MVD: Advanced Graphics 1

15 - Shadows

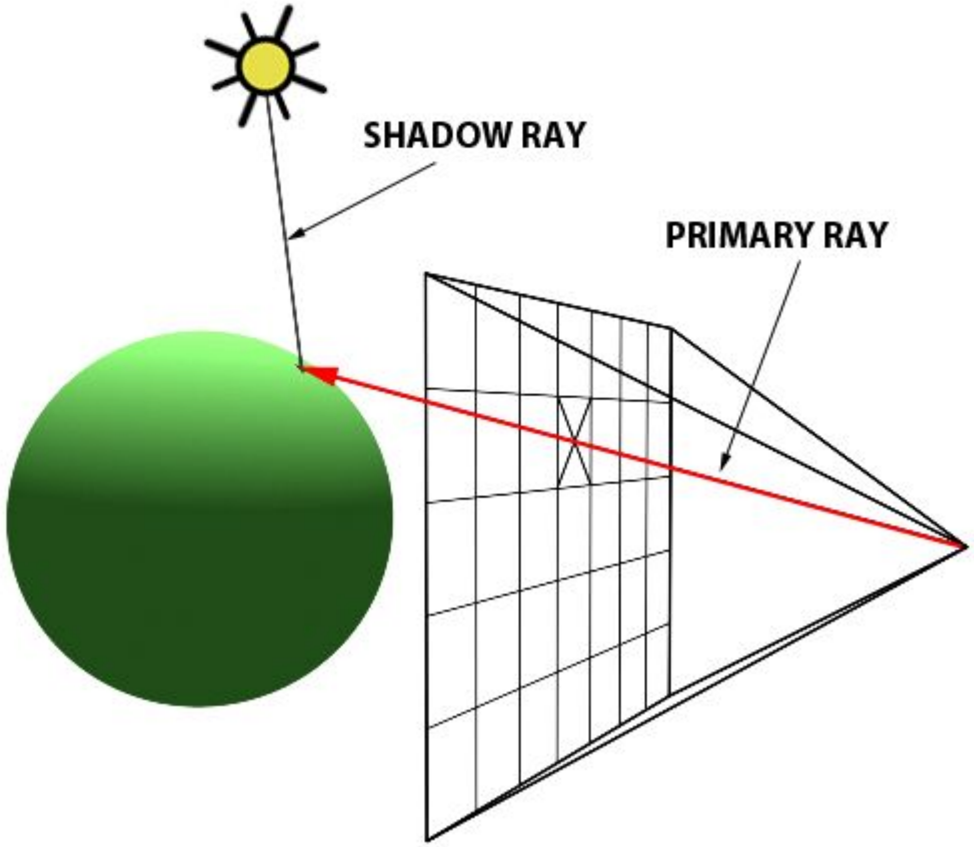
alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Drawing Shadows

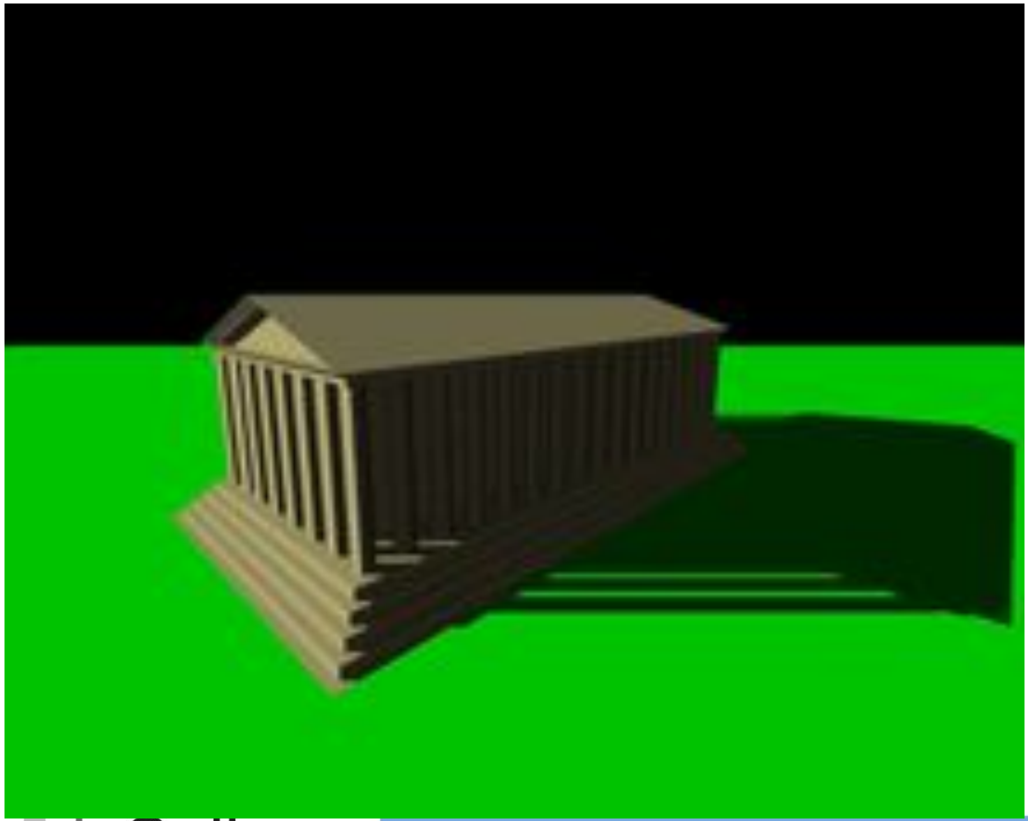
In ray tracing, shadow are pretty easy.

But calculating the ray from every fragment to every light is extremely expensive

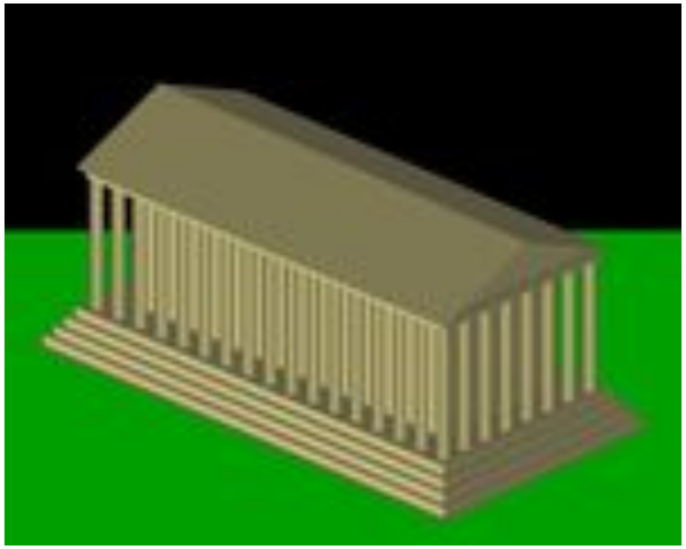


Shadow mapping (1978)

“Whatever the light sees, gets lit”



From light’s point of view

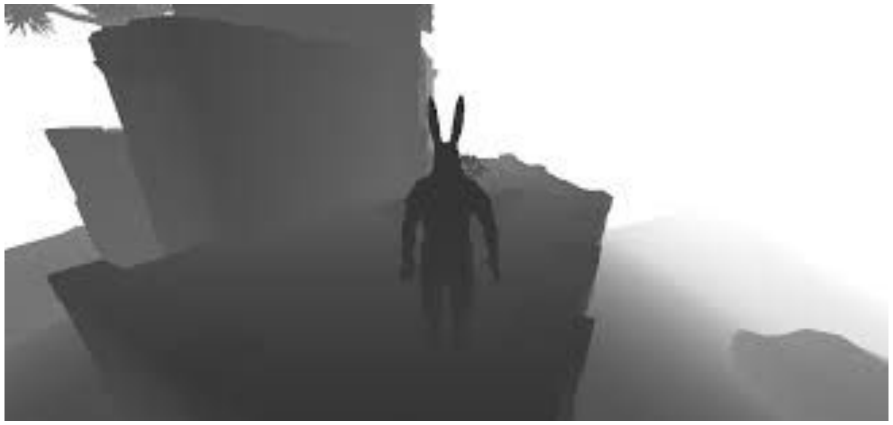


Light as a camera

Shadow mapping key trick to treat a light as if it were a camera, and render the scene from the light's perspective.

BUT, instead of storing the colour buffer, we only store the depth buffer. Why?

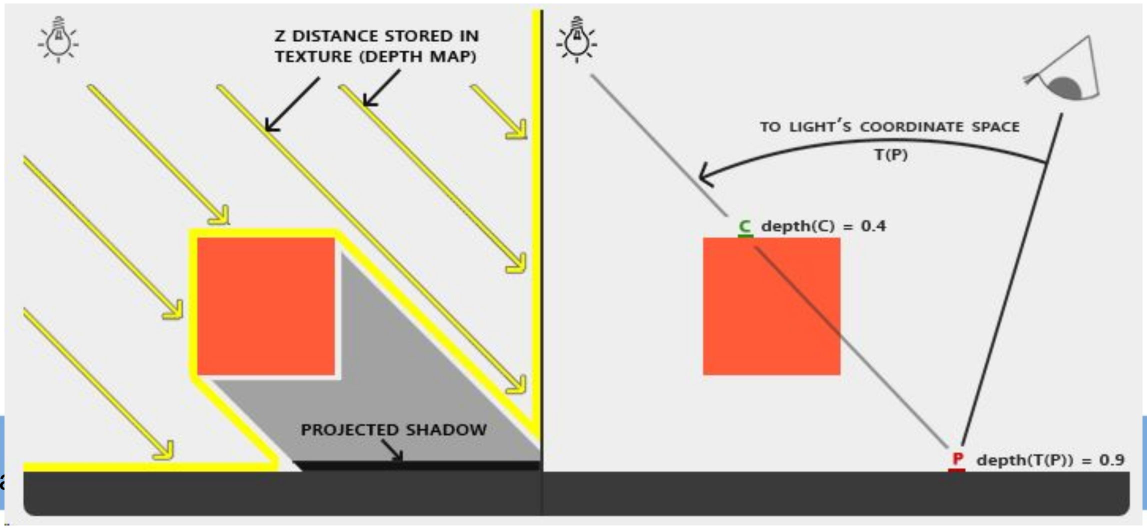
Becuase the depth buffer stores the **distance to light**



The light depth buffer

When we render the scene from the point of view of camera, we measure the distance to the light of each fragment.

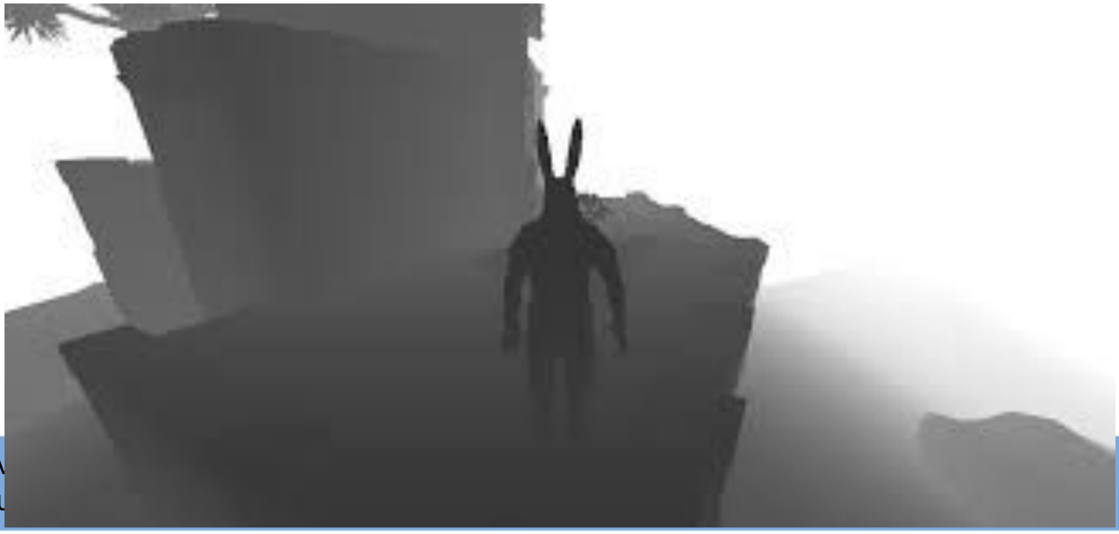
Then we compare that distance with the distance stored in shadow map. If shadow map distance is smaller, we draw the fragment in shadow.


Z DISTANCE STORED IN TEXTURE (DEPTH MAP)
PROJECTED SHADOW
TO LIGHT'S COORDINATE SPACE
T(P)
C depth(C) = 0.4
P depth(T(P)) = 0.9

Shadow mapping steps

1st render pass:

- using the light's view_projection matrix
- draw to a renderbuffer
- write depth to a texture which only stores depth = **shadow map**



Shadow mapping steps

2nd render pass:

- use main camera view_projection
- draw to screen
- in shader:
  - calculate distance from fragment to light
  - use light's view_projection matrix to calculate fragment position in shadow map
  - if dist_fragment_light > dist_shadow_map
    - **draw SHADOW!**

Scene considerations

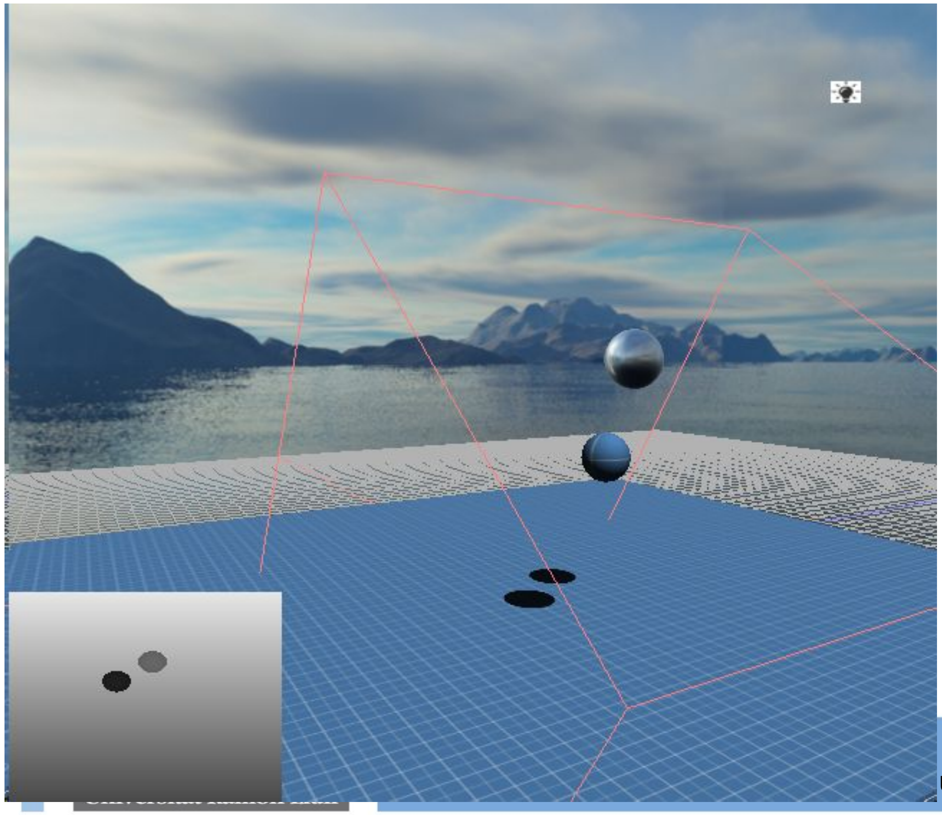
Light must be either directional or spot light (we'll speak later about point light

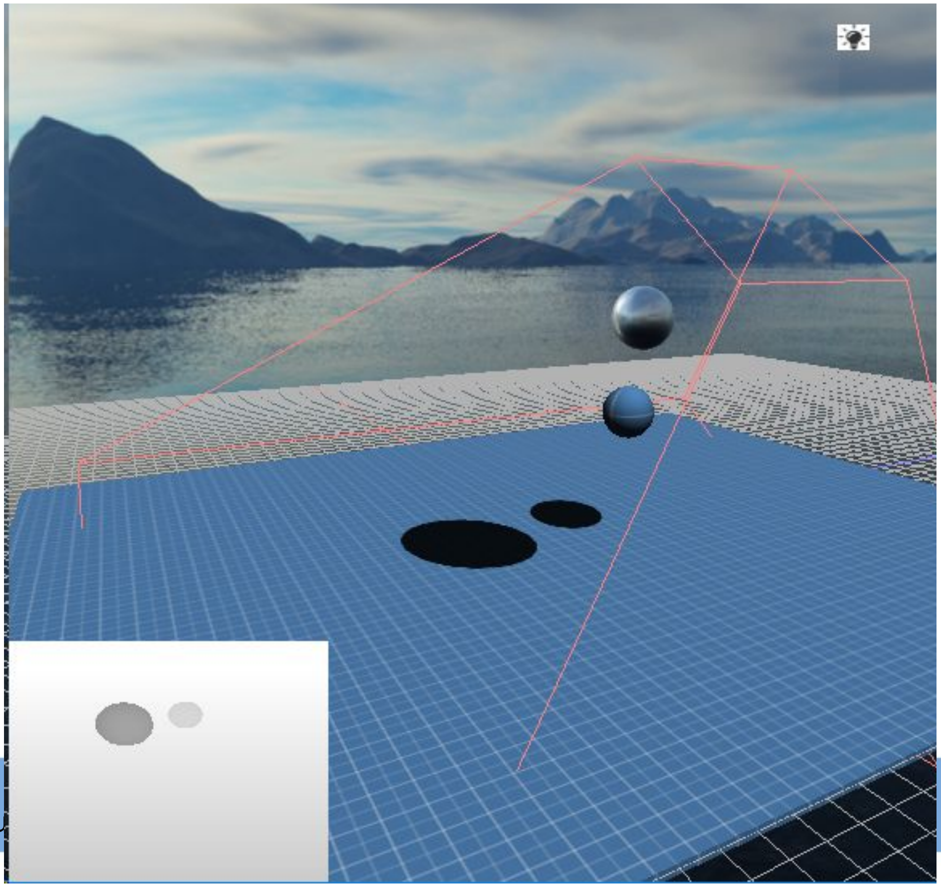
All object to be shadowed (and drawn) must be inside light frustum

Light can be perspective or orthographic

Perspective vs Orthographic

Orthographic better for directional lights, perspective for spot lights

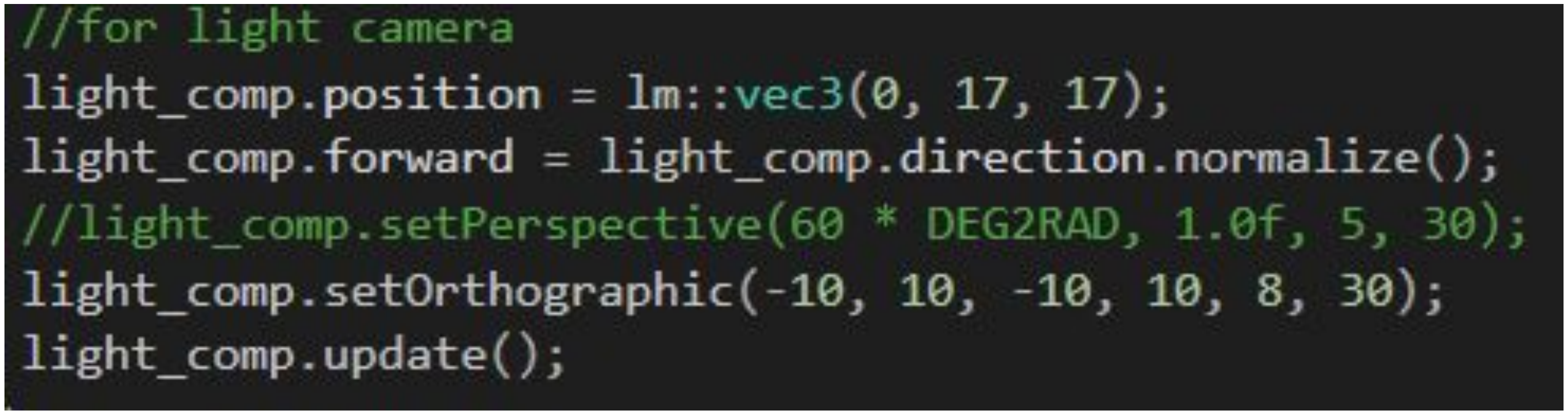




Creating a shadow map

Make light a camera. How? Simply get it to inherit camera

Add light properties in game.cpp

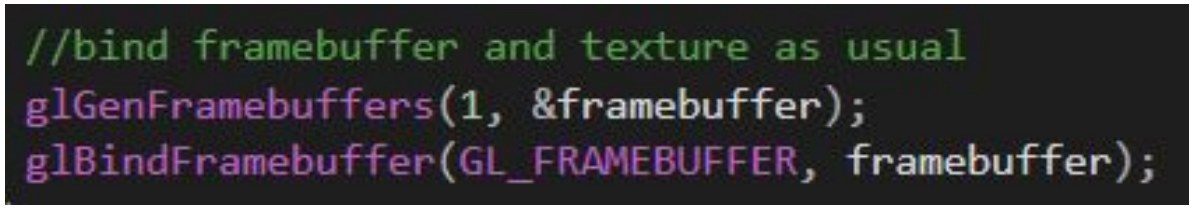


```
//for light camera
light_comp.position = lm::vec3(0, 17, 17);
light_comp.forward = light_comp.direction.normalize();
//light_comp.setPerspective(60 * DEG2RAD, 1.0f, 5, 30);
light_comp.setOrthographic(-10, 10, -10, 10, 8, 30);
light_comp.update();
```

Debug system should draw frustum for you

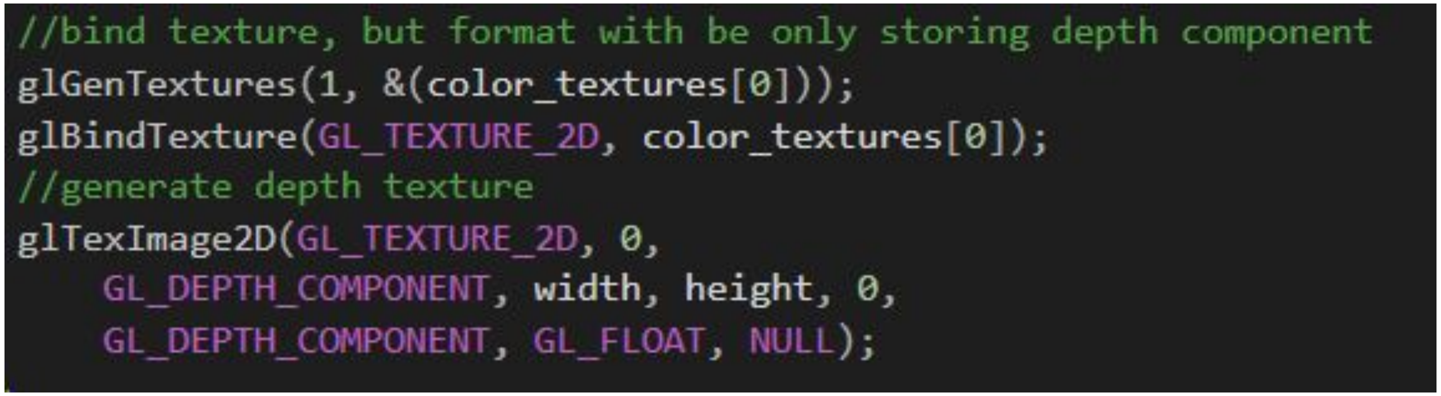
Create and init shadow buffer

Framebuffer::initDepth



```
//bind framebuffer and texture as usual
glGenFramebuffers(1, &framebuffer);
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
```

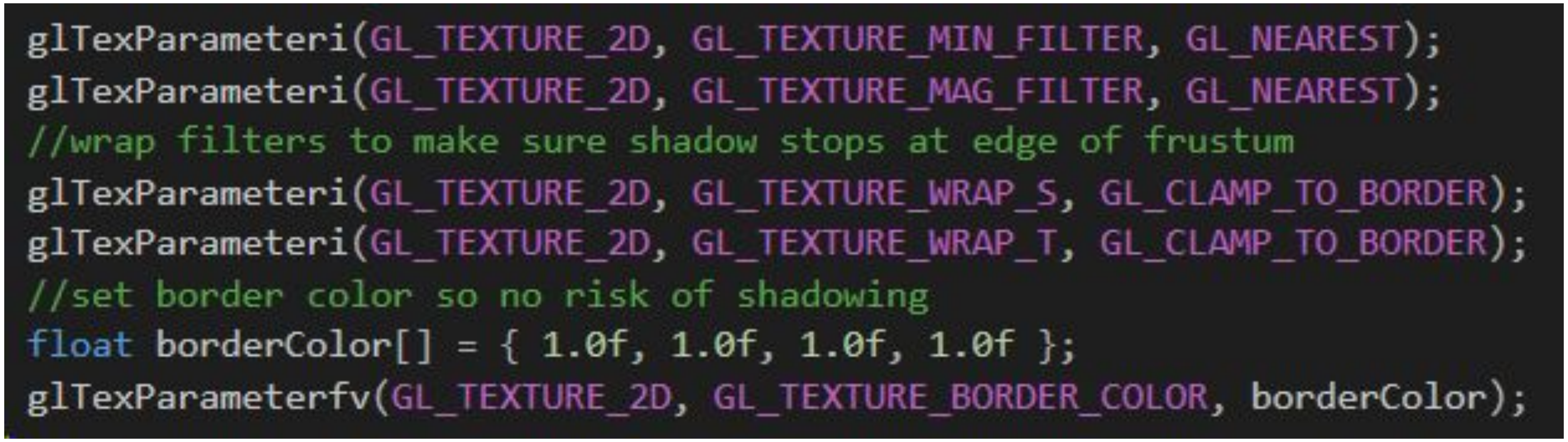
create framebuffer and texture as usual, but note that the format of the texture is not GL_RGB:



```
//bind texture, but format with be only storing depth component
glGenTextures(1, &(color_textures[0]));
glBindTexture(GL_TEXTURE_2D, color_textures[0]);
//generate depth texture
glTexImage2D(GL_TEXTURE_2D, 0,
    GL_DEPTH_COMPONENT, width, height, 0,
    GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

Create and init shadow buffer

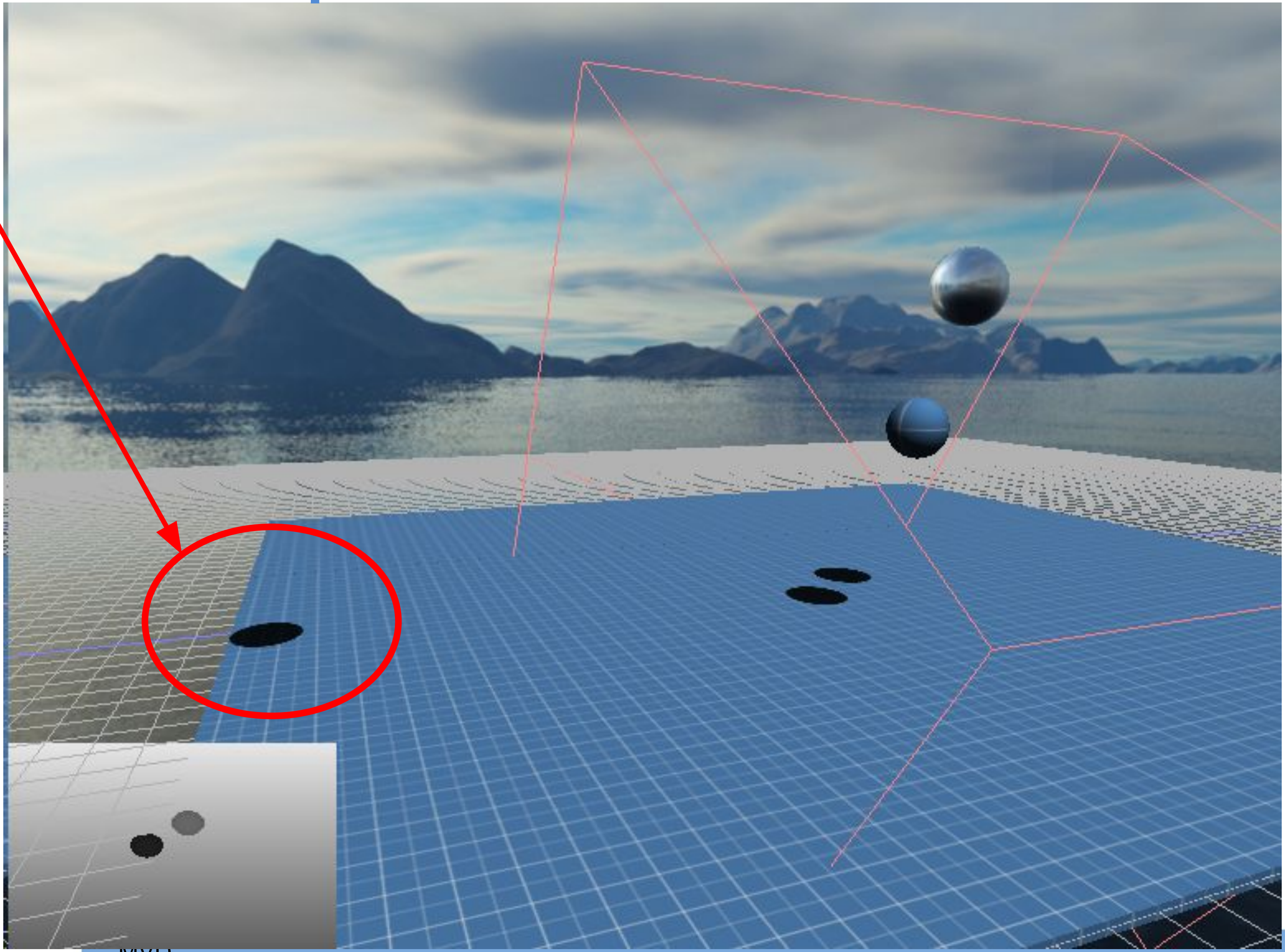
Also note that we have to set two additional filters, to clamp the texture to the border, and a ‘border color’ to stop fake shadows outside frustum



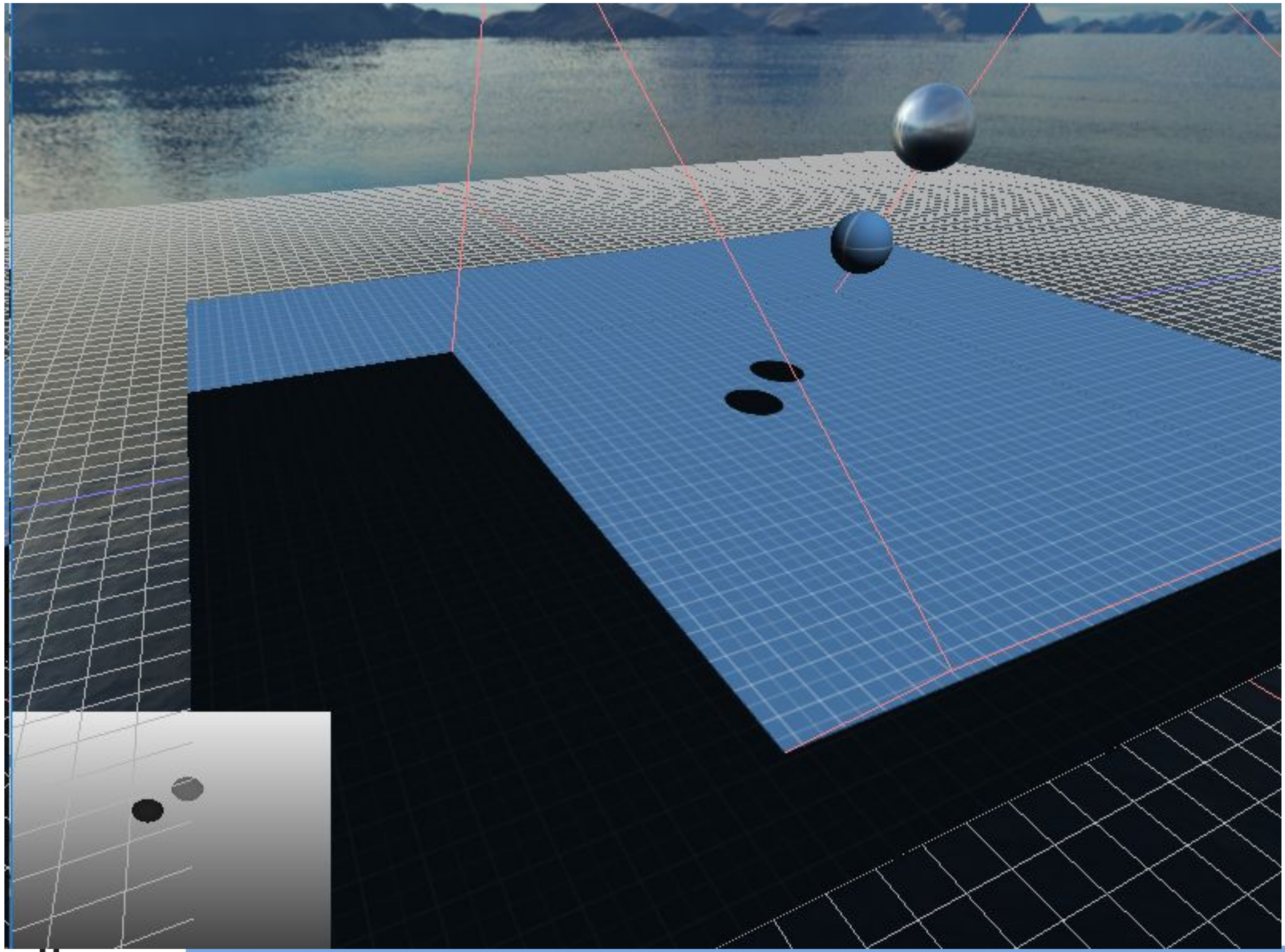
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
//wrap filters to make sure shadow stops at edge of frustum
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
//set border color so no risk of shadowing
float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

If we don't clamp to border...

Oooops!

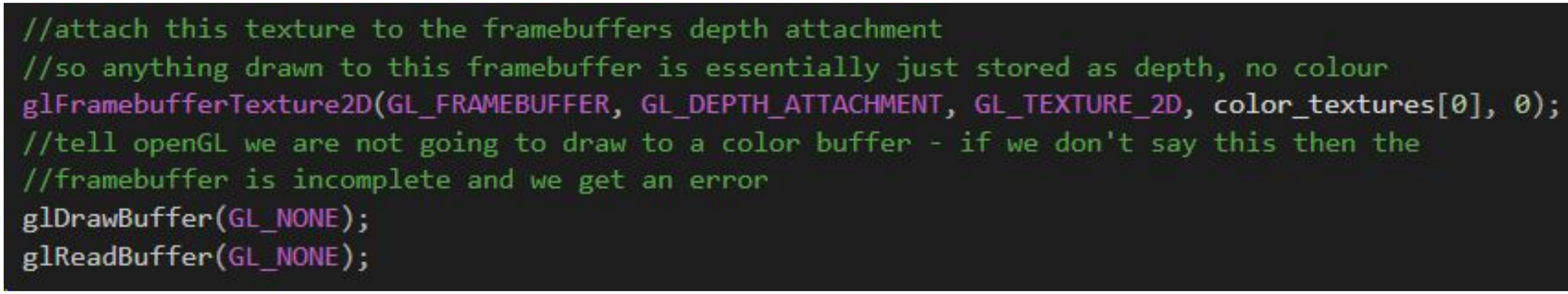


If we don't set border colour



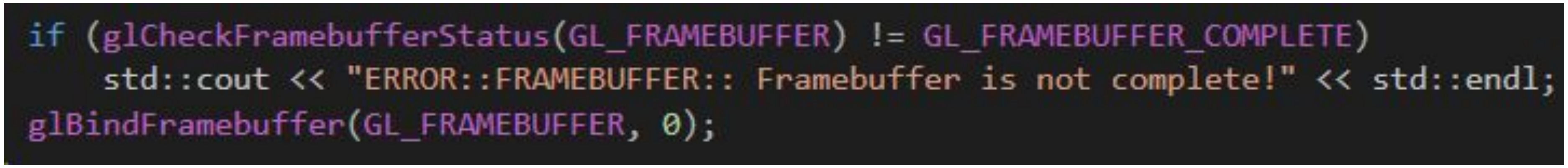
Complete framebuffer

No need to create renderbuffer. Just attach depth texture to the depth component, and tell openGL that this framebuffer has NO color (neither for reading nor writing)



```
//attach this texture to the framebuffers depth attachment
//so anything drawn to this framebuffer is essentially just stored as depth, no colour
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, color_textures[0], 0);
//tell openGL we are not going to draw to a color buffer - if we don't say this then the
//framebuffer is incomplete and we get an error
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

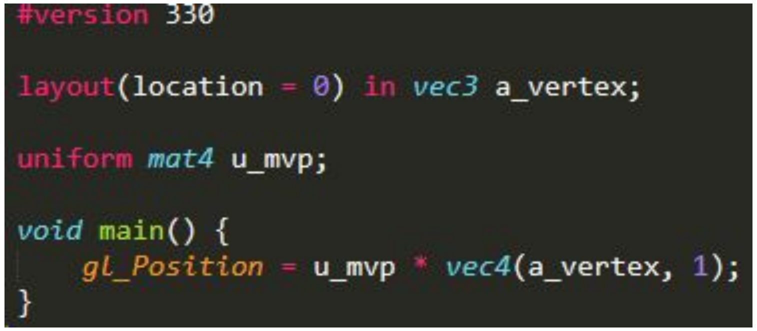
Safety first, safety second!



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

DepthShader

The vertex shader is just a simple mvp shader

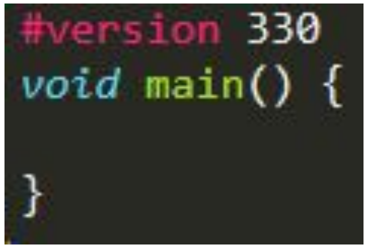


```
#version 330

layout(location = 0) in vec3 a_vertex;

uniform mat4 u_mvp;

void main() {
    gl_Position = u_mvp * vec4(a_vertex, 1);
}
```

and the fragment shader…?

DepthShader

We don’t write any colour!



```
#version 330
void main() {

}
```

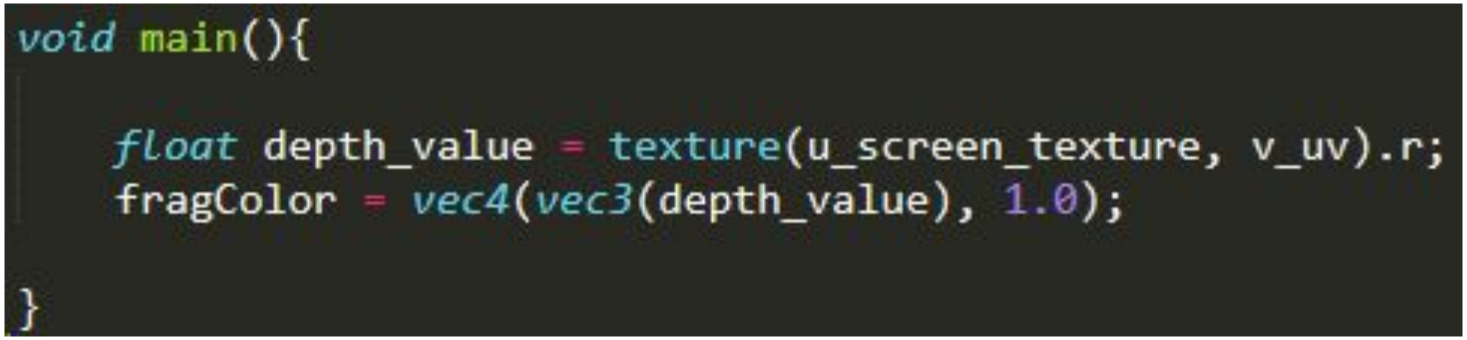
Tasks

1. Look at GraphicsSystem. defined variables for shadowmap
2. Complete initDepth in GraphicsUtilities
3. Call initDepth on this buffer with resolution of 1024x1024
4. Complete renderDepth function
5. Make a variable and compile a depth shader
6. Add a shadow pass to the scene
   a. bind shadow buffer
   b. render all meshes from perspective of light[0]
7. View shadowmap in bottom left corner

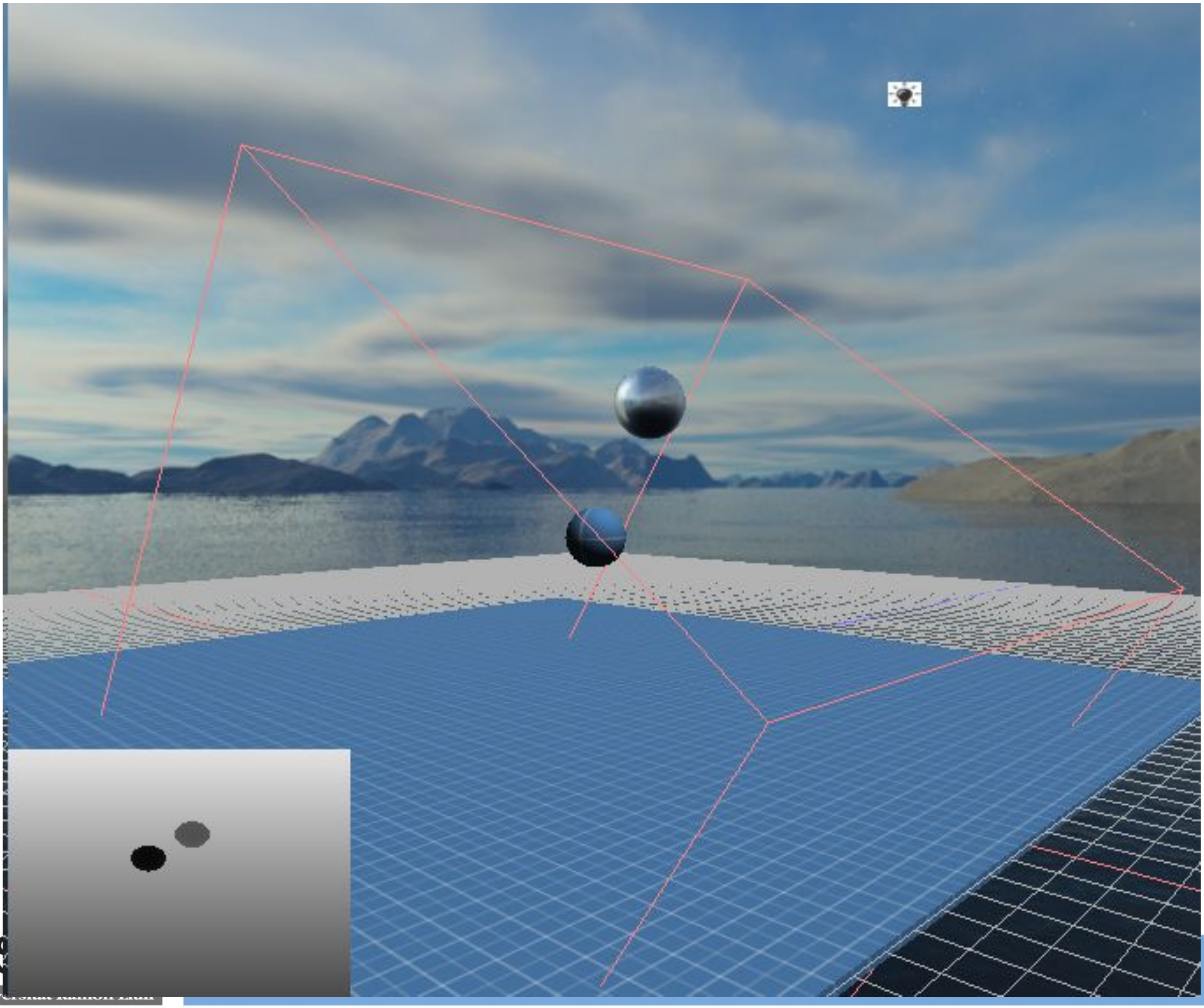
Viewing the shadow buffer

Need a special screen-space shader

Only sample first channel (there are no others)

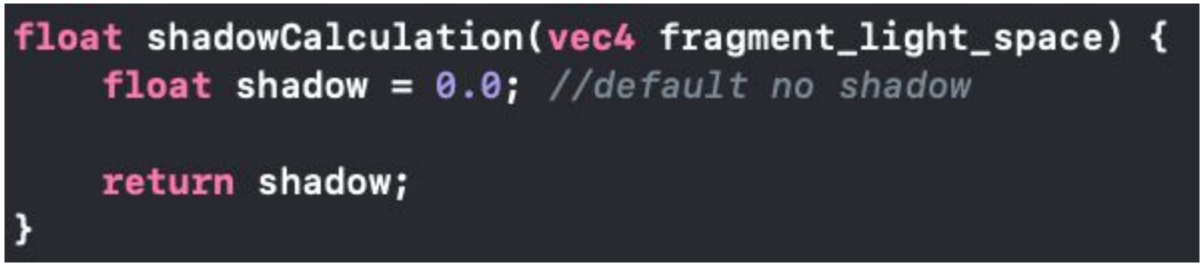


```
void main(){

    float depth_value = texture(u_screen_texture, v_uv).r;
    fragColor = vec4(vec3(depth_value), 1.0);

}
```



Shadow shader

Add function to any shader which has lights



```
float shadowCalculation(vec4 fragment_light_space) {
    float shadow = 0.0; //default no shadow

    return shadow;
}
```

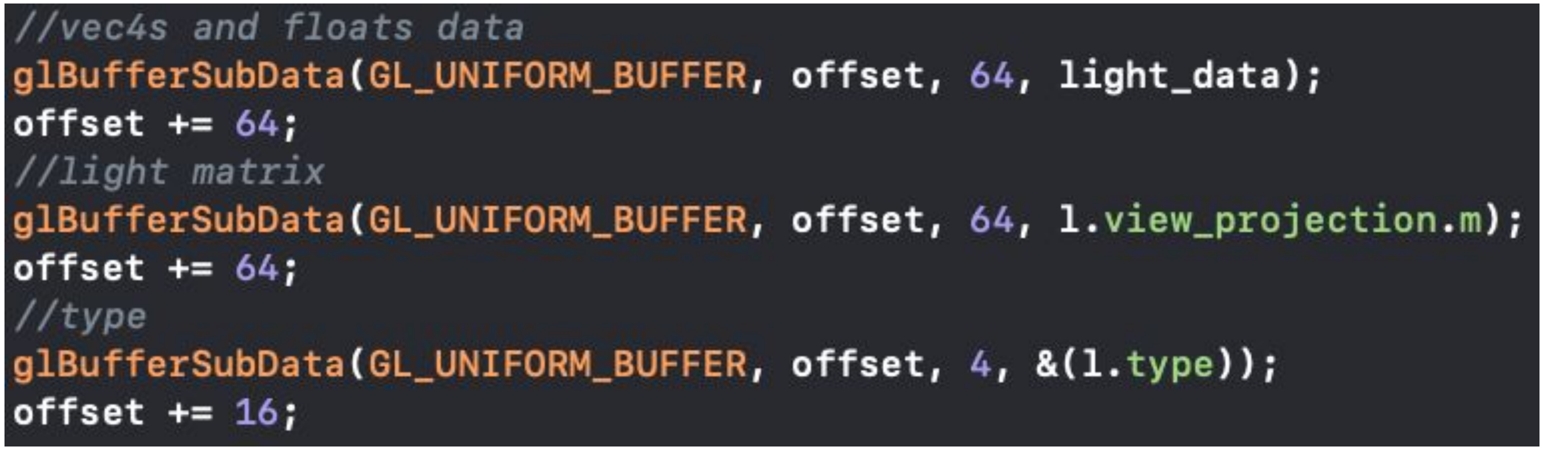
then multiply colour of light by (1-shadowCalculation)

(in full engine you need to check for light type etc.)

Calculating position in light space

Modify light UBO to include the light’s matrix.

A mat4 is 64 bit floats



```
//vec4s and floats data
glBufferSubData(GL_UNIFORM_BUFFER, offset, 64, light_data);
offset += 64;
//light matrix
glBufferSubData(GL_UNIFORM_BUFFER, offset, 64, l.view_projection.m);
offset += 64;
//type
glBufferSubData(GL_UNIFORM_BUFFER, offset, 4, &(l.type));
offset += 16;
```

Don’t forget to increase the UBO size

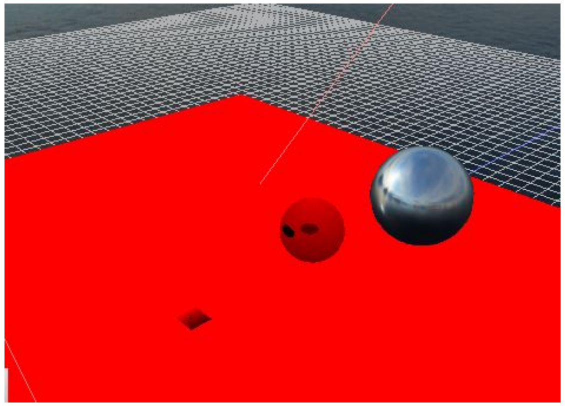
Upload shadow map

We'll just upload one for now, until next class

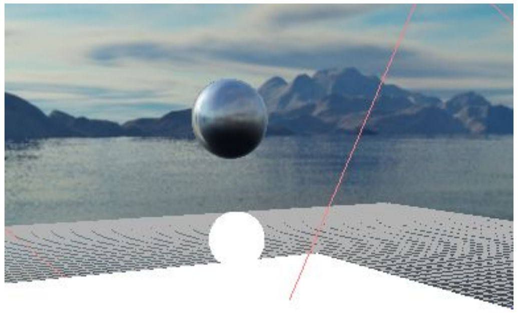
In setMaterialUniforms

```
//shadow stuff
shader_->setTexture(U_SHADOW_MAP, shadow_frame_.color_textures[0], 2);
```

Test!



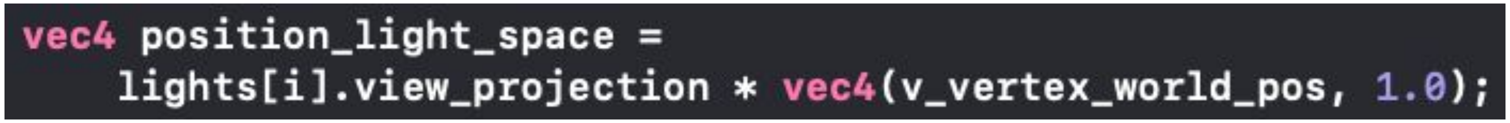
u_shadow_map texture should be like this



lights[0].view_projection[4][4] should = 1

Calculating fragment position in light space

Multiply fragment world space by light view_projection



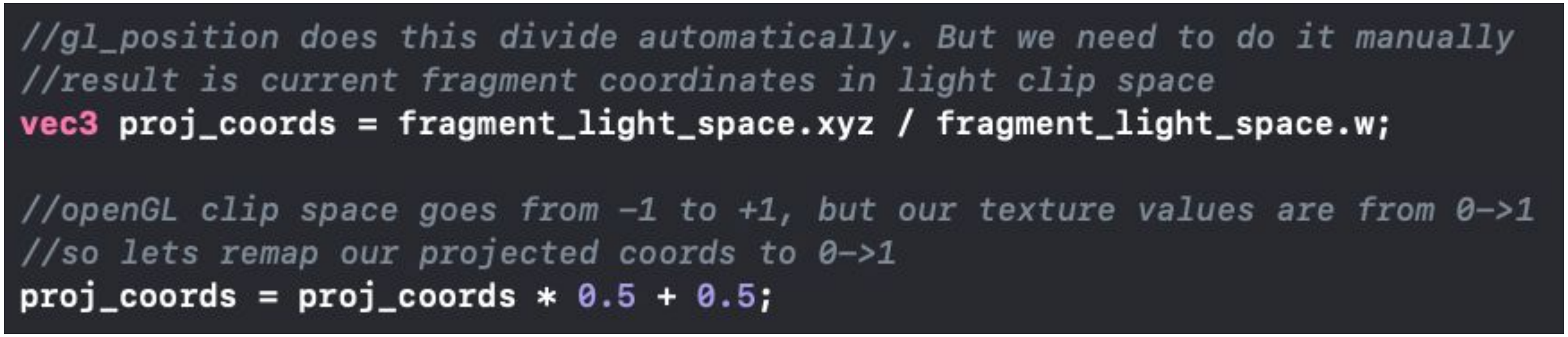
```
vec4 position_light_space =
    lights[i].view_projection * vec4(v_vertex_world_pos, 1.0);
```

pass this to our shadow function

Light clip-space to texture space

Don’t forget homogenous coordinates! Divide by zero to normalize the vector.

Now we are in NDC but must convert to 0->1

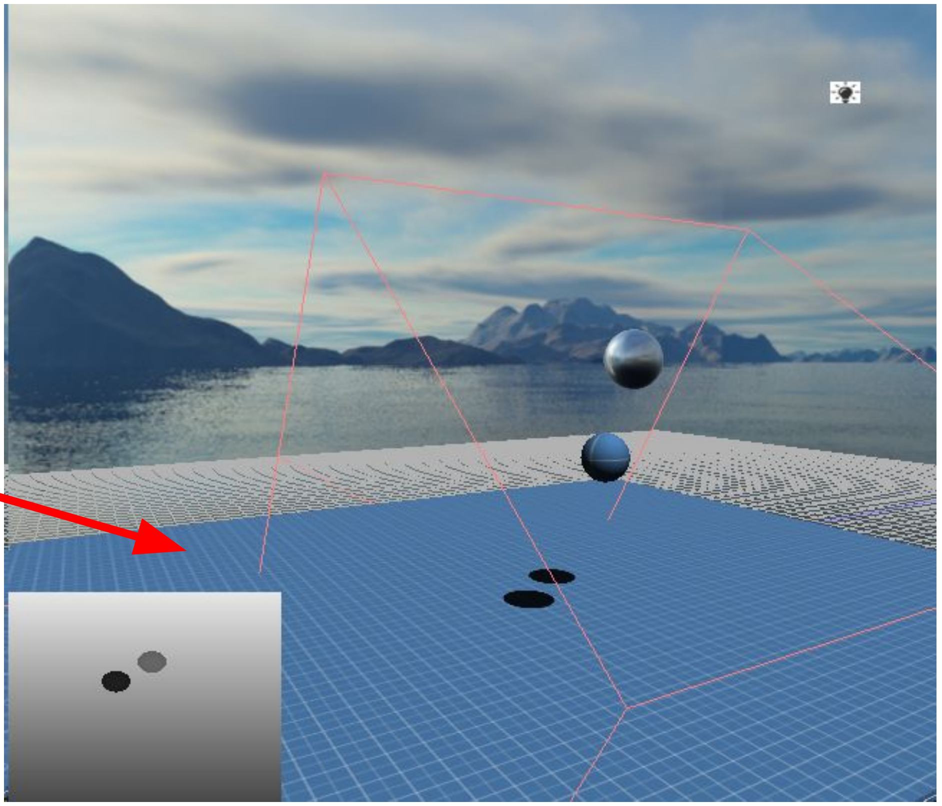


```
//gl_position does this divide automatically. But we need to do it manually
//result is current fragment coordinates in light clip space
vec3 proj_coords = fragment_light_space.xyz / fragment_light_space.w;

//openGL clip space goes from -1 to +1, but our texture values are from 0->1
//so lets remap our projected coords to 0->1
proj_coords = proj_coords * 0.5 + 0.5;
```

This works for both **texture coords and z-buffer coords** - very useful!

Clamping light space

Anything fragment the frustum doesn't touch is outside light space



We don't want to waste time calculating this, so we clamp between 0->1

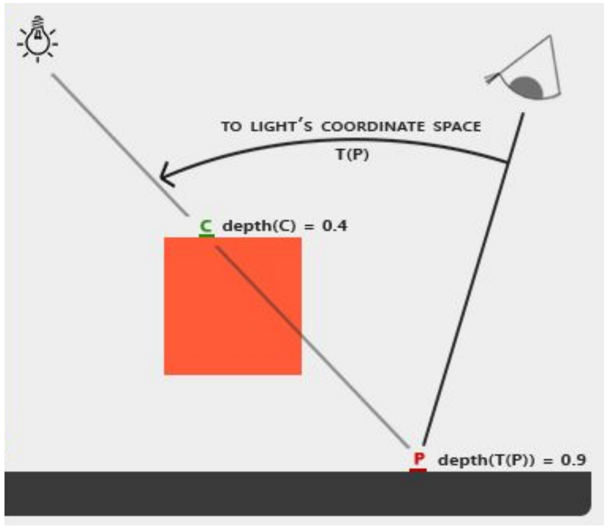
```
if (clamp(proj_coords, 0.0, 1.0) == proj_coords) {
```

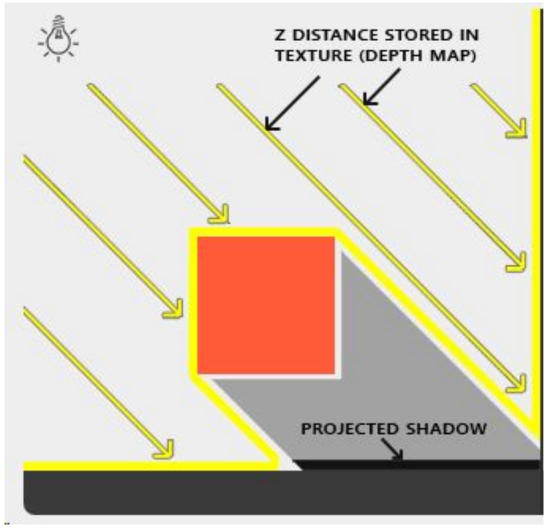
Comparing depth

We can now get two distance:

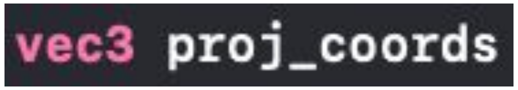
1) **Distance of fragment to light in rendered frame** - calculated from re-projection of fragment in light space
2) **Distance of fragment stored in shadow map** - calculated from texture look up

Need to compare them!





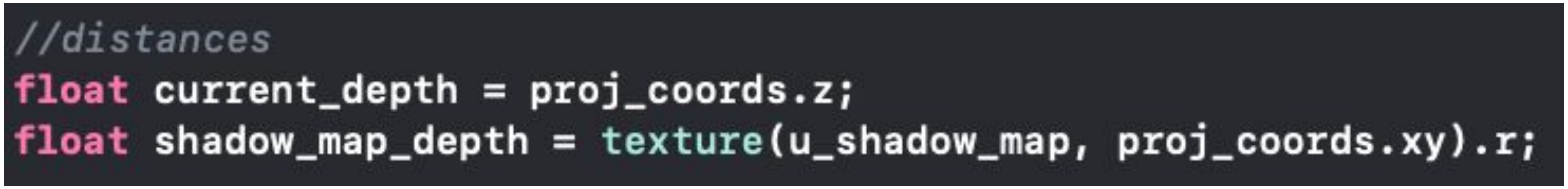
How to get distances



```
vec3 proj_coords
```

.z = distance from light in current frame

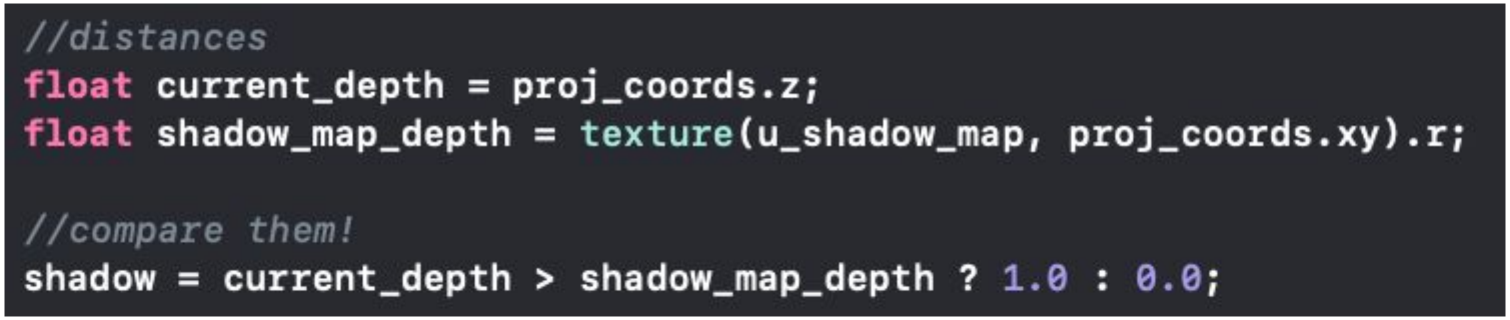
.xy = position in shadow map from where to get sample (don’t forget sample is a single channel in texture)



```
//distances
float current_depth = proj_coords.z;
float shadow_map_depth = texture(u_shadow_map, proj_coords.xy).r;
```

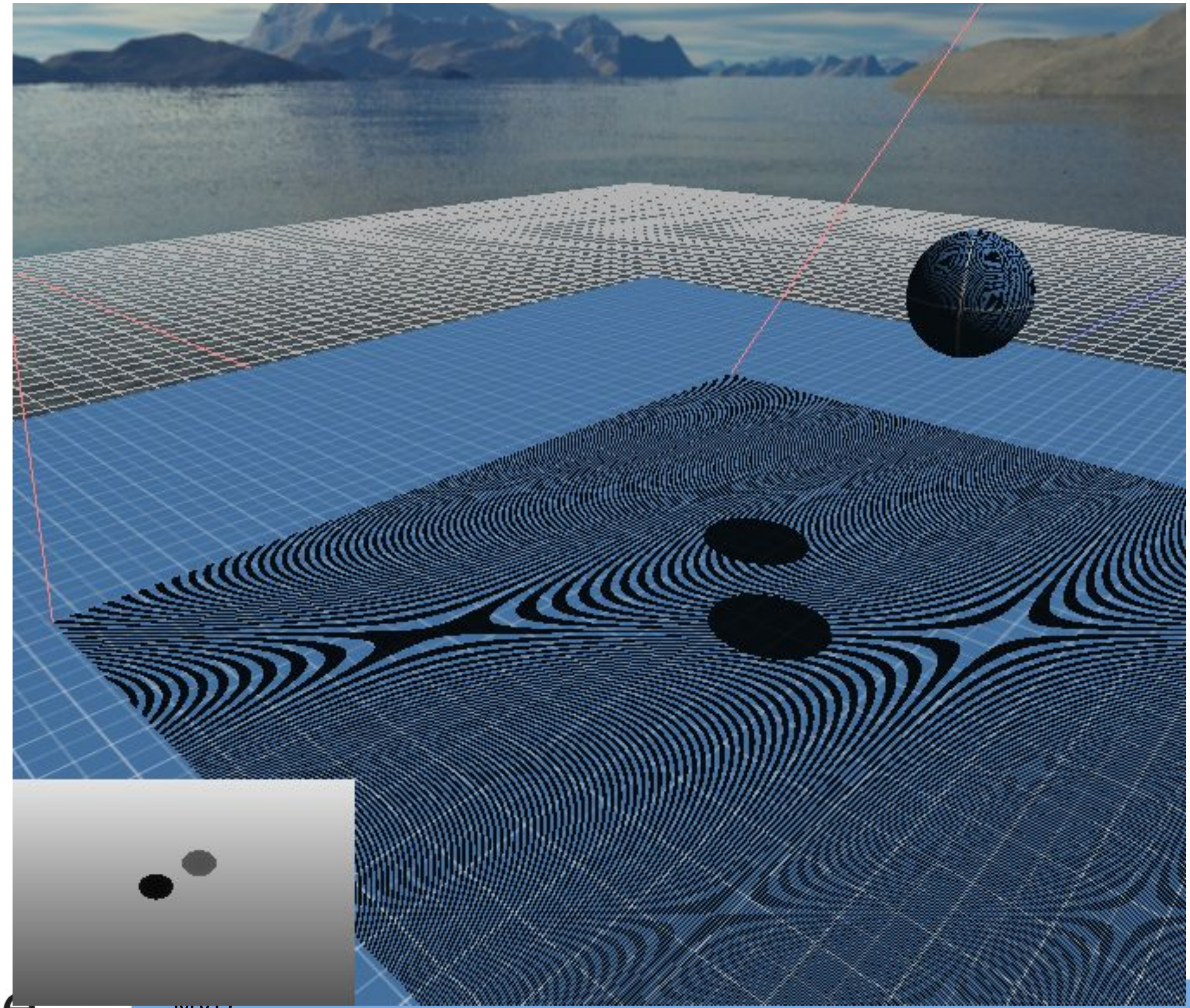
Compare distances

If current fragment distance is greater than shadow map, then draw shadow!



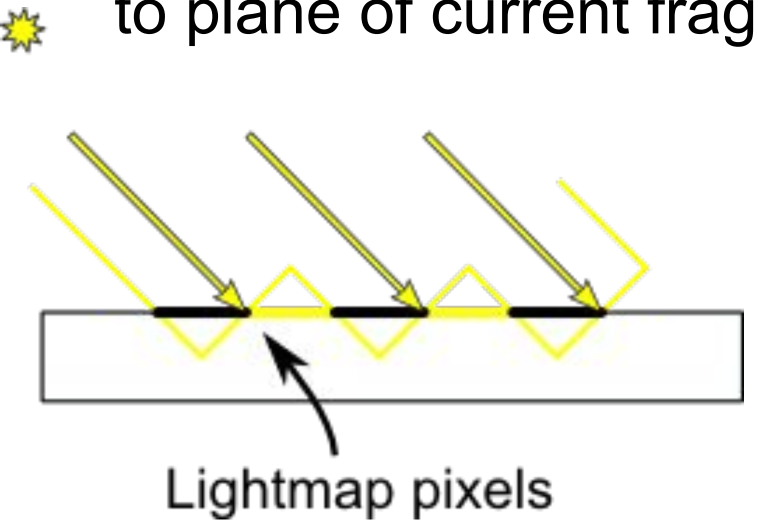
```
//distances
float current_depth = proj_coords.z;
float shadow_map_depth = texture(u_shadow_map, proj_coords.xy).r;

//compare them!
shadow = current_depth > shadow_map_depth ? 1.0 : 0.0;
```

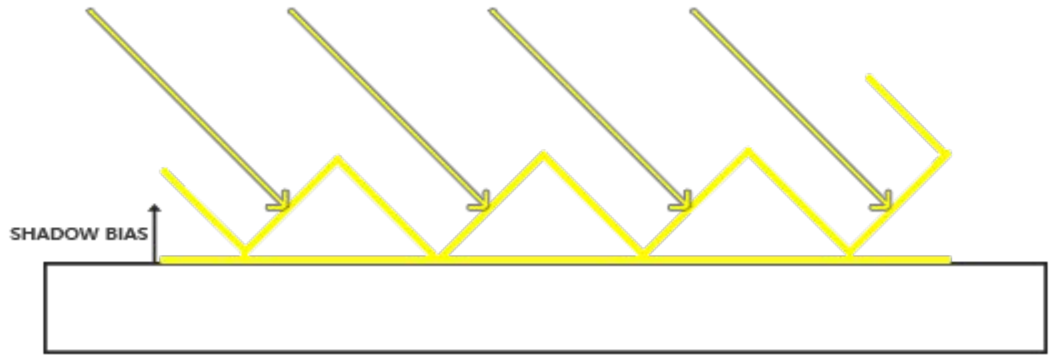
errrm...



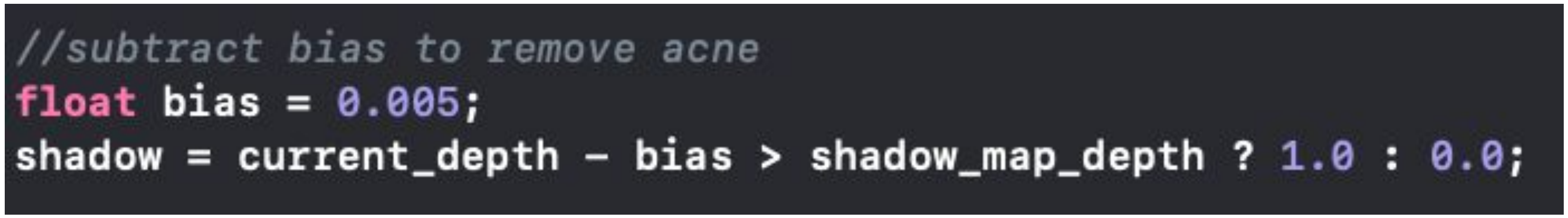
Shadow acne

Occurs because plane of shadow map fragment is different to plane of current fragment:



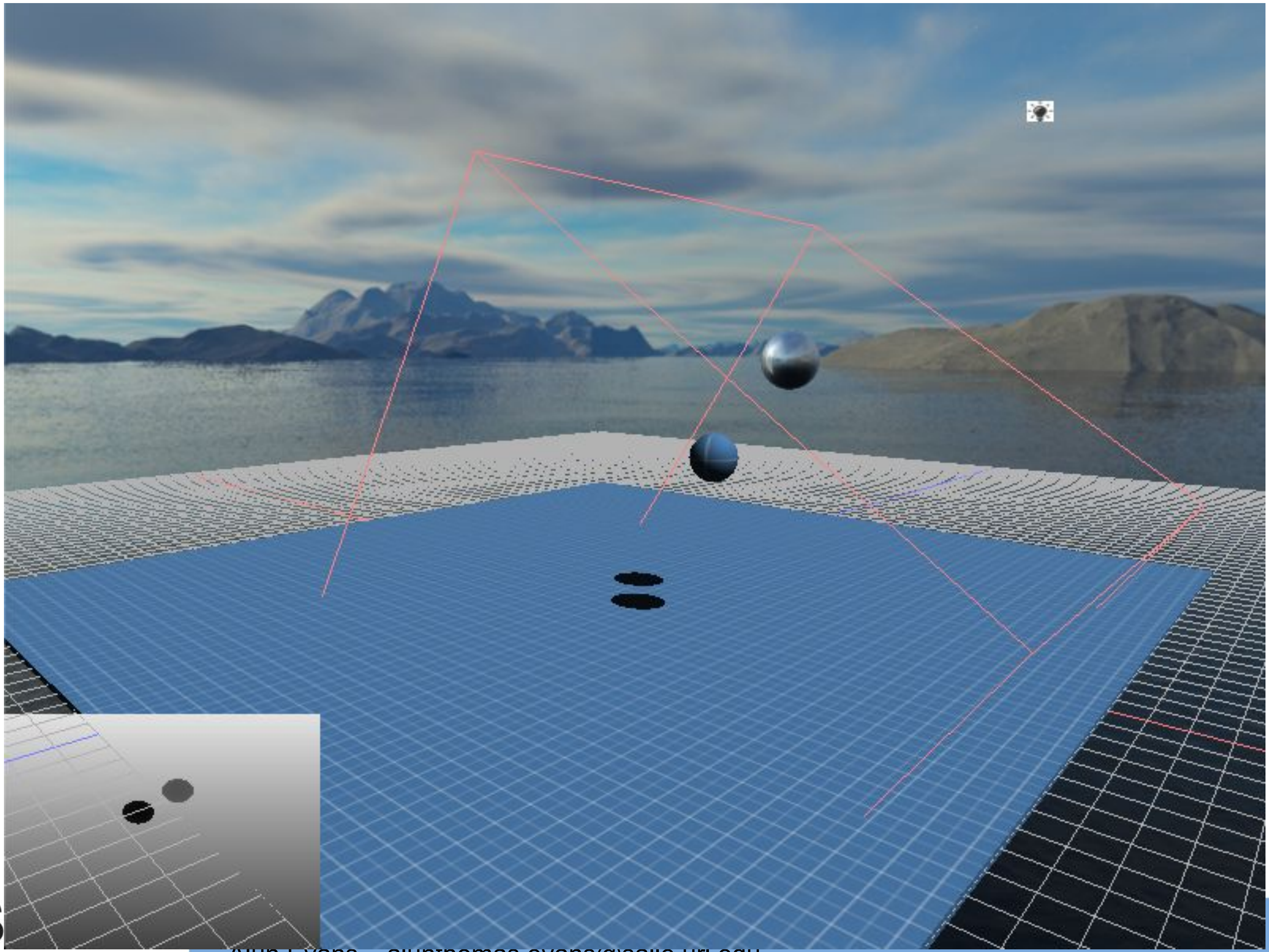


Solve by adding a *bias*



```
//subtract bias to remove acne
float bias = 0.005;
shadow = current_depth - bias > shadow_map_depth ? 1.0 : 0.0;
```

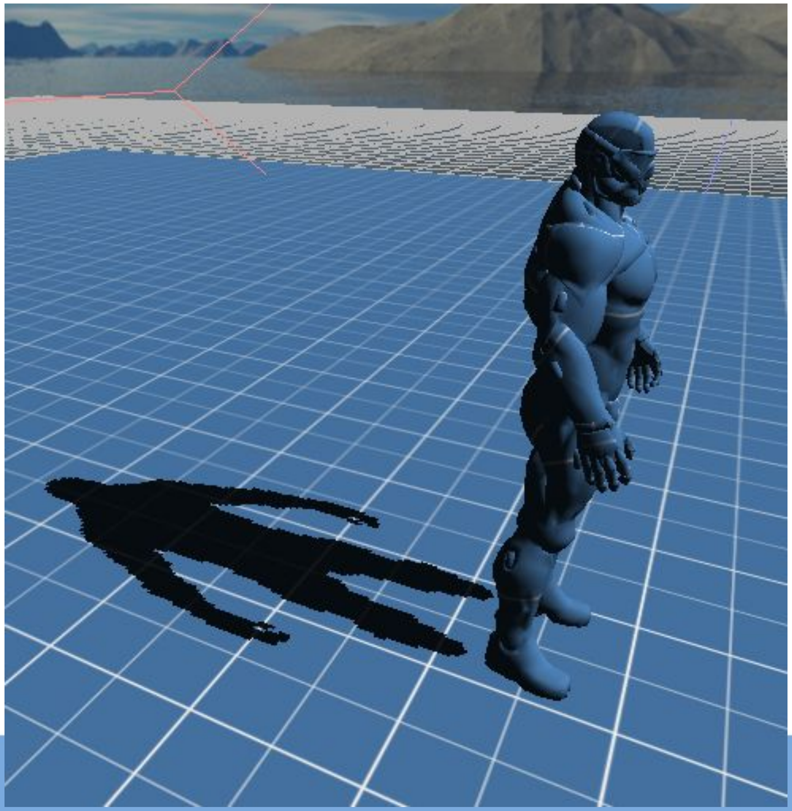
Hooray!



“Peter panning”

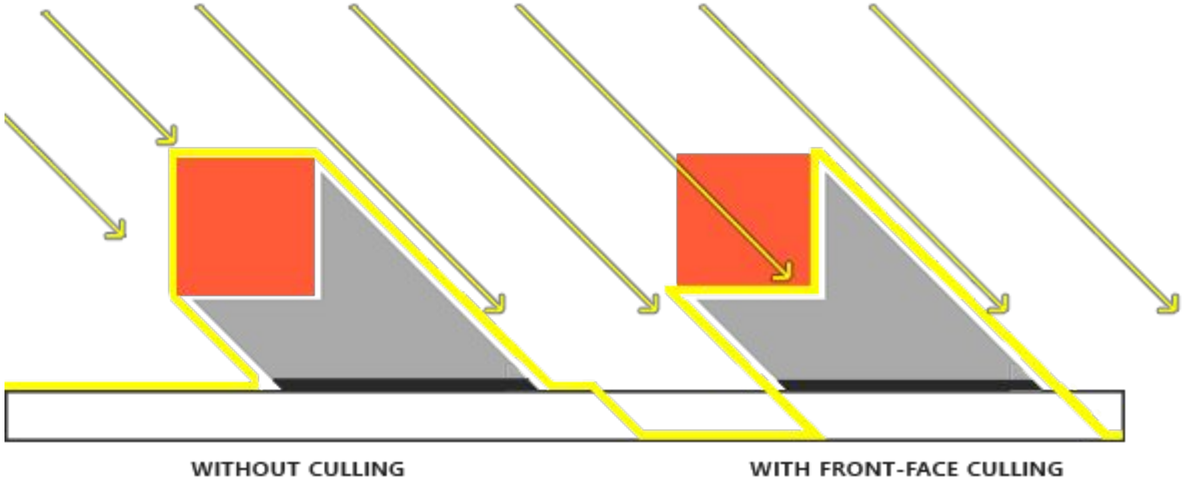
Bias creates a problem, particularly with perspective projection, when objects are ‘on-top’ of shaded surface

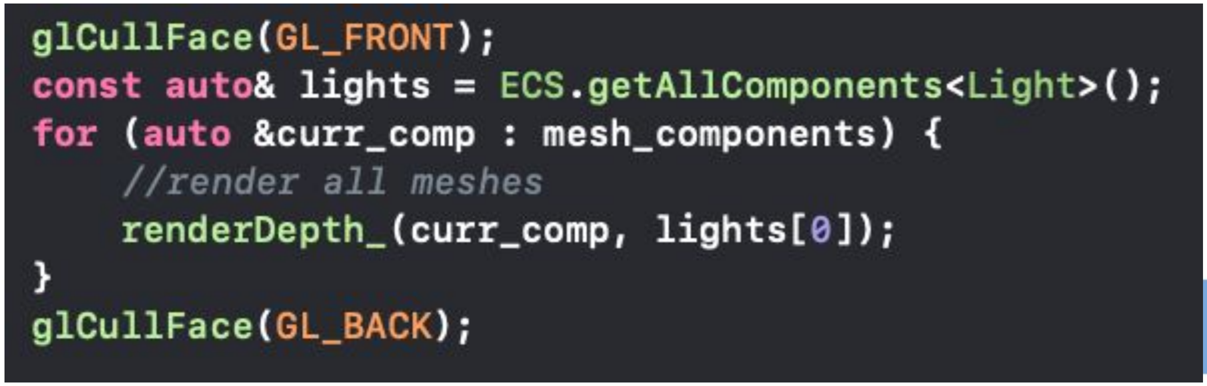
It looks like the object is floating!



Solve peter panning: shadow using back faces

By painting the shadowmap using the back faces of the object, we ensure that the shadow is ‘closer’, which removes peter panning





```
glCullFace(GL_FRONT);
const auto& lights = ECS.getAllComponents<Light>();
for (auto &curr_comp : mesh_components) {
    //render all meshes
    renderDepth_(curr_comp, lights[0]);
}
glCullFace(GL_BACK);
```

Advanced bias stuff

Getting the ‘right’ amount of bias is complex as peter panning is affected by

i) distance of object to shadowed surface

ii) angle of light to shadowed surface

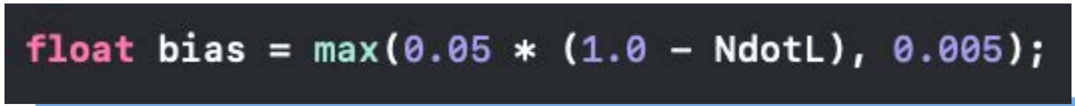
iii) resolution of shadow map

iv) size of frustum

v) culling

vi) perspective v orthographic light

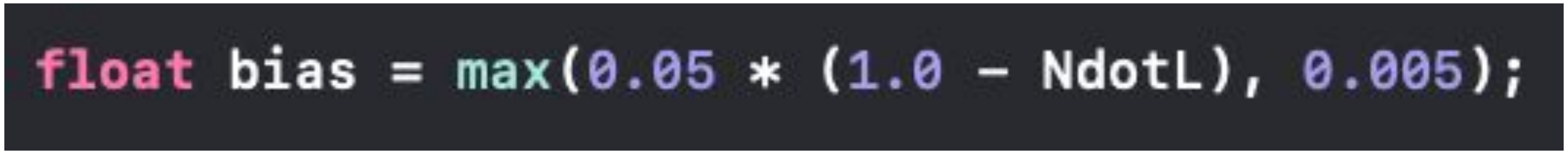
There is no ‘right’ value. g light:



```
float bias = max(0.05 * (1.0 - NdotL), 0.005);
```

Advanced bias

You can help things by applying a different bias to surfaces facing



```
float bias = max(0.05 * (1.0 - NdotL), 0.005);
```

This is known as ‘**slope-scale depth bias’** as it uses the slope of the surface (relative to light) to modify the bias. The slope is represented by the surface normal

(BTW Direct3D 10+ does this automatically)

Task: Experimentation time

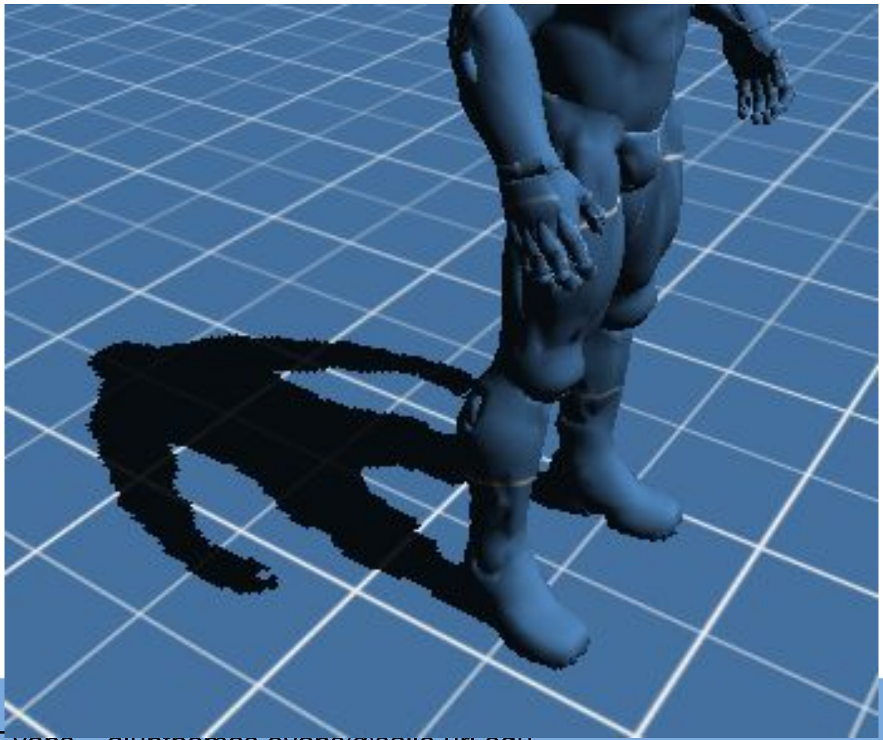
Experiment with:

1) Loading different object (e.g. the nanosuit)
2) Rotate to cast self shadows
3) Setting the correct bias value
4) Changing the resolution of the shadow map
5) Swapping orthographic vs perspective projection
6) Changing frustum near and far planes

Take a screenshot of the effect of changing each of these, dump into a document

Softening the shadow border

Currently the shadow is pixelated, due to projection of pixels of shadow map (projection aliasing)



Soft shadows - general approach

The general idea of soft-shadowing is to sample *more than one pixel* in the shadow map, and average the results. e.gñ

A	B

Shadow value for A = (A + B) / 2

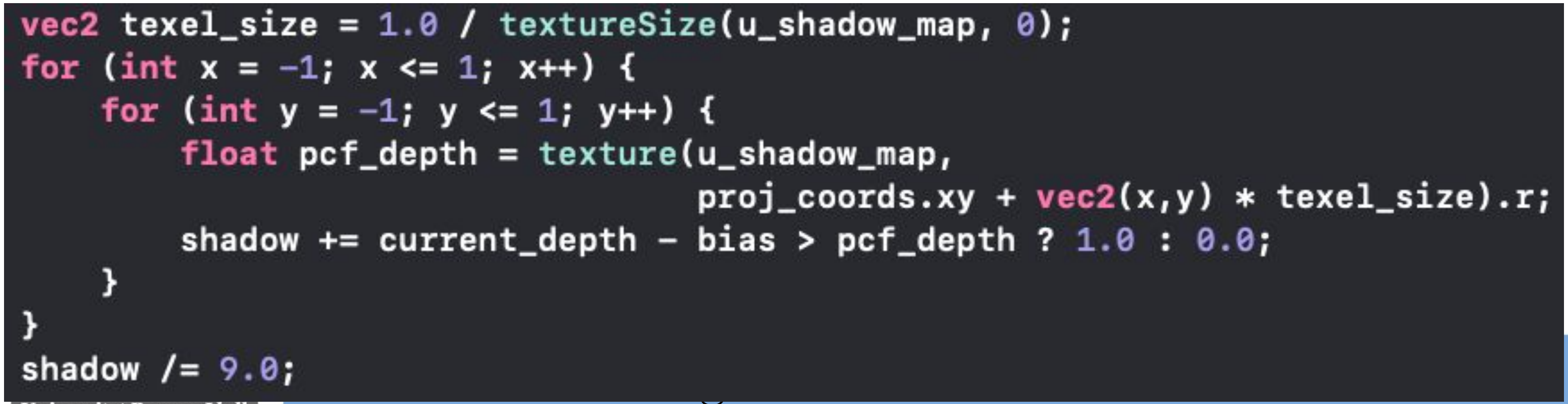
So if A is in shadow, but B isn't, then

shadow value of A = 0.5

Percentage closer filtering

1	1	1
1	T/9	1
1	1	1

Sample 9 texels around center. Add them up and divide by 9. Use textureSize to get size of texel in 0->1



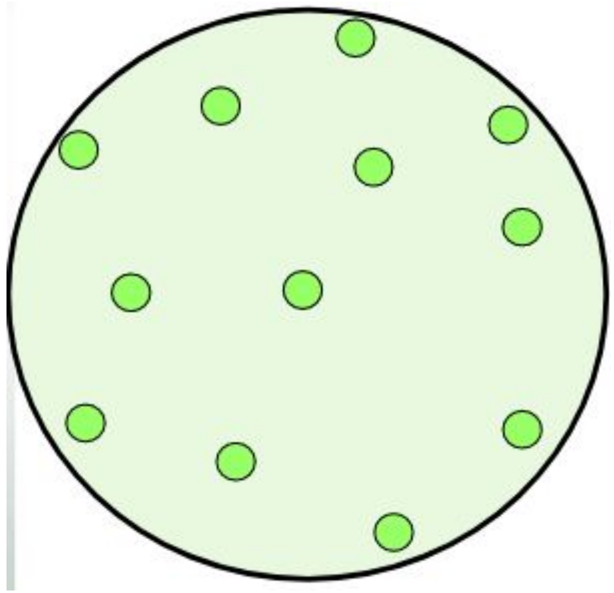
```
vec2 texel_size = 1.0 / textureSize(u_shadow_map, 0);
for (int x = -1; x <= 1; x++) {
    for (int y = -1; y <= 1; y++) {
        float pcf_depth = texture(u_shadow_map,
                                  proj_coords.xy + vec2(x,y) * texel_size).r;
        shadow += current_depth - bias > pcf_depth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

Poisson sampling

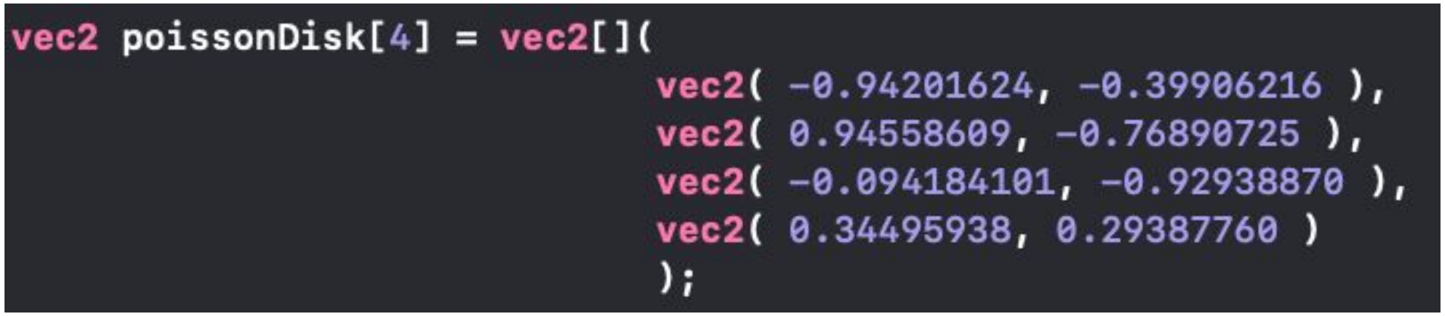
“Humans prefer noise to aliasing”

To reduce samples, we can find “random” samples in a disk around the texel.

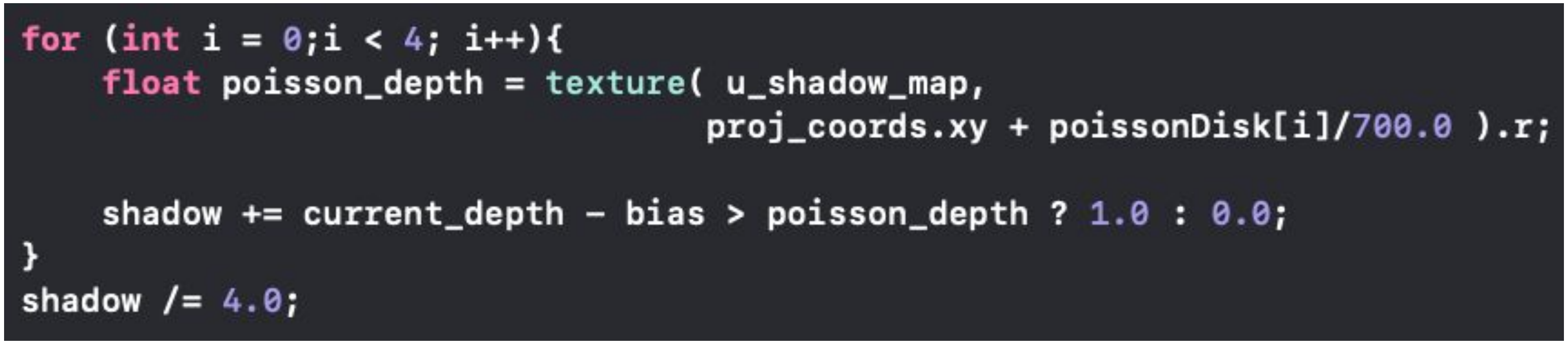
This is called poisson sampling



Poisson sampling code



```
vec2 poissonDisk[4] = vec2[](
                            vec2( -0.94201624, -0.39906216 ),
                            vec2( 0.94558609, -0.76890725 ),
                            vec2( -0.094184101, -0.92938870 ),
                            vec2( 0.34495938, 0.29387760 )
                            );
```



```
for (int i = 0;i < 4; i++){
    float poisson_depth = texture( u_shadow_map,
                                  proj_coords.xy + poissonDisk[i]/700.0 ).r;

    shadow += current_depth - bias > poisson_depth ? 1.0 : 0.0;
}
shadow /= 4.0;
```

But this code is not random!!

It’s just the same code repeated! We need a random function

```
float random(vec4 seed4){
    float dot_product = dot(seed4, vec4(12.9898,78.233,45.164,94.673));
    return fract(sin(dot_product) * 43758.5453);
}
```

and a random number generator between 0-4 for the index

```
int index = int(4*random(vec4(gl_FragCoord.xyy, i))) % 4;
```

seed for our random number generator is combination of i and the current fragment coord (could be vertex position)

Task: screenshot the differences

Implement PCF with

- 9 taps (3x3)
- 16 taps (4x4)
- 25 taps (5x5)

Implement Poisson filter

- with 4 taps non-random
- 4 taps random
- Do some research to find a larger poisson filter

Advanced shadow maps

Shadow mapping is a **HACK**!!

Unfortunately, it’s a necessary hack for drawing shadows with rasterisation. Doubly unfortunately, and like all hacks, its easy to do it wrong.

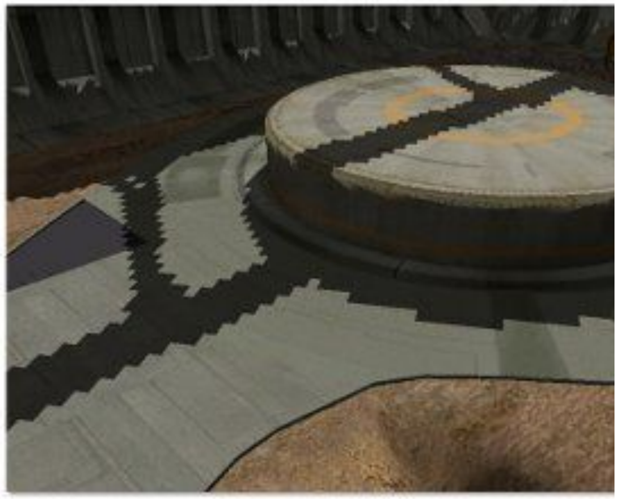
To get around the hack, shadow mapping has seen a whole bunch of research. Most of it is really interesting.

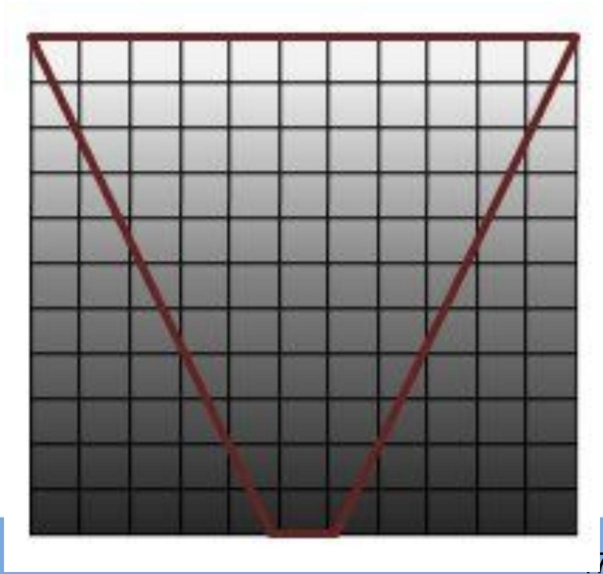
Here are some of the highlights:

Automatic frustum fitting

Perspective aliasing is what happens when objects at the ‘back’ of the frustum have more aliased shadows

This is because the perspective projection matrix scales more at far plane

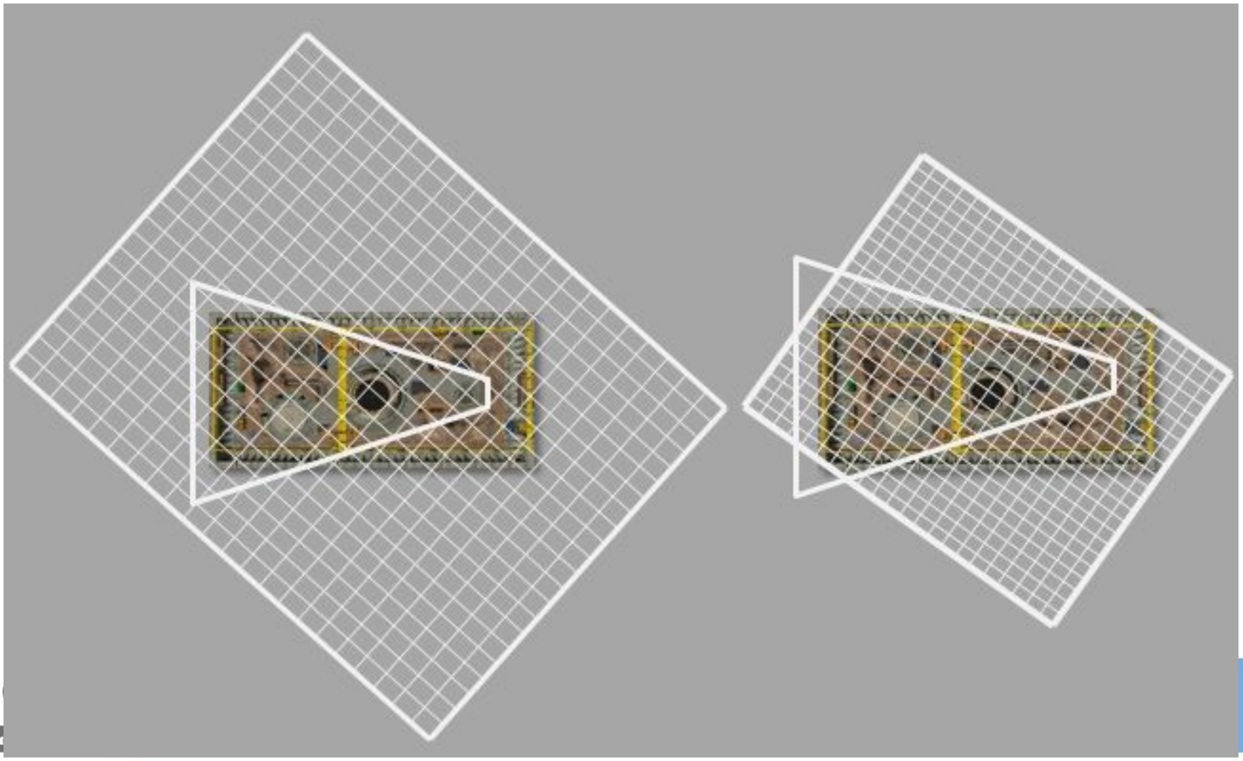




Automatic frustum fitting (2)

Light frustum = rectangular grid. Perspective frustum = camera.

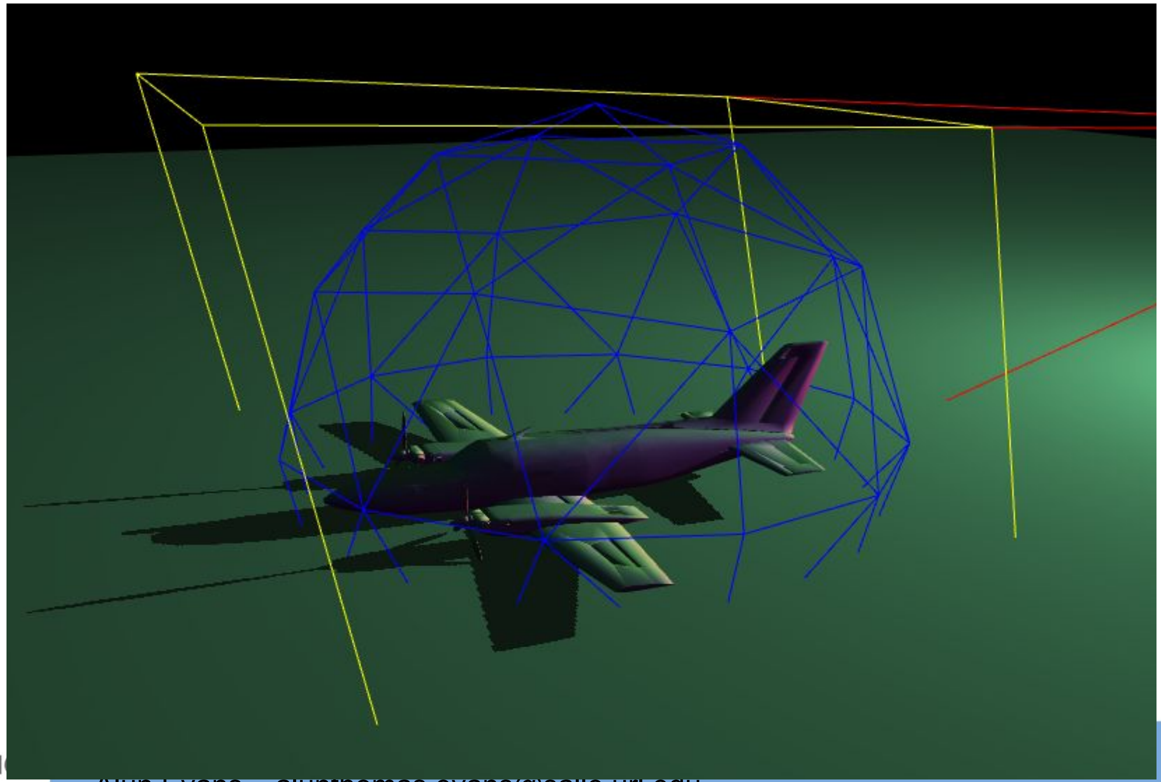
Left - wasted resolution. Right - fit light frustum to bounding box of object (or scene)



Automatic Frustum fitting (3)

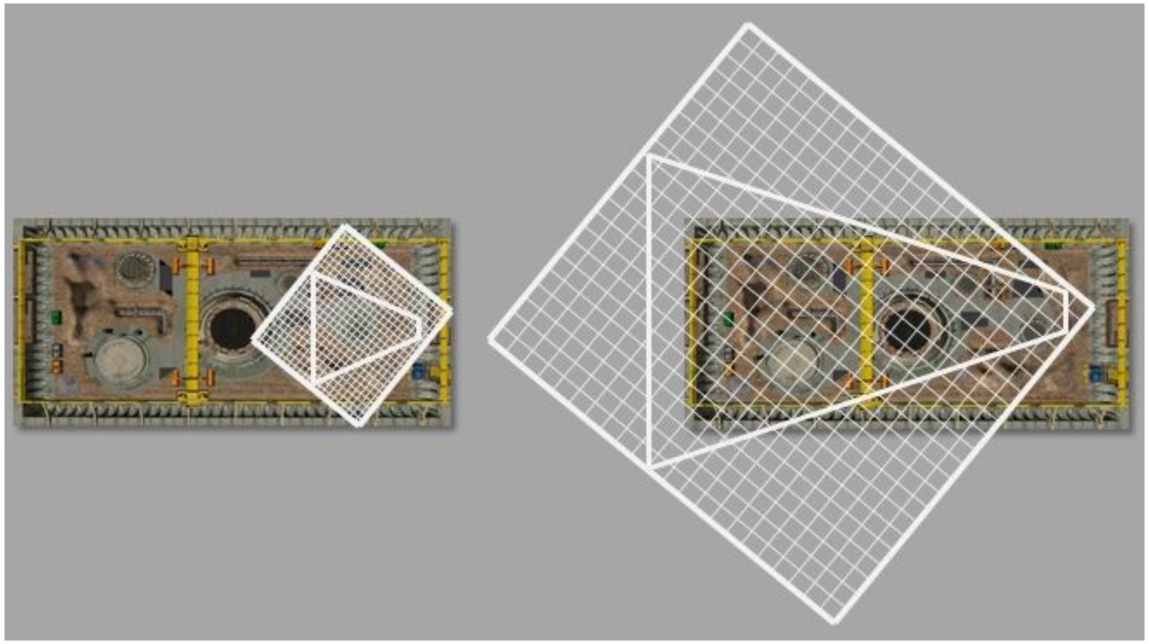
Real-time example in WebGL

http://evanw.github.io/lightgl.js/tests/shadowmap.html



Automatic frustum fitting

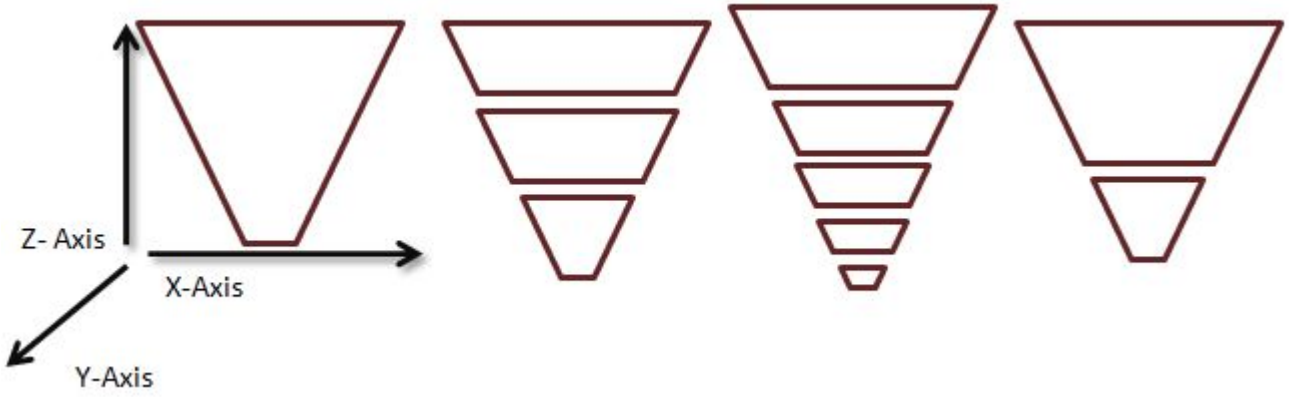
Another approach is to fit the light frustum to the camera frustum



The advantage of this is that it is highly optimal, as it only shadows things that the camera can see.

Cascading shadow maps

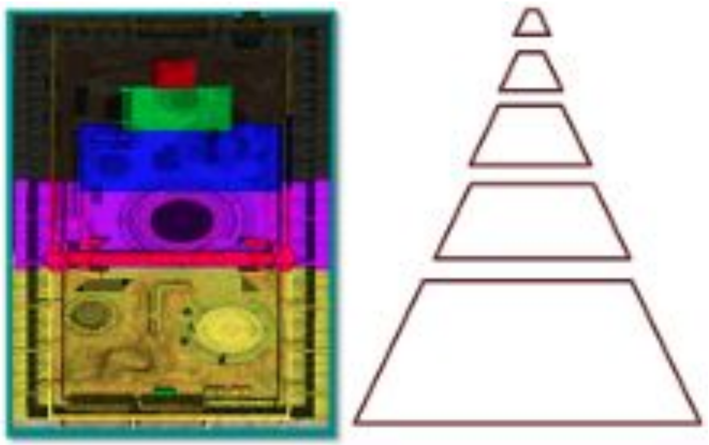
Cascading shadow maps are a way of varying the resolution of the shadow map depending on the depth of the projection.

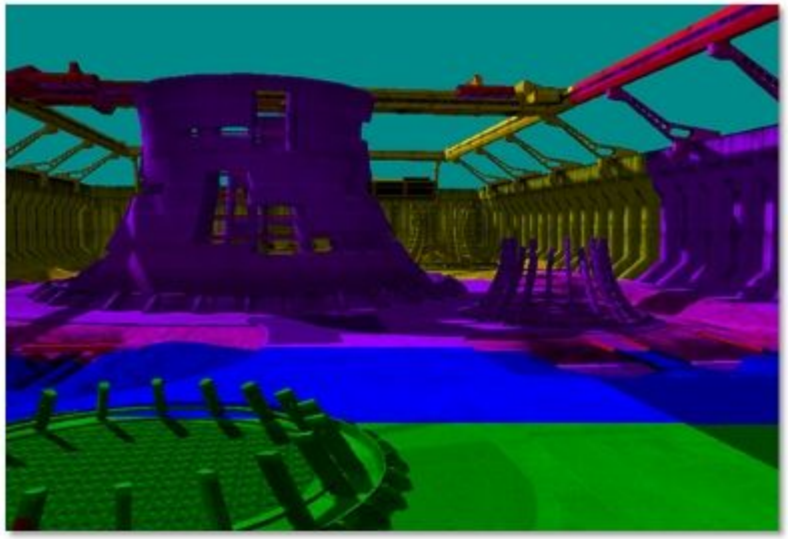


The approach is to partition the frustum based on depth, and apply a different shadow map in different region of frustum

Cascading Shadow Maps (2)

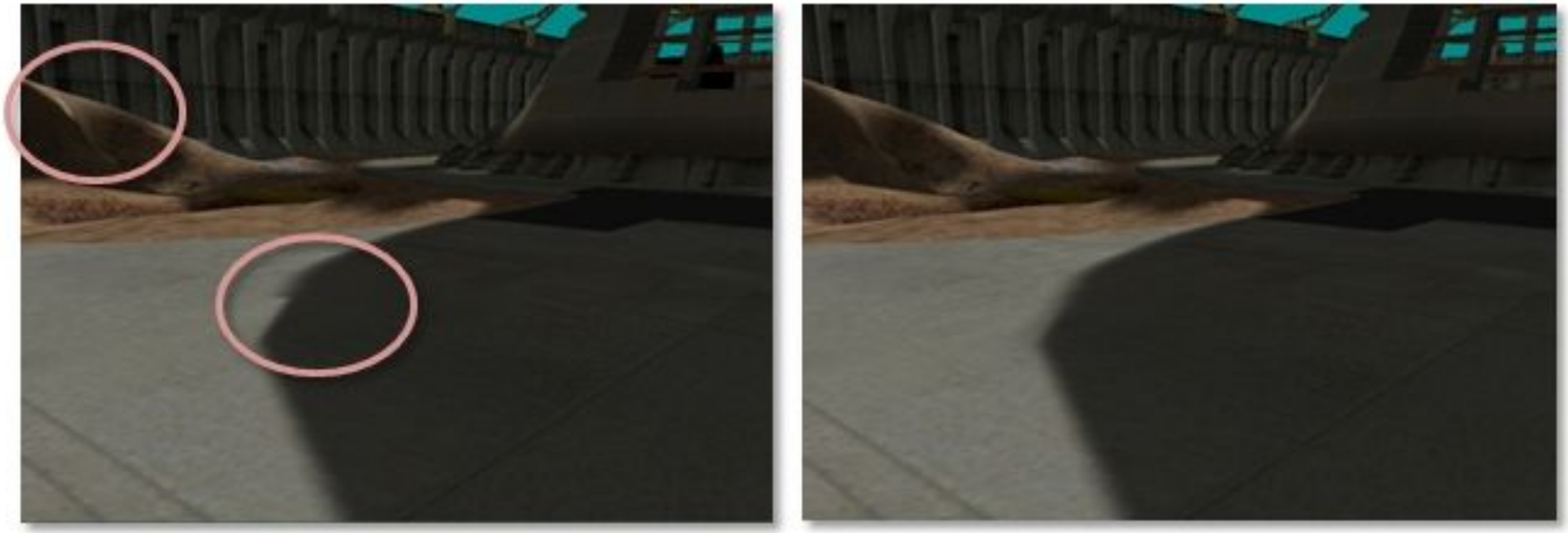
Applying to scene





Cascading Shadow Maps (3)

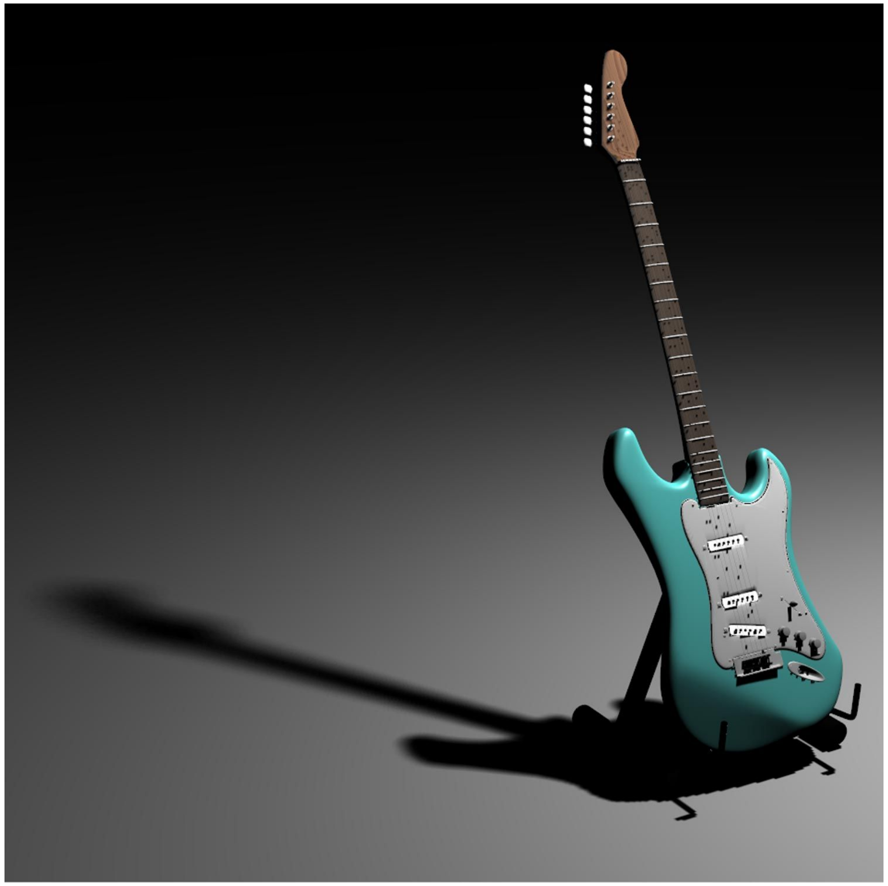
To avoid jump in cascade levels, create a band where you sample **both** levels, and blend between the two:



Percentage closer soft shadows (2)

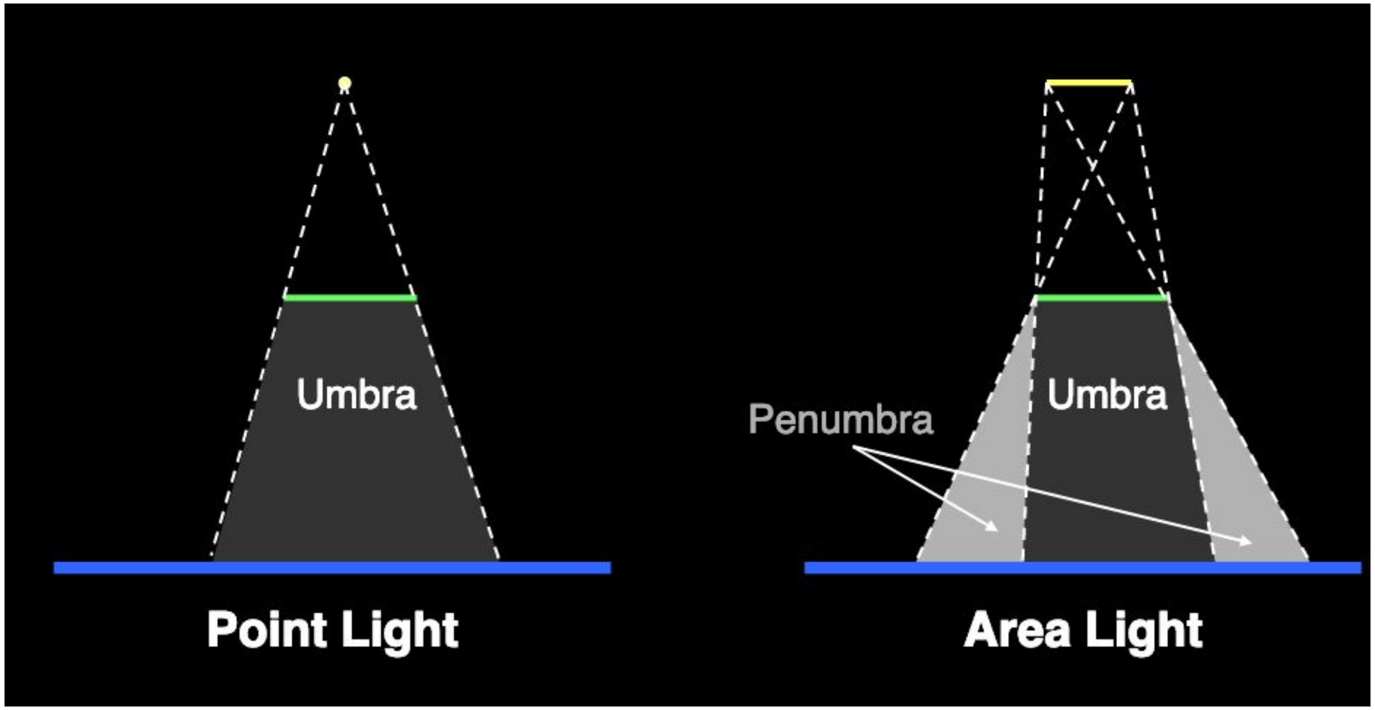
In the real world, the shadow 'blur' changes according to the distance to the collider

The blurry bit is called the 'penumbra'

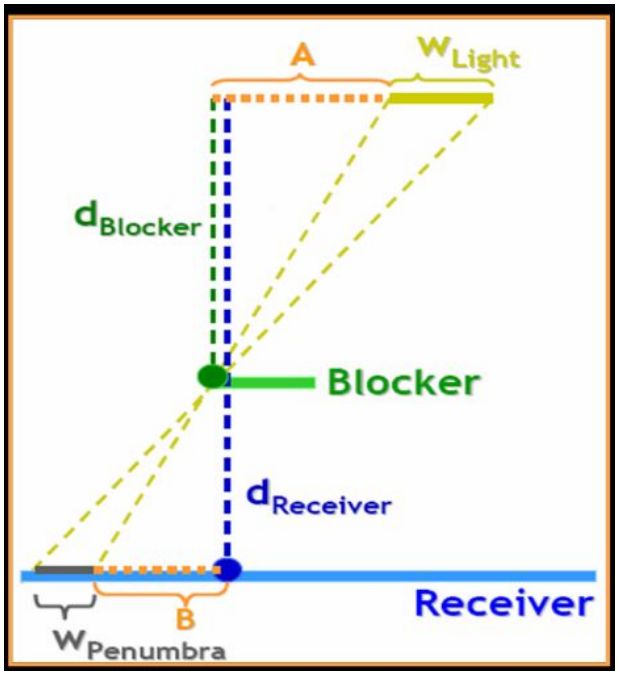


PCSS (2)

This occurs because light sources are frequently not a single point - they are an *area*.



PCSS (3)



PCSS directly addresses this by representing each light source as an area.

Estimates penumbra width using relative distances blocker, receiver and light

Varies PCF kernel width depending on this value.

More expensive as need to 'search' for blocker

...and loads more

Variance shadow maps

Exponential shadow maps

Light-space perspective shadow maps

Computation masking and mipchain dilation

Silhouette mapping

Receiver plane depth bias

….

References

Pointlight shadows

https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows

Bias and Automatic Frustum Fitting

https://docs.microsoft.com/en-us/windows/desktop/dxtecharts/common-techniques-to-improve-shadow-depth-maps

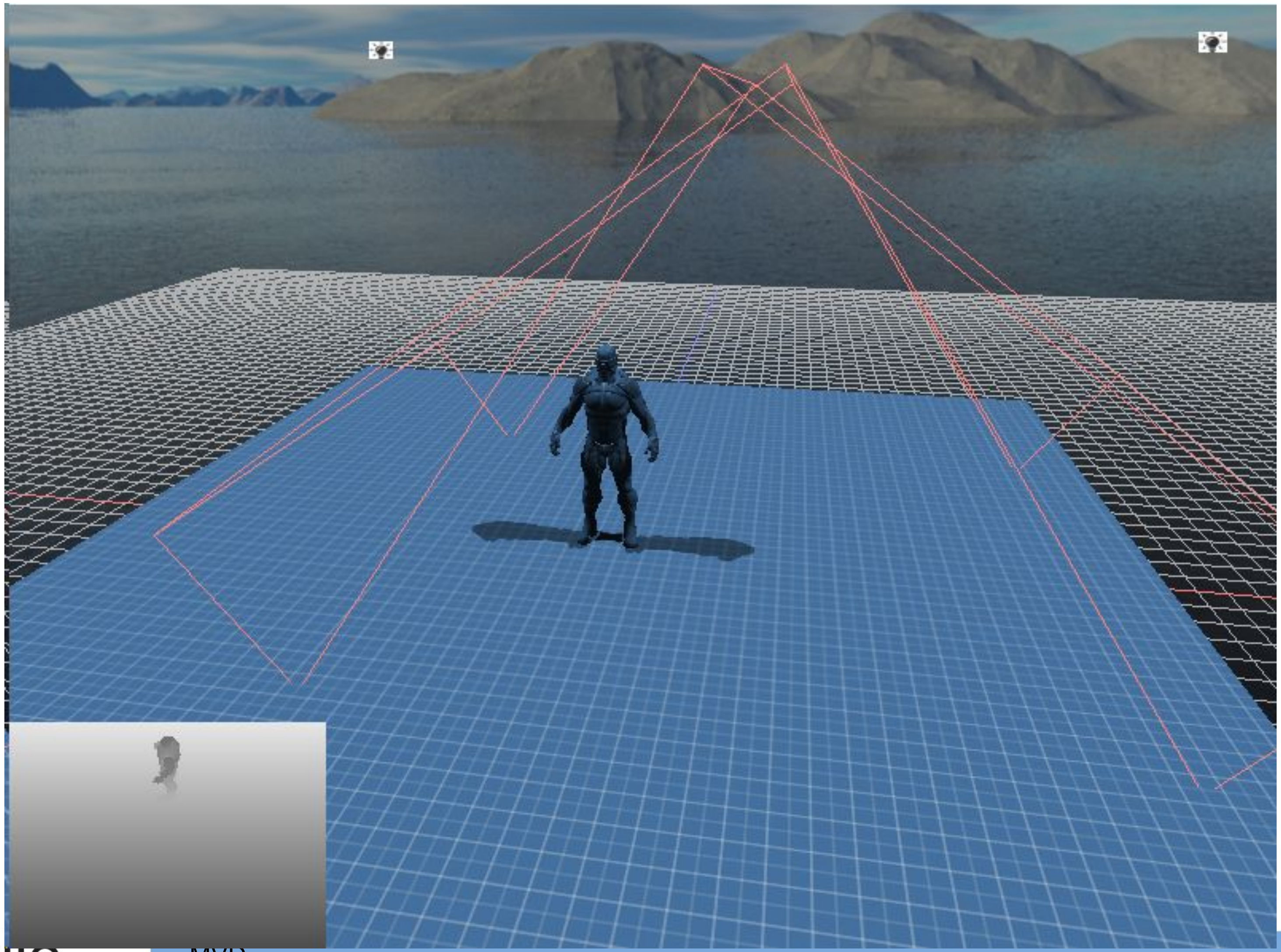
Cascading Shadow Maps

https://docs.microsoft.com/en-us/windows/desktop/dxtecharts/cascaded-shadow-maps

Percentage Closer Soft Shadows

https://http.download.nvidia.com/developer/presentations/2005/SIGGRAPH/Percentage_Closer_Soft_Shadows.pdf

Multiple lights



Task

Modify our engine to include multiple lights.

Considerations

- Add resolution of shadow map to light comp
- Create array of samplers in shader
- Create array of framebuffers (in Game::lateInit)
- Reserve first MAX_LIGHTS texture units for lights
- Need to activate textures and set sampler uniforms 'the old fashioned way'
- reduce diffuse color of lights to avoid burning