Stenciled Shadow Volume Optimizations (1)

- Use GL_QUAD_STRIP rather than GL_QUADS to render extruded shadow volume quads
- Requires determining possible silhouette loop connectivity
- Mix Zfail and Zpass techniques
  - Pick a single formulation for each shadow volume
  - Zpass is more efficient since shadow volume does not need to be closed
  - Mixing has no effect on net shadow depth count
  - Zfail can be used in the hard cases

Stenciled Shadow Volume Optimizations (2)

- Pre-compute or re-use cache shadow volume geometry when geometric relationship between a light and occluder does not change between frames
  - Example: Headlights on a car have a static shadow volume w.r.t. the car itself as an occluder
- Advanced shadow volume culling approaches
  - Uses portals, Binary Space Partitioning trees, occlusion detection, and view frustum culling techniques to limit shadow volumes
  - Careful to make sure such optimizations are truly correct

Stenciled Shadow Volume Optimizations (3)

- Take advantage of ad-hoc knowledge of scenes whenever possible
  - Example: A totally closed room means you do not have to cast shadow volumes for the wall, floor, ceiling
- Limit shadows to important entities
  - Example: Generate shadow volumes for monsters and characters, but not static objects
  - Characters can still cast shadows on static objects
- Mix shadow techniques where possible
  - Use planar projected shadows or light-maps where appropriate

Stenciled Shadow Volume Optimizations (4)

- Shadow volume’s extrusion for directional lights can be rendered with a GL_TRIANGLE_FAN
- Directional light’s shadow volume always projects to a single point at infinity

Hardware Enhancements: Depth Clamp (1)

- What is depth clamping?
  - Boolean hardware enable/disable
  - When enabled, disables the near & far clip planes
  - Interpolate the window-space depth value
  - Clamps the interpolated depth value to the depth range, i.e., \([\min(n,f), \max(n,f)]\)
  - Assumimg \(\text{glDepthRange}(n,f)\)
  - Geometry “behind” the far clip plane is still rendered
    - Depth value clamped to farthest \(Z\)
    - Similar for near clip plane, as long as \(w<0\), except clamped to closest \(Z\)

Hardware Enhancements: Depth Clamp (2)

- Advantage for stenciled shadow volumes
  - With depth clamp, \(P\) (rather than \(P_{\infty}\)) can be used with our robust stenciled shadow volume technique
  - Marginal loss of depth precision regained
  - Orthographic projections can work with our technique with depth clamping
- NV_depth_clamp OpenGL extension
  - Easy to use
    - \(\text{glEnable(GL_DEPTH_CLAMP_NV)}\)
    - \(\text{glDisable(GL_DEPTH_CLAMP_NV)}\)
  - GeForce3 & GeForce4 Ti support (soon)
Hardware Enhancements: Two-sided Stencil Testing

- Current stenciled shadow volumes required rendering shadow volume geometry twice
  - First, rasterizing front-facing geometry
  - Second, rasterizing back-facing geometry
- Two-sided stencil testing requires only one pass
  - Two sets of stencil state: front- and back-facing
  - Boolean enable for two-sided stencil testing
  - When enabled, back-facing stencil state is used for stencil testing back-facing polygons
  - Otherwise, front-facing stencil state is used
  - Rasterizes just as many fragments, but more efficient for CPU & GPU

Hardware Enhancements: Two-sided Stencil Testing (2)

- NV_stencil_two_side OpenGL extension
  - Enable applies if GL_STENCIL_TEST also enabled
    - glEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
    - glDisable(GL_STENCIL_TEST_TWO_SIDE_NV);
  - Control of front- and back-facing stencil state update
    - glActiveStencilFaceNV(GL_FRONT);
    - glActiveStencilFaceNV(GL_BACK);
  - Existing stencil routines (glStencilOp, glStencilMask, glStencilFunc) update the active stencil face state
  - glClear and non-polygon primitives always use the front-facing stencil state
  - Expect on future GPUs

Usage of NV_stencil_two_side & EXT_stencil_wrap

OLD SCHOOL

gDepthMask(0);
gColorMask(0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);

NEW SCHOOL

gDepthMask(0);
gColorMask(0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);
gActiveStencilFaceNV(GL_BACK);
gDepthMask(0);
gClear(GL_STENCIL_MASK, 0);
gClearColor(0.0,0.0,0.0,0.0);
gColorMask(0.0,0.0,0.0,0.0);
gEnable(GL_CULL_FACE);
gEnable(GL_STENCIL_TEST);
gEnable(GL_STENCIL_TEST_TWO_SIDE_NV);
gActiveStencilFaceNV(GL_FRONT);

New approach calls renderShadowVolumePolygons() just once.

Algorithms for Soft Shadows

Hard vs. Soft Shadow

- point source
- area source
- umbra
- penumbra
- umbra penumbra

Sampling the Light Source

- Use arbitrary hard shadow algorithm
  - Select point sample on area light source
  - Render hard shadows
  - Sum up weighted result (e.g. accumulation buffer)
Sampling the Light Source

Example: Ground plane shadow texture

1. Initialize FB (white)
2. For each sample point do
   2a. Render scene
   2b. Subtract $1/N$ from FB only once for each pixel (stencil)!

Result

Number of samples == number of passes
Problematic for complex scenes
$N$ samples == $(N+1)$ levels of shadow
- Fully lit, fully shadowed
- $(N-1)$ levels of penumbra
- How much samples?

Plateau Soft Drop Shadows

Say we want to paint a soft shadow for an image.

Plateau Idea

“Paint” each shadow’s edge, blurring it as its height from the plane increases.

Forming Plateaus

Create the shadow object
Apply rendering as texture
Render from above
Plateau Result

- Soft shadows, at the cost of finding the silhouette and drawing cones & sheets.

Plateau Limitations

- Overstated umbrae, penumbrae are not physically correct, and like Nguyen.

Stenciled Shadow Volumes with Multiple Lights

- Three colored lights. Diffuse/specular bump mapped animated characters with shadows. 34 fps on GeForce4 Ti 4600; 80+ fps for one light.

Stenciled Shadow Volumes for Simulating Soft Shadows

- Cluster of 12 dim lights approximating an area light source. Generates a soft shadow effect; careful about banding. 8 fps on GeForce4 Ti 4600.

Penumbral Wedge Shadows

- Generates soft shadows for area light sources
- Based on original work by Tomas Akenine-Möller and Ulf Assarsson:
  - http://graphics.cs.lth.se/research/shadows/
  - A new rendering algorithm follows

Penumbral Wedge Shadows

- General procedure
  - First render ordinary stencil shadows
  - For edge silhouette edge, generate a wedge that represents the extent of the penumbra
  - For edge wedge, apply a correction to the stencil shadows that softens the hard shadow outline
**Penumbral Wedge Shadows**

- **Area Light**
- **Shadow Castor**
- **Outer Penumbra**
- **Inner Penumbra**

**A Penumbral Wedge**

- **Inner Penumbra**
- **Outer Penumbra**
- **Extruded Silhouette Edge**

---

**Soft Shadow Correction**

- Lighting pass for ordinary stencil shadows uses stencil test
  - 0 in stencil buffer at a particular pixel means light can reach that pixel
  - Nonzero means pixel is in shadow

- For soft shadows, we use alpha blending during lighting pass
  - Value in the alpha channel represents how much of the area light is covered
  - 0 means entire light source visible from a particular pixel
  - 1 means no part of light source is visible (fully shadowed)
Soft Shadow Correction

After rendering stencil shadows, the stencil buffer contains integer values. Each value represents the number of shadow volumes covering a particular pixel.

To make fractional corrections, we need to be able to treat the integer stencil values as either fixed-point or floating-point numbers. We have two options...

Option 1: Render the shadow volumes into a 16-bit floating-point render target instead of the ordinary stencil buffer.

Option 2: Copy the stencil values into the alpha channel and shift them left by some number of fraction bits.

Copying stencil values to alpha
- Requires the OpenGL extension GL_NV_copy_depth_to_color
- After copying, we need to scale the alpha values since a 1 in the stencil buffer is now 1/255 in the alpha channel
- Scaling by 31.875 gives us 3.5 bit fixed-point in the alpha channel

Now we need to make fractional corrections to the stencil values.
- For each inner half of a penumbral wedge, we subtract a fraction.
- For each outer half of a penumbral wedge, we add a fraction.
- Value becomes 0.5 at original stencil boundaries.
Penumbra Wedge Rendering

- How do we know which pixels need a correction?
- Each penumbral wedge is divided into two halves
  - The inner half-wedge
  - The outer half-wedge
- Both halves are bounded on one side by the extruded silhouette edge used for stencil shadows

Penumbra Wedge Rendering

In the vertex program, we compute the three outside bounding planes of a half-wedge
- Send these planes to the fragment program in viewport space!
- Allows us to do a quick test to determine whether a viewport-space point is outside the half-wedge

Penumbra Wedge Rendering

Sort half-wedges into two batches:
- 1) Those for which camera is on the positive side of the silhouette edge
- 2) Those for which camera is on the negative side of the silhouette edge
- Extruded silhouette plane normal always points outward from shadow volume

Penumbra Wedge Rendering

In the fragment program, we test the viewport-space position of the point in the frame buffer against three half-wedge bounding planes
- We will use the depth test to reject points on the wrong side of the extruded silhouette edge
Rendering Outer Half-wedges
- Half-wedges for which camera is on positive side of silhouette plane
  - Render front faces when $z$ test fails
- Half-wedges for which camera is on negative side of silhouette plane
  - Render back faces when $z$ test passes

Penumbral Wedge Rendering
- After all wedges have been rendered, scale the alpha channel by 8 to get pure fraction
  - Render a full-screen quad 3 times
  - Double alpha each time
  - Restricted to light scissor rectangle
  - Color channels masked off

- If the value was greater than one, then it's saturated to one, corresponding to fully shadowed
- Then render lighting pass, multiplying source color by one minus destination alpha

- Using a floating-point visibility buffer avoids scaling step
- Visibility values still need to be copied to alpha channel from render target
Small Light Area
Shadows sharper, rendering faster

Large Light Area
Shadows softer, interact more, rendering slower

Semi-penumbral Shadows
- Method for speeding up penumbral wedge soft shadows
- Only render outer half-wedges
- Less correct, but still looks good
- Lose the ability to cast shadows that have no point of 100% light occlusion

Semi-penumbral Shadows
Instead of full penumbra:
Render outer half of penumbra only:

Inner and outer half-wedges rendered
Only outer half-wedges rendered

Penumbra Maps: Approximate Soft Shadows in Real-Time
Chris Wyman and Charles Hansen
University of Utah
Why Approximate Soft Shadows?

- Important reasons to add soft shadows:
  - Realism and spatial cues
  - Yet accurate soft shadows = expensive
  - Not all applications can afford this expense
  - Plausible soft shadows may be enough

Related Work

- Lots of fast, accurate, but not interactive
  - Some examples: Drettakis and Fiume '94, Hart et. al '99, Stark and Riesenfeld '00
- Interactive methods have varied problems
  - Many expensive samples (Heckbert and Herf '97)
  - No self-shadowing (Soler and Sillion '98, Brabec and Seidel '02)
  - Cannot shadow arbitrary objects (Haines '01)
  - Expensive per-pixel operations in hardware (Brabec and Seidel '02, Akeinne-Möller and Assarsson '02)
  - Becomes hardware fill-rate bound (Akeinne-Möller and Assarsson '02)

Advantages of Penumbra Maps

- Builds on simple idea of "shadow plateaus" introduced by Haines ('01)
- Plausible soft shadows
- Hard upon contact, soft with distance
- Simple implementation on graphics hardware
- Hides some aliasing
- One sample per pixel

Penumbra Map Assumptions

- A hard shadow is a reasonable approximation for a shadow’s umbra
- Object silhouettes remain constant over light’s surface

Key Insight

- When using a hard shadow as the umbra, all of the approximate penumbra is visible from the center of the light
- Allows storage of penumbral intensity in a separate map we call a penumbra map

Creating Penumbra Map

- Compute shadow map (for hard shadow)
- Compute object silhouette from light’s center
- Compute cones at silhouette vertices
- Compute sheets connecting vertices (along silhouette edges)
Computing Cones

- For each silhouette vertex
- Find distance from light's center to vertex
- Find distance from vertex to far plane
- Using these distances and the light radius \( L_r \), compute \( C_r \) using similar triangles

Computing Sheets

- Create quads at each silhouette edge tangent to the adjacent cones
- May not be planar
- Subdivide significantly non-planar quads for good results

Shadowing Complex Objects

- Can not just draw quad with 0 (shadowed) at A and 1 (illuminated) at C
- Result depends on current fragment \( F \) on quad and point \( P \) in the shadow map

Use Fragment Program to Generate Map

1. \( F_{\text{world}} = \text{GetWindowCount}(F) \)
2. \( Z_P = \text{TexCoord}(X_{\text{world}}, F_{\text{world}}) \)
3. \( Z_P = Z_{\text{world}} \)
4. if \( Z_P > Z_L \) DiscardFragment()
5. \( Z_P = \text{ConvertToWorldSpace}(Z_P) \)
6. \( Z_P = \text{ConvertToWorldSpace}(Z_P) \)
7. \( f = (Z_P - Z_L) / (Z_P - Z_L) \)
8. \( f = \sqrt{f} \)
9. OutputShadow = 1
10. OutputPenumbra = \( f \)

Rendering

- Compare fragment’s depth to shadow map to determine if light is completely blocked
- If not completely shadowed, index into penumbra map to determine percentage of light reaching surface
- Multiple lights requires multiple shadow and penumbra maps

Implementation

- Use OpenGL with ARB vertex and pixel program extensions
- Implemented on a 2.0 GHz Pentium 4 with an ATI Radeon 9700 Pro
- Use a brute force silhouette extraction
- All our results use a spherical light source
Video

Results

More Results

Changing Penumbra Map Size

Problems

Blending Issues

- Framerates are ~18 Hz with 10242 shadow map, penumbra map, and image size
- Note the pathtraced image uses the larger light
- Use 10k triangle bunny to generate shadows

- Framerates are ~15 Hz for 1024^2 resolution

- Shadows are not accurate
  - Less accurate as occluders move further away from shadowed objects
  - Assume silhouettes constant over light
  - Noticeable pops on cube
  - No problems with other objects
- Blending overlapping penumbrae
  - Occurs on a per-pixel basis
  - No geometric info in the hardware
  - Artifacts at silhouette concavities

- In these three cases, overlapping penumbrae should be handled differently
  - No geometric information in the pixel program means no quick way to decide in hardware
  - We always choose the darkest pixel (left image)
Future Work

- Better way to handle blending?
- Perhaps get the entire penumbra by using vertex programs to play with silhouettes
- Move everything into hardware
  - [McCool '00] finds silhouettes using a shadow map (future hardware capabilities should allow this in hardware)

Acknowledgements

- Work supported by NSF Grants 9977218 and 9978099
- Aaron Lefohn, Milan Ikits, and Joe Kniss provided graphics hardware programming tips
- Numerous reviewers whose helpful comments improved the paper significantly

Same as Smoothies....

Rendering Fake Soft Shadows with Smoothies

Eric Chan Frédéric Durand

Laboratory for Computer Science
Massachusetts Institute of Technology