

Advanced Graphics



Global Illumination

What's wrong with raytracing?

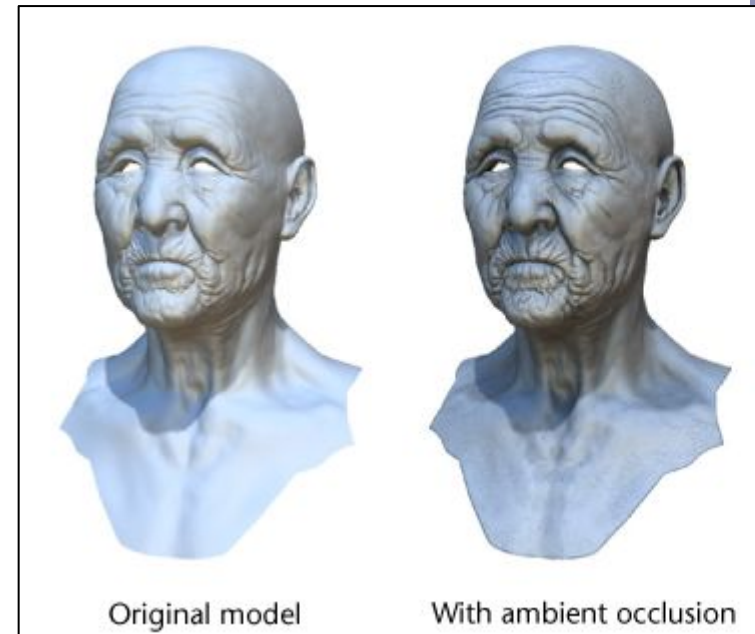
- Soft shadows are expensive
- Shadows of transparent objects require further coding or hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (color bleeding, such as in the Cornell Box—notice how the sides of the inner cubes are shaded red and green.)
- Fundamentally, the ambient term is a hack and the diffuse term is only one step in what should be a recursive, self-reinforcing series.



The *Cornell Box* is a test for rendering Software, developed at Cornell University in 1984 by Don Greenberg. An actual box is built and photographed; an identical scene is then rendered in software and the two images are compared.

Ambient occlusion

- *Ambient illumination* is a blanket constant that we often add to every illuminated element in a scene, to (inaccurately) model the way that light scatters off all surfaces, illuminating areas not in direct lighting.
- *Ambient occlusion* is the technique of adding/removing ambient light when other objects are nearby and scattered light wouldn't reach the surface.
- Computing ambient occlusion is a form of *global illumination*, in which we compute the lighting of scene elements in the context of the scene as a whole.



Ambient occlusion in action



Ambient occlusion in action



Ambient occlusion in action



Ambient occlusion in action



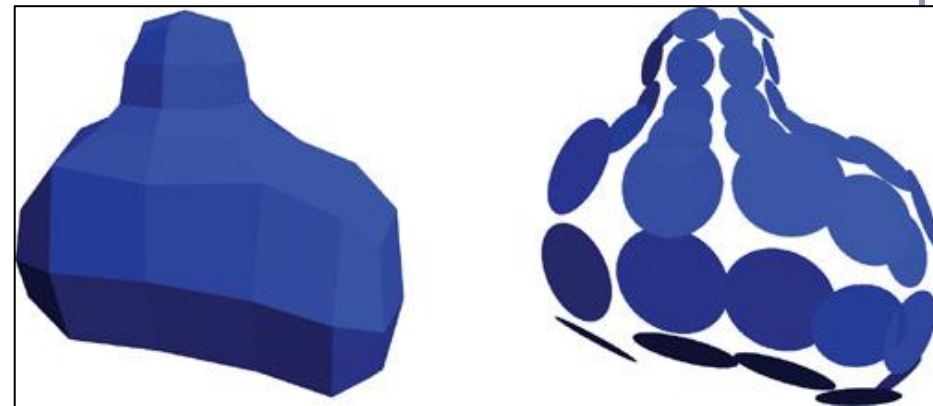
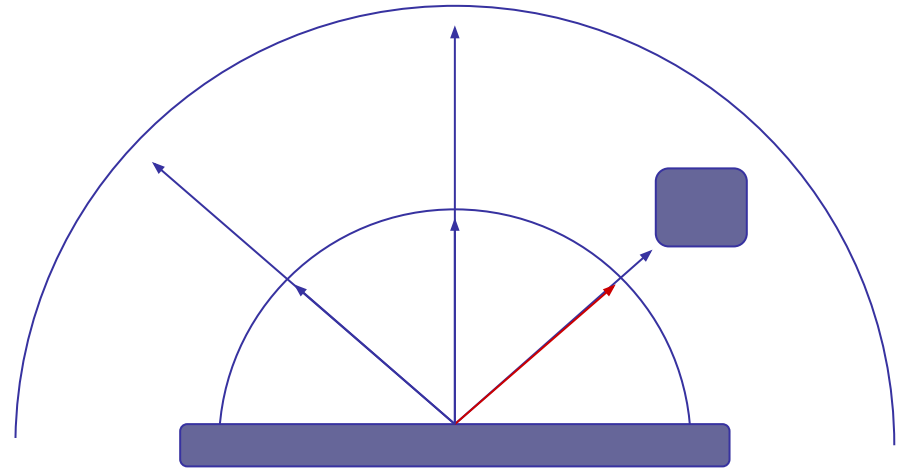
Ambient occlusion - Theory

We can treat the background (the sky) as a vast ambient illumination source.

- For each vertex of a surface, compute how much background illumination reaches the vertex by computing how much sky it can ‘see’
- Integrate occlusion A_p over the hemisphere around the normal at the vertex:

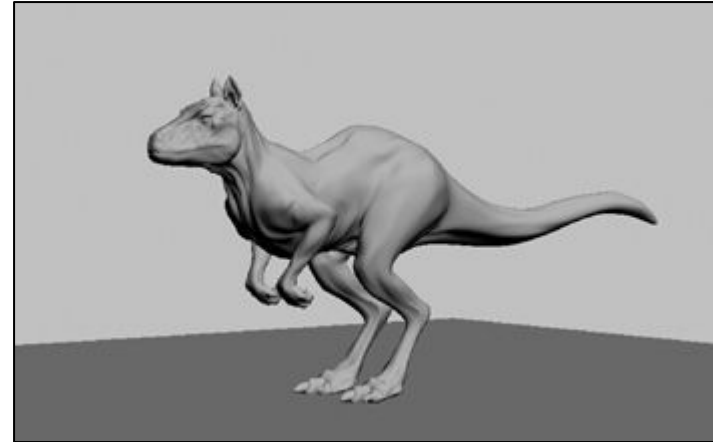
$$A_{\bar{p}} = \frac{1}{\pi} \int_{\Omega} V_{\bar{p}, \hat{\omega}} (\hat{n} \cdot \hat{\omega}) d\omega$$

- A_p occlusion at point p
- n normal at point p
- $V_{p, \omega}$ visibility from p in direction ω
- Ω integrate over area (hemisphere)



Ambient occlusion - Theory

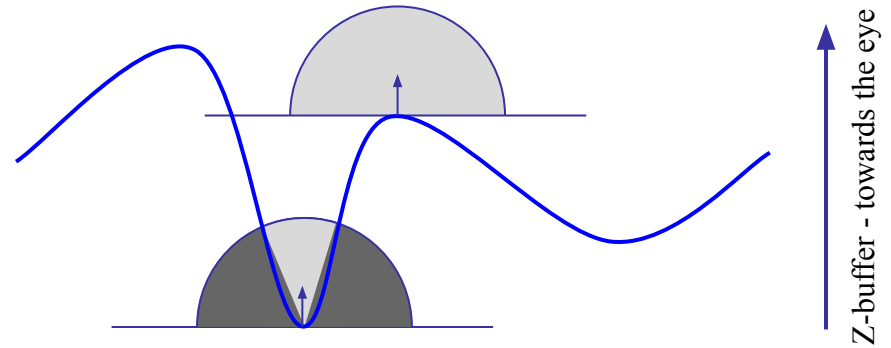
- This approach is very flexible
- Also very expensive!
- To speed up computation, randomly sample rays cast out from each polygon or vertex (this is a *Monte-Carlo* method)
- Alternatively, render the scene from the point of view of each vertex and count the background pixels in the render
- Best used to pre-compute per-object “*occlusion maps*”, texture maps of shadow to overlay onto each object
- But pre-computed maps fare poorly on animated models...



Screen Space Ambient Occlusion ("SSAO")

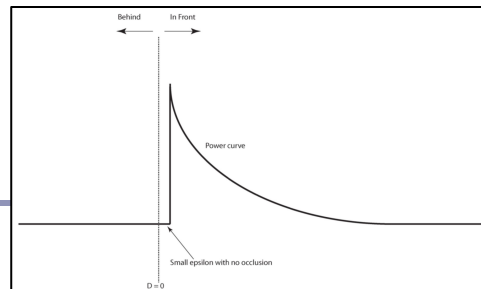
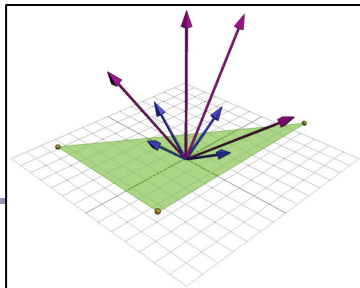
"True ambient occlusion is hard,
let's go hacking."

- Approximate ambient occlusion by comparing z-buffer values in screen space!
- Open plane = unoccluded
- Closed 'valley' in depth buffer = shadowed by nearby geometry
- Multi-pass algorithm
- Runs entirely on the GPU



Screen Space Ambient Occlusion

1. For each visible point on a surface in the scene (ie., each pixel), take multiple samples (typically between 8 and 32) from nearby and map these samples back to screen space
2. Check if the depth sampled at each neighbor is nearer to, or further from, the scene sample point
3. If the neighbor is nearer than the scene sample point then there is some degree of occlusion
 - a. Care must be taken not to occlude if the nearer neighbor is too much nearer than the scene sample point; this implies a separate object, much closer to the camera
4. Sum retained occlusions, weighting with an occlusion function

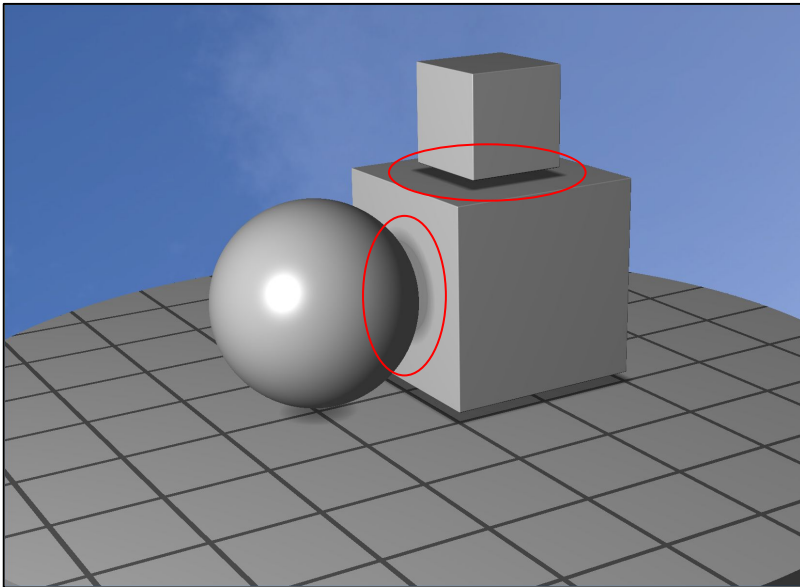


SSAO example- Uncharted 2



4) Low Pass Filter (significant blurring)

Ambient occlusion and Signed Distance Fields

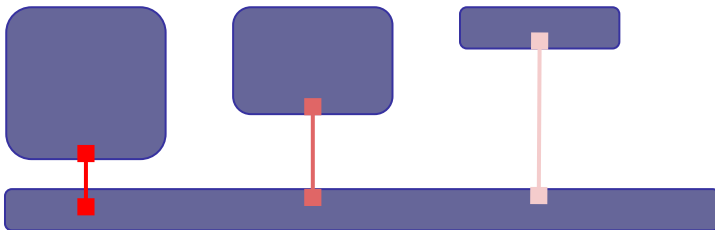


In a nutshell, SSAO tries to estimate occlusion by asking, “how far is it to the nearest neighboring geometry?”

With signed distance fields, this question is almost trivial to answer.

```
float ambient(vec3 pt, vec3 normal) {  
    return abs(getSdf(pt + 0.1 * normal)) / 0.1;  
}
```

```
float ambient(vec3 pt, vec3 normal) {  
    float a = 1;  
    int step = 0;  
  
    for (float t = 0.01; t <= 0.1; ) {  
        float d = abs(getSdf(pt + t * normal));  
        a = min(a, d / t);  
        t += max(d, 0.01);  
    }  
    return a;  
}
```



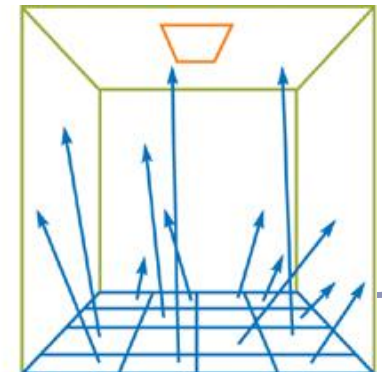
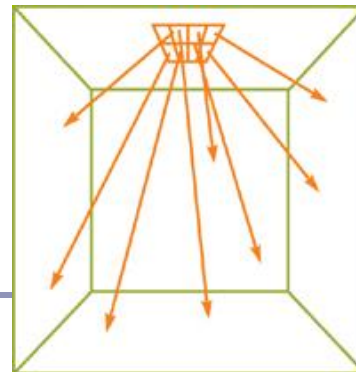
Radiosity

- *Radiosity* is an illumination method which simulates the global dispersion and reflection of diffuse light.
 - First developed for describing spectral heat transfer (1950s)
 - Adapted to graphics in the 1980s at Cornell University
- Radiosity is a finite-element approach to global illumination: it breaks the scene into many small elements (*patches*) and calculates the energy transfer between them.



Radiosity—algorithm

- Surfaces in the scene are divided into *patches*, small subsections of each polygon or object.
- For every pair of patches A, B, compute a *view factor* (also called a *form factor*) describing how much energy from patch A reaches patch B.
 - The further apart two patches are in space or orientation, the less light they shed on each other, giving lower view factors.
- Calculate the lighting of all directly-lit patches.
- Bounce the light from all lit patches to all those they light, carrying more light to patches with higher relative view factors. Repeating this step will distribute the total light across the scene, producing a global diffuse illumination model.



Radiosity—mathematical support

The ‘radiosity’ of a single patch is the amount of energy leaving the patch per discrete time interval.

This energy is the total light being emitted directly from the patch combined with the total light being reflected by the patch:

$$B_i = E_i + R_i \sum_{j=1}^n B_j F_{ij}$$

This forms a system of linear equations, where...

B_i is the radiosity of patch i ;

B_j is the radiosity of each of the other patches ($j \neq i$)

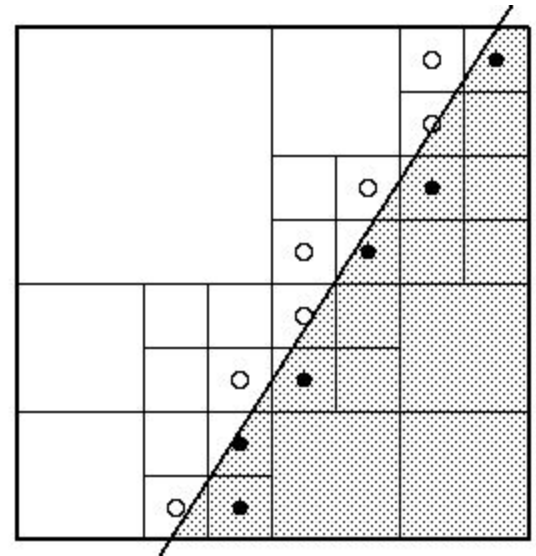
E_i is the emitted energy of the patch

R_i is the reflectivity of the patch

F_{ij} is the view factor of energy from patch i to patch j .

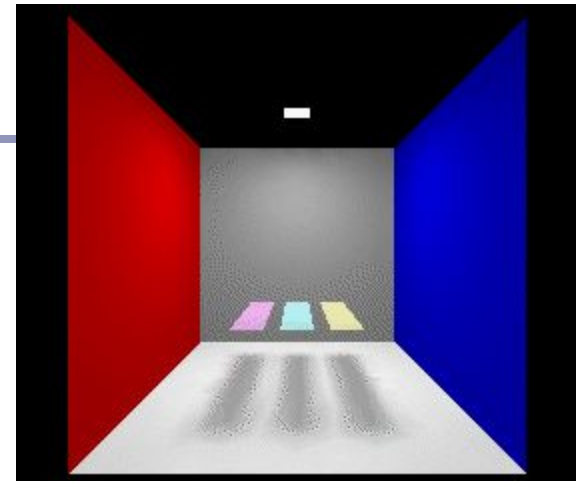
Radiosity—form factors

- Finding form factors can be done procedurally or dynamically
 - Can subdivide every surface into small patches of similar size
 - Can dynamically subdivide wherever the 1st derivative of calculated intensity rises above some threshold.
- Computing cost for a general radiosity solution goes up as the square of the number of patches, so try to keep patches down.
 - Subdividing a large flat white wall could be a waste.
- Patches should ideally closely align with lines of shadow.

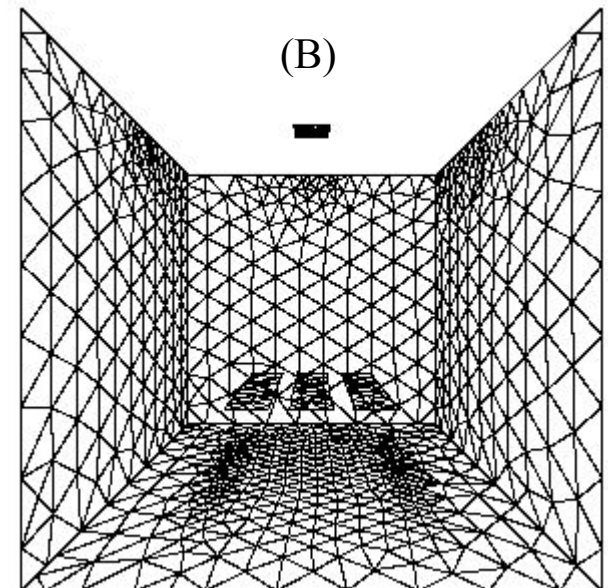
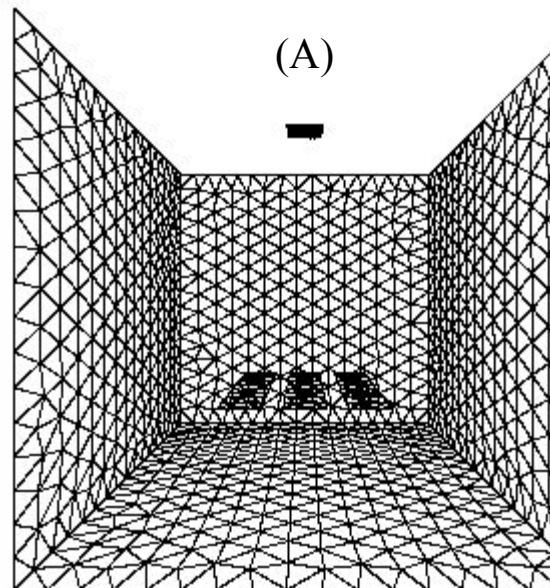


Radiosity—implementation

- (A) Simple patch triangulation
- (B) Adaptive patch generation: the floor and walls of the room are dynamically subdivided to produce more patches where shadow detail is higher.



Images from “Automatic generation of node spacing function”, IBM (1998)
<http://www.trl.ibm.com/projects/meshing/nsp/nspE.htm>

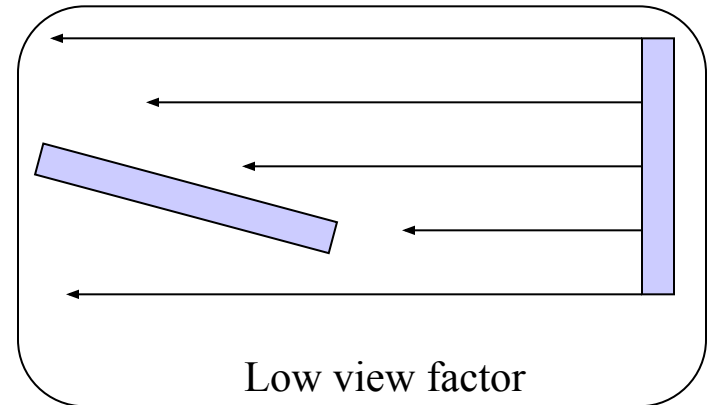
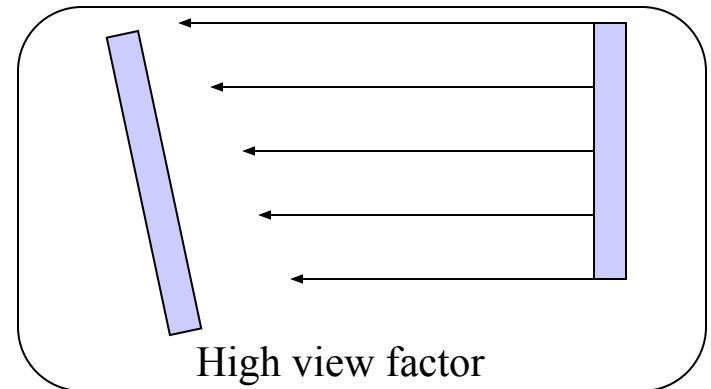
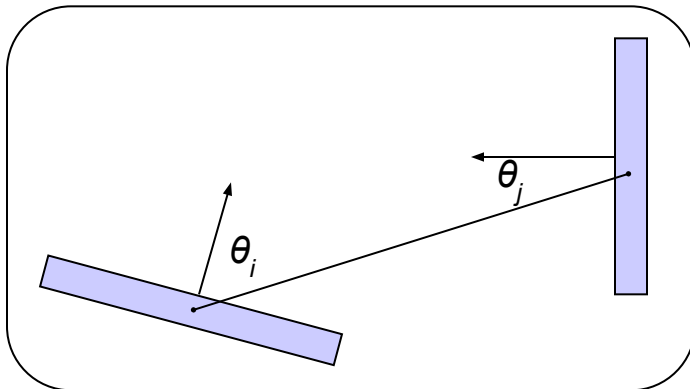


Radiosity—view factors

One equation for the view factor between patches i, j is:

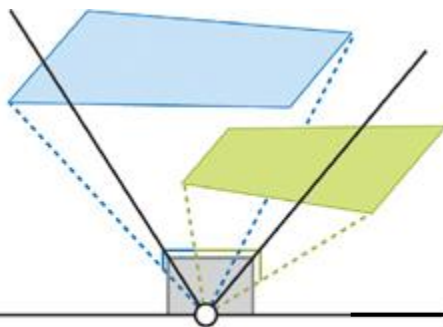
$$F_{i \rightarrow j} = \frac{\cos \theta_i \cos \theta_j}{\pi r^2} V(i, j)$$

...where θ_i is the angle between the normal of patch i and the line to patch j , r is the distance and $V(i, j)$ is the visibility from i to j (0 for occluded, 1 for clear line of sight.)

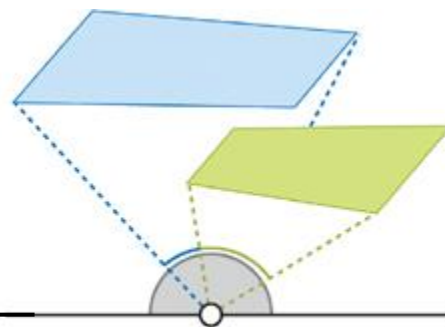


Radiosity—calculating visibility

- Calculating $V(i,j)$ can be slow.
- One method is the *hemicube*, in which each form factor is encased in a half-cube. The scene is then ‘rendered’ from the point of view of the patch, through the walls of the hemicube; $V(i,j)$ is computed for each patch based on which patches it can see (and at what percentage) in its hemicube.
- A purer method, but more computationally expensive, uses hemispheres.



Hemicube Projection



Hemispherical Projection

Note: This method can be accelerated using modern graphics hardware to render the scene. The scene is ‘rendered’ with flat lighting, setting the ‘color’ of each object to be a pointer to the object in memory.

Radiosity gallery



Image from *A Two Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*, John R. Wallace, Michael F. Cohen and Donald P. Greenberg (Cornell University, 1987)



Image from *GPU Gems II*, nVidia



Teapot (wikipedia)

Shadows, refraction and caustics

- Problem: shadow ray strikes transparent, refractive object.
 - Refracted shadow ray will now miss the light.
 - This destroys the validity of the boolean shadow test.
- Problem: light passing through a refractive object will sometimes form *caustics* (right), artifacts where the envelope of a collection of rays falling on the surface is bright enough to be visible.



This is a photo of a real pepper-shaker.
Note the caustics to the left of the shaker, in and outside of its shadow.

Photo credit: Jan Zankowski

Shadows, refraction and caustics

- Solutions for shadows of transparent objects:
 - Backwards ray tracing (Arvo)
 - *Very* computationally heavy
 - Improved by stencil mapping (Shenya et al)
 - Shadow attenuation (Pierce)
 - Low refraction, no caustics
- More general solution:
 - *Photon mapping* (Jensen)→



Photon mapping

Photon mapping is the process of emitting photons into a scene and tracing their paths probabilistically to build a *photon map*, a data structure which describes the illumination of the scene independently of its geometry.

This data is then combined with ray tracing to compute the global illumination of the scene.



Image by Henrik Jensen (2000)

Photon mapping—algorithm (1/2)

Photon mapping is a two-pass algorithm:

1. Photon scattering

- A. Photons are fired from each light source, scattered in randomly-chosen directions. The number of photons per light is a function of its surface area and brightness.
- B. Photons fire through the scene (re-use that raytracer, folks.) Where they strike a surface they are either absorbed, reflected or refracted.
- C. Wherever energy is absorbed, cache the location, direction and energy of the photon in the *photon map*. The photon map data structure must support fast insertion and fast nearest-neighbor lookup; a *kd-tree*¹ is often used.

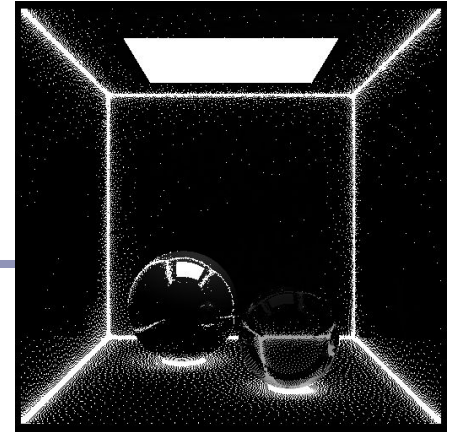


Image by Zack Waters

Photon mapping—algorithm (2/2)

Photon mapping is a two-pass algorithm:

2. Rendering

- A. Ray trace the scene from the point of view of the camera.
- B. For each first contact point P use the ray tracer for specular but compute diffuse from the photon map and do away with ambient completely.
- C. Compute radiant illumination by summing the contribution along the eye ray of all photons within a sphere of radius r of P .
- D. Caustics can be calculated directly here from the photon map. For speed, the caustic map is usually distinct from the radiance map.

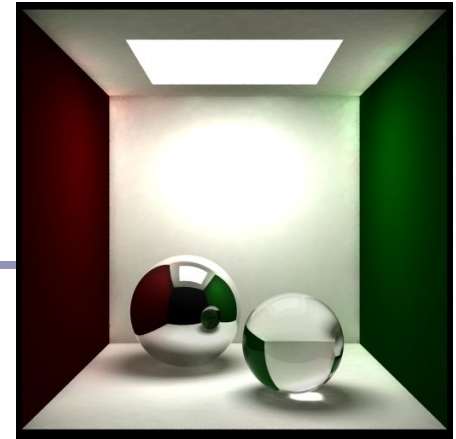
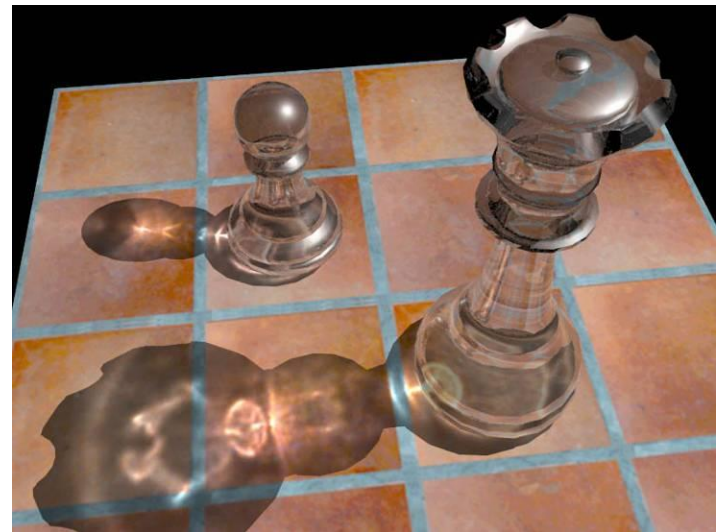


Image by Zack Waters

Photon mapping is probabilistic

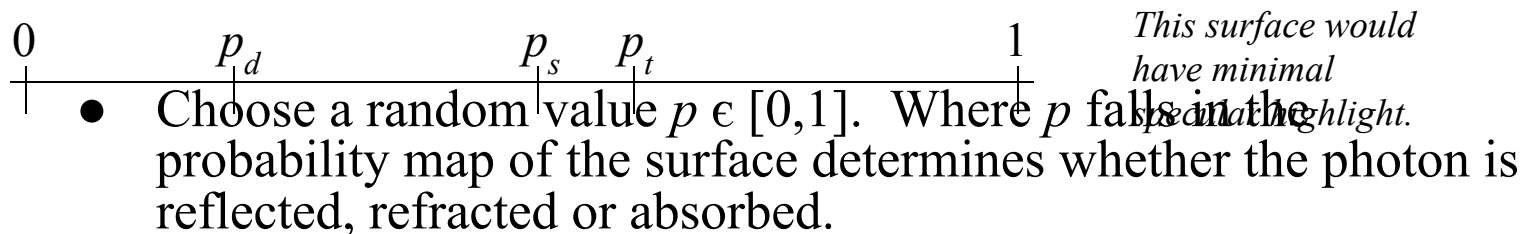
This method is a great example of *Monte Carlo integration*, in which a difficult integral (the lighting equation) is simulated by randomly sampling values from within the integral's domain until enough samples average out to about the right answer.

- This means that you're going to be firing *millions* of photons. Your data structure is going to have to be very space-efficient.

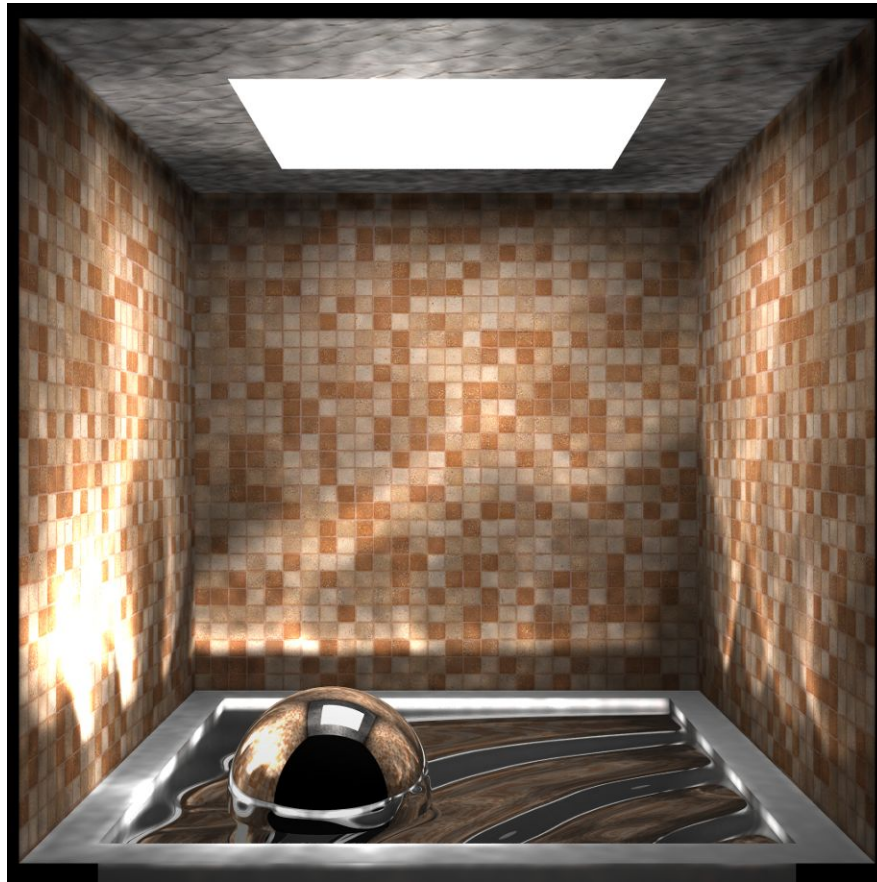


Photon mapping is probabilistic

- Initial photon direction is random. Constrained by light shape, but random.
- What exactly happens each time a photon hits a solid also has a random component:
 - Based on the diffuse reflectance, specular reflectance and transparency of the surface, compute probabilities p_d , p_s and p_t where $(p_d + p_s + p_t) \leq 1$. This gives a probability map:



Photon mapping gallery



http://web.cs.wpi.edu/~emmanuel/courses/cs563/writes_ups/zackw/photon_mapping/PhotonMapping.html



<http://graphics.ucsd.edu/~henrik/images/global.html>



<http://www.pbrt.org/gallery.php>

References

Shirley and Marschner, “Fundamentals of Computer Graphics”, Chapter 24 (2009)

Ambient occlusion and SSAO

- “GPU Gems 2”, nVidia, 2005. Vertices mapped to illumination.
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter14.html
- MITTRING, M. 2007. Finding Next Gen – CryEngine 2.0, Chapter 8, SIGGRAPH 2007 Course 28 – Advanced Real-Time Rendering in 3D Graphics and Games, Siggraph 2007, San Diego, CA, August 2007.
http://developer.amd.com/wordpress/media/2012/10/Chapter8-Mittring-Finding_NextGen_CryEngine2.pdf
- John Hable’s presentation at GDC 2010, “Uncharted 2: HDR Lighting” (filmicgames.com/archives/6)

Radiosity

- nVidia: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter39.html
- Cornell: <http://www.graphics.cornell.edu/online/research/>
- Wallace, J. R., K. A. Elmquist, and E. A. Haines. 1989, “A Ray Tracing Algorithm for Progressive Radiosity.” In *Computer Graphics (Proceedings of SIGGRAPH 89)* 23(4), pp. 315–324.
- Buss, “3-D Computer Graphics: A Mathematical Introduction with OpenGL” (Chapter XI), Cambridge University Press (2003)

Photon mapping

- Henrik Jensen, “Global Illumination using Photon Maps”: <http://graphics.ucsd.edu/~henrik/>
- Henrik Jensen, “Realistic Image Synthesis Using Photon Mapping”
- Zack Waters, “Photon Mapping”:
http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html