Advanced Stencil Shadow and Penumbral Wedge Rendering

Eric Lengyel

NAUGHTY DOG

SAN FRANCISCO CA
MAR 7-11
GDC 05

Overview

- Stencil shadow optimizations
- Penumbral wedge rendering
- Demonstration
- Questions

Stencil Shadow Pros

- Very accurate and robust
- Nearly artifact-free
  - Faceting near the silhouette edges is the only problem
- Work for point lights and directional lights equally well
- Low memory usage

Stencil Shadow Cons

- Too accurate — hard edges
  - Need a way to soften
- Very fill-intensive
  - Scissor and depth bounds test help
- Significant CPU work required
  - Silhouette determination
  - Building shadow volumes

Hardware Support

- GL_EXT_stencil_two_side
- GL_ATI_separate_stencil_func
  - Both allow different stencil operations to be executed for front and back facing polygons
- GL_EXT_depth_bounds_test
  - Helps reduce frame buffer writes
- Double-speed rendering

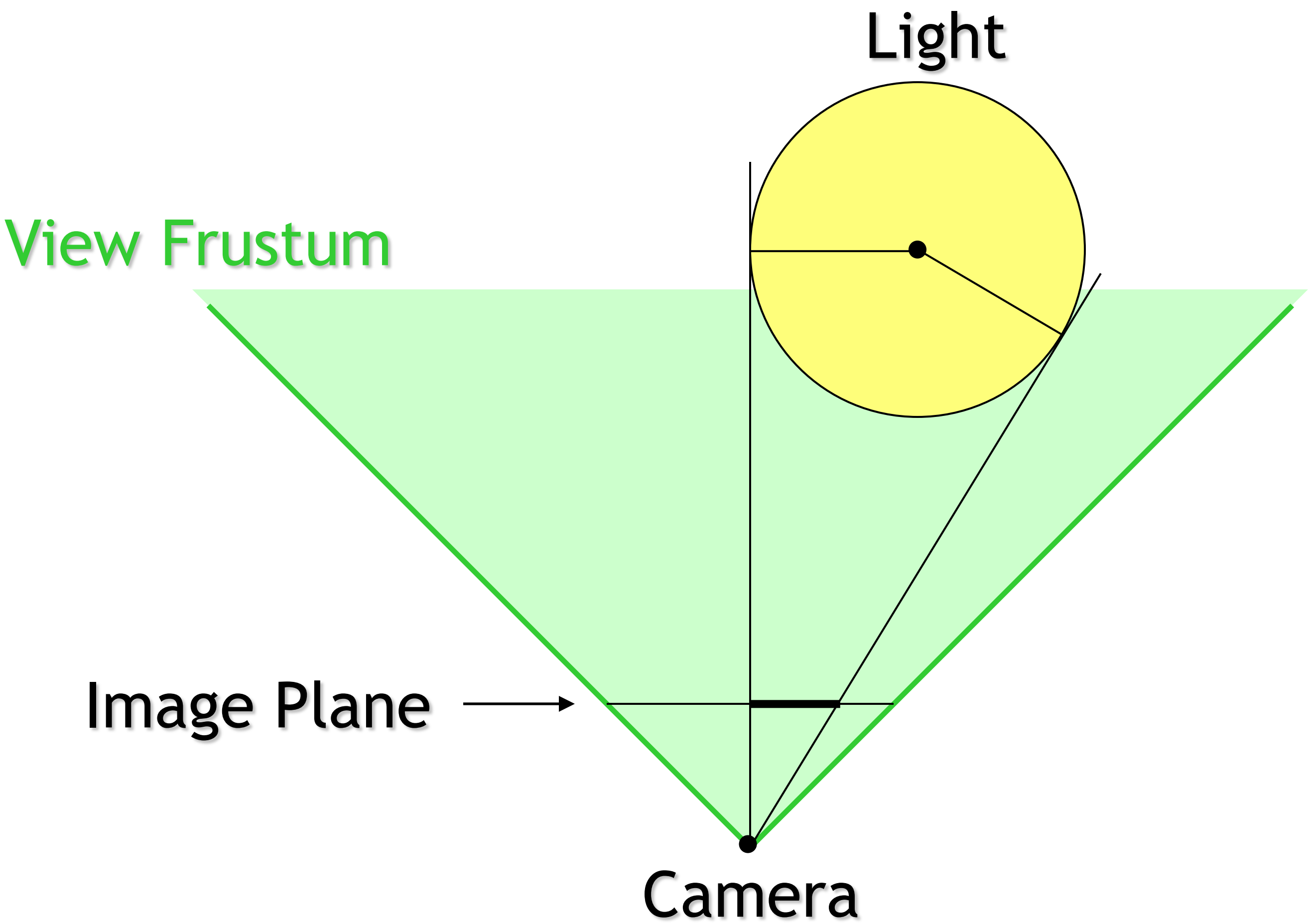
Scissor Optimizations

- Most important fill-rate optimization for stencil shadows
- Even more important for penumbral wedge shadows
- Hardware does not generate fragments outside the scissor rectangle — very fast

Scissor Optimizations

- Scissor rectangle can be applied on a per-light basis or even a per-geometry basis
- Requires that lights have a finite volume of influence

Light Scissor

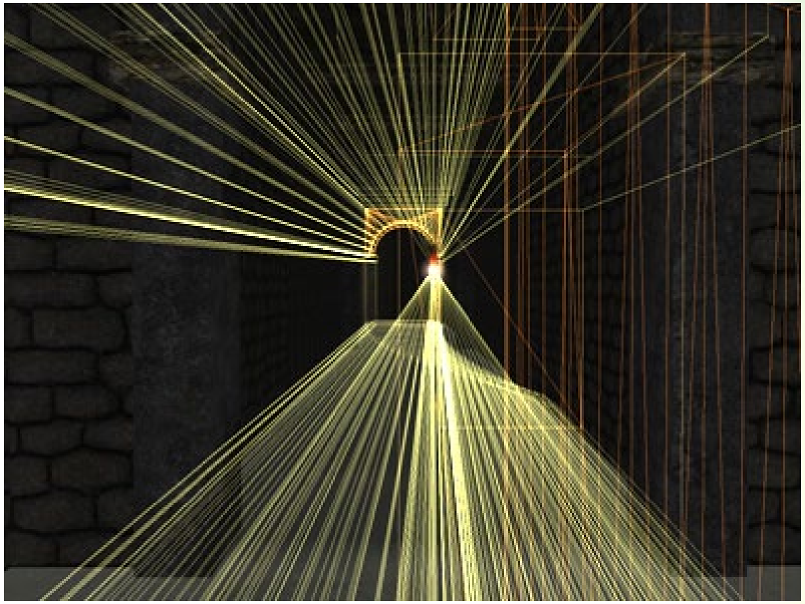


Light Scissor

- Project light volume onto the image plane
- Intersect extents with the viewport to get light's scissor rectangle
- Mathematical details at:
  - http://www.gamasutra.com/features/20021011/lengyel_01.htm

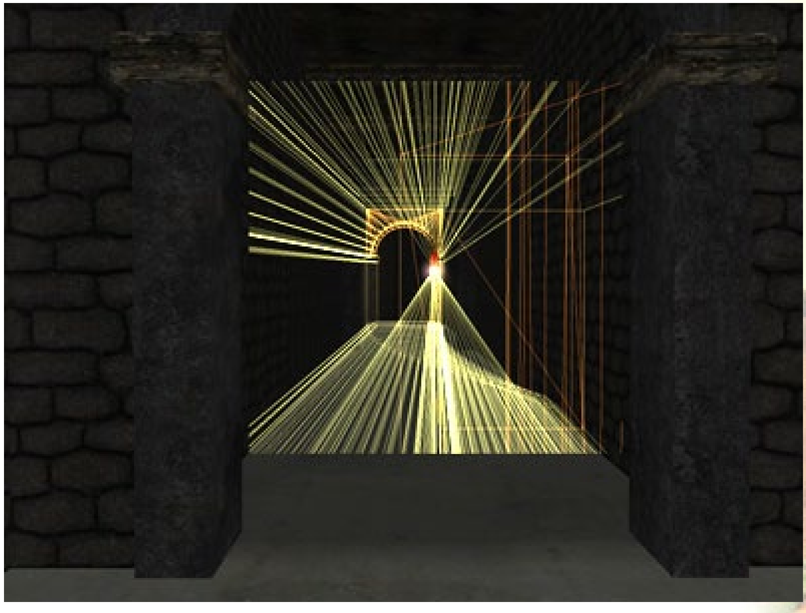
No Light Scissor

Shadow volumes extend to edges of viewport

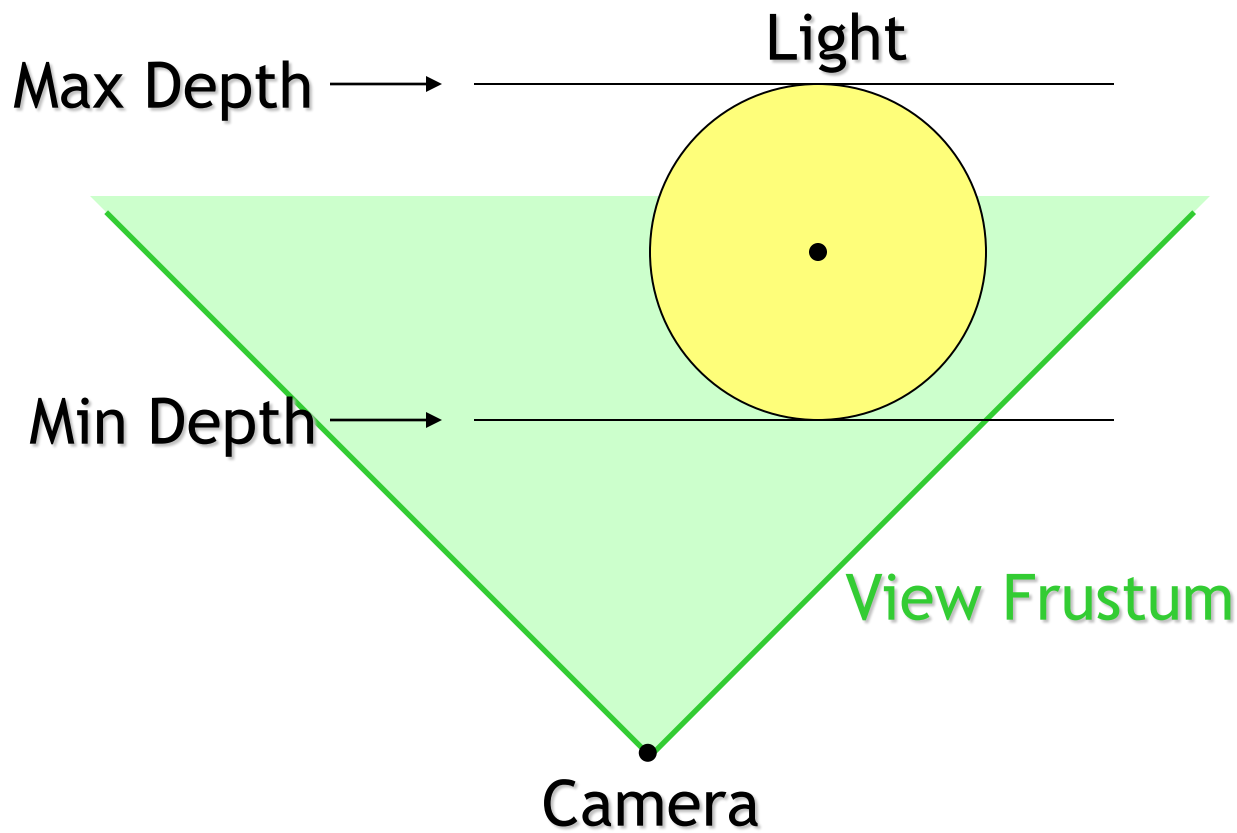


Light Scissor

Shadow volume fill reduced significantly



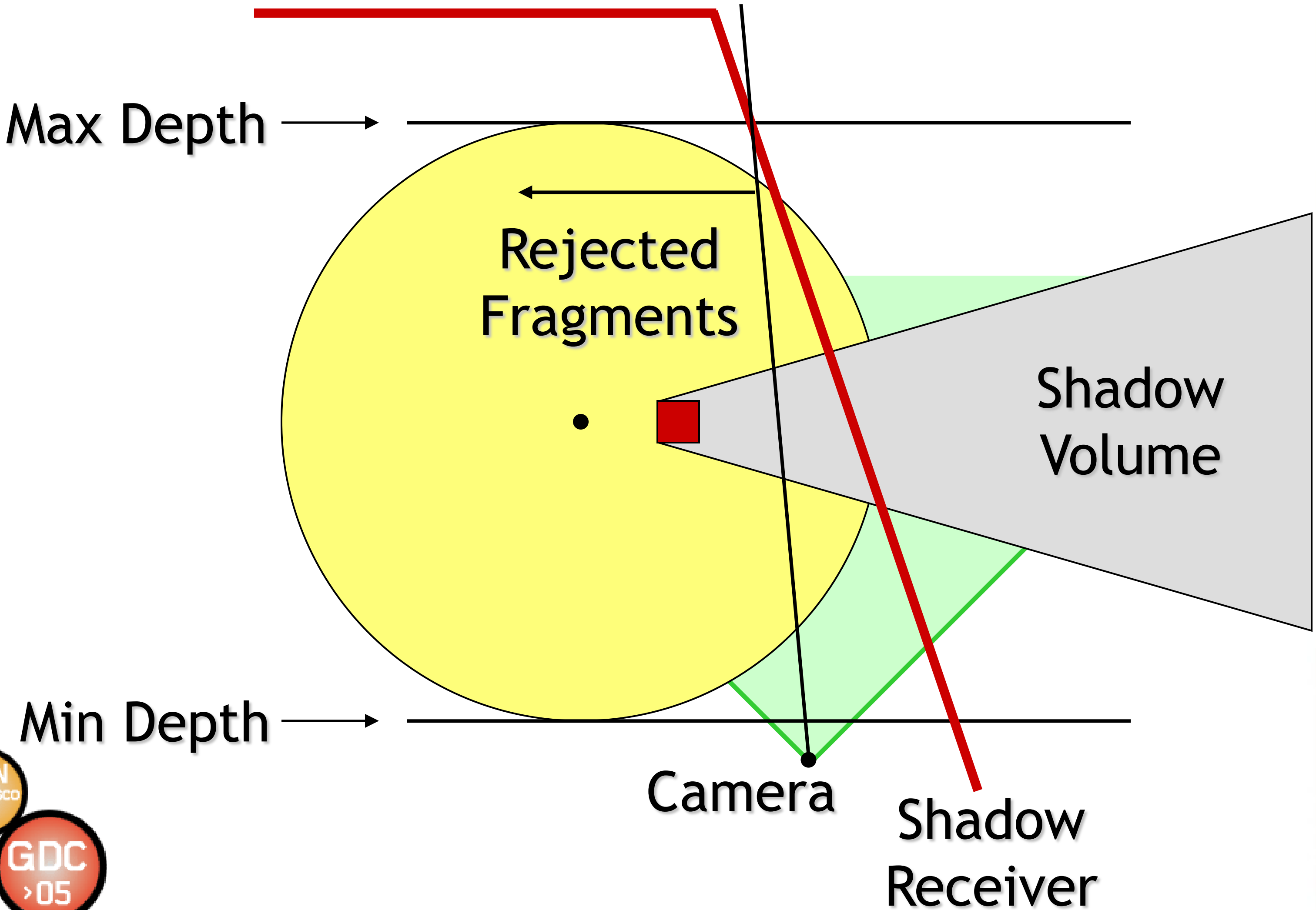
Depth Bounds Test



Depth Bounds Test

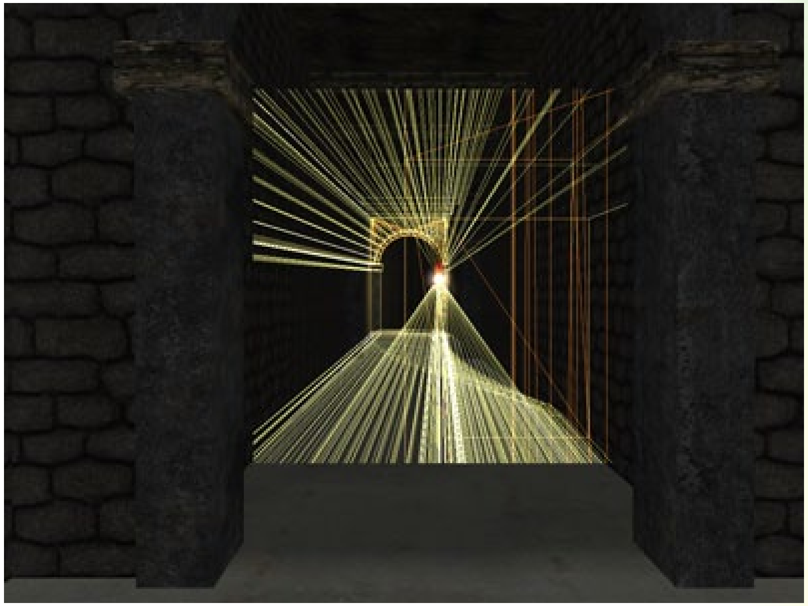
- Like a z scissor, but...
- Operates on values already in the depth buffer, *not* the depth of the incoming fragment
- Saves writes to the stencil buffer when shadow-receiving geometry is out of range

Depth Bounds Test



No Depth Bounds Test

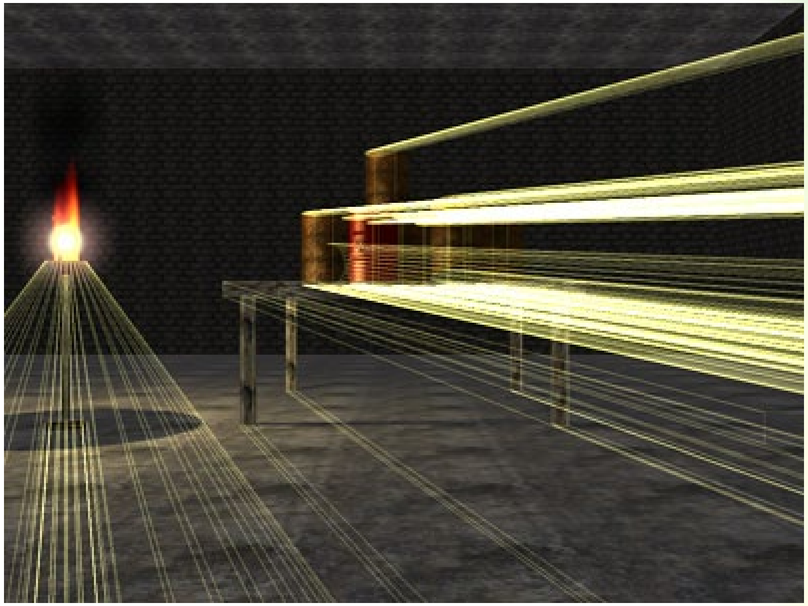
Shadow volumes extend closer to and further from camera than necessary



Depth Bounds Test

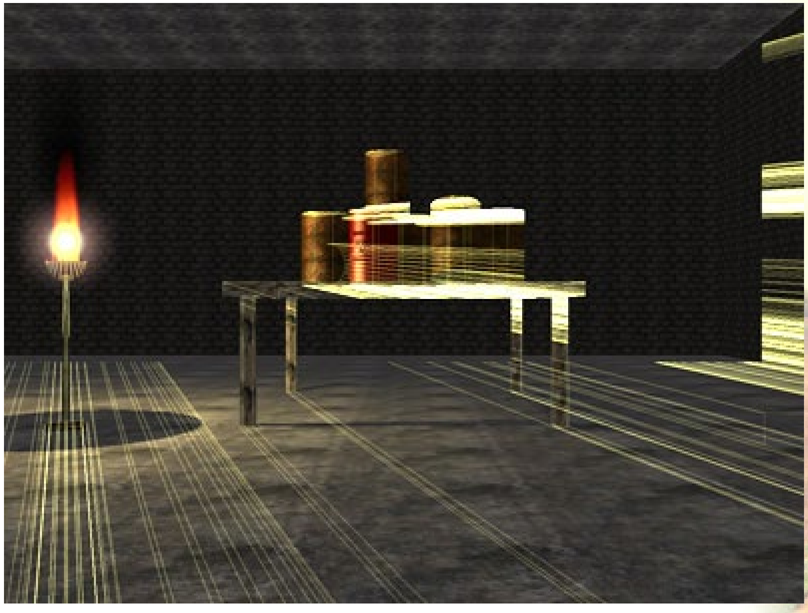
Shadow volume fill outside depth bounds is removed

No Depth Bounds Test



A lot of extra shadow volume fill where we know it can't have any effect

Depth Bounds Test



Parts that can't possibly intersect the environment removed

Depth Bounds Test

- Depths bounds specified in viewport coordinates
- To get these from camera space, we need to apply projection matrix and viewport transformation
- Apply to points (0,0,*z*,1)

Depth Bounds Test

- Let P be the projection matrix and let [d_{min}, d_{max}] be the depth range
- Viewport depth *d* corresponding to camera space z is given by

$$d = \frac{d_{max} - d_{min}}{2}\left(\frac{P_{33}z + P_{34}}{P_{43}z + P_{44}}\right) + \frac{d_{max} + d_{min}}{2}$$

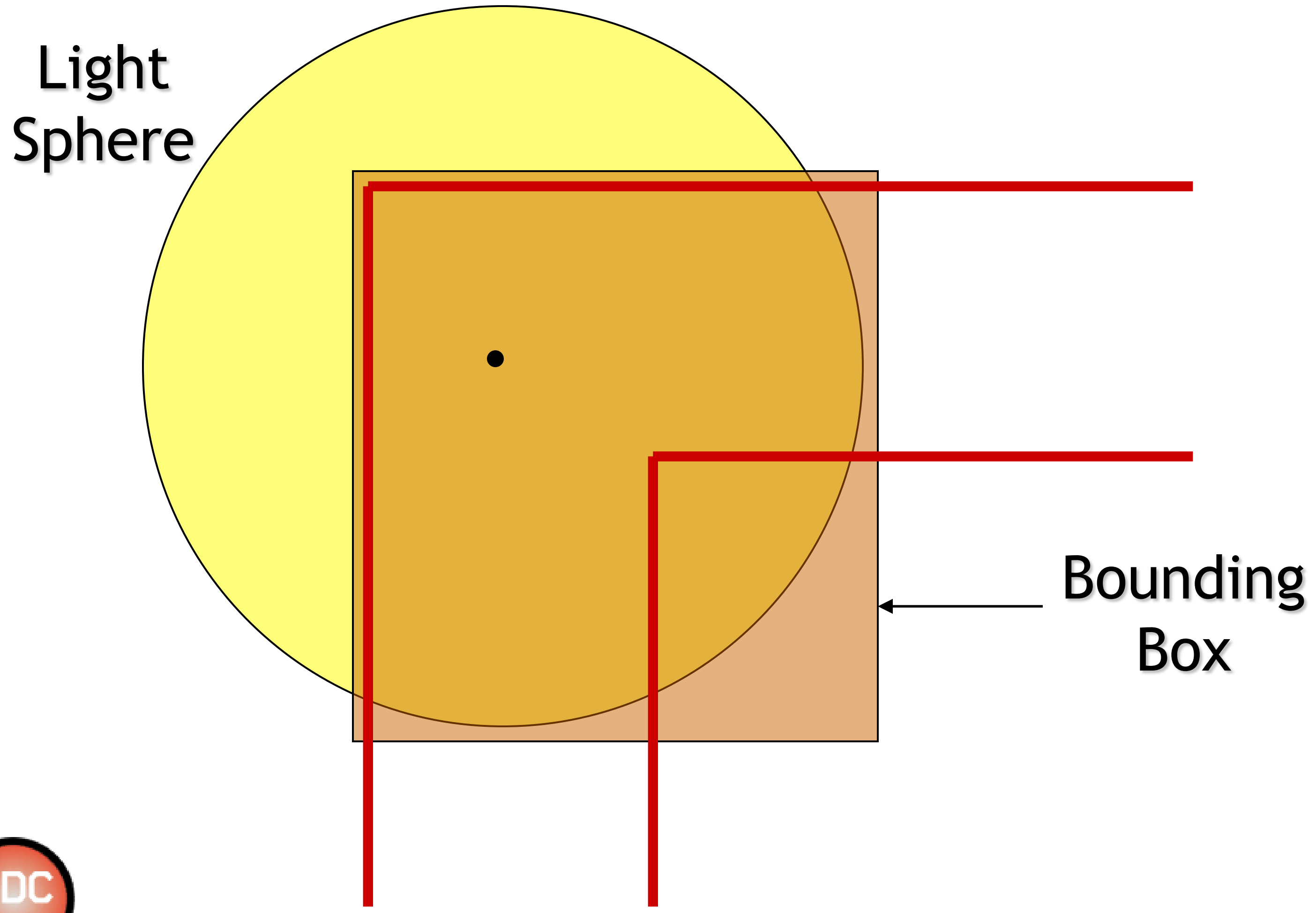
Geometry Scissor

- We can do much better than a single scissor rectangle per light
- Calculate a scissor rectangle for each geometry casting a shadow

Geometry Scissor

- Define a bounding box for the light
  - Doesn't need to contain the entire sphere of influence, just all geometry that can receive shadows
  - For indoor scenes, the bounding box is usually determined by the locations of walls

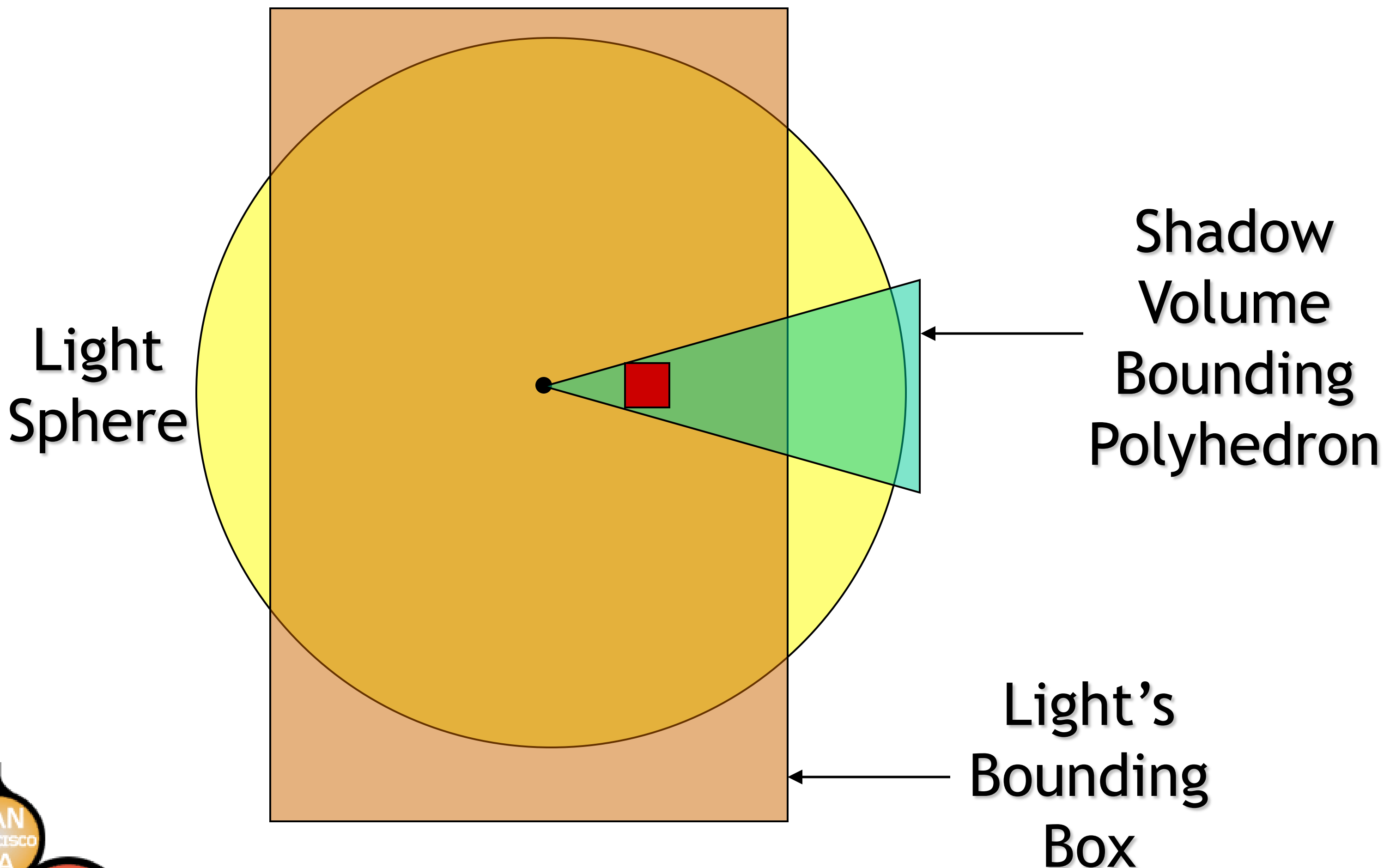
Geometry Scissor



Geometry Scissor

- For each geometry, define a simple bounding polyhedron for its shadow volume
  - Construct a pyramid with its apex at the light's position and its base far enough away to be outside the light's sphere of influence
  - Want pyramid to be as tight as possible around geometry

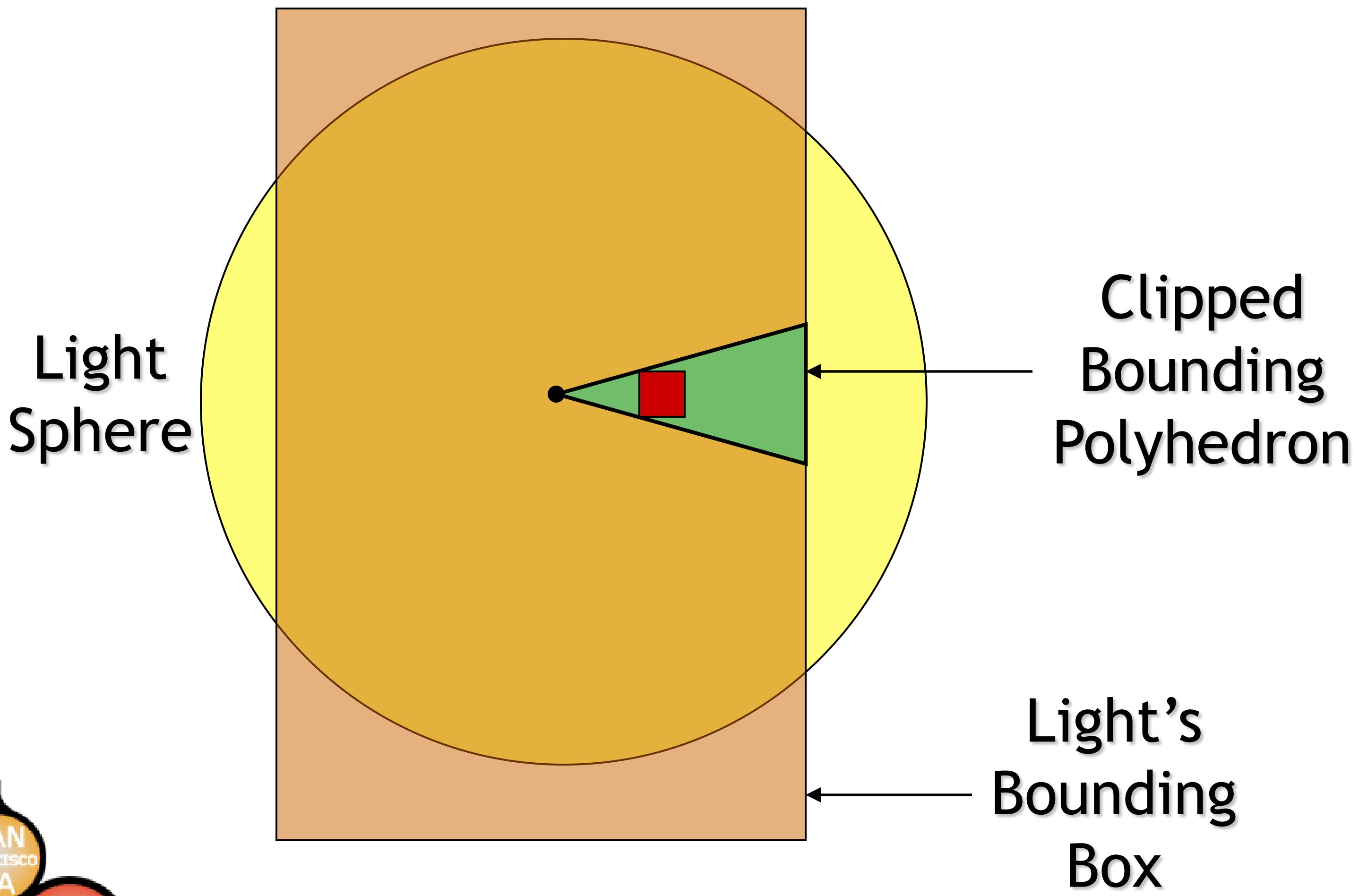
Geometry Scissor



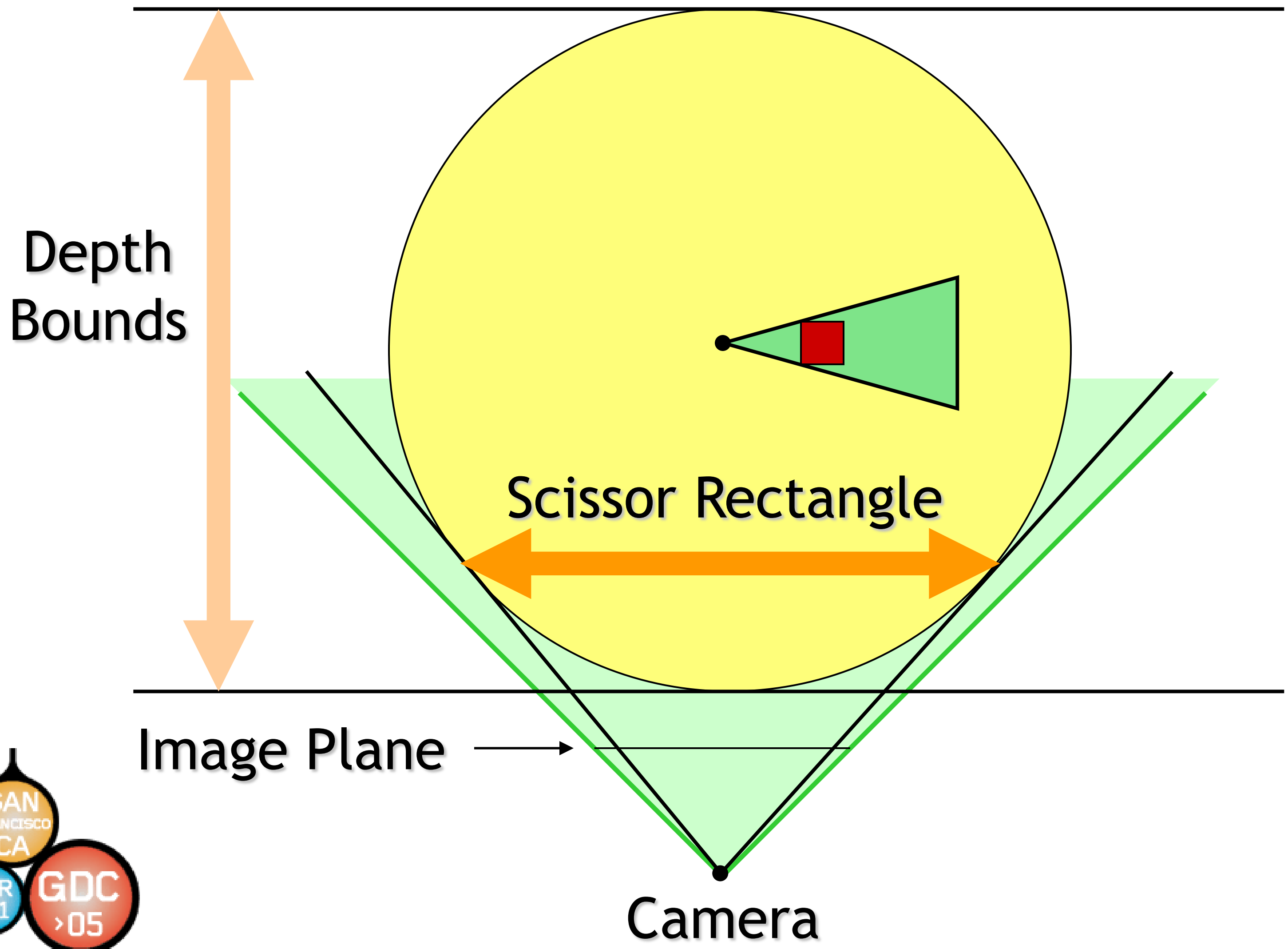
Geometry Scissor

- Clip shadow volume's bounding polyhedron to light's bounding box
- Project vertices of resulting polyhedron onto image plane
- This produces the boundary of a much smaller scissor rectangle
- Also gives us a much smaller depth bounds range

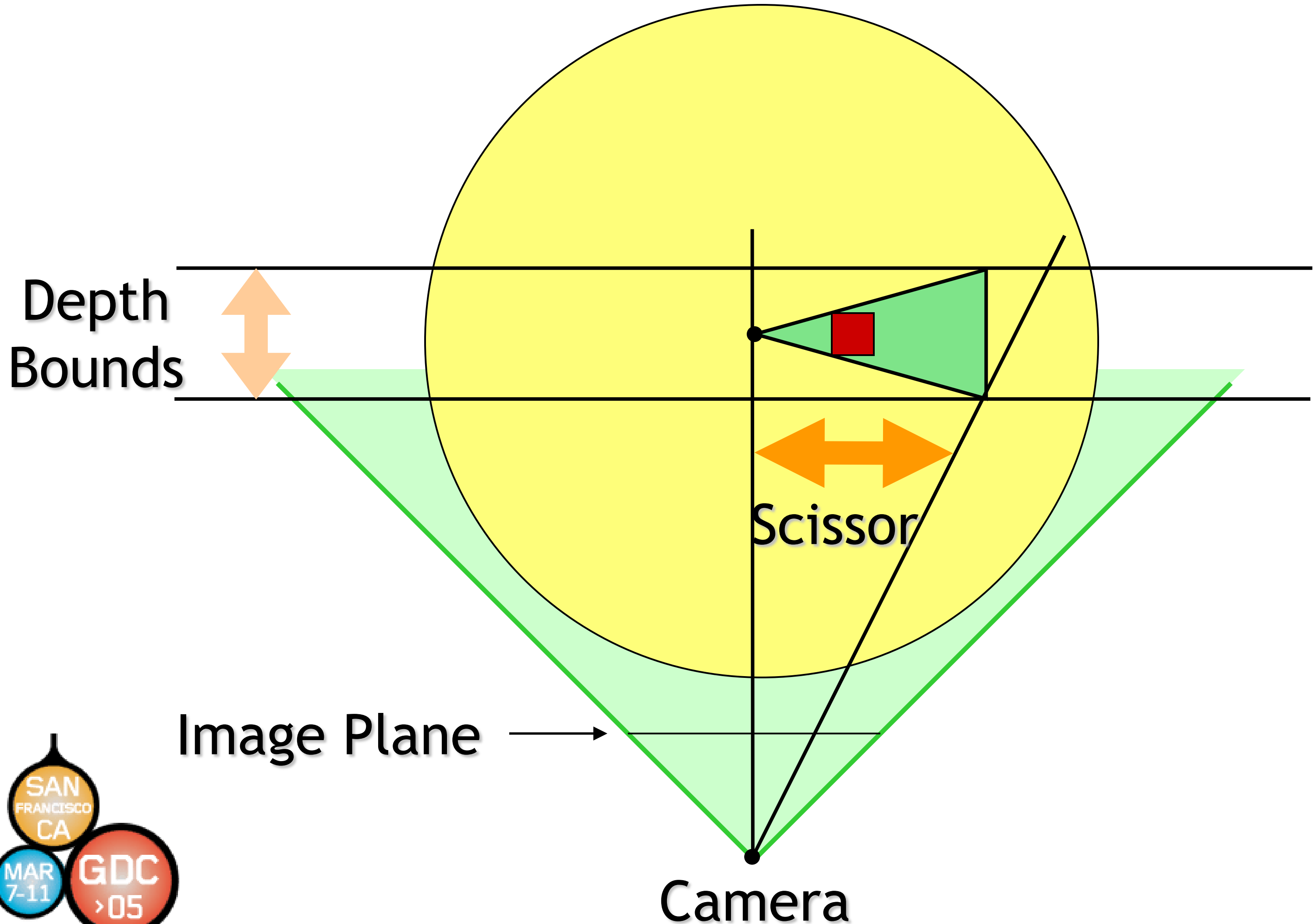
Geometry Scissor



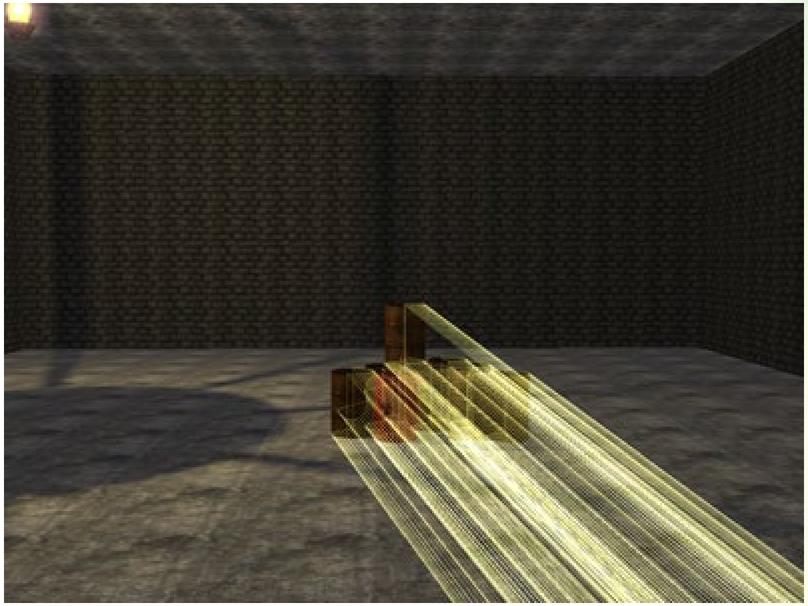
Geometry Scissor



Geometry Scissor

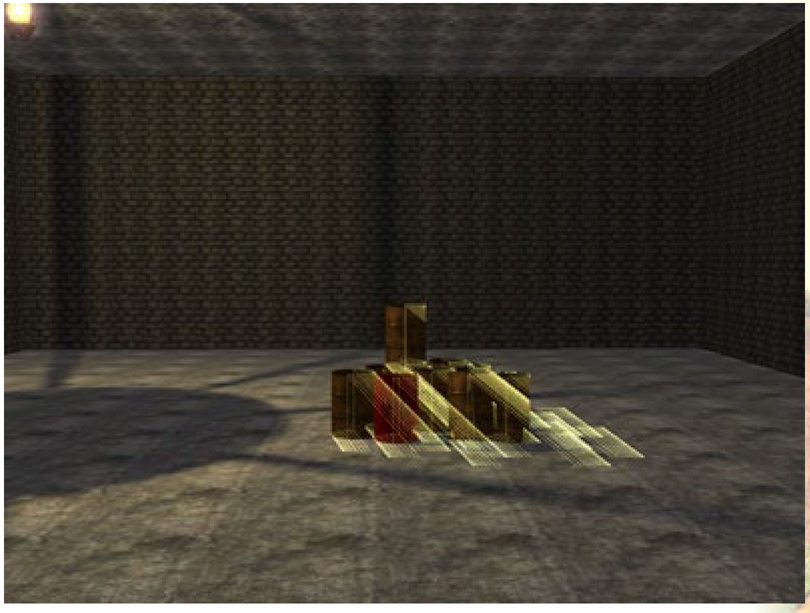


No Geometry Scissor



Light scissor rectangle and depth bounds test are no help at all in this case

Geometry Scissor



Shadow volume fill drastically reduced

Scissor and Depth Bounds

- Performance increase for ordinary stencil shadows not spectacular
- Real-world scenes get about 5-8% faster using per-geometry scissor and depth bounds test
- Hardware is doing very little work per fragment, so reducing number of fragments is not a huge win

Scissor and Depth Bounds

- For penumbral wedge rendering, it's a different story
- We will do much more work per fragment, so eliminating a lot of fragments really helps
- Real-world scenes can get 40-45% faster using per-geometry scissor and depth bounds test

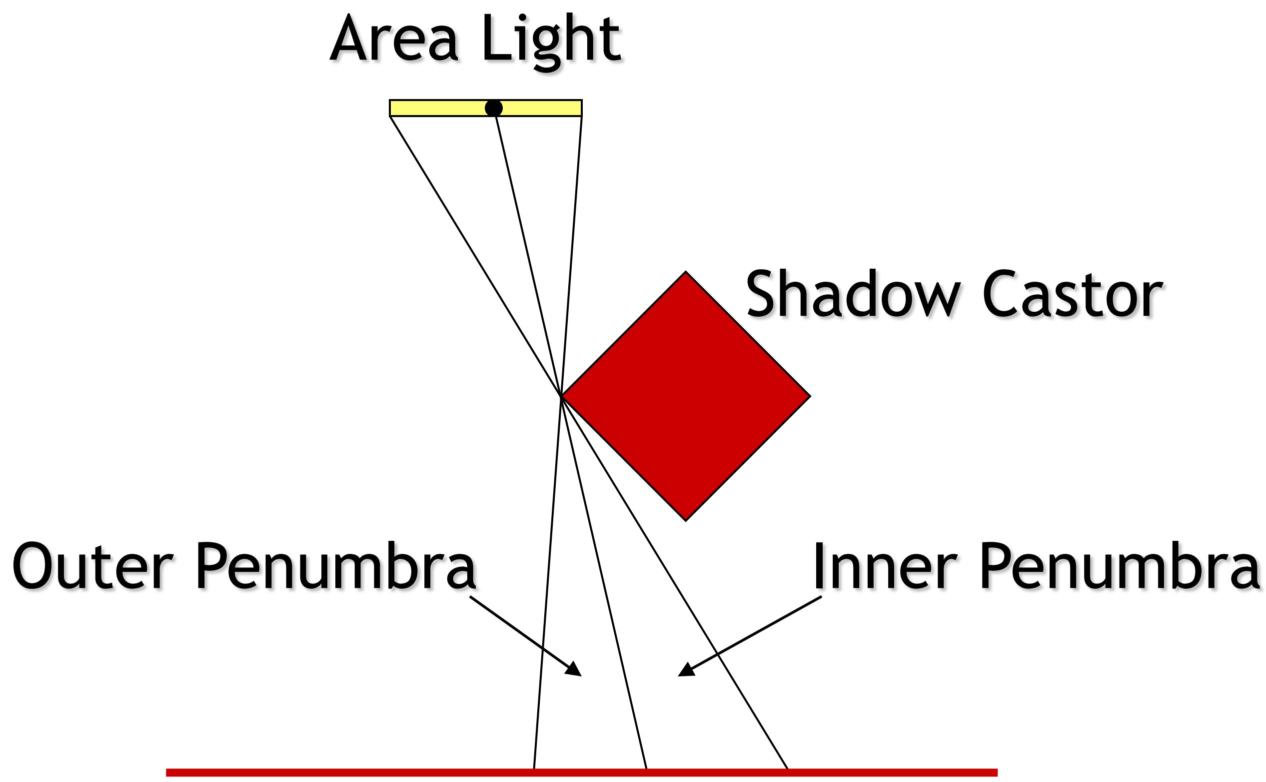
Penumbral Wedge Shadows

- 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

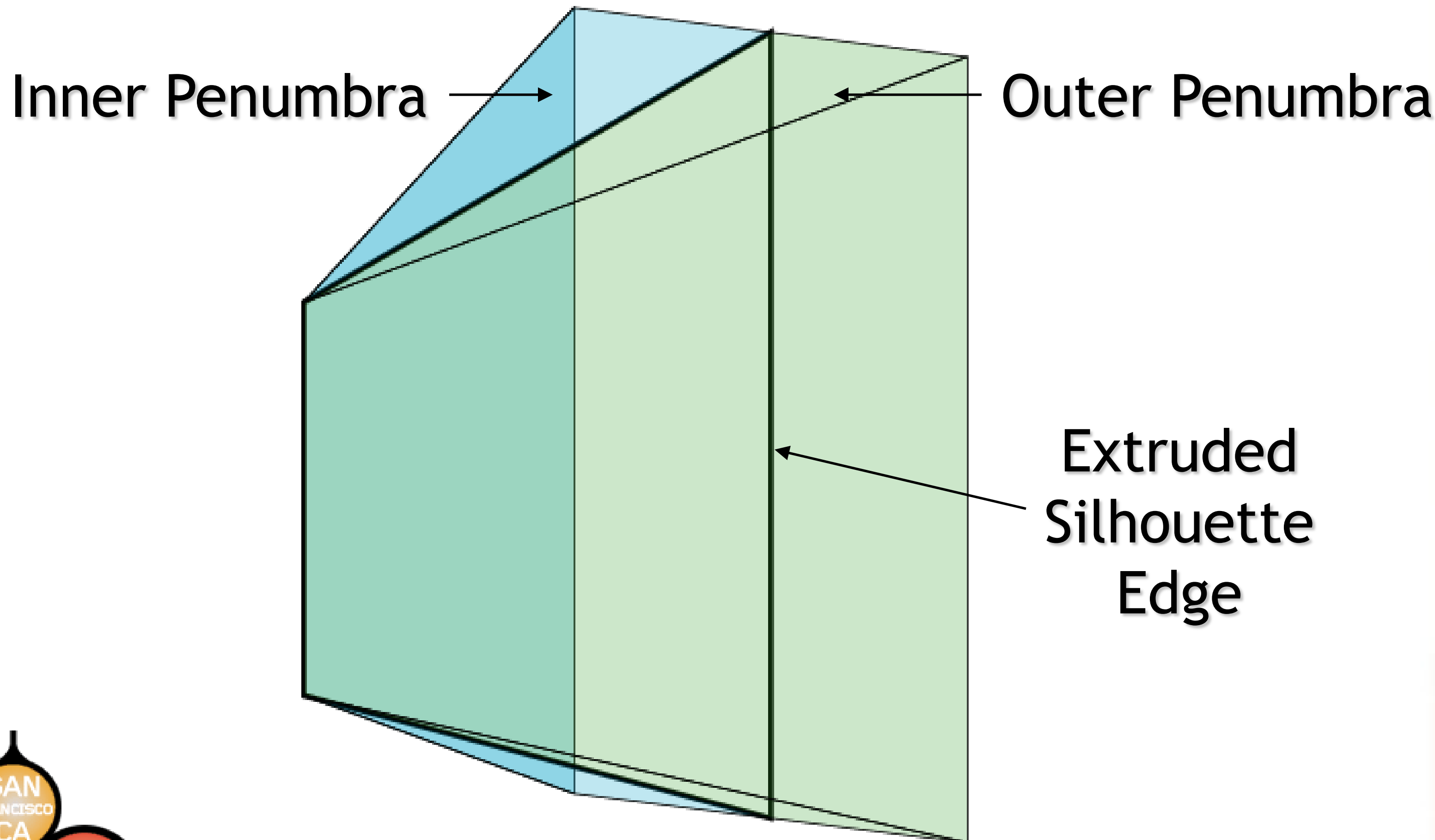
Penumbral Wedge Shadows

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

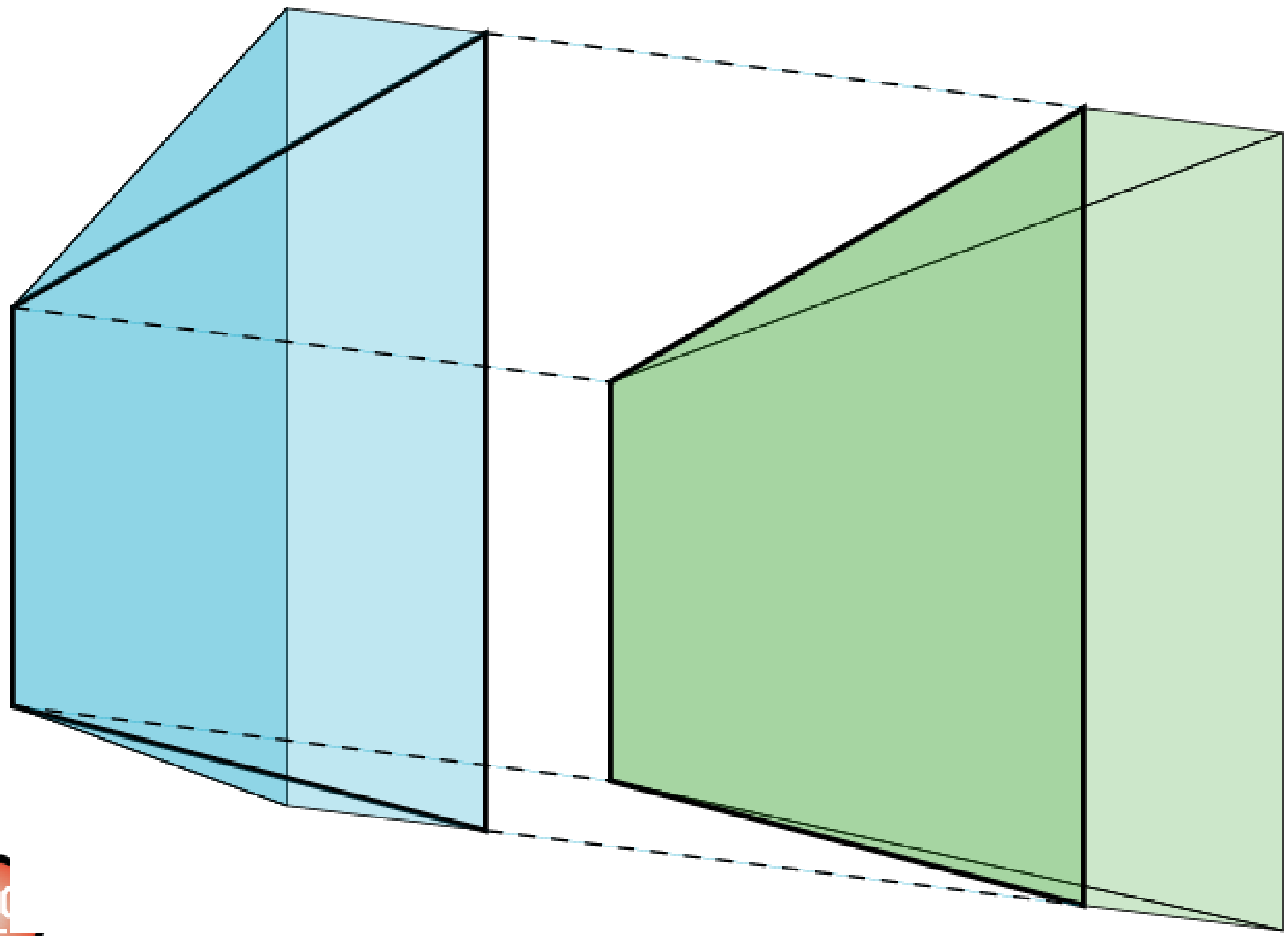
Penumbral Wedge Shadows



A Penumbral Wedge

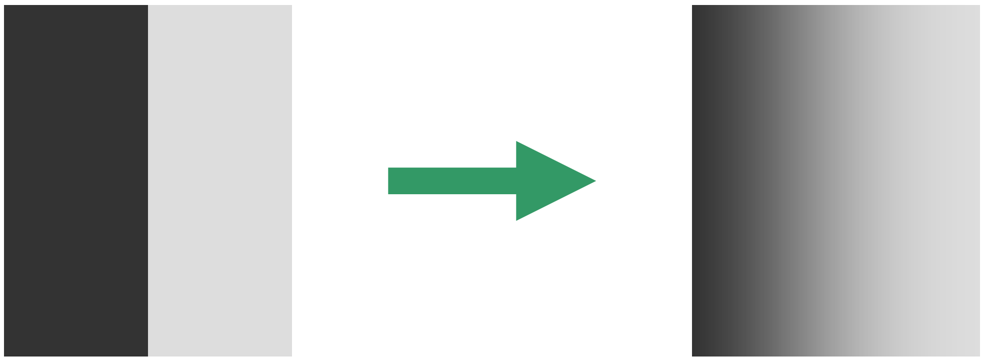


A Penumbral Wedge



Soft Shadow Correction

- Darken area inside outer penumbra
- Lighten area inside inner penumbra



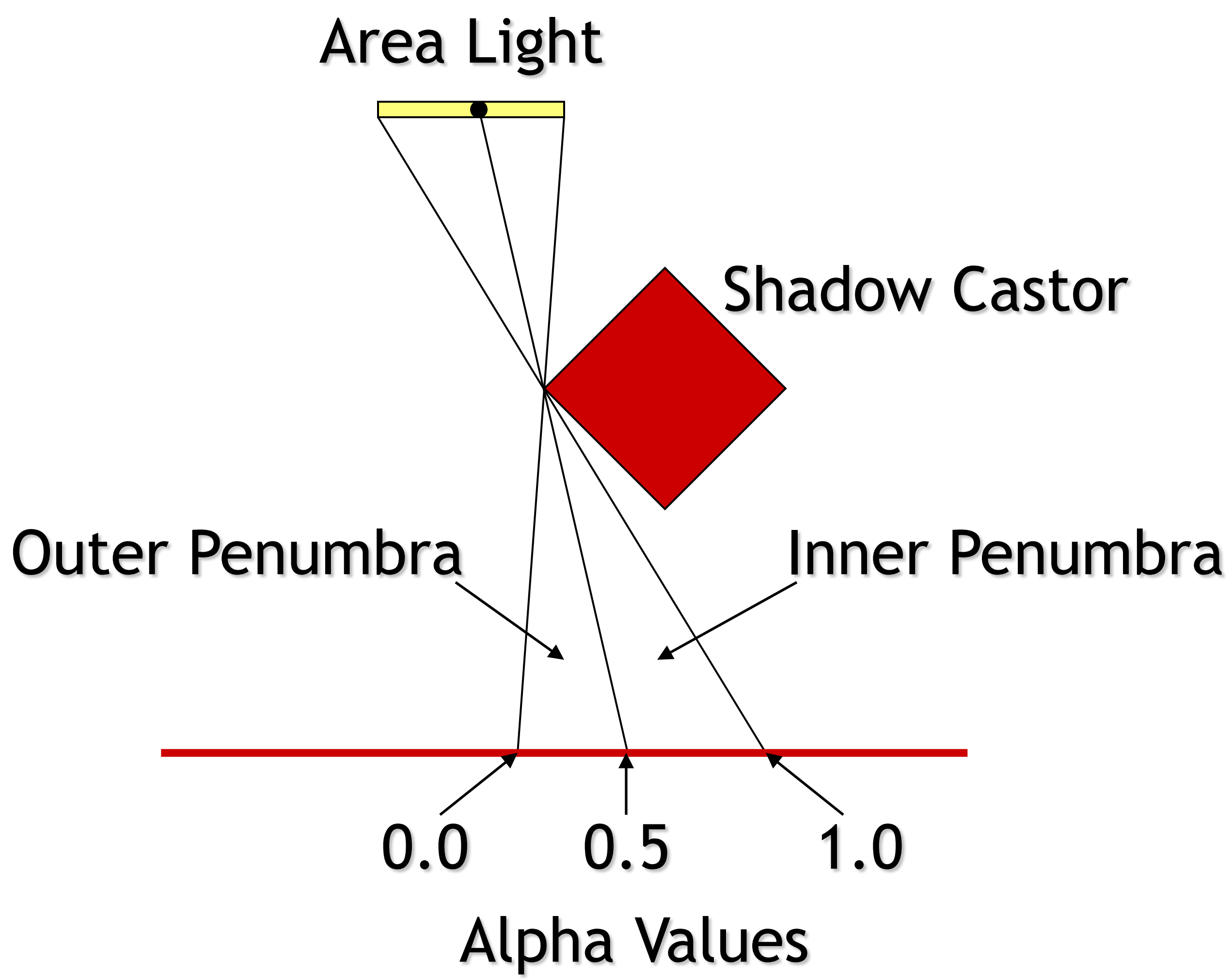
Soft Shadow Correction

- 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

Soft Shadow Correction

- 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



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

Soft Shadow Correction

- 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…

Soft Shadow Correction

- 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

Soft Shadow Correction

- Rendering shadow volumes into a floating-point render target
  - Requires hardware that can do this
  - We need floating-point blending
  - We lose two-sided rendering unless we can access a facing register
  - We lose double-speed rendering

Soft Shadow Correction

- 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

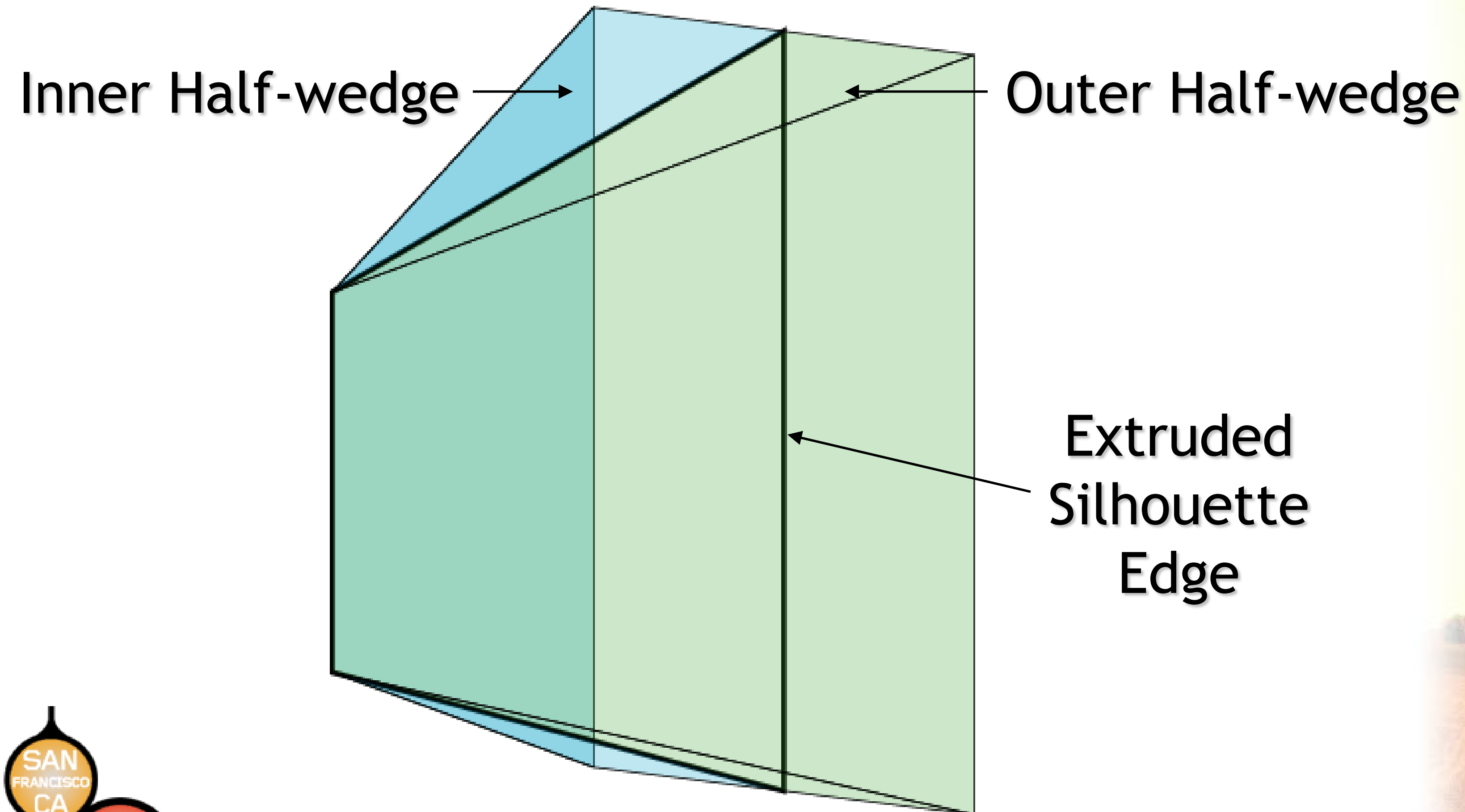
Soft Shadow Correction

- 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

Penumbral 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

Penumbral Wedge Rendering



Penumbral 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

Penumbral 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

Penumbral Wedge Rendering

- What's a viewport-space point in the frame buffer?
  - The *x* and *y* viewport coordinates are available to fragment programs in the `fragment.position` register
  - We need to read the *z* coordinate from a depth texture
  - The coordinates (*x*, *y*, *z*, 1) give the location of the point already rendered

Penumbral Wedge Rendering

- Bounding plane tests

```
!!ARBfp1.0
TEMP  vssp, temp;
TEX   vssp.z, fragment.position, texture[0], RECT;
SWZ   vssp.xyw, fragment.position, x, y, 0, 1;
DP4   temp.x, vssp, fragment.texcoord[0];
DP4   temp.y, vssp, fragment.texcoord[1];
DP4   temp.z, vssp, fragment.texcoord[2];
KIL   temp.xyzz;
```

Penumbral Wedge Rendering

- Early-out code sequence (Nvidia)

```
!!ARBfp1.0
OPTION NV_fragment_program2
TEMP  vssp, temp;
TEX   vssp.z, fragment.position, texture[0], RECT;
SWZ   vssp.xyw, fragment.position, x, y, 0, 1;
DP4C  temp.x, vssp, fragment.texcoord[0];
DP4C  temp.y, vssp, fragment.texcoord[1];
DP4C  temp.z, vssp, fragment.texcoord[2];
RET   (LE.xyzz);
```

Penumbral Wedge Rendering

- In preceding code, `texture[0]` is a copy of the depth buffer
- Texture coordinates 0, 1, 2 hold the 4-component plane vectors for the three outside bounding planes
  - If the dot product between the surface point and any plane is negative, then the point is outside the half-wedge

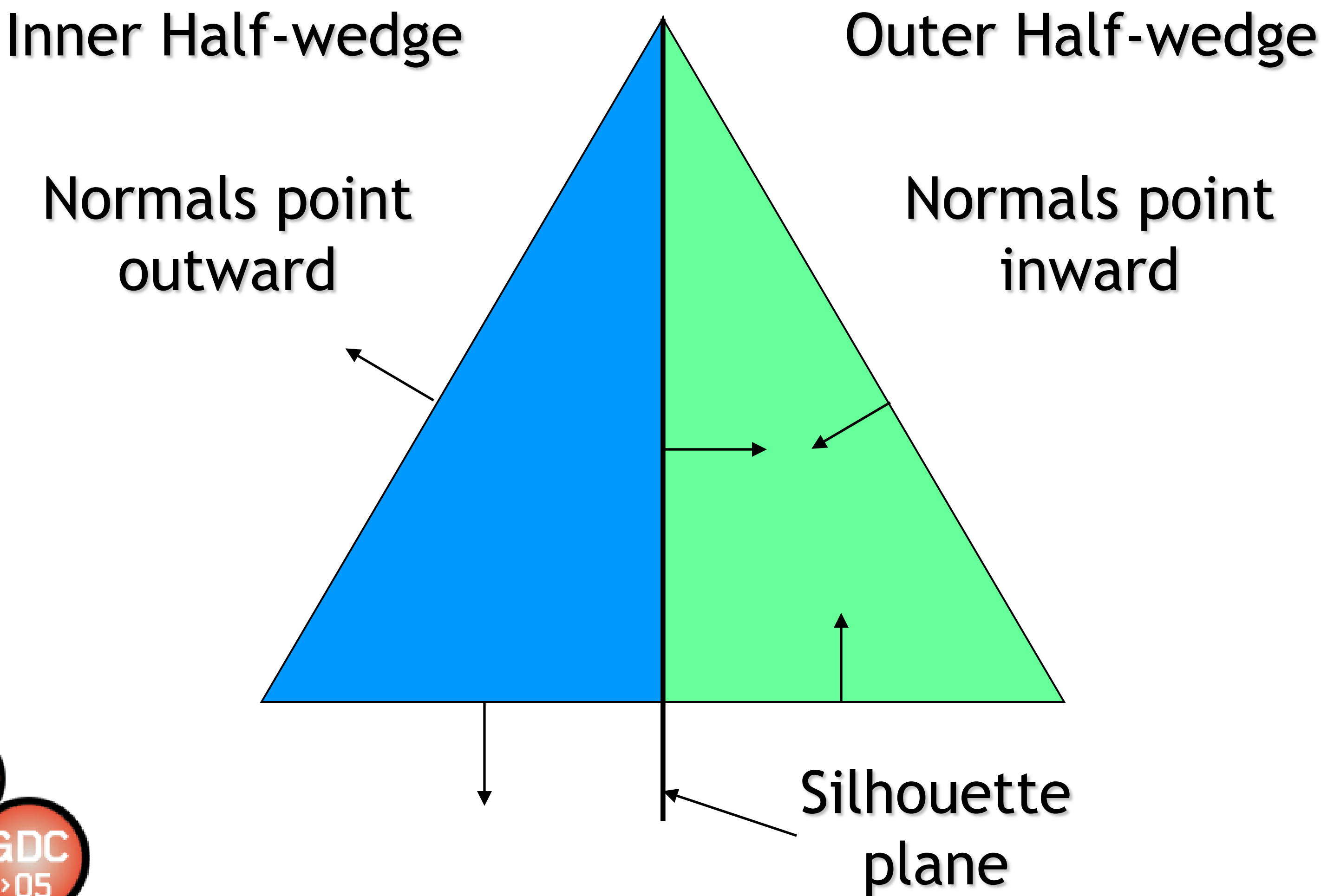
Penumbral Wedge Rendering

- Still have extruded silhouette plane to worry about
  - We take care of it using the *z* test
  - Render inner half-wedges and outer half-wedges separately
  - For both groups of half-wedges divide into two batches...

Penumbral 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

Penumbral Wedge Rendering

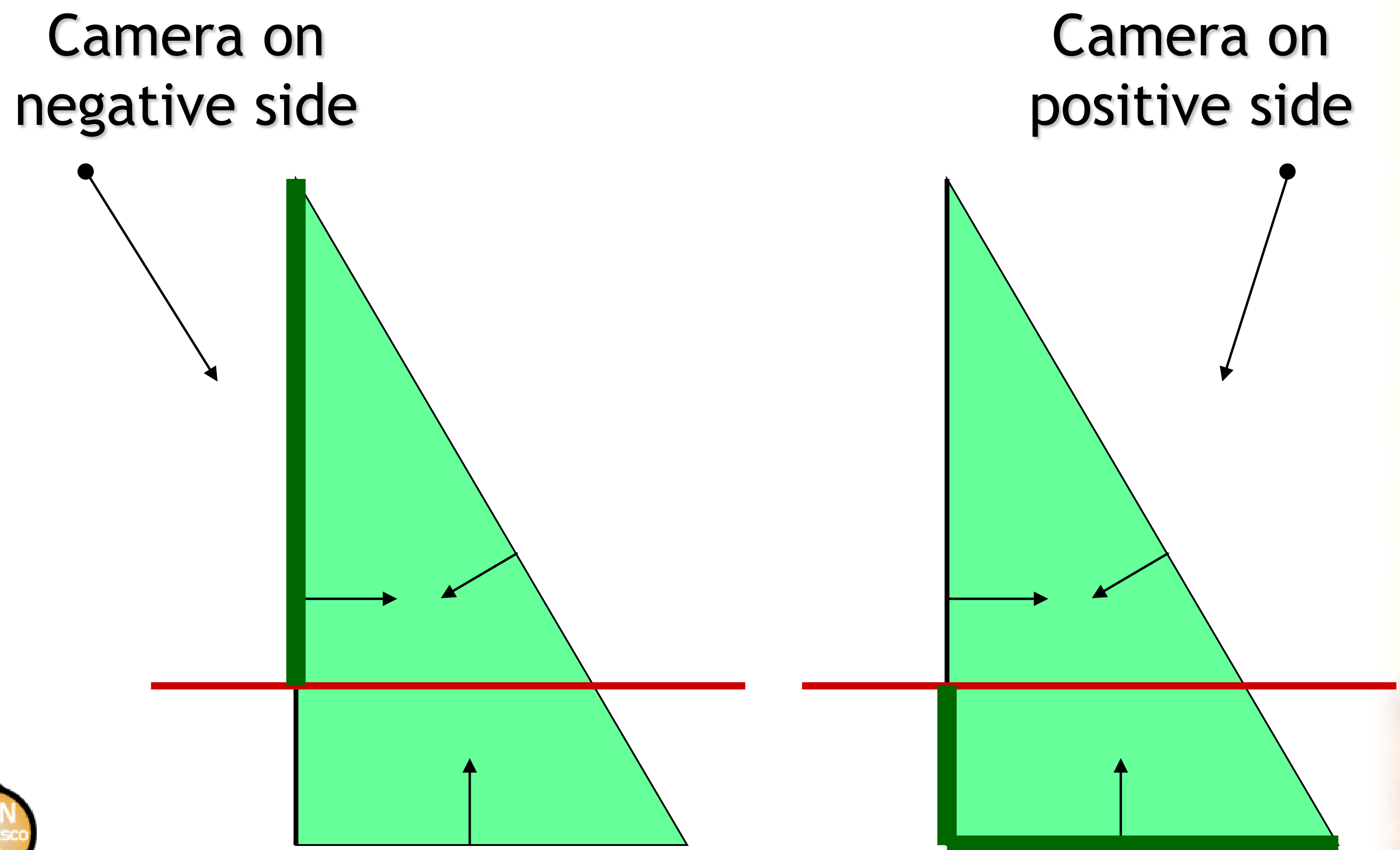


Rendering Outer Half-wedges

- 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

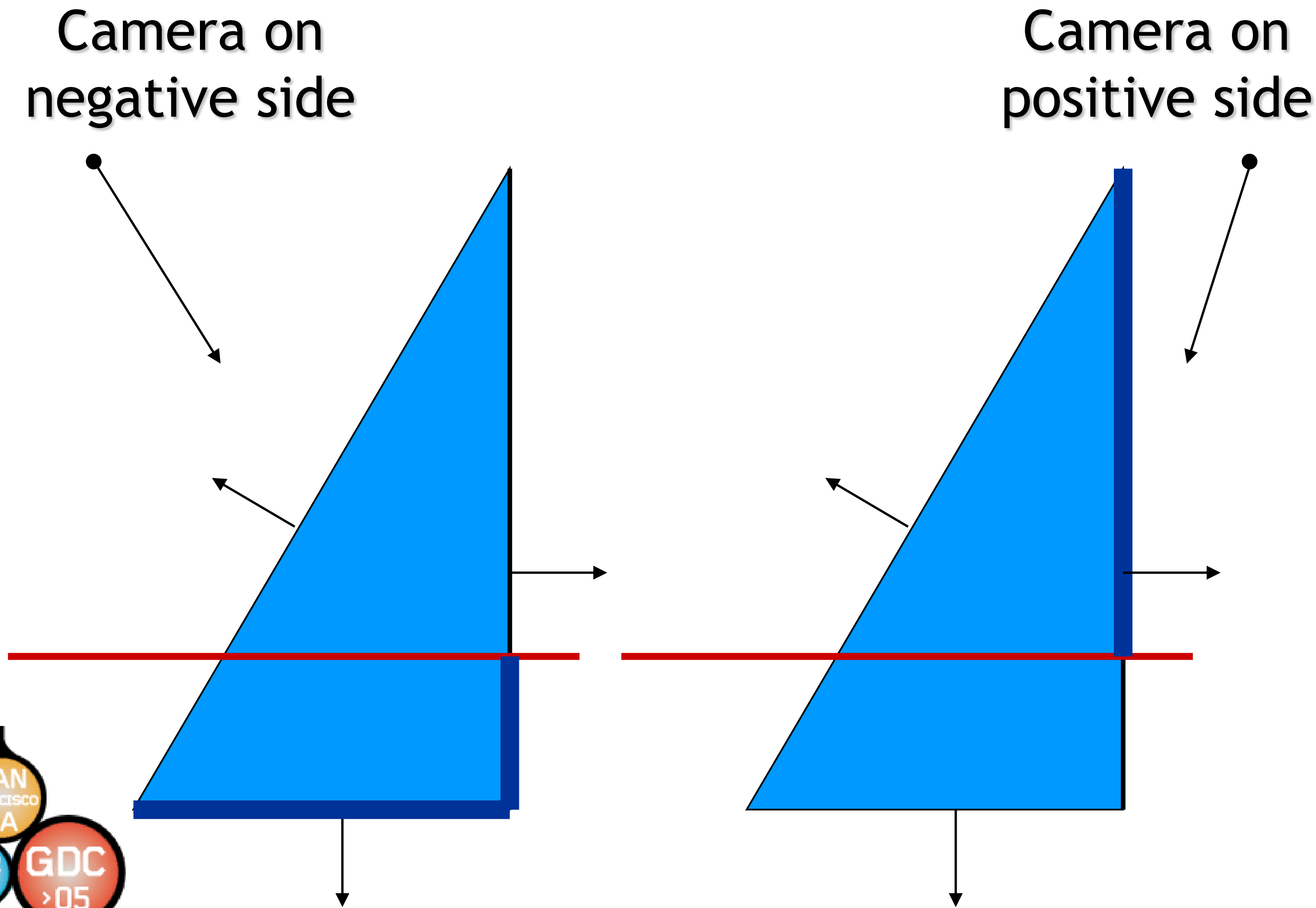
Rendering Outer Half-wedges



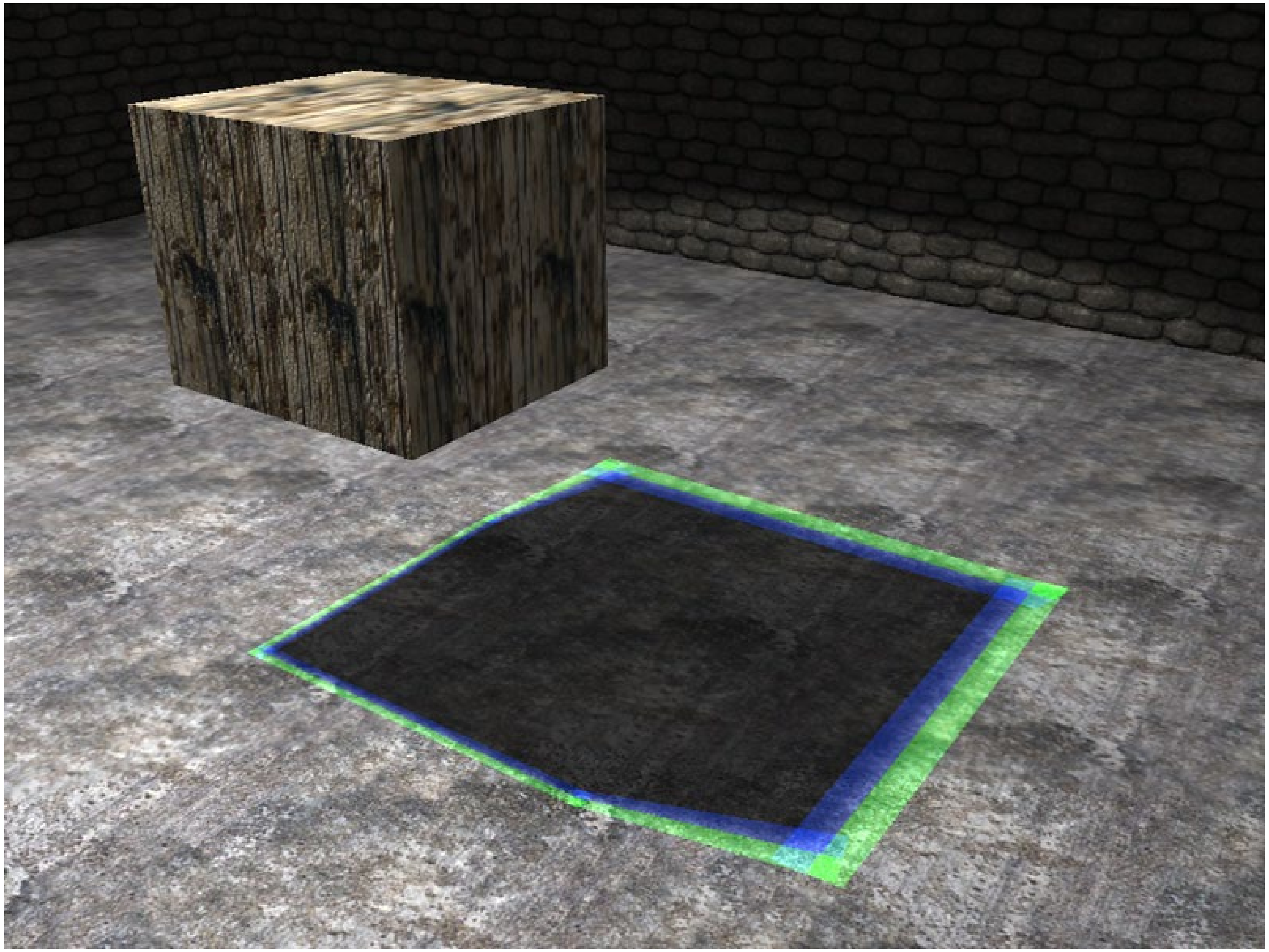
Rendering Inner Half-wedges

- Half-wedges for which camera is on positive side of silhouette plane
  - Render front faces when *z* test passes
- Half-wedges for which camera is on negative side of silhouette plane
  - Render back faces when *z* test fails

Rendering Inner Half-wedges



Penumbral Wedge Rendering



Penumbral Wedge Rendering

- How much do we add or subtract?
- For each pixel covered by an inner half-wedge, we subtract the fraction of light that is visible
- For each pixel covered by an outer half-wedge, we add the fraction of light that is occluded

Area Light Occlusion

- First, we need to transform the surface point into local light space
- Recall that we have the coordinates in viewport space:

```
TEX   vssp.z, fragment.position, texture[0], RECT;
SWZ   vssp.xyw, fragment.position, x, y, 0, 1;
```

Area Light Occlusion

- Precalculate the transformation from viewport space to light space
- Apply in fragment program:

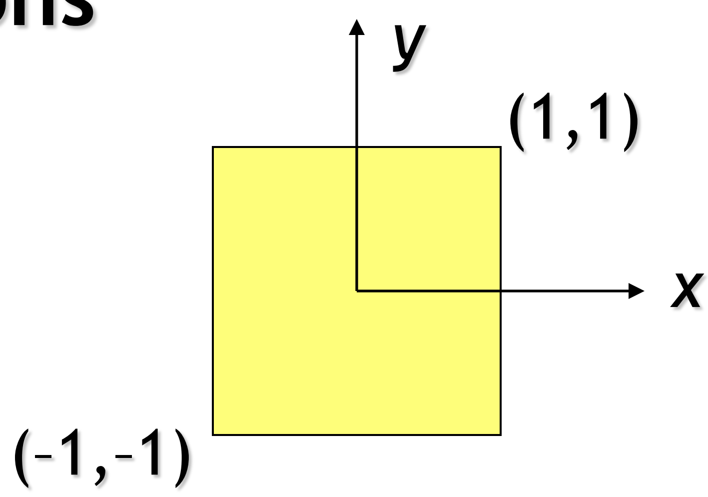
```
TEMP    lssp;
DP4     lssp.x, vssp, xform_light[0];
DP4     lssp.y, vssp, xform_light[1];
DP4     lssp.z, vssp, xform_light[2];
DP4     lssp.w, vssp, xform_light[3];
DIV     lssp, lssp, lssp.w;
```

Area Light Occlusion

- Division by the *w* coordinate is necessary because we passed through the inverse of the projection matrix between viewport space and light space
- In light space, the *z* axis is perpendicular to the plane of the area light

Area Light Occlusion

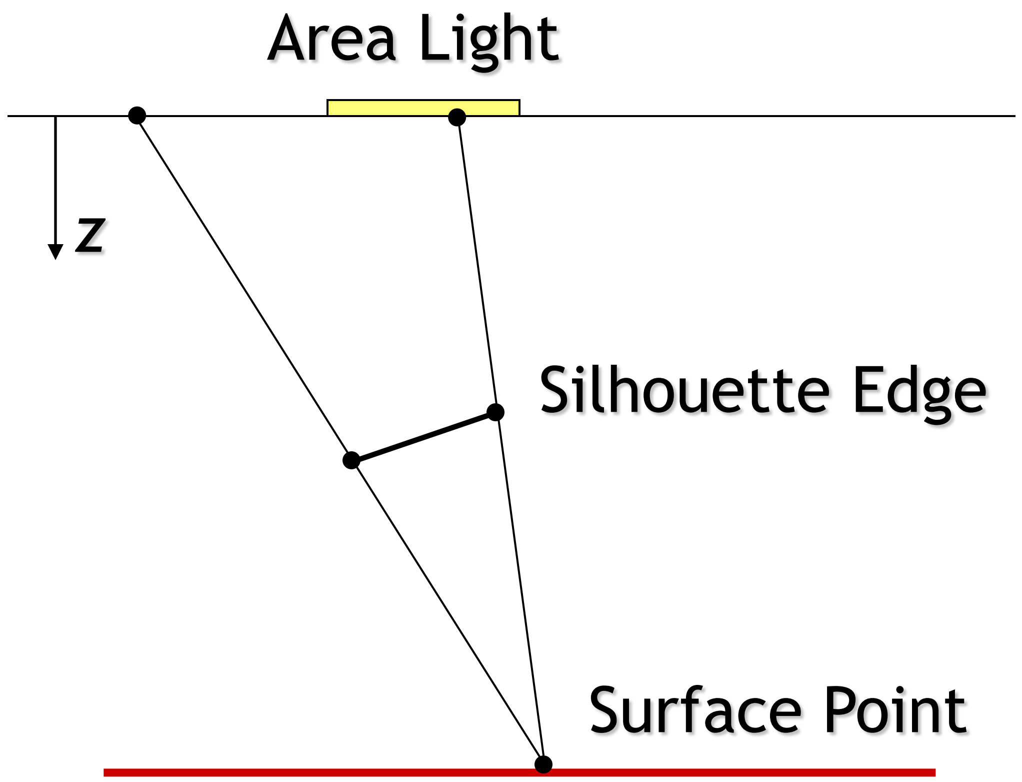
- We also adjust the transformation to light space so that an arbitrarily-sized rectangular light area is mapped into [-1,1] in both *x* and *y* directions



Area Light Occlusion

- Next, we need to project the endpoints of the silhouette edge onto the light plane
- The vertex program can transform these points from object space to light space and pass them to the fragment program

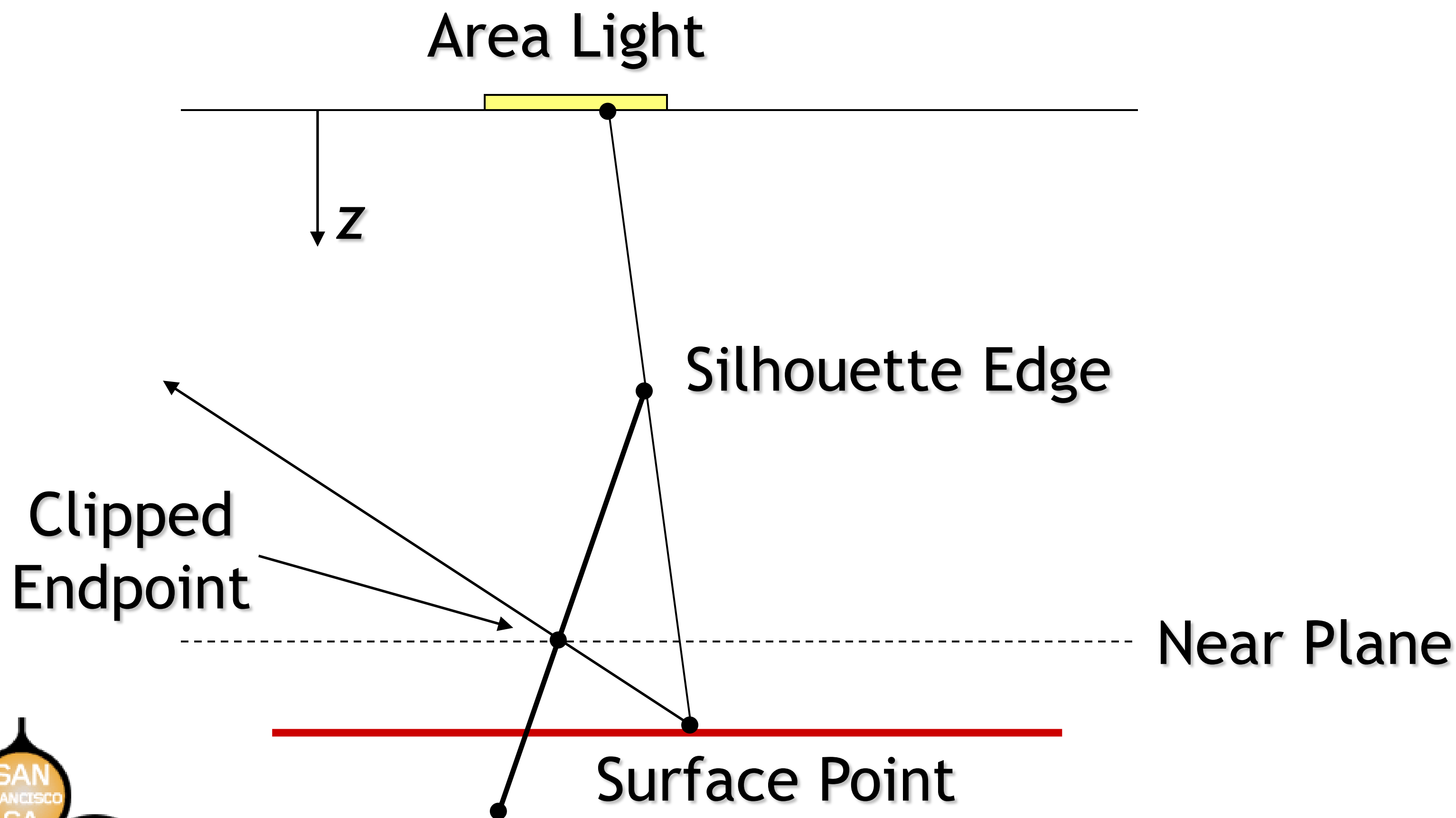
Area Light Occlusion



Area Light Occlusion

- But what if one of the endpoints can't be projected because it doesn't lie between the surface point and the light plane?
- Solution: clip the silhouette edge to a local "near plane" first

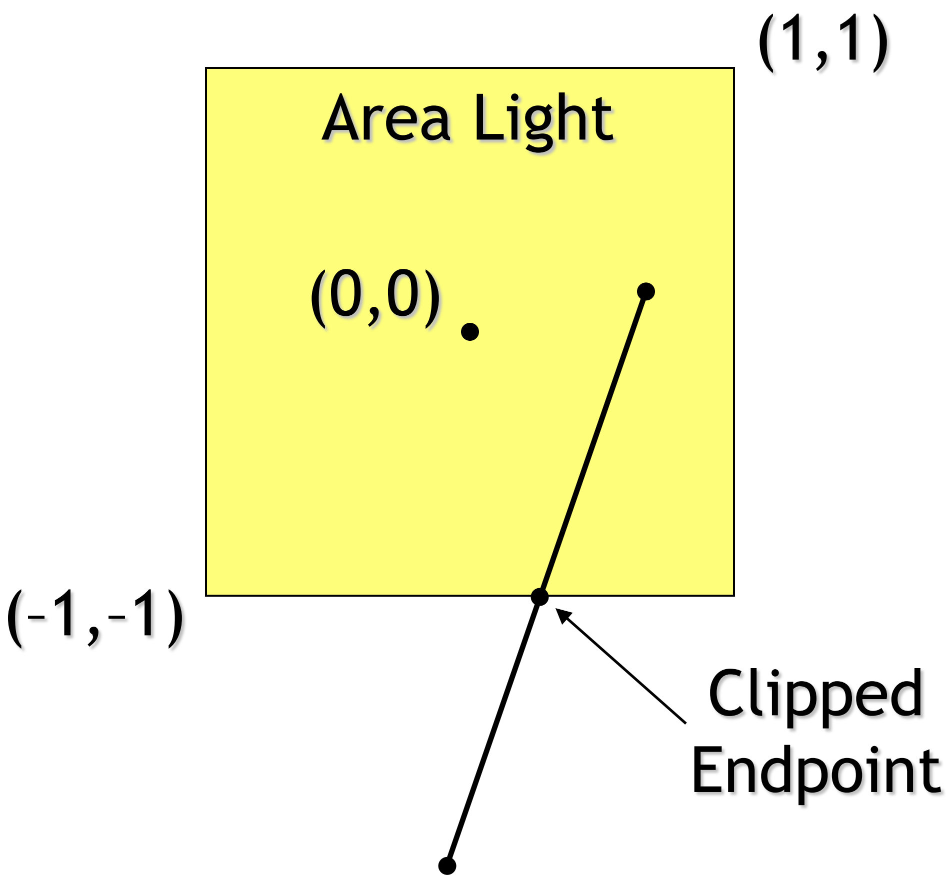
Area Light Occlusion



Area Light Occlusion

- Once endpoints have been projected onto the light plane, we are in a 2D world
- Next step is to clip the projected edge to the [-1,1] square

Area Light Occlusion



Area Light Occlusion

- Earlier implementations perform clipping in 3D against the four planes connecting the area light to the light-space surface point
  - Requires 36 fragment program instructions
  - But robust, and didn't need near plane clip step

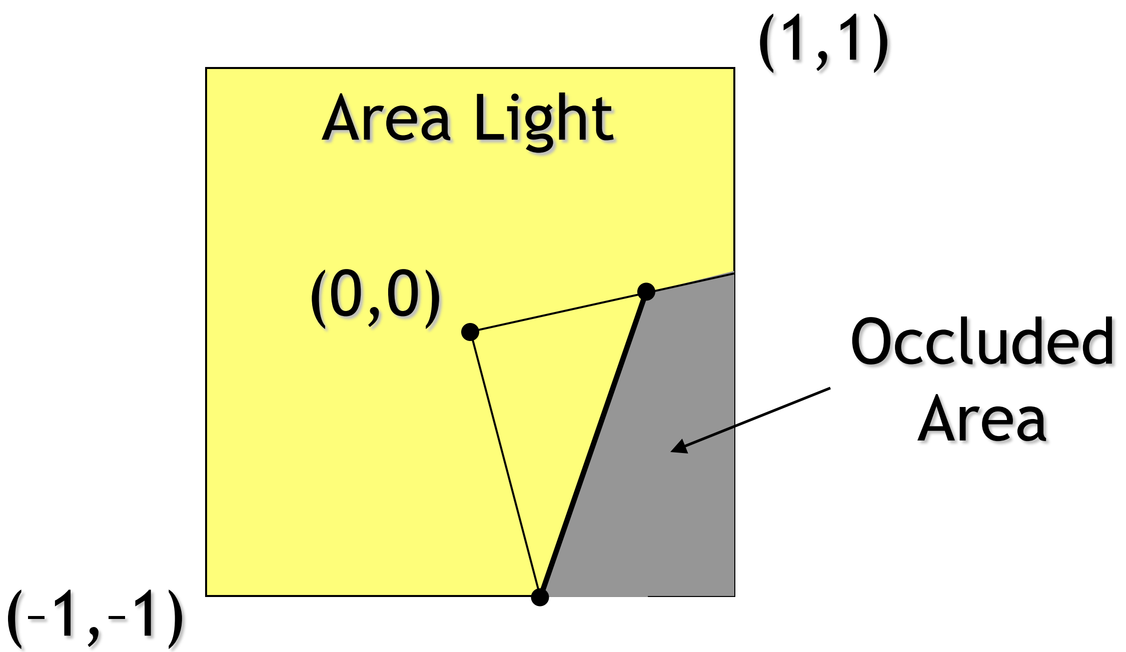
Area Light Occlusion

- However, clipping against near plane in 3D first and then clipping against the four sides of the light area in 2D is much faster
  - Total of 23 fragment program instructions
  - Also robust

Area Light Occlusion

- Last step is to determine how much of the light source is occluded by the extruded silhouette edge
- We do this by calculating the area of the sector subtended by the clipped edge

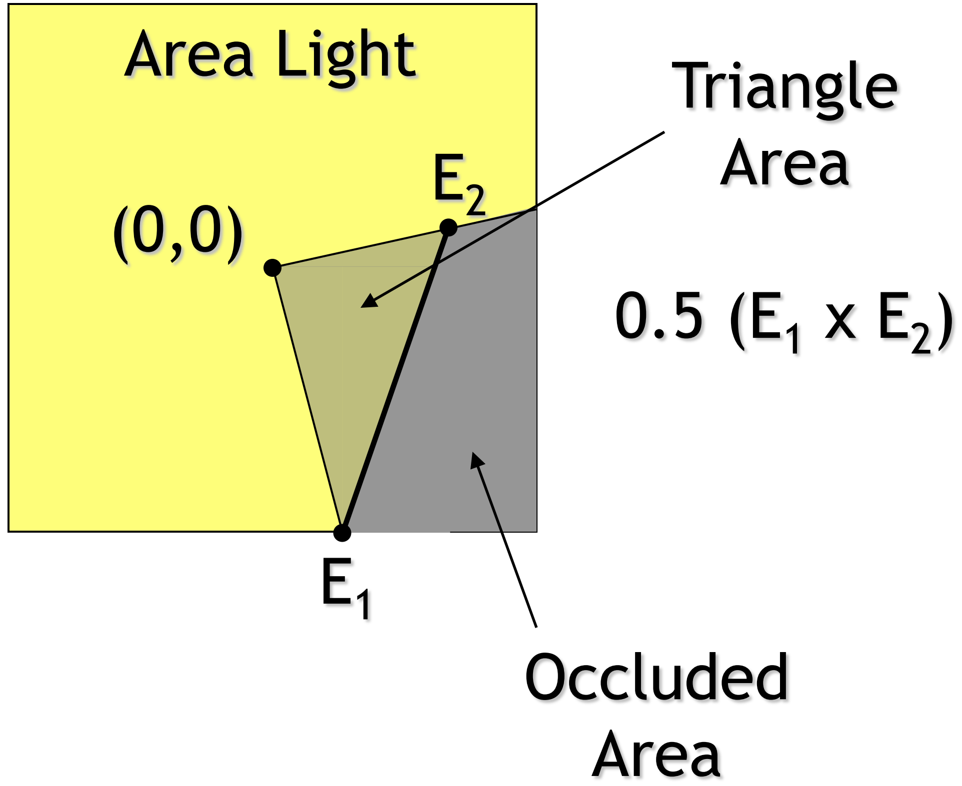
Area Light Occlusion



Area Light Occlusion

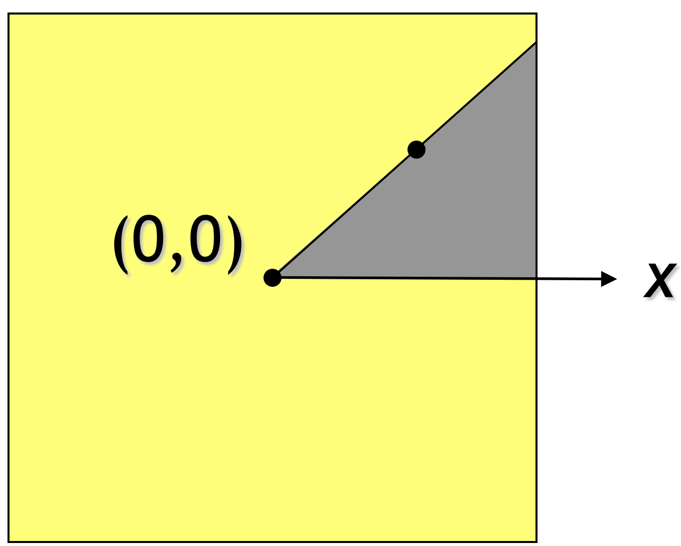
- The occluded area is equal to the total area between the two line segments connecting the center of the light and the two endpoints minus the area of the triangle formed by the center and the two endpoints
- Calling the endpoints E_1 and E_2...

Area Light Occlusion



Area Light Occlusion

- Total area between positive *x* axis and the direction to any endpoint is fetched from a cube map texture



Area Light Occlusion

- We look up the sector area for both endpoints and subtract to get the area between the lines
- Then subtract the triangle area, given by half the cross product between the two endpoints
- Scale everything by 1/8 for 3.5 bit fixed point result

Area Light Occlusion

```
# Endpoints are stored in edge.xy and edge.zw

# Look up sector areas
TEX     area.x, edge.xyxx, texture[1], CUBE;
TEX     area.y, edge.zwzz, texture[1], CUBE;

# Subtract areas and scale by 1/8
SUB     area.z, area.x, area.y;
MADC    area, |area.z|, 0.125, {-0.0625, -0.125, 0.0, 0.0};

# If area > 0.5, replace with 1 - area
MOV     area.w (GT.x), -area.y;
```

Area Light Occlusion

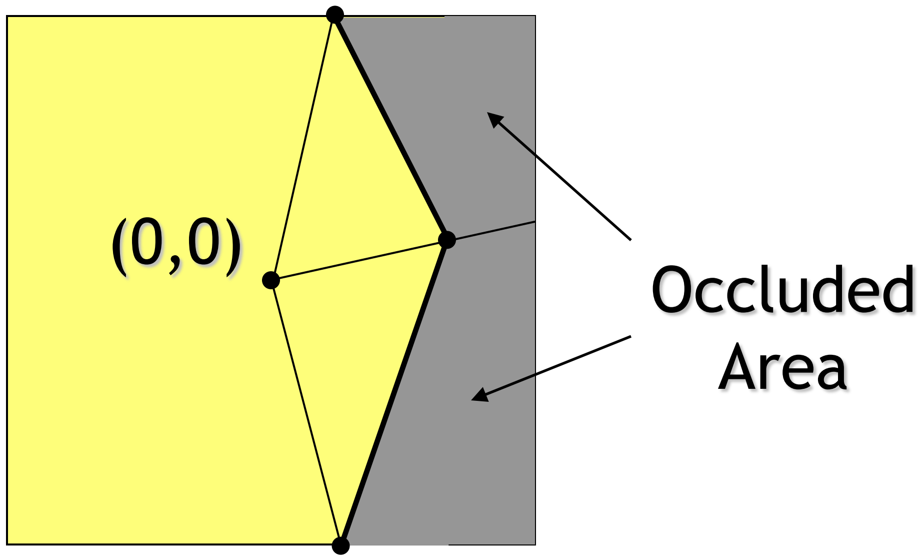
```
# Calculate area of triangle
MUL    temp.xy, edge.xyxy, edge.wzwz;
SUB    temp.w, temp.x, temp.y;

# Fractional area of triangle relative to whole
  light area is 1/8 of cross product

# Subtract it from total area and scale by
# additional factor of 1/8 for fixed point
MAD    result.color, |temp.w|, -0.015625, area.w;
```

Area Light Occlusion

- Occluded areas from multiple wedges add together



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

Penumbral Wedge Rendering

- 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

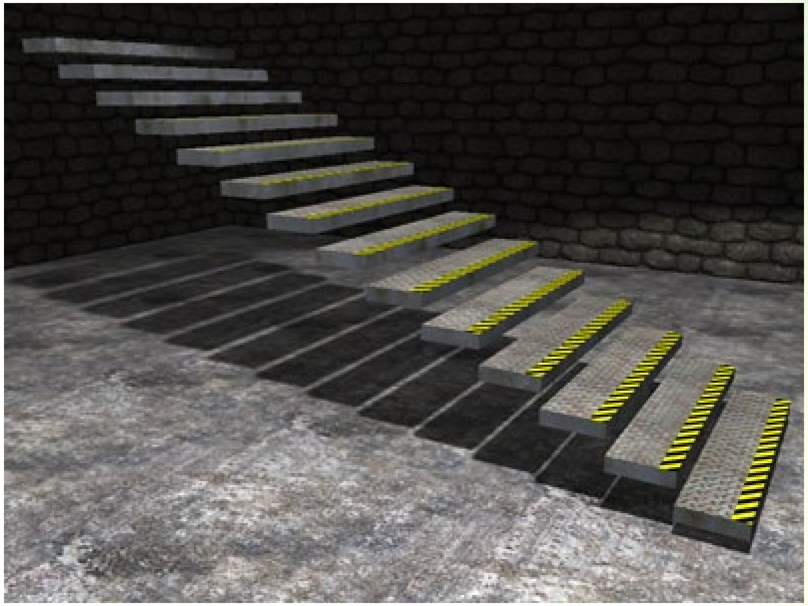
```
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
```

Penumbral Wedge Rendering

- Using a floating-point visibility buffer avoids scaling step
- Visibility values still need to be copied to alpha channel from render target

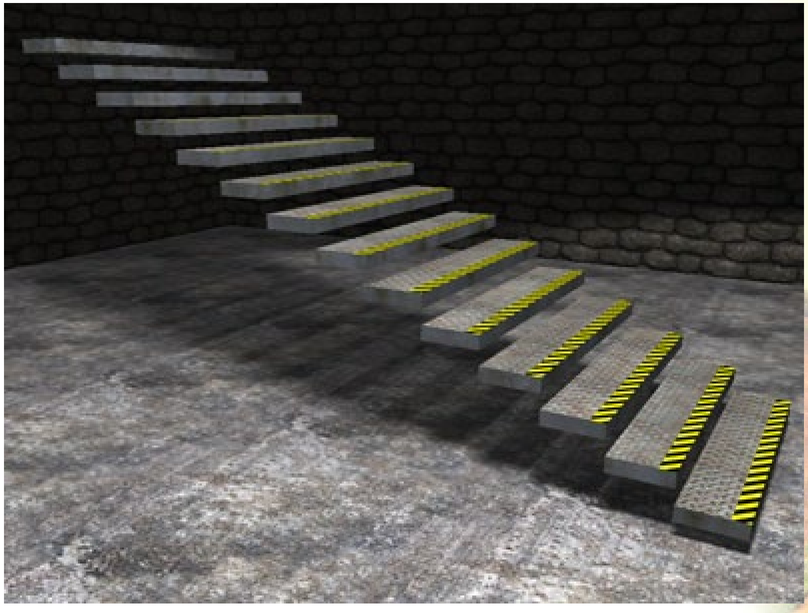
Small Light Area

Shadows sharper, rendering faster



Large Light Area

Shadows softer, interact more, rendering slower

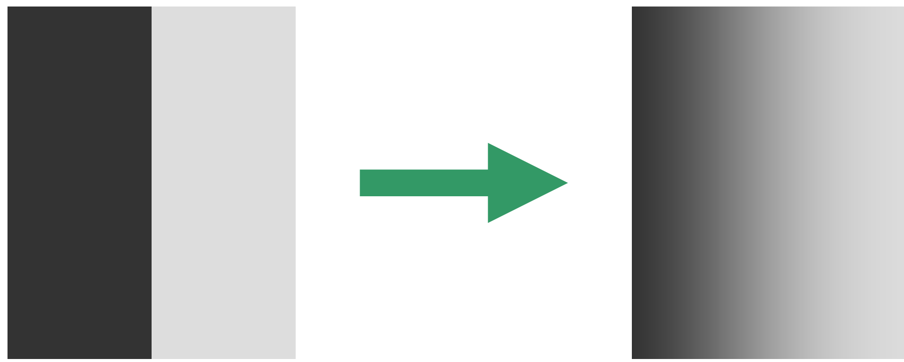


Semi-penumbral Shadows

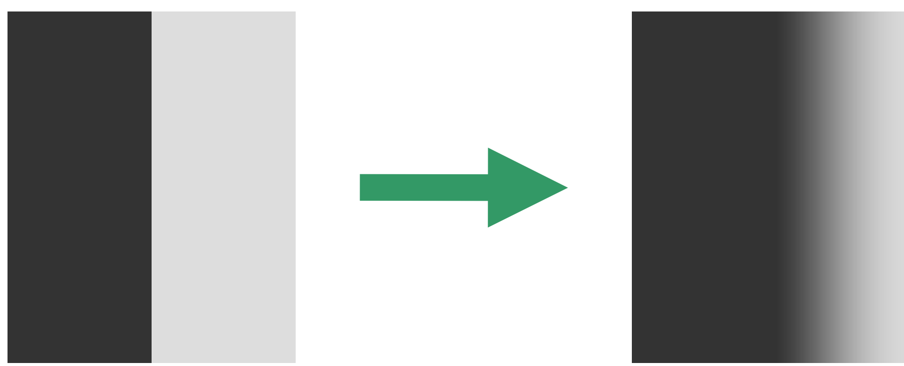
- Method for speeding up penumbral wedge soft shadows
- Only render outer half-wedges
- Less correct, but still looks good
- Lose the ability to cast shadows that have no point of 100% light occlusion

Semi-penumbral Shadows

Instead of full penumbra:



Render outer half of penumbra only:



Inner and outer half-wedges rendered

Only outer half-wedges rendered

Questions?

- lengyel@terathon.com

- Slides and source code available:

https://terathon.com/