CS123 Lab 09 : Framebuffer Objects and Bloom Lighting

"The display is the computer."

-Jen-Hsun Huang

1 Introduction

To run the demo for this lab, type `cs123_lab09_bloomdemo`.

Fig. 1: Screenshot of the TA bloom lighting implementation with the XYZ RGB dragon running at 220 FPS

In this lab you will learn about framebuffer objects by implementing a simple bloom lighting in OpenGL and GLSL. Bloom lighting tries to emulate real world imaging effects by producing a feather or blur around bright areas of light in an image.[1] In pratice, framebuffer objects can be used to implement a variety of 2D image effects, as you will see in this lab. By the end of this lab you should know enough to implement Filter on the GPU (if you really felt like it).

1.1 Framebuffer Objects (FBOs)

FBOs were added to OpenGL giving it flexible off screen rendering capabilities. It is possible to think about an FBO as an offscreen render area, where you can draw a scene to. Each framebuffer object can have color buffers, as well

[1] http://en.wikipedia.org/wiki/Bloom_%28shader_effect%29

as depth an stencil buffers attached to it. However, the FBO does not need to have all the buffers attached, for example a framebuffer with just a color buffer (no depth buffer) can be used when you do not need depth testing. It is also possible to attach multiple color buffers to a single framebuffer and write to all the color buffers in the fragment shader (this is extremely useful when implementing a deferred renderer[2] and you want to output the surface normals and vertex positions in one shader pass, among other things). Furthermore, once an FBO has been rendered to, it is possible to access the color buffer, depth buffer, and stencil buffers via textures. One of the main advantages of FBOs is the ability to apply shaders when rendering to FBOs, allowing for multiple manipulations of a rendered scene. To understand how this works, lets take a look at the simple example below.

1.1.1 A Simple Usage Scenario - Blur

Let's say we wanted to blur a rendered scene on the GPU using shaders. We would first

1. Render the scene to an offscreen framebuffer object, applying whatever shaders we want (reflection, refraction, etc.) - now the scene is stored in the attached texture of that framebuffer object
2. Draw a texture mapped quadrilateral covering the entire screen, with the texture from the framebuffer drawn to above bound - effectively pasting the image on the viewport.

At the end of these two steps we would have the regular scene on the screen - so we've achieved nothing.... But what if, during the second step, we applied a fragment shader when drawing the quadrilateral covering the screen? The fragment shader would be applied once for each pixel covering the screen, or each pixel in the rendered scene. In pseudocode, this might look something like:

```
1  //Drawing code

       framebuffer_1.bind();
4      apply_perspective_camera(); //setup the camera
       reflection_shader.bind(); //bind a cool metal shader
       render_scene(); //all objects are now drawn to framebuffer
           1
7      reflection_shader.release();
       framebuffer_1.release(); //resume drawing to the regular
           buffer (screen)
       apply_orthogonal_camera(); //setup an orthographic
           projection - this makes it really easy to draw a quad
           covering the entire screen
10     framebuffer_1.texture().bind(); //bind the texture
           containing the scene
       blur_shader.bind(); //bind our blur shader
       render_textured_quad(w, h); //draw a quad covering the
           width and height of the screen
```

[2] http://en.wikipedia.org/wiki/Deferred_shading

```
13          framebuffer_1.texture().release();
            blur_shader.release();  //we have now drawn a blurred scene
                 to the screen

16  //Blur shader

            vec4 color;
19          foreach neighbor of gl_TexCoord[0].st:
                    color += texture2D(tex, neighbor.pos)
            gl_FragColor = color / numneigbors;
```

Now what if, instead of drawing to the screen in the second step, we drew to another framebuffer, say framebuffer 2, and applied a sharpen shader during that step. Then after that we could draw from framebuffer 2 back to framebuffer 1, applying yet another shader. In this way it is possible to ping-pong between framebuffers applying a different shader during each draw, before finally drawing to the screen. In this way, we can apply multiple post-process effects.

1.1.2 Using Framebuffer Objects in OpenGL

Framebuffer objects in OpenGL are managed much like the other objects (such as textures) are also managed - in effect, in order to allocate a framebuffer you just need to make a series of OpenGL calls with the correct paramaters (such as `glGenFramebuffers`). However, for this lab we will use Qt's wrapper around framebuffer objects since it's easier to use, and we'd rather not have you waste time looking up the ten lines needed to bind framebuffers and then wondering why it doesn't work.

1.2 Bloom Lighting

The basic idea of the bloom algorithm is simple. First locate areas of high intensity (brightness). Keep these areas and subtract the rest of the render (turn other pixels to black). Blur this intensity image multiple times at different kernel widths. Combine the different blur renders and a render of the original scene to create the bloom. In pictures this looks something like this:

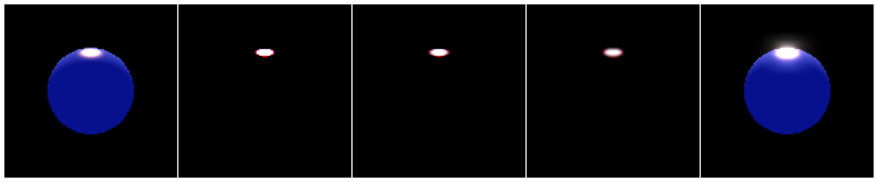

Fig. 2: The bloom lighting process

If this explanation isn't clear, don't worry, it will become clear once you start to implement it.

2 Implementation

For this lab, we have provided you with a simple drawing engine to try and simplify the coding portion. When you have time, you should take a detailed look through the code - you might want to use a similar framework for your final project. All the code you need to modify is in drawengine.cpp in the methods marked with a todo.

To begin working on this lab, first copy the code out of `/course/cs123/support_code/lab09/`. Make sure to copy all the files (including those in subfolders - use `cp -r`). Open up `lab09.pro` using `qtcreator`.

2.1 Step 0 - Getting the Scene to Show up

Take a look in `GLWidget::paintGL()`. We have rendered the entire scene into an offscreen multisample framebuffer, framebuffer zero (`fbo_0`). We then copy the contents of framebuffer zero into framebuffer one (`fbo_1`). Because framebuffer zero is a multisample framebuffer (because scene antialiasing is nice), we cannnot modify it's contents directly (this would result in an error). Keep this in mind when working on the rest of the lab...you should never use framebuffer zero.

If you were to run the program now you would get a black screen (one of the most annoying graphics bugs you can get). Why? The scene is being rendered to a framebuffer off screen, so theres nothing ever being rendered to the screen. The first thing you will want to do is to take the texture that the scene has been rendered to a render it to the screen. To do this, you will want to render a textured quadrilateral across the whole screen with the texture that has been stored in framebuffer one. Basically, you will want to do something like this:

```
        applyOrthogonalCamera(width, height); //switch to the
            orthogonal camera so we can easily draw a quad covering
             the entire screen
2       glBindTexture(GL_TEXTURE_2D, framebuffer_objects_["fbo_1"
            ]->texture()); //bind the texture in framebuffer one
        renderTexturedQuad(width, height, true); //render a
            quadrilateral across the screen
        glBindTexture(GL_TEXTURE_2D, 0); //unbind the bound texture
```

Run the program. The scene should appear.

2.2 Step 1 - Segmenting Bright Areas

2.2.1 Background

The first shader you will write will be simple. Notice first that the framebuffers we create in `createFramebufferObjects()` are stored in a hashmap which you can then access by the name of the framebuffer, giving a `QGLFramebufferObject`. The framebuffers are first allocated when the `GLWidget` is instantiated. These framebuffers are then reallocated if the window resizes, since we also want to resize the framebuffers, otherwise things will look weird. The framebuffers are

deallocated in the destructor of the draw engine. Notice that only framebuffer zero (the framebuffer we draw the full scene to) has a depth buffer attached, sicne we do not need a depth buffer when drawing a single quad to the screen, there is no reason to waste memory and computation on depth.

Also note that the framebuffers are allocated with a floating point texture type (`GL_RGB16F_ARB`), where as in previous labs you probably used `GL_RGBA`. This means that we can store floating point values (instead of integer) in the texture - with the main advantage being that when we calculate lighting in the reflection and refraction shaders, we can store values greater than (1.0 - white), which makes it easy to determine which parts of the image are bright. This ability makes HDR and bloom effects much easier, since it gives us a larger dynamic range (we can differentiate between more intensity values).

2.2.2 What You Need to Do

Before edting the fragment shader, you should add the new render pass. For this step, we want to draw the new contents into framebuffer two by binding that framebuffer, and rendering a quadrilateral with the texture stored in framebuffer one (our scene texture), while applying the shader. In other words, after drawing the scene to the screen (step 0), you will want to

1. Bind framebuffer two (try using `QGLFramebufferObject`'s `bind()` method)
2. Bind the "brightpass" shader (note that the shaders are also stored in a hash map called shader_programs_, so you can access the brightpass shader with `m_shaderPrograms["brightpass"]` , which is type `QGLShaderProgram`. Hint: try using `QGLShaderProgram`'s `bind()` function to bind your shader)
3. Bind framebuffer one's texture
4. Draw a quadrilateral
5. Unbind the shader (Hint: try using `QGLShaderProgram`'s `release()` method to unbind the shader program)
6. Unbind the texture
7. Unbind the framebuffer (try using `QGLFramebufferObject`'s `release()` method)

If you were to run your program again, nothing would have changed because we're drawing into the second framebuffer in this step and not to the screen. Unfortunately this makes it hard to debug since you can't see what the heck is going on. Temporarily comment out steps one and seven in your code. If all the steps above are correct, your screen should now be grayscale - this is because the current brightpass shader (in brightpass.frag) is really a grayscale conversion shader which you need to change...

Open up brightpass.frag in you fravorite editor.

Before writing your shader, first notice how this shader is written - instead of multiplying each color channel separately, we use a dot product to calculate the luminance in one operation. Unlike CPUs GPUs are optimized for vector operations, so that operations such as adding and subtracting vector 4s and 3s is done in one operation - instead of four different adds[3]. The same applies for the dot product. Keep this in mind for future shader code.

You will want to modify brightpass.frag to output the sample color instead of the grayscale luminance value, if the luminance value is greater than one. In other words, this shader should turn all pixels with luminance < 1.0 to black.)Recall that this is a floating point texture, so we can have luminance values > 1.0)

If you run your program, it should look something like the picture below.

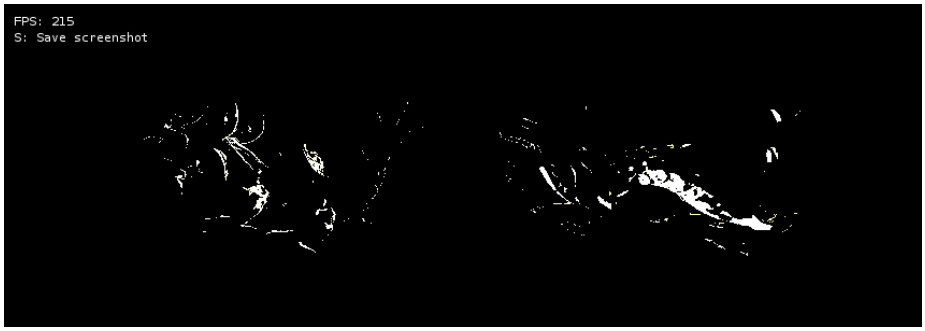

Fig. 3: Hello world!

Now that you know your shader works, make sure to uncomment steps one and seven before moving on to the next step. If you run your program it should be back to normal (display the original scene).

2.3 Step 2 - Blurring

2.3.1 Background

The next step is to blur the results of the bright pass shader (above). We will want to blur the bright pass at several kernel widths, so that the light "blooms" outwards. However, this becomes more and more expensive to do as the kernel size increases. Instead of increasing the blur radius, we could just downsize the image, apply the smaller blur, and then scale the image back up. Of course we lose some quality, but for our purposes, the quality difference is negligible.

We will implement a slight variation of the scale down blur scale up method. Recall that scaling is just interpolation of colors from the source image (in Filter you placed a triangle filter over a certain location in the source image). Instead

[3] The same functionality can be obtained on CPUs using the streaming SIMD extensions (SSE) SIMD instruction set, but that's out of the scope of this lab - see the optimization helpsession for more info.

of actually changing the image size, we will just change our sample positions in the source image. So instead of making larger blur kernels, we will just sample at farther points in the source image. A nice feature of the GPU is that if we ask for a texture sample at a non integer location, it will automatically interpolate the color from the neighboring pixels. Another nice feature is that if you sample at a point not in the image (ex: $(-.1, 0)$), it will automatically be wrapped around or clamped for you (depending on the OpenGL state). These fatures makes this method of blurring easy to implement.

An important thing to remember is that texture coordinates in OpenGL and GLSL range from zero to one. So if we are at pixel (x, y), in order to get the pixel to the right, we need to access pixel $(x + 1.0/tex\_width, y + 0)$ so the offset value here is $(1.0/tex\_width, 0)$. Similarly, to acces the pixel below (x, y), we would need to access pixel $(x+0, y+1.0/tex\_height)$. Take a look at `GLWidget::createBlurKernel(int radius, int w, int h, GLfloat *kernel, GLfloat *offsets)`. You don't need to change anything in this method, but notice how it fills out both the kernel values for a Gaussian blur and the offsets from the sample pixel for each kernel value.

2.3.2 What You Need to Do

First uncomment the part in draw_frame that is marked to be uncommented for this step of the lab. This code simply loops through all the different blur radii and adds the result of each blur pass to the screen (and also translates the result by the correct amount). If you were to run your program now, it would look really weird, since you haven't written the blur code yet.

Go to the render_blur method. This method should run a blur on the texture stored in framebuffer two and store the results onto framebuffer one. We have already filled out some of the code for you. You will want to add the framebuffer drawing code to this method. It should be very similar to the framebuffer drawing code for the bright pass segmentation above, except the source and destination framebuffers are now reversed. Also, instead of binding your brightpass shader, you will want to bind the blur shader.

Open up `blur.frag`. Note that we specify a maximum kernel size here, since we cannot dynaimcally allocate an array based on a uniform value on the GPU. Also note that it has a few uniform values. Before writing the blur shader, go back to `GLWidget::renderBlur(float width, float height)` and pass these uniform values into the shader.

You will want to pass

1. The offset values (the x, y offsets for sample locations) (try using `QGLShaderProgram`'s `setUniformValueArray(...)` method) Note that since we want to pass x, y tuples from our offset array (which looks like $[x_0, y_0, x_1, y_1 \ldots x_n, y_n]$) so that we get a vec2 in the shader program, `int tupleSize` should be equal to two.

2. The kernel values. Note that since this is just an array of floats, `int tupleSize` should equal one.

Go back to blur.frag now. It is easy to apply our blur since we have the offsets and the kernel values. In this shader program, you will want to convolve the texture with the kernel values (like in Filter). Once you are done writing your blur shader, run your program. If all goes well, it should look like the picture at the top of this handout.

3 If There's Time Left Over

If there happens to be time left over, try implementing some filters in shader code, ie. try converting some of your filters you implemented for the Filter project on the GPU - you should find that everything runs much faster.

4 HDR

HDR is a simple addition to rendering where after outputting the pixel values in the first stage, we then remap (tone mapping) them down to $[0 \ldots 1]$. (In particular, the larger range of light values calculated in the lighting stage is compressed into the visible range of light values). The mapping function used is dependent on the effect you want to achieve. See http://en.wikipedia.org/wiki/Tone_mapping for more information.