CSCI 441 - Lab 05
Friday, September 29, 2017

LAB IS DUE BY **MONDAY OCTOBER 9 11:59 PM**!!

Today, we'll be making our scene look pretty by adding textures to our objects.

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

This lab will make use of the SOIL (Simple OpenGL Image Library) library. We used this for A2, but it is included again with this lab.

Step 0 – Copy Libraries

Copy the CSCI441 v1.2 Library into your Z:/CSCI441/include directory. If you don't already have SOIL set up, copy it over to your include and lib directories.

Step 1 – Loading a Texture

Start off by building the existing code and run it. Hooray, it works!

We're going to load in a total of two textures. The first is a PPM, which is an ASCII text file so we can read and parse it. The first line is the format of the PPM, the second line is the size of our PPM (W H), and then there are W*H*3 rows where every three rows make up the RGB components of a single texel. The function `CSCI441::loadPPM()` is provided for you, you will need to reference it correctly. This is `TODO #1`: to load in "brick.ppm" located in the textures/ folder.

Now that we have an array of our brick data, we can register our image data with OpenGL. First for `TODO #2a`, call the `registerOpenGLTexture()` function with the proper arguments. Now fill in `TODO #2b` to actually register the image. This involves a series of steps and all the code is in the slides that you'll need. The OpenGL API for each method may be useful as well so you know what the parameters mean. The steps you need to complete are:

1. Enable 2D texturing
2. Request a handle for our texture
3. Bind the texture to `GL_TEXTURE_2D` so we can set the proper properties
4. Set our environment mode to `GL_MODULATE`
5. Set our Min & Mag filters
6. Set how to wrap S & T
7. Finally, register the texture with `glTexImage2D()`

The `glTexImage2D()` function returns a `bool` so you can add error checking to make sure the texture registered properly if you wish. Recall for these many parameters, the format is an RGB image. Our data is stored as a char* array. A char is equivalent to an unsigned int as far as memory size is concerned.

We needed to go through these steps because if you wish to make a transparent texture, then you'll need to read in the RGB data from one image (such as a PPM) and then the A data from a second image (such as a PPM) and then put together the RGBA data yourself.

Now we are ready to apply our texture!

Step 2 – Applying the Texture to the Teapot

This step is super easy since our Teapot has texture coordinates already! At `TODO #3a` in our `renderScene()` function, we need to bind a texture to use.

1. Make sure 2D texturing is enabled
2. Bind the texture for our brick texture

Compile and run.

Q1: How does the teapot look?
Q2: Do any other objects get textured? (Hit the 1-8 keys to see different objects)

When you view the partial disk (press key '8'), does anything happen to the grid? If you see it get darker, think about what steps have happened. We drew our axes with lighting disabled, we drew our grid with lighting disabled, and then we turn on the lights and texturing to draw our objects. The next frame, we draw our axes and grid with lighting disabled but the texture is still on. The texture is being applied to our 2D lines! Complete `TODO #3b` to turn off texturing AFTER we've finished drawing our objects.

Compile and run. Fixed?

Neat, that was too easy. Let's make our own object to texture now.

Step 3 – Applying the Texture to Our Own Object (Quad)

First, at `TODO #4a` create your own object. Draw a quad using a **triangle strip** so that we only have four vertices. Make sure that you consider winding order! Ideally have it aligned with the XY plane and square.

Compile and run.

Q3: Hit the '9' key to see your object. Look cool and boring?

Does it look dark? We set vertices, but we forgot normals! Go back and now for `TODO #4b` add normals to each vertex that match the surface normal based on winding order.

Compile and run. Brighter? Perfect.

Let's load a second image to texture this quad with. This time we'll load in the textures/mines.png image. This image format is a bit more complex and we don't want to worry about reading in the messy binary information.

There are many libraries available to load image files, we will use the Simple OpenGL Image Library (SOIL). [Alternatives exist, such as the Resilient Image Library (ResIL) so if you plan on using non-common images then perhaps check out ResIL – but SOIL will meet our needs just fine.]

At `TODO #5`, load in the image using SOIL and it will return to you the handle. Usage information for SOIL can be found on their website (http://www.lonesock.net/soil.html). The call we are most interested in is:

```
GLuint texHandle =
     SOIL_load_OGL_texture(
          "pathToImage",
          SOIL_LOAD_AUTO,
          SOIL_CREATE_NEW_ID,
          SOIL_FLAG_MIPMAPS
               | SOIL_FLAG_INVERT_Y
               | SOIL_FLAG_COMPRESS_TO_DXT
     );
```

The arguments to this function are the image path, tell SOIL to register the texture, create a new texture handle to use, and then create mipmaps, invert the y axis (the image would come in upside down otherwise), and compress the image. This function will also register our texture with OpenGL. But it does not set any of our texture properties for us.

Now at `TODO #6` set the texture parameters for S, T, Min & Mag filters. Go back to `TODO #3a` and change the texture being applied to this Mines texture.

Compile and run.

Q4: Is the teapot now Mined up?

Good, now we'll put it on our quad. Back at `TODO #4c`, we need to specify texture coordinates for each vertex to map to. This is done through use of `glTexCoord2f( s, t )`. Just as we specified a normal for each vertex, we'll need to specify a texture coordinate for each. Map each vertex to the four corners of our image, that is from (0,0) to (1,1).

Compile and run.

Q5: Do you see the Mines logo on your quad? How is it oriented?

Rearrange the texture coords so that the image is right side up and readable.

Now change the texture coords to range from (0,0) to (2,2);

Compile and run.

Q6: How does the quad look now?

You can change the texture parameters (either clamping values or environment mode) to get different effects.

Once you've textured your quad, you're done!

Q7: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q8: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q9: How long did this lab take you?
Q10: Any other comments?

To submit this lab, zip together your source code (with image), Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L05.zip. Upload this on to Canvas under the L05 section.

LAB IS DUE BY **MONDAY OCTOBER 9 11:59 PM**!!