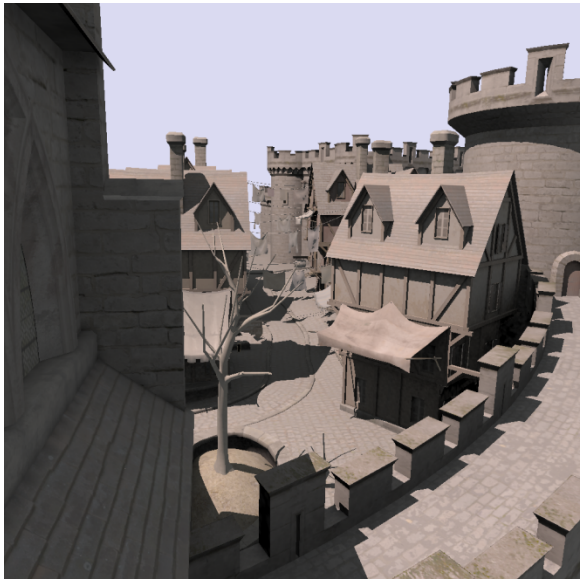
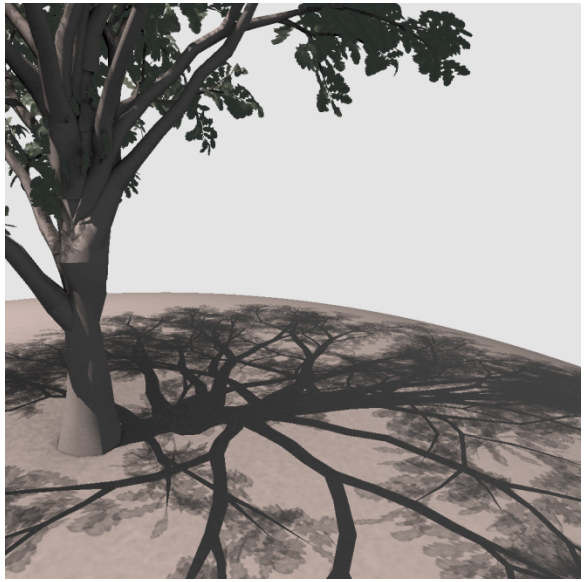
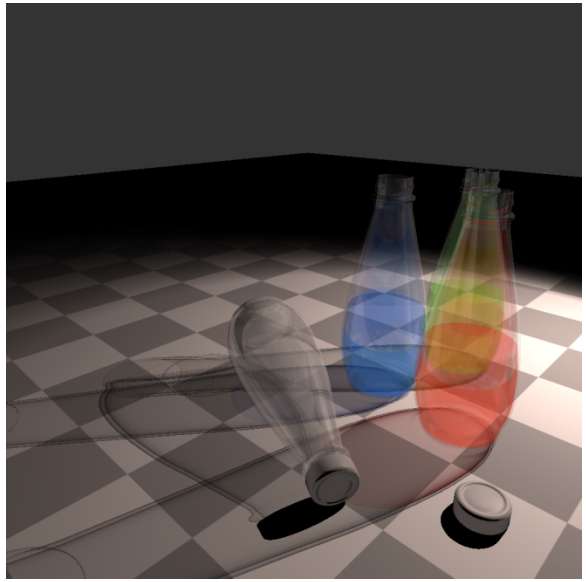
CENG 477
Introduction to Computer Graphics

Shadows in Forward Rendering

Shadows

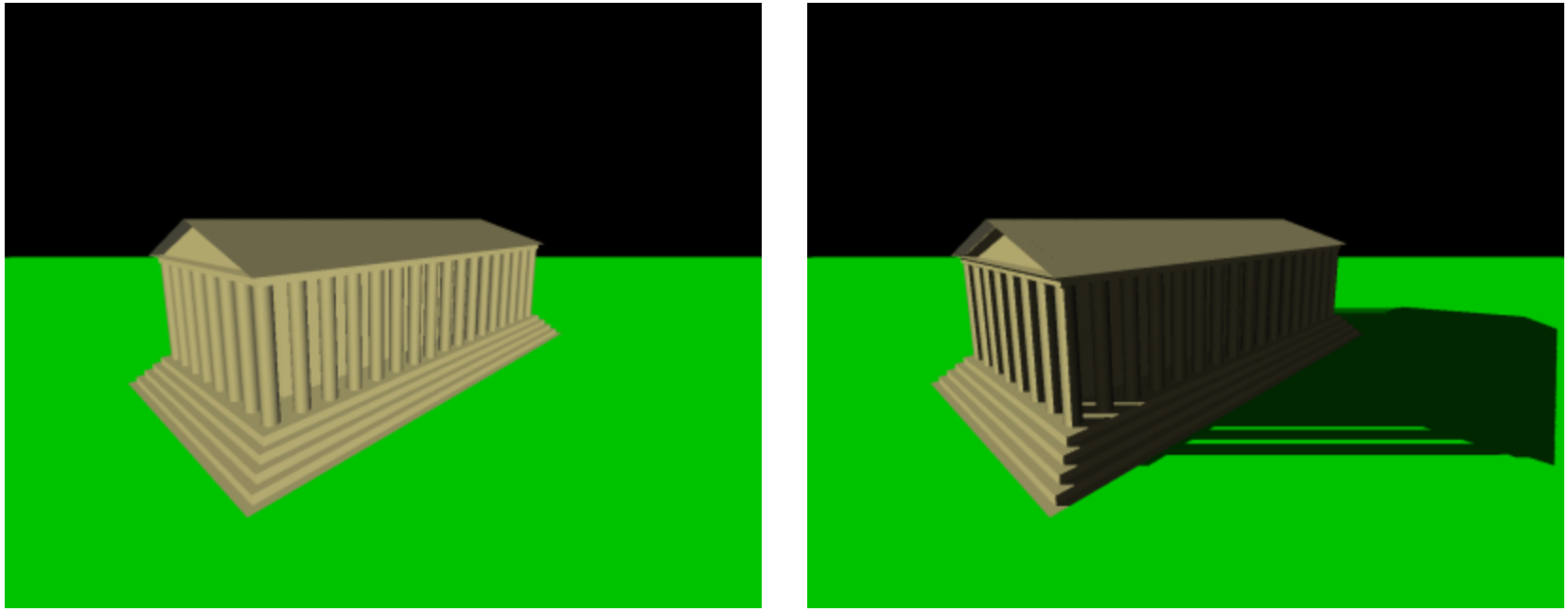
- Shadows are an important element of visual realism

From Sintorn et al. – Siggraph Asia 2012

Shadows

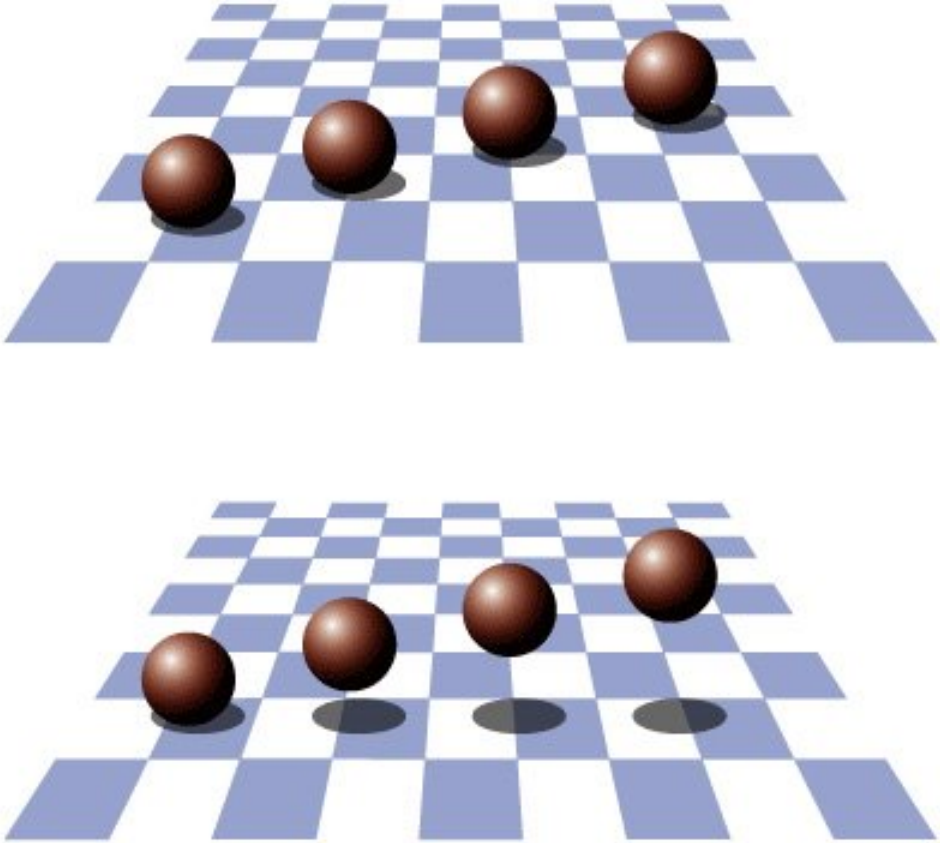
- Shadows give important cues about light positions



From wikipedia.com

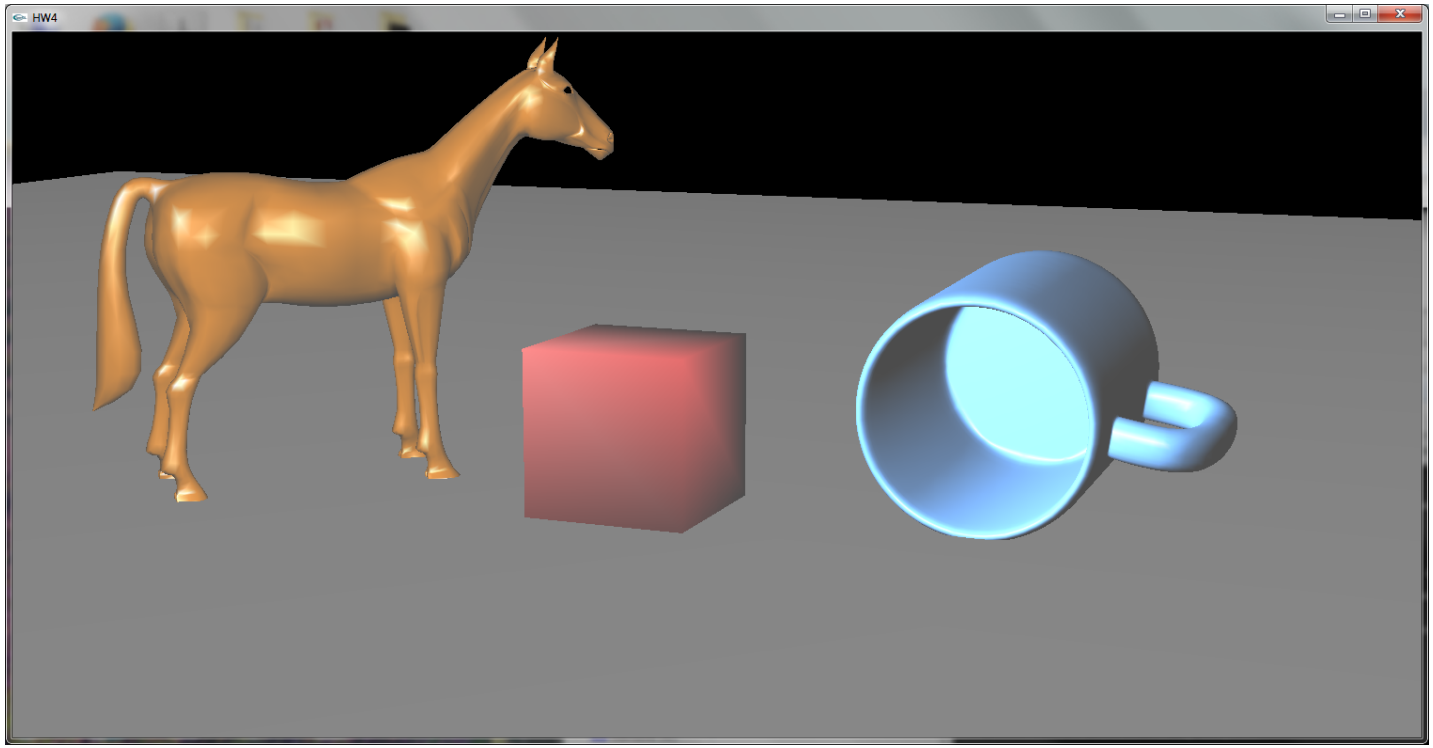
Shadows

- Shadows also give cues about object positions



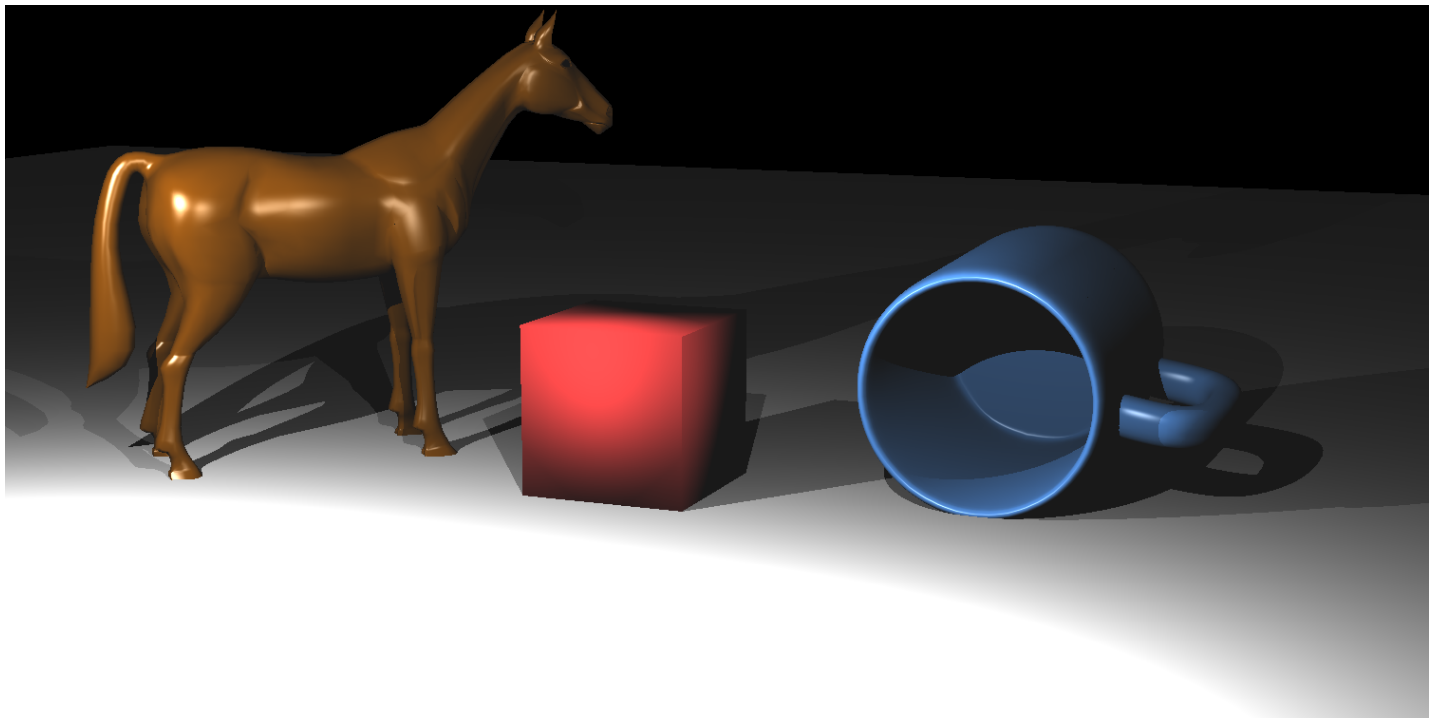
Shadows in OpenGL

- OpenGL does not have built-in support for shadows



Shadows in OpenGL

- Compare this to a ray traced image:



Shadows in OpenGL

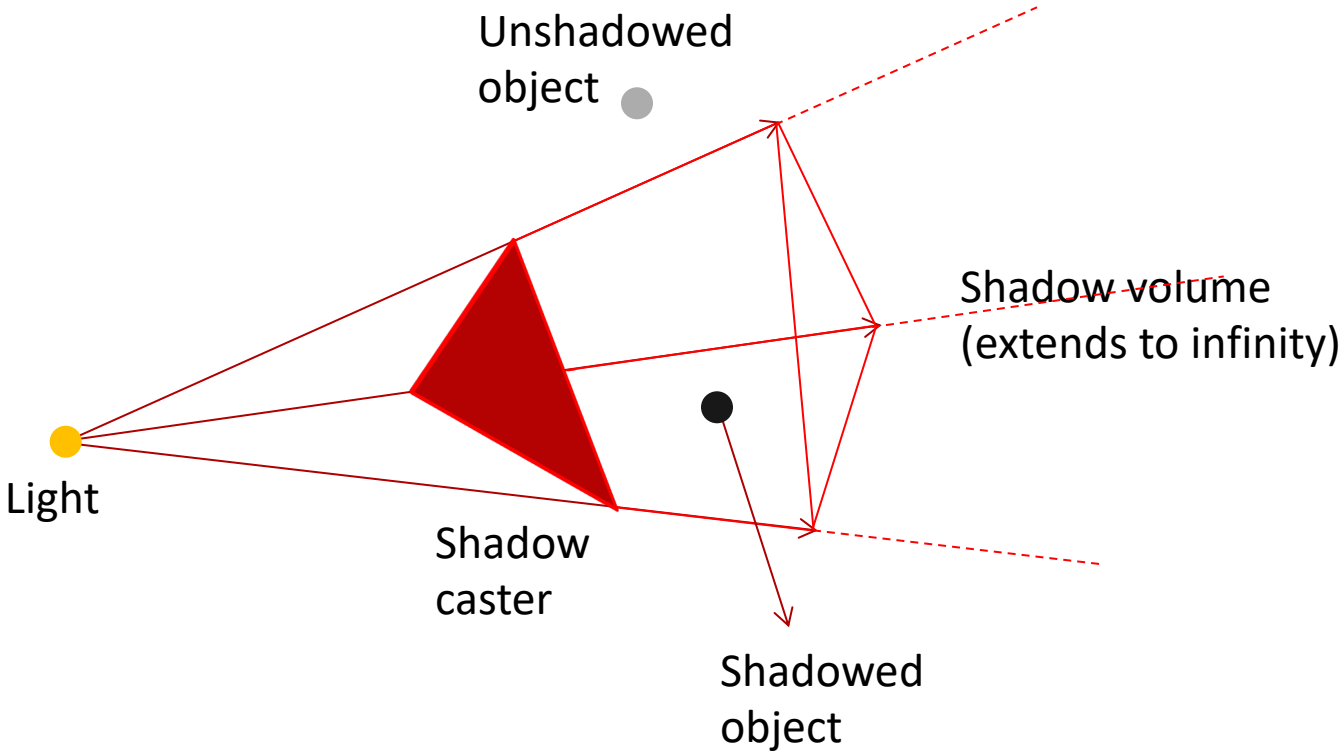
- OpenGL does not natively support generating shadows (neither does D3D)
  - That is, it does not have a function like glMakeShadow()!
- But, several shadowing algorithms can easily be implemented using features of OpenGL (or D3D)
  - Stencil buffer
  - Depth buffer
  - Render to texture
  - …

Generating Shadows

- Two algorithms that are commonly used are:
  - Shadow volumes (Crow, 1977)
  - Shadow mapping (Williams, 1978)
- Both algorithm has advantages and disadvantages and many variants
- Still an active research area:
  - An improved shadow volume algorithm presented at Siggraph Asia 2011: An Efficient Alias-free Shadow Algorithm for Opaque and Transparent Objects using per-triangle Shadow Volumes, by Sintorn et al.
- We'll study the basic versions of these algorithms

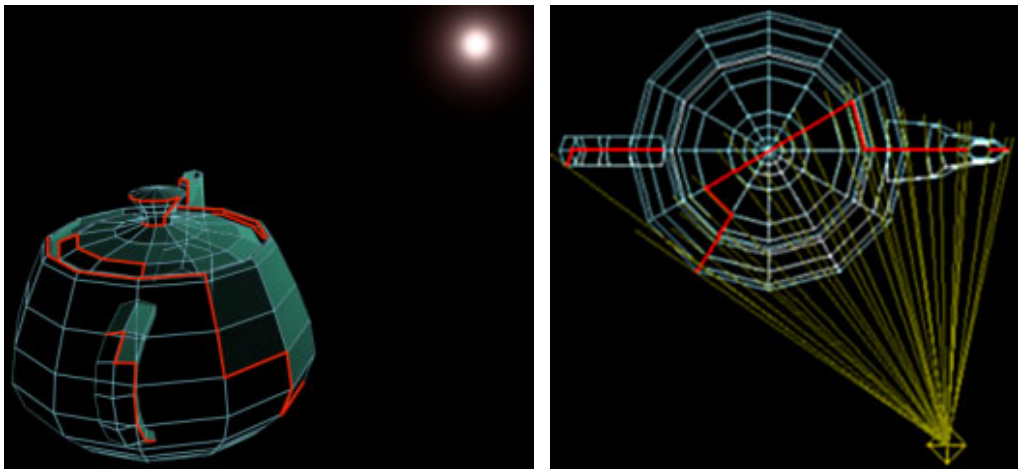
Shadow Volumes

- The idea is to create a 3D shape that represents the shadow that is casted by an object



Shadow Volumes

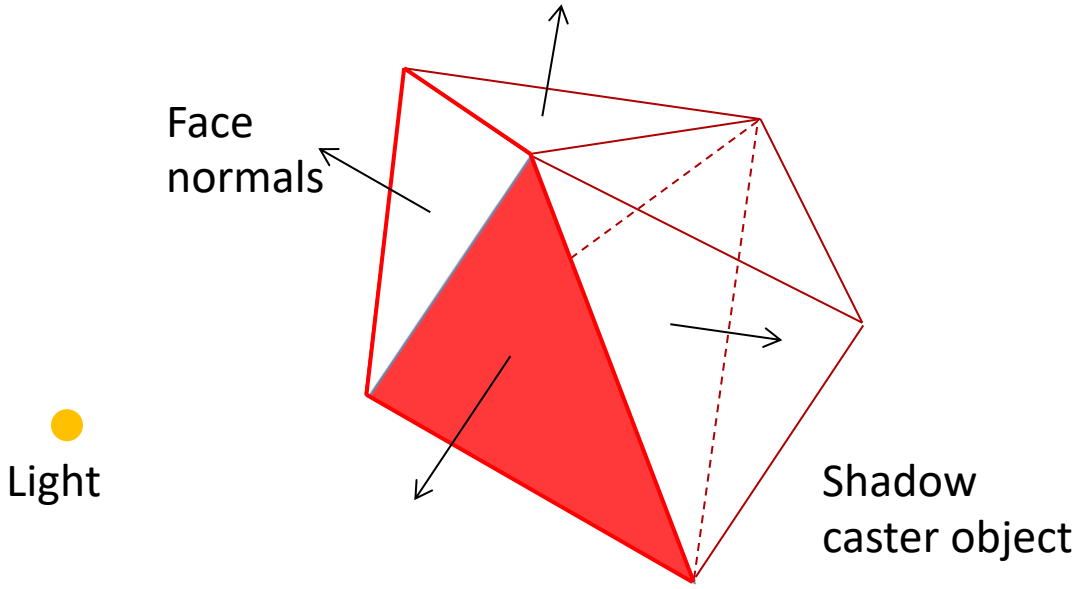
- A shadow volume can be created for any arbitrary object.
- We need to determine the contour edges (silhouette-edges) of the object as seen from the light source
- A contour edge has one adjacent polygon facing the light source and the other away from the light source



From John Tsiombikas

Contour Edges

- A contour edge has one adjacent polygon facing the light source and the other away from the light source
- We can use dot product to decide whether a face is toward or away from the light source



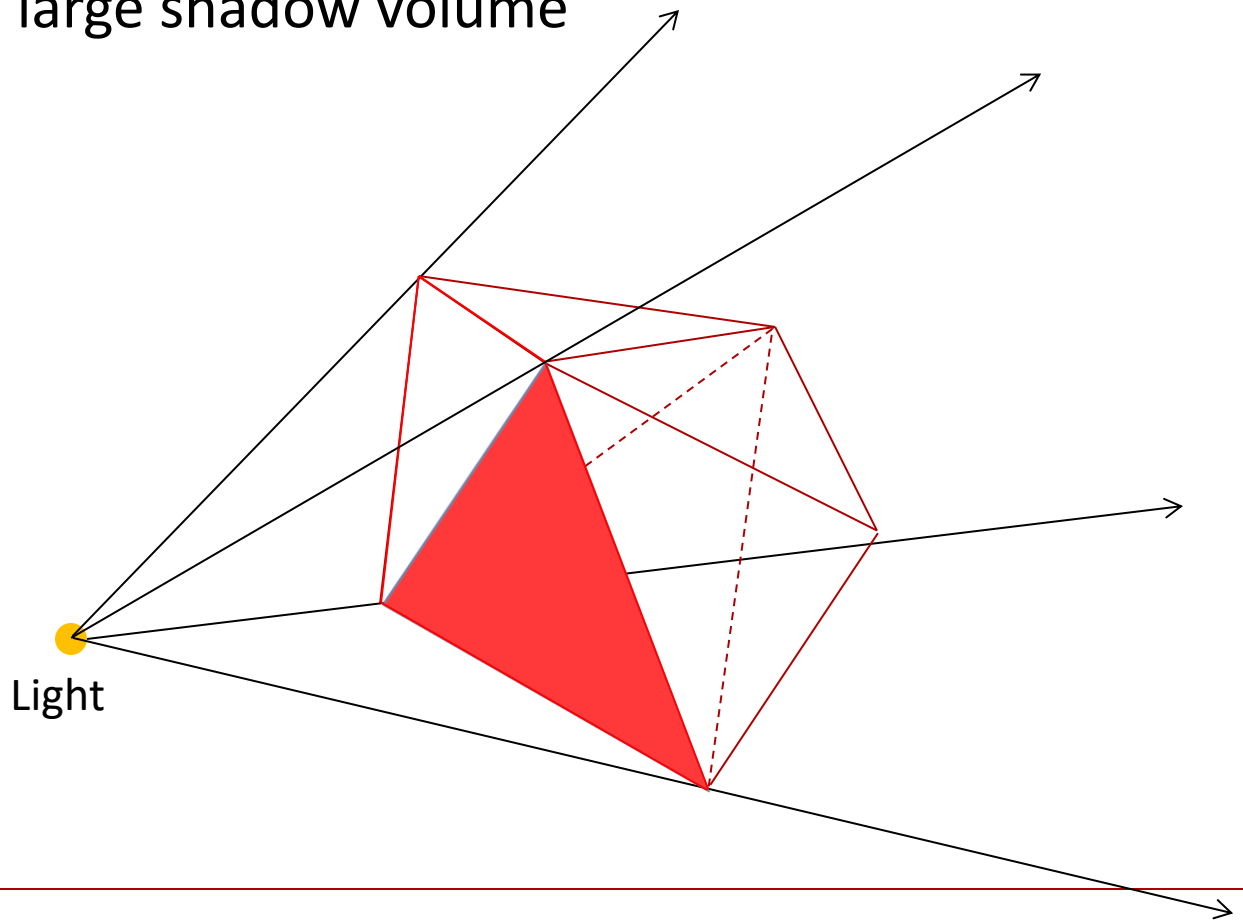
Contour Edges

```
// transform the light to the coordinate system of the object
LightPosition = Inverse(ObjectWorldMatrix) * LightPosition;
for (every polygon) {
   IncidentLightDir = AveragePolyPosition - LightPosition;
   // if the polygon faces away from the light source....
   if (DotProduct(IncidentLightDir, PolygonNormal) >= 0.0) {
      for (every edge of the polygon) {
         if (the edge is already in the contour edge list) {
            // then it can't be a contour edge since it is
            // referenced by two triangles that are facing
            // away from the light
            remove the existing edge from the contour list;
         } else {
            add the edge to the contour list;
         }
      }
   }
}
```

From John Tsiombikas

Extruding Contour Edges

- Once the contours are found, we need to extrude them to create a large shadow volume



Extruding Contour Edges

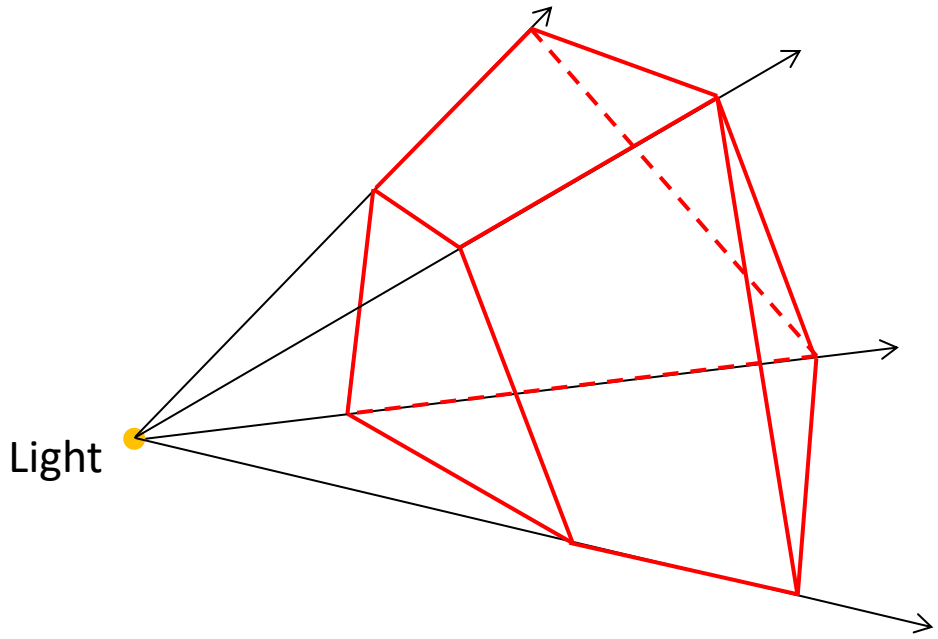
- Extrusion amount should be large enough to cover all objects which can receive shadow from this light source

```
ExtrudeMagnitude = A_BIG_NUMBER;
for (every edge) {
   ShadowQuad[i].vertex[0] = edge[i].vertex[0];
   ShadowQuad[i].vertex[1] = edge[i].vertex[1];
   ShadowQuad[i].vertex[2] = edge[i].vertex[1] + ExtrudeMagnitude *
      (edge[i].vertex[1] - LightPosition);
   ShadowQuad[i].vertex[3] = edge[i].vertex[0] + ExtrudeMagnitude *
      (edge[i].vertex[0] - LightPosition);
}
```

From John Tsiombikas

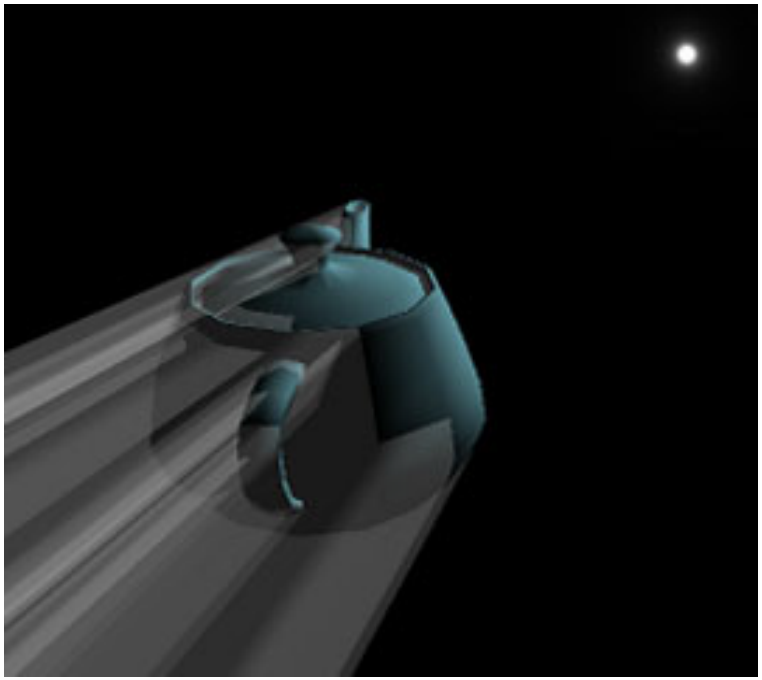
Extruding Contour Edges

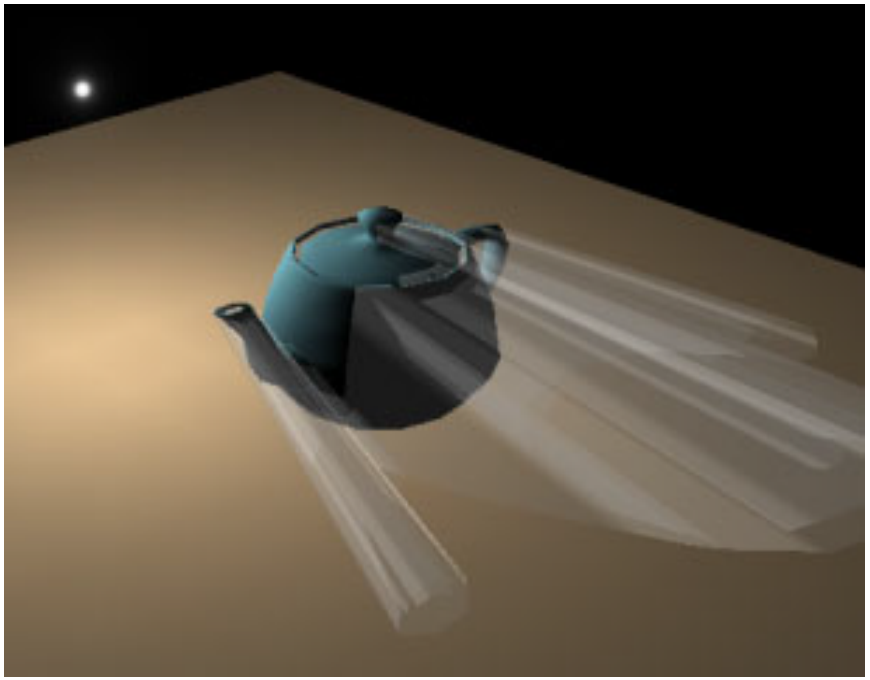
- The shadow caster object's contour vertices serve as the *cap* of the shadow volume.
- The *bottom* can also be capped to obtain a closed volume.



Extruding Contour Edges

- This is how it looks like for a complex object

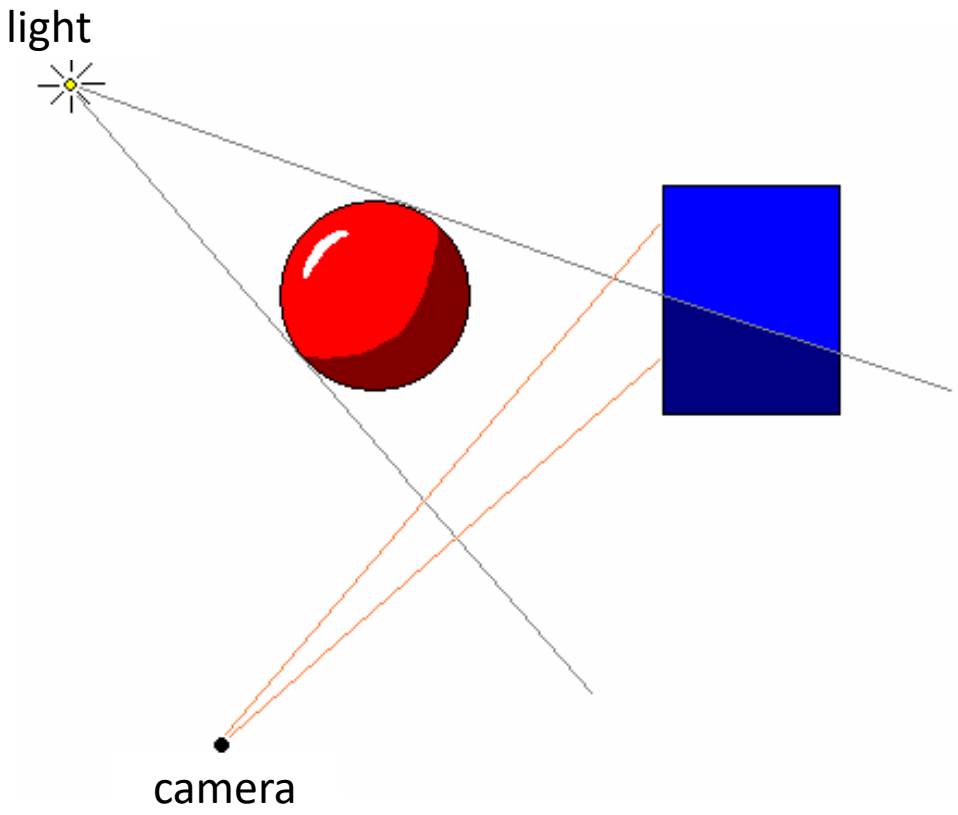




From John Tsiombikas

Rendering Shadows

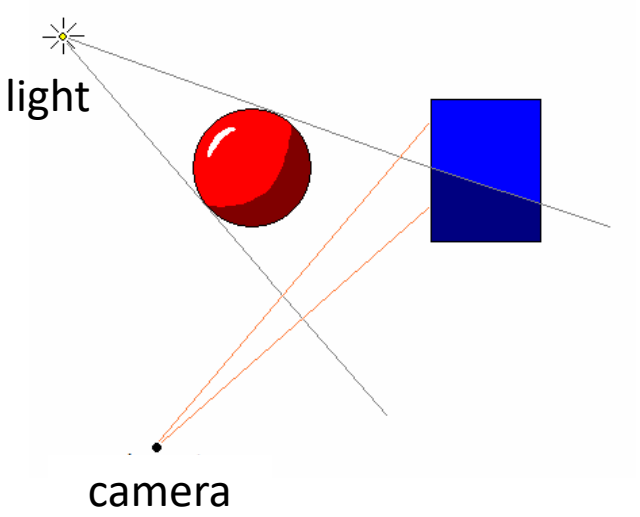
- Now what? Any ideas about how we can proceed?



From John Tsiombikas

Rendering Shadows

- Assume a ray originating from the eye
- If it enters the shadow volume from a front face and exits from a back face, the point it reaches is not in shadow
- However, if it does not exit before hitting the object, the point should be in shadow



Rendering Shadows

- But we are not ray tracing! How can we know if the ray enters or exits?
- This is where depth and stencil buffers come in handy
- Stencil buffer:
  - An integer buffer that stores a value for every pixel (usually an 8-bit value)
  - We can clear it to any value that we want (glClearStencil(int) and glClear(GL_STENCIL_BUFFER_BIT))
  - Has operations such as increment and decrement based on the result of the depth test (glStencilOp(sfail, dfail, dpass))
- Think of stencil buffer as a counter buffer, which keeps a counter for every pixel

Shadow Volume Algorithm (Part I)

- Clear the depth (to 1.0) and stencil buffer (to 0)
- Enable depth testing and disable stencil testing
- Render the scene with ambient light (note that ambient light does not produce shadows). This updates the depth buffer value for every pixel that corresponds to an object
- Disable writing to the depth buffer

Color Buffer

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	100	100	100	100	0	0
0	0	100	100	100	100	0	0
0	0	100	100	100	100	0	0
0	0	100	100	100	100	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Depth Buffer

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	0.5	0.5	0.5	0.5	1.0	1.0
1.0	1.0	0.5	0.5	0.5	0.5	1.0	1.0
1.0	1.0	0.5	0.5	0.5	0.5	1.0	1.0
1.0	1.0	0.5	0.5	0.5	0.5	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Stencil Buffer

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Shadow Volume Algorithm (Part II)

- Draw the front faces of the shadow volume. Increment the stencil value for every pixel that passes the depth test (glStencilOp(GL_KEEP, GL_KEEP, **GL_INCR**))

Stencil Buffer – Front Pass

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Stencil value for fragments behind the shadow volume's front faces' will be incremented

Shadow Volume Algorithm (Part II)

- Draw the back faces of the shadow volume. Decrement the stencil value for every pixel that passes the depth test (glStencilOp(GL_KEEP, GL_KEEP, **GL_DECR**))

Stencil Buffer – Back Pass

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Stencil value for fragments behind the shadow volume's back faces' will be decremented

Fragments outside the volume will have zero stencil value

Shadow Volume Algorithm (Part III)

- Enable stencil testing such that pixels whose stencil value is zero will be rendered
- Enable writing to the depth buffer and clear it
- Enable the point light source
- Enable additive blending so the contribution of passing pixels will be added to the previous ambient values (glBlendFunc(GL_ONE, GL_ONE))

Color Buffer

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	150	150	150	150	0	0
0	0	150	150	150	150	0	0
0	0	100	100	100	100	0	0
0	0	100	100	100	100	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Depth Buffer

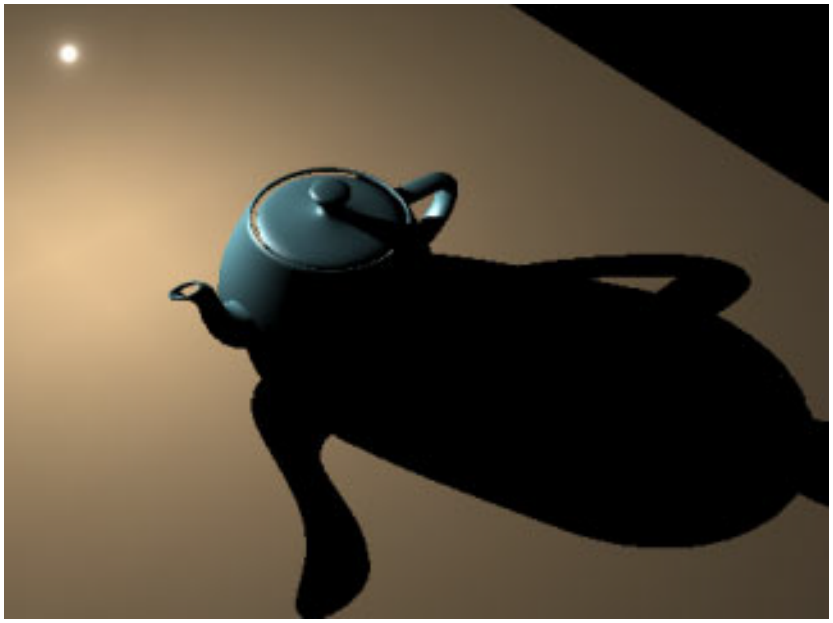
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	0.5	0.5	0.5	0.5	1.0	1.0
1.0	1.0	0.5	0.5	0.5	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Stencil Buffer

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Shadow Volume Algorithm

- No blending versus blending:



From John Tsiombikas

Used Applications

- Doom 3 is the most well-known example (source code available at: https://github.com/id-Software/DOOM-3-BFG)

From http://http.developer.nvidia.com/GPUGems/gpugems_ch09.html

Pros and Cons

- Requires preprocessing of the scene geometry to create the shadow volume
- Needs to be updated if lights and/or objects move
- Can be time consuming for complex geometry
- Requires multiple rendering passes for each frame
  - Ambient pass
  - For each light:
    - Compute SV
    - SV front-face pass
    - SV back-face pass
    - Final light source pass
- No need to update shadow volume if only the camera moves

Pros and Cons

- Inaccurate models can cause leaking artifacts

From http://http.developer.nvidia.com/GPUGems/gpugems_ch09.html

Pros and Cons

- For speed-up, two versions of objects can be created:
  - High polygon version is used for actual rendering of the object
  - Low polygon version is used to cast shadows of the object

From wikipedia.com

Questions

- How can the shadow volume algorithm be extended to deal with:
  - Camera inside the shadow volume?
  - Multiple light sources and multiple shadow casters?
  - Transparent shadow casters?
- Further reading:
  - http://http.developer.nvidia.com/GPUGems/gpugems_ch09.html
  - http://www.angelfire.com/games5/duktroa/RealTimeShadowTutorial.htm

Shadow Mapping

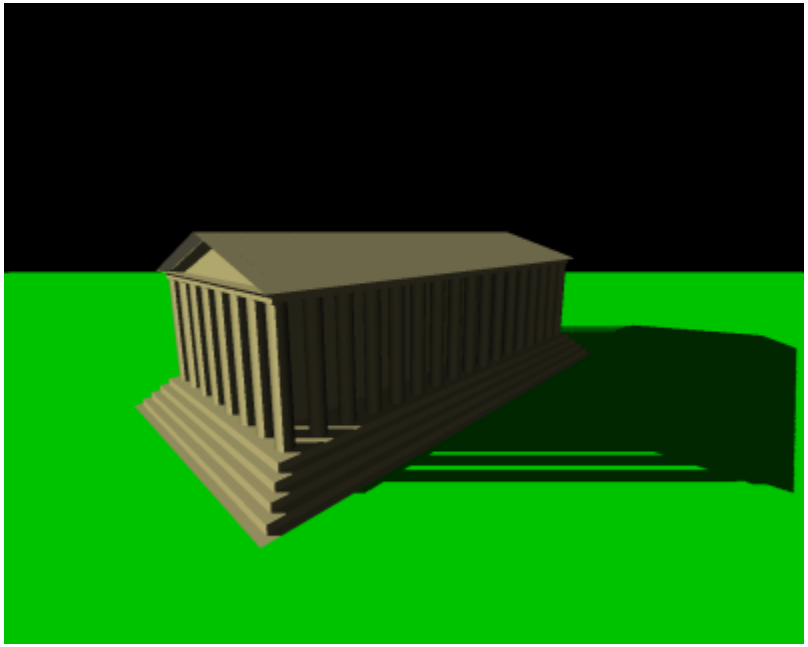
- The idea introduced by Lance Williams in his 1978 paper "Casting Curved Shadows on Curved Surfaces"
- Image space technique
- Advantages:
  - No knowledge or processing of scene geometry is required
  - No need to use stencil buffer
  - Fewer rendering passes than shadow volumes
- Disadvantages:
  - Still requires multiple passes
  - Aliasing artifacts may occur (shadows may look jaggy)

Shadow Mapping

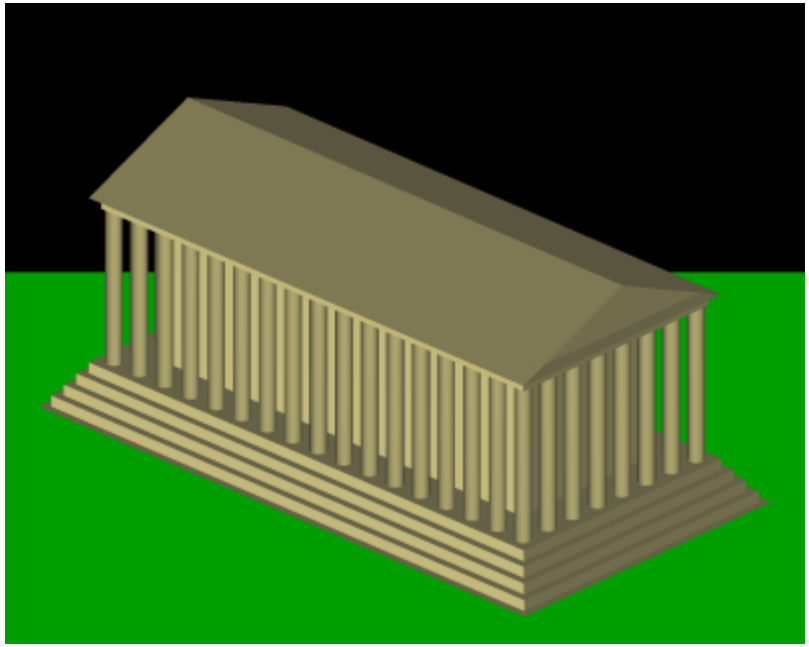
- **Part I:** Render the scene from the point of view of the light source (as if the light source was a camera)
  - Objects that are not visible are in shadow with respect to that light source
- **Part II:** Determine whether an object as seen from the camera is in shadow in the “light’s view”

Part I: Rendering from the Light Source

- Pretend that there is a camera at the light position
- Use perspective projection for spot (and point) lights
- Use orthographic projection for directional lights



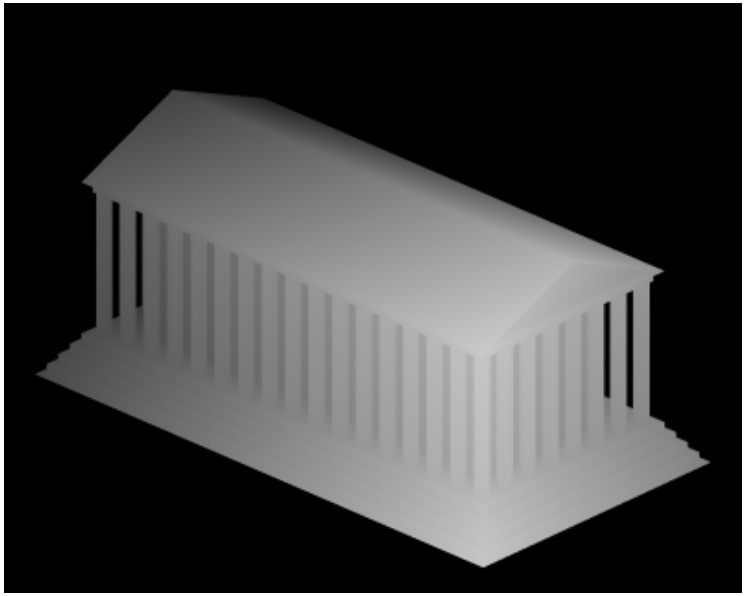
Original camera view



Light source view

Part I: Extracting the Depth Map

- Actually, we only need the depth buffer values from the light source view
- Save this depth buffer to an off-screen texture (FBOs)



Depth map from the light's view

- *1 – z* is shown for visualization purposes
- Normally, *z* is larger for further away points

Part II: Rendering from the Camera

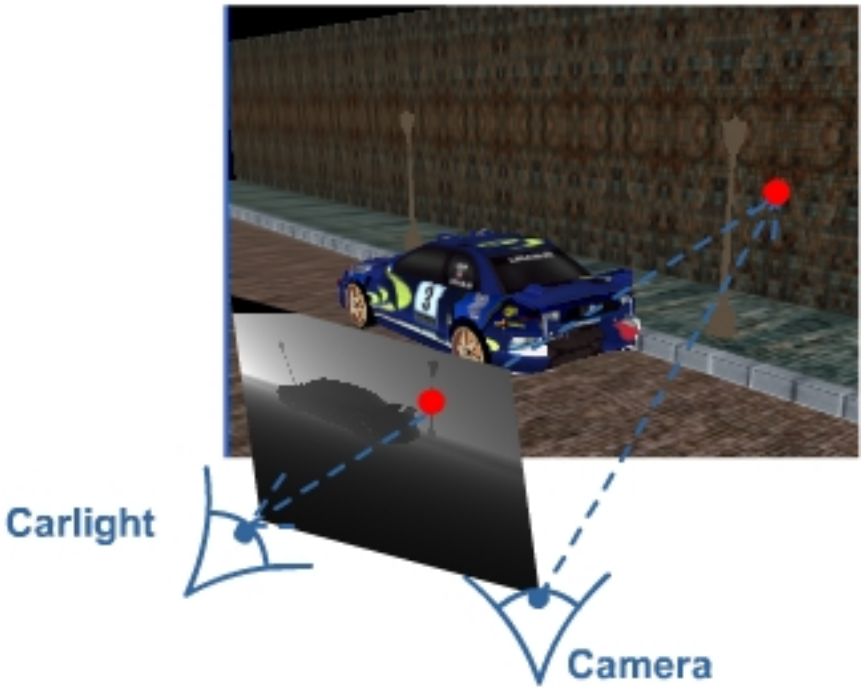
- Render the scene from the camera view as usual
- For every pixel, compare the depth value of that pixel w.r.t. the light source (**R**) to the stored value in the depth texture (**D**):

  R = D: Object was directly visible from the light source
  R > D: Object was behind another object in the light's view

- But there is a problem:
  - Pixel (i, j) in the camera view will not belong to the same object as pixel (i, j) in the light view as they look at the scene from different positions (and with different orientations)

Projective Texturing

- For every pixel in the camera view, we need to find the corresponding pixel in the light's view



From http://www.riemers.net/images/Tutorials/DirectX/Csharp/Series3

Projective Texturing

- Assume **inPos** represents the world coordinates of an object. Its camera coordinates are computed by:

outPosCamera = cameraWorldViewProjection * **inPos**;

- Its light view coordinates are computed by:

outPosLight = lightWorldViewProjection * **inPos**;

- We can use **outPosLight** to look up the depth texture we generated in Part I
- But this value is in range [-1,1] in all axis (CVV)

Projective Texturing - Bias

- Therefore, we need to multiply it with a *bias* matrix to bring all components to [0, 1] range:

$$bias = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

outPosLight = bias * lightWorldViewProjection * **inPos**;

Shadow Check

- Remember that **R** is the depth value of the pixel from the light's view and **D** is the stored depth in the texture:

  R = D: Object was directly visible from the light source
  R > D: Object was behind another object in the light's view

- **R** and **D** can be found by:

  R = **outputPosLight.z**
  D = texLookUp(shadowMap, **outputPosLight.xy**)

- Now we can perform the shadow check:

```
if (R > (D + 0.00001))
   Output.Color = Input.Color * 0.5;
else
   Output.Color = Input.Color;
```

Darkening factor

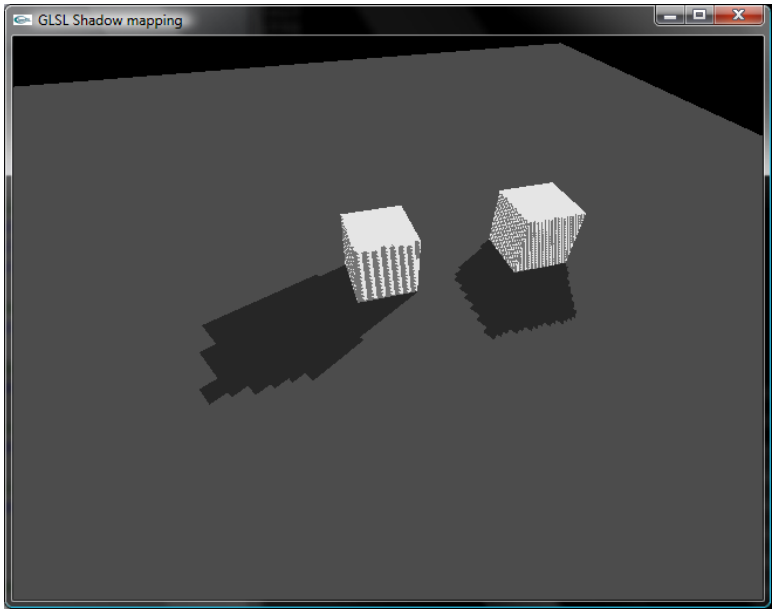
Note that a small bias
is added to avoid self-shadowing
just like as in ray tracing

Shadow Mapping

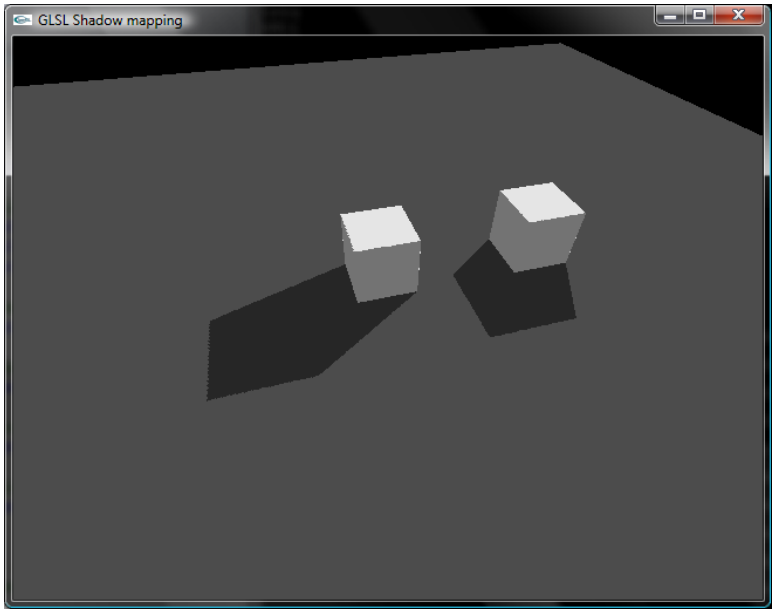
- More to read from:
  - Projective texturing:
    - http://developer.nvidia.com/object/Projective_Texture_Mapping.html
      http://en.wikipedia.org/wiki/Projective_texture_mapping
  - OpenGL fixed function implementation:
    - http://www.paulsprojects.net/tutorials/smt/smt.html
  - OpenGL shader (GLSL) implementation:
    - www.gamedev.net/community/forums/topic.asp?topic_id=316147
    - http://sombermoon.com/shadowmappingdoc.html

Importance of Shadow Map Resolution

- If the depth texture we create in Part I does not have enough resolution, we can see blocking artifacts:



160x120 shadow map

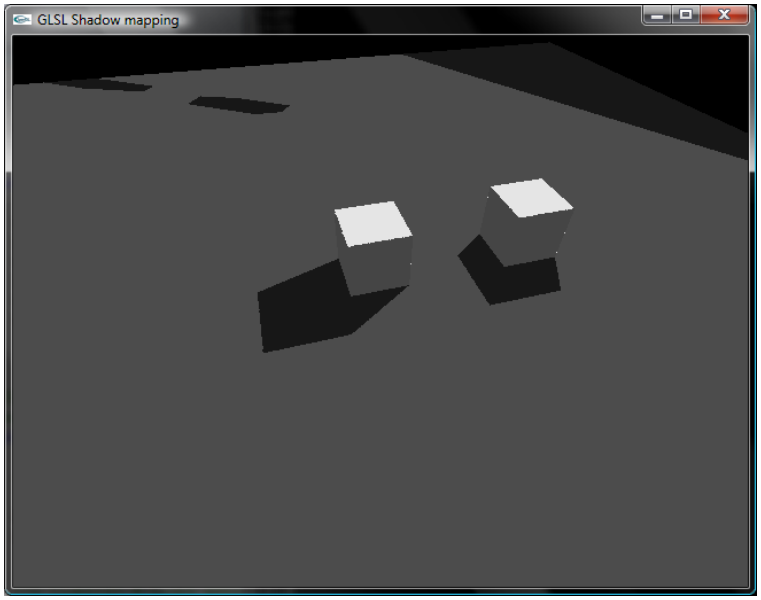


1280x960 shadow map

From http://bytechunk.net

Projection Artifacts

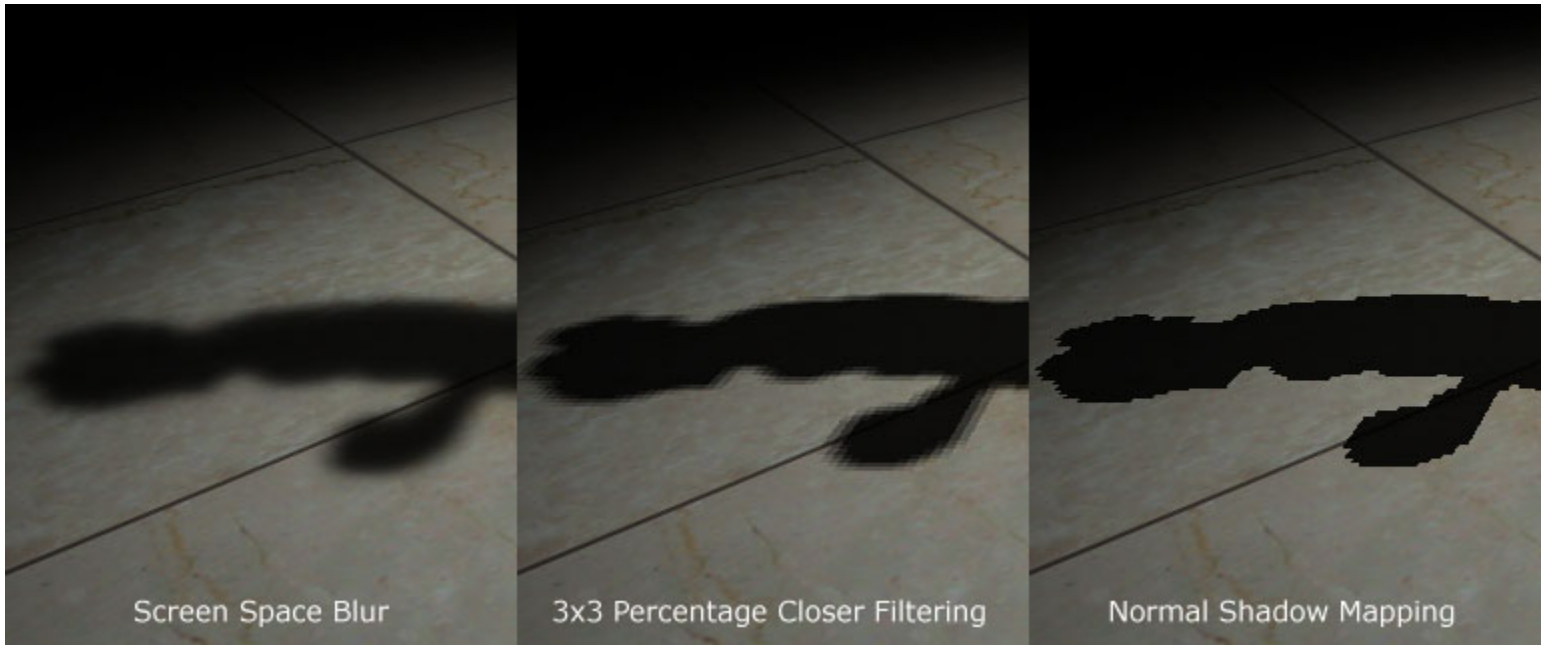
- If an object falls outside the viewing frustum of the light, we can see artifacts with a naïve implementation:



- For how to fix these, read more at:
  - http://bytechunk.net/shadowmapping/index.php

Improving the Shadow Quality

- Shadow map can be filtered in various ways to create soft shadows:



From http://www.gamedev.net

Applications Using Shadow Maps

- Most games use shadow mapping

Rage

Applications Using Shadow Maps

- Riddick 2: Assault on Dark Athena

From http://uk.ps3.ign.com

Applications Using Shadow Maps

- Dragon Age

Applications Using Shadow Maps

- Assassin's Creed