Ray Tracing
and
Radiosity

Reading Material

MUST read

- These slides
- OH 102-113 by Magnus Bondesson
  - Ray Tracing, ray/polygon intersection, Radiosity
- OH 264-280
  - Computational Geometry (beräkningsgeometri)
  - Voronoi regions, Delaunay triangulation (läs båda översiktligt)
  - Marching Cubes
  - CSG (Constructive Solid Geometry)

May also read:

- Angel, chapter 12
  - (12.6, 12.7 och 12.8 är överkurs)

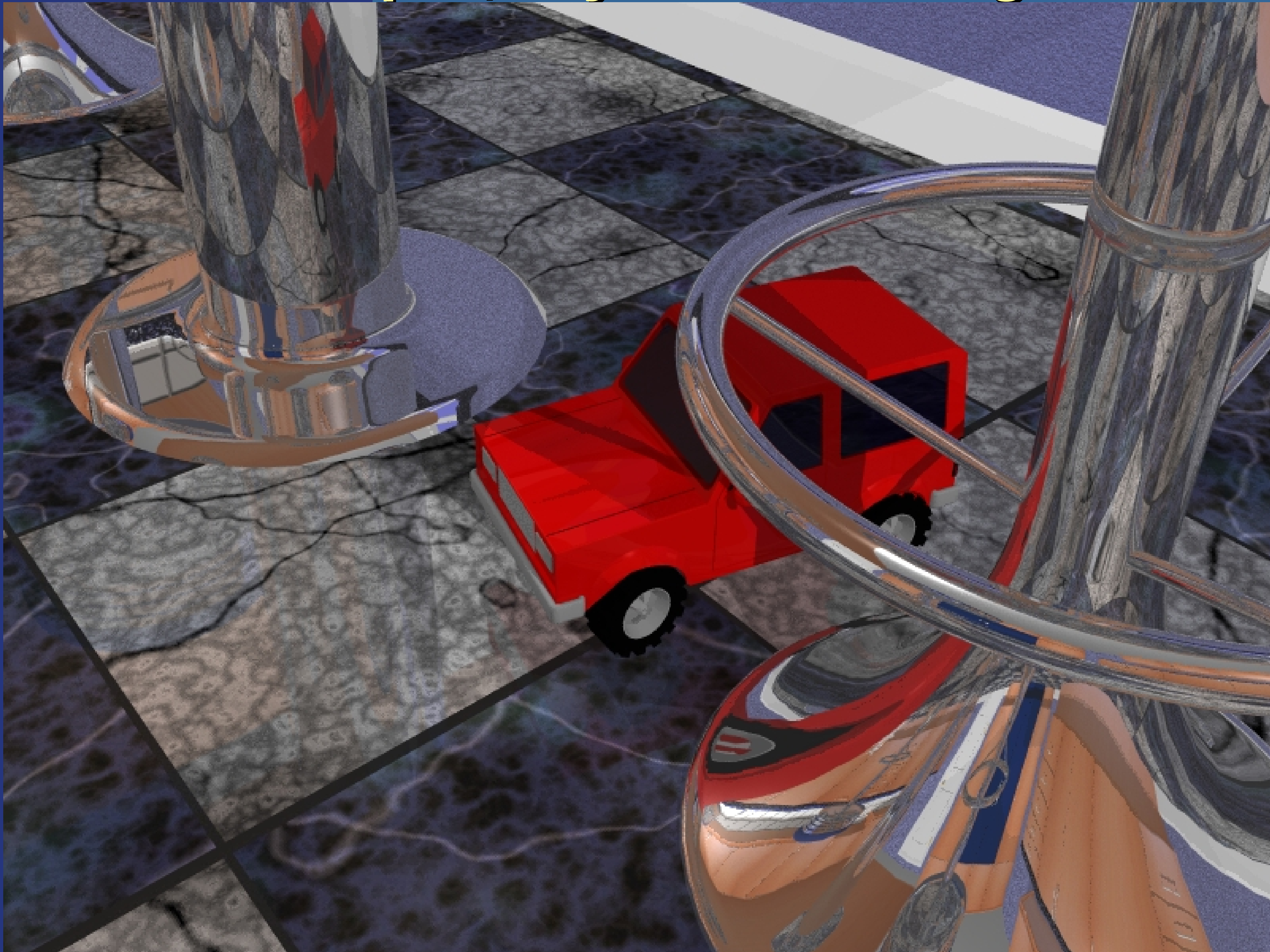
What is ray tracing?

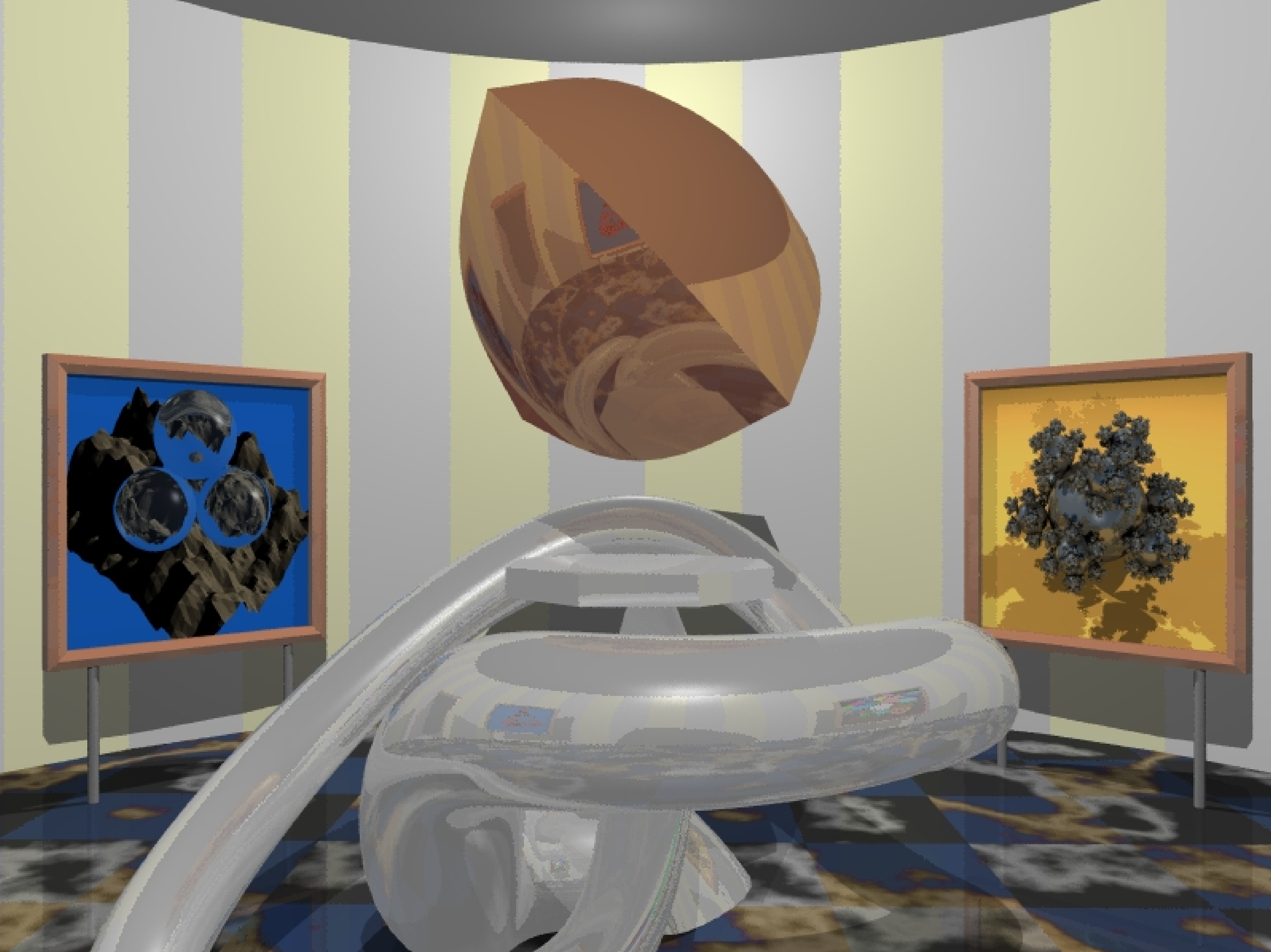
- Another rendering algorithm
  - Fundamentally different from polygon rendering (using e.g., OpenGL)
  - OpenGL
    - renders one triangle at a time
    - Z-buffer sees to it that triangles appear ”sorted” from viewpoint
    - Local lighting --- per vertex
  - Ray tracing
    - Renders one pixel at a time
    - Sorts per pixel
    - Global lighting equation (reflections, shadows)
    - Per pixel

What is the point of ray tracing?

- Higher quality rendering
  - Global lighting equation (shadows, reflections, refraction)
  - Accurate shadows, reflections, refraction
  - More accurate lighting equations
- Is the base for more advanced algorithms
  - Global illumination, e.g., photon mapping
- It is extremely simple to write a (naive) ray tracer
- A disadvantage: it is inherently slow!

Some simple, ray traced images…





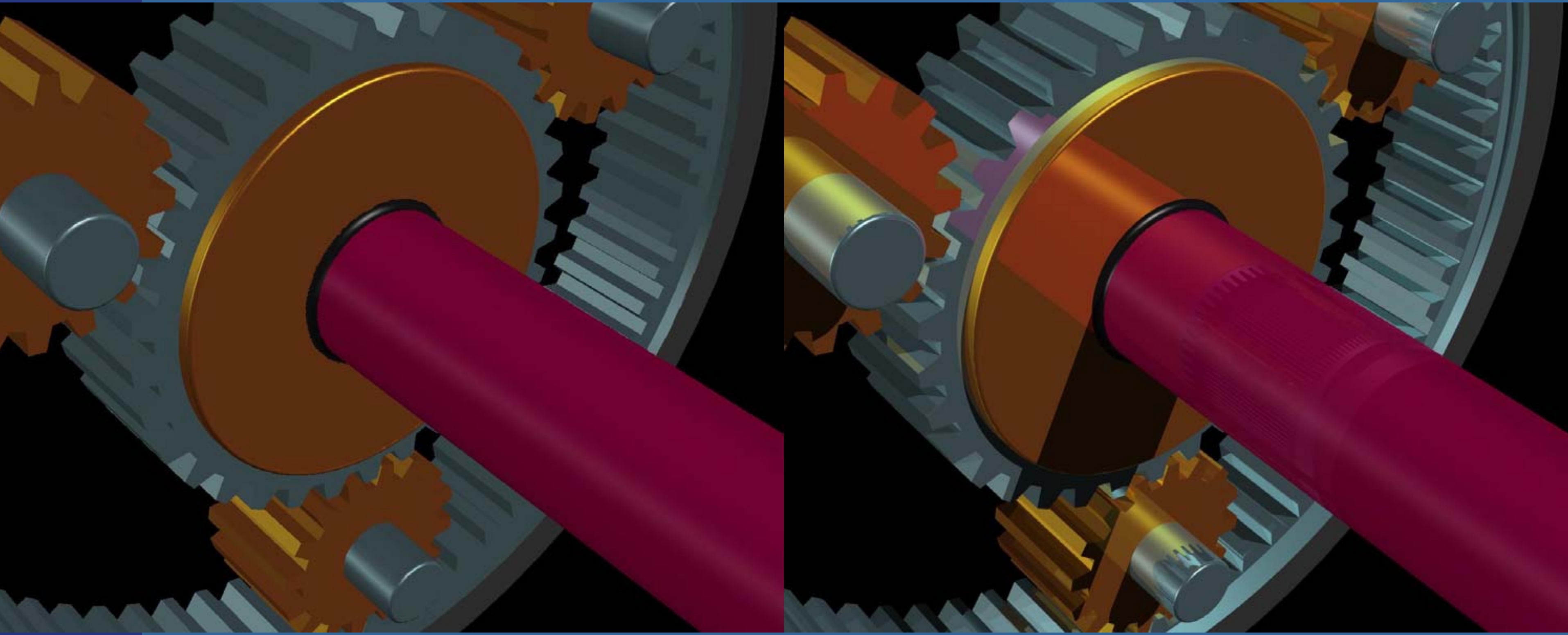
Again: it is simple to write a ray tracer! A la Paul Heckbert

```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*
A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s-->sph)b=vdot(D,U=vcomb(-1.,P,s->cen)),u=b*b-vdot(U,U)+s->rad*s
->rad,u=u>0?sqrt(u):1e31,u=b-u>1e-7?b-u:b+u,tmin=u>=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s->ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s->cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l-->sph)if((e=l
->kl*vdot(N,U=vunit(vcomb(-1.,P,l->cen))))>0&&intersect(P,U)==l)color=vcomb(e
,l->color,color);U=s->color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta*
eta*(1-d*d);return vcomb(s->kt,e>0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s->ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s->kd,
color,vcomb(s->kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}/*minray!*/
```

Which rendering algorithm will win at the end of the day?

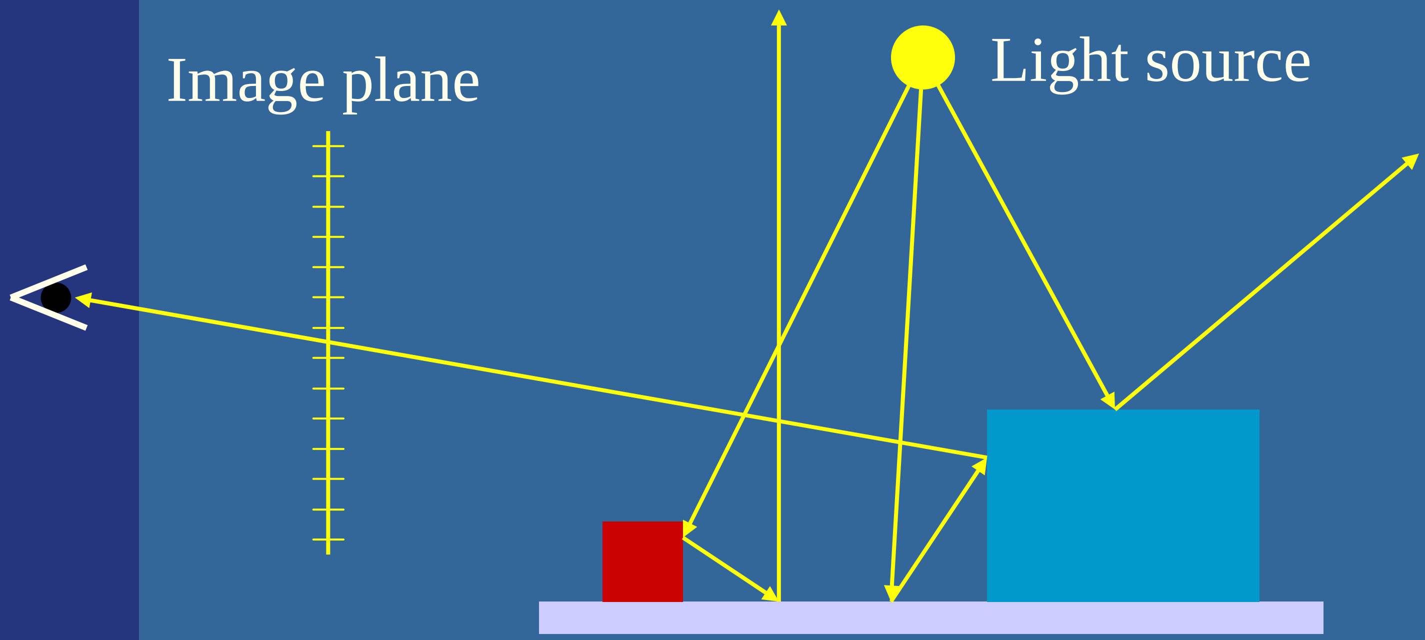
- Ray tracing or polygon rendering?
- Ray tracing is:
  - Slow
  - But realistic
  - Therefore, focus is on creating faster algorithms, and possible hardware
- Polygon rendering (OpenGL) is:
  - Fast (simple to build hardware)
  - Not that realistic
  - Therefore, focus is on creating more realistic images using graphics hardware
- Answer: right now, it depends on what you want, but for the future, no one really knows

Side by side comparison
Images courtesy of Eric Haines



To be physically correct, follow photons from light sources…

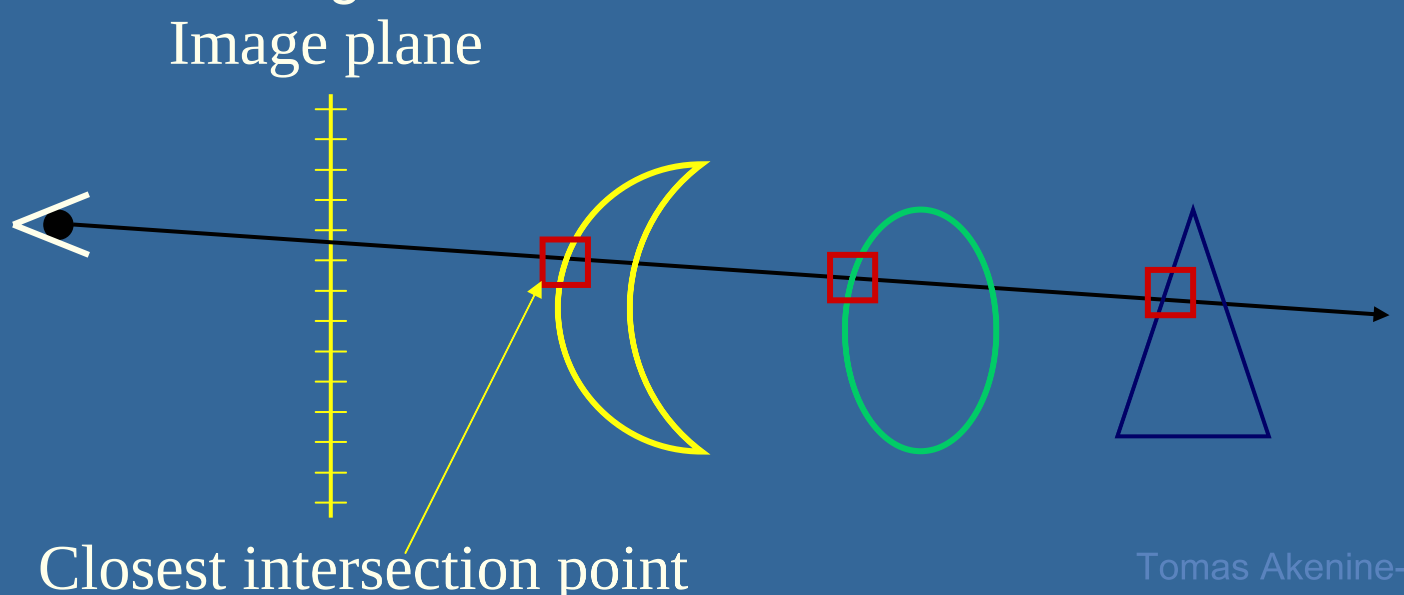
- Not what we do for a simple ray tracer
  - Though this is almost what we do for more advanced techniques (photon mapping)



- Not effective, not many rays will arrive at the eye

Follow photons backwards from the eye: treat one pixel at a time

- Rationale: find photons that arrive at each pixel
- How do one find the visible object at a pixel?
- With intersection testing
  - Ray, $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$, against geometrical objects
  - Use object that is closest to camera!
  - Valid intersections have $t > 0$
  - t is a signed distance



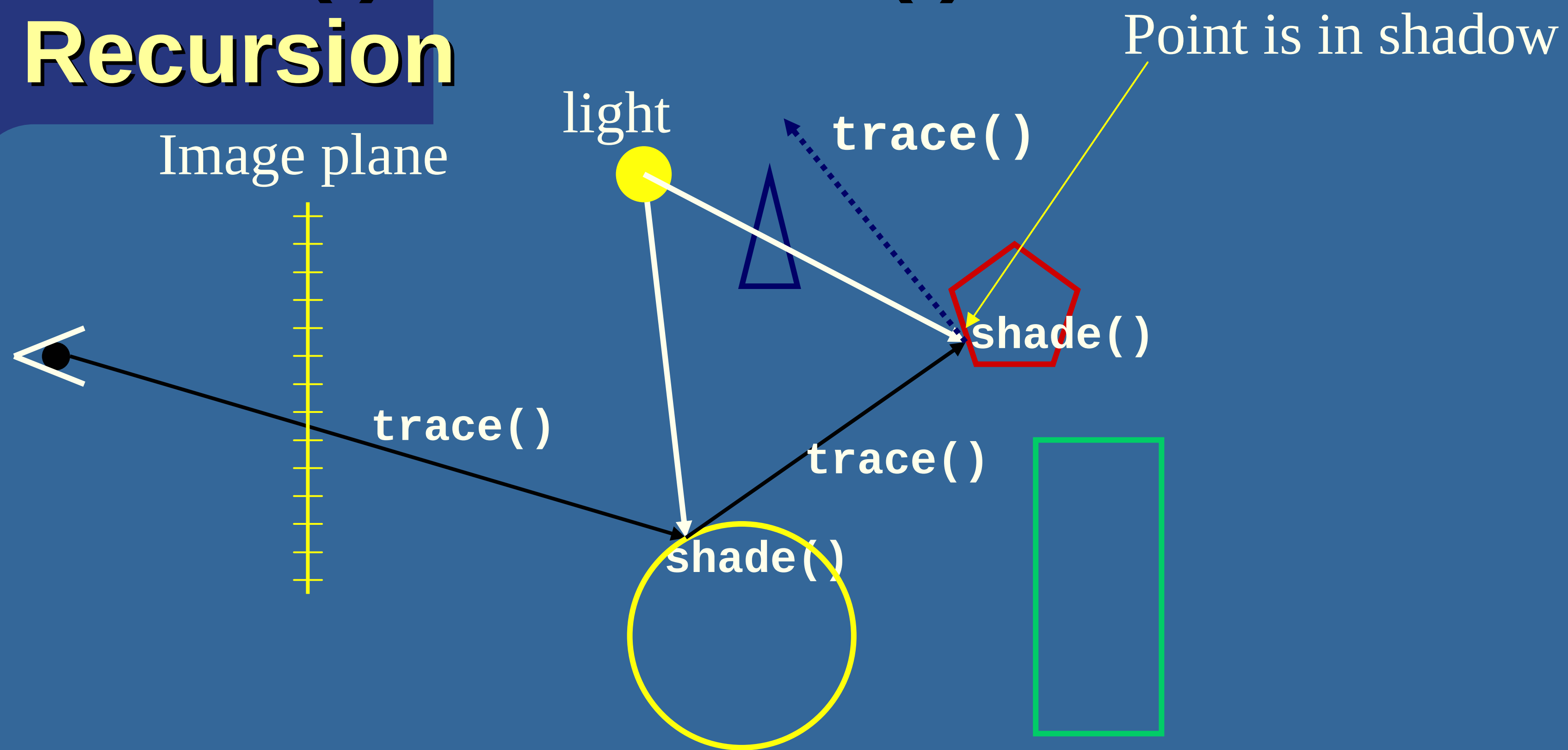
Finding closest point of intersection

- Naively: test all geometrical objects in the scene against each ray, use closest point
  - Very very slow!
- Be smarter:
  - Use spatial data structures, e.g.:
  - Bounding volume hierarchies
  - Octrees
  - BSP trees
  - Grids (not yet treated)
  - Or a combination (hierarchies) of those
- See Advanced Computer Graphics, EDA425, for more

`trace()` and `shade()`

- We now know how to find the visible object at a pixel
- How about the finding the color of the pixel?
- Basic ray tracing is essentially only two functions that recursively call each other
  - **`trace()`** and **`shade()`**
- **`trace()`**: finds the first intersection with an object and calls shade() for the hit point
- **`shade()`**: computes the lighting at that intersection point

trace() and shade(): Recursion



- First call `trace()` to find first intersection
- `trace()` then calls `shade()` to compute lighting
- `shade()` then calls `trace()` for reflection and refraction directions

trace() in detail

```
Color trace(Ray R)
{
      float t;    bool hit;
      Object O;
      Color col;
      Vector P,N; // point & normal at intersection point
      hit=findClosestIntersection(R,&t,&O);
      if(hit)
      {
            P=R.origin() + t*R.direction();
            N=computeNormal(P,O);
            // flip normal if pointing in wrong dir.
            if(dot(N,R.direction()) > 0.0) N=-N;
            col=shade(t,O,R,P,N);
      }
      else col=background_color;
      return col;
}
```

shade( ) computes lighting

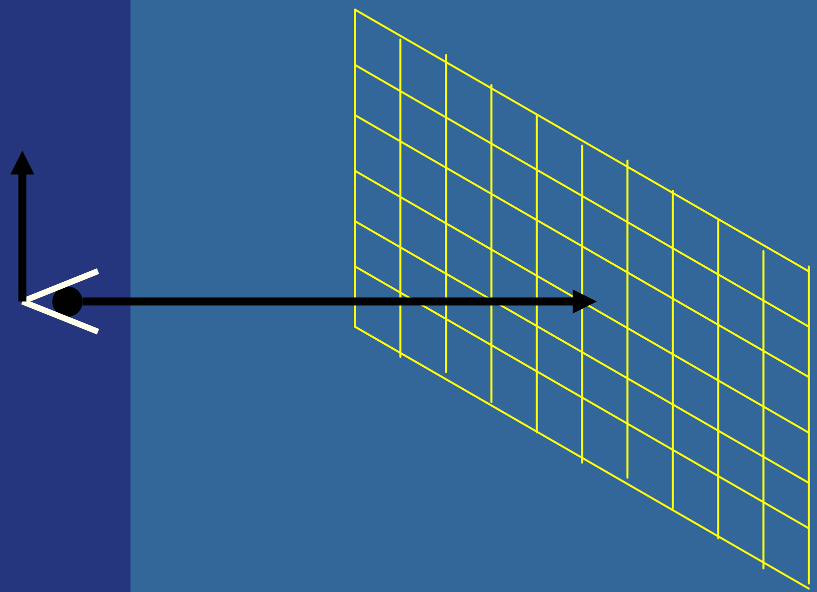
- For now, we will use the simple standard lighting equation that we used so far
- Ambient+Diffuse+Specular
- However, we also spawn new rays in the:
  - Reflection and
  - Refraction direction
- Can use more advanced models
  - Simple to exchange --- a strength of ray tracing

shade( ) in detail

```
Color shade(Ray R, Mtrl &m, Vector P,N)
{
      Color col;
      Vector N,P,refl,refr;
      for each light L
      {
            if(not inShadow(L,P))
                  col+=DiffuseAndSpecular();
      }
      col+=AmbientTerm();
      if(recursed_too_many_times()) return col;
      refl=reflectionVector(R,N);
      col+=m.specular_color()*trace(refl);
      refr=computeRefractionVector(R,N,m);
      col+=m.transmission_color()*trace(refr);
      return col;
}
```

Who calls `trace()` or `shade()`?

- Someone need to spawn rays
  - One or more per pixel
  - A simple routine, **`raytraceImage()`**, computes rays, and calls **`trace()`** for each pixel.



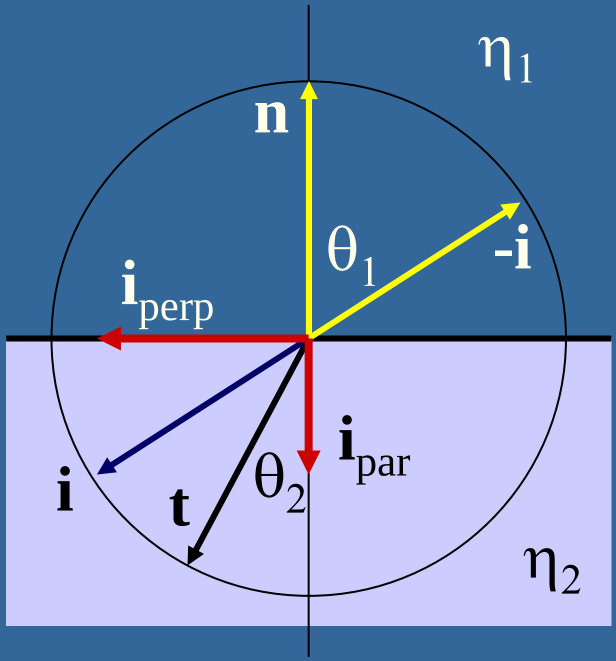
- Use camera parameters to compute rays
  - Resolution, fov, camera direction & position & up

When does recursion stop?

- Recurse until ray does not hit something?
  - Does not work for closed models
- One solution is to allow for max N levels of recursion
  - N=3 is often sufficient (sometimes 10 is sufficient)
- Another is to look at material parameters
  - E.g., if specular material color is (0,0,0), then the object is not reflective, and we don't need to spawn a reflection ray
  - More systematic: send a weight, w, with recursion
  - Initially w=1, and after each bounce, w*=O.specular_color(); and so on.
  - Will give faster rendering, if we terminate recursion when weight is too small (say <0.05)

Refraction: Need a transmission direction vector, t

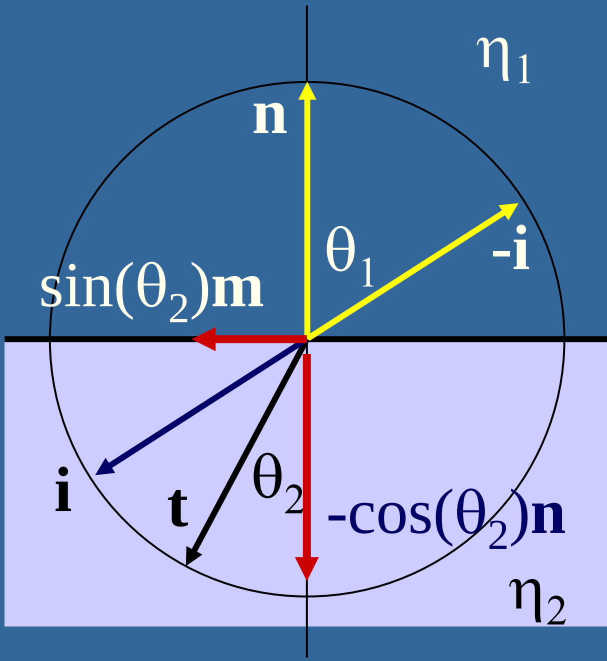
- **n**, **i**, **t** are unit vectors
- η_1 & η_2 are refraction indices
- $c_1=\cos(\theta_1)=-\mathbf{n}\cdot\mathbf{i}$
- Decompose **i** into:
- $\mathbf{i}_{par}=-c_1\mathbf{n}$, $\mathbf{i}_{perp}=\mathbf{i}+c_1\mathbf{n}$
- $\mathbf{t}=\sin(\theta_2)\mathbf{m} - \cos(\theta_2)\mathbf{n}$, where
- $\mathbf{m}=\mathbf{i}_{perp}/\|\mathbf{i}_{perp}\|=(\mathbf{i}+c_1\mathbf{n})/\sin(\theta_1)$



Known as Heckbert's method

Modified by Ulf Assarsson, 2004

Refraction:



$c_1=\cos(\theta_1)=-\mathbf{n}\cdot\mathbf{i}$

- $\mathbf{i}_{par}=-c_1\mathbf{n}$, $\mathbf{i}_{perp}=\mathbf{i}+c_1\mathbf{n}$
- $\mathbf{t}=\sin(\theta_2)\mathbf{m} - \cos(\theta_2)\mathbf{n}$, where
- $\mathbf{m}=\mathbf{i}_{perp}/||\mathbf{i}_{perp}||=(\mathbf{i}+c_1\mathbf{n})/\sin(\theta_1)$
- Use Snell's law:
  - $\sin(\theta_2)/\sin(\theta_1)= \eta_1/\eta_2 = \eta$
- => $\mathbf{t} = \sin(\theta_2)\ (\mathbf{i}+c_1\mathbf{n})/\sin(\theta_1) - \cos(\theta_2)\mathbf{n}$,
- i.e., $\mathbf{t} = \eta\mathbf{i} + (\eta c_1 - c_2)\ \mathbf{n}$, where $c_2 =\cos(\theta_2)$
- Simplify: $c_2=\mathbf{sqrt}[\ 1 - \eta^2(1-c_1{}^2)\ ]$
  - Pythagoras: $\cos(\theta_2)^2 = 1^2 - \sin(\theta_2)^2$
  - $\sin(\theta_2) = \eta\ \sin(\theta_1)$
  - $\sin(\theta_2)^2 = \eta^2\ (1-\cos(\theta_1)^2)$

Known as Heckbert's method

Modified by Ulf Assarsson, 2004

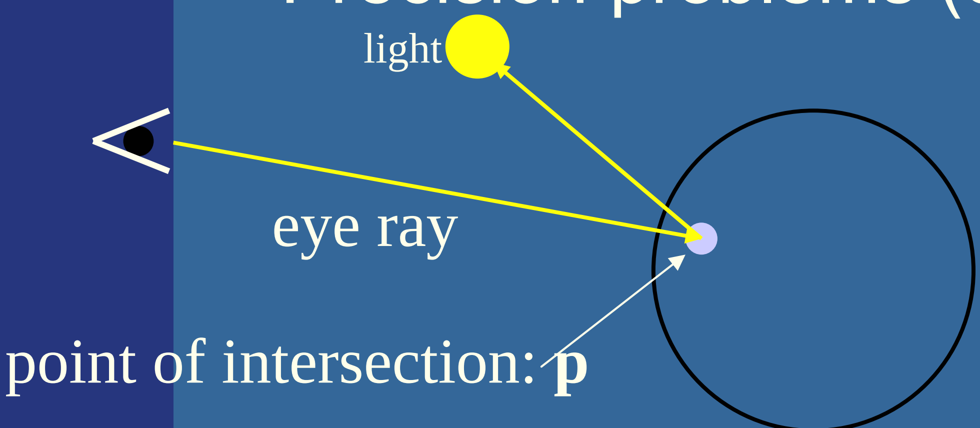
Image with a refractive object

Some refraction indices, η

- Measured with respect to vacuum
  - Air: 1.0003
  - Water: 1.33
  - Glass: around 1.45 – 1.65
  - Diamond: 2.42
  - Salt: 1.54
  - Lead (bly): 2.6
- Note 1: the refraction index varies with wavelength, but we often only use one index for all three color channels, RGB
- Note 2: can get Total Internal Reflection (TIR)
  - Means no transmission, only reflection
  - Occurs when c_2 is imaginary (see formula 2 slides back)

In `trace()`, we need a function `findClosestIntersection()`

- Use intersection testing (from a previous lecture) for rays against objects
- Intersection testing returns signed distance(s), *t*, to the object
- Use the *t* that is smallest, but *>0*
- Naive: test all objects against each ray
  - Better: use spatial data structures (more later)
- Precision problems (exaggerated):



The point, **p**, will be incorrectly self-shadowed, due to imprecision

Solution: after **p** has been computed, update as: $\mathbf{p}'=\mathbf{p}+\varepsilon\mathbf{n}$
($\mathbf{n}$ is normal at p, ε is small number >0)

In `shade()`, we need a function `inShadow()`

- Compute distance from intersection point, **p**, to light source: t_{max}
- Then use intersection:
  - Point is in shadow if $0 < t < t_{max}$ is true for at least one object

More info…

- This was ray tracing at it simplest
- We can do lots more…
  - Faster
  - More realistic
  - Better filtering and sampling
  - More advanced geometry (spheres, cylinder, Bezier surfaces, etc) – not only triangles
  - Programmable shading is easy
- Some nice books on this topic:
  - Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989.
  - Shirley, *Realistic Ray Tracing*, AK Peters, 2000.
  - Jensen, *Realistic Image Synthesis using Photon Mapping*, AK Peters, 2001.

Real-Time Ray Tracing

- Low level optimizations
  - SSE
  - Precomputation of constants per frame, e.q., ray-sphere test, primary rays
- Low resolution (320x200 – 640x400)
- Adaptive sub sampling
- Frameless rendering (motion blur)
- Others, like reprojection, reuse shading computations, simple shadows, single-level reflections...

DEMO

The following slides are from

MIT EECS 6.837, Popović

http://courses.csail.mit.edu/6.837/lect/October_27.pdf

Radiosity

- Treats fully diffuse indirect illumination (illumination from fully diffuse reflections)

direct illumination
(0 bounces)

1 bounce

2 bounces

images by Micheal Callahan
http://www.cs.utah.edu/~shirley/classes/cs684_98/students/callahan/bounce/

Radiosity

Careful calibration and measurement allows for comparison between physical scene & simulation

photograph

simulation

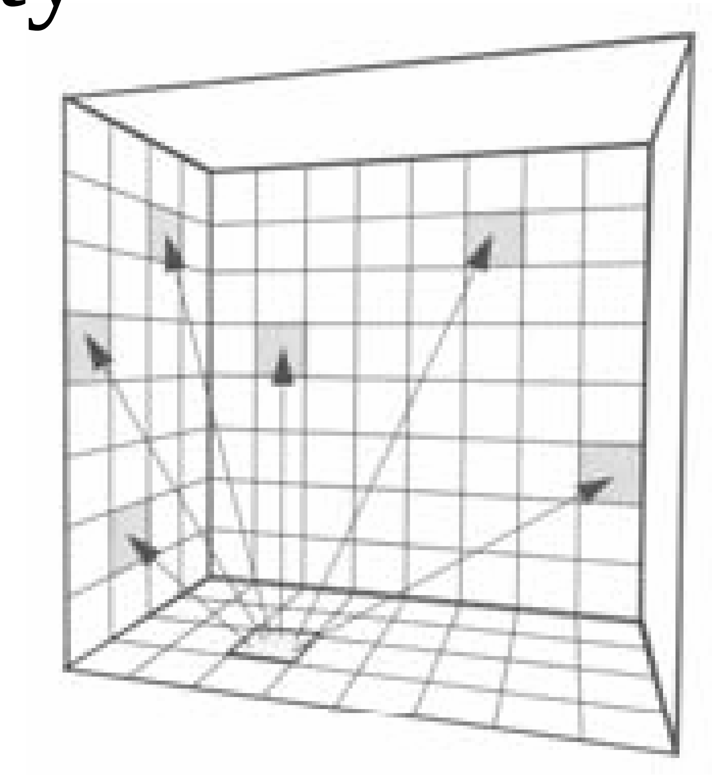
Light Measurement Laboratory
Cornell University, Program for Computer Graphics

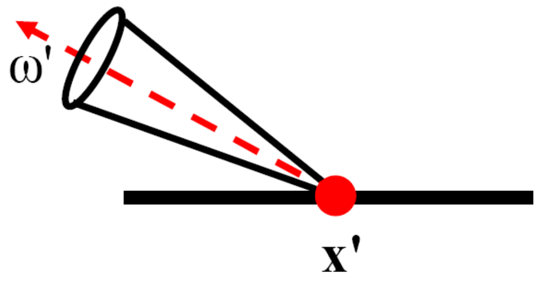
Radiosity

Prerequisites:

- Surfaces are assumed to be perfectly Lambertian (diffuse)
  – reflect incident light in all directions with equal intensity
- The scene is divided into a set of small areas, or patches.
- The radiosity, B_i, of patch *i* is the total rate of energy leaving a surface.
- Units for radiosity:

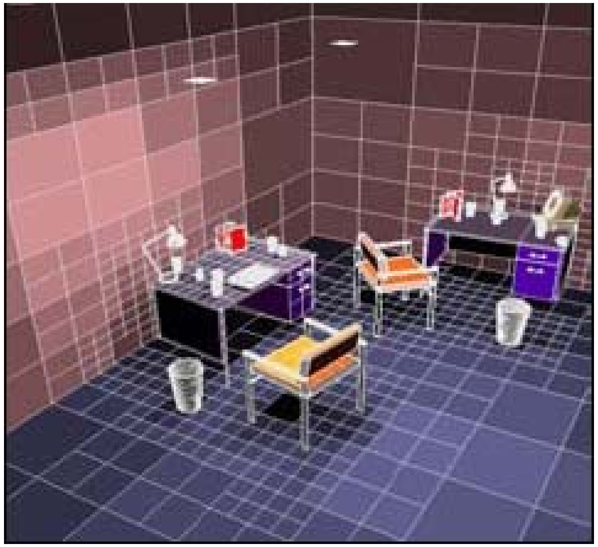
  Watts / steradian * meter2





Discrete Radiosity Equation

Discretize the scene into n patches, over which the radiosity B_i is constant



$$B_i = E_i + \overbrace{\rho_i}^{\text{reflectivity}} \sum_{j=1}^{n} \underbrace{F_{ij}}_{\text{form factor}} B_j$$

n simultaneous equations with n unknown B_i values can be written in matrix form:

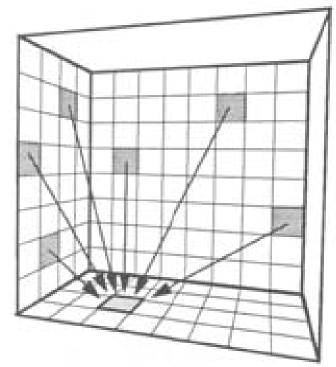
$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & & \\ \vdots & & \ddots & \\ -\rho_n F_{n1} & \cdots & \cdots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

A solution yields a single radiosity value B_i for each patch in the environment, a view-independent solution.

Solving the Radiosity Matrix

The radiosity of a single patch *i* is updated for each iteration by *gathering* radiosities from all other patches:

$$\begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_i \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_i \\ \vdots \\ E_n \end{bmatrix} + \begin{bmatrix} & & & \\ & & & \\ & & & \\ \rho_i F_{i1} & \rho_i F_{i2} & \cdots & \rho_i F_{in} \\ & & & \\ & & & \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_i \\ \vdots \\ B_n \end{bmatrix}$$

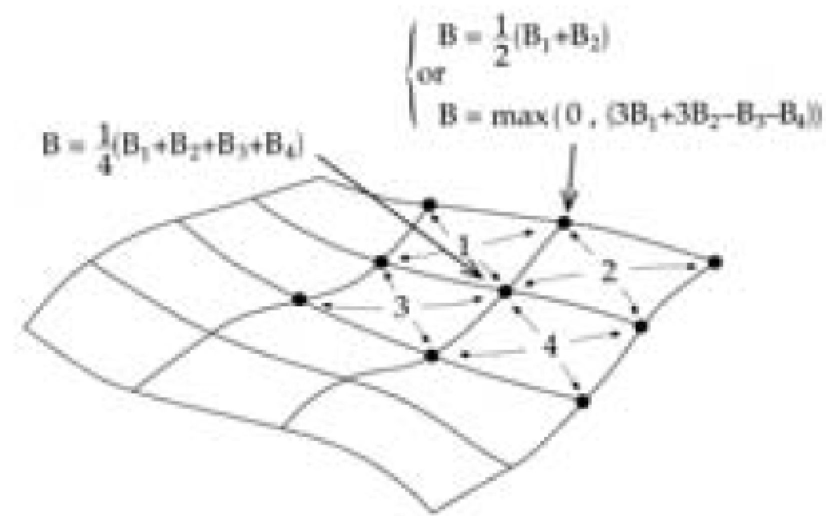


This is a Gauss-Seidel method for solving linear equations.

Computing Vertex Radiosities

To get smooth shading

- B_i radiosity values are constant over the extent of a patch.
- How are they mapped to the vertex radiosities (intensities) needed by the renderer?
  - Average the radiosities of patches that contribute to the vertex
  - Vertices on the boundary are assigned radiosity values by extrapolation

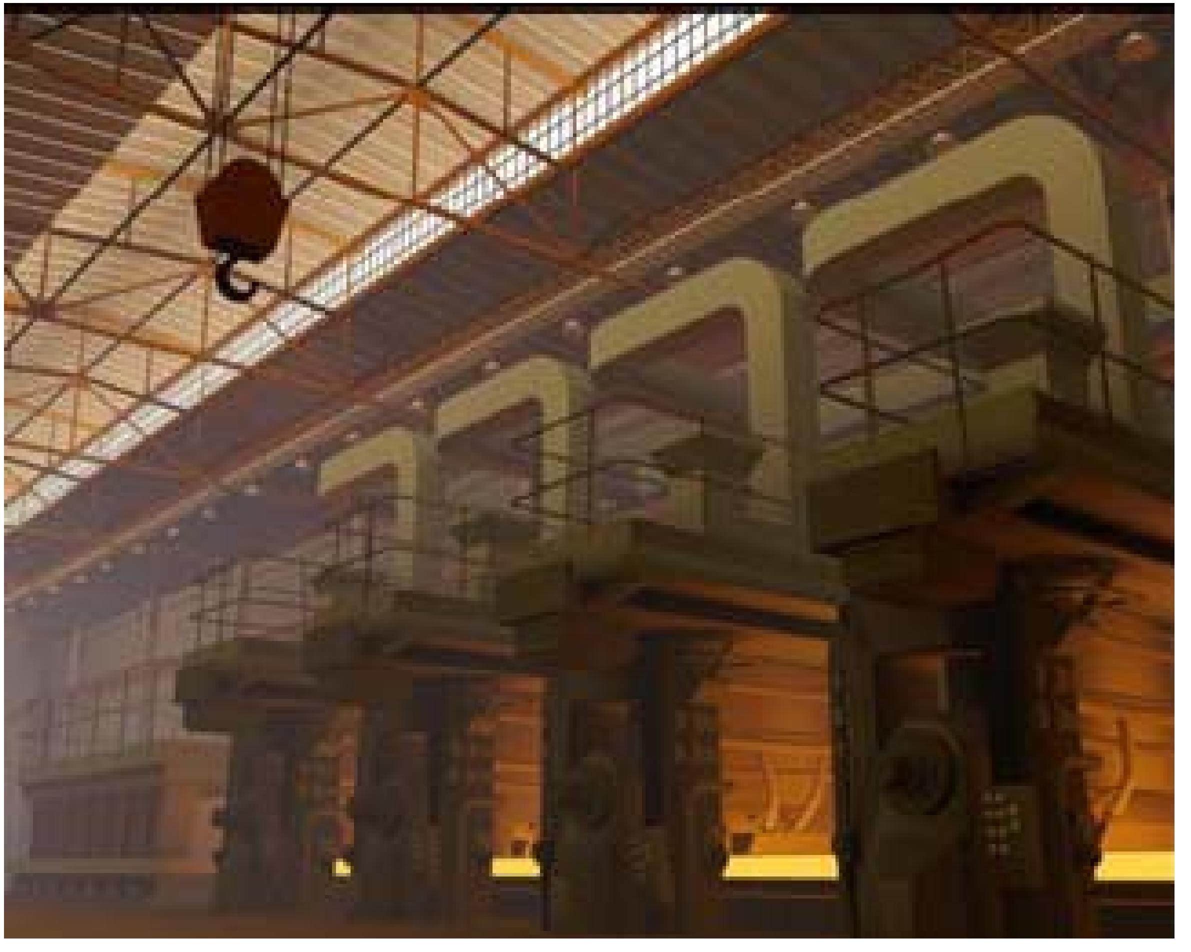


Radiosity

Museum simulation. Program of Computer Graphics, Cornell University.

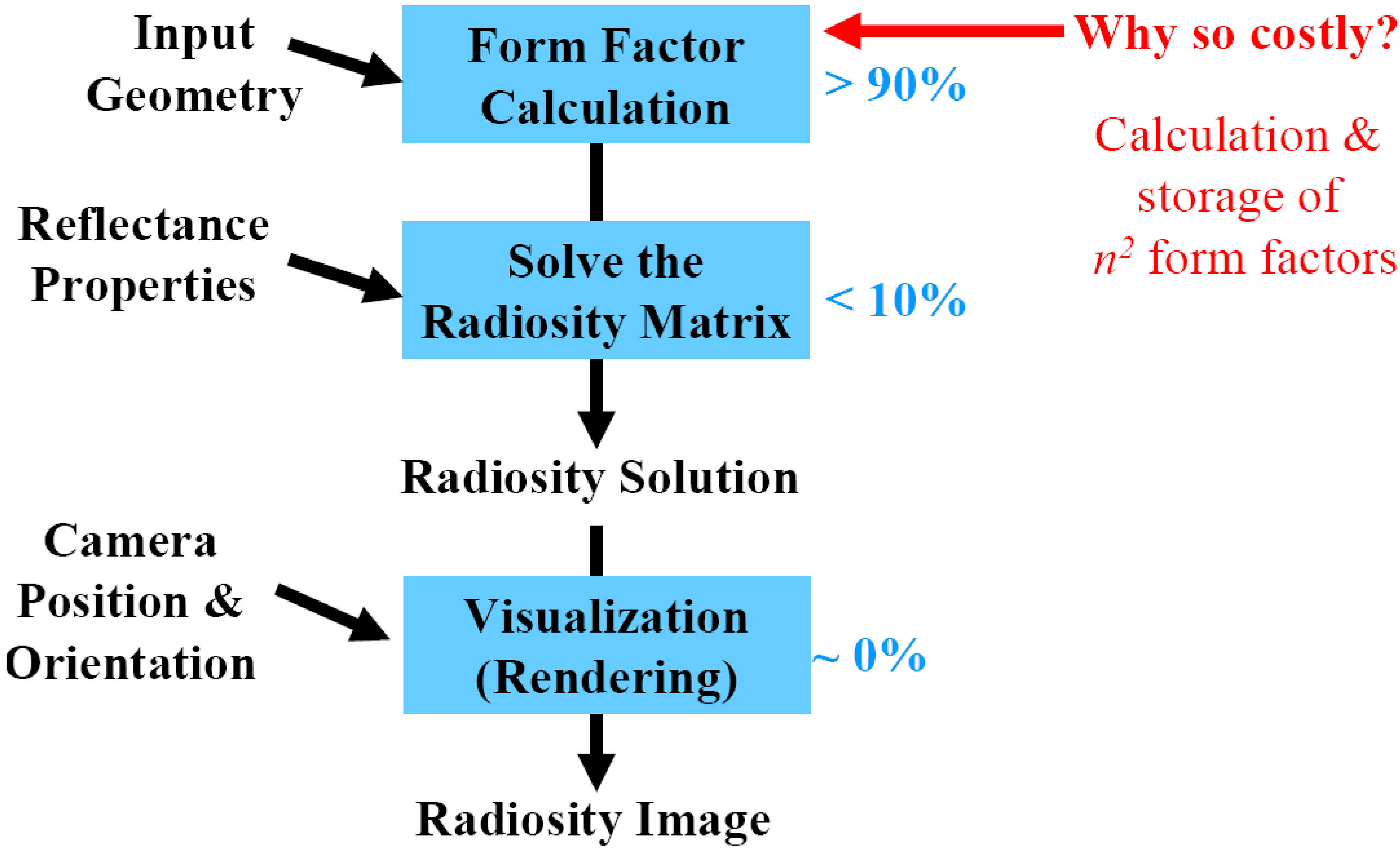
50,000 patches. Note indirect lighting from ceiling.

Radiosity



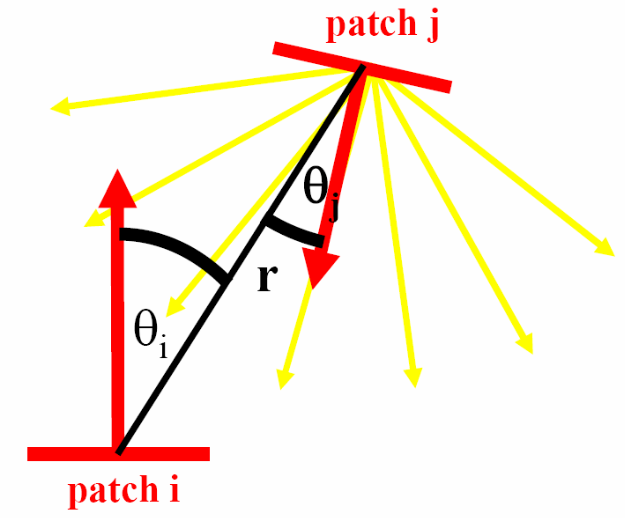
Factory simulation. Program of Computer Graphics, Cornell University. 30,000 patches.

Stages in a Radiosity Solution



Calculating the Form Factor F_{ij}

F_{ij} = fraction of light energy leaving patch j that arrives at patch i

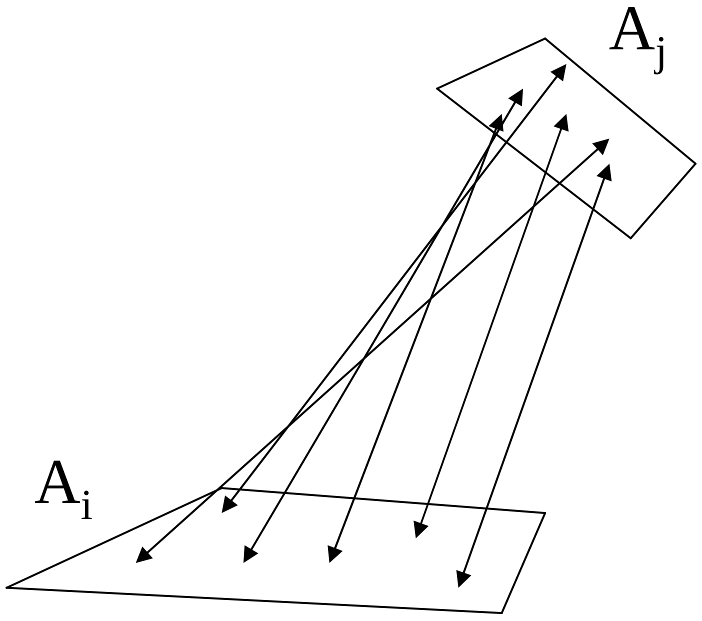


$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \cos\theta_j}{\pi r^2} V_{ij} \, dA_j \, dA_i$$

Form Factor from Ray Casting

- Cast *n* rays between the two patches
  - *n* is typically between 4 and 32
  - Compute visibility
  - Integrate the point-to-point form factor

- Permits the computation of the patch-to-patch form factor, as opposed to point-to-patch



One final example image

Lightscape **http://www.lightscape.com**