DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Shadows

Shadows – Global Illumination

If it looks like computer graphics,
It is not good computer graphics

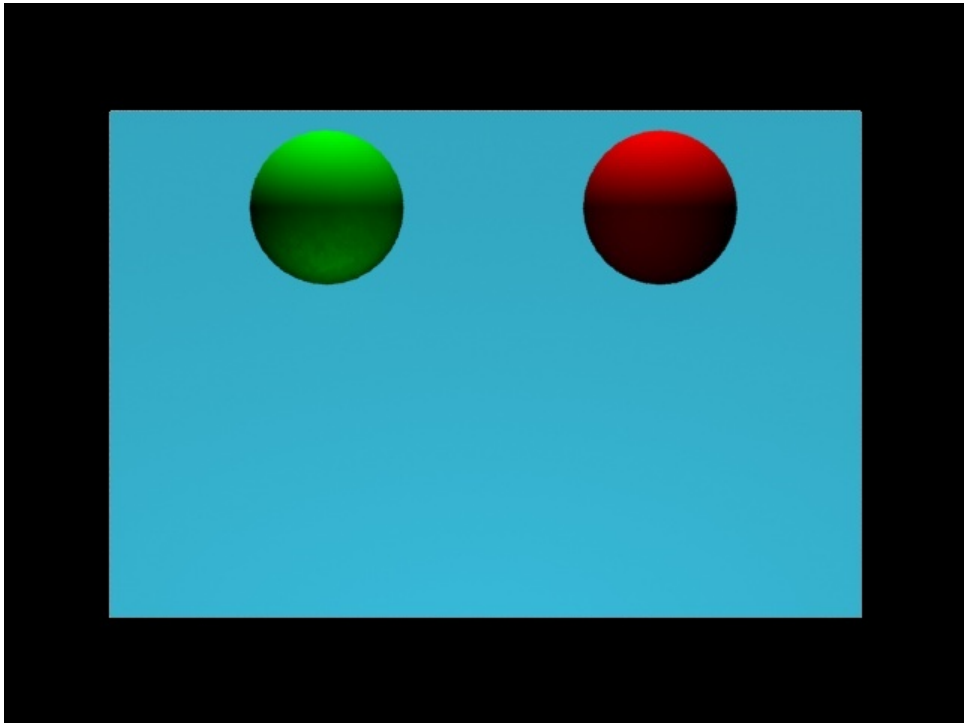
-- Jeremy Birn

Introduction

- The role shadows play in our visual perception of an environment is subtle, yet it is a vital one.
- Shadow position and orientation provide information on how objects in space relate to each other.
- Shadows can also convey information of other objects that are not even in view.

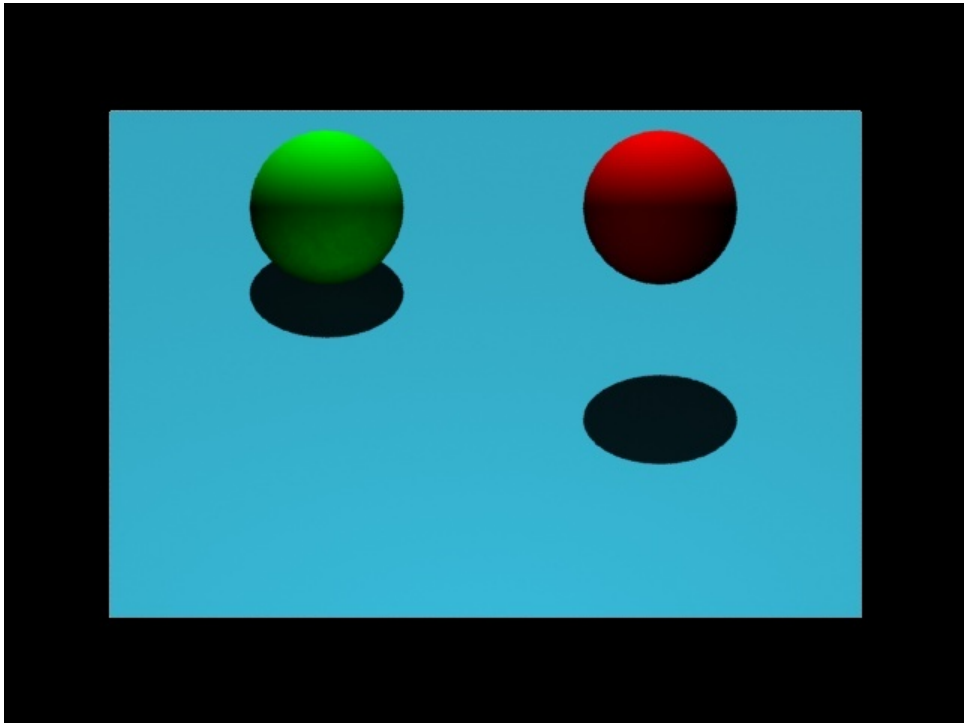
Spheres without Shadow

- Cannot tell sphere's position relative to the plane nor the viewer.



Spheres with Shadow

- The red sphere is farther from the plane and closer to the viewer.



Spheres with Shadow *(cont'd)*

- Shadows provide the missing information.
- They 'anchor' the spheres position in space relative to the plane and to the viewer.
- It is important to deal with shadow in computer graphics.

Shadow Properties

- In order to render shadows, two shadow properties must be calculated. They are:
  - Shadow shape
  - Shadow intensity

Shadow Shape

- Trivial to calculate when the shadow is cast by a simple object onto a plane.
- Get very complicated when the blocker(s) and/or the receiver(s) are not ‘simple’.

Shadow Intensity

- Shadow is formed due to the local decrease in direct light because some objects block the light path to the surface being viewed.

- Light intensity of a shadowed surface does not usually reduce to zero. Why?

Shadow Intensity

- Shadow is formed due to the local decrease in direct light because some objects block the light path to the surface being viewed.
- Surface under shadow receives light reflected from other objects (indirect illumination)
  - Ambient term in the Phong Model
- Standard shadow algorithms do not deal with indirect light.

Shadow Types

- As far as shadow calculations are concerned, light sources can be categorized into:
  - Point light source
  - Area light source

Point Light Source

- Point light source is represented as an single point in space. It emits light from that point.
- A given point on a surface can either see the light completely or not see the light at all.
- Because of the binary nature of the light visibility, the shadow produced will have hard edges.
- Rarely occurs in real life because most light sources are not idealized point light sources.

Hard shadows

https://maverick.inria.fr/Research/RealTimeShadows/hard_vs_soft.html#

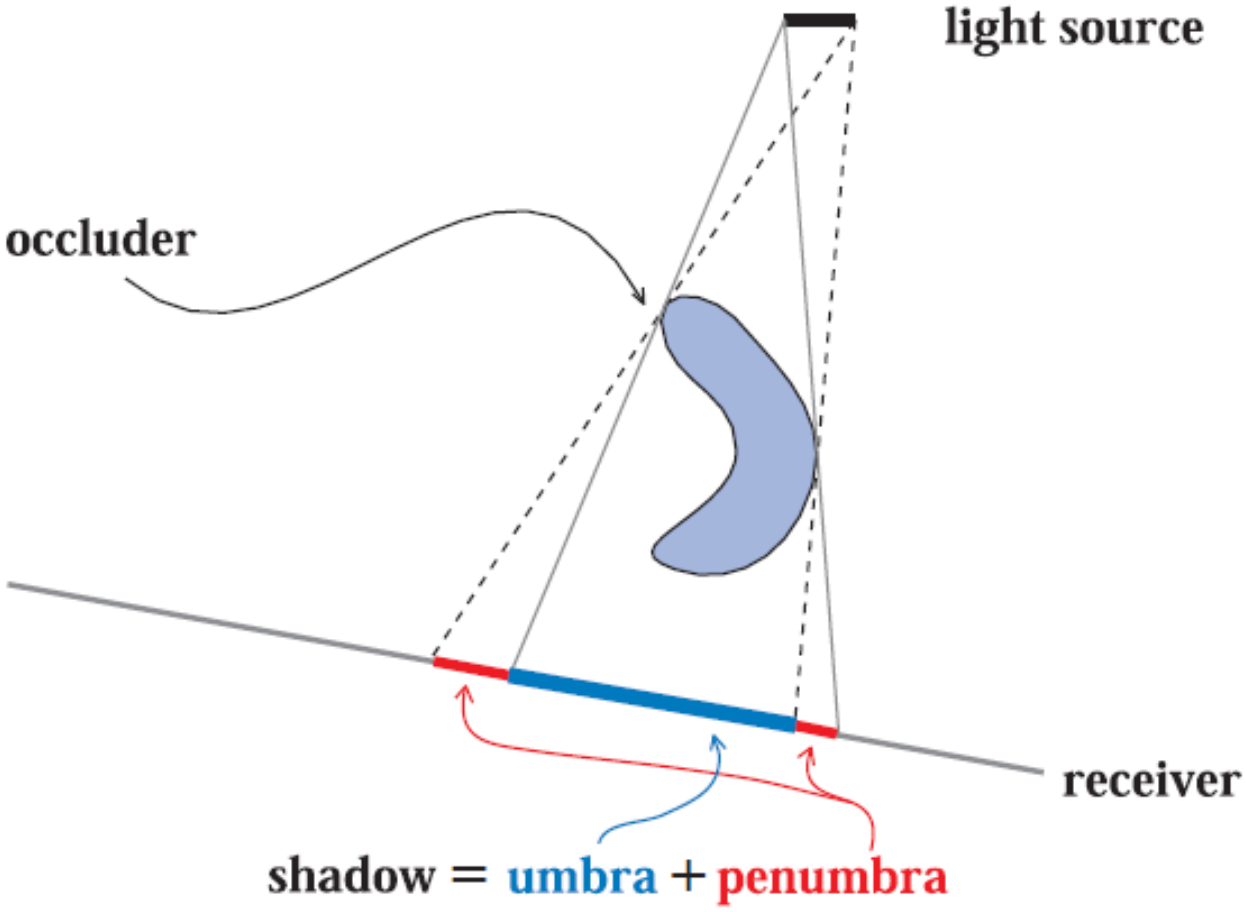
Area Light Source

- In area light source, light is emitted from an area in space instead of just a single point.
- A given point on a surface can see the light completely, not see it at all, or see it partially.
- Due to the light visibility state from a given point, the shadow produced is categorized into umbra and penumbra.

Area Light Source *(cont'd)*

- Umbra is the region on the surface that cannot see the light at all.
- Penumbra is the region of the surface that can see the light partially.
- By definition, the penumbra always surrounds the umbra.
- The relative size of these regions depends on the area light size, its distance to the blocker and distance from blocker to receiver.

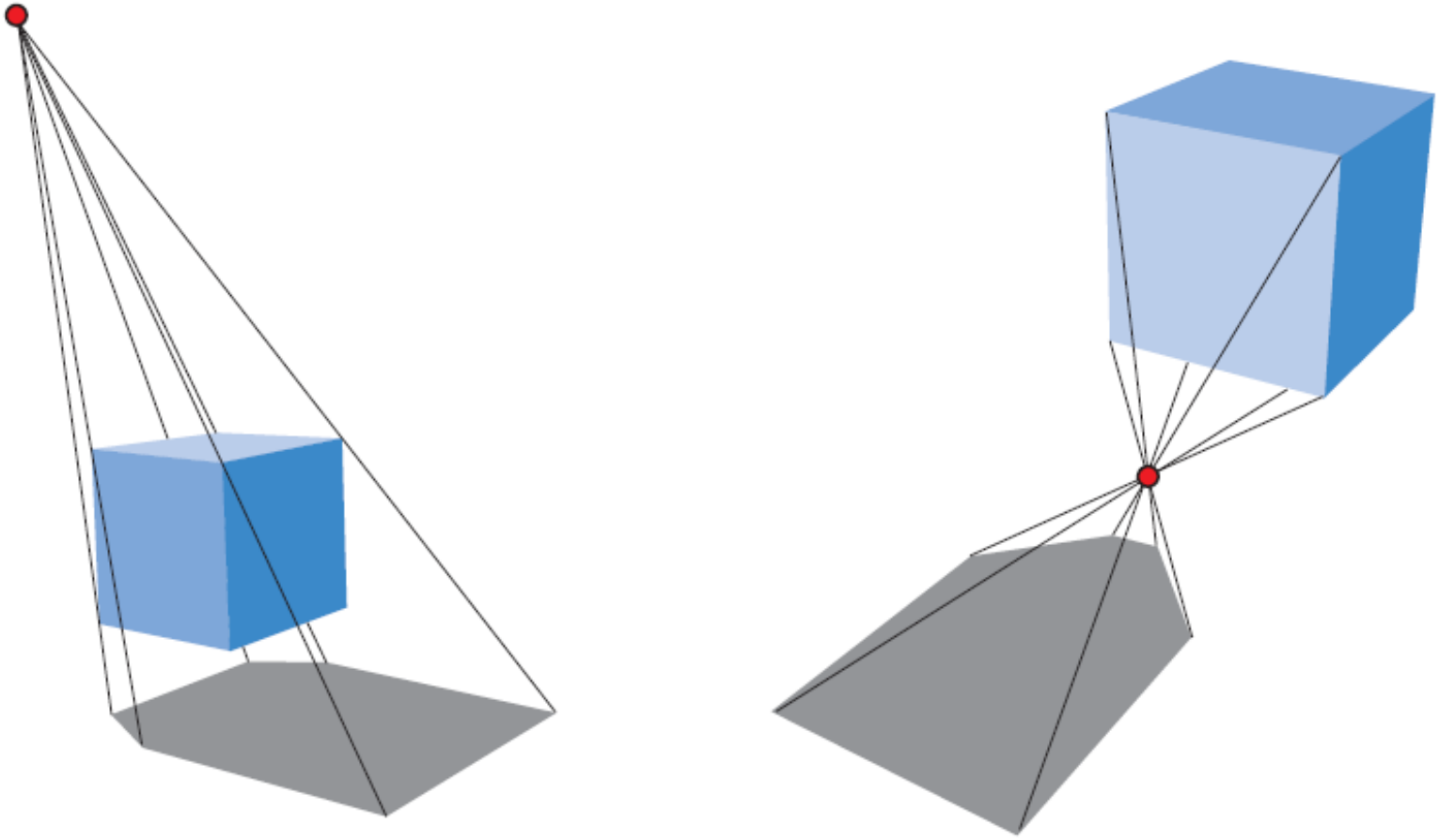
Shadows



Soft Shadows

https://maverick.inria.fr/Research/RealTimeShadows/hard_vs_soft.html#

Reverse Projection



Shadow Generation Algorithms

- Can be categorized into:
  - Integration of shadow polygons with a scan line hidden surface removal algorithm.
  - Shadow generation based on transformation and clipping.
  - Shadow volumes.
  - Shadow calculation using depth buffer.
  - Shadow calculation in recursive ray tracer.
  - Shadow calculation by radiosity method.

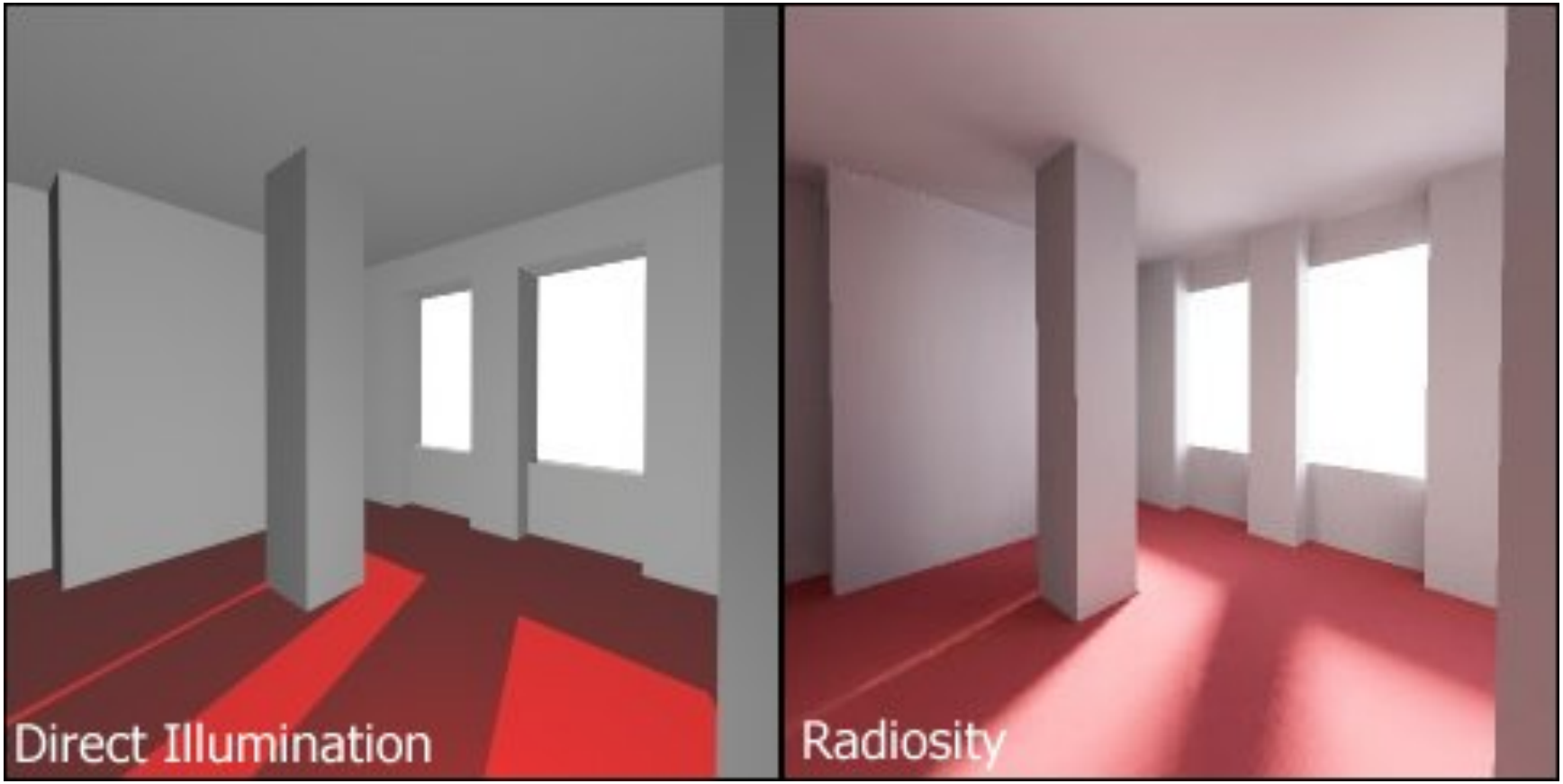
Shadow Generation Algorithms *(cont'd)*

- The ray tracing and radiosity methods will produce the most realistic shadow.
- Unfortunately, they are expensive and cannot be done in real time on low-end machines.
  - Search "real-time ray tracing" in Youtube/Google
  - NVIDIA Optix & Iray SDKs for programmers
- For real time application, shadow volumes and shadow map are the most commonly used methods.

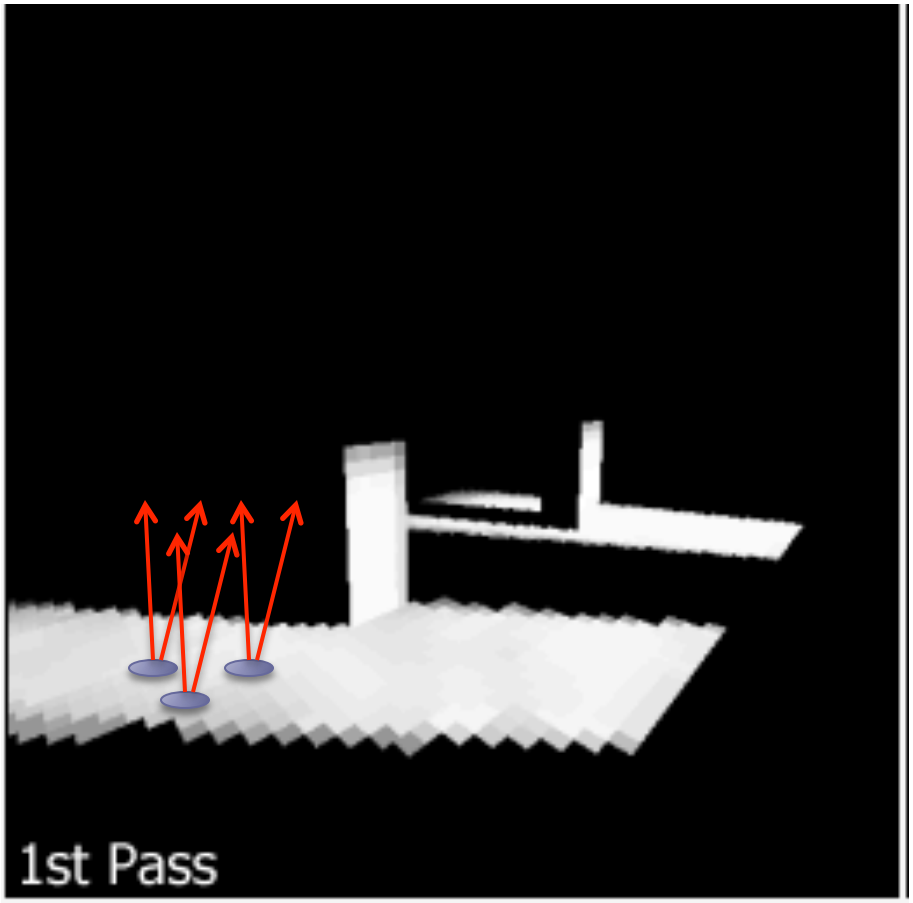
Ray Tracing

Radiosity

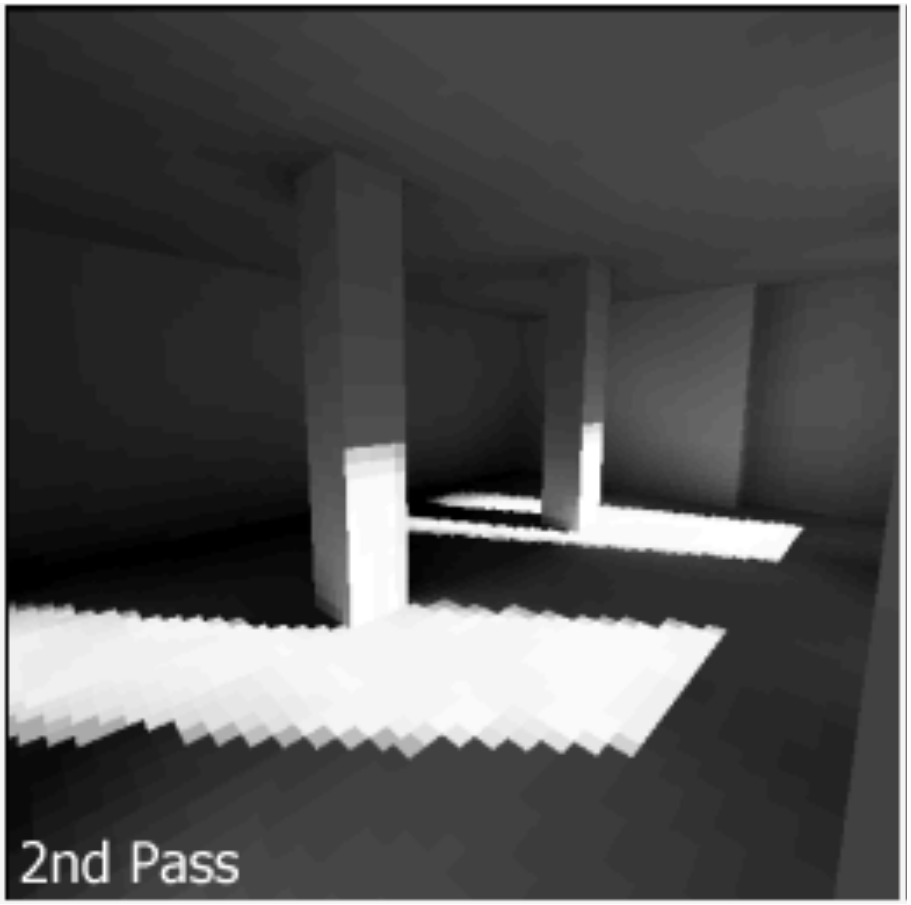
Comparing Phong Model to Radiosity



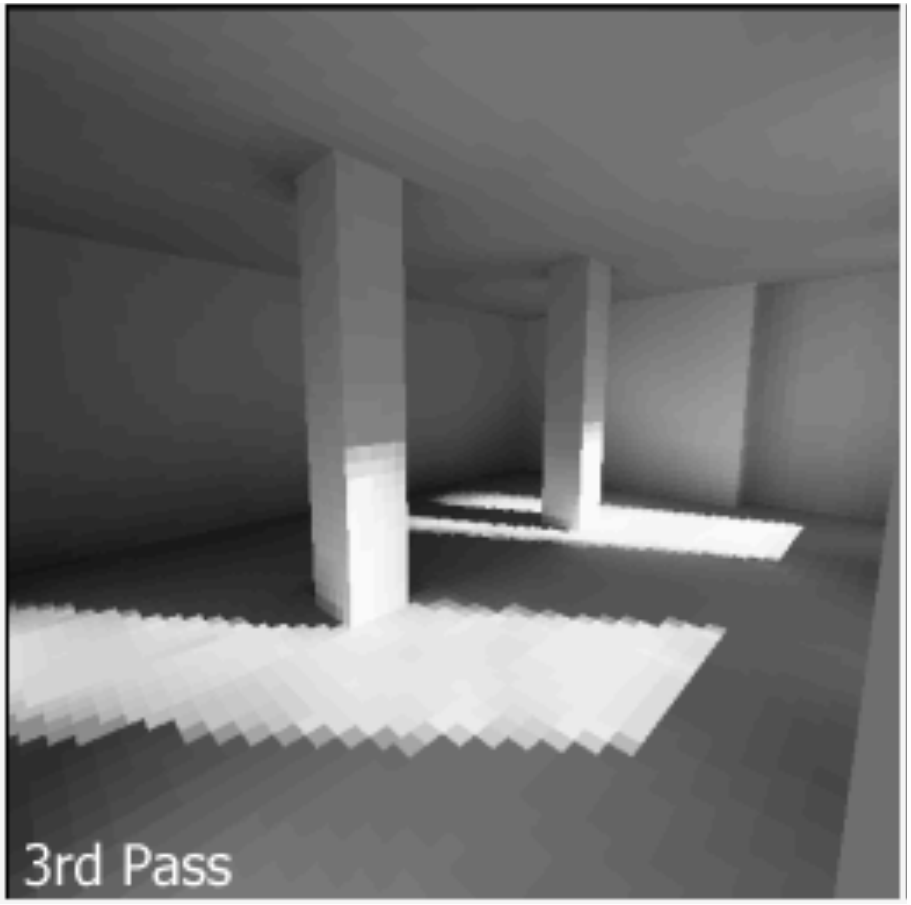
Radiosity Solution – Pass 1



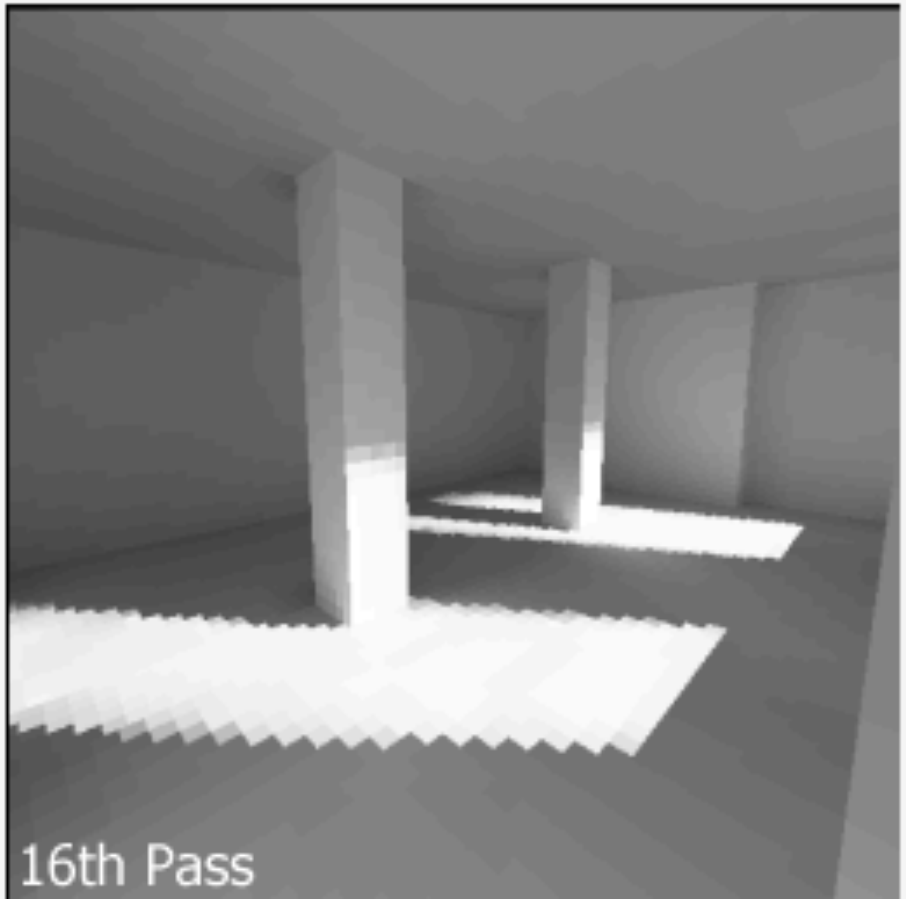
Radiosity Solution – Pass 2



Radiosity Solution – Pass 3



Radiosity Solution – Pass 16



Shadows in Phong Model

Shadow in Phong Model

- One very simple method to generate shadow in the scene is done by modifying the Phong lighting equation.

$$I_{tot} = I_{emissive} + I_{globalAmbient} K_a +$$

$$\sum_{i=0}^{n-1} Att_i * (I_{ambient})_i + Att_i * SpotEffect_i * Shdw_i * (I_{diffuse} + I_{specular})_i$$

- Where:
  - `Shdw` is 1 if the pixel being rasterized can see $light_i$. It is 0 otherwise.

Shadow in Phong Model *(cont'd)*

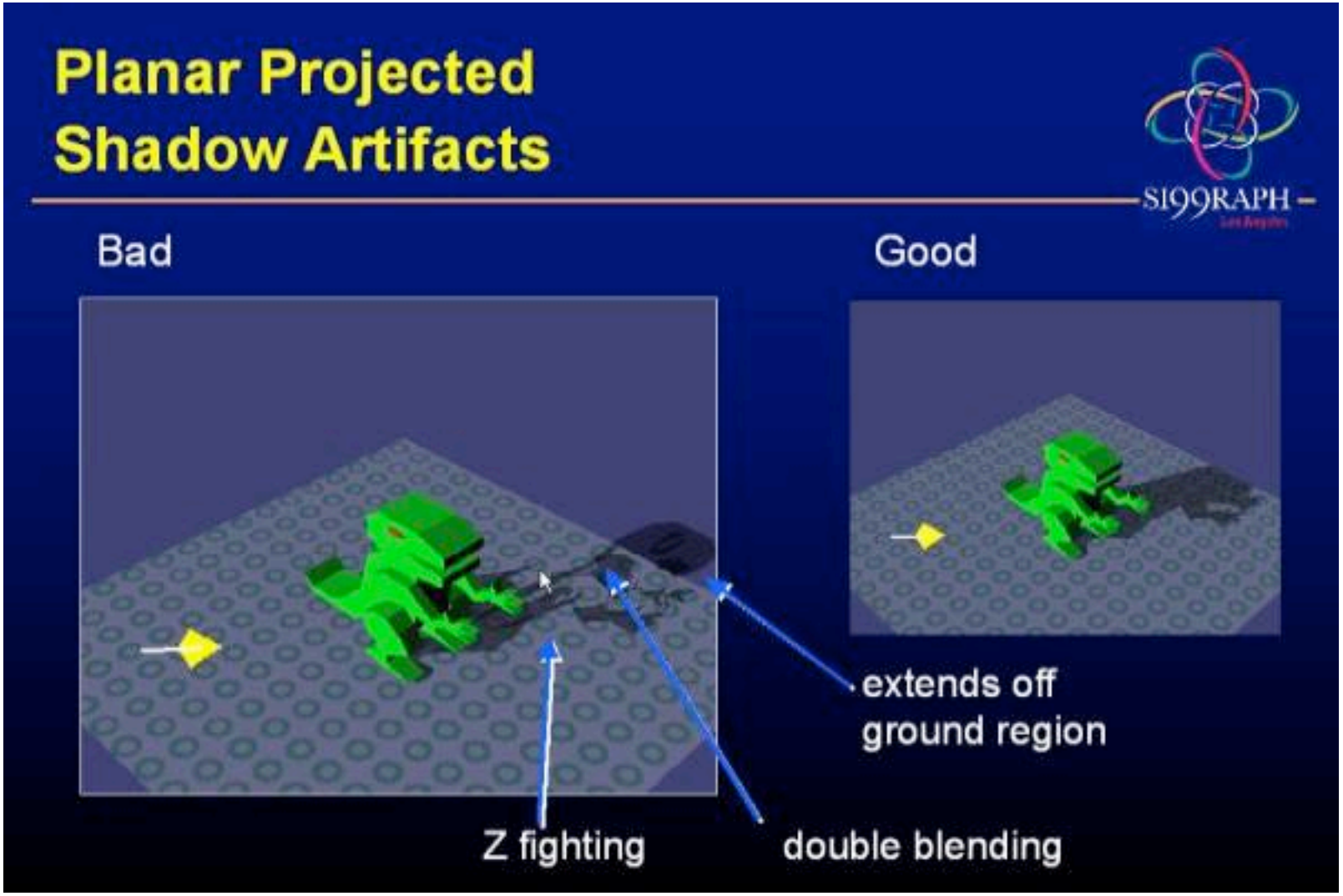
- Integrating the shadow calculation into the Phong lighting equation will produce a correct hard shadow.

- However, it is very expensive as a ray intersection has to be calculated **for each pixel of each object** to **each light** in the scene.

Planar Shadow

- Another simple method to generate shadow is by projecting the shadow blocker object onto a plane (typically the ground plane).

- It works by transforming the blocker object's vertices onto the plane and rendering the transformed object as a translucent object on top of the plane.



Shadow Volumes

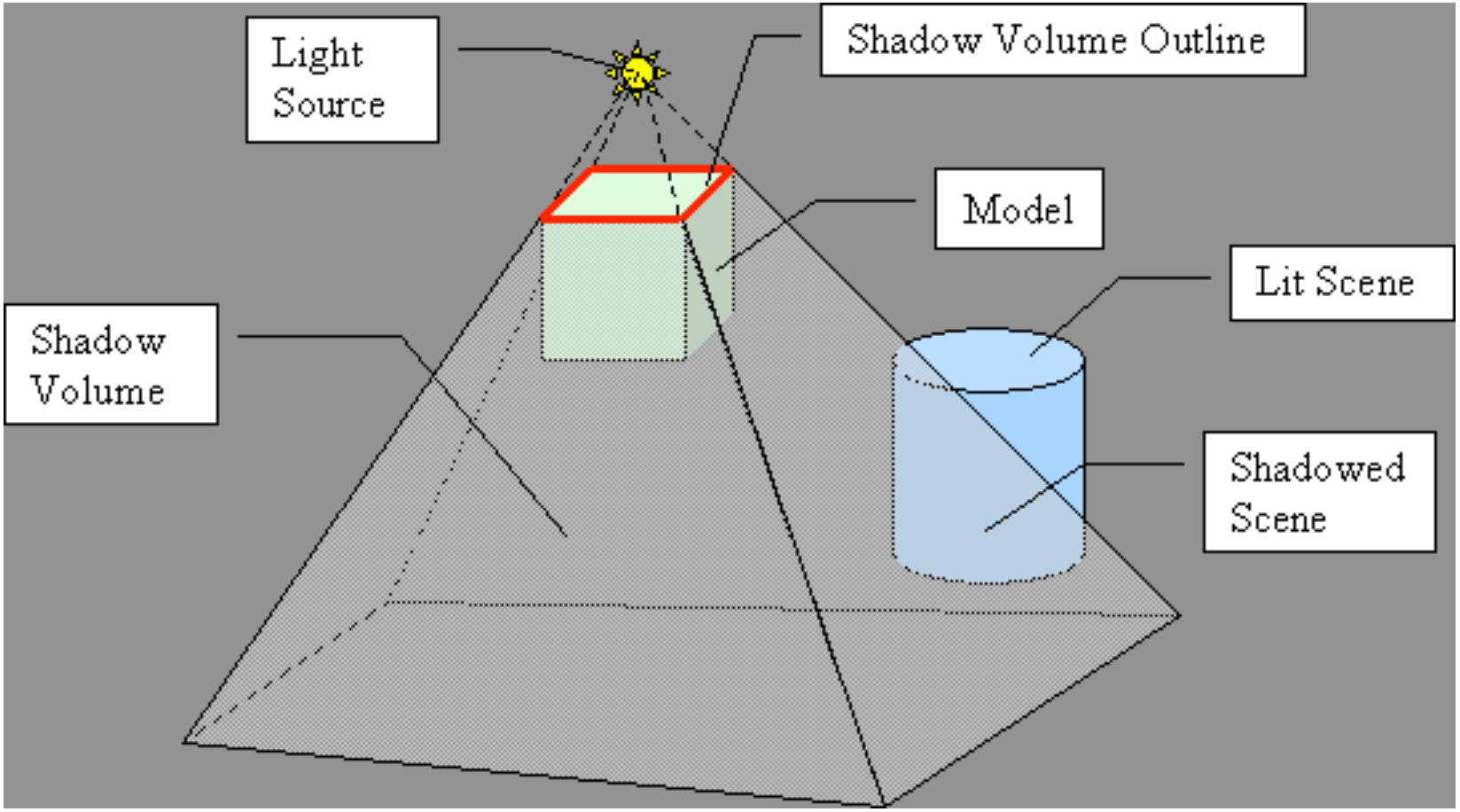
Shadow Volume

- The algorithm uses point light sources and cast shadow off polygonal objects only.
- Basically, given a point light and a blocker object, generate a projected shadow volume by which other objects within the volume is in shadow.
- Projected shadow volume is the region of space swept by the blocker object silhouette from the light and bounded also by the blocked back face (from light point of view) and view frustum.

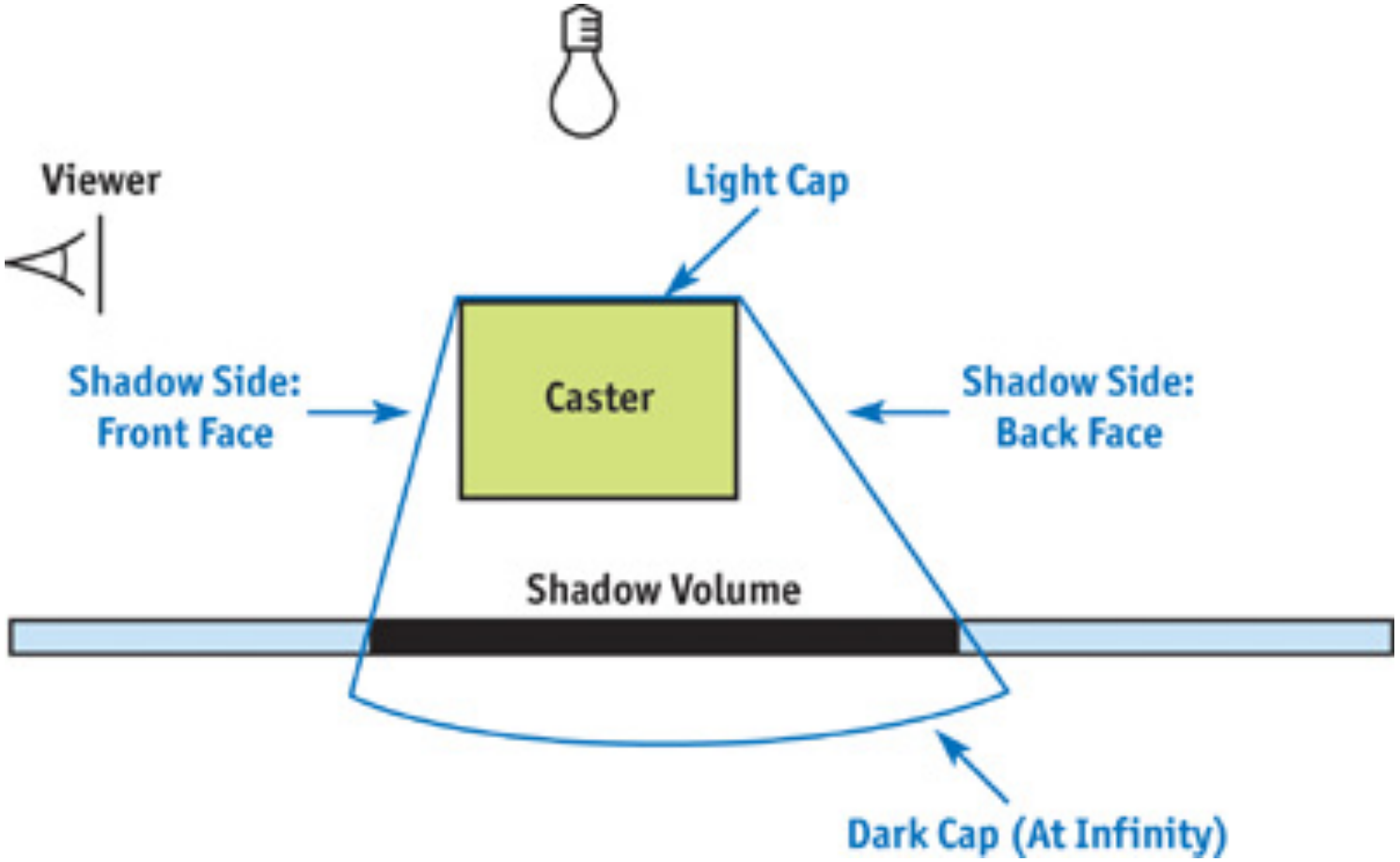
Shadow Volume *(cont'd)*

- It is a two pass-process:
  - Generate shadow volumes from each light
  - While rendering the scene, check if a given pixel is within a shadow volume
- How to generate a shadow volume from a point light and a blocker object?
- How to efficiently check if a pixel is within some volume?

Shadow Volume Schematic



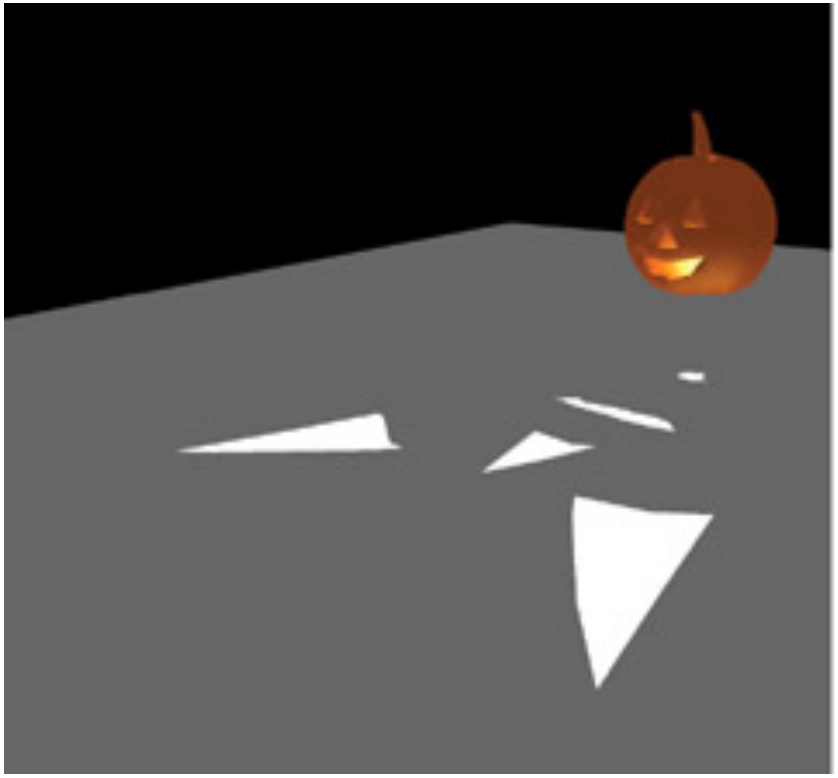
Shadow Volume Constituents



Advantages of Shadow Volumes

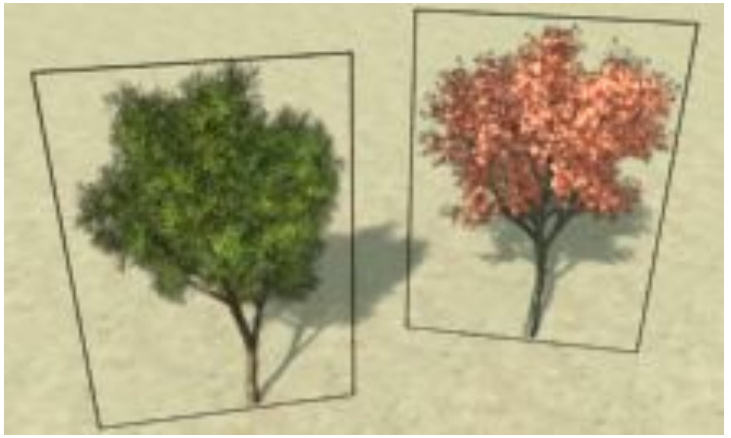
Self Shadowing Effect (FREE!)

More Shadows …

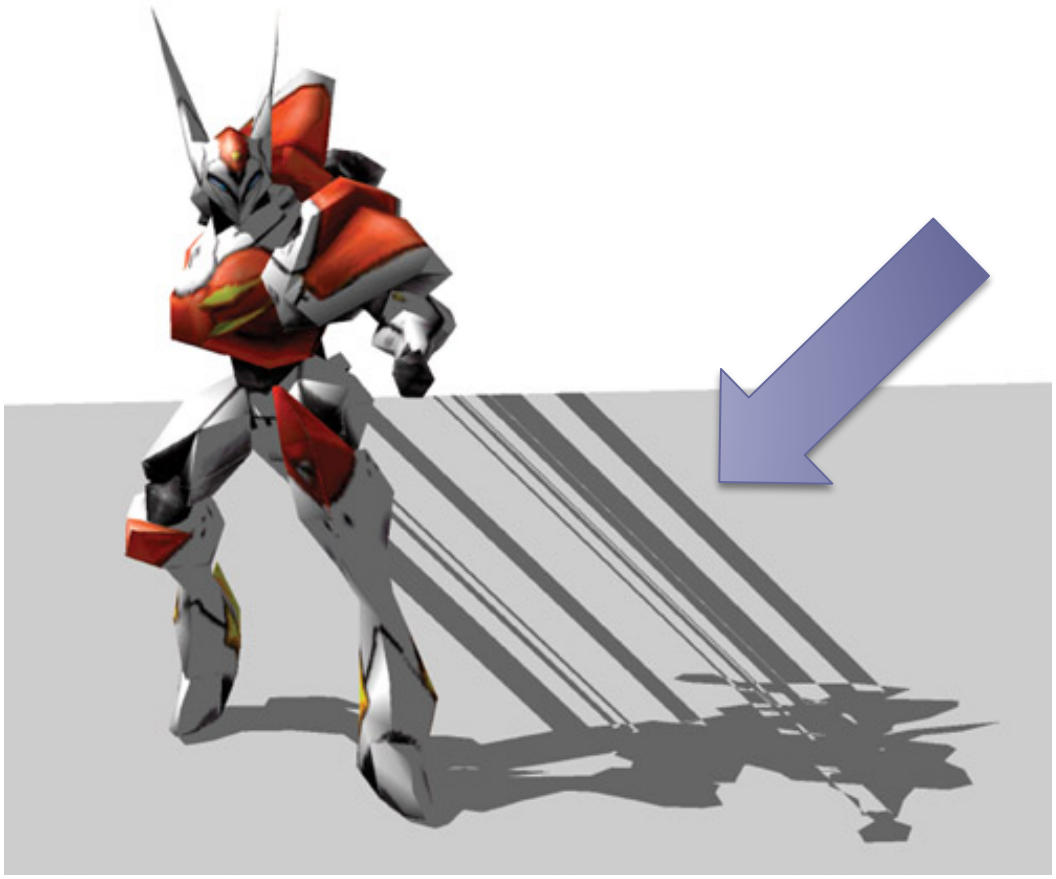


Where Shadow Volumes are not applicable -

- Shadow volumes require the object to have a well-defined geometry
- Does not work with
  - Billboards
  - Alpha-matte polygons
  - Particle system



Cracks in the Mesh



Best Applications for Shadow Volumes

- Camera and lights do not interfere with the environment
- All objects (shadows casters) lie strictly between the shadow receiver and the light objects

- Similar to Real-time Strategy Games

Age of Empires III

- Consistent light direction
- Camera does not come between light and shadow caster
  - Reverse Projection problems avoided

Shadow Volume Algorithm

- Two phases
  - Generating / Identifying silhouette edges
  - Rendering shadow volumes into the stencil buffer
  - Repeat for EACH light source
- A *Multipass* lighting algorithm

Single-Pass Vs Multipass

- All lights in the scene can be processed simultaneously
- The lighting contribution for every fragment is computed by a direct equation
- (Programming Assignment – Phong Illumination Model)

- Compute global light values
  - Ambient
- Compute for each light (separate pass)
  - Diffuse and specular
- Final step
  - Combine the light values from ambient and emissive with the contribution of each light

Generating Shadow Volume

- In order to generate a shadow volume for a given object from a given point light, we need to identify the silhouette of the object from the light point of view.
- Once the silhouette is identified, we duplicate each edge of the silhouette and project the duplicate on the far plane.
- From each pair of the original edge and its duplicate, we can construct a quad that makes one of the shadow volume side.

Generating Shadow Volume *(cont'd)*

- Pseudo code on generating a shadow volume for a given object from a given point light:
  - Identify the silhouette of the object from the light point of view
  - Let L be the light position
  - For each edge in the silhouette
    - Let E be the current edge
    - Let A_0 and B_0 be the two end points of E
    - Duplicate A_0 and B_0. Let A_1 and B_1 be the duplicate
    - Place A_1 along the vector LA_0 at a large distance away from A_0. Do the same for B_1 with respect to LB_0.
    - Construct quad $A_0A_1B_1B_0$ and add it to the shadow volume face list.

Identifying the Silhouette

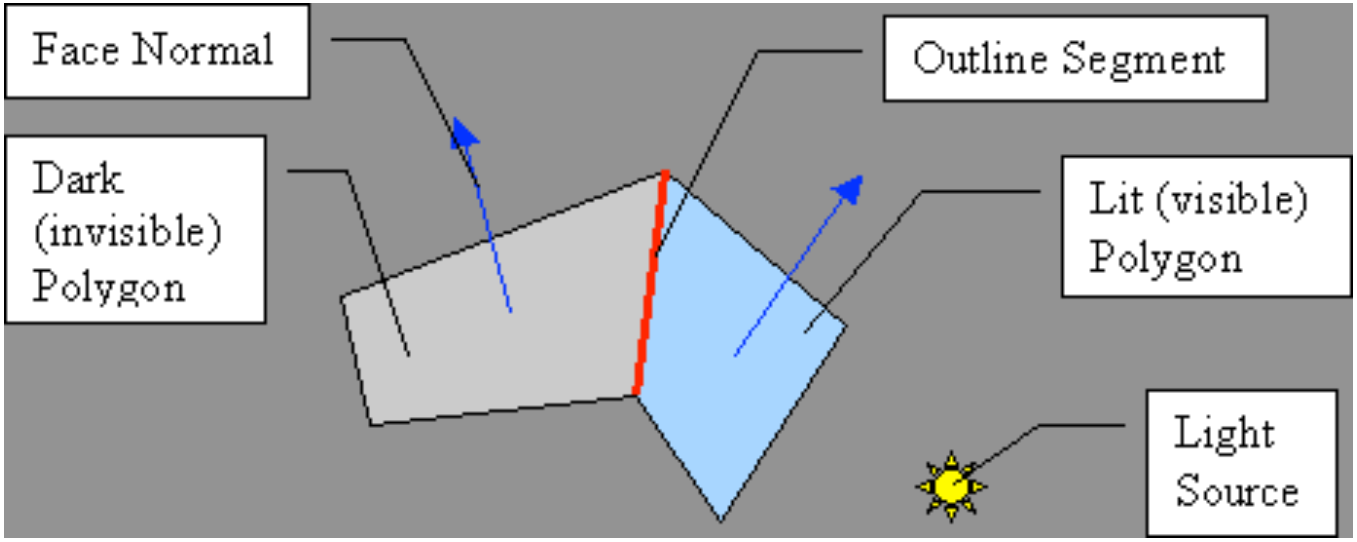
- What is the silhouette of an object?

Identifying the Silhouette *(cont'd)*

- What is the silhouette of an object?

  The silhouette of an object is a collection of edges where each of those edge:
  - Shared by a triangle that can see the light and a triangle that cannot see the light
  - Belong to a triangle that can see the light but have no neighbor along that edge

Silhouette



Identifying the Silhouette *(cont'd)*

- In addition to the usual data stored for each triangle, we also need to store a light visibility flag and the 'neighboring information'.
- The light visibility flag is a boolean that is true if the triangle can see the light and false otherwise.
- The 'neighboring information' for a given triangle is an index (or a pointer), for each of its edge, to a triangle that share that particular edge.

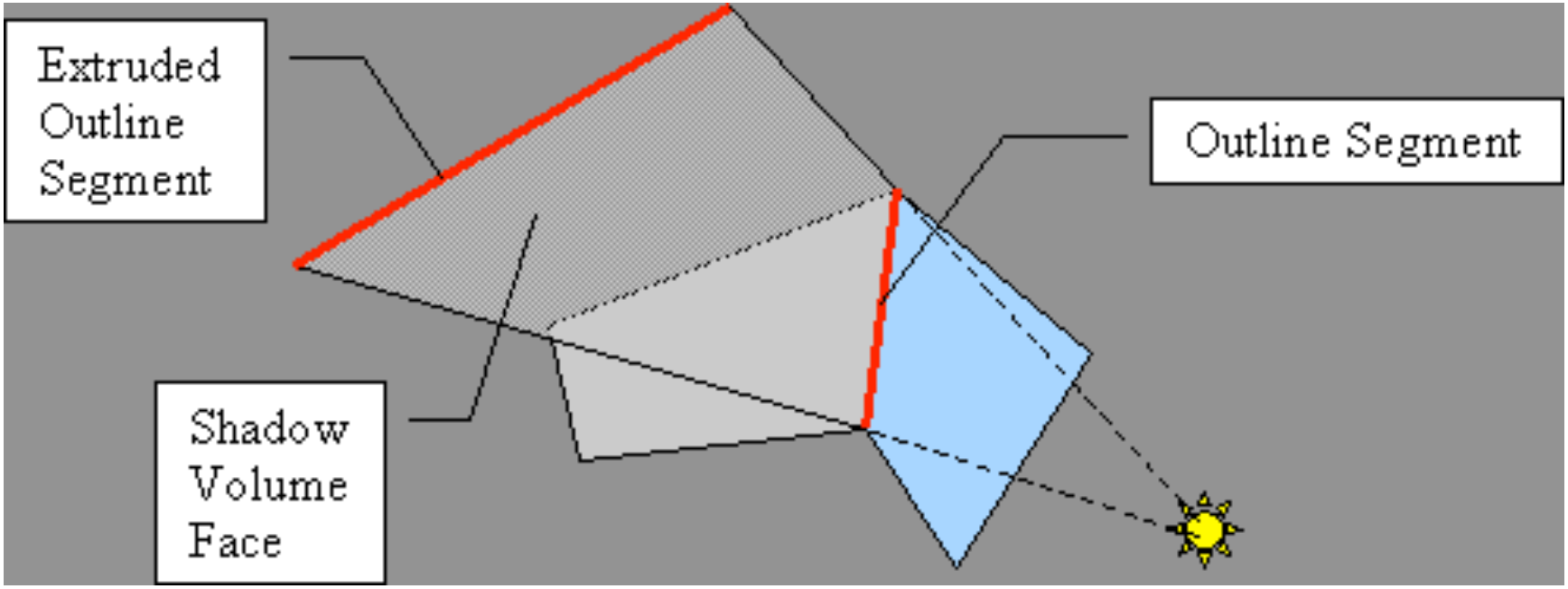
Identifying the Silhouette *(cont'd)*

- Pseudo code to identify the silhouette:
  - For each triangle T in the object
    - Set the light visibility to true if it can see the light, false otherwise.
  - For each triangle T in the object
    - If T cannot see the light, continue
    - For each edge E in T
      - If E is not shared with other triangle or if E is shared with a triangle that cannot see the light, add E to the silhouette edge list

Generating Shadow Volume - Note

- Instead of identifying the silhouette and generating the shadow volume from it, we can generate a shadow volume for each triangle in the object that can see the light.
- However, the amount of memory could get really large and it could get really expensive to process the data.
- It is not practical in general, but for the simplest object.

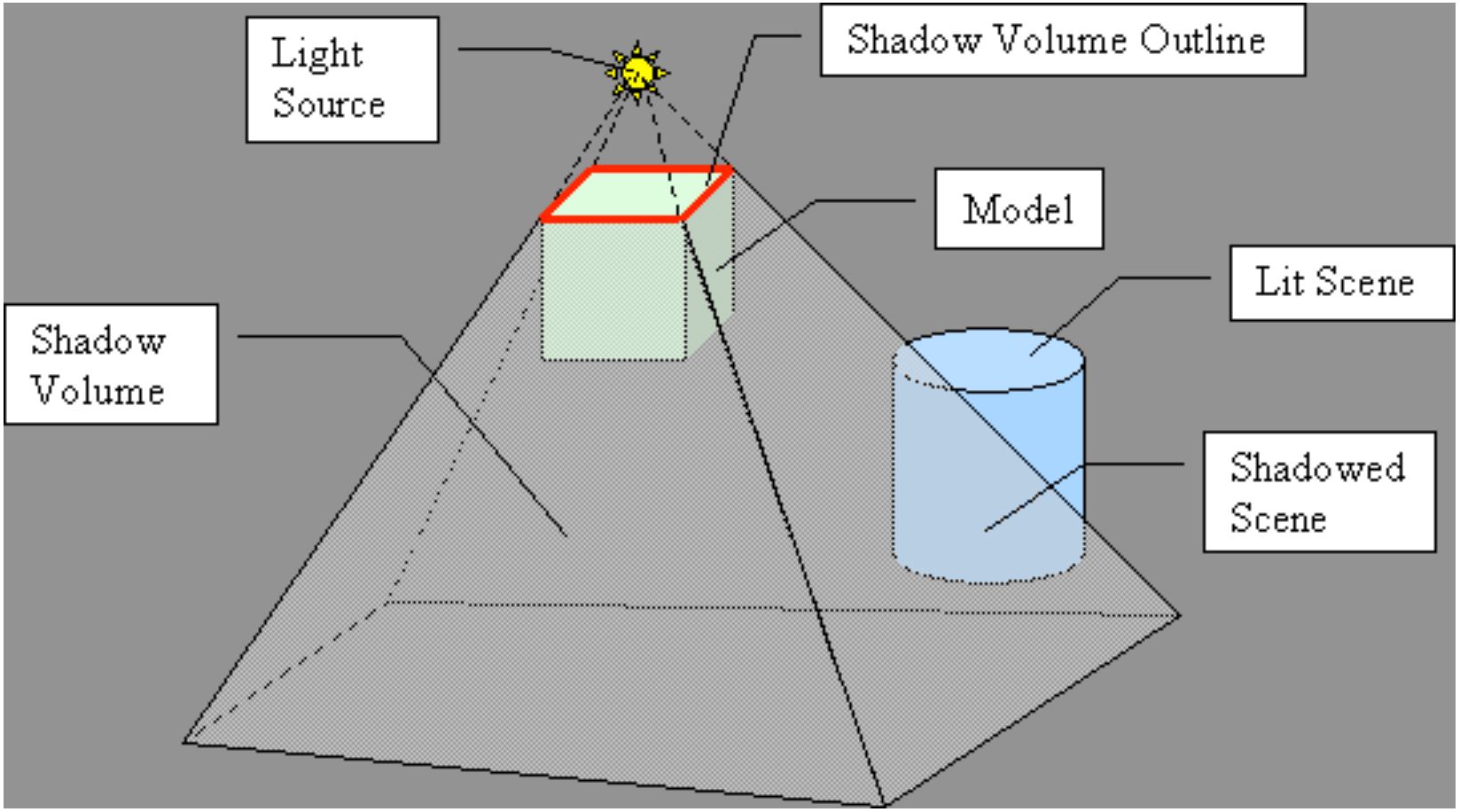
Extruding Silhouette Edges



Using Shadow Volume

- Utilizing the shadow volume during the actual scene rendering is quite straight forward.
- Basically, for each pixel being rendered, trace a ray from the eye toward the pixel and count the shadow volume faces the ray intersects.
- Increment the counter for each front facing shadow volume face, decrement otherwise.
- If the counter is positive at the end, the pixel is in shadow. If it is zero, it is lit.

Shadow Volume Schematic



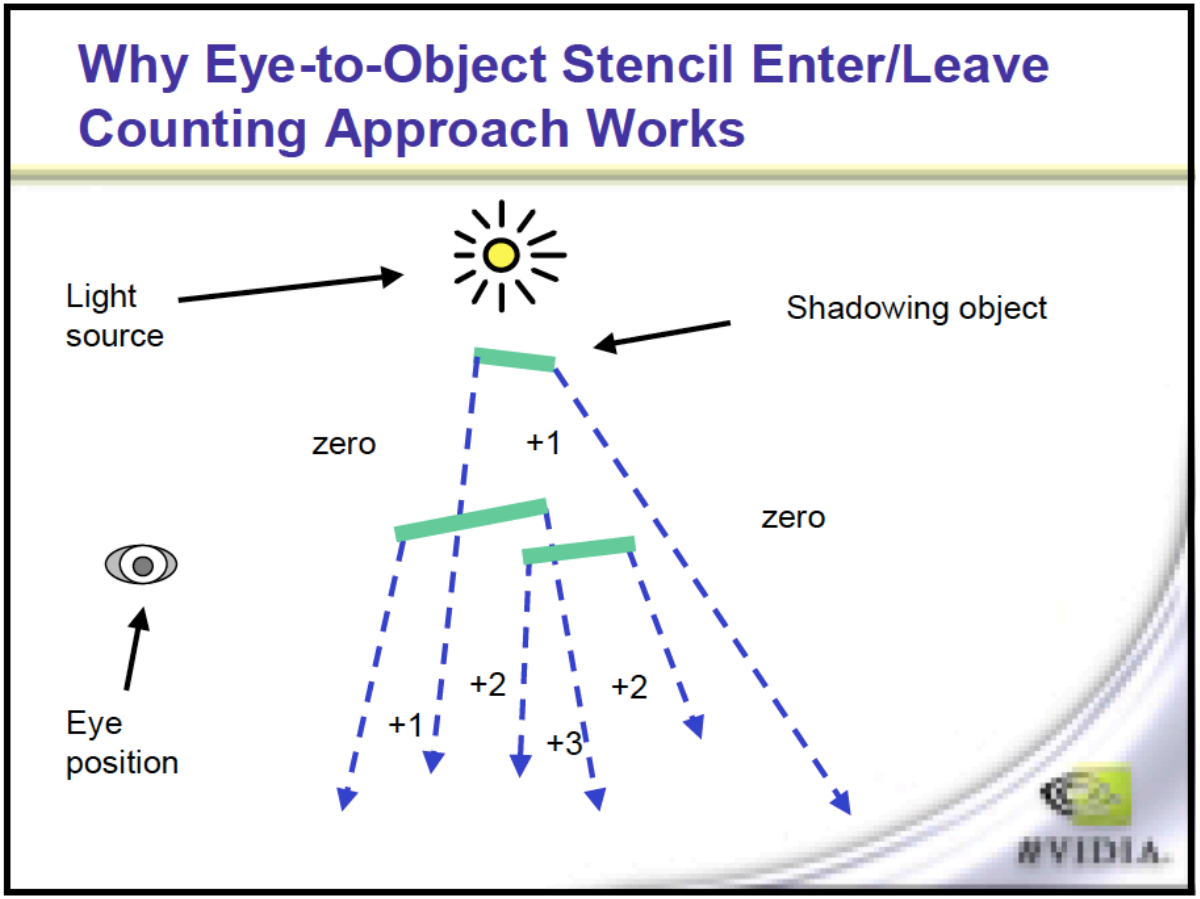
Using Shadow Volume Efficiently

- In OpenGL, we can use the stencil buffer to do the shadow volume faces counting.
- Two variants of the stencil buffer method:
  - z-pass (use the ray from camera toward the pixel)
  - z-fail (use the ray from the pixel toward to the back of the frustum along the view vector)
- In both methods, we start by rendering the scene as usual and then disable the depth buffer write and color buffer write.

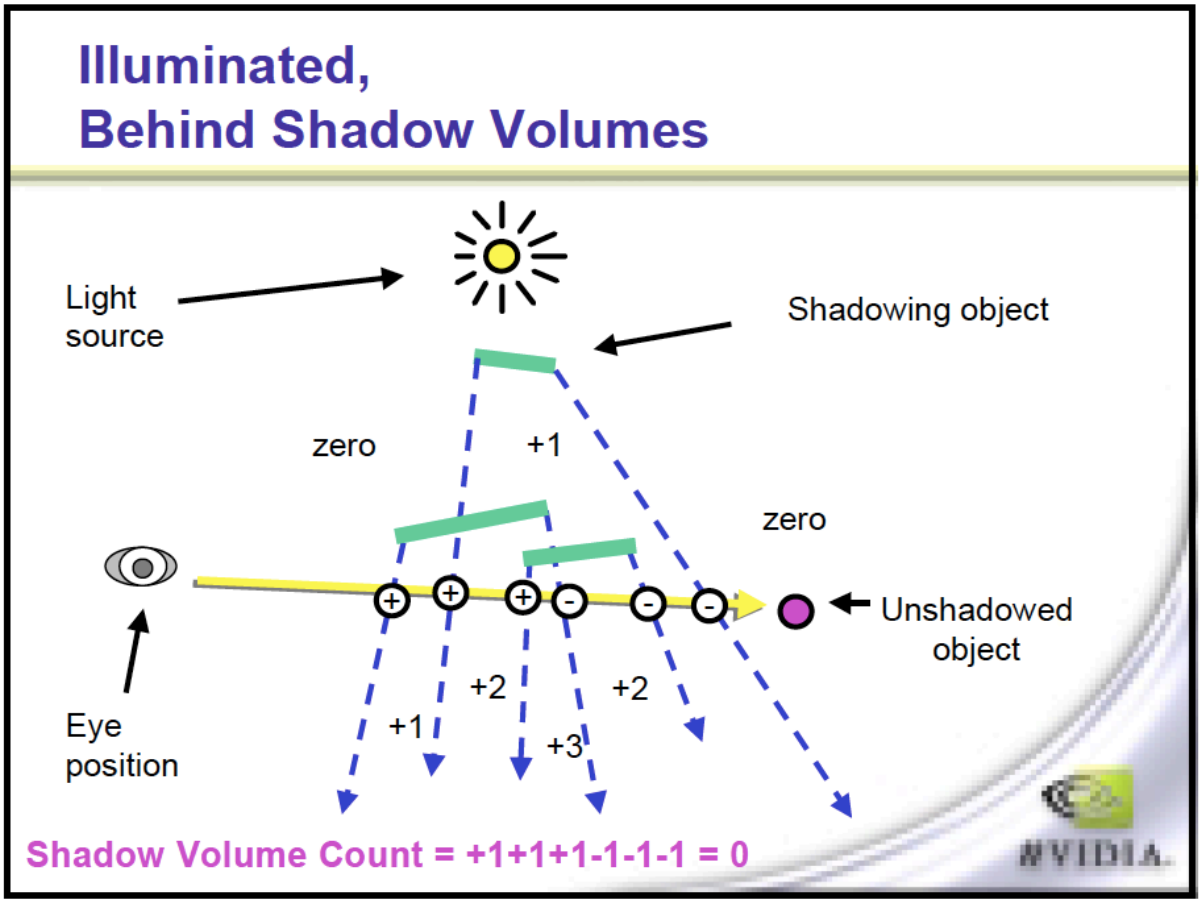
Z – Pass

- Pseudo code for Z-pass method:
  - Clear the stencil buffer with the number of shadow volume the eye is in
  - Set the **stencil operation to increment when depth check passes** and render all **front facing** shadow volume faces
  - Set the **stencil operation to decrement when depth check passes** and render all **back facing** shadow volume faces
  - Disable depth buffer check and enable color buffer write
  - Set stencil function to equal to 0 and stencil operation to nothing, i.e. render the pixel if the value in the stencil buffer is 0.
  - Set blend mode to something that reduces screen intensity
  - Render a screen size translucent quad (cheap shadows)

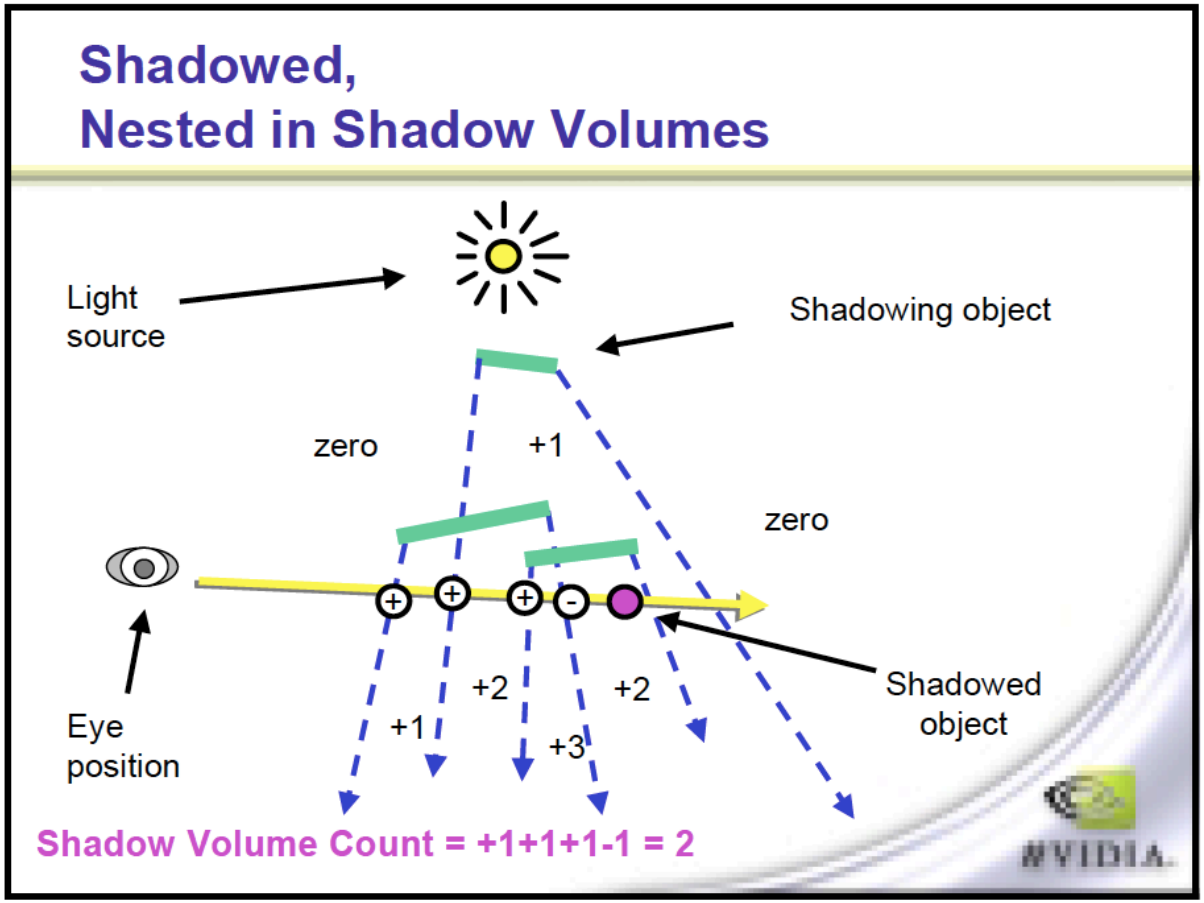
Z-pass Algorithm



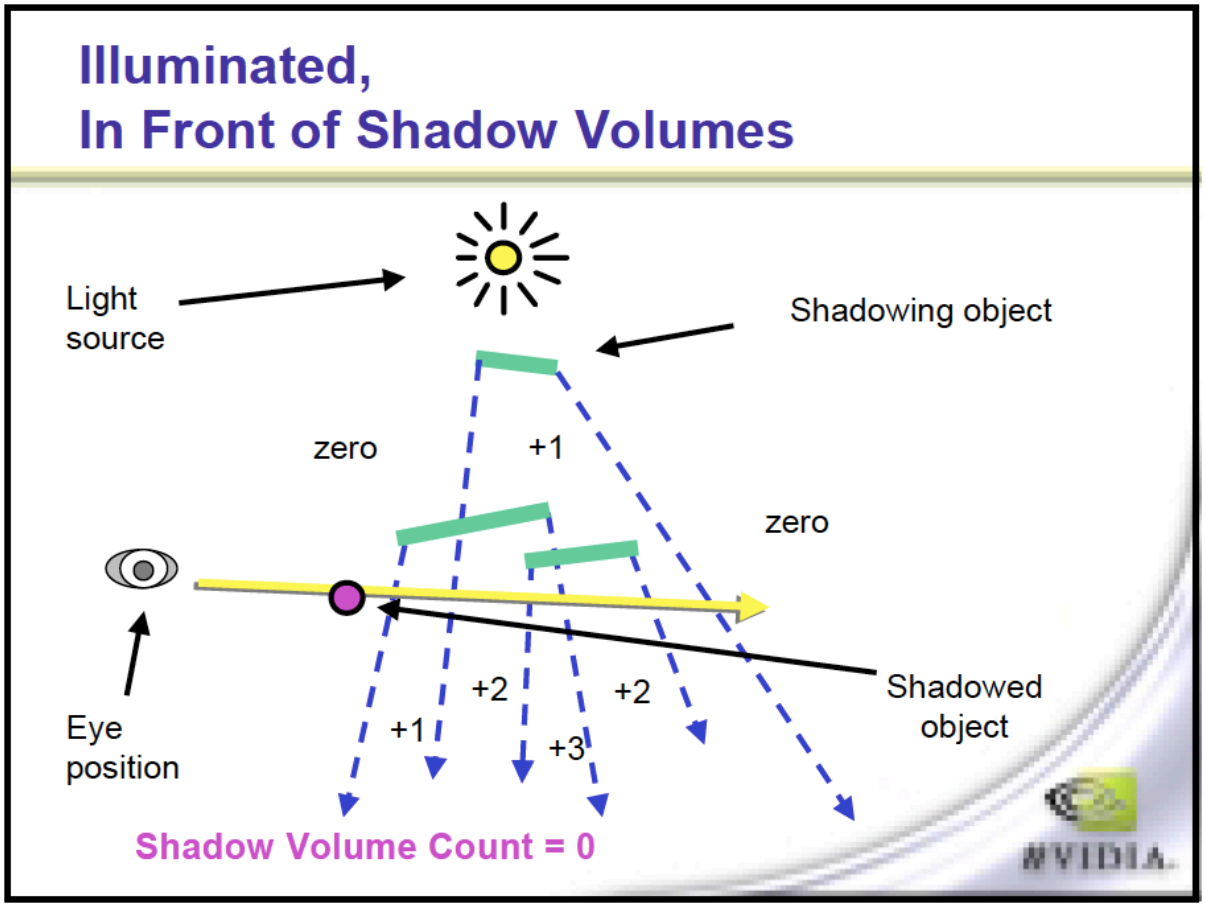
Example 1



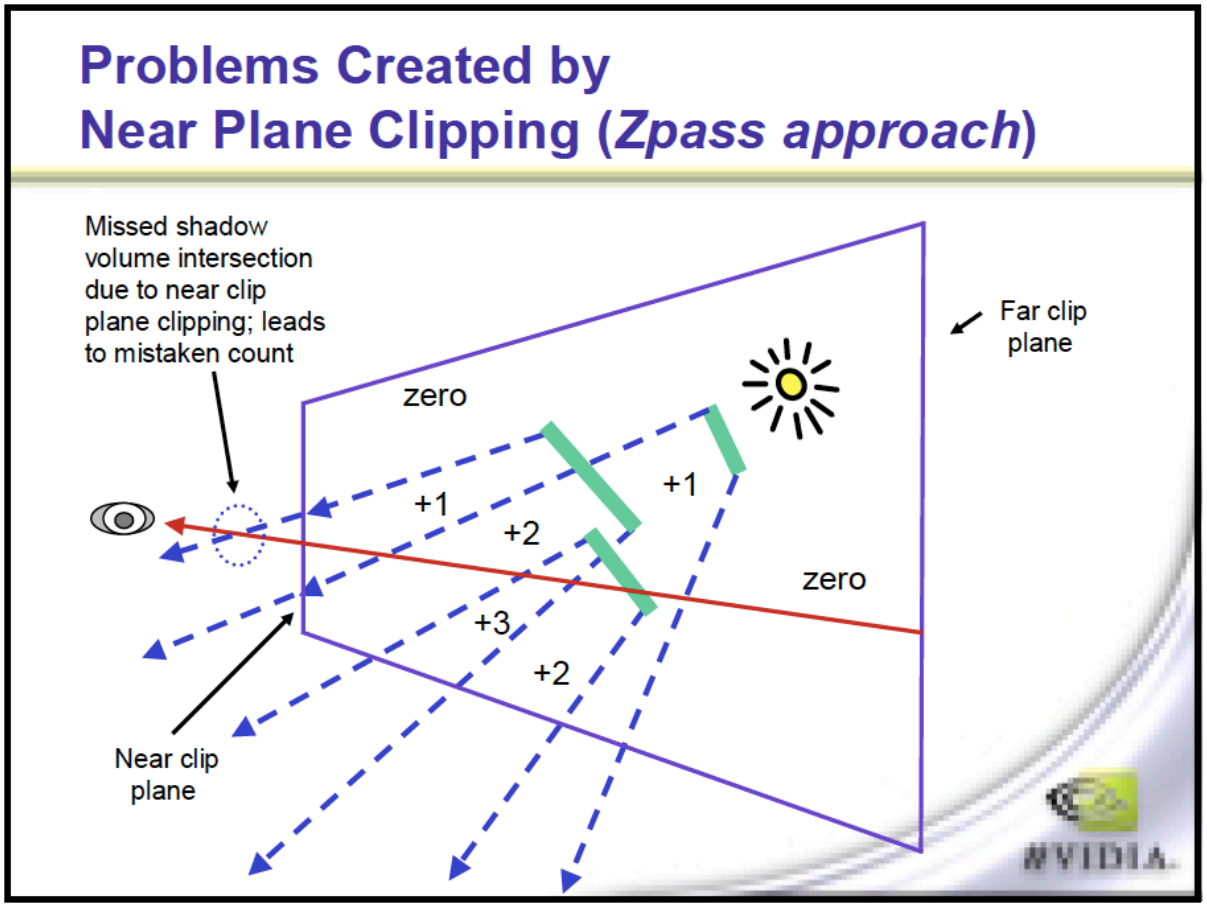
Example 2



Example 3



Houston, we have a problem!



Z – Fail

- Pseudo code for Z-fail method:
  - Clear the stencil buffer to 0
  - Set the **stencil operation to increment when depth check fail** and render all **back facing** shadow volume faces
  - Set the **stencil operation to decrement when depth check fail** and render all **front facing** shadow volume faces
  - Disable depth buffer check and enable color buffer write
  - Set stencil function to NOT equal to 0 and stencil operation to nothing, i.e. render the pixel if the value in the stencil buffer is NOT 0.
  - Set blend mode to something that reduce screen intensity
  - Render a screen size translucent quad

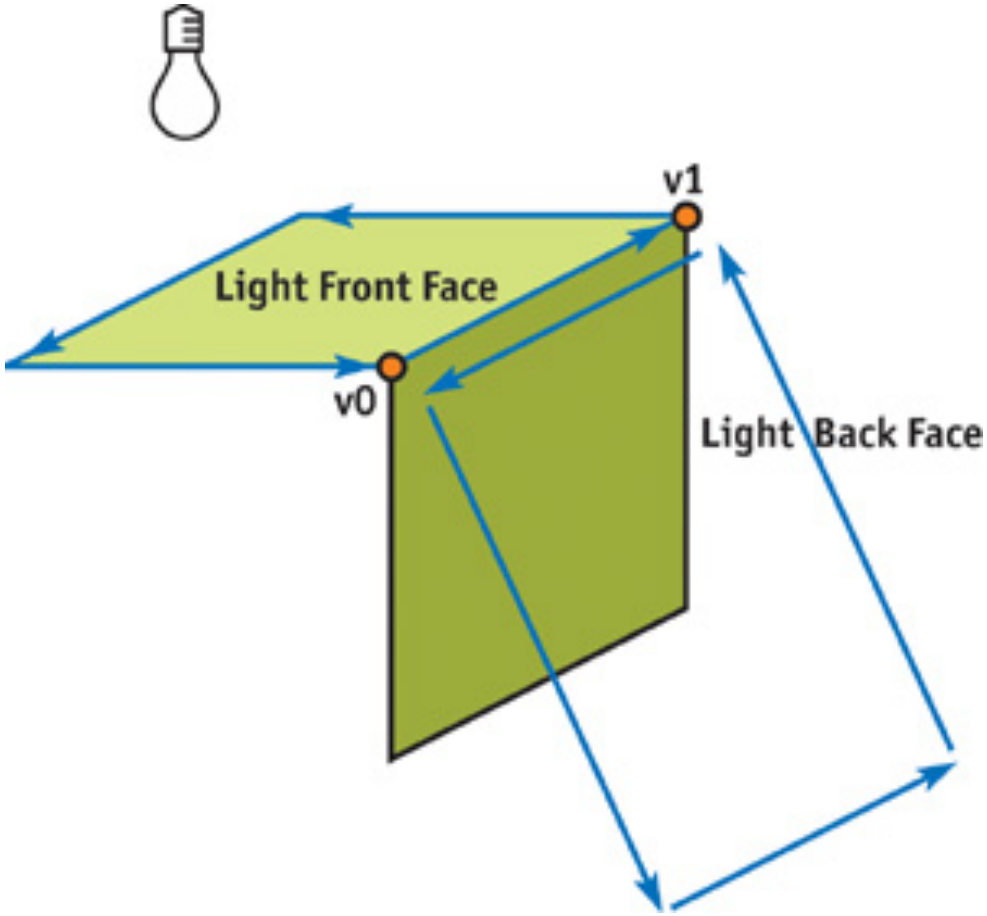
Comparing z-pass and z-fail

- In the z-pass method, the final counter might be wrong if the shadow volume faces get clip by the front clip plane.
- In the z-fail method, the final counter might be wrong if the shadow volume faces get clip by the back clip plane.
- Use OpenGL extension to clamp instead of clip or use shader to do it.
  - glEnable(GL_DEPTH_CLAMP)
  - OFF, by default in OpenGL

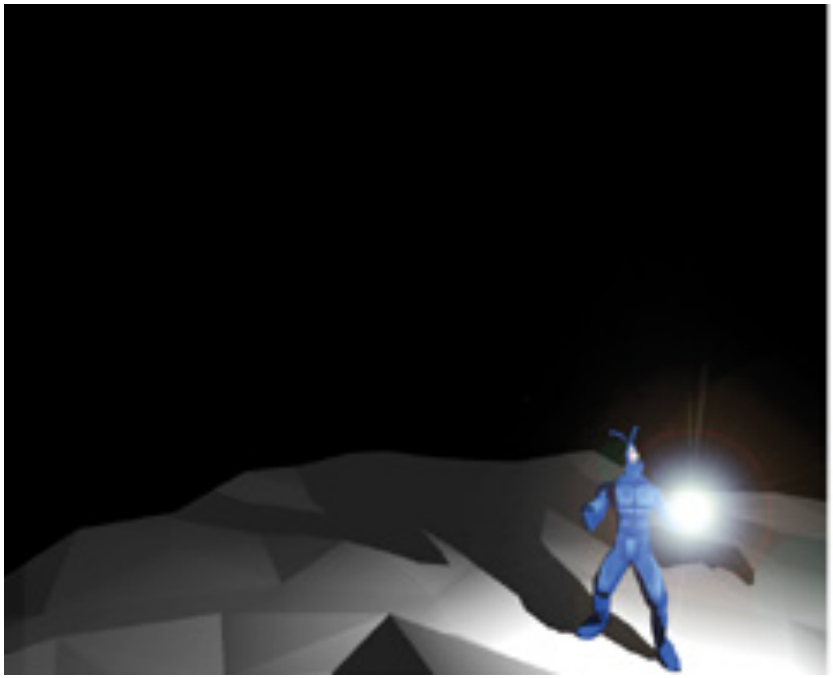
Pro and Con of Shadow Volume

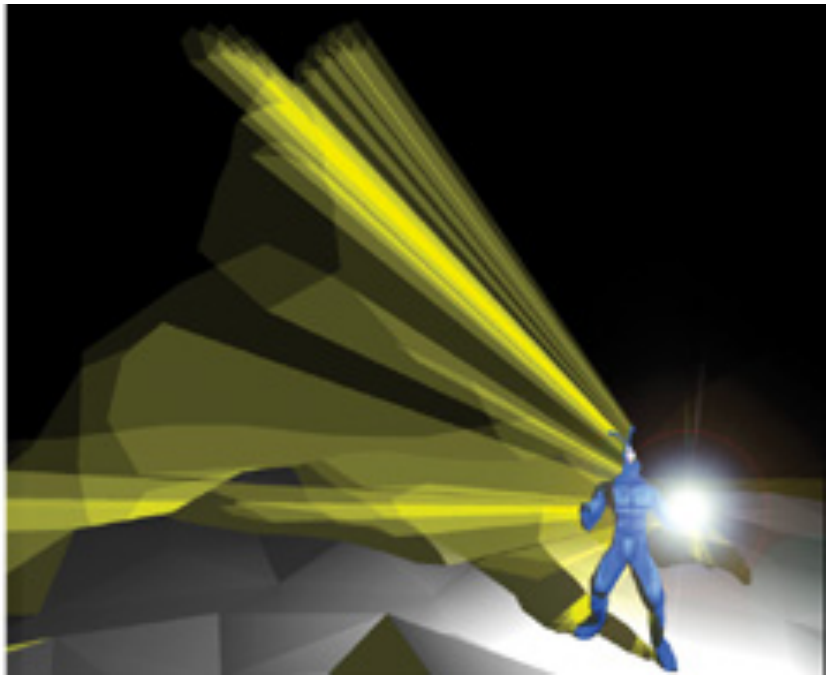
- Pro
  - Precise shadow boundary
  - If the light and object does not move, volume can be pre-calculated
- Con
  - If lights are moving or the blocker objects are moving, the shadow volume have to be calculated each frame. It could get expensive on the CPU.

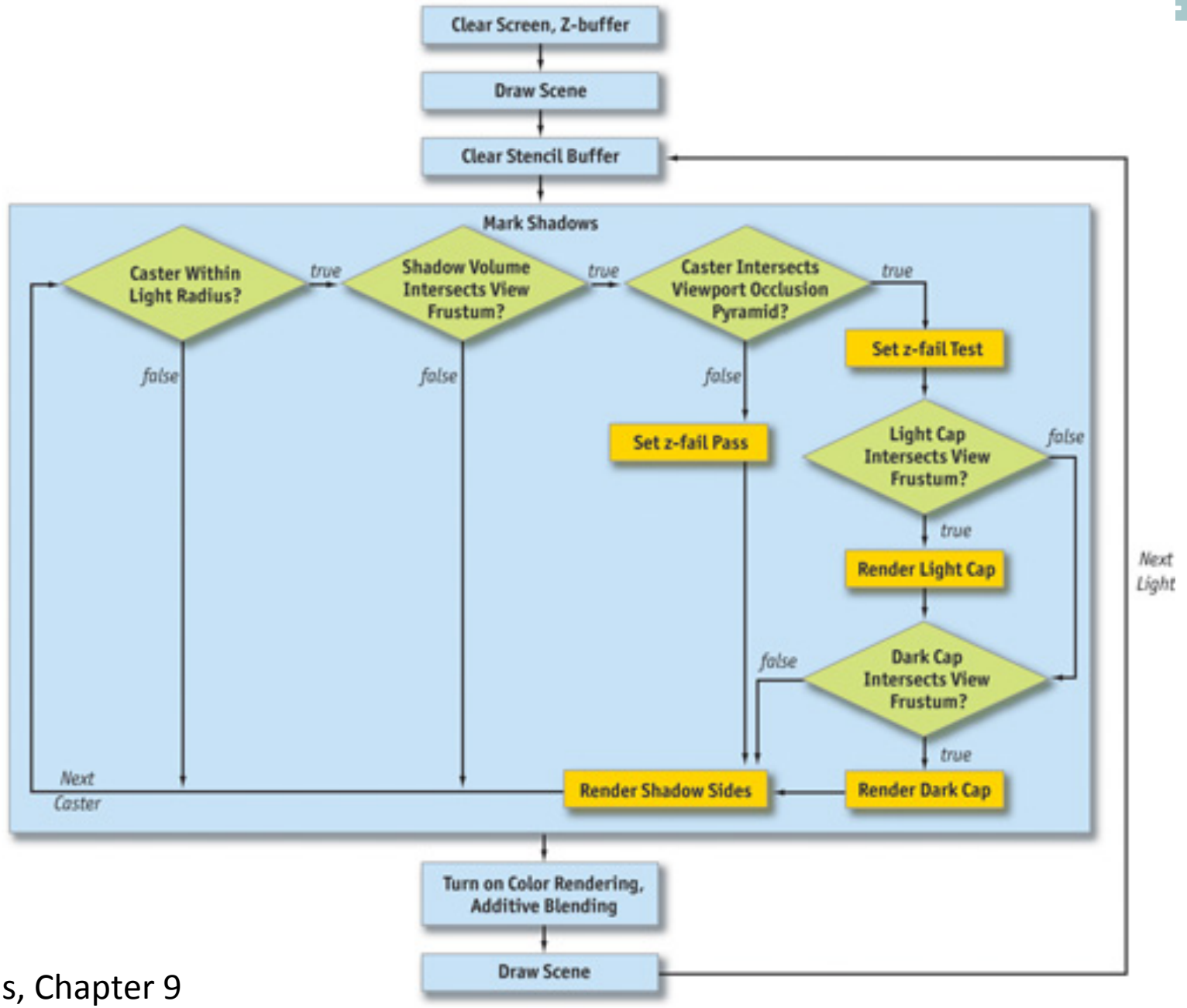
Polygon Winding for Shadow Volumes



Output







GPU Gems, Chapter 9

GPU-based Shadow Volumes

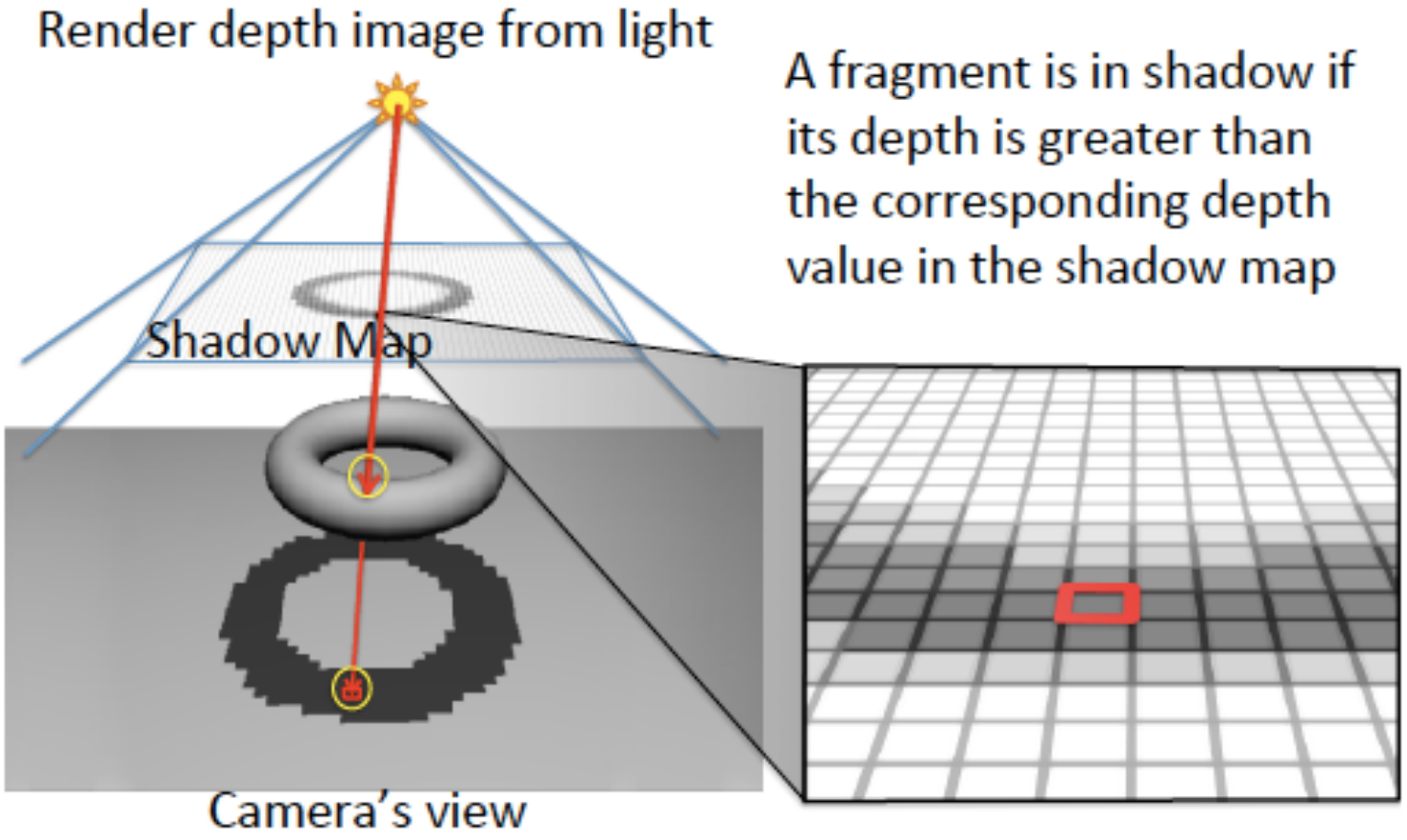
- GPU Gems 3
  - CHAPTER 11. Efficient and Robust Shadow Volumes using Hierarchical Occlusion Culling and Geometry Shaders
  - https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch11.html

Shadow Mapping

Shadow Mapping

- Image-space shadow determination
  - No more 3D polygons and points at infinity!
- Proposed by Lance Williams in 1978
  - Same year as Blinn proposed bump mapping
- Completely image-space based
  - No knowledge of scene geometry required
  - Not dependent on the #vertices in the objects

Basic Idea



But …

- Images ←→ Aliasing artifacts
  - Shadow maps suffer from aliasing artifacts
- Possible solutions
  - Multiple samples per pixel
  - Filtering
- Trivia – Shadow Mapping was used to implement shadows in the first Toy Story movie by Pixar

Shadow Map

- One of the easiest shadow algorithms to implement.
- Like the shadow volume algorithm, it is a two pass process.
- In first pass, render the scene from the point of view of the light. At this point, we only need the depth buffer.
- In second pass, render the scene like usual, however, for each pixel being rendered, if its transformed position in light space fail the depth test with respect to the previously generated depth buffer, the pixel is in shadow.

Generating Shadow Map

- Similar to projective texturing, place a “virtual camera” at the light position and orient it with the light orientation.
- Render the scene’s depth in view space to a texture map.
- Since we need only the depth value, minimize the amount of work in the shader.

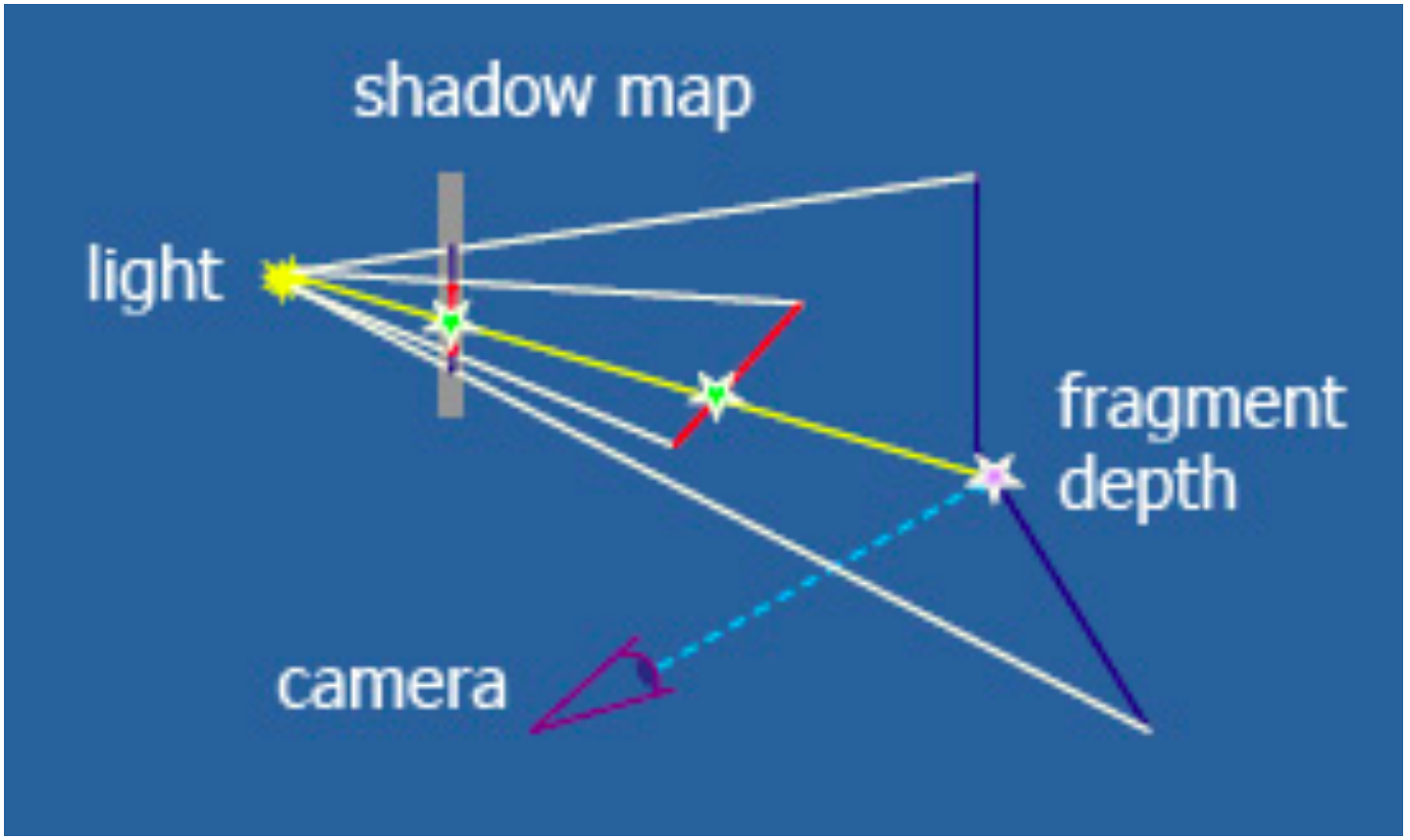
Using Shadow Map

- Set the texture generated in the 1st pass as one of the input texture for the shadow receiver object(s).
- While rendering each pixel of the shadow receiver object(s):
  - Transform the pixel position into the light space
  - Convert the position into texture coordinate
  - Access the depth value from the depth texture map
  - If the pixel depth value is greater than the depth value from the depth texture, it means there is an object that is closer to than the current pixel, hence the pixel is in shadow.

Using Shadow Map *(cont'd)*

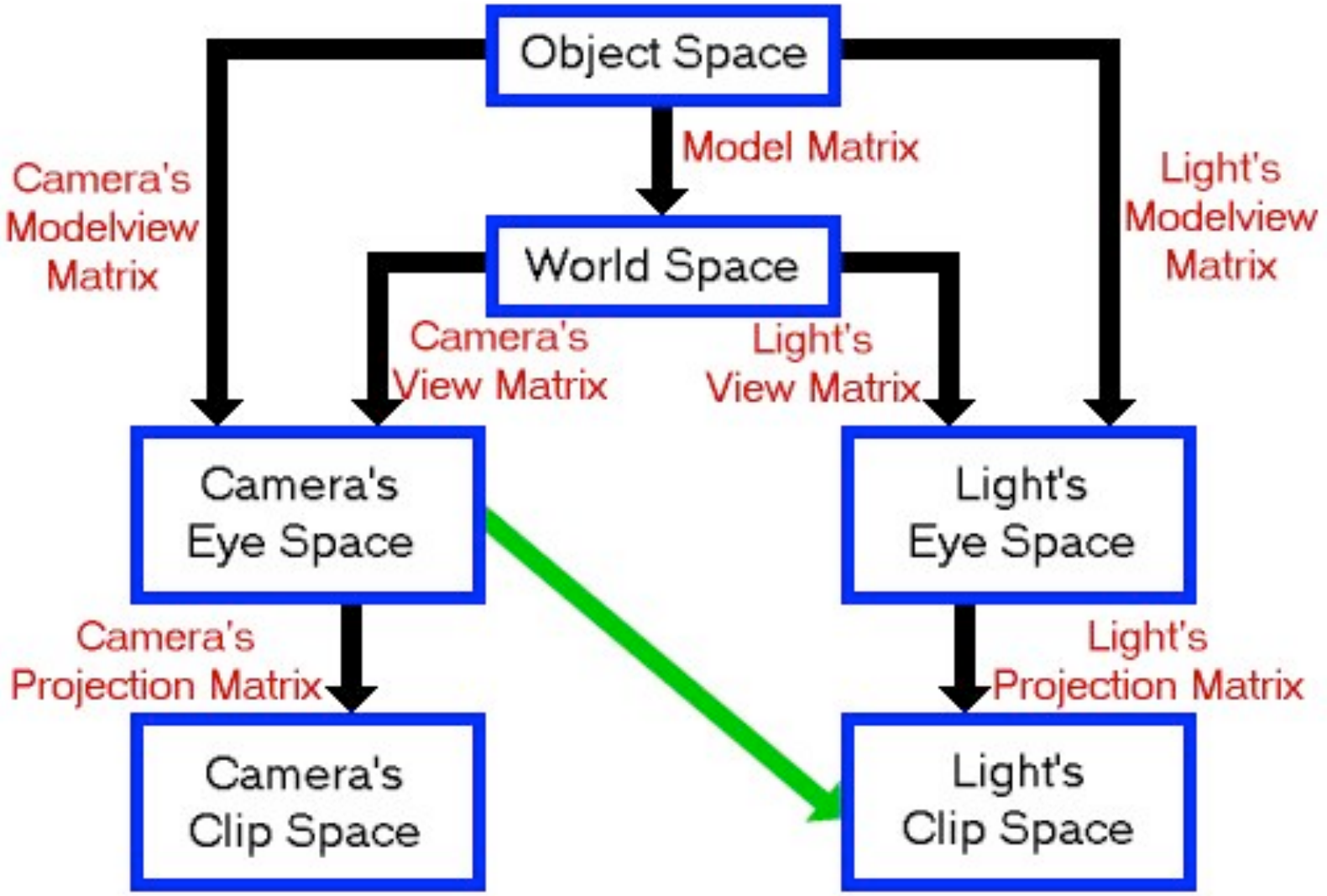
- Once it is determined that a pixel is in shadow, adjust the diffuse and specular component of the pixel color accordingly.
- Typically, multiply the light components by some value between 0 to 1 to make it darker.
- A better approach is to adjust the diffuse and specular component in the lighting equation based on the shadow status.

Shadow Mapping in Action



Converting Pixel to Light Space

- How to convert pixel location to light space?



Converting Pixel to Light Space *(cont'd)*

- Matrices needed to transform the pixel position:
  - The view-to-world transformation matrix W. It is just the inverse of the world-to-view transform.
  - The world-to-light transformation matrix L_V. It is just a viewing matrix with the light position and orientation as the camera position and orientation.
  - The light projection matrix L_P. The projection matrix used to project objects in light space to the texture map. Use the same projection matrix as the one used during shadow map creation.
  - The texture coordinate mapping matrix L_S. It is a matrix used to map values from [-1, 1] to [0, 1].

Converting Pixel to Light Space *(cont'd)*

- Once L_V, L_S, L_P and W are calculated:

$$V_L = L_V * W * V_V$$
$$TC = L_S * L_P * V_L$$

- Where:
  - V_V is a vertex in view space
  - V_L is the transformed vertex in light space.
  - TC is the homogenous texture coordinate

- What is V_L is used for?

Converting Pixel to Light Space *(cont'd)*

- From V_L the depth of the pixel in the light space can be calculated.

Shadow Mapping in OpenGL

Step-by-step approach

OpenGL Steps for Shadow Mapping

Create OpenGL texture object

```
glGenTextures( 1, &shadowMapTexID );
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, shadowMapTexID );
…
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24,
SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT, 0,
GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL );
```

OpenGL Shadow Mapping (2)

Set up an FBO and attach shadow map texture as single depth attachment

```
glGenFrameBuffers(1, &fboID);

glBindBuffer( GL_FRAMEBUFFER, fboID );

glFrameBufferTexture2D(GL_FRAMEBUFFER,
GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, shadowMapTexID,
0 );
```

OpenGL Shadow Mapping (3)

Setup shadow matrix (S) using light modelview matrix (MV_L), projection matrix (P_L) and bias matrix (B)

```
MV_L = Mat_LookAt( lightPosition, targetPosition,
vector3(0,1,0) );

P_L = Mat_Perspective( 60.0, 1.0, 1.0, 25.0 );

B = bias matrix to bring the range from [-1,1] to
[0,1]

S = B * P_L * MV_L;
```

OpenGL Shadow Mapping (4)

Bind the FBO and render the scene from the P-o-V of the light.

```
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glClear(GL_DEPTH_BUFFER_BIT);
glViewport(0,0, SHADOWMAP_WIDTH, SHADOWMAP_HEIGHT);
glCullFace(GL_FRONT); // THIS is important for
correct “biasing” of surfaces
// Render scene
glCullFace(GL_BACK);
```

OpenGL Shadow Mapping (5)

Disable FBO, restore camera viewport, render scene from camera P-o-V

```
glBindBuffer(GL_FRAMEBUFFER, 0);

glViewport(0, 0, WIDTH, HEIGHT);

// Render scene
```

Shadow Mapping – GPU (Vertex Shader)

Multiply world space coordinates of incoming vertex with S, the shadow map matrix, to obtain the shadow texture coordinates.

```
#version 330 core // Important for modern GPU features

uniform mat4 MtoW; // model-to-world matrix
uniform mat4 S;    // shadow matrix
…
out vec4  vShadowCoords;
…
vShadowCoords = S * ( MtoW * vec4(vertex, 1) );
```

Shadow Mapping – GPU (Fragment Shader)

- Look up the depth value from shadow map texture and compare it with the interpolated value from vertex shader

```
smooth in vShadowCoords;

if( vShadowCoords.w > 1 ){
    float shadow = textureProj( shadowMap, vShadowCoords );
    diffuse = mix( diffuse, diffuse*shadow, 0.5 );
}
```

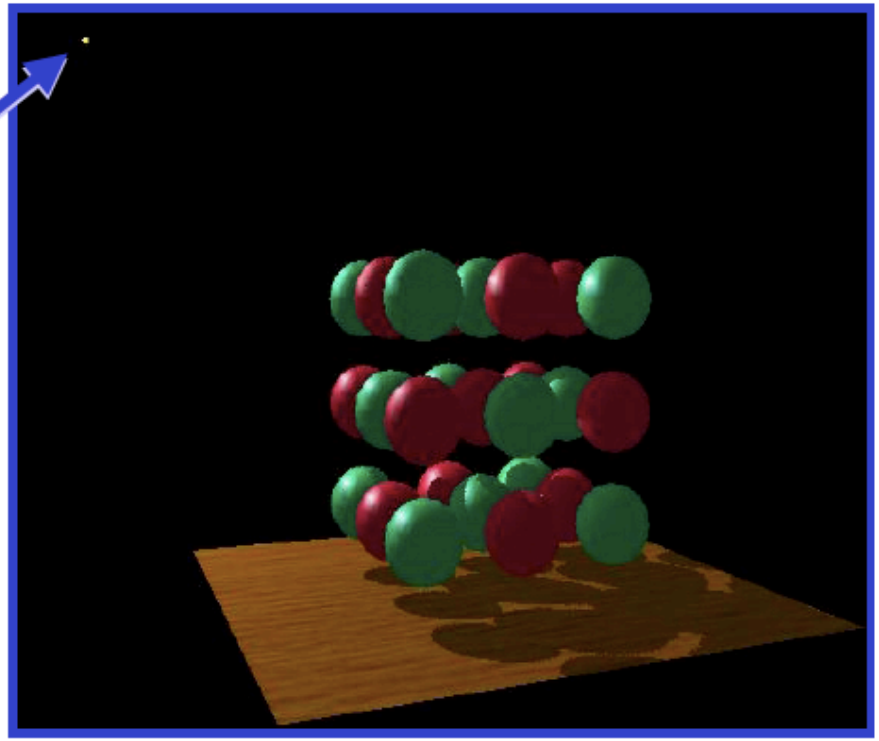
Shadow Map in OpenGL - Note

- When storing the depth value in the 1st pass, offset the depth value back a little bit to prevent the Z-fighting in the 2nd pass.
- In the 2nd pass, if fragment normal is facing away from the light, fragment is in shadow.
- Instead of rendering the front face in the 1st pass, render the back face.

Visualizing the Shadow Mapping Technique (1)

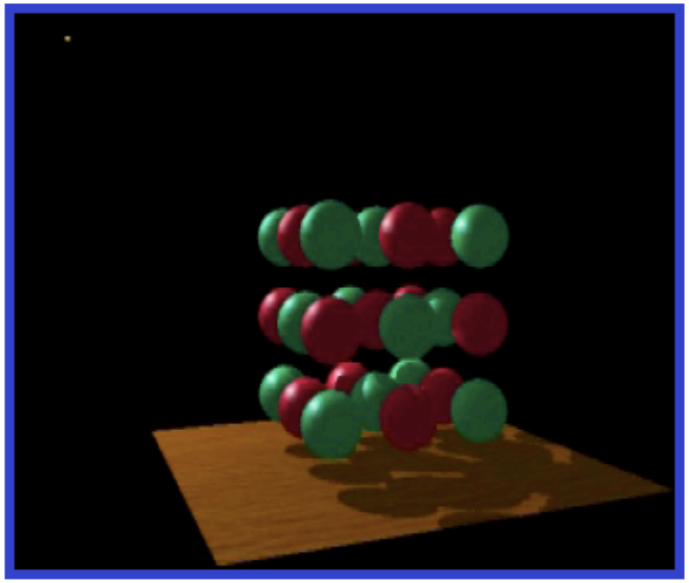
- A fairly complex scene with shadows

the point light source

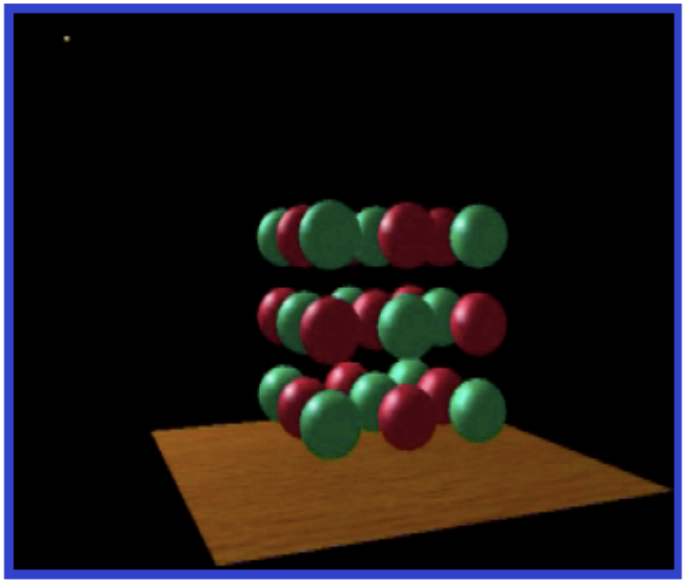


Visualizing the Shadow Mapping Technique (2)

- Compare with and without shadows



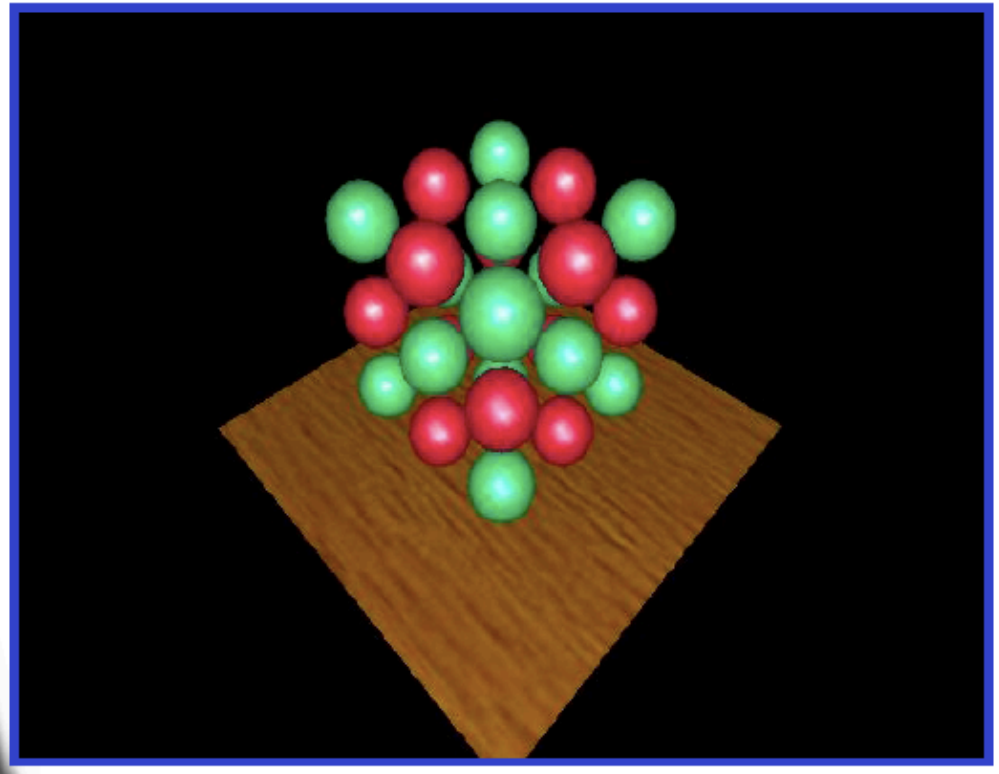
with shadows

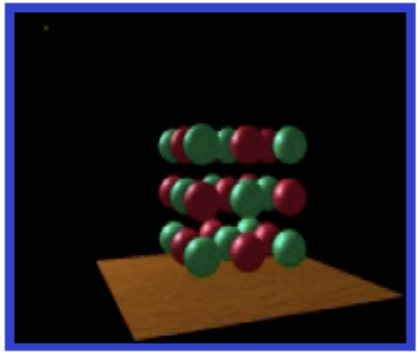


without shadows

Visualizing the Shadow Mapping Technique (3)

- The scene from the light's point-of-view

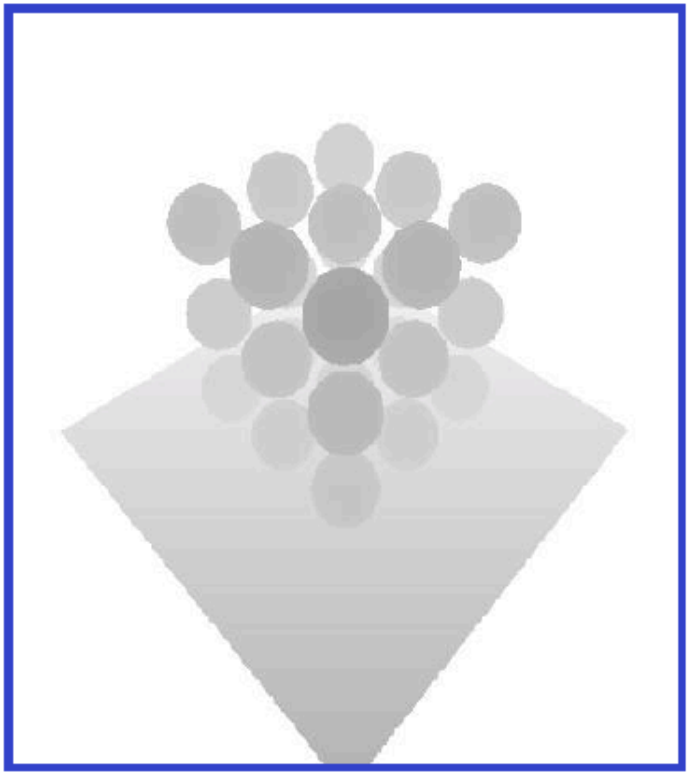


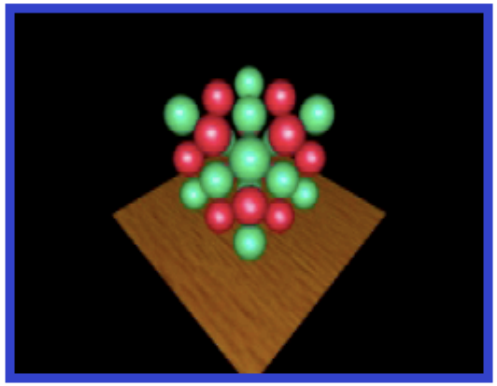


FYI: from the eye's point-of-view again

Visualizing the Shadow Mapping Technique (4)

- The depth buffer from the light's point-of-view

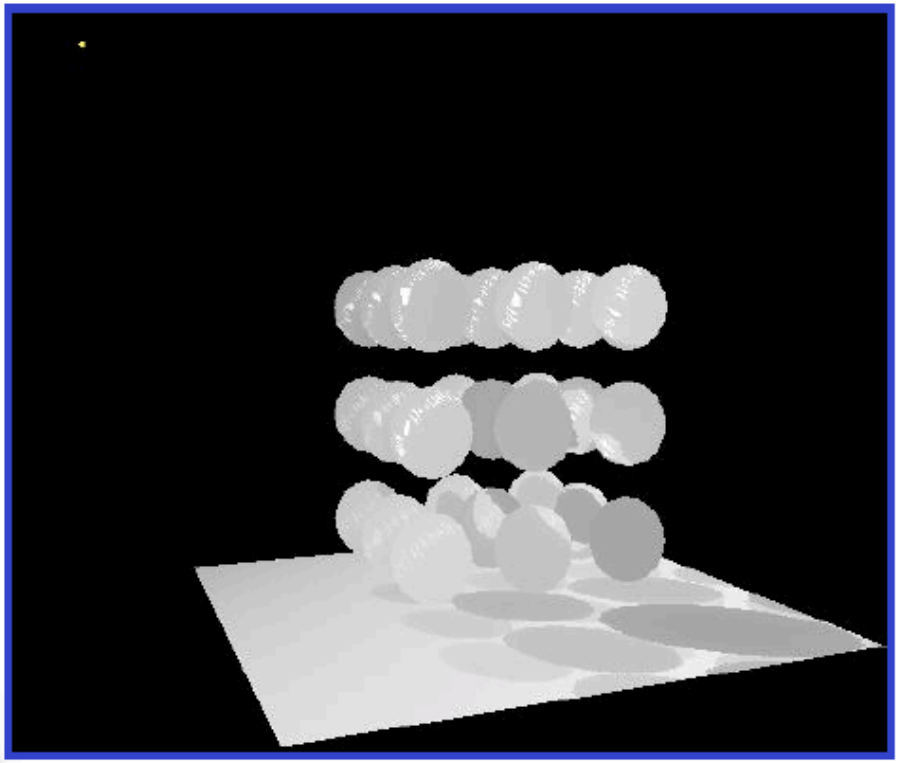


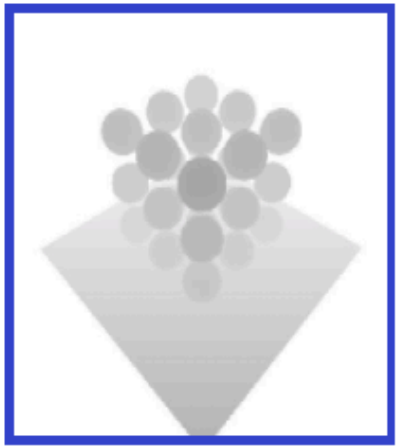


FYI: from the light's point-of-view again

Visualizing the Shadow Mapping Technique (5)

- Projecting the depth map onto the eye's view



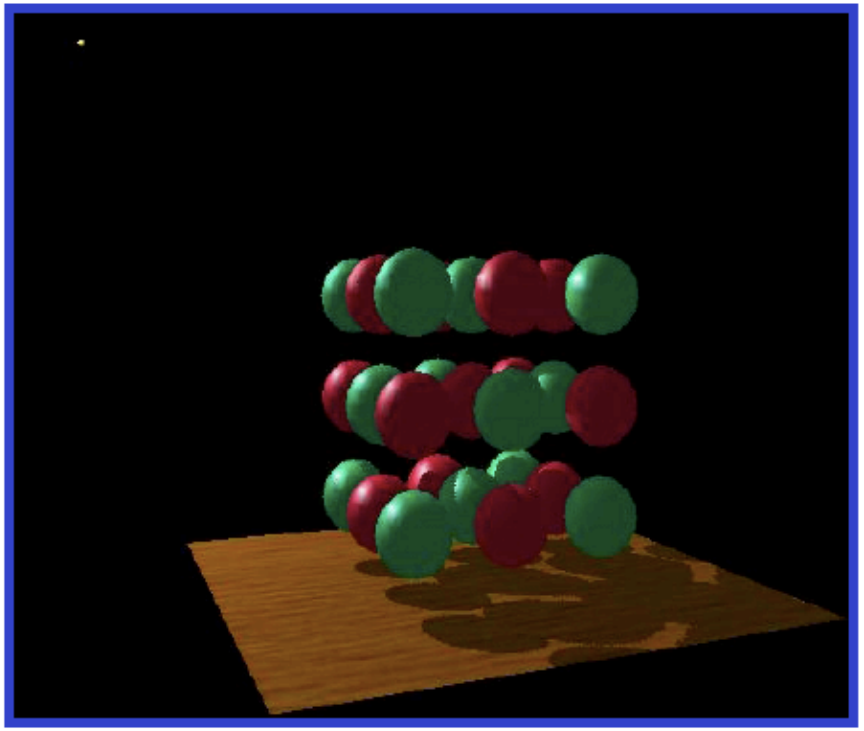


FYI: depth map for light's point-of-view again

Visualizing the Shadow Mapping Technique (7)

- Scene with shadows

Notice how specular highlights never appear in shadows



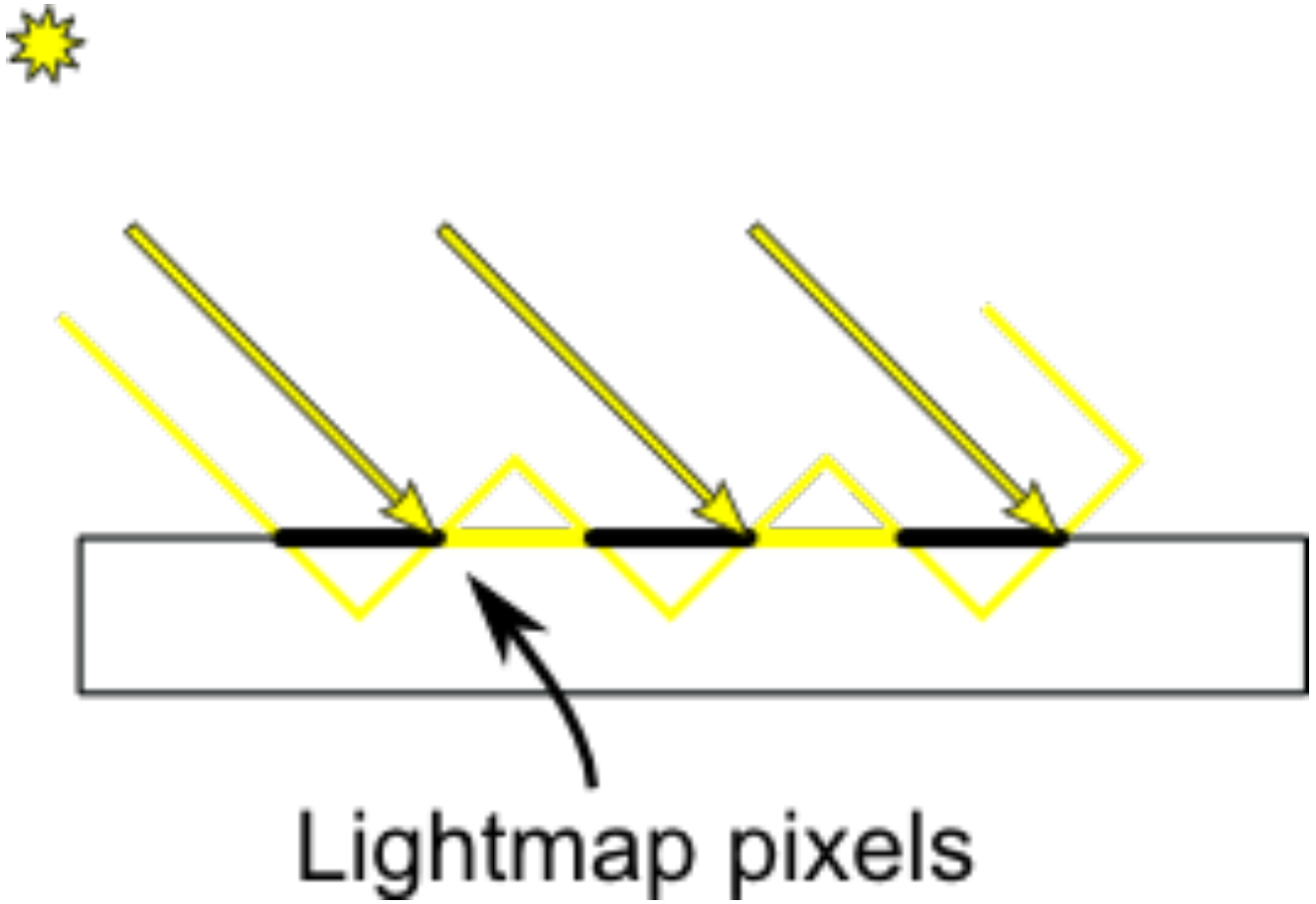
Notice how curved surfaces cast shadows on each other

Pros and Cons of Shadow Map

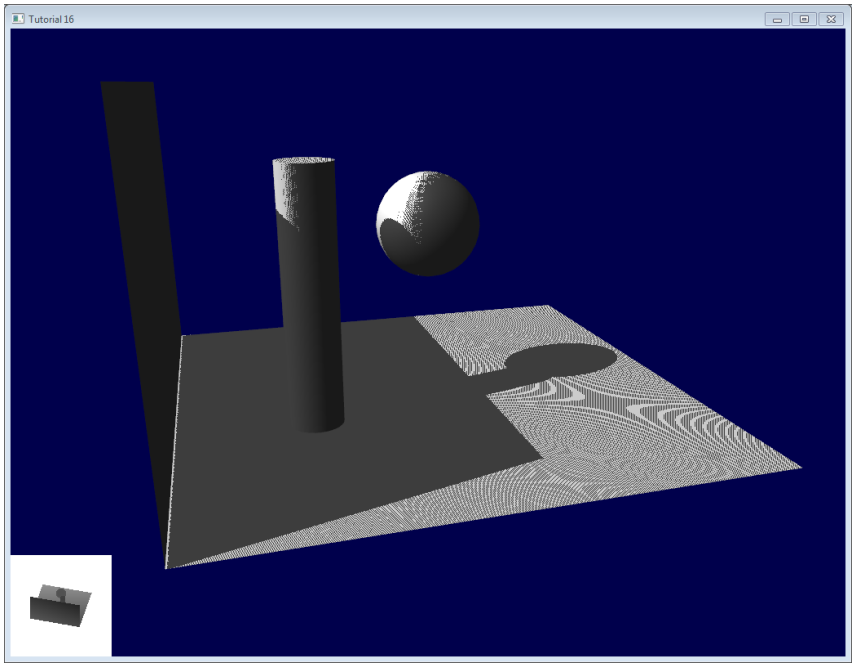
- Pros:
  - Works with arbitrary geometry
  - Simple to implement

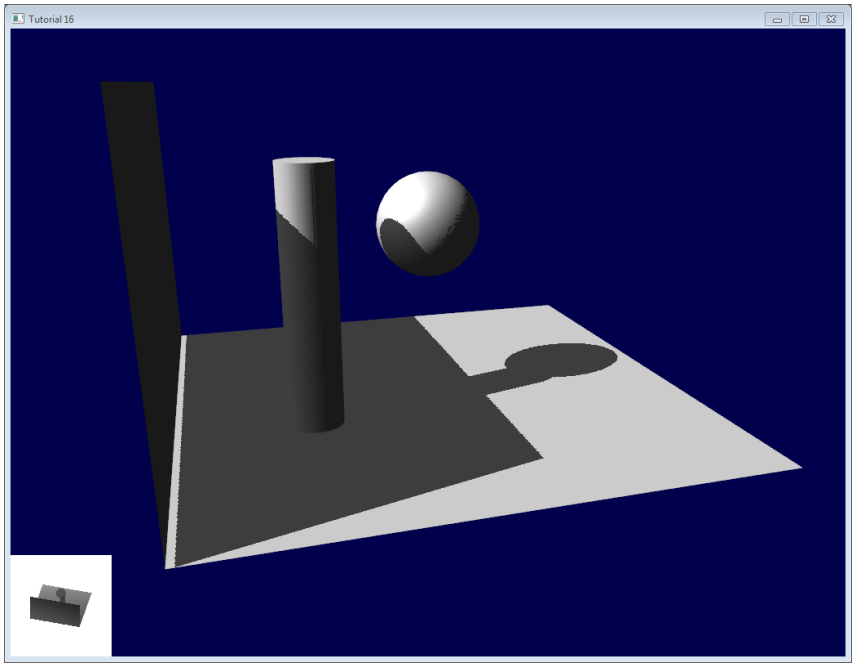
- Cons:
  - Memory requirement. Need one shadow map per light.
  - Aliasing around the edges
  - Z-buffer accuracy could be an issue

Bias and offset

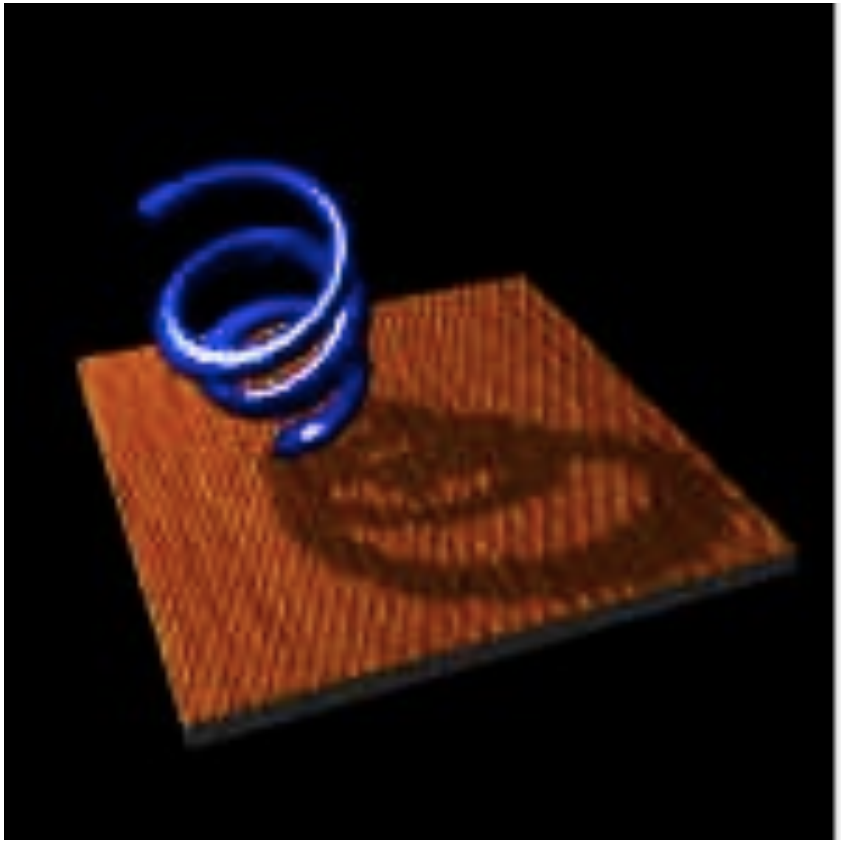


“Shadow Acne”

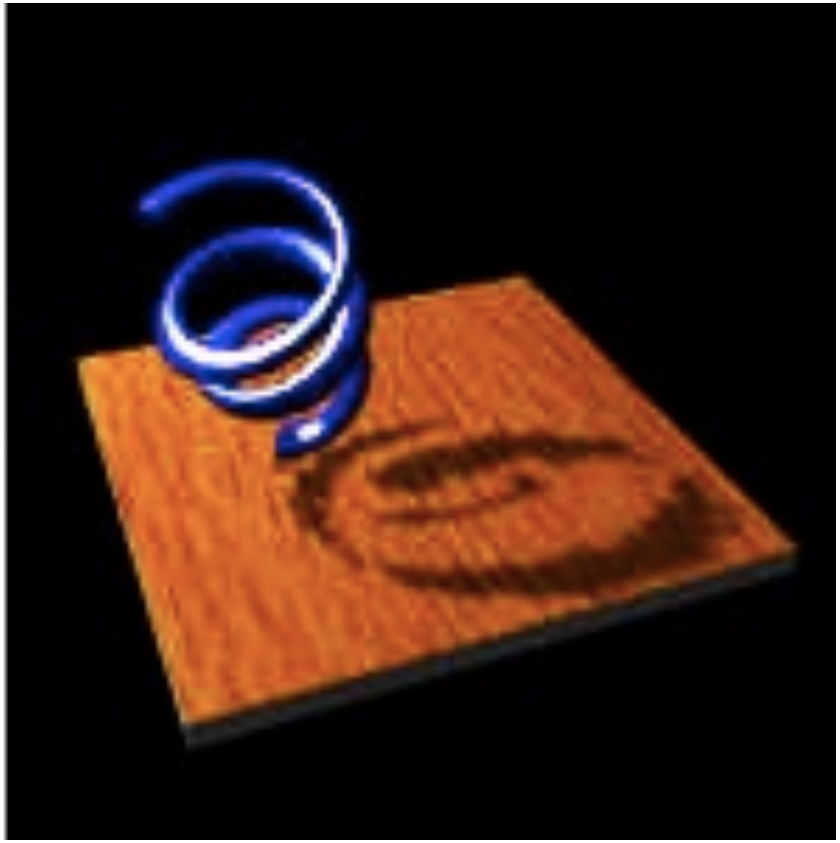




Bias and Offset



Too small – self shadowing



Too large – light leakage

Soft Shadow

- Two different approach:
  - Visually plausible
  - Physically plausible

Multi-Sampling Lights

Multi-Sampling Lights

- The basic idea is to approximate area light with multiple sample points on its surface and rendering the scene multiple times. Once for each sample point on the light.

- The final result is the average of the result from each render pass.

Multi-Sampling Lights *(cont'd)*

- Pseudo-Code:
  - Let N be the number of light sample and P be the list of sample position.
  - For each position in P
    - Setup the light at P
    - Render the scene with a hard shadow algorithm (shadow volume, shadow maps, etc).
    - Accumulate 1/N * the result into the final result buffer
  - Display the final result buffer

Multi-Sampling Lights *(cont'd)*

- To optimize the algorithm, do not do full shading in each pass.

- Instead, accumulate the shadow values and render the scene with full shading at the end taking into account the accumulated shadow values.

Pro and Cons of Multi-Sampling Lights

- Pros:
  - Physically plausible and gives accurate result given enough sample
  - Can be used to approximate any light
- Cons:
  - Slow
  - Not practical for real-time use

Multi-Sampling Lights *(cont'd)*

Multi-Sampling Lights *(cont'd)*

Multi-Sampling Lights *(cont'd)*

Percentage Closer Filtering

Percentage Closer Filtering

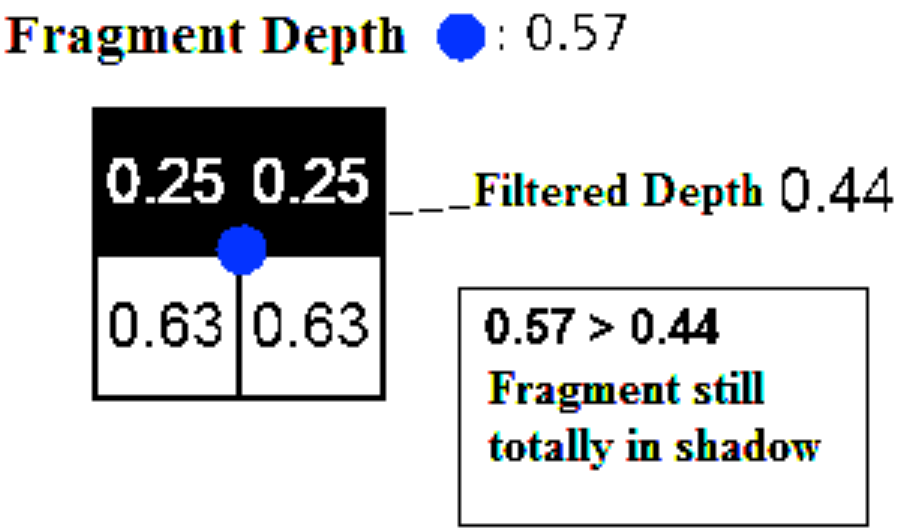
- Fast and simple filtering technique.
- An enhancement to the basic shadow map algorithm.
- Shadow map algorithm generates hard shadows. Why?

Percentage Closer Filtering *(cont'd)*

- The idea is to measure the percentage of the non-shadowed area being closer to the light source. How?

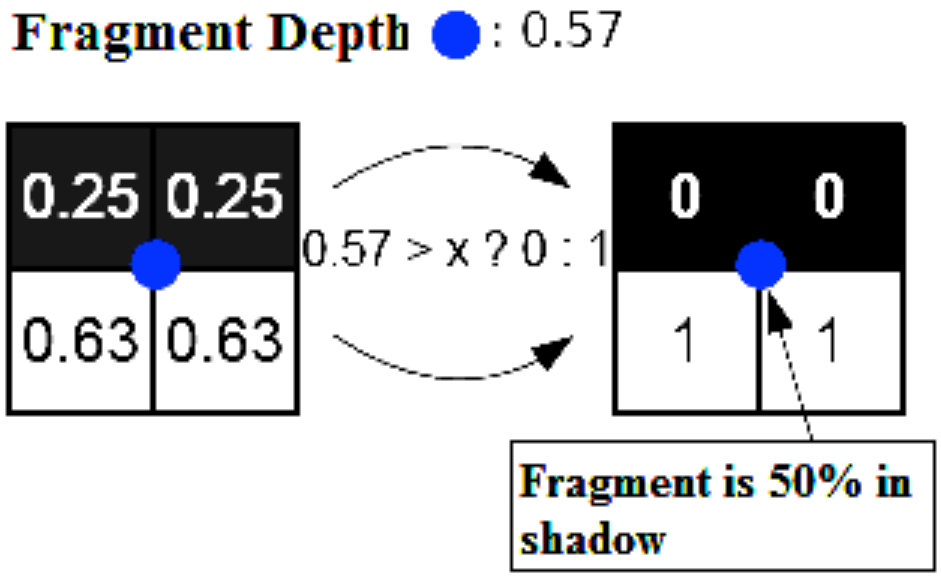
Percentage Closer Filtering *(cont'd)*

- Bilinear filtering does not work with depth map.



Percentage Closer Filtering *(cont'd)*

- Filter the light visibility state
- The filter area is typically a rectangle, but it could be any 2D shape.



Percentage Closer Filtering *(cont'd)*

- Bigger filter area will result in softer shadow
- Higher sample density will result in better accuracy.
- Can adjust the filter size to simulate light size

Percentage Closer Filtering in OpenGL

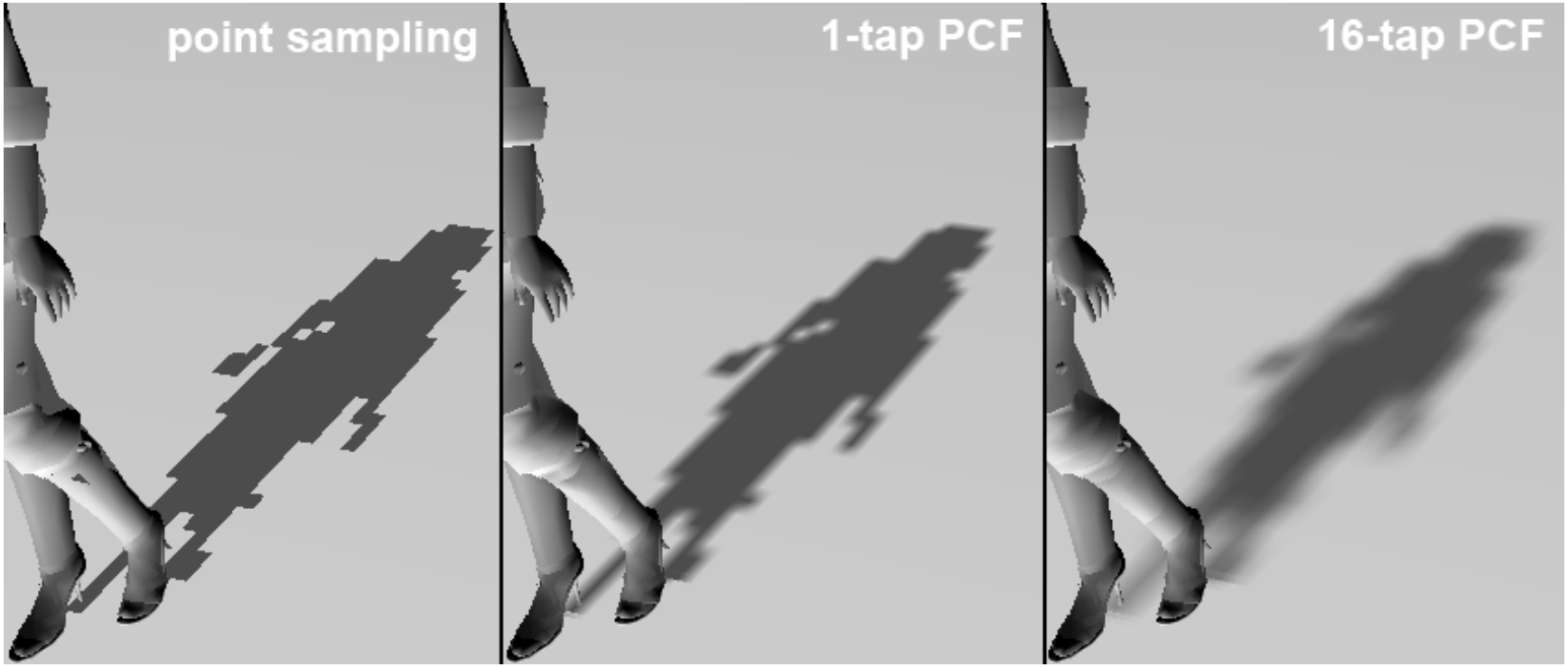
- To implement percentage closer filtering, we need to do a slight modification to the 2nd pass of the shadow map algorithm.
- Basically, instead of sampling the shadow map only once, sample it multiple times within the filtering area.
  - use “textureProjOffset(..)” GLSL call
- Use the average of the depth test result to control the shadow darkness.

Pros and Cons of Percentage Closer Filtering

- Pros:
  - Easy to integrate into the shadow map algorithm
  - Reduce aliasing problem from the basic shadow map algorithm
  - Quite fast and gives decent result

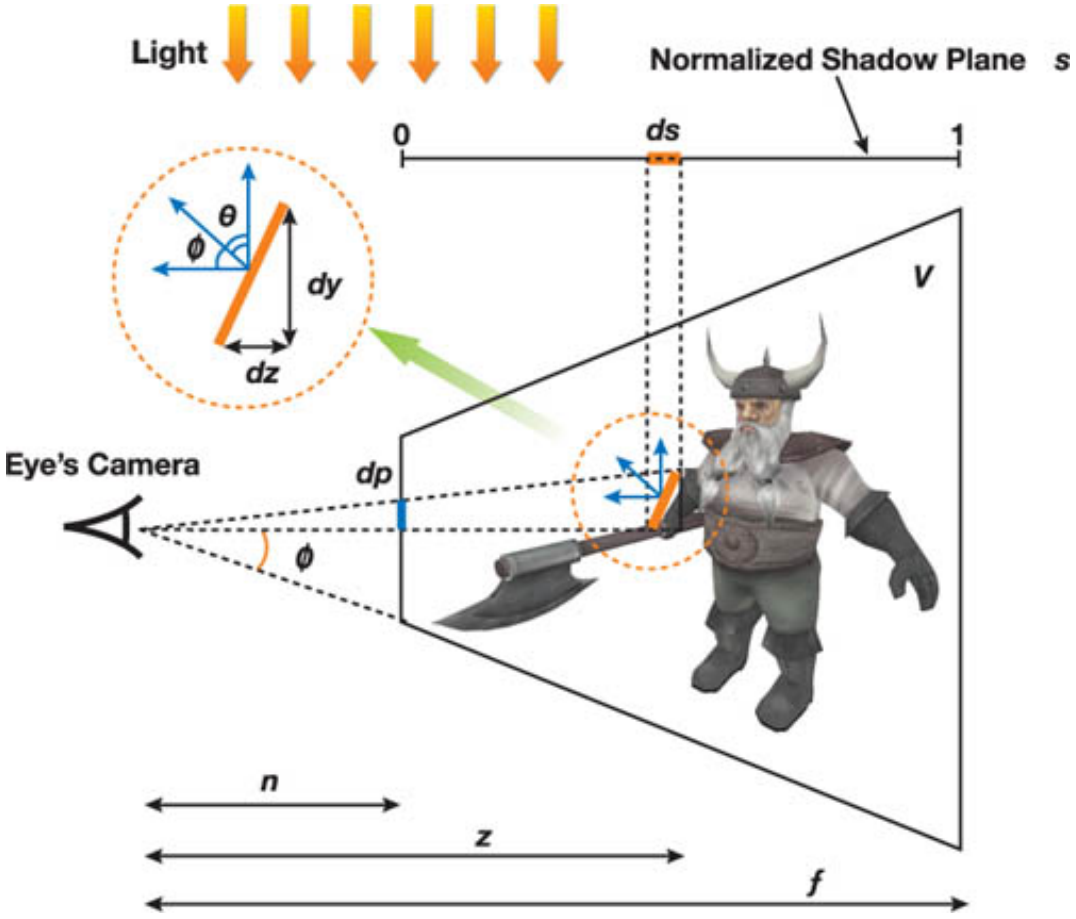
- Cons:
  - Will soften the shadow the same amount regardless light or blocker object distance
  - Big filter with high sample density could adversely affect performance due to the number of texture sampling.

Example

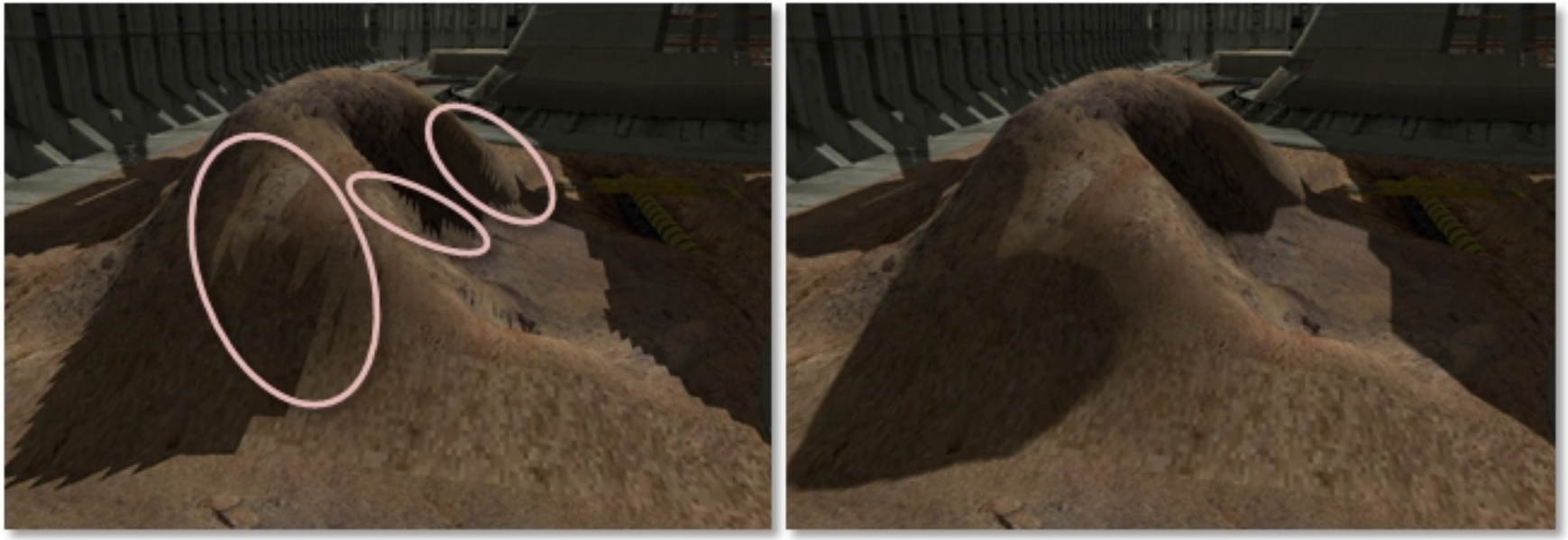


Early Work - Pixar

Shadow Maps Aliasing

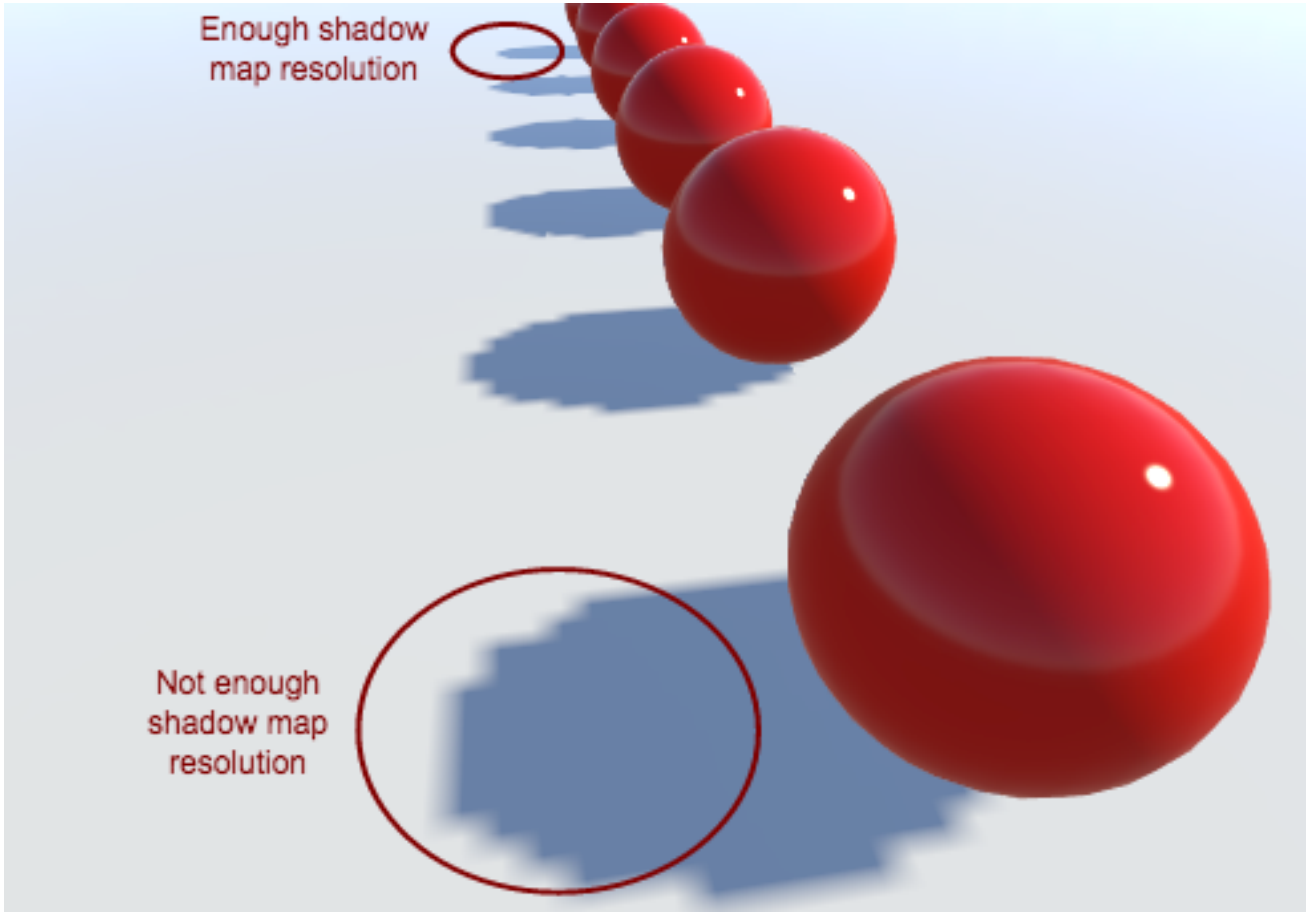


Projective Aliasing

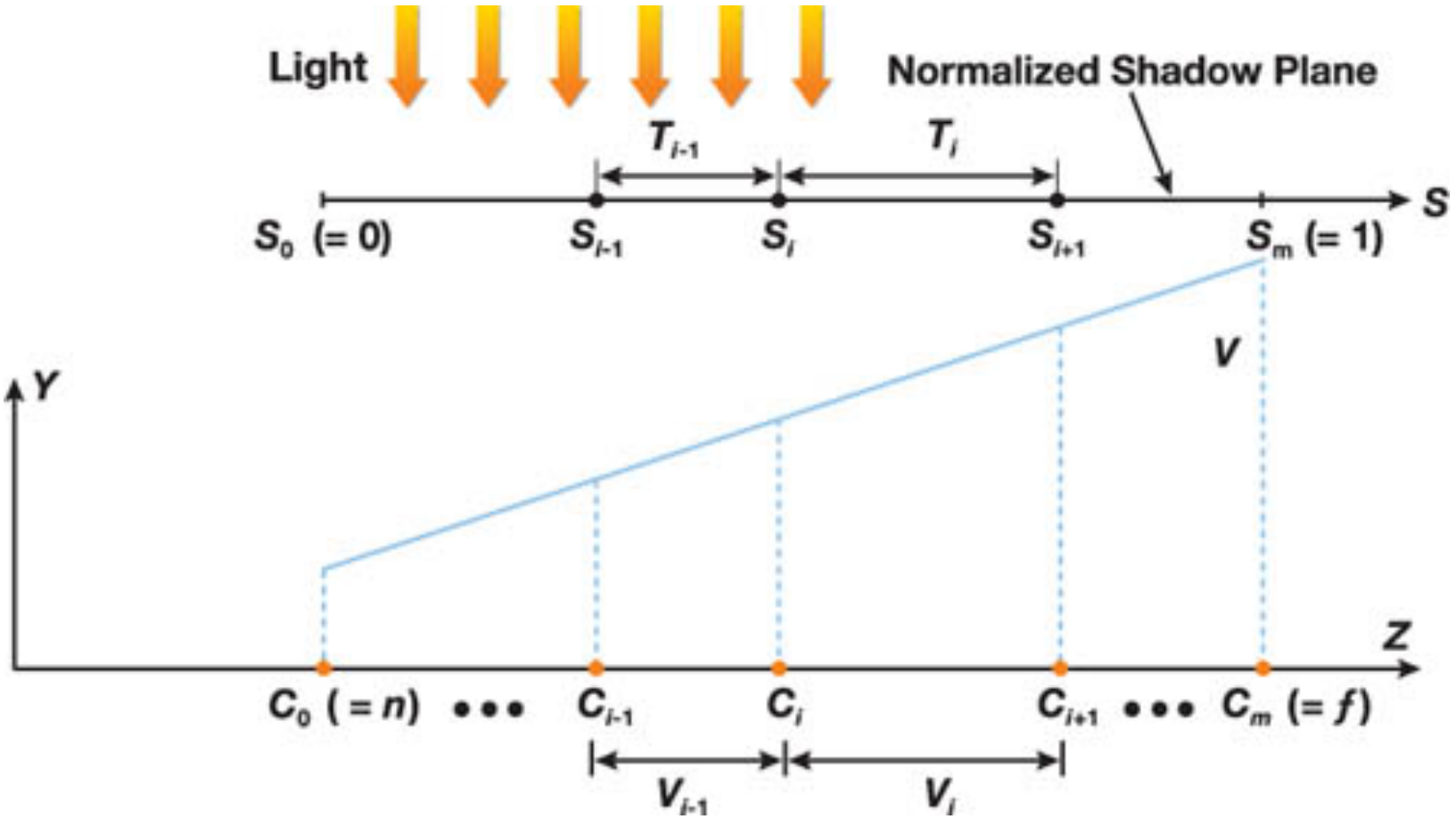


https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx

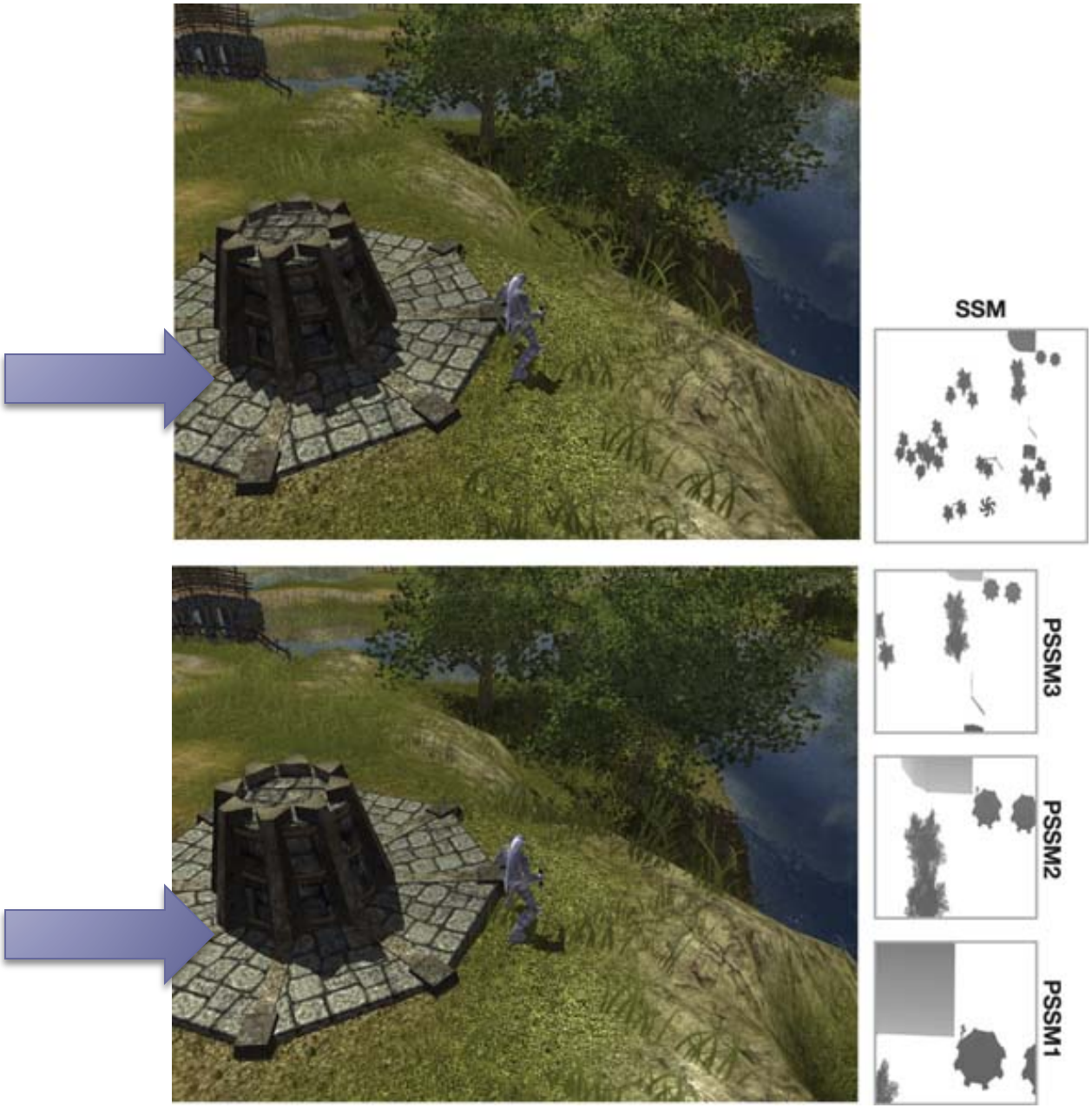
Aliasing Example (Unity)



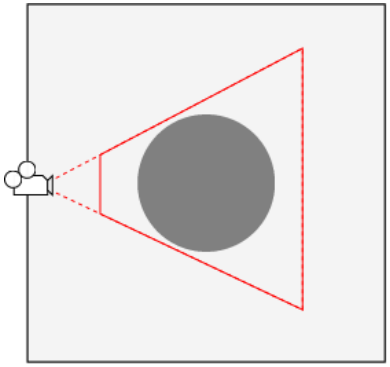
Parallel Split Shadow Maps (PSSM)

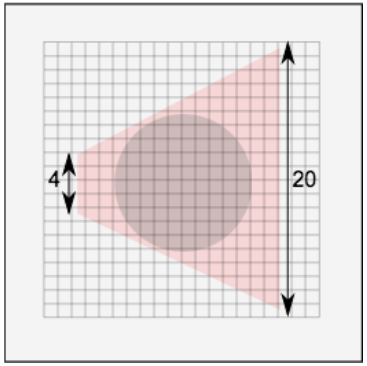


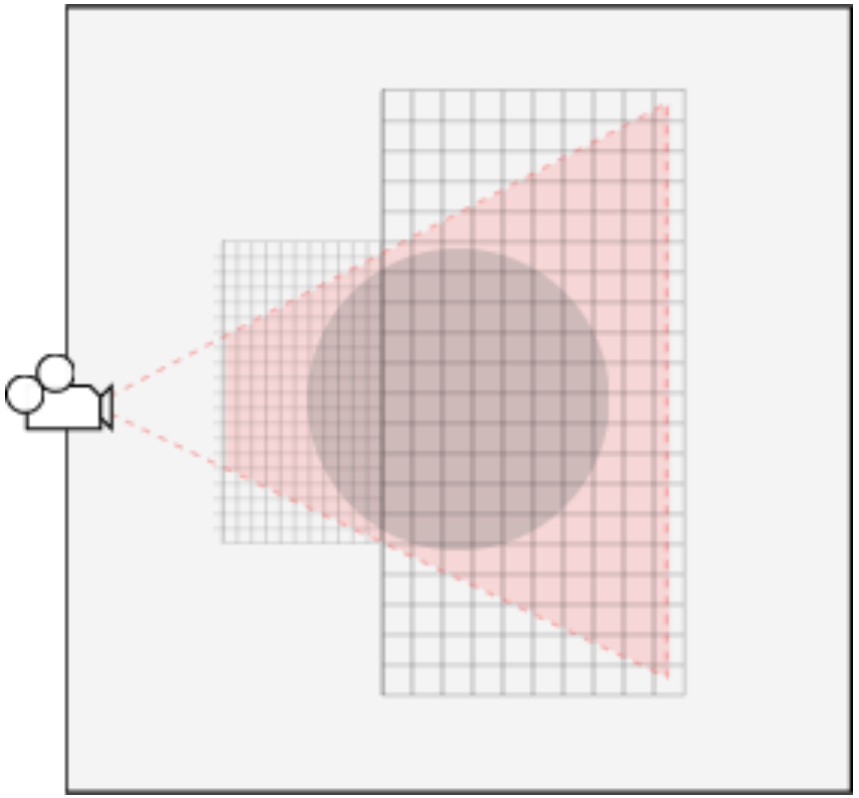
Results



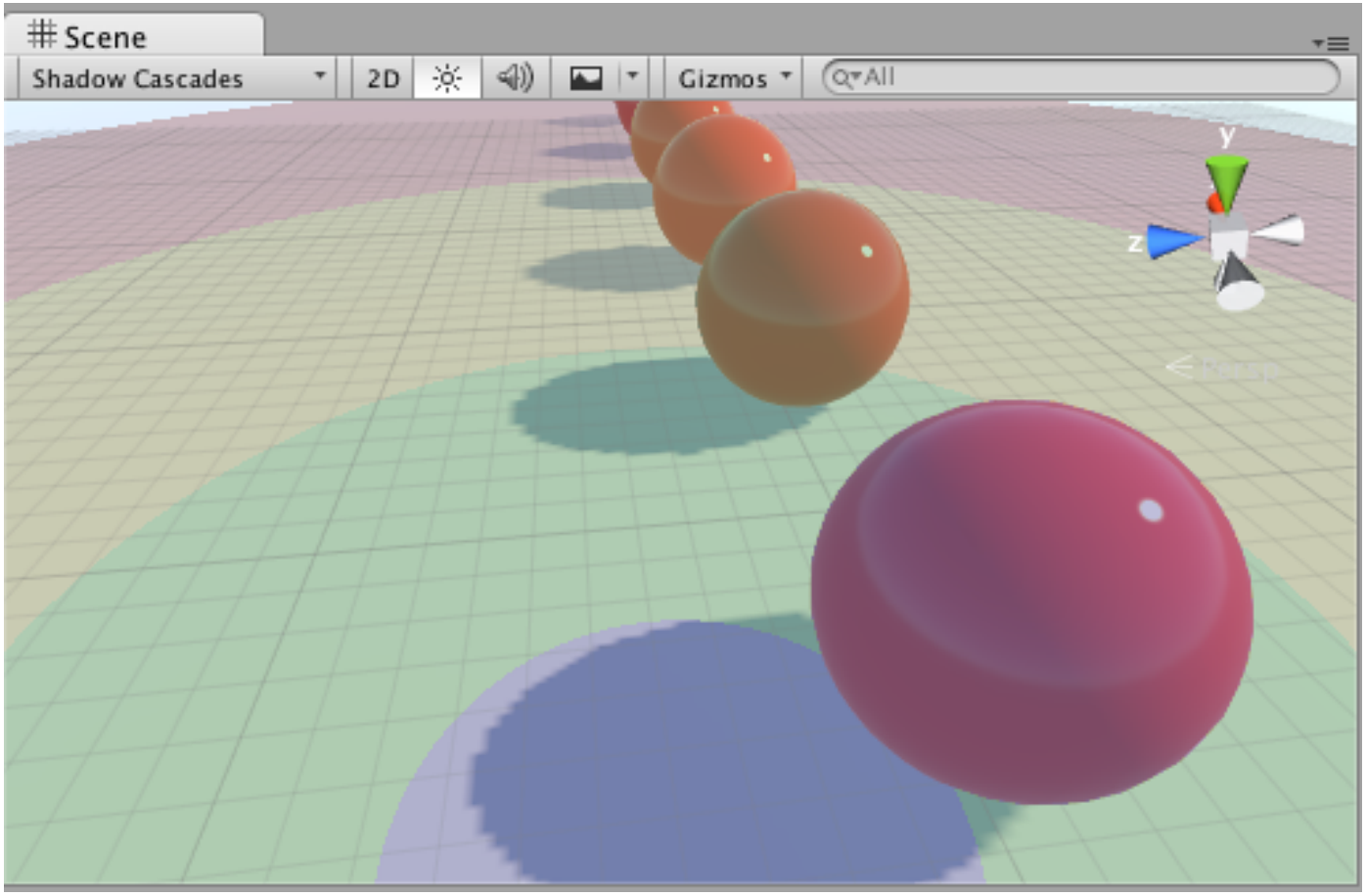
Cascaded Shadow Maps



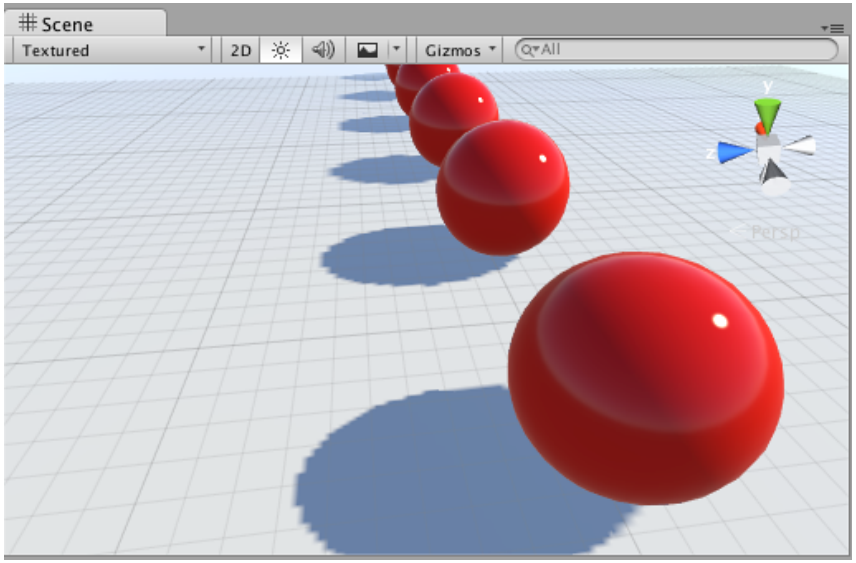


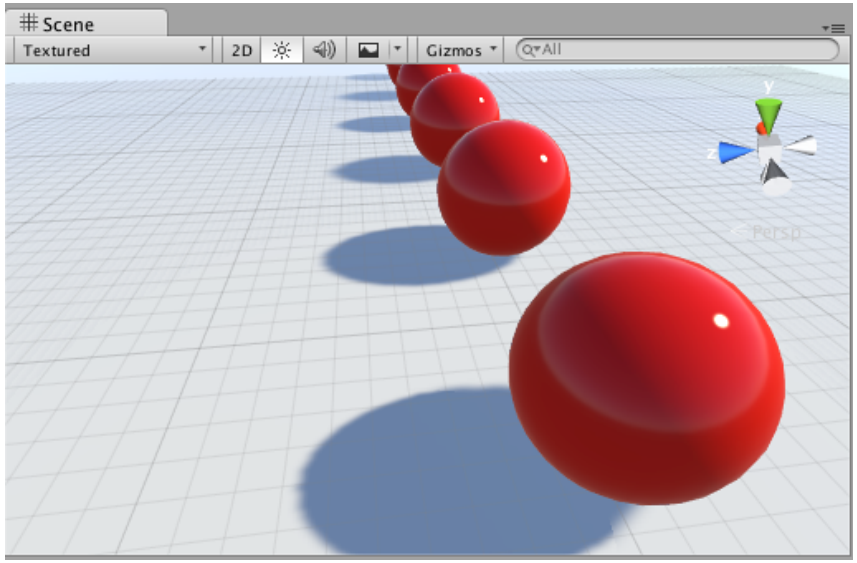


Cascaded Shadow Map Example (Unity)



Comparison





References

- **Movania, OpenGL Development Cookbook**
- McGuire M., “Efficient Shadow Volume Rendering,” GPU Gems (on Moodle)
- Shreiner, D., Woo, M., Neider, J. and Davis, T., “OpenGL Programming Guide”, Fifth Edition
- Everitt, Cass, “Shadow Mapping,” NVIDIA Presentation (on Moodle)
- Real-time Rendering, 3rd Edition
- Real-time Shadows, CRC Press
- Matthew Fisher, Stanford (http://graphics.stanford.edu/~mdfisher/Shadows.html)
- MSDN documentation (https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324(v=vs.85).aspx)