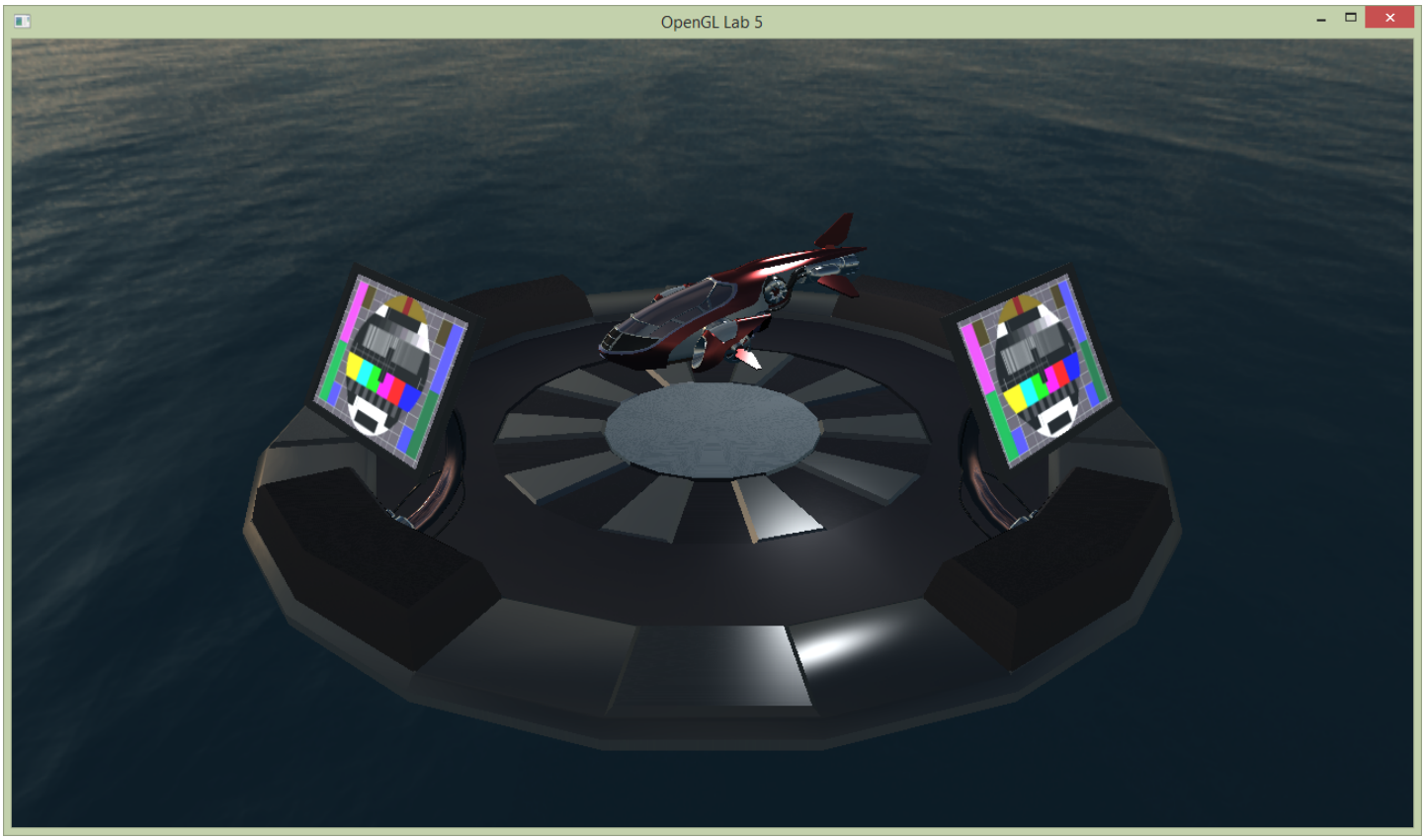
Tutorial 5: Render to Texture
Dynamically update textures and learn about post processing.

- Home

Introduction
In the previous tutorials, we have rendered directly to the default framebuffer. In this tutorial, we are going to introduce framebuffer objects, so that we can render to a texture (updating it every frame) and do post processing effects.

Run the code and observe the fighter on the landing pad. The shading is very dull. Start by copying your implementation of `calculateDirectIllumination()` and `calculateIndirectIllumination()` in `shading.frag` from tutorial 4 to the place-holder implementation in this tutorial (the shader is called `shading.frag` also in this tutorial). The result should look like below:



There are monitors beside the landing pad which currently have a static emissive texture showing a tv test image. We will replace this texture with a video feed from a security camera hoovering on the opposite side of the platform. This is done in two steps:

1. Render from the security cameras point of view to a framebuffer.
2. Use the color texture from the framebuffer as emissive texture.

First we need to make a framebuffer. We have provided a skeleton for making framebuffers in `FboInfo`, but it is not completed yet. In the constructor, we have generated two textures, one for color and one for depth, but we have not yet bound them together to a framebuffer.

Task 1: Setting up Framebuffer objects
Similarily to how we create a vertex array object and bind buffers to it (see tutorial 1 and 2), we create a framebuffer and attach textures to it. At task 1, generate a framebuffer and bind it:

```
// >>> @task 1
glGenFramebuffers(1, &framebufferId);
glBindFramebuffer(GL_FRAMEBUFFER, framebufferId);
```

Then attach the color texture as color attachment 0 (there may be many color attachments per framebuffer):

```
// bind the texture as color attachment 0 (to the currently bound framebuffer)
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, colorTextureTarget, 0);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
```

And attach the depth texture as the depth attachment (there can only be one per framebuffer):

```
// bind the texture as depth attachment (to the currently bound framebuffer)
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthBuffer, 0);
```

Now we have an easy way to create framebuffers. In the end of `initGL()`, there is a section devoted to creation of framebuffers. We will create five framebuffers and push them to the vector `fboList`, enough framebuffers for the mandatory and optional assignments in this tutorial.

```
int w, h;
SDL_GetWindowSize(g_window, &w, &h);
const int numFbos = 5;
for (int i = 0; i < numFbos; i++) {
    fboList.push_back(FboInfo(w, h));
}
```

We have now intialized framebuffers with the initial size of the window. The window may however be changed by the user, and we therefore reallocate the textures when the resolution changes. This is already performed in the beginning of `display()`. Have a look at how this is implemented.

Task 2: Rendering to the FBO
We will now render from the security cameras point of view, and we do this before we render from the users camera. Let's render to the first framebuffer in our `fboList`. Bind the framebuffer at `@task 2` with:

```
// >>> @task 2
FboInfo &securityFB = fboList[0];
glBindFramebuffer(GL_FRAMEBUFFER, securityFB.framebufferId);
```

And directly after you have bound the framebuffer, set the viewport and clear it:

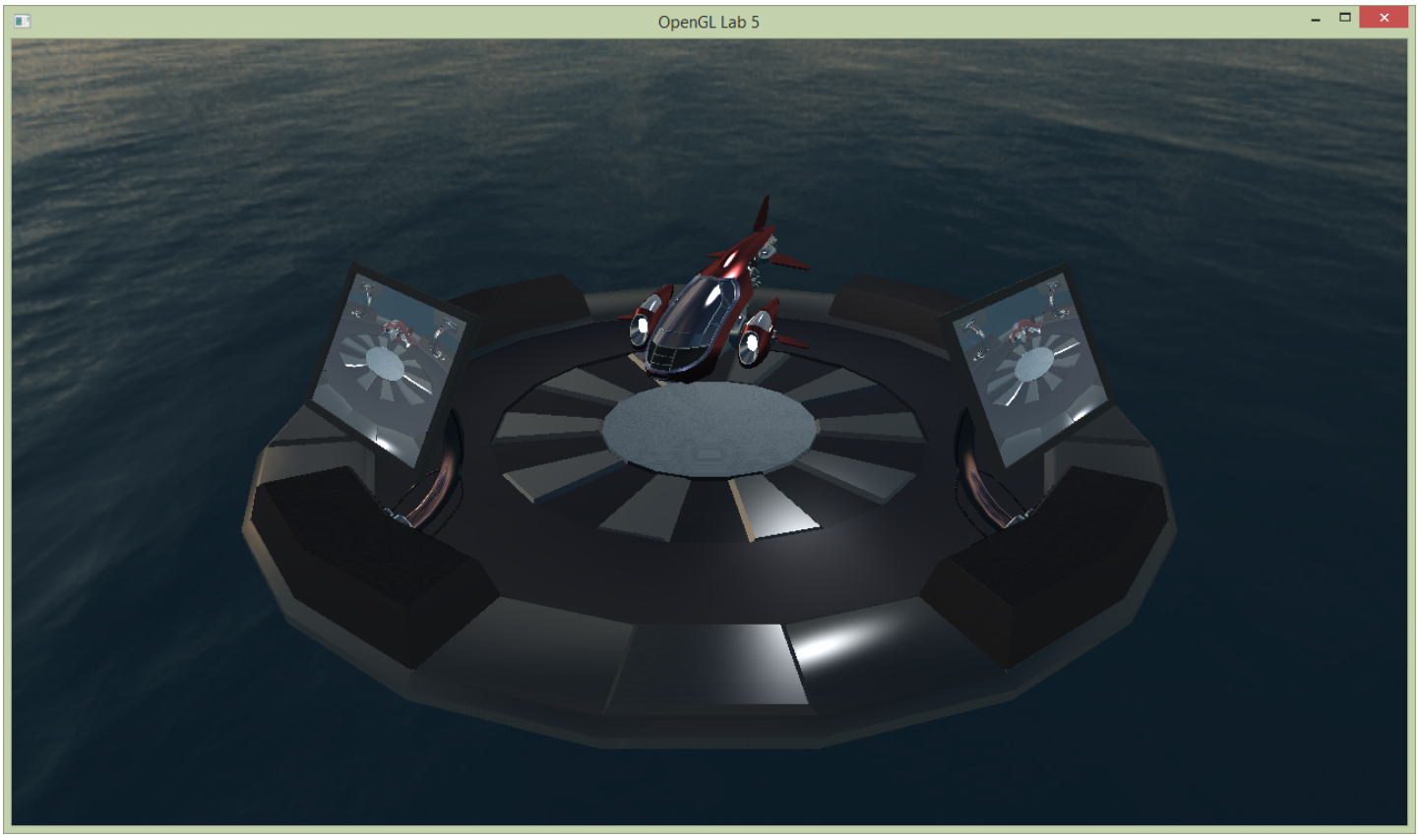
```
glViewport(0, 0, securityFB.width, securityFB.height);
glClearColor(0.2, 0.2, 0.8, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Now, render the scene to this framebuffer, just as we do from the camera, but skip the rendering of the security camera obj-model (since we are rendering from within it). Use the view and projection matrix that already is provided: `securityCamViewMatrix` and `securityCamProjectionMatrix`.

Now we are to use the color attachment as a texture (`emissiveMap` in `shading.frag`) when we render the landing pad from the users view below. We will just change the texture id used in the landing pad model.

```
labhelper::Material &screen = landingpadModel->m_materials[8];
screen.m_emission_texture.gl_id = securityFB.colorTextureTarget;
```

The result should look like below. You can change the security cameras direction by pressing the right mouse button and moving the mouse. Try it out!



Task 3: Rendering the FBO fullscreen
Let's now render scene as seen from the main camera on an FBO too. We will use this for the following tasks.

Start by replacing the default framebuffer with the next available FBO in `fboList`, and then render normally. If you run the application now you will see a black screen, since nothing has been rendered to the default framebuffer.

To see the scene rendered on the screen again:

1. Bind the default framebuffer and clear it.
2. Set `postFxShader` as the active shader program.
3. Bind the framebuffer texture to texture unit 0.
4. Draw a quad that covers the viewport to start a fragment shader for each pixel on the screen (see below).

To draw a quad to the screen you can use the already provided function `labhelper::drawFullScreenQuad();`

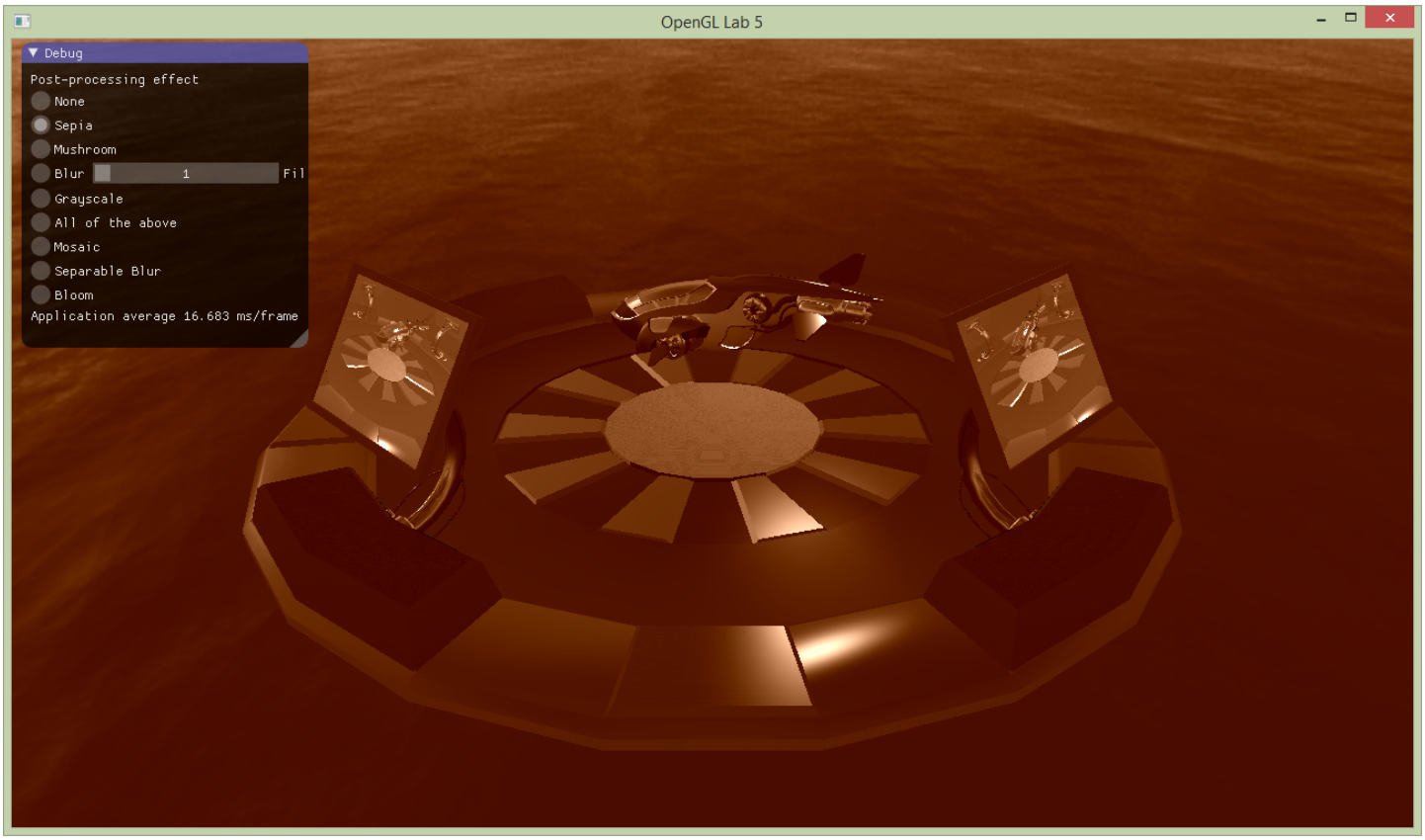
Task 4: Post processing
Post processing is perhaps the most common use for render to texture today, with the very likely exception of shadow maps. Most, if not all, games make use of a post-processing pass to change aspects of the look of the game, creating effects such as motion blur, depth of field, 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 fullscreen quad this is the same as each pixel.

There are multiple post processing effects implemented in the `postFxShader`, the shader we used in the previous task to draw the framebuffer to the fullscreen. To change the effect used and pass some needed parameters to these effects we are going to need to set some uniforms for the shader. Add the following code after `postFxShader` has been set as active and before drawing the fullscreen quad:

```
labhelper::setUniformSlow(postFxShader, "time", currentTime);
labhelper::setUniformSlow(postFxShader, "currentEffect", currentEffect);
labhelper::setUniformSlow(postFxShader, "filterSize", filterSizes[filterSize - 1]);
```

The currently used effect can be controlled with the uniform `currentEffect` which is set from the gui. **You can toggle gui visibility by pressing G** (you can also comment out the `if` surrounding the call to `gui()` in `main()`). With the Sepia post processing, that mimics a toning technique of black-and-white photography, the result looks like below.



Inspect the postFx.frag shader again. 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 helper function to access the texture:

```
vec4 textureRect(in sampler2D tex, vec2 rectangleCoord)
{
return texture(tex, rectangleCoord / textureSize(tex, 0));
}
```

This allows us to sample the texture with pixel coordinates of the glsl built in variable gl_FragCoord as texture coordinates, which supplies the screen space coordinates (within the render target) of the fragment being shaded. Normally, textures are sampled with coordinates in the range [0, 1].

The functions are used from the main function in the shader, try out different ones, and combine them. Notice the effect which is a 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);
```

Uses a primitive box filter to blur the image. This method is low quality and expensive; test using a large filter and note the FPS counter in the debug overlay. 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.

Task 5: Post processing - Mosaic
You shall now add another effect. The effect is called Mosaic and the result is shown below. 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 explain what you have done.

Task 6: [Optional] Efficient Blur
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.

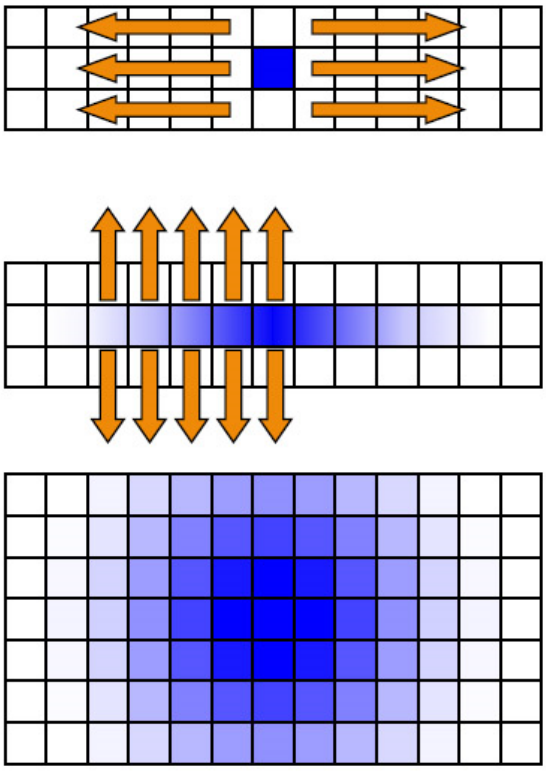


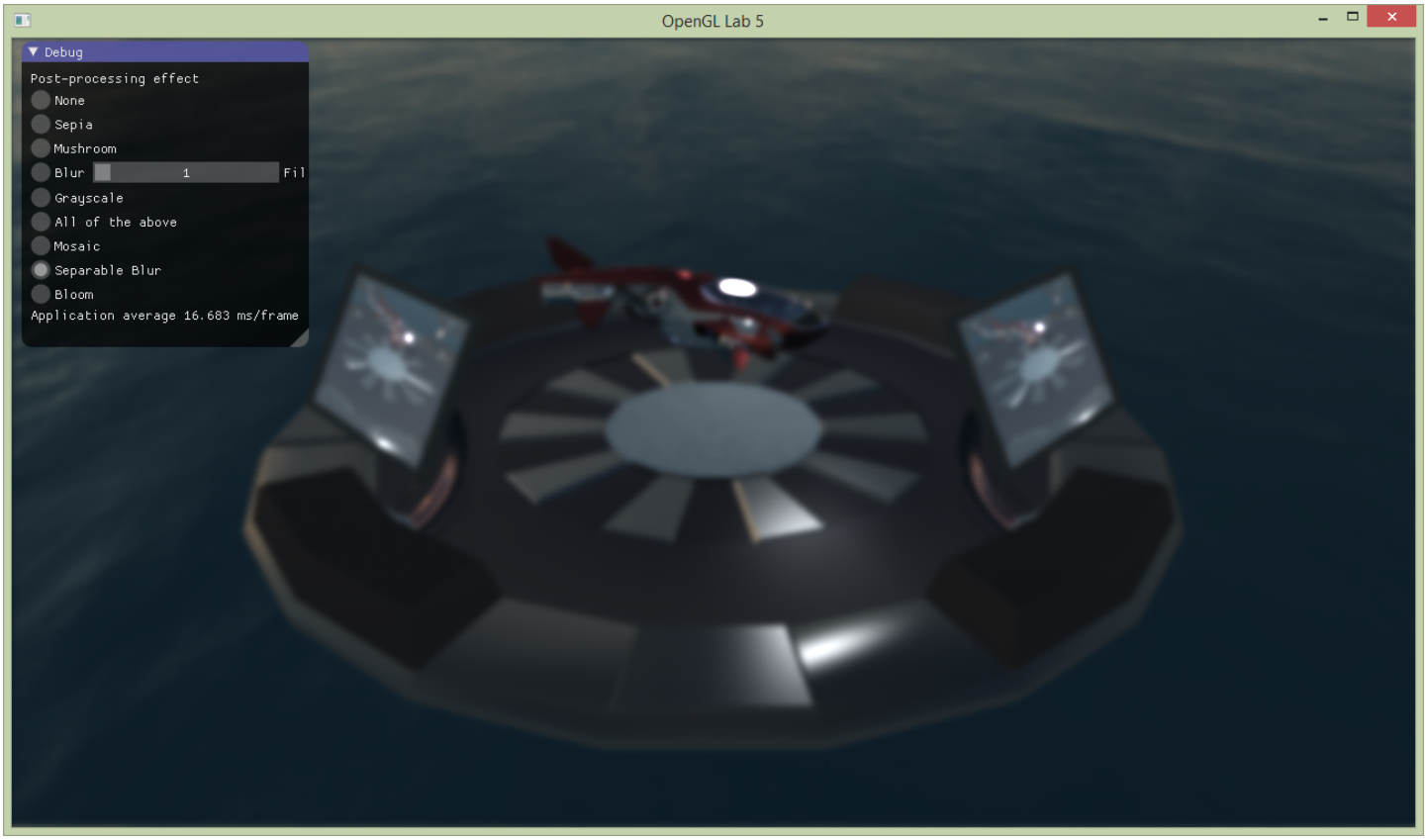
Image taken from ATI's presentation

To implement this in our tutorial, we will use two more 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.

We have provided you with shaders implementing the horizontal and vertical filter kernels, see `shaders/horizontal_blur.frag` and `shaders/vertical_blur.frag`. Load these together with the vertex `shadershaders/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 an fbo (here called `horizontalBlurFbo`).
    a. Use the shader horizontalBlurShader.
    b. Bind the postProcessFbo.colorTextureTarget as input frame texture.
2. Render a full-screen quad into an fbo (here called `verticalBlurFbo`).
    a. Use the shader verticalBlurShader.
    b. Bind the horizontalBlurFbo.colorTextureTarget as input frame texture.

Perform this before the post processing. Now the verticalBlurFbo.colorTextureTarget contains the blurred version of the rendered image. Bind it to texture unit 1, and sample from this unit in the post processing shader when the separable blur effect is selected. The effect looks as below:

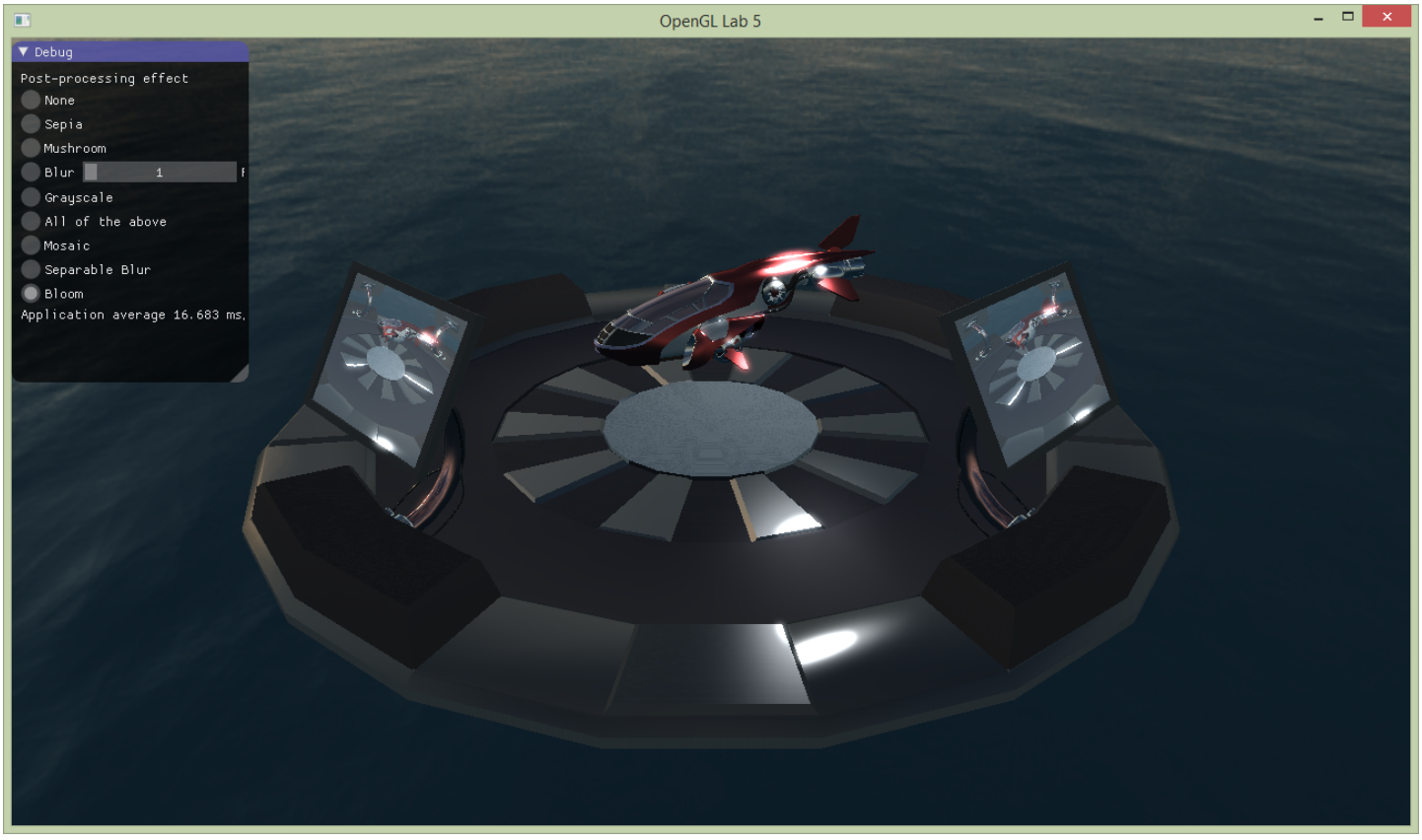


Task 7: [Optional] 6: Bloom
Bloom, or glow, makes bright parts of the image bleed onto darker neighboring bits. This creates an effect akin to what our optical system produces when things are really bright. Therefore this can create the impression that parts of the image are far brighter than what can 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, use the fifth created FBO (here called `cutoffFbo`), and draw a full-screen pass into it. When visualized it should look like this:

Then use the cutoffFbo as input to the blur, which should produce a result, a lot like the image below.

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.



When done, show your result to one of the assistants. Ask them more about post processing!