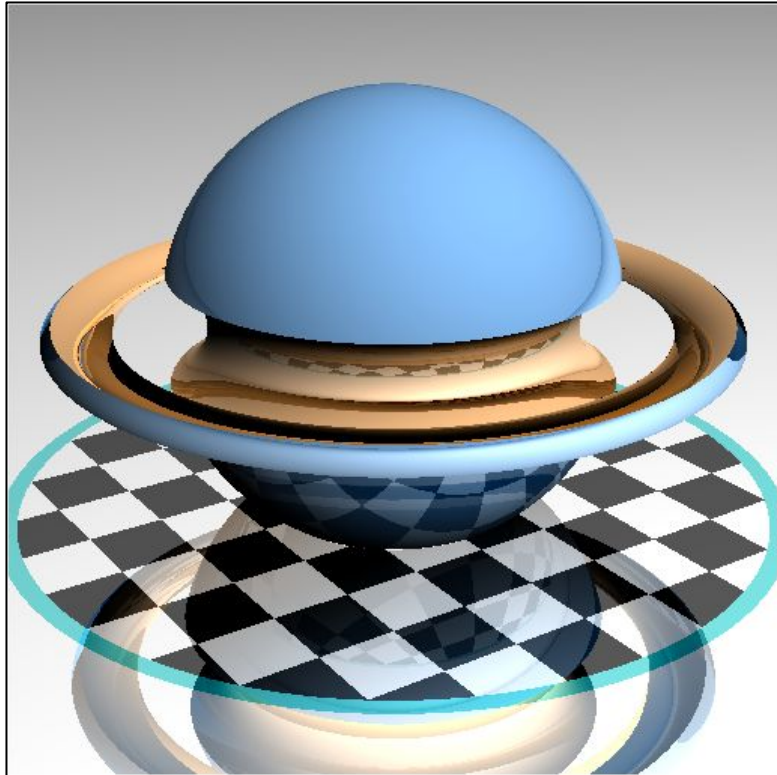


# Advanced Graphics

---



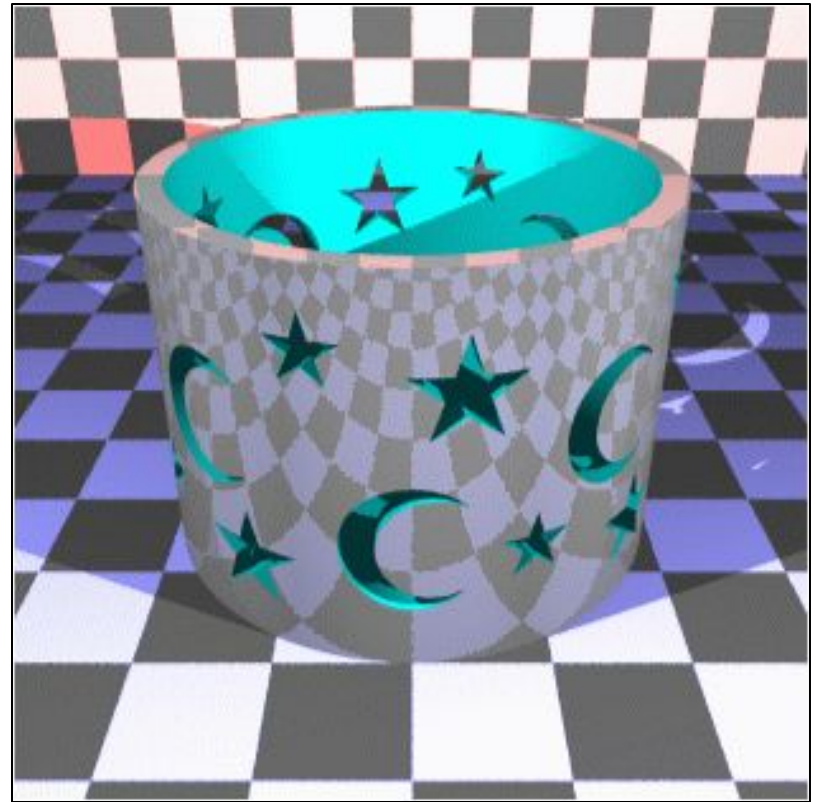
## *Advanced Scenes and Global Illumination*

# Constructive Solid Geometry

---

*Constructive Solid Geometry* (CSG) builds complicated forms out of simple primitives.

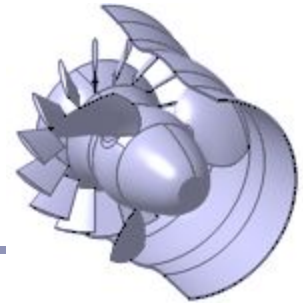
These primitives are combined with basic boolean operations: add, subtract, intersect.



CSG figure by Neil Dodgson

# Constructive Solid Geometry

---



CSG models are easy to ray-trace but difficult to polygonalize

- Issues include choosing polygon boundaries at edges; converting adequately from pure smooth primitives to discrete (flat) faces; handling ‘infinitely thin’ sheet surfaces; and others.
- This is an ongoing research topic.

CSG models are well-suited to machine milling, automated manufacture, etc

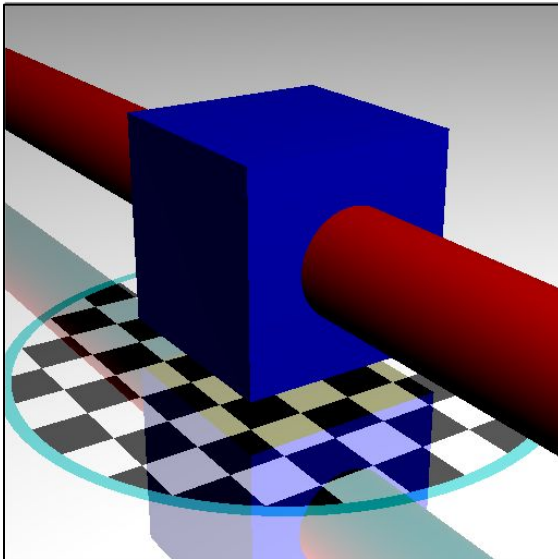
- Great for 3D printers!

# Constructive Solid Geometry

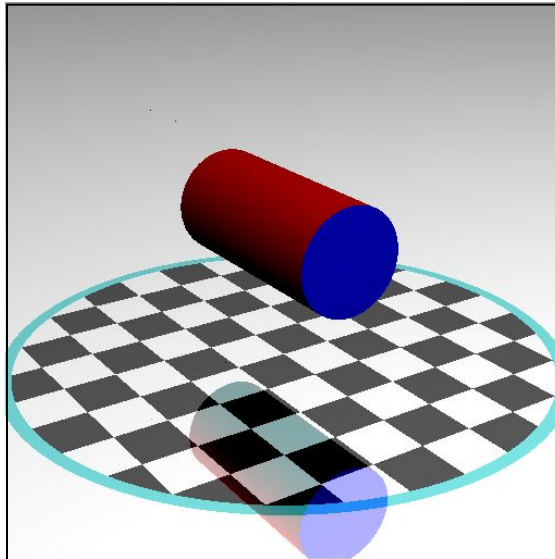
---

Three operations:

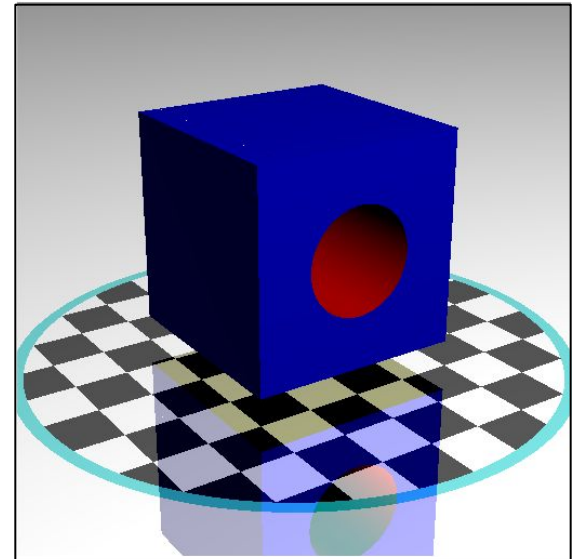
1. *Union*



2. *Intersection*



3. *Difference*



# Constructive Solid Geometry

---

CSG surfaces can be described by a binary tree, where each leaf node is a primitive and each non-leaf node is a boolean operation.

(What would the *not* of a surface look like?)

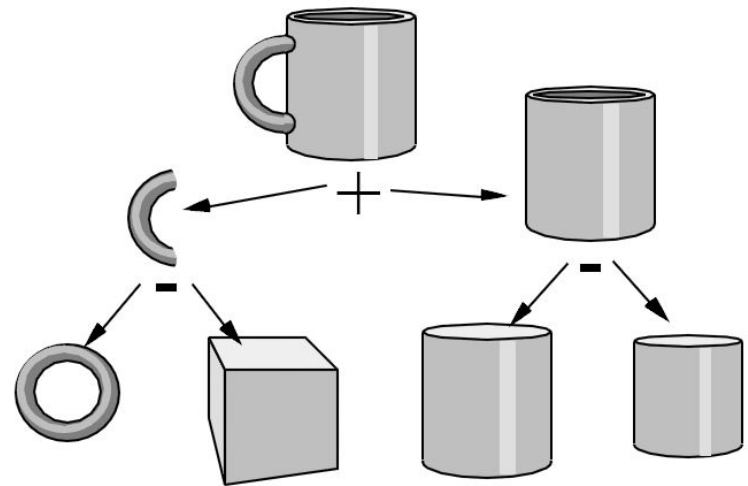


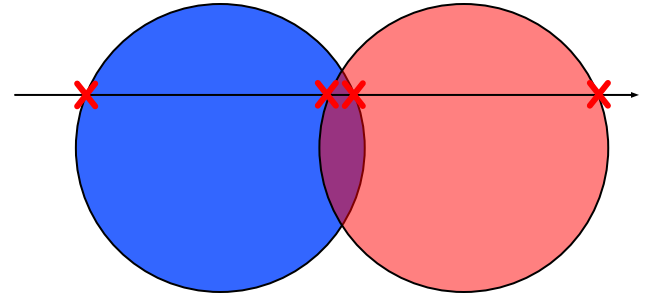
Figure from Wyvill (1995) part two, p. 4

# Constructive Solid Geometry

---

For each node of the binary tree:

- Fire ray  $r$  at  $A$  and  $B$ .
- List in  $t$ -order all points where  $r$  enters of leaves  $A$  or  $B$ .
  - You can think of each intersection as a quad of booleans--  
( $wasInA$ ,  $isInA$ ,  $wasInB$ ,  $isInB$ )
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

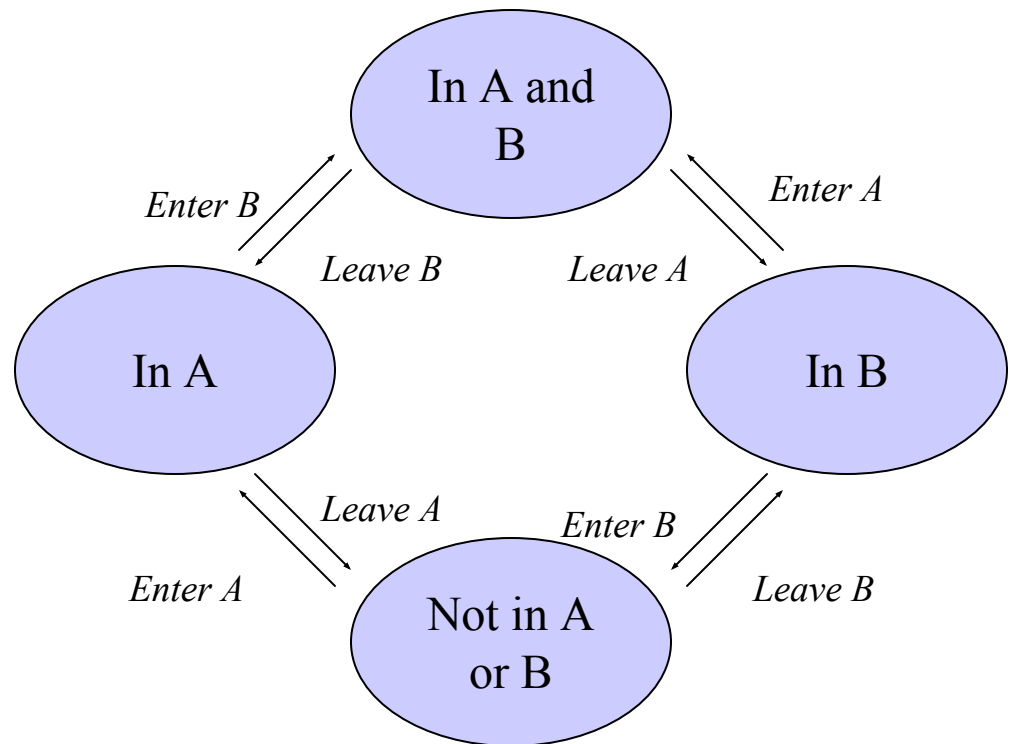


# Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

For each operation, retain those intersections that transition into or out of the critical state(s).

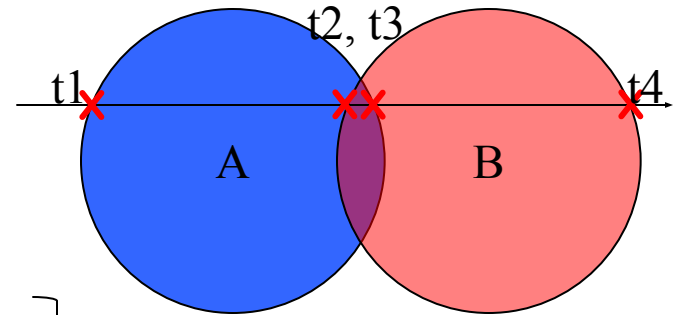
- Union: {In A | In B | In A and B}
- Intersection: {In A and B}
- Difference: {In A}



# Ray-tracing CSG models

## Example: Difference (A-B)

A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No

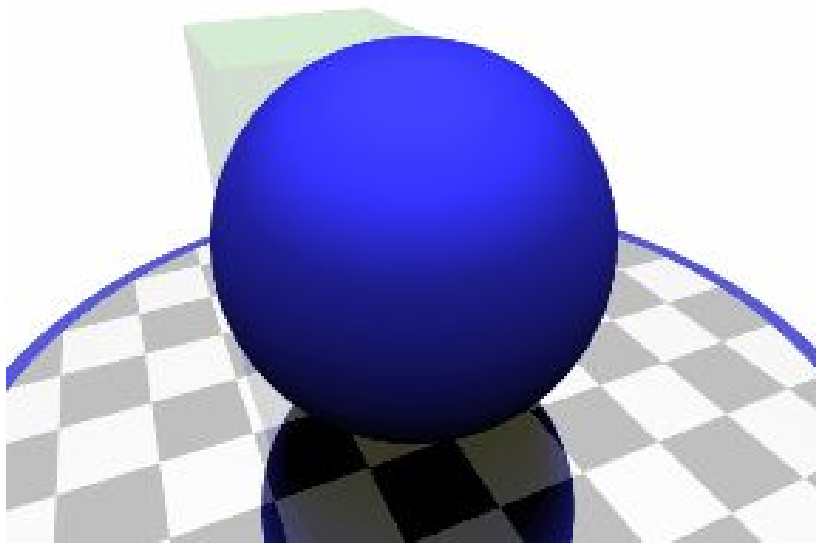


```
difference =  
( (wasInA != isInA) &&  
  (!isInB) && (!wasInB) )  
||  
( (wasInB != isInB) &&  
  (wasInA || isInA) )
```

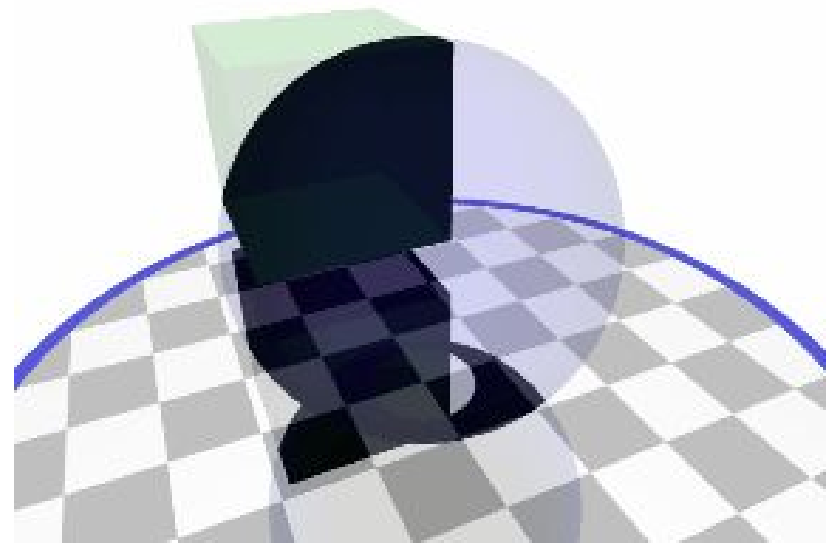


# CSG in action

---



Difference



Intersection

# 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.

# Radiosity

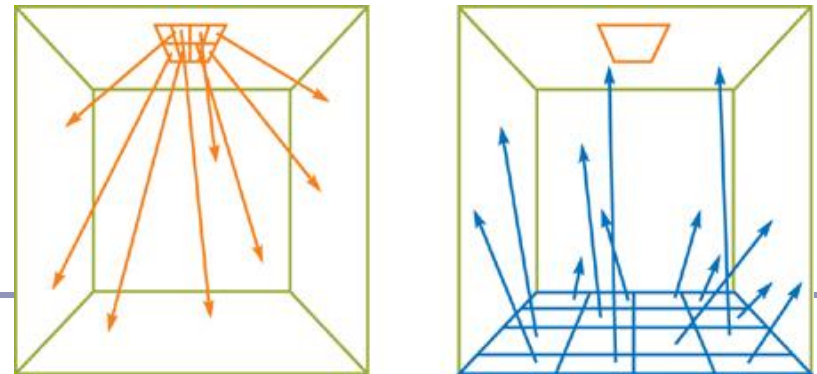
- *Radiosity* is an illumination method which simulates the global dispersion and reflection of diffuse light.
  - First developed for describing spectral heat transfer (1950s)
  - Adapted to graphics in the 1980s at Cornell University
- Radiosity is a finite-element approach to global illumination: 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 *form factors* (also called *patches*), small subsections of each polygon or object.
- For every pair of form factors A, B, compute a *view 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 total 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:

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

where...

$B_i$  is the radiosity of patch  $i$ ;

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

$E_i$  is the emitted energy of the patch

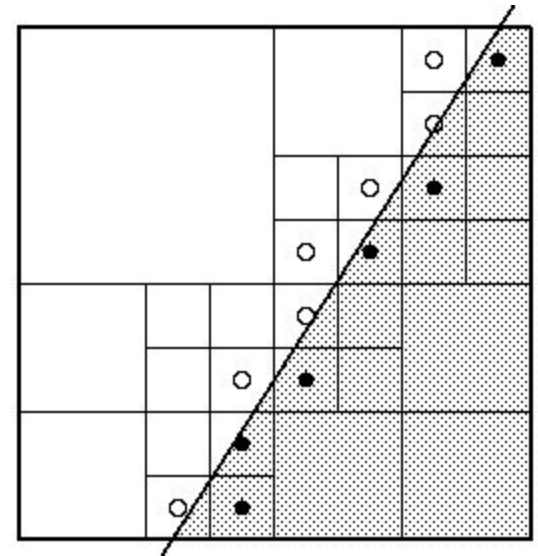
$R_i$  is the reflectivity of the patch

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

# Radiosity—form factors

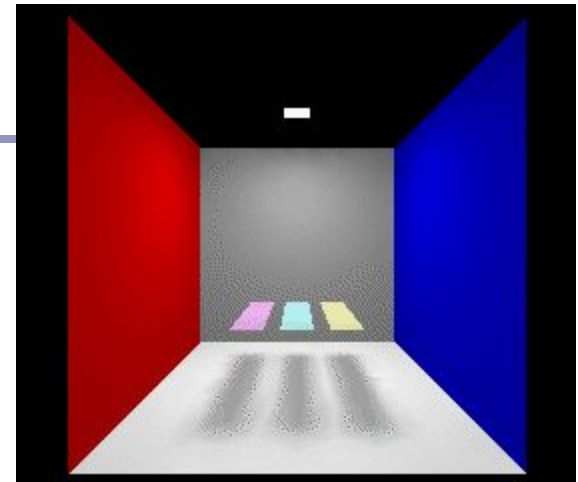
---

- Finding form factors can be done procedurally or dynamically
  - Can subdivide every surface into small patches of similar size
  - Can dynamically subdivide wherever the 1<sup>st</sup> 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.

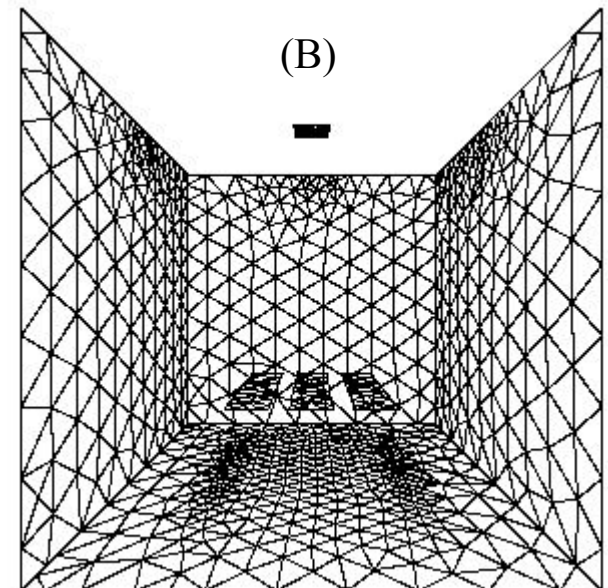
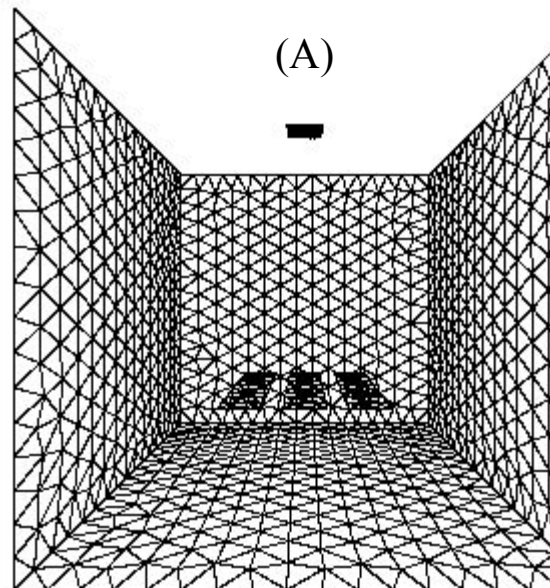


# Radiosity—implementation

- (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>

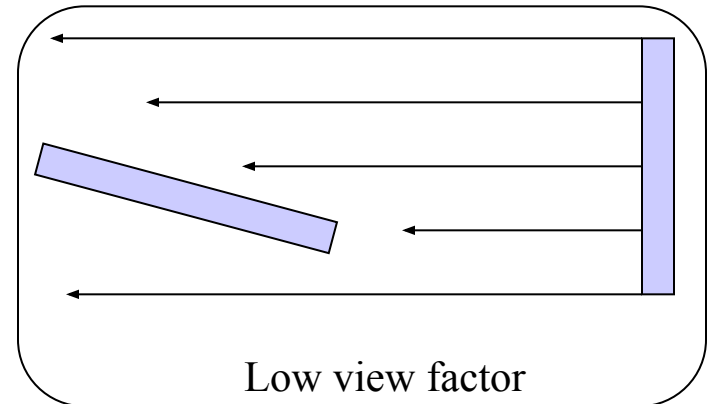
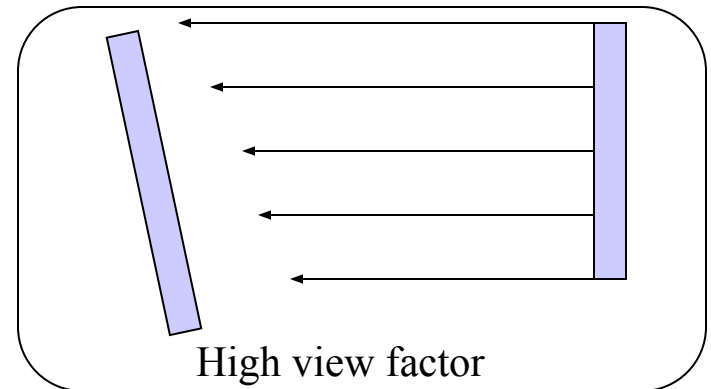
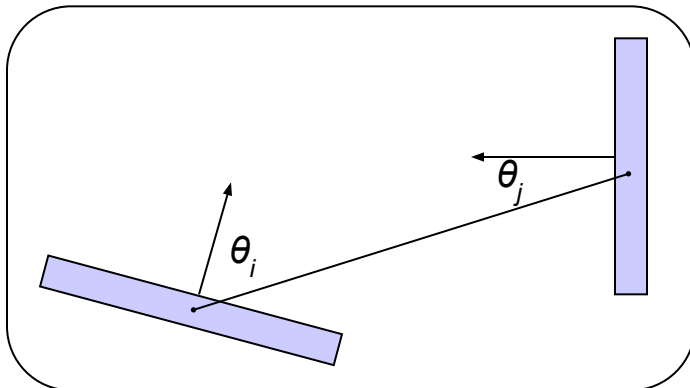


# Radiosity—view factors

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  $\theta_i$  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.)

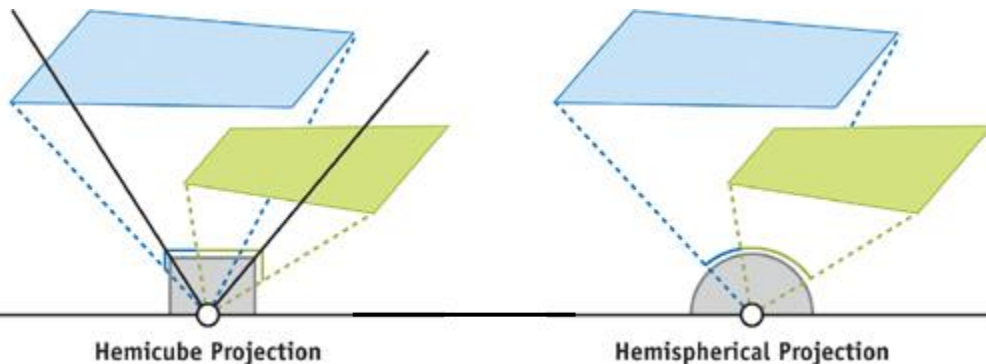




# Radiosity—calculating visibility

---

- 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.



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.



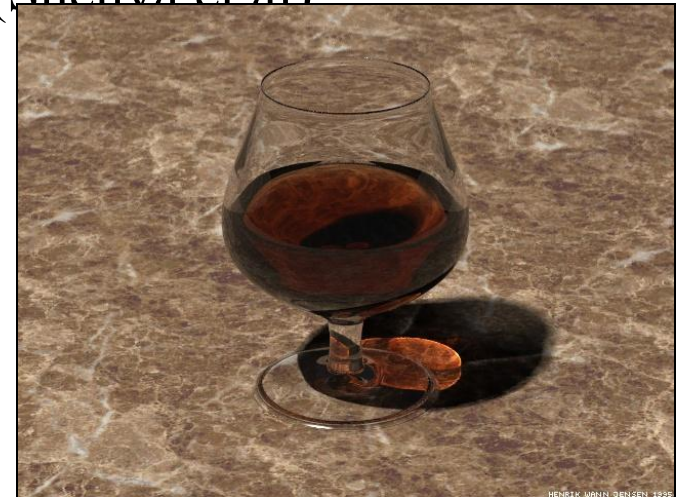
This is a photo of a real pepper-shaker.  
Note the caustics to the left of the shaker, in and outside of its shadow.

*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)

---

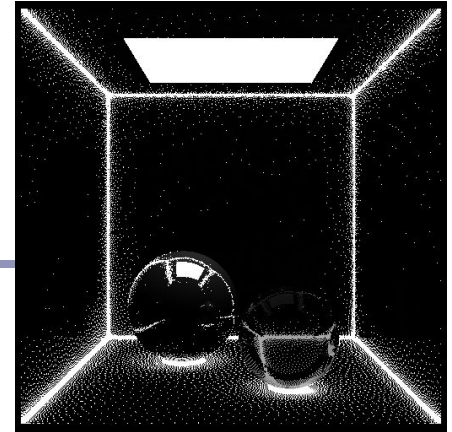


Image by Zack Waters

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*<sup>1</sup> is often used.



## 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.

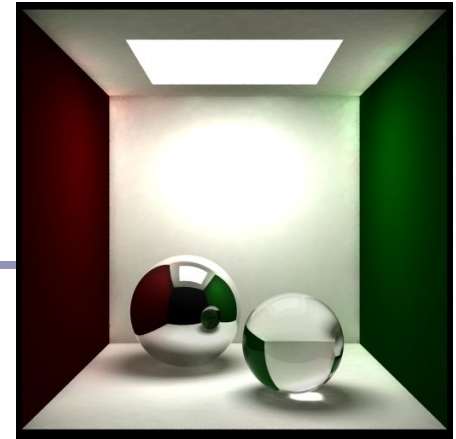


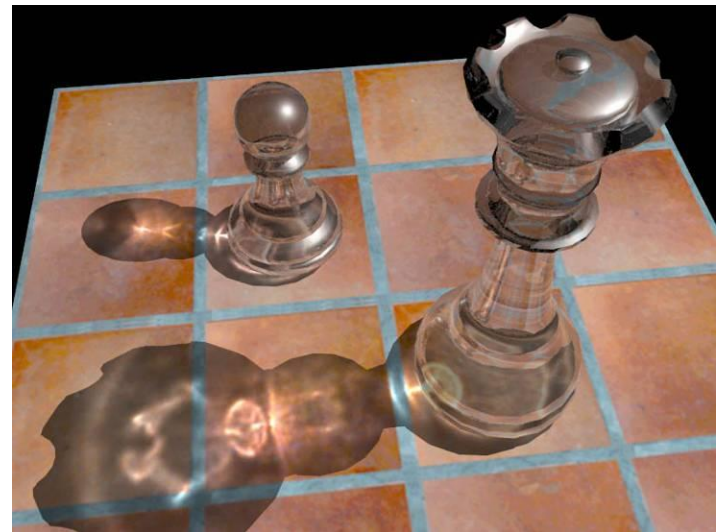
Image by Zack Waters

# 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.



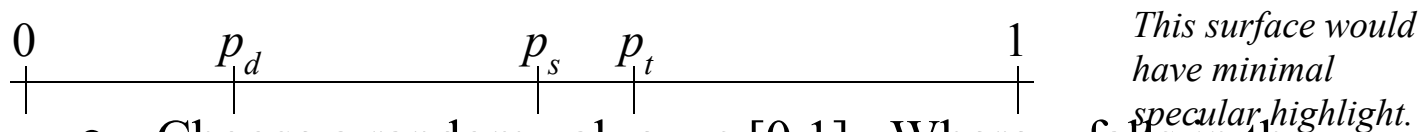
[http://www.okino.com/conv/imp\\_jt.htm](http://www.okino.com/conv/imp_jt.htm)



# Photon mapping is probabilistic

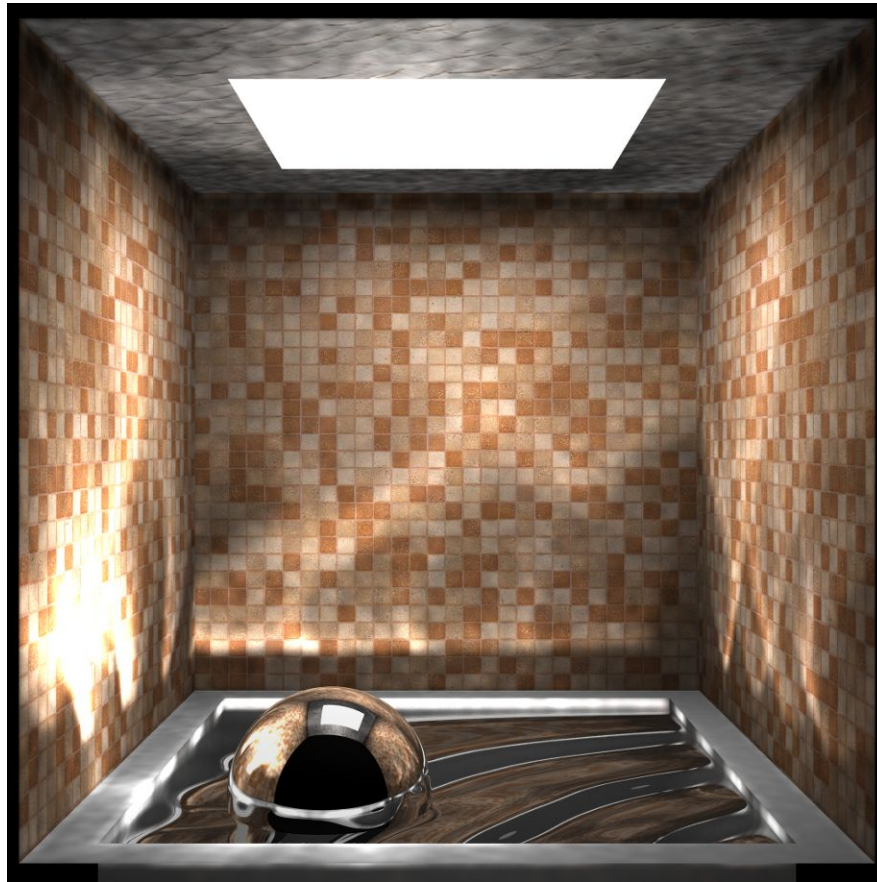
---

- 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:



- Choose a random value  $p \in [0, 1]$ . Where  $p$  falls in the probability map of the surface determines whether the photon is reflected, refracted or absorbed.

# Photon mapping gallery



[http://web.cs.wpi.edu/~emmanuel/courses/cs563/write\\_ups/zackw/phot\\_on\\_mapping/PhotonMapping.html](http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/phot_on_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)

## Radiosity

- nVidia: [http://http.developer.nvidia.com/GPUGems2/gpugems2\\_chapter39.html](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 Jenson, “Global Illumination using Photon Maps”: <http://graphics.ucsd.edu/~henrik/>
- Zack Waters, “Photon Mapping”: [http://web.cs.wpi.edu/~emmanuel/courses/cs563/write\\_ups/zackw/photon\\_mapping/PhotonMapping.html](http://web.cs.wpi.edu/~emmanuel/courses/cs563/write_ups/zackw/photon_mapping/PhotonMapping.html)