MVD: Advanced Graphics 1

14 - Render to Texture

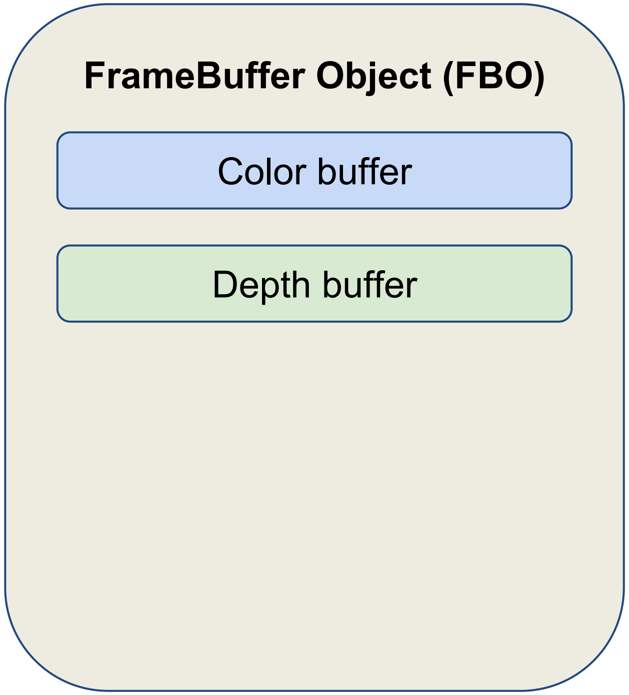
aluntthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Framebuffer

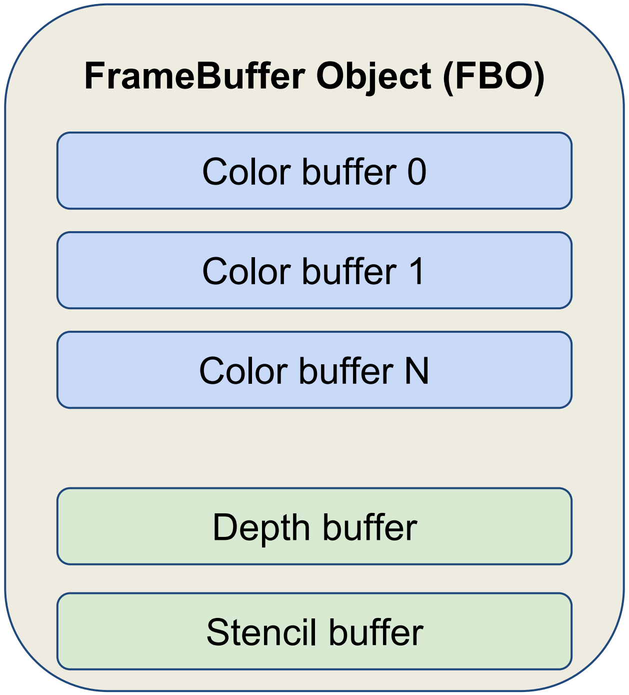
A framebuffer is a collection of different buffer that contain information that we use to render the scene.

So far we have only used a single colour buffer, and the depth buffer



Framebuffer

But you can multiple colour buffer, as well as the depth buffer and also a stencil buffer



FrameBuffer Object (FBO)

Color buffer 0

Color buffer 1

Color buffer N

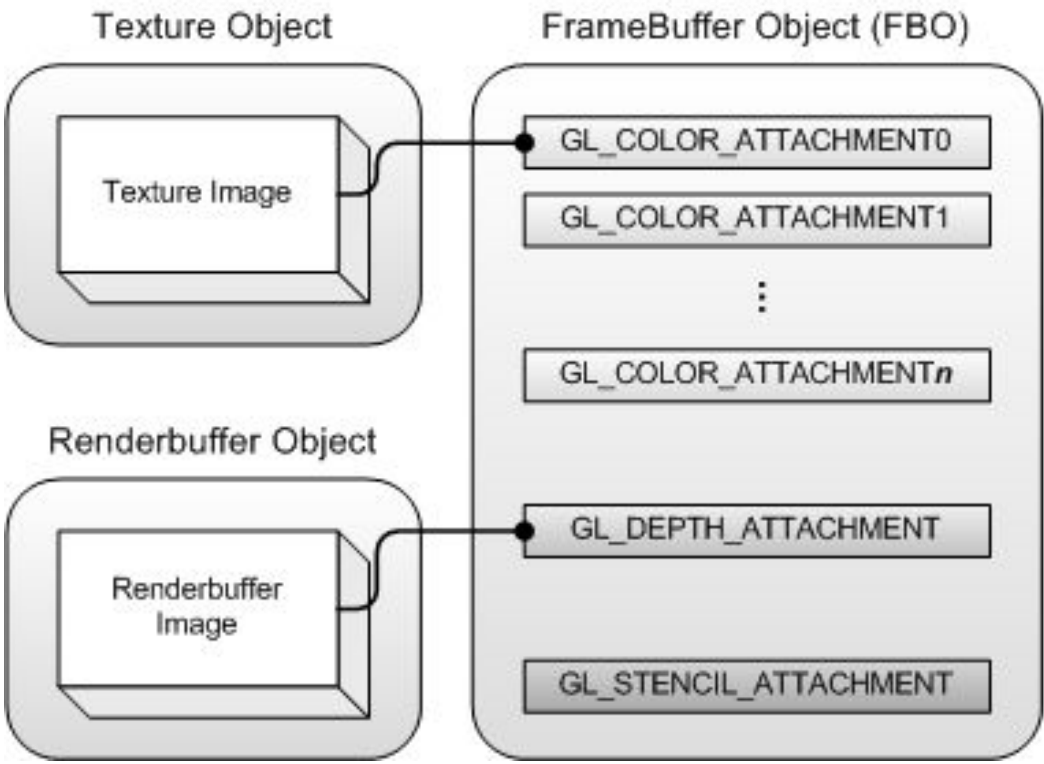
Depth buffer

Stencil buffer

Framebuffer vs renderbuffer

A renderbuffer object is a native opengl format for super fast writing

It is used for **depth and stencil attachments**



Framebuffer 0

Framebuffer 0 is **the screen**

By default, when we call drawElements we are drawing directly to framebuffer 0, which gets piped directly to the screen.

But there are *N* framebuffers available. We can draw to those framebuffers and **save their result in a texture object.**

(In fact, we can output to several different textures per draw!)

Why framebuffers?

Framebuffers allow us to render to textures, which can then be composed when drawing the final image to screen

Post-processing effects

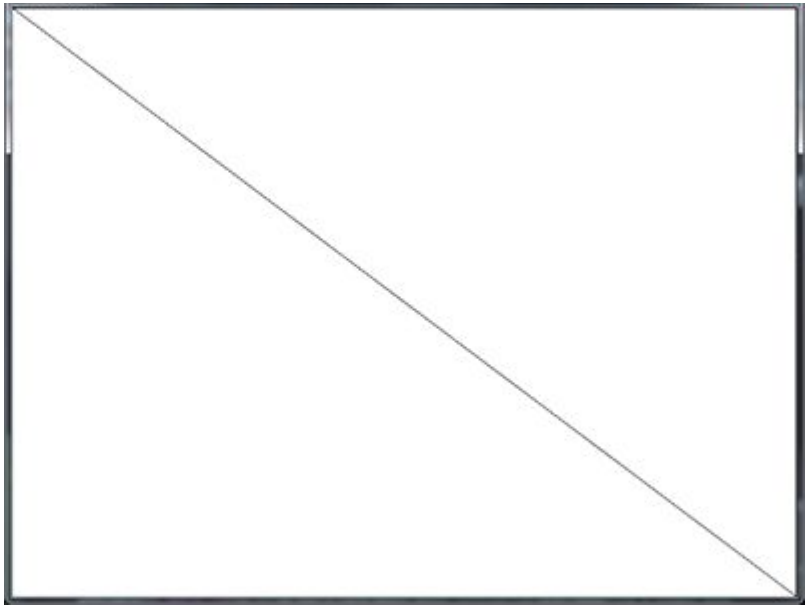
Shadow mapping

Deferred rendering

Screen space, textured quad

Framebuffers > 0 are, by default, hidden. The easiest way to view the framebuffer result is to pass its output as a texture to a screen-space quad

This is a simple quad from -1 to +1 in x and y



Task: draw screen space quad

Look at GraphicsUtilities::createPlaneGeometry()

Create a screen space shader to paint red

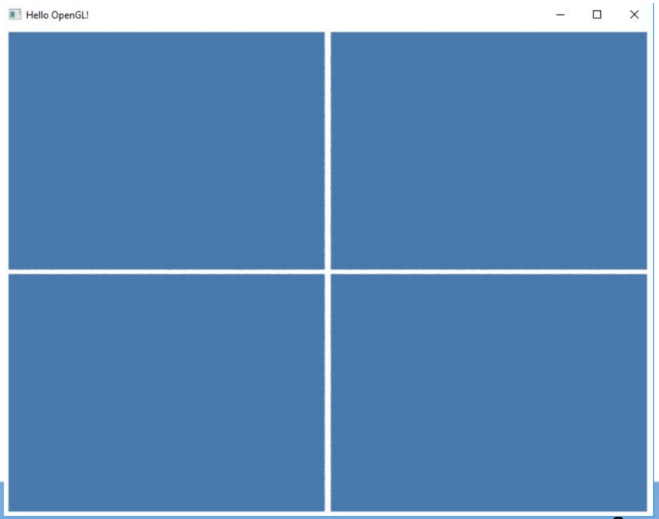
create a private member variables in graphics system

- to store plane geometry
- to store shader

init new geometry and shader;

render in update - disable depthtest

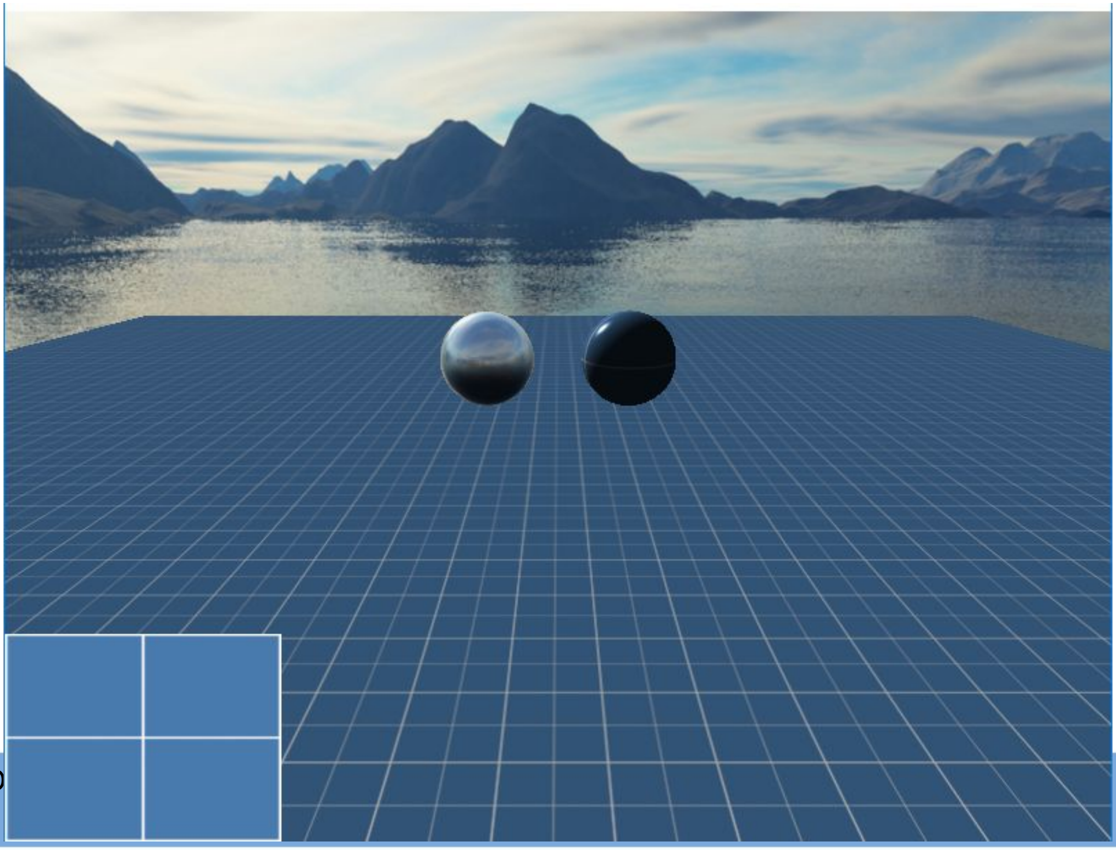
modify shader to draw test texture



Drawing to portion of screen

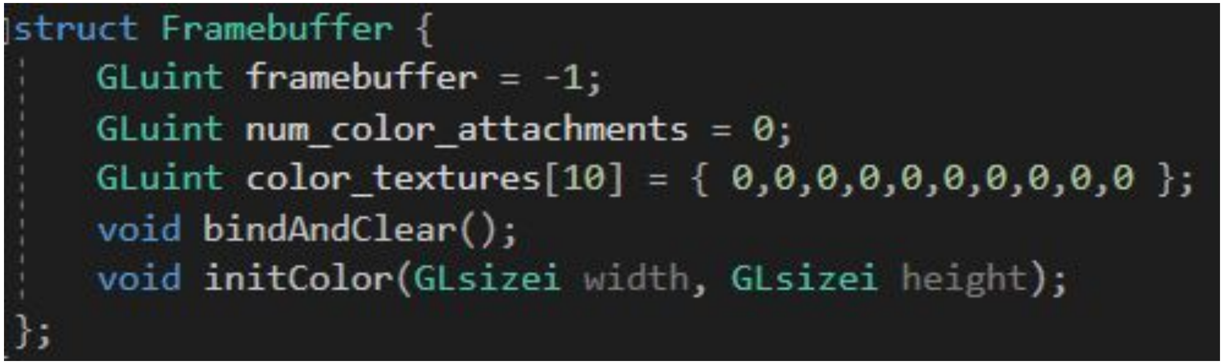
glViewport(bottom, left, width, height);

e.g. divide width and height by 4:

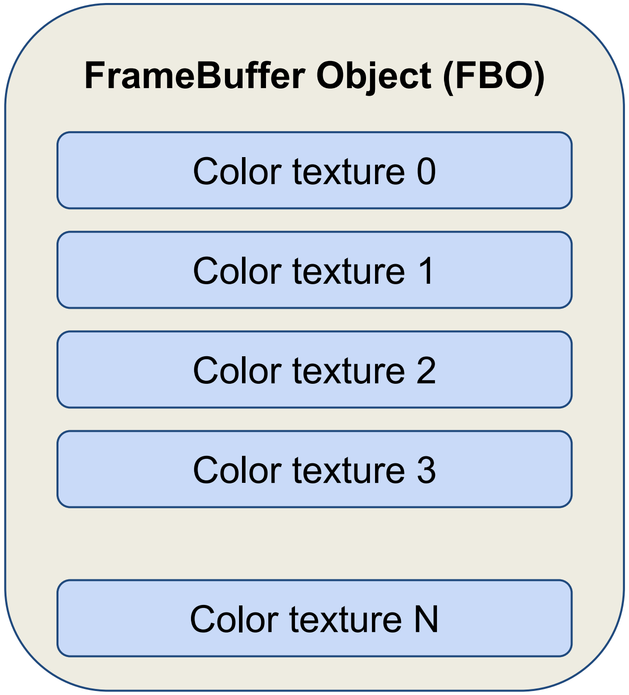


Creating a frame buffer

Define a new struct



```
struct Framebuffer {
    GLuint framebuffer = -1;
    GLuint num_color_attachments = 0;
    GLuint color_textures[10] = { 0,0,0,0,0,0,0,0,0,0 };
    void bindAndClear();
    void initColor(GLsizei width, GLsizei height);
};
```



FrameBuffer Object (FBO)

Color texture 0

Color texture 1

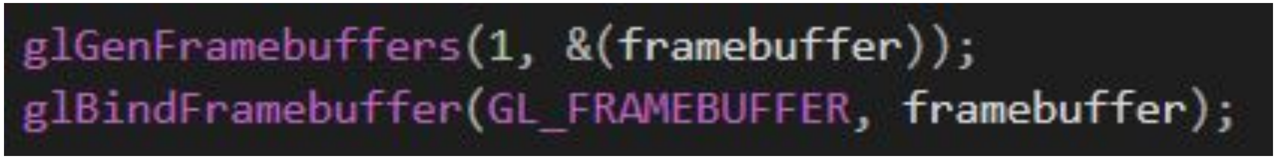
Color texture 2

Color texture 3

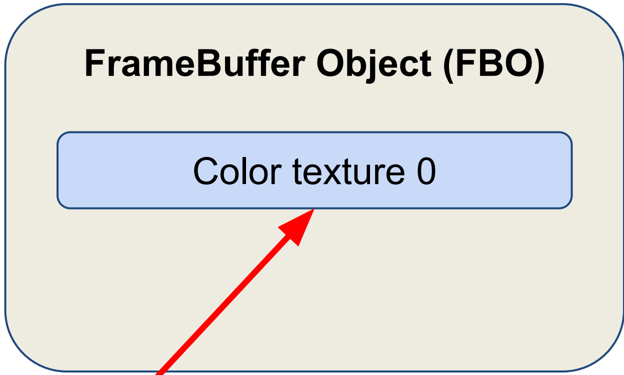
Color texture N

Creating a texture for the color buffer

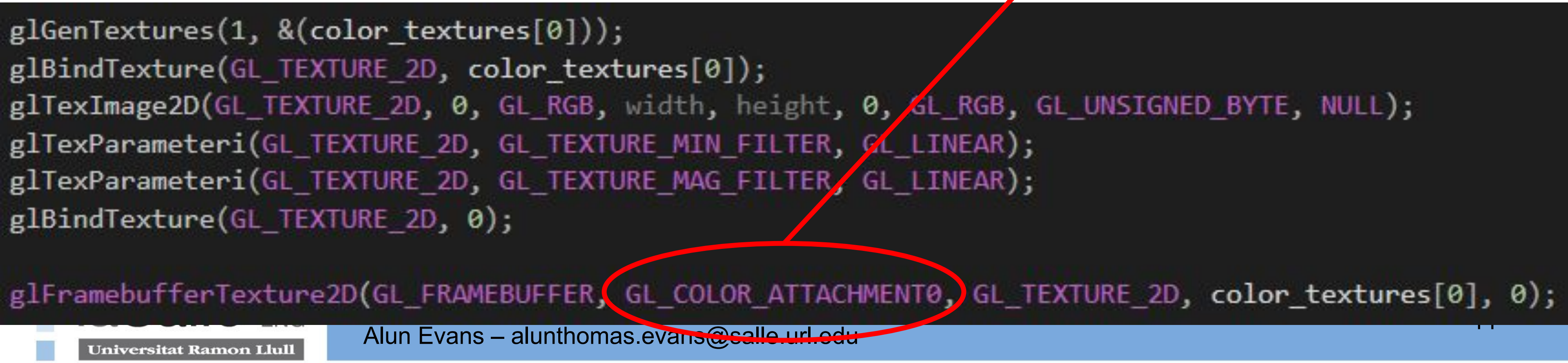
Create framebuffer



```
glGenFramebuffers(1, &(framebuffer));
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```



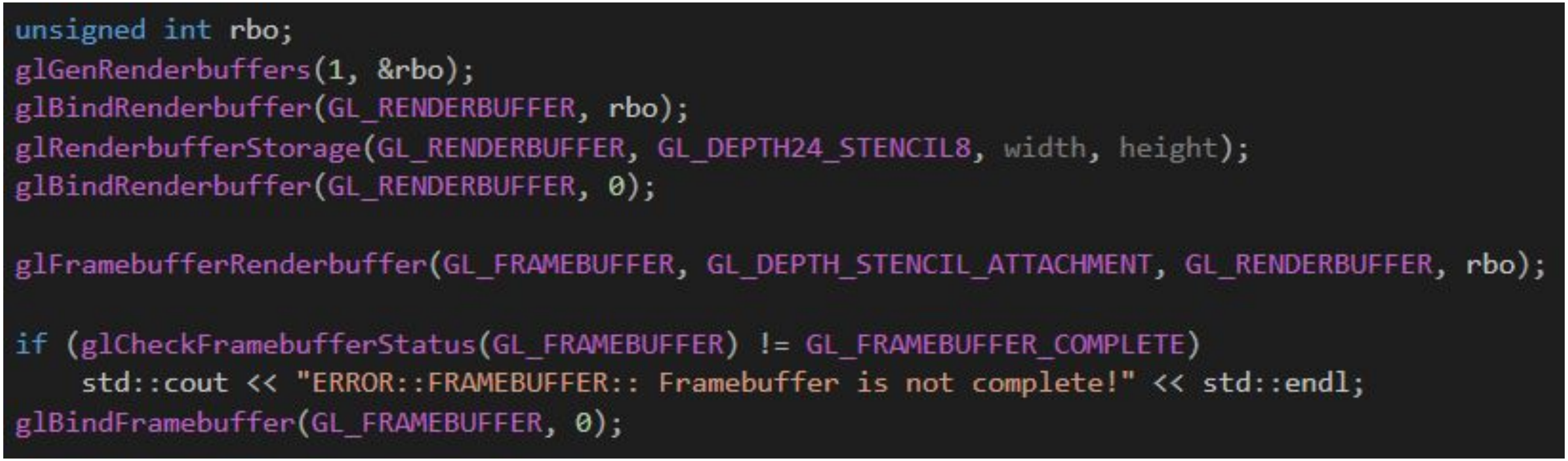
Create a single texture, **bind it to slot 0**



```
glGenTextures(1, &(color_textures[0]));
glBindTexture(GL_TEXTURE_2D, color_textures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, color_textures[0], 0);
```

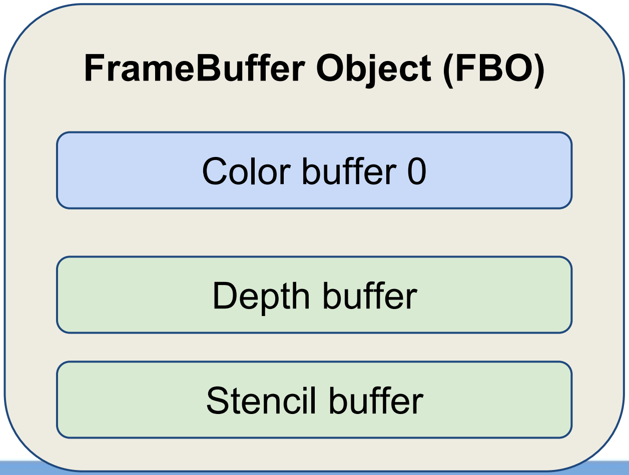
Creating a render buffer for depth



```
unsigned int rbo;
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, width, height);
glBindRenderbuffer(GL_RENDERBUFFER, 0);

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

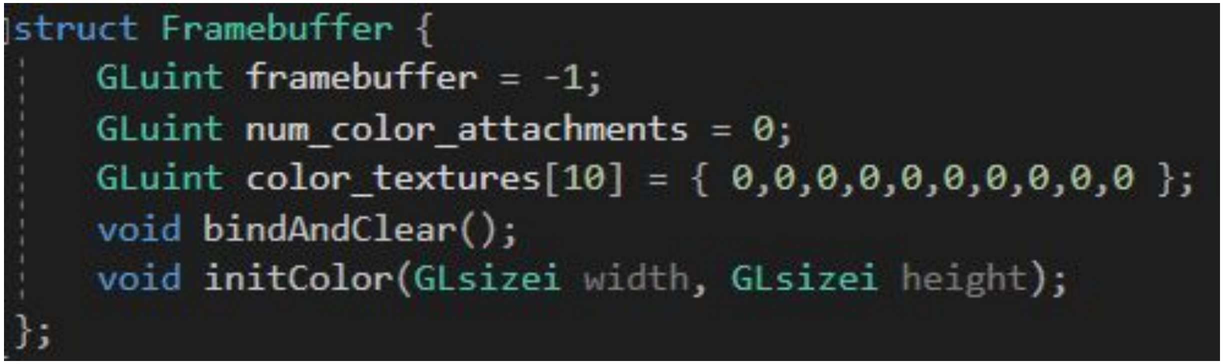


FrameBuffer Object (FBO)

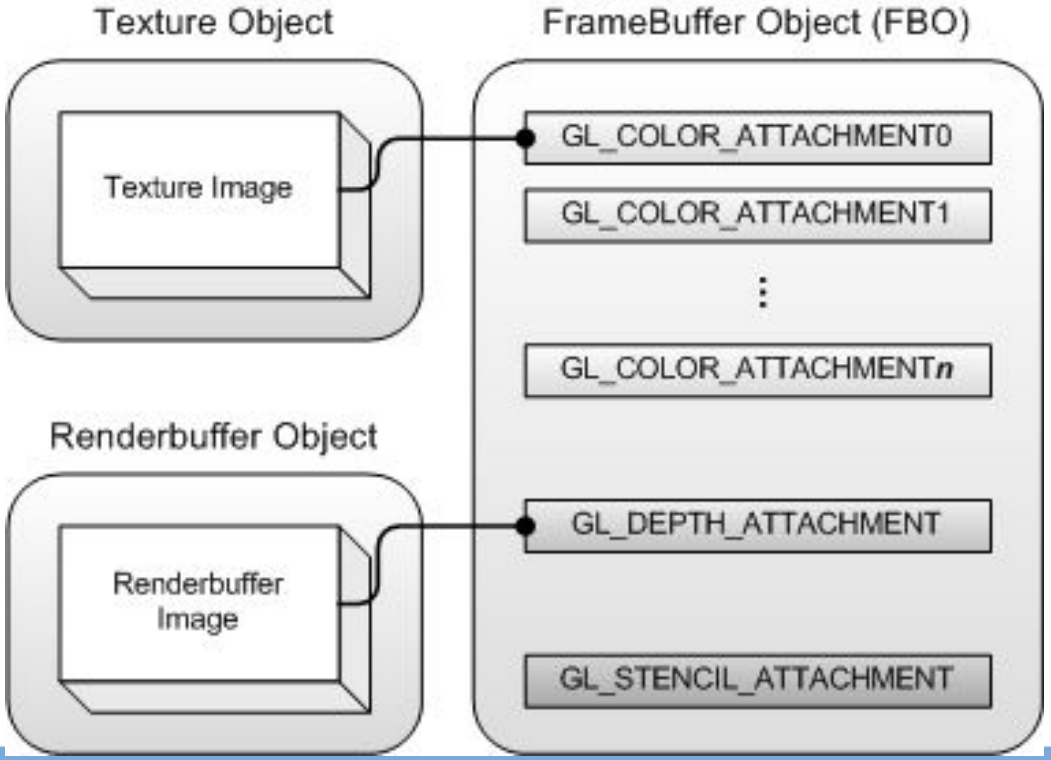
Color buffer 0

Depth buffer

Stencil buffer

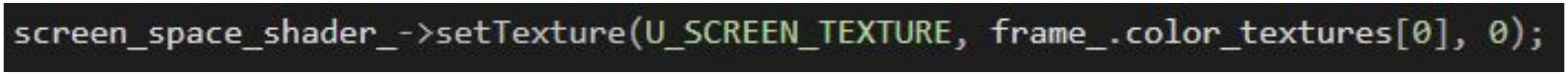


```
struct Framebuffer {
    GLuint framebuffer = -1;
    GLuint num_color_attachments = 0;
    GLuint color_textures[10] = { 0,0,0,0,0,0,0,0,0,0 };
    void bindAndClear();
    void initColor(GLsizei width, GLsizei height);
};
```



Task:Create frame buffer

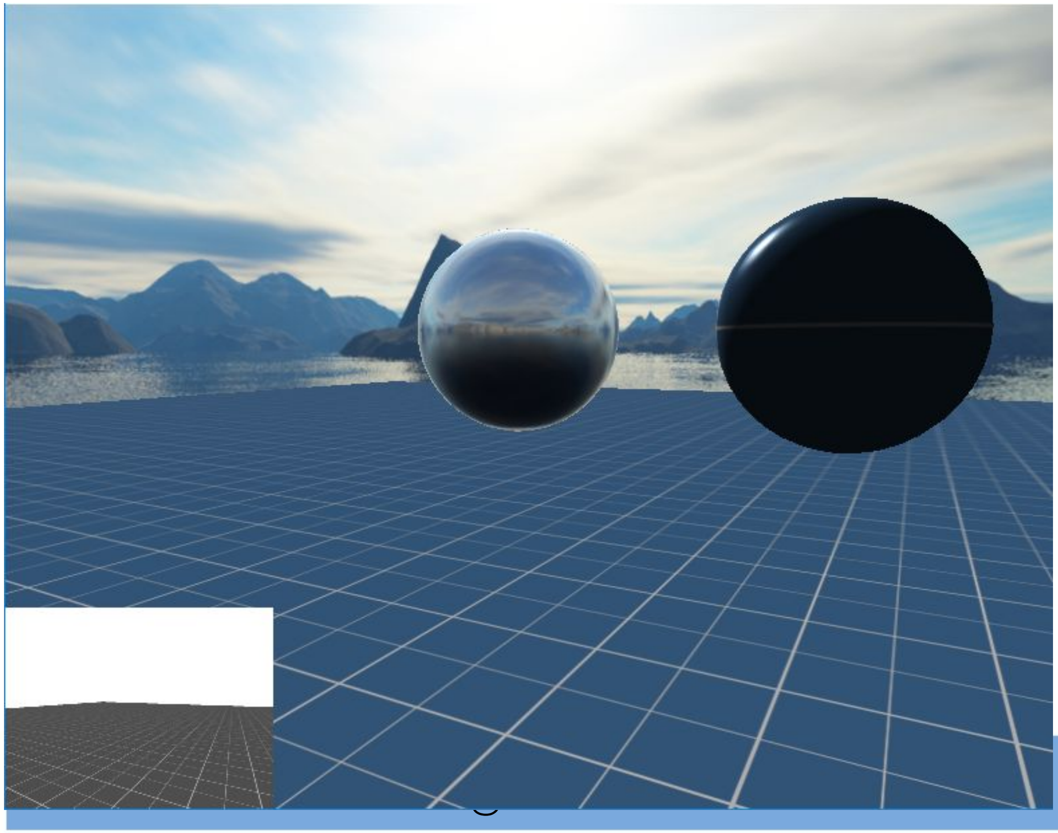
1. fill Framebuffer::initColor with correct code
2. Add a Framebuffer member variable to Graphics System
3. call initColor on that member variables in GraphicsSystem:init
4. render scene twice - first after calling bindAndClear of framebuffer; second after calling bindandClearScreen
5. Send frame texture to screen shader when drawing screen quad



```
screen_space_shader_->setTexture(U_SCREEN_TEXTURE, frame_.color_textures[0], 0);
```

Selectively rendering meshes

By adding a flag to the mesh component, you can select which meshes to render to scene



Post processing

Post-processing in graphics involves render a scene to a texture, then rendering that texture to a screen quad, applying a certain image-based filter

Can you think of post-processing effects?

Post processing

Post-processing in graphics involves render a scene to a texture, then rendering that texture to a screen quad, applying a certain image-based filter

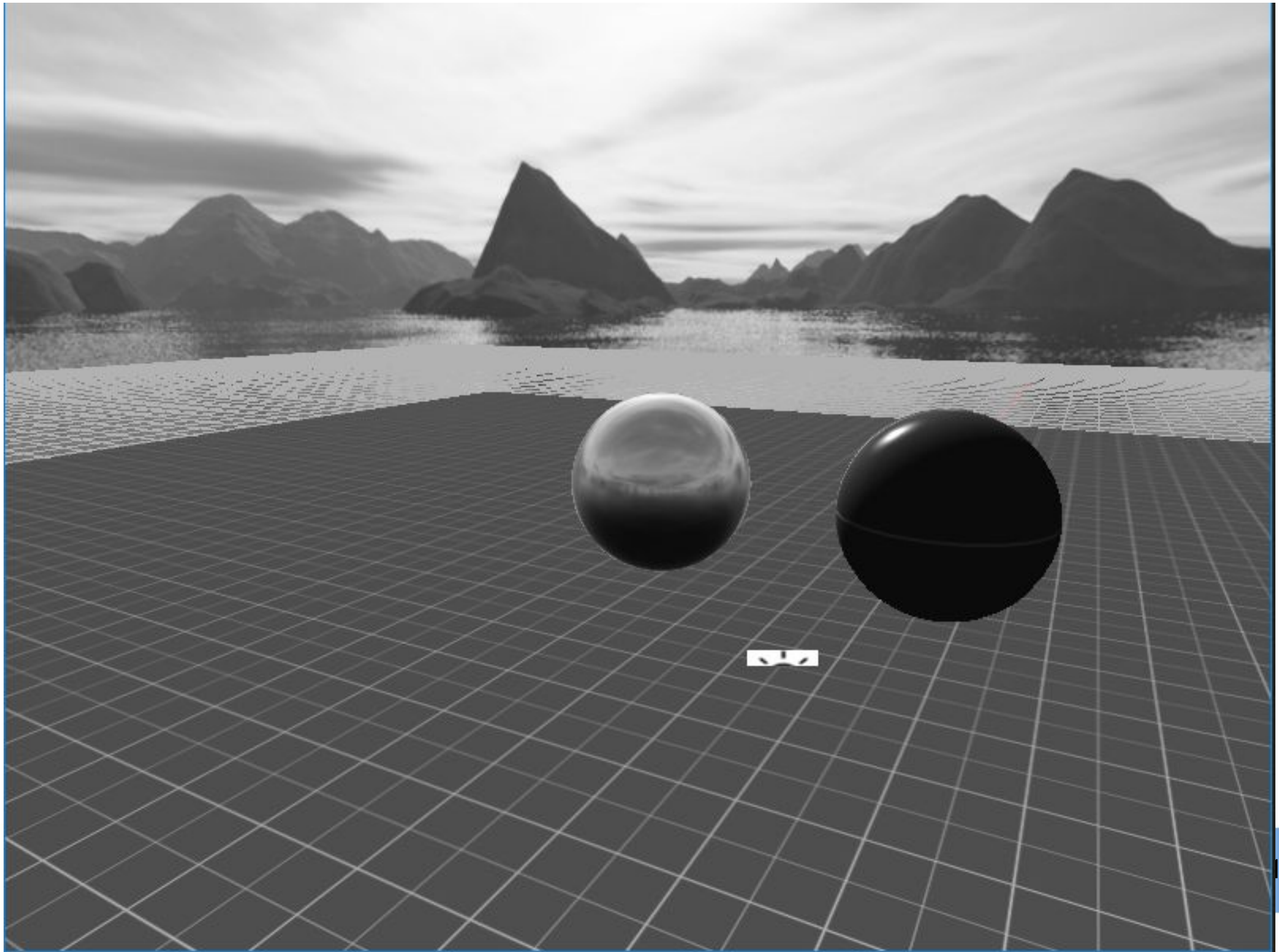
Can you think of post-processing effects?

Simple: Colour grading, dithering, grain, blur, edge filtering

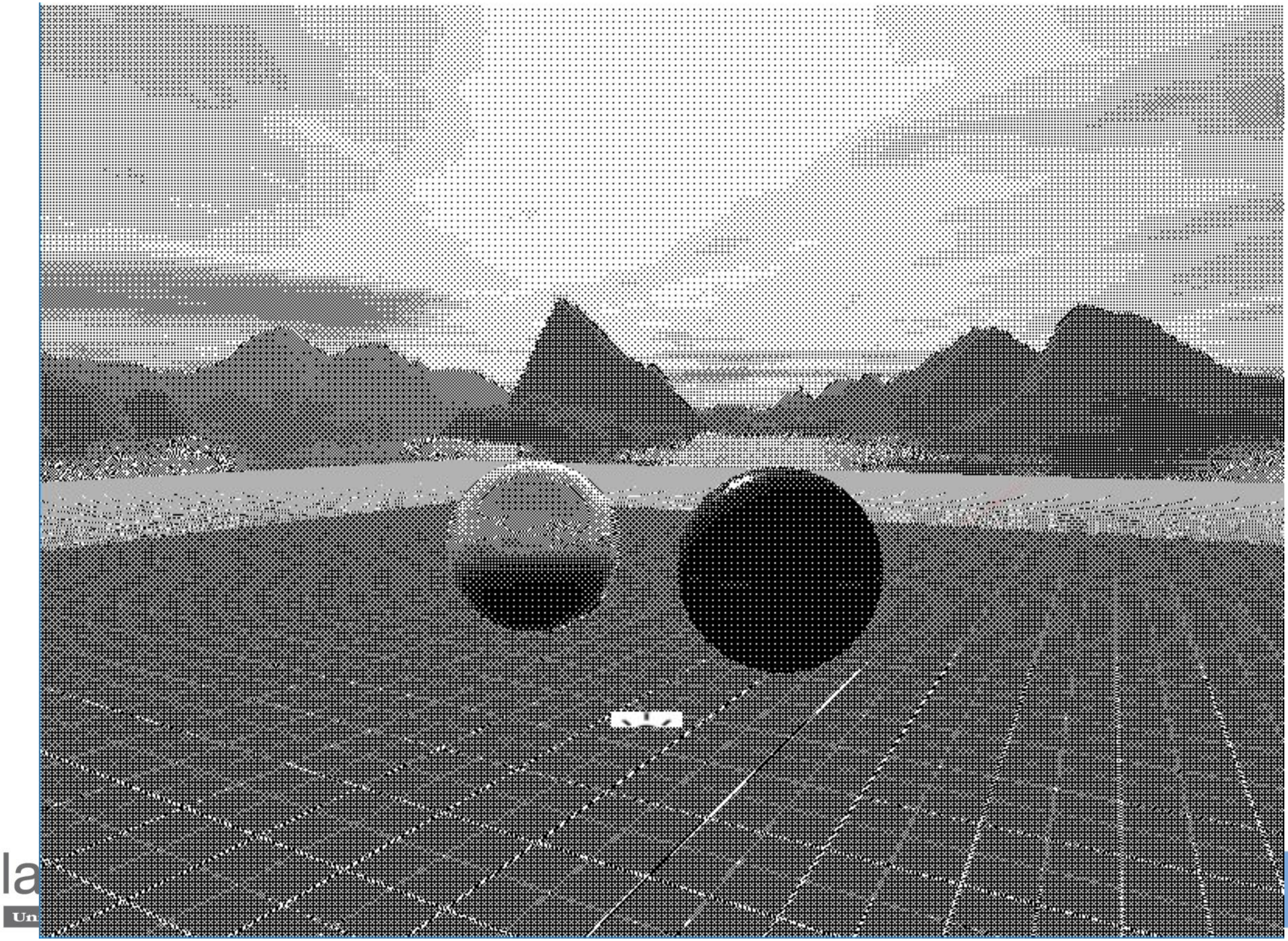
Complex: bloom, depth of field, motion blur, ambient occlusion

Simple B&W colour grade

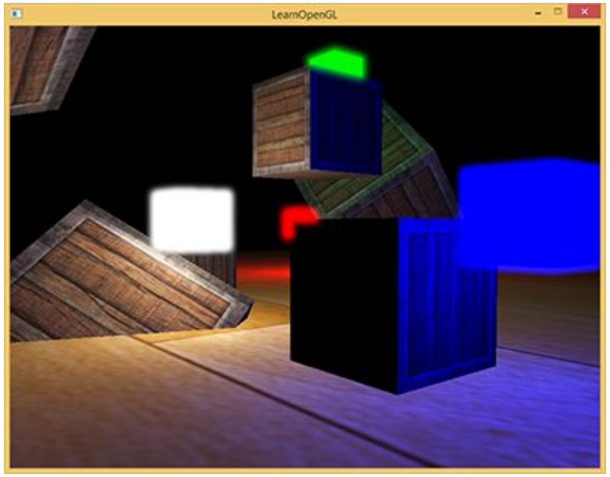
```
float average = 0.2126 * col.r + 0.7152 * col.g + 0.0722 * col.b;
```



B&W with Dithering



Bloom



Bloom makes objects glow

The basic approach is

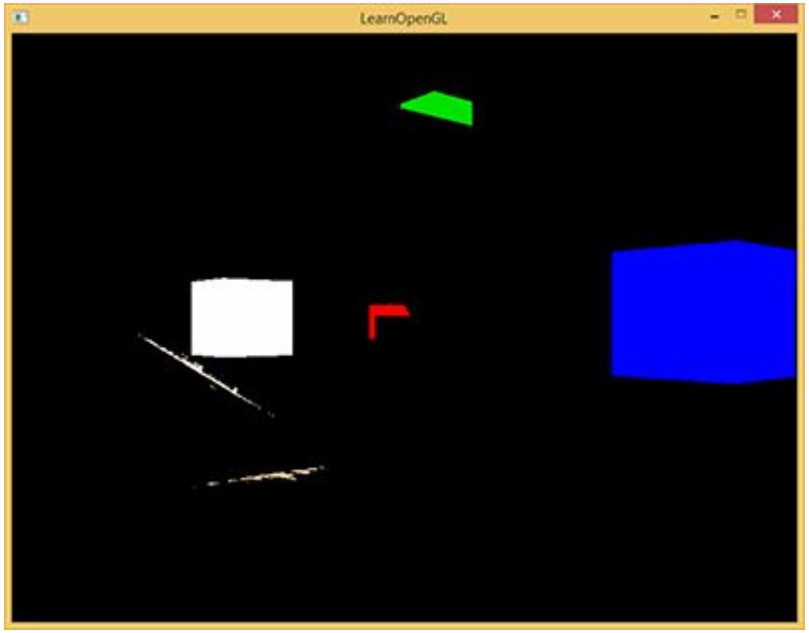
Render bloomable objects to texture

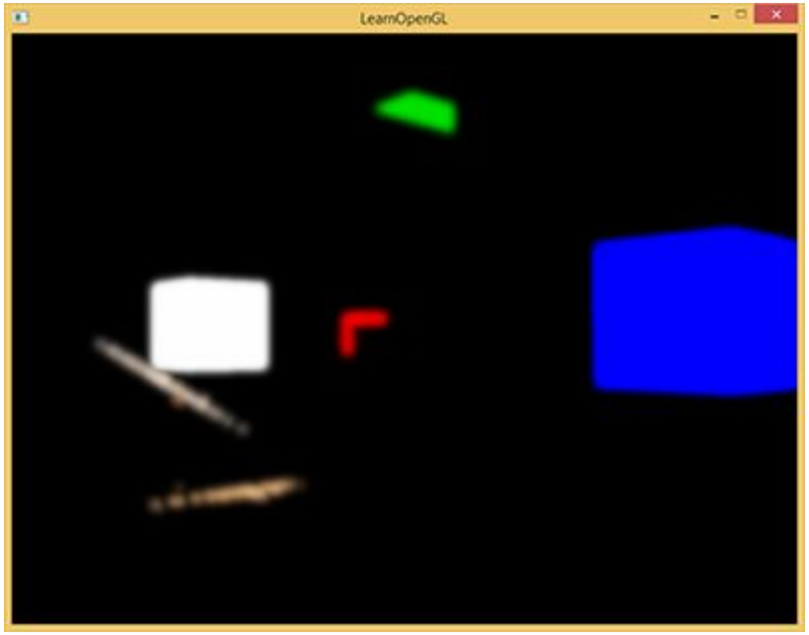
Blur that texture

Mix final scene with blurred texture

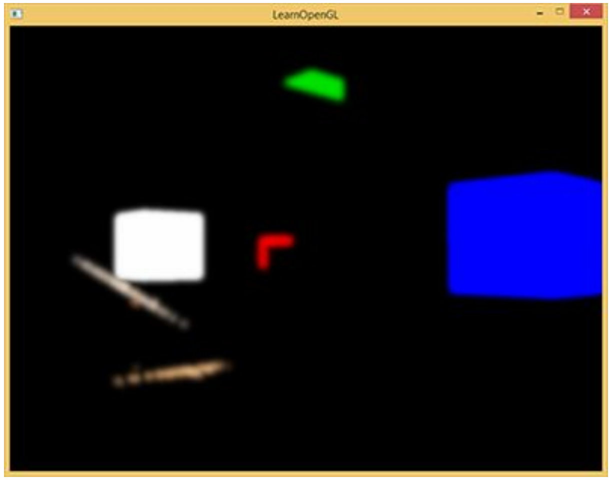
Render bloomable objects to texture

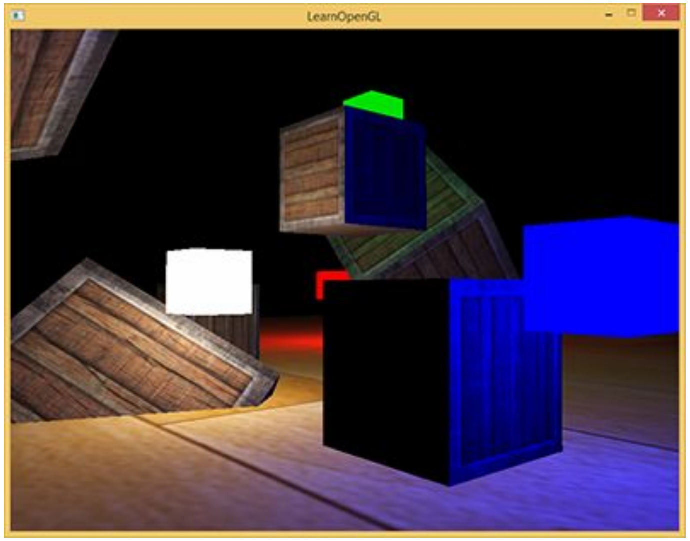
Bloomable objects - either selected manually OR thresholded pixels from image

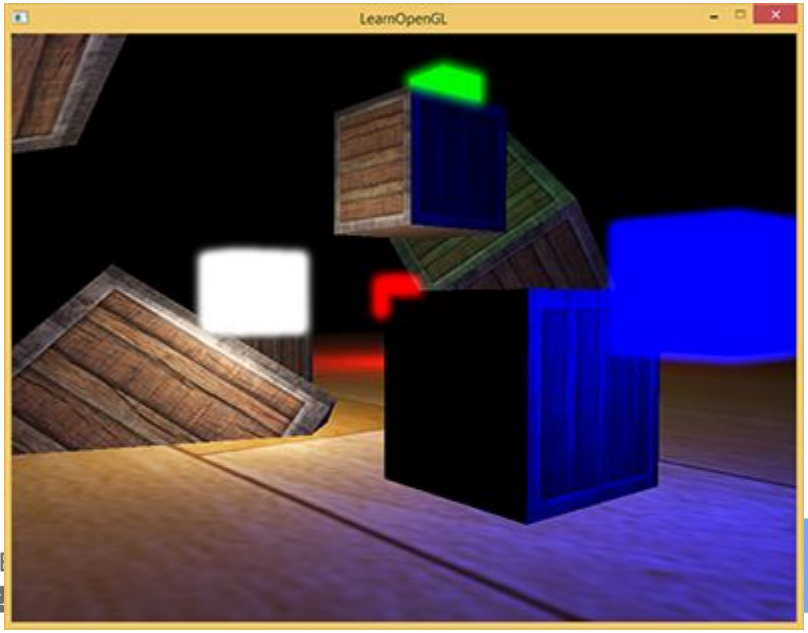




Mix blurred texture with scene

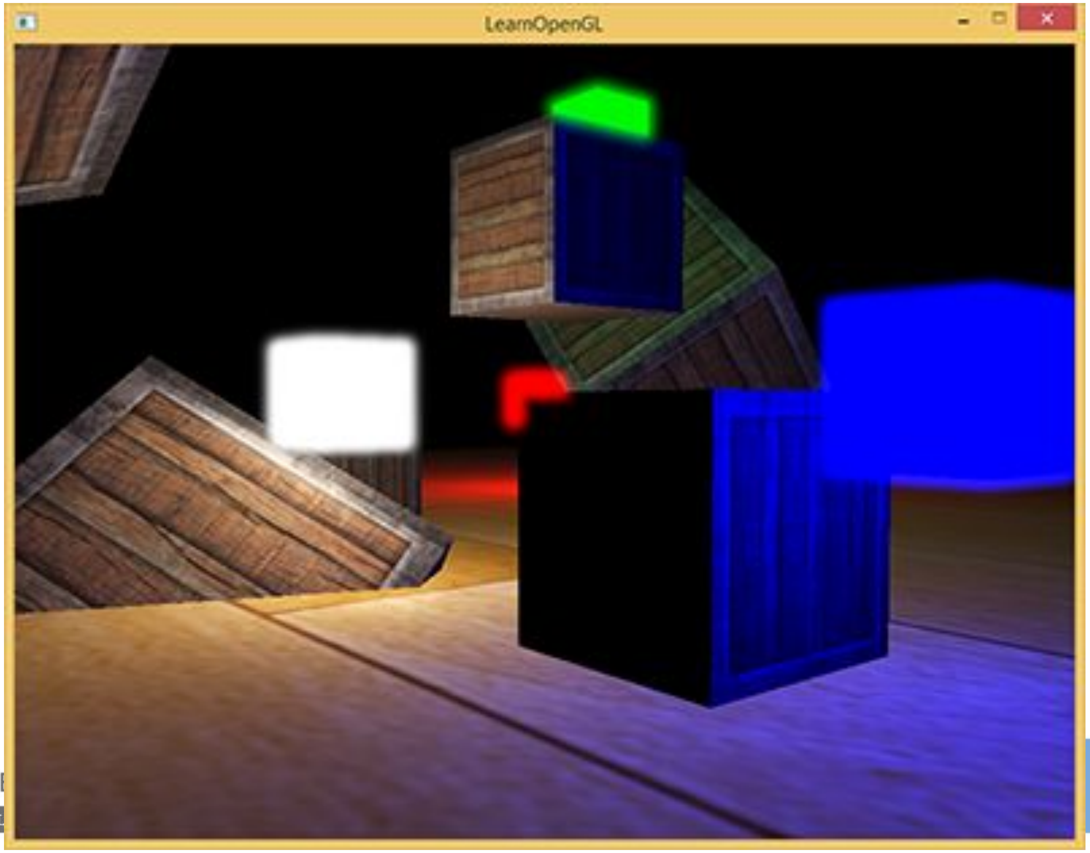






Bloom works best when you mix object with a point light

+ add a dark background



1st deliverable: post-processing

Write a post-processing pipeline.

Could be a *colour grading* shader, or *dither* or *grain*

- use imGUI to change variables in real time

Or go for a bloom effect

RECOMMENDED