A3 – User Interface & Constant Buffers

Overview

Last time, you started working with vertex & index buffers by wrapping them up in a Mesh class. This time, you'll be creating the Direct3D resources and C++ structures necessary to pass extra data to particular steps of the rendering process. The D3D resource you need is called a *constant buffer.*

Before that, however, you'll be integrating a very popular user interface system called "Dear ImGui". This will allow you to very quickly and easily add a debug user interface to your engine for testing and experimentation. For this assignment, you'll use this user interface to change the data in the constant buffer mentioned above as your program runs.

Task Overview

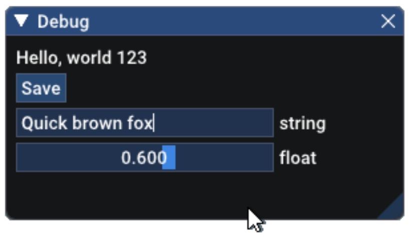
Here is the high-level overview of tasks, which are explained in more detail on the pages that follow:

- ❑ Implement the **Dear ImGui** library
    - ❑ **Download the code** files from GitHub and **integrate Dear ImGui** into your engine
    - ❑ **Create a small custom UI window** for your project
- ❑ Set up your engine to use a single **constant buffer**
    - ❑ **Read** a whole lot about constant buffers
    - ❑ **Use this constant buffer** in both C++ and HLSL to send data to a shader
- ❑ **Hook up** your C++ constant buffer data to your user interface
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**.

Dear ImGui – A user interface library

Dear ImGui is a very popular and easy to use graphical user interface library for C++ that easily can be plugged into most rendering engines. It is available at https://github.com/ocornut/imgui. A custom user interface can be built with very little code, as shown below.

```
ImGui::Text("Hello, world %d", 123);
if (ImGui::Button("Save"))
    MySaveFunction();
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```



Task 1: Integrate Dear ImGui

Begin by reading the associated Dear ImGui reading on MyCourses. It explains where to get the code and how to integrate it into your project, and includes examples of creating simple interface elements.

Go integrate Dear ImGui into your project now!

Once you have the UI system up and running properly, you should see Dear ImGui's demo window on your screen. Your next task is to **build a small custom interface** in its own window. The exact UI you build is up to you, but it should display the following at a minimum:

- The framerate, which can be retrieved with `ImGui::GetIO().Framerate`
- The window dimensions, from the Game class's windowWidth and windowHeight variables

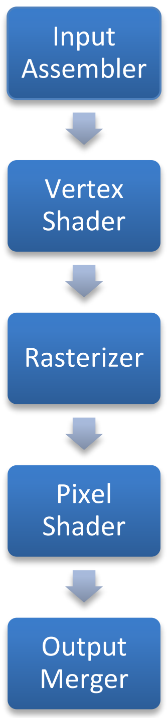
That's it for this task. You'll be adding more to your UI at the end of this assignment.

Next Up: Providing External Data to Shaders

The next major task is to provide some arbitrary data, in addition to your vertices and indices, to the rendering process. Before you jump into the meat of this task, you'll need some background info.

Rendering Pipeline Review

Direct3D 11's Rendering Pipeline



The Rendering Pipeline – the steps Direct3D and the GPU go through once we tell it to render (or "draw") something – can be seen to the right. The buffers you've created inside your Mesh class represent the input data to this process. The data from the individual vertices in the active vertex buffer gets passed through, and altered by, the various stages of the pipeline on its way to becoming pixels on the screen.

Some of these stages, like the Input Assembler and the Rasterizer, are fixed-function: they always perform the same basic operations, though there are some settings we can tweak. For instance, we can change the Rasterizer's *Fill Mode*, allowing us to choose between solid (filled-in) or wireframe (edge-only) triangles. Options like this must be set before we initiate the rendering process.

Other stages, like the Vertex Shader and Pixel Shader, are completely programmable: we can (and, in fact, must) write the code that gets executed at those stages. In HLSL, Direct3D's shader language, each shader is written as a function with parameters (the data flowing *into* that shader from a previous stage) and a return type (the data flowing *out* of that shader). Overall, each stage receives, as input, the data from the previous stage, hence the term *pipeline*.

However, it's quite common to also provide external data – information directly from our C++ code – that we want to incorporate into the work we're doing in a shader. For instance, we'll eventually need to know where our 3D camera is in the scene to properly render objects from that point of view. It doesn't make sense to store the matrices that make up our 3D camera in a vertex buffer; they're not mesh data and they change quite frequently. Instead, we'll need to pass these matrices to a special area of memory on the GPU (another buffer) that can be accessed directly from Vertex Shader code.

> *If you're familiar with OpenGL and GLSL shaders, you may have used **uniform variables** before. These are shader variables that can have their data set directly from C++. This is exactly what we're doing in this assignment, though Direct3D takes a more low-level approach: it handles large chunks of memory at once rather than the individual variables therein. Why? Because it's generally a bit more efficient to send a big chunk of data to the GPU all at once than sending multiple, smaller variables worth of data one at a time.*

Buffers

Any time we want to provide data to the rendering pipeline, that data needs to be in a buffer in GPU memory. You've already made several of these, specifically vertex buffers and index buffers. They're simply chunks of memory that store numbers: vertex buffers hold vertices, each of which is made up of vectors of float values, while index buffers hold unsigned integers.

When creating these buffers, by calling `device->CreateBuffer()`, the data is copied directly into GPU memory. The C++ object you end up with, an `ID3D11Buffer` object, does not actually contain the data; it simply holds the underlying GPU address of the data, among other things, which the Direct3D API uses under the hood. In other words, the `ID3D11Buffer` object is how we tell the API which buffer we're referring to, without knowing exactly where it lives. This is part of the API's job: handle the lower-level details and give us a simple interface to describe what we want to do.

Vertex and index buffers are special in that they represent the input to the rendering pipeline. If you want pixels on the screen, you need to rasterize geometry, which means you need vertices.   The functions `IASetVertexBuffers()` and `IASetIndexBuffer()` are how we denote which vertex and index buffers are currently active, meaning they're used as the source vertices & indices for the next draw call. The vertices will be arranged into sets of triangles by the first stage of the pipeline, the Input Assembler, and then fed into the next stage, the Vertex Shader. The vertex shader program is invoked (run) once for each vertex we're drawing, and each of those shaders receives as its input a single vertex worth of data. The shader's main job is to transform each vertex's position into screen coordinates.

But what about non-vertex data, like transformation matrices, we might need for that task? How do we provide more data for the shader that isn't part of a vertex buffer? As you might imagine, we need *another* buffer: specifically, a *constant buffer*.

Constant Buffers

As the name suggests, a constant buffer is a buffer – a chunk of memory – on the GPU where we can store data. More specifically, constant buffers are areas of GPU memory accessible to one or more shaders as that shader code is executed. Since multiple shader threads are using that data in parallel, they're allowed to read from it but they can't write to it. Hence, the *constant* part of "constant buffer": the data in a constant buffer cannot be altered by a shader. We can, however, overwrite the data in that buffer from C++ as necessary.

Ultimately, this data is just made up of numbers: single scalar values, vectors and/or matrices. These numbers are usually float values, though integers are possible, too. Exactly how many numbers and in what order are up to us.

Data Decisions

Before starting, we need to figure out what kind of data you'll be passing from C++ to your shaders. While future assignments will have you pass matrices, lighting information and material properties, we'll be keeping things very simple for this assignment: you'll be passing a 4-component color used to **tint** the meshes you're drawing as well as a 3D vector that represents an **offset** for your geometry, effectively moving it to a different position on the screen.

> *We'll be replacing some of these specific pieces of data in future assignments, especially once we start working with the DirectX Math library. For now, the goal is to ensure we're simply able to get that data to the vertex shader.*

Task 2: Defining the *cbuffer* in HLSL

Open up the VertexShader.hlsl file in the starter code. Your first job is to define the constant buffer in HLSL, the keyword for which is ***cbuffer***.

Start by defining the following constant buffer at the top of VertexShader.hlsl. Be sure to match each line exactly, including the order and different data types for the two variable declarations.

```
cbuffer ExternalData : register(b0)
{
	float4 colorTint;
	float3 offset;
}
```

Let's break down the individual pieces of the definition above:

`cbuffer ExternalData`

Each cbuffer needs an identifier, though that identifier is never referenced directly in shader code. Here, the identifier ***`ExternalData`*** is used to signify *to the programmer* the intent of this cbuffer. Some advanced optimizations require us to define multiple cbuffers within a single shader, so being able to name a cbuffer can be helpful for organization.

`: register(b0)`

This piece of the definition might seem strange at first glance. Recall from class that GPU resources, like buffers and textures, must be *bound* to the pipeline to be used during rendering. Binding a resource to the pipeline means that a reference to it is placed in a particular *register* (sometimes called a *slot*). You can think of these registers as pointers to the various resources in GPU memory. If an appropriate resource is bound to one of these pre-defined registers, it is accessible during rendering.

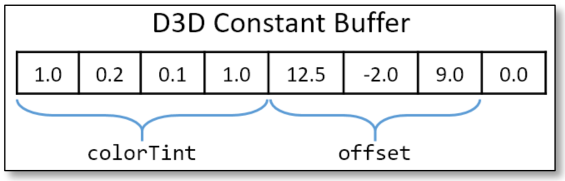
Up to 15 constant buffer resources can be bound to the pipeline simultaneously, each in a different slot. The ***`register`*** keyword tells the shader which slot we're referring to when accessing the variables in this cbuffer. In general, we probably won't use more than one or two per shader.

The ***`b0`*** identifier is not arbitrary; all constant buffers are bound to ***b*** registers (b for buffer). The ***`0`*** is an index, signifying which of the ***b*** registers we're accessing.

```
{
	float4 colorTint;
	float3 offset;
}
```

The body of a cbuffer definition is where we declare the variables that will hold external data (data sent in from C++). A single constant buffer can hold up to 4096 elements, where each element is either a 1-, 2-, 3- or 4-component vector (yes, a 1-component vector is essentially just a scalar). In the sample code above, our cbuffer has two elements: a *4-component float vector* called `colorTint`, followed by a *3-component float vector* called `offset`.

The order here is important for a few reasons. The first is that we're not just declaring variables, we're actually defining where, in the actual GU buffer, these variables will get their data. If you think of a constant buffer resource – that chunk of memory on the GPU – as a big array of numbers, what we're saying here is the first four numbers in that "array" are the `colorTint`, and the next three numbers are the `offset`. We're essentially describing how we intend to interpret a bunch of numbers sitting in memory.



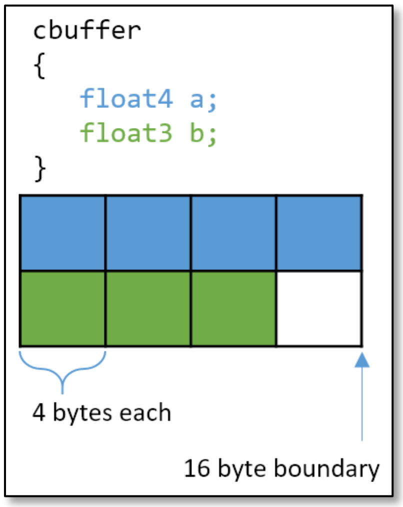
The second reason why the order of variables in our cbuffer matters is because we'll need to match that exact same layout over in C++. As mentioned earlier, the most efficient way to update data on the GPU is to send a big chunk of it all at once, rather than sending it one variable at a time. Given that, we'll need to put together a "chunk of memory" in C++ with the same layout as our cbuffer. This allows us to copy that data to the GPU with a single Direct3D command.

Luckily for us, creating a struct in C++ will allow us to arrange a set of variables in memory in the order we want. We simply need to ensure that our shader's cbuffer definition and our C++ struct definition always match; if we change one, we must update the other, or data won't be where we expect it to be.

> *At this point, you may be thinking "There has to be an easier way. This was easier in OpenGL. Why can't we just say 'hey shader, the variable* offset *now has the data (x,y,z)' from C++?" As it turns out, in Direct3D, there is no built-in mechanism for referring to a single shader variable from C++. We can build that functionality ourselves, but it doesn't exist directly within the API.*
>
> *This might seem like an oversight, but it all boils down to how Direct3D was designed: it provides access to GPU features, but it explicitly doesn't build much extra (potentially less-than-efficient) functionality on top of those features. OpenGL is different in this regard, though it also provides this kind of memory management: the equivalent to Direct3D's* constant buffer *in OpenGL is called a* uniform block*, and it works effectively the same way.*
>
> *So, if we want extra functionality, we have to build it ourselves. This is great for AAA devs who understand how to fully exploit a GPU, but is certainly less than ideal for those of us who are still learning the ropes. To that end, I'll be providing a set of helper libraries to simplify much of this in a future assignment, but I want everyone to understand the innards before we simplify them.*

The third reason why the order of variables in a cbuffer matters is due to *data packing*. When the HLSL compiler encounters a cbuffer definition, it may adjust exactly where the variables sit in memory according to its *packing rule*. This only affects the compiled code of our shaders; we won't necessarily know it is happening. (Note that this might not affect us during this assignment or even the next one, but it will eventually, so I want to at least mention it here.)

What is data packing? What does this actually mean for us? What is this rule? Let's start with the rule: *no vector defined in a cbuffer may cross a 16-byte boundary*. To put it another way, think of a cbuffer like a 2D array: each element is a number taking up exactly 4 bytes and each row is exactly four elements (16 bytes) wide. Each vector we declare in the cbuffer must fit within a single "row". See the diagram to the right for an example.

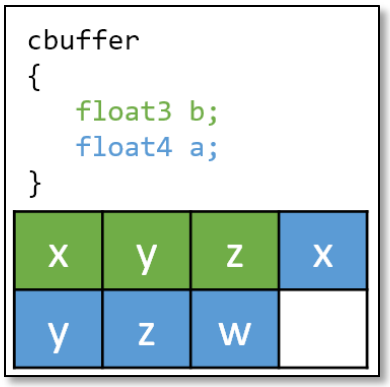


```
cbuffer
{
   float4 a;
   float3 b;
}
```

In reality these buffers are more like 1D arrays, but hopefully the illustration can help you visualize what's going on. In this example, the rule has **not** been violated, as each vector fits within a single "row" (it does *not* cross a 16-byte boundary).

What happens if we simply alter the order of the variables in the cbuffer? What would that do to the memory layout?

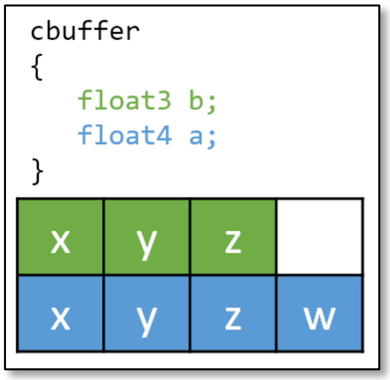
In the next example to the right, the two variables in the cbuffer have been swapped: the float3 comes before the float4. Mapping this out, we can see that the rule **has been violated**: the second vector straddles a 16-byte boundary (it doesn't fit nicely in a single "row").



```
cbuffer
{
   float3 b;
   float4 a;
}
```

What happens when this occurs? The HLSL compiler fixes this issue for us automatically, though it never actually *tells us* about it. Our code remains the same, but the compiled shader will have some padding between the two vectors, such that the rule is no longer violated.

Take a look at the last example to the right. Notice that the code hasn't changed, but after the compiler has its way with our cbuffer, the second vector has been shifted; the rule has been followed.



```
cbuffer
{
   float3 b;
   float4 a;
}
```

So, what's the big deal? Remember that we need to have a C++ struct that matches our cbuffer? If the cbuffer's layout is changed without our knowledge, then our C++ struct **won't actually match!** Data will not be where we expect it; the second vector will start earlier than expected. That's a pretty big deal and can lead some late nights debugging shader code without a clue as to why our data appears malformed.

For now, stick to the order specified (float4, then float3) and you won't run into any issues. There are multiple ways of handling issues related to data packing, which we'll cover when we get more in-depth with shaders. If you have further questions at the moment, feel free to ask! This article on HLSL packing rules from MSDN has more examples as well.

Task 2 – Complete!

Ok, still with me? You should have added 5 lines of code (including curly braces) to VertexShader.hlsl and read more about cbuffers than you ever thought possible. Everything should still compile and run at this point, though nothing will look different yet.

Task 3: Using *cbuffer* Variables in HLSL

All variables declared inside cbuffers are considered to be in the global scope within a shader file. This means we can use those variables directly within shader code. Open VertexShader.hlsl for this task.

Start by altering the line that is setting `output.screenPosition`, such that the `offset` variable is **added** to `input.localPosition`. It'll look like this:

```
output.screenPosition = float4(input.localPosition + offset, 1.0f);
```

If you compile and run your project, everything should still look the same. Even though we're adding to the of the vertices, `offset` doesn't have any data yet, meaning it's effectively (0,0,0).

Next, alter the line that sets `output.color` such that `input.color` is **multiplied** by `colorTint`:

```
output.color = input.color * colorTint;
```

You should notice a stark different after compiling and running now: all of your geometry is black! Similar to `offset`, the `colorTint` variable is effectively (0,0,0,0), causing the output color to also become black. That's fine for now.

Once we set the buffer up properly from C++, these variables will have data and things will look better.

Task 4: Defining the C++ Struct

The next step is to define a C++ struct that matches the layout of our shader's cbuffer. Start by creating a new header file (no .cpp file necessary) in your project named something like "BufferStructs.h". Though we'll only be using these in the Game class for this assignment, we'll be adding more eventually.

At the top of your new header file, add the following #include for the math library we'll be using:

```
#include <DirectXMath.h>
```

The DirectX Math library contains structures for vectors, matrices and quaternions, as well as the functions to manipulate them. We'll talk more about DirectX Math in class soon. All of DirectX Math is in the DirectX namespace, though remember that it's bad form to put using statements in header files, so be sure you're prepending the data types with "DirectX::" for this task.

Next up is the struct definition itself. You can name the struct and its variables anything you'd like, as they do not have to match any identifiers in shader. The only thing that must absolutely match the shader's cbuffer is the order and type of the variables. Create a struct that looks like the following:

```
struct VertexShaderExternalData
{
	DirectX::XMFLOAT4 colorTint;
	DirectX::XMFLOAT3 offset;
};
```

That's it for task 4. You'll be using this struct in tasks 5 and 6.

> *If our shader had two separate cbuffers, we'd need two structs. If our vertex shader and our pixel shader each had two distinct cbuffers, we'd need four structs in C++. As you can imagine, this can quickly get out of hand.*
>
> *We're effectively defining how our data will be laid out for two different processers: our CPU and our GPU. This requires doing some of the same work in both C++ and HLSL. A more dynamic solution would certainly be preferable, though it'd take a while to build ourselves. I'll be providing one in a future assignment. For now, we'll keep things relatively simple.*

Task 5: Creating a New D3D Resource: The Constant Buffer

At this point you've set up how you intend to use a constant buffer within a shader (the cbuffer definition). Now it's time to actually create that constant buffer resource in GPU memory. This will be remarkably similar to creating a vertex buffer or an index buffer, as a constant buffer is the same kind of Direct3D resource: an `ID3D11Buffer`. This means you'll be making a variable to hold the resource, filling out a description of how we intend to use it and then asking Direct3D to create it. The key difference will be with how we fill out the description.

Make the Variable

Start by going to the Game.h header file and making a private variable to hold a `ComPtr<ID3D11Buffer>`. Name it something like "`vsConstantBuffer`".

Calculate an Appropriate Size

The actual creation of the resource should happen at startup, so head to the end of the `Init()` method in Game.cpp. Here is where you'll fill out the buffer description and call `device->CreateBuffer()`.

Before any of that, however, you'll need to do a little math to get the size of the buffer correct. Vertex and index buffers have no constraints on their size, but constant buffers do: the byte width of a constant buffer must be a multiple of 16. This matches up with that whole "16-byte boundary" rule mentioned back in Task 2.

So, the byte width of our buffer needs to be a multiple of 16. It also needs to be greater than or equal to the size (in bytes) of the struct we defined in Task 4. There are several ways of handling this dynamically, as we'll have to repeat this for each struct, but we can do it pretty concisely with some simple math. Here's an example in action, though feel free to experiment with other ways if you'd like:

```
// Get size as the next multiple of 16 (instead of hardcoding a size here!)
unsigned int size = sizeof(VertexShaderExternalData);
size = (size + 15) / 16 * 16; // This will work even if the struct size changes
```

Adding 15 ensures that we either go past the next multiple of 16, or if `size` is already a multiple, we *almost* get to the next multiple. The integer division tells us how many 16's would fit (ignoring the remainder). Then we finally get back to a *multiple* of 16 with the multiplication step.

Describing the Constant Buffer

Now that we have an appropriate *size* calculated, we can fill out the description, like so:

```
// Describe the constant buffer
D3D11_BUFFER_DESC cbDesc    = {}; // Sets struct to all zeros
cbDesc.BindFlags            = D3D11_BIND_CONSTANT_BUFFER;
cbDesc.ByteWidth            = size; // Must be a multiple of 16
cbDesc.CPUAccessFlags       = D3D11_CPU_ACCESS_WRITE;
cbDesc.Usage                = D3D11_USAGE_DYNAMIC;
```

There are several differences here from the previous assignment's buffer setup. First, we're telling D3D that we intend to bind this buffer as a *constant buffer*. Next, we're setting the CPU Access Flag to "*CPU Access Write*", meaning we intend to write to this buffer from C++ (but never read from it). Lastly, we're setting the usage to *dynamic*, which tells D3D that this buffer will be read from the GPU and written to by the CPU, probably fairly often (at least once per frame). Overall, these three settings give Direct3D enough information to smartly decide how best to create and manage this resource.

> *There are multiple ways of setting up a buffer like this and of filling it with data each frame. The way I'm describing in this assignment matches the best practices given in the 2005 GDC presentation by NVIDIA's John McDonald, specifically slide 19. I don't expect you to read these slides, but they're linked if you're interested.*

Creating the Buffer

The last step is to actually create the buffer. The major difference here is that we won't be setting any initial data. The data we send to a constant buffer changes often and is usually the result of changes within our game entities, camera and/or lights. Since we don't have any of that yet, and since we'll be overwriting the constant buffer with new data every frame, we can leave out the initial data.

Add the following code after describing the constant buffer:

```
device->CreateBuffer(&cbDesc, 0, vsConstantBuffer.GetAddressOf());
```

Compile and run your program to ensure there are no errors. Nothing will change on the screen, but do pay attention to the Output window in Visual Studio: if everything is done right so far, there shouldn't be any Direct3D warnings or errors listed in there.

Task 6: Sending Data to the Constant Buffer Resource and Binding it

First things first, be sure that you're including your "BufferStructs.h" header at the top of Game.cpp.

You've set up your vertex shader to use data from a resource on the GPU (a constant buffer). You've also created that resource. The last steps are to fill that resource with data, which generally happens every frame, and to bind it to the pipeline so the shader can see it. For this assignment, both of these steps will happen in the `Draw()` method of Game, though we'll relocate them in future assignments.

As these steps would be considered part of "setting up the pipeline", they need to occur *before* you actually tell Direct3D to draw anything. Head to the area of the `Game::Draw()` method, just after the "Frame Start" section (clearing the back buffer and depth buffer) but *before* actually drawing meshes.

Collecting Data Locally

Create a local variable of your new struct directly in `Game::Draw()` and fill out its variables, like so:

```
VertexShaderExternalData vsData;
vsData.colorTint     = XMFLOAT4(1.0f, 0.5f, 0.5f, 1.0f);
vsData.offset        = XMFLOAT3(0.25f, 0.0f, 0.0f);
```

Feel free to customize the numbers, though a few things to note: Colors are defined as 4 float values representing red, green, blue and alpha, where each value should be between 0.0 and 1.0 (think of them as being between 0% and 100%). Reminder that float literals need a lowercase "f" after them. This is a tint, not a color replacement, so it will be multiplied by each vertex's color. Alpha is ignored currently, so changing that won't do anything. As for the offset, the numbers should be very small (between -1 and 1), as everything is in "screen space". Next assignment, you'll replace this basic translation offset with a full transformation matrix.

Copying to the Resource

Now that you've collected all the data you intend to copy to the constant buffer, and that data has the same layout as the cbuffer in our shader, we're good to copy. There are technically a few ways of doing this, but only one that works with the options we've used when setting up this buffer. We need to ***map*** the resource, which will *lock* it (prevent the GPU from using it) and give us a temporary pointer to its memory. Once mapped, we can do a standard C++ `memcpy()` to very quickly copy our struct over. Then we ***unmap*** the resource, which *unlocks* it so the GPU can use it once again.

There isn't much room to get creative here, so your code will pretty much look like this:

```
D3D11_MAPPED_SUBRESOURCE mappedBuffer = {};
context->Map(vsConstantBuffer.Get(), 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedBuffer);

memcpy(mappedBuffer.pData, &vsData, sizeof(vsData));

context->Unmap(vsConstantBuffer.Get(), 0);
```

The `D3D11_MAPPED_SUBRESOURCE` is a struct that will hold, most importantly, a pointer to the resource's memory after mapping occurs. `Map()` takes the resource we're mapping – `vsConstantBuffer` – as well as a few parameters we can leave as null. It also takes `D3D11_MAP_WRITE_DISCARD`, which tells D3D that it can safely (and quickly) discard all data currently in the buffer. Then we simply `memcpy()` the entire C++ struct directly into the resource and `Unmap()` it.

Wait, why do we need to do this map/unmap stuff? Don't we have a pointer? Can't we just memcpy() to the address of our vsConstantBuffer variable? Good questions! The short answer is no, we can't just use that address, as it's not a GPU address.

Remember that, much like vertex and index buffers, the constant buffer is a just pointer to a C++ ID3D11Buffer object. That ID3D11Buffer object internally contains several pieces of information that the API needs, including the raw GPU memory address where the resource lives. However, we generally aren't allowed to access that GPU address. One reason for this obfuscation is that Direct3D may need to move the buffer around in GPU memory, meaning that the underlying GPU address could change at any moment. We know, though, that we do need that address to physically copy data from our system RAM to the GPU's RAM. What do we do?

The solution is that our application can request that the underlying GPU resource (the buffer itself) be temporarily locked so that Direct3D knows not to move or otherwise use it until we unlock it. Being able to map (lock) and unmap (unlock) the buffer lets us flag to Direct3D when we're directly using the address. Ideally, the code between calls to map() and unmap() is very fast, as locking the buffer will cause any GPU work that requires it to sit and wait! Once we unmap, Direct3D & the GPU can continue with their work. However, there's no guarantee that the address will be the same in the future, so we shouldn't save it for later; we'll need to map it again next time.

Compiling and running right now should cause no problems, though we still won't see any changes.

Binding the Constant Buffer

There's one more step: binding our constant buffer to the correct place in the pipeline. Essentially, we haven't "hooked up" our resource (GPU memory) to the cbuffer register in the vertex shader; in other words: the vertex shader doesn't know where to look for its variables' data.

This is easy enough to do with a single method call:

```
context->VSSetConstantBuffers(
     0,     // Which slot (register) to bind the buffer to?
     1,     // How many are we activating?  Can do multiple at once
     vsConstantBuffer.GetAddressOf()); // Array of buffers (or the address of one)
```

> *Notice that the method is called `VSSetConstantBuffers` (plural). It can be used to set more than one constant buffer for the vertex shader stage at once, which can cut down on API calls (more efficient!). That's why the last parameter is effectively an array of buffers.*

The first parameter is the register to which we're binding. Over in our shader, we defined the cbuffer to be bound to register **`b0`**. We need to match that `0` here or the shader won't find the data. The second parameter is how many buffers we're binding at once. If we're binding more than one at a time, each is bound to the next register numerically. Lastly, we specify either an array of buffers or the address of a single buffer (which is the same thing in C++).

Testing Things Out

If everything is done correctly, you should finally see the results on the screen:

- All of the colors are tinted red (or whatever color you specified as a tint)
- All of the geometry should be moved to right (or whatever direction you specified as an offset)

If you're able to see these results, you're done with this task! If not: double check each task above, toss me a message on Slack and/or swing by my office hours.

Task 7: Using your Shiny New User Interface

You've got a custom user interface. You've got a color and an offset used during rendering. I want to be able to change that color and offset with the UI while the program is running.

Make it happen!

Below are some tips to get you started. Note that you'll be augmenting or replacing these pieces in the next assignment, so keep this simple.

- You'll probably need variables for these pieces of data, as they need to persist across frames
- To edit a 3-component vector with ImGui, use the DragFloat3() function
- To edit a 4-component color with ImGui, use the ColorEdit4() function

Where could you go from here?

We are still pretty early in the process, so don't fret if this isn't exciting quite yet. If you wanted each mesh to have its own color and/or offset, you'd need to send new data to the constant buffer resource *between* each `DrawIndexed()` call, like so:

```
ChangeDataInConstantBuffer() // Map, memcpy, unmap
mesh1->Draw()
ChangeDataInConstantBuffer() // Map, memcpy, unmap
mesh2->Draw()
Etc.
```

This *could* be done directly here within `Game::Draw()`, though in the next assignment we'll be setting up a more robust architecture for exactly this step.

Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. Remember to follow the steps in the "Preparing a Visual Studio project for Upload" PDF on MyCourses to shrink the file size.