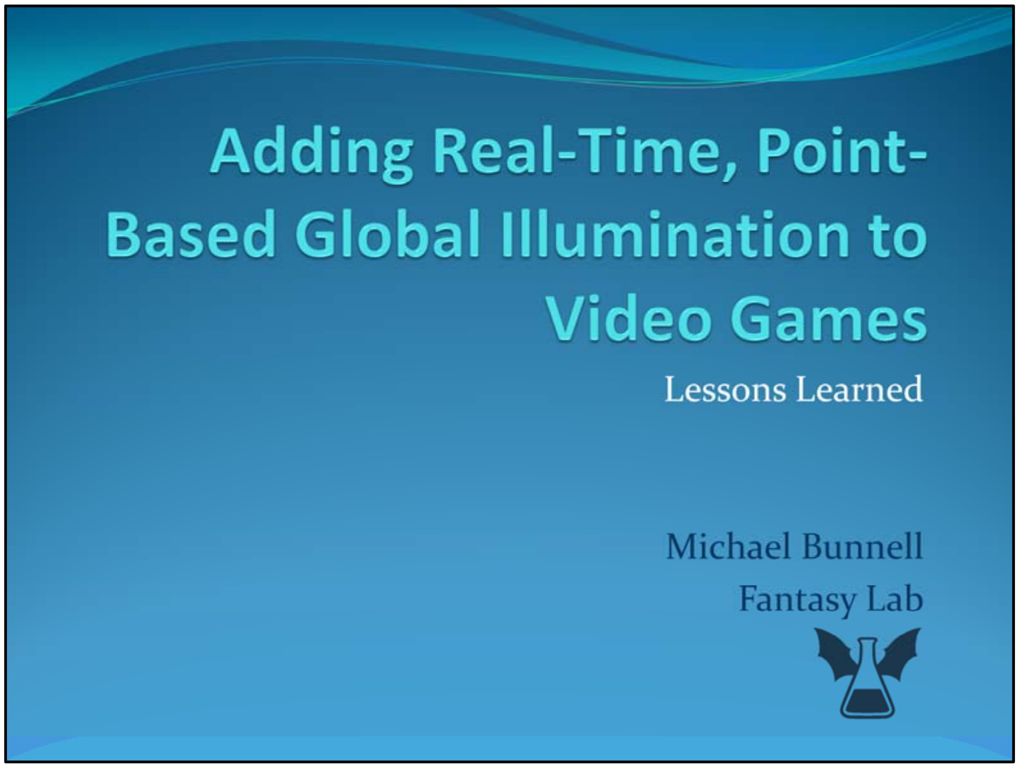
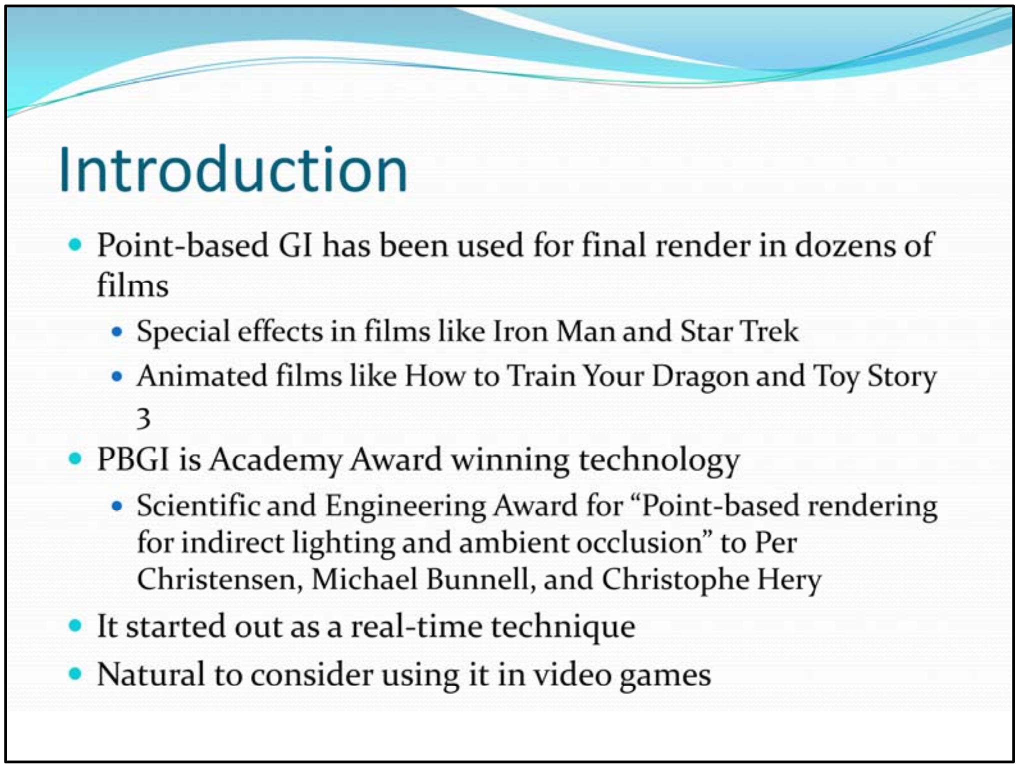


Adding Real-Time, Point-Based Global Illumination to Video Games

Lessons Learned

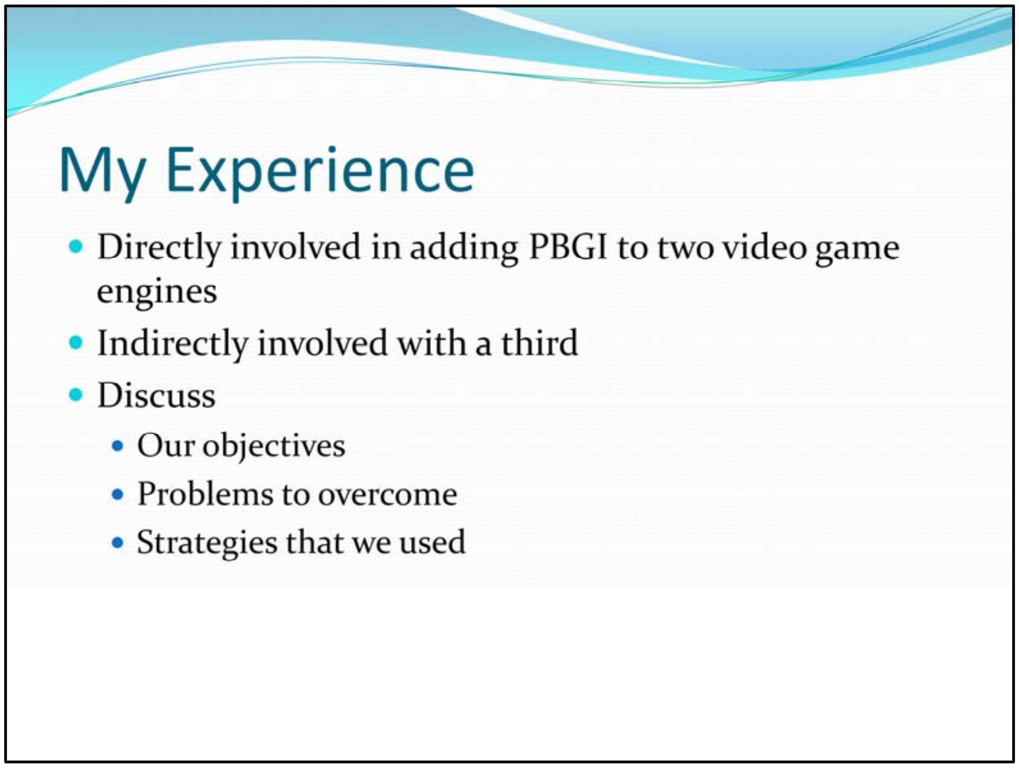
Michael Bunnell
Fantasy Lab



Introduction

- Point-based GI has been used for final render in dozens of films
  - Special effects in films like Iron Man and Star Trek
  - Animated films like How to Train Your Dragon and Toy Story 3
- PBGI is Academy Award winning technology
  - Scientific and Engineering Award for "Point-based rendering for indirect lighting and ambient occlusion" to Per Christensen, Michael Bunnell, and Christophe Hery
- It started out as a real-time technique
- Natural to consider using it in video games

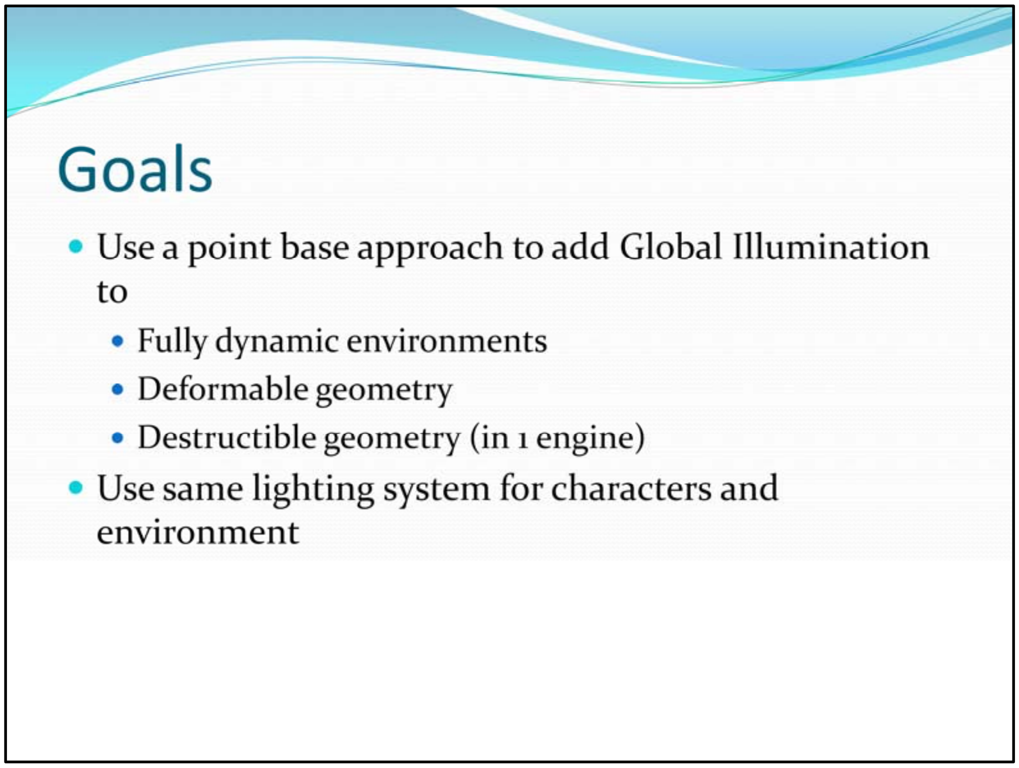
Point-based global illumination is now a standard tool for film-quality renderers. Since it started out as a real-time technique it is only natural to consider using it in video games too.



My Experience

- Directly involved in adding PBGI to two video game engines
- Indirectly involved with a third
- Discuss
  - Our objectives
  - Problems to overcome
  - Strategies that we used

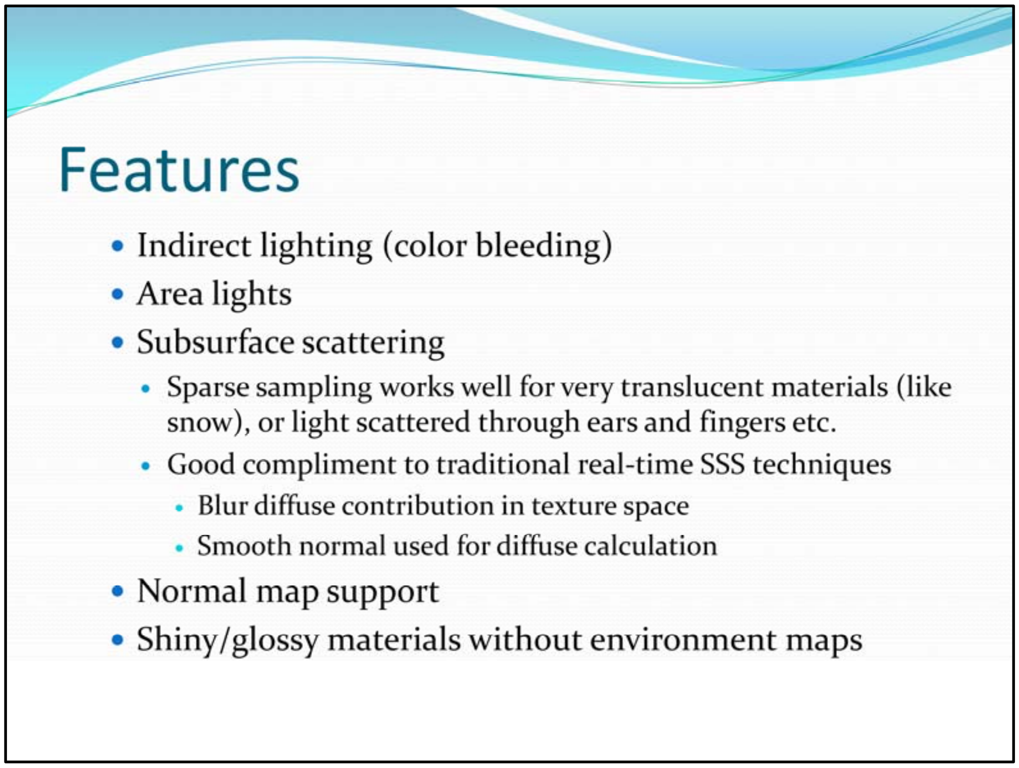
I hope that relating what I learned from the efforts of adding PBGI to three different game engines will be useful. I would like you to come away with an idea of the results you can achieve and what it takes to add PBGI to a game engine (or other real-time application).



Goals

- Use a point base approach to add Global Illumination to
  - Fully dynamic environments
  - Deformable geometry
  - Destructible geometry (in 1 engine)
- Use same lighting system for characters and environment

For all three projects there were some very ambitions goals. All projects needed GI in dynamic environments and deformable geometry. One had destructible environments. Also, we wanted to use the same lighting for characters and the environment. Since 1996 when lightmaps were introduced in he video game "Quake", artists have had to deal with trying to match dynamic lighting on characters to the pre-baked lighting of the environment. It's a lot of work and it never looks quite right.

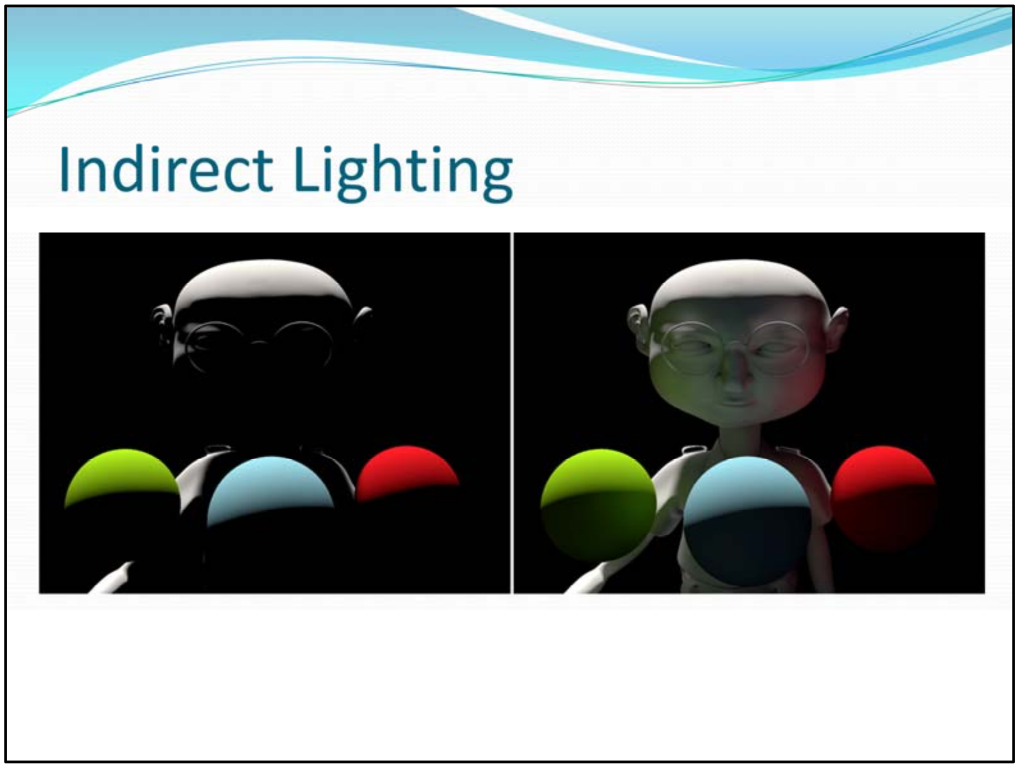


Features

- Indirect lighting (color bleeding)
- Area lights
- Subsurface scattering
  - Sparse sampling works well for very translucent materials (like snow), or light scattered through ears and fingers etc.
  - Good compliment to traditional real-time SSS techniques
    - Blur diffuse contribution in texture space
    - Smooth normal used for diffuse calculation
- Normal map support
- Shiny/glossy materials without environment maps

We wanted indirect lighting, of course, but we also wanted area lights. Unlike film renderers, real-time renders don't usually support area lights because of computation cost. We threw in subsurface scattering because we knew we could use the same sample points for it. We knew we could use standard real-time techniques to handle the short scatter distances, but they don't handle light traveling through the surface. We wanted to handle light shining though ears, fingers, wings etc.

Normal mapping support is a must. Practically everything is normal mapped in games these days. Also we wanted to support specular material properties without using environment mapping.

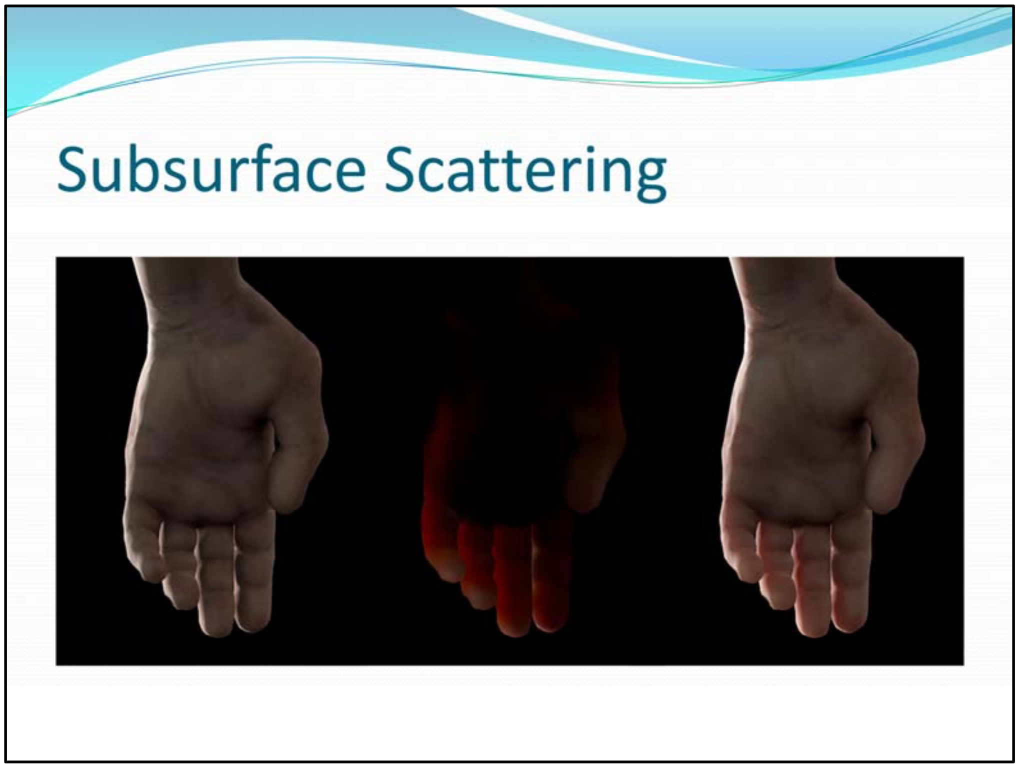


Indirect Lighting

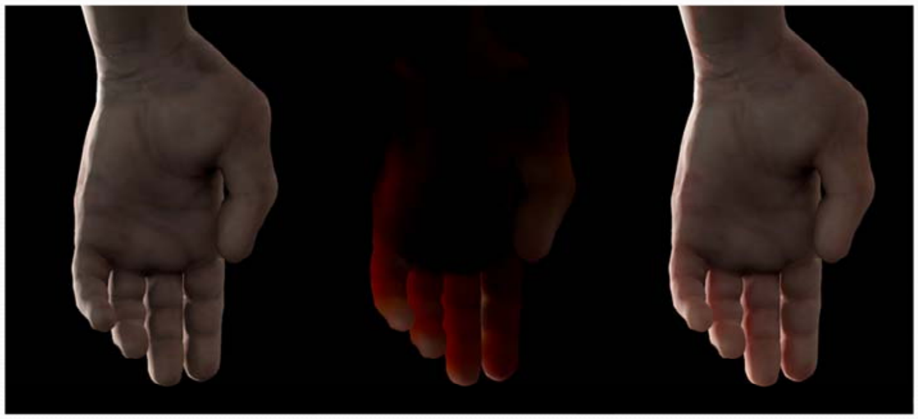
This slide illustrates one of the major features: indirect lighting. The image on the left is lit with direct light only. The one on the right adds indirect lighting. See the color bleeding from the balls to the character’s face.

Area Lights

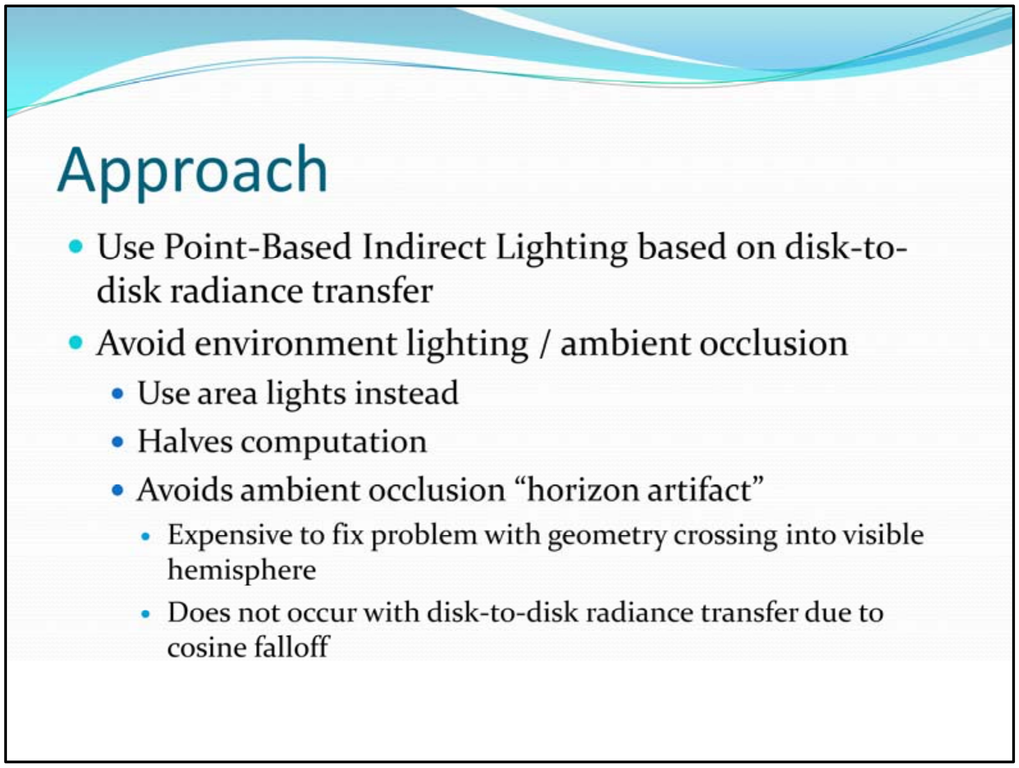
Area lights is another major feature. If you can handle indirect lighting then area lights of any shape are simple. You just set the material to emit light regardless of what light it receives.



Subsurface Scattering



Subsurface Scattering is the third major feature. The hand on the left has no SSS. The SSS results rendered in the center image are added to get the image on the right. We specify the subsurface scatter using a per-component distance value. It may not be as precise as other methods, but the end result is better because it is easy to understand and control. Hand sampled at only 225 points.



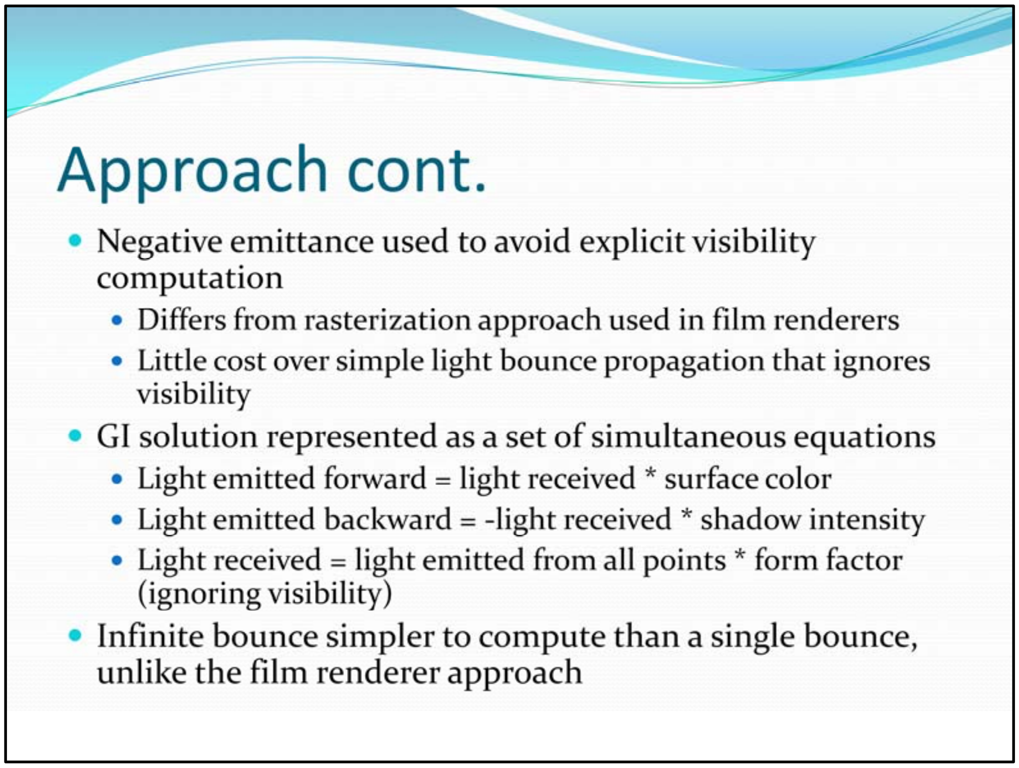
Approach

- Use Point-Based Indirect Lighting based on disk-to-disk radiance transfer
- Avoid environment lighting / ambient occlusion
  - Use area lights instead
  - Halves computation
  - Avoids ambient occlusion "horizon artifact"
    - Expensive to fix problem with geometry crossing into visible hemisphere
    - Does not occur with disk-to-disk radiance transfer due to cosine falloff

The PBGI approach we take is based on disk-to-disk radiance transfer. We save the irradiance results so we can avoid recomputing them in static areas. Also, it allows us to use them as a starting point when solving for irradiance for the next frame.

We don't compute ambient occlusion since it takes just as much time as indirect lighting and we can achieve the same results with area lights. Ambient occlusion is only used to pre-compute occlusion data related to normal mapping.

Also, point-based ambient occlusion is prone to artifacts that are noticeable when animated. Doing the necessary work to remove the artifacts is computationally expensive. Disk-to-disk radiance transfer, on the other hand, does not have that problem because it includes a cosine falloff term. A point crossing into or out of the visible hemisphere of another point will not cause "popping".

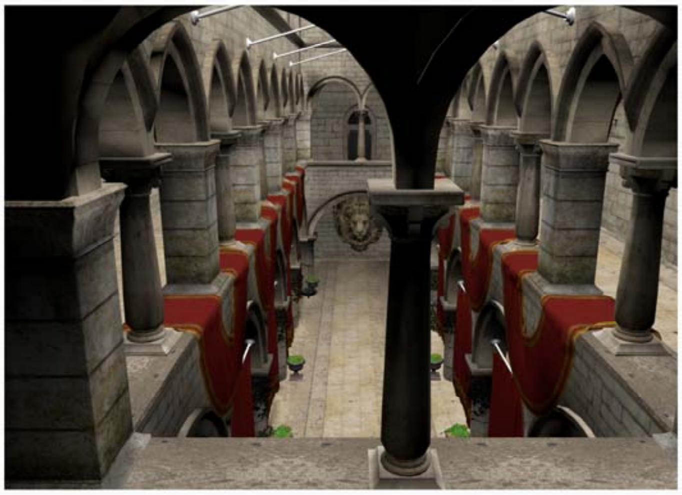


Approach cont.

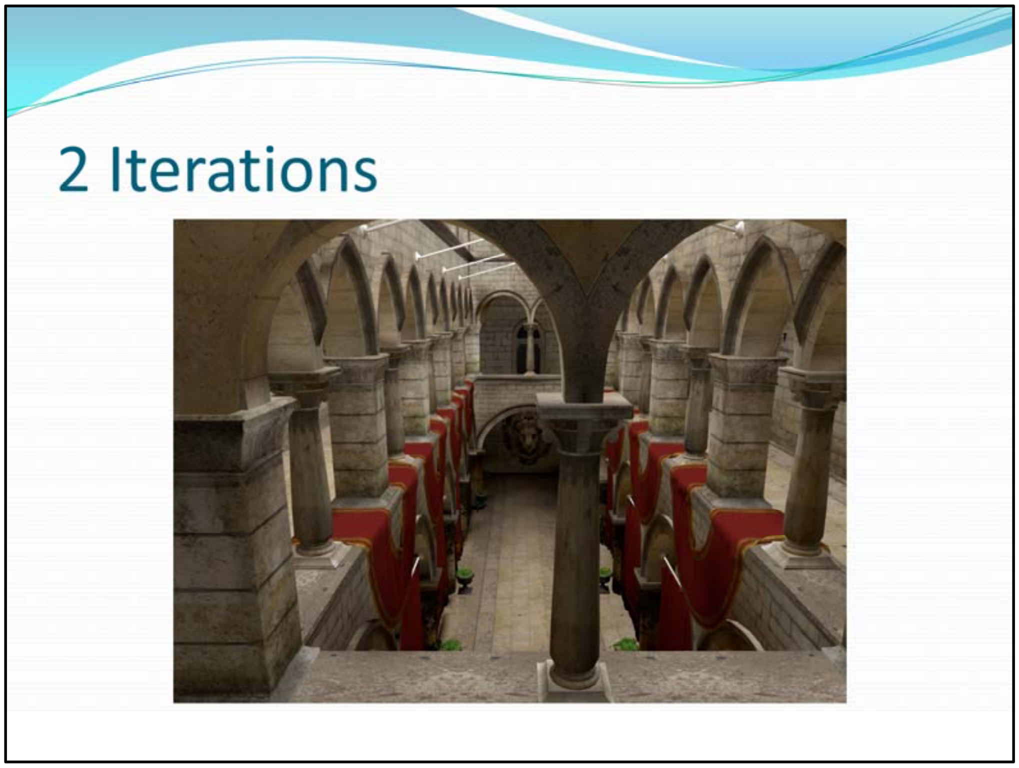
- Negative emittance used to avoid explicit visibility computation
  - Differs from rasterization approach used in film renderers
  - Little cost over simple light bounce propagation that ignores visibility
- GI solution represented as a set of simultaneous equations
  - Light emitted forward = light received * surface color
  - Light emitted backward = -light received * shadow intensity
  - Light received = light emitted from all points * form factor (ignoring visibility)
- Infinite bounce simpler to compute than a single bounce, unlike the film renderer approach

A major difference between our real-time approach and a film based approach to GI is the use of negative emittance. It's a very inexpensive way to avoid explicit visibility calculations. Each sample point's emittance is positive in the direction of its normal and negative in the opposite direction. We get a GI result by solving a set of simultaneous equations.

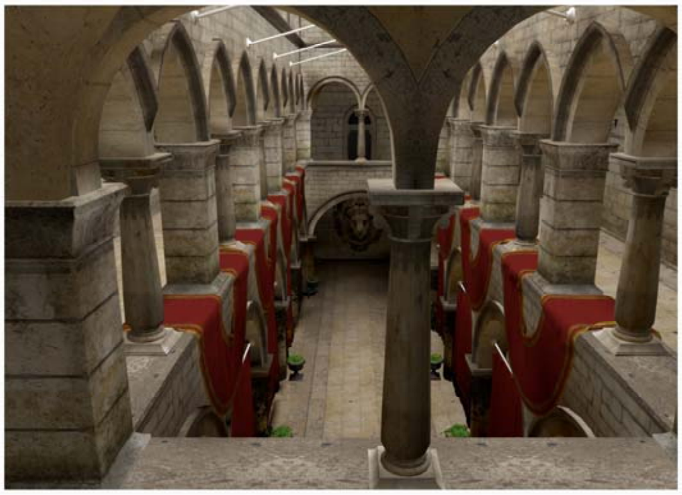
1 Iteration



It may seem like an exaggeration to suggest an infinite bounce solution. However, in practice the GI solution converges vary quickly. Here we see that GI for a fairly complex data converges in only 5 iterations. After one iteration we see only direct light (from sky light) and no shadows.

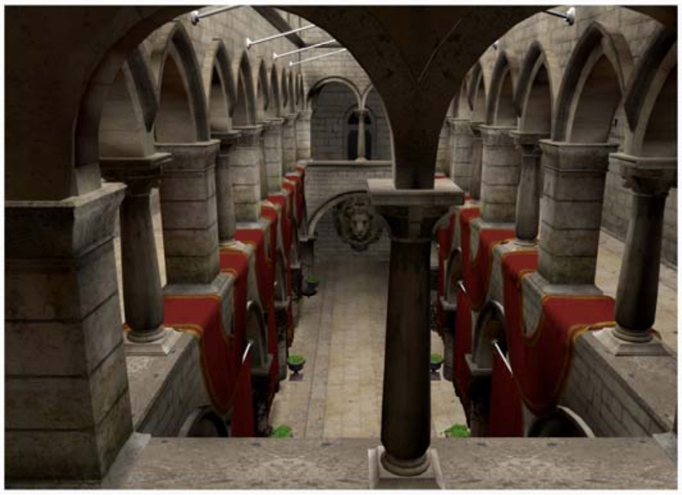


2 Iterations

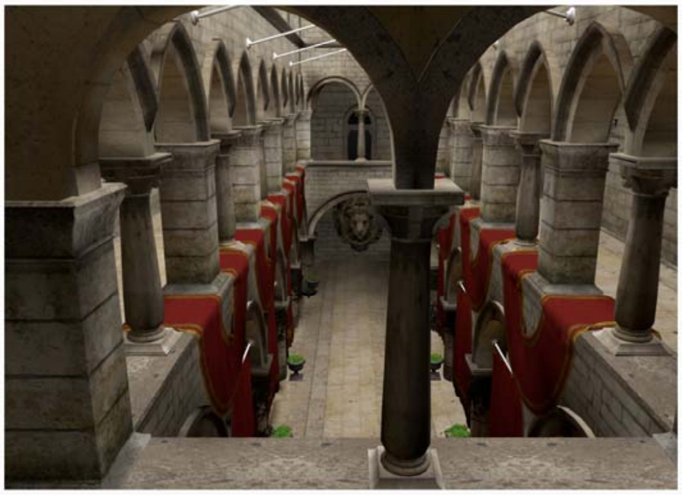


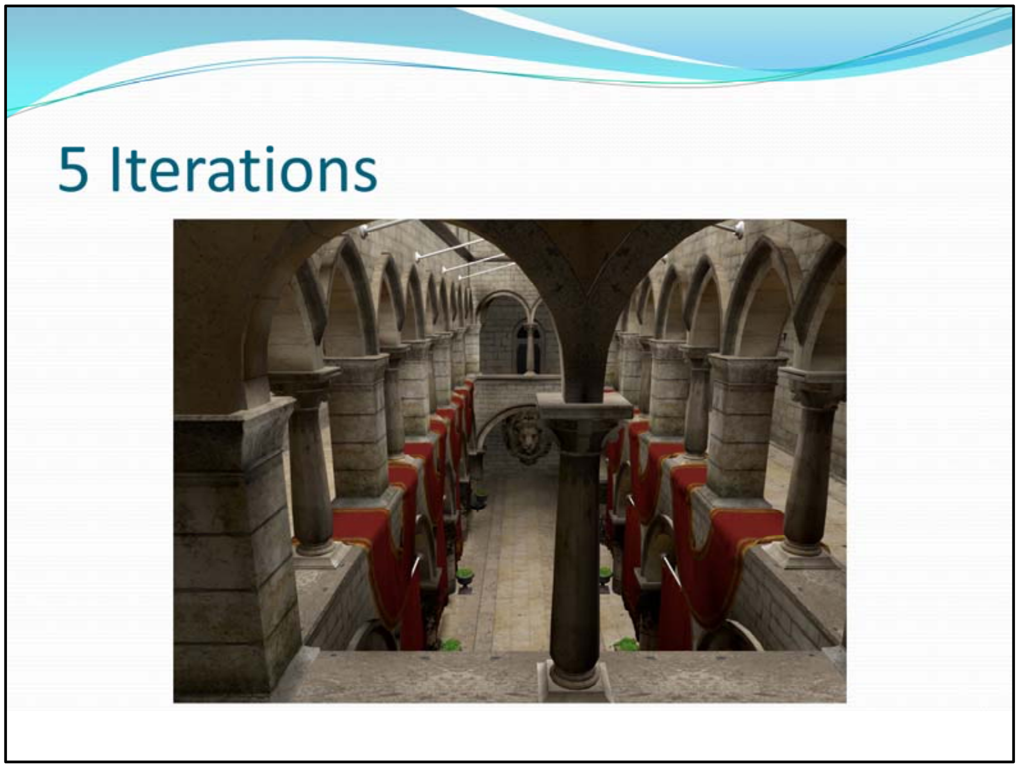
After a second iteration we start to see shadows, but they are too dark. The indirect reflection is too bright in spots too.

3 Iterations

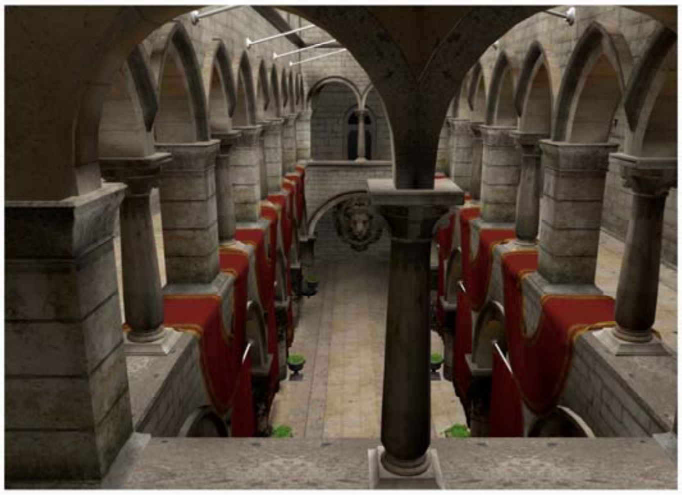


4 Iterations

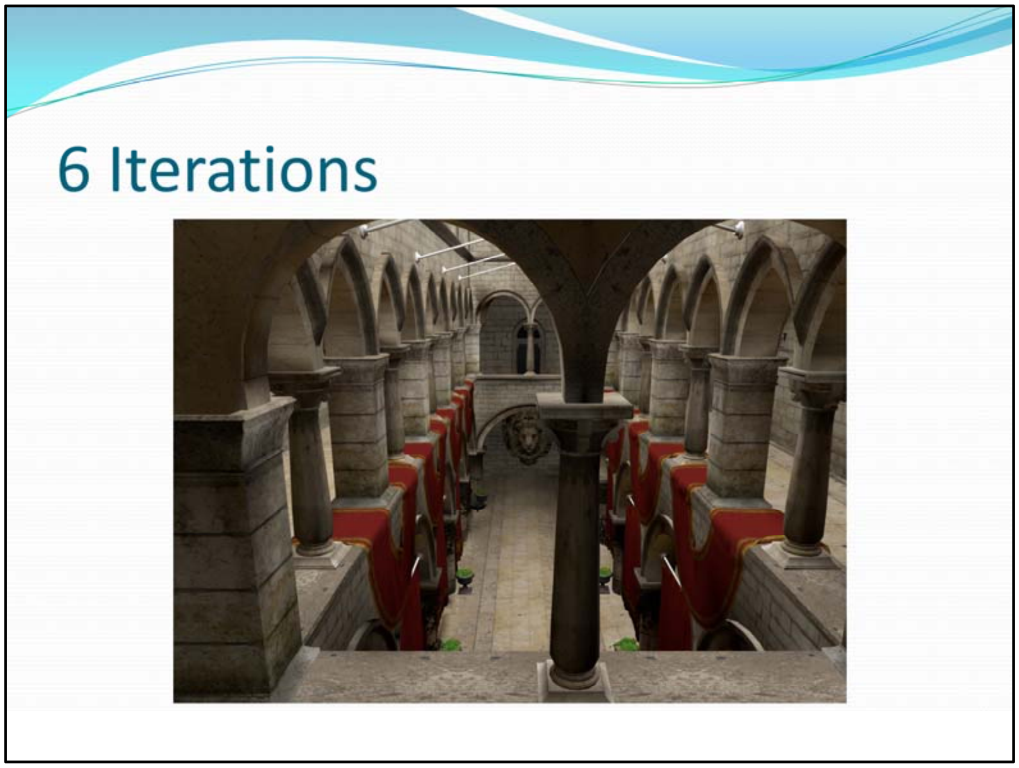




5 Iterations

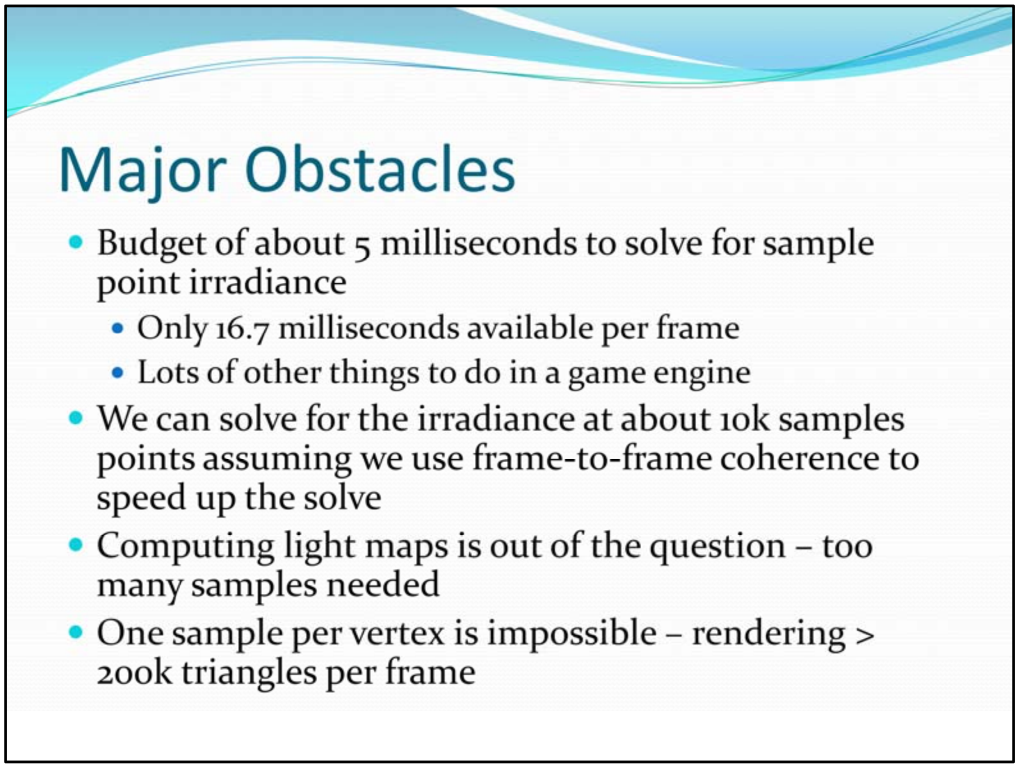


The difference between 4 iterations and 5 is barely visible.



6 Iterations

6 iterations yields the same result as 5 except for some LSB difference. The results converged at 5 iterations.



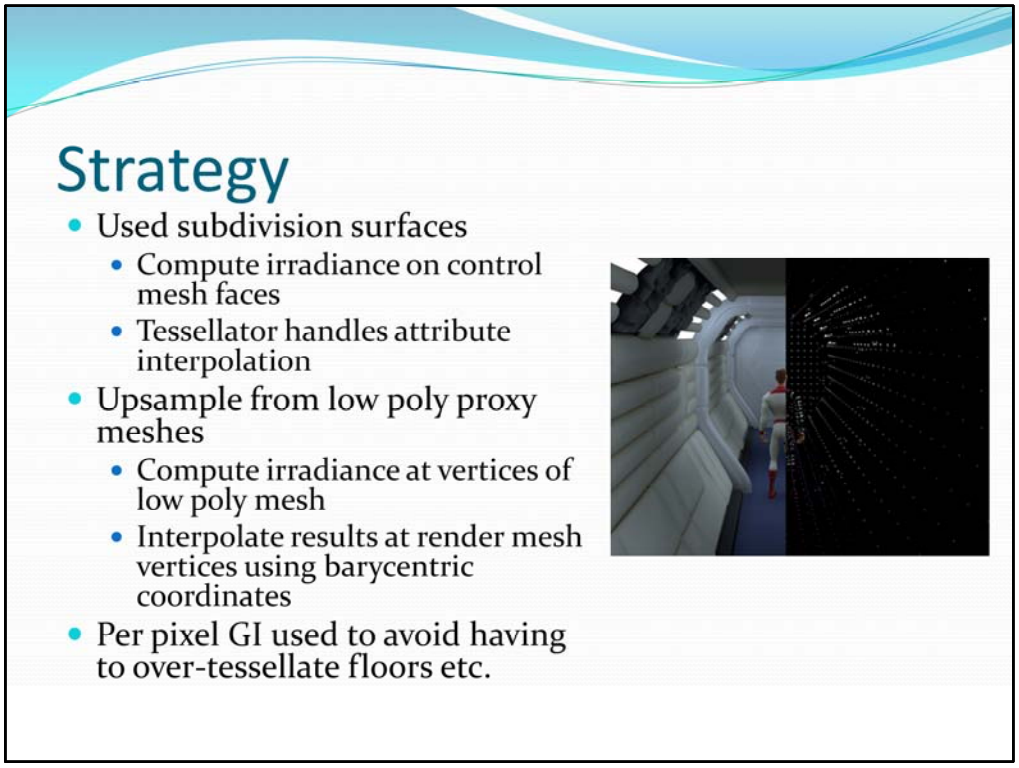
Major Obstacles

- Budget of about 5 milliseconds to solve for sample point irradiance
  - Only 16.7 milliseconds available per frame
  - Lots of other things to do in a game engine
- We can solve for the irradiance at about 10k samples points assuming we use frame-to-frame coherence to speed up the solve
- Computing light maps is out of the question – too many samples needed
- One sample per vertex is impossible – rendering > 200k triangles per frame

The biggest obstacle that we had to face is that we only have 5 milliseconds to compute the GI for each sample point. We were working on a game engine not a tech demo. We could not afford to take up all the processing power/time just for GI. There are lots of other things the game engine has to do: physics, animation, running scripts, etc. not to mention rendering the frame.

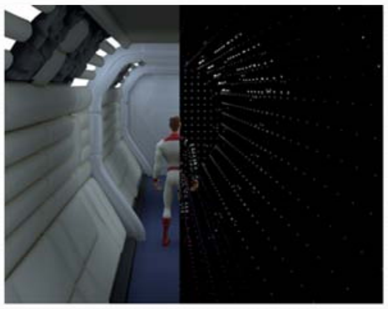
5 ms only lets us solve for the irradiance (amount of light received) at about 10k samples points, depending on the target hardware and other factors. (assuming that we use frame-to-frame coherence to speed up the solve). That's not a lot of time. Using a light-mapping type approach was out of the question. We would need orders of more samples to do that.

Even creating one sample per vertex of each surface's mesh is impossible. The game renders hundreds of thousands of triangles per frame.



Strategy

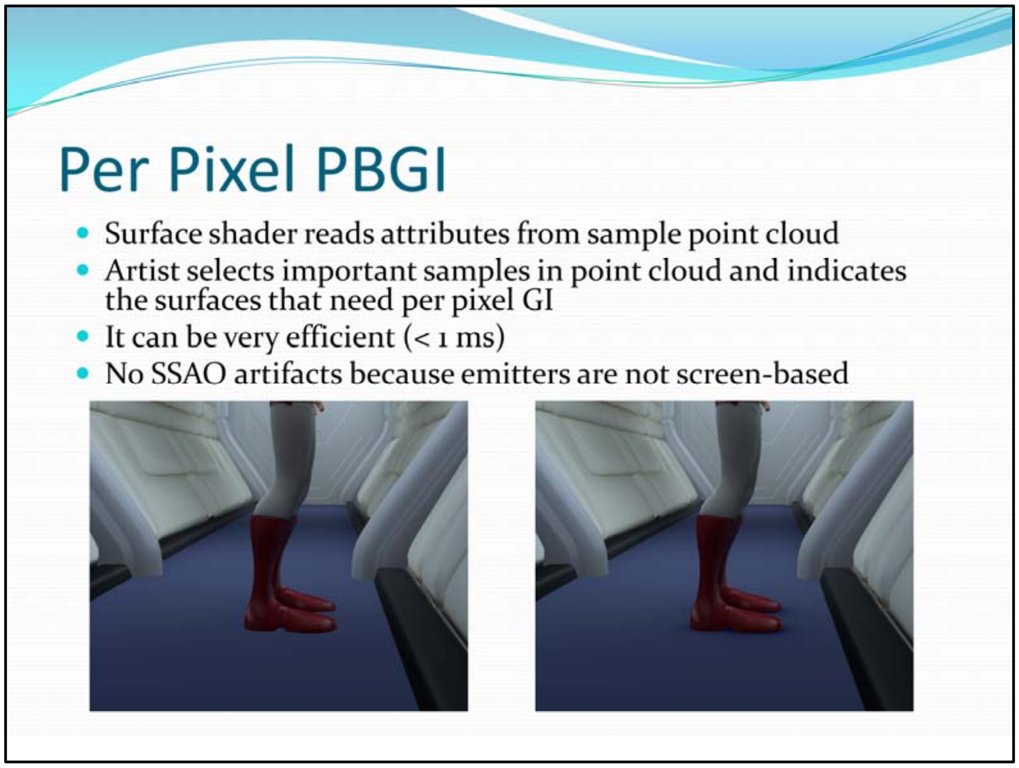
- Used subdivision surfaces
  - Compute irradiance on control mesh faces
  - Tessellator handles attribute interpolation
- Upsample from low poly proxy meshes
  - Compute irradiance at vertices of low poly mesh
  - Interpolate results at render mesh vertices using barycentric coordinates
- Per pixel GI used to avoid having to over-tessellate floors etc.



One of the engines uses subdivision surfaces for everything and they are tessellated on the fly. An easy solution is to create samples on the subdivision surface control mesh and interpolate the GI results with the tessellator.

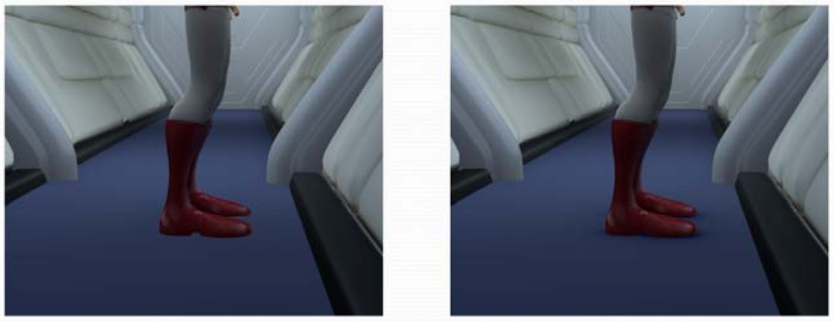
The other 2 engines used polygons for everything. They had LOD support, but it was based on decimating a high-resolution mesh. Typical characters in the game were 4k to 8k triangles at the highest LOD. We ended up using a sample proxy mesh scheme. We could use a low detail version of the mesh to sample and then interpolate the results for the render mesh.

We could not afford to subdivide floors enough to get good contact (or near contact) shadows from small items (like feet) without blowing our sample budget. We chose a per-pixel technique to handle that scenario.



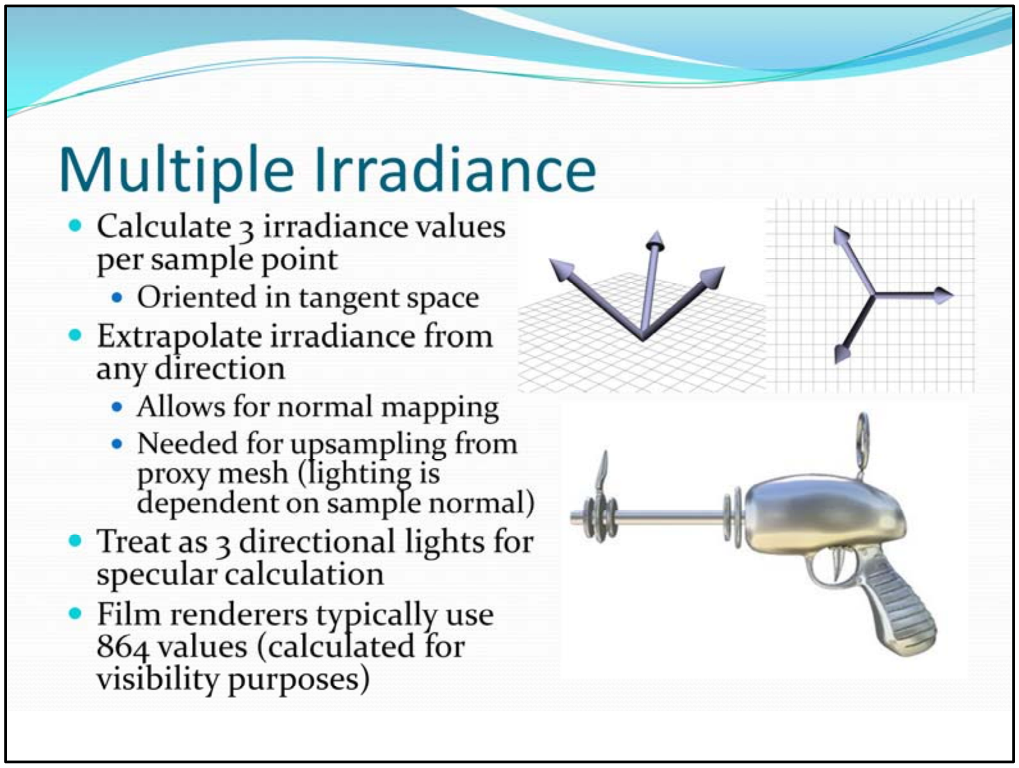
Per Pixel PBGI

- Surface shader reads attributes from sample point cloud
- Artist selects important samples in point cloud and indicates the surfaces that need per pixel GI
- It can be very efficient (< 1 ms)
- No SSAO artifacts because emitters are not screen-based



The per-pixel GI technique works by running a code in the surface shader that looks at the sample points and calculates the radiance transfer from them. It looks at the sample data after the GI solution is solved so it knows how much light is reaching each sample. Since it is run on every pixel (of the desired surface) it cannot afford to traverse the entire sample hierarchy. Instead we require the artist to indicate what parts of surfaces will come in close contact and only the corresponding samples are considered in the shader.

In the image on the left the samples on the floor are good enough to receive proper lighting except around the feet, which seem to be ignored. In the image on the right the floor is rendered with per pixel GI enabled creating a nice contact shadow under the feet.



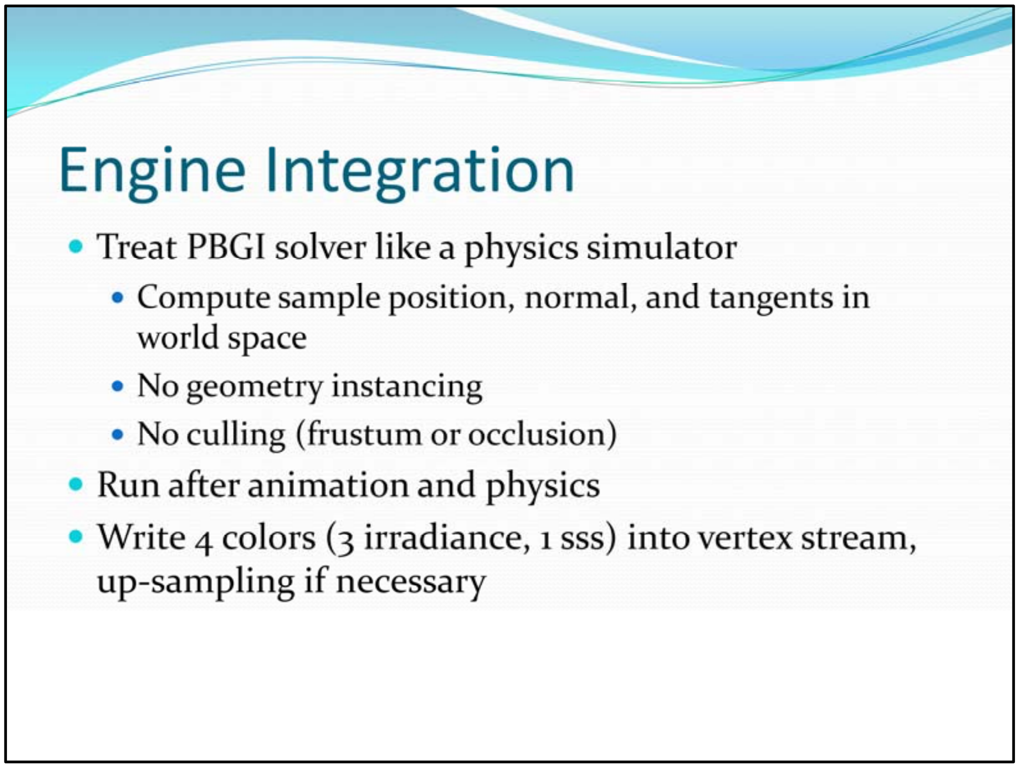
Multiple Irradiance

- Calculate 3 irradiance values per sample point
  - Oriented in tangent space
- Extrapolate irradiance from any direction
  - Allows for normal mapping
  - Needed for upsampling from proxy mesh (lighting is dependent on sample normal)
- Treat as 3 directional lights for specular calculation
- Film renderers typically use 864 values (calculated for visibility purposes)

Instead of computing a single irradiance value per sample we calculate three, looking in three different directions around the normal. Using multiple irradiance values lets us extrapolate an irradiance value for any direction. (Yes, extrapolate. The weighting values can be less than 0 and greater than 1).

Having 3 irradiance values lets us support normal mapping (one of our requirements). It's also used for upsampling from the sample mesh so we can take into account the render mesh's normals.

Also, it lets use a cheap and easy hack to support specular material properties without having to resort to environment maps.

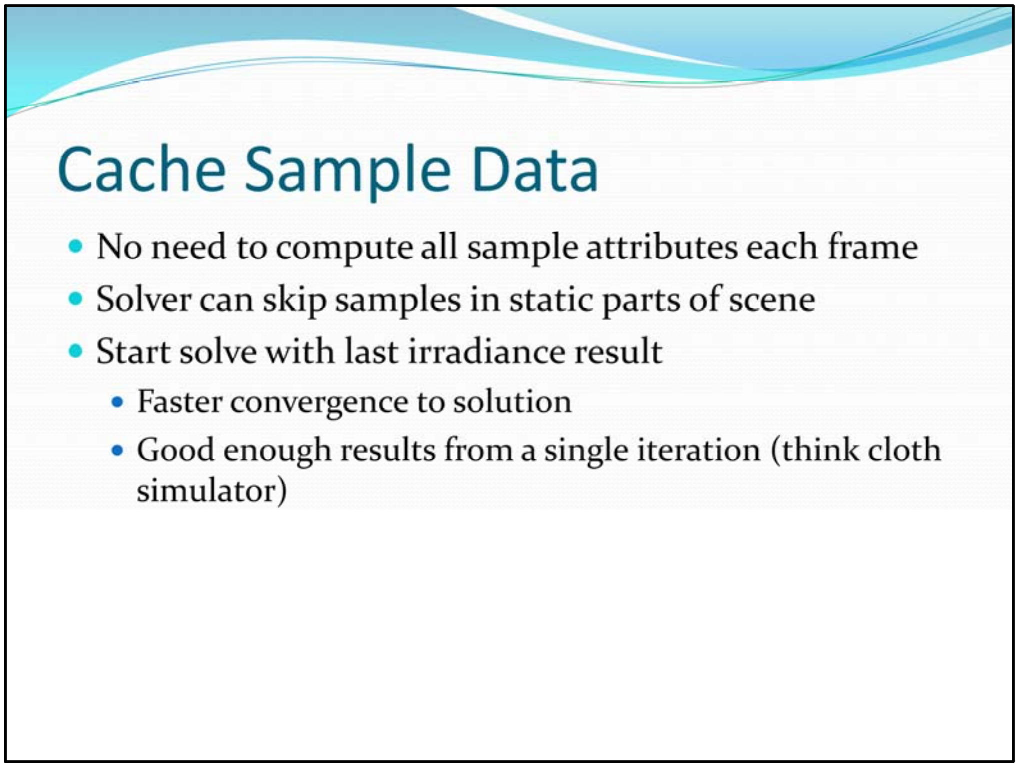


Engine Integration

- Treat PBGI solver like a physics simulator
  - Compute sample position, normal, and tangents in world space
  - No geometry instancing
  - No culling (frustum or occlusion)
- Run after animation and physics
- Write 4 colors (3 irradiance, 1 sss) into vertex stream, up-sampling if necessary

For engine integration we basically treat the PBGI system like a physics simulator. We set all the sample information in world space. We don't allow geometry instancing (since each version of an object needs its own attributes and results). Also, we need to work with the entire environment, not just what is visible. Light reflecting off a red wall should not disappear just because the wall is not visible.

Once the GI solver is run we up-sample (or tessellate for subds) the 3 irradiance values (3 components each) and the SSS value into the vertex stream for render.



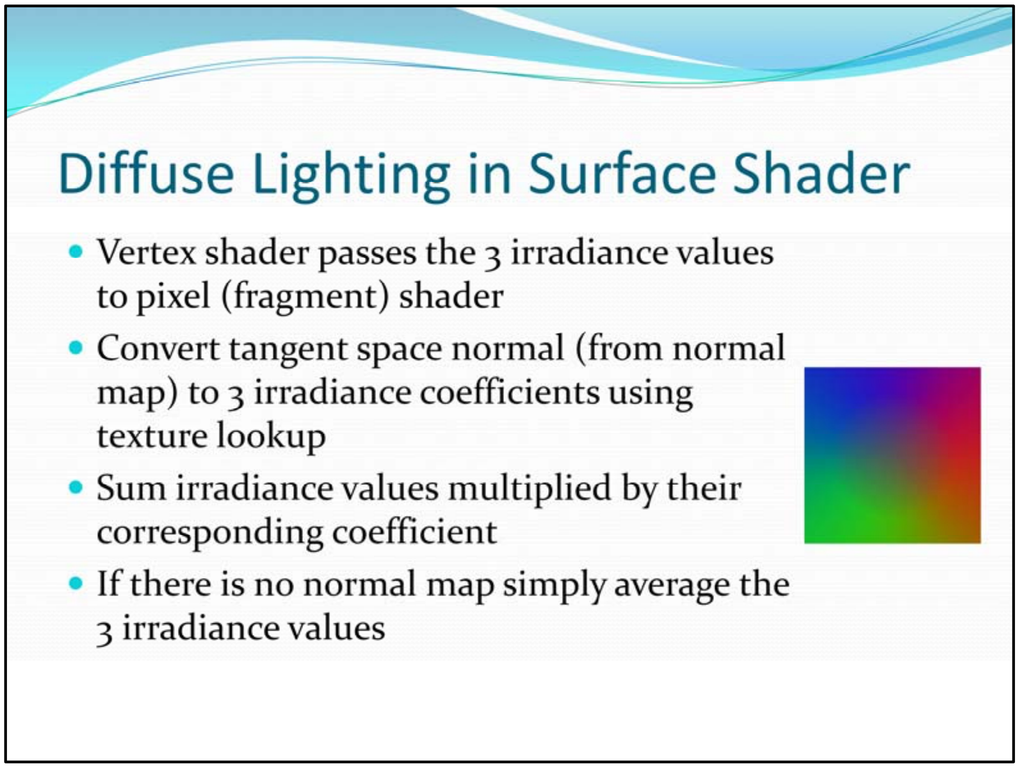
Cache Sample Data

- No need to compute all sample attributes each frame
- Solver can skip samples in static parts of scene
- Start solve with last irradiance result
    - Faster convergence to solution
    - Good enough results from a single iteration (think cloth simulator)

All attributes of each sample (position, normal, tangent, area, reflection color etc.) are cached along with the last irradiance result. That way only the attributes that change each frame need to be written into the sample hierarchy.

Since the results are cached portions of the samples can be skipped to improve performance. Generally only used for static parts of the scene.

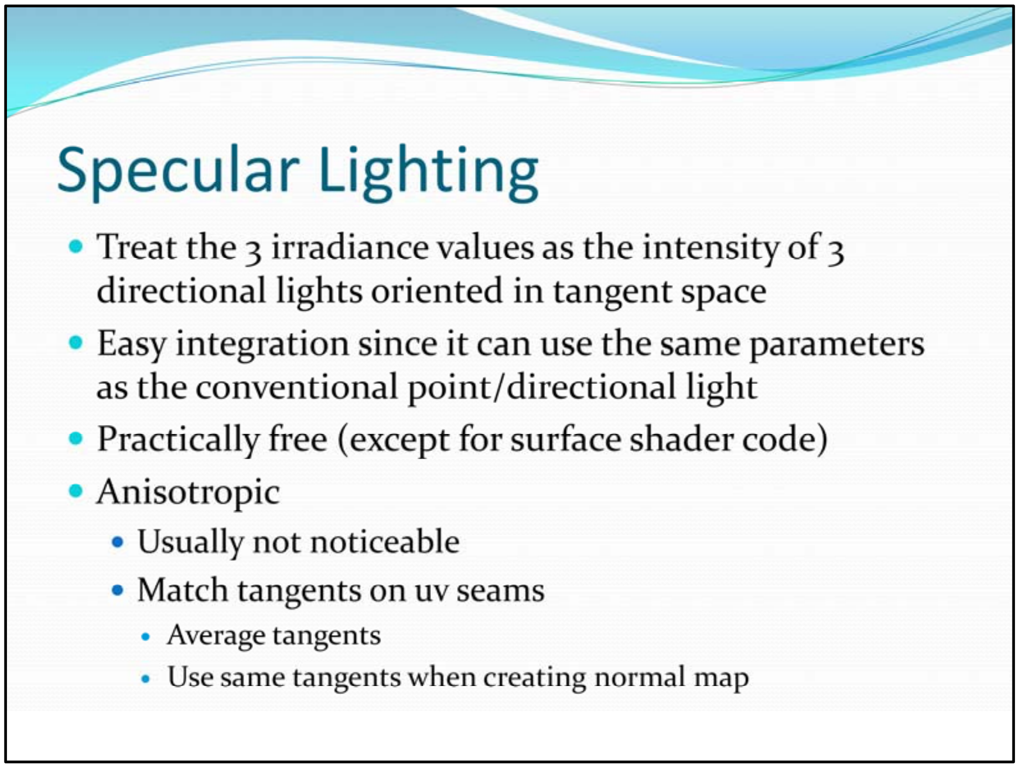
Caching the results lets the solver start with an approximation closer the final solution than all black. Generally it takes 4-5 passes for convergence, but one pass is generally fine when starting from a previous solution.



Diffuse Lighting in Surface Shader

- Vertex shader passes the 3 irradiance values to pixel (fragment) shader
- Convert tangent space normal (from normal map) to 3 irradiance coefficients using texture lookup
- Sum irradiance values multiplied by their corresponding coefficient
- If there is no normal map simply average the 3 irradiance values

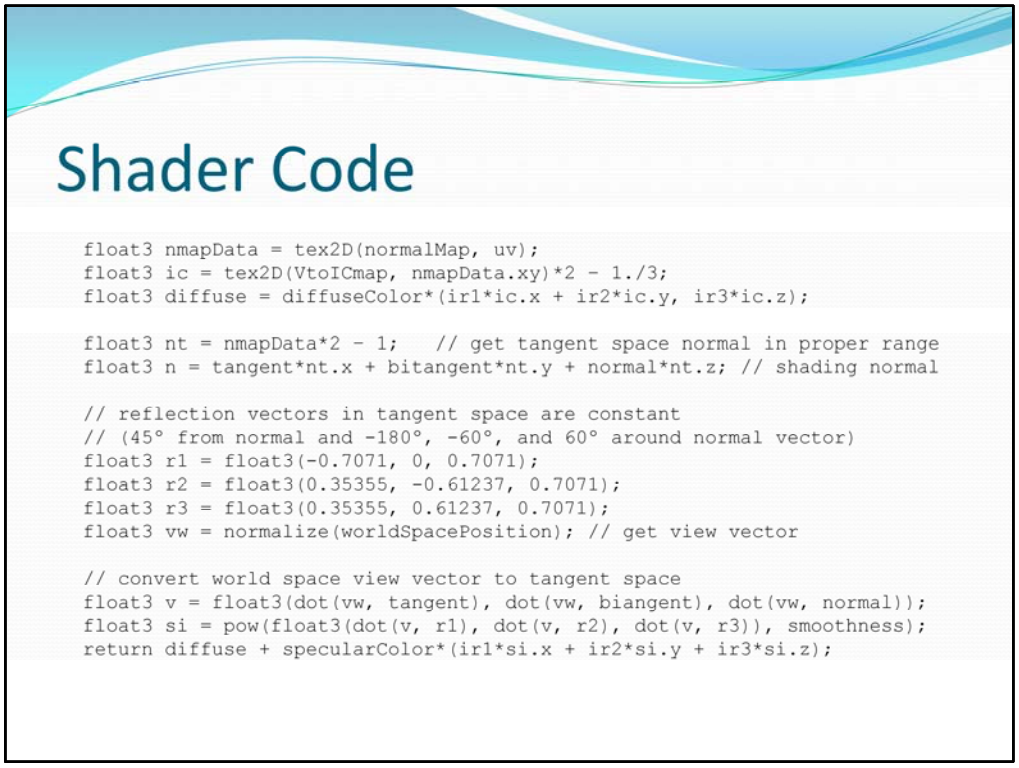
Computing the diffuse term in the surface shader using the three irradiance values at that point is pretty straightforward. If there is no normal map then average the three values and multiply them by the diffuse color. If there is a normal map then convert the normal to three irradiance coefficients using a fixed texture map like the one pictured in this slide. The coefficients need to multiplied by 2 and biased by -1/3 then multiplied by their corresponding irradiance value and summed together then multiplied by the diffuse color.



Specular Lighting

- Treat the 3 irradiance values as the intensity of 3 directional lights oriented in tangent space
- Easy integration since it can use the same parameters as the conventional point/directional light
- Practically free (except for surface shader code)
- Anisotropic
  - Usually not noticeable
  - Match tangents on uv seams
    - Average tangents
    - Use same tangents when creating normal map

The specular term is created using the same 3 irradiance values. In this case they are treated as the intensities of 3 directional lights oriented around the normal. Specular lighting is computed using the Phong reflection model’s specular term for each of the “lights” added together.



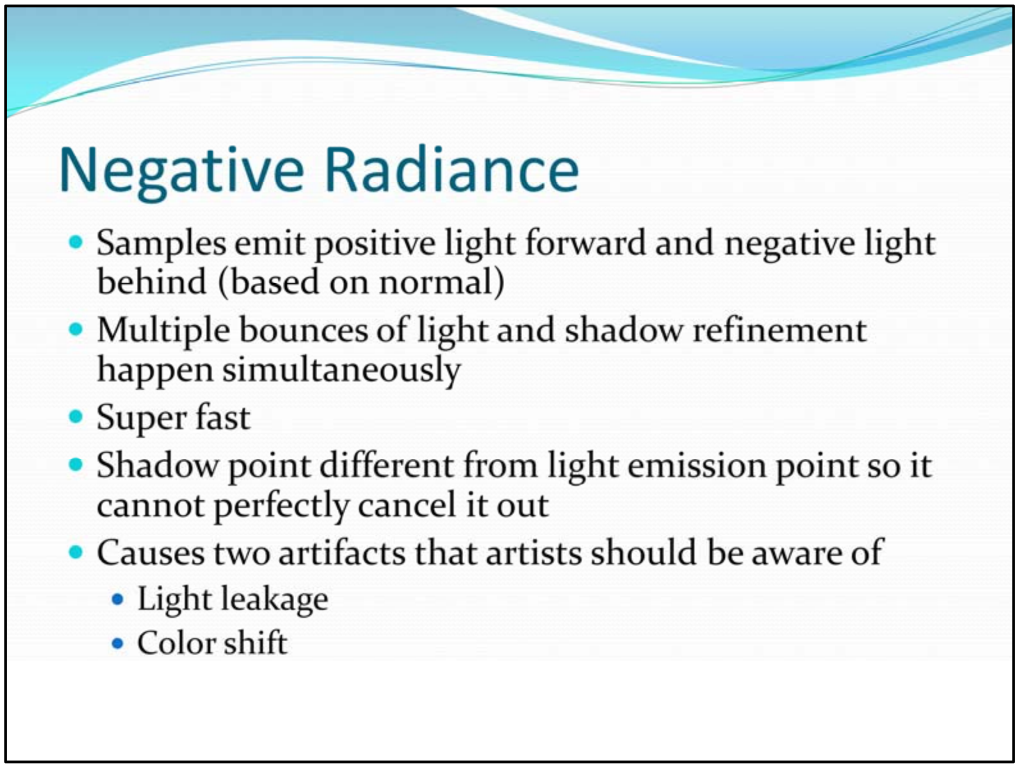
Shader Code

```
float3 nmapData = tex2D(normalMap, uv);
float3 ic = tex2D(VtoICmap, nmapData.xy)*2 - 1./3;
float3 diffuse = diffuseColor*(ir1*ic.x + ir2*ic.y, ir3*ic.z);

float3 nt = nmapData*2 - 1;   // get tangent space normal in proper range
float3 n = tangent*nt.x + bitangent*nt.y + normal*nt.z; // shading normal

// reflection vectors in tangent space are constant
// (45° from normal and -180°, -60°, and 60° around normal vector)
float3 r1 = float3(-0.7071, 0, 0.7071);
float3 r2 = float3(0.35355, -0.61237, 0.7071);
float3 r3 = float3(0.35355, 0.61237, 0.7071);
float3 vw = normalize(worldSpacePosition); // get view vector

// convert world space view vector to tangent space
float3 v = float3(dot(vw, tangent), dot(vw, biangent), dot(vw, normal));
float3 si = pow(float3(dot(v, r1), dot(v, r2), dot(v, r3)), smoothness);
return diffuse + specularColor*(ir1*si.x + ir2*si.y + ir3*si.z);
```

We use the raw normal map data to look up the irradiance coefficients. The diffuse result is simply the diffuse color times the sum of the irradiance values multiplied by their corresponding coefficient.

The irradiance direction vectors are constant in tangent space. We consider them to be pointing away from the surface so we negate their x and y components to create reflection vectors for them.

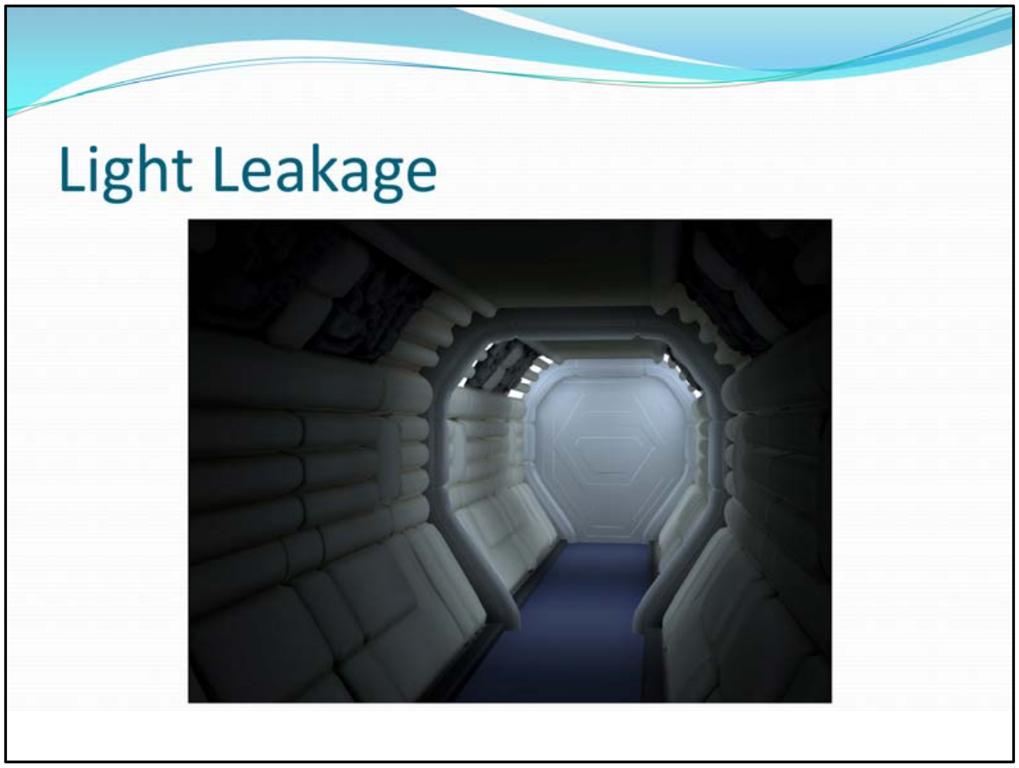
We use the normal and tangent vectors as a basis to convert the view vector to tangent space. The dot product of that vector with each irradiance vector taken to the power of “smoothness” gives the intensity coming from each of these virtual directional lights. We use the corresponding irradiance value to “color” the intensity and multiply the total by the material’s specular color.



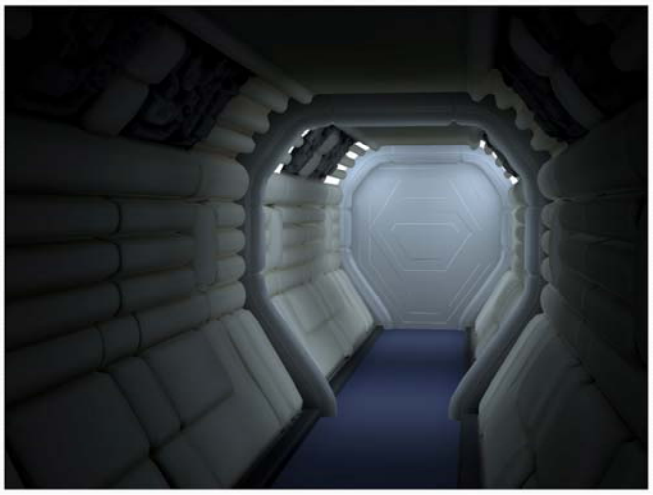
Negative Radiance

- Samples emit positive light forward and negative light behind (based on normal)
- Multiple bounces of light and shadow refinement happen simultaneously
- Super fast
- Shadow point different from light emission point so it cannot perfectly cancel it out
- Causes two artifacts that artists should be aware of
  - Light leakage
  - Color shift

Out GI solver implementation uses negative radiance to handle visibility, basically casting very soft shadows. Since radiance is subtracted from a different point from where it was originally generated it is impossible for it to cancel perfectly. The result can be light leakage and color shift. Usually these artifacts are not noticeable, but if they do crop up there are ways to fix them.

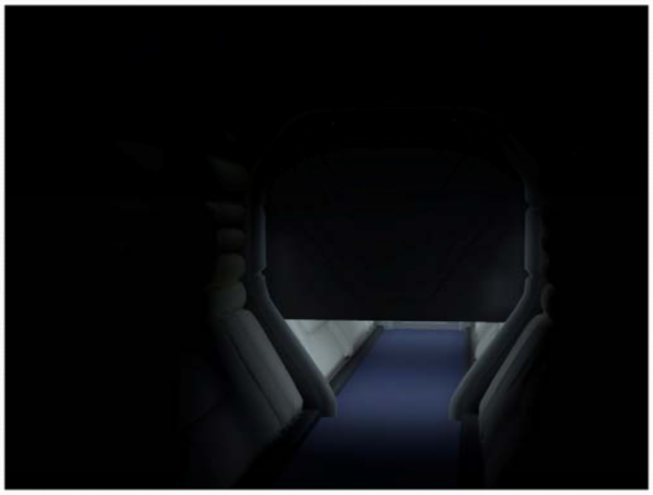


Light Leakage



Here is a corridor lit using area lights at the far end.

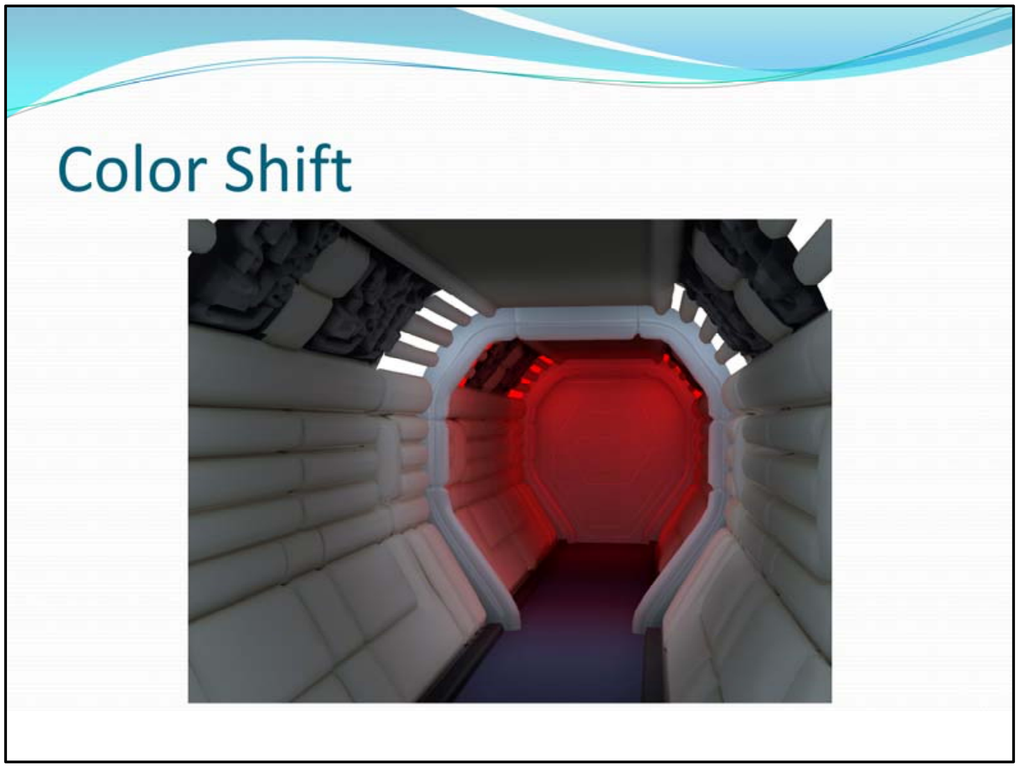
Light Leakage



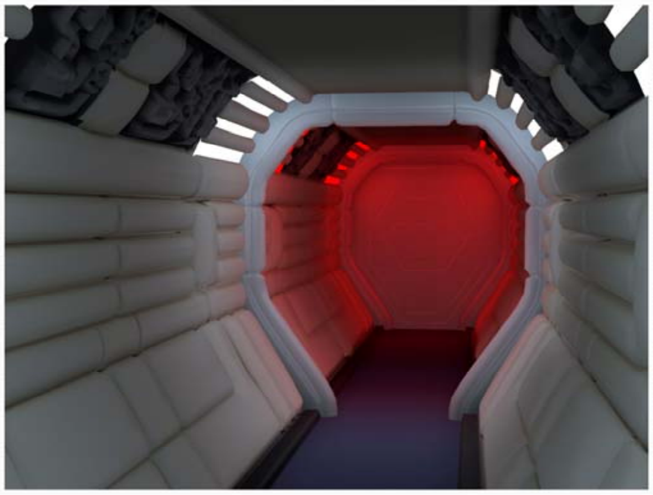
As the door closes light leakage is not noticeable since there is still light shining through the opening.

Light Leakage

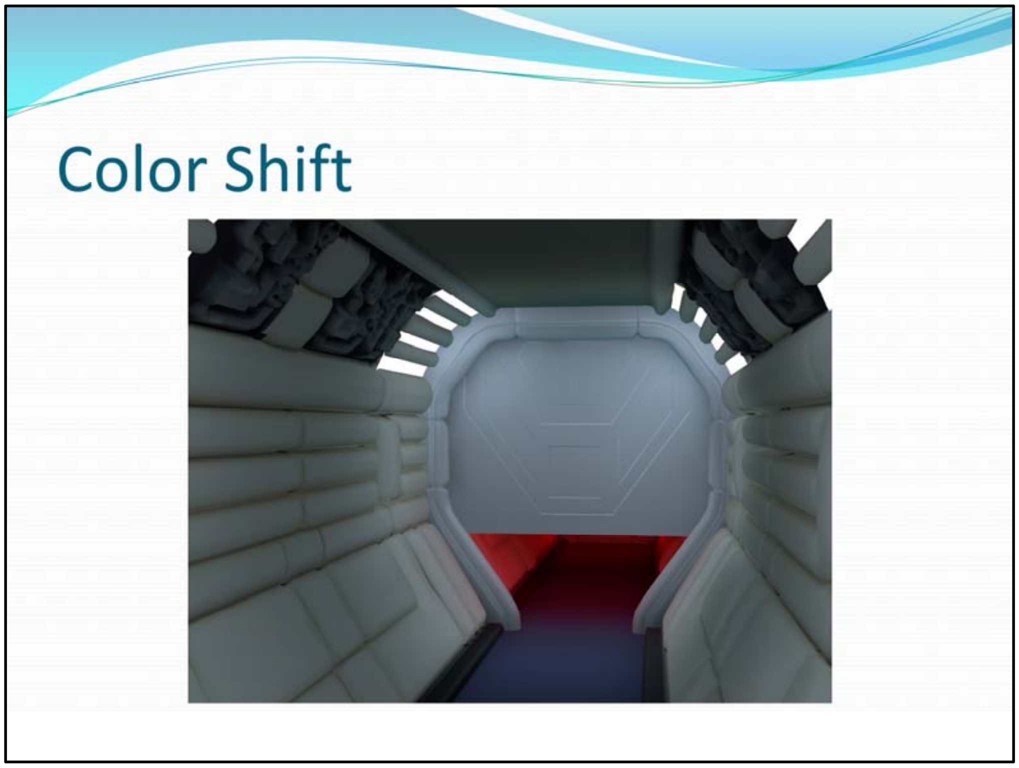
Once the door is completely shut, though. It is obvious that the room should be completely dark, but it is not. Artists can fix this problem by increasing the shadow intensity of the point samples in the door. (or never have completely dark room. How can I play this game if I can’t see anything?)



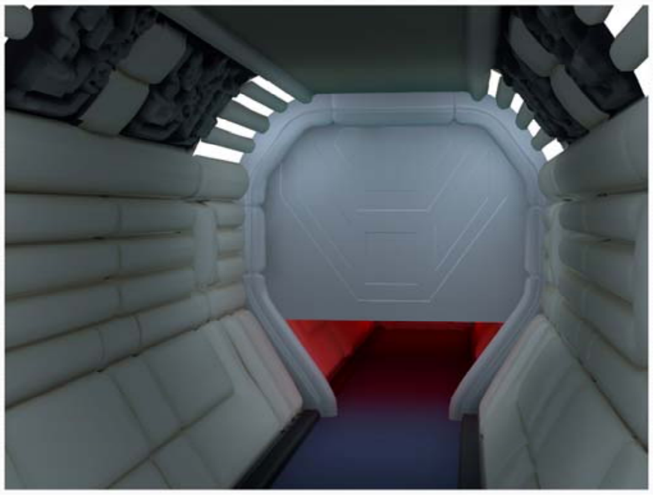
Color Shift



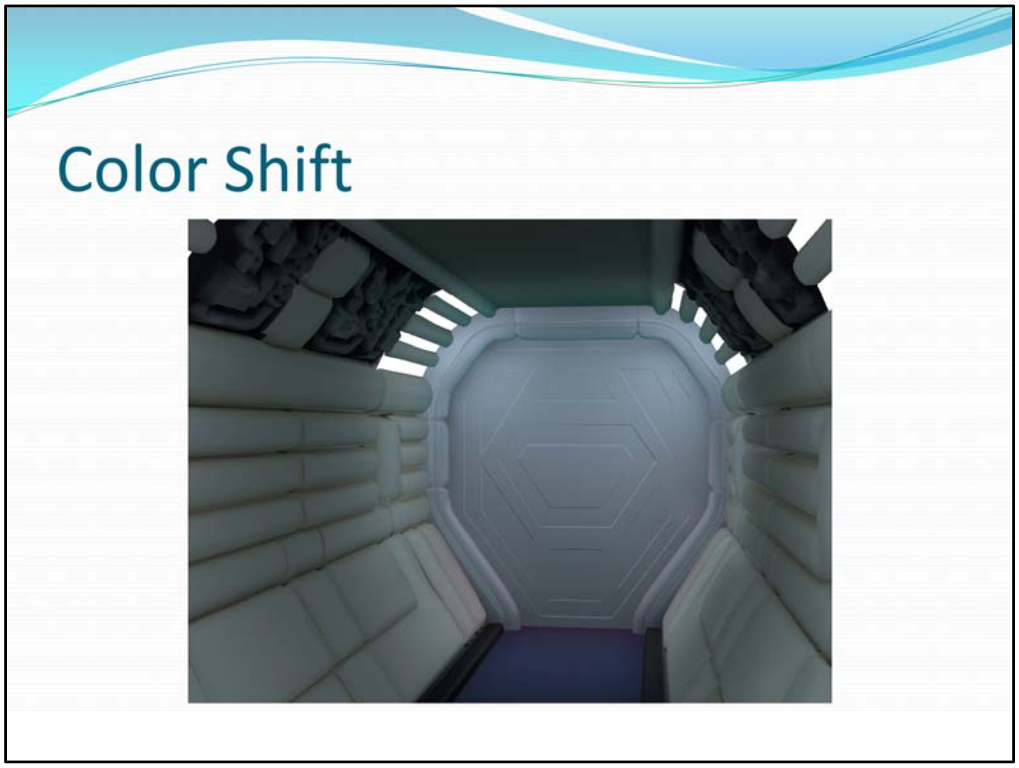
This next problem is trickier. Here we have a corridor lit with area lights. The lights at the far end are red.



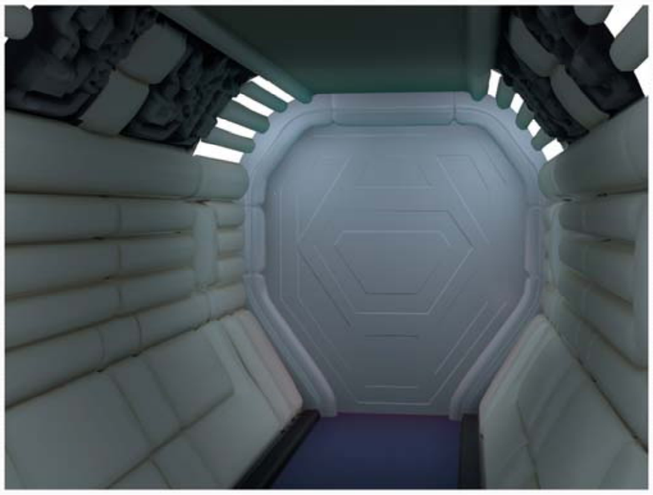
Color Shift



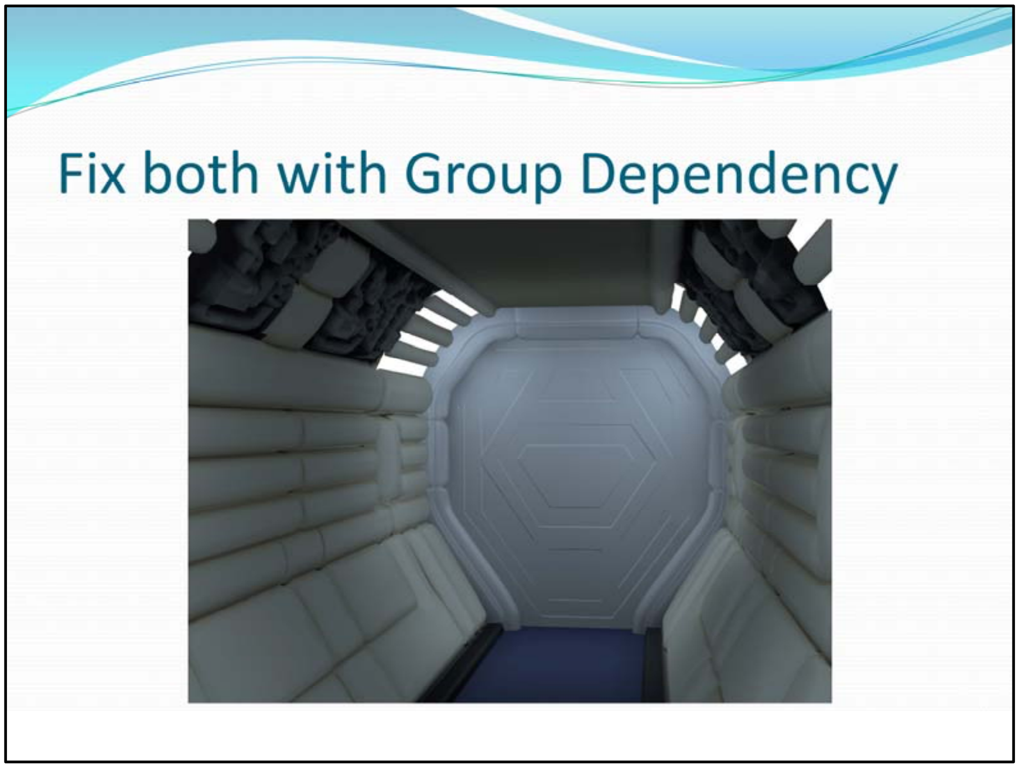
As the door closes we see a cyan shift near the top of it cause from subtracting too much red.



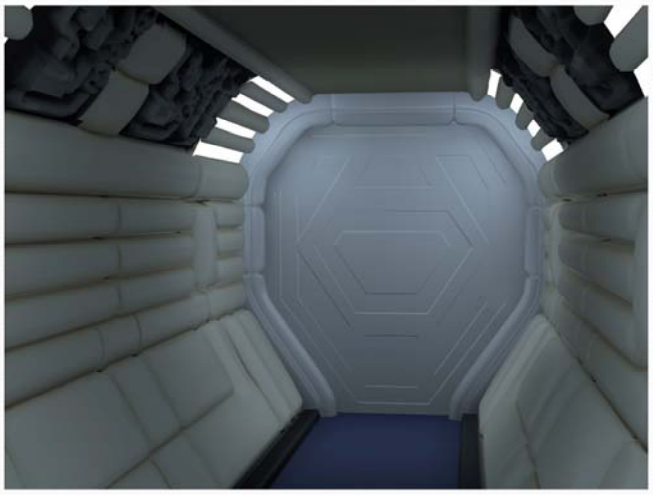
Color Shift



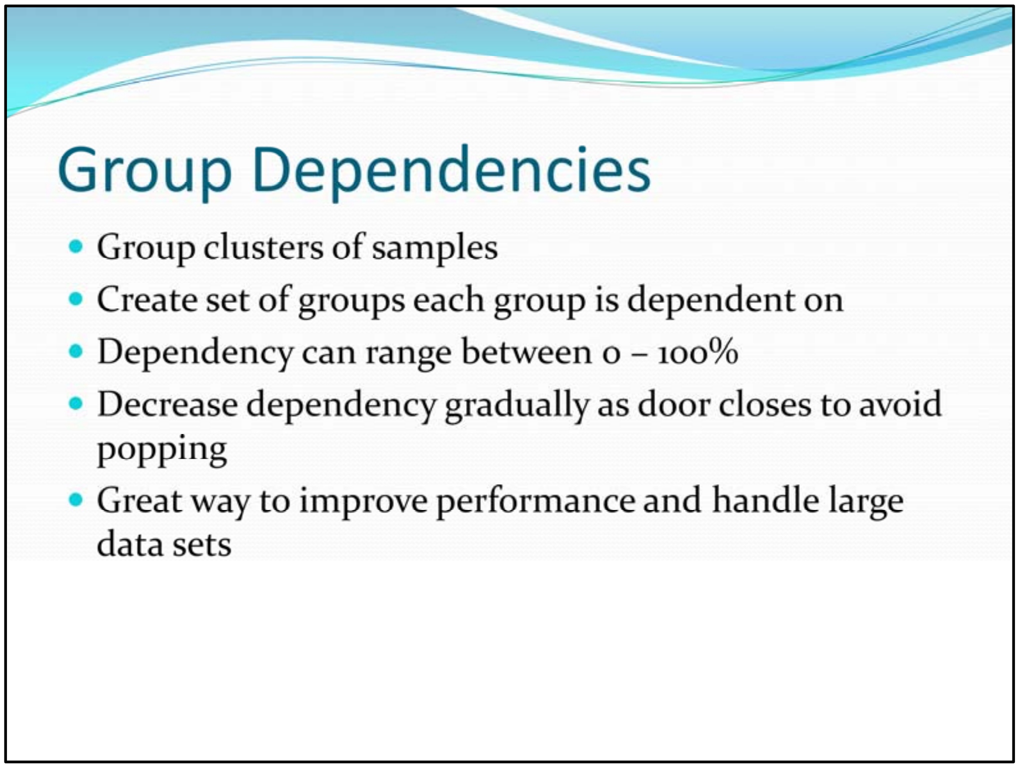
Once the door is completely closed the cyan shift is greater near the top of the door, but there is red light leaking through the bottom of the door. Artists can try to solve this problem by adjusting the shadow intensity values, but there is a better way.



Fix both with Group Dependency



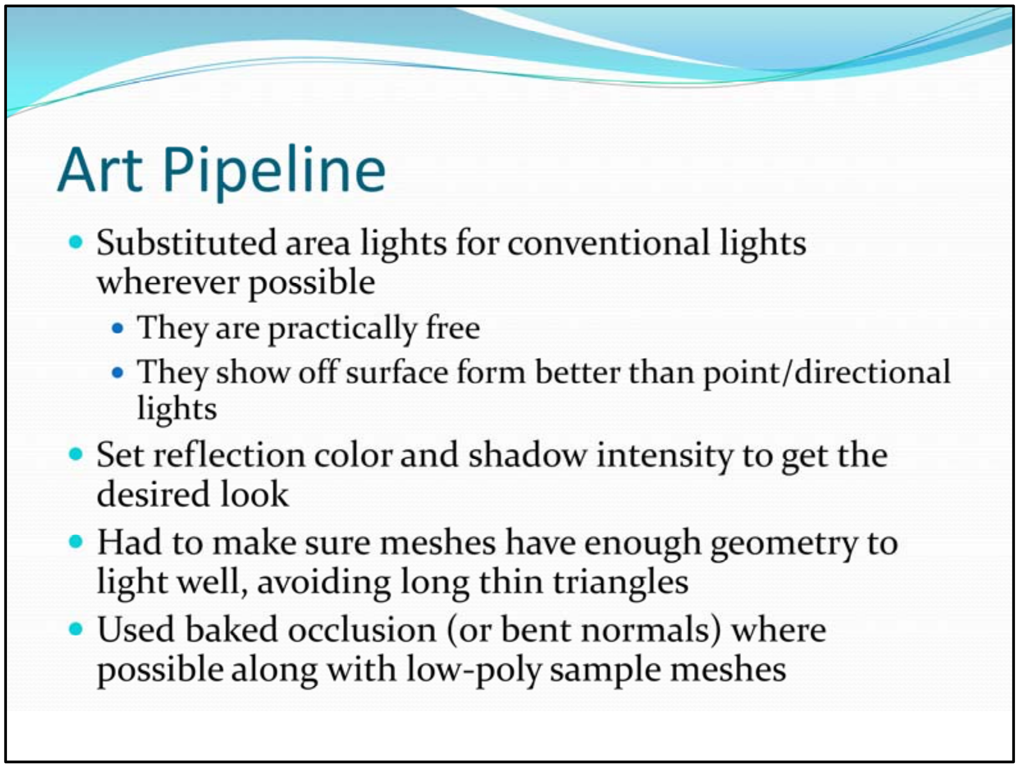
If the samples of the two parts of the corridor are placed into separate groups then we can set there dependencies such that there will be no light leaking through (positive or negative). We simply set the dependency between to two groups to 1 when the door is open and 0 when the door is closed.



Group Dependencies

- Group clusters of samples
- Create set of groups each group is dependent on
- Dependency can range between 0 – 100%
- Decrease dependency gradually as door closes to avoid popping
- Great way to improve performance and handle large data sets

Grouping clusters of samples is a great way to control lighting, improve performance, and manage large game levels. (A cluster of samples are the samples from a single model or mesh).



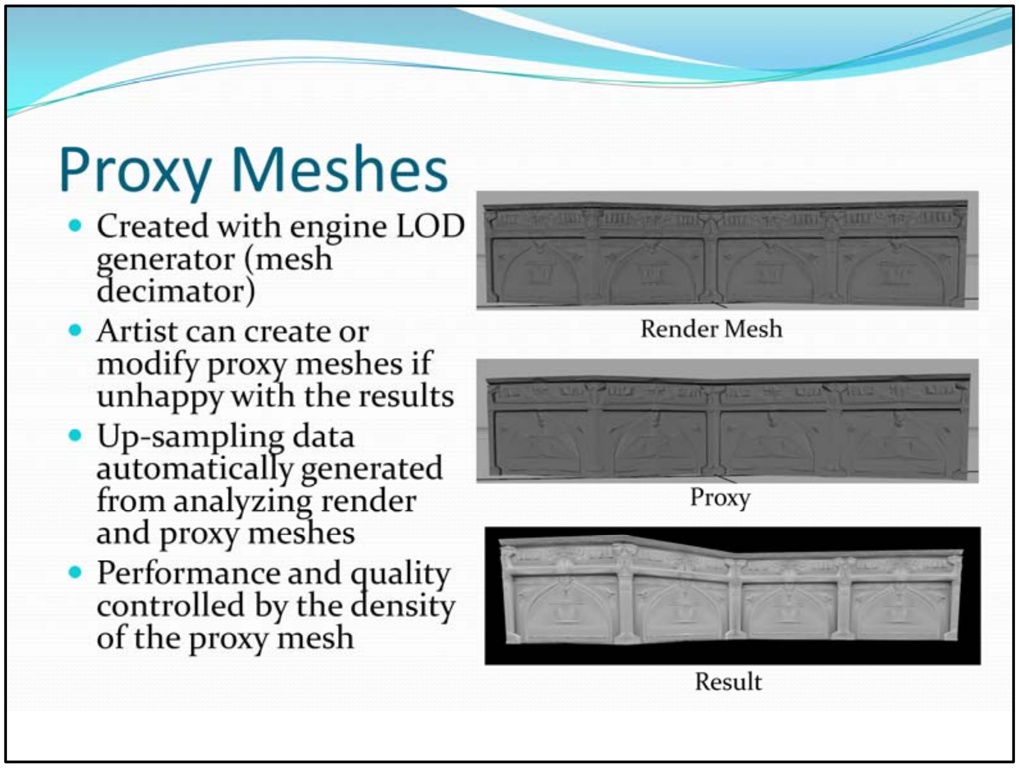
Art Pipeline

- Substituted area lights for conventional lights wherever possible
  - They are practically free
  - They show off surface form better than point/directional lights
- Set reflection color and shadow intensity to get the desired look
- Had to make sure meshes have enough geometry to light well, avoiding long thin triangles
- Used baked occlusion (or bent normals) where possible along with low-poly sample meshes

Once the PBGI system was in place artists removed most of the conventional lighting. Some they replaced with area lights. Others were simply no longer needed once indirect lighting was added. Some of the engine performance lost adding PBGI was recovered by reducing the number of lights.

The artists did play with the reflection color and shadow intensity of samples to get look they wanted, but mostly they used the default values.

These last two points are suggestions really. The artists did not really modify the geometry for lighting and did not bake occlusion for composite objects as much as I would like.



Proxy Meshes

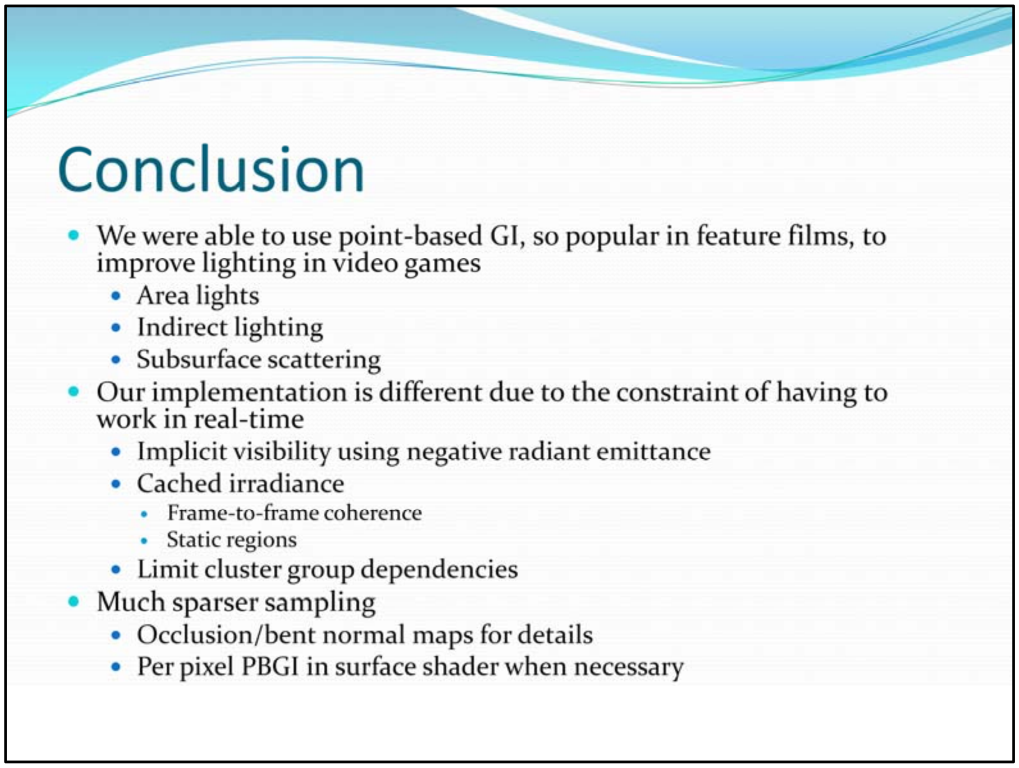
- Created with engine LOD generator (mesh decimator)
- Artist can create or modify proxy meshes if unhappy with the results
- Up-sampling data automatically generated from analyzing render and proxy meshes
- Performance and quality controlled by the density of the proxy mesh

Render Mesh

Proxy

Result

Adding proxy meshes did not really affect the art pipeline since we made it basically automatic. We did allow the artists to specify the LOD of the point sample mesh or supply a custom mesh if they wanted too, though.



Conclusion

- We were able to use point-based GI, so popular in feature films, to improve lighting in video games
  - Area lights
  - Indirect lighting
  - Subsurface scattering
- Our implementation is different due to the constraint of having to work in real-time
  - Implicit visibility using negative radiant emittance
  - Cached irradiance
    - Frame-to-frame coherence
    - Static regions
  - Limit cluster group dependencies
- Much sparser sampling
  - Occlusion/bent normal maps for details
  - Per pixel PBGI in surface shader when necessary

We were able to use point based GI, which has become so popular in feature films, to improve lighting in video games. Specifically, we were able to add support for area lights (of any shape), indirect lighting, and improved SSS in fully dynamic environments.

However, due to the severe time-restraint of having to render in real-time, our implementation is quite different from what is used for film rendering. Specifically, we use negative radiance instead of explicitly calculating visibility. Also, we cache the irradiance result to take advantage of frame-to-frame coherence, and to save calculating irradiance in static regions. We also allow artists to set dependencies of groups of clusters to reduce computation time (and artifacts caused by negative radiance).

Lastly, we deal with sparse sampling by relying on occlusion maps (or bent normal maps) and using per pixel GI to handle contact (or near contact) shadows.