Projective Shadows

D. Sim Dietrich Jr.
Topics

- Projective Shadow Types
- Implementation on DirectX 7 HW
- Implementation on DirectX8 HW
- Integrating Shadows into an engine
Types of Projective Shadows

Static Cookies
- Typically Black & White pre-calculated textures
- A burnt-in occlusion map
- Not for moving objects or lights
- Think of the batman spot light
- Soft Edges

Render To Texture Cookies
- Like a Static Cookie, but can be dynamic
- If light or object moves or animates, can re-render
- Clear to white, draw object in black
- Project onto receiver geometry
- Soft Edges
Cookie of House
Types of Projective Shadows

Render To Texture Depth Map
- Uses one or more color channels to store depth from the light
- Can be 8-9 bit precision
- 8-bit precision causes Bias Problems
- Hard Edges

Render To Texture ObjectID Map
- Sort objects relative to light
- Render each object with own ID into map
- No self-shadowing :-(
- No self-shadowing artifacts :-)
- Hard Edges
Types of Projective Shadows

- Combination of Depth & ObjectID Map
  - Uses per-Object Depth range
    - 8 bits of precision per object – plenty in most cases
  - Uses inter-object IDs to get infinite range
  - Hard edges
Types Of Projective Shadows

- Z-Buffer Shadows
  - Render from light’s point of view into Z buffer only
  - Select Z buffer as a texture
    - D3DUSAGE_DEPTHSTENCIL
  - HW Compares r/q texture coordinates to depth stored at s/q, r/q in Z buffer texture
  - Returns 0..1 filtered result as a color
  - GF3/4Ti only right now
    - Other HW support coming
  - Soft Edges
8 bit Depth Shadow Buffers

First Clear Surface to 0xff000000.
Select Depth lookup texture into t0

Select Light’s Depth shadow map into t1
  texture.a
  selectarg1 // grab this pixel’s depth

  current.a
  subtract
  texture.a // compare to this pixel’s position relative to light

Color Channels can hold lighting function, use alpha for depth
Set Alpha Test to Render light if alpha != 0
8 bit Shadow Buffers

- If your light range is large, 8 bits may not be enough across the entire range.

- If you can separate your objects cleanly (i.e., characters walking around in front of each other), you might use ObjectIDs to get more range.

- You can combine ObjectID with 8-bit range in order to use the precision across only each object.

- In other words, each object has its own 8-bits of shader precision.
Depth/ObjectID Shadow Buffers

By using multiple color channels, you can compare several values at once with a single subtract.

A Dot Product3 operation can then sum up the results and replicate into the alpha channel.

Alpha test can then be used to check that all or none of the values passed their checks.
First Clear Surface to 0x7f7f7f7f.
Select Depth/ObjectID lookup texture into t0
Select Light’s Depth/ObjectID shadow map into t1

    texture
    selectarg1 // grab this pixel’s depth

    current
    subtract
    texture    // compare to this pixel’s position relative to light
                // minus 0x80808080
    1-current
    dot3
    tfactor    // tfactor has 2/255,2/255,2/255, 1. Adds up difference terms

Set Alpha Test to draw if alpha != 6/255
Set Blending Mode to SRC.COLOR*ZERO + DEST.COLOR*ZERO
to draw black when in shadow
Depth/ObjectID Shadow Details

Since the DOT3 operation expects biased & scaled values you need to play some biasing games

Your depth texture should only contain color values from 0x80-0xff ( ie ‘positive’ values )

When constructing the shadow map ( ie rendering the distance from the light into a texture ), subtract 0x80 from the depth

This makes the subtract operation return a value in the range ‘0x80-0xff’, which DOT3 can handle
Depth/ObjectID Shadow Details

- ObjectID shadows can have issues
  - Adjacent co-planar objects with different IDs will alias with each other
    - To work around in a single pass requires PS.1.1 or greater
  - Easiest to just make sure objects that can’t be separated by a plane are rendered with the same ID
If you try to use the same depth function during shadow construction & shadow testing phases, the object can self-occlude due to precision & aliasing issues.
8-Bit Self-Shadowing Artifacts

If we use a reasonable bias that prevents incorrect self-shadowing, the shadows don’t start from the correct place anymore.
Bias Tricks

- One way to get a built-in bias is to render just back-faces into the shadow map.
- This makes aliasing happen more on the back side of objects, which tends to be dark anyway due to the regular lambertian lighting equation.
- Only works for closed objects.
Resolution Aliasing

- Stair-step artifacts come from the limited resolution in the shadow map.
Resolution Aliasing

This can be reduced by using larger texture maps, but there are limits. This image uses a 512x512 map.
Conclusions: DirectX 7 HW

For cards without pixel shader functionality or Z-based shadow buffers

- Skip self shadowing
  - Your Choice of artifacts – shadow poke-through or light poke-through
  - Worst part is frame-to-frame inconsistency

Use Static & Dynamic Cookies

- No self-shadowing, but no precision problems either
- Can soften very cheaply
- Can render in 16-bit – use dithering to further add a bit of noise
- Skip z buffer, you only need color
How to Use Cookies

- Identify Caster / Receiver Pairs
- Between each pair, create a cookie
- Each cookie will contain a projection of all occluders from the light to the receiver
How to Use Cookies

The last object in the occluder chain needs to modulate all nearer occluders together:

- Render objects in black multiple times into cookie
  - More Geometry
- Render to texture, modulating previous cookies over each other
  - More RTT stalls
How To Use Cookies: Resampling

One can create cookies at the light’s projection plane
- Simplest, but you can lose resolution further away
  - Can be a good thing for fake bilinear softness
  - Can cache results easier

Or, create cookie right in front of receiver
- Maximizes resolution
- But less cacheable, and sharper
How to use Cookies

Create texture render target, color only, no Z
Clear to white
  Or, if using directional light, clear to light color
  Or, if using spot light, render quad with spotlight color & cosine falloff burnt in

Render object into texture
  Leave border alone
    Clear Z to zero with lines around borders
    Remap viewport to skip border
    Overwrite border with color after object drawn
How to use Cookies

Resample cookie with texture units to fake soft shadows

Can either:

- Make filter kernel size relate to distance from occluder, which is more or less correct
- Relate kernel size to distance from light
  - Wrong, but similar to above in some cases
- Just Blur for ‘softness’

Perform two diagonally offset bilinear samples

- Or 3, or 4, depending on # of texture fetchers
- Do multiple passes if necessary
Rotating/Jittering Texture Samples

Ordered Grid

Rotated/Jittered Grid

A sampling pattern that doesn’t match the texture’s grid gives more edge coverage values
Rotating/Jittering Texture Samples

A sampling pattern that doesn’t match the texture’s grid gives more edge coverage values.
Another approach is to simply rotate your shadow map a bit, say 22 degrees or so, with respect to the world, thus wasting some space for the sake of smoother shadow edges.
Rotating/Jittering Texture Samples

Do the same with cube-map based shadows, don’t make them line up with world-space, but rotate them with respect to the world axes.

The main thing is that the sampling pattern doesn’t exactly match your source data, which is grid-based.
Conclusions: DirectX 8 HW

- If HW has Z-based shadows, use them
  - Plenty of precision
  - Faster than updating Z & Color
  - Can get 4 jittered blended lookups in a single pass
  - Can use directly as cheaply as using a cookie
  - Can render to cookie using shadow map to soften further

- Else, use Cookies
  - Can’t really use Z shadows for cube-mapped omni lights
    - Could try range-based depths, but it will be wrong in between vertices