EECS 487: Interactive Computer Graphics

Lecture 30:
- Environment mapping
- Radiance map
- Accumulation buffer

Environment Mapping

The key to depicting a shiny-looking material is to provide something for it to reflect
- proper reflection requires ray tracing, expensive
- can be simulated with a pre-rendered environment, stored as a texture
- imagine object is enclosed in an infinitely large sphere or cube
- rays are bounced off object into environment to determine color

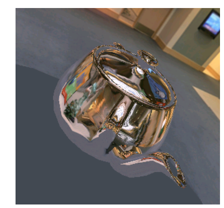
EM*surface color = reflection mapping

Environment Mapping

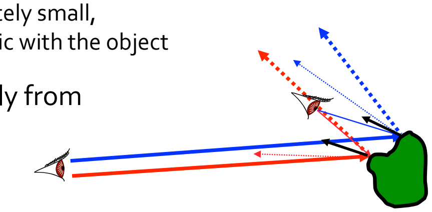
Steps:
- load environment map
- for each reflective pixel, compute its normal
- compute the reflection vector from the normal and view vectors
- use the reflection vector to compute an index into the environment map in the reflection direction
  - note: we're not computing ray intersection between the reflection vector and the environment!
- use the texel at the index to color the pixel

Shortcomings:
- no inter-object reflection
  ⇒ works well when there's just a single object
- no self-reflection

Environment Mapping



Model:
- environment is infinitely far away
- all reflections as seen from the same, far away view point:
  - object approximated as an infinitely small, perfectly mirroring ball concentric with the object
- reflected color computed only from the direction of reflection
  - not from the position on surface
  - determined by surface normal
  - no ray-environment intersection computation



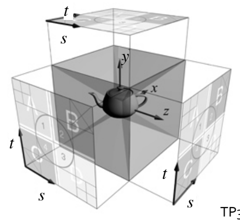
Cube Map

Most popular and fastest: easy to produce with rendering system or by photography from center of object, once for each side of cube

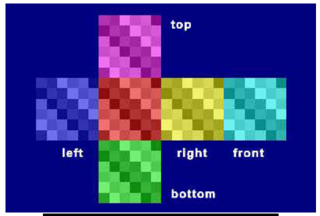
Simple texture-coordinates calculation

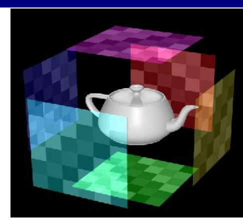
Texture creation from scene:
- view independent
- uniform sampling/resolution

Supports bilinear filtering and mipmapping

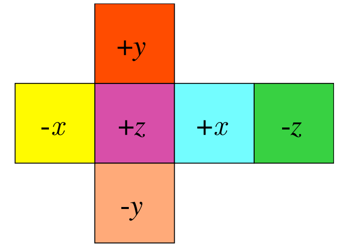


TP3





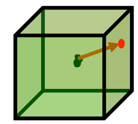
Computing Reflection

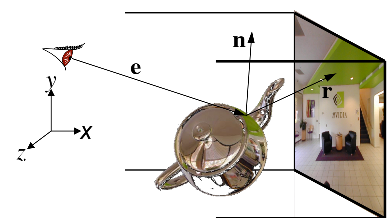


Steps:

1. compute reflection vector, $\mathbf{r}$
   - $\mathbf{e}$ from eye to vertex
   - $\mathbf{n}$ normal in eye coordinates
   - $\mathbf{r} = \mathbf{e} - 2\mathbf{n}(\mathbf{n} \cdot \mathbf{e})$
2. reflection is a function of one direction: largest absolute value of $\mathbf{r}$'s components determines the cube face to reflect
   - example: $\mathbf{r} = (5, -1, 2)$ gives $+x$ as the reflected face
3. divide $\mathbf{r}$ by the value of the "reflection" coordinate (5) and map to $[0,1]$:

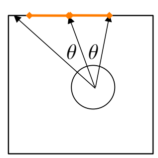
$$
\begin{aligned}
(s,t) &= ((y + x)/2x, (z + x)/2x) \\
&= ((-1/5 + 1)/2, (2/5 + 1)/2) \\
&= (0.4, 0.7)
\end{aligned}
$$





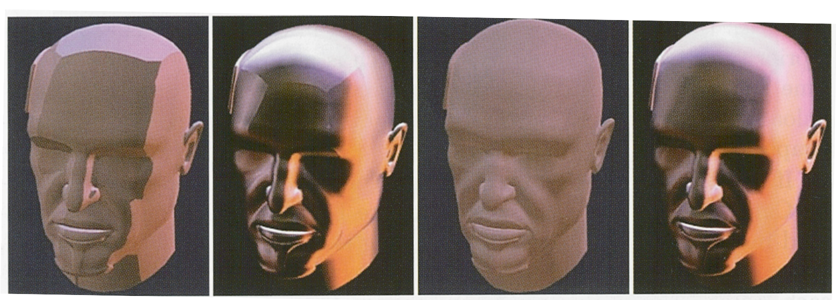
Cube Map: Disadvantages

Angular size of texel varies across a cube face



Usually doesn't interpolate across cube faces
⇒ cube edges are reflected on object

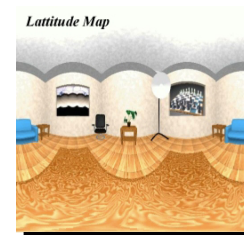
Tools such as AMD's *CubeMapGen* can fix these problems



RTR3

Methods to Create EM

- Cube map
- Latitude/longitude projections map
  - created by painting
  - oversampling of poles compared to the equator
- Spherical map
  - gazing ball
  - fisheye lens
- Parabolic map



Gazing Ball (Light Probe)

Created by photographing a reflective sphere

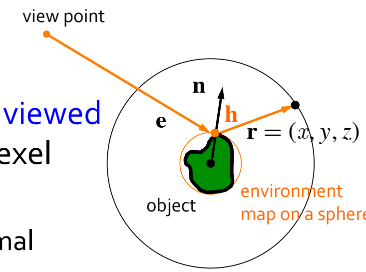
Maps all directions to a circle

Reflection indexed by normal

Texture creation from scene:
- resolution function of orientation
- view dependent: must regenerate EM when camera moves or will see the same thing

Hanrahan09

Sphere Mapping



Use a texture map of a sphere viewed from infinity use $\mathbf{r}$ to look up texel
- the eye vectors are parallel
- $\mathbf{r}$ determined only by surface normal

Want: compute texture coordinates (s, t) from $\mathbf{r}$
- texture is not really pasted to the inside of the environment sphere, but projected (next slide)
- object can be approximated as an infinitely small, perfectly mirroring ball concentric with the object
- map the normals of an object to the corresponding normals of a sphere

Zhang08

Computing (s, t) from $\mathbf{r}$

Observation: $\mathbf{n}$ can be expressed in terms of $\mathbf{r}$ and $\mathbf{e}$:

$$\mathbf{r} = \mathbf{e} - 2\mathbf{n}(\mathbf{n} \cdot \mathbf{e})$$

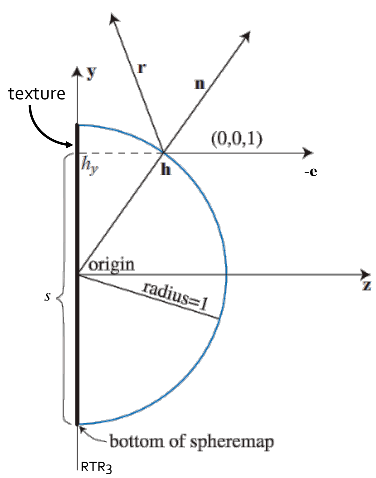
$$\mathbf{r} - \mathbf{e} = \alpha\mathbf{n}$$

$$\alpha\mathbf{n} = \mathbf{r} - \mathbf{e} = \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}$$

$$\frac{\alpha\mathbf{n}}{\|\alpha\mathbf{n}\|} = \begin{bmatrix} x/p \\ y/p \\ (z+1)/p \\ 0 \end{bmatrix}$$

where $\|\alpha\mathbf{n}\| = p = \sqrt{x^2 + y^2 + (z+1)^2}$

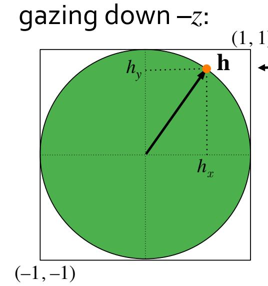
since $|\mathbf{r}| = 1$, $p = \sqrt{2(z+1)}$



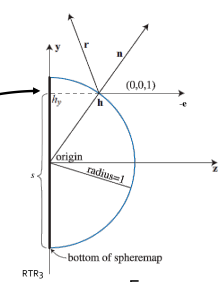
Zhang08

Computing (s, t) from $\mathbf{r}$

unit sphere in eye space, gazing down $-z$:



texture



$$\mathbf{h} = \begin{bmatrix} h_x \\ h_y \\ \sqrt{1 - h_x^2 - h_y^2} \\ 0 \end{bmatrix}$$

$$\frac{\alpha\mathbf{n}}{\|\alpha\mathbf{n}\|} = \begin{bmatrix} x/p \\ y/p \\ (z+1)/p \\ 0 \end{bmatrix}$$

where $p = \sqrt{2(z+1)}$

$h_x = x/p \qquad h_y = y/p$

scaled to texture space [0,1]:

$$s = \frac{1}{2}h_x + \frac{1}{2} \qquad t = \frac{1}{2}h_y + \frac{1}{2}$$

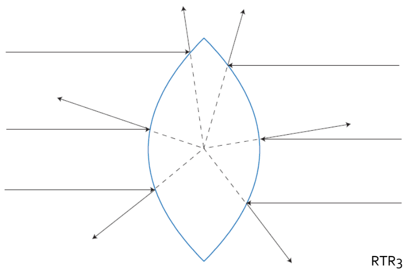
$$\boxed{s = \frac{x}{2p} + \frac{1}{2} \qquad t = \frac{y}{2p} + \frac{1}{2}}$$

Zhang08

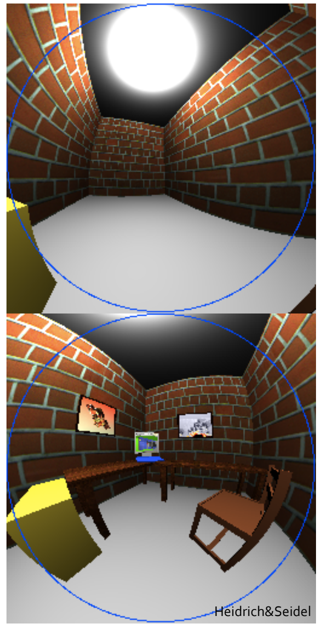
Parabolic Map

Uses z-component of reflected vector to determine texel

Texture creation from scene:
- view independent
- uniform sampling
- maps hard to create



RTR3



Heidrich&Seidel

OpenGL 2.1

Sphere map:

```
// insert where the texture is created
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Cube map:
- load six images, one for each face with:
  `glTexImage2D(target)`
- texture coordinates generated using
  `glTexGen*(…,GL_TEXTURE_GEN_MODE,GL_REFLECTION_MAP);`
  `glEnable(GL_TEXTURE_CUBE_MAP);`

Functions deprecated

Cube Mapping with GLSL

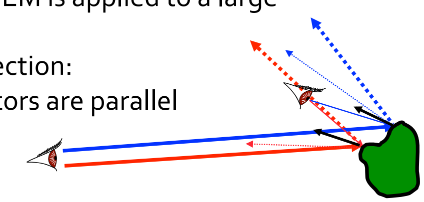
```
==== OpenGL app: initialize texture sampler to texture unit 0 ====
GLuint cubeid = glGetUniformLocation(myprog, "mycube");
glUniform1i(cubeid,0);  // assign texure unit 0 to cubeid

==================== // vertex shader: compute r =================
varying vec3 r;

void main() {
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
  vec3 n = normalize(gl_NormalMatrix * gl_Normal);
  vec4 e = gl_ModelViewMatrix * gl_Vertex;

  r = reflect(-e,n);
}
==================== // fragment shader ===========================
varying vec3 r;
uniform samplerCube mycube;

void main() {
  gl_FragColor = textureCube(mycube, r);
}
```

Angelo

Limitation of EM

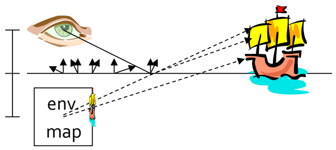
Environment map assumes object infinitesimally small and reflections infinitely far away

EM errors hard to notice on non-flat objects, but doesn't work well for flat/planar surfaces,
- reflected rays usually do not vary bymore than a few degrees ⇒ a small part of the EM is applied to a large area
- worse with orthographic projection: all orthographic reflected vectors are parallel

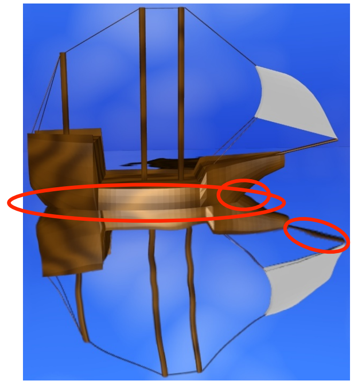


Limitations of EM

Can simulate reflection from still water



But not wavy water (via bump map)



Boat reflected in wavy water rendered using an environment map

Can you find three things wrong with this picture?

1. reflection doesn't meet boat
2. reflection behind the boat
3. environment map magnified

Harto8

Diffuse Reflectance

With EM, each texel is a directional light source:

- for perfectly specular surfaces, only lighting in the reflected direction contributes to lighting
- in the diffuse case, lighting is integrated over the hemisphere above a point
- cost of computing diffuse color of a point ($\mathbf{c}$) is on the order of the number of texels in the EM!

$$\mathbf{c} = m \sum_{j=1\ldots k} s^{(j)} \max((\mathbf{l}^{(j)} \bullet \mathbf{n}), 0)$$

- k directional lights (texels)
- $\mathbf{l}^{(j)}$: direction of light j
- $s^{(j)}$: intensity of light j
- $\mathbf{n}$: surface normal
- m: material reflectance

Schulze

Irradiance Map

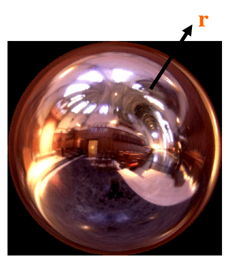
Precomputation of diffuse reflection

Observations:

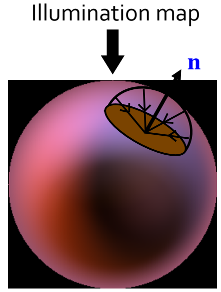
- irradiance at various points differs only on incoming directions and the surface normal
- all points with the same normal reflect the same irradiance
- (method limited to lighting contribution from a distant environment!)

Idea:

- precompute sum for all possible normals
- store results in a second environment map called the diffuse (irradiance environment) map
- radiance map indexed by surface normal



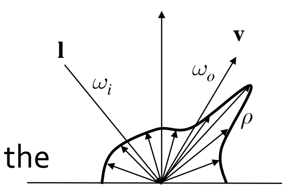
Illumination map



Irradiance map

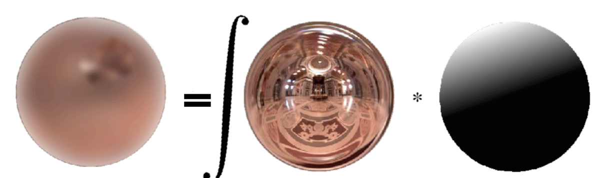
Schulze,Ramamoorth

Glossy Surfaces



More generally, given the BRDF of the surface, the Reflectance Equation is:

$$L(\omega_o) = \int_{-\pi/2}^{\pi/2} L(\omega_i)\rho(\omega_i, \omega_o)d\omega_i$$



Reflected Light = ∫ Incident Light * BRDF (ρ)

need to (pre-)compute irradiance*BRDF

Ramamoorthi,Kautz

Interactive Visual FX

Anti-aliasing:
- accumulation buffer

Camera effects:
  - motion blur
  - depth of field
- accumulation buffer

Shadows:
- projected (soft) shadows
  - stencil buffer
- depth buffer as shadow map

Other global illumination effects:
- reflection
- refraction
- color bleed (one bounce)
- caustics

Environmental effects:
- participating medium and volume rendering
- particle systems
- fluid dynamics

Limitations of GL_MULTISAMPLE

No control of:
- number of samples, can't have adaptive quality/performance trade-off
- sample locations: can't do stochastic sampling or adaptive sampling or use different sampling patterns (perhaps different per pixel)
- averaging function (filter shapes and extents)

We can use the accumulation buffer to address most of the shortcomings of GL_MULTISAMPLE (except for per-pixel sampling pattern, and at the cost of slower performance)

The Accumulation Buffer

Same size as the color buffer, used to hold (accumulate) results from partial computation

Deprecated since OpenGL 3.1

Instead, use framebuffer object with floating-point pixel format
- same concept as accumulation buffer

Multisampling with the Accumulation Buffer

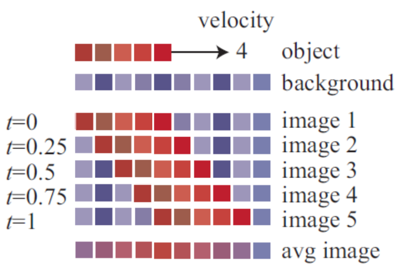
```
glutInitDisplayMode(… | GLUT_ACCUM);
// set up desired rendering modes

glAccum(GL_LOAD, 0.0); // or glClear(GL_ACCUM_BUFFER_BIT);

for (int i=0; i<n; ++i) {
  // specify sampling location for the i-th pass
  // by offsetting the frustrum
  // (google accpersp.c for the redbook source samples)
  render(scene); // to color buffer

  // accumulate the color buffer (multiplied by
  // a weight) to the accumulation buffer
  glAccum(GL_ACCUM, sampleweight[i]);
}

// copy the accumulation buffer to color buffer
glAccum(GL_RETURN, 1.0);
```

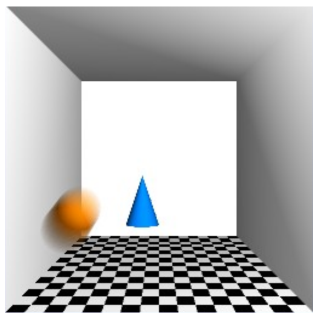
Akeley07

Motion Blur

Sample the scene k times, place the moving object(s) at a new location each time

Each sample contributes $1/k$-th of the final color:
`glAccum(GL_ACCUM, 1/k)`



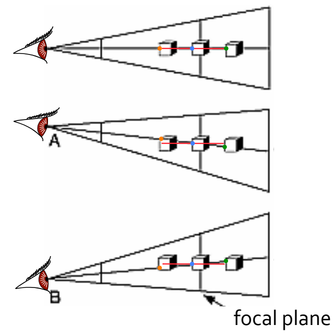


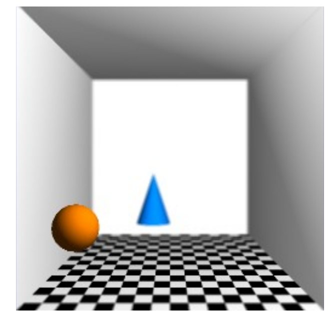
Hart,RTR3

Depth of Field

Sample the scene k times, each time with a slightly different eye position, but such that the focal plane bounded by the frustrum is the same in each sample

Each sample contributes $1/k$-th of the final color:
`glAccum(GL_ACCUM, 1/k)`





Hart

Shadows for Interactive Rendering

Let's start with hard shadows

Phong illumination model with hard shadows:

$$\mathbf{c}_t = \mathbf{c}_g + m_e + \sum_{k=1}^{n} s_{spot}^{(k)} (\mathbf{c}_a^{(k)} + v^{(k)} f(d^{(k)})(\mathbf{c}_d^{(k)} + \mathbf{c}_s^{(k)}))$$

(k: light number, not exponentiation!)
- includes visibility term ($v^{(k)} = 1$), if a light can "see" the point
- if point is in shadow, only ambient term applies

How to determine if point is in shadow?

Merrell

Computing Shadows

Planar receiver
- projected shadows

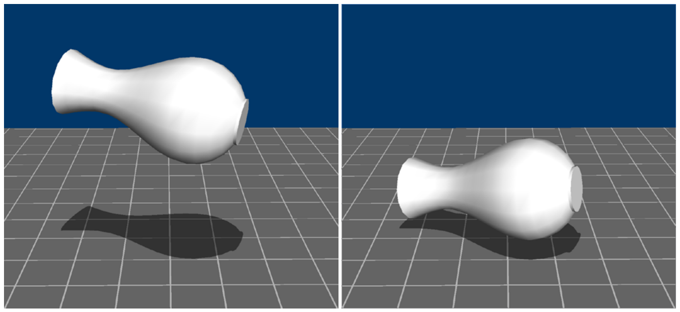
Non-planar receiver
- shadow maps
- projective texture
- shadow volumes

All performed in real-time/interactive (sort of)

Projected Shadows

Ways to think about shadows:

- as a dark volume of space
- as places not seen from a light source looking at the scene
- as a separate object
  - project object to the receiver and draw it a second time

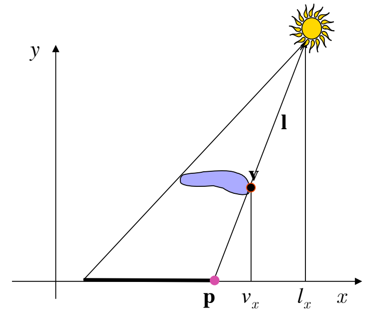


Akenine-Möller02,Durand

Projected Shadows

Works only with point lights and planar receivers

- project occluders onto receivers
- using shadow projection matrix



For projection onto $y = 0$:

$$\frac{l_y}{l_x - p_x} = \frac{v_y}{v_x - p_x}$$

$$l_y v_x - l_y p_x = l_x v_y - p_x v_y$$

$$l_y v_x - l_x v_y = p_x (l_y - v_y)$$

$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

$$\mathbf{p} = \mathbf{M}\mathbf{v}$$

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ p_w \end{bmatrix} = \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

What's the projection matrix for planar receiver in general, other than for $y = 0$?

Projected Shadows

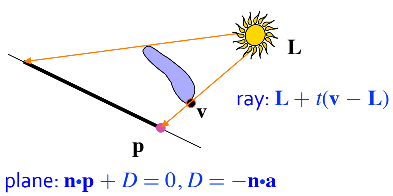
Point $\mathbf{p}$ is at the intersection of ray and plane if (mixing notation):

$$\mathbf{n} \cdot \mathbf{p} + D = 0,\ D = -\mathbf{n} \cdot \mathbf{a}$$

$$\mathbf{n} \cdot (\mathbf{L} + t(\mathbf{v} - \mathbf{L})) + D = 0$$

$$t = -\frac{D + \mathbf{n} \cdot \mathbf{L}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{L})}$$

$$\mathbf{p} = \mathbf{L} - \left(\frac{D + \mathbf{n} \cdot \mathbf{L}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{L})} \right) (\mathbf{v} - \mathbf{L})$$



Shadow projection matrix ($\mathbf{M}$) can now be computed:

$$\mathbf{p} = \mathbf{M}\mathbf{v}$$

$$\mathbf{M} = \begin{bmatrix} \mathbf{n} \cdot \mathbf{L} + D - L_x n_x & -L_x n_y & -L_x n_z & -L_x D \\ -L_y n_x & \mathbf{n} \cdot \mathbf{L} + D - L_y n_y & -L_y n_z & -L_y D \\ -L_z n_x & -L_z n_y & \mathbf{n} \cdot \mathbf{L} + D - L_z n_z & -L_z D \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{L} \end{bmatrix}$$

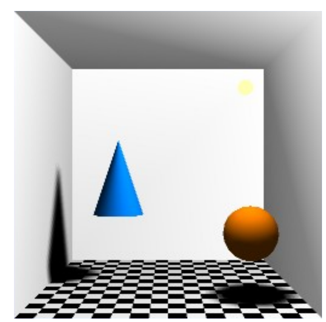
Yu

Soft Shadows

Sample the scene k times, with object projected onto the receiver, each time with a slightly different light position

Each sample contributes $1/k$-th of the final shadow color:

```
glAccum(GL_ACCUM, 1/k)
```



Hart