This is taken from the off-screen rendering example in the current OpenGL SuperBible book.

Note that this is the “new” OpenGL 4.5 way to do things, as some of these functions are only available in 4.5

In global.h and global.cpp:

// In our case, the off-screen framebuffer has a colour and depth component.

// (Note that they could have many different components)

extern GLuint g\_FBO; // = 0;

extern GLuint g\_FBO\_colourTexture; // = 0;

extern GLuint g\_FBO\_depthTexture; // = 0;

extern GLint g\_FBO\_SizeInPixes; // = 512 the WIDTH of the framebuffer, in pixels;

// Initially the buffer is square...

GLuint g\_FBO = 0;

GLuint g\_FBO\_colourTexture = 0;

GLuint g\_FBO\_depthTexture = 0;

GLint g\_FBO\_SizeInPixes = 512; // = 512 the WIDTH of the framebuffer, in pixels;

Sometime after you init everything in OpenGL (I have mine in “main.cpp”):

//GLuint g\_FBO = 0;

//GLuint g\_FBO\_colourTexture = 0;

//GLuint g\_FBO\_depthTexture = 0;

glCreateFramebuffers( 1, &g\_FBO );

glBindFramebuffer( GL\_FRAMEBUFFER, g\_FBO );

glGenTextures( 1, &g\_FBO\_colourTexture );

glBindTexture( GL\_TEXTURE\_2D, g\_FBO\_colourTexture );

glTexStorage2D( GL\_TEXTURE\_2D, 1, GL\_RGB8, g\_FBO\_SizeInPixes, g\_FBO\_SizeInPixes);

glTexParameteri( GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR );

glTexParameteri( GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR );

glGenTextures( 1, &g\_FBO\_depthTexture );

glBindTexture( GL\_TEXTURE\_2D, g\_FBO\_depthTexture );

glTexStorage2D( GL\_TEXTURE\_2D, 1, GL\_DEPTH\_COMPONENT32F, g\_FBO\_SizeInPixes, g\_FBO\_SizeInPixes);

*Here, the width and height are the same, so it’s square, but it can be any size and aspect ratio you’d like*

glFramebufferTexture( GL\_FRAMEBUFFER,

GL\_COLOR\_ATTACHMENT0,

g\_FBO\_colourTexture, 0 );

glFramebufferTexture( GL\_FRAMEBUFFER,

GL\_DEPTH\_ATTACHMENT,

g\_FBO\_depthTexture, 0 );

static const GLenum draw\_bufers[] = { GL\_COLOR\_ATTACHMENT0 };

glDrawBuffers( 1, draw\_bufers );

// Point back to default frame buffer

glBindFramebuffer( GL\_FRAMEBUFFER, 0 );

In the render, you set the framebuffer to this new, offscreen texture:

glBindFramebuffer( GL\_FRAMEBUFFER, ::g\_FBO );

And clear the buffer. NOTE: This is ***NOT*** the clear the screen, but the new framebuffer

glViewport( 0, 0, g\_FBO\_SizeInPixes, g\_FBO\_SizeInPixes );

GLfloat zero = 0.0f;

GLfloat one = 1.0f;

glClearBufferfv( GL\_COLOR, 0, &zero );

glClearBufferfv( GL\_DEPTH, 0, &one );

Then draw the entire scene (WITHOUT the off-screen quad)

Also: ***DON’T*** present the back/frame buffer (call the glutSwapBuffers() in freeGLUT or glfwSwapBuffers()GLFW) at this point as there’s nothing on the actual screen (you’ve been drawing to the off-screen framebuffer, not the “regular” framebuffer). If you call this after rendering to the off-screen, then again call it when you draw the quad (see below), then the screen will flicker (because it’s “presenting” 2x per frame, with one frame that has nothing on it).

Then point the framebuffer back to the default:

glBindFramebuffer( GL\_FRAMEBUFFER, 0 );

You then connect the frame buffer texture to the shader (either to the current shader or another shader). This is the “g\_FBO\_colourTexture” texture listed above.

// 1. Choose the texture unit

glActiveTexture( GL\_TEXTURE0 );

// 2. Set this texture to the texture unit (in the framebuffer)

glBindTexture( GL\_TEXTURE\_2D, g\_FBO\_colourTexture );

// 3. Assign the sampler (in the shader) to the same texture unit

::g\_pTheShaderManager->SetUniformVar1i(::g\_pTheShaderManager->GetCurrentShaderID(),

"texOffScreen\_00",

0 ); // "GL\_TEXTURE0"  
// Or this:  
glSetUniform1i( textOffset\_ID, 0 ); // Where “0” is for GL\_TEXTURE0 (if that’s where you want it)

Then draw the quad. You can:

\* Use exactly the same draw call, being sure to ONLY draw the quad, and have it positioned close to the camera

\* Switch to orthographic projection and, again, only draw the quad

The shader you are using – initially – is trivial:

**void main(void)  
{**

**out\_Colour = texture( texOffScreen\_00, ex\_UV\_x2.xy );**

**return;**

**}**

Note that this simple shader will be replaced by some very elaborate thing.

Also, we will be altering this to write to a more complicated back buffer – with multiple textures attached – to render colour, normals, etc.

**Using gl\_FragCoord to copy to a fullscreen quad:**

You can try to position your quad in “the right spot” to render, but this can be tricky. Even in orthographic projection, it can be annoying to set the quad, the camera, etc. Also, this will change as the screen size changes.

An alternative is to use the built in “fragment coordinates” variable, gl\_FragCoord, to generate the texture cords.

gl\_FragCoord returns the xyz on screen (i.e. pixel) location, as well as the depth. Note that that x & y are integers, not floats, indicating how far – in pixels – you are from the “corner” of the screen (lower left). This means that the maximum value will change when you change the size of the window.

To convert to texture coordinates, you need to take into account the screen width and depth:

vec2 textCoords = vec2( gl\_FragCoord.x / screenWidth, gl\_FragCoord.y / screenHeight );  
out\_Colour = texture( offscreenTexture, textCoords);

One other “catch”: remember that the fragment shader won’t run unless there are pixels to be written to the screen, so you have to draw *something.* The flip side of that is that you can be pretty “rough” in your positioning, so long as whatever you’re drawing – even a single triangle (AMD and nVidia’s recommendation) – to the screen. In fact, you can just draw a big triangle/quad with a camera really close to it, with the vast majority of the object off screen, and it will generate a perfect image at any resolution.

Keep in mind that this is just for doing full-screen deferred rendering – if you just want to render the texture back onto something, you still have to deal with camera, texturing, and sizing issues.