Efficient Shader Tricks

That Will Impress Your Friends!

Chris Oat
3D Application Research Group
ATI Research, Inc.

Outline

- Multi-Layered Materials - *30 minutes*
  - Depth parallax
  - Light diffusion
- Ambient Aperture Lighting – *30 minutes*
  - Visibility aperture
  - Area light sources
  - Hard & Soft shadows

Part 1 of 2:

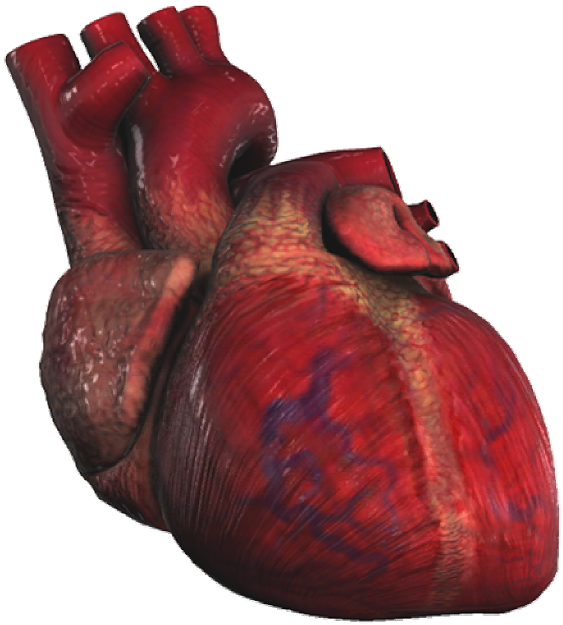
Multi-Layered Materials

Introduction

- Rendering semi-transparent, multi-layered materials
  - Surface with multiple texture layers
  - Texture layers are blended in some way
- Old way: Multi-texture blending
  - Lerping two textures looks flat
  - Layers are squashed together
- New way: Combination of techniques
  - Normal mapping
  - Transparency masking
  - Parallax offset mapping
  - Image filtering

Motivation

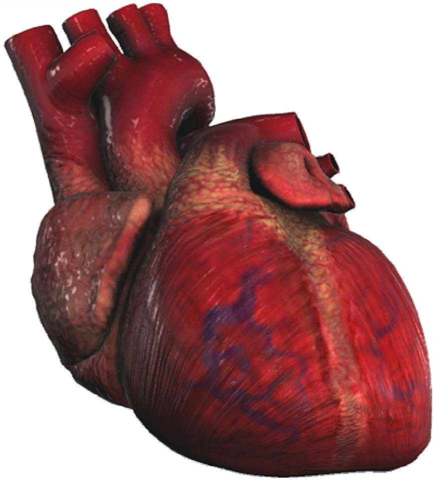
- Many real world surfaces have volumetric material properties:
  - Examples: Biological tissues, cloudy atmosphere, aliens, etc…
- These materials get their unique appearance from:
  - Multiple, semi-transparent “layers”
    - Each layer has some opacity value
  - Complex light interactions:
    - Light diffusion: blurry subsurface layers
  - Perspective:
    - Sub-layers have depth
- Traditionally you might use a volume renderer to achieve this look
  - Ray tracing isn’t practical for us
  - Choose the most important visual components and approximate them!



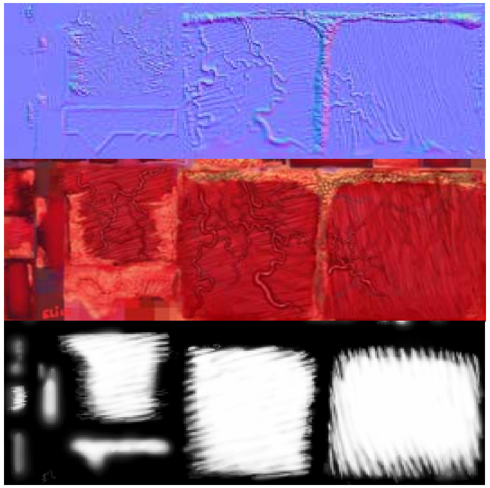
What are we trying to approximate?

- Volumetric material
  - Volume approximated with multiple discreet layers
  - Layers are semi-transparent
  - Layers reflect, absorb, and transmit light
- Visually important properties:
  - Inter-layer occlusion
    - Layers store opacity in alpha
  - Depth parallax
    - Parallax due to layer depth or thickness
  - Light diffusion
    - Light scatters between layers
- How can we achieve the look and still be fast?
  - Alpha blend/composite
  - Parallax offsetting
  - Blurring

Two layer example: Human Heart



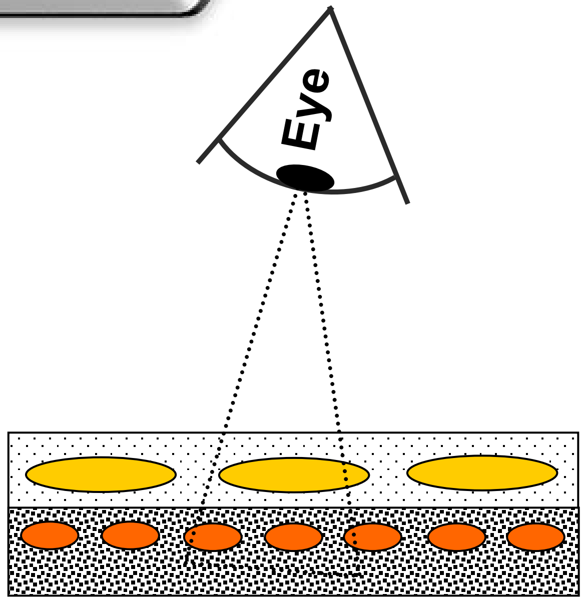
Outer Layer: Normal, Base, Opacity maps

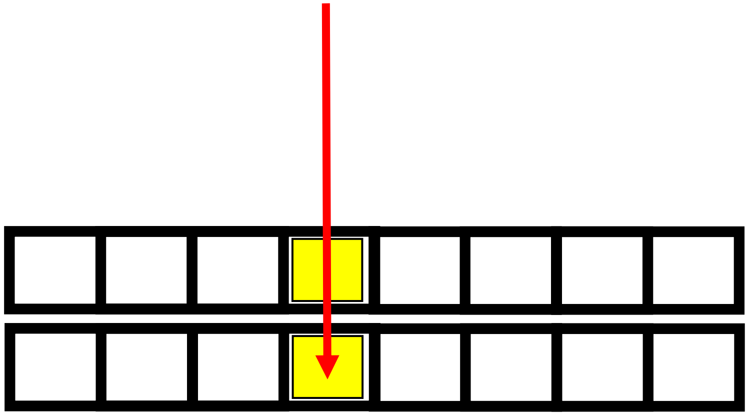


Inner Layer: Base map

- Each layer stored as a texture map
- Opaque texels occlude the texels below them
  - ***LERP*** layers based on alpha
  - This gets layer occlusion working
  - But results in ***flat*** looking composite
- In order to give the impression of layer depth, a form of ***parallax mapping*** is needed!
  - Texture Coordinates for inner layer are computed in the shader
    - Based on **viewing angle** and **layer depth**

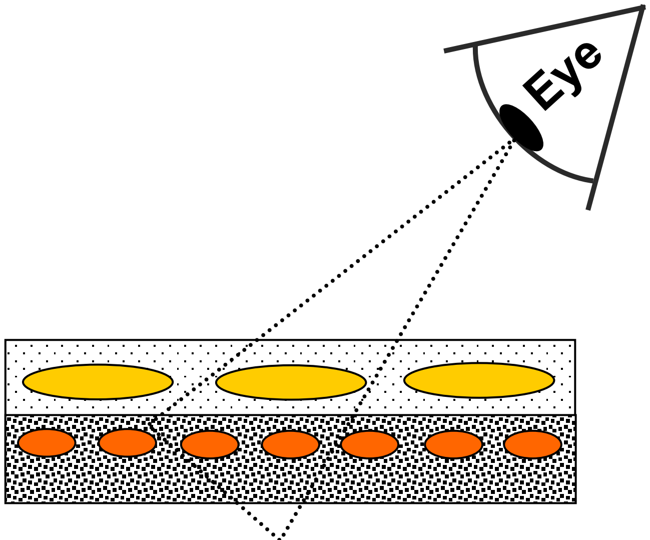
Multi-layer depth parallax

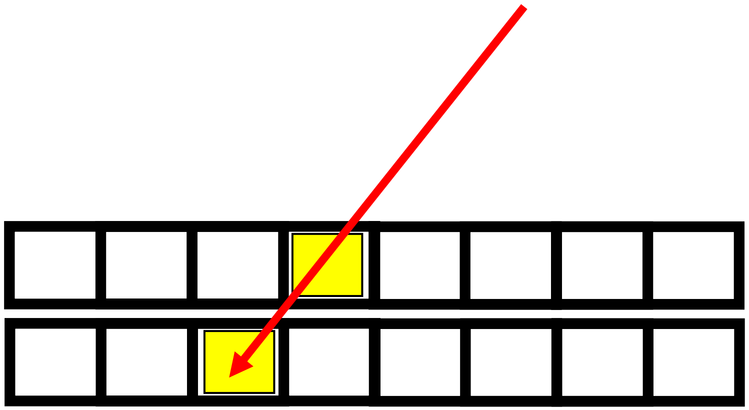




- Make the material look volumetric
- Depth parallax
  - Shift in apparent position due to change in view
  - Inner layer shifts with respect to outer layer
  - Shift is more pronounced as depth increases
- Can’t use surface layer’s UV coordinate to sample inner layer’s texture

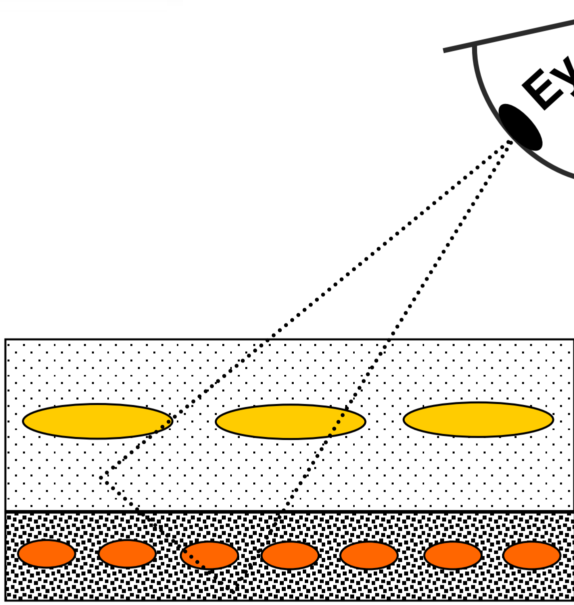
Multi-layer depth parallax

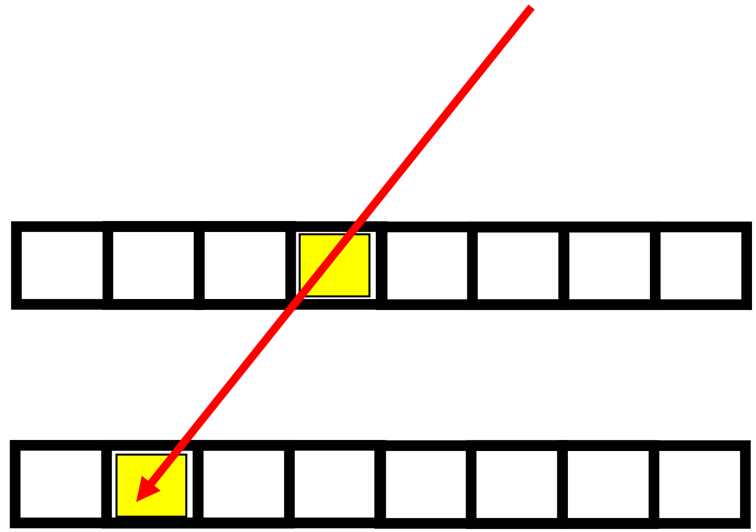




- Make the material look volumetric
- Depth parallax
  - Shift in apparent position due to change in view
  - Inner layer shifts with respect to outer layer
  - Shift is more pronounced as depth increases
- Can’t use surface layer’s UV coordinate to sample inner layer’s texture

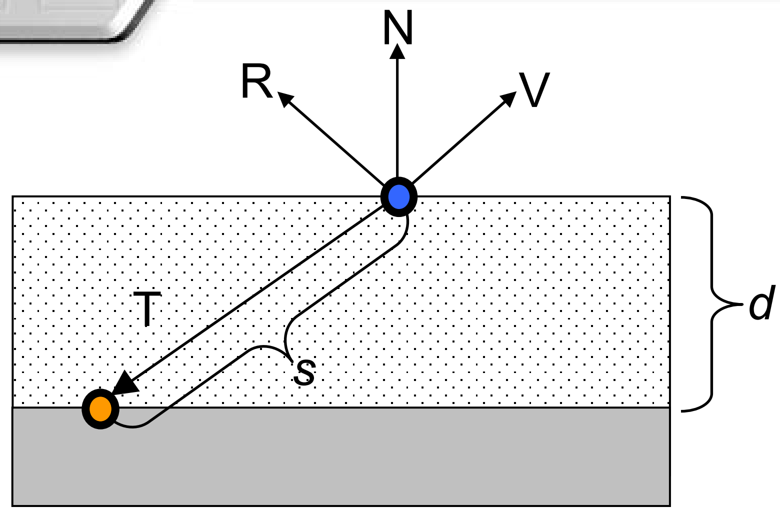
Multi-layer depth parallax





- Make the material look volumetric
- Depth parallax
  - Shift in apparent position due to change in view
  - Inner layer shifts with respect to outer layer
  - Shift is more pronounced as depth increases
- Can't use surface layer's UV coordinate to sample inner layer's texture

Inner layer's texture coordinates



= Outer UV coordinate: <u,v>

= Inner UV coordinate: <u',v'>

$$\vec{R} = -\vec{V} - 2*dot(-\vec{V},\vec{N})*\vec{N}$$

$$\vec{T} = \langle \vec{R}_x, \vec{R}_y, -\vec{R}_z \rangle$$

$$s = d / \left| \vec{T}_z \right|$$

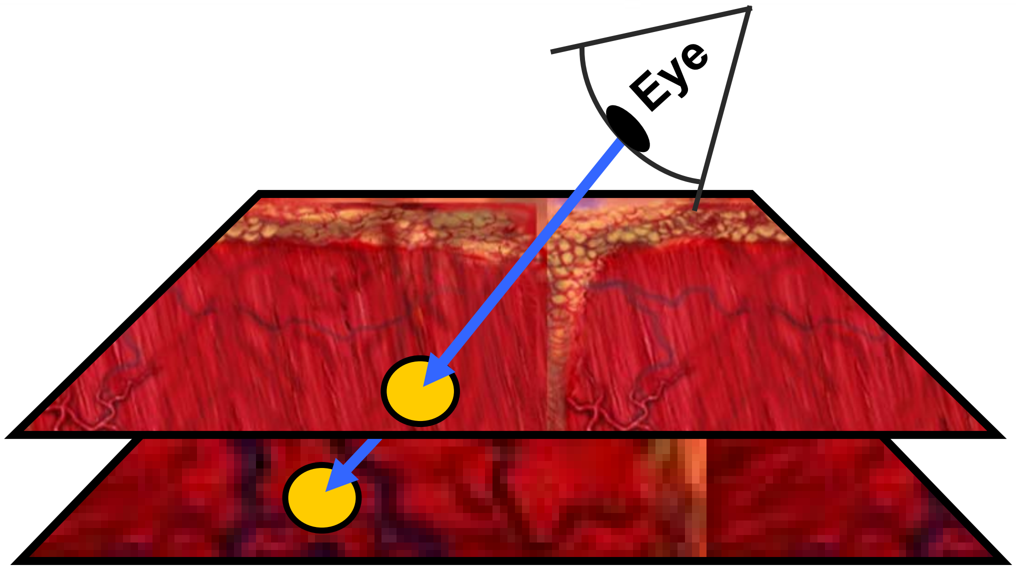
$$\langle u', v' \rangle = \langle u, v \rangle + s \langle \vec{T}_x, \vec{T}_y \rangle$$

- Layers are assumed to be parallel in tangent space
  - Layer depth **d** is homogeneous for a given layer
- Find inner layer's texture coordinate
  1. Find view vector = **V**
  2. Reflect **V** about **N**ormal (from normal map) = **R**
  3. Reflect **R** about surface plane = transmission vector **T**
     - In tangent space, we simply negate **R.z** component
  4. Find distance **s** along **T** to inner layer: Function of distance **d** between layers
  5. Use **T** and **s** this to find inner layer's texture coodinate

Parallax offset

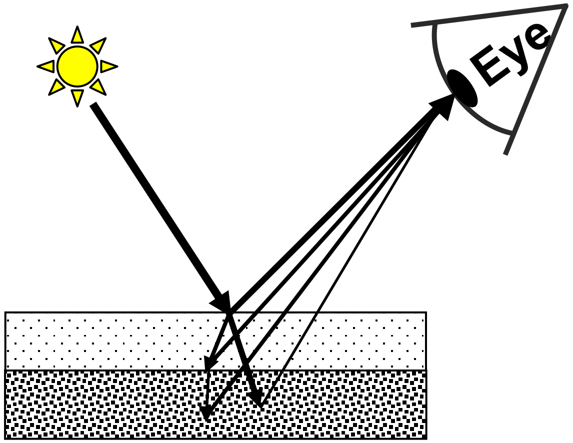
```
// Compute inner layer's texture coordinate and transmission depth
// vTexCoord: Outer layer's texture coordinate
// vViewTS: View vector in tangent space
// vNormalTS: Normal in tangent space (sampled normal map)
// fLayerThickness: Distance from outer layer to inner layer
float3 ParallaxOffsetAndDepth ( float2 vTexCoord, float3 vViewTS,
                                float3 vNormalTS, float fLayerThickness )
{
        // Tangent space reflection vector
        float3 vReflectionTS = reflect( -vViewTS, vNormalTS );

        // Tangent space transmission vector (reflect about surface plane)
        float3 vTransTS = float3( vReflectionTS.xy, -vReflectionTS.z );

        // Distance along transmission vector to intersect inner layer
        float fTransDist = fLayerThickness / abs(vTransTS.z);

        // Texel size: Hard coded for 1024x1024 texture size
        float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

        // Inner layer's texture coordinate due to parallax
        float2 vOffset = vTexelSize * fTransDist * vTransTS.xy;
        float2 vOffsetTexCoord = vTexCoord + vOffset;

        // Return offset texture coordinate in xy and transmission dist in z
        return float3( vOffsetTexCoord, fTransDist );
}
```

Parallax creates the illusion of depth



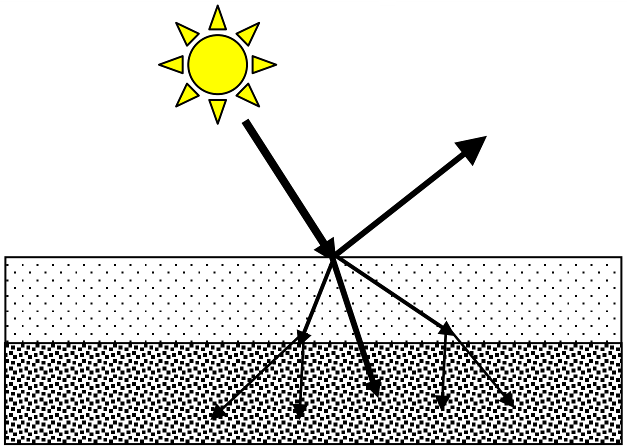
- The offset texture coordinate is used for sampling from the inner layer's texture
- This creates the illusion of depth or volume even though the surface geometry is flat
- We still need a way to light the inner layer convincingly…

Multi-layer light diffusion



- Light scatters as it enters a material
  1. Light reaches surface
  2. Some reflects back to eye
  3. Some scatters further into the material
  4. GOTO 2
- Physically based models for scattering are slow
- Get the look without doing the math!
  - Light reflected back to eye from surface
  - Light scatters on its way in
  - Light scatters on its way out

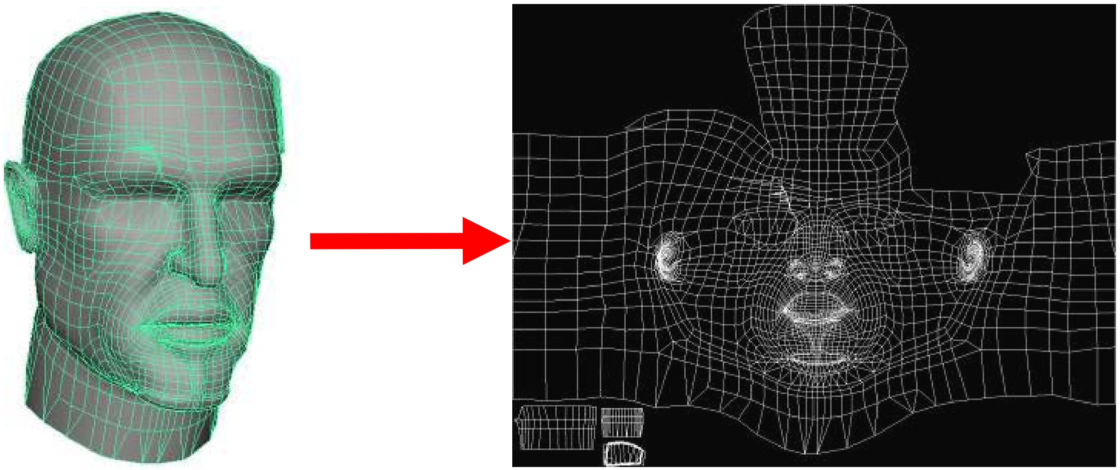
Getting the look: Incoming light



- Surface layer lit as usual (N.L)
  - Accounts for light that doesn't enter material
- Inner layer is more evenly lit
  - Transmitted light scatters onto layer from many directions

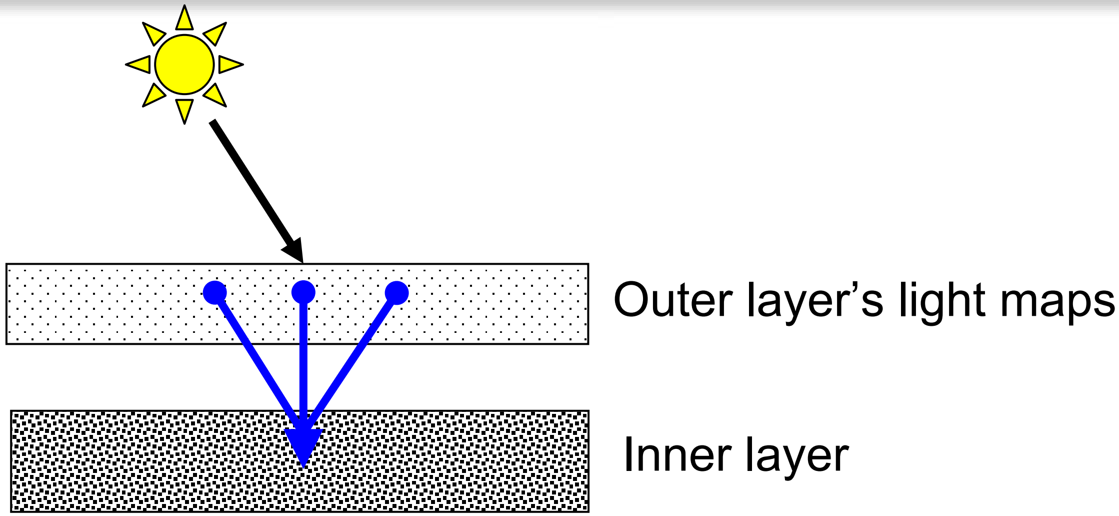
- Texture space lighting
  - Render diffuse lighting into an off-screen texture using texture coordinates as positions
  - Acts like a dynamic light map for the outer layer

Getting the look: Incoming light



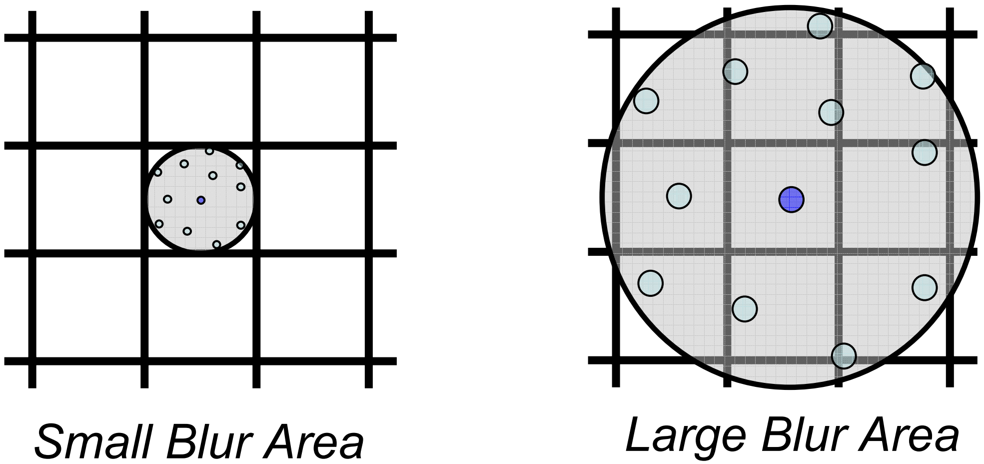
- Texture space lighting
  - Render diffuse lighting into an off-screen texture
  - Light as a 3D model *but draw into texture*
  - Vertex shader outputs texture coordinates as projected “positions” then the rasterizer does the unwrap
  - Vertex shader computes light vectors based on 3D position and interpolates
  - **This is a light map for the outer layer**
  - HLSL implementation online: Dave Gosselin’s Skin Rendering Slides
    - www.ati.com/developer/gdc/D3DTutorial_Skin_Rendering.pdf

Getting the look: Incoming light



- For the inner layer's lighting, use a blurred version of the outer layer's light map
- This gives us smooth, diffused lighting on the inner layer
- The amount of blurring depends on the thickness of the outer layer
  - Use a variable sized blur kernel

Poisson disc kernel

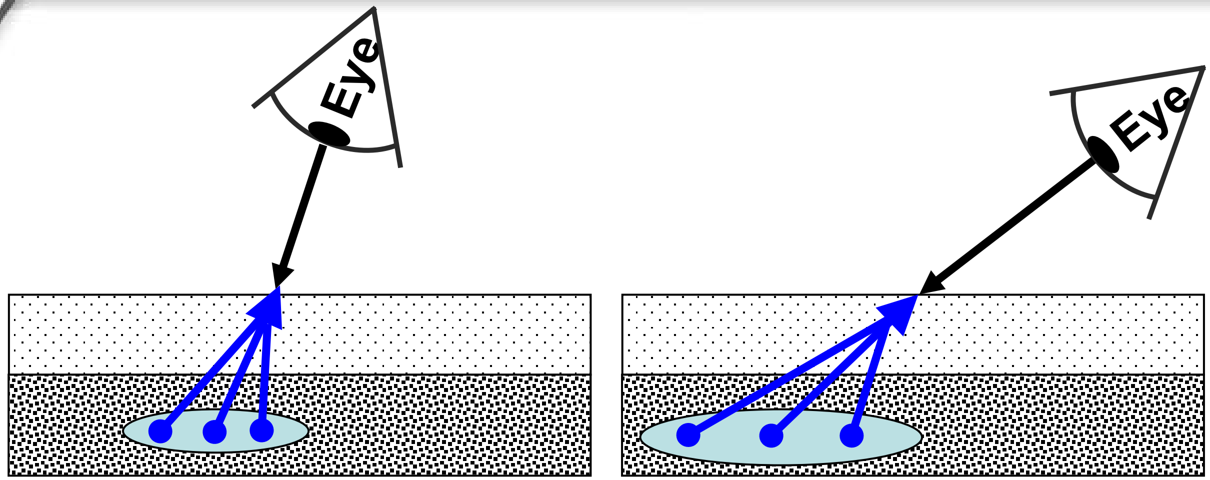


- A Poisson disc kernel is ideal since it can be resized dynamically based on the amount of light diffusion you want
- Kernel takes a fixed number of taps from source texture
- Taps are distributed randomly on a unit disc (Poisson distribution)
- Disc size can be scaled on a per-pixel basis for more or less blurring
- Our disc's radius is based on layer thickness
  - Thicker layer results in more blurring

Growable Poisson disc

```
// Growable Poisson disc (13 samples)
// tSource: Source texture sampler
// vTexCoord: Texture space location of disc's center
// fRadius: Radius if kernel (in texel units)
float3 PoissonFilter ( sampler tSource, float2 vTexCoord, float fRadius )
{
   // Hard coded texel size: Assumes 1024x1024 source texture
   float2 vTexelSize = float2( 1.0/1024.0, 1.0/1024.0 );

   // Tap locations for unit disc
   float2 vTaps[12] = {float2(-0.326212,-0.40581),float2(-0.840144,-0.07358),
                       float2(-0.695914,0.457137),float2(-0.203345,0.620716),
                       float2(0.96234,-0.194983),float2(0.473434,-0.480026),
                       float2(0.519456,0.767022),float2(0.185461,-0.893124),
                       float2(0.507431,0.064425),float2(0.89642,0.412458),
                       float2(-0.32194,-0.932615),float2(-0.791559,-0.59771)};

   // Take a sample at the disc's center
   float3 cSampleAccum = tex2D( tSource, vTexCoord );

   // Take 12 samples in disc
   for ( int nTapIndex = 0; nTapIndex < 12; nTapIndex++ )
   {
      // Compute new texture coord inside disc
      float2 vTapCoord = vTexCoord + vTexelSize * vTaps[nTapIndex] * fRadius;

      // Accumulate samples
      cSampleAccum += tex2D( tSource, vTapCoord );
   }

   return cSampleAccum / 13.0; // Return average
}
```

Getting the look: Outgoing lighting

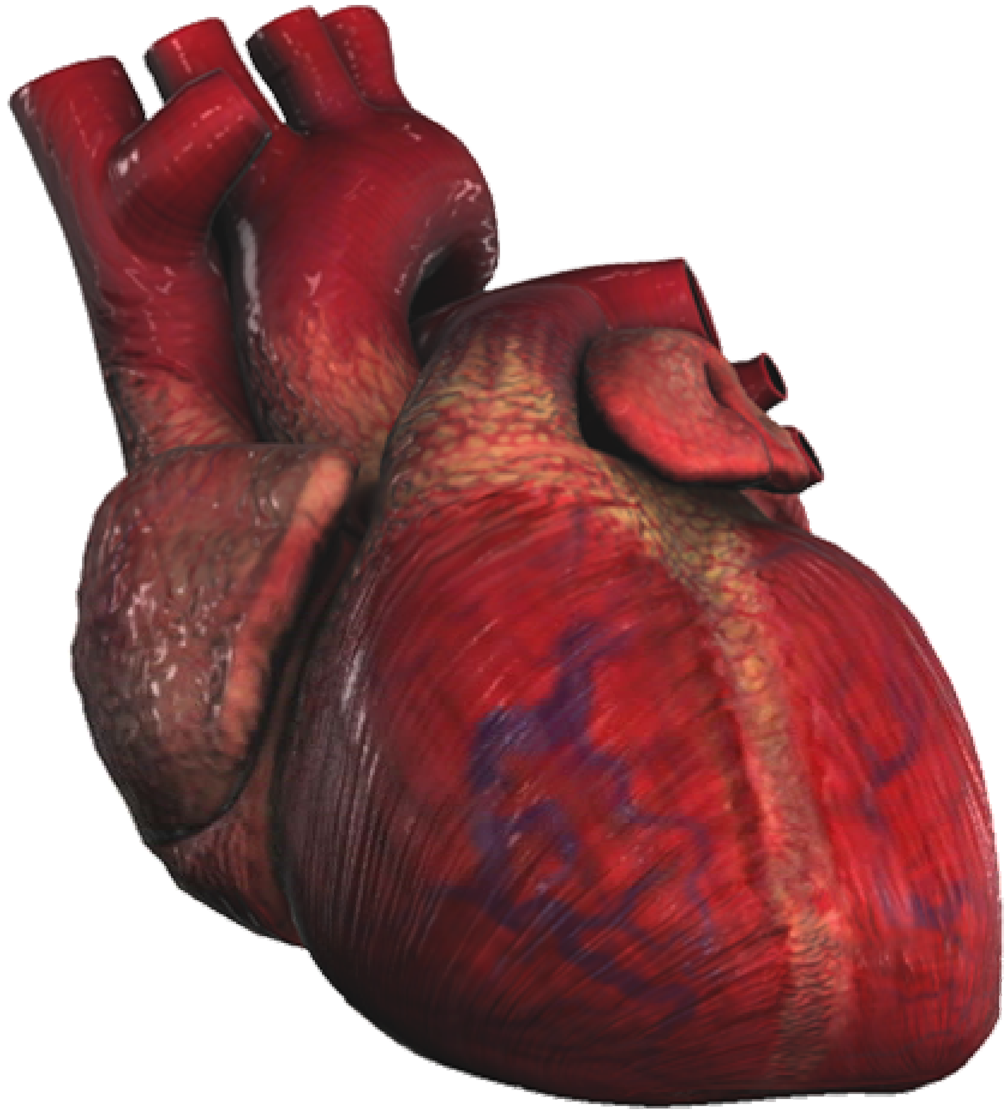


- The blurred light map approximates light scattering as it enters the material
- Light also scatter as on it’s way back out of the material
- This has the effect of a the Inner layer’s base map appearing blurry
- Use Growable Poisson Disc filter for sampling inner layer’s base map
  - This time ***kernel size depends on transmission distance*** through material
  - Not just layer thickness
  - Kernel is ***centered around*** the inner layer’s ***parallax offset texture coordinate***
- Inner layer now looks blurry
  - The more material you’re looking through, the blurrier it will look

Putting it all together

```
// Sample from outer layer's base map and light map textures
float3 cOuterDiffuse = tex2D(tLightMap, i.vTexCoord);
float4 cOuterBase = tex2D(tOuterBaseMap, i.vTexCoord); // Opacity in alpha channel


// Compute parallax offset texture coordinate for sampling from inner layer textures
// returns UV coord in X and Y and transmission distance in Z
float3 vOffsetAndDepth = ParallaxOffsetAndDepth(i.vTexCoord, vViewTS,
                                                vNormalTS, fLayerThicknes);


// Poisson disc filtering: blurry light map (blur size based on layer thickness)
float3 cInnerDiffuse = PoissonFilter(tLightMap, vOffsetAndDepth.xy, fLayerThickness);


// Poisson disc filtering: blurry base map (blur size based on transmission distance)
float3 cInnerBase = PoissonFilter(tInnerBaseMap, vOffsetAndDepth.xy, vOffsetAndDepth.z);


// Compute N.V for additional compositing factor (prefer outer layer at grazing angles)
float fNdotV = saturate( dot(vNormalTS, vViewTS) );


// Lerp based on opacity and N.V (N.V prevents artifacts when view is very edge on)
float3 cOut = lerp(cOuterBase.rgb*cOuterDiffuse.rgb,
                   cInnerBase.rgb*cInnerDiffuse.rgb,
                   cOuterBase.a * fNdotV);
```

Demo: Beating human heart



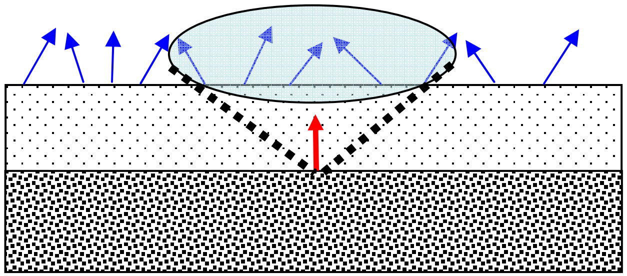
Taking it to the next level

- Increase complexity
  - **More than two layers**
  - Use the same techniques I've shown here
  - This is more expensive, but looks really good!

- Improve quality
  - **Inter-layer shadowing**
  - Scale light map samples by their corresponding opacities from base map
  - Keeps light from passing through opaque regions
  - In practice this doesn't make a huge difference, as you can't see what's below an opaque region unless you're looking at it very edge on on
  - More important when you're using many layers or a very deep/thick material

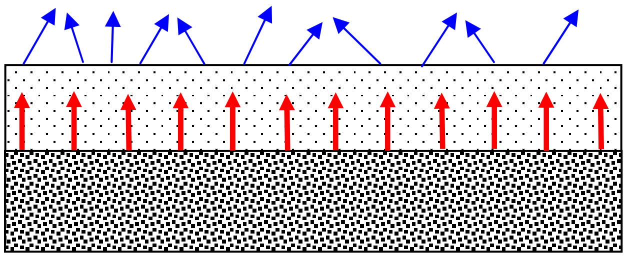
- Improve performance
  - **Eliminate the off screen render targets**
  - Two suggestions for eliminating renderable texture (light map)
  - See next slide… →

Lighting optimization 1



- Outer layer lit as usual
  - Use a **normal map** for high **frequency surface detail**
- Instead of using a blurred light map for the Inner layer's lighting
  - Use a ***modified Poisson Disc*** filter kernel
  - Take **multiple samples** from outer layer's **Normal Map**
    - Compute **N.L** for **each sample**
    - **Average** all the **N.L** computations
- Eliminates the need for a renderable texture!

Lighting optimization 2



- Outer layer lit as usual
  - Use a **normal map** for high **frequency surface detail**
- Instead of using a blurred light map or multiple normal map samples for the Inner layer's lighting:
  - Use the **geometric normal** for computing **N.L**
  - **Smoother, lower frequency lighting**
  - In practice, this works quite well and it's a lot faster
- Eliminates the need for a renderable texture!
- Reduces texture bandwidth requirements by eliminating one of the Poisson disc filtering steps

Part 2 of 2:

Ambient Aperture Lighting

What is Ambient Aperture lighting?

- Shading model that uses apertures to approximate a visibility function
  - ***Precomputed*** visibility
  - **Dynamic** spherical area light sources
  - Dynamic point light sources
  - Hard & Soft shadows
- Similar to horizon mapping, but allows for area light sources
- The "ambient" comes from the fact that we use a modified ambient occlusion calculation to find an aperture of average visibility
- Developed with ***Terrain rendering*** in mind but can be used for other things as well…

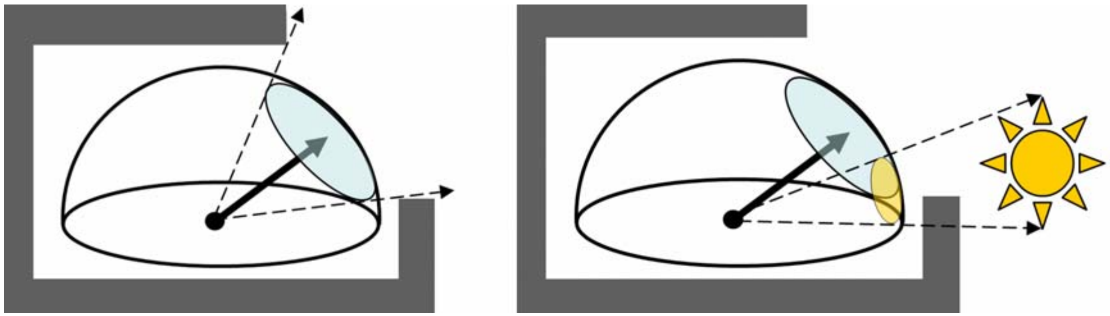
What are the applications?

- Non-deformable models
  - Terrains
  - Static scene elements
    - Buildings
    - Statues

- Dynamic spherical area light sources
  - Hard & Soft shadows

- Applications where performance is critical and rendering must still look realistic (but not necessarily physically correct)

How does it work?



- Ambient aperture lighting works in 2 stages
- ***Precomputation Stage***
  - Visibility function is computed at every point on mesh
    - Per-vertex or per-pixel
  - Visibility function is stored using a spherical cap
  - ***Spherical cap*** stores an average, contiguous region of visibility
    - A spherical cap is a portion of a sphere cut off by a plane (a hemisphere itself is a spherical cap)
- ***Rendering Stage***
  - Spherical cap ***acts as an aperture***
  - Aperture is used to restrict incoming light so that it only enters the from visible (un-occluded) directions
  - Area light sources are projected onto the hemisphere and are clipped against the aperture
  - This determines how much of their light passes through the aperture

Precomputation stage

- The precomputation stage can be thought of as a two step process:

- **Step 1:**
  - Find visible area
    - Area of hemisphere that is unoccluded by the surrounding scene
  - This serves as the area of our aperture/spherical cap

- **Step 2:**
  - Find average direction of visibility
    - Just like finding a bent normal
    - Average of all un-occluded rays fired from a given point
  - This serves as the orientation of our aperture/spherical cap

Visible area (aperture size)

$$VisibleArea(x) = 2\pi \int_{\Omega} V(x,\omega)\,d\omega$$

- For every point on the mesh (vertex/pixel):
  - Cast a bunch of rays
  - Determine what percentage of rays reach infinity (un-occluded)
    - Gives you a percentage of visibility
    - Like finding ambient occlusion but you don't weight samples by cos(theta)
  - Multiply by 2PI (area of unit hemisphere)
    - Gives you an average area of visibility
- The average area of visibility is used as our aperture size.
  - We are making the assumption that the visible area on the hemisphere forms a contiguous circular region (i.e. a spherical cap)
- We'll need the arc length of the cap's radius at render time:
  - arc length of radius = **acos(-area/2PI + 1)**
- Single float value, stored per vertex/pixel

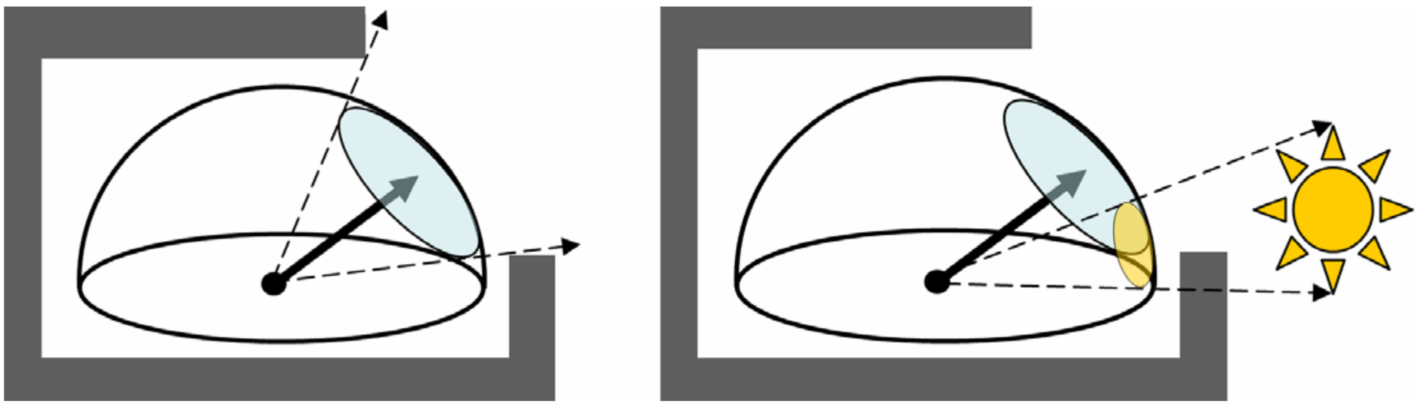
Visible direction (aperture orientation)

$$VisibleDir(x) = \int_{\Omega} V(x,\omega)\,\omega\, d\omega$$

- For every point on the mesh (vertex/pixel):
  - Cast a bunch of rays
  - Determine average direction for which rays reach infinity (un-occluded)
    - This is frequently referred to as a ***bent normal***

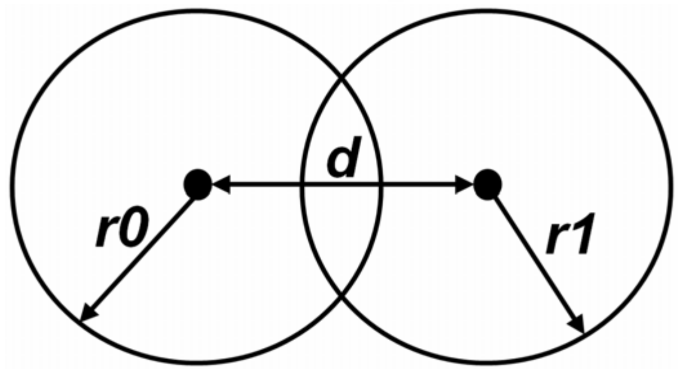
- This gives you the average direction of visibility
- Use this for your aperture’s orientation
- A float3 per vertex/pixel

How to render using apertures?



- Project spherical area light source onto hemisphere
- Projected area light source covers some area of the hemisphere
  - Projected sphere forms a spherical cap, just like our aperture
- Find the intersection of the projected light's spherical cap and the aperture's spherical cap
- Once the area of intersection is found, we know the portion of the light source that passes through the aperture

Finding area of intersection



- Intersection area of two spherical caps is a function of the arc lengths of their radii (r0, r1) **and** the distance between their centroids (d)

- If **d >= r0 +r1**
  - ***No intersection***
  - Thus area is 0

- If **min(r0,r1) <= max(r0,r1)-d**
  - ***Fully intersected***
  - Use the area of the smallest cap
  - Area of cap: $(2\pi - 2\pi\cos(\min(r1, r0)))$

- Otherwise…

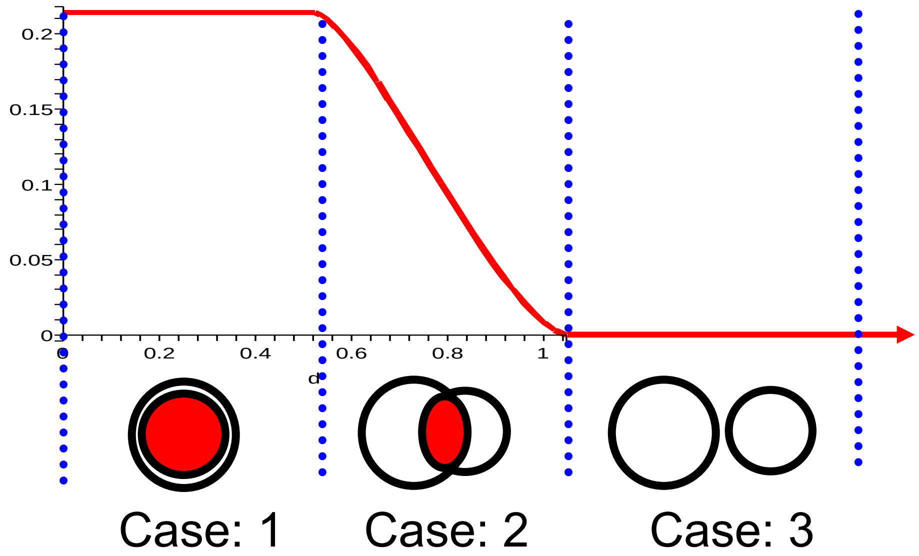
Spherical cap intersection

$$-2\arccos\left(\frac{\cos(d)-\cos(r0)\cos(r1)}{\sin(r0)\sin(r1)}\right)-$$
$$2\pi\cos(r1)+2\cos(r0)\arccos\left(\frac{-\cos(r1)+\cos(d)\cos(r0)}{\sin(d)\sin(r0)}\right)-$$
$$2\pi\cos(r1)+2\cos(r1)\arccos\left(\frac{-\cos(r0)+\cos(d)\cos(r1)}{\sin(d)\sin(r1)}\right)+2\pi$$

- **Oh no!**
- After all our simplifications, we’re left with this monster to solve!
- Let’s take a closer look at the intersection area function…

*Simplified form of intersection area function given by [Tovchigrechko]

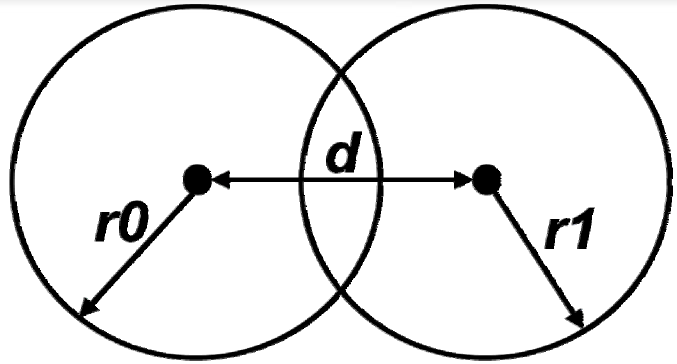
Intersection function



- Case 1 and 3 handled by our early outs
  - ***Case 1*** : *Full intersection*
  - **Case 3** : *No intersection*

- Intersection area decreases as caps move away from each other
- Rate of falloff is inversely proportional to the area of the two spherical caps
  - Bigger caps have slower falloff
  - Smaller caps have faster falloff

Smoothstep saves the day



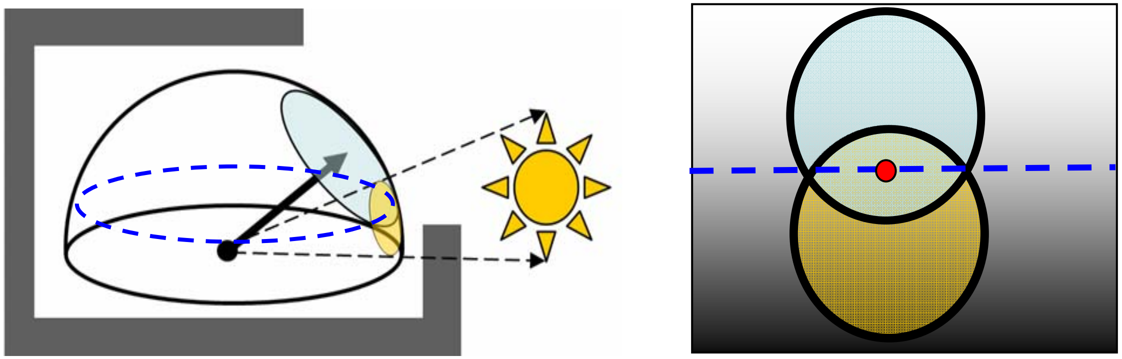
$$\underbrace{(2\pi - 2\pi\cos(\min(r1, r0)))}_{\text{Area of smallest spherical cap}} \ \text{smoothstep}\left(0,1,1-\frac{d-|r0-r1|}{r0+r1-|r0-r1|}\right)$$

- Case 1: **Full intersection**
    - Smoothstep returns 1
- Case 2: **Partial intersection**
    - Smoothstep returns smooth falloff (depending on amount of overlap)
    - Gives a smooth transition from full intersection to no intersection
- Case 3: **No intersection**
    - Smoothstep returns 0

Intersection area approximation

```
// Approximate the are of intersection of two spherical caps
// fRadius0 : First cap's radius (arc length in radians)
// fRadius1 : Second caps' radius (in radians)
// fDist : Distance between caps (radians between centers of caps)
float SphericalCapIntersectionAreaFast ( float fRadius0, float fRadius1, float fDist )
{
   float fArea;

   if ( fDist  <= max(fRadius0, fRadius1) - min(fRadius0, fRadius1) )
   {
      // One cap in completely inside the other
      fArea = 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }
   else if ( fDist >= fRadius0 + fRadius1 )
   {
      // No intersection exists
      fArea = 0;
   }
   else
   {
      float fDiff = abs(fRadius0 - fRadius1);
      fArea = smoothstep(0.0,
                         1.0,
                         1.0-saturate((fDist-fDiff)/(fRadius0+fRadius1-fDiff)));
      fArea *= 6.283185308 - 6.283185308 * cos( min(fRadius0,fRadius1) );
   }
   return fArea;
}
```

Don’t forget about our friend Lambert



- Reflectance is determined by the area of intersection ***and*** Lambert’s Cosine Law
    - Find a vector to the **centroid** for the region of intersection
    - This is estimated by averaging the aperture’s vector and the light’s vector
    - Scale the intersection area by N.Vcentroid
        - IntersectionArea * saturate(N.Vcentroid)
    - This provides a Lambertian falloff as the light source approaches the horizon

- Just another approximation on top of all the others we’re making ☺
- Assumes the area above intersection’s centroid is about the same as the area below the intersection’s centroid
    - ***Negative error above*** the centroid **cancels** the ***positive error below*** the centroid

Ambient light

- We now have a function for finding direct lighting from area light sources, but we’d like to incorporate some form of ambient light to account for light scattered in from the sky

- Treat sky as if it were a giant area light behind the sun:
  - Compute area light/aperture intersection
  - If area of intersection is less that area of aperture, fill the missing space with indirect “ambient light”
    - For a terrain, use the average sky color (lowest MIP level of sky dome?)
      - Blue during the day
      - Redish-pink at sun set
      - Black at night

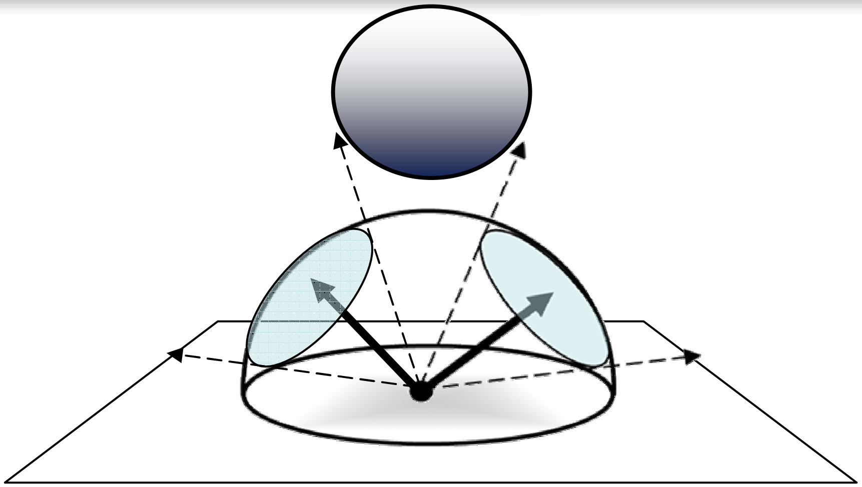
- Works better than the standard constant ambient term
  - Only applies to areas that aren’t being lit directly and aren’t totally occluded from the outside world

Demo: Terrain

What are the benefits of this technique?

- Area light sources
  - Better than N.L with point light sources
  - Hard shadows for small area light sources
  - Soft shadows for large area lights sources
- Small storage requirements
  - Just 4 floats per-vertex or per-pixel
  - Or 3 floats if you store aperture orientation in tangent space and derive z component in your shader
- Doesn’t require additional transforms
  - Shadow maps require transforming model one or more extra times
- Very cheap to compute
  - Just a handful of vertex shader or pixel shader instructions
  - Gives pleasing results

What are the potential downfalls?



- Assumes visible region is contiguous and circular
  - Sphere over plane (see example)
  - **Which way should visibility aperture point?**
  - Visible region is a band around the horizon, this is poorly approximated by a spherical cap
- Multiple light sources don’t occlude each other
  - You’d have to compute area of overlap to make sure you don’t over light
  - In practice this isn’t necessarily a huge issue (people expect 2 light sources to make things twice as bright)
- Assumes non-local light sources
  - Light source can’t be between point being shaded and it’s blocker
  - Results in incorrect shadowing
- Works well with terrains
  - Terrains typically have nicely behaving visibility functions
  - Occlusion is a band along the horizon
  - Visibility region is generally a contiguous, circular region somewhere in the sky

Taking it to the next level

- Multiple visibility apertures
  - Fixes case where you're in a room with multiple windows
  - Multiple contiguous regions of visibility

- Occlusion "anti-apertures"
  - Contiguous regions of occlusion
  - Fixes sphere over plane case
  - Spherical cap intersection gives amount of occlusion rather than amount of light

Preprocessor optimizations

- Speed up or even eliminate the preprocessing step
  - Exploit the fact that Aperture can be computed using modified ambient occlusion and bent normal preprocessors
- Google for:
  - **GPU accelerated ambient occlusion**
    - Improve preprocessing speed
    - D3DX provides a GPU accelerated SH direct lighting function
      - First coefficient can be used to approximate visible area
      - Next 3 coefficients approximate average visible direction
  - **Dynamic ambient occlusion**
    - Eliminate the need to preprocess
    - Allows for deformable meshes
    - Probably isn't realistic for your performance needs

Conclusion

- Two techniques that use various mathematical ***approximations*** and make ***simplifying assumptions*** to enable us to render ***expensive looking*** graphics

- Multi-Layered Materials
  - Depth parallax
  - Light diffusion

- Ambient Aperture Lighting
  - Area light sources
  - Hard & Soft shadows

Thank you!

I would like to thank **Pedro Sander** for his thoughtful discussion and collaboration on the Ambient Aperture work.

Thanks to **Eli Turner** for providing the human heart model and textures.

References

- Kaneko, T. et al., "Detailed Shape Representation with Parallax Mapping," ICAT, 2001.
- Max, N. L. *Horizon Mapping: Shadows for Bump-mapped Surfaces. The Visual Computer 4,* 2 (July 1988), 109-117.
- Oat, Chris, "Rendering Semitransparent Layered Media," *ShaderX 3: Advanced Rendering with DirectX and OpenGL,* Charles River Media, 2005.
- Pharr, Matt, "Layered Media for Subsurface Shaders," *Advanced Renderman*, SIGGRAPH 2002 Course Notes.
- Tovchigrechko, A. and Vakser, I.A. 2001. How common is the funnel-like energy landscape in protein-protein interactions? *Protein Sci.* 10:1572-1583.
- Welsh, Terry, "Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces," Infiniscape Tech Report, 2003.

Questions?

Chris Oat

coat@ati.com

These slides are available for download:

www.ati.com/developer

We're hiring!

3D Applications **R**esearch **G**roup

Demo Team

Research Team

Tools team

Visit our web site:

http://www.ati.com/companyinfo/careers/

Mention 3DARG and the team you're interested in joining!