Lab 7 - Framebuffer Objects

Support code: /course/cs123/src/labs/lab07
Demo: /course/cs123/bin/cs123_lab07_demo

Intro

Up until this point, everything we have rendered with OpenGL has been displayed directly on the screen. This was fine for our purposes so far, but many graphics techniques require a scene to be rendered to a texture, which can then be used when rendering the final image on the screen.

In this lab we will learn how to render to a texture. This will open up many possibilities for doing cool things with OpenGL in the future.

Getting Started

Copy the support code into a personal directory and open up glwidget.cpp. In initializeGL(), we have already created some shader programs (with shaders you will be writing), and initialized a sphere. The Phong shaders are already written for you.

Let's start by simply rendering the sphere to the screen.

**Task 1:**
*In paintGL(), render m_sphere. Things to note:*

- Clear the color and depth buffers using glClear before drawing.
- Use the shader program m_phongProgram.
- Use m_view and m_projection to set the “view” and “projection” uniforms. Set the “model” uniform to the identity matrix.

You should see a white sphere on the screen. The camera can orbit and zoom just like in the terrain lab. But Oh No!, the sphere is tiny and in the corner! Why? Because of something called the viewport. OpenGL has a viewport, and will scale your scene to it. You may remember this from lecture as the windowing transformation. To fix it, we need to resize the viewport whenever the window changes size.

**Task 1.5:**
*In resizeGL(), resize the viewport with glViewport. Use m_width and m_height. Note:*

- glViewport takes 4 arguments, x, y, w, h. x and y specify the bottom left corner of the viewport. We want this to be at the bottom left corner of the window, so set these to 0,0
Framebuffer Objects

Our next goal is to render this sphere to a texture. To do this we will use something called a framebuffer object. We’ve seen the word “framebuffer” before, at the end of the OpenGL pipeline:

A framebuffer is a collection of textures\(^1\) that you can render into. These can include one or more color textures, a depth texture, and a stencil texture (don’t worry about that one). So far we’ve been rendering to the default framebuffer, whose color is automatically displayed on the screen after each call to paintGL().

In this lab we will create our own framebuffer objects, or FBOs. We will implement a class called FramebufferObject, which will be a wrapper for some OpenGL functions (a la OpenGLShape from Lab 1).

Take a look at framebufferobject.cpp. In create(), we will first create an FBO and a color texture with the given dimensions. The texture code should be mostly familiar from Lab 3.

**Task 2:**
In the FrameBufferObject constructor,

1. Create a new FBO using glGenFramebuffers and m_ID.
2. Create a new texture using glGenTextures and m_colorTextureID.
3. Bind the texture.
4. Initialize the texture using glTexImage2D.
   a. Like in Lab 3, use 0 for “level,” GL_RGBA for “format” and “internalFormat,” and GL_UNSIGNED_BYTE for “type.”
   b. For “width” and “height,” use the method’s parameters.
   c. Set “data” to 0, which leaves the pixel data uninitialized for now.
5. Set the texture’s min and mag filters to GL_LINEAR using glTexParameteri.

\(^1\) Technically it’s a collection of textures and renderbuffers. More on this later.
Next we have to attach the texture to the FBO. FBOs have a number of “attachment points” that allow you to render into different textures.

Whenever a fragment shader outputs a single color, it will by default be written to the first color attachment of the currently bound FBO. The first color attachment point is called GL_COLOR_ATTACHMENT0; that's the one we want to attach our texture to.

Task 3:
To attach the texture to the FBO in the FrameBufferObject constructor,
1. Bind the FBO using glBindFramebuffer. Use GL_DRAW_FRAMEBUFFER as the target, which indicates that we will eventually be using this framebuffer for drawing.
2. Make sure the texture is bound to GL_TEXTURE_2D.
3. Attach the texture to the FBO using glFramebufferTexture2D. Use 0 as the “level,” and figure out the other parameters using the documentation.
4. Clean up by unbinding the FBO and texture.

The color texture is now attached to the FBO! Next we'll fill in the bind() method. This should set up OpenGL’s state so that shapes are rendered into the texture we created above.

This method will bind the FBO and also make sure OpenGL’s viewport has the same size as our texture. The viewport is the rectangular area of the screen (or pixel buffer) that OpenGL uses for rendering.
The viewport boundaries are set with glViewport. To render on an entire texture, \( x \) and \( y \) should be 0, and \( w \) and \( h \) should match the texture dimensions.

**Task 4:**
In FramebufferObject::bind(), bind the FBO to the draw framebuffer (like above), and set the viewport using glViewport. (The texture dimensions are member variables.)

Time to test out our FBO class! Let’s try rendering our sphere into a texture.

**Task 5:**
\[ \text{a. In GLWidget::resizeGL(...), initialize m_FBO1 using its .reset(...) method with new FrameBufferObject(...). Use the dimensions passed into resizeGL as the parameters to create the FrameBufferObject.} \]
\[ \text{b. Bind m_FBO1 at the beginning of GLWidget::paintGL(). This should make all subsequent rendering go into the FBO.} \]

When you run your program the screen should now be black, because the sphere is rendered into a texture instead of the default framebuffer. To make sure texture looks correct, we’ll render a fullscreen quad and texture map it (like the ostrich in Lab 3)!

---

2 resizeGL(...) is called by QTs window manager between initializeGL() and the first call to paintGL(), as well as every time the window is resized. On certain scaled displays, waiting for resizeGL(...) (rather than using QGLWidget::width() and height()) is the most accurate way to find the correct dimensions for glViewport. We instantiate Frame Buffers here to ensure that they are created with the correct size both at program initialization and every time the window is resized.
Rendering a Fullscreen Quad

First we need to set up a quad that stretches across the whole screen. When rendering this quad we'll use the vertex shader quad.vert. Take a look at that vertex shader, which has vertex attributes for position and a UV-coordinates. Notice that the position attribute is not transformed at all, so it should already be in clip coordinates.

**Task 6:**
*Initialize m_quad (a smart pointer to an OpenGLShape) to be two triangles that stretch across the whole screen. Remember: in clip coordinates the screen goes from -1 to 1 in x and y.*

Now let's draw the quad using m_textureProgram, which uses the shaders quad.vert and texture.frag. Currently texture.frag always outputs red – we'll fix that in a bit, but for now that will be a useful way to make sure our quad is set up correctly.

We need to render the quad into the default framebuffer (AKA the screen). To render into the default framebuffer after previously binding an FBO, you can
- bind 0 to the draw framebuffer,
- call glViewport with the screen dimensions (use GLWidget’s width() and height() methods), and
- clear the color and depth buffers using glClear. glClear only operates on the currently bound framebuffer, so we usually need to do it once for every framebuffer we render into.

**Task 7:**
*In paintGL(), after rendering the sphere into an FBO, render the fullscreen quad to the screen using m_textureProgram. The screen should now be completely red.*

Now let’s texture map the quad. This should basically be a repeat from Lab 3, but instead of binding an ostrich texture we'll bind the FBO’s color texture.

**Task 8:**
- *In paintGL(), before drawing the quad, bind m_FBO1’s color texture using glBindTexture. You can get the handle for the FBO’s color texture by calling getColorTextureID().*
- *In texture.frag, sample the texture uniform using the given UV-coordinates. You should now see a sphere rendered back on the screen!*

Your lighting is probably messed up now, like in the image below. We'll fix that next.
Using a Renderbuffer

Why is the lighting messed up now? You’re actually seeing some of the back faces of the sphere rendered on top of the front faces. One way to fix this would be to enable GL_CULL_FACE to ignore back faces of triangles. We shouldn’t have to do that, though, because we enabled GL_DEPTH_TEST for you, which should make sure that closer polygons are always in front.

OpenGL’s depth test works by storing the depth of each pixel and only updating a pixel if a closer depth comes along. Therefore it only works when the framebuffer has a depth attachment. The default framebuffer automatically comes with a depth attachment, but we have to make our own when using a custom FBO.

One option is to attach a texture to the depth attachment, like we did with the color attachment. Another option is to attach a renderbuffer. A renderbuffer contains an image just like a texture, but it cannot be sampled after rendering. The advantage is that it is faster because it is a more optimized render target. Because of this, you should use textures when you need to sample values after rendering, and renderbuffers when you don’t.

We don’t need to sample depth values in this lab, so let’s use a renderbuffer for the depth attachment.

Task 8:
In the FrameBufferObject constructor,
1. Create a new renderbuffer using glGenRenderbuffers and m_depthBufferID.
2. Bind the renderbuffer using glBindRenderbuffer.
3. Initialize the renderbuffer using glRenderBufferStorage. Use GL_DEPTH_COMPONENT as the “internalformat,” and use the same dimensions as the color texture.
4. Attach the renderbuffer to the FBO’s depth attachment using glFramebufferRenderBuffer. Some tips:
   a. Make sure the FBO is bound first.
   b. Use GL_DEPTH_ATTACHMENT as the “attachment.”
c. Use GL_RENDERBUFFER as the “renderbuffertarget.”

Run your program. The lighting should now be correct!

**Blurring the Sphere**

Let’s now use FBOs to do something we couldn’t do otherwise: blur the sphere! You already know how to blur an image on the CPU. Now we’ll do it on the GPU in a fragment shader!

Think about why we need FBOs for this: A fragment shader runs on each pixel in isolation, but blurring requires averaging *multiple pixels* together. In order to access multiple pixels in a fragment shader, they need to already be stored in a texture.

We’re going to use a separable blur kernel, just like in Filter. Let’s start by just blurring horizontally. We’ll write a new fragment shader, horizontalBlur.frag, which will be similar to texture.frag but will read multiple texels and take a weighted average. (Recall that a texel is just a pixel of a texture image.)

To access multiple texels in horizontalBlur.frag we need a way to offset the UV-coordinates by an amount equivalent to one texel. GLSL has a built-in function textureSize that returns the width and height of a texture. In our case, `textureSize(tex, 0).xy` would return `vec2(960, 540)`. To get the width and height of a texel in UV space, simple take `1.0 / textureSize(tex, 0).xy`.

**Task 9:**

- **Implement horizontal blurring in horizontalBlur.frag** by offsetting the UV-coordinates in the x-direction and taking a weighted average. Use a linear blur kernel and a support width of 20 or so. Note that you can use for-loops in GLSL with the same syntax as in C. (There are some restrictions on the loop conditions; if you have trouble just hard-code the loop boundaries.) Also ignore edge cases for now – we’ll handle that later.
- **In paintGL(), use m_horizontalBlurProgram instead of m_textureProgram to render the full-screen quad.**

Your sphere should now be blurred horizontally!

Our pipeline currently looks like this:
Now let's blur it vertically. You'll need to use a second FBO here, because you shouldn't write to the same texture as you're reading from. Here's a diagram of the new pipeline:

![Diagram of the new pipeline](image)

**Task 10:**
- Implement vertical blurring in `verticalBlur.frag`. Again, ignore edge cases for now.
- In `initializeGL()`, initialize `m_FBO2` using the `reset(...)` method with `new FrameBufferObject(...)`. 
- In `paintGL()`, render the horizontally blurred image into `m_FBO2`. Make sure to clear the `m_FBO2`'s color and depth buffers!
- Then bind `m_FBO2`'s color texture, and use the `m_verticalBlurProgram` to render a fully blurred sphere to the screen.

That's it, you've implemented blurring on the GPU! If you've done it right, you should be able to rotate the sphere and see each frame blurred instantly. Much faster than CPU blurring, right?

**Texture Wrap Parameters**

Now let's clean up the edge cases that we left out of the shaders. With no edge case handling, we are sometimes sampling the texture with UV-coordinates outside the range \([0, 1]\). What does OpenGL do in that case?

By default, OpenGL wraps around to the other side of the texture. This isn’t what we want. To see this yourself, try translating the sphere and see what happens when it's at the edge of the image.
Instead, we can tell OpenGL to clamp UV-coordinates to the appropriate range. In other words, coordinates <0 will be clamped to 0 and coordinates >1 will be clamped to 1. To do this we will adjust the texture parameters for texture wrapping.

**Task 11:**
In paintGL(), after each time you bind one of the textures, call glTexParameteri twice: once with a “pname” of GL_TEXTURE_WRAP_S, and once with a “pname” of GL_TEXTURE_WRAP_T. In both cases, set “param” to GL_CLAMP_TO_EDGE.

You should no longer see artifacts when the sphere is translated to the edge of the screen.