Raytracing

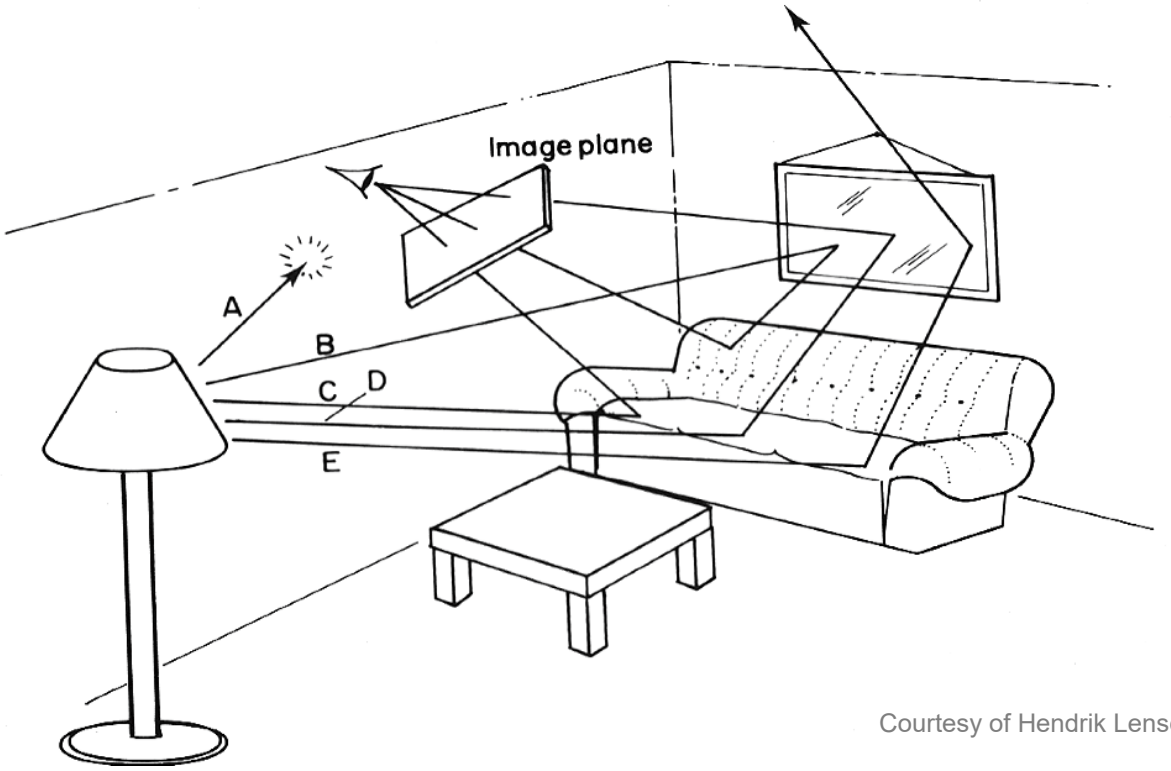
Chapter 20

May 17, 2016

Ray tracing

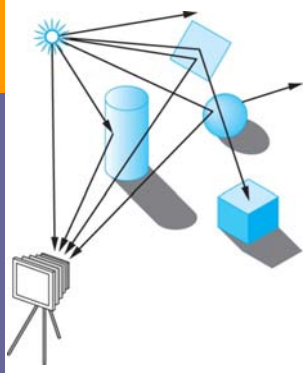
- Different approach to organizing the rendering process
- It is pixel and ray based, instead of triangle based.
- Historically, it has not been hardware accelerated, and is used for offline rendering.
- It is very flexible and so various optical effects can be put in very easily.
- It is at the heart of photorealistic methods of light simulation.
- We will cover it only briefly.

Ray tracing pipeline



Courtesy of Hendrik Lensch

Ray Tracing



Loop ordering

- *Rasterization-based* rendering

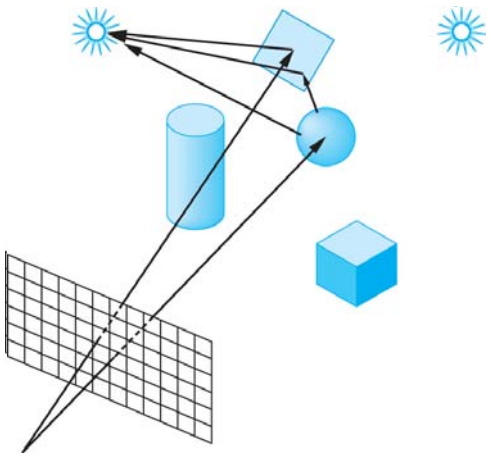
```
initialize z-buffer
for all triangles
  for all pixels covered by the triangle
    compute color and z
    if z is closer than what is already in the z-buffer
      update the color and z of the pixel
```

- Basic ray tracing

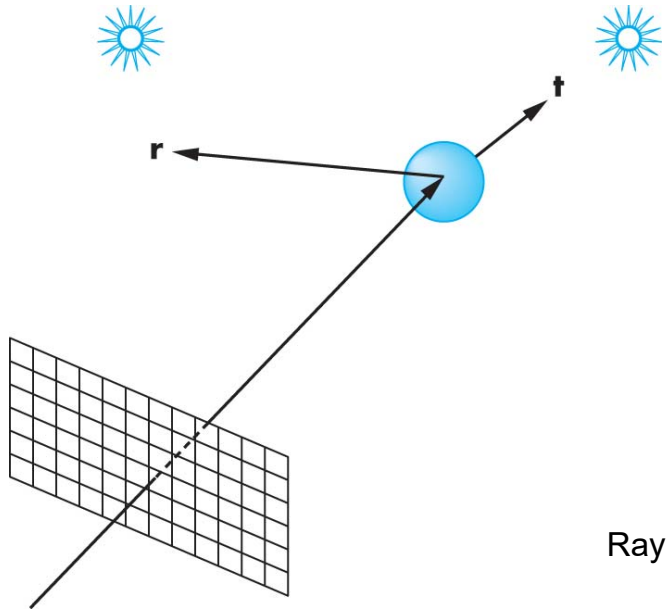
```
for all pixels on the screen
  for all objects seen in this pixel
    if this is the closest object seen at the pixel
      compute color and z
      set the color of the pixel
```

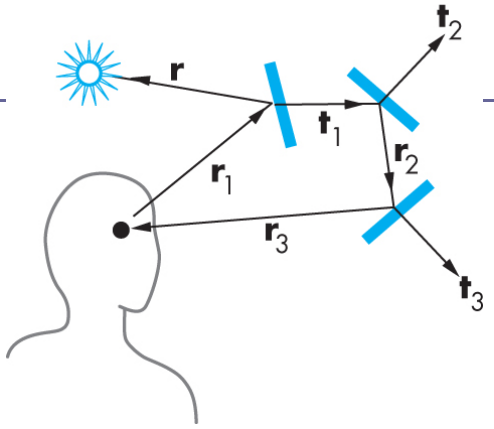

Shadow Ray

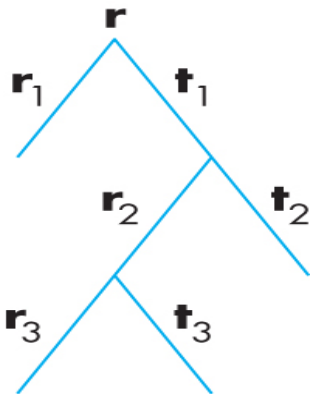
- In ray tracing, rather than immediately applying our reflection model, we first check whether the point of intersection between the cast ray and the surface is illuminated.
- We compute **shadow rays** from the point on the surface to each source.
  - If a shadow ray intersects a surface before it meets the source, the light is blocked from reaching the point under consideration, and this point is in shadow, at least from this source.
  - If the surfaces are highly reflective, we follow the shadow ray as it bounces from surface to surface until it either goes off to infinity or intersects a source.



Recursive process







Ray tree

A simple ray tracer

- color c = **trace**(point p, vector d, int step)
- color local, reflected, transmitted;
- point q; normal n;
- if (step>max) return (background_color):
- q = intersect (p, d, status);
  if (status == light_source) return(light_source_color);
  if (status == no_intersection) return(backgound_color);
- n = nomal(q);
  r = reflect (q, n);
  t = transmit(q, n);
- local = phong(q,n,r);
  reflected = **trace**(q,r,step+1);
  transmitted = **trace**(q, r, step+1);
- return(local + reflected + transmitted);

Benefits of rasterization

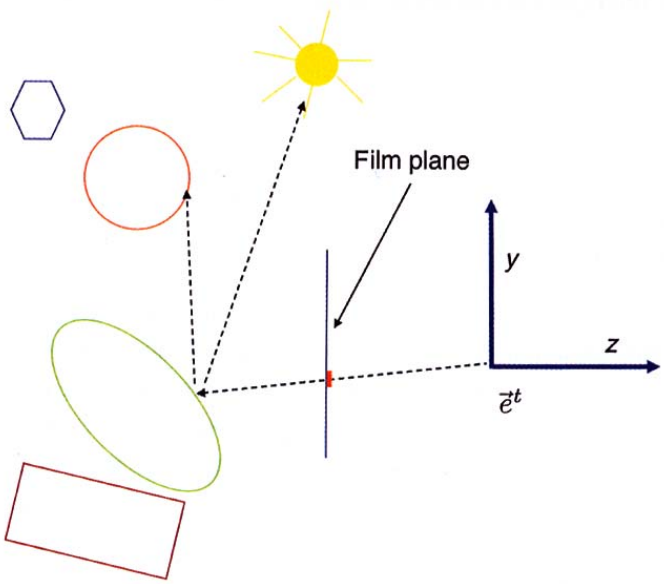
- This algorithm has the nice property that each triangle in the scene is touched only once, and in a predictable order.
  - Good for memory usage
- Setup computation is done only once and can be enhanced with fancy shading computation and even with multipass algorithms.
  - Occlusion culling algorithm can be applied beforehand.

Benefits of ray tracing

- Using ray tracing, we never waste computation calculating the color of an occluded object.
- Because we have an ordered list of intersections along a ray (nonrefractive) transparency is easy to model.
- Using ray intersections, we can directly render smooth objects without first having to dice them up into triangles.
- It is easy to render solid objects described by volumetric set operations.
- It is easy to model perfect mirror reflection, and to compute shadows.

Intersection

- The main computation needed in ray tracing is computing the intersection of a geometric ray $(\tilde{p}, \vec{d})$ with an object in the scene.
  - Here $\tilde{p}$ is the start of the ray, which goes off in the direction $\vec{d}$



Scene geometry

Ray-plain

- Plane described by the equation $Ax + By + Cz + D = 0$
- We start by representing every point along the ray using a single parameter λ

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} + \lambda \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

- Plugging this into the plane equation, we get

$$\begin{aligned} 0 &= A(p_x + \lambda d_x) + B(p_y + \lambda d_y) + C(p_z + \lambda d_z) + D \\ &= \lambda(Ad_x + Bd_y + Cd_z) + Ap_x + Bp_y + Cp_z + D \end{aligned}$$

- So,

$$\lambda = \frac{-Ap_x - Bp_y - Cp_z - D}{Ad_x + Bd_y + Cd_z}$$

  - λ tells us where along the ray the intersection point is

Ray-triangle

- Negative valued λ are backwards along the ray.
- Comparisons between λ values can be used to determine which, among a set of planes, is the first one intersected along a ray.
- *ABCD* per triangle, d_{xyz} per pixel, p_{xyz} per camera

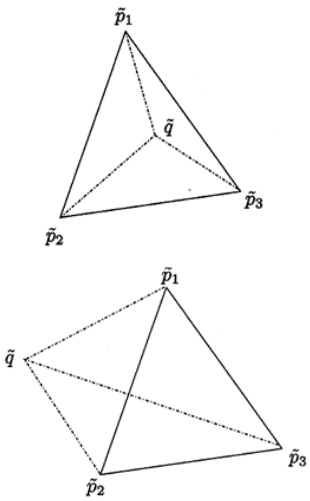
- In the first step, we compute the *A, B, C, D* of the plane supporting the triangle, and compute the ray-plane intersection as above.
- Next, we need a test to determine if the intersection point is inside or outside of the triangle.
- We can build such a test using the "counter clockwise" calculation seen earlier in culling

Ray-triangle

- Suppose we wish to test if a point $\tilde{q}$ is inside or outside of a triangle $\Delta(\tilde{p}_1\tilde{p}_2\tilde{p}_3)$ in 2D
- Consider the three "sub" triangle

$$\Delta(\tilde{p}_1\tilde{p}_2\tilde{q}), \Delta(\tilde{p}_1\tilde{q}\tilde{p}_3) \text{ and } \Delta(\tilde{q}\tilde{p}_2\tilde{p}_3)$$

- When $\tilde{q}$ is inside of $\Delta(\tilde{p}_1\tilde{p}_2\tilde{p}_3)$, then all three sub-triangles will agree on their clockwisedness. When $\tilde{q}$ is outside, then they will disagree.



Ray-sphere

- Suppose we have a sphere with radius R and center $\mathbf{c}$ modeled as the set of points $[x, y, z]^t$ that satisfy the equation $(x - c_x)^2 + (y - c_u)^2 + (z - c_z)^2 - r^2 = 0.$
  Plugging it into the plane equation, we get

$$\begin{aligned} 0 &= (p_x + \lambda d_x - c_x)^2 + (p_y + \lambda d_y - c_y)^2 + (p_z + \lambda d_z - c_z)^2 - r^2 \\ &= (d_x^2 + d_y^2 + d_z^2)\lambda^2 + (2d_x(p_x - c_x) + 2d_y(p_y - c_y) + 2d_z(p_z - c_z))\lambda \\ &\quad +(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 - r^2 \end{aligned}$$

- Find its real roots using quadratic formula.
- If there are 2 real roots, these represent 2 intersections.
  - 1 root: intersection is tangential
  - No root: ray misses the sphere.
- Normal of the sphere at $[x, y, z]^t$ is in the direction $[x - c_x, y - c_y, z - c_z]^t$

Early Rejection

- When computing the intersection between a ray and the scene, instead of testing every scene object for intersection with the ray, we may use auxiliary data structures to quickly determine that some set of objects is entirely missed by the ray.
- For example, one can use a simple shape (say a large sphere or box) that encloses some set of objects.
  - Given a ray, one first calculates if the ray intersects this volume.
  - If it does not, then clearly this ray misses all of the objects in the bounded set, and no more ray intersection tests are needed.
- This idea can be further developed with hierarchies and spatial data structure.

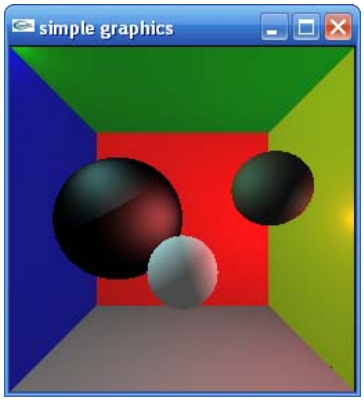
Secondary Rays

- To determine if a scene point is in shadow, one follows a “shadow ray” from the observed point towards the light to see if there is any occluding geometry.
- Another easy calculation that can be done is mirror reflection (and similarly refraction). In this case, one calculates the bounce direction “bounce ray” off in that direction: $B(\vec{w}) = 2(\vec{w} \cdot \vec{n})\vec{n} - \vec{w}$

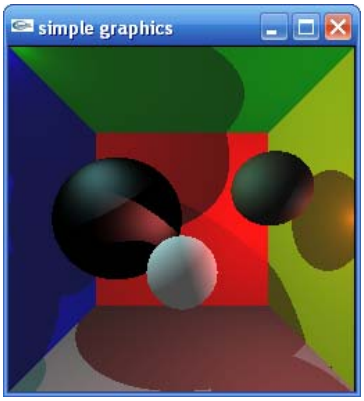
Secondary Rays

- The color of the point hit by this ray is then calculated and used to determine the color of the original point on the mirror.
- This idea can be applied recursively some number of times to simulate multiple mirror.

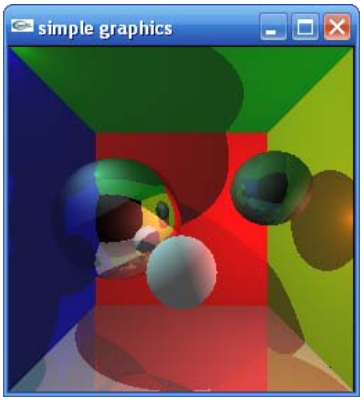
Ray-casting only



Ray-casting + shadow



Rec. ray-tracing + shadow



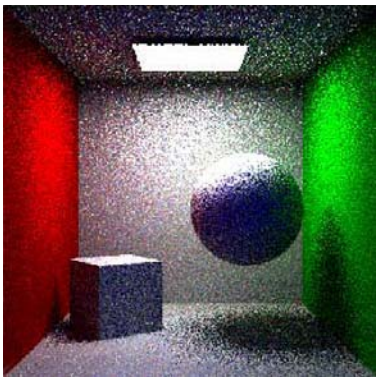
Even more rays

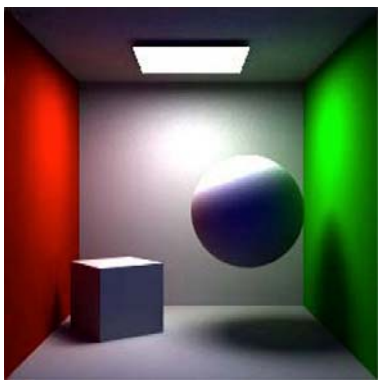
- More realistic optical simulation requires the computation of integrals, and these integrals can often be approximated by summing up contributions along a set of samples.
- Computing the values for these samples often involves tracing rays through the scene.
- For example, we may want to simulate a scene that is illuminated by a large light with finite area. This, among other things, results in soft shadow boundaries.

Even more rays

- Lighting due to such *area light sources* can be approximated by sending off a number of shadow rays towards the area light and determining how many of those rays hit the light source.
- Other similar effects such as focus effects of camera lenses and inter-reflection are discussed can be calculated by tracing many rays through the scene.

4 rays per pixel



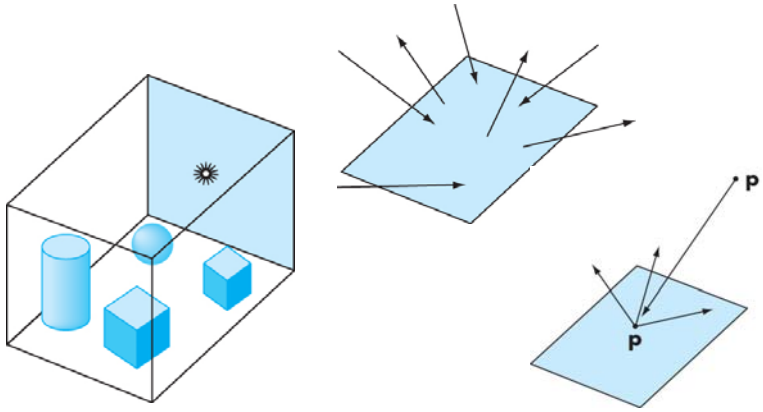


1024 rays per pixel

Global Illumination (GI)

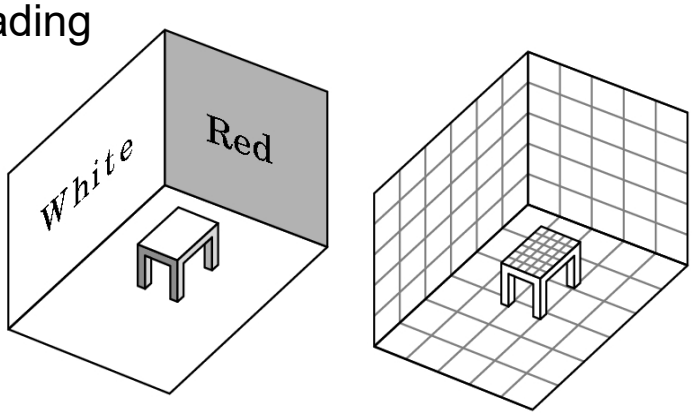
- Creates images that are more physically accurate than any other rendering technique
  - It calculates indirect illumination on objects, including diffuse, glossy and specular inter-reflection between surfaces and transmission of light from other objects.
- Global illumination rendering methods are based on the principles of illumination theory and energy transfer.

- Examples
  - Radiosity
  - Photon mapping



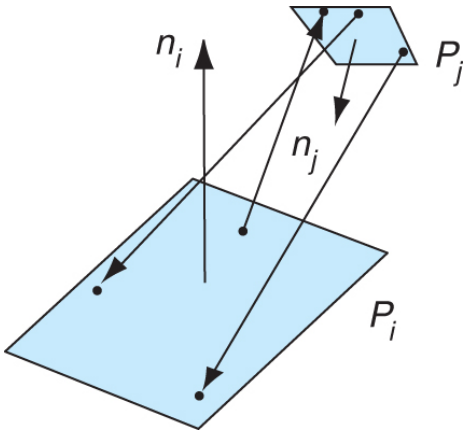
Radiosity

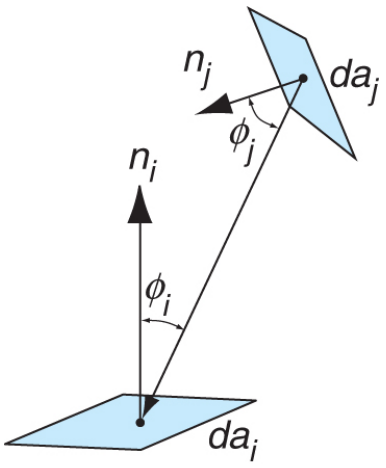
- Basic idea
  - Break up the scene into small flat polygons (patches) that are assumed to be perfectly diffuse
  - Consider patches pair-wise to determine **form factors** (describing how the light energy leaving one patch affects the other)
  - Compute the shading



Radiosity

- Diffuse-diffuse interaction
- Radiosity equation





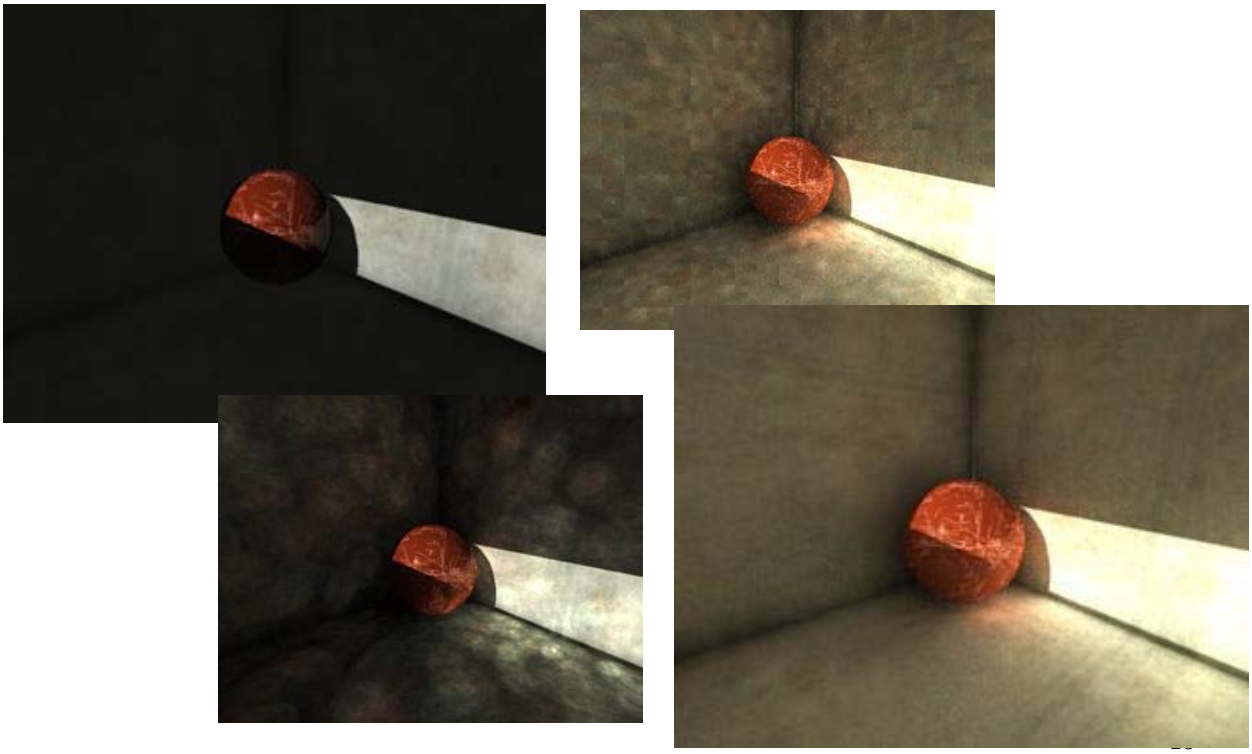
Radiosity

Photon mapping

- Photon maps are collections of small energy packets that are emitted into the scene to represent the way light travels through space.
- The values of photons as they are reflected, absorbed or transmitted through surfaces and values are used to compute GI.
- Caustics are a striking light effect rendered with this technique.
  - Caustics are created when specular light is focused or dispersed by reflection or refraction, and are typically seen when light travels through crystal or water.

Photon mapping

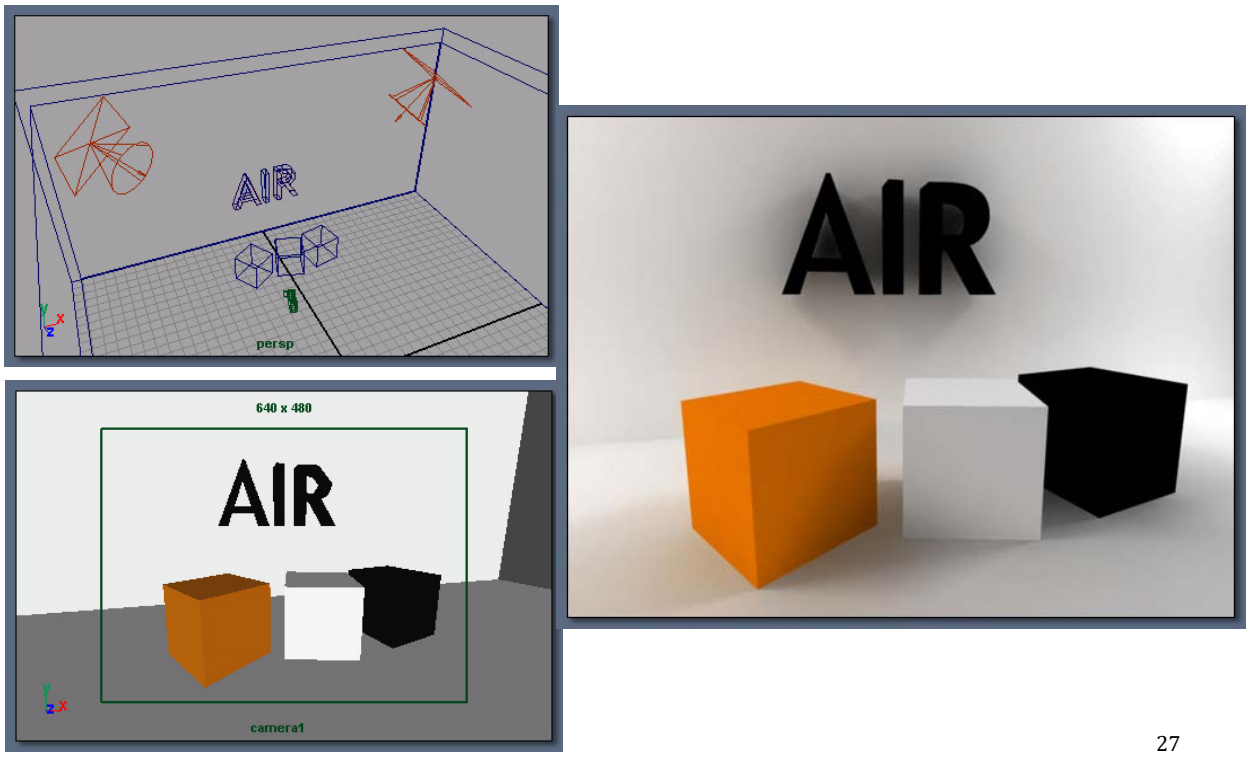
SoftImage (XSI)



Photon mapping

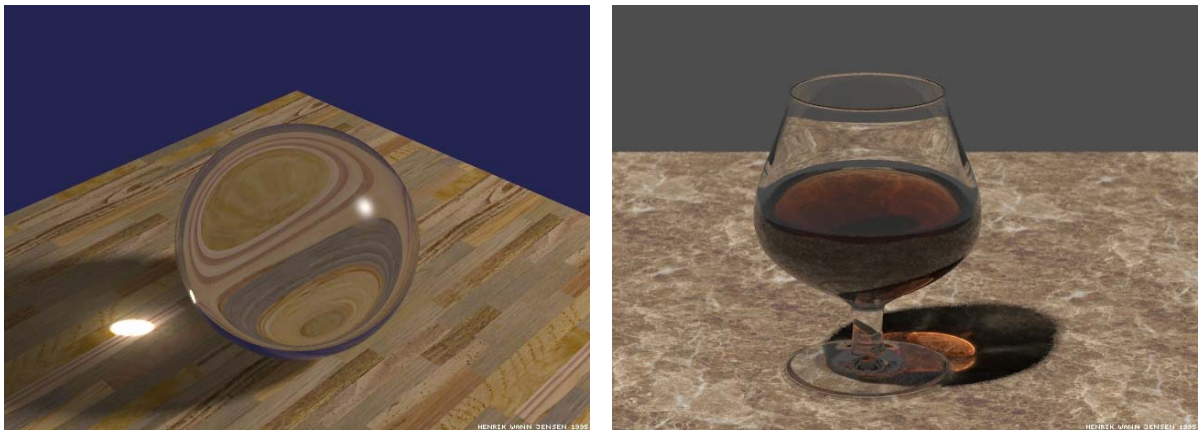
http://www.3dtotal.com/

3DMax



Caustics

Images created by Prof. Henrik Jensen @UCSD



Photon Mapping

Images created by Prof. Henrik Jensen @UCSD

The Light of Mies van der Rohe (Stanford Univ) Siggraph 2000

Subsurface scattering

Images created by Prof. Henrik Jensen @UCSD

2001