Content:

**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

Computing Reflection

Steps:
1. compute reflection vector, $r$
   • $e$ from eye to vertex
   • $n$ normal in eye coordinates
   • $r = e - 2n(e \cdot n)$

2. reflection is a function of one direction: largest absolute value of $r'$s components determines the cube face to reflect
   • example: $r = (5, -1, 2)$ gives $+x$ as the reflected face

3. divide $r$ by the value of the “reflection” coordinate ($5$) and map to $[0,1]$:
   $$ (s, t) = \left( \frac{(y + x) / 2x, (z + x) / 2x}{5}, \left( \frac{-1/5 + 1}{2}, \frac{2/5 + 1}{2} \right) \right) $$
   $$ = (0.4, 0.7) $$

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

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

Sphere Mapping

Use a texture map of a sphere viewed from infinity use \( r \) to look up texel
  • the eye vectors are parallel
  • \( r \) determined only by surface normal

Want: compute texture coordinates \((s, t)\) from \( r \)
  • 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

Computing \((s, t)\) from \( r \)

Observation: \( n \) can be expressed in terms of \( r \) and \( e \):

\[
\begin{align*}
  r &= e - 2n(n \cdot e) \\
  r - e &= \alpha n \\
  \alpha n &= r - e = \begin{bmatrix}
  x/p \\
  y/p \\
  (z+1)/p \\
  0
\end{bmatrix}
\]

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

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

Computing \((s, t)\) from \( r \)

Unit sphere in eye space, gazing down \(-z\):

\[
\begin{align*}
  s &= \frac{1}{2} h_x + \frac{1}{2} \\
  t &= \frac{1}{2} h_y + \frac{1}{2}
\end{align*}
\]

\[
\begin{bmatrix}
  h_x \\
  h_y \\
  \sqrt{1 - h_x^2 - h_y^2} \\
  0
\end{bmatrix}
\]

where \( p = \sqrt{2(z+1)} \)
Parabolic Map
Uses \( z \)-component of reflected vector to determine texel

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

Sphere map:
```c
// 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:
  ```c
  glTexImage2D(target);
  ```
- texture coordinates generated using
  ```c
  glTexGen*(..., GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
  ```
  ```c
  glEnable(GL_TEXTURE_CUBE_MAP);
  ```

Functions deprecated

Cube Mapping with GLSL
```c
==== OpenGL app: initialize texture sampler to texture unit 0 ====
GLuint cubeid = glGetUniformLocation(myprog, "mycube");
glUniform1i(cubeid, 0);  // assign texture 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);
}
```

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 by more 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)

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 \( c \) is on the order of the number of texels in the EM!

\[
c = m \sum_{j=1}^{k} s(j) \max((l(j) \cdot n), 0)
\]

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

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

Glossy Surfaces

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

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

need to (pre-)compute irradiance*BRDF
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

```c
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);
```
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:

$$\text{glAccum(GL\_ACCUM, } 1/k)$$

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 frustum is the same in each sample.

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

$$\text{glAccum(GL\_ACCUM, } 1/k)$$

Shadows for Interactive Rendering

Let’s start with hard shadows.

Phong illumination model with hard shadows:

$$\mathbf{c}_t = \mathbf{c}_a + m_s \sum_{k=1}^{n} s_{\text{soft}}^{(k)} (\mathbf{c}_u^{(k)} + v^{(k)} f(d^{(k)})(\mathbf{c}_d^{(k)} + \mathbf{c}_i^{(k)}))$$

$k$: 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?

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

Projected Shadows

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

\[
\mathbf{n} \cdot \mathbf{p} + D = 0, \quad D = -\mathbf{n} \cdot \mathbf{a}
\]

\[
\mathbf{n} (\mathbf{L} + t (\mathbf{v} - \mathbf{L})) + D = 0
\]

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

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

Plane: \( \mathbf{n} \cdot \mathbf{p} + D = 0, \quad D = -\mathbf{n} \cdot \mathbf{a} \)

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

\[
\mathbf{p} = \mathbf{M} \mathbf{v}
\]

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

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:

\[
glAccum(GL\_ACCUM, 1/k)
\]