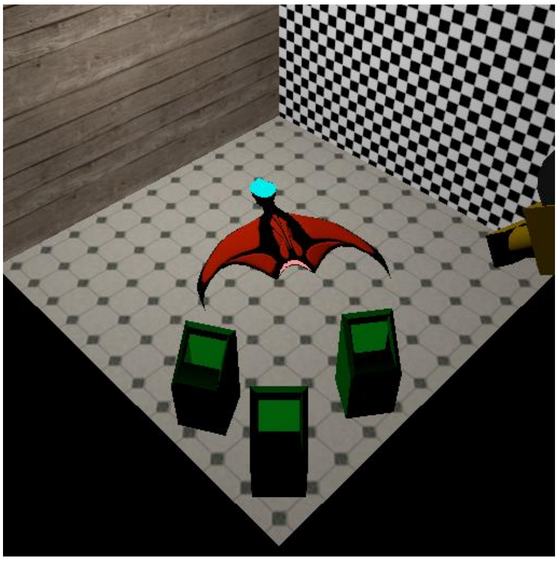
Lab5–Render To Texture

Introduction

In this tutorial we are going to create dynamic textures, by rendering to them. 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.

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 that 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 the image on the right. Note that the consoles do not have any screens yet. We will fix this pretty shortly.

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.

Frame Buffer Objects

We have in previous tutorials created textures, and loaded them with data from files using DevIL. However, to fill them with rendered data we must first attach them to something called a *Frame BufferObject,* 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.

First we need to create the color texture, into which we will be rendering. At the end of `initGL()`, 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 (or 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.

However, sometimes it is useful to use a placeholder image. We will do this to make sure that rendering the security screens is working properly before we start rendering to the texture. Comment out the call to `glGenTextures` and to `glTexImage2D,` and add code to load a texture instead:

```
//glGenTextures(1, &texFrameBuffer);
texFrameBuffer = ilutGLLoadImage("tvTestCard.jpg");
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 lets DevIL generate the texture and load the image into it. Conveniently the resolution of `tvTestCard.jpg` is also 512 by 512.

We also have to declare a variable, which stores the reference to the texture, called "`texFrameBuffer`".Declare somewhere in global scope an integer `GLuint texFrameBuffer;`.

Now we need to show the texture (that is the test image we've just loaded) on the security console screens. To achieve this, we have copied the vertex geometry for the screens into a function called `drawSecurityScreenQuad()`. Invoke this function just before the call to render the security console model (in the function `drawSecurityConsole()`).

```
...
drawSecurityScreenQuad();
securityConsoleModel->render();
...
```

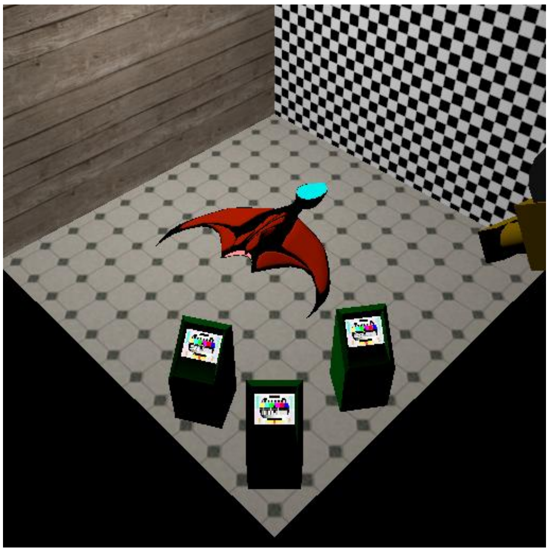
Next, to bind the texture, insert the following just before the call to `drawSecurityScreenQuad()`:

```
glBindTexture(GL_TEXTURE_2D, texFrameBuffer);
```

The 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 to the right. Run your program and compare – you should now have three security consoles showing a test image.

It's about time we did some rendering to this texture.



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 call allocates a new FBO, and the second binds it for use with subsequent commands (notice that this is analogous to how textures and VBOs are managed). Remember to declare the variable `frameBuffer` in global scope.

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. Declare the variable `depthBuffer` and add the following.

```
        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) such that subsequent 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. It will also give you a more complete and correct view of how the associations work. While we have the FBO bound we specify the *drawbuffer* 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 were multiple color attachments, we would use:

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

We would also added another `glBindFragDataLocation(…, 1, …)`; 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 are 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 render the scene from the point of view of the **security camera** we must make use of the FBO we created earlier. To 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 lag a frame behind.

Any rendering commands following these lines will now end up drawing into the texture. Next we call `drawScene()` (note that this is in addition to the call already present further down, and which renders to the default frame buffer):

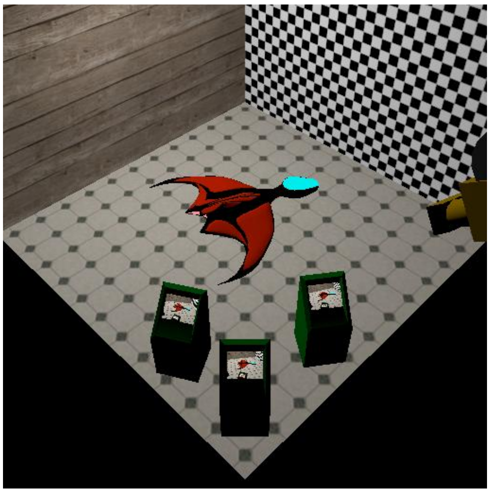
```
	drawScene(shaderProgram,
	  lookAt(securityCamPos, securityCamTarget, up),
	  perspectiveMatrix(45.0f, 1.0f, 1.5f, 100.0f));
```

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, in order to make the ordinary scene drawing end up on screen. Insert the below after the call to `drawScene`:

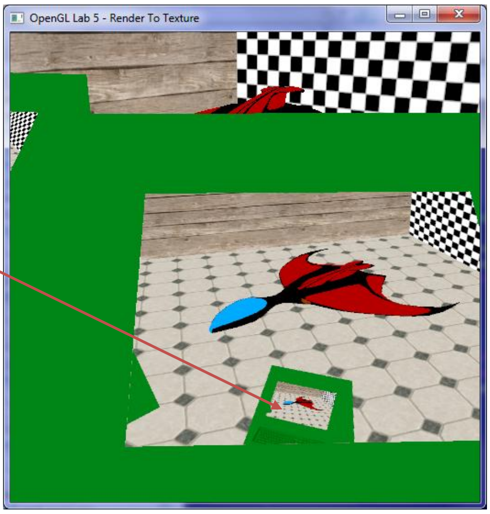
```
    // Bind the default frame buffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

If you run the program now, the result should be like shown here to the right.



Undefined Behavior

If you zoom in on one of the console screens you may see the consoles in the texture, complete with the screen (i.e. recursive behavior). You also may not! See the image to the right for an example, here we see the space ship, but one console is missing.



This uncertainty arises from the fact that we are actually breaking the OpenGL law, or standard: We use the texture in the scene, while it is being rendered to. That is, when we call `drawScene` to render to the texture, it binds the very same texture in `drawSecurityConsole`. That this is not healthy should be obvious, if you think about it for a while.

Now, we might expect OpenGL to report an error message in this kind of situation. However, *it does not.* OpenGL is performance oriented, and mandating excessive error checks may make API overhead higher, therefore it is common for this kind of misuse to result in **undefined** behavior, which is the OpenGL way of giving the implementer (i.e. AMD or NVIDIA) free reins.

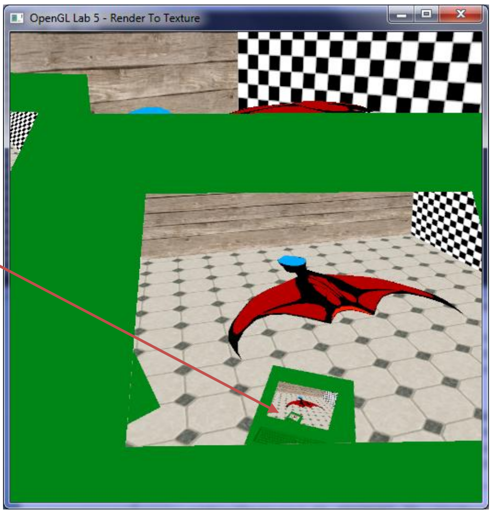
A straightforward way to correct this is to copy the texture before using it, then we never render to it while in use. For example: render to texture A, copy A to texture B, bind texture B and render the scene. Another way is to alternate between textures being rendered to and used, which should be more efficient, but also a little more complicated to set up.

Let’s try the first solution, that is, copy the texture after drawing to it. First we need to create another texture, call it `texFrameBuffer2`. Next, initialize it in the exact same way as `texFrameBuffer`. Now we want to copy the frame buffer contents into this texture after drawing the scene, which is fortunately simple. Just after drawing the scene into the screen frame buffer, and while it is still bound, add:

```
    // copy to second texture
    glBindTexture( GL_TEXTURE_2D, texFrameBuffer2 );
    glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, 512, 512);
    glBindTexture( GL_TEXTURE_2D, 0);
```

Then we must change which texture is used when drawing the security screen to be `texFrameBuffer2`.

The result should now look like the image to the right. Notice how the console is shown recursively on screen. If you squint you can even make out the red pixel or so that is the space ship in the third recursion. In principle, it goes on forever.



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

____

____

____

Post Processing

Post processing is perhaps the most common use for render to texture today, with the very likely exception of shadow maps. Most, if not all, games make use of a post-processing pass to change aspects of the look of the game, creating effects such as motion blur, depth of field, bloom, simple color changes, magic mushrooms, and more.

Conceptually post processing is simple: instead of rendering the scene to the screen, it is rendered an off-screen render target of the same size. Next, this render target is used as a texture when rendering a full screen quad and a fragment shader can be used to change the appearance. Remember that a fragment shader is executed once for each fragment, and for a full screen quad this is the same as each pixel.

Add a new FBO to the program named `postProcessFrameBuffer`, with a texture and depth buffer, as before. Call the texture `texPostProcess`, and make sure to match the size of the default frame buffer (e.g. window size). The steps are the same as we have just used to make the security screen render target. However, to make it easier to access the texture using pixel coordinates in the post processing shader later, we must use the target `GL_TEXTURE_RECTANGLE_ARB` instead of `GL_TEXTURE_2D`, when setting up and binding the texture. Note that we could use an ordinary 2D texture instead and use `texelFetch`, to sample the textures. However, this requires integer coordinates, which disables bilinear filtering, which we make use of to extend the blur kernel.

Now ensure that this frame buffer is bound instead of the default when the scene is rendered. That is, replace the buffer used as target here with `postProcessFrameBuffer`. Remember to bind the default FBO again after drawing the scene to the post processing FBO.

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

Now, nothing will show up on screen, as the scene is now drawn into a texture. To verify that it is working, we will add a step to copy the contents of a frame buffer to another. After scene rendering, add the following:

```
	glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
	glBindFramebuffer(GL_READ_FRAMEBUFFER, postProcessFrameBuffer);
	glBlitFramebuffer(0, 0, w, h, 0, 0, w, h, GL_COLOR_BUFFER_BIT,
		GL_NEAREST);
```

This will copy the entire contents of the bound `GL_READ_FRAMEBUFFER` (here identified by `postProcessFrameBuffer`) into the bound `GL_DRAW_FRAMEBUFFER` (here the default 0). (The term “blit” is derived from ***Bl**ock **I**mage **T**ransfer,* a common operation in computer graphics.)

Running the program now should yield exactly the same result as earlier, before we started adding post processing. This means that the texture `texPostProcess`, now contains the rendered scene.

Now then, we will add a fragment shader and do some actual post *processing* (not just post copying). To this end, there is already a second shader pair loaded by the tutorial, "`postFx.vert/postFx.frag`". Comment out the code to blit the frame buffer again, and instead make sure to bind the default FBO

```
	glBindFramebuffer(GL_FRAMEBUFFER, 0);
	glViewport(0, 0, 512, 512);
	glClearColor(0.6,0.0,0.0,1.0);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Then use the shader `postFxShader`, and set the parameters needed:

```
	setUniformSlow(postFxShader, "frameBufferTexture", 0);
	glBindTexture(GL_TEXTURE_RECTANGLE_ARB, texPostProccess);
	setUniformSlow(postFxShader, "time", currentTime);
```

Finally we draw a full screen quad by calling the function `drawFullScreenQuad()`. This function is defined at the end of the main file. If everything has gone according to plan, you should now see something like the image to the right, where the image has been subjected to a good old sepia tone filter. (Sepia tone is commonly seen in old photographs. Wikipedia has more information about this.)

Inspect the `postFx.frag` shader. There are several functions defined that can be used to achieve different effects. Notice that they affect different properties to achieve the effect: the wobbliness is affected by changing the input coordinate, blur samples the input many times, while the two last simply change the color sample value.

Note that we use a special type of sampler to access the texture, `sampler2DRect`. This allows us to use screen coordinates to sample the texture (normally, textures are sampled with coordinates in the range [0, 1]). This allows the use of the built in OpenGL variable `gl_FragCoord` as texture coordinates, which supplies the screen space coordinates (within the render target) of the fragment being shaded.

The functions are used from the main function in the shader, try out different ones, and combine them. Notice the last line which is a commented out variation that chains all effects (except grayscale).

```
	vec2 mushrooms(vec2 inCoord);
```

Perturbs the sampling coordinates of the pixel and returns the new coordinates. These can then be used to sample the frame buffer. The effect uses a sine wave to make us feel woozy. Can you make it worse?

```
	vec3 blur(vec2 coord);
```

Samples a region of the frame buffer around the input coordinates, using Gaussian filter weights to blur the image as the kernel width is not that large, it doesn't produce a very large effect. Making it larger is both tedious and expensive, for real time purposes a separable blur is preferable, which requires several passes. We will explain this process in the (*optional*) Section **Efficient Blur and Bloom** below.

```
	vec3 grayscale(vec3 sample);
```

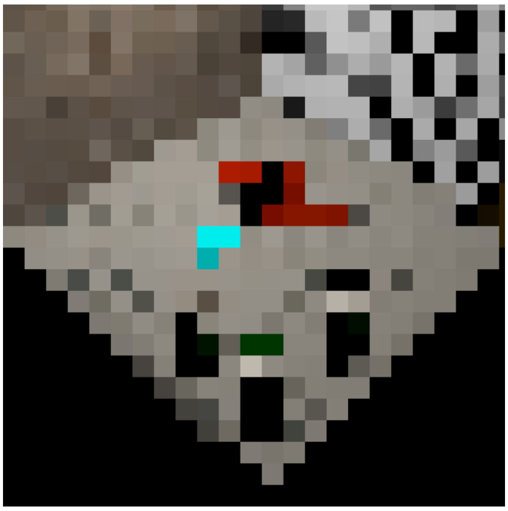
The `grayscale()` function simply returns the luminance (perceived brightness) of the input sample color.

```
	vec3 toSepiaTone(vec3 rgbSample);
```

The `toSepiaTone()` function converts the color sample to sepia tone (by transformation to the YIQ color space), to make it look somewhat like an old photo.

Experiment with the different effects, for example change the colorization in the sepia tone effect, can you make it red? Also try combining them. Try to understand how each one produces its result.

Assignment: You shall now add another effect. The effect is called Mosaic and the result is shown to the right. Each square block of pixels shows the color of the same pixel (single sample, no averaging needed), for example the top right or some such. Implement this effect by adding a new function in the fragment shader. Consider the pre-made effects: what part of the data do you need to change?



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.

Optional: Efficient Blur and Bloom

Heavy blur requires sampling a large area. To implement such large filter kernels efficiently, we can exploit the fact that the Gaussian filter kernel can be decomposed into a vertical and horizontal component, which are then executed as two consecutive passes. The process is illustrated below.

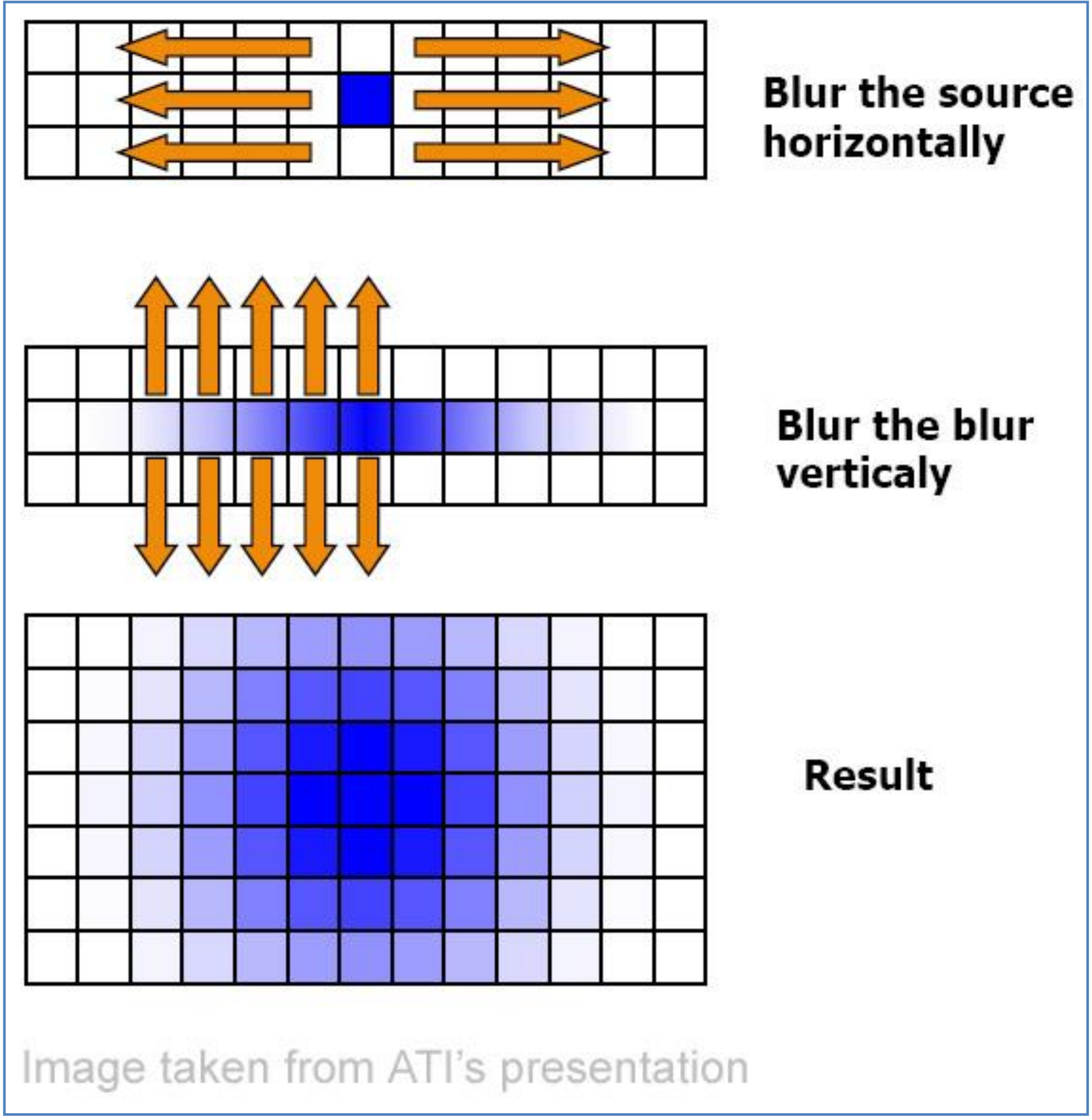


Image taken from ATI's presentation

To implement this in our tutorial, we will need to add two new FBOs: one to store the result of the first, horizontal, blur pass, and then another to receive the final blur after the vertical blur pass. Note that, in practice, we can just ping-pong between buffers, to save storage space. However, this adds confusion, and we want the blur in a separate buffer to create bloom.

The barrage of global variables to keep track of all these FBOs and textures is becoming a bit much, so now we will introduce a helper function, and structure to, to help create and use post-processing FBOs.

```
struct FboInfo
{
  GLuint id;
  GLuint colorTextureTarget;
  GLuint depthBuffer;
  int width;
  int height;
};
```

This `struct` is used to store the relevant parts of our post processing FBOs: the FBO id, the attached color texture, the depth renderbuffer, and also the size. Add this declaration at the start of the main file, somewhere before the variable declarations. Next we will introduce a function to initialize the data structure.

```
FboInfo createPostProcessFbo(int width, int height);
```

Declare this early in the file, such that it can be used in the rendering. Then you should implement it. Place the function body at the end of the file. Essentially all you need to do is copy the code you wrote to create the original FBO, but make sure to store the resulting ID's in an instance of `FboInfo`, which should be returned at the end of the function.

Next, to make sure this now works correctly, replace the `GLuint postProcessFrameBuffer`, and its associated texture, `texPostProcess`, with the new helper structure, e.g.:

```
FboInfo postProcessFbo;
```

Then also replace the initialization code in `initGL()`, and you'll probably have to update `display()` as well. After this you should be able to run your program and the result should still be the same. After all, the OpenGL commands issued should remain the same.

Now we are ready to introduce the two new FBOs we need to produce the blur, which is simply reduced to one line of code each, as shown below. Also add code to initialize these, they should have the same dimensions as the original post processing FBO.

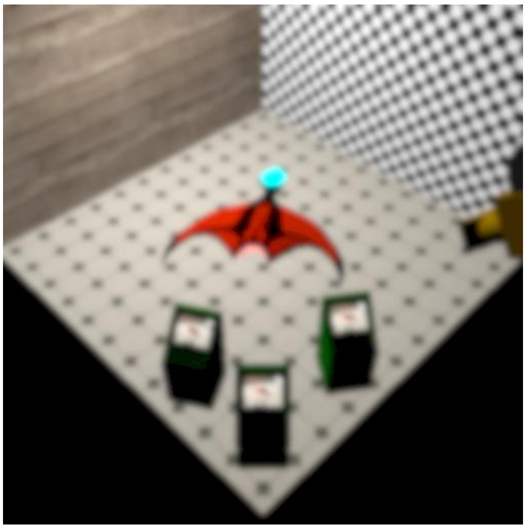
```
FboInfo horizontalBlurFbo;
FboInfo verticalBlurFbo;
```

In the tutorial we have provided you with shaders implementing the horizontal and vertical filter kernels, see `shaders/horizontal_blur.frag`, `shaders/vertical_blur.frag`. Load these together with the vertex shader`shaders/postFx.vert`, and store the references in variables named `horizontalBlurShader` and `verticalBlurShader`. To render the blur, use this algorithm:

1. Render a full-screen quad into the `horizontalBlurFbo`.
    a. Use the shader `horizontalBlurShader`.
    b. Bind the `postProcessFbo.colorTextureTarget` as input frame texture.
2. Render a full-screen quad into the `verticalBlurFbo`.
    a. Use the shader `verticalBlurShader`.
    b. Bind the `horizontalBlurFbo.colorTextureTarget` as input frame texture.

Implement this algorithm as a function. Call the function after the scene has been rendered into the post processing FBO. Now the `verticalBlurFbo.colorTextureTarget` contains the blurred version of the rendered image.

To check that the result is the expected, use `glBlitFramebuffer`, as shown earlier in the tutorial. If it works you should now see something like the image to the right. Nice and blurry.

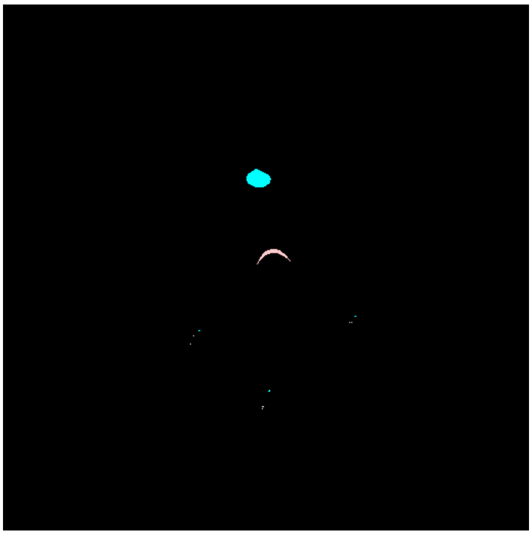


Now, blur itself is perhaps not the most desirable effect we can think of. Fortunately, we can use it to create one which is often pretty high on the list of demand:

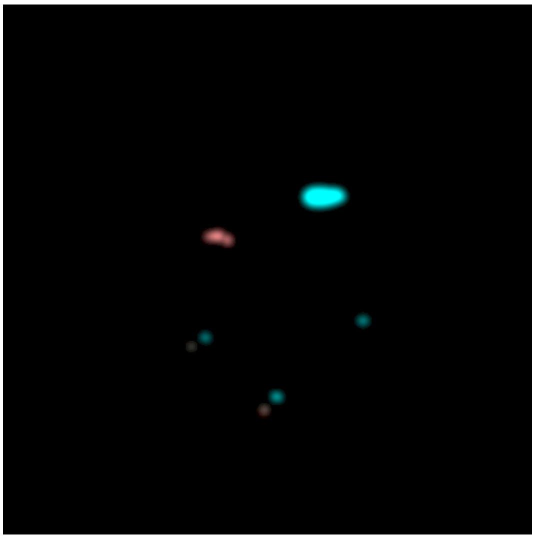
Bloom!

Bloom, or glow, makes bright parts of the image bleed onto darker neighboring bits. This creates an effect, akin to what our optical system produces when things are really bright. Therefore this can create the impression that parts of the image are far brighter than what can actually be represented on a screen. Cool. But how to do that?

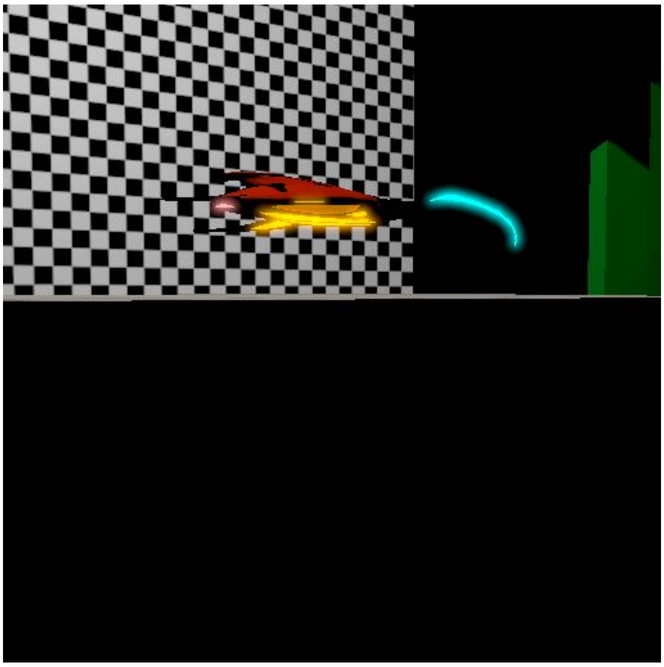
The only thing we really need to add is a *cutoff* pass, before blurring the image, to remove all the dark portions of the scene. There is a shader for this purpose too: `shaders/cutoff.frag`. Load the shader, create a new FBOInfo `cutoffFbo`, and draw a full-screen pass into it. When visualized using blit it should look like this:



Then use the `cutoffFbo` as input to the blur, which should produce a result, a lot like the last image on the right.



Finally, all we need to do is to add this to the, unblurred, frame buffer (which should still be untouched in `postProcessFbo`). This can be achieved by simply rendering a full screen quad using additive blending, into this frame buffer. Another way is to bind it to a second texture unit during the post processing pass, and sample and add in the post processing shader. In our case, this last should be the easiest option. The screen shot below shows the bloom effect, where the blooming parts are also boosted by a factor of to, to create a somewhat over the top bloom effect.



Further Reading

As always check out the course book, chapters?. The GPU Gems series is also a good bet, and available online (books 1-3, at least), GPU Gems 1 and GPU Gems 3, both dedicate a part to post-processing effects.

Further reading on Gaussian blurring can be found in the blog used as the source of coefficient calculation magic in this tutorial. Some additional information is available

Otherwise, Google tends to be useful with the right keywords, some of which you have hopefully picked up in this tutorial.