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

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

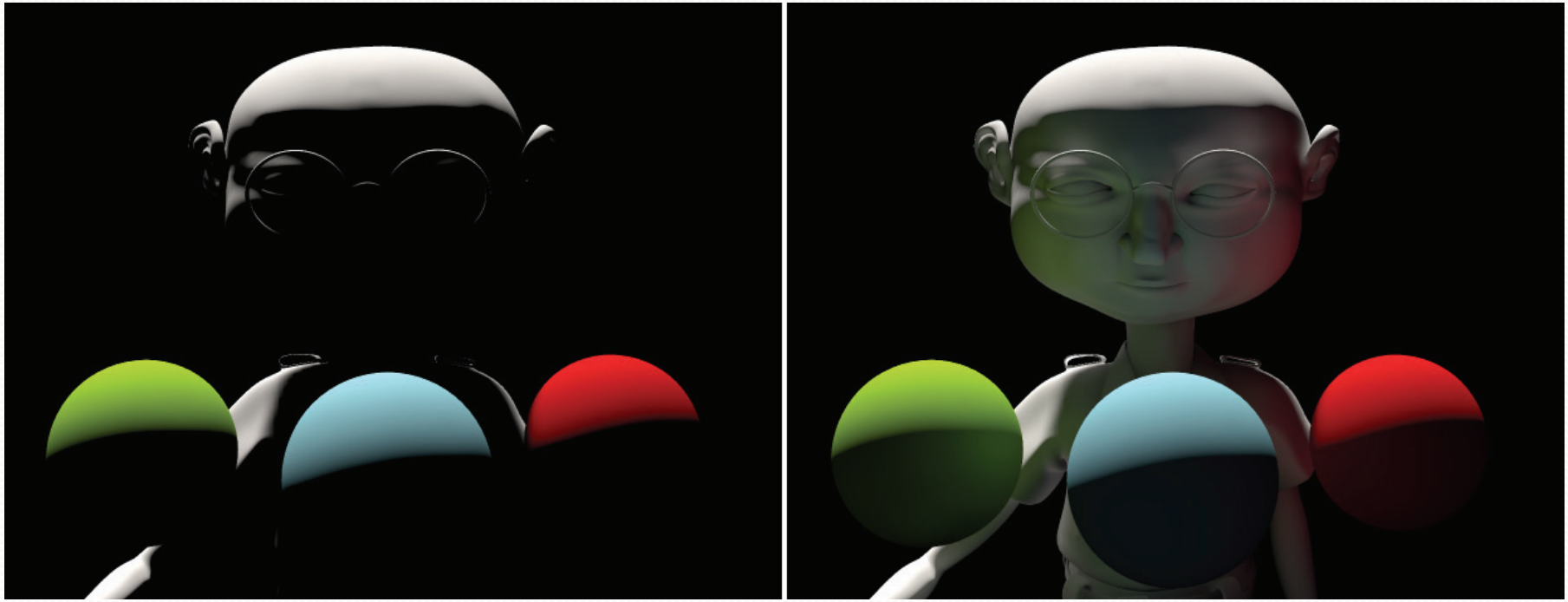
Goals

- 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

Features

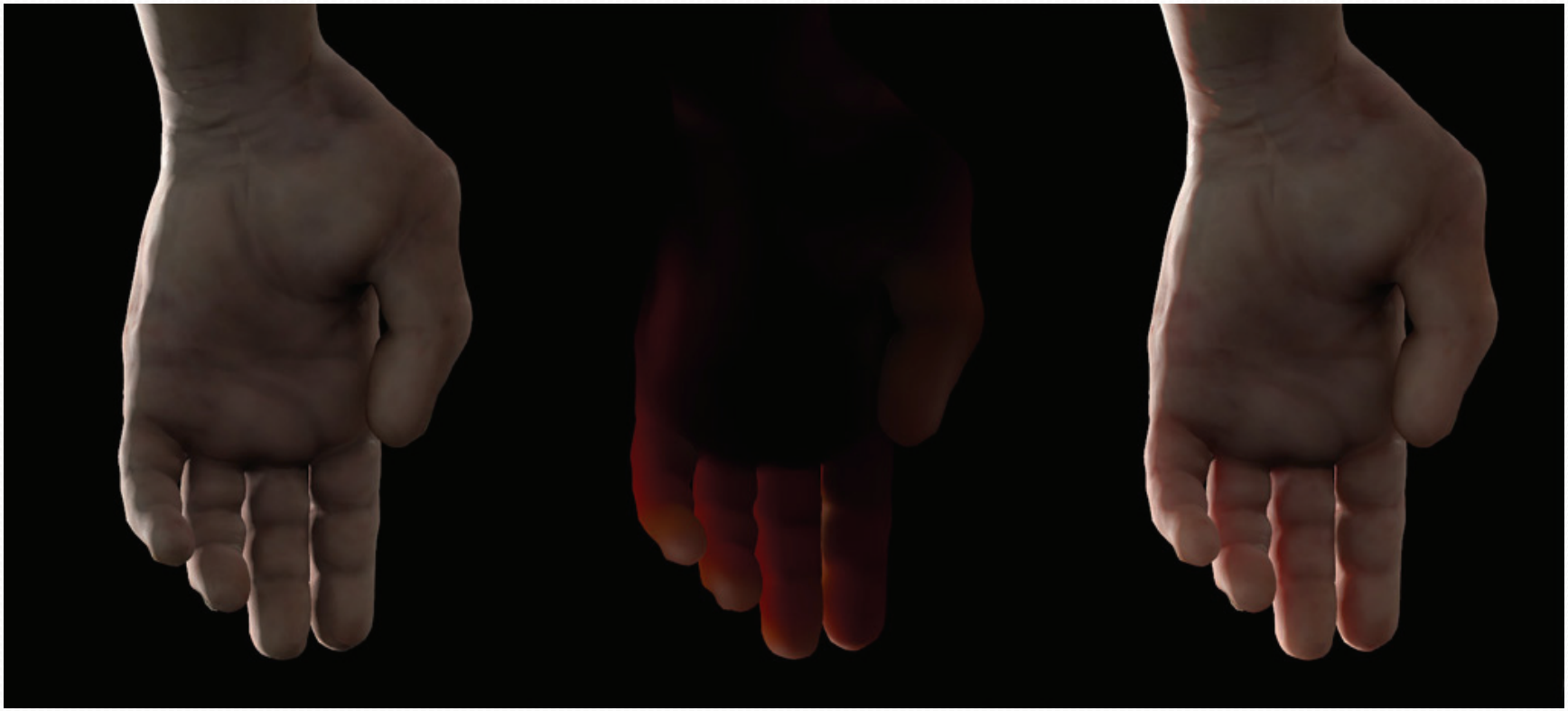
- Indirect lighting (color bleeding)
- Area lights
- Subsurface scattering
  - Sparse sampling works well for very translucent materials (like snow), or light scattered through ears and fingers etc.
  - Good compliment to traditional real-time SSS techniques
    - Blur diffuse contribution in texture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

Indirect Lighting



Area Lights

Subsurface Scattering



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

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

1 Iteration

2 Iterations

3 Iterations

4 Iterations

5 Iterations

6 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

Strategy

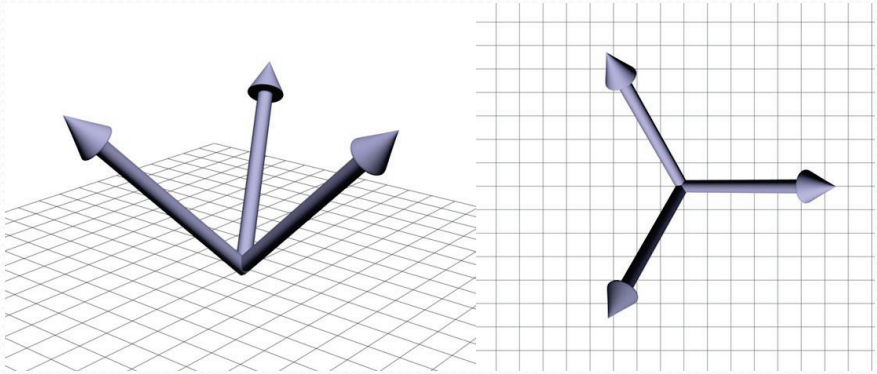
- Used subdivision surfaces
  - Compute irradiance on control mesh faces
  - Tessellator handles attribute interpolation
- Upsample from low poly proxy meshes
  - Compute irradiance at vertices of low poly mesh
  - Interpolate results at render mesh vertices using barycentric coordinates
- Per pixel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

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

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)



Engine Integration

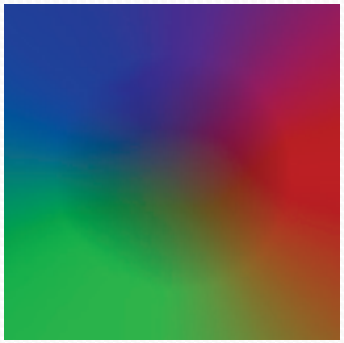
- Treat PBGI solver like a physics simulator
  - Compute sample position, normal, and tangents in world space
  - No geometry instancing
  - No culling (frustum or occlusion)
- Run after animation and physics
- Write 4 colors (3 irradiance, 1 sss) into vertex stream, up-sampling if necessary

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)

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



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

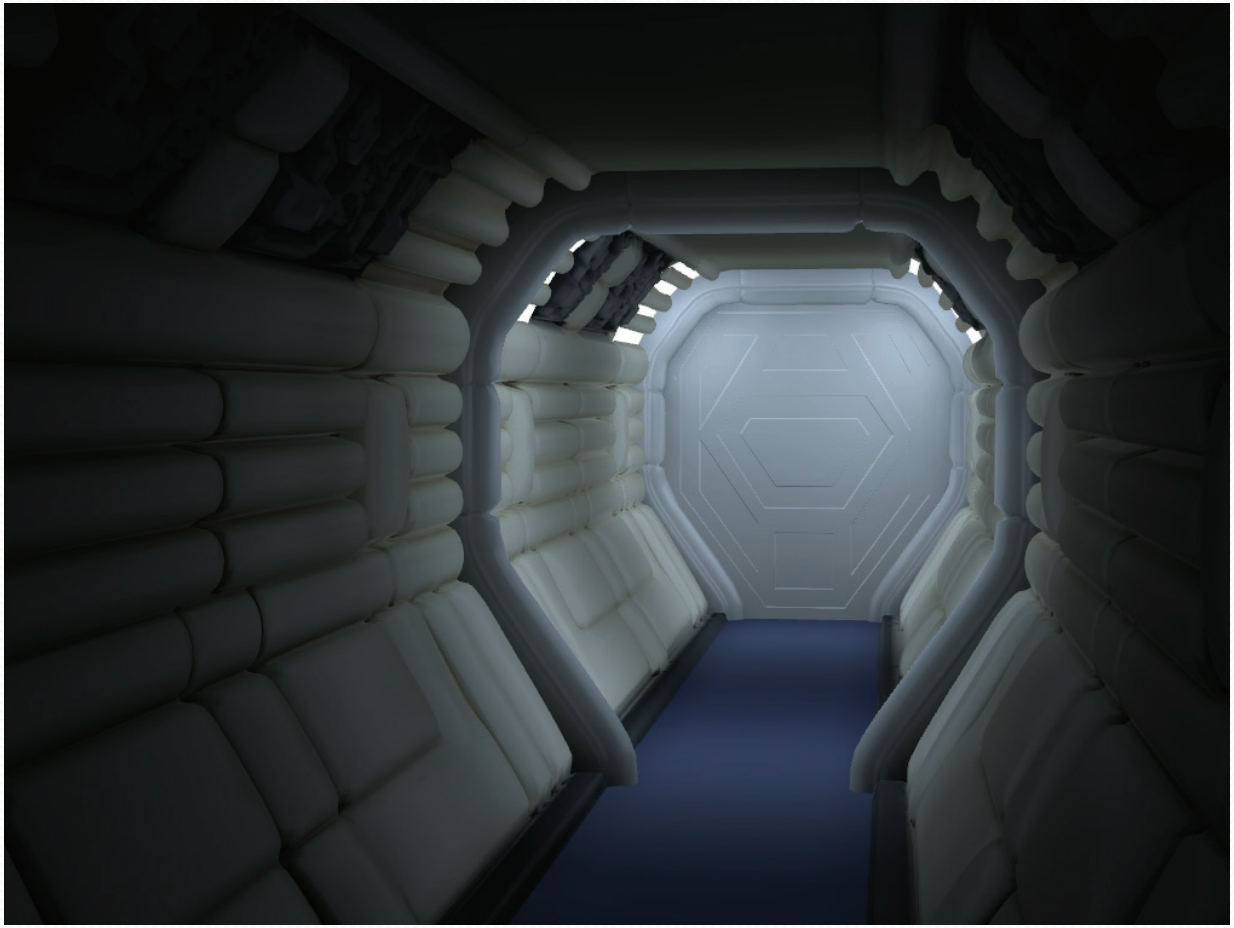
Shader Code

```
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);
```

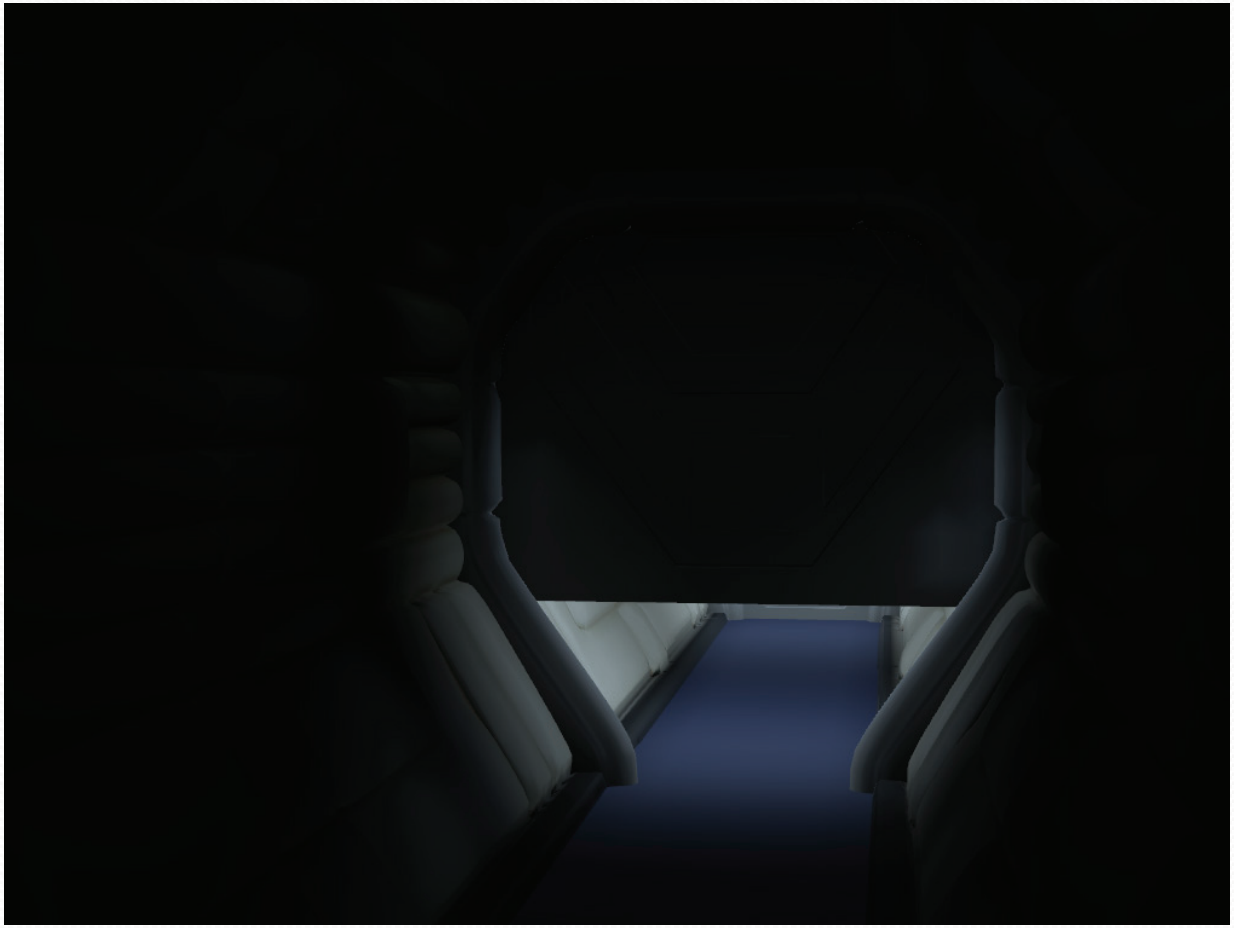
Negative Radiance

- 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

Light Leakage

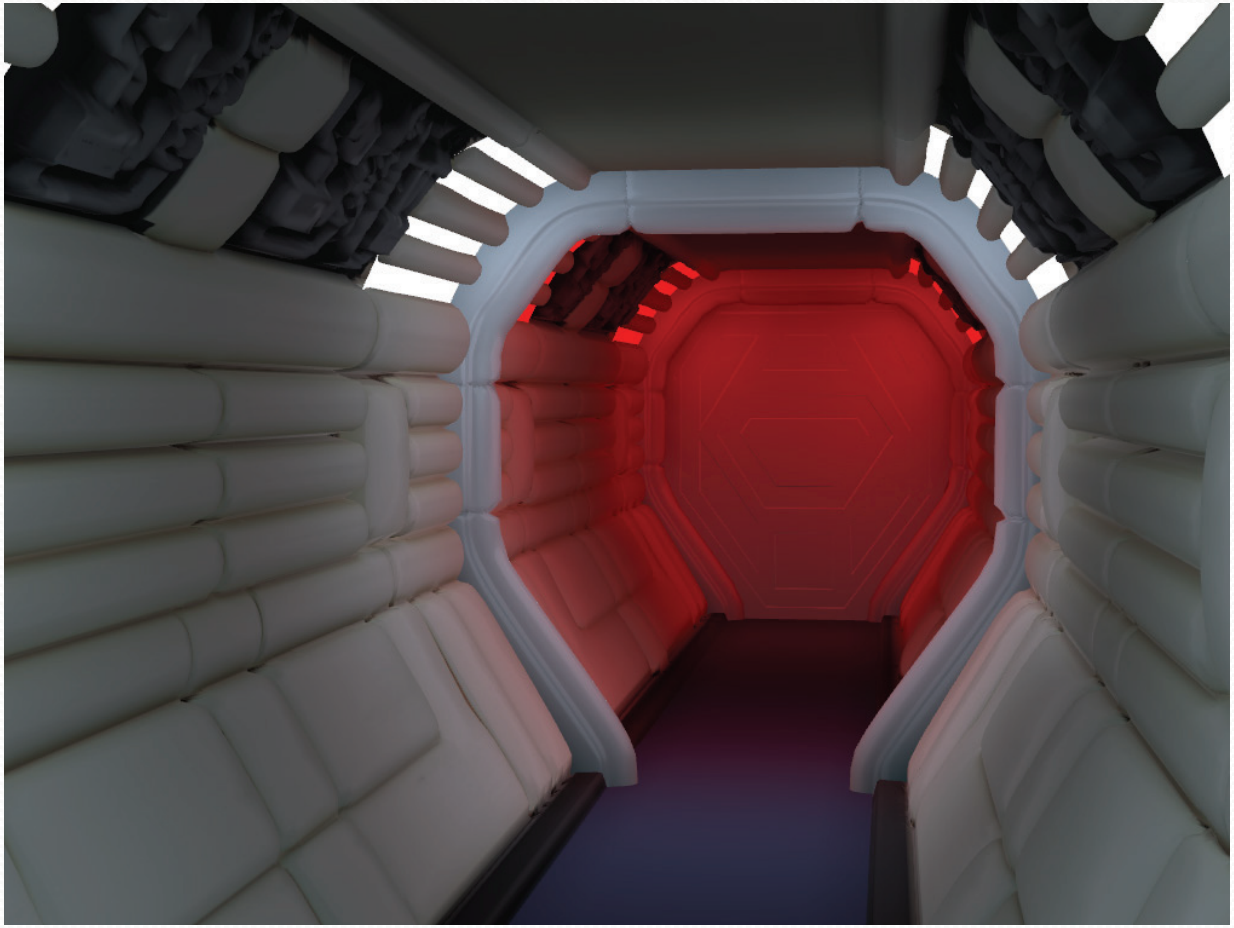


Light Leakage

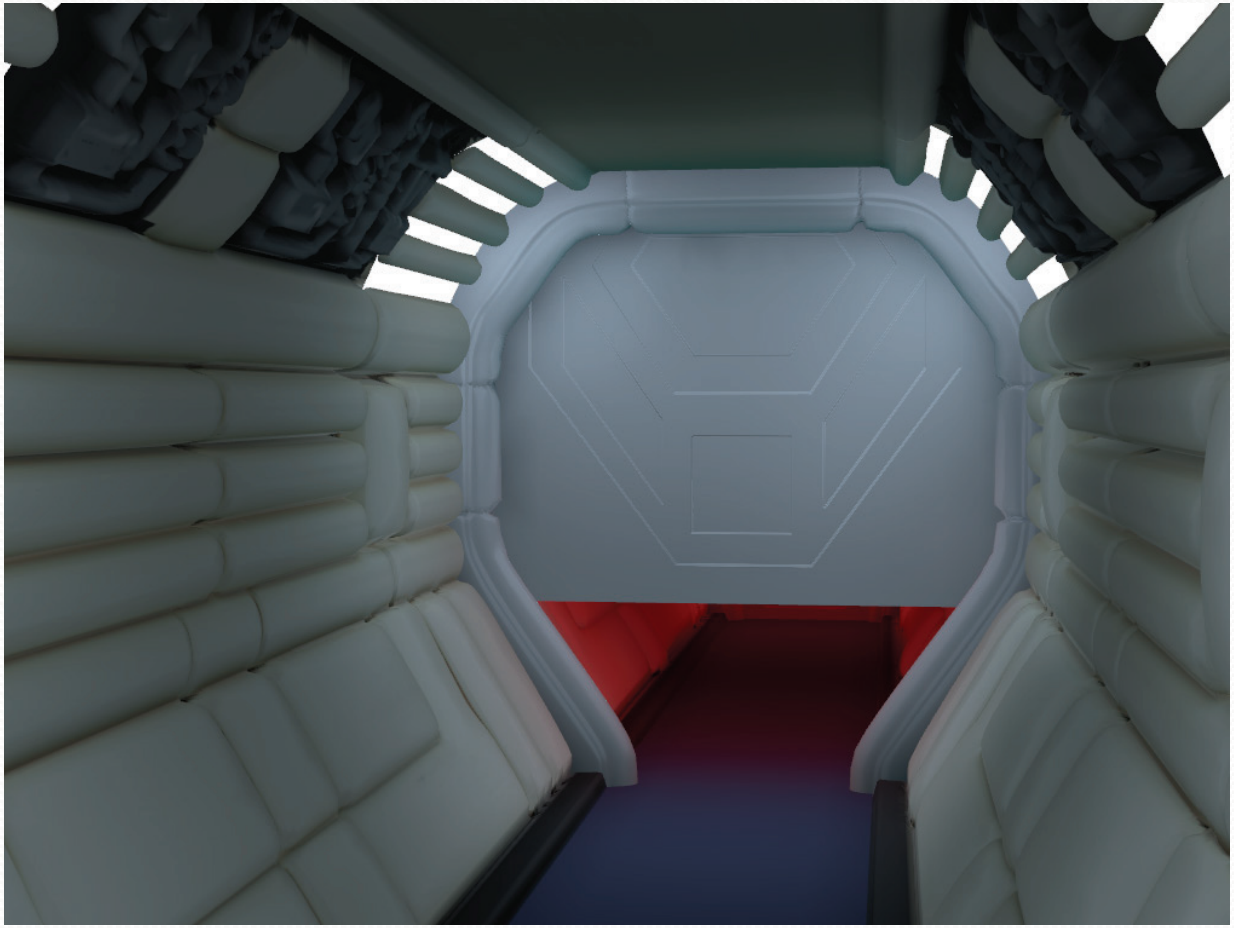


Light Leakage

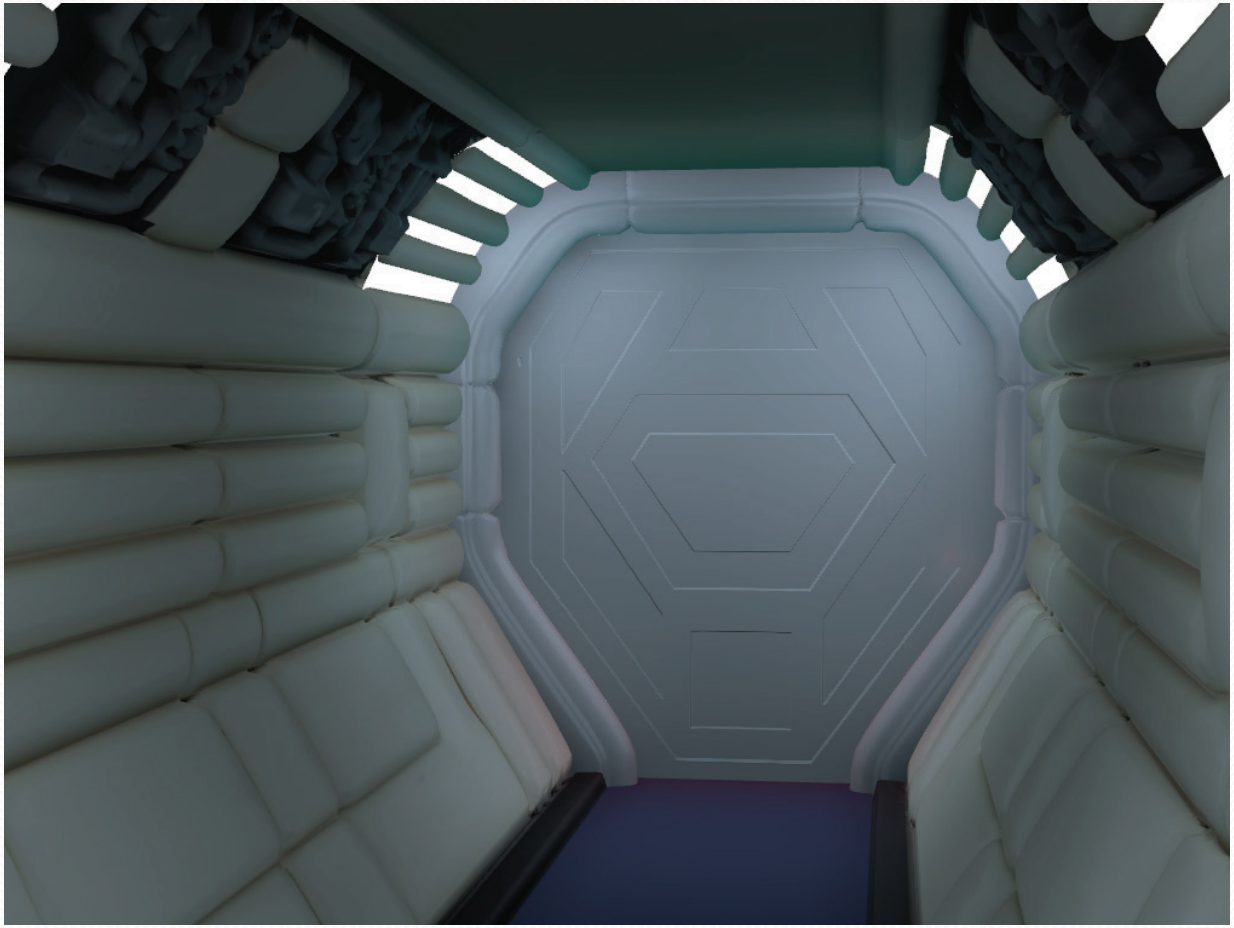
Color Shift



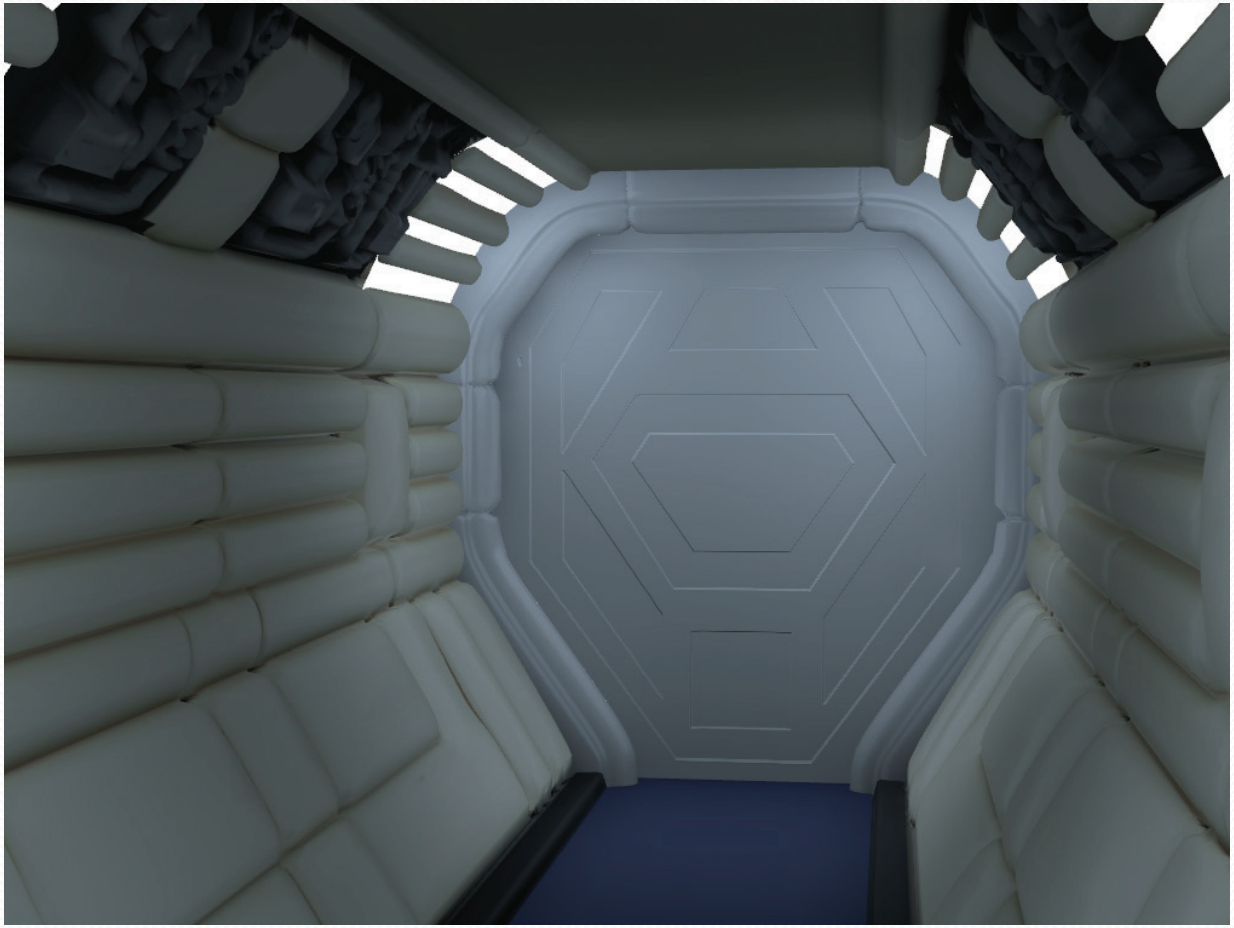
Color Shift



Color Shift



Fix both with Group Dependency



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

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

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

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