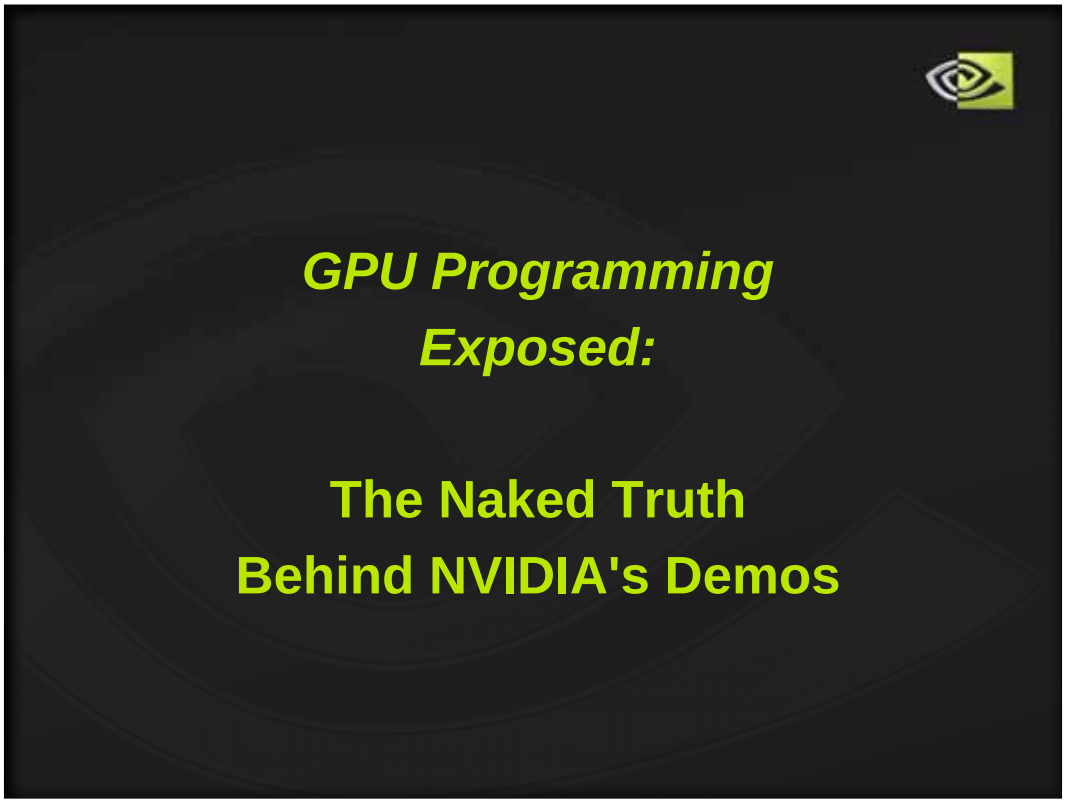

GPU Programming
Exposed:
The Naked Truth
Behind NVIDIA's Demos

Luna Demo

- Translucency
- Displacement Maps
- Ray Traced Eye
- Luna Suit Shader

Motivation for Translucency

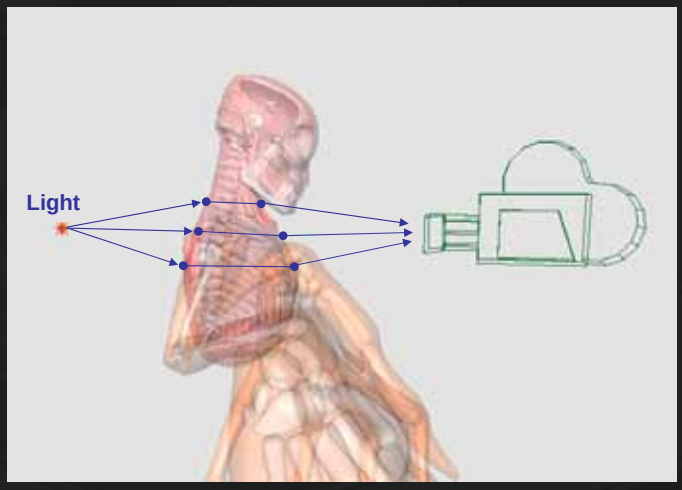
- To render more fleshy, organic objects
- Art direction called for extremely bright lights

- Translucency is cool ☺

Basic Idea

- Assume character is between a bright light & camera
- Determine surface thickness from camera pov
- Based on thickness, do a dependent texture lookup to find the color of the skin
- Calculate facing ratio of the object & camera to light
- Based on the facing ratio, combine normal and translucent colors

Try to approximate amount of light going through the surface



Computing Thickness

- Render back faces into fp16 buffer writing pixel position
- Render front faces
  - Fetch the back face position for the given pixel using pixel's screen position
  - Find the distance between front and back faces
  - Normalize the distance into [0,1] so we can use it as a texture coordinate later
- But…

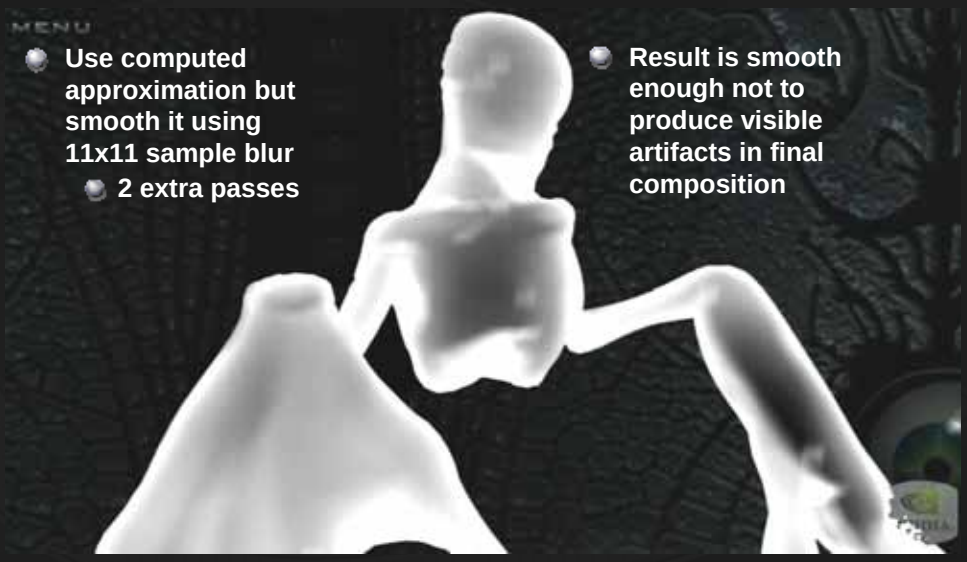
Computing Thickness

- What we find is not true thickness
- Results are jaggy and edgy
- Cannot use it as is

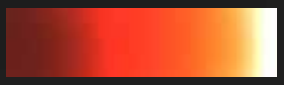
- Alternate method: Render all front faces additively, then all the back faces with subtractive blending

Computing Thickness

- Use computed approximation but smooth it using 11x11 sample blur
  - 2 extra passes
- Result is smooth enough not to produce visible artifacts in final composition



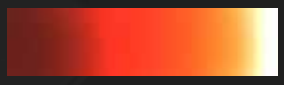
Translucent Skin Color



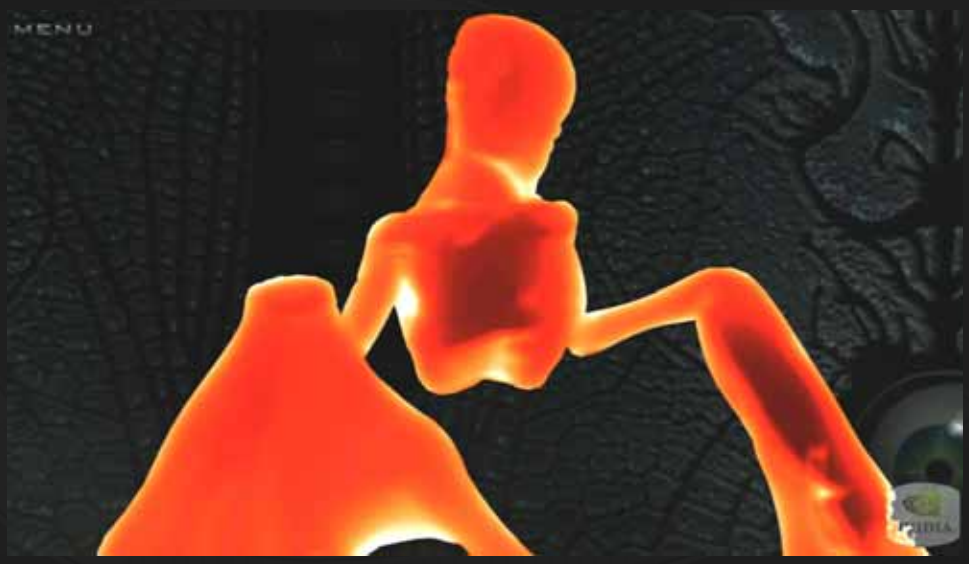
- Given thickness of the surface, find the color of the skin when the light travels through it
- Use the normalized thickness as a texture coordinate for the following texture
  - Allows for good control over how each of the densities of the surface looks and how quickly they change
  - Thicker parts end up dark red, membranes end up faint orange

Other Sources

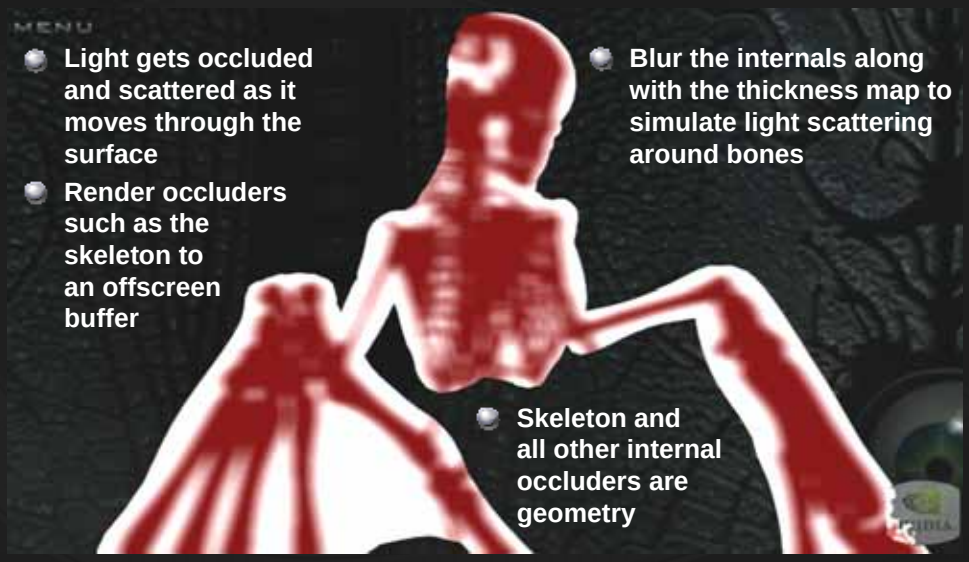
- GPU Gems Article “Real-Time Approximations to Subsurface Scattering” by Simon Green
- More samples of using texture lookup tricks to achieve interesting lighting effect in the NVIDIA Developer SDK (anisotropic lighting, etc.) http://developer.nvidia.com/object/sdk_home.html



Layer in final composition

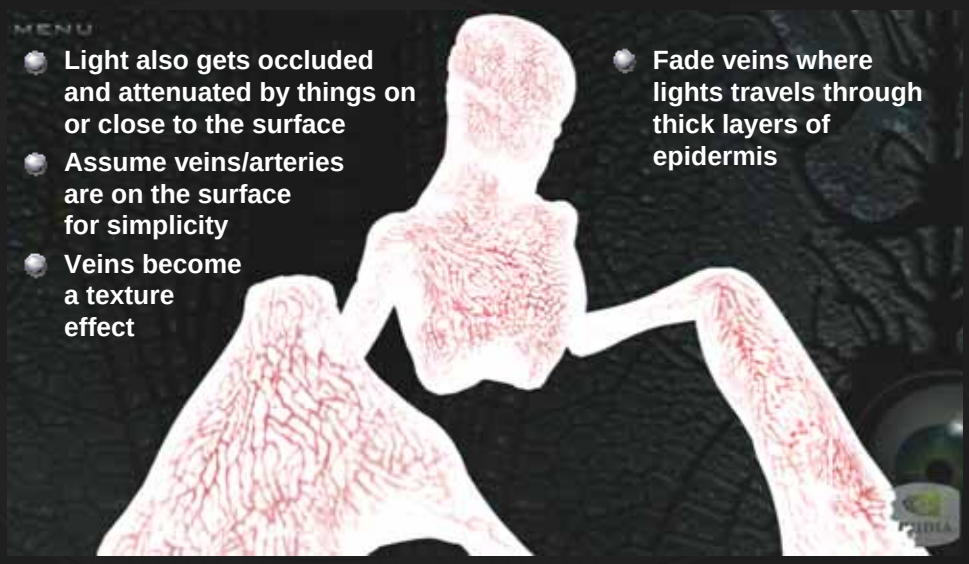


Internal Occluders



- Light gets occluded and scattered as it moves through the surface
- Render occluders such as the skeleton to an offscreen buffer
- Blur the internals along with the thickness map to simulate light scattering around bones
- Skeleton and all other internal occluders are geometry

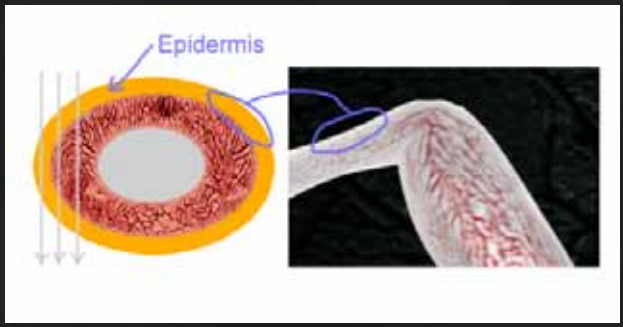
Surface Occluders



- Light also gets occluded and attenuated by things on or close to the surface
- Assume veins/arteries are on the surface for simplicity
- Veins become a texture effect
- Fade veins where lights travels through thick layers of epidermis

Vein visibility

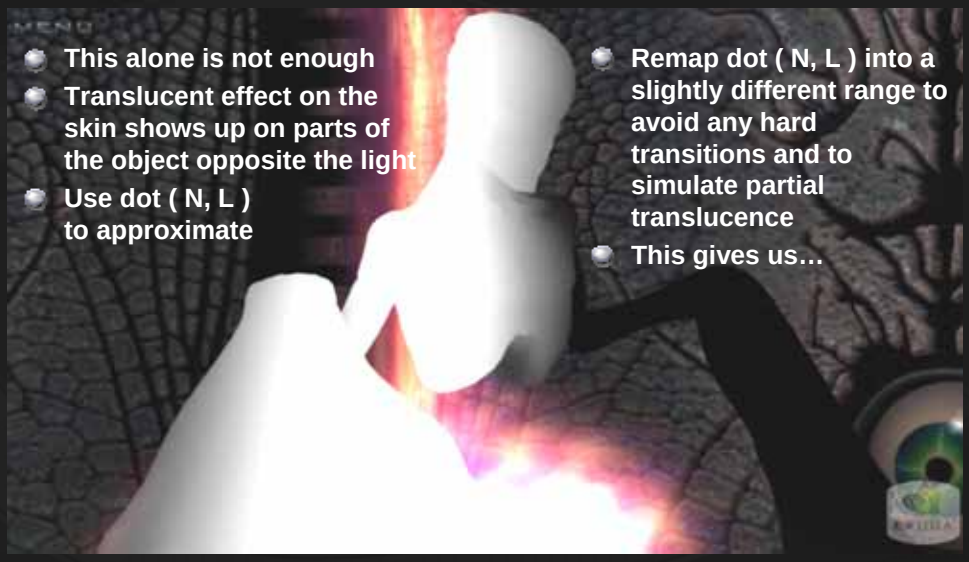
- When light travels through thick layers of epidermis, veins are not as visible
- Change the vein layer accordingly



Areas of Influence

- Need to find translucent areas based on light, object and camera position
  - Pow(dot(V, L), n) is a reasonable approximation
  - V – vector from a fragment to the camera
  - L – vector from the light to a fragment
  - n – some power to make it look good
- In case of the vertical plasma beam, use horizontal view plane and plasma light line intersection to approximate L
- This gives us…

Areas of Influence

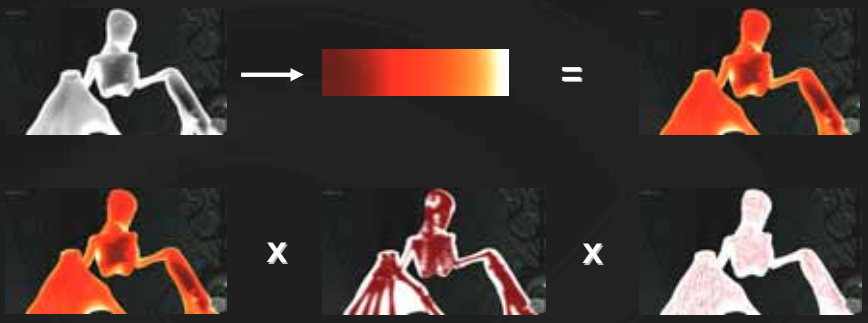


- This alone is not enough
- Translucent effect on the skin shows up on parts of the object opposite the light
- Use dot (N, L) to approximate
- Remap dot (N, L) into a slightly different range to avoid any hard transitions and to simulate partial translucence
- This gives us…

Combine layer with previous for more interesting area of influence

Interpolant between combined translucent color and base shaded color

Compositing Final Translucent Color



- This gives us…

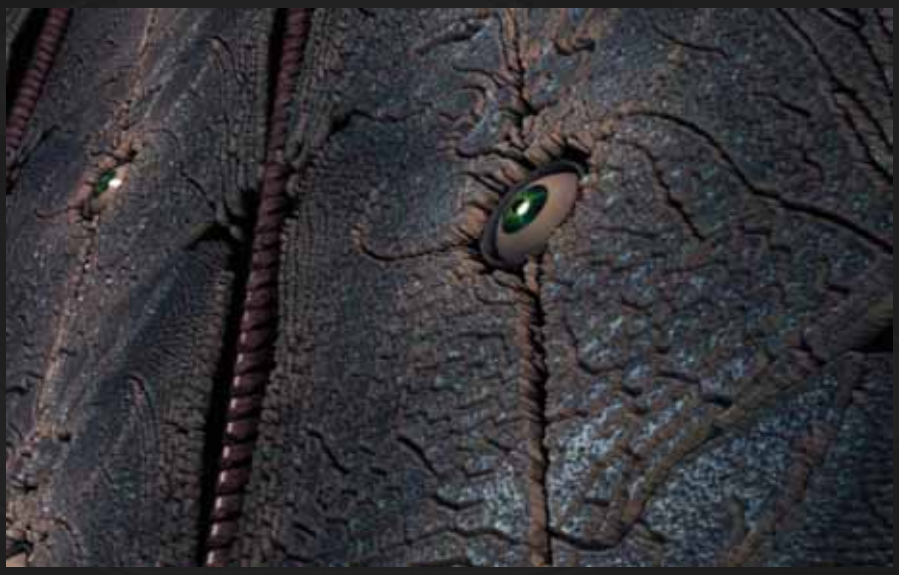
Add bloom to bright areas to diffuse light

Interpolate with the base shaded image

Base Shaded Image, before translucency

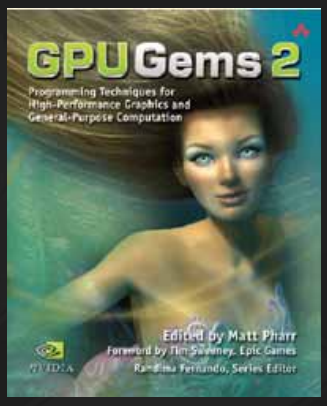
Final Image

Displacement Mapping with Occlusion



Efficient Computation of Distance Maps

- See Section 8.4
- Source/tools on GPU Gems 2 CD
- Danielsson, Per-Erik. 1980. "Euclidean Distance Mapping." *Computer Graphics and Image Processing* 14, pp. 227–248.

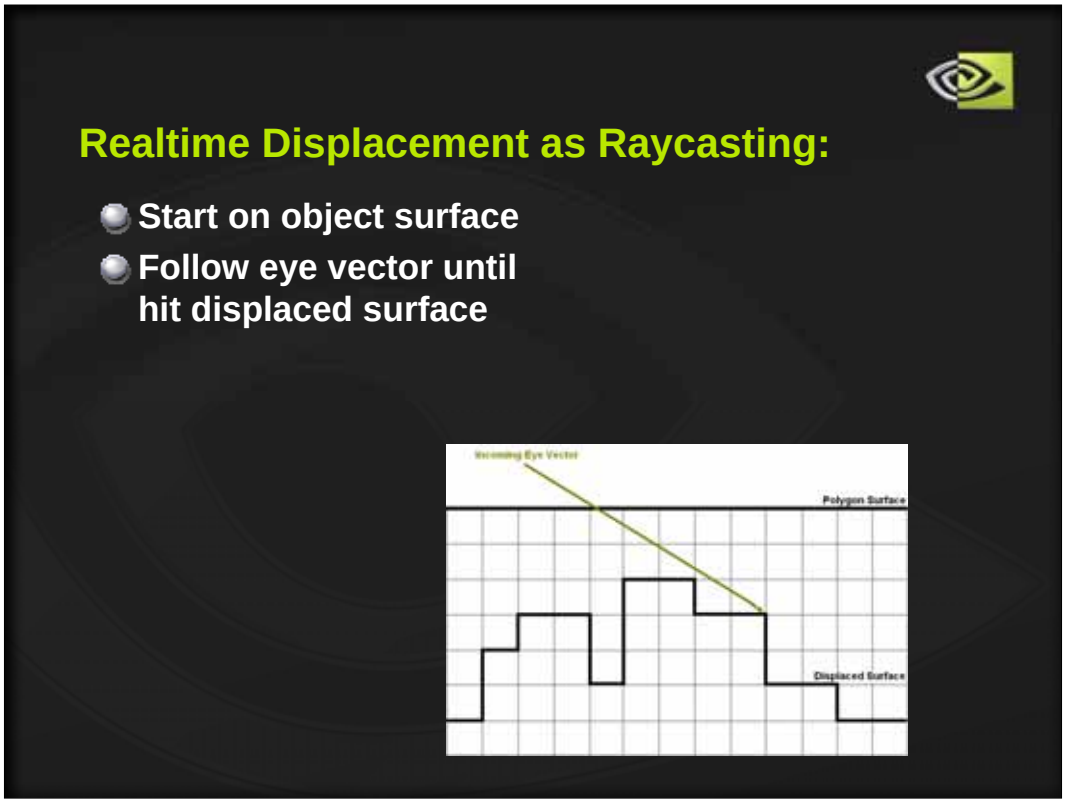


Parallax Mapping – In games now

- Parallax mapping
  - T. Kaneko et al. “Detailed Shape Representation with Parallax Mapping.” In *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, Tokyo, Dec. 2001.
- Valid for smoothly varying height fields
  - No occlusion
  - No large displacements
  - No high frequency features

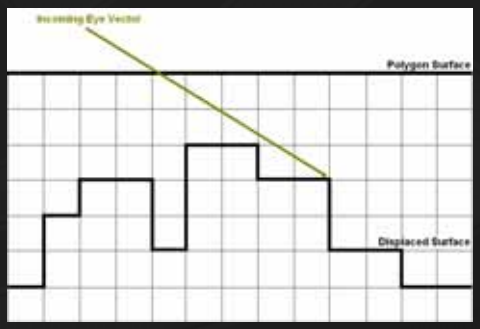
Displacement Mapping with Occlusion

- Resolves self-occlusion
- Better for more uneven surfaces
  - Carved walls with deep relief
  - Brick / stone
  - Grate


Realtime Displacement as Raycasting:
Start on object surface
Follow eye vector until hit displaced surface
Incoming Eye Vector
Polygon Surface
Displaced Surface

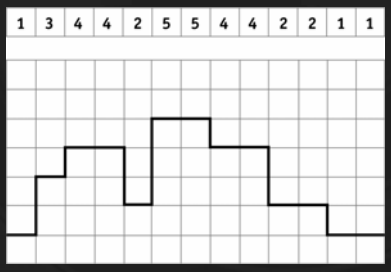
Basic idea: Marching through 3D Texture

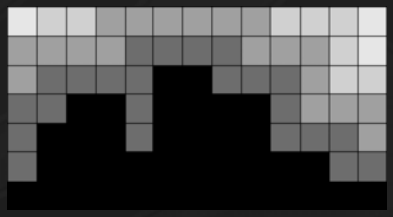
- Create Volumetric texture
  - '1' in empty voxels, '0' in voxels on or in surface
- Fragment shader
  - TanEyeVec, TexCoordIter (U, V, 1.0)
  - Iteratively increment TexCoordIter by scaled TanEyeVec



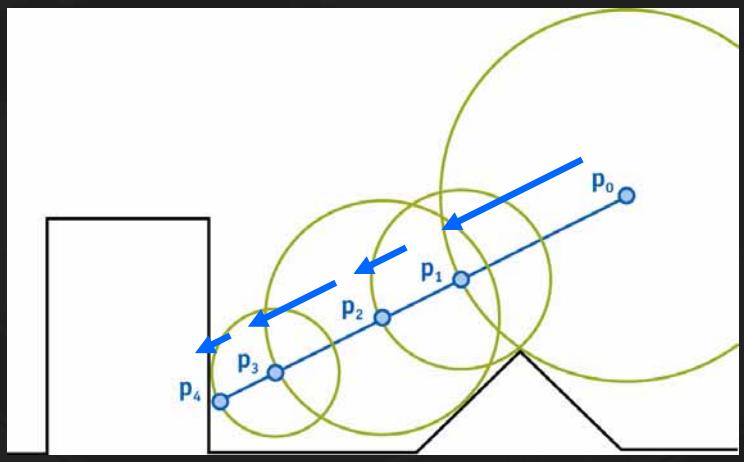
Basic idea: Can we do better?

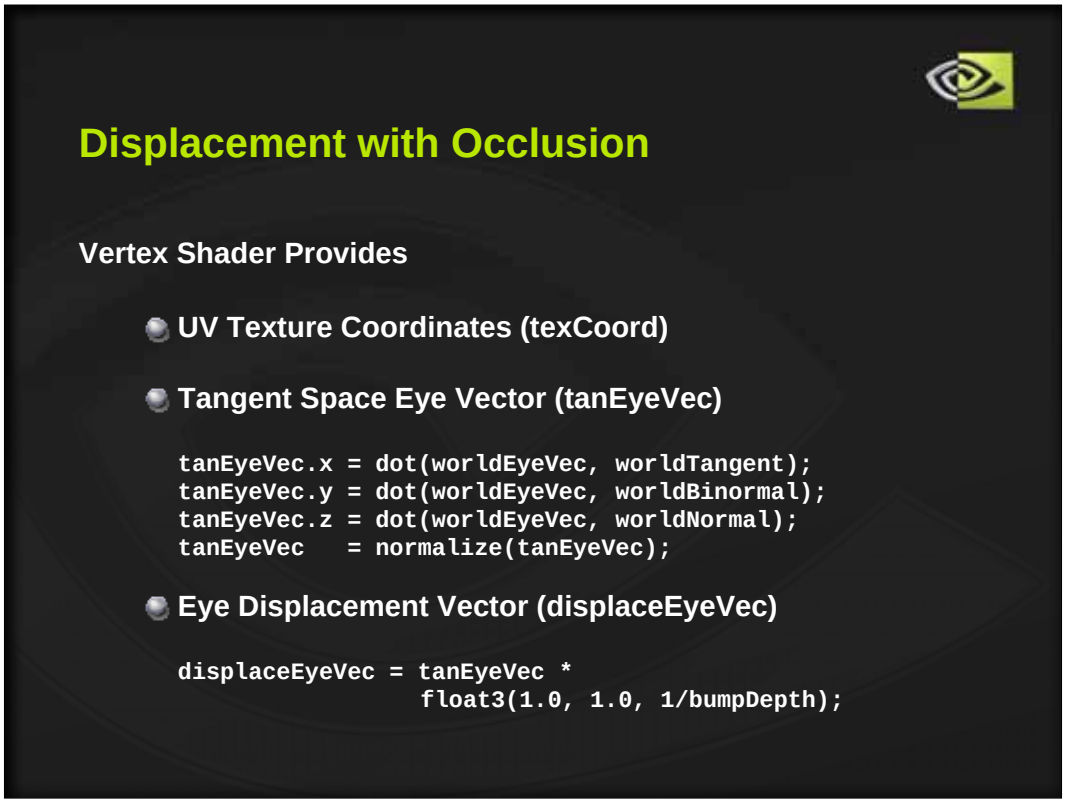
- Steps too big: will miss features
- Steps too small: will waste performance
- What if we store the distance to nearest surface?

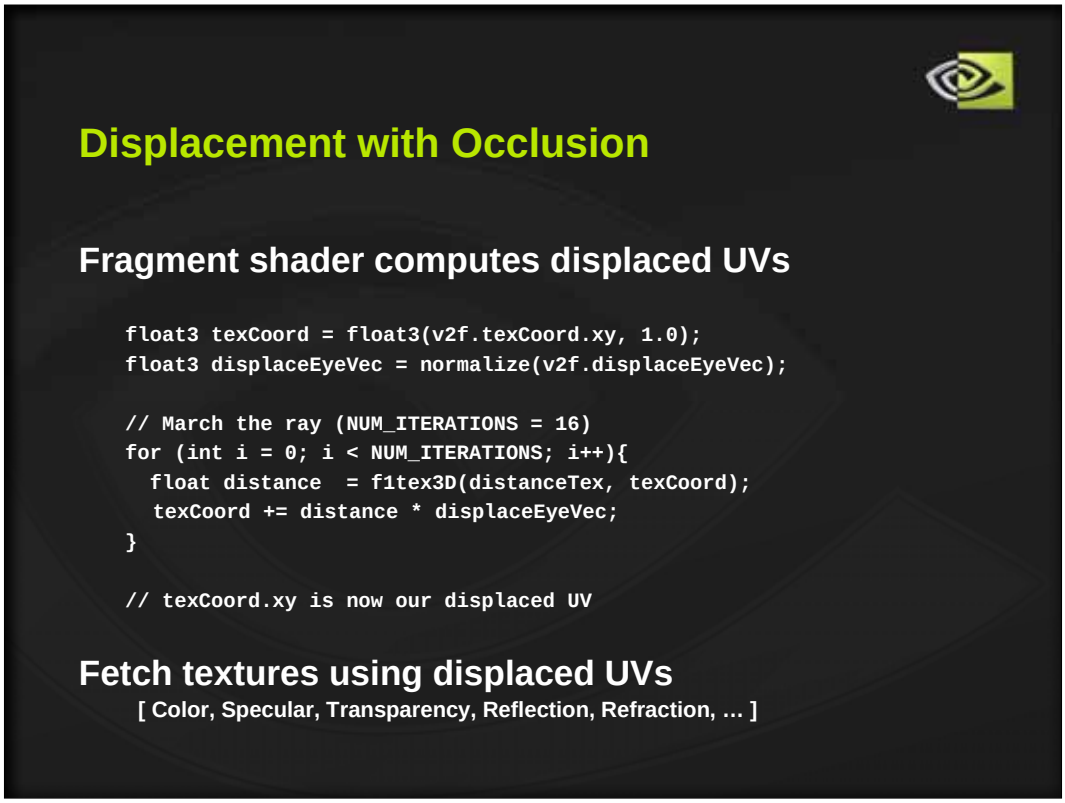




Nonuniform steps: Sphere Tracing




Displacement with Occlusion
Vertex Shader Provides
UV Texture Coordinates (texCoord)
Tangent Space Eye Vector (tanEyeVec)
tanEyeVec.x = dot(worldEyeVec, worldTangent);
tanEyeVec.y = dot(worldEyeVec, worldBinormal);
tanEyeVec.z = dot(worldEyeVec, worldNormal);
tanEyeVec = normalize(tanEyeVec);
Eye Displacement Vector (displaceEyeVec)
displaceEyeVec = tanEyeVec *
float3(1.0, 1.0, 1/bumpDepth);


Displacement with Occlusion
Fragment shader computes displaced UVs
float3 texCoord = float3(v2f.texCoord.xy, 1.0);
float3 displaceEyeVec = normalize(v2f.displaceEyeVec);
// March the ray (NUM_ITERATIONS = 16)
for (int i = 0; i < NUM_ITERATIONS; i++){
float distance = f1tex3D(distanceTex, texCoord);
texCoord += distance * displaceEyeVec;
}
// texCoord.xy is now our displaced UV
Fetch textures using displaced UVs
[Color, Specular, Transparency, Reflection, Refraction, …]

Performance

- Each iteration is {tex; mad;}
  - Single cycle on GeForce FX, 6 Series, 7 Series
- Number of iterations depends on
  - Volume texture resolution
  - Smoothness of data
  - 16 iterations plenty for our tests
- Performance with simple lighting:
  - 90M pixels/s on GeForce 6800 GT
  - 180M pixels/s on GeForce 7800 GTX
- Simple parallax mapping still faster
  - Use when surface or angles prevent occlusion

Issue 1: Texture Stretching

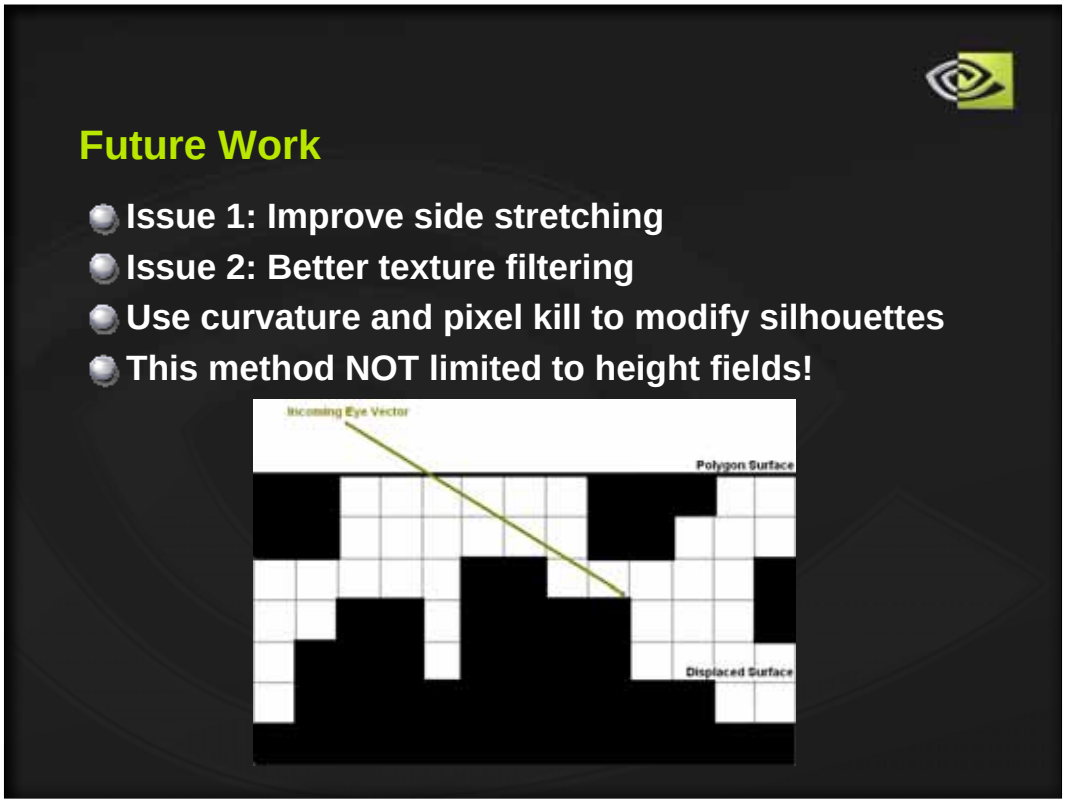
UVs were applied facing the normal

3D material or noise?

Issue 2: Texture Filtering

Displaced shiny bits can twinkle

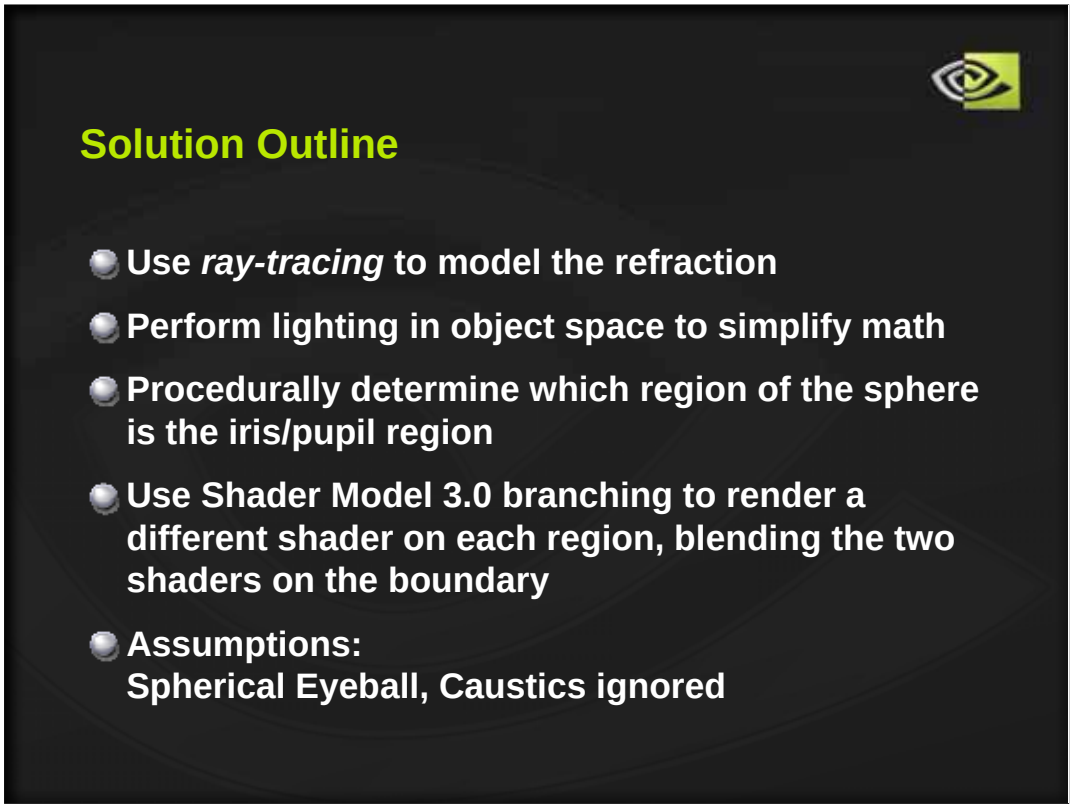
- Mipmap bias? Multiple texture samples?
- Intelligent post-render blurring?


Future Work
Issue 1: Improve side stretching
Issue 2: Better texture filtering
Use curvature and pixel kill to modify silhouettes
This method NOT limited to height fields!
Incoming Eye Vector
Polygon Surface
Displaced Surface

A practical application of real-time ray-tracing on today's GPUs

The Challenge

- Create a realistic 3D eyeball
  - At times during the demo the eye will be full-screen
- Problems to solve:
  - Refraction of light through the cornea
  - Wet and shiny eyeball surface
  - Transparency: light might pass through the side of the cornea and hit something else in the scene



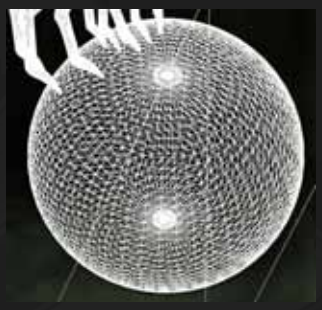
Computing lighting in object space is key for the ray-tracing calculations: the ray-sphere intersection code becomes very simpliy when the sphere is centered at the origin. And normals are computed easily by normalizing positions.

Dealing with ray-traced geometry

- Drawing procedural iris geometry using ray-tracing requires some new tricks:
  - New shadow coordinates must be computed for correct shadow mapped shadows
- The boundary between the ray-traced geometry and the white of the eyeball is a discontinuity
  - Can we achieve a smooth transition on the edge?

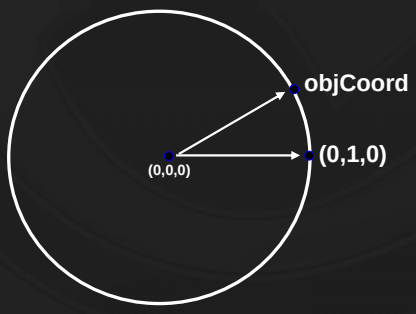
Start with a smooth polygon sphere

- A simple polygon sphere (50x50) was used, which ignores the fact that the eye actually bulges.
- No other geometry is created using triangles.
- The iris is defined procedurally by ray-tracing.



Procedurally compute the Iris Region

- The vertex shader passes in the object space coordinates
- A dot product in the fragment shader is used to determine the perfectly smooth edge of the iris



dot(objCoord, (0, 1, 0)) > 0.805?

We found full 32 bit precision ray-tracing math is required for precision reasons. So avoid using half floats until the lighting operations are computed.

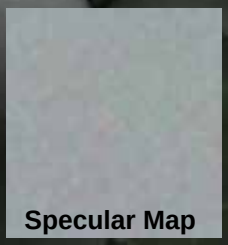
(0,1,0) is an arbitrary decision for the eye to face the positive y direction.

0.805 is the cos of the angle between (0,1,0) and any vector to a point on the iris edge (change this 0.805 value to make the iris region smaller or bigger)

The white of the eye

- 1 spotlight and 2 point lights + ambient
- 16 tap uniform 4x4 spread shadow lookup for the spotlight above
- Subtle bump/vein map
- Soft-wrap diffuse…


Diffuse Map


Specular Map

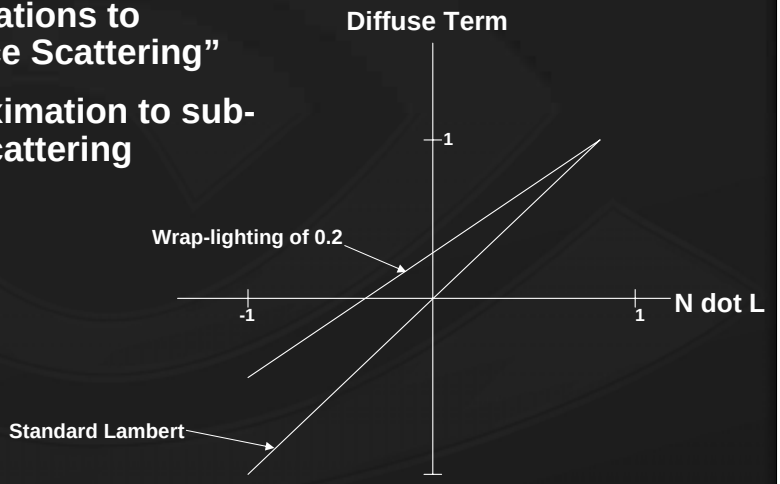

Bump Map

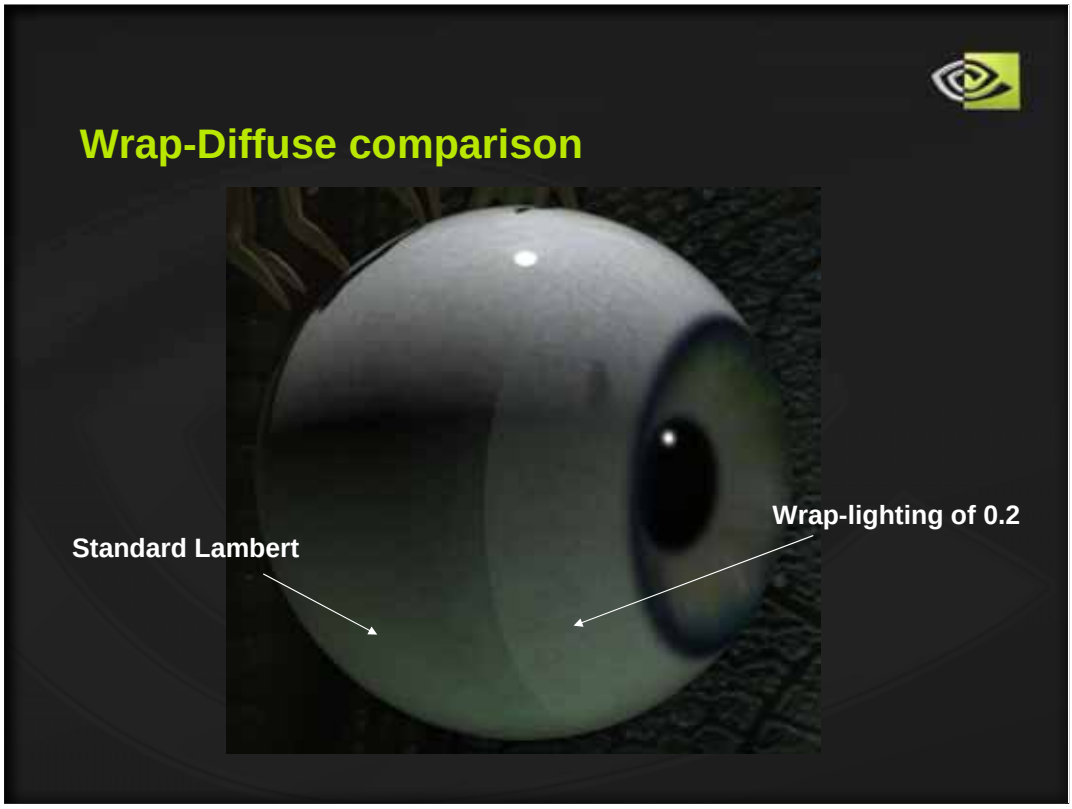
Soft wrap diffuse lighting

- GPU Gems (Simon Green):

  "Real-Time Approximations to Subsurface Scattering"

- An approximation to sub-surface scattering





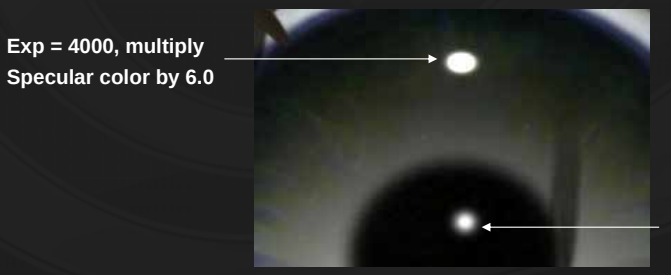
The math used for the wrap diffuse lighting:

```
float wrap = 0.2;
float wrap_diffuse = max( 0, ( dot( N, L ) +  wrap ) / ( 1 + wrap ) );
```

Now use wrap_diffuse instead of the tradition dot(N, L) term for diffuse lighting.

Environmental reflections & Cornea layer

- Simulate a transparent, smooth, reflective cornea layer on top of the entire eye
- Two components:
  - 1) reflection cube map attenuated by a fresnel term
  - 2) very bright specular highlight with a very high exponent



fresnel = 0.05 + 0.95 * (1-saturate(dot(N,-V)))^5.

total_light = (diff+spec)*(1-fresnel)+reflLight*fresnel;

Blinn-phong specular highlight: exponent = 4000. specular color of pure white (1,1,1) * 6.0.

Fresnel reflection & Cornea highlights only

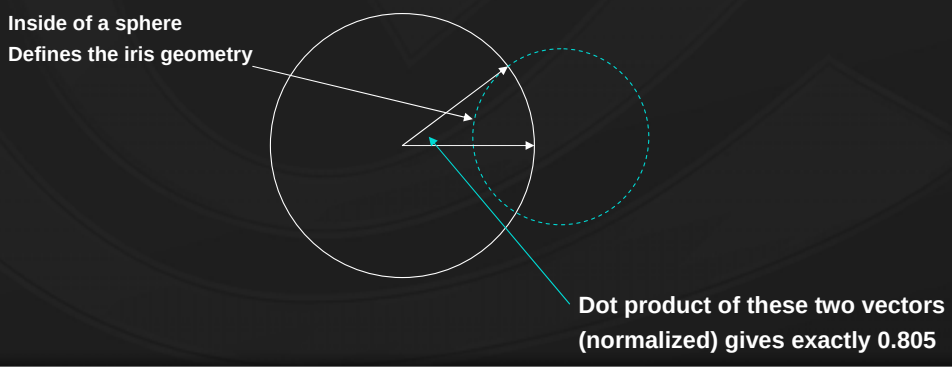
The reflected light added is subtle, but…

Fresnel vs. no Fresnel

The fresnel term blends into the dark, reflected light at the edges, changing the look dramatically

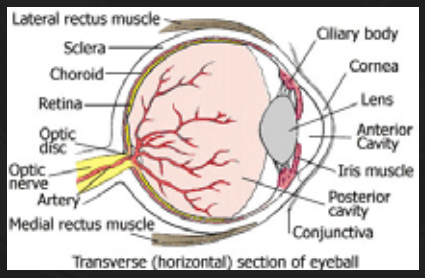
Creating the iris

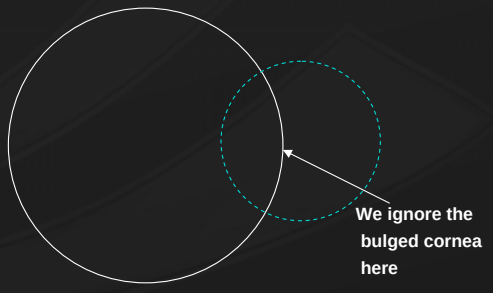
- Model the iris geometry as a sphere:
  - Easy to intersect with using ray tracing
  - Computing the normals is trivial given the position
  - The 2nd sphere is positioned perfectly so that our dot product test exactly matches the points where the 2 spheres intersect
  - Not technically the correct shape, but we need it spherical for lighting…



But the cornea and iris are not spheres!

- The cornea bulges a bit, and the iris is basically flat.
- However, the bulge is exaggerated in this diagram and a spherical eye looks fine
- We need a slightly spherical iris for lighting reasons…

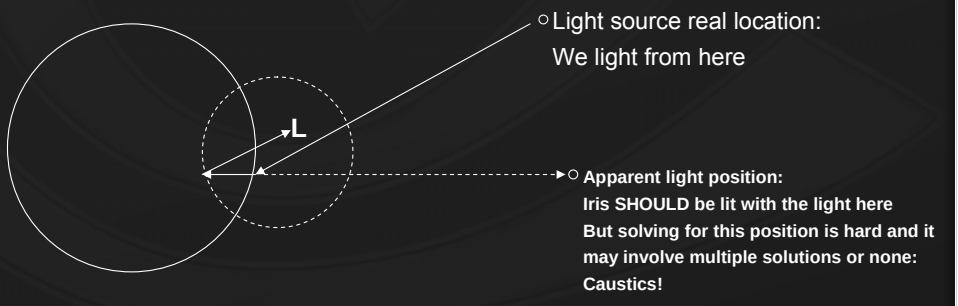




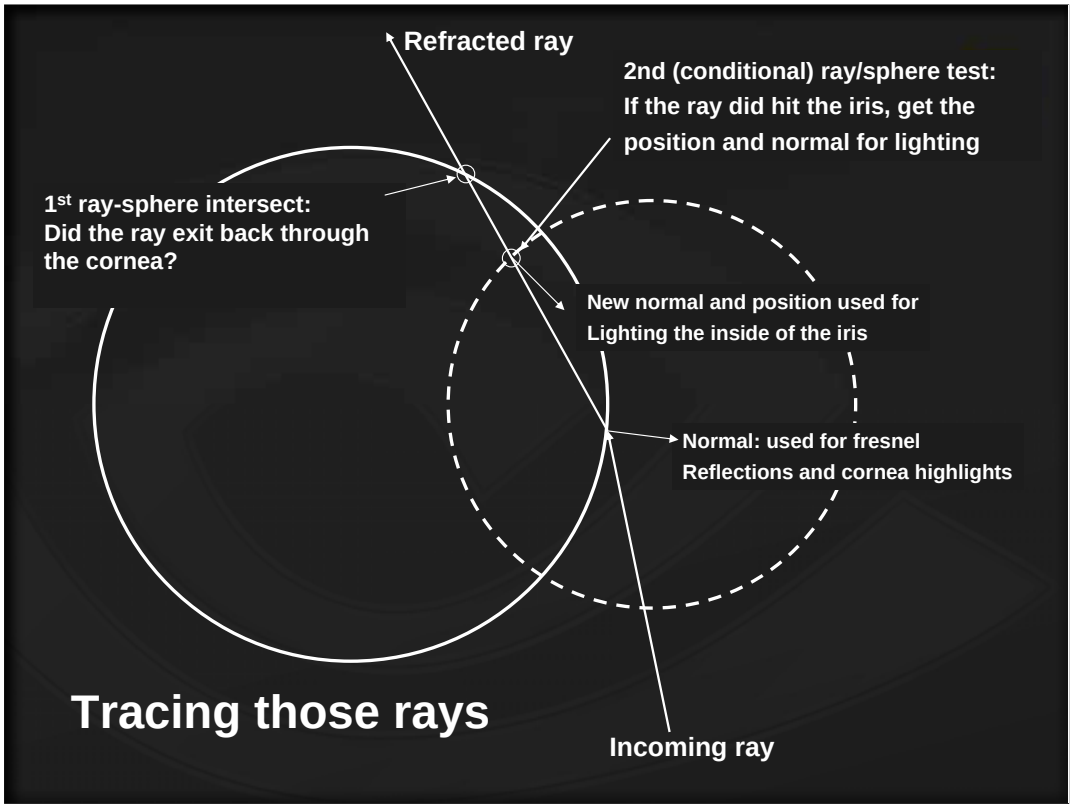
The bulge is exaggerated in this diagram. It's very subtle, so a perfect sphere was chosen, but the bulge could be modeled and still work with this technique because the incoming ray does not ray-intersect with the eyeball geometry. Only the refracted ray performs an intersection with the eyeball geometry, and so long as the back of the eyeball remains spherical, the cornea can be warped.

Lighting on the Iris

- The iris is flat in real life, but the refraction causes the lighting to behave as though it were effectively shaped like a satellite dish (i.e. spherical)
- The iris not correctly lit, because caustics are hard
- BUT, by modeling the iris as a sphere and ignoring caustics, similar results are achieved



Caustics come about because multiple rays entering the cornea with refraction can land on the same position (focusing of the rays), or even if there is only one ray per position, the density of rays changes with the focus causing more lighting in some areas versus others. This technique is difficult to model accurately (photon mapping can be used). But for an eyeball, not much focusing occurs. The bending of the light does affect the lighting though: causing a flat surface to look rounded.



Ray tracing overview:

1) In the fragment shader the object space coord, normal and eye position are passed in (object space calculations are key, because they reduce the complexity of the math substantially)

2) The dot product test decides if the current pixel is on the iris region or not

3) If so, the incoming ray is computed and, from that, the refracted ray

4) The first ray-sphere intersection is done against the procedural sphere that is perfectly aligned with the polygon sphere

5) A second dot product test with the previous ray-intersection point determines if we hit the iris or if the ray exited the eye back through the cornea (it is noted later that this dot product and the previous ray-intersection can sometimes be skipped)

6) If the iris is hit, a second ray-sphere intersect is computed to determine where, and a normal is then computed. Then the new light vectors, half angles and other information required for lighting at that position is computed

RAY INTERSECT MATH:

```
// Note: this assumes sphere space coordinates: the center of the sphere is at (0,0,0)
// rayo = ray original (point)
// rayd = ray direction (vector)
float SphereIntersect( float sphere_radius_squared, float3 rayo, float3 rayd )
{
    float3 v = float3( 0, 0, 0 ) - rayo;
    float b = dot( v, rayd );
    float disc = ( b * b ) - dot( v, v ) + sphere_radius_squared;
    disc = sqrt( disc );
    float t2 = b + disc;
    float t1 = b - disc;
   return t2;  // we know for our purposes that we always want t2
}
// The result, t2, gives us the length along our input ray where the intersection occurs
float3 intersection_coord = rayo + t2 * rayd;
float3 intersection_normal = normalize( intersection_coord); // since we are a sphere
```

REFRACTION MATH:

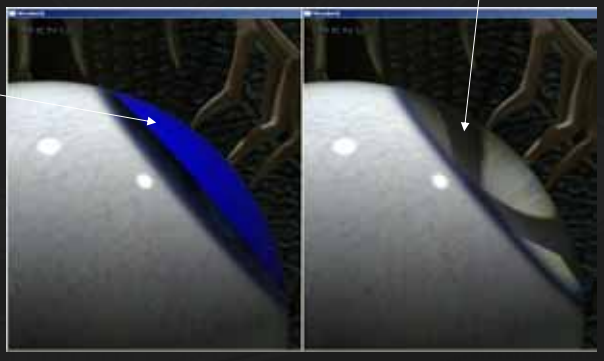
```
// it is assumed that the in_ray vector and normal are normalized
// in_u = index of refraction
float3 refract( float3 in_ray, float3 in_normal, float in_u )
{
  float3 result;
  float cosPhi = dot( -in_ray, in_normal );
  float cosTheta = sqrt( 1 – in_u * in_u * ( 1 - cosPhi * cosPhi ) );
  result = in_u * in_ray + ( in_u * cosPhi - cosTheta ) * in_normal;
  return normalize( result );
}
```

What if the light goes back through the cornea?

- If the index of refraction is 0.723, light NEVER escapes the eyeball.
- 0.723 is the (unverified) index of refraction for the cornea of a human eyeball

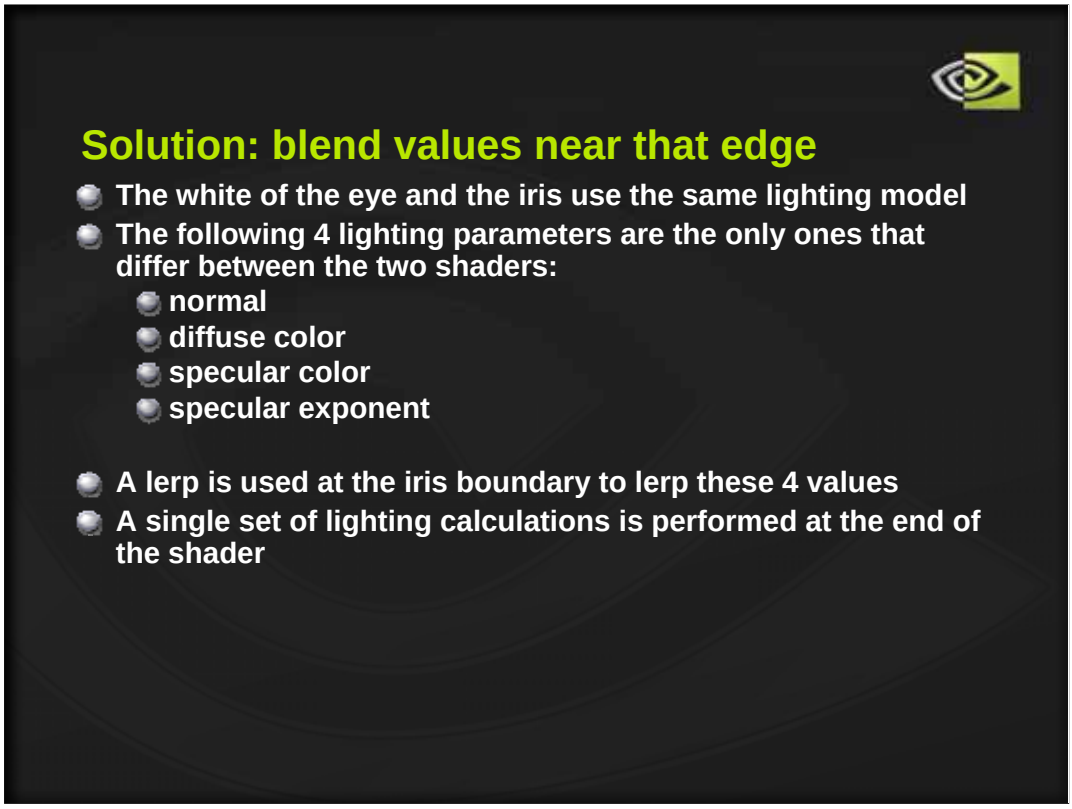
Refractive index of 0.723

No (or low) refraction: In this case it is necessary to handle the case when the ray escapes the eye. In this rendering a simple blue constant value was returned.



Hard edge resolution

- If the techniques described thus far are used, a hard edge between the white of the eye (even if the diffuse map goes to black there) and the iris region is seen



float DotProductTest = dot(objCoord, (0, 1, 0)); // used earlier to detect the iris

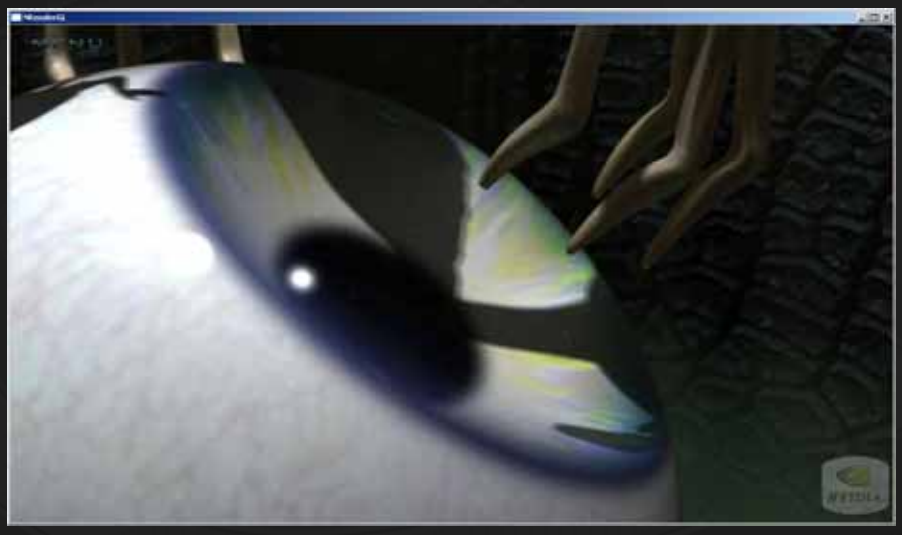
t = 1 - (DotProductTest - 0.805000) / (1 - 0.805000);

t = pow(t, 6.0); // this shifts the blend region to the very edge of the iris

For each of the listed lighting parameters, a lerp is performed with the computed t value. Then a single set of lighting computations are executed.

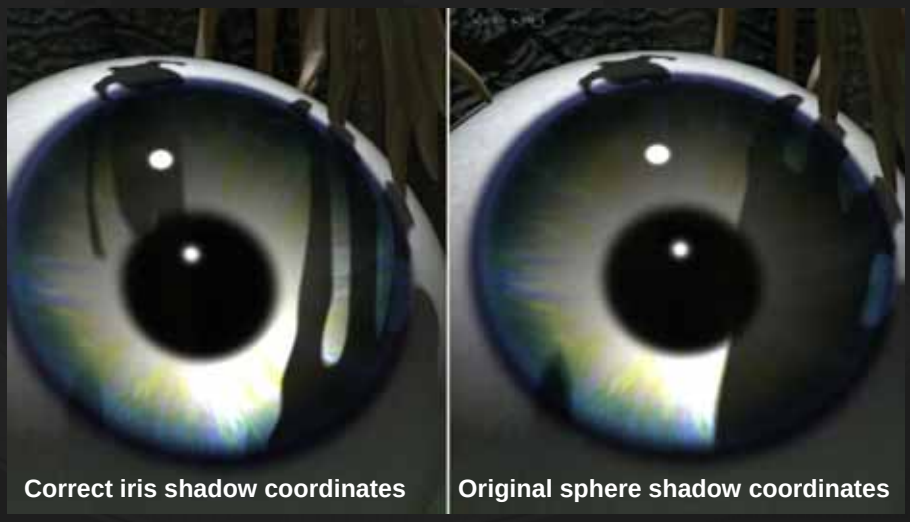
e.g.: diffCol = t * eyeWhiteDiffCol + (1 – t) * irisDiffCol;

No more hard edges



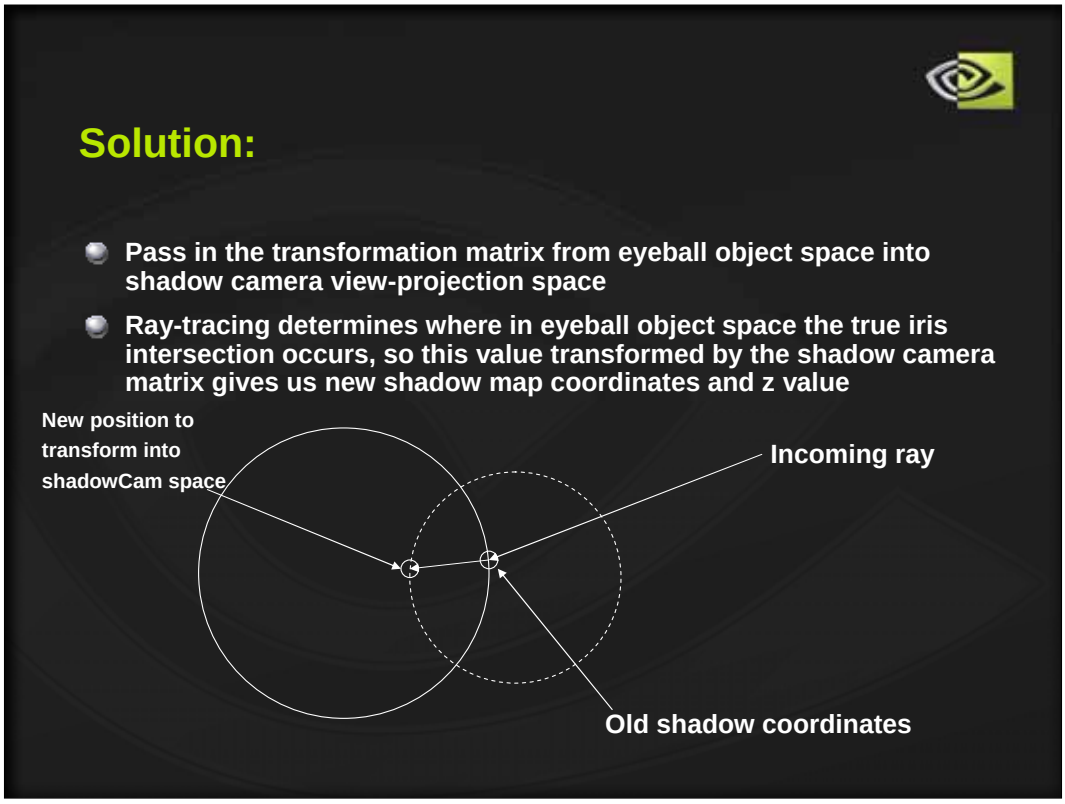
Shadows inside the eye

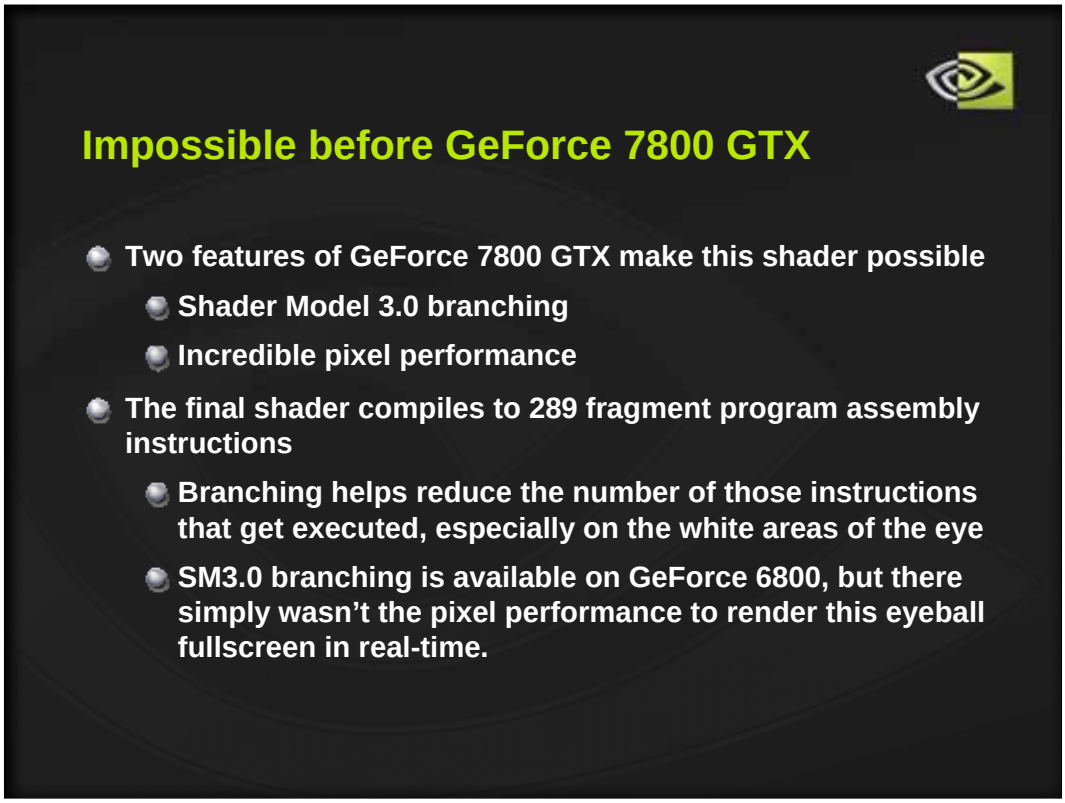
- How are the correctly warped shadows computed?

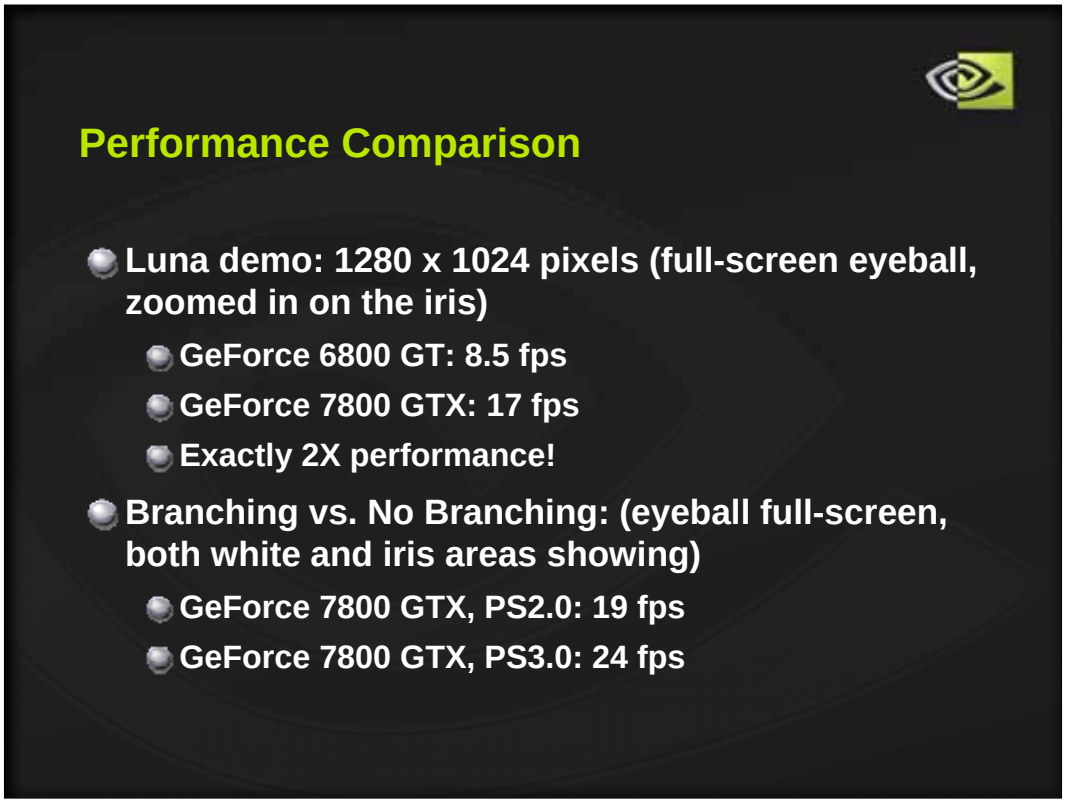


Correct iris shadow coordinates

Original sphere shadow coordinates


Solution:
Pass in the transformation matrix from eyeball object space into shadow camera view-projection space
Ray-tracing determines where in eyeball object space the true iris intersection occurs, so this value transformed by the shadow camera matrix gives us new shadow map coordinates and z value
New position to transform into shadowCam space
Incoming ray
Old shadow coordinates


Impossible before GeForce 7800 GTX
Two features of GeForce 7800 GTX make this shader possible
Shader Model 3.0 branching
Incredible pixel performance
The final shader compiles to 289 fragment program assembly instructions
Branching helps reduce the number of those instructions that get executed, especially on the white areas of the eye
SM3.0 branching is available on GeForce 6800, but there simply wasn’t the pixel performance to render this eyeball fullscreen in real-time.


Performance Comparison
Luna demo: 1280 x 1024 pixels (full-screen eyeball, zoomed in on the iris)
GeForce 6800 GT: 8.5 fps
GeForce 7800 GTX: 17 fps
Exactly 2X performance!
Branching vs. No Branching: (eyeball full-screen, both white and iris areas showing)
GeForce 7800 GTX, PS2.0: 19 fps
GeForce 7800 GTX, PS3.0: 24 fps

Applicability

- The rendering performance of this technique scales linearly with the number of pixels drawn:
  - Easily applied to the eyes of 100 characters, all on screen simultaneously
  - Refraction is quite subtle unless the eyes are fairly large on screen but is still detectable
  - The shadow map coordinate correction is probably un-necessary for the eyes in a character's head

Fancy Hero Suits, 30% off!

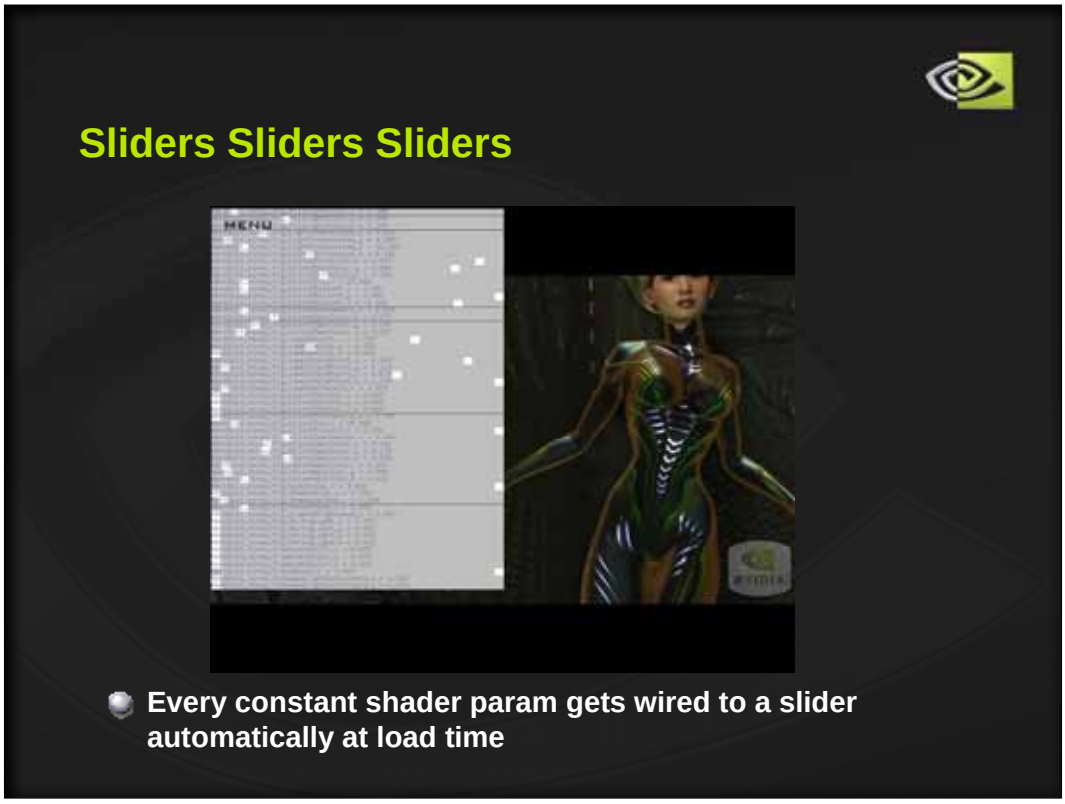
The Challenge

- Artist sketches called for a metallic suit:
  - Multiple metal layers with different properties
  - Lots of fine detail

Early concept sketch

From Start to Finish

- GeForce 7800 GTX has huge pixel shader performance, so let's use it
  - Create a very general/flexible shader
  - Use compositing: light several times with different settings and do compositing in real-time
  - Wire every possible parameter to a slider
  - Let the artist be as creative as possible
  - Optimize shader code once the final look is decided



There were initially more sliders than seen here. ☺

All constant inputs to vertex and fragment shaders are scanned at load time and pages of sliders are created and wired up automically.

During the prototyping stage, all math was done full precision and a custom mapping of texture inputs and constant inputs allowed a constant value and 3 texture inputs to each affect every shading parameter.

At the top of the shader 3 textures were sampled:

```
float3 texture1 = tex2D( diffTex, coords );
float3 texture2 = tex2D( specTex, coords );
float3 texture3 = tex2D( alphaTex, coords );
```

Then we build a custom vector:

```
float3 special = float3( 1.0, texture1.y, texture2.y, texture3.y ); // chosing y for the green channel, arbitrary choice
```

Now for any lighting parameter:

```
uniform float4 g_layer1SpecAmount;  // a 4 component constant input per parameter, all driven with sliders
```

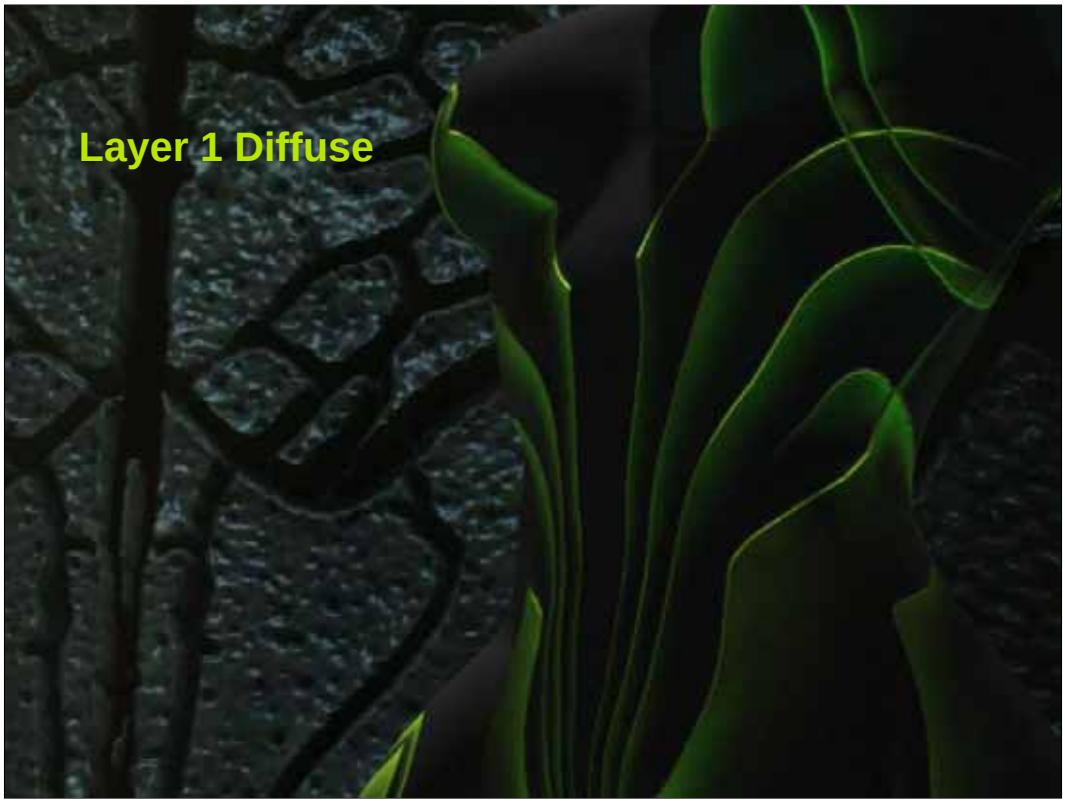
Then: `float finalLayer1SpecAmount = dot( special, g_layer1SpecAmount );`

So now g_layer1SpecAmount.x is a constant spec amount value,
g_layer1SpecAmount.y makes the diffuse texture affect the specular amount,
g_layer1SpecAmount.y makes the spec texture affect spec amount, etc…

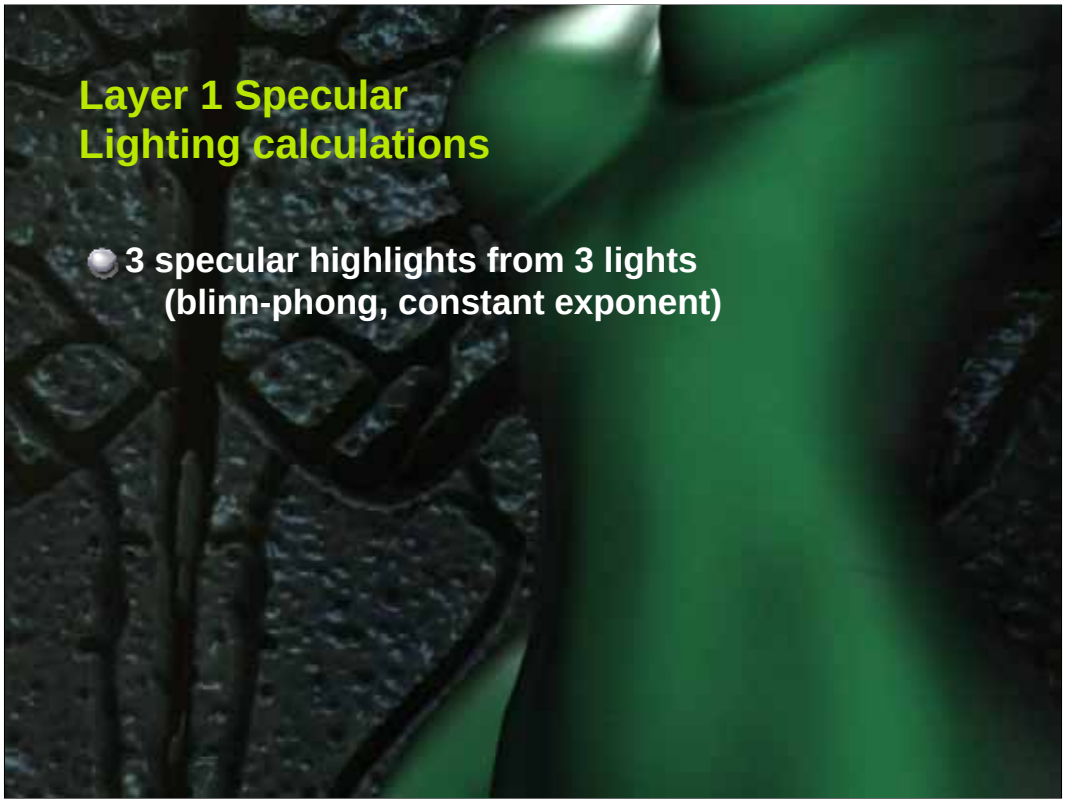
In the final shader most of these constant values were left at 0, (no parameter had all 3 textures driving it in addition to a constant), so this generality was removed during the code optimization step.

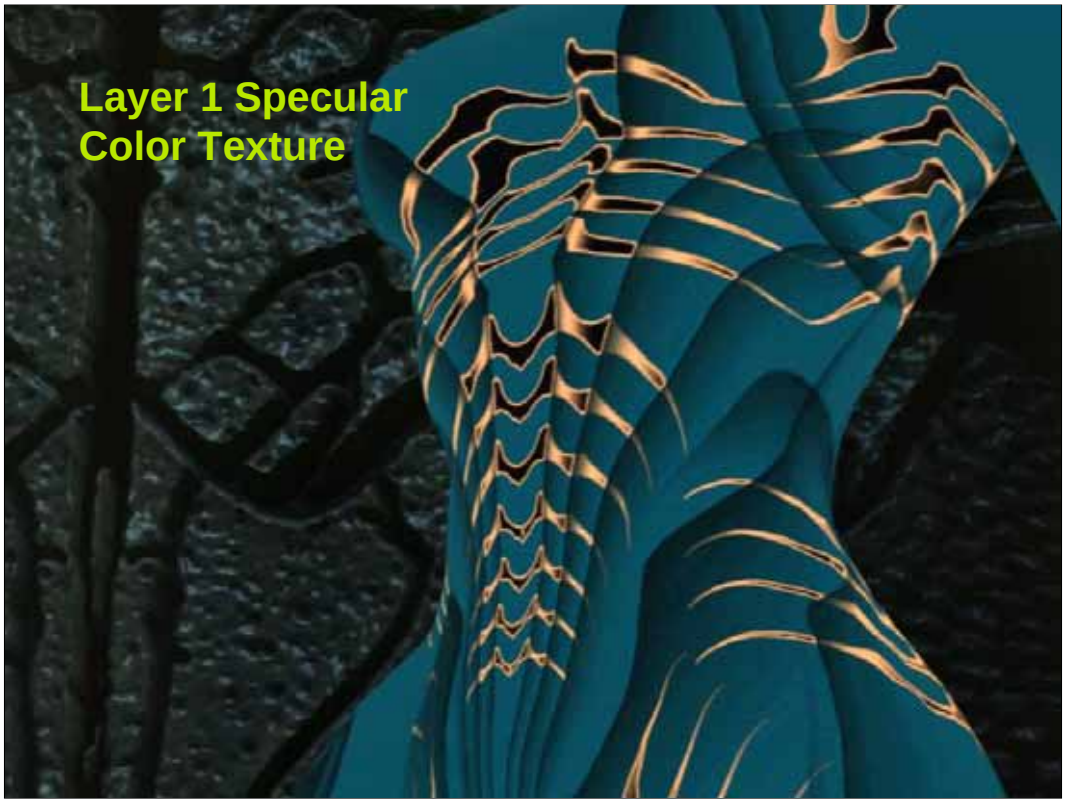
Solution Outline

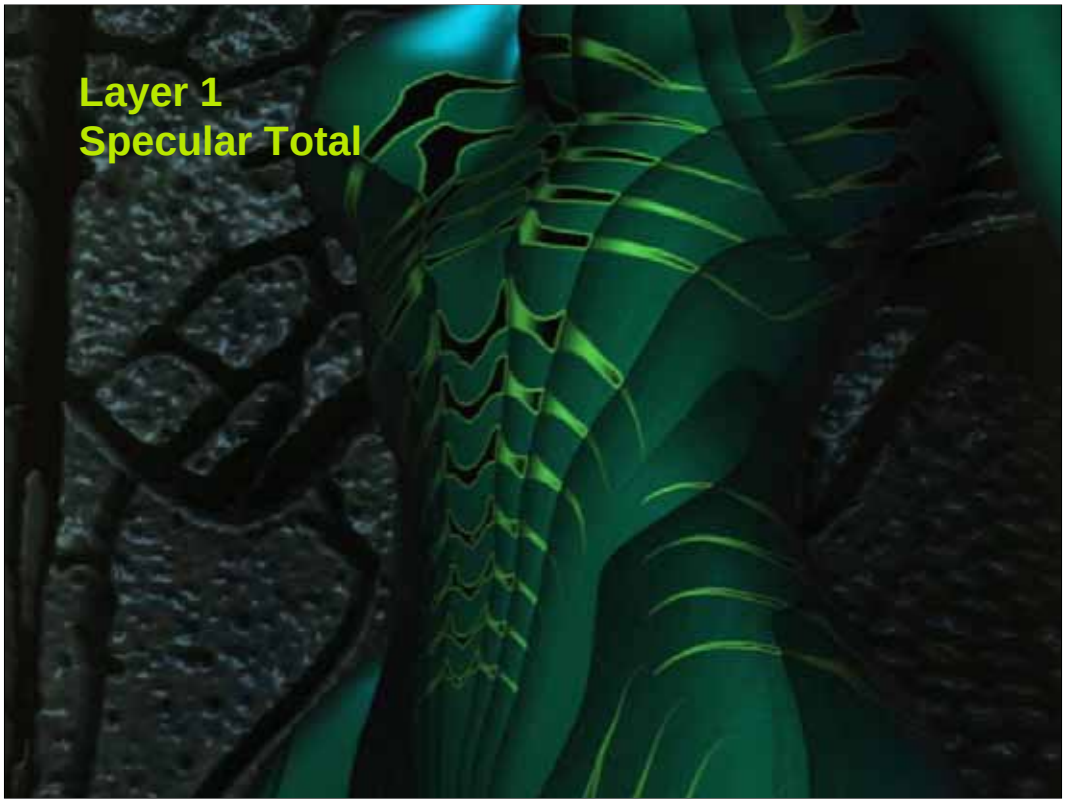
- The shader consists of 3 layers, composited in the fragment shader on top of each other
- Layer 1 defines a diffuse layer with a subtle, broad, noisy specular
- Layer 2 defines sharp metallic layers that appear to sit underneath layer 1 (but are actually compositied on top of layer 1)
- Layer 3 defines a silhouette lighting effect that provides metallic reflections

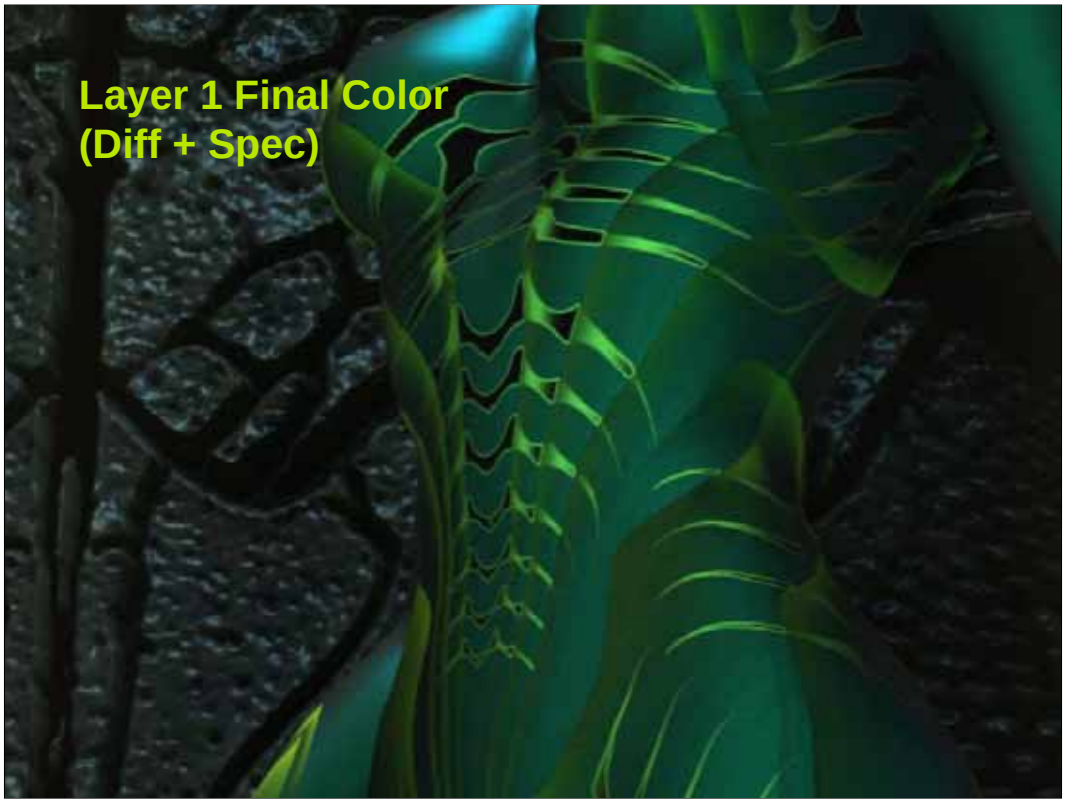

Layer 1 Diffuse

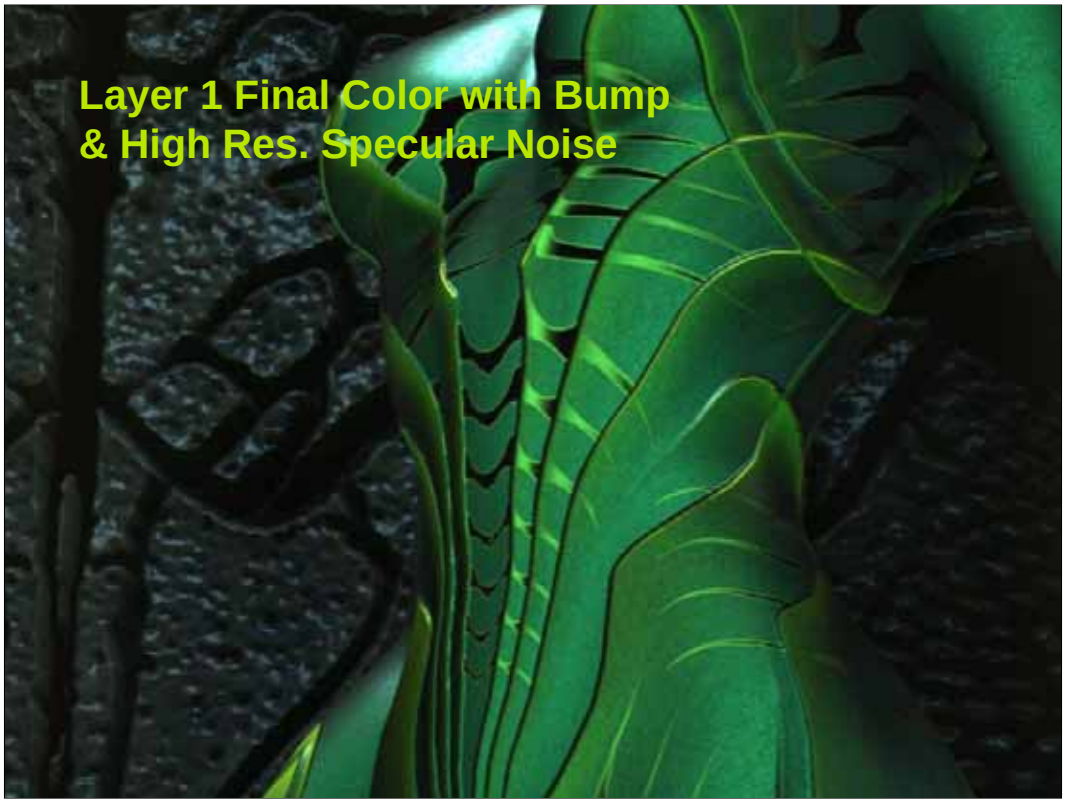

Layer 1 Specular
Amount Texture


Layer 1 Specular
Lighting calculations
3 specular highlights from 3 lights
(blinn-phong, constant exponent)

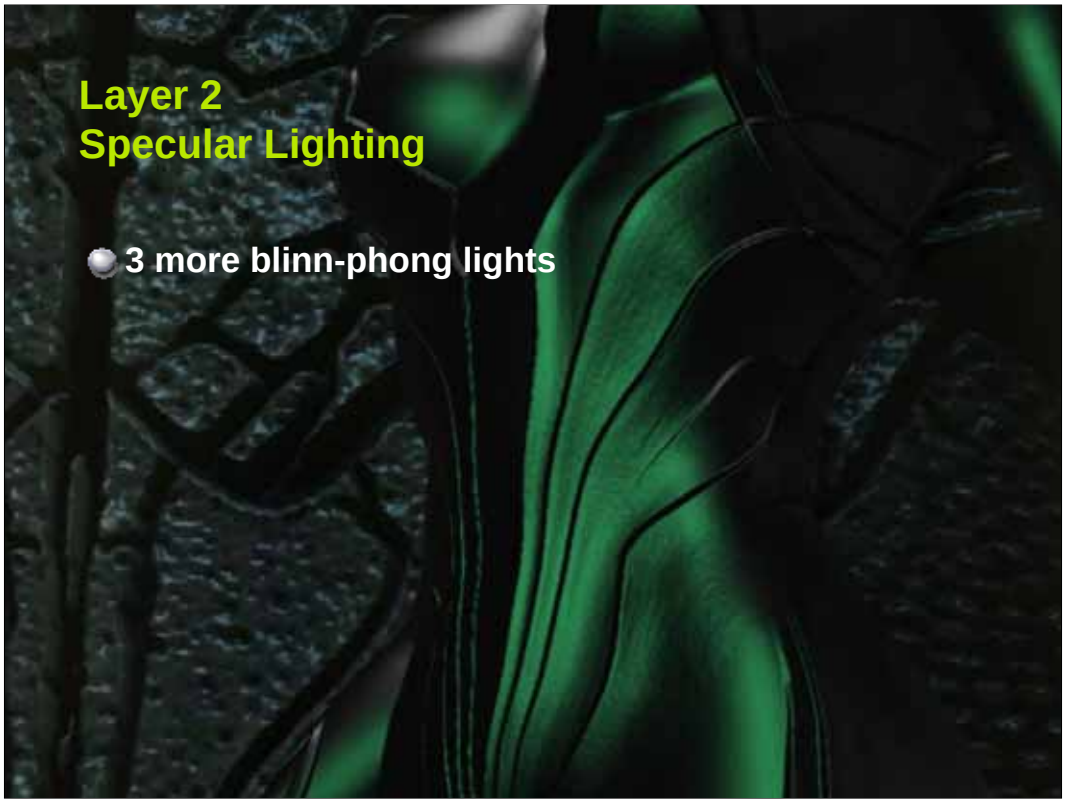

Layer 1 Specular
Color Texture


Layer 1
Specular Total

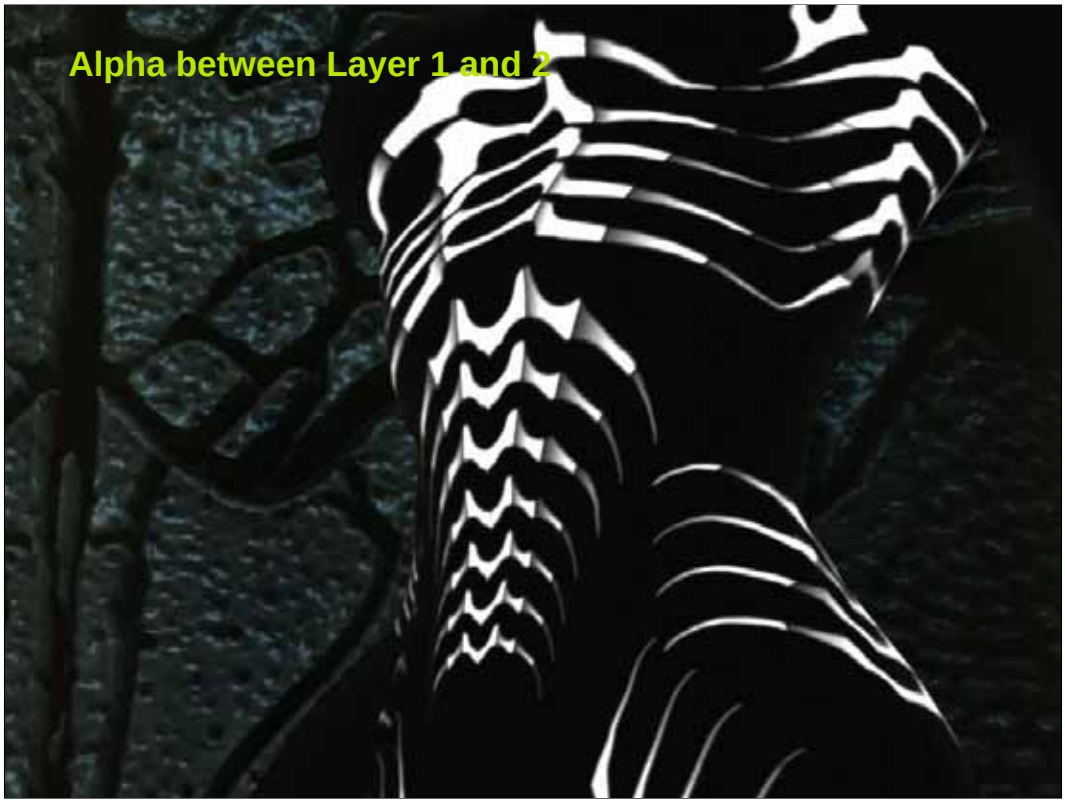

Layer 1 Final Color
(Diff + Spec)


Layer 1 Final Color with Bump
& High Res. Specular Noise

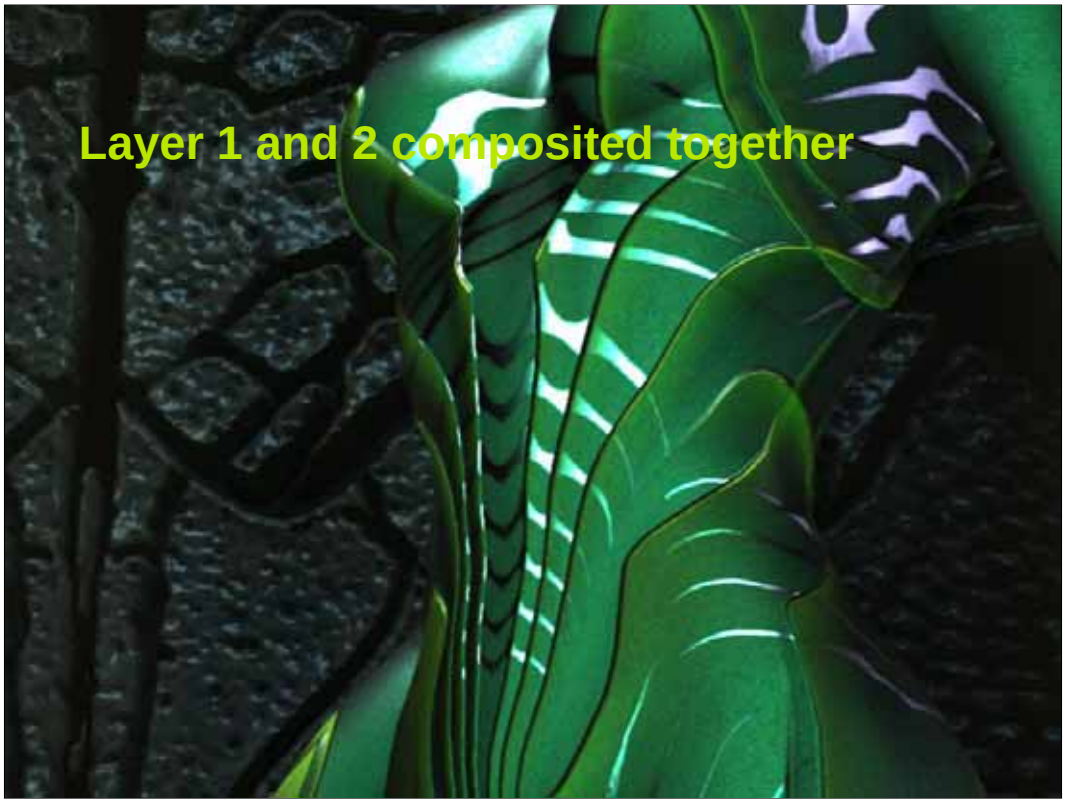

Layer 2
Specular Color


Layer 2
Specular Lighting
3 more blinn-phong lights

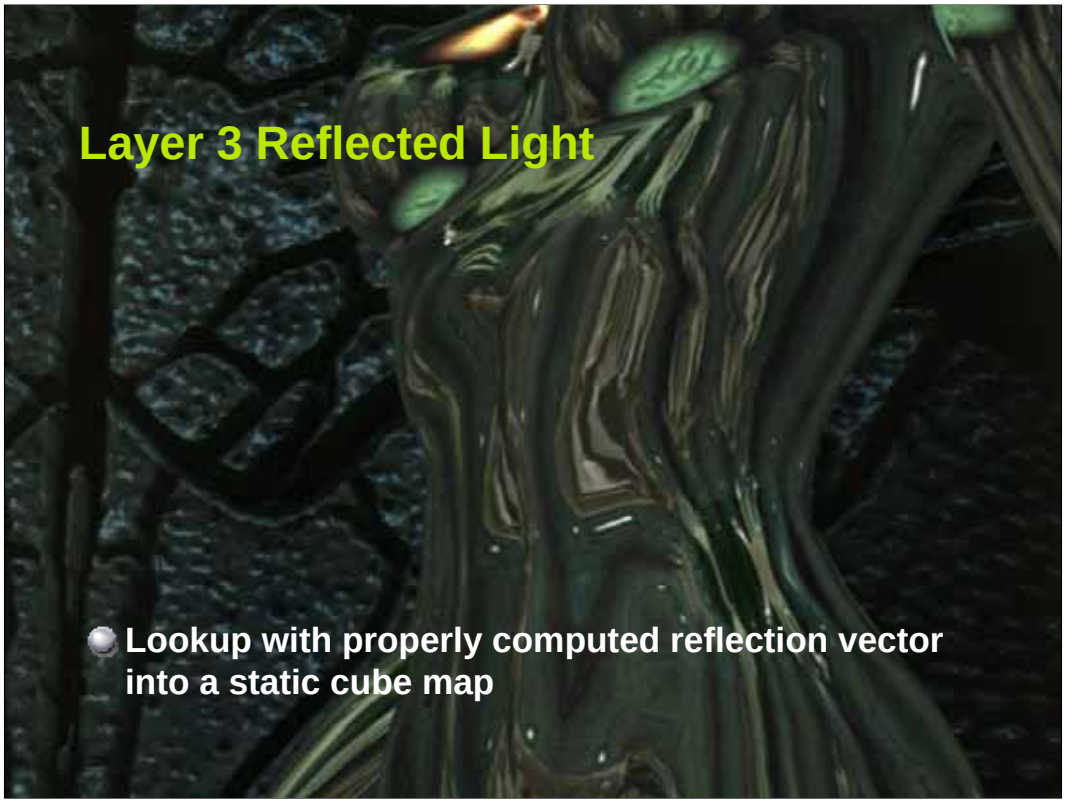

Layer 2
Final Color
No diffuse



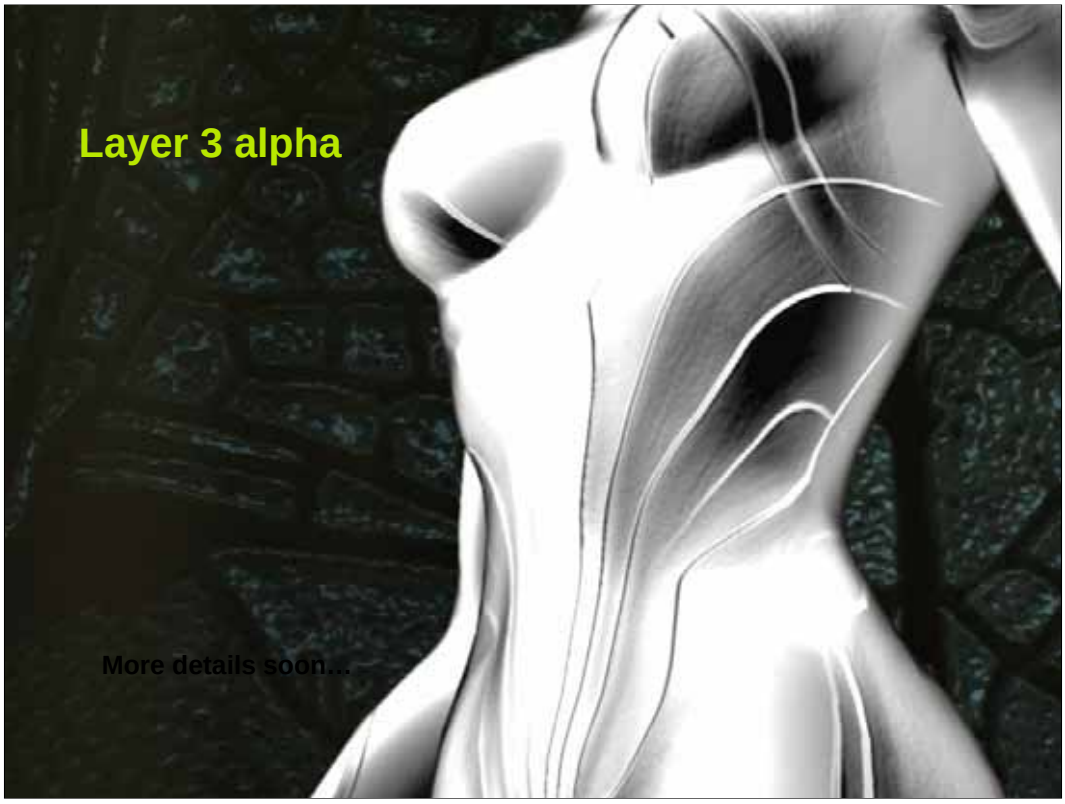
Final Color = alpha * Layer 2 + (1 – alpha) * Layer 1

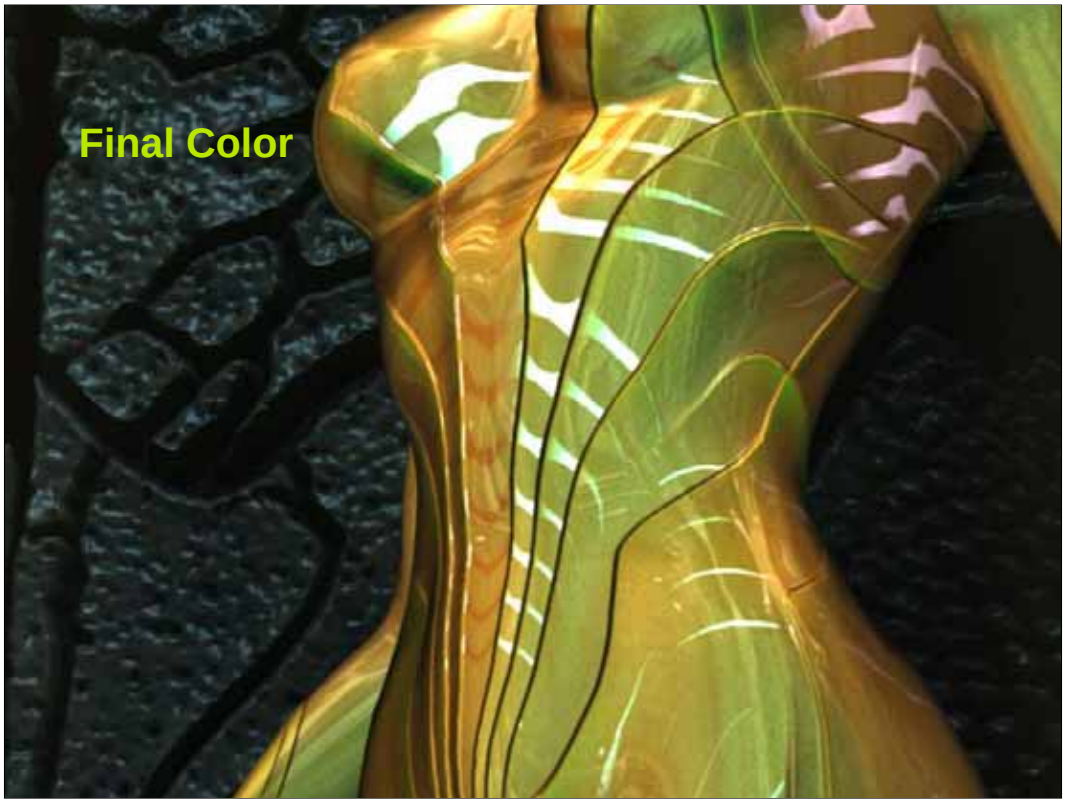


Final Color = alpha * Layer 2 + (1 – alpha) * Layer 1


Layer 3 Reflected Light
Lookup with properly computed reflection vector into a static cube map

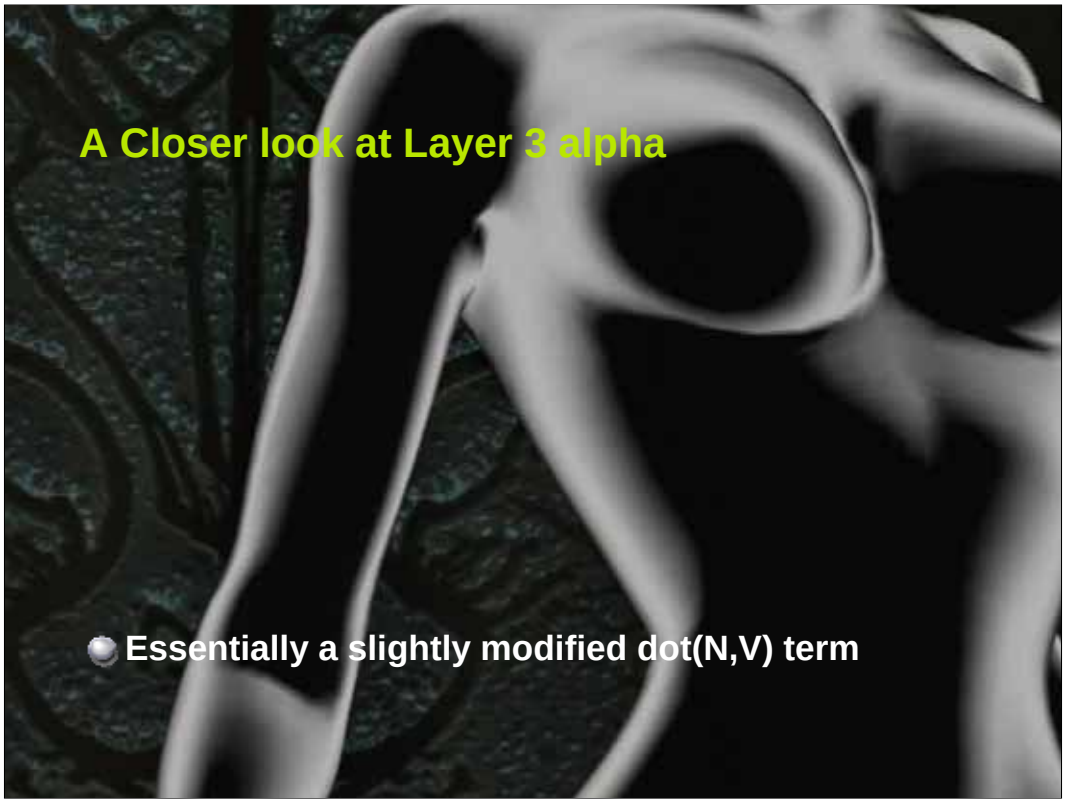

Layer 3
Color


Layer 3 alpha
More details soon…

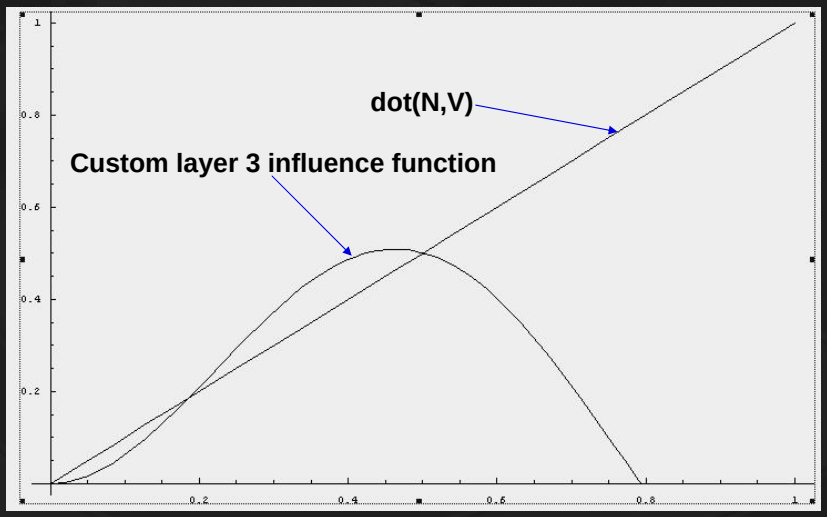


Final color = Layer3Color * alpha + (1 – alpha) * (Layer1and2composited)

The previous shots had tons of the layer3 effect as a result of the plasma flying by, lighting her suit


A Closer look at Layer 3 alpha
Essentially a slightly modified dot(N,V) term

Comparison of dot(N,V) and the custom falloff



N dot V provides a silhouette lighting term to work with: the shiny, reflective layer (layer 3) is applied at grazing angles

The N dot V is run through a cubic polynomial to smoothen the ramp

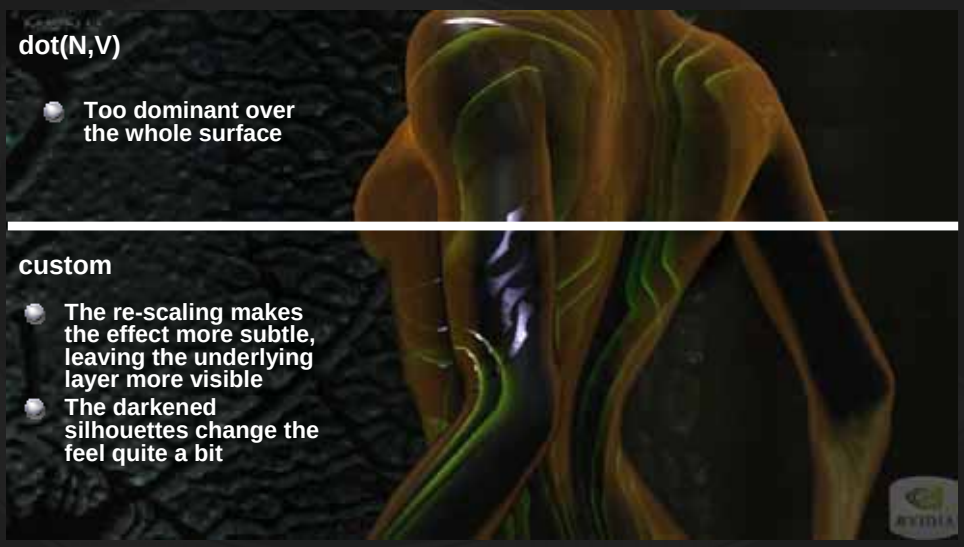
The result is scaled to shift it's area of influence as N dot V ranges from 0 to 1

Finally, the dampening factor is mulitplied in to bring the result back to 0 near extreme grazing angles

This changes the appearance of the highly reflective layer

Visual Comparison

Visual Comparison



Final shader stats

- 238 pixel shader instructions
- 8 texture inputs per pixel
- 24 texture samples per pixel
- 6 Blinn-Phong specular calculations per pixel

Mad Mod Mike Demo

- Indirect Lighting
- Depth of Field
- Omnidirectional Shadows

- 25 renderpasses
- Extremely complex shading:
  - Mike: 85 – 150 instructions
  - Environment: 74 – 144 instructions

Direct vs. Indirect Lighting

Direct Lighting:

- Light that falls directly on a surface and illuminates it.

Indirect Lighting:

- Light that bounces off of a surface (acquiring some of its color in the process) and then hits *another* surface, illuminating it.

Direct Lighting

In this image, Mike is lit by just two point lights.

Notice...

- That a good portion of his body is completely dark (unlit).
- That the color of nearby surfaces doesn't affect his lighting at all.

Indirect Lighting

Here, we've added *indirect* lighting to the main character.

Notice...

- Mike picks up the strong red bounced light from the tool chest.
- Formerly unlit areas now feature *realistic* ambient light.

More examples [direct light only]

More examples [plus indirect light]

Indirect Lighting

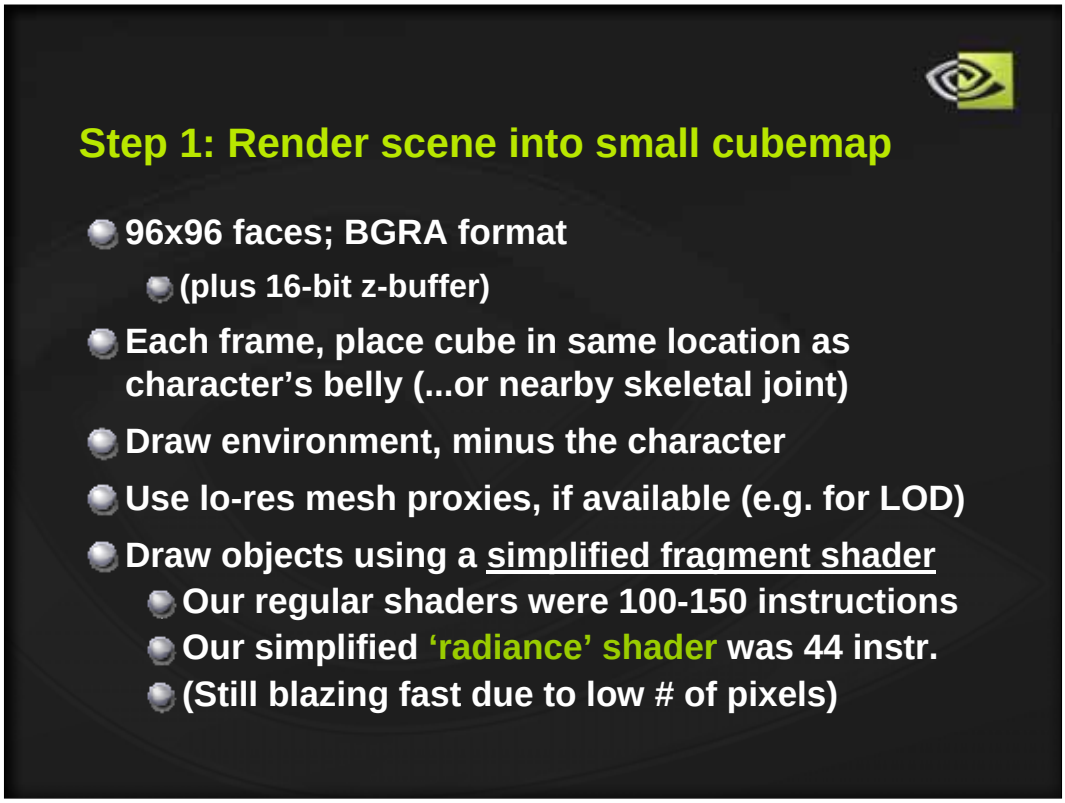
- Why not use many dim point lights to achieve the effect?
  - Would need *dozens* of lights
    - At least one light for each salient piece of furniture
    - Several lights for each wall, floor, etc.
  - Would need to set & maintain the lights'...
    - ...positions
    - ...colors (based on average object colors)
    - ...*intensities* (lots of assumptions & approximations here!)
  - Slow, ugly, high-maintenance...

Indirect Lighting

A much simpler approach:

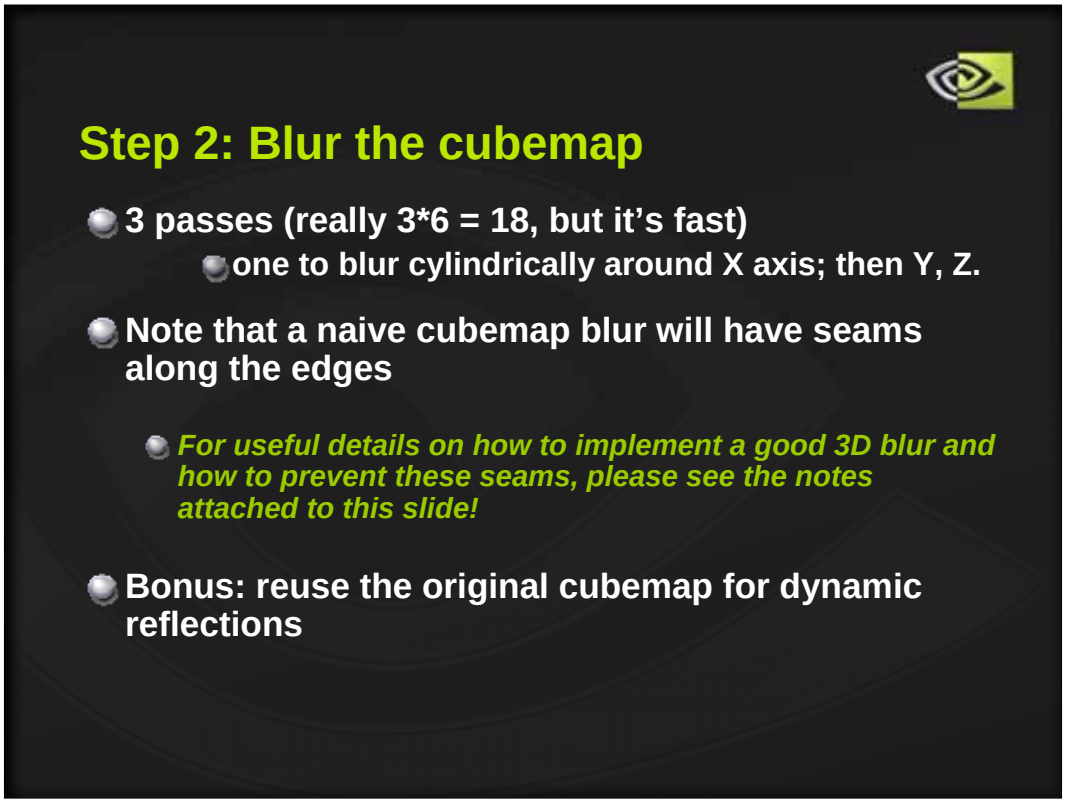
Each frame...

1. Render scene (minus main character) into a small cubemap.
2. Blur the cubemap.
3. When lighting the character, use samples from the blurred cubemap to augment the traditional (direct) lighting.


Step 1: Render scene into small cubemap
96x96 faces; BGRA format
(plus 16-bit z-buffer)
Each frame, place cube in same location as character's belly (...or nearby skeletal joint)
Draw environment, minus the character
Use lo-res mesh proxies, if available (e.g. for LOD)
Draw objects using a simplified fragment shader
Our regular shaders were 100-150 instructions
Our simplified 'radiance' shader was 44 instr.
(Still blazing fast due to low # of pixels)

Simplified shader

- Compute simple diffuse (N • L) lighting for 2 point lights; skip specular lighting.
- Sample main “color” texture only (no bump, spec, etc. maps)
  - Use +1 mipmap bias since results will be blurred anyway
- Attenuate the brightness of the final color you write by the distance of the shaded point from the cube (character) center!
  - Do this so faraway objects contribute less light (really, color)
  - Near objects contribute 100%; far objects darkened to ~20%
- You can over-saturate the final color to exaggerate the effect.
- We didn’t factor in shadows, but probably should have!



geometry: draw a 1x1x1 cube (12 polys) into each face of each cube
only 2 of the 12 will actually draw pixels
don't reuse any vertices
encode the cube's "face normal" in each vertex's TEXCOORD0

vertex shader: one for each blur axis (X, Y, Z)
vertex shader computes 7 vectors & passes them on to fragment shader in TEXCOORD0..6
first one is just the normalized world-space vertex position

```
float3 cube_lookup_dir =
                normalize(world_space_vertex_position);
float3 baseVec = cube_lookup_dir;
v2f.tex0 = baseVec;
```

next 3 are that vector repeatedly rotated by some fixed angle around X axis (or Y/Z)
we used 11.5 degrees (0.2 radians)
last 3 are that (original) vector repeatedly rotated in the opposite direction
(this way, we only need to take sin and cos once per vertex)

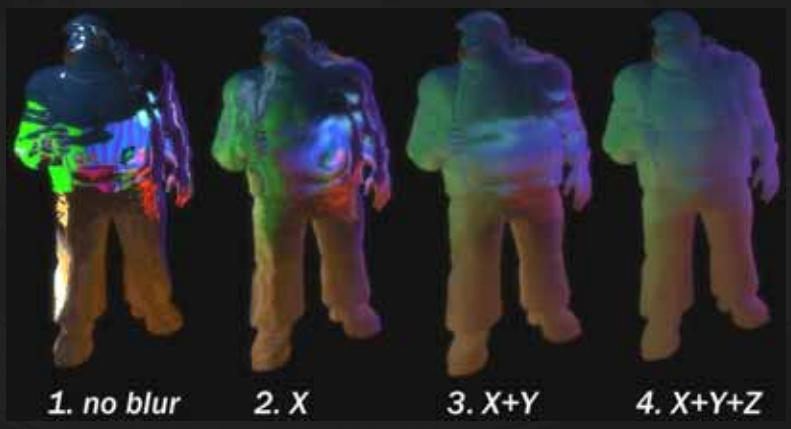
Seams:
The cubemap will have visible seams in it, where colors don't match up at face edges after blurring.
Solution: in the vertex shader, push "baseVec" away from the face normal by a small amount.
Amount depends on size of cubemap you're reading *from*.

```
// 64²: -.019, 80²: -.0182; 96²: -.0175; 128²: -.016
const float bias = -0.0175;
float3 baseVec = lerp(cube_lookup_dir, a2v.tex0, bias);
```

Fragment shader:
Given 7 vectors from the vertex shader, takes 13 samples of the src. cube
7 along the vectors from the vertex shader
6 along interpolated vectors
every sample has a little constant random jitter (looks better)

Indirect Lighting

- Here's a chrome Mike, perfectly reflecting the 4 stages of the blurred cube map.
- As you can see, it's important to do all 3 cylindrical blurs!



Step 3: Light the Character

- When lighting the character, use samples from the blurred cubemap to augment the direct lighting.
- The big question: what vector should we use to sample the cubemap?
- Reminder: cubemaps are indexed by a *single* 3D vector!
  - (Ideally, we'd want a 6D volume texture that we sample using surface positions & normals. But that's a little ways off…)

Indirect Lighting

- Sample the blurred cube using the:

  vertex normal

- Looks as if environment is very far away; sub-optimal for a close environment
  - Way too many green highlights (from left)
  - Feet aren't picking up much brown/purple (from bottom)

Indirect Lighting

- Sample the blurred cube using the:

  vertex position relative to character's (or cube's) center

- An improvement, but would be better yet if surface normal had *some* impact on the lighting…

Indirect Lighting

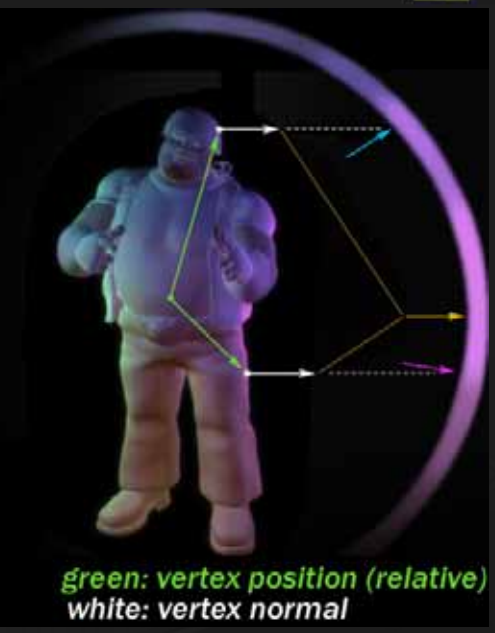
- In practice, blend the two vectors together
- Here, the blended sampling vector is based mostly on the normal, but vertex position has an influence
- ~ Analogous to Bjorke's *localized reflections* technique in GPU Gems.
- Resulting lighting looks very realistic!

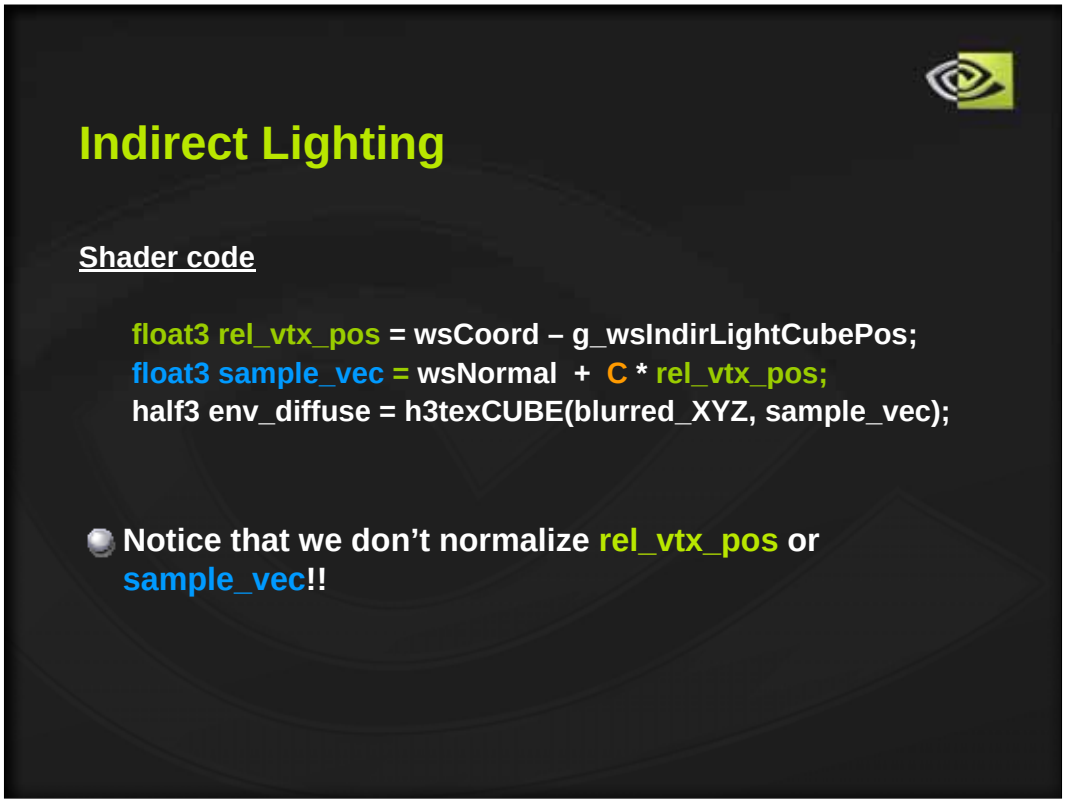
Technical Explanation

- At right, consider the point on the helmet vs. the point on the leg
- Both points have same normal direction (white arrows)
- If we use *just the normals* to sample the cubemap, both will sample it at the orange arrow and return a pinkish color.
- This feels wrong for the helmet point - intuitively, *the helmet sample vector should hit closer to the end of the dashed line!*
- The blue vector is what we'd need for that...
- Build blue vector by adding a bit of the relative vertex position (green arrow) to the normal (white arrow):

  blue arrow = white arrow + C*green arrow

  sample vector = normal + C*rel_vtx_pos
- Bends the normal "up" for the helmet and "down" for the leg, so both hit the right spot! (blue arrow for helmet; pink arrow for leg)
- Works just the same in 3 dimensions.




Indirect Lighting
Shader code
float3 rel_vtx_pos = wsCoord – g_wsIndirLightCubePos;
float3 sample_vec = wsNormal + C * rel_vtx_pos;
half3 env_diffuse = h3texCUBE(blurred_XYZ, sample_vec);
Notice that we don’t normalize rel_vtx_pos or sample_vec!!

Indirect Lighting

How much to mix in the relative vertex position (‘C’) depends on how far away the environment is (roughly) and how large the character is (in world space units)...

1. environment infinitely far away: use C == 0
2. environment is a large room: use C ~= 2.5 / wsCharHeight
3. environment is a small room: use C ~= 7.0 / wsCharHeight

- ‘wsCharHeight’ is rough estimate of the height of your character, in world space units.
- “small room” means about the size of the rooms in the Mad Mod Mike demo (relative to the size of Mike).
- these values were empirically determined; tweak to your liking!

Other Notes

- A mix of ~75% direct & ~25% indirect lighting looks best.
  - Traditional direct lighting (N • L) should provide most of the light, and is where the effects of shadows & bump maps are most visible.
  - Indirect light approximates light bouncing off of walls and objects, helping the character look truly immersed in the environment.
- To add Indirect Lighting for multiple characters, you'd have to render & blur a separate cubemap for each one.

Indirect Lighting

- **Finally, if you have any shadows in your shader, slightly dim the env_diffuse term where there is shadow**
  - **env_diffuse *= (0.8 + 0.2*shadow_mult);**
  - **not exactly accurate, but it looks nice**

- **Final lighting equations:**
  - diffuse_light = env_diffuse
    + $\sum$ ((N • L_i) * $LightColor_i$ * $shadow\_mult_i$);
  - final_color = diffuse_light * diffuse_color
    + spec_light * spec_color;

Indirect Lighting

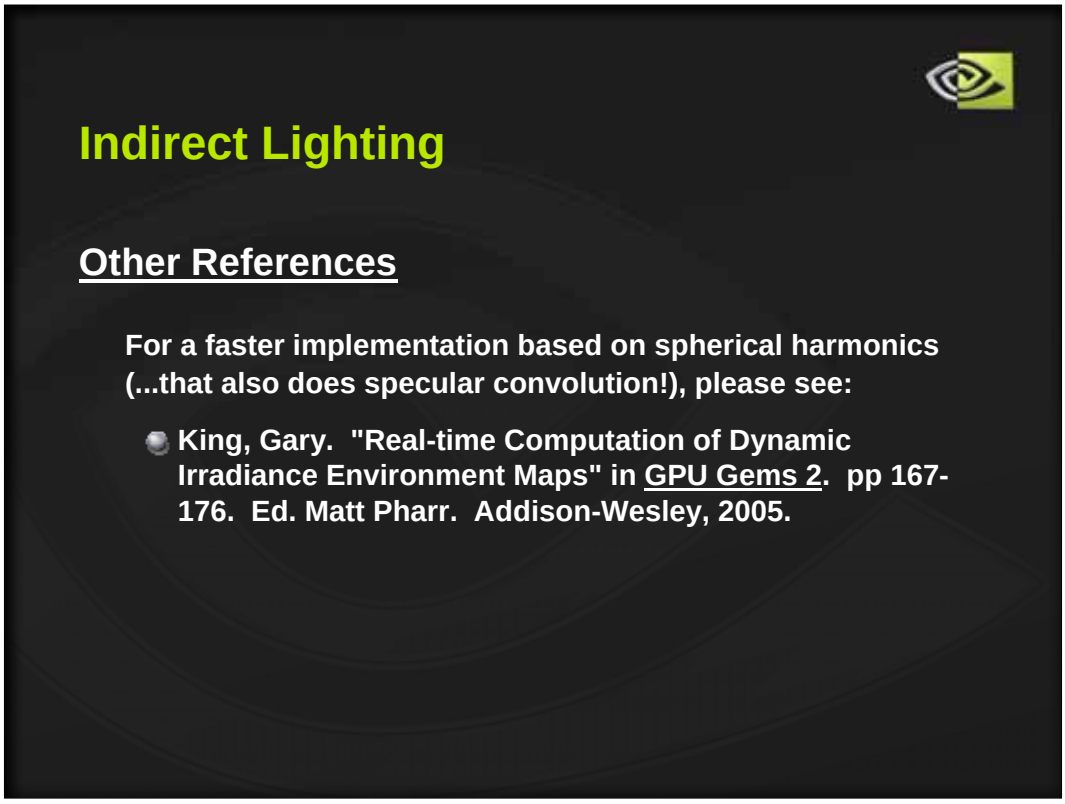
Performance:

Mad Mod Mike running on a GeForce 7800 GTX:

- Render-to-cube pass — 1.0 ms / frame *
- Cube-blurring passes (together) — < 0.9 ms / frame
- Extra final shading cost — + 1.0 – 1.5 ms / frame **
- Total cost: — 2.9 – 3.4 ms / frame
- Cost, in FPS: — costs about 2.5 fps at 30 fps

* depends on # of objects, # of vertices, and depth complexity (amount of overdraw). These #s are for Mad Mod Mike demo.

** depends on # of pixels the character occupies on-screen


Indirect Lighting
Other References
For a faster implementation based on spherical harmonics (...that also does specular convolution!), please see:
King, Gary. "Real-time Computation of Dynamic Irradiance Environment Maps" in GPU Gems 2. pp 167-176. Ed. Matt Pharr. Addison-Wesley, 2005.

Depth of Field

- Photos in the real world have a *focal depth*, a distance at which objects are in focus. Objects nearer or farther get progressively blurrier.
- In computer graphics, though, every pixel in a rendered image is perfectly in focus - *unless you work hard to avoid it.*

Depth of Field: Layered method

Layered method: render objects to two different images (layers), blur the background layer, and composite.

- (+) Looks good when each object fits neatly into either the “foreground” or “background”
- (–) Objects “pop” as you move around the scene and they migrate between foreground & background layers
- (–) Looks bad when a single object should span both layers (...e.g. a long railing, table, etc).

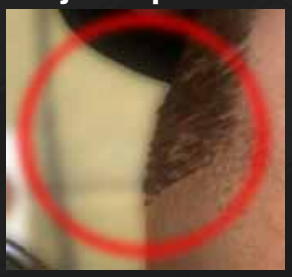
Depth of Field: Depth-based blur method

Depth-based blur method: blurring kernel radius is dilated *per-pixel* based on Z depth

(+) Maintenance-free (don't have to group objects into "foreground" vs "background" every frame)

(+) Looks good when a single object spans both

(–) When blurring, pixels from foreground "bleed" onto blurry background; this looks really bad!



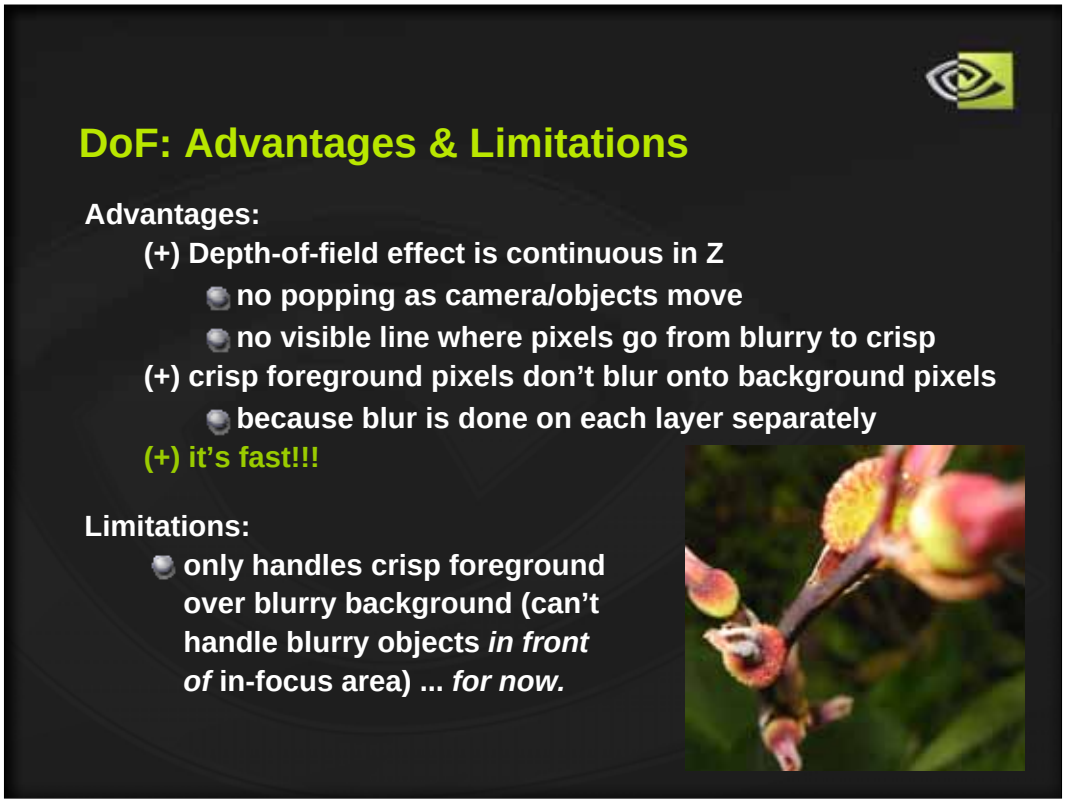
Depth of Field: Combined method

Our solution: combine the two approaches

- Render the scene twice: a "far" pass and a "near" pass.
- Two passes are identical except for the camera's near/far clip planes.
  - All objects exist in both passes *(...although most will be view-frustum culled from at least one pass).*
- Then blur each image *(separately!)* using depth-based blur

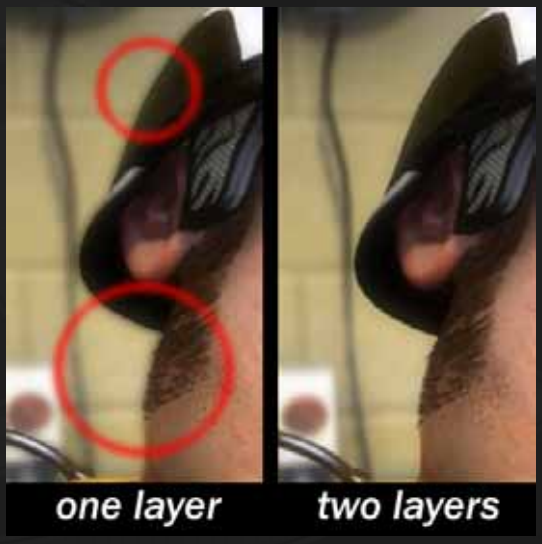
Process

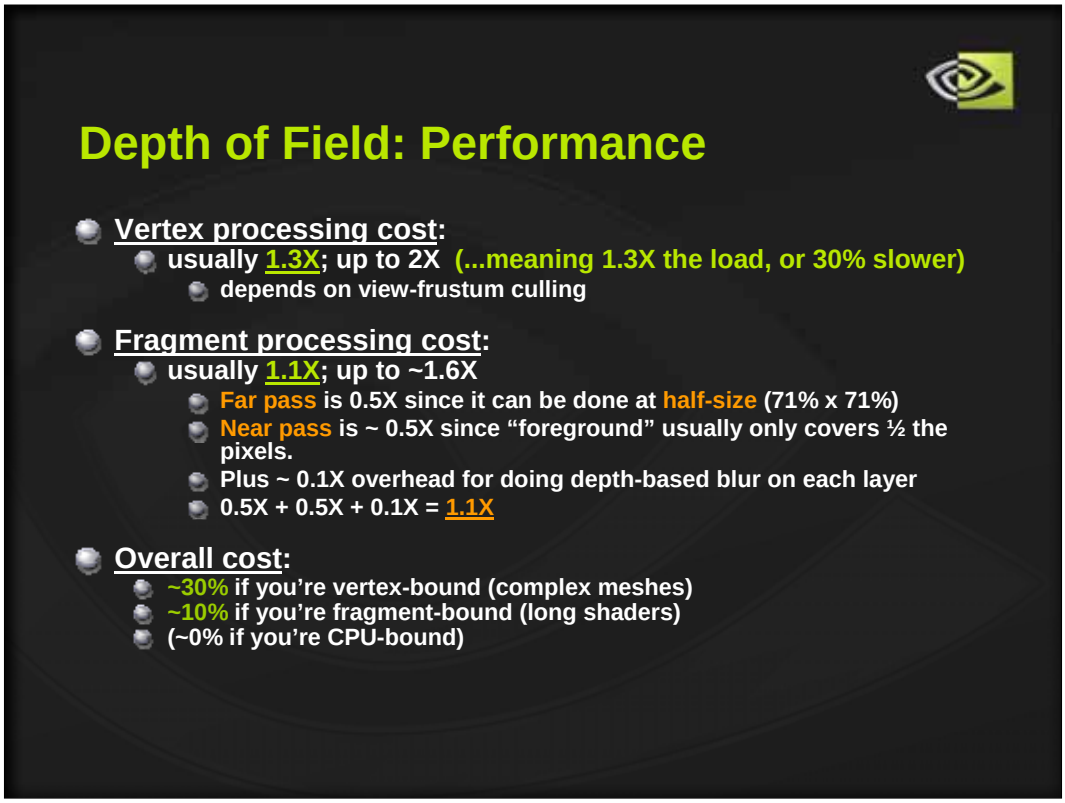
1. Render far pass.
2. Draw a fullscreen quad to the near pass, sampling the far pass results (color + depth) & doing depth-based blur as you go. Write depth as 1.0.
3. Render near pass objects on top of that “background” image.
4. Do depth-based blur on the near pass.


DoF: Advantages & Limitations
Advantages:
(+) Depth-of-field effect is continuous in Z
no popping as camera/objects move
no visible line where pixels go from blurry to crisp
(+) crisp foreground pixels don't blur onto background pixels
because blur is done on each layer separately
(+) it's fast!!!
Limitations:
only handles crisp foreground over blurry background (can't handle blurry objects in front of in-focus area) ... for now.

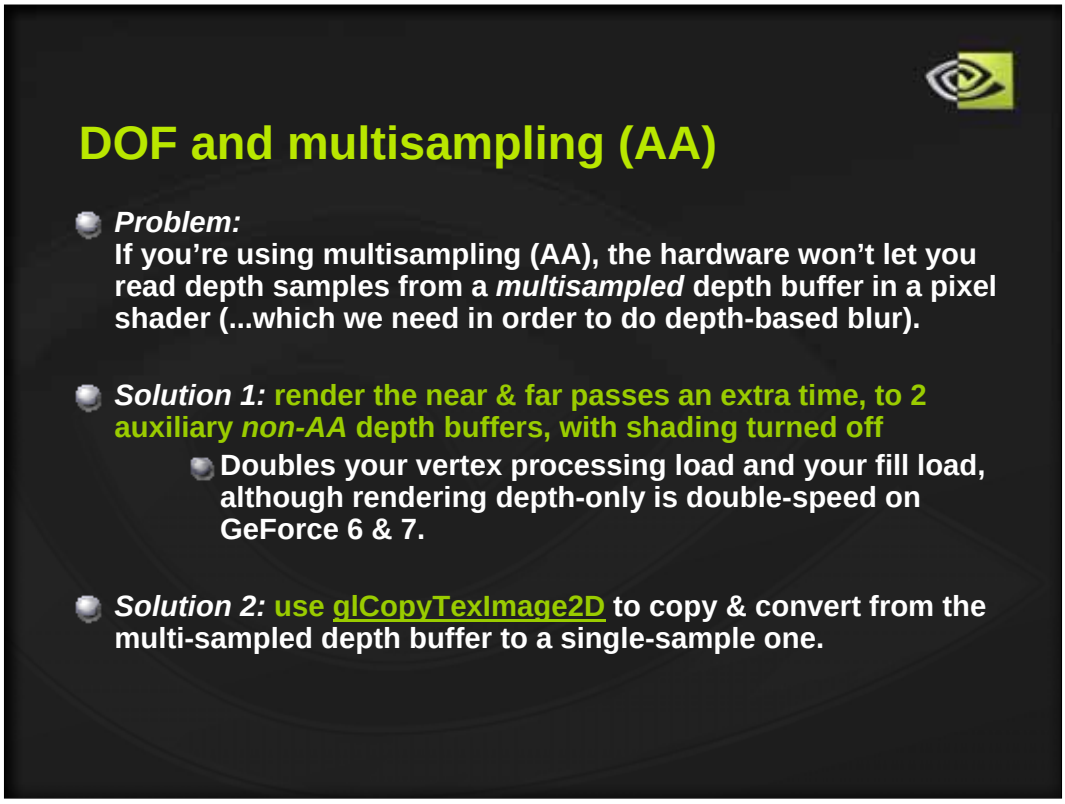
Foreground in focus, background blurry

- Notice that with only one layer, the “near” pixels contaminate the blurring of the “far” pixels. (Not enough information!)
- With two layers, the foreground doesn’t bleed onto the background.





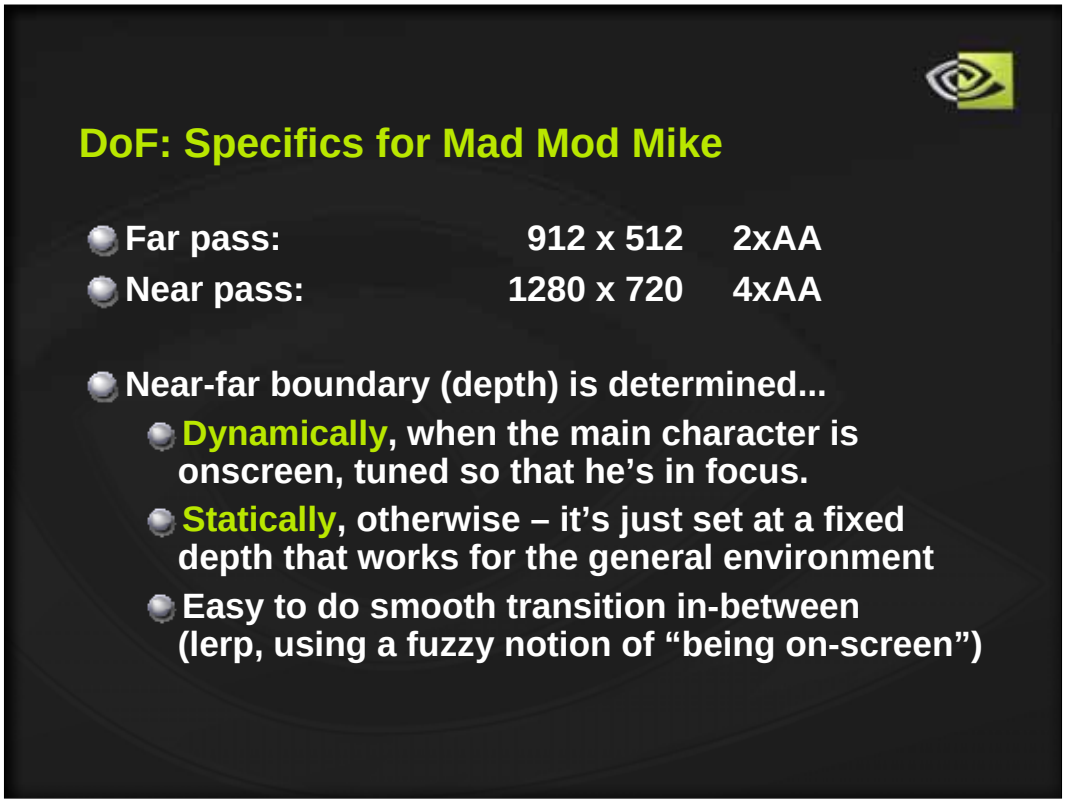
vertex processing cost:

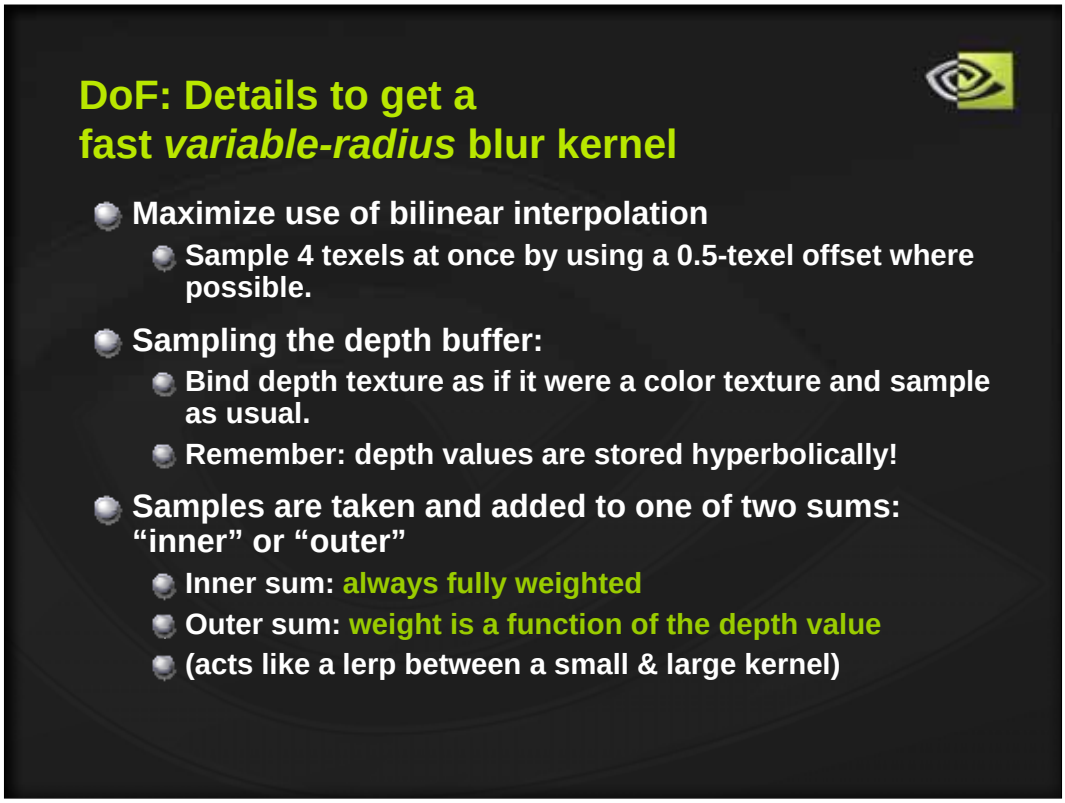

DOF and multisampling (AA)
Problem:
If you're using multisampling (AA), the hardware won't let you read depth samples from a multisampled depth buffer in a pixel shader (...which we need in order to do depth-based blur).
Solution 1: render the near & far passes an extra time, to 2 auxiliary non-AA depth buffers, with shading turned off
Doubles your vertex processing load and your fill load, although rendering depth-only is double-speed on GeForce 6 & 7.
Solution 2: use glCopyTexImage2D to copy & convert from the multi-sampled depth buffer to a single-sample one.

DoF: One more tricky part!

- When rendering objects to the near pass, use depth-compare function of LESS instead of LESS_EQUAL.
- This guarantees that all pixels with depth values of 1.0000 came from the *far* pass.
- Then, in the depth-based blur of the near pass, don't blur pixels whose depth value is exactly 1.0000, since they came from the far pass.
  - *Otherwise all pixels from the far pass would get double-blurred and - worse yet - crisp foreground objects would bleed color onto the blurry background objects.*
- Recommendation: use 24-bit depth for near pass

  (or if using AA, just for the auxiliary depth buffer);
  - Otherwise, lack of precision will cause some of the farthest near-pass pixels to not get blurred at all, creating a band of crispy pixels.


DoF: Specifics for Mad Mod Mike
Far pass: 912 x 512 2xAA
Near pass: 1280 x 720 4xAA
Near-far boundary (depth) is determined...
Dynamically, when the main character is onscreen, tuned so that he's in focus.
Statically, otherwise – it's just set at a fixed depth that works for the general environment
Easy to do smooth transition in-between (lerp, using a fuzzy notion of "being on-screen")



* depth values are stored hyperbolically, so blur radius has to be a funky function of depth buffer lookup; hack it for speed, though (accuracy does *not* matter)

we used:

far pass:

```
half2 blur_rad = saturate(   (f.xx – half2(0,1)*0.5 )*2    );
```

near pass:

```
half blur_rad = saturate( (f - 0.85) * (1.0/(1 - 0.85)) );
```

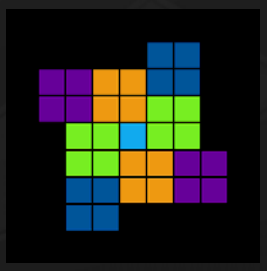
(we should have used a 1D texture lookup to accelerate these!)

*if far pass is at lower resolution, you'll want to blit first to a same-size texture for this blur operation (so bilinear interp. alignment is right), then blit *that* to background of near pass with non-texel-aligned samples.

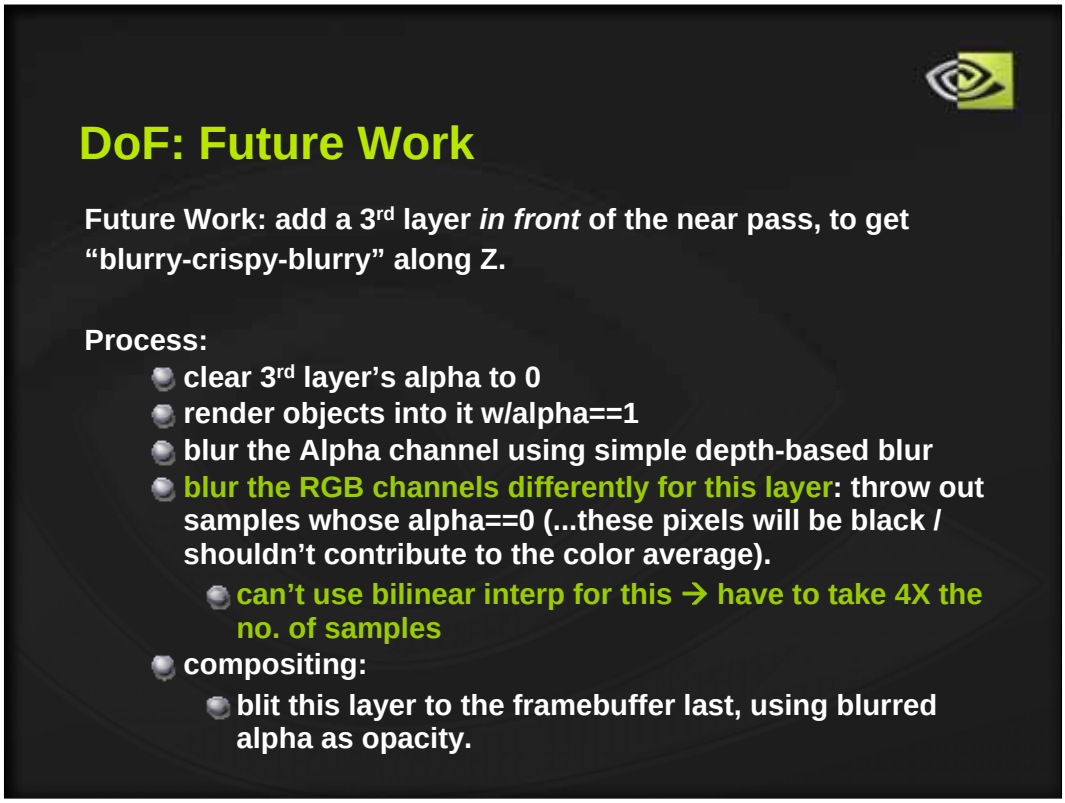
* *Note: multiply sample (and its weight*

DoF: Details to get a fast *variable-radius* blur kernel

- **Far pass:** 17 samples (top image)
  - inner sum = 5 samples; weight is 17
  - outer sum = 12 samples; weight is 0..48 (as depth ranges from 0..1)



- **Near pass:** 9 samples (bottom image)
  - inner sum = 5 samples; weight is 17
  - outer sum = 4 samples; weight is 0..16 (as depth ranges from 0..1)
  - smaller kernel because near pass is less blurry



Shader code:

```
half4  samp0  = sample(x0y0).rgba;
...
half4  sampN  = sample(xNyN).rgba;
half   blurred_alpha = (samp0.a + ... + sampN.a);
half3  blurred_rgb = (samp0.rgb * samp0.a + ... + sampN.rgb * sampN.a) / blurred_alpha;
outColor = half4(blurred_rgb, blurred_alpha);
// (note: this shader code is incomplete – blur kernel needs to be resizeable)
```

Other potential future work:

Allow larger maximum blur radius for the far pass

- Mad Mod Mike: far (blurry) pass looked good with an up-to-65-pixel kernel.
- For a third layer, you'd probably need "mega-blur"
  - for larger blurs, definitely faster to use a 2-pass separable convolution! (1D blur in X, then Y)

Get "net" blur kernel for nearest pixels in far pass, & farthest pixels in near pass, to be virtually identical

- this would eliminate any visible artifacts at the near-far boundary.
- in practice, though, close is good enough.

Omnidirectional Shadows

Our criteria for hard vs. soft shadows:

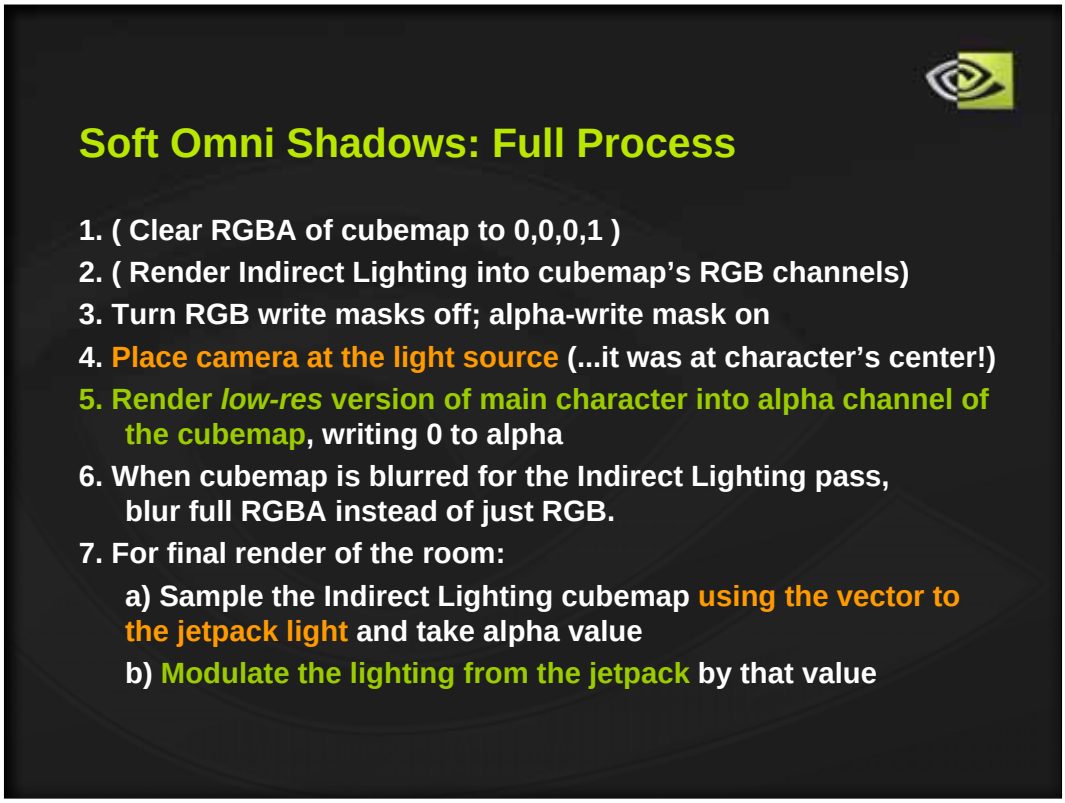
Objects close to the light should cast soft shadows; faraway objects should cast hard shadows.

Mike's jetpack casts 3 types of shadows:

1. *Soft omnidirectional* shadows that his body casts on the room
2. *Hard omnidirectional* shadows that room objects cast on each other
3. (Hard planar shadows that he casts on himself)

Soft Omni Shadows

- Soft omni shadows piggyback on the Indirect Lighting cubemap; they sit in the alpha channel.
- They are projective... they have no concept of a depth test
- Means shadows are projected or painted on, like decals, depth-unaware
- Works well as long as you know the shadow receivers (room objects) won't come between the light and the shadow-casters (Mike).


Soft Omni Shadows: Full Process
1. (Clear RGBA of cubemap to 0,0,0,1)
2. (Render Indirect Lighting into cubemap's RGB channels)
3. Turn RGB write masks off; alpha-write mask on
4. Place camera at the light source (...it was at character's center!)
5. Render low-res version of main character into alpha channel of the cubemap, writing 0 to alpha
6. When cubemap is blurred for the Indirect Lighting pass, blur full RGBA instead of just RGB.
7. For final render of the room:
a) Sample the Indirect Lighting cubemap using the vector to the jetpack light and take alpha value
b) Modulate the lighting from the jetpack by that value

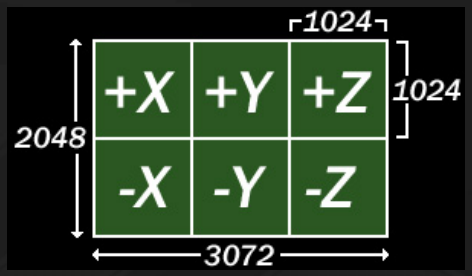
*6:

Hard Omni Shadows

- Hard omni shadows would ideally use a depth cubemap with 1024x1024 faces, but hardware doesn't support *sampling from* depth cubemaps yet.
- Instead, use a "virtual" cubemap (or "VCM"), which is a 3072 x 2048 2D depth texture that we conceptually divide into six 1024 x 1024 tiles.
  - Think of it as an unwrapped cubemap.
- Implementation based partially on the paper:
  - "Efficient Omnidirectional Shadow Maps" Gary King & William Newhall. In ShaderX3, pp 435-448. Charles River Media, 2004.

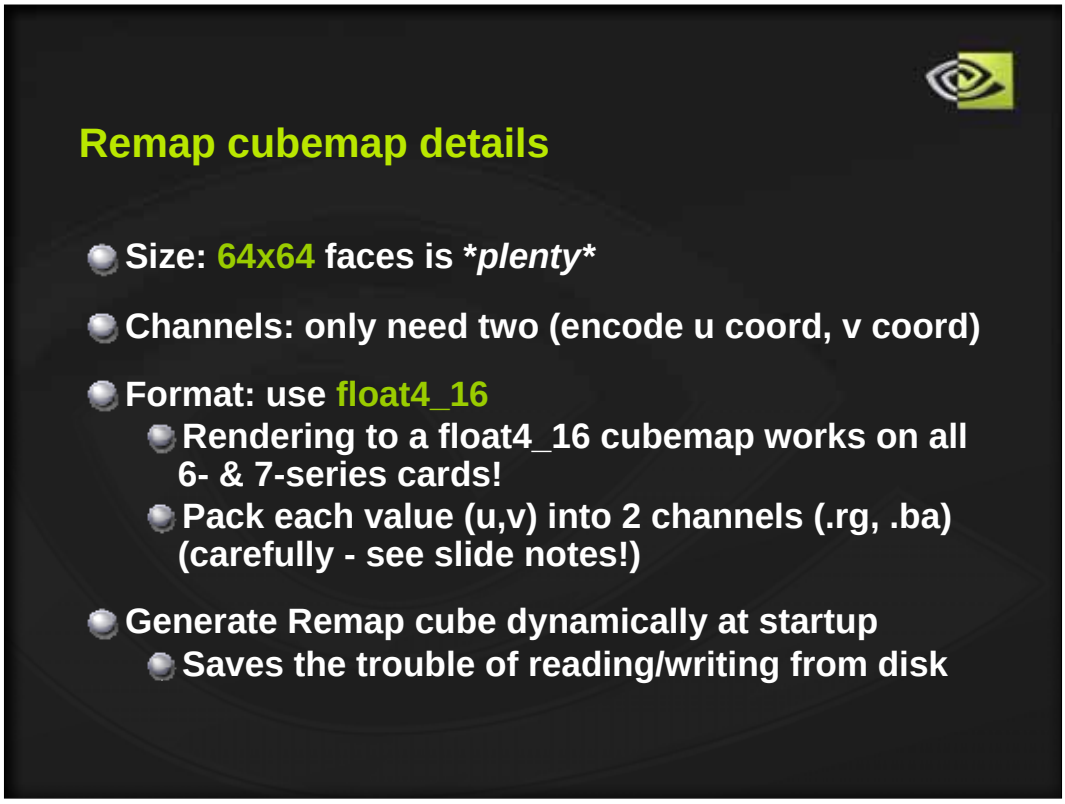
Rendering Depth to the VCM [shadow-casting]

- Render room objects [depth only / no shading] up to six times, to six different 1024 x 1024 tiles (viewports) on the 3072 x 2048 texture.
- Each viewport's camera is aligned to a different axis: { +X +Y +Z –X –Y –Z }
- *(Performance note: most objects get view-frustum culled in 3-5 of the passes, reducing vertex & fill loads)*



Sampling *from* the VCM: [shadow-receiving]

- Cubemaps are usually indexed (sampled) via a normal (unit-length direction) vector...
- ...but we have a *virtual* cubemap, indexed by 2D (u,v) coordinates
- Solution: create a “Remap cubemap” to translate.
  - *given a normal, the Remap cubemap tells you the equivalent u / v coords at which to sample the *virtual* (unwrapped) cubemap.*



why the other formats don't work:

float4_32 you can't sample with bilinear interpolation

float2_32 cubemaps aren't supported by hardware yet

float2_16 has insufficient precision for large values (~3071)

HILO (2-ch, 16-bit uint) would work very well, but you can't render to it. (Works if you want to generate your Remap cubemap offline & save it to disk.)

BGRA won't work (with u in .rg and v in .ba) **with bilinear interpolation** because when you sample it, the 4 values are interpolated *in only 8 bits*, and reconstruction fails.

float4_16:

-you can't pack two floats into 4 halves, using the pack_2half / unpack_2half (bitwise) instructions, and read them back properly **using bilinear interpolation**. The reason is because these instructions store the upper & lower 16 bits, and when reading them back from the texture, the hardware fetches 4 float4_!6 values, interpolates them as halves, and THEN would do the bitwise reconstruction – but the data is bogus at that point.

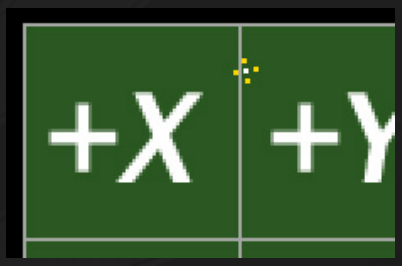
-see the hidden slides at the end of the Mad Mod Mike presentation for an acceptable way of packing the 2 values into 4 fp16 channels.

Reducing Aliasing

- As with any shadow map, use *Percentage-Closer Filtering* (bilinear interpolation on the VCM shadow test)
- Can also take multiple taps (samples) and average the results
  - Fastest to sample the Remap cube just *once*, then offset the resulting UV coords *in 2D*. *(see image)*

Danger: samples can dip into neighboring tiles!

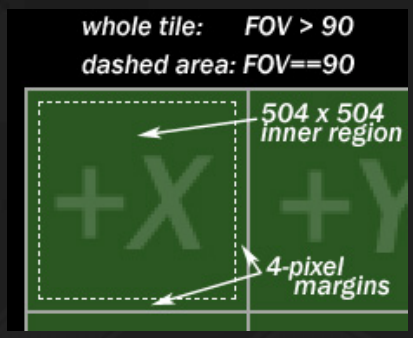
- Can be caused by single tap... (due to bilinear interpolation)
- Or by multiple taps (due to offsets)



*

Solution: Add a ‘margin’ to each tile

1. Render to the 6 viewports with a camera angle slightly > 90 degrees
2. Construct your Remap cubemap to compensate



- Please see the slides at the end of this presentation for full details on:
  - Determining the FOV (field of view) to use for your shadow-casting camera
  - How to dynamically construct the Remap cubemap

Hard Omni Shadows

- Fragment shader for receiving hard omnidirectional shadows:
  - About 6 instructions for 1 tap
  - About 16 instructions for 4 taps
    - [suitable for GeForce 7-series' breakfast]
  - *For fragment shader code and other details, please see the notes attached to this slide*

* Shader must know near & far clip plane values (n, f) for the shadow-casting passes.
Best if these are constants – can be folded into other computations
•Also needs to know VCM face size if you're doing multiple taps, for offsetting UV's
•Shader code follows...

```
// const inputs:
//   need to know the clip plane distances used
//   the in shadow-casting passes
const float n = 10.0;  // near clip plane distance
const float f = 1000.0; // far clip plane distance
const float VCM_facesize = 1024;

// derived values
// details: see "Efficient Omnidirectional Shadow Maps" paper
const float2 cLightQ = float2(
    -2*f*n/(f-n),       // GL: 2fn/(f-n)      DX: f/(f-n)
    (f+n)/(f-n)         // GL: (f+n)/(f-n)    DX: n*f/(f-n)
);
const float  c1 = 0.5*cLightQ.x;
const float  c2 = 0.5 + 0.5*cLightQ.y - zbias_const;
const float2 VCM_size = VCM_facesize * float2(3,2);
const float2 VCM_size_inv = 1.0/VCM_size;

// use Remap cube to get uv coords for sampling the VCM
// note: do all vector computations in world space! (ws)
float3 wsOmniLightVec = g_wsOmniLightPos - v2f.worldCoord.xyz;
float4 temp = f4texCUBE(RemapCube, wsOmniLightVec);
const float2 weights = float2( 8192.0, 8.0 );
temp *= weights.xyxy;          // unpack [.rg -> u coord, .ba -> v coord ]
float3 shadowCoord;
shadowCoord.x = temp.x+temp.y;
shadowCoord.y = temp.z+temp.w;

// use 'max' function to figure out which cube face
// our point projects to, then get Z for our pt.
// *along that face's axis*
// details: see "Efficient Omnidirectional Shadow Maps" paper
float3 LightVecAbs = abs(wsOmniLightVec);
float MA = max(max(LightVecAbs.x, LightVecAbs.y), LightVecAbs.z);
shadowCoord.z = c1/MA + c2;

option 1: single tap
half shadow_mult = h1texcompare2D(VirtCubeMap, shadowCoord );

option 2: multiple taps
// one possibility: use a rotated 4-queens-like pattern...
half shadow_mult = 0.25*(
   h1texcompare2D(VirtCubeMap, shadowCoord + float3(
                                                    VCM_size_inv.xy*float2(-0.4, 1.0), 0))
 + h1texcompare2D(VirtCubeMap, shadowCoord + float3(
                                                    VCM_size_inv.xy*float2(-1.0,-0.4), 0))
 + h1texcompare2D(VirtCubeMap, shadowCoord + float3(
                                                    VCM_size_inv.xy*float2( 0.4,-1.0), 0))
 + h1texcompare2D(VirtCubeMap, shadowCoord + float3(
                                                    VCM_size_inv.xy*float2( 1.0, 0.4), 0))
);
```

Other artifacts:

- At first, you might see large, dark square areas *(top image)* where points are in shadow that shouldn't be
- To fix, decrease shadow-casting-pass FOV slightly by a fudge factor
  - Ours was FOV *= 0.989
- Don't drop FOV *too* far, or you'll get shadow discontinuities where the projected 'virtual cubemap' face changes *(bottom image)*
- It can also help to tweak the polygon offset scale & bias for your shadow-casting pass

Near clip plane

- Caution: don't let shadow-caster objects get closer to the light than the near clip plane distance
- ...or this will happen →
- Solution: decrease the near clip plane distance. (Other tips in slide notes...)

Decreasing the near clip plane distance: Not too much, though! –

* depth values are stored hyperbolically...

* If only using depth16, try to keep near:far ratio at 1:100 (i.e. near=7, far=700) or less for optimal Z-precision.

Mad Mod Mike

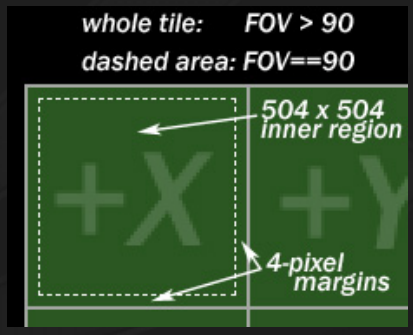
Hard Omni Shadows [HIDDEN SLIDE]

Adding margin pixels to each tile

Example: imagine you have 512x512 tiles in your VCM and you want a 4-pixel "margin" in each tile.

→ Let N = 512, b = 4

- Formula for the camera field of view (FOV) to use for rendering into the VCM:
  - fov = 2 * atan((N/2) / (N/2 – b))
  - yields an FOV that is >90 degrees
- Remap cube must compensate for this...
  - Each tile of the VCM now holds a > 90-degree FOV rendering...
  - So the Remap cube must only point us to samples within a *sub-region* of each tile on the VCM: the region holding the equivalent of a 90-degree-FOV rendering.



Hard Omni Shadows [HIDDEN SLIDE]

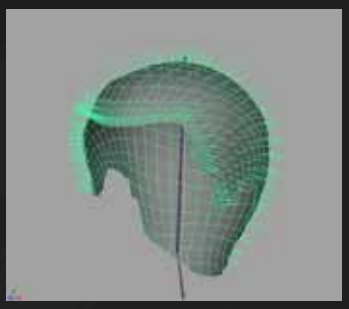
Generating the remap cubemap:

- create a float4_16 cubemap with 64x64 faces
- draw one fullscreen quad to each cube face
  - use triangle strip with indices = { 0, 1, 2, 3, 0, 1 } //draws 4 triangles; 2 get backface culled
- vertex shader just passes through the position & texcoord0 data (UV's).
- fragment shader just writes the packed UV coords directly to the RG + BA outputs.
  Packing code:
  - const float f = 8.0; // must match value in shadow-receiving shader!
  - float2 temp = saturate(v2f.c_texCoord.xy);
  - float2 msb = floor(temp * f) / f;
  - float2 lsb = frac(temp * f); //(temp - msb) * f;
  - out.xyzw = float4(msb.x, lsb.x, msb.y, lsb.y);
- The four XY's for each face:
  - (+)X: { (1,1) (-1,1) (1,-1) (-1,-1) } (-)X: { (-1,1) (1,1) (-1,-1) (1,-1) }
  - (+)Y: { (-1,-1) (1,-1) (-1,1) (1,1) } (-)Y: { (-1,1) (1,1) (-1,-1) (1,-1) }
  - (+)Z: { (-1,1) (-1,-1) (1,1) (1,-1) } (-)Z: { (1,1) (1,-1) (-1,1) (-1,-1) }
  - Note: these values are for GL; if using D3D, might need to flip sign on the Y coords!
- The four UV's for each face:
  - let i = { 0 for +X/-X, 1 for +Y/-Y, 2 for +Z/-Z }
  - let j = { 0 for +X/+Y/+Z, 1 for -X/-Y/-Z }
  - let u0 = (b) / N + i*0.33333 let v0 = (b) / N + j*0.5
  - let u1 = (N-b) / N + i*0.33333 let v1 = (N-b) / N + j*0.5
  - UV coords for the 4 verts = { (u0,v0), (u0, v1), (u1, v0), (u1, v1) }

Previous work: Nalu

- Hair procedurally grown from a mesh
- Hair Guides driven by a Particle System
- Physics in world-space
- Physics is blended with a static haircut
  - Static haircut stored in local-space
  - Allows basic shape control

Nalu's hair is not good enough



- Rigid scalp
- Lack of artistic control
  - Little control, zero styling
- Hair properties uniform across the scalp
  - Missed opportunity for optimization
- Bottom-line: not good enough

See Chapter 23 of *GPU Gems 2*
http://developer.nvidia.com/GPUGems2

Nalu's hair is not good enough

- Nalu's hair was mainly shaped by the physics simulation
  - Little control, zero styling

New demos, new needs

- Mad Mod Mike has a short beard
  - On a rather fleshy and malleable face
- Need stylized hair
  - Meaning not procedurally created
  - Luna: schoolgirl & hero
  - Mad Mod Mike also has a haircut
- Both demos require better
  - Styling, Flexibility & Control
  - Without a major performance penalty

New demos, new needs

- Mike's "biker" look required a beard

New demos, new needs

Mike's beard: hair on a non-rigid surface

- Mad Mod Mike uses blendshapes and skinning
  - Blendshapes = morphing
  - Skinning = mesh driven by a skeleton
  - Order = morph, then skin
- The skin surface motion can't be expressed by a single matrix
  - Too bad: we need the hair guides to follow the skin motion

Mike's beard: hair on a non-rigid surface

- Problem: Need the hair guides to be placed correctly in World Space
- No information to skin/blendshape the hair
  - No skin weights
  - No morph targets
- Solution: store the guides in a space that stays constant.
  - Texture Space!

Mike's beard: hair on a non-rigid surface

- Texture Space is normally used for bump mapping
  - In our case it could be called "surface space"
- There is one basis (matrix) for each point on the surface
  - You might already have it at the vertex level
- Each guide hair should have an associated basis
- At load time, transform the guide hairs into texture space for storage

Mike's beard: hair on a non-rigid surface

- At each frame all the basis are re-computed
  - The easy way is to encode a basis in each morph target, then proceed to morph & skin
- Use the basis as a matrix to transform the guides to World Space
- You've got hair on a malleable surface!

The need for stylized hair

- Underwater hair had a purpose
  - Looks cool
  - No haircut needed ☺
  - Can't do that forever…
- Styling means tools
  - A custom one was used for Dawn/dusk
    - Not integrated in art pipeline
  - "Shave and Haircut" is a Maya plug-in
    - Artists can actually use it
    - Hair becomes tweakable with a known set of tools

The need for stylized hair

- Shave & Haircut screenshot
  - By Joe Alter. (www.joealter.com)

The need for stylized hair

- Hair can be stylized down to the Control Point level
- Shave & Haircut creates NURBS curves
  - NURBS information is discarded
  - Simple lines would have been good enough
    - Bezier curves are generated by the application
- Control points are exported as a text file
- But that's not all…

Stylized hair - additional hair controls

- Information stored as textures applied to the scalp
  - Density
    - How many interpolated hair per “strand”
  - Color
  - Kill
    - Prevent interpolated hair from being drawn
      - Still being processed
  - Length
    - Additional length control
    - Great for procedural usage (random…)

Stylized hair - additional hair controls

- These controls allow a better utilization of the resources: Guide hairs & Vertices
- Use more hair where it matters
  - On the hairline (Luna)
  - Side burns, beard (Mad Mod Mike)
- Reduce it where it can’t be seen
  - Top of the head (Luna)
  - Under the helmet (Mad Mod Mike)

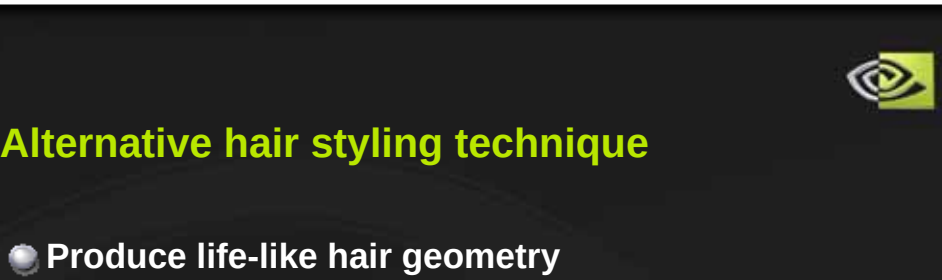
Stylized hair - additional hair controls

- Good hair for less!

123000 vertices	70000 vertices
(screenshot)	(screenshot)

The need for stylized hair

- Screenshots
  - MMM hair, Luna school, Luna hero
  - Kill, Density…
  - 6 total (3x2)



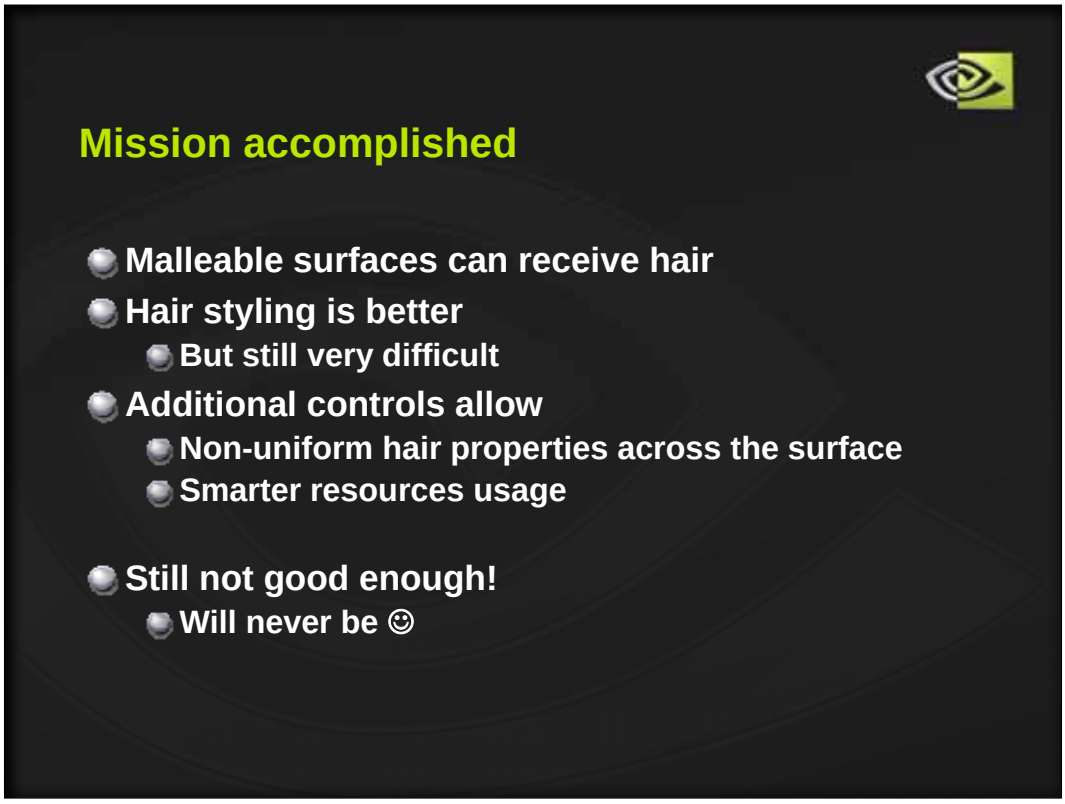
Alternative hair styling technique

- Produce life-like hair geometry
  - "Modeling Hair from Multiple Views"
  - Hair geometry (lines, curves) extracted from photos

Microsoft Research Asia and the Hong Kong University of Science & Technology
Eyal Ofek, Yichen Wei, Long Quan, Heung-Yeung Sum

Alternative hair styling technique

- Comes with per-vertex color too!
- (+) easy acquisition
- (+) non labor-intensive
- (+) fairly accurate
- (-) does not capture invisible parts
- (-) some hairstyles are difficult to mimic (curly...)


Mission accomplished
Malleable surfaces can receive hair
Hair styling is better
But still very difficult
Additional controls allow
Non-uniform hair properties across the surface
Smarter resources usage
Still not good enough!
Will never be ☺

Future work

- Styling remains very challenging
- Hair dynamics, evaluation, tessellation needs to be faster
  - CPU is too slow, move work to the GPU
- Better resources usage
  - Introduce better Level Of Details (LOD)
    - Based on distance, viewport
- Improve shading
- Introduce smarter physics (frictions…)
- And more…

References

- Lighting
  - Steve Marschner
- Shadowing
  - Opacity shadow maps
- Styling
  - Shave & Haircut
- Styling
  - Modeling hair from multiple views
- Books
  - GPU Gems 2
    http://developer.nvidia.com/GPUGems2/