# Lab 5 – Render To Texture

### Introduction

In this tutorial we are going to create a dynamic texture, by rendering to it. This can be used to create a wide range of effects, for example reflections and dynamic environment maps. It is also the basis for shadow mapping, which we will encounter in the next tutorial. We will now use the technique in to create a surveillance camera thingy, which is another popular use for render to texture.

### Frame Buffer Objects

We have in previous tutorials created textures, and loaded them with file data using DevIL. However, to fill them with rendered data we must first attach them to something called a *Frame Buffer Object,* or FBO for short (See OpenGL spec §4.4). We have already used one, in OpenGL there is a default FBO that presents the rendered result to the screen (or in a window).

An FBO can have multiple *textures* and *render buffers,* attached. This can be used to produce several textures simultaneously. For this tutorial we will need only one color texture target, and a depth buffer.

Let us create an FBO; to do this, add the following at the end of `initGL`:

```
glGenFramebuffers(1, &frameBuffer);
// Bind the framebuffer such that following commands will affect it.
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
```

The first function allocates a new FBO object, and the second binds it for use with subsequent commands (notice that this is analogous to how textures and VBOs are managed).

Then we will create a texture into which we will be rendering, directly after add:

```
// Create a texture for the frame buffer, with specified filtering,
// rgba-format and size
glGenTextures(1, &texFrameBuffer);
glBindTexture(GL_TEXTURE_2D, texFrameBuffer);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, NULL);
```

This creates a 512 by 512 texel (pixel) texture, with rgba channels, and sets some parameters. Note that we pass `NULL` to `glTexImage2D` as data pointer. This tells OpenGL to allocate the space, but to not initialize it. This is fine, as we will render to it. Sometimes, though, it is useful to put a placeholder image in place.

Next we will attach the texture to the FBO (remember that the FBO is still bound):

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D, texFrameBuffer, 0);
```

We also need to create a depth buffer for the FBO, and associate it with the frame buffer. Since we will not use the depth as a texture, we create a render buffer (See OpenGL spec §4.4.2) instead of a texture.

```
glGenRenderbuffers(1, &depthBuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 512, 512);
// Associate our created depth buffer with the FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
            GL_RENDERBUFFER, depthBuffer);
```

To see that it all went according to plan we perform:

```
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE)
{
      fatal_error( "Framebuffer not complete" );
}
```

There are a number of ways a frame buffer can be incomplete, it can for example lack color attachments etc. See OpenGL spec §4.4.4 for many more details.

Finally we will bind the default frame buffer (0 or zero) so that other commands do not affect the our FBO.

```
// Restore current binding (rendering) to the default frame buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

***Optional:* Controlling Output Buffers**

This is somewhat advanced information, not strictly necessary as the default for a newly created FBO will work fine. This is, however, needed in situations where there is more than one color attachment, and to form a complete and correct view of how the associations work. While we have the FBO bound we specify the *draw buffer* to use using the command:

```
glDrawBuffer(GL_COLOR_ATTACHMENT0);
```

This sets the 0th draw buffer to the render target to be directed to the texture or render buffer at `GL_COLOR_ATTACHMENT0`. This corresponds to what we set up in `glBindFragDataLocation`.

If there are multiple color attachments we would use:

```
GLenum tgts[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
glDrawBuffers(2, tgts);
```

Then if we also added another `glBindFragDataLocation(…, 1, …)`, then this would correspond to `GL_COLOR_ATTACHMENT1` (See OpenGL spec §4.4).

Perhaps confusingly, we can reverse the draw buffer order, as below (notice the 0 and 1 changing places):

```
GLenum tgts[2] = { GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT0 };
glDrawBuffers(2, tgts);
```

In which case the out variable `"fragmentColor"` ends up in `GL_COLOR_ATTACHMENT1` and vice versa. This is just one example of how in OpenGL there is often many ways to do the same thing. This flexibility allows efficient solutions to be created, but can sometimes lead to non-obvious code.

### Rendering to the FBO

To make things interesting, the program loads some models. There is a spaceship of some kind, a surveillance camera and three security consoles. Our job is to ensure what is seen through the, high-tech, brown, security camera is shown on the screens in the consoles.


**Assignment:** Before you go on run the program to see how the scene looks. It should look like this: Notice the consoles are showing the test image.

Inspect the function `display()` notice how the code sets up the viewport, shader and view and projection matrices from the (input controlled) camera. It then calls `drawScene`, to draw the scene geometry, independent of view, which we will find useful very soon indeed.

To render the scene from the **security camera** we must make use of the FBO we created earlier. We bind the FBO, set the viewport and clear the attached targets. Add this at the beginning of `display`:(after the first call to `glUseProgram`):

```
// bind the frameBuffer as our render target, this means that render
// operations will end up affecting the texture 'texFrameBuffer'
// that we attached to the frame buffer earlier.
glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);

// We must also set the viewport for the frame buffer to match the
// size of the texture, if it is not the same portions may not be
// drawn or things may end up outside of the texture (as in not be
// visible).
glViewport(0, 0, 512, 512);

// Clear the color/depth buffers of the current FBO
// (i.e. the attached textures and render buffers)
glClearColor(0.6,0.0,0.0,1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

We must perform the rendering to texture before the rendering of the scene to the default render target, as we will use the result as a texture. If we were to do it after, we'd be a frame behind.

Any rendering commands following these lines will now end up drawing into the texture. Next we call `drawScene` again:

```
drawScene(shaderProgram,
  lookAt(securityCamPos, securityCamTarget, up),
  perspectiveMatrix(45.0f, 1.0f, 1.5f, 100.0f));
```

This we pass in a view matrix looking from the `securityCamPos` towards the `securityCamTarget`. We also create a perspective projection. Notice that the near plane is 1.5 units out, this is important as if it is too close we will just see the inside of the security camera! Also note that the aspect ratio is 1.0, this is not strictly correct, since it should match that of the security console screens.

We're now done rendering to our FBO! All we need to do to conclude is to bind the default frame buffer. Insert the below after the call to `drawScene`:

```
// Bind the default frame buffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

If you run the program now, there should be no noticeable difference! Hidden away we should have a dynamically updated texture.

Now we need to render the texture on the security console screens. To achieve this we have duplicated the geometry for the screens in a function called `drawSecurityScreenQuad`. Invoke this function **just before** the call to render the security console model (in the function `drawSecurityConsole`). We need to do it before rendering the model, as the depth test otherwise will discard the geometry.

```
…
drawSecurityScreenQuad();
securityScreenModel->render();
…
```

Next, insert the following, just before the call to `drawSecurityScreenQuad`:

```
setUniformSlow(shaderProgram, "has_diffuse_texture", 1);
glBindTexture(GL_TEXTURE_2D, texFrameBuffer);
```

These uniforms we set are those required by the shader (`simple.frag`) to enable texturing. This shader is designed to work with the `OBJModel` class, which sets certain material parameters.


The expected result is shown here to the right. Notice that one of the security consoles is shown inside the security console. This occurs because we use the texture in the scene, while it is being rendered to. This is, however, not guaranteed to work. To be certain, we would need to make a copy of the texture before rendering starts.

We now have rendered to a texture and used it on another object. We are well on our way to a number of great special effects...

**Assignment:** We *can* draw the `drawSecurityScreenQuad` after the `securityScreenModel` if we instead use `glDepthFunc(GL_LEQUAL)` (the default is `GL_LESS`). Try this; insert a call to `glDepthFunc(GL_LEQUAL)`, somewhere towards the end of `initGL` (but do not change the order of drawing just yet). What happens?

Explain why: _____

_____

_____

Now either remove the call to `glDepthFunc` again, or draw the quad after the model.

**Assignment**: Discuss in the group (if you are working alone try to find some else who is in the same situation) uses for rendering to textures, what kind of effects can you think of?

_____

_____

***When done, show your result to one of the assistants. Have the finished program running and be prepared to answer some questions about what you have done.***