Assignment 6 –
3D Models, Materials & Shaders

Overview

Now that you can draw and control simple entities and move around in the 3D world, it's time to get more interesting geometry into your engines. To ease the burden of multiple shaders and constant buffers, you'll be integrating a set of helper classes called **SimpleShader**. You'll then **create a simple material system** that represents the shaders and other surface data required when drawing a mesh.

Next, you'll be **loading 3D models** from .OBJ files and using them in place of, or in addition to, the geometry already in your scene. Since you can already draw sets of triangles, this just boils down to a new way of filling up your vertex and index buffers.

While we won't be getting to lighting and shading until the next assignment, you'll be making your objects look a little more interesting now by creating a **new, custom shader**.

Task Overview

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

- ❑ Integrate the **Simple Shader** helper classes into your engine
- ❑ Create a **material class** that represent a set of shaders and other surface data
- ❑ Update your vertex format and **load meshes** from external .OBJ files
- ❑ **Reconfigure your shaders**, moving variables around and passing new data through the pipeline
- ❑ Create and utilize a **second pixel shader** of your own design
- ❑ Ensure you have no **warnings**, **memory leaks** or **D3D resource leaks**

Task 1: SimpleShader

SimpleShader is a set of classes I wrote (stored in a single .cpp file and a single header file) that simplify Direct3D/HLSL shader loading, input layout creation, constant buffer management and CPU → GPU data copies. In other words, it'll ease much of your existing and future work.

The following section describes how to make use of SimpleShader in your existing project.

> *If Simple Shader is so useful, why didn't we start with it? Good question! I used to start the semester with Simple Shader in the starter code. Over the years of teaching this course, however, one of the biggest pieces of feedback I got was that folks didn't really understand constant buffers and GPU data copying. Simple Shader hid so much of it that it felt like a mystery. These days we start at a lower level so we can get our hands dirty, then simplify.*

Getting the Files

The first step is to grab the source files – SimpleShader.cpp and SimpleShader.h – from GitHub: https://github.com/vixorien/SimpleShader. Copy the two files into your project directory and add them to the Visual Studio project. Be sure to **#include "SimpleShader.h"** anywhere this assignment asks you to use SimpleShaders.

Updating the Game Class to use SimpleShader

The Game.h header contains ID3D11VertexShader and ID3D11PixelShader ComPtrs. *Replace* those with `shared_ptr<SimpleVertexShader>` and `shared_ptr<SimplePixelShader>`. You can also *entirely remove* the ID3D11InputLayout object, as it will be handled automatically by SimpleVertexShader.

`Game::LoadShaders()`

Delete **all** of the code that is currently in Game::LoadShaders() and replace it with the following (updating variable names and/or pointer management as necessary):

```
vertexShader = std::make_shared<SimpleVertexShader>(device, context,
	FixPath(L"VertexShader.cso").c_str());

pixelShader = std::make_shared<SimplePixelShader>(device, context,
	FixPath(L"PixelShader.cso").c_str());
```

Our compiled shader files (.cso) are located in the same folder as the executable by default, so the above code is using my FixPath() helper function to ensure we're looking in that folder. Since the Direct3D helper functions for loading shaders under the hood require wide characters, the string literal must be preceded by an **L**.

Note that SimpleVertexShader reverse engineers an ID3D11InputLayout from the given vertex shader and sets it when activating the shader, meaning this eliminates manually creating/using an Input Layout ourselves.

`Game::Init()` and/or `Game::Draw()`

Remove any calls to `VSSetShader()` and `PSSetShader()`, which manually activate Direct3D shaders, in `Game::Init()`, `Game::Draw()` or elsewhere. You'll replace these in task 2.

Testing

You won't be able to visually test these changes just yet, as you won't really be utilizing the SimpleShaders until task 2. However, your code should compile at this point.

Task 2: Materials

Materials define how the surface of a particular mesh looks when it's drawn: the shaders, the textures and various other drawing-related data. Your task is to create a simple *material* class that will begin to wrap this kind of data, so that a single material definition can be shared by multiple game entities. Remember that we only want to load resources, like shaders, once and share them among various materials and entities.

For this assignment, your material class will need, at a minimum, **three fields**: a single XMFLOAT4 for color tint and shared_ptrs to a SimpleVertexShader and a SimplePixelShader object. Each of these should be passed in to the **constructor** of the material, stored in corresponding fields and have appropriate **Get** and **Set** methods.

> *Note: Direct3D objects use ComPtrs, but SimpleVertexShader and SimplePixelShader are not COM objects, so don't use ComPtr! Use shared_ptr instead (or raw pointers, if that's your jam).*

Creating Materials

You only have one of each shader in your project at this point, so the material class may seem like overkill. You'll be making a second pixel shader soon; the material class is laying groundwork for that.

Make at least three Materials in Game::Initialize() – shared_ptrs again – *after* the shader loading code but *before* the code that creates entities. Depending on how you structured your code so far, this may require a little reorganization.

Blue, Red, and Yellow Cubes Under a Standard Desk Lamp Light Source

Blue, Red, and Yellow Cubes Under a Red Lamp

Blue, Red, and Yellow Cubes Under a Blue Lamp

Blue, Red, and Yellow Cubes Under a Yellow Lamp

Each material will have the same vertex and pixel shaders for now, but each should have a different color tint. You'll be connecting these to various GameEntities next.

A Note on Color Tints

Color tinting is achieved by multiplying colors. More specifically: multiplying their red, green and blue channels independently. Therefore, a blue tint (0,0,1) on a red object (1,0,0) results in black: (0,0,1) * (1,0,0) = (0,0,0). It's rare in real life to see a room with pure red objects and only a single, perfectly blue light source, but it would look much the same.

See this image for a real-world example →

Please choose color tints that still allow me to see your objects!

Connecting Materials & Entities

Now that you can make materials, edit your GameEntity class to keep track of which material it will use. This is a pretty straightforward update: keep a shared_ptr<Material> as a field and require it in the constructor. Add appropriate Get() and Set() methods, too.

Update your game entity creation code by passing a material to each one.

Activating the Shaders

Up until now, our code has assumed we'll only ever have one set of shaders. The vertex and pixel shaders were set once, since they were appropriate for every single entity. This, however, isn't necessarily true anymore, as some materials might contain different shaders (which will happen by the end of this assignment). This means you'll need to swap which shaders are active before drawing *each entity*, since each entity's material might use a different vertex and/or pixel shader.

Where exactly you activate the shaders for each entity depends on where you're currently drawing those entities. If your GameEntity class has its own `Draw()` method that does the majority of the drawing work, then that's a fine place to activate the shaders (before actually drawing the mesh):

```
material->GetVertexShader()->SetShader();
material->GetPixelShader()->SetShader();
```

If, however, you have all of your Direct3D render calls in a big loop in `Game::Draw()`, then that's where you'll need to get the current entity's material and perform the aforementioned steps.

```
entity->GetMaterial()->GetVertexShader()->SetShader();
entity->GetMaterial()->GetPixelShader()->SetShader();
```

There are many other ways of organizing exactly when and where these steps occur. In any case, they all need to happen right before you tell Direct3D to draw an object. For instance, you could have a method called PrepareMaterial() that takes a Camera and is in charge of setting all shader data in addition to activating the shaders. This isn't necessary for this assignment, but feel free to refactor in the future if your project is starting to feel a bit messy.

An advanced engine would offload the material and shader preparation to a Renderer class. The Renderer would even go so far as sorting objects by material (or by mesh/material combinations) so it doesn't have to swap shaders and other GPU resources between each draw call.

Preparing Data for the GPU

Previously, you had to create a struct and fill it with data before you could memcpy it to the GPU in one swoop. SimpleShader handles all of that for you now, allowing you to directly refer to shader variables from C++.

Head to where you're filling your VertexShaderExternalData struct with various matrices and a color tint – probably directly in Game::Draw() or a Draw() method in GameEntity. Replace the struct code entirely with code similar to the following example. Adjust the strings to ***match the names from your cbuffer*** in VertexShader.hlsl and any other implementation details as necessary:

```
std::shared_ptr<SimpleVertexShader> vs = material->GetVertexShader();

vs->SetFloat4("colorTint", material->GetColorTint());    // Strings here MUST
vs->SetMatrix4x4("world", transform.GetWorldMatrix());   // match variable
vs->SetMatrix4x4("view", camera->GetView());             // names in your
vs->SetMatrix4x4("projection", camera->GetProjection()); // shader's cbuffer!
```

CPU to GPU Copy

The code that mapped the constant buffer, memcpy'd the struct and unmapped? Replace it all with:

```
vs->CopyAllBufferData(); // Adjust "vs" variable name if necessary
```

Removing the Leftovers

By this point, Simple Shader should be handling all of the constant buffer wrangling for you. This means you can (and should) **remove** the following unnecessary code:

- The constant buffer resource, probably named vsConstantBuffer, declared in Game.h
- The creation and setting of that buffer in Game.cpp
- The setting of an Input Layout, probably in Game::Draw()
- The entire BufferStructs.h file itself, as you won't need it anymore

Testing

Test your program **before moving on**! You should still see the meshes from previous assignments, though some will probably have different color tints now (as denoted by their materials).

Task 3: Loading 3D Models

.OBJ Files

As we discussed in class, .OBJ files are one of the easiest model files to handle because of their simplicity and the fact that they're text-based. You can read more about the file format on Wikipedia, or refer to the lecture slides from class. On MyCourses, you'll find an Assets folder under Individual Assignments. It includes a zip file of various, simple .obj models for use with this and future assignments.

.OBJ files generally contain 3 useful pieces of information per vertex:

- Vertex position in 3D space
- UV coordinate (for texture mapping)
- Normal (mostly used for lighting)

.OBJ files also contain information about the faces (triangles) of the 3D model. The vertices of these faces are comprised of combinations of the above pieces of data. OBJ files can get fancier than this, but your engine only needs to support these to add lighting and textures in future assignments.

Updating your Vertex Definition and Vertex Shader Input Struct

Before actually loading an .OBJ file, you'll need to make sure your vertex definition is compatible. Go into your Vertex.h file and adjust the Vertex struct as follows:

- **Remove** the XMFLOAT4 color
- **Add** an XMFLOAT3 normal
- **Add** an XMFLOAT2 uv

This will break your existing code for the time being. You'll need to adjust *any and all* vertices you've created in previous assignments by removing the colors, then adding normals and uv coordinates. For right now, the normals can be (0,0,-1) to point at the camera, and the uv's can be (0,0). Alternatively, you can entirely remove your old meshes, as you'll be loading 3D models now.

If you recall, the input to a vertex shader must match the layout of the vertex buffer. This means you'll need to also update the VertexShaderInput struct defined in VertexShader.hlsl. Do this by *removing* the float4 color, and *adding* both a float3 normal (with the NORMAL semantic) and a float2 uv (with the TEXCOORD semantic). As this isn't a cbuffer, no need to worry about alignment or padding.

Order matters! Make sure your shader struct *exactly* matches your newly-updated C++ Vertex struct.

Since you no longer have a per-vertex color value coming into the vertex shader, you'll need to change the `output.color` assignment in VertexShader.hlsl. For now, set `output.color` equal to the color tint.

> *Note: It would be better to remove color tint entirely from the vertex shader and instead give it directly to the pixel shader. That's where we ultimately need it, and passing it VS -> PS just adds Rasterizer work. That's a task for later, however, as it involves a whole new constant buffer.*

Updating the Mesh Class

Your Mesh class is capable of handling *any number of vertices*, including vertices with *3D positions*. In other words, it can already handle 3D models! You just need to get that data into the buffers.

Your existing Mesh class will be responsible for loading 3D models in addition to its existing tasks. Create a second constructor that accepts the name of a file to load (a `const char*` parameter) along with the Direct3D device. This will allow you to either pass in data for the mesh using the old constructor, or simply provide a filename to load through this new constructor. Inside this new constructor, use the provided model-loading code (see below) to read the file and store the data in several temporary `std::vector` objects. Then it's up to you to use that data to create new Direct3D Vertex and Index Buffers – steps you've done before.

On MyCourses, you'll find a file called "Assignment 6 - OBJLoader.txt". This file contains the C++ code you'll need to copy/paste into your new Mesh constructor. This code works, but it is just a snippet (not an entire method). Copy and paste it into your new mesh constructor, adjust the *filename* variable to match your constructor parameter and then utilize the "verts" and "indices" variables at the end of the code to create vertex and index buffers for this particular Mesh.

It may be useful to first move your existing *Direct3D buffer creation code* to a helper method that takes the same data as the original Mesh constructor. This way, both constructors can simply call this method (rather than duplicating that code over and over). If you do this, use `&verts[0]` and `&indices[0]` to pass the address of the first element of each std::vector to this new helper method.

Getting Models onto the Screen

Copy the provided example .OBJ files to a folder in your project. There are several models provided:

- Cube
- Cylinder
- Helix
- Quad (square made of 2 triangles)
- Quad_Double_Sided (visible from both front and back)
- Sphere
- Torus

While you *could* drop them directly into the x64/Debug folder, it's cleaner to instead create an "Assets" folder (with sub-folders for "Models", "Textures", etc.) in your project's root directory. You can use the provided FixPath() helper function to ensure your relative paths begin at the program's executable file, then go up two folders, like so:

```
std::make_shared<Mesh>(FixPath(L"../../Assets/Models/sphere.obj").c_str(), device);
```

Testing

Test by loading several 3D meshes and using them with your existing entities (or making more). All of your other drawing code can remain the same; you should see these 3D meshes on screen.

Other Formats?

The OBJ loading code I've given you is exceedingly simple. In fact, it won't handle every OBJ file you might find online. Below are some options if you want a more robust OBJ loader or the ability to load other 3D model file formats. These are **not required** for this assignment, but I want you to know what other options are available. You might find them useful for the project later this semester.

TinyOBJLoader: https://github.com/syoyo/tinyobjloader
An open source OBJ model loading library.

The Open Asset Importer Library: http://assimp.org/
Open-source library to load most 3D model formats, including OBJ and FBX. (Yes, that's really the URL.)

Task 4: Update Shaders

Now that you have SimpleShader at your disposal, you'll update your existing shaders to pass some new data *through* the pipeline and pass other data *directly to* the pixel shader. Much of this is setup for future tasks and assignments. Here is the general outline, with some hints for each step.

- Move the colorTint global shader variable from the vertex shader to the pixel shader
  - This requires creating a new cbuffer in the pixel shader with just a colorTint variable
  - The name and register can be the same as the vertex shader's cbuffer (as each shader stage has its own set of registers)
  - Adjust any code in the pixel shader that is using the colorTint so it uses the new variable

- Update VertexToPixel to remove colorTint and add a UV coordinate
  - Do this in both the vertex and pixel shaders
  - The UV coordinate is a float2 and should use the semantic TEXCOORD

- Ensure the vertex's UV coordinate is sent through the pipeline (returned from the vertex shader)
  - This one's easy: `output.uv = input.uv;`

- Update C++ to support the new location of the color tint variable
  - It's in the pixel shader instead of the vertex shader now
  - Don't forget to call CopyAllBufferData() on the SimplePixelShader, too!

Why are we passing the UV coordinate down to the pixel shader? Eventually you'll need UV coordinates for texture mapping. However, for this assignment, it gives you a set of numbers that will be different for each pixel of a 3D model (since the Rasterizer interpolates the data across the face of each triangle). This may be quite useful for the next task, and the data is already sitting there. We might as well use it!

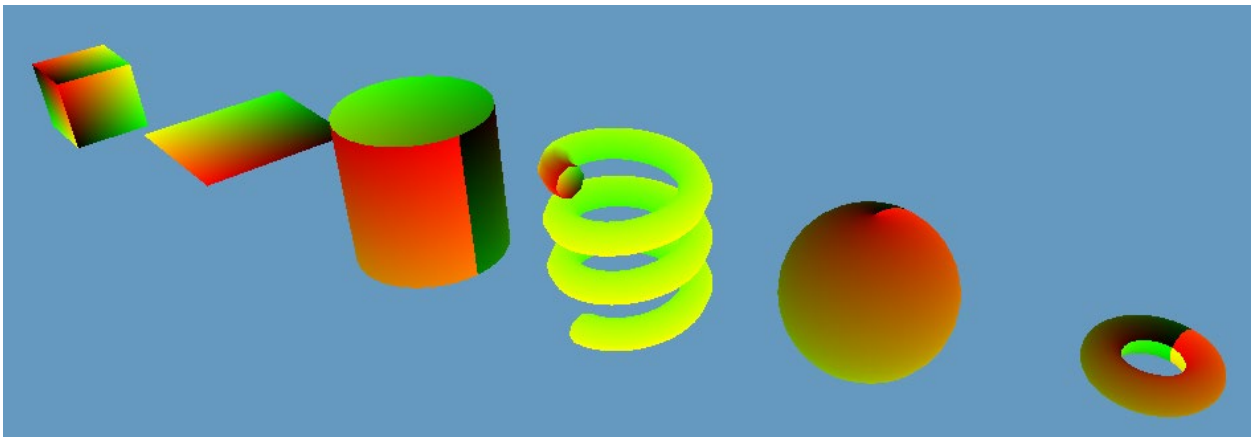
What about the vertex normal? It's not as necessary yet, and it needs to be adjusted before it's in a truly useful form. The next assignment, Lighting, will make use of it.

Test your program again. Everything should look the same as the last task. You should also check the *Output window in Visual Studio* while your program is running to see if you're getting any Direct3D errors or warnings. If so, you'll need to fix those issues before moving on. The error messages should be somewhat useful, but feel free to reach out if you're having trouble deciphering them.

To test the UV coordinates, you can temporarily return them as if they were a color, like so:

```
return float4(input.uv, 0, 1); // Adjust for your variable names
```

If the UV coordinates are getting to the pixel shader correctly, your models will look similar to the following examples. Remove this line once you're sure the data is correct.



Temporarily returning various shader data as a color is a great way to do quick & dirty visual debugging.

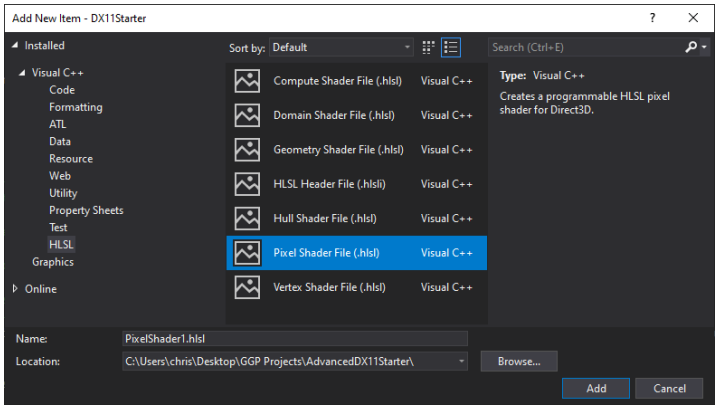
Task 5: Create a Second, Custom Pixel Shader

Your job for this last task is to create a second pixel shader of your own design, load it and apply it to several of the objects in your scene. This shader should create an interesting, procedural pattern or animation across the face of each object to which it's applied.

Creating a New Shader

The first step here is to create the new shader. You'll start it out with the same code as your existing pixel shader and adjust from there.

To create a new shader file in Visual Studio, right click within your project in Solution Explorer, choose Add > New Item, HLSL on the left, Pixel Shader in the center and name it something appropriate. `CustomPS.hlsl` is fine for now.

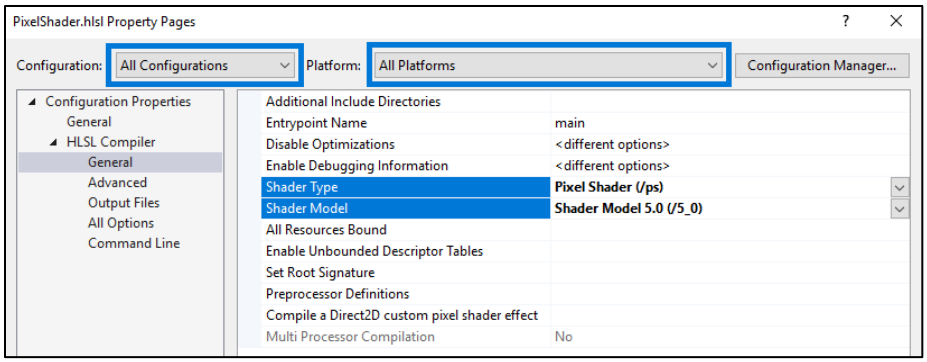


Setting Compiler Properties

You'll need to set proper compiler options for *every* new shader you add to your project. In Visual Studio's Solution Explorer, **right click on a new shader** you've added and choose "Properties".

First, check the current **Configuration** and **Platform** settings (the two dropdown boxes at the top of the window). Ideally, you want to be changing compiler properties for *All Configurations* and *All Platforms*. Change those dropdowns accordingly (see the screenshot below).

> *If you skip this step, you might only be setting options for Debug mode. This means if you switch to Release mode, your shader settings will be incorrect (and they might not compile).*



Now that you're setting options for all configs and all platforms, expand "HLSL Compiler" > "General".

Change the **Shader Type** option to the appropriate value (Pixel Shader for this one). If you don't set this, Visual Studio has no idea what kind of shader you've included and will just assume it's a vertex shader. As each shader type has different requirements for input and output, this won't work if your shader isn't actually a vertex shader.

Lastly, change the **Shader Model** option to Shader Model 5.0, which is the DirectX 11 shader version.

Copy the Old Shader and Apply

Copy all of the code (cbuffer, structs and main) from PixelShader.hlsl into this new shader. That'll give you a starting point you know works. Remove the code inside main() in this new shader and simply return a solid color for now:

```
return float4(1, 0, 1, 1); // Returns purple (or another color of your choice)
```

Set up your Game class to load this new shader (using SimplePixelShader), create a new material with it and apply that material to one or more of the game entities in your scene. Ideally this will be applied to entities that use your new 3D models, but feel free to experiment with others.

Test! Those entities should now show up as solid purple (or whatever color you chose).

Make it Interesting

The very last step is to make this shader do something interesting! Since we don't have textures or lights yet, you'll essentially be procedurally creating some sort of pattern across the face of each model. Note that you should ***not*** attempt to implement lighting equations here! That's the next assignment.

To help get you started, here is a reminder of the data you currently have available:

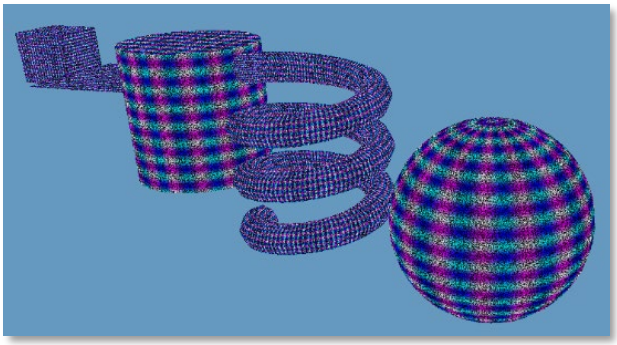
- The actual pixel coordinates, like (200, 300), of this pixel, from input.screenPosition.xy
- The UV coordinates of this pixel from input.uv (usually 0-1, but can go higher)
- The colorTint, which is the same for every pixel, but may be different for each material

If you want your pattern to change over time, you'll need access to a value that changes every frame. The totalTime variable over in Game.cpp's Update() and Draw() would be perfect here. This would require an extra cbuffer float variable, and an update to C++ to pass totalTime to the shader.

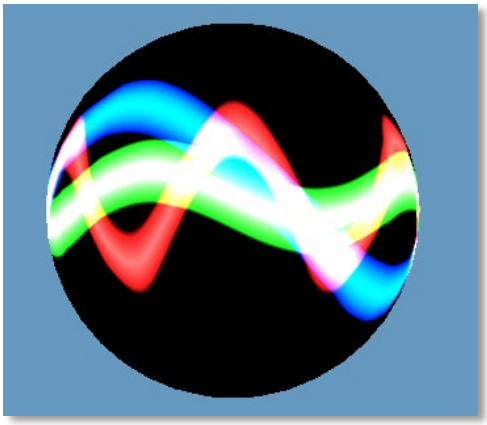
Examples

So…what should you do? As the goal is to experiment with shaders, the specifics are up to you. You could poke around on https://shadertoy.com for inspiration, though remember that most examples there are far outside the scope of this assignment (and many of them use features custom to that site).

To the right you'll see an example of a shader that layers a colored sin/cos pattern on top of a noise function. While you can aim to mimic this to start, yours should look different by the time you're done.



This example uses the model as a "canvas" of sorts. We know the U coordinate goes around the sphere, while the V goes from the top to the bottom. Thus, we have our 2D coordinates for our "canvas" that wraps around the model. The U value is used to create a sine wave, and the V value is compared against the sine's value to determine how intense the color should be. The is done three times, with various offsets and colors for each, and added together.



Ideas

Here are some ideas to get you started:

- You can treat the pixel's uv coordinate as a **position on a 2D canvas**
- Create one or more variables in C++ that are **editable with ImGui**, then send those values to the shader every frame. This would allow you to control your shader "live" (as the program runs).
- Create a simple **sin() or cos() pattern** using the uv coordinates (or other input)
- Generate **multiple separate patterns** at different scales and add their results together
- Create a **time-based animation** by incorporating totalTime as stated above
- Use math or if statements to create a **repeating pattern** like bricks or another geometric shape
- Implement a **noise generation function** like Perlin or Simplex noise
- Or use this *very simple* **pseudo-random function** for noise, which takes a 2D vector as input

```
float random(float2 s)
{
   return frac(sin(dot(s, float2(12.9898,78.233))) * 43758.5453123);
}
```

Odds & Ends

.OBJ Models & Git

Most Git repositories use a .gitignore file to keep unnecessary compiler and IDE files out of the repo. This is great and you should always do this! However, it just so happens that one of the C++ compiler files that we usually want to ignore has the file extension .obj.

See the issue here?

If you're using Git for your project, it will **absolutely ignore** all of your .obj 3D model files and never commit them! There are several potential solutions to this problem:

1. If all of your assets are in a folder (perhaps called Assets), you can set up the .gitignore file to never ignore that folder or its contents.
2. Another easy fix is to remove the single line in the .gitignore file that specifically calls out .obj files. Since the compiler's .obj files are almost always contained in another folder that *is* definitely ignored, this shouldn't impact your repo negatively at all.
3. Alternatively, you could rename the file extension of all of your 3D models to something else. (Changing a file extension doesn't change the contents of the file.) You would then just need to update any code that references the files to use the new extension.

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.