

Pre-lighting in Resistance 2

Mark Lee

mlee@insomniacgames.com

The problem

```
for each dynamic light
    for each mesh light intersects
        render mesh with lighting
```

- $O(M*L)$
- Too much redundant texture lookups.
- Hard to optimize, we were often vertex bound.
- Each object we render needs to track lights which illuminate it.

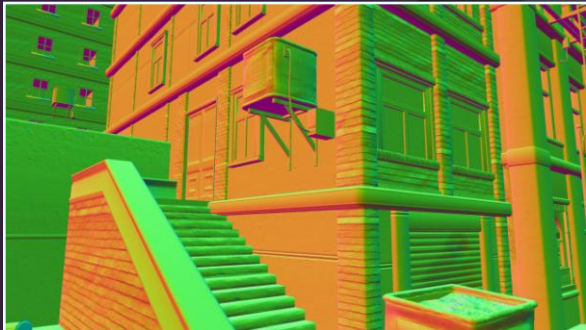
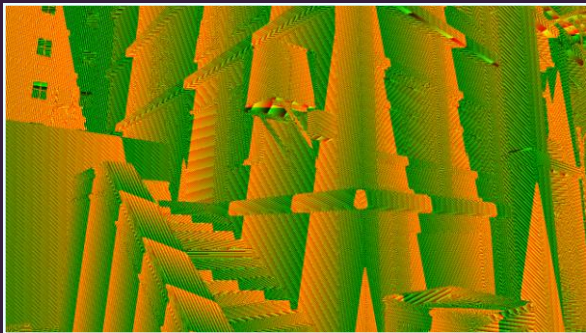
One solution

```
for each mesh  
    render mesh  
for each light  
    render light
```

- $O(M+L)$
- Lighting is decoupled from geometry complexity.

G-Buffer

- Caches inputs to lighting pass to multiple buffers (G-buffer).
- All lighting is performed in screen space.
- Nicely separates scene geometry from lighting, once geometry is written into G-Buffer, it is shadowed and lit automatically.
- G-buffer also available for post processing.



do lighting



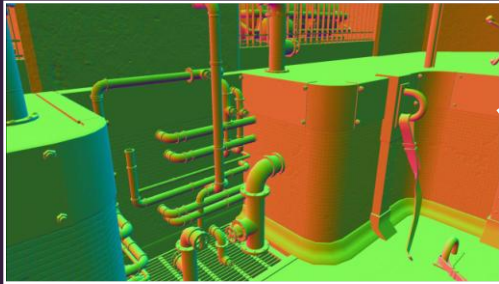
G-Buffer issues

- Prohibitive memory footprint.
 - 1280*720 MSAA buffer is 7.3mb, multiplied by 5 is 38mb
- Unproven technology on the PS3.
- A pretty drastic change to implement.

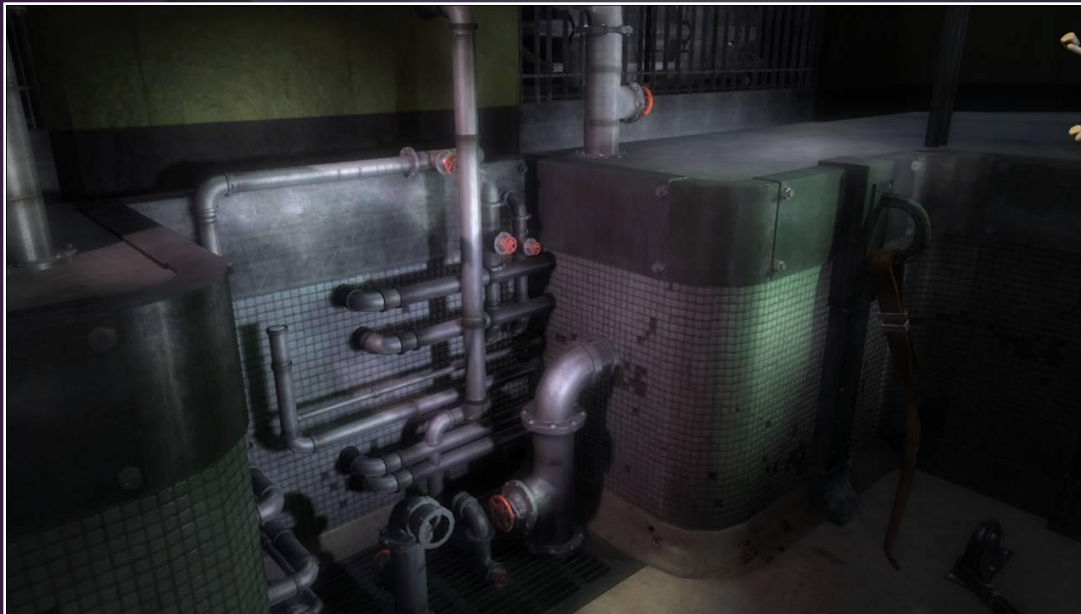
Pre-lighting / Light pre-pass

Like the G-Buffer approach except

- It only caches normals and minimal material properties in an initial geometry pass.
- A screen space pre-lighting pass is done ***before*** the main geometry pass.
- All the other material properties are supplied in a second geometry pass.



pre-lighting



render
scene



Step 1 – Depth & normals

(and some other stuff)

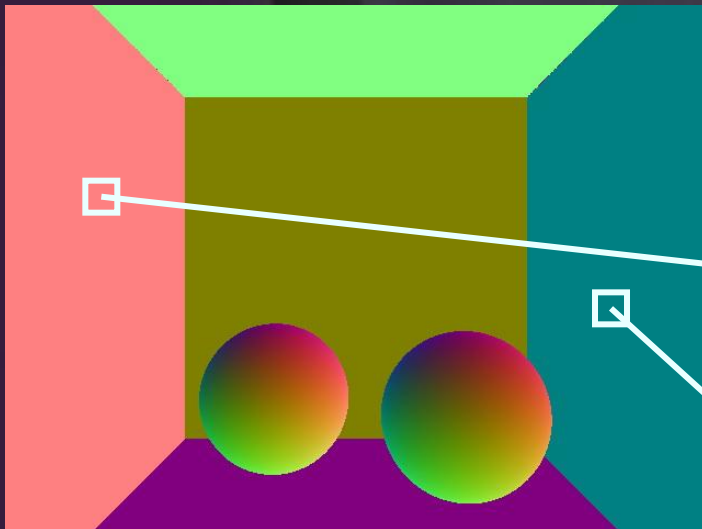
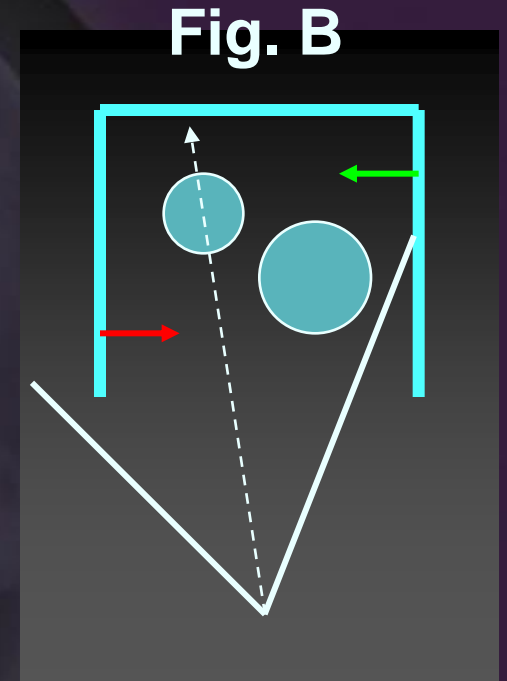
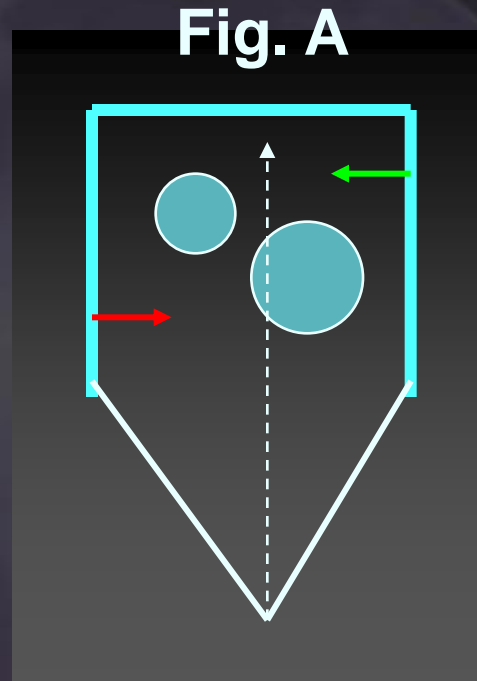
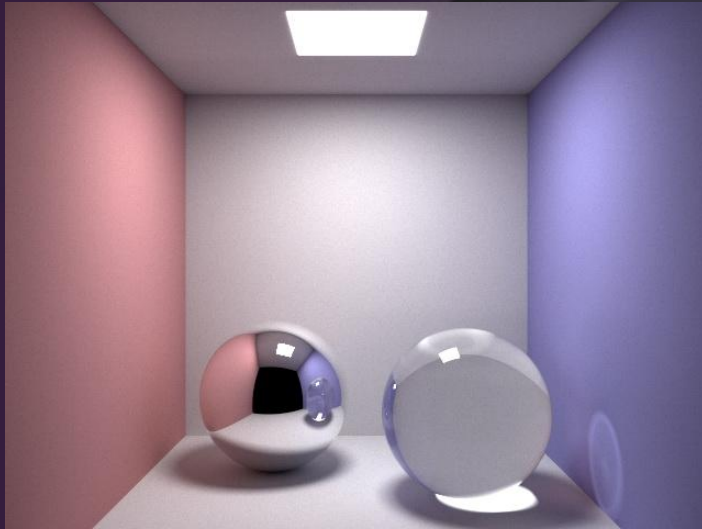
Writing depth and normals

- R2 used 2x MSAA.
- Write out normals when you are rendering your early depth pass.
- Use primary render buffer to store normals.
- Write specular power into alpha channel of normal buffer.
 - Use discard in fragment programs to achieve alpha testing.
- Normals are stored in viewspace as 3 8-bit components for simplicity.

The viewspace normal myth

- Store viewspace x and y, and reconstruct z...
 - i.e. $z = \sqrt{1 - x^2 - y^2}$
- Widespread misconception that this is valid.
- Z can go negative due to perspective projection.
- When z goes negative, errors are subtle and continuous so easy to overlook.
- We store the full xyz components for simplicity.

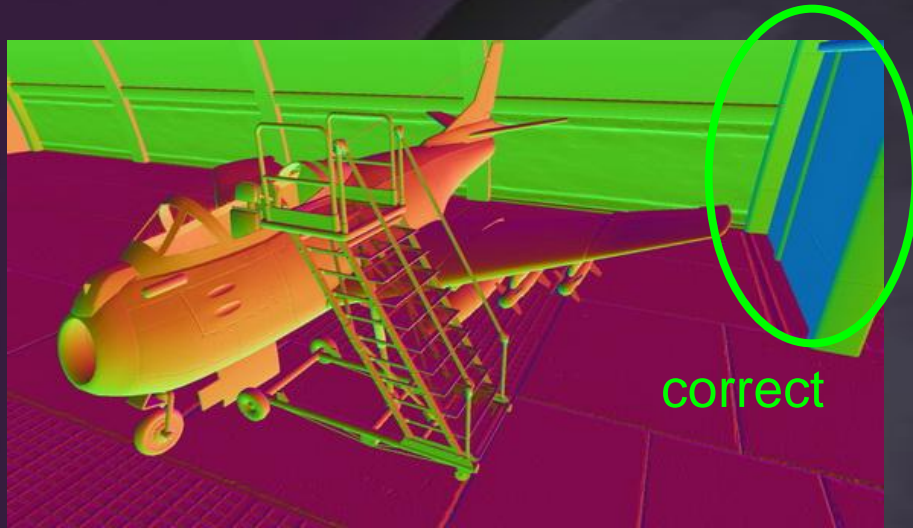
The viewspace normal myth



normal = (1, 0, 0)

normal = (-1, 0, 0)

Viewspace normal error






Step 2 – Depth resolve

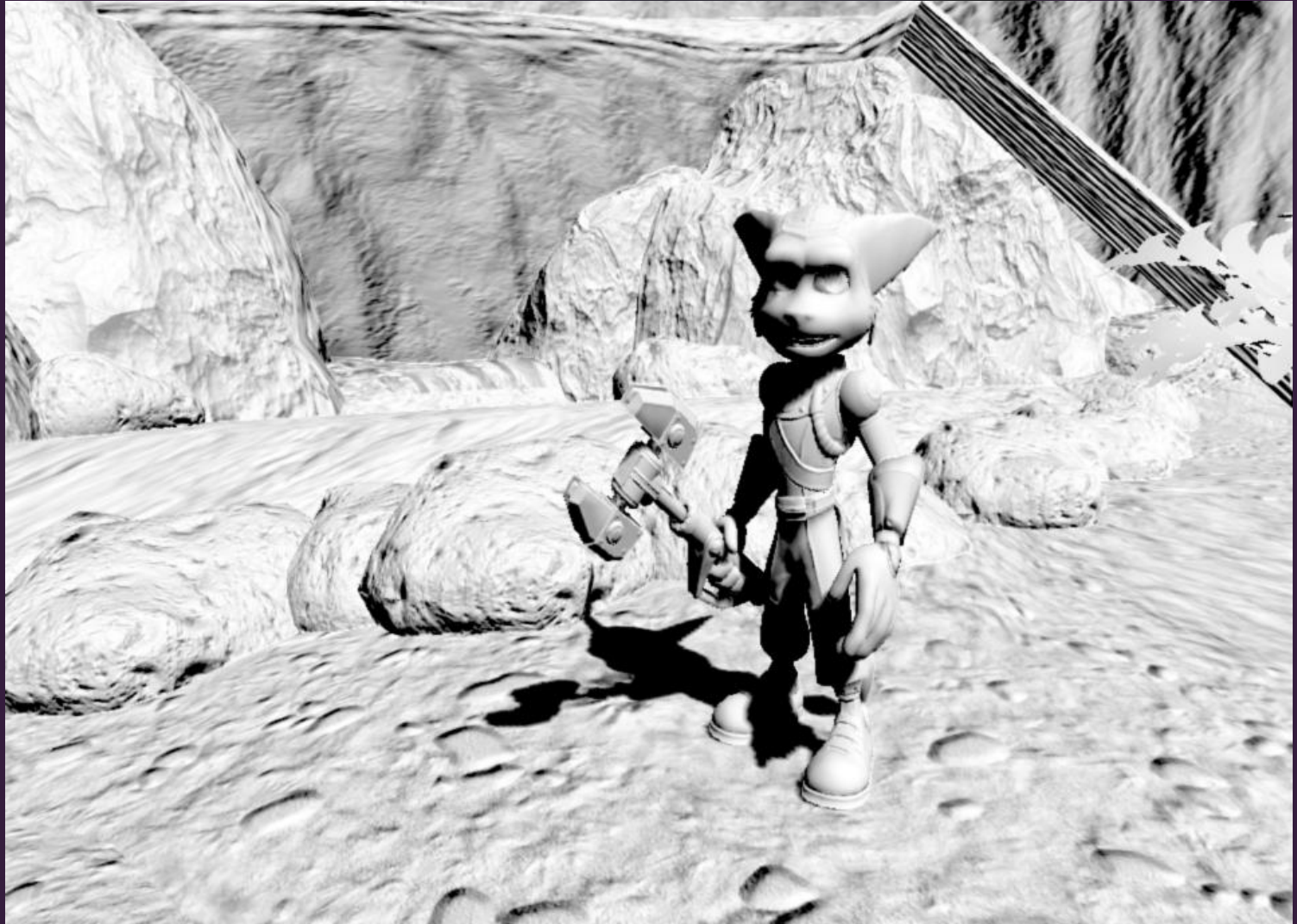
Depth resolve

- Convert MSAA to non-MSAA resolution.
- Our lighting and shadow buffers are non-MSAA, this allows us to use stencil optimizations.
- No extra work, we're just moving it earlier.
- The same final depth buffer is used for all non-MSAA rendering post color resolve.
- But what if my geometry is heavily vertex bound?



Step 3 – Accumulate sun
shadows

Sun shadows



Sun shadows

- All done in screen space.
- All sun shadows from static geometry are precomputed in lightmaps.
- We just want to accumulate sun shadows from dynamic casters.

```
for each dynamic caster
    compute OBB      // use collision rays
merge OBBs where possible
for each OBB
    render sun shadow map
for each sun shadow map
    render OBB to stencil buffer
    render shadow map to sun shadow buffer
```

Sun shadows

- Full screen backface darkening pass.
- Min blend used to accumulate.
- Use lighting buffer as temporary memory, copy to an 8-bit texture afterwards.

Which pixels to shadow?



Which pixels to shadow?




Which pixels to shadow?



Which pixels to shadow?





Step 4 – Accumulate dynamic
lights

Accumulating light

- Render all spotlight shadow maps using D16 linear depth.
- For each light:
 - Lay down stencil volumes.
 - Rendered screen space projected quad covering light.
- Stenciling vs. Tiling.
- Single buffer vs. MRT, LDR vs. HDR.

Accumulating light

- MSAA vs. non-MSAA
 - Diffuse, shadowing, projected, etc. are all done at non-MSAA resolution.
 - Specular is 2x super sampled.
- These buffers are available to all subsequent rendering passes for the rest of the frame.

Dynamic lights

$$result = C(mp, \sum_{i=1}^n P(l_i, gp))$$

where:

- l is our set of lights
- gp is the limited set of geometric/material properties we choose to store for each pixel
- mp is the full set of material properties for each pixel
- P is the function evaluated in the pre-lighting pass for each light (step 4)
- C is our combination function which evaluates the final result (step 5)

Lambertian lighting example

$$result = C(mp, \sum_{i=1}^n P(l_i, gp))$$

$$lambertian = albedo * \sum_{lights} (l_{col} * l_{att} * (normal \cdot l_{dir}))$$

- gp = normal
- P = function inside of sigma
- mp = albedo
- $C = mp * P$

Specular lighting example

$$result = C(mp, \sum_{i=1}^n P(l_i, gp))$$

$$spec = gloss * \sum_{lights} (l_{dir} \cdot Refl(v, n)^p * l_{col} * l_{att})$$

- gp = specular power
- P = function inside of sigma
- mp = gloss
- $C = mp * P$

Limitations

- Very limited range of materials can be factored in this way.
 - Extra storage for extra material properties.
 - Conditionally execute different fragment shader code paths depending on material.
- In Resistance 2, more complex material types were done with forward rendering.



Step 5 – Render scene

Rendering the scene

- Scene is rendered identically to before with the addition of the lighting and sun shadow buffer lookups.
- Smart shadow compositing function used:
 - “in shadow” global ambient level defined
 - input light level determined from baked lighting
 - geometry is only shadowed if baked light level is above this threshold

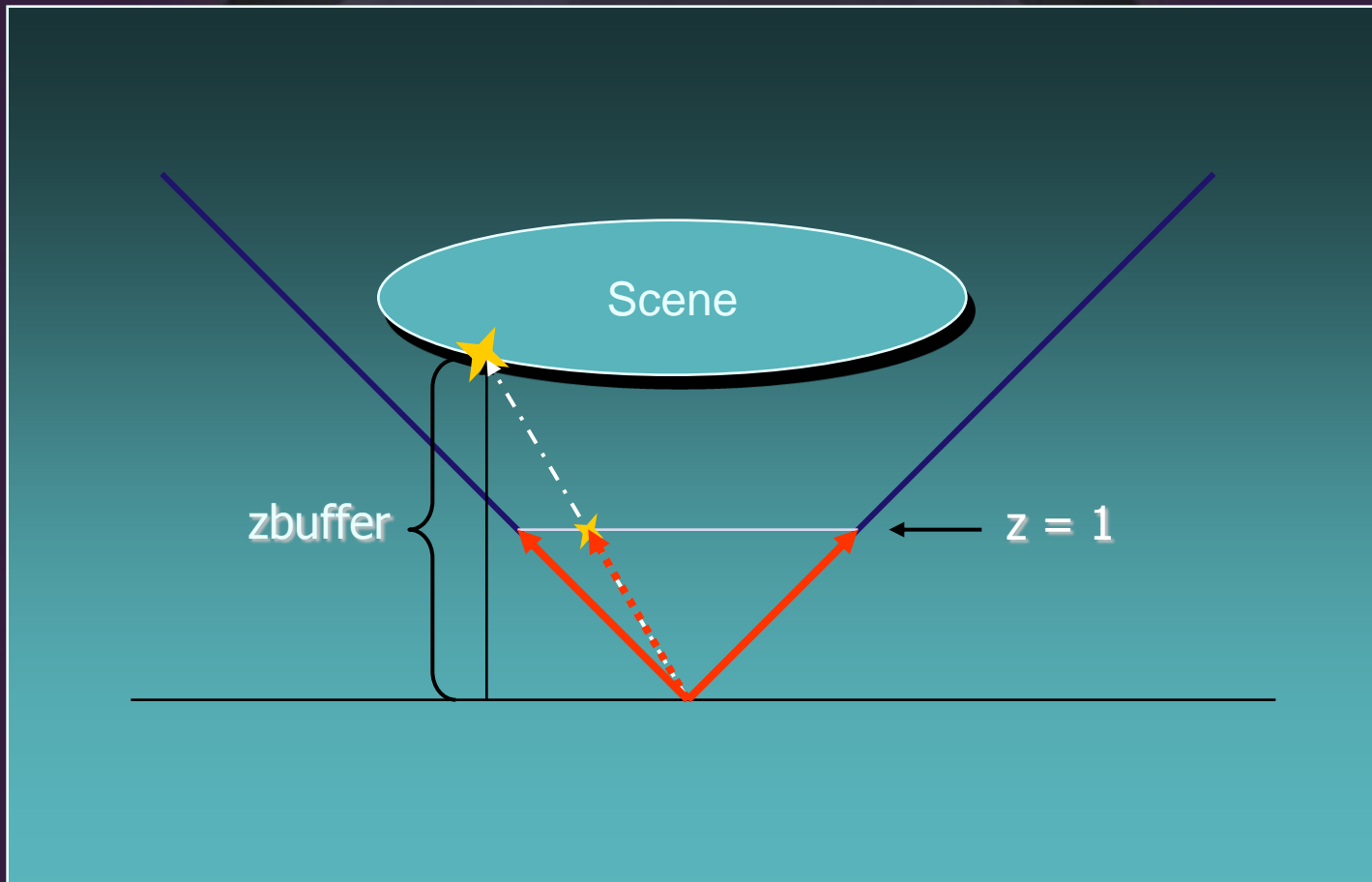


Implementation tips

Reconstructing position

- Don't store it in your G-buffer.
- Don't do a full matrix transform per pixel.
- Instead interpolate vectors from camera such that view $z = 1$.
- Technique not confined to viewspace.

Reconstructing position



Reconstructing depth

- W-buffering isn't supported on the PS3.
 - Linear shadow map tricks don't work.
- Z-buffer review (D3D conventions):
 - The z value is 0 at near clip and far at the far clip.
 - The w value is 0 at viewer and far at the far clip.
 - z/w is in the range 0 to 1.
 - This is scaled by 2^{16} or 2^{24} depending on our depth buffer bit depth.

Reconstructing depth

$$z = f(vz - n) / (f - n)$$

$$w = vz$$

$$zw = z / w$$

$$zb = zw * (2^d - 1)$$

where:

f = far clip

n = near clip

vz = view z

zb = what is stored
in the z-buffer

d = bit depth of
the z buffer



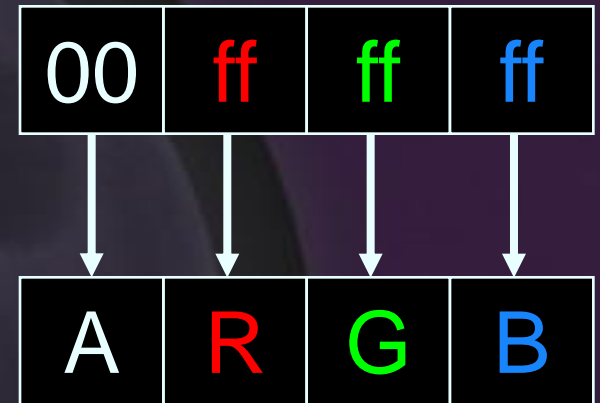
Recovering hyperbolic depth

- Alias an argb8 texture to the depth buffer.
- Using a 24-bit integer depth buffer, 0 maps to near clip and 0x00ffffff maps to far clip.
- In C we would do

```
float(r<<16 + g<<8 + b) / range
```

- When dealing with floats it becomes

```
(r * 65536.f + g * 256.f + b) / range
```



Recovering hyperbolic depth

- But....
- Texture inputs come in at a (0, 1) range
- The texture integer to float conversion isn't operating at full float precision
 - We are magnifying error in the red and green channels significantly
- Solution: round each component back to 0-255 integer boundaries

```
float3 rgb = f3tex2D(g_depth_map, uv).rgb;  
rgb = round(rgb * 255.0);  
float zw = dot(rgb, float3( 65536.0, 256.0, 1.0 ));  
zw *= 1.0 / 16777215.0;
```

Recovering linear depth

- Recall that:

$$z = f(vz - n) / (f - n)$$

$$w = vz$$

$$zw = (f(vz - n) / (f - n)) / vz$$

- Solving for vz , we get:

$$vz = 1.0 / (zw * a + b)$$

where:

$$a = (f - n) / -fn$$

$$b = 1.0 / n$$

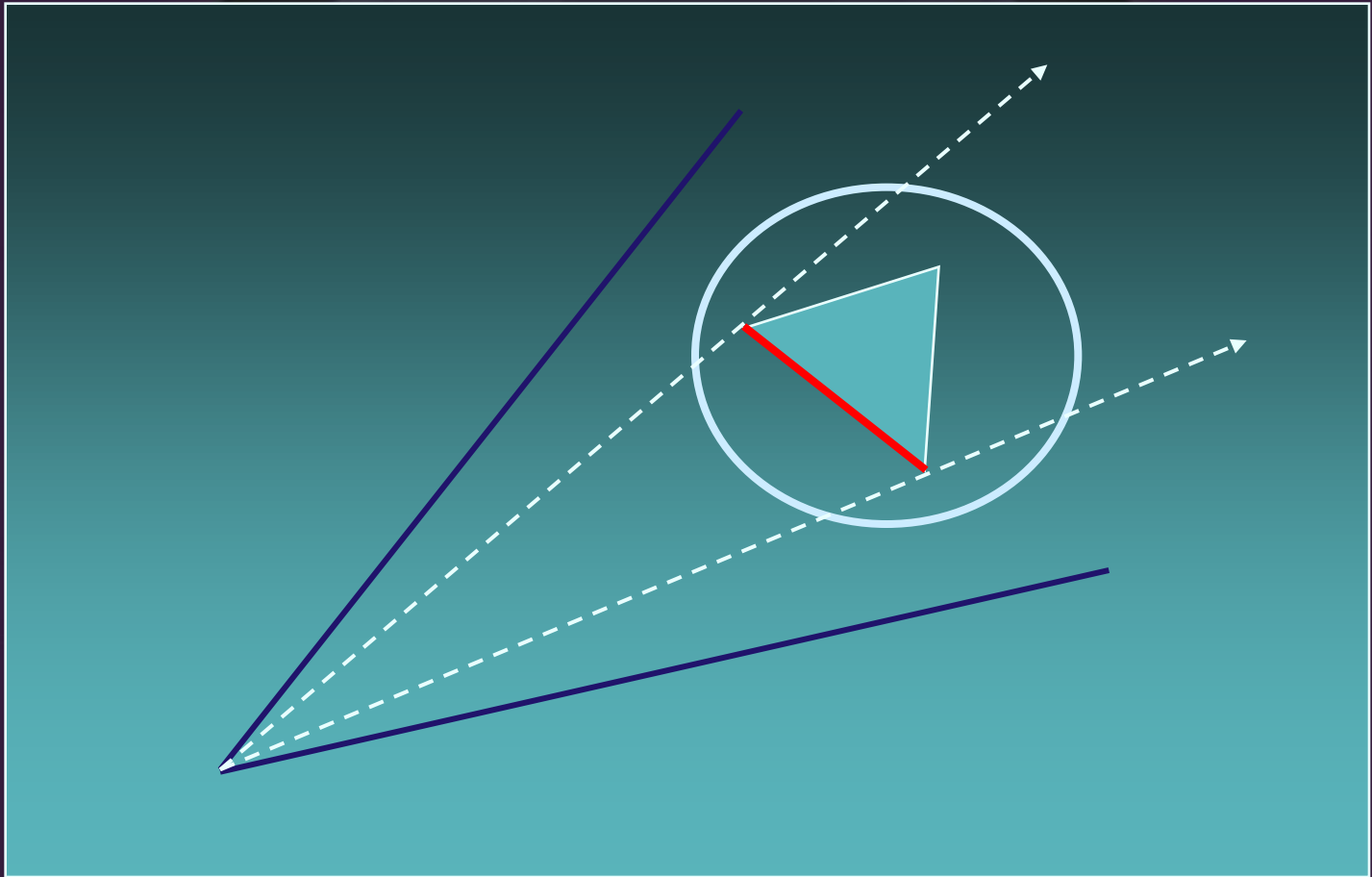
- Reconstructed linear depth precision will be very skewed – but it's skewed to where we want it.

Stenciling algorithm

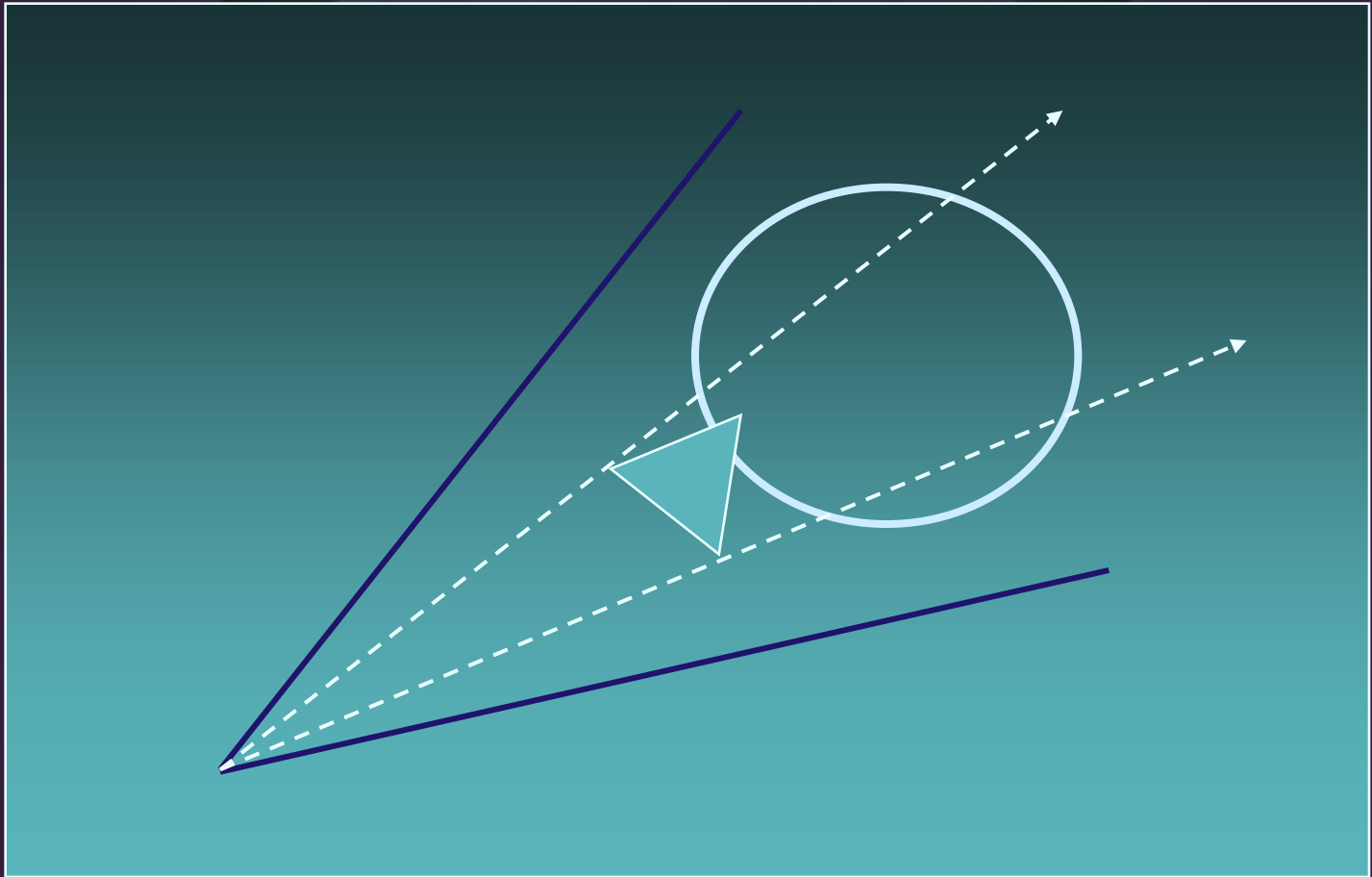
```
clear stencil buffer
if( front facing and depth test passes )
    increment stencil
if( back facing and depth test passes )
    decrement stencil
render light only to pixels which have
non-zero stencil
```

- Stencil shadow hardware does this.
- Set culling and depth write to false when rendering volume.
- Same issues as stencil shadows
 - Make sure light volumes are closed.
 - This only works if camera is outside light volumes.

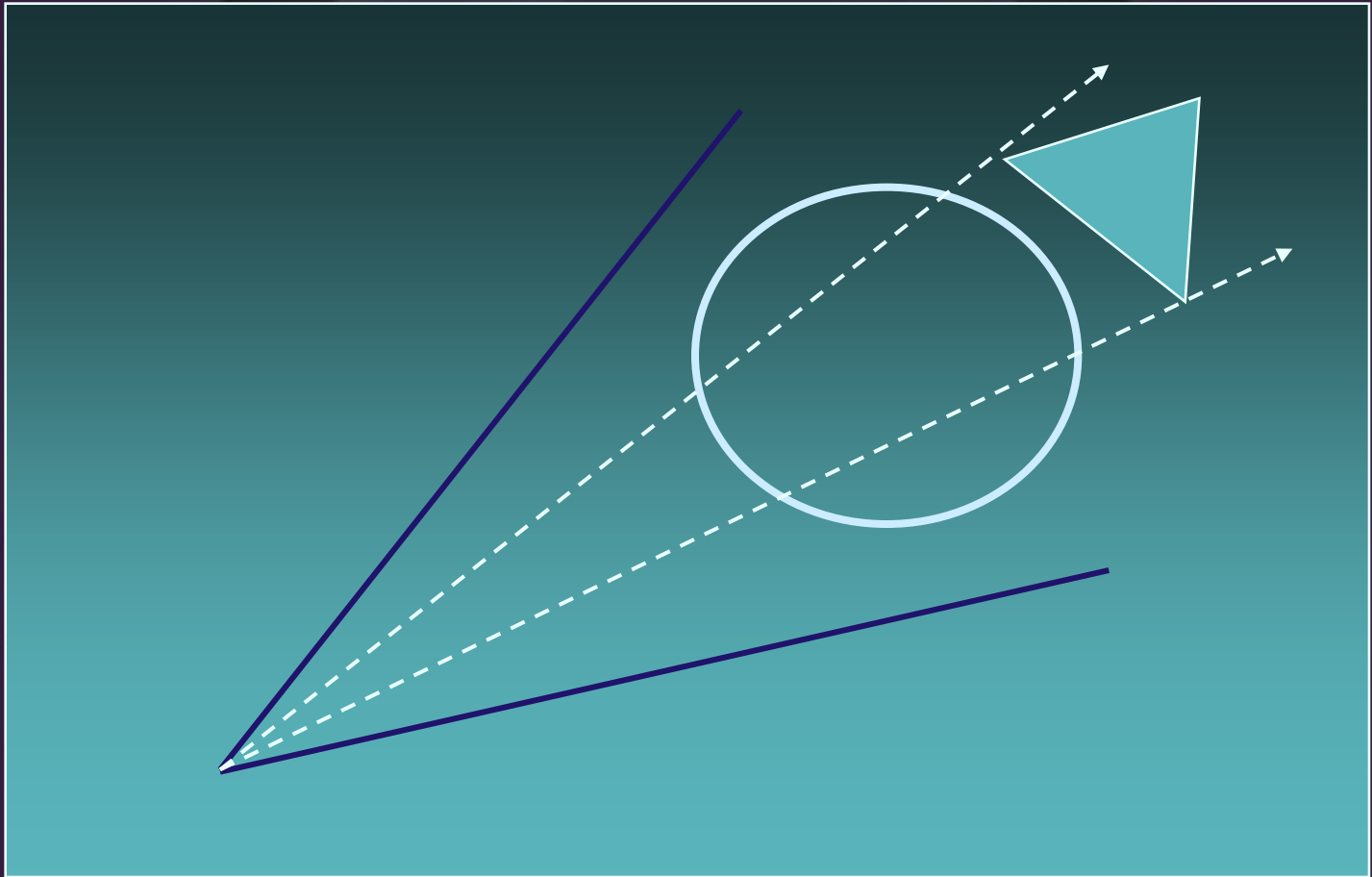
Object is inside light volume



Object is near side of light volume



Object is far side of light volume

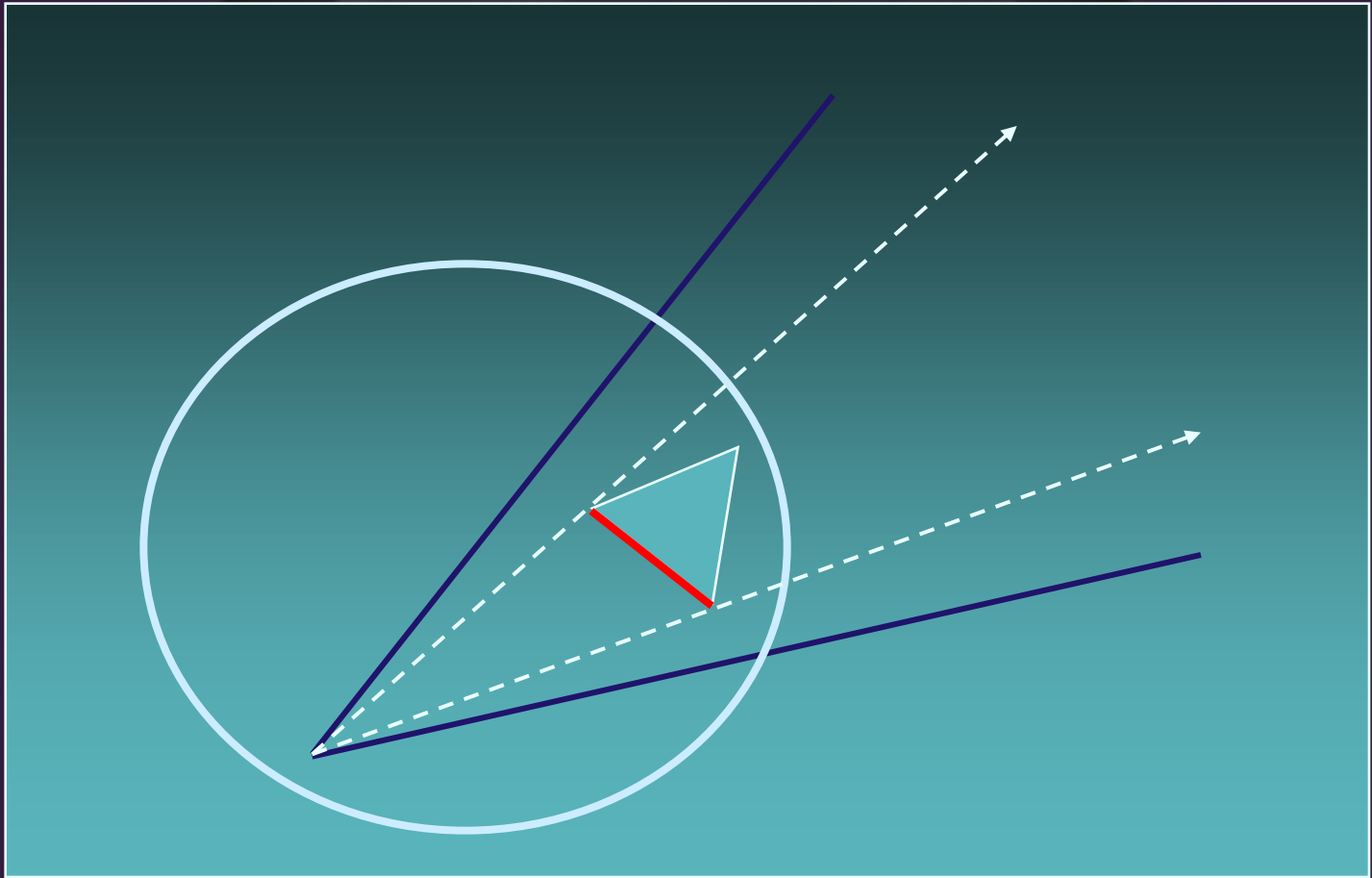


If the camera goes inside light volume

```
clear stencil buffer
if( front facing and depth test fails )
    wrap increment stencil
if( back facing and depth test fails )
    wrap decrement stencil
render light only to pixels which have
non-zero stencil
```

- Switch to depth fail stencil test.
- Only do this when we have to, this disables z-cull optimizations.
 - Typically we'll need some fudge factor here.
- Stenciling is skipped for smaller lights.

If the camera goes inside light volume



Pros and cons

G-Buffer

- Requires only a single geometry pass. Good for vertex bound games.
- More complex materials can be implemented.
- Not all buffers need to be updated with matching data, e.g. decal tricks.

Pre-lighting / Light pre-pass

- Easier to retrofit into "traditional" rendering pipelines. Can keep all your current shaders.
- Lower memory and bandwidth usage.
- Can reuse your primary shaders for forward rendering of alpha.

Problems common to both approaches

- Alpha blending is problematic.
 - MSAA and alpha to coverage can help.
- Encoding different material types is not elegant.
 - Coherent fragment program dynamic branching can help.

See also

- Deferred Rendering in Killzone 2 - Guerrilla Games
http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf
- The Light Pre-Pass Renderer – Wolfgang Engel
<http://www.wolfgang-engel.info/RendererDesign.zip>

Questions?



mlee@insomniacgames.com