Shadows

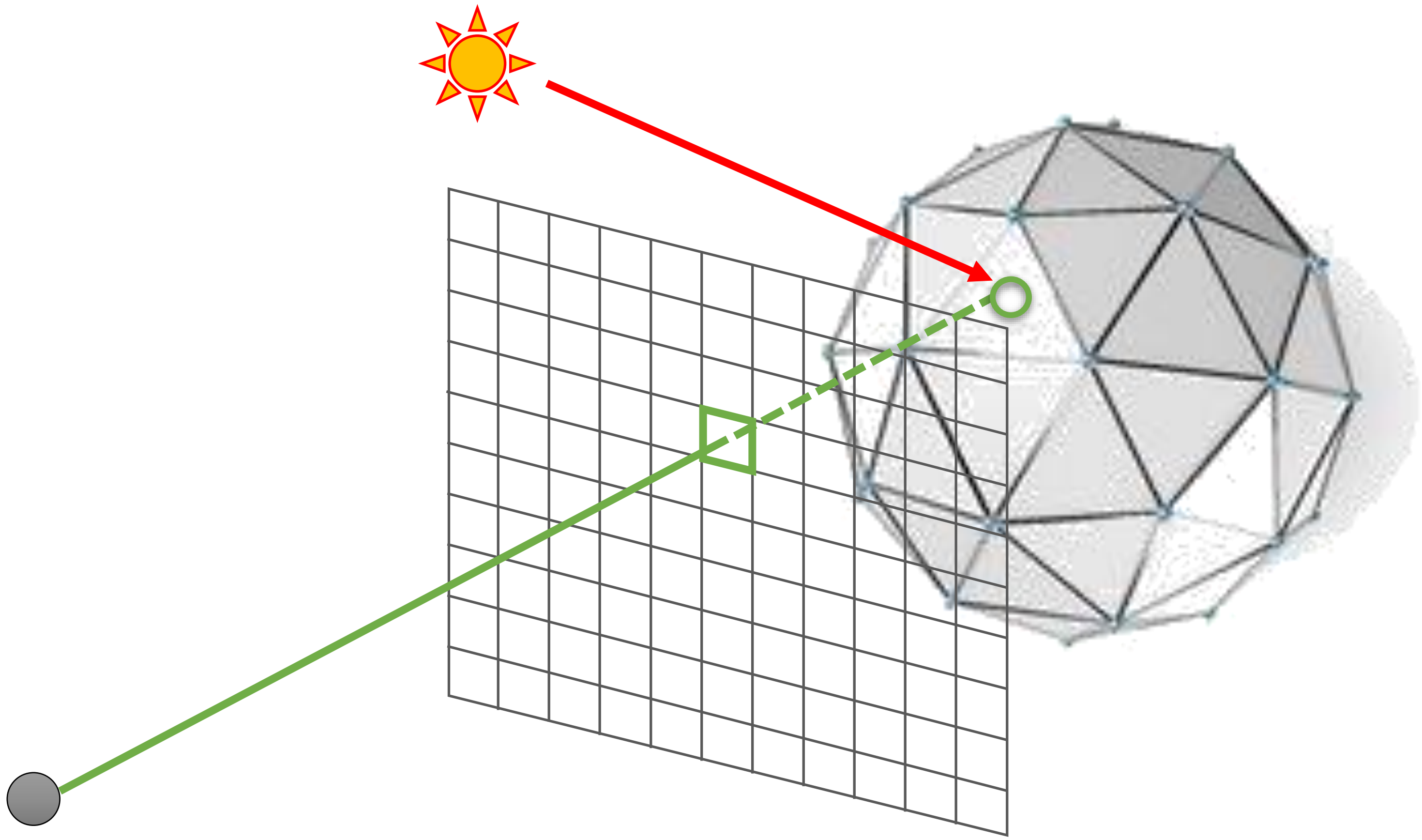
Computer Graphics

Yu-Ting Wu

Shadow Map

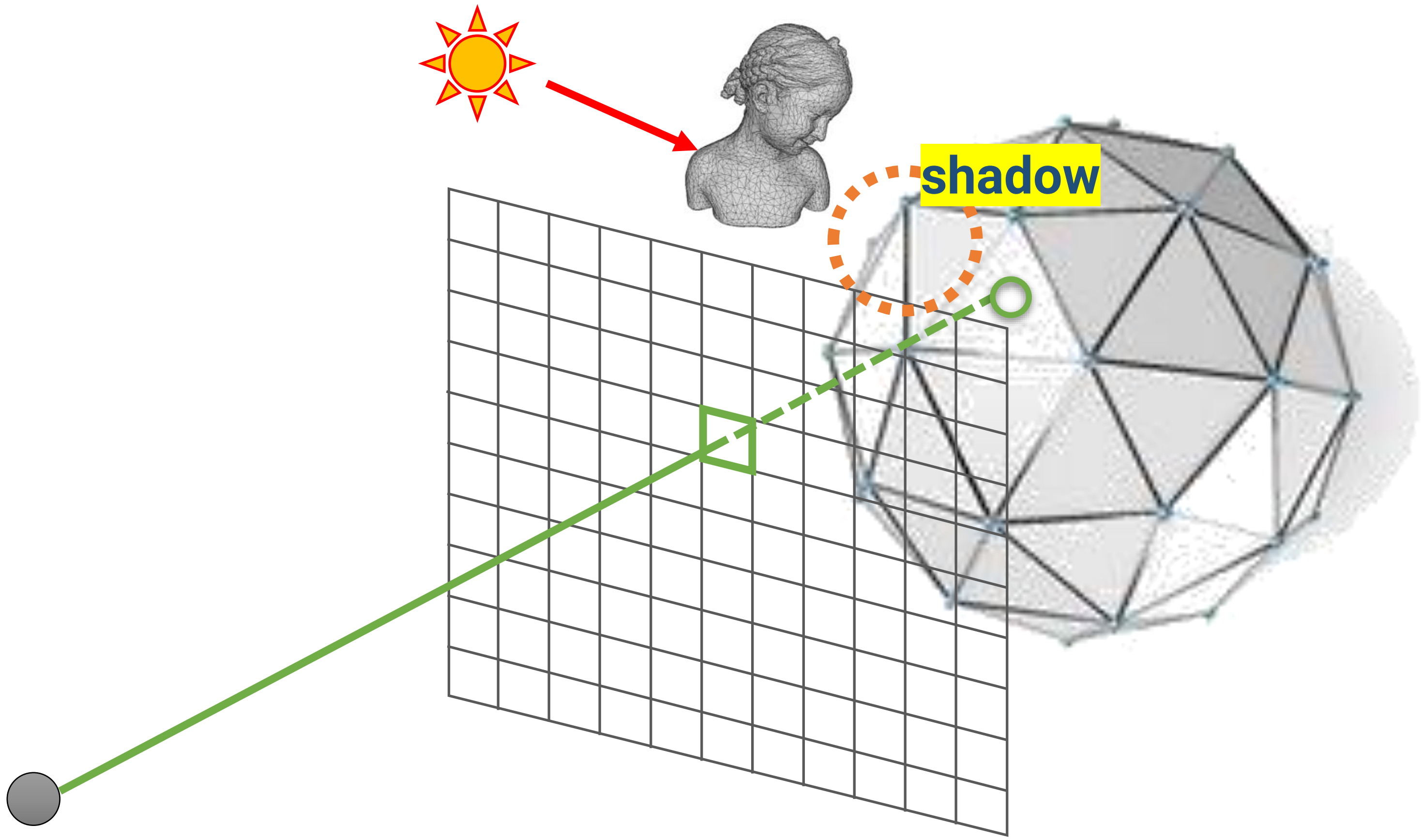
Shadow

- So far, we consider the light to be fully visible to a shading point



Shadow (cont.)

- It is common that a lighting direction is occluded by some other objects



Shadow (cont.)

- Shadows are very important to provide depth cues

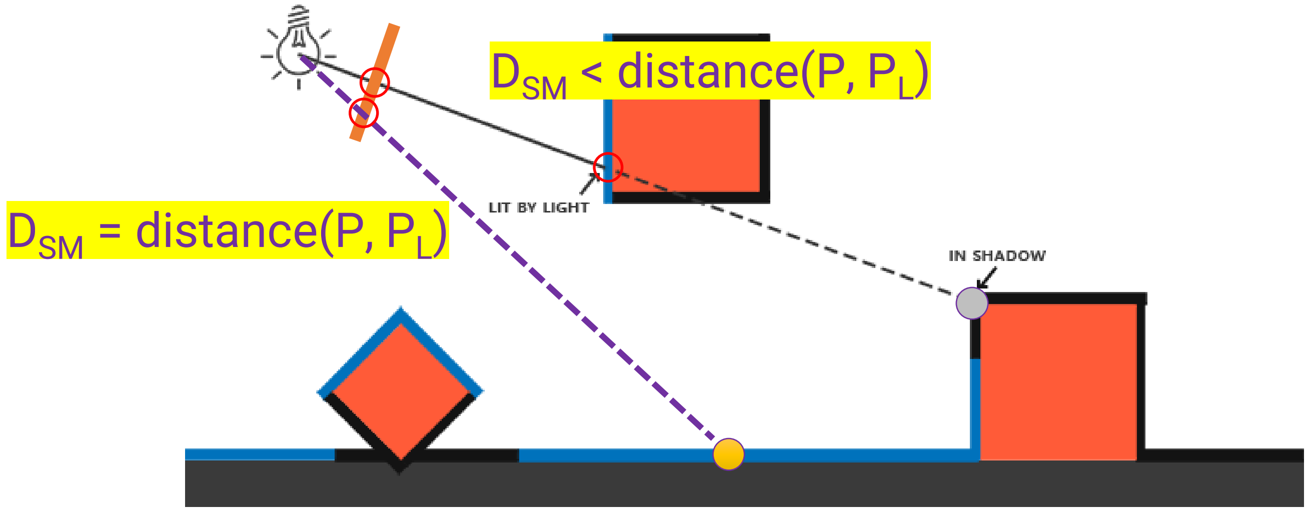
Shadow Map Overview

- Like the case of transparency, rendering shadows is difficult for rasterization because **each polygon only has its own information**
  - It does not know which triangle blocks the light, so it cannot determine the shadow attenuation in its fragment shader

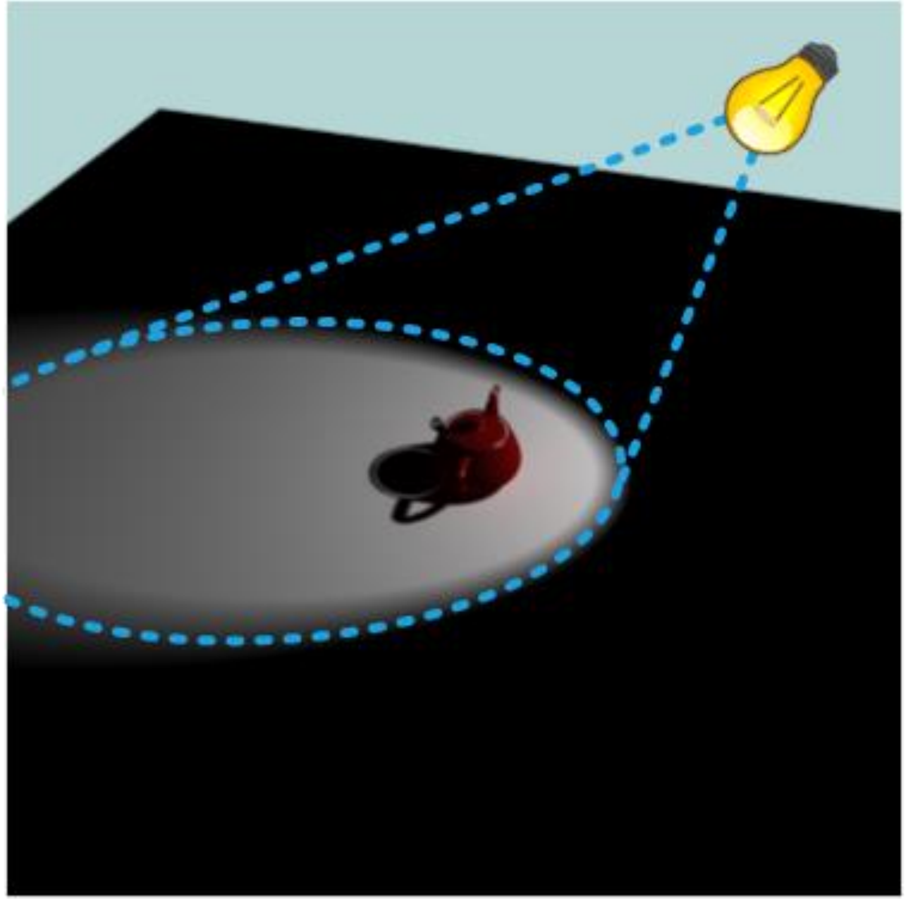
- **Shadow map** is a two-pass rendering technique for simulating shadows using rasterization

Shadow Map Overview (cont.)

- Major concept
  - **First pass: rendering a depth map from the light position**
    - Record the closest surface from the light and generate a **shadow map**
  - **Second pass: rendering from the camera**
    - During lighting computation, lookup the shadow map to determine the shadow



Shadow Map Overview (cont.)



final rendering
(rendering from the camera view)

shadow map
(rendering from the light view)

Shadow Map Overview (cont.)

- Major concept
  - https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

rendering from the light view

```
// 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();
```

rendering from the camera view

Shadow Map for Directional Lights

- **First pass: shadow map generation**

rendering from the light view

```
// 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();
```

bind the render target to shadow map FBO

rendering from the light view

bind to default screen

Shadow Map for Directional Lights (cont.)

- **First pass: shadow map generation**
  - Create a FBO for the shadow map

```
// configure depth map FBO
// -----------------------
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// create depth texture
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_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);
// attach depth texture as FBO'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);
```

shadow map resolution

DL_DEPTH_COMPONENT(16/24/32F)

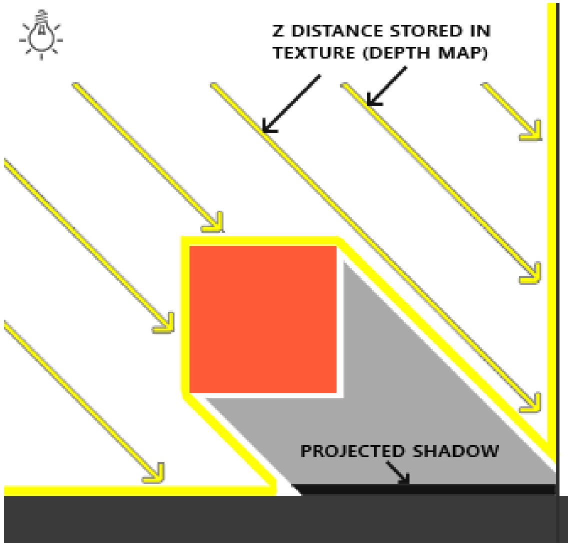
tell OpenGL we don't need a color buffer

Shadow Map for Directional Lights (cont.)

- **First pass: shadow map generation**
  - Choose a proper resolution

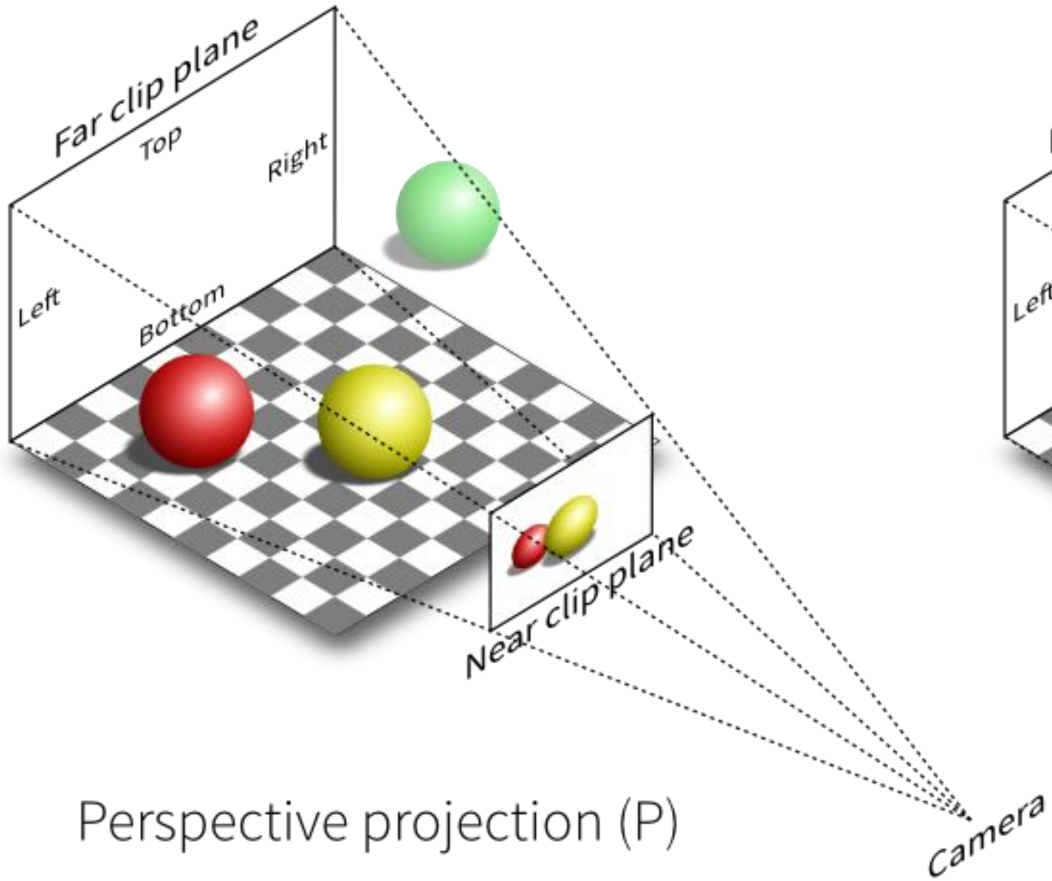
Shadow Map for Directional Lights (cont.)

- **First pass: shadow map generation**
  - A directional light does not have a light position
  - We set the camera to a position somewhere **along the lines of the light direction**
  - Use **orthogonal projection**

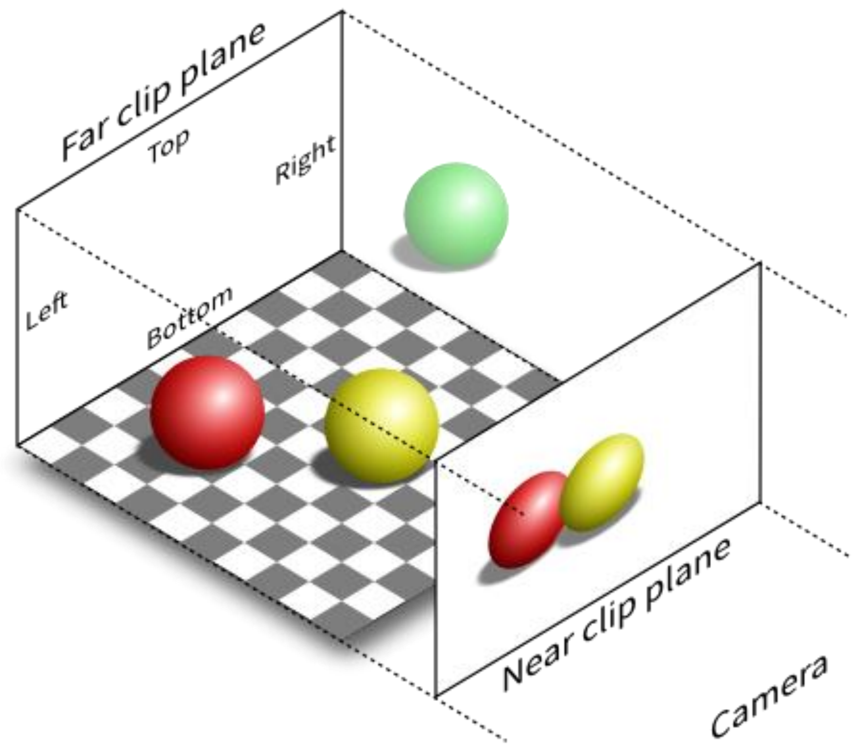



```
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
//lightProjection = glm::perspective(glm::radians(45.0f), (GLfloat)SHADOW_WIDTH / (
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
```

Recap: Projective Camera Models



Perspective projection (P)



Orthographic projection (O)

Shadow Map for Directional Lights (cont.)

- **First pass: shadow map generation**
  - 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);
}
```

Proj. * View Matrix (lightSpaceMatrix) · World Matrix (model) · Object Space (vec4(aPos, 1.0))

  - Fragment Shader (do nothing)

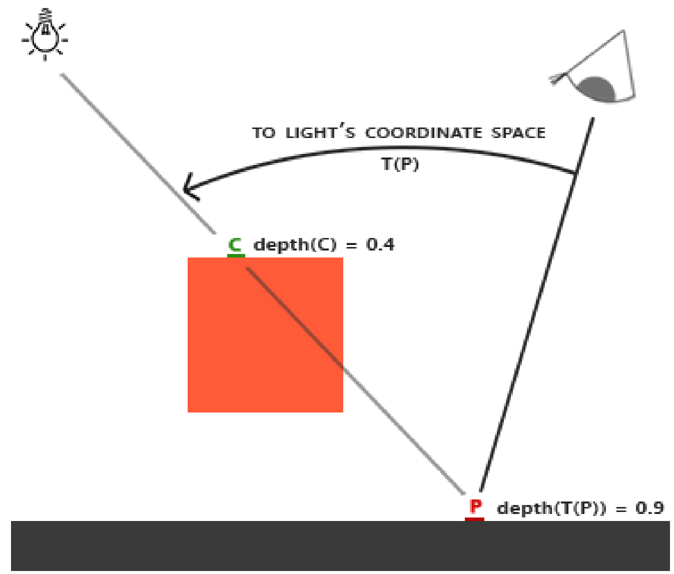
```
#version 330 core

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

Shadow Map for Directional Lights (cont.)

- **Second pass: normal rendering**
  - Render the scene from the camera
  - Look up the shadow map to determine shadows during lighting computation

```
// 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();
```

rendering from the camera view



Shadow Map for Directional Lights (cont.)

- **Second pass: normal rendering**
  - 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;

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 * vec4(vs_out.FragPos, 1.0);
}
```

Clip Space coordinate in the shadow map

Shadow Map for Directional Lights (cont.)

- **Second pass: normal rendering**
  - Fragment Shader

```
#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()
{
    ...
    FragColor = vec4(lighting, 1.0);
}
```

Shadow Map for Directional Lights (cont.)

- **Second pass: normal rendering**
  - Fragment Shader

```
void main()
{
    vec3 color = texture(diffuseTexture, fs_in.TexCoords).rgb;
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightColor = vec3(1.0);
    // ambient
    vec3 ambient = 0.15 * lightColor;
    // diffuse
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float diff = max(dot(lightDir, normal), 0.0);
    vec3 diffuse = diff * lightColor;
    // specular
    vec3 viewDir = normalize(viewPos - fs_in.FragPos);
    float spec = 0.0;
    vec3 halfwayDir = normalize(lightDir + viewDir);
    spec = pow(max(dot(normal, halfwayDir), 0.0), 64.0);
    vec3 specular = spec * lightColor;
    // calculate shadow
    float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
    vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular)) * color;

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

Shadow Map for Directional Lights (cont.)

- **Second pass: normal rendering**
  - Fragment Shader

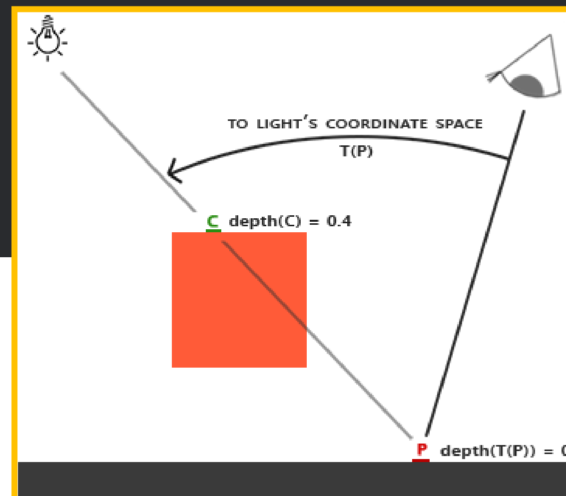
```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range 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;
}
```

to NDC [-1, 1]

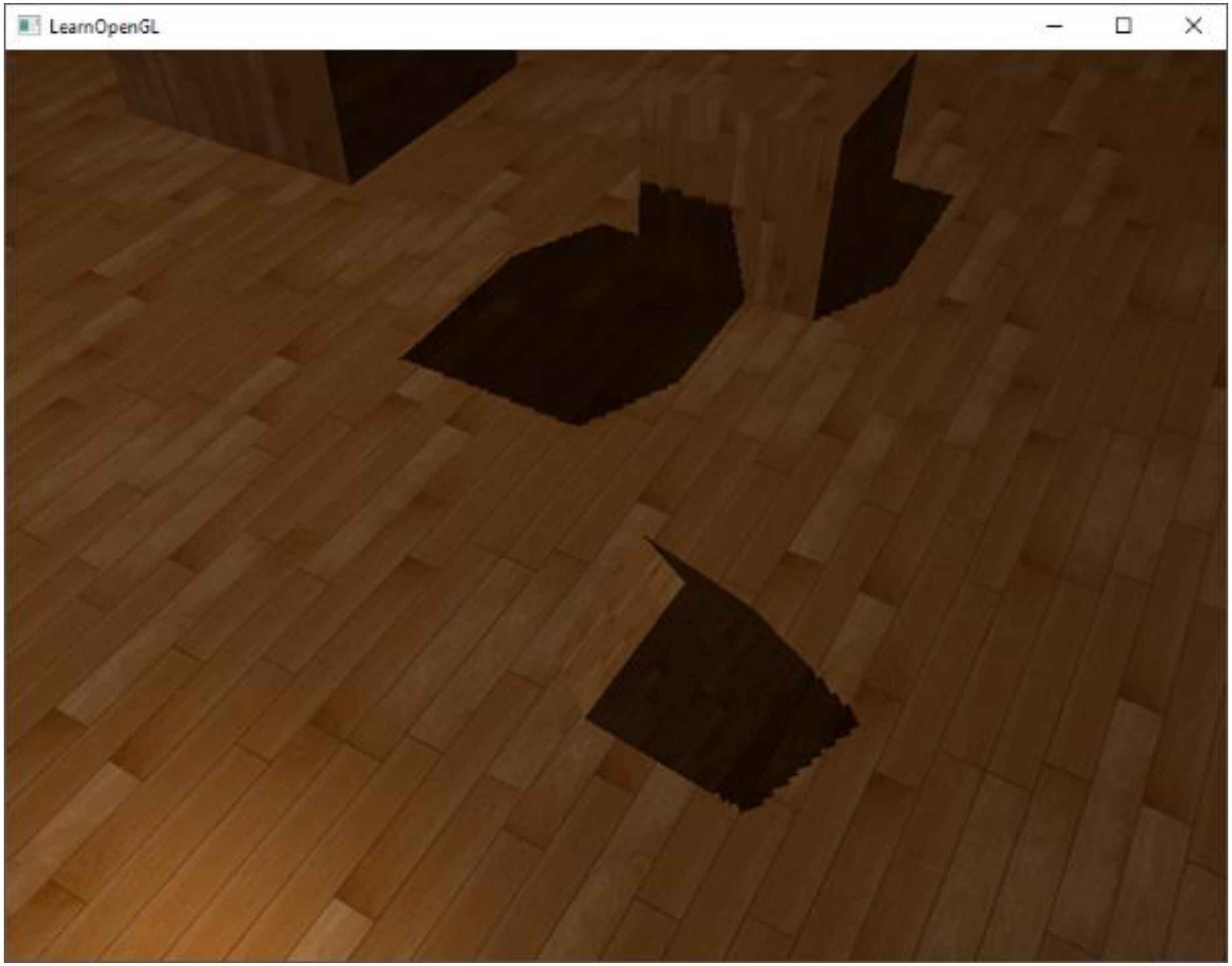
to [0, 1] for looking up the shadow map

(1, 1)

(0, 0)

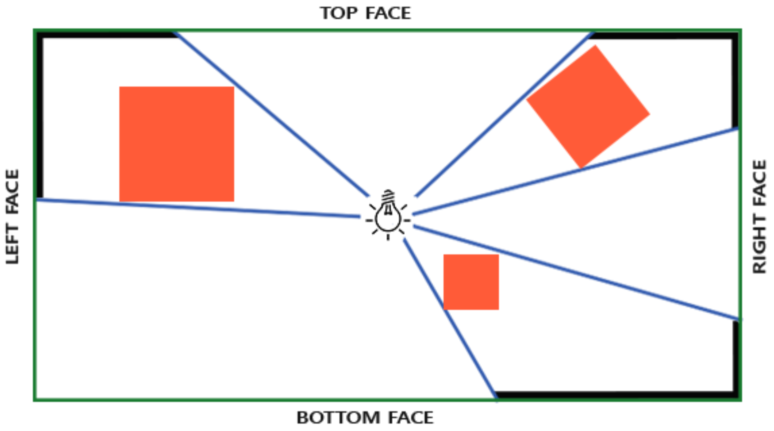


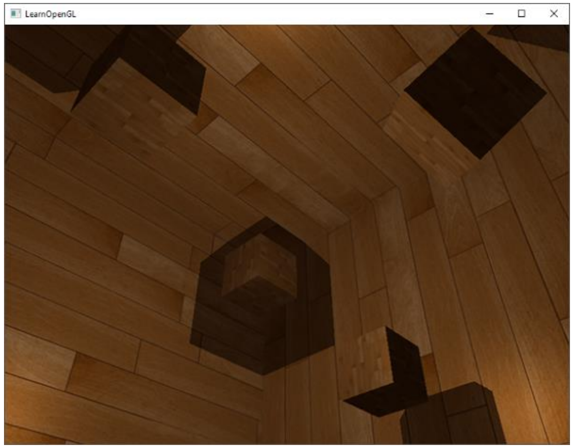
Shadow Map for Directional Lights (cont.)



Shadow Map for Point / Spot Lights

- Generate a shadow map for a **spotlight** is intuitive
  - Locate the camera at the position of the spotlight
  - Use the direction of the spotlight for viewing direction
  - Use **perspective** projection instead of orthogonal projection
- For a point light, you need to render the scene depth into a **cubemap** because the light emits in omni directions
  - https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows





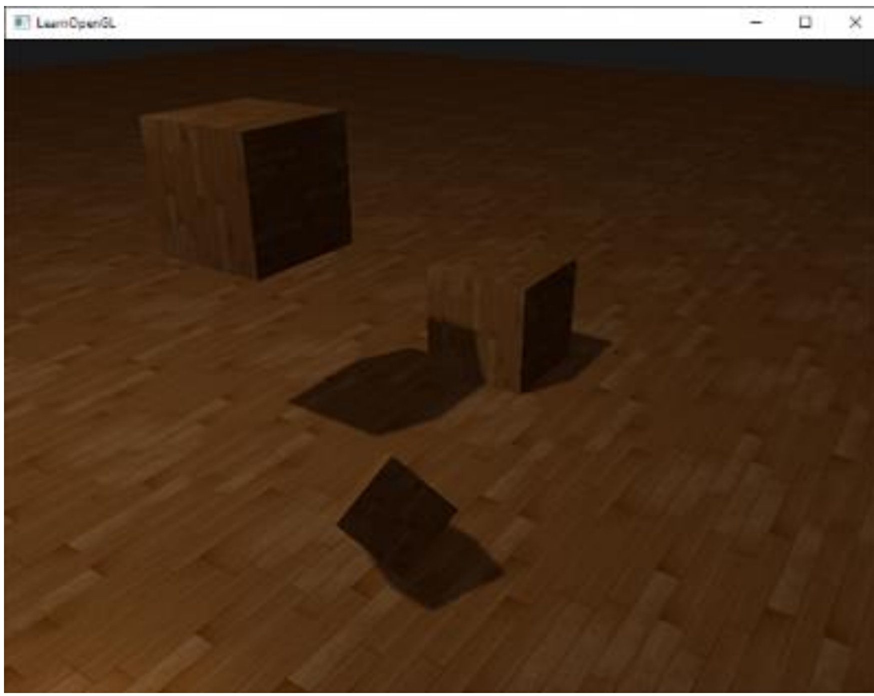
Percentage Closer Filtering

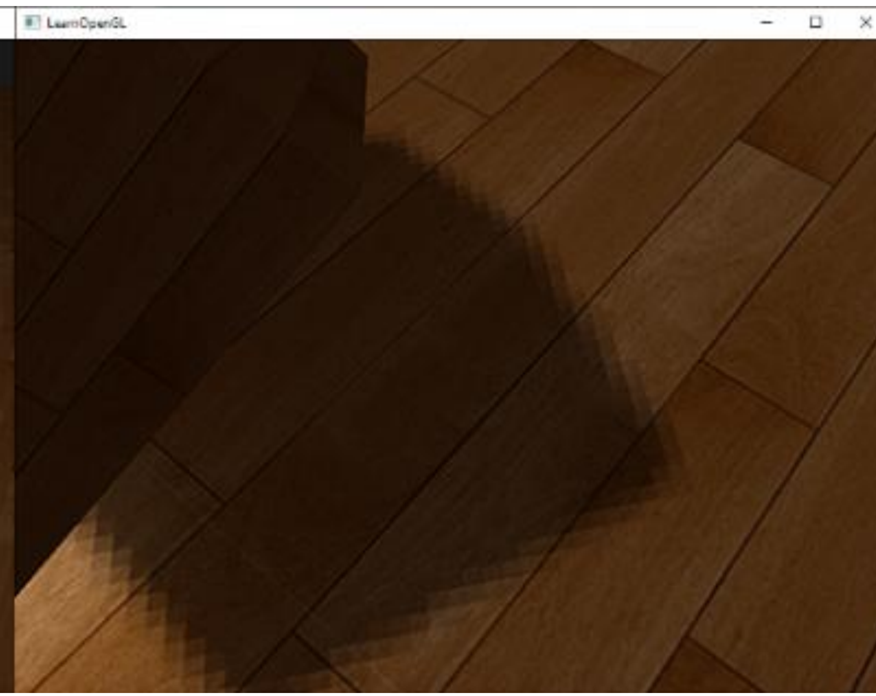
- The shadow map has a fixed (and limited) resolution
- A single lookup of a shadow map often produces jagged blocky edges

- We can reduce these blocky shadows by increasing the depth map resolution, or
- **Sampling more than once** from the depth map, each time with slightly different texture coordinates, and **averaging** the results

Percentage Closer Filtering

```
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;
```

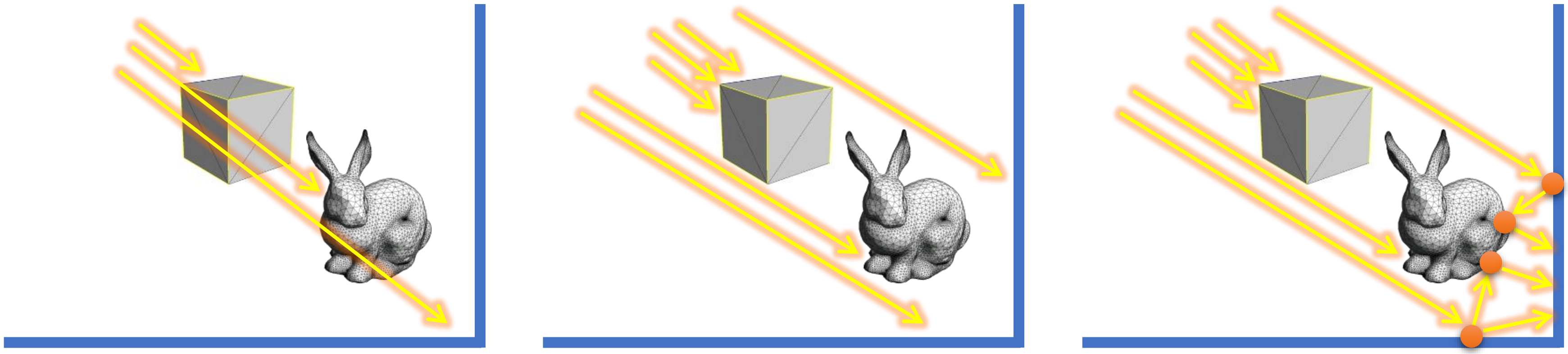




Ambient Occlusion

Recap: Global Illumination

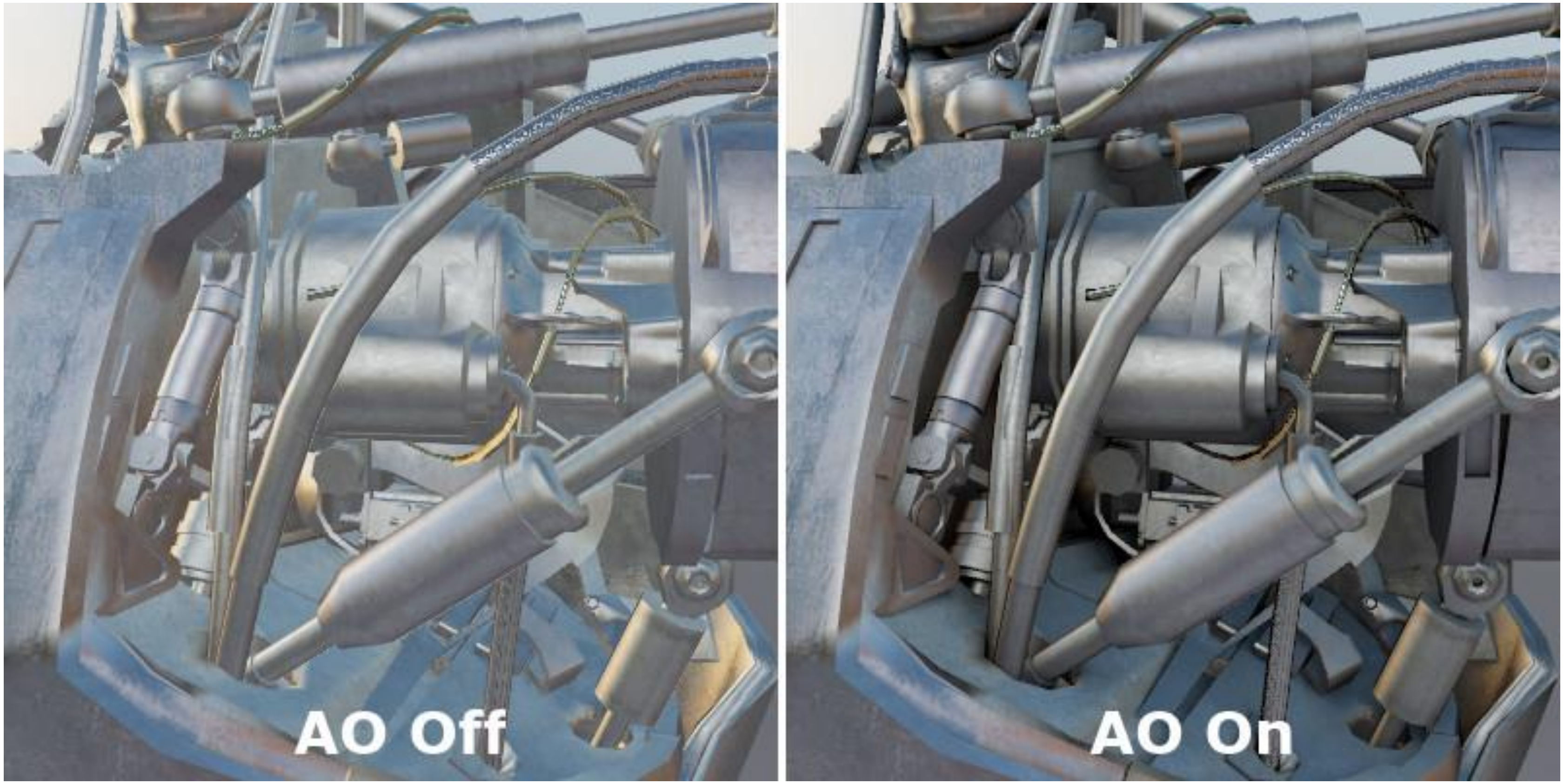
- Global illumination includes multi-bounce lighting
- Very expensive to compute
- In Phong lighting model, a **constant ambient term** is used to account for disregarded illumination
  - However, this produces a "flat", "non-photo-realistic" appearance



local illumination | direct illumination | global illumination

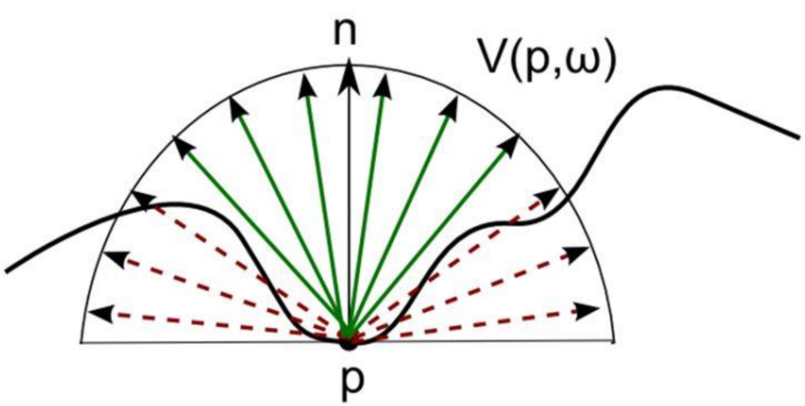
Ambient Occlusion

- Ambient occlusion (AO) is a popular technique to approximate global illumination

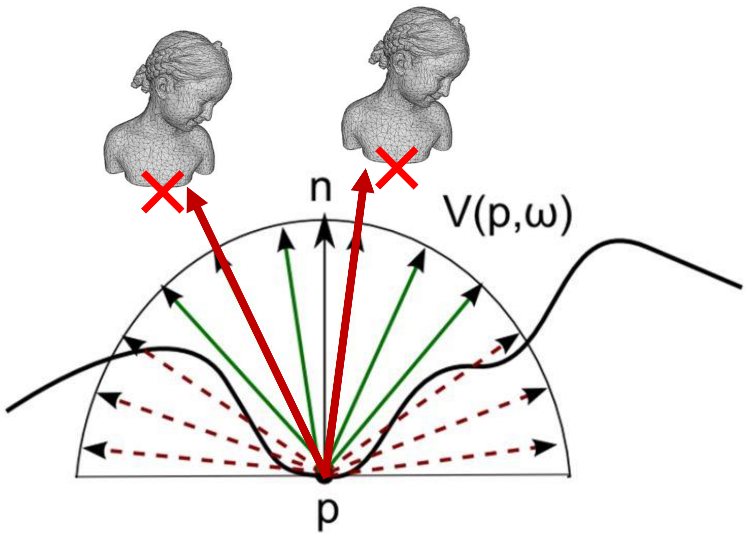


Ambient Occlusion (cont.)

- **Ambient occlusion (AO)** is a popular technique to approximate global illumination
  - Modulate ambient light by the surface's **accessibility**
  - Greatly enhance **depth perception** with a relatively low cost

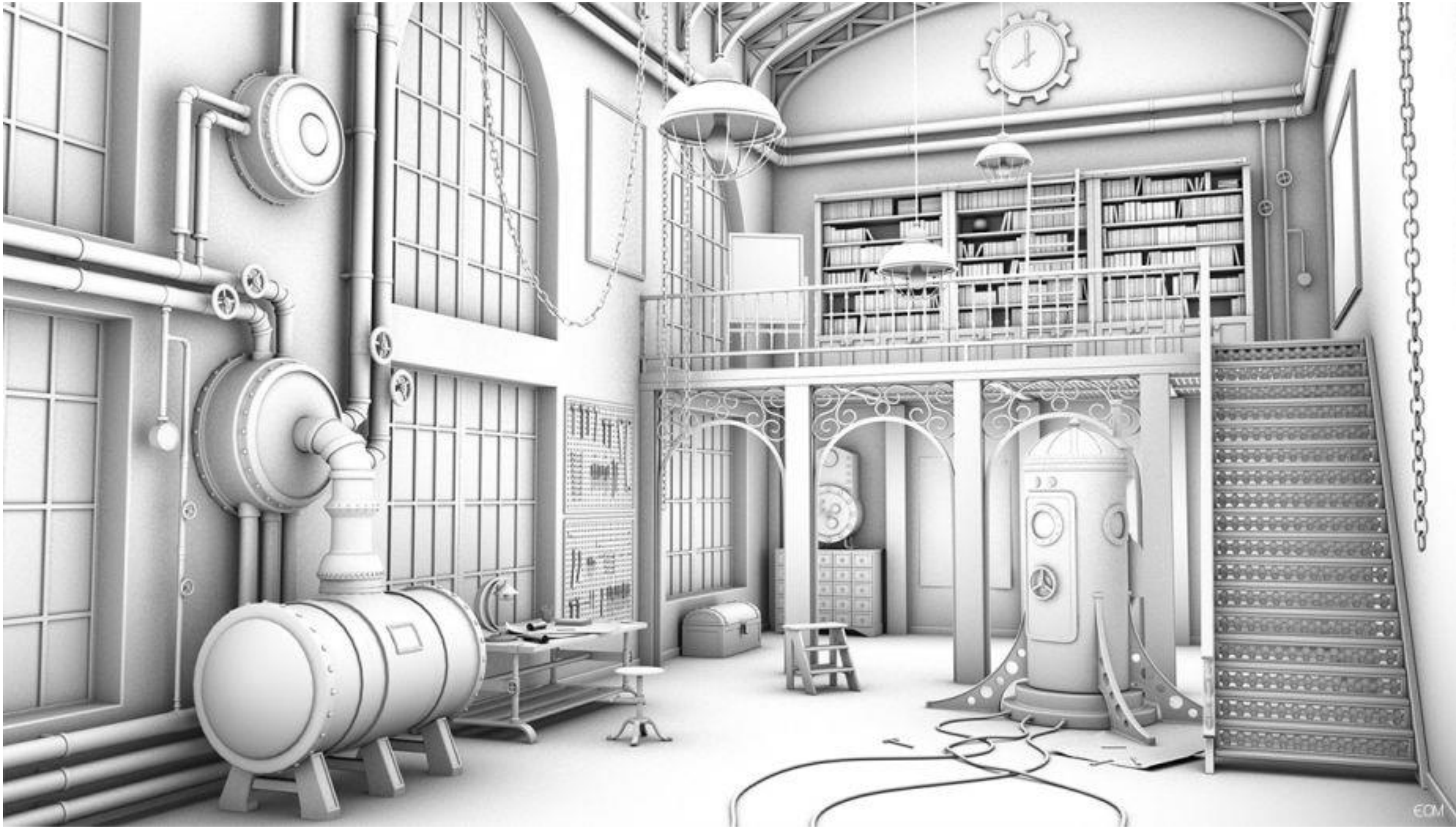


$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$

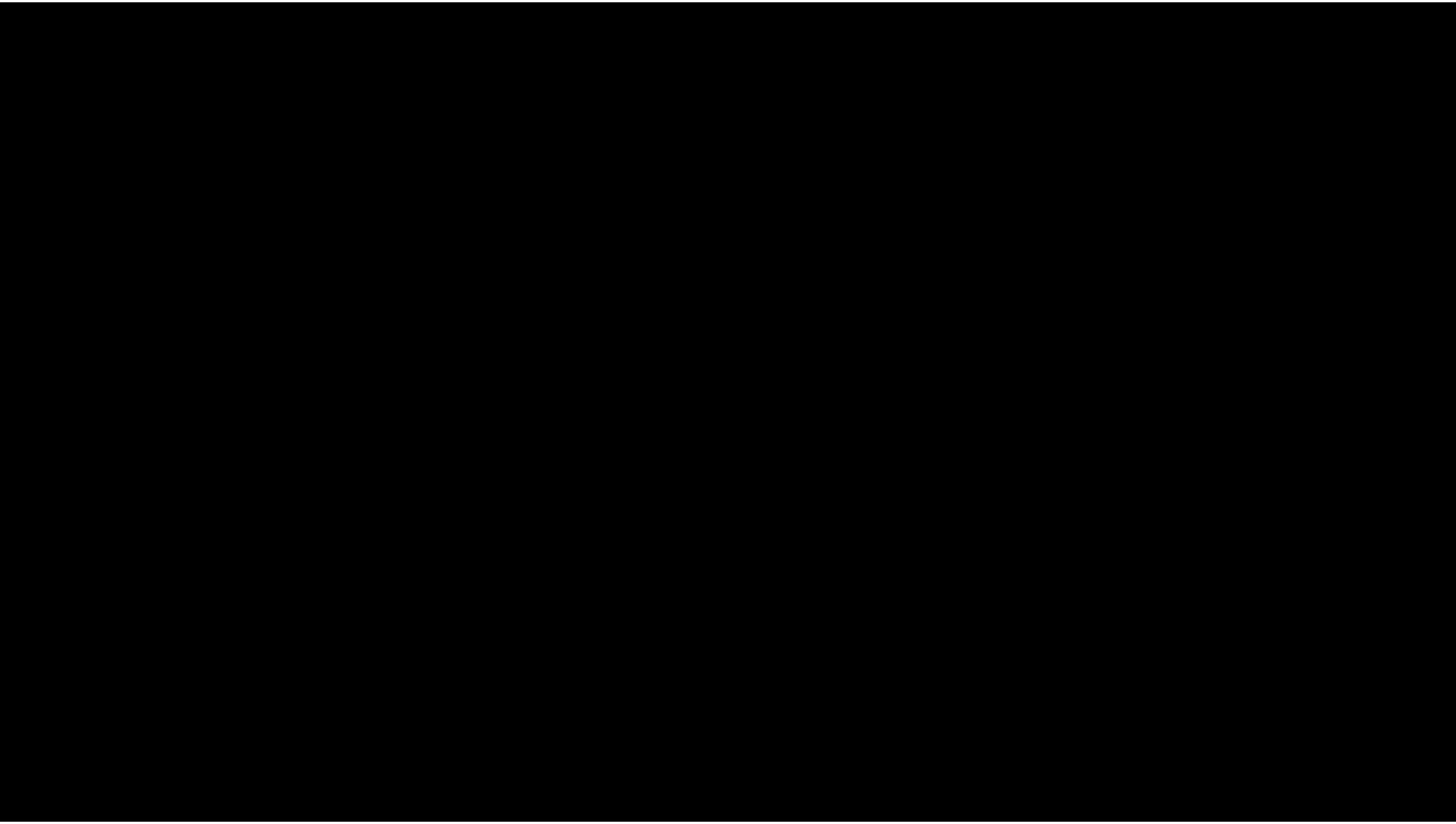


$$AO(\mathbf{p},\mathbf{n}) = \frac{1}{\pi}\int_{\Omega} V(\mathbf{p},\omega)\,\mathbf{n}\cdot\omega\, d\omega,$$

Ambient Occlusion (cont.)

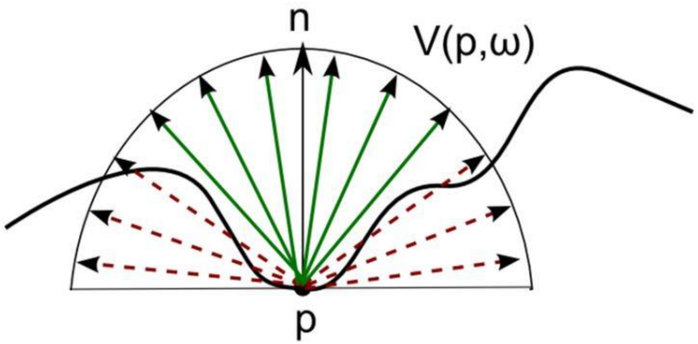


Ambient Occlusion (cont.)

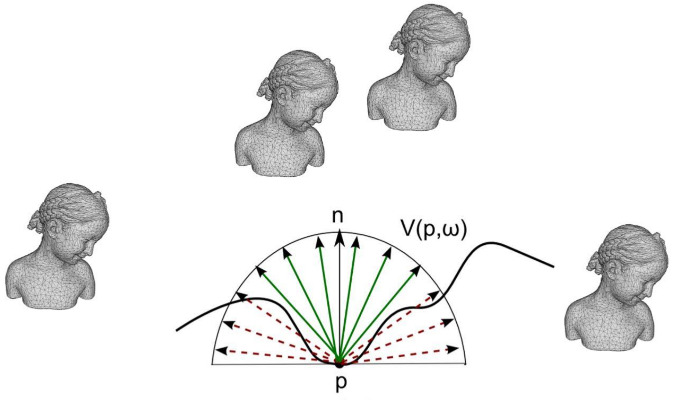


Ambient Occlusion

- To compute AO, you need to know whether the ambient light is occluded in a direction
- In ray tracing, you can **trace rays** to determine the visibility
- For rasterization; however, this is difficult because **each polygon only knows its information** (again!)
  - Performance is also an issue!



$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega)\, \mathbf{n} \cdot \omega d\omega,$$



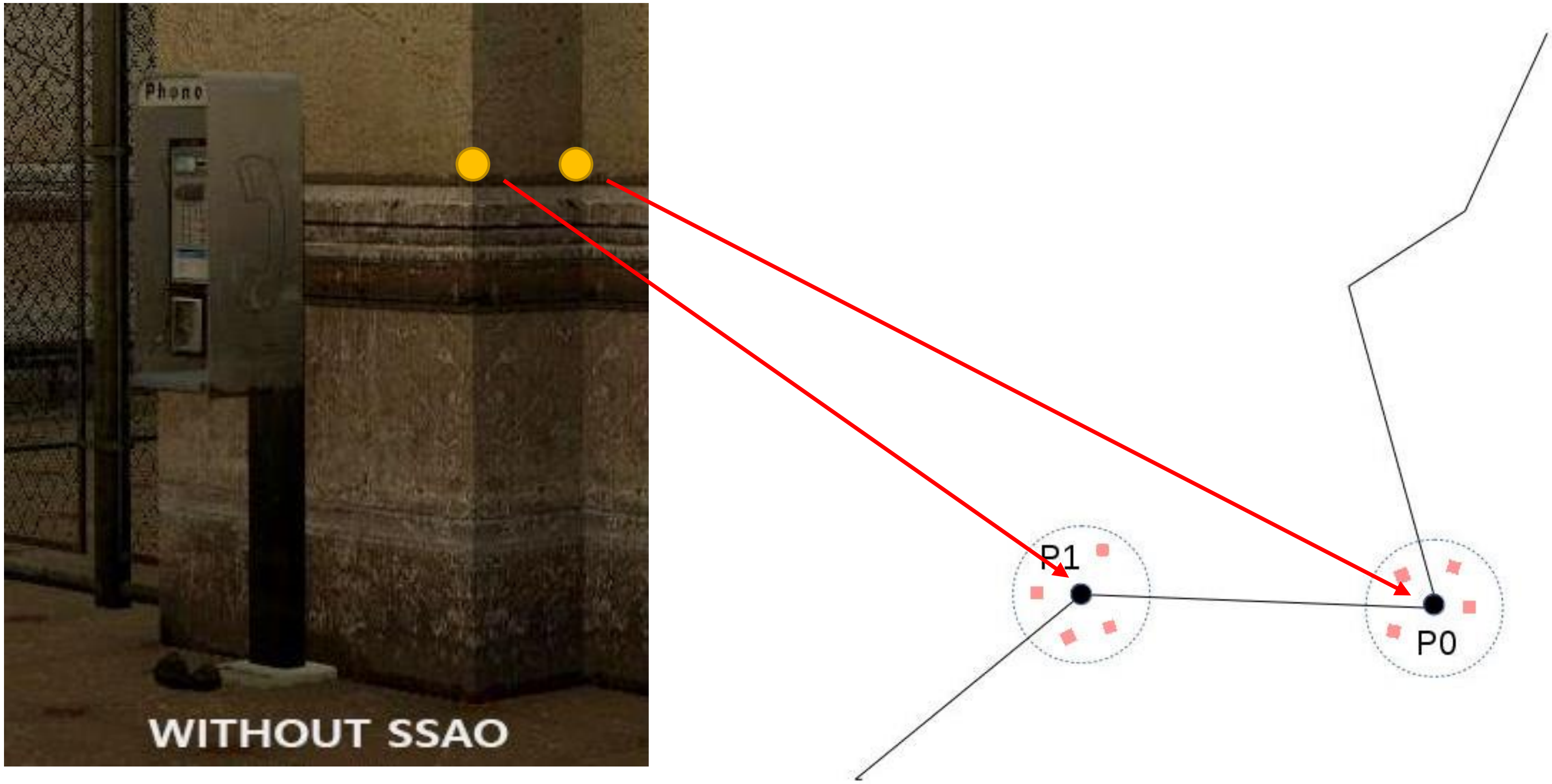
$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega)\, \mathbf{n} \cdot \omega d\omega,$$

Screen-space Ambient Occlusion

- Crytek implemented a real-time solution for ***Crysis***
  - Quickly became the yardstick for game graphics
  - Known as screen-space ambient occlusion (**SSAO**)
- Major idea
  - Find nearby occluders in the **depth buffer (screen-space)**

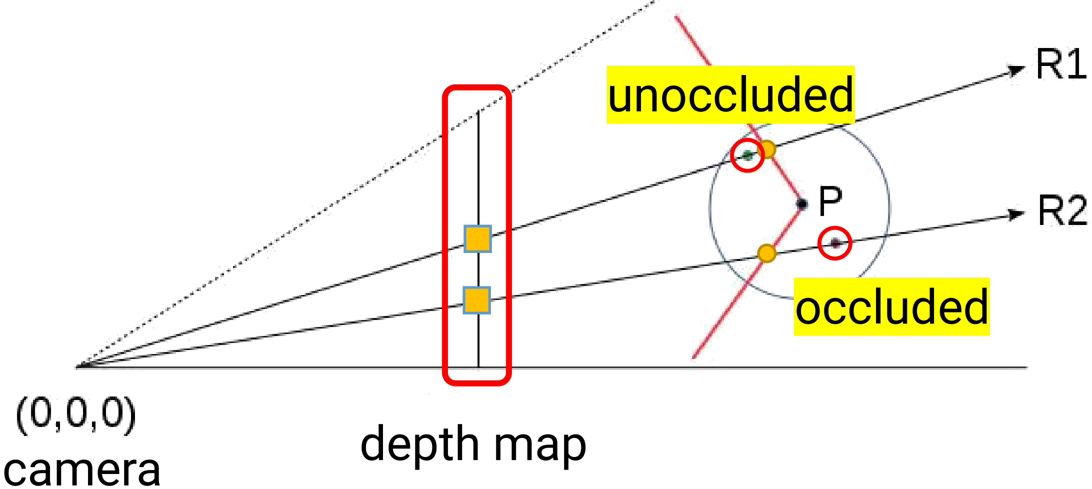
Screen-space Ambient Occlusion (cont.)

- Method
  - Generate samples within a sphere around the shading point (fragment)



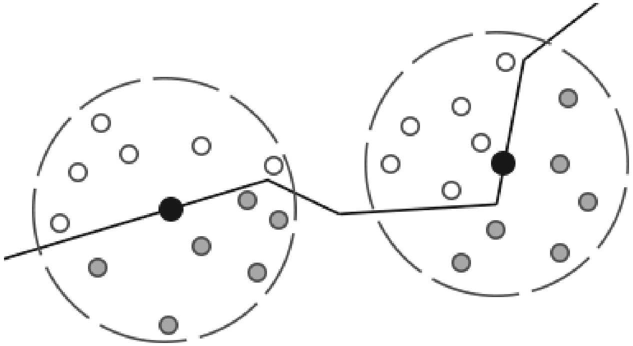
Screen-space Ambient Occlusion (cont.)

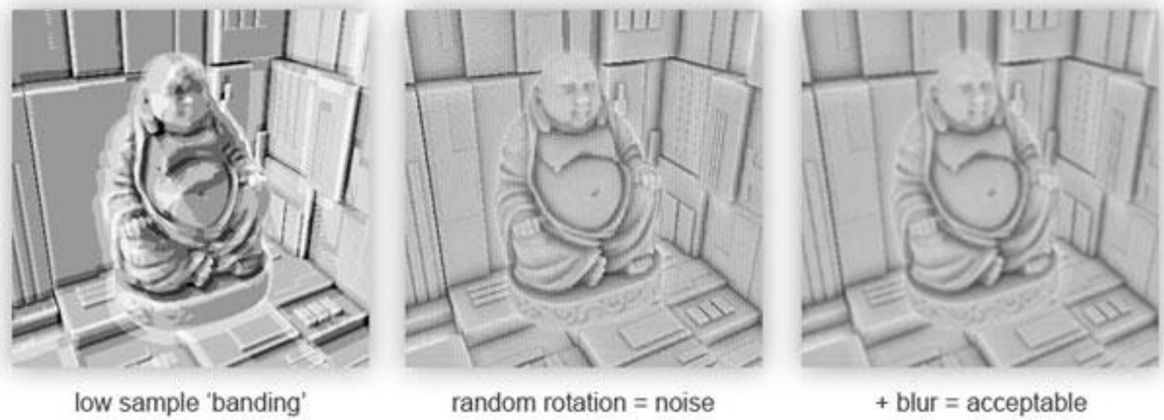
- Method
  - Project the samples back to the depth map from the camera
  - Compare the depth values
  - Average the testing results (**AO**)
  - Modulate the ambient term with **(1 – AO)**



Screen-space Ambient Occlusion (cont.)

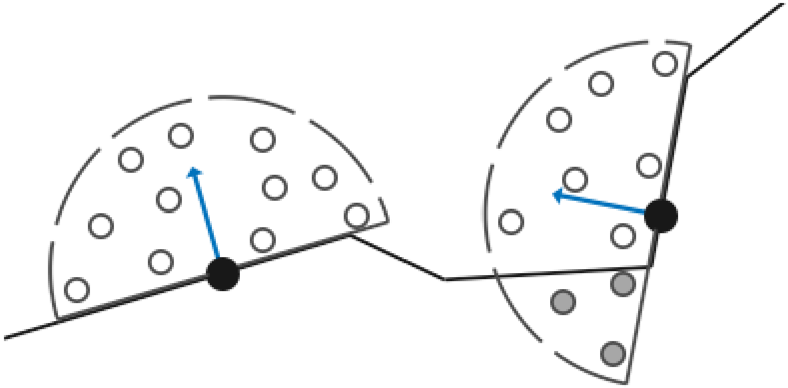
- Strike a balance for the sample count (a compromise between **quality** and **performance**)
- Use some techniques to trade artifacts (banding) with noise, and later removed them by filtering
  - Obtain acceptable results with few samples





Screen-space Ambient Occlusion (cont.)

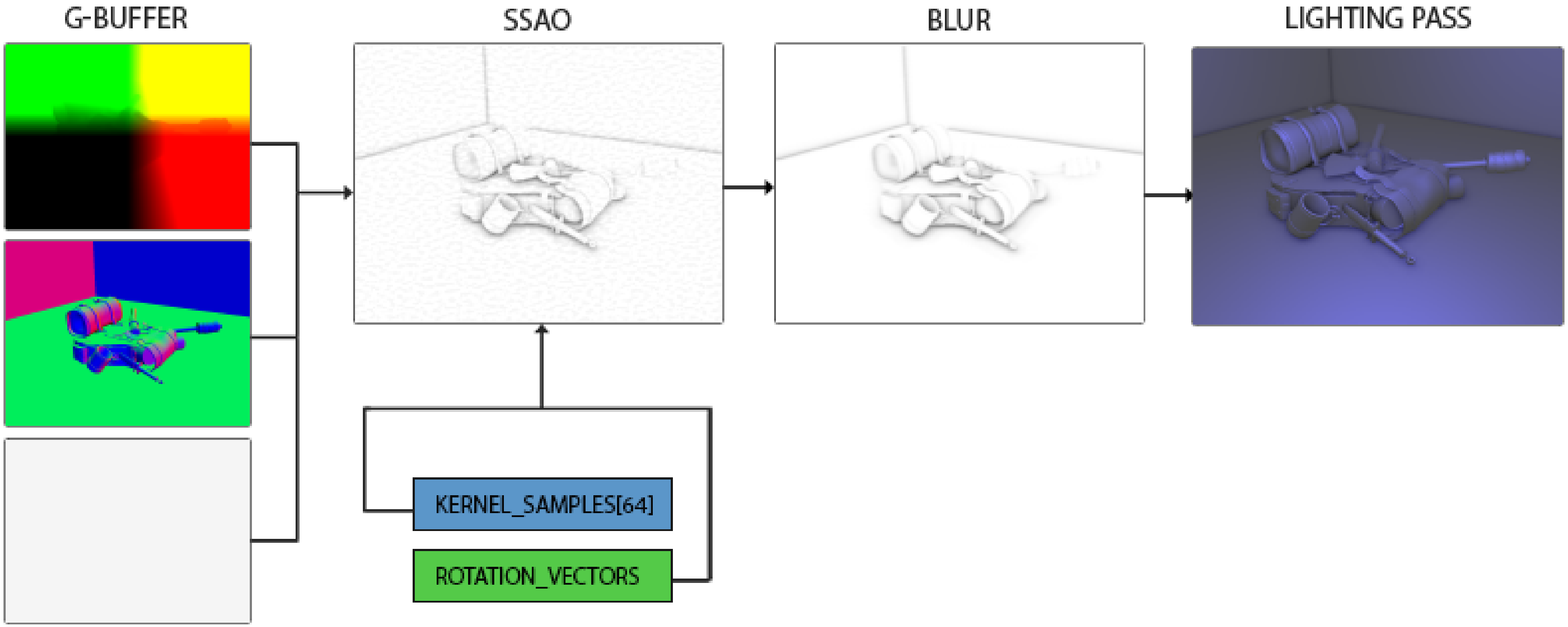
- Problem and improvement
  - Generate samples within a sphere produces results that are too dark
    - Why? Half of the samples are underneath the surface
  - Solution: use **hemisphere** (oriented by **normal**) instead



rotate by the TBN matrix!

Screen-space Ambient Occlusion (cont.)

- An implementation
  - https://learnopengl.com/Advanced-Lighting/SSAO



SSAO in Games

SSAO in Games (cont.)


END OF PRESENTATION
ANY QUESTIONS?