CSCI 441 - Lab 12
Friday, November 15, 2017
LAB IS DUE BY **MONDAY DECEMBER 3 11:59 PM**!!

Today, we'll use a framebuffer object (FBO) to perform some simple postprocessing on our scene – we'll convert the image from color RGB to grayscale.

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

Before you get started, be sure to copy the contents of the include/CSCI441 folder to your Z:/CSCI441/include/CSCI441 folder. This includes some Framebuffer Utilities to help you (slightly). If you want to pull the full code with some documentation, it's available on git (https://github.com/jpaoneMines/csci441).

Step 1 – Welcome to Night Vale

Compile and run the lab. Everything will start. You should see a skybox, a ground plane, and little street scene. We're able to render our scene.

Take a look at how everything is set up. There are many global variables at the top (you'll be using some of them as we get going). Now jump down to our main() function – there are many setup steps we go through:

1. Initialize GLFW
2. Setup OpenGL
3. Setup GLEW
4. Load our Shaders onto the GPU and get our uniform/attribute locations
5. Load all our vertex data onto the GPU via VAOs
6. Load all the textures for scene onto the GPU
7. Setup the Framebuffer

Your code will be going into Step 7 and then in `main()` as well. But first, take a look how other parts of the code is set up.

Look in `setupShaders()` to see how this application is using 3 shader programs. We need to get the uniform and attribute locations for each of them. Note the use of descriptive variable names. We need to know which variable is associated with which shader program and which uniform/attribute.

Likewise take a look at how `setupBuffers()` arranges all of the VAOs/VBOs.

Now go down to `renderScene()`. When we draw the skybox, platform, and model we are using two different shaders. Note how we toggle between them and set the uniforms each time. And if you want, open up modelLoader3.hpp to look at the `draw()` function and where the rest of some shader variables are set.

Finally, the shader files `texturingPhong.v.glsl` and `texturingPhong.f.glsl` are my Phong Shaders (which apply lighting per-fragment) and modulates with texture as well.

Alright, we've gotten the lay of the land. Let's dig in.

Step 2 – FBO Schwarz

Everything will go into the `setupFramebuffer()` method at `TODO #1`. Get out that FBO Checklist as the laundry list begins. Perform these steps in order:

1. Generate a new Framebuffer descriptor (you have a global – `fbo` – for this purpose)
2. Bind it to the `GL_FRAMEBUFFER`
3. Create an unsigned int to store the renderbuffer for the depth buffer that we'll need (even though we are rendering off screen, we still need to perform the depth test so need an associated renderbuffer for that)
4. Generate a new Renderbuffer descriptor with the variable you just created
5. Bind it to the `GL_RENDERBUFFER`
6. Allocate memory on the GPU for renderbuffer. Use `glRenderbufferStorage()` and pass four arguments:
    a. The target – this must be `GL_RENDERBUFFER`
    b. The internal format – for a depth buffer this is stored as `GL_DEPTH_COMPONENT`
    c. The buffer width – we'll set it to our framebuffer width (check your globals)
    d. The buffer height – we'll set it to our framebuffer height (check your globals)
7. Attach the depth buffer to the framebuffer. Use `glFramebufferRenderbuffer()` with the following four arguments:
    a. The framebuffer target – in Step 2 we specified `GL_FRAMEBUFFER`, so it must match
    b. Where to attach the render buffer to – since our renderbuffer corresponds to a depth buffer, we need to attach it to `GL_DEPTH_ATTACHMENT`
    c. The renderbuffer target – must be `GL_RENDERBUFFER`
    d. The descriptor for our renderbuffer – whatever you named it in Step 3
8. Generate a new texture handle that will ultimately store the color values for the framebuffer (check your globals)
9. Bind this texture to be active
10. Allocate space for the texture but send no data using `glTexImage2D()` with 9 arguments
    a. The texture target – seems to always be `GL_TEXTURE_2D` for us (after Thanksgiving it won't though!)
    b. Mipmap level – 0
    c. Internal format – our fragment shader are outputting vec4 so this will then be `GL_RGBA` to capture all four channels
    d. Width – our framebuffer width
    e. Height – our framebuffer height
    f. Border – must be 0
    g. Format – again will be `GL_RGBA`
    h. Data Type – it is internally stored as `GL_UNSIGNED_BYTE`
    i. The data – well there is none, so `NULL`
11. Set the min and mag filter along with how to wrap s & t. Linear filters and clamp to edge are appropriate choices
12. Attach the texture to the framebuffer using `glFramebufferTexture2D()` and 5 arguments:
    a. The target – must be `GL_FRAMEBUFFER`
    b. Where to attach the texture to – we want to store the color information so we'll attach to `GL_COLOR_ATTACHMENT0` since we only have one
    c. The texture target – you guessed it, `GL_TEXTURE_2D` again
    d. The texture handle – the same one we bound in Step 9
    e. The mipmap level – must be 0

And there we have it, a 12 Step Program. At this point, we need to check the status to make sure we've done everything correctly. This is where we would make the call to `glCheckFramebufferStatus()`. I've provided you with a utility function to do this and then it checks the return value to print a detailed status message if there was an error along the way. Call the function with

```
CSCI441::FramebufferUtils::printFramebufferStatusMessage(GL_FRAMEBUFFER);
```

We can also verify the contents of the framebuffer output by calling the second utility function

```
CSCI441::FramebufferUtils::printFramebufferInfo( GL_FRAMEBUFFER, fbo );
```

And this will show us the attachment point for where the output is stored.

If you are seeing

```
[FBO]: Framebuffer initialized completely!
```

Then move on!

Step 3 – Two Passes

Now we're set to do our two passes. At `TODO #2`, we'll do our first pass. Currently, it looks like it always has. Let's change that.

Pass 1
Begin by binding our FBO. Next, we want to make sure we are rendering the right size. For the viewport and the projection matrix, change the width and height to match the width and height of our framebuffer. Then after we call our `renderScene()` function to draw everything, call `glFlush()` to ensure OpenGL has finished everything. And Pass 1 is done!

Pass 2
Jump down `to TODO #3` and we need to fill in everything for our second pass. The first step is to unbind the FBO and return to rendering to our window. This is done by calling the same `glBindFramebuffer()` method but passing in the value `0` as the FBO descriptor.

Next we need to set our viewport to be the size of the window. Once that is done, we will clear both the color and depth buffer to remove any prior frames.

There is already a VAO set up for you that draws a textured quad. Jump to `LOOKHERE #1` to see where it is setup. The quad only has X and Y coordinates. Therefore, for our second pass we want to set up a 2D orthographic projection. Luckily, `glm` has a handy method `ortho()` to do this for us. Google the documentation for what you need to pass. The dimensions should range from -1 to 1.

We need to use our postprocessing shader program and pass the projection matrix as a uniform to this shader.

Finally we bind the framebuffer texture handle, bind our textured quad VAO, and draw the elements using a triangle strip of 4 vertices.

Drum roll….compile….run….we should see our original scene back!

Step 4 – Post Processing

Right now our post processing shader is just passing through the texture. Open `shaders/grayscale.v.glsl` and `shaders/grayscale.f.glsl` to see what they are doing. Nothing too fancy. But wait, there's a `TODO #A` in the fragment shader! One last step to do.

We want to convert the RGB component of our texel to grayscale. The process is three steps:

1. First compute the sum of the R, G, and B components
2. Then compute the average of the three components (divide sum by 3)
3. Finally set the final color to this average for all three components of R, G, B

Rerun the program and now it's black and white!

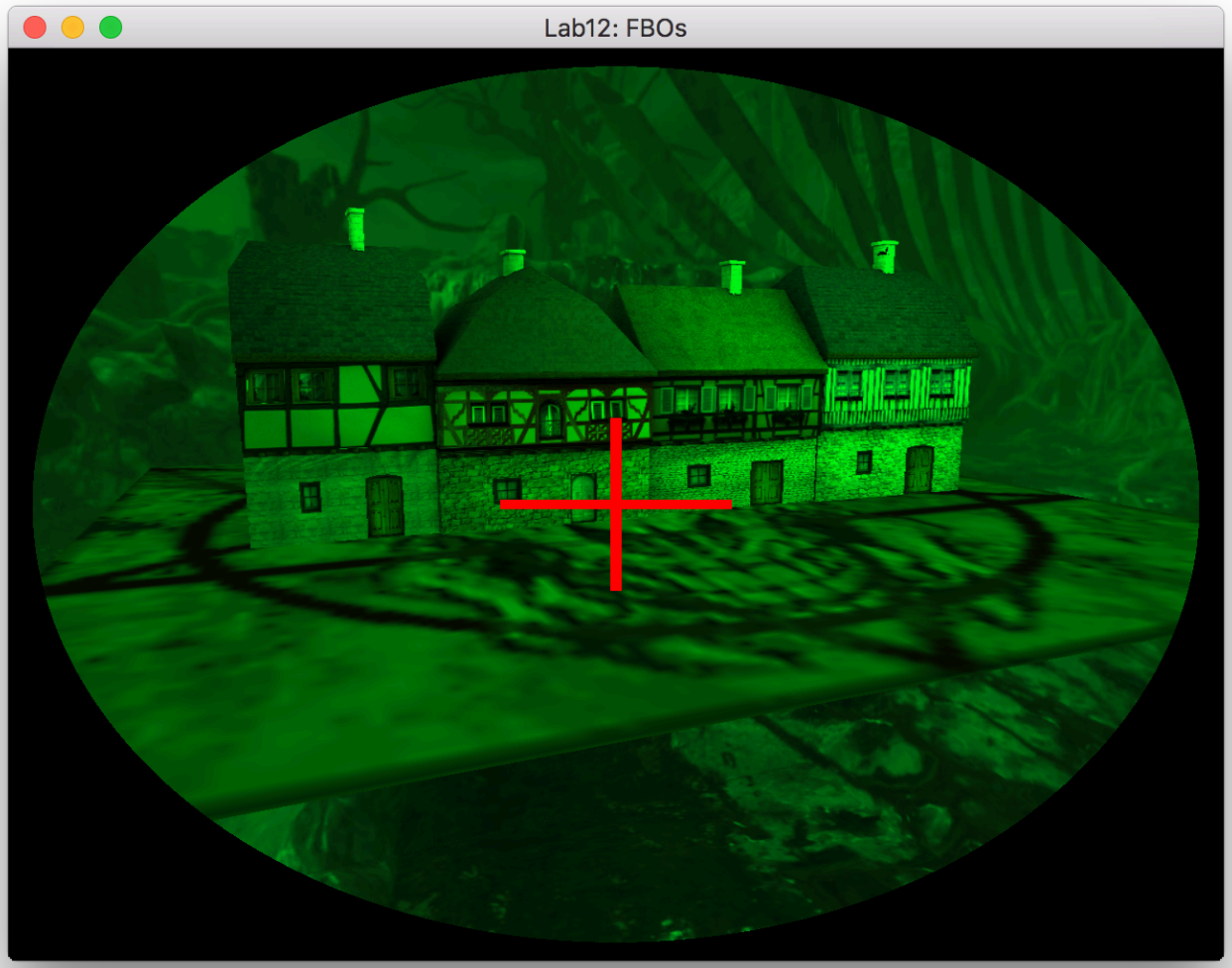
At this point you're done! Woohoo! Want to try another postprocessing technique? Read on then. (it's not required, just FYI)

Instead of computing the average of RGB, we'll do a couple of conversions. Once we get our texel, we'll scale each component by the following values (0.1, 0.95, 0.2). If we run it at this point, everything is green. Neat. Let's add the scope view.

For this, we will look at the texture coordinate. If the texture coordinate is too far from the center, then we'll set our final color to be black instead of the previously computed value. To do this test, we can compute the vector from the center of our image (0.5, 0.5) to the texture coordinate. By calling the length() function on this value, it will give us the distance from our texel to the center. If this length is greater than 0.48, we'll set our final color to black. Run and you should now have an ellipse that is rendered.

The crosshairs will function similarly. We need to do each leg separately. First, if the s coordinate of our texture coordinate is between (0.495, 0.505) and the t coordinate is between (0.405, 0.595) then we'll set the final color to be red. Second, we'll do the same comparison but switch values. If the s coordinate of our texture coordinate is between (0.405, 0.595) and the t coordinate is between (0.495, 0.505) then we'll set the final color to be red. Run and we have our scope view.

Nothing changed with what we were rendering, just how we were coloring the final fragments. Enjoy.



Q1: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q2: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q3: How long did this lab take you?
Q4: Any other comments?

To submit this lab, zip together your source code, Makefile, screenshot, and README.txt with questions. Name the zip file <HeroName>_L12.zip. Upload this on to Canvas under the L12 section.

LAB IS DUE BY **MONDAY DECEMBER 3 11:59 PM**!!