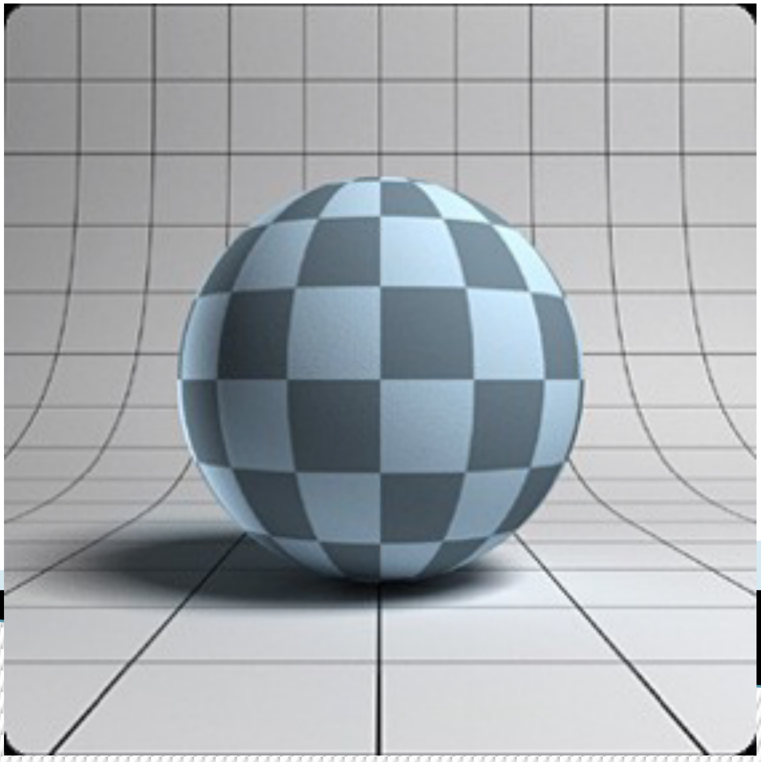
CS452/552; EE465/505

Shadow Mapping

4-07–15



Outline

- Shadow Mapping

Read:

- Angel, Chapter 5:
  - 5.10 Projections and Shadows
  - 5.11 Shadow Maps
- WebGL Programming Guide: Chapter 10
- WebGL Academy: www.webglacademy.com,
- Learning WebGL: learningwebgl.com lesson 16, render-to-texture

Project#2 posted due: April 23rd

Shadows

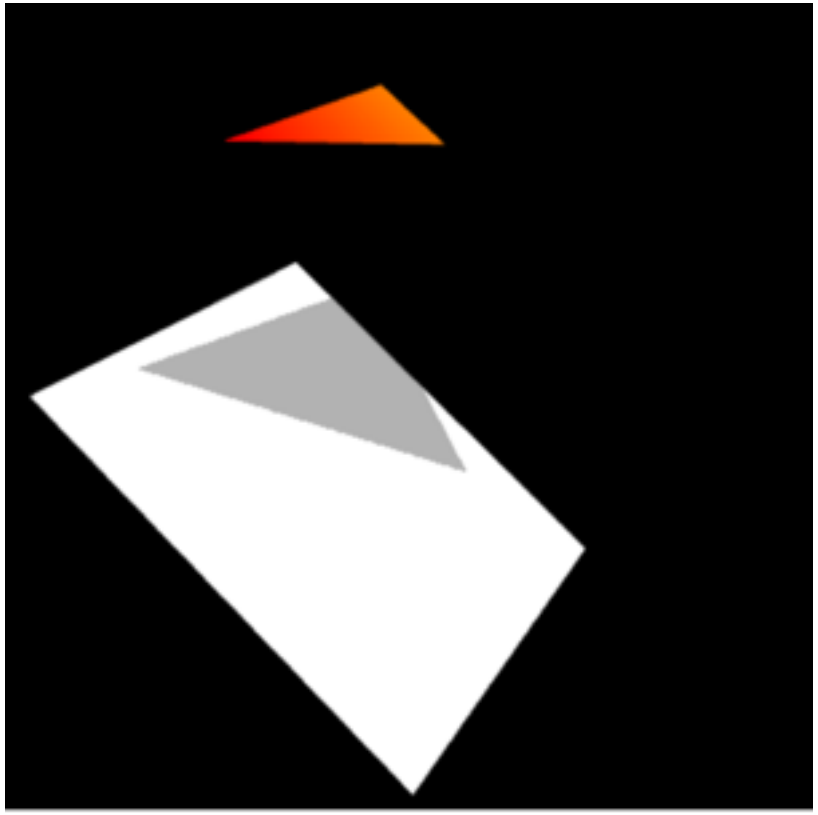
- Shadows increase scene realism
  - real world has shadows
  - good for depth perception
  - good for immersive games
    - dramatic effects
    - spooky effects
  - Other art forms use effectively

source: warmphotos.net

source: fractalenlightenment.com

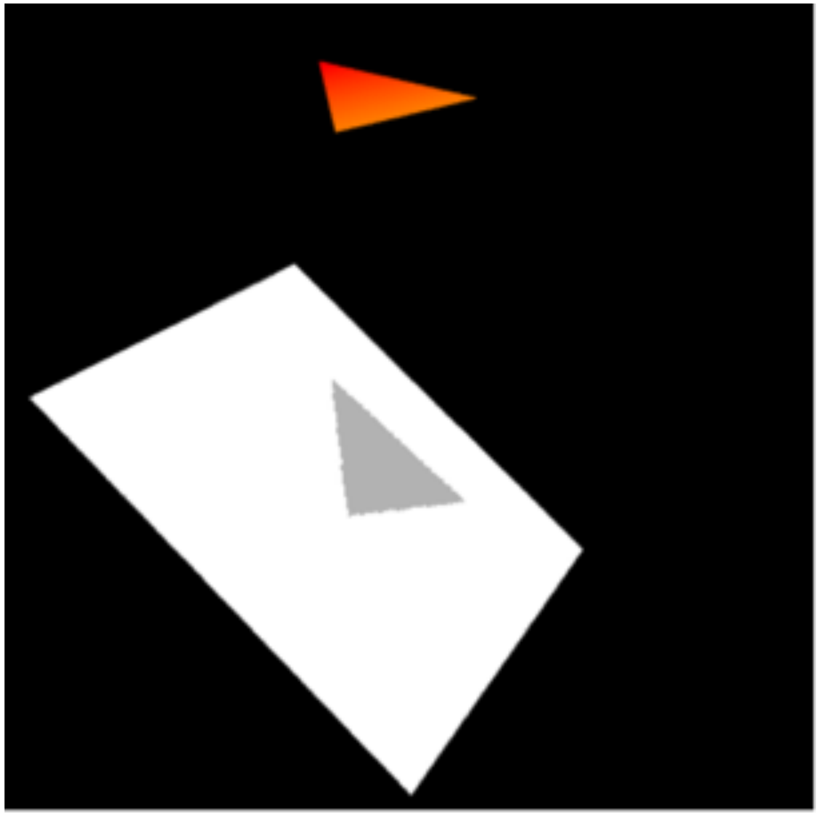
Examples: WebGL Programming Guide, Chapter 10

Shadow.html



shadow of red triangle cast onto slanted white rectangle

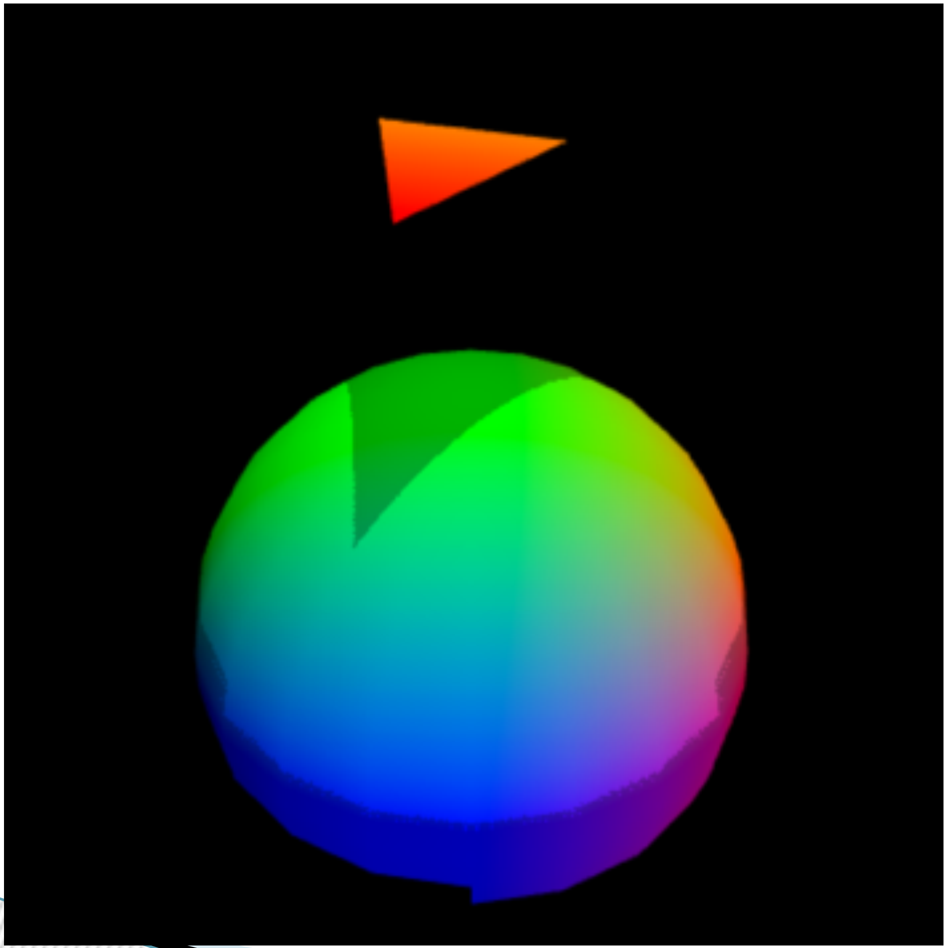
Shadow_highp.html



same, but uses more precision (fragment shader changed)

Examples: WebGL Programming Guide, Chapter 10

Shadow_highp_sphere.html



high precision shadow of red triangle cast onto sphere

Flashlight in the Eye Graphics

- When do we not see shadows in a real scene?
- When the only light source is a point source at the eye or center of projection
  - Shadows are behind objects and not visible
- Shadows are a global rendering issue
  - Is a surface visible from a source
  - May be obscured by other objects

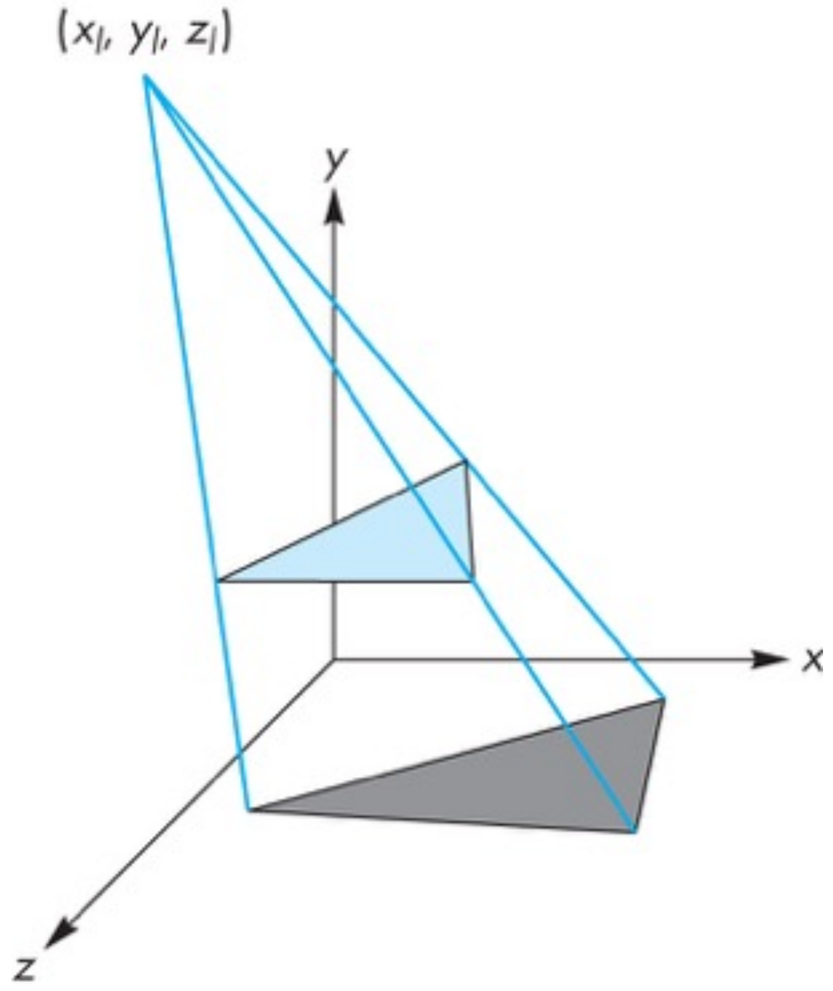
Shadows in Pipeline Renders

- Note that shadows are generated automatically by a ray tracers
  - feeler rays will detect if no light reaches a point
  - need all objects to be available
- Pipeline renderers work an object at a time so shadows are not automatic
  - can use some tricks: projective shadows
  - multi-rendering: shadow maps and shadow volumes

Projective Shadows

- Oldest methods
  - Used in flight simulators to provide visual clues
- Projection of a polygon is a polygon called a **shadow polygon**
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point source onto a surface

Shadow Polygon



Computing Shadow Vertex

1. Source at (x_l, y_l, z_l)
2. Vertex at (x, y, z)
3. Consider simple case of shadow projected onto ground at $(x_p, 0, z_p)$
4. Translate source to origin with $T(-x_l, -y_l, -z_l)$
5. Perspective projection

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}$$

6. Translate back

Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
2. Compute two model view matrices as uniforms
3. Send model view matrix for original triangle
4. Render original triangle
5. Send second model view matrix
6. Render shadow triangle
   - Note shadow triangle undergoes two transformations
   - Note hidden surface removal takes care of depth issues

Generalized Shadows

- Approach was OK for shadows on a single flat surface
- Note with geometry shader we can have the shader create the second triangle
- Cannot handle shadows on general objects
- Exist a variety of other methods based on same basic idea
- We'll pursue methods based on projective textures

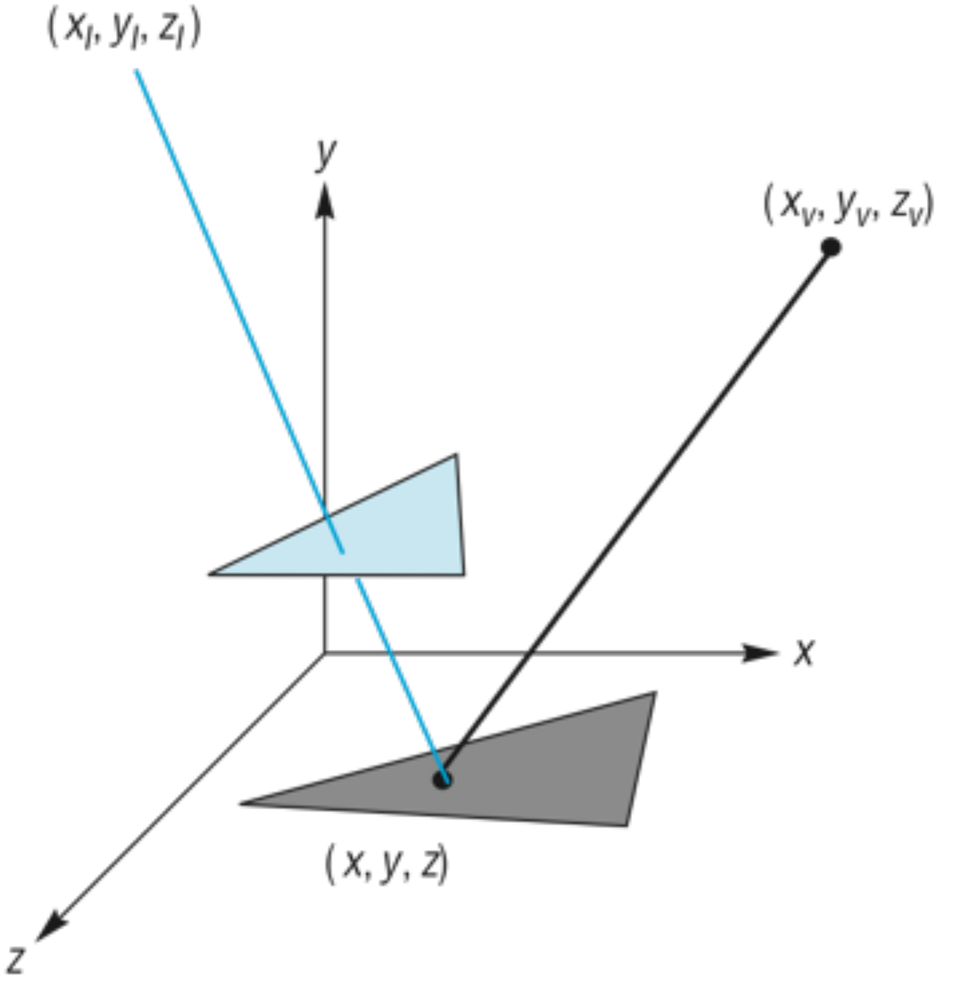
Image Based Lighting

- We can project a texture onto the surface in which case we are treating the texture as a "slide projector"
- This technique the basis of projective textures and image based lighting
- Supported in desktop OpenGL and GLSL through four dimensional texture coordinates
- Not yet in WebGL

Shadow Maps

- If we render a scene from a light source, the depth buffer will contain the distances from the source to nearest lit fragment.
- We can store these depths in a texture called a **depth map** or **shadow map**
- Note that although we don't care about the image in the shadow map, if we render with some light, anything lit is not in shadow.
- Form a shadow map for each source

Shadow Mapping



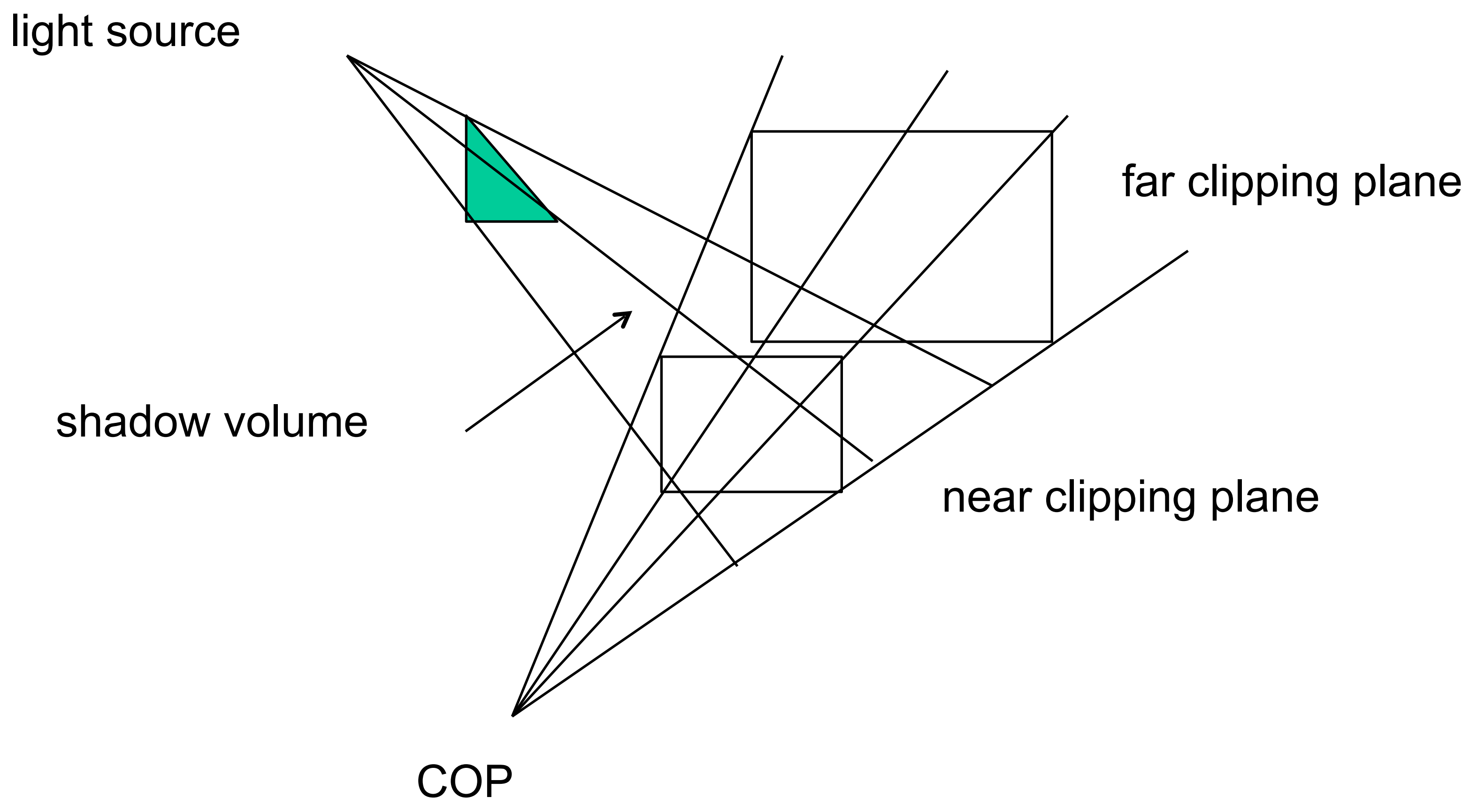
Final Rendering

- During the final rendering we compare the distance from the fragment to the light source with the distance in the shadow map
- If the depth in the shadow map is less than the distance from the fragment to the source the fragment is in shadow (from this source)
- Otherwise we use the rendered color

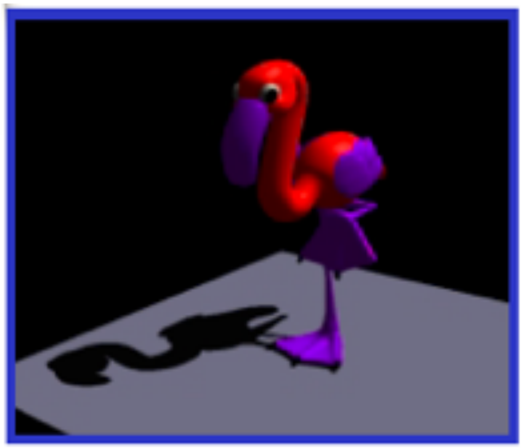
Implementation

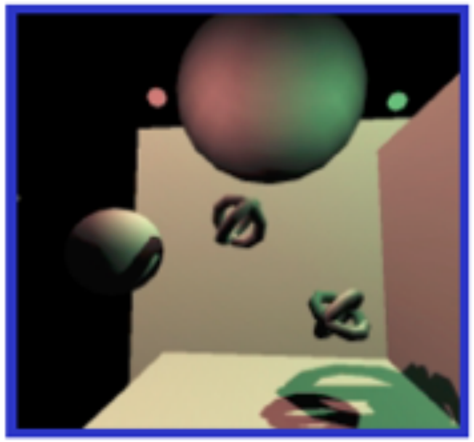
- Requires multiple renderings
- Recall render-to-texture
  - gives us a method to save the results of a rendering as a texture
  - almost all work done in the shaders

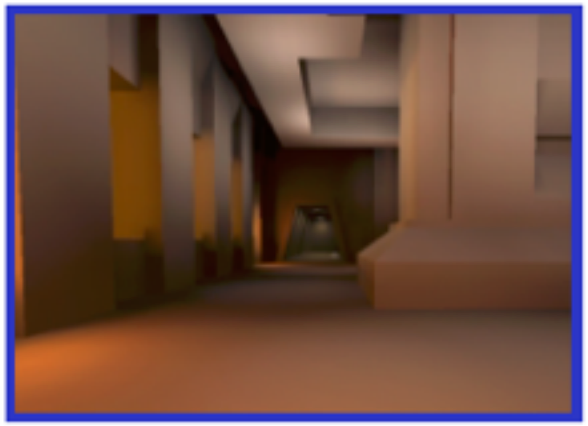
Shadow Volumes

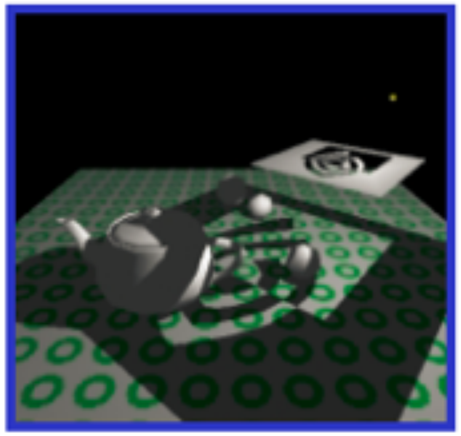


Common Real-time Shadow Techniques


Projected planar shadows


Shadow volumes


Light maps


Hybrid approaches

source: Mark Killgard, NVIDIA, GDC, 2001

Projecting Shadows

Algorithm	Pros	Cons
baked shadowlight maps	*best visual quality*	• *not dynamic* • *memory cost of textures*
projected texture shadows	*dynamic, easy to soften*	• *many visual artifacts* • *tricky to set up* • *extra rendering pass*
volumetric shadows	*dynamic, high quality*	• *computational expense* • *hard to set up*

Problems with Common Shadow Techniques

- Projected planar shadows
  - works well only on flat surfaces
- Stenciled shadow volumes
  - determining the shadow volume
- Light maps
  - unsuitable for dynamic shadows

In general, hard to get everything shadowed well

Another technique: Shadow Mapping

- Image-space shadow determined
  - Lance Williams published the basic idea in 1978
    - (also the year that Jim Blinn invented bump mapping)
  - Image-space algorithm
    - means no knowledge of the scene's geometry is required
    - must deal with aliasing artifacts
  - Well known software rendering technique
    - Pixar's RenderMan uses the technique
    - Basic shadowing technique for Toy Story, etc.

Shadow Mapping Concept

- Basic Idea:
  - the pixels “seen” by the light are lit
  - all other pixels are in shadow

Shadow Mapping Concept

- Two pass algorithm:
  - 1st pass: render the scene from the light's point of view and "remember" which pixels the light has seen
    - the result is a "shadow map"
    - essentially a 2D function indicating the depth of the closest pixels to the light
  - 2nd pass: render the scene as the camera sees it
    - if we "remembered" that pixel
      - gl_FragColor = ambient+diffuse+specular
    - else
      - gl_FragColor = ambient

Shadow Mapping Concept

- How do we “remember” which pixels are seen from the light’s point of view?

- Render the screen to the depth texture with GL_DEPTH_TEST enabled
  - OpenGL/WebGL allows you to create a framebuffer and render to it

Shadow Mapping Concept

- Depth testing from the light's point-of-view
  - Two pass algorithm
  - First, render the depth buffer from the light's point-of-view
    - the result is a "shadow map"
    - essentially a 2D function indicating the depth of the closest pixels to the light
  - This shadow map is used in the second pass

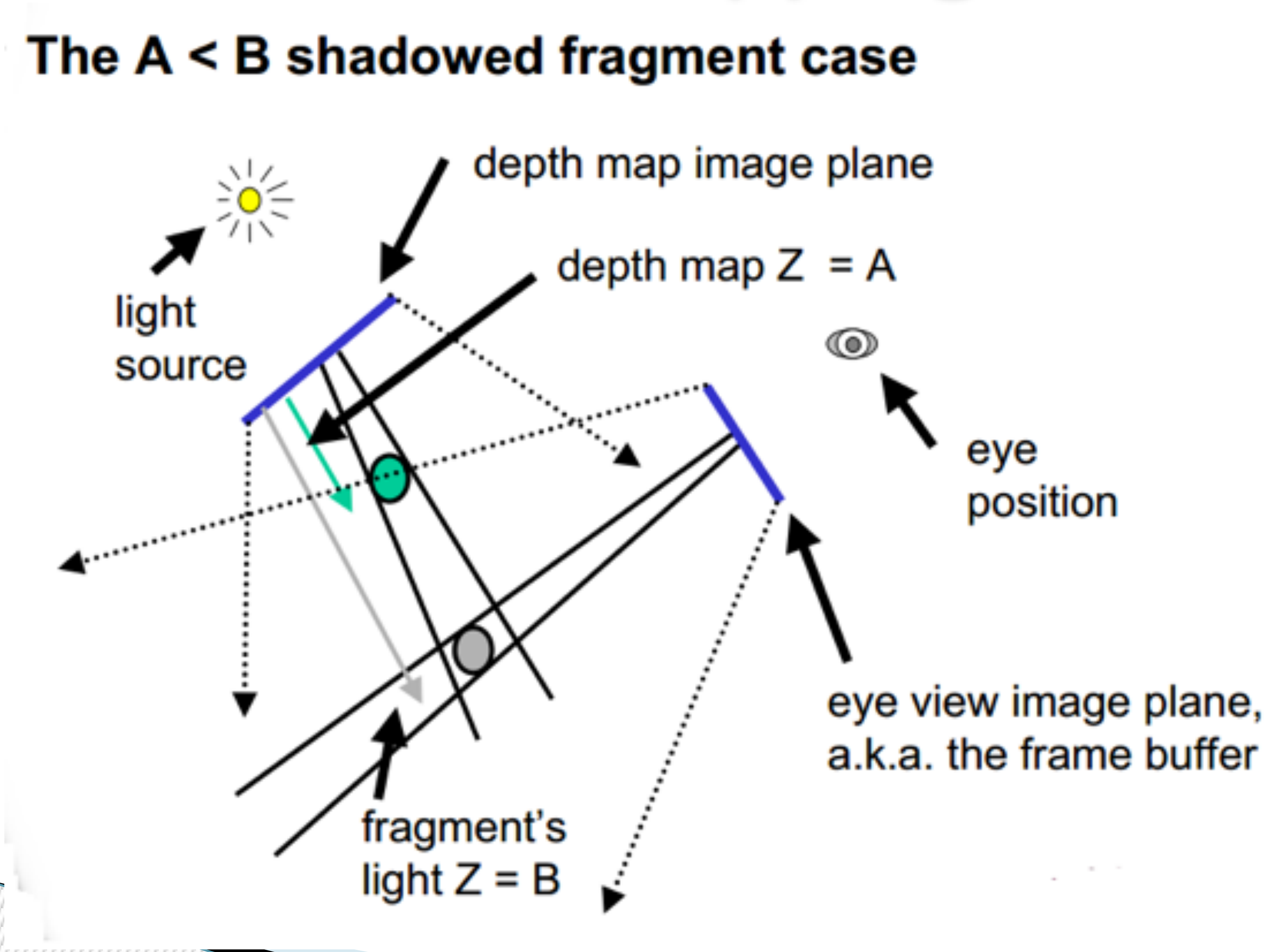
Shadow Mapping Concept

- Shadow determination with the depth map
  - Second, render scene from the eye's point-of-view
  - For each rasterized fragment
    - determine fragment's XYZ position relative to the light
    - this light position should be set up to match the frustum used to create the depth map
    - compare the depth value at light position XY in the shadow map to the fragment's light position Z

Shadow Mapping Concept

- The Shadow Map Comparison
  - Two values
    - A = Z value from the depth map at fragment's light XY position
    - B = Z value of the fragment's XYZ light position
  - If B is greater than A, then there must be something closer to the light than the fragment
    - so the fragment is shadowed
  - If A and B are approximately equal, the fragment is lit

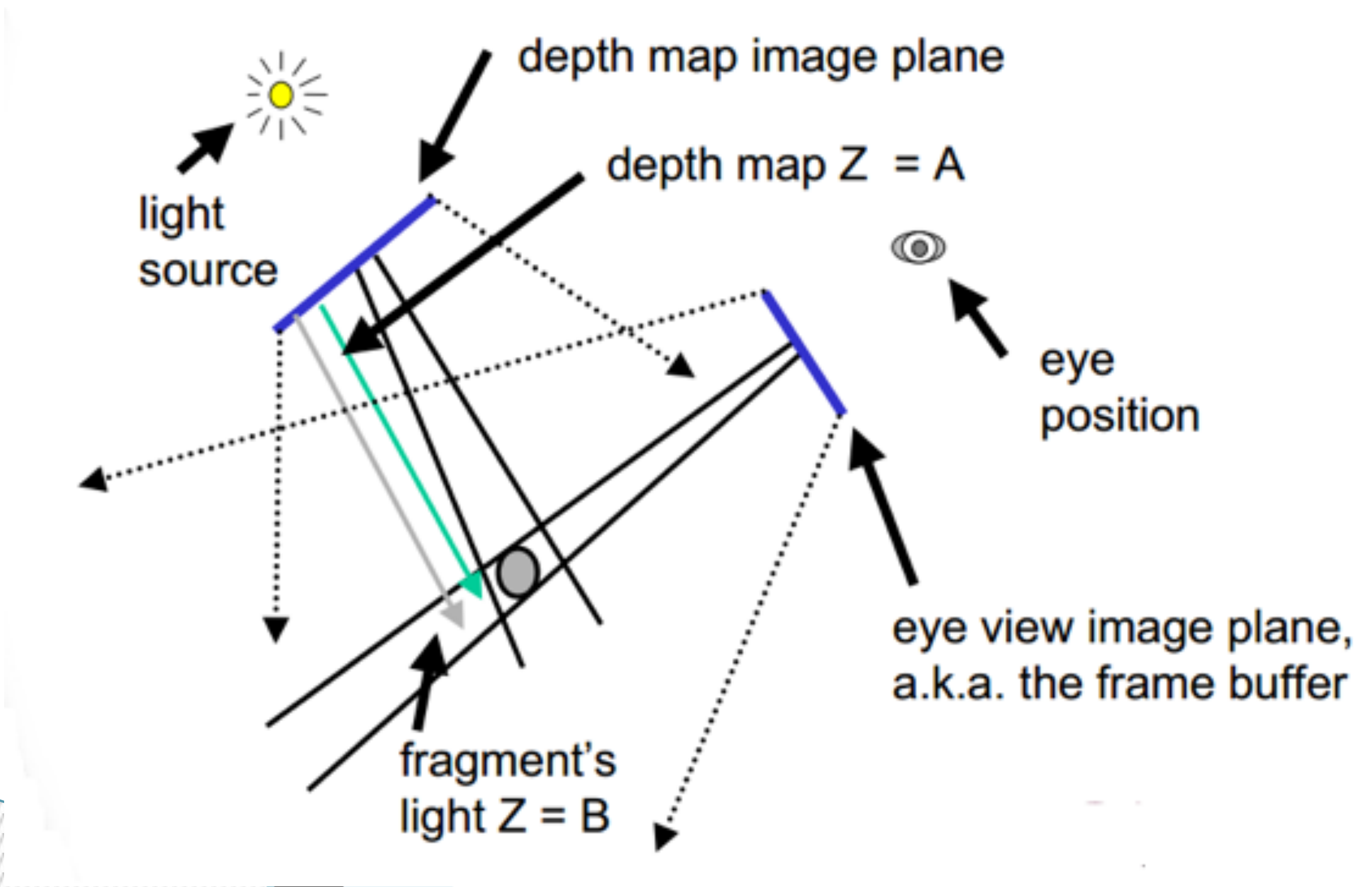
2D Shadow Mapping



source: Mark Killgard, NVIDIA, GDC, 2001

2D Shadow Mapping

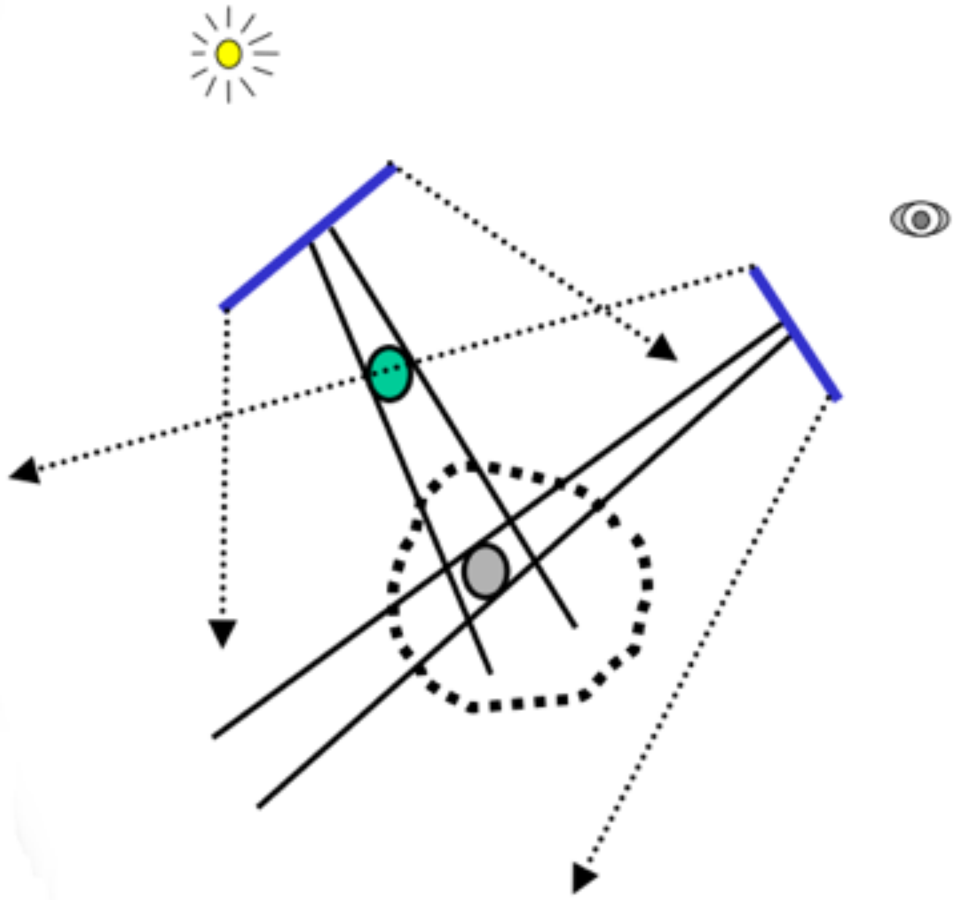
The A ≅ B unshadowed fragment case



source: Mark Killgard, NVIDIA, GDC, 2001

2D Shadow Mapping

Note image precision mismatch!

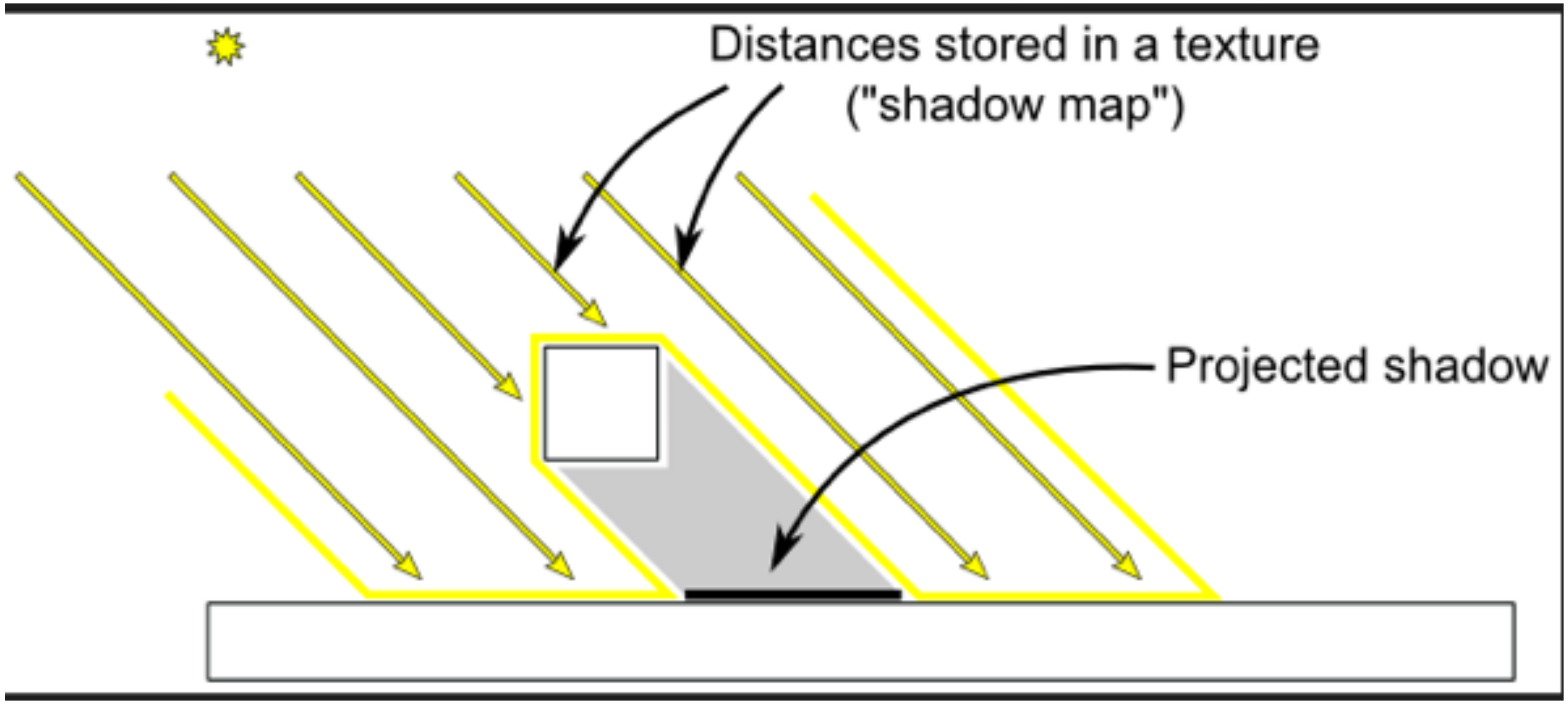


The depth map could be at a different resolution from the framebuffer

This mismatch can lead to artifacts

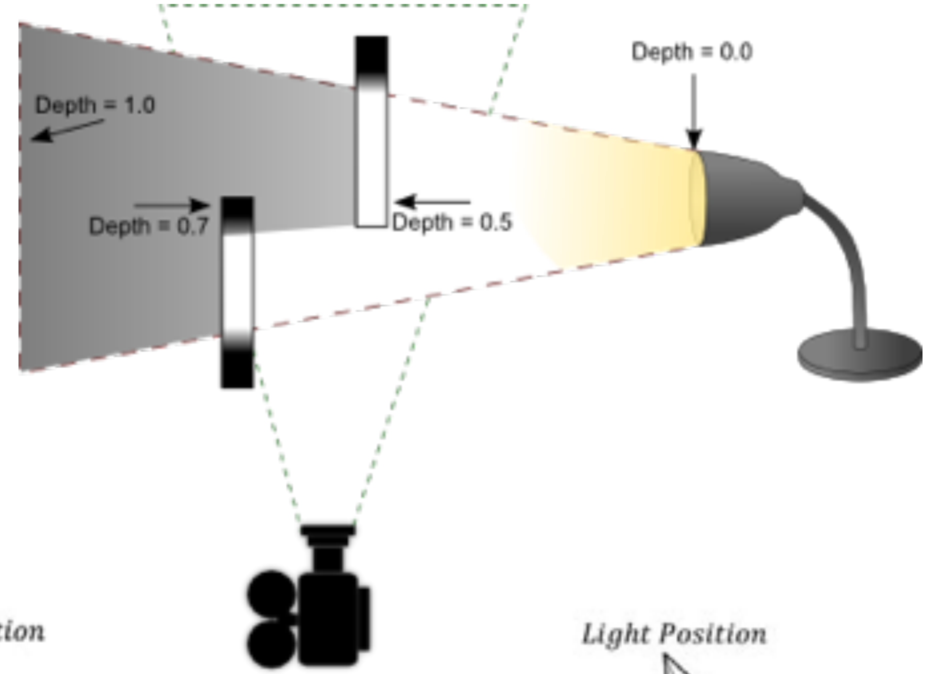
source: Mark Killgard, NVIDIA, GDC, 2001

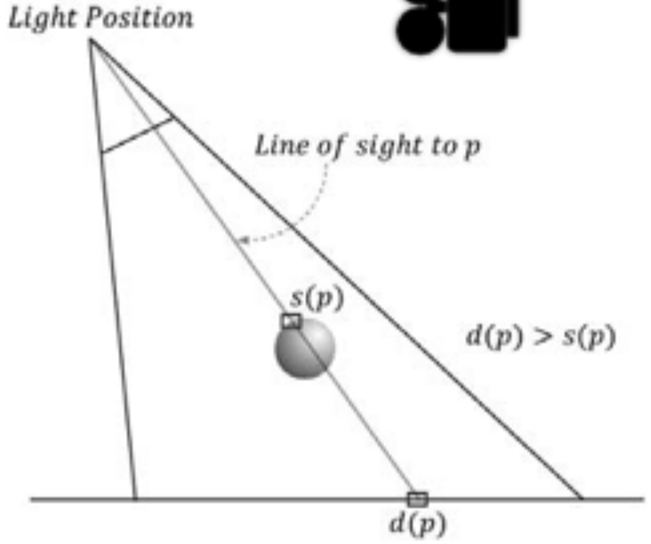
Simple shadow mapping

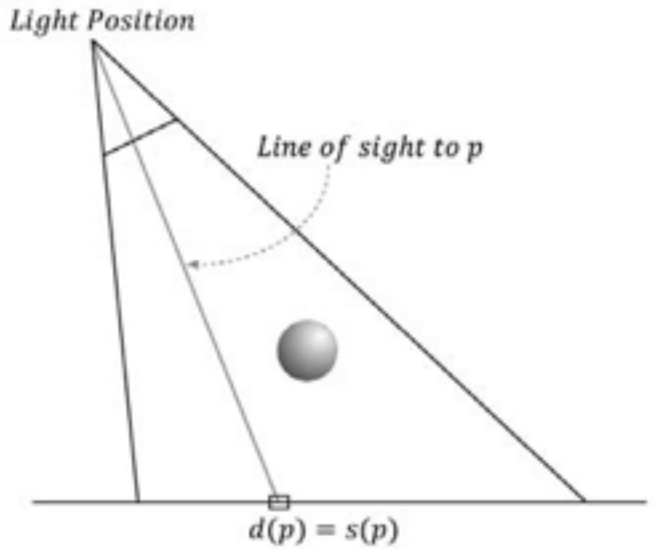


- Assume:
  - one light: multiple lights require multiple depthmaps
  - directional light: spotlights require more processing
  - the light is stationary: dynamic shadows are complex

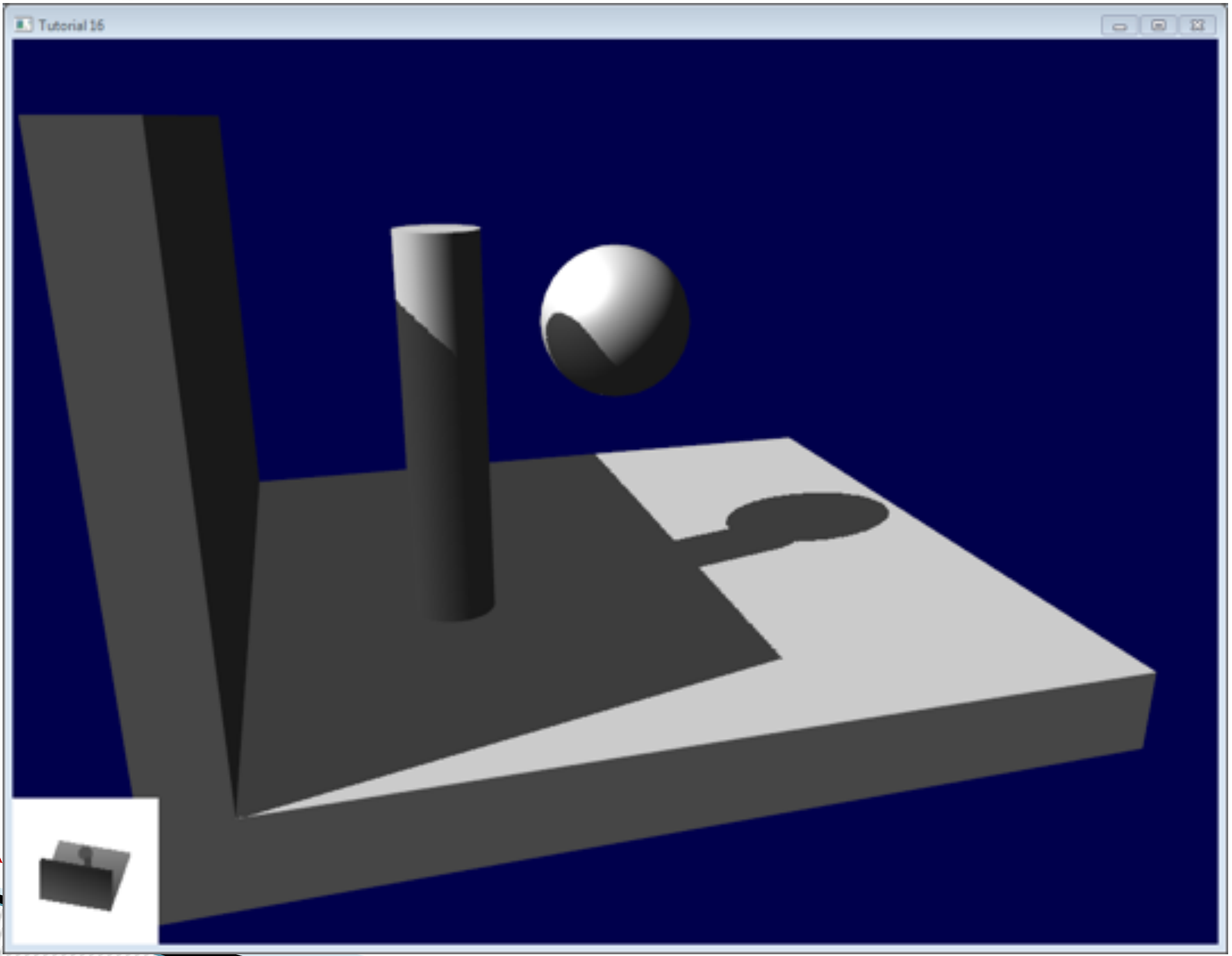
Using Depth


Depth = 0.0
Depth = 1.0
Depth = 0.7
Depth = 0.5


Light Position
Line of sight to p
s(p)
d(p) > s(p)
d(p)

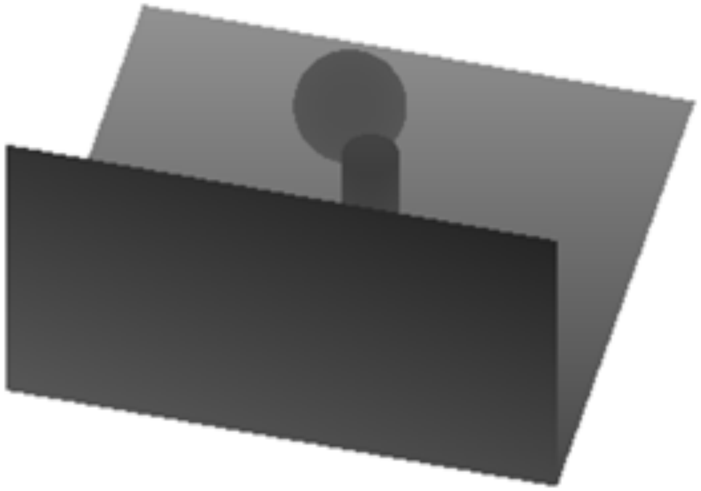

Light Position
Line of sight to p
d(p) = s(p)

Example Scene



1st Pass: Rendering a Shadow Map

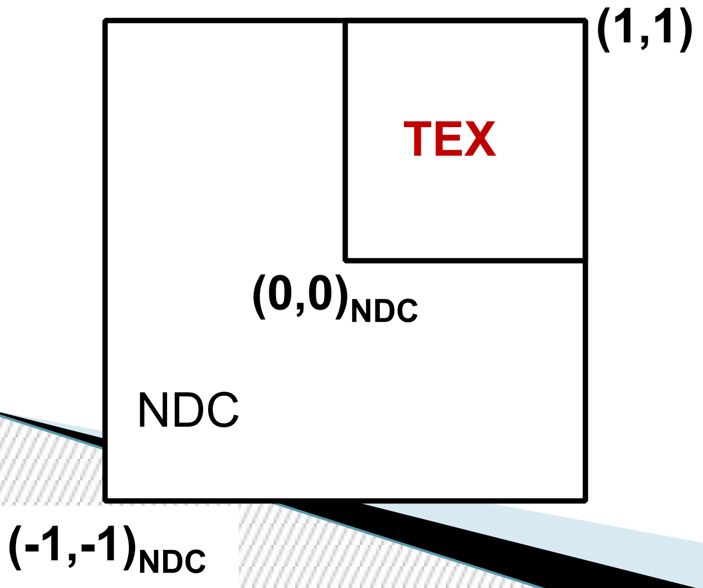
- Typically more than twice as fast as a normal render, because only low precision depth is written, instead of both the depth and the color
- Memory bandwidth is the biggest performance issue



- Example:
  - dark color means a small z
  - so, the upper right corner of the wall is near the camera
  - at the opposite, white means z=1 (in homogeneous coordinates), so this is very far

2nd Pass: Using the Shadow Map

- For each fragment that we compute, we must test whether it is “behind” the shadow map or not.
- To do this, must compute the current fragment’s position in *the same space* as the one we used when creating the shadowmap S. So, we need to transform it once with the usual MVP matrix, and another time with the depthMVP matrix.
- However, multiplying the vertex’ position by depthMVP will give homogeneous coordinates, which are in [-1,1]; but texture sampling must be done in [0,1]



- For example, a fragment in the middle of the screen will be in (0,0) in homogeneous coordinates; but since it will have to sample the middle of the texture, the UVs will have to be (0.5,0.5)

2nd Pass: Using the Shadow Map

- Can be fixed by tweaking in the fragment shader
- More efficient, though, to multiply the homogeneous coordinates by the following matrix
  - divides coordinates by 2 (the diagonal: [-1,1] -> [-0.5,0.5]
  - and translates them (lower row: [-0.5,0.5] -> [0,1])

```
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;
```

2nd Pass: Vertex Shader

- gl_Position is the position of the vertex as seen from the current camera
- ShadowCoord is the position of the vertex as seen from the light

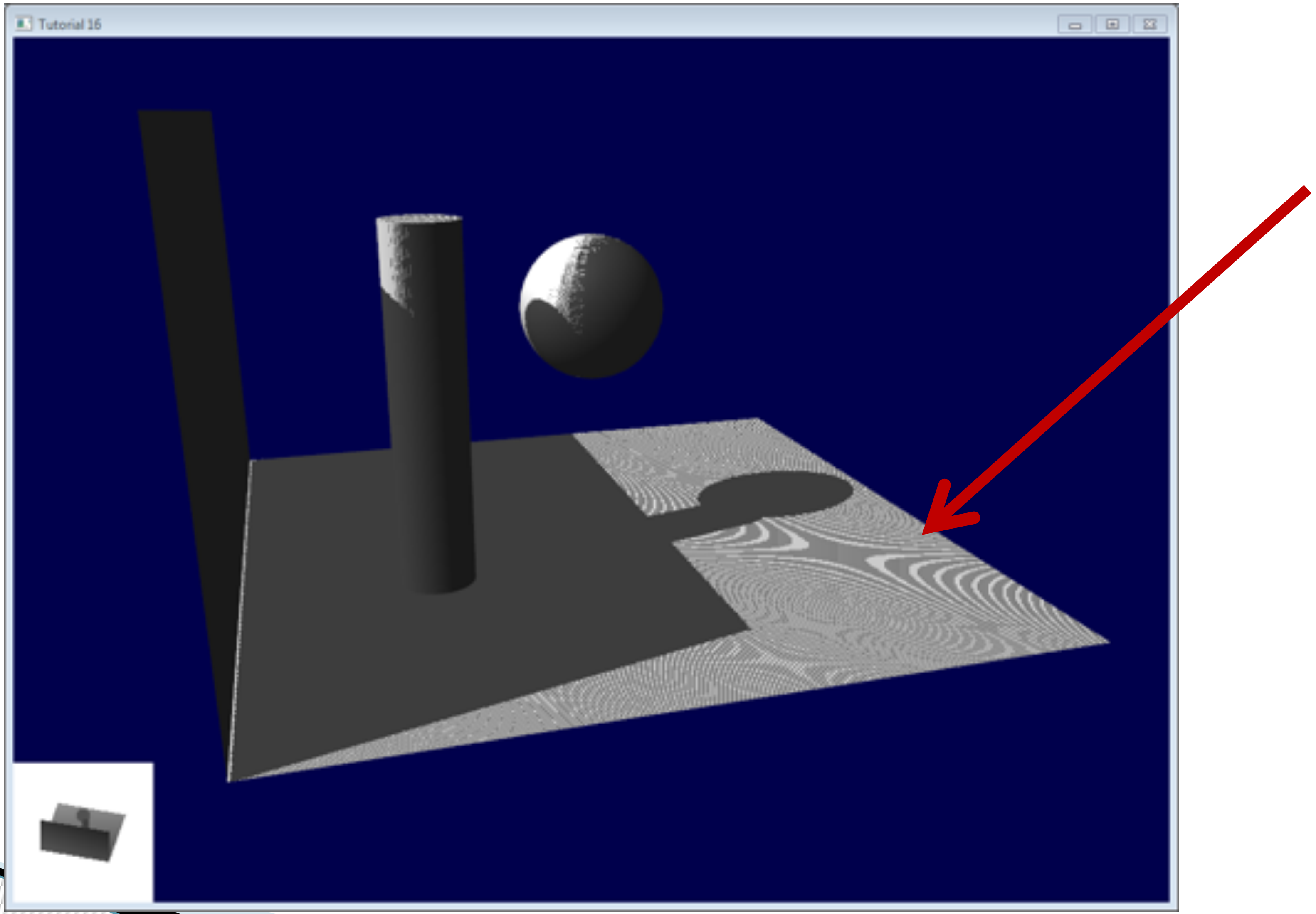
```
// Output position of the vertex, in clip space : MVP * position
gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

// Same, but with the light's view matrix
ShadowCoord = DepthBiasMVP * vec4(vertexPosition_modelspace,1)
```

2nd Pass: Fragment Shader

- texture(shadowMap, ShadowCoord.xy).z is the distance between the light and the nearest occluder
- ShadowCoord.z is the distance between the light and the current fragment
  - so, if the current fragment is further than the nearest occluder, this means we are in the shadow (of that nearest occluder)
  - Use this to modify the shading

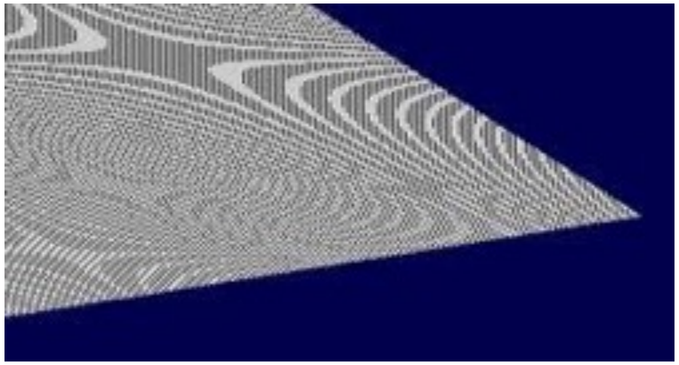
```
float visibility = 1.0;
if ( texture( shadowMap, ShadowCoord.xy ).z  <  ShadowCoord.z){
    visibility = 0.5;
}

color =
 // Ambient : simulates indirect lighting
 MaterialAmbientColor +
 // Diffuse : "color" of the object
 visibility * MaterialDiffuseColor * LightColor * LightPower * cosTheta+
 // Specular : reflective highlight, like a mirror
 visibility * MaterialSpecularColor * LightColor * LightPower * pow(cosAlpha,5);
```

Result: Shadow Acne!

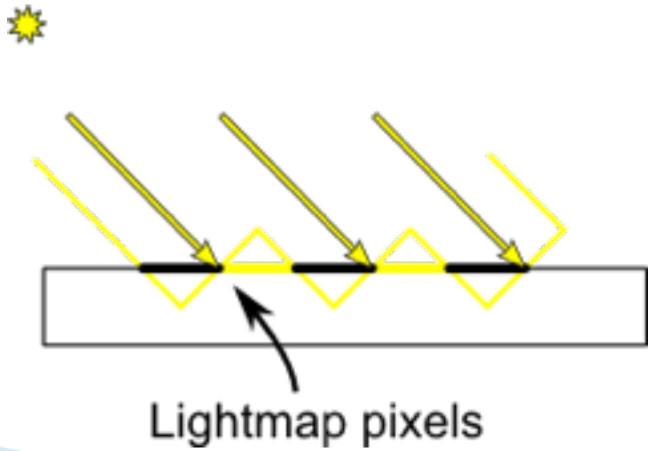


Problems:

- Known as "shadow acne"



- Consider the following:

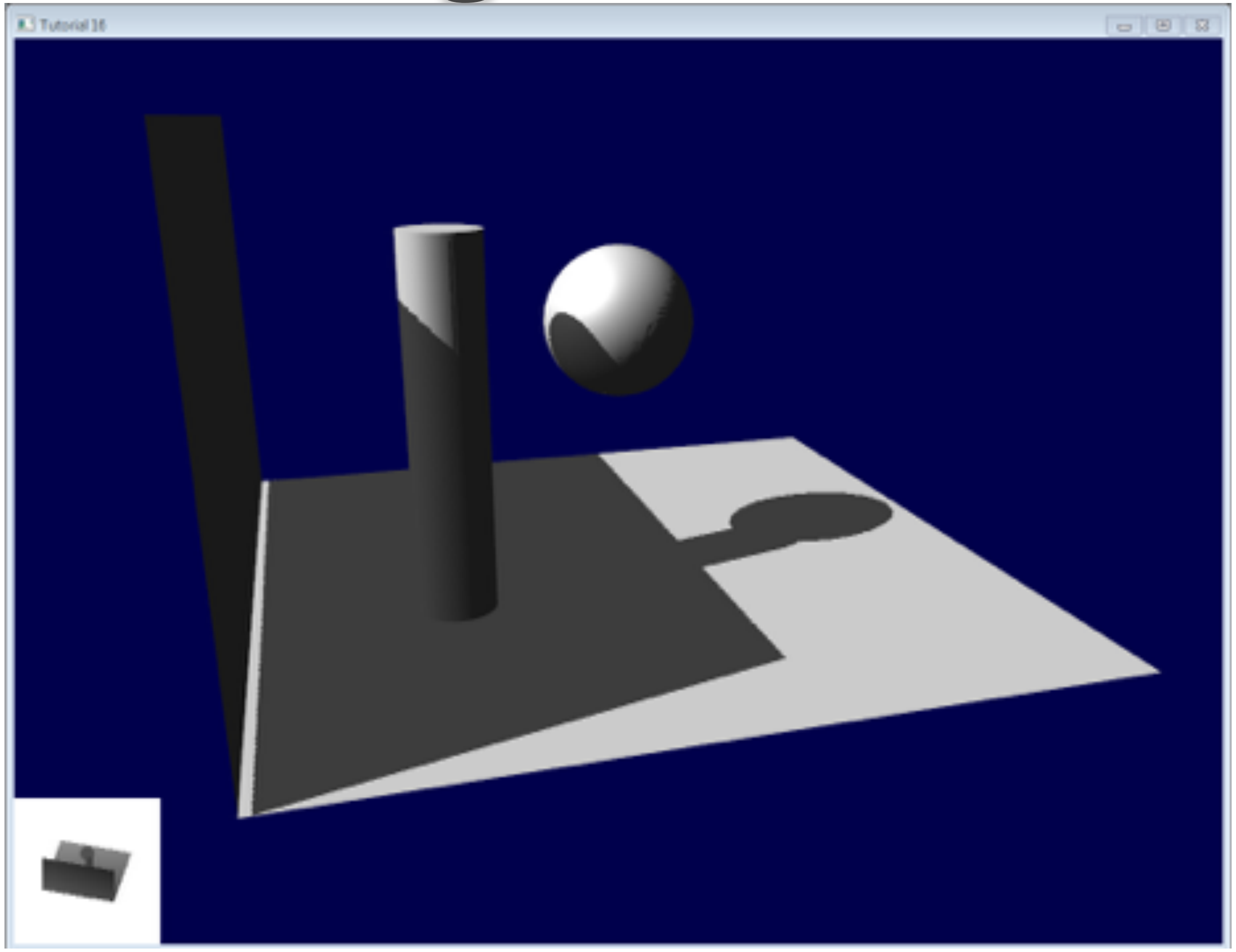


Fixing Shadow Acne

- The usual “fix” is to add an error margin
  - we only shade if the current fragment’s depth (in light space) is really far away from the lightmap value
  - that is, add a “bias”

```
float bias = 0.005;
float visibility = 1.0;
if ( texture2D( shadowMap, ShadowCoord.xy ).z  <
                                          ShadowCoord.z-bias){
    visibility = 0.5;
}
```

Result: using a Bias



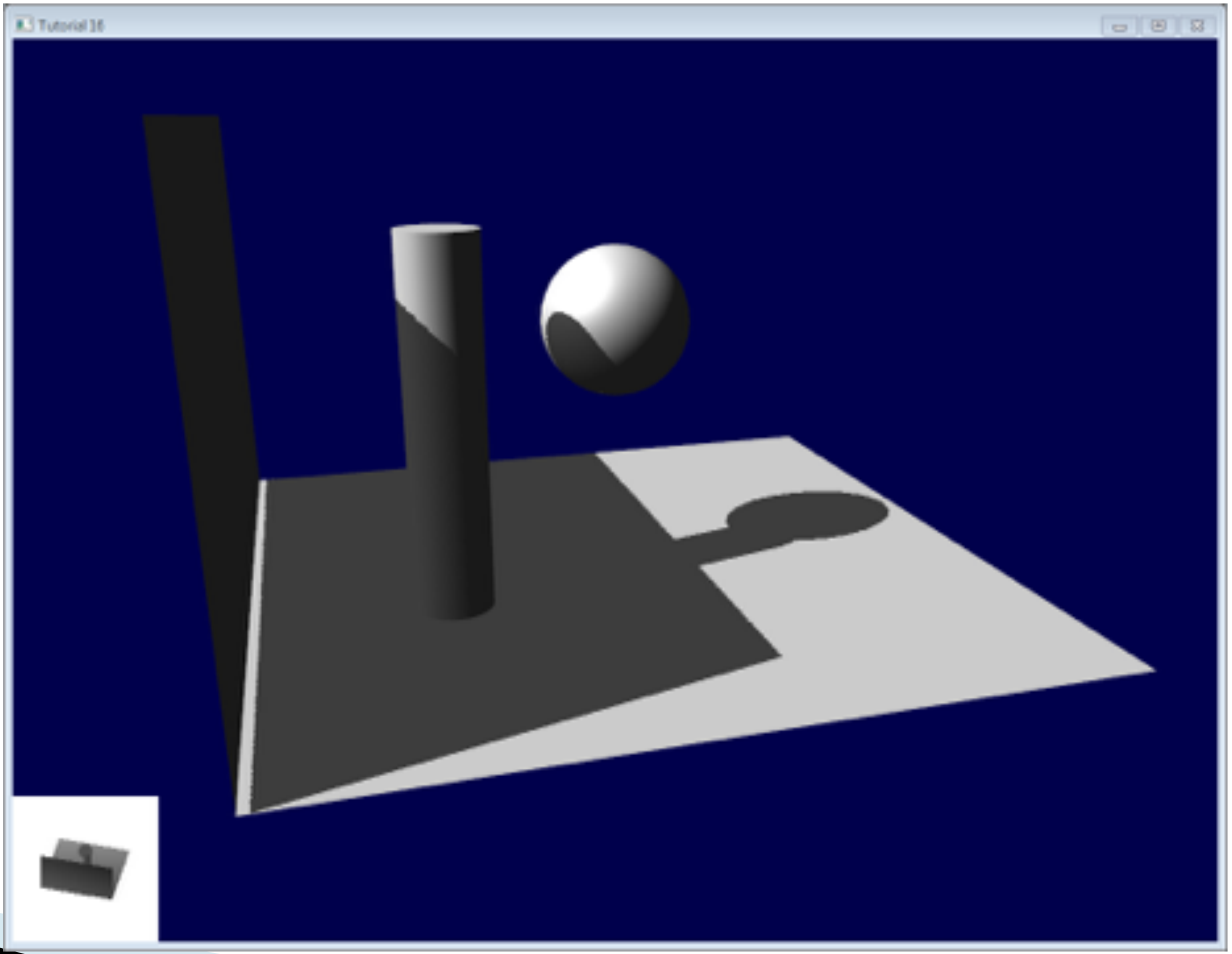
- Another problem: artifact between the ground and the wall is worse; also a bias of 0.005 seems too much on the ground, but not enough on the surface (some artifacts on the cylinder & sphere)

Tweaking the Bias

- Common approach: modify the bias according to the slope

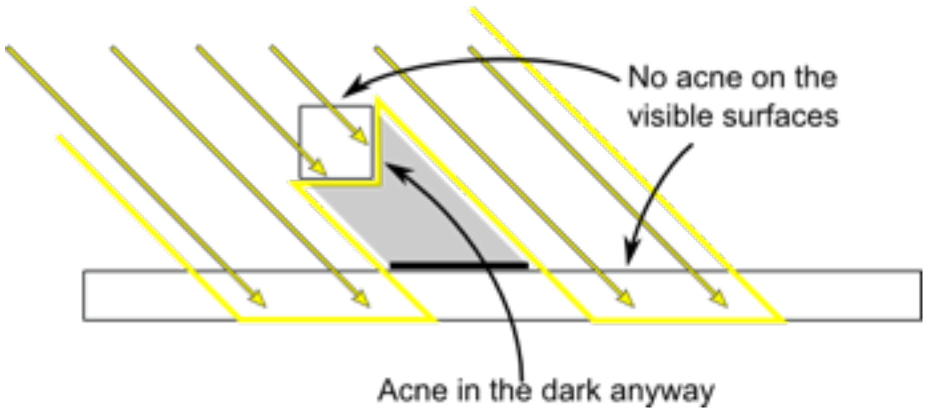
```
float bias = 0.005*tan(acos(cosTheta)); // cosTheta is dot( n,l ), clamped between 0 + 1
bias = clamp(bias, 0,0.01);
```

- No more shadow acne!



Another trick

- Render only the back faces in the shadow map
- Only works with some geometries, with thick walls, but at least the acne will be on surfaces in the shadow

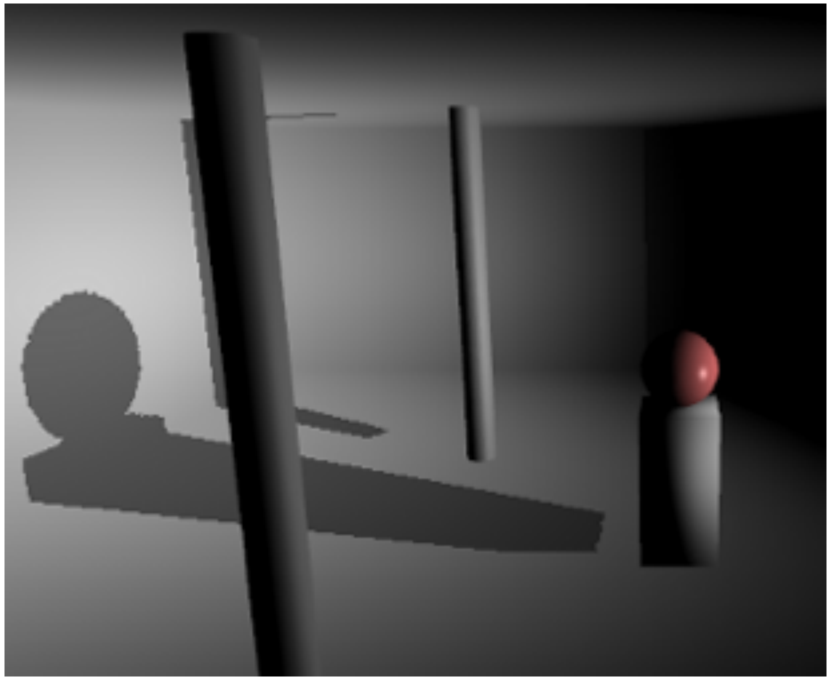


```
// We don't use bias in the shader, but instead we draw back faces,
// which are already separated from the front faces by a small distance
// (if your geometry is made this way). In OpenGL this is done as follows:
glCullFace(GL_FRONT); // Cull front-facing triangles ; draw only back-facing ones

glCullFace(GL_BACK); // Cull back-facing triangles -> draw only front-facing  ones
```

  - When rendering the scene, render normally (backface culling)

Peter Panning

- Shadow acne is gone, but some shadows are detached from the object; this is called "Peter Panning"



from Walt Disney's Peter Pan, 1953

Peter Panning: correction

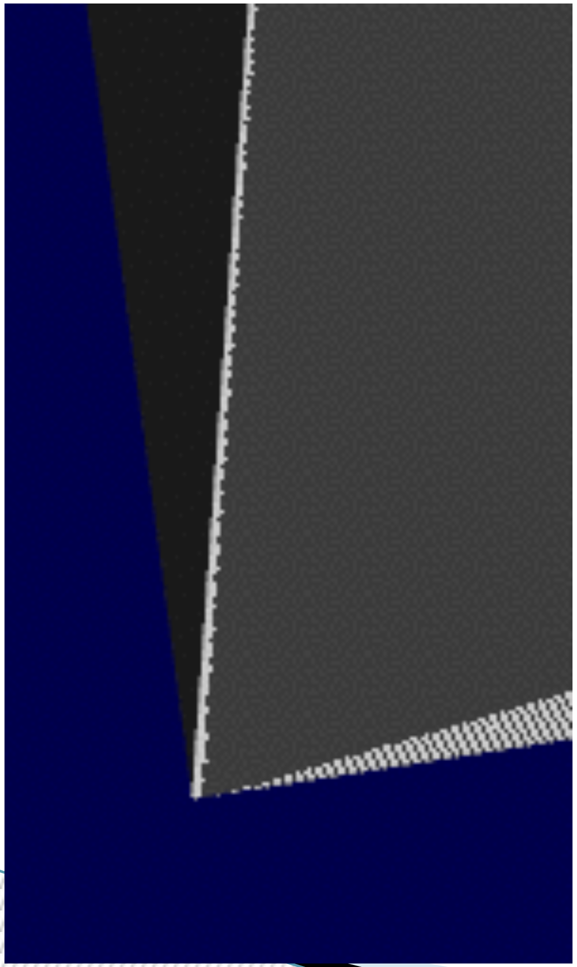
- One way to minimize both shadow acne and peter panning is to use linear depth.
  - That is, instead of using the projected !-coordinate, we instead calculate the distance between the vertex and the light source in view space
  - Still need to map the result between 0.0 and 1.0
  - Combined with filtering algorithms, gives good results
    - VSM: Variance Shadow Maps
    - ESM: Exponential Shadow Maps

Peter Panning: correction

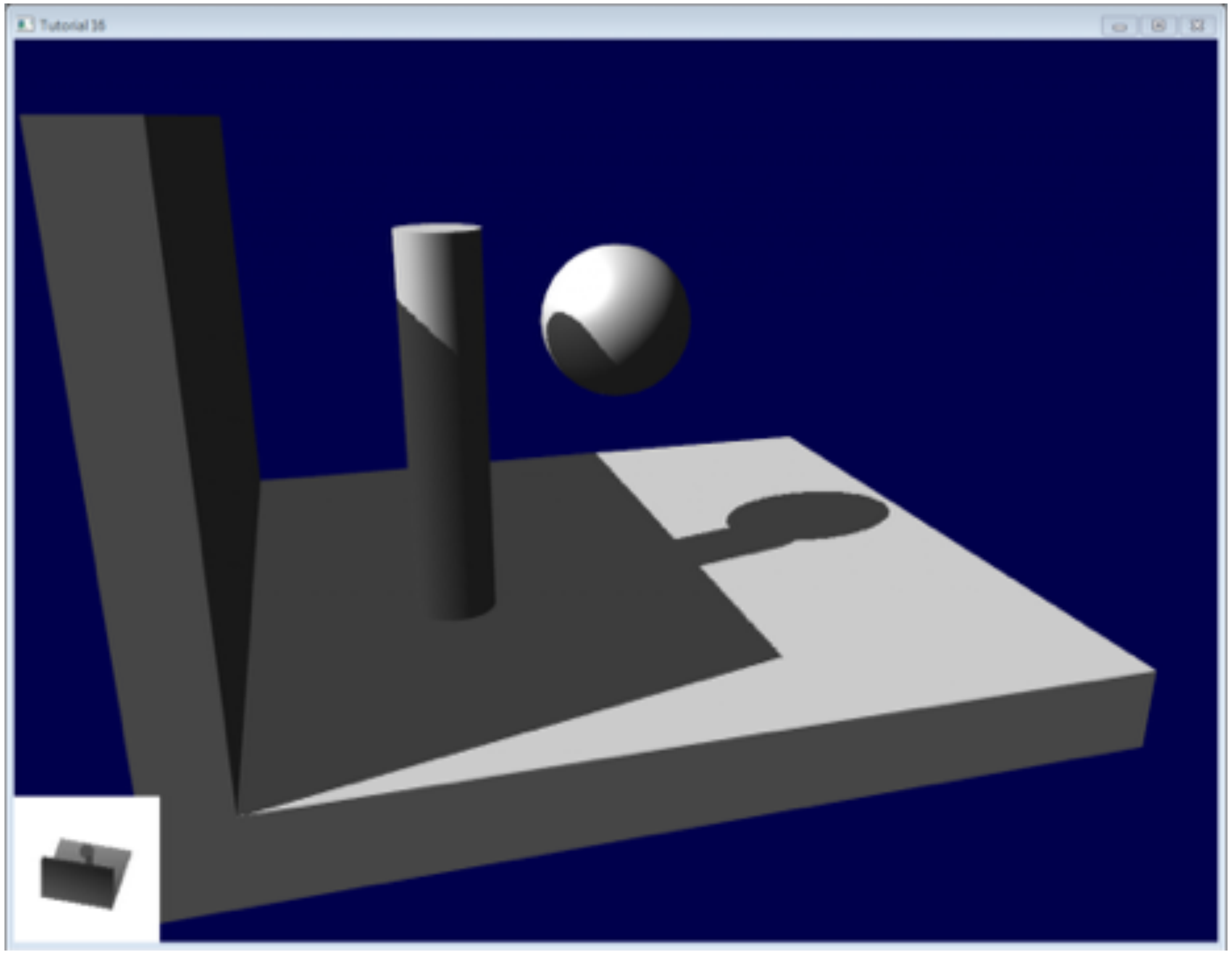
- Another approach: avoid thin geometry
  - Solves Peter Panning: if the geometry is more deep than your bias, you're all set
  - You can turn on backface culling when rendering the lightmap, because now there is a polygon of the wall which is facing the light, which will occlude the other side, which wouldn't be rendered with backface culling
- Drawback: more triangles to render (two times per frame!)

Peter Panning Example

With Peter Panning:

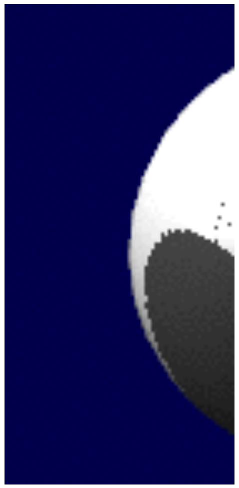


Without Peter Panning:



Aliasing

- Notice that there is still aliasing on the border of the shadow.
  - one pixel is white, the next is black, without a smooth transition inbetween



  - easy solution: change the shadowmap's sampler type to sampler2DShadow
  - can also add Poisson Sampling

Improvements

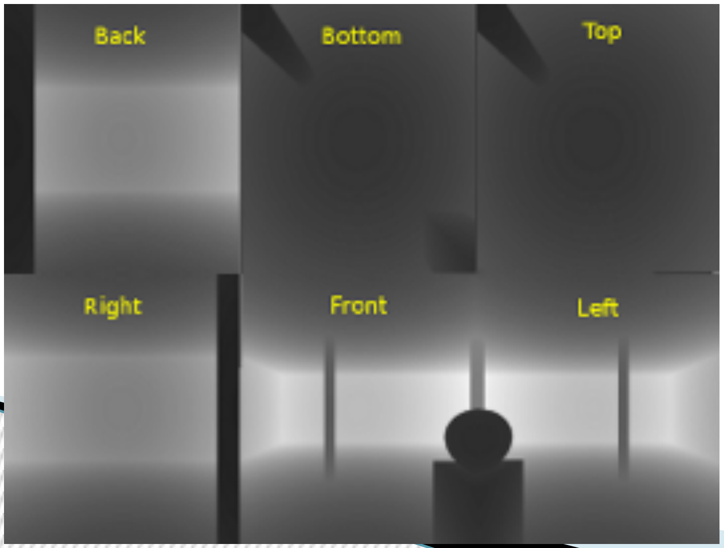
- Early bailing
  - instead of taking 16 samples for each fragment, take 4 distant samples.
    - if either all 4 are in the light, or in the shadow, assume the other 12 also are
    - if not all 4 are in the light (or shadow), process more

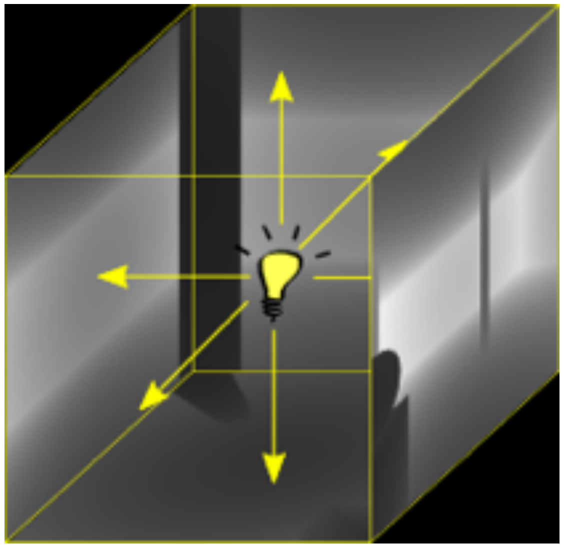
Improvements

- Spot lights
  - easy to add: change the orthographic projection matrix into a perspective one
  - also, take into account the perspective in the shader

Improvements

- Point lights
  - similar to spot light, but with depth cubemaps
  - a cubemap is a set of 6 textures
    - it is not accessed with standard UV coordinates, but with a 3D vector representing a direction
  - the depth is stored for all directions in space, which makes it possible for shadows to be cast around the point light





Improvements

- Combination of several lights
  - each light requires an additional rendering of the scene in order to produce the shadowmap
    - requires a lot of memory

Improvements

- Automatic light frustum
  - the light frustum used here was hand-crafted to contain the whole scene
    - better to avoid this
    - projection matrix of the light should be as tight as possible
- Exponential shadow maps
  - limit aliasing by assuming that a fragment which is in the shadow, but near the light surface, is “somewhere in the middle”
    - related to the bias, but the test is not binary

Improvements

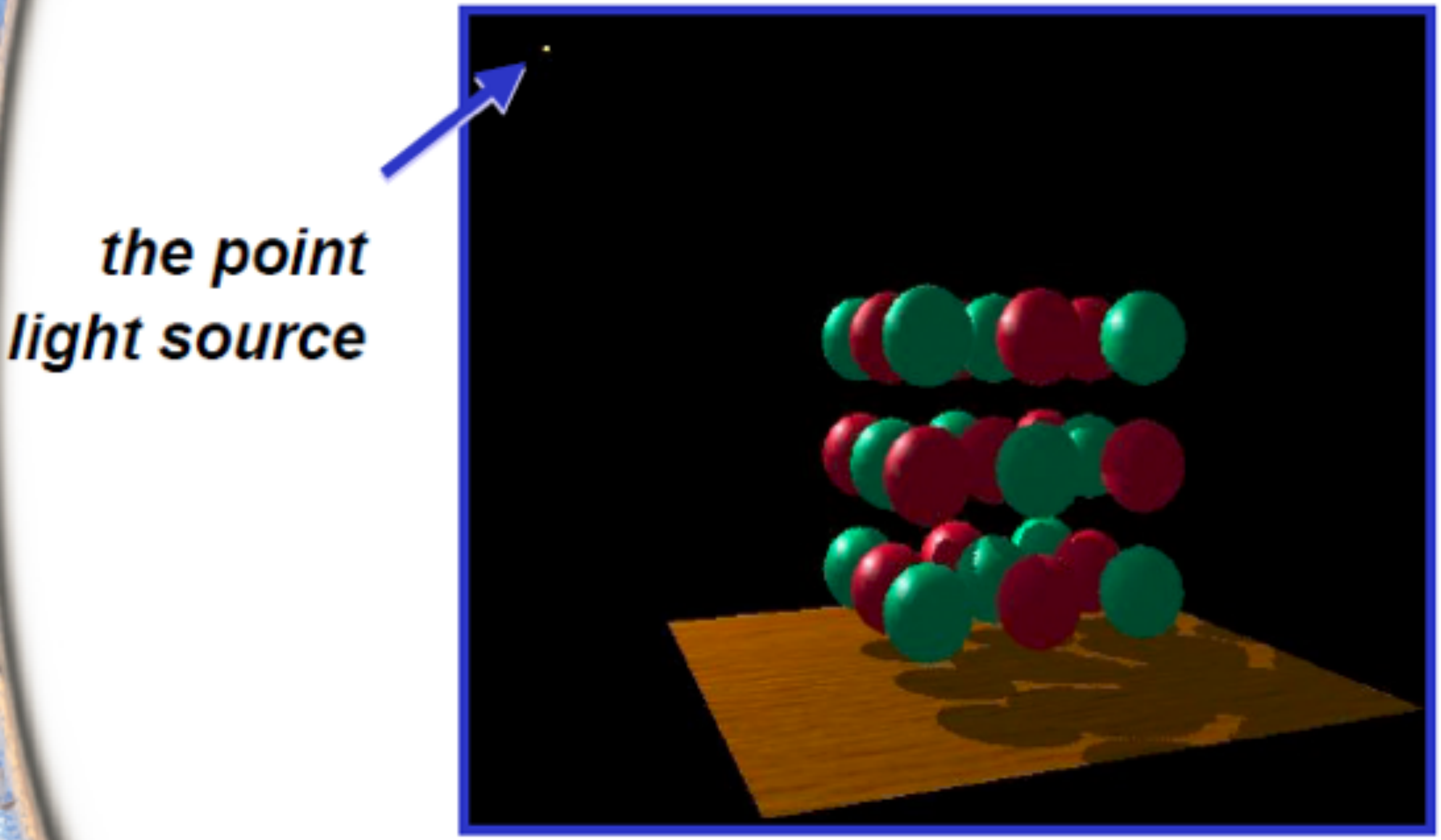
- Light-space perspective Shadow Maps
  - LiSPSM tweaks the light project matrix in order to get more precision near the camera

- Cascaded shadow maps
  - CSM deals with the same problem as LiSPSM, but uses a different approach
    - uses several (2 to 4) standard shadow maps for different parts of the view frustum
    - has a good balance of complexity to quality

- Parallel-Split Shadow Maps

Visualizing Shadow Mapping

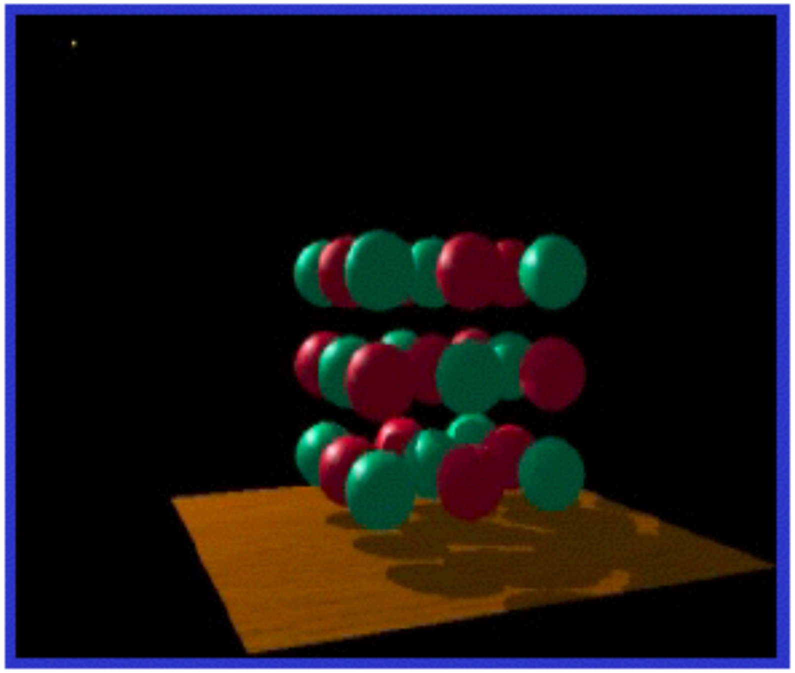
- A fairly complex scene with shadows

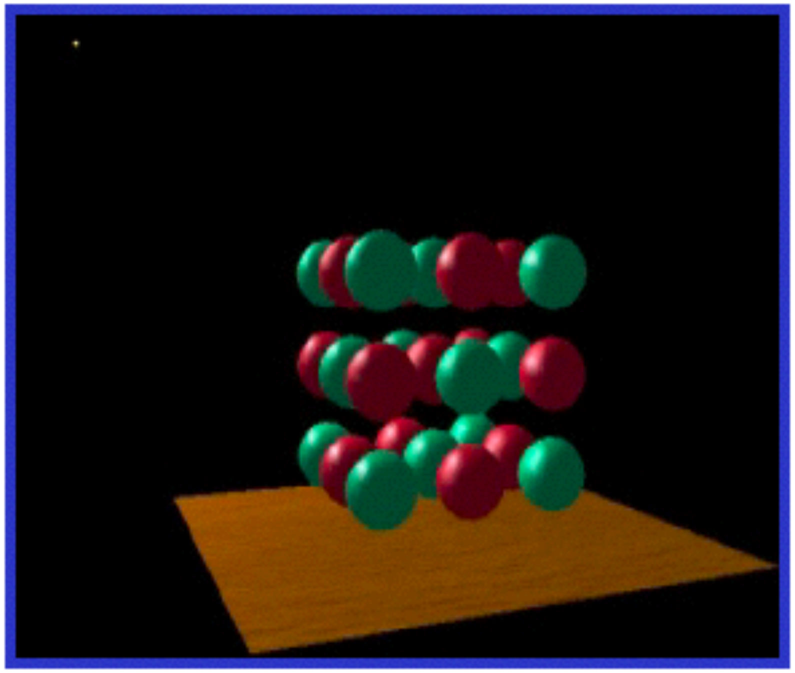


source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

- Compare with and without shadows

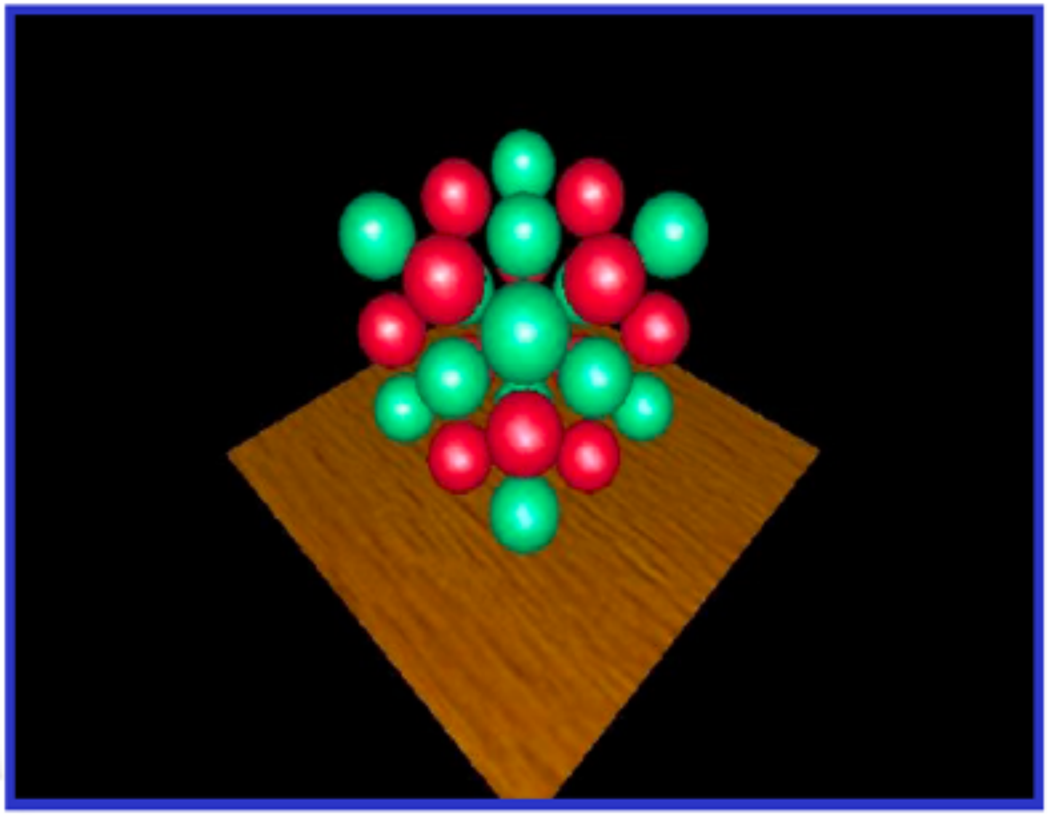

with shadows

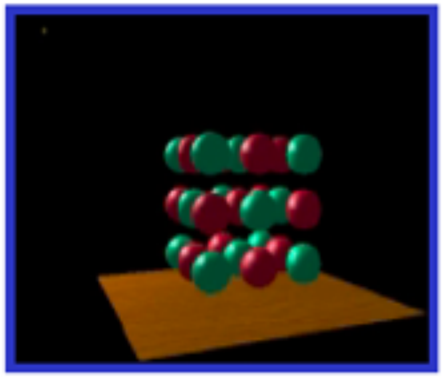

without shadows

source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

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



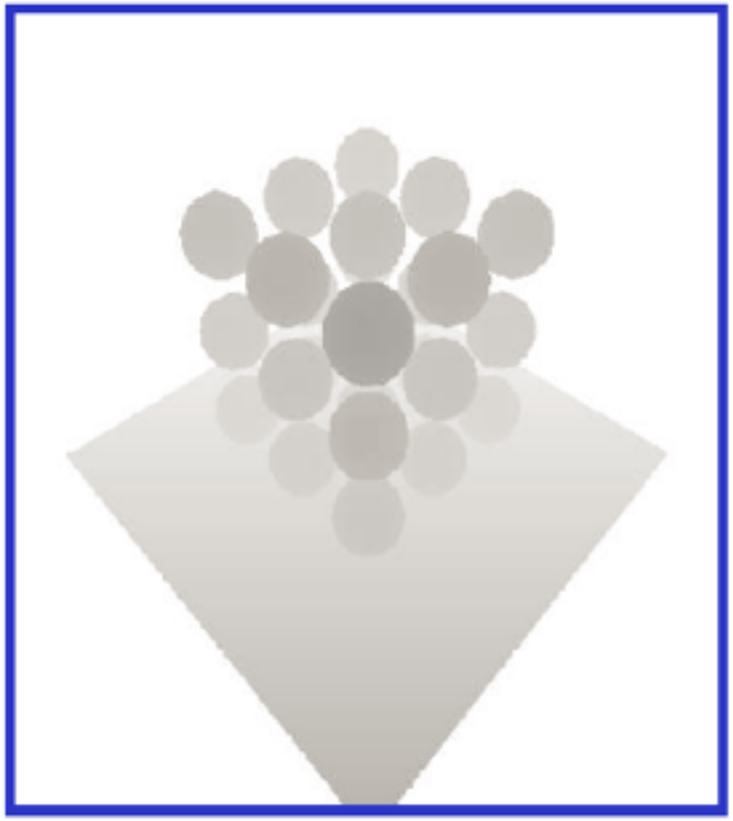


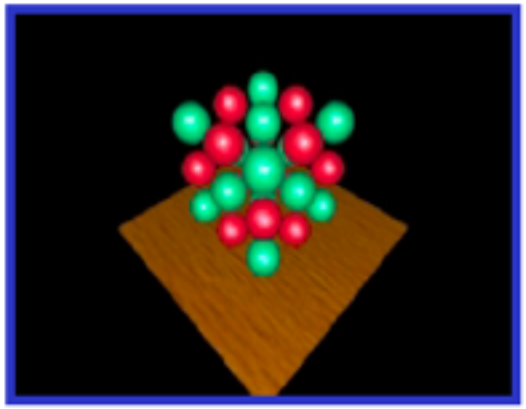
FYI: from the eye's point-of-view again

source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

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



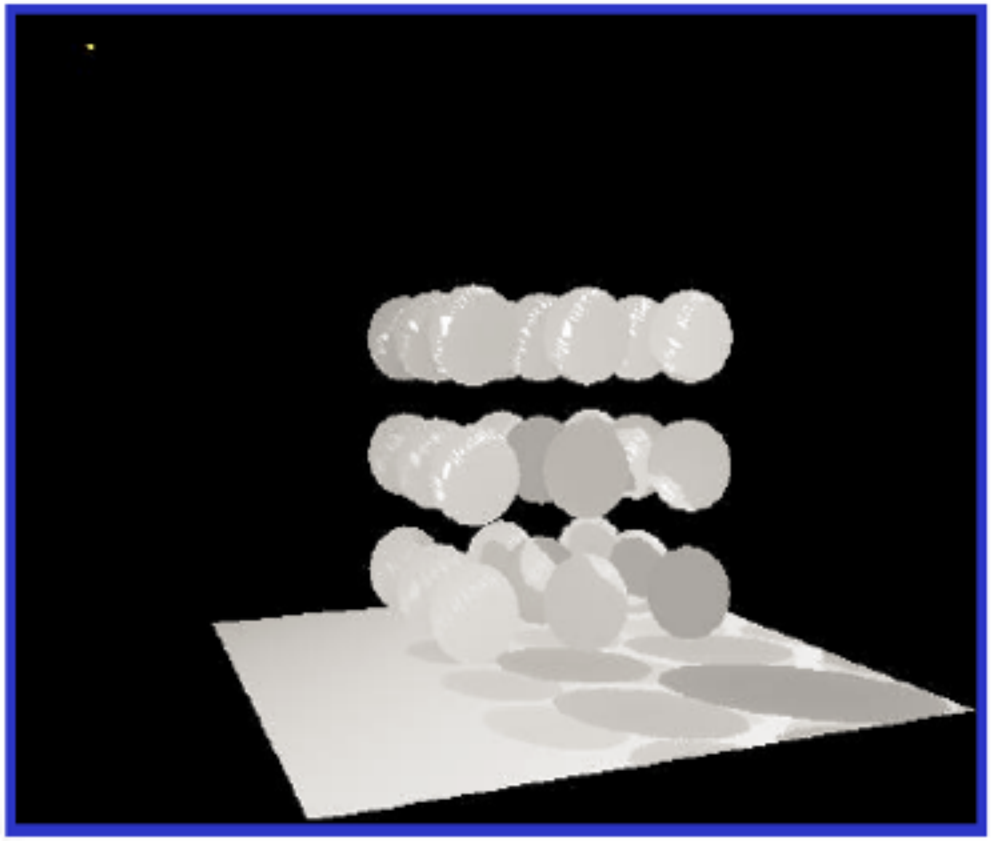


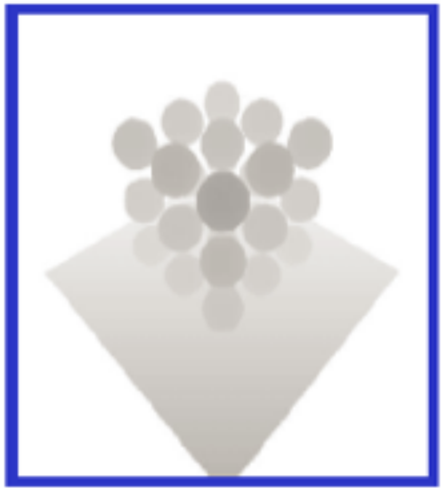
FYI: from the light's point-of-view again

source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

- Projecting the depth map onto the eye's view



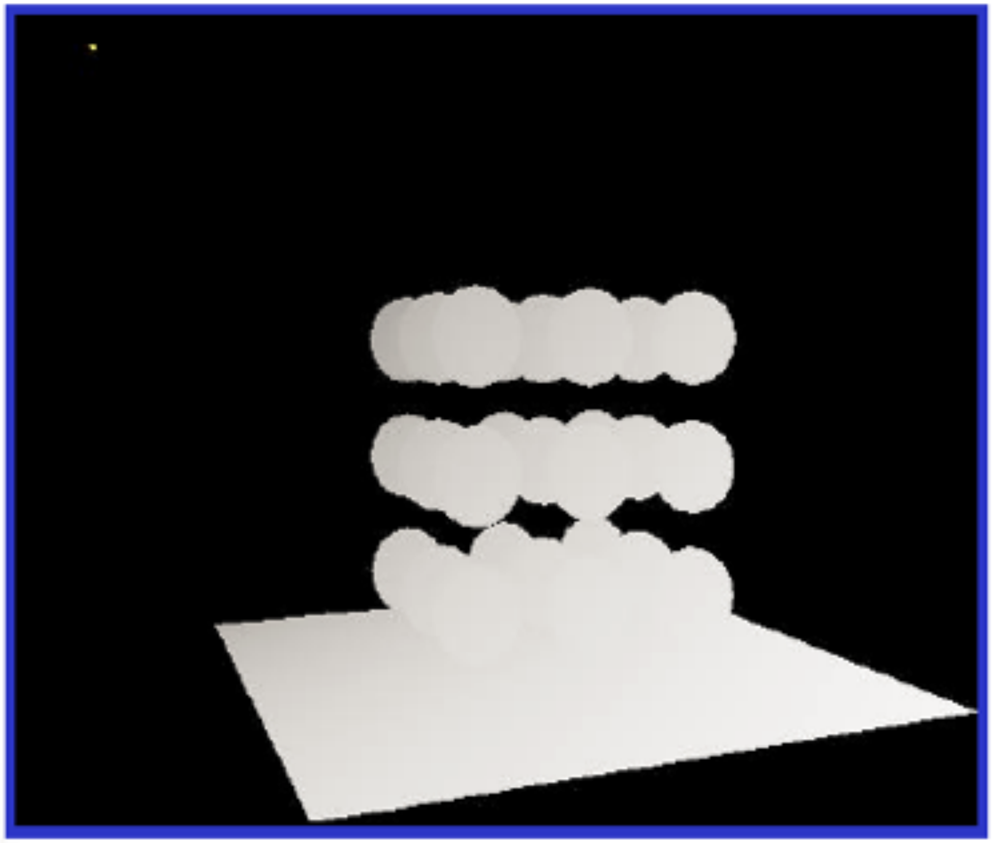


FYI: depth map for light's point-of-view again

source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

- Projecting light's planar distance onto eye's view

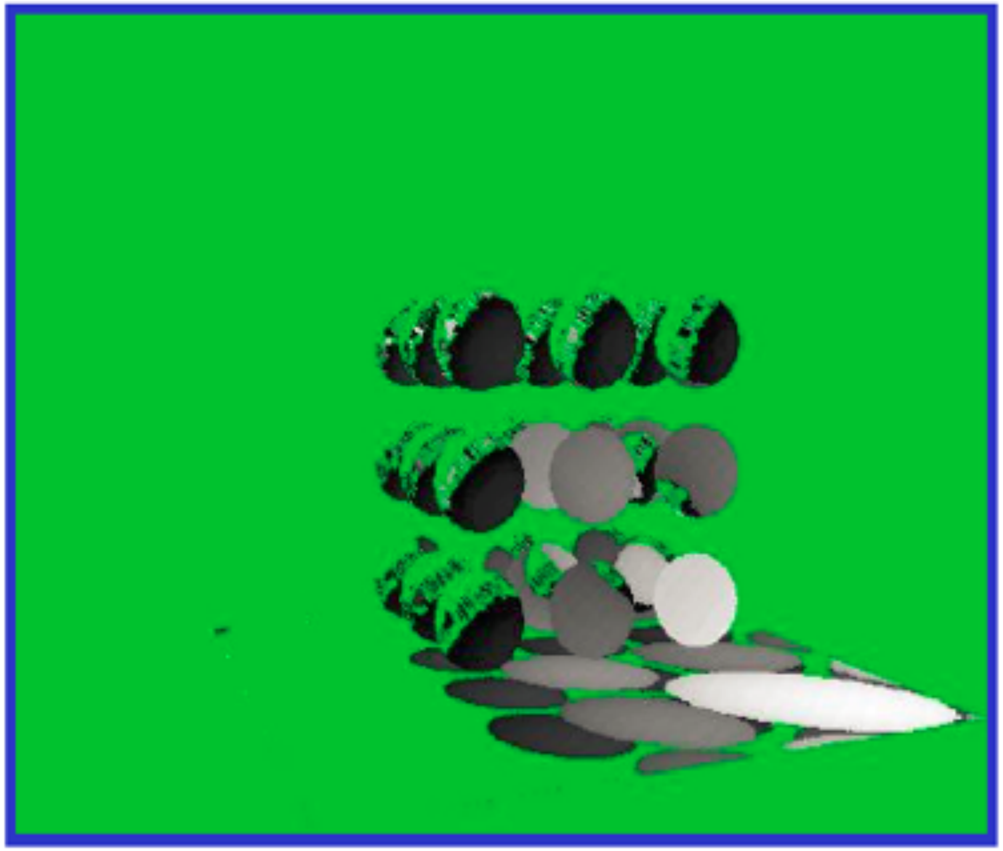


source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

- **Comparing light distance to light depth map**

Green is where the light planar distance and the light depth map are approximatel y equal



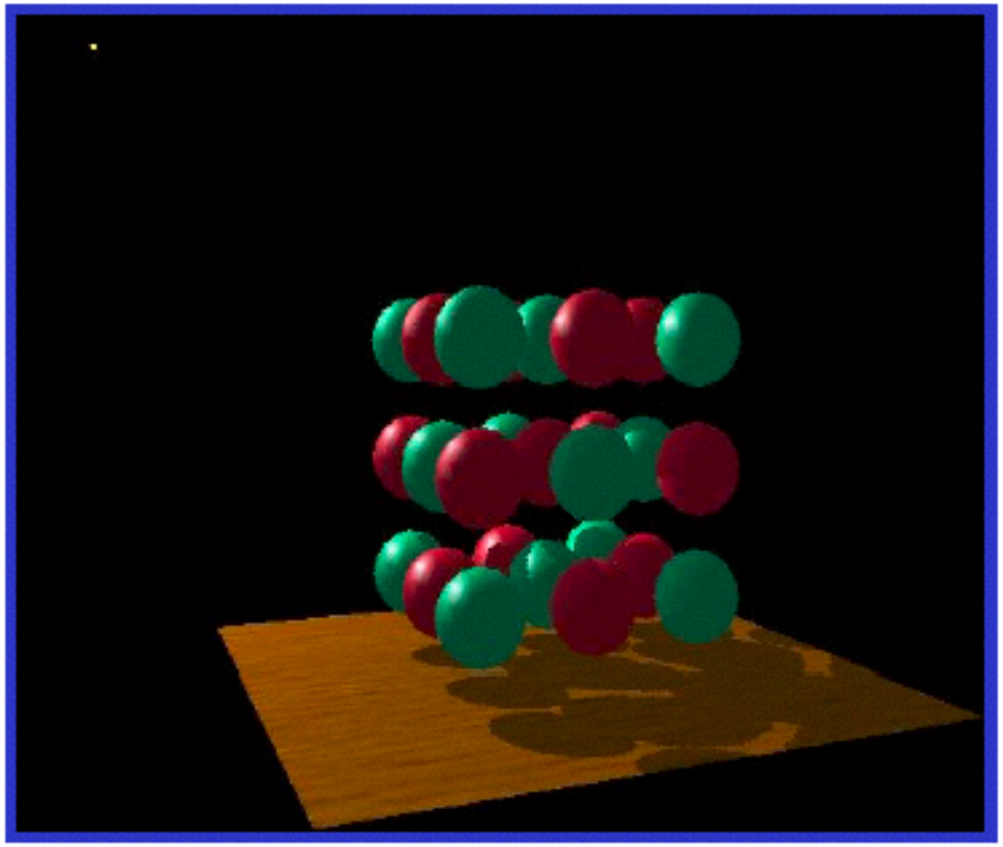
Non-green is where shadows should be

source: Mark Killgard, NVIDIA, GDC, 2001

Visualizing Shadow Mapping

- Scene with shadows

Notice how specular highlights never appear in shadows



Notice how curved surfaces cast shadows on each other

source: Mark Killgard, NVIDIA, GDC, 2001

Pixar: Luxo, Jr.

- Luxo Jr. has 2 animated lights and 1 overhead light
  - three shadow maps dynamically generated per frame, complex geometry, done in OpenGL