Assignment 2 – Meshes

Due by 11:59PM on Thursday, January 24th.

# Overview

One triangle is great, but more is better. You’re going to start setting up the basics of a graphics engine by generalizing the “geometry” code that comes with the starter files. This will allow you to easily draw more sets of triangles (meshes) to the screen. Your tasks are described in the “Getting to Work” section on page 5. But first: some background information.

# Geometry Basics

The process of “putting stuff on your screen” in DirectX is called *drawing*. DirectX, as with most modern graphics APIs, can draw 3 types of geometric primitives: points, lines and triangles. Anything more complex is really just a combination of these (for instance, 3D models are just a collection of triangles). All three geometric primitives are made up of vertices: points in space that describe more complex geometry.

## Vertices

Vertices can (and in games usually do) contain multiples pieces of information related to the geometry they represent. The simplest vertex would only contain a position in either 2D or 3D space. If you’ve ever done any modeling and texturing in Maya or another modeling package, you’ve probably worked with UV coordinates. These coordinates control how a texture is applied to each triangle, and are actually stored at each vertex along with its position.

We’ll be starting with fairly simplistic vertex data: a position in 3D space and a color. We’ll add more data to the vertices as the needs of our engines grow, and potentially even support multiple types of vertices.

## Vertex Struct

The Vertex.h file in the starter project has a struct called *Vertex*. It contains two variables – a position (3 float values) and a color (4 float values) – in that order. A useful feature of structs in C++ is that their layout in memory is predictable. If I create a *Vertex* variable, I can be sure that it’ll contain exactly 7 float values, the first 3 of which are the position and the last 4 of which are the color. If I change the order of the variables in the struct definition, that layout in memory changes accordingly.

Let’s say that, instead of a single variable, I make an array of *Vertex* structs. I can be sure that all of the data in the array is in contiguous memory (in a row). So the data in a *Vertex* array would look like this in memory:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data:** | **float3** | **float4** | **float3** | **float4** | **float3** | **float4** |
| Meaning: | (position) | (color) | (position) | (color) | (position) | (color) |
| Vertex: | Vertex 1 | | Vertex 2 | | Vertex 3 | |

Luckily, this is exactly how DirectX would like its vertex data: Each vertex having the same layout, all next to each other in memory. The code in Game.cpp, lines 148 - 153, creates a literal array of Vertex structs. We can be sure the data in that array will remain in the same order in memory.

However, we can’t just use a raw C++ array when we draw. We need to create a special DirectX object, called a *Vertex Buffer*, which can store this data in the memory of the GPU itself. This is important because copying data from your CPU-side memory (RAM) to GPU memory takes a not-insignificant amount of time. If we care about efficiency, which is extremely important in a game, it’s much better to have this data in GPU memory while the game is running. Once we copy the data to a *Vertex Buffer* we don’t actually need it anymore in our C++ code, which is why the vertex array is a local variable on the stack.

## DirectX Data Types

Creating *Vertex Buffers*, or any DirectX-controlled object, isn’t the most straightforward process if you haven’t done it before. You’ll come across DirectX data types that look like one of the following:

* **ID3D11Buffer** – These always start with ID3D11
* **D3D11\_BUFFER\_DESC** – These are always all capital letters, and start with D3D11

Notice that the first example (ID3D11Buffer) begins with the letter “I”, which stands for interface. These kinds of data types are always created by DirectX, and you’ll always need to store them as pointers. You can’t actually create them yourself – you can’t call *new* on them – because they’re abstract classes. Instead, you’ll call a DirectX function that creates the object for you, passing in a description of the object you’d like to receive and the *address* of a *pointer* to one of these variables.

Yes, that’s technically a pointer to a pointer. This is how the DirectX API works: You make an empty pointer to one of these objects, and when DirectX creates the object it’ll fill out that pointer for you. If this pointer business is giving you a headache, feel free to stop by my office hours for a more in-depth explanation.

The other all-caps example (D3D11\_BUFFER\_DESC) is either a struct to be filled out by you or a predefined enum value used as a parameter somewhere. In this case D3D11\_BUFFER\_DESC represents the description of a buffer you’d like DirectX to create for you. You’d create a local variable of this type, set all of its member variables appropriately, and then pass it to a method called CreateBuffer().

## Creating a Vertex Buffer

The code in Game.cpp, between lines 166 - 181, creates a vertex buffer. It first creates a description of the vertex buffer (the *vbd* variable). This is simply a local variable on the stack because we won’t need it after we create the buffer. The member variables of this description are mostly defaults that we don’t need to discuss yet, with the exception of *Usage*, *ByteWidth* and *BindFlags*.

*Usage* tells DirectX what kind of read/write permission we expect the resource to need. We end up going with D3D11\_USAGE\_IMMUTABLE, which signifies that we’ll never change the data stored in the buffer, and that the buffer will be read only by the GPU. This is one of the fastest options, resulting in the drivers storing the data directly in GPU memory.

*ByteWidth* is how big the vertex buffer should be, in bytes. This is the size of a single Vertex multiplied by the total number of vertices we want to store.

*BindFlags* tells DirectX how we intend to use this buffer. Remember that a buffer is really just a bunch of memory. By choosing the correct bind flag, DirectX internally makes some decisions about how best to store and use the buffer. For vertex buffers, the correct value is D3D11\_BIND\_VERTEX\_BUFFER.

Notice there’s another struct here called D3D11\_SUBRESOURCE\_DATA. This can be used in a few places, but here it’s used to tell DirectX where to find the data we want to copy to the vertex buffer upon creation.

The last line of code here actually calls the CreateBuffer() method. This method is being called on the *device* variable, which represents the “Direct3D Device”: one of our doorways into DirectX. The *device* is primarily used for resource creation. The three parameters to the method are the buffer description, the initial data (optional) and the pointer that will hold the resulting buffer object.

Almost all DirectX methods return an HRESULT, which tells us if the function succeeded or failed (and if so, what the error was). It’s basically just an integer. If you find that things aren’t working as intended, you may want to store the results of any problematic DirectX methods and check them. If a method succeeds, the result should be equivalent to the constant S\_OK.

## Index Buffers

We’ve created an array of vertices and given it to DirectX in the form of a Vertex Buffer. So we’re ready to draw? Drawing vertices with DirectX *can* be done with just the individual vertex data, but we’re going to look at an advanced option: *Index Buffers*. An index buffer is just a list of indices –a bunch of integers – which specify the order the GPU should use the vertices.

*Index Buffers* are useful for a few reasons. First, index buffers can contain the same index multiple times, meaning we can share vertices among triangles. Second, geometric shapes like triangles have a *front side* and a *back side*, defined by the *winding order* of the vertices. Winding order refers to the order of the vertices, either clockwise or counter-clockwise, when looking at a triangle in 3D space. Our indices are used to determine this order.

0

1

2

DirectX defines the front of a triangle as having a clockwise winding order and, by default, only renders the front side of triangles. This is an optimization we’ll discuss later, and can be changed if necessary.

Here’s an example of vertex sharing. Take the mesh to the right: it has two triangles but only 4 unique vertices. If our Vertex Buffer only has 4 vertices in it, DirectX needs to know how to combine them in sets of three when drawing. This is where the *Index Buffer* comes in. It literally contains indices into the Vertex Buffer.

0

1

2

3

By default, every three indices in an *Index Buffer* defines one triangle. A valid *Index Buffer* for the mesh above would contain the following integers: 0, 1, 2, 0, 2, 3. Line 160 in Game.cpp defines the indices used by the starter code. It’s simply an array of unsigned integers that correspond to the vertices in the Vertex Buffer, which will need to be copied to an actual *Index Buffer*. This may seem like overkill for such a simple shape, but we’re just using it as an example here.

Lines 188 - 203 in Game.cpp create that *Index Buffer*. The steps are very similar to the Vertex Buffer creation. The main differences are how to calculate the ByteWidth of the buffer, and the fact that the buffer is being bound as an Index Buffer instead of a Vertex Buffer by the BindFlags.

## Activating Buffers & Drawing

Once you have the two geometry-related buffers you need, you can activate them and start drawing. Lines 277 - 280 in Game.cpp *set* (or activate) the vertex and index buffers. The methods used are IASetVertexBuffers() and IASetIndexBuffer(), which are part of the “Direct3D Device Context” object. The *device context* (or *context* for short) is the object through which we change settings and issue drawing-related commands to DirectX. These two methods work slightly differently, in terms of parameters, because there are some advanced uses of vertex buffers which require more data.

Once the buffers are set, drawing is accomplished through the DrawIndexed() method of the device context. See line 287. The first parameter tells DirectX how many indices to use from the currently set index buffer. It will then use that many indices to look up corresponding vertices in the vertex buffer and draw those triangles. The other two parameters are used if you’d like to only draw a subset of your geometry. An advanced optimization is to combine multiple sets of geometry into one vertex buffer, then draw different portions of it as necessary, rather than swapping between different buffers.

# Getting to Work

Currently the starter code defines geometry and creates the appropriate buffers in the CreateBasicGeometry() method. It also sets the buffers during the draw step, then actually draws with them in Draw(). Your job is to generalize these steps by moving them to a class specifically designed to create and keep track of the geometry: a Mesh class.

For this assignment, you will be starting with a copy of the starter code I gave you. Future assignments will build on this one and each other.

## Task Overview

Here is the high level overview of what you’ll need to do for this assignment, which is explained in more detail on the pages that follow:

* Create a **Mesh class** to hold geometry data (vertices & indices) in DirectX buffers
* Create **three Mesh objects**, with different geometry, in the Game class
* Alter the Draw() method in Game to **draw your meshes** to the screen
* Clean up any and all **memory leaks**, and be sure to **release your DX resources** properly
* Clean up any and all **warnings** in your code (seriously, don’t submit code with warnings)

## The Mesh Class

You have two options when dealing with a class like this:

1. This class knows how to create the requisite buffers, but other than the constructor it’s just a dumb container for that data. It has a few private variables and some “get” methods.
2. The class knows how to both create and use the buffers. It keeps track of them with private variables and has methods for activating the buffers and drawing with them.

For this assignment, either architecture described above works. I’m going to explain what you’ll need for the first option above; the second option would require some extra methods and parameters. One such parameter would be a reference to the D3D device context, so each Mesh object is able to activate (set) itself and issue its own Draw commands.

An advanced engine will generally go with the first option on the previous page, as it allows all rendering-related logic to be placed in some kind of Renderer class. The Renderer can then sort renderable entities into groups based on their mesh, material or transparency options, as it then won’t need to swap rendering options as often. Another big advantage here is with *Instanced Rendering*. Instanced Rendering draws multiple copies of a mesh using a single draw call, resulting in less overhead and better performance. To do this, your engine must first collect data about everything it intends to draw (like the world matrix of each entity), send that data to the GPU and then issue the single draw command. This is much harder to do if every object is in charge of drawing itself.

I encourage you to think about the overall architecture as you work through these assignments. You may want to customize the Mesh class a little more once you have the basic assignment working.

Create a new class in the starter code project called Mesh. Include the following private variables and functions at a minimum:

* Two ID3D11Buffer pointers, one for the *vertex buffer* and one for the *index buffer* of this mesh
* An integer specifying how many indices are in the mesh’s index buffer
  + This is used when drawing
* A constructor that creates the two buffers from the appropriate arrays. You should copy, paste and adjust the code from the CreateBasicGeometry() method as necessary. The constructor will need the following parameters:
  + An array of Vertex objects. (If you haven’t used C++ in a while, remember that arrays and pointers are kind of the same thing, so this can just be a pointer to a Vertex)
  + An integer specifying the number of vertices in the vertex array
  + An array of (again, just a pointer to) unsigned integers for the indices
  + An integer specifying the number of indices in the index array
  + A pointer to an ID3D11Device object, which is required to create buffers
* A destructor which calls the Release() function on both buffers. Cleaning up DirectX objects is your responsibility. Every time you create something, you must release it.
* A GetVertexBuffer() method, which simply returns the pointer to the vertex buffer object
* A GetIndexBuffer() method, which does the same thing for the index buffer
* A GetIndexCount() method, which returns the number of indices this mesh contains.
  + Remember that we need that information whenever we draw anything in DirectX.

Here are a few odds and ends:

* You’ll need to include the <d3d11.h> header file anywhere you want to reference DirectX “stuff”
* You’ll also need to include the “Vertex.h” file, as that is where our custom Vertex struct is defined.

## Putting It All Together

It’s time to start using the Mesh class. You’re going to create at least 3 Mesh objects and use them to draw 3 different shapes to the screen. Start by creating 3 private Mesh variables in Game.h. If these are pointers to Mesh objects, you’ll need to create them using *new* and delete them in the Game destructor.

In CreateBasicGeometry(), you can keep the array of vertices and the array of indices (named “vertices” and “indices”) as one of your shapes if you’d like. Remove the vertex *buffer* and index *buffer* sections of code (the description and creation steps). You’re going to replace them with one of your Mesh objects. Create it here, passing in the array of vertices, indices and whatever else your constructor requires. If you can run your code at this point without any run-time errors, you’re in good shape.

You’ll eventually need to create two more Mesh objects here with different sets of geometry in them. You don’t need to go crazy, just something other than the same triangle in the same place on the screen will suffice. The steps will be similar: Create an array of vertices, an array of indices and the corresponding Mesh object.

Lastly, change the code in Draw() to draw your three Mesh objects. Near the end of Draw() you’ll find the methods that set the vertex and index buffers, and the call to DrawIndexed(). You’ll need to repeat these three steps for each Mesh you intend to draw, and they must *all* happen before the Present() method is called. You’ll be editing or replacing the existing 3 method calls (IASetVertexBuffers, IASetIndexBuffer and DrawIndexed) as well as adding several more. Everything else in the method can remain the same.

## Common Issues

A few things to keep in mind if things aren’t working immediately:

* Remember that the winding order of the triangles is important. An incorrect winding order could result in DirectX not drawing a particular triangle. You can change the winding order by changing the order of the indices you’re putting into an index buffer.
* You’ll need to update the first parameter of DrawIndexed() if you’re drawing more than 1 triangle at a time.
* Inside the Mesh class’s constructor: Be sure to actually SAVE the index count parameter to the Mesh class’s private index count variable!
* IASetVertexBuffers() wants the *address* of a *pointer* to a vertex buffer. Really it can work with an array of buffers, but since we’re just using a single buffer you can just use the address.
  + The address can’t directly be the result of a function call! Meaning the following snippet of code will result in an error: &(myMesh->GetVertexBuffer())
  + To fix this, you’ll need to store the result of GetVertexBuffer() in a variable, and use the address of that variable as the parameter.
* IASetIndexBuffer() works with just the pointer itself.

## All Done

Once you can see your three shapes on the screen, you’re done. Since you’re probably not using the original buffer variables (“vertexBuffer” and “indexBuffer”) anymore, you can safely remove them and their associated code from the Game class.

# Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses.