Advanced Graphics

Complex Scenes and Global Illumination

Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

Cover image: "*Cornell Box*" by Steven Parker, University of Utah.

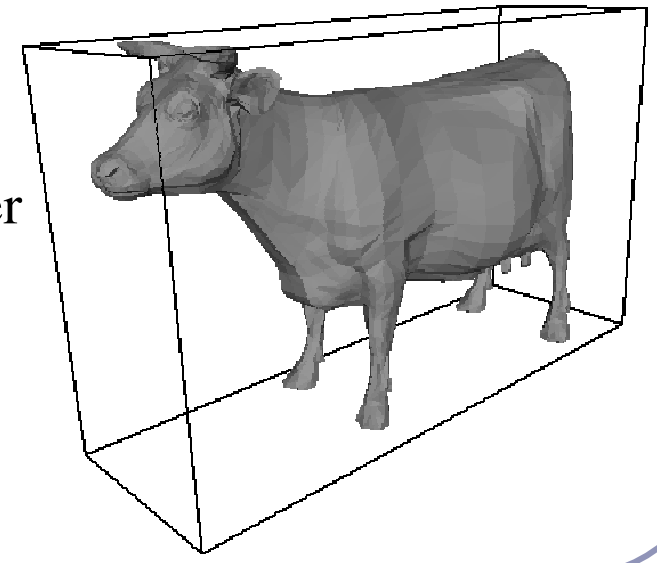
A tera-ray monte-carlo rendering of the Cornell Box, generated in 2 CPU years on an Origin 2000. The full image contains 2048 x 2048 pixels with over 100,000 primary rays per pixel (317 x 317 jittered samples). Over one trillion rays were traced in the generation of this image.

Speeding up ray-tracing with *bounding volumes*

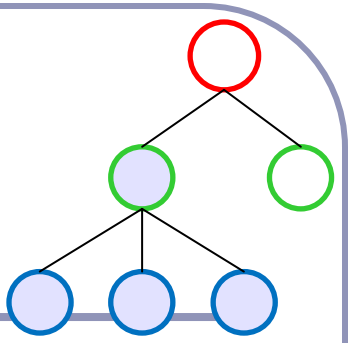
- You can speed up ray tracing for complex scenes by precomputing acceleration structures:
  - Organize the objects in your scene, *or*
  - Organize the space that holds the scene
- A common method of organizing objects is with a *hierarchy of bounding volumes*.

Types of bounding volumes

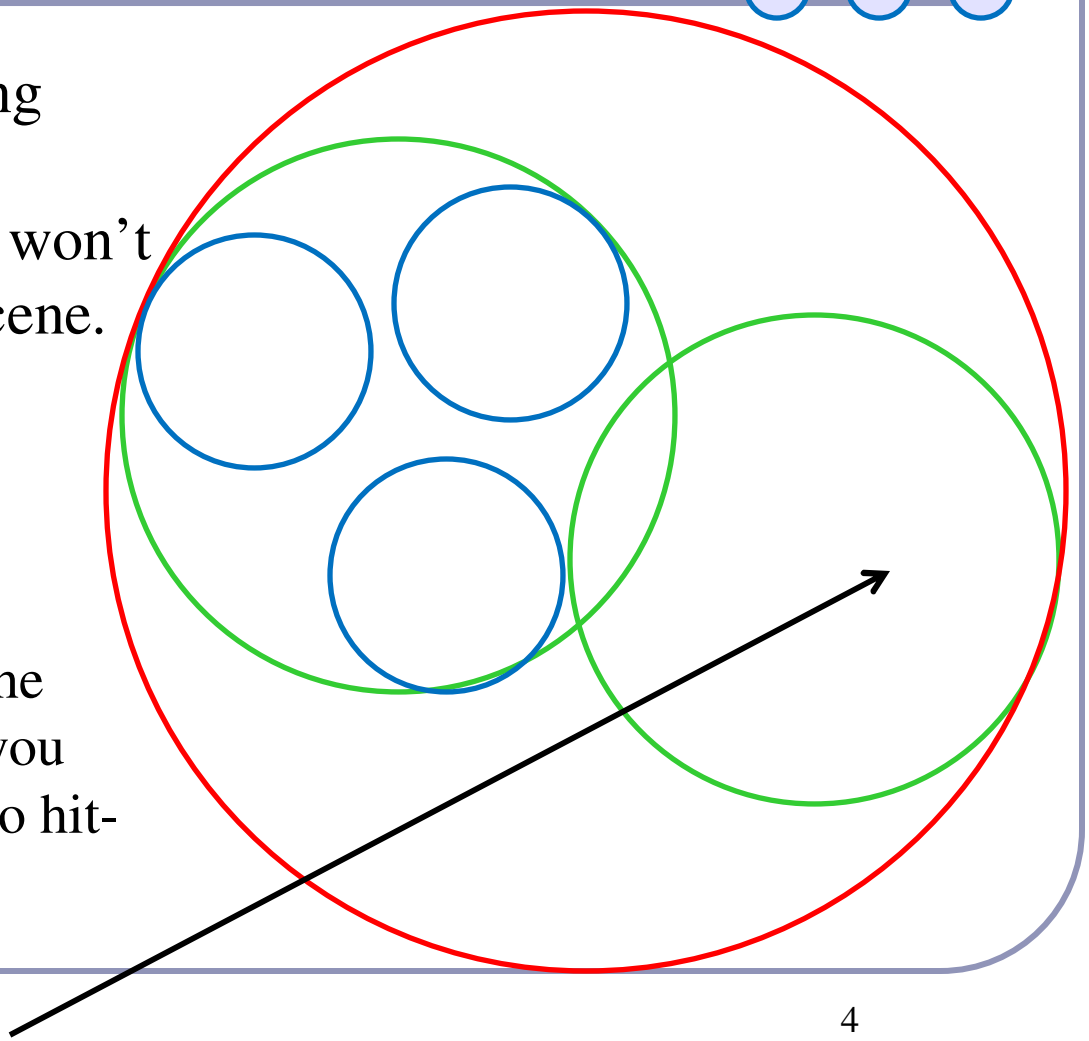
- Bounding volumes help to quickly accelerate volumetric tests, such as “does the ray hit the cow?”
  - choose fast hit testing over accuracy
  - ‘bboxes’ don’t have to be tight
- *Axis-aligned bounding boxes*
  - max and min of x/y/z.
- *Bounding spheres*
  - max of radius from some rough center
- *Bounding cylinders*
  - common in early FPS games



Bounding volumes in hierarchy

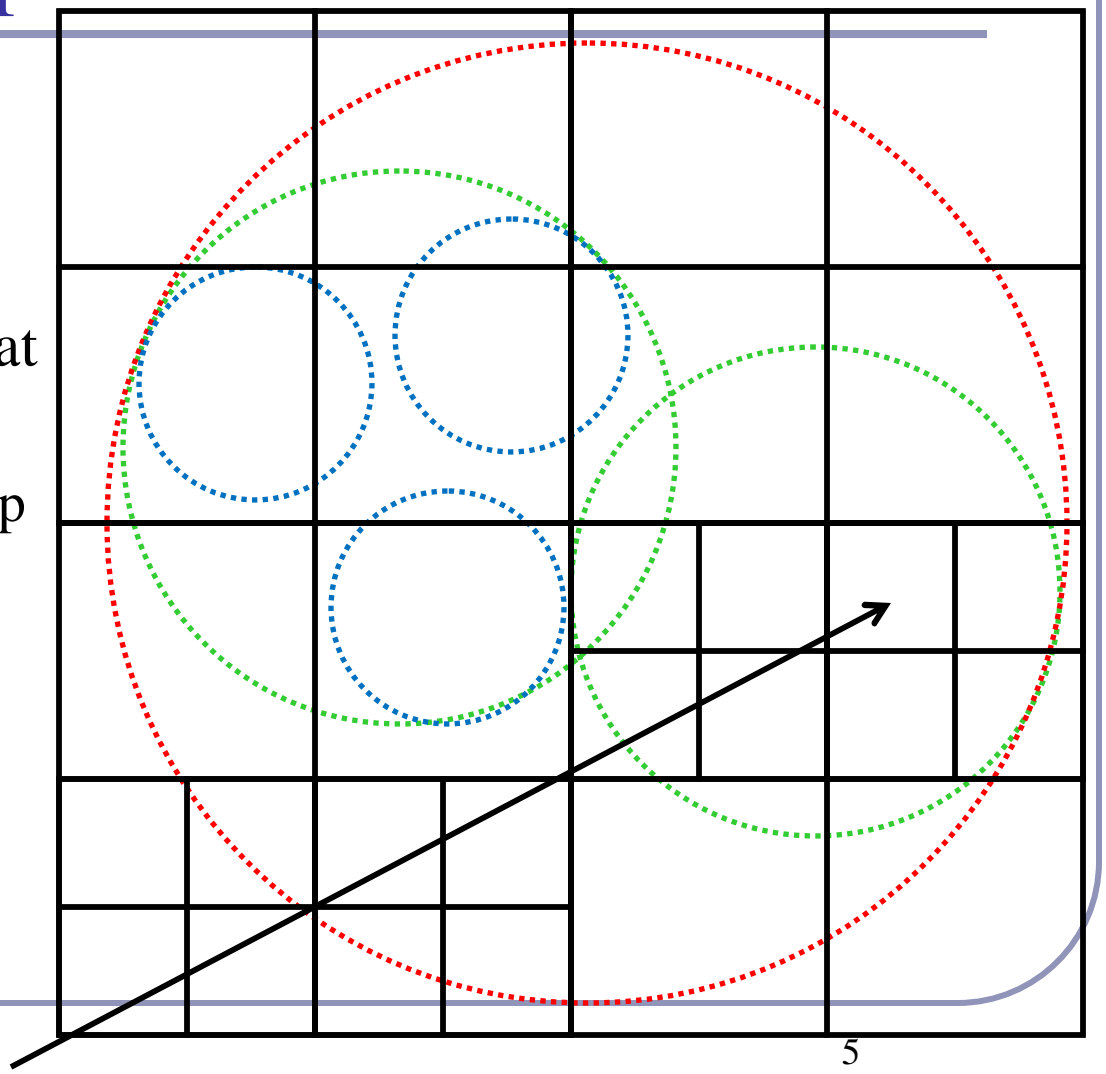


- Hierarchies of bounding volumes allow early discarding of rays that won't hit large parts of the scene.
  - Pro: Rays can skip subsections of the hierarchy
  - Con: Without spatial coherence ordering the objects in a volume you hit, you'll still have to hit-test every object



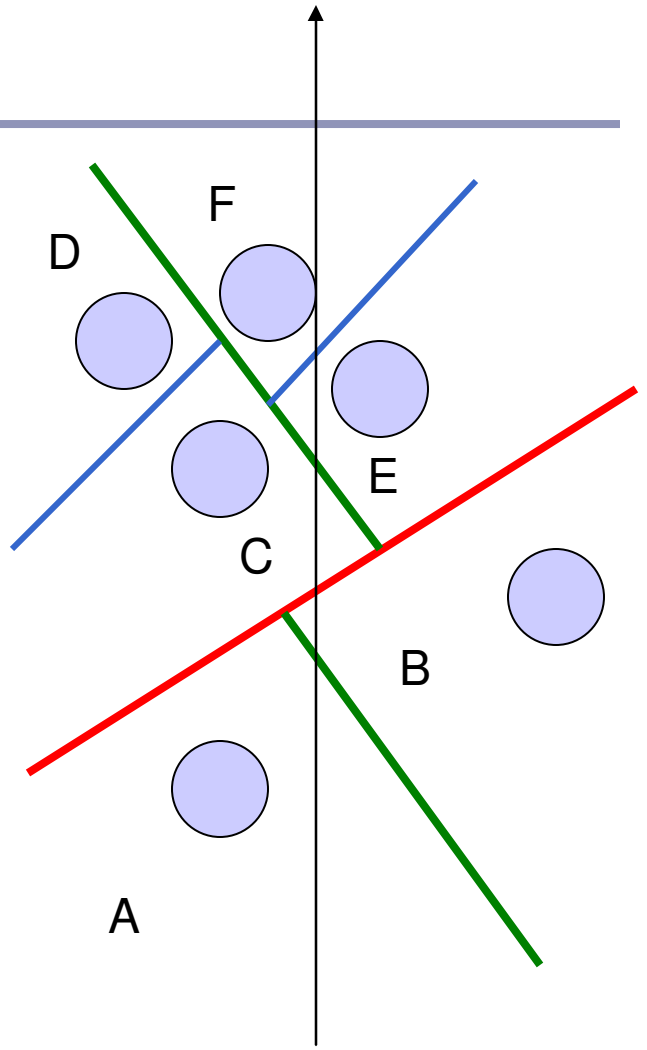
Subdivisions in space

- Split space into cells and list in each cell every object in the scene that overlaps that cell.
  - Pro: The ray can skip empty cells
  - Con: Most objects will overlap many cells



Popular acceleration structures: *BSP Trees*

- The *BSP tree* partitions the scene into objects in front of, on, and behind a tree of planes.
  - When you fire a ray into the scene, you test all near-side objects before testing far-side objects.
- Problems:
  - choice of planes is not obvious
  - computation is slow
  - plane intersection tests are heavy on floating-point math.



Popular acceleration structures: *kD-Trees*

- The *kd-tree* is a simplification of the BSP Tree data structure
  - Space is recursively subdivided by axis-aligned planes and points on either side of each plane are separated in the tree.
  - The *k*d-tree has O(n log n) insertion time (but this is very optimizable by domain knowledge) and O($n^{2/3}$) search time.
  - *k*d-trees don't suffer from the mathematical slowdowns of BSPs because their planes are always axis-aligned.

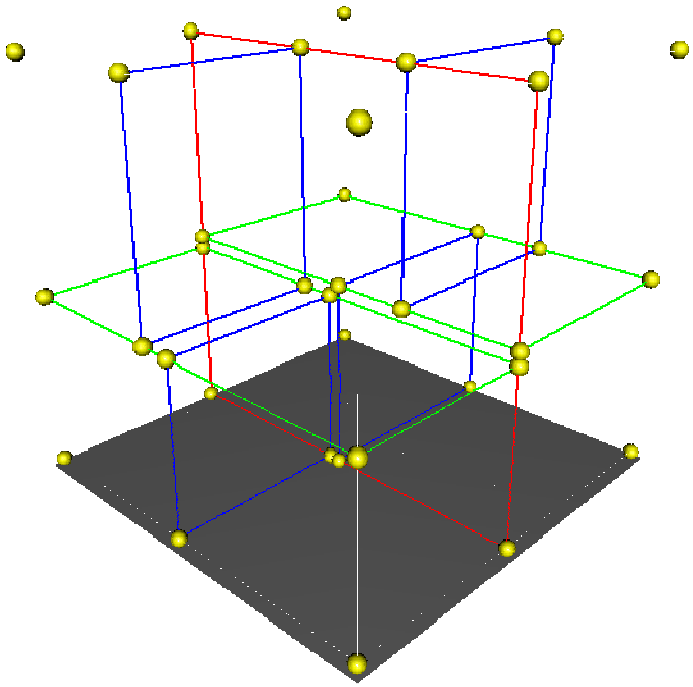


Image from Wikipedia, bless their hearts.

Popular acceleration structures: *Bounding Interval Hierarchies*

- The *Bounding Interval Hierarchy* subdivides space around the volumes of objects and shrinks each volume to remove unused space.
  - Think of this as a "best-fit" *k*d-tree
  - Can be built dynamically as each ray is fired into the scene

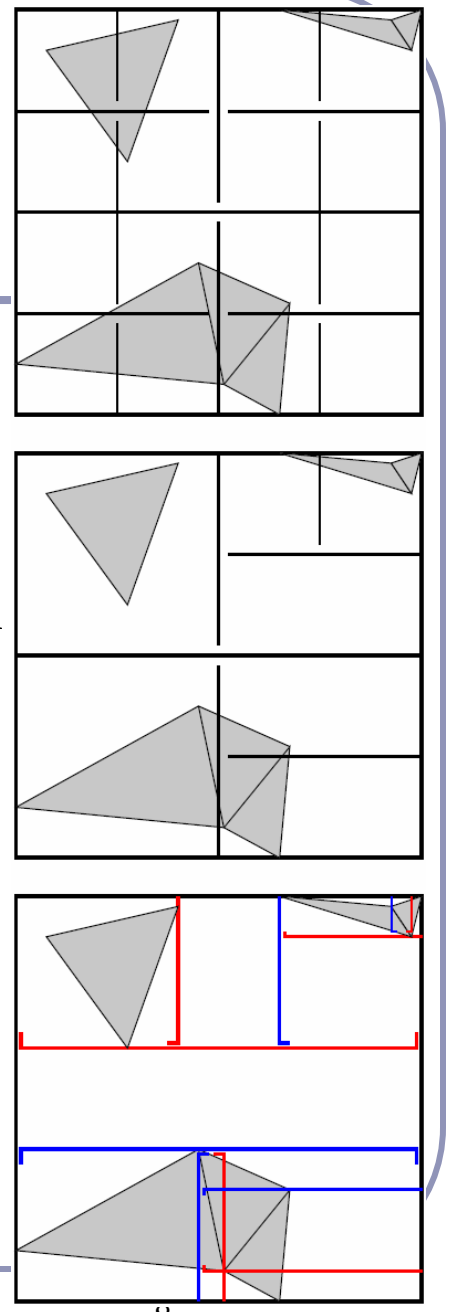
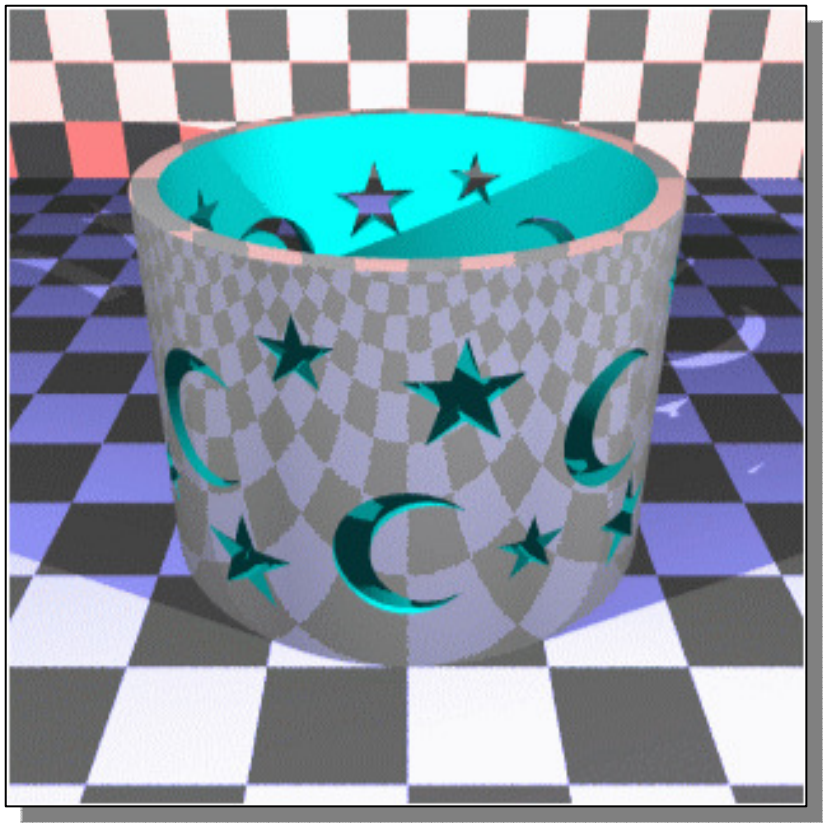


Image from Wächter and Keller's paper, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics (2006)

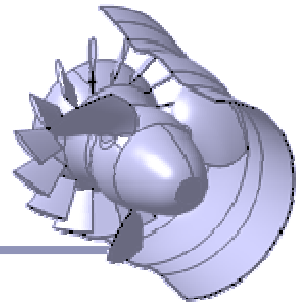
Applications of ray tracing: *Constructive Solid Geometry*

- *Constructive Solid Geometry* (CSG) builds up 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

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?)

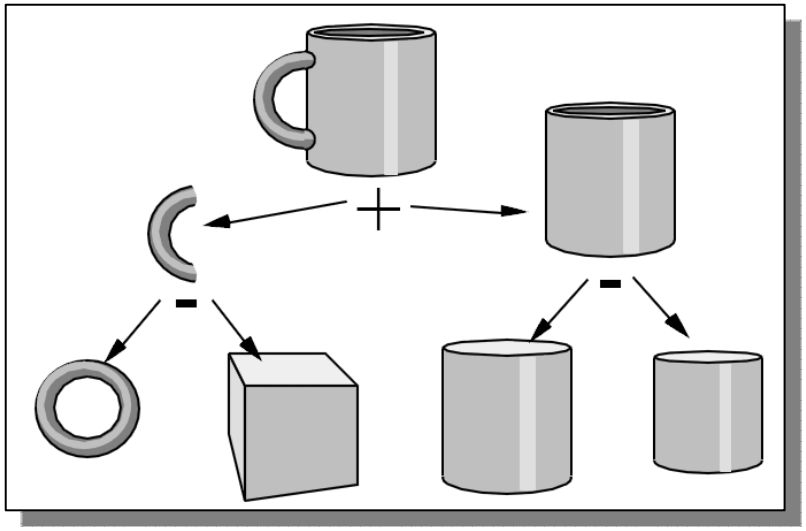
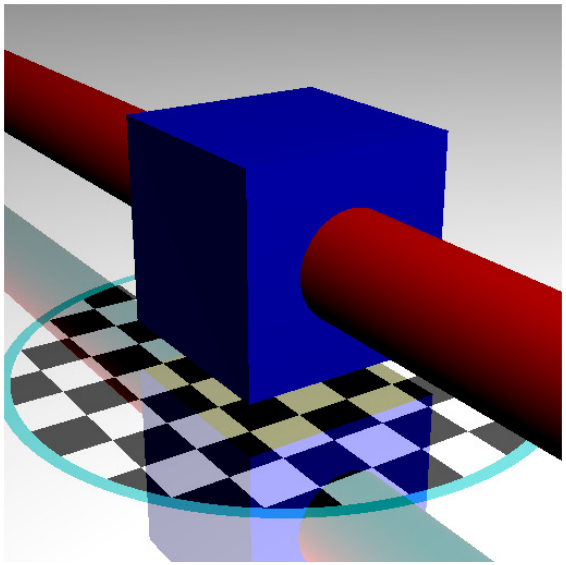


Figure from Wyvill (1995) part two, p. 4

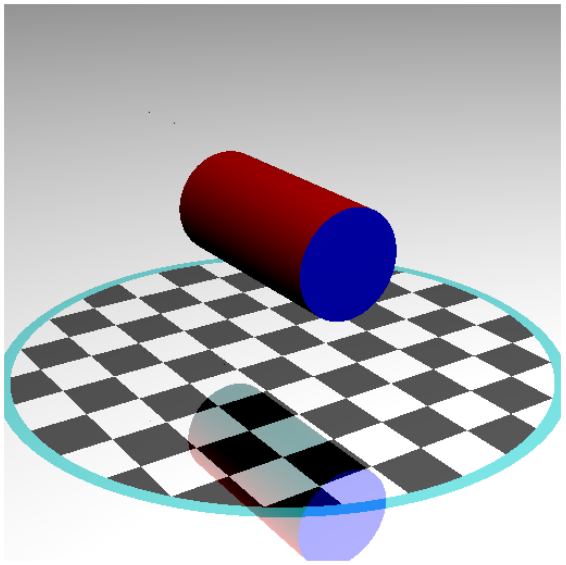
Constructive Solid Geometry

Three operations:

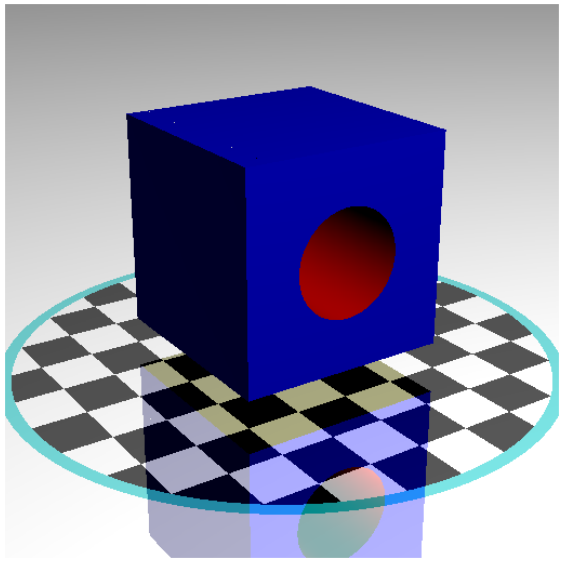
1. *Union*



2. *Intersection*

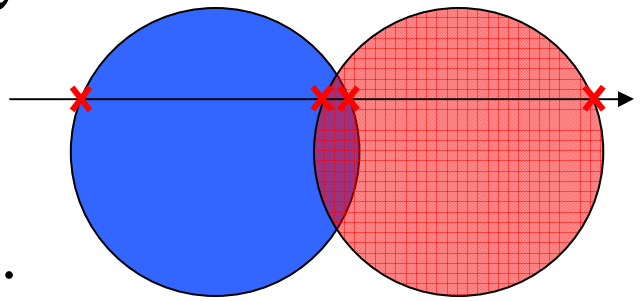


3. *Difference*



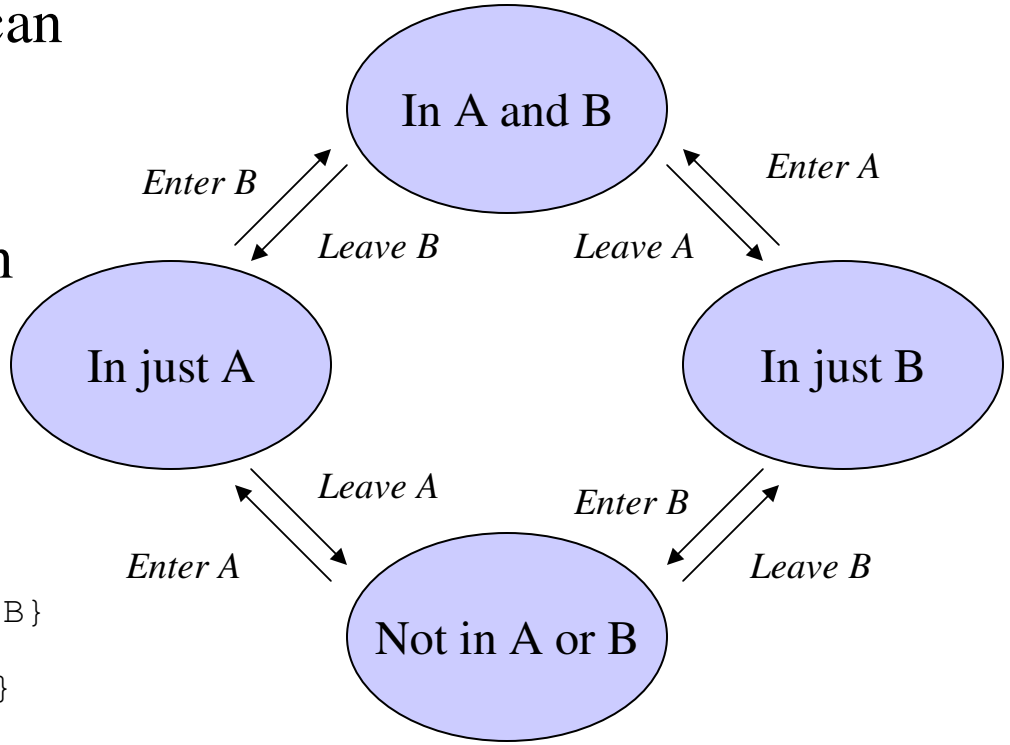
Ray tracing CSG models

- For each node of the binary tree:
  - Fire your 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

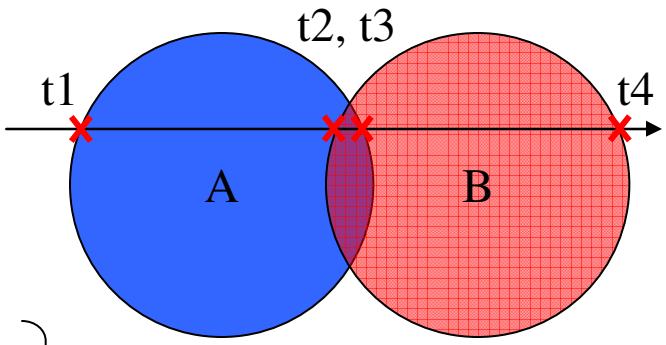
- Each boolean operation can be modeled as a state machine.
- For each operation, retain those intersections that transition into or out of the critical state(s).
  - Union: `{In A | In B | In A and B}`
  - Intersection: `{In A and B}`
  - Difference: `{In A}`



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

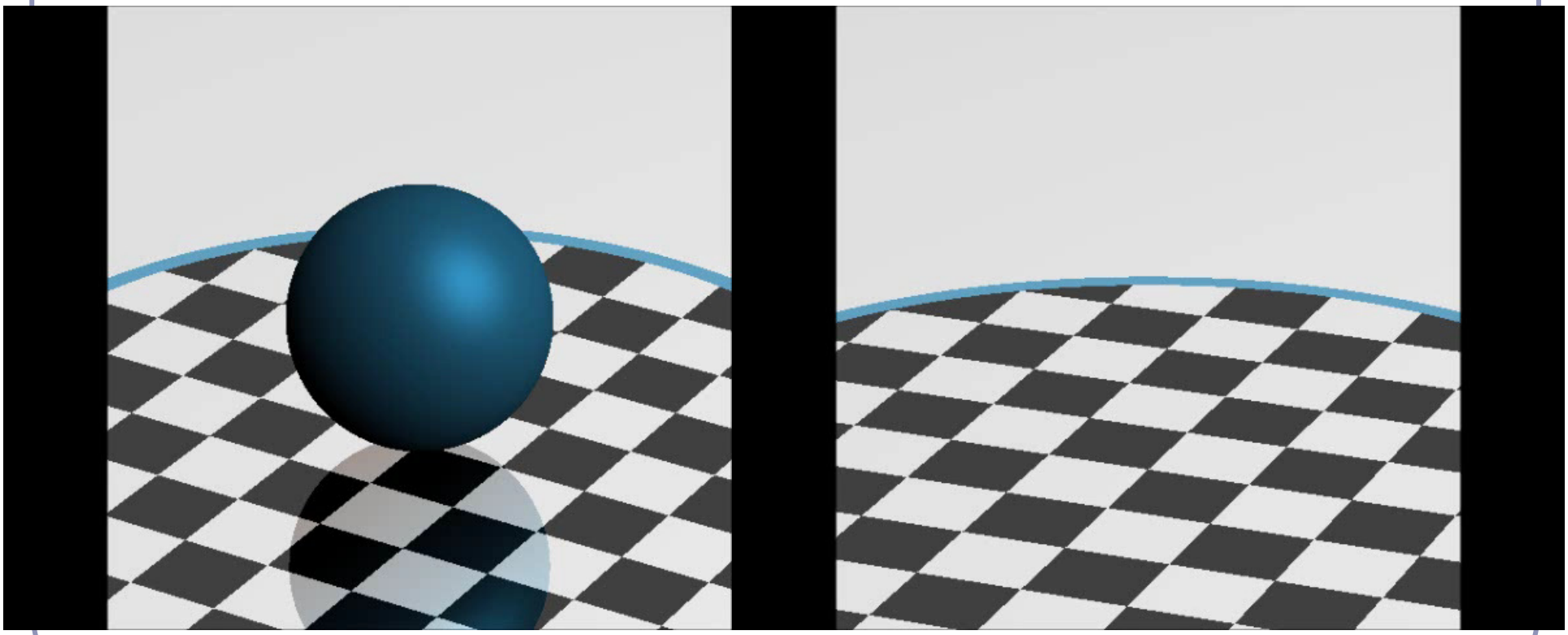



```
difference =
((wasInA != isInA) &&
 (!isInB)&&(!wasInB))
||
((wasInB != isInB) &&
 (wasInA || isInA))
```

CSG in action

Difference

Intersection



What's wrong with raytracing?

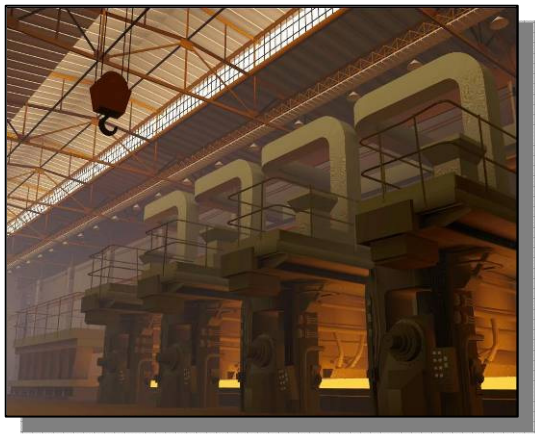
- Soft shadows are expensive
- Shadows of transparent objects require further coding or hacks
- Lighting off reflective objects follows different shadow rules from normal lighting
- Hard to implement diffuse reflection (color bleeding, such as in the Cornell Box—notice how the sides of the inner cubes 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.

Global Illumination: *Radiosity*

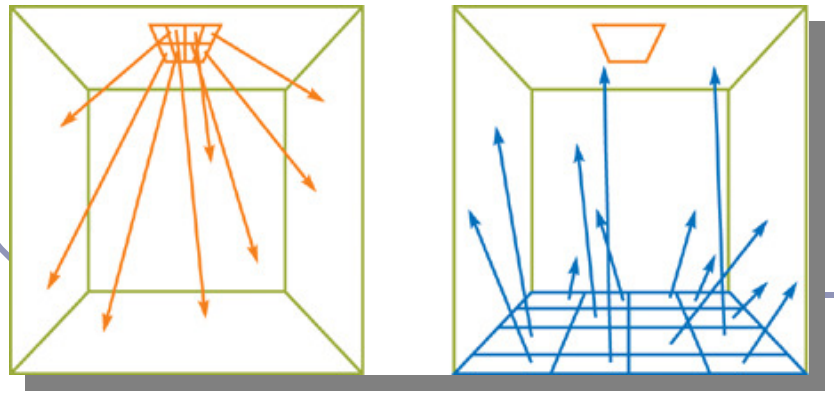
- *Radiosity* is an illumination method which simulates the global dispersion and reflection of diffuse light.
  - First developed for describing spectral heat transfer in the 1950s
  - Adapted to graphics in the 1980s at Cornell
- Radiosity is a finite-element approach to global illumination: it breaks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



Images from Cornell University's graphics group
http://www.graphics.cornell.edu/online/research/

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.



Note: very unfortunately, some literature uses the term ‘form factor’ for the view factor as well.

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:

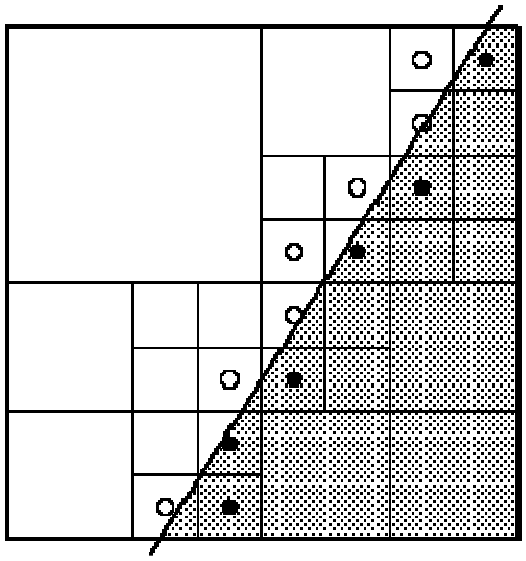
$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij}$$

where…

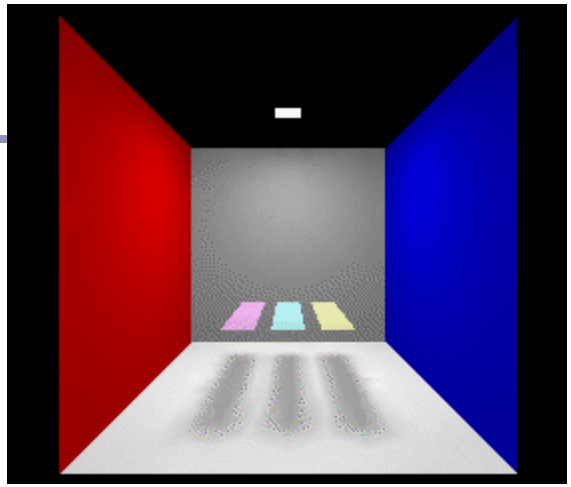
- B_i is the radiosity of patch i;
  B_j is the cumulative radiosity of all other patches ($j \neq i$)
- E_i is the emitted energy of the patch
- R_i is the reflectivity of the patch
- F_{ij} is the view factor of energy from patch i to patch j.

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 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.



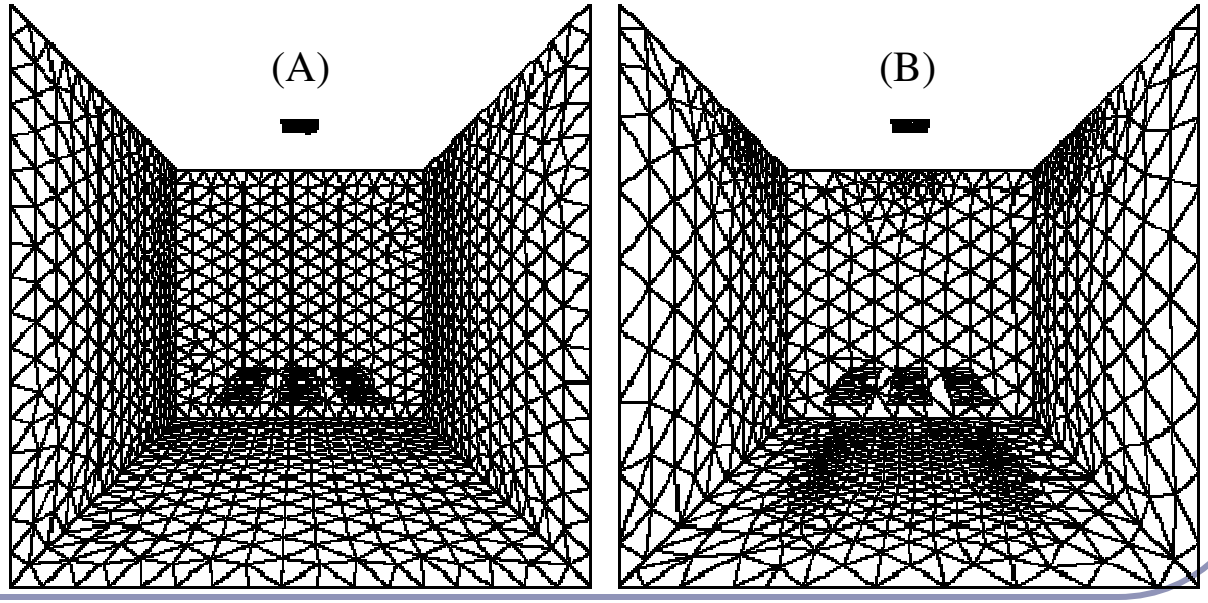
Radiosity—implementation



(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

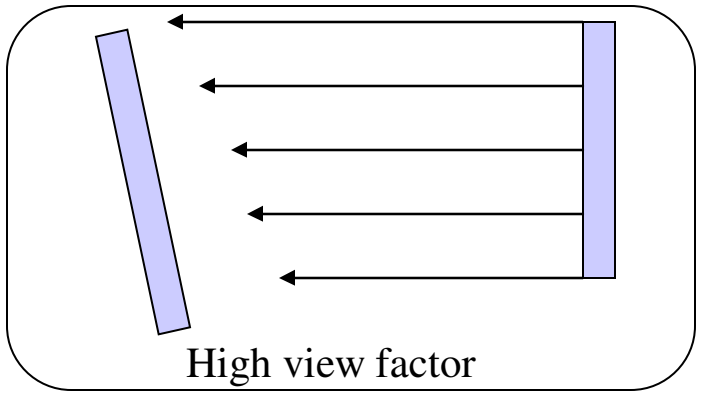


Radiosity—view factors

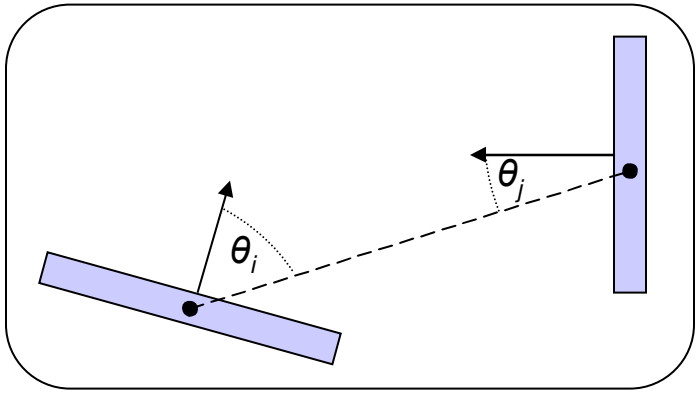
- One equation for the view factor between patches *i*, *j* is:

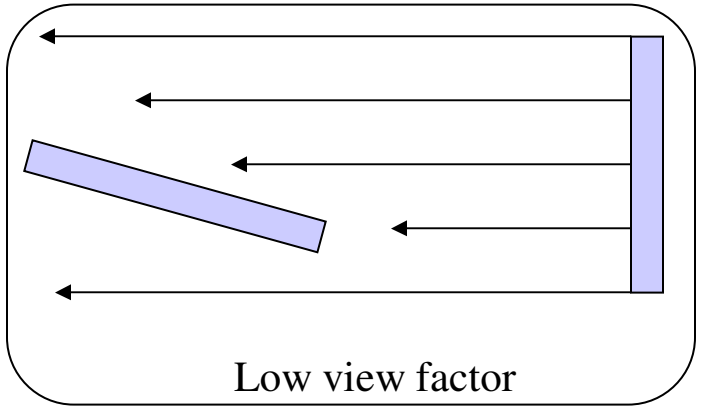
$$Fi \rightarrow j = \frac{\cos\theta_i \cos\theta_j}{\pi r^2} V(i, j)$$

…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.)



High view factor

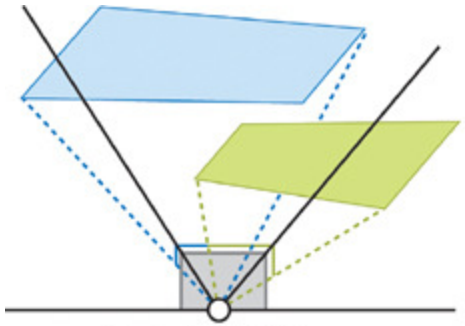




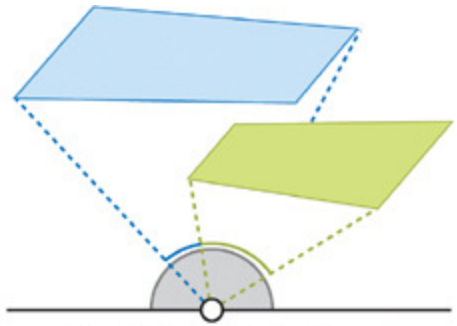
Low view factor

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

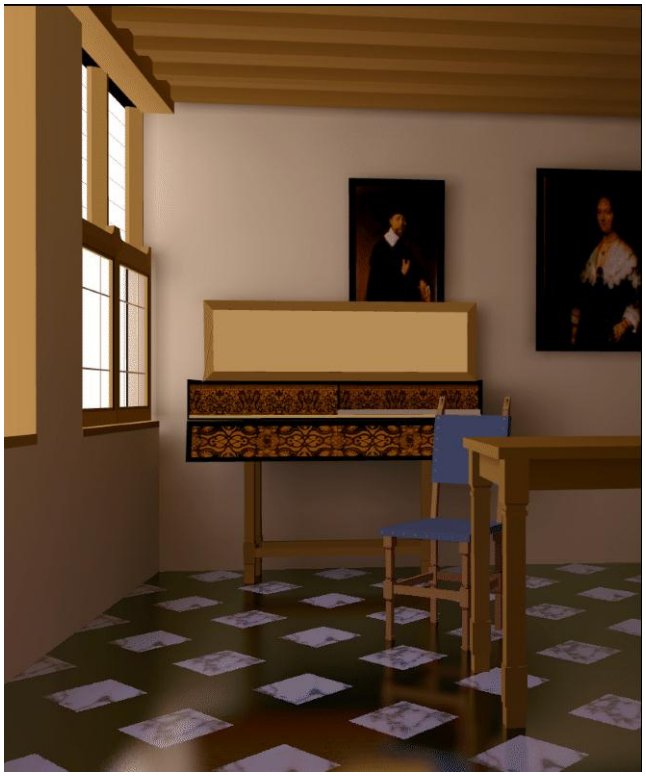


Image from *A Two Pass Solution to the Rendering Equation: a Synthesis of Ray Tracing and Radiosity Methods*,
John R. Wallace, Michael F. Cohen and Donald P. Greenberg
(Cornell University, 1987)

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)→

Image from http://graphics.ucsd.edu/~henrik/
Generated with photon mapping

Global Illumination: *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)

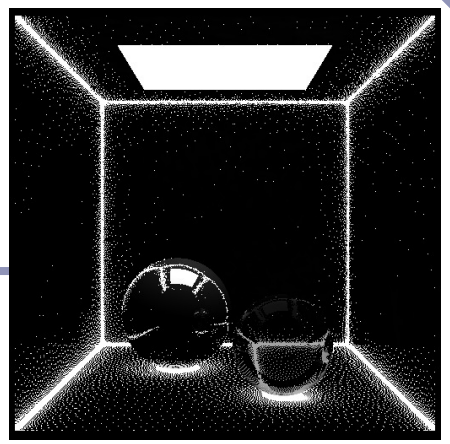


Image by Zack Waters

Photon mapping is a two-pass algorithm:

1. Photon scattering
    1. 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.
    2. Photons fire through the scene (re-use that raytracer, folks.) Where they strike a surface they are either absorbed, reflected or refracted.
    3. 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 is common.

Photon mapping—algorithm (2/2)

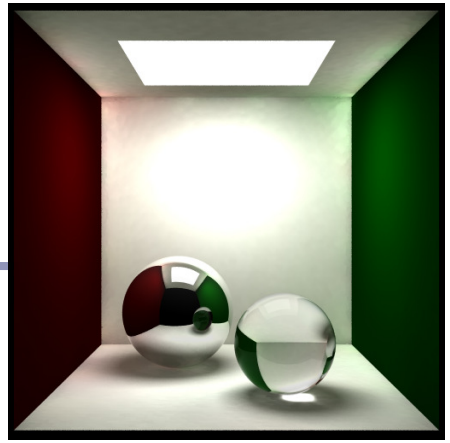


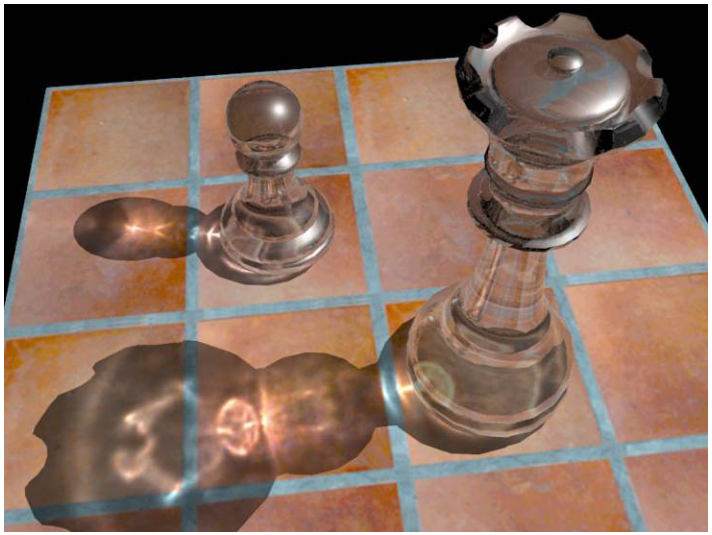
Image by Zack Waters

Photon mapping is a two-pass algorithm:

2. Rendering
    1. Ray trace the scene from the point of view of the camera.
    2. 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.
    3. Compute radiant illumination by summing the contribution along the eye ray of all photons within a sphere of radius *r* of *P*.
    4. Caustics can be calculated directly here from the photon map. For speed, the caustic map is usually distinct from the radiance map.

Photon mapping

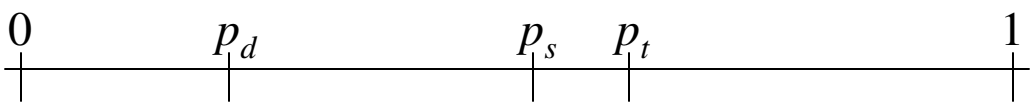
- This method is a great example of *Monte Carlo integration*, in which a difficult integral (the lighting equation) is simulated by randomly sampling values from within the integral's domain until enough samples average out to about the right answer.
  - This means that you're going to be firing *millions* of photons. Your data structure is going to have to be very space-efficient!



http://www.okino.com/conv/imp_jt.htm

Photon mapping

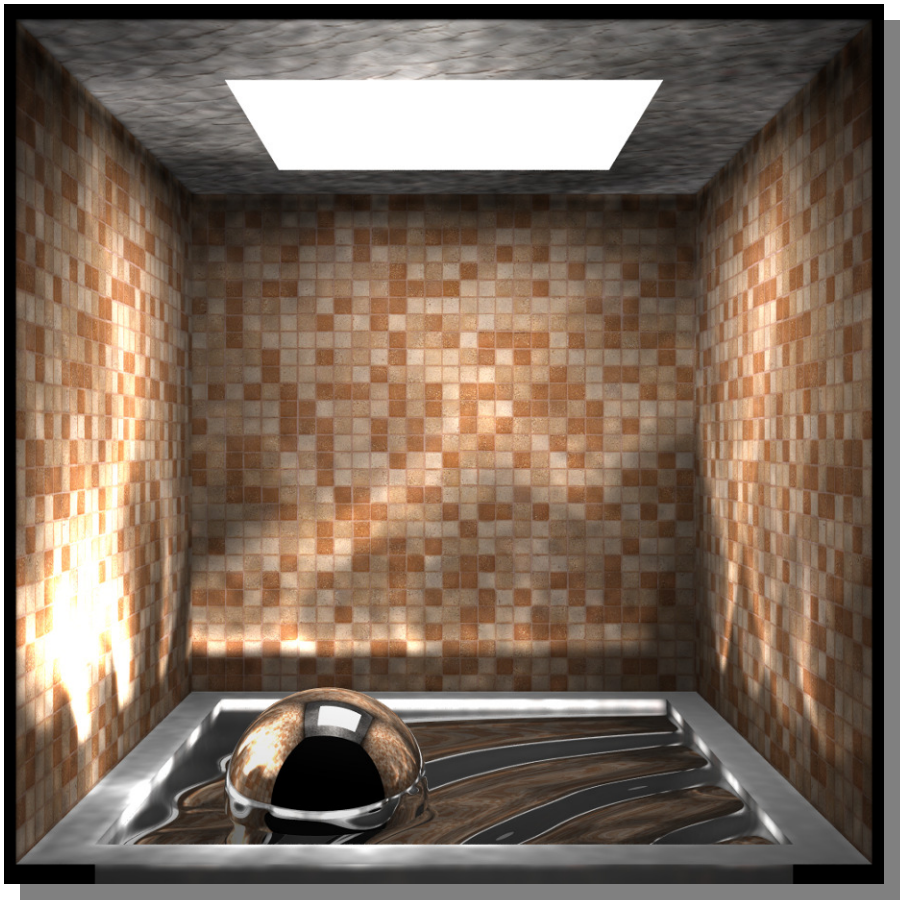
- Initial photon direction is random. Constrained by light shape, but random.
- What exactly happens each time a photon hits a solid also has a random component:
  - Based on the diffuse reflectance, specular reflectance and transparency of the surface, compute probabilities p_d, p_s and p_t where $(p_d+p_s+p_t)\leq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



This surface would have minimal specular highlight.

  - 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/photon_mapping/PhotonMapping.html

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

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

References

Bounding Interval Hierarchy

- Wächter and Keller, *Instant Ray Tracing: The Bounding Interval Hierarchy*, Eurographics, 2006

CSG

- G. Wyvill, *Practical Ray Tracing (parts 1 and 2)*, University of Otago, 1995

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 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