Advanced Computer Graphics
Introduction to Ray-Tracing

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

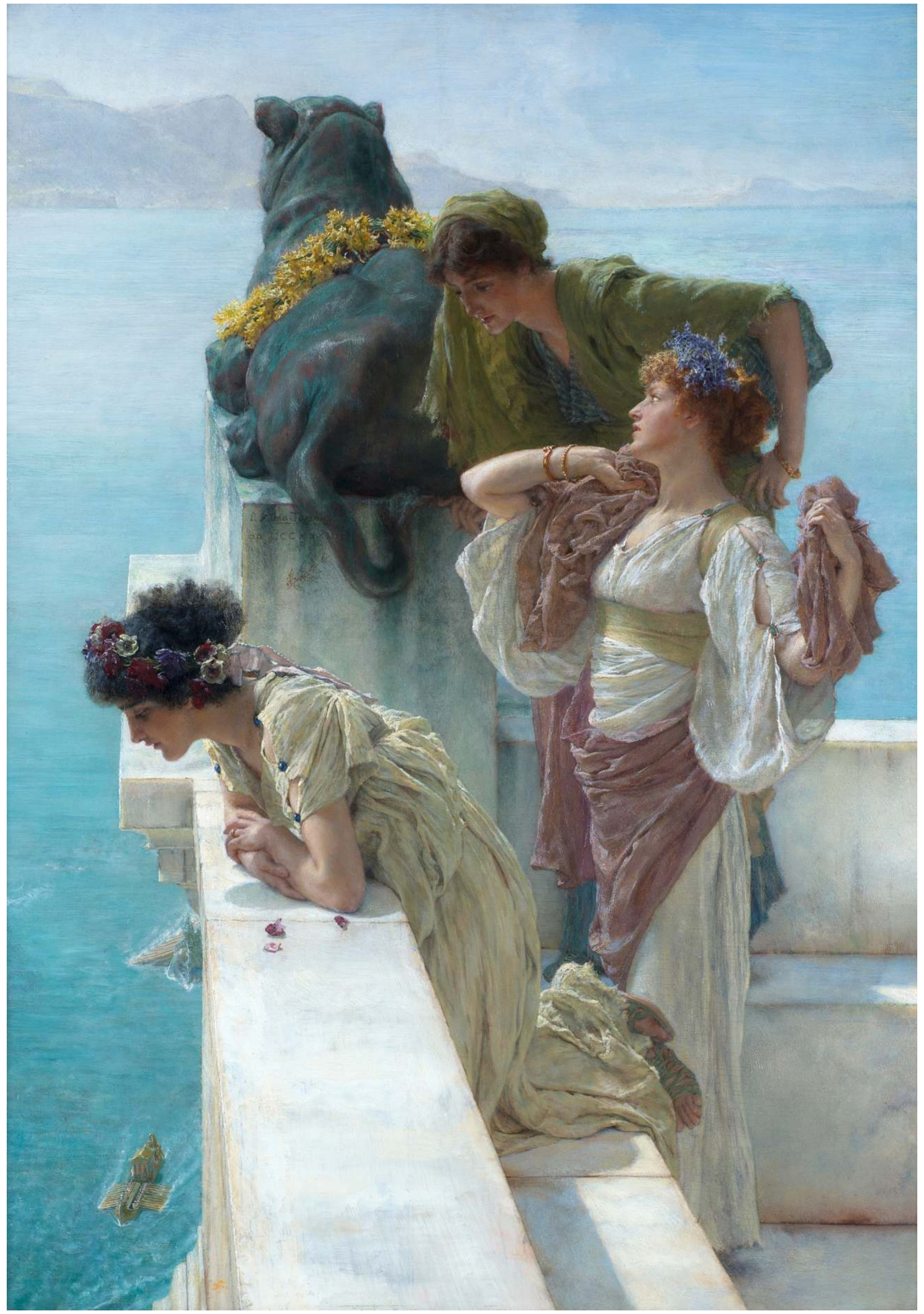
The Ongoing Quest for Realistic Images

"Parrhasios, it is recorded, entered into a competition with Zeuxis, who produced a picture of grapes so successfully represented that birds flew up to the stage buildings [in the theater, which served at that time as a public art gallery]; whereupon Parrhasios himself produced such a realistic picture of a curtain that Zeuxis, proud of the verdict of the birds, requested that the curtain should now be drawn and the picture displayed; and when he realized his mistake, with a modesty that did him honor he yielded up the prize, saying that whereas he had deceived the birds, Parrhasios had deceived him, an artist."

Plinius, der Ältere, 5th century B.C.

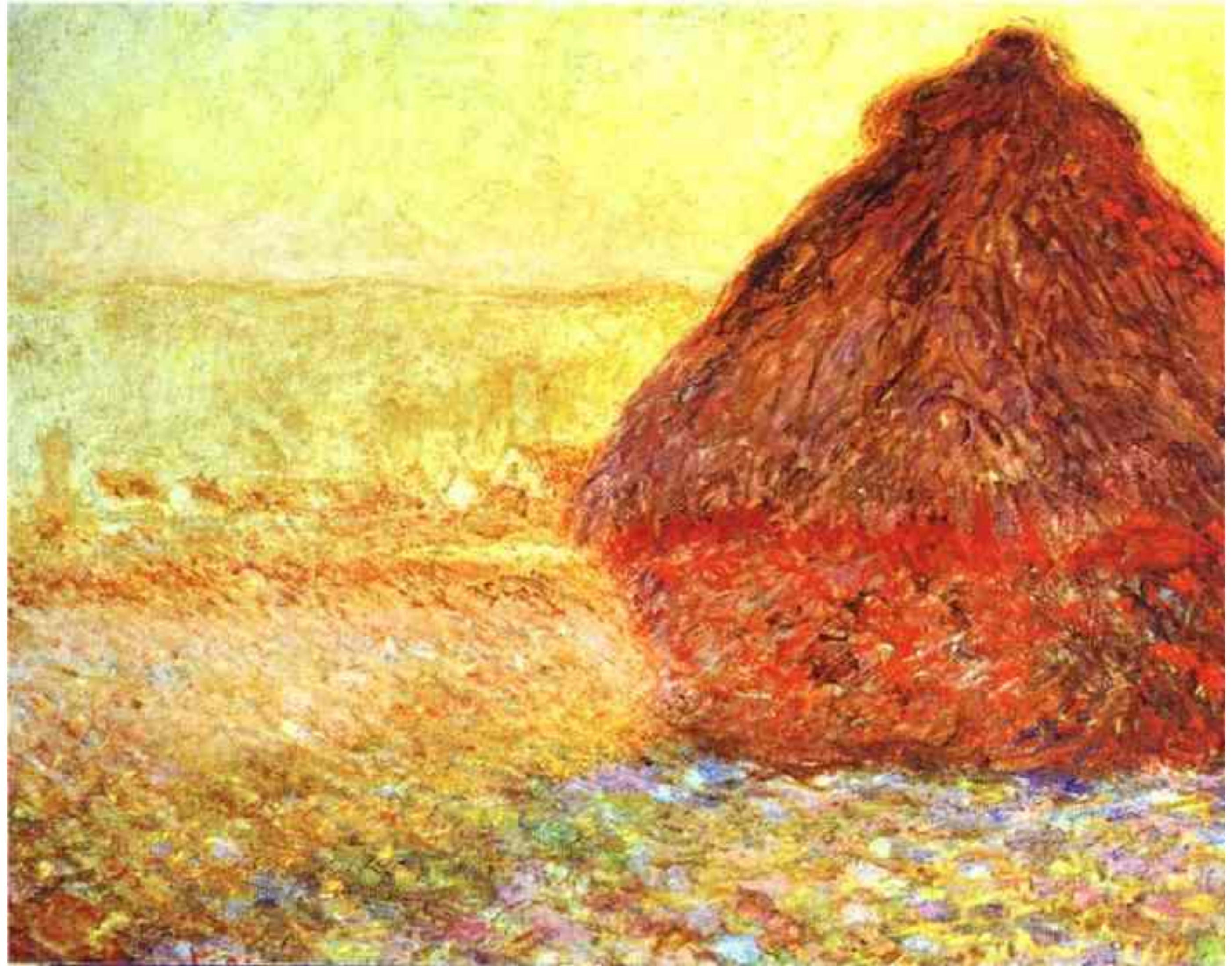
Examples from the History of Fine Art

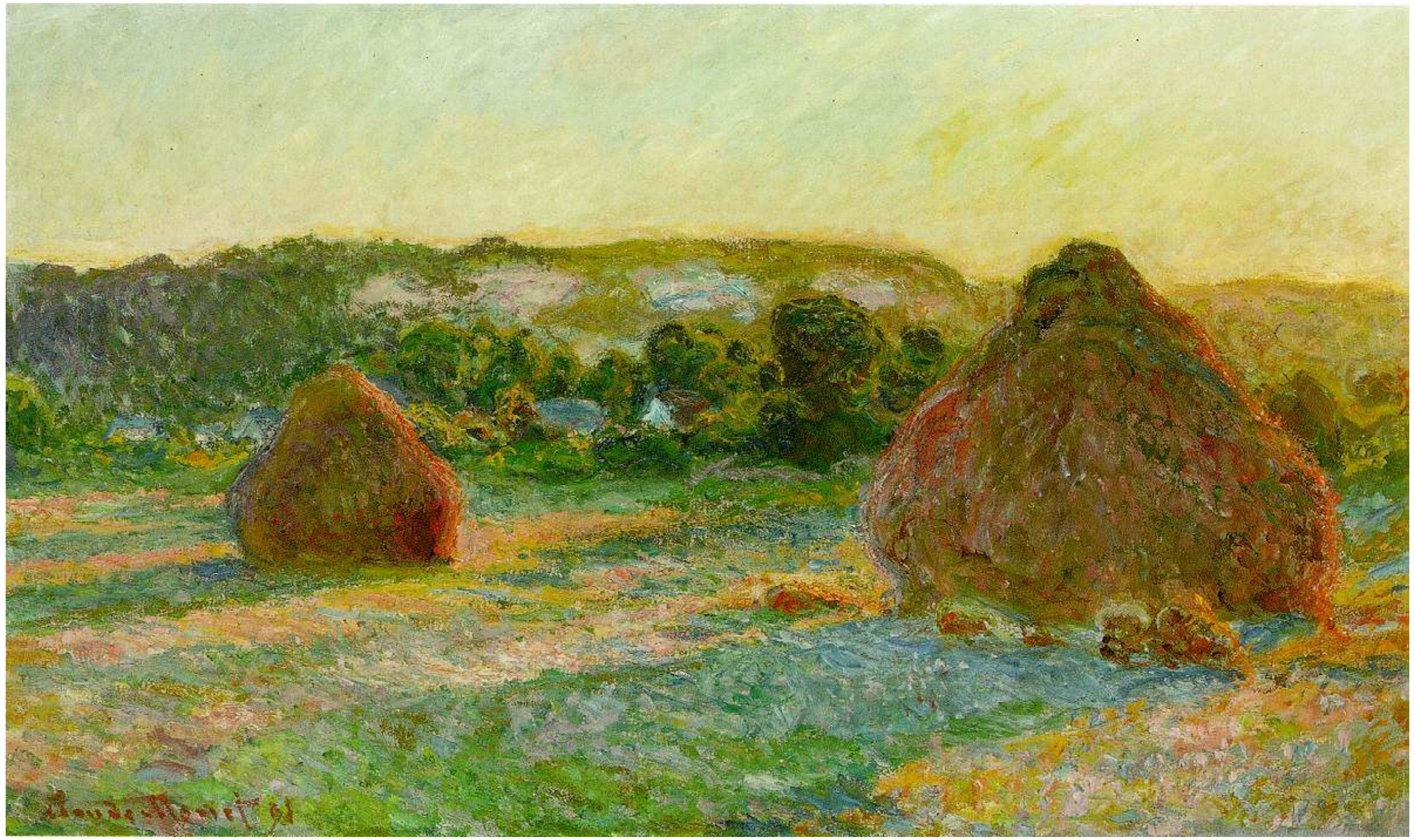
Willem Claesz. Heda, circa 1600-1663



Alma-Tadema: A Solicitation

By Contrast ...





Claude Monet's Haystacks

Effects and Phenomena of Physically Correct Rendering?

https://www.menti.com/alw8aciidqq7

Effects Needed for Physically Correct Rendering

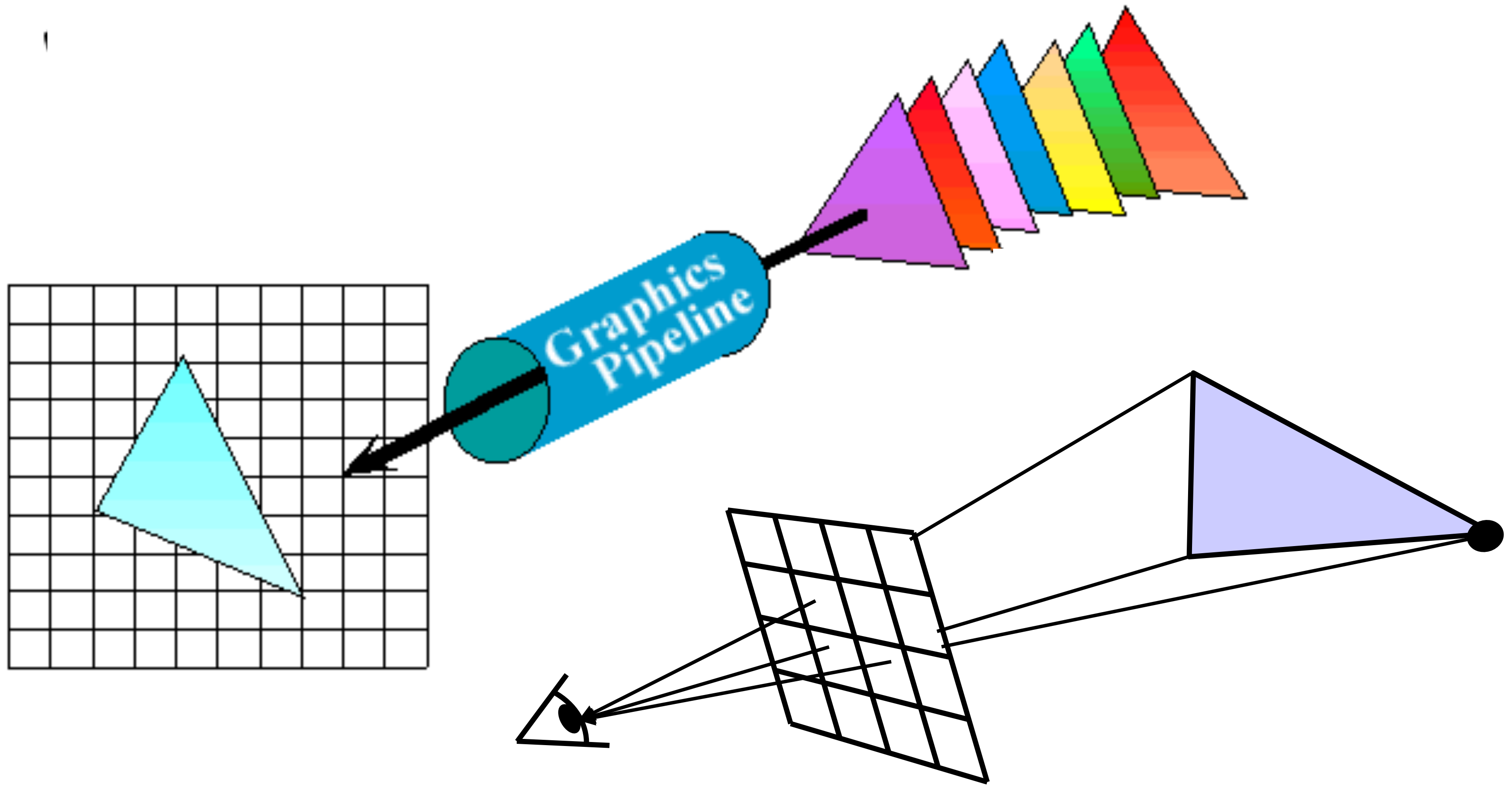
- Remember one of the local lighting models from CG1
- All *local* lighting models fail to render one or more of the following effects:
  - Soft Shadows (Halbschatten)
    - Hard shadows (Schlagschatten) can be done using multi-pass OpenGL rendering (see CG1)
  - Indirect lighting (sometimes also in the form of *"color bleeding"*)
  - Reflection of the scene on glossy surfaces, e.g., mirrors, polished surfaces, etc.
  - Refraction, e.g., through water or glass surfaces
  - Diffraction (Beugung)
  - Participating media, e.g., fog, haze, dust in air
  - ...

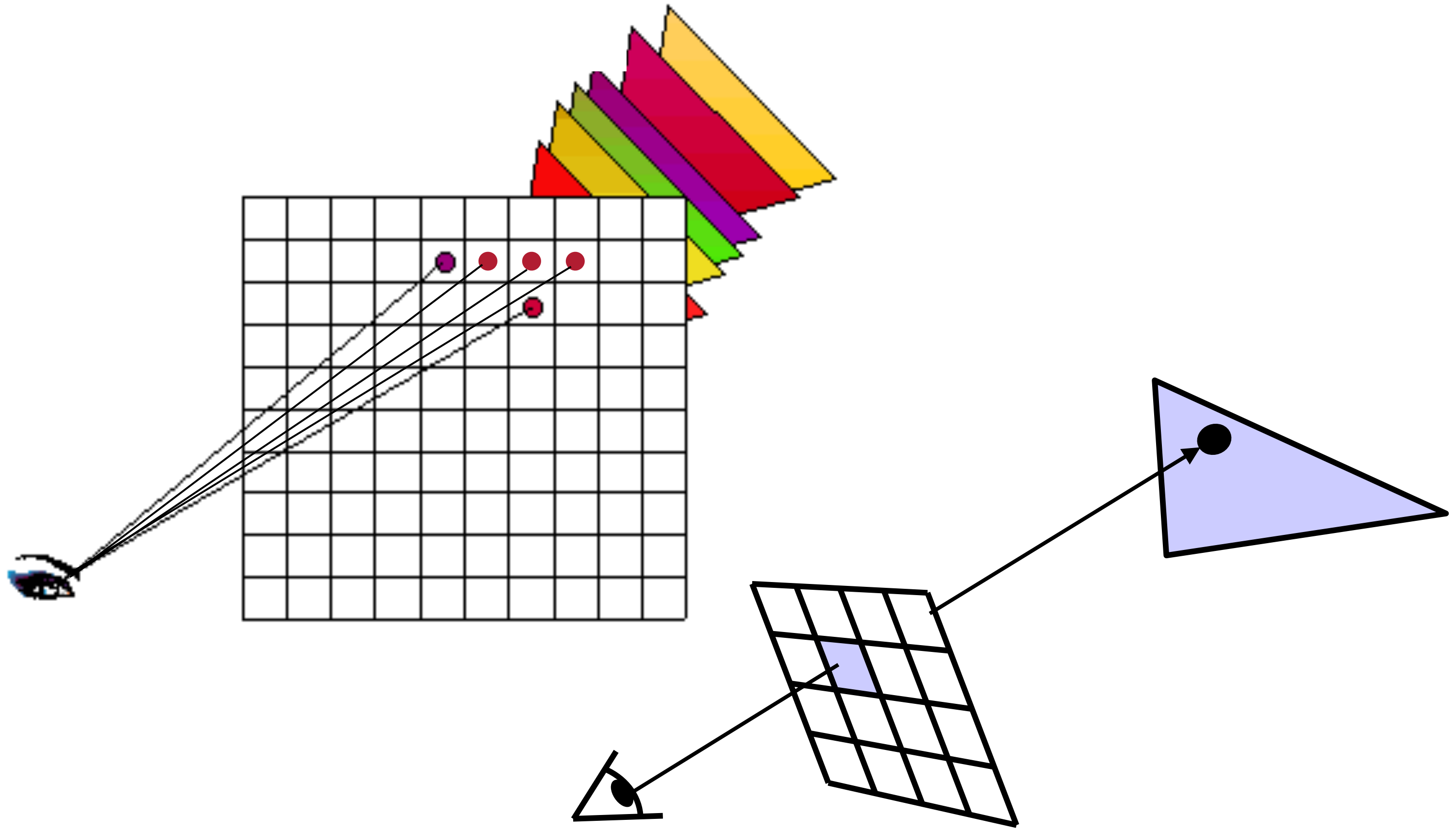
➢ Global Illumination

The Principle of Ray-Tracing vs. Principle of Polygonal Rendering

Polygonal rendering (think "OpenGL") is a "forward-mapping" approach

Ray-casting can be considered an "inverse mapping" approach





```
for each polygon:
  for each pixel:
    ...
```

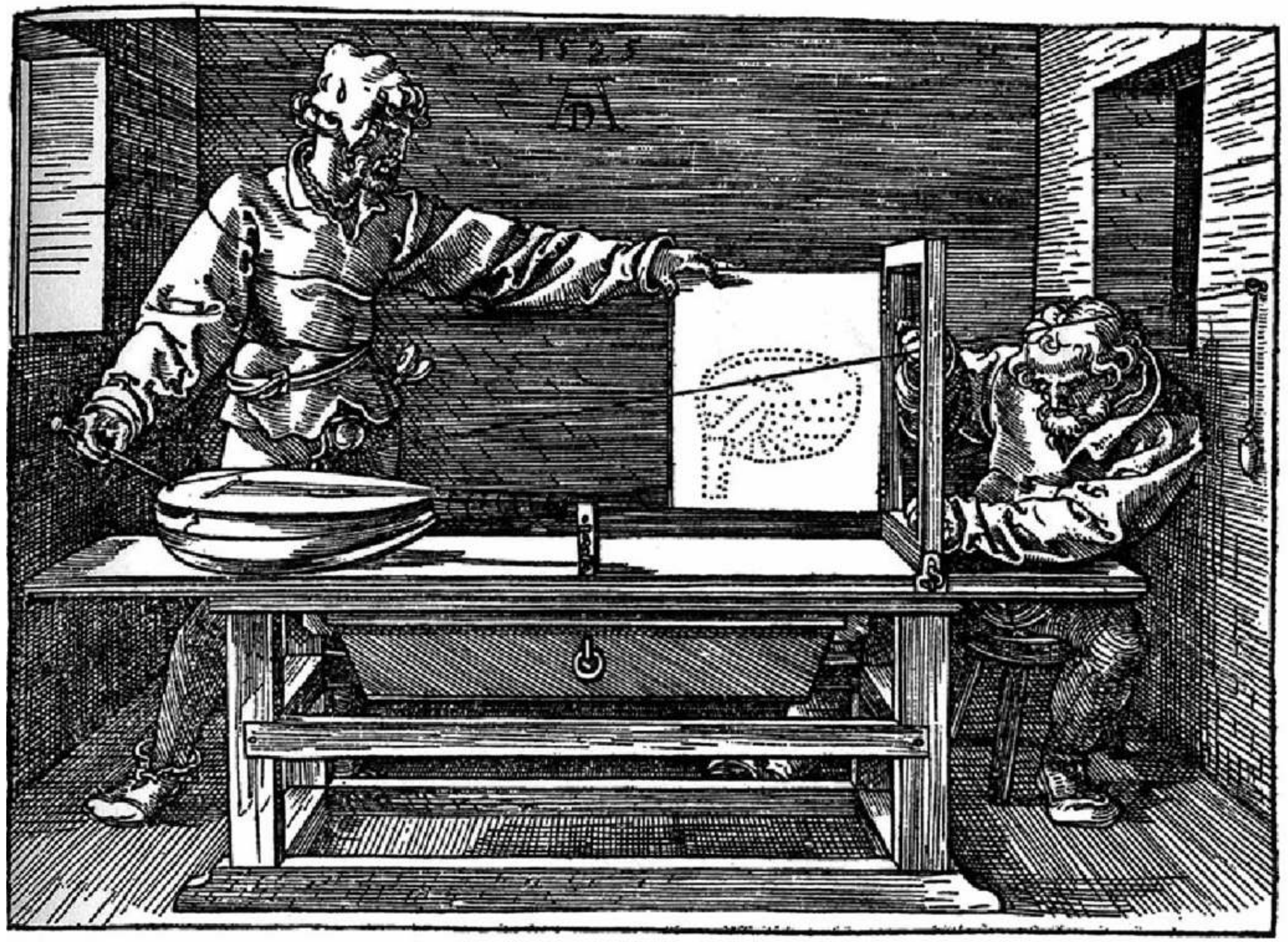
```
for each pixel:
  for each polygon:
    ...
```

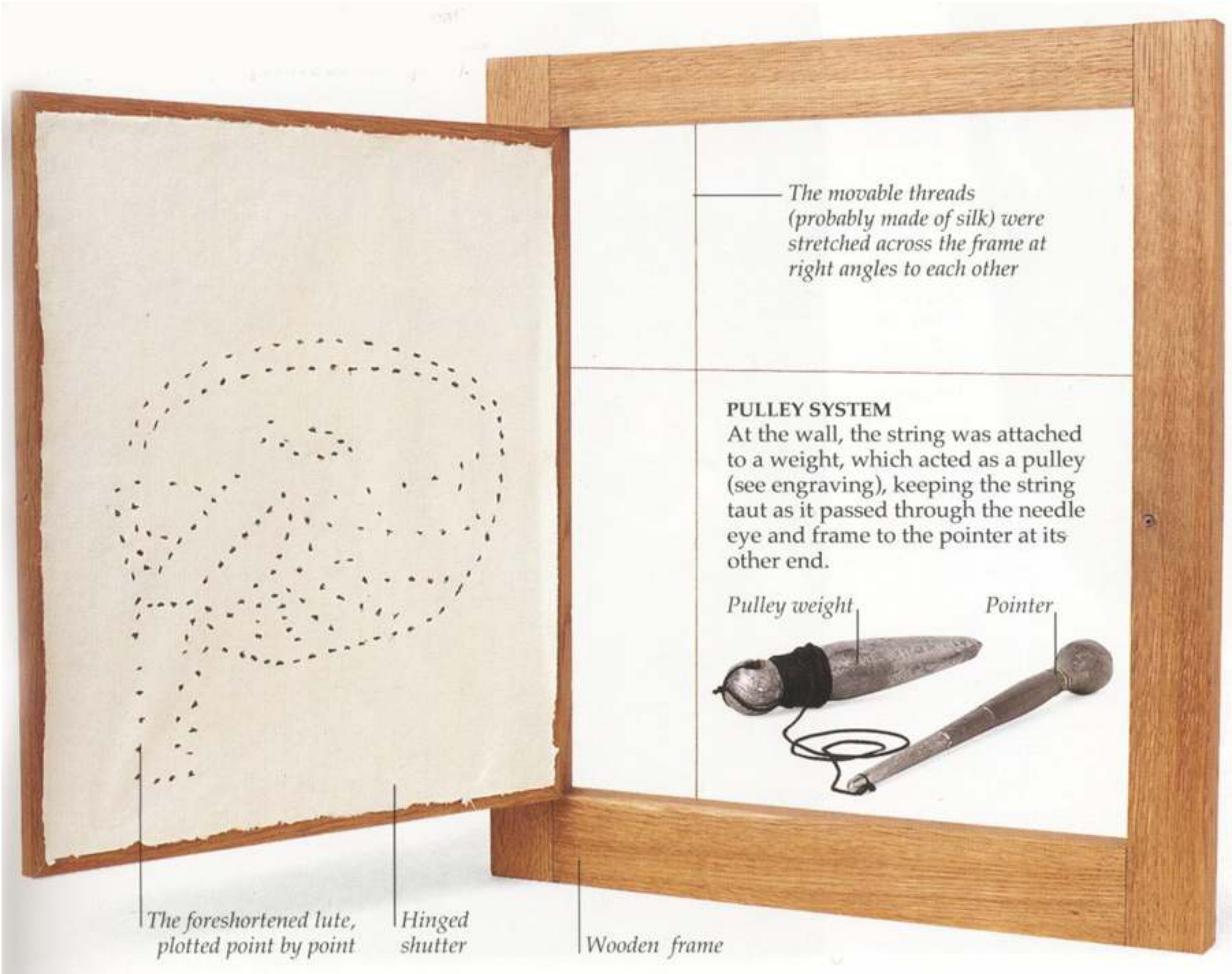
Probably the Oldest Conception of "Ray-Tracing"

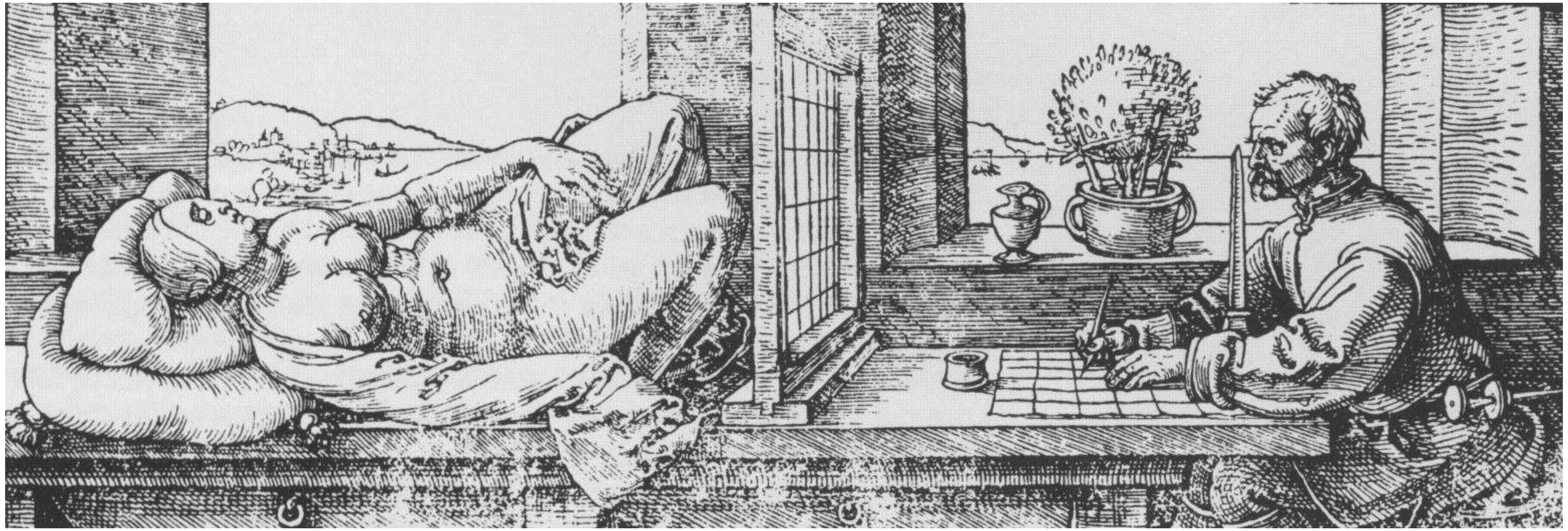
Emission theory
(conjectured by most Greek scientists
and held until around 1500 among Western scientists)

Albrecht Dürer's "Ray Casting Machines"

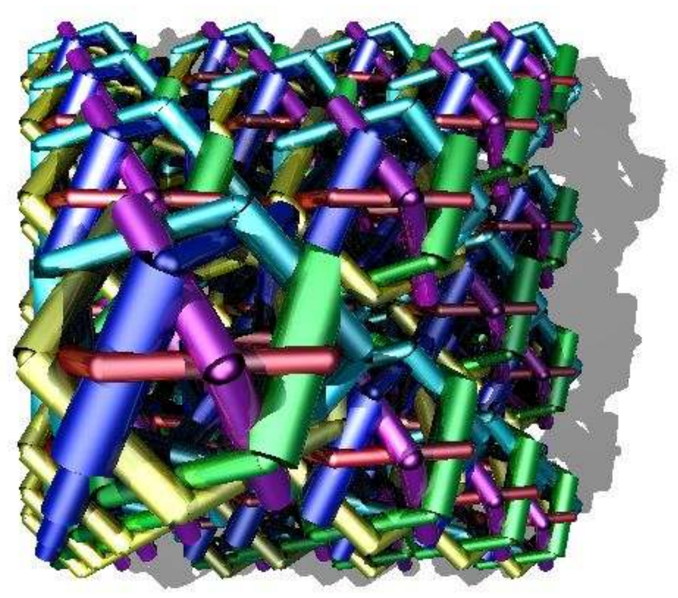
[16th century]

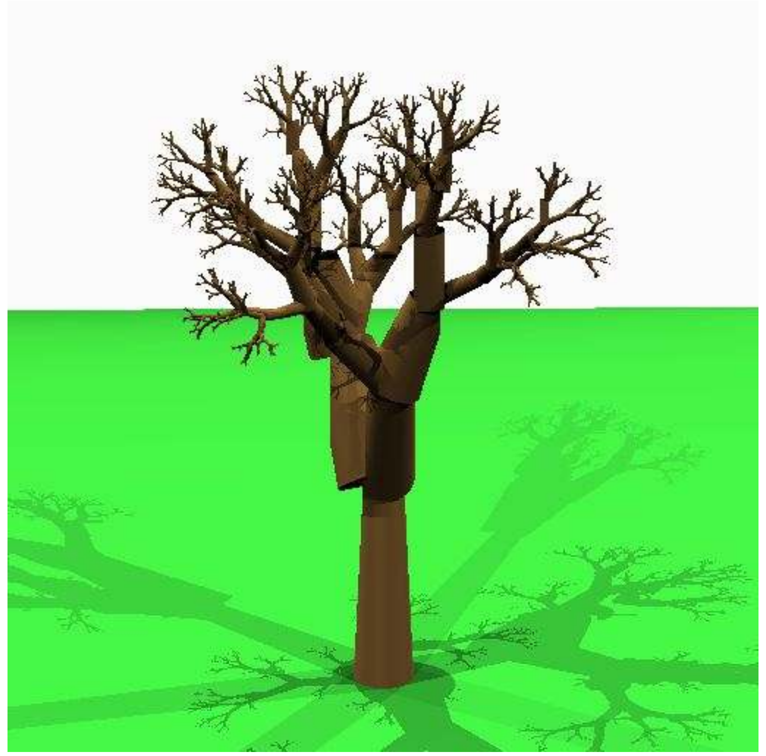
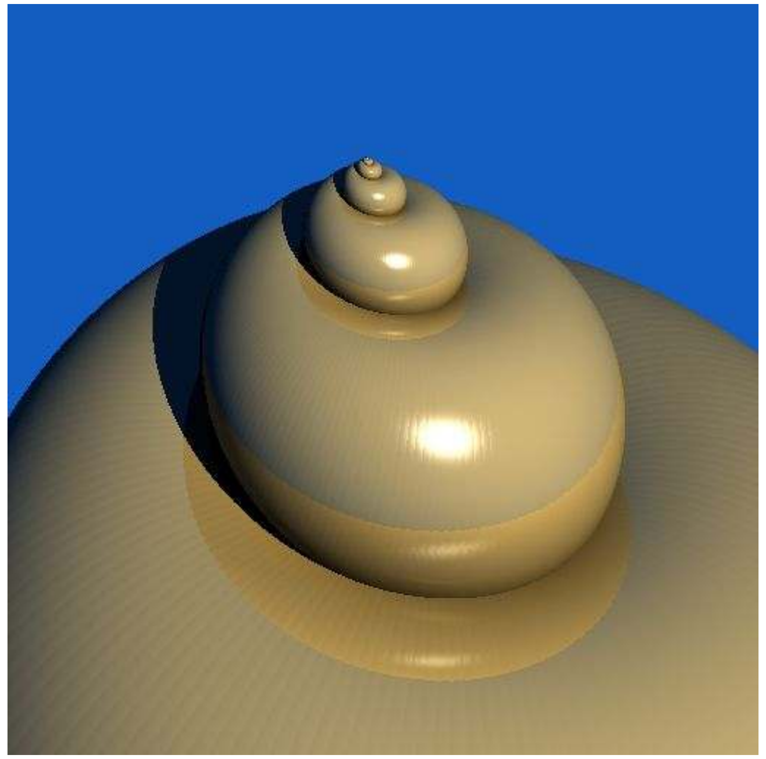
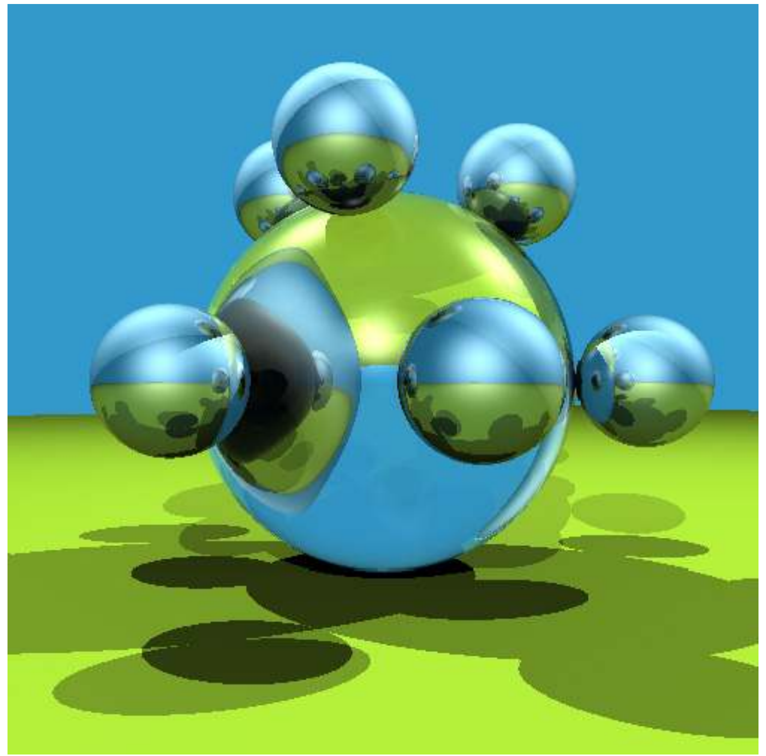


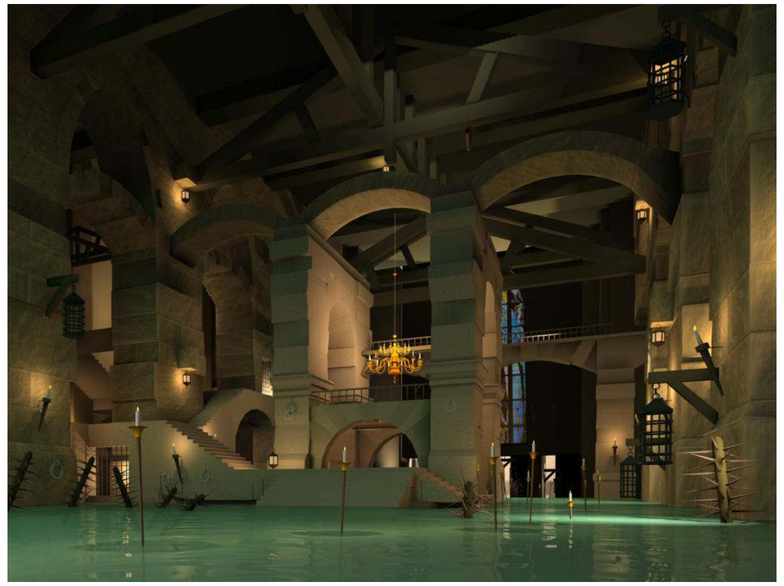
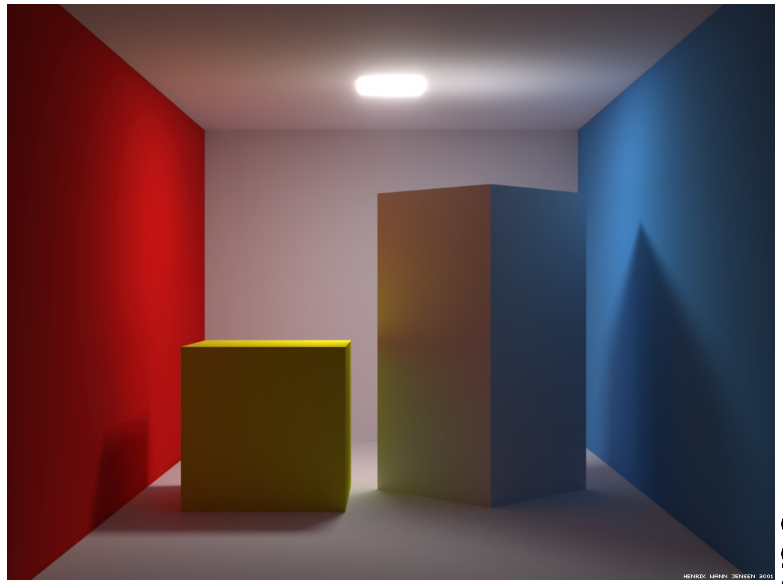


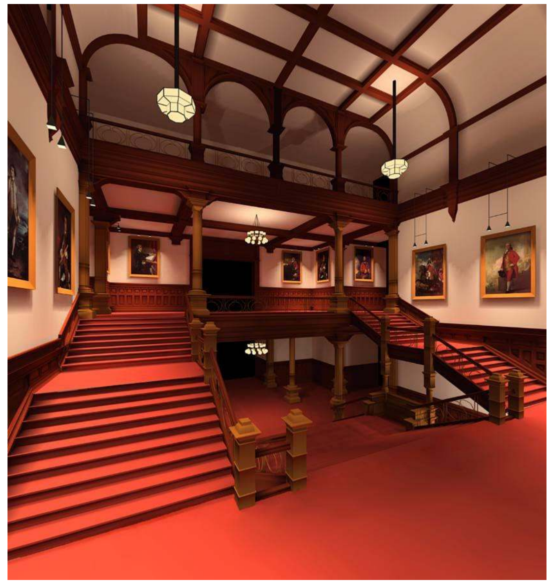


Examples of Ray-Traced Images

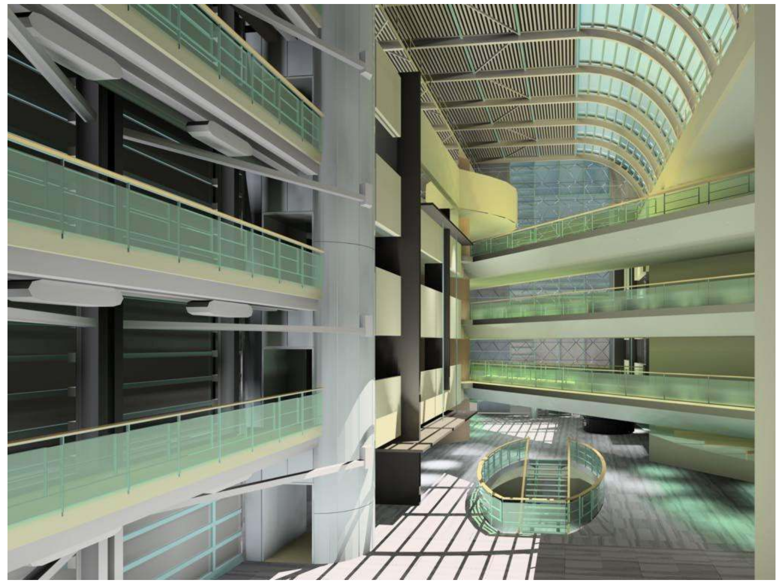


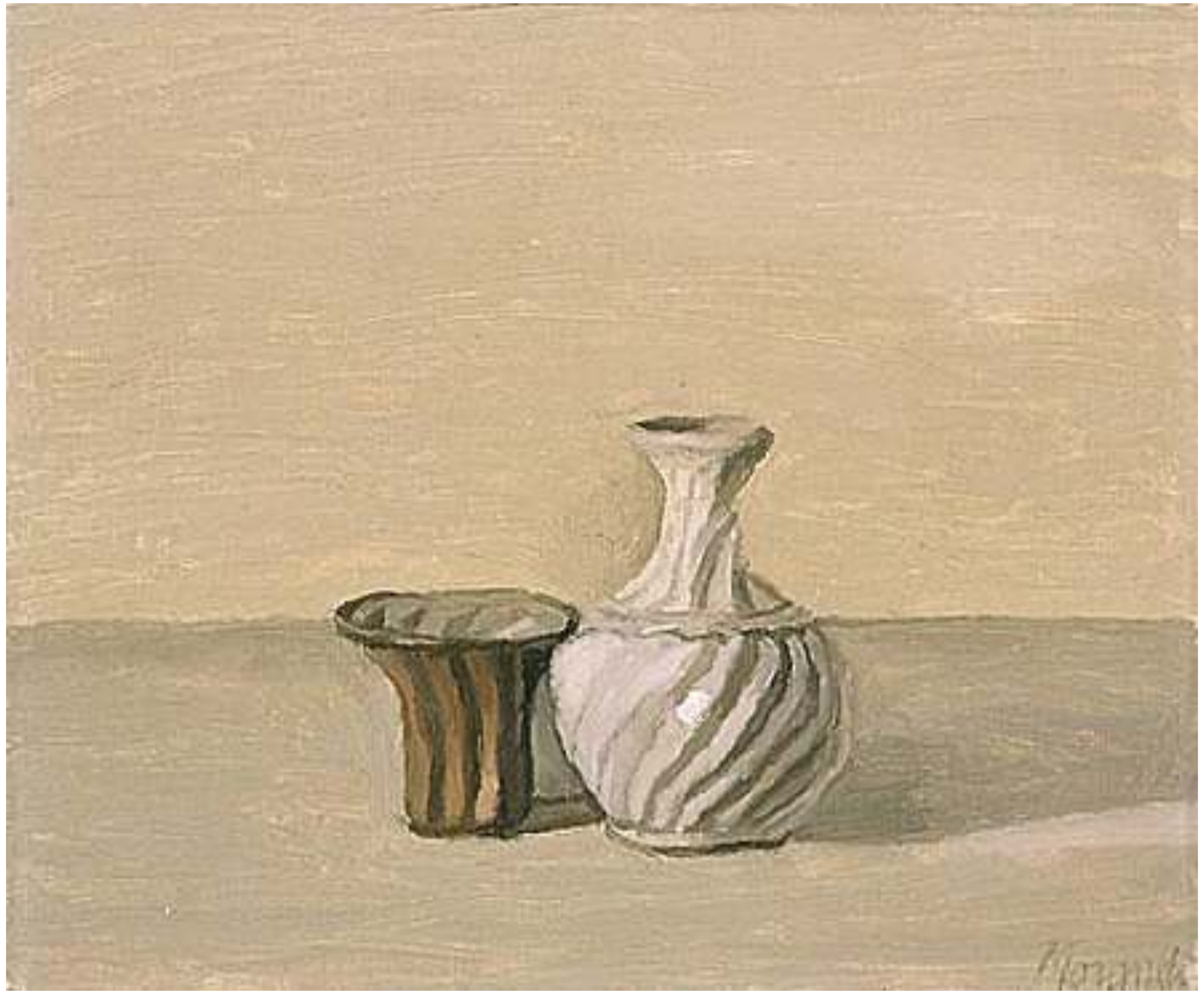




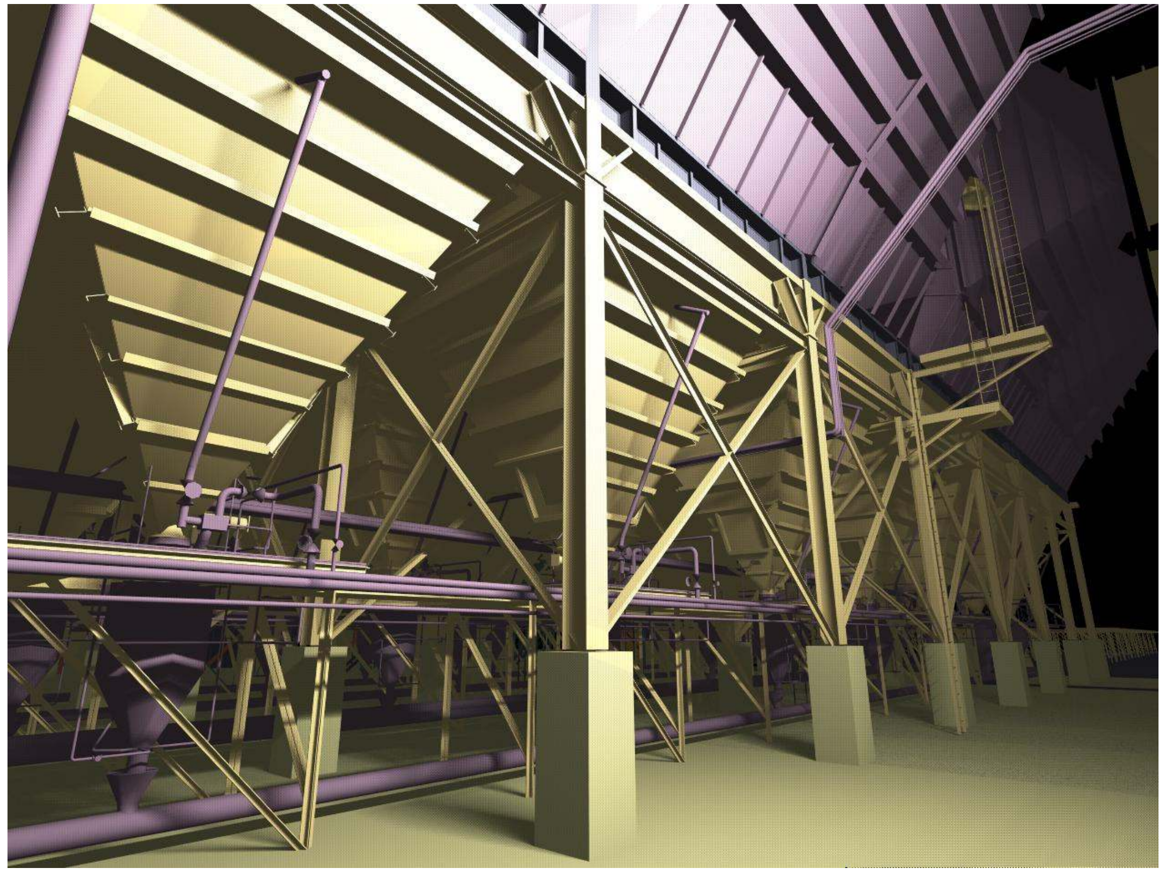
Jensen, Lightscape


Intermission: Giorgio Morandi





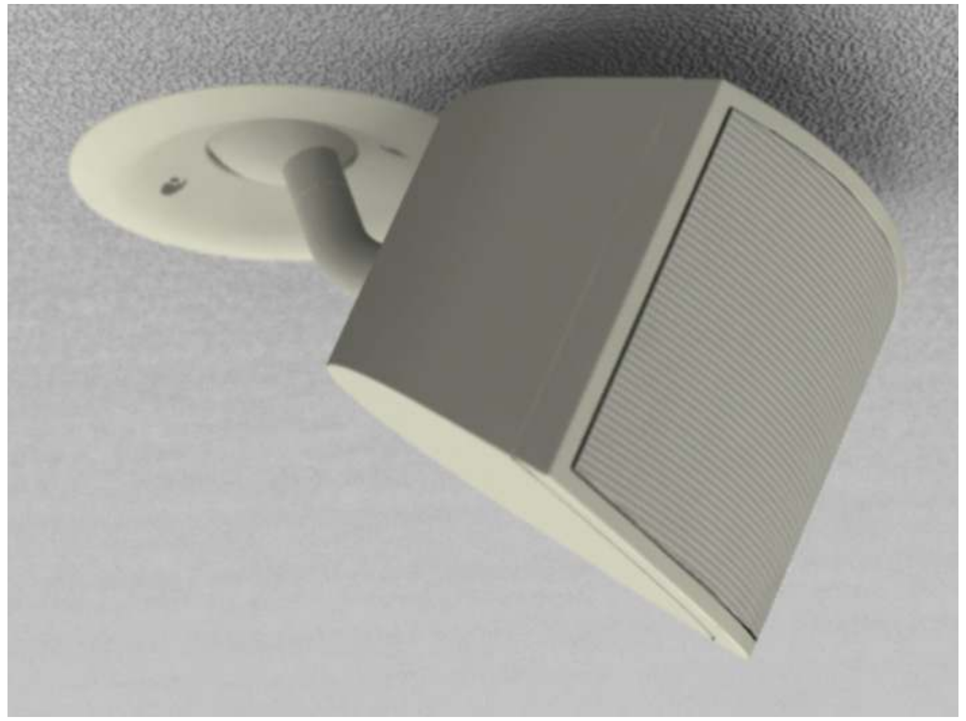



Ray Tracing in the Animation Industry

Doc Hudson's chrome bumper
with two levels of ray-traced reflection.
(Copyright 2006 Disney/Pixar)

Ray-traced wine glasses from *Ratatouille*.
(Copyright 2007 Disney/Pixar)

Fake or Real?

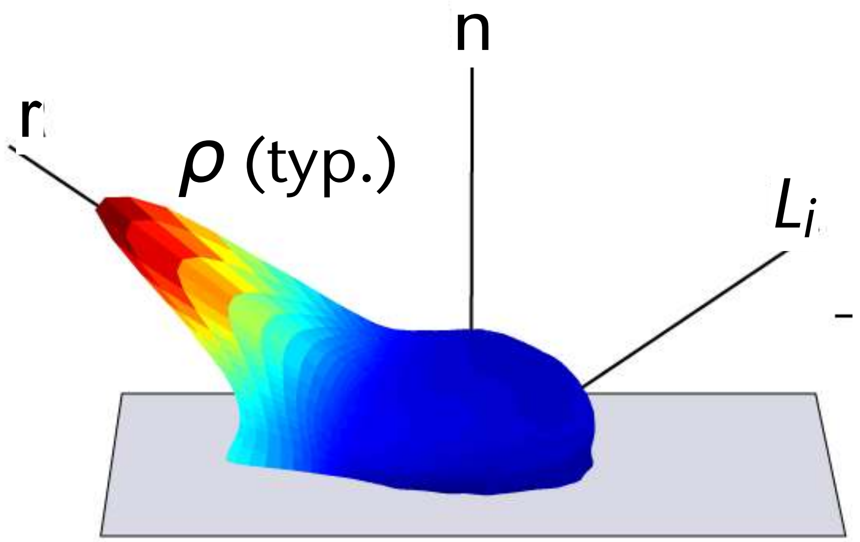


The Rendering Equation


Kajiya

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

$$L_o(x,\omega_o) = L_e(x,\omega_o) + \int_\Omega \rho(x,\omega_o,\omega_i) L_i(x,\omega_i)\cos(\theta_i)\,d\omega_i$$



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

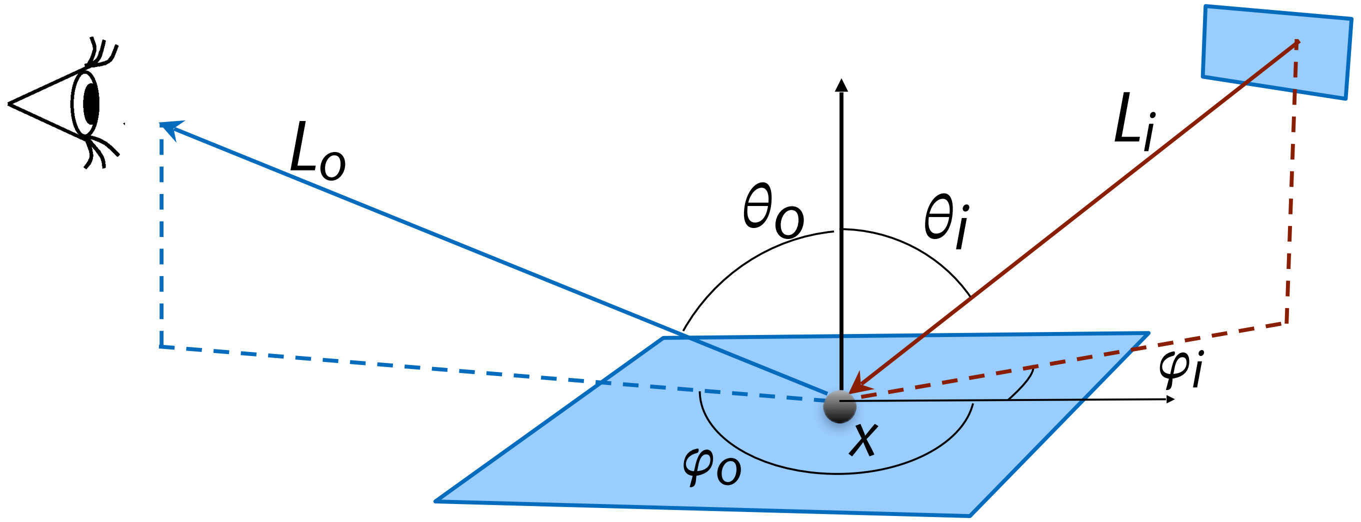
L_e = the "intensity" of light *emitted* (i.e., "produced") from x into direction ω_o

L_o = the "intensity" of light *reflected* from x into direction ω_o

ρ = function of the reflectance coefficient = BRDF

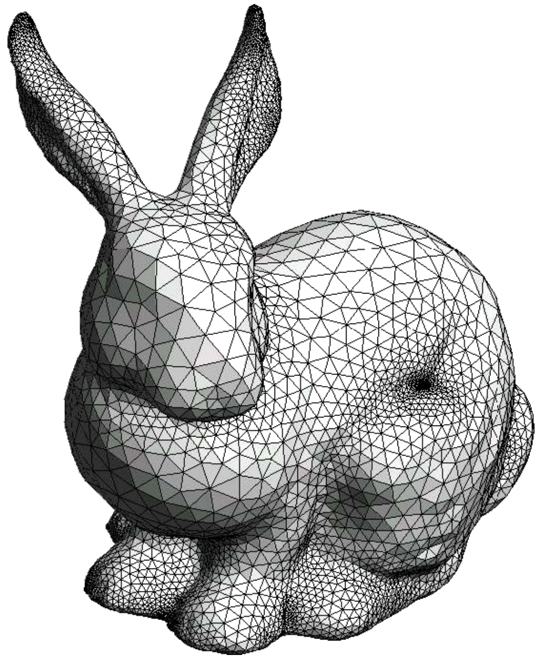
$\omega = (\theta,\varphi)$ = a direction (two polar angles)

Ω = hemisphere around the normal

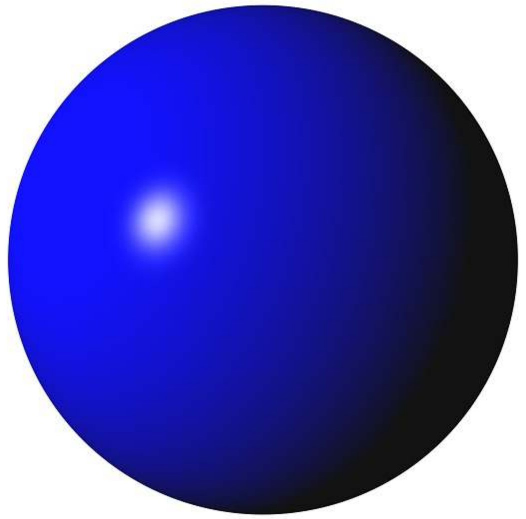


Output

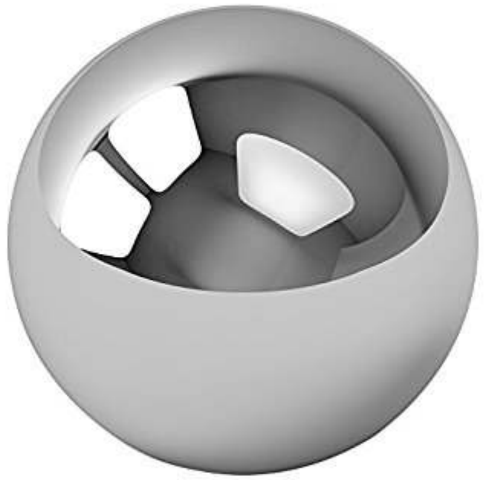
Inputs



Geometry



Material/Reflectance



Illumination

$$L_o = L_e + \int_{\Omega} \rho \cdot L_i \cdot \cos(\theta)\, \mathrm{d}\omega$$

Approximations to the Rendering Equation

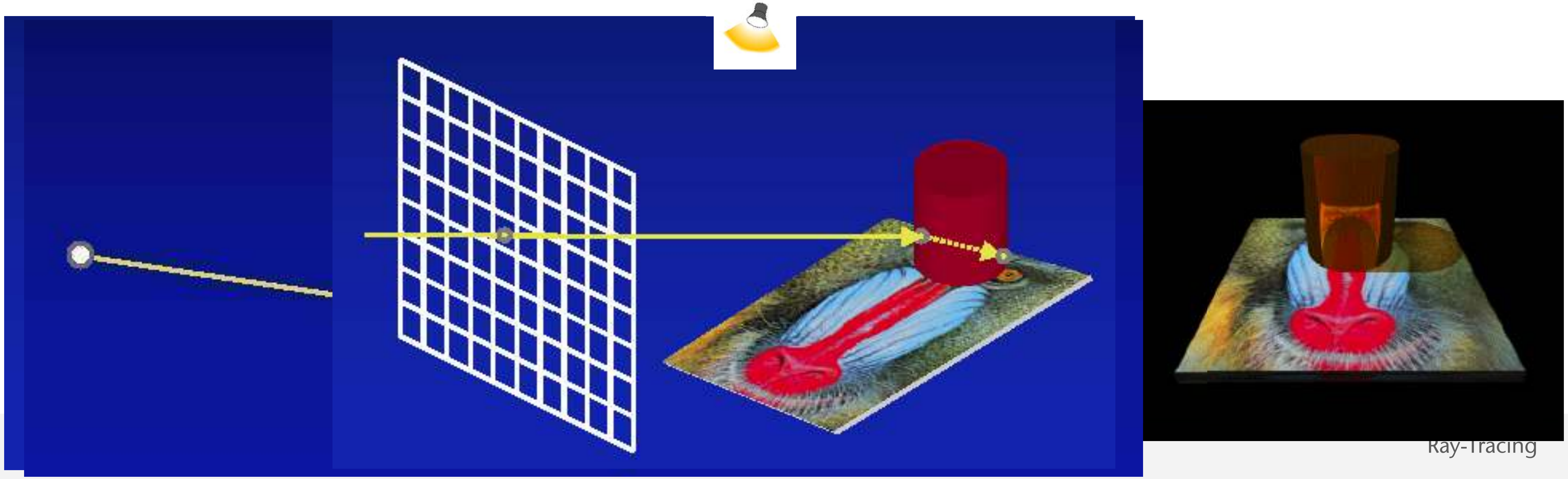
- Solving the rendering equation is impossible!
- Observation: the rendering equation is a recursive function
- Consequently, a number of approximation methods have been developed that are based on the idea of following rays:
  - Ray tracing (Whitted, Siggraph 1980,
    "An Improved Illumination Model for Shaded Display")
  - Lots of variations today:
    e.g., photon mapping, bi-directional path tracing
- Current state of the art:
  - Ray-tracing (aka. path tracing), combined with photon tracing, combined with Monte Carlo methods, combined with denoising filter

Turner Whitted,
Microsoft Research

The Simple "Whitted-Style" Ray-Tracing

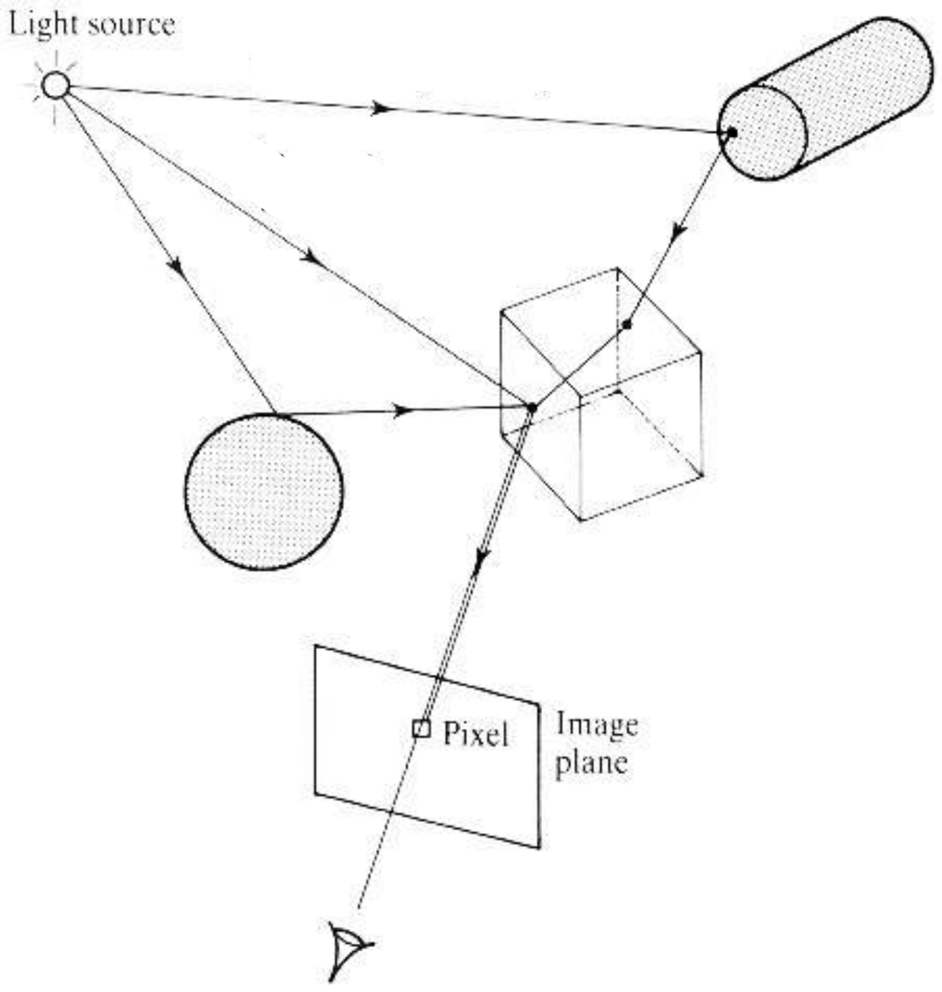
- Synthetic camera = viewpoint + image plane in world space

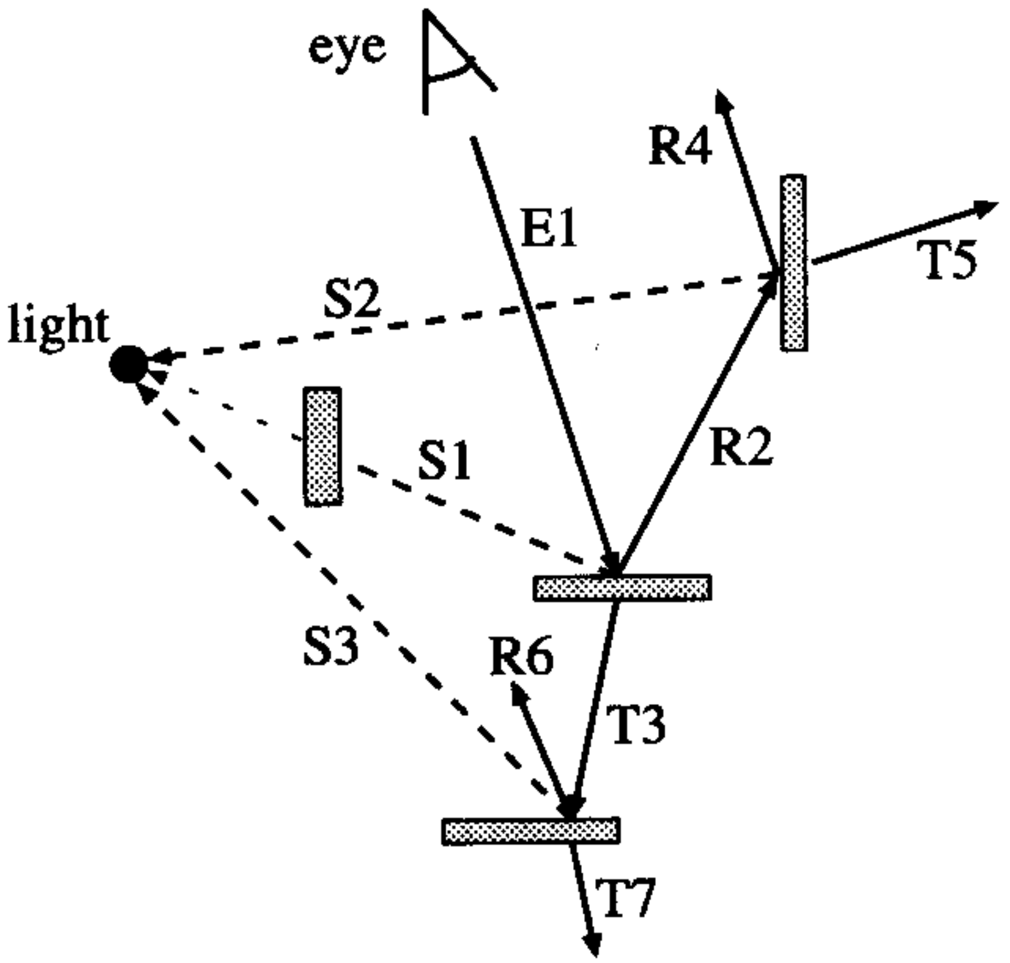
1. Shoot rays from camera through every pixel into scene (primary rays)
2. Compute the first hit with *any* of the objects in scene
3. From there, shoot rays to all light sources (*shadow feelers*)
4. If a shadow feeler hits another obj → point is in shadow w.r.t. that light source.
   Otherwise, evaluate 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

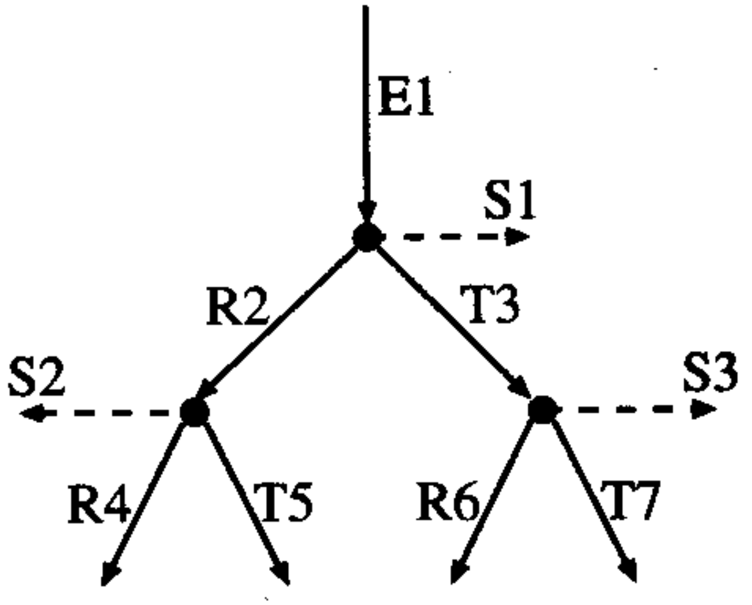


The Ray Tree

- 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:

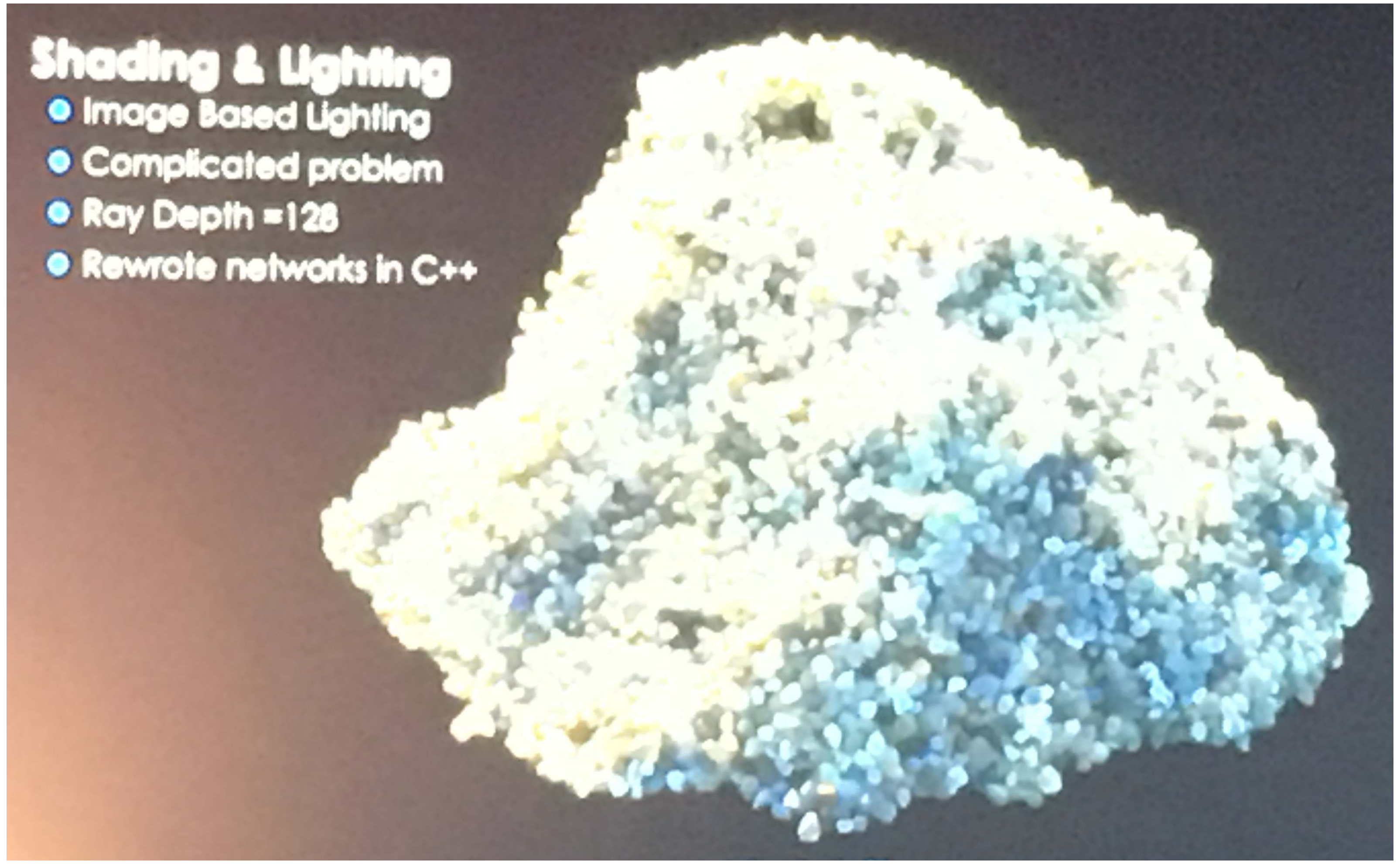






E1 = primary ray
Ri = reflected rays
Ti = transmitted rays
Si = shadow rays

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

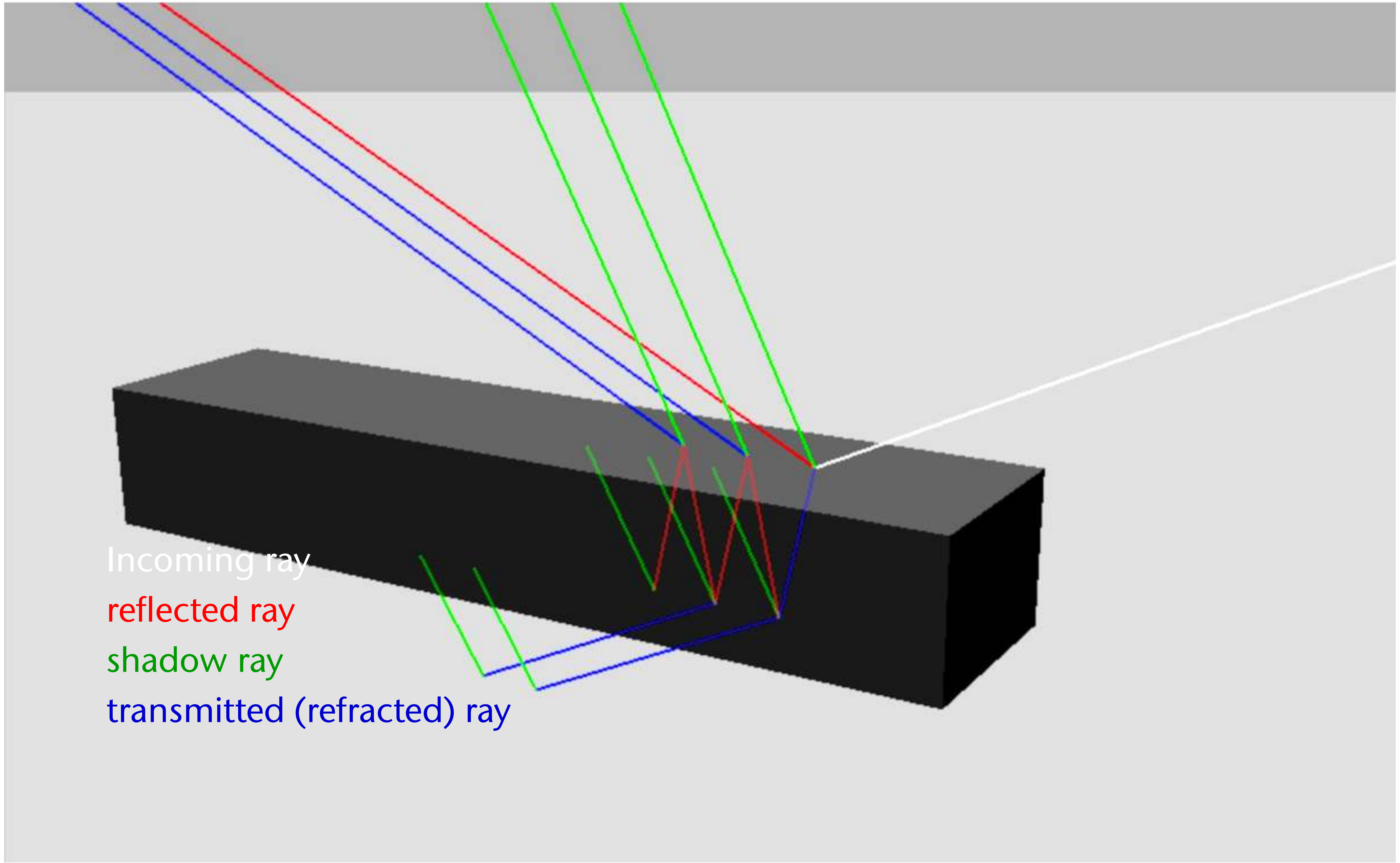


Max ray depth = 128 (!)

https://renderman.pixar.com/stories/piper

Excerpt from "Piper", Pixar 2017

Visualizing the ray tree can be very helpful for debugging



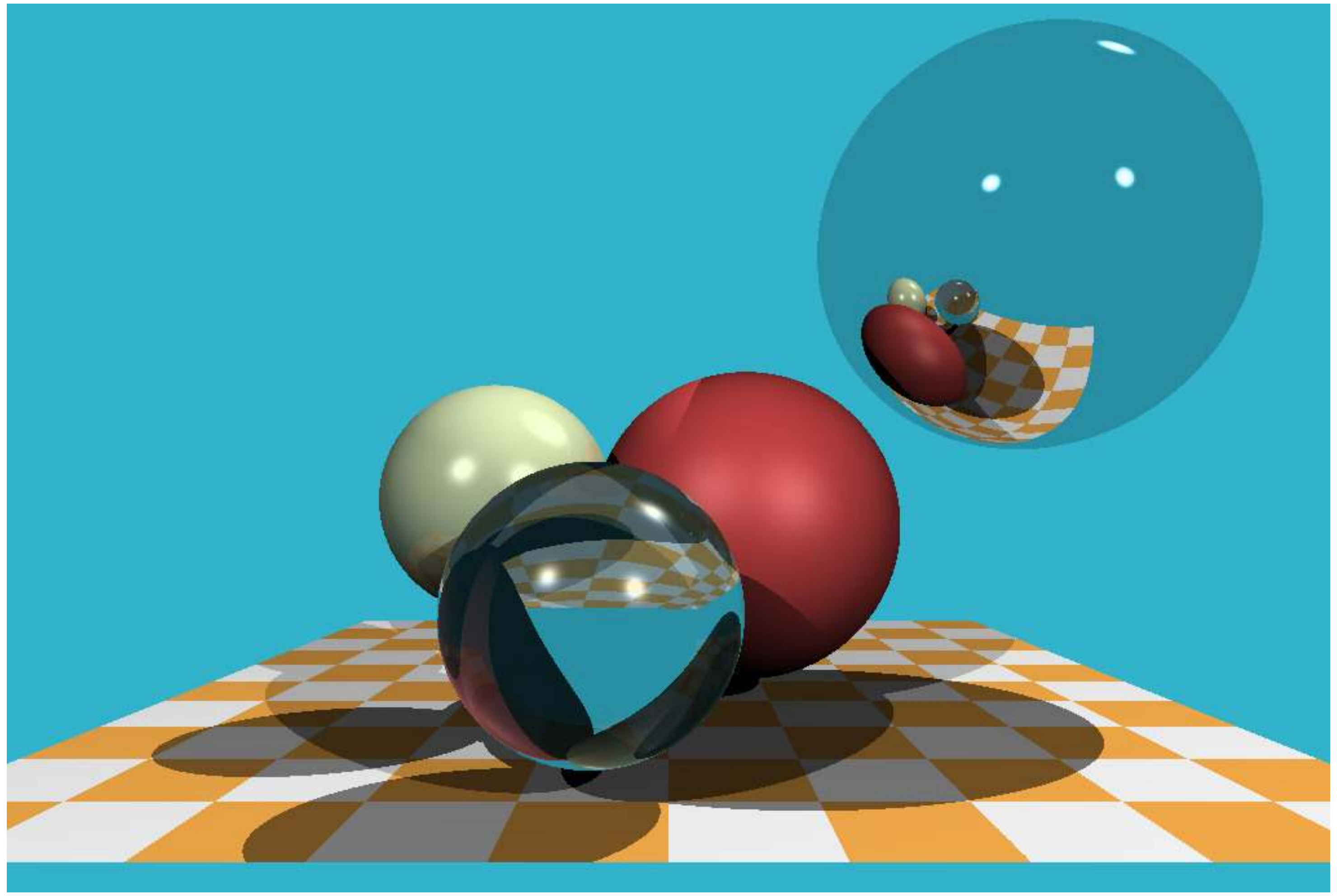
A Little Bit of Ray-Tracing Folklore

Paul Heckbert's business card (back), ca. 1994:

```
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!*/
```

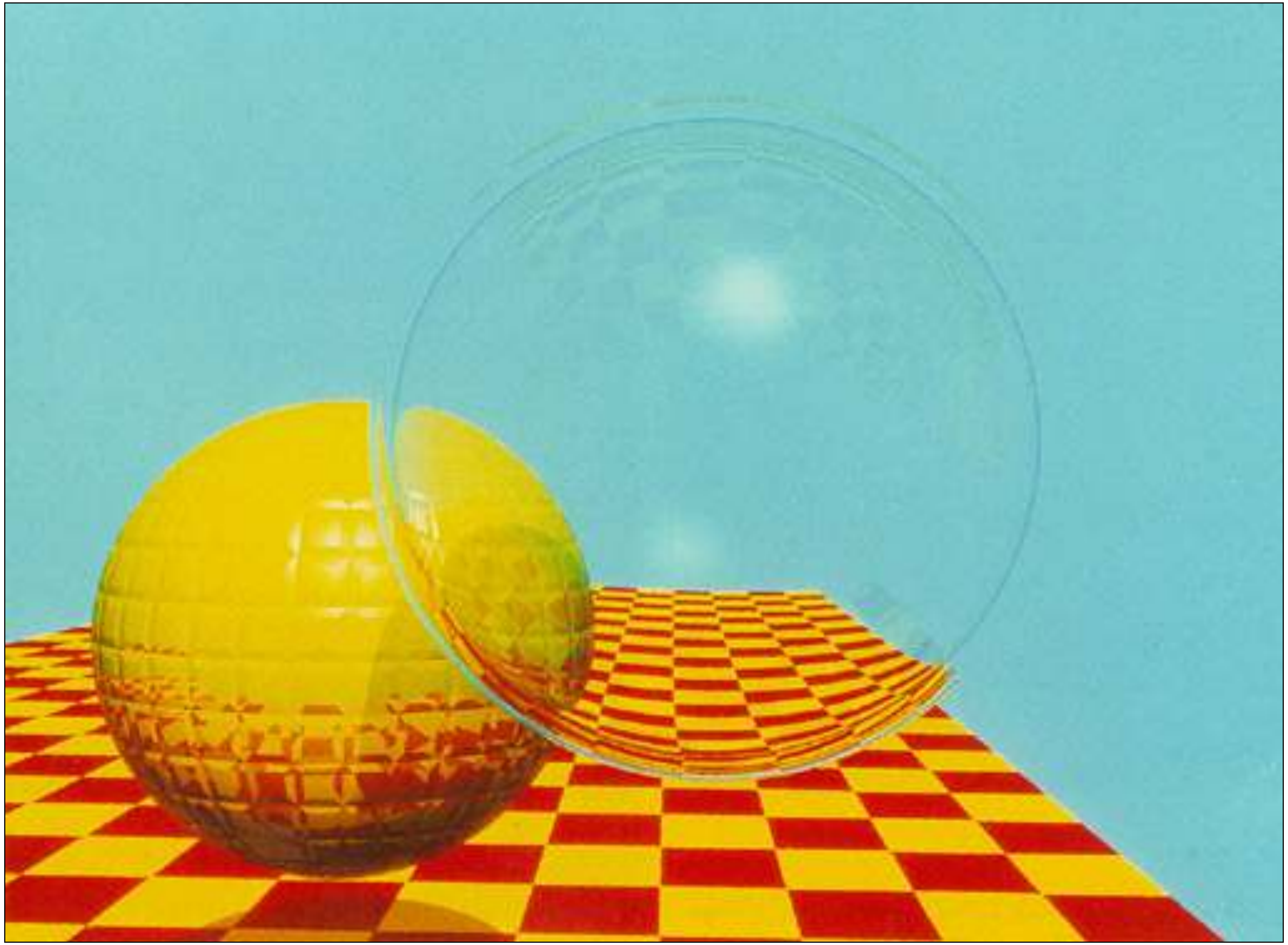
(Also won the *International Obfuscated C Code Contest*)!

Another ray tracer in 256 lines of C++:



https://github.com/ssloy/tinyraytracer

One of the First Ray-Traced Images



Turner Whitted 1980

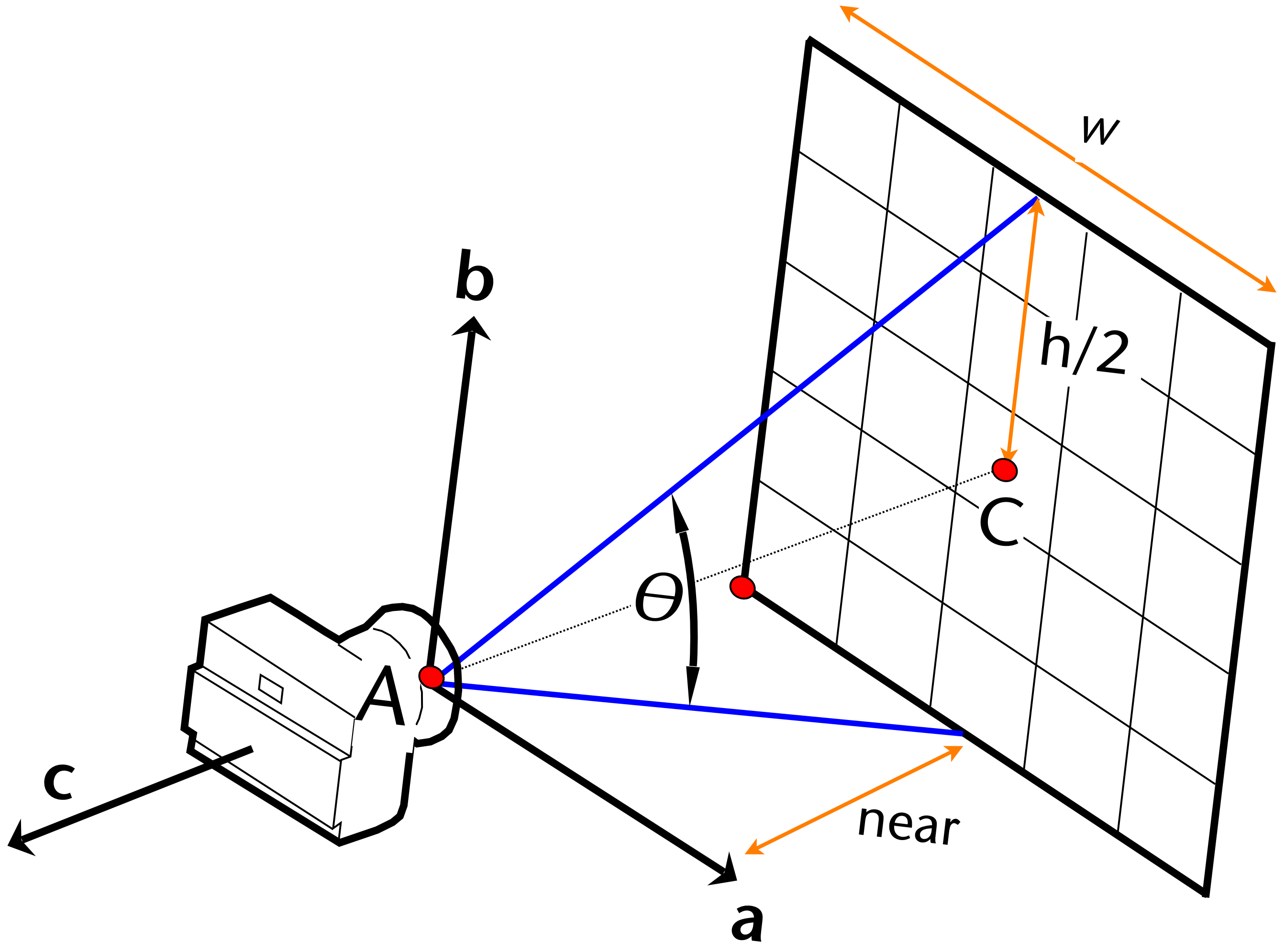
Basic Definition of Terminology

- Ray casting = geometric algorithm to compute intersections of primary rays with the scene (aka. ray-based visibility)
- Ray tracing / Path tracing = algorithm to compute global illumination by recursively(!) shooting rays in all kinds of directions
- Ray types: primary rays, secondary rays, shadow rays/feelers
-

Basic Ingredients Needed for Ray-Tracing

1. Primary rays ⟶ camera model
2. Secondary rays (reflected and transmitted rays) ⟶ (geometric) optics laws
3. Combining all incoming light (from secondary rays and shadow feelers) into "one" outgoing light ⟶ lighting models

A Simple Camera Model (Ideal Pin-Hole Camera)

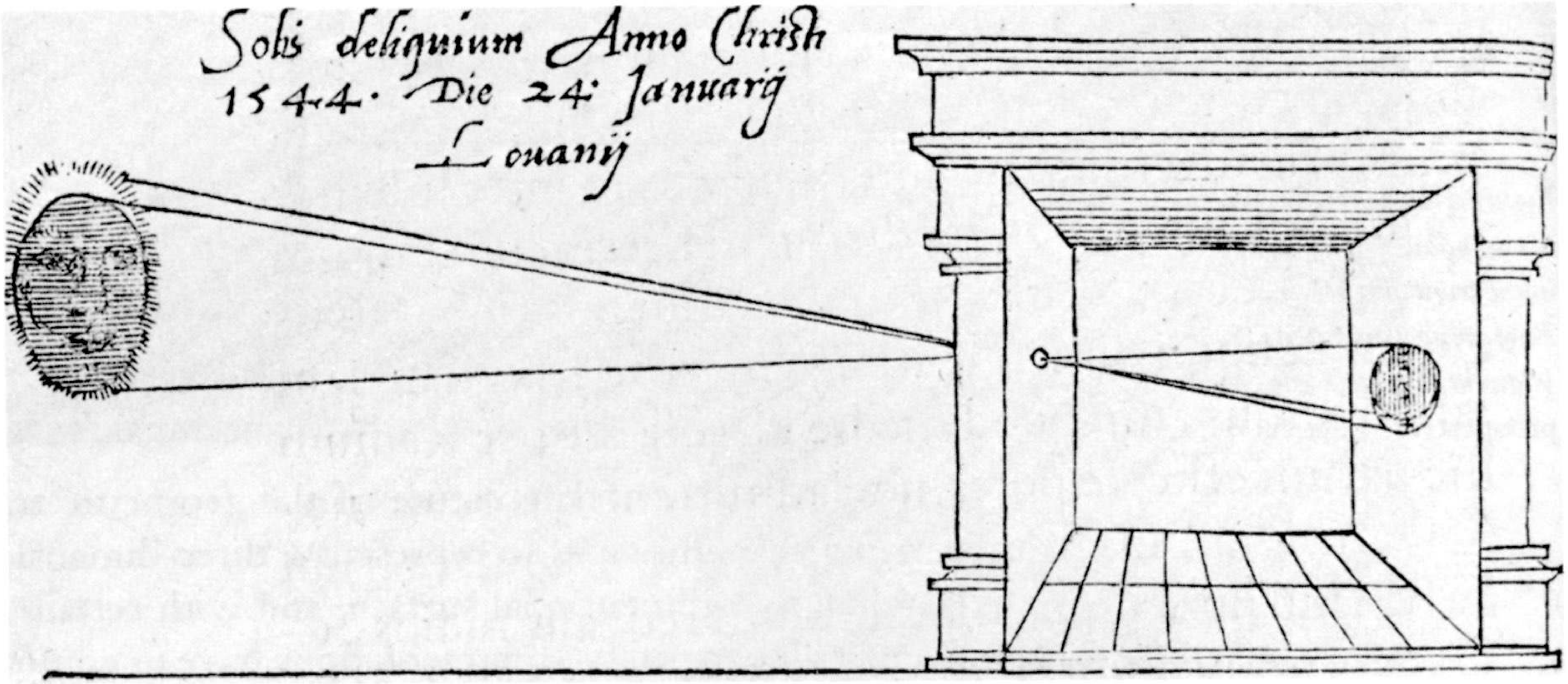


$$\frac{h}{2} = \text{near} \cdot \tan \frac{\theta}{2}$$

The main loop of ray-tracers

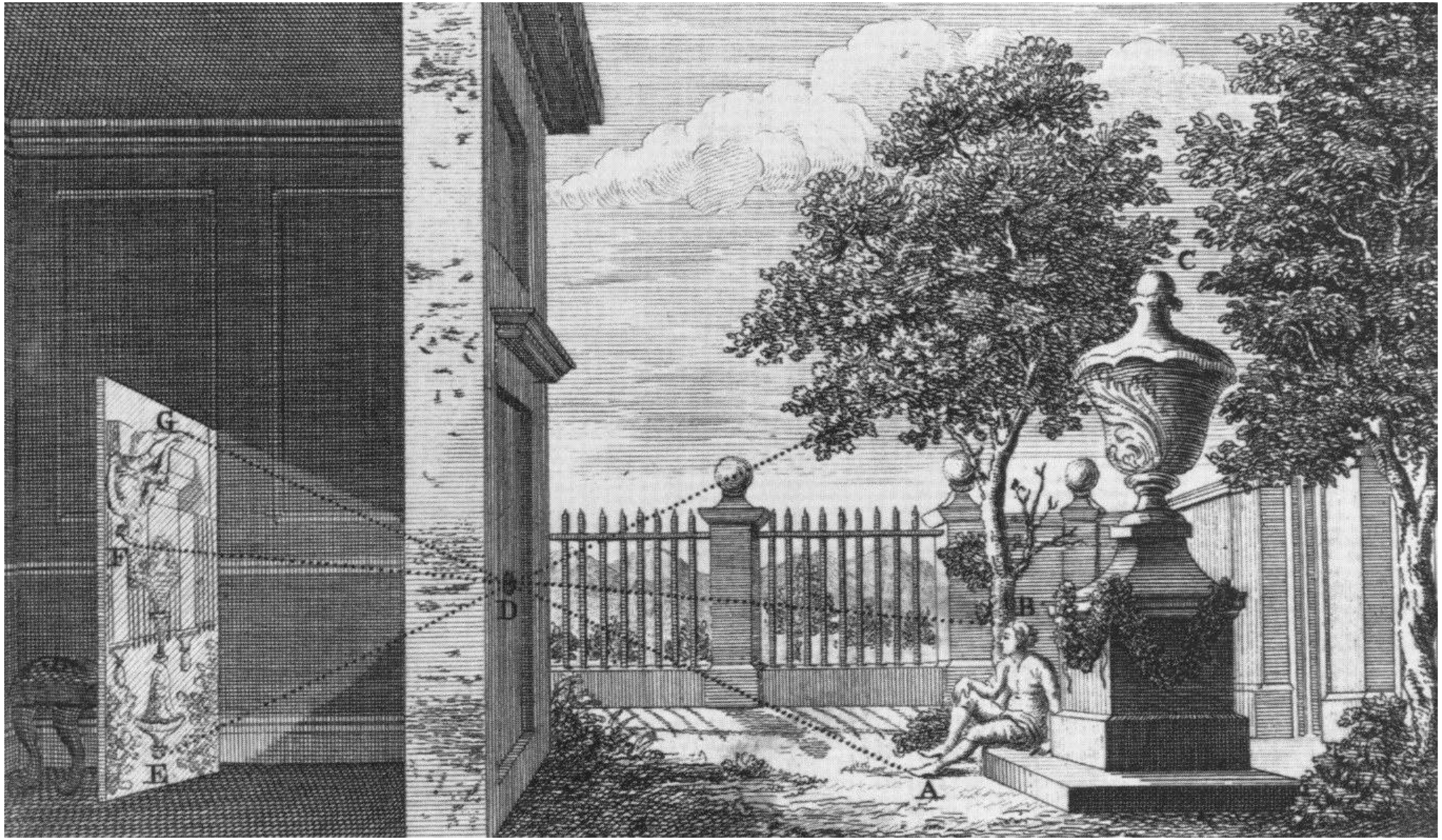
```
def gen_prim_rays( vec3 a, vec3 b,
                   vec3 A, vec3 C ):
  for i = 0 .. hor_res:
    for j = 0 .. vert_res:
      ray.from = A
      s = (i/hor_res - 0.5) * h
      t = (j/vert_res - 0.5) * w
      ray.at = C + s*a + t*b
      vec3 color = traceRay( 0, ray )
      putPixel( x, y, color )
```

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

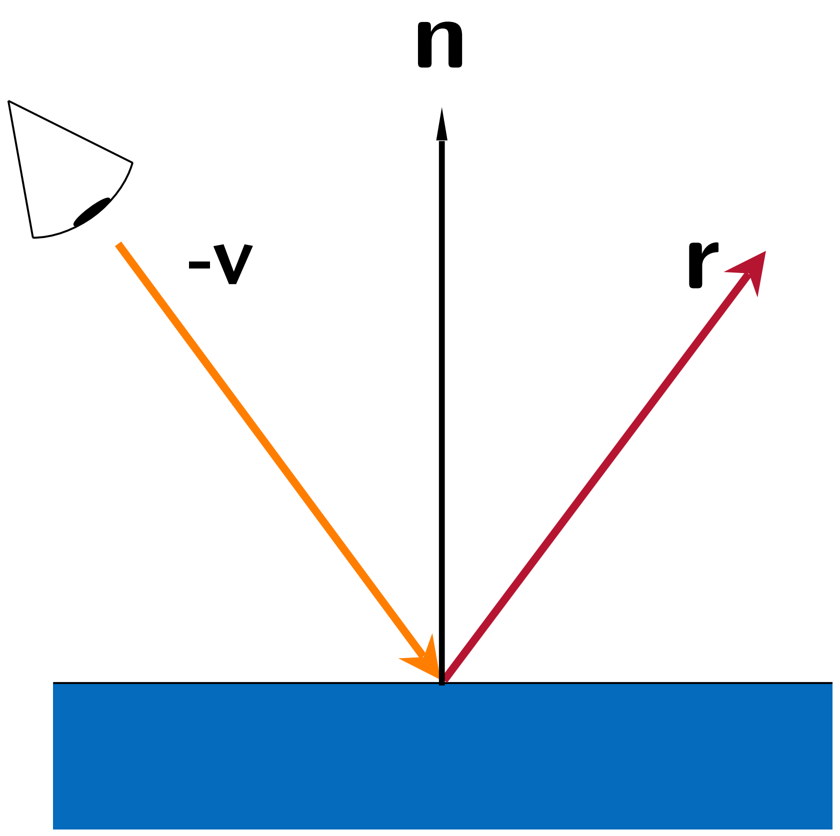
Quiz:
How was this funny projection achieved?

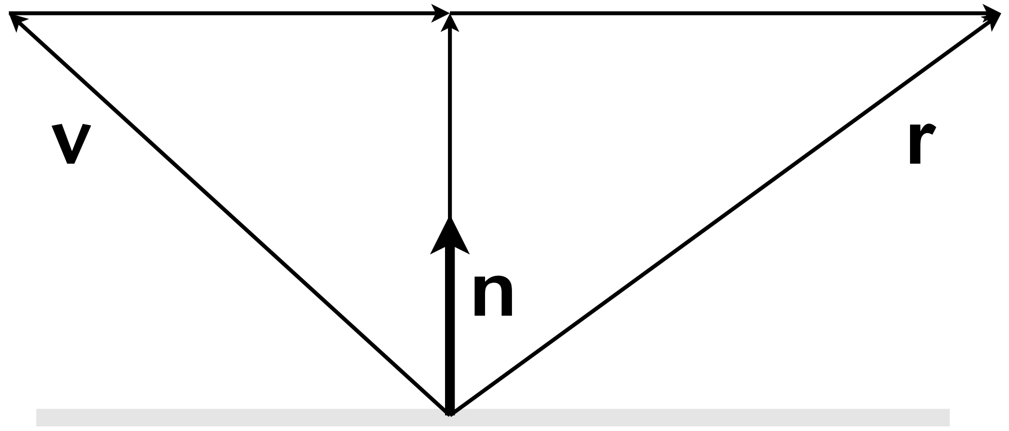
Generation of Secondary Rays

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

$$\mathbf{r} = \big((\mathbf{v}\cdot\mathbf{n})\cdot\mathbf{n} - \mathbf{v}\big)\cdot 2 + \mathbf{v}$$
$$= 2(\mathbf{v}\cdot\mathbf{n})\cdot\mathbf{n} - \mathbf{v}$$

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





Specular Reflection in Whitted-Style Raytracing

- Just an additional term in the lighting model:

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

L_r = reflected light coming in from direction r

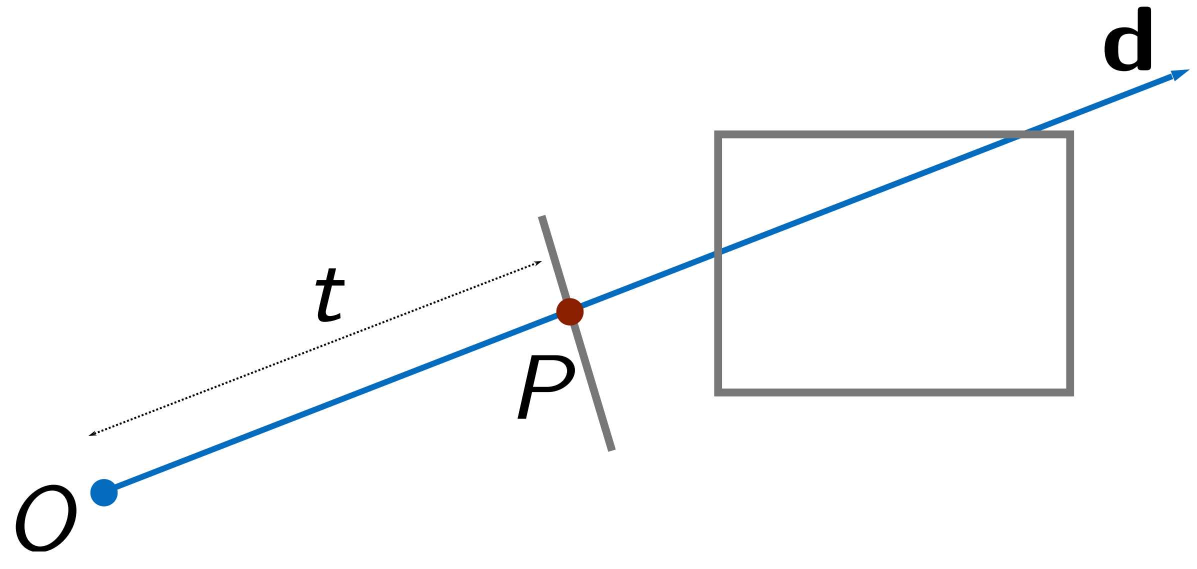
i.e, here we consider only specular reflection (i.e., no scattering)

k_s = material coefficient for specular reflection (the "color" of the object)

Intersection Computations Ray against Primitive

- Given: a set of objects (e.g., polygons, spheres, ...) and a ray

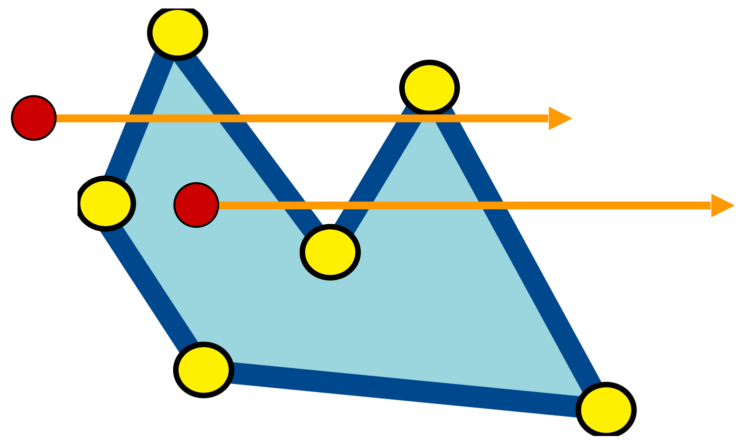
$$P(t) = O + t \cdot \mathbf{d}$$

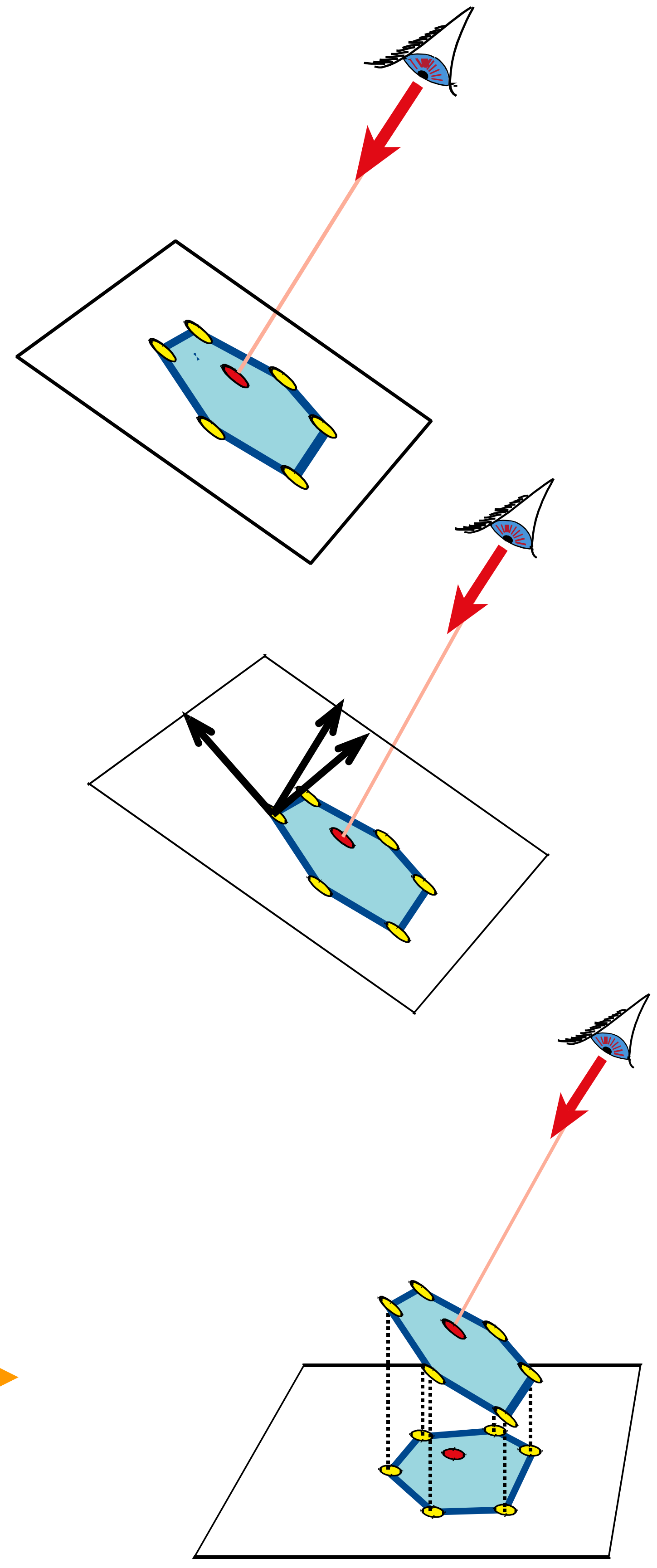


- Wanted: the line parameter t of the *first* intersection point $P = P(t)$ with the scene
- Amounts to the major part of the computation time
  - Solution: acceleration data structures, e.g., octree or bounding volume hierarchy

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:
  - Odd-even test using another (2D) ray:
    - #intersections = odd ⇔ point is inside
  - (In case of triangle, use barycentric coord test)





The Complete Ray-Tracing-Routine

```
traceRay( depth, ray ):
  hit = intersect( ray )
  if no hit:
    return no color
  reflected_ray = reflect( ray, hit )
  reflected_color = traceRay( depth+1, reflected_ray )
  if hit.material is transparent:
    refracted_ray = refract( ray, hit )
    refracted_color = traceRay( depth+1, 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
```

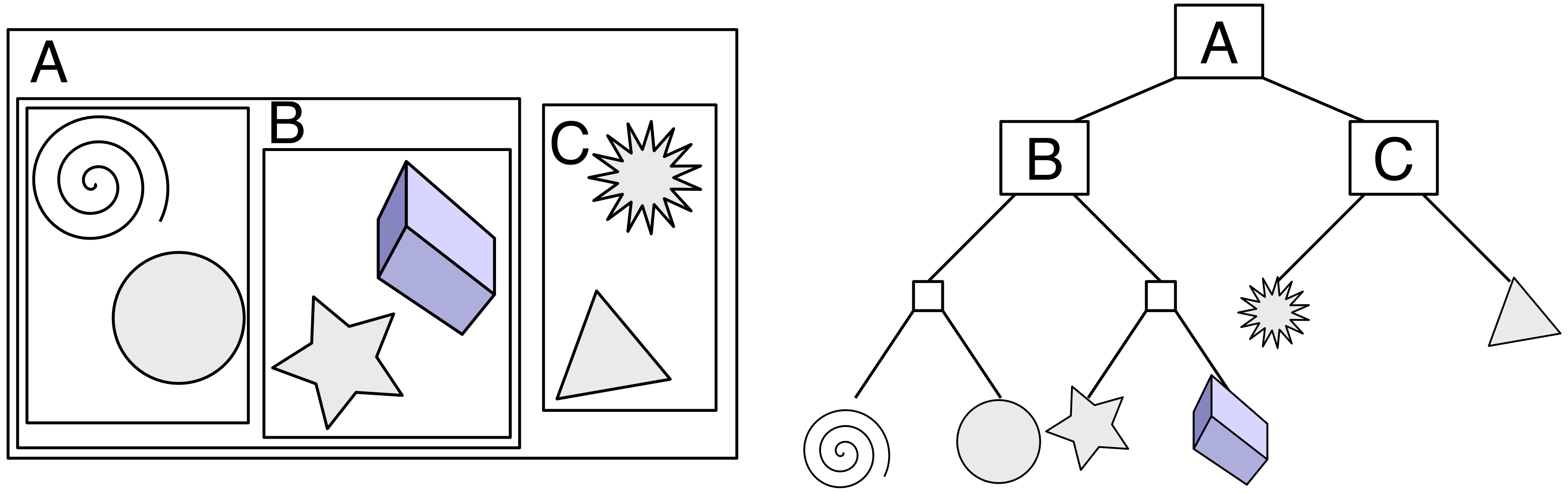
`hit` is a data structure (a struct or an instance of a class) that contains all infos about the intersectin between the ray and the scene, e.g., the intersection point, a pointer to the object, normal, …

The `intersect` function can be optimized compared to the one at the beginning; in addition, only intersection points *before* the light source are relevant.

Evaluates the lighting model for the hit object

Ausblick: Ray-Tracing auf der GPU

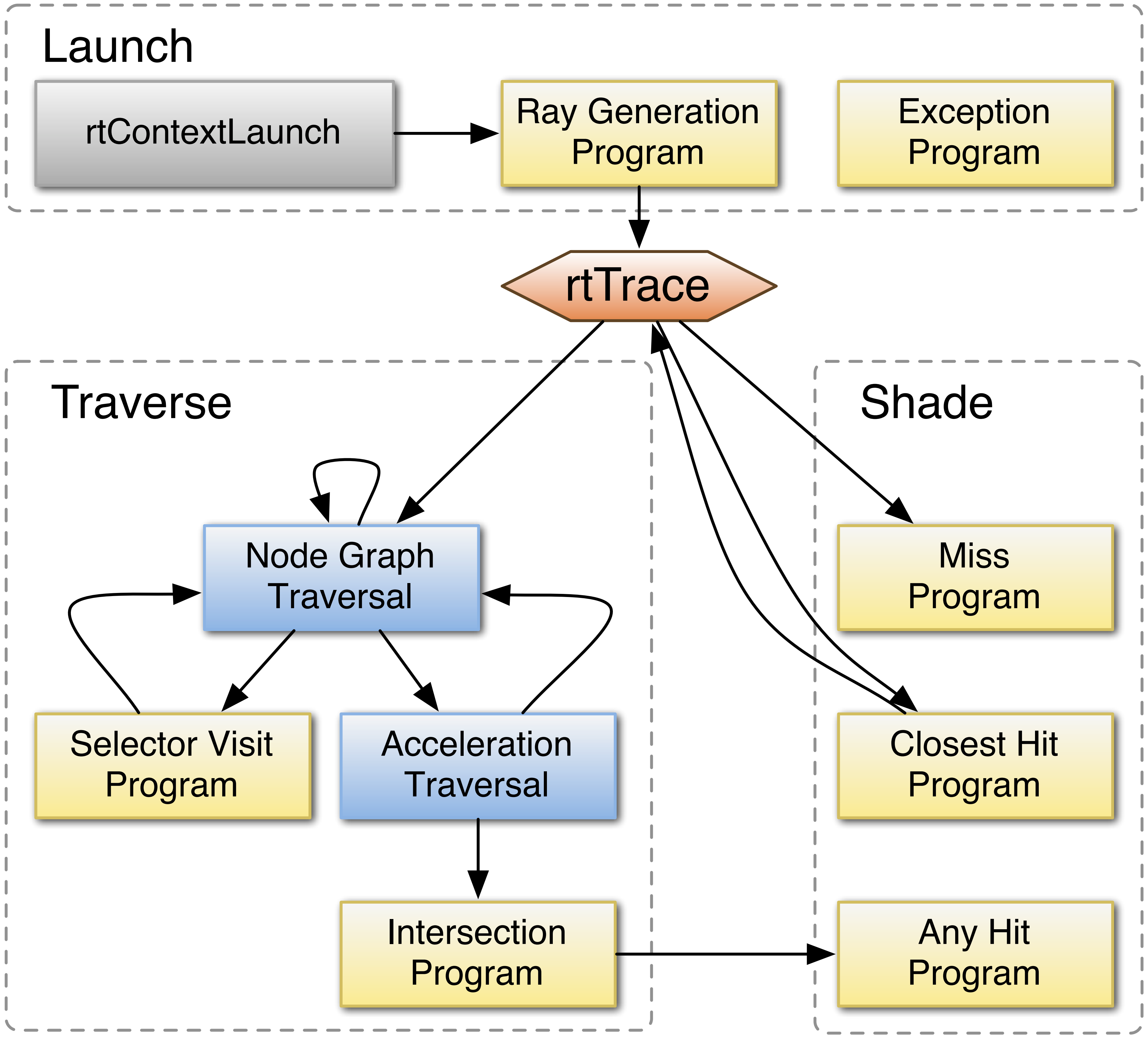
- Betrachte die Main Loop eines RT's: "embarrassingly parallel"
- Idee: pro Strahl ein Thread auf der GPU
- Bottleneck: Ray-Scene Schnittberechnung (Szene = 1M-100M Pgons)
- Konsequenz: verwende Acceleration Data Structure
  - Übliche Kandidaten sind: 3D-Gitter, kd-tree, Bounding-Volume-Hierarchy (BVH)
  - Siehe *Advanced Computer Graphics* [CG2] und/oder *Computational Geometry*)
- OptiX (Nvidia):
  verwendet BVH's
  - Sprache: C/C++ plus
    etwas GLSL, plus eigene
    kleine Erweiterungen (eine DSL)



Das Framework, High-Level Sicht

- Features/Funktionen von OptiX:
  - Erzeugt zu geg. Menge von Pgons eine BVH, ziemlich schnell, direkt auf der GPU
  - Traversiert die BVH für eine Menge von Rays gleichzeitig
    - Start mit Primary Rays, jeder einzelne wird durch App-provided RTprogram's generiert
  - Führt für verschiedene Aufgaben RTprogram's ("Shader") aus
- Host:
  - Lädt Menge Polygone hoch, stößt BVH-Konstruktion an
  - Lädt die RTprogram's hoch; insbesondere für
    - Generierung der primary rays
    - Behandlung eines echten (nähesten) Hit-Points ("closest hit program")
    - Behandlung, falls kein Schnitt ("miss program")
    - Behandlung von irgend einem Schnitt ("any hit program"), für Shadow Feelers
  - Stößt Generierung der Primary Rays an (und damit den ganzen Prozess)

Das OptiX Execution Model



Beispiel für ein RTprogram: der Generator für Primary Rays (viele Details fehlen)

```
RT_PROGRAM void pinhole_camera()
{
    size_t2 screen = output_buffer.size();
    float2 d = make_float2(launch_index) /
               make_float2(screen) * 2.0f - 1.0f;
    float3 ray_origin = eye;
    float3 ray_direction = normalize(d.x*a + d.y*b + C);
    Ray ray(ray_origin, ray_direction, ... );
    PerRayData_radiance prd;
    rtTrace( bvh_root, ray, prd );
    output_buffer[launch_index] = make_color(prd.result);
}
```

Jeder Thread bekommt seinen eigenen `launch_index`

GLSL-Syntax

Vordefinierte Vektoren **a, b, c** (wie auf Folie "Simple Camera Model")

OptiX ruft dieses RTprogram *parallel* für jedes Pixel des Framebuffers auf

Beispiel für ein Closest-Hit RTprogram (viele Details ausgelassen)

```
struct BasicLight
{
    float3 pos;
    float3 color;
};
rtBuffer<BasicLight> lights;

RT_PROGRAM void closest_hit()
{
   float3 world_shade_normal =
       normalize( rtTransformNormal(RT_OBJECT_TO_WORLD, shading_normal) );
   float3 color = ambient_light_color;
   float3 hit_point = ray.origin + t_hit * ray.direction;
   for ( int i = 0; i < lights.size(); ++i )
   {
      BasicLight light = lights[i];
      float3 L = normalize( light.pos - hit_point );
      float nl = optix::dot( shading_normal, L );
      if ( nl > 0 )
         color += Kd * nDl * light.color;
   }
   prd_radiance.result = color;
}
```