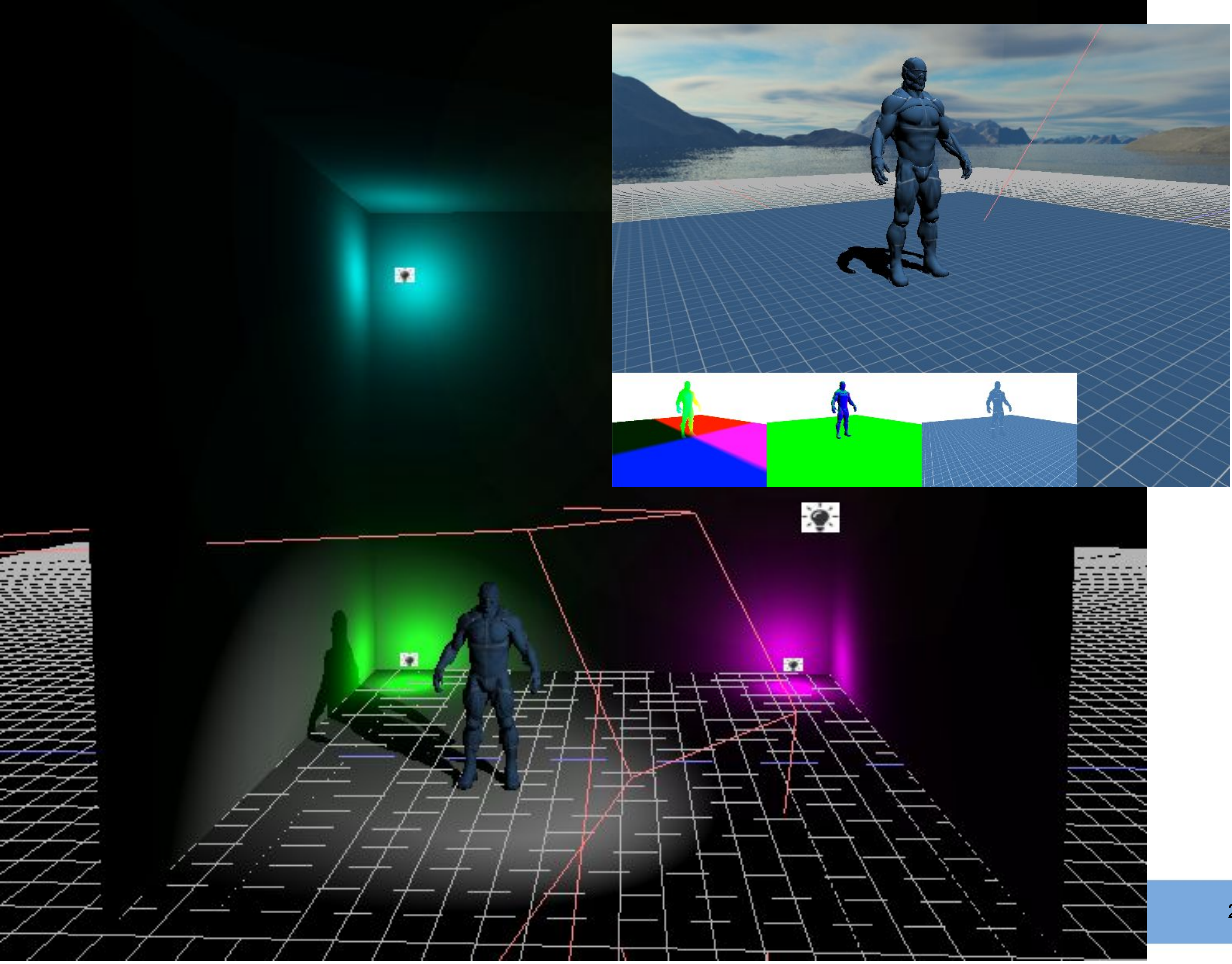
MVD: Advanced Graphics 1

16 - Deferred Rendering

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull



Deferred Rendering / Shading

Defer = "***delay until later***"

Deferred Rendering

Screen-space technique that calculates final colours of fragments by using input from textures drawn in previous pass

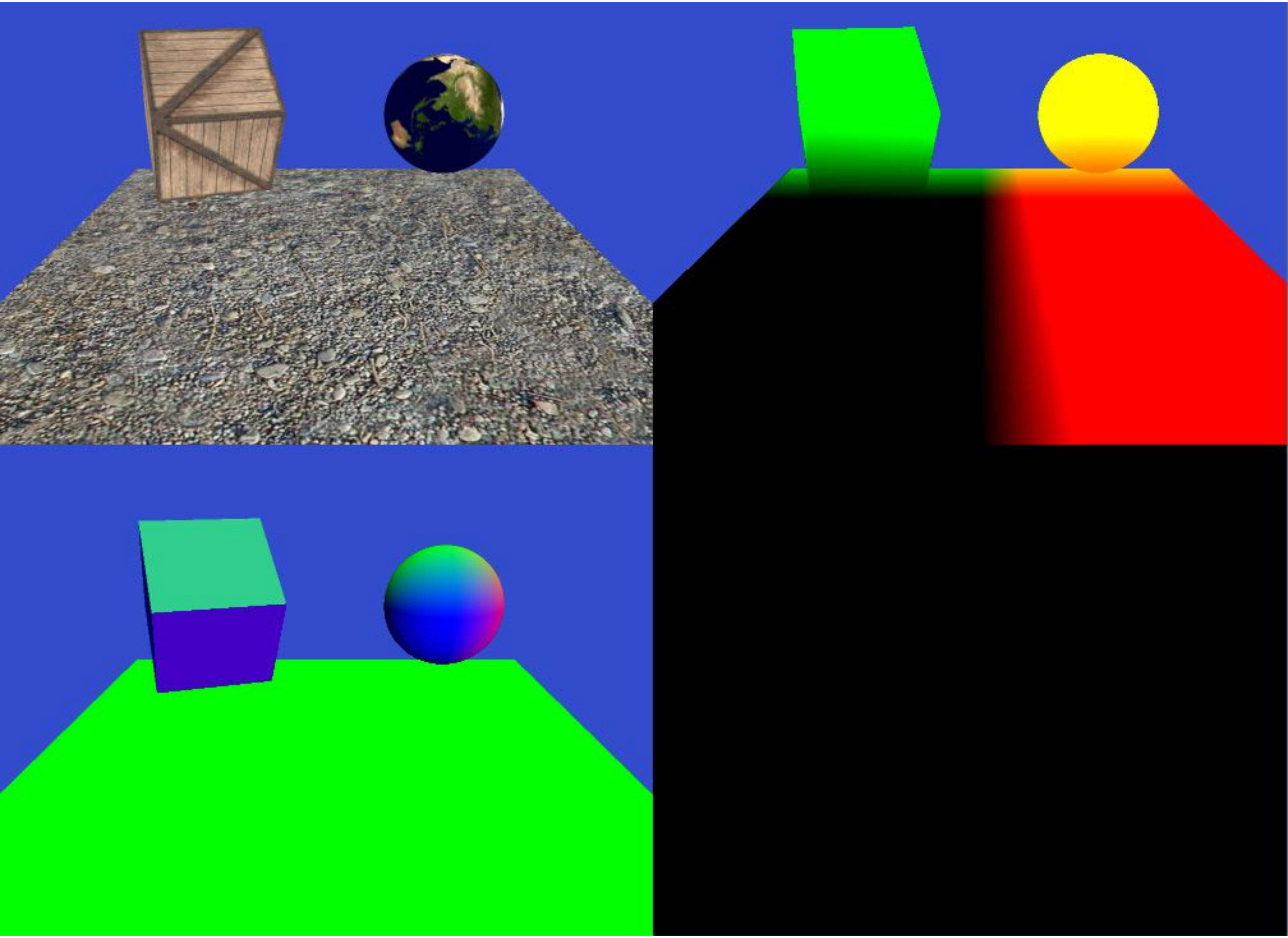
Pass 1: the geometry pass

Renders different components of objects in scene to a framebuffer with several color textures - this combined buffer is called the Geometry buffer or **G-buffer**.

The classic G-buffer contains one texture each for:

- vertex positions (world space)
- normals (model space)
- diffuse color

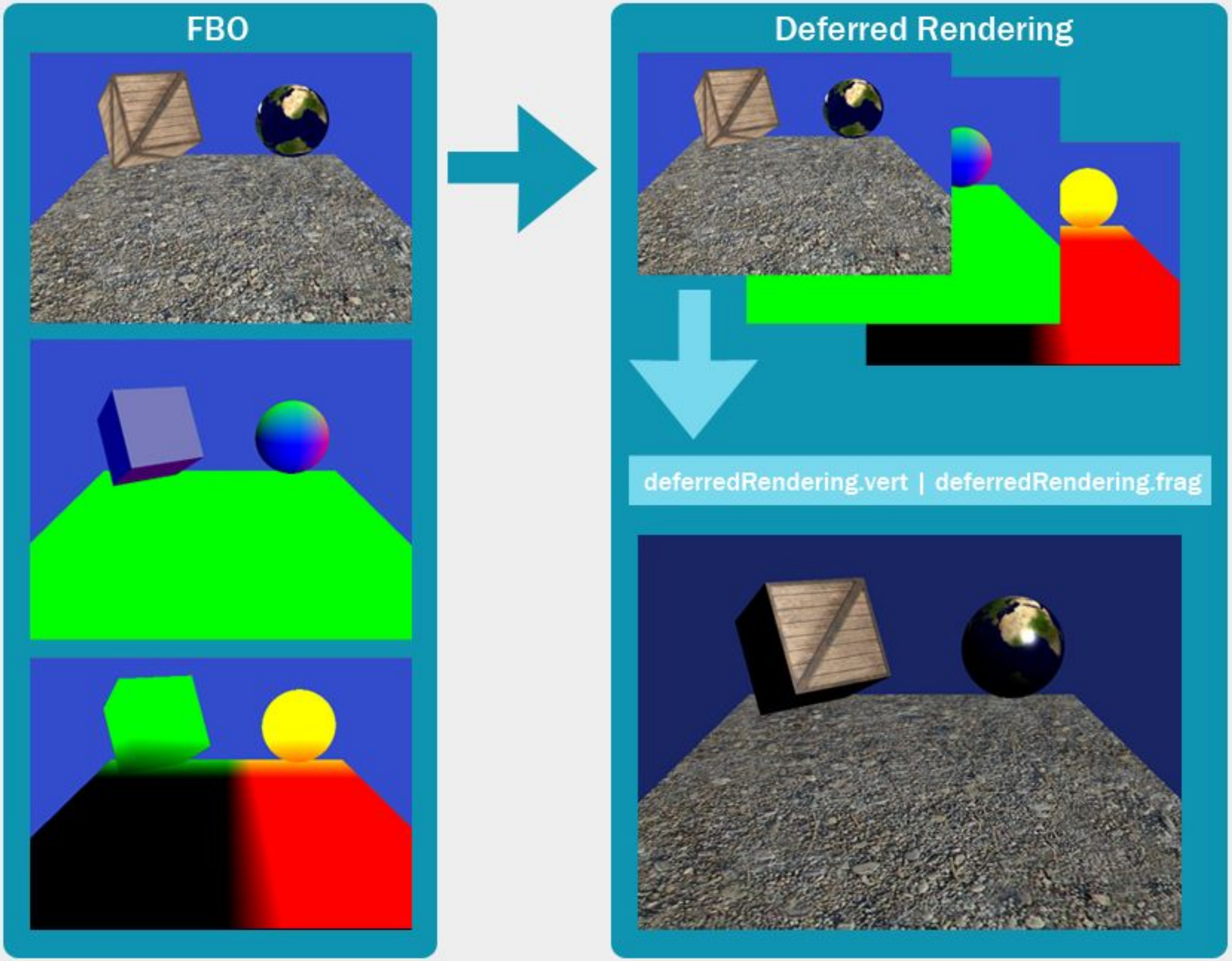
There is NO lighting in the G-buffer!



Pass 2: Lighting pass

Use textures as input to **quad painted in screen space**

Use information in textures, and light information, to calculate illumination of each fragment


FBO
Deferred Rendering
deferredRendering.vert | deferredRendering.frag

Why deferred?

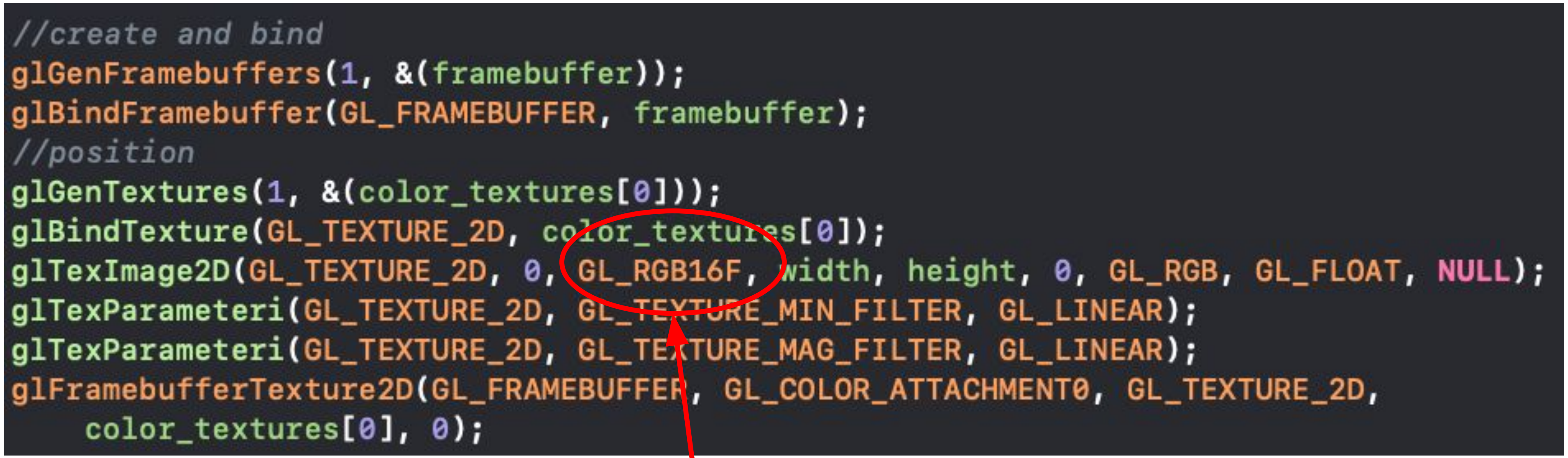
We light every fragment precisely **once**. So no wasted light calculations.

Can also add further optimisations that allow us to render many many more lights

BUT, uses more memory than forward shading, and doesn't support blending (used for transparent objects)

Creating a Gbuffer

Familiar code to start:



```
//create and bind
glGenFramebuffers(1, &(framebuffer));
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
//position
glGenTextures(1, &(color_textures[0]));
glBindTexture(GL_TEXTURE_2D, color_textures[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, width, height, 0, GL_RGB, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
    color_textures[0], 0);
```

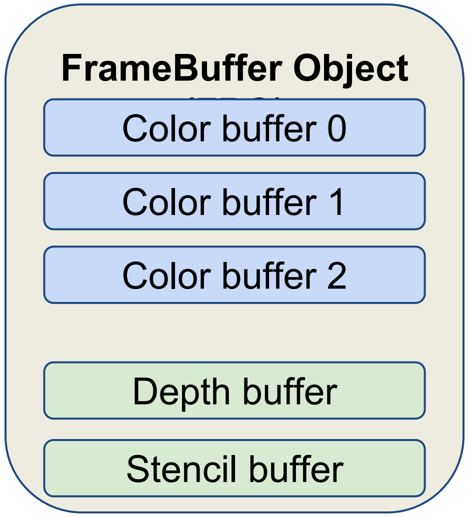
Note format of texture - 16-bit float per channel

Creating a Gbuffer (2)

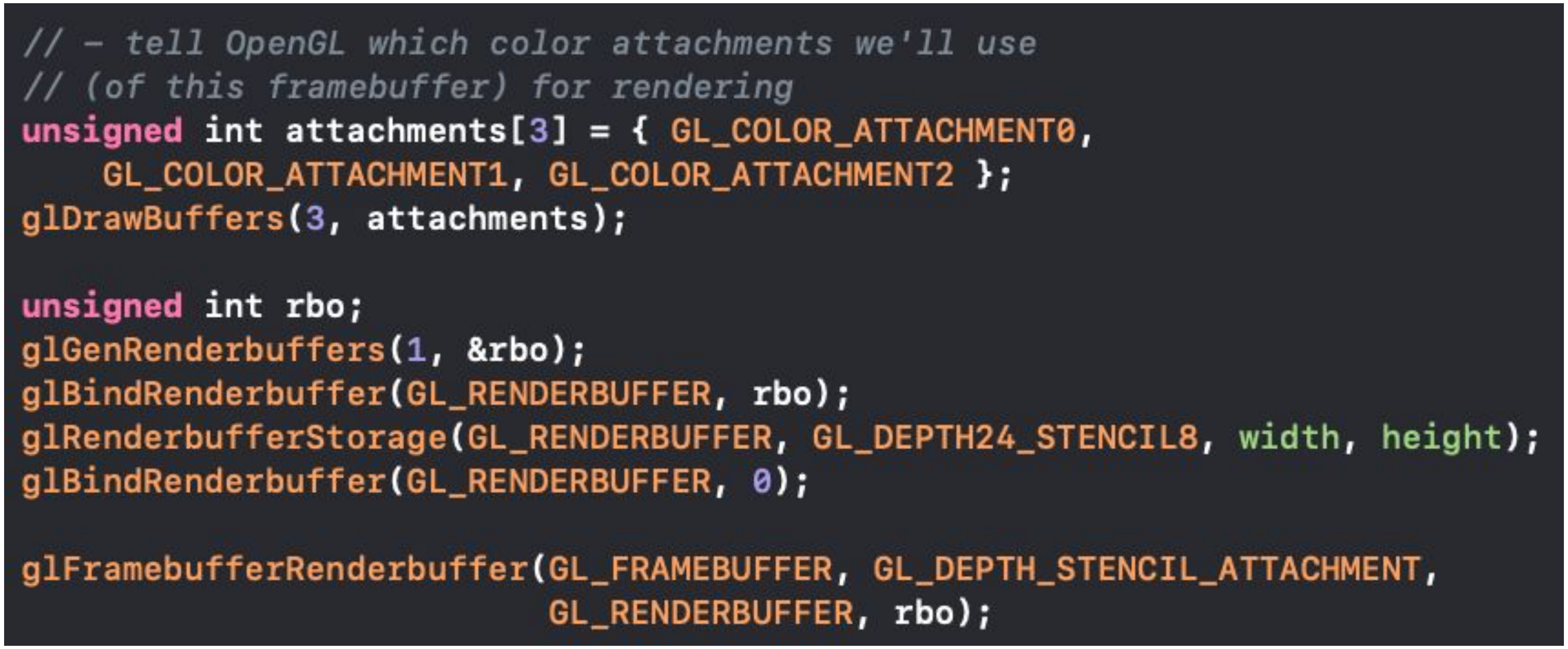
```
//normal
glGenTextures(1, &(color_textures[1]));
glBindTexture(GL_TEXTURE_2D, color_textures[1]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB16F, width, height, 0, GL_RGB, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D,
    color_textures[1], 0);

//diffuse + specular (in A channel)
glGenTextures(1, &(color_textures[2]));
glBindTexture(GL_TEXTURE_2D, color_textures[2]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D,
    color_textures[2], 0);
```

Create two more textures. 3rd texture stores diffuse color in RGB and specular in A (all GL_UNSIGNED_BYTE)

Creating a Gbuffer(3)

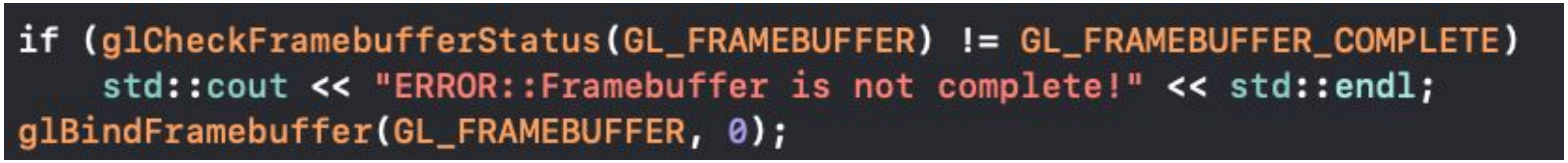


Tell OpenGL which attachments we’re using, and create renderbuffer object



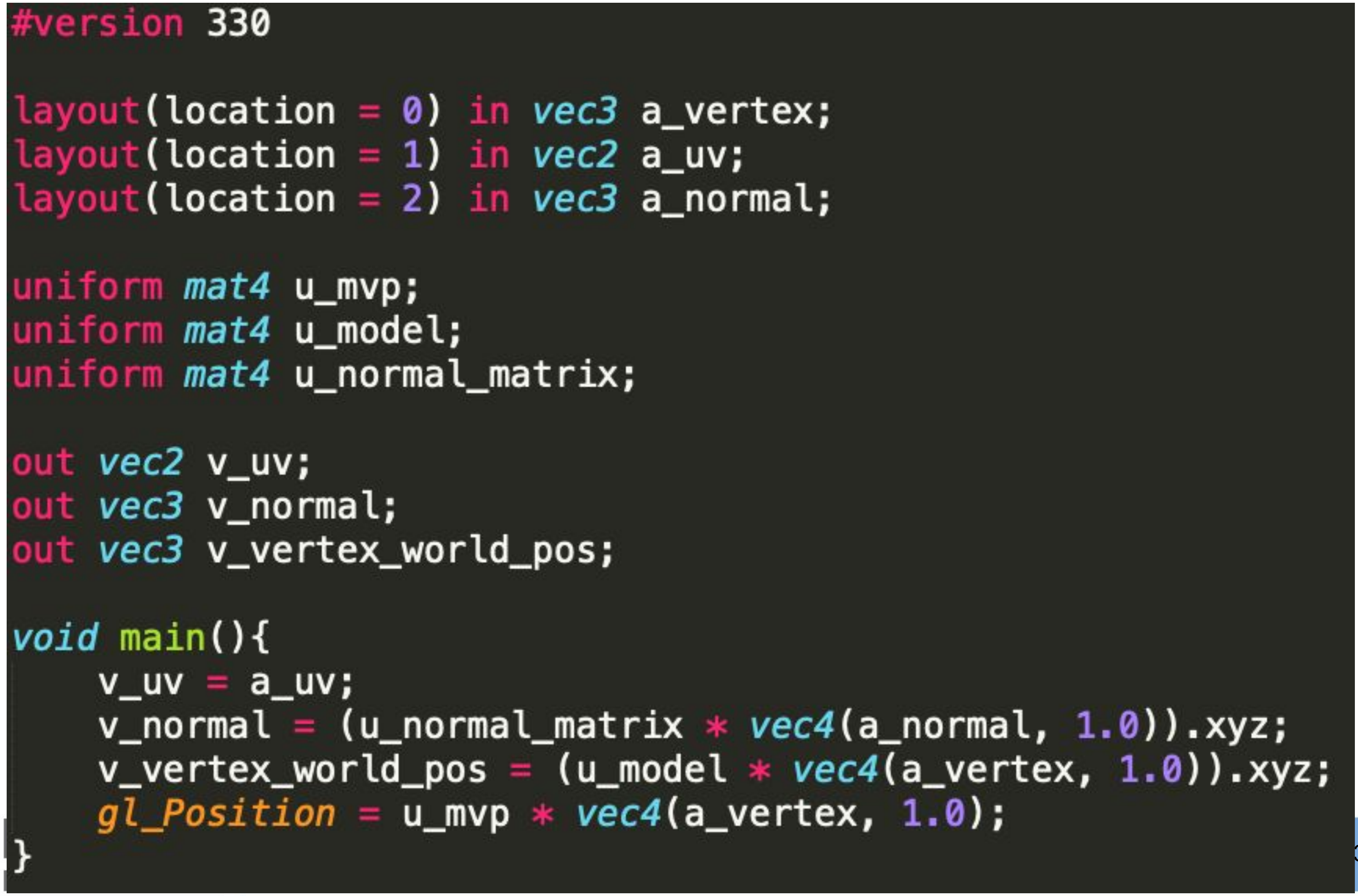
```
// - tell OpenGL which color attachments we'll use
// (of this framebuffer) for rendering
unsigned int attachments[3] = { GL_COLOR_ATTACHMENT0,
    GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT2 };
glDrawBuffers(3, attachments);

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);
```

Safety First, Safety Second!



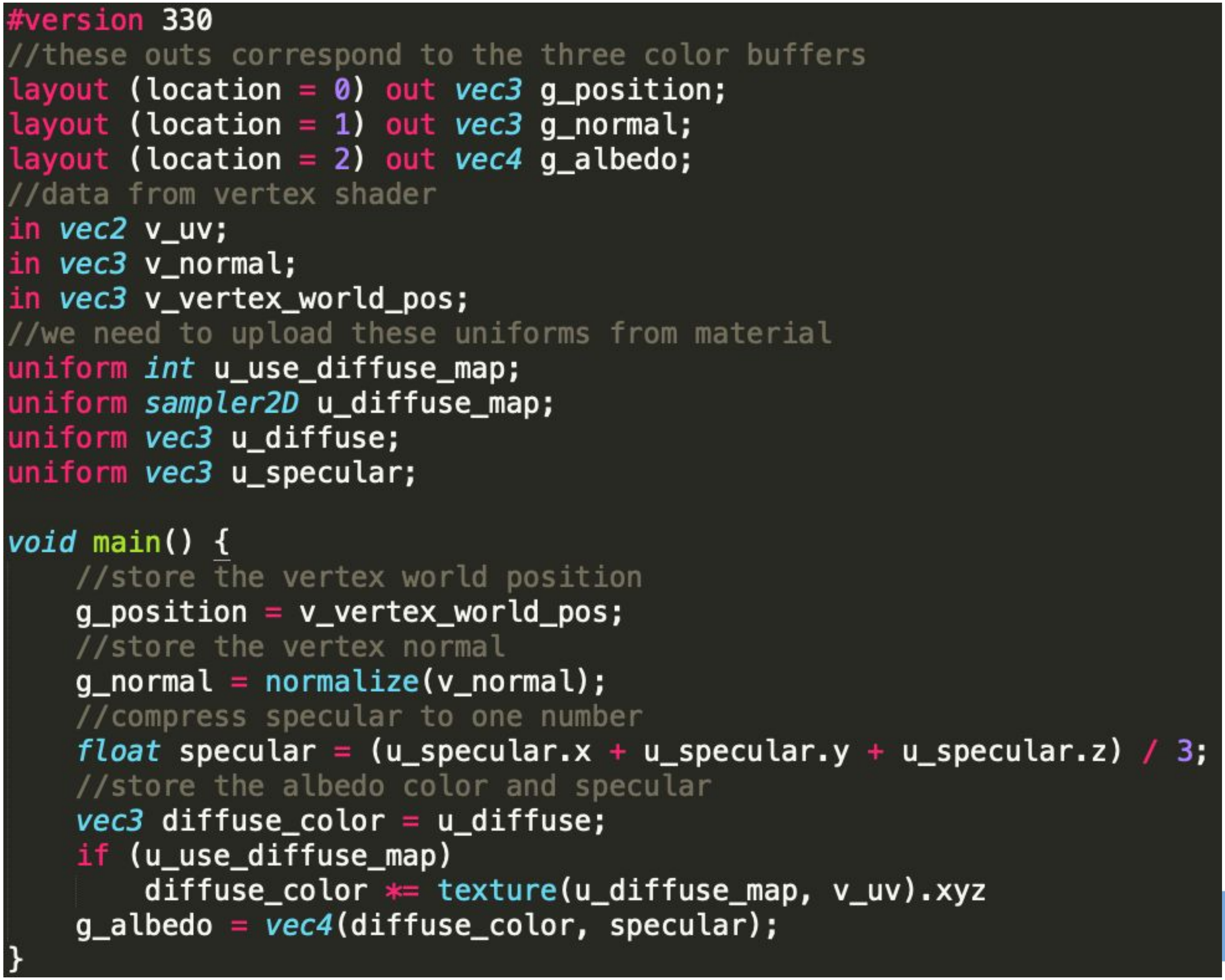
```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
    std::cout << "ERROR::Framebuffer is not complete!" << std::endl;
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

The Gbuffer shader: vertex is standard



```
#version 330

layout(location = 0) in vec3 a_vertex;
layout(location = 1) in vec2 a_uv;
layout(location = 2) in vec3 a_normal;

uniform mat4 u_mvp;
uniform mat4 u_model;
uniform mat4 u_normal_matrix;

out vec2 v_uv;
out vec3 v_normal;
out vec3 v_vertex_world_pos;

void main(){
    v_uv = a_uv;
    v_normal = (u_normal_matrix * vec4(a_normal, 1.0)).xyz;
    v_vertex_world_pos = (u_model * vec4(a_vertex, 1.0)).xyz;
    gl_Position = u_mvp * vec4(a_vertex, 1.0);
}
```

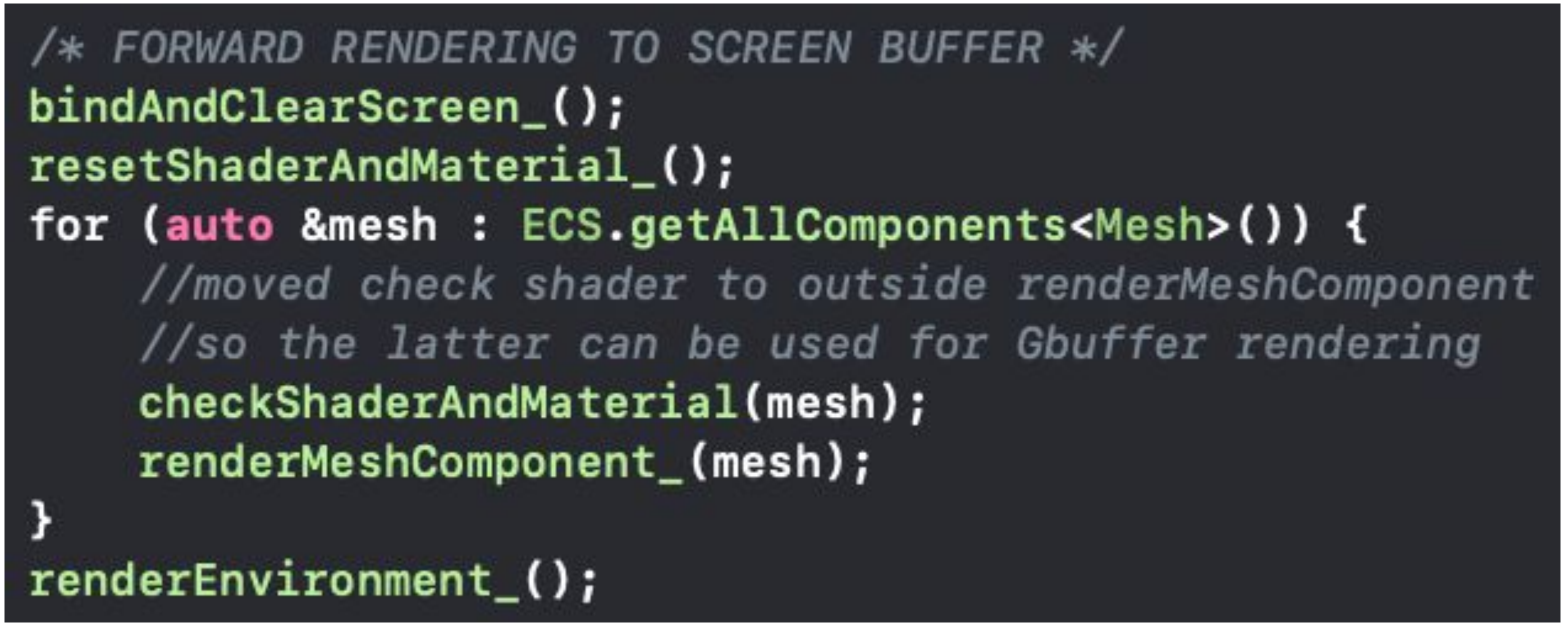
Gbuffer fragment shader is new!



```
#version 330
//these outs correspond to the three color buffers
layout (location = 0) out vec3 g_position;
layout (location = 1) out vec3 g_normal;
layout (location = 2) out vec4 g_albedo;
//data from vertex shader
in vec2 v_uv;
in vec3 v_normal;
in vec3 v_vertex_world_pos;
//we need to upload these uniforms from material
uniform int u_use_diffuse_map;
uniform sampler2D u_diffuse_map;
uniform vec3 u_diffuse;
uniform vec3 u_specular;

void main() {
    //store the vertex world position
    g_position = v_vertex_world_pos;
    //store the vertex normal
    g_normal = normalize(v_normal);
    //compress specular to one number
    float specular = (u_specular.x + u_specular.y + u_specular.z) / 3;
    //store the albedo color and specular
    vec3 diffuse_color = u_diffuse;
    if (u_use_diffuse_map)
        diffuse_color *= texture(u_diffuse_map, v_uv).xyz
    g_albedo = vec4(diffuse_color, specular);
}
```

Screen Rendering code changes

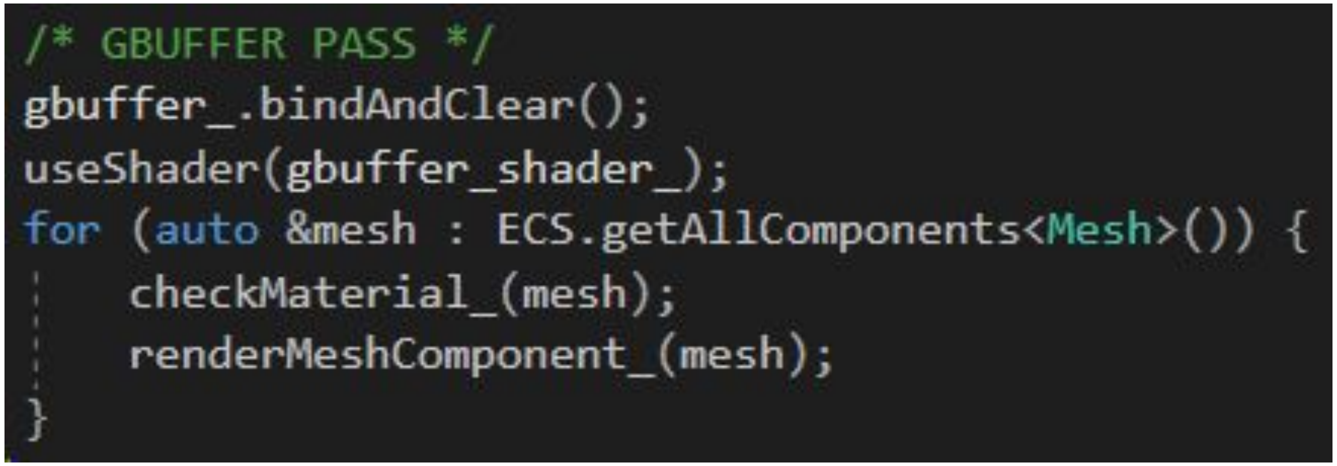
Removed checkShaderandMaterial from out of renderMeshComponent. This allows us to use renderMeshComponent to render Gbuffer too



```
/* FORWARD RENDERING TO SCREEN BUFFER */
bindAndClearScreen_();
resetShaderAndMaterial_();
for (auto &mesh : ECS.getAllComponents<Mesh>()) {
    //moved check shader to outside renderMeshComponent
    //so the latter can be used for Gbuffer rendering
    checkShaderAndMaterial(mesh);
    renderMeshComponent_(mesh);
}
renderEnvironment_();
```

Gbuffer rendering

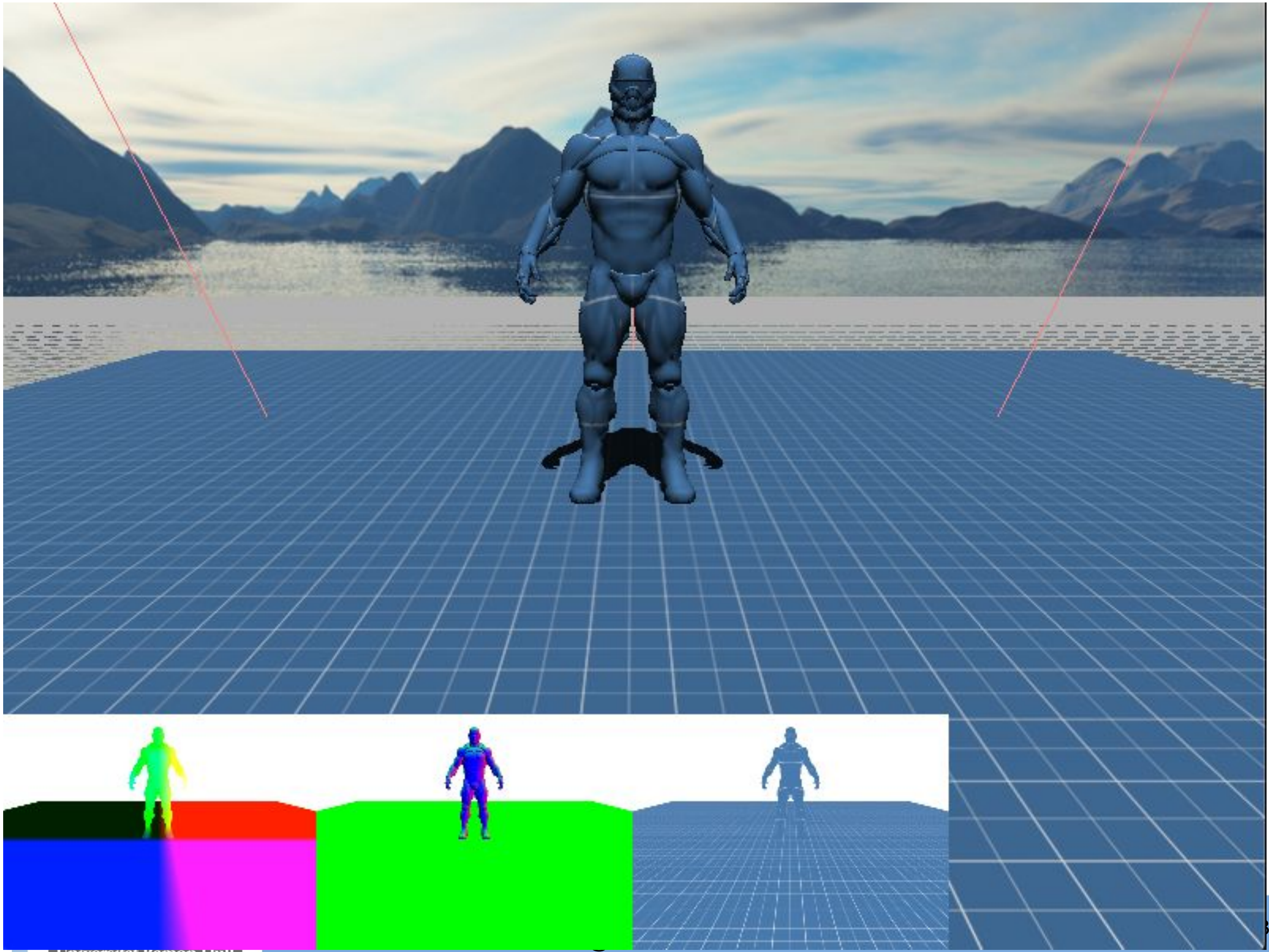
Just check material! Not shader



```
/* GBUFFER PASS */
gbuffer_.bindAndClear();
useShader(gbuffer_shader_);
for (auto &mesh : ECS.getAllComponents<Mesh>()) {
    checkMaterial_(mesh);
    renderMeshComponent_(mesh);
}
```

Task: Render G-buffers

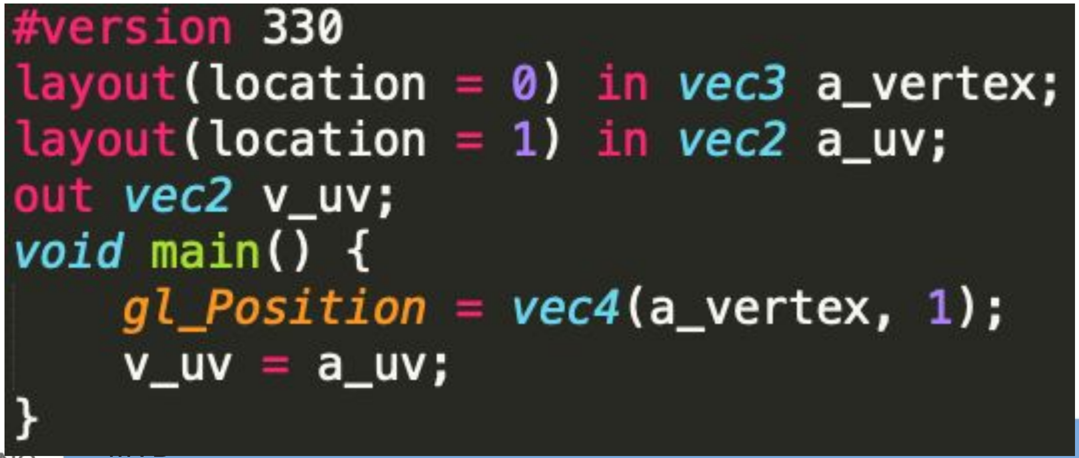
1. Look at Framebuffer::initGbuffer, and gbuffer-related vars in GraphicsSystem.h
2. Create shader
3. In GraphicsSystem, load gbuffer objects and shader object
4. Add pass to render to Gbuffer
    a. Can use same renderMeshComponent() and setMaterialUniforms() etc.
5. Draw Gbuffer color buffers to bottom of screen



Shading the Gbuffer

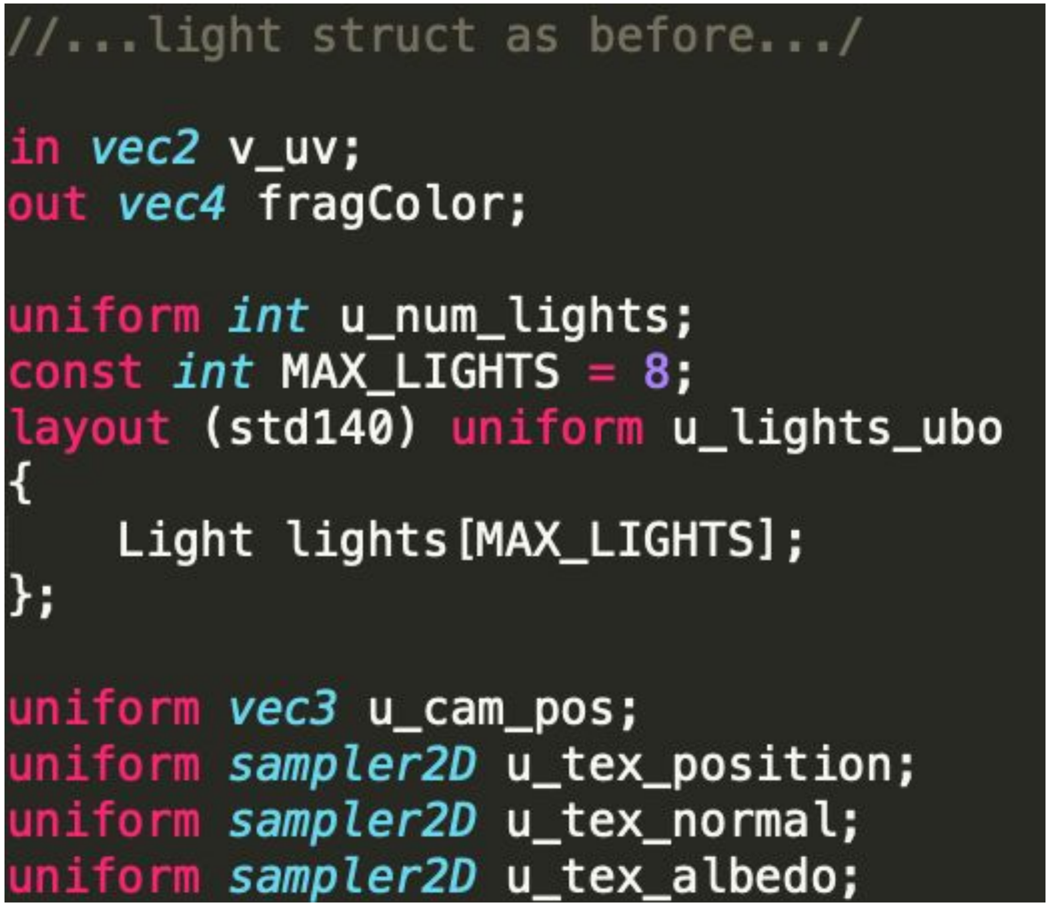
We now draw the different Gbuffer textures onto a screen-space quad, blending them according to lighting calculations

Vertex shader is a simple screen space shader

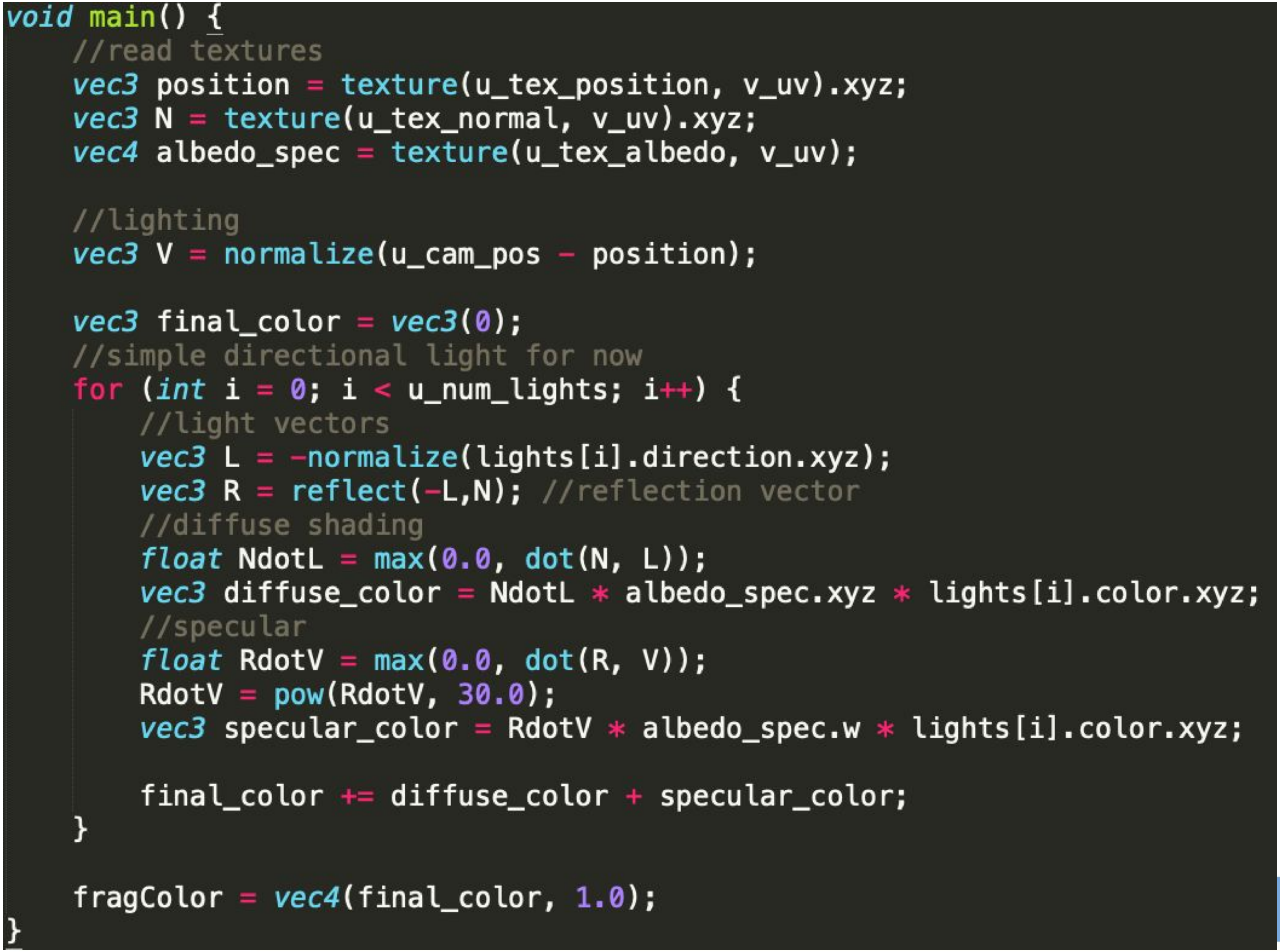


```
#version 330
layout(location = 0) in vec3 a_vertex;
layout(location = 1) in vec2 a_uv;
out vec2 v_uv;
void main() {
    gl_Position = vec4(a_vertex, 1);
    v_uv = a_uv;
}
```

Fragment shader with simple light



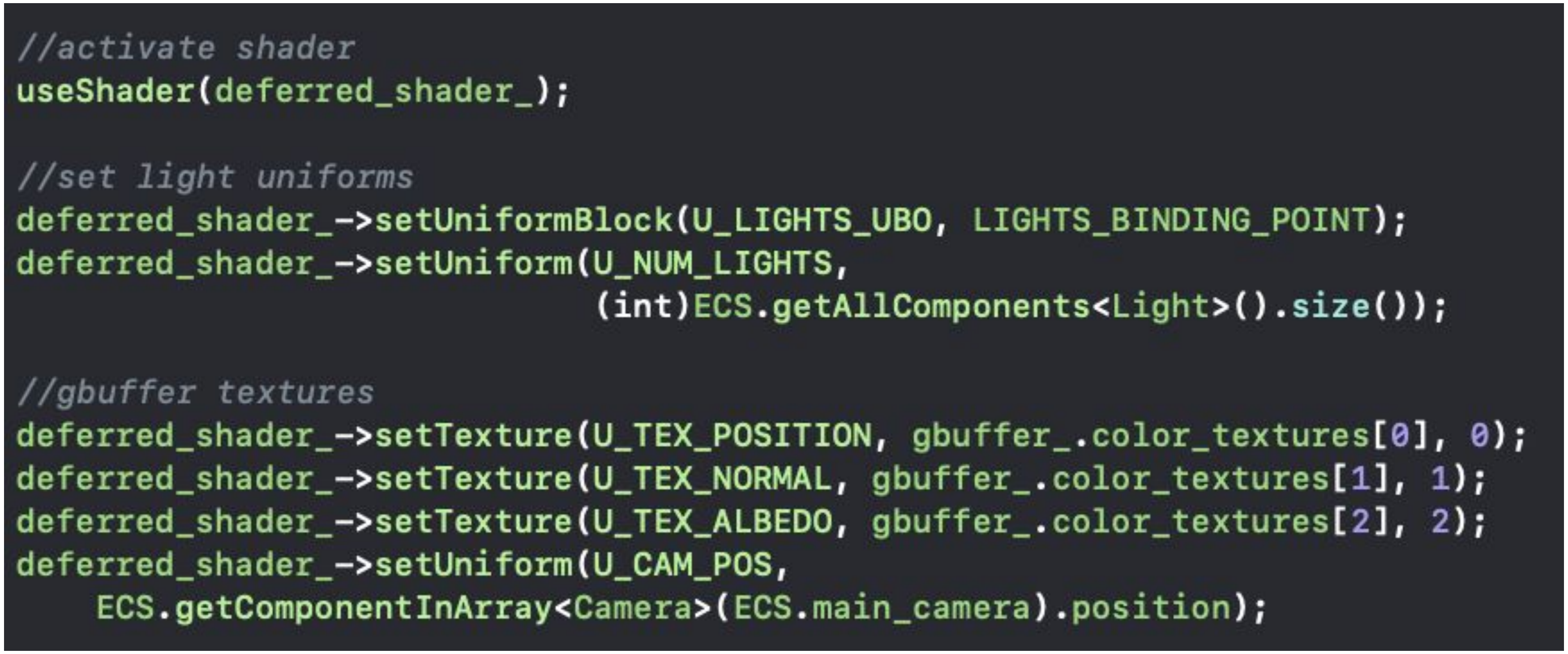
```
//...light struct as before.../

in vec2 v_uv;
out vec4 fragColor;

uniform int u_num_lights;
const int MAX_LIGHTS = 8;
layout (std140) uniform u_lights_ubo
{
    Light lights[MAX_LIGHTS];
};

uniform vec3 u_cam_pos;
uniform sampler2D u_tex_position;
uniform sampler2D u_tex_normal;
uniform sampler2D u_tex_albedo;
```



```glsl
void main() {
    //read textures
    vec3 position = texture(u_tex_position, v_uv).xyz;
    vec3 N = texture(u_tex_normal, v_uv).xyz;
    vec4 albedo_spec = texture(u_tex_albedo, v_uv);

    //lighting
    vec3 V = normalize(u_cam_pos - position);

    vec3 final_color = vec3(0);
    //simple directional light for now
    for (int i = 0; i < u_num_lights; i++) {
        //light vectors
        vec3 L = -normalize(lights[i].direction.xyz);
        vec3 R = reflect(-L,N); //reflection vector
        //diffuse shading
        float NdotL = max(0.0, dot(N, L));
        vec3 diffuse_color = NdotL * albedo_spec.xyz * lights[i].color.xyz;
        //specular
        float RdotV = max(0.0, dot(R, V));
        RdotV = pow(RdotV, 30.0);
        vec3 specular_color = RdotV * albedo_spec.w * lights[i].color.xyz;

        final_color += diffuse_color + specular_color;
    }

    fragColor = vec4(final_color, 1.0);
}
```

renderGbuffer_()

Activate shader and set uniforms



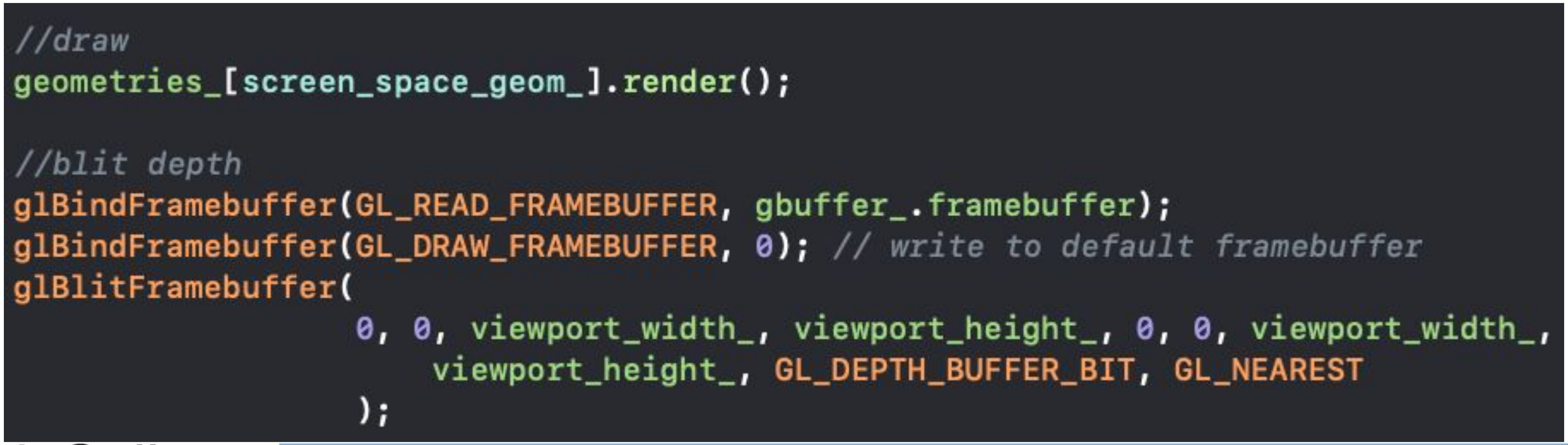
```
//activate shader
useShader(deferred_shader_);

//set light uniforms
deferred_shader_->setUniformBlock(U_LIGHTS_UBO, LIGHTS_BINDING_POINT);
deferred_shader_->setUniform(U_NUM_LIGHTS,
                             (int)ECS.getAllComponents<Light>().size());

//gbuffer textures
deferred_shader_->setTexture(U_TEX_POSITION, gbuffer_.color_textures[0], 0);
deferred_shader_->setTexture(U_TEX_NORMAL, gbuffer_.color_textures[1], 1);
deferred_shader_->setTexture(U_TEX_ALBEDO, gbuffer_.color_textures[2], 2);
deferred_shader_->setUniform(U_CAM_POS,
    ECS.getComponentInArray<Camera>(ECS.main_camera).position);
```

Blitting the depth values

1st draw geometry

‘Blit’ means copy from one buffer another.

We want to copy the **depth values** from the gbuffer renderbuffer to the screen’s renderbuffer



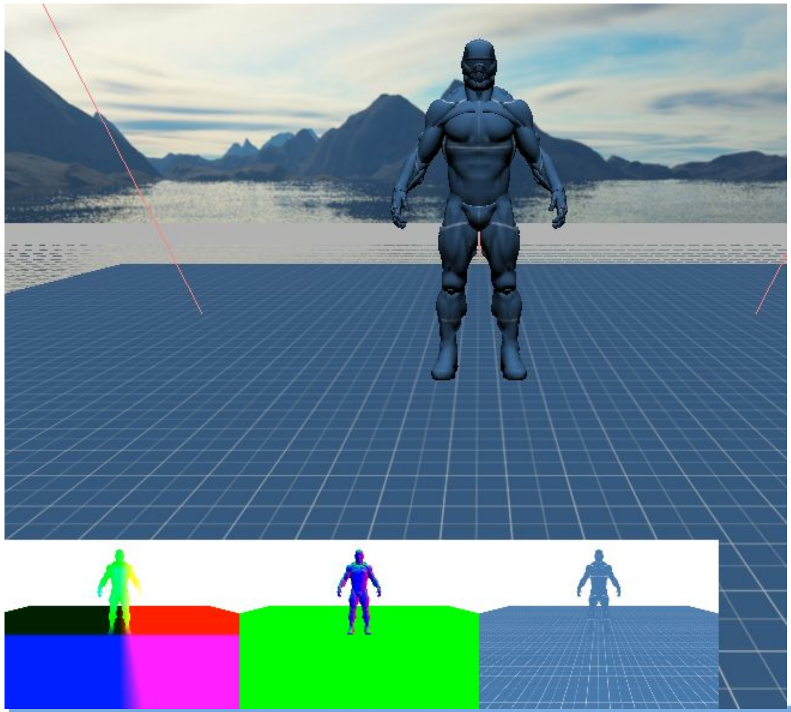
```
//draw
geometries_[screen_space_geom_].render();

//blit depth
glBindFramebuffer(GL_READ_FRAMEBUFFER, gbuffer_.framebuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); // write to default framebuffer
glBlitFramebuffer(
                  0, 0, viewport_width_, viewport_height_, 0, 0, viewport_width_,
                      viewport_height_, GL_DEPTH_BUFFER_BIT, GL_NEAREST
                  );
```

What to draw and when?

Draw Gbuffer first

Blit Gbuffer depth to screen buffer

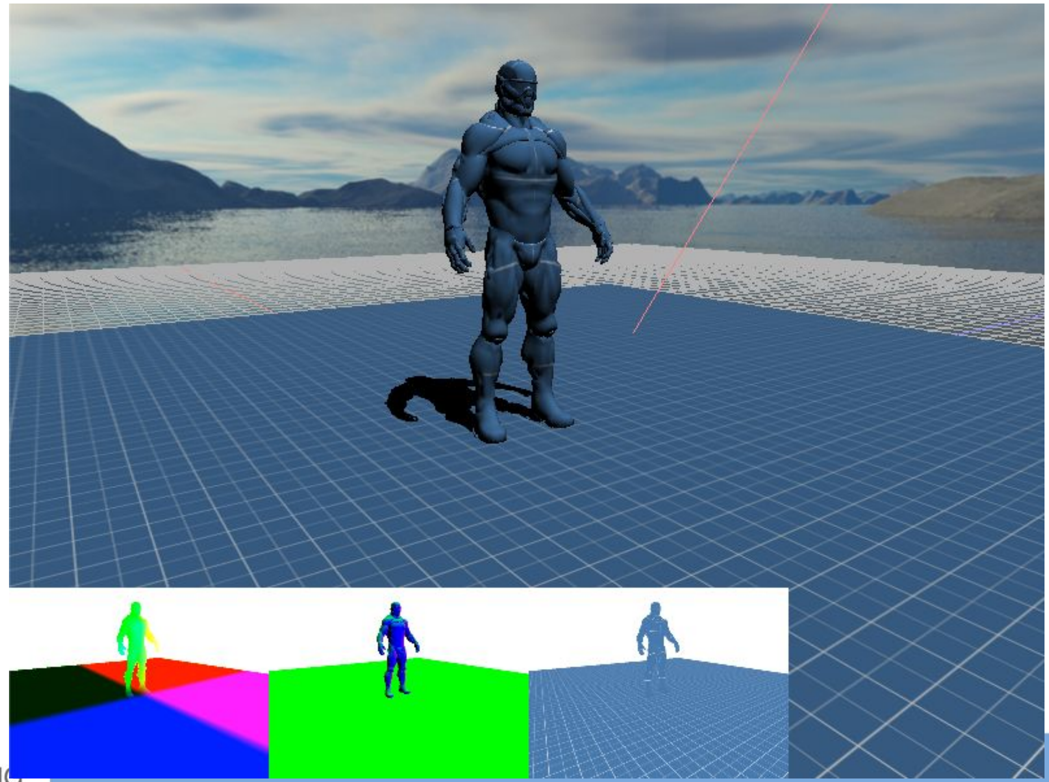
now can draw anything we want in forward pass



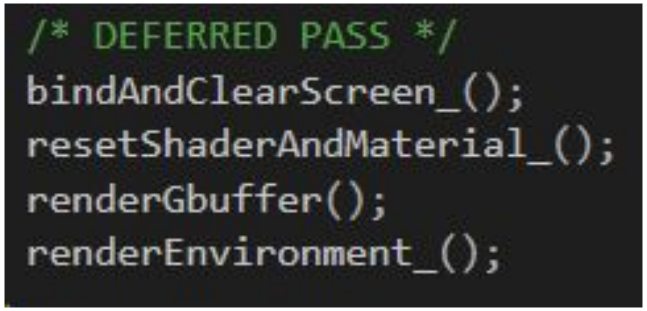
Shadows?

Same deal.

Deferred pass use vertex world position



Task



```
/* DEFERRED PASS */
bindAndClearScreen_();
resetShaderAndMaterial_();
renderGbuffer();
renderEnvironment_();
```

1. Write, declare and compile the deferred shader
2. Write a renderGBuffer function
3. Draw Gbuffer
4. Modify shader to include shadows and different light types
   a. need to send shadowmaps to deferred shader