Computergrafik

Matthias Zwicker
Universität Bern
Herbst 2016

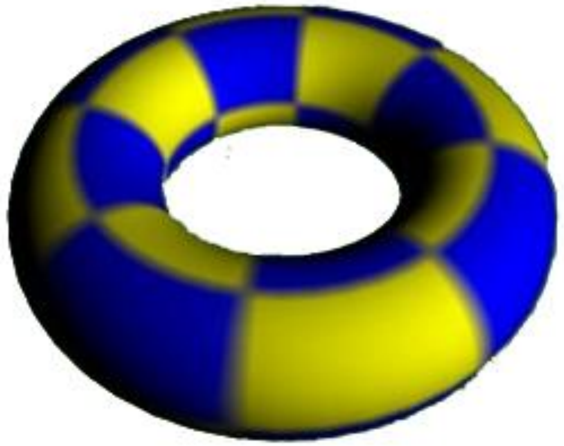
Today

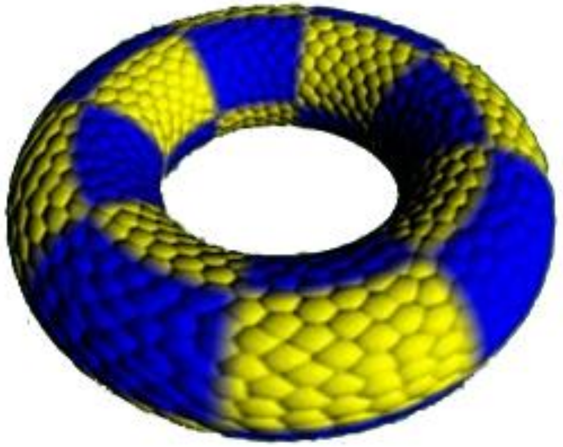
- Bump mapping
- Shadows
- Shadow mapping
- Shadow mapping in OpenGL

Bump mapping

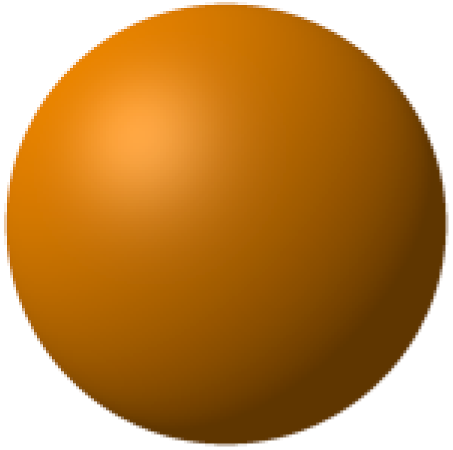
- Surface detail is often the result of small perturbations in the surface geometry
  - Modeling detailed surfaces would lead to impractical number of triangles
- Bump mapping alters the surface normal
  - Normals are encoded in texture maps
  - Provide the illusion of small scale surface detail
  - Does not change geometry (triangles)
- Requires per-pixel shading using a fragment program

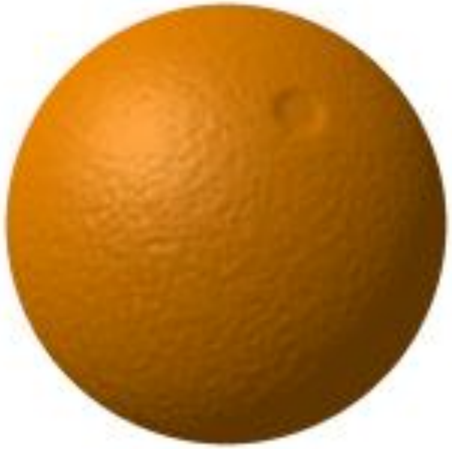
Bump mapping

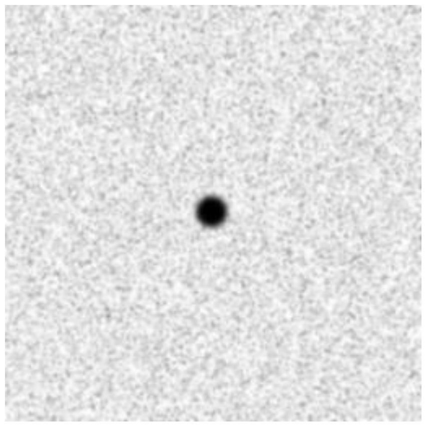

No bump mapping


With bump mapping

Bump mapped plane


No bump mapping


With bump mapping

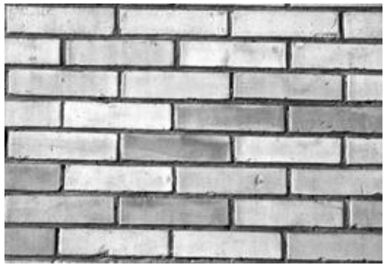

Bump texture

Bump mapping

1. Generating and storing bump maps
2. Rendering with bump maps

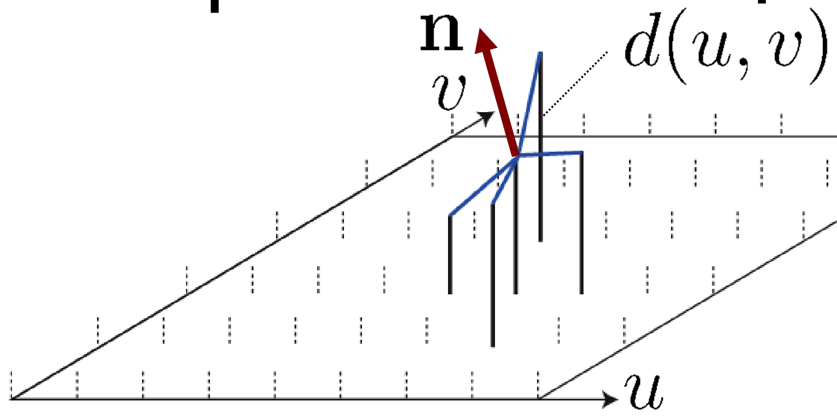
Generating bump maps

- Usually done in a pre-process
- Input
  - Texture map that encodes small surface displacements
  - Height field
  - E.g., use gray scale image as height values
- Output
  - Texture map that encodes normals of displaced surface
  - This texture will be stored as an image, read by the application



Generating bump maps

- Start with displacement map (height field)



- Normal

$$\mathbf{n}(u,v) = \frac{\partial[d(u,v),u,v]}{\partial u} \times \frac{\partial[d(u,v),u,v]}{\partial v}$$

- Discrete case using central differencing

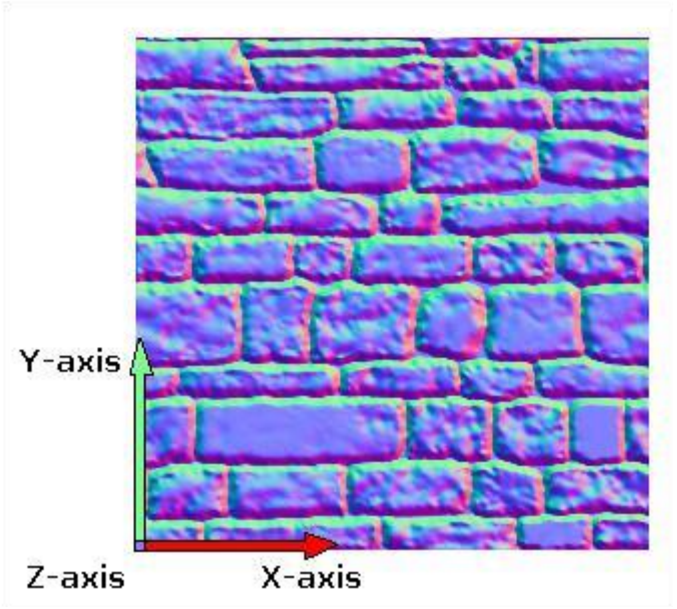
$$\mathbf{n}(u,v) = \frac{[d(u+\Delta u,v) - d(u-\Delta u,v), 2\Delta u, 0]}{2\Delta u} \times \frac{[d(u,v+\Delta v) - d(u,v-\Delta v), 0, 2\Delta v]}{2\Delta v}$$

  - Usually, $\Delta u, \Delta v = 1$

- Normalize length of normal!

Storing bump maps

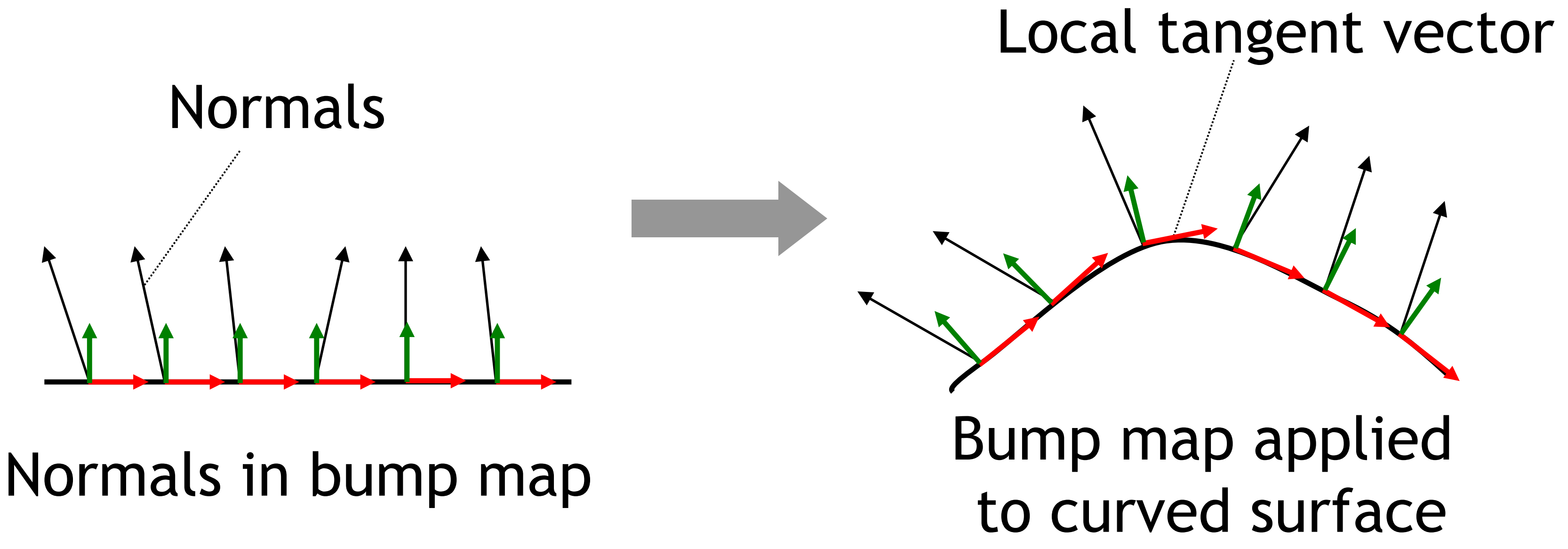
- Encode normal direction in RGB color channels
  - Coordinates of unit normal are in $[-1..1]^3$
  - Need to map range $[-1..1]$ to $[0..255]$ for all channels



RGB encoded bump map

Rendering with bump maps

- When applying a bump map to a curved surface, how are the normals specified in the bump map related to the surface?
- Normals are defined relative to local tangent/normal vectors

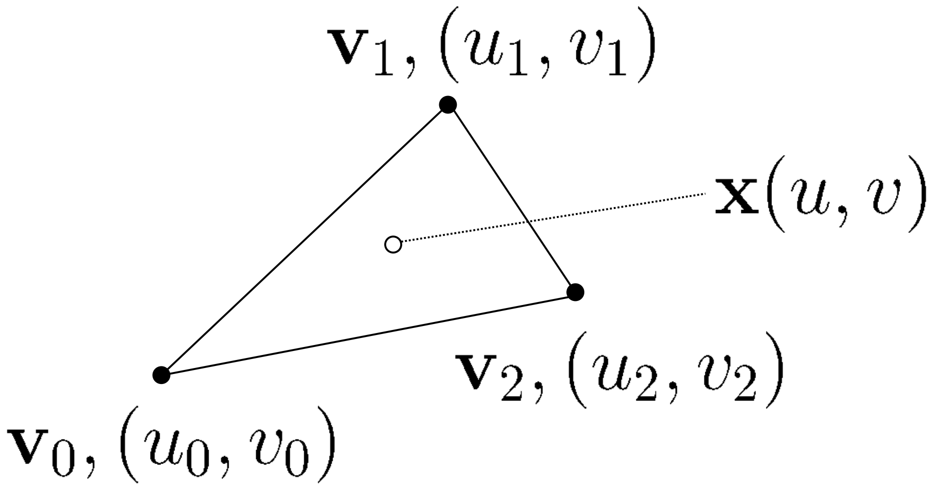


Rendering with bump maps

- Bump map normals are defined in tangent space
  - Defined by two tangent vectors and normal
- Will define tangent space for each triangle
  - Texture coordinates provide parameterization of each triangle, i.e., parametric patch $\mathbf{x}(u,v)$
  - Compute tangent vectors using partial derivatives of parameterization
- For shading, will need to transform normals from tangent space to camera space

Tangent space

- Triangle with texture coordinates can be expressed as parametric surface $\mathbf{x}(u,v)$
  - Triangle vertices in object space $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$
  - Texture coordinates (u_0, v_0), (u_1, v_1), (u_2, v_2),


$\mathbf{v}_1, (u_1, v_1)$
$\mathbf{x}(u, v)$
$\mathbf{v}_2, (u_2, v_2)$
$\mathbf{v}_0, (u_0, v_0)$

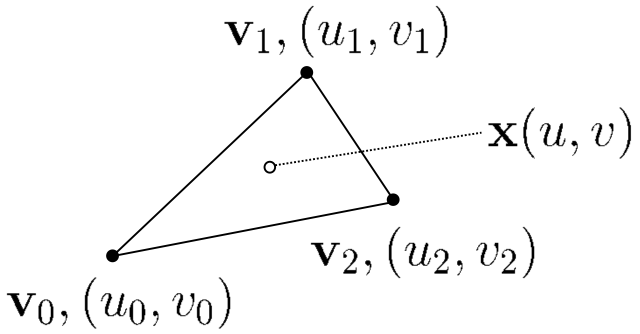
- Interpolation constraints: we know

$$\mathbf{x}(u_0, v_0) = \mathbf{v}_0, \quad \mathbf{x}(u_1, v_1) = \mathbf{v}_1, \quad \mathbf{x}(u_2, v_2) = \mathbf{v}_2$$

Tangent space

- Solve for affine function

$$\mathbf{x}(u,v) = \begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$


$\mathbf{v}_1, (u_1, v_1)$
$\mathbf{x}(u,v)$
$\mathbf{v}_2, (u_2, v_2)$
$\mathbf{v}_0, (u_0, v_0)$

- Using constraints at vertices

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} \begin{bmatrix} u_0 & u_1 & u_2 \\ v_0 & v_1 & v_2 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix}$$

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 & \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix} \begin{bmatrix} u_0 & u_1 & u_2 \\ v_0 & v_1 & v_2 \\ 1 & 1 & 1 \end{bmatrix}^{-1}$$

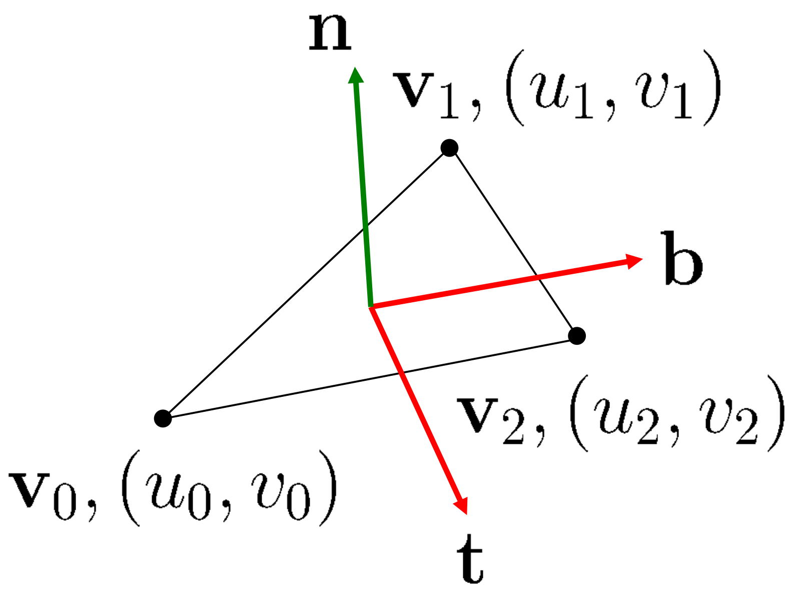
Tangent space

- Tangent space defined by two tangent vectors (tangent $\mathbf{t}$, "bi-tangent" $\mathbf{b}$), and normal vector $\mathbf{n}$

$$\mathbf{t} = \frac{\partial \mathbf{x}(u,v)}{\partial u} = \begin{bmatrix} m_{0,0} \\ m_{1,0} \\ m_{2,0} \end{bmatrix}$$

$$\mathbf{b} = \frac{\partial \mathbf{x}(u,v)}{\partial v} = \begin{bmatrix} m_{0,1} \\ m_{1,1} \\ m_{2,1} \end{bmatrix}$$

$$\mathbf{n} = \mathbf{t} \times \mathbf{b}$$


$\mathbf{n}$
$\mathbf{v}_1, (u_1, v_1)$
$\mathbf{b}$
$\mathbf{v}_2, (u_2, v_2)$
$\mathbf{v}_0, (u_0, v_0)$
$\mathbf{t}$

- $\mathbf{t}, \mathbf{b}, \mathbf{n}$ defined in object space coordinates
- Tangent, bi-tangent not orthogonal in general
- No normalization necessary

Normal in object space

- Normal map stores normals in tangent coordinates
  - Basis vectors **t, b, n**
- Can transform normal from tangent to object space
  - Given values $[bm_0, bm_1, bm_2]$ from bump map
  - Unpacked from [0..1] to range [-1..1]

$$\mathbf{n}_{objectspace} = \begin{bmatrix} \mathbf{t} & \mathbf{b} & \mathbf{n} \end{bmatrix} \begin{bmatrix} bm_0 \\ bm_1 \\ bm_2 \end{bmatrix}$$

Storing tangent vectors

Before rendering

- For each triangle, compute tangent, bi-tangent vector
- At each vertex, average tangent, bi-tangent vectors over adjacent triangles to get smooth transitions between triangles
- Store tangent vector as additional vertex attributes
  - Only one tangent vector and normal necessary
  - Second tangent vector computed on the fly

Rendering

Vertex shader

- Per-vertex input
  - Vertex position, normal, tangent vector in object space
  - Bump map texture coordinates
- Compute bi-tangent vector
- Transform everything to camera space using modelview matrix
- Output to fragment shader (will be interpolated to each pixel)
  - Vertex position, texture coordinates, tangent, bi-tangent, normal vector in camera space
  - Bump map texture coordinates

Rendering

Fragment shader

- Transform normal $[bm_0, bm_1, bm_2]$ stored in bump map to camera coordinates
  - Use $\mathbf{t}, \mathbf{b}, \mathbf{n}$ basis to transform to object space
  - Use modelview matrix to transform from object space to camera space

$$\mathbf{n}_{camerspace} = \left[\text{ modelview } \right] \left[\begin{array}{ccc} \mathbf{t} & \mathbf{b} & \mathbf{n} \end{array} \right] \left[\begin{array}{c} bm_0 \\ bm_1 \\ bm_2 \end{array} \right]$$

  - Normalize $\mathbf{n}_{camerspace}$
- Perform lighting in camera coordinates

Variations

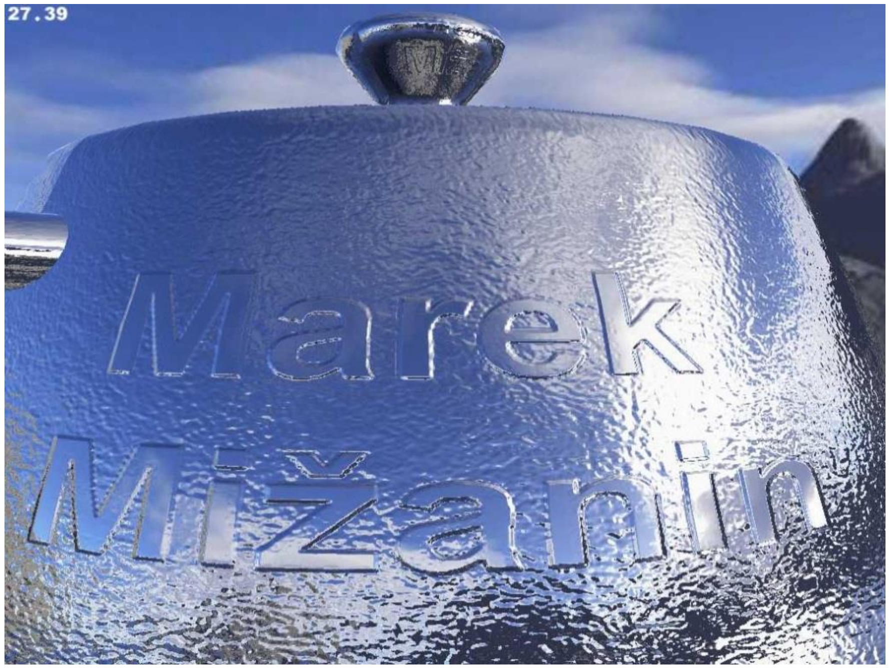
- Perform lighting in different coordinate system than camera space
  - Object space
  - Tangent space
- Tangent space is more efficient
  - Transform light direction to tangent space in vertex shader
  - Rasterizer interpolates it across triangle
  - No need to transform bump mapped normal at each pixel (in fragment shader)

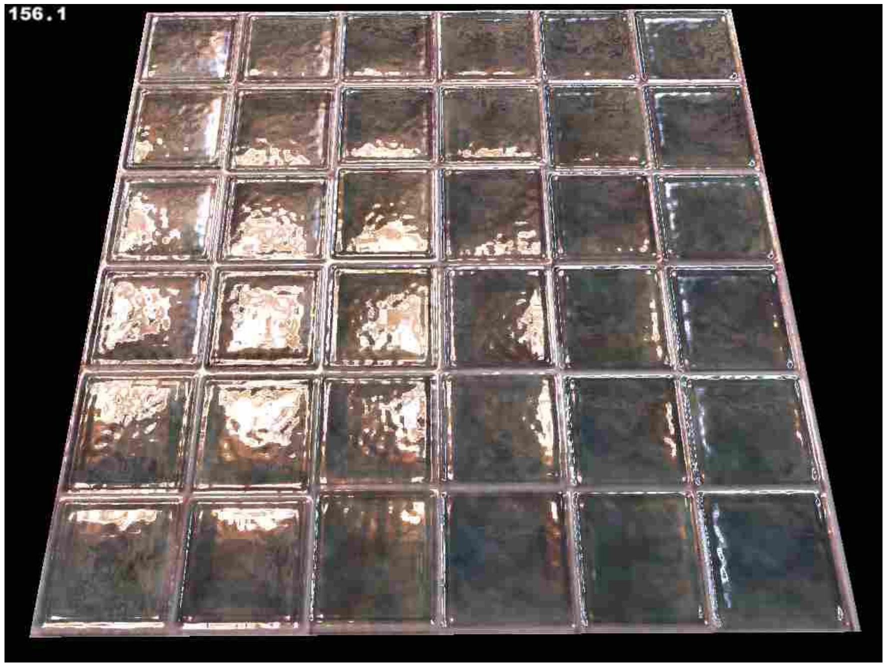
Caveats

- Need mesh with texture coordinates to define tangent space
- Avoid triangles with zero area in texture space
  - Cannot compute valid tangent space
- Avoid triangles with negative area in texture space
  - May happen when texture is mirrored
- Avoid non-uniform stretching of bump map

Combination with env. map

- “Environment mapped bump mapping” (EMBM)
- Use bump mapped normal to compute reflection vector, look up cube map

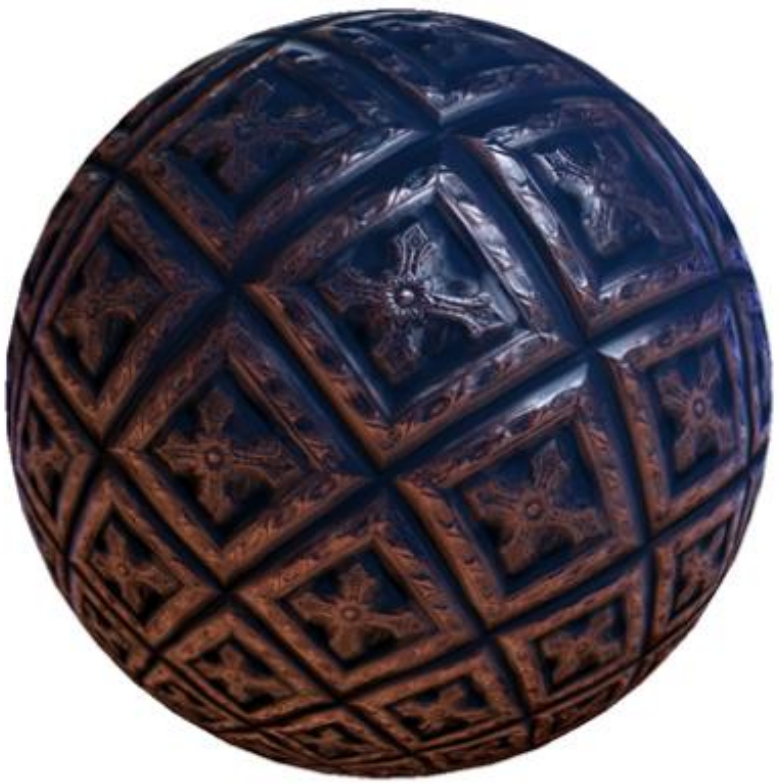
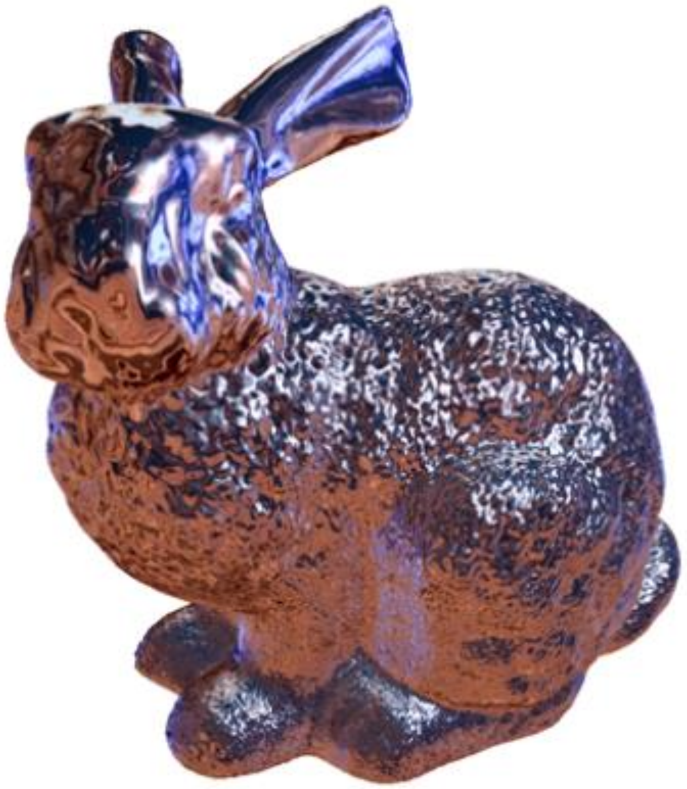




http://zanir.wz.cz/?paged=3&lang=en

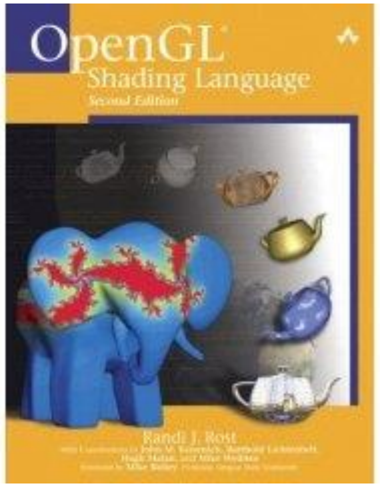
Env. mapped bump mapping

- Use additional 'dirt' texture to modulate strength of reflection from environment map

Tutorials

- Caution, slightly different derivation
  http://www.blacksmith-studios.dk/projects/downloads/bumpmapping_using_cg.php
- OpenGL shading language book
  - Bump mapping uses shading in tangent space

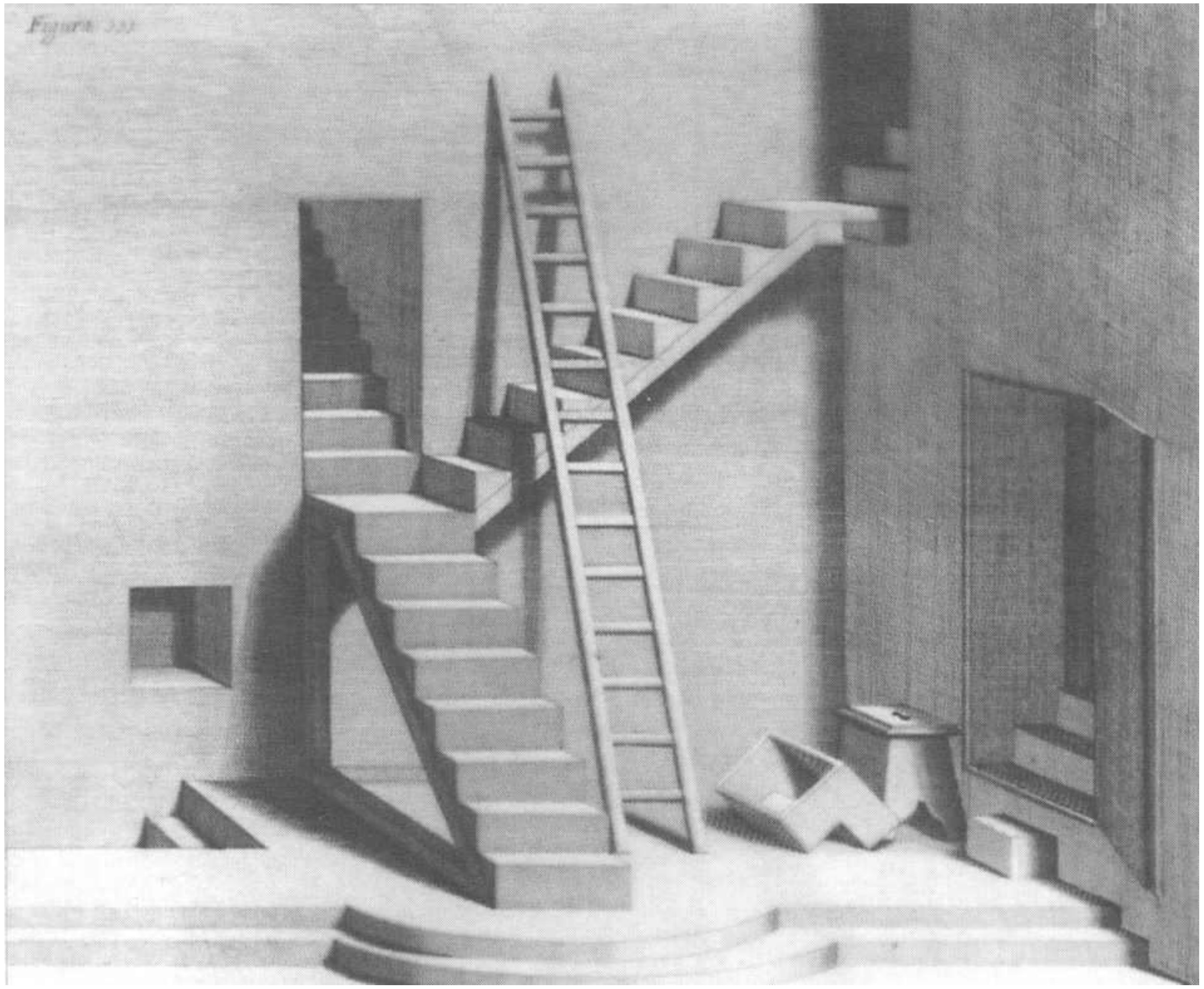


Today

- Bump mapping
- Shadows
- Shadow mapping
- Shadow mapping in OpenGL

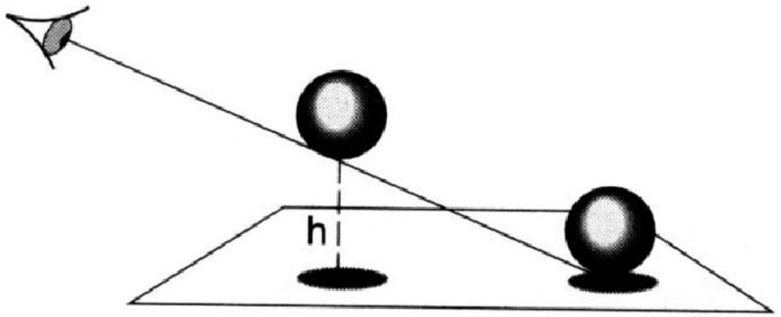
Why are shadows important?

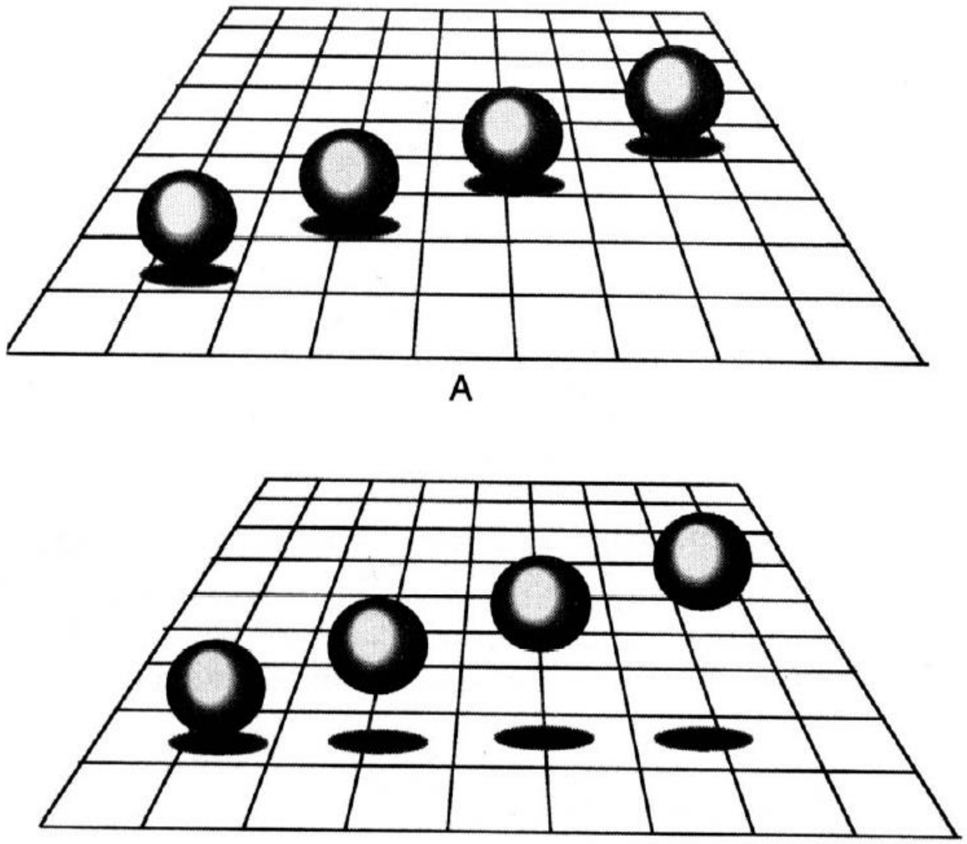
- Cues on scene lighting



Why are shadows important?

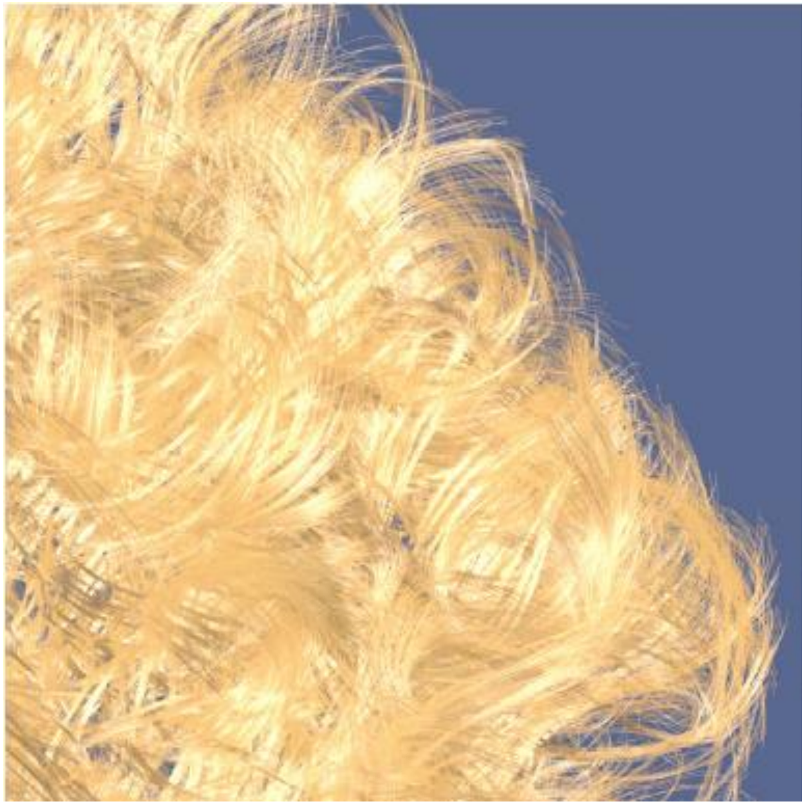
- Contact points
- Depth cues

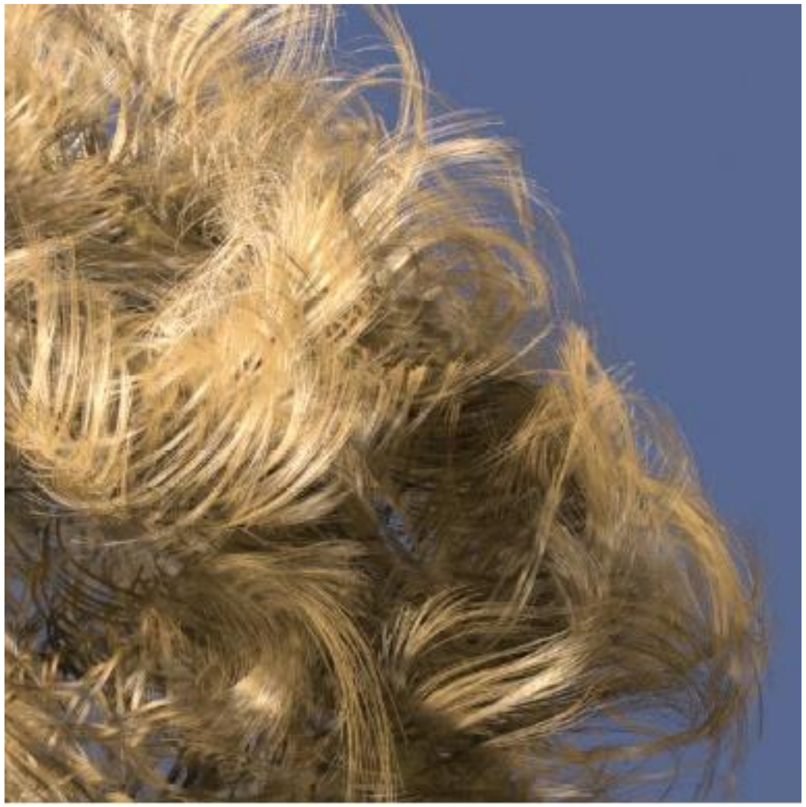




Why are shadows important?

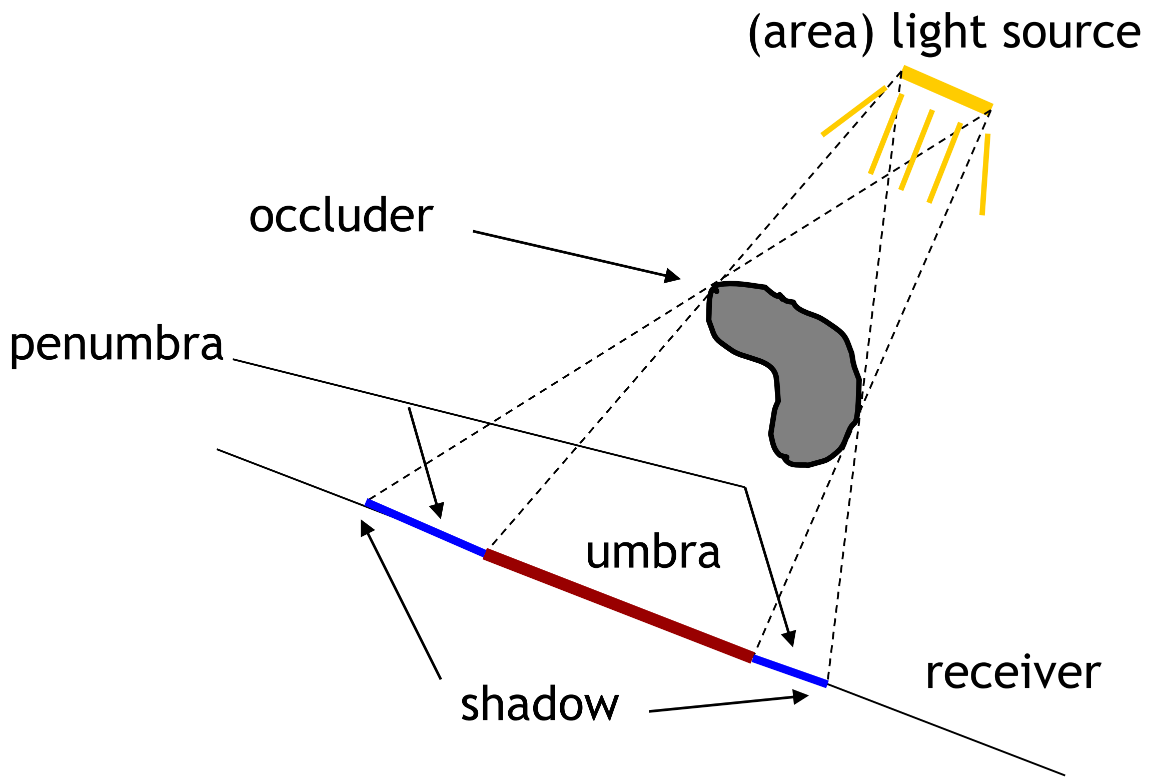
- Realism


Without self-shadowing


Without self-shadowing

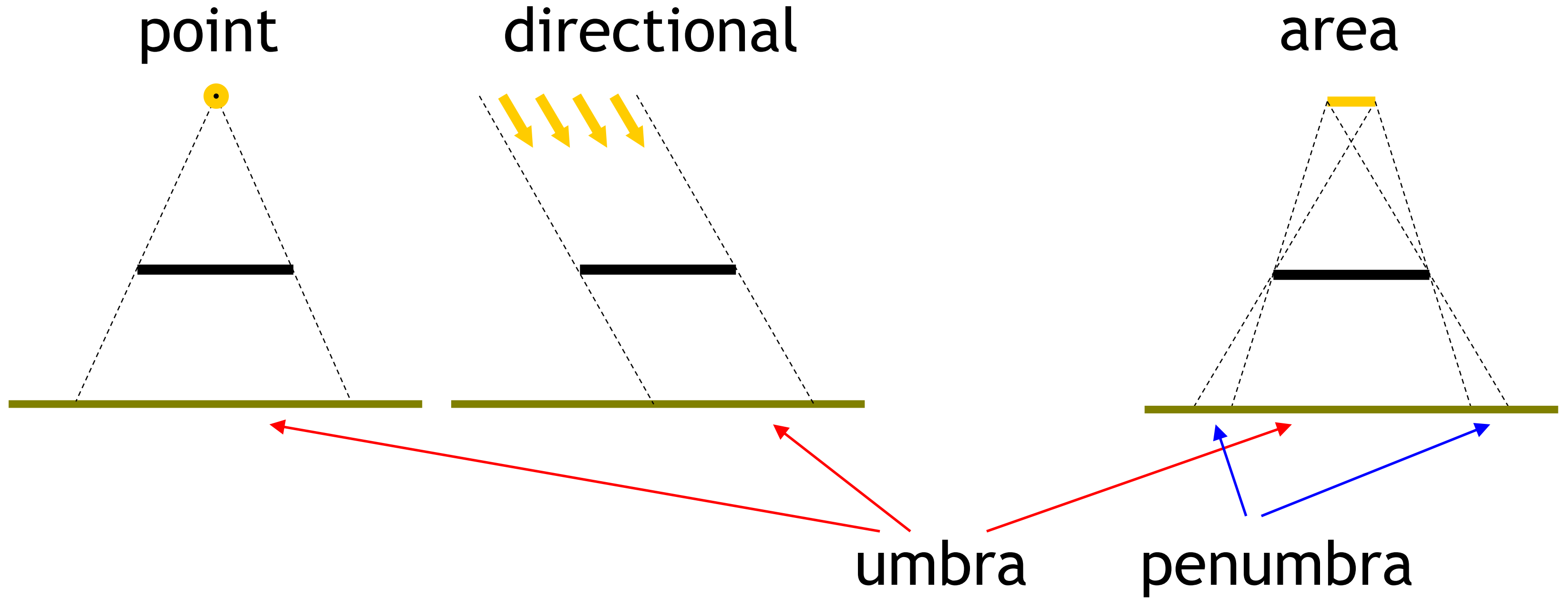
Terminology

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

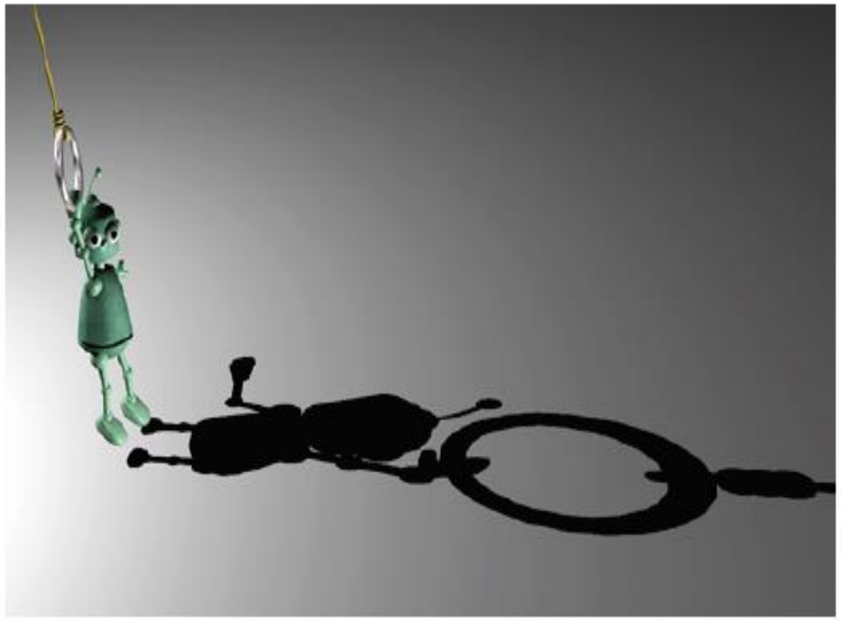


Hard and soft shadows

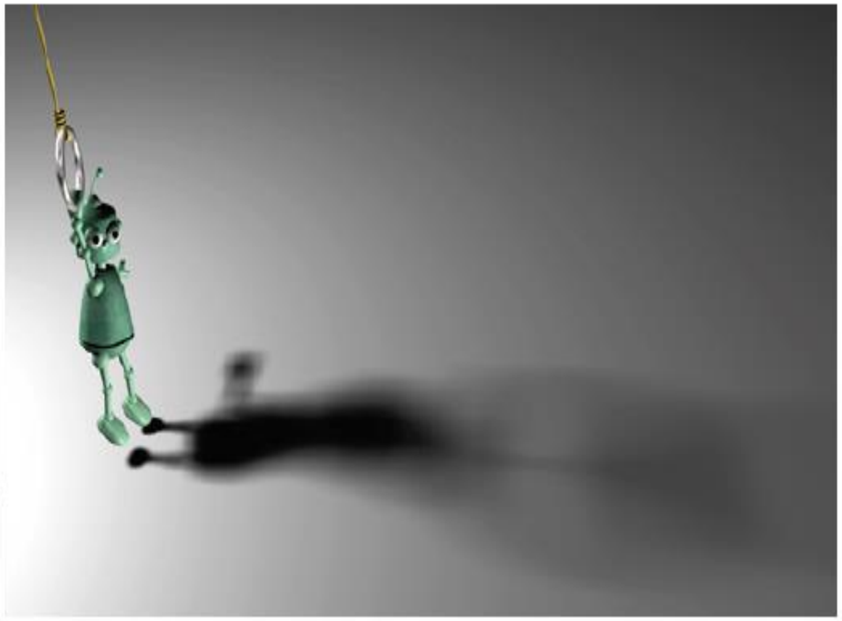
- Point and directional lights lead to hard shadows, no penumbra
- Area light sources lead to soft shadows, with penumbra



Hard and soft shadows



Hard shadow,
point light source



Soft shadow,
area light source

Shadows for interactive rendering

- Focus on hard shadows
  - Soft shadows often too hard to compute in interactive graphics
- Two main techniques
  - Shadow mapping
  - Shadow volumes
- Many variations, subtleties
- Still active research area

Today

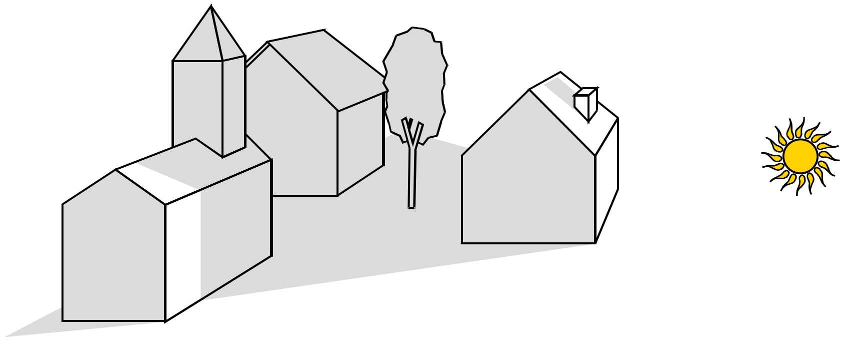
- Bump mapping
- Shadows
- Shadow mapping
- Shadow mapping in OpenGL

Shadow mapping

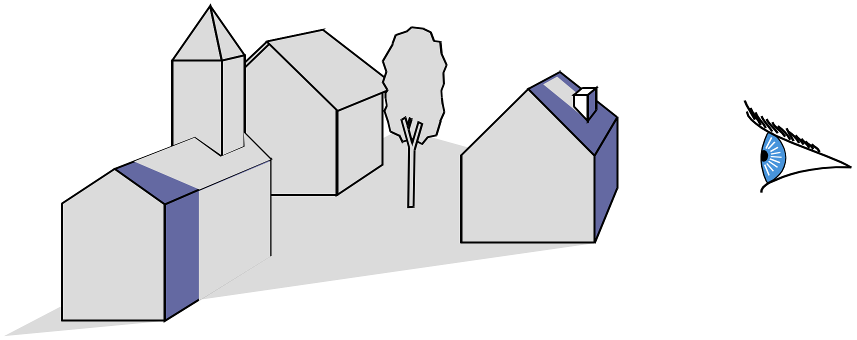
http://en.wikipedia.org/wiki/Shadow_mapping

Main idea

- Scene point is lit by light source if it is visible from light source
- Determine visibility from light source by placing camera at light source position and rendering scene



Scene points are lit if visible from light source

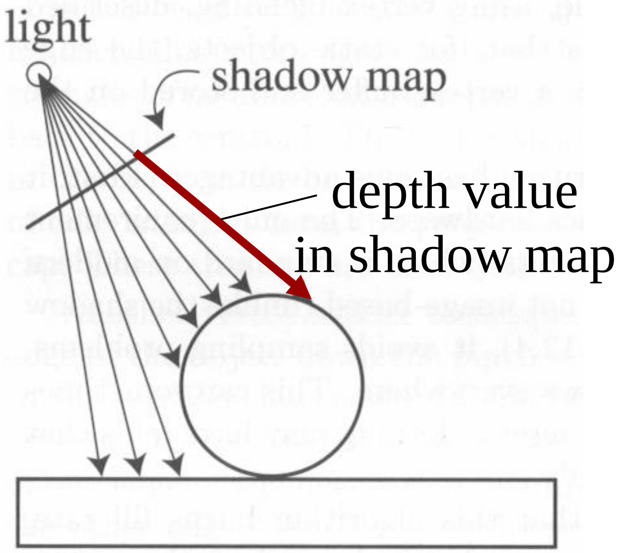


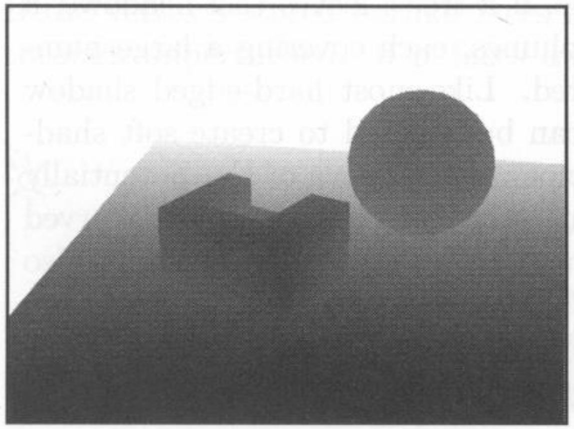
Determine visibility from light source by placing camera at light source position

Two pass algorithm

First pass

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



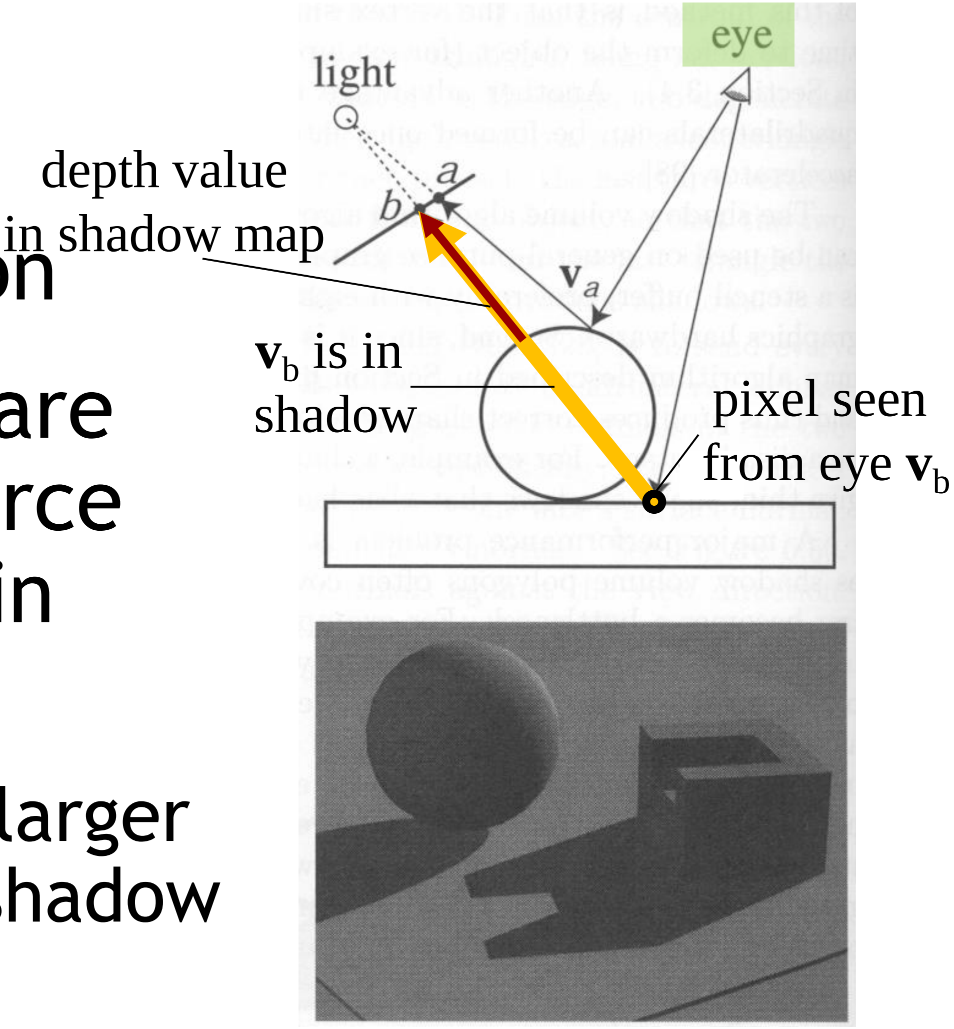


Depth image seen from light source

Two pass algorithm

Second pass

- Render scene from camera (eye) position
- At each pixel, compare distance to light source (yellow) with value in shadow map (red)
  - If yellow distance is larger than red, we are in shadow
  - If distance is smaller or equal, pixel is lit


eye
light
depth value in shadow map
a
b
$\mathbf{v}_a$
$\mathbf{v}_b$ is in shadow
pixel seen from eye $\mathbf{v}_b$

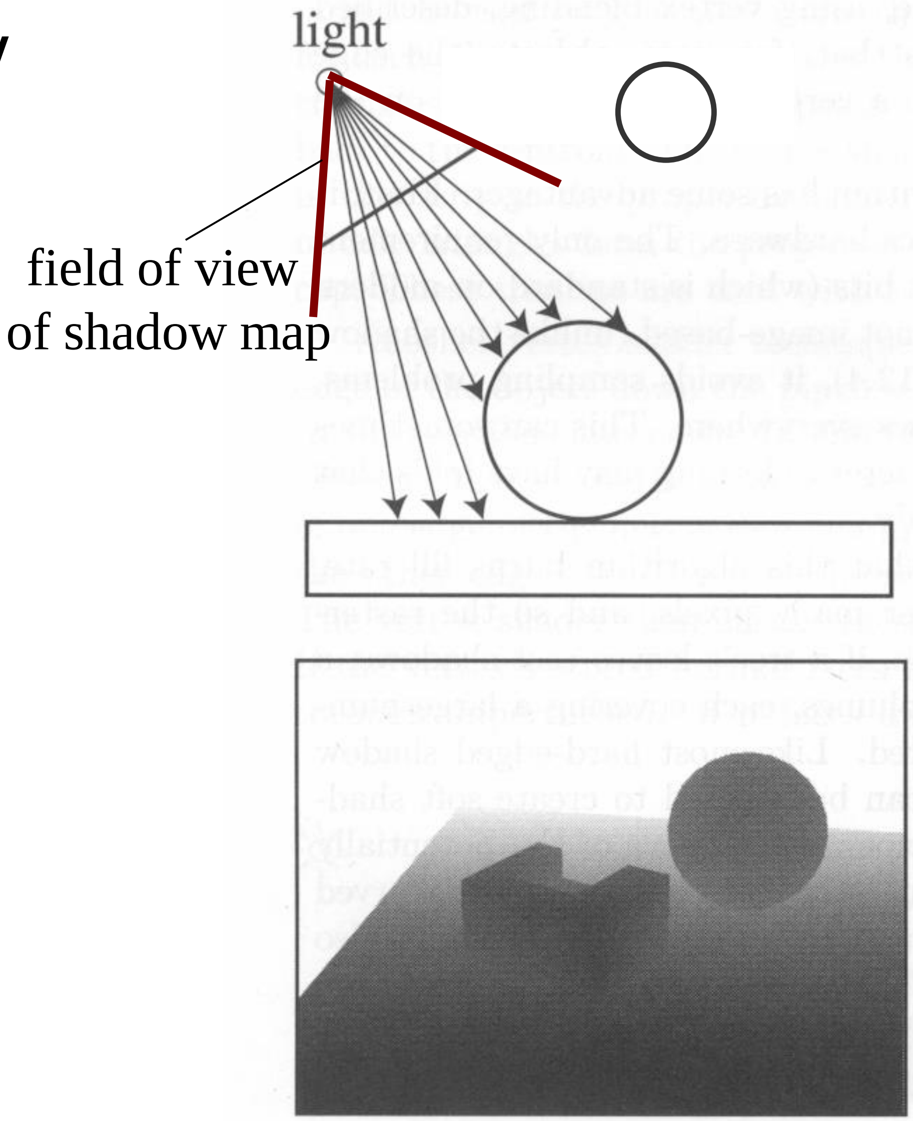
Final image with shadows

Issues

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

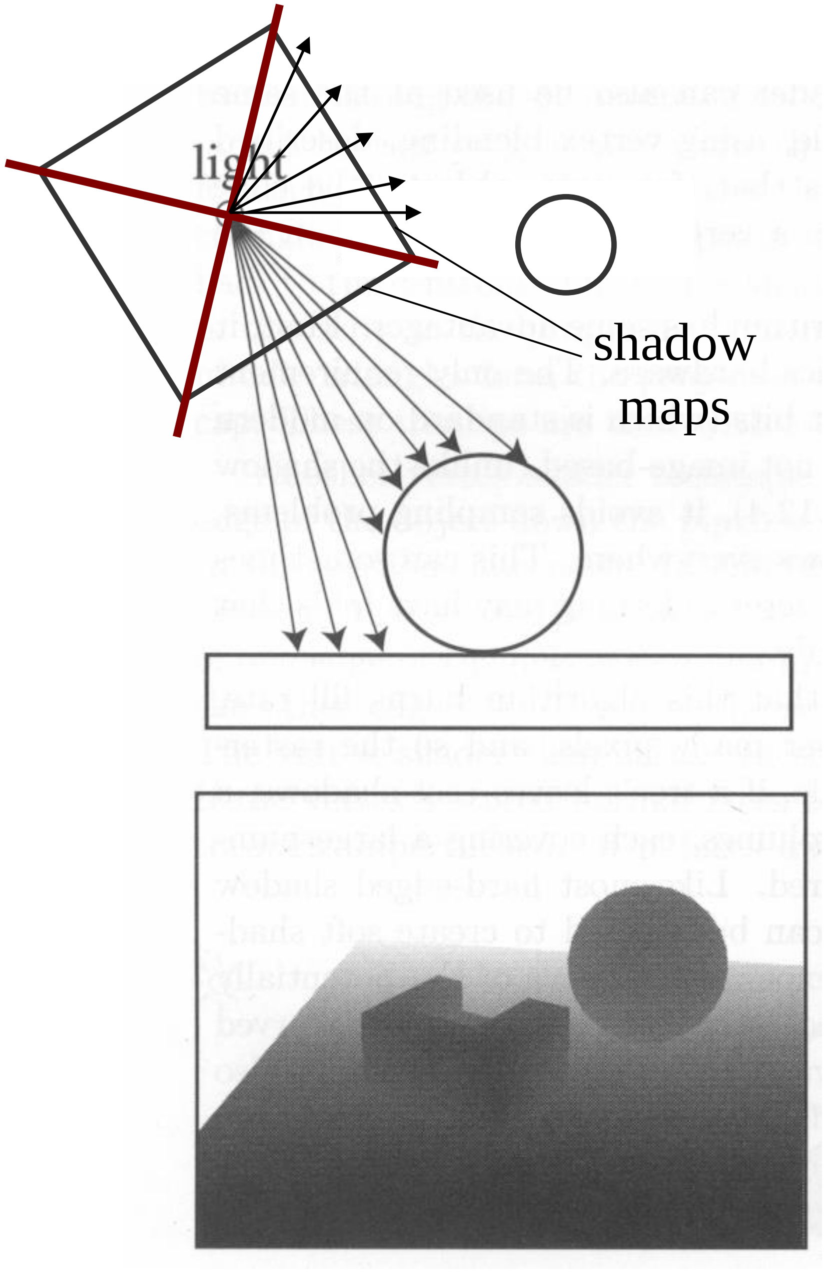
Limited field of view

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



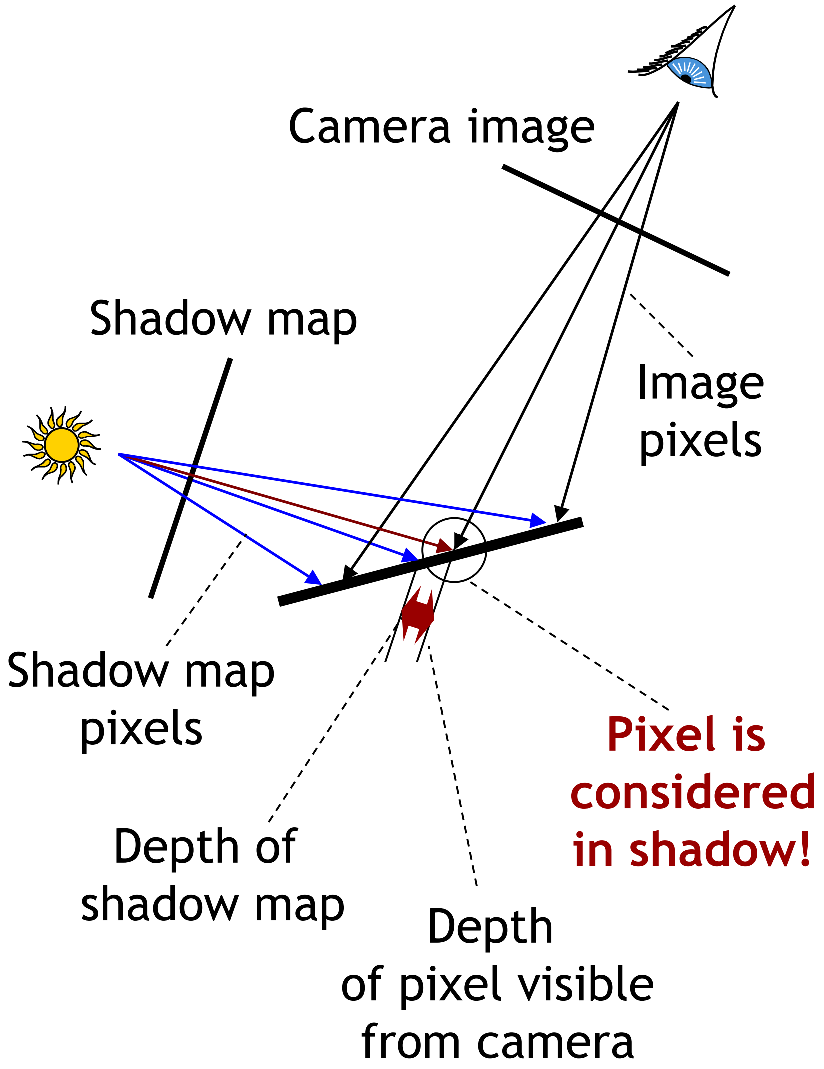
Limited field of view

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



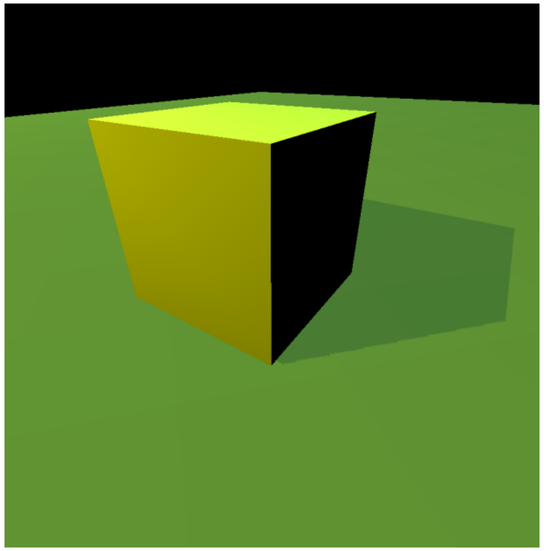
z-fighting

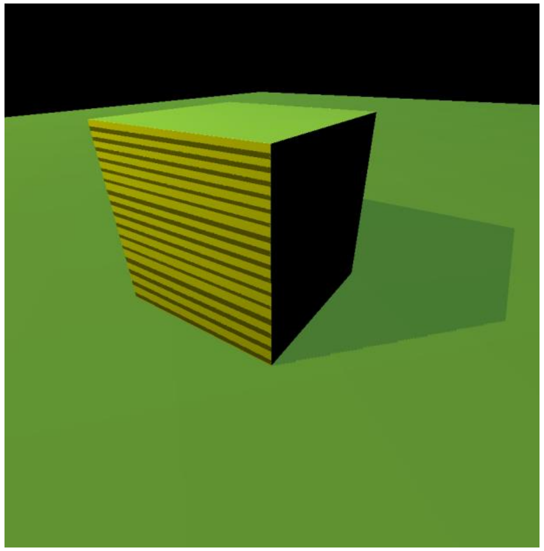
- In theory, depth values for points visible from light source are equal in both rendering passes
- Because of limited resolution, depth of pixel visible from camera could be larger than shadow map value
- Need to add bias in first pass to make sure pixels are lit

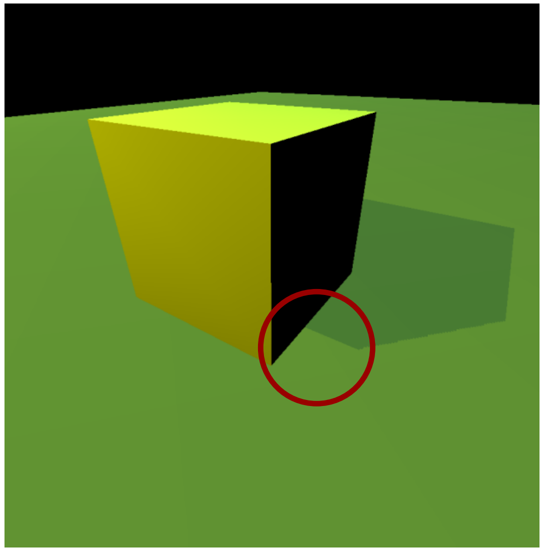


Solution

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

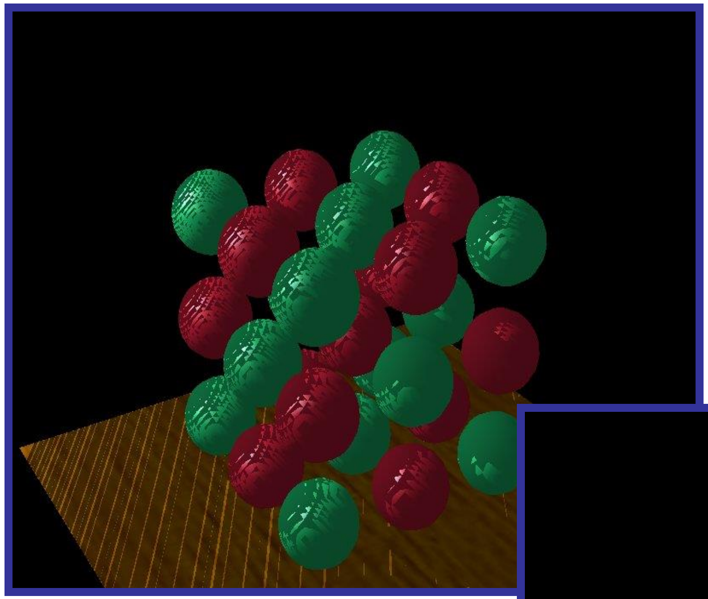

Correct bias


Not enough bias

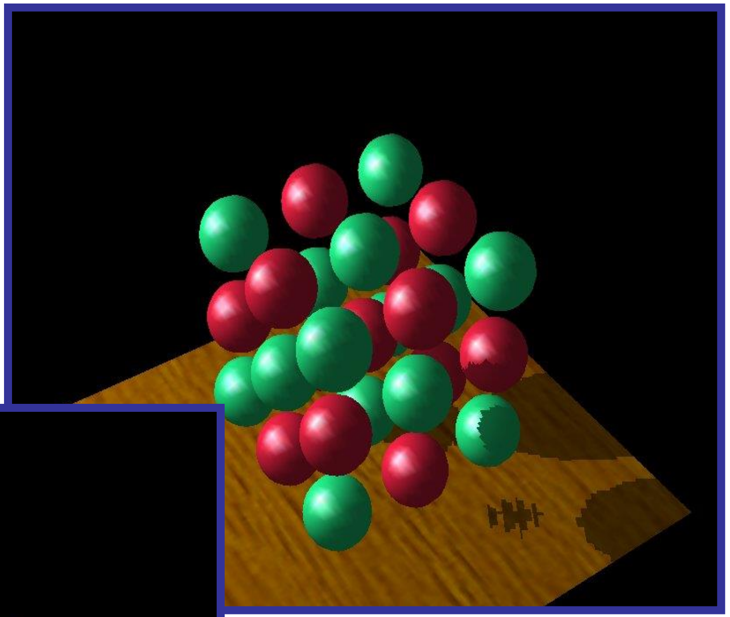

Too much bias

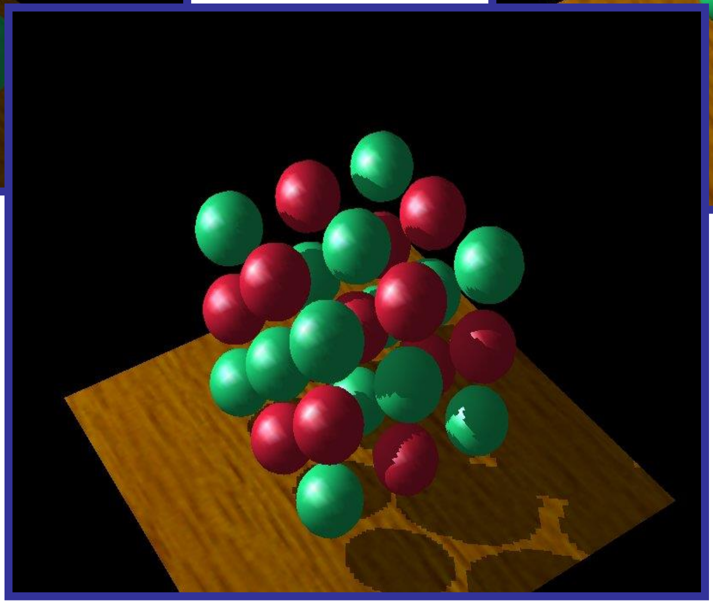
Bias

Not enough



Too much

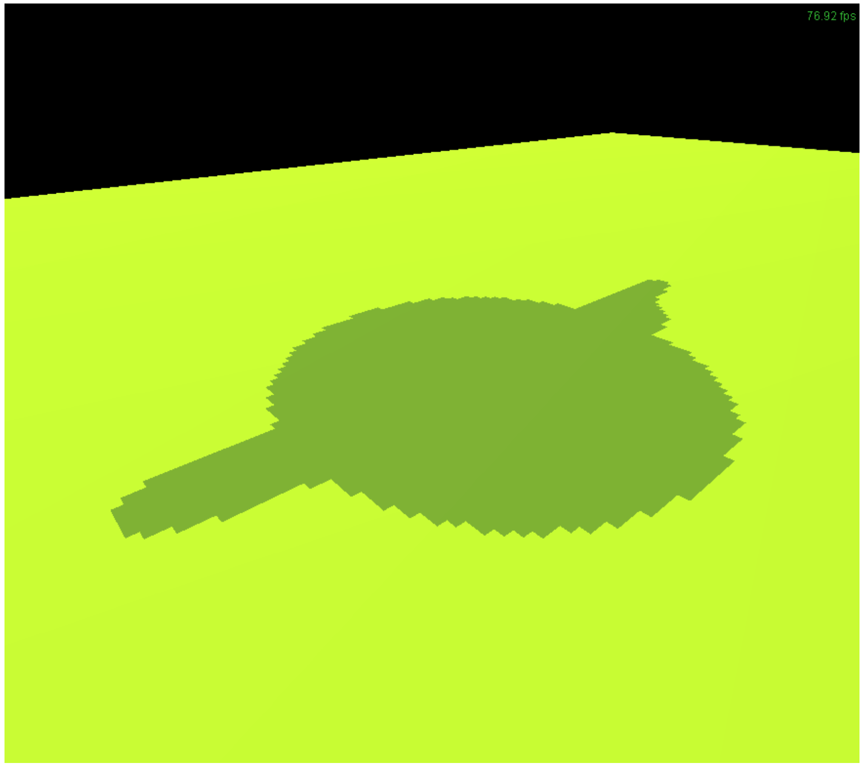


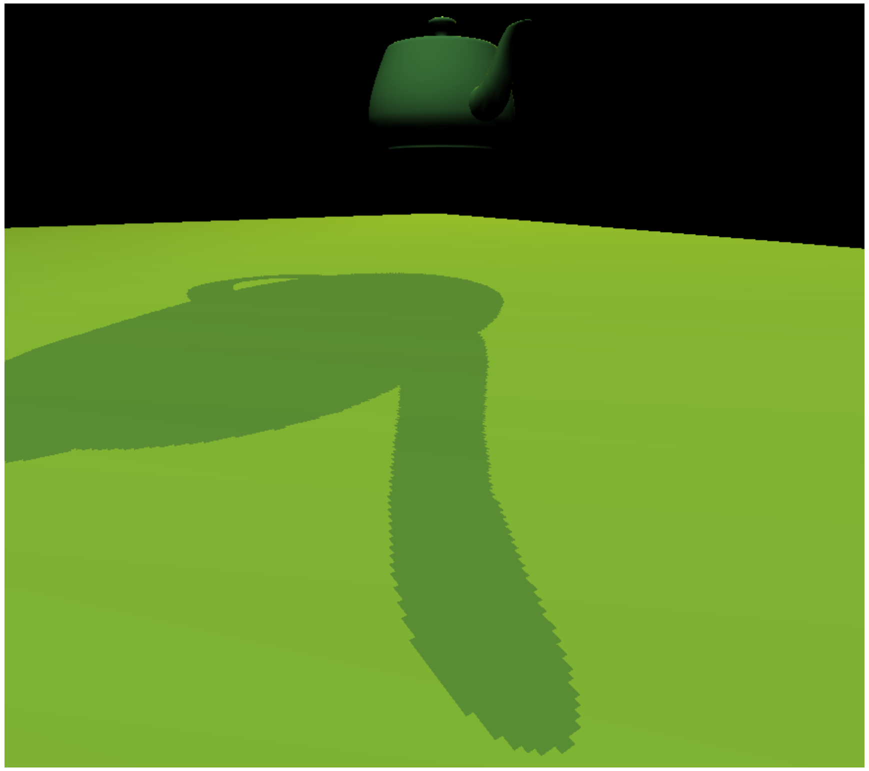


Correct

Sampling problems

- Shadow map pixel may project to many image pixels
- Ugly stair-stepping artifacts

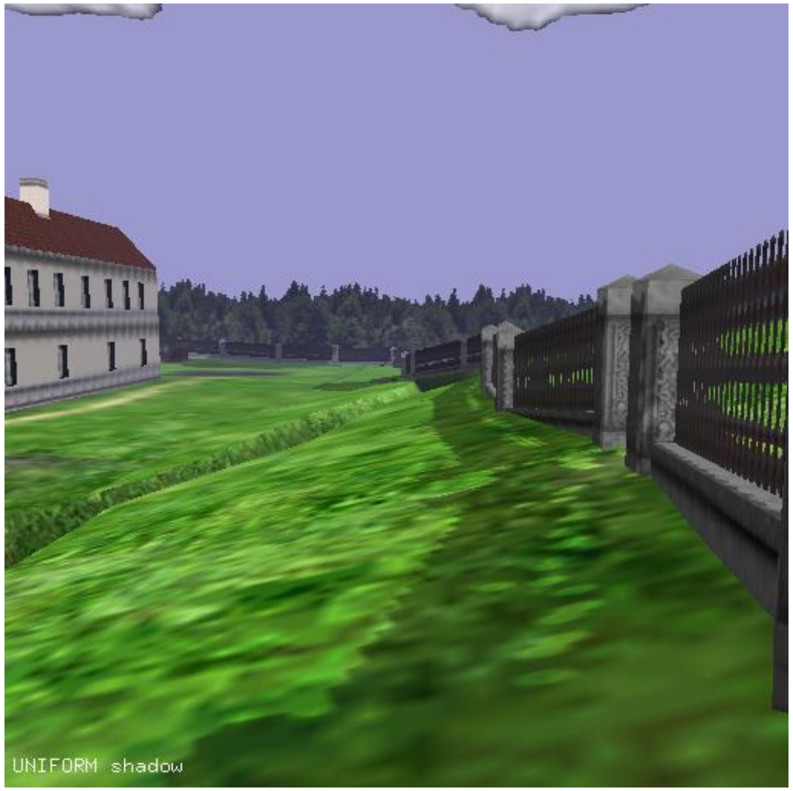




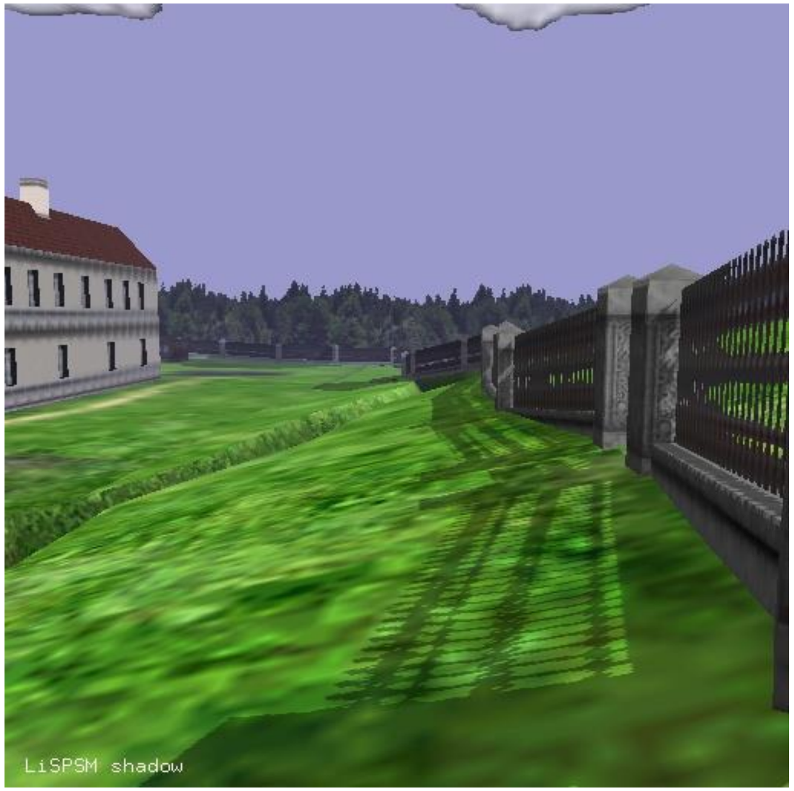
Solutions

- Increase resolution of shadow map
  - Not always sufficient
- Split shadow map into several slices
- Tweak projection for shadow map rendering
  - Light space perspective shadow maps (LiSPSM) http://www.cg.tuwien.ac.at/research/vr/lispsm/
  - With GLSL source code!
- Combination of splitting and LiSPSM
  - Basis for most serious implementations
  - List of advanced techniques see http://en.wikipedia.org/wiki/Shadow_mapping

LiSPSM



Basic shadow map

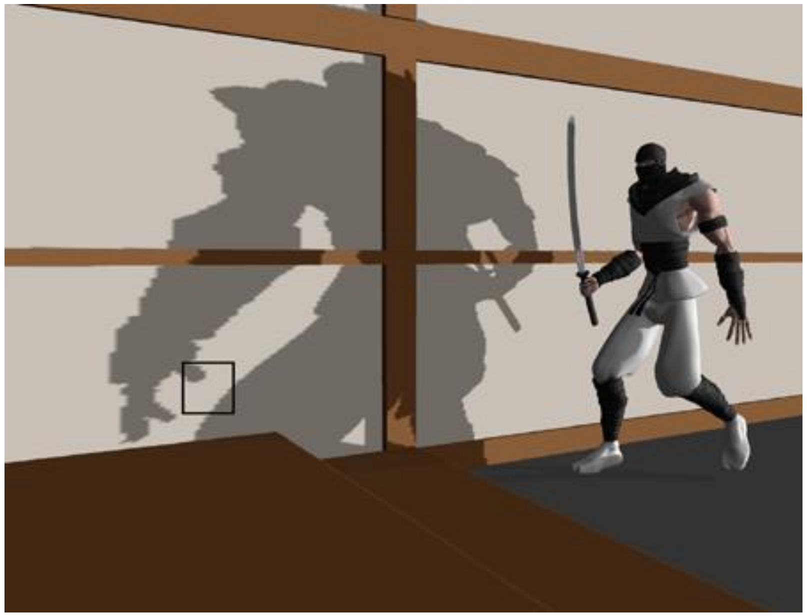


Light space perspective shadow map

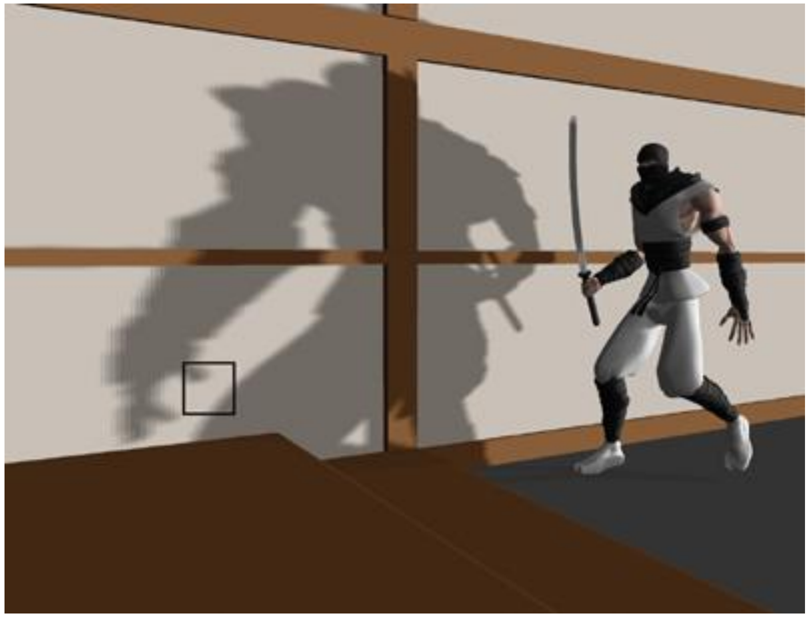
Percentage closer filtering

- Goal: avoid stair-stepping artifacts
- Similar to texture filtering, but with a twist

Simple shadow mapping



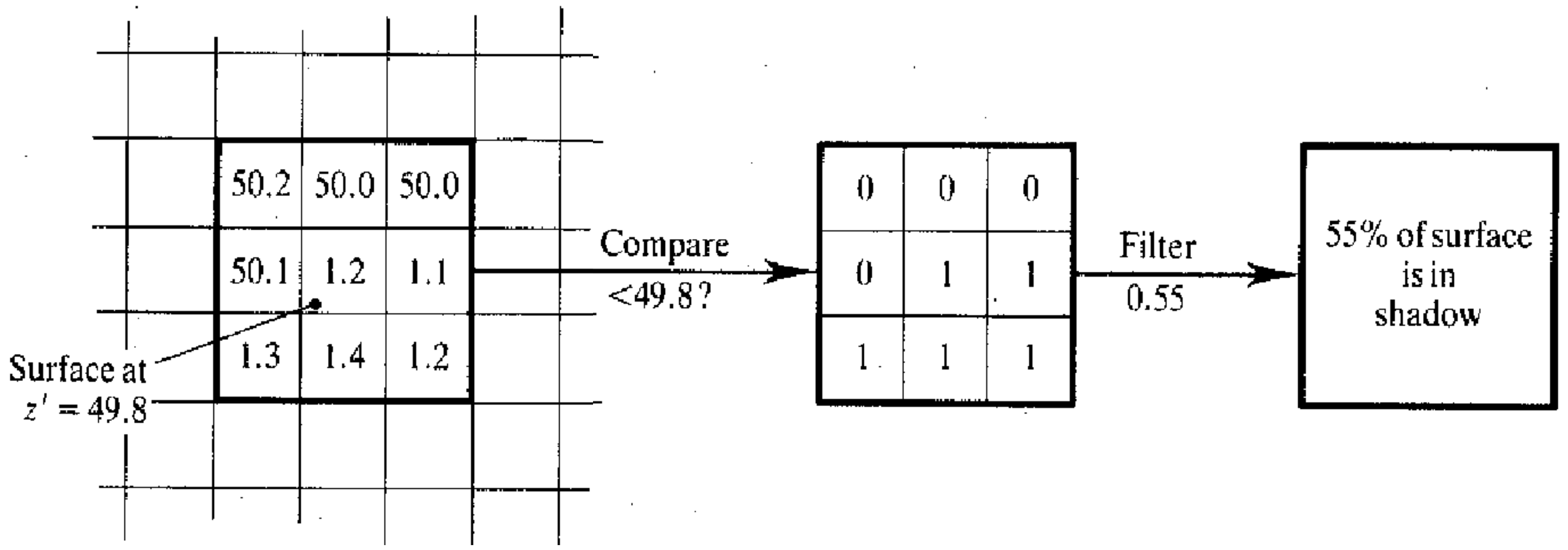
Percentage closer filtering



http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html

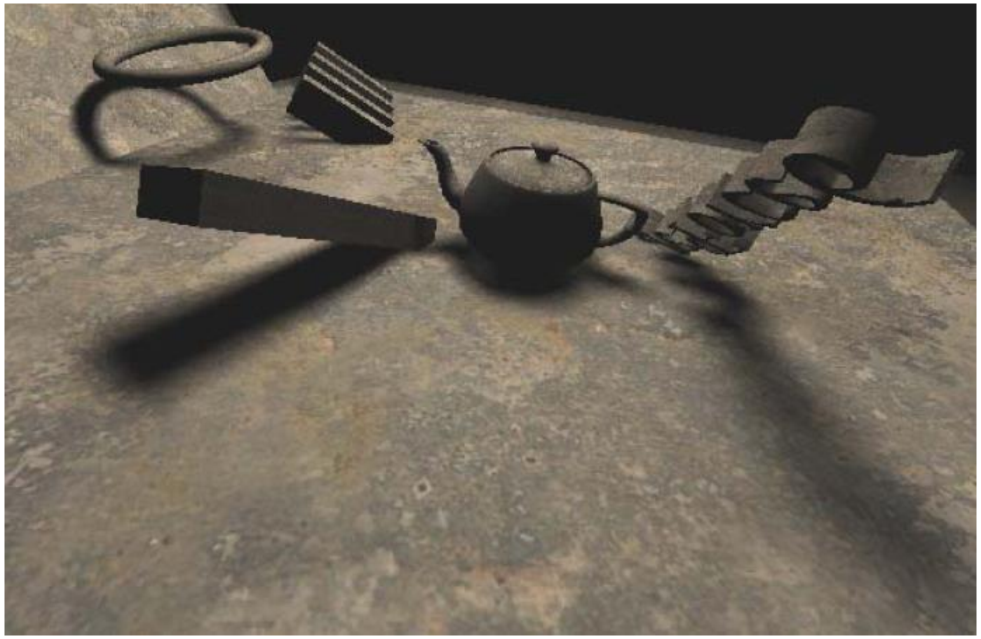
Percentage closer filtering

- Instead of looking up one shadow map pixel, look up several
- Perform depth test for each shadow map pixel
- Compute percentage of lit shadow map pixels



Percentage closer filtering

- Supported in hardware for small filters (2x2 shadow map pixels)
- Can use larger filters (look up more shadow map pixels) at cost of performance penalty
- Fake soft shadows
  - Larger filter, softer shadow boundary

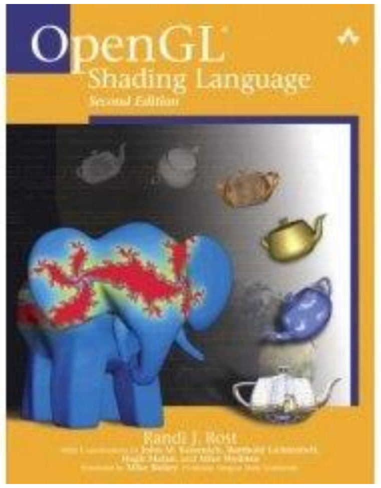


Today

- Bump mapping
- Shadows
- Shadow mapping
- Shadow mapping in OpenGL

Shadow mapping with OpenGL

- Recommended book: „OpenGL Shading Language“ by Randi Rost



First pass

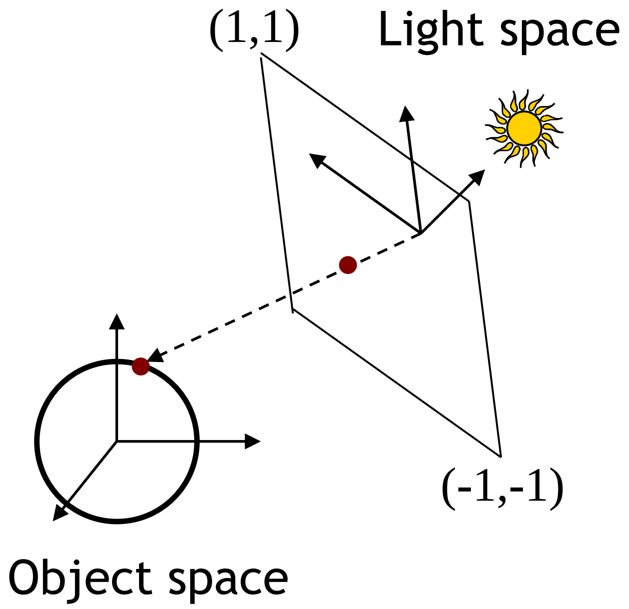
- Render scene by placing camera at light source position
- Compute light view (look at) matrix
  - Similar to computing camera matrix from look-at, up vector
  - Compute its inverse to get world-to-light transform
- Determine view frustum such that scene is completely enclosed
  - Use several view frusta/shadow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

First pass

- Each vertex point is transformed by

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

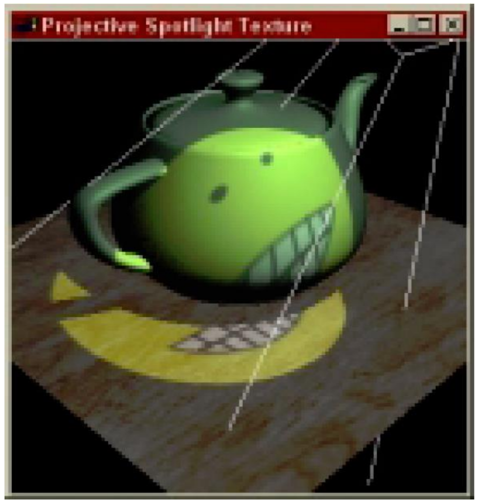
  - Object-to-world (modeling) matrix $\mathbf{M}$
  - World-to-light space matrix $\mathbf{V}_{light}$
  - Light frustum (projection) matrix $\mathbf{P}_{light}$



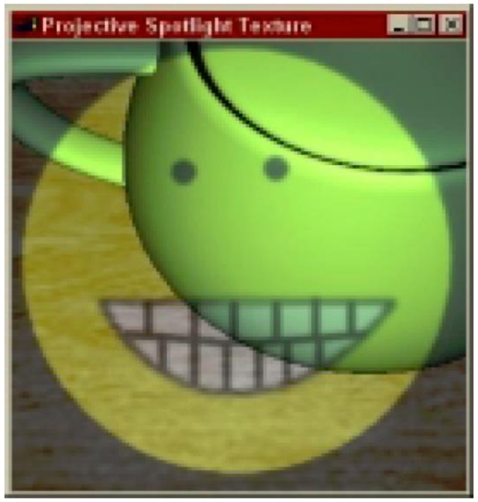
- Remember: points within frustum are transformed to unit cube $[-1,1]^3$ by projection matrix $\mathbf{P}_{light}$

First pass

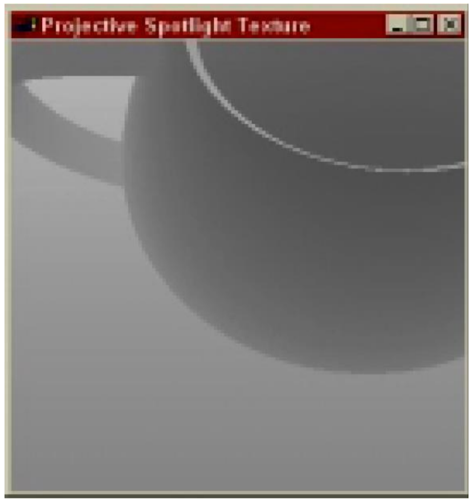
- Use glPolygonOffset to apply depth bias
- Store depth image in a texture
  - Use glCopyTexImage with internal format GL_DEPTH_COMPONENT



Final result
with shadows



Scene rendered
from light source



Depth map
from light source

Second pass

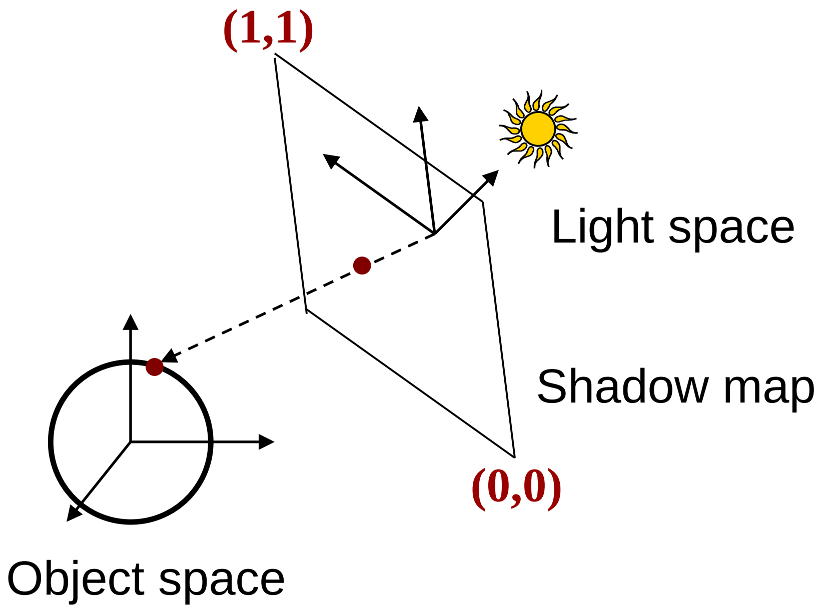
- Render scene from camera
- At each pixel, look up corresponding location in shadow map
- Compare depths with respect to light source
- Shade accordingly

Looking up shadow map

- 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 (set it as texture matrix, see below)

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

- After perspective projection we have shadow map coordinates



Looking up shadow map

- 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 fragment shader
- 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

- Compute matrix **T** and pass to shader as uniform

In vertex shader

- Declare and access matrix **T** as uniform
- Multiply vertex positions with **T** and pass result to fragment shader

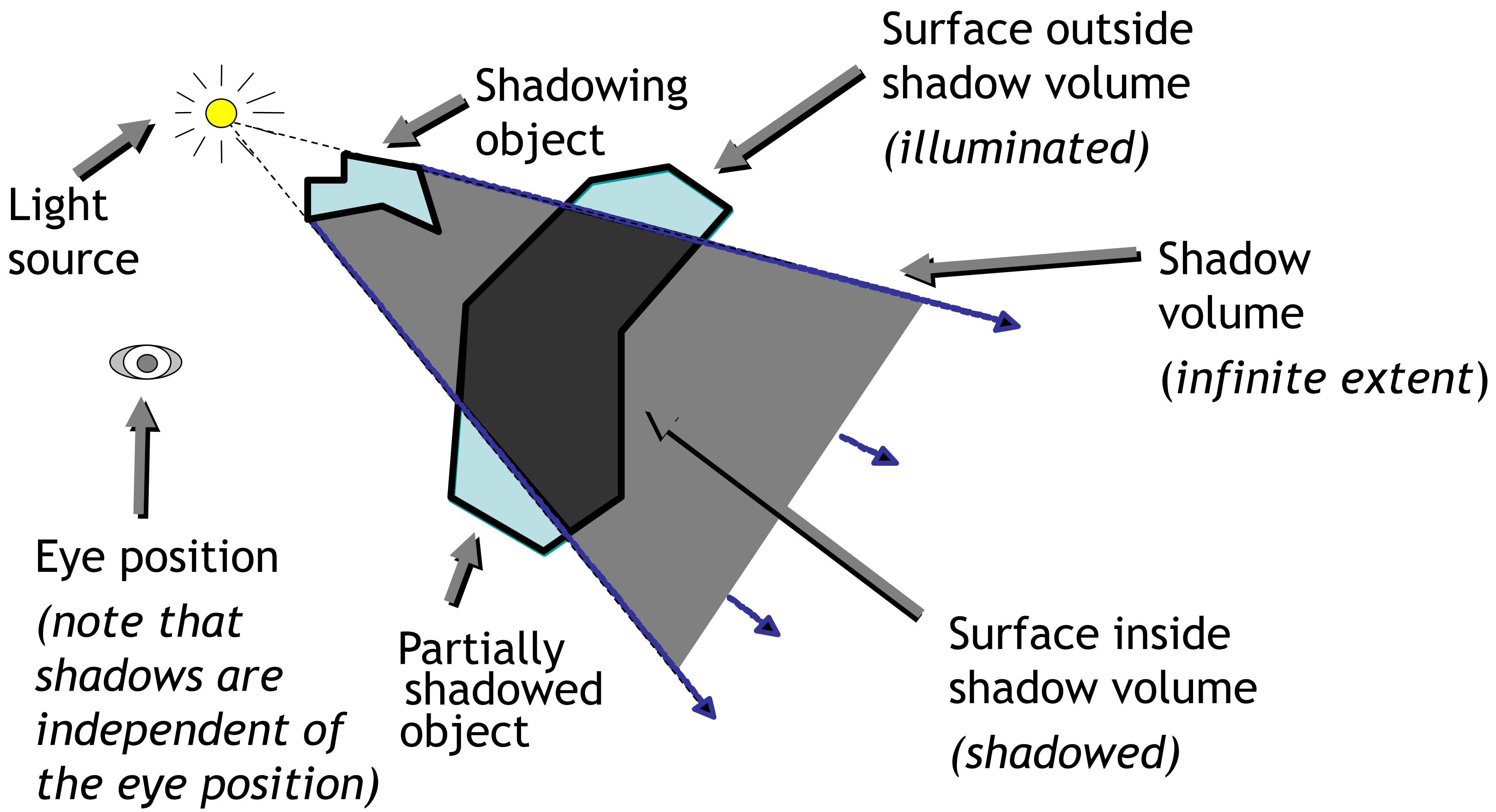
In fragment shader

- Declare shadow map as sampler2DShadow
- Look up shadow map using projective texturing with
  vec4 textureProj(sampler2D, vec4, float bias)

GLSL 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

Shadow volumes

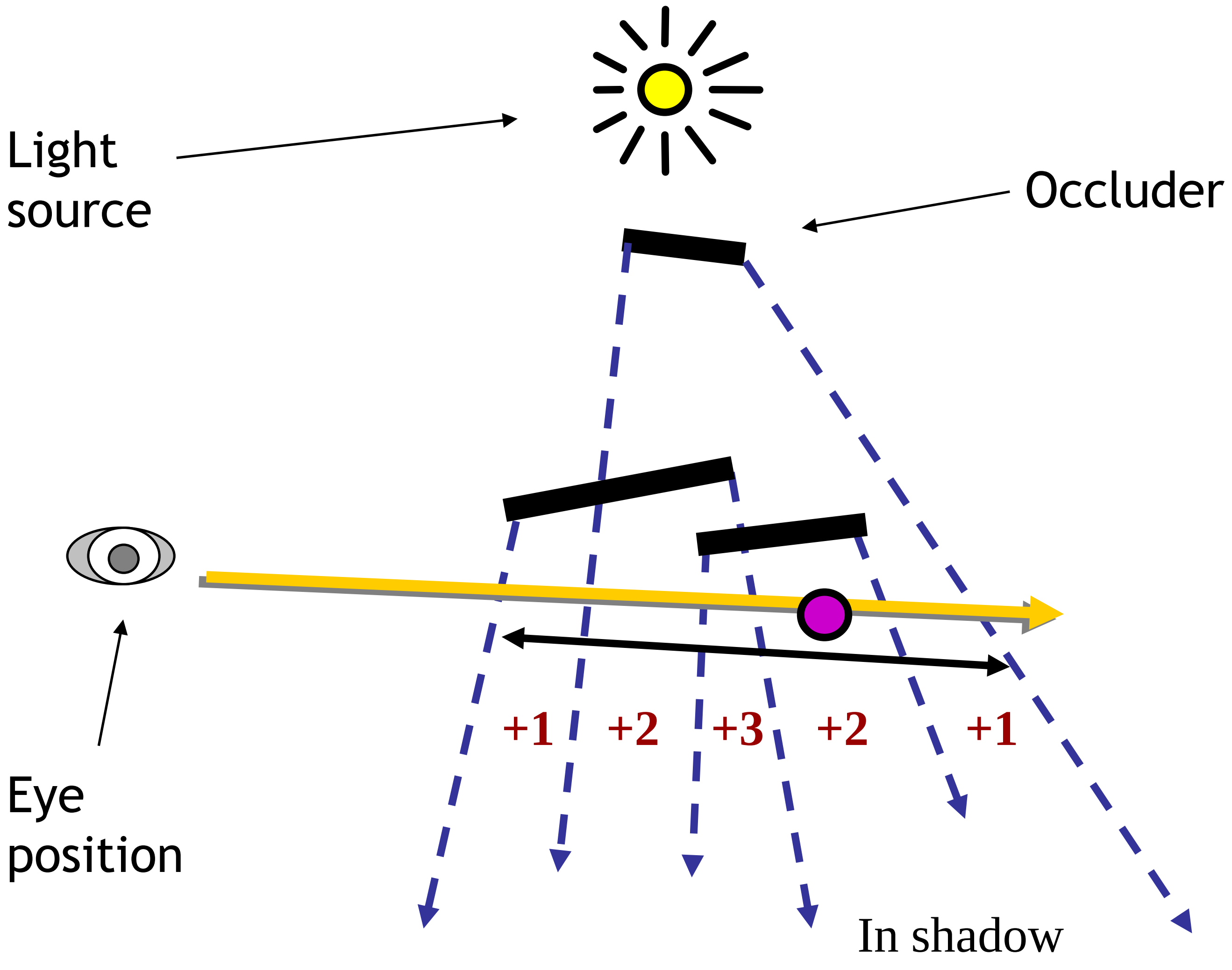

Light source
Shadowing object
Surface outside shadow volume *(illuminated)*
Shadow volume (*infinite extent*)
Eye position *(note that shadows are independent of the eye position)*
Partially shadowed object
Surface inside shadow volume *(shadowed)*

In shadow or not

- Test if surface visible in given pixel is inside or outside shadow volume

1. Allocate a counter per pixel
2. Cast a ray into the scene, starting from eye, going through given pixel
3. Increment the counter when the ray enters the shadow volume
4. Decrement the counter when the ray leaves the shadow volume
5. When we hit the object, check the counter.
    - If counter > 0, in shadow
    - Otherwise, not in shadow

In shadow or not

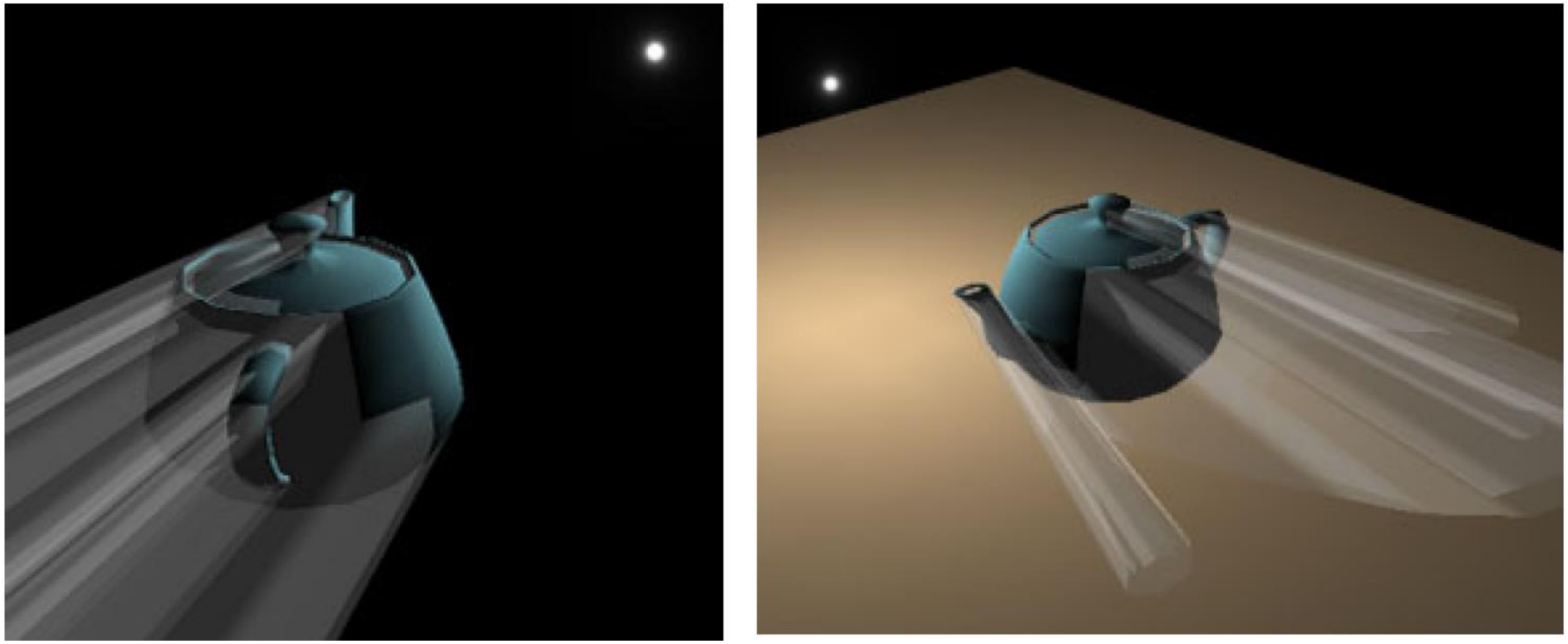


Implementation in rendering pipeline

- Ray tracing not possible to implement directly
- Use a few tricks...

Shadow volume construction

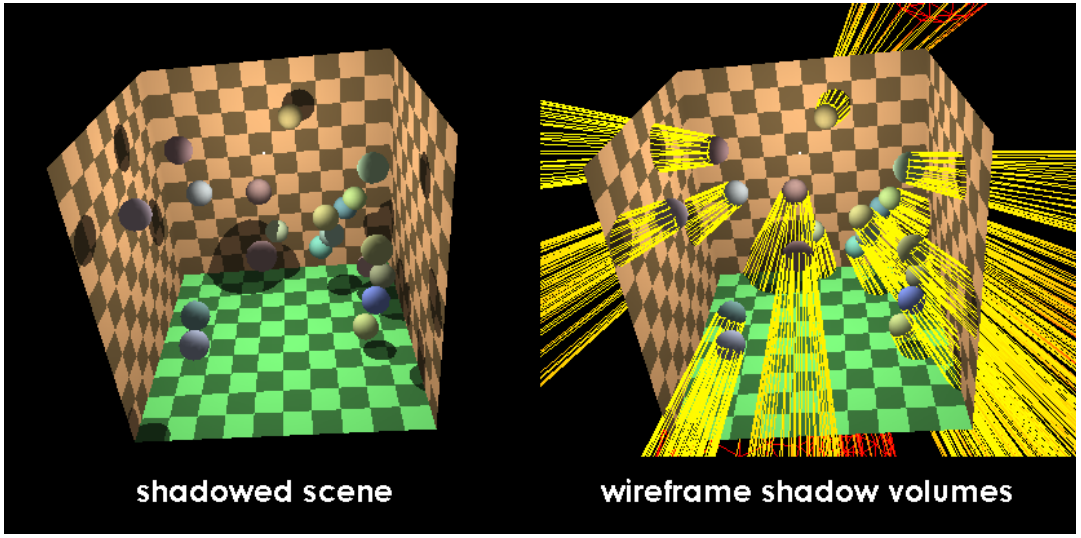
- Need to generate shadow polygons to bound shadow volume
- Extrude silhouette edges from light source



Extruded shadow volumes

Shadow volume construction

- Needs to be 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



Shadow test without ray tracing

Using the stencil buffer

- A framebuffer channel (like RGB colors, depth) that contains a per-pixel counter (integer value)
- Available in OpenGL
- Stencil test
  - Similar to depth test (z-buffering)
  - Control whether a fragment is discarded or not
  - Stencil function: is evaluated to decide whether to discard a fragment
  - Stencil operation: is performed to update the stencil buffer depending on the result of the test

Shadow volume algorithms

Z-pass approach

- Count leaving/entering shadow volume events as described
- Use stencil buffer to count number of visible (i.e. not occluded from camera) front-facing and back facing shadow volume polygons for each pixel
- If equal, pixel is not in shadow

Z-fail approach

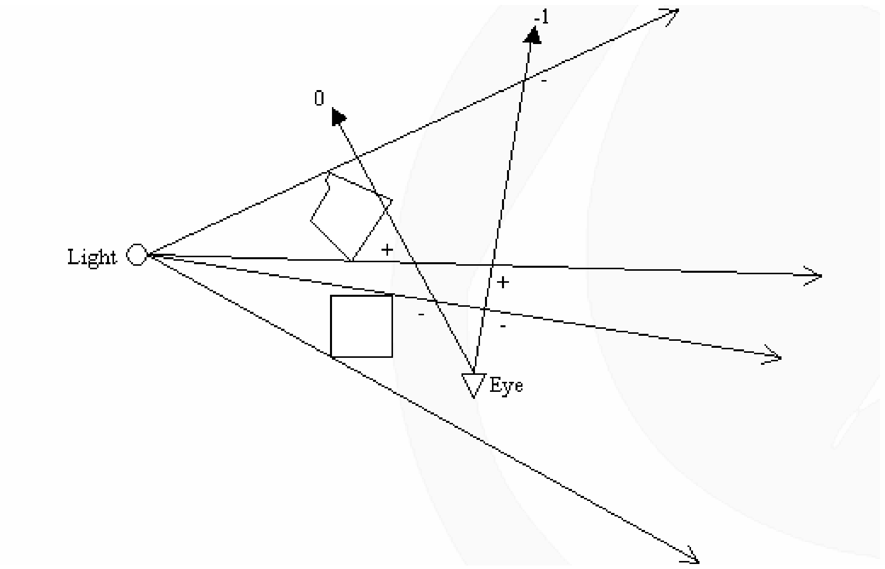
- Count number of invisible (i.e. occluded from camera) front-facing and back-facing shadow volume polygons
- If equal, pixel is not in shadow

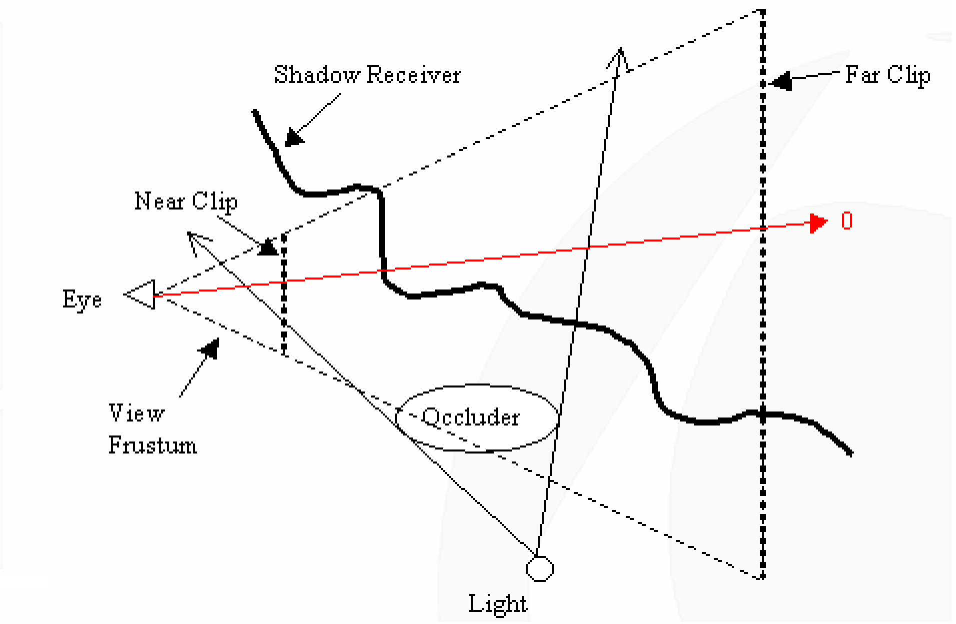
Z-pass approach: details

- Render scene with only ambient light
  - Update depth buffer
- Turn off depth and color write, turn on stencil, keep the depth test on
- Init stencil buffer to 0
- Draw shadow volume twice using face culling
  - 1st pass: render front faces and increment stencil buffer when depth test passes
  - 2nd pass: render back faces and decrement when depth test passes
- At each pixel
  - Stencil != 0, in shadow
  - Stencil = 0, lit
- Render the scene again with diffuse and specular lighting
  - Write to framebuffer only pixels with stencil = 0

Issues

- Z-pass fails if
  - Eye is in shadow
  - Shadow polygon clipped by near clip plane





Shadow volumes

- Pros
  - Does not require hardware support for shadow mapping
  - Pixel accurate shadows, no sampling issues
- Cons
  - More CPU intensive (construction of shadow volume polygons)
  - Fill-rate intensive (need to draw many shadow volume polygons)
  - Expensive for complex geometry
  - Tricky to handle all cases correctly
  - Hard to extend to soft shadows

Shadow maps

- Pros:
  - Little CPU overhead
  - No need to construct extra geometry to represent shadows
  - Hardware support
  - Can fake soft shadows easily
- Cons:
  - Sampling issues
  - Depth bias is not completely foolproof
- Shadow mapping has become more popular with better hardware support

Resources

- 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://developer.nvidia.com/object/robust_shadow_volumes.html
  http://www.gamedev.net/reference/articles/article1873.asp

Next time

- Advanced topics, outlook