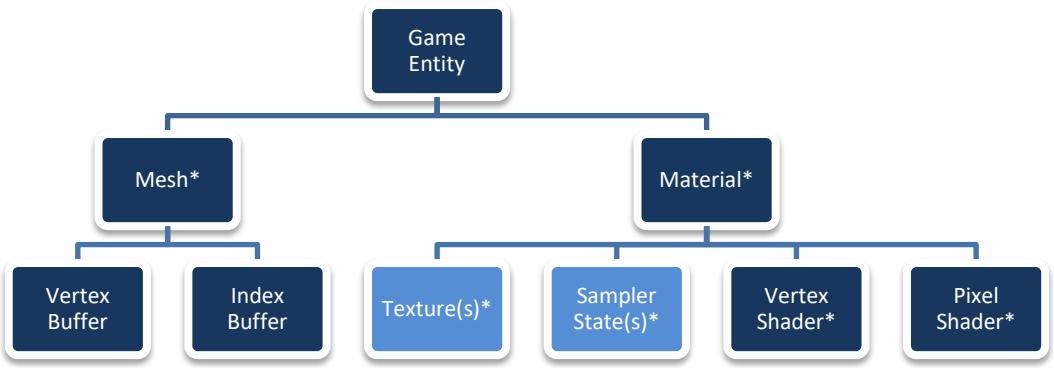
Assignment 8 – Textures

Overview

You now have game entities, meshes and materials working together with a camera system and lights. The last piece of a basic rendering engine is handling **textures**.



Task Overview

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

- ❑ Add the **DirectX Toolkit** to your project
- ❑ Load at least **two textures** and (eventually) share them among various materials
- ❑ Create at least one **sampler state**, which defines options for texture sampling
- ❑ Update your **material system** to support textures and sampler states
- ❑ Update your shader(s) to support **texture sampling**, using the result as a surface color
- ❑ Add an **extra texture-related feature** to your engine
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

Task 1: Include the DirectX Tool Kit

You'll need the DirectX Tool Kit for this assignment (and future assignments, and eventually your project), because it's the easiest way to load a texture in a format Direct3D understands.

Including DirectX Tool Kit through NuGet Package Manager

Check out the slides from class for instructions on including the DirectX Tool Kit through Visual Studio's NuGet package manager. I suggest using the "directxtk_desktop_win10" version of the tool kit. If you're not on windows 10, the "directxtk_desktop_2019" version should work identically.

Task 2: Texture Loading

Find Some Textures

Find **at least two images** to use for this task. While they could be anything, it's better to use textures that are appropriate for objects in 3D scenes – wood, bricks, grass, foliage, metal, etc. – instead of pictures of text or random memes. You can do a Google Image search for "game textures" and see what comes up, or you could trypolyhaven.com, ambientcg.com or textures.com (requires registration for 15 free downloads per day). There is also a zip with a few textures & specular maps on MyCourses.

Regardless of where you find textures, you might find that the site lists several different types of textures for each material, such as *diffuse*, *albedo*, *color*, *normal*, *height*, *roughness*, *metalness* and/or *AO*. You want the images labeled either **color**, **diffuse** or **albedo**, which are terms for the basic surface color. We'll be discussing some of the other texture maps in future classes.

Either way, grab at least two images and put them in an appropriate folder so your program can find them. PNGs are preferable since it's a lossless format but JPGs do work.

Loading Textures

To load a texture using the DirectX Tool Kit, you'll need to call the `CreateWICTextureFromFile()` function. It's defined in the "WICTextureLoader.h" header, and it is part of the DirectX namespace. There are two overloads for the function, one that takes the context and one that doesn't (among other parameters). You'll want to use the overload that takes **both** the device and the context as the first two parameters, as this overload will automatically generate mipmaps for you.

When calling `CreateWICTextureFromFile()`, pass in the following parameters:

- ID3D11Device*
- ID3D11DeviceContext*
- File path string (it's a wide string, so remember the "L" on the string literal)
- ID3D11Resource** - A reference to the texture (which we don't actually need) – use **0** or **nullptr**
- ID3D11ShaderResourceView** - Address of an SRV pointer, which we can send to a shader later

In your code, right before creating materials, try to load a texture. You'll first need to define a ComPtr<ID3D11ShaderResourceView>. This will be shared among multiple materials. However, since it's a ComPtr, you won't need to keep a reference to it in Game to clean up later.

To ensure the texture is actually loading, either capture the return value (it's an HRESULT), or put a break point after the call to `CreateWICTextureFromFile()` and check your SRV pointer. An HRESULT of S_OK or a pointer that isn't null (all zeros) means it worked.

If it didn't work, you should also be getting some Direct3D warnings in Visual Studio's output window with more information. Check the file path. Once this is working, be sure to load your *second texture*.

Task 3: Create a Sampler State

In addition to the texture itself, shaders require a sampler state. This object defines all the options used when the shader pulls colors from the texture, such as the address and filter modes. You'll need to create a ComPtr<ID3D11SamplerState> to hold this object. Much like the SRVs, this will be shared among your various materials.

Create a sampler state in your code, again before you create any materials, by first defining a local D3D11_SAMPLER_DESC variable. You'll need to ensure all of the members have a valid value. This means you should either set all of them yourself, or zero out the memory of the struct before setting just the parameters listed below.

The following members are required for texture mapping with mipmaps:

- **AddressU**, **AddressV** and **Address** W
  - Defines how to handle addresses outside the 0 – 1 UV range
  - You must set all three of these to something other than zero!
  - D3D11_TEXTURE_ADDRESS_WRAP is the most common value, allowing for tiling
- **Filter**
  - How to handle sampling "between" pixels
  - D3D11_FILTER_MIN_MAG_MIP_LINEAR is a common, but basic, filter (trilinear filtering)
  - D3D11_FILTER_MIN_MAG_MIP_POINT has no interpolation (like Minecraft)
  - D3D11_FILTER_ANISOTROPIC is the best option here, as it allows textures on surfaces to look good even at extreme angles.
    - If you go this route, you'll also need to set the sampler description's **MaxAnisotropy** to a value between 1 and 16 (higher is better but slower).
- **MaxLOD**
  - If this value is zero, mipmapping is effectively turned off
  - Use the constant **D3D11_FLOAT32_MAX** to enable mipmapping at any range

The last step in creating the sampler state is to call the device's `CreateSamplerState()` method, passing in the address of your description and the address of your ID3D11SamplerState pointer. You can verify this works by using a breakpoint again, or checking the return value (HRESULT) of the method. If it didn't work, you should again be getting some Direct3D warnings in Visual Studio's output window with more information.

Task 4: Adding Textures and Samplers to your Material Class

As materials define what a surface looks like, they need to keep track of texture(s) and sampler state(s) (since those define how we interact with the textures). In addition to the actual ComPtrs for these D3D resources, you also need to track the names of each texture and sampler on the HLSL side, as that's how you'll be activating (binding) them through SimpleShader before drawing:

```
// Example: Setting SRVs and samplers using SimpleShader
ps->SetShaderResourceView("SurfaceTexture", textureSRV);
ps->SetSamplerState("BasicSampler", samplerOptions);
```

However, unlike the above examples, you don't want to hardcode these string names into your code, as different shaders may use different textures (and, potentially, different *amounts* of textures).

So, you need a way to store an unknown number of data pairs (string and D3D object). Sounds like a *hash table* (or two) would work here. You might know these as Dictionaries in C# or associative arrays in other languages. They map one piece of data (a *key*, which is often a string) to another piece of data (a *value*, which can be any type). In C++, the templated version of a hash table is an *unordered_map*.

Include <unordered_map> in your material header and define two unordered maps in the header:

```
// Yes, these definitions are quite long.  But they'll do the job really well.
std::unordered_map<std::string, Microsoft::WRL::ComPtr<ID3D11ShaderResourceView>> textureSRVs;
std::unordered_map<std::string, Microsoft::WRL::ComPtr<ID3D11SamplerState>> samplers;
```

Adding Textures and Samplers

Next, you need a way of adding textures and samplers, one at a time, to a material. This can be done with some simple methods: AddTextureSRV() and AddSampler(). Each should take a string and a corresponding ComPtr, then add those to the matching unordered_map. The code for actually adding data to an unordered_map is quite concise:

```
textureSRVs.insert({shaderName, srv});
```

After creating materials in the Game class, you should add textures and/or samplers to those materials by calling AddTextureSRV() or AddSampler(). **Do this now.** The number and names you use depend on what's been defined in the material's pixel shader. The next task will use "SurfaceTexture" and "BasicSampler" as example names, so feel free to use those here or customize them.

Binding Textures and Samplers

The last step is to actually activate (or bind) these resources before drawing an entity. If your material has a method like PrepareMaterial() where it sets up data for drawing, you could do it there. Otherwise, it might be useful to create such a method, or one specifically for looping through the unordered_maps and setting the resources. In either case, the code will look something like this:

```
for (auto& t : textureSRVs) { ps->SetShaderResourceView(t.first.c_str(), t.second); }
for (auto& s : samplers) { ps->SetSamplerState(s.first.c_str(), s.second); }
```

Task 5: Texture Sampling in the Pixel Shader

To properly sample a texture in a pixel shader, you'll need UV coordinates. This task assumes you're already sending these into the pixel shader from a previous assignment.

The pixel shader will need a few more things. Define the following global variables near the top of the shader file (outside of the actual shader function and cbuffers). These will be set from C++ through SimpleShader. Feel free to adjust variable names as you see fit.

```
Texture2D SurfaceTexture  : register(t0); // "t" registers for textures
SamplerState BasicSampler : register(s0); // "s" registers for samplers
```

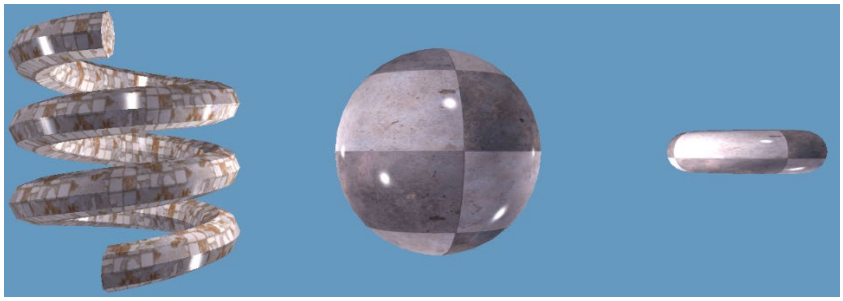
In the pixel shader function itself, you'll need to sample the texture and use the resulting color as part of your lighting equation. Sampling a texture is quite easy as long as you have the 3 required pieces: a texture, a sampler and uv coordinates. Each texture essentially has a `.Sample()` function, which takes a sampler and a float2. The function returns a float4: a color from the texture. You'll only need the first three components as we're not using alpha yet, so swizzle them out of the full color like so:

```
// Adjust the variables below as necessary to work with your own code
float3 surfaceColor = SurfaceTexture.Sample(BasicSampler, input.uv).rgb;
```

The color from the texture represents the surface's inherent color. It should be tinted by the material's **color tint**, then used with the scene's **ambient light** and finally shaded by other **lights** in your scene. You're probably already doing most of that; just be sure it incorporates this new surface color.

Assuming your materials actually have textures and samplers, and those resources are being set before drawing each entity, running your program now should result in textures on your lit objects. You may want to adjust your light colors to look better with your textures. If all else fails, use white lights.

This scene has 5 white lights (3 directional, 2 point) and uses the textures below



Task 6: Gettin' Fancy

For the last task, come up with **an extra texture-related feature** to add to your engine. Please add a ReadMe.txt file to your project that details exactly what you've chosen to do (so I don't miss it).

Some examples:

- The ability to arbitrarily scale and/or offset your textures.
  - This is done by scaling / offsetting uv coordinates on the fly in the shader.
  - The data should be sent in from C++ (don't hardcode it in the shader).
  - The actual scales and/or offsets are generally considered material properties.

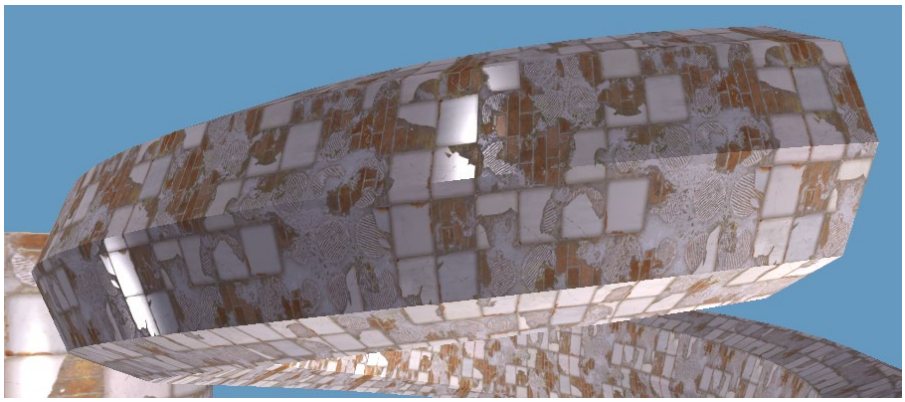
- Add the ability to use a specular map in your shader, controlling per-pixel shininess.
  - You would read a single color channel from a greyscale specular map.
  - Use that 0 – 1 value to scale the result of your specular calculation.
  - Note: Using this as a straight roughness replacement won't work well, especially for extremely low values. Definitely just scale the specular result.

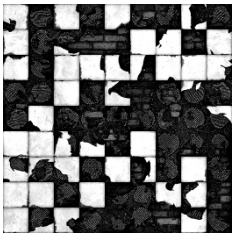
- Add the ability to use a second texture as an overlay or mask on the base surface color.
  - Sample this texture and use its color to modulate the surface color in an interesting way.

- Another idea of your choosing.

Example of extra features in action

This material uses a specular map (see below) to limit the shininess of each pixel.
It also scales the UV coordinates, allowing the texture to be repeated many times.





ImGui?

There is no explicit ImGui requirement for this assignment, though please don't remove your existing ImGui work! If you do want to incorporate it, it's entirely possible to display textures with ImGui.

ImGui has a function called Image() that takes a void pointer to your graphics API's texture view – in our case, that's a *shader resource view*. You can simply cast a raw ID3D11ShaderResourceView pointer (use .Get() on the ComPtr) to a void pointer and pass that as the first parameter to ImGui::Image().

Using this, you could display & control materials in the same way you handle entities or lights. Again, this is not a requirement for this assignment. Feel free to play around!

Looking Good!

At this point you have lit and textured 3D models rendered using custom shaders. That's technically enough to make your own 3D game. Not too shabby for half a semester!

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.

Seriously, this is important! The DirectX Toolkit can really bloat your project size, so be sure to remove the files as outline in the PDF mentioned above; the toolkit will be automatically re-downloaded from NuGet the next time you open and build your project.