

Assignment 2 – Meshes

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 detail in the "Getting to Work" section, starting on page 6. But first: a task overview and some background information.

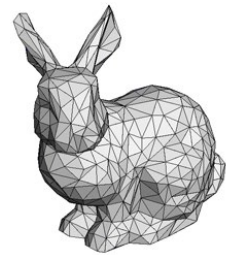
Task Overview

Here is a high-level overview of this assignment.

- ☐ Create a **Mesh class** to hold geometry data (vertices & indices) in Direct3D 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
- ☐ Remove code that is now **redundant** or **unused** in Game
- ☐ General Requirements
 - ☐ Ensure you have no **D3D warnings or errors** (*check the Visual Studio output window*)
 - ☐ Clean up all C++ **memory leaks** (*mostly unnecessary if using smart pointers correctly*)
 - ☐ Properly **release your D3D resources** (*mostly unnecessary if using ComPtrs correctly*)
 - ☐ Clean up any and all **warnings** in your code (*seriously, don't submit code with warnings*)

Geometry Basics

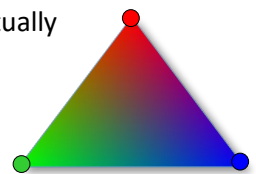
The process of "putting stuff on your screen" is called *drawing*. Direct3D, as with most modern graphics APIs, can only draw 3 types of geometric primitives: points, lines and triangles. Anything more complex is really just a combination of these (for instance, quads are simply two triangles that share an edge and 3D models are collections of many 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 multiple pieces of information related to the geometry they represent. The simplest vertex would only contain a position in 3D space. A more complex vertex would contain additional data like the normal of the surface at that point and UV coordinates for texture mapping. (UV coordinates control how a texture is applied to each triangle, and are actually stored at each vertex along with 3D positional data.)

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, the layout in memory changes accordingly.

Let's say that, instead of a single variable, I make an array of three *Vertex* structs. Arrays store their data in continuous memory (in a row). So, the data in that *Vertex* array would look like this in memory:

Data:	x,y,z	r,g,b,a	x,y,z	r,g,b,a	x,y,z	r,g,b,a
Meaning:	(position)	(color)	(position)	(color)	(position)	(color)
Vertex:	Vertex 1		Vertex 2		Vertex 3	

Luckily, this is exactly how Direct3D would like its vertex data: Each vertex having the same layout, all next to each other in memory. The code below, from *CreateGeometry()* in *Game.cpp*, creates a literal array of *Vertex* structs. We can be sure the data in that array will remain in the same order in memory.

```
Vertex vertices[] =
{
    { XMFLLOAT3(+0.0f, +0.5f, +0.0f), red },
    { XMFLLOAT3(+0.5f, -0.5f, +0.0f), blue },
    { XMFLLOAT3(-0.5f, -0.5f, +0.0f), green },
};
```

However, we can't just use a C++ array to draw triangles. We need to create a special Direct3D object, called a *Vertex Buffer*, which stores its data in the memory of the GPU itself. This is necessary for two reasons: one, copying data from your CPU-side memory (RAM) to GPU memory takes a not-insignificant amount of time and two, the GPU needs the data if we want to use it for drawing. It's better to copy data once for each unique shape and keep all of that data on the GPU than to copy it over and over each time we want to draw something different. Games will have multiple vertex buffers in GPU memory at once, each holding the vertices representing a different 3D model.

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 in the starter code is just a local variable on the stack.

If you wanted to make use of the vertex data in C++ while the game is running, perhaps for collisions or detecting mouse clicks on 3D objects, you would need to keep a copy of that vertex data somewhere in C++ (presumably within a Mesh class). Getting it back from the GPU is technically possible but will severely impact performance. The GPU is running asynchronously from our C++ code; asking for data back requires C++ to wait for the GPU to "catch up", causing a stall (or hiccup) in the pipeline. We'll discuss this further in class when we cover Rendering Pipelines.

Direct3D Data Types

Creating *Vertex Buffers*, or any Direct3D-controlled object, isn't as straightforward as creating a standard C++ object. You'll come across Direct3D data types that look like the following:

- **ID3D11Buffer** – These always start with ID3D11 and are *always pointers*
- **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 Direct3D, 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 Direct3D 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 Direct3D API works: You make an empty pointer, and when Direct3D 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. Note: This works the same way with both raw pointers as well as ComPtr's, the smart pointer for COM objects like Direct3D resources.

The other all-caps example, D3D11_BUFFER_DESC, is a struct to be filled out by you. In this case D3D11_BUFFER_DESC represents the description of a buffer you'd like Direct3D to create for you. You 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 CreateGeometry() method in Game.cpp creates a vertex buffer. It first creates a *description* of the vertex buffer we want (the *vbd* variable), as shown below. This is simply a local variable on the stack; we won't need it again after the buffer is created. 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*.

```
D3D11_BUFFER_DESC vbd    = {};  
vbd.Usage             = D3D11_USAGE_IMMUTABLE;  
vbd.ByteWidth         = sizeof(Vertex) * 3; // 3 total vertices  
vbd.BindFlags          = D3D11_BIND_VERTEX_BUFFER;  
vbd.CPUAccessFlags     = 0;  
vbd.MiscFlags          = 0;  
vbd.StructureByteStride = 0;
```

Usage tells Direct3D 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 once it's stored in the buffer, and that the buffer will be accessed 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: the size of a single Vertex multiplied by the total number of vertices we want to store. The more vertices you have, the bigger this must be.

BindFlags tells Direct3D how we intend to use this buffer at run time. Remember that a buffer is really just a bunch of memory. By choosing the correct bind flag, Direct3D internally makes some decisions about where to store the data and how best to utilize the buffer when we activate (or *bind*) it. For vertex buffers, the correct value is `D3D11_BIND_VERTEX_BUFFER`.

Notice there's another struct here called `D3D11_SUBRESOURCE_DATA`, as shown below. This can be used in a few places, but here it's used to tell Direct3D where to find the data we want to copy to the vertex buffer upon creation.

```
D3D11_SUBRESOURCE_DATA initialVertexData = {};  
initialVertexData.pSysMem = vertices; // pSysMem = Pointer to System Memory
```

The next line of code here actually calls the `CreateBuffer()` method. This method is being called on the *device* object, which represents the "Direct3D Device": one of our doorways into the Direct3D API. The *device* is primarily used for resource creation. The three parameters to the method are the buffer description, the initial data's location and the pointer that will hold the resulting buffer object.

```
device->CreateBuffer(&vbDesc, &initialVertexData, vertexBuffer.GetAddressOf());
```

In the starter code, I'm using a ComPtr for each of my Direct3D objects. As a smart pointer, ComPtr has a .Get() method to retrieve the raw, underlying C++ pointer as needed. However, since what we really need is the address of that underlying pointer (a pointer to a pointer), you'll see I'm using the ComPtr's .GetAddressOf() method here instead.

Almost all Direct3D 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 Direct3D methods and check them. If a method succeeds, the result should be equivalent to the constant `S_OK`.

Side Note: What Is This Pointer?

We've created a vertex buffer on the GPU that holds our geometry. And we have a C++ pointer called "vertexBuffer". Is this a pointer directly to a memory address on the GPU? **Nope!**

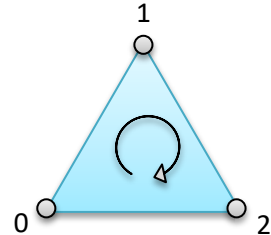
The `vertexBuffer` variable is a standard C++ object of type `ID3D11Buffer*`, stored in system RAM. Internally, it holds several pieces of data, one of which is the *actual memory address* on the GPU that holds our vertex data. However, we're not allowed to know that address directly; it's the API's job to create, destroy and manage GPU resources and memory. (In fact, there's a chance that the API might need to move objects around in GPU memory, so the actual address of the vertices could change. If this happens, the API will update that internal address in our C++ object and we'll be none the wiser.)

Any API call that requires the vertex buffer on the GPU will simply require our `vertexBuffer` variable. Direct3D knows how to use it behind the scenes.

Index Buffers

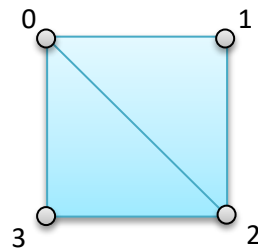
We've created an array of vertices and given it to Direct3D in the form of a Vertex Buffer. So, we're ready to draw? Almost! Drawing vertices with Direct3D *can* be done with just the individual vertex data, but we're going to look at an advanced option: *Index Buffers*. An index buffer is just a list of indices – 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, resulting in less vertex processing by the GPU. 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. The indices are used to determine this order.



Direct3D 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, Direct3D 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.



By default, every group of three indices in an *Index Buffer* defines one triangle. A valid *Index Buffer* for the mesh to the right would be: 0, 1, 2, 0, 2, 3.

The code below defines the actual indices used by the starter code. It's simply an array of unsigned integers that correspond to the vertices in the Vertex Buffer. These numbers need to be copied to an actual *Index Buffer* on the GPU. This may seem like overkill for such a simple shape, but we're just using it as an example in the starter code.

```
unsigned int indices[] = { 0, 1, 2 };
```

The second half of CreateGeometry() in Game.cpp creates that *Index Buffer*. The steps are very similar to 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. The following lines, found in Draw() in Game.cpp *set* (or *activate*, or *bind*) the vertex and index buffers.

```
context->IASetVertexBuffers(0, 1, vertexBuffer.GetAddressOf(), &stride, &offset);  
context->IASetIndexBuffer(indexBuffer.Get(), DXGI_FORMAT_R32_UINT, 0);
```

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 Direct3D. 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 a call to the `DrawIndexed()` method of the device context.

```
context->DrawIndexed(3, 0, 0);
```

The first parameter tells Direct3D 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 unrelated geometry into one large vertex buffer, then draw different portions of it as necessary, rather than swapping between different buffers (which, again, takes a non-zero amount of time).

Side Note: What the Heck is a ComPtr?

Direct3D resources, such as *vertex buffers*, *index buffers* and even the Direct3D *device* object itself, are stored in special kinds of objects. These resource objects abide by the COM, or [Component Object Model](#), interface. They can’t be instantiated directly (using *new* in C++), nor can they be deleted directly. Instead, they’re created through the Direct3D API and have a method we can call – `Release()` – to notify the API when they’re no longer needed.

While you *can* use raw C++ pointers with Direct3D objects, you might be interested in using smart pointers instead. Because of the aforementioned restrictions, however, Direct3D objects can’t be used with standard smart pointers like `unique_ptr`. Luckily, Microsoft provides a smart pointer specifically for COM objects: the [ComPtr](#). You’ll see these in use in several places throughout the starter code.

You will be using ComPtrs for your Direct3D resources; it will greatly simplify their usage and, especially, their cleanup (no need to call `Release()` yourself). If you’re not sure where to begin, feel free to mimic the declaration and usage of ComPtrs throughout the starter code. For a few more examples of Direct3D-specific ComPtr usage, see [this article](#) on GitHub.

Getting to Work

Currently the starter code defines geometry and creates the appropriate buffers in the `CreateGeometry()` 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. Each future assignment will build on the work you do for the previous assignment. This means skipping an assignment will often prevent you from completing future assignments, as they assume the previous work has been completed and is in working order. If you're having issues, please reach out ASAP!

The Mesh Class

You have two options when architecting a class like this:

1. This class will know how to *create* the requisite buffers, but other than the constructor it's just a dumb container for that data. It will have a few private variables and some "get" methods.
2. The class will know how to both *create* and *use* the buffers. It will keep track of them with private variables and will have methods for *activating* the buffers and *drawing* with them.

For this assignment, you'll be implementing the *second option* above. However, the following section explains the minimum setup necessary for both the *first and second option* above, as it may be useful to swap exactly how you use these meshes later on.

In other words, you may be asked to write a few methods you won't necessarily use just yet!

An advanced engine will generally go with the first option above, as it allows all rendering-related logic to be placed in some kind of centralized Renderer class. The Renderer can then sort render-able entities into groups based on their mesh, material or transparency options, as it then won't need to swap render settings as often. For us, having a Draw() method will save time.

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, though, your engine must first collect data about everything it intends to draw (like the world matrix of each entity), send that collective data to the GPU in a single resource and then issue the proper instanced 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. Feel free to customize the Mesh class a little more once you have the basic assignment working.

Making a Mesh

Create a new class called Mesh in the starter code project. This class should **NOT** inherit from anything. If there is data or a variable from Game.cpp you think you'll need, you'll have to pass it in as a parameter. Include the following *at a minimum*:

Private Variables

- Two ID3D11Buffer ComPtrs, one for the *vertex buffer* and one for the *index buffer* of this mesh
 - Use `Microsoft::WRL::ComPtr<ID3D11Buffer>` as the data type
 - Notice that, when using ComPtr, you do not need the `*` symbol after ID3D11Buffer
- A ComPtr to an ID3D11DeviceContext object, used for issuing draw commands
- An integer specifying how many indices are in the mesh's index buffer, used when drawing

Public Methods

- `GetVertexBuffer()` method to return the pointer to the vertex buffer object
- `GetIndexBuffer()` method, which does the same thing for the index buffer
- `GetIndexCount()` method, which returns the number of indices this mesh contains.
 - Remember that we need this information whenever we draw anything in Direct3D.
- `Draw()` method, which sets the buffers and tells DirectX to draw the correct number of indices
 - Refer to `Game::Draw()` to see the code necessary for setting buffers and drawing

Constructor & Destructor

- 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 unsigned integers for the indices (or a pointer to a single unsigned integer)
 - An integer specifying the number of indices in the index array
 - A ComPtr to an ID3D11Device object, used for creating buffers
 - A ComPtr to an ID3D11DeviceContext object, used for activating buffers & drawing
- Since we're using smart pointers, your destructor won't have much to do (it'll be empty). Properly cleaning up Direct3D objects is your responsibility. Smart pointers will do this for you if you're using them correctly. However, if you *were* using raw C++ pointers, you'd need to call `Release()` on them instead of deleting them, since the Direct3D API created them.

Here are a few reminders:

- You'll need to include the `<d3d11.h>` header file if you want to reference Direct3D "stuff"
- Include `<wrl/client.h>` when using ComPtrs for Direct3D objects
- Also include "Vertex.h" as necessary, 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 different Mesh objects** and use them to draw **3 distinct shapes** to the screen. One of them can be a triangle, but the other two must be different shapes! Keep in mind that anything more complex will be made up of *multiple triangles*.

I highly suggest using smart pointers for your C++ objects. As meshes will eventually be shared among different game entities in a future assignment, you should use `std::shared_ptr` for them. You'll need to `#include <memory>` for smart pointers. The easiest (and proper) way to create a shared pointer for a new object is to use `std::make_shared()` and pass the constructor parameters directly in; you **don't** need to make an object first and then wrap it in a shared pointer. Declaration & definition example:

```
std::shared_ptr<Mesh> triangle = std::make_shared<Mesh>(param1, param2, etc);
```

Start by creating 3 private `std::shared_ptr<Mesh>` variables in `Game.h` (or, even better, a `std::vector` of them). If you're feeling bold and want to use raw C++ pointers, you'll need to create them using *new*, and *delete* the pointers in the `Game` destructor.

Yes, these should absolutely be pointers. You want to be sure you're in complete control of the lifetime of the objects, and you especially don't want their destructors to be called when your local variables go out of scope, as that would clean up your Direct3D resources too soon!

In `CreateGeometry()`, 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 a Mesh object. Create that Mesh object 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 errors, you're in good shape.

You'll eventually need to create **two more Mesh objects** here with *different* sets of geometry. You don't need to go crazy; just something other than a single 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. See page 5 of this document for a reminder of how indices work.

Lastly, change the code in `Game::Draw()` to draw your three Mesh objects. In the middle of `Game::Draw()` you'll find the code that sets the vertex and index buffers, and a call to `DrawIndexed()`. You'll be removing these 3 *context* method calls (`IASetVertexBuffers`, `IASetIndexBuffer` and `DrawIndexed`) and replacing them with a single call to your first mesh's `Draw()` method. Then call `Draw()` on your other two meshes. Everything else in `Game::Draw()` method should remain the same.

Note that, when calling a mesh's `Draw()` function, several things happen: The active vertex & index buffers are changed and a draw command is sent to Direct3D. By calling `Draw()` on three meshes, the API calls go like this: change buffers, draw, change buffers, draw, change buffers, draw. This is the basic set up of rendering using a graphics API: change settings to prepare for the first object, draw that object, change settings to prepare for the next object, draw that next object.

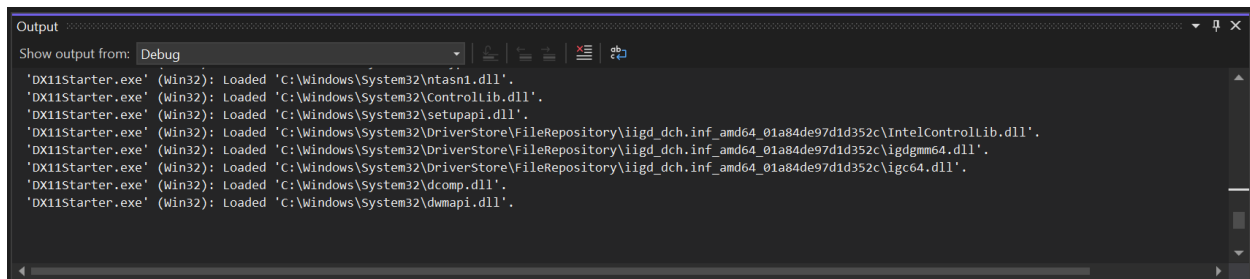
Common Issues

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

- Always check Visual Studio's output window at run time for Direct3D errors & warnings
- Remember that the winding order of the triangles is important. An incorrect winding order could result in your triangle being skipped during rasterization. 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()` so that the number you're passing in matches the number of indices in that Mesh. You should have written a method to get this.
- 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.
 - Simply call `myMesh->GetVertexBuffer().GetAddressOf()`
 - If you weren't using smart pointers, you might run into a weird C++ issue here: The address can't directly be the result of a function call! Meaning the following snippet of code could 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. Again, this issue is present *only* if you're using raw C++ pointers to store your Vertex and Index buffer in your mesh class.
- `IASETIndexBuffer()` works with just the pointer itself: `myMesh->GetVertexBuffer().Get()`

Checking for Leaks & D3D Errors: Visual Studio's Output Window

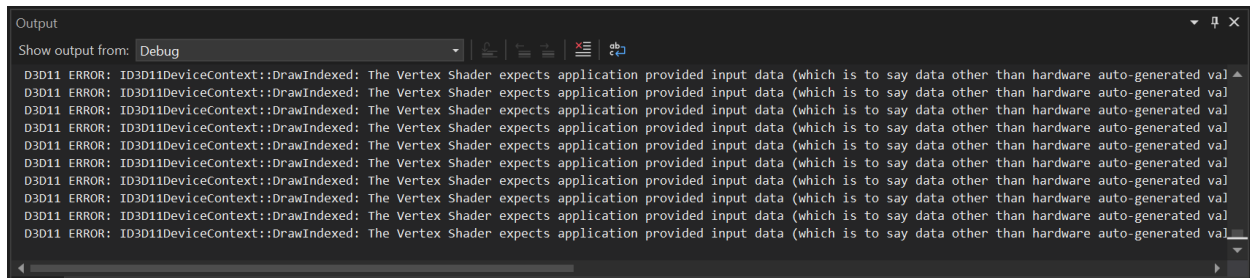
Be sure you have the Output window in Visual Studio visible while running your program, as it will contain any Direct3D warnings or errors that occur. When your program ends, it will also report any memory leaks or D3D object leaks (which you'll need to clean up). Here's a quick look at the output window when our code runs correctly. The type of output you see here is normal and expected.



```
Output
Show output from: Debug
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\ntasn1.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\Controllib.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\setupapi.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\DriverStore\FileRepository\iigd_dch.inf_amd64_01a84de97d1d352c\IntelControllib.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\DriverStore\FileRepository\iigd_dch.inf_amd64_01a84de97d1d352c\igdgmm64.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\DriverStore\FileRepository\iigd_dch.inf_amd64_01a84de97d1d352c\igc64.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\dcomp.dll'.
'DX11Starter.exe' (Win32): Loaded 'C:\Windows\System32\dwmapi.dll'.
```

D3D Warnings & Errors

If, however, you have any D3D warnings or errors while your program runs, it may look like this:

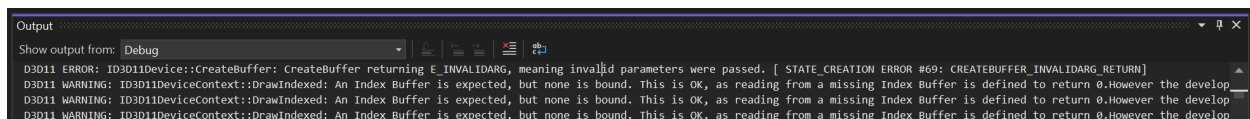


```
Output
Show output from: Debug
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
D3D11 ERROR: ID3D11DeviceContext::DrawIndexed: The Vertex Shader expects application provided input data (which is to say data other than hardware auto-generated val)
```

Here we can see an error that is occurring over and over. The error messages are often fairly detailed, but may use terminology you're not familiar with. As there are too many potential errors to list here, my suggestion is to see if the message makes sense and, if not, reach out for assistance debugging!

The error above is due to the Input Layout being null (I simply commented out the line that creates it in Game::LoadShaders()) so that an example error would occur).

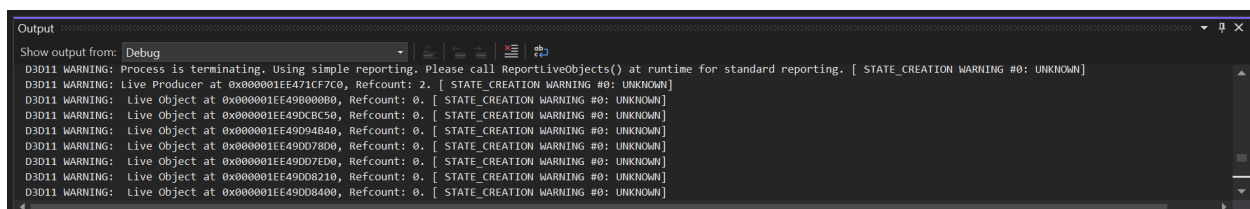
You may also find warnings/errors that occur just once, often when the creation of some resource fails. This is often due to one or more parameters being defined incorrectly, as in the error below (which has an invalid parameter when attempting to create an Index Buffer). However, if a resource isn't created, this can trickle down into other errors that repeat each frame. Be sure you scroll up in the output window to see if there was an initial error that is causing the rest of them.



```
Output
Show output from: Debug
D3D11 ERROR: ID3D11Device::CreateBuffer: CreateBuffer returning E_INVALIDARG, meaning invalid parameters were passed. [ STATE_CREATION ERROR #69: CREATEBUFFER_INVALIDARG_RETURN]
D3D11 WARNING: ID3D11DeviceContext::DrawIndexed: An Index Buffer is expected, but none is bound. This is OK, as reading from a missing Index Buffer is defined to return 0. However the developer should ensure that an Index Buffer is bound before drawing.
D3D11 WARNING: ID3D11DeviceContext::DrawIndexed: An Index Buffer is expected, but none is bound. This is OK, as reading from a missing Index Buffer is defined to return 0. However the developer should ensure that an Index Buffer is bound before drawing.
D3D11 WARNING: ID3D11DeviceContext::DrawIndexed: An Index Buffer is expected, but none is bound. This is OK, as reading from a missing Index Buffer is defined to return 0. However the developer should ensure that an Index Buffer is bound before drawing.
```

D3D Object Leaks

If your program ends while there are still "live" Direct3D objects (objects that the API thinks are still in use), you'll get the following type of output:



```
Output
Show output from: Debug
D3D11 WARNING: Process is terminating. Using simple reporting. Please call ReportLiveObjects() at runtime for standard reporting. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Producer at 0x000001EE471CF7C0, Refcount: 2. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49B00B00, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49DCBC50, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49D94B40, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49D078D0, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49D07ED0, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49D08210, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
D3D11 WARNING: Live Object at 0x000001EE49D08400, Refcount: 0. [ STATE_CREATION WARNING #0: UNKNOWN]
```

If you see something like this, you have D3D objects you haven't properly cleaned up. If you were using ComPtrs for your D3D objects (and you were using them correctly), you should never see this kind of warning. In either case, if you see this, you've messed something up and should fix it. Unfortunately, while this output definitively tells you something is wrong, it is *not* very helpful in tracking the issue down. Ensure each D3D object you create is being cleaned up before the program ends. Again, if you're at a loss for how to fix this, please reach out!

C++ Memory Leaks

Lastly, if you see the following type of output after the program ends, you have one or more traditional C++ memory leaks:

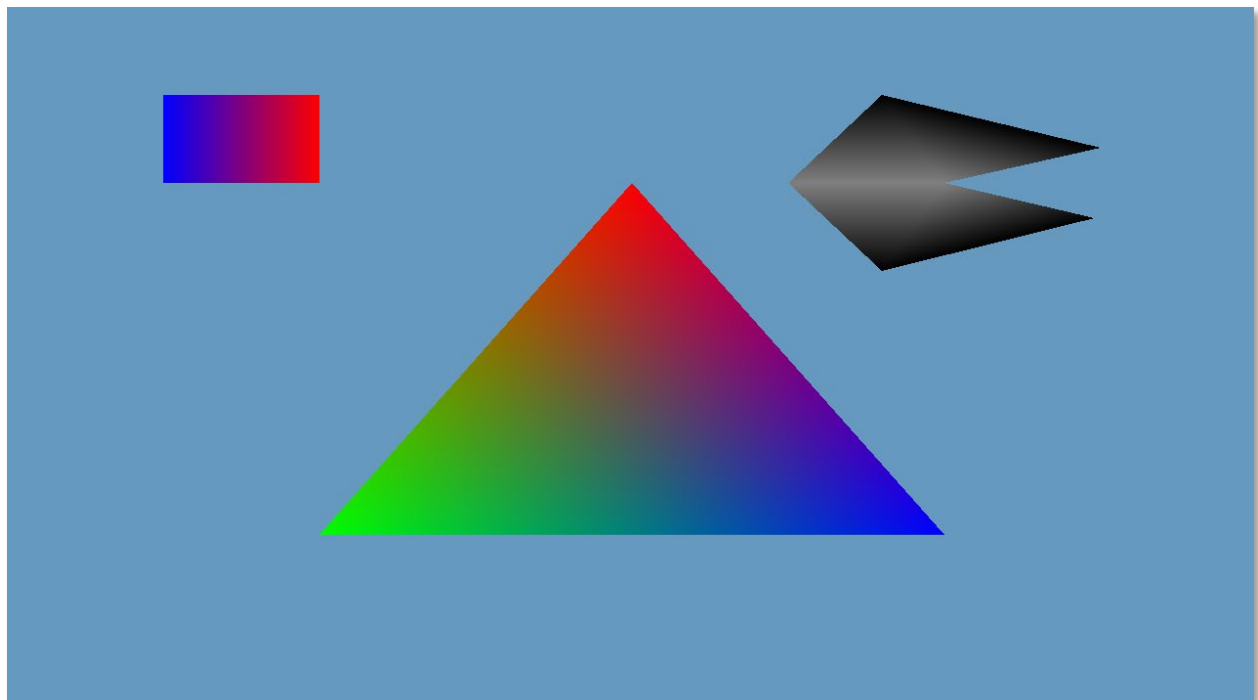
```
Output
Show output from: Debug
Detected memory leaks!
Dumping objects ->
(207) normal block at 0x00000185C460F868, 396 bytes long.
Data: <
> CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
(206) normal block at 0x00000185C4613980, 36 bytes long.
Data: <
> CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
(205) normal block at 0x00000185C4707C30, 3996 bytes long.
Data: <
> CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD
Object dump complete.
The program '[17584] DX11Starter.exe' has exited with code 0 (0x0).
```

Again, while this is useful for knowing you *have* a leak, it's not great info for tracking it down. If you're using smart pointers (and using them correctly), you hopefully won't see this. To begin debugging this, check your code for any uses of the **new** operator. Each one should have a corresponding **delete** (or **delete[]** in the case of an array) within the same overall scope. In other words, if class X *makes* something, class X is responsible for *deleting* that thing.

All Done

Once you can see your three shapes on the screen (and your frame rate hasn't crashed down to single digits), 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.

Here is an example of minimally acceptable output with 3 distinct meshes.



Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. Please see the **“Preparing a Visual Studio project for Upload” PDF** on MyCourses for information on how to properly remove unnecessary files before zipping your project.

Seriously: Please do not submit zip files for this assignment that are tens or hundreds of megabytes in size. The resulting zip should be a couple hundred kilobytes or smaller for this assignment!