Advanced Computer Graphics
Introduction to Ray-Tracing and Physically-Based Rendering

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

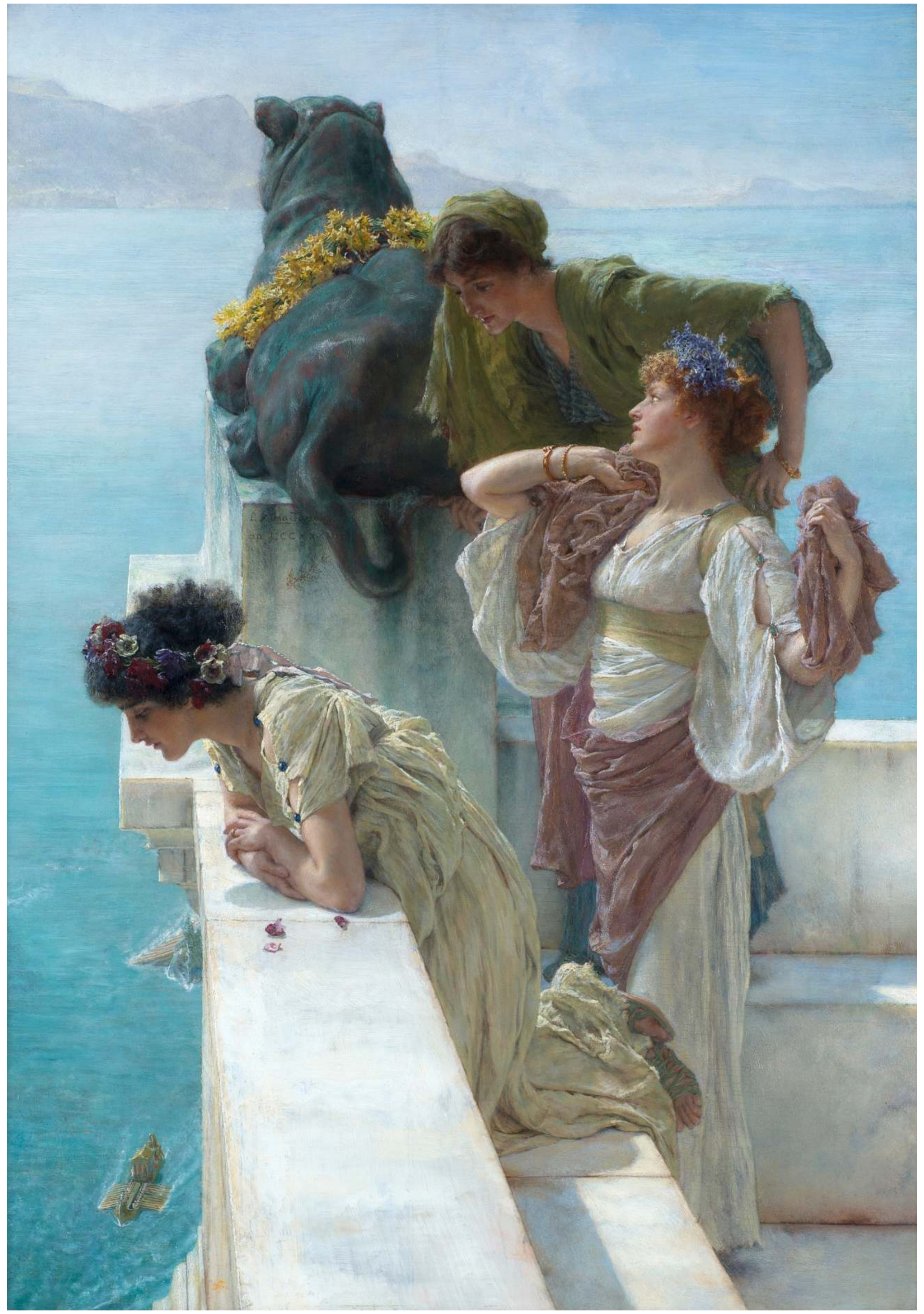
The Ongoing Quest for Realistic Images

"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."

– Pliny the Elder, 5th century B.C.

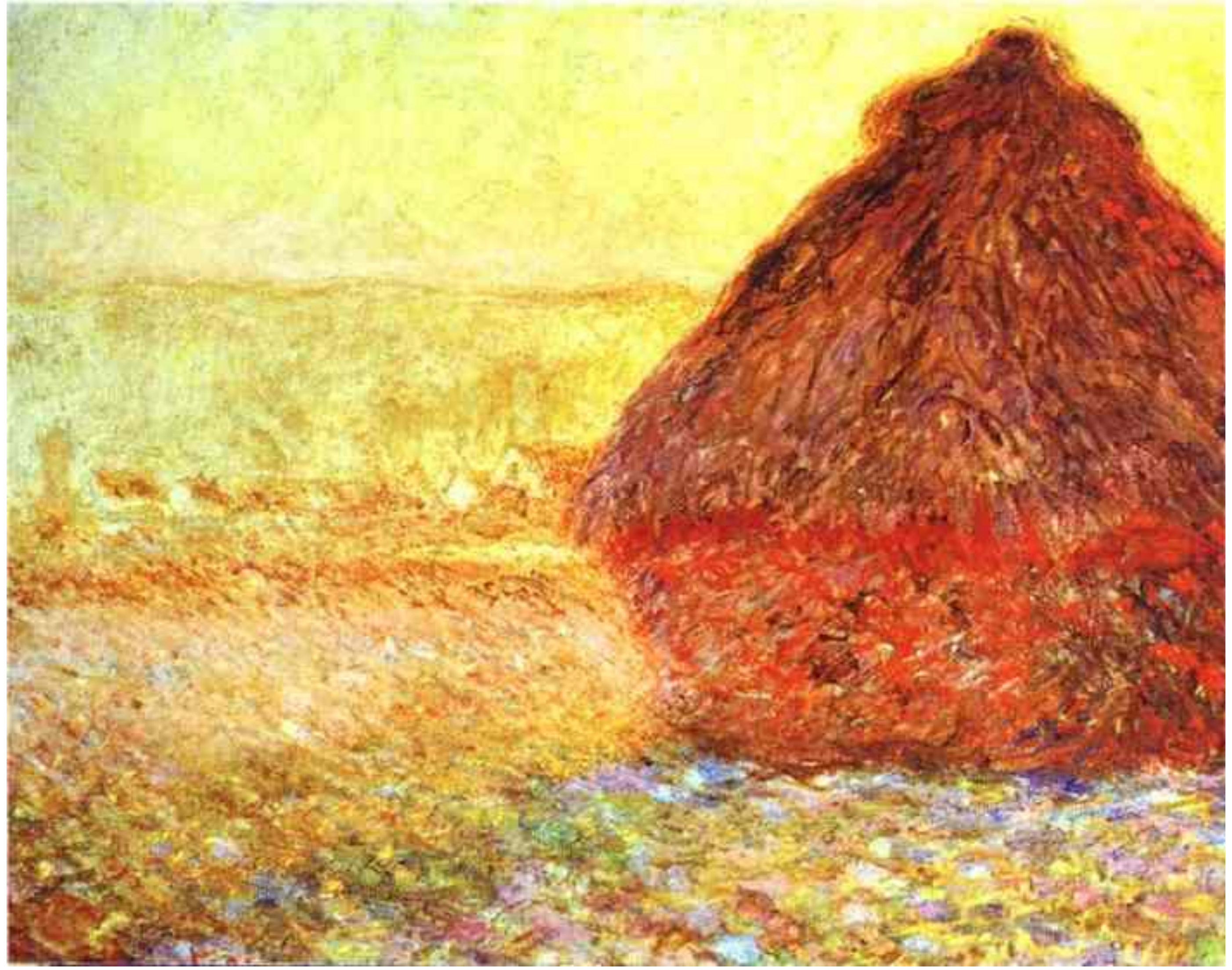
Examples from the History of Fine Art

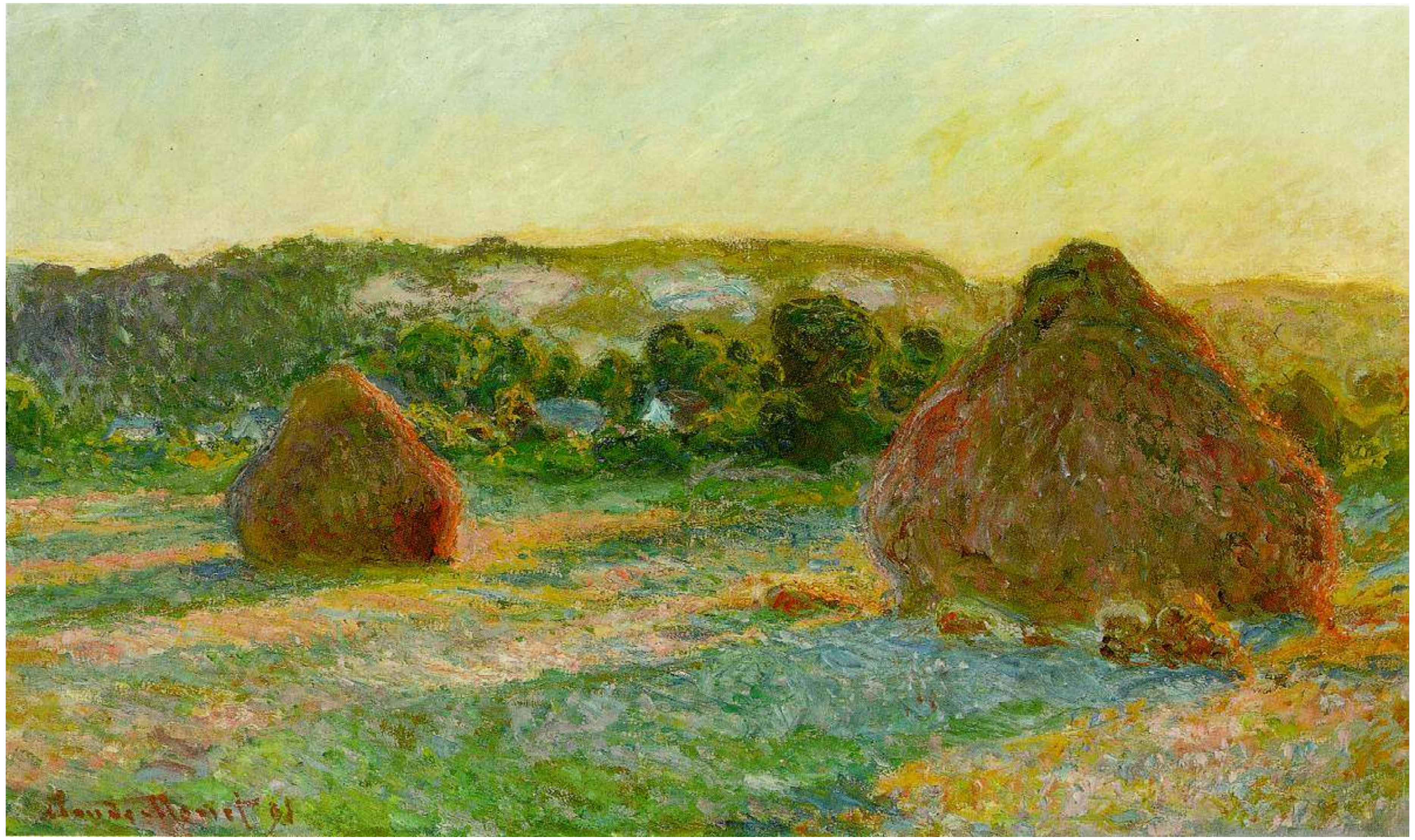
Willem Claesz. Heda, circa 1600-1663



Alma-Tadema: A Solicitation

By Contrast ...





Claude Monet's Haystacks

Effects Needed for Physically Correct Rendering

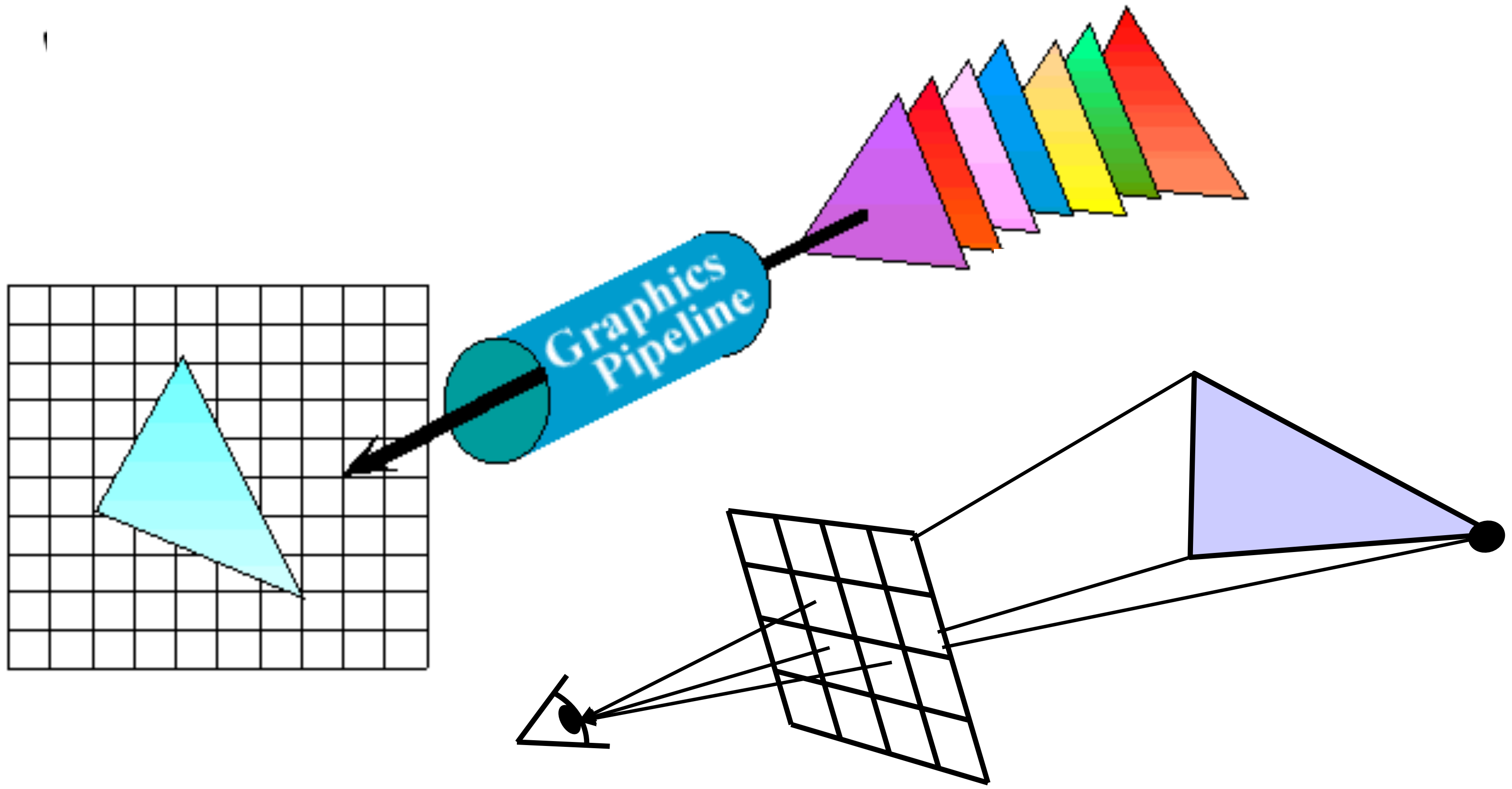
- Remember one of the local lighting models from CG1
- All *local* lighting models fail to render one or more of the following effects:
    - Soft Shadows (Halbschatten)
        - Hard shadows (Schlagschatten) can be done using multi-pass OpenGL rendering (see CG1)
    - Indirect lighting (sometimes 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
    - ...

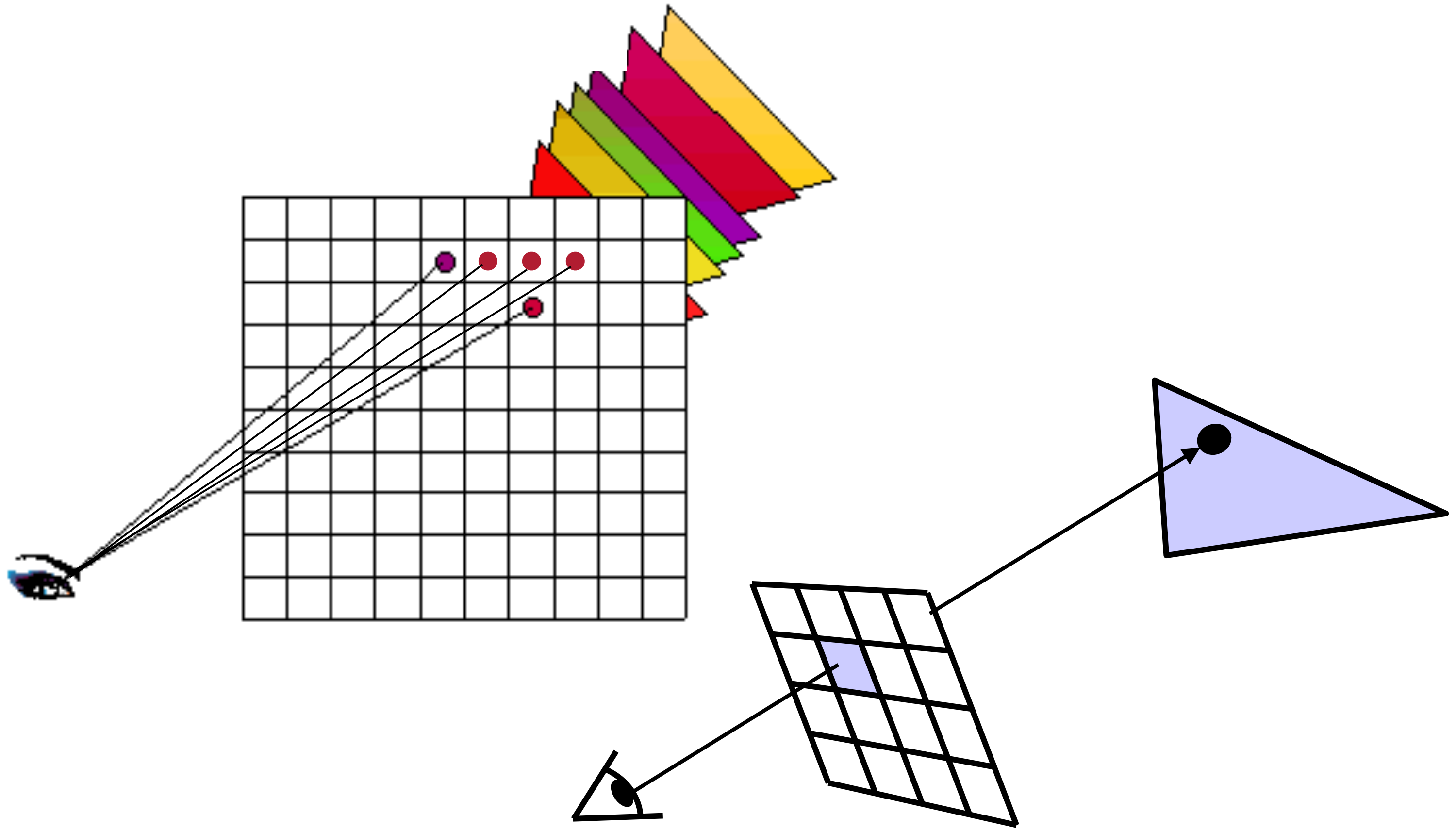
➢ Global Illumination

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

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

Raytracing can be considered an "inverse mapping" approach





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

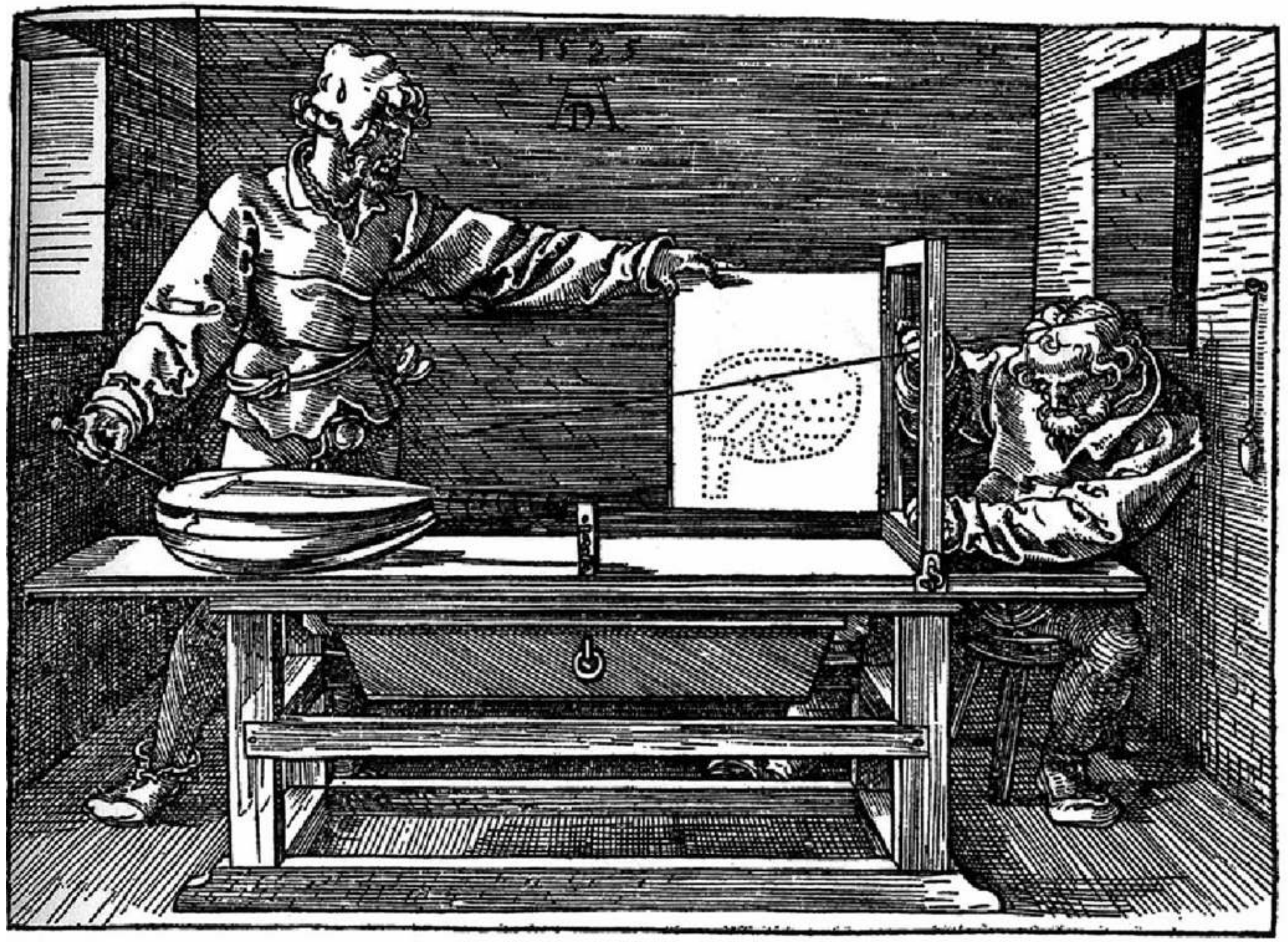
```
for each pixel:
  for each polygon:
    ...
```

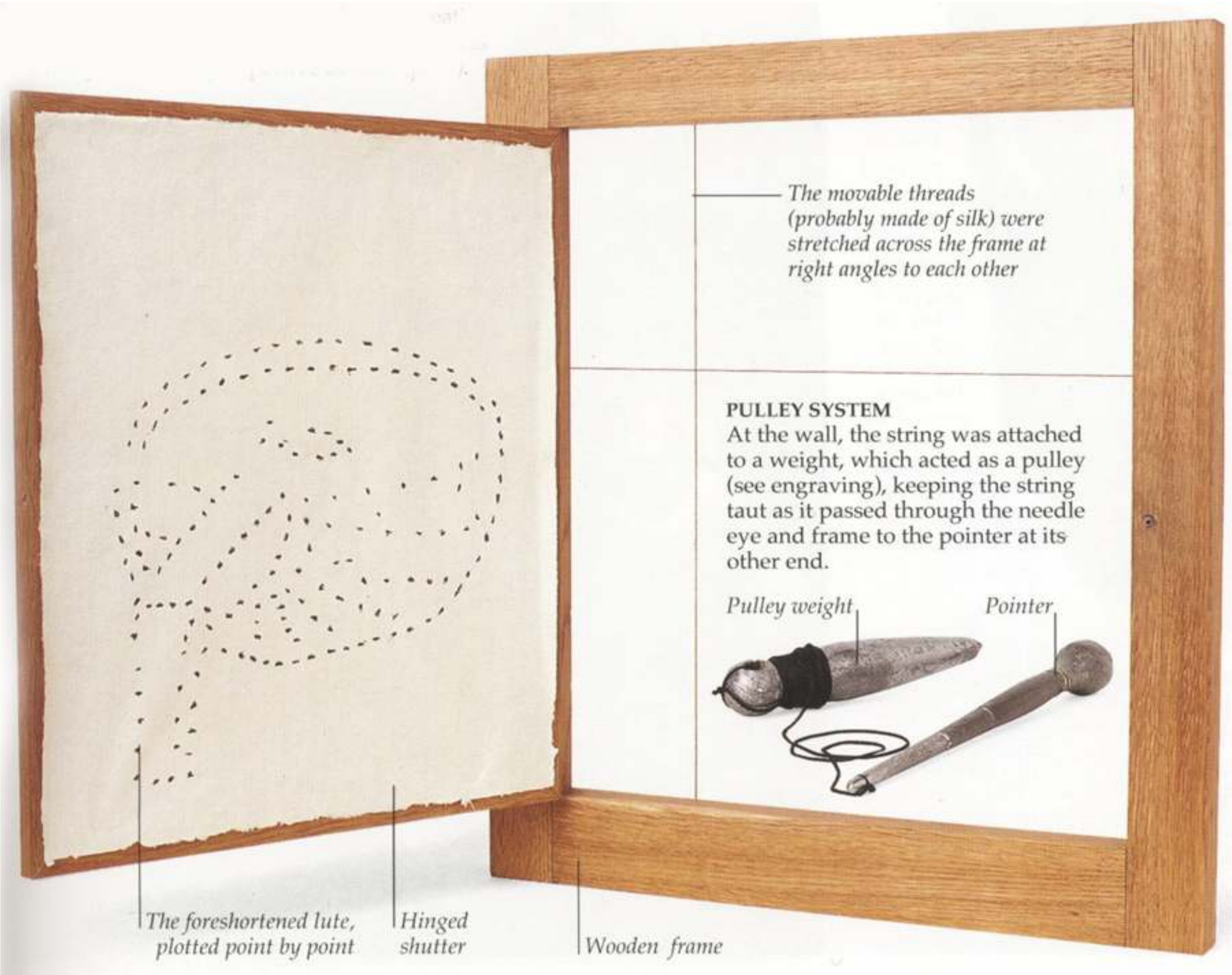
Probably the Oldest Conception of "Ray-Tracing"

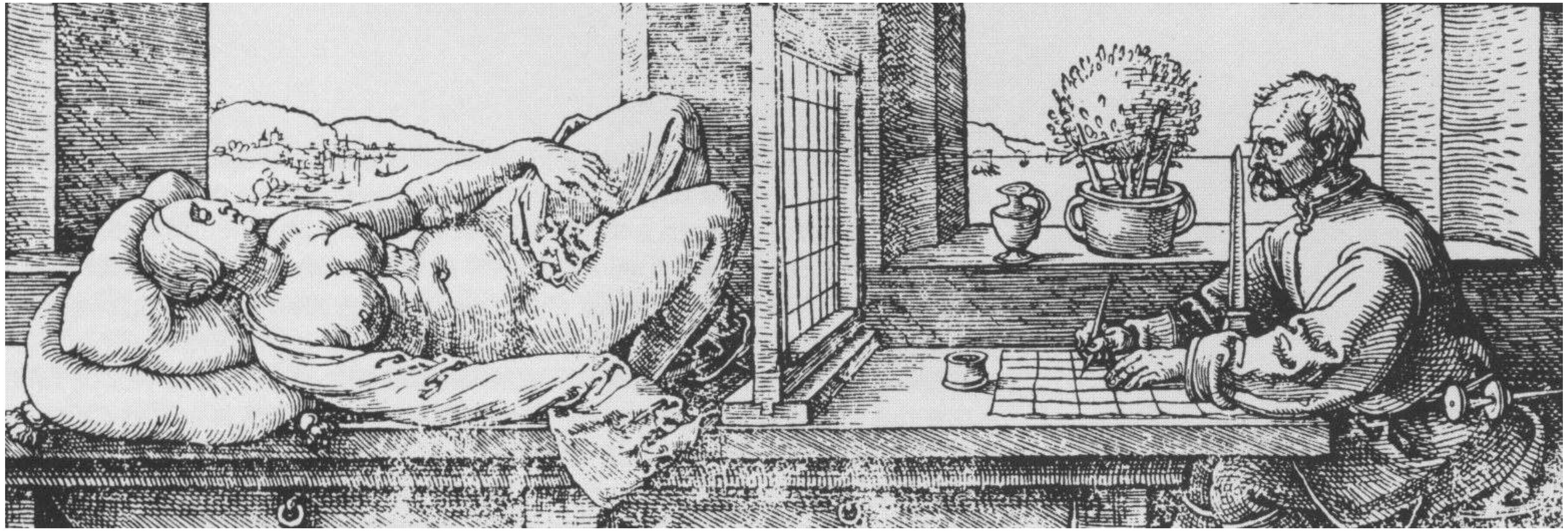
Emission theory
(conjectured by most Greek scientists
and held until around 1500 among Western scientists)

Albrecht Dürer's "Ray Casting Machines"

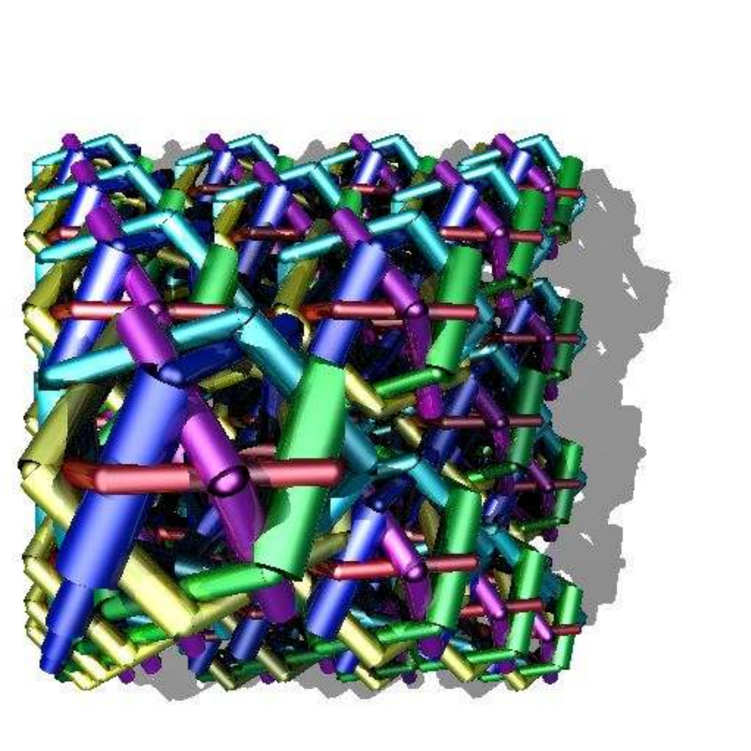
[16th century]

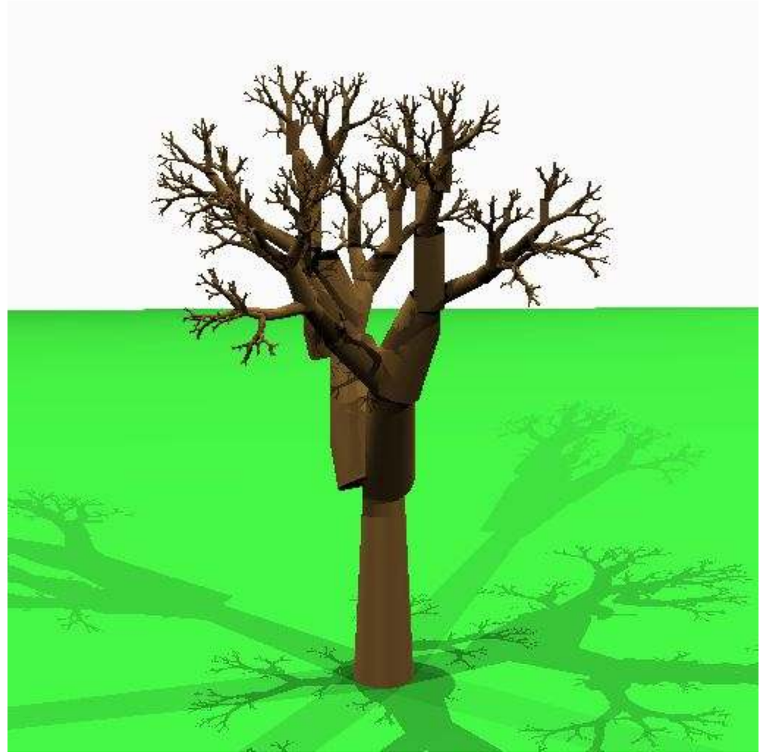


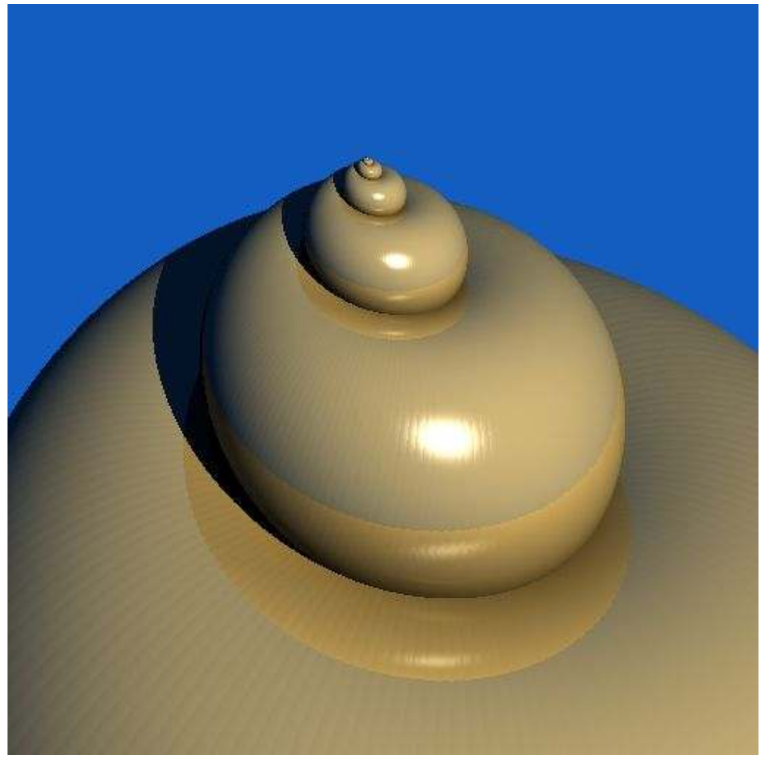


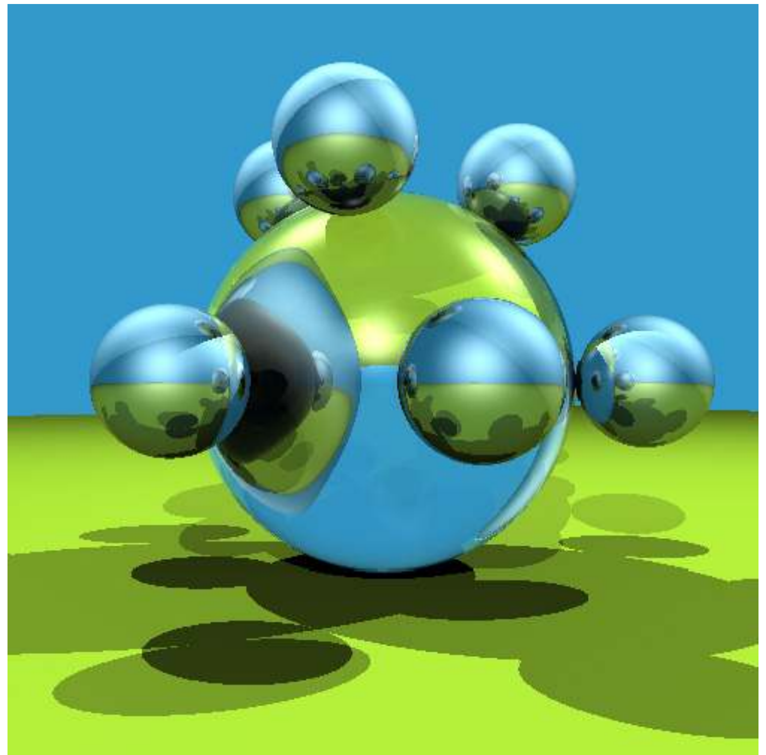


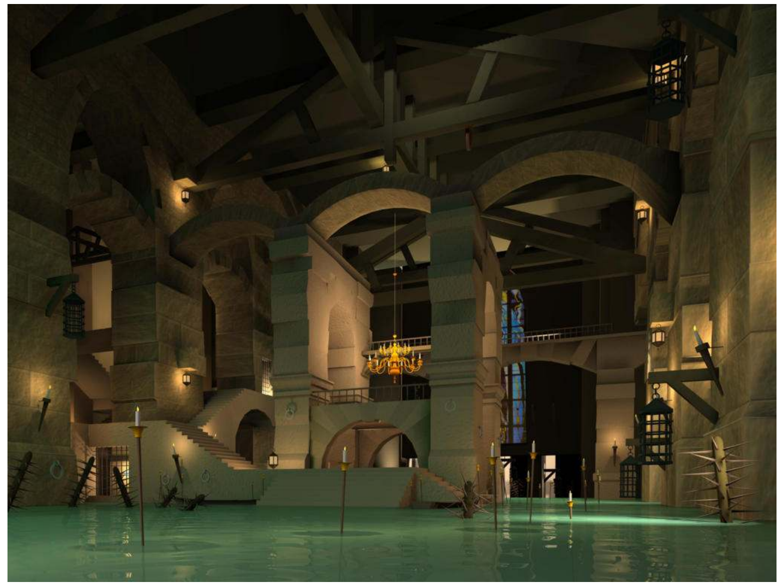
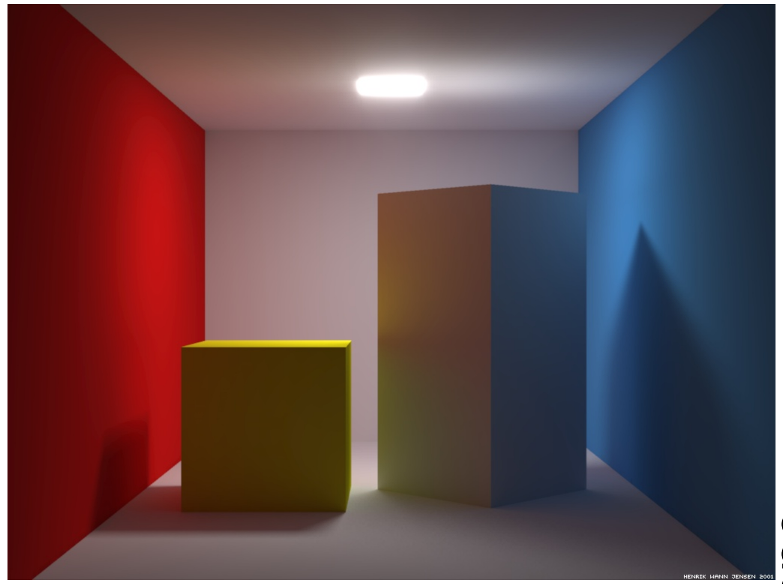
Examples of Ray-Traced Images

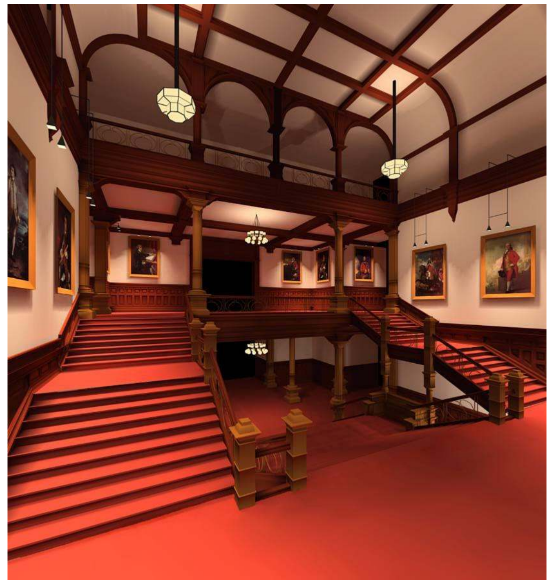


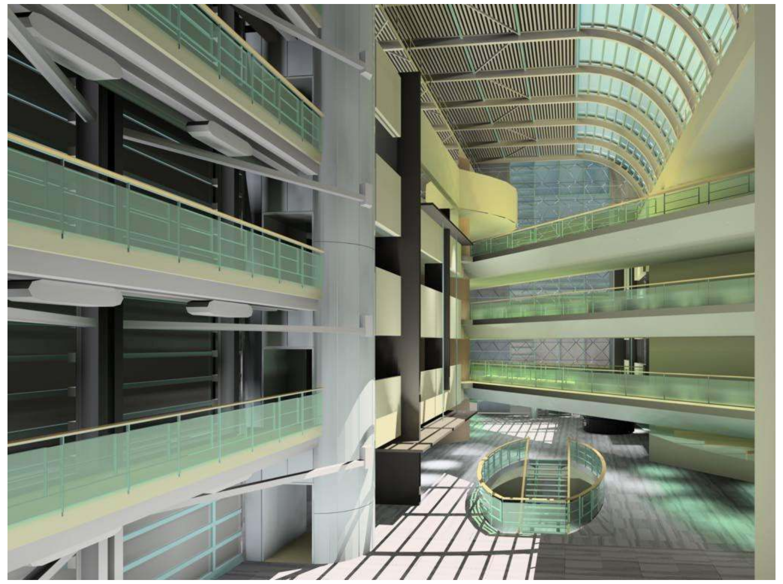






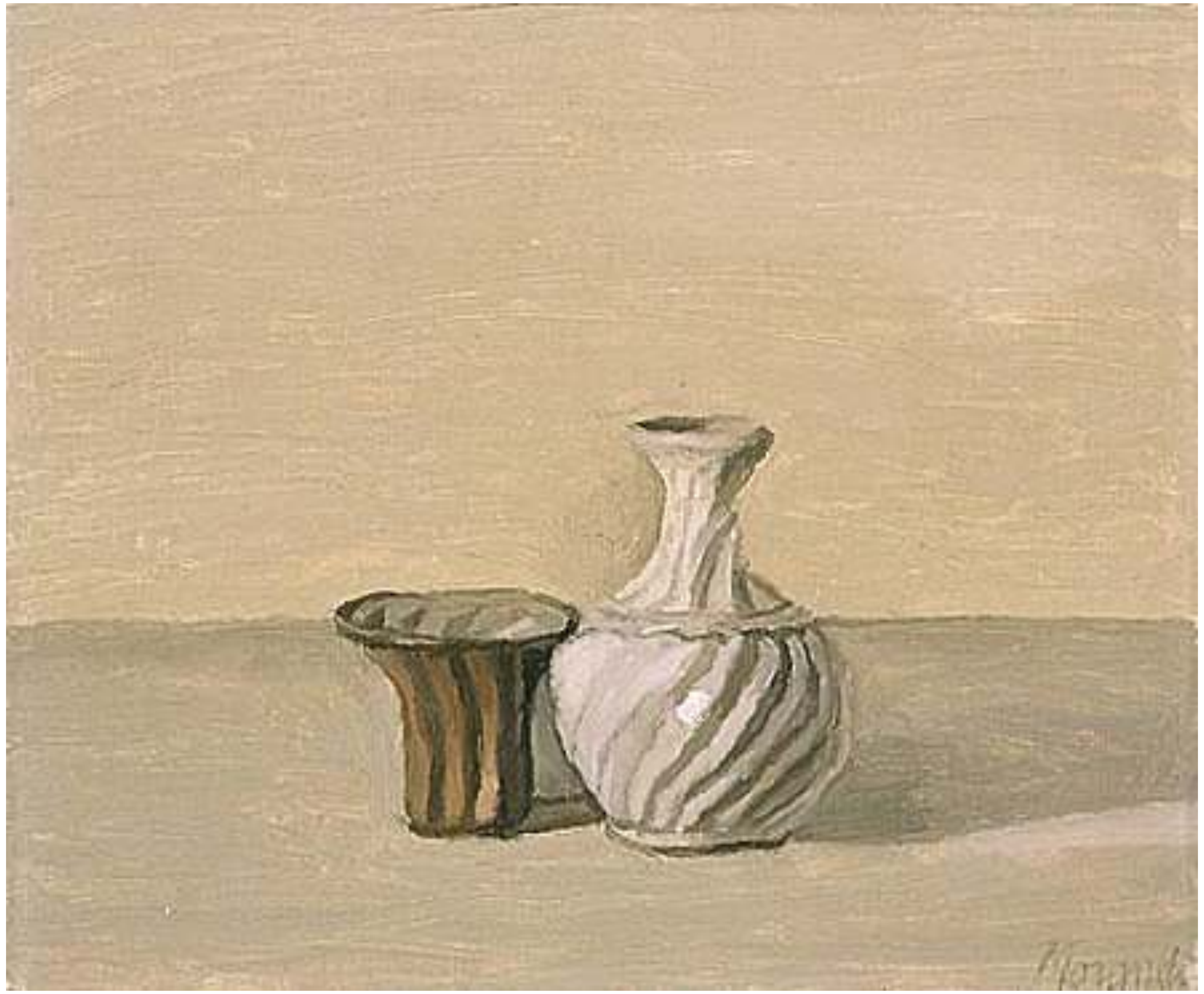




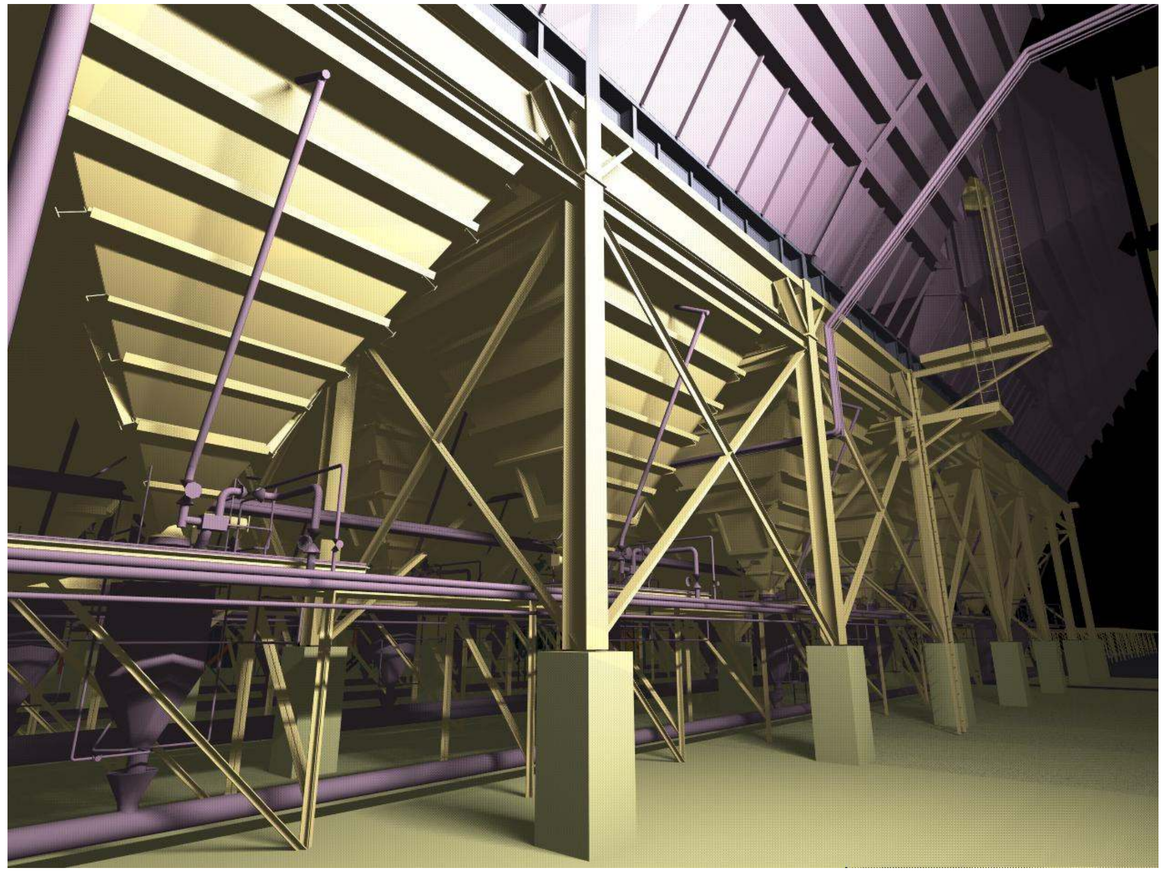






Jensen, Lightscape

Intermission: Giorgio Morandi



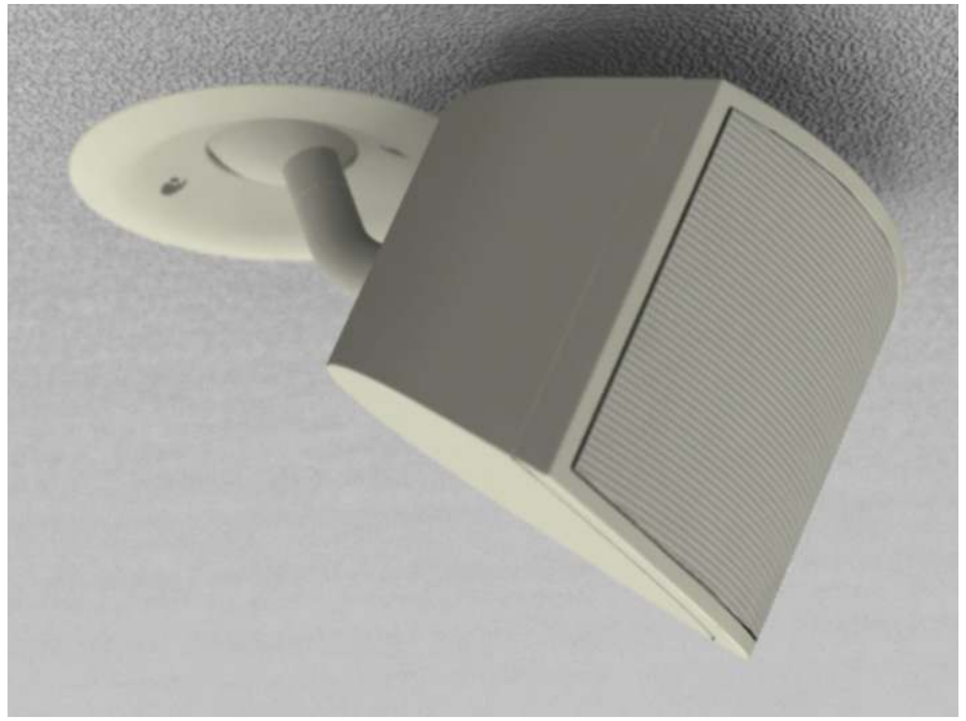


Ray Tracing in the Animation Industry

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?

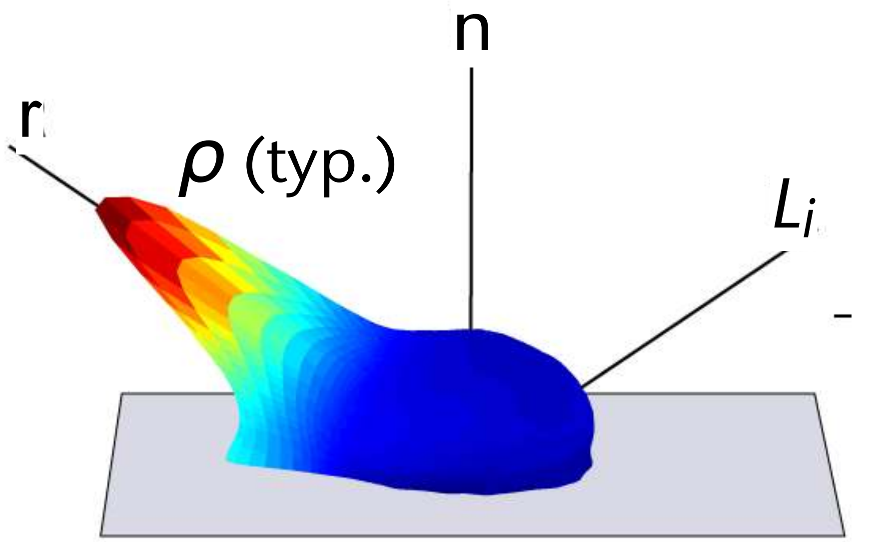


https://www.menti.com/86xyuy7f9e

The Rendering Equation

- 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

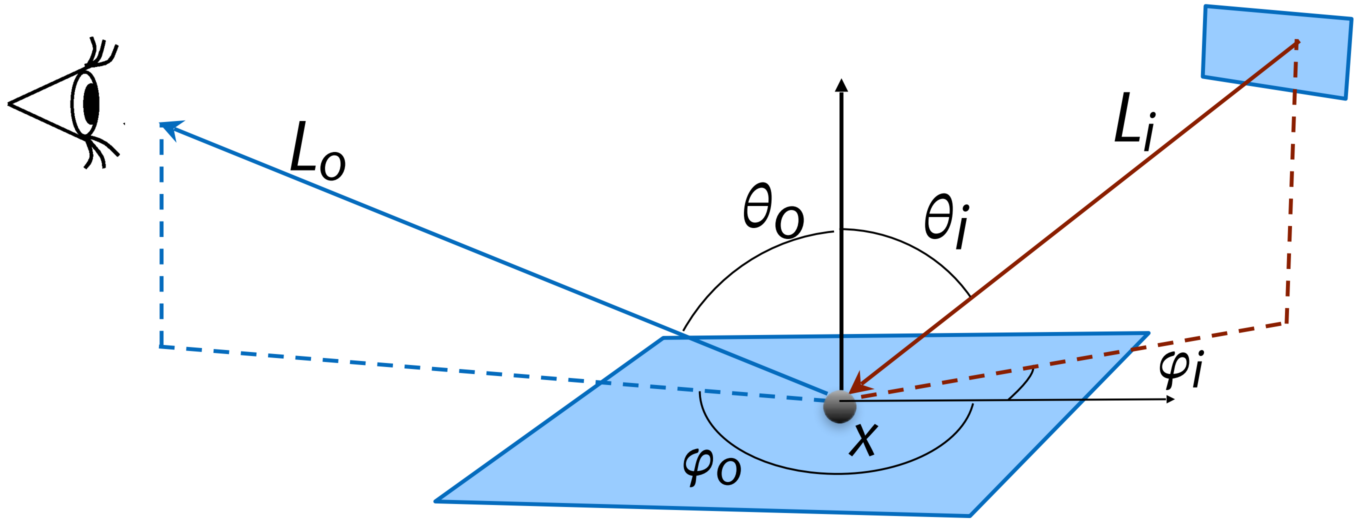
L_e = the "intensity" of light *emitted* (i.e., "produced") from x into direction ω_o

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

ρ = function of the reflectance coefficient = BRDF (to be def'd properly later)

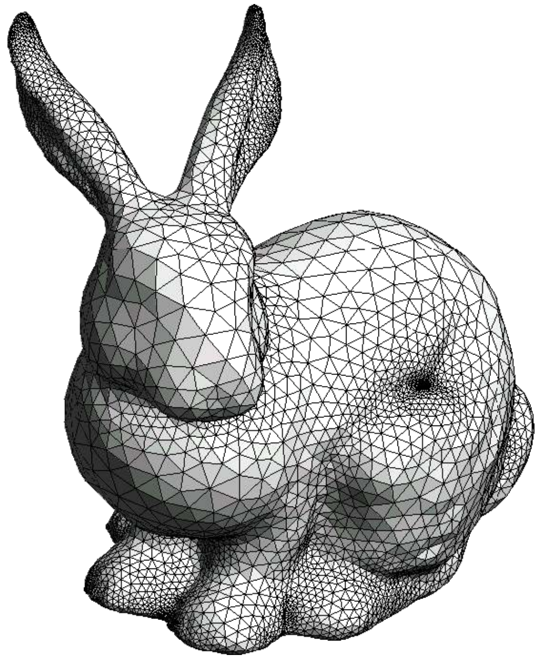
$\omega = (\theta,\varphi)$ = a direction (two polar angles)

Ω = hemisphere around the normal

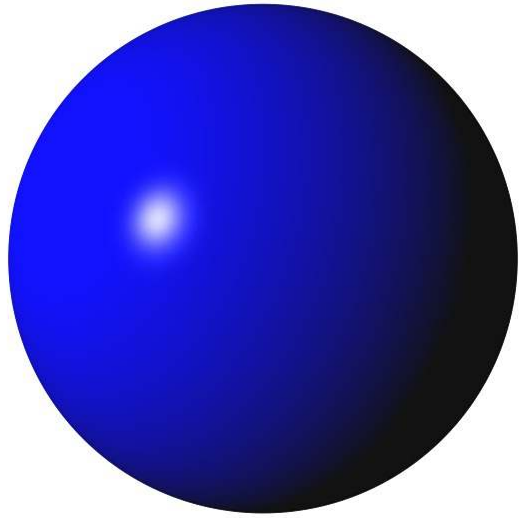


Output

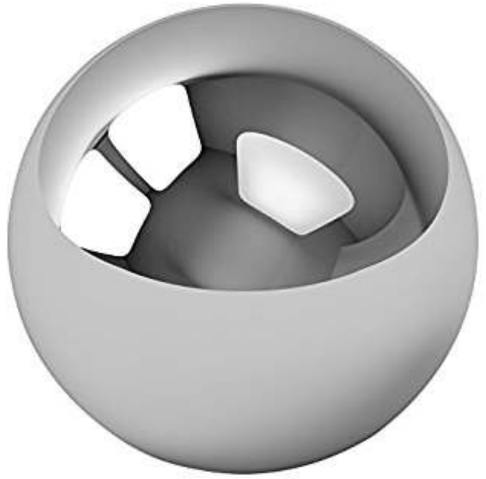
Inputs



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

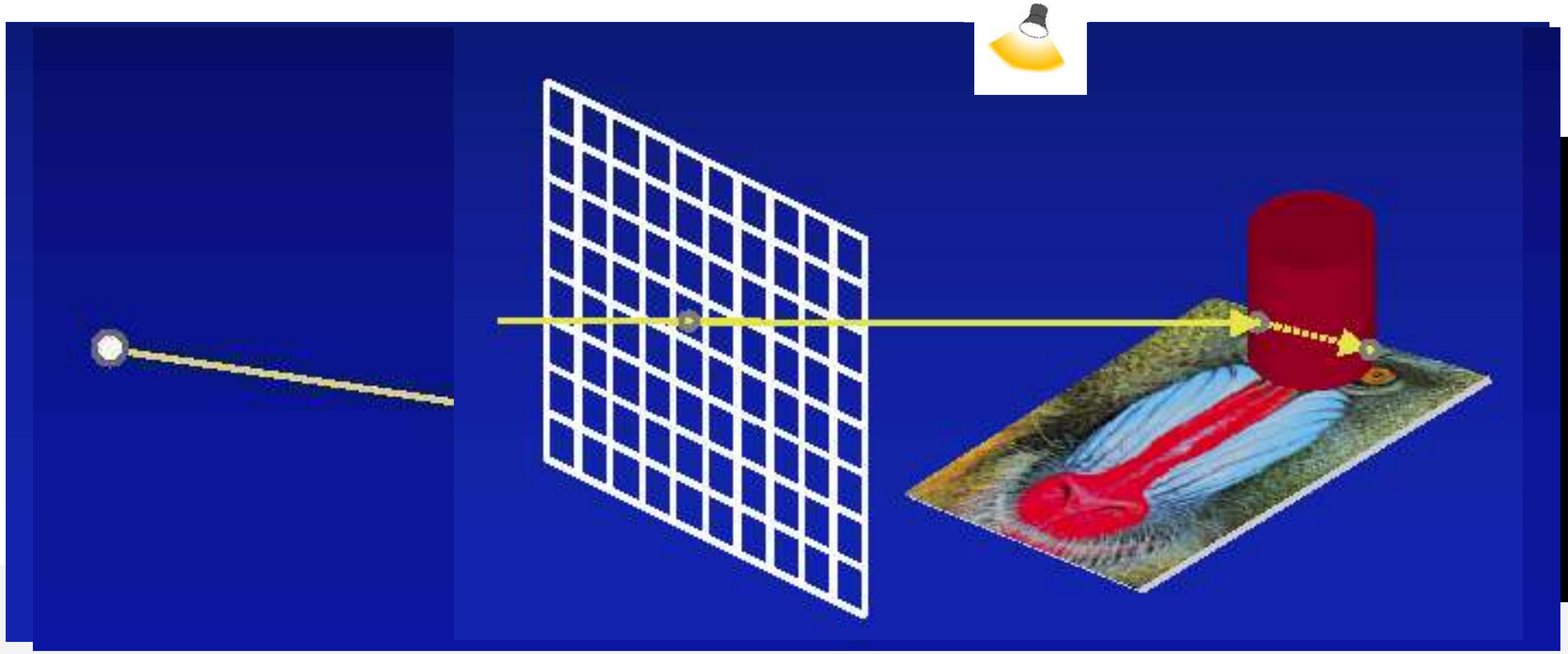
- Solving the rendering equation is impossible!
- Observation: the rendering equation is a recursive function
- Consequently, a number of approximation methods have been developed that are based on the idea of following rays:
  - Ray tracing (Whitted, Siggraph 1980, "An Improved Illumination Model for 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

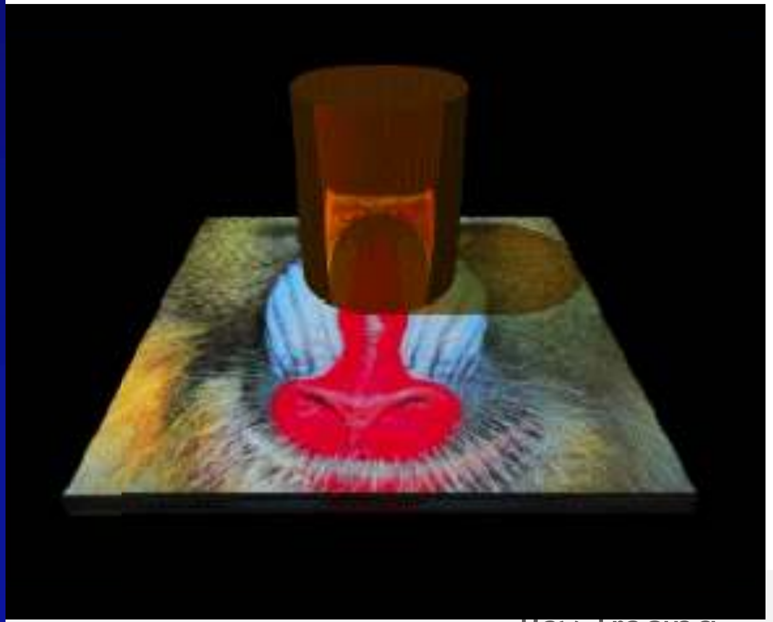
Turner Whitted,
Microsoft Research

The Simple "Whitted-Style" Ray-Tracing

- Synthetic camera = viewpoint + image plane in world space

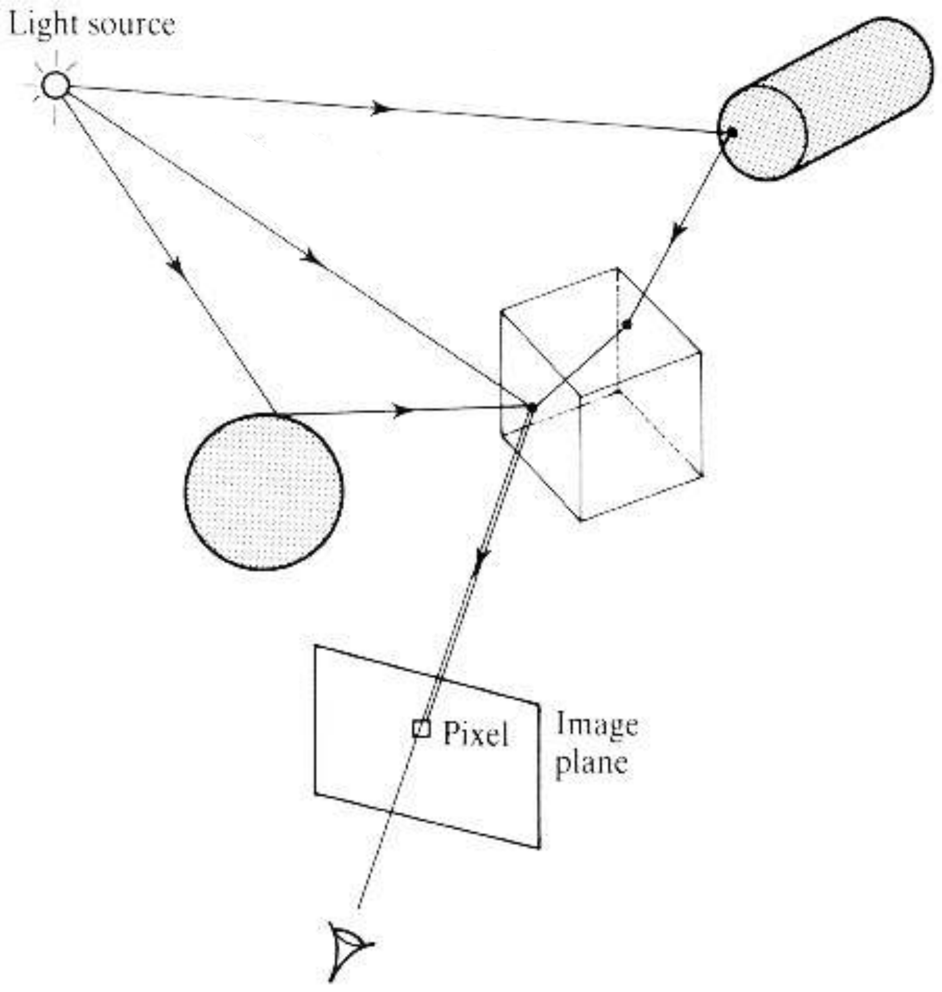
1. Shoot rays from camera through every pixel into scene (primary rays)
2. Compute the first hit with *any* of the objects in scene
3. From there, shoot rays to all light sources (*shadow feelers*)
4. If a shadow feeler hits another obj → point is in shadow w.r.t. that light source. Otherwise, evaluate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

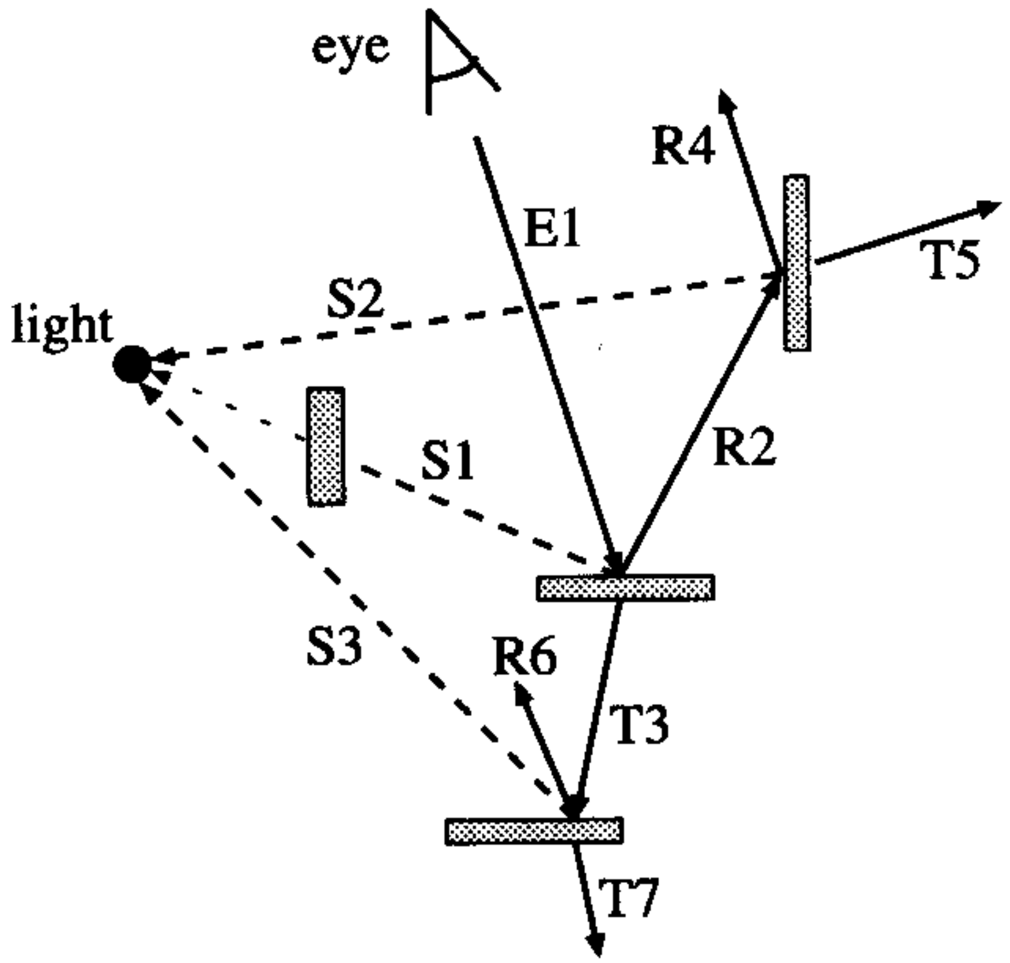


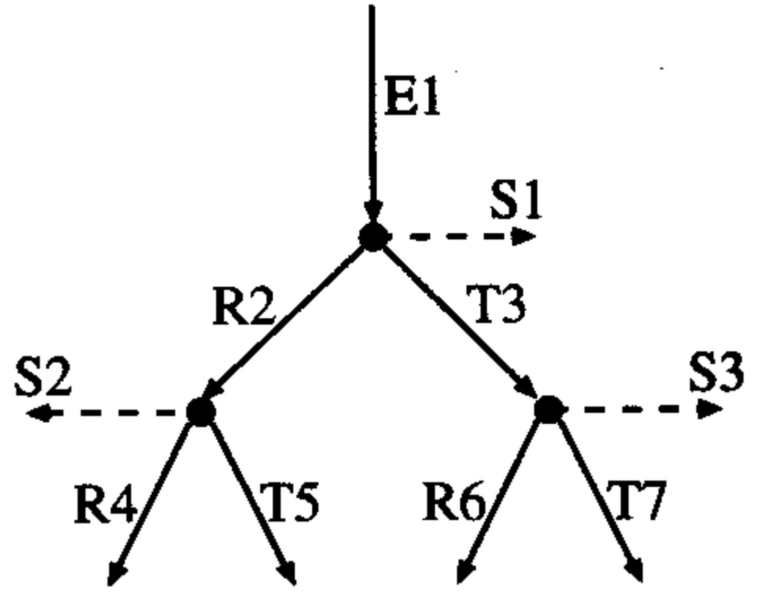


The Ray Tree

- 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:

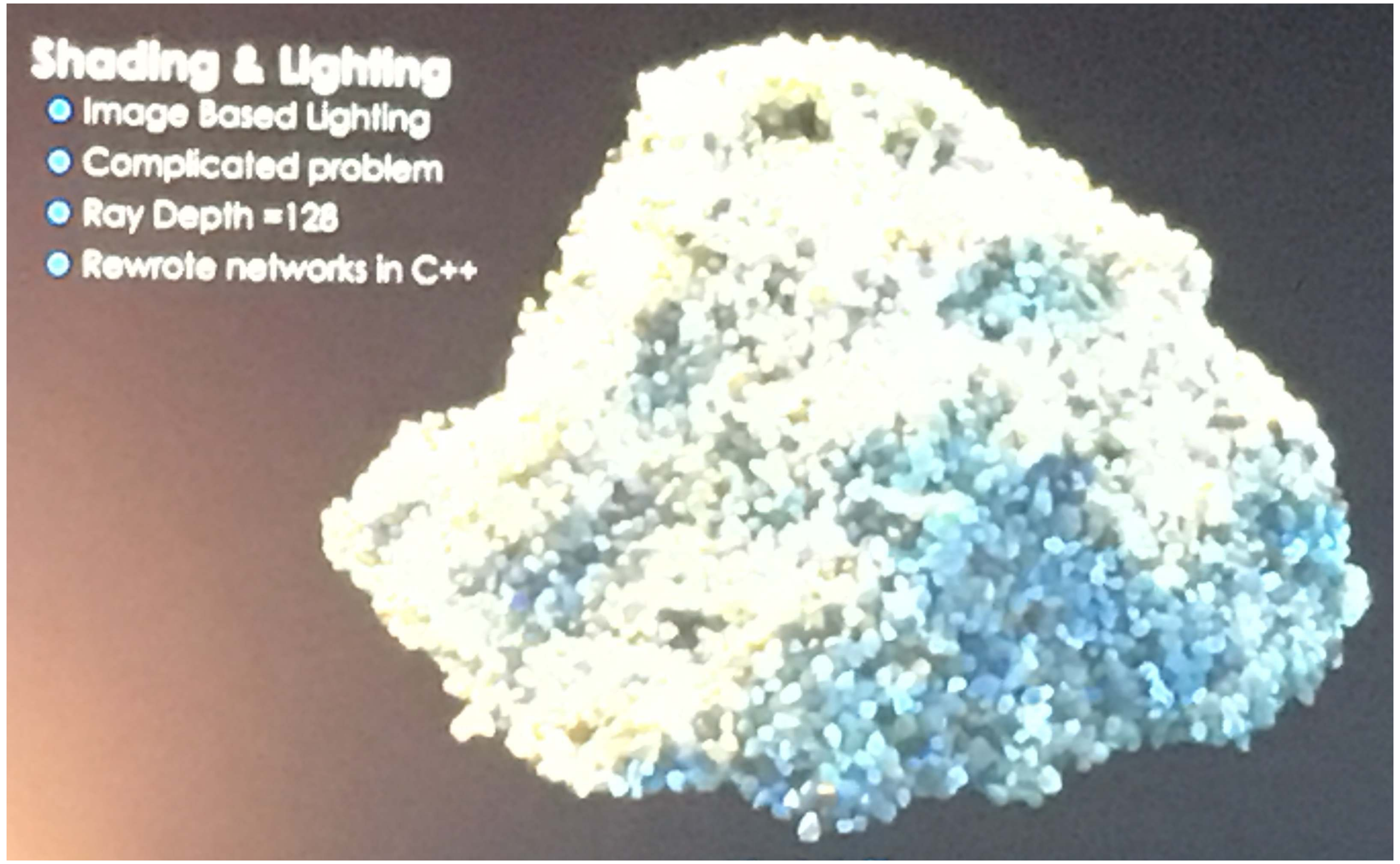






E1 = primary ray
Ri = reflected rays
Ti = transmitted rays
Si = shadow rays

- 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)

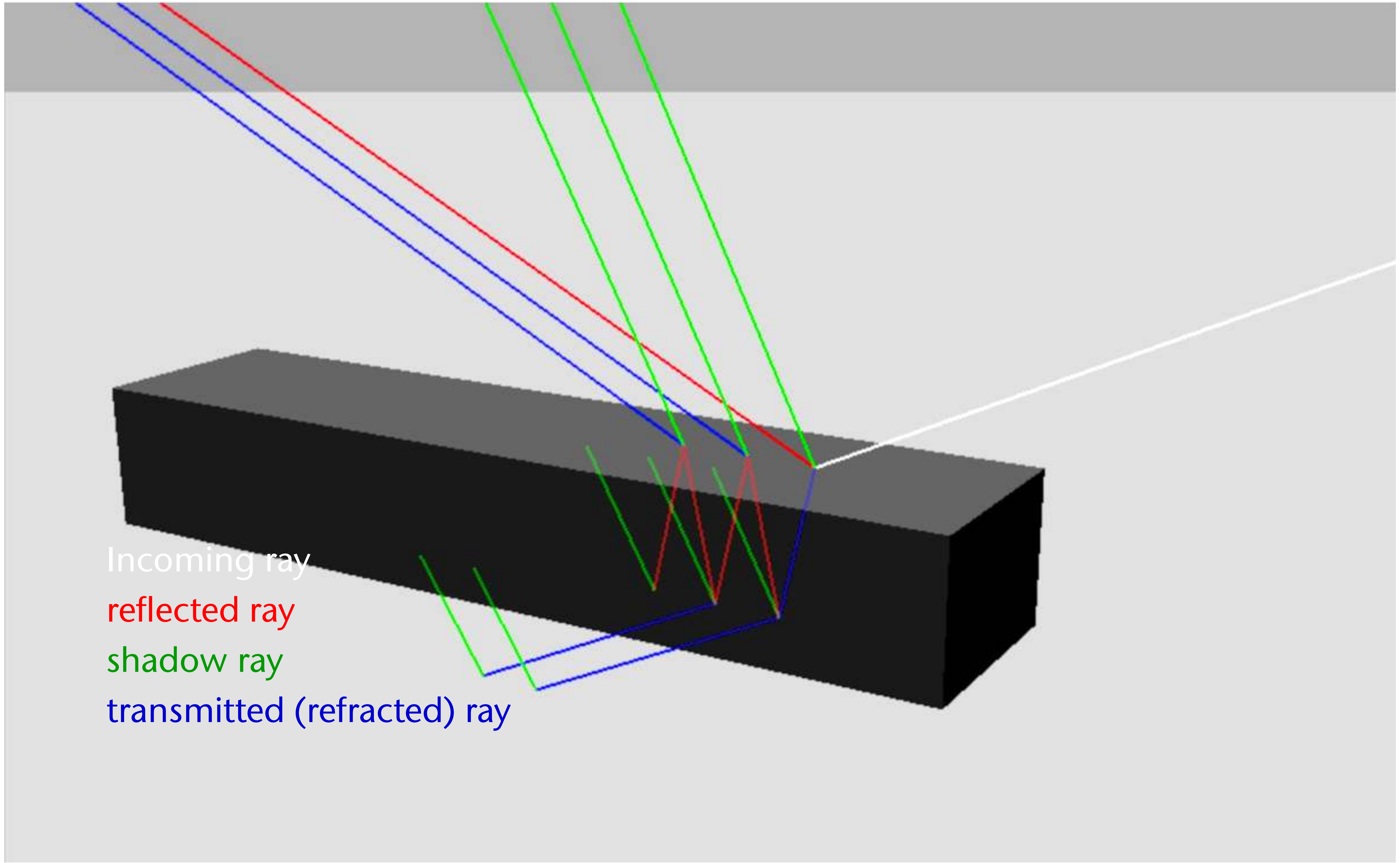


Max ray depth = 128 (!)

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

Excerpt from "Piper", Pixar 2017

Visualizing the ray tree can be very helpful for debugging



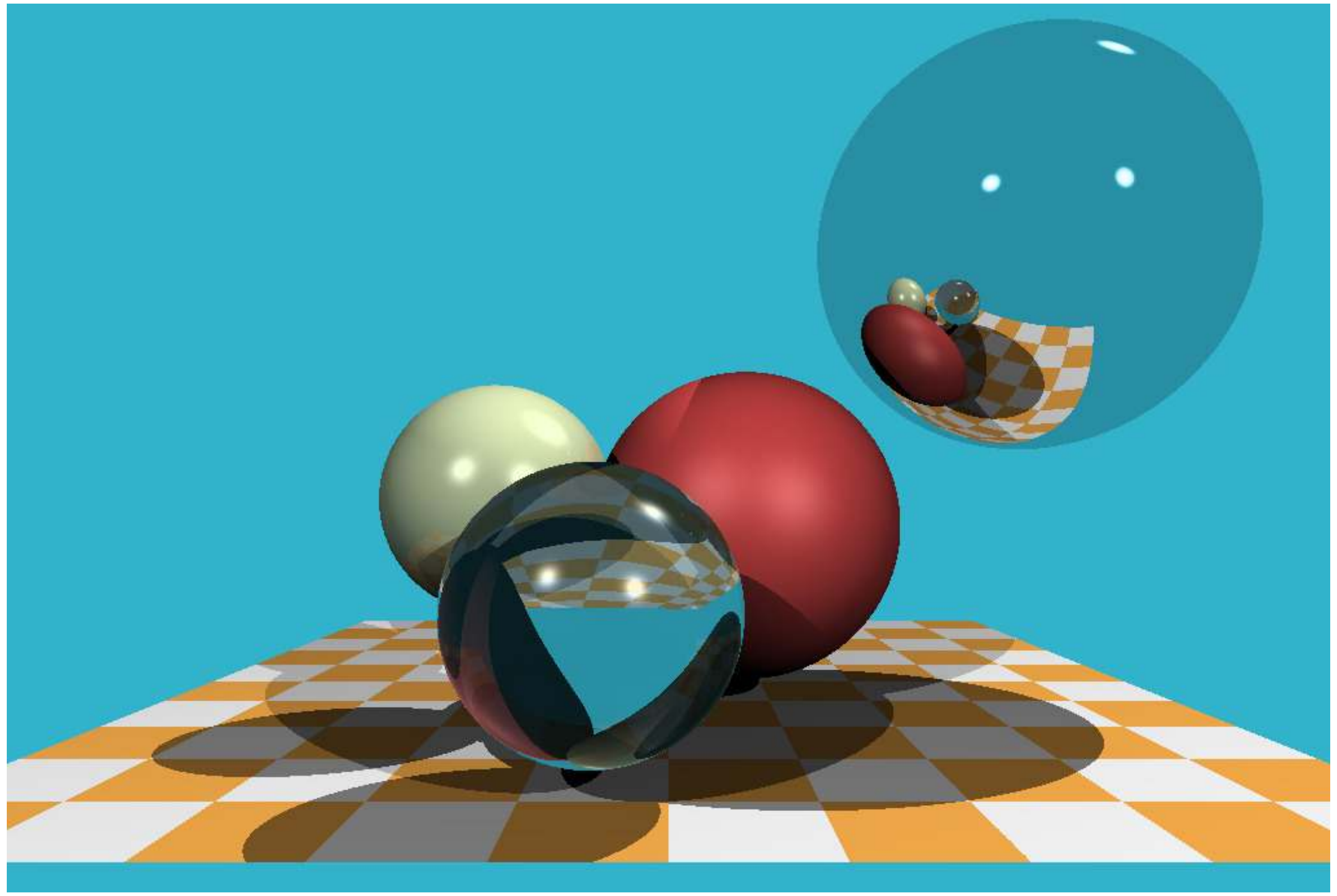
A Little Bit of Ray-Tracing Folklore

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!*/
```

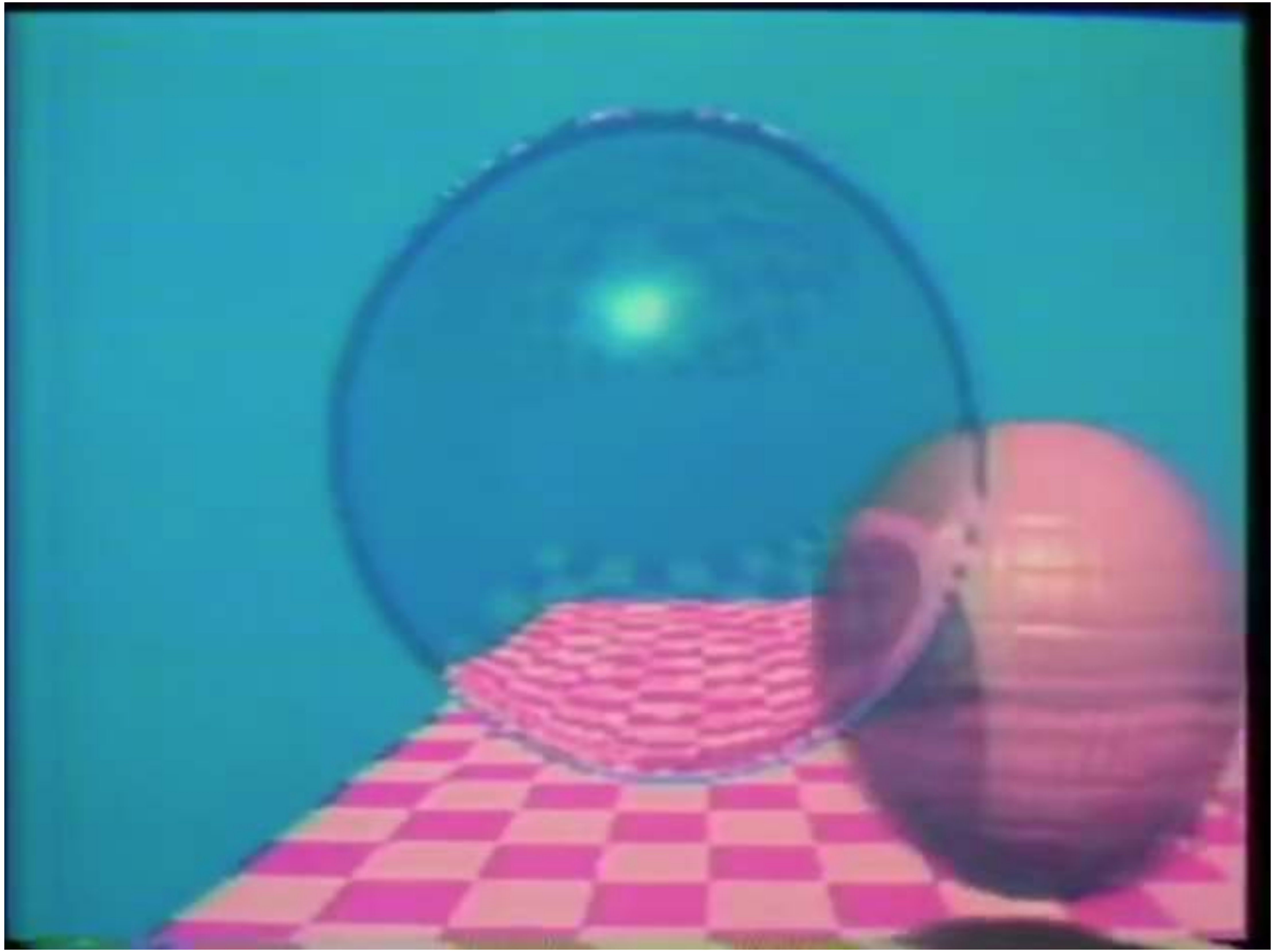
(Also won the *International Obfuscated C Code Contest)!*

Another ray tracer in 256 lines of C++:



https://github.com/ssloy/tinyraytracer

The First Ray-Traced Movie



Turner Whitted, 1978? 1980?

Basic Definition of Terminology

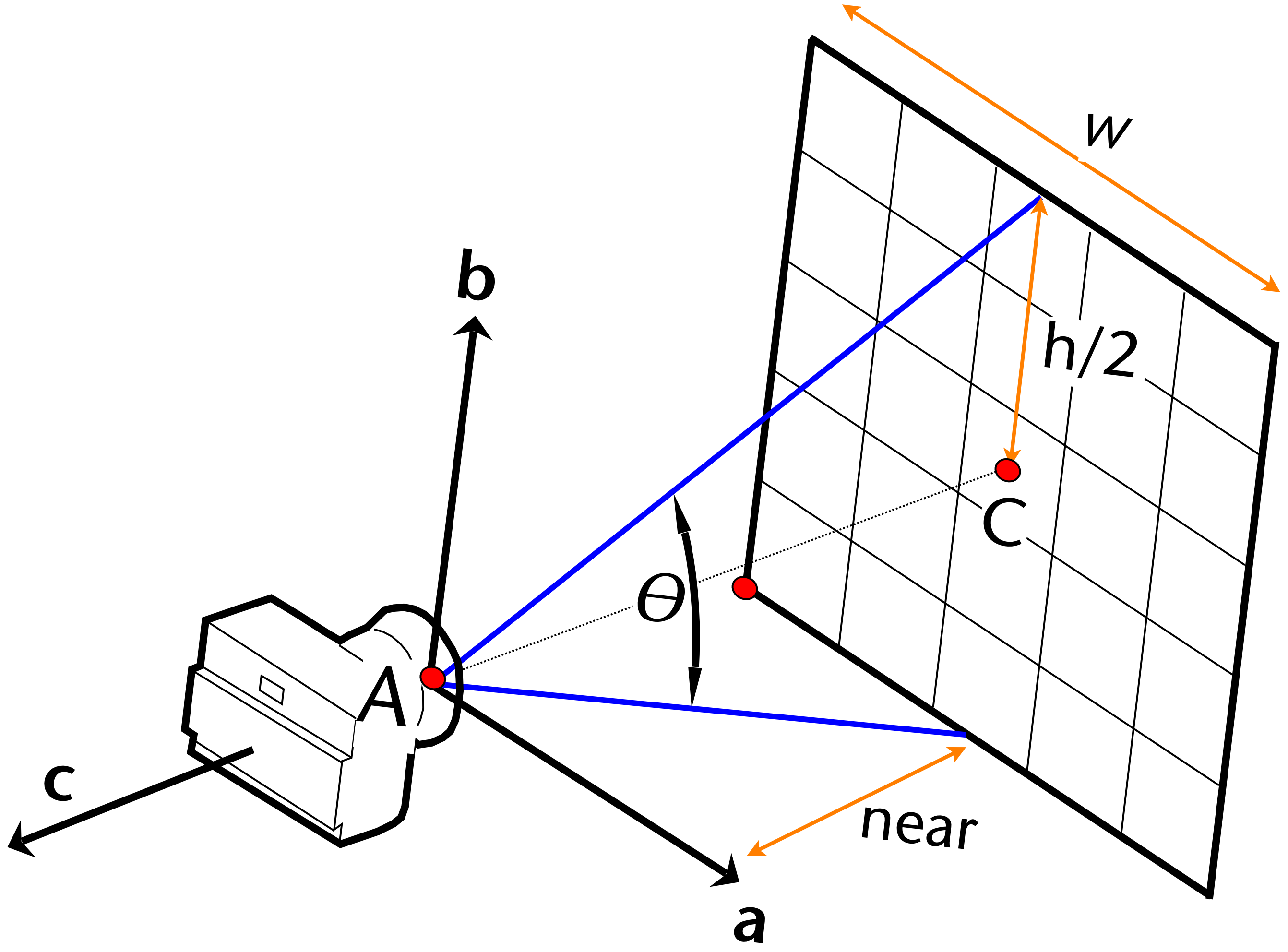
- Ray tracing = geometric algorithm to compute intersections of rays with the scene (aka. ray-based visibility)
- Path tracing = algorithm to compute global illumination by shooting rays in all kinds of (random) directions (aka. random sampling, aka. Monte Carlo integration)

Basic Ingredients Needed for Ray-Tracing

1. Primary rays $\longrightarrow$ camera model
2. Secondary rays and shadow feelers $\longrightarrow$ (geometric) optics laws
3. Combining all incoming light into "one" outgoing light $\longrightarrow$ lighting models

- Note: shadow feelers are special types of rays, are usually handled special
  - So, we have 3 types of rays

A Simple Camera Model (Ideal Pin-Hole Camera)

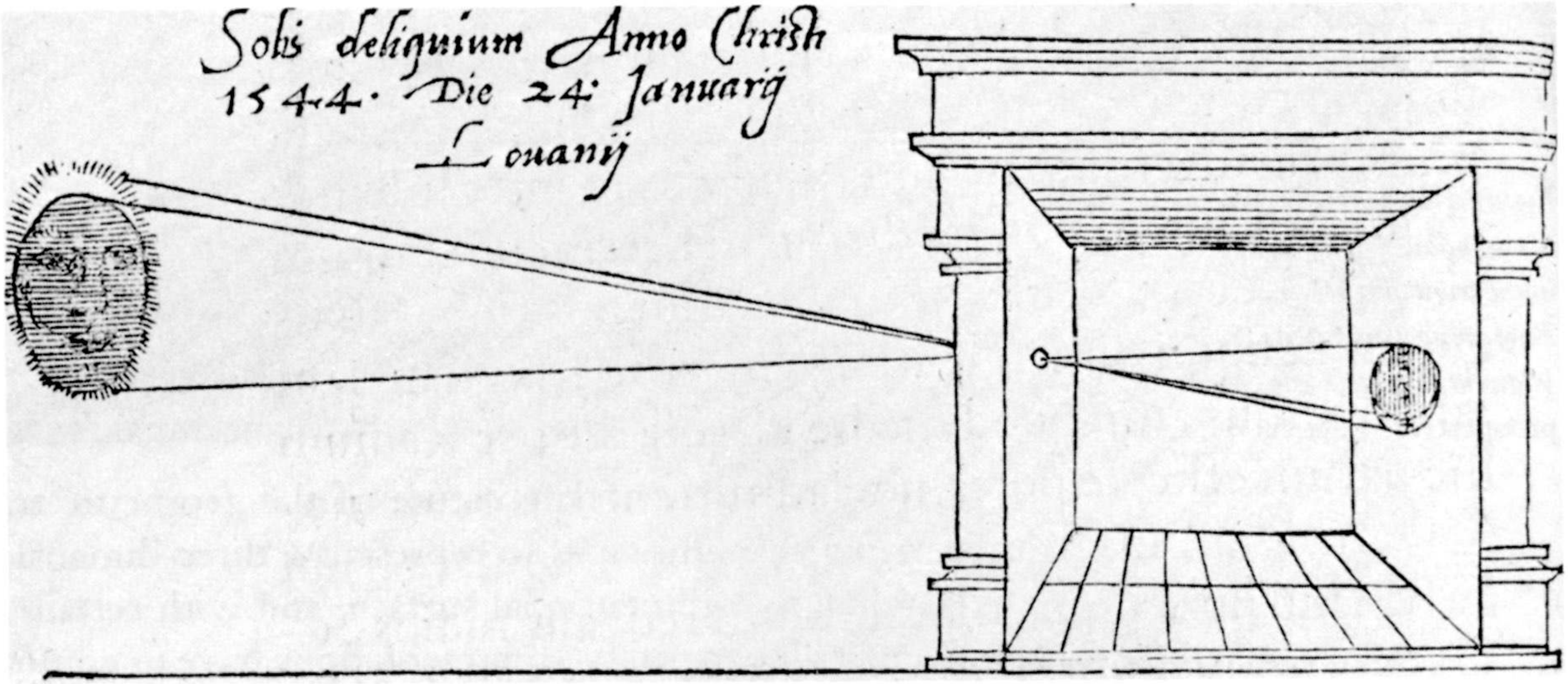


$$\frac{h}{2} = \text{near} \cdot \tan\frac{\theta}{2}$$

The main loop of ray-tracers

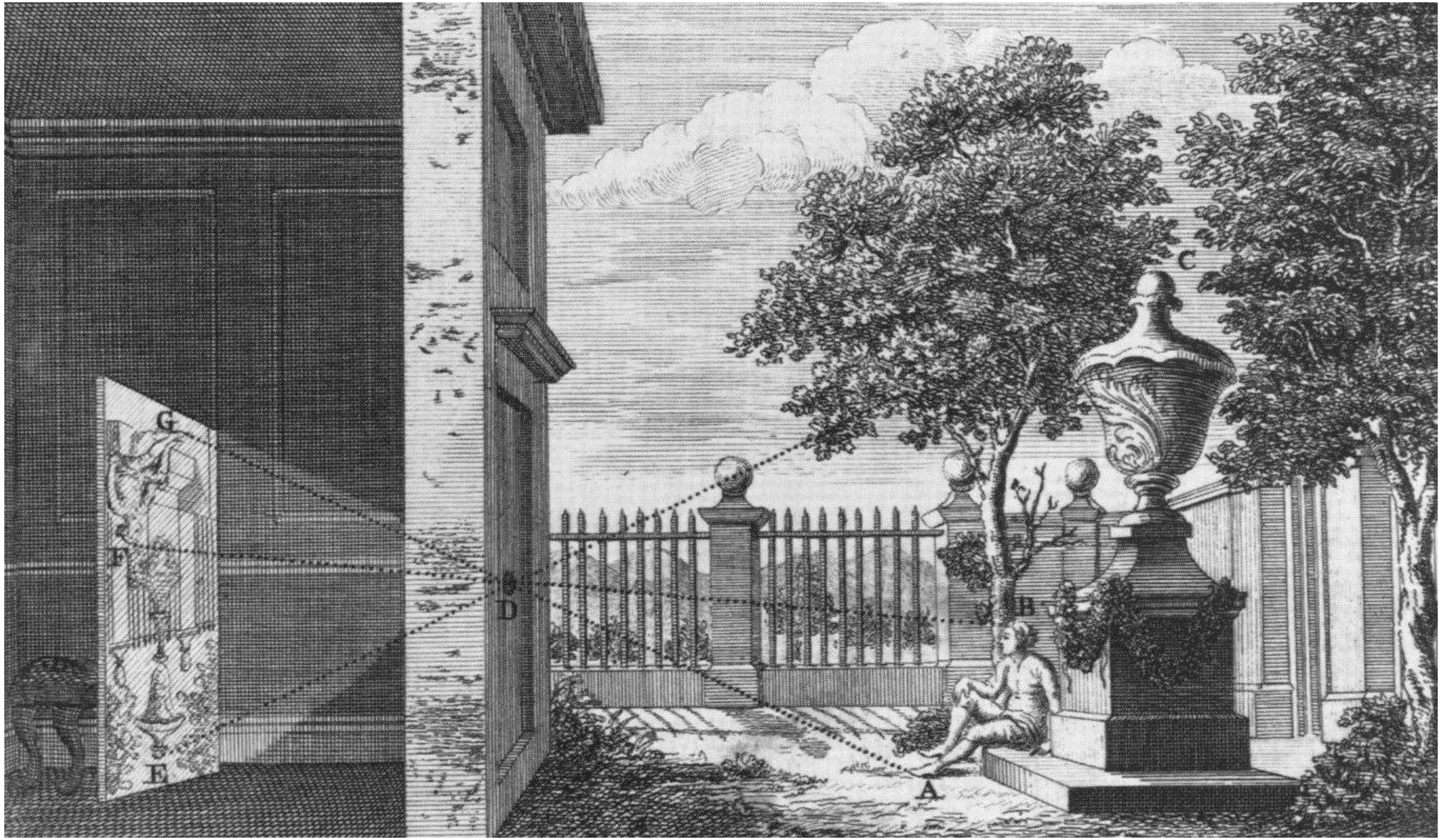
```
def gen_prim_rays( vec3 a, vec3 b,
                   vec3 A, vec3 C ):
  for i = 0 .. hor_res:
    for j = 0 .. vert_res:
      ray.from = A
      s = (i/hor_res - 0.5) * h
      t = (j/vert_res - 0.5) * w
      ray.at = C + s*a + t*b
      vec3 color = traceRay( 0, ray )
      putPixel( x, y, color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

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?

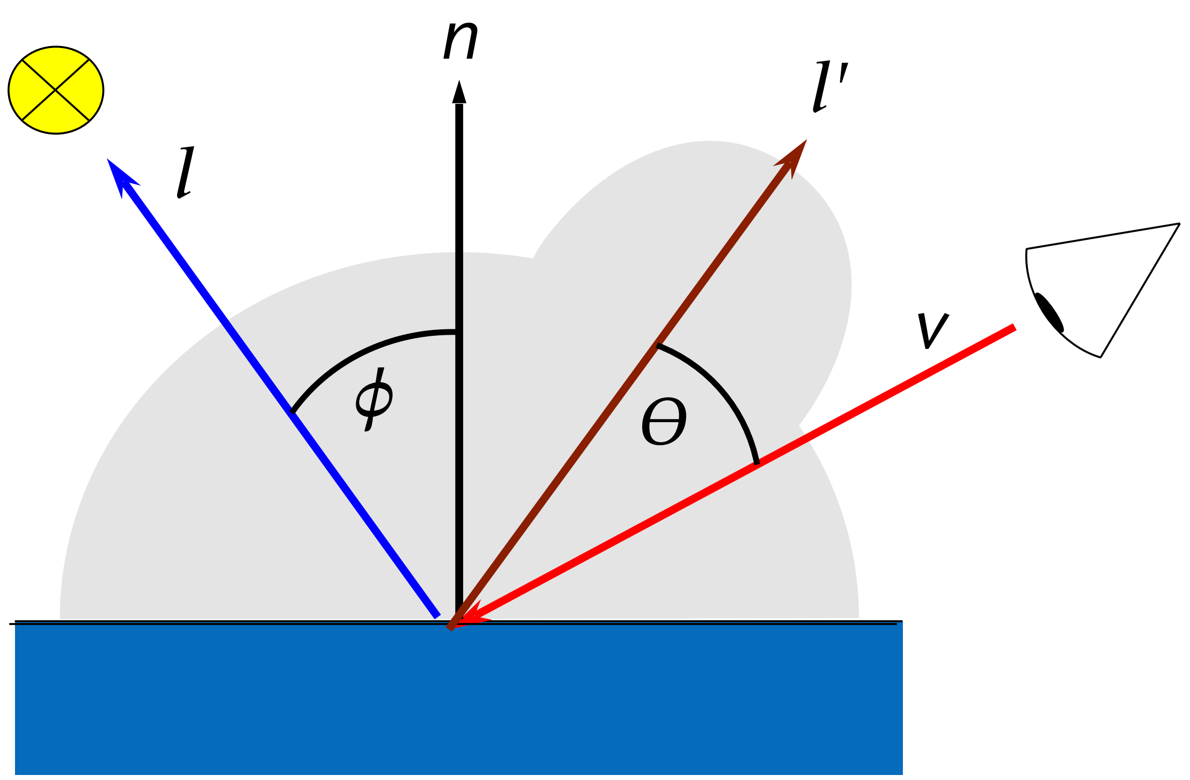
A Local Lighting Model

- 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



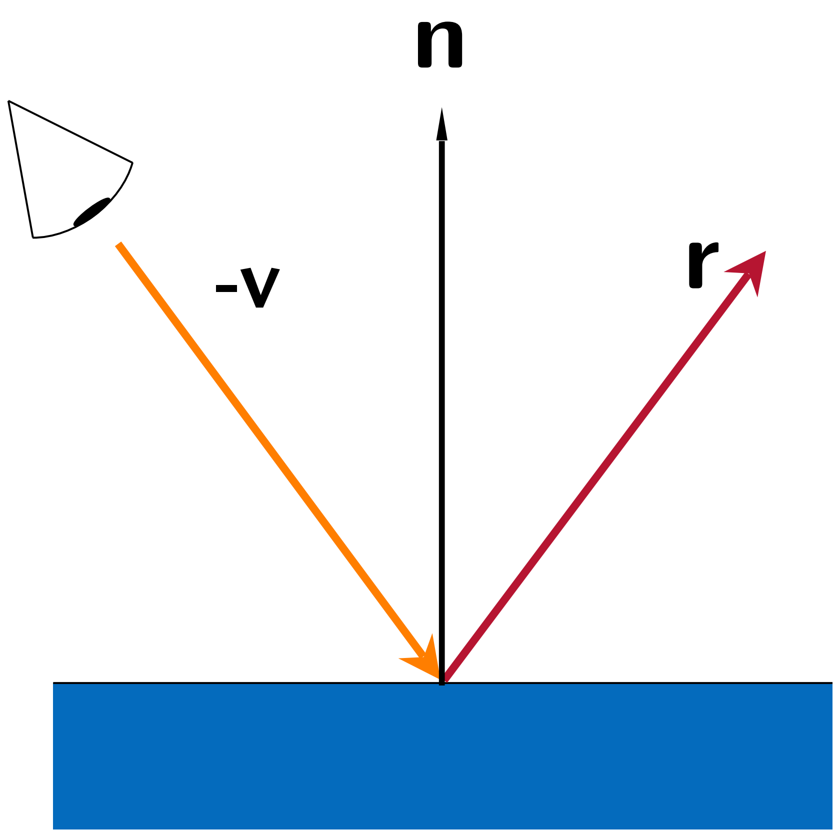
- Of course, we add a light source only, if it is visible!

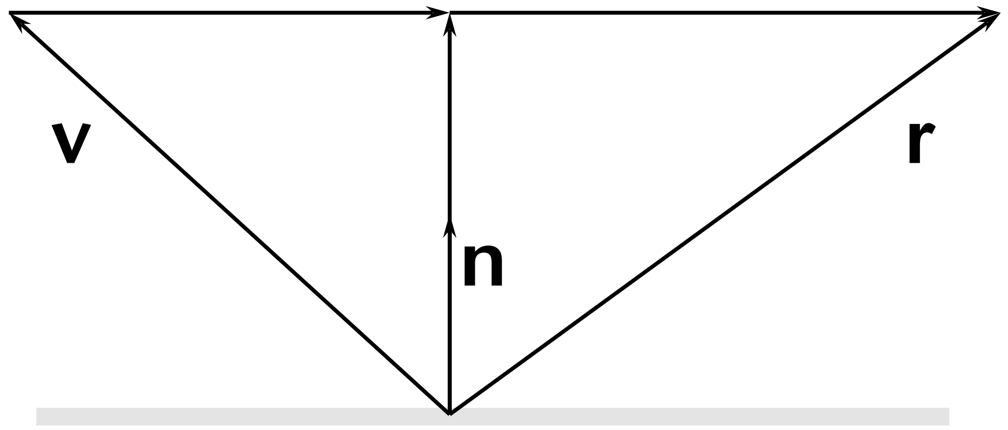
Generation of Secondary Rays

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

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

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





Specular Reflection

- 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

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

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

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

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

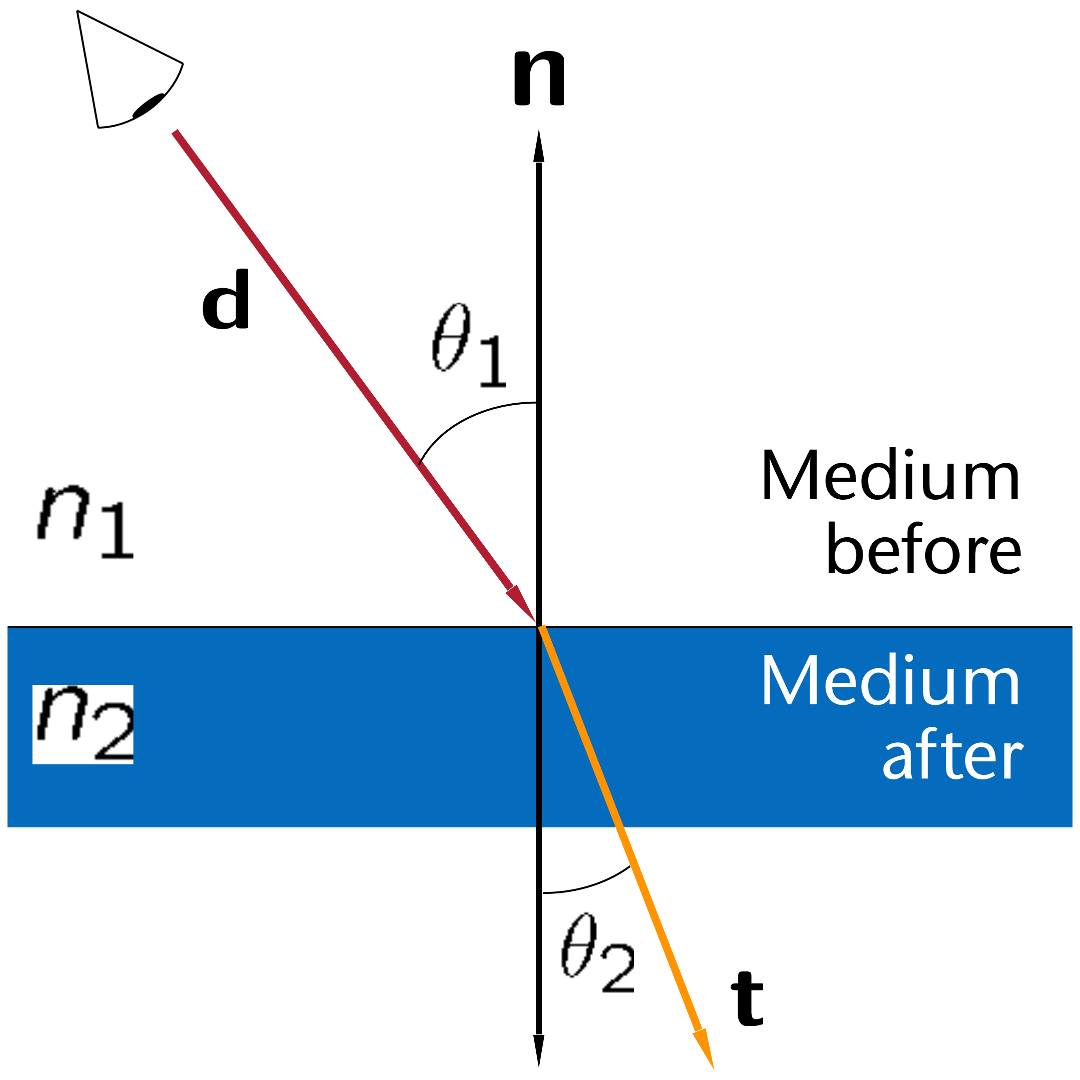
$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

- Computation of the refracted ray:

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

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

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



Typical indices of refraction (IOR)

Air	Water	Glass	Diamond
1.0	1.33	1.5 - 1.7	2.4

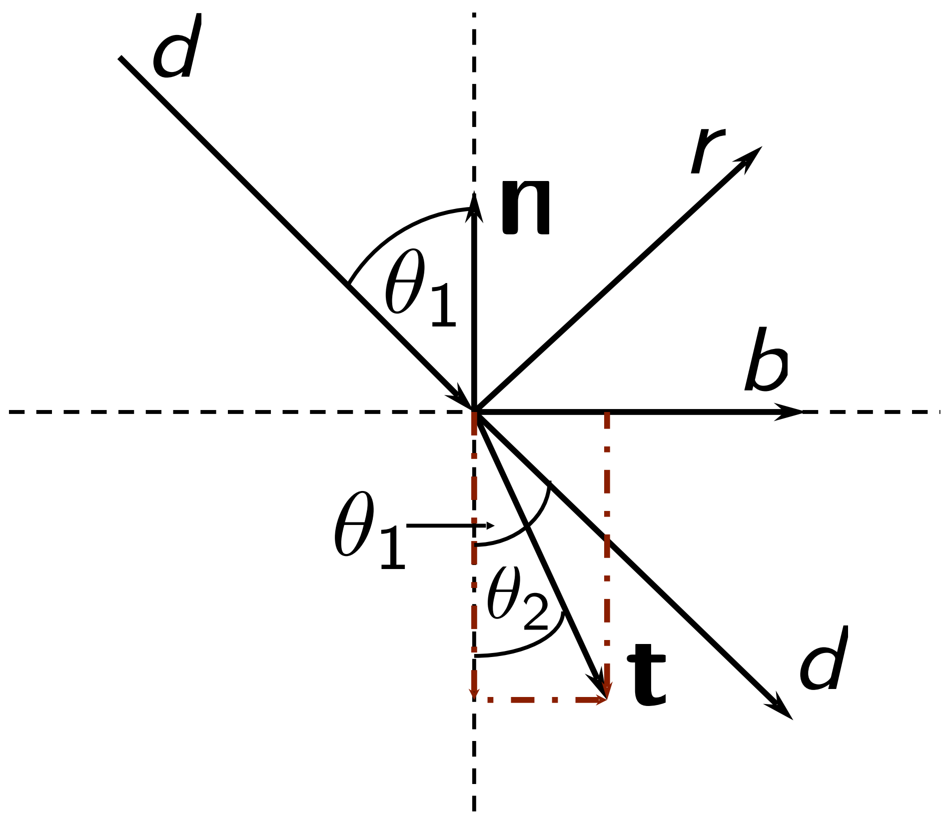
FYI: Derivation of the Equation on the Previous Slide

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

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

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

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



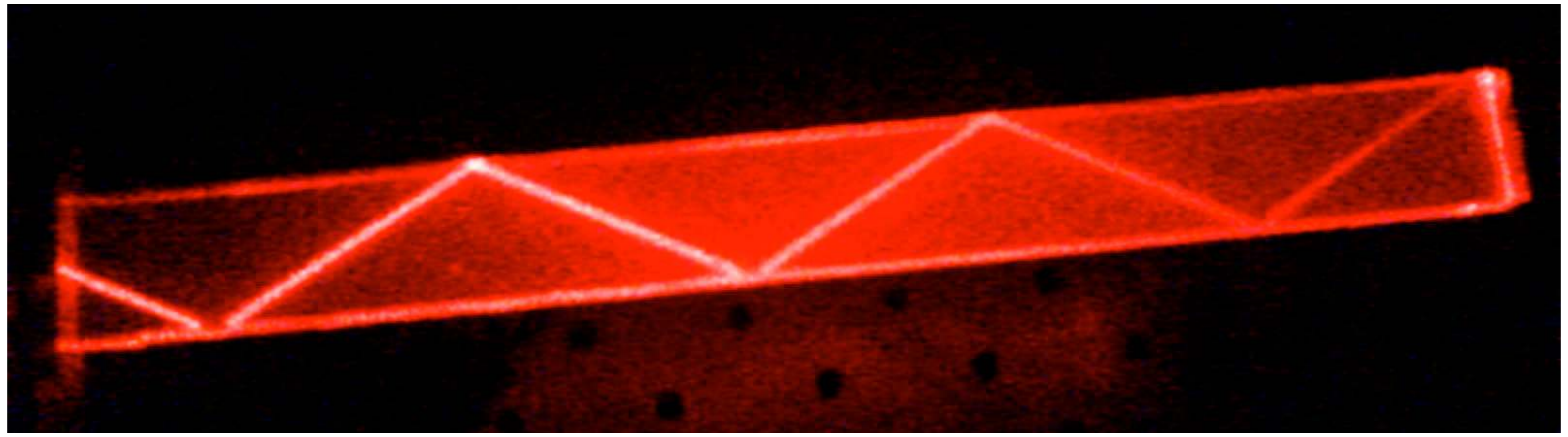
$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2}$$
$$\cos\theta_1 = \mathbf{n} \cdot (-\mathbf{d})$$

$\cos\theta_2$ ausrechnen:

$$\sin\theta_2 = \frac{n_1}{n_2} \sin\theta_1$$
$$\sin^2 + \cos^2 = 1$$
$$\cos^2\theta_2 = 1 - (\frac{n_1}{n_2} \sin\theta_1)^2$$

- Total reflection occurs, whenever the following condition holds:

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



Specular Transmission

- 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

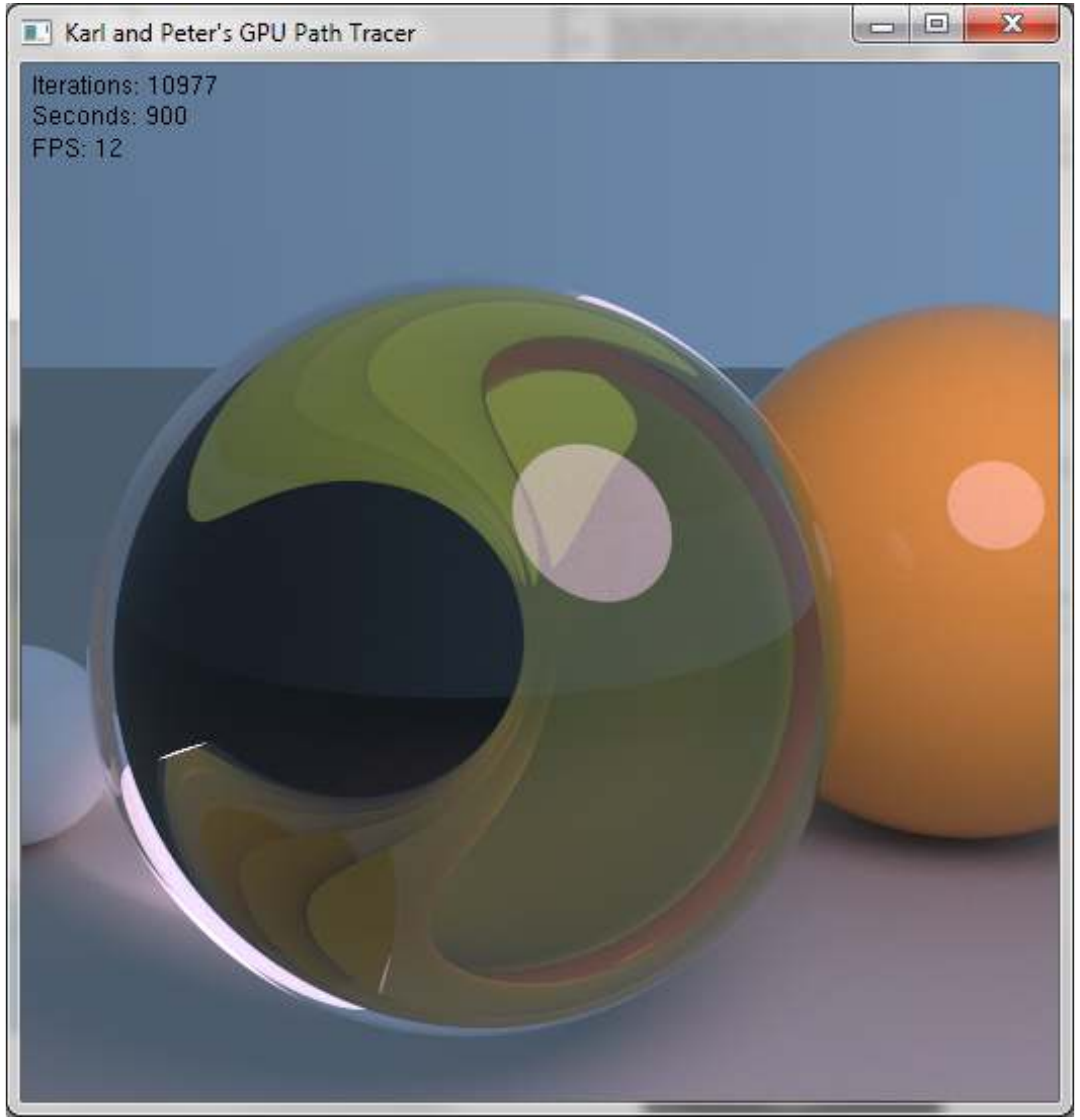
  (the "color" of the transparent material)

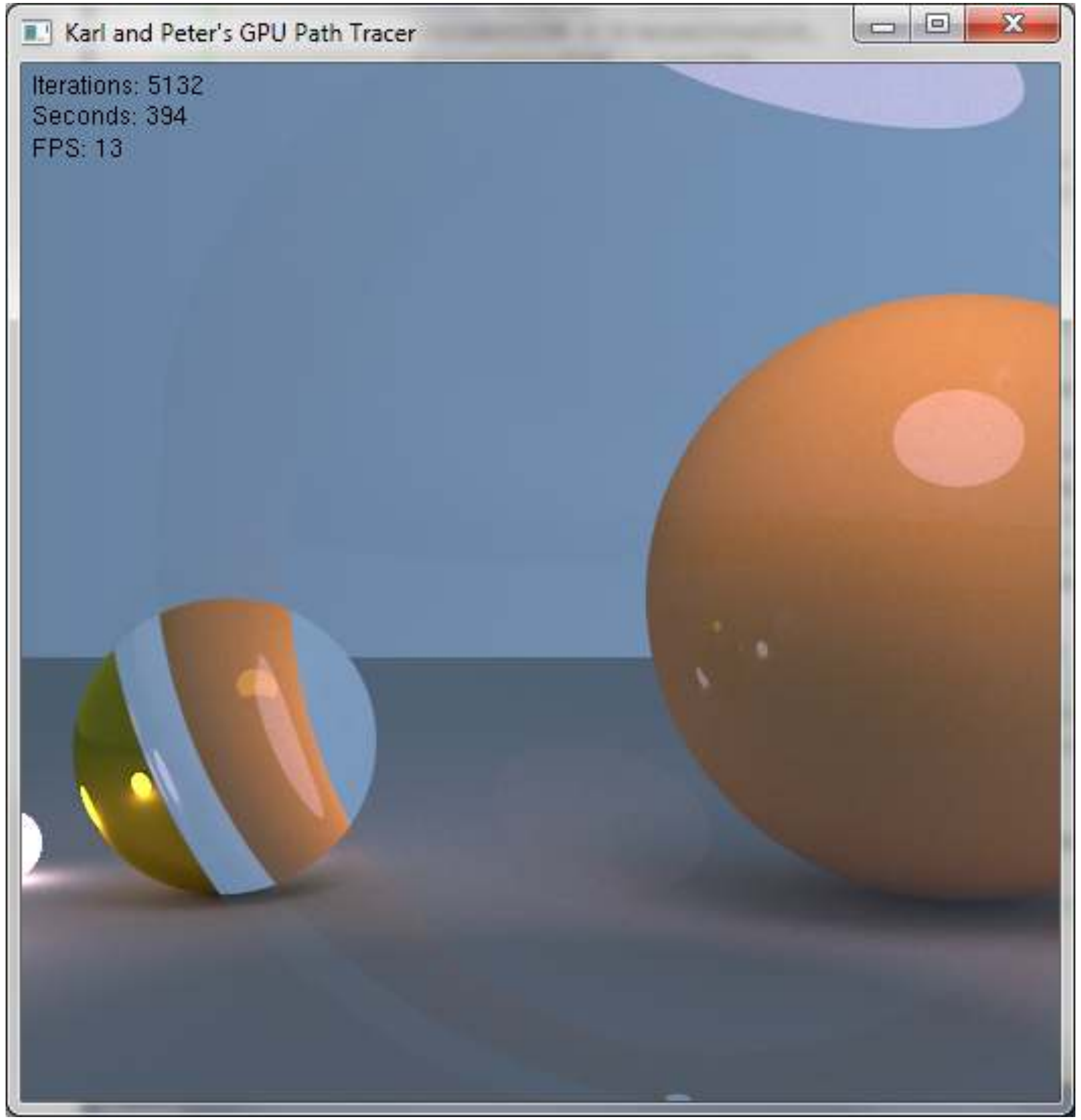
- L_r and L_t are calculated recursively, of course!

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 in the "wrong" direction?
  - Which direction is the wrong one anyway?

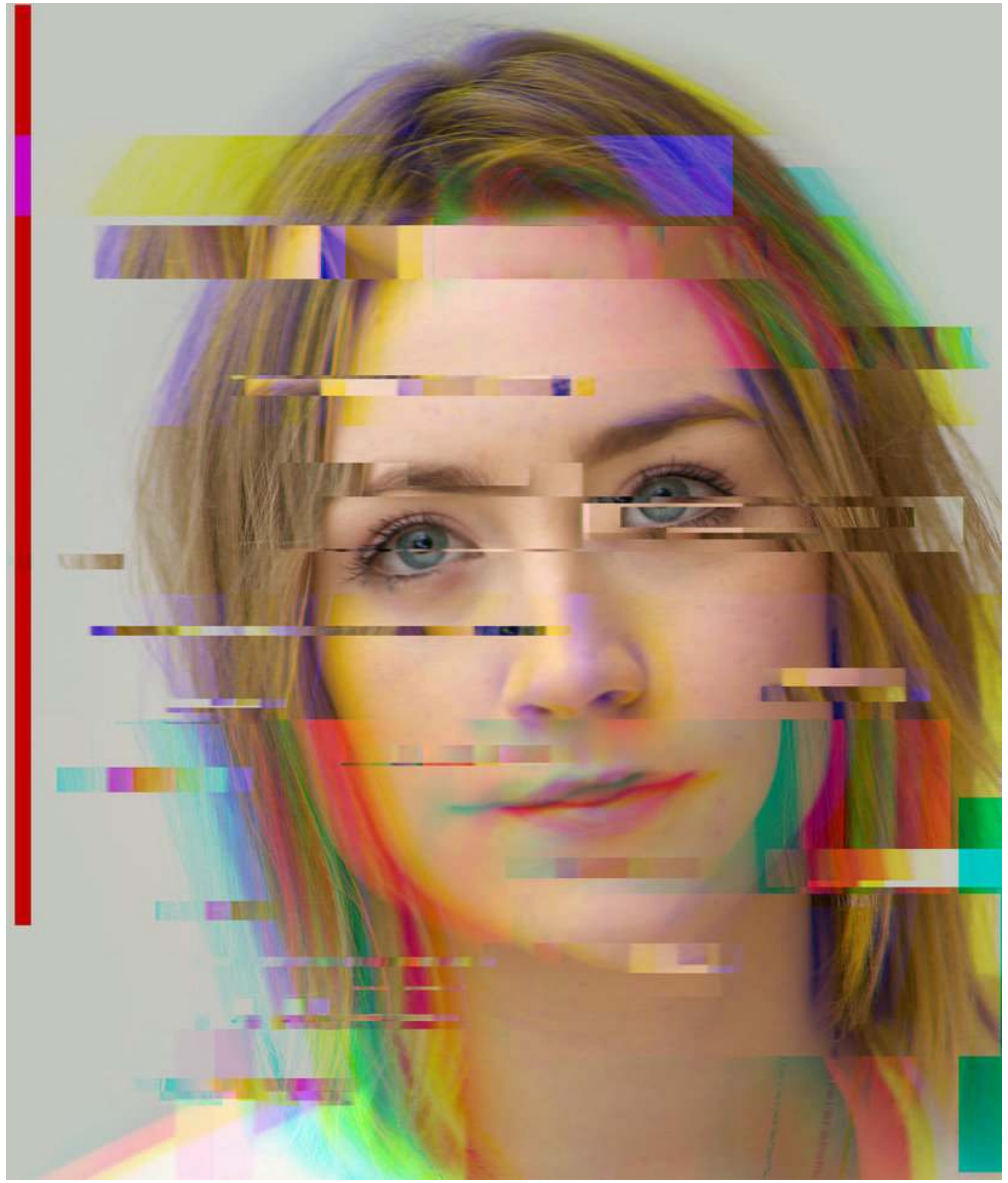
Glitch Pictures Showing Incorrect Refraction



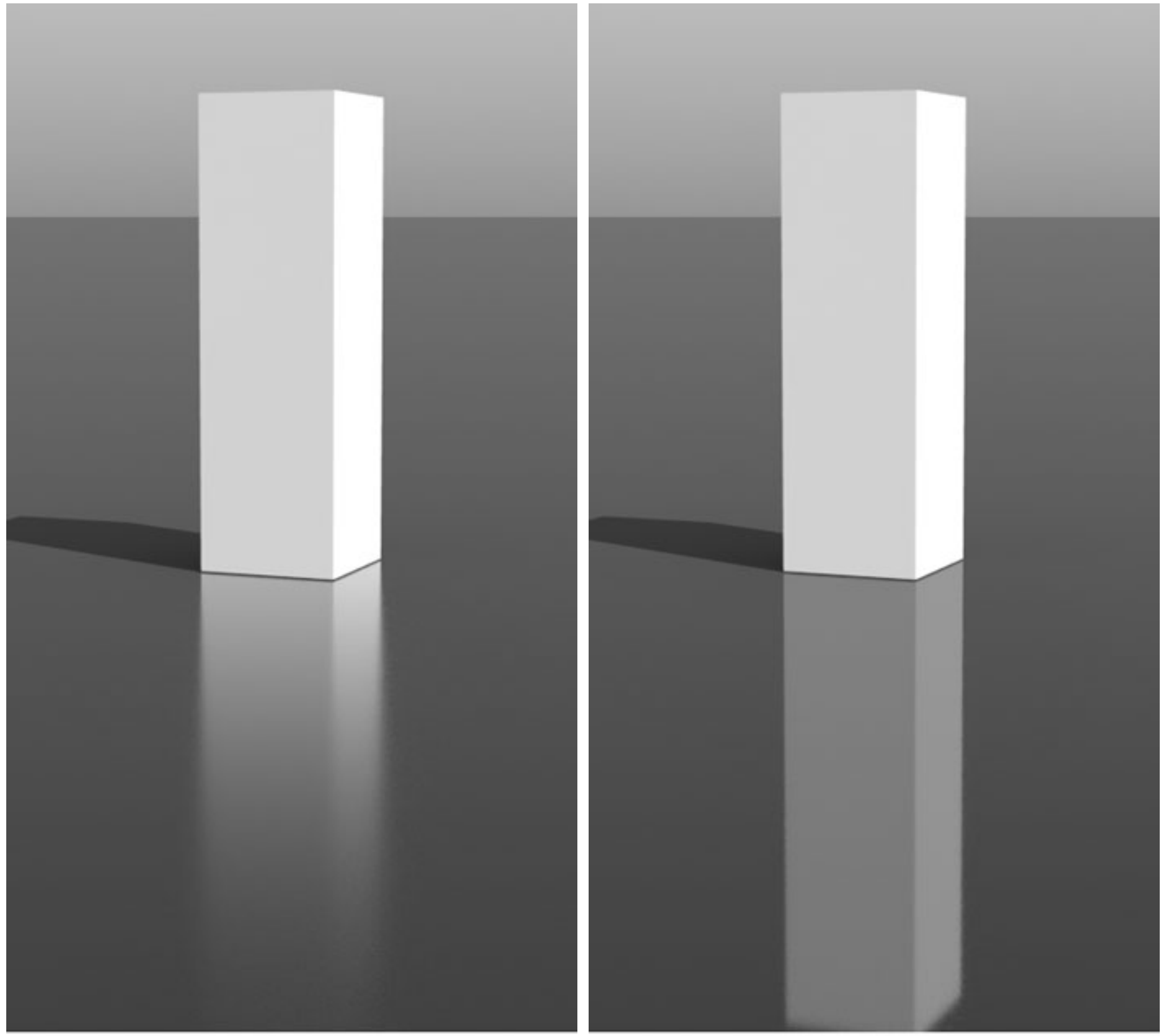


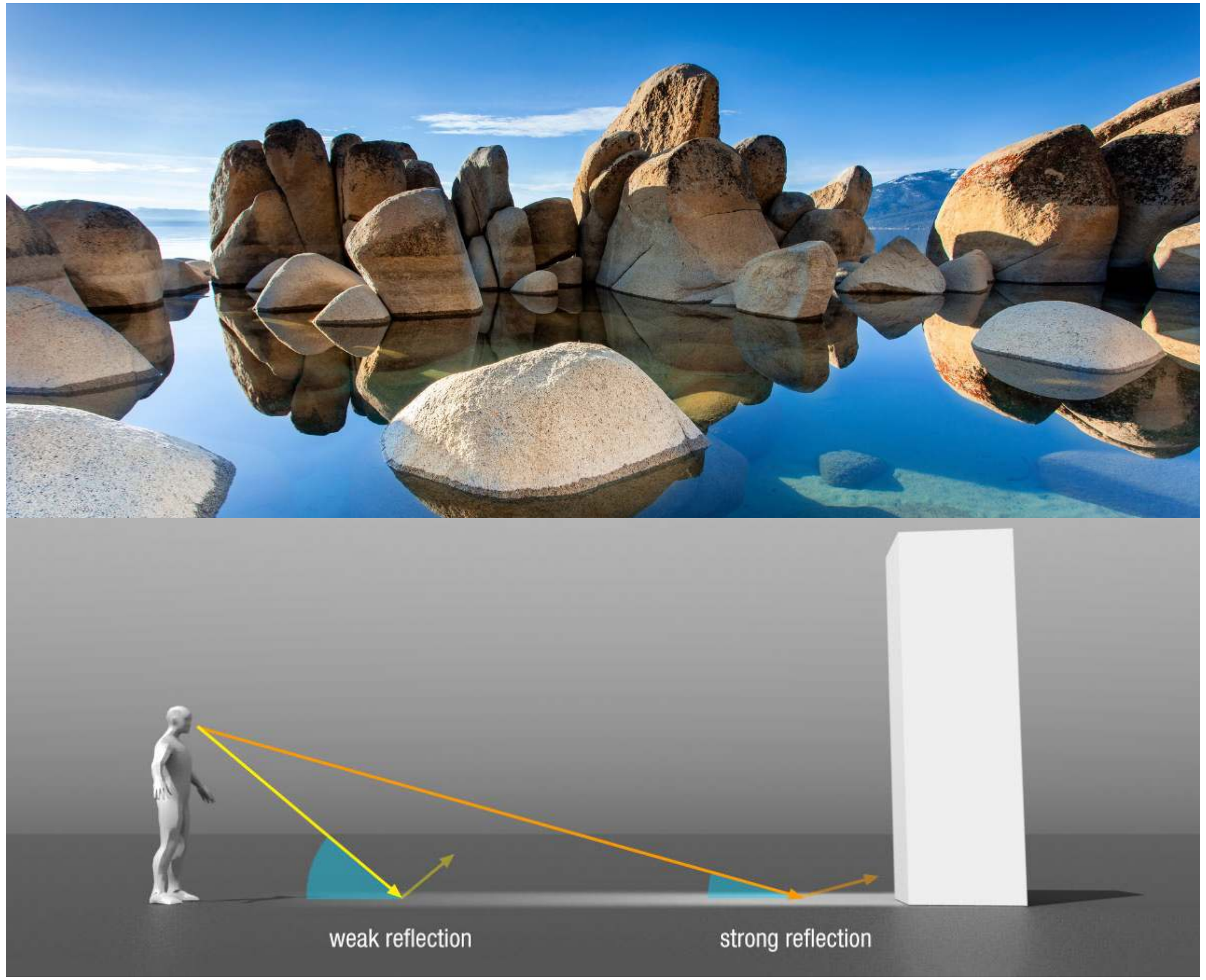
Source: yiningkarlli (http://igad2.nhtv.nl/ompf2)

Digression: Glitch Art (aka "Dirty Art")



Which Effect Can We Not Simulate Correctly (Yet)?





Remember our naïve summation of all incoming lights:

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

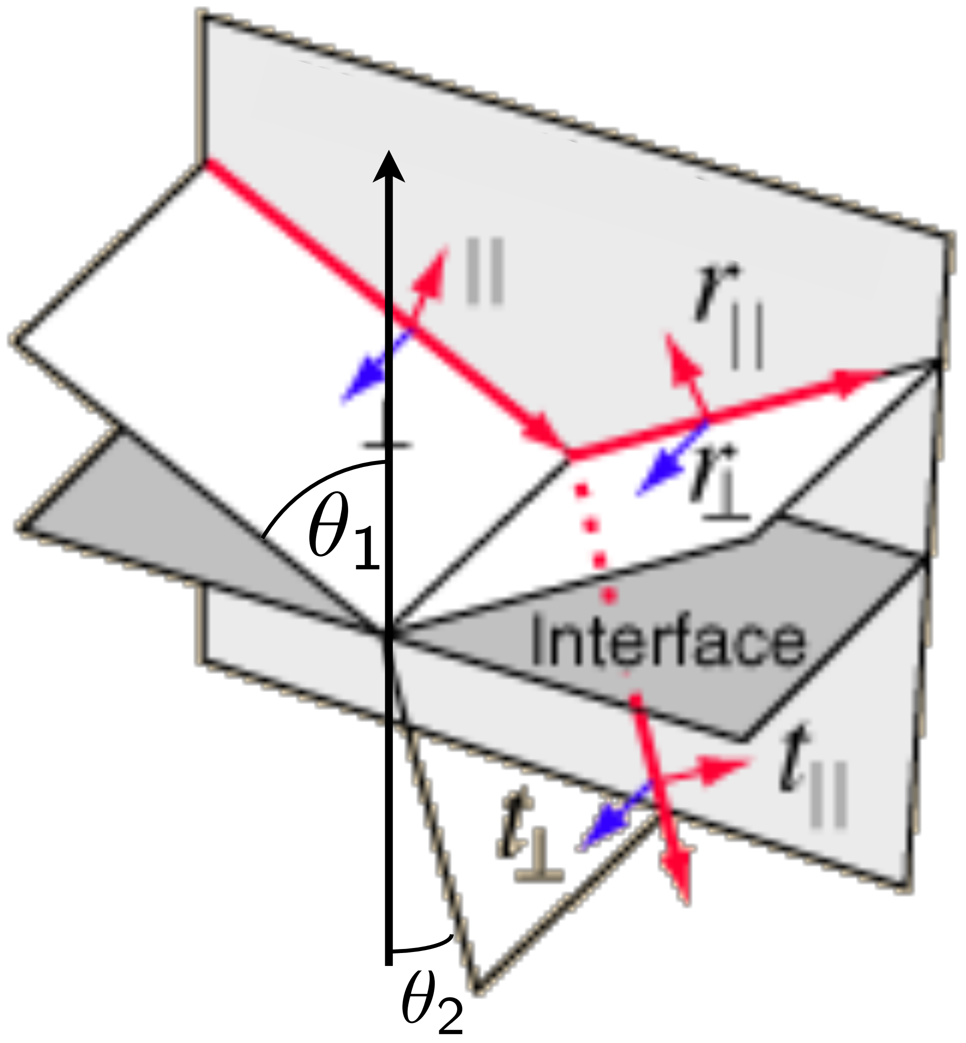
The Fresnel Terms for Translucent/Transparent Objs

- The reflectivity ρ depends on the refractive indices of the involved materials, and on the angle of incidence:

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

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

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



- The correct summation of the incoming lights is:

$$L_{\text{total}} = L_{\text{Phong}} + \rho k_s L_r + (1-\rho) k_t L_t$$

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

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

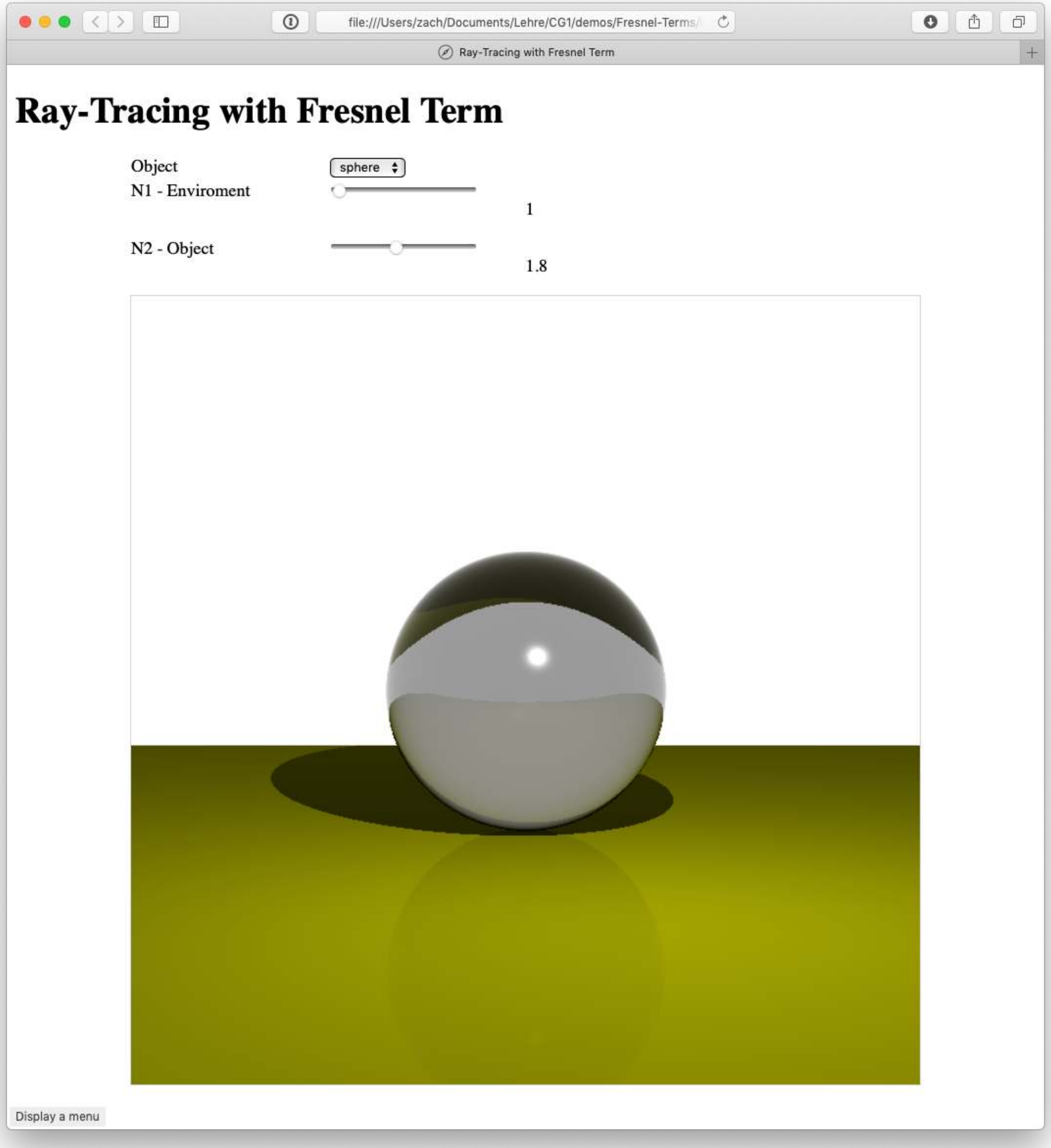
  - I.e., when moving perpendicularly from air to glass, 4% of the light is reflected, the rest is refracted

- Common approximation of the Fresnel term [Schlick 1994]:

$$\rho(\theta) \approx \rho_0 + (1-\rho_0)\,(1-\cos\theta)^5 \;, \qquad \rho_0 = \left(\frac{n_2-1}{n_2+1}\right)^2$$

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

Demo for Refraction Including Fresnel Terms

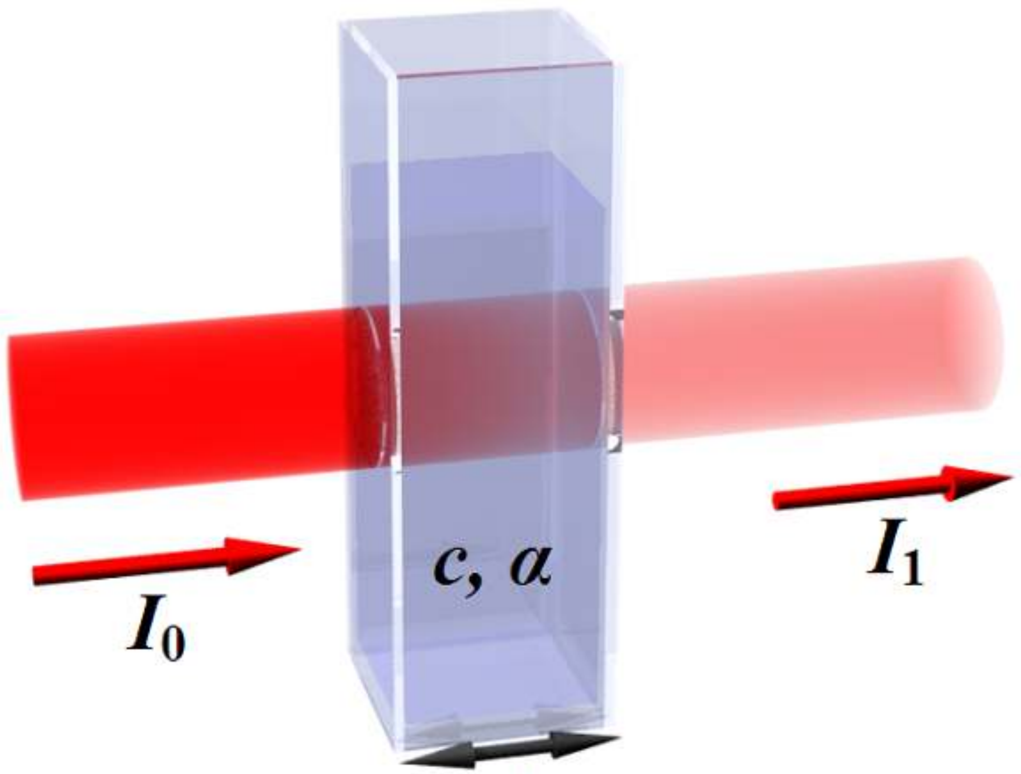


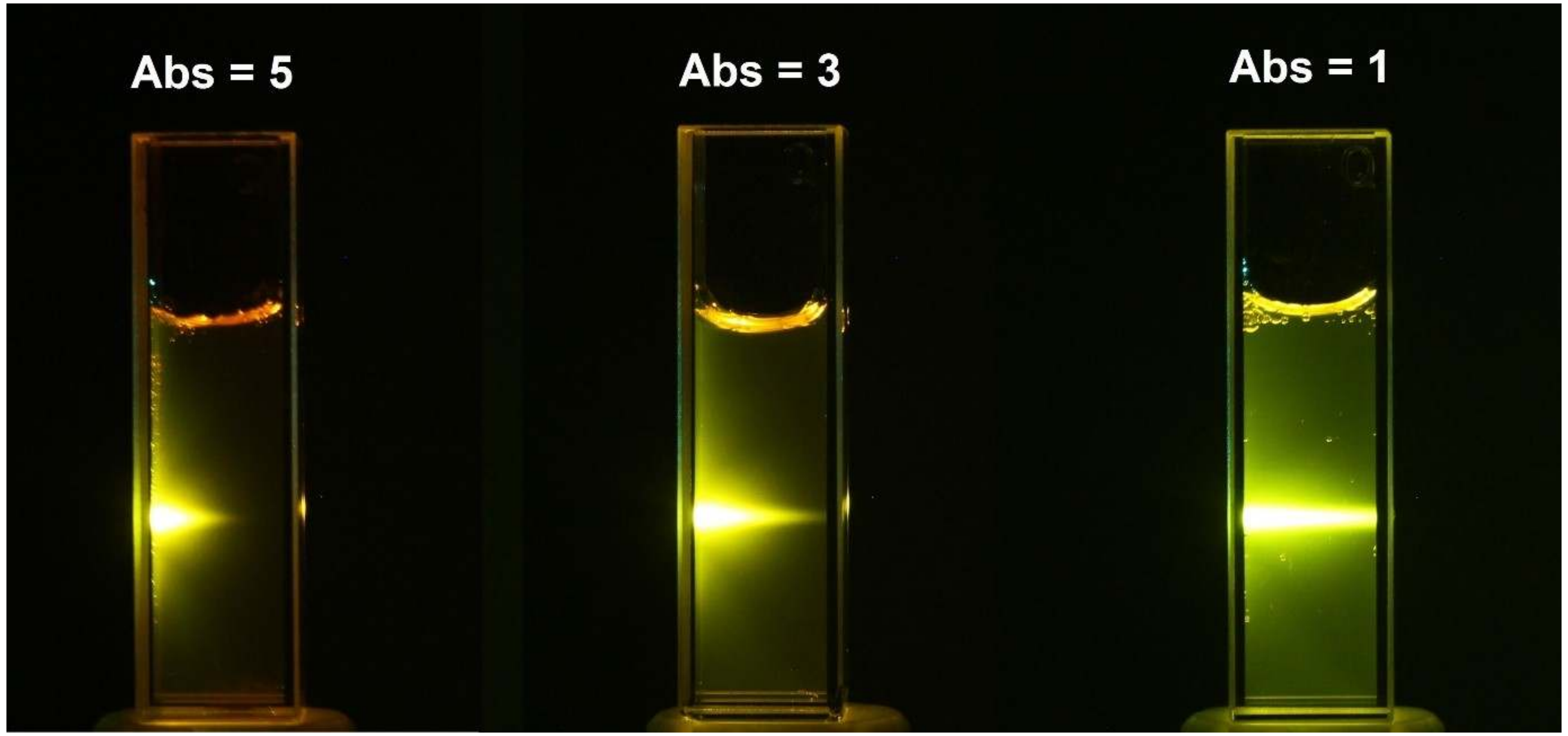
Attenuation (Dämpfung) in Participating Media

- 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:

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

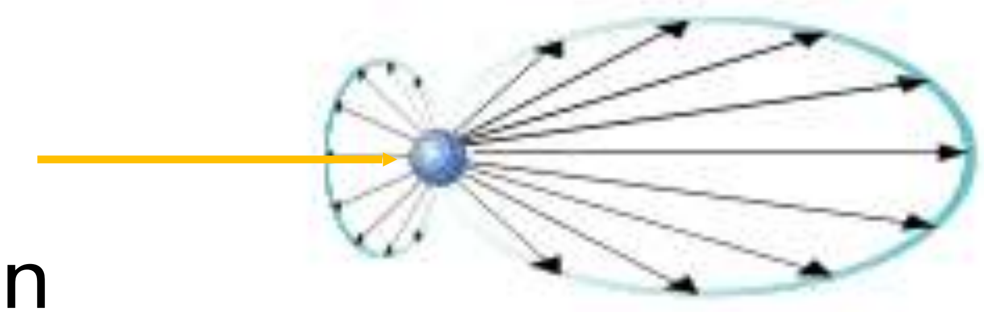
with α = some material constant, and
s = distance travelled in medium



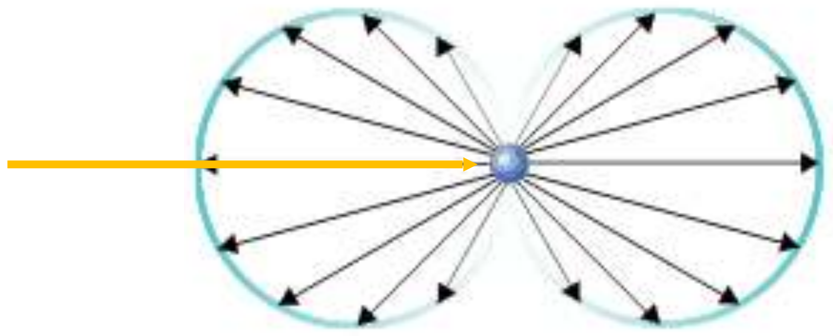


Scattering in Participating Media

1. Mie scattering



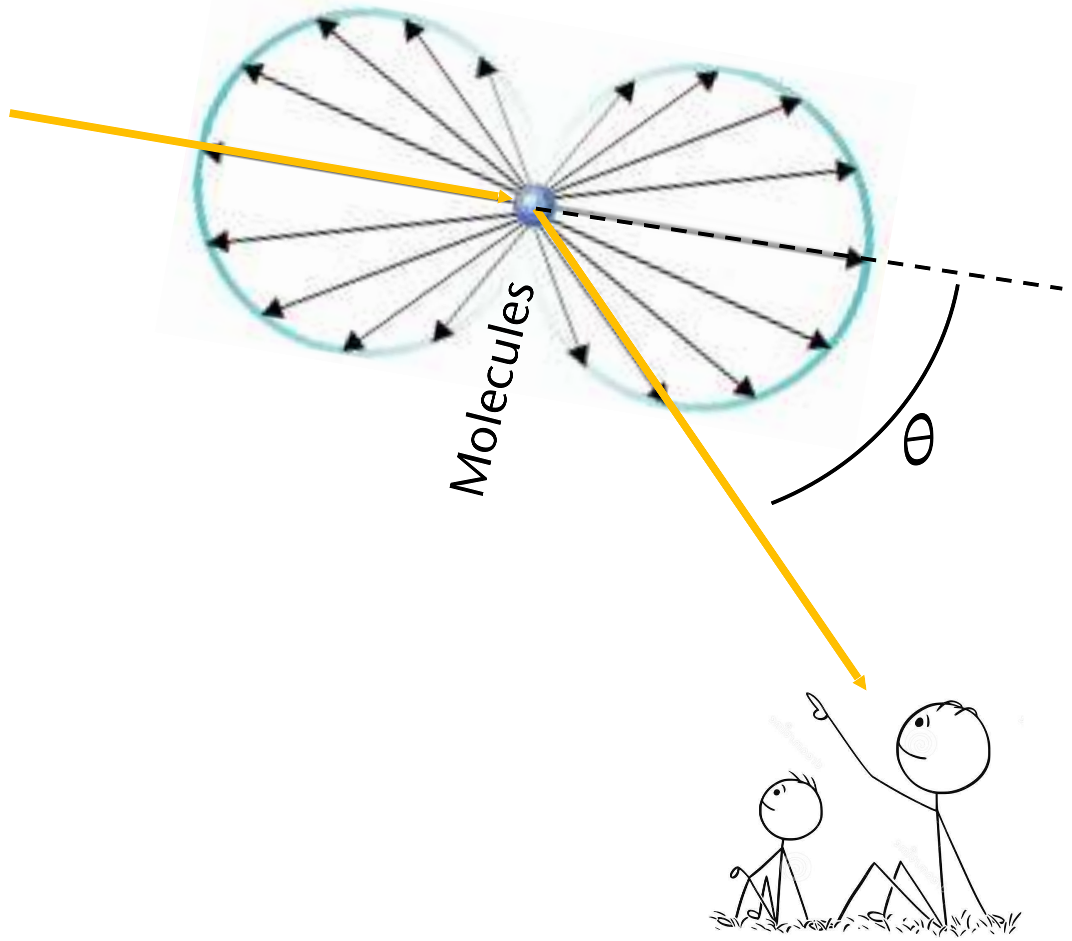
   - Shape of distribution
   - Size of particles > wavelength (e.g., haze or dust particles)
   - Scattered energy does *not* depend on wavelength



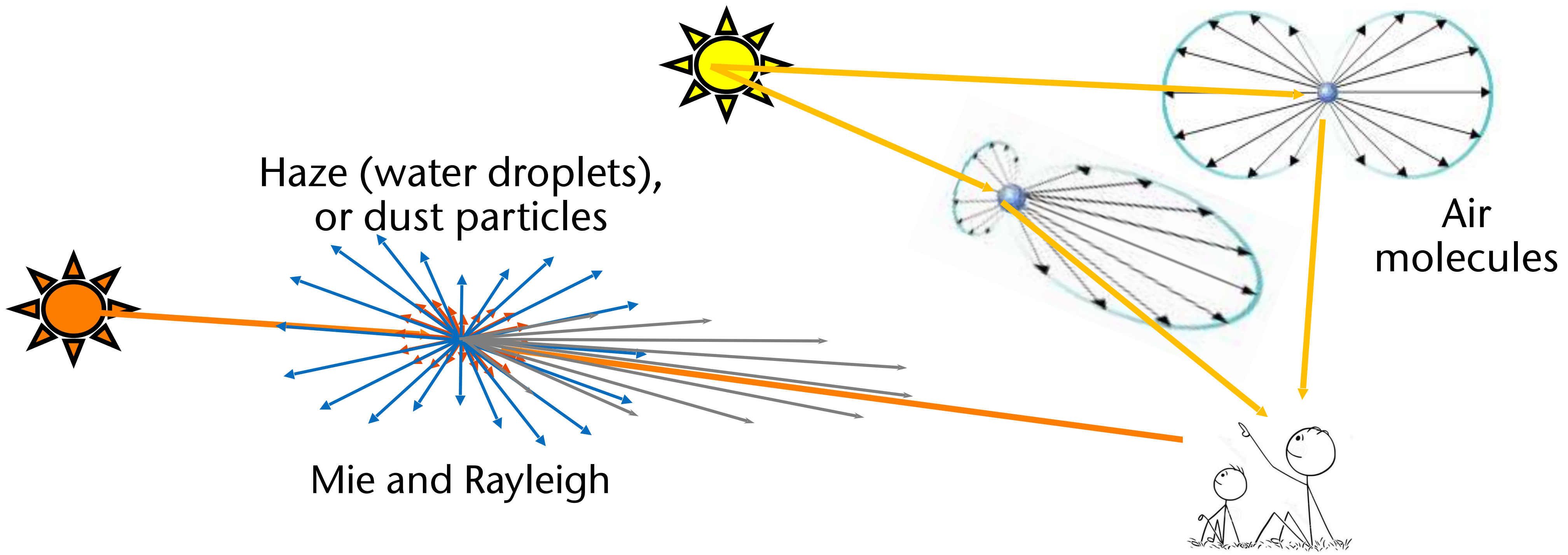
2. Rayleigh scattering
   - Shape of distribution
   - Size of particles < 0.1 * wavelength, i.e. molecules (O_2, NO)
   - Scattered energy *does* depend on wavelength

- Equation of Rayleigh scattering:

$$L_{\text{out}} = k_{ss} \frac{1+\cos^2\theta}{2} \frac{1}{\lambda^4} L_{\text{in}}$$

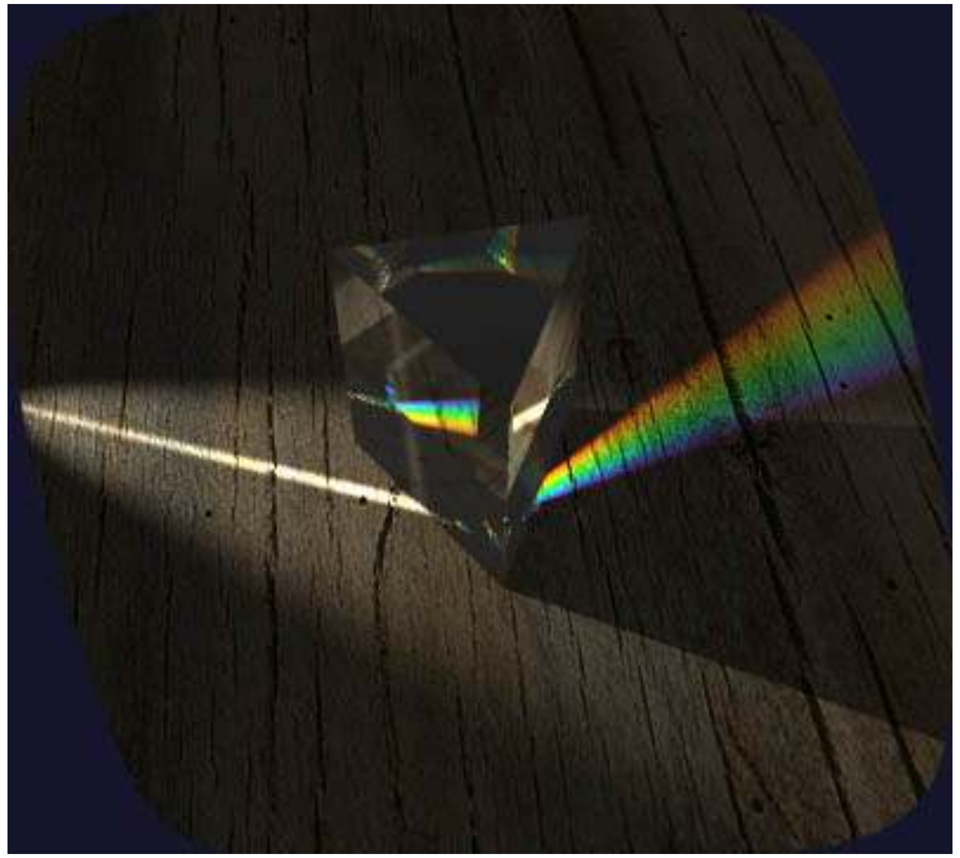


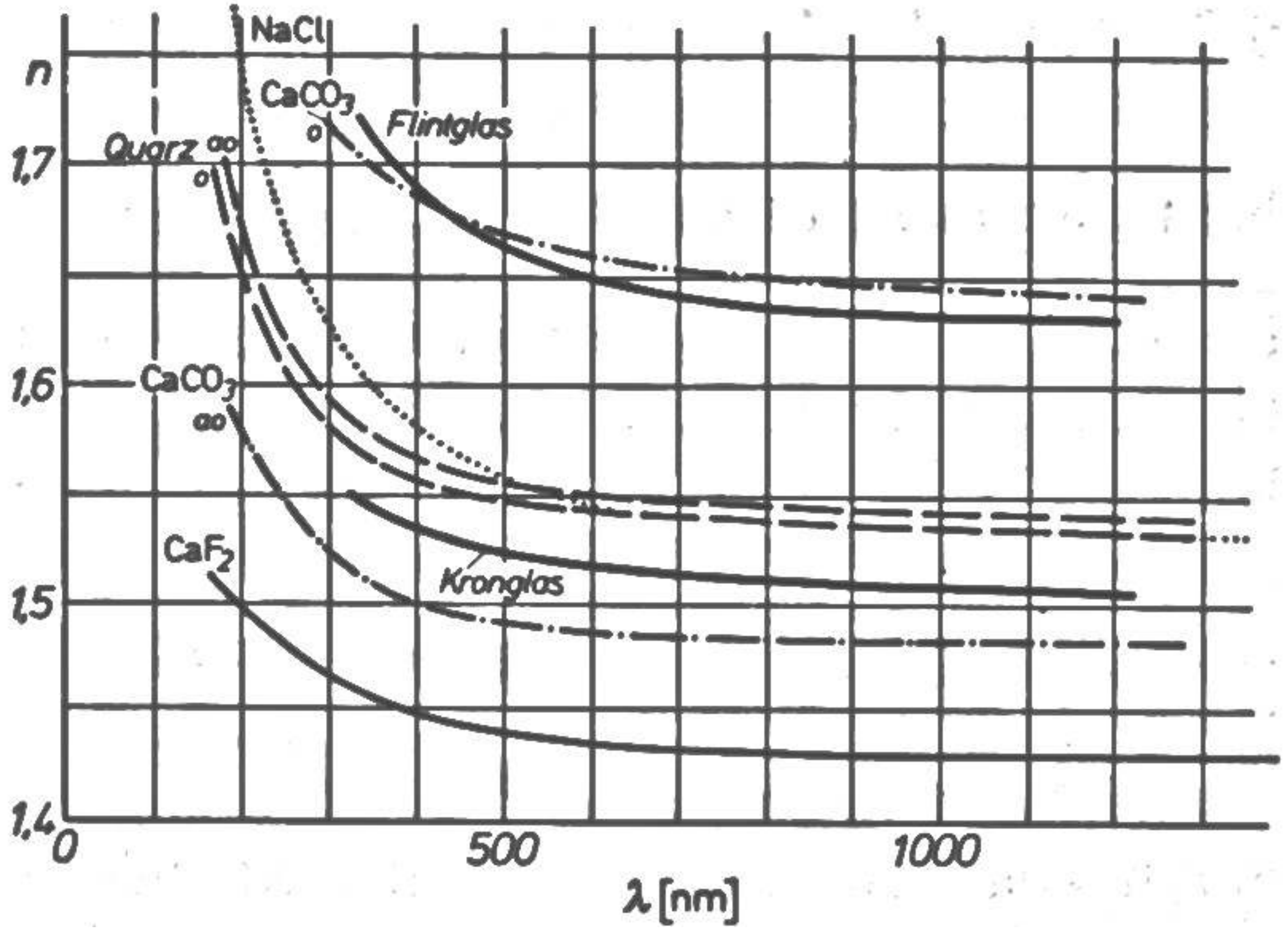
- Blue sky, white sky, red sky:
  - Remember, the atmosphere is a relatively thin hull!
  - During sunset, the path through air is "long"
  - "In-scattering" and "out-scattering" (blue vs. red sky)



Dispersion, Spectral Raytracing

- In reality, the refractive index *n* depends on the wavelength!
- This effect cannot be modelled any more with simple "RGB light"; this requires a spectral ray-tracer
  - Instead of 3 channels, we simulate 10+ channels





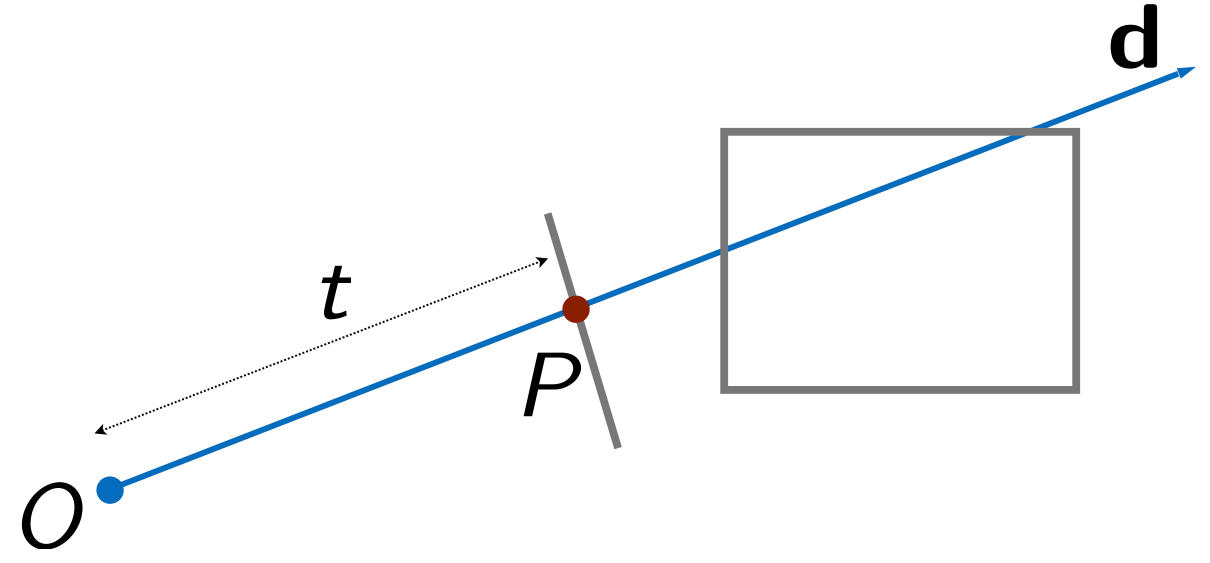
Giovanni Battista Pittoni, 1725,
"An Allegorical Monument to Sir
Isaac Newton"

Example with Fresnel Terms and Dispersion (RGB only)

Intersection Computations Ray against Primitive

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

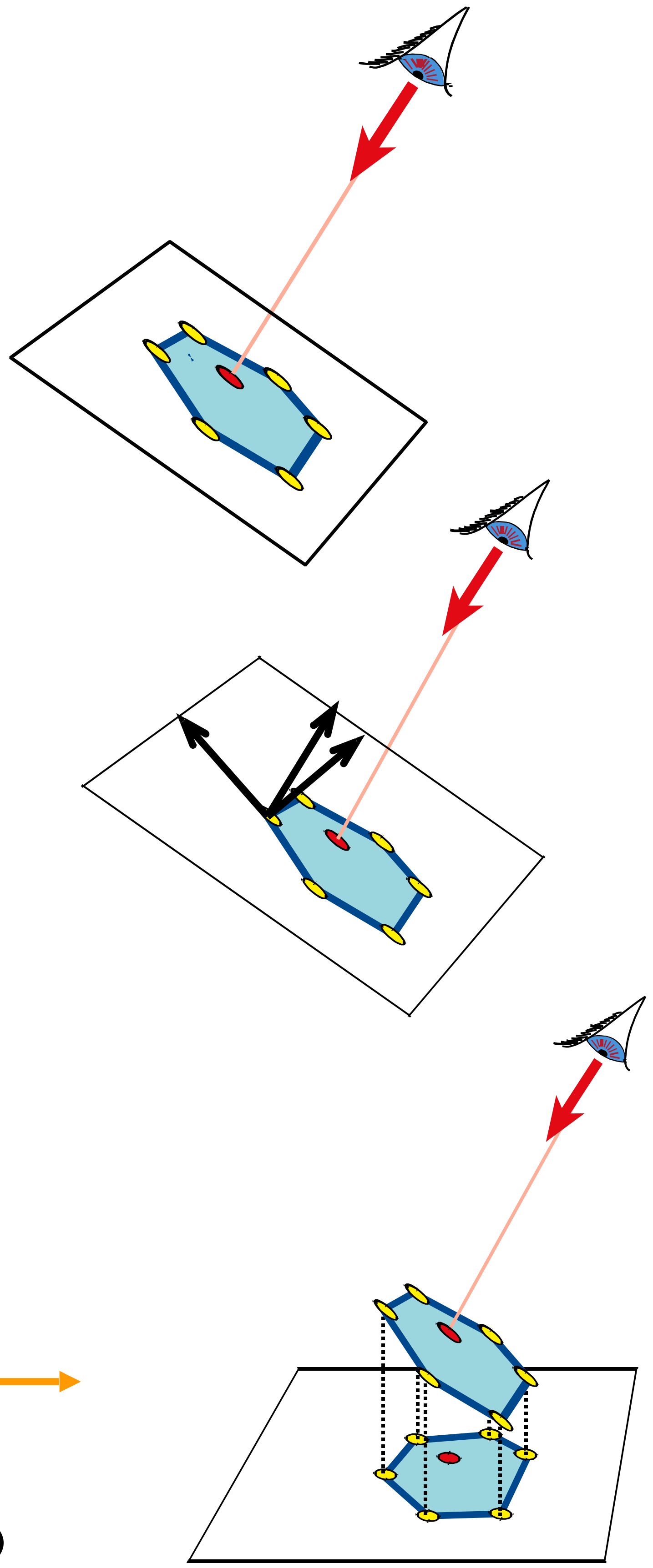
$$P(t) = O + t \cdot \mathbf{d}$$

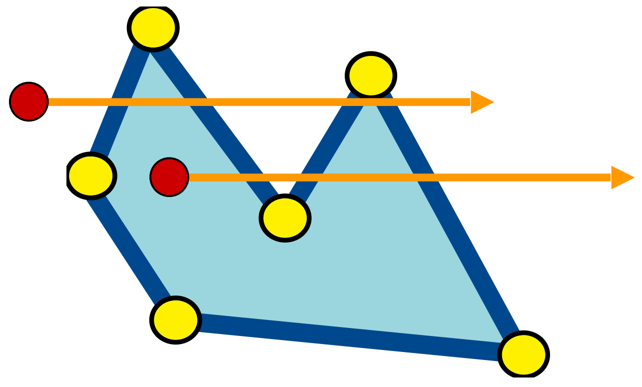


- Wanted: the line parameter t of the *first* intersection point $P = P(t)$ with the scene

Intersection of Ray with General 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
  - If triangle ⟶ barycentric coord test





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, ...

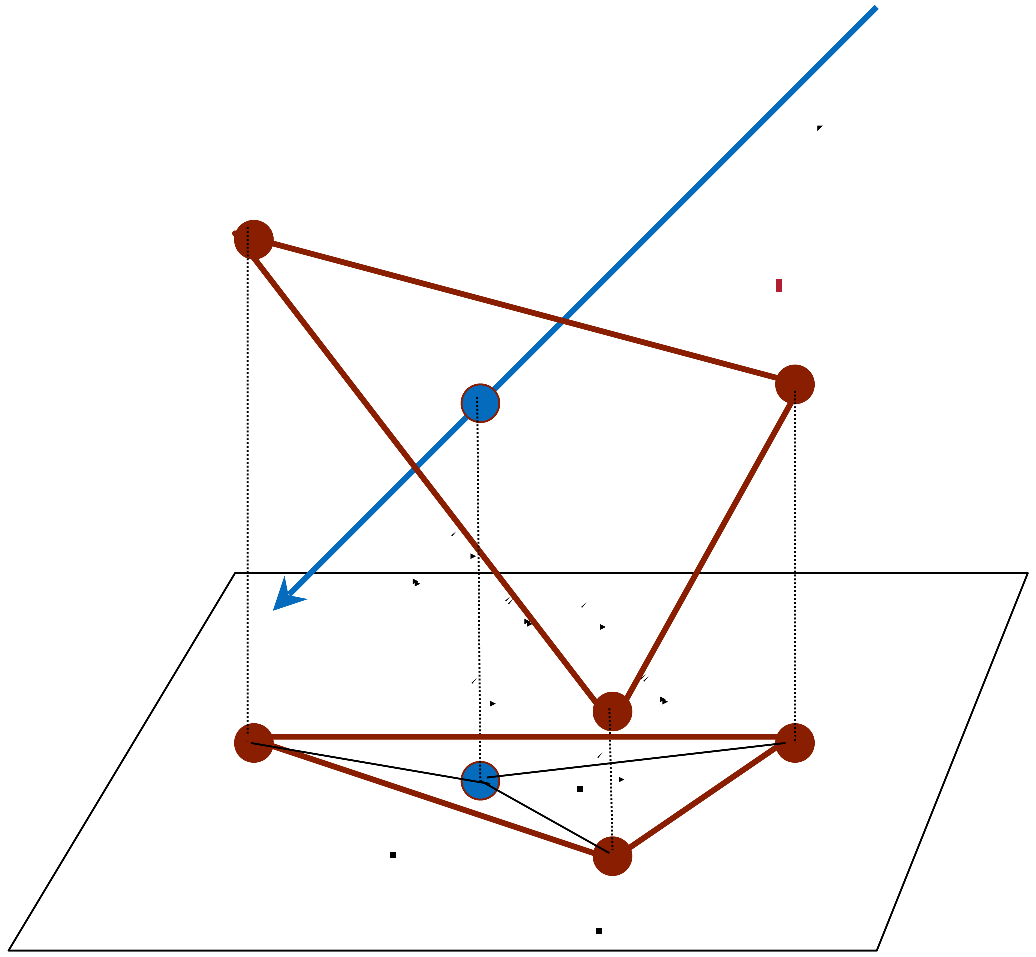
The `intersect` function can be optimized compared to the one at the beginning; in addition, only intersection points *before* the light source are relevant.

Evaluates the lighting model for the hit object

Intersection of Ray with Triangle

[Badouel 1990]

- Use same method like ray—polygon; or
- Be clever: use projection and barycentric coords
- Intersect ray with plane (implicit form) $\rightarrow t \rightarrow$ point in space
- Project point & triangle on coord plane
- Compute barycentric coords of 2D point
- Barycentric coords of 2D point (α,β,γ) =
  barycentric coords of orig. 3D point! (w/o proof)
- 3D point is in triangle $\Leftrightarrow \quad \alpha,\beta,\gamma > 0$, with $\alpha + \beta + \gamma = 1$
- Alternative method: see Möller & Haines "Real-time Rendering"
- (Faster method exists, if intersection point is not needed [Segura & Feito])



Alternative Intersection Method for Ray—Triangle [Möller]

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

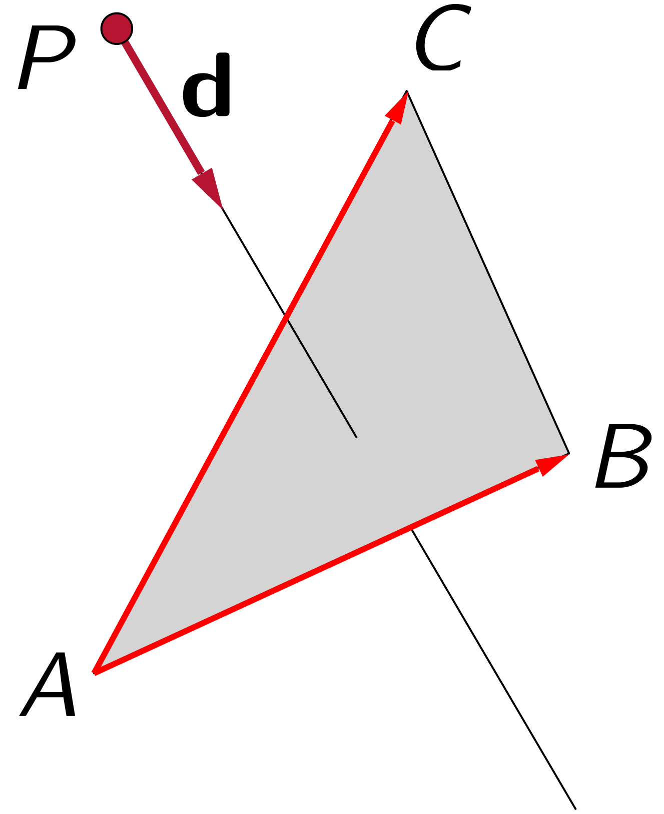
$$-t \cdot \mathbf{d} + r \cdot (B - A) + s \cdot (C - A) = P - A$$

- 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:

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

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

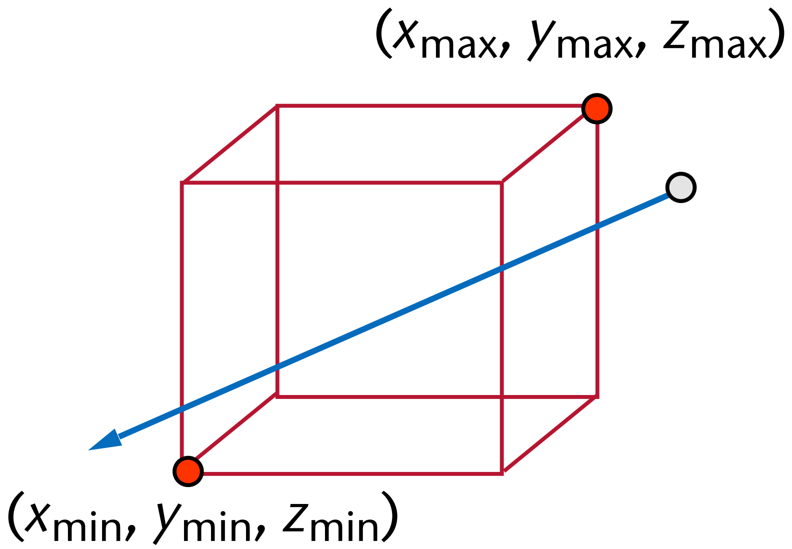
$$\begin{pmatrix} t \\ r \\ s \end{pmatrix} = \frac{1}{(\mathbf{d} \times \mathbf{v}) \cdot \mathbf{u}} \cdot \begin{pmatrix} (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{v} \\ (\mathbf{d} \times \mathbf{v}) \cdot \mathbf{w} \\ (\mathbf{w} \times \mathbf{u}) \cdot \mathbf{d} \end{pmatrix}$$

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

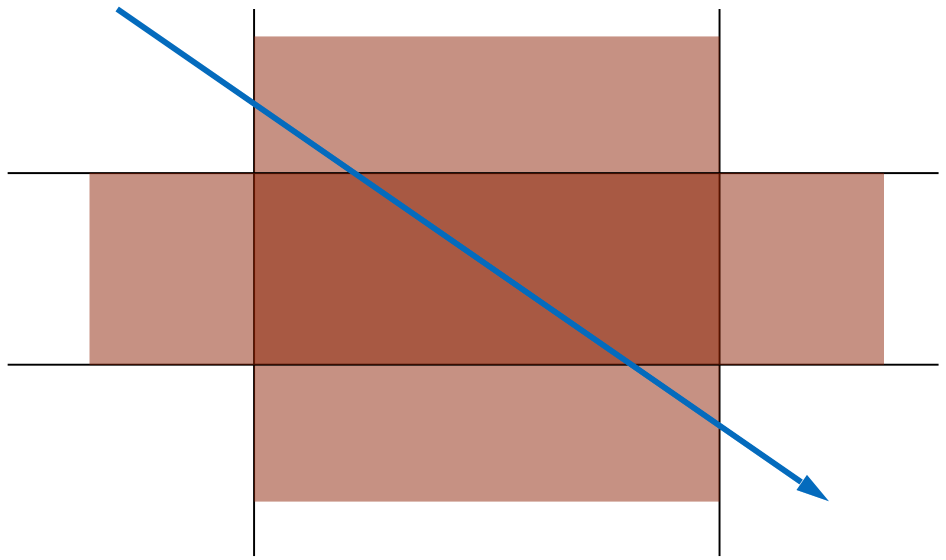
Intersection of Ray and Box

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

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

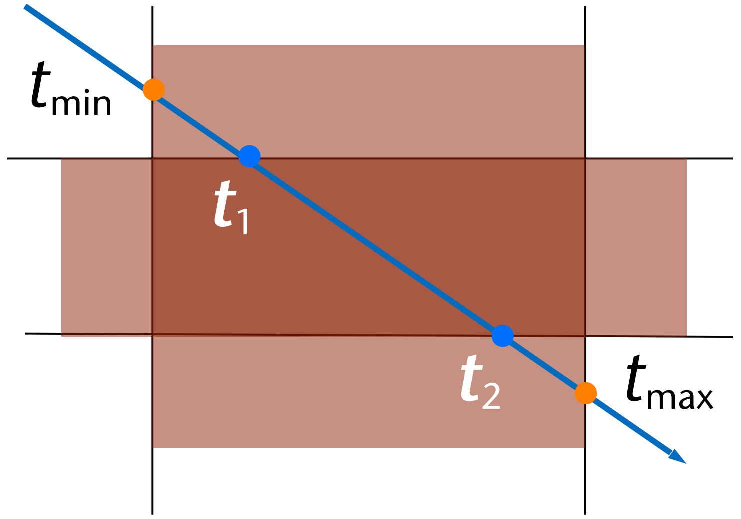


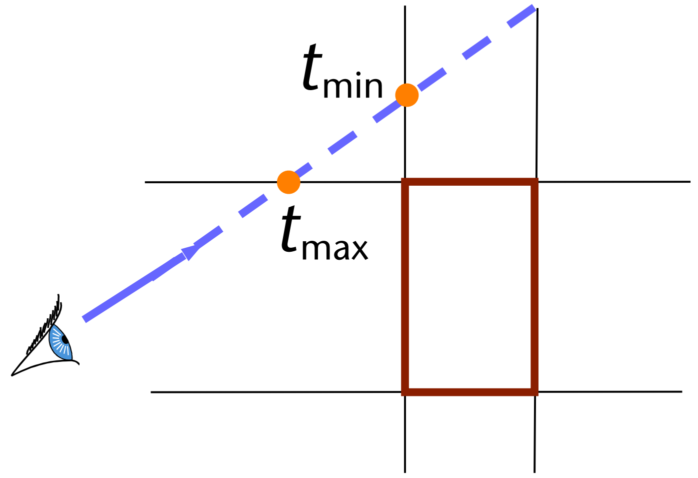
- Idea of the algorithm:
  - A box is the intersection of 3 *slabs* (*slab* = subset of space enclosed between two parallel planes)
  - Each slab cuts away a specific interval of the ray
  - So, successively consider two parallel (= opposite) planes of the box

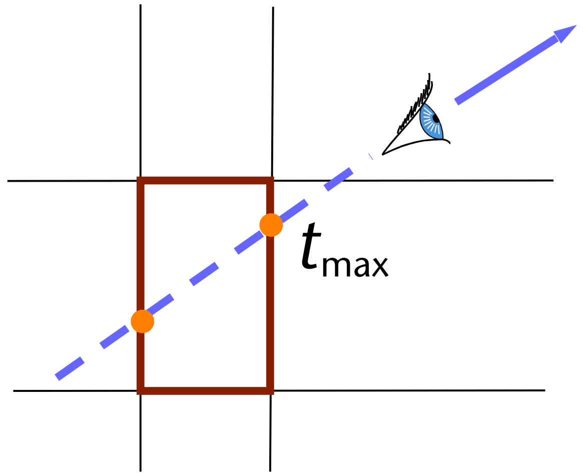


The Algorithm

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



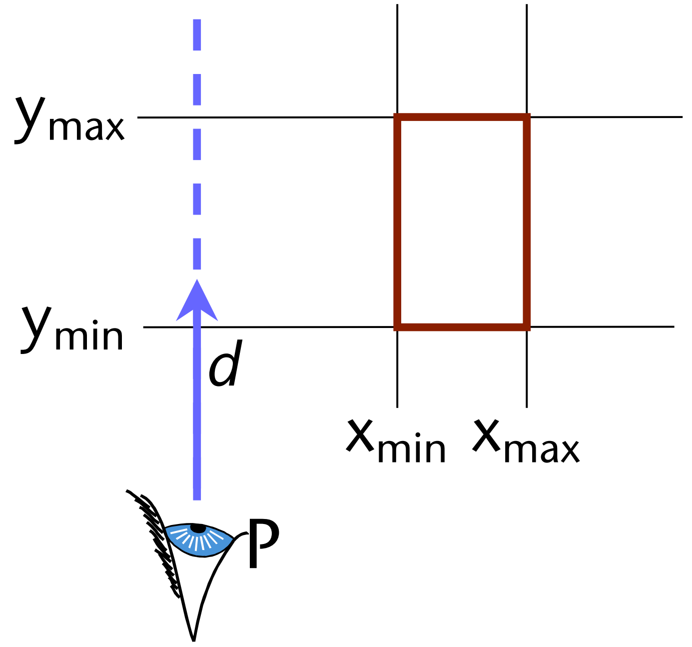




Remarks

- Optimization: both planes of a slab have the same normal $\rightarrow$ can save one dot product
- Remark: the algorithm also works for "tilted" boxes (called *OBBs = oriented bounding boxes*)
- Further optimization: in case of AABB, exploit the fact that the normal has exactly one component = 1, others = 0!
- Warning: "shit happens"
  - Here: test for parallel situations!
  - In case of 2D AABB:

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



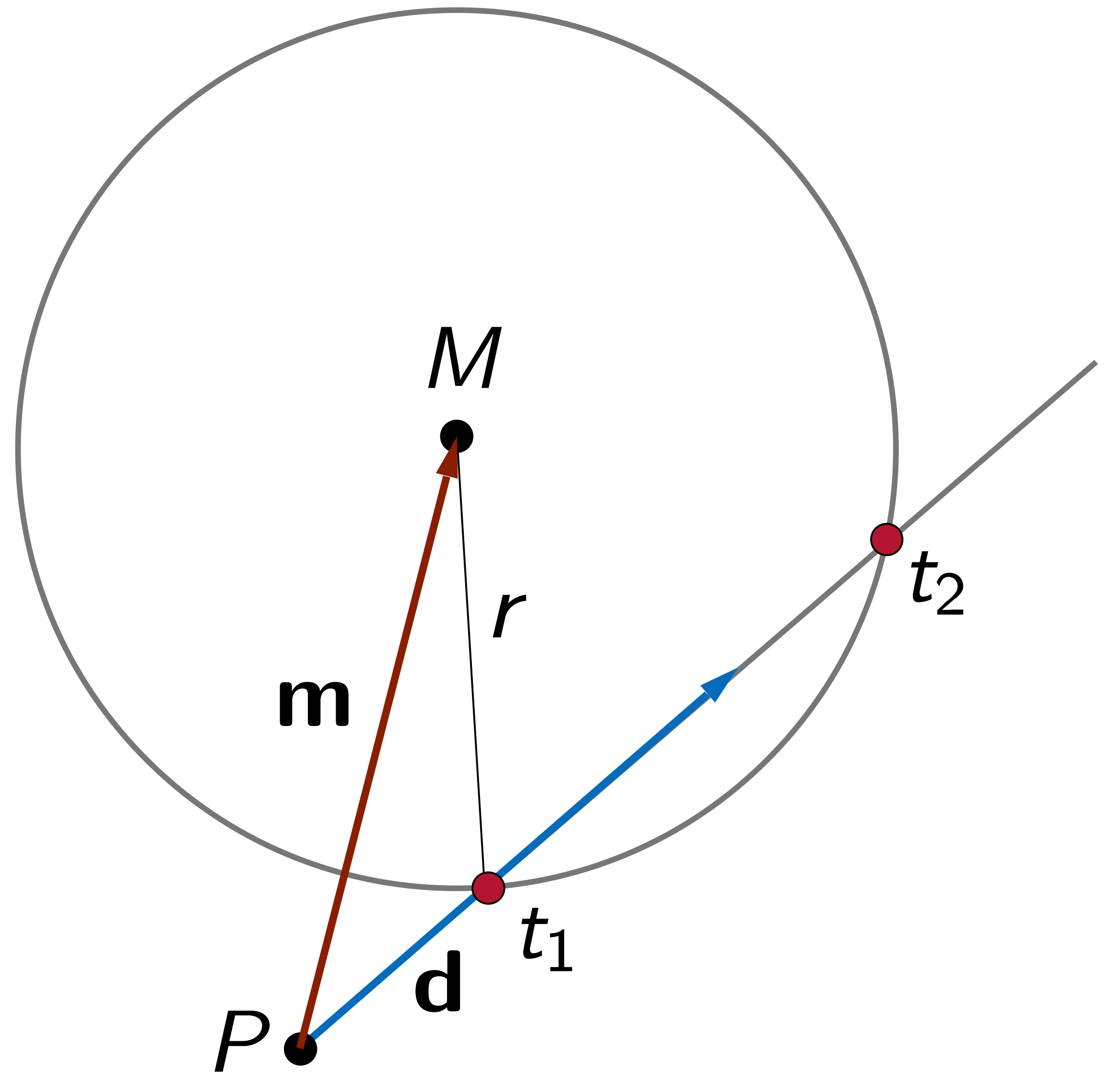
Intersection Ray—Sphere

- Assumption: **d** has length 1
- The geometric method:

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

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

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

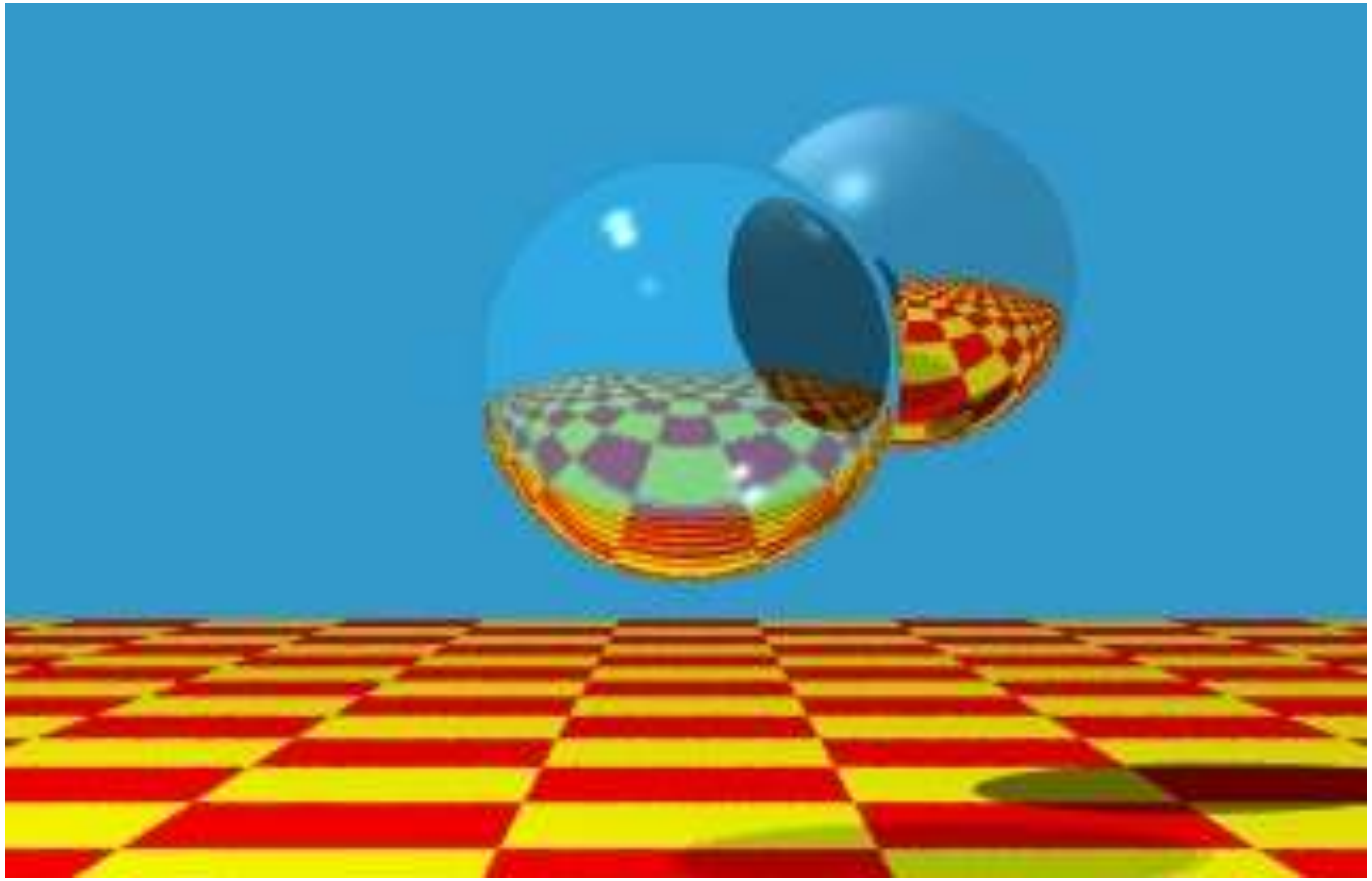


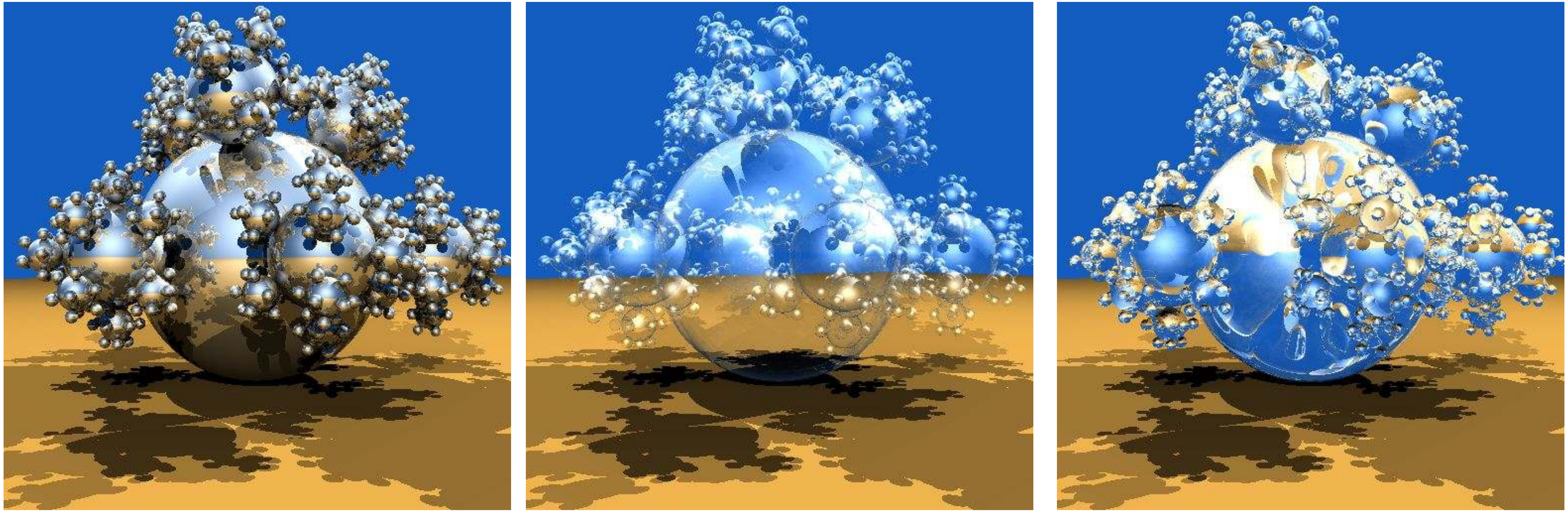
- The algebraic method: insert ray equation into implicit sphere equation
- There are many more approaches ...

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

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
```

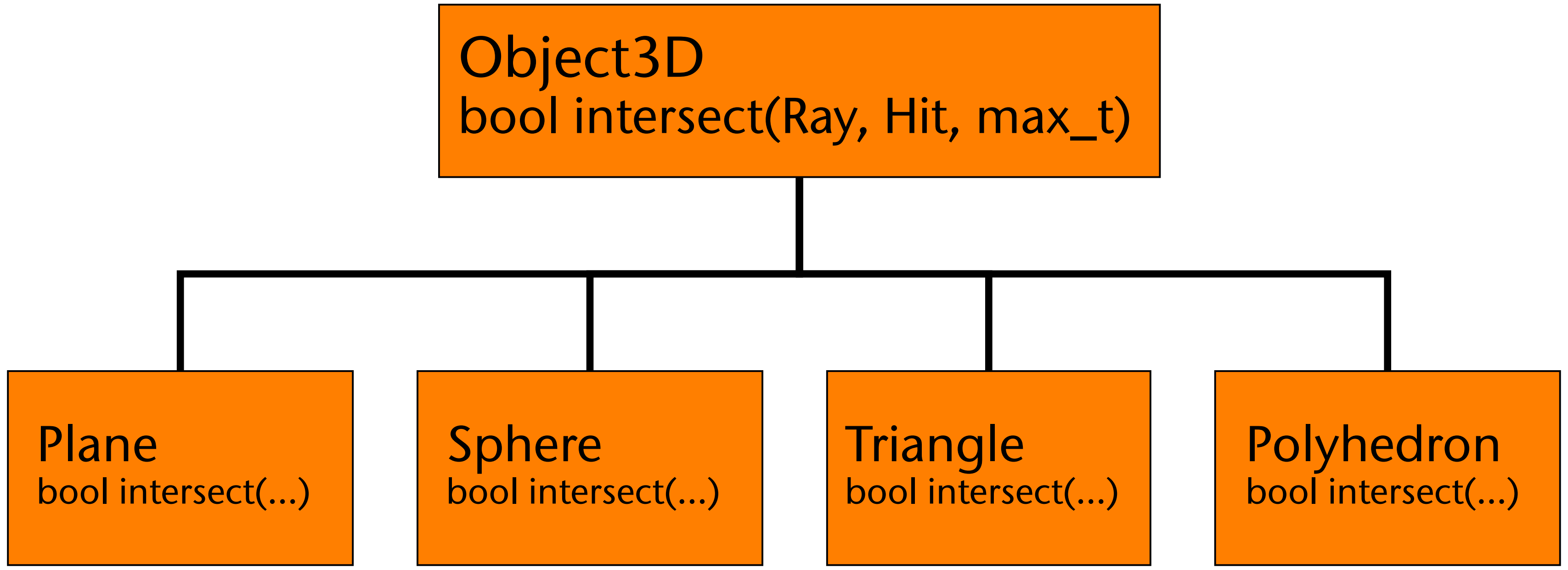
- A class for rays:

```
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
```

- Object3D = abstract base class for all geometry primitives




```
// abstract intersection methods: ray against any object
virtual bool closestIntersection( Intersection * hit ) = 0;
virtual bool anyIntersection( const Ray & ray, float max_t,
                              Intersection * hit ) = 0;
// normal at hit point
virtual Vector calcNormal( Intersection * hit ) = 0;
// material of object
int getMaterialIndex() const;
```

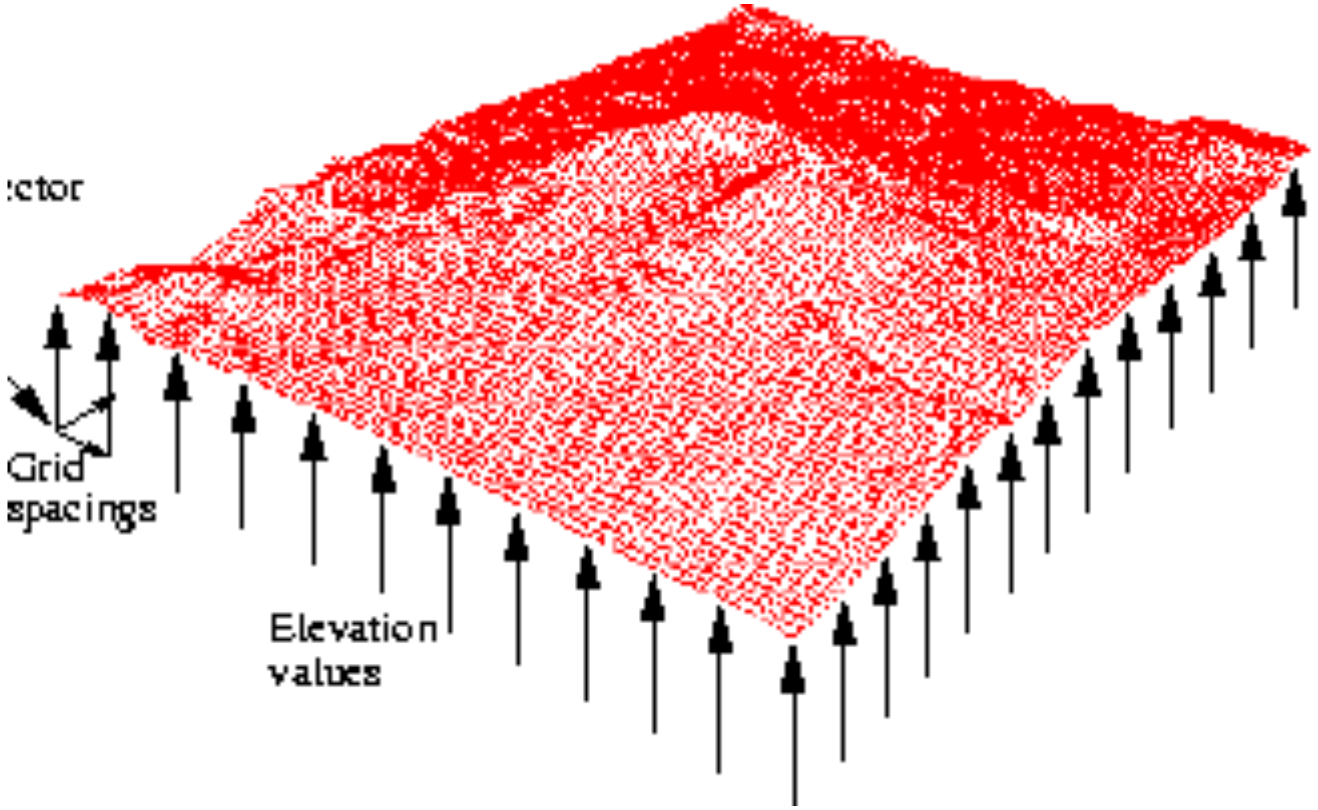
- Camera:
  - Captures all properties of a virtual camera, e.g., *from, at, up, angle*
  - Generates primary rays for all pixels
- Scene:
  - Stores all data about the scene
    - List of all objects
    - List of all materials
    - List of all light sources
    - Camera
  - Offers methods for calculating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

Ray-Tracing Height Fields

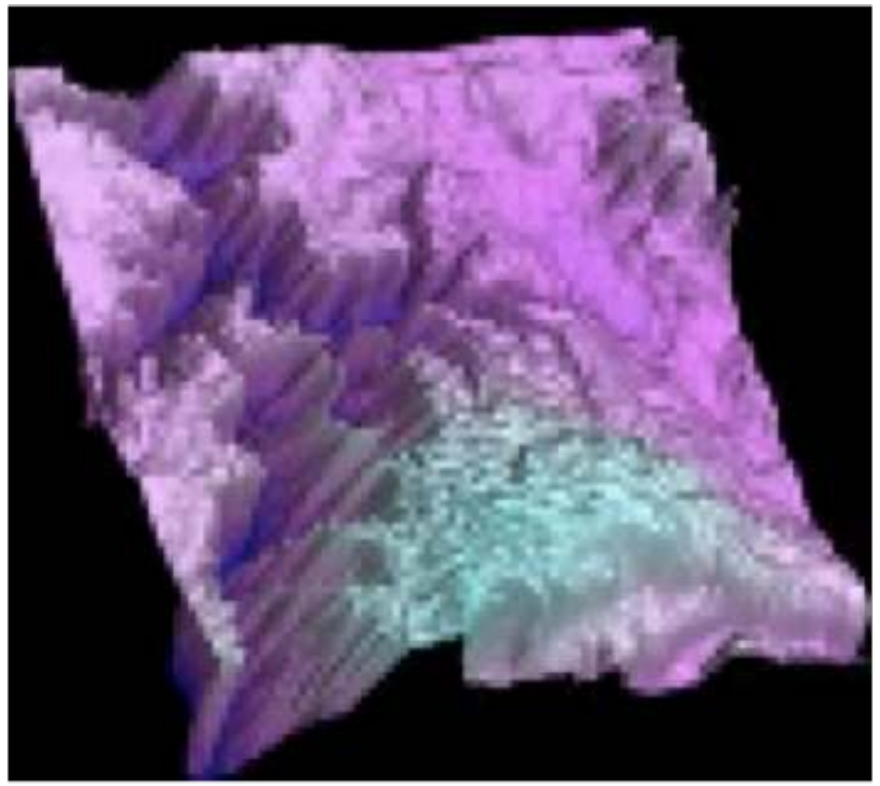
- Height Field = all kinds of surfaces that can be described by such a function

$$z = f(x, y)$$

  - Examples: terrain, measurements sampled on a plane, 2D scalar field



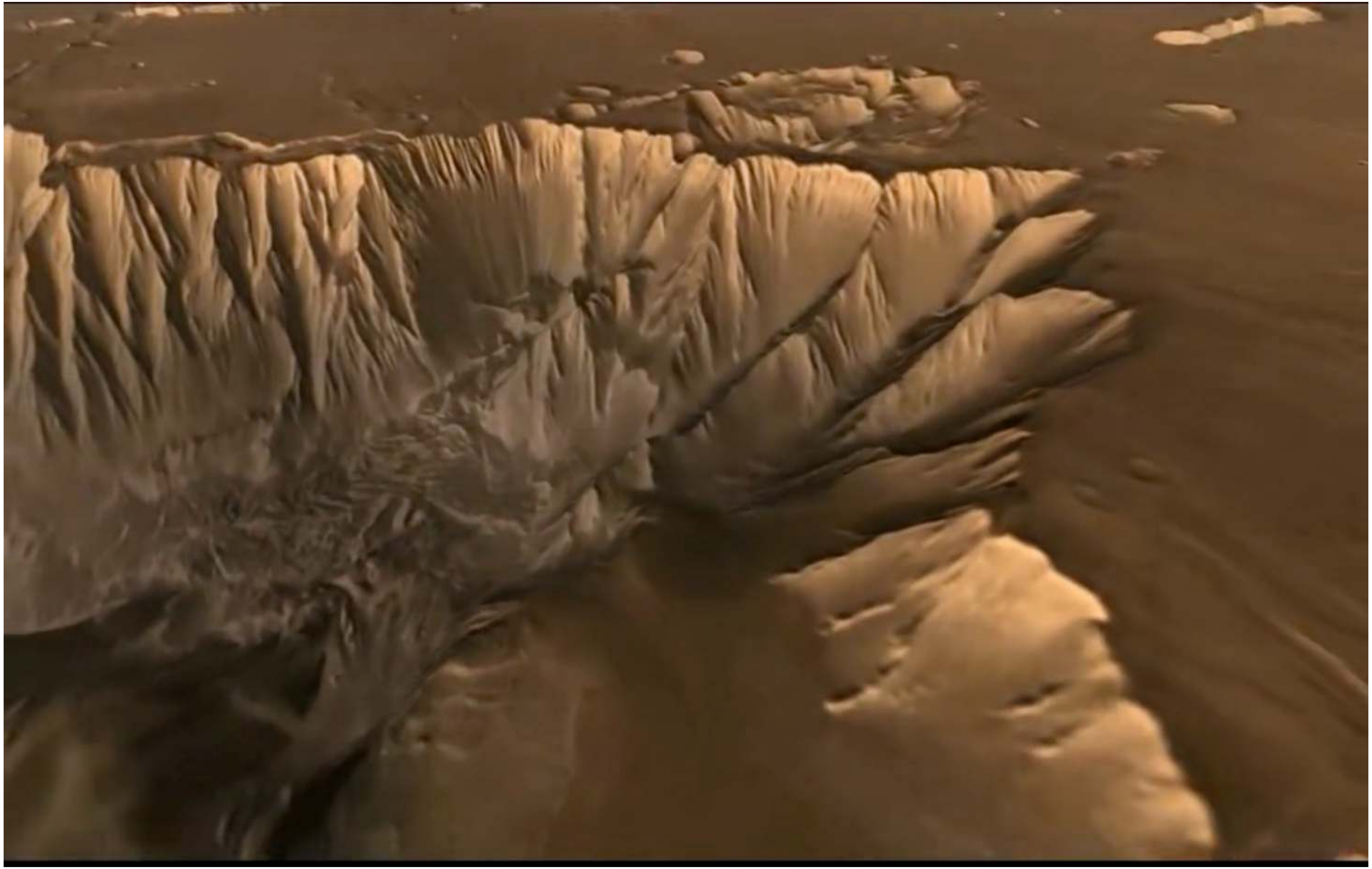
Height field as Bitmap



Rendered

Example for Terrains

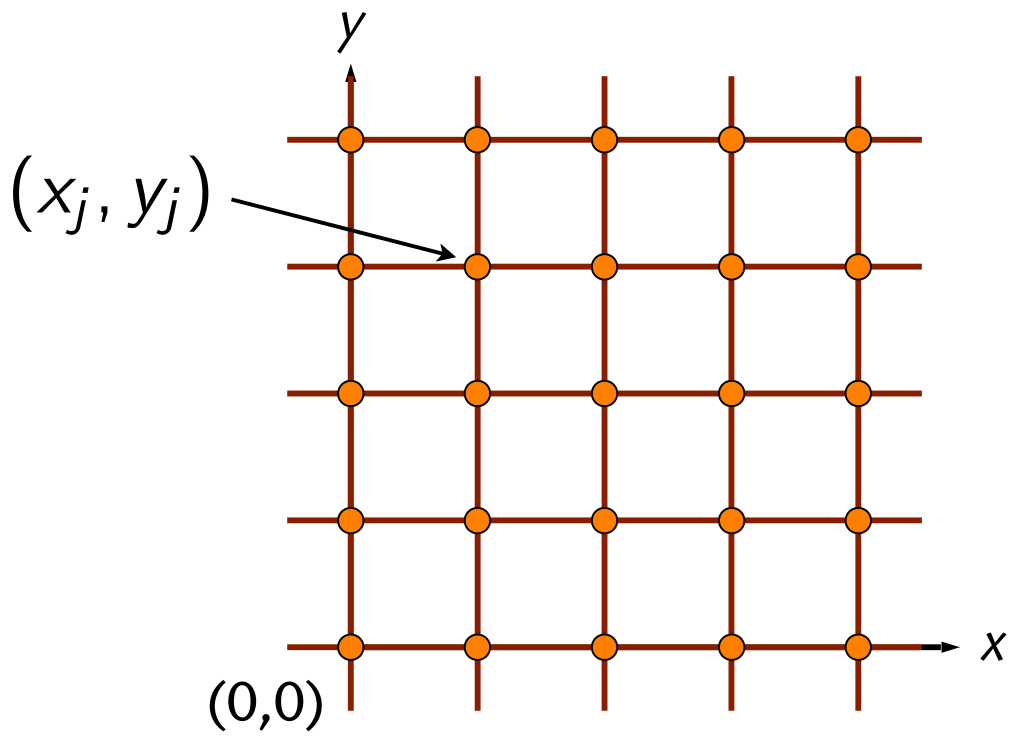
Bonn University

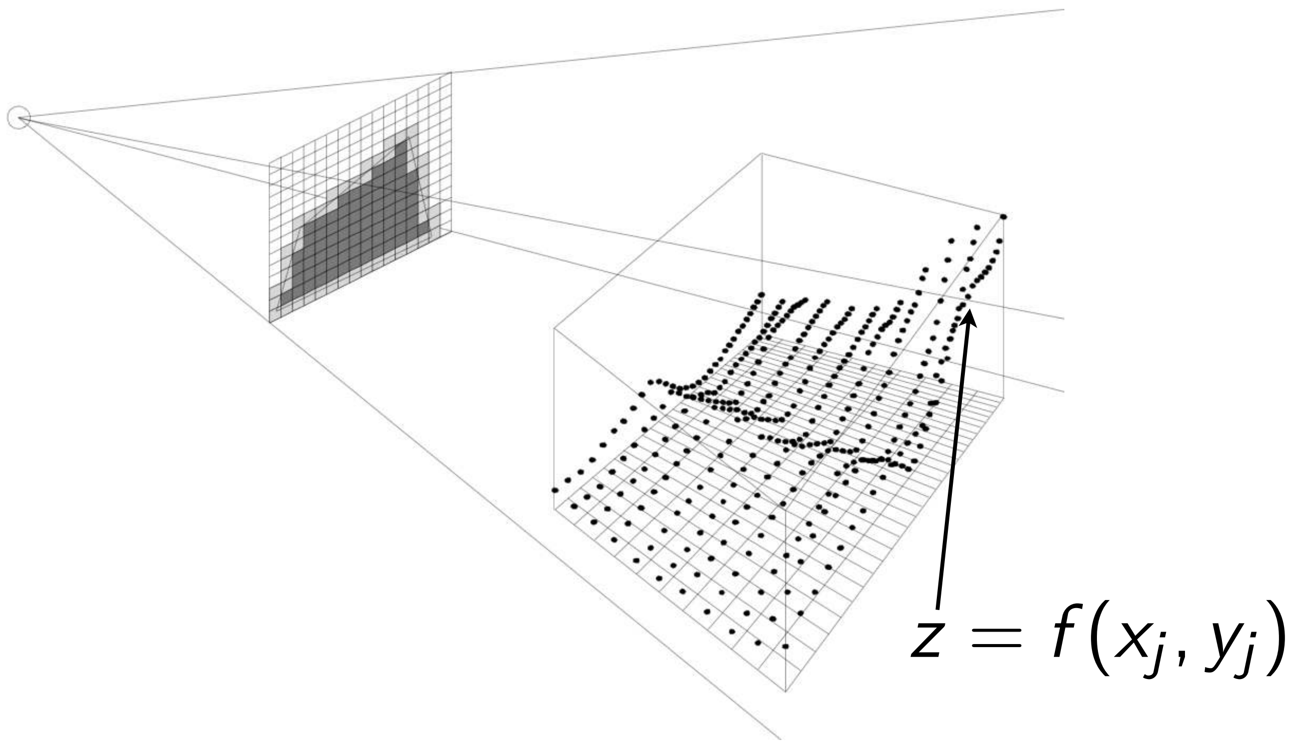


Vallis Marineris, Mars; presented by Phil Christensen, Arizona State University (http://mars.jpl.nasa.gov)

The Situation

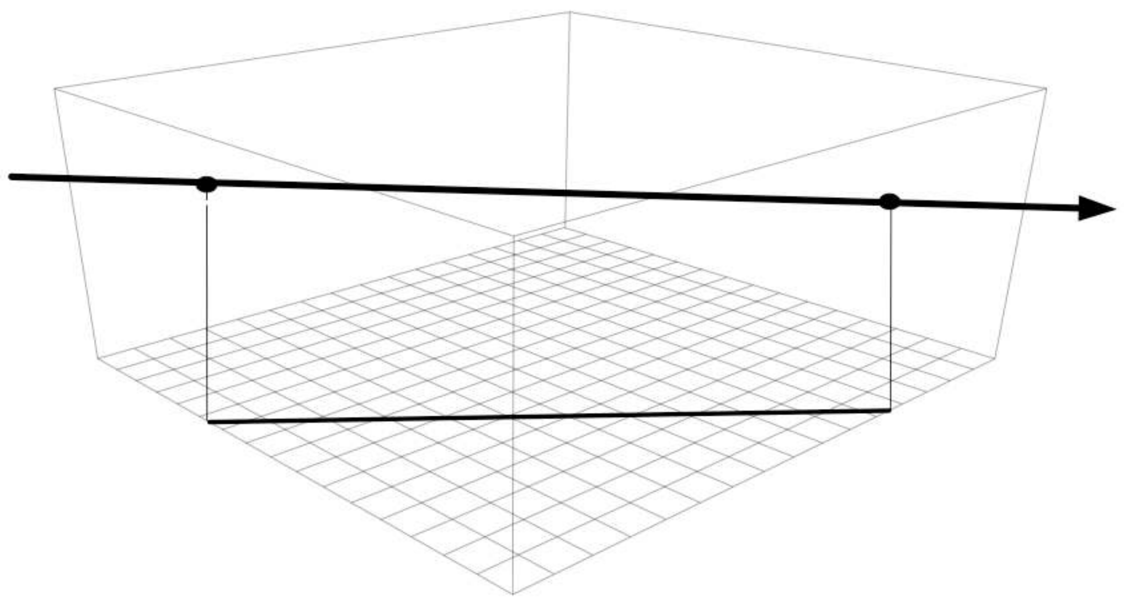
- Given:
  - Ray
  - Array [0…n]x[0…n] with heights
- The naïve method to ray-trace a height field:
  - Convert to $2n^2$ triangles, test ray against each triangle
  - Problems: slow, needs lots of memory
- Goal: direct ray-tracing of a height field represented as 2D array



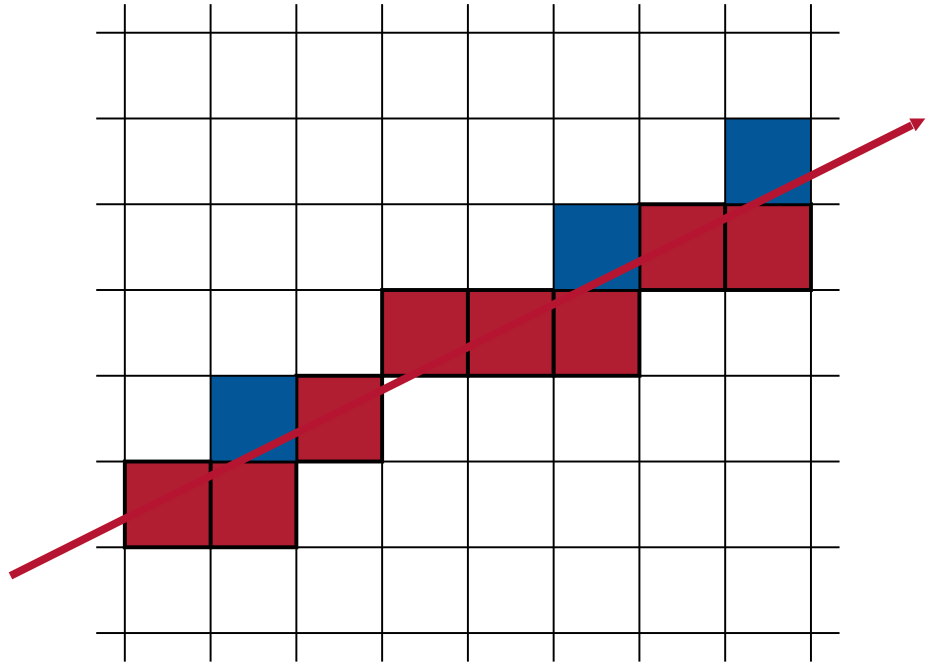


The Method

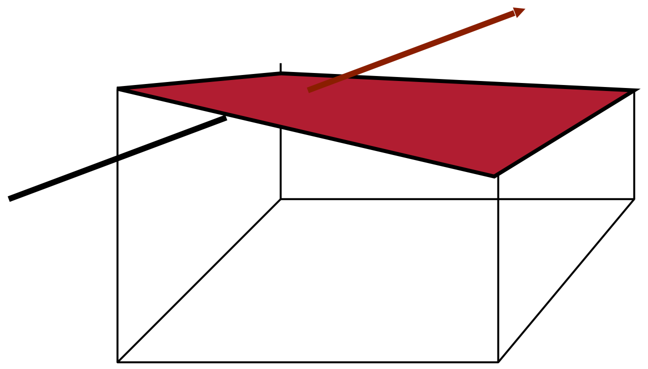
1. Reduce the dimension:
   - Project ray into xz plane



2. Visit all cells that are hit by the ray, starting with the nearest one
   - Notice similarity to scan conversion!
   - Use one of the algorithms from CG1 😀

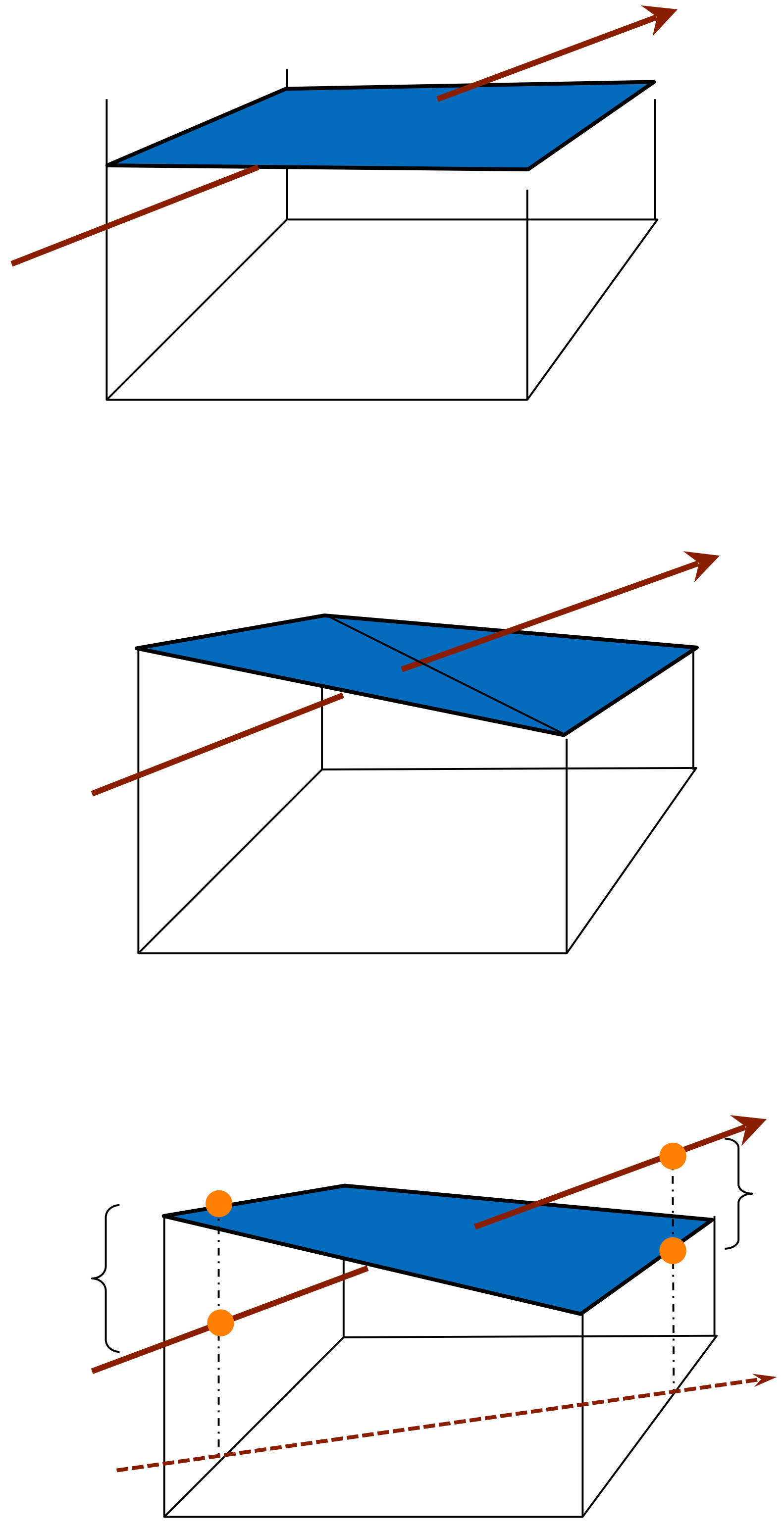


3. Test ray against the surface patch spanned by the 4 corners of the cell



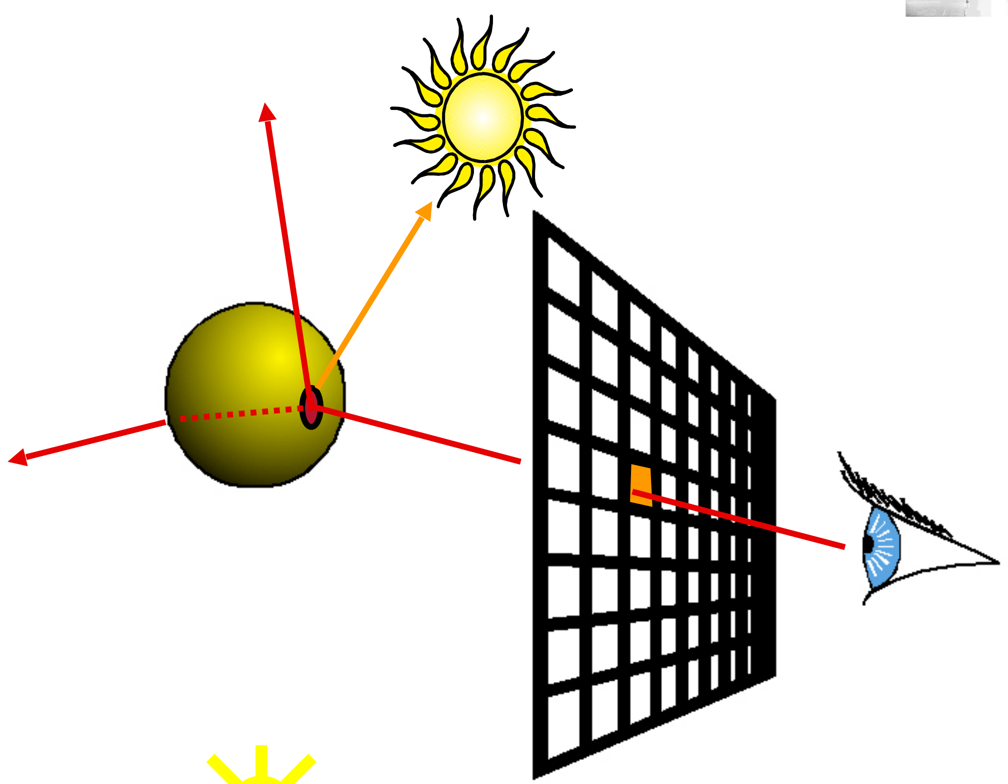
Intersection of Ray with Surface Patch of a Cell

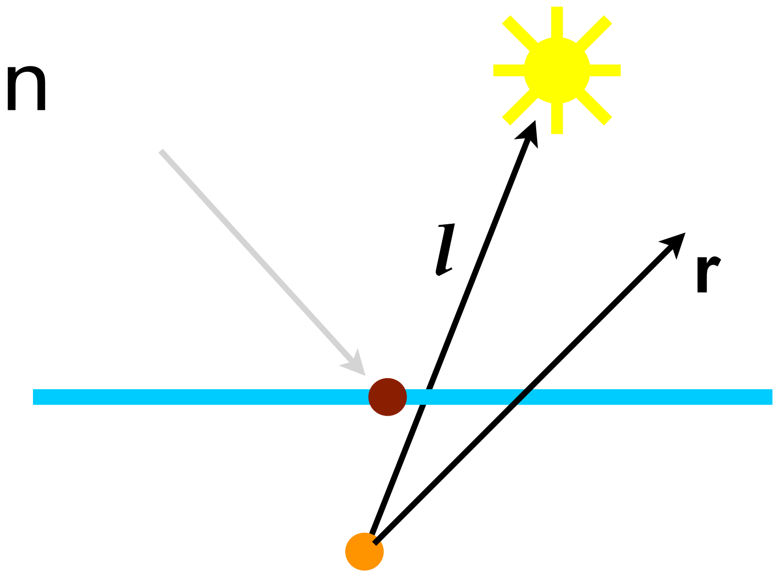
- Naïve methods:
  - "Nearest neighbor":
    - Compute average height of the 4 corner height values
    - Intersect ray with horizontal square of that average height
    - Problem: very imprecise
  - Tessellate by 2 triangles:
    - Construct 2 triangles from the 4 corner points
    - Problem: tessellation is not unique, diagonal edge could produce severe artefact
- Better: *bilinear interpolation*
  - Surface = bilinear interpolation of heights along x and y
    - (The surface is called a *parabolic hyperboloid*)
  - Insert ray equation into bilinear equation of surface ⟶ quadratic equation for line parameter *t*

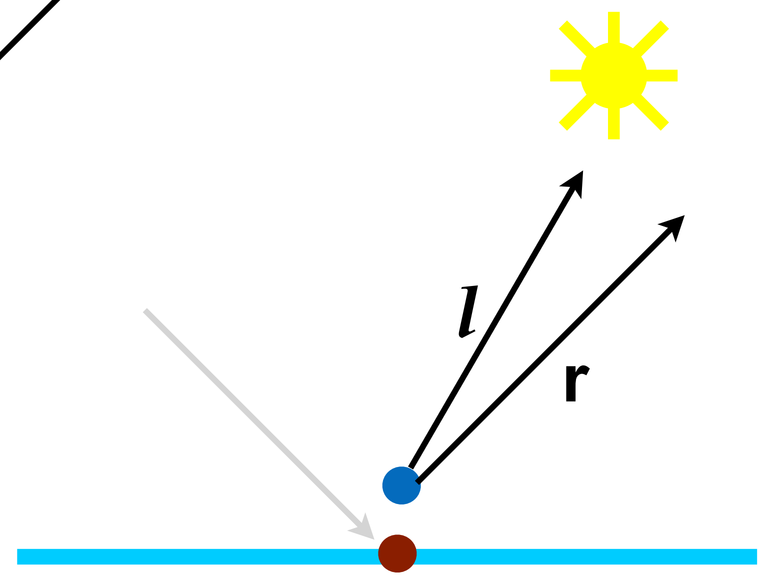


The evil ε

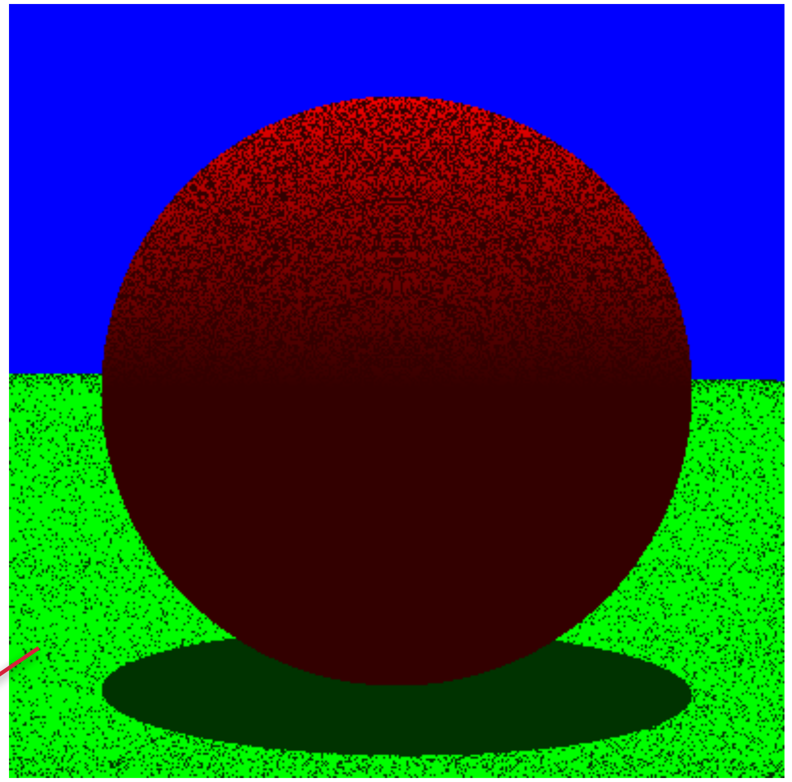
- What happens when the origin of a ray is "exactly" on the surface of an object?
- Remember: floating-point calculations are always *imprecise*!
  - Consequence: in subsequent ray-scene intersection tests, the ray origin might appear to be inside the original object!
  - Further consequence: we get wrong further intersection points!
- "Solution": move the origin of the ray by a small ε along the direction of the (new) ray

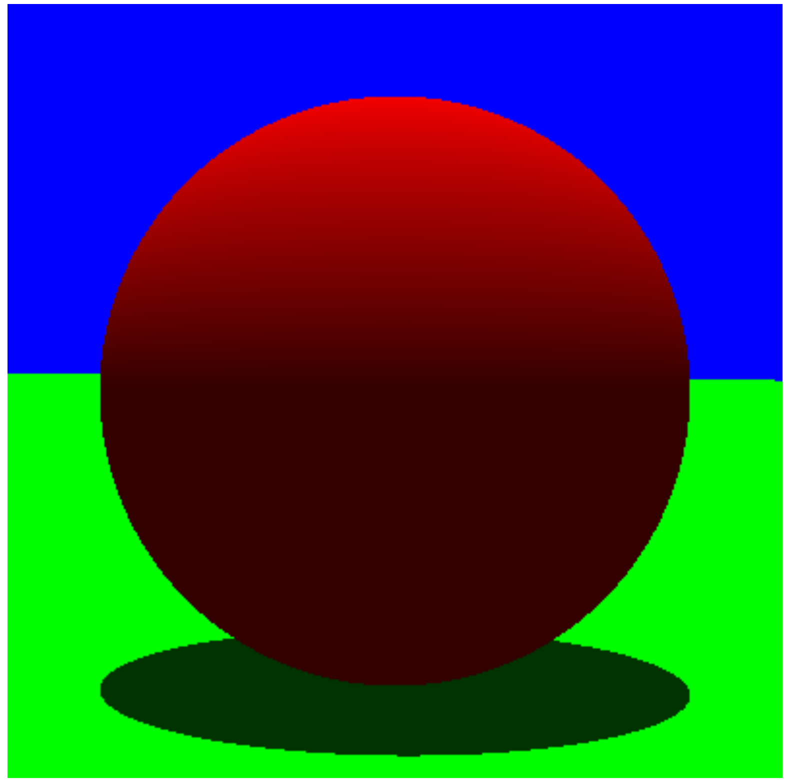






More Glitch Pictures

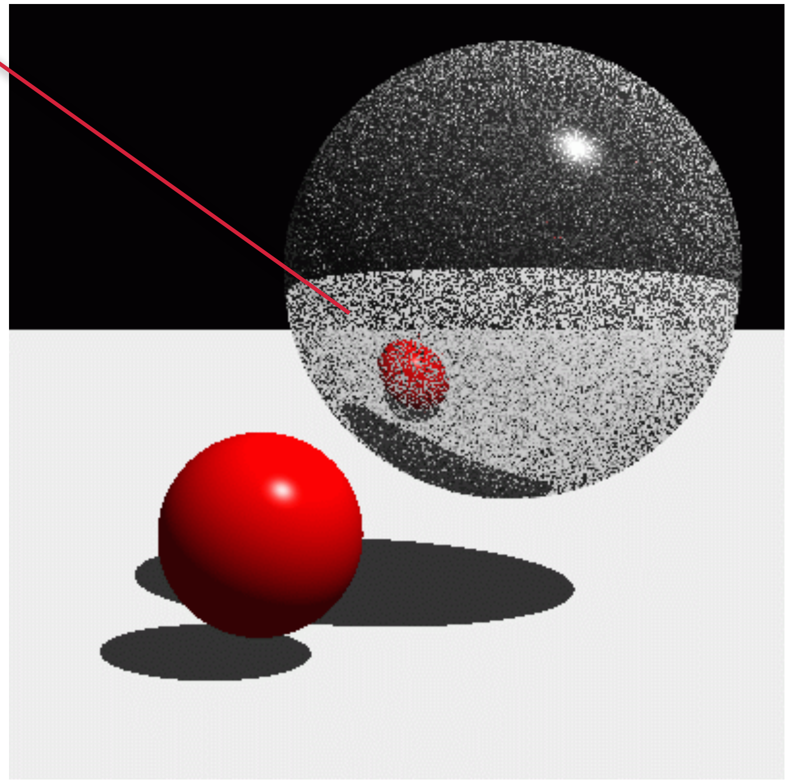


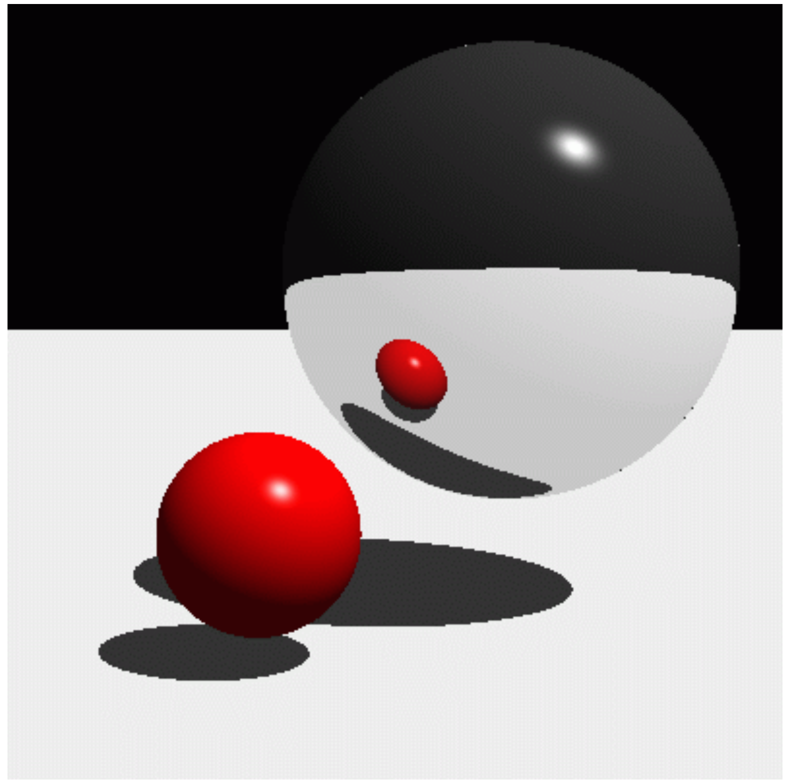


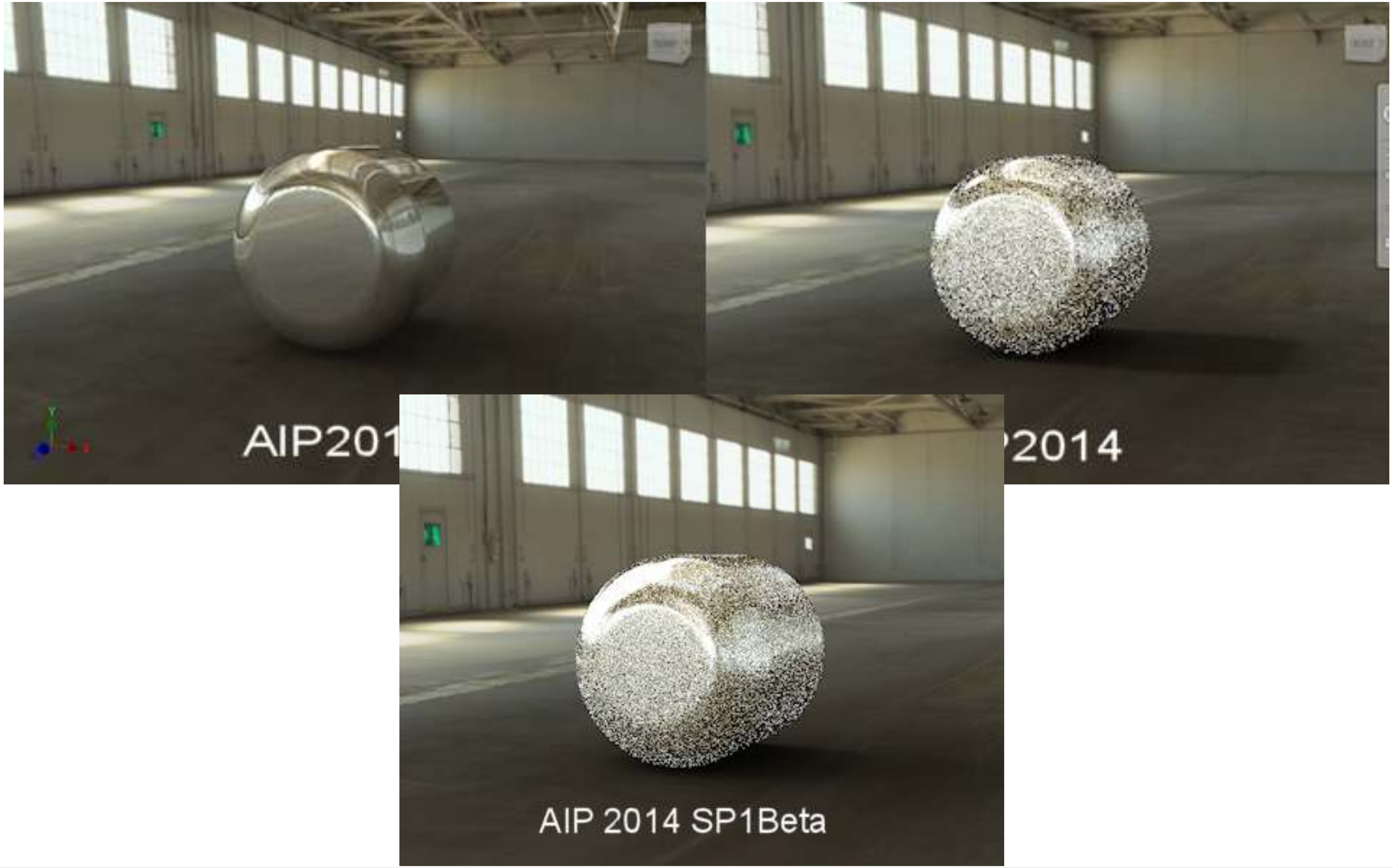
"Speckles"

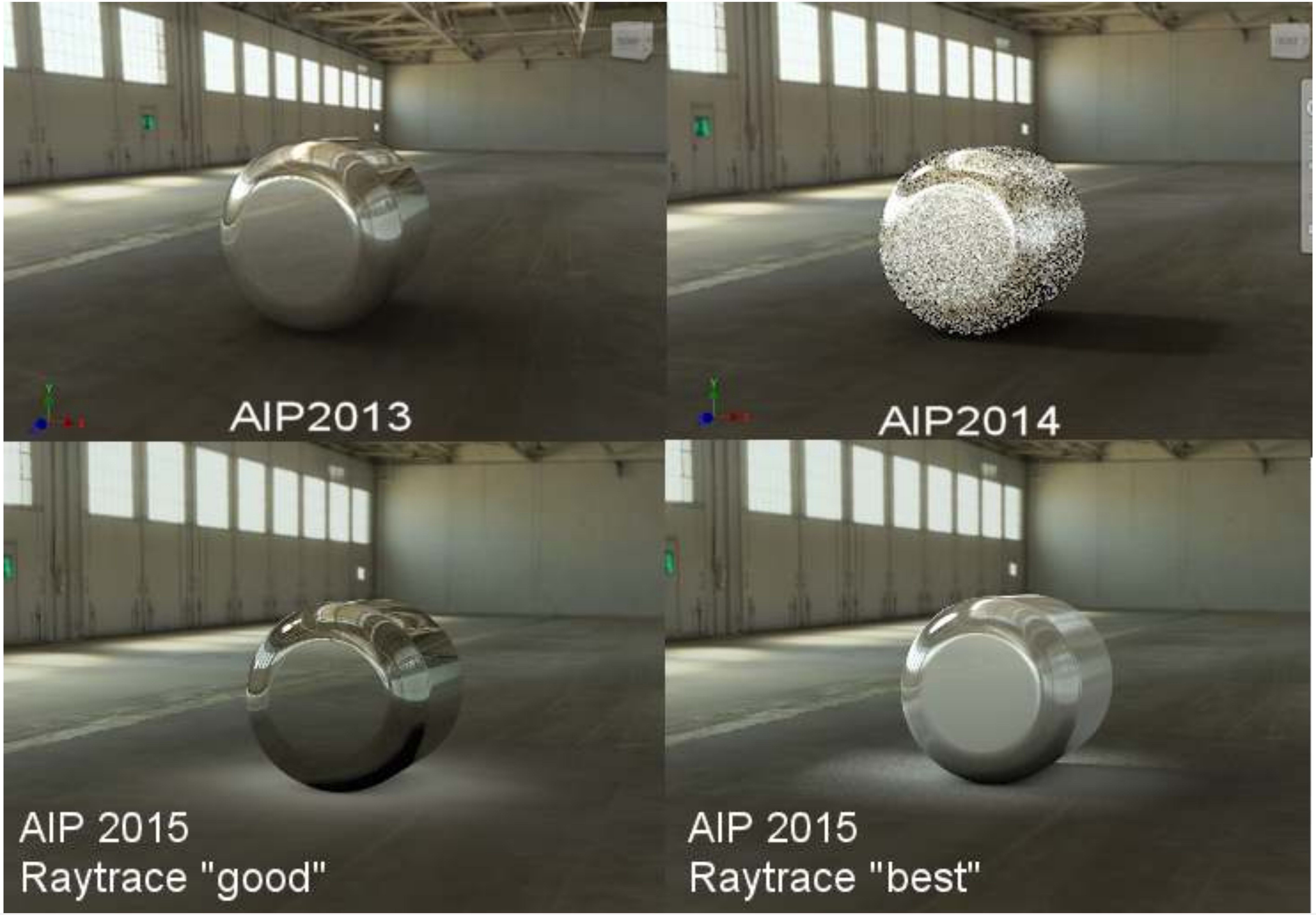
Without epsilon

With epsilon






AIP201
2014
AIP 2014 SP1Beta


AIP2013
AIP2014
AIP 2015
Raytrace "good"
AIP 2015
Raytrace "best"

Advantages & Disadvantages

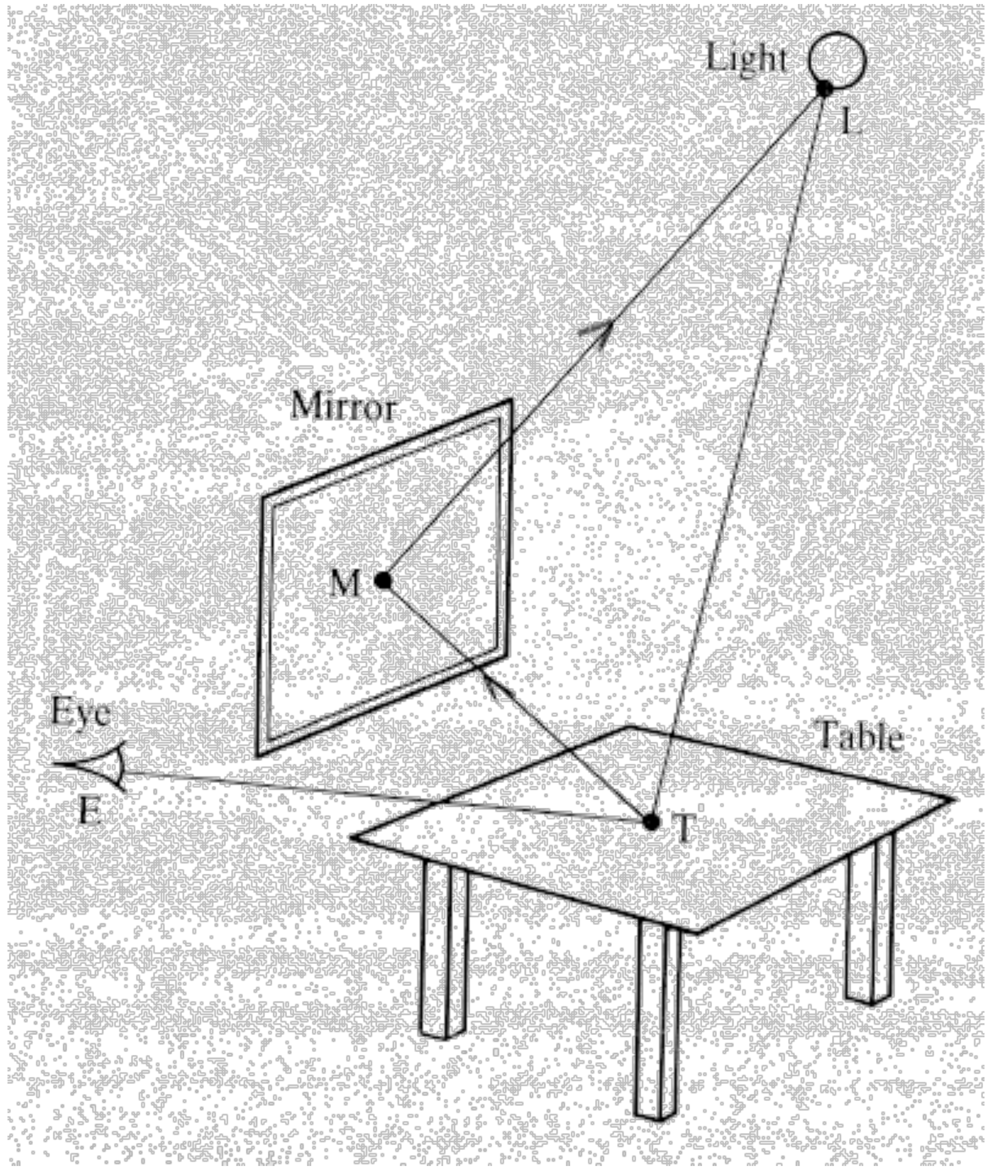
- Scan conversion:
  - Fast (for a number of reasons)
    - Well-suited for real-time graphics
  - Supported by all graphics hardware
  - Only heuristic solutions for shadows and transparent objects
  - No interreflections
- Raytracing:
  - Offers general and simple (in principle) solution for global effects, such as shadows, interreflection, transparent objects, etc.
  - Much slower (unless you cast only primary rays)
  - Not directly supported by most graphics hardware (is currently changing)
  - Difficult to achieve real-time rendering

Other Advantages of Ray-Tracing

- Shines with scenes that contain lots of glossy/shiny surfaces and transparent objects
- Fairly easy to incorporate other object representations (e.g., CSG, smoke, fluids, ...)
  - Only prerequisite: must be possible to compute the intersection between ray and object, and to compute the normal at the point of intersection
- No separate clipping step necessary

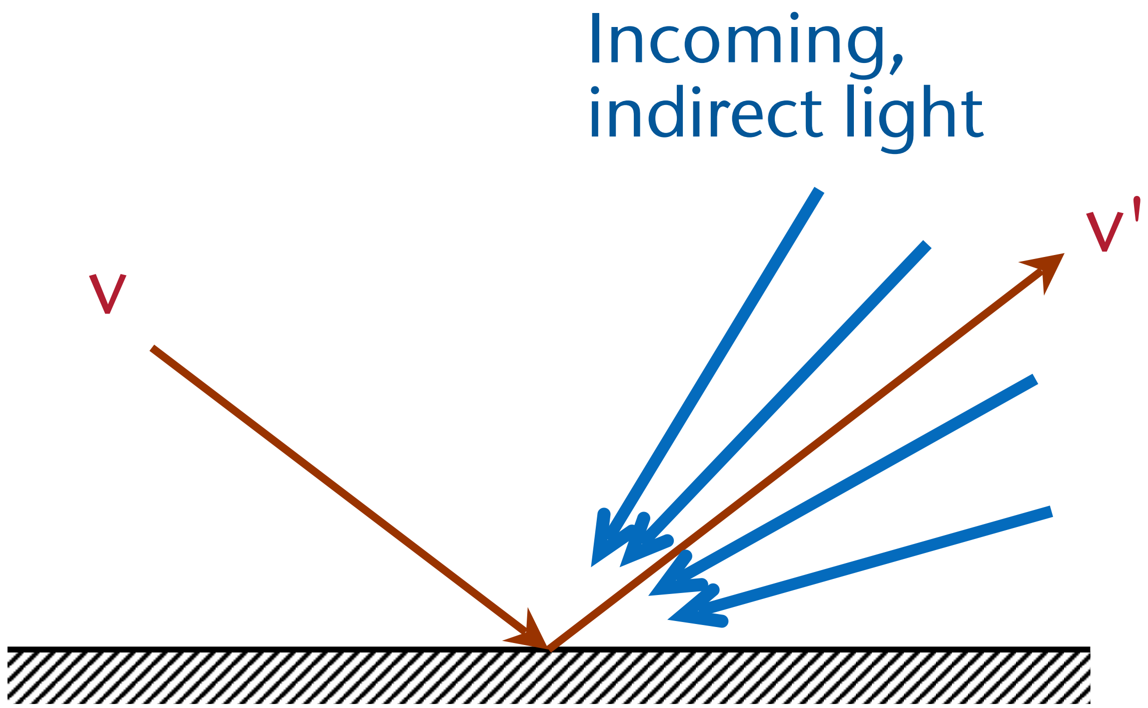
Limitations of (Simple) Whitted-Style Ray-Tracing

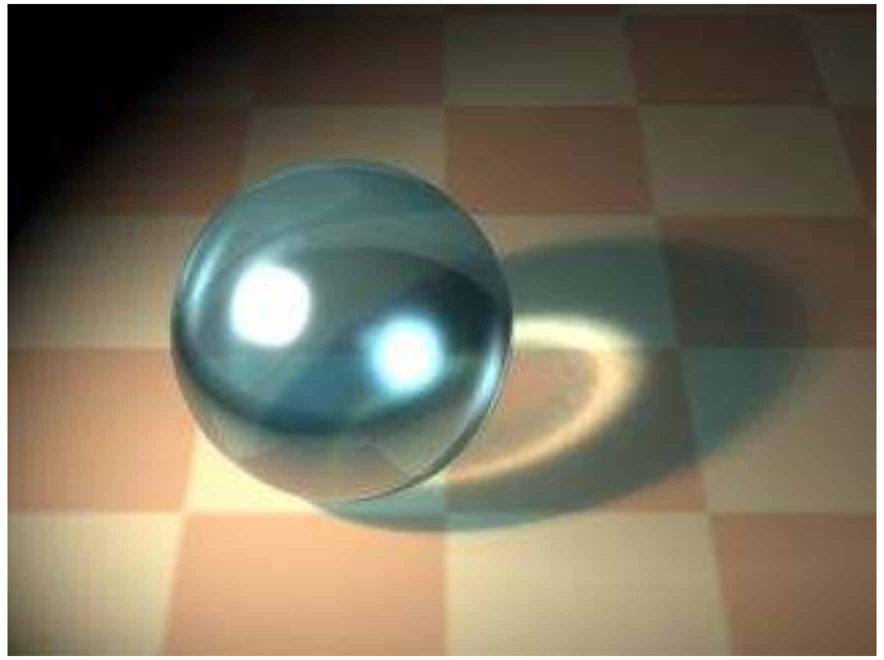
- No indirect lighting (e.g., by mirrors)
- No soft shadows (because only point light sources are modeled)
- Only specular (ideal) reflections
- Only perfect (specular) refractions
- With each camera movement, the complete ray tree must be recomputed, although an object's diffuse shading does not depend on the camera's position
- For all of these disadvantages, a number of remedies have been proposed …

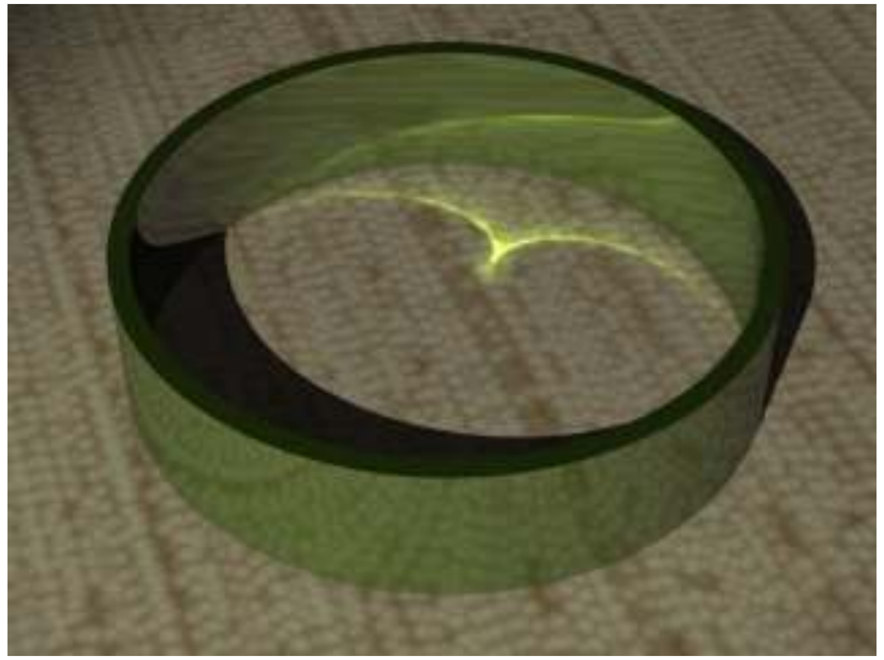


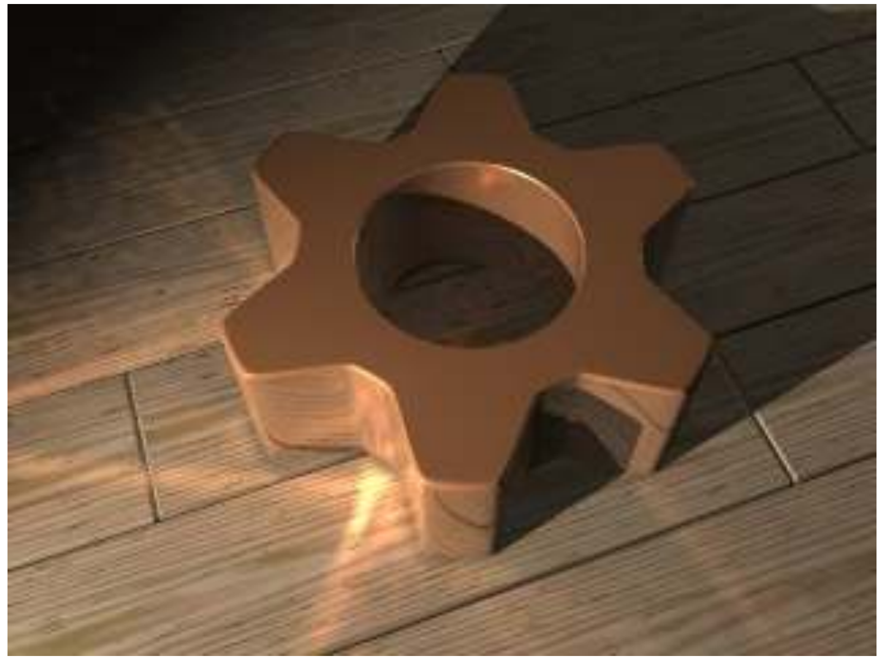
Example for the Problem of (Missing) Indirect Lighting: Caustics

- Caustics = reflected/transmitted light is concentrated in a point or, possibly curved, line on the surface of another object
- Problem:
  - Ray-tracing follows light paths *backwards*
  - Simple ray-tracing follows only *one* path







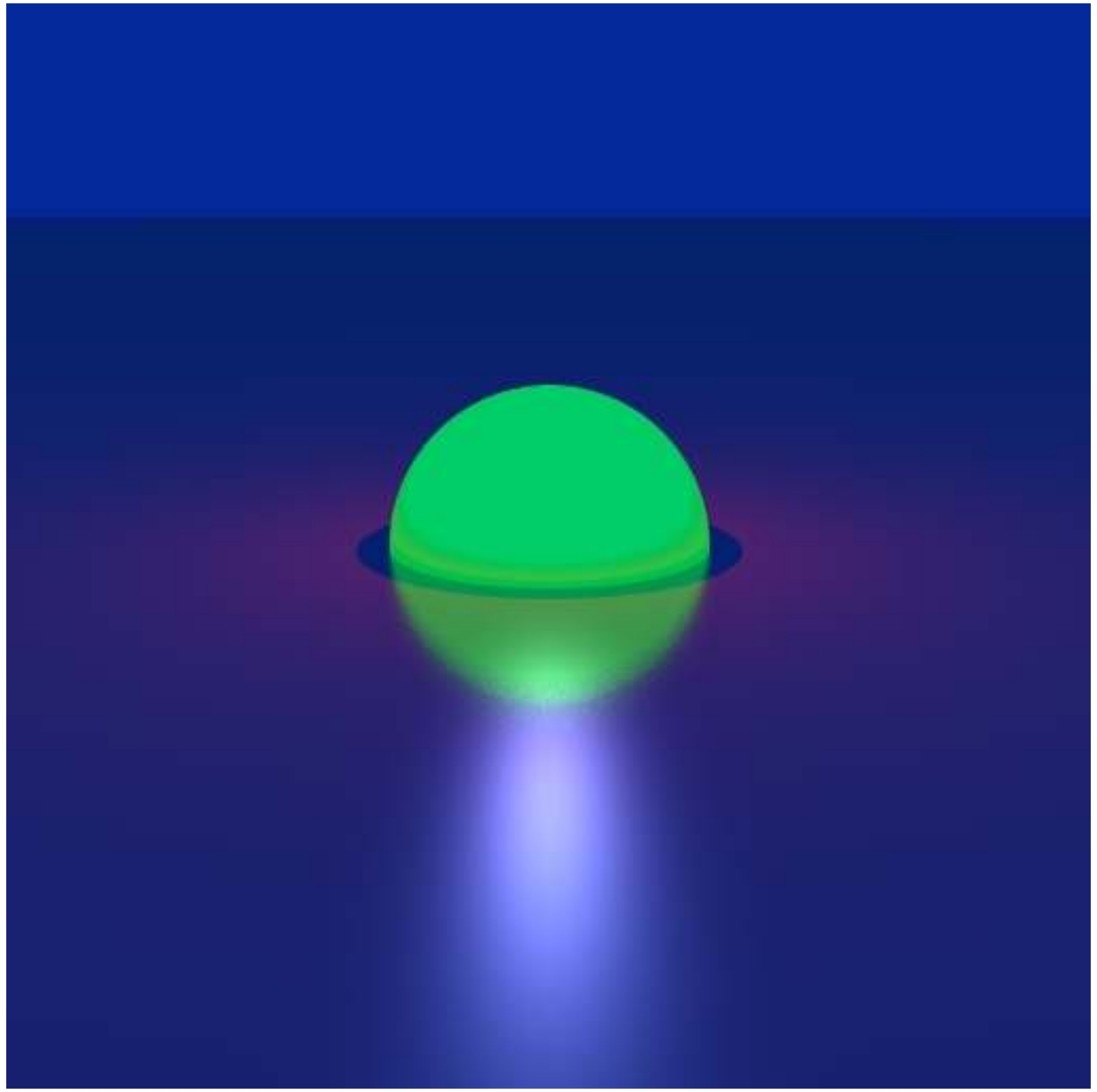


Other Problem: Aliasing

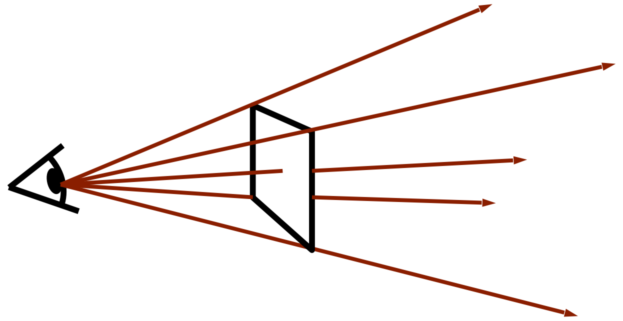
- One ray per pixel → causes typical aliasing artefacts:
  - "Jaggies"
  - Moiré effects

Solution: Distribution Ray Tracing

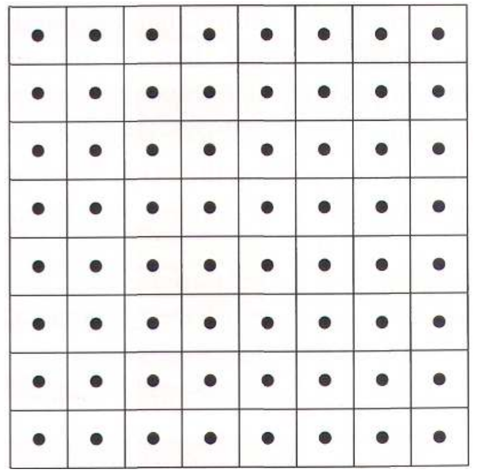
- Simple modification of ray-tracing to achieve
  - Anti-aliasing
  - Soft shadows
  - Depth-of-field
  - Shiny/glossy (specular-diffuse) surfaces
  - Motion blur
- Was proposed under a different name:
  - "Distributed Ray Tracing"
  - Don't use that name ("distributed" = verteilt)



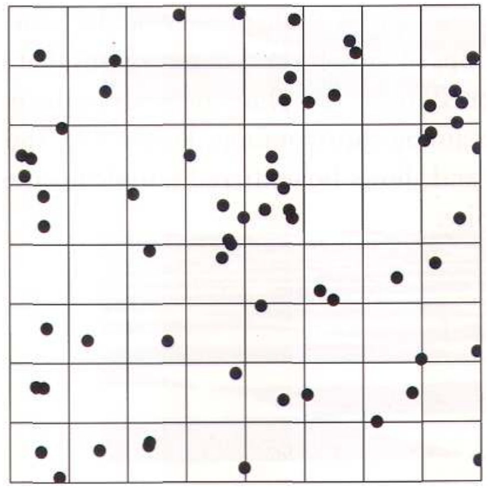
Anti-Aliasing with Ray-Tracing (a Quick Tour of Sampling)



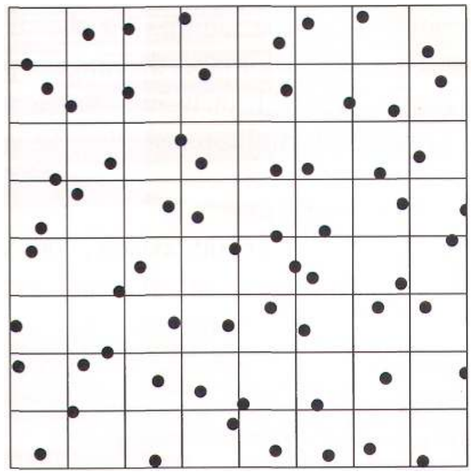
- Shoot many rays per pixel, instead of just one, and average retrieved colors (hence "distribution")
- Four methods for constructing the rays:



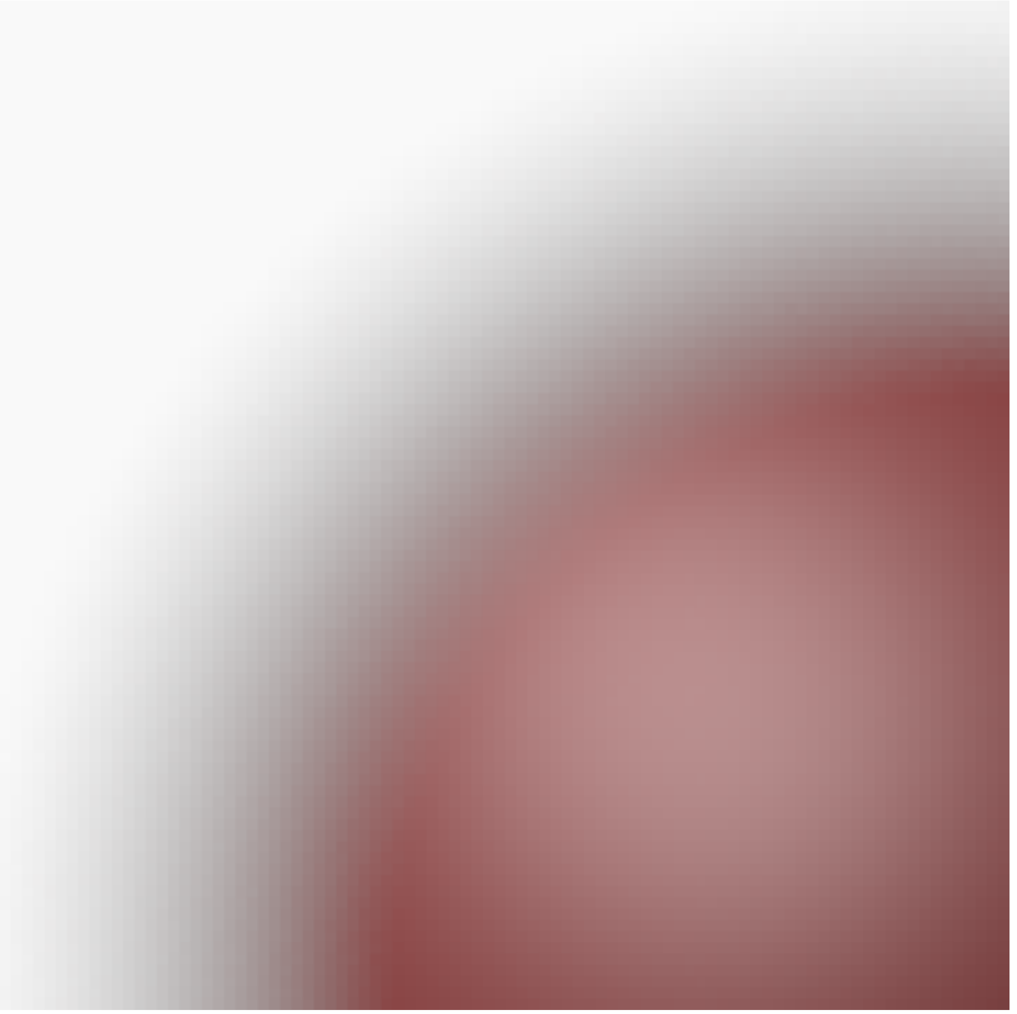
1. Regular sampling
   - Perhaps problems with Moiré patterns
2. Random sampling
   - Can contain arbitrarily big "holes" and arbitrarily close samples
   - Might result in noisy images



3. Stratification (aka jittered grid): combination of regular and random sampling, e.g., by placing a grid over the pixel, and picking one random position per cell



Results in a Real Raytraced Image

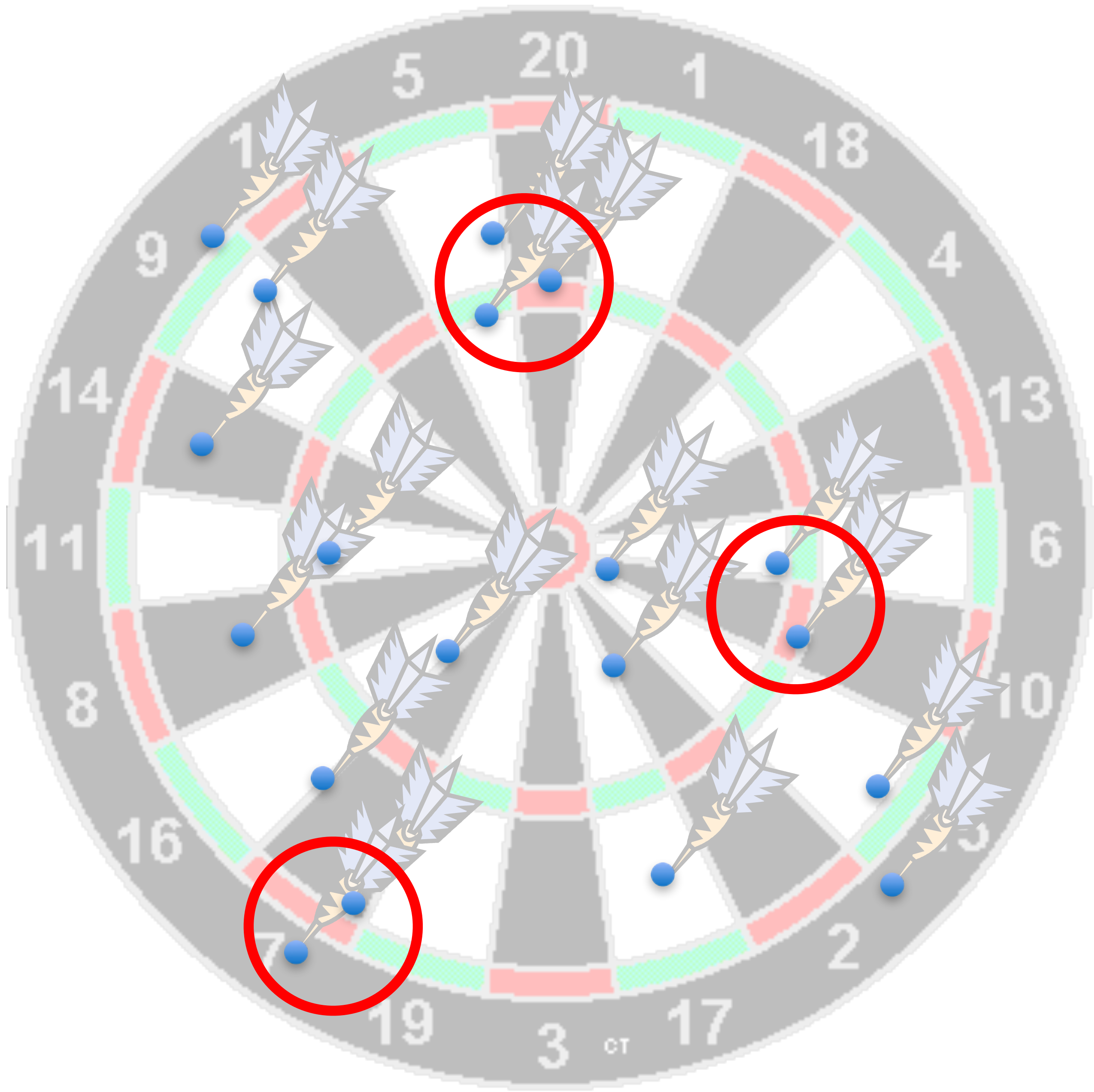


Reference

Random

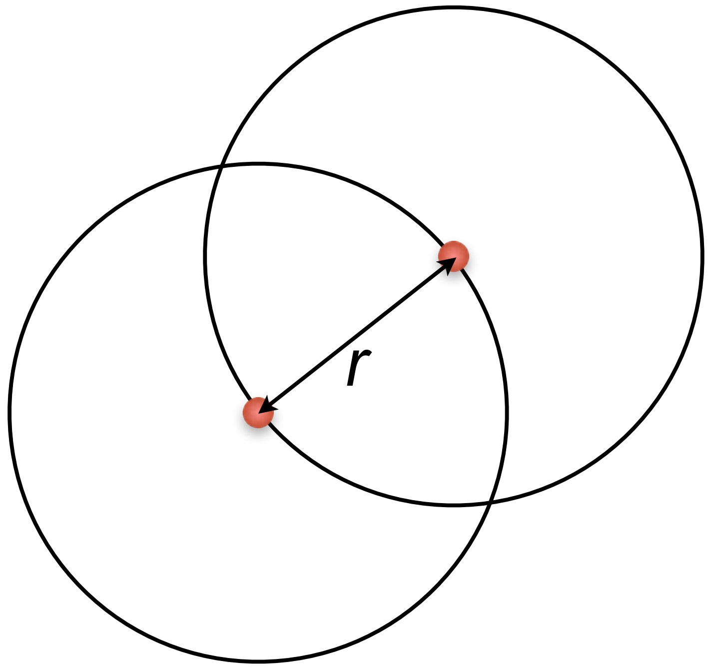
Stratified (jittered)

Dart Throwing

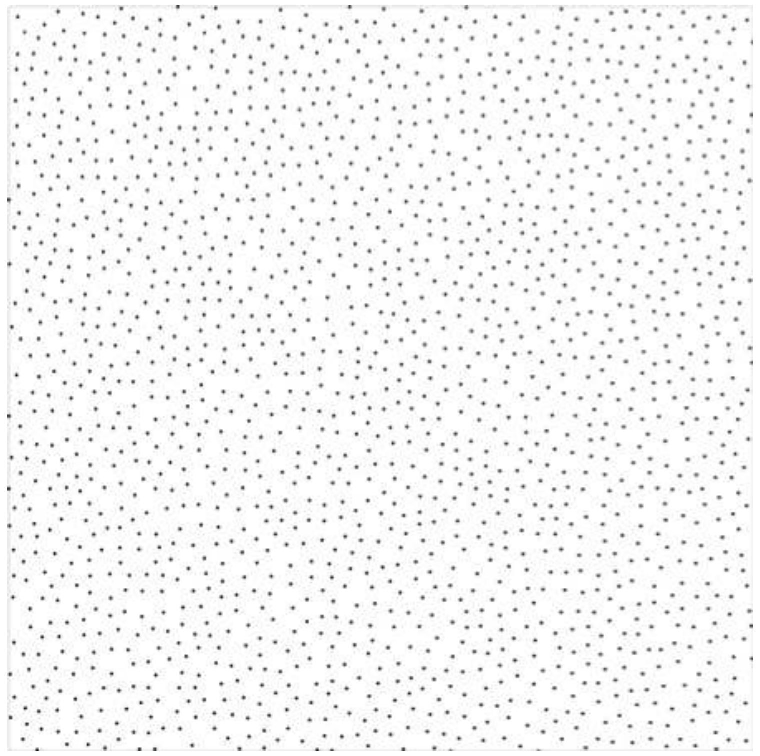


Poisson Disk Sampling

- Definition:
  A set of sample points (in a specific domain) is a Poisson disk sampling with radius *r* iff
  1. it is "as random as possible"; and,
  2. it satisfies the Poisson disk criterion, i.e.,
     no two points are closer than the minimum distance *r*.



- Example Poisson sampling:



- Formal definition:
  The point set $S = \{ \mathbf{p}_i \}$ is a (maximal) Poisson disk sampling of some given domain D (e.g., a polygon), iff the following 3 conditions hold:
  1. Empty disk property: $\forall \mathbf{p}, \mathbf{q} \in S, \mathbf{p} \neq \mathbf{q} : \|\mathbf{p} - \mathbf{q}\| \geq r$
  2. Maximality: $\forall \mathbf{x} \in D \; \exists \mathbf{p} \in S : \|\mathbf{p} - \mathbf{x}\| < r$
  3. Bias-free: the likelihood of a (new) sample being inside any subdomain $D' \subseteq D$ is proportional to the area of the subdomain, provided D' is completely outside all prior samples' disks (this is uniform sampling from the uncovered area)
- Many algorithms for Poisson sampling construction relax one of these conditions

- Importance of Poisson disk sampling:
  - Best quality for *N* samples [Cook 1986]
  - Poisson disk distribution occurs in natural objects (e.g., retina cells)
  - Equivalent to "blue noise" spectrum = void in low frequency range, noise in high frequencies

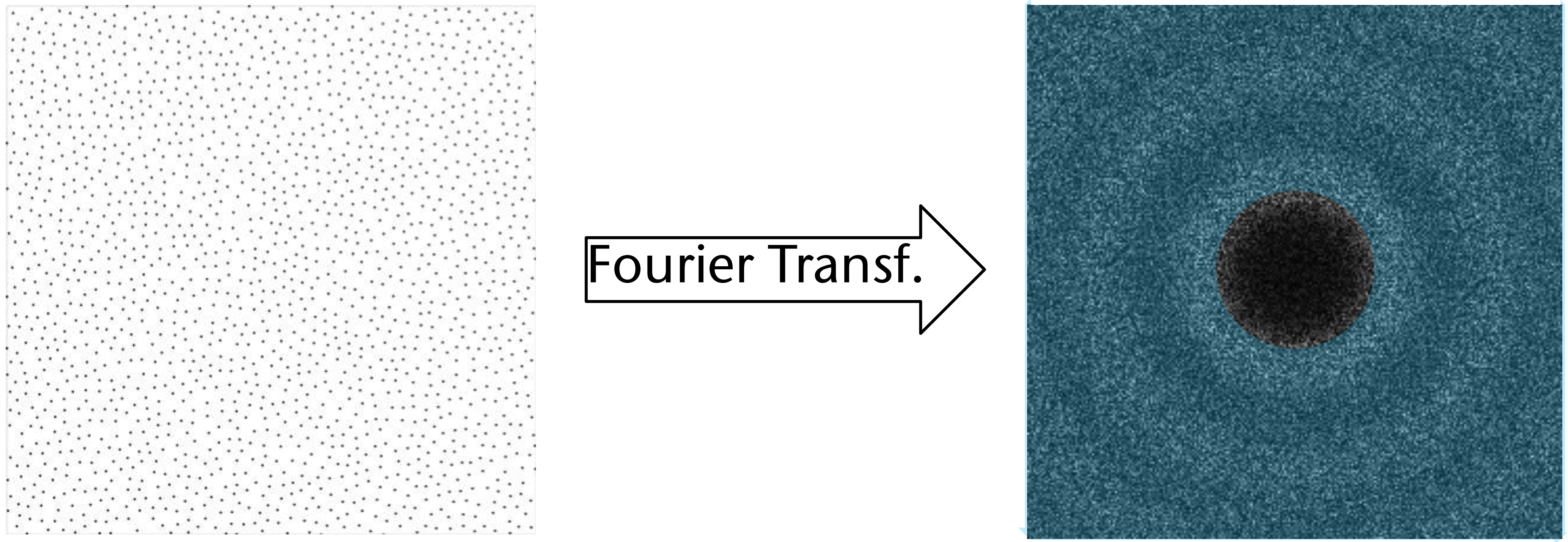
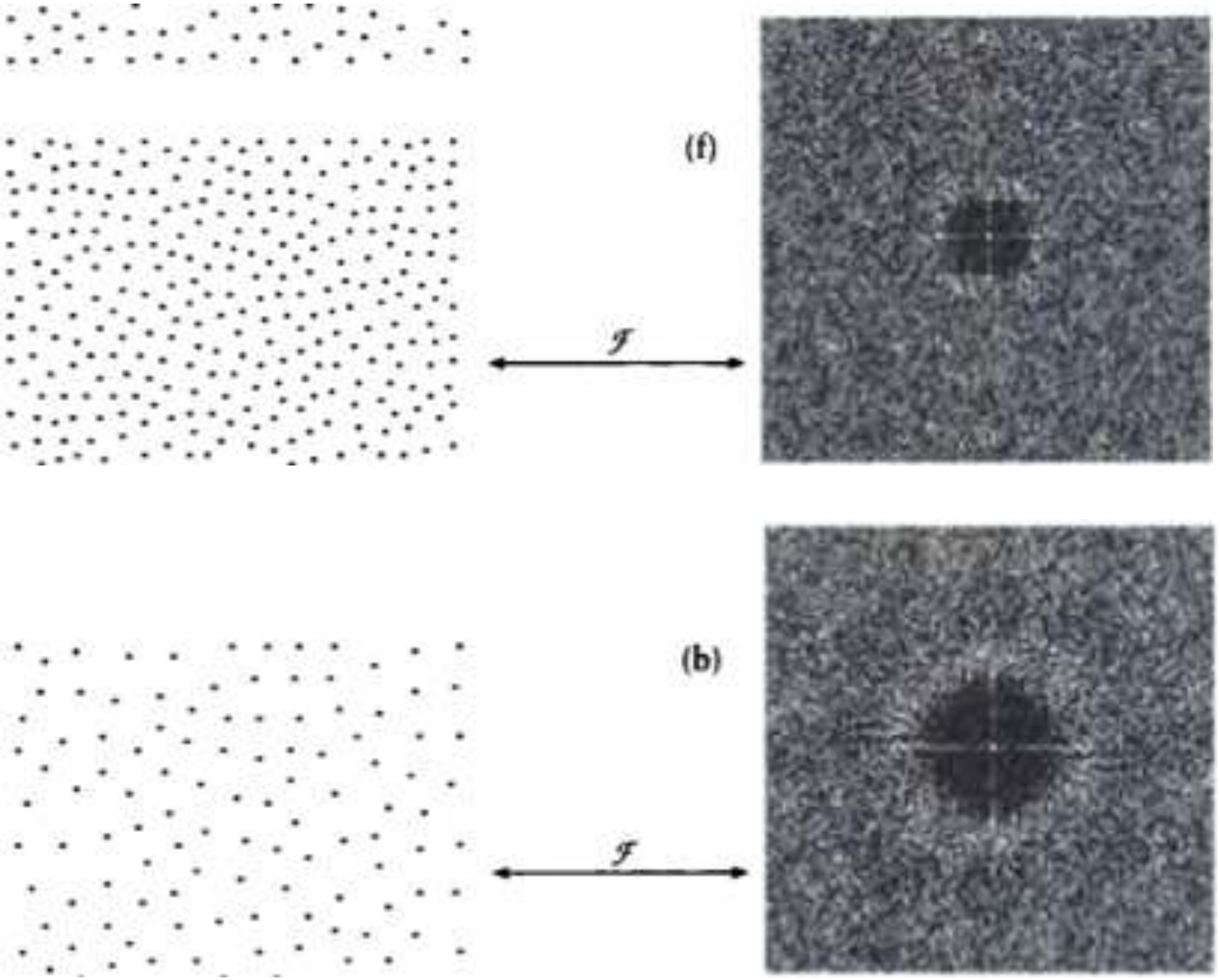
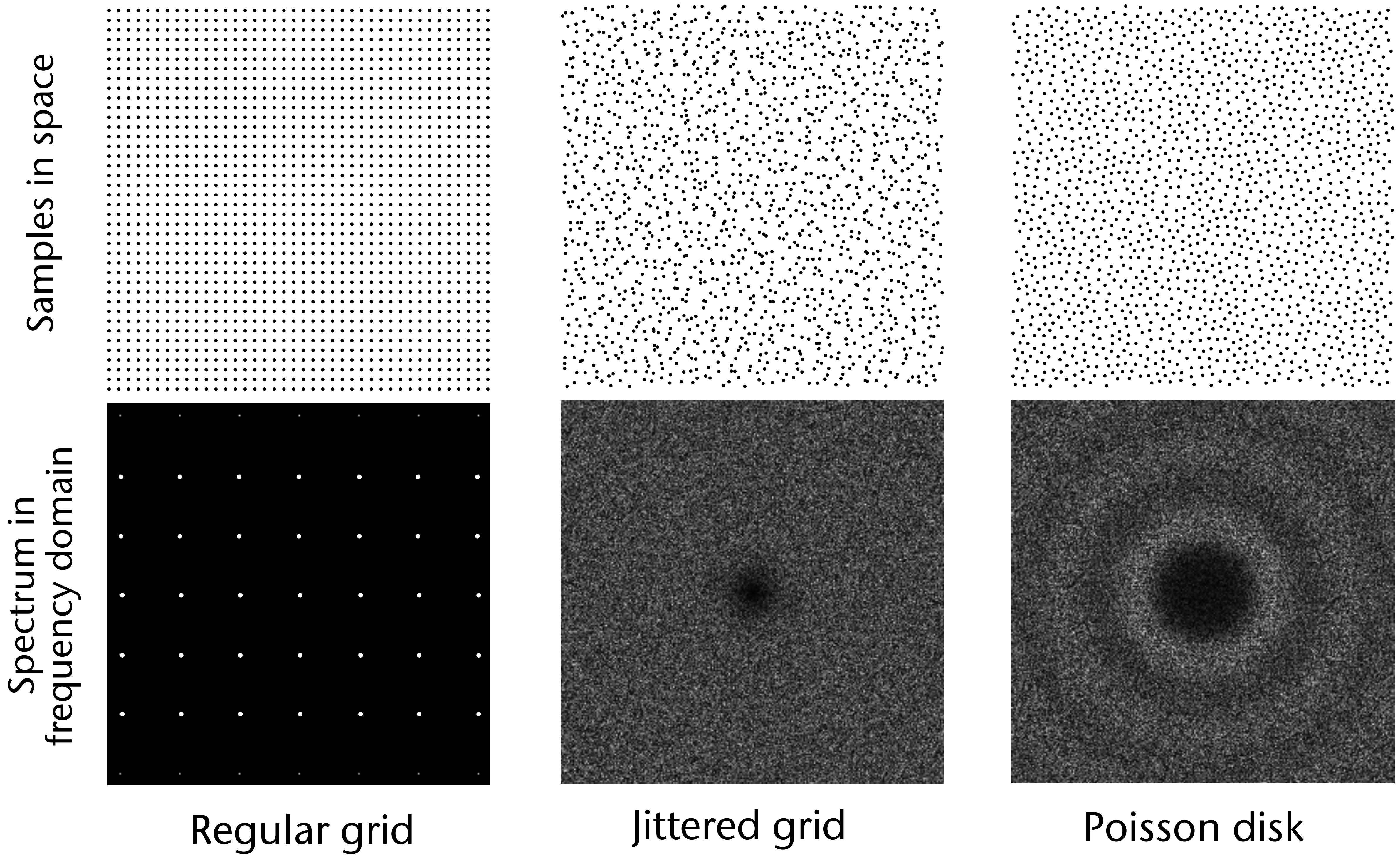


Image domain Frequency domain

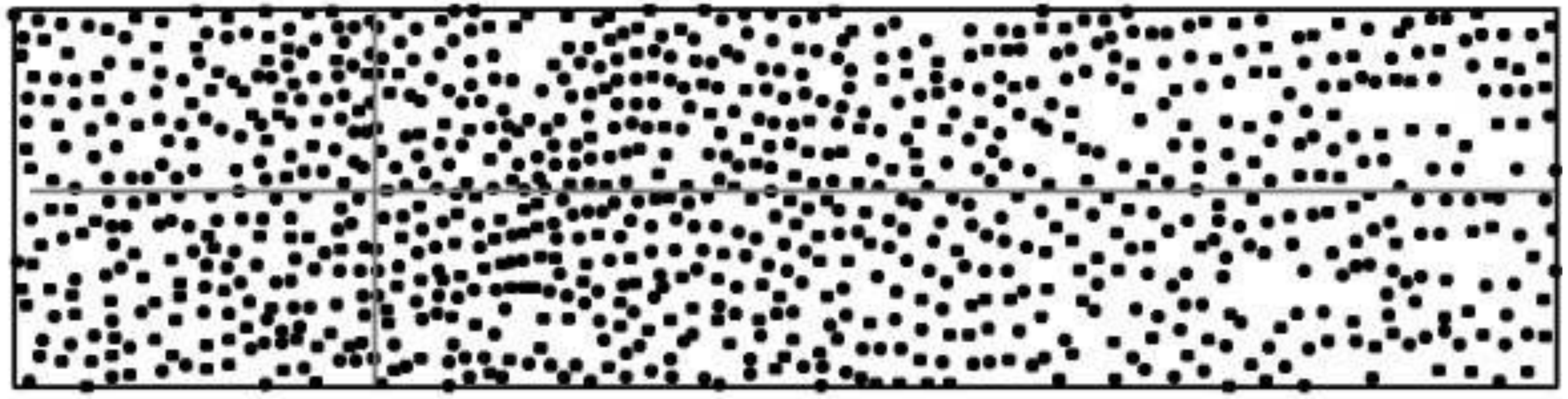
Relationship between sampling density and "ring" of artefact-free frequencies



Comparison of Spectrums of Different Sampling Patterns



Found in nature, too

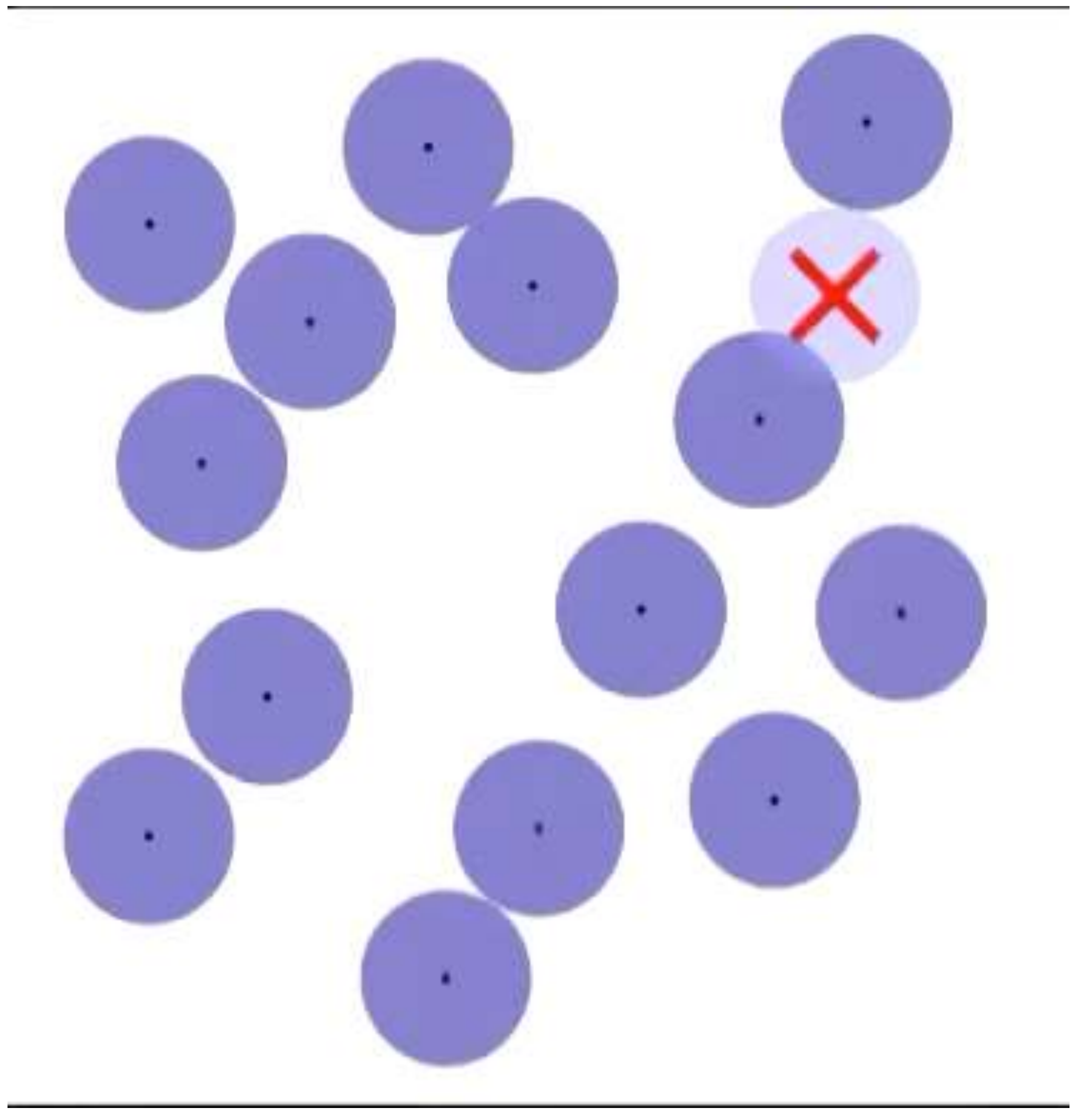


Medium-long wavelength cones of a squirrel monkey

Generating Poisson-Disk Samplings

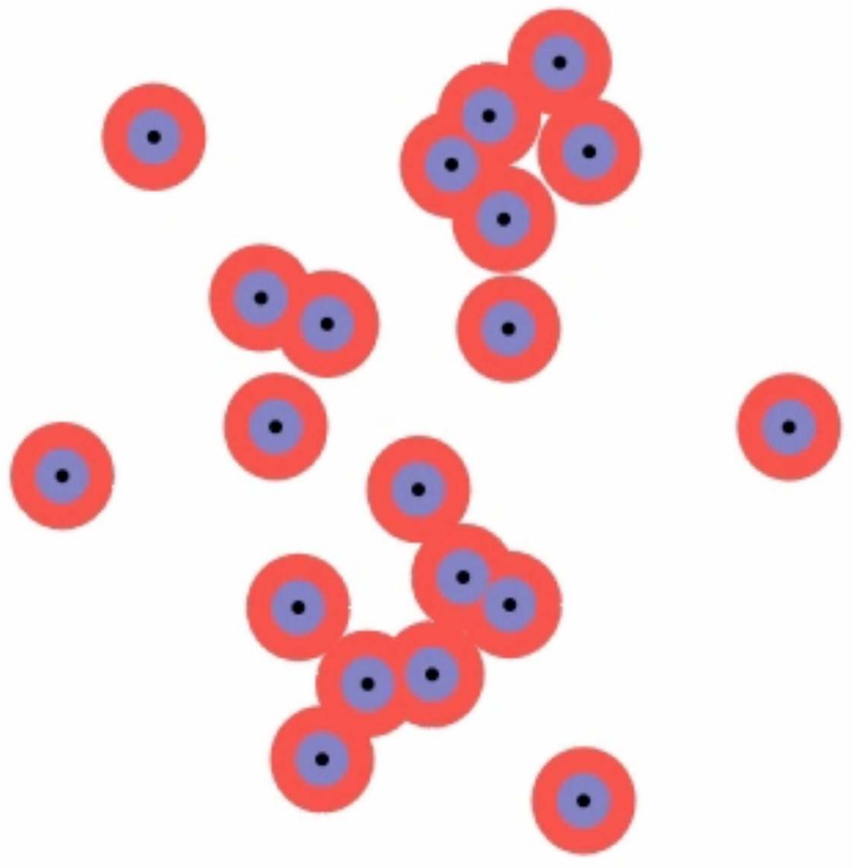
- The dart-throwing algorithm (ground truth, Cook 1986):

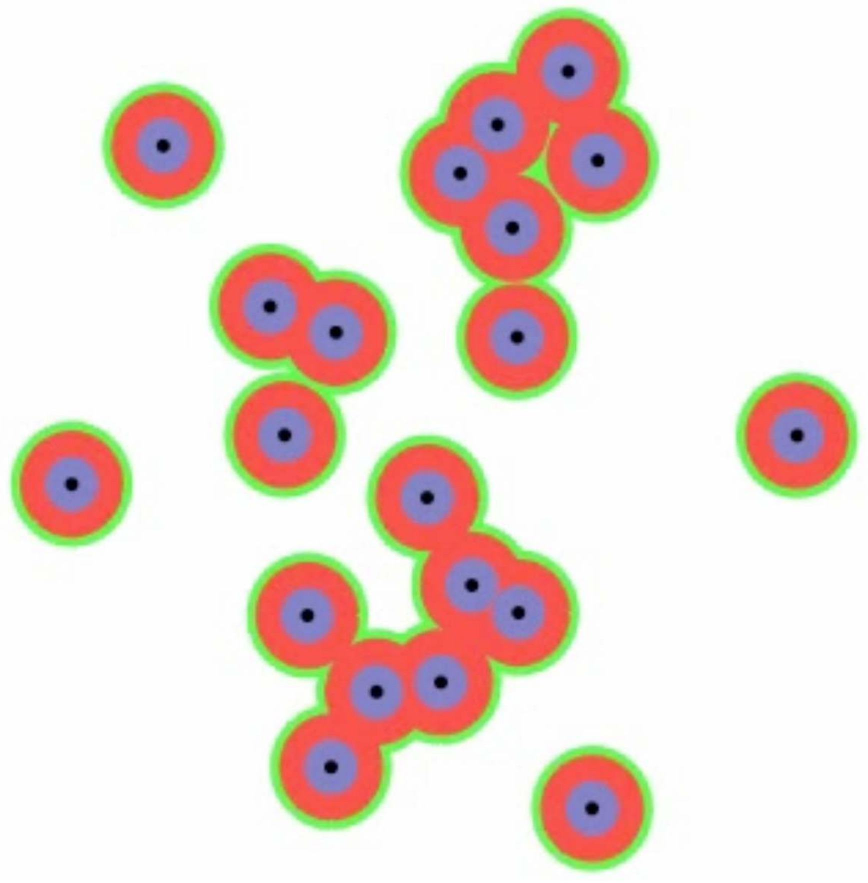
```
i = 0
while i < N:
  pi = ( rand(0,1), rand(0,1) )
  d = min. distance of pi to all other p0, ..., pi-1
  if d > r:
    i ++                        // accept the sample
```



- Problem: it is very slow!

- A possible remedy:
  - Represent "forbidden" (red) and allowed (white) region
  - Throw darts only in white region
- Problem: representation of the white region is very complex
- Idea:
  - Generate new points only on *permissible boundary* of current sample set



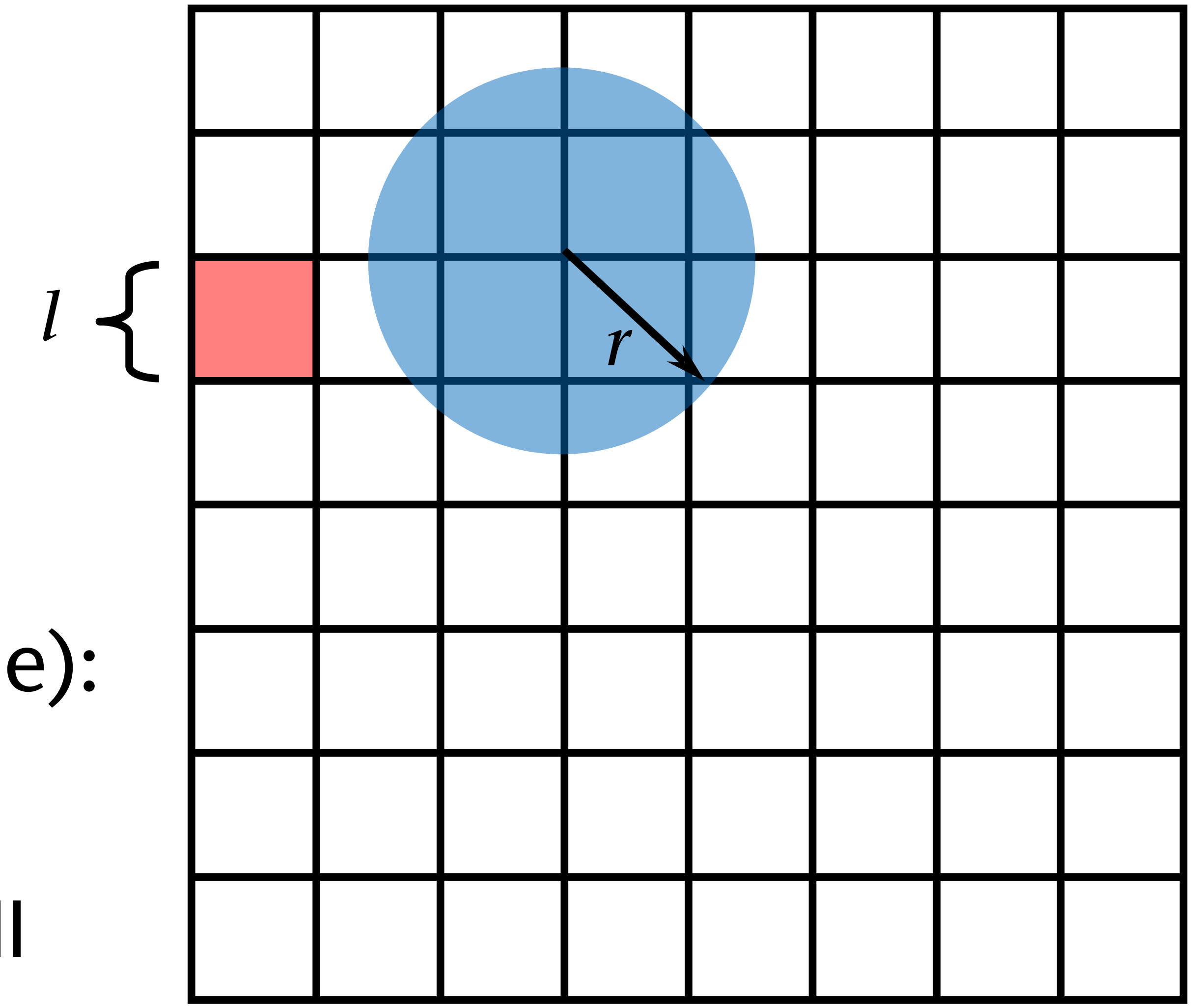


Fast (Sequential) Algorithm for Poisson Disk Pattern

- Works in any dimension *d*
- Store samples (to be constructed) in array *S*[]
- Maintain background grid with cell size

$$l = \frac{r}{\sqrt{d}}$$

  - No cell can contain more than one sample
  - Grid = *d*-dimensional integer array (store as hash table):
    - Value = 0 $\longrightarrow$ cell is empty
    - Value = *i* $\longrightarrow$ *S*[*i*] contains coords of the sample in this cell



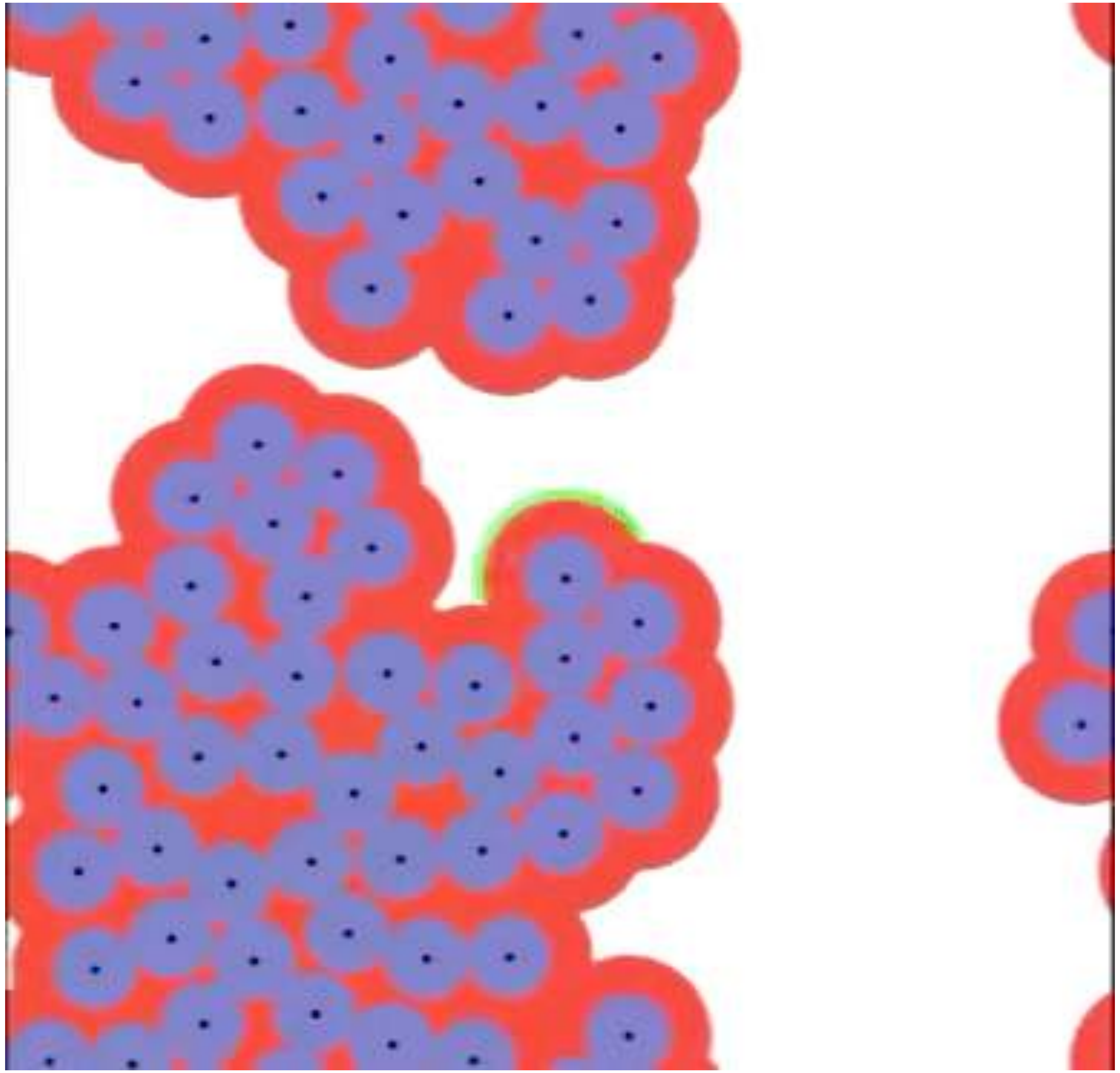
Algorithm

[Blend of Bridson 2007 with Dunbar and Humphreys 2005]

- Output: *S* = list of *N* samples (with Poisson disk property)
- During runtime, maintain *A* = list of "active" samples (indices into *S*), which represent the "border" of the sampling

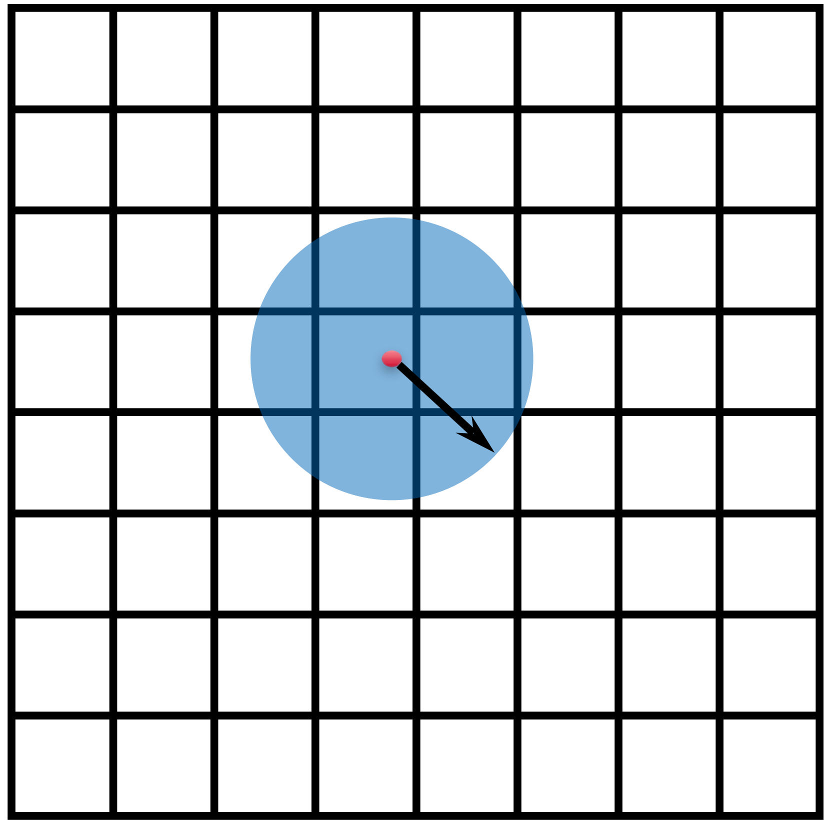
```
set S[0] = random point in domain (with uniform distrib.)
store pointer to S[1] in grid cell containing S[1]
A = {0}
while A is not empty:                                    (+)
  choose random index j in A
  repeat at most k times:
    generate a candidate point q randomly and uniformly
      on the boundary of A[j]'s disk
    if min. distance of q to all points in S >= r:       (*)
      add q to A, to S, and in grid
  if no new q was found after k attempts:
    delete A[j] from A  // A[j] is no longer on "outskirts"
```

Algorithm Visualization



Complexity

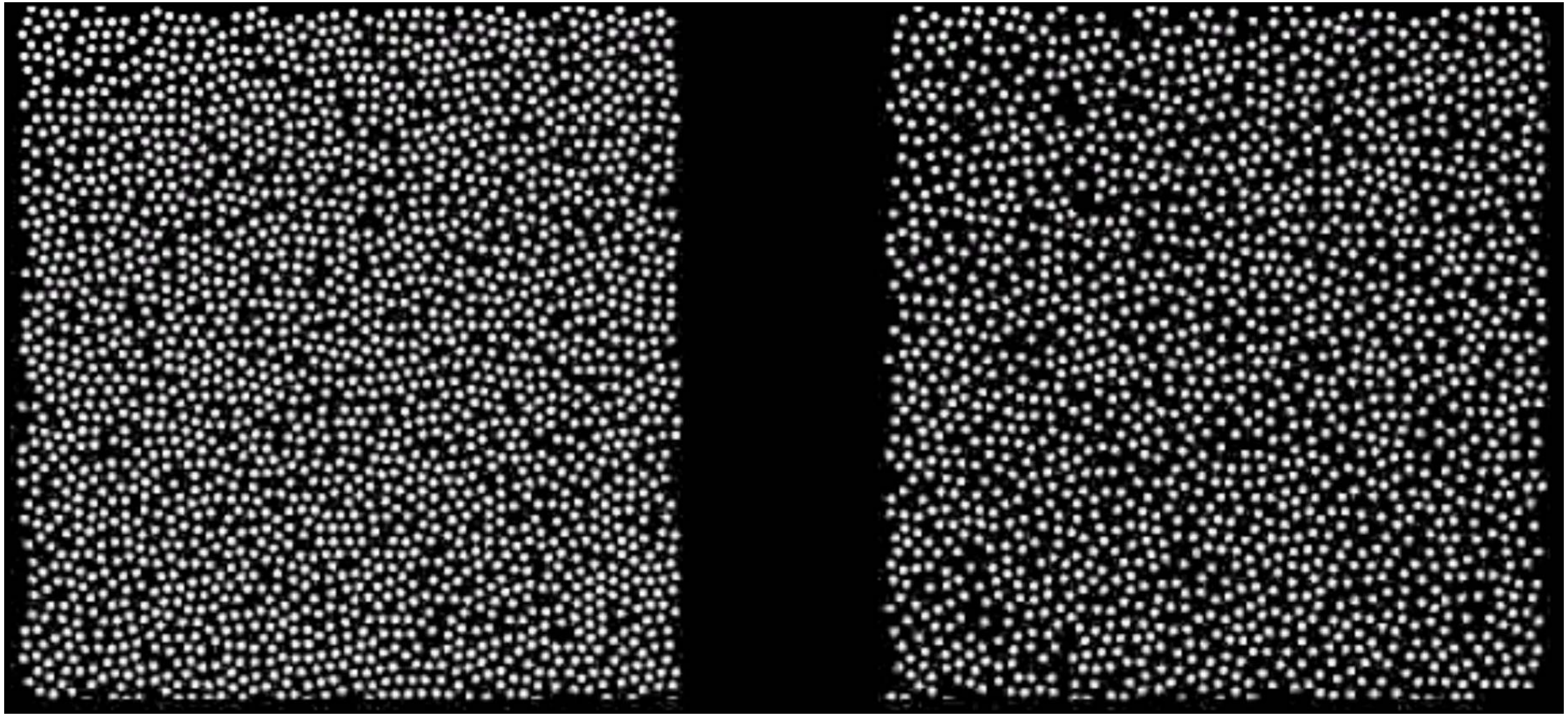
- On step (*):
  - No need to compute minimum distance from q to all S
  - Just verify that no point in S is inside disk of radius *r* around q
  - Visit 4^d neighboring cells (incl. own), each can contain at most 1 point
  - Distance computation $\in O(d)$
- On the # iterations of the while loop (+):
  - Let N = #iterations where a point $\in$ S is deleted from A
  - This happens exactly N times
  - In all other iterations, a point is added to S (and A)
  - Every point gets inserted in A and deleted from A exactly once
  - Overall: $2N$-1 iterations, each is in $O(d4^d)$ = const (if d=const)
- Overall complexity $\in O(N \cdot d4^d)$
  - If d is considered constant $\longrightarrow$ O(N)



Results

Boundary Sampling

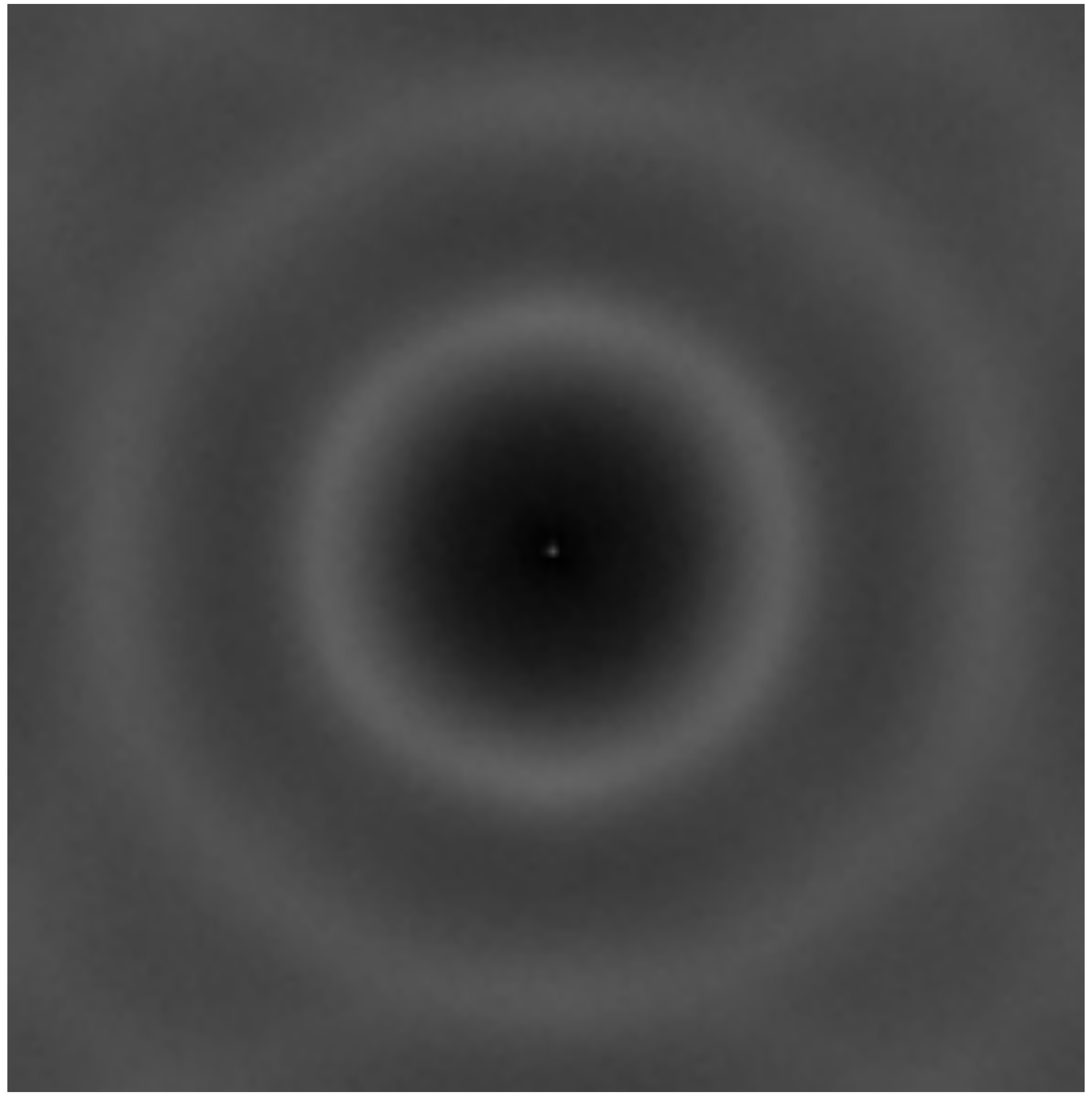
Dart Throwing



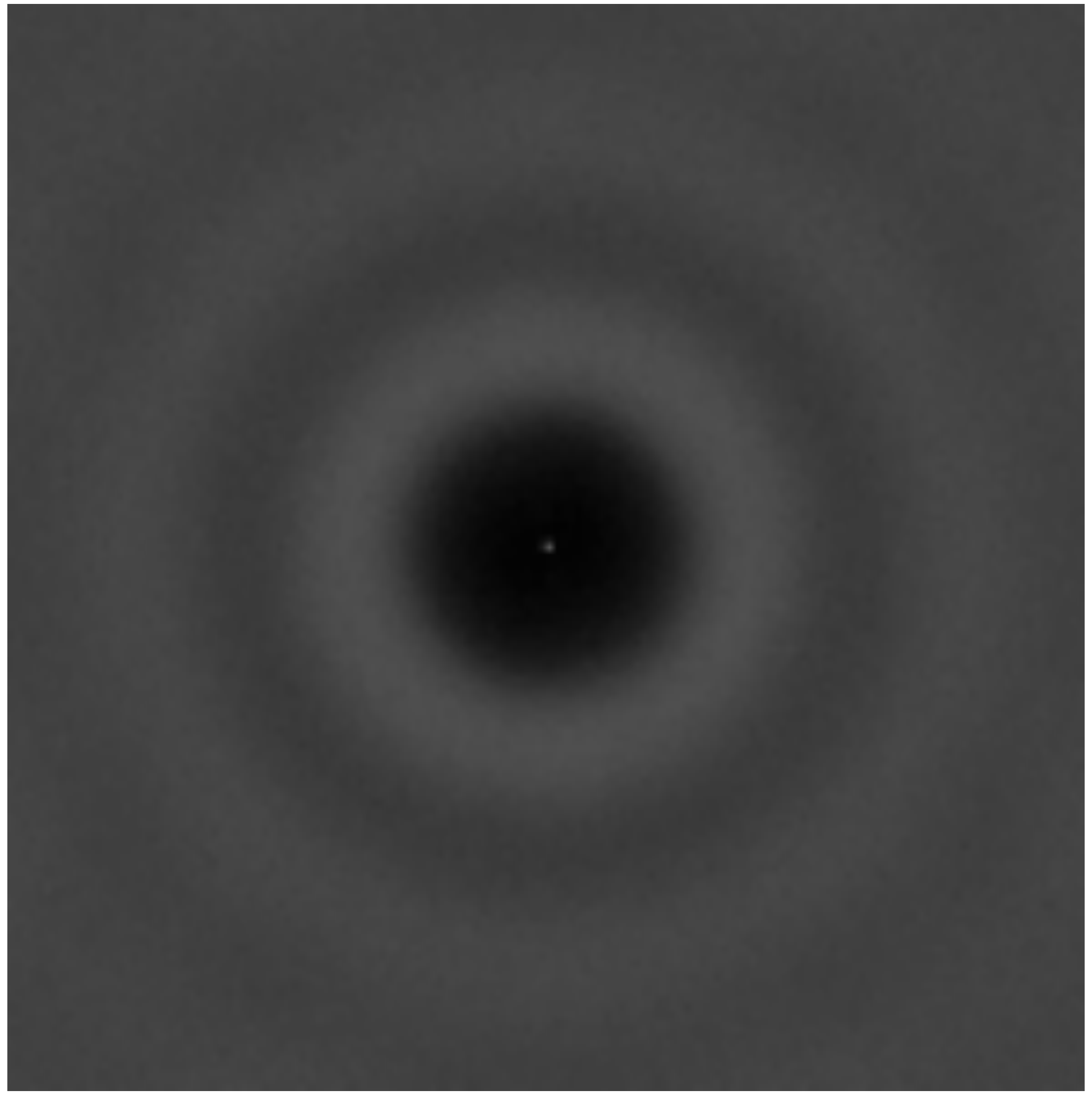
Denser, and
somewhat more regular

- Boundary sampling generates very good blue noise spectrum due to its extremely regular and dense sampling of the plane

Boundary Sampling

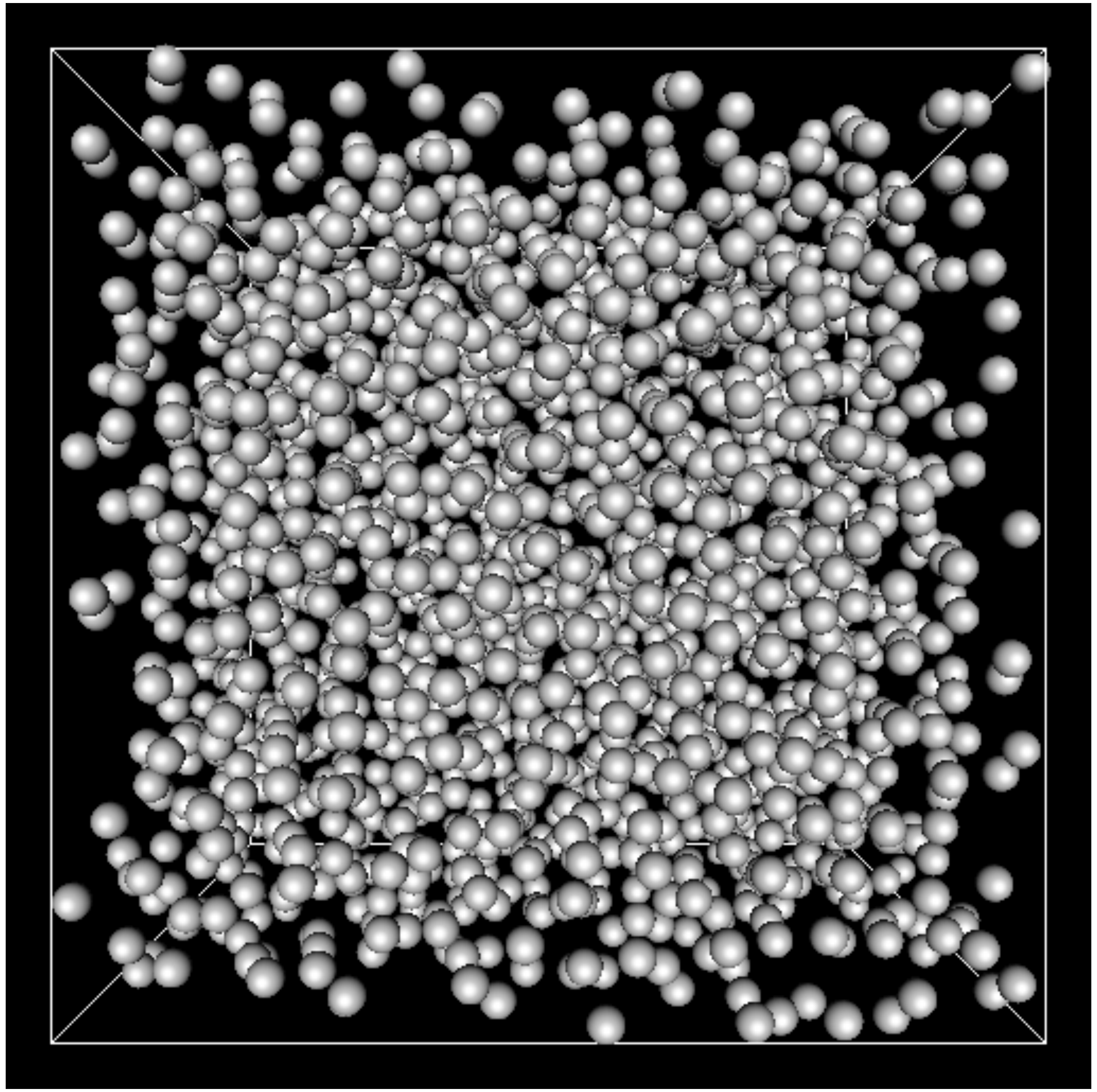


Dart Throwing



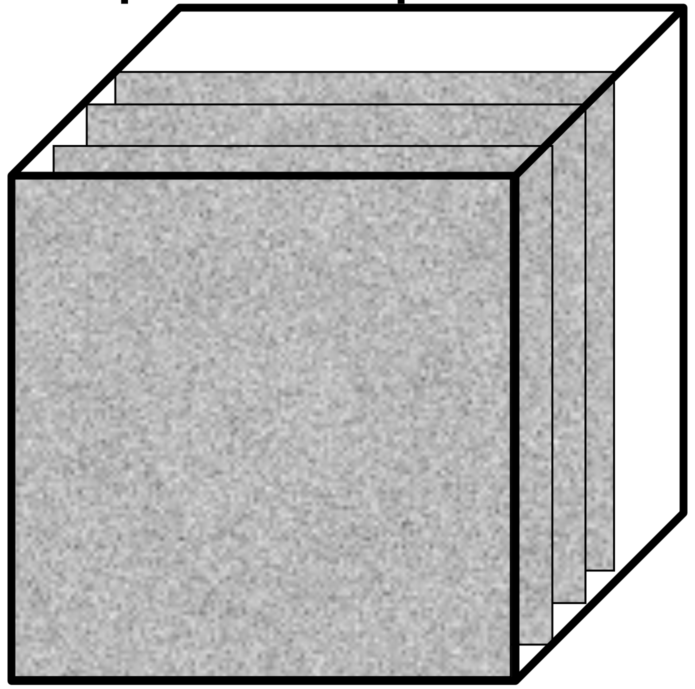
Frequency spectrum, averaged over many sample sets generated with each method, respectively

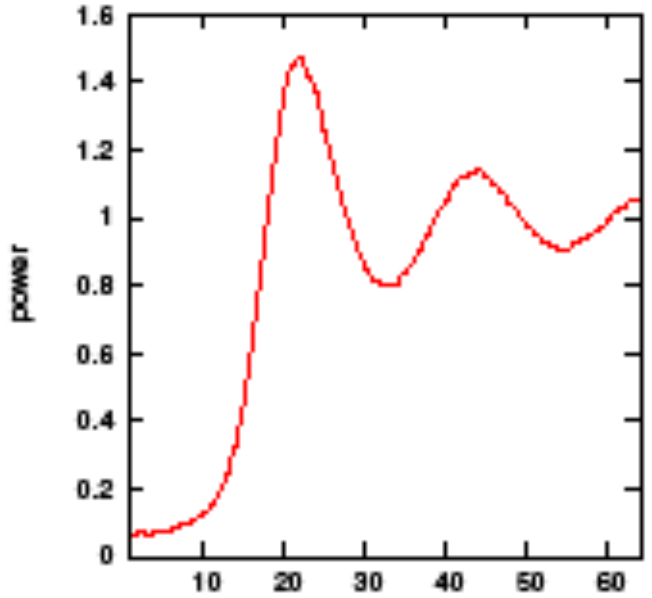
Sampling in Higher Dimensions



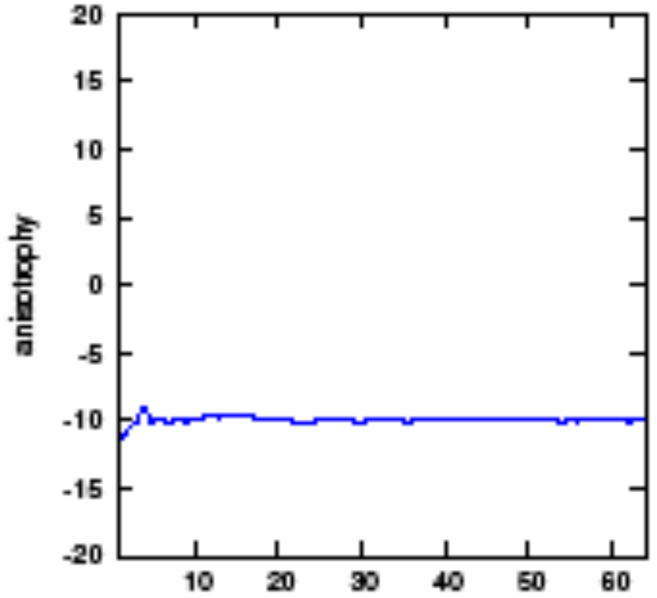
3D samples

power spectrum





radial mean



radial variance

Performance

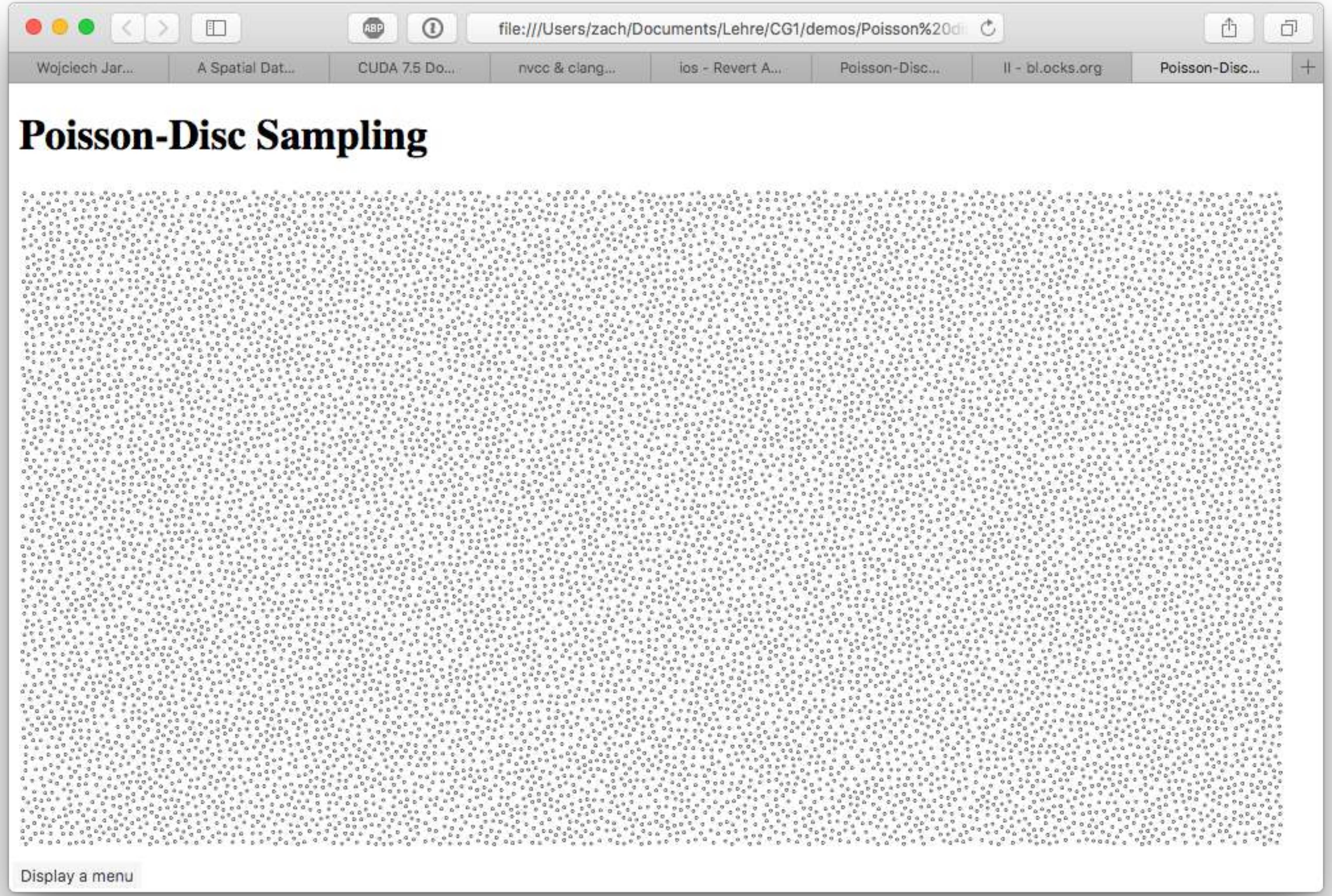
- Parallelization is possible and not too difficult [Wei 2008]
  - "Just" have to prevent "congestion" of grid access operations
- Comparison of three methods generating 2D Poisson disk samplings:

	Parallel algo [Wei 2008]	Sequ. boundary sampling [Dunbar, et al.]	Hierarchical dart throwing [White et al. 2007]
#samples/sec	4000 k	200 k	210 k

- Parallel generation of Poisson disk samplings in different dimensions [Wei 2008]:

	2D	3D	4D	5D	6D
#samples/sec	4000 k	550 k	43 k	2 k	180

Demo


file:///Users/zach/Documents/Lehre/CG1/demos/Poisson%20d
Wojciech Jar...
A Spatial Dat...
CUDA 7.5 Do...
nvcc & clang...
ios - Revert A...
Poisson-Disc...
II - bl.ocks.org
Poisson-Disc...
Poisson-Disc Sampling
Display a menu

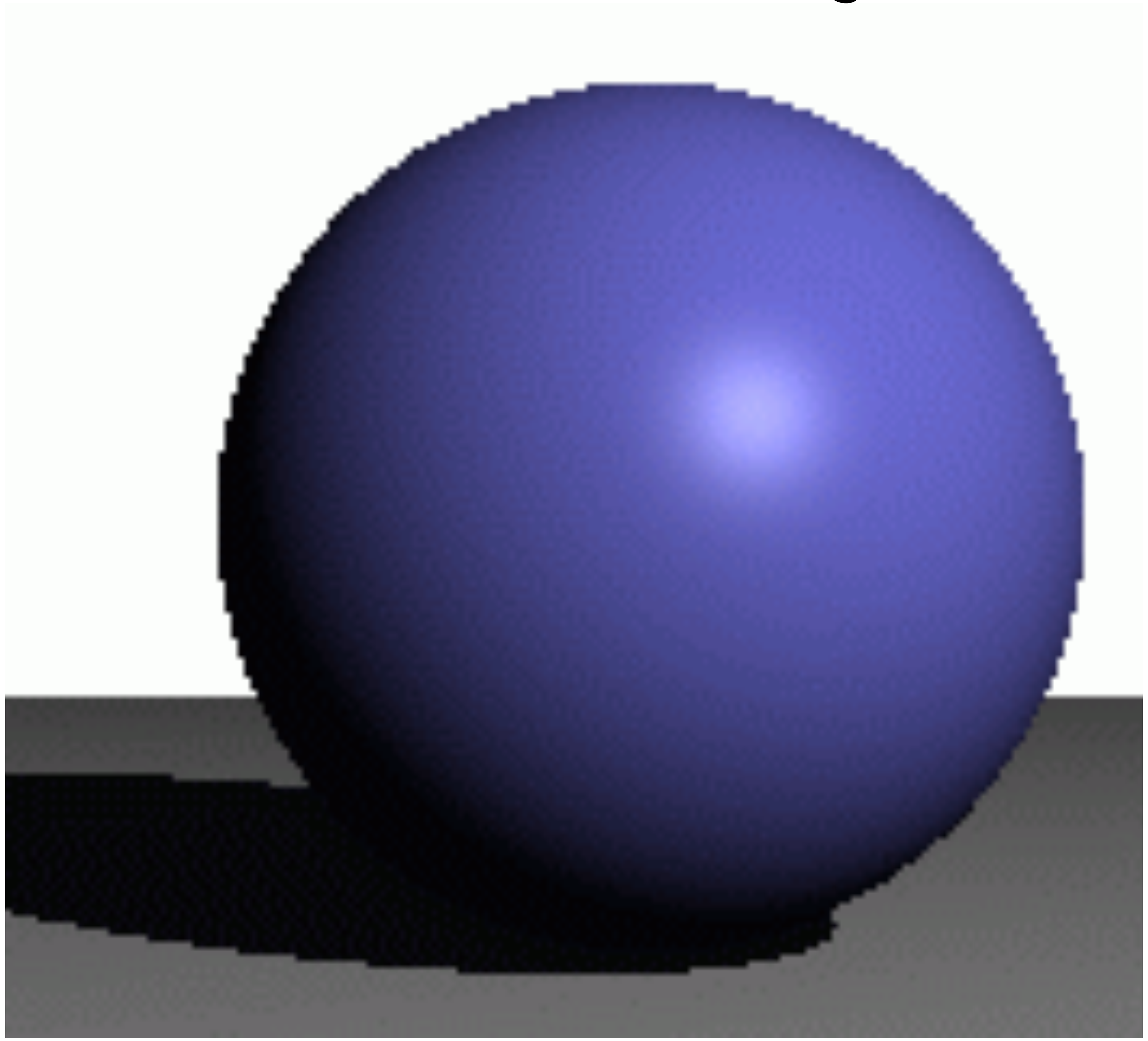
Result in Real Raytraced Image

Stratified (jittered), 4 samples/pixel

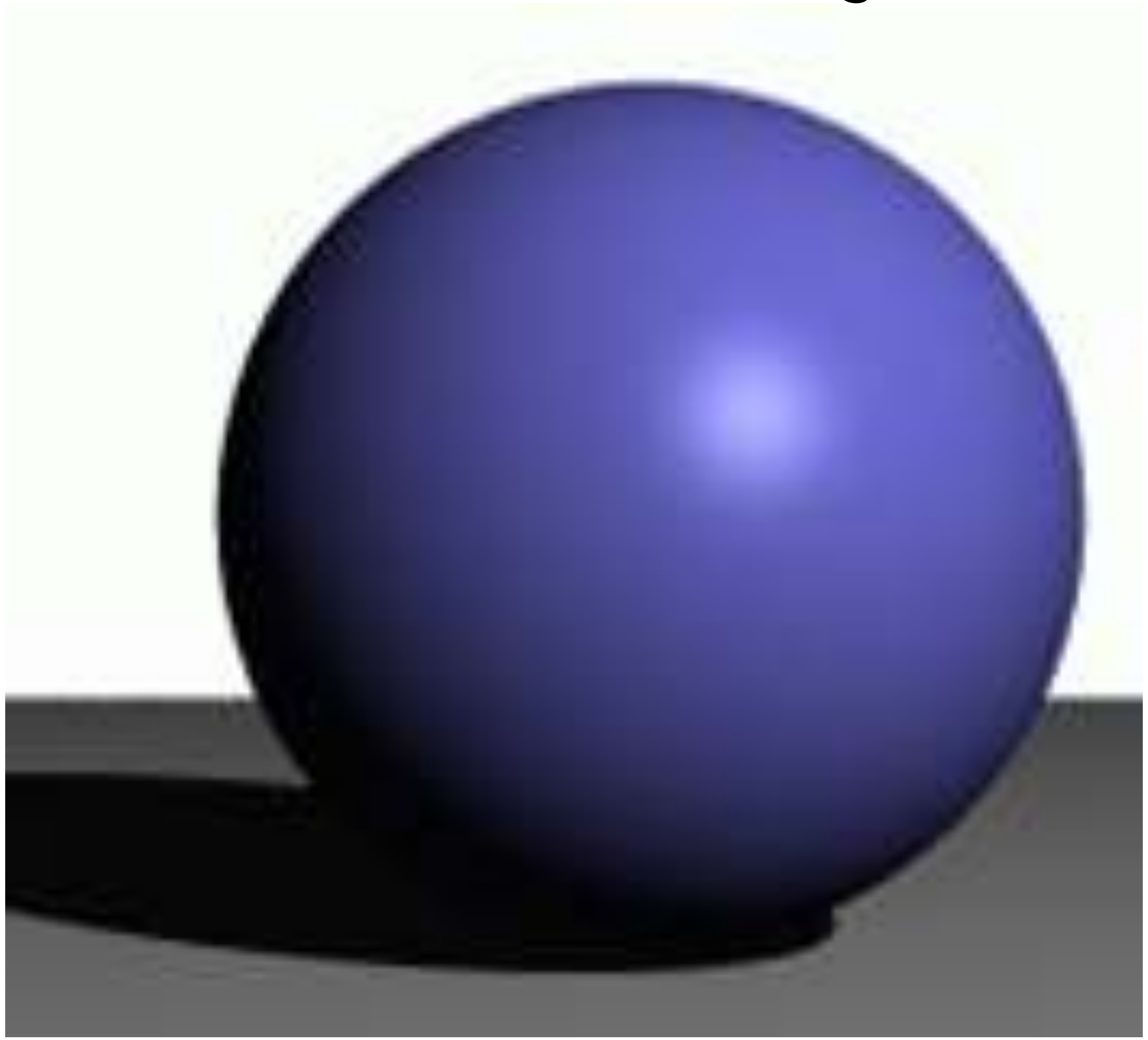
Approx. Poisson disk, 4 samples/pixel

Result

No Anti-Aliasing

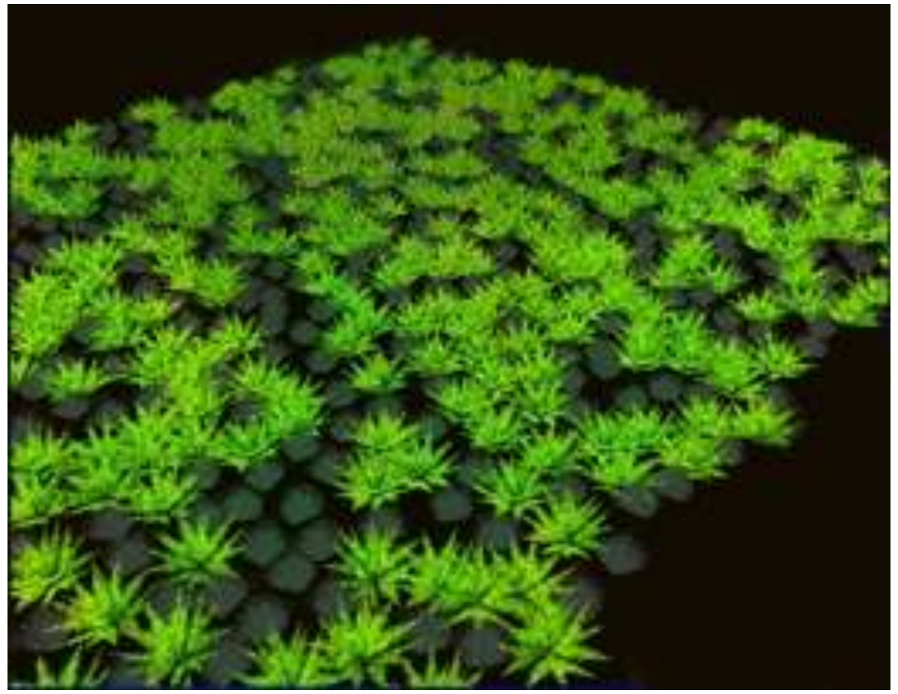


With Anti-Aliasing

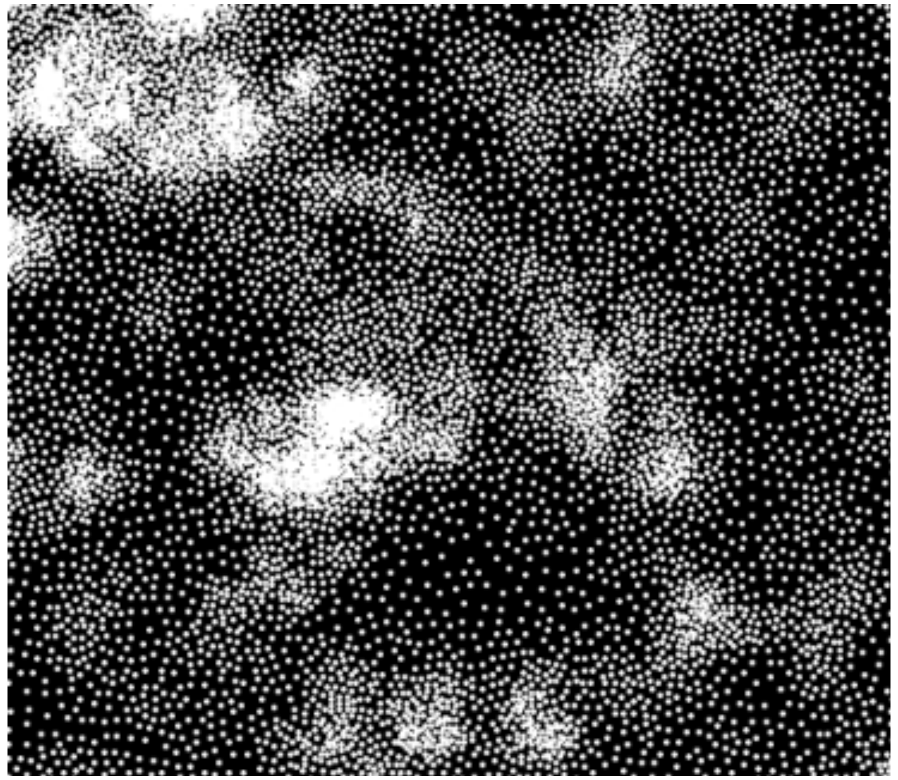


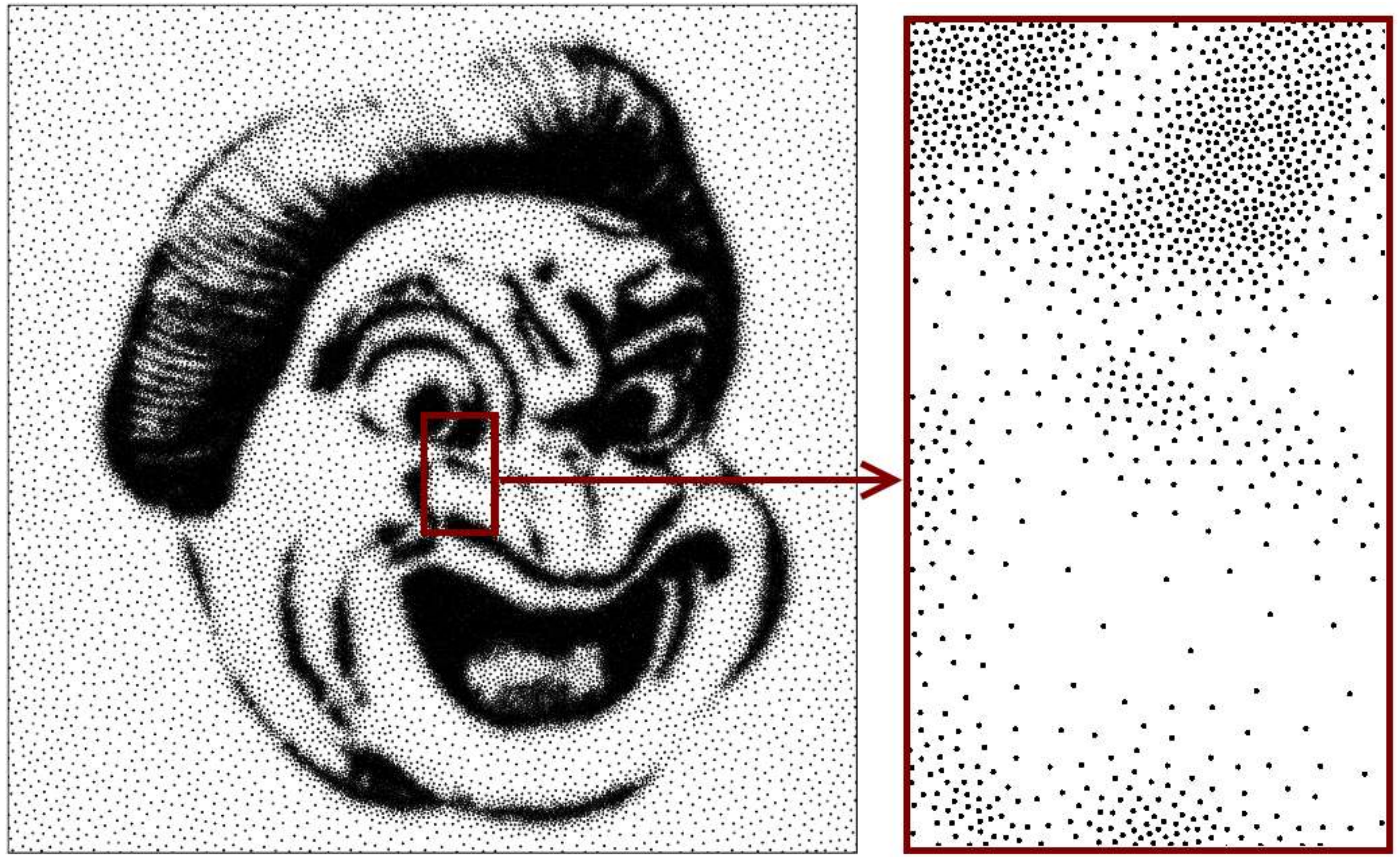
Digression: Many Other Applications of Poisson Sampling (w/o Details)

- Placing objects randomly, but neither "too far" from each other, nor "too close" to each other
  - E.g., for "terrain dressing"

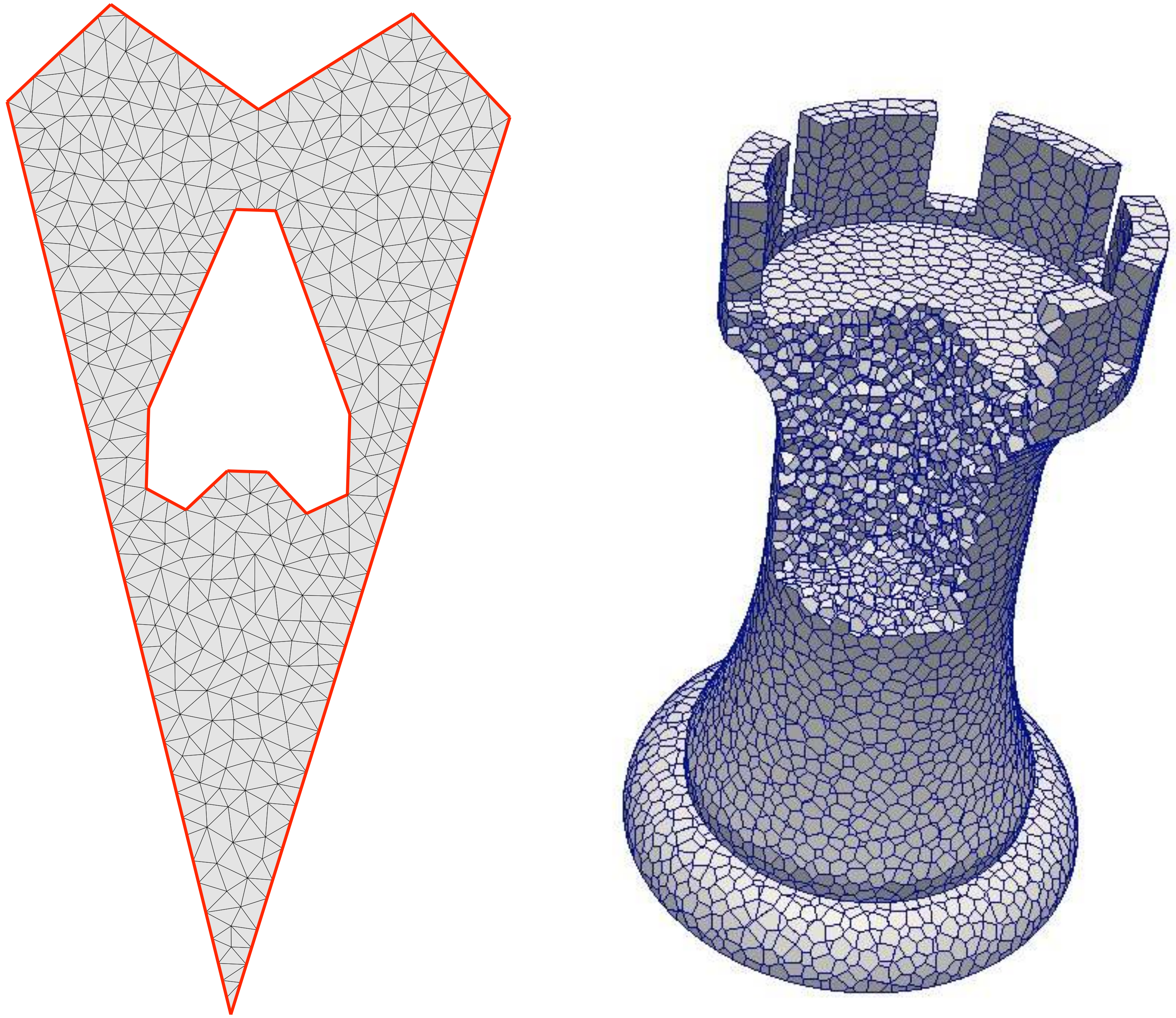


- Poisson sampling with local density control:
  - Radius of disks is not constant
  - Could be driven by Perlin noise
  - Could be a given as grayscale image



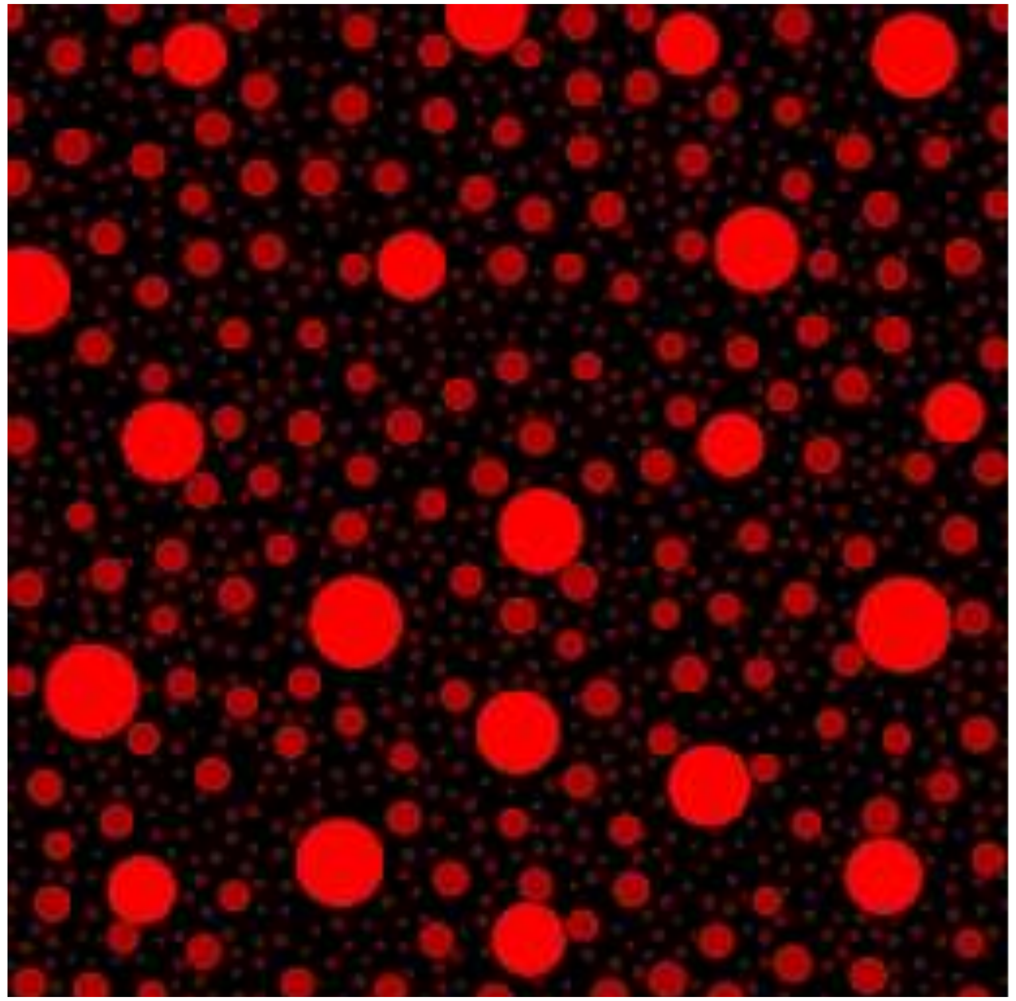


Dithering/stippling



Meshing
(interior triangulations/tetrahedralizations)

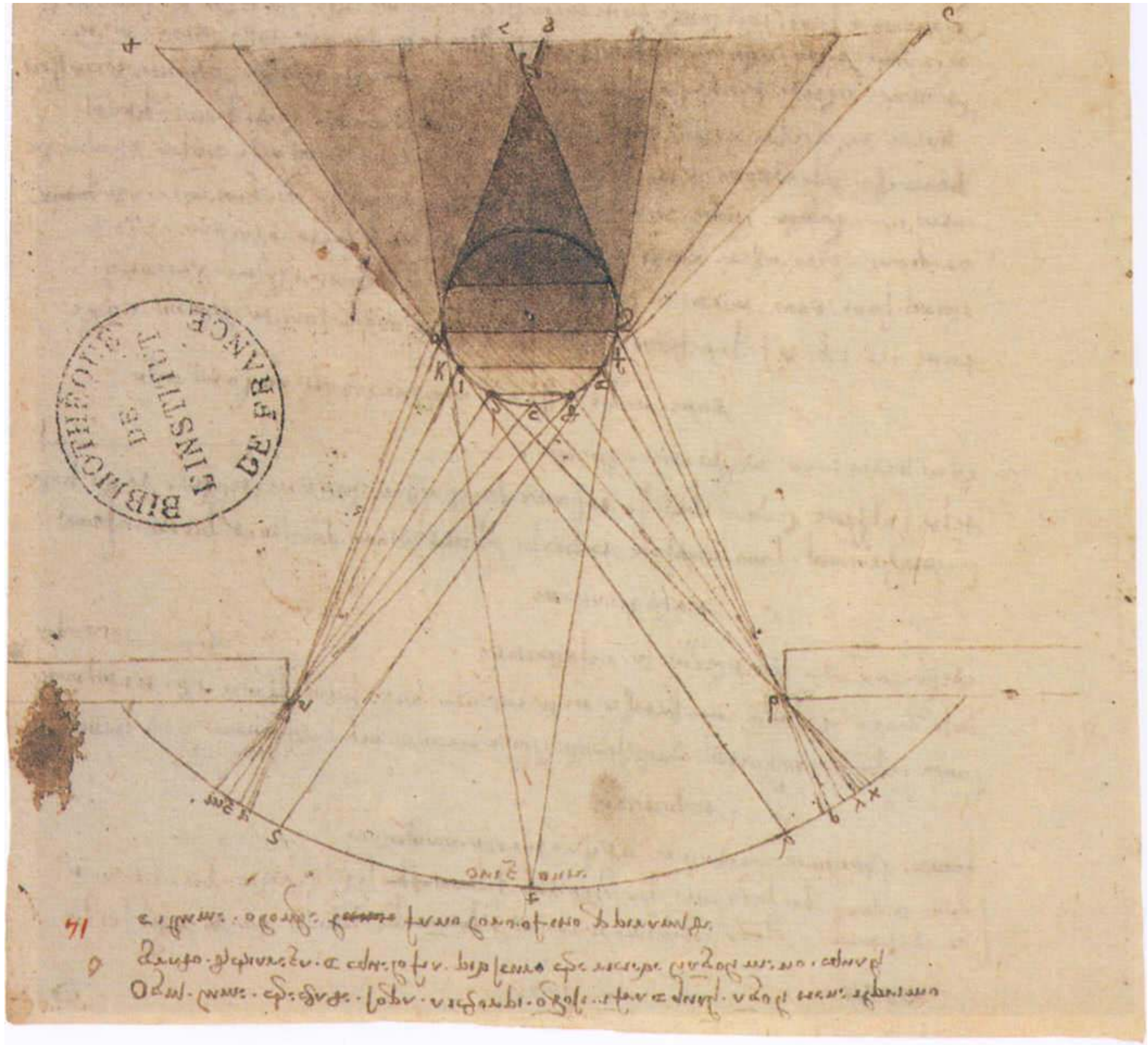
- Texture synthesis:
  1. Create raw image with Poisson disks of 3 different radii
  2. Post-process image (blurring, etc.)
  3. Use resulting image as bump map (see CG1)



Soft Shadows, Penumbra

"The shadows are as important as the light."
(*Charlotte Brontë*)

- Behind a lighted object, there are 3 regions:
  - Completely lighted
  - Umbra = completely in shadow
  - Penumbra = partially in shadow



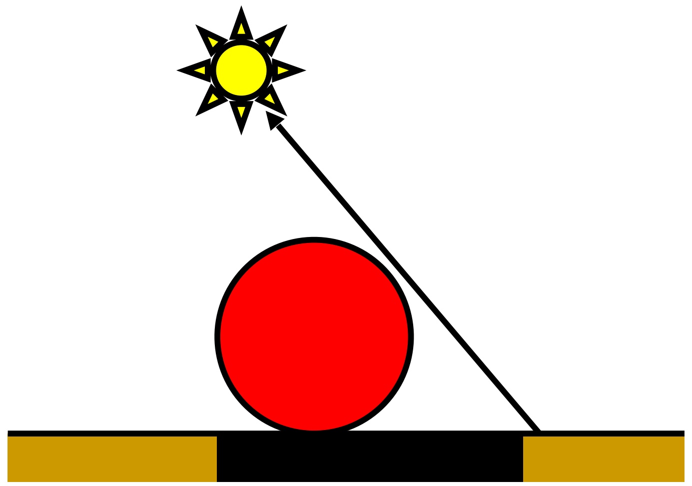
XVI. Léonard de Vinci (1452-1519). Lumière d'une fenêtre sur une sphère ombreuse avec (en partant du haut) ombre intermédiaire, primitive, dérivée et (sur la surface, en bas) portée. Plume et lavis sur pointe de métal sur papier. 24 x 38 cm. Paris, Bibliothèque de l'Institut de France (ms. 2185 ; B.N. 2038, f° 14 r°).

In Reality ...

… and in Ray-Tracing

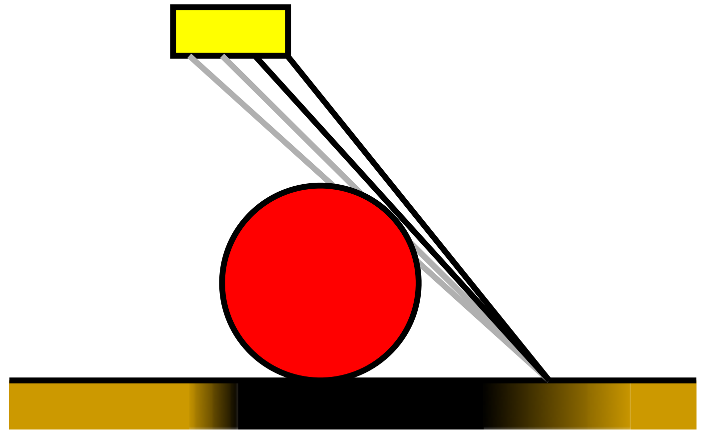
- So far, only 1 shadow feeler per light in the Phong model:

$$L_{\text{Phong}} = \sum_{j=1}^{n} s_j \cdot \rho(\Phi_j, \Theta_j) \cdot I_j \quad , \; s_i = \begin{cases} 1 \, , & \text{light source visible} \\ 0 \, , & \text{invisible} \end{cases}$$

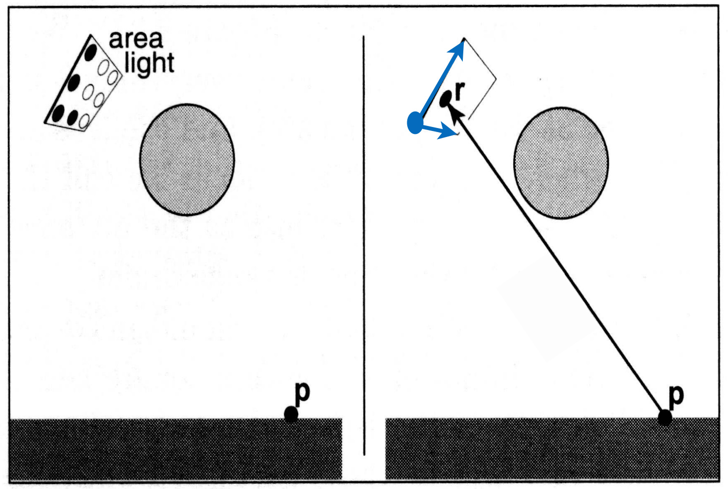


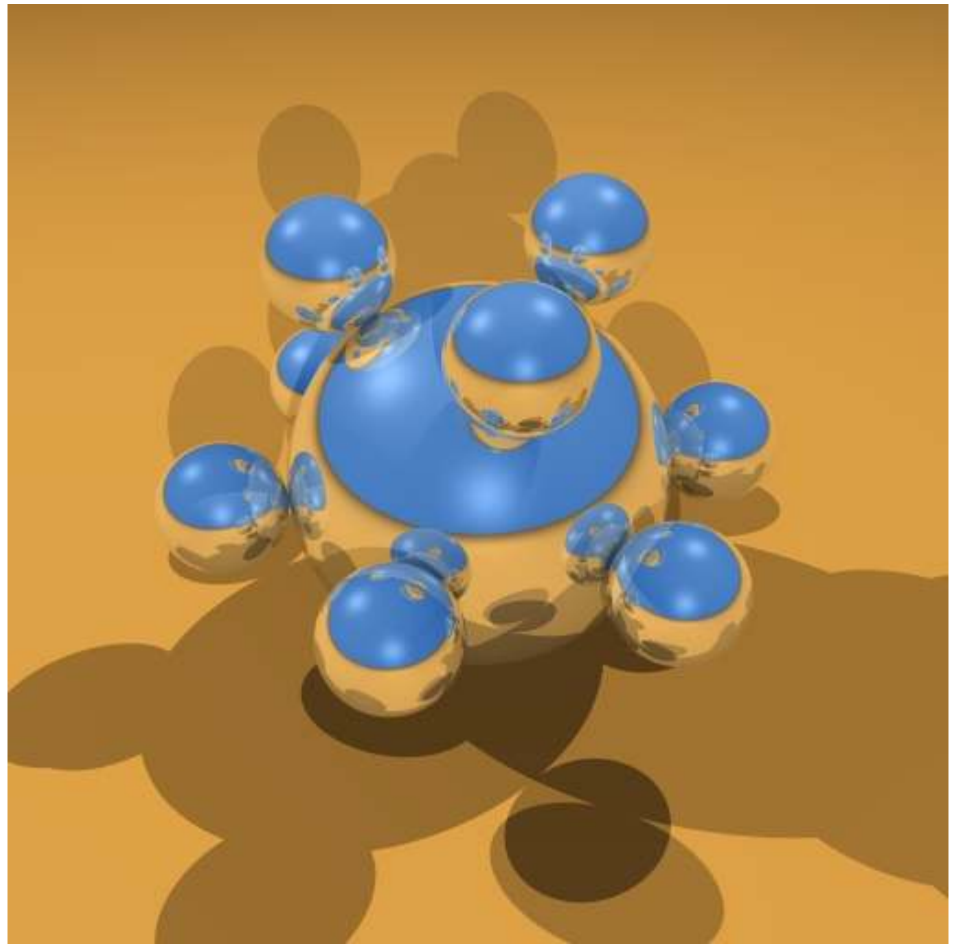
- Improvement: send many shadow feelers

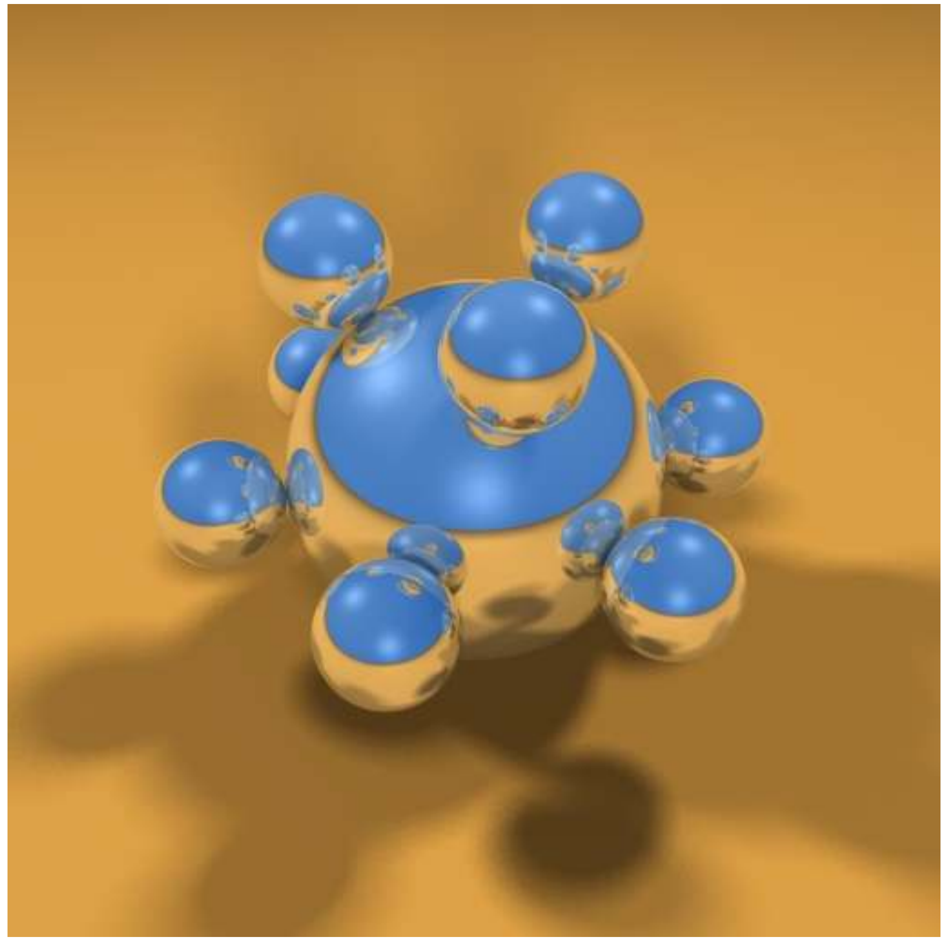
$$s_i = \frac{\#\ \text{passing shadow rays}}{\#\ \text{shadow rays sent}}$$

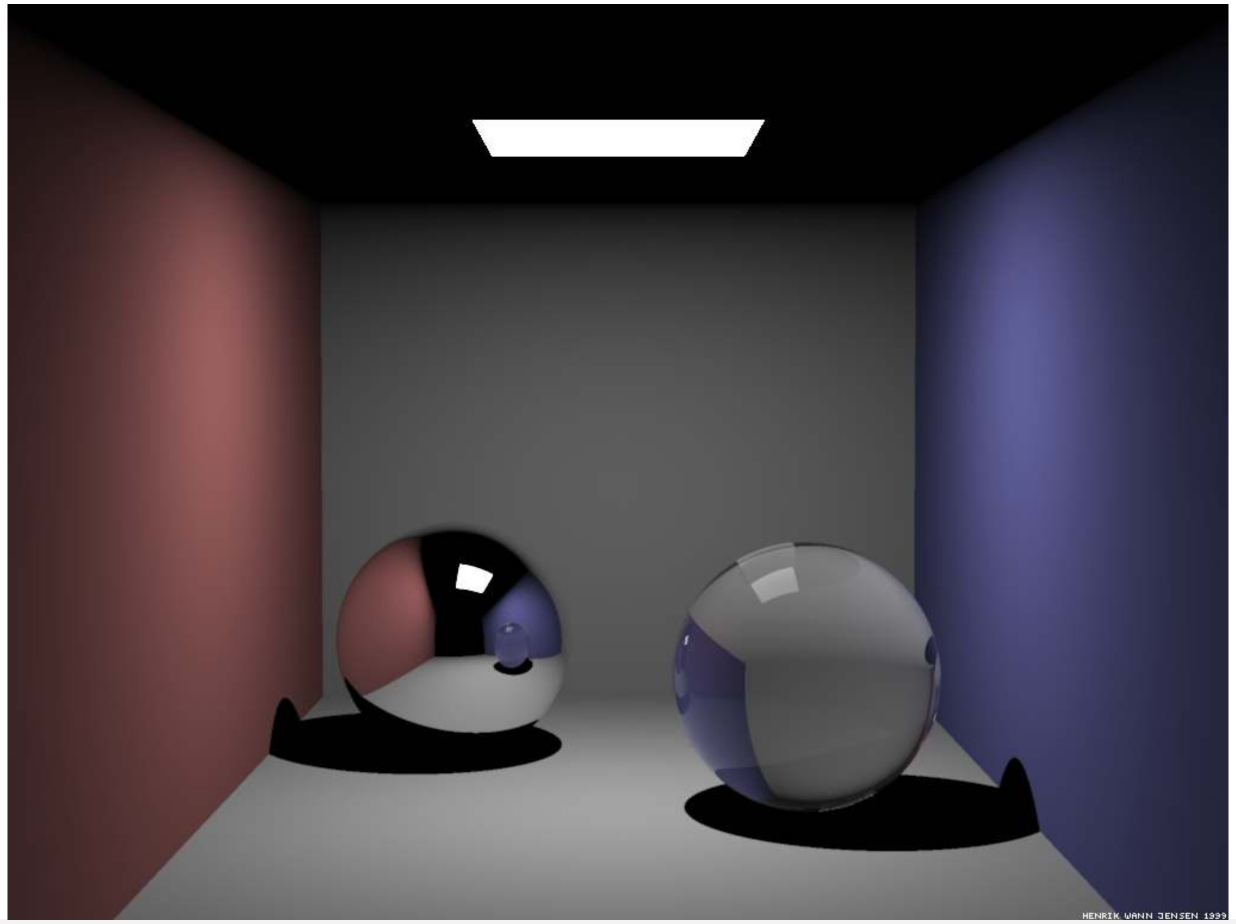


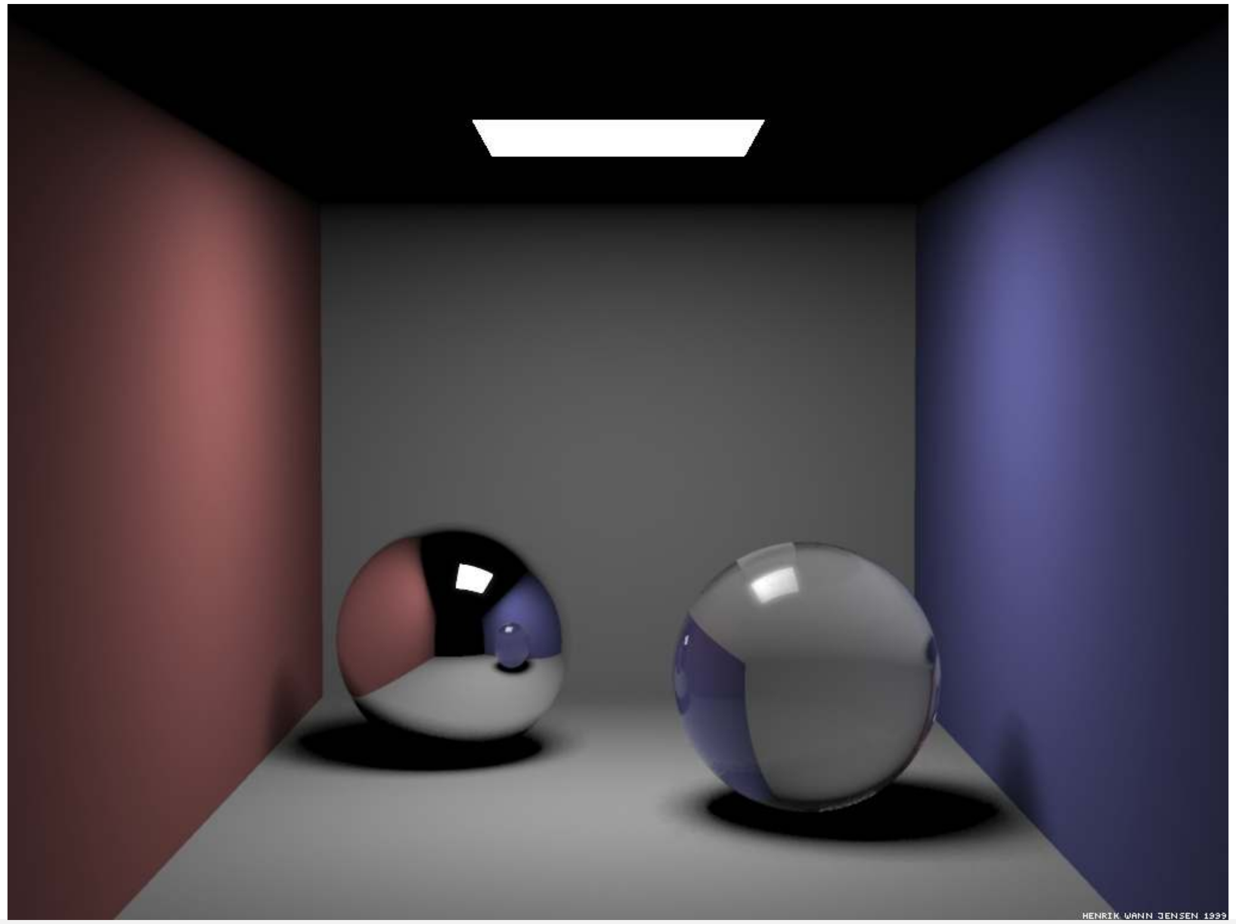
- Three ways of sampling a lightsource:
  1. Regular sampling of the area of the lightsource
  2. Random sampling
  3. Stratified sampling (just like with pixels/AA)





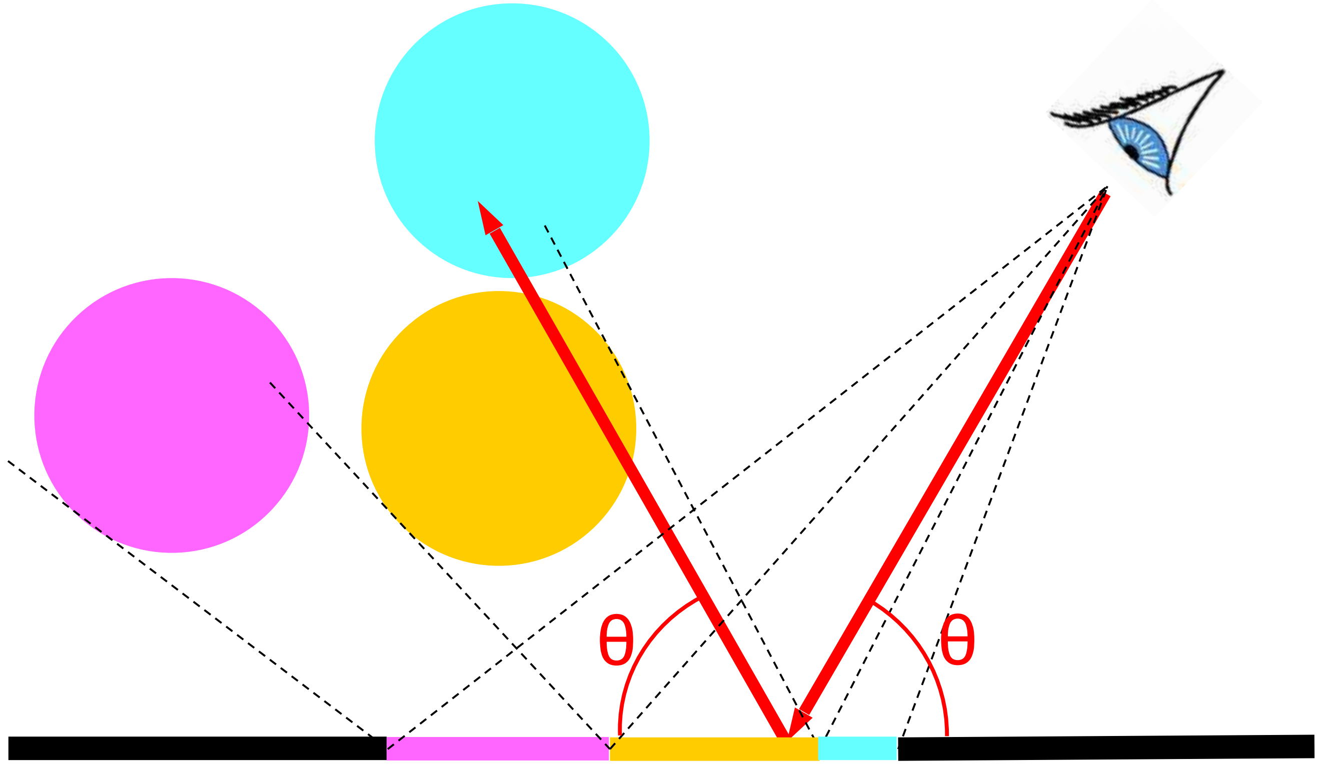






Better Glossy/Specular Reflection

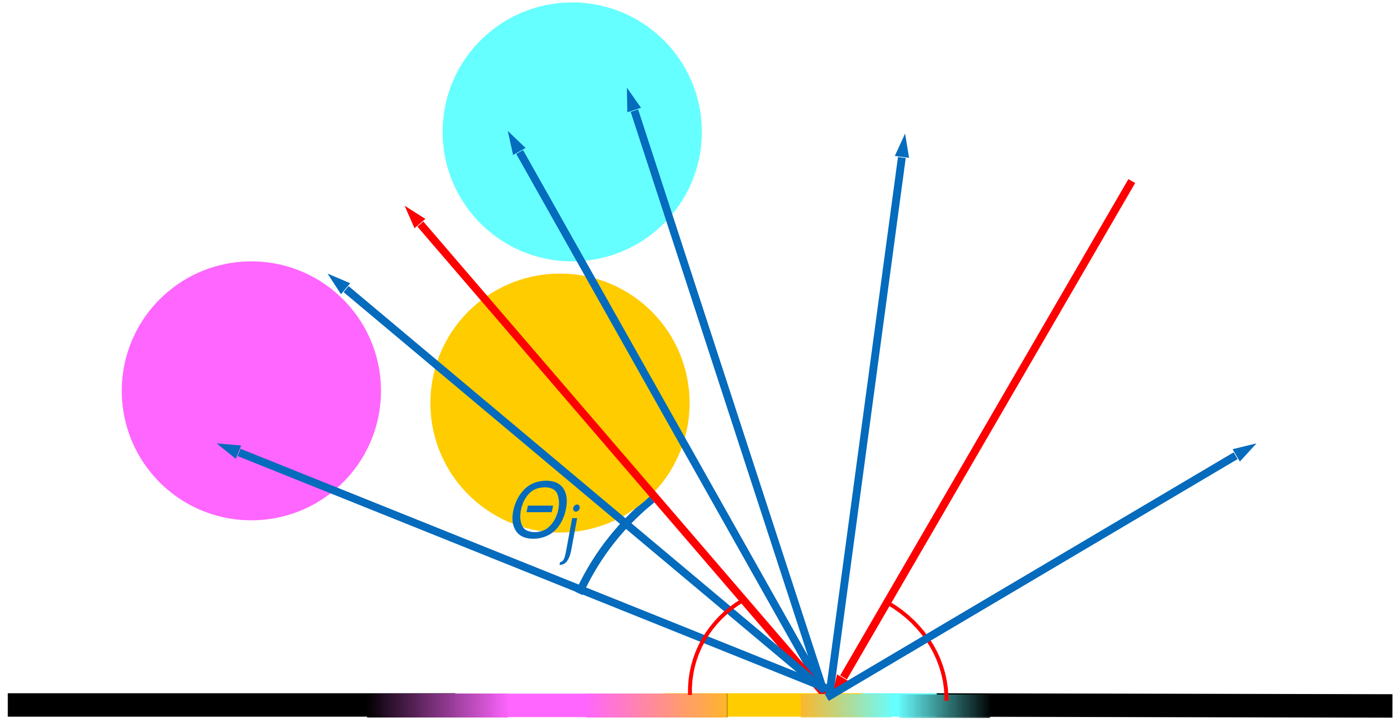
- So far, only one reflected ray:
  - Problem, if the surface should be matte-glossy ...



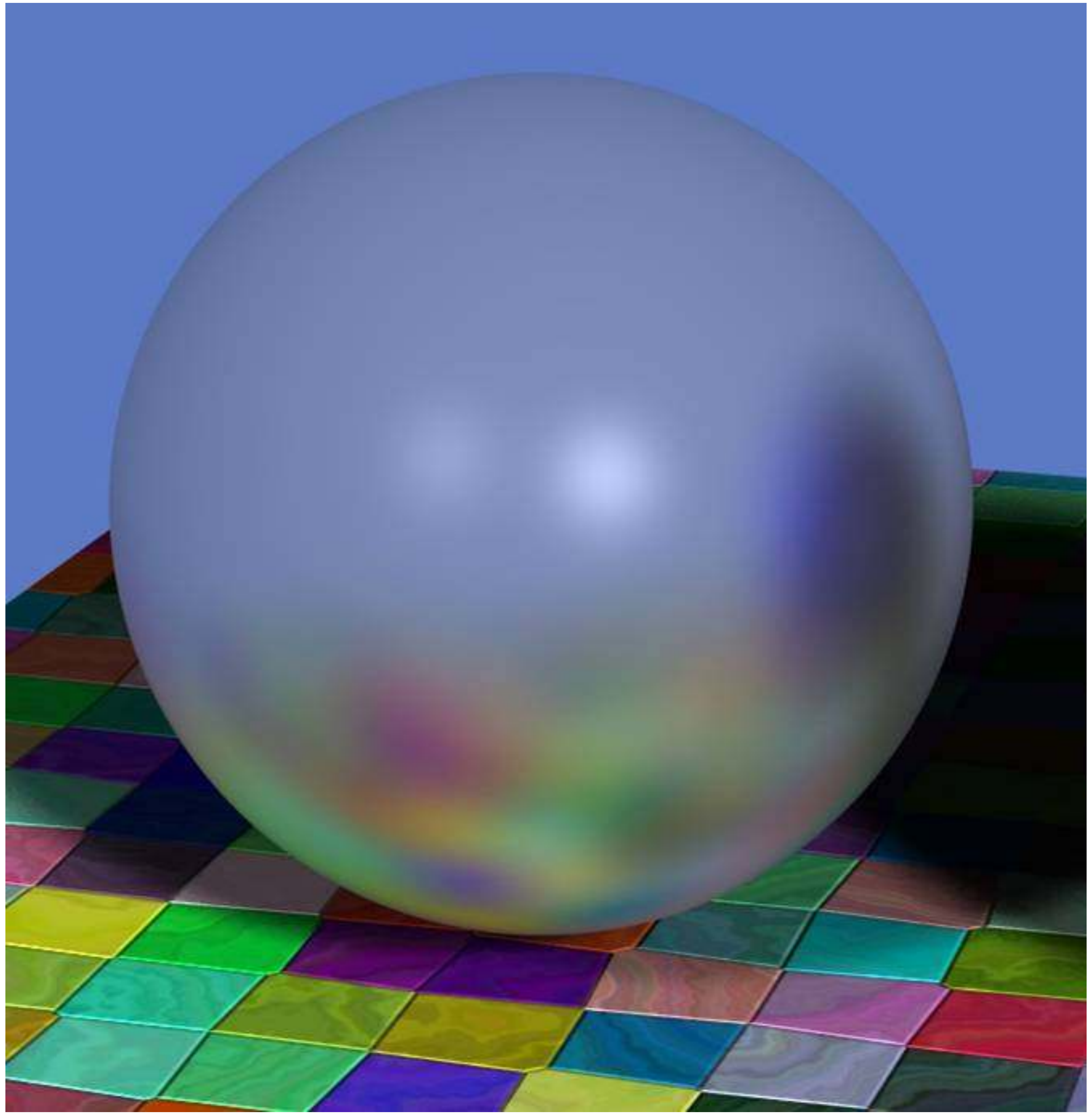
- Solution (somewhat brute-force):
  - Shoot many secondary, "reflected" rays
  - Accumulate, weighted by the power-cosine law

$$\cos^p \Theta_j$$

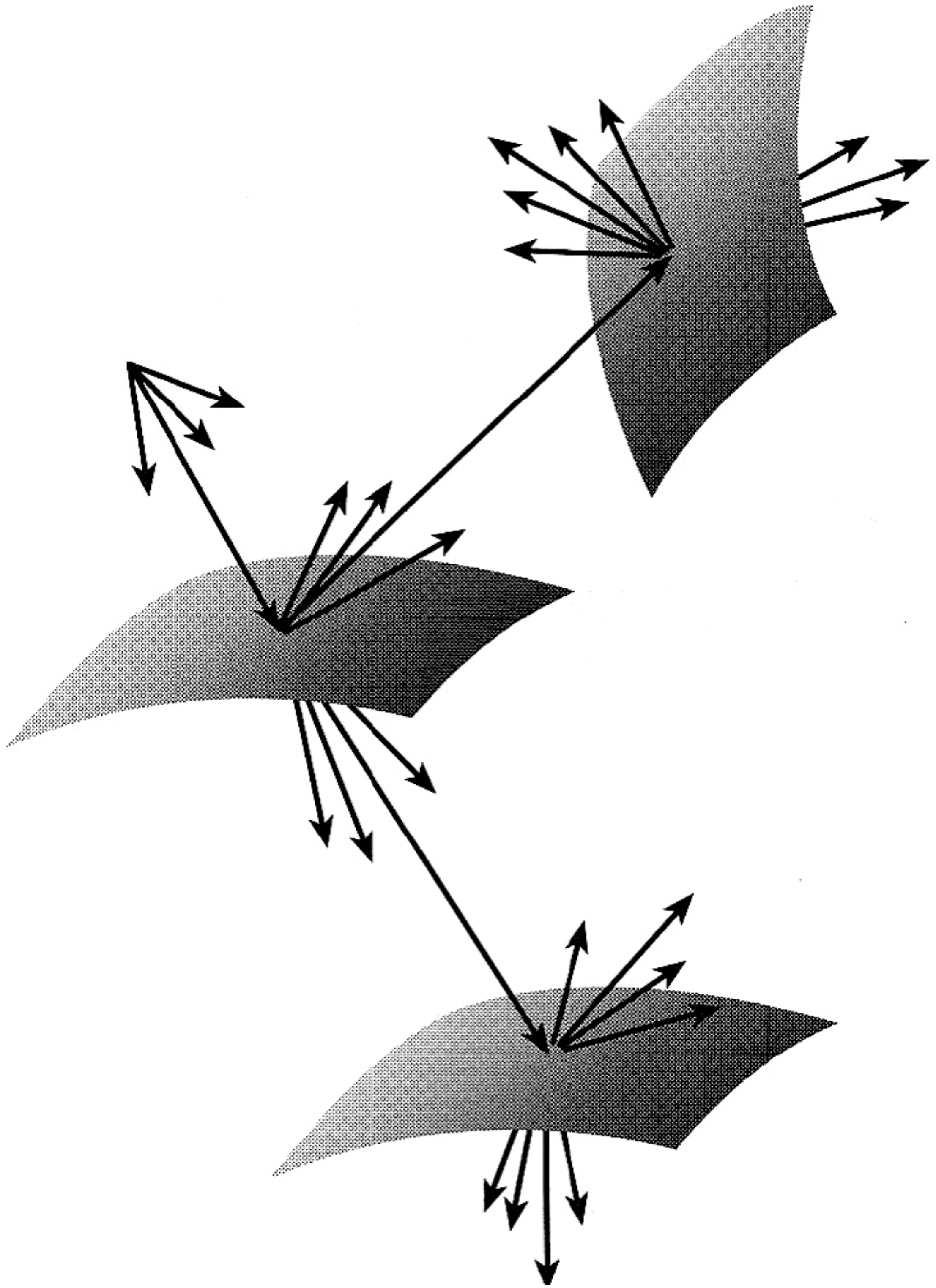
(notice coincidence with the Phong model)



Example



The ray tree



Bremen

Generating a Poisson Disk Distribution on the Sphere

- For sake of simplicity: generate just an approximation
- Prerequisite is Mitchell's *best candidate* algorithm:

```
set S[1] = random point in domain (with uniform distrib.)
for i = 2 .. N:
  C = generate m (random) candidate points
  for all p in C: compute dist(p,S) = min. distance to S
  pick p* in C with maximal dist(p*,S) → add p* to S
```

- Usually, m is increased with iteration count i, e.g. $m = i \cdot q$ with q = "quality" parameter
- Generates only an *approximate* Poisson disk sampling (not maximal one)
- Advantage: can add more points later / refine existing sampling pattern

Now for the Poisson disk distribution

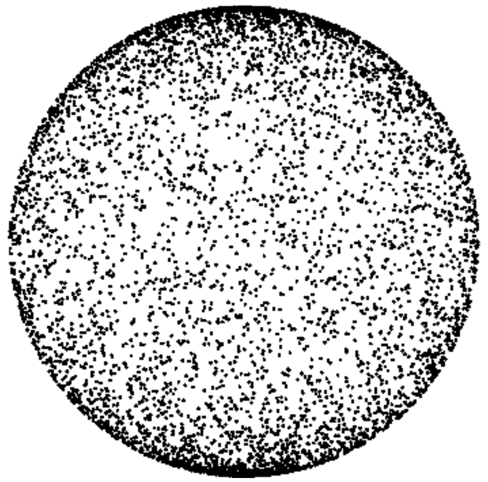
- Use Mitchell's best candidate algorithm directly on the sphere:
  - Generate the candidate points directly on the sphere (w/ uniform distribution!)
  - Computing the distance:
    - Should use geodesic distance on the sphere
    - Here, just approximate it by Euclidean distance
    - Works okay because we are just interested in *comparisons* of distances anyway, and

$$\|\mathbf{p}_1 - \mathbf{q}\| < \|\mathbf{p}_2 - \mathbf{q}\| \;\Leftrightarrow\; \text{geo-dist}(\mathbf{p}_1, \mathbf{q}) < \text{geo-dist}(\mathbf{p}_2, \mathbf{q})$$

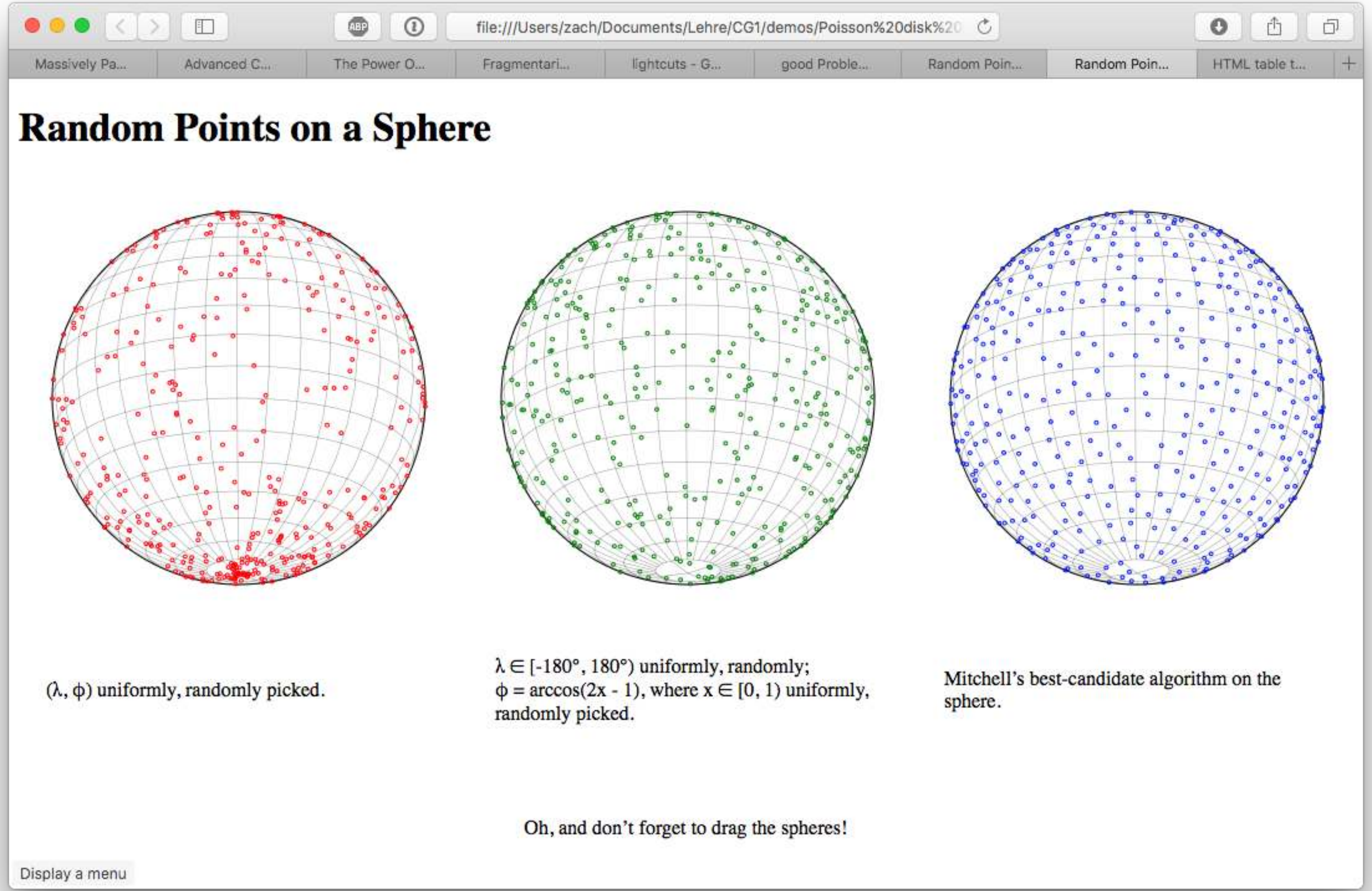
  - This "shortcut" will not work on 2-manifolds that have non-convex parts!

How to Generate Random Points on a Sphere with Uniform Distribution?

- Naïve approach 1: generate random uniform *x, y, z*, then normalize (project onto sphere)
- Naïve approach 2: pick spherical coordinates (λ, φ) randomly and uniformly
- Method 1:
  - Generate three random numbers *x, y, z* using *Gaussian* distribution, then normalize (*x,y,z*)
- Method 2 (works less well in high dimensions):
  - Generate random uniform *x, y, z* (i.e., in unit cube)
  - Reject, if || (x,y,z) || > 1 (also reject, if < ε)
  - Normalize remaining (*x,y,z*)

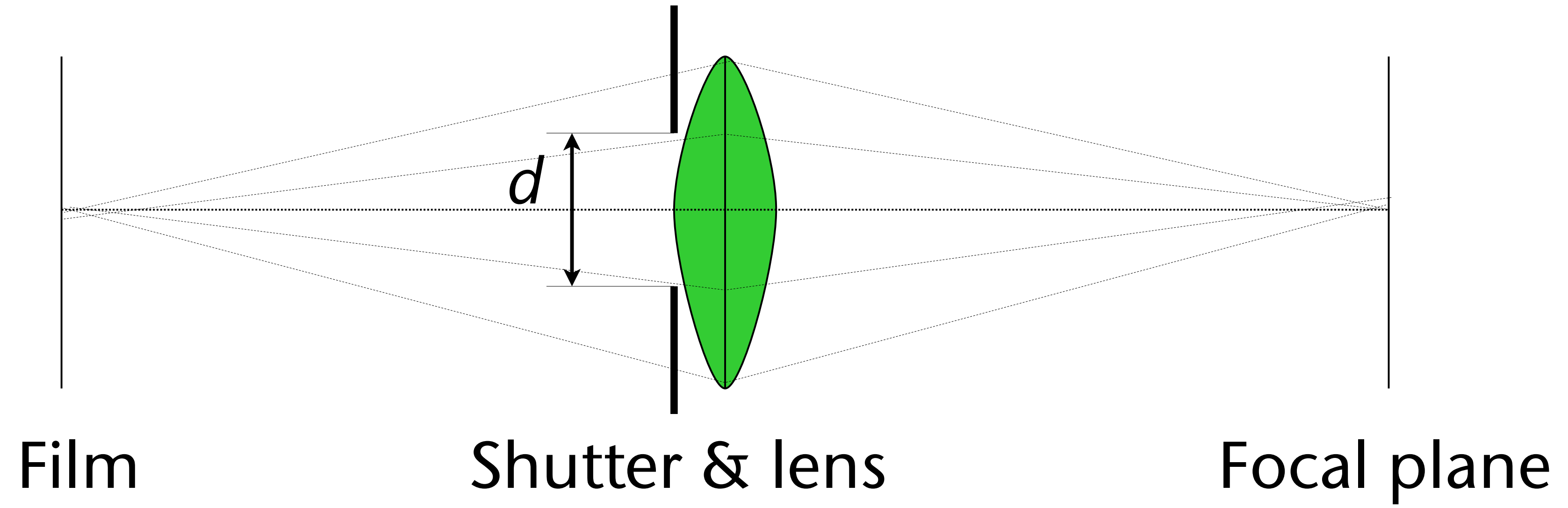


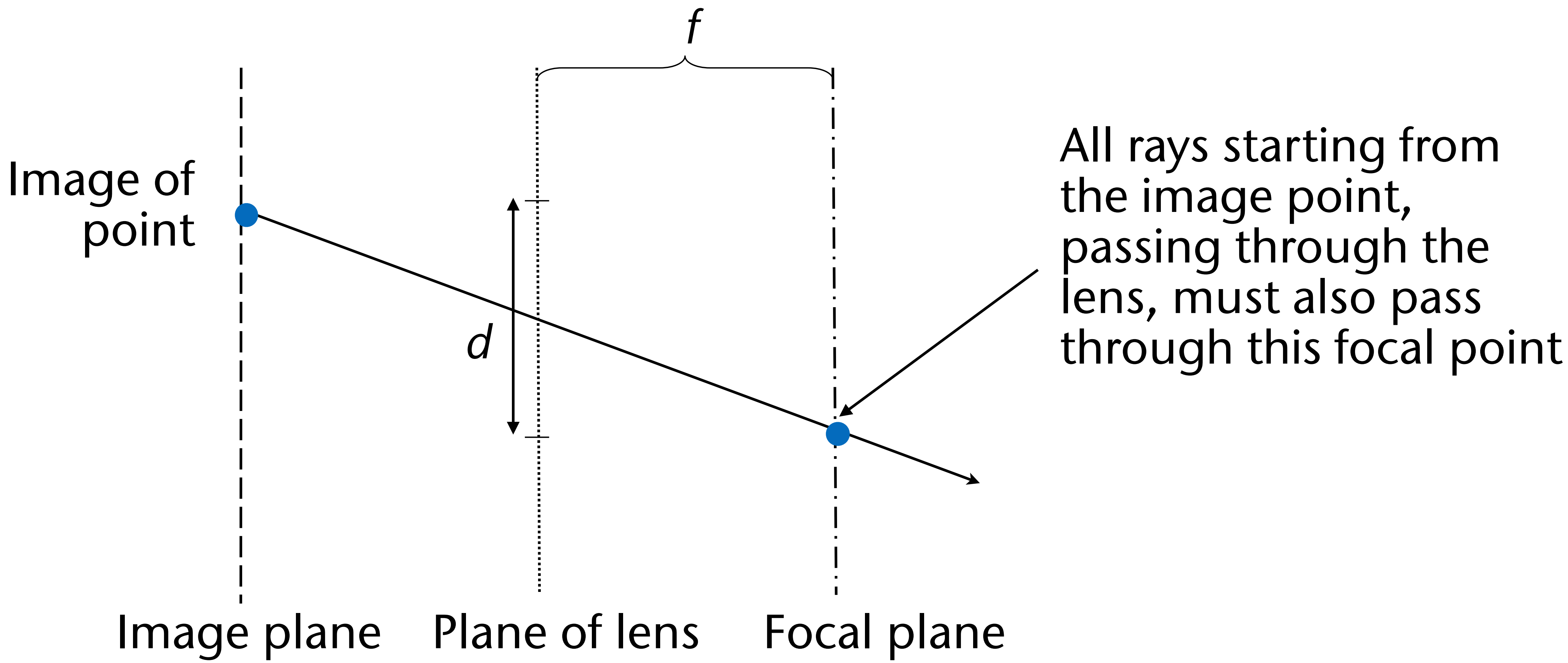
Demo


file:///Users/zach/Documents/Lehre/CG1/demos/Poisson%20disk%20
Massively Pa...
Advanced C...
The Power O...
Fragmentari...
lightcuts - G...
good Proble...
Random Poin...
Random Poin...
HTML table t...
Random Points on a Sphere
(λ, ϕ) uniformly, randomly picked.
λ ∈ [-180°, 180°) uniformly, randomly;
ϕ = arccos(2x - 1), where x ∈ [0, 1) uniformly, randomly picked.
Mitchell's best-candidate algorithm on the sphere.
Oh, and don't forget to drag the spheres!
Display a menu

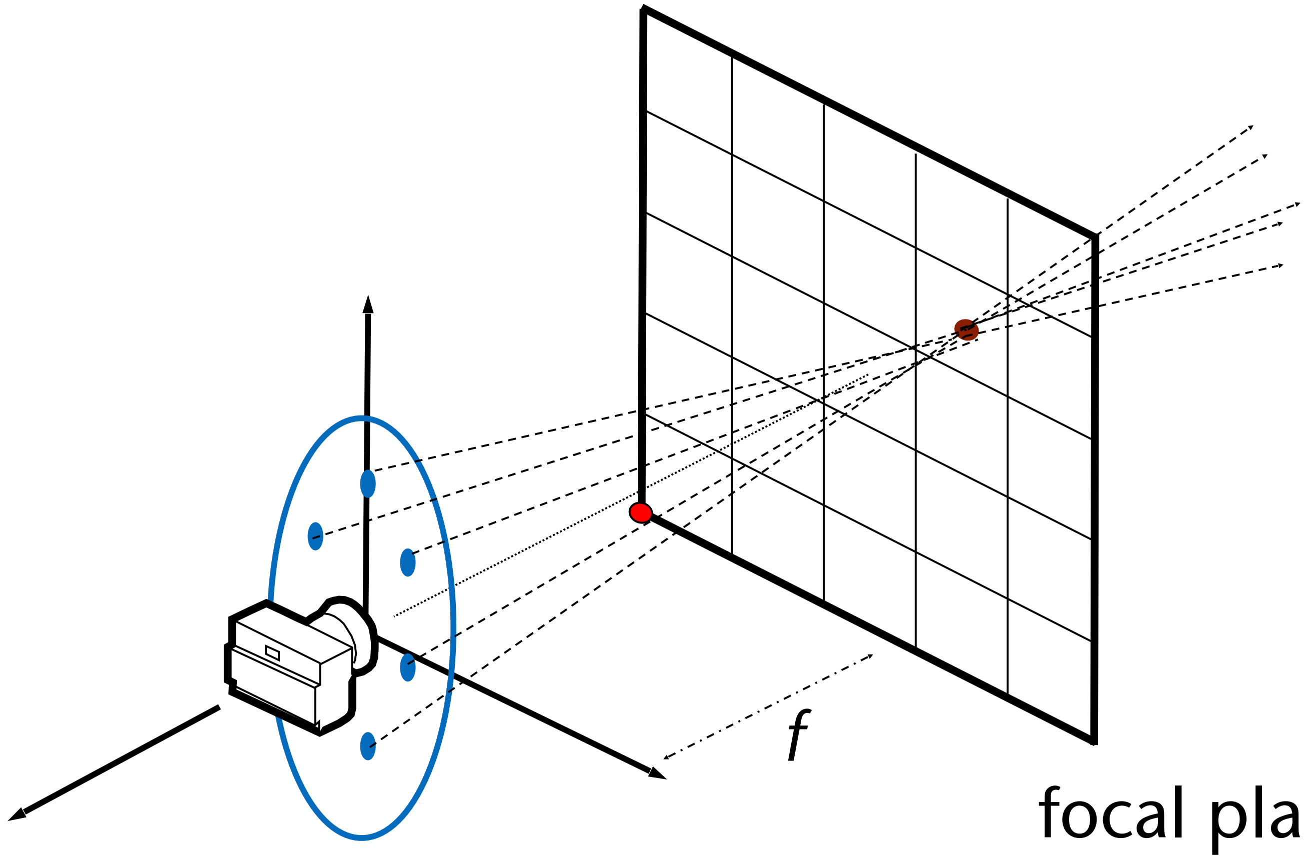
Depth-of-Field (Tiefen(un-)schärfe)

- So far: ideal pin-hole camera model
- For depth-of-field, we need to model real cameras



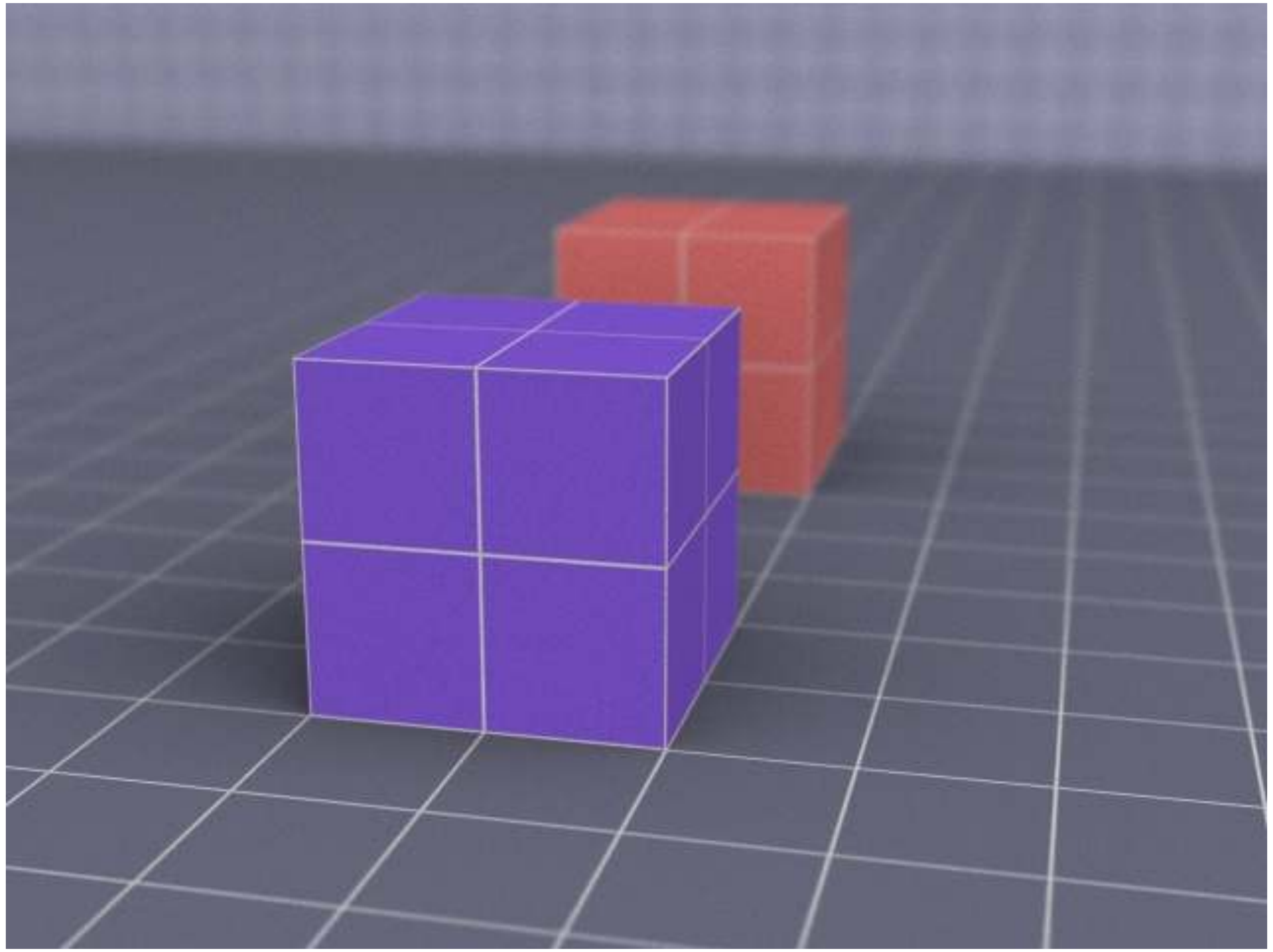


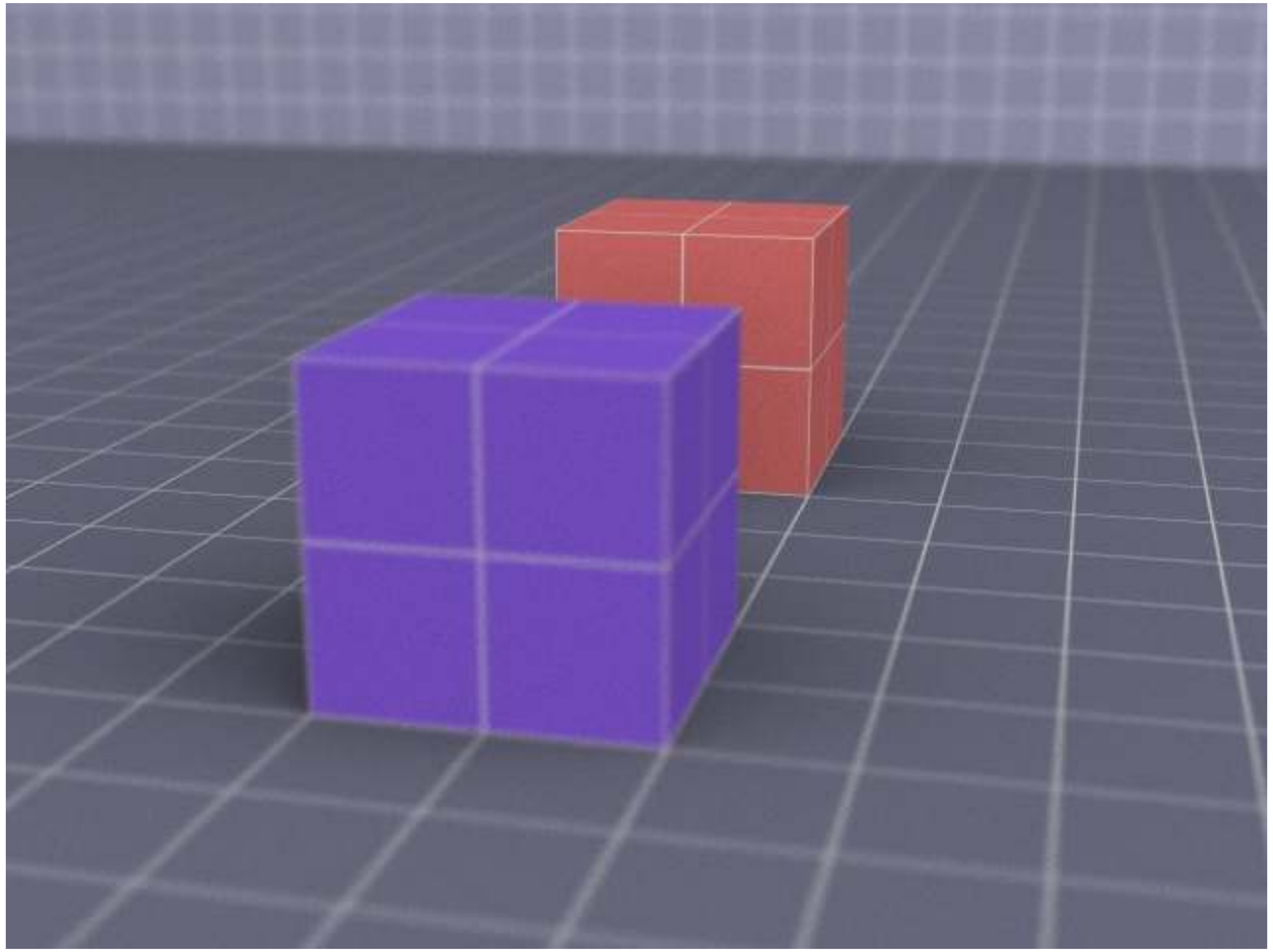
- A class `LensCamera` would generate rays like this:
  - Sample the whole shutter aperture by some distribution, shoot ray from each *sample* point through focal plane = image plane



- Remark:
  - Again, use one of the four sampling methods for sampling the disc (= shutter)

Examples



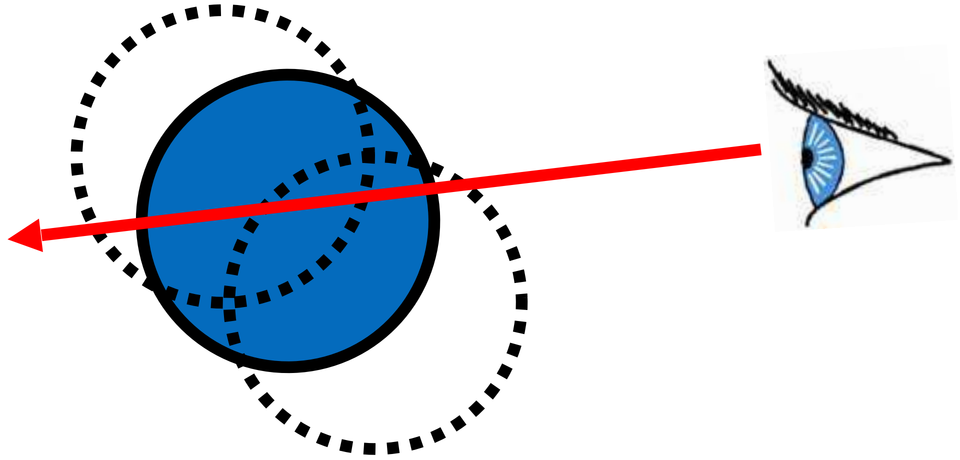


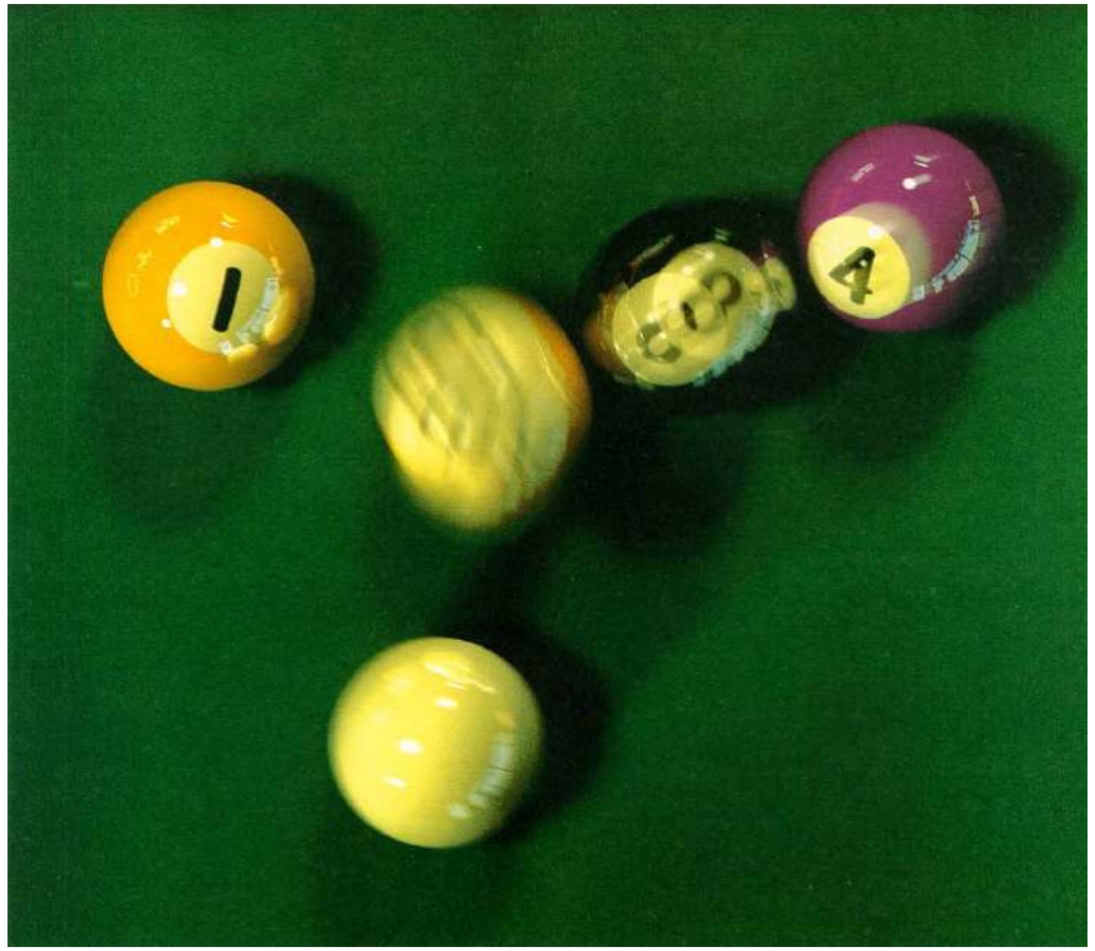
Shutter Speed Artifacts

- A slower shutter speed means the shutter is open for longer time
- This can result in motion blur

Motion Blur (Bewegungsunschärfe)

- Goal: compute "average" image for time interval $[t_0, t_1]$ during which objects move
- Sample time interval with several $t \in [t_0, t_1]$
- Shoot one primary ray per time t
- When computing the hit points (and secondary rays), use positions $P = P(t)$ for all objects
- Average color of all rays for one pixel





Common Myths

- Myth: rasterization is linear, ray-tracing is sub-linear in the number of primitives
  - Truth: rasterization uses LODs and various culling techniques
- Myth: rasterization is ugly, ray-tracing is clean
  - Truth: when optimized, both are ugly
- Myth: ray-tracing is embarrassingly parallel
  - Truth: not when optimization techniques are employed
  - Historical note: when rasterization came up, people thought that is embarrassingly parallel, too
- Myth: ray-tracing and rasterization are incompatible
  - Truth: they can co-exist just fine
  - Example: the film *Cars* by Pixar (reflections, for instance, were done using ray-tracing, rest was rasterization)