Computer Graphics II – Shadow Mapping

Kai Lawonn

Introduction

- Shadows result of the absence of light due to occlusion
- Shadows add realism to a lighted scene and make it easier for a viewer to observe spatial relationships between objects
- They give a greater sense of depth to our scene and objects

Introduction

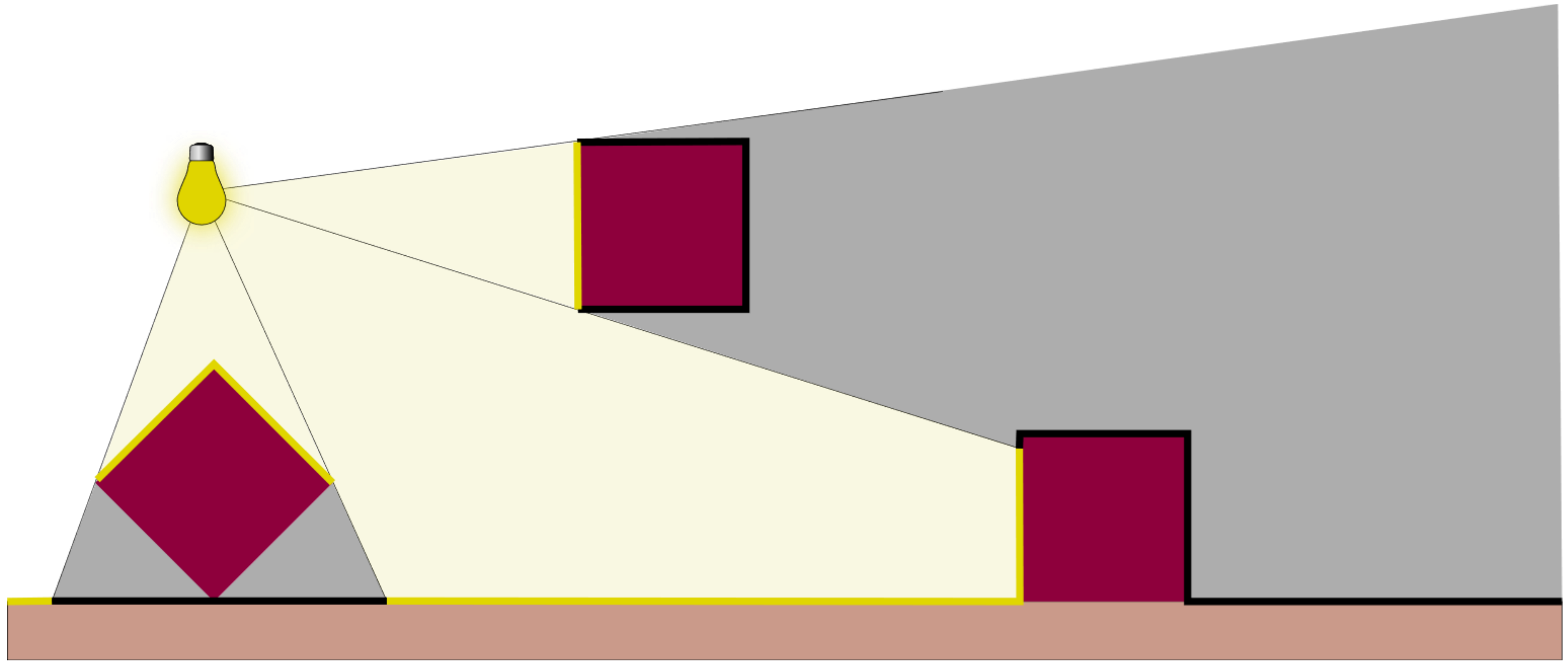
- Shadows are tricky to implement
- There are several good shadow approximation techniques, but they all have their little quirks and annoyances
- One technique used by most videogames that gives decent results and is relatively easy to implement is shadow mapping
- Shadow mapping is not too difficult to understand, doesn’t cost too much in performance and is quite easily extended into more advanced algorithms

Shadow Mapping

- Idea: we render the scene from the light, everything seen is lit and everything else must be in shadow
- Imagine a floor with a large box between itself and a light source
- Light source will see this box and not the floor, thus, floor should be in shadow

Shadow Mapping

- Yellow lines represent the fragments that the light source can see
- Occluded fragments are shown as black lines (shadowed)



Shadow Mapping

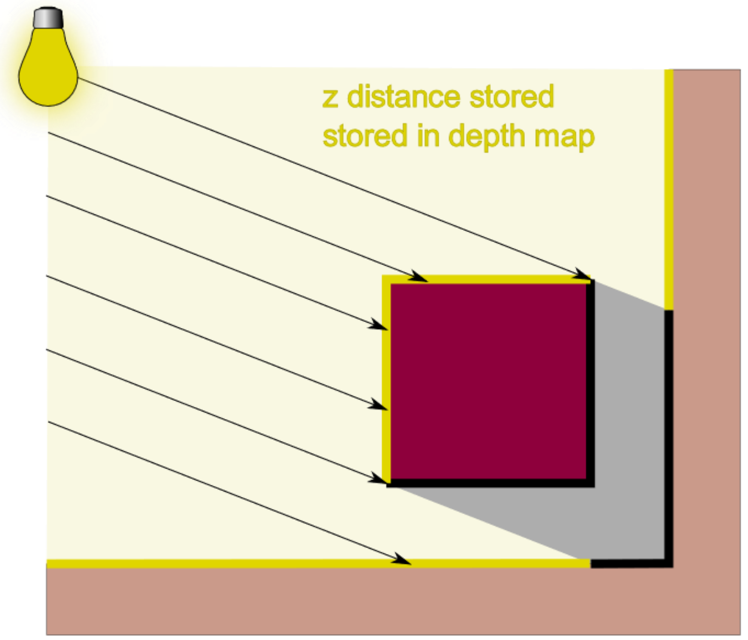
- Get a point on the ray where it first hit an object and compare this closest point to other points on this ray
- Then basic test to see if a test point’s ray position is further down the ray than the closest point if so, test point must be in shadow
- Iterating through thousands of light rays is extremely inefficient → real-time rendering?
- Do similar, but without casting light rays, using the depth buffer

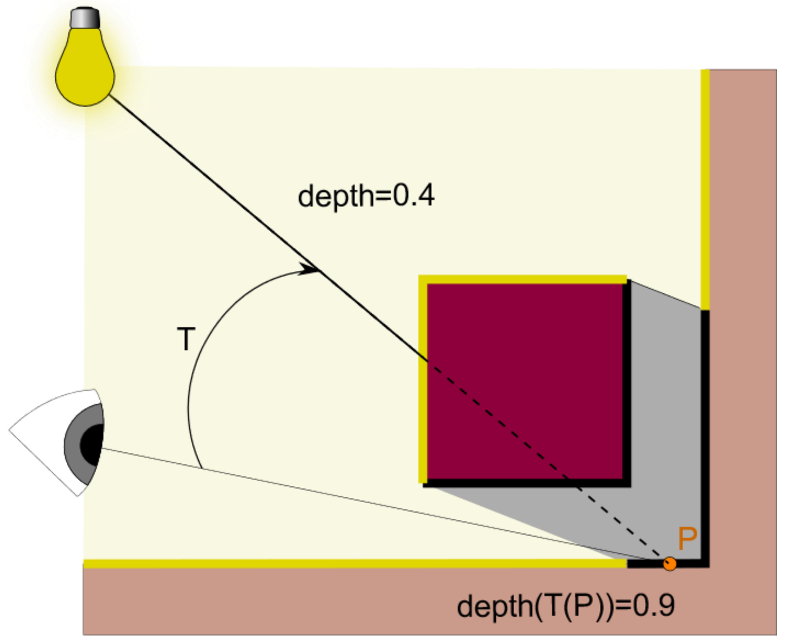
Shadow Mapping

- Value in the depth buffer corresponds to the depth of a fragment clamped to [0,1] from the camera’s point of view
- Render the scene from the light’s perspective and store the resulting depth values in a texture
- Then, can sample the closest depth values as seen from the light’s perspective
- Depth values show the first fragment visible from the light’s perspective
- We store all these depth values in a texture that we call a depth map or shadow map

Shadow Mapping

- Left: directional light source (light rays parallel) casting a shadow
- Create depth map by rendering the scene (from the light) using a view and projection matrix specific to that light source
- This projection and view matrix together form a transformation that transforms any 3D position to the light's visible coordinate space



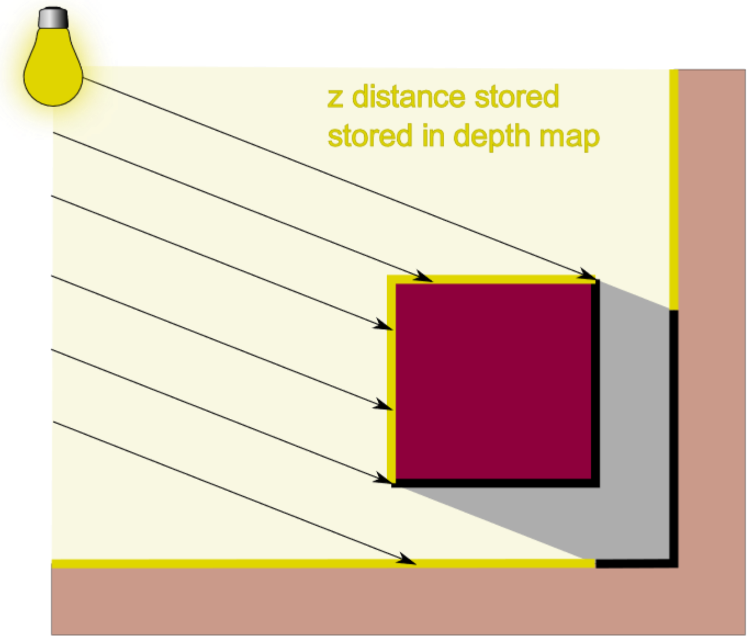


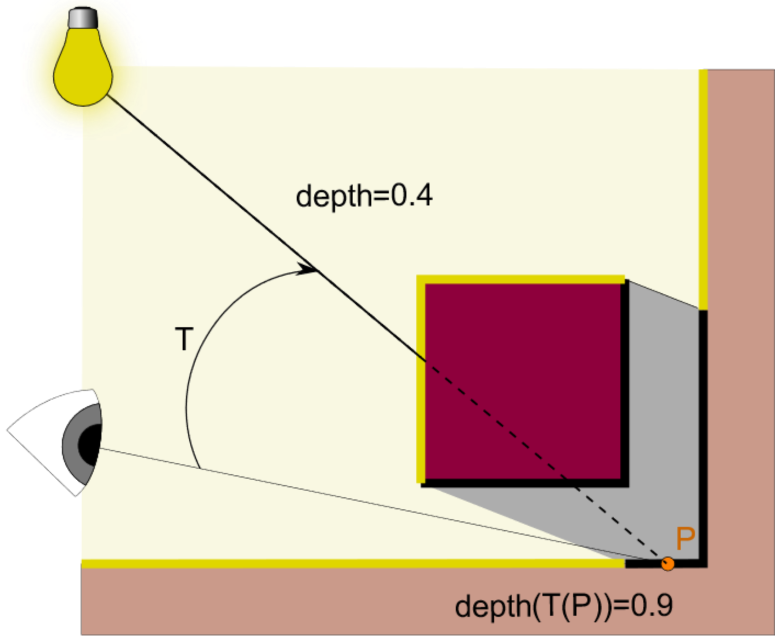
Directional light has not a position (modelled infinitely far away)

For shadow mapping, need to render the scene from the light → from a position somewhere along the lines of the light direction

Shadow Mapping

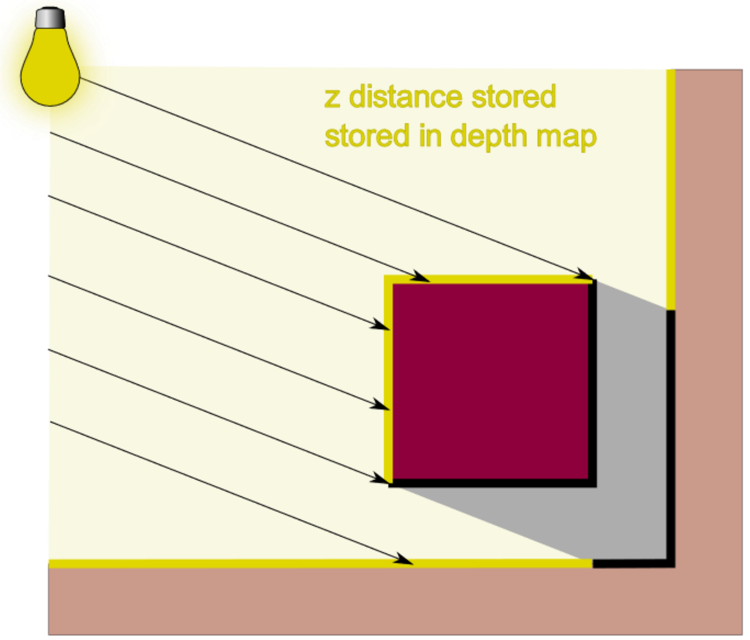
- Right: same directional light and the viewer
- Render a fragment (orange), have to determine whether shadowed
- Transform point to the light's coordinate space

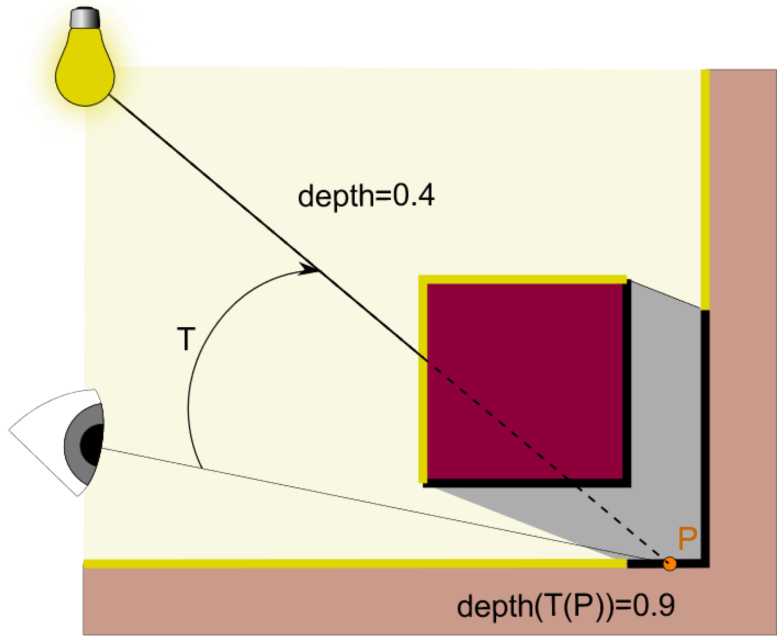




Shadow Mapping

- From the light its z coordinate is 0.9
- Can also index the depth map at this point, which is 0.4
- Depth map returned a depth smaller than the depth at point P → P is occluded and thus in shadow





Shadow Mapping

- Shadow mapping consists of two passes:
- 1st render the depth map
- 2nd render the scene as normal and use the generated depth map to calculate whether fragments are in shadow

Depth Map

Depth Map

- 1st pass requires a depth map
- The depth map is the depth texture from the light's perspective
- Need to store the rendered result of a scene into a texture → need framebuffers again

Depth Map

- First, create a framebuffer object for rendering the depth map:

```
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
```

Depth Map

- Next, create a 2D texture for the framebuffer's depth buffer:

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH,
              SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Depth Map

- Generated depth texture can attach as framebuffer’s depth buffer:

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
                                                           depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Depth Map

- 1st: generating the depth map
- The complete rendering stage of both passes will looks a bit like this:

```
// 1. first render to depth map
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 map)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

Light Space Transform

- Unknown: ConfigureShaderAndMatrices function
- 2nd pass: projection and view matrices are set and the relevant model matrices per object
- However, in 1st pass use a different projection and view matrix to render the scene from the light's point of view

Light Space Transform

- Modelling a directional light source → light rays are parallel
- Use an orthographic projection matrix for the light source where there is no perspective deform:

```
float near_plane = 1.0f, far_plane = 7.5f;
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane,
                                                                    far_plane);
```

- Projection matrix indirectly determines the range of what is visible (clipped)
- Make sure projection frustum contains the objects in the depth map

Light Space Transform

- To create a view matrix, use glm::lookAt function; this time with the light source's position looking at the scene's center:

```
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
```

Light Space Transform

- These, gives a light space transformation matrix (transforms each world-space vector into the space as visible from the light source):

```
lightSpaceMatrix = lightProjection * lightView;
```

- lightSpaceMatrix is the transformation matrix T (recall image)
- Can render the scene as usual as long as we give the shader the light-space equivalents of the projection and view matrices
- To save performance we’re going to use a different, but much simpler shader for rendering to the depth map

Render to Depth Map

- Want to use a simple shader that only transforms the vertices to light space
- For such a simple shader, use the following vertex shader:

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

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

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

Render to Depth Map

- Have no color buffer, so can simply use an empty fragment shader:

```
#version 330 core

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

Render to Depth Map

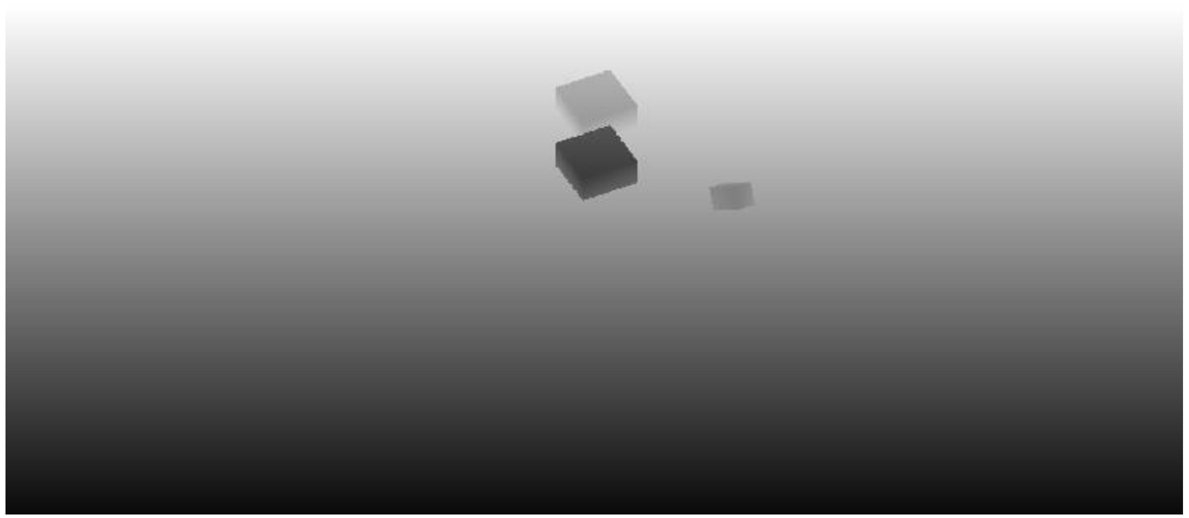
- Rendering the depth buffer becomes:

```
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
renderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Render to Depth Map

- Is a filled depth buffer holding the closest depth of each visible fragment from the light’s perspective
- By projecting this texture onto a 2D quad that fills the screen we get something like this:



Render to Depth Map

- Rendering the depth map onto a quad (fragment shader):

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D depthMap;

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    FragColor = vec4(vec3(depthValue), 1.0);
}
```

Rendering Shadows

Rendering Shadows

- With a generated depth map, can start generating shadows
- Check if fragment is in shadow is executed in the fragment shader, but we do the light-space transformation in the vertex shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} vs_out;
…
```

Rendering Shadows

```
…
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 lightSpaceMatrix;

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

Rendering Shadows

- Fragment shader to render the scene uses the Blinn-Phong lighting
- Within the fragment shader calculate a shadow value that is either 1.0 when the fragment is in shadow or 0.0 when not in shadow
- Diffuse and specular colors are multiplied by this shadow component
- Shadows are rarely completely dark due to light scattering → leave the ambient color out of the shadow multiplications

Rendering Shadows

```
#version 330 core
out vec4 FragColor;

in VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoords;
    vec4 FragPosLightSpace;
} fs_in;

uniform sampler2D diffuseTexture;
uniform sampler2D shadowMap;
uniform vec3 lightPos;
uniform vec3 viewPos;

float ShadowCalculation(vec4 fragPosLightSpace)
{
  …
}

void main()
{
    // Blinn-Phong
    …
    // calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;
    FragColor = vec4(lighting, 1.0);
}
```

Rendering Shadows

- First, transform light-space fragment position in clip-space to NDCs
- When output a clip-space vertex position to gl_Position (vertex shader), OpenGL does a perspective divide (transform clip-space coordinates in $[-w, w]$ to $[-1,1]$ by dividing w)

Rendering Shadows

- Clip-space FragPosLightSpace is not passed to the fragment shader via gl_Position → manual perspective division:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
   …
}
```

Rendering Shadows

When using an orthographic projection matrix the w component of a vertex remains untouched so this step is actually quite meaningless.

However, it is necessary when using perspective projection so keeping this line ensures it works with both projection matrices.

Rendering Shadows

- Depth from depth map is in $[0,1]$ and we want to use projCoords to sample from the depth map, so we transform NDC to $[0,1]$:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    …
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    …
}
```

Rendering Shadows

- With these, can sample the depth map
- This gives us the closest depth from the light’s point of view:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    …
    // get closest depth value from light's perspective (using [0,1] range
    // fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    …
}
```

Rendering Shadows

- To get the current depth, retrieve the projected vector’s z coordinate which equals the depth of the fragment from the light’s perspective:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
   …
   // get depth of current fragment from light's perspective
   float currentDepth = projCoords.z;
   …
}
```

Rendering Shadows

- Actual comparison is a check whether currentDepth is higher than closestDepth if so, the fragment is in shadow:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    …
    // check whether current frag pos is in shadow
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}
```

Rendering Shadows

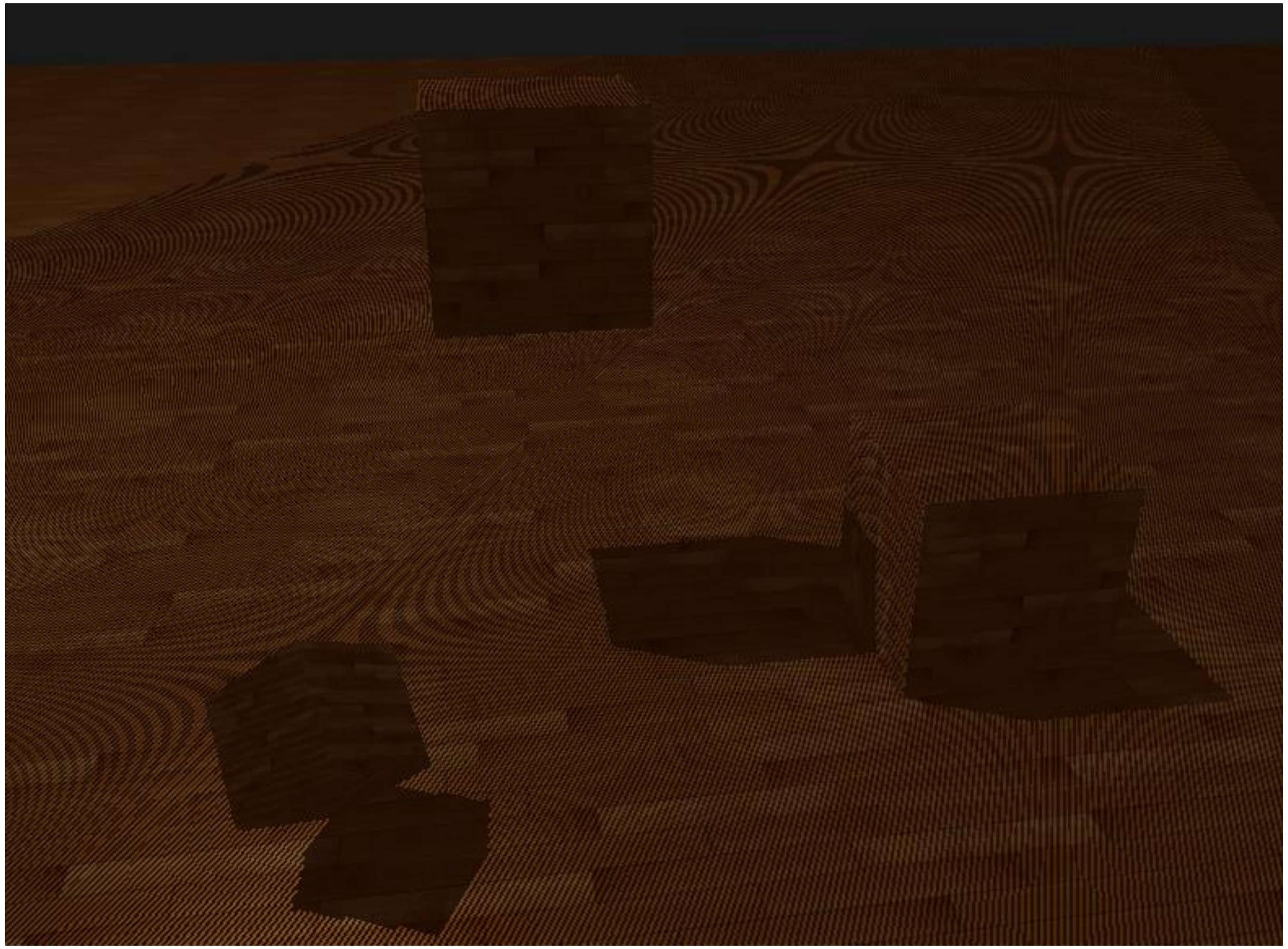
- The complete ShadowCalculation function then becomes:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range
    // fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // check whether current frag pos is in shadow
    float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;

    return shadow;
}
```

F5…

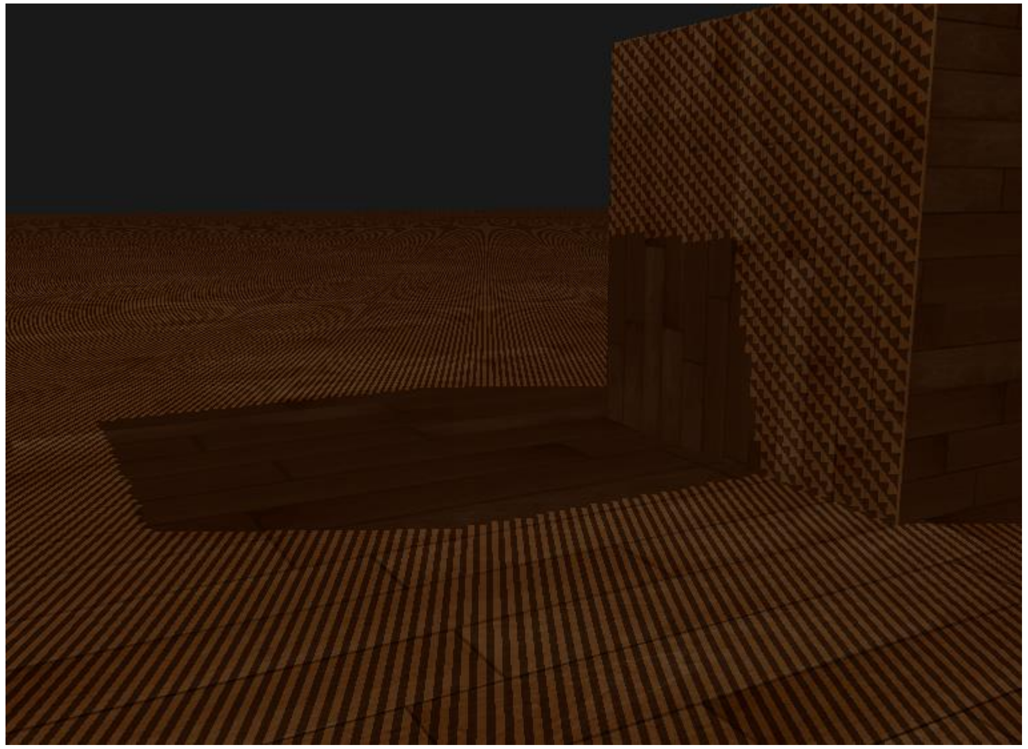
- … there will be shadows



Improving Shadow Maps

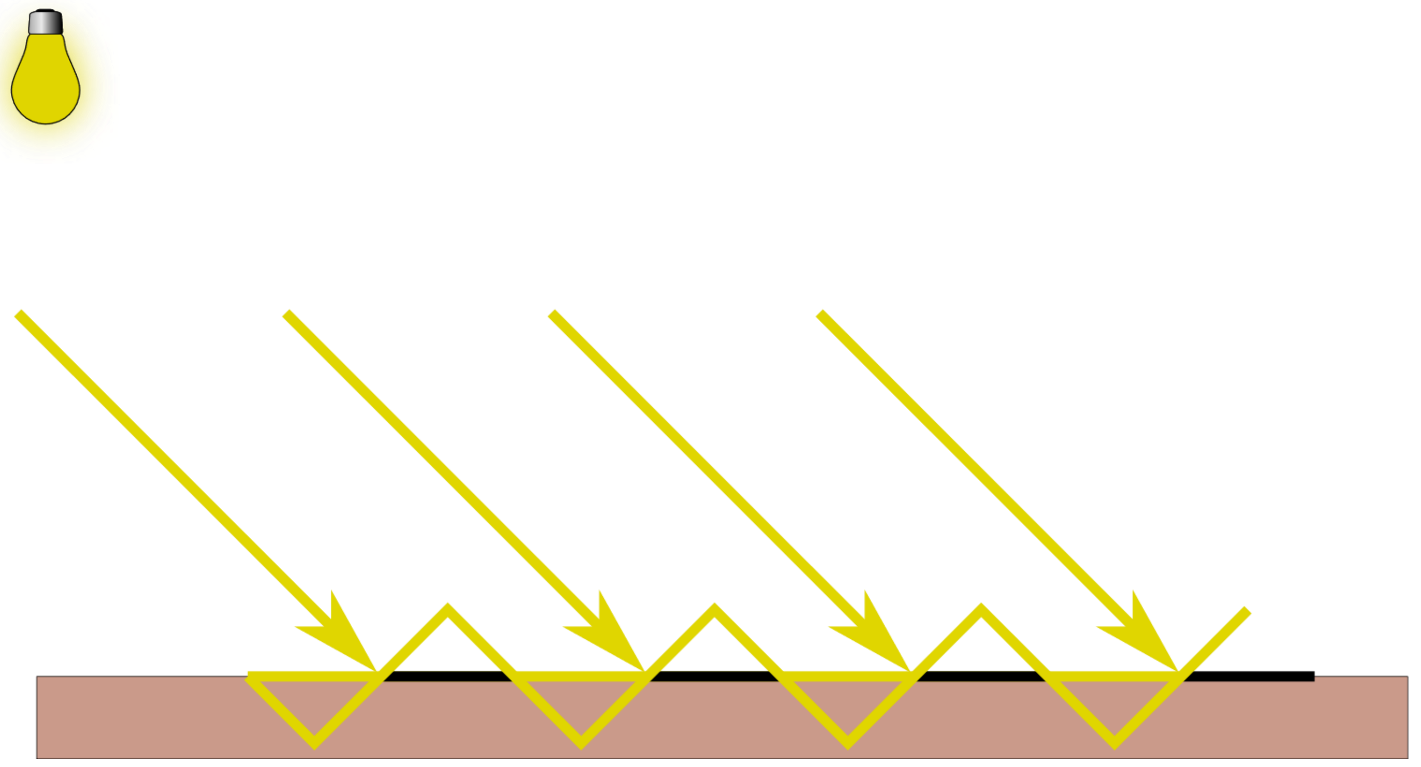
Introduction

- Something is wrong from the previous image
- A closer zoom shows us a very obvious Moiré-like pattern:



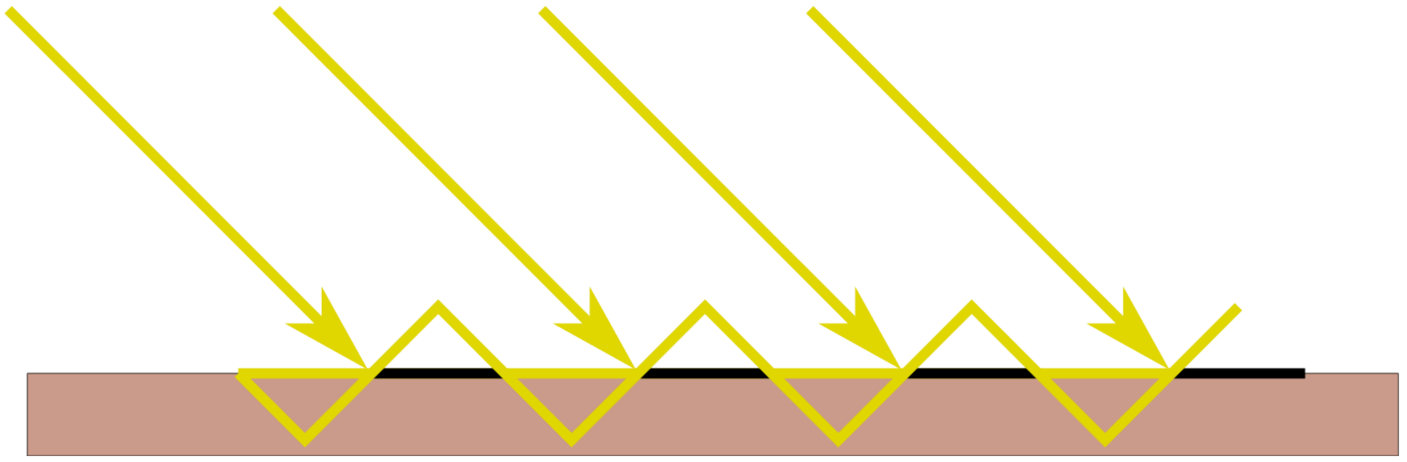
Shadow Acne

- Large part of the floor quad rendered with alternating black lines
- This shadow mapping artifact is called shadow acne:



Shadow Acne

- Shadow map limited by resolution, multiple fragments can sample same value from depth map when far away from the light source
- Image shows floor, each tilted panel represents a single texel of the depth map
- As you can see, several fragments sample the same depth sample

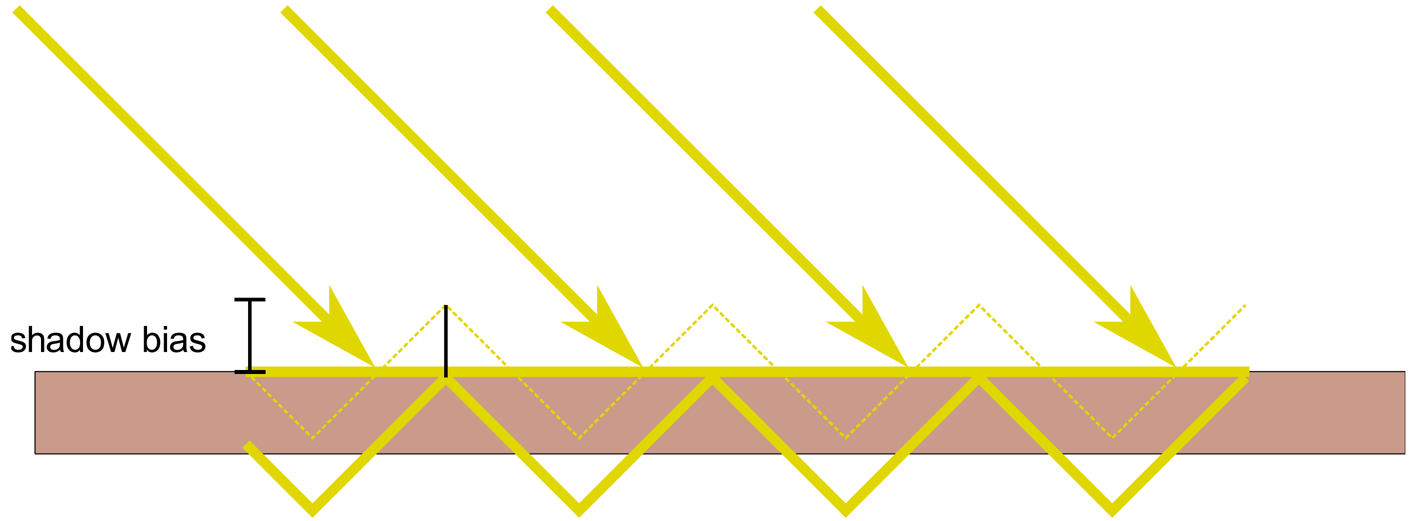


Shadow Acne

- Becomes an issue when the light source looks at an angle towards the surface as in that case the depth map is also rendered from an angle
- Several fragments access the same tilted depth texel while some are above and some below the floor → shadow discrepancy
- Then, some fragments are in shadow and some are not

Improving Shadow Maps

- Solve this issue with a small little hack called a shadow bias
- Simply offset the depth of the surface (or the shadow map) by a small bias amount such that fragments are not incorrectly considered below the surface

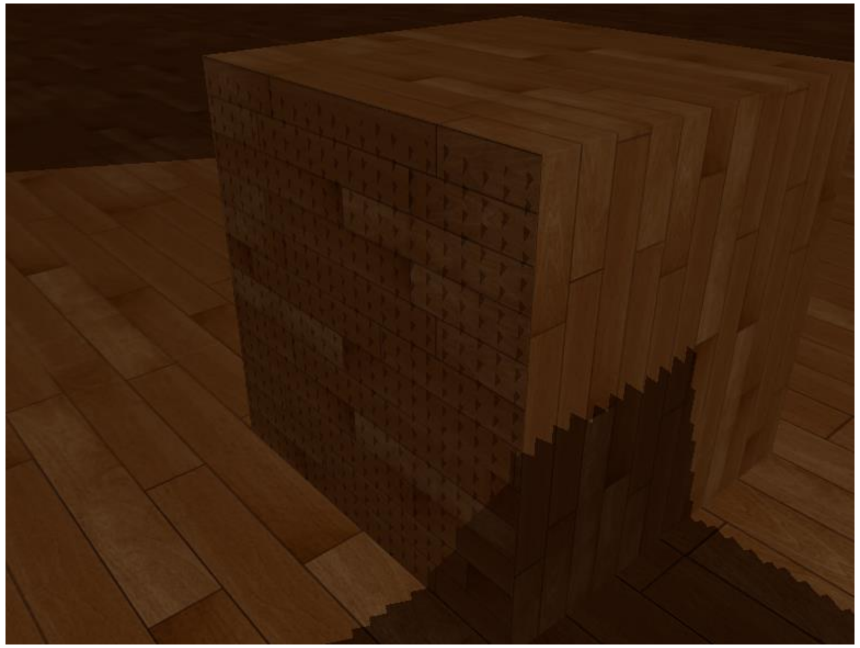


Improving Shadow Maps

- With the bias, get a depth smaller than the surface's depth → entire surface is correctly lit without any shadows
- We can implement such a bias as follows:

```
float bias = 0.005;
float shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

Improving Shadow Maps



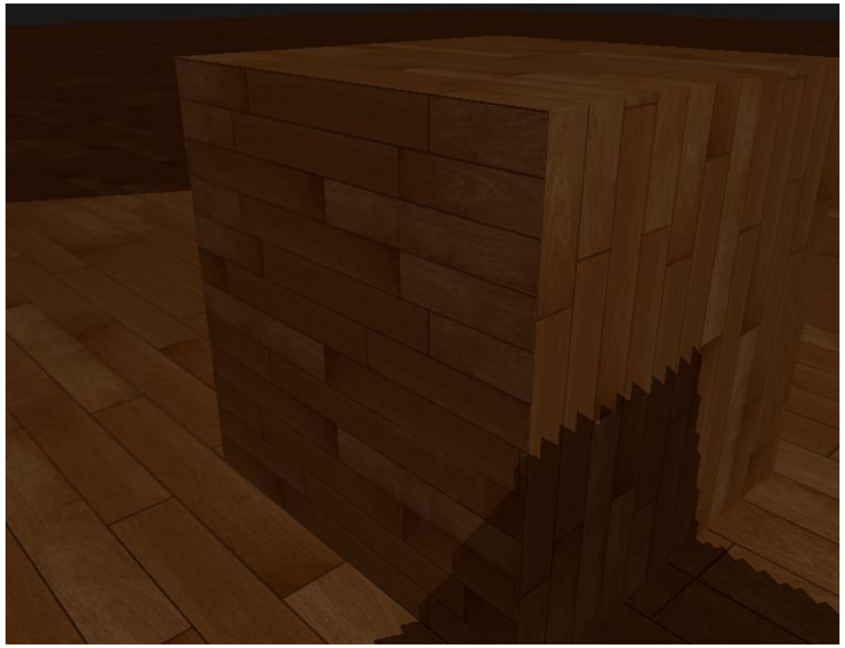
- Bias of 0.005 almost solves the issues, but some surfaces that have a steep angle to the light source might still produce shadow acne
- Better: change the amount of bias based on the surface angle towards the light:

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
```

- Maximum bias of 0.05 and a minimum of 0.005 (based on normal and light direction)

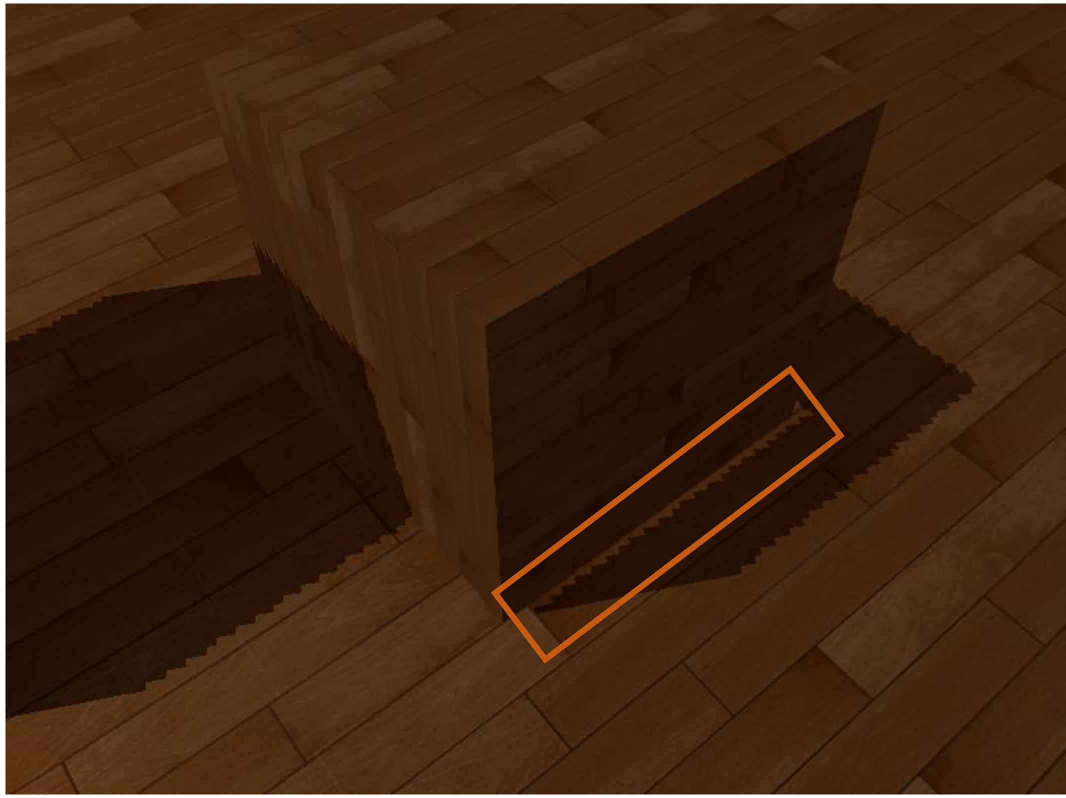
F5…

- ... much better results
- Choosing correct bias value(s) requires some tweaking (different in each scene)
- Mostly, incrementing the bias until all acne is removed



Peter Panning

- A disadvantage of using a shadow bias is that you're applying an offset to the actual depth of objects
- Bias might become large enough to see a visible offset of shadows compared to the actual object locations:

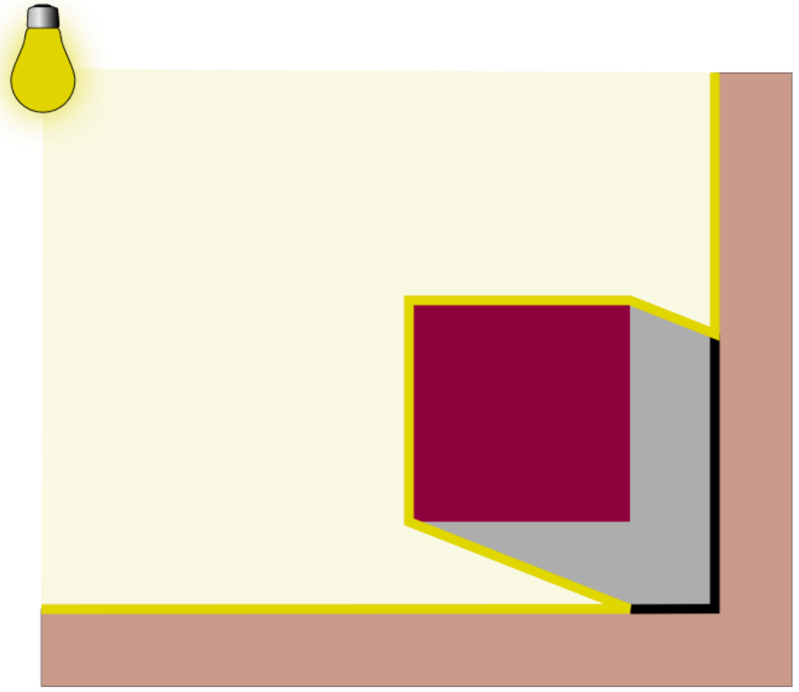


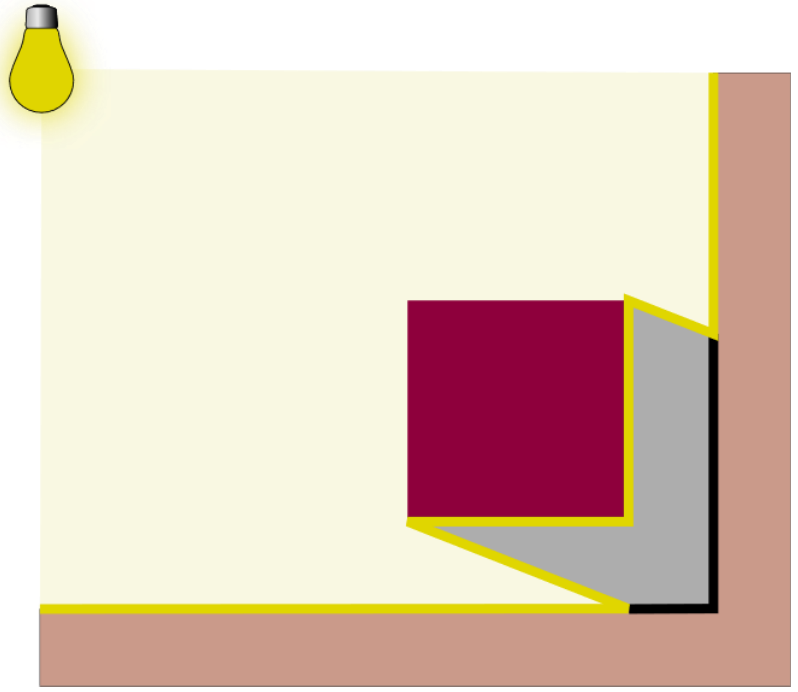
Peter Panning

- This artifact is called Peter panning (objects slightly detached from their shadows)
- May happen, when OpenGL culls back-faces
- Use a trick to solve most peter panning issue by using front face culling when rendering the depth map

Peter Panning

- Only need depth values for the depth map (not matter for solid objects whether take the depth of front faces or back faces)
- Using their back face depths does not give wrong results as it does not matter having shadows inside objects; cannot see there anyways





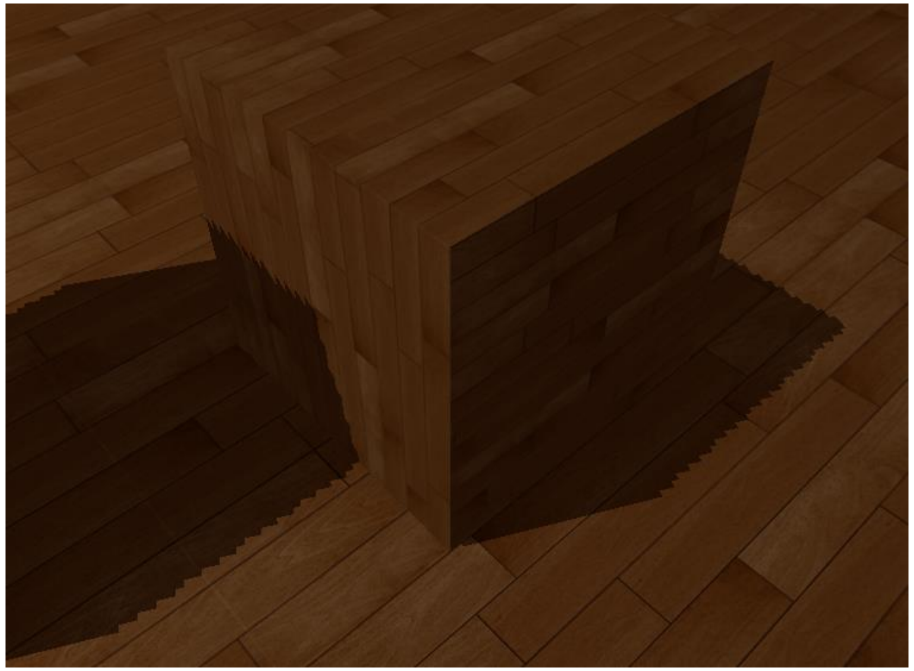
Peter Panning

- To mostly fix Peter panning we cull front faces
- Note that you need to enable GL_CULL_FACE first

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
…
renderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glDisable(GL_CULL_FACE);
```

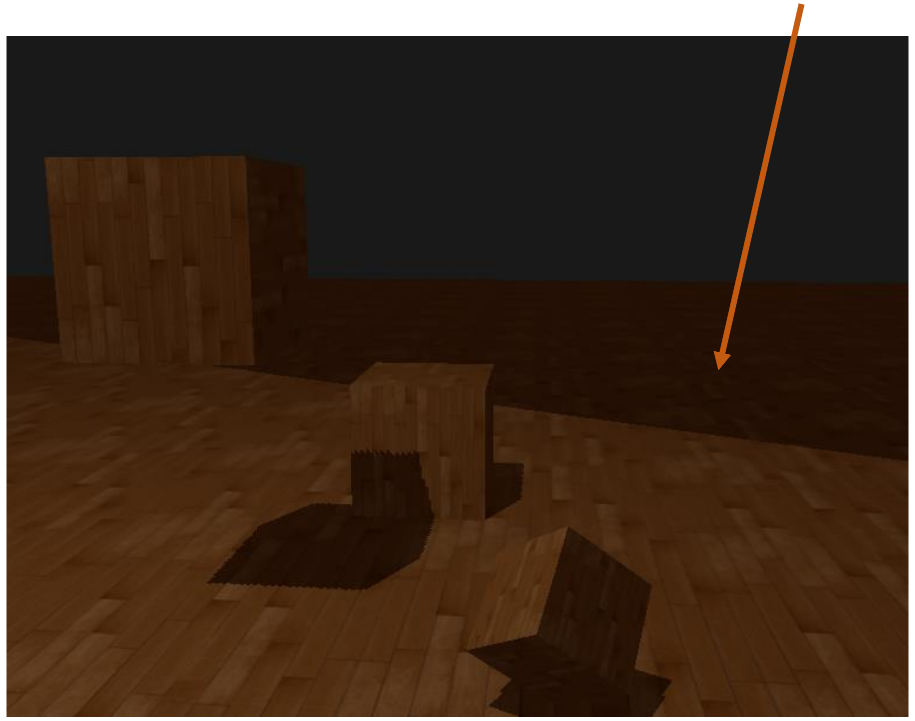
Peter Panning

- Solves the issue, only for solid objects (have an inside without openings)
- Works perfectly fine on cubes, but won’t on the floor (culling front face removes the floor)
- Floor is a single plane and would completely be culled
- Solve Peter panning with this, care has to be taken to only cull the front faces of objects
- Also objects close to the shadow receiver (like the distant cube) might still give incorrect results
- Care should be taken to use front face culling on objects where it makes sense
- However, with normal bias values one can generally avoid peter panning



Over Sampling

- Another problem is that some regions outside the light’s visible frustum are in shadow
- Projected coordinates outside the light’s frustum higher than 1.0 → will sample the depth texture outside its default range of $[0,1]$
- Based on wrapping method get incorrect depth results not based on the real depth values from the light source



Over Sampling

- Reason: we set the depth map’s wrapping options to GL_REPEAT
- Better: all coordinates outside the depth map’s range have a depth of 1.0, these coordinates will never be in shadow (no object have a depth larger than 1.0)
- Achieve this by storing a border color and set the depth map’s texture wrap options to GL_CLAMP_TO_BORDER:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

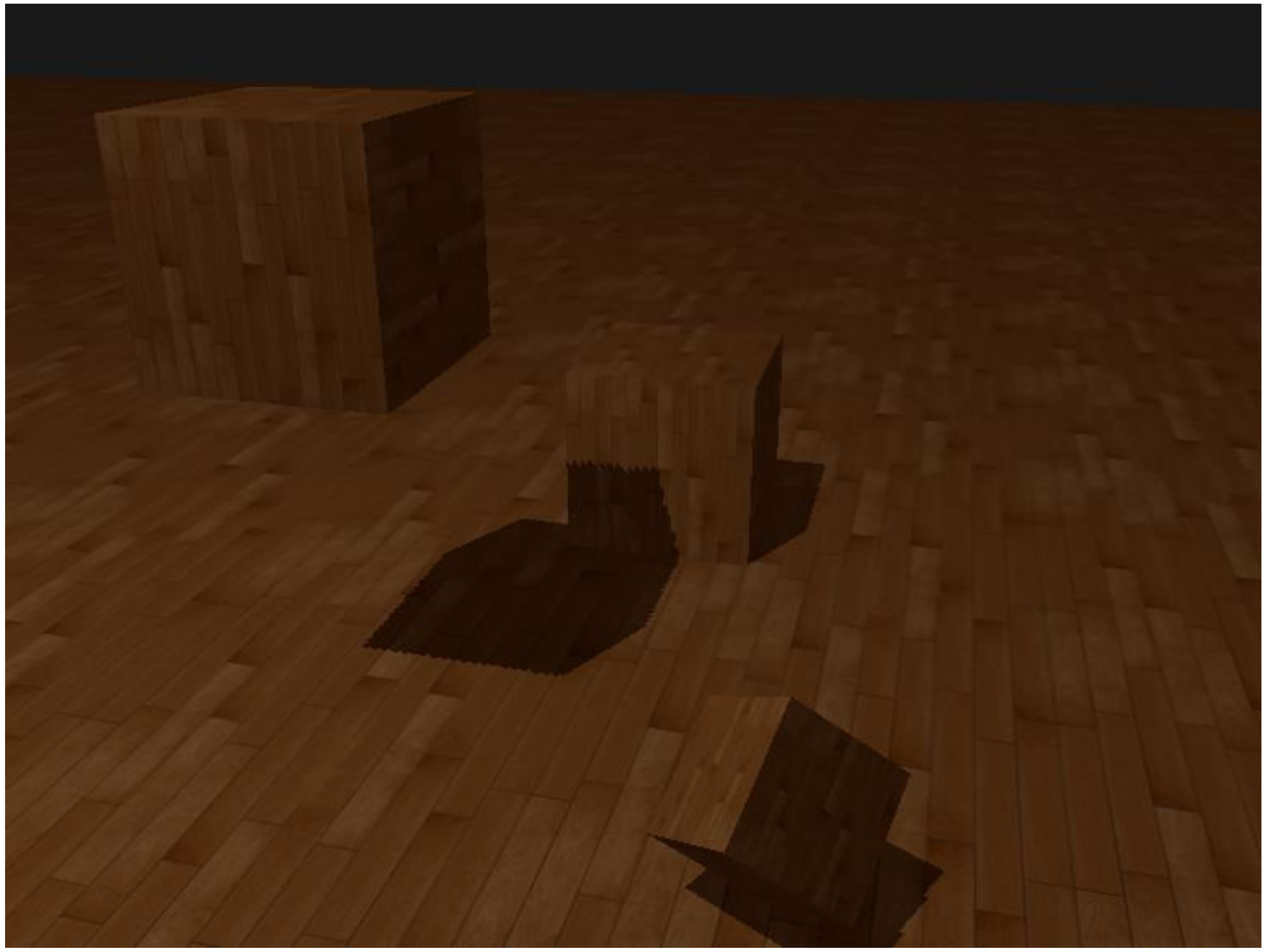
Over Sampling

- Projected coordinate is further than the light's far plane when its z coordinate is larger than 1.0
- Wrapping method not work anymore (compare the z component with the depth map values → returns true for z larger than 1.0)
- Fix: simply force the shadow value to 0.0 whenever the projected vector's z coordinate is larger than 1.0:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
   …
   if(projCoords.z > 1.0)
        shadow = 0.0;
    return shadow;
}
```

F5…

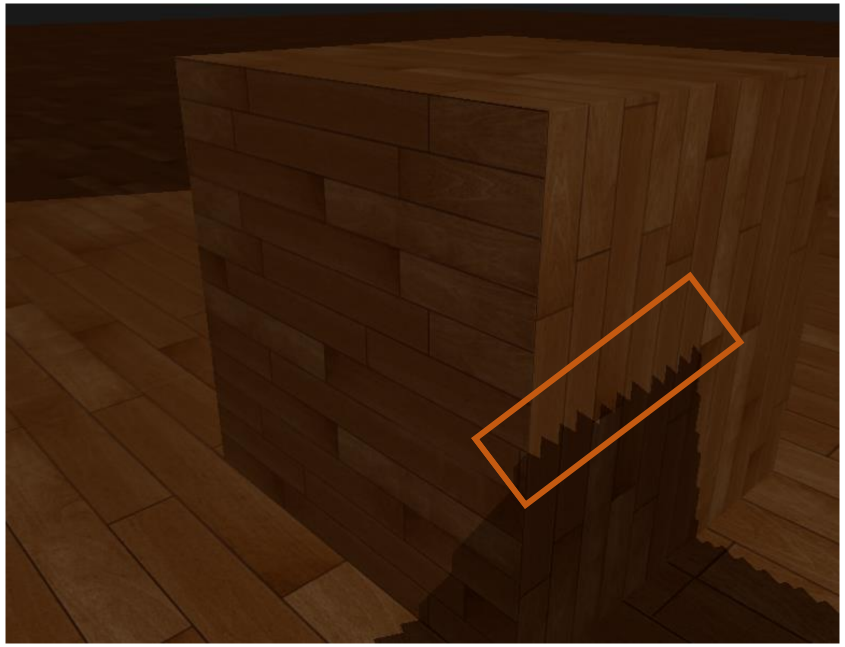
- … problem solved



Percentage–Closer Filtering (PCF)

Introduction

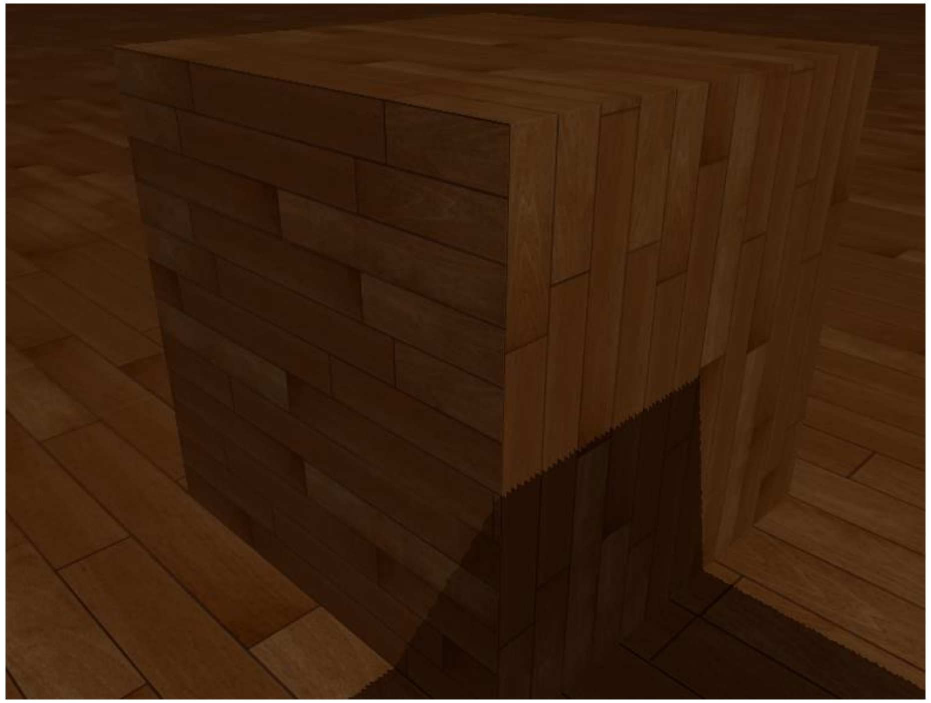
- Zoom in on the shadows the resolution dependency of shadow mapping quickly becomes apparent
- Depth map fixed resolution, depth frequently spans more than one fragment per texel
- Thus, multiple fragments sample the same depth value from the depth map and come to the same shadow conclusions, which produces these jagged blocky edges



Introduction

- Reduce blocky shadows by increasing the depth map resolution or by trying to fit the light frustum as closely to the scene as possible
- Here, change resolution:

```
const unsigned int SHADOW_WIDTH = 4096;
const unsigned int SHADOW_HEIGHT = 4096;
```



Introduction

- Another (partial) solution is called percentage-closer filtering (PCF), a term with many different filtering functions that produce softer shadows, making them appear less blocky or hard
- Idea is to sample more than once from the depth map, each time with slightly different texture coordinates
- For each individual sample we check whether it is in shadow or not
- All the sub-results are then combined and averaged and we get a nice soft looking shadow

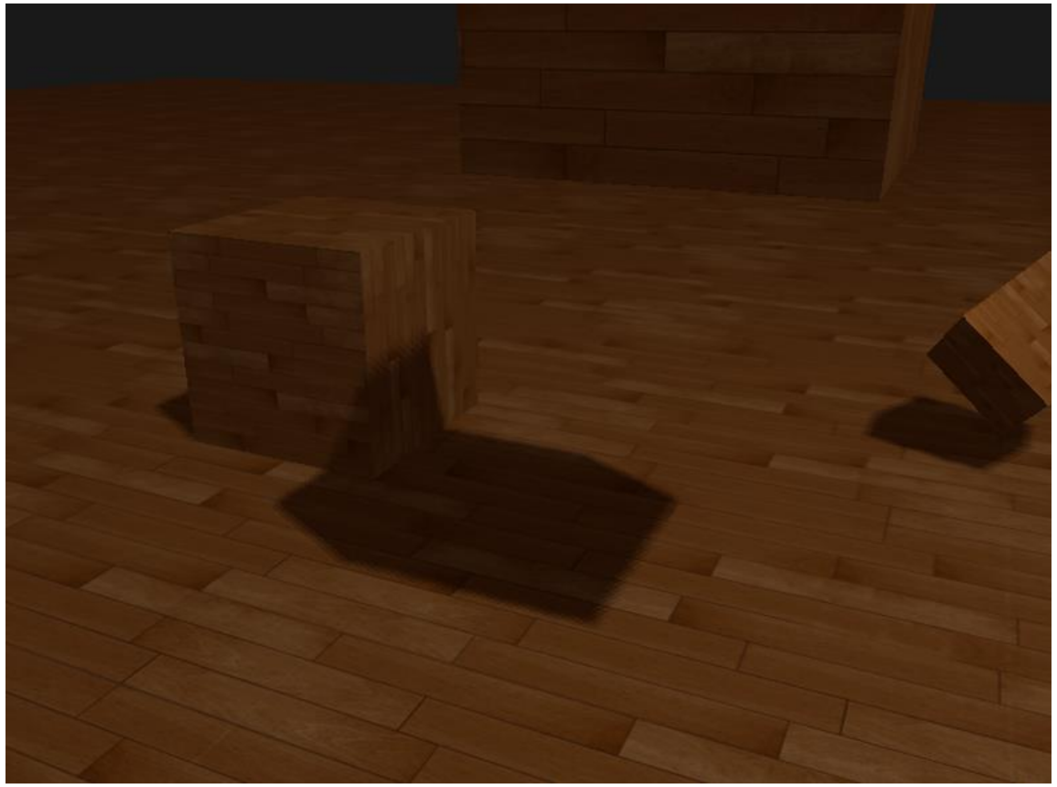
PCF

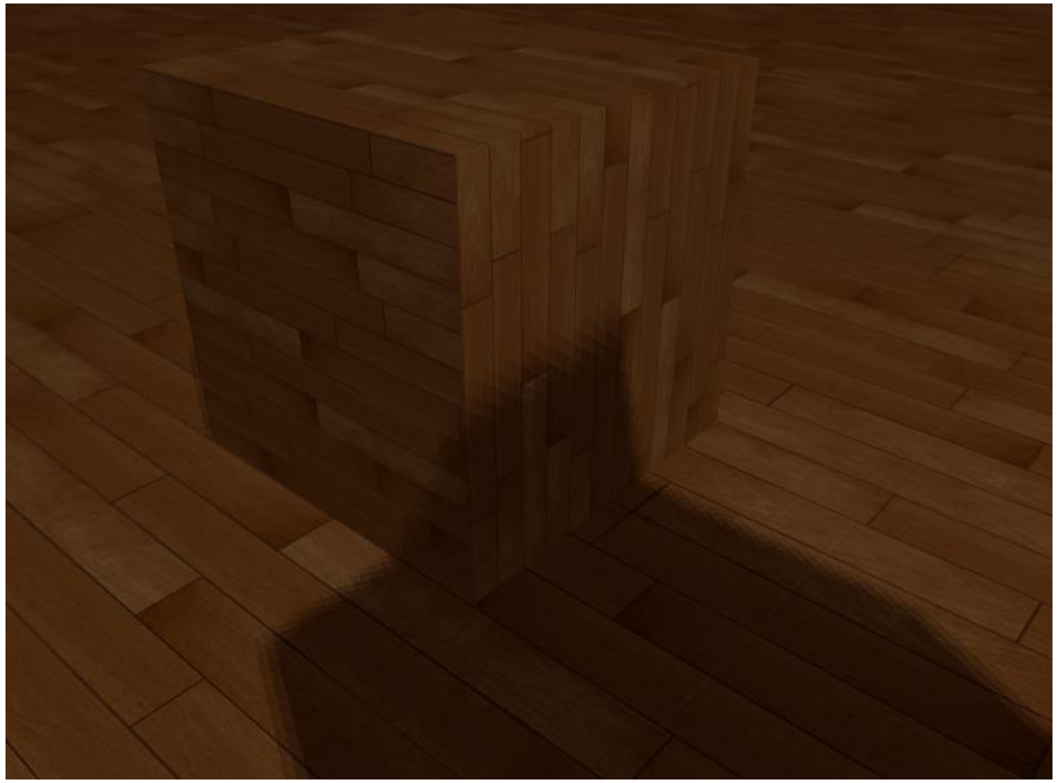
- Simple implementation of PCF (sample the surrounding texels of the depth map and average the results):

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
   for(int y = -1; y <= 1; ++y)
   {
      float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y)*
                                                  texelSize).r;
      shadow += currentDepth - bias > pcfDepth  ? 1.0 : 0.0;
   }
}
shadow /= 9.0;
```

F5…

- … better from a distance





sampler2DShadow*

sampler2DShadow

- Change und add texture parameters:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
```

sampler2DShadow

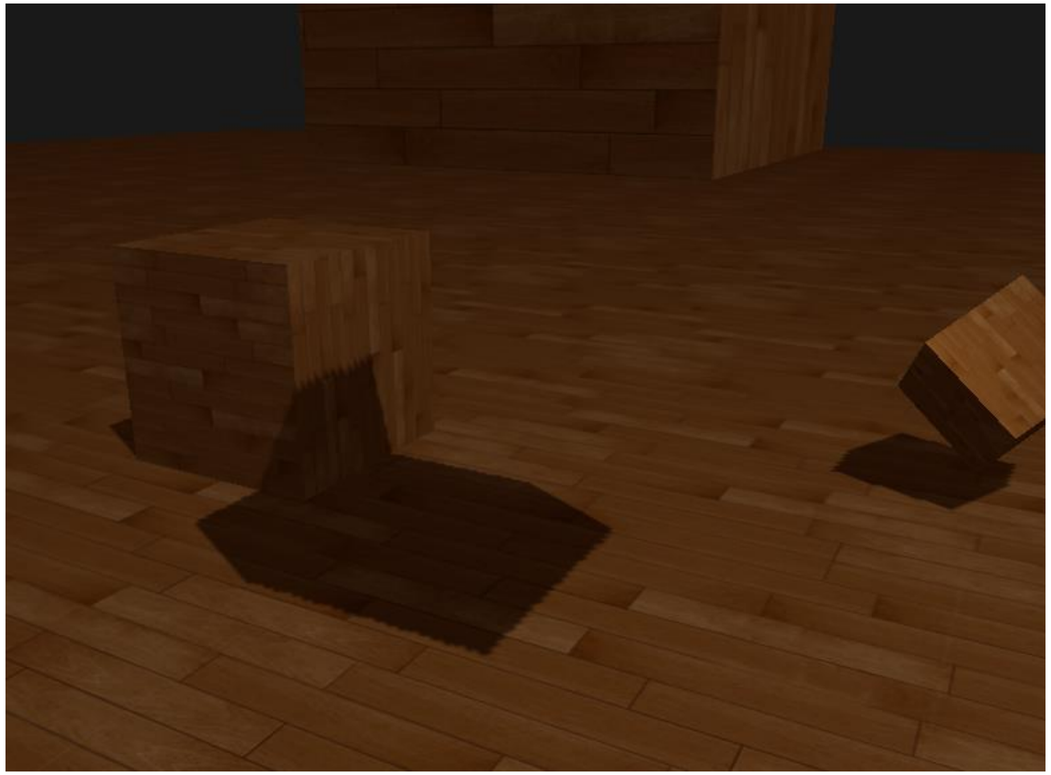
- Fragment Shader:

```
//uniform sampler2D shadowMap;
uniform sampler2DShadow shadowMap;
…
float ShadowCalculation(vec4 fragPosLightSpace)
{
   vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
   projCoords = projCoords * 0.5 + 0.5;
   projCoords.z-=0.005;

   float closestDepth = 1-texture(shadowMap, projCoords);
   return closestDepth;
}
```

F5…

- … simple shadows



sampler2DShadow

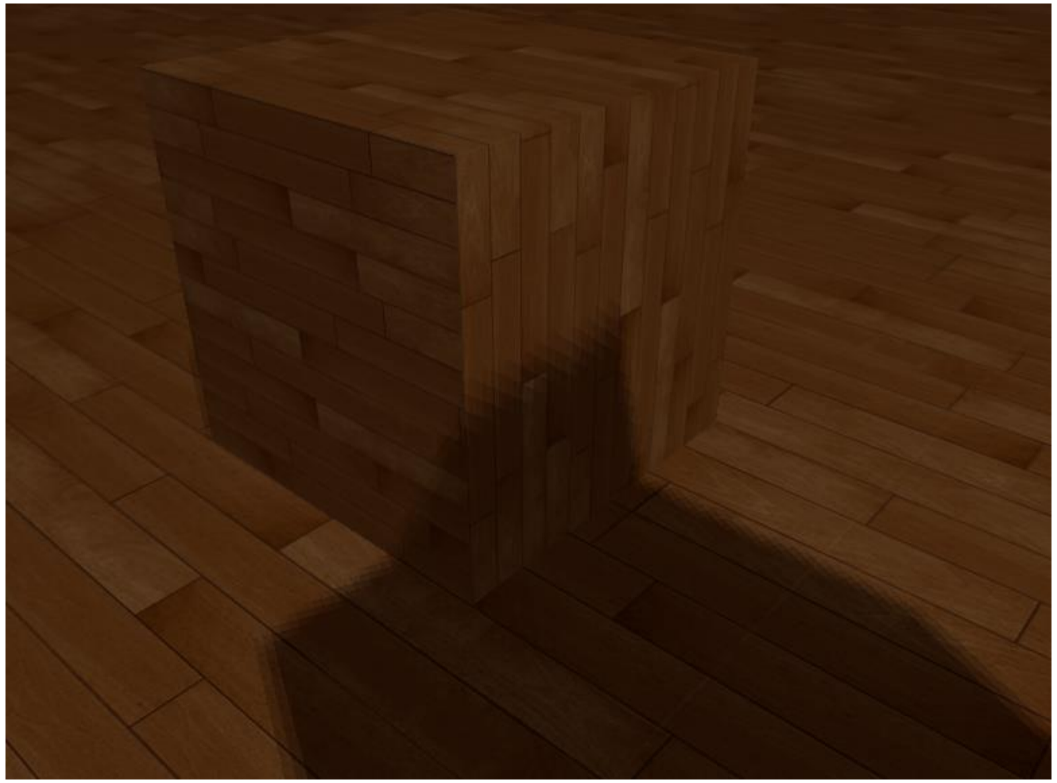
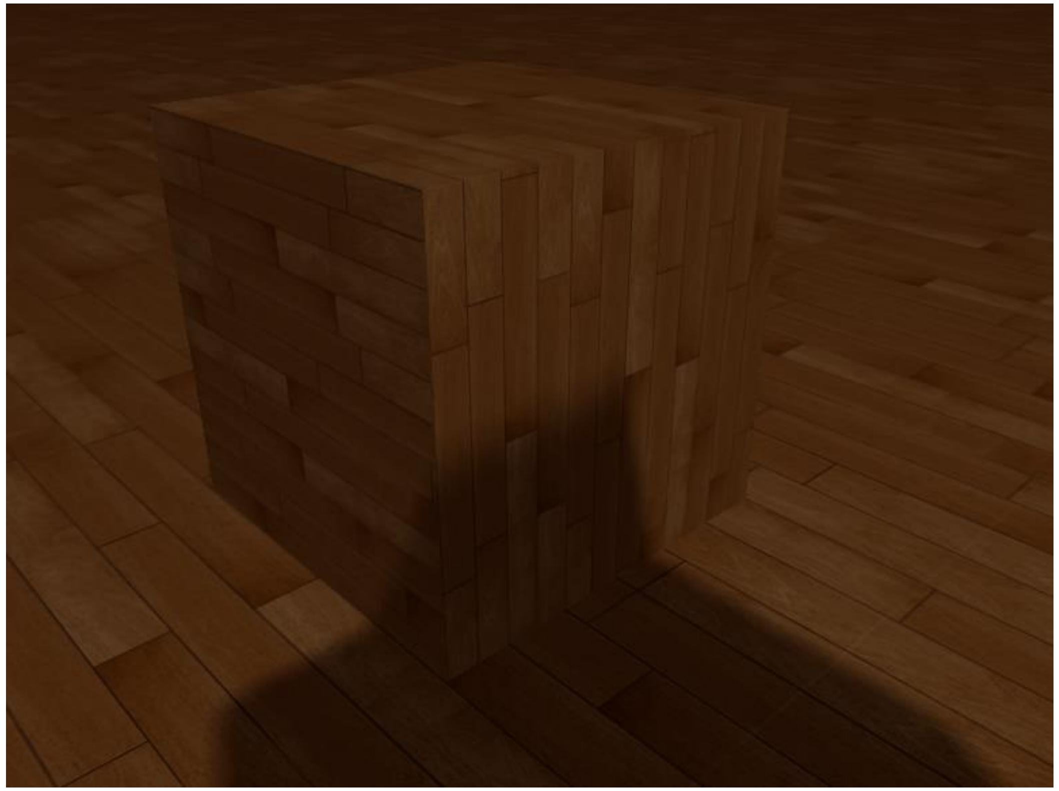
- Add PCF:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
   vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
   projCoords = projCoords * 0.5 + 0.5;
   float shadow = 0.0;
   vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
   float bias = max(0.05 * (1.0 - dot(normalize(fs_in.Normal), normalize(lightPos -
                                                       fs_in.FragPos))), 0.005);

   for(int x = -1; x <= 1; ++x)
   {
     for(int y = -1; y <= 1; ++y)
     {
        shadow += texture(shadowMap, vec3(projCoords.xy + vec2(x, y) *
                                              texelSize, projCoords.z-bias));
        }
    }
    shadow /= 9.0;
    return 1-shadow;
}
```

F5…

- … much better

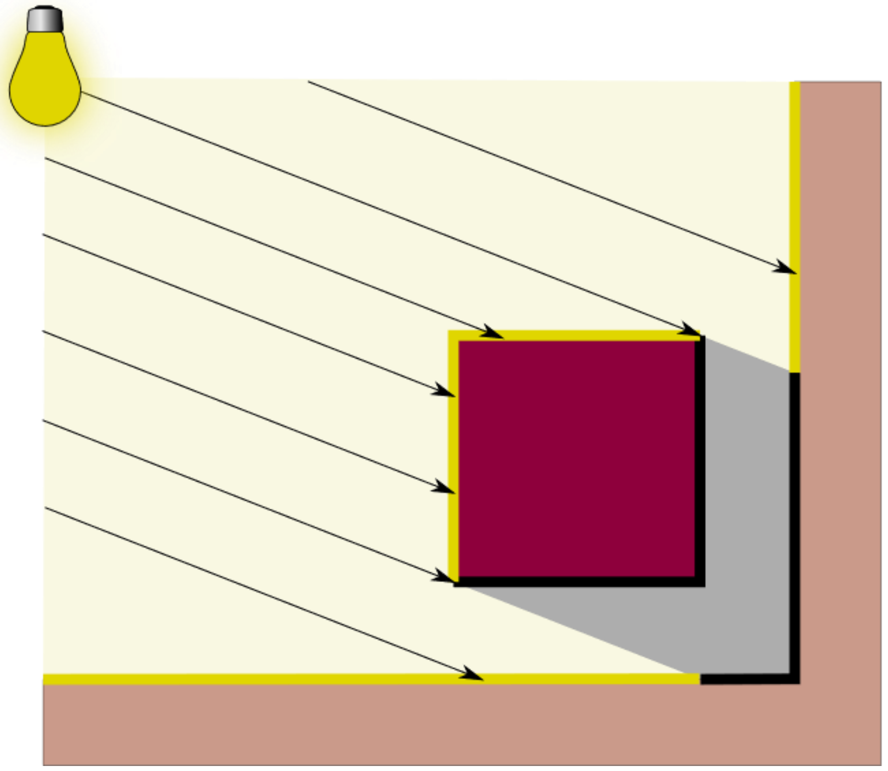
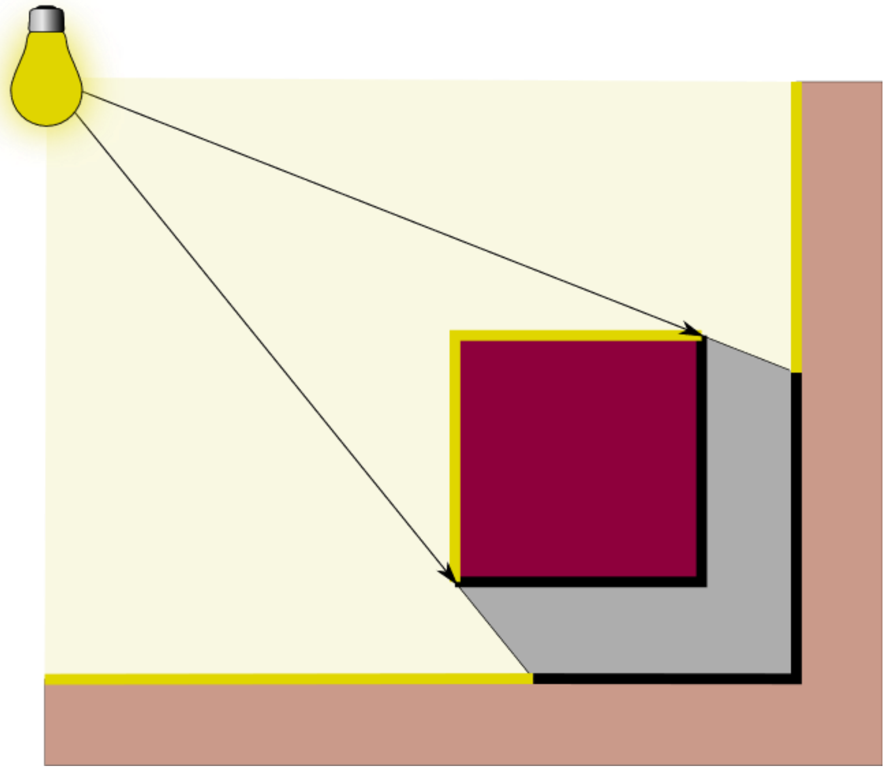
Orthographic vs. Projection

Introduction

- Difference between rendering the depth map with an orthographic or a projection matrix
- Orthographic projection does not deform the scene with perspective → view/light rays are parallel (great for directional lights)

Introduction

- Perspective projection matrix deforms vertices based on perspective which gives different results:

Introduction

- Another subtle difference with a perspective projection matrix: visualizing the depth buffer will often give an almost completely white result
- With perspective projection depth is transformed to non-linear depth values with most of its noticeable range close to the near plane
- To properly view the depth values (as with orthographic) transform the non-linear depth values to linear

Introduction

```glsl
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D depthMap;
uniform float near_plane;
uniform float far_plane;

// required when using a perspective projection matrix
float LinearizeDepth(float depth)
{
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * near_plane * far_plane) / (far_plane + near_plane - z * (far_plane - near_plane));
}

void main()
{
    float depthValue = texture(depthMap, TexCoords).r;
    FragColor = vec4(vec3(LinearizeDepth(depthValue) / far_plane), 1.0); // perspective
    //FragColor = vec4(vec3(depthValue), 1.0); // orthographic
}
```

Questions???