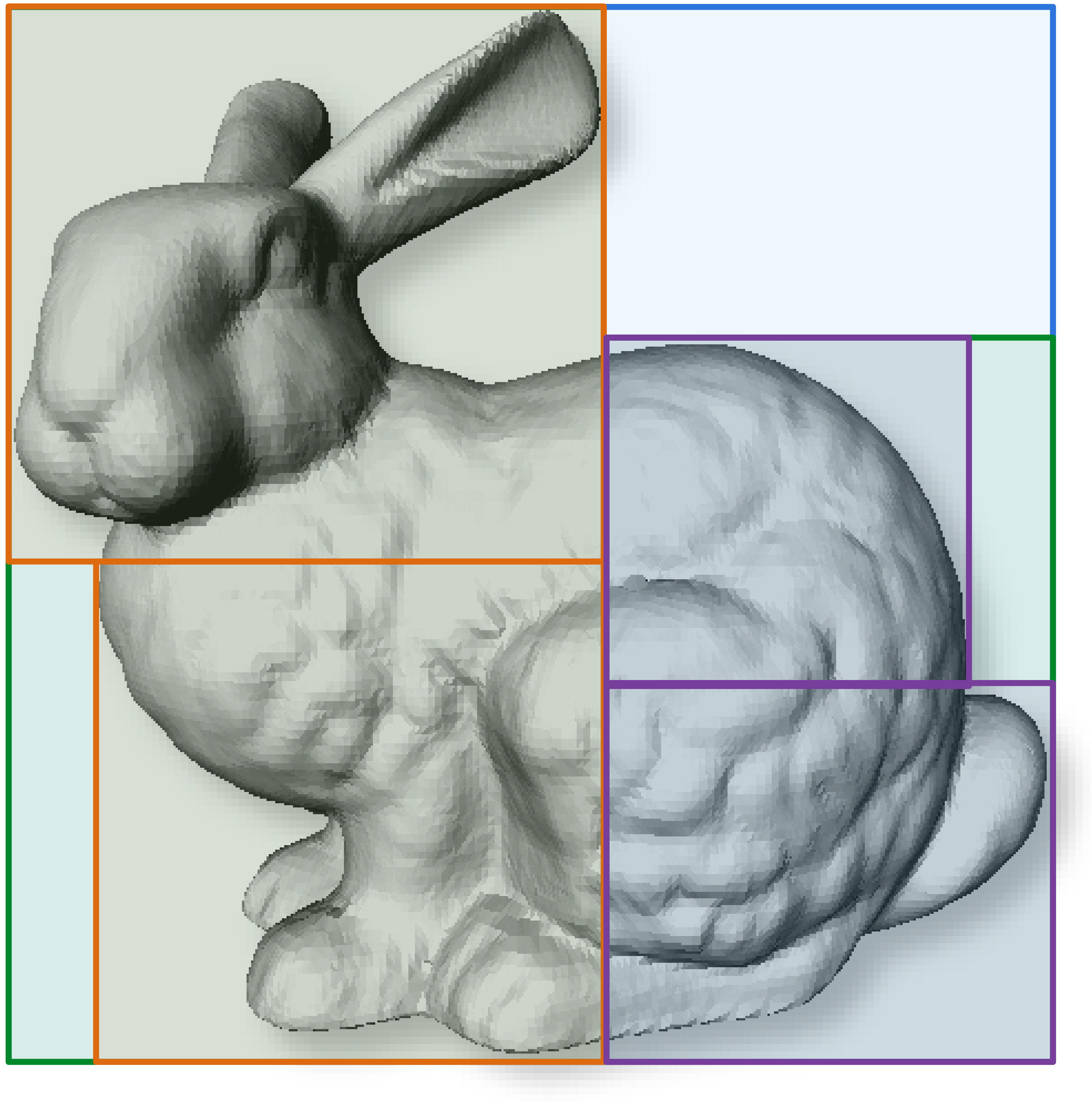
Ray tracing and geometric representations



15-468, 15-668, 15-868
Physics-based Rendering
Spring 2023, Lecture 2

http://graphics.cs.cmu.edu/courses/15-468

Course announcements

- Programming assignment 1 will be posted on Friday 1/27 and will be due two weeks later.
- Take-home quiz 1 will be posted on Tuesday 1/24 and will be due a week later.
- Office hours *for this week only* (will finalize starting next week based on survey results):
  - Yannis–Thursday 3-4 pm, Smith Hall (EDSH) 225.

Course announcements

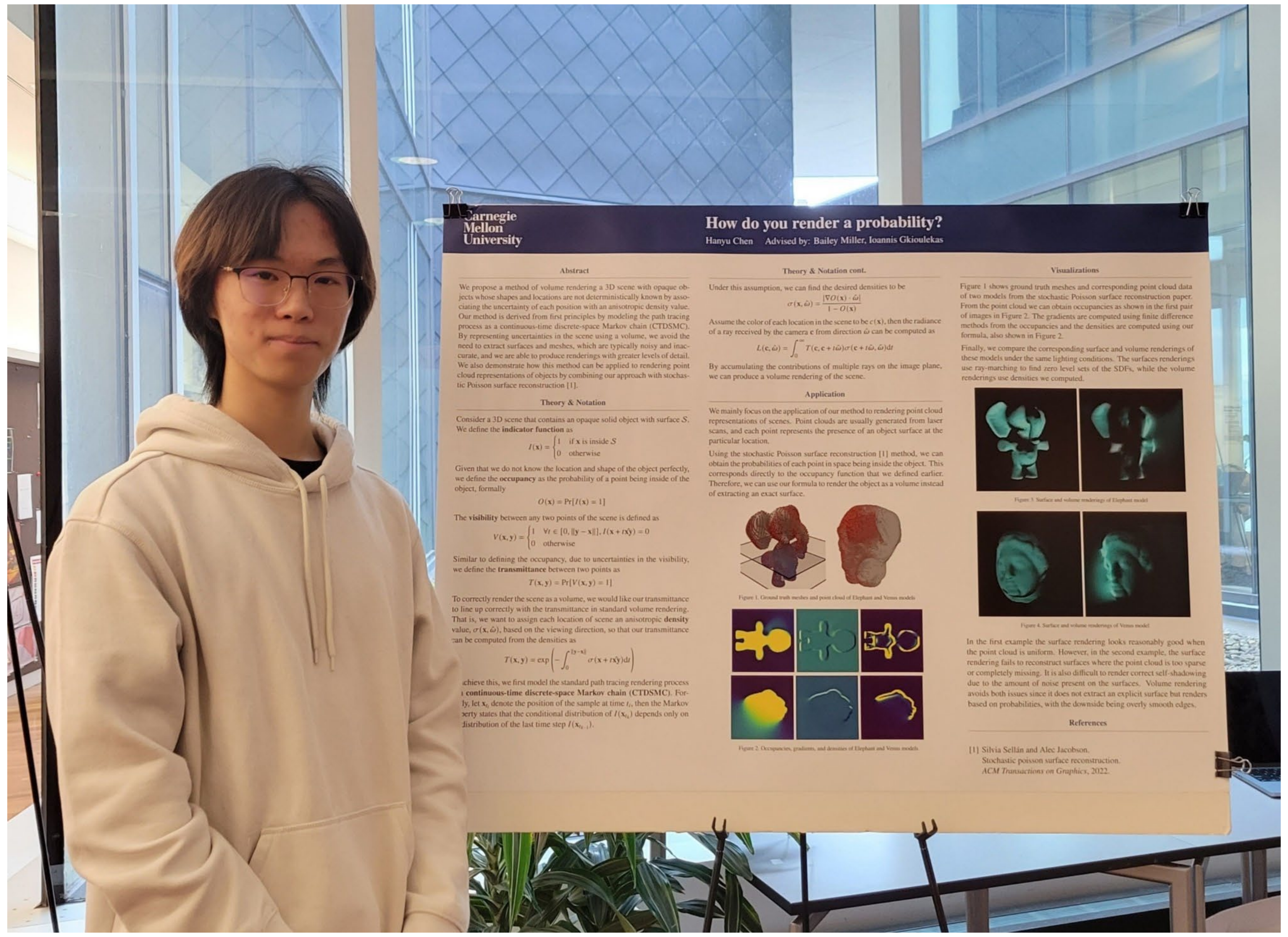
- Is anyone not on Piazza?

https://piazza.com/class/lctj7gng8wql4/

- Is anyone not on Canvas?

https://canvas.cmu.edu/courses/33678

- Is anyone not on Slack?



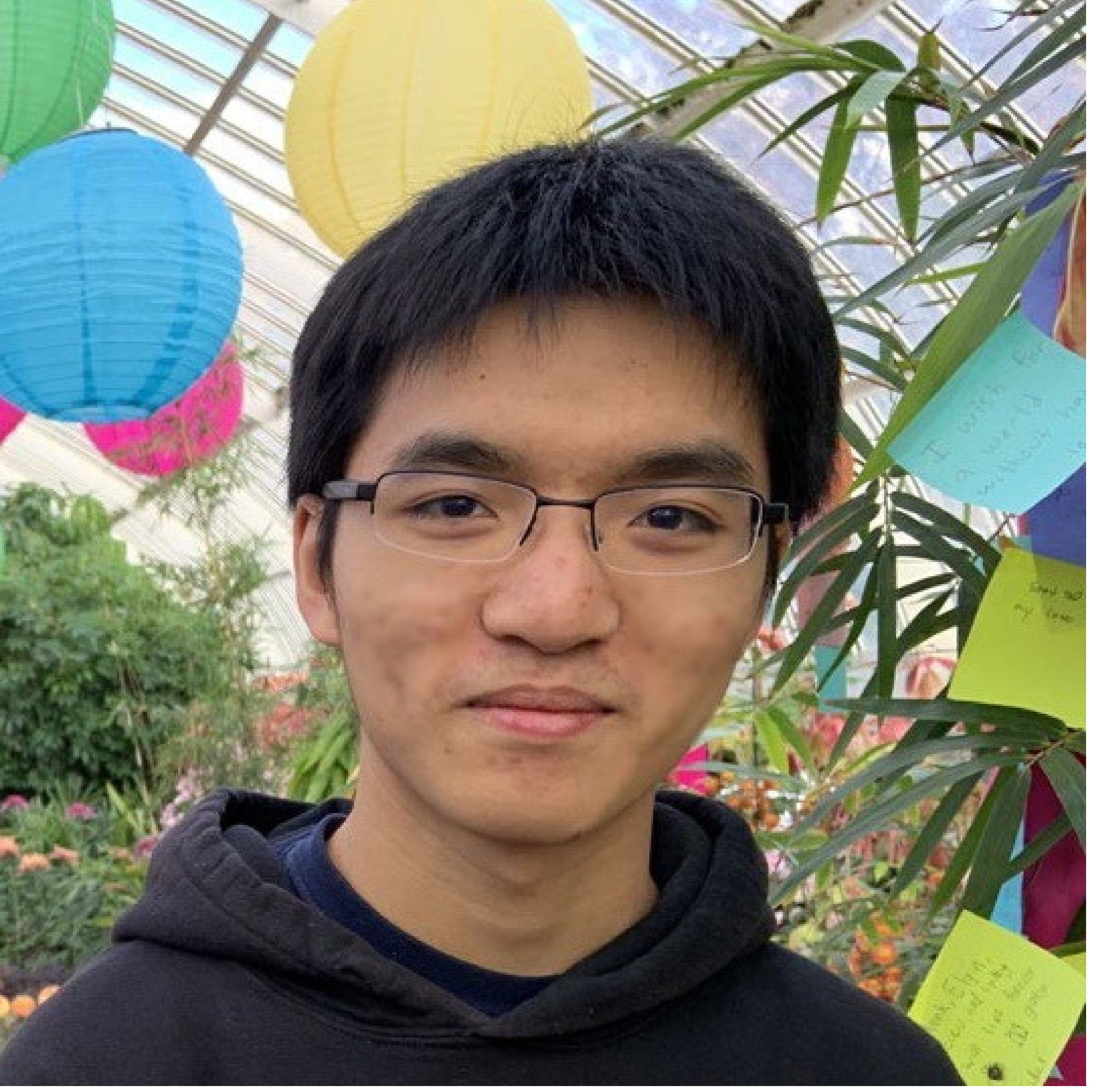
some random computer vision related work...

and some differentiable rendering related work...

Hanyu Chen

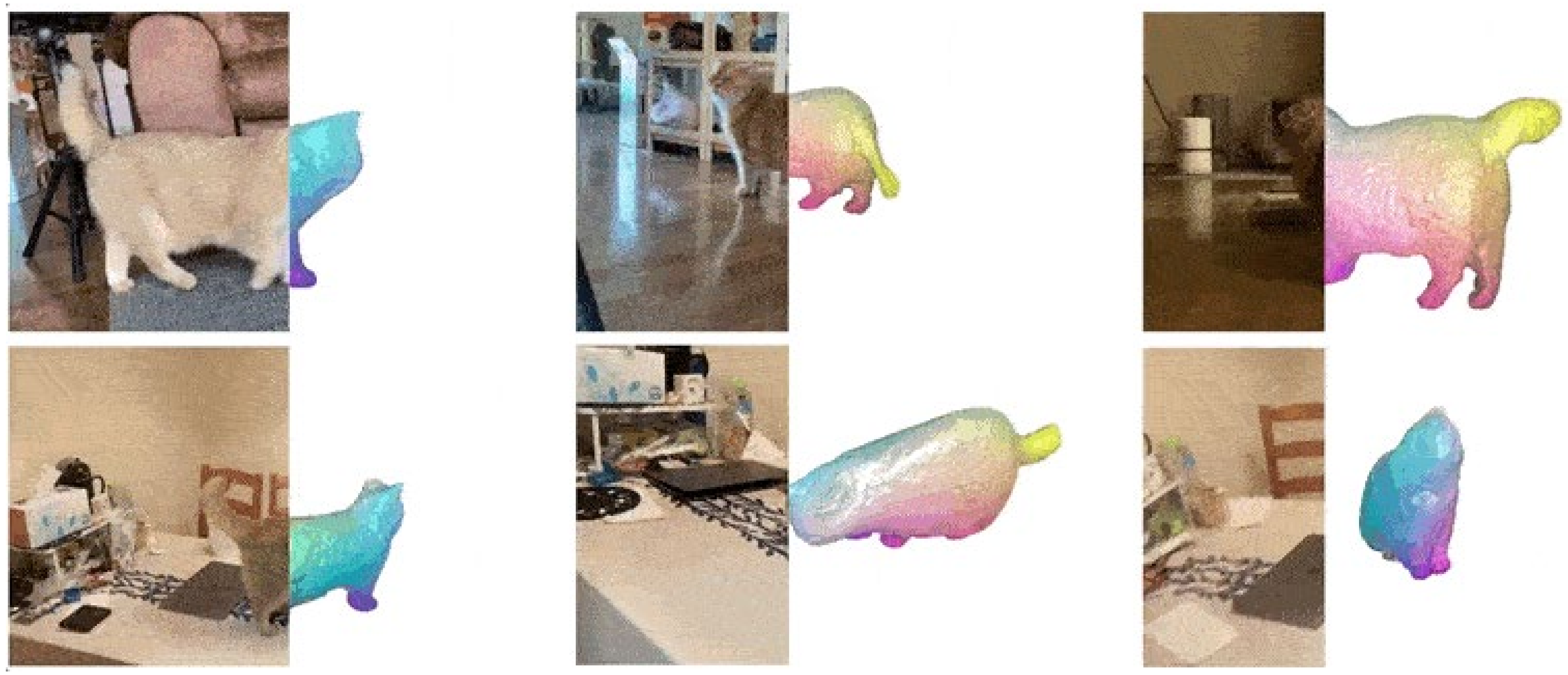
15-468 TA, Senior in Computer Science & Math

- Research interest broadly in computer vision/computer graphics/rendering
- Currently working with Yannis in neural rendering & surface reconstruction related research

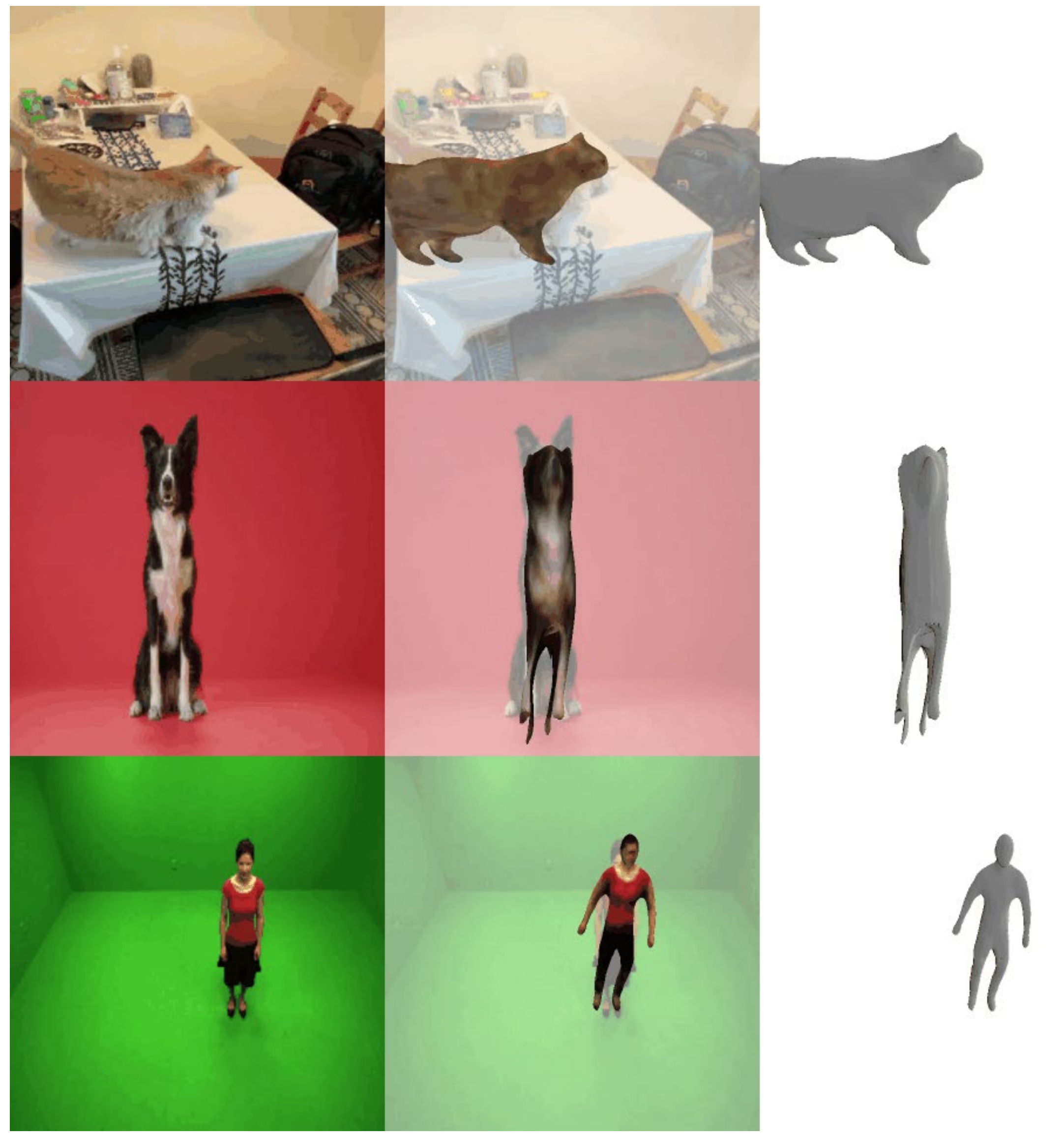


Jeff Tan (jefftan@andrew.cmu.edu)
15-468 TA and Senior in Computer Science

Research: Neural rendering for real-time dynamic 3D reconstruction
Advised by Prof. Deva Ramanan



Yang et al. (CVPR 2022)



Tan et al. (in submission)

Overview of today’s lecture

- Introduction to ray tracing.
- Intersections with geometric primitives.
- Triangular meshes.

Slide credits

Most of these slides were directly adapted from:

- Wojciech Jarosz (Dartmouth).

Two forms of 3D rendering

Rasterization: object point to image plane

- start with a 3D object point
- apply transforms
- determine the 2D image plane point it projects to

Ray tracing: image plane to object point

- start with a 2D image point
- generate a ray
- determine the visible 3D object point

Inverse processes

Two forms of 3D rendering

Rasterization

```
for (each triangle)
    for (each pixel)
        if (triangle covers pixel)
            keep closest hit
```

Triangle-centric

Ray tracing

```
for (each pixel or ray)
    for (each triangle)
        if (ray hits triangle)
            keep closest hit
```

Ray-centric

After a slide by Frédo Durand

Rasterization advantages

Modern scenes are more complicated than images

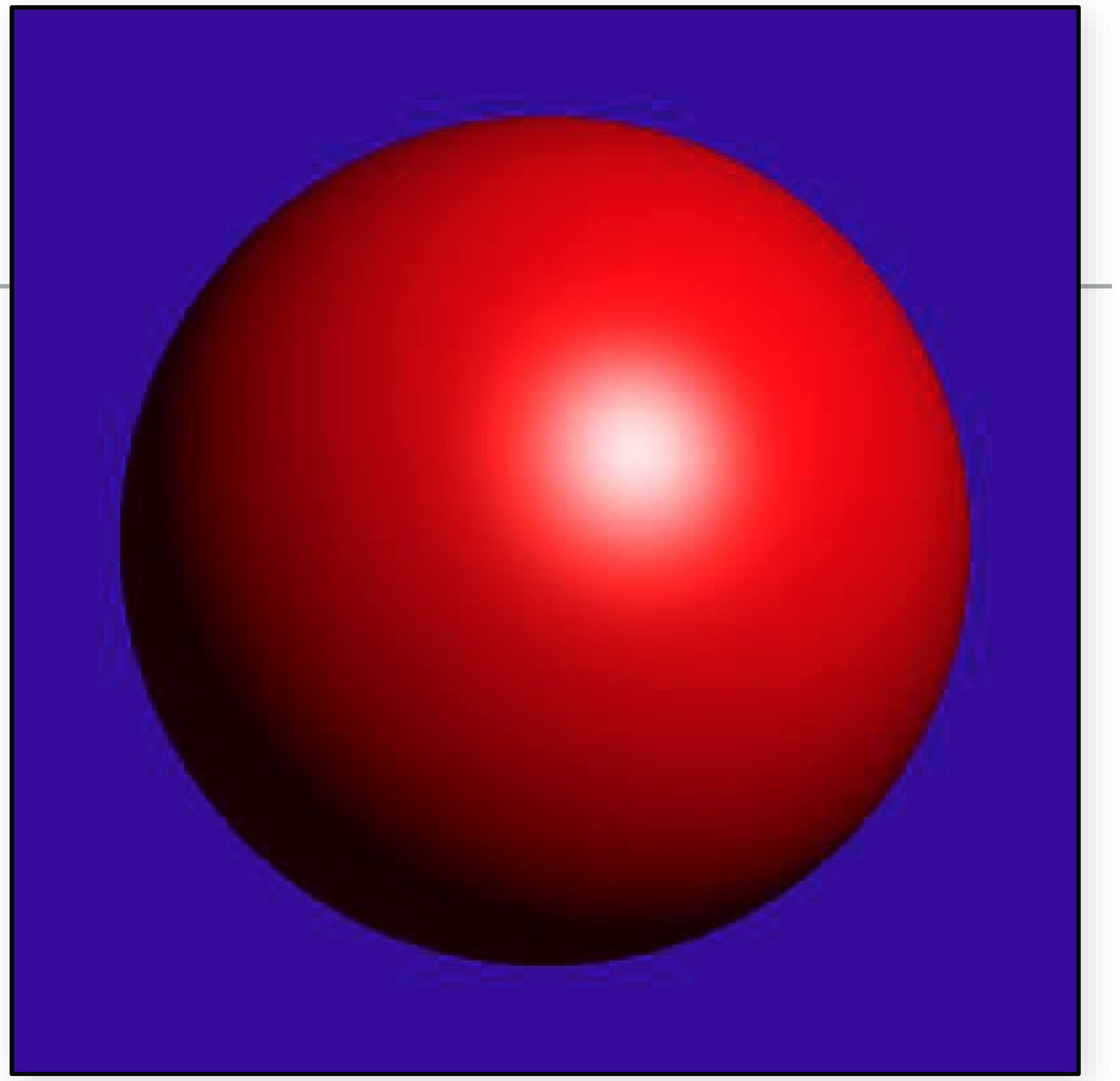
- A 1920x1080 frame (1080p) at 64-bit color and 32-bit depth per pixel is 24 MB (not that much)
    - of course, if we have more than one sample per pixel this gets larger, but e.g. 4x supersampling is still a relatively comfortable ~100 MB
- Our scenes are routinely larger than this
    - This wasn’t always true

A rasterization-based renderer can *stream* over the triangles, no need to keep entire dataset around

- Allows parallelism and optimizations of memory systems

After a slide by Frédo Durand

Rasterization limitations



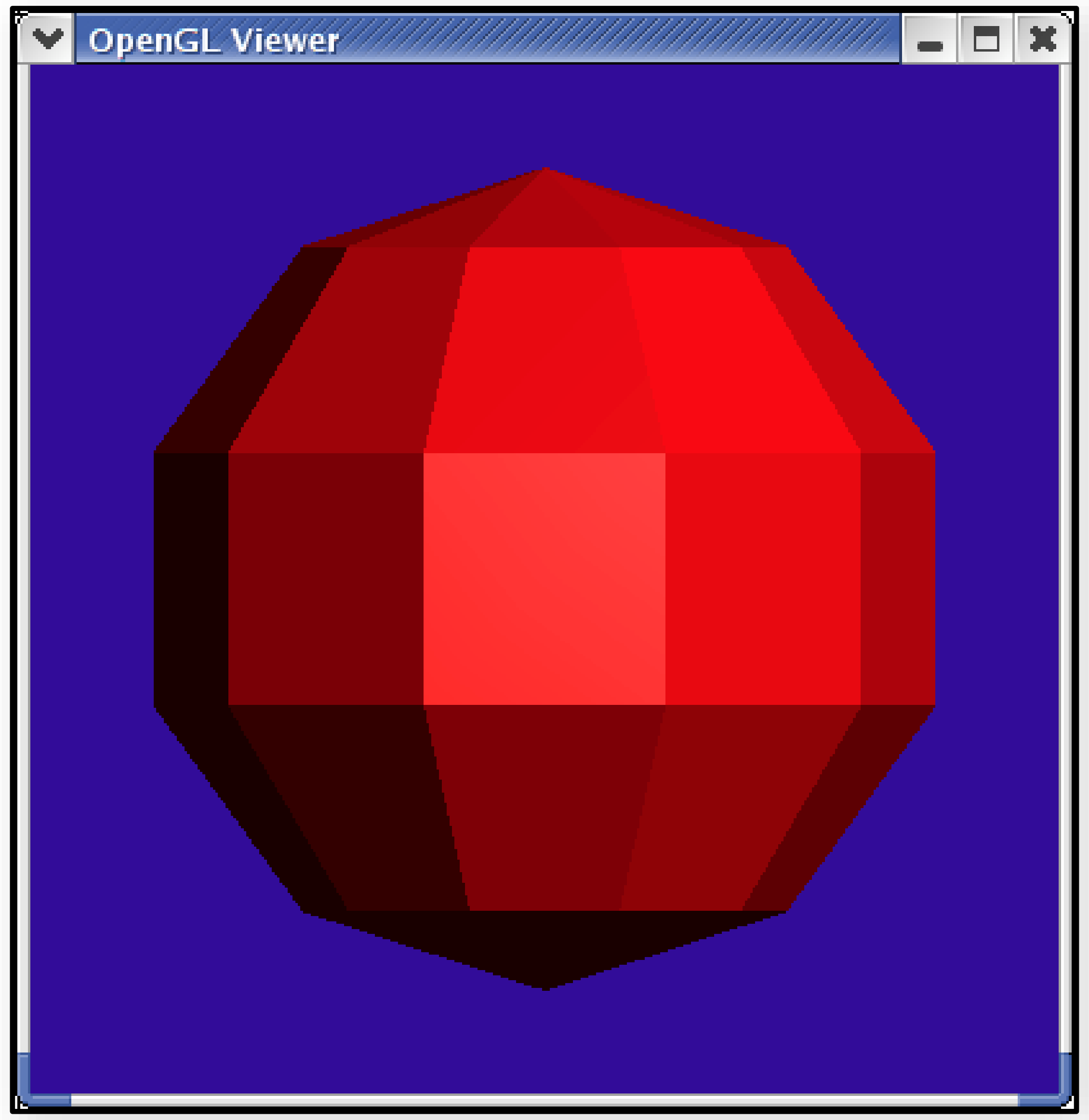
Restricted to scan-convertible primitives

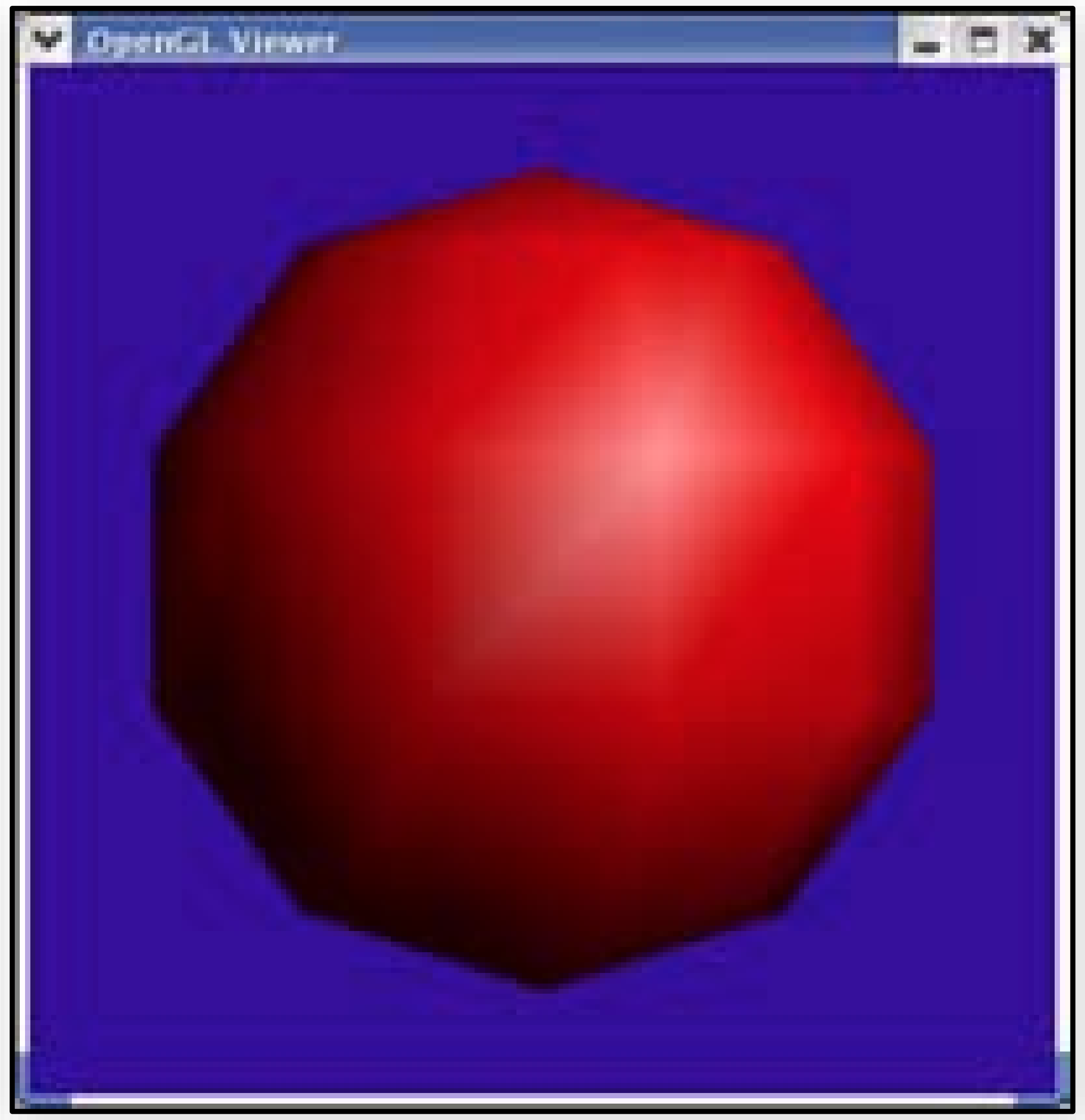
- Pretty much: triangles

Faceting, shading artifacts

- This is largely going away with programmable per-pixel shading, though

No unified handling of shadows, reflection, transparency





After a slide by Frédo Durand

Ray/path tracing

Advantages

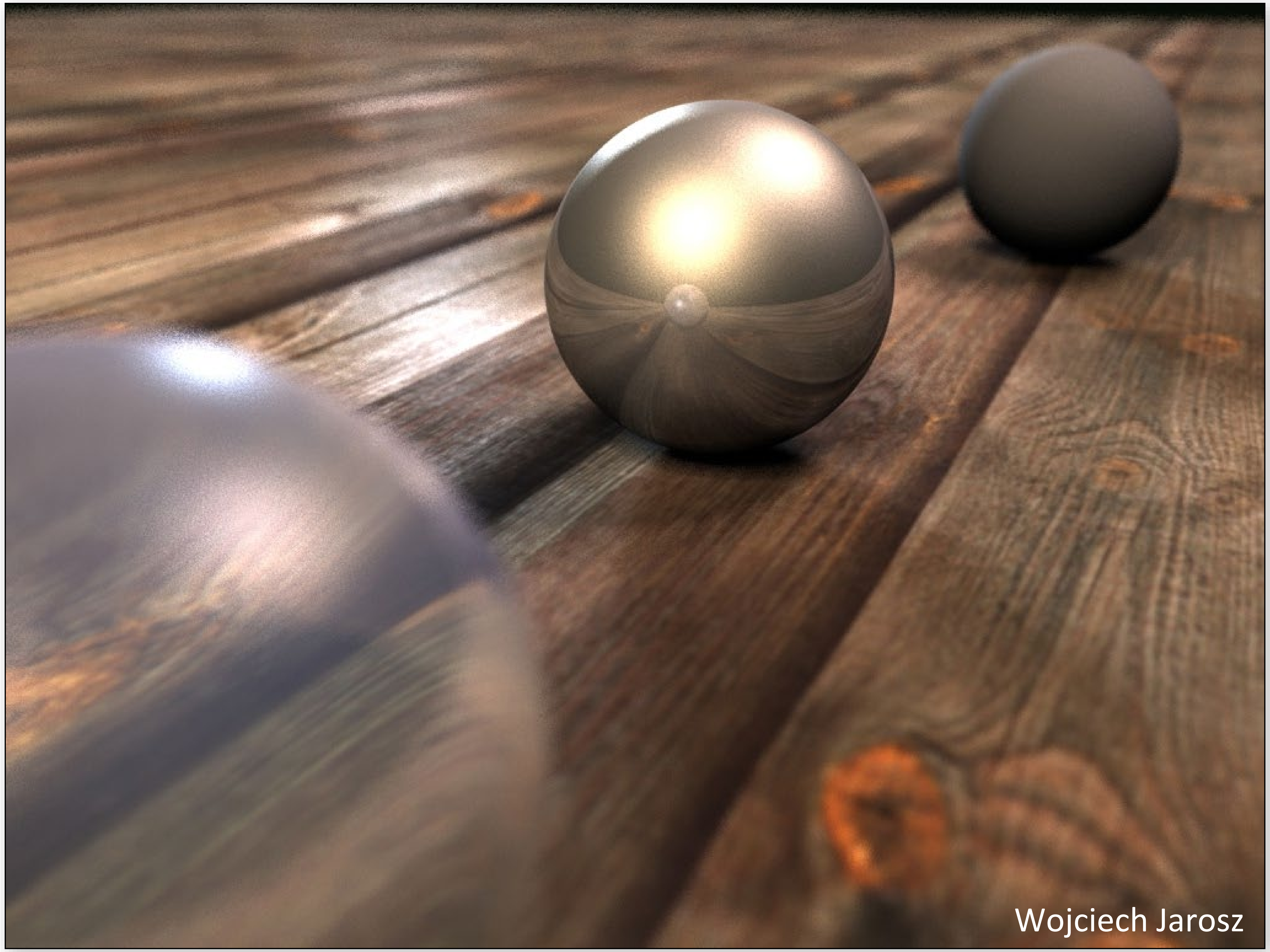
- Generality: can render anything that can be intersected with a ray
- Easily allows recursion (shadows, reflections, etc.)

Disadvantages

- Hard to implement in hardware (lacks computation coherence, must fit entire scene in memory, bad memory behavior)
    - Not such a big point anymore given general purpose GPUs
- Has traditionally been too slow for interactive applications
- Both of the above are changing rather rapidly right now!

After a slide by Frédo Durand

A ray-traced image



Ray tracing today

Rapid change in film industry

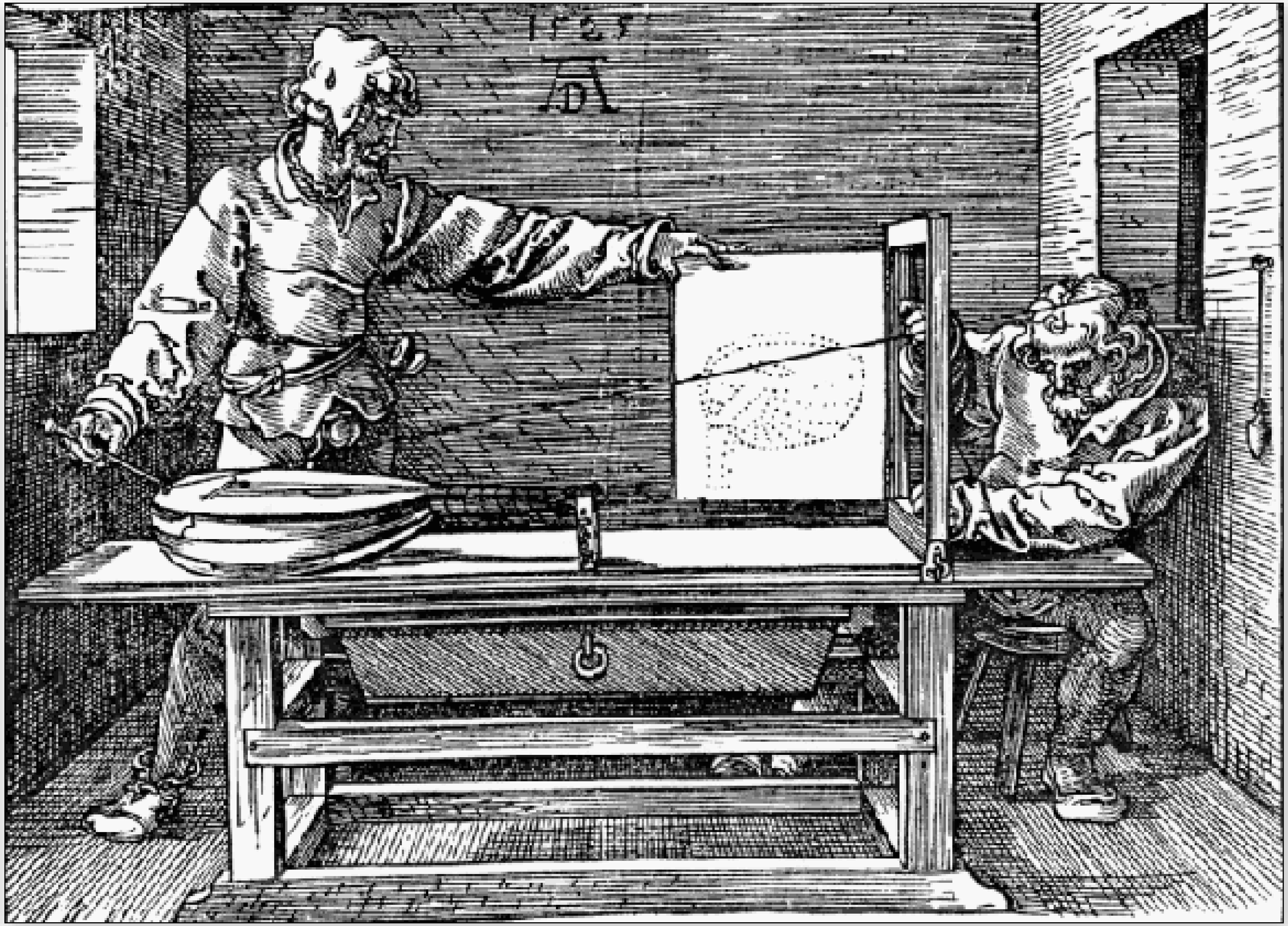
2008:

- Most CGI in films rendered using micro-polygon rasterization.
- "You'd be crazy to render a full-feature film with ray/path tracing."
- Ray/path tracing mostly interesting to academics

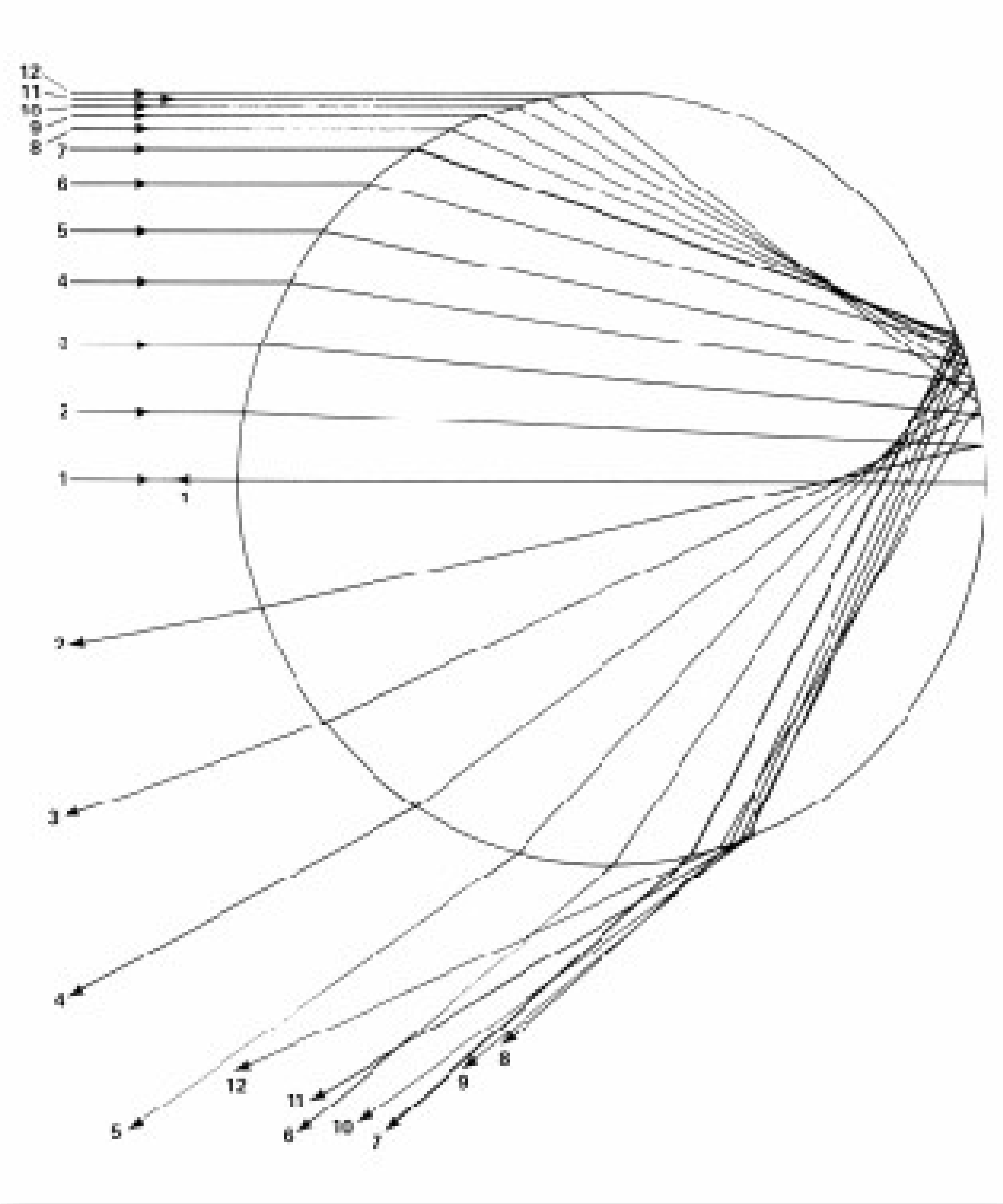
2018:

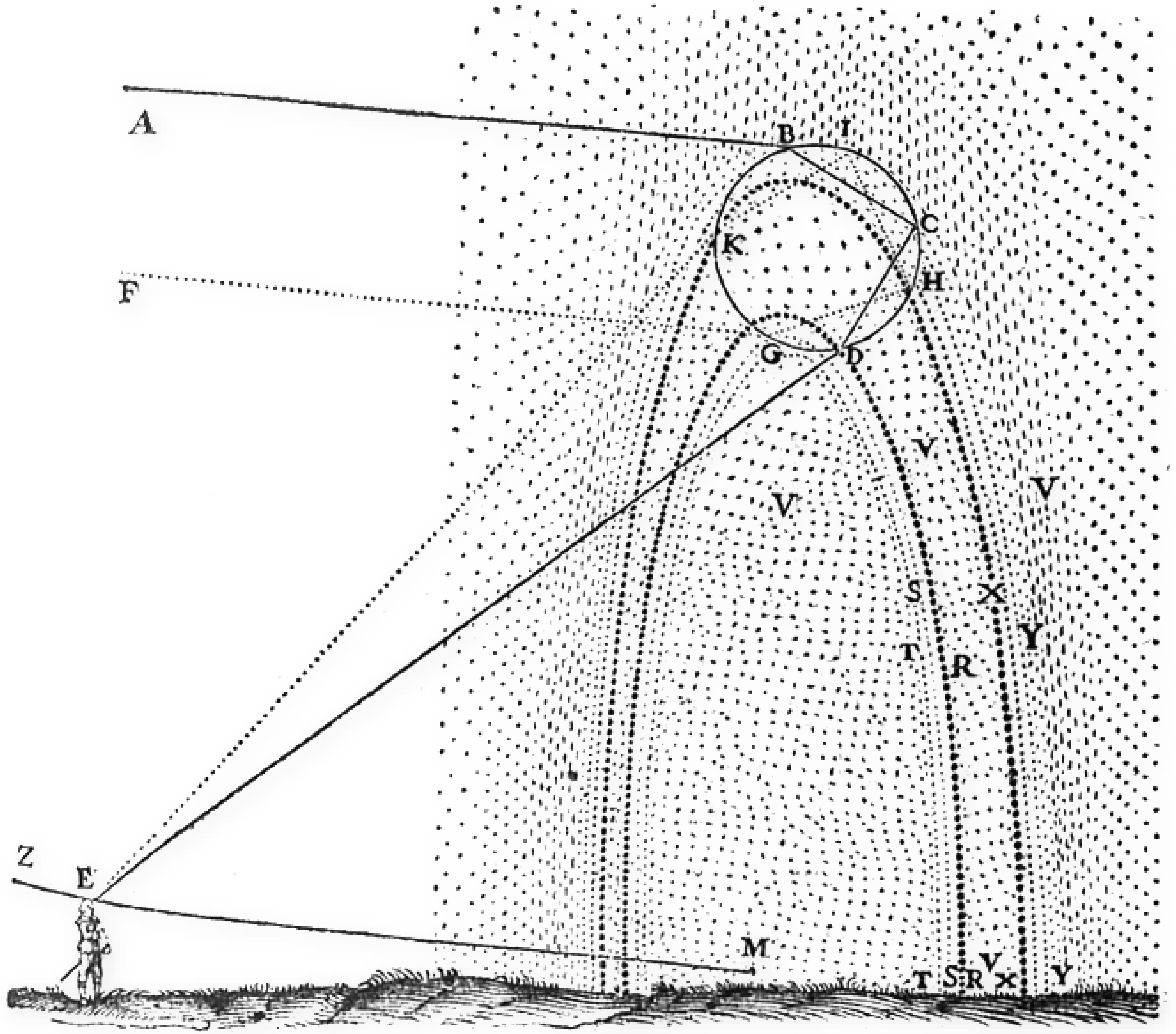
- Most major films now rendered using ray/path tracing.
- "You'd be crazy ***not*** to render a full-feature film using path tracing."

Albrecht Dürer (1525)

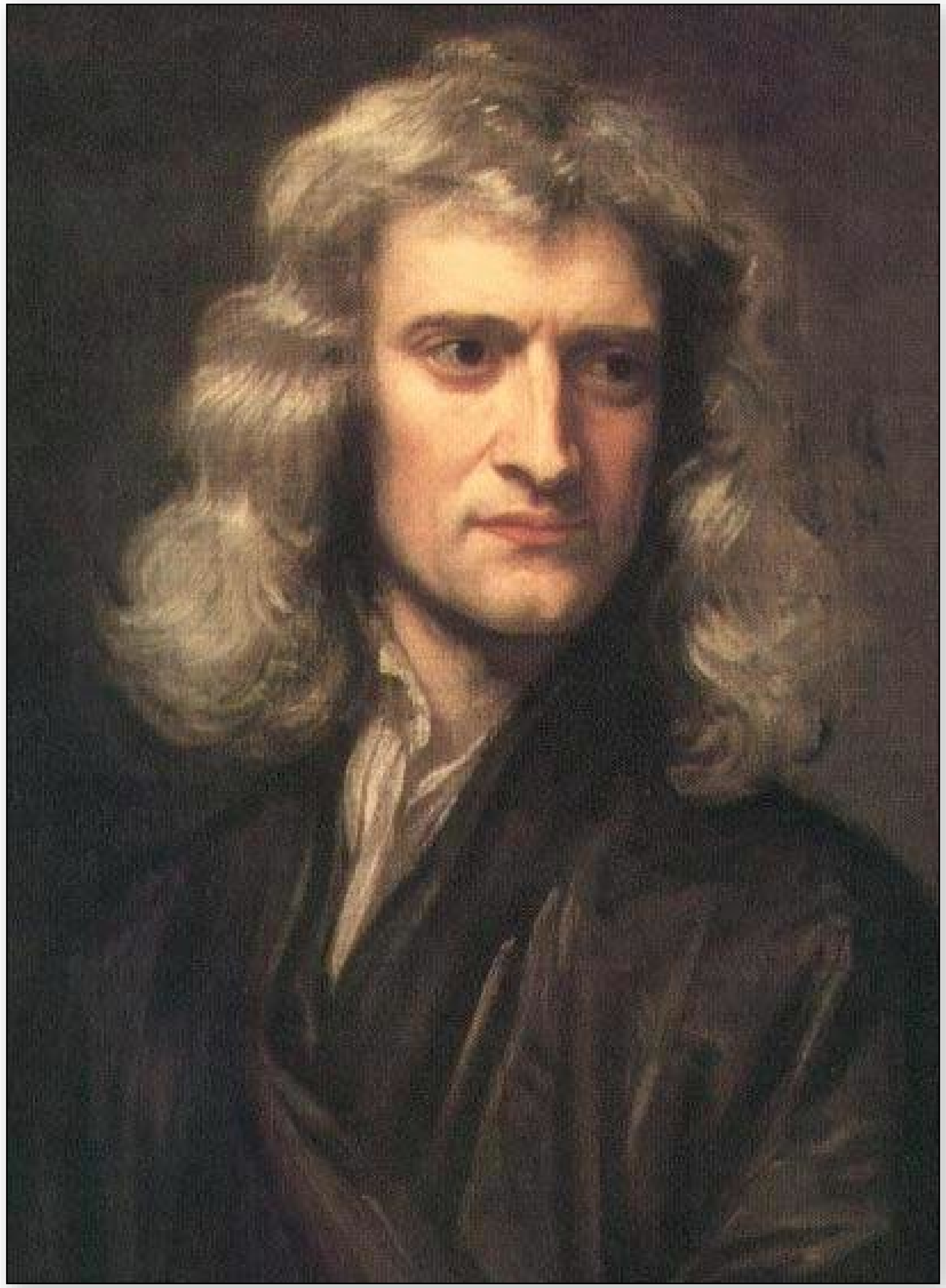


René Descartes (1650)

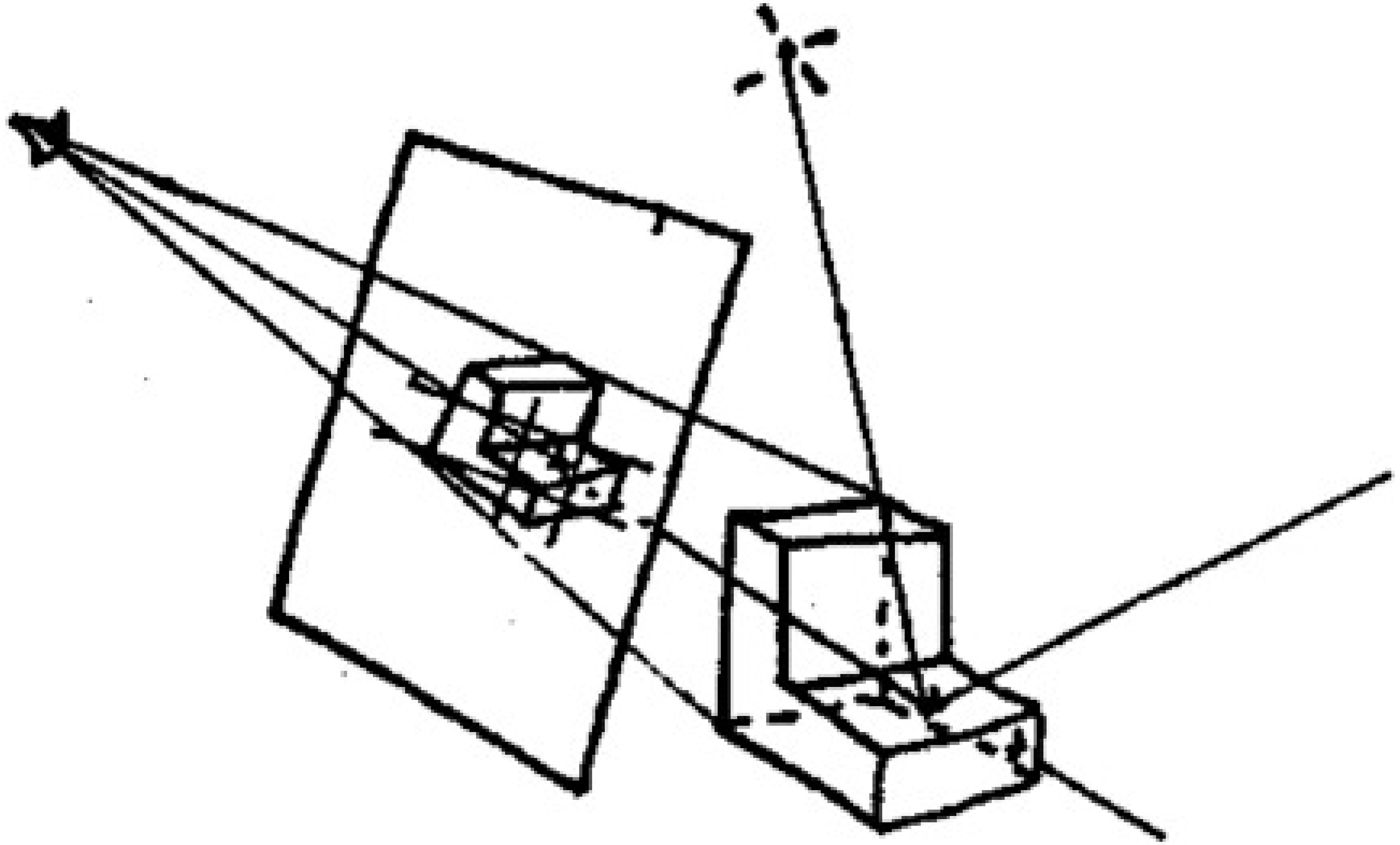




Isaac Newton (1670)



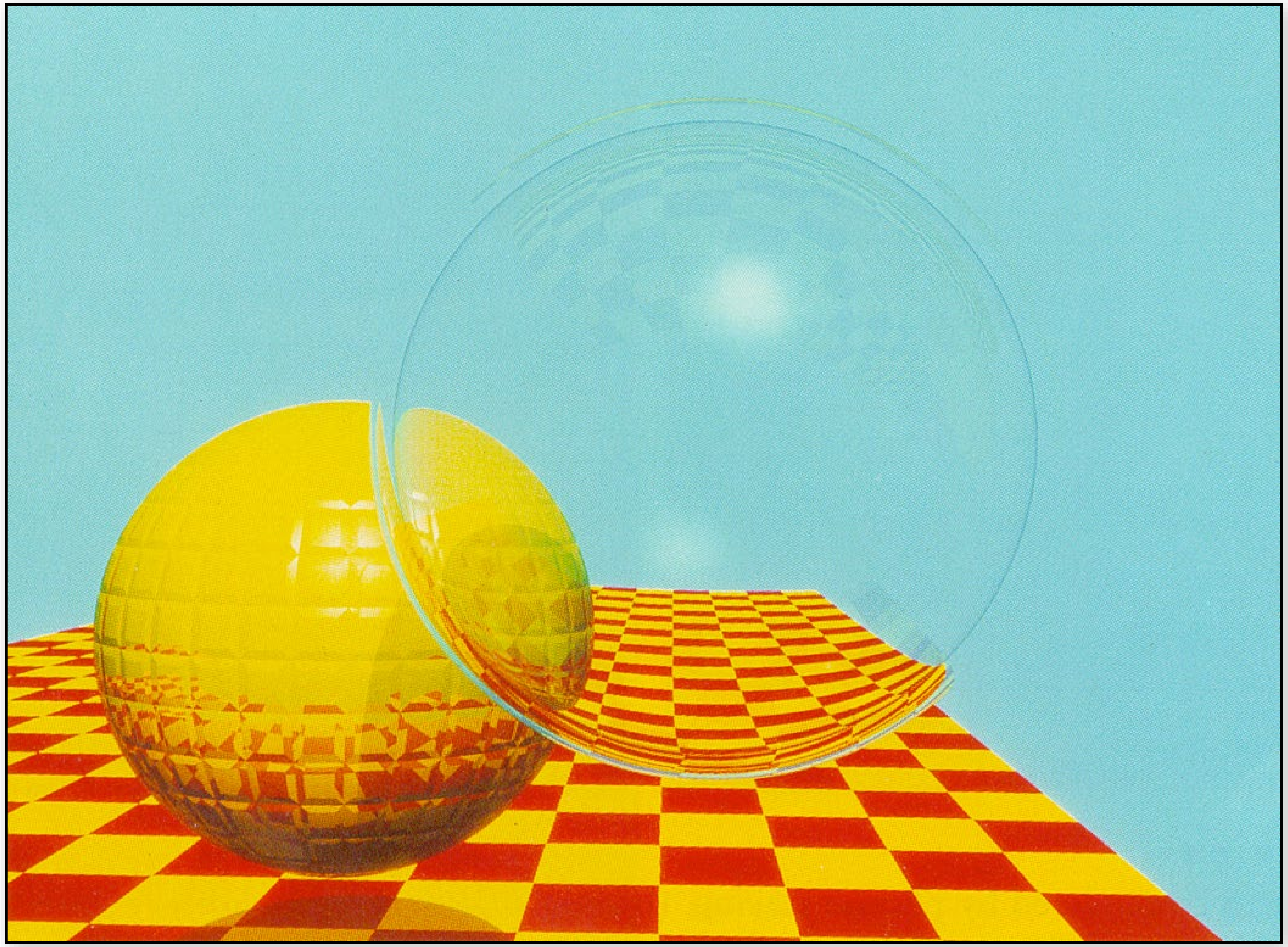
Appel (1968)



Ray casting

- Generate an image by sending one ray per pixel
- Check for shadows by sending a ray towards the light

Whitted (1979)



recursive ray tracing (reflection & refraction)

Light Transport - Assumptions

Geometric optics:

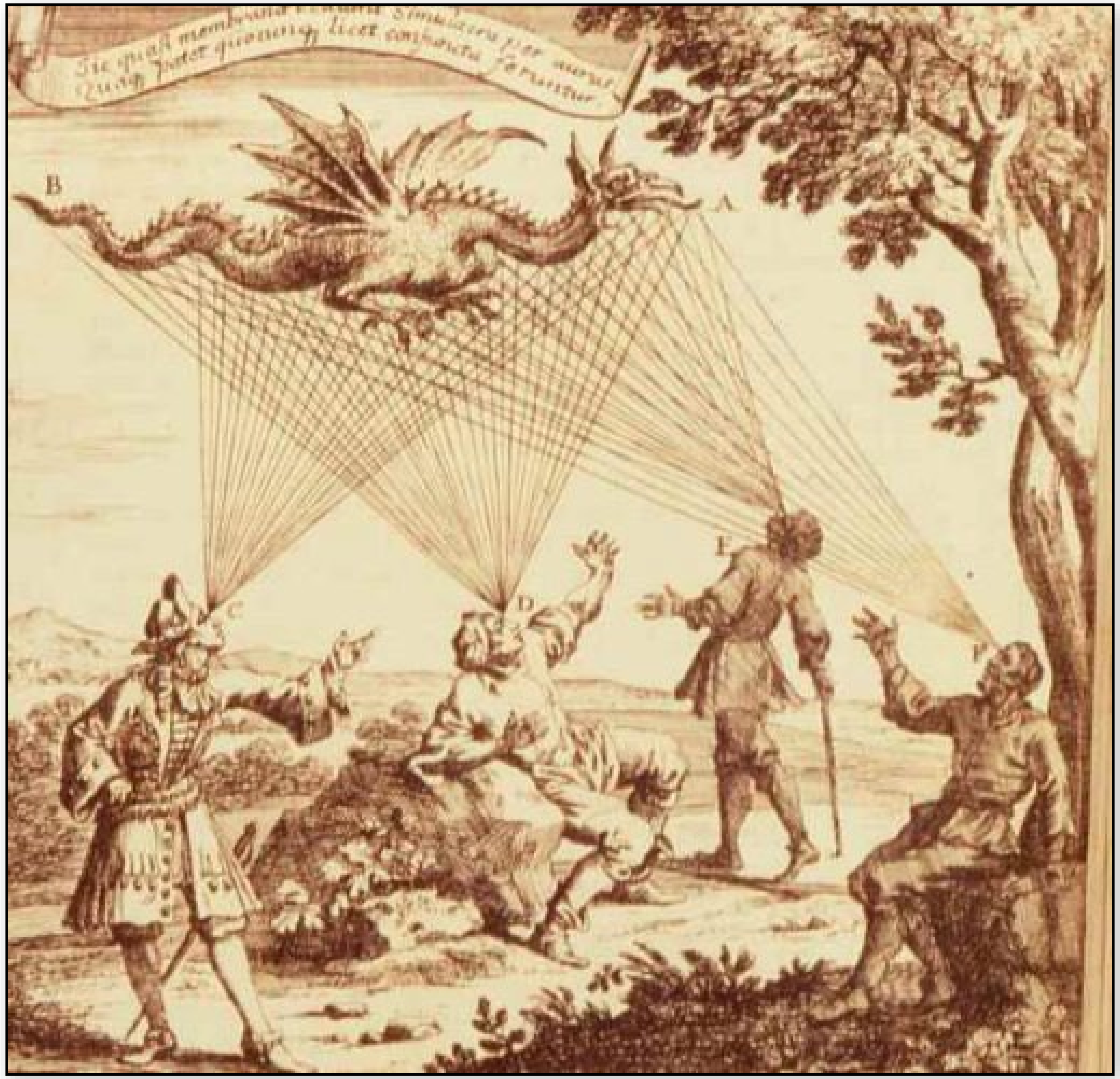
- no diffraction, no polarization, no interference

Light travels in a straight line in a vacuum

- no atmospheric scattering or refraction
- no gravity effects

Color can be represented as three numbers: (R,G,B)

Emission theory of vision



Eyes send out “feeling rays” into the world

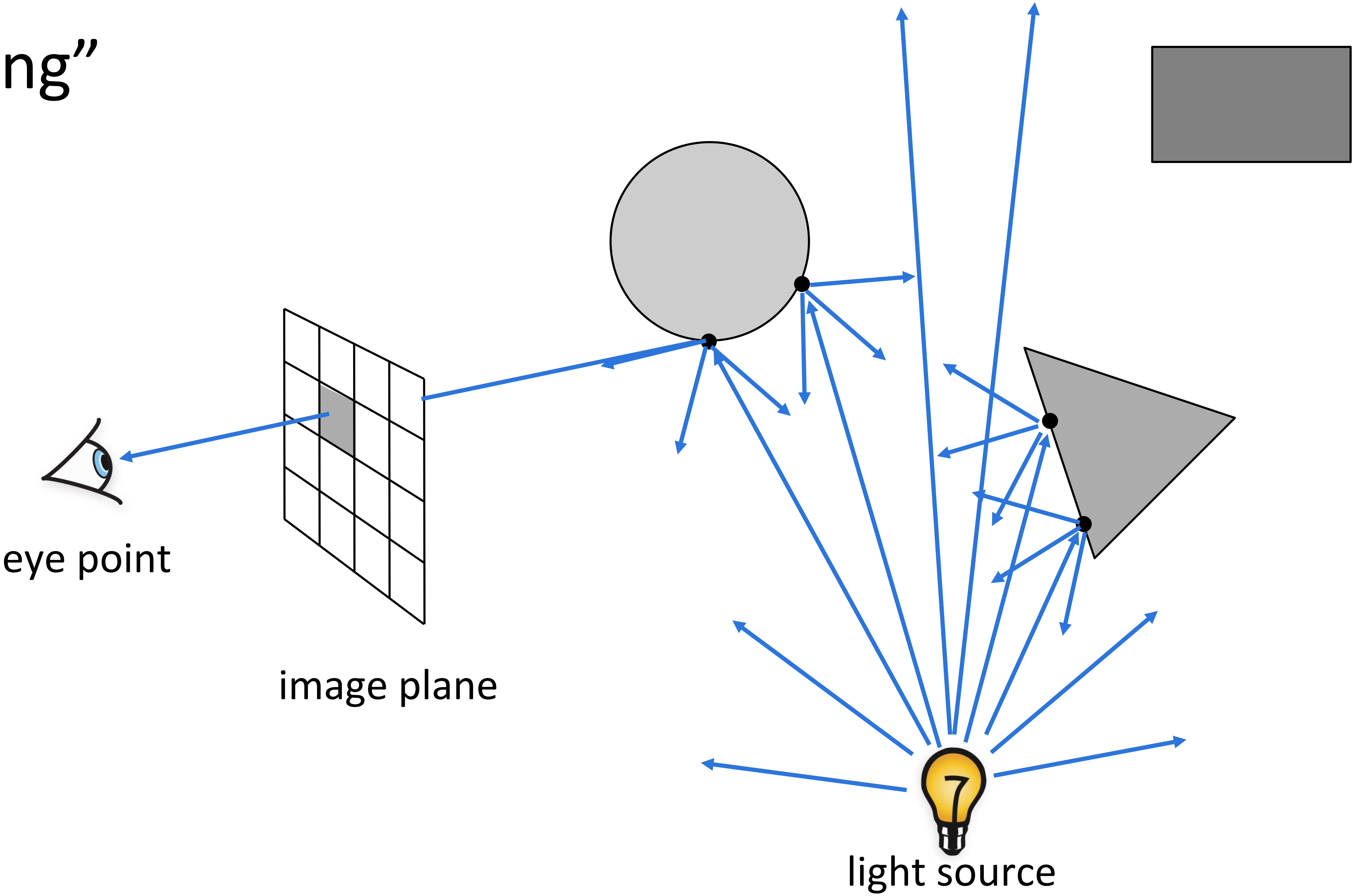
Supported by:

- Ancient greeks
- 50% of US college students*

*http://www.ncbi.nlm.nih.gov/pubmed/12094435?dopt=Abstract

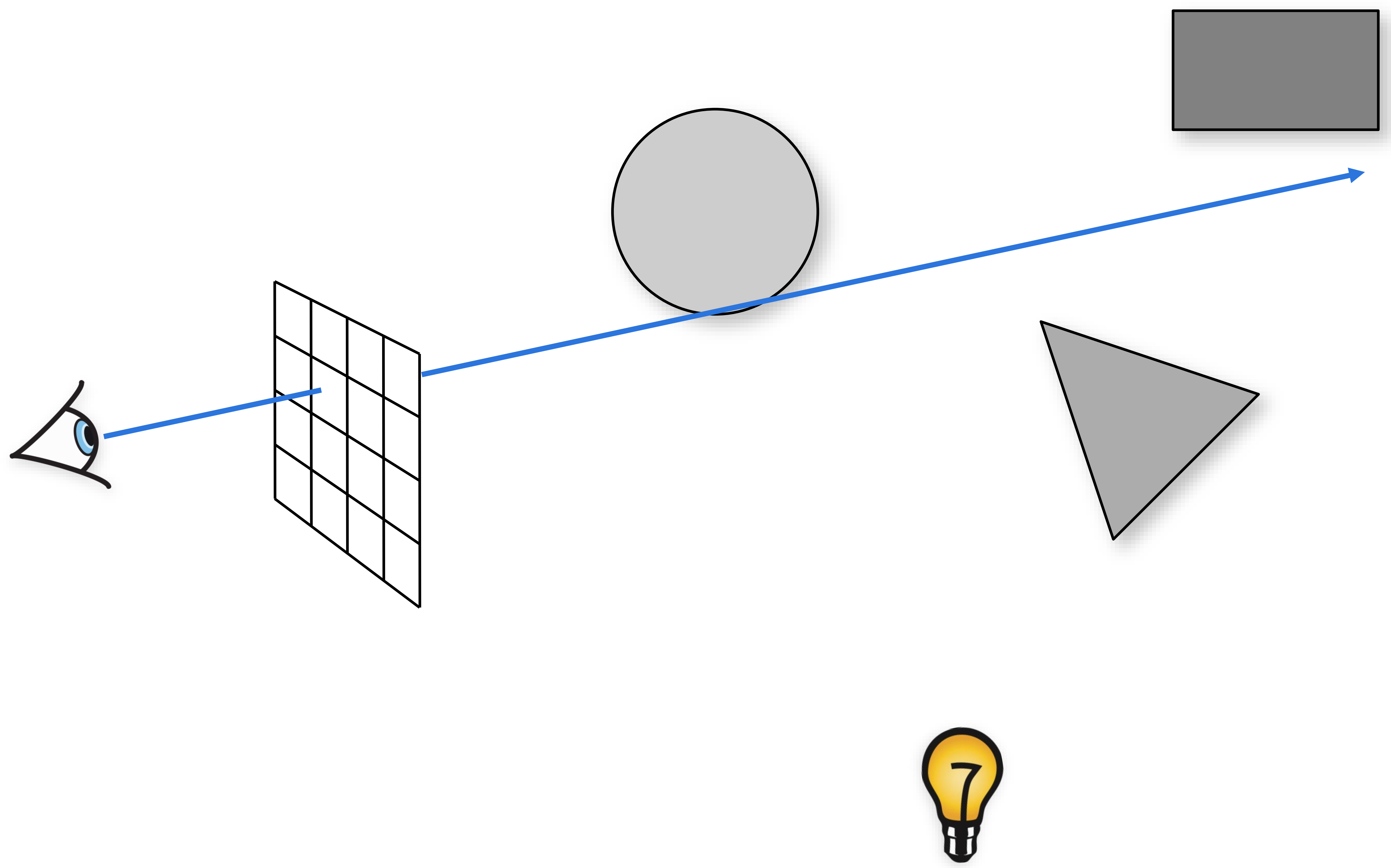
Ray Tracing - Overview

"light tracing"



Basic Ray Tracing Pipeline

Ray Generation

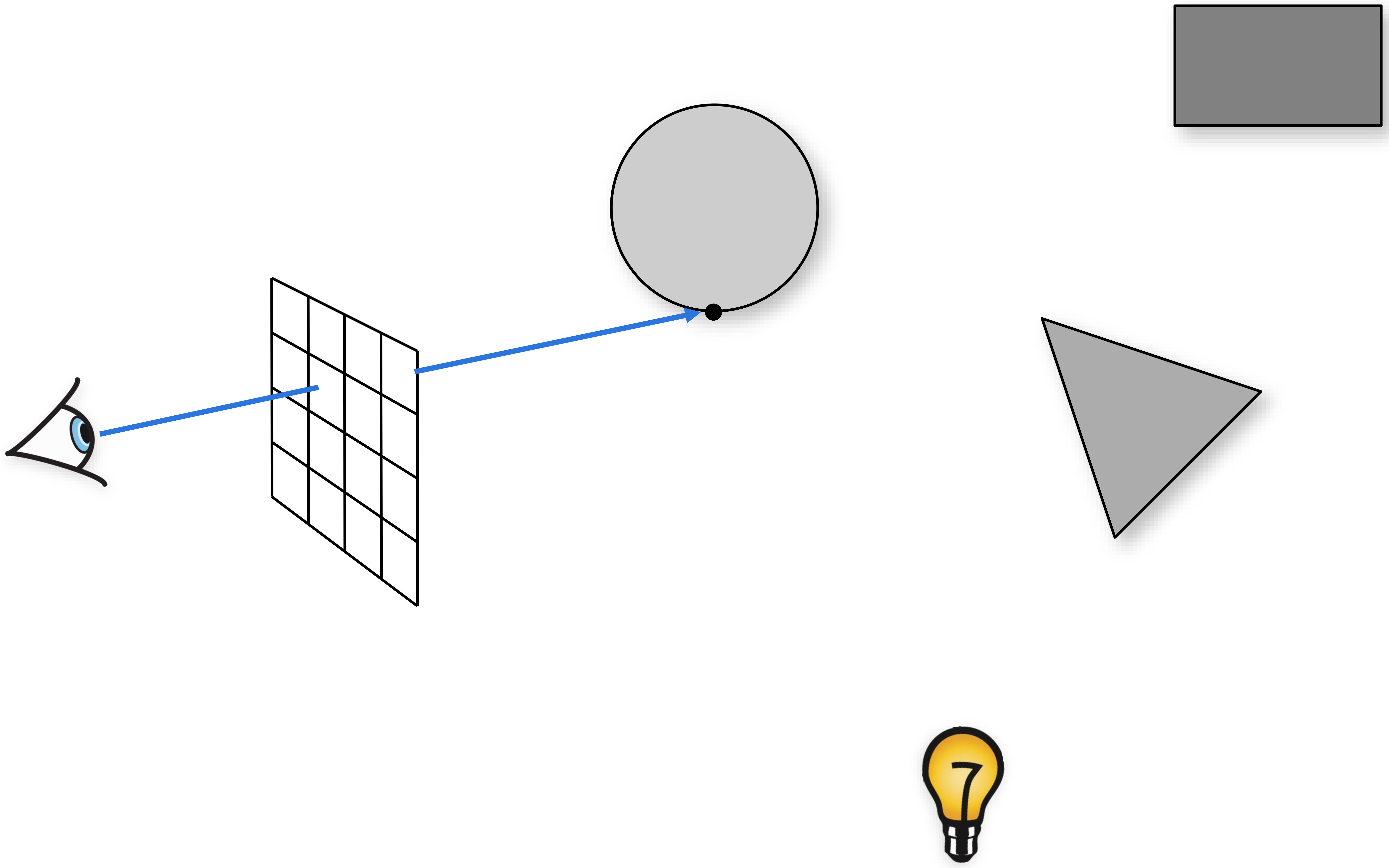


Basic Ray Tracing Pipeline

Ray Generation

↓

Intersection



Basic Ray Tracing Pipeline

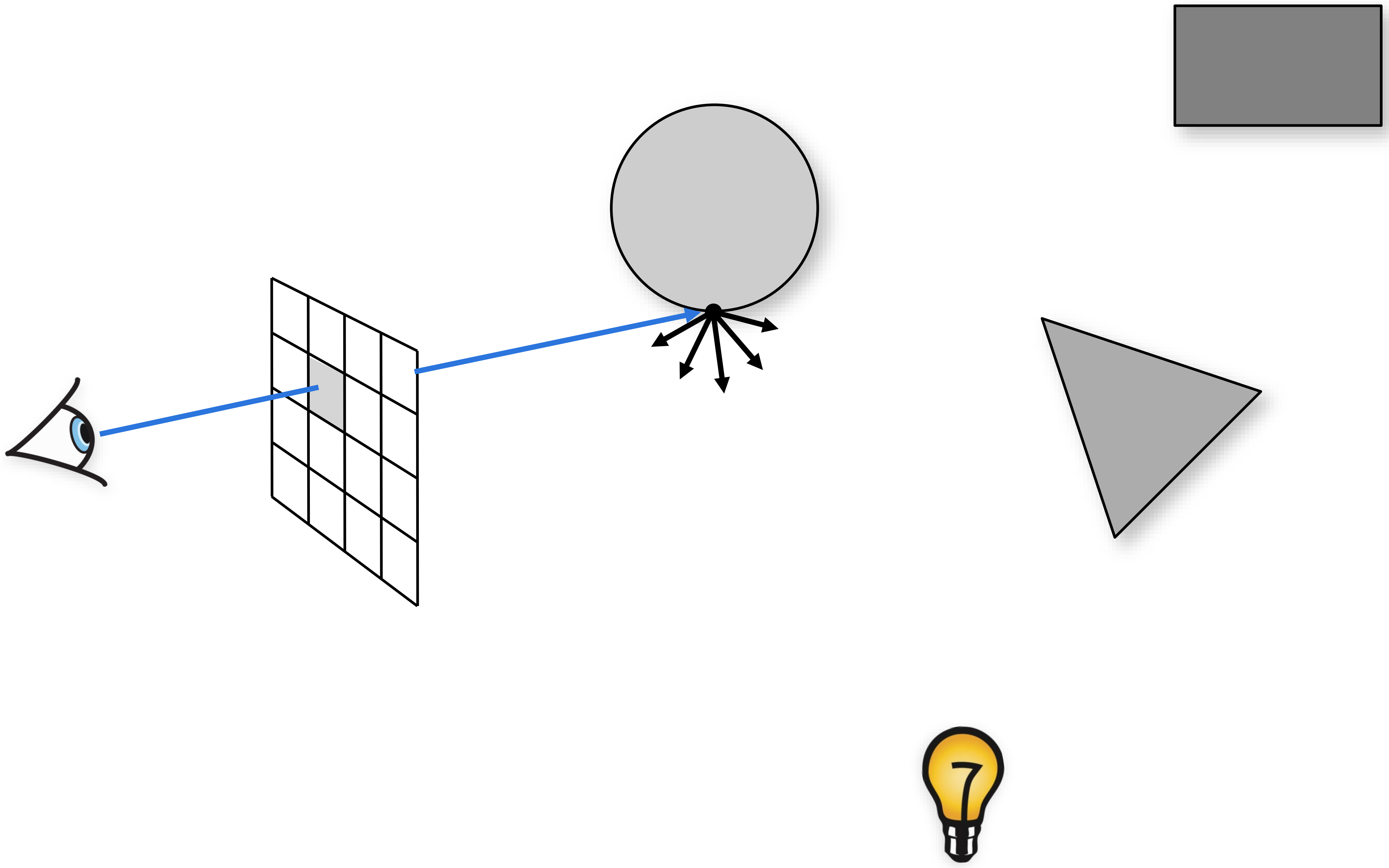
Ray Generation

↓

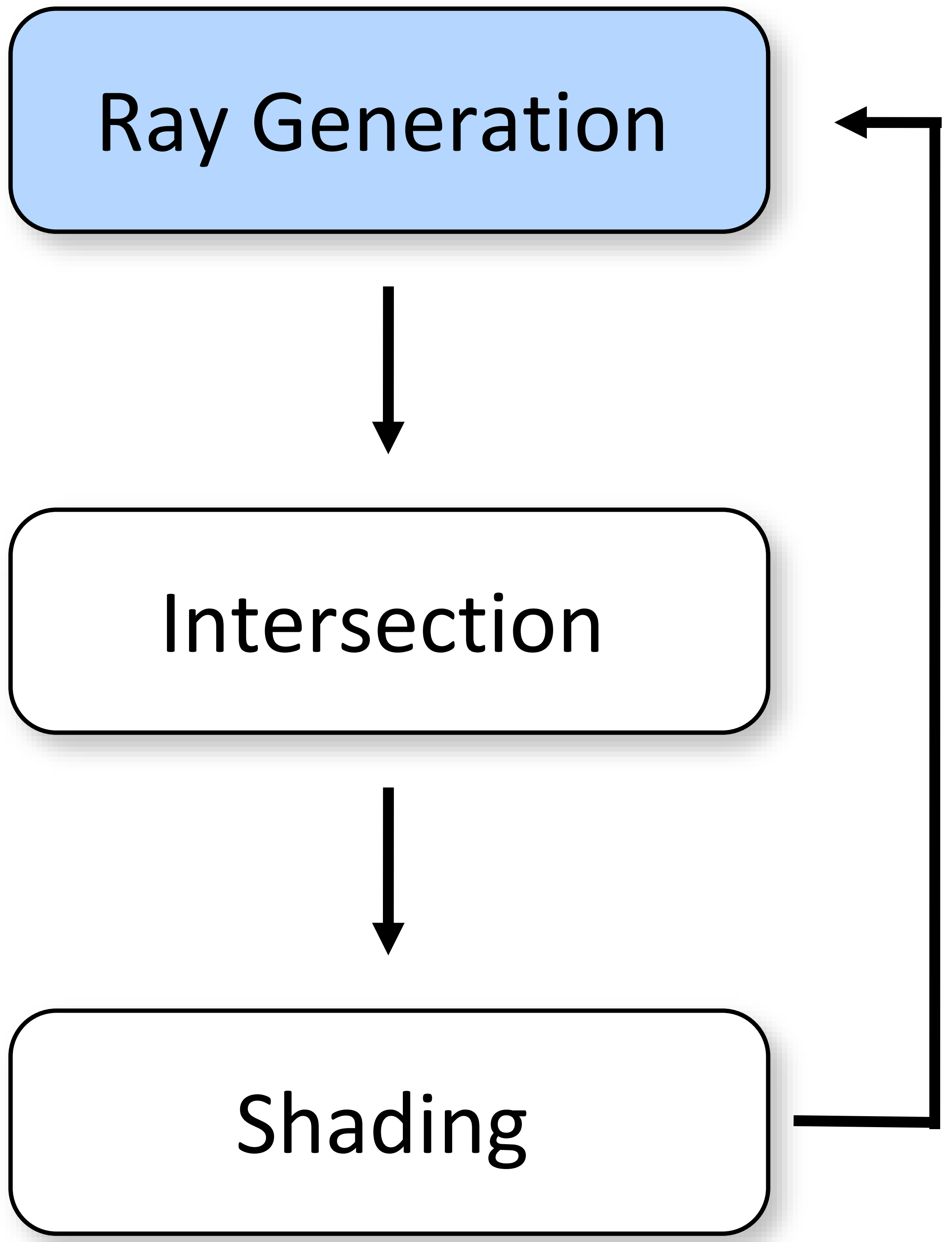
Intersection

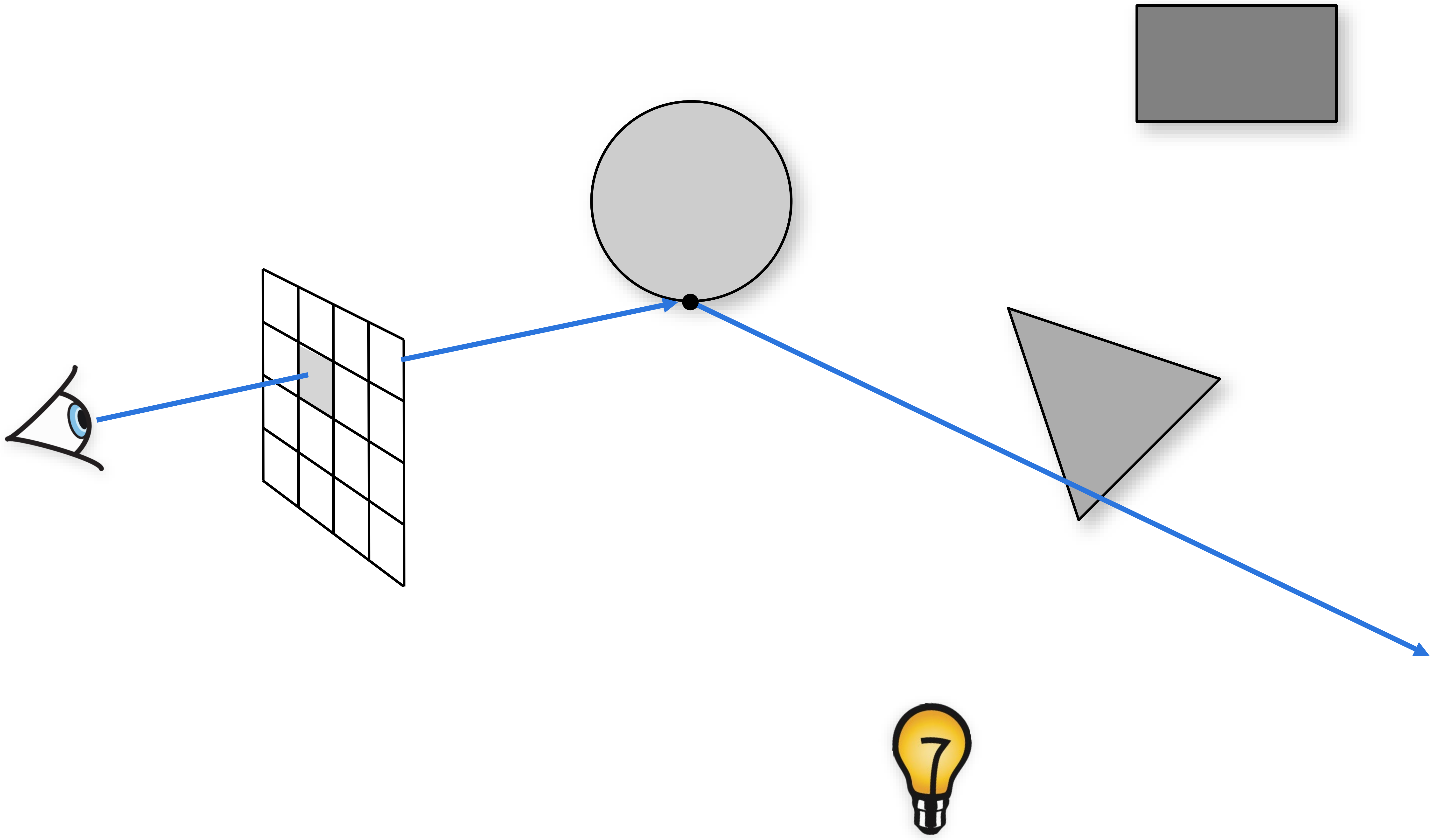
↓

Shading

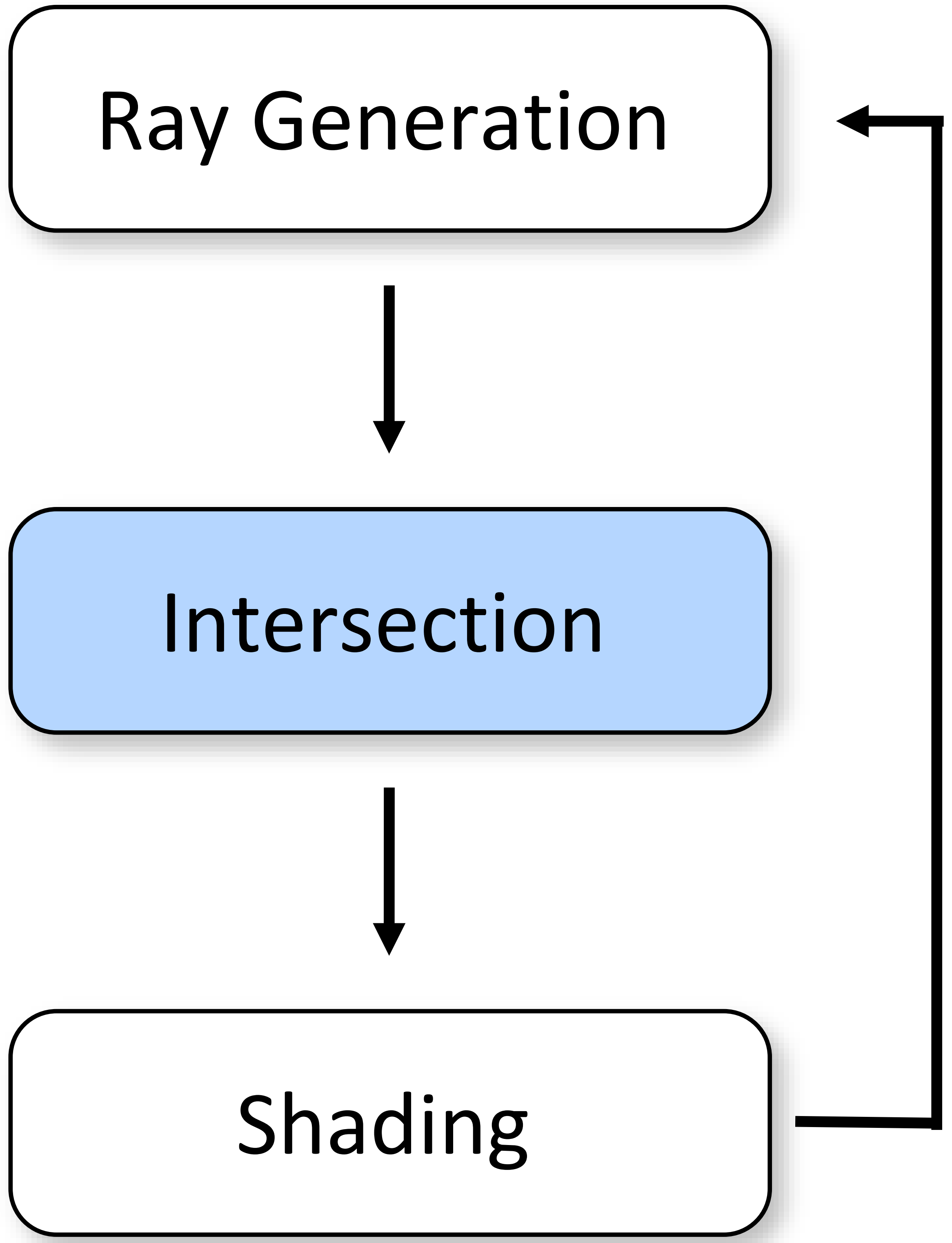


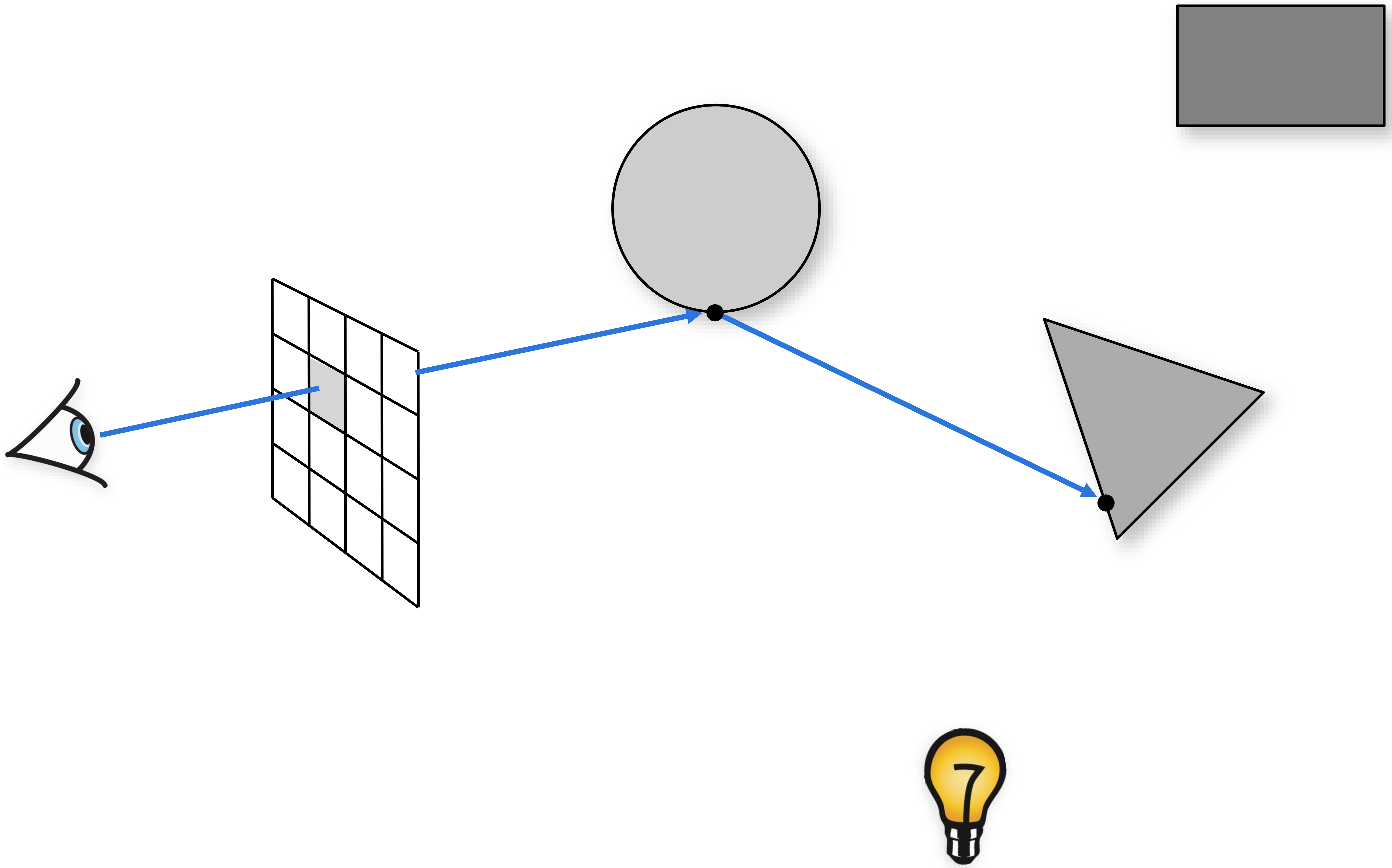
Basic Ray Tracing Pipeline



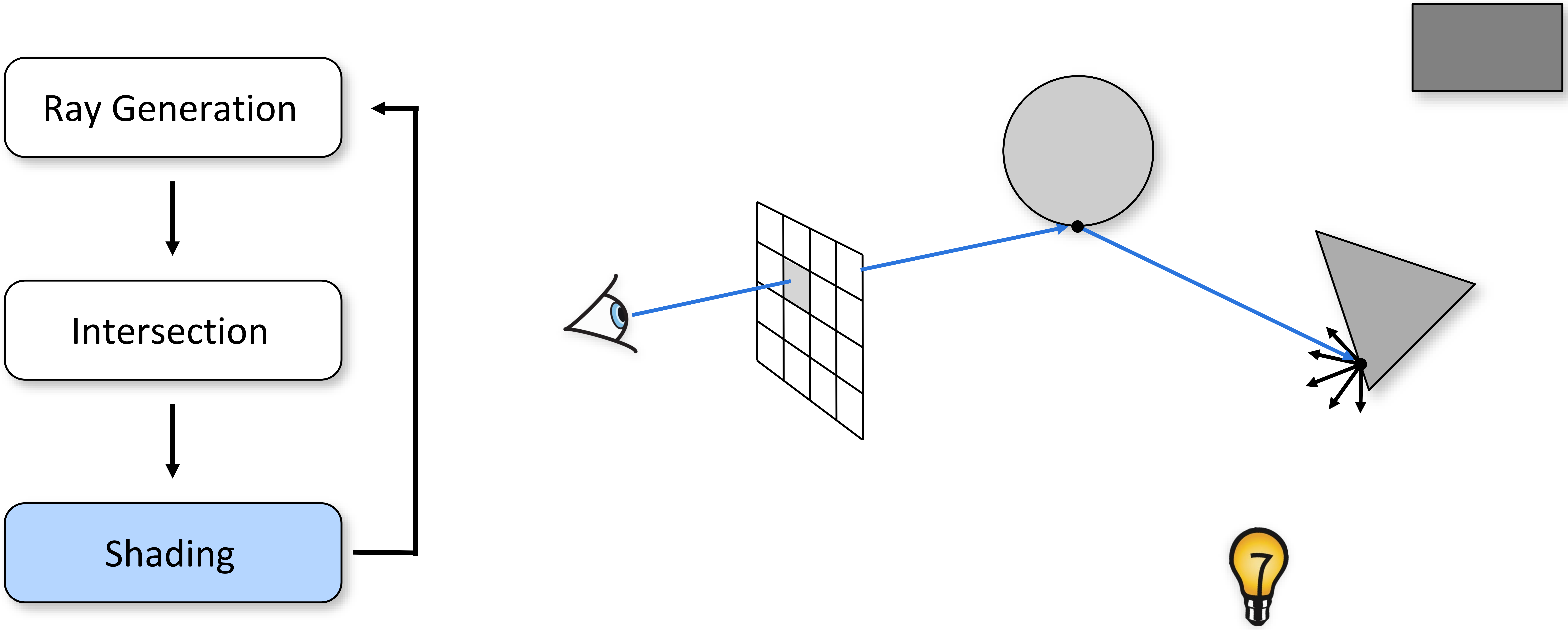


Basic Ray Tracing Pipeline

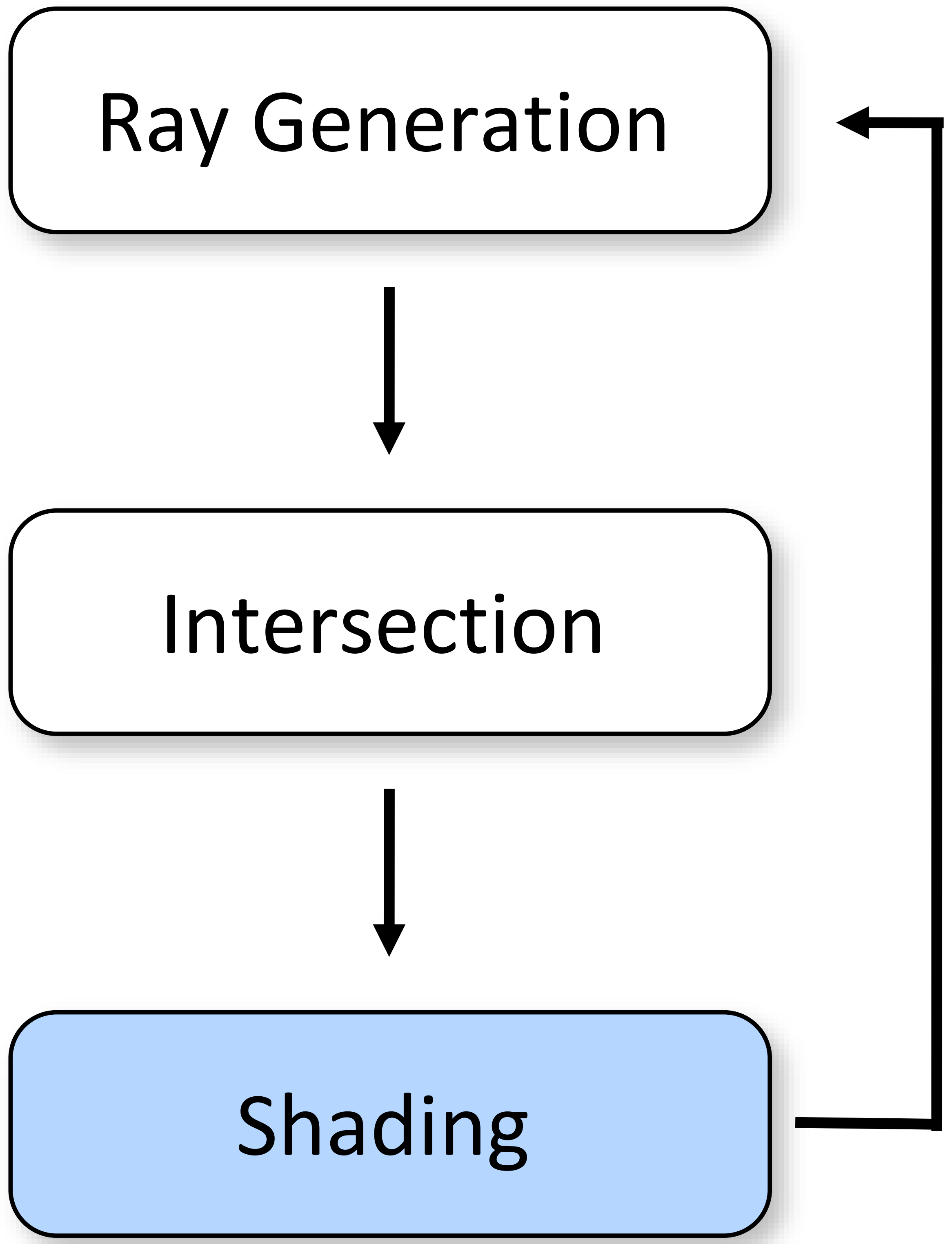


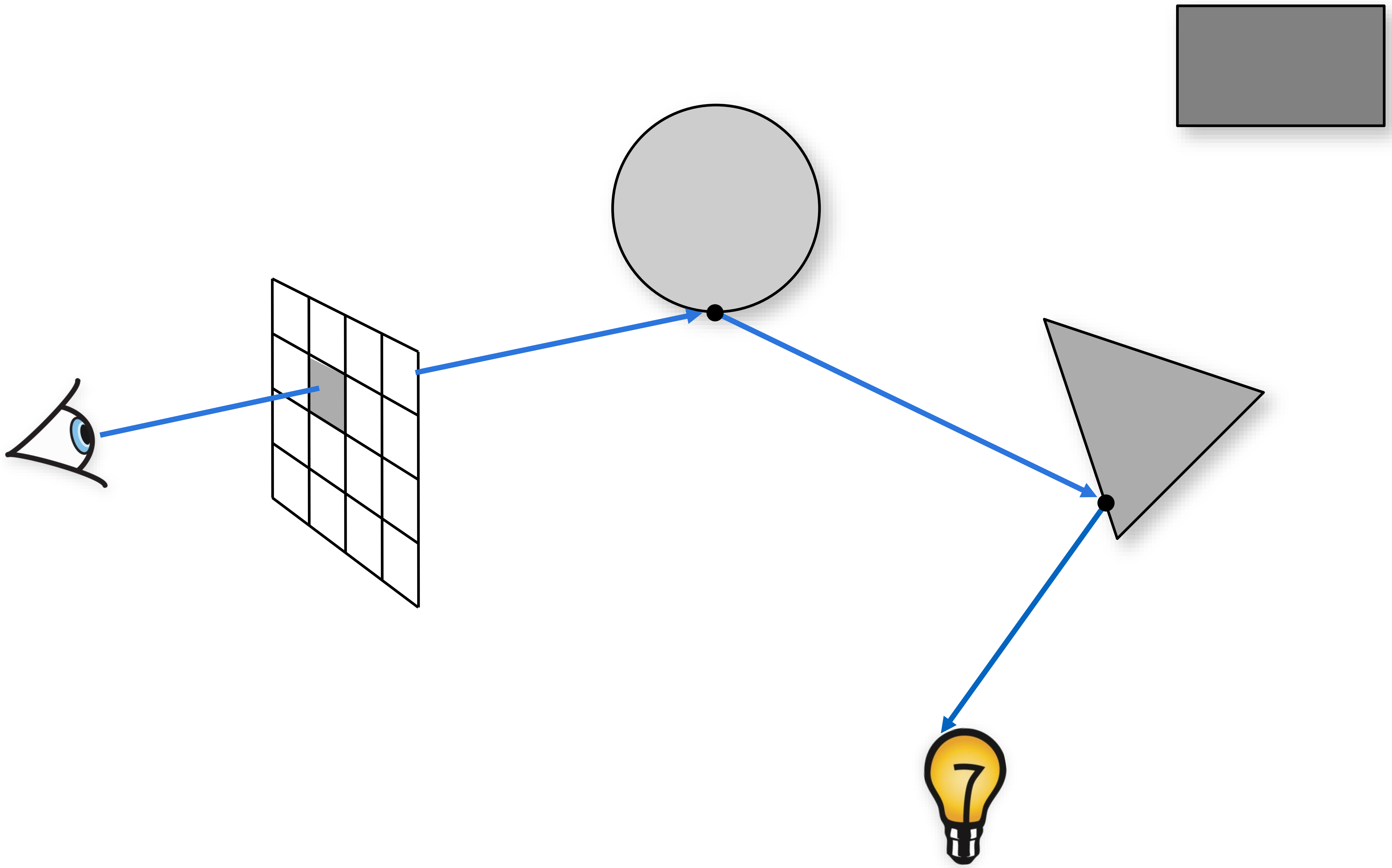


Basic Ray Tracing Pipeline



Basic Ray Tracing Pipeline





Ray Tracing Pseudocode

```
rayTraceImage()
{
   parse scene description

   for each pixel
      ray = generateCameraRay(pixel)
      pixelColor = trace(ray)
}
```

Ray Tracing Pseudocode

```
trace(ray)
{
   hit = find first intersection with scene
       objects

   color = shade(hit)
   return color
}
```

might **trace** more rays (recursive)

Ray Tracing Pseudocode

```
rayTraceImage()
{
   parse scene description

   for each pixel
      ray = generateCameraRay(pixel)
      pixelColor = trace(ray)
}
```

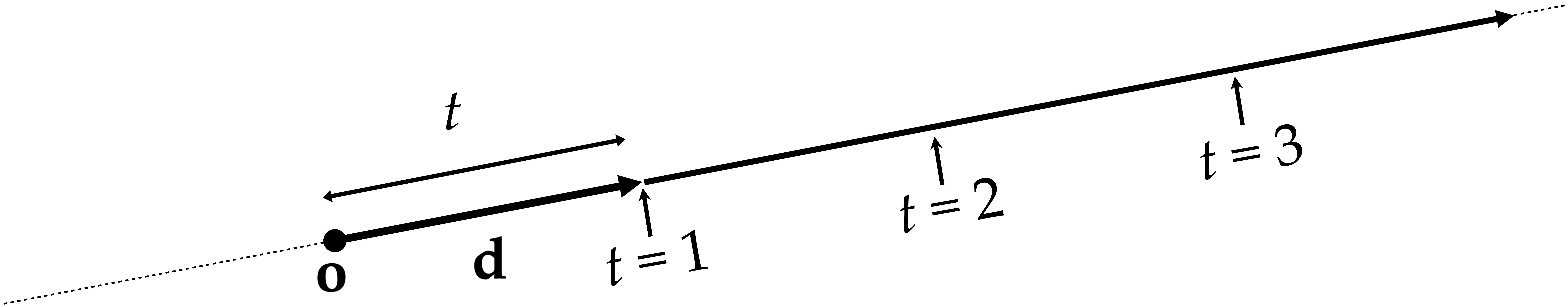
what is a ray? how do we generate a camera ray?

Ray: a half line

Standard representation: origin (point) $\mathbf{o}$ and direction $\mathbf{d}$

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

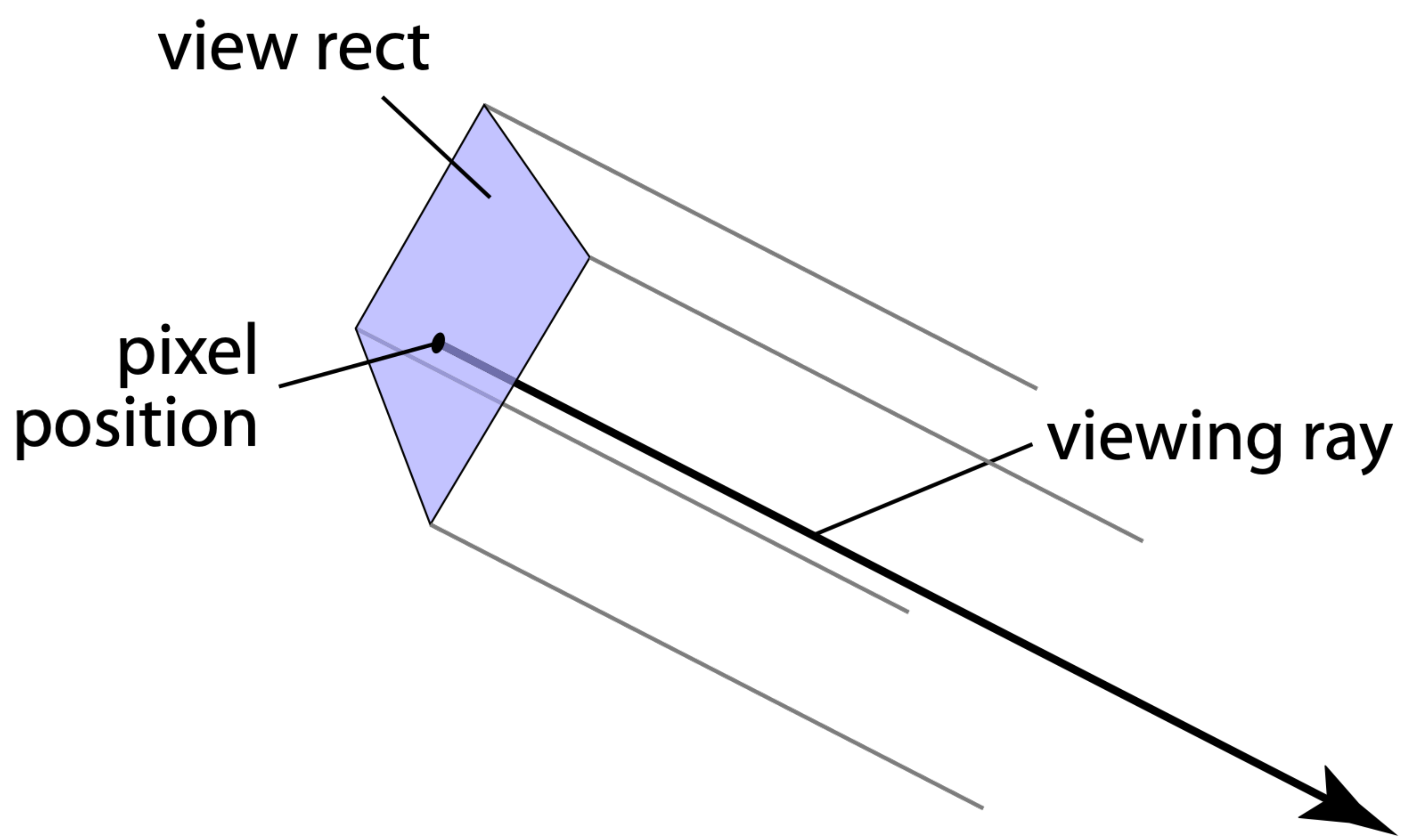
- this is a parametric equation for the line
- lets us directly generate the points on the line
- if we restrict to $t > 0$ then we have a ray
- note replacing $\mathbf{d}$ with $a\mathbf{d}$ does not change ray (for $a > 0$)



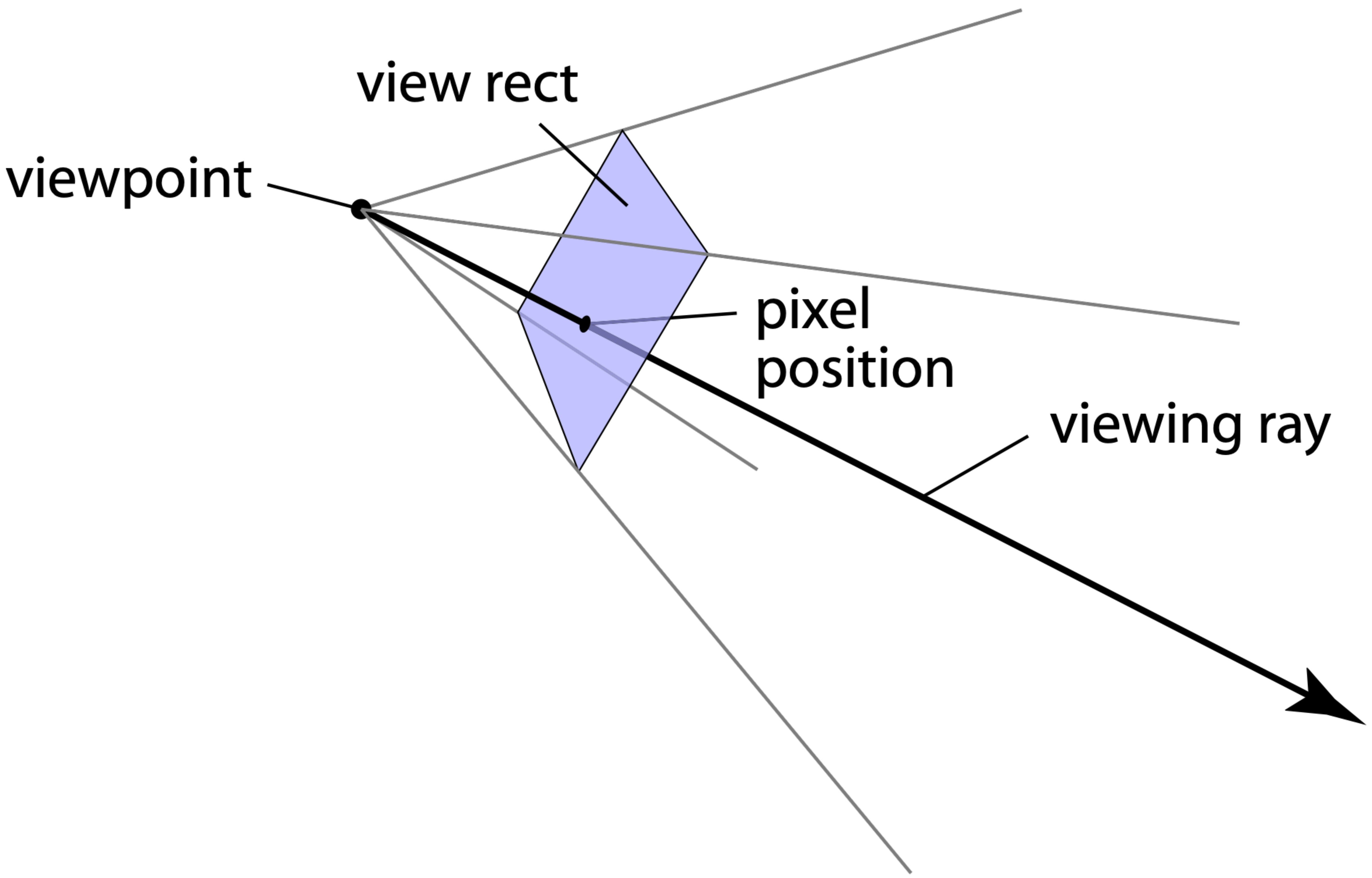
After a slide by Steve Marschner

Generating eye rays

Orthographic



Perspective



After a slide by Steve Marschner

Pinhole Camera (Camera Obscura)

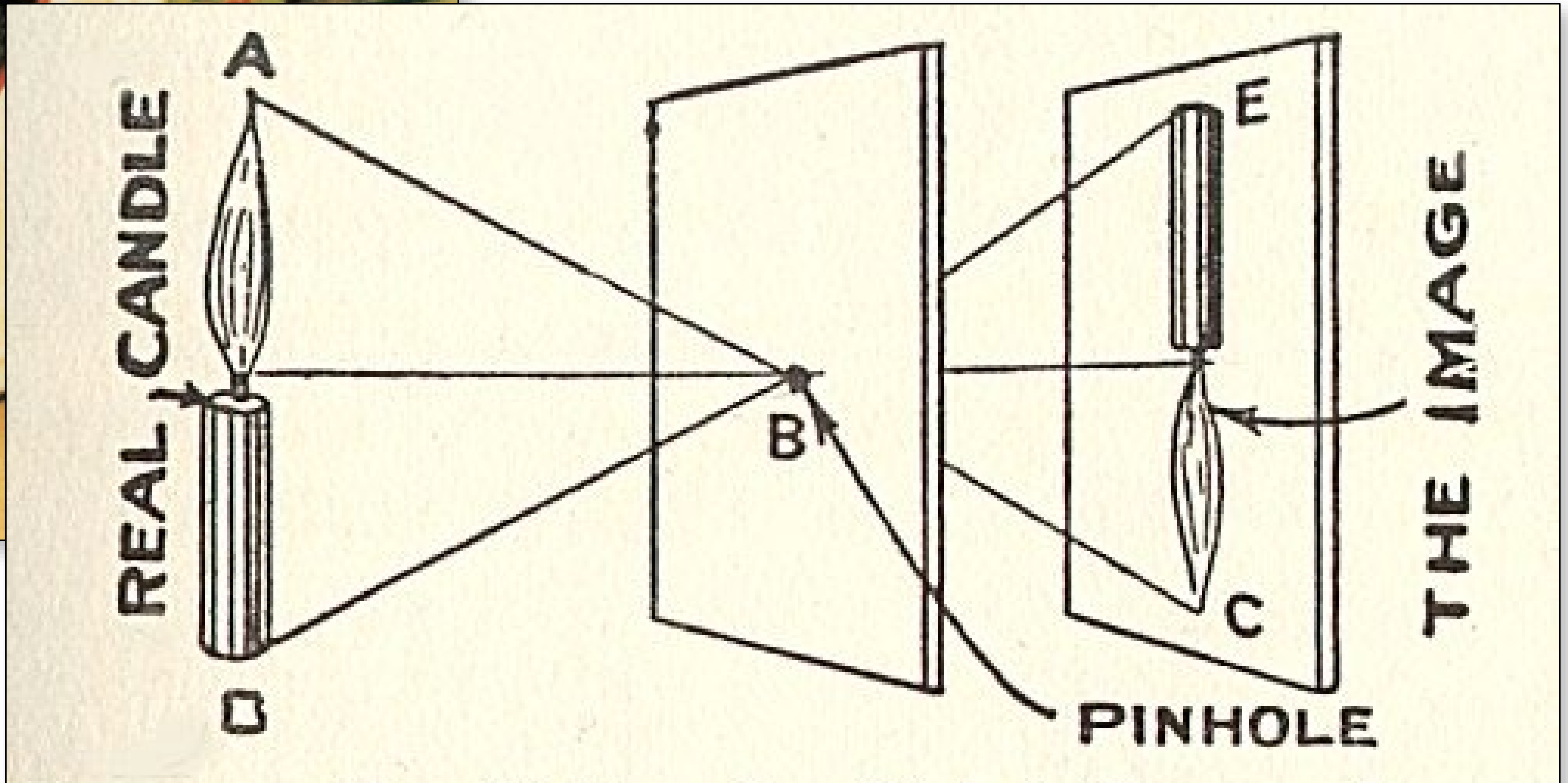
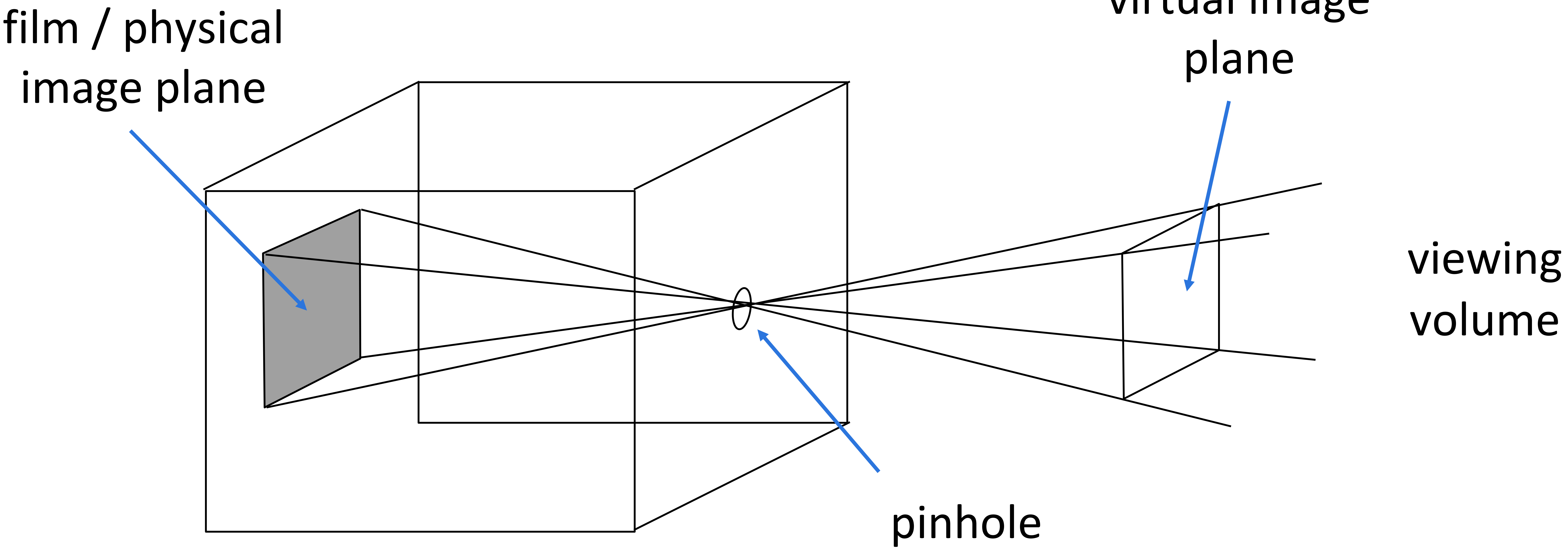


FIG. 131.—How Light and a Pinhole Form an Image.

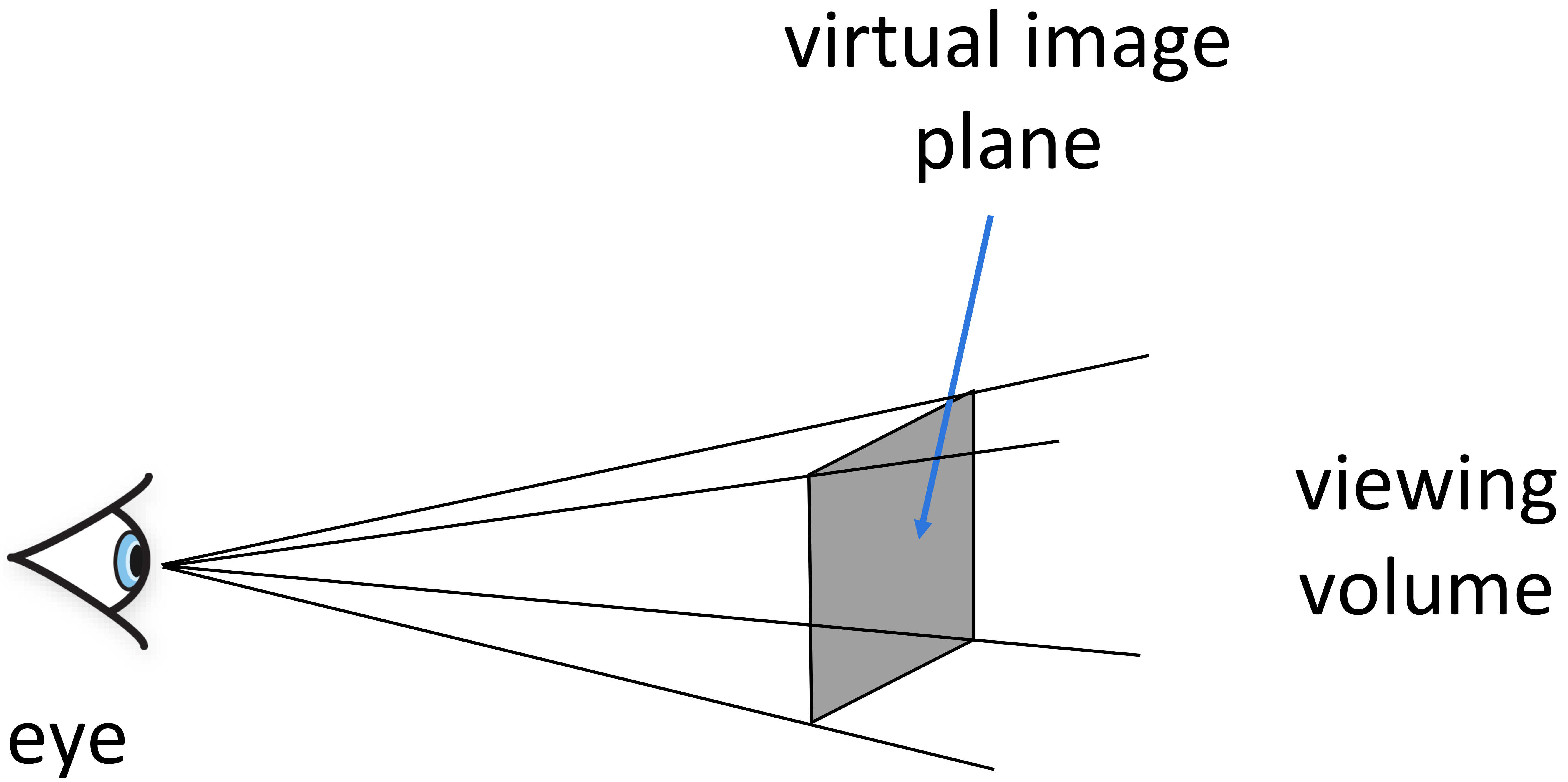
Pinhole Camera

Pinhole Camera



Pinhole Camera

Pinhole Camera



Generating eye rays—perspective

Establish view rectangle in X–Y plane, specified by, e.g.

- l, r, t, b

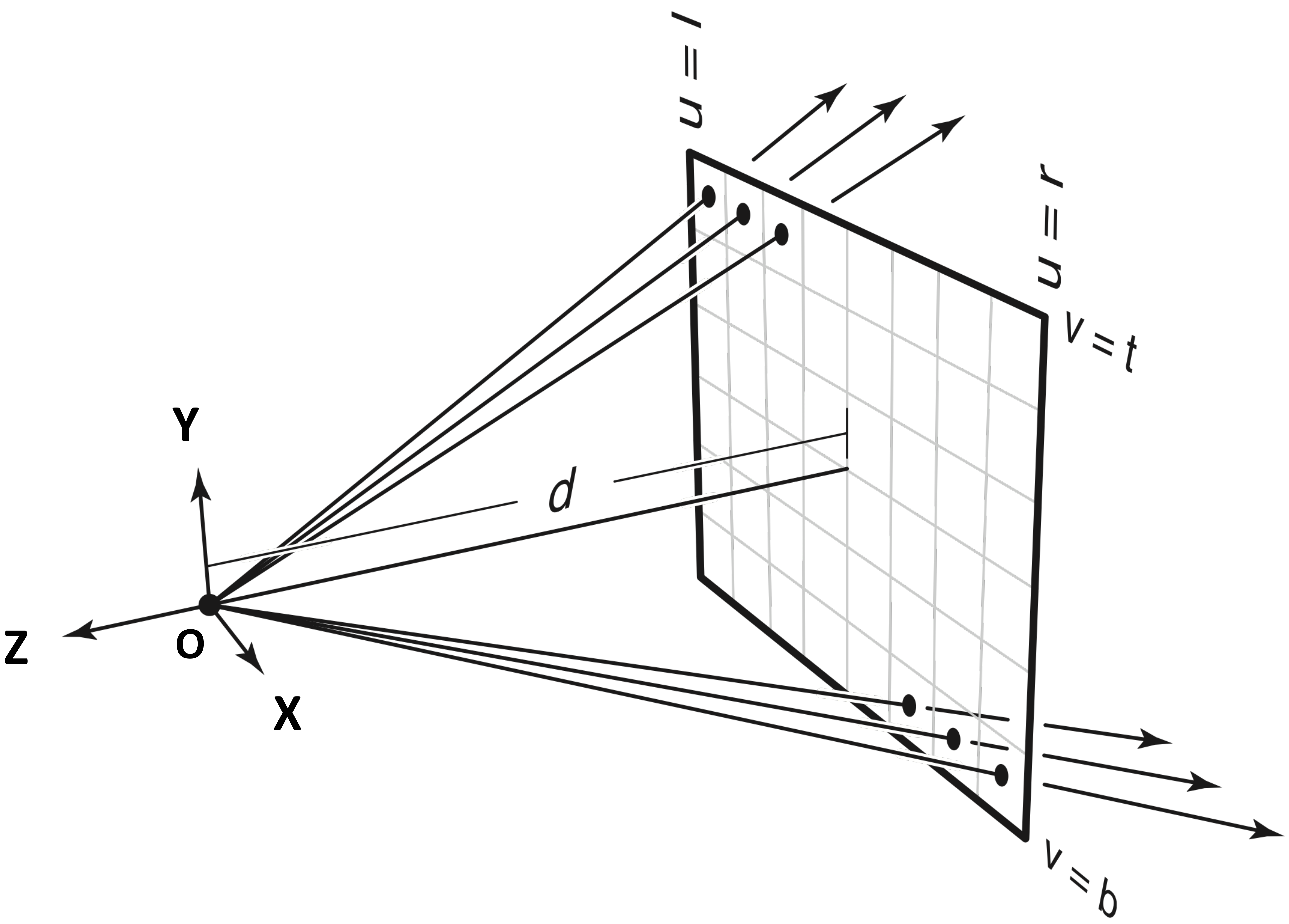
Place rectangle at $z = -d$

$$\mathbf{s} = [u, v, -d]^T$$

$$\mathbf{d} = \mathbf{s}$$

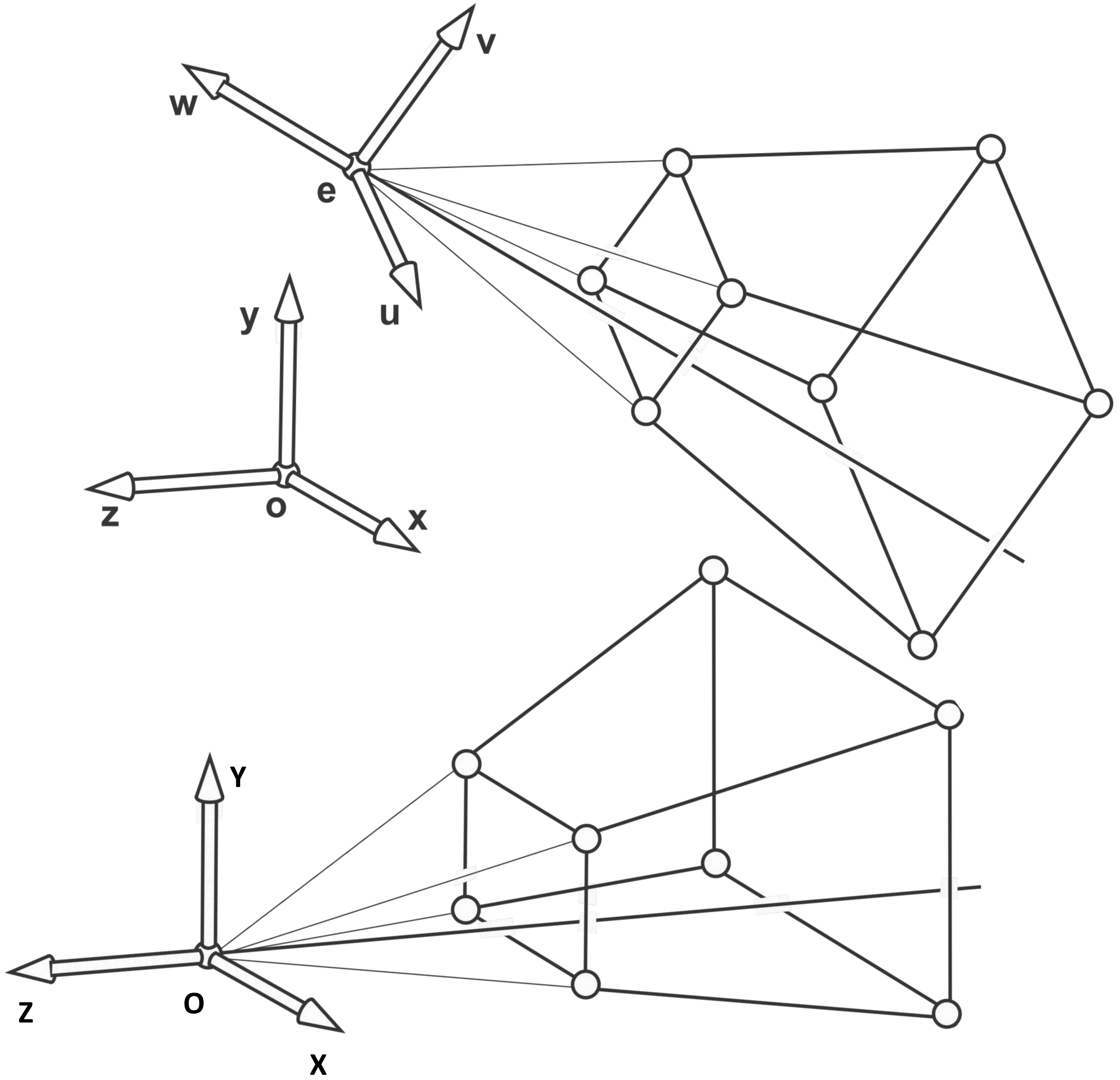
$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

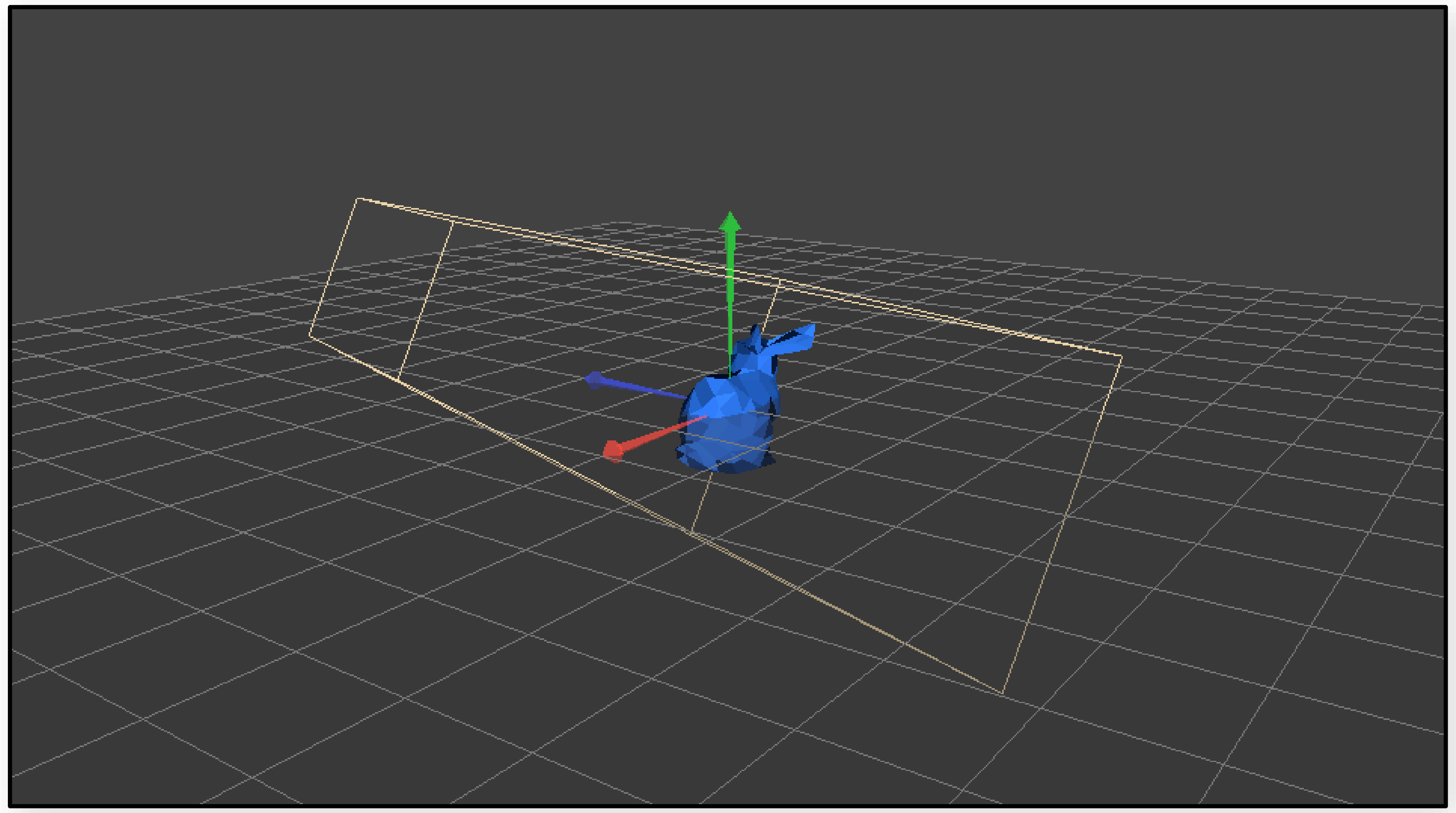
Does distance *d* matter?

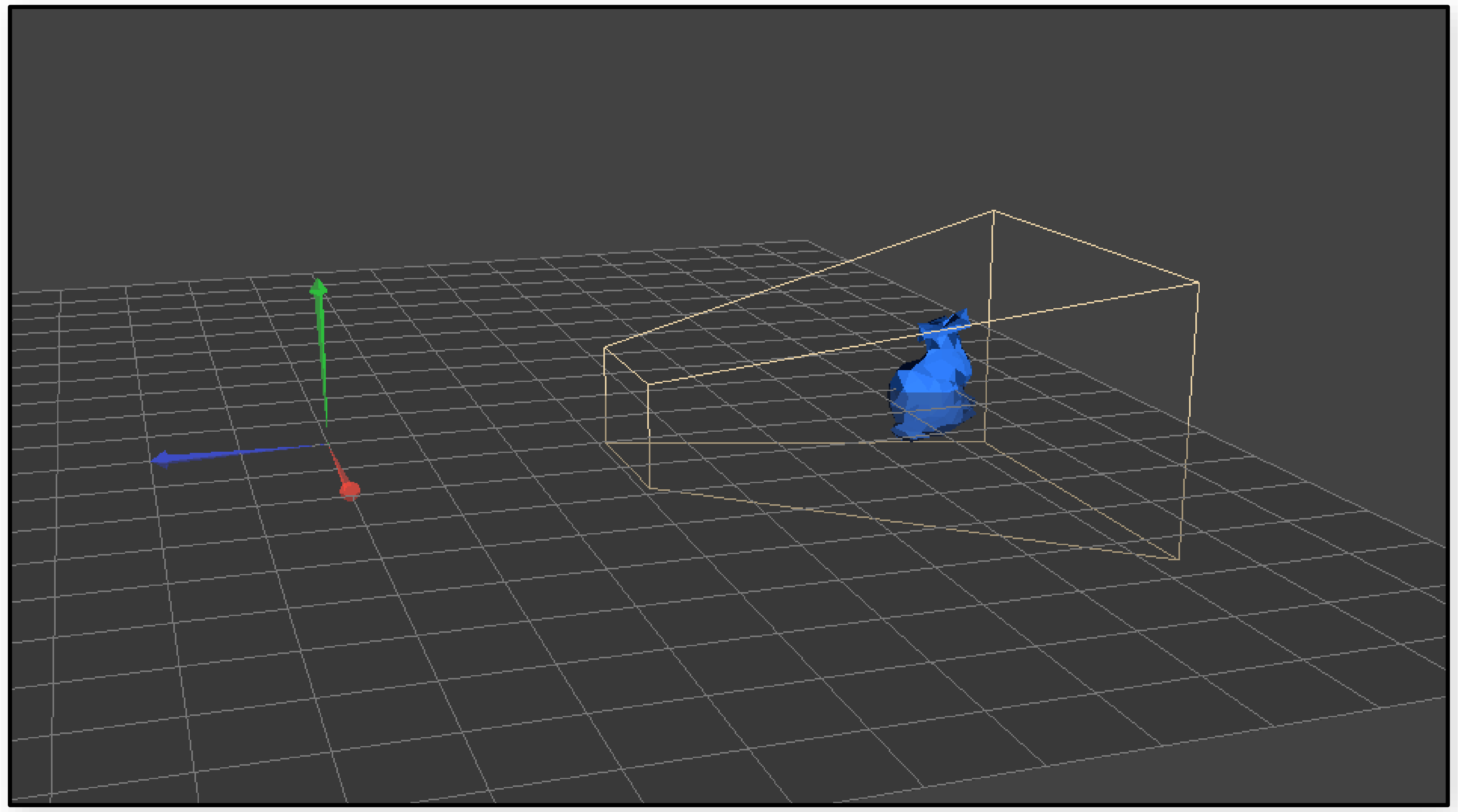


After a slide by Steve Marschner

Placing the camera in the scene



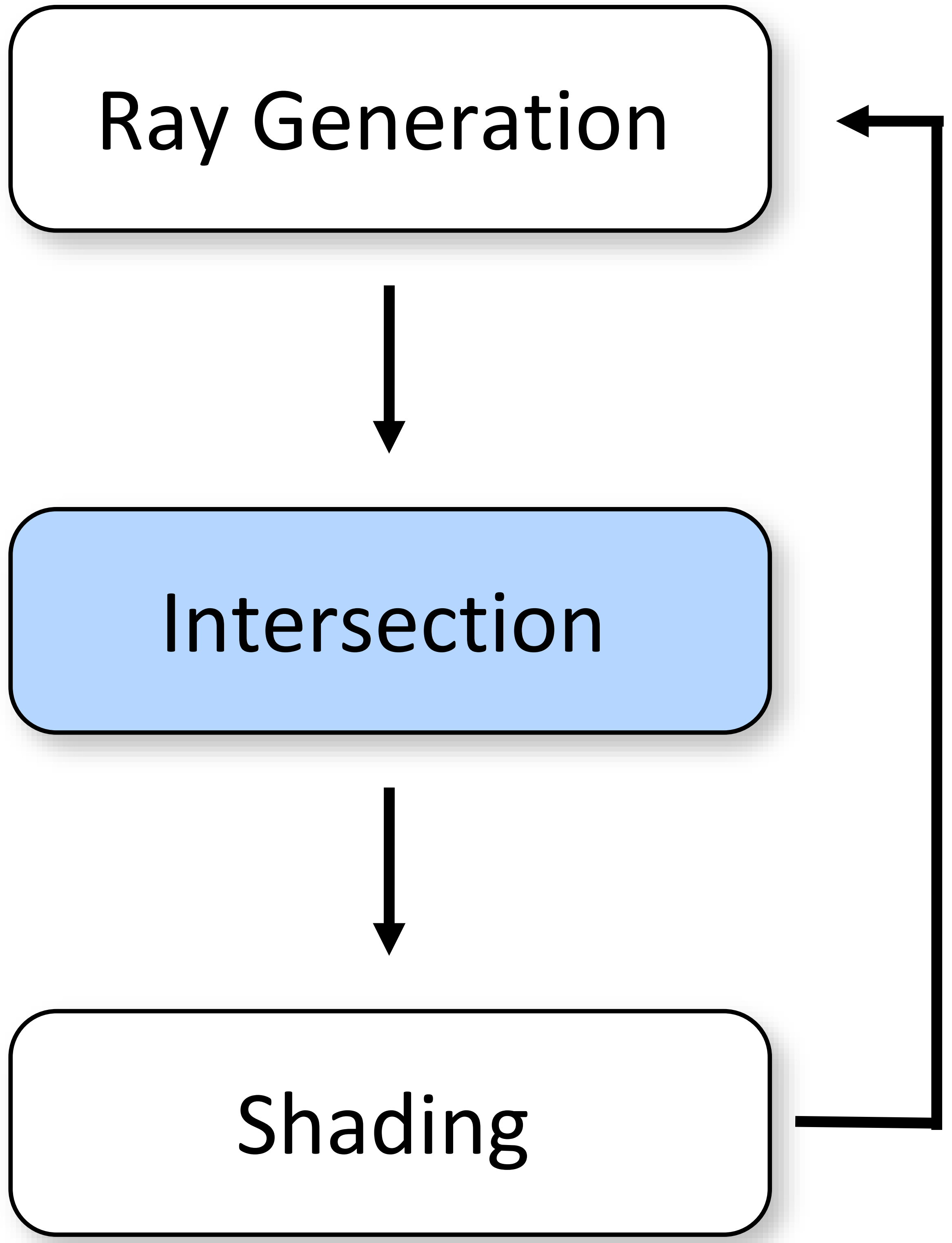


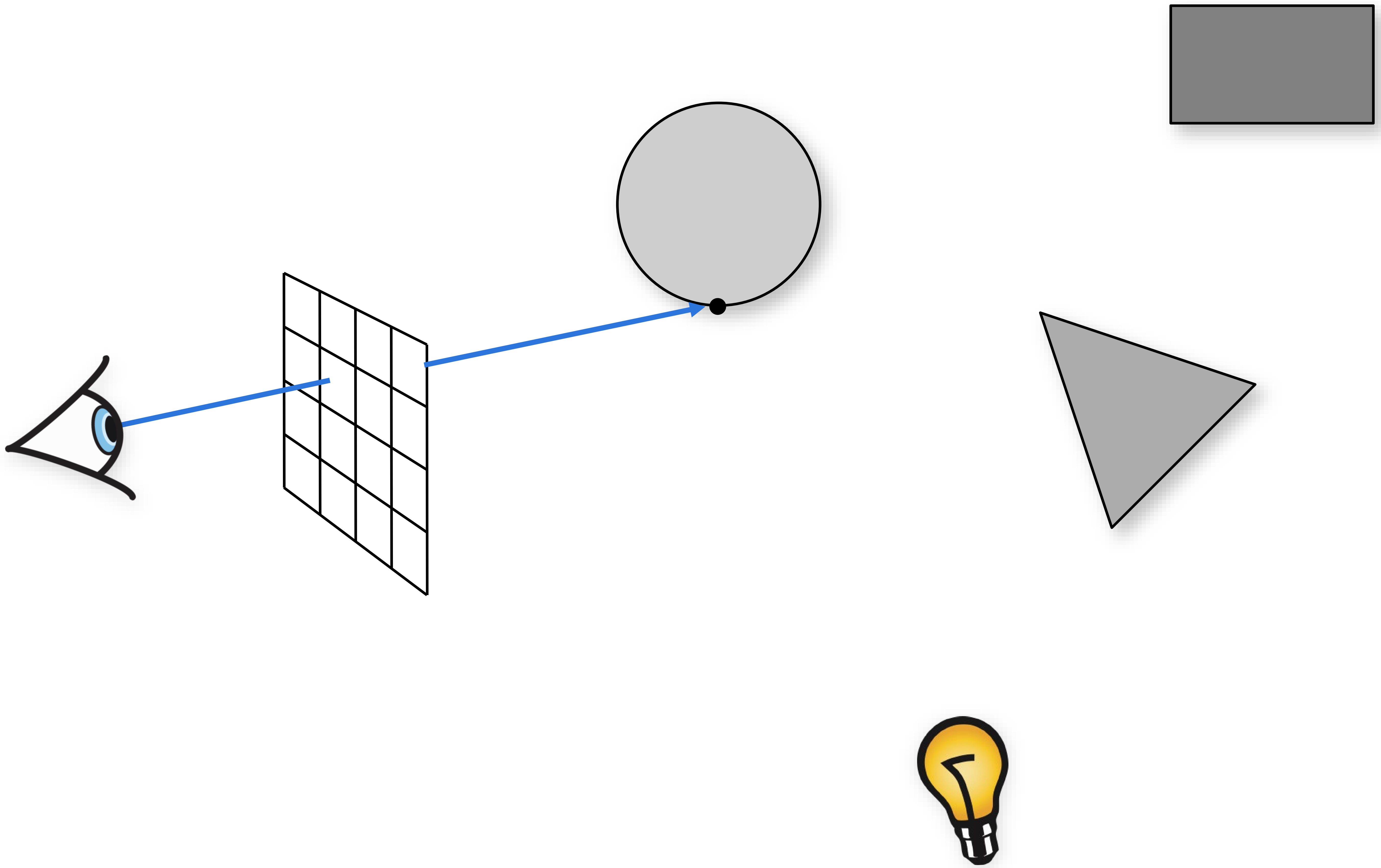


Generating eye rays—orthographic

How do you generate a ray for an orthographic camera?

Ray-Surface Intersections





Ray-Surface Intersections

Surface primitives

- spheres
- planes
- triangles
- general implicits
- etc.

Ray-Sphere Intersection

Algebraic approach:

- Condition 1: point is on ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Condition 2: point is on sphere: $\|\mathbf{x} - \mathbf{c}\|^2 - r^2 = 0$

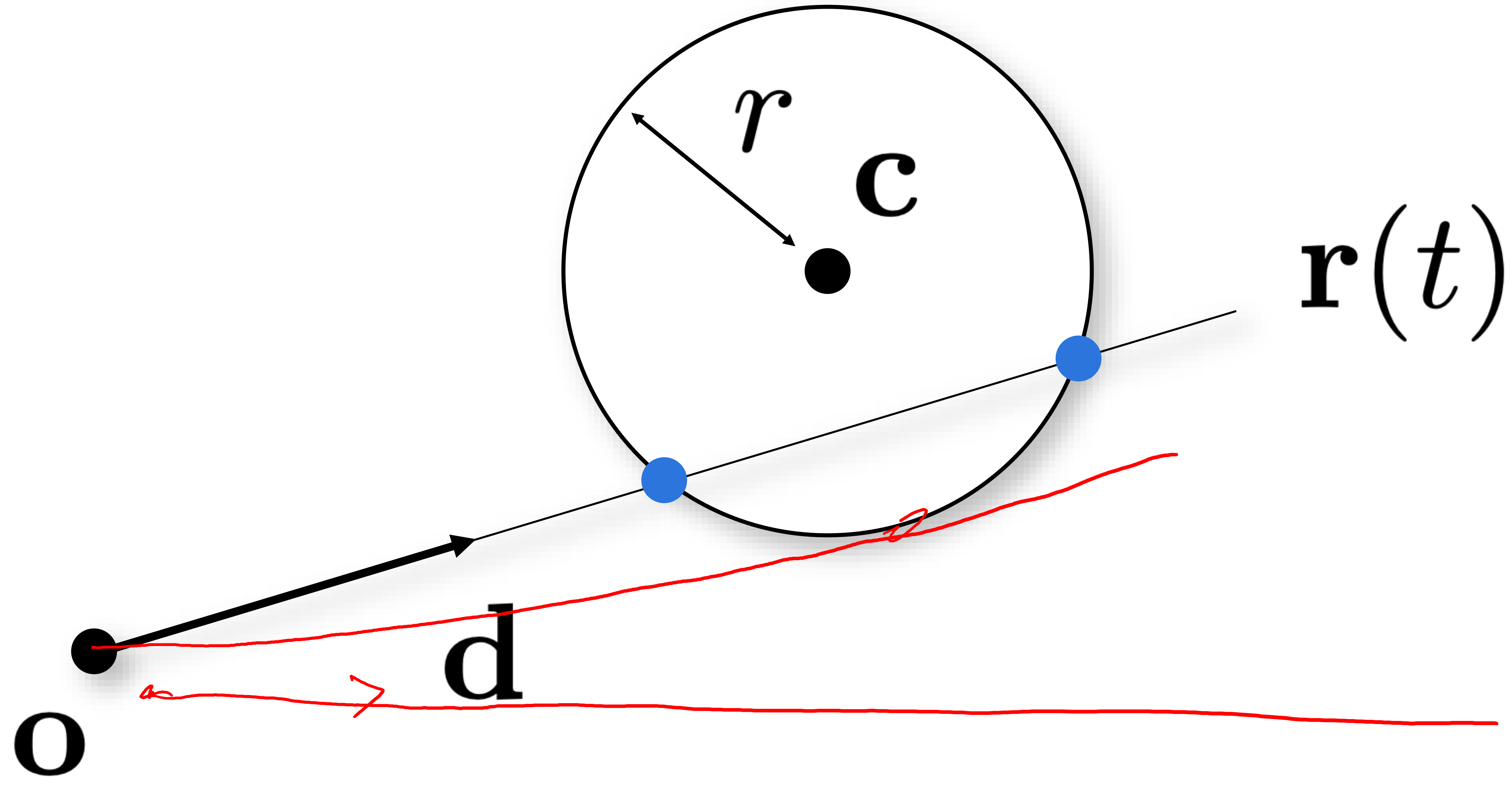
point of interest

center

radius

- substitute and solve for t:

$$\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 - r^2 = 0$$



Ray-Sphere Intersection

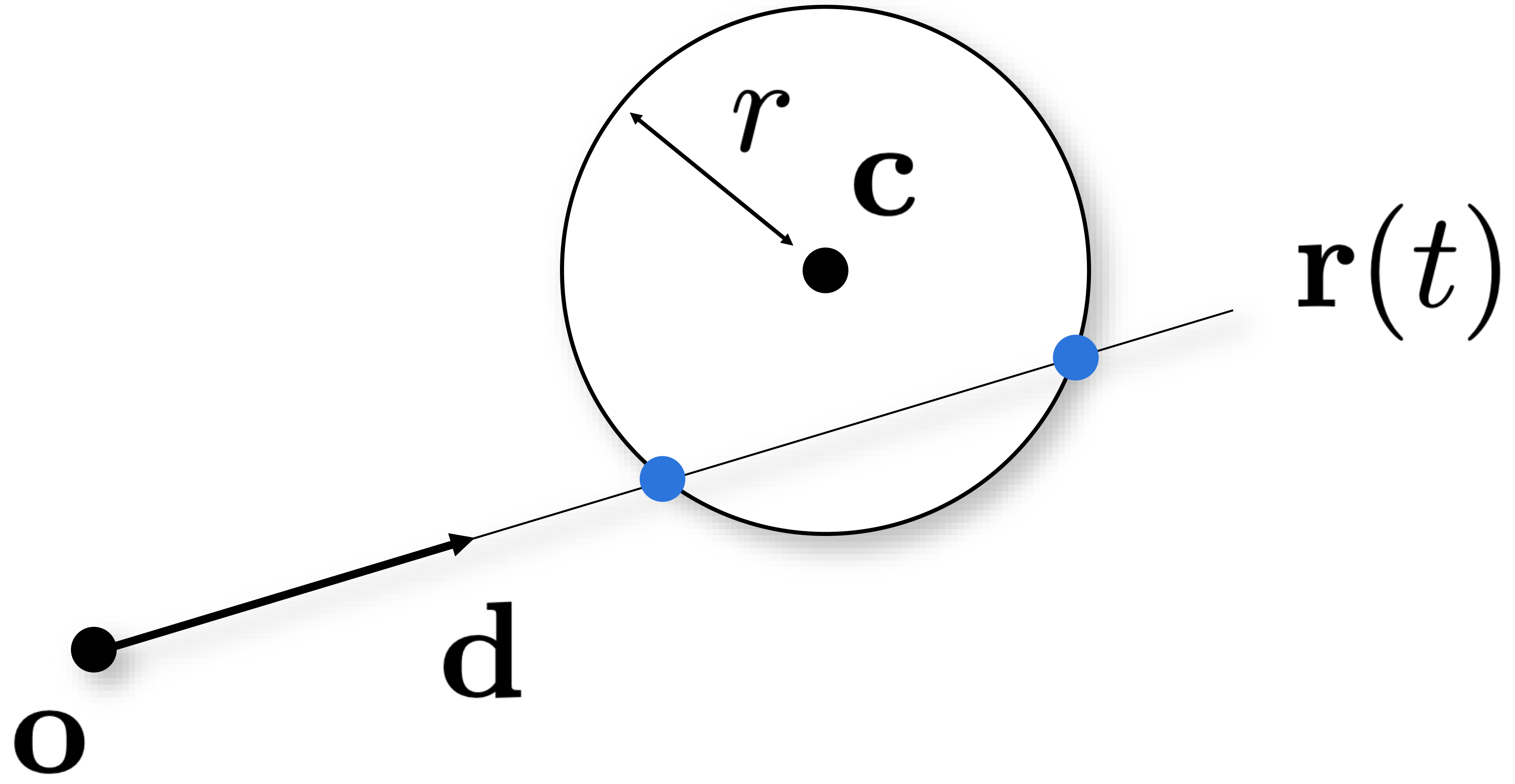
substitute and solve for t

$$\|\mathbf{o} + t\mathbf{d} - \mathbf{c}\|^2 - r^2 = 0 \longrightarrow \begin{aligned}&(\mathbf{o}_x + t\mathbf{d}_x - \mathbf{c}_x)^2 + \\ &(\mathbf{o}_y + t\mathbf{d}_y - \mathbf{c}_y)^2 + \\ &(\mathbf{o}_z + t\mathbf{d}_z - \mathbf{c}_z)^2 - r^2 = 0\end{aligned}$$

which reduces to: $At^2 + Bt + C = 0$

Solve for t using quadratic equation:

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$



What happens when square root is zero or negative?

Ray-Surface Intersections

Surface primitives

- spheres
- planes
- triangles
- general implicits
- etc.

Ray-Plane Intersection

Plane equation (implicit)

Algebraic form:

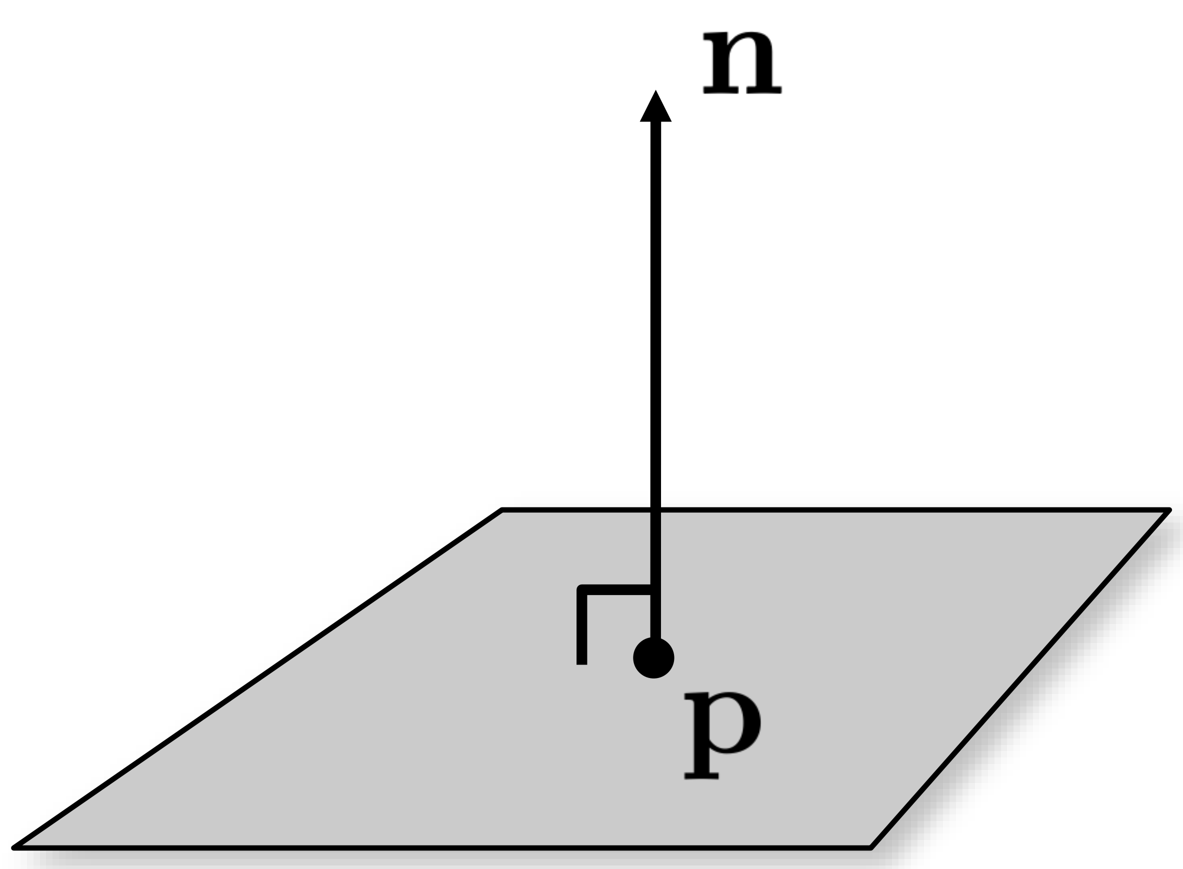
$$ax + by + cz + d = 0$$

Ray-Plane Intersection

Plane equation (implicit)

$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$$

point of interest | point on plane | plane normal

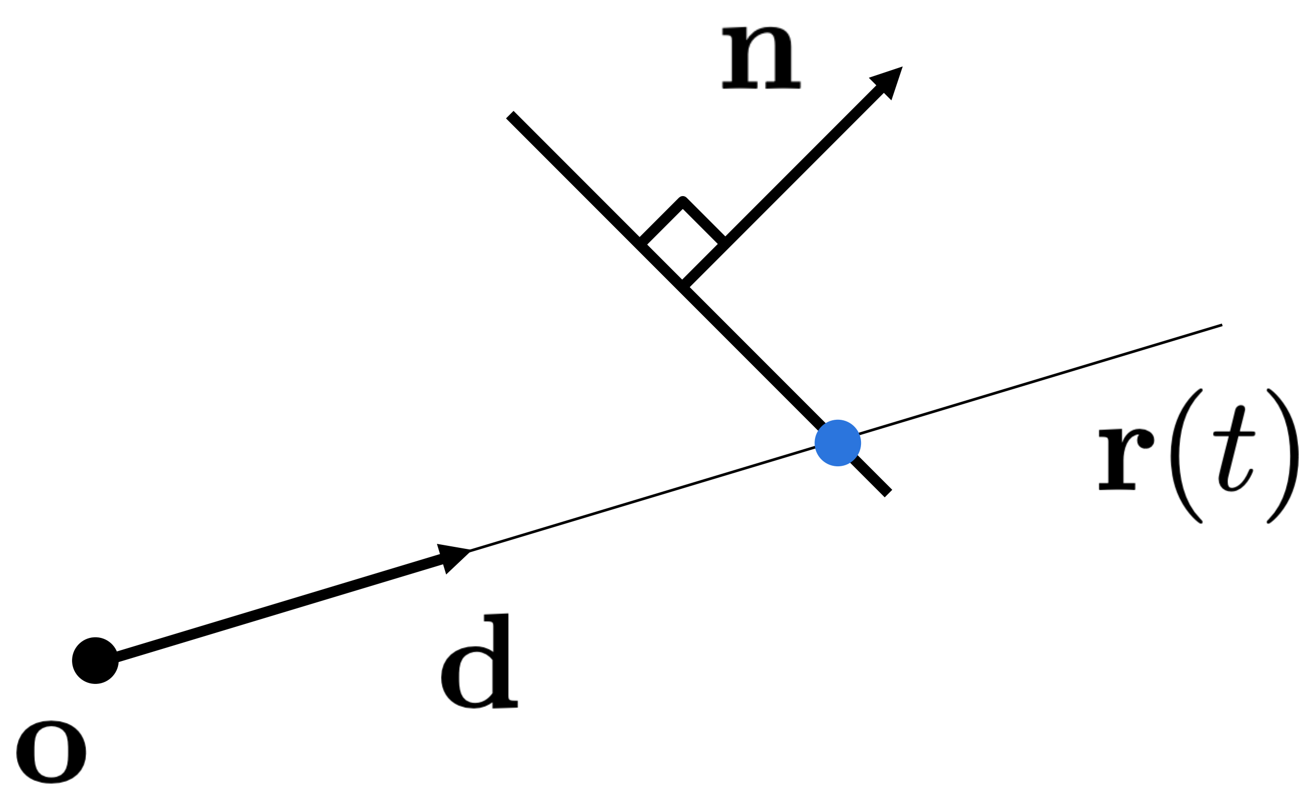


substitute ray equation for $\mathbf{x}$ and solve for t

$$(\mathbf{o} + t\mathbf{d} - \mathbf{p}) \cdot \mathbf{n} = 0$$

$$t\mathbf{d} \cdot \mathbf{n} + (\mathbf{o} - \mathbf{p}) \cdot \mathbf{n} = 0$$

$$t = -\frac{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$



Ray-Surface Intersections

Surface primitives

- spheres
- planes
- triangles
- general implicits
- etc.

Ray-Triangle intersection

Condition 1: point is on ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$

Condition 2: point is on plane: $(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$

Condition 3: point is on the inside of all three edges

First solve 1&2 (ray–plane intersection) for t:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{p}) \cdot \mathbf{n} = 0$$

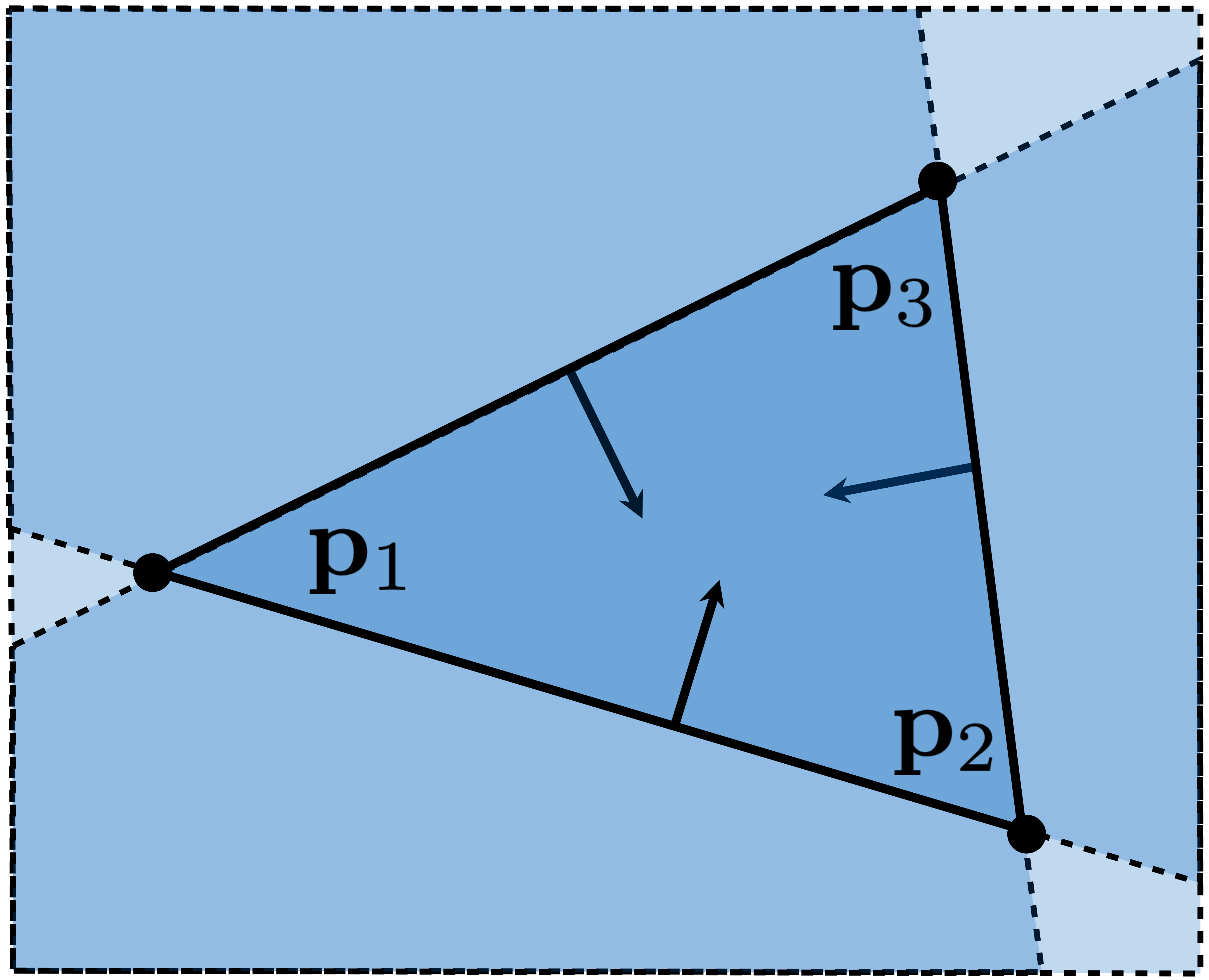
$$t = -\frac{(\mathbf{o} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Several options for 3

After a slide by Steve Marschner

Ray-Triangle intersection (Approach 1)

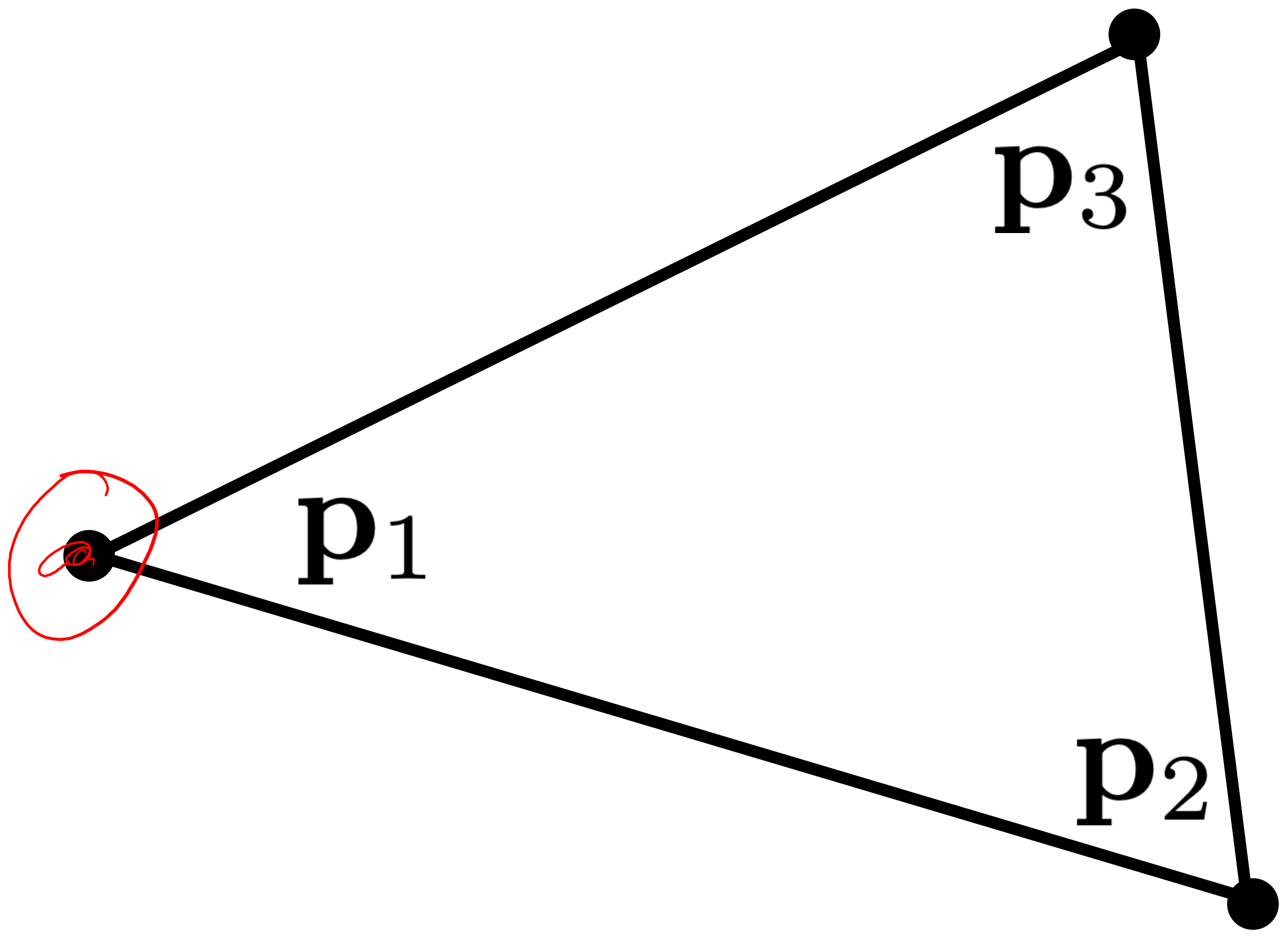
In plane, triangle is the intersection of 3 half spaces



Ray-Triangle intersection (Approach 1)

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

Which way does $\mathbf{n}$ point?



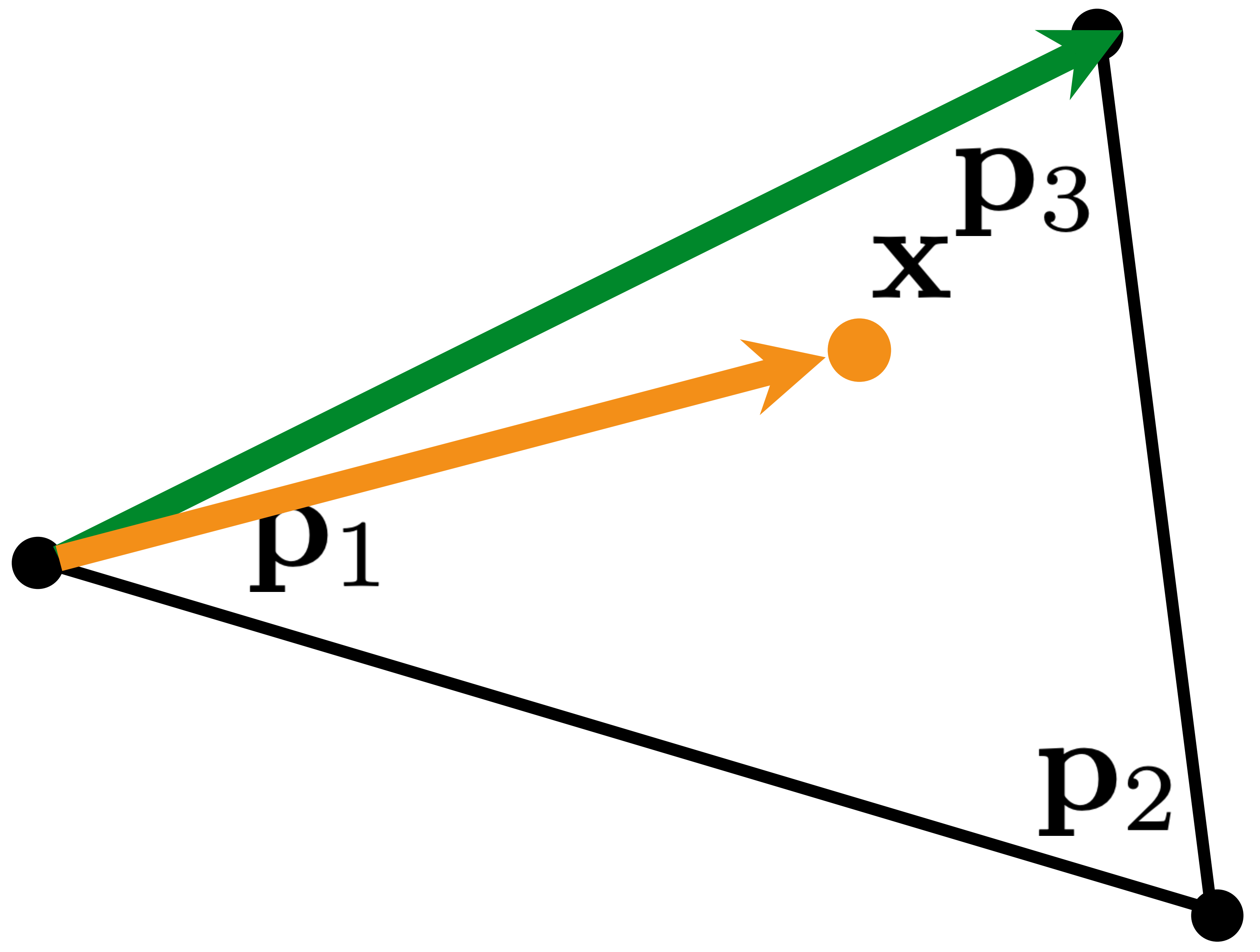
Ray-Triangle intersection (Approach 1)

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

$$\mathbf{n}_{\mathbf{x}13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

Which way does $\mathbf{n}$ point?

What about $\mathbf{n}_{\mathbf{x}13}$?



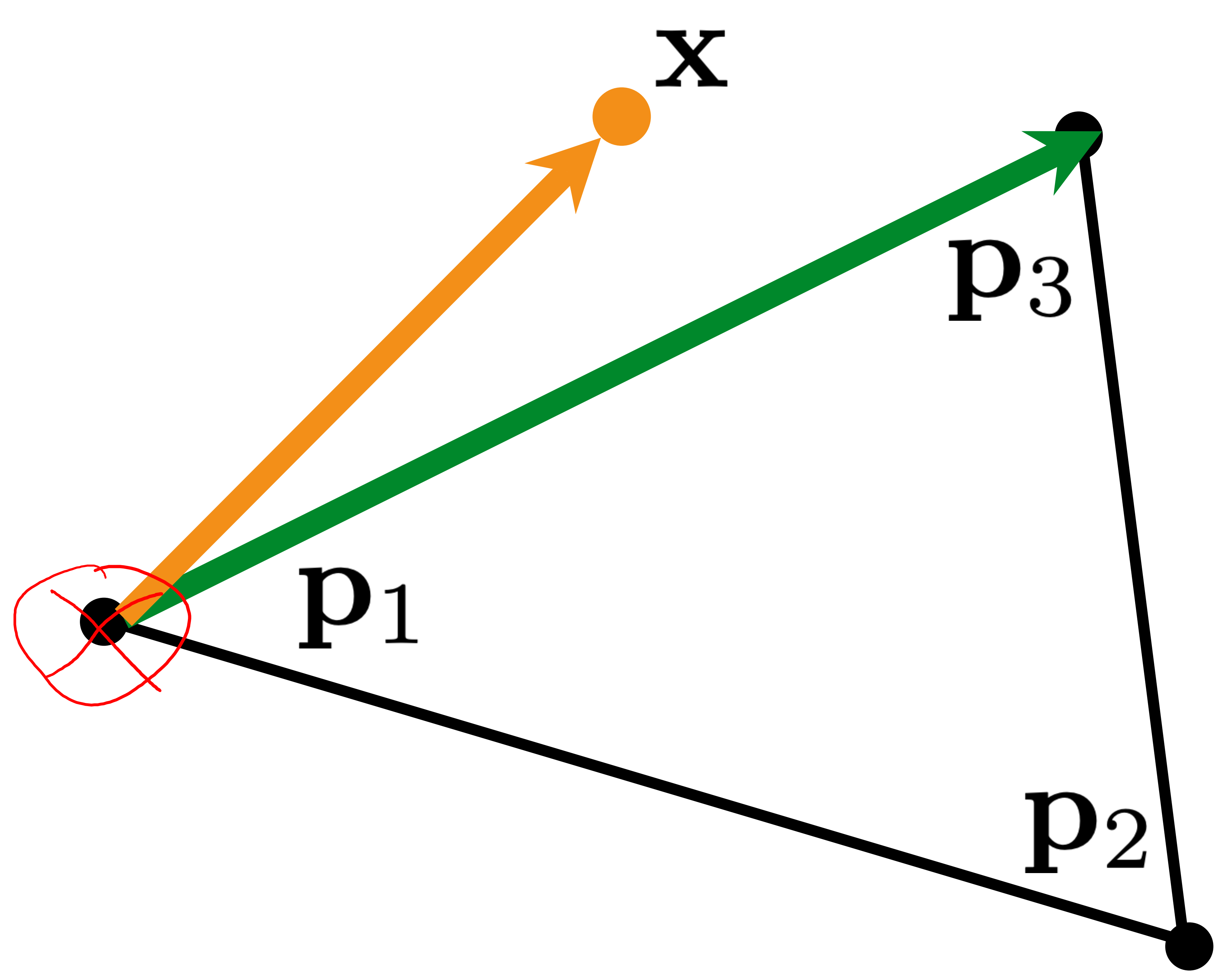
Ray-Triangle intersection (Approach 1)

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$
$$\mathbf{n}_{\mathbf{x}13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

Which way does $\mathbf{n}$ point?

What about $\mathbf{n}_{\mathbf{x}13}$?

- How about now?



Ray-Triangle intersection (Approach 1)

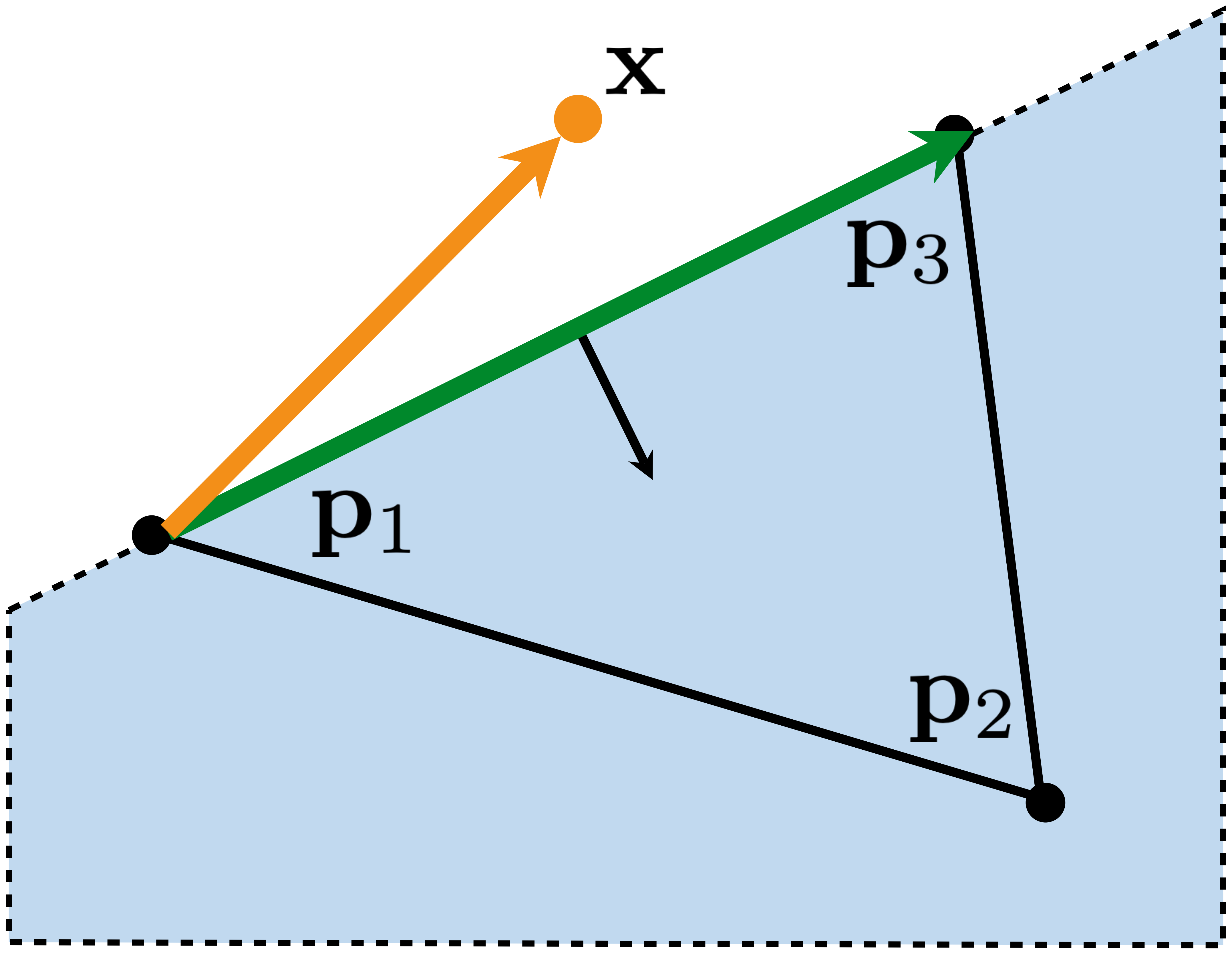
$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

$$\mathbf{n}_{\mathbf{x}13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

Which way does $\mathbf{n}$ point?

What about $\mathbf{n}_{\mathbf{x}13}$?

- How about now?
- Edge test: $(\mathbf{n}_{\mathbf{x}13} \cdot \mathbf{n}) < 0$



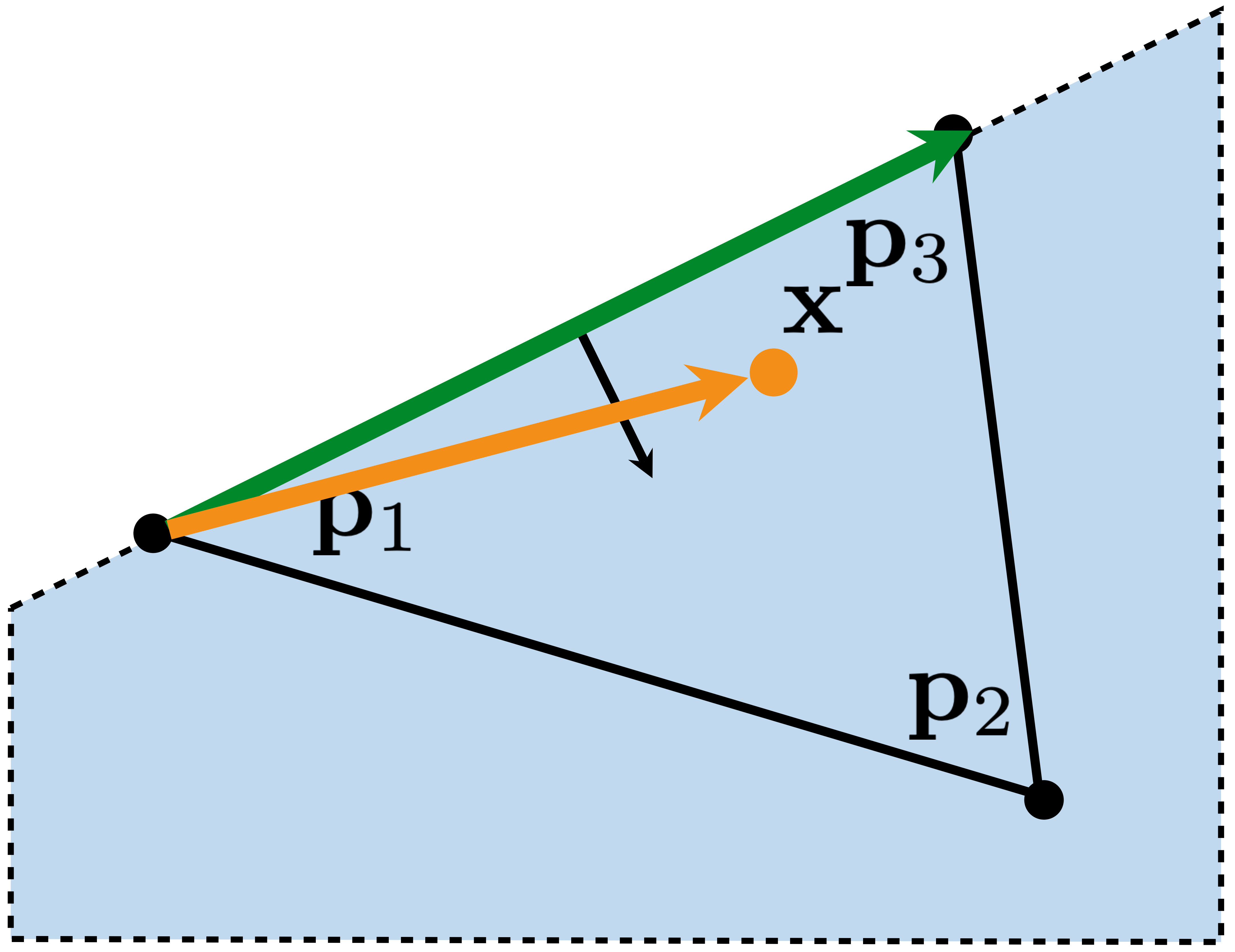
Ray-Triangle intersection (Approach 1)

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$
$$\mathbf{n}_{\mathbf{x}13} = (\mathbf{x} - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

Which way does $\mathbf{n}$ point?

What about $\mathbf{n}_{\mathbf{x}13}$?

- How about now?
- Edge test: $(\mathbf{n}_{\mathbf{x}13} \cdot \mathbf{n}) < 0$



Ray-Triangle Intersection (Approach 2)

Intersect ray with triangle's plane

Test whether hit-point is within triangle

- compute sub-triangle areas α, β, γ
- test inside triangle conditions

Barycentric coordinates

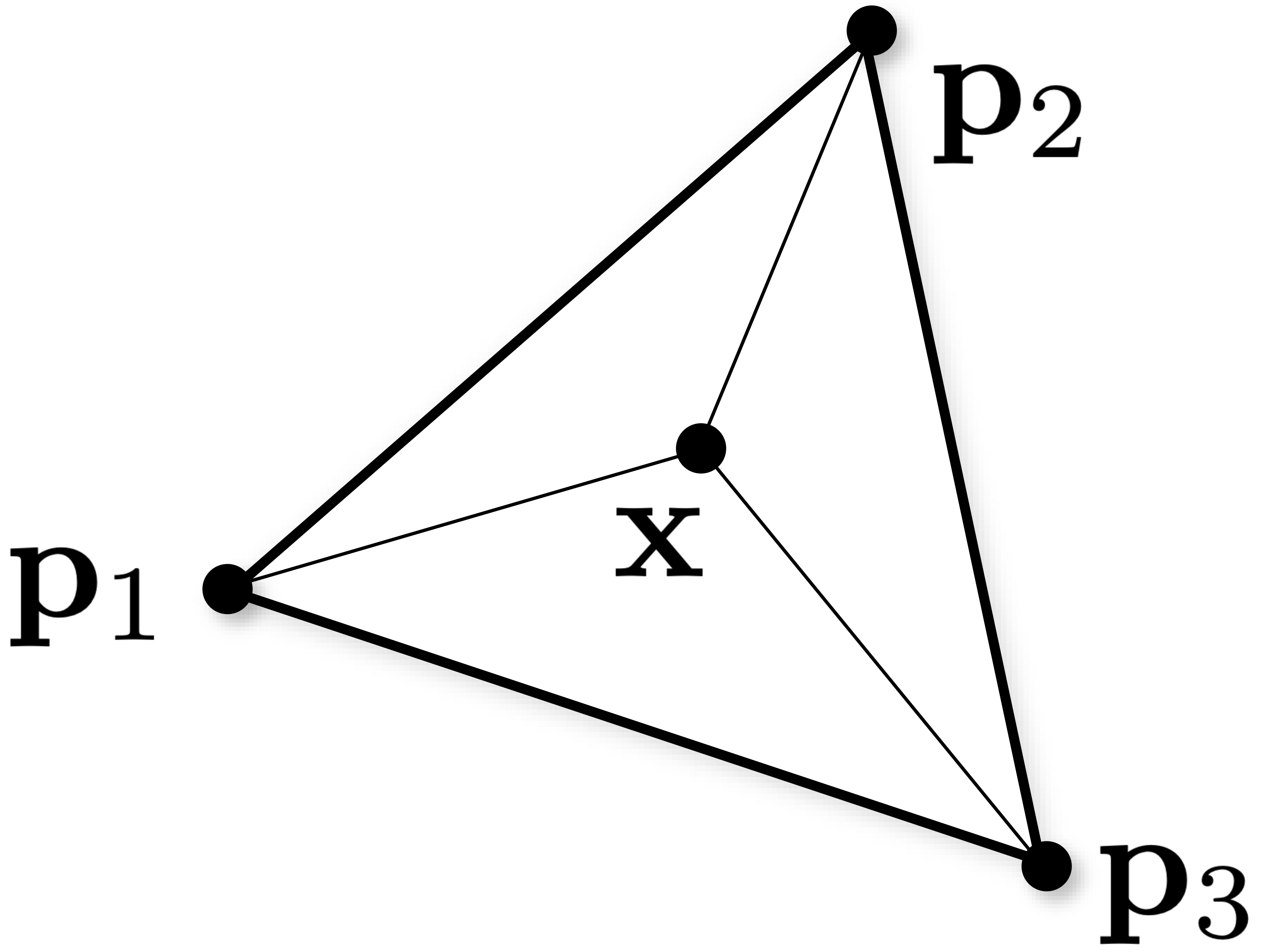
Barycentric coordinates: $\mathbf{x}(\alpha, \beta, \gamma) = \alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + \gamma \mathbf{p}_3$

Inside triangle conditions:

$$\alpha + \beta + \gamma = 1 \quad 0 \leq \alpha \leq 1$$

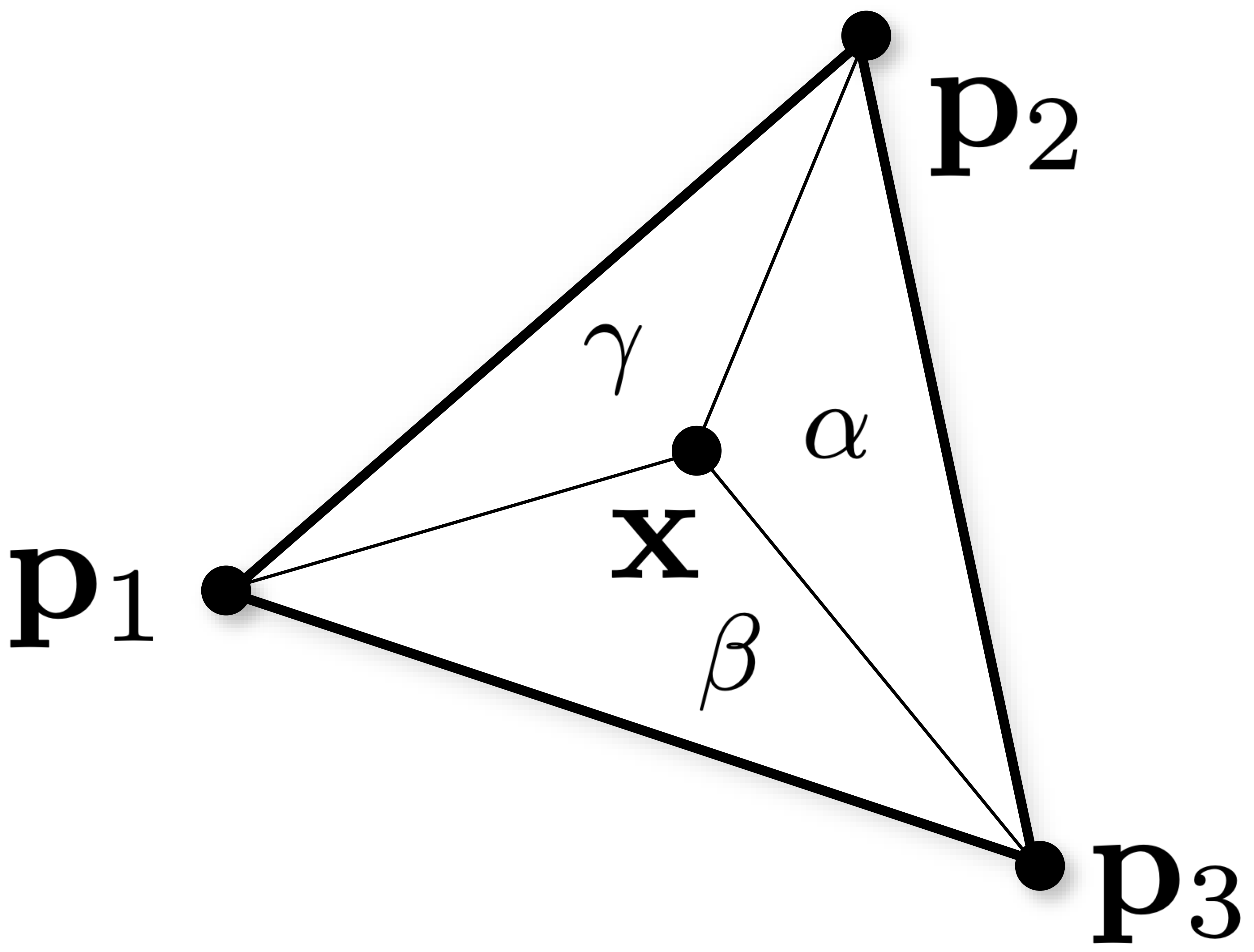
$$\gamma = 1 - \alpha - \beta \quad 0 \leq \beta \leq 1$$

$$0 \leq \gamma \leq 1$$



Interpretations of barycentric coords

Sub-triangle areas



$$\alpha = |\Delta \mathbf{p}_2\mathbf{p}_3\mathbf{x}| / |\Delta \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3|$$

$$\beta = |\Delta \mathbf{p}_1\mathbf{p}_3\mathbf{x}| / |\Delta \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3|$$

$$\gamma = |\Delta \underline{\mathbf{p}_1}\underline{\mathbf{p}_2}\mathbf{x}| / |\Delta \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3|$$

$$\mathbf{x} = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$$

Ray-Triangle Intersection (Approach 3)

Insert ray equation: $$\alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + (1 - \alpha - \beta)\mathbf{p}_3 = \mathbf{o} + t\mathbf{d}$$

$$\alpha(\mathbf{p}_1 - \mathbf{p}_3) + \beta(\mathbf{p}_2 - \mathbf{p}_3) + \mathbf{p}_3 = \mathbf{o} + t\mathbf{d}$$

$$\alpha(\mathbf{p}_1 - \mathbf{p}_3) + \beta(\mathbf{p}_2 - \mathbf{p}_3) - t\mathbf{d} = \mathbf{o} - \mathbf{p}_3$$

$$\alpha \mathbf{a} + \beta \mathbf{b} - t\mathbf{d} = \mathbf{e}$$

Solve directly

Can be much faster!

$$\begin{bmatrix} -\mathbf{d} & \mathbf{a} & \mathbf{b} \end{bmatrix} \begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \mathbf{e}$$

Ray-Surface Intersections

Other primitives

- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.

Intersecting transformed primitive?

Option 1: Transform the primitive

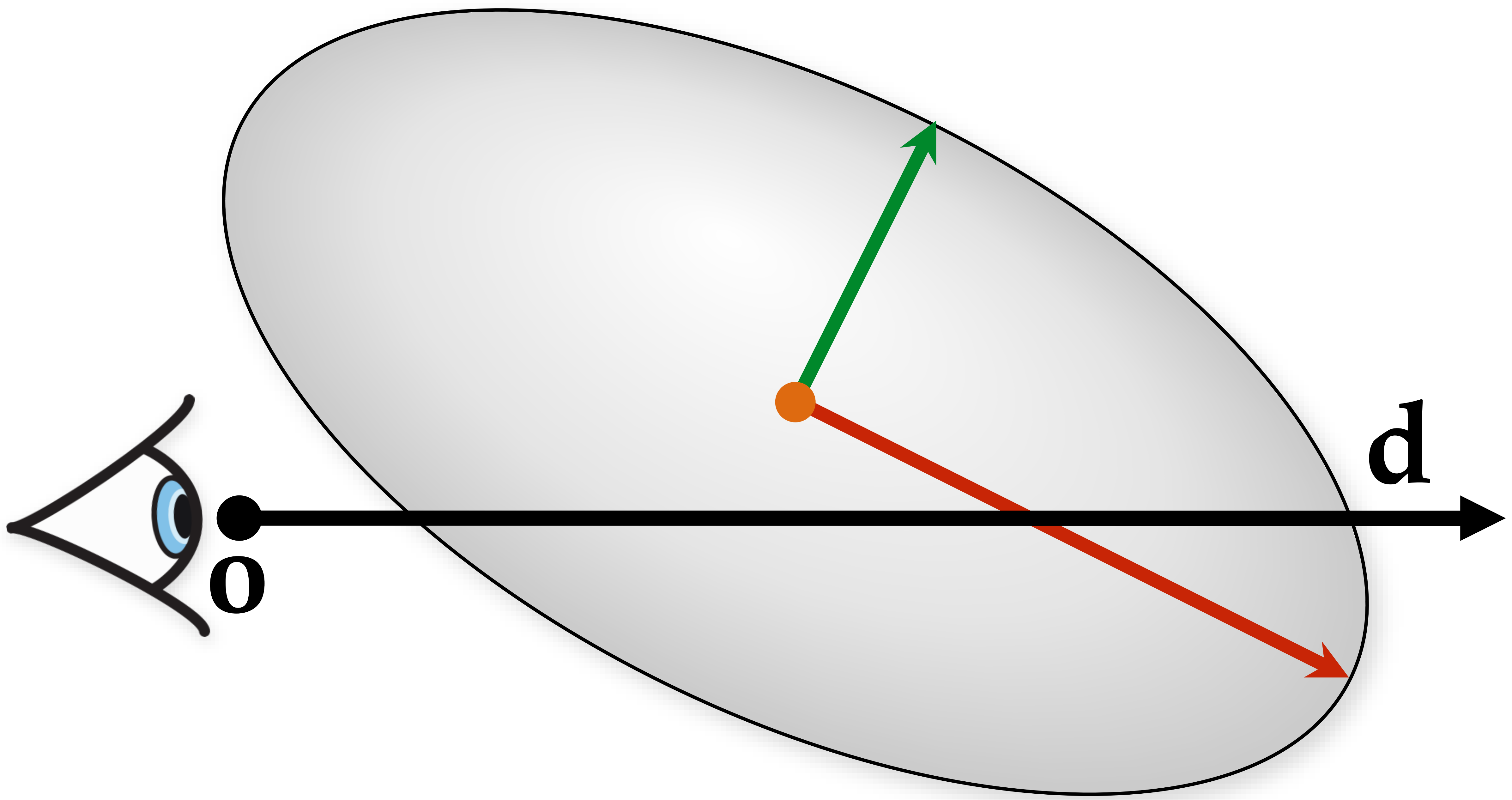
- simple for triangles, since they transform to triangles
- other primitives get more complicated (e.g. sphere $\longrightarrow$ ellipsoid)

Option 2: Transform the ray (by the inverse transform)

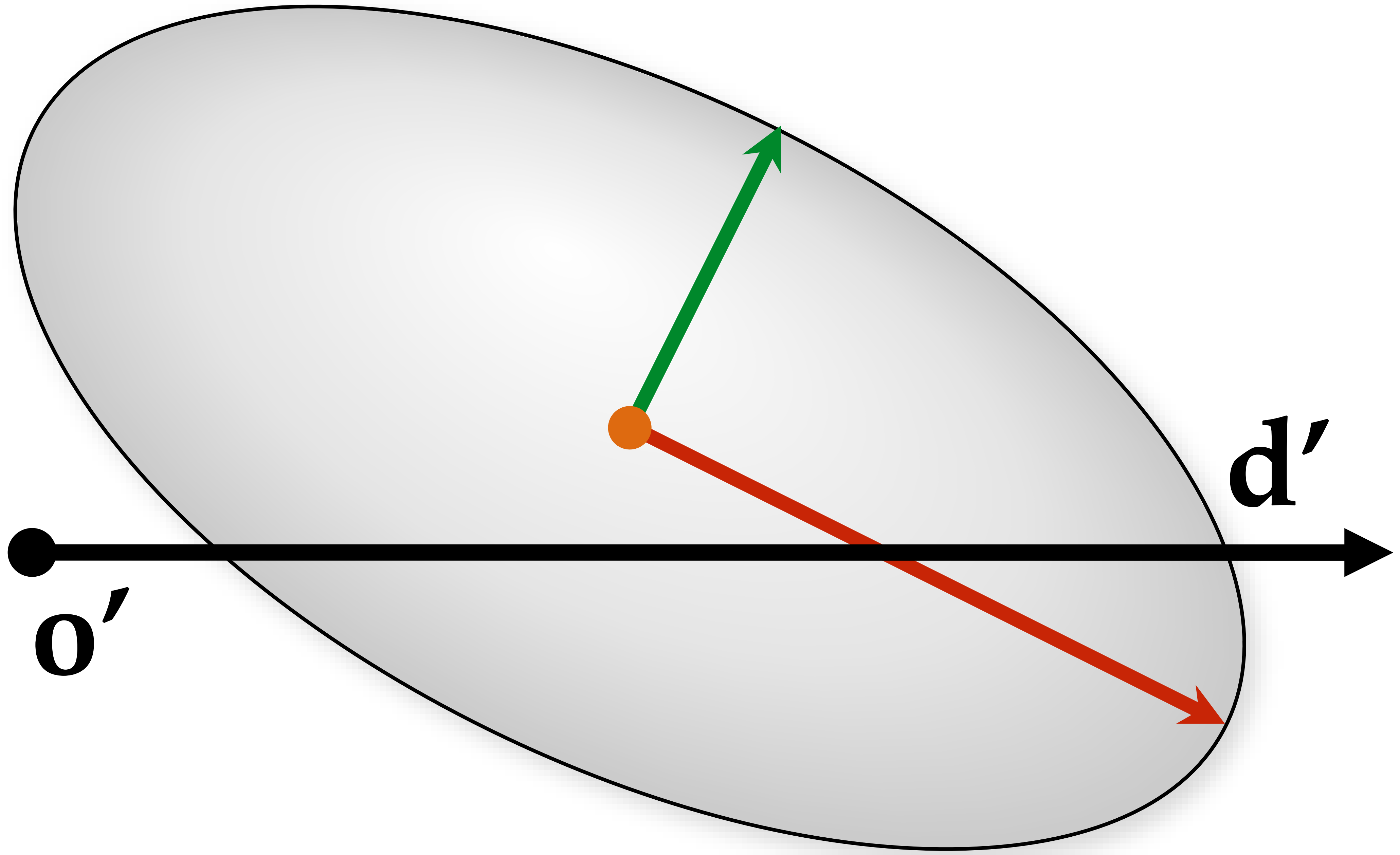
- more elegant; works on any primitive
- allows simpler intersection tests
  (e.g., just use existing sphere-intersection routine)

Intersection and coordinate systems

World space

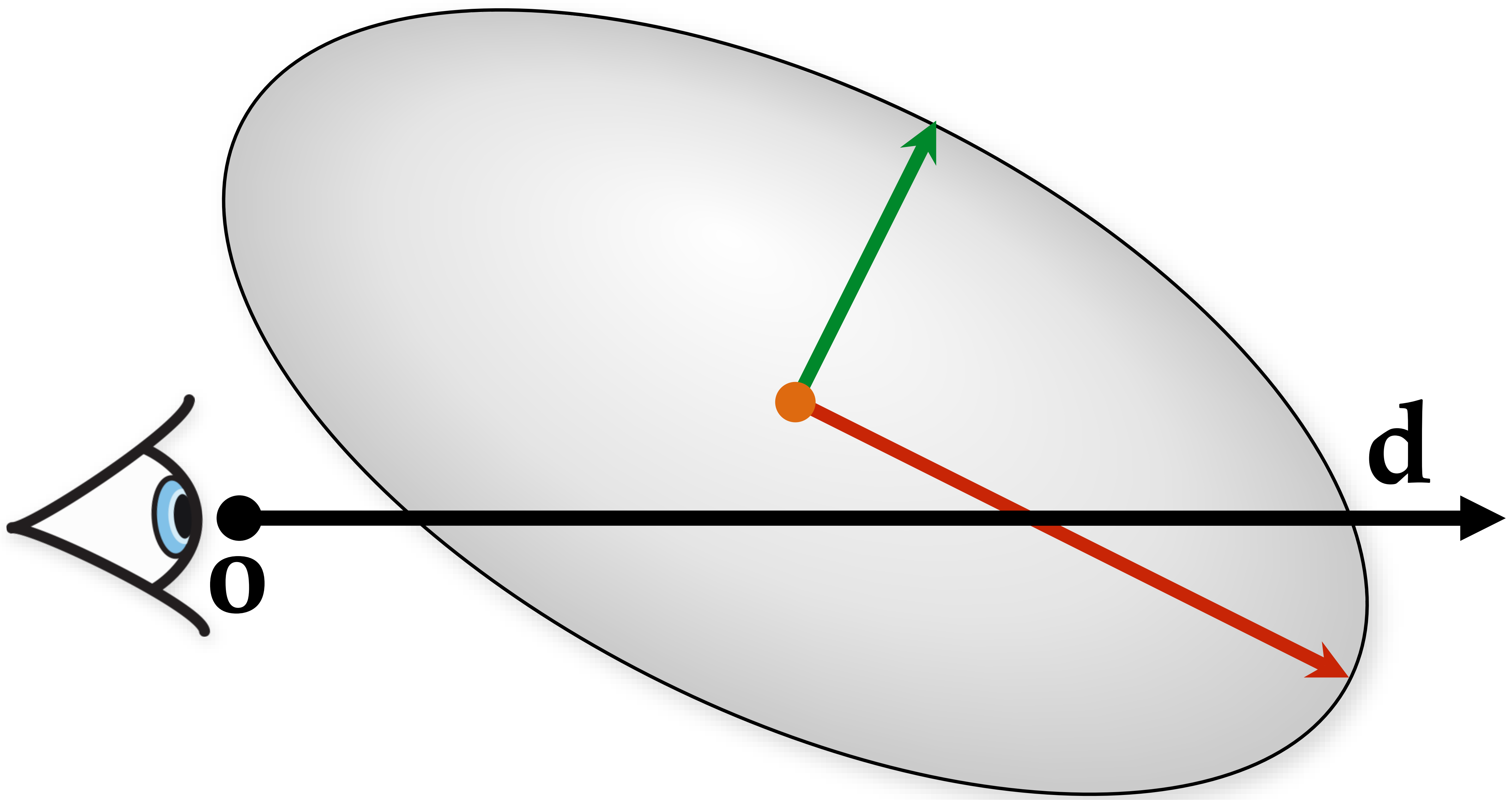


Local space

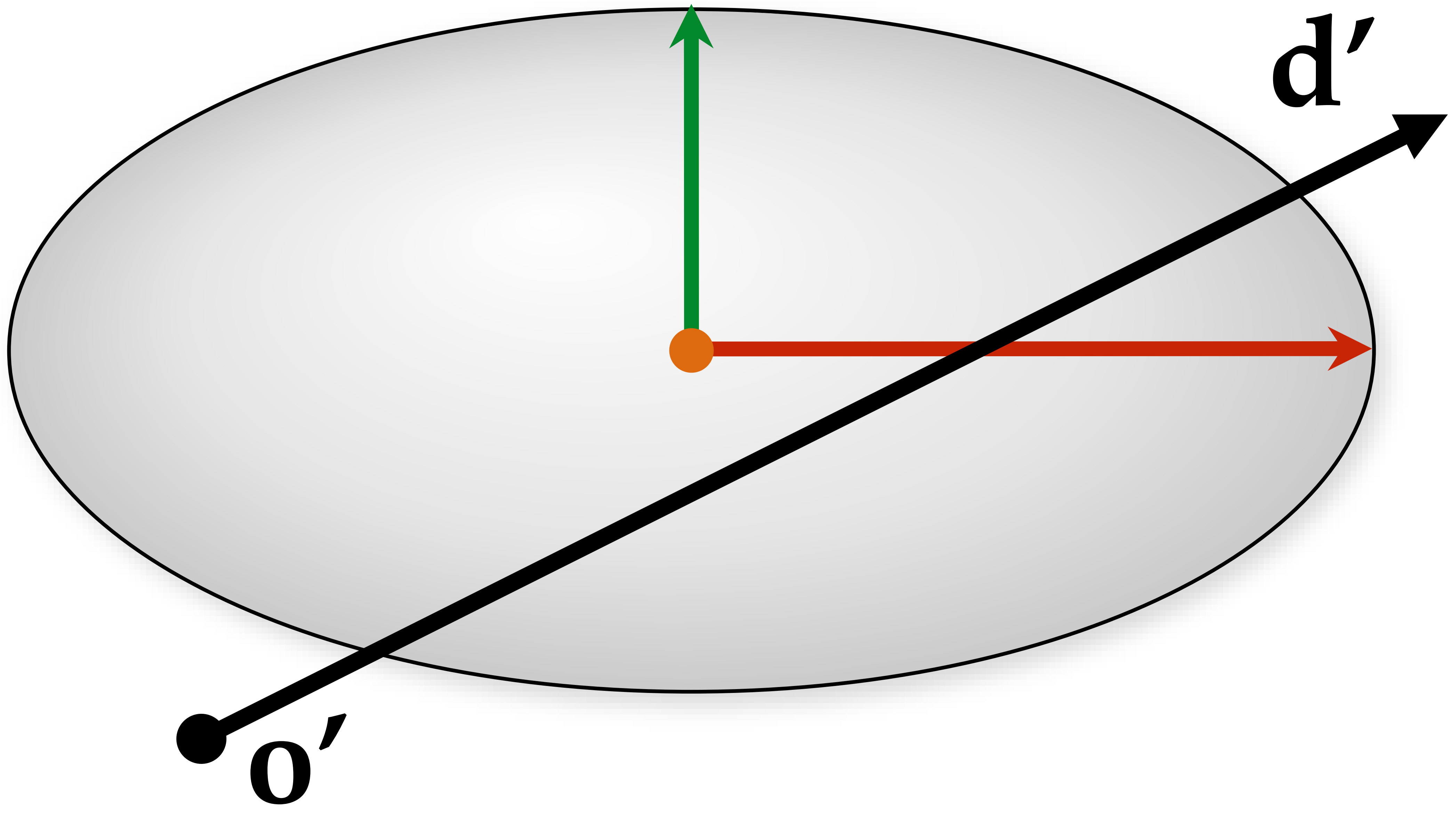


Intersection and coordinate systems

World space

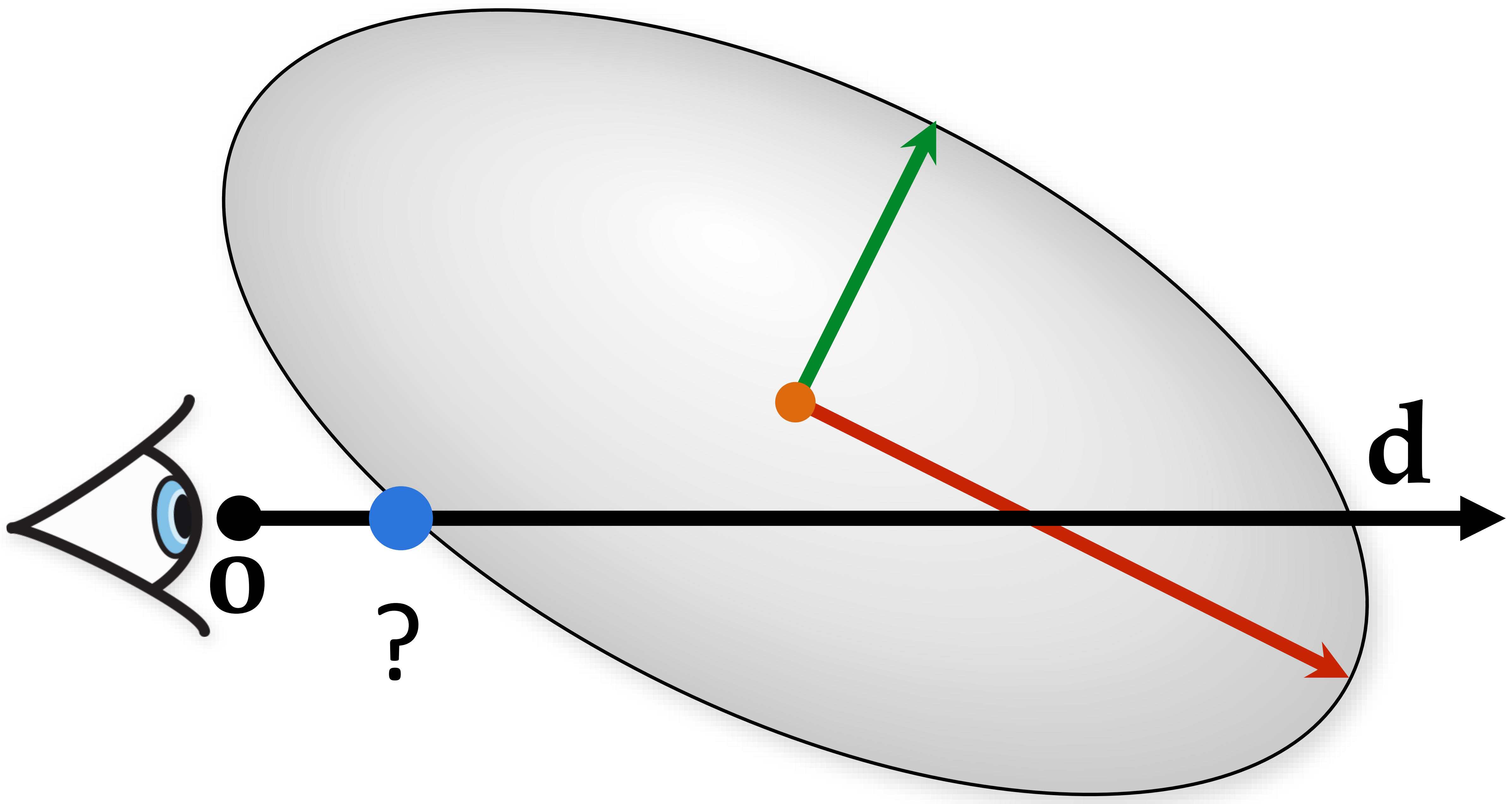


Local space

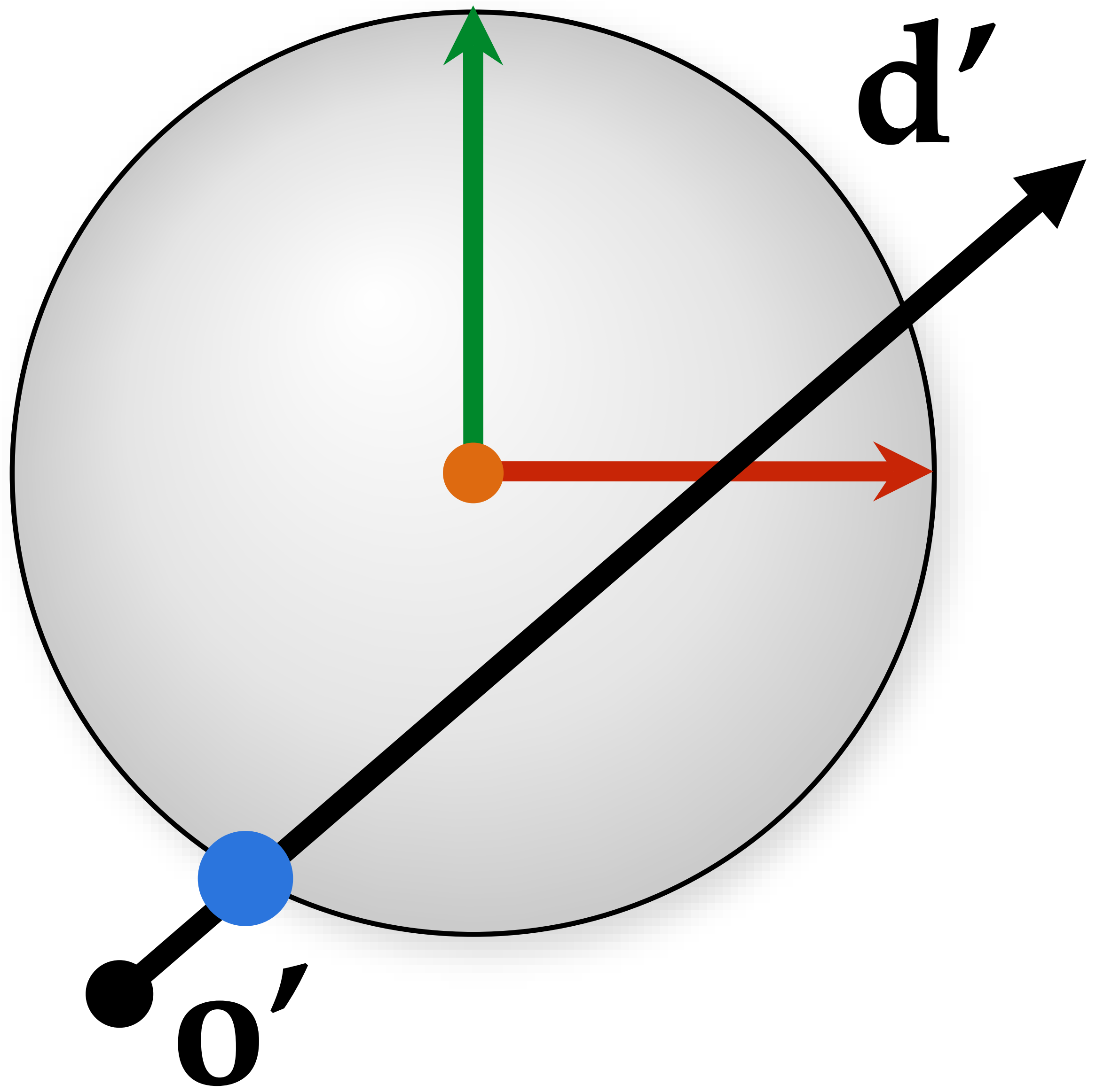


Intersection and coordinate systems

World space



Local space



We have a sphere now

But with a different ray

Transformations in homogeneous coords

A 3D transformation matrix:

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{24} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix}$$

affine

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ m_{31} & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & m_{44} \end{bmatrix}$$

A 3D homogenous vector:

$$\mathbf{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix}$$

A position has $w \neq 0$, and a direction has $w = 0$

Transformations

Matrix-vector multiplication, $M\mathbf{v}$, transforms the vector

A translation matrix:

$$M_{\mathbf{t}} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

A scaling matrix:

$$M_{\mathbf{s}} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Intersection and coordinate systems

Have a transform M, a ray $\mathbf{r}(t)$, and a surface S

To intersect:

1. Transform ray to local coords (by inverse of M)
2. Call surface intersection
3. Transform hit data back to global coords (by M)

How to transform a ray $\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$ by M^{-1}?

- $\mathbf{r}'(t) = M^{-1}\mathbf{p} + tM^{-1}\mathbf{d}$
- Remember: $\mathbf{p}$ forms as a point, $\mathbf{d}$ as a direction!

Ray-Surface Intersections

Other primitives

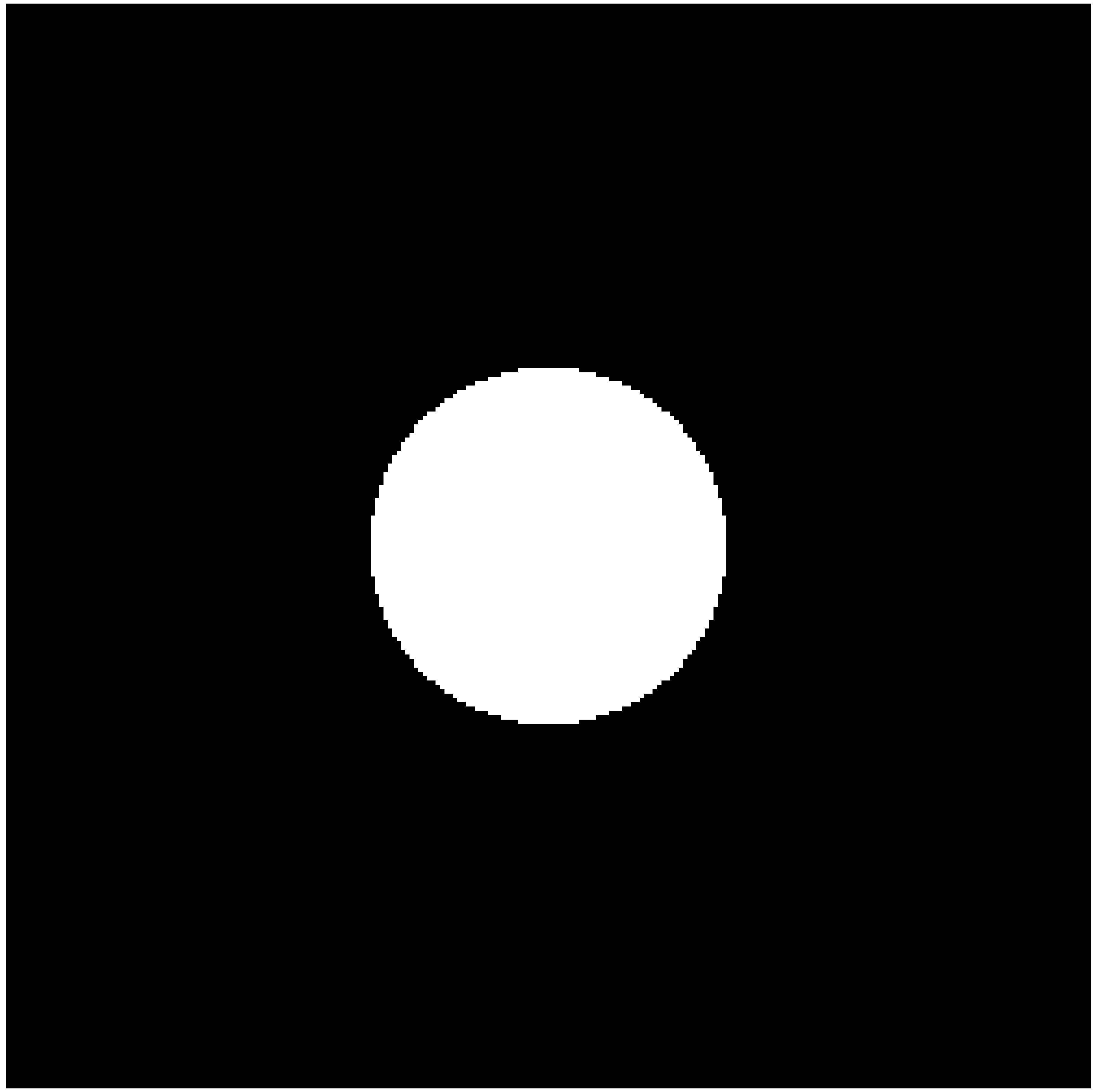
- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.

Image so far

With eye ray generation and sphere intersection

```
parse scene description

for each pixel:
   ray = camera.getRay(pixel);
   hit = s.intersect(ray, 0, +inf);
   if hit:
      image.set(pixel, white);
```



After a slide by Steve Marschner

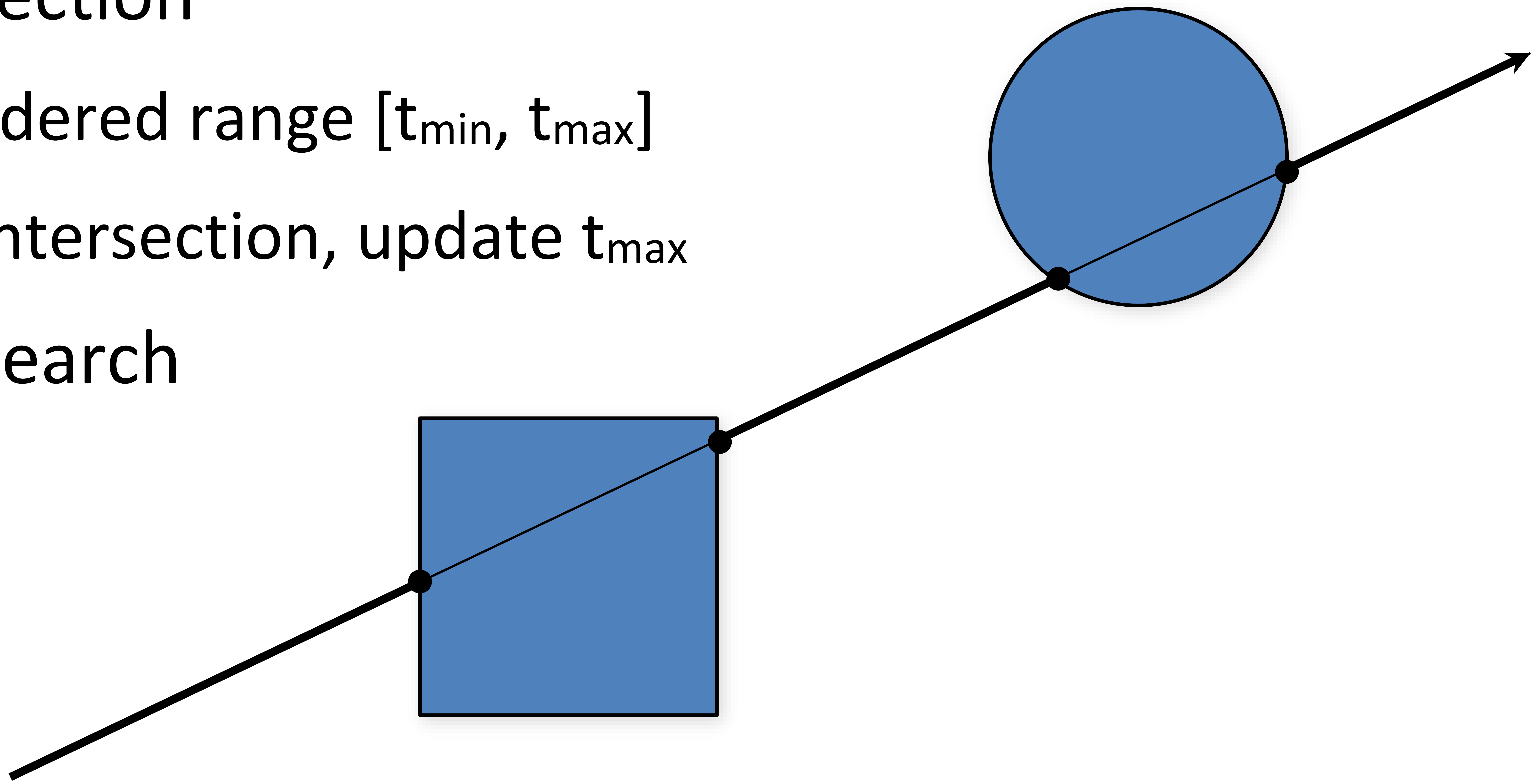
Intersecting many shapes

Intersect each primitive

Pick closest intersection

- Only within considered range [t_{min}, t_{max}]
- After each valid intersection, update t_{max}

Essentially a line search



Intersection against many shapes

The basic idea is:

```
Surfaces::intersect(ray, tMin, tMax):
    tBest = +inf; firstHit = null;
    for s in surfaces:
        hit = s.intersect(ray, tMin, tBest);
        if hit:
            tBest = hit.t;
            firstHit = hit;
    return firstHit;
```

- this is linear in number of surfaces but there are sublinear methods (acceleration structures)

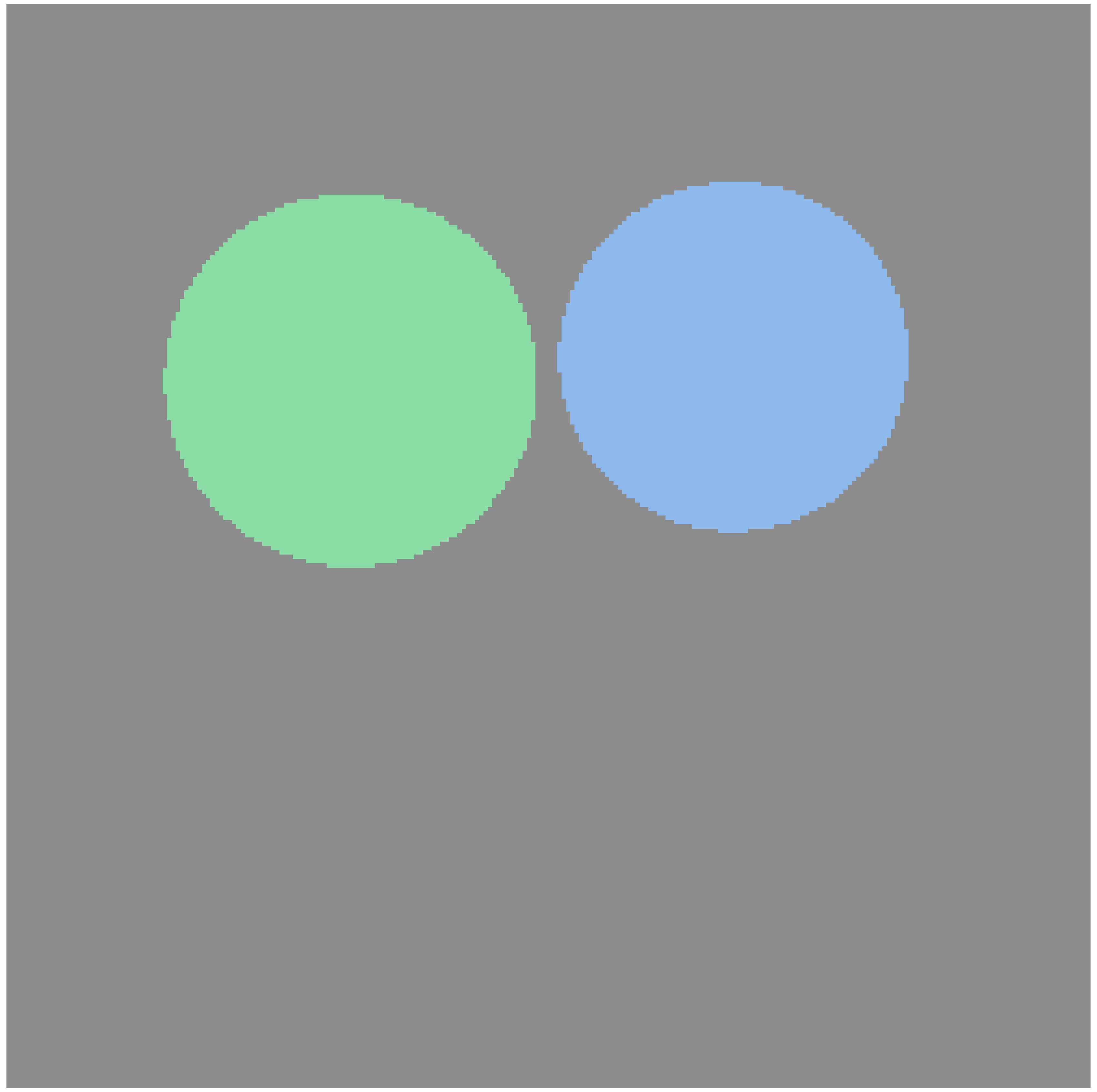
After a slide by Steve Marschner

Image so far

With eye ray generation and scene intersection

```
for each pixel:
   ray = camera.getRay(pixel);
   c = scene.trace(ray, 0, +inf);
   image.set(pixel, c);

Scene::trace(ray, tMin, tMax):
   hit = surfaces.intersect(ray, tMin, tMax);
   if (hit)
      return hit.color();
   else
      return backgroundColor;
```



After a slide by Steve Marschner

Ray-Surface Intersections

Other primitives

- cylinder
- cone, paraboloid, hyperboloid
- torus
- disk
- general polygons, meshes
- etc.

How should we represent complex geometry?

How are they obtained?

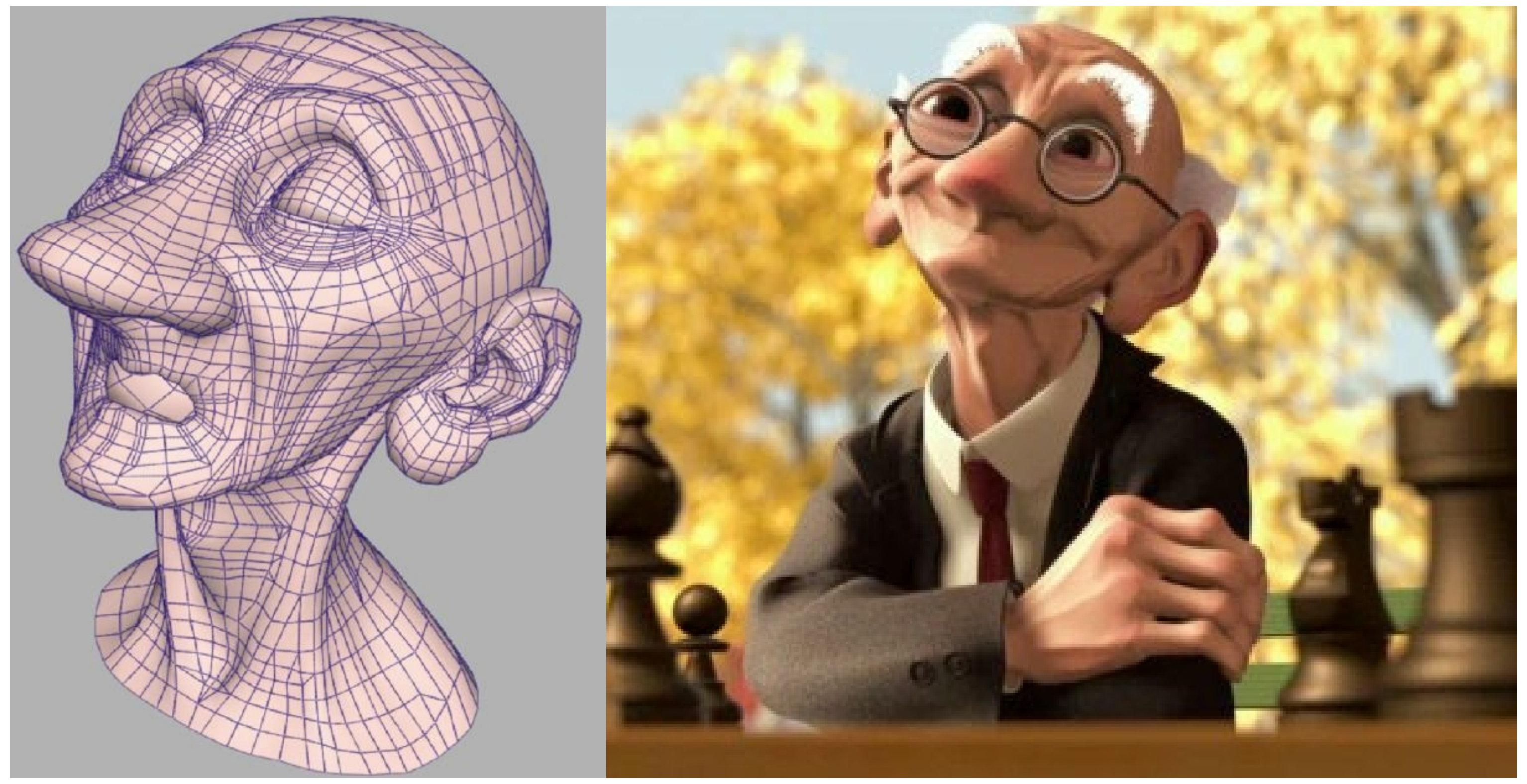
- modeled by hand
- scanned

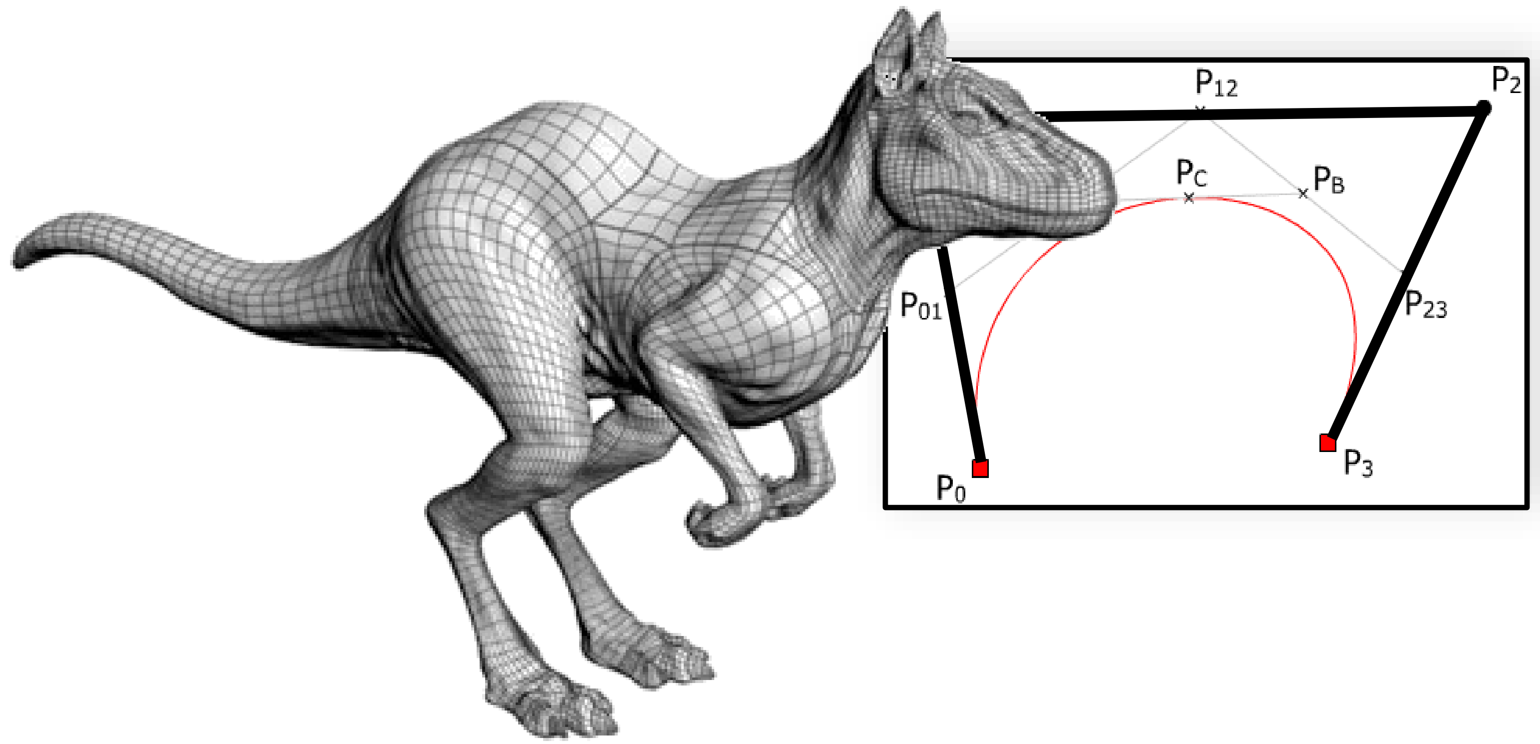
What operations must we support?

- modeling/editing
- animating
- **texturing**
- **rendering**

Surface representation zoo!

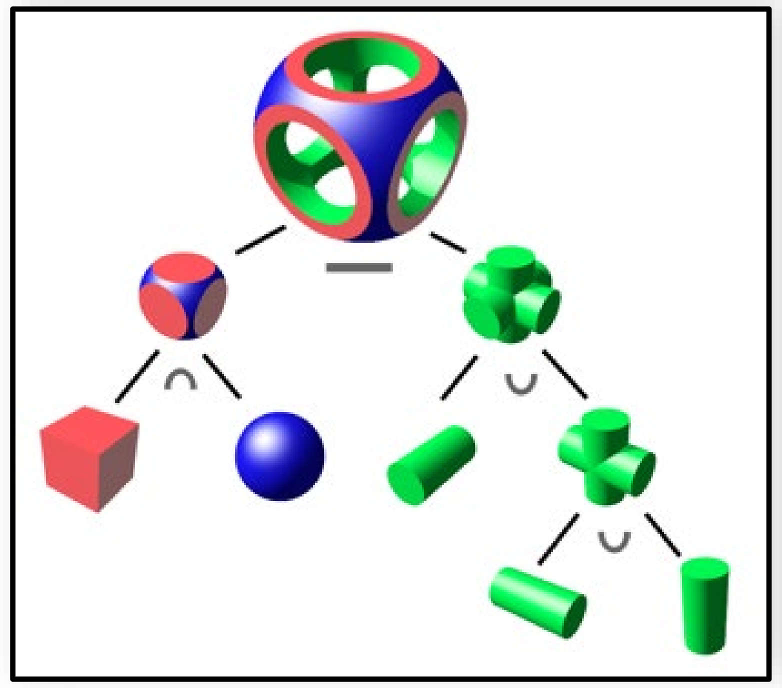
Parametric

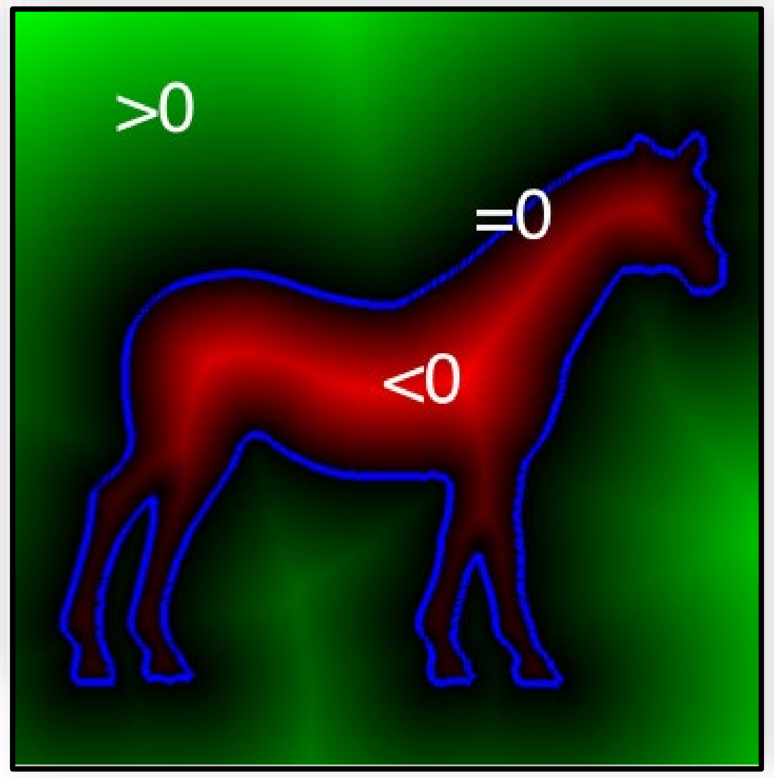


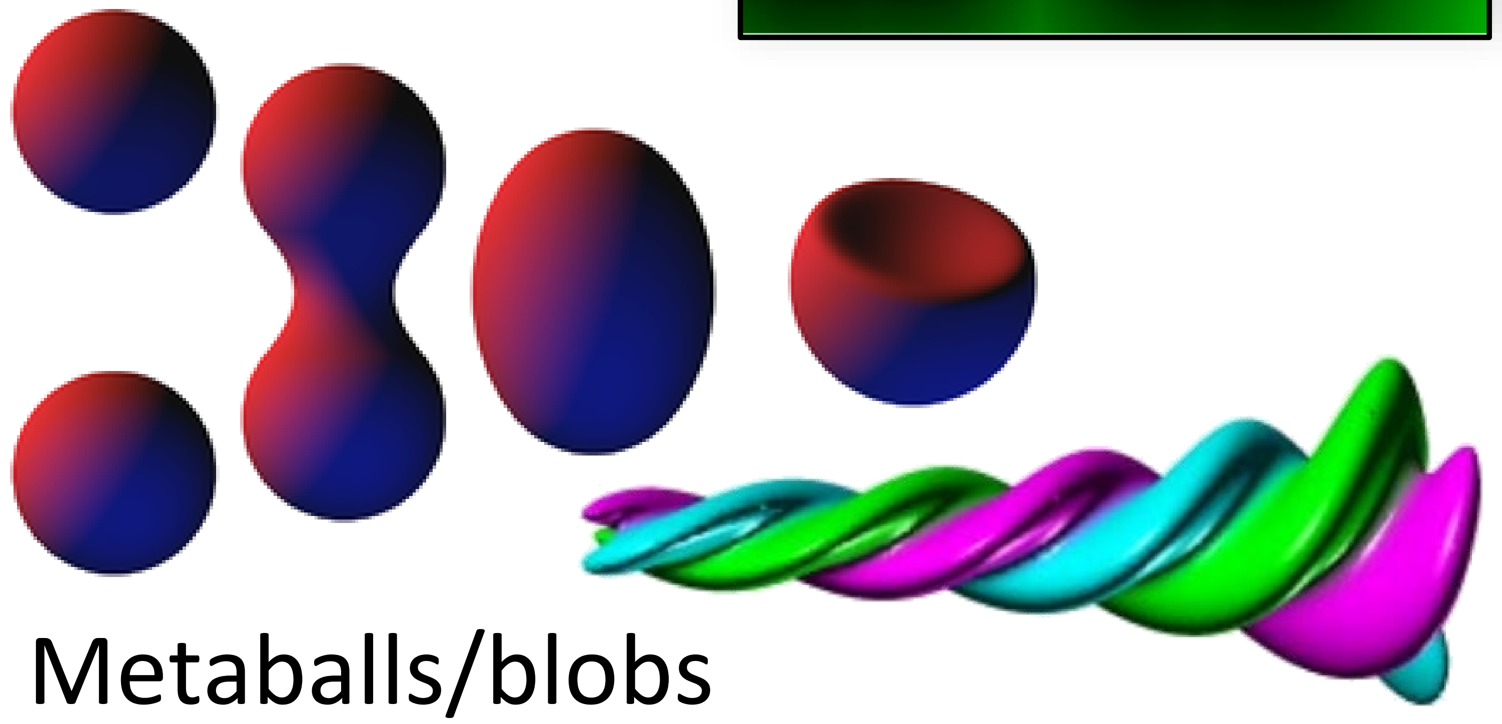


- Splines, tensor-product surfaces
- Subdivision surfaces

Implicit

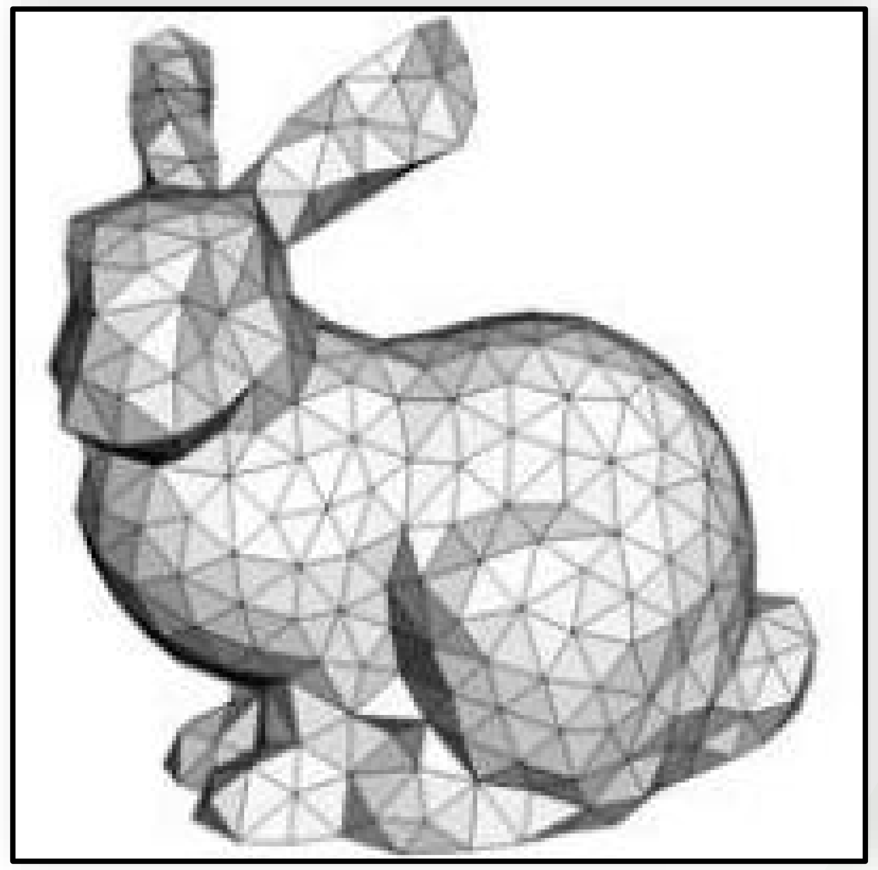


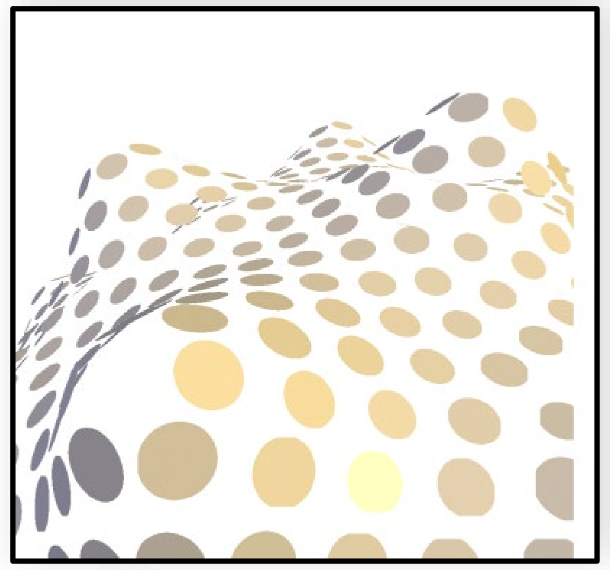


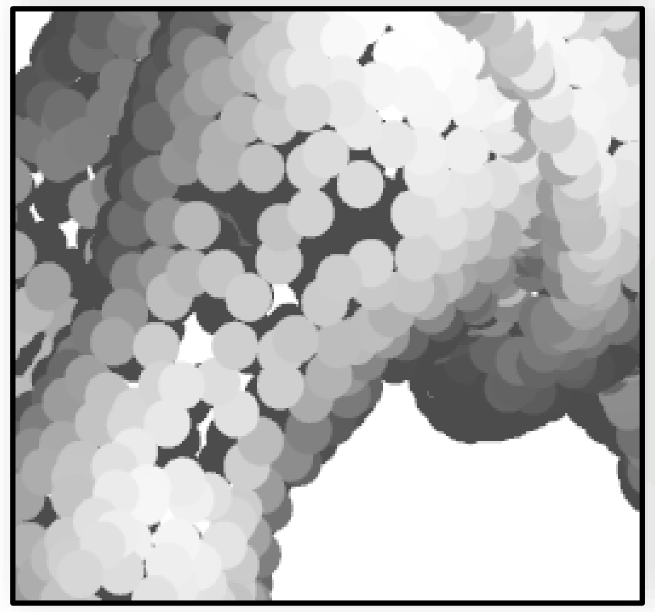


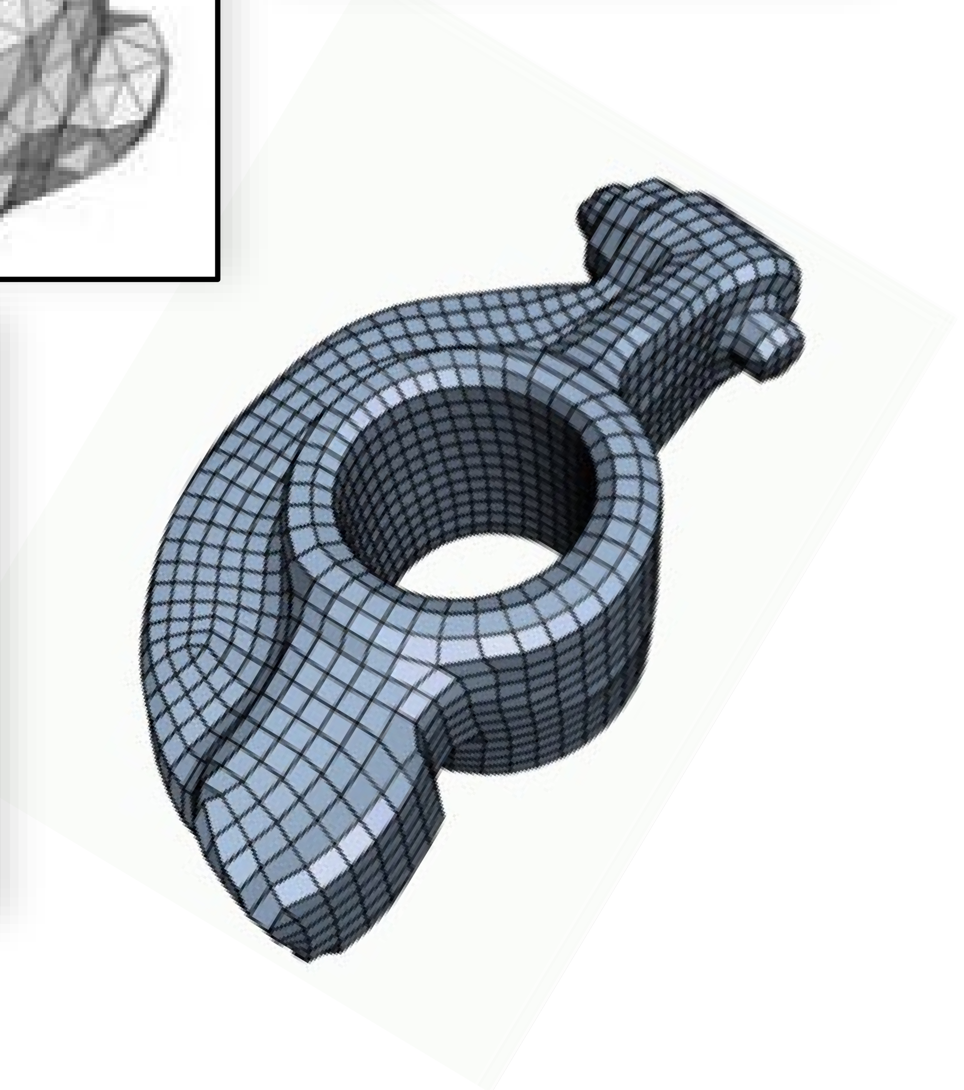
- Metaballs/blobs
- Distance fields
- Procedural, CSG
- Neural nets

Discrete/Sampled







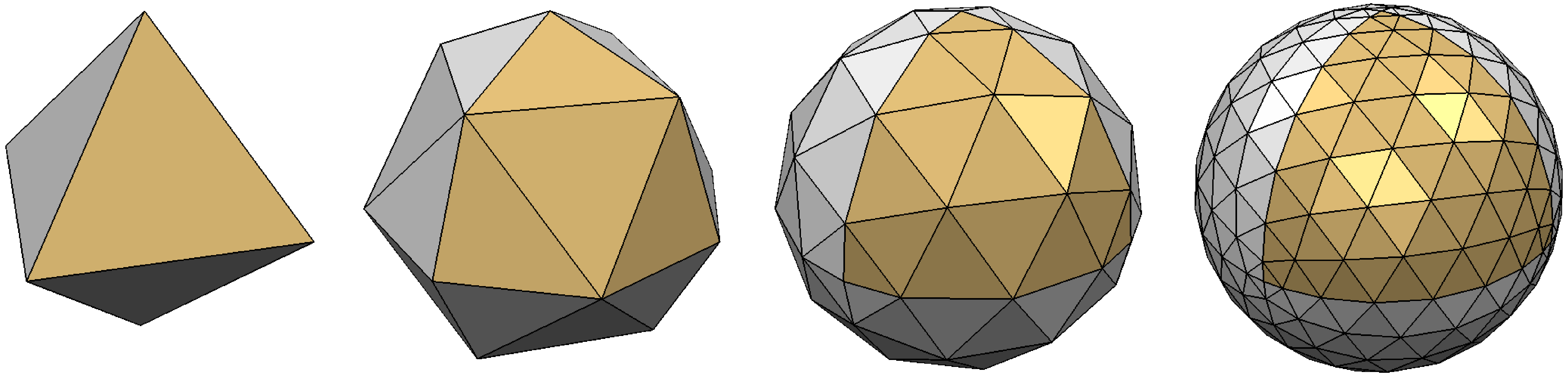


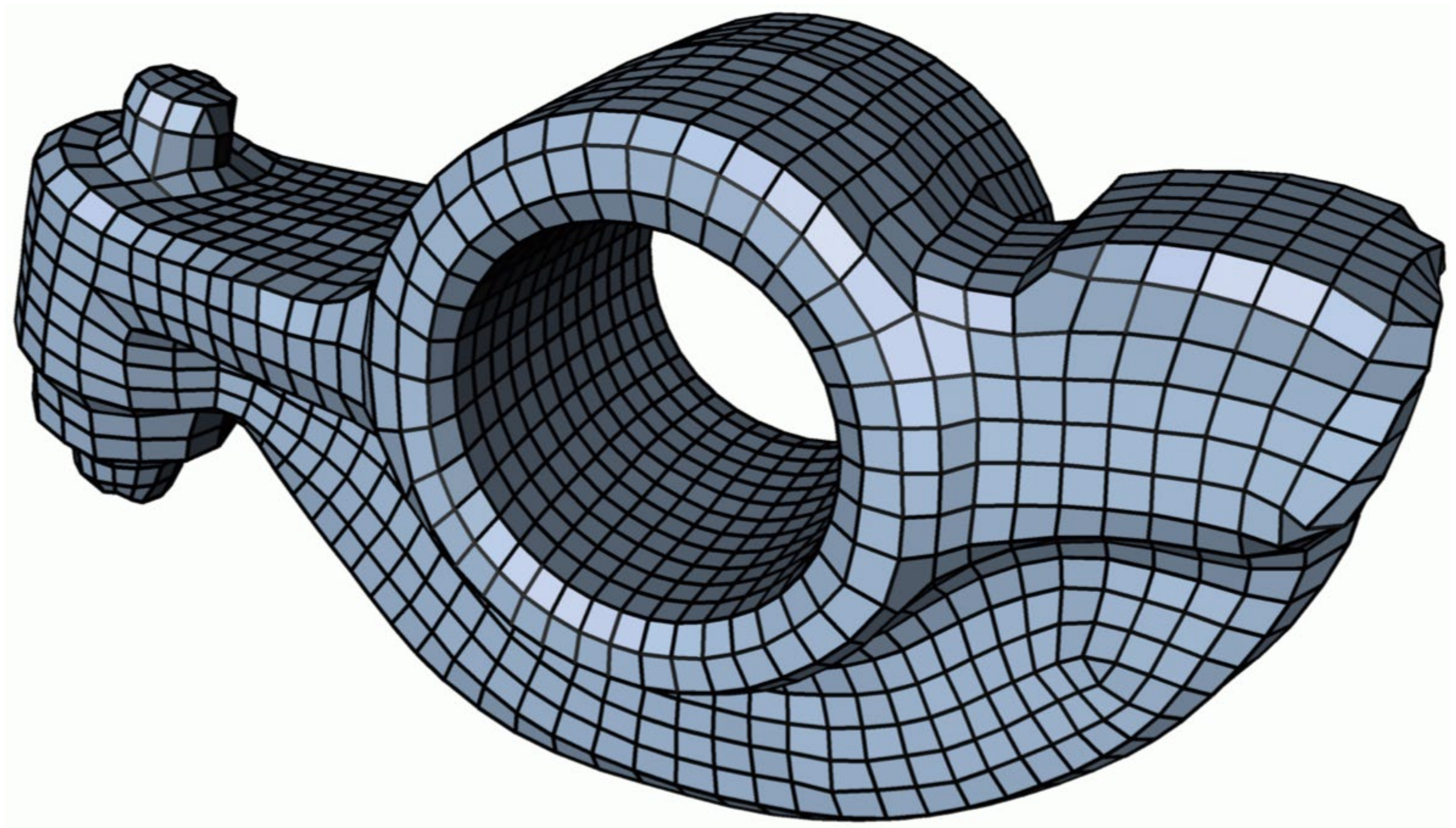
- Meshes
- Point set surfaces

Polygonal Meshes

Boundary representations of objects

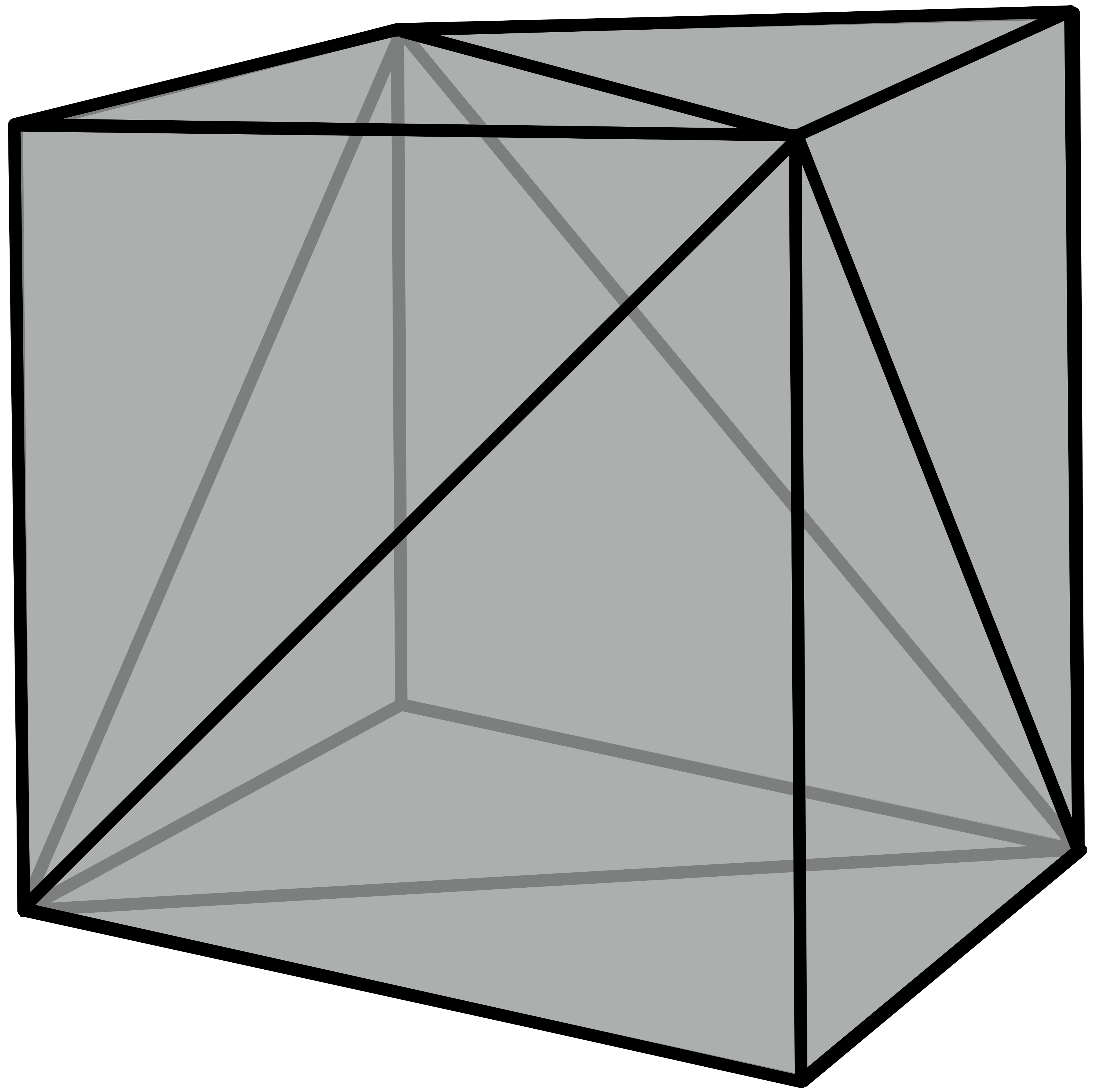
- Piecewise linear





After a slide by Olga Sorkine-Hornung

A small triangle mesh



12 triangles, 8 vertices

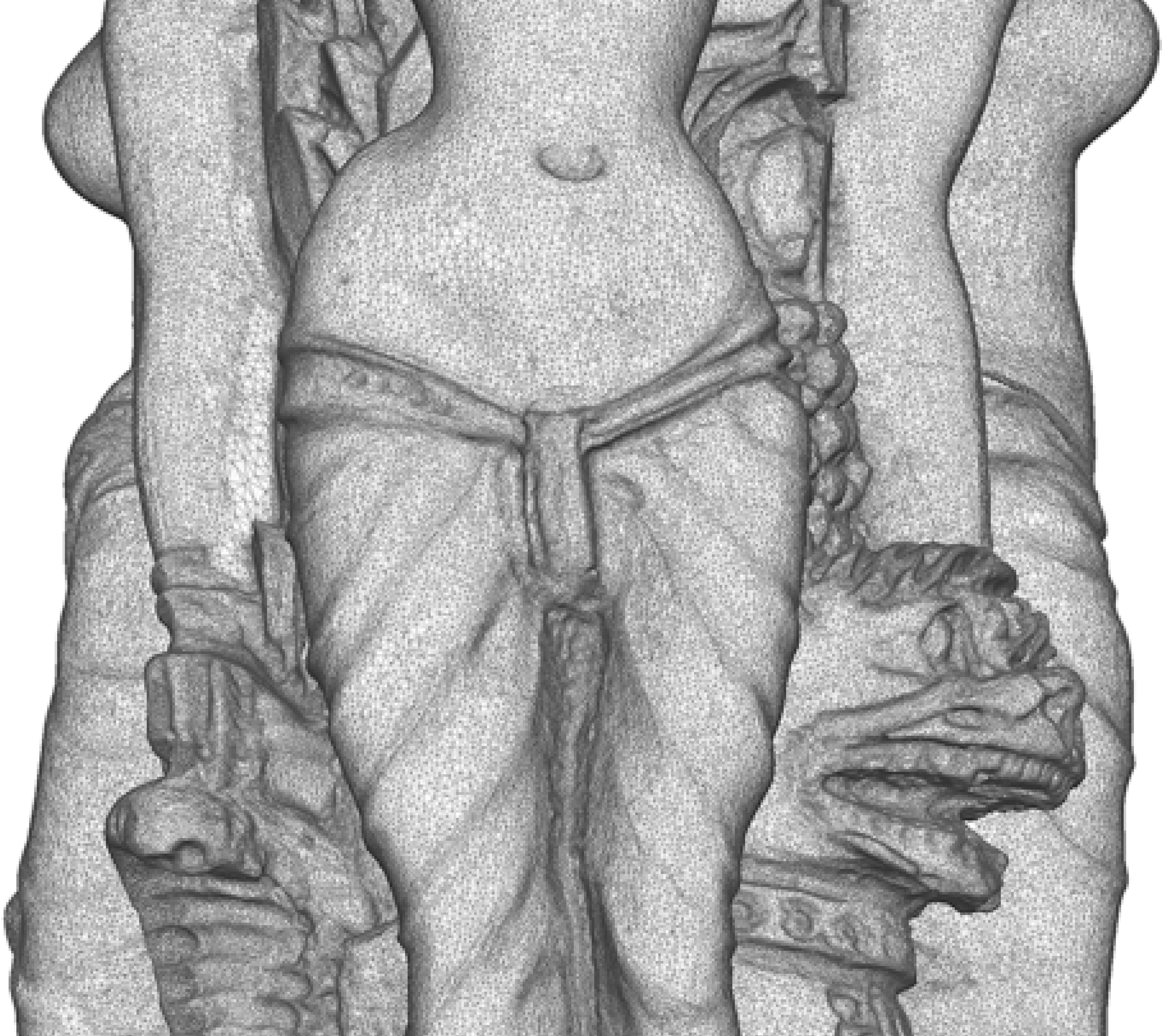
After a slide by Steve Marschner

A large mesh

10 million triangles from a high-resolution 3D scan

After a slide by Steve Marschner

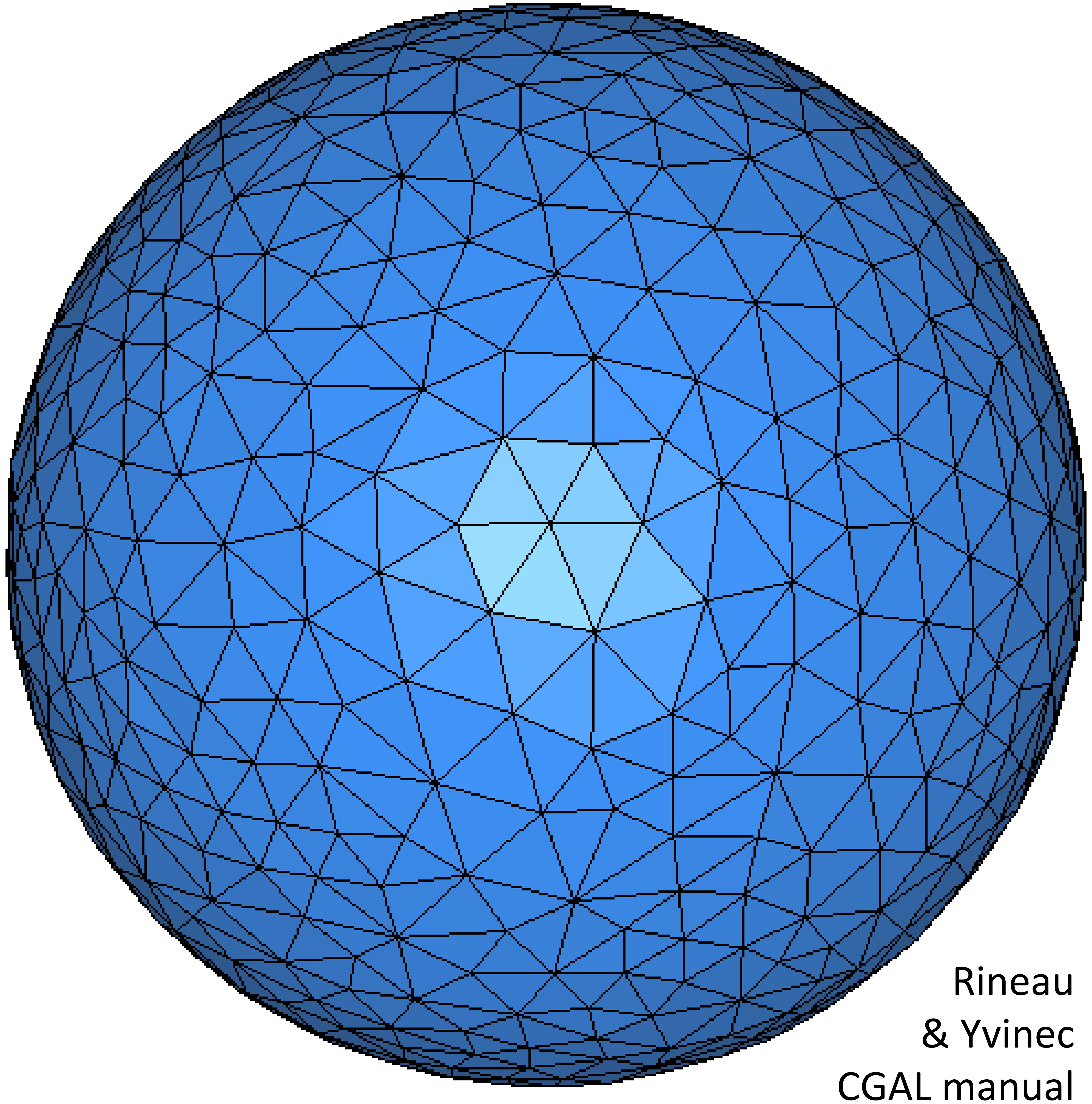
Traditional Thai sculpture—scan by XYZRGB, inc., image by MeshLab project



After a slide by Steve Marschner

After a slide by Steve Marschner

spheres



Rineau
& Yvinec
CGAL manual

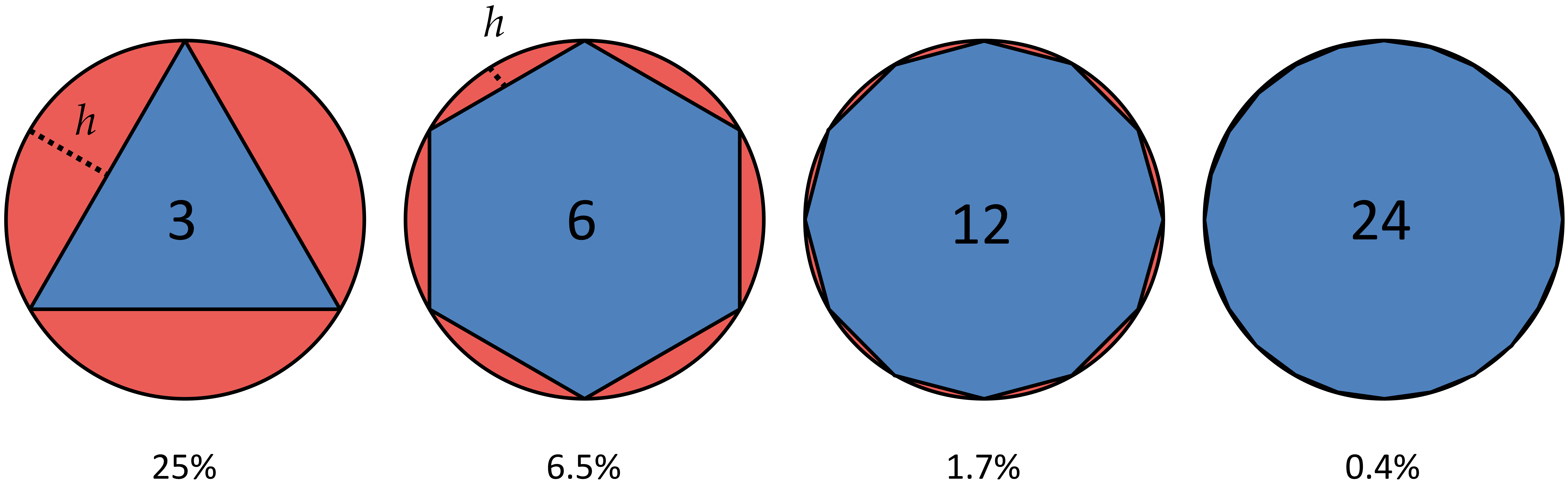
approximate
sphere

After a slide by Steve Marschner

Meshes as Approx. of Smooth Surfaces

Piecewise linear approximation

- Error is $O(h^2)$

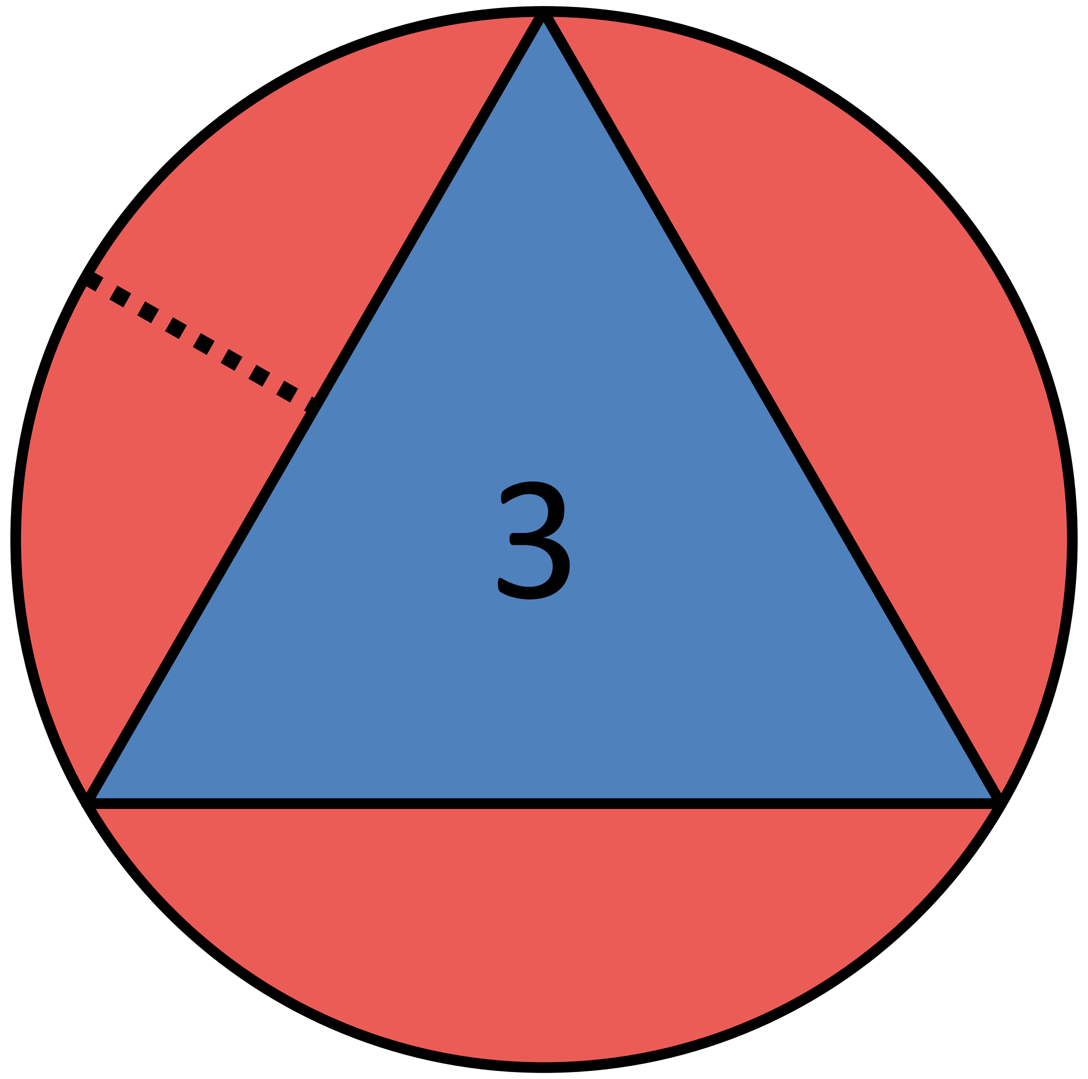


After a slide by Olga Sorkine-Hornung

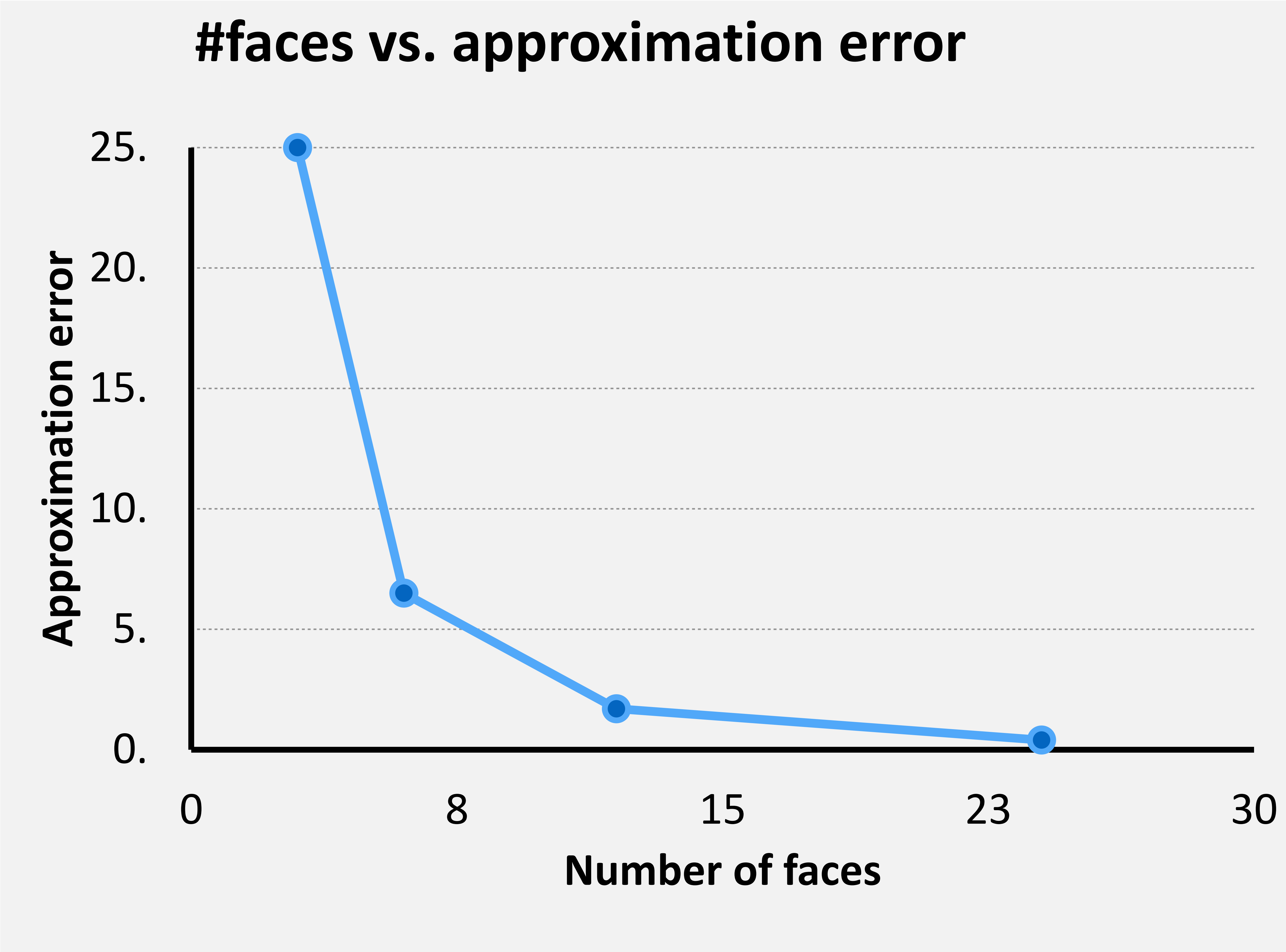
Meshes as Approx. of Smooth Surfaces

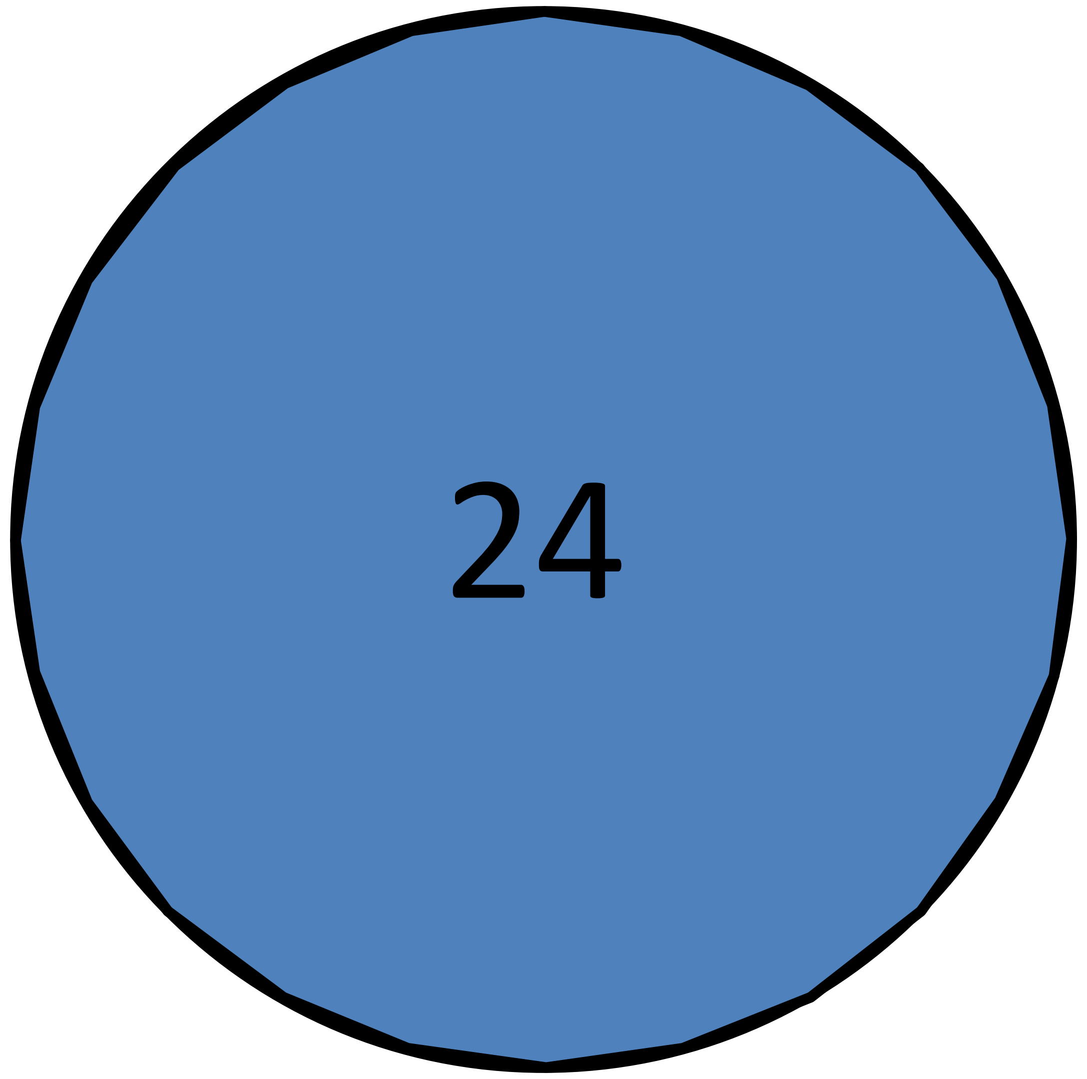
Piecewise linear approximation

- Error is $O(h^2)$



25%





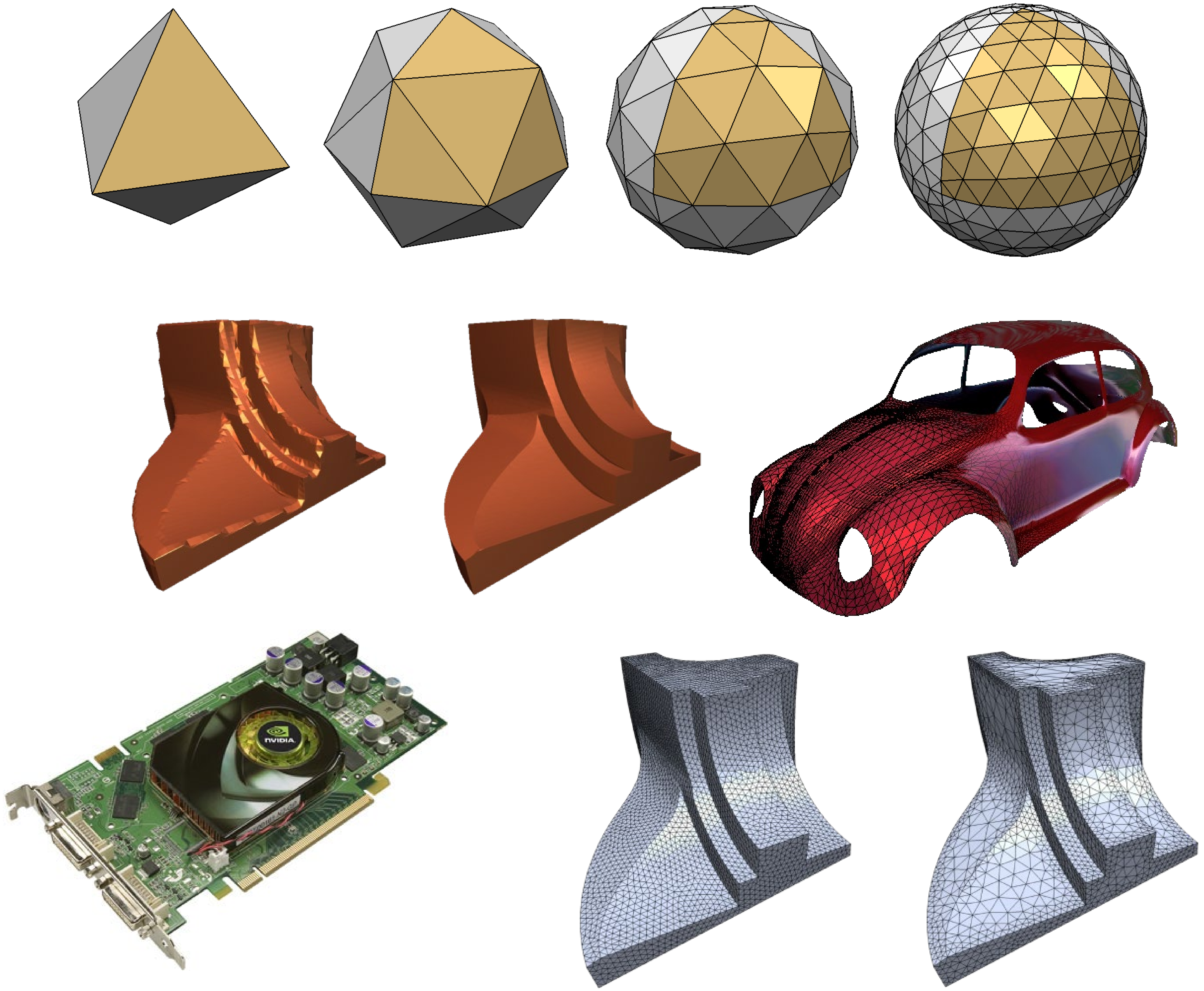
0.4%

After a slide by Olga Sorkine-Hornung

Polygonal Meshes

Polygonal meshes are a good representation

- approximation $O(h^2)$
- arbitrary topology
- piecewise smooth surfaces
- adaptive refinement
- efficient rendering



After a slide by Olga Sorkine-Hornung

Data Structures: What should be stored?

After a slide by Olga Sorkine-Hornung

Geometry: 3D coordinates

Attributes

- Normal, color, texture coordinates
- Per vertex, face, edge

Connectivity

- Adjacency relationships

Separate Triangle List or Face Set (STL)

Face: 3 vertex positions

Storage:

- 4 Bytes/coordinate (using 32-bit floats)
- 36 Bytes/face

Wastes space

Triangles			
0	x0	y0	z0
1	x1	y1	z1
2	x2	y2	z2
3	x3	y3	z3
4	x4	y4	z4
5	x5	y5	z5
6	x6	y6	z6
…	…	…	…

Indexed Face Set (OBJ, OFF, WRL)

Vertex: position

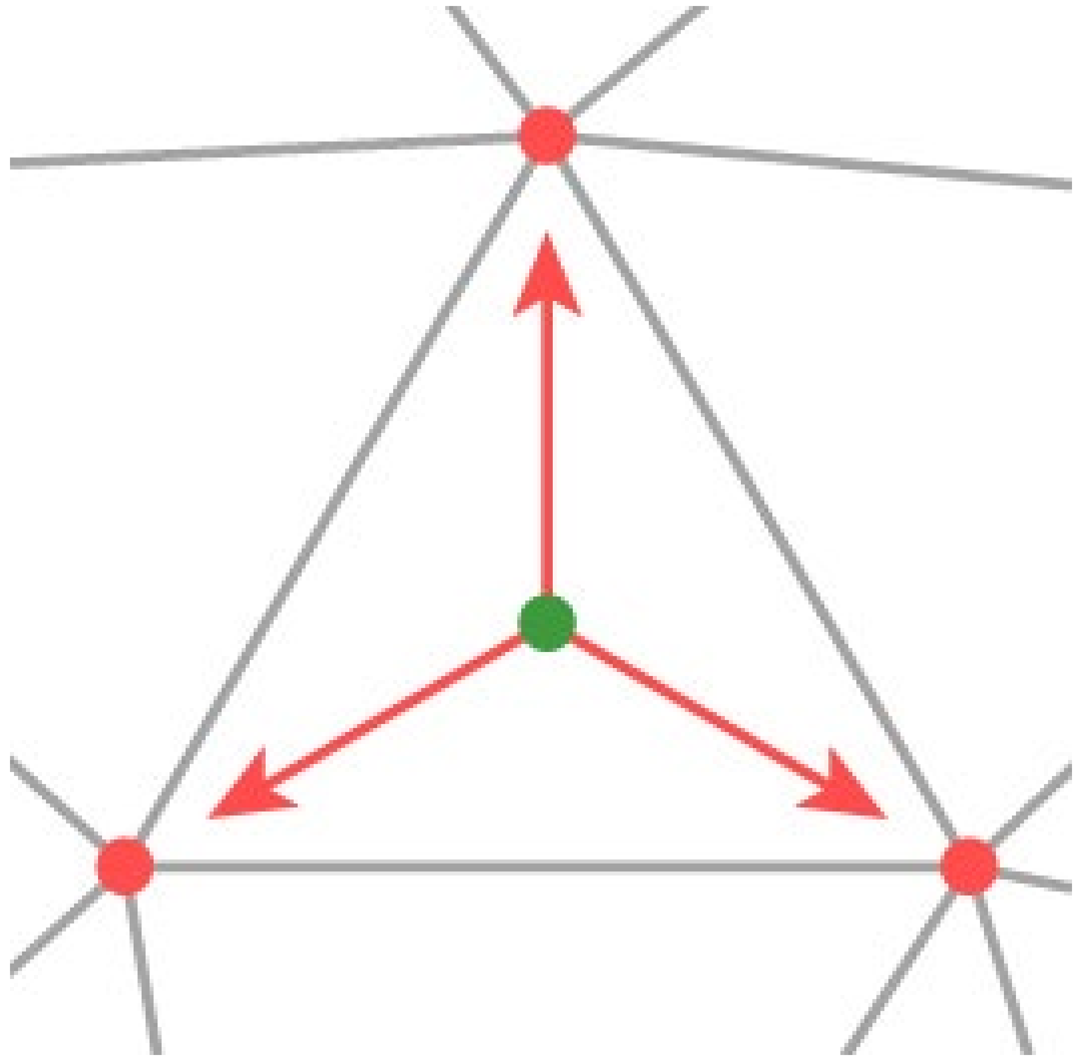
Face: vertex indices

Storage:

- 12 Bytes/vertex
- 12 Bytes/face

Reduces wasted space

Even better with per-vertex attributes



Triangles			
t0	v0	v1	v2
t1	v0	v1	v3
t2	v2	v4	v3
t3	v5	v2	v6
…	…	…	…

Vertices			
v0	x0	y0	z0
v1	x1	x1	z1
v2	x2	y2	z2
v3	x3	y3	z3
v4	x4	y4	z4
v5	x5	y5	z5
v6	x6	y6	z6
…	…	…	…

Data on meshes

Often need to store additional information besides just the geometry

Can store additional data at faces, vertices, or edges

Examples

- colors stored on faces, for faceted objects
- information about sharp creases stored at edges
- any quantity that varies *continuously* (without sudden changes, or *discontinuities*) gets stored at vertices

After a slide by Steve Marschner

Key types of vertex data

Surface normals

- when a mesh is approximating a curved surface, store normals at vertices

Texture coordinates

- 2D coordinates that tell you how to paste images on the surface

Positions

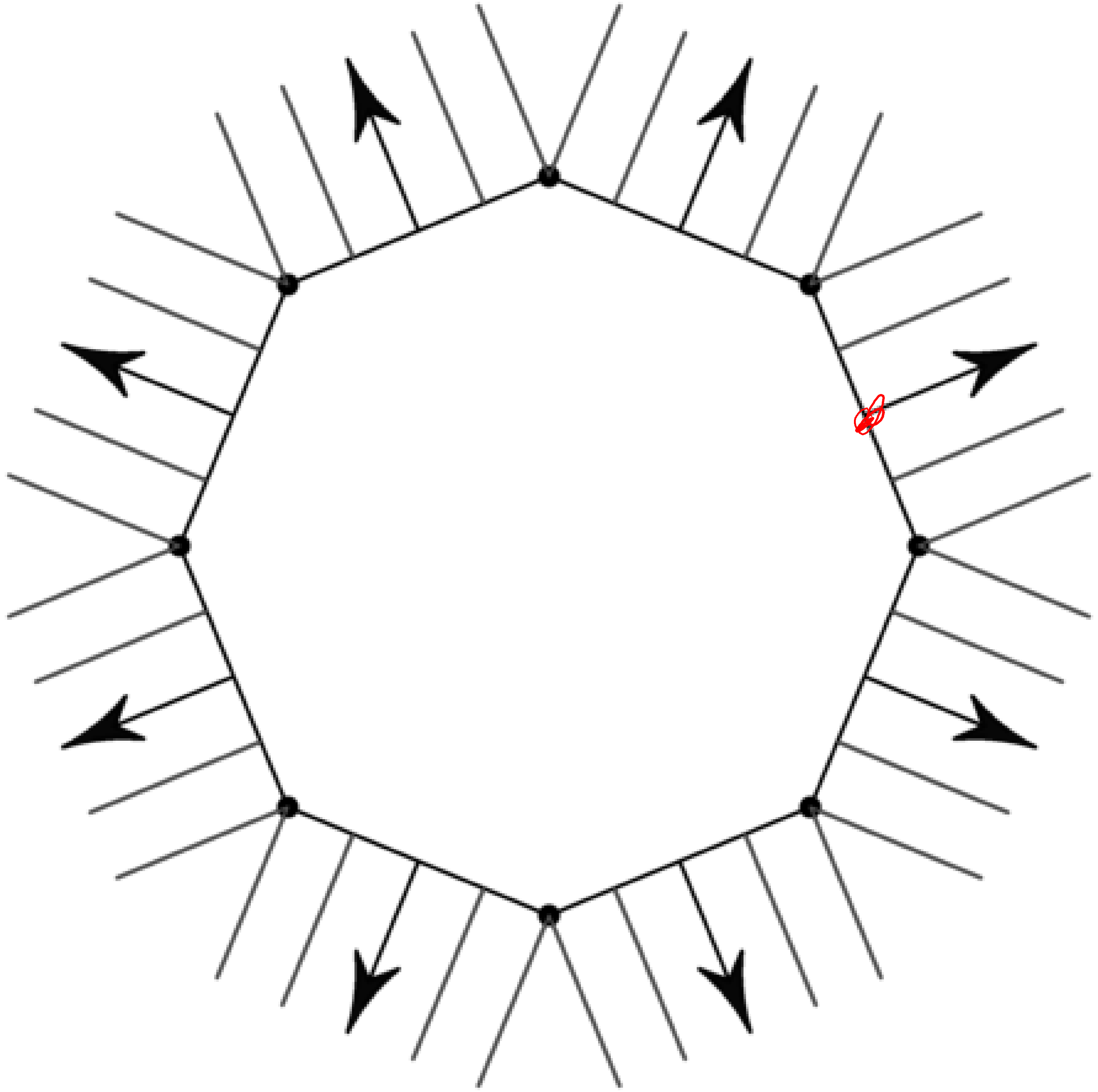
- at some level this is just another piece of data

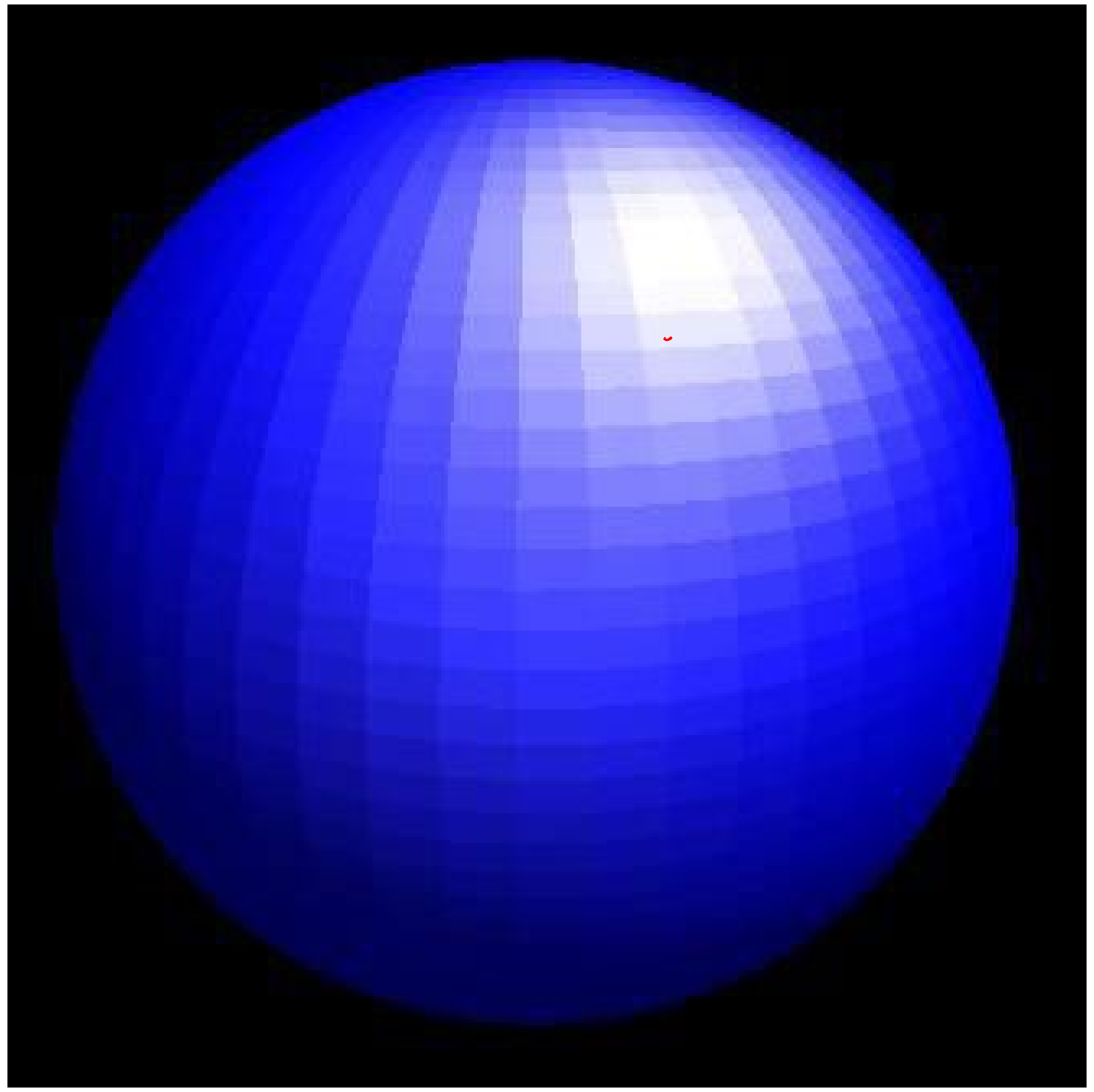
After a slide by Steve Marschner

Defining normals

Face normals: same normal for all points in face

- geometrically correct, but faceted look





After a slide by Jon Denning

Problems with face normals

Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases

- error is $O(h^2)$

But the surface normals don't converge so well

- normal is constant over each triangle, with discontinuous jumps across edges
- error is only $O(h)$

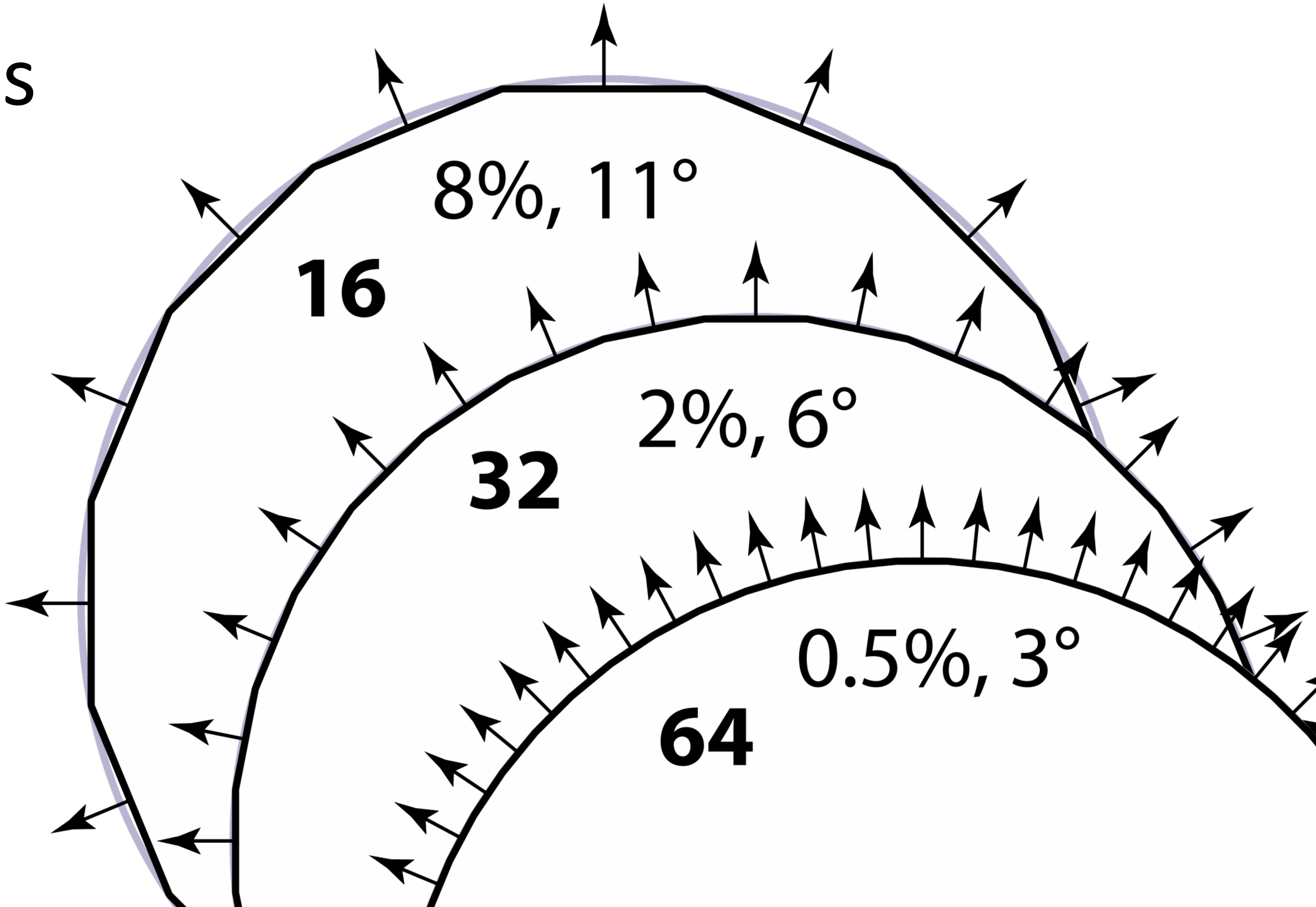
After a slide by Steve Marschner

Problems with face normals—2D example

Approximating circle with increasingly many segments

Max error in position error drops by factor of 4 each step

Max error in normal only drops by factor of 2



After a slide by Steve Marschner

Problems with face normals—solution

Piecewise planar approximation converges pretty quickly to the smooth geometry as the number of triangles increases

- for mathematicians: error is $O(h^2)$

But the surface normals don't converge so well

- normal is constant over each triangle, with discontinuous jumps across edges
- for mathematicians: error is only $O(h)$

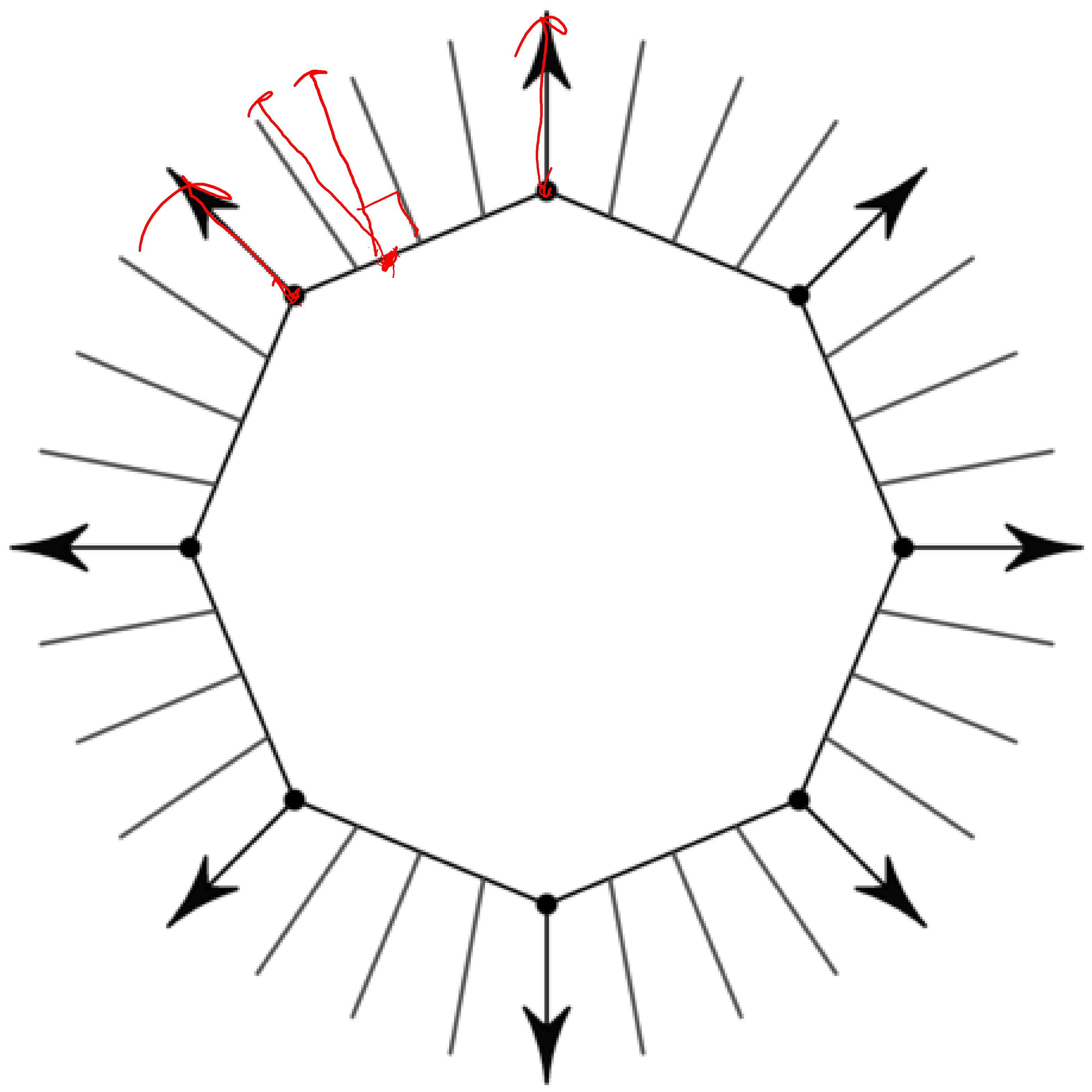
Better: store the "real" normal at each vertex, and *interpolate* to get normals that vary gradually across triangles

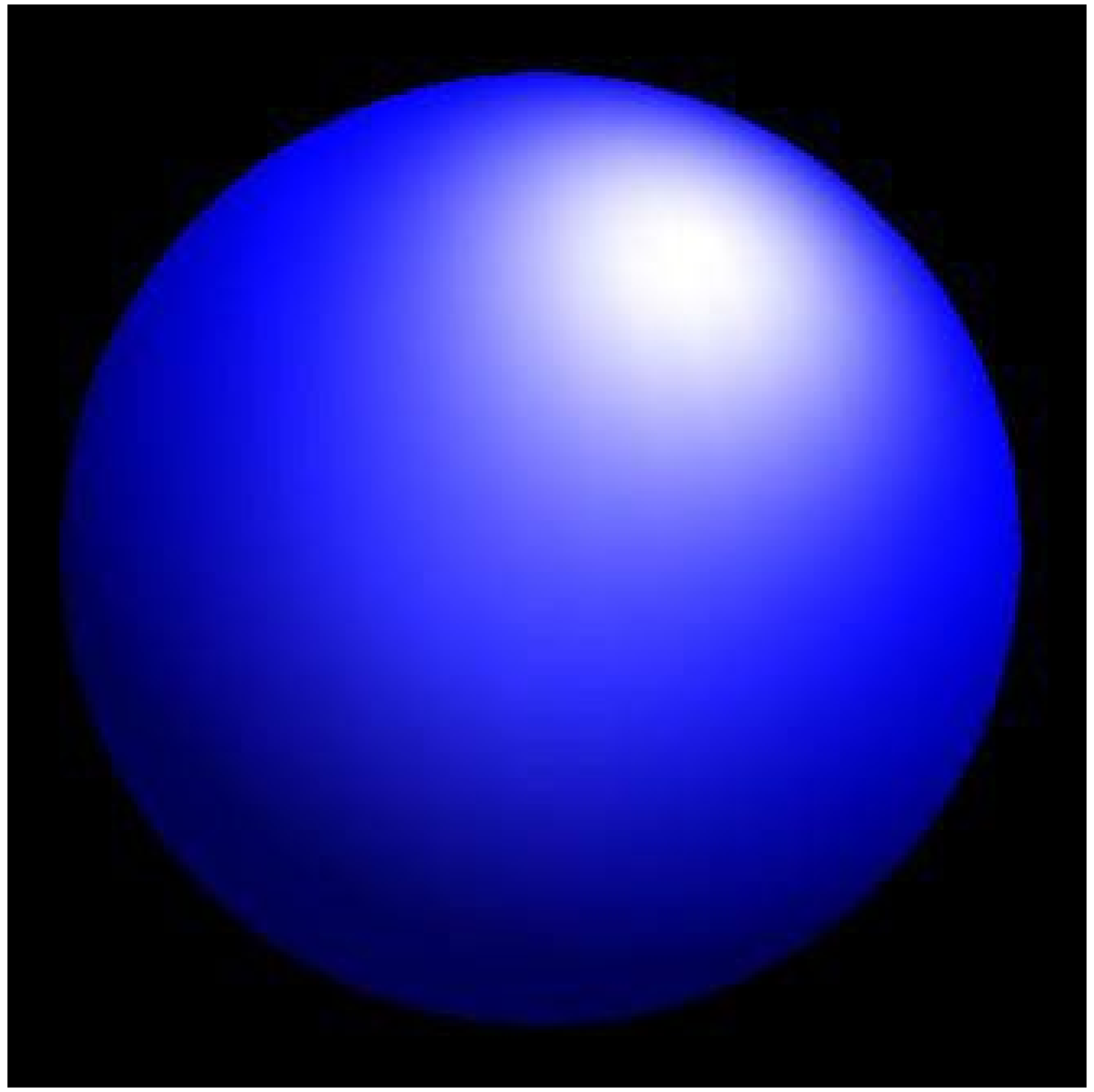
After a slide by Steve Marschner

Defining normals

Vertex normals: store normal at vertices, interpolate in face

- geometrically “inconsistent”, but smooth look

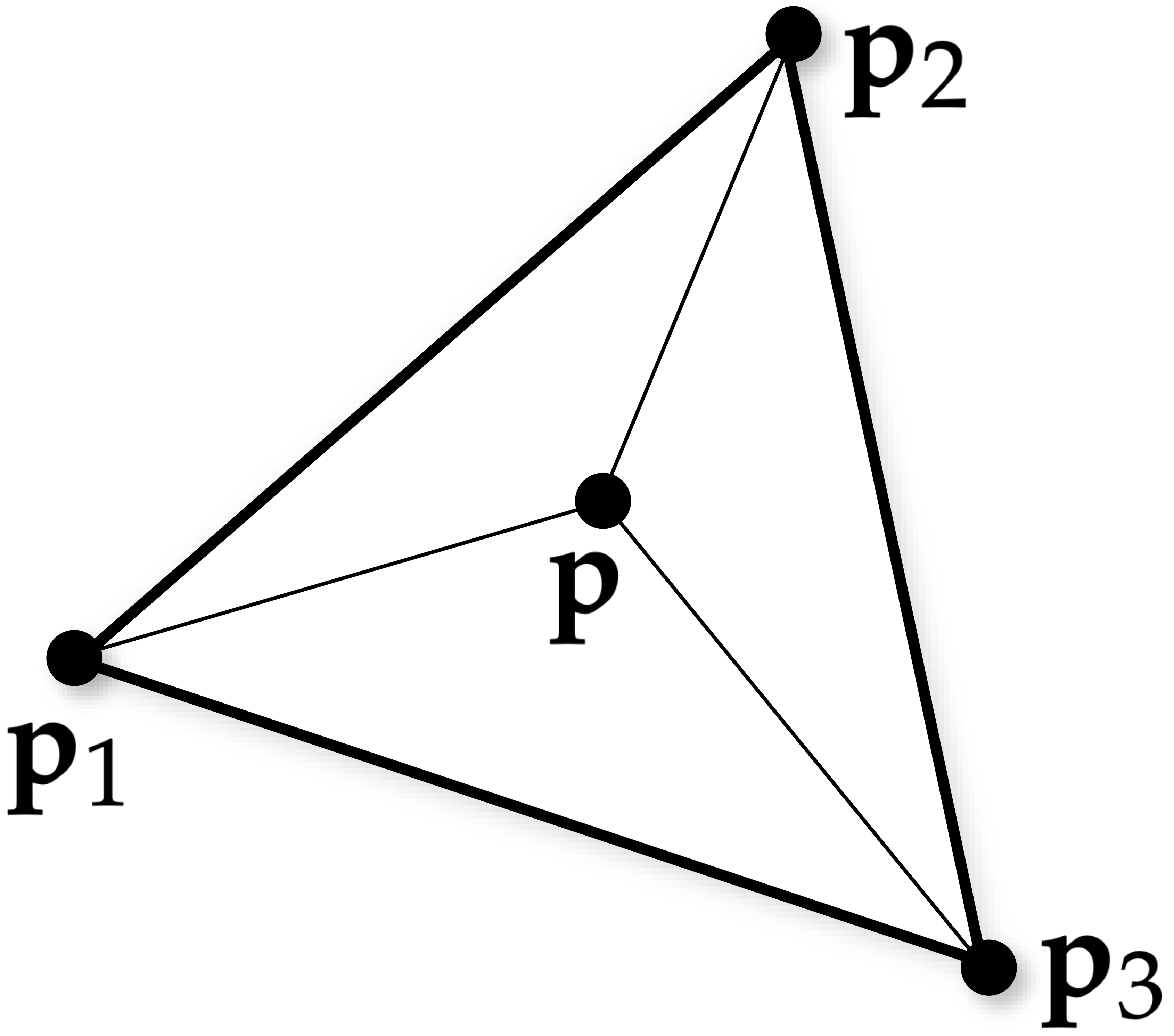




After a slide by Jon Denning

Barycentric coordinates

Barycentric interpolation: $$\mathbf{p}(\alpha, \beta, \gamma) = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$$



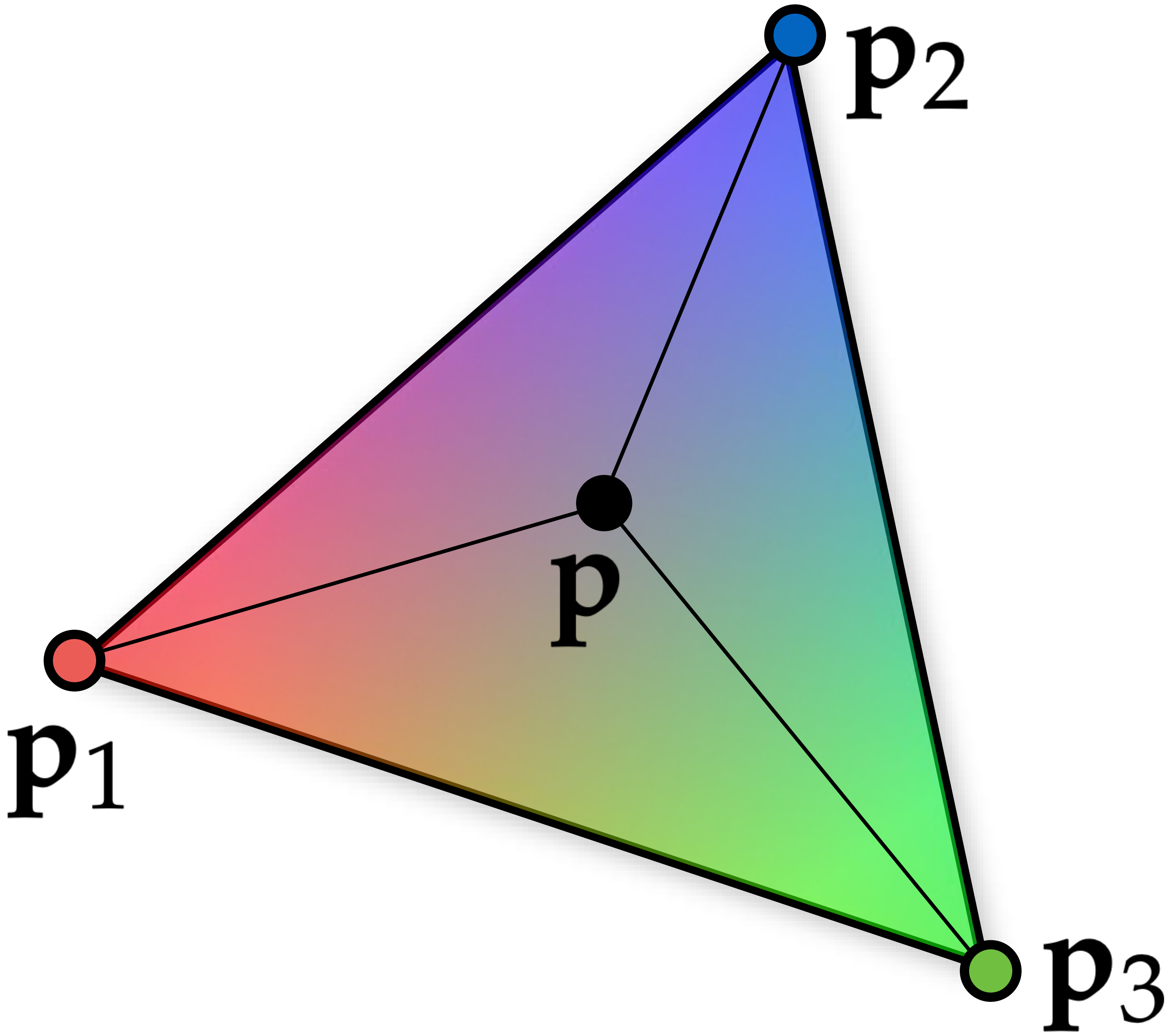
Can use this eqn. to interpolate any vertex quantity across triangle!

Barycentric coordinates

Barycentric interpolation:

$$\mathbf{p}(\alpha, \beta, \gamma) = \alpha \mathbf{p}_1 + \beta \mathbf{p}_2 + \gamma \mathbf{p}_3$$

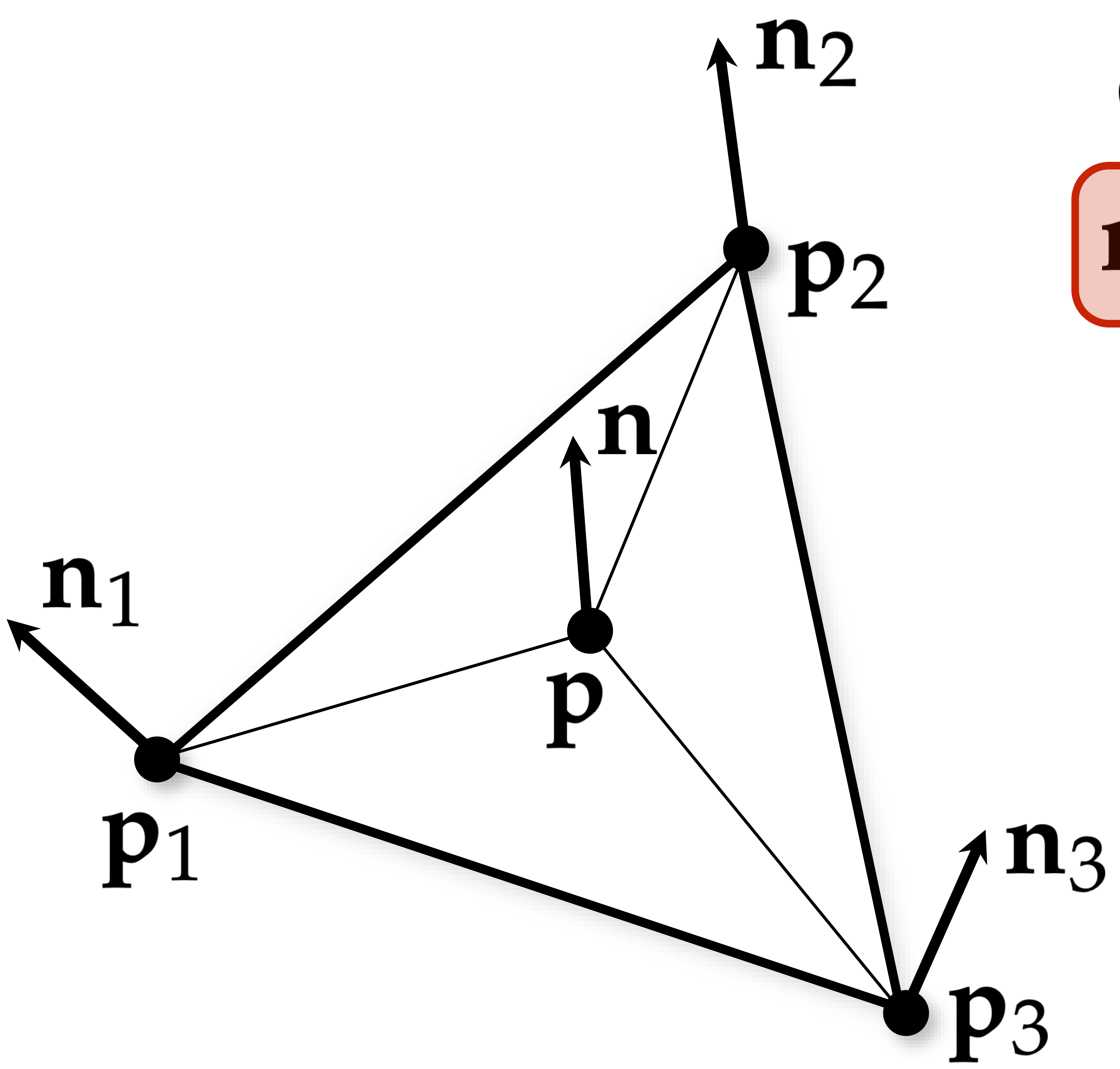
$$\mathbf{c}(\alpha, \beta, \gamma) = \alpha \mathbf{c}_1 + \beta \mathbf{c}_2 + \gamma \mathbf{c}_3$$



Can use this eqn. to interpolate any vertex quantity across triangle!

Barycentric coordinates

Barycentric interpolation:



$$\mathbf{p}(\alpha,\beta,\gamma) = \alpha\mathbf{p}_1 + \beta\mathbf{p}_2 + \gamma\mathbf{p}_3$$

$$\mathbf{c}(\alpha,\beta,\gamma) = \alpha\mathbf{c}_1 + \beta\mathbf{c}_2 + \gamma\mathbf{c}_3$$

$$\boxed{\mathbf{n}(\alpha,\beta,\gamma)} = \alpha\mathbf{n}_1 + \beta\mathbf{n}_2 + \gamma\mathbf{n}_3$$

not guaranteed to be unit length

Can use this eqn. to interpolate any vertex quantity across triangle!

Realism through geometric complexity

Andreas Byström

Ray Tracing Acceleration

Ray-surface intersection is at the core of every ray tracing algorithm

Brute force approach:

- intersect every ray with every primitive
- many unnecessary ray-surface intersection tests

Ray Tracing Cost

"the time required to compute the intersections of rays and surfaces is over 95 percent" [Whitted 1980]

Cost = $O(n_x \cdot n_y \cdot n_o)$

- (number of pixels) · (number of objects)
- Assumes 1 ray per pixel

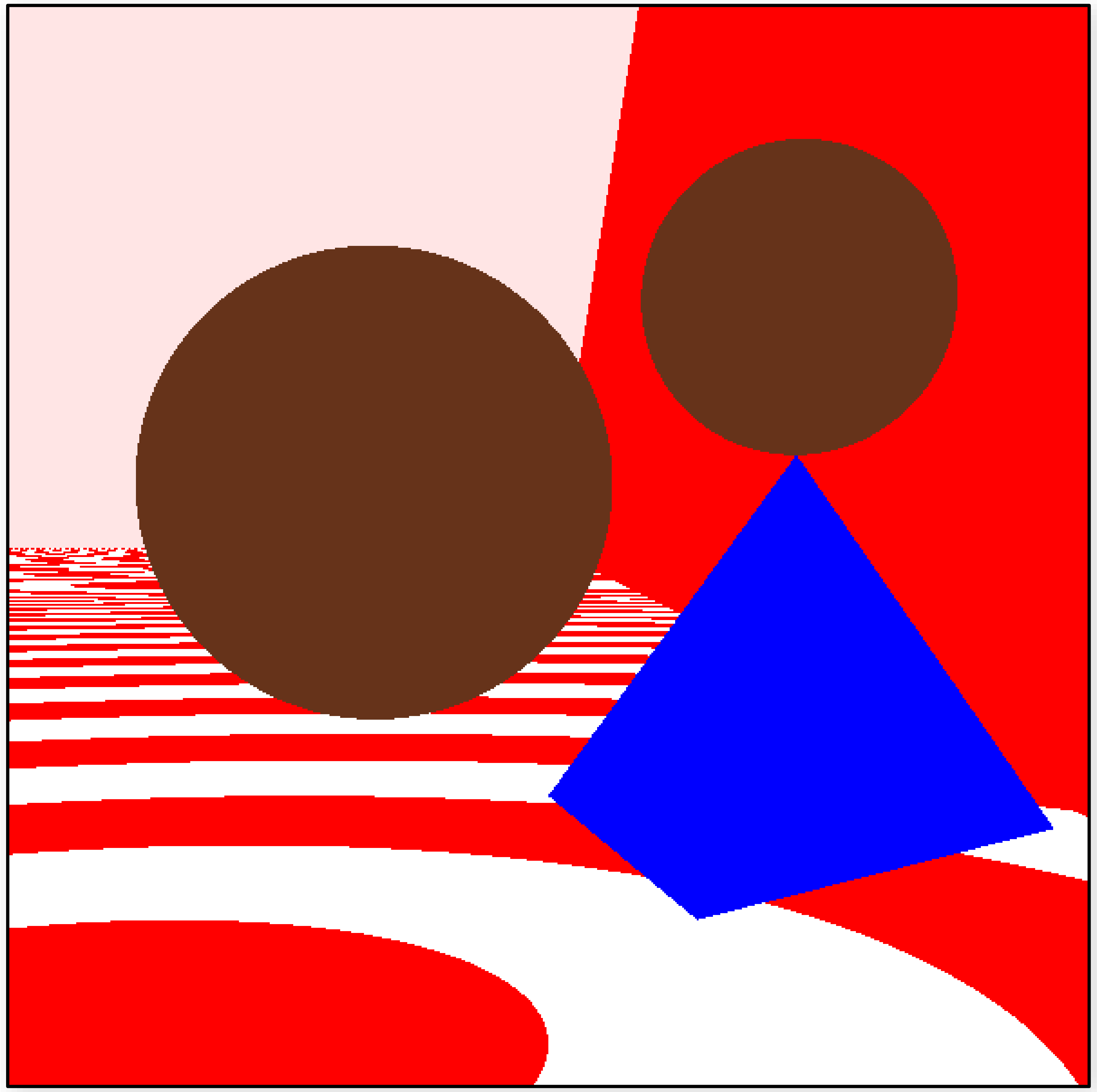
Example: 1024 x 1024 image of a scene with 1000 triangles

- Cost is (at least) 10^9 ray-triangle intersections

Typically measured per ray:

- Naive: $O(n_o)$ - linear with number of objects

$O(n_o)$ Ray Tracing (The Problem)



8 primitives → 3 seconds

50K trees each with 1M polygons = 50B polygons

→ **594 years!**

Sub-linear Ray Tracing

50K trees each with 1M polygons = 50B polygons → **11 minutes**

300,000,000x speedup!

The solution

Improve efficiency of ray-surface intersections by constructing **acceleration structures**.

- A spatial organization of objects in a scene to minimize the necessary number of ray-object intersection tests.

Spatial sorting/subdivision (e.g. grid, kd-tree, ochre)

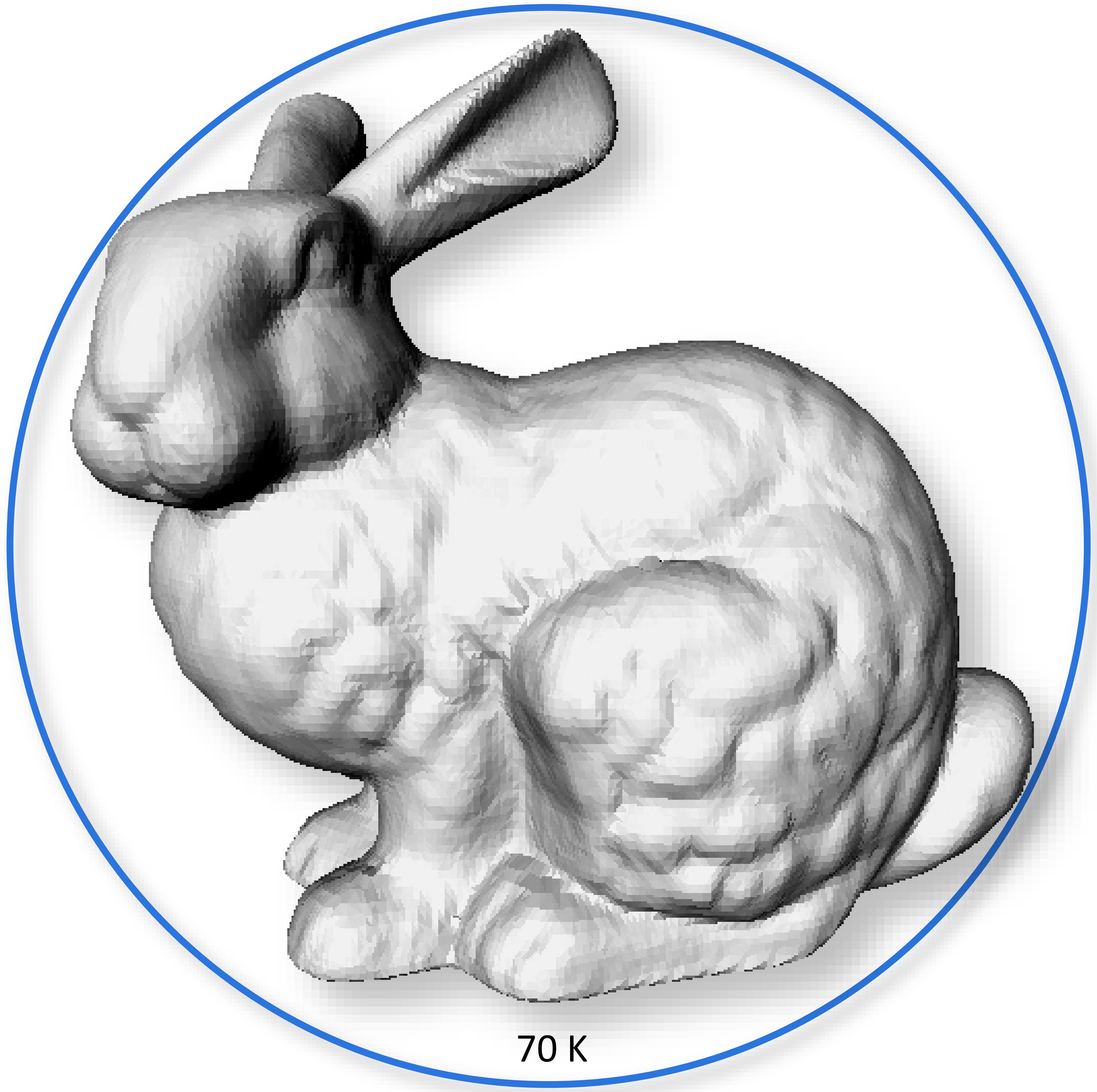
- Decompose **space** into disjoint **regions** & assign objects to regions

Object sorting/subdivision (bounding volume hierarchy)

- Decompose **objects** into disjoint **sets** & bound using simple volumes for fast rejection

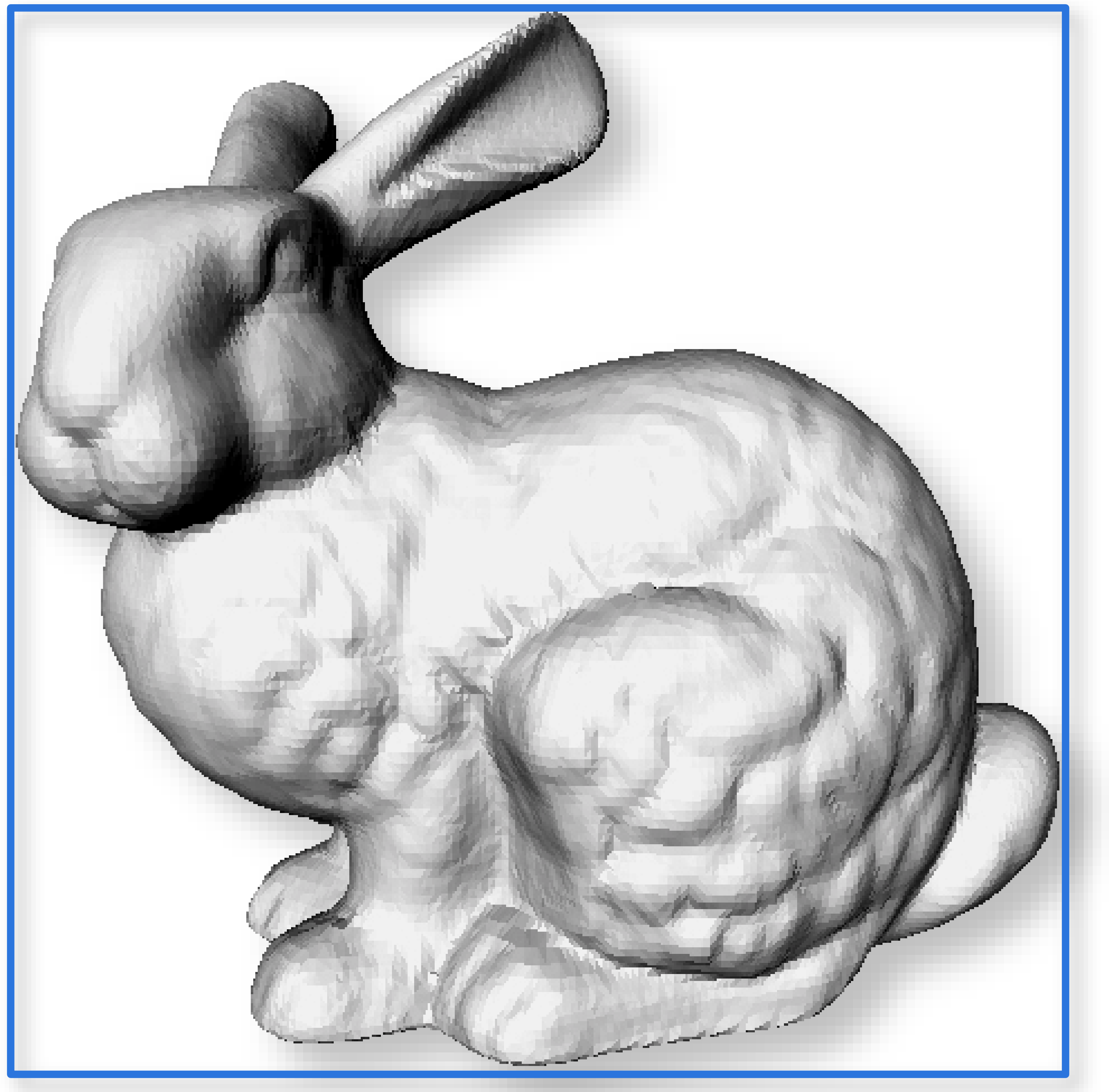
Bounding Volumes

Spheres



Bounding Volumes

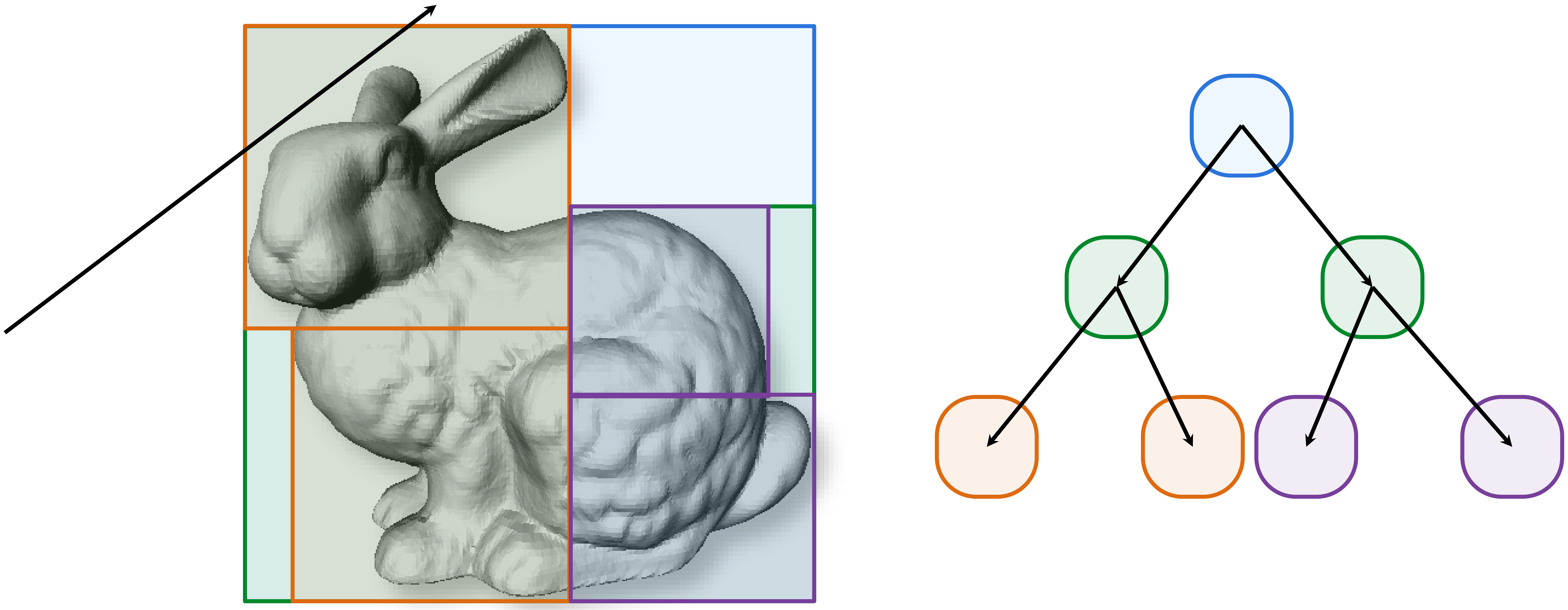
Axis-aligned bounding boxes (most common)



70 K

Bounding Volumes Hierarchies

Now do this hierarchically!



BVH Traversal

```
void BVHNode::intersectBVH(ray, &hit):
    if (bound.hit(ray)):
        if (leaf):
            leaf.intersect(ray, hit);
        else:
            leftChild.intersectBVH(ray, hit);
            rightChild.intersectBVH(ray, hit);
```

Constructing BVHs

Top-down:

- partition objects along an axis and create two sub-sets

Bottom-up:

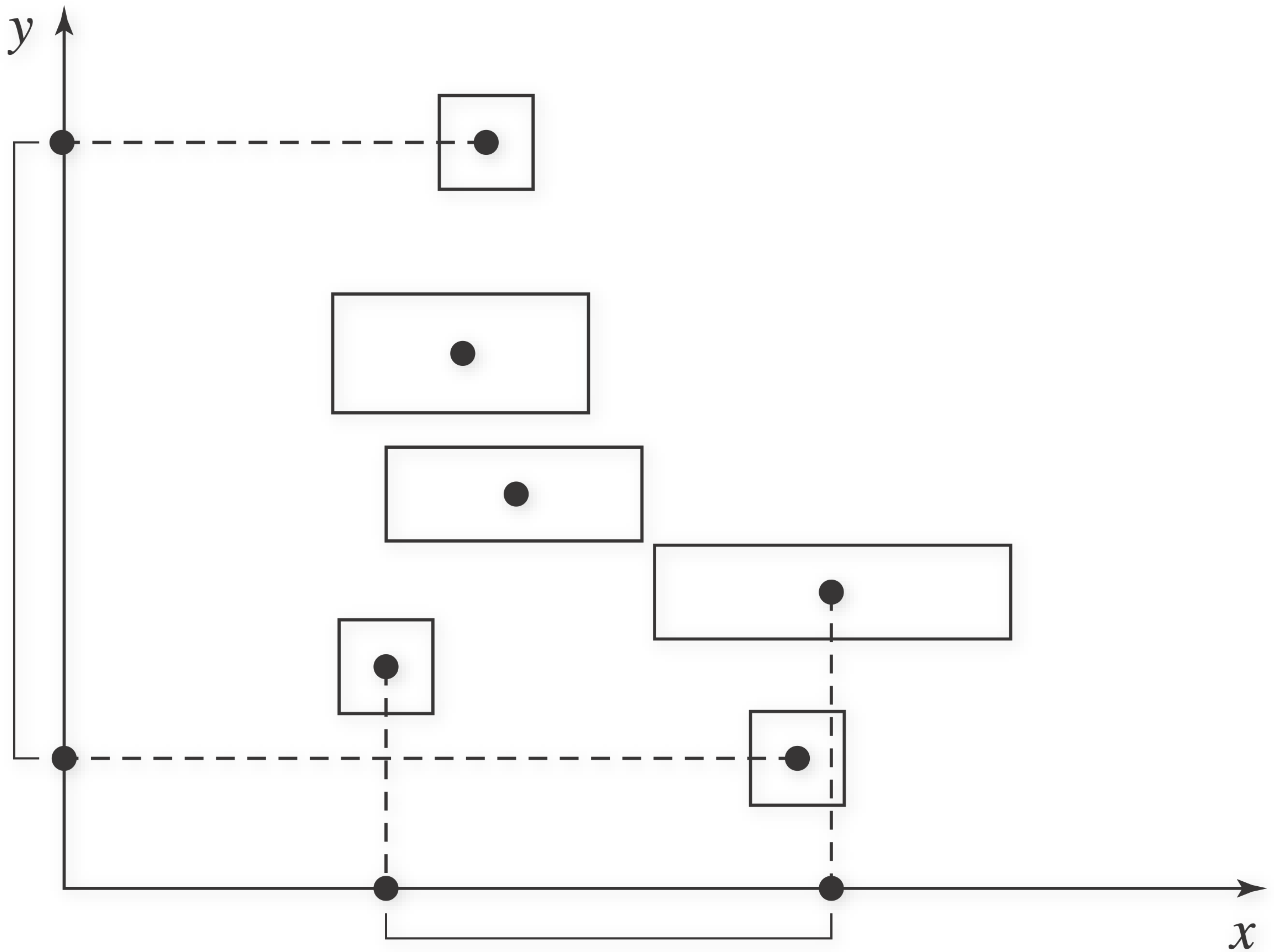
- recursively group nearby objects together

Divisive (top-down) BBH construction

1. Choose split axis
2. Choose split plane location
3. Choose whether to create leaf or split + repeat

Many strategies for each of these steps

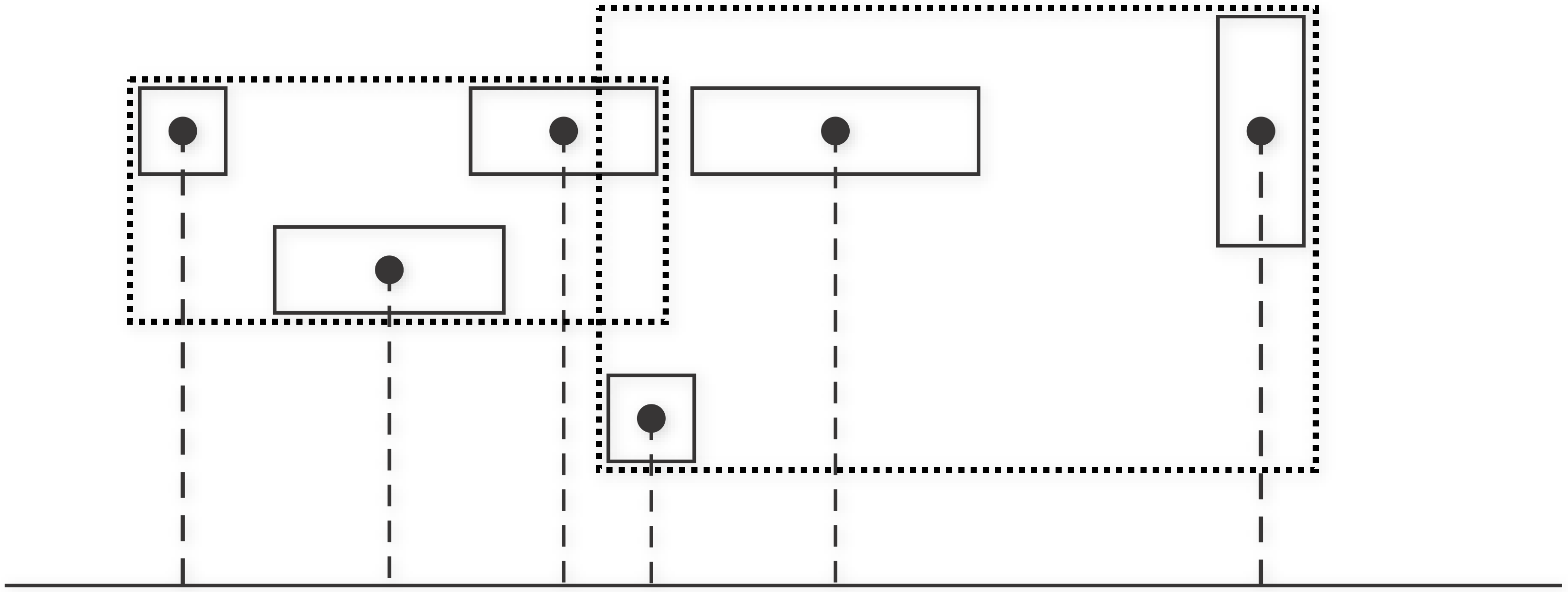
Choosing axis based on centroid extents



PBRe2 fig. 4.8

Object-median splitting

1. Sort bbox centroids along split axis
2. Take take first half as left child, second half as right



PBRe2 fig. 4.11