Advanced Computer Graphics
Ray-Tracing

G. Zachmann
University of Bremen, Germany
cgvr.informatik.uni-bremen.de

Effects Needed for Realistic Rendering

- Remember one of the local lighting models from CG1?
- All local lighting models fail to render one of the following effects:
  - (Soft) Shadows (Halbschatten)
  - Reflection on glossy surfaces, e.g., mirrors (Reflexionen)
  - Refraction, e.g., on water or glass surfaces (Brechung)
  - Indirect lighting (sometimes in the form of *"color bleeding"*)
  - Diffraction (Beugung)
  - ...

➢ Global Illumination

The Rendering Equation

- Goal: photorealistic rendering
- The "solution": the rendering equation

[Kajiya, Siggraph 1986]

$$L_r(x,\omega_r) = L_e(x,\omega_r) + \int_\Omega \rho(x,\omega_r,\omega_i) L_i(x,\omega_i)\cos(\omega_i)d\omega_i$$

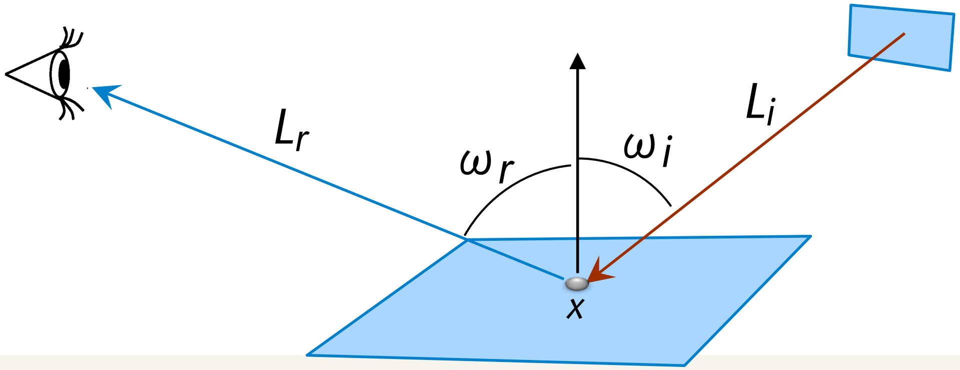
L_i = the "amount" of light *incident* on x from direction ω_i

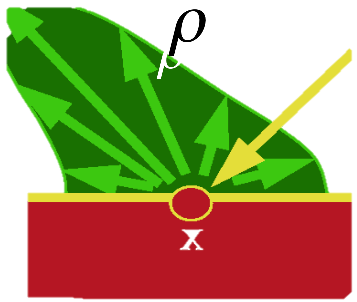
L_e = the "amount" of light *emitted* (i.e., "produced") from x into direction ω_r

L_r = the "amount" of light *reflected* from x into direction ω_r

ρ = function of the reflection coefficient (= BRDF, see CG1)

Ω = hemisphere around the normal





Approximations to the Rendering Equation

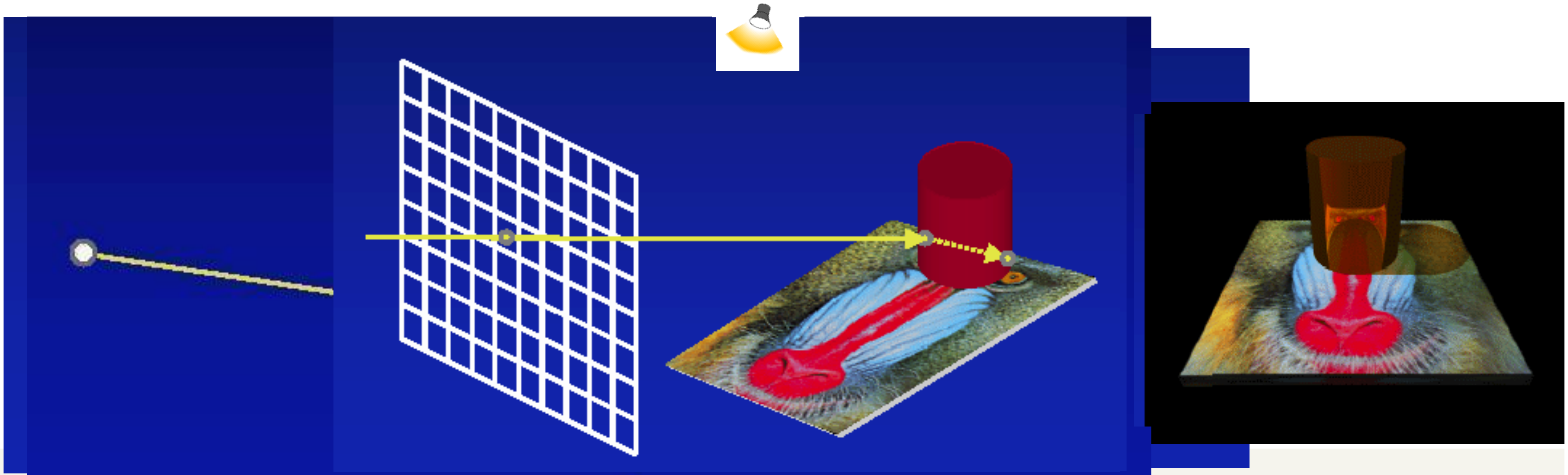
- Solving the rendering equation is impossible!
- Observation: the rendering equation is a recursive function
- Consequently, a number of approximation methods have been developed that are based on the idea of following rays:
  - Ray tracing [Whitted, Siggraph 1980, "An Improved Illumination Model for Shaded Display"]
  - Radiosity [Goral et. al, Siggraph 1984, "Modeling the Interaction of Light between diffuse Surface"]
- Current state of the art:
  - Ray-tracing, combined with photon tracing, combined with Monte Carlo methods

Turner Whitted,
Microsoft Research

The Simple "Whitted-style" Ray-Tracing

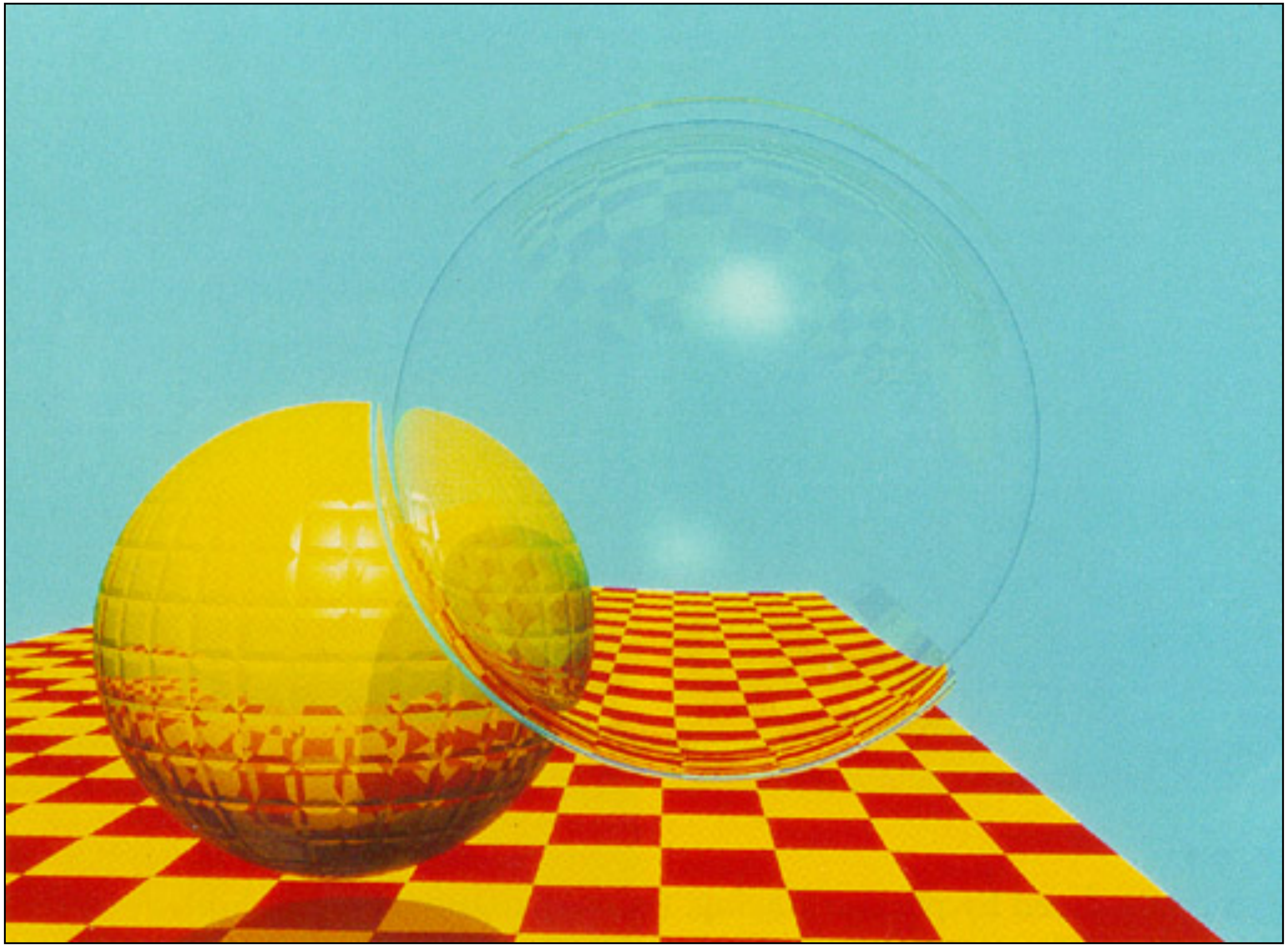
- Synthetic camera = viewpoint + image plane in world space

1. Shoot rays from camera through every pixel into scene (primary rays)
2. If the ray hits more than one object, then consider only the first hit
3. From there, shoot rays to all light sources (*shadow feelers*)
4. If a shadow feeler hits another obj → point is in shadow w.r.t. that light source. Otherwise, evaluate a lighting model (e.g., Phong [see CG1])
5. If the hit obj is glossy, then shoot reflected rays into scene (secondary rays) → recursion
6. If the hit object is transparent, then shoot refracted ray → more recursion



- Assumptions in the simple Whitted-style ray-tracing:
  - Point light sources
  - Many more …
- Limitations: can model only ..
  - Specular (ideal) reflections,
  - Perfect refractions,
  - Hard shadows

One of the First Ray-Traced Images



Turner Whitted 1980

A Little Bit of Ray-Tracing Folklore

- The principle of ray-tracing is so easy that you can write a "complete" ray-tracer on the back of a business card:

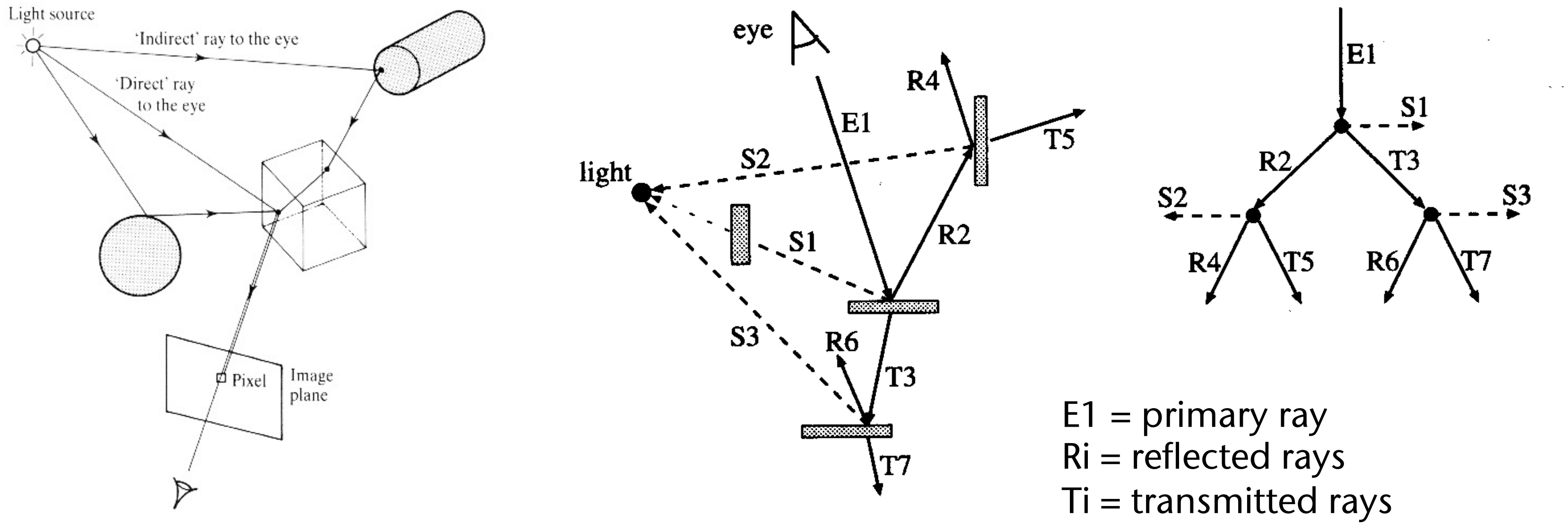
```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
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!*/
```

(Also won the *International Obfuscated C Code Contest*)

[Paul Heckbert, ca. 1994]

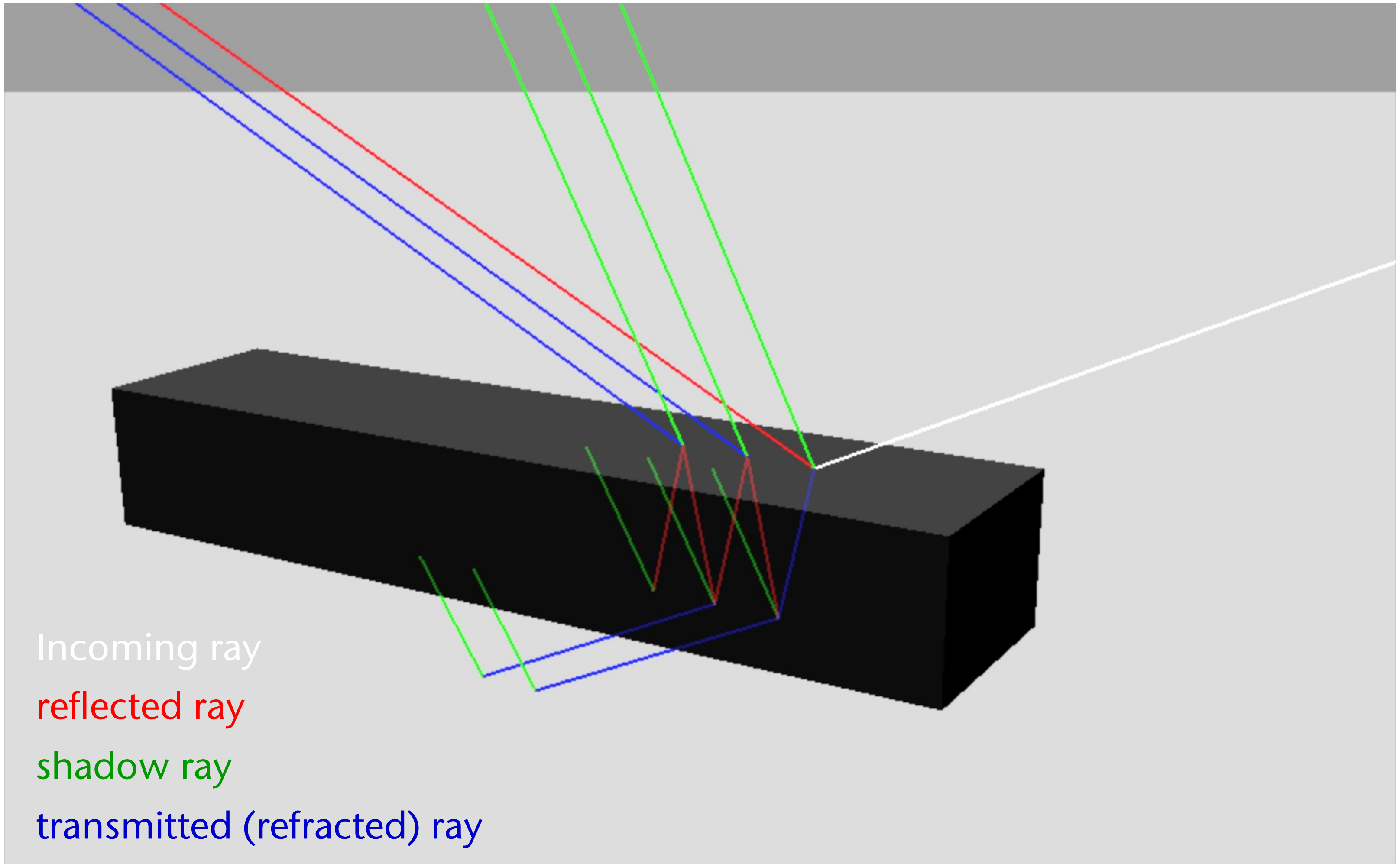
The Ray Tree

- Basic idea of ray-tracing: construct ray paths from the light sources to the eye, but follow those paths "backwards"
- Leads (conceptually!) to a tree, the ray tree:

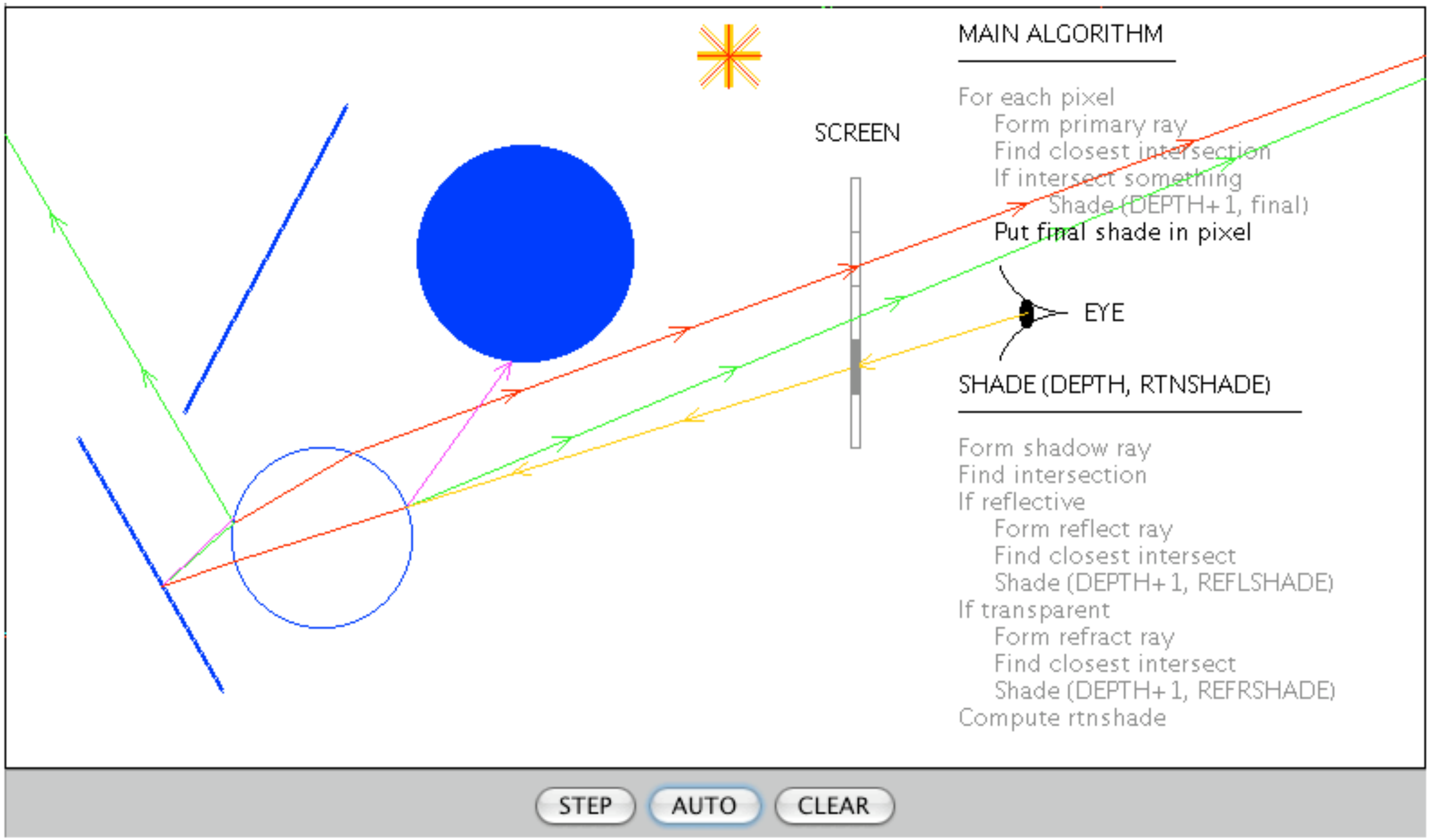


E1 = primary ray
Ri = reflected rays
Ti = transmitted rays
Si = shadow rays

- Visualizing the ray tree can be very helpful for deubgging



Interactive Demo

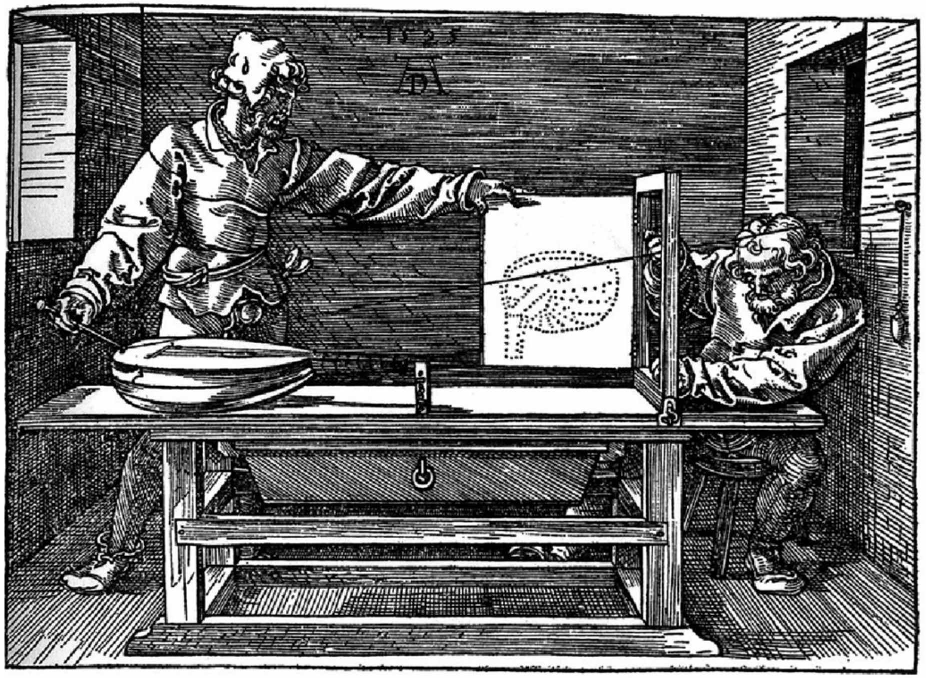


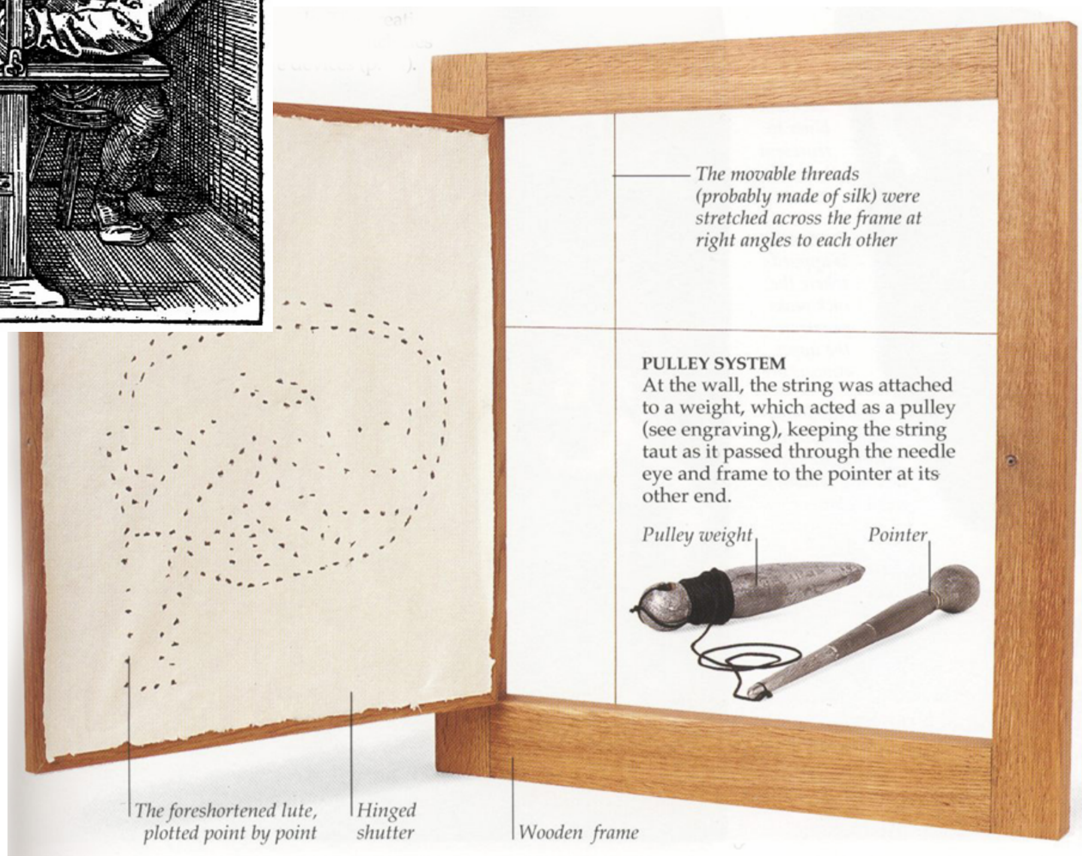
http://www.siggraph.org/education/materials/HyperGraph/raytrace/rt_java/raytrace.html

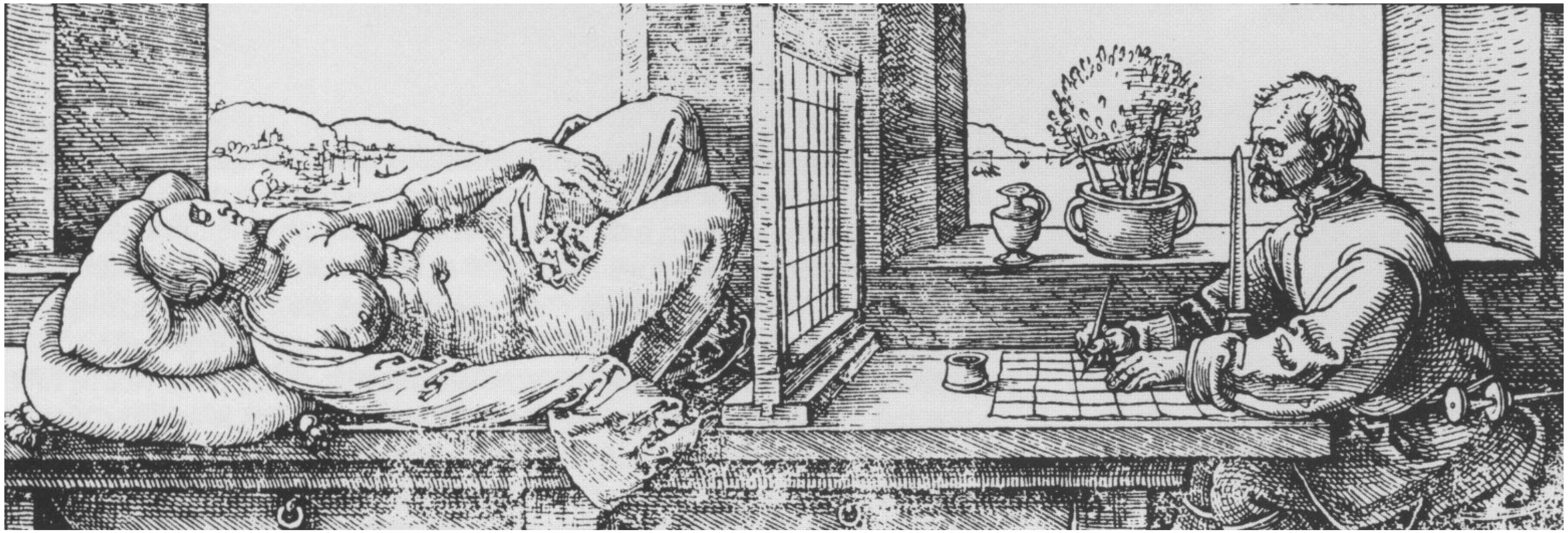
Digression

- The ancient explanation for our capability of seeing: "seeing rays"

Albrecht Dürer's "Ray Casting Machines" [16th century]

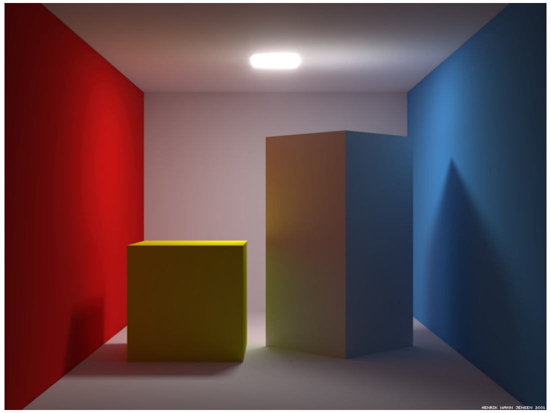


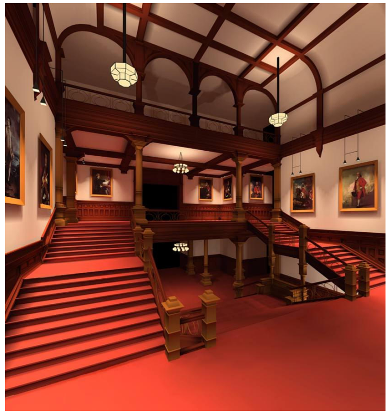


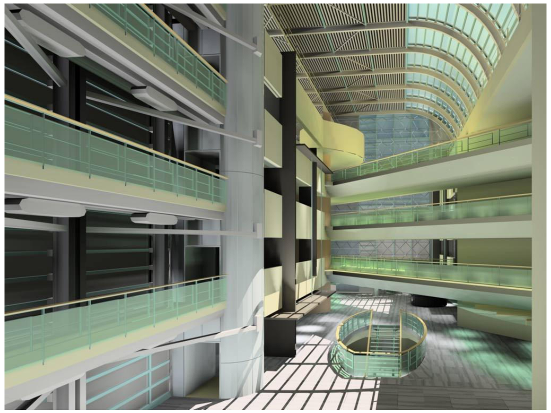


Examples of Ray-Traced Images



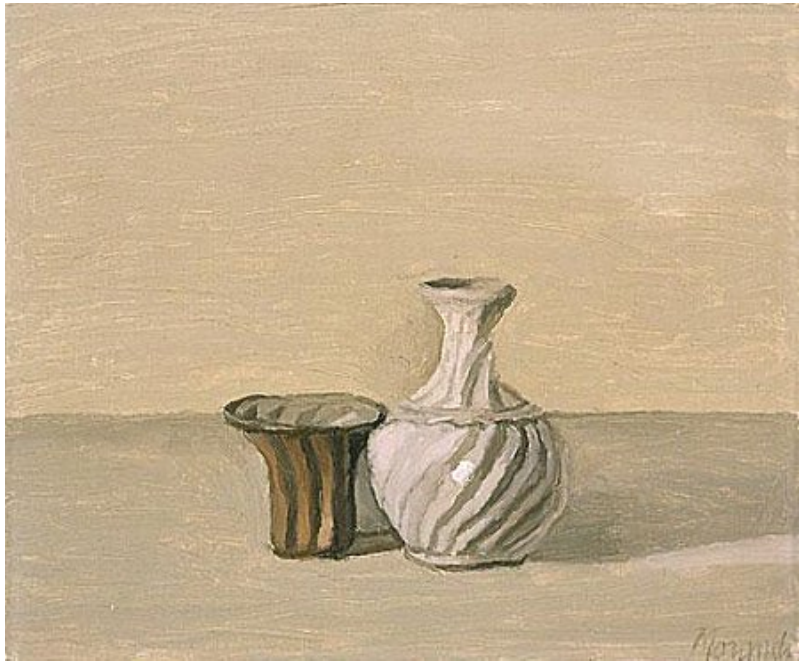


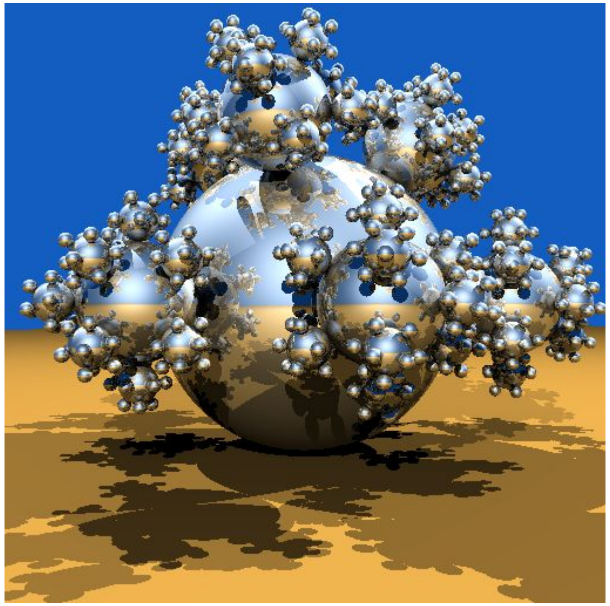



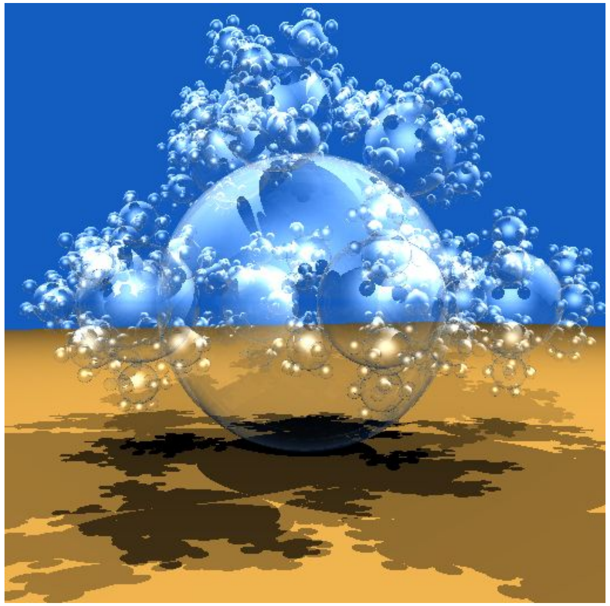


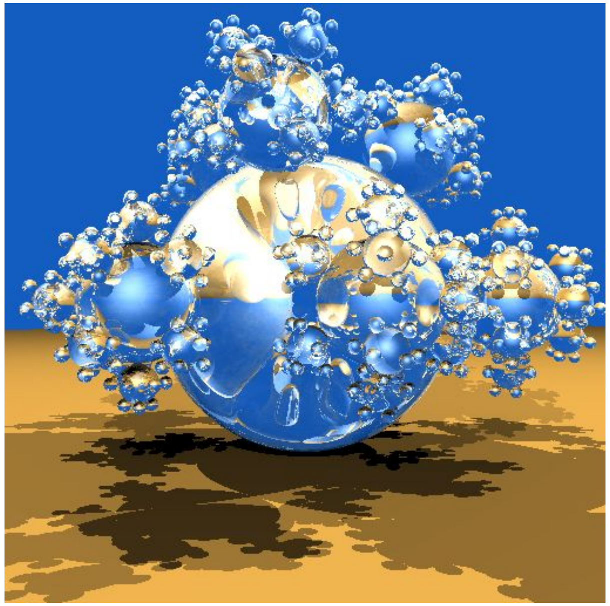

Jensen, Lightscape

Intermission: Giorgio Morandi

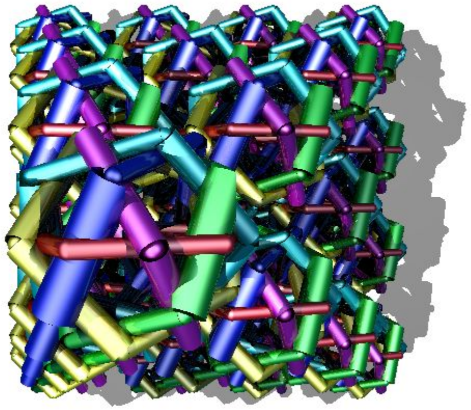


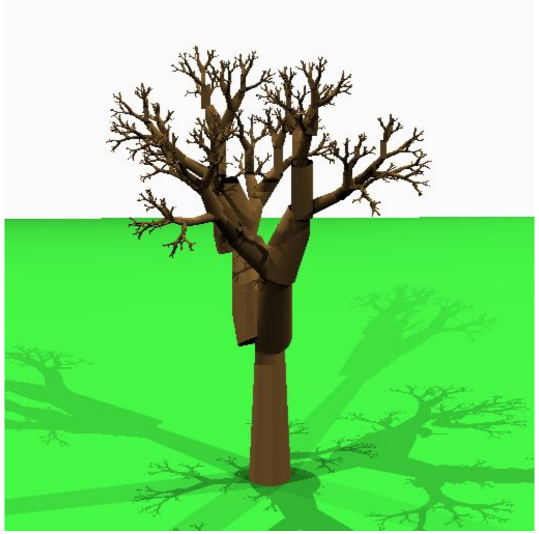
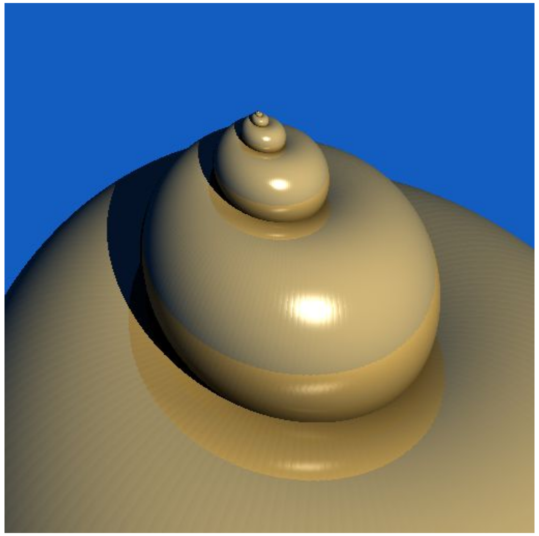
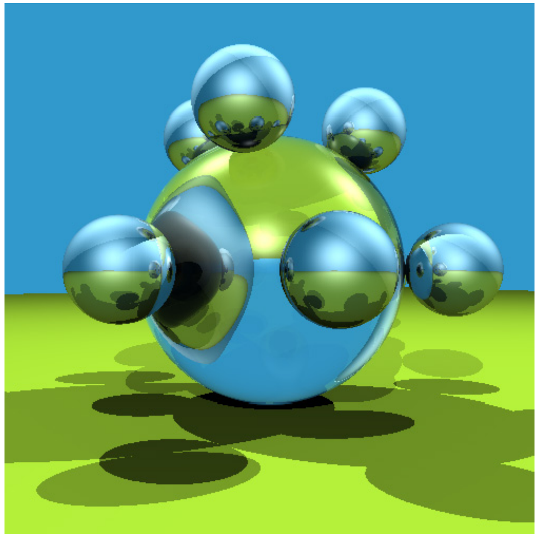


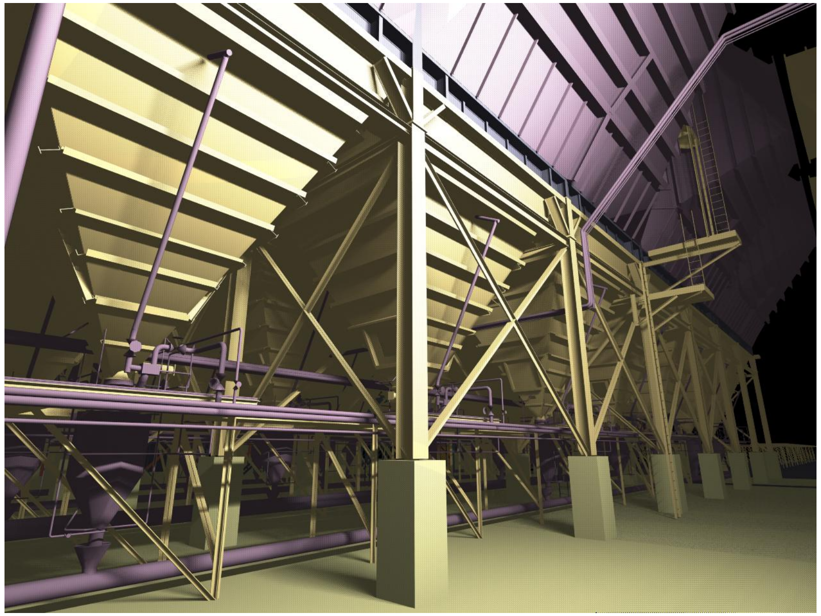




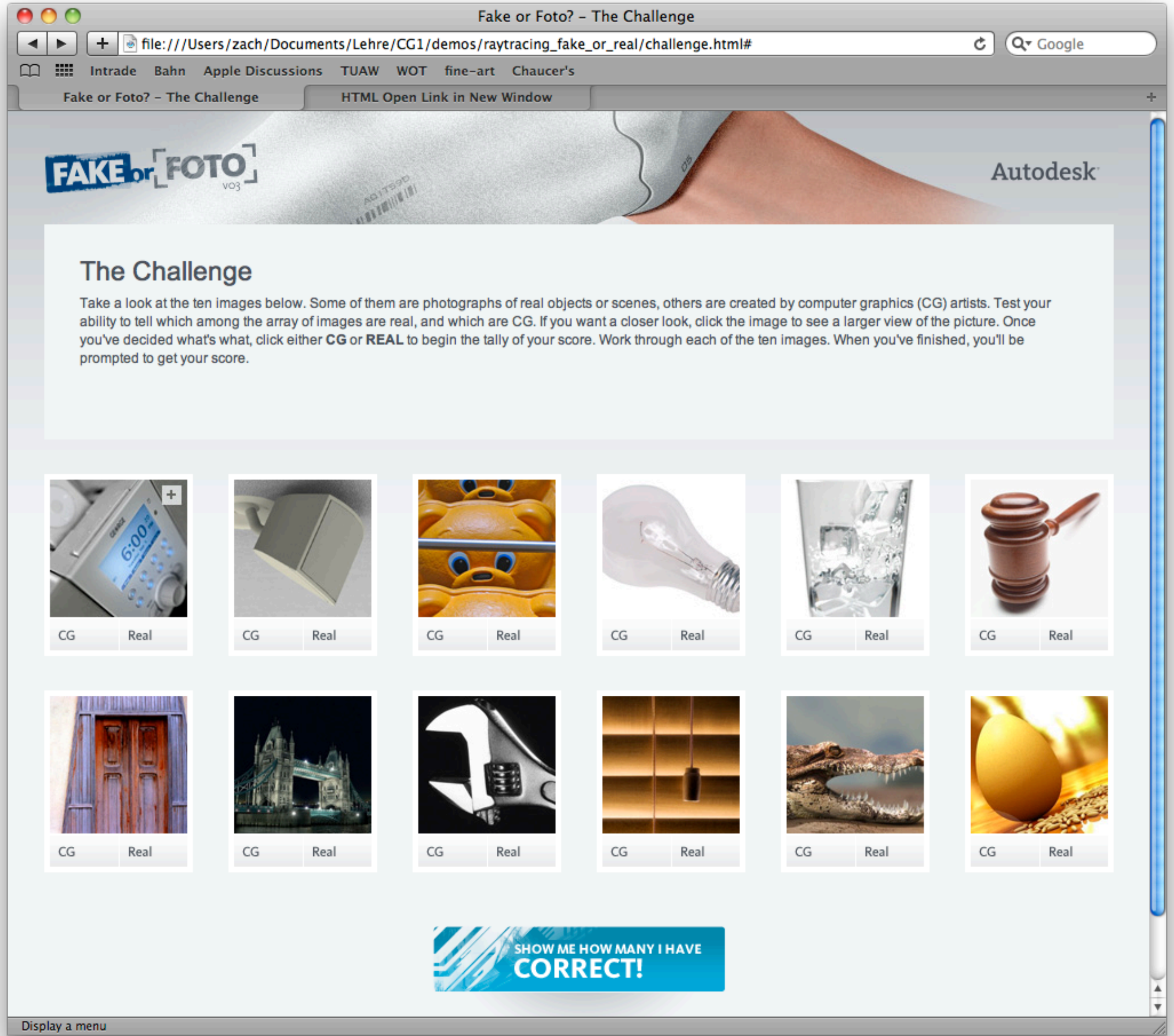
The "sphere flake" from the *standard procedural databases* (SPD) by Eric Haines [http://www.acm.org/tog/resources/SPD/].



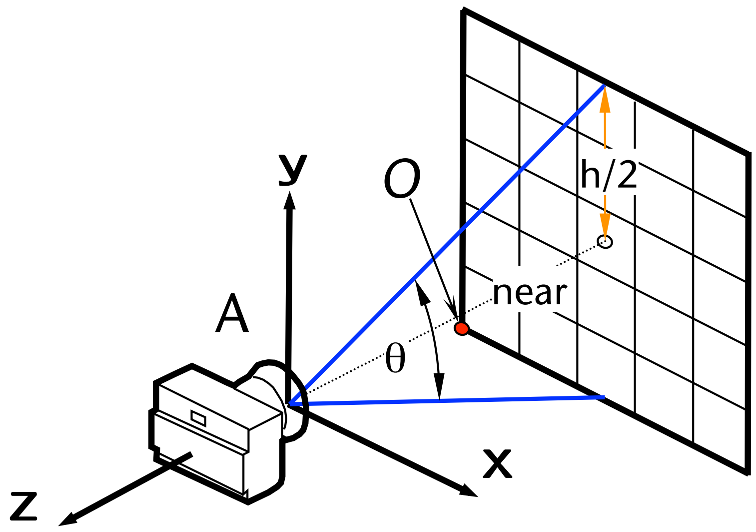


Fake or Real?



Fake or Foto? – The Challenge
file:///Users/zach/Documents/Lehre/CG1/demos/raytracing_fake_or_real/challenge.html#
Google
Intrade Bahn Apple Discussions TUAW WOT fine-art Chaucer's
Fake or Foto? – The Challenge
HTML Open Link in New Window
FAKE or FOTO
Autodesk
The Challenge
Take a look at the ten images below. Some of them are photographs of real objects or scenes, others are created by computer graphics (CG) artists. Test your ability to tell which among the array of images are real, and which are CG. If you want a closer look, click the image to see a larger view of the picture. Once you've decided what's what, click either CG or REAL to begin the tally of your score. Work through each of the ten images. When you've finished, you'll be prompted to get your score.
CG Real
SHOW ME HOW MANY I HAVE
CORRECT!
Display a menu

The Camera (Ideal Pin-Hole Camera)



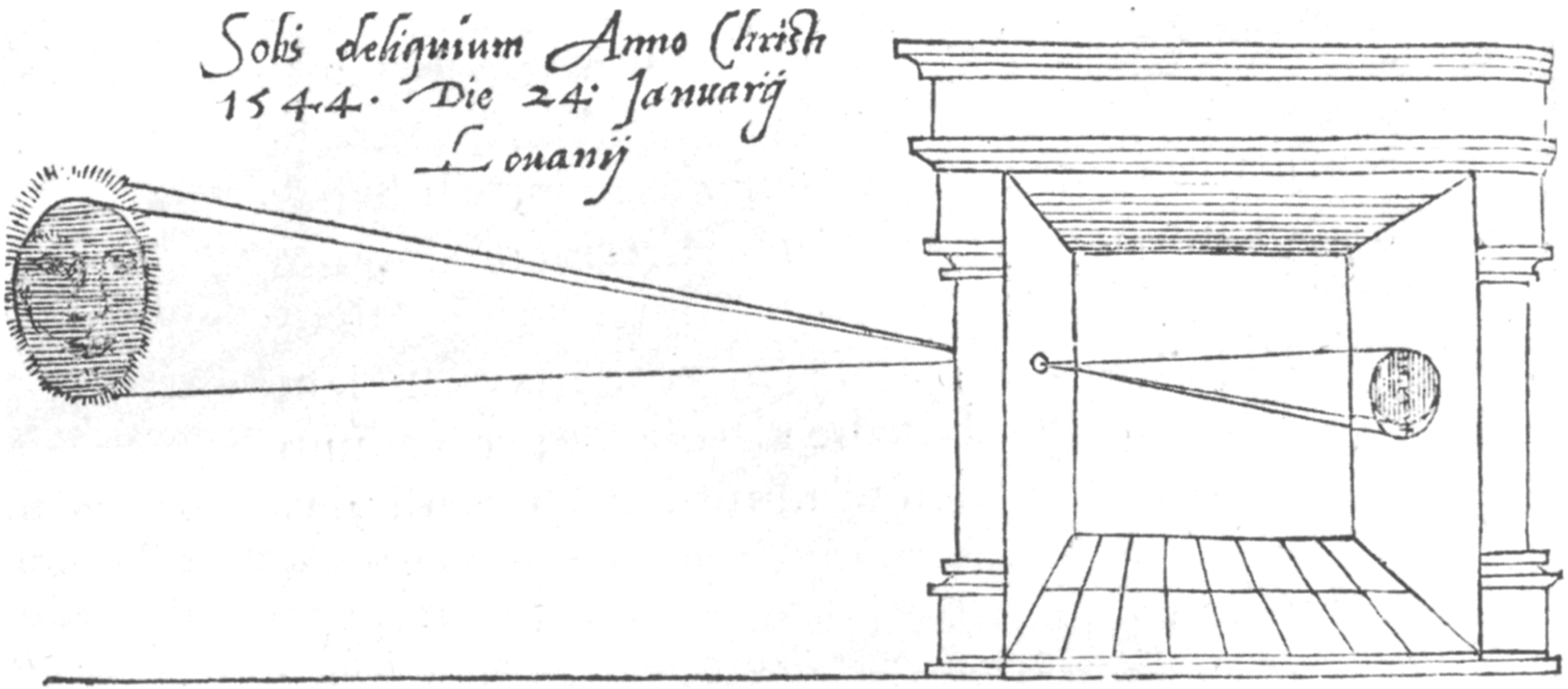
$$\frac{h}{2} = \text{near} \cdot \tan \frac{\theta}{2}$$

$$O = A - \text{near} \cdot \mathbf{z} - \frac{w}{2}\mathbf{x} - \frac{h}{2}\mathbf{y}$$

The main loop of ray-tracers

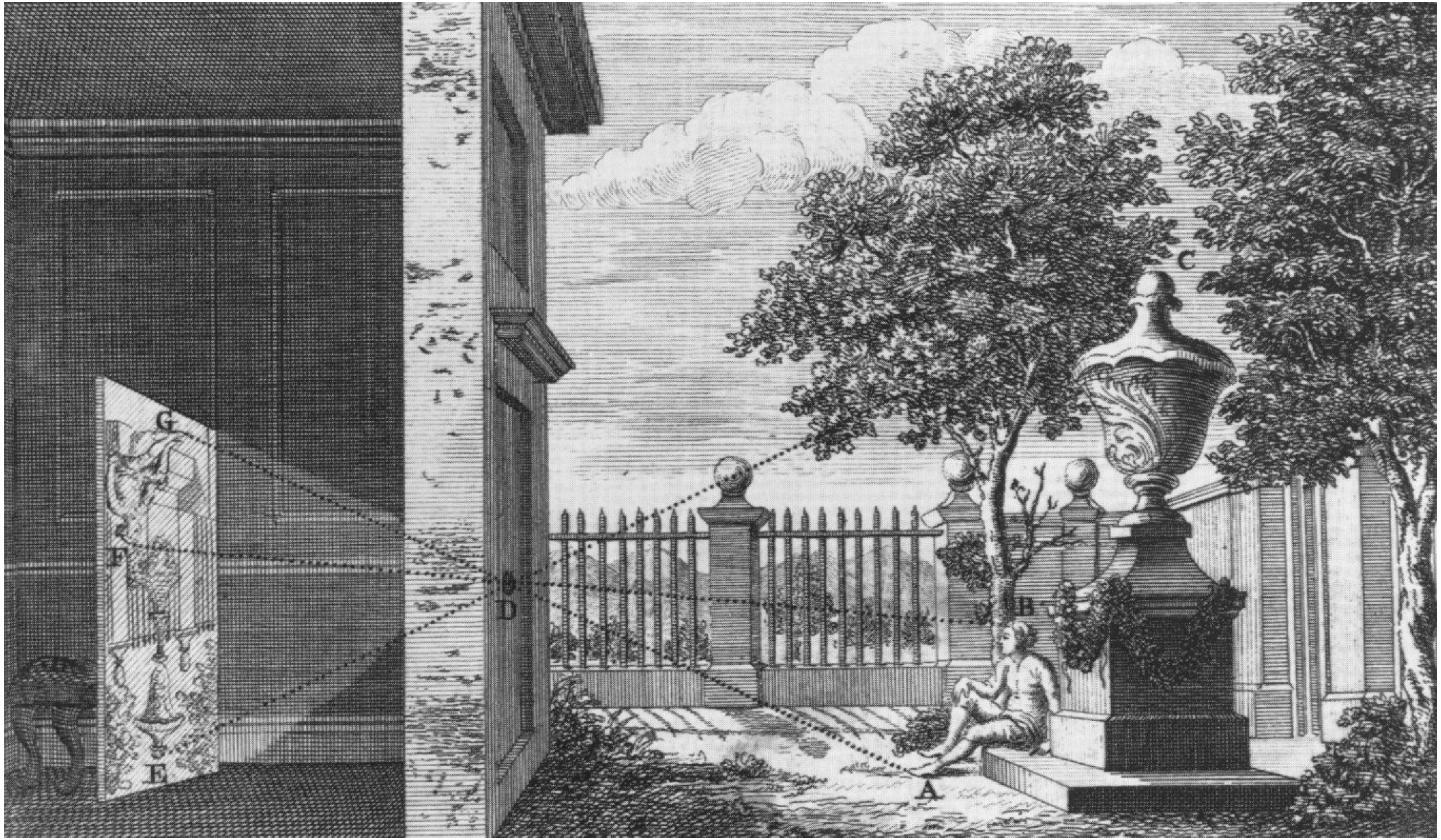
```
for ( i = 0; i < height; i ++ )
  for ( j = 0; j < width; j ++ )
    ray.from = A
    t = (i/height - 0.5) * h
    s = (j/width - 0.5)  * w
    ray.at = O + s·x + t·y
    trace( 0, ray, & color );
    putPixel( x, y, color );
```

Probably the Oldest Depiction of a Pinhole Camera



R. Gemma Frisius, 1545

The Camera Obscura



Digression: Johannes Vermeer

Other Strange Cameras

- With ray-tracing, it is easy to implement non-standard projections
- For instance: fish-eye lenses, projections on a hemi-sphere (= the dome in Omnimax theaters), panoramas

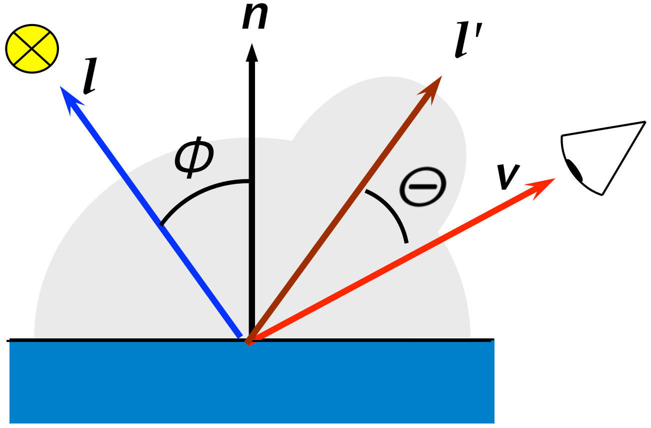
The Lighting Model

- We will use Phong (for sake of simplicity)
- The light emanating from a point on a surface:

$$L_{\text{total}} = L_{\text{Phong}} + \ldots \text{ more terms (later)}$$

$$L_{\text{Phong}} = \sum_{j=1}^{n} \left(k_d \cos \Phi_j + k_s \cos^p \Theta_j \right) \cdot I_j$$

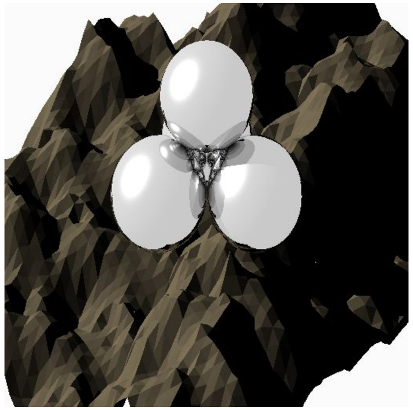
k_d = reflection coefficient for diffuse reflection
k_s = reflection coefficient for specular reflection
I_j = light coming in from j-th light source
n = number of light sources



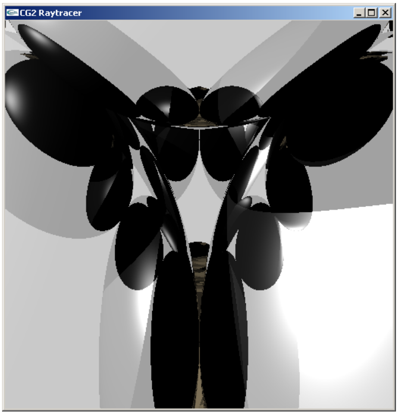
- Of course, we add a light source only, if it is visible!

Stopping Criteria for the Recursion

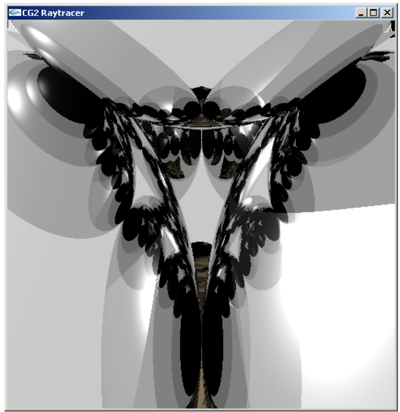
- Each recursive algorithm needs a criterion for stopping:
  - If the maximum recursion depth is reached (fail-safe criterion)
  - If the contribution to a pixel's color is too small (decreases with $depth^n$)



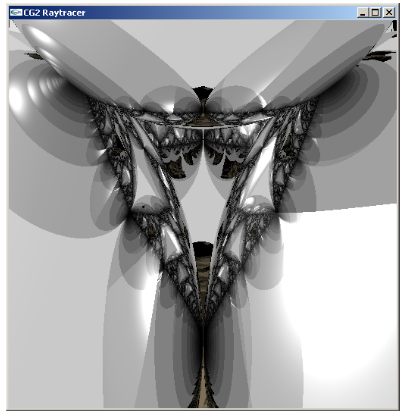
Scene overview



Recursion depth: 3



Recursion depth: 5



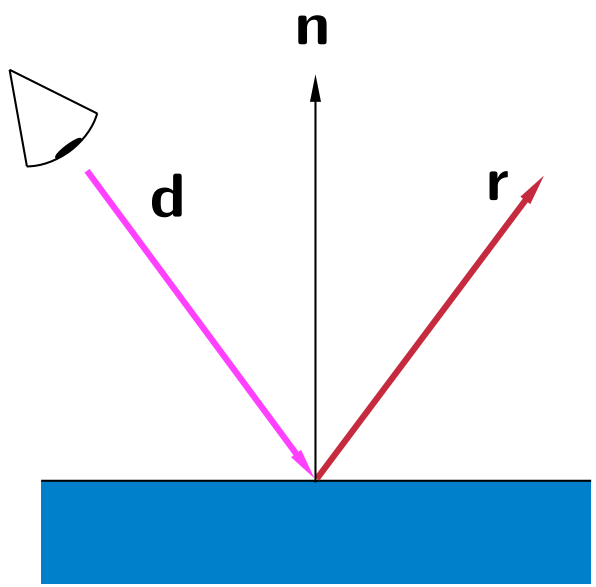
Recursion depth: 100

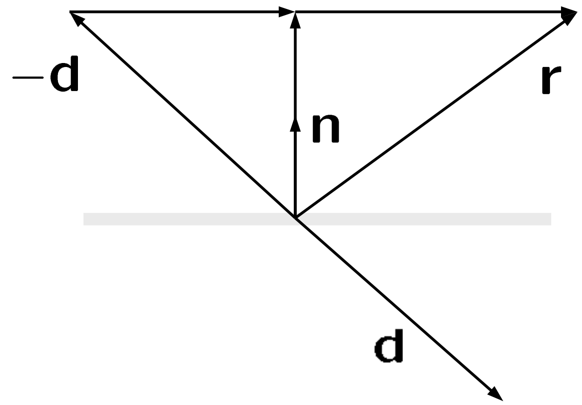
Secondary Rays

- Assumption: we found a hit for the primary ray with the scene
- Then the *reflected ray* is:

$$\begin{aligned}\mathbf{r} &= \big((-\mathbf{d}\cdot\mathbf{n})\cdot\mathbf{n} - (-\mathbf{d})\big)\cdot 2 + (-\mathbf{d})\\ &= \mathbf{d} - 2(\mathbf{d}\cdot\mathbf{n})\cdot\mathbf{n}\end{aligned}$$

with $\|\mathbf{n}\| = 1$



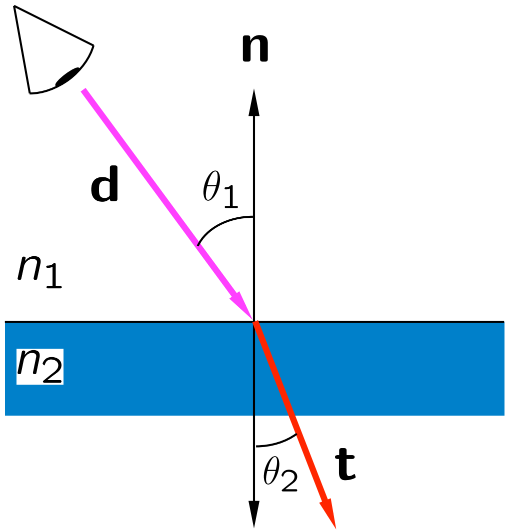


- Additional term in the lighting model:

$$L_{\text{total}} = L_{\text{Phong}} + k_s L_r + \ldots \text{ more terms (later)}$$

L_r = reflected light coming in from direction r
(= perfect reflection)
k_s = material coefficient for specular reflection

The Refracted Ray (a.k.a. Transmitted Ray)



- Law of refraction [Snell, ca.1600] :

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

- Computation of the refracted ray:

$$\mathbf{t} = \frac{n_1}{n_2}(\mathbf{d} + \mathbf{n}\cos\theta_1) - \mathbf{n}\cos\theta_2$$

$$\cos\theta_1 = -\mathbf{dn}$$

$$\cos^2\theta_2 = 1 - \frac{n_1^2}{n_2^2}\left(1 - (\mathbf{dn})^2\right)$$

- Typical refractive indices:

Luft	Wasser	Glas	Diamant
1.0	1.33	1.5 - 1.7	2.4

Derivation of the Equation on the Previous Slide

$$|\mathbf{n}| = |\mathbf{b}| = 1$$

$$\mathbf{t} = \cos\theta_2 \cdot (-\mathbf{n}) + \sin\theta_2 \cdot \mathbf{b}$$

$$\mathbf{d} = \cos\theta_1 \cdot (-\mathbf{n}) + \sin\theta_1 \cdot \mathbf{b}$$

$$\mathbf{b} = \frac{\mathbf{d} + \mathbf{n} \cdot \cos\theta_1}{\sin\theta_1}$$

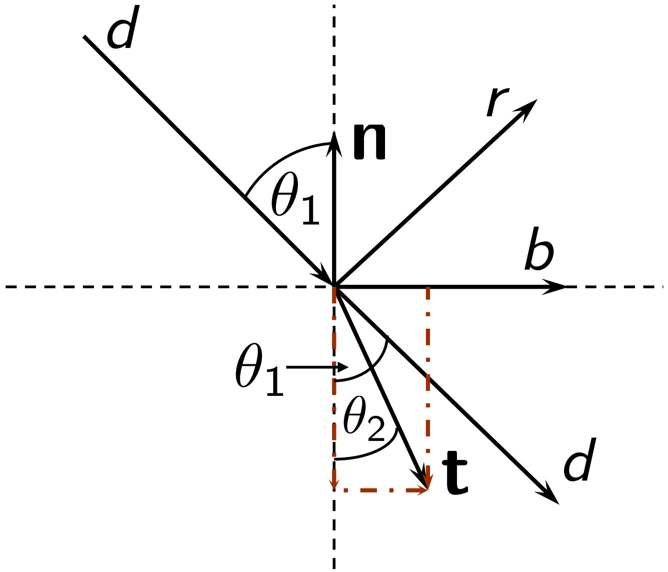
$$\mathbf{t} = -\mathbf{n} \cdot \cos\theta_2 + \frac{\sin\theta_2}{\sin\theta_1}(\mathbf{d} + \mathbf{n} \cdot \cos\theta_1)$$

$\cos\theta_2$ ausrechnen:

$$\sin\theta_2 = \frac{n_1}{n_2}\sin\theta_1$$

$$\sin^2 + \cos^2 = 1$$

$$\cos^2\theta_2 = 1 - (\frac{u_1}{u_2}\sin\theta_1)^2$$



$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2}$$

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

- Total reflection occurs, whenever the following condition holds:

$$\text{if radicand} < 0 \quad \Leftrightarrow \quad \cos^2\theta_1 \leq 1 - \frac{n_2^2}{n_1^2}$$

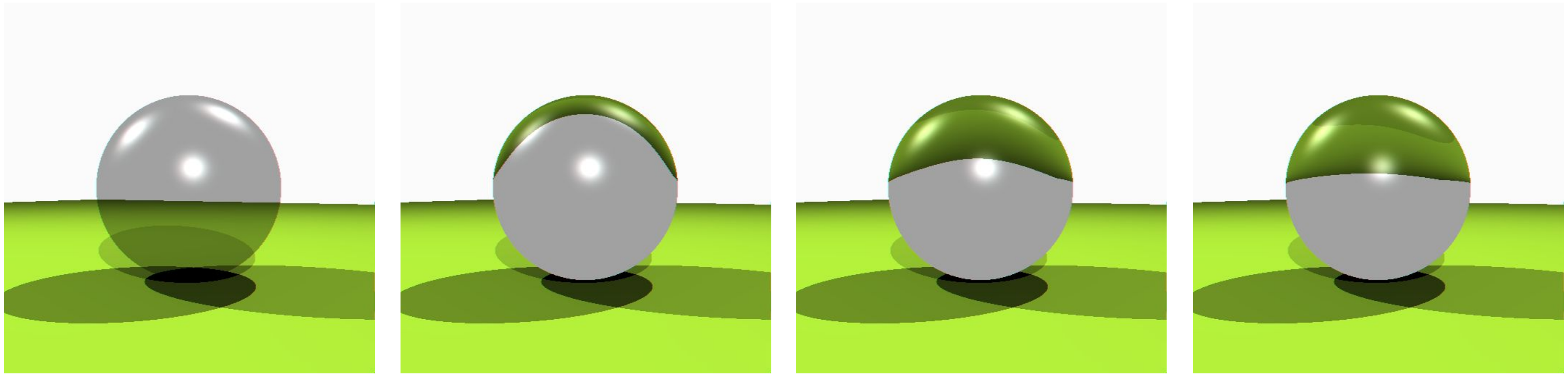
- The complete lighting model (for now):

$$L_{\text{total}} = L_{\text{Phong}} + k_s L_r + k_t L_t$$

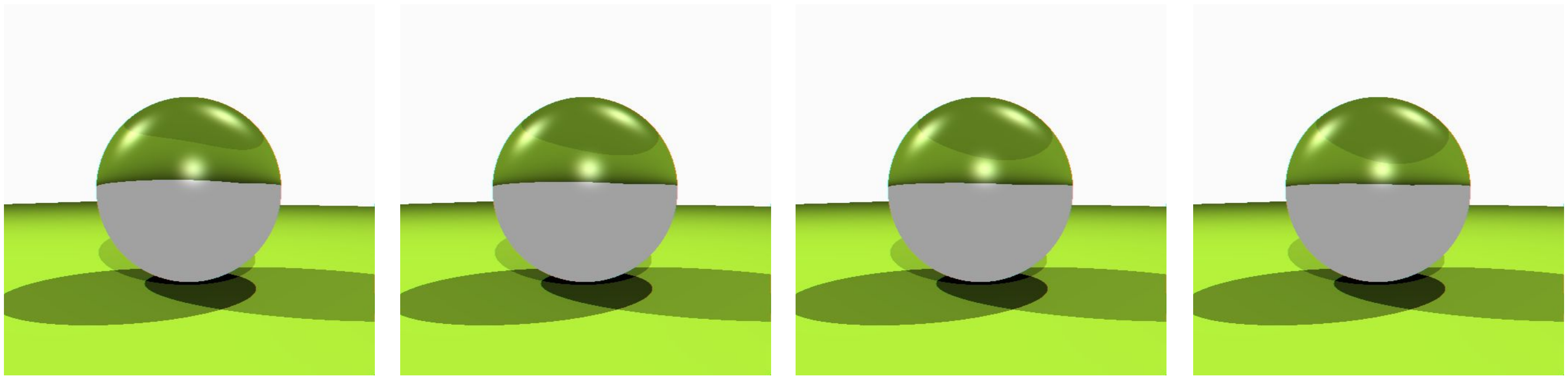
L_t = transmitted light coming in from direction t

k_t = material coefficient for refraction

The Effect of the Refractive Index



n=1.0 n=1.1 n=1.2 n=1.3

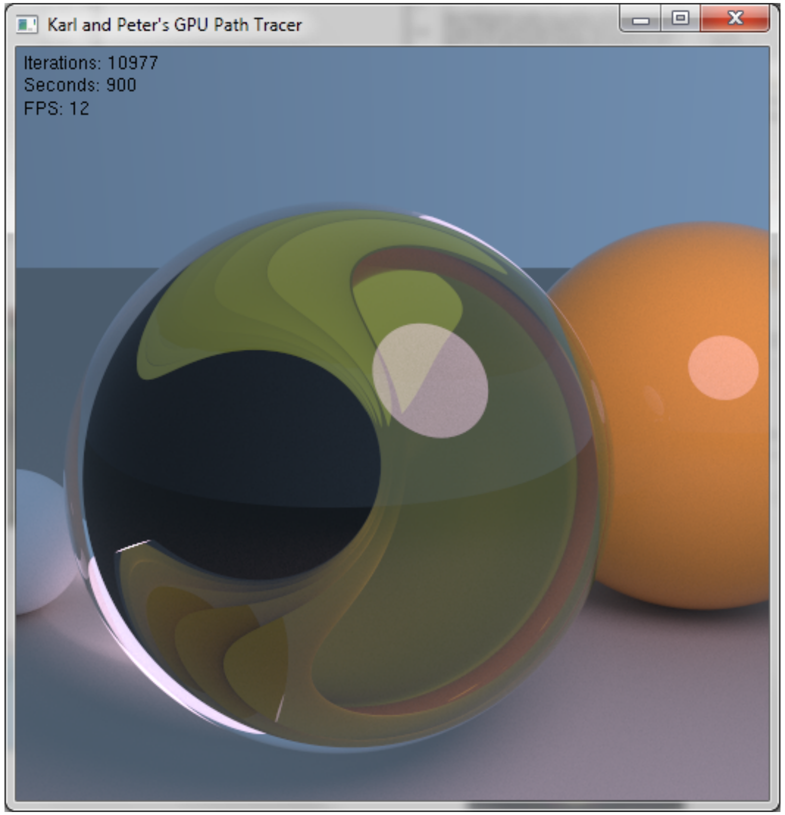


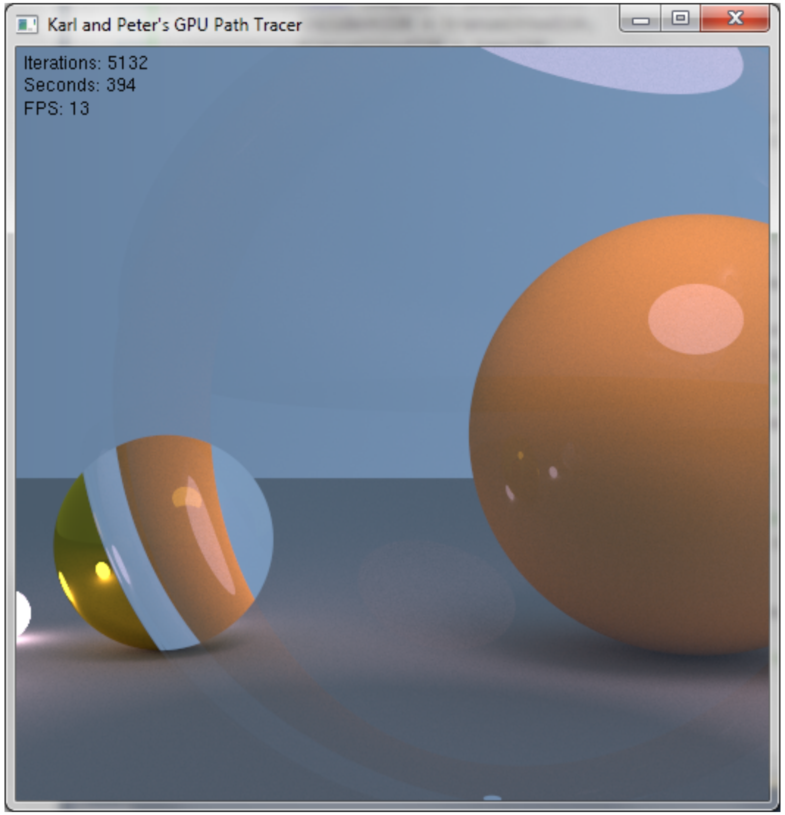
n=1.4 n=1.5 n=1.6 n=1.7

Which One is the "Correct" Normal?

- Food for thought: do the computations of the reflected and transmitted rays also work, if the normal of the surface is pointing into the "wrong" direction?
  - Which direction is the wrong one anyway?

Glitch Pictures: Incorrect Refraction





Source: yiningkarlli (http://igad2.nhtv.nl/ompf2)

Which Effect Can We Not Quite Simulate Correctly (Yet)?

The Fresnel Terms

- When moving from one medium to another, a specific part of the light is reflected, the rest is always refracted
- The reflection coefficient ρ depends on the refractive indices of the involved materials, and on the angle of incidence:

$$\rho_{\|} = \frac{n_2 \cos\theta_1 - n_1 \cos\theta_2}{n_2 \cos\theta_1 + n_1 \cos\theta_2}$$

$$\rho_{\perp} = \frac{n_1 \cos\theta_1 - n_2 \cos\theta_2}{n_2 \cos\theta_1 + n_1 \cos\theta_2}$$

$$\rho = \frac{1}{2} \cdot \left(\rho_{\|}^2 + \rho_{\perp}^2 \right)$$

- 1-ρ = the amount of the transmitted light

- Example:
  - Air (n = 1.0) to glass (n = 1.5), angle of incidence = perpendicular:

$$\rho_{\parallel} = \frac{1.5-1}{1.5+1} = \frac{1}{5} \quad \rho_{\perp} = \frac{1-1.5}{1.5+1} = \frac{1}{5} \quad \rho = \frac{1}{2} \cdot \frac{2}{25} = 4\%$$

  - I.e., when moving perpendicularly from air to glass, 4% of the light is reflected, the rest is refracted
- Approximation of the Fresnel terms [Schlick 1994]:

$$\rho(\theta) \approx \rho_0 + (1-\rho_0)\,(1-\cos\theta)^5$$

$$\rho_0 = \left(\frac{n_2-1}{n_2+1}\right)^2$$

where ρ_0 = Fresnel term for perpendicular angle of incidence, and
θ = angle between ray and normal in the thinner medium
(i.e., the larger angle)

Example for Refraction with Fresnel Terms

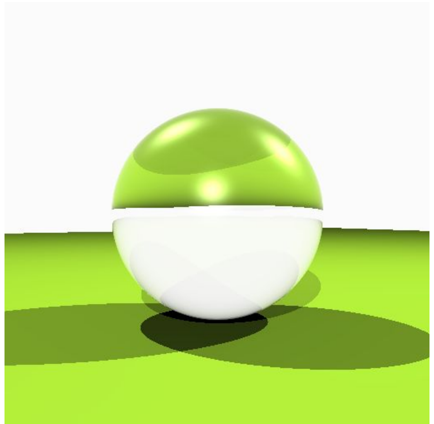
n=1.0

n=1.1

n=1.2

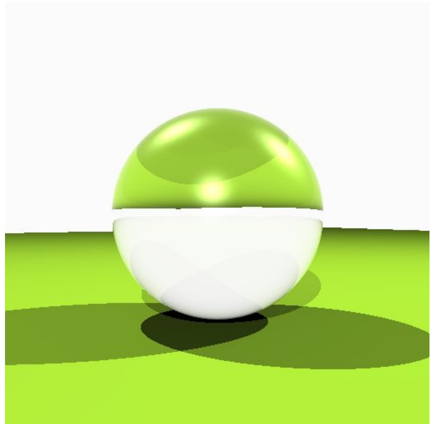
n=1.3

n=1.4



n=1.5

n=1.6



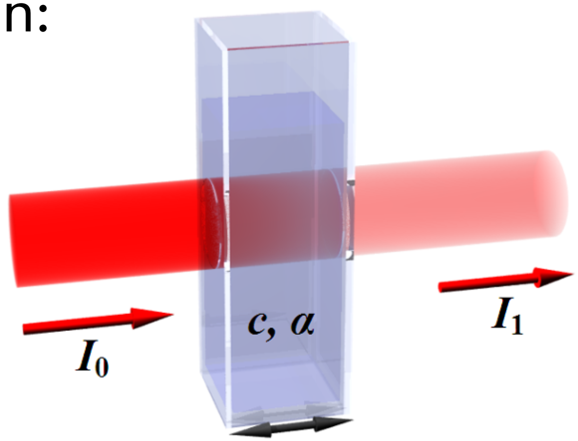
n=1.7

Attenuation (Dämpfung) in Participating Media

- When light travels through a medium, its intensity is attenuated, depending on the length of its path through the medium
- The Lambert-Beer Law governs this attenuation:

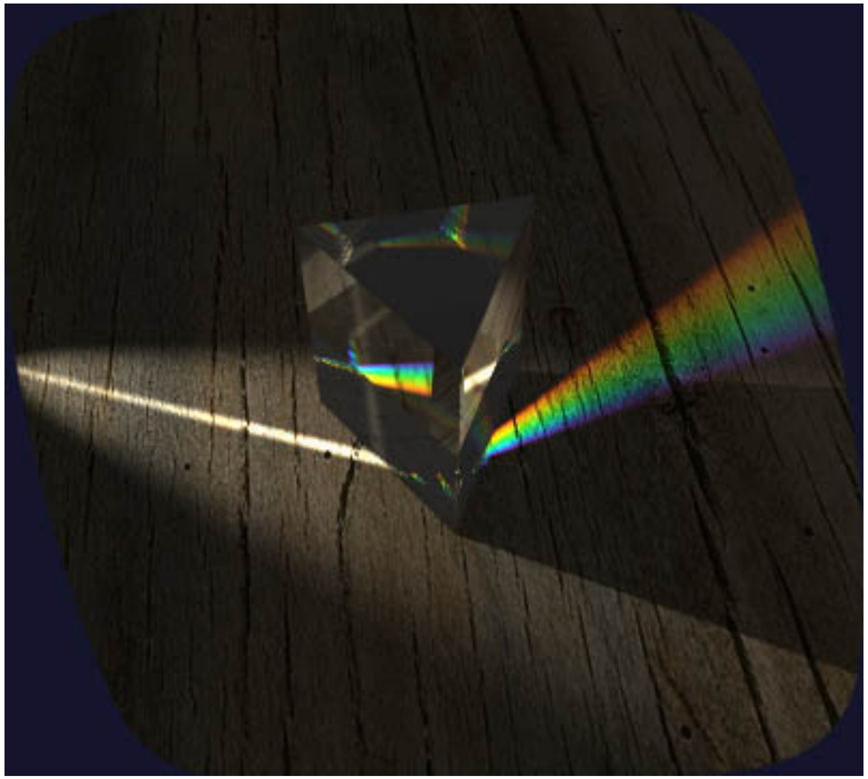
$$I(s) = I_0 e^{-\alpha s}$$

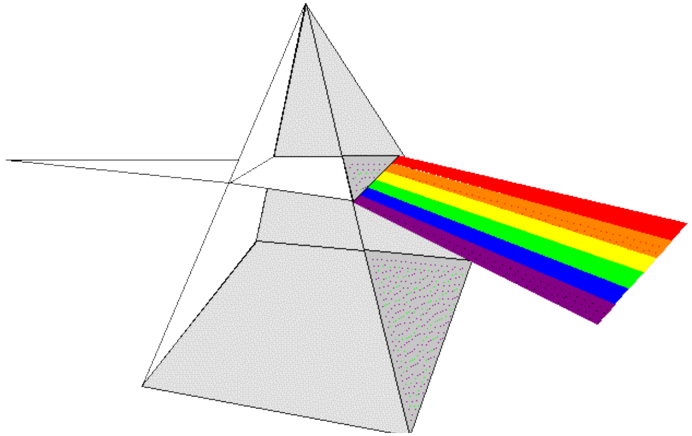
where α = some material constant, and
s = the distance travelled in the medium

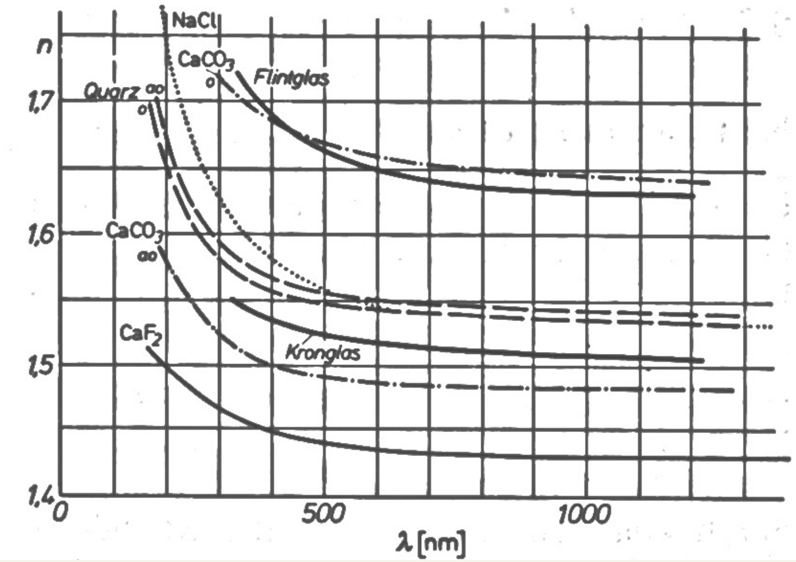


Dispersion

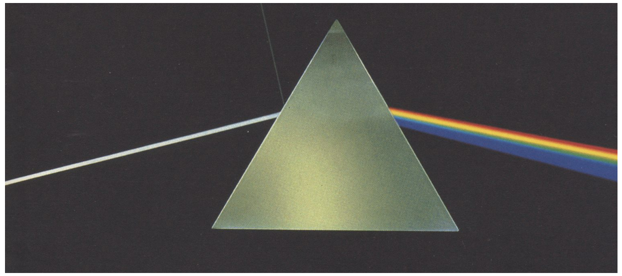
- In reality, the refractive index depends on the wavelength!
- This effect cannot be modelled any more with simple "RGB light"; this requires a spectral ray-tracer







Giovanni Battista Pittoni, 1725,
"An Allegorical Monument to Sir Isaac Newton"



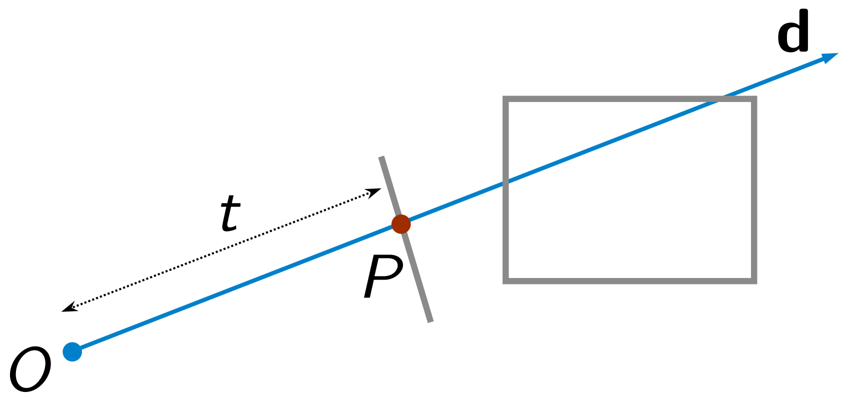
Pink Floyd, *The Dark Side of the Moon*

Example with Fresnel Terms and Dispersion

Intersection Computations Ray-Primitive

- Amount to the major part of the computation time
- Given: a set of objects (e.g., polygons, spheres, ...) and a ray

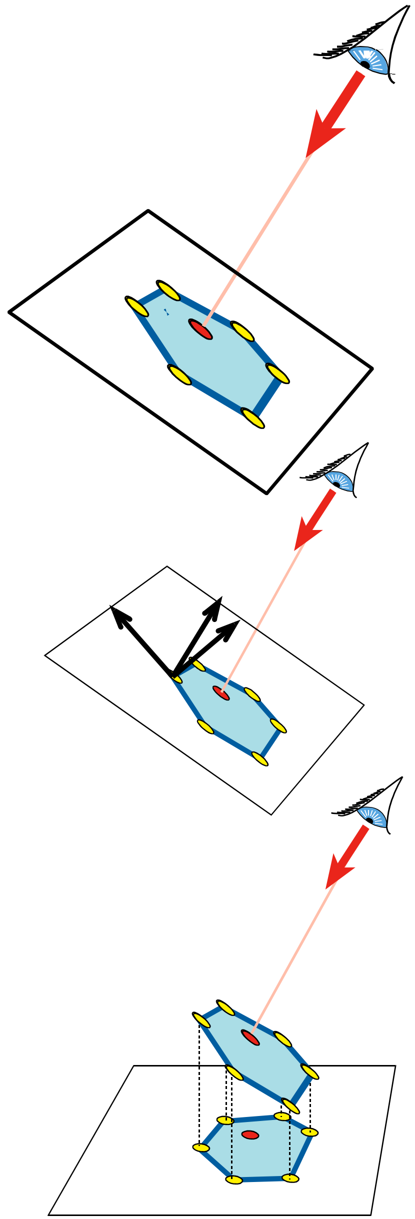
$$P(t) = O + t \cdot \mathbf{d}$$

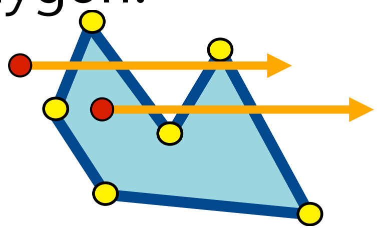


- Wanted: the line parameter t of the *first* intersection point $P = P(t)$ with the scene

Intersection of Ray with Polygon

- Intersection of the ray (parametric) with the supporting plane of the polygon (implicit) → point
- Test whether this point is in the polygon:
  - Takes place completely in the plane of the polygon
  - 3D point is in 3D polygon ⇔ 2D point is in 2D poly
- Project point & polygon:
  - Along the normal: too expensive
  - Orthogonal onto coord plane: simply omit one of the 3 coords of all points involved
- Test whether 2D point is in 2D polygon:
  - Count the number of intersection between (another, 2D) ray and the 2D polygon





Interludium: the Complete Ray-Tracing-Routine

```
traceRay( ray ):
  hit = intersect( ray )
  if no hit:
    return no color
  reflected_ray = reflect( ray, hit )
  reflected_color = traceRay( reflected_ray )
  refracted_ray = refract( ray, hit )
  refracted_color = traceRay( refracted_ray )
  for each lightsource[i]:
    shadow_ray = compShadowRay( hit, lightsource[i] )
    if intersect(shadow_ray):
      light_color[i] = 0
  overall_color = shade( hit,
                         reflected_color,
                         refracted_color,
                         light_color )
  return overall_color
```

hit is a data structure (a struct or an instance of a class) that contains all infos about the intersectin between the ray and the scene, e.g., the intersection point, a pointer to the object, normal, …

The **intersect** function can be optimized considerably compared to **traceRay**; in addition, only intersection points **before** the light source are relevant.

Evaluates the lighting model of the hit object.

Typical Classes in the Software Architecture of a Raytracer

- Class for storing lightsources (here, just positional light sources):

```
Vector m_location;       // Position
Vector m_color;          // Farbe
```

- Class for storing the material of surfaces:

```
Vector m_color;          // Farbe der Oberfläche
float m_diffuse;         // Diffuser / Spekularer
float m_specular;        // Reflexionskoeff. [0..1]
float m_phong;           // Phong-Exponent
```

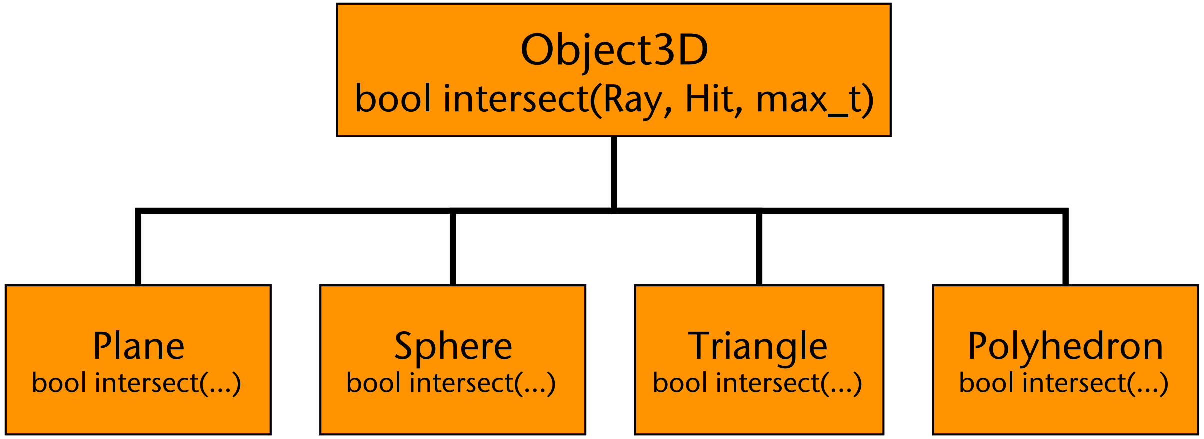
- A class for rays:

```
Vector m_origin;         // Aufpunkt des Strahls
Vector m_direction;      // Strahlrichtung
```

- Class for passing around data about intersections (hit):
  - Important class
  - Records all kinds of information about an intersection point

```
Ray      m_ray;          // Strahl
float    m_t;            // Geradenparameter t
Object*  m_object;       // Geschnittenes Objekt
Vector   m_location;     // Schnittpunkt
Vector   m_normal;       // Normale am Schnittpunkt
```

- Object = abstract base class for all geometry primitives




```
// Schnittpunkt von Strahl mit Objekt
virtual bool closestIntersection( Intersection * hit ) = 0;
virtual bool anyIntersection( const Ray & ray, float max_t,
                              Intersection * hit ) = 0;

// Normale am Schnittpunkt
virtual Vector calcNormal( Intersection * hit ) = 0;

// Material des Objekts
int getMaterialIndex() const;
```

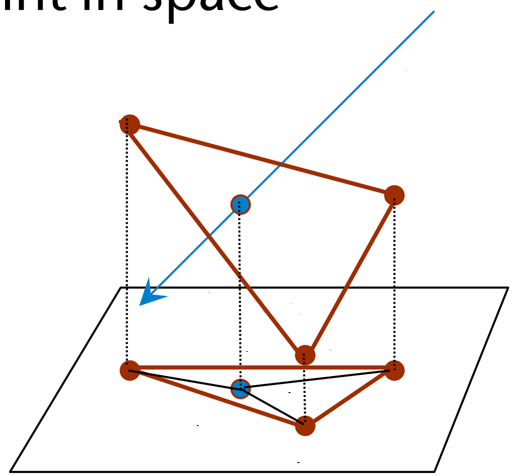
- Camera:
  - Captures all properties of a virtual camera, e.g., from, at, up, angle
  - Generates primary rays for all pixels
- Scene:
  - Stores all data about the scene
    - List of all objects
    - List of all materials
    - List of all light sources
    - Camera
  - Offers methods for calculating intersection between ray and geometry
  - Usually also stores some acceleration data structure

Interscetion of Ray with Triangle

[Badouel 1990]

- Use same method like ray—polygon; or
- Be clever: use barycentric coords + projection
- Intersect ray with plane (implicit form) → t → point in space
- Project point & triangle on coord plane
- Compute baryzentric coords of 2D point
- baryzentric coords of 2D point (α,β,γ) =

  baryzentric coords of orig. 3D point!
- 3D point is in triangle ⇔ $\alpha,\beta,\gamma > 0$, $\alpha+\beta+\gamma = 1$
- Alternative method: see Möller & Haines "Real-time Rendering"
- Ex. faster method, if intersection point not needed [Segura & Feito]



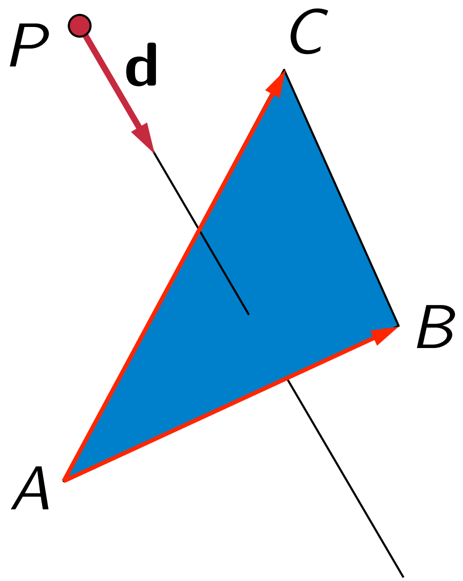
Alternative Intersection Method for Ray—Triangle

- Line equation: $X = P + t \cdot \mathbf{d}$
- Plane equation: $X = A + r \cdot (B - A) + s \cdot (C - A)$
- Equate both:

$$-t \cdot \mathbf{d} + r \cdot (B - A) + s \cdot (C - A) = P - A$$

- Write it in matrix form:

$$\begin{pmatrix} \vdots & \vdots & \vdots \\ -\mathbf{d} & \mathbf{u} & \mathbf{v} \\ \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} t \\ r \\ s \end{pmatrix} = \mathbf{w}$$



where

$$\mathbf{u} = B - A$$

$$\mathbf{v} = C - A$$

$$\mathbf{w} = P - A$$

- Use Cramer's rule:

$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{u}, \mathbf{v})} \cdot \begin{pmatrix} \det(\mathbf{w}, \mathbf{u}, \mathbf{v}) \\ \det(-\mathbf{d}, \mathbf{w}, \mathbf{v}) \\ \det(-\mathbf{d}, \mathbf{u}, \mathbf{w}) \end{pmatrix}$$

$$\det(\mathbf{a}, \mathbf{b}, \mathbf{c}) = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$$

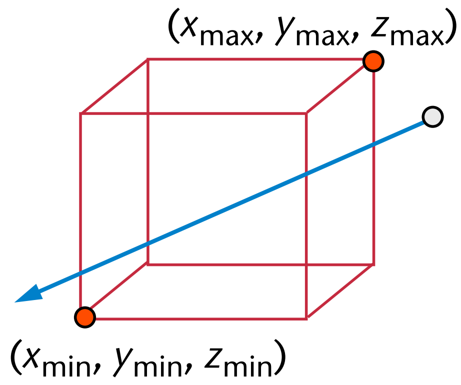
$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{v}) \cdot \mathbf{u}} \cdot \begin{pmatrix} (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{v} \\ (\mathbf{d} \times \mathbf{v}) \cdot \mathbf{w} \\ (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{d} \end{pmatrix}$$

- Cost: 2 cross products + 4 dot products
- Yields both line parameter t and barycentric coords of hit point
- Still need to test whether s,t in [0,1] and s+t <= 1

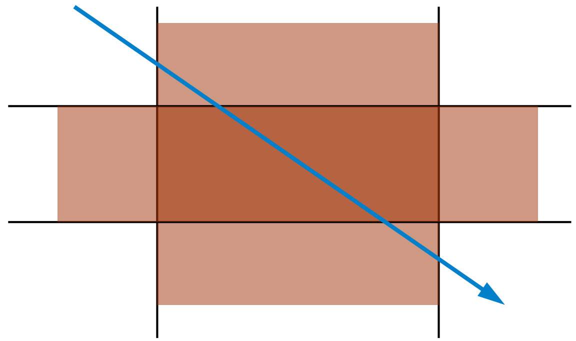
Intersection of Ray and Box

- Box (Quader) is most important bounding volume!
- Here: just axis-aligned boxes (AABB = *axis-aligned bounding box*)
- AABB is usually specified by two extremal points

$$(x_{min}, y_{min}, z_{min}) \text{ and } (x_{max}, y_{max}, z_{max})$$

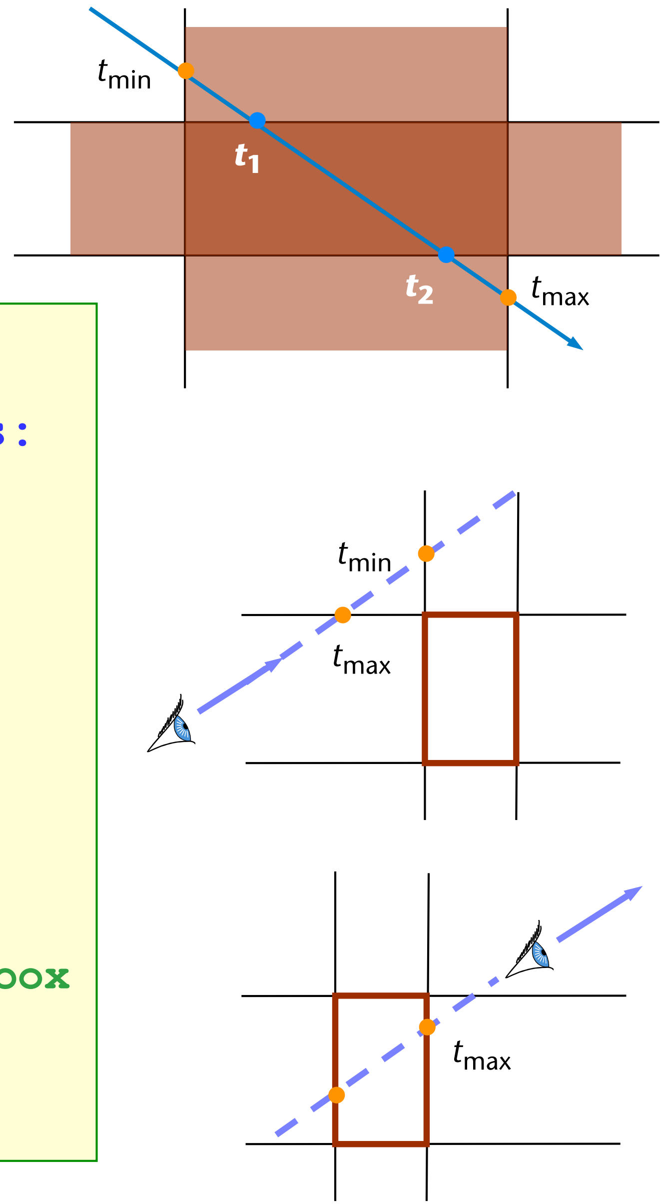


- Idea of the algorithm:
  - A box is the intersection of 3 *slabs* (*slab* = subset of space enclosed between two parallel planes)
  - Each slab cuts away a specific interval of the ray
  - So, successively consider two parallel (= opposite) planes of the box

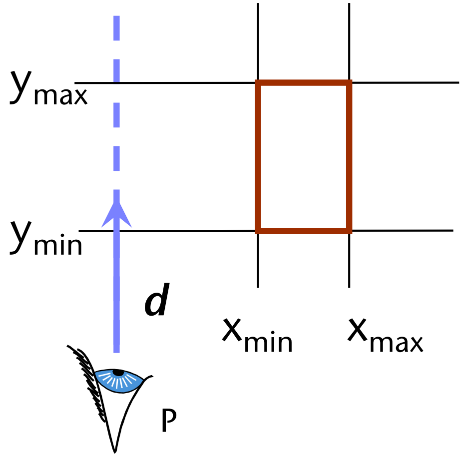


- The algorithm:

```
let t_min = -∞ , t_max = +∞
loop over all (3) pairs of planes:
  intersect ray with both planes
  → t1, t2
  if t2 < t1:
    swap t1, t2
  // now t1 < t2 holds
  t_min ← max(t_min, t1 )
  t_max ← min(t_max, t2 )
// now: [t_min,t_max] = interval inside box
if t_min > t_max → no intersection
if t_max < 0    → no intersection
```



- Optimization: both planes of a slab have the same normal ⟶ can save one dot product
- Remark: the algorithm also works for "tilted" boxes (called *OBBs = oriented bounding boxes*)
- Further optimization: if AABB, exploit fact that normal has exactly one component = 1, other = 0!
- Warning: "shit happens"
  - Here: test for parallel situations!
  - In case of AABB:




```
if |dx| < ε:
  if Px < xmin || Px > xmax:
    ray doesn't intersect box
  else:
    t1, t2 = ymin, ymax      // or vice versa!
```

Intersection Ray—Sphere

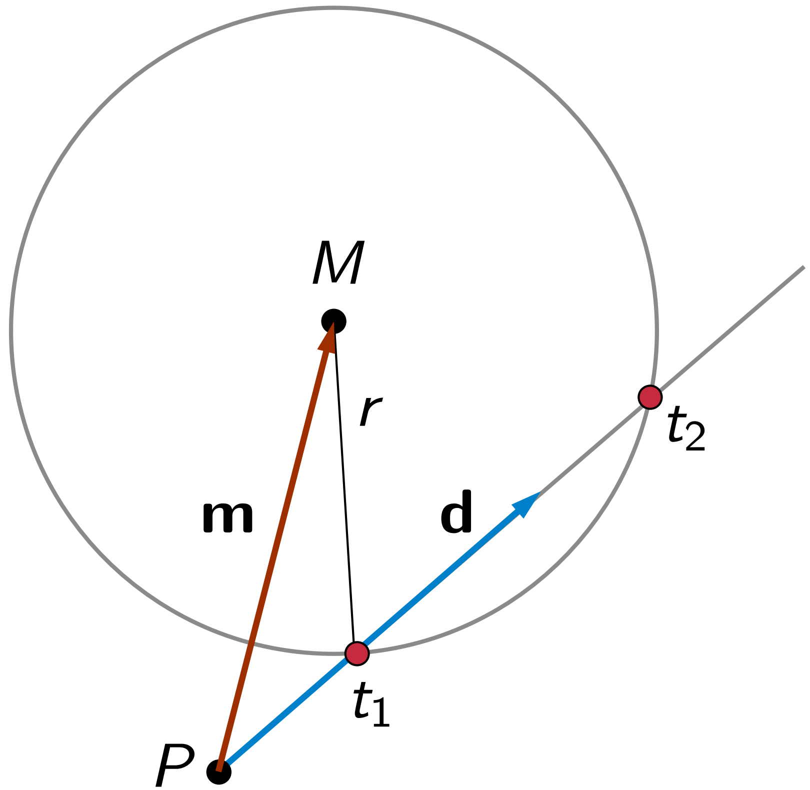
- Assumption: **d** has length 1

- The geometric method:

$$\left| t \cdot \mathbf{d} - \mathbf{m} \right| = r$$

$$(t \cdot \mathbf{d} - \mathbf{m})^2 = r^2$$

$$t^2 - 2t \cdot \mathbf{md} + \mathbf{m}^2 - r^2 = 0$$

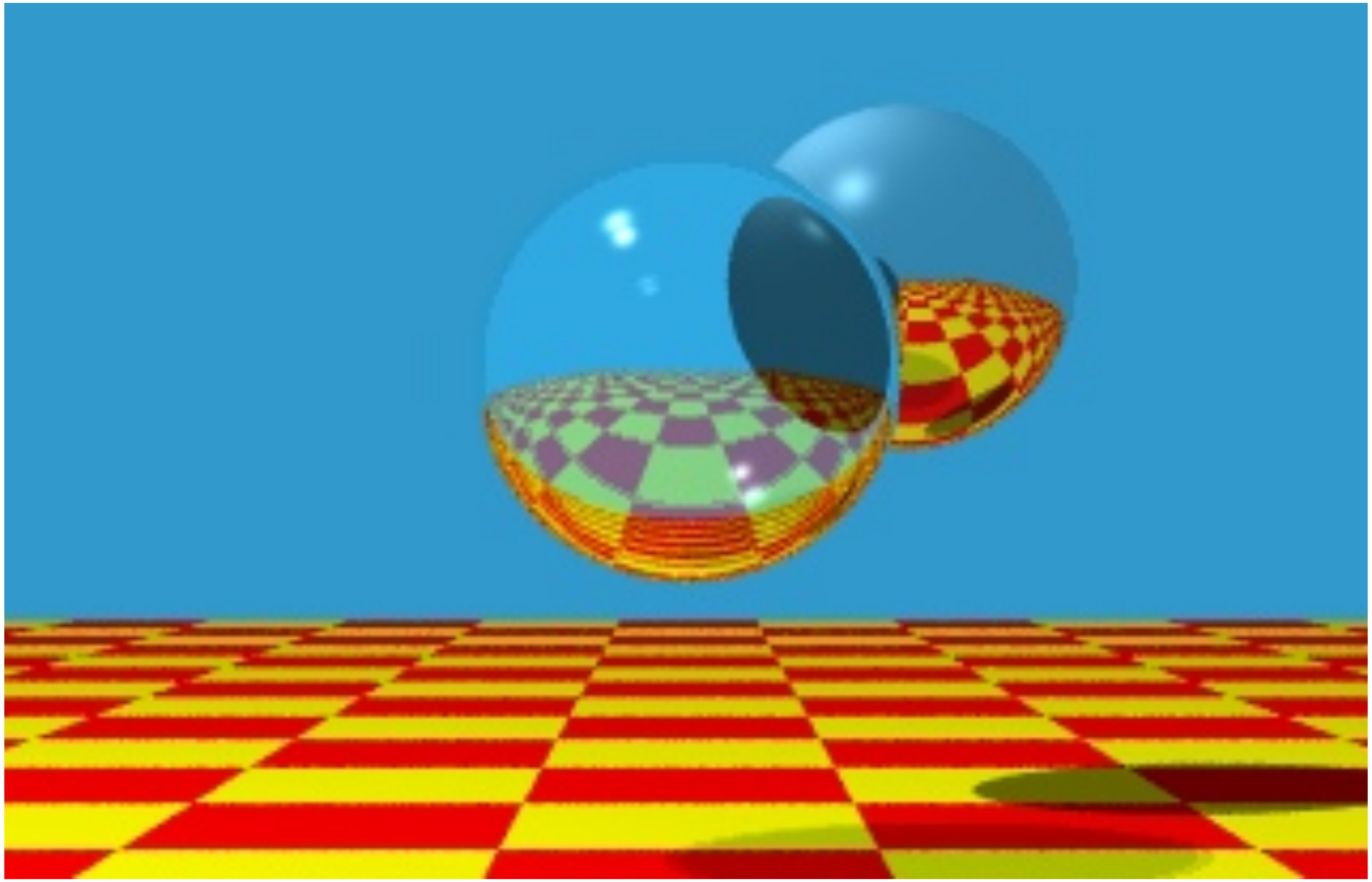


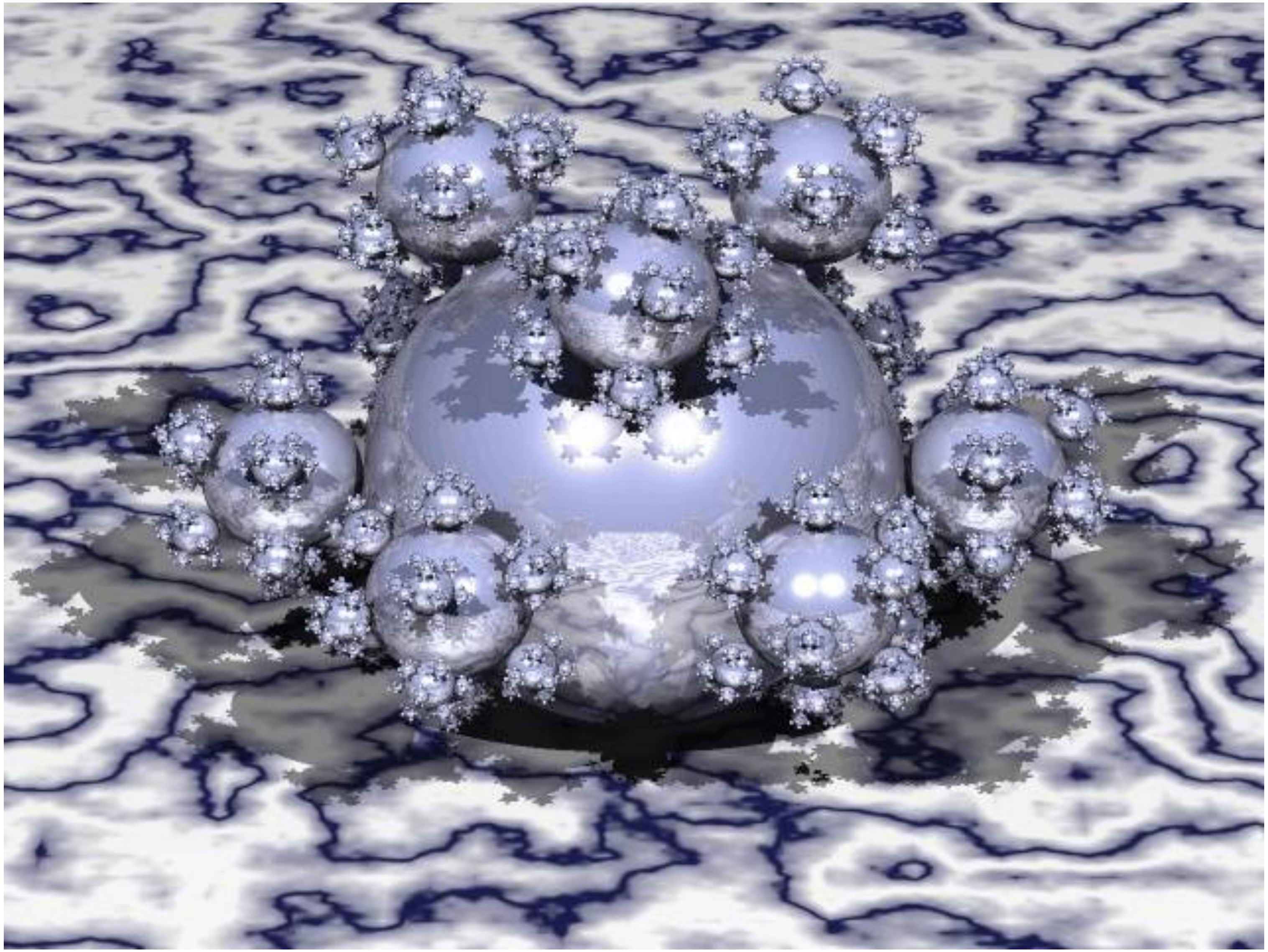
- The algebraic method:
  insert ray equation into implicit sphere equation
- There are many more approaches ...

- The algorithm, with small optimization:

```
calculate m²−r²
calculate b = m·d
if  m²−r² >= 0          // ray origin is outside sphere
    and b <= 0:         // and direction away from sphere
then
     return "no intersection"
let d = b² − m² + r²
if d < 0:
    return "no intersection"
if  m²−r² > ε :
    return t₁ = b − √d  // enter; t₁ is > 0
else:
    return t₂ = b + √d  // leave; t₂ is > 0   (t₁<0)
```

- Ray-sphere intersection is so easy that all ray-tracers have spheres as geometric primitives! ☺





The "sphere flake"