Assignment 6 – Textures

Due by 11:59PM on Thursday, February 21st.

# Overview

You should have game entities, meshes and materials working together with a camera system and some lights. The last piece of the puzzle for these individual assignments is to implement texture mapping with at least two different textures. Luckily, as long as you’ve been keeping up with the assignments, this one should be relatively simple.

Your material system defines the surface of a particular entity when drawn. You’re going to be adding the ability to specify a texture to go along with the lighting of that surface, and a sampler state which defines how that texture is sampled. To do this you’ll need to include the DirectX Tool Kit (if you haven’t already) for texture loading. You’ll also be updating your Material class and shaders.

# Tasks

## Include the DirectX Tool Kit

You need the DirectX Tool Kit for this assignment (and eventually your group projects), because it’s the easiest way to load a texture in a format DirectX understands. You have two options for including the DirectX Tool Kit: Include it through Visual Studio’s NuGet package manager (the preferred method), or download and include the code and project yourself. You’ll only need to do one of these two methods.

**Including DirectXTK through NuGet Package Manager**

If you’d like to go the NuGet Package route, the DirectXTK wiki on GitHub has a [write-up of the steps](https://github.com/Microsoft/DirectXTK/wiki/Adding-the-DirectX-Tool-Kit). You’ll only need to perform the steps listed in the “NuGet package manager” section. Note that the DirectX\_2015 version is designed to also work for Visual Studio 2017.

**Including DirectXTK Yourself**

Download and unzip the [DirectX Tool Kit](https://github.com/Microsoft/DirectXTK). Inside, you’ll find several subfolders and a large amount of visual studio projects and solution files. The easiest thing to do is create a subfolder in your project’s directory called DirectXTK and copy the provided files and folders there. You only need to keep the project file that correspond with the version of Visual Studio you’re using (such as DirectXTK\_Desktop\_2017). Since you’ll be including the project in your own solution, you can delete or ignore the solution (.sln) files as well. They’re just examples.

Next, follow the instructions in the [DirectX Tool Kit Documentation](https://github.com/Microsoft/DirectXTK/wiki/Adding-the-DirectX-Tool-Kit) for “Project-to-project references” (scroll down a bit on the page to find this section). You do **not** need to add all of the headers as specified in the “Adding the headers” instructions, as you can simply add the individual header files to your code as needed.

**Note:** You will NOT simply be copying the code files to your own project. You must to add the entire provided project to your solution and set up the proper references if you go this route!

## Find Some Textures

Find a few (at least two) images to use for this assignment. They can be anything you want, but if you’re looking for some decent game-like assets you may want to try [textures.com](http://textures.com/) or [opengameart.org](http://opengameart.org). If nothing else, you can also do a Google Image search for “game textures” and see what comes up.

Either way, grab at least two images and put them in an appropriate folder so your program can eventually find them.

## Loading a Texture

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’s part of the DirectX namespace. There are two overloads for the function, one that takes the device context and one that doesn’t (among other parameters). You want to use the overload that takes **both** the device and the context as the first two parameters, as it 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) – pass in **0**
* 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 have an ID3D11ShaderResourceView pointer defined, which can be a local variable for now. You’ll be passing it to a material 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.

## Creating a Sampler State

In addition to the texture itself, a shader requires a sampler state to actually sample colors. The sampler state defines all the options used when the shader pulls colors from the texture, such as the address and filter modes.

Create a sampler state in your code, again before you create any materials, by first defining an ID3D11SamplerState pointer and a D3D11\_SAMPLER\_DESC struct 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 you need. 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 a usual value (wrapping textures)
* **Filter**
  + How to handle sampling “between” pixels
  + D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR is usual (trilinear filtering)
* **MaxLOD**
  + This should be larger than zero for mipmapping to work
  + Use D3D11\_FLOAT32\_MAX

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.

## Update the Material Class

Since the material class should define what a surface looks like, it should also keep track of a texture and a sampler state (since that defines how we interact with the texture). Add two more fields to the Material class: an ID3D11ShaderResourceView pointer and an ID3D11SamplerState pointer. Add the same kind of parameters to the constructor, as well as corresponding *get* methods. This all allows you to share these resources among multiple materials, as it doesn’t make sense to load the same resource multiple times.

For each material you’ve created in previous assignments, be sure to pass in the texture and sampler state you’ve made above before drawing with it. Depending on how you’ve set up your code, this could be happening somewhere in the material class itself, or the entity’s draw method, or even directly in Game.cpp.

Whenever you’re passing matrices into your material’s Simple Vertex Shader, you should also be passing the texture and sampler state to the material’s Simple Pixel Shader. This can be done with the following Simple Shader methods: SetSamplerState and SetShaderResourceView. They both take the name of the variable in the shader, and then the actual data to set.

## Update the Shaders

To properly sample a texture in a pixel shader, you’ll need UV coordinates. These coordinates come from the vertices, which means they’ll be available in the vertex shader. Make sure you’re passing them through to the pixel shader. Update your VertexToPixel structs in both shaders to include a float2 uv coordinate (with the TEXCOORD semantic).

In the **vertex shader**, simply copy the input’s UV to the output’s UV. No adjustments necessary.

The **pixel shader** will need a few more things. Again, ensure you’ve updated the VertexToPixel struct to match the vertex shader’s output. Define the following global variables in the shader file (outside of the actual shader function). These will be set from C++, through the Simple Shaders. Feel free to adjust the variable names as necessary.

* Texture2D diffuseTexture : register(t0);
* SamplerState basicSampler : register(s0);

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() method, which takes a sampler and a float2. The method returns a float4, which is a color from the texture.

// Adjust the variables below as necessary to work with your own code

float4 surfaceColor = diffuseTexture.Sample( basicSampler, input.uv );

The color you get from sampling the texture should be shaded by any lights in your scene. Multiply the result of each lighting calculation (directional, ambient, etc.) by the texture color.

Running your program now should result in textures on your lit objects.

## Not Bad

At this point you should have lit and textured 3D models rendering using custom shaders in a modern graphics API. Not bad for the first half of the semester. Once you begin forming your groups for the project, it will probably makes sense to look through each member’s code and choose the best implementations from each assignment.

# Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses. As usual, remember to ensure you have no warnings, memory leaks or DX resource leaks!