Hard shadows

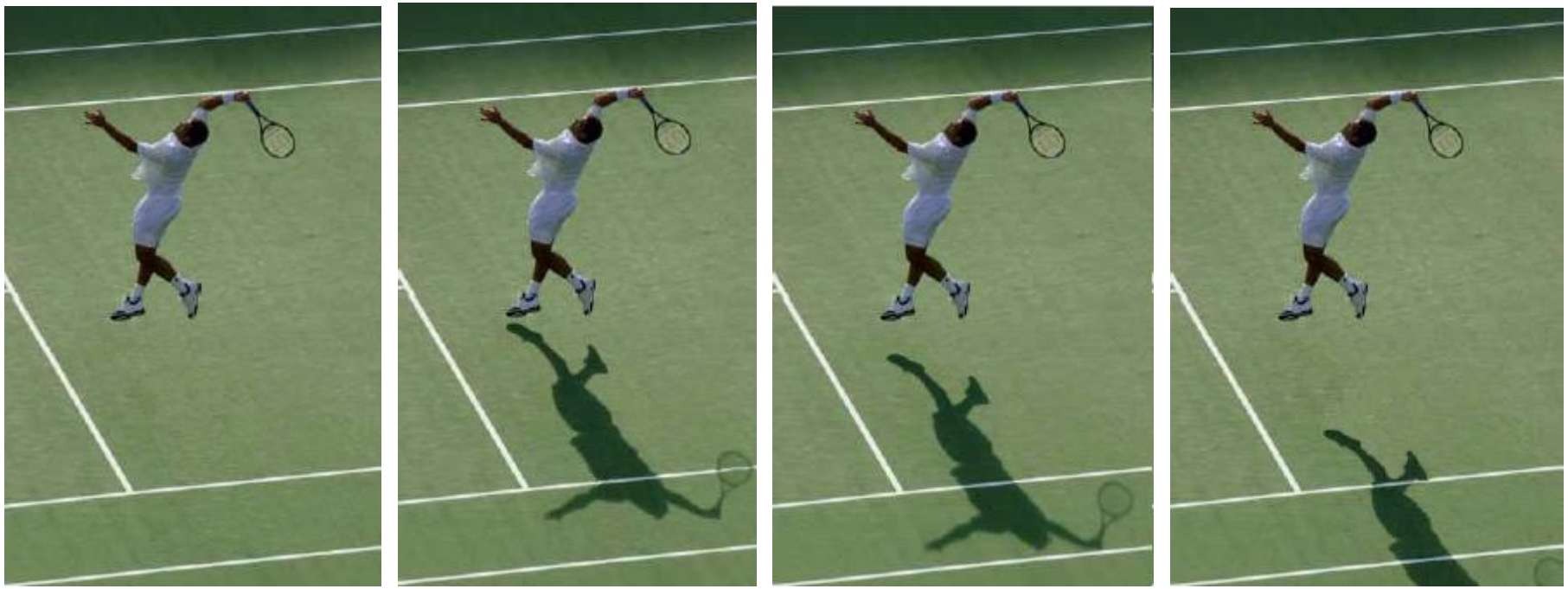
Pere-Pau Vázquez
ViRVIG – UPC

Outline

- Motivation
- Shadow Maps
- Shadow Volumes

Motivation

- Why do we need shadows?
  - How High is the tennis player?



Without Shadow | 3 inches? | 1 foot? | 3 feet?

Motivation

- Why do we need shadows?
  - Shadows are popular in games!

Motivation

- Shadows do significantly contribute to the realism of rendered images
- Global effect which is expensive to compute
- Lot of work done in this area
- Which algorithms are suitable for interactive applications ?

Motivation

- Shadows play an important role in our understanding of 3D geometry:
  Help to understand relative object position and size in a scene

Motivation

- Shadows can also help us understanding the geometry of a complex receiver

Motivation

- Finally, shadows provide useful visual cues that help in understanding the geometry of a complex occluder

Motivation

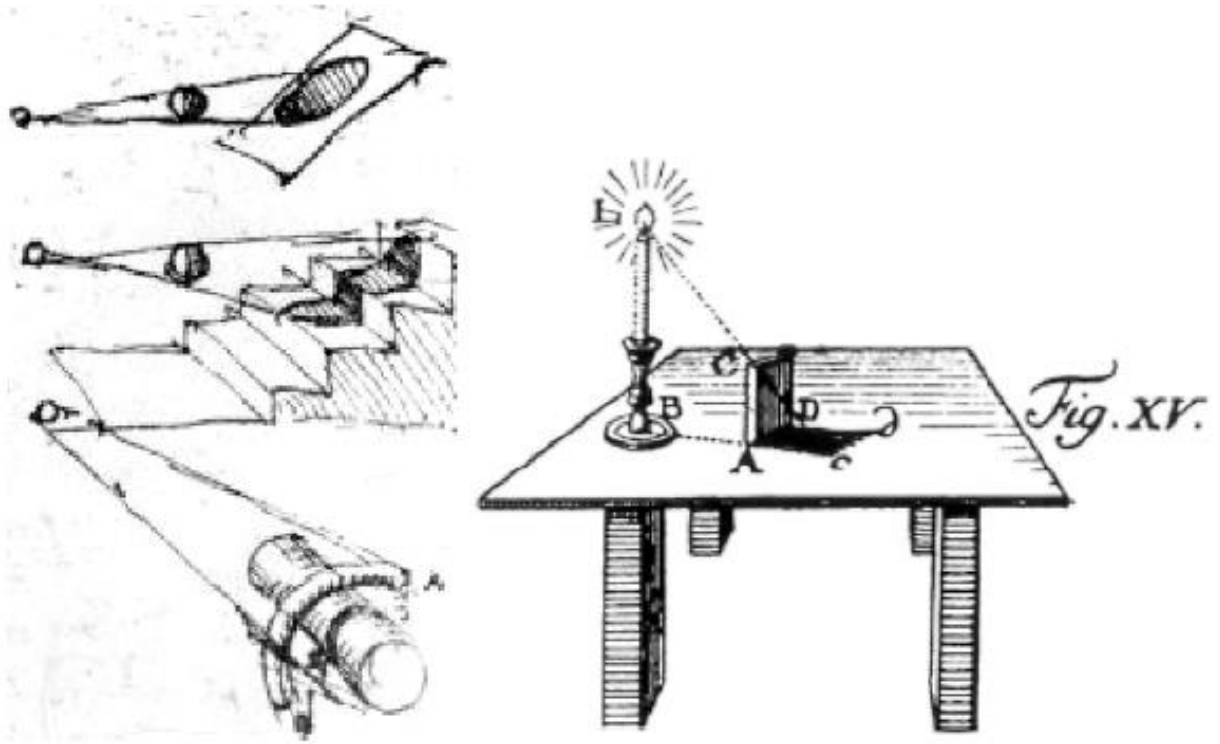


Figure 1: *Left: Study of shadows by Leonardo da Vinci*[48] — *Right: Shadow construction by Lambert*[35].

Motivation

- Efficient methods for computing shadows are still challenging task in computer graphics

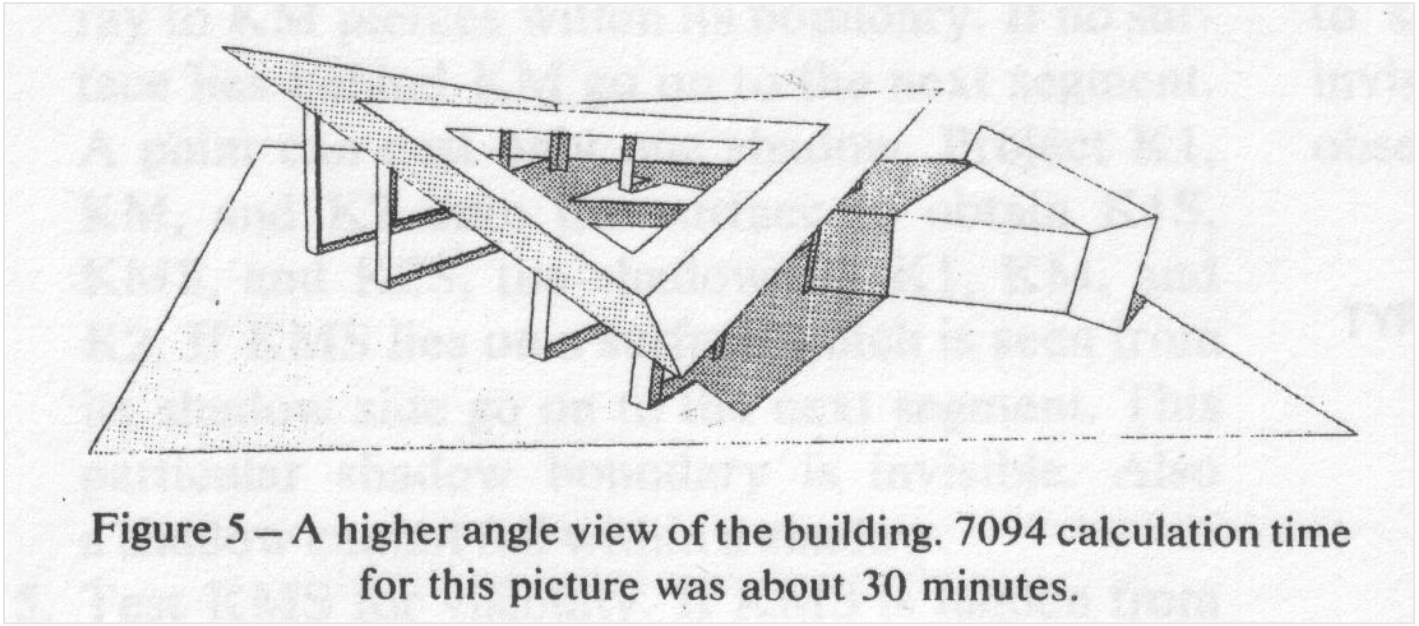
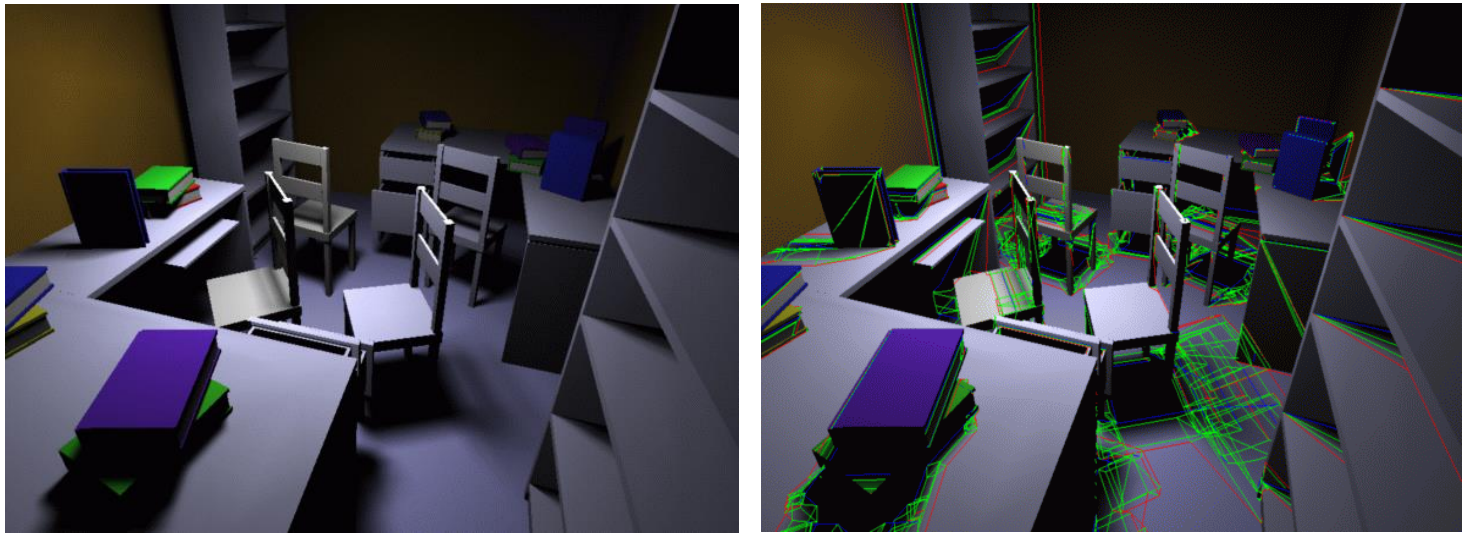


Figure 5 – A higher angle view of the building. 7094 calculation time for this picture was about 30 minutes.

  - Early work on shadow techniques [Arthur Appel, IBM Research, 1968]

Motivation

- Many algorithms proposed to compute shadows



- But only a few techniques are suitable for real-time or interactive applications

Motivation

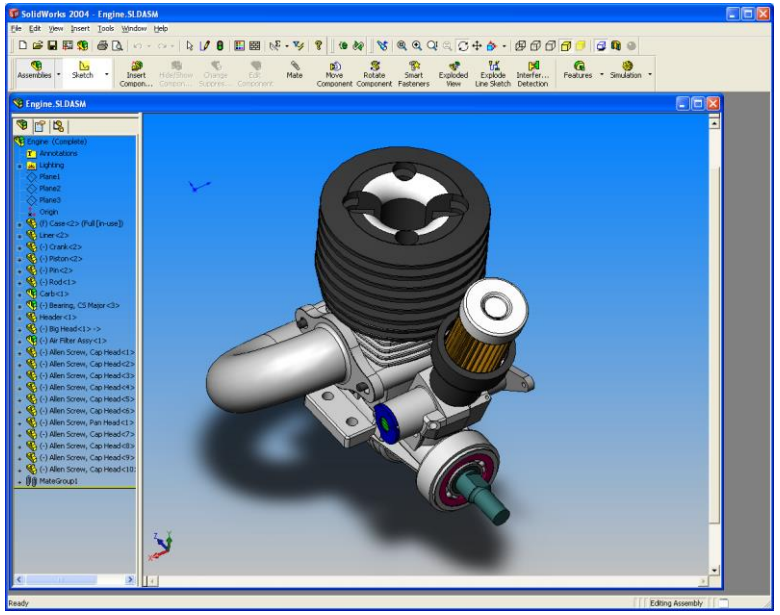
- Refresh rate ~ 30 to 60 fps
  - Games
  - VR Simulations
  - Virtual TV Set

Doom3

- Only hardware-accelerated methods
  - CPU resources dedicated for other tasks
    - Sound, physics, AI, input

Motivation

- Refresh rate ~ 10 to 30 fps
  - CAD/CAM
  - Digital Content Creation (DCC)
  - VR Interaction

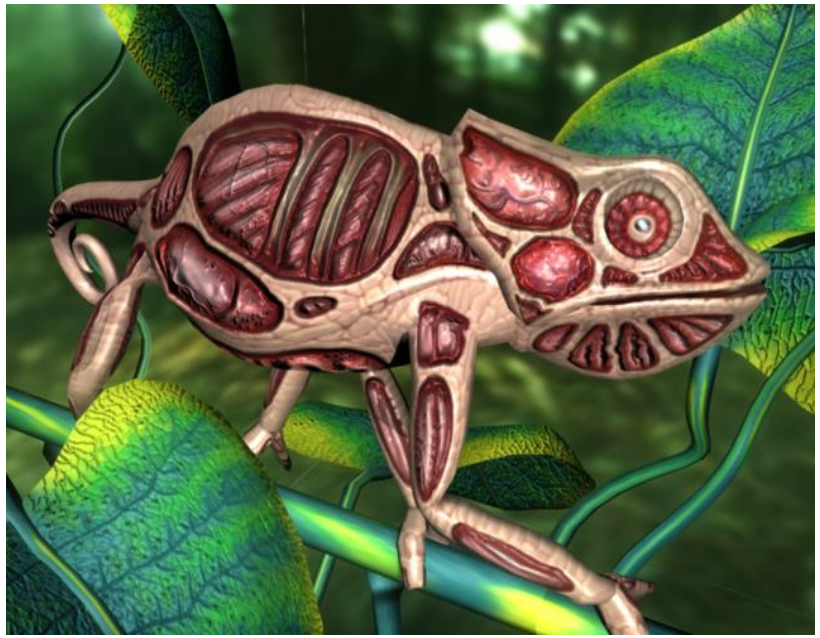


SolidWorks

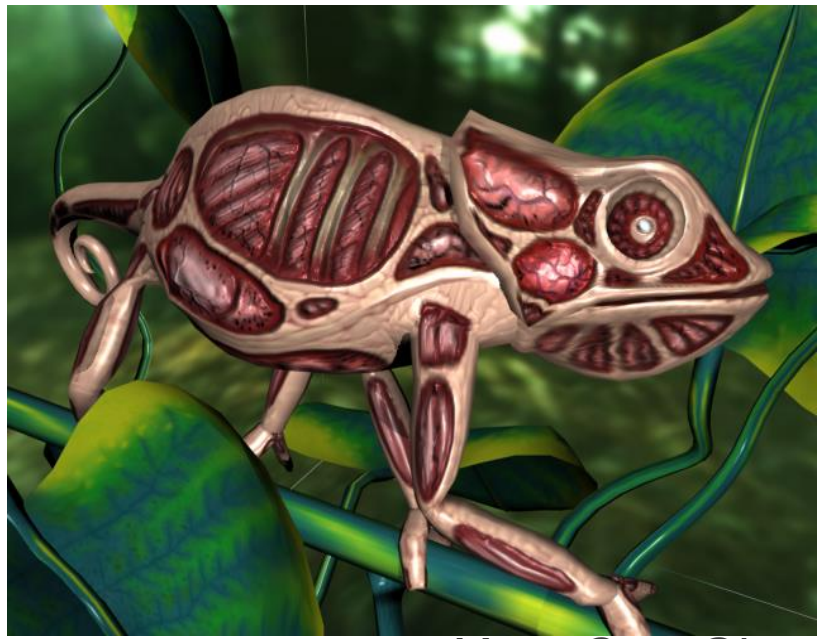
- Hybrid approaches possible
  - Combined CPU & GPU shadow technique

Motivation

- New group of applications
  - Hardware-accelerated rendering methods for offline rendering (preview, production)
    - Example: Alias Maya6 hardware-renderer



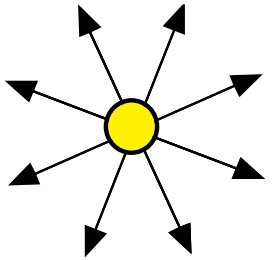
Maya Software

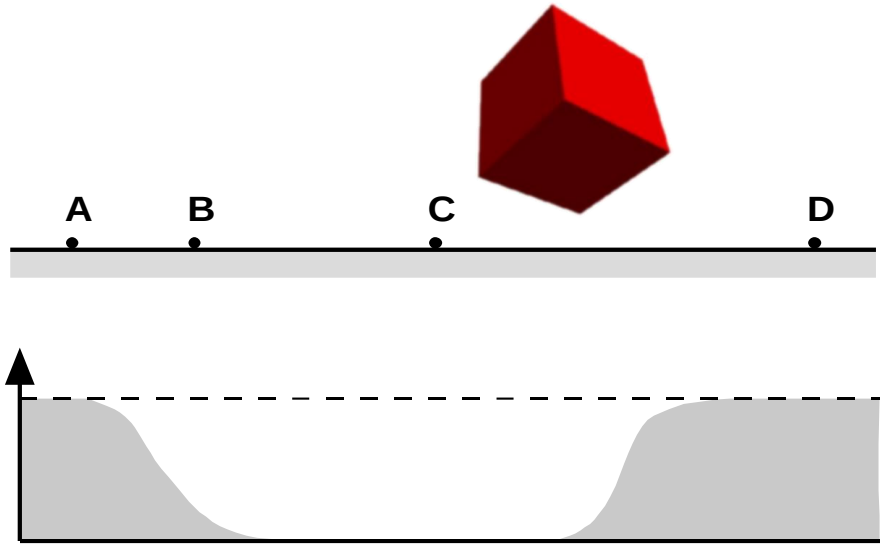


Maya OpenGL

Motivation

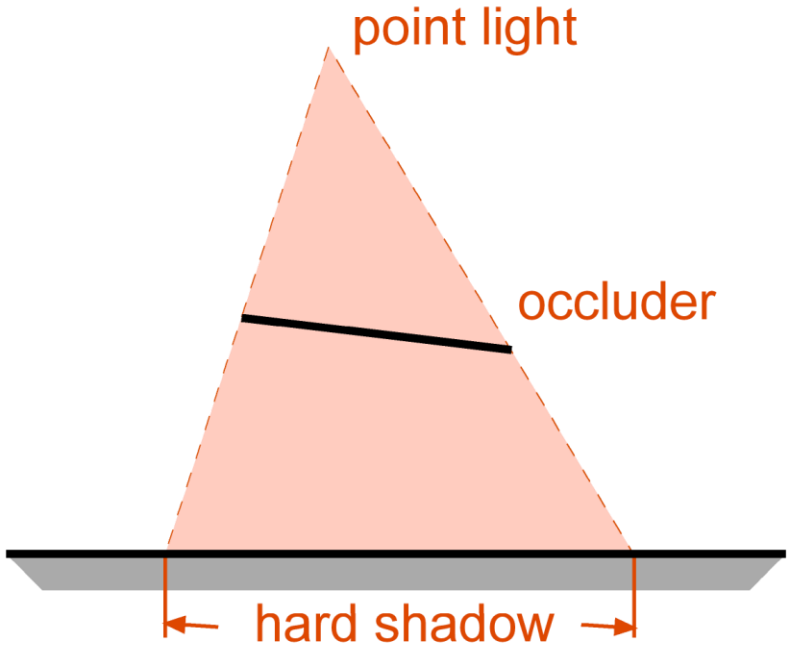
- To compute shadows we need to analyze the spatial relationship of
  - Receiver
  - Light source
  - Occluders
- Result: intensity distribution on receiver



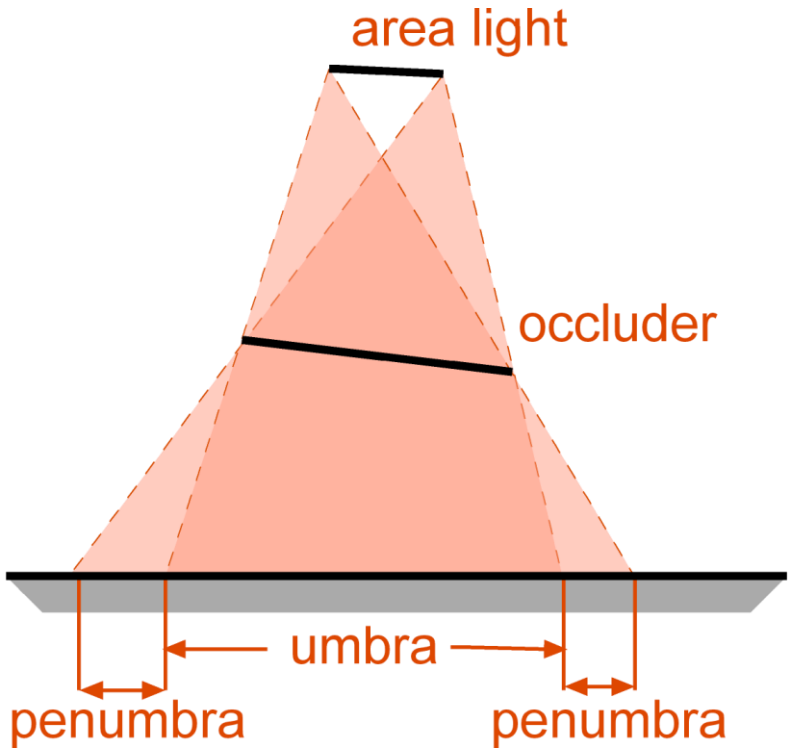


Motivation

- Hard shadows vs. Soft shadows



Hard shadow edges
- unreal (like geometry)
+ easy computation



Soft shadow boundaries
+ realistic
- expensive

Motivation

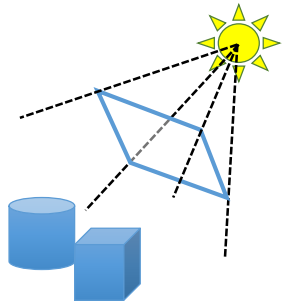
- Ways of thinking about shadows
  - As volumes of space that are dark
    - Shadow Volumes [Franklin Crow 77]
  - As places not seen by a light source looking at the scene
    - Shadow Maps [Lance Williams 78]

Outline

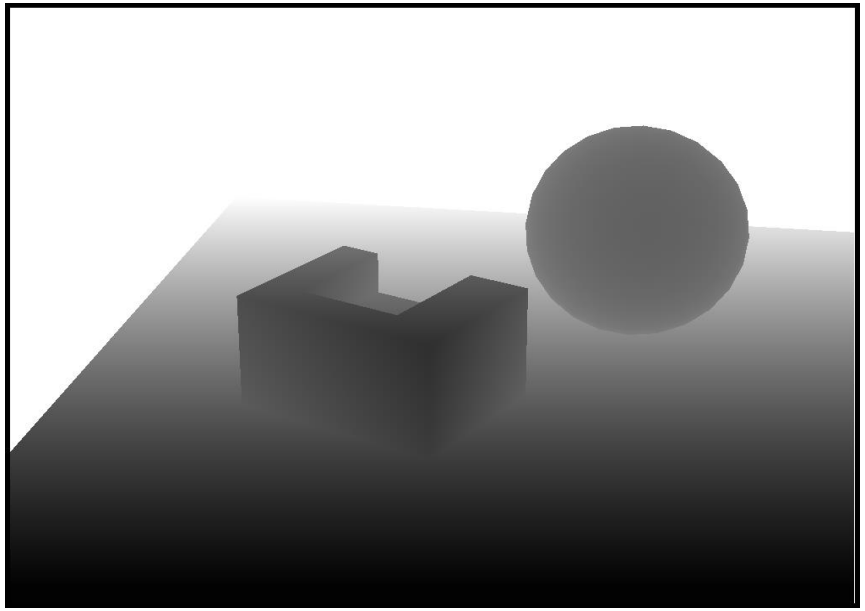
- *Motivation*
- Shadow Maps
- Shadow Volumes

Shadow Maps

Shadow Map (light's view)

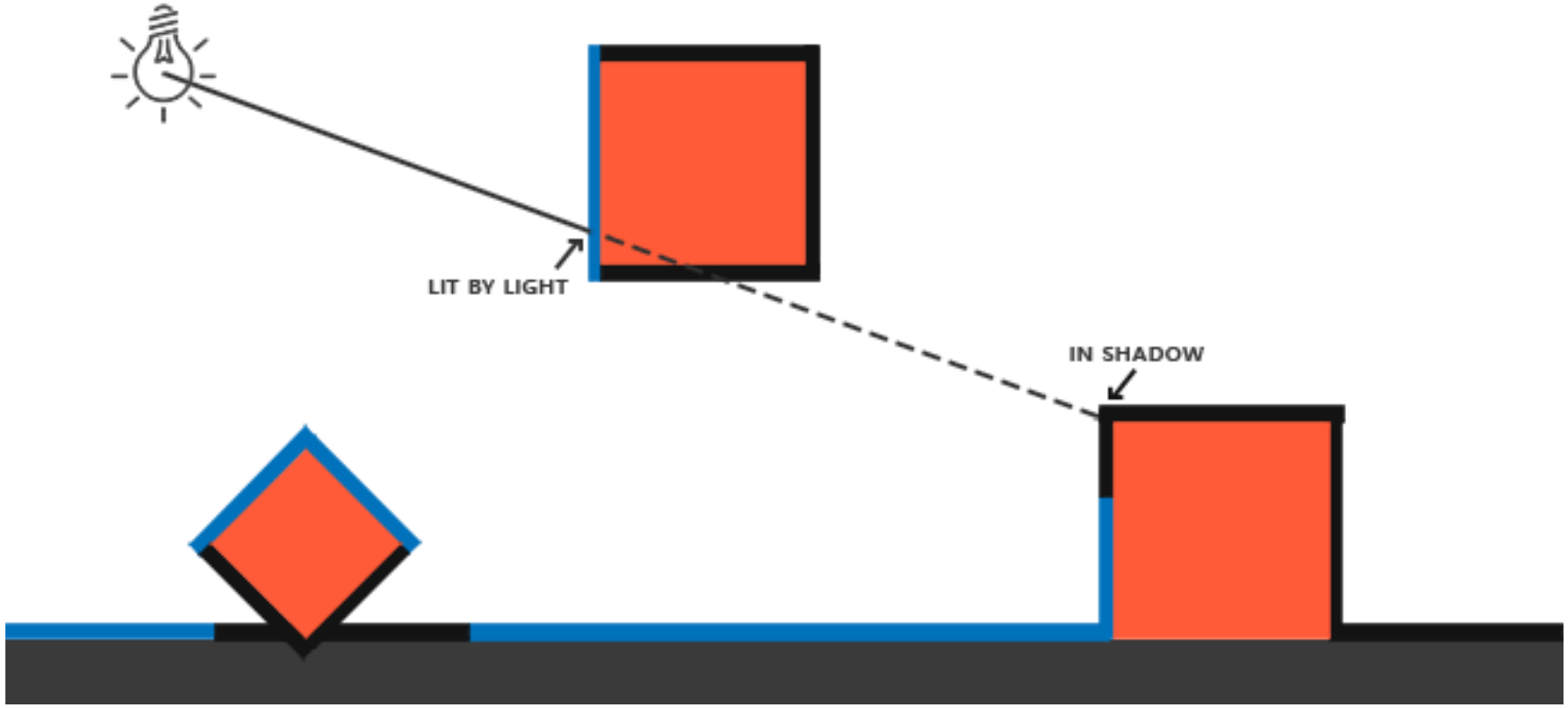


- Basic Algorithm – the simple explanation:
  - Render image from light source
    - Represents geometry in light
  - Render from camera
    - Test if rendered point is
    - visible in the light's view
      - If so -> point in light
      - Else -> point in shadow



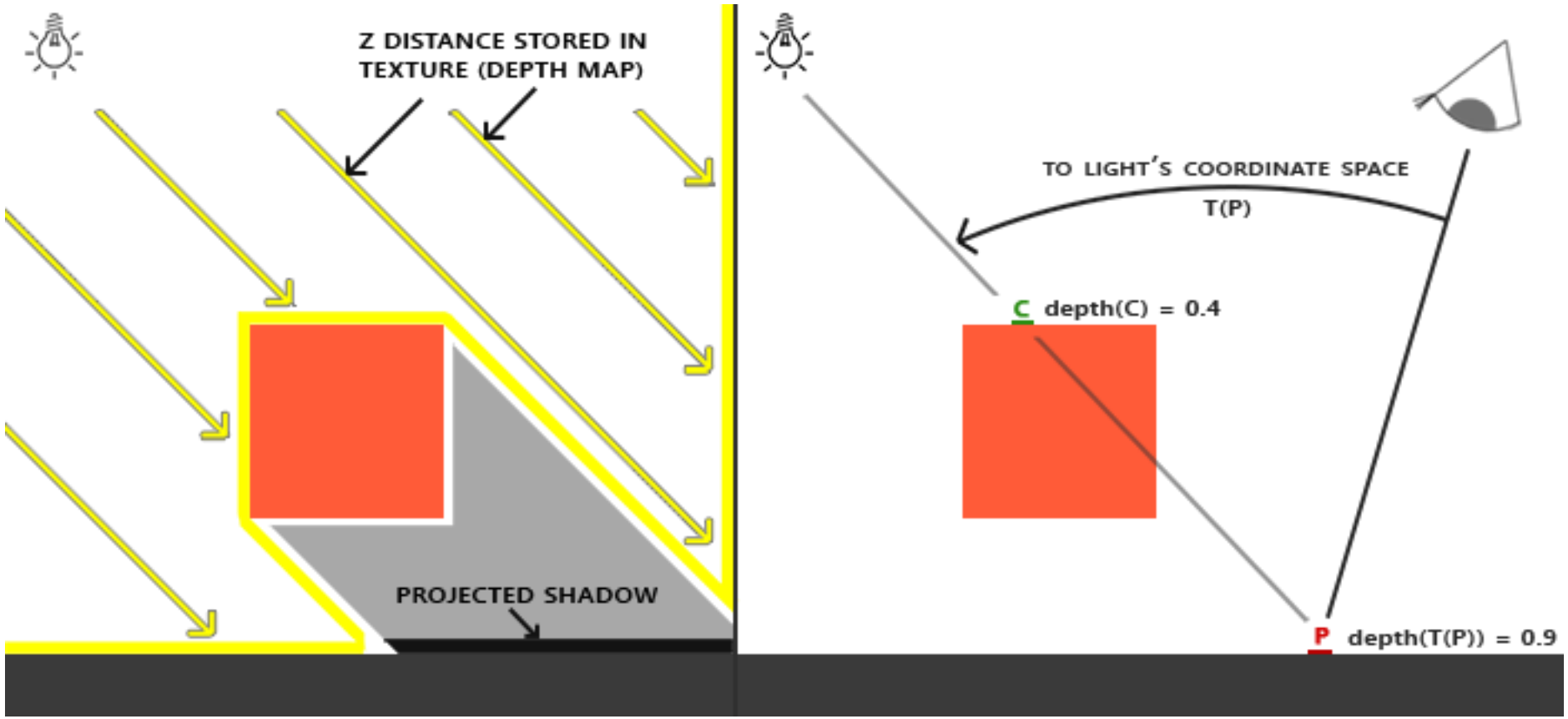
Shadow Maps

- Basic Algorithm – illustrated:



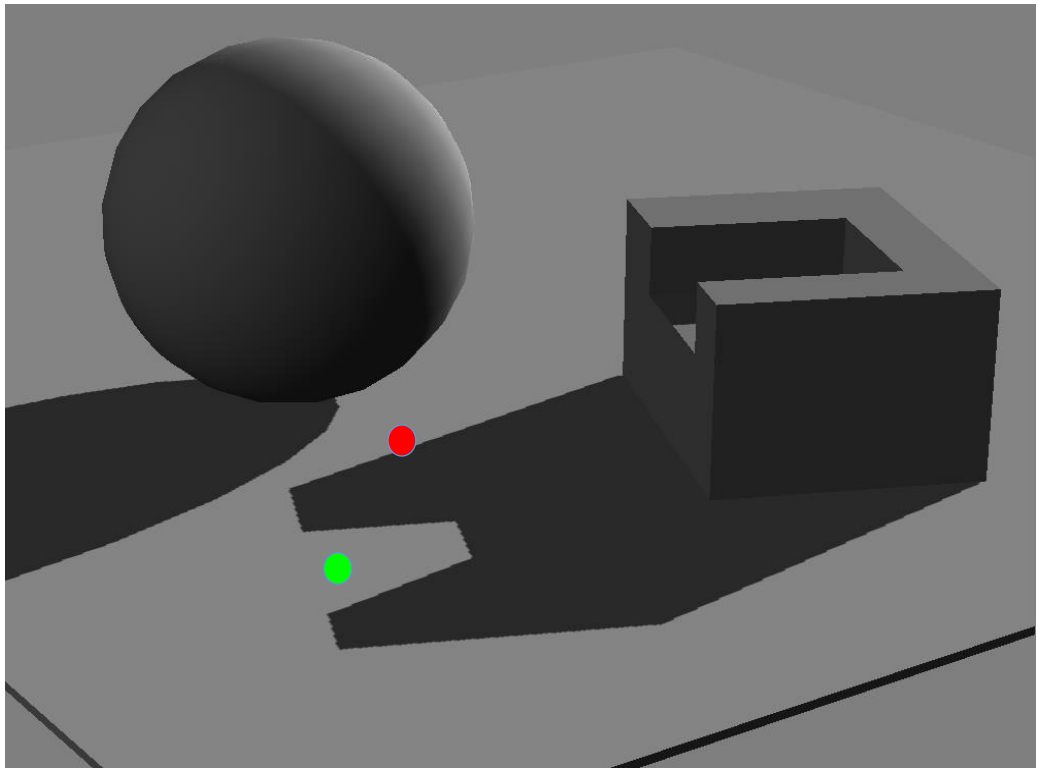
Shadow Maps

- Basic Algorithm – illustrated:

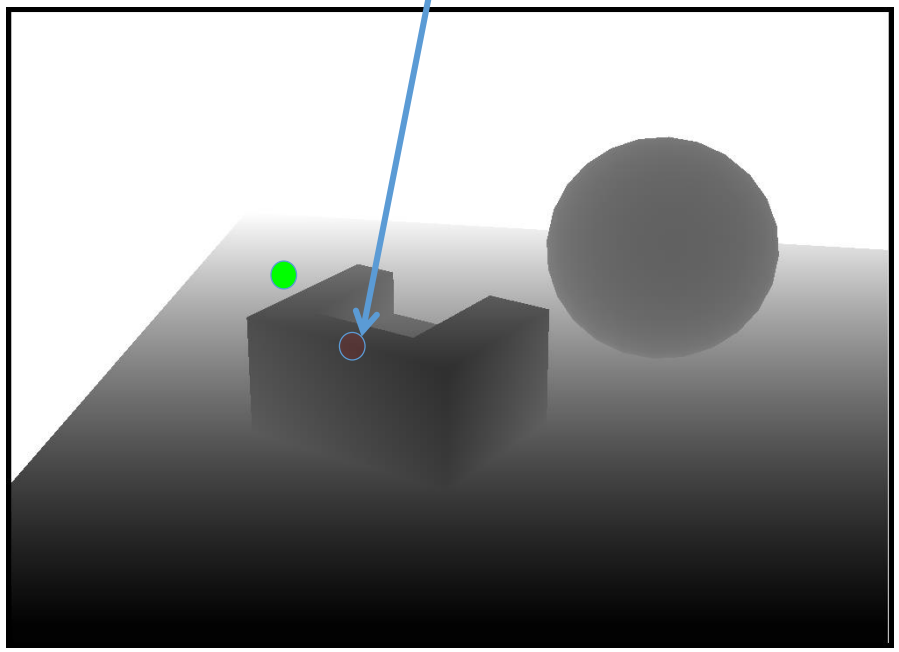


Shadow Maps

Point not represented in shadow map (point is behind box)



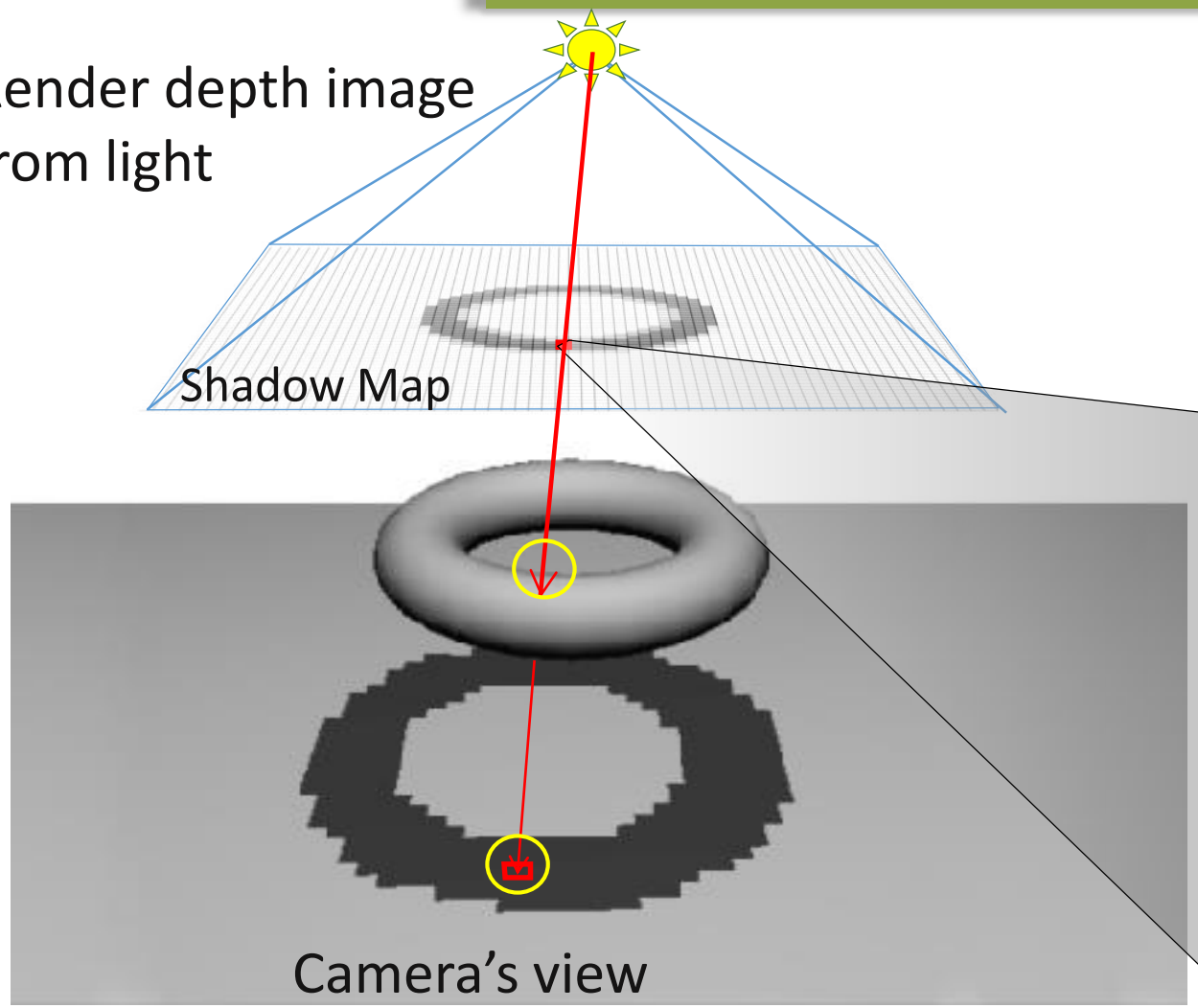
Camera's view



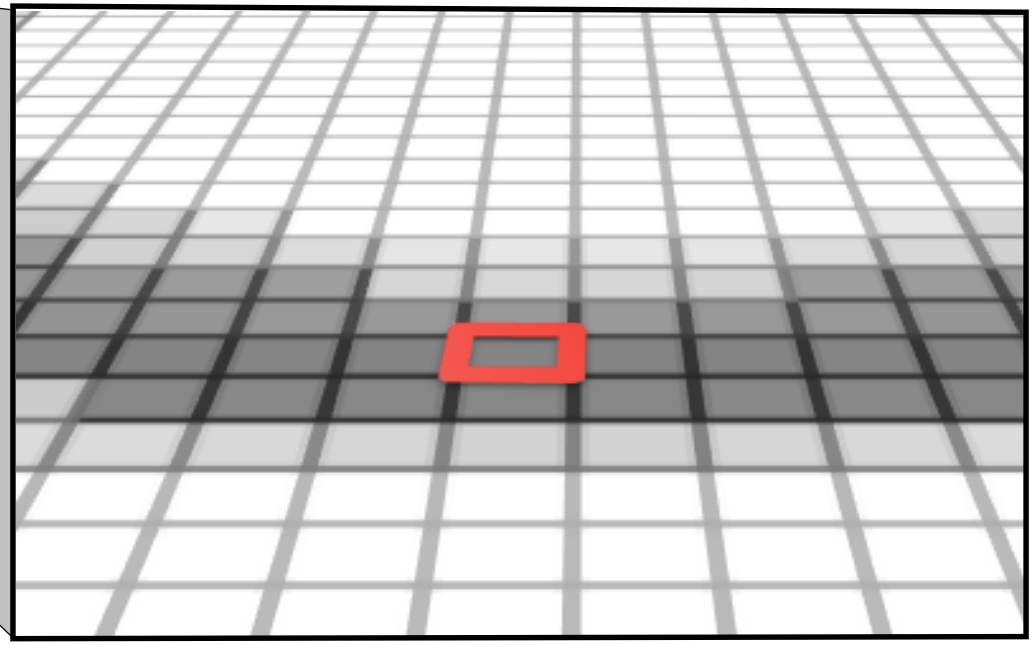
Light's view
(Shadow Map)

Depth Comparison

Render depth image from light



A fragment is in shadow if its depth is greater than the corresponding depth value in the shadow map



Shadow Maps. Implementation

- Initial configuration parameters (background color, depth test…)

```
// Dark blue background
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

// Enable depth test
glEnable(GL_DEPTH_TEST);

// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);

// Cull triangles which normal is not towards the camera
glEnable(GL_CULL_FACE);
```

Shadow Maps. Implementation

- Initial configuration parameters (background color, depth test…)

```
// Dark blue background
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

// Enable depth test
glEnable(GL_DEPTH_TEST);

// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);

// Cull triangles which normal is not towards the camera
glEnable(GL_CULL_FACE);
```

Shadow Maps. Implementation

- Initial configuration parameters (background color, depth test…)

```
// Dark blue background
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

// Enable depth test
glEnable(GL_DEPTH_TEST);

// Accept fragment if it closer to the camera than the former one
glDepthFunc(GL_LESS);

// Cull triangles which normal is not towards the camera
glEnable(GL_CULL_FACE);
```

Shadow Maps. Implementation

- Generate the framebuffer object (optionally multiple textures)

```
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

Shadow Maps. Implementation

- Generate the depth texture

```
GLuint depthTexture;
glGenTextures(1, &depthTexture);
glBindTexture(GL_TEXTURE_2D, depthTexture);
```

Shadow Maps. Implementation

- Configure the depth texture

```
// Creating the internal memory and data structures with a void texture
glTexImage2D(GL_TEXTURE_2D, 0,GL_DEPTH_COMPONENT16, 1024, 1024, 0,GL_DEPTH_COMPONENT, GL_FLOAT, 0);

// Configuration parameters (for posterior access)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_R_TO_TEXTURE);

// Attach the depth
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTexture, 0);
```

Shadow Maps. Implementation

- Ensure we render with no color
  - Typically 2x-4x faster on modern GPUs
  - Can render with stencil too
    - Depth+stencil are generated “at the same time”
  - Usually called “depth-only pass”

```
glDrawBuffer(GL_NONE);
```

Shadow Maps. Implementation

- Create shaders & pass parameters

```
// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders( "ShadowMapping_SimpleVersion.vertexshader", "ShadowMapping_SimpleVersion.fragmentshader" );

// Get a handle for our "myTextureSampler" uniform
GLuint TextureID  = glGetUniformLocation(programID, "myTextureSampler");

// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
GLuint DepthBiasID = glGetUniformLocation(programID, "DepthBiasMVP");
GLuint ShadowMapID = glGetUniformLocation(programID, "shadowMap");
```

Shadow Maps. Implementation

- Create shaders & pass parameters

```
// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders( "ShadowMapping_SimpleVersion.vertexshader", "ShadowMapping_SimpleVersion.fragmentshader" );
```

```
// Get a handle for our "myTextureSampler" uniform
GLuint TextureID  = glGetUniformLocation(programID, "myTextureSampler");

// Get a handle for our "MVP" uniform
GLuint MatrixID = glGetUniformLocation(programID, "MVP");
GLuint DepthBiasID = glGetUniformLocation(programID, "DepthBiasMVP");
GLuint ShadowMapID = glGetUniformLocation(programID, "shadowMap");
```

Shadow Maps. Implementation

- Render from the light source

```
// Compute the MVP matrix from the light's point of view
glm::vec3 lightPos(5, 20, 20);
glm::mat4 depthProjectionMatrix = glm::perspective<float>(45.0f, 1.0f, 2.0f, 50.0f);
glm::mat4 depthViewMatrix = glm::lookAt(lightPos, lightPos-lightInvDir, glm::vec3(0,1,0));

glm::mat4 depthModelMatrix = glm::mat4(1.0);
glm::mat4 depthMVP = depthProjectionMatrix * depthViewMatrix * depthModelMatrix;

// Send our transformation to the currently bound shader, in the "MVP" uniform
glUniformMatrix4fv(depthMatrixID, 1, GL_FALSE, &depthMVP[0][0]);

drawGeometryDepth(indices, vertexbuffer, elementbuffer);
```

Shadow Maps. Implementation

- Render from the light source

```
// Compute the MVP matrix from the light's point of view
glm::vec3 lightPos(5, 20, 20);
glm::mat4 depthProjectionMatrix = glm::perspective<float>(45.0f, 1.0f, 2.0f, 50.0f);
glm::mat4 depthViewMatrix = glm::lookAt(lightPos, lightPos-lightInvDir, glm::vec3(0,1,0));

glm::mat4 depthModelMatrix = glm::mat4(1.0);
glm::mat4 depthMVP = depthProjectionMatrix * depthViewMatrix * depthModelMatrix;

// Send our transformation to the currently bound shader, in the "MVP" uniform
glUniformMatrix4fv(depthMatrixID, 1, GL_FALSE, &depthMVP[0][0]);

drawGeometryDepth(indices, vertexbuffer, elementbuffer);
```

Shadow Maps. Implementation

- Render from the light source. Shaders:

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

// Values that stay constant for the whole mesh.
uniform mat4 depthMVP;

void main(){
  gl_Position =  depthMVP * vec4(vertexPosition_modelspace,1);
}
```

Shadow Maps. Implementation

- Render from the light source. Shaders:

```
#version 330 core

// Ouput data
layout(location = 0) out float fragmentdepth;


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

Shadow Maps. Implementation

- Render from the observer's view

```
// Render to the screen
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,windowWidth,windowHeight); // Render on the whole framebuffer, complete from the lower left corn

glEnable(GL_CULL_FACE);
glCullFace(GL_BACK); // Cull back-facing triangles -> draw only front-facing triangles

// Clear the screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Shadow Maps. Implementation

- Render from the observer's view

```
// Use our shader
glUseProgram(programID);

// Compute the MVP matrix from keyboard and mouse input
computeMatricesFromInputs();
glm::mat4 ProjectionMatrix = getProjectionMatrix();
glm::mat4 ViewMatrix = getViewMatrix();
//ViewMatrix = glm::lookAt(glm::vec3(14,6,4), glm::vec3(0,1,0), glm::vec3(0,1,0));
glm::mat4 ModelMatrix = glm::mat4(1.0);
glm::mat4 MVP = ProjectionMatrix * ViewMatrix * ModelMatrix;
```

Shadow Maps. Implementation

- Render from the observer's view

```
glm::mat4 biasMatrix(
  0.5, 0.0, 0.0, 0.0,
  0.0, 0.5, 0.0, 0.0,
  0.0, 0.0, 0.5, 0.0,
  0.5, 0.5, 0.5, 1.0
);

glm::mat4 depthBiasMVP = biasMatrix*depthMVP;
```

Shadow Maps. Implementation

- Render from the observer's view

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(DepthBiasID, 1, GL_FALSE, &depthBiasMVP[0][0]);

// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);

drawGeometry(indices, vertexbuffer, elementbuffer, uvbuffer, normalbuffer, Texture);
```

Shadow Maps. Implementation

- Render from the observer's view

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(DepthBiasID, 1, GL_FALSE, &depthBiasMVP[0][0]);

// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);

drawGeometry(indices, vertexbuffer, elementbuffer, uvbuffer, normalbuffer, Texture);
```

Shadow Maps. Implementation

- Render from the observer's view

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(DepthBiasID, 1, GL_FALSE, &depthBiasMVP[0][0]);

// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);

drawGeometry(indices, vertexbuffer, elementbuffer, uvbuffer, normalbuffer, Texture);
```

Shadow Maps. Implementation

- Render from the observer's view

```
// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(DepthBiasID, 1, GL_FALSE, &depthBiasMVP[0][0]);

// Bind our texture in Texture Unit 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, Texture);
// Set our "myTextureSampler" sampler to use Texture Unit 0
glUniform1i(TextureID, 0);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);

drawGeometry(indices, vertexbuffer, elementbuffer, uvbuffer, normalbuffer, Texture);
```

Shadow Maps. Implementation

- Render from the observer's view

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glUniform1i(ShadowMapID, 1);

drawGeometry(indices, vertexbuffer, elementbuffer, uvbuffer, normalbuffer, Texture);



// Swap buffers
glfwSwapBuffers(window);
glfwPollEvents();
```

Shadow Maps. Implementation

- Render from the observer's view. Vertex shader

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

// Output data ; will be interpolated for each fragment.
out vec2 UV;
out vec4 ShadowCoord;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 DepthBiasMVP;

```

Shadow Maps. Implementation

- Render from the observer's view . Vertex shader

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

// Output data ; will be interpolated for each fragment.
out vec2 UV;
out vec4 ShadowCoord;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 DepthBiasMVP;

```

Shadow Maps. Implementation

- Render from the observer's view . Vertex shader

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

// Output data ; will be interpolated for each fragment.
out vec2 UV;
out vec4 ShadowCoord;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 DepthBiasMVP;
```

Shadow Maps. Implementation

- Render from the observer's view . Vertex shader

```
17  void main(){
18
19      // Output position of the vertex, in clip space : MVP * position
20      gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
21
22      ShadowCoord = DepthBiasMVP * vec4(vertexPosition_modelspace,1);
23
24      // UV of the vertex. No special space for this one.
25      UV = vertexUV;
26  }
27
```

Shadow Maps. Implementation

- Render from the observer's view . Vertex shader

```
17  void main(){
18
19    // Output position of the vertex, in clip space : MVP * position
20    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
21
22    ShadowCoord = DepthBiasMVP * vec4(vertexPosition_modelspace,1);
23
24    // UV of the vertex. No special space for this one.
25    UV = vertexUV;
26  }
27
```

Shadow Maps. Implementation

- Render from the observer's view. Vertex shader

```
17  void main(){
18
19    // Output position of the vertex, in clip space : MVP * position
20    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
21
22    ShadowCoord = DepthBiasMVP * vec4(vertexPosition_modelspace,1);
23
24    // UV of the vertex. No special space for this one.
25    UV = vertexUV;
26  }
27
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
1   #version 330 core
2
3   // Interpolated values from the vertex shaders
4   in vec2 UV;
5   in vec4 ShadowCoord;
6
7   // Ouput data
8   layout(location = 0) out vec3 color;
9
10  // Values that stay constant for the whole mesh.
11  uniform sampler2D myTextureSampler;
12  uniform sampler2DShadow shadowMap;
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
1   #version 330 core
2
3   // Interpolated values from the vertex shaders
4   in vec2 UV;
5   in vec4 ShadowCoord;
6
7   // Ouput data
8   layout(location = 0) out vec3 color;
9
10  // Values that stay constant for the whole mesh.
11  uniform sampler2D myTextureSampler;
12  uniform sampler2DShadow shadowMap;
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
#version 330 core

// Interpolated values from the vertex shaders
in vec2 UV;
in vec4 ShadowCoord;

// Ouput data
layout(location = 0) out vec3 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;
uniform sampler2DShadow shadowMap;
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
14  void main(){
15
16      // Light emission properties
17      vec3 LightColor = vec3(1,1,1);
18
19      // Material properties
20      vec3 MaterialDiffuseColor = texture( myTextureSampler, UV ).rgb;
21
22      float visibility = texture( shadowMap, vec3(ShadowCoord.xy, (ShadowCoord.z)/ShadowCoord.w) );
23
24      color = visibility * MaterialDiffuseColor * LightColor;
25
26  }
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
14  void main(){
15
16    // Light emission properties
17    vec3 LightColor = vec3(1,1,1);
18
19    // Material properties
20    vec3 MaterialDiffuseColor = texture( myTextureSampler, UV ).rgb;
21
22    float visibility = texture( shadowMap, vec3(ShadowCoord.xy, (ShadowCoord.z)/ShadowCoord.w) );
23
24    color = visibility * MaterialDiffuseColor * LightColor;
25
26  }
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

```
14  void main(){
15
16    // Light emission properties
17    vec3 LightColor = vec3(1,1,1);
18
19    // Material properties
20    vec3 MaterialDiffuseColor = texture( myTextureSampler, UV ).rgb;
21
22    float visibility = texture( shadowMap, vec3(ShadowCoord.xy, (ShadowCoord.z)/ShadowCoord.w) );
23
24    color = visibility * MaterialDiffuseColor * LightColor;
25
26  }
```

Shadow Maps. Implementation

- Render from the observer's view. Fragment shader

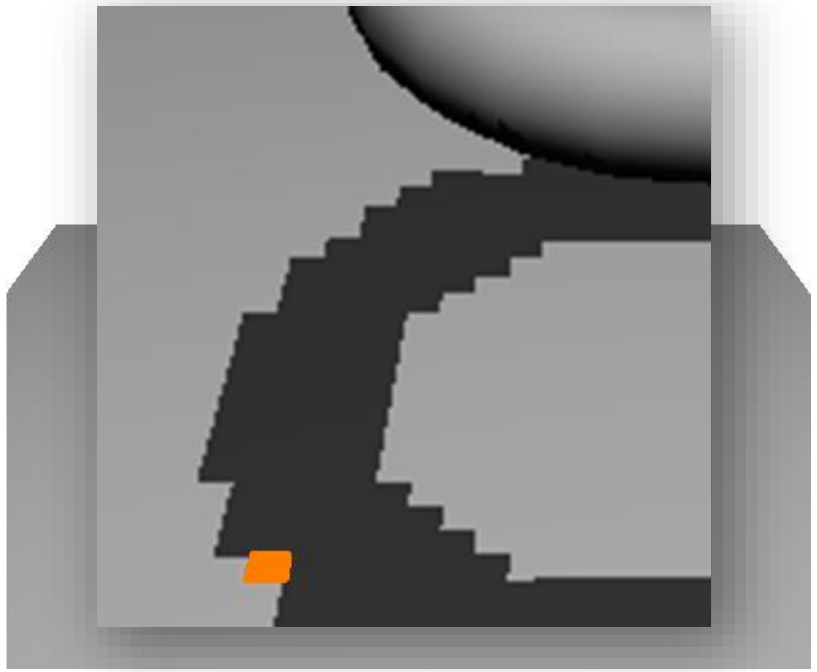
```
14  void main(){
15
16    // Light emission properties
17    vec3 LightColor = vec3(1,1,1);
18
19    // Material properties
20    vec3 MaterialDiffuseColor = texture( myTextureSampler, UV ).rgb;
21
22    float visibility = texture( shadowMap, vec3(ShadowCoord.xy, (ShadowCoord.z)/ShadowCoord.w) );
23
24    color = visibility * MaterialDiffuseColor * LightColor;
25
26  }
```

Shadow Maps

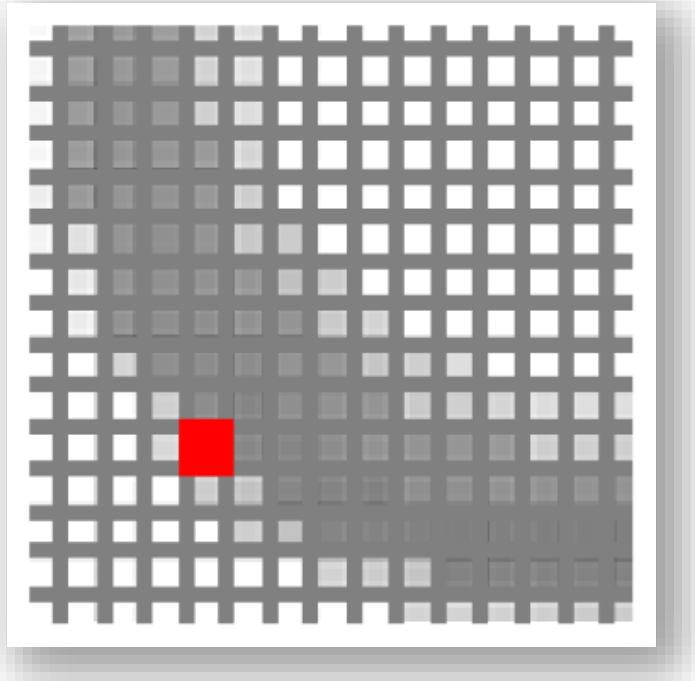
- Pros
  - Very efficient: “This is as fast as it gets”
- Cons...

Shadow Maps - Problems

- Low Shadow Map resolution results in jagged shadows



from viewpoint



from light

Shadow Maps - Problems

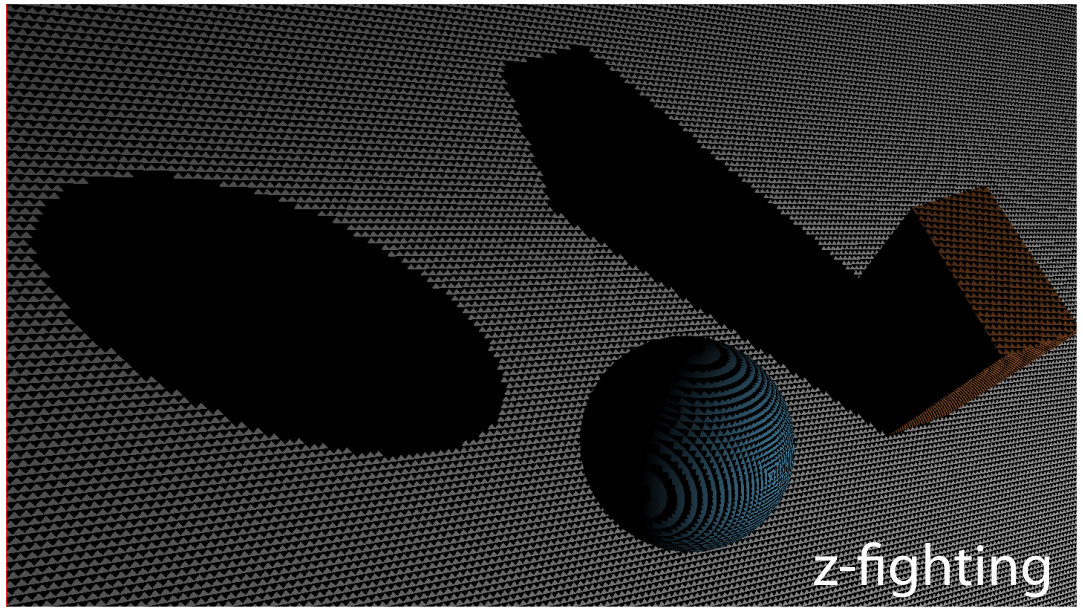
In addition:

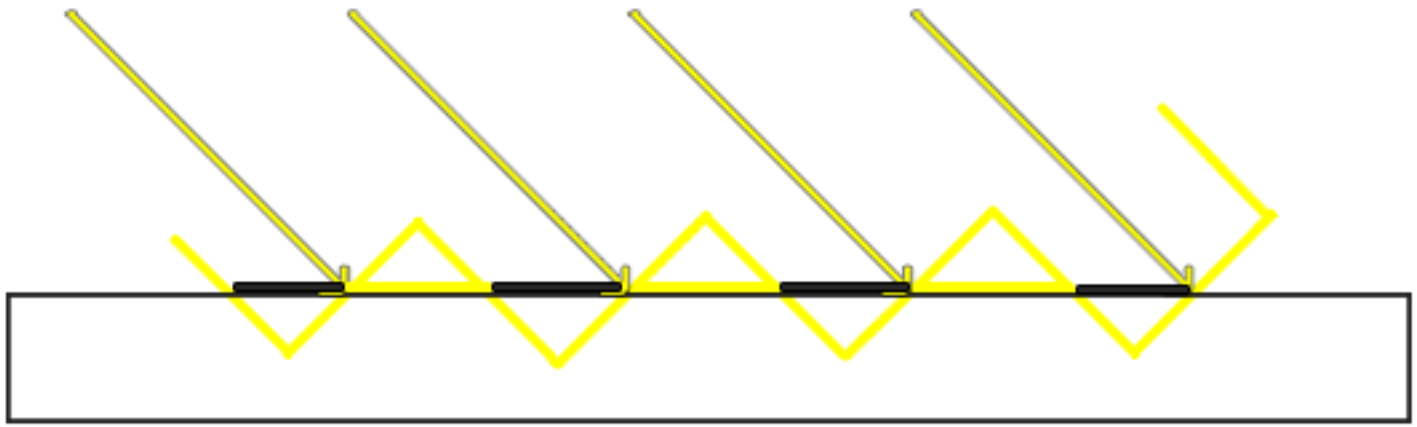
- A tolerance threshold (bias) needs to be tuned **for each scene** for the depth comparison

Shadow Maps - Problems

Typical problems:

- Shadow acne, due to z-fighting
  - Light position and view position are very similar, floating point operations lead to z-fighting

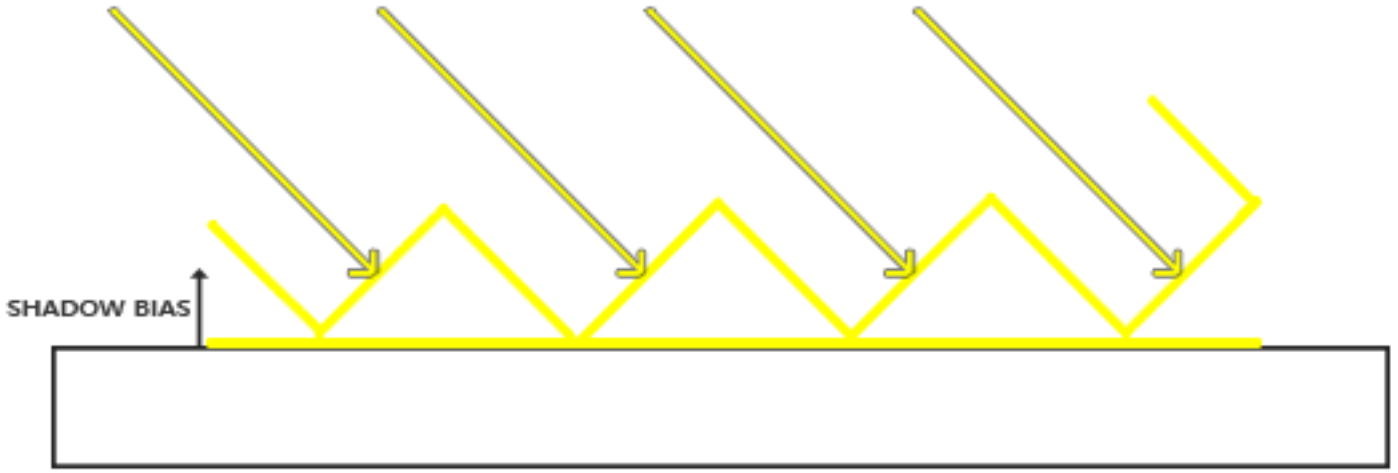




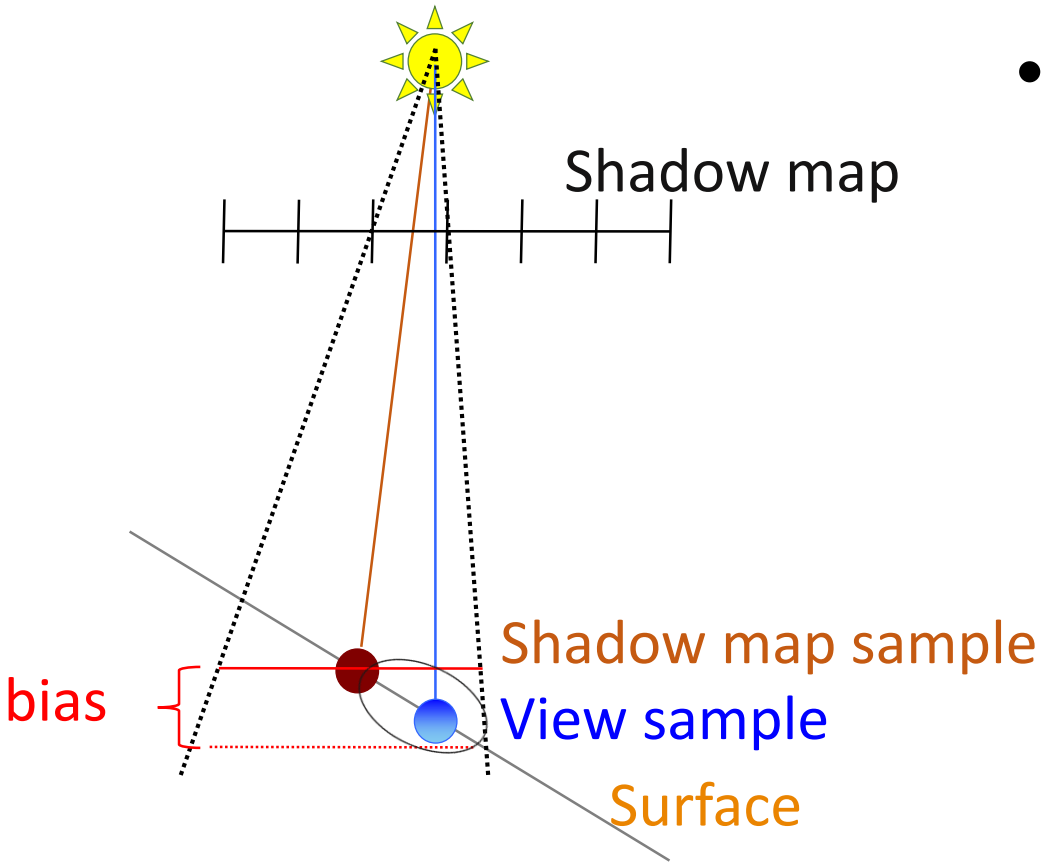
Shadow Maps - Problems

Typical problems:

- Shadow acne, can be solved using shadow bias (offset the depth of the surface – or the shadow map – a small bias to avoid this

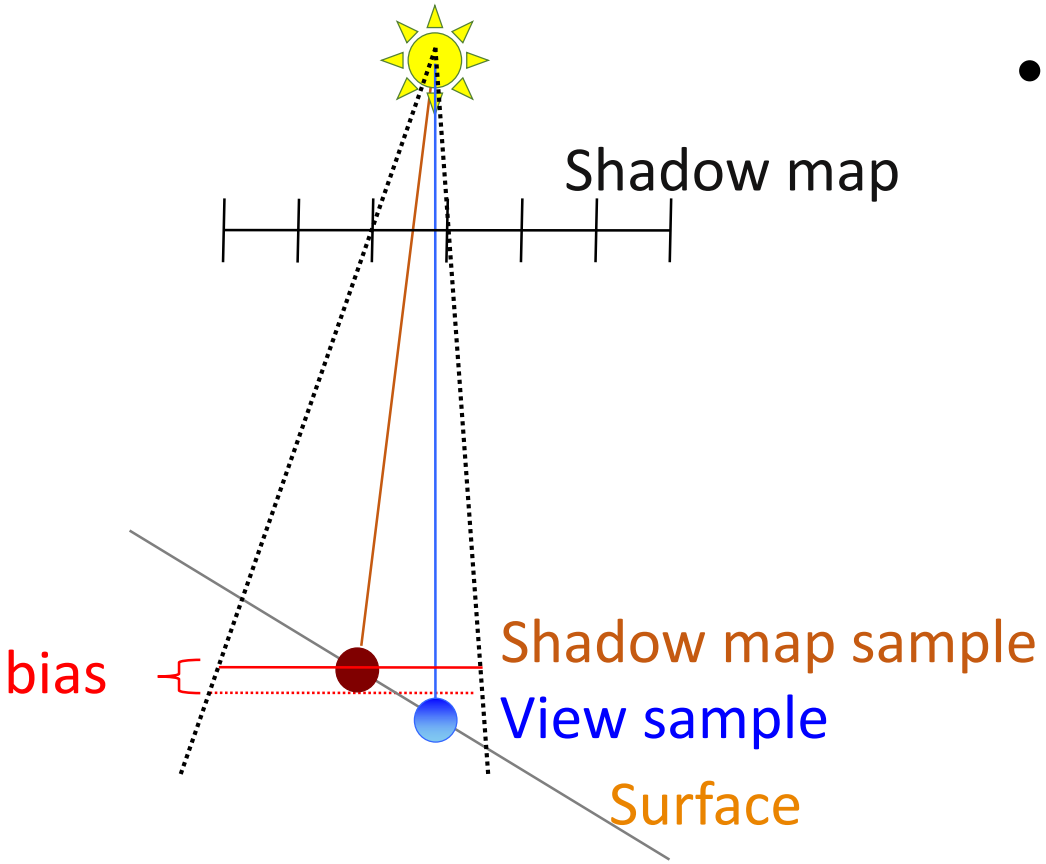


Bias

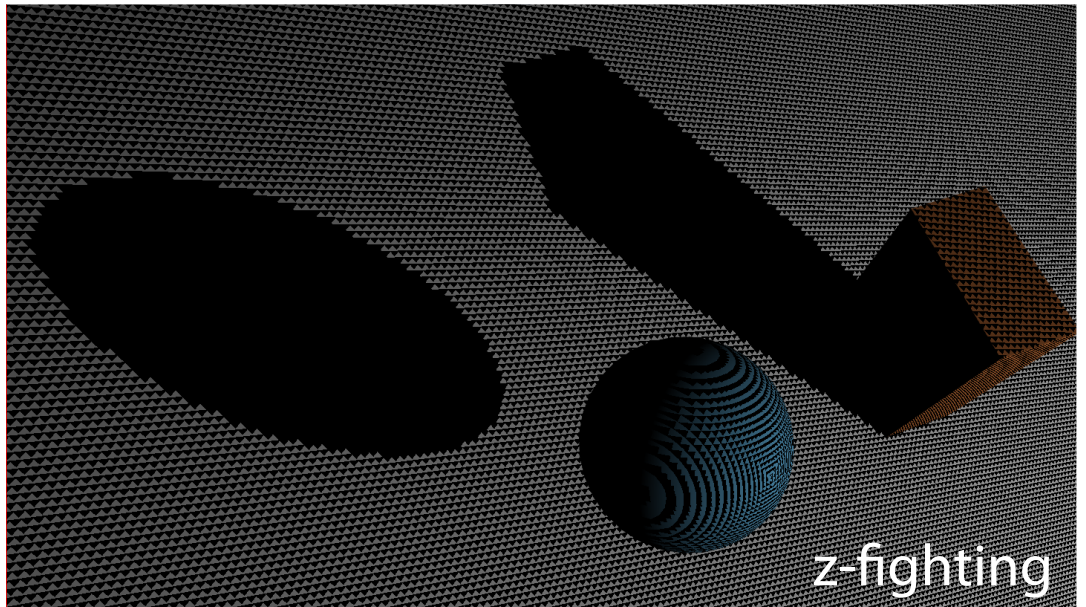


- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

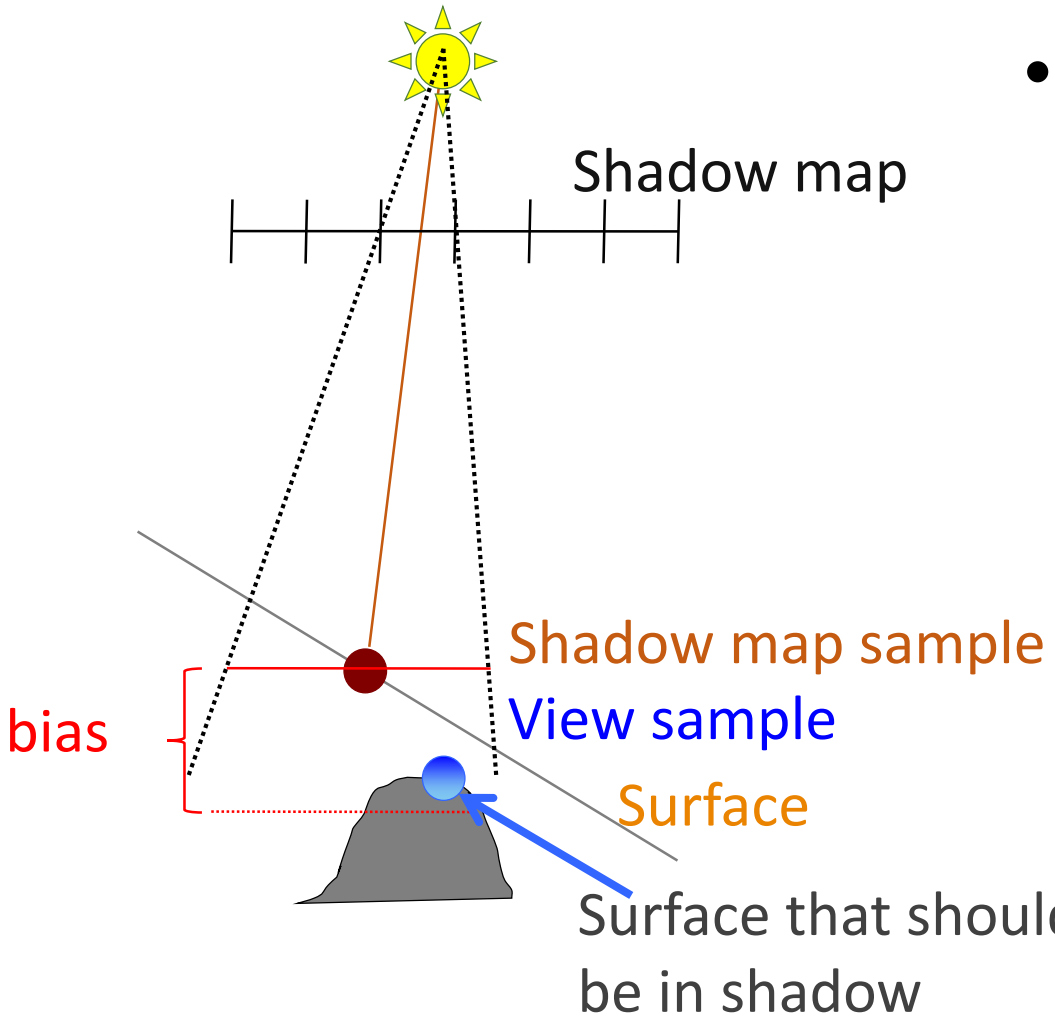
Bias



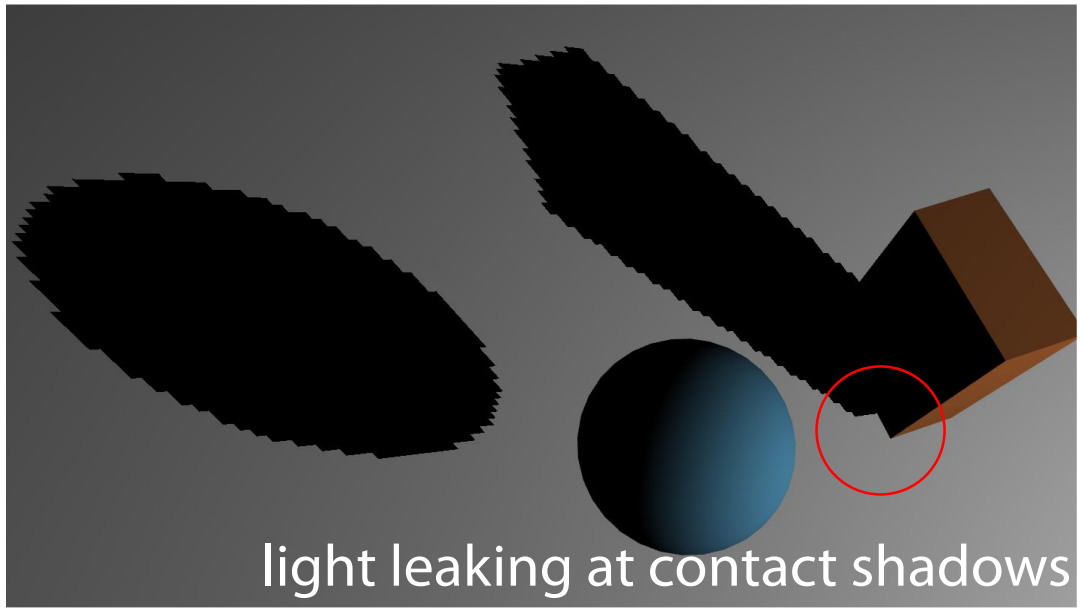
- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing



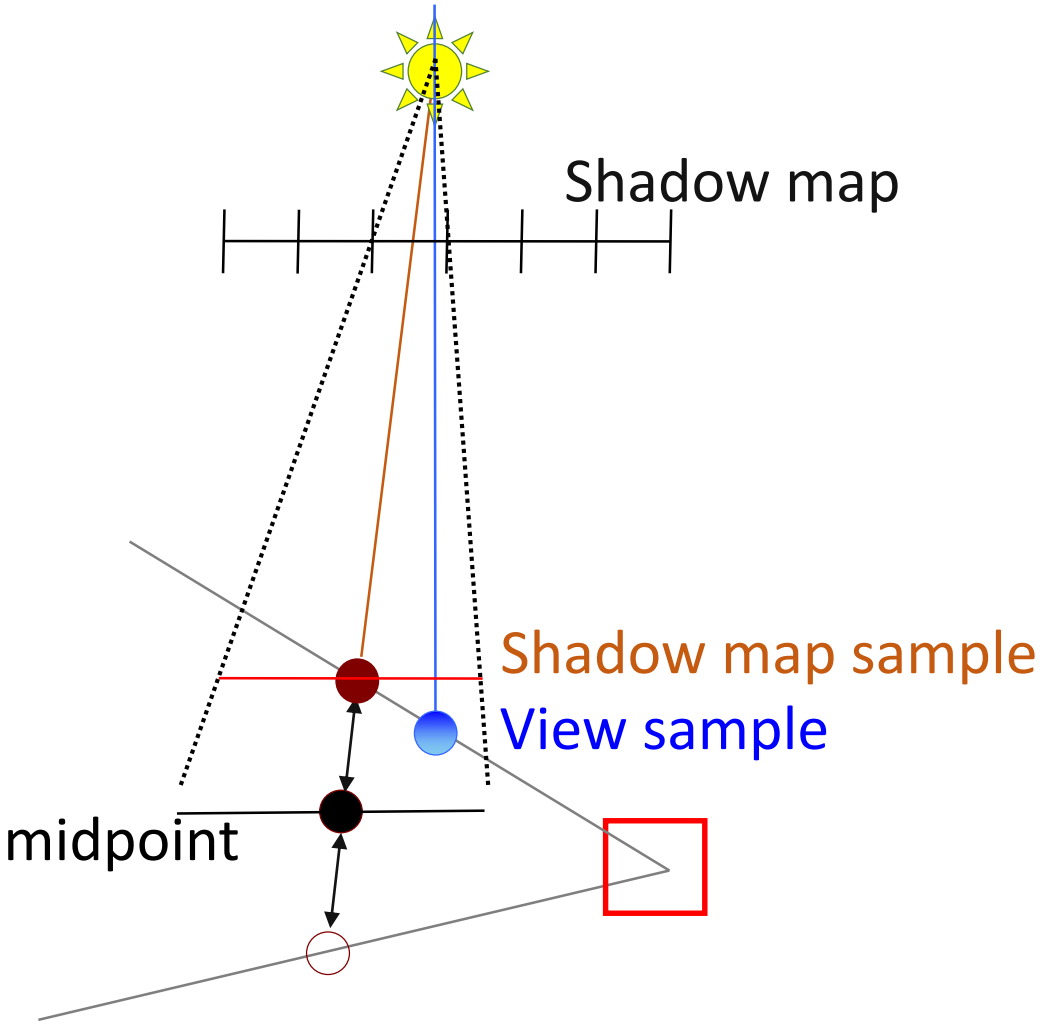
Bias



- Too much bias leads to light leaking (aka peter panning)

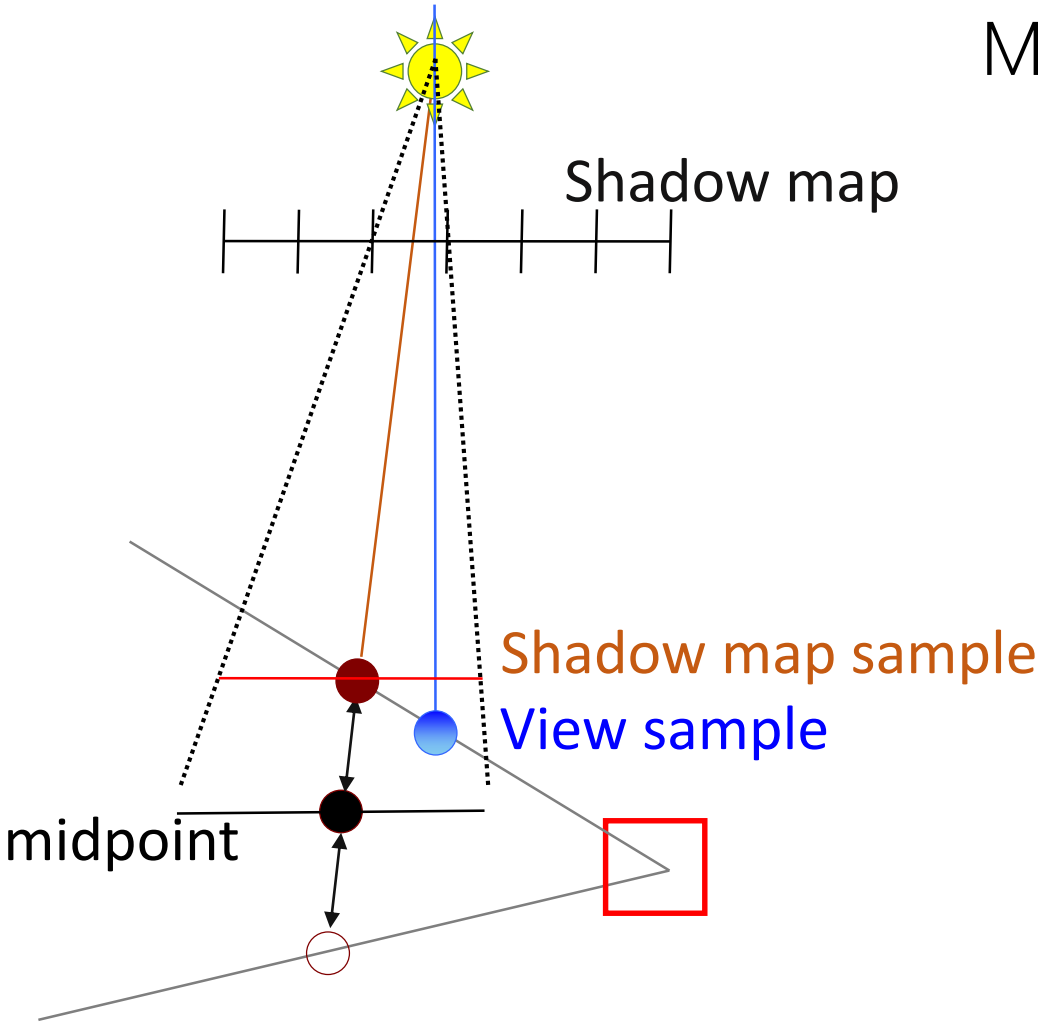


Ameliorating the Bias

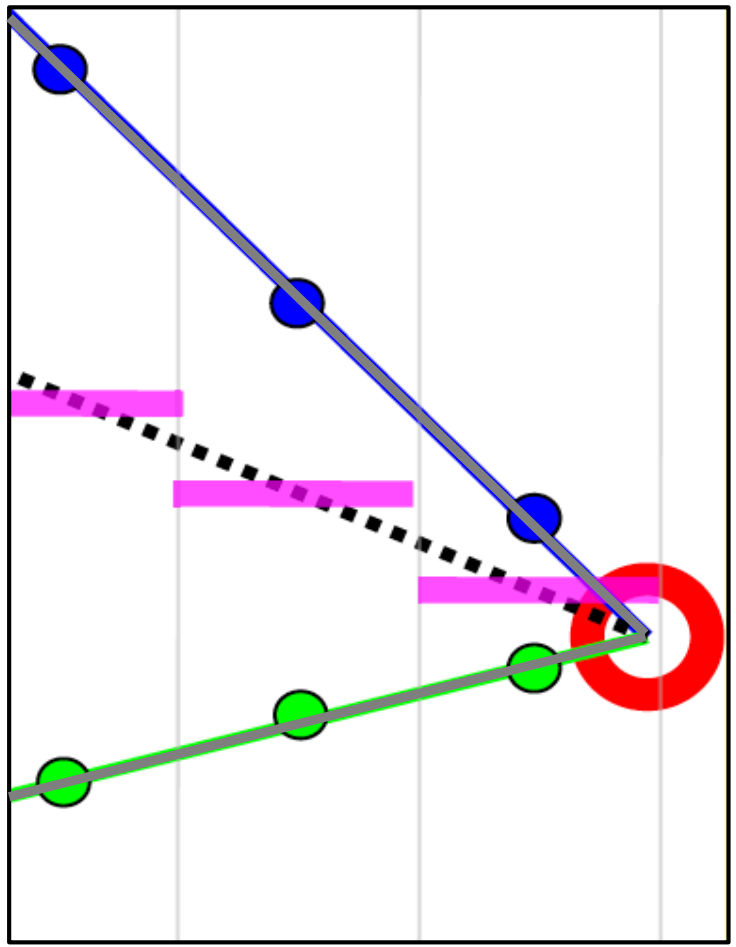


- Midpoint Shadow Maps [Woo 92]
- Second Depth Shadow Mapping [Wang and Molnar94]
- Dual Depth Layer [Weiskopf and Ertl 04]

Ameliorating the Bias

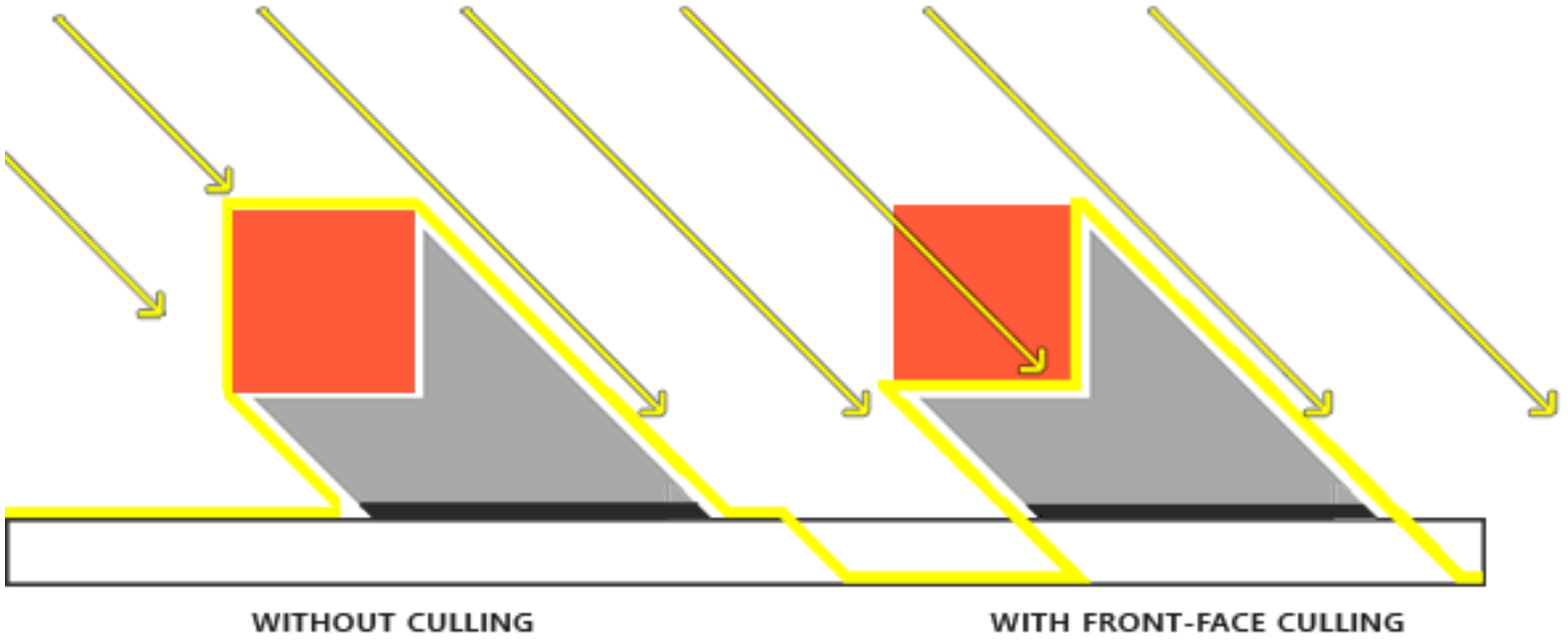


Midpoint Shadow Maps [Woo 92]



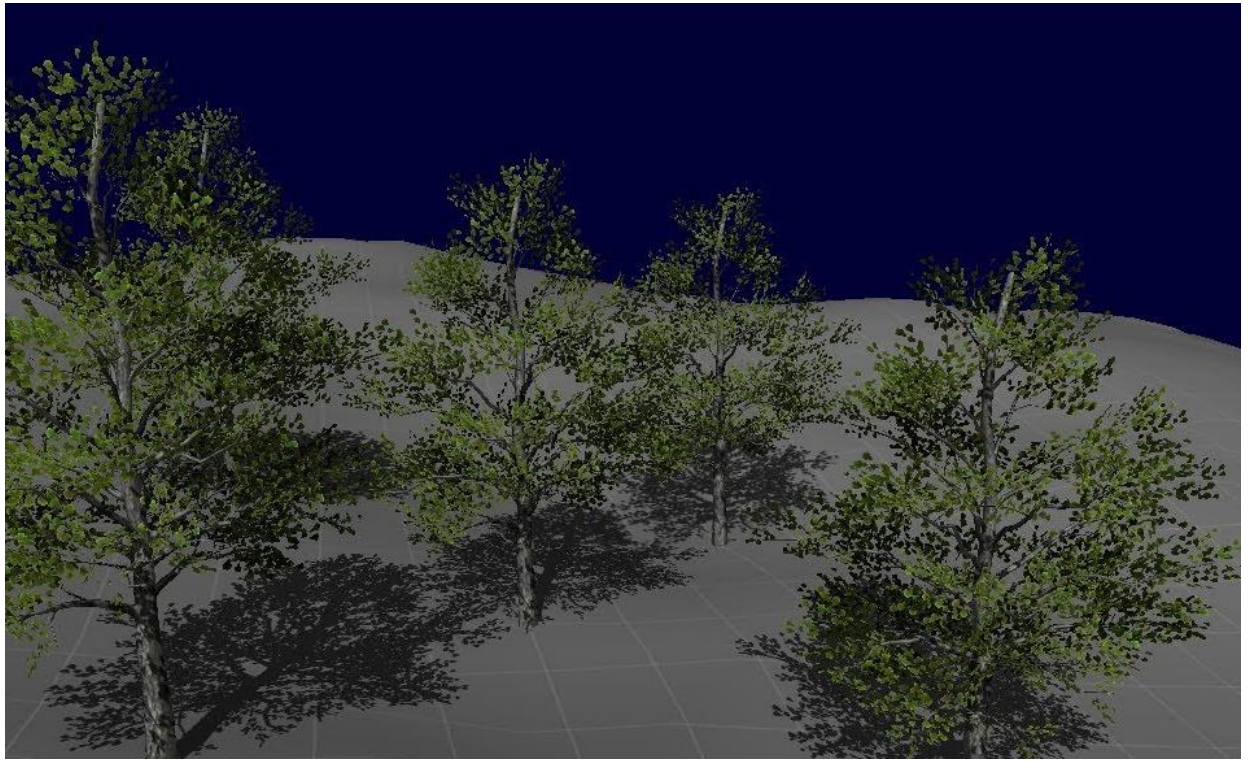
Ameliorating the Bias

- We can also use the back faces for the depth map creation
  - Using front-face culling



Shadow Maps

- Shadow maps are the basis of most modern shadowing techniques
  - E.g. cascaded shadow maps

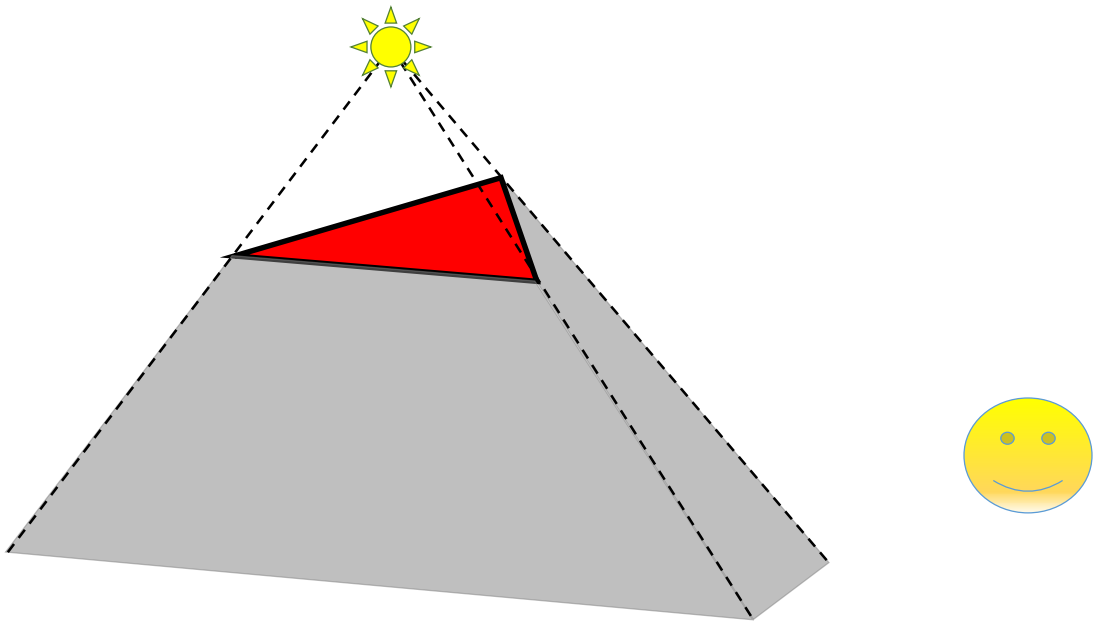


Outline

- *Motivation*
- *Shadow Maps*
- Shadow Volumes

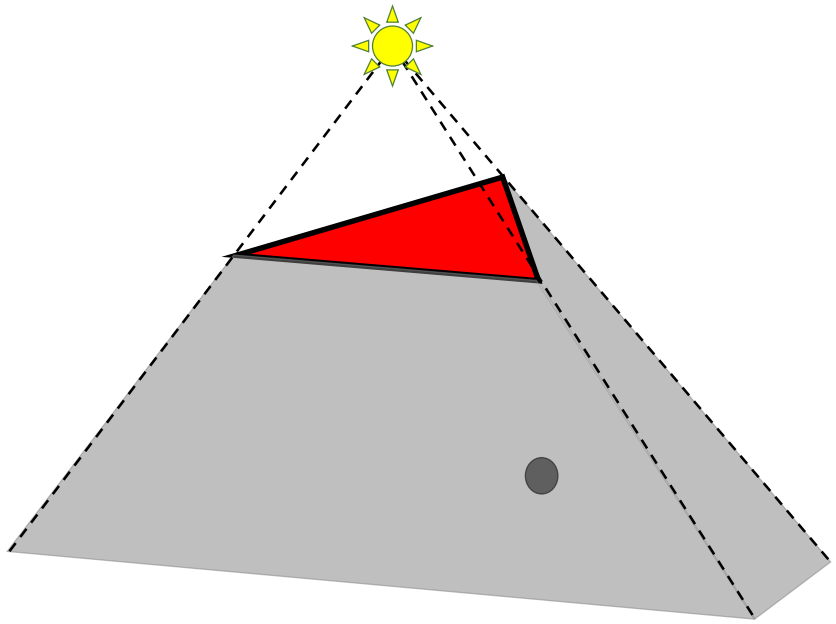
Shadow Volumes. Concept

- Concept
  - Create volumes of “space in shadow" from each triangle
    - Each triangle creates 3 quads that extends to infinity



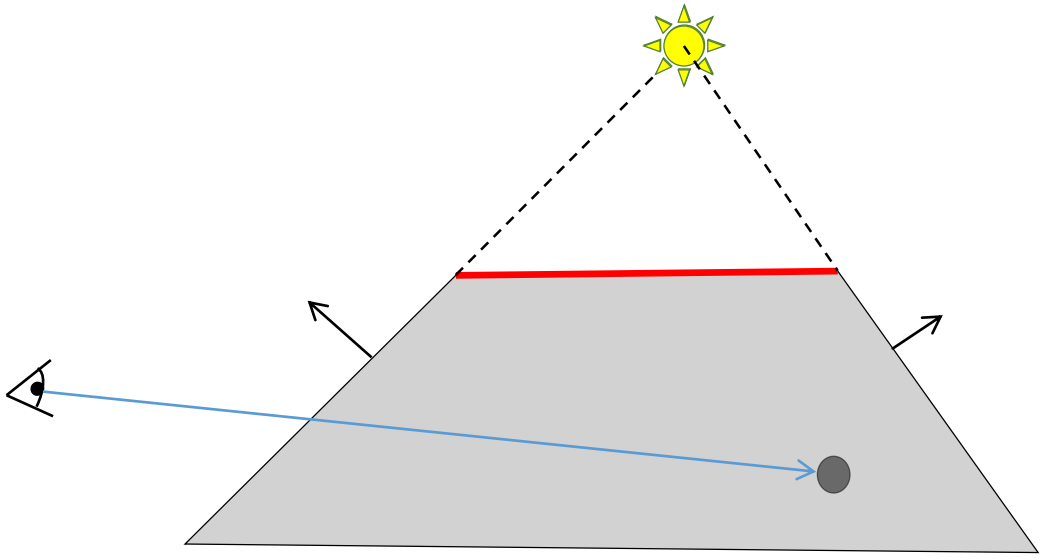
Shadow Volumes. Concept

- To test a point, count how many shadow volumes it is located within. One or more means point is in shadow



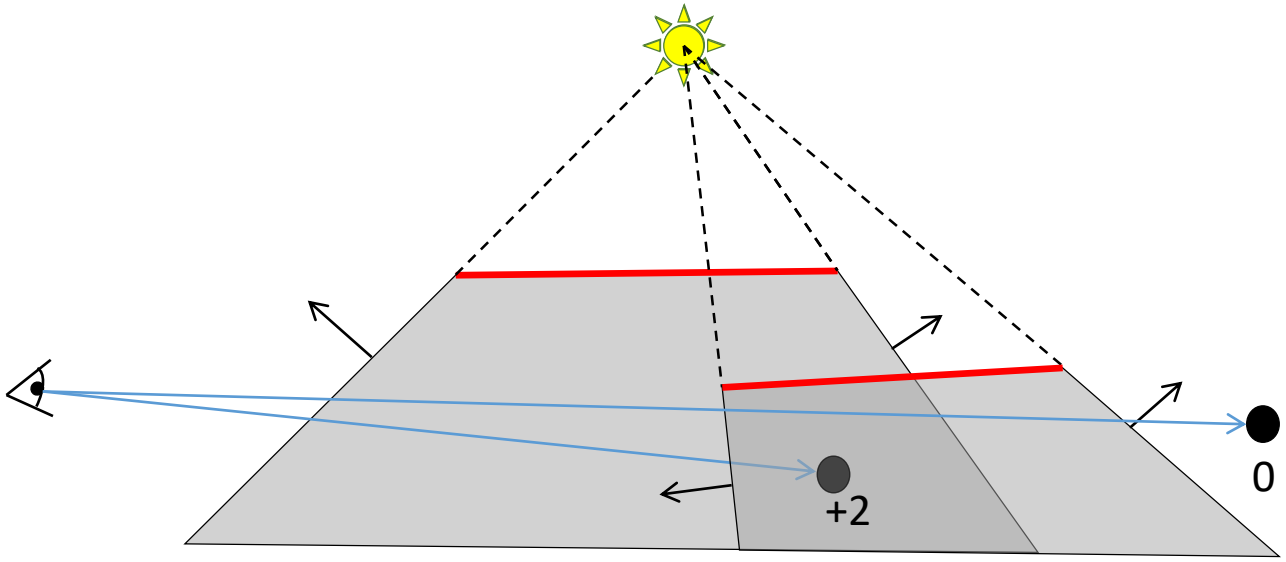
Shadow Volumes. Concept

- To test a point, count how many shadow volumes it is located within. One or more means point is in shadow



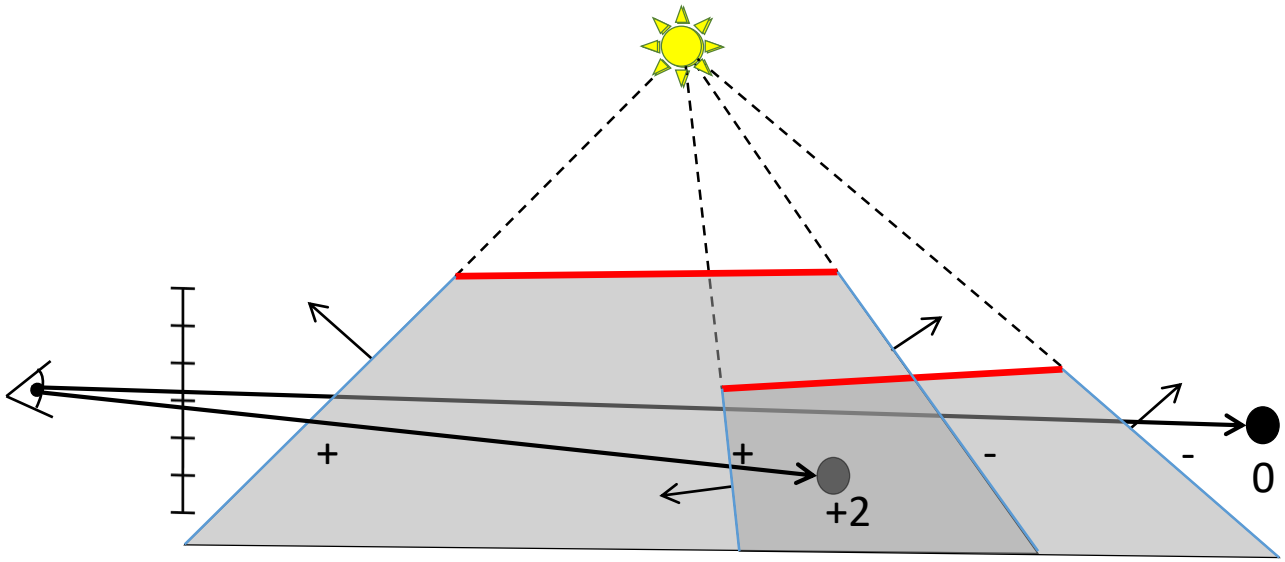
Shadow Volumes. Concept

- To test a point, count how many shadow volumes it is located within. One or more means point is in shadow



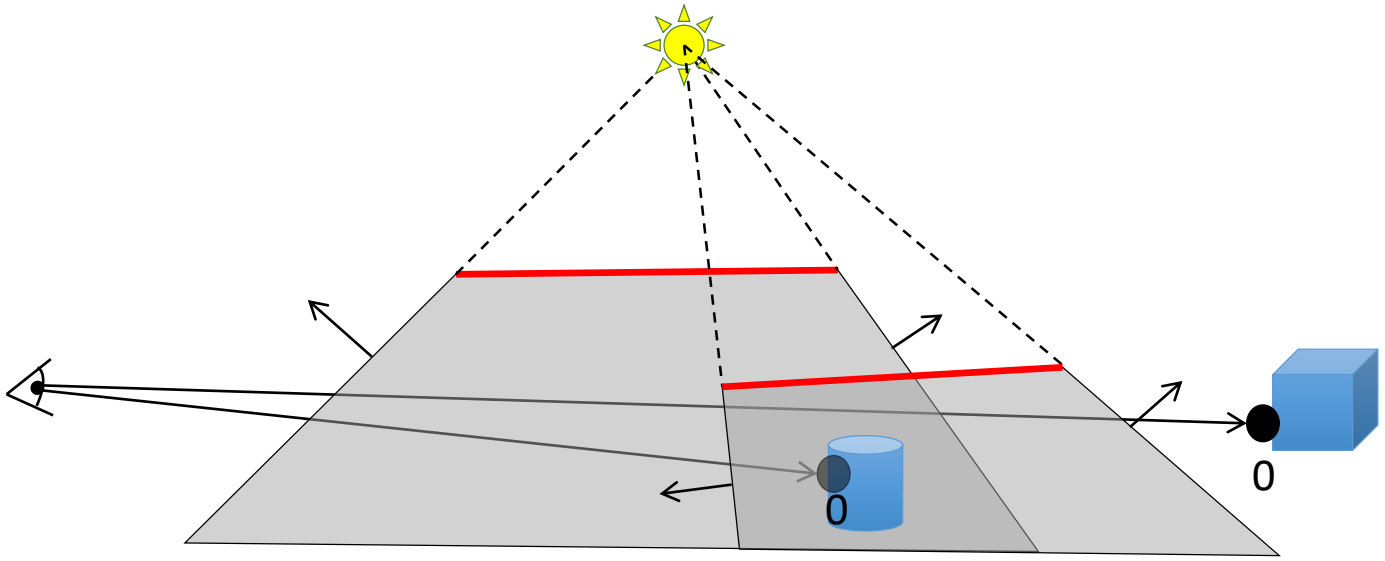
Shadow Volumes. Concept

- A counter per pixel
- If we go through more frontfacing than backfacing polygons, then the point is in shadow



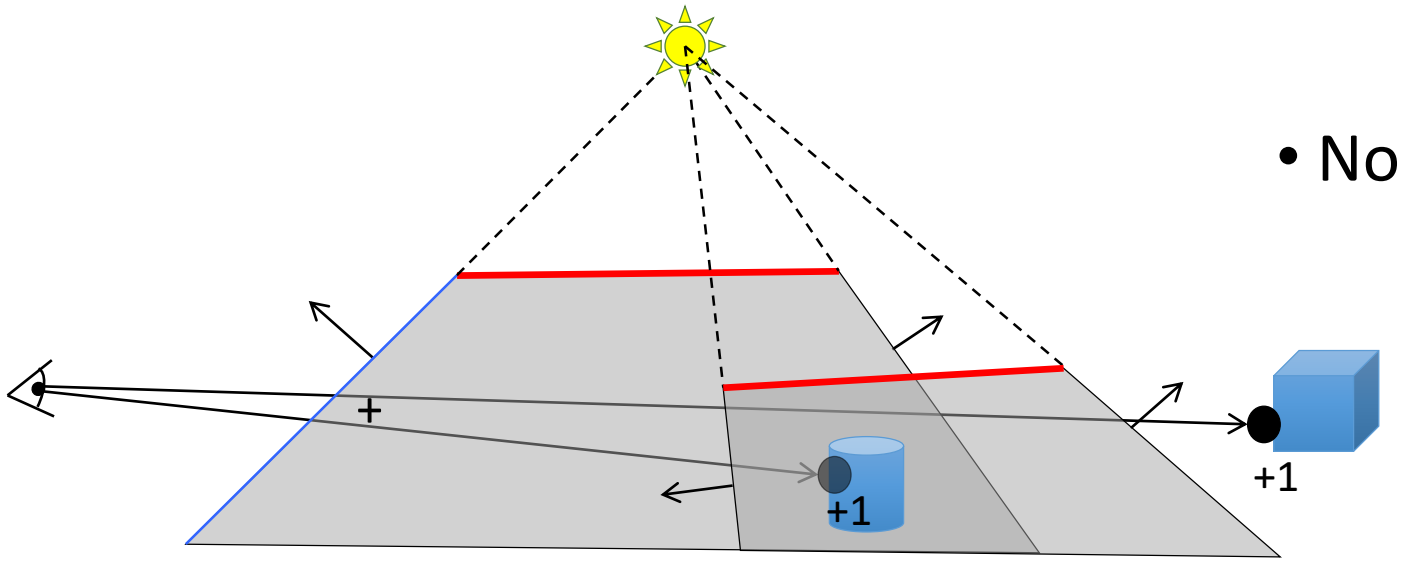
Shadow Volumes. Stencil buffer

- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



Shadow Volumes. Stencil buffer

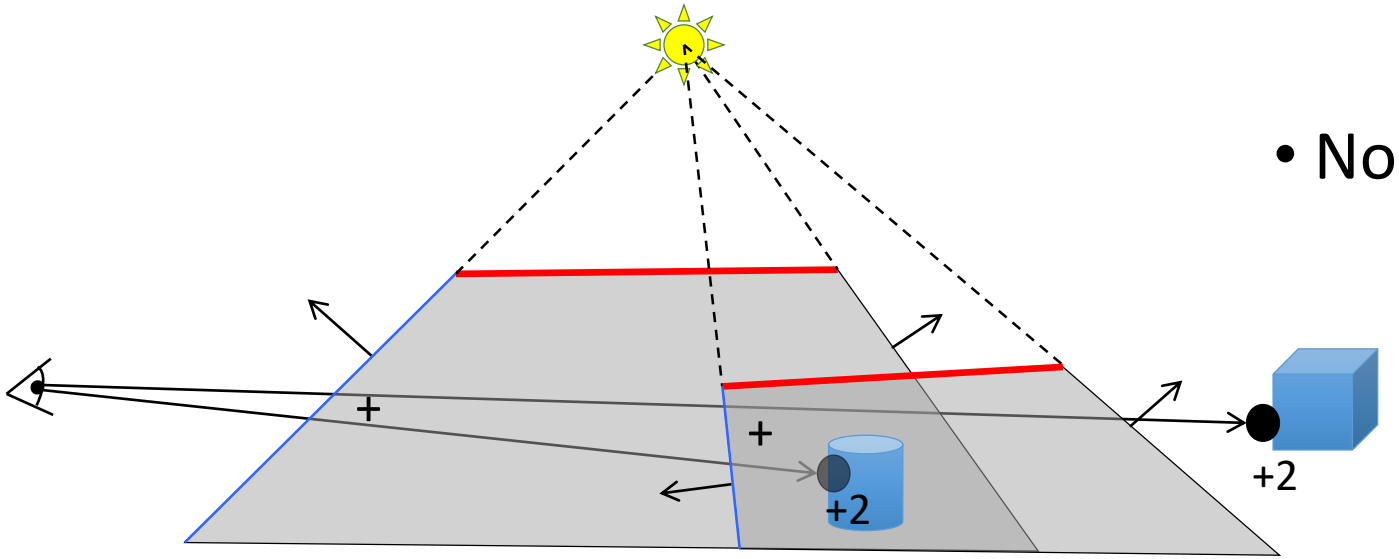
- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



- No updating of z-buffer

Shadow Volumes. Stencil buffer

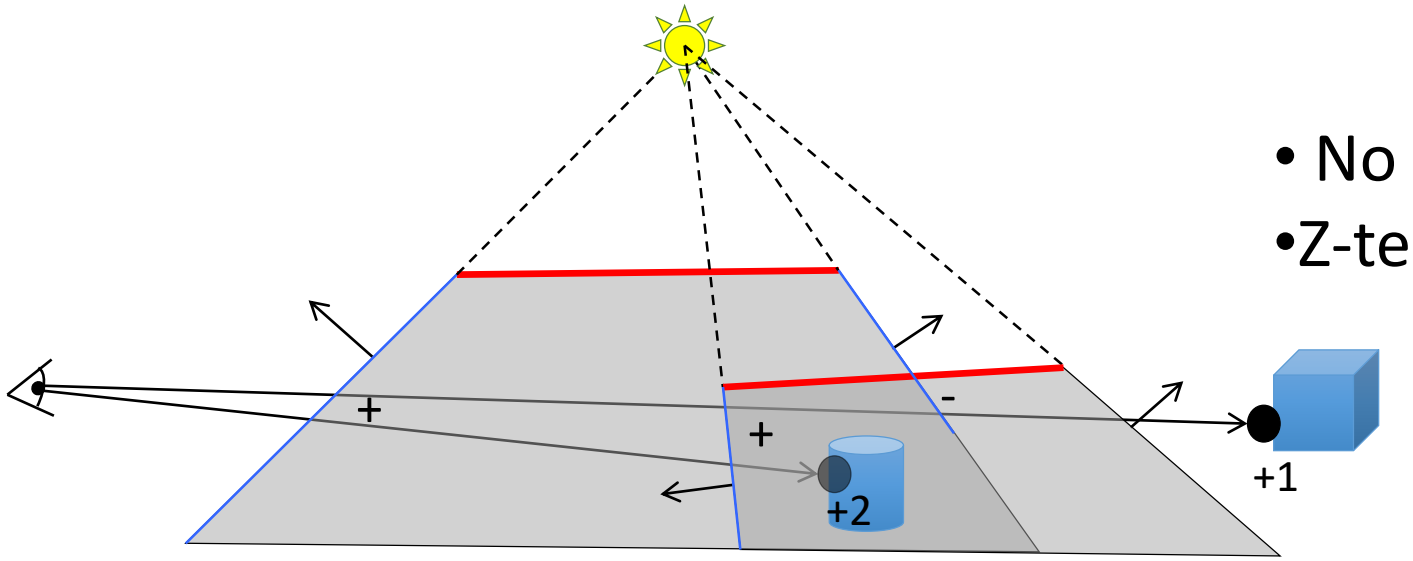
- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



- No updating of z-buffer

Shadow Volumes. Stencil buffer

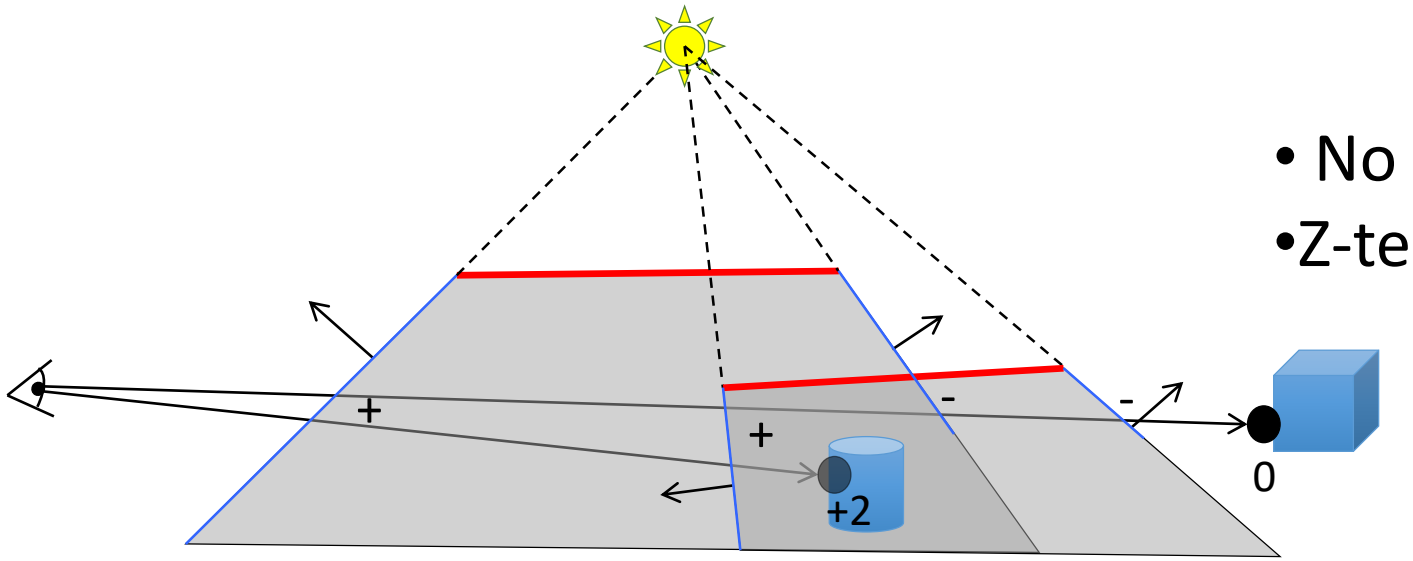
- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



- No updating of z-buffer
- Z-test is enabled as usual

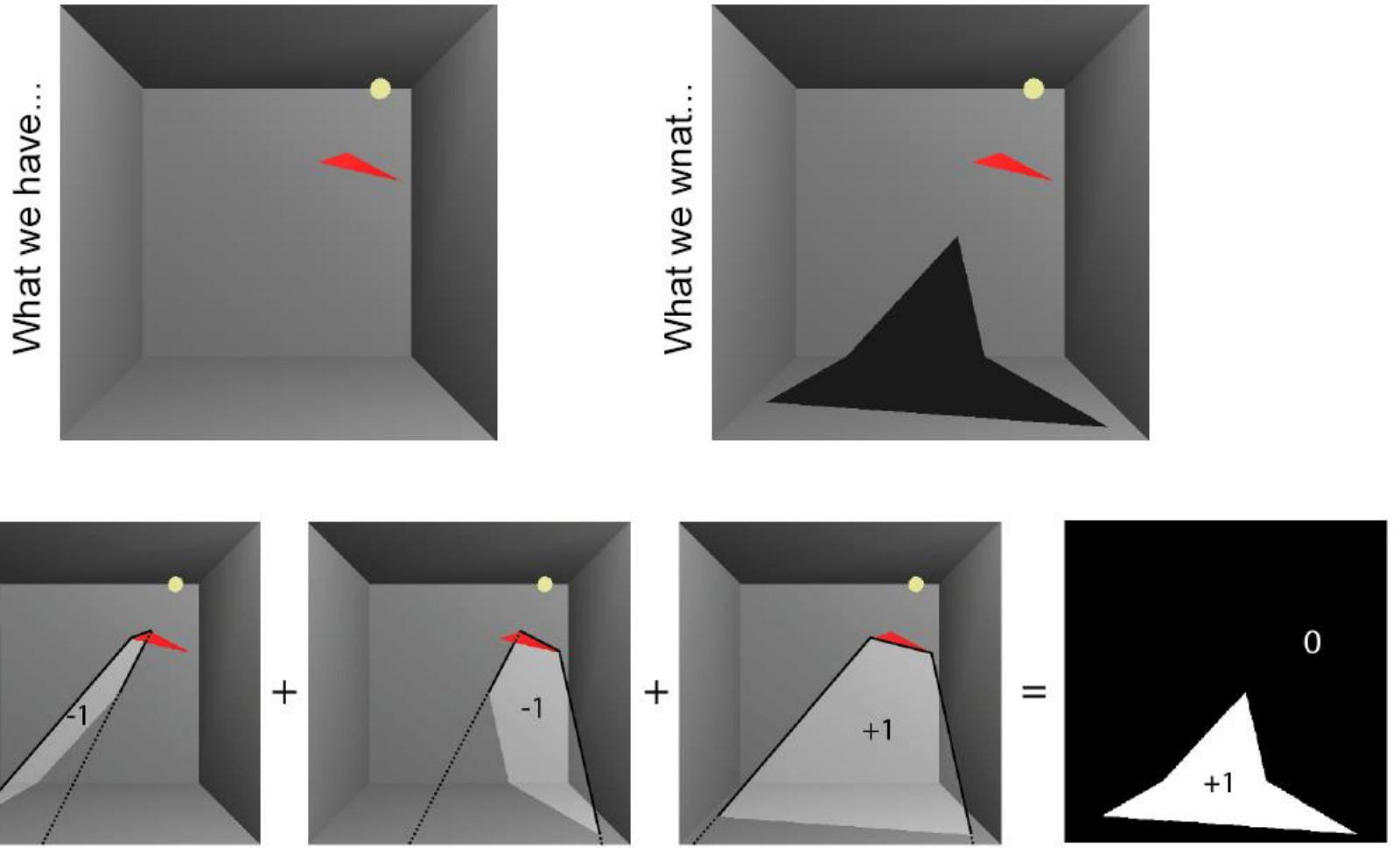
Shadow Volumes. Stencil buffer

- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



- No updating of z-buffer
- Z-test is enabled as usual

Z-pass: How the stencil buffer is used

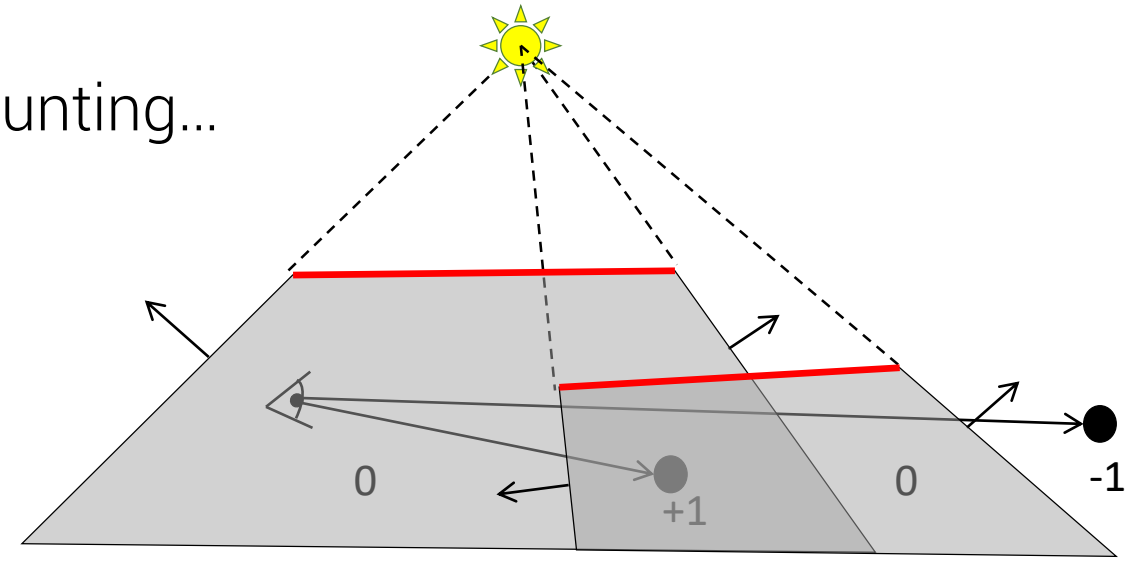


Shadow Volumes. Stencil buffer

- A three pass process [Heidmann91]
  - 1st pass: Render ambient lighting
  - 2nd pass:
    - Draw to stencil buffer only
      - Turn off updating of z-buffer and writing to color buffer but still use standard depth test
      - Set stencil operation to
        - incrementing stencil buffer count for frontfacing shadow volume quads, and
        - decrementing stencil buffer count for backfacing shadow volume quads
    - use glStencilOpSeparate(…)
  - 3rd pass: Render diffuse and specular where stencil buffer is 0.

Shadow Volumes. Eye Location Problem

- If the eye is located inside one or more shadow volumes, then the count will be wrong
- Solution:
  - Offset stencil buffer with the #shadow volumes that the eye is located within
  - Or modify the way we do the counting…

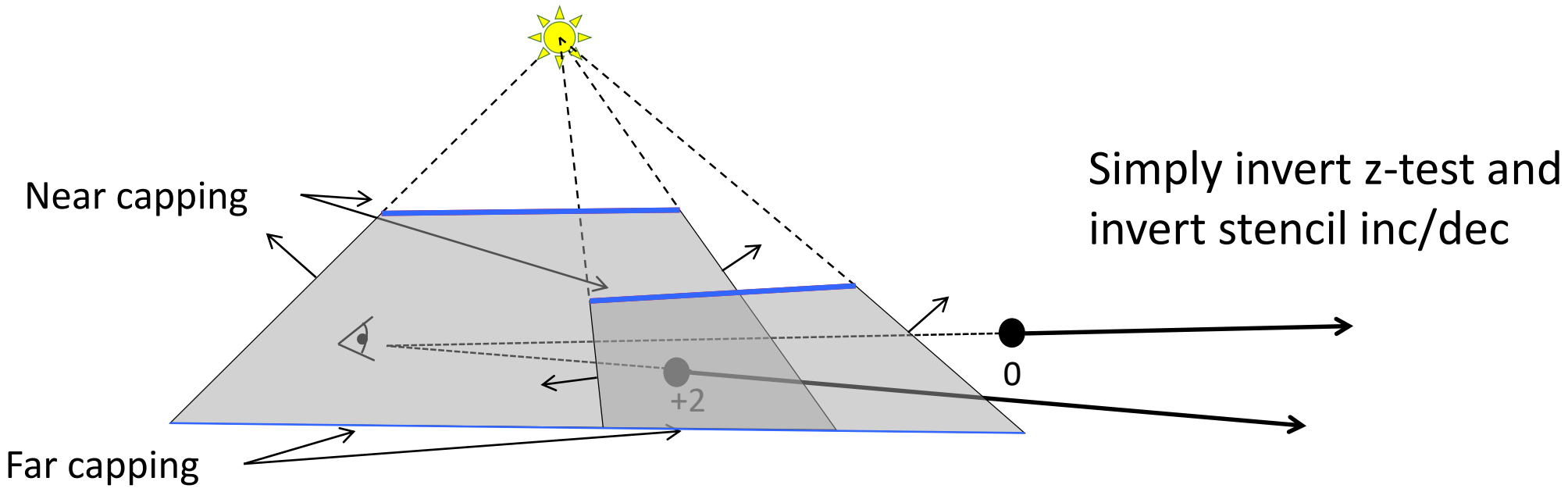


Shadow Volumes. The Z-fail Algorithm

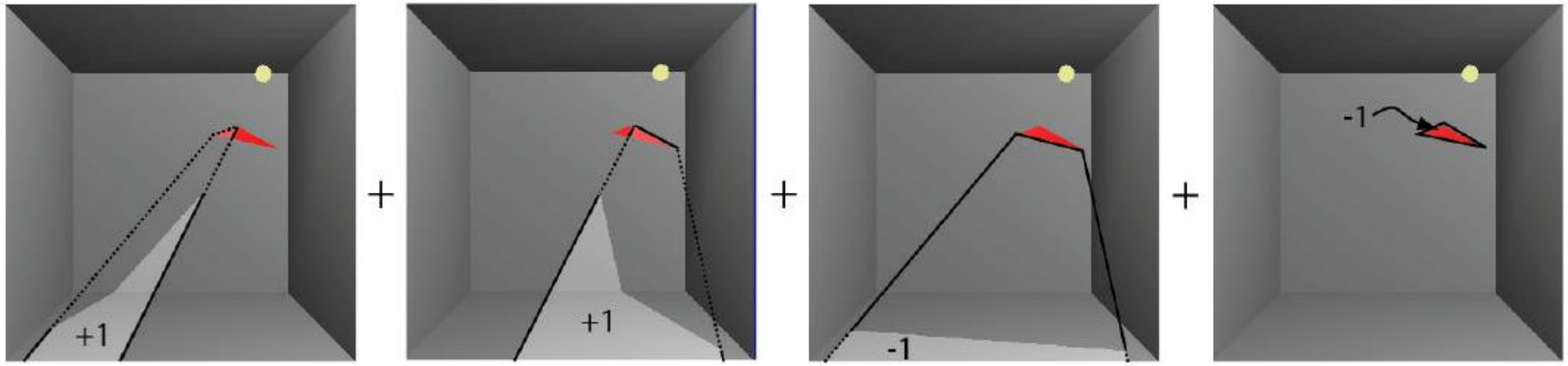
- By [Carmack00] and [Bilodeau and Songy 99]
  - “Carmacks Reverse”
- Count to infinity instead of to the eye
  - **Add caps to the volume frustums**
  - We can choose any reference location for the counting
  - A point in light avoids any offset
  - **Infinity is always in light – if we cap the shadow volumes at infinity**

Shadow Volumes. The Z-fail Algorithm

- By [Carmack00] and [Bilodeau and Songy 99]
  - Count to infinity instead of to the eye



Z-fail



Compared to Z-pass:

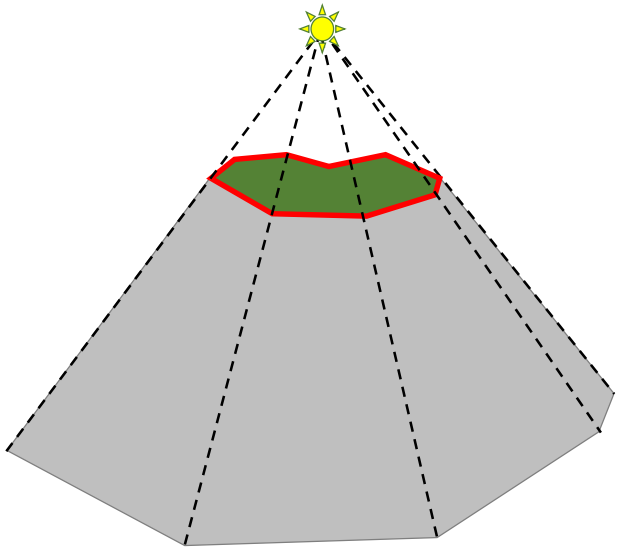
Invert z-test

Invert stencil inc/dec

I.e., count to infinity instead of from eye.

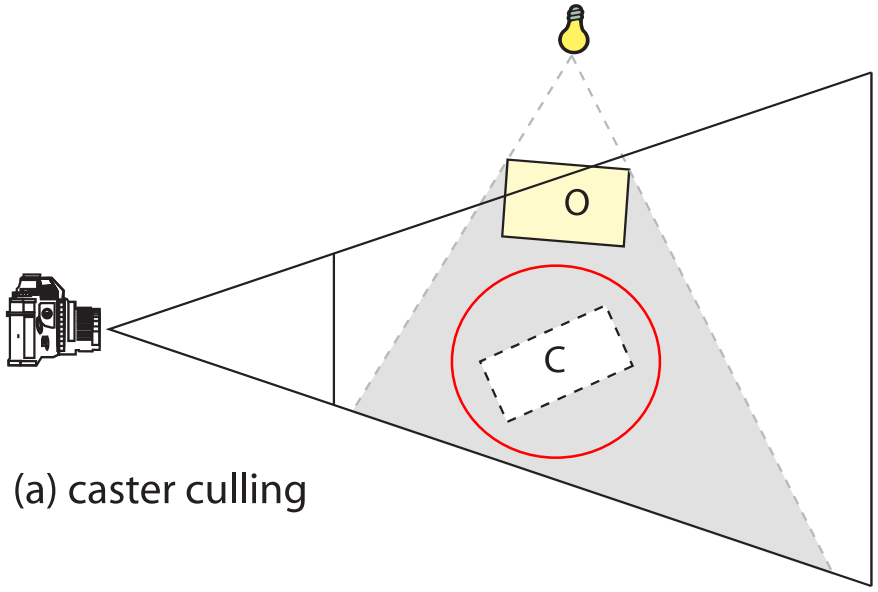
Shadow Volumes - Summary

- Pros:
  - High quality
- Cons:
  - OVERDRAW



Shadow Volumes. Culling and Clamping

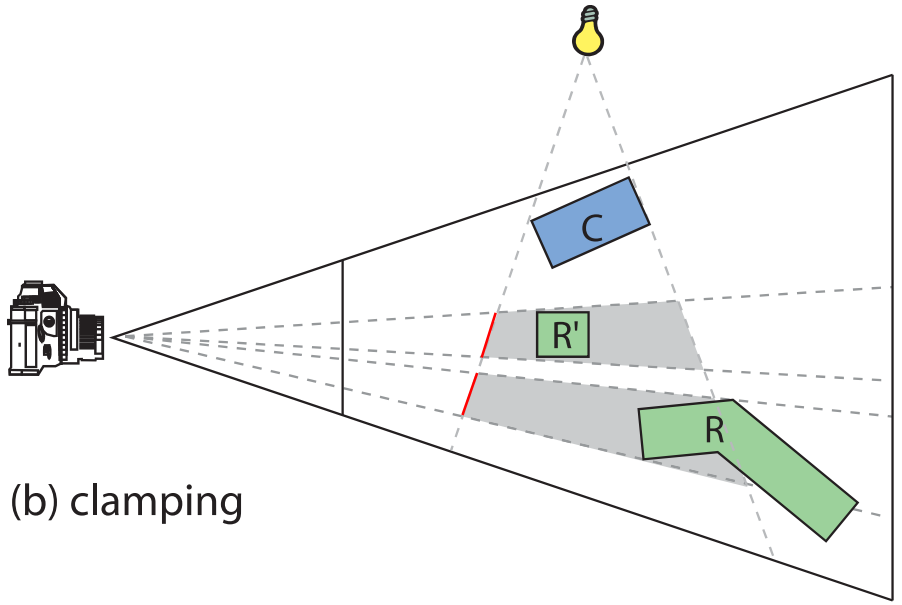
- Culling of Shadow Volumes [Lloyd et al. 2004][Stich et al. 2007]
  - Culling of Shadow Casters if it is located totally within shadow
    - Tested against a shadow depth map



(a) caster culling

Shadow Volumes. Culling and Clamping

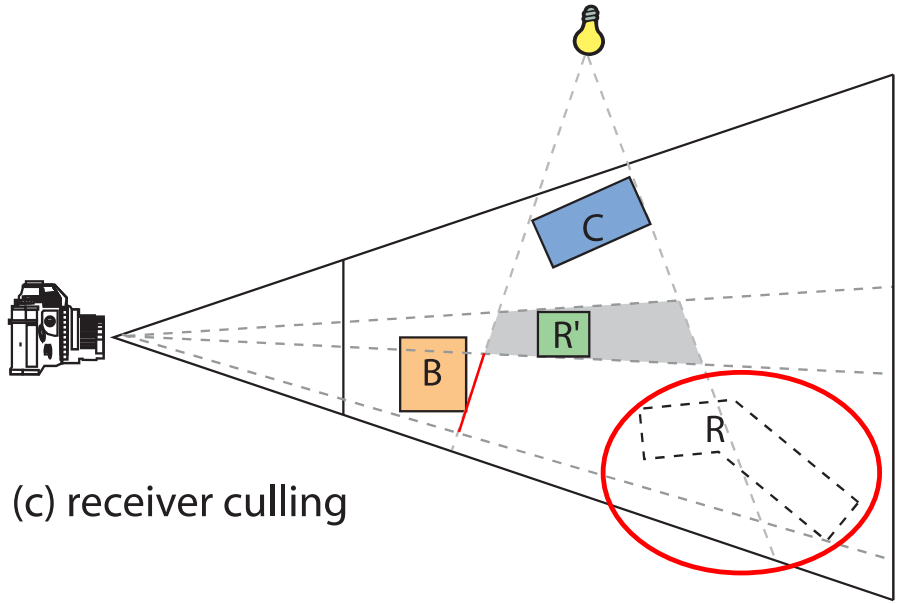
- Clamping of Shadow Volumes
  [Lloyd et al. 2004] [Eisemann and Decoret 2006]
  - Idea: Only render parts of shadow quads that affects a shadow receiver
    - Tested against AABB around shadow receivers



(b) clamping

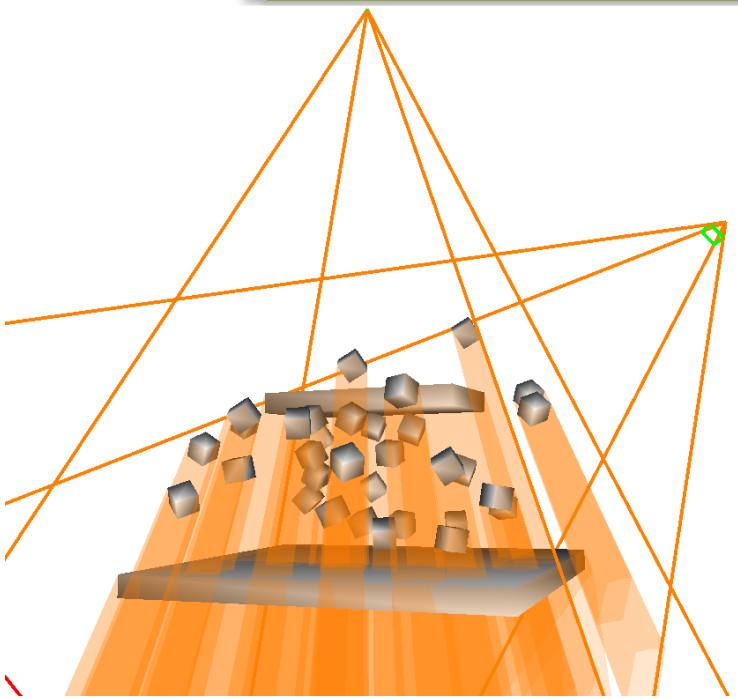
Shadow Volumes. Culling and Clamping

- Culling of Shadow Volumes
  [Lloyd et al. 2004] [Eisemann and Decoret 2006]
  - Receiver Culling
    - Idea: Cull part of shadow volumes where shadow receivers are not visible from the eye

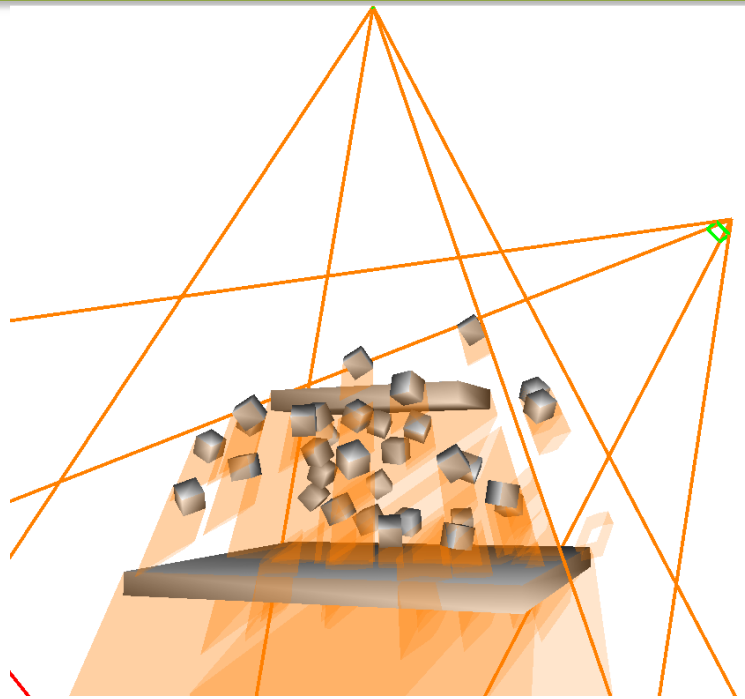


(c) receiver culling

Shadow Volumes. Culling and Clamping



Shadow volumes

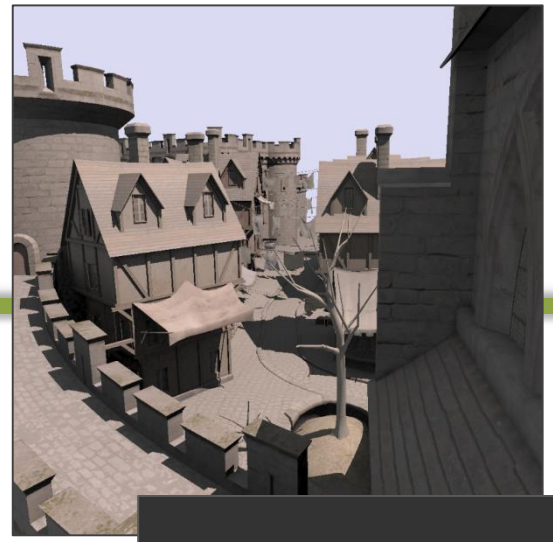


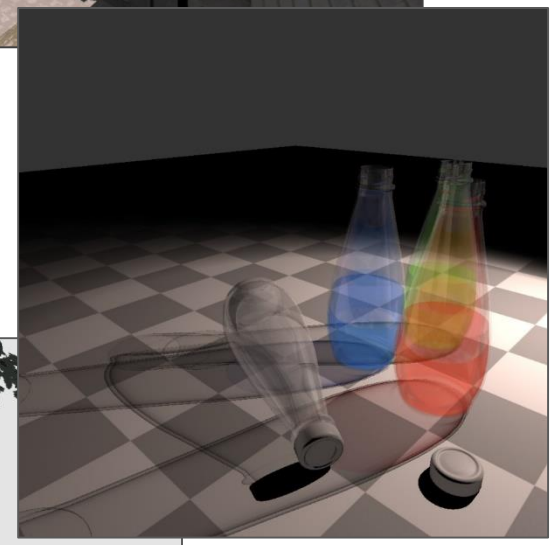
CC Shadow volumes

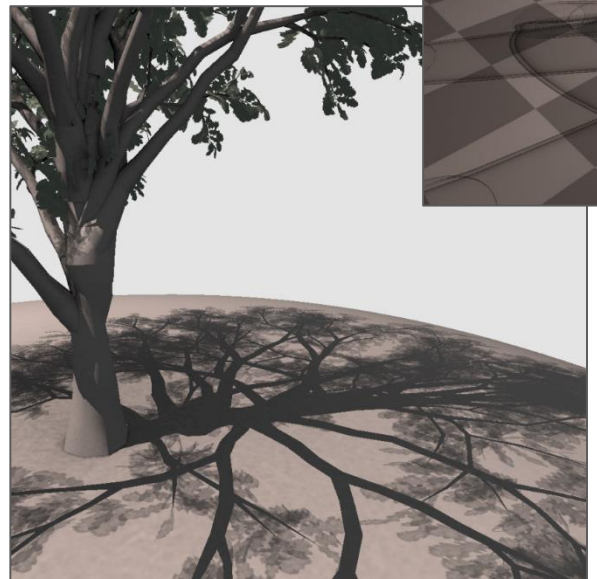
Illustrates reduced depth complexity when using Culling and Clamping

Per-Triangle Shadow Volumes

- Per-triangle Shadow Volumes [Sintorn et al., 2011]
  - Idea:
    - Return to using one shadow volume per triangle
      - 100% robust + easiest.
    - Achieve efficiency by changing rasterization primitive from shadow quad to triangle shadow volume against hierarchical shadow buffer
    - Using CUDA rasterization
  - Allows for a much more general and robust algorithm





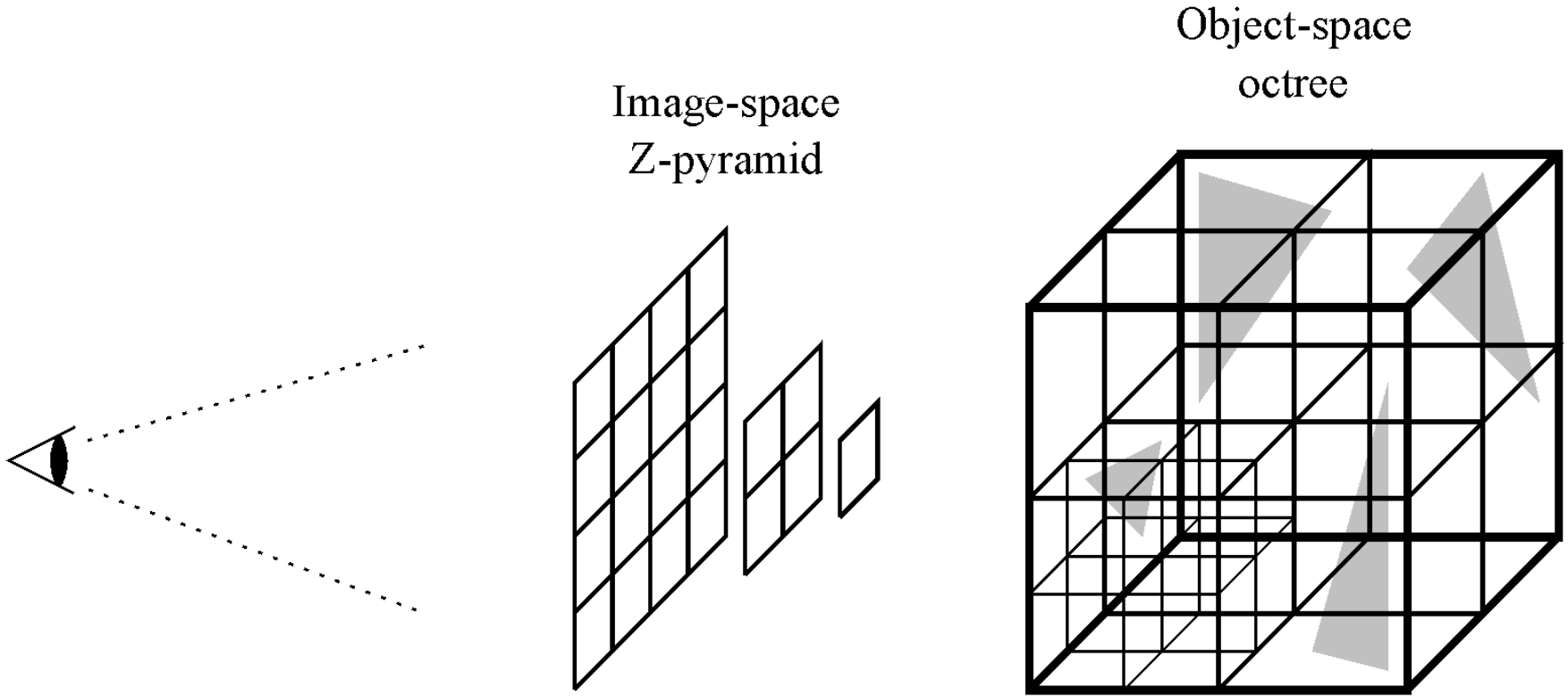


Per-Triangle Shadow Volumes

- Render the scene off-screen with ambient color, direct lighting
  - Store samples (color & depth)
    - Will evaluate visibility of each sample
      - Using a shadow buffer (one bit per sample)
    - Need to classify visibility wrt the faces of the frustum generated by each triangle
  - Use a two-level hierarchical depth buffer and a two-level hierarchical shadow buffer

Per-Triangle Shadow Volumes

- Hierarchical depth buffer stores minimum depth per tile
  - Accelerates depth tests by using higher levels
    - Can be cached (less memory required)



Per-Triangle Shadow Volumes

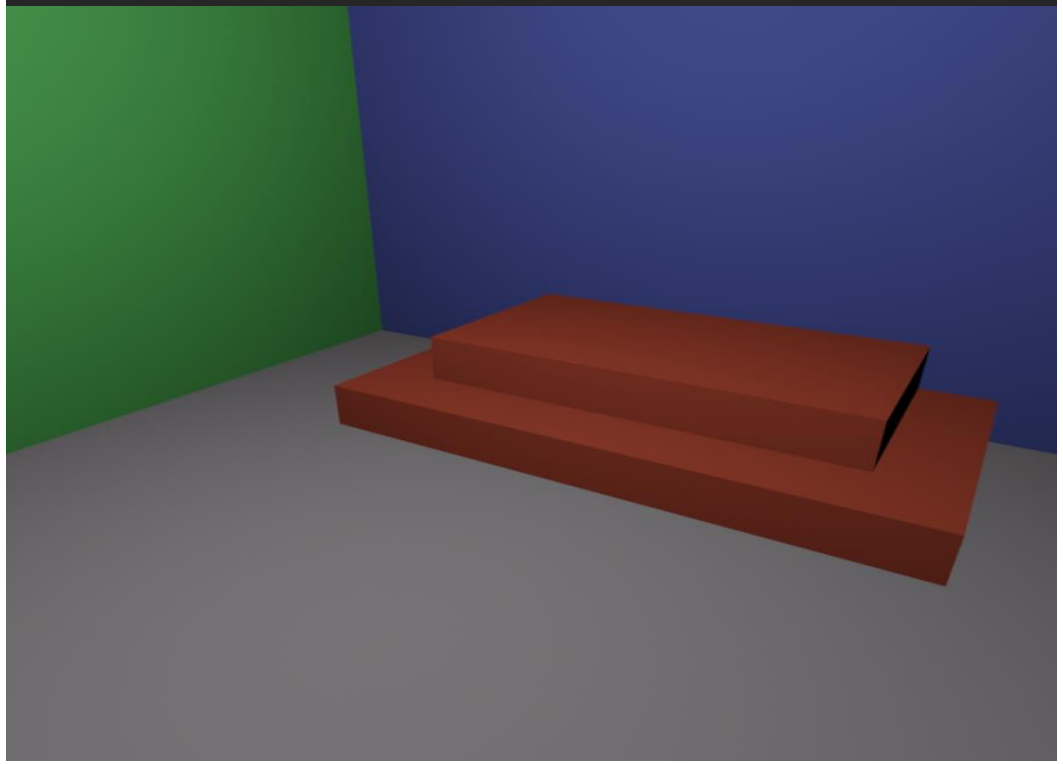
- Upper level of hierarchical depth buffer contains min and max values
  - Define a bounding box (bounding frustum) named tile
    - Need to determine whether samples lie *inside* the box

Per-Triangle Shadow Volumes

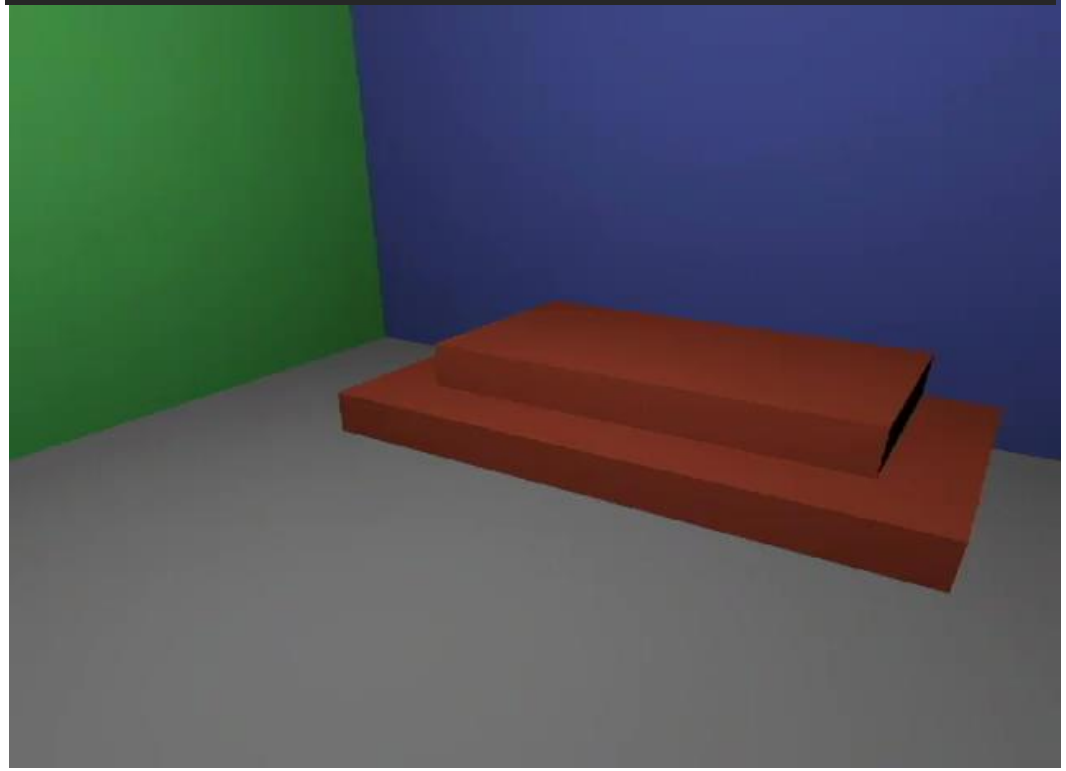
- Classification algorithm
  - Test if tile lies inside shadow volume frustum
    - True: all samples in shadow
      - Mark the shadow buffer bit
      - Abandon the tile
    - False: no samples in shadow
      - Discard the tile
    - Cannot be classified:
      - Go for the lower level
- Implemented in CUDA

Per-Triangle Shadow Volumes

View from camera

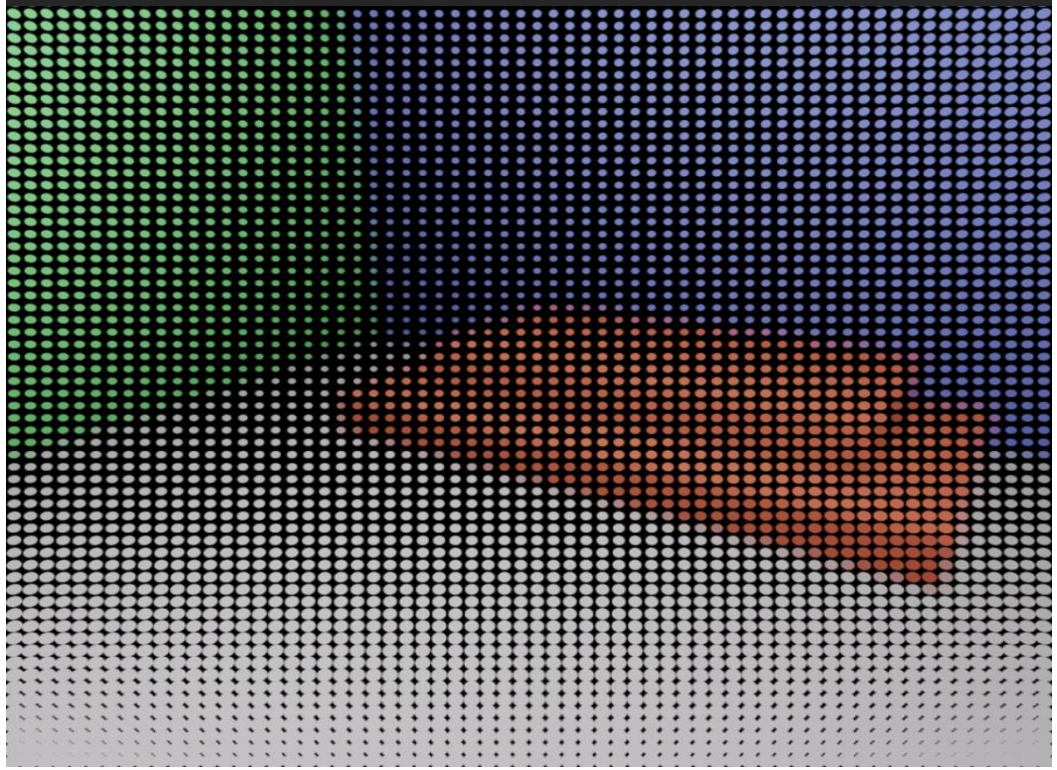


Alternative view

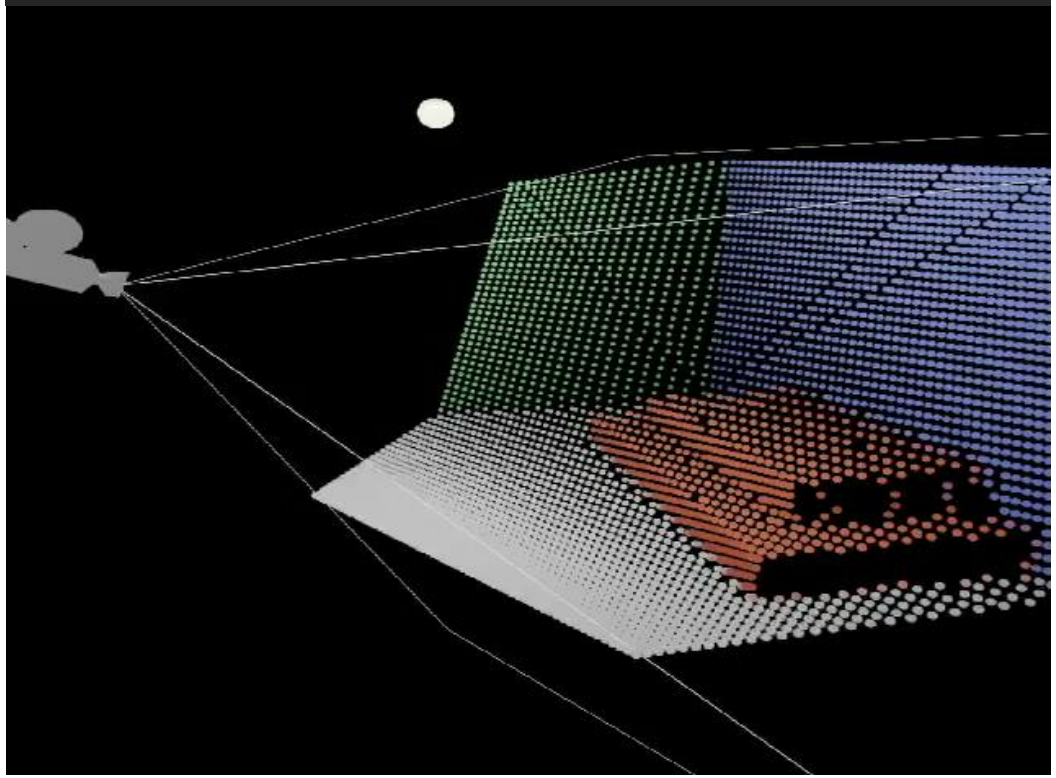


Per-Triangle Shadow Volumes. Algorithm. Building the depth hierarchy

View from camera

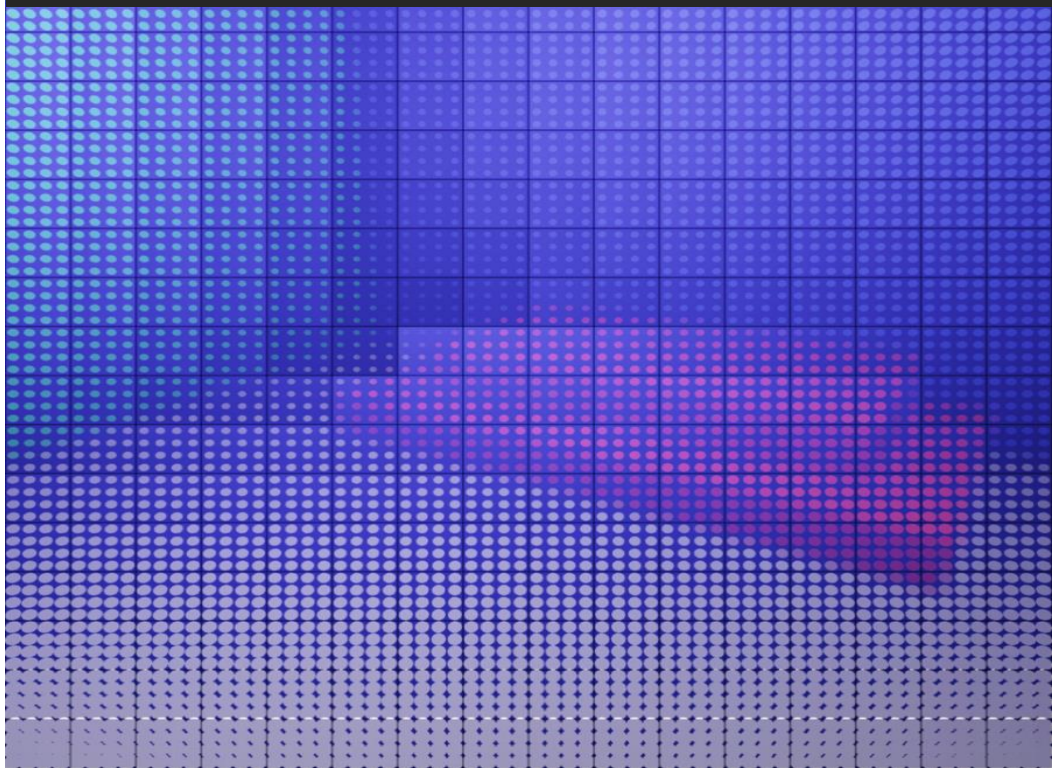


Alternative view

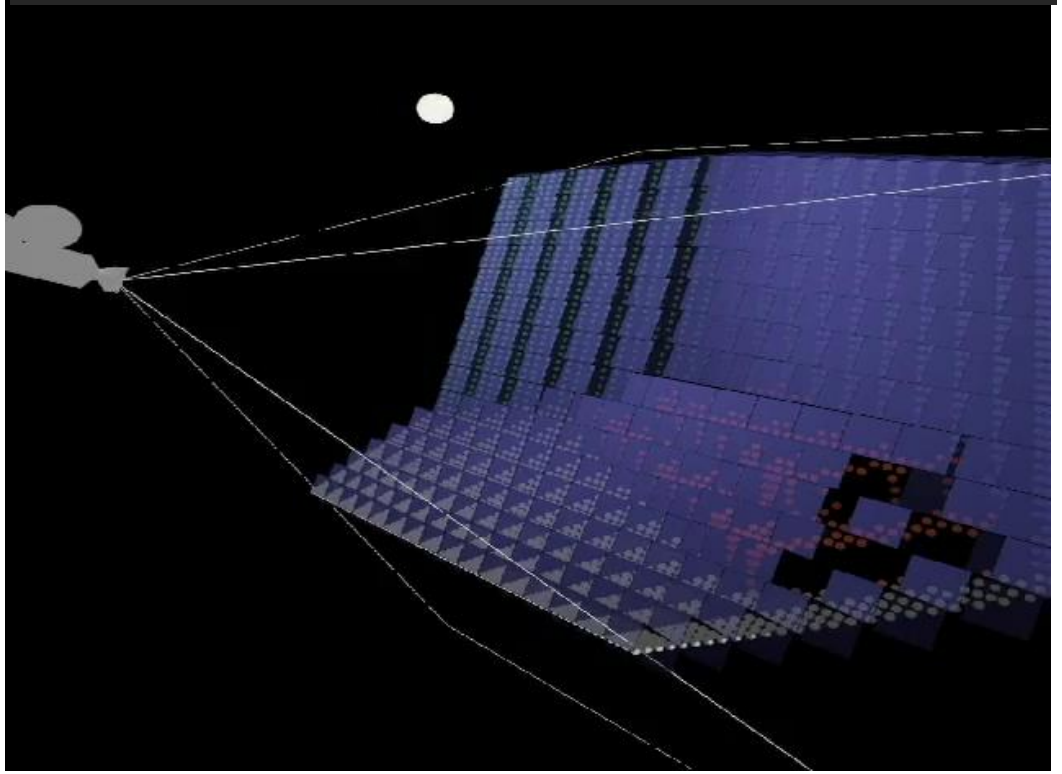


Per-Triangle Shadow Volumes. Algorithm. Building the depth hierarchy

View from camera

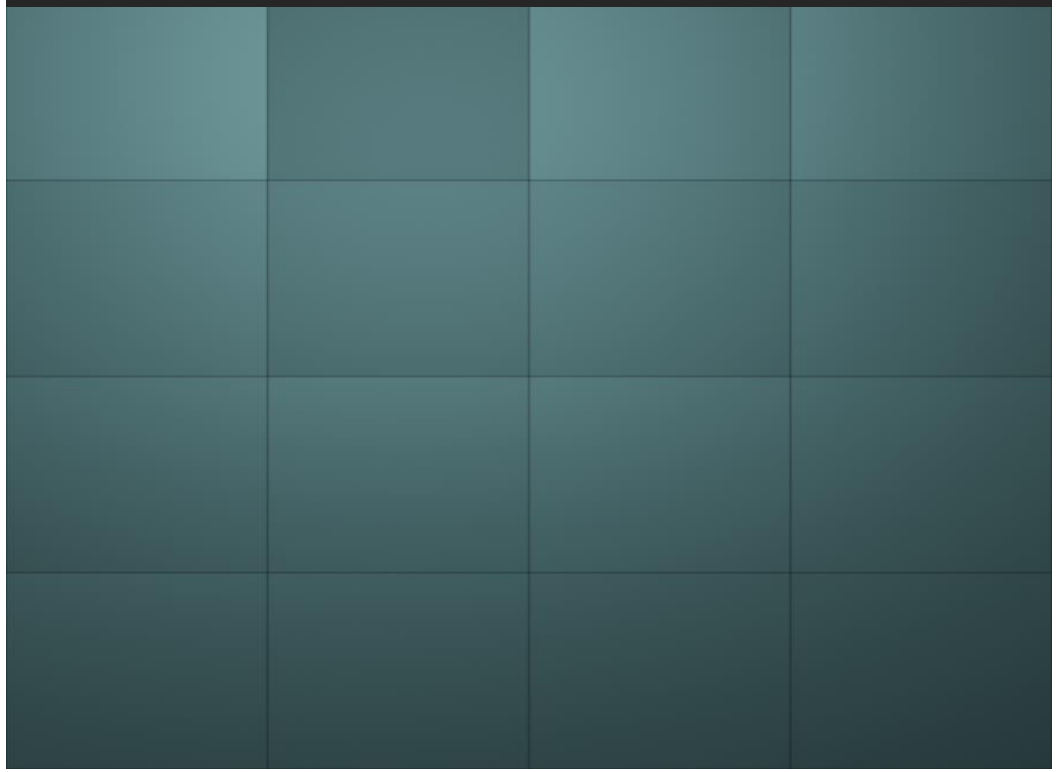


Alternative view

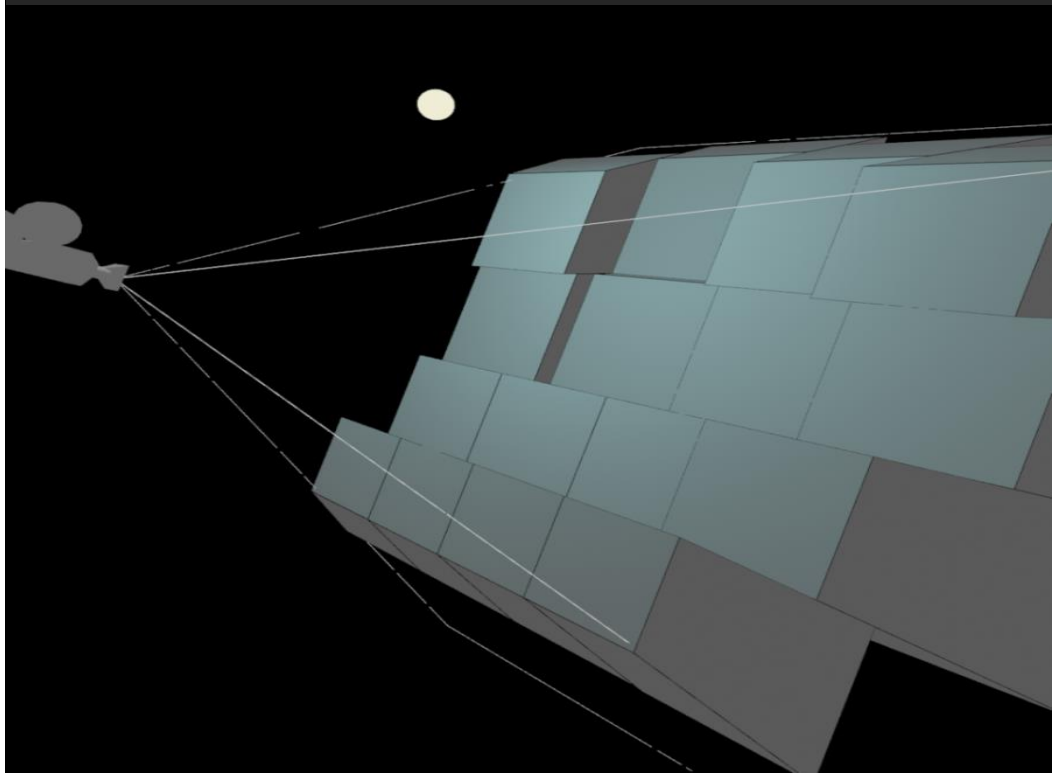


Per-Triangle Shadow Volumes. Algorithm. Building the depth hierarchy

View from camera

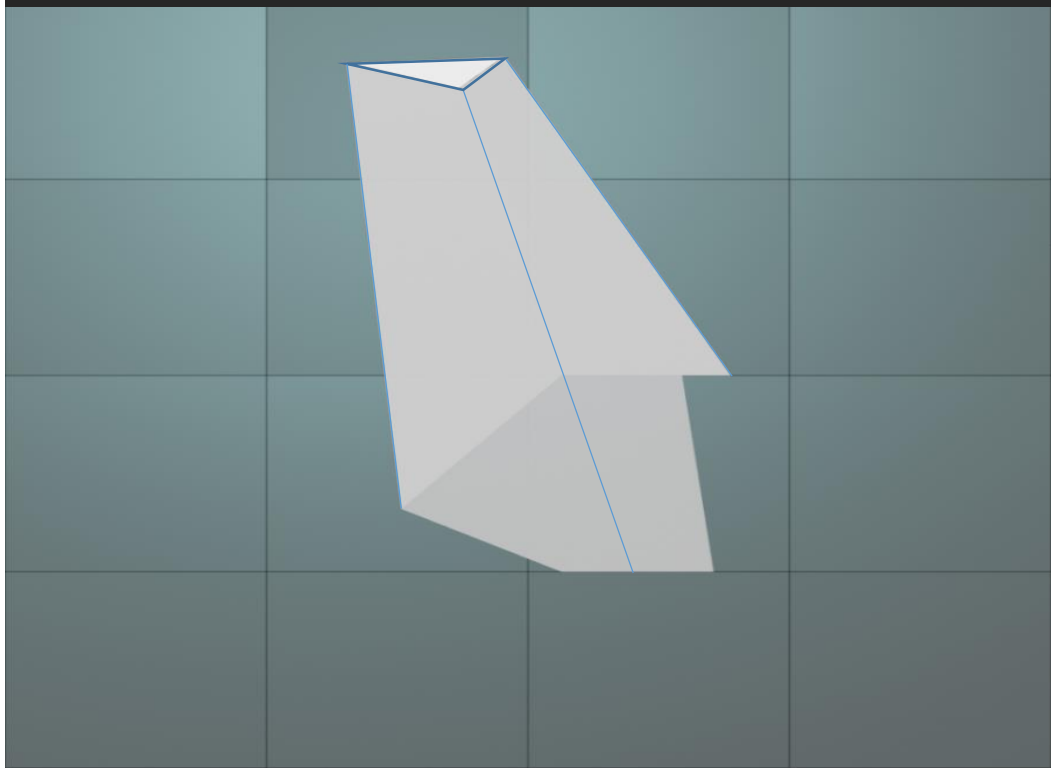


Alternative view

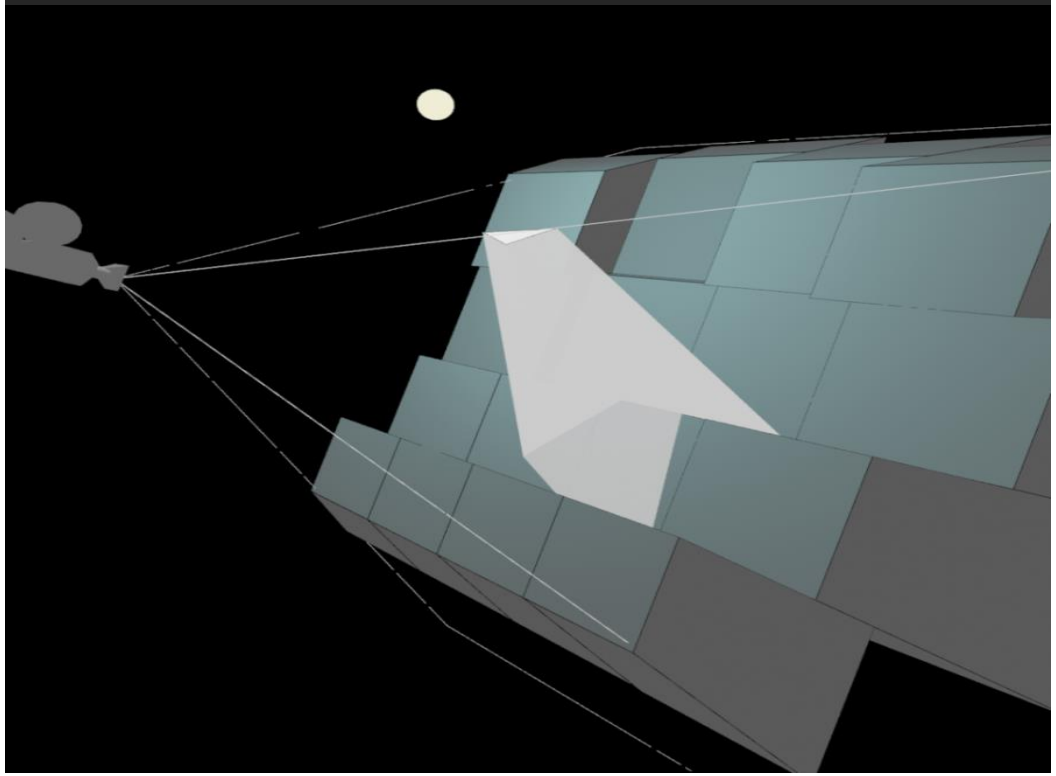


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy

View from camera

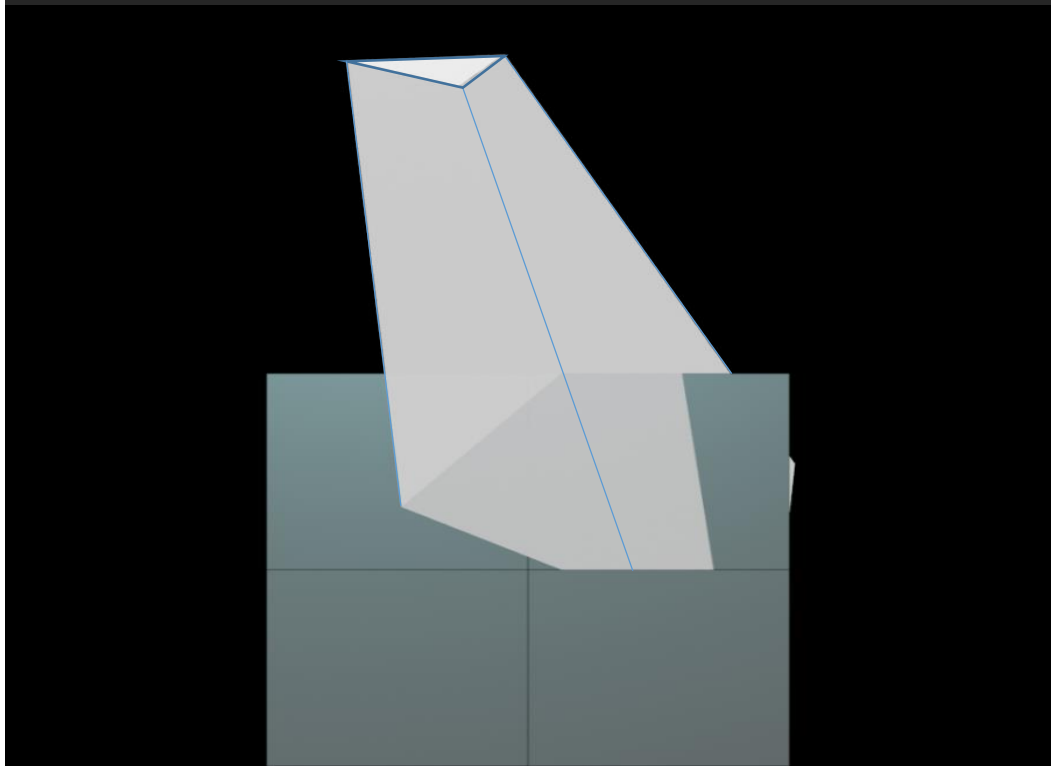


Alternative view

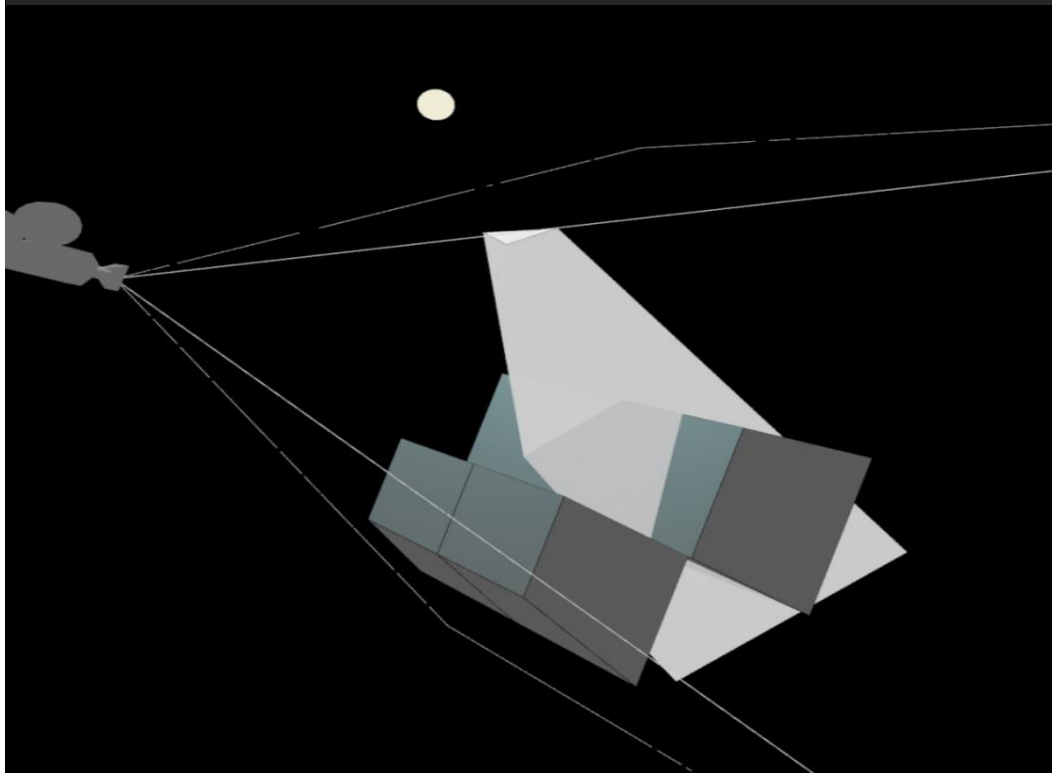


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Cull 4x4 tiles

View from camera

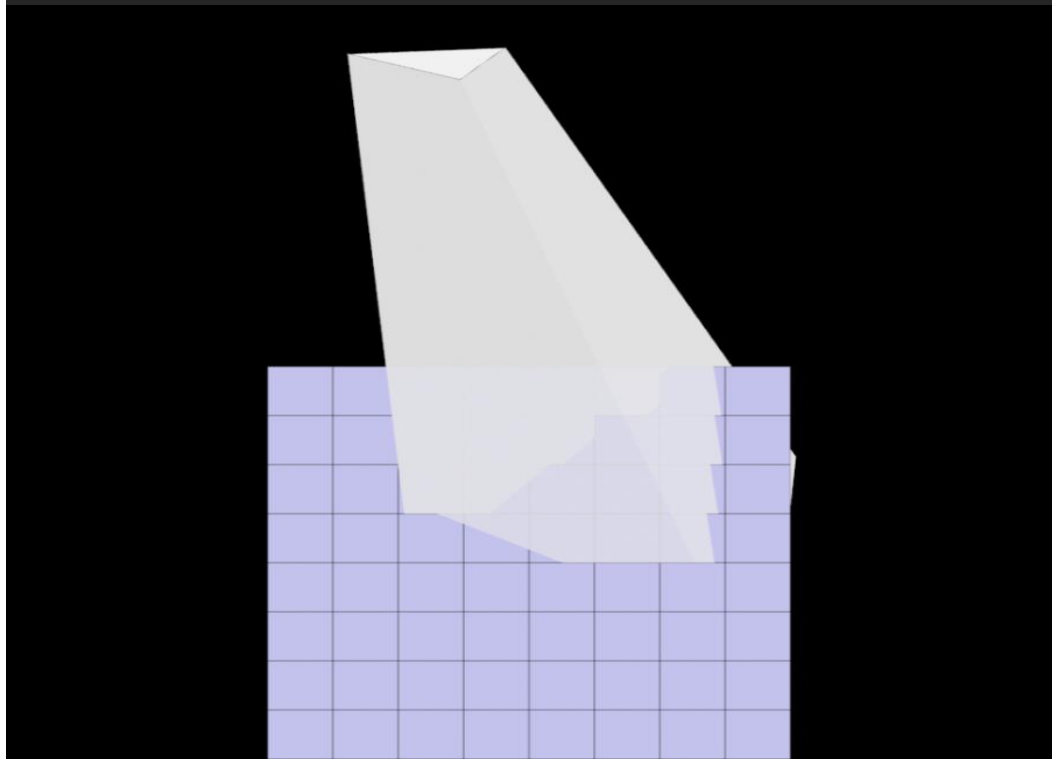


Alternative view

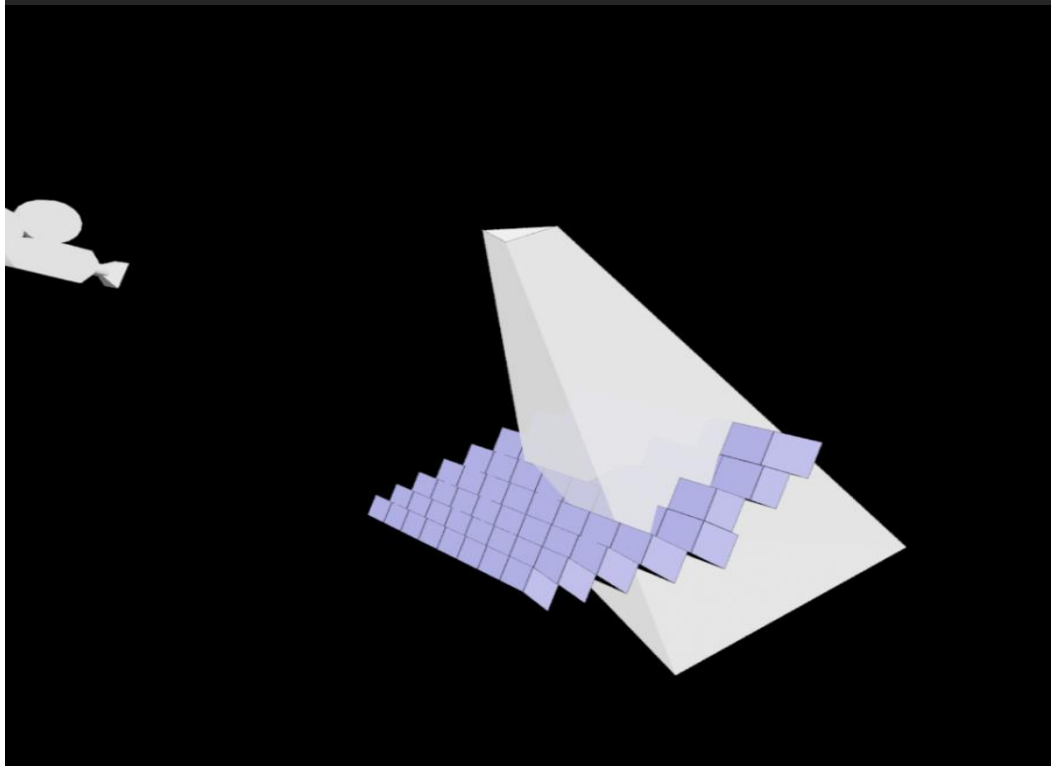


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Cull 4x4 tiles

View from camera

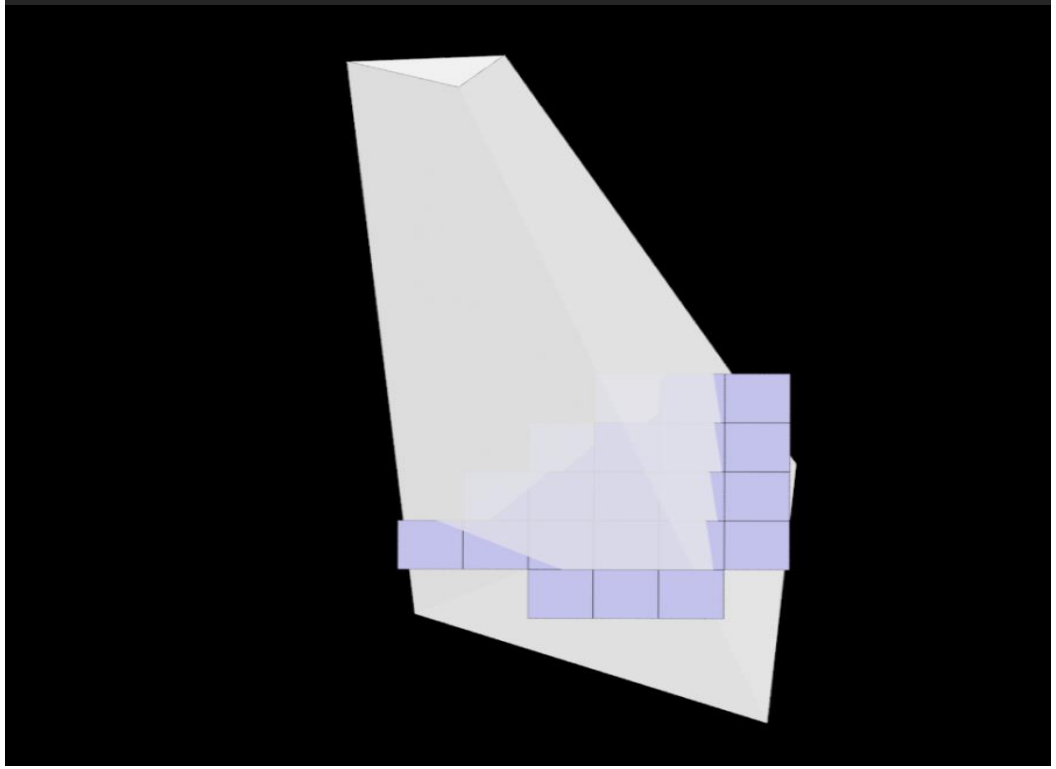


Alternative view

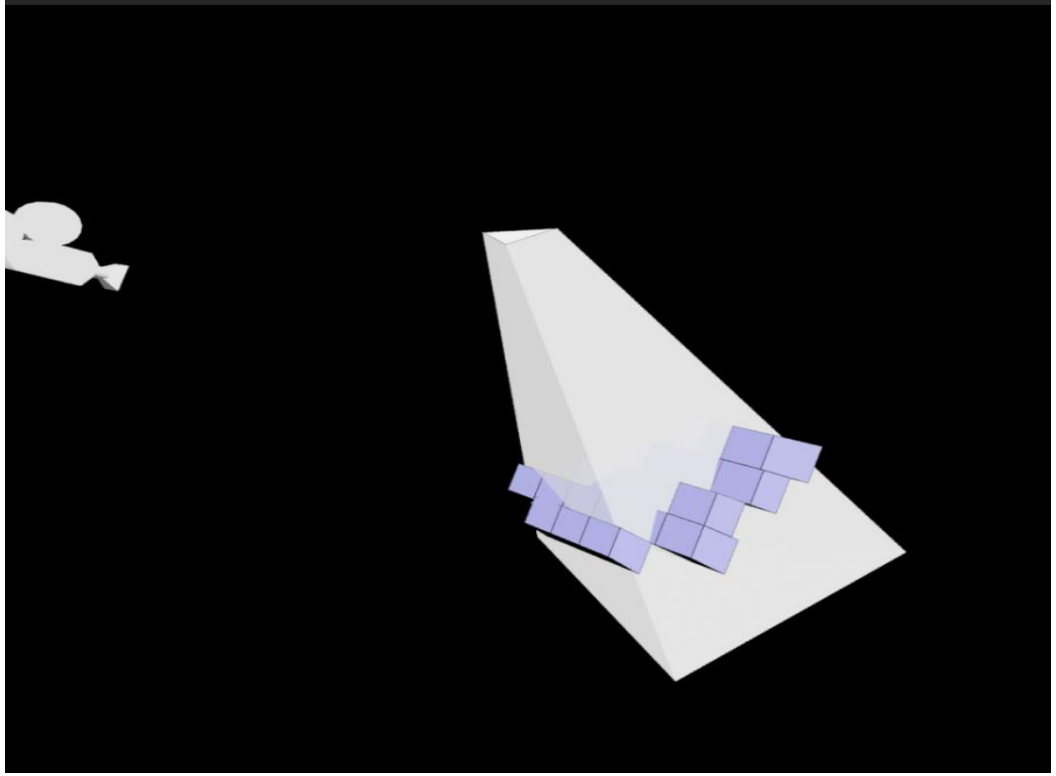


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Cull 4x4 tiles

View from camera

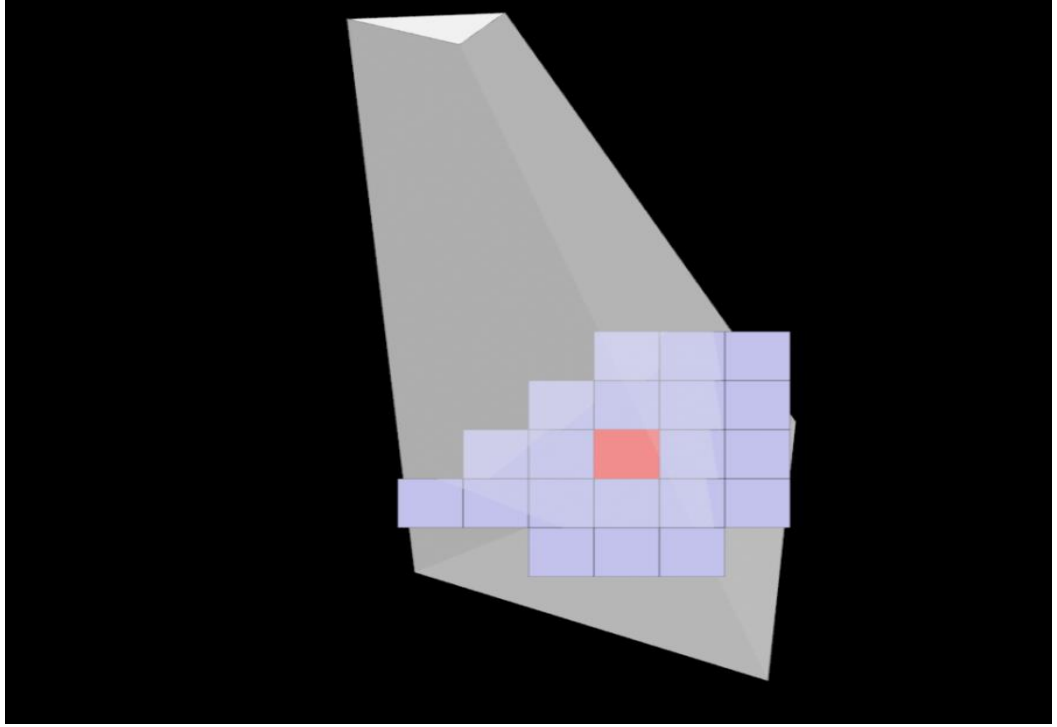


Alternative view

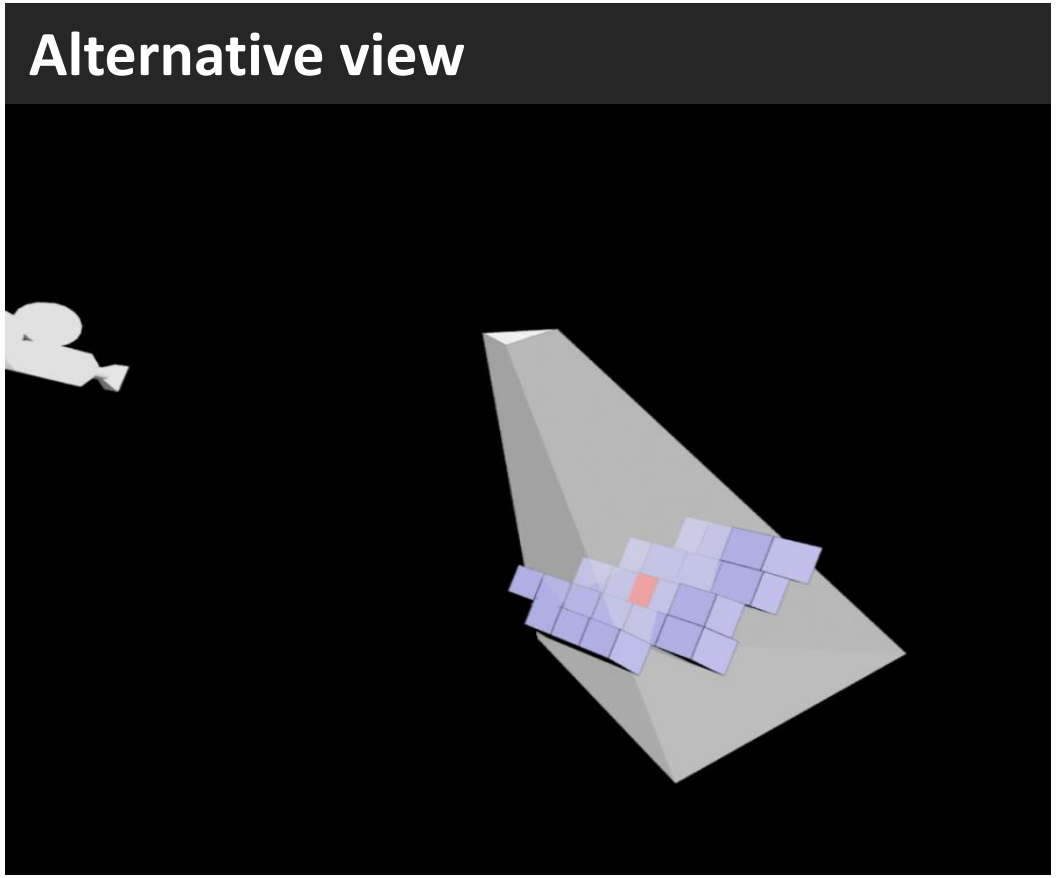


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Trivial Accept

View from camera

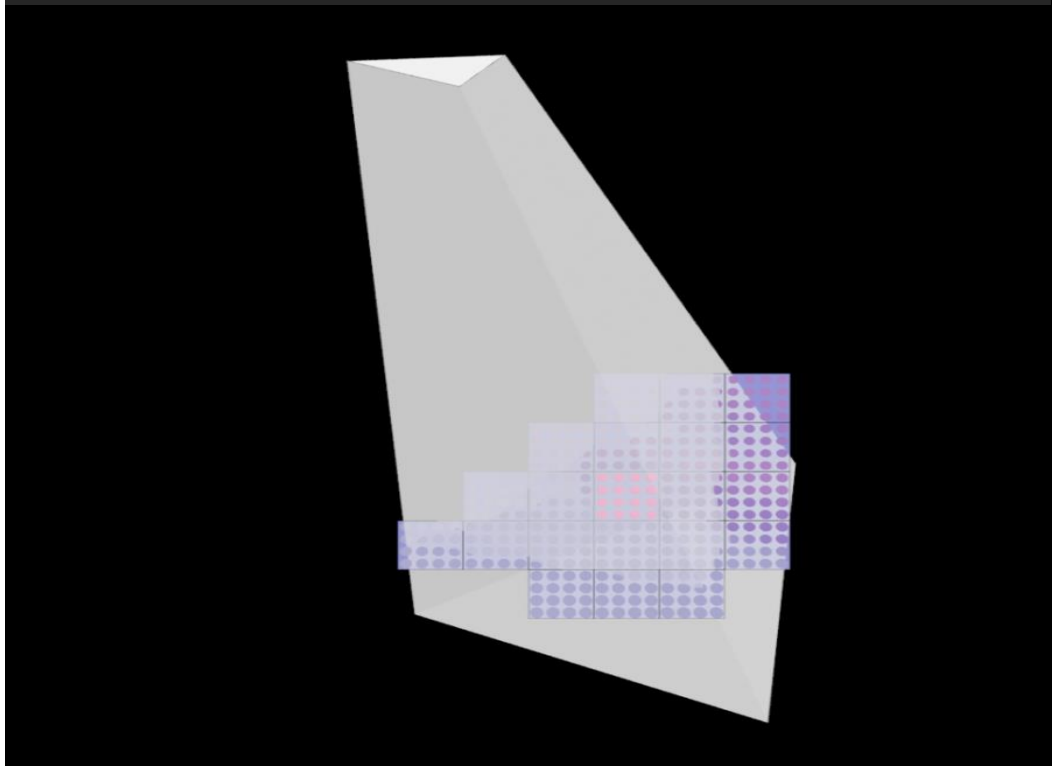


Alternative view

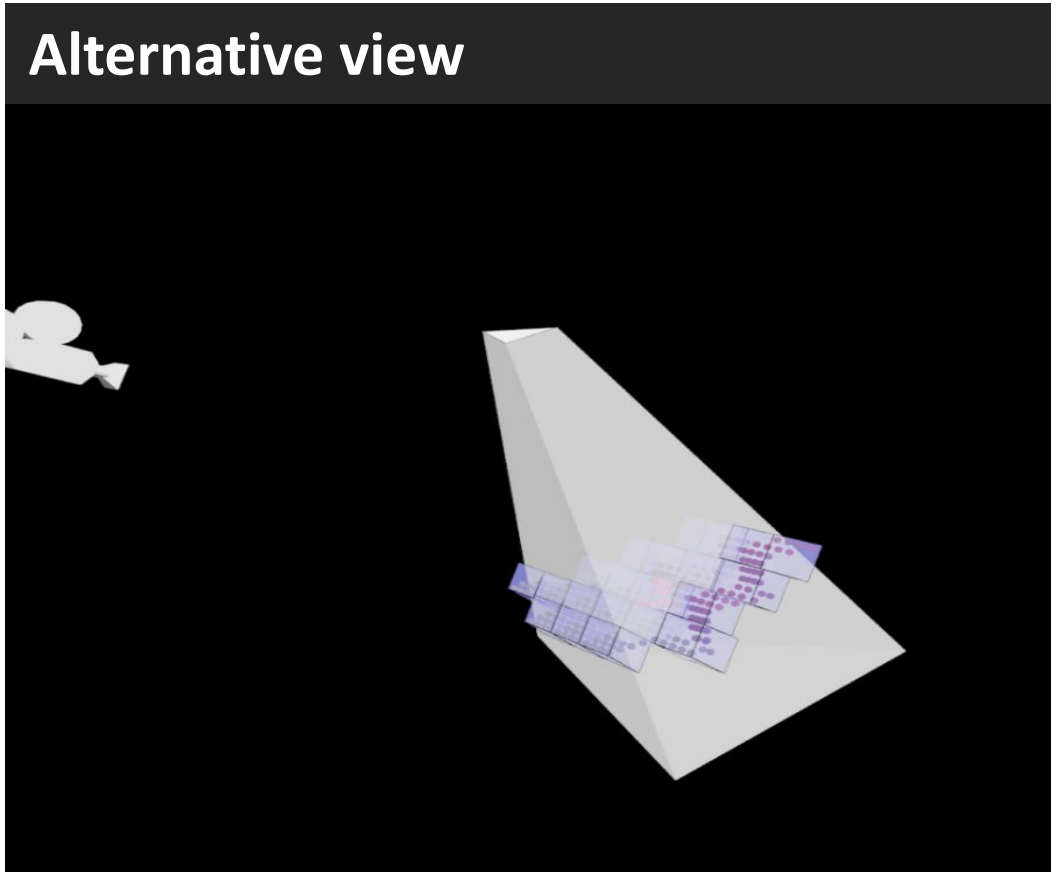


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Trivial Accept

View from camera

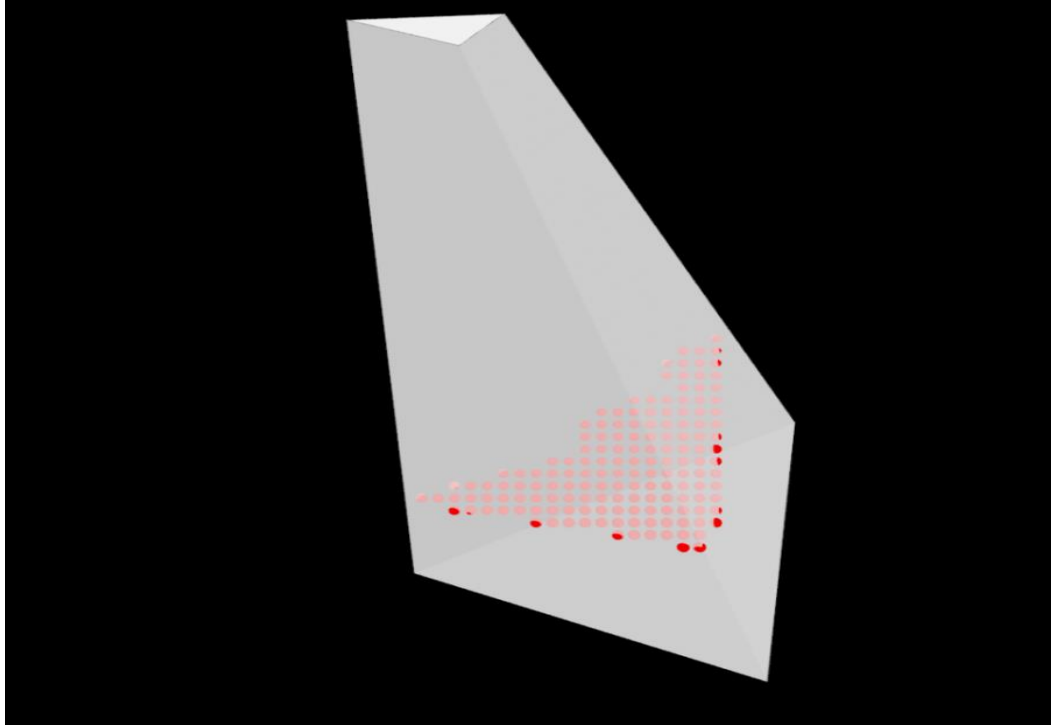


Alternative view

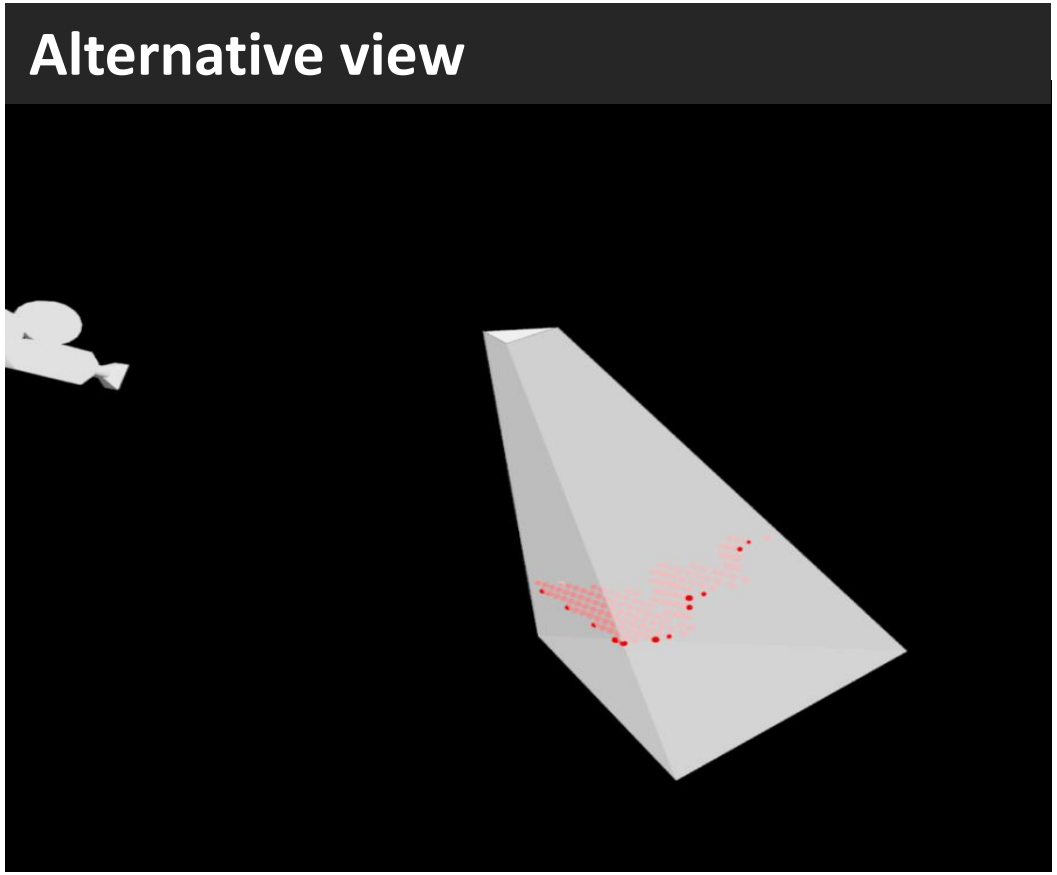


Per-Triangle Shadow Volumes. Algorithm. Traversing the hierarchy – Test samples

View from camera

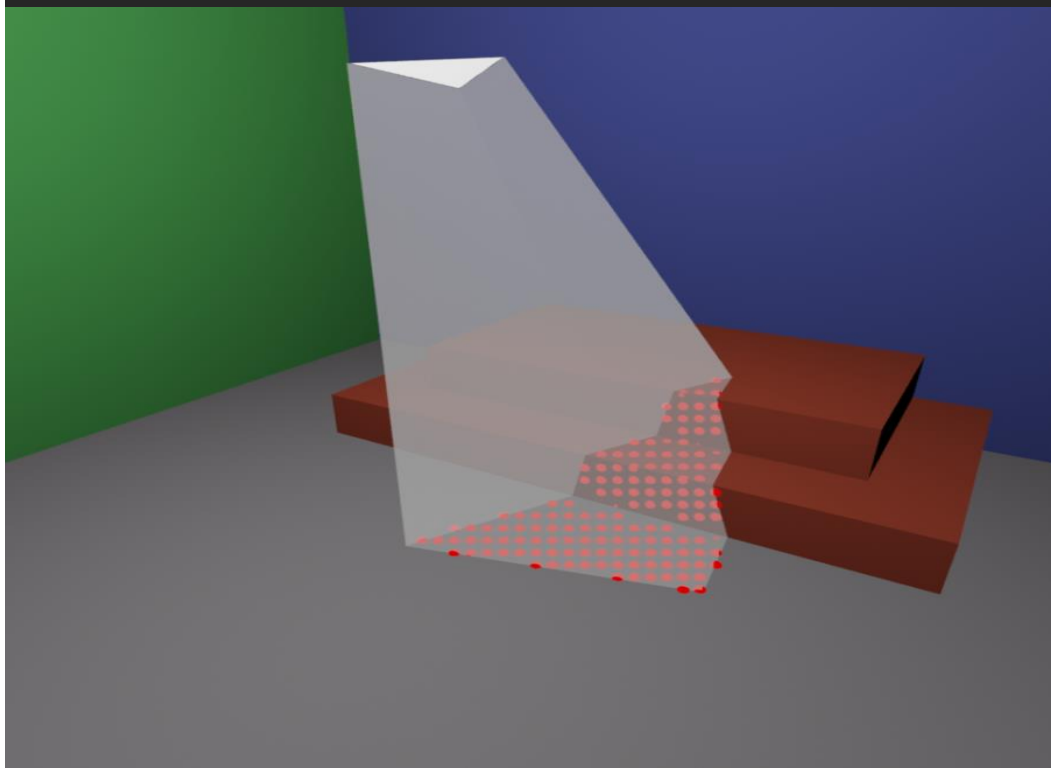


Alternative view

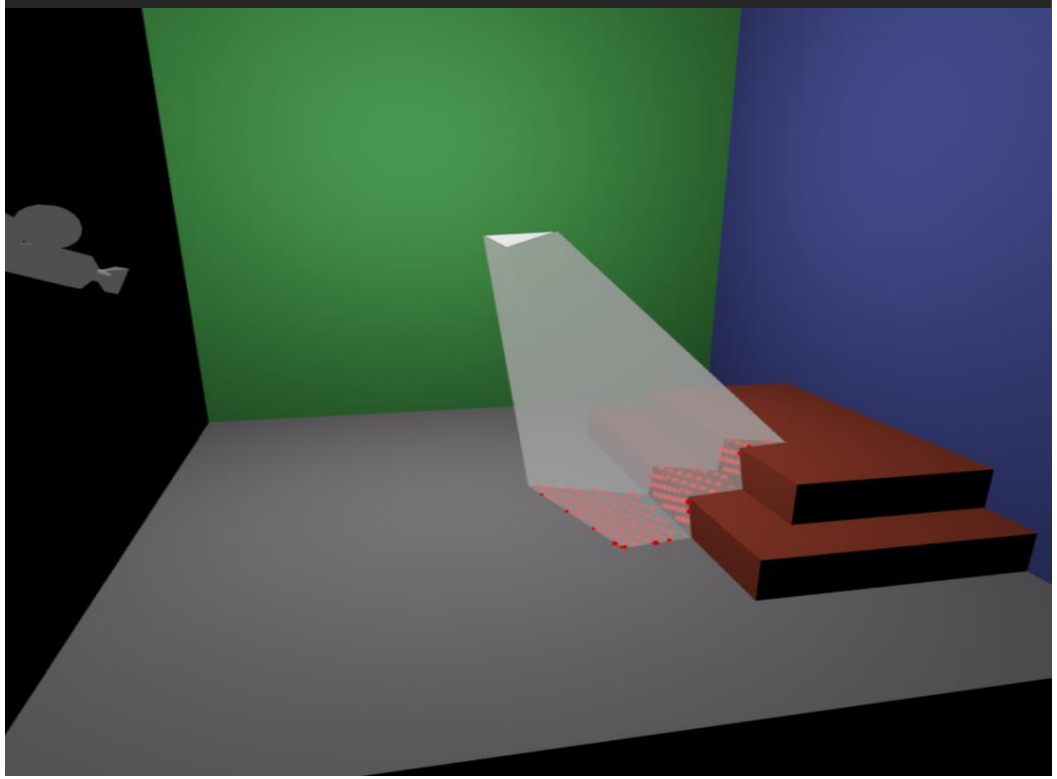


Per-Triangle Shadow Volumes. Algorithm

View from camera



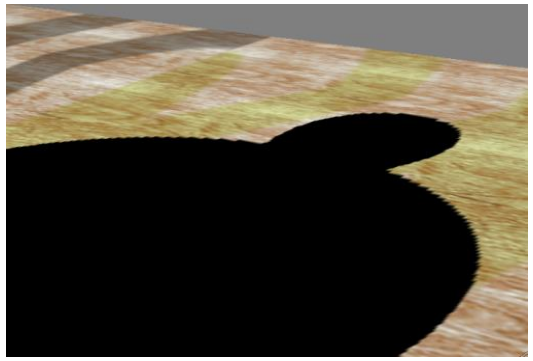
Alternative view



Shadow Maps vs Shadow Volumes

Shadow Maps

- *Good*: Handles any rasterizable geometry, lookup: **constant cost** regardless of complexity, map can sometimes be reused. **Very fast.**
- *Bad*: Frustum limited. **Jagged shadows** if res too low, **biasing** headaches.
  - Solution: next talk…



Shadow Volumes

- *Good*: shadows are **sharp**. Handles omni-directional lights.
- *Bad*: **3 passes.** Shadow polygons must be generated and rendered → lots of polygons & **fill**
  - Solution: Culling & Clamping or Per-triangle SV using CUDA.

Thanks

- Used slides from:
  - Ulf Assarsson, Lukai Lan, and others

Hard shadows

Pere-Pau Vázquez
ViRVIG – UPC