Computer Graphics II – Point Shadows

Kai Lawonn

Introduction

- Last lecture, learned to create dynamic shadows with shadow mapping
- Works for directional lights as the shadows are only generated in a single direction of the light source
- Also known as directional shadow mapping as the depth (or shadow) map is generated from just the direction the light is looking at

Introduction

- Now, focus on generation of dynamic shadows in all surrounding directions
- This is perfect for point lights as a real point light would cast shadows in all directions
- This technique is known as point (light) shadows or more formerly as omnidirectional shadow maps

Introduction

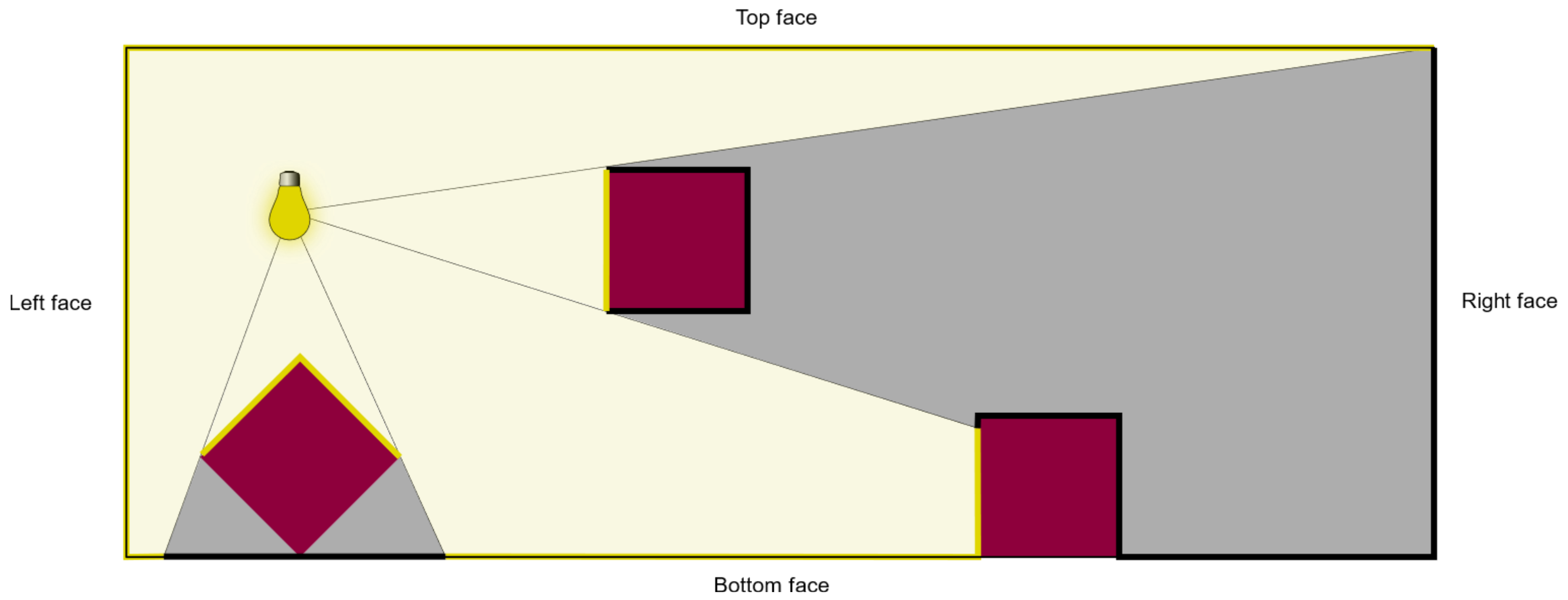
- Algorithm mostly the same to directional shadow mapping:
  - Generate a depth map from the light’s perspective(s)
  - Sample depth map based on the current fragment position
  - Compare each fragment with the stored depth value (is it in shadow?)
- Main difference between both shadow mapping techniques is the depth map used

Introduction

- Depth map requires rendering a scene from all surrounding directions of a point light
- Normal 2D depth map will not work → cubemap
- Cubemap store environment data with only 6 faces, render entire scene to each of the faces and sample these as the point light's surrounding depth values

Introduction

- Generated depth cubemap passed to the lighting fragment shader
- It samples the cubemap with a direction vector to obtain the depth (from the light's perspective) at that fragment



Generating the Depth Cubemap

Generating the Depth Cubemap

- Have to render the scene 6 times: once for each face
- One (quite obvious) way to do → render scene 6 times with 6 different view matrices, each time attaching a different cubemap face to a framebuffer object:

```
for (unsigned int i = 0; i < 6; i++)
{
   GLenum face = GL_TEXTURE_CUBE_MAP_POSITIVE_X + i;
   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, face,
                                                            depthCubemap, 0);
   BindViewMatrix(lightViewMatrices[i]);
   RenderScene();
}
```

Generating the Depth Cubemap

- Can be quite expensive (lot of render calls are necessary for just a single depth map)
- Going to use an alternative (more organized) approach using a little trick in the geometry shader that allows us to build the depth cubemap with just a single render pass

Generating the Depth Cubemap

- First, create a cubemap:

```
unsigned int depthCubemap;
glGenTextures(1, &depthCubemap);
```

- Generate each cubemap faces as 2D depth-valued texture images:

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
for (unsigned int i = 0; i < 6; ++i)
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
              GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0,
              GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

Generating the Depth Cubemap

- Set the texture parameters:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Generating the Depth Cubemap

- Normally attach single face of a cubemap texture to the framebuffer object and render the scene 6 times, each time switching the depth buffer target of the framebuffer to a different cubemap face
- But, use a geometry shader to render to all faces in a single pass, can directly attach the cubemap as a framebuffer's depth attachment using glFramebufferTexture:

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Generating the Depth Cubemap

- glDrawBuffer and glReadBuffer: only care about depth values when generating a depth cubemap → tell OpenGL this FBO does not render to a color buffer

Generating the Depth Cubemap

- With omnidirectional shadow maps, two render passes:
  - 1st: generate the depth map
  - 2nd use depth map in the normal render pass to create shadows
- With FBO and the cubemap this process looks a bit like this:

```
// 1. first render to depth cubemap
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// 2. then render scene as normal with shadow mapping (using depth cubemap)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
RenderScene();
```

Light Space Transform

- Framebuffer and cubemap are set
- Now, transform scene’s geometry to the relevant light spaces in all 6 directions of the light
- Similar to shadow mapping, need a light space transformation matrix T, but this time one for each face

Light Space Transform

- Light space transformation contains a projection and a view matrix
- Projection matrix use a perspective projection matrix (light source represents a point in space)
- Each light space transformation uses the same projection matrix:

```
float aspect = (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT;
float near_plane = 1.0f;
float far_plane  = 25.0f;
glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f), aspect,
                                        near_plane, far_plane);
```

Light Space Transform

- Important to note, the field of view parameter is set to 90 degrees
- By setting this, make sure viewing field is exactly large enough to properly fill a single face of the cubemap (all faces align correctly to each other at the edges)

```
float aspect = (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT;
float near_plane = 1.0f;
float far_plane  = 25.0f;
glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f), aspect,
                                            near_plane, far_plane);
```

Light Space Transform

- Projection matrix does not change per direction → re-use it for each of the 6 transformation matrices
- We do need a different view matrix per direction
- With glm::lookAt, create 6 view directions, each looking at a single direction of the cubemap (order: right, left, top, bottom, near and far)

Light Space Transform

```
std::vector<glm::mat4> shadowTransforms;
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3( 1.0f,  0.0f,  0.0f), glm::vec3(0.0f, -1.0f,  0.0f)));
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3(-1.0f,  0.0f,  0.0f), glm::vec3(0.0f, -1.0f,  0.0f)));
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3( 0.0f,  1.0f,  0.0f), glm::vec3(0.0f,  0.0f,  1.0f)));
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3( 0.0f, -1.0f,  0.0f), glm::vec3(0.0f,  0.0f, -1.0f)));
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3( 0.0f,  0.0f,  1.0f), glm::vec3(0.0f, -1.0f,  0.0f)));
shadowTransforms.push_back(shadowProj * glm::lookAt(lightPos, lightPos +
        glm::vec3( 0.0f,  0.0f, -1.0f), glm::vec3(0.0f, -1.0f,  0.0f)));
```

Depth Shaders

- To render depth values to a depth cubemap, need three shaders:
  - vertex shader
  - geometry shader
  - fragment shader

Depth Shaders

- Geometry shader responsible for transforming all world-space vertices to 6 different light spaces
- Vertex shader transforms vertices to world-space and directs them to the geometry shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(aPos, 1.0);
}
```

Depth Shaders

- Geometry shader takes as input 3 triangle vertices and a uniform array of light space transformation matrices
- Geometry shader is responsible for transforming the vertices to the light spaces

Depth Shaders

- Geometry shader has a built-in variable gl_Layer specifying which cubemap face to emit a primitive to
- When left alone the geometry shader just sends its primitives further down the pipeline as usual, but when we update this variable we can control to which cubemap face we render to for each primitive
- This works only when cubemap texture attached to active framebuffer

Depth Shaders

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices=18) out;
uniform mat4 shadowMatrices[6];
out vec4 FragPos; // FragPos from GS (output per emitvertex)
void main()
{
    for(int face = 0; face < 6; ++face)
    {
        gl_Layer = face; // built-in variable specifying face to render
        for(int i = 0; i < 3; ++i) // for each triangle's vertices
        {
            FragPos = gl_in[i].gl_Position;
            gl_Position = shadowMatrices[face] * FragPos;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```

Depth Shaders

- Last lecture, used an empty fragment shader and let OpenGL figure out the depth values of the depth map
- This time calculate own (linear) depth as the linear distance between each fragment position and the light source’s position
- Calculating own depth values makes the later shadow calculations a bit more intuitive

Depth Shaders

```
#version 330 core
in vec4 FragPos;

uniform vec3 lightPos;
uniform float far_plane;

void main()
{
    float lightDistance = length(FragPos.xyz - lightPos);

    // map to [0,1] range by dividing by far_plane
    lightDistance = lightDistance / far_plane;

    // write this as modified depth
    gl_FragDepth = lightDistance;
}
```

Omnidirectional Shadow Maps

Omnidirectional Shadow Maps

- To render the omnidirectional shadows, bind a cubemap texture instead of a 2D texture as the depth map and pass the light projection’s far plane variable to the shaders

```
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
// ... set uniforms
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
// ... bind other textures
renderScene();
```

Omnidirectional Shadow Maps

- Vertex and fragment shader are similar to the original shadow mapping shaders: but fragment shader no longer requires a fragment position in light space as we can now sample the depth values using a direction vector
- So vertex shader no longer needs to transform its position vectors to light space so we can exclude the FragPosLightSpace variable

Omnidirectional Shadow Maps

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
out vec2 TexCoords;
out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
} vs_out;
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = transpose(inverse(mat3(model))) * aNormal;
    vs_out.TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Omnidirectional Shadow Maps

- Fragment shader’s Blinn-Phong lighting code is exactly the same as we had before with a shadow multiplication at the end:

```
#version 330 core
out vec4 FragColor;
uniform samplerCube depthMap;
…

void main()
{
   …
   float shadow = ShadowCalculation(fs_in.FragPos);
   vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

   FragColor = vec4(lighting, 1.0);
}
```

Omnidirectional Shadow Maps

- Retrieve the depth of the cubemap
- Stored the depth as the linear distance between a fragment and the light position:

```
float ShadowCalculation(vec3 fragPos)
{
    // get vector between fragment position and light position
    vec3 fragToLight = fragPos - lightPos;
    // is the fragment to light vector to sample from the depth map
    float closestDepth = texture(depthMap, fragToLight).r;
    …
```

Omnidirectional Shadow Maps

- The closestDepth value is currently in the range $[0,1]$, transform it back to $[0, far\_plane]$ by multiplying it with far_plane:

```
…
// linear range [0,1], let's re-transform it back to original depth value
closestDepth *= far_plane;
```

Omnidirectional Shadow Maps

- Next, retrieve the depth value between the current fragment and the light source by taking the length of fragToLight:

```
…
// get linear depth as the length between the fragment and light position
float currentDepth = length(fragToLight);
```

Omnidirectional Shadow Maps

- Now, compare both depth values to determine whether the current fragment is in shadow
- Also include a shadow bias:

```
    …
    // test for shadows
    float bias = 0.05; // we use a much larger bias since depth is now in
                       // [near_plane, far_plane] range
    float shadow = currentDepth -  bias > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

Omnidirectional Shadow Maps

- Complete `ShadowCalculation`:

```
float ShadowCalculation(vec3 fragPos)
{
    vec3 fragToLight = fragPos - lightPos;
    float closestDepth = texture(depthMap, fragToLight).r;
    closestDepth *= far_plane;
    float currentDepth = length(fragToLight);
    float bias = 0.05;
    float shadow = currentDepth -  bias > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

Omnidirectional Shadow Maps

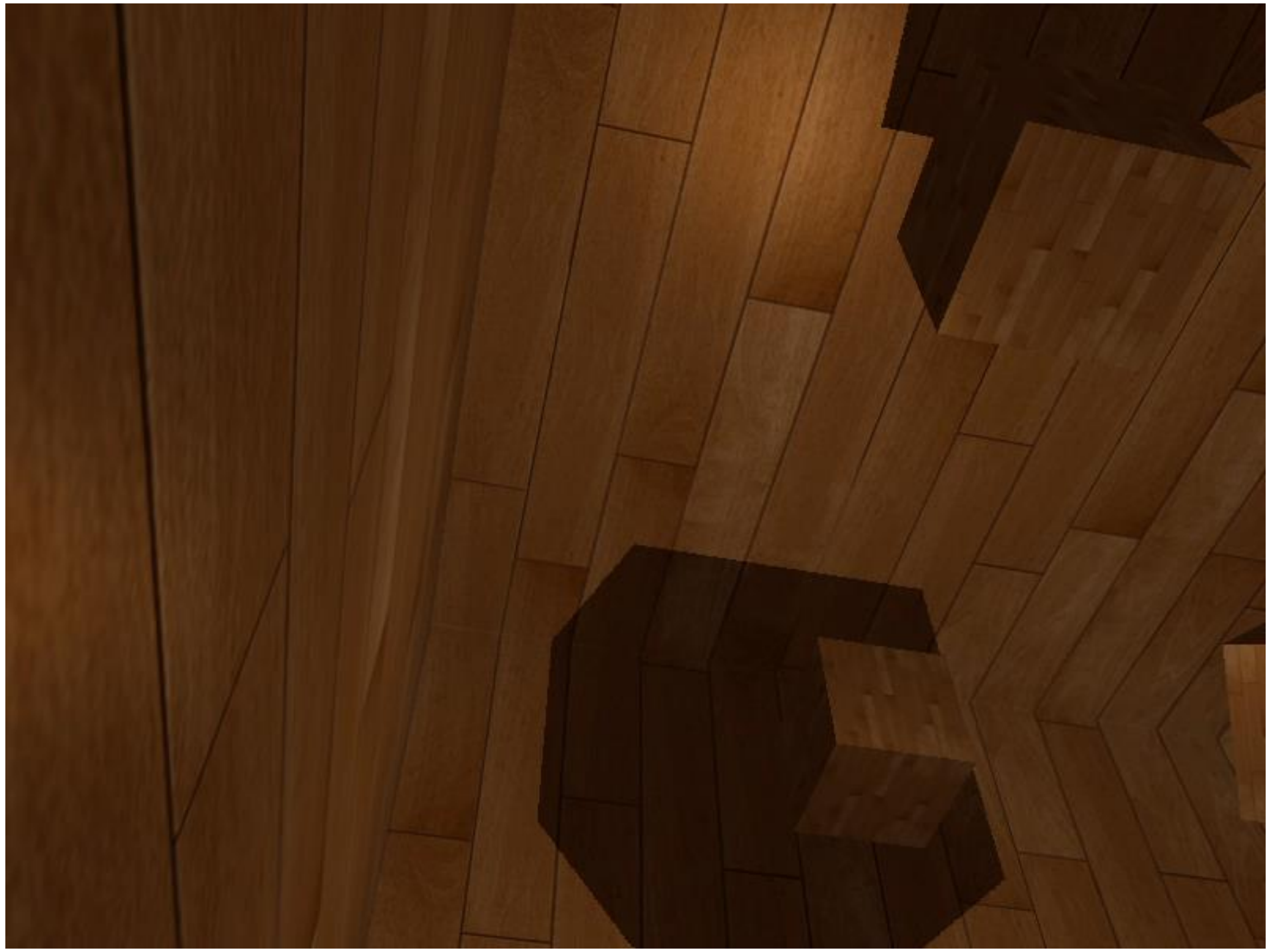
- Move the light a bit:

```
glm::vec3 lightPos(0.0f, 0.0f, 0.0f);
while (!glfwWindowShouldClose(window))
{
        float currentFrame = glfwGetTime();
         deltaTime = currentFrame - lastFrame;
         lastFrame = currentFrame;

         // move light position over time
         lightPos.z = sin(glfwGetTime() * 0.5) * 3.0;
…
```

F5…

- … point shadows



Visualizing Cubemap Depth Buffer

- In case something goes wrong: makes sense to do debugging with validating whether the depth map was built correctly
- Because do not have a 2D depth map texture anymore visualizing the depth map becomes a bit less obvious

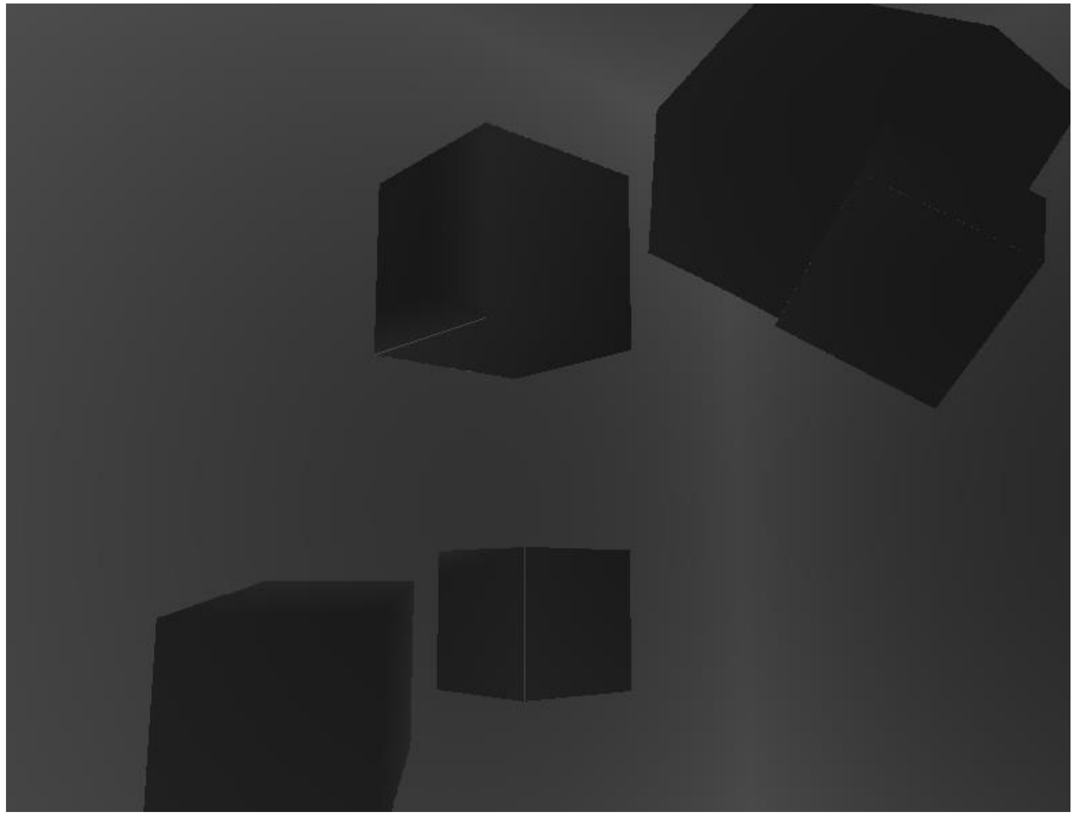
Visualizing Cubemap Depth Buffer

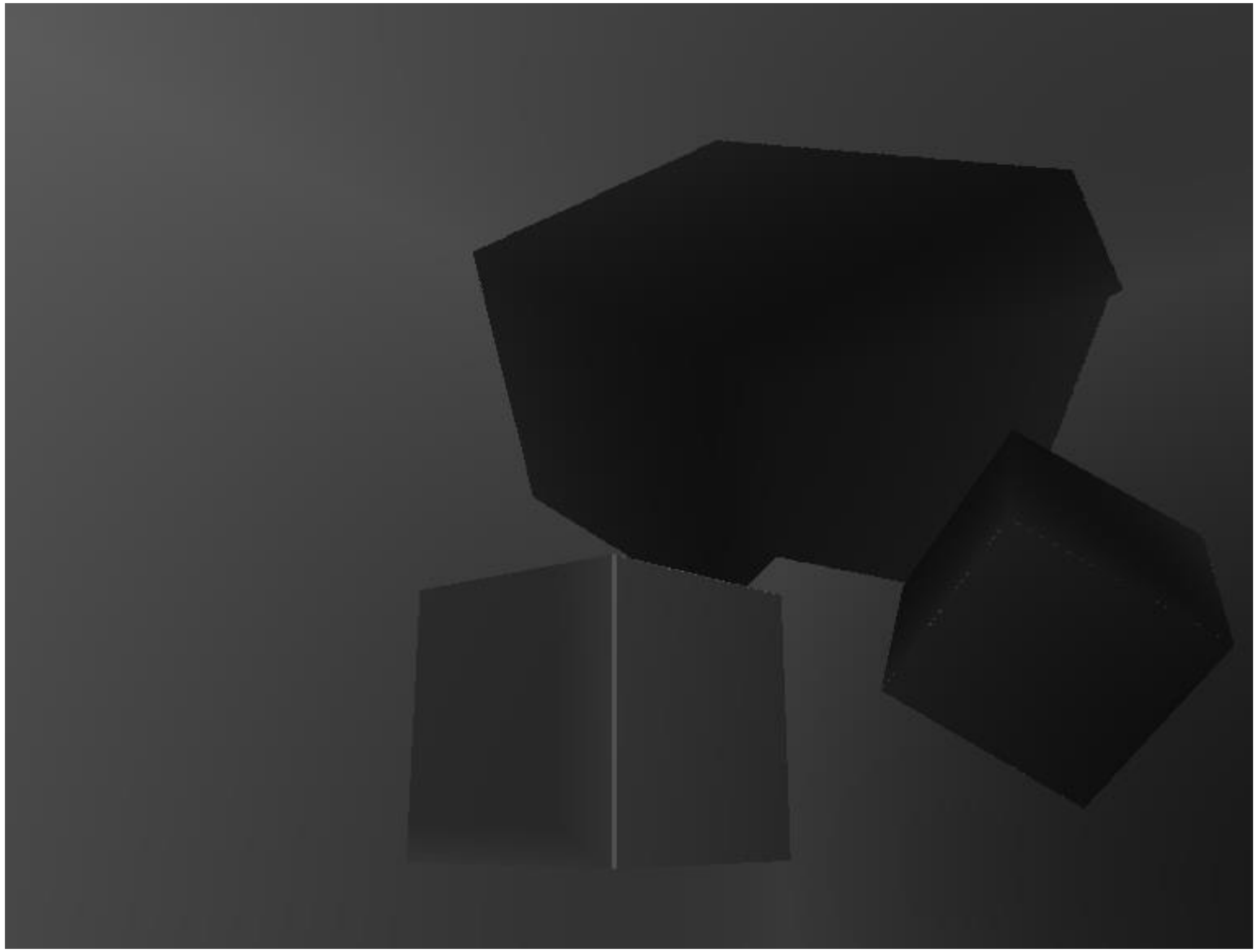
- To visualize the depth buffer, take the normalized ($[0,1]$) closestDepth variable in the ShadowCalculation function and display it as:

```
FragColor = vec4(vec3(closestDepth / far_plane), 1.0);
```

F5…

- … depth buffer

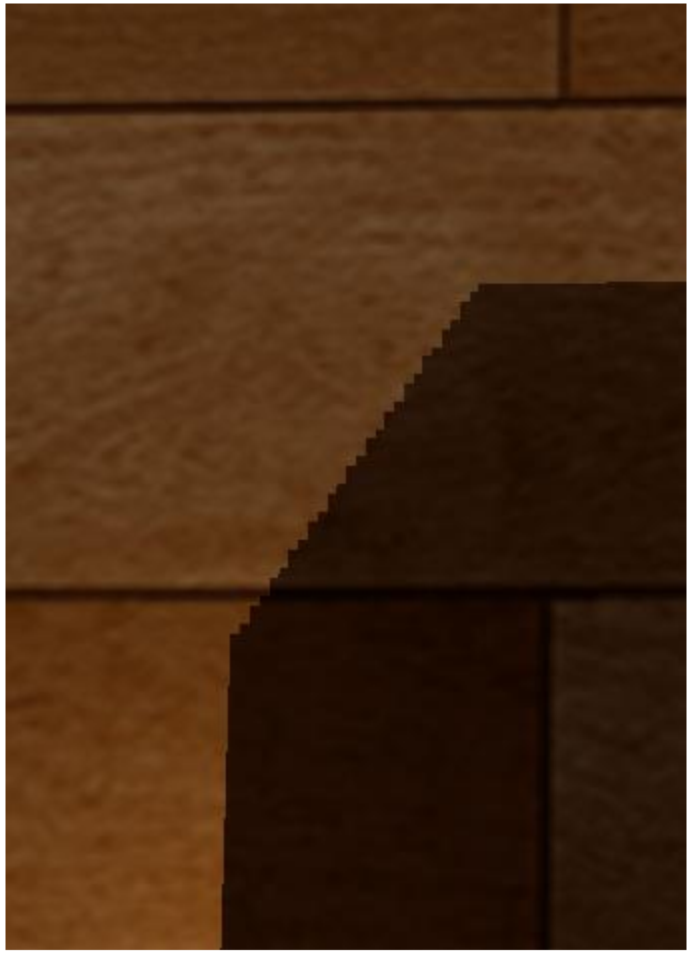




Percentage–closer Filtering (PCF)

Introduction

- Omnidirectional shadow maps based on same principles of traditional shadow mapping → has the same resolution dependent artifacts
- Zoom in closely, see jagged edges
- Percentage-closer filtering or PCF smooth out jagged edges by filtering multiple samples around the fragment position and average the results



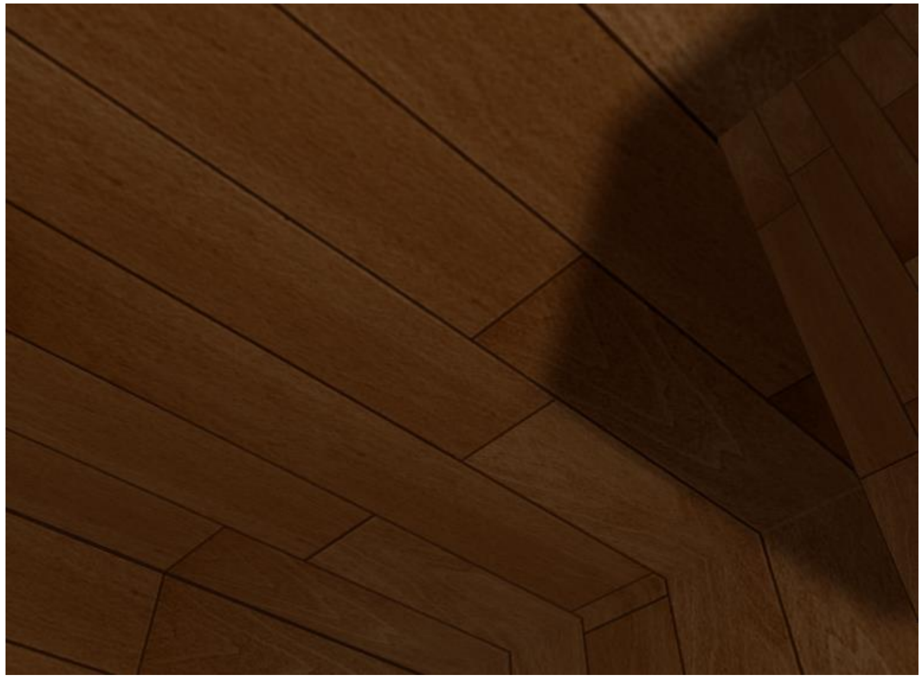
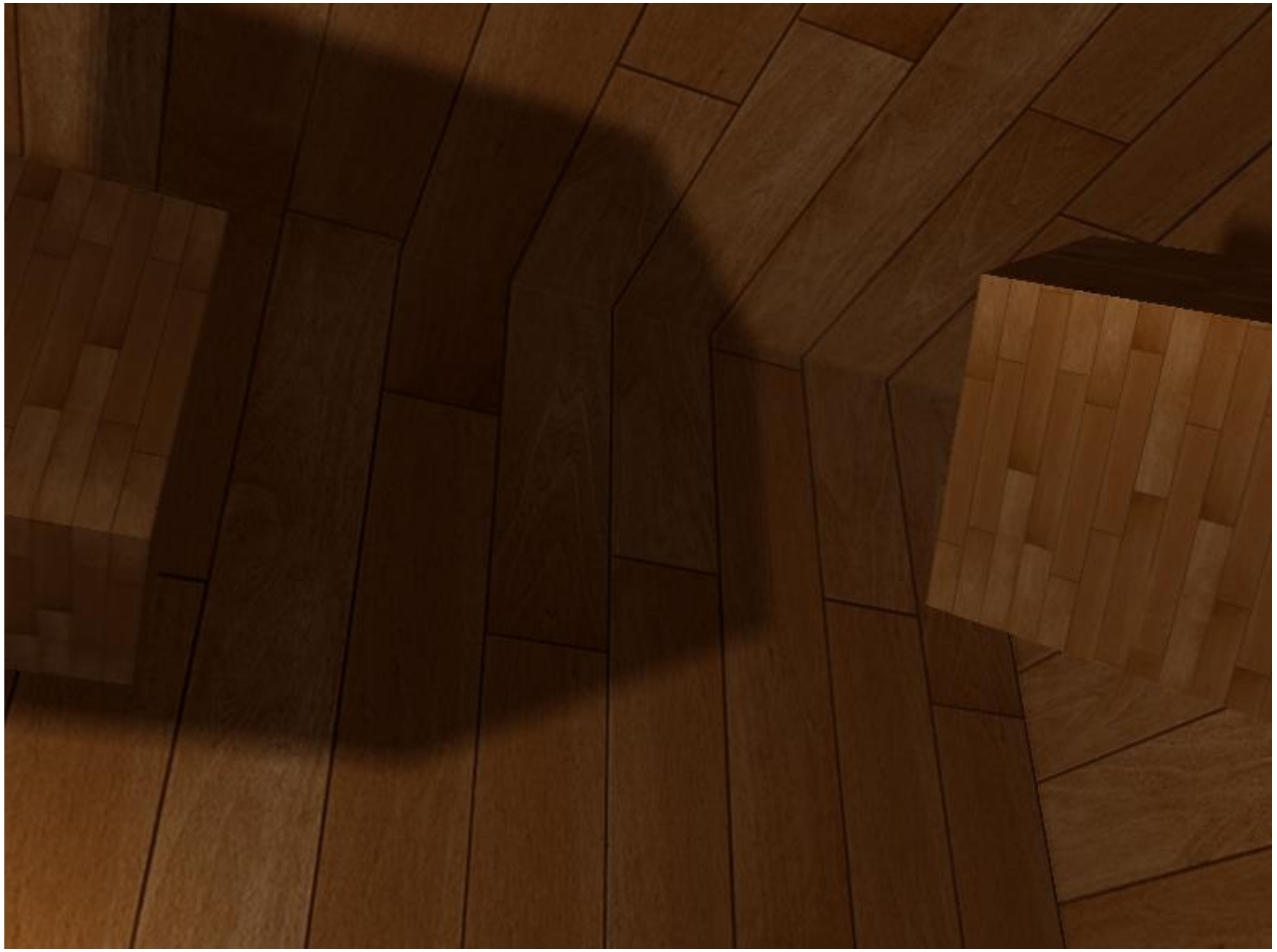
PCF

- Same simple PCF filter and add a third dimension (need 3D direction vectors to sample from a cubemap):

```
float shadow = 0.0;
float bias = 0.05;
float samples = 4.0;
float offset = 0.1;
for(float x = -offset; x < offset; x += offset / (samples * 0.5)){
   for(float y = -offset; y < offset; y += offset / (samples * 0.5)){
      for(float z = -offset; z < offset; z += offset / (samples * 0.5))
      {
         float closestDepth = texture(depthMap, fragToLight + vec3(x, y, z)).r;
         closestDepth *= far_plane;   // Undo mapping [0;1]
         if(currentDepth - bias > closestDepth)
            shadow += 1.0;
      }
   }
}
shadow /= (samples * samples * samples);
```

F5…

- …soft

PCF

- Most samples redundant (sample close to the original direction vector), more sense to only sample in perpendicular directions
- No (easy) way to figure out which sub-directions are redundant this becomes difficult
- One trick: take an array of offset directions that are all roughly separable, e.g., each points in completely different directions → reducing number of sub-directions that are close together

PCF

- Below we have such an array of a maximum of 20 offset directions:

```
vec3 gridSamplingDisk[20] = vec3[]
(
   vec3(1, 1,  1), vec3( 1, -1,  1), vec3(-1, -1,  1), vec3(-1, 1,  1),
   vec3(1, 1, -1), vec3( 1, -1, -1), vec3(-1, -1, -1), vec3(-1, 1, -1),
   vec3(1, 1,  0), vec3( 1, -1,  0), vec3(-1, -1,  0), vec3(-1, 1,  0),
   vec3(1, 0,  1), vec3(-1,  0,  1), vec3( 1,  0, -1), vec3(-1, 0, -1),
   vec3(0, 1,  1), vec3( 0, -1,  1), vec3( 0, -1, -1), vec3( 0, 1, -1)
);
```

PCF

- Then, adapt the PCF algorithm to take a fixed amount of samples from sampleOffsetDirections and use these to sample the cubemap
- The advantage is that we need a lot less samples to get visually similar results to the first PCF algorithm

PCF

```
float shadow = 0.0;
float bias = 0.15;
int samples = 20;
float viewDistance = length(viewPos - fragPos);
float diskRadius = 0.05;
for(int i = 0; i < samples; ++i)
{
   float closestDepth = texture(depthMap, fragToLight + gridSamplingDisk[i] *
                                                                  diskRadius).r;
   closestDepth *= far_plane;   // undo mapping [0;1]
   if(currentDepth - bias > closestDepth)
      shadow += 1.0;
}
shadow /= float(samples);
```

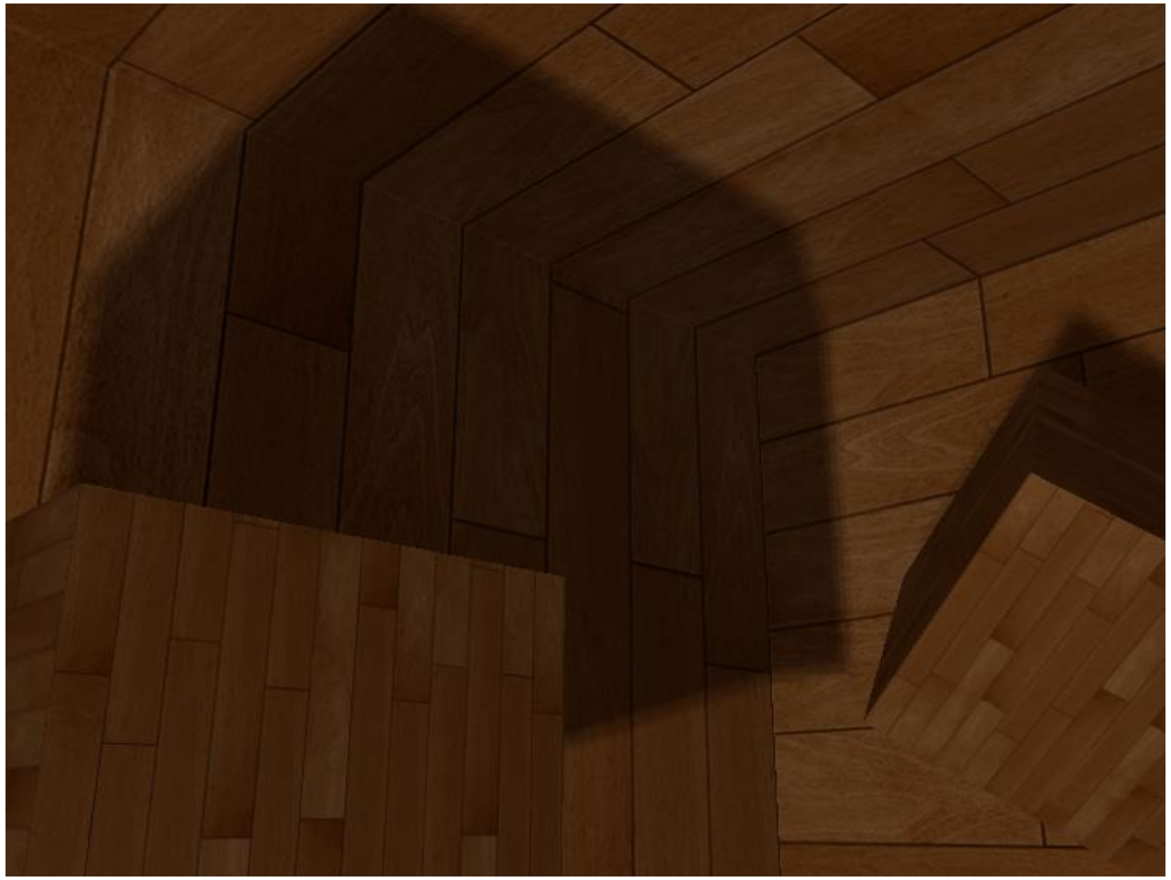
PCF

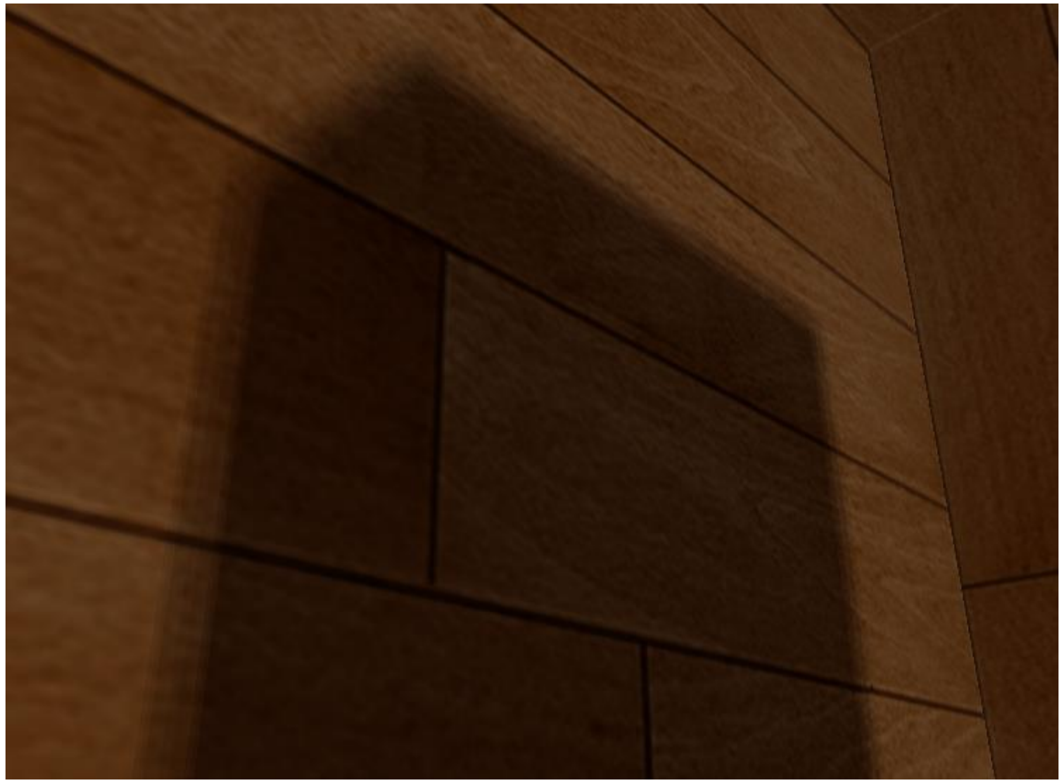
- Another trick: change the diskRadius based on how far the viewer is away from a fragment (increase the offset radius by the distance to the viewer, making the shadows softer when far away and sharper when close by)

```
float diskRadius = (1.0 + (viewDistance / far_plane)) / 25.0;
```

F5…

- … soft





Questions???