CSCI 441 - Lab 06
Friday, October 09, 2020
LAB IS DUE BY **FRIDAY OCTOBER 16 11:59 PM**!!

Today, we'll make our objects look pretty by adding textures to our objects. Please note that this lab has a high number of TODOs but they are finely broken down for each individual step that needs to be done.

Please answer the questions as you go inside your `README.txt` file.

Step 0 – Setup CMakeLists.txt & CLion

This project has two `CMakeLists.txt` files. The first is located at the top level and the second is located inside of the `src/` folder. Be sure to update the `include_directories()` and `link_directories()` appropriately in each file.

Additionally, in CLion be sure to set the Run > Edit Configurations > Working Directory to the parent .. for the lab06 target.

There are technically two targets in this lab (`lab06` and `stbimage`). Upon completion of this lab, you will have a stand alone STB Image Library that you can compile future programs against. This will help prevent redeclaration errors in your code. To use the STB Image library, you'll need to include `<stb_image.h>` and link against `stbimage`.

Step 1 – Applying a Texture Shader

Start off by building the existing code and run it. Wow, that's bright! It's all white. Do take note of what this lab gives you – a working arcball camera. Use the left mouse button to rotate around the scene. Press Control and then click to zoom in/out. You can also use the scroll wheel (if present) to zoom in/out. Press 1-8 to view different objects.

We need to make it a bit more colorful and apply some premade images to our objects. We'll go through the same process as last time by first setting up the shaders on the GPU then hooking up the CPU to support the shader. Generally, this is the process you will want to go through – write the shader first, then connect to the CPU to it.

Part I – The Vertex Shader

There's not much for the vertex shader to do beyond pass information through to the fragment shader. The three steps to be done inside the vertex shader are:

A. Add an attribute corresponding to the texture coordinate of the vertex
B. Add a varying output to pass the texture coordinate through to the fragment shader
C. Pass the attribute value through to the varying

You'll find these three steps waiting for you inside of `shaders/lab06.v.glsl`.

Part II – The Fragment Shader

The actual heavy lifting for texturing is performed within the fragment shader, but the heavy lifting is handled for you. There's really not much to be done in the fragment shader either.

D. Add a varying input to receive the texture coordinate from the vertex shader
E. Create a uniform to correspond to the texture map we should sample from. The data type for textures are special GLSL types. We will be working with 2D textures, so the data type needs to be `sampler2D`.
F. Perform the texture lookup and store the corresponding texel. We will make use of the texture() function within GLSL to do the lookup. This function expects two arguments (1) the texture map to sample from (2) the texture coordinate to sample at. This function returns a vec4 value corresponding to the RGBA data. Store the resulting texel value.
G. Assign the fragment color to be our texel color.

You'll find these four steps inside of `shaders/lab06.f.glsl`.

And that's the end of our shader work. To do standard texturing and applying it in a decal manner, those are the steps we need to complete. We could perform our illumination calculations from Lab05 and then modulate the texel color with the illumination color.

Step 2 – Connecting to a Texture Shader

The remainder of our work will be done in `main.cpp` and on the CPU. Begin by taking a walk through the code. It's been refactored slightly. If you look at the main() function, there are now three main steps:

- initialize()
  - Create our OpenGL context and setup EVERYTHING (GLFW, GLEW, OpenGL, shaders, buffers, textures)
- run()
  - Perform our draw loop, render the scene and update the objects in our world
- shutdown()
  - Free up all of the memory on the CPU & GPU

Each of the functions then walk through the various steps that occur, but this better matches the overall flow of our OpenGL programs. We need to do one time setup, run our program, and then clean up our memory.

Part I – Loading and Registering a Texture

The first thing we'll do is load the image data in to CPU memory and then transfer it to the GPU. We will create a helper function to perform this for us. It is already created and called loadAndRegisterTexture(). This function will handle a single texture and we'll be able to call it for every texture we wish to create.

There's already some starting steps for you. The reading of the image into CPU memory is completed for you. There is then some error checking to ensure the load was successful. We need to fill in the pieces to now send the image data to the GPU. Let's walk through the registration process:

01. Begin by generating a new texture handle. Use the function `glGenTextures()` and the already made `textureHandle` variable.
02. Now bind the handle to be active. Use the function `glBindTexture()` and set the target as `GL_TEXTURE_2D` since we are working with two dimensional textures.

03. Set the mag filter to be linear. Use `glTexParameteri()` to set the texture parameters.
04. Set the min filter to be linear.
05. Set the S coordinate wrapping to be repeat.
06. Set the T coordinate wrapping to be repeat.
07. Send the data to the GPU using the `glTexImage2D()` function.

The glTexImage2D() function takes a total of 9 (!) parameters. In order, here is the explanation of each parameter:

1) The texture target. We are still working with 2D textures, so use the same value we used in TODO #2.
2) The Level of Detail (Lod) this texture corresponds to. This is our base (and only) image so it will be level 0 (zero).
3) The internal format the image is stored in – or rephrased as the number of channels the image has. This must be one from a list of formats. The constant `STORAGE_TYPE` is holding the corresponding value you should use based on the number of channels our image has. *Note: this value will vary if you are using different image types and color spaces.*
4) The width of the image.
5) The height of the image.
6) The size of the border. Interestingly enough, this parameter is deprecated and no longer used but MUST BE 0 (zero).
7) The format of the pixel data. For us, this matches the `STORAGE_TYPE` we provided to argument three.
8) The data type of the data in the image data array we will send over. We are storing our image data in an unsigned char array. For the purpose of memory size, this is equivalent to `GL_UNSIGNED_BYTE`.
9) The actual data array!

This walks us through the process of registering a texture with the GPU and the function then returns the associated texture handle for this image. We now need to use this function.

First at TODO #08 we'll need to create a global texture handle to correspond to the image we are going to load.

Now in `setupTextures()` at TODO #09 call the `loadAndRegisterTexture()` function to load the `assets/textures/metal.jpg` image. Set the result of this function call to your global texture handle.

The texture's loaded to the GPU. Let's now get the locations from our shader so we can send the data to the correct spot in our shader.

Part II – Querying our Shader

We added two inputs to our shader program (1) the attribute for the texture coordinate (2) the texture map. We need to add global variables to store these locations. Add these at TODOs #10 and #11.

Now inside of `setupShaders()`, at TODO #12 look up the locations of your attribute and uniform.

We'll set one the texture map uniform at this point since it will never change throughout our program. At TODO #13 set the value for our texture map uniform. OpenGL can have multiple textures bound at once –

though we are not doing that yet, we're just using one. These are zero-indexed, so we'll tell the shader to load the first texture that is bound. This is accomplished by using the `glUniform1i()` function and passing a value of 0 (zero).

That's all the information we need from our shader. Now, we're ready to send texture data to the GPU to render our image.

Part III – Add Texture Coords to our VBO

Currently, our vertex data only contains information about positions. We need to add the texture coordinate data. This will need to occur in three places for our different objects. The first place we'll specify it is for the ground plane that we are drawing with our own quad and VBO.

Begin at TODO #14 to add the `s` and `t` components to our vertex data.

Now we need to specify the texture coordinate for each vertex. We'll orient the image with the quad. Provide the following texture coordinate for each corresponding vertex position:

Position	TexCoord
(-10, 0, -10)	(0, 0)
(10, 0, -10)	(1, 0)
(-10, 0, 10)	(0, 1)
(10, 0, 10)	(1, 1)

As the final step for our VBO, at TODO #16 connect the VBO to the Shader by setting up the vertex attribute. Our data is interleaved, each texture coordinate is made up of two floats, and begins at the fourth float in the array.

We're now sending the texture coordinate data over just for our ground platform. We need to do this for all the other objects we're drawing too. The CSCI441 objects library allows you to hook it up to a shader by specifying the attribute locations it should send data to. TODO #17 displays where we need to tell the library the texture coordinate location. Take note of the large block comment accompanying this function call for how you would use this library with multiple shaders.

We would now have the ground platform and our 3D shapes being textured. The only one that is left is our OBJ models – Suzanne! At TODO #18 send the texture coordinate location over.

At this point, we've completed the hook ups to our shader! We have specified the texture to apply and the texture coordinates for how it should be applied. Run your program and check out the metal objects in your world.

Step 3 – More Textures

We're going to expand the scene in two ways. The first will be to apply different textures to our ground and to our objects. The second will be to create our own quad and alter the texture coordinates to view the effect.

Part I – A Second Texture

If you looked inside of the `assets/textures/` folder, you'll have seen there was a second image living in there. Repeat TODOs #8 and #9 and read in this second image the same way you had done the metal image. Your `setupTextures()` function should now first read in the metal image and then second read in the Mines image.

Run your program at this point. Alright, a new texture! But everything is now all Mined up. Think through the process of registering our texture:

1. Generate a texture handle
2. Bind it to make it active
3. Set our parameters
4. Send our image data to the GPU

It's Step 2 that's so critical. When we are registering our textures, we bind them and the last texture read remains active and will be applied. In order to apply the appropriate texture to the corresponding objects, we need to bind the corresponding texture handle prior to drawing.

At TODO #19, bind the metal texture handle prior to drawing our ground platform. Then at TODO #20 bind the Mines texture handle prior to drawing any of our objects.

Run your program again and the ground should be metal while the objects are Mines.

Part II – A Second Quad

At TODO #21, you'll find commented out the starting block to draw another quad. This one is oriented in the XY-plane ranging from (-2.5, -2.5, 0) to (2.5, 2.5, 0). It also has the texture spanning the full quad. Uncomment the block where it says BEGIN/END. You'll need to repeat the TODO #16 step to hook up this VBO to our shader.

Run your program at this point and press 8 to see your quad spinning.

Let's now begin changing the texture coordinates for this quad. Currently both s and t range from 0 to 1. Change t to range from 0 to 3.

Run your program.

Q1: How does the quad look now? Why?

Change the s coordinate for the TR vertex to also be three and then run your program.

Q2: How does the quad look now? Why?

Lastly, ensure the s coordinate is also ranging from 0 to 3, just as t is. Run your program and it should look a bit cleaner.

Part III – Being Good Little Programmers

The final step to complete the texturing process is to kindly delete the textures off of the GPU when we are done with our program. TODO #22 puts this in the right spot. It works just the same as

`glDeleteBuffers()`, but we'll use the function `glDeleteTextures()` instead. Be sure to delete both the metal and Mines texture.

You can rerun the program at this point. You won't notice any visual difference, but you'll sleep well knowing you did proper memory management.

Step 4 – Copy the stbimage Library

To use the STBImage library in other labs/assignments/projects, you now have a completed stbimage library. To copy the library file, you'll need to dig a little bit. Inside of `cmake-build-debug/src` you will find `libstbimage.a`. Copy this file to where your lib/ files are. (For instance Z:/CSCI441/lib).

Now, whenever you wish to use stbimage you would `#include <stb_image.h>` and then link against `stbimage`.

Q3: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q4: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q5: How long did this lab take you?
Q6: Any other comments?

To submit this lab, zip together all your source code and README.txt with questions. Name the zip file <HeroName>_L06.zip. Upload this on to Canvas under the L06 section.

LAB IS DUE BY **FRIDAY OCTOBER 16 11:59 PM**!!