if you've only ever looked at the performance of your program, or libraries like STL, in debug mode, you haven't seen the complete picture.

DXCore::CreateConsoleWindow

Since our program is a Win32 graphical application, it doesn't have a console attached to it. However, it's often useful to have a console available while debugging. To that end, this method creates a console and redirects the stdin, stdout and stderr streams to it.

It also disables the close button on the console window, as closing the console also closes the overall application. If that's desirable behavior to you, comment out the last 3 lines in this method.

DXCore::ProcessMessage

This is the member function that is called from our static WindowProc() function, allowing us to both handle messages and interact with our game loop. The majority of this function is a big switch statement, each case handling a different message.

When we handle a message ourselves, we return zero to inform the OS that we're done with it. If, however, it's a message we don't care about, we still have to report that to Windows. Hence the final line of this function, DefWindowProc(), which tells Windows to use the default processing for this particular message.

Below you'll find a quick list of the messages this method handles:

- **WM_DESTROY**: The window is closing, so we post a quit message
- **WM_MENUCHAR**: Alt was pressed, suppress beeping for incorrect key combinations
- **WM_GETMINMAXINFO**: Prevent the window from getting too small
- **WM_SIZE**: The window size has changed, so we need to resize
    - We get a constant stream of these messages while dragging a window's border
    - Performance also tanks since we're constantly resizing buffers on the GPU
- **WM_MOUSEWHEEL**: Capture mouse wheel input and forward it to the input manager
- **WM_INPUT**: Handles raw mouse input from the device itself
- **WM_SETFOCUS, WM_KILLFOCUS, WM_ACTIVATE**: Track whether our window is in focus or not

Game Header

Game.h includes the following headers:

- DXCore.h – Since Game inherits from DXCore
- DirectXMath.h – The math library we'll be using

It also defines several Direct3D resources (as ComPtrs) we'll need to draw a triangle:

- vertexBuffer – Holds the vertices of our triangle on the GPU
- indexBuffer – Holds indices, on the GPU, into the vertex buffer
- pixelShader & vertexShader – Shadercode the GPU can execute while rendering
- inputLayout – Describes the layout of an individual vertex

I'll be discussing the methods in the next section.

Game Class

This class houses our application-specific code. DXCore has all of the stuff *every game* needs, so Game will have the stuff *our game* needs. This is where you can really begin customizing.

Much of the code in here will be expanded upon, altered and sometimes even replaced in future assignments. Its current state represents the bare minimum needed to get a single triangle on the screen. It may seem like a lot, and it is, which is why we have starter code.

> *Is inheritance necessary here? Nope! But it creates a logical delineation between the “core” Direct3D bits and the “game". If you really don’t like that, refactor to your heart's content.*

Includes & Namespaces

Obviously, we need Game.h. We’re also including Vertex.h, which defines the structure of vertices we’ll be sending to the GPU. Since this will be used in several places in the future, it makes sense to separate it out into its own file. The input manager’s header, Input.h, and the Helpers.h header are also included.

Another library file, “d3dcompiler.lib”, and the associated header, <d3dcompiler.h>, are specified here as well. These provide a helper function we’ll be using to load a compiled shader file at start up. We’re also using the DirectX namespace since our math library resides there (though it’s not in use yet).

Game Constructor

The constructor receives the application’s unique handle when created in WinMain(), and passes it to the base constructor. Also being passed to the base constructor are the title bar text, window size and a Boolean for showing additional stats. The “vsync” variable determines if we want to lock the framerate to the refresh rate (vertical synchronization) of the monitor. Feel free to customize!

Internally, the constructor creates a console window (only in debug mode). If you don’t want the console, or you’d like it to exist in release mode for some reason, here is where you can change those things. You can edit or remove the sample printf() call as well.

Game Destructor

This is where you should delete any objects you create within the Game class. **You are responsible for cleaning up your memory leaks and unreleased Direct3D resources!** You will lose points if you turn in assignments with memory leaks (or warnings, for that matter). As mentioned in the DXCore section, the starter code is using ComPtr’s, so the preexisting Direct3D objects should be cleaned up automatically.

Game::Init

Called exactly once by DXCore after the window & Direct3D are ready, but before the game loop begins.

It currently calls two helper methods, each of which does a piece of the basic initialization for the game. You should alter, add to or otherwise adjust these to suit your own needs in later assignments.

The last thing Init() does is set some default graphics state. First, we set the default primitive topology ("Hey Direct3D, I want you to render triangles"). Then we set the input layout and activate shaders.

> *Since Direct3D saves state, you only need to set these once unless you plan on rendering multiple different types of primitives (like triangles and lines) or mixing different shaders in one frame. However, completely removing these lines of code will have different results based on your GPU manufacturer and current drivers. For instance, removing the topology call may default to triangles on one machine, to points on another, and may hard crash the GPU on yet another.*

Game::LoadShaders

Direct3D has a fully-programmable rendering pipeline. This means to draw anything with Direct3D, you must use shaders. There's literally no other option.

Our shader code (stored in .hlsl files) is actually compiled for us into compiled shader object files (.cso files) at build time in Visual Studio. Visual Studio will even detect and highlight syntax errors, provide line numbers, etc. It's quite nice. So, we write .hlsl files, but load .cso files. Direct3D comes with everything we need to load and use shaders.

The other thing happening in this method is the creation of an **Input Layout**, which describes how our GPU should interpret the raw numeric data that represents our 3D models. The reason this is handled immediately after loading our vertex shader is that the layout must be verified against an actual shader to ensure it is valid. Since we already have the compiled vertex shader loaded at this point, it makes sense to take care of the input layout here as well.

Game::CreateGeometry

If we want to draw something on the screen, in this case a triangle, we need the data to do it. Here we're defining that data: 3 vertices and 3 indices.

Each vertex is made up of a 3-component position (XYZ), and a 4-component color (RGBA). This exact vertex definition can be found in Vertex.h. We then create a Direct3D GPU resource called a Vertex Buffer to store the vertices directly in GPU memory.

> *Since we're not dealing with any matrix math yet, the numbers we plug in for the vertex positions will be interpreted by the graphics card as final screen coordinates. These screen coordinates, often called* Normalized Device Coordinates*, are resolution-agnostic; they do not correspond to individual pixels. Instead, the bounds of the screen on X and Y are -1 to 1, with (0,0) being the center of the screen. The Z value represents the depth of the pixel inside our 3D "scene", and will only be valid (visible to us) if it falls between 0 (our near clip plane) and 1 (our far clip plane). In future assignments, we'll be creating several matrices – world, view and projection – to have more control over these coordinate systems.*

One option when drawing is to also provide indices into our vertex list, so that neighboring triangles can potentially share vertex data (reducing the overall size of our memory footprint and the workload when rendering). While it's not strictly necessary for such a simple example, we're creating indices and an Index Buffer to showcase how they work.

The vertex and index buffers will be used later to actually draw the geometry.

Game::OnResize

Our game class sometimes wants to be notified of window resizes so it can (eventually) update things like the aspect ratio of our camera. We also need to call the base class version to ensure everything else under the hood, like the back buffer, gets updated appropriately.

Game::Update

This is the first half of the game loop, and is where all of your game logic and input should be handled. For now, it simply checks for the escape key and quits if necessary.

Game::Draw

This is where we'll actually put stuff on the screen. There are a few things we need to do every frame, regardless of what we're drawing. Some of the code here will be moved elsewhere in future assignments as you create game entities and material systems.

At the beginning of each frame, we need to clear everything we rendered last frame. By default, we're clearing the screen to Cornflower Blue (feel free to change). We then call Direct3D methods to clear both the Render Target View (the colors) and the Depth Stencil View (occlusion information).

Now we can actually draw something to the screen. First, we activate a particular set of vertex and index buffers. (Since we only have one of each, we could just set them once at start up, but it's a demo.)

We can now call DrawIndexed() to tell the GPU to use the currently set vertex buffer, index buffer, shaders and other options to kick-start the rendering pipeline and get some pixels on the screen. We'll need to call DrawIndexed, or some other draw command, *each time* we want to draw something to the screen. We do not have to re-set resources that remain the same between our draws, such as the input layout, but we may need to change things like buffers and shaders.

Lastly, the user won't see anything until we tell the swap chain to Present(). This should be done only once per frame, after we've drawn everything. Afterwards, we need to rebind the render target and depth buffer, as the swap chain will unbind it automatically after Present(). Note that we're using a Boolean variable of the Game class, *vsync*, (among others) to determine if this call to Present() should limit our framerate to our monitors refresh rate or allow it to run as fast as possible.

Helpers.h and Helpers.cpp

This is a great place to put global helper functions. For now, it has a few related to strings and file paths.

GetExePath

Gets the file path to this executable. You won't really need this yourself, but the next function uses it.

As it turns out, the default relative path for a program is different when running through Visual Studio and when running the .exe directly, which makes it a pain to properly load files external files (like textures and shaders). This behavior could be fixed by changing a setting in Visual Studio, but the option is stored in a user file (.suo), which is ignored by most version control systems by default. Meaning: the option must be changed every on every PC. So instead, here's a helper. Note that it returns a *wstring*.

> #### *Reminder: What's a wstring?*
> *The **char** data type is usually 8 bits, meaning it can hold 256 different values. However, there are certain character sets, like Unicode, that need to represent more than 256 different characters. To facilitate this, C++ has a special type of character called a **wide character**, which takes up more than 8 bits. The data type of a wide character is **wchar_t**, and just like the **string** class, there is also a **wstring** class. Many of the Win32 API functions require wide character strings rather than a string made up of 8-bit characters, so that's what I've defaulted to here.*

FixPath

This function takes a relative file path (or just a filename) and prepends it with the full path to the executable. The result is the full path to the specified "relative" file. This will make your life easier if you ever want to launch your project by double clicking on the .exe file rather than loading up Visual Studio.

WideToNarrow / NarrowToWide

We'll be using wide character strings for most things, but you may find yourself wanted to convert between string/wstring. There is no implicit cast, so I've included two helpers if you need them.

Other Files

- **Vertex.h**: Contains the definition of our basic vertex data
- **PixelShader.hlsl**: Basic pixel shader for drawing (OpenGL calls these Fragment Shaders)
- **VertexShader.hlsl**: Basic vertex shader for drawing
- **Input.h** and **Input.cpp**: Input manager singleton to simplify mouse/keyboard input

These will be discussed in more detail in class and in future assignments.

Pheww, now what?

If you made it here, that's awesome. You probably still have some questions though. Please ask them in the appropriate survey on MyCourses, which I'll be answering in class, and/or during class itself, as other students might also benefit from the answers.