# CSE 167:
# Introduction to Computer Graphics
# Lecture #14: Shadows

Jürgen P. Schulze, Ph.D.
University of California, San Diego
Fall Quarter 2012

# Announcements

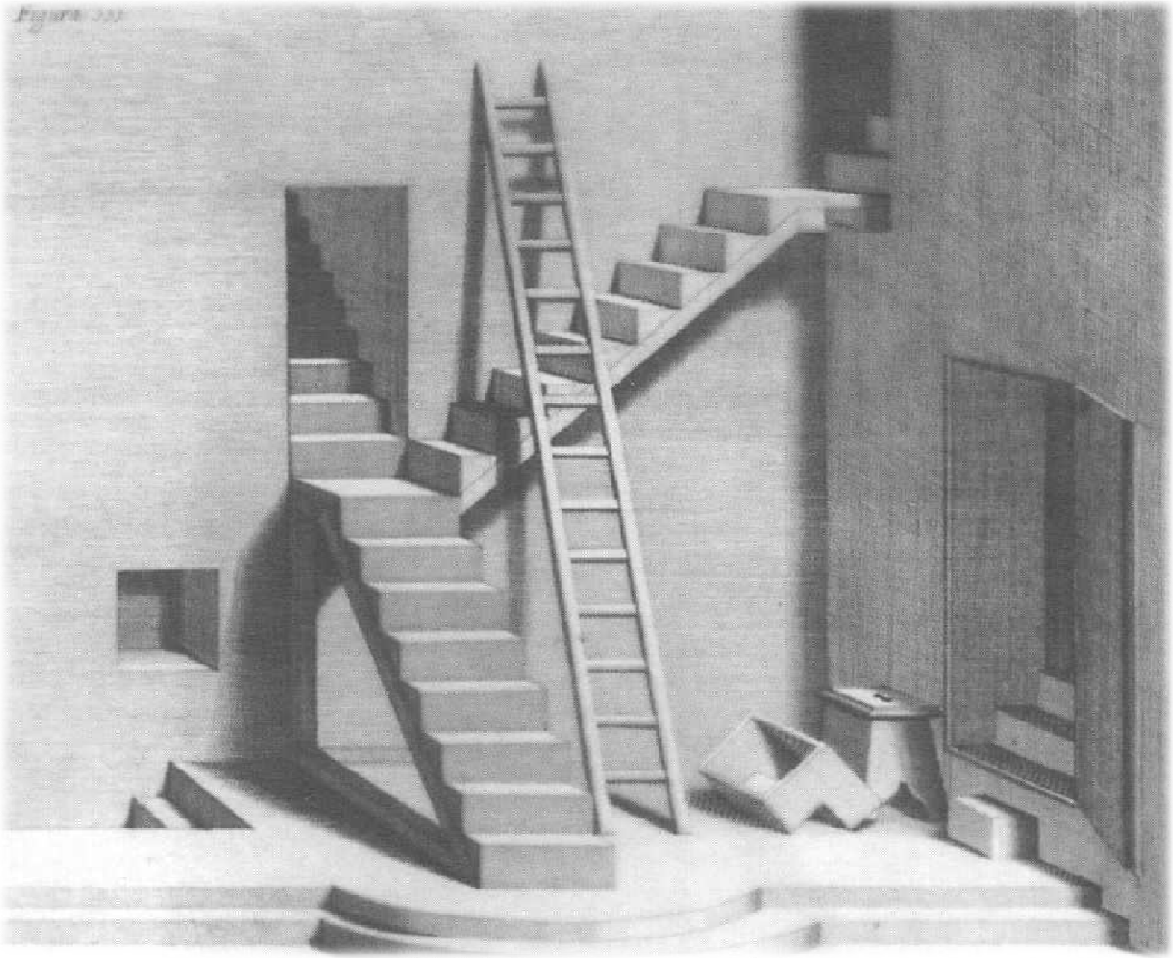
- Homework assignment #6 due tomorrow, Friday, Nov 16
  - Late submission deadline: Monday, Nov 2
- Final project description will be on-line tomorrow
  - Due on Thursday of finals week

# Lecture Overview

- Shadows
- Shadow Mapping
- Shadow Volumes

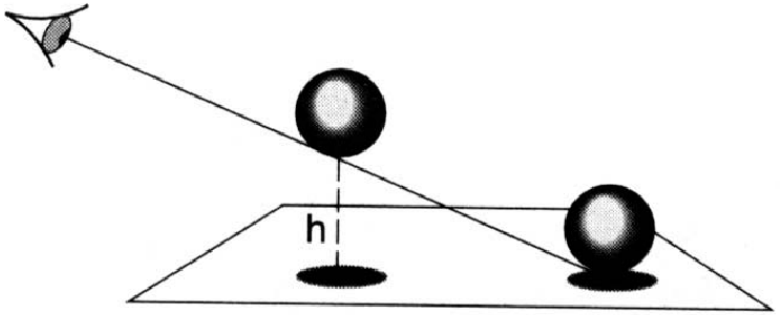
# Why Are Shadows Important?

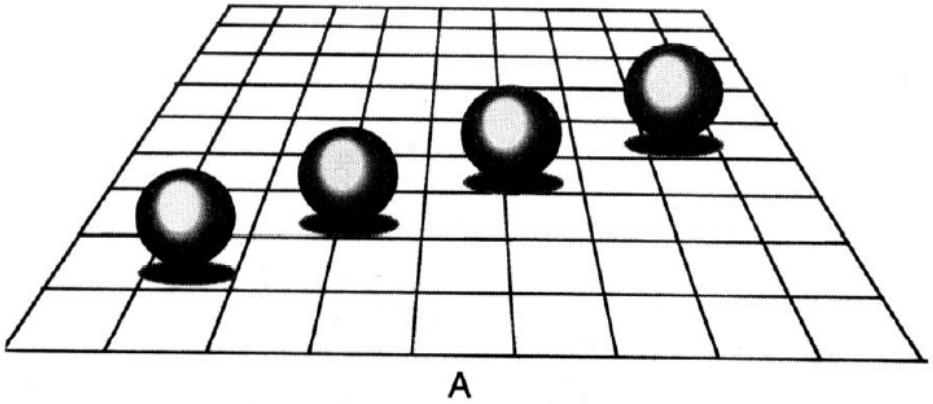
- Give additional cues on scene lighting

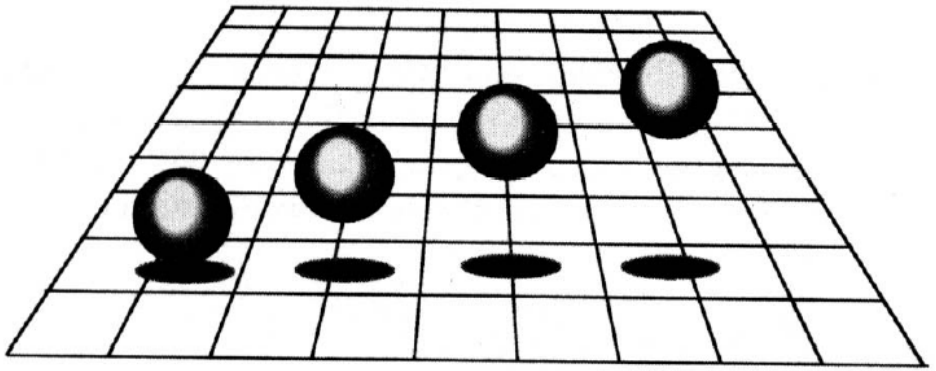


# Why Are Shadows Important?

- Contact points
- Depth cues

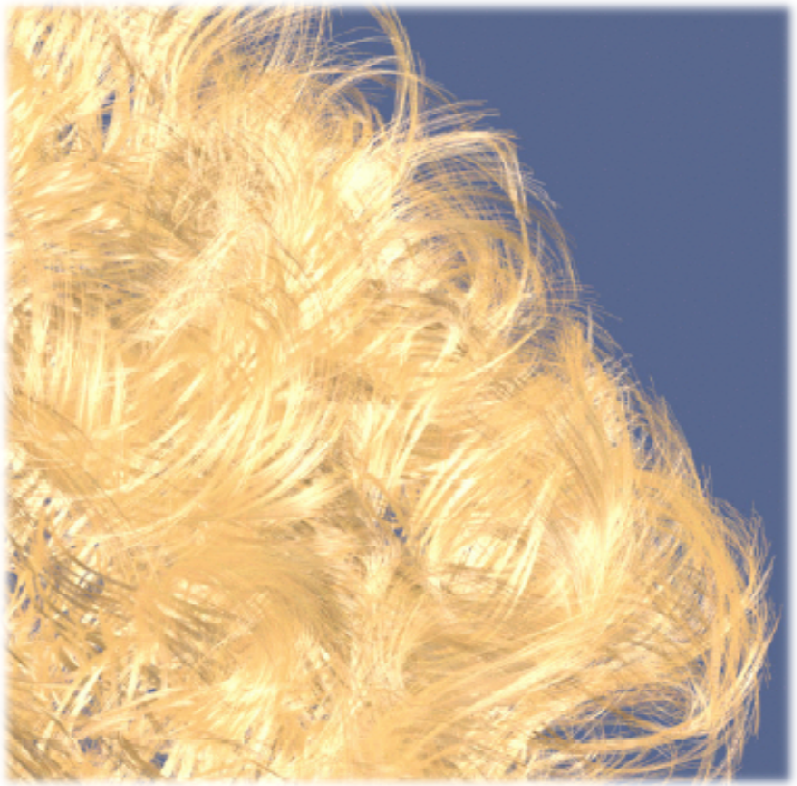


# Why Are Shadows Important?

- Realism

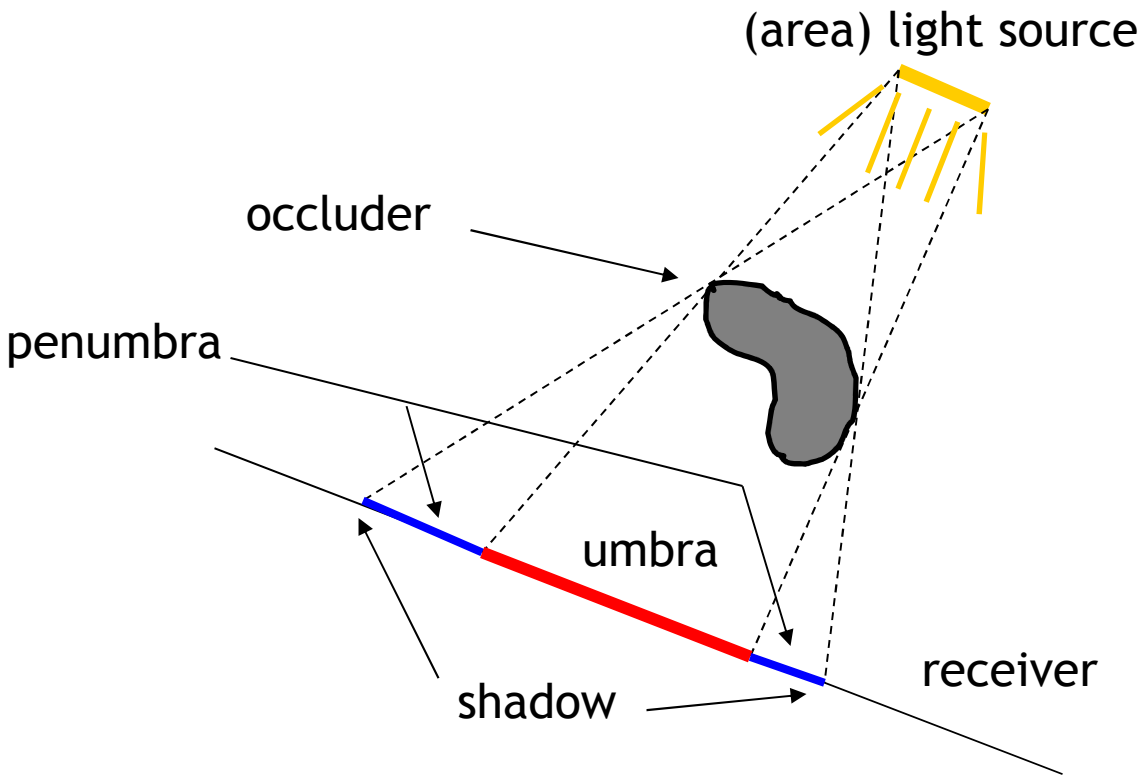


Without self-shadowing

With self-shadowing

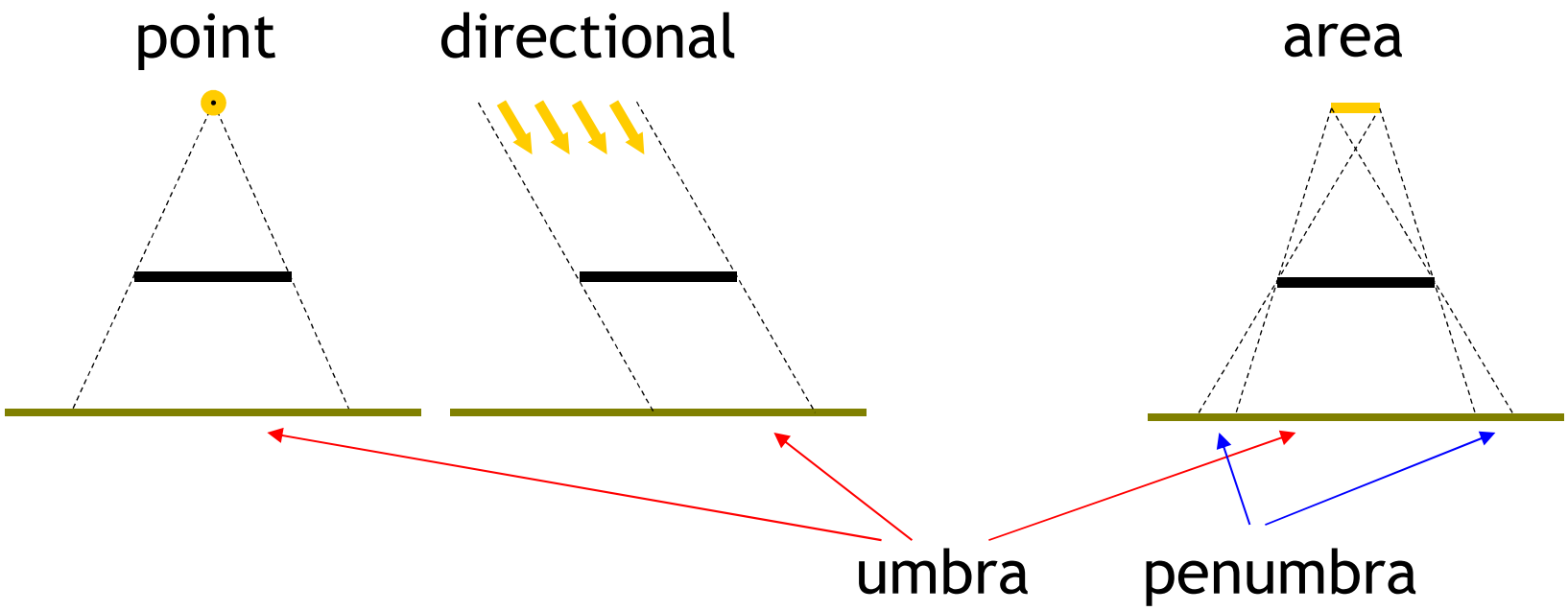
# Terminology

- Umbra: fully shadowed region
- Penumbra: partially shadowed region

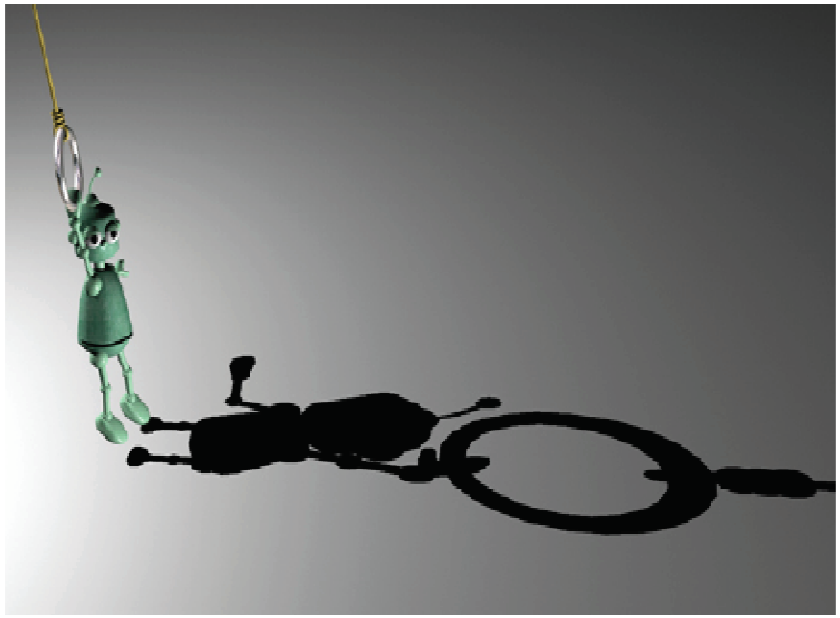


# Hard and Soft Shadows

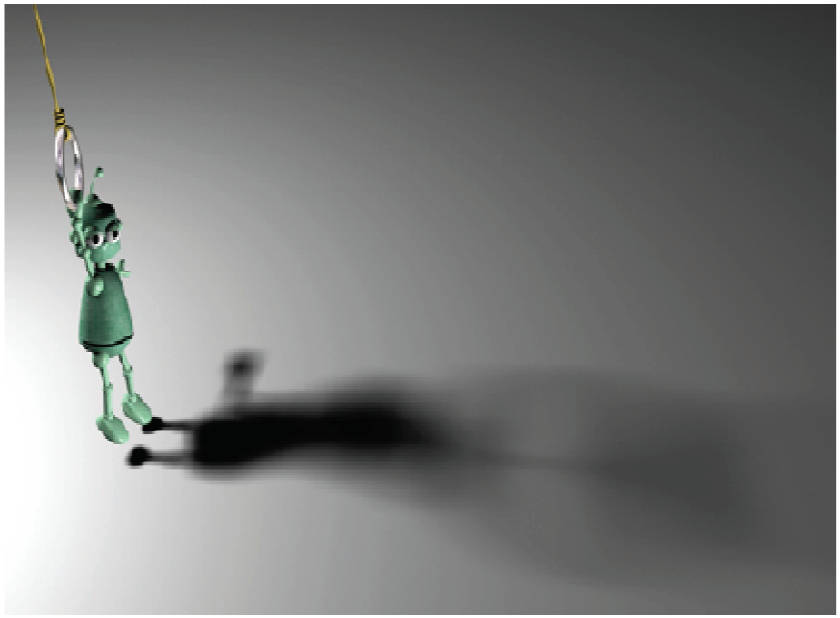
- Point and directional lights lead to hard shadows, no penumbra
- Area light sources lead to soft shadows, with penumbra



# Hard and Soft Shadows



Hard shadow from point light source



Soft shadow from area light source

# Shadows for Interactive Rendering

- In this course: hard shadows only
  - Soft shadows hard to compute in interactive graphics
- Two most popular techniques:
  - Shadow mapping
  - Shadow volumes
- Many variations, subtleties
- Active research area

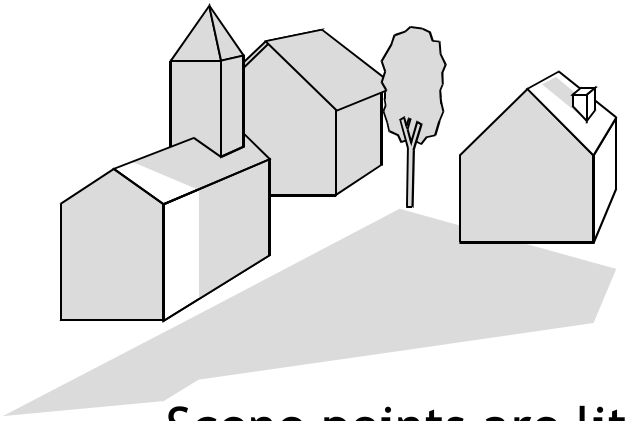
# Lecture Overview

- Shadows
- Shadow Mapping
- Shadow Volumes

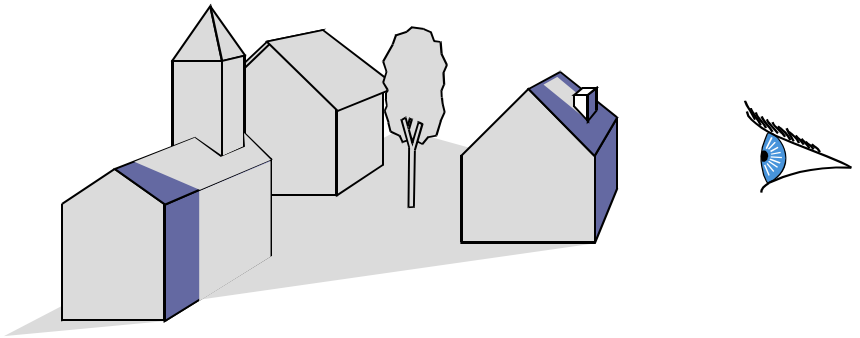
# Shadow Mapping

**Main Idea**

- A scene point is lit by the light source if visible from the light source
- Determine visibility from light source by placing a camera at the light source position and rendering the scene from there



Scene points are lit if visible from light source

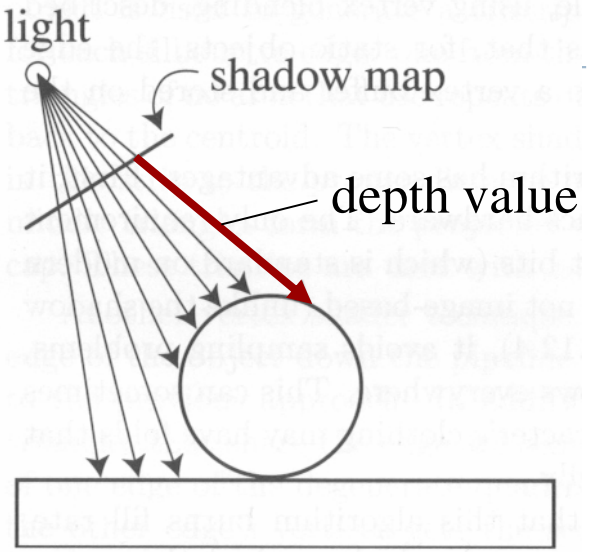


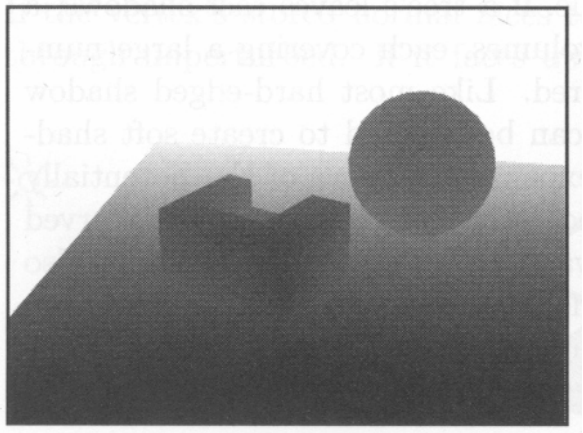
Determine visibility from light source by placing camera at light source position

# Two Pass Algorithm

**First Pass**

- Render scene by placing camera at light source position
- Store depth image (*shadow map*)



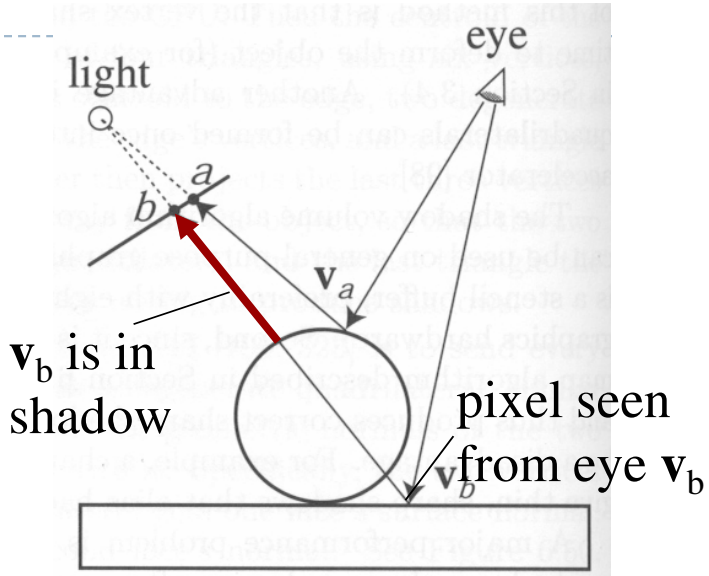



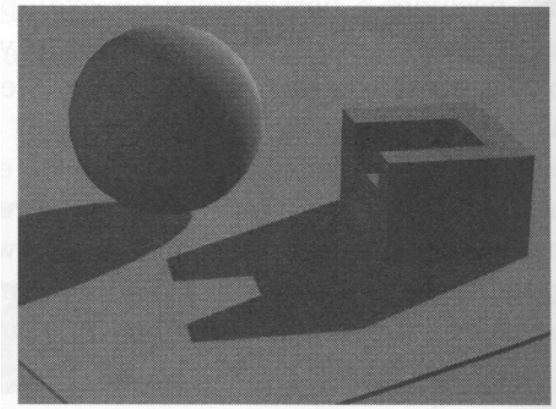
Depth image as seen from light source

# Two Pass Algorithm

**Second Pass**

- Render scene from camera position
- At each pixel, compare distance to light source with value in shadow map
  - If distance is larger, pixel is in shadow
  - If distance is smaller or equal, pixel is lit



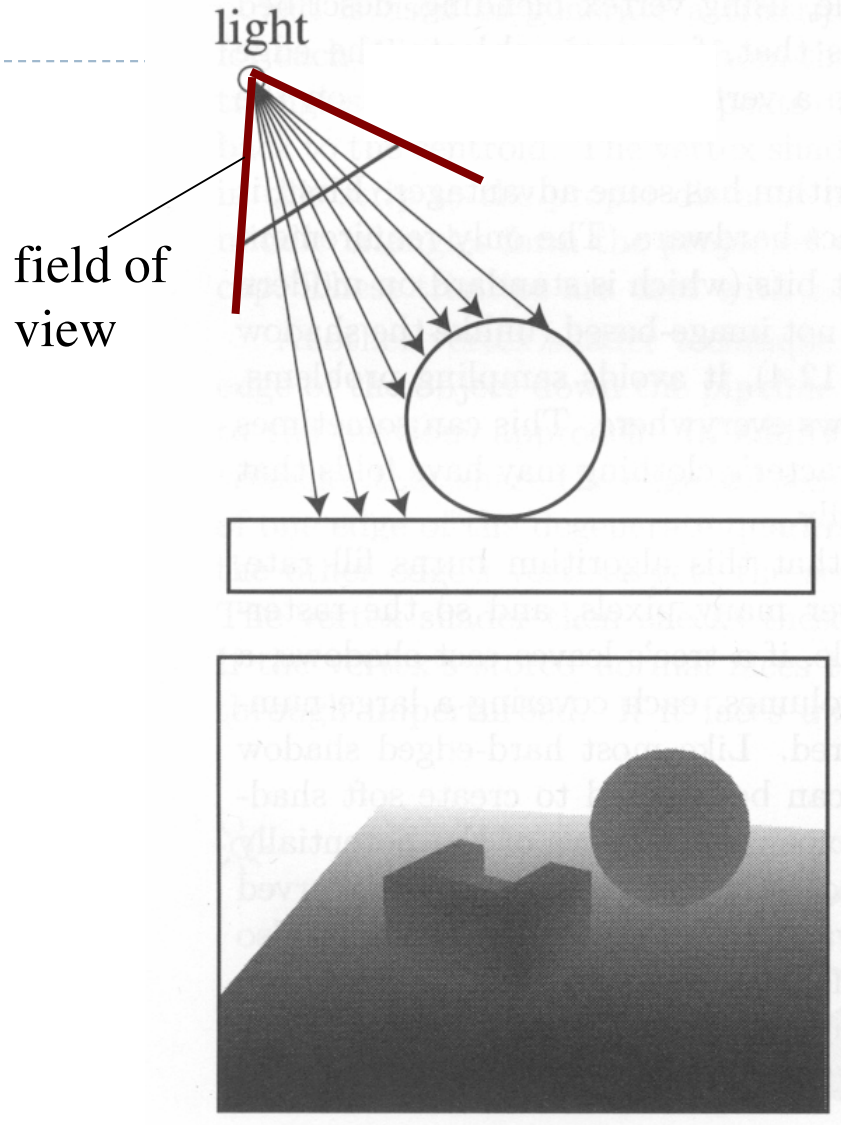
Final image with shadows

# Issues With Shadow Maps

- Limited field of view of shadow map
- Z-fighting
- Sampling problems

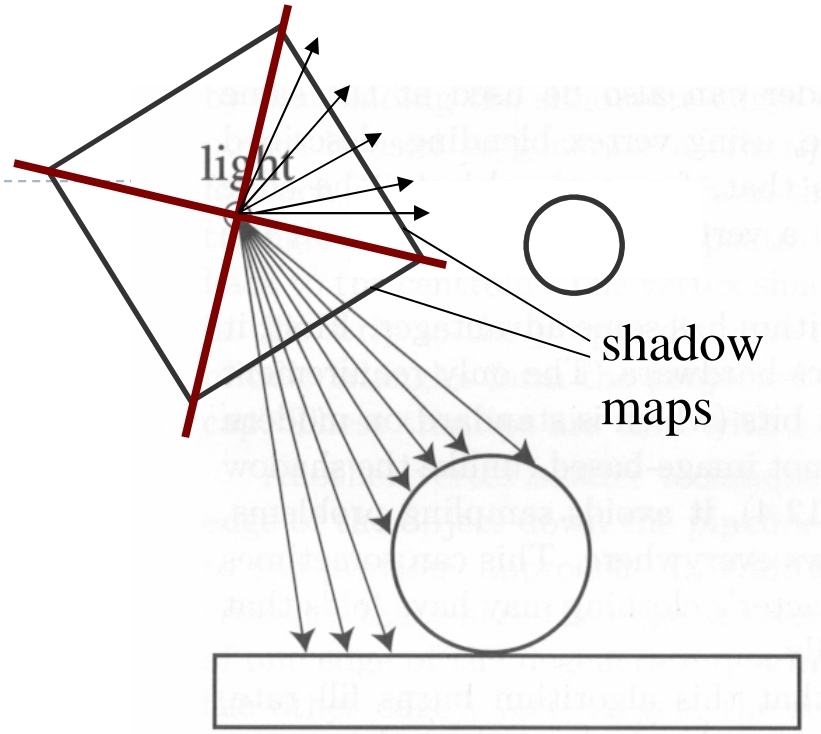
# Limited Field of View

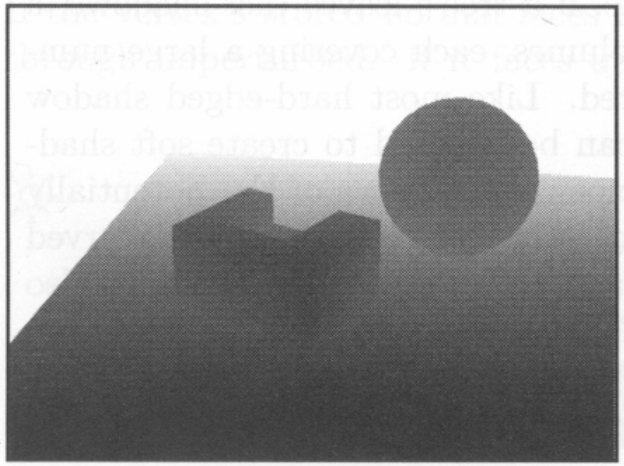
- What if a scene point is outside the field of view of the shadow map?



# Limited Field of View

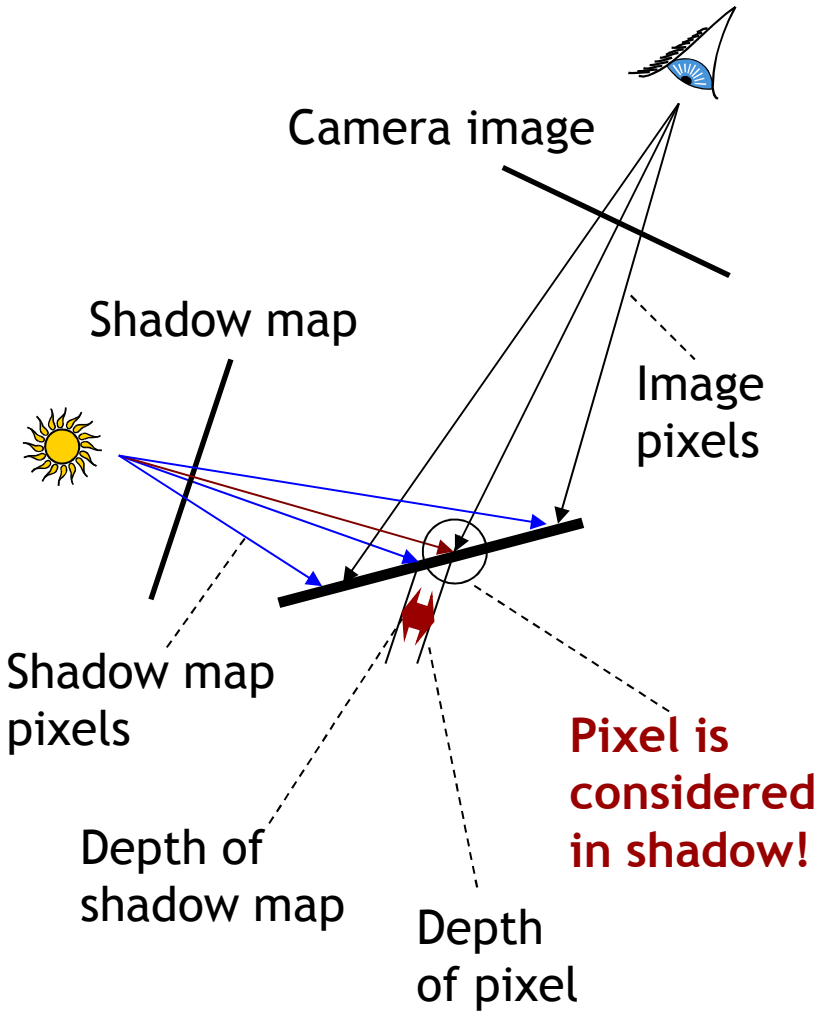
- What if a scene point is outside the field of view of the shadow map?
  → Use six shadow maps, arranged in a cube
- Requires a rendering pass for each shadow map



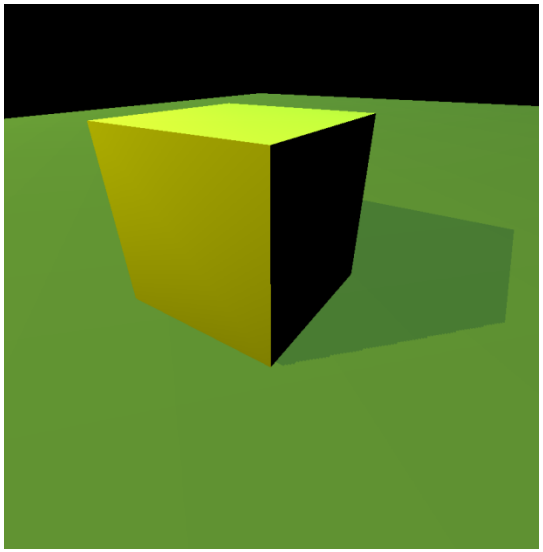
# Z-Fighting

- Depth values for points visible from light source are equal in both rendering passes
- Because of limited resolution, depth of pixel visible from light could be larger than shadow map value
- Need to add bias in first pass to make sure pixels are lit

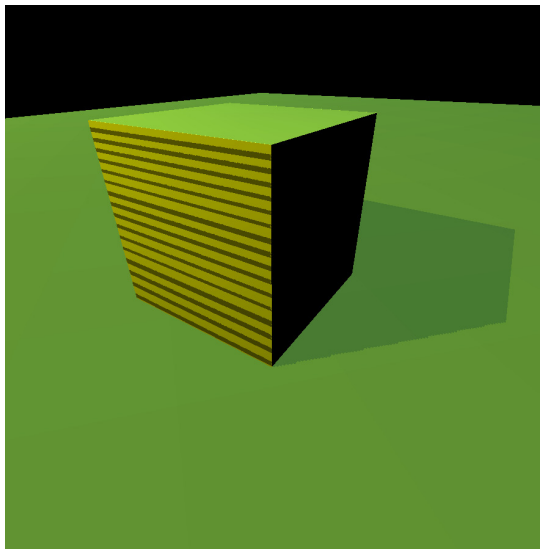


# Solution: Bias

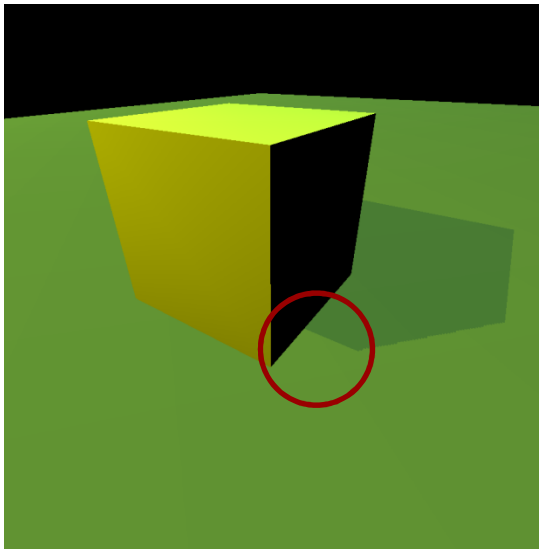
- Add bias when rendering shadow map
  - Move geometry away from light by small amount
- Finding correct amount of bias is tricky



Correct bias



Not enough bias

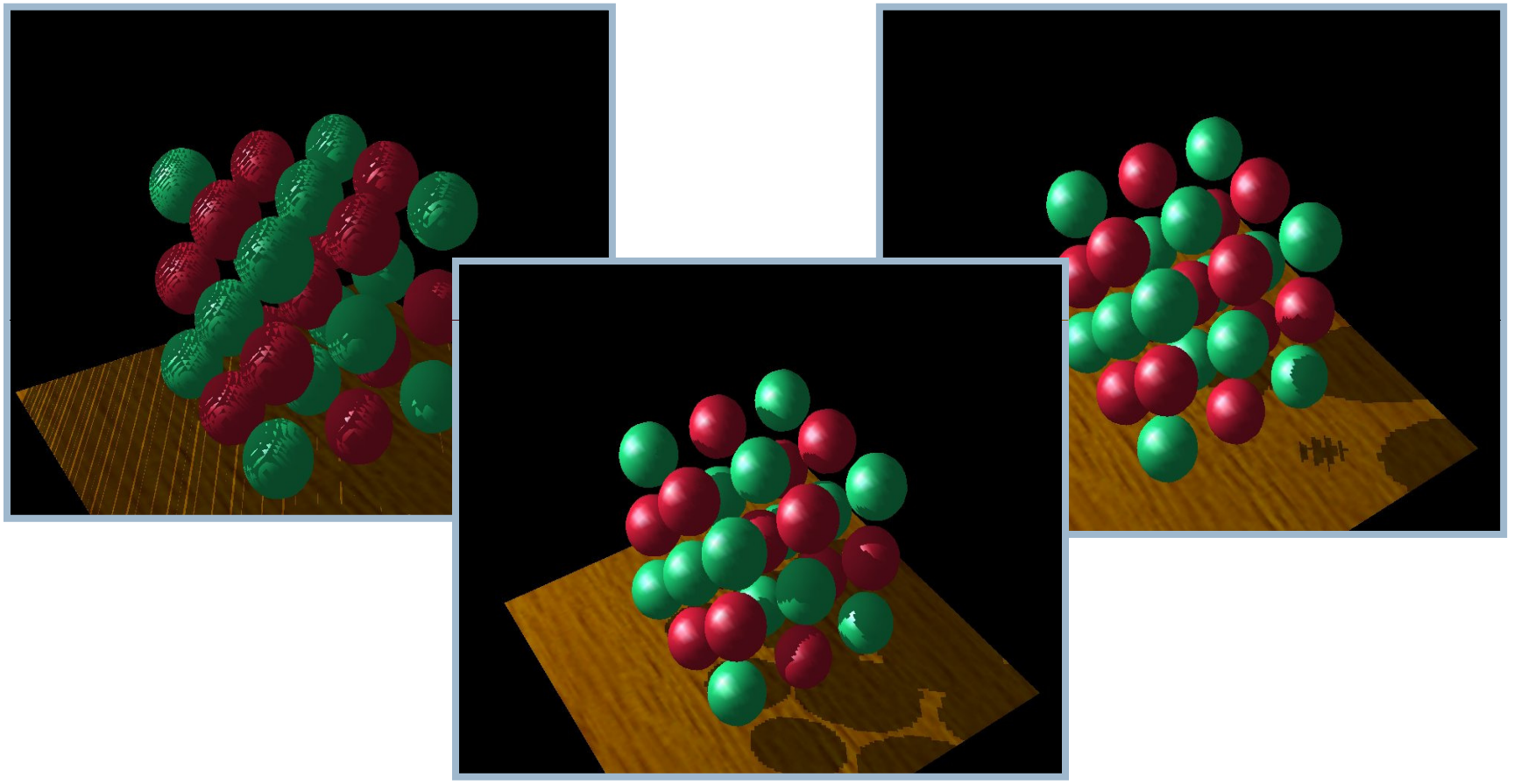


Too much bias

# Bias Adjustment

Not enough

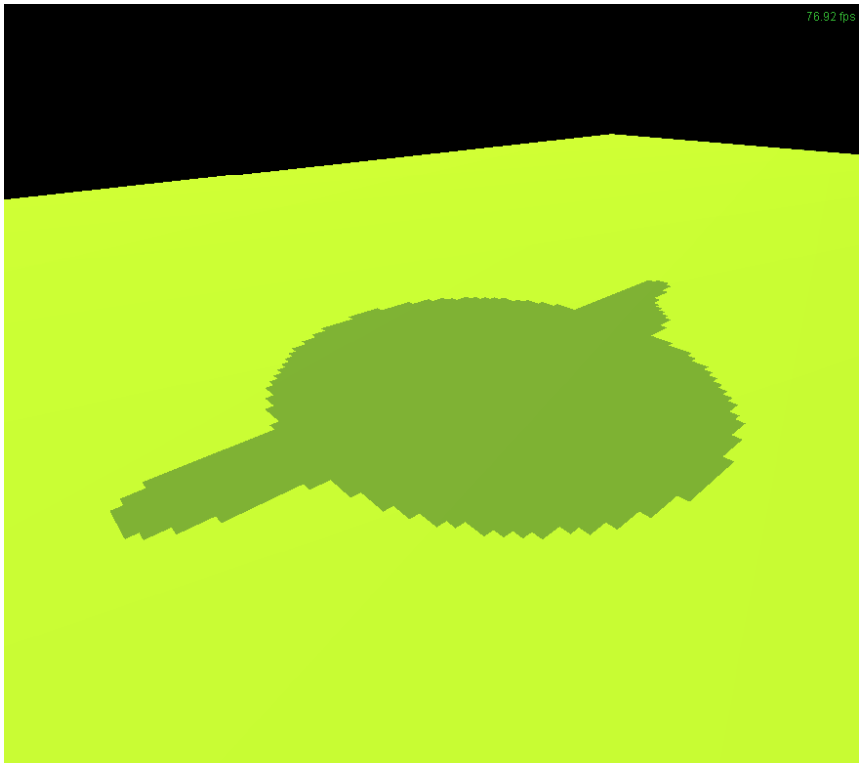
Too much



Just right

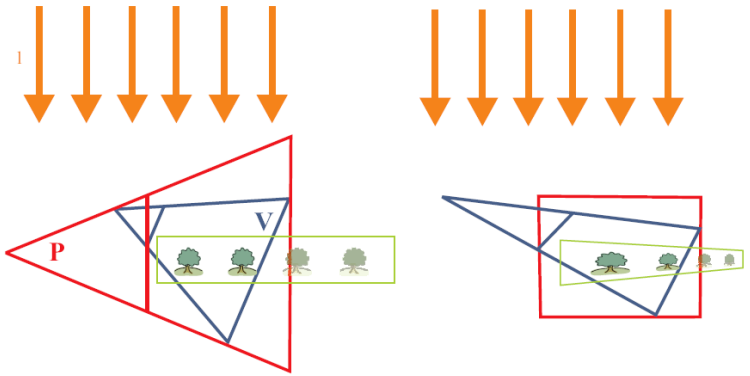
# Sampling Problems

- Shadow map pixel may project to many image pixels
  → Stair-stepping artifacts




# Solutions

- Increase resolution of shadow map
  - Not always sufficient
- Split shadow map into several tiles
- Tweak projection for shadow map rendering
  - Light space perspective shadow maps (LiSPSM)
    http://www.cg.tuwien.ac.at/research/vr/lispsm/




- Combination of splitting and LiSPSM
  - Basis for most serious implementations

# Shadow Mapping With GLSL

**First Pass**

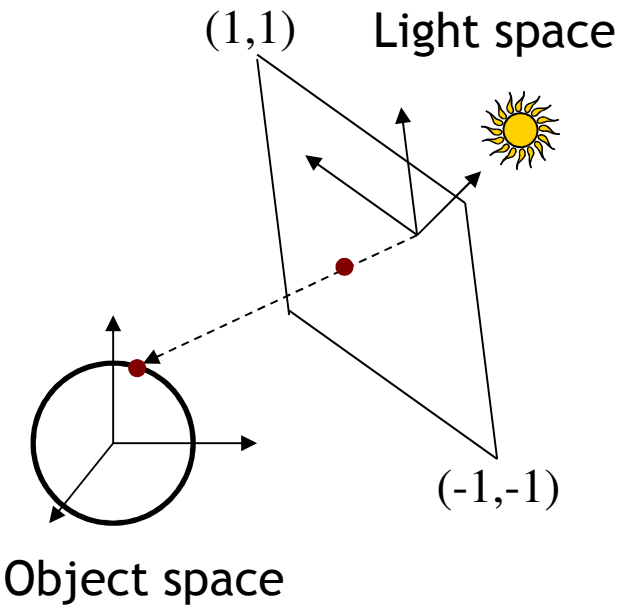
- Render scene by placing camera at light source position
- Compute light view (look at) matrix
  - Similar to computing camera matrix from look-at, up vector
  - Compute its inverse to get world-to-light transform
- Determine view frustum such that scene is completely enclosed
  - Use several view frusta/shadow maps if necessary

# First Pass

- Each vertex point is transformed by

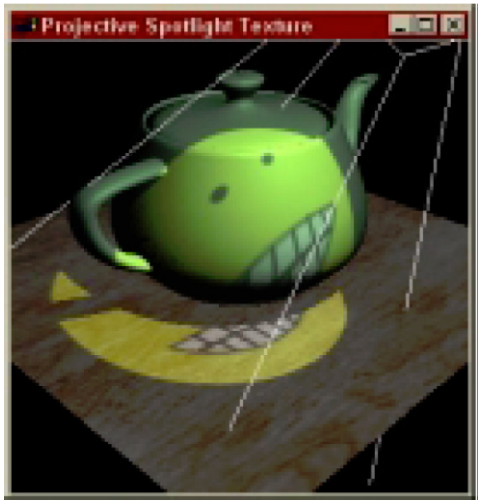
$$\mathbf{P}_{light}\mathbf{V}_{light}\mathbf{M}$$

  - Object-to-world (modeling) matrix $\mathbf{M}$
  - World-to-light space matrix $\mathbf{V}_{light}$
  - Light frustum (projection) matrix $\mathbf{P}_{light}$
- Remember: points within frustum are transformed to unit cube $[-1,1]^3$



# First Pass

- Use glPolygonOffset to apply depth bias
- Store depth image in a texture
  - Use glCopyTexImage with internal format GL_DEPTH_COMPONENT




Final result
with shadows


Scene rendered
from light source


Depth map
from light source

# Second Pass

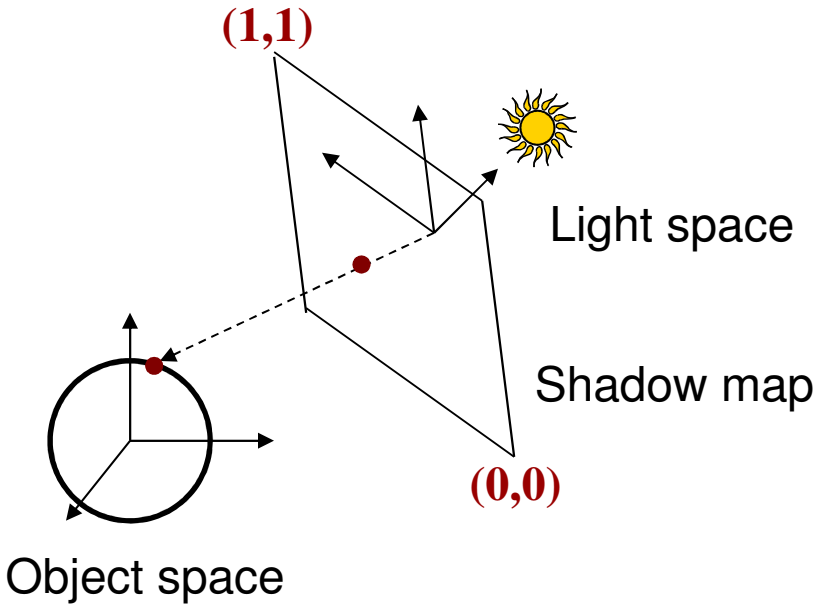
- Render scene from camera
- At each pixel, look up corresponding location in shadow map
- Compare depths with respect to light source

# Shadow Map Look-Up

- Need to transform each point from object space to shadow map
- Shadow map texture coordinates are in $[0,1]^2$
- Transformation from object to shadow map coordinates

$$\mathbf{T} = \begin{bmatrix} 1/2 & 0 & 0 & 1/2 \\ 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_{light}\mathbf{V}_{light}\mathbf{M}$$

- $\mathbf{T}$ is called texture matrix
- After perspective projection we have shadow map coordinates



# Shadow Map Look-Up

- Transform each vertex to normalized frustum of light

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Pass s,t,r,q as texture coordinates to rasterizer
- Rasterizer interpolates s,t,r,q to each pixel
- Use projective texturing to look up shadow map
  - This means, the texturing unit automatically computes s/q,t/q,r/q,1
  - s/q,t/q are shadow map coordinates in $[0,1]^2$
  - r/q is depth in light space
- Shadow depth test: compare shadow map at (s/q,t/q) to r/q

# GLSL Specifics

**In application**

- Store matrix **T** in OpenGL texture matrix
- Set using `glMatrixMode(GL_TEXTURE)`

**In vertex shader**

- Access texture matrix through predefined uniform `gl_TextureMatrix`

**In fragment shader**

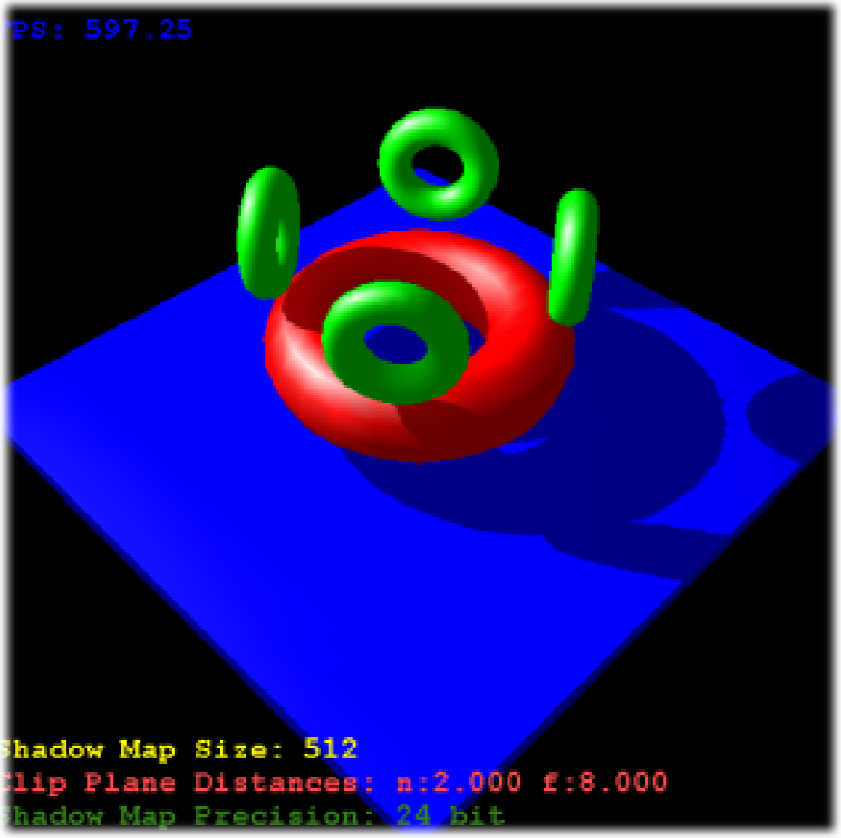
- Declare shadow map as sampler2DShadow
- Look up shadow map using projective texturing with `vec4 texture2DProj(sampler2D, vec4)`

# Implementation Specifics

- When you do a projective texture look up on a `sampler2DShadow`, the depth test is performed automatically
  - Return value is $(1,1,1,1)$ if lit
  - Return value is $(0,0,0,1)$ if shadowed
- Simply multiply result of shading with current light source with this value

# Demo

- Shadow mapping demo from
  http://www.paulsprojects.net/opengl/shadowmap/shadowmap.html



# Lecture Overview

- Shadows
- Shadow Mapping
- Shadow Volumes

# Shadow Volumes

NVIDIA md2shader demo

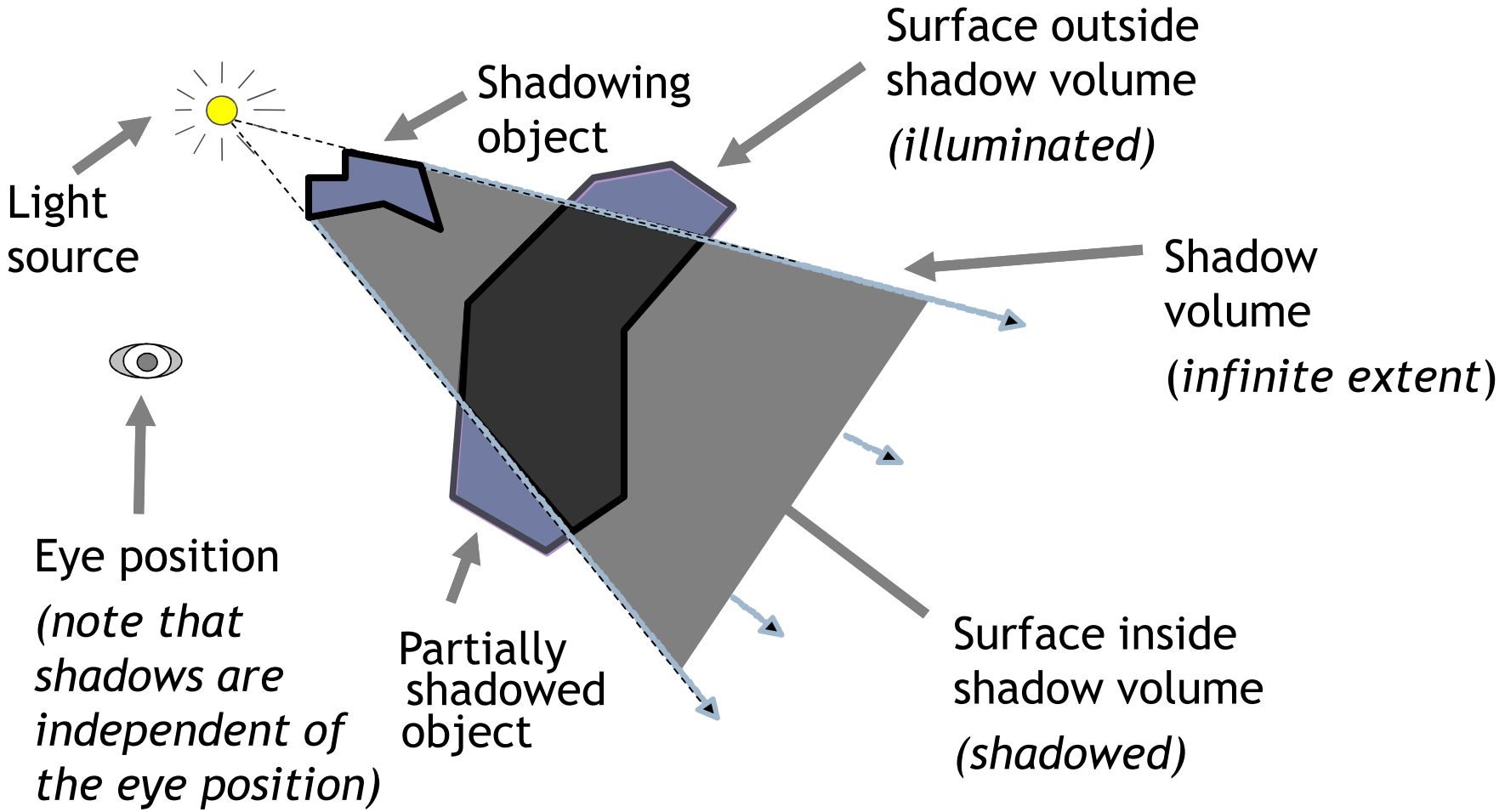
# Shadow Volumes

- A single point light source splits the world in two
  - Shadowed regions
  - Unshadowed regions
  - Volumetric shadow technique
- A shadow volume is the boundary between these shadowed and unshadowed regions
  - Determine if an object is inside the boundary of the shadowed region and know the object is shadowed

# Shadow Volumes

- Many variations of the algorithm exist
- Most popular ones use the stencil buffer
  - Depth Pass
  - Depth Fail (a.k.a. Carmack's Reverse, developed for Doom 3)
  - Exclusive-Or (limited to non-overlapping shadows)
- Most algorithms designed for hard shadows
- Algorithms for soft shadows exist

# Shadow Volumes

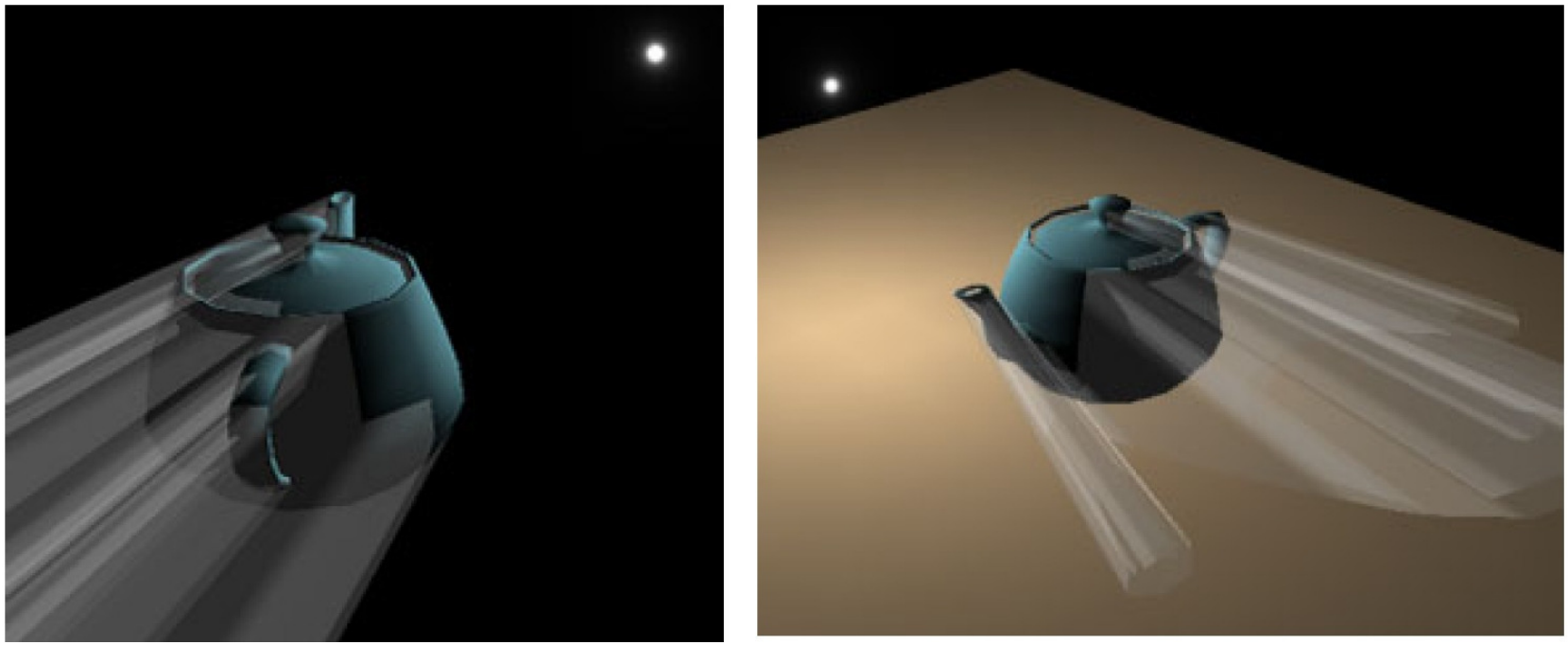


# Shadow Volume Algorithm

- High-level view of the algorithm
  - Given the scene and a light source position, determine the geometry of the shadow volume
  - Render the scene in two passes
    - Draw scene with the light *enabled*, updating ***only*** fragments in *unshadowed* region
    - Draw scene with the light *disabled*, updated ***only*** fragments in *shadowed* region

# Shadow Volume Construction

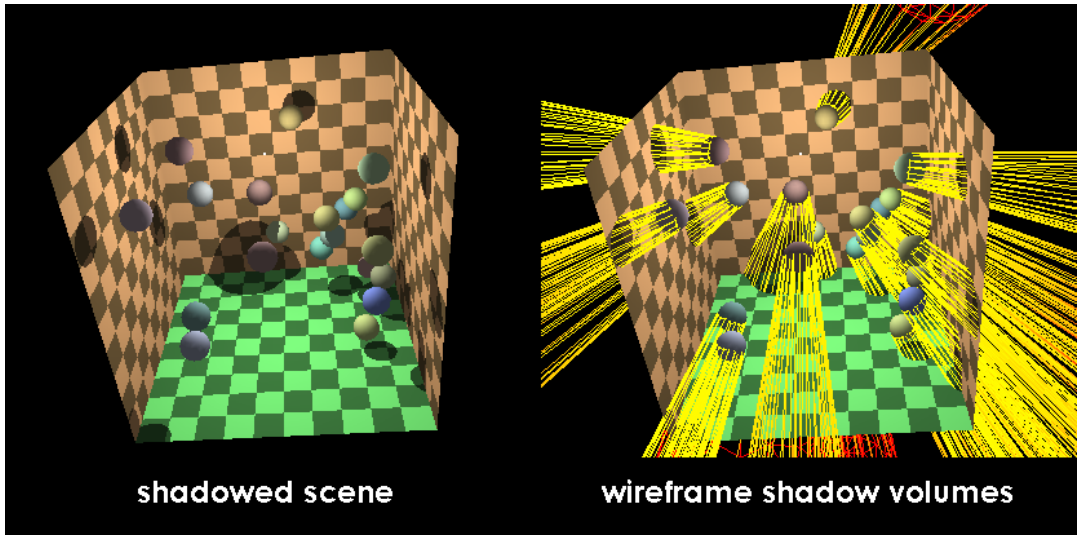
- Need to generate shadow polygons to bound shadow volume
- Extrude silhouette edges from light source



Extruded shadow volumes

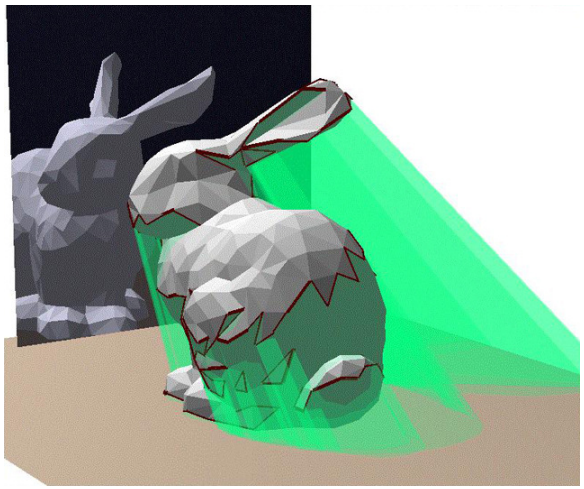
# Shadow Volume Construction

- Done on the CPU
- Silhouette edge detection
  - An edge is a silhouette if one adjacent triangle is front facing, the other back facing with respect to the light
- Extrude polygons from silhouette edges



# Stenciled Shadow Volumes

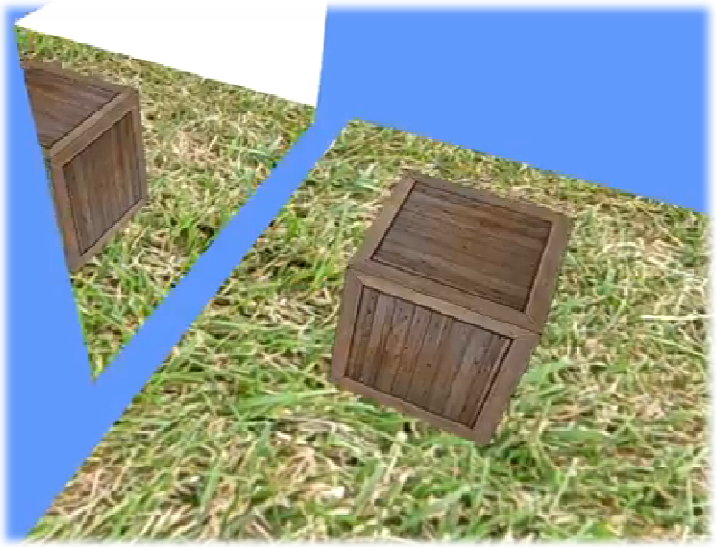
- Advantages
  - Support omnidirectional lights
  - Exact shadow boundaries
- Disadvantages
  - Fill-rate intensive
  - Expensive to compute shadow volume geometry
  - Hard shadow boundaries, not soft shadows
  - Difficult to implement robustly



*Source: Zach Lynn*

# The Stencil Buffer

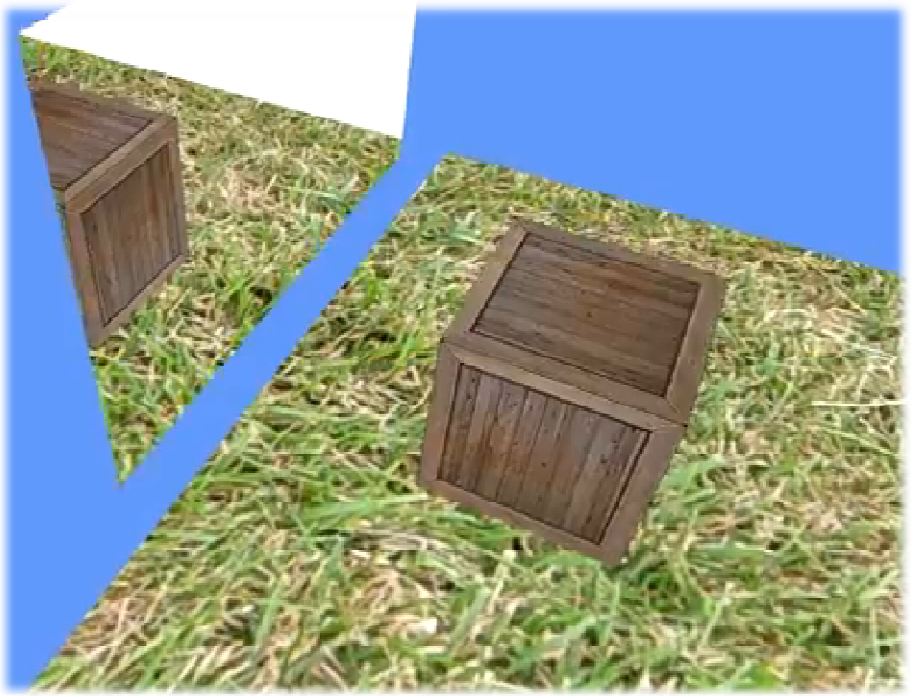
- Per-pixel 2D buffer on the GPU
- Similarities to depth buffer in way it is stored and accessed
- Stores an integer value per pixel, typically 8 bits
- Like a stencil, allows to block pixels from being drawn
- Typical uses:
  - shadow mapping
  - planar reflections
  - portal rendering



*Source: Adrian-Florin Visan*

# Video

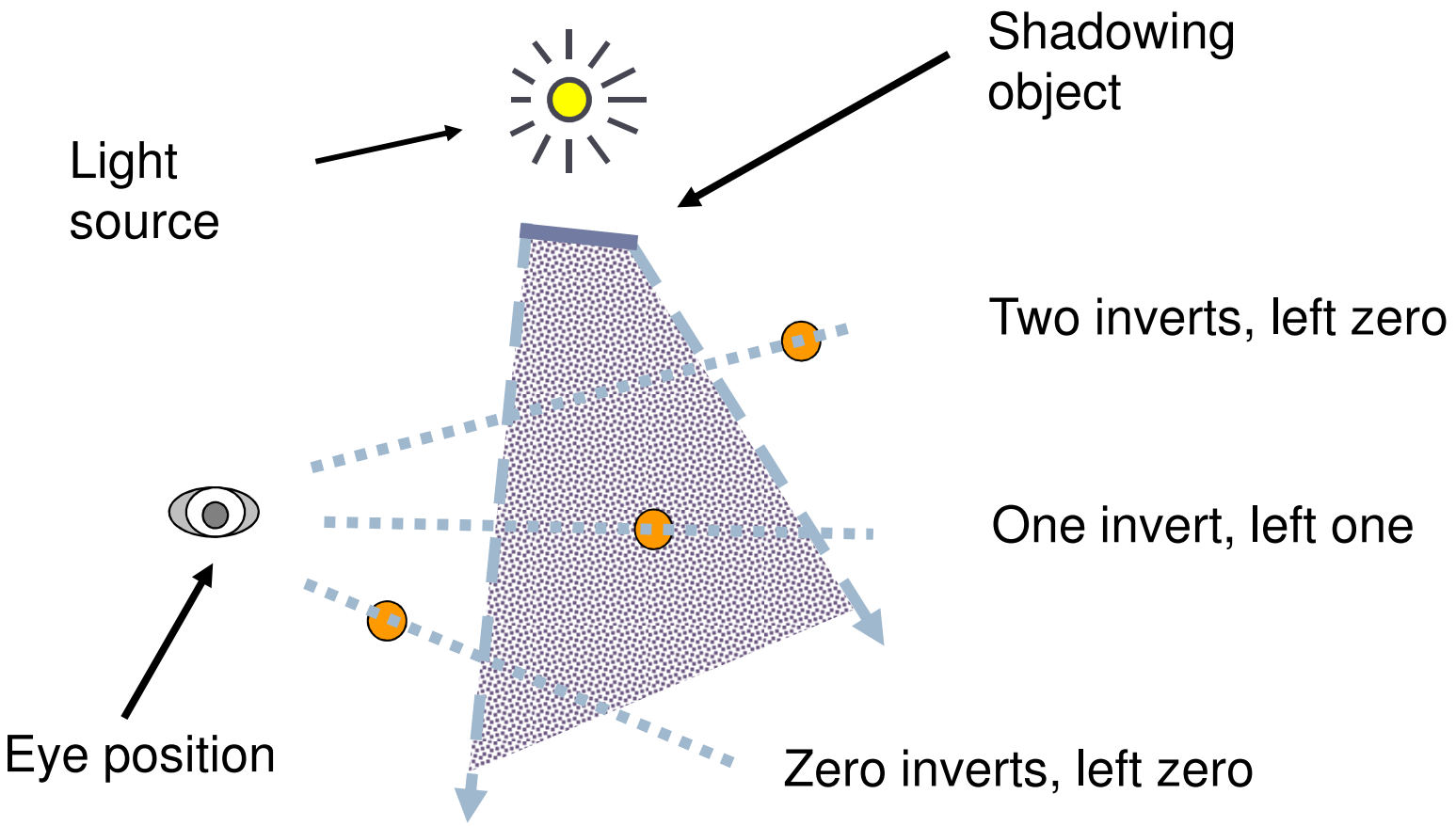
- Using the stencil buffer, rendering a stencil mirror tutorial
  - http://www.youtube.com/watch?v=3xzq-YEOIsk



# Tagging Pixels as Shadowed or Unshadowed

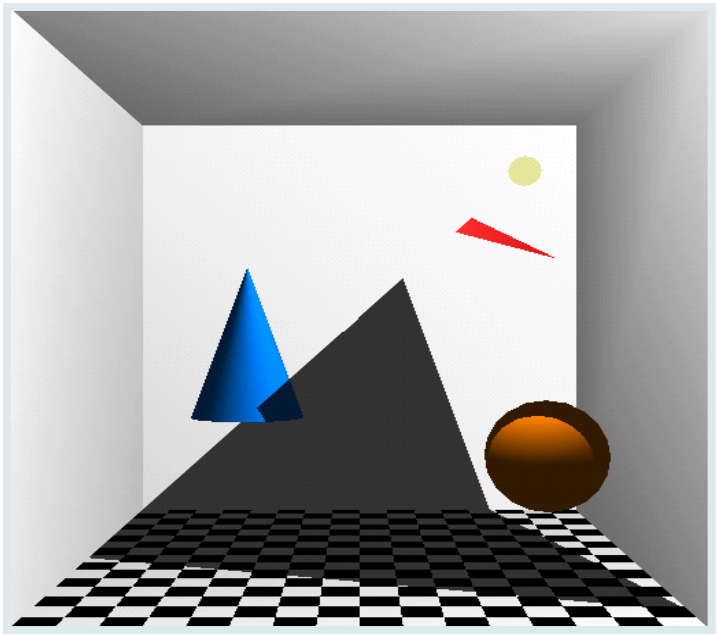
- The stenciling approach
  - Clear stencil buffer to zero and depth buffer to 1.0
  - Render scene to leave depth buffer with closest Z values
  - Render shadow volume into frame buffer with depth testing but *without* updating color and depth, but *inverting* a stencil bit (Exclusive-Or method)
  - This leaves stencil bit set within shadow

# Stencil Inverting of Shadow Volume

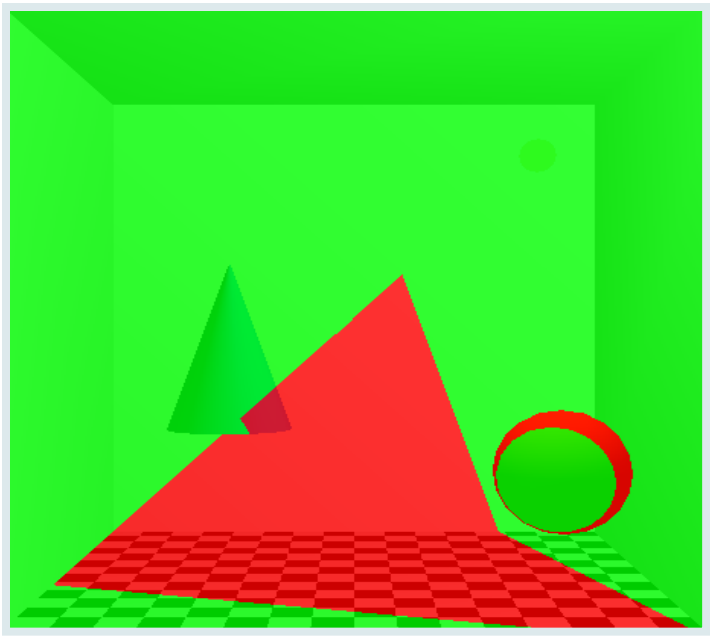


# Visualizing Stenciled Shadow Volume Tagging

**Shadowed scene**



**Stencil buffer contents**



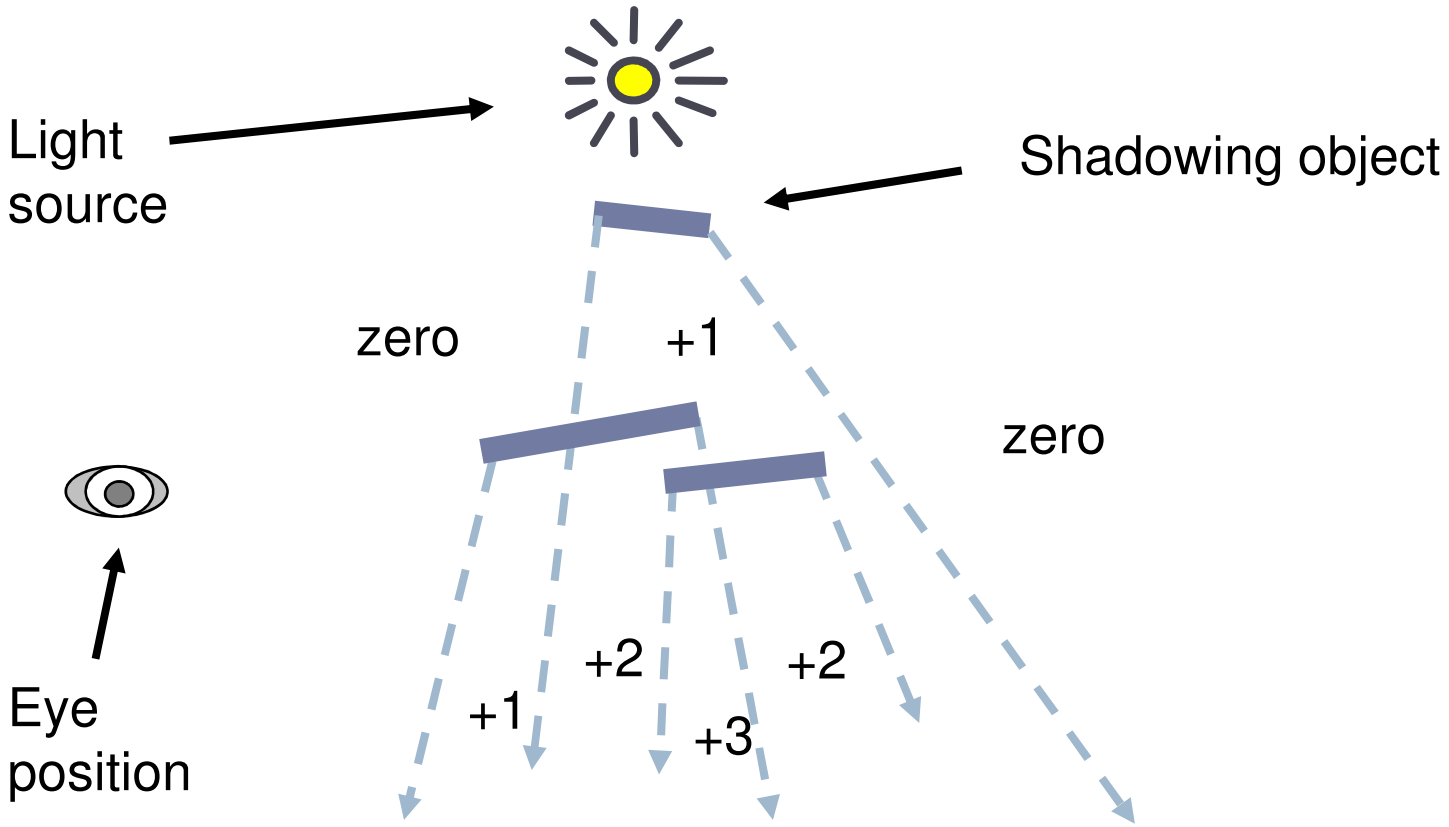
*red = stencil value of 1*
*green = stencil value of 0*

GLUT *shadowvol* example credit: Tom McReynolds

# For Shadow Volumes With Intersecting Polygons

- Use a stencil enter/leave counting approach
  - Draw shadow volume twice using face culling
    - 1st pass: render *front* faces and *increment* when depth test passes
    - 2nd pass: render *back* faces and *decrement* when depth test passes
  - This two-pass way is more expensive than invert
  - Inverting is better if all shadow volumes have no polygon intersections

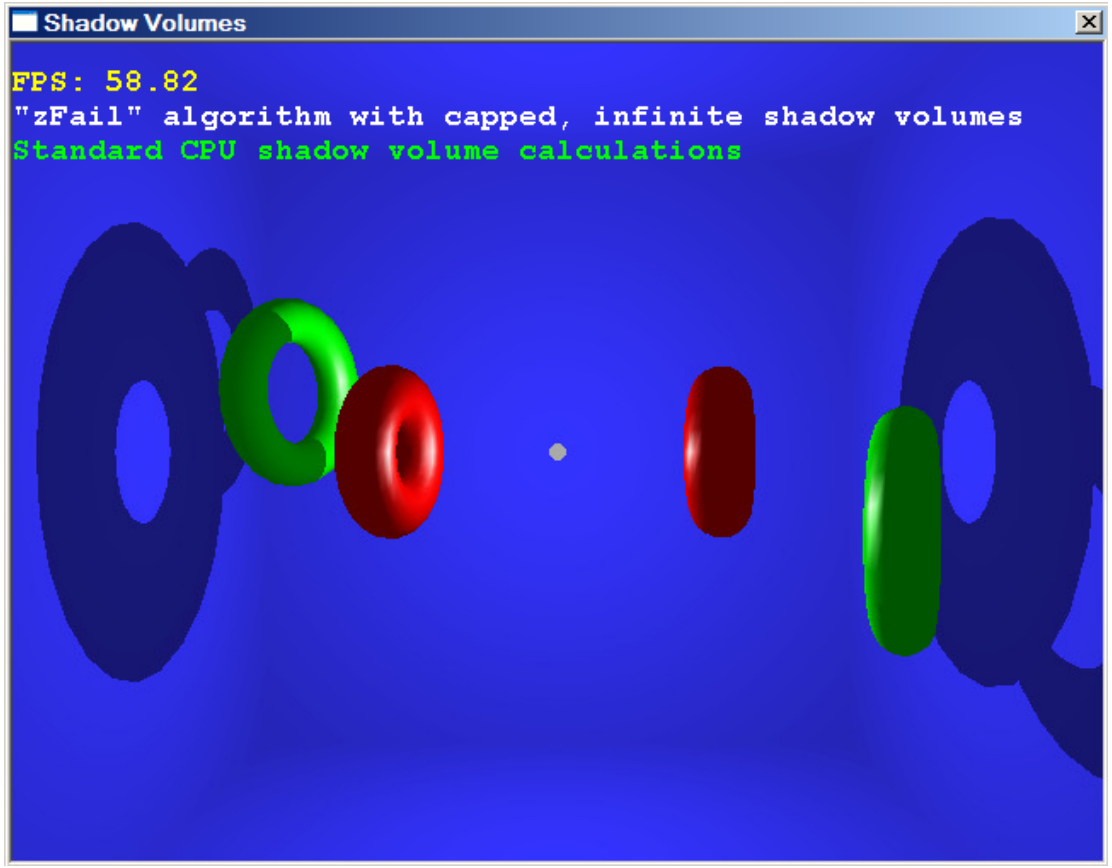
# Increment/Decrement Stencil Volumes



# Shadow Volume Demo

- URL:

http://www.paulsprojects.net/opengl/shadvol/shadvol.html

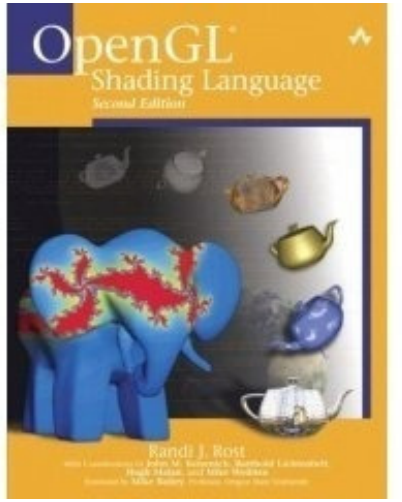


# Resources for Shadow Rendering

- Overview, lots of links
  http://www.realtimerendering.com/
- Basic shadow maps
  http://en.wikipedia.org/wiki/Shadow_mapping
- Avoiding sampling problems in shadow maps
  http://www.comp.nus.edu.sg/~tants/tsm/tsm.pdf
  http://www.cg.tuwien.ac.at/research/vr/lispsm/
- Faking soft shadows with shadow maps
  http://people.csail.mit.edu/ericchan/papers/smoothie/
- Alternative: shadow volumes
  http://en.wikipedia.org/wiki/Shadow_volume
  http://www.gamedev.net/reference/articles/article1873.asp

# More on Shaders

- OpenGL shading language book
  - "Orange Book"




- Shader Libraries
  - GLSL:
    - http://www.geeks3d.com/geexlab/shader_library.php
  - HLSL:
    - NVidia shader library
    - http://developer.download.nvidia.com/shaderlibrary/webpages/shader_library.html