Shadows

What are shadows?
What distinguishes a point in shadow from a lit point?

Types of light sources
- Point, directional
- Area lights and generally shaped lights
  - Not considered here
  - Later: ray-tracing for such light sources

Problem statement
A shadow algorithm for point and directional lights determines which scene points are
- Visible from the light source (i.e., illuminated)
- Invisible from the light source (i.e., in shadow)
Thus: shadow casting is a visibility problem!

Types of Shadow Algorithms

Object Space
- Like object space visibility algorithms, the method computes in object space which polygon parts that are illuminated and which are in shadow
  - Individual parts are then drawn with different intensity
  - Typically slow, O(n^2), not for dynamic scenes

Image Space
- Determine visibility per pixel in the final image
  - Sort of like depth buffer
  - Shadow maps
  - Shadow volumes

Course News
Assignment 3 (project)
- Due April 1
Reading
  - Chapter 11.8, 10
Next Friday
  - Will be out of town for committee meeting
  - Replacement TBD

Credits
The following shadow mapping slides are taken from Mark Kilgard’s OpenGL course at Siggraph 2002.
Shadow Mapping Concept (1)

Depth testing from the light's point-of-view
Two pass algorithm
First, render depth buffer from the light's point-of-view
- The result is a "depth map" or "shadow map"
- Essentially a 2D function indicating the depth of the
  closest pixels to the light
This depth map is used in the second pass

Shadow Mapping Concept (2)

Shadow determination with the depth map
Second, render scene from the eye's point-of-view
For each rasterized fragment
- Determine fragment's XYZ position relative to the light
- This light position should be setup to match the
  frustrum used to create the depth map
- Compare the depth value at light position XY in the
  depth map to fragment's light position Z

The Shadow Mapping Concept (3)

The Shadow Map Comparison
Two values
- \( A = Z \) value from depth map at fragment's light XY
  position
- \( B = Z \) value of fragment's XYZ light position
If \( B \) is greater than \( A \), then there must be something
  closer to the light than the fragment
- Then the fragment is shadowed
If \( A \) and \( B \) are approximately equal, the fragment is lit

Shadow Mapping with a Picture in 2D (1)

The \( A < B \) shadowed fragment case

A scene with fairly complex shadows

Shadow Mapping with a Picture in 2D (2)
Visualizing the Shadow Mapping Technique (7)

Complete scene with shadows

Notice how specular highlights never appear in shadows

Notice how curved surfaces cast shadows on each other

In Practice: Depth Map Precision Issues

Have to add a little offset to depth map values to account for limited precision

Too little bias, everything begins to shadow

Too much bias, shadow starts too far back

Just right

What is Projective Texturing?

An intuition for projective texturing
- The slide projector analogy

About Projective Texturing (1)

First, what is perspective-correct texturing?
- Normal 2D texture mapping uses (s, t) coordinates
- 2D perspective-correct texture mapping
  - Means (s, t) should be interpolated linearly in eye-space
  - So compute per-vertex s/w, t/w, and 1/w
  - Linearly interpolated these three parameters over polygon
  - Per-fragment compute s' = (s/w) / (1/w) and t' = (t/w) / (1/w)
  - Results in per-fragment perspective correct (s', t')

About Projective Texturing (2)

So what is projective texturing?
- Now consider homogeneous texture coordinates
  - {s, t, r, q} \rightarrow (s/q, t/q, r/q)
  - Similar to homogeneous clip coordinates where
    \[(x, y, z, w) = (x/w, y/w, z/w)\]
  - Idea is to have (s/q, t/q, r/q) be projected per-fragment

Back to the Shadow Mapping Discussion . . .

Assign light-space texture coordinates to polygon vertices
- Transform eye-space (x, y, z, w) coordinates to the light’s view frustum (match how the light’s depth map is generated)
- Further transform these coordinates to map directly into the light view’s depth map
  - Expressible as a projective transform
    \[(s/q, t/q)\] will map to light’s depth map texture
**Shadow Map Operation**

**Next Step:**
- Compare depth map value to distance of fragment from light source.
- Different GPU generations support different means of implementing this:
  - Today’s GPUs: pixel shader.
  - Earlier: special hardware for implementing this feature (e.g. SGI), or just using alpha blending [Heidrich’99].

---

**Issues with Shadow Mapping (1)**

**Not without its problems**
- Prone to aliasing artifacts.
- “Percentage closer” filtering helps this.
- Normal color filtering does not work well.
- Depth bias is not completely foolproof.
- Requires extra shadow map rendering pass and texture loading.
- Higher resolution shadow map reduces blockiness.
  - But also increase texture copying expense.

---

**Hardware Shadow Map Filtering Example**

**GL NEAREST: blocky, GL LINEAR: antialiased edges**

Low shadow map resolution used to heighten filtering artifacts.

---

**Issues with Shadow Mapping (2)**

**Not without its problems**
- Shadows are limited to view frustums.
- Could use six view frustums for omni-directional light.
- Objects outside or crossing the near and far clip planes are not properly accounted for by shadowing.
  - Move near plane in as close as possible.
  - But too close throws away valuable depth map precision when using a projective frustum.

---

**More Examples**

**Complex objects all shadow**

---

**More Examples**

**Even the floor casts shadow**

Note shadow leakage due to infinitely thin floor.
Could be fixed by giving floor thickness.
Combining Projective Texturing for Spotlights

Use a spotlight-style projected texture to give shadow maps a spotlight falloff

Combining Shadows with Atmospherics

Shadows in a dusty room

- Stimulate atmospheric effects such as suspended dust
- Construct shadow map
- Draw scene with shadow map
- Modulate projected texture image with projected shadow map
- Blend back-to-front shadowed slicing planes also modulated by projected texture image

Credit: Case Beckett

Shadow Maps

Approach for shadows from point light sources

Surface point is in shadow if it is not visible from the light source.
- Use depth buffer to test visibility:
  - Render scene from the point light source
  - Store resulting depth buffer as texture map
  - For every fragment generated while rendering from the camera position, project the fragment into the depth texture taken from the camera, and check if it passes the depth test.

Shadow Volumes

Use new buffer: stencil buffer

- Just another channel of the framebuffer
- Can count how often a pixel is drawn

Algorithm (1):

- Generate silhouette polygons for all objects
  - Polygons starting at silhouette edges of object
  - Extending away from light source towards infinity
  - These can be computed in vertex programs

Algorithm (2):

- Render all original geometry into the depth buffer
  - I.e. do not draw any colors (or only draw ambient illumination term)
- Render front-facing silhouette polygons while incrementing the stencil buffer for every rendered fragment
- Render back-facing silhouette polygons while decrementing the stencil buffer for every rendered fragment
- Draw illuminated geometry where the stencil buffer is 0, shadow elsewhere
**Shadow Volumes**

*Image by ATI*

**Discussion:**
- Object space method therefore no precision issues
- Lots of large polygons: can be slow
  - High geometry count
  - Large number of pixels rendered

---

**Coming Up:**

*Next Week:*
- Ray-tracing
- Global illumination