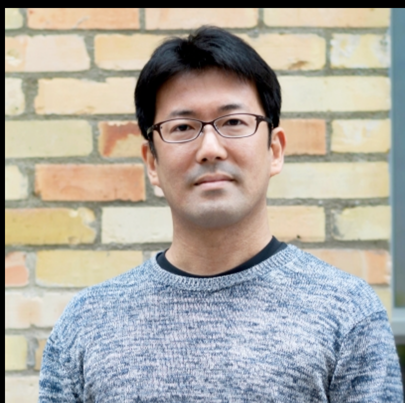
Introduction to Computer Graphics

Authored by:
Toshiya Hachisuka

Presented by:
Kenshi Takayama

About the slides author



UNIVERSITY OF WATERLOO

- Toshiya Hachisuka（蜂須賀　恵也）
  first last
- Used to teach this class (2015~2020)
- Now Assoc. Prof. @UWaterloo
  - https://cs.uwaterloo.ca/~thachisu/

Today

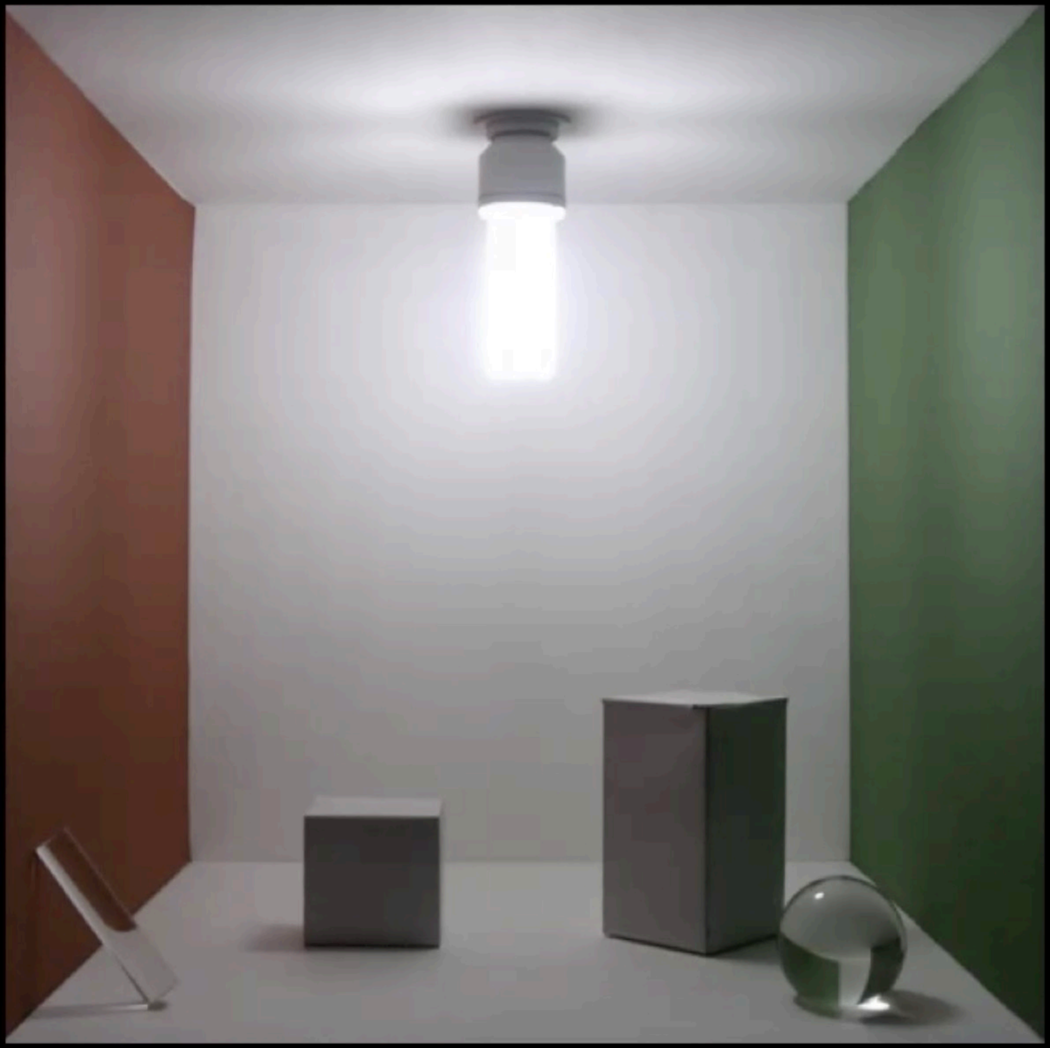
- Introduction to ray tracing
- Basic ray-object intersection


LuxRender

LuxRender

“Turing Test” - Cornell Box

Simulated



Measured

How can we generate realistic images?

Rendering

Input data

Light sources

Shapes

Materials

Camera data

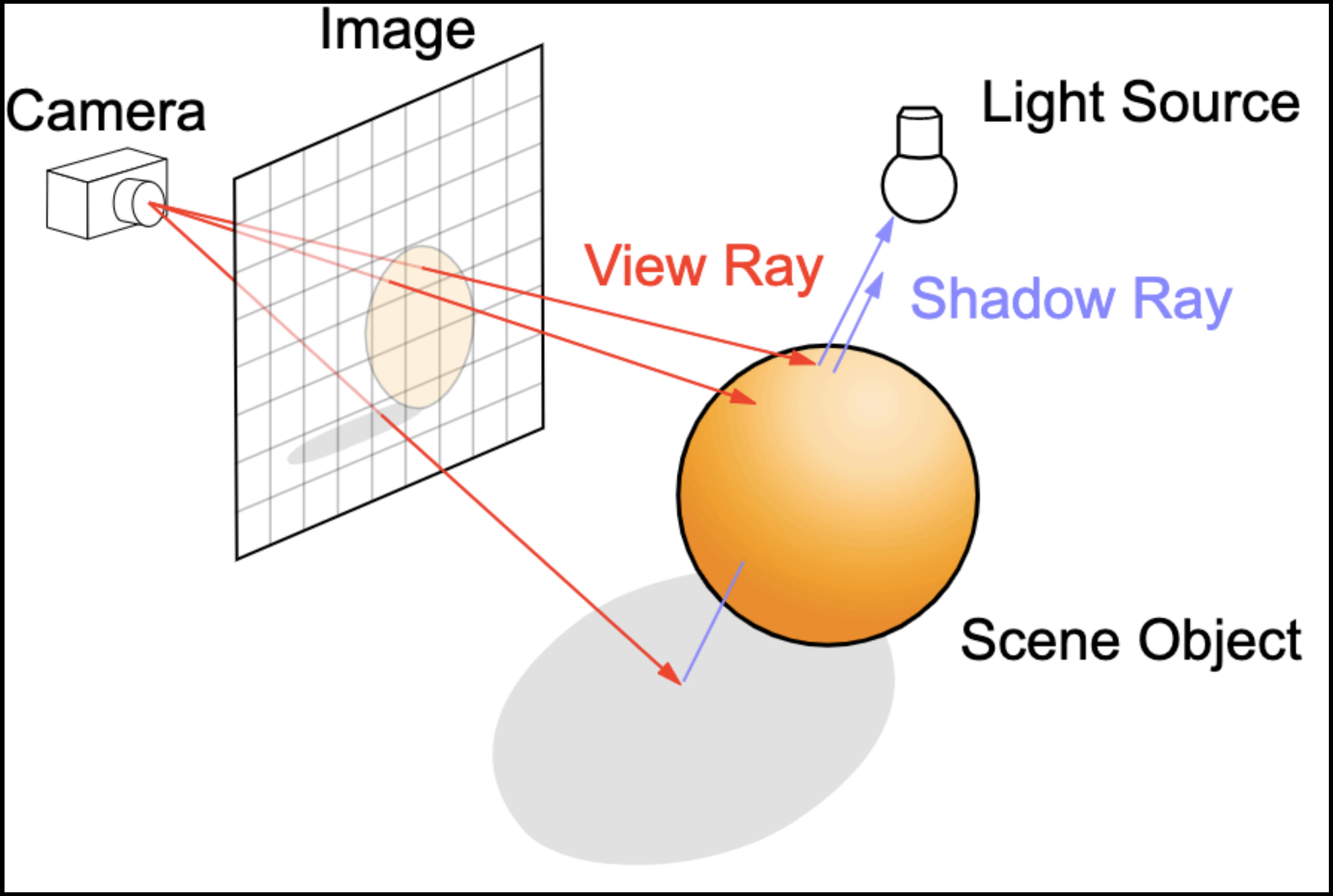
→ Computation →

Image

Interdisciplinary Nature

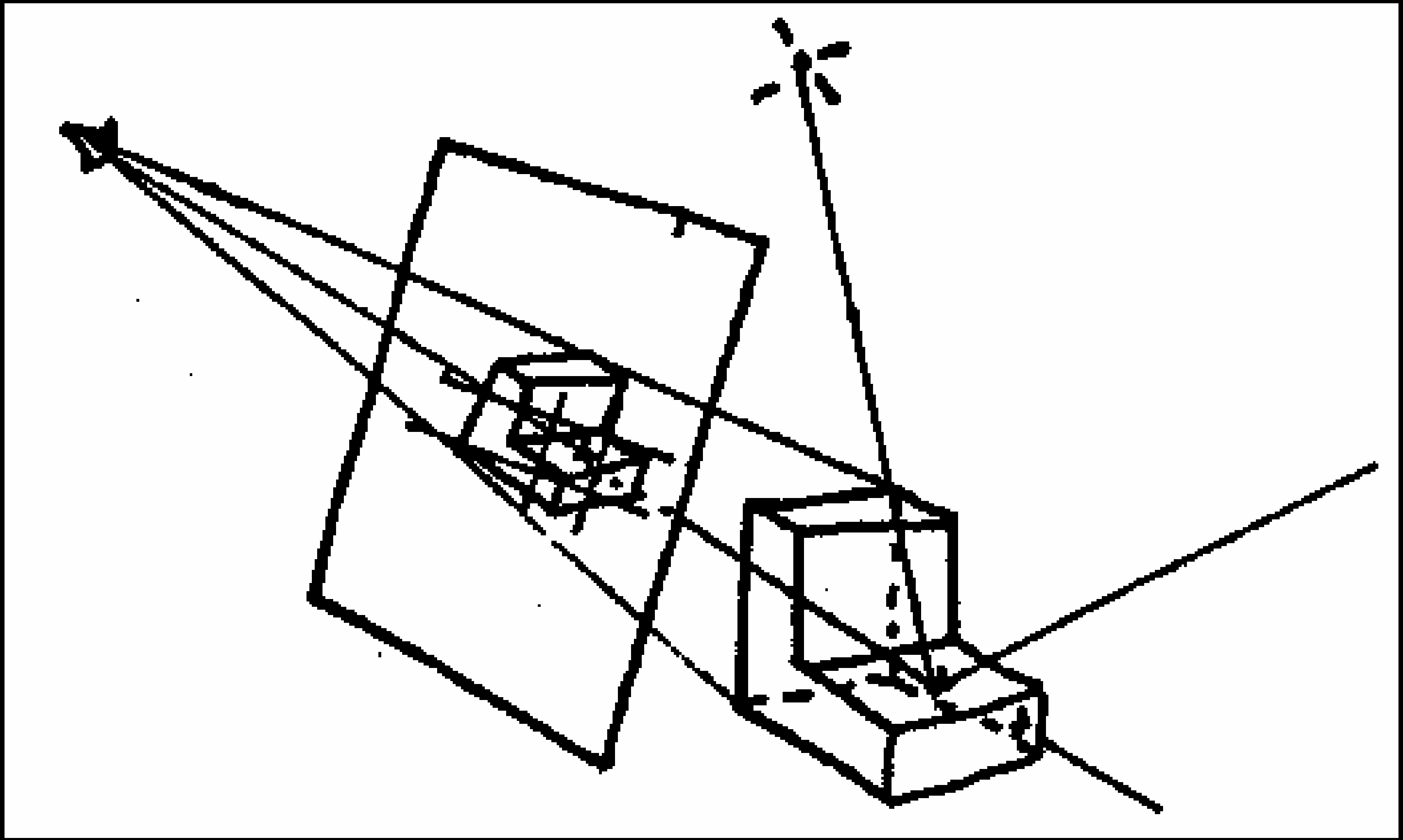
- Computer Science
  - Algorithms
  - Computational geometry
  - Software engineering
- Physics
  - Radiometry
  - Optics
- Mathematics
  - Algebra
  - Calculus
  - Statistics
- Perception
- Art

Ray Tracing - Concept



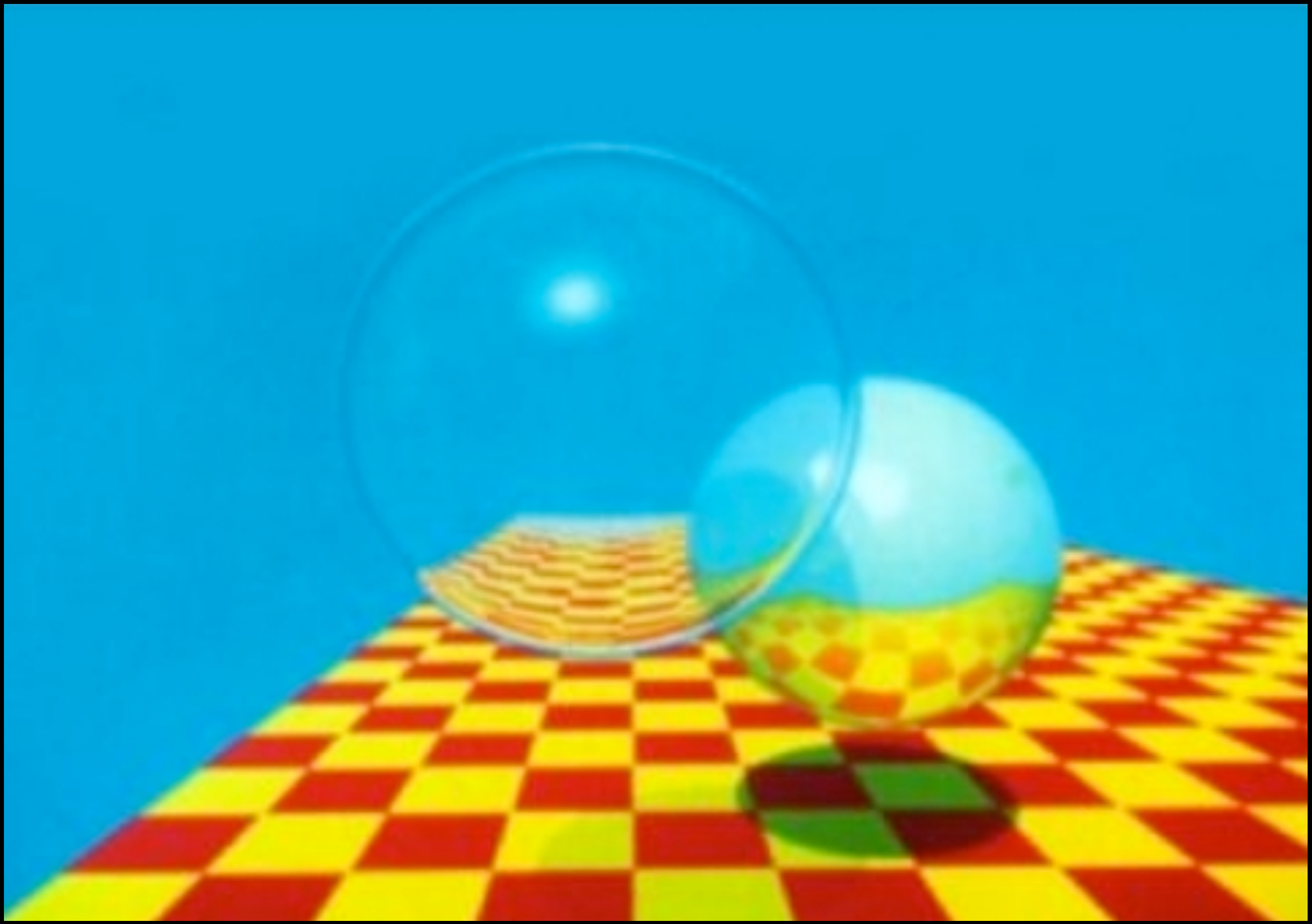
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)#/media/File:Ray_trace_diagram.svg

Ray Tracing [Appel 1968]



Generate images with shadows using ray tracing

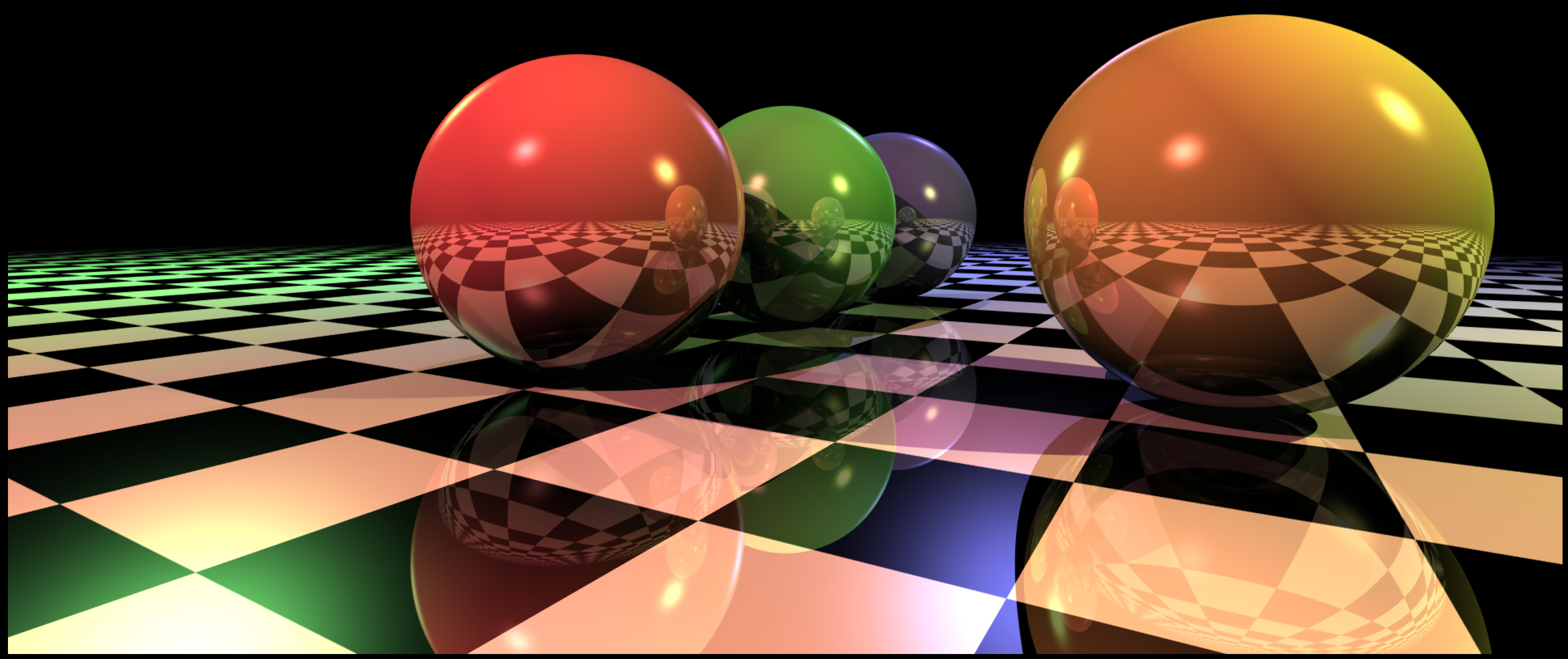
Ray Tracing [Whitted 1979]



Recursive ray tracing for reflections/refractions

Whitted Ray Tracing Today

- Runs realtime on a GPU!



http://alexrodgers.co.uk

Whitted Ray Tracing Today

- Runs realtime on a GPU!

You are going to implement something like this!

http://alexrodgers.co.uk

Ray Tracing - Pseudocode

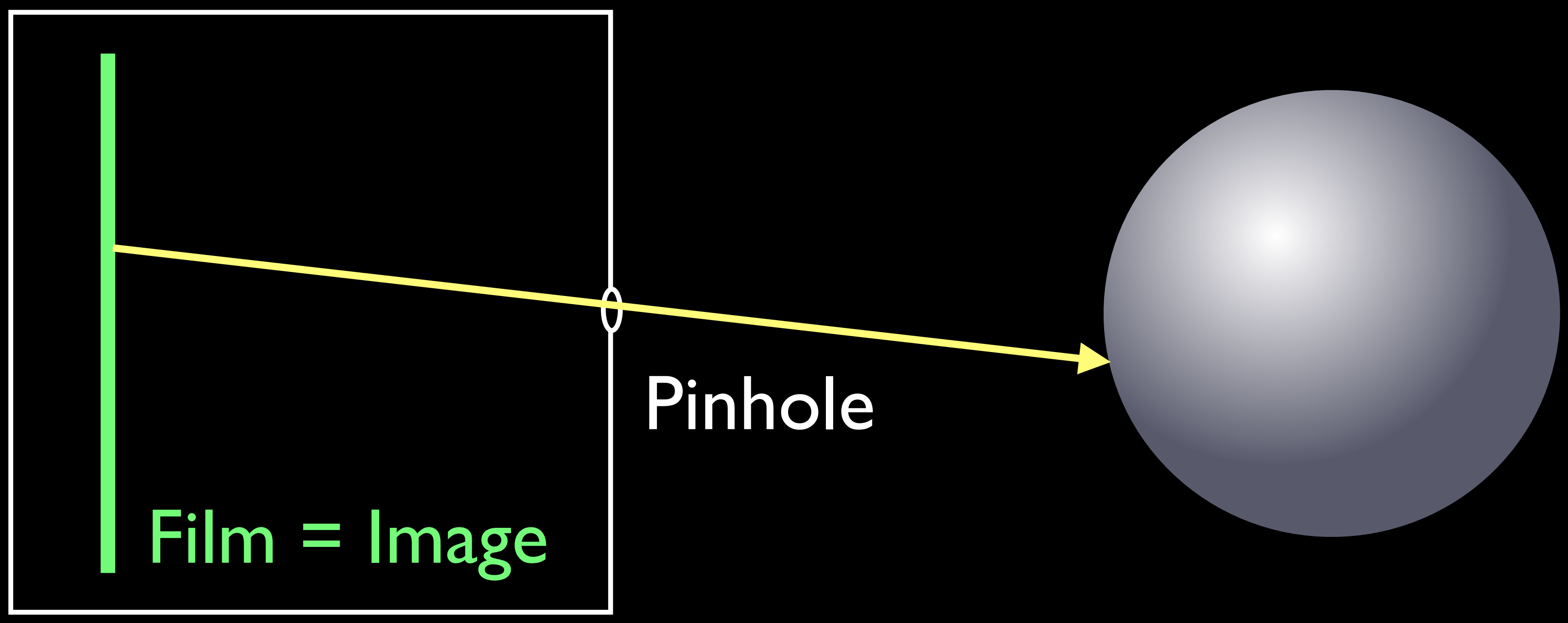
```
for all pixels {
    ray = generate_camera_ray( pixel )
    for all objects {
        hit = intersect( ray, object )
        if "hit" is closer than "first_hit" {first_hit = hit}
    }
    pixel = shade( first_hit )
}
```

Ray Tracing - Data Structures

```
class object {
    bool intersect( ray )
}

class ray {
    vector origin
    vector direction
}
```

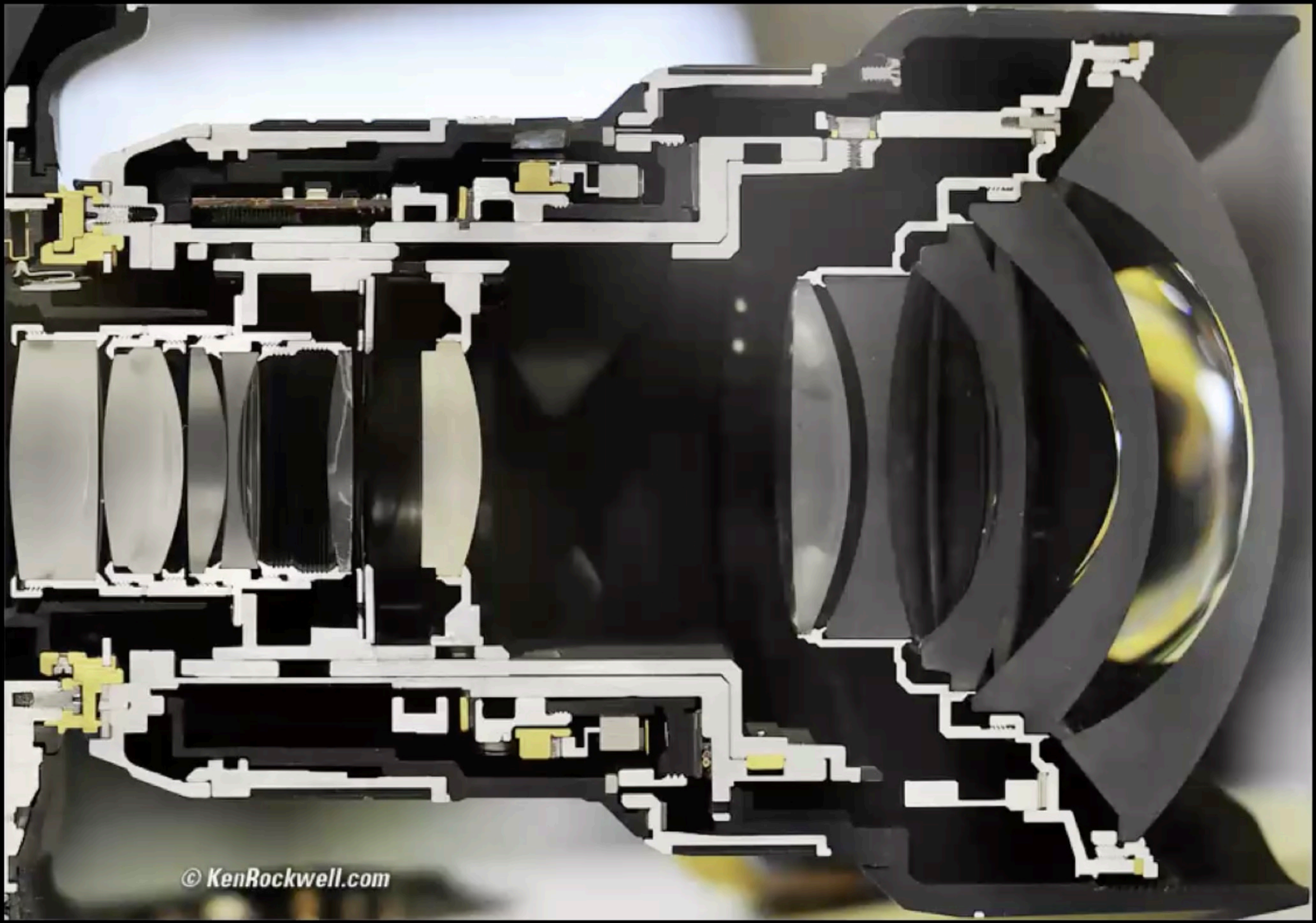
Pinhole Camera



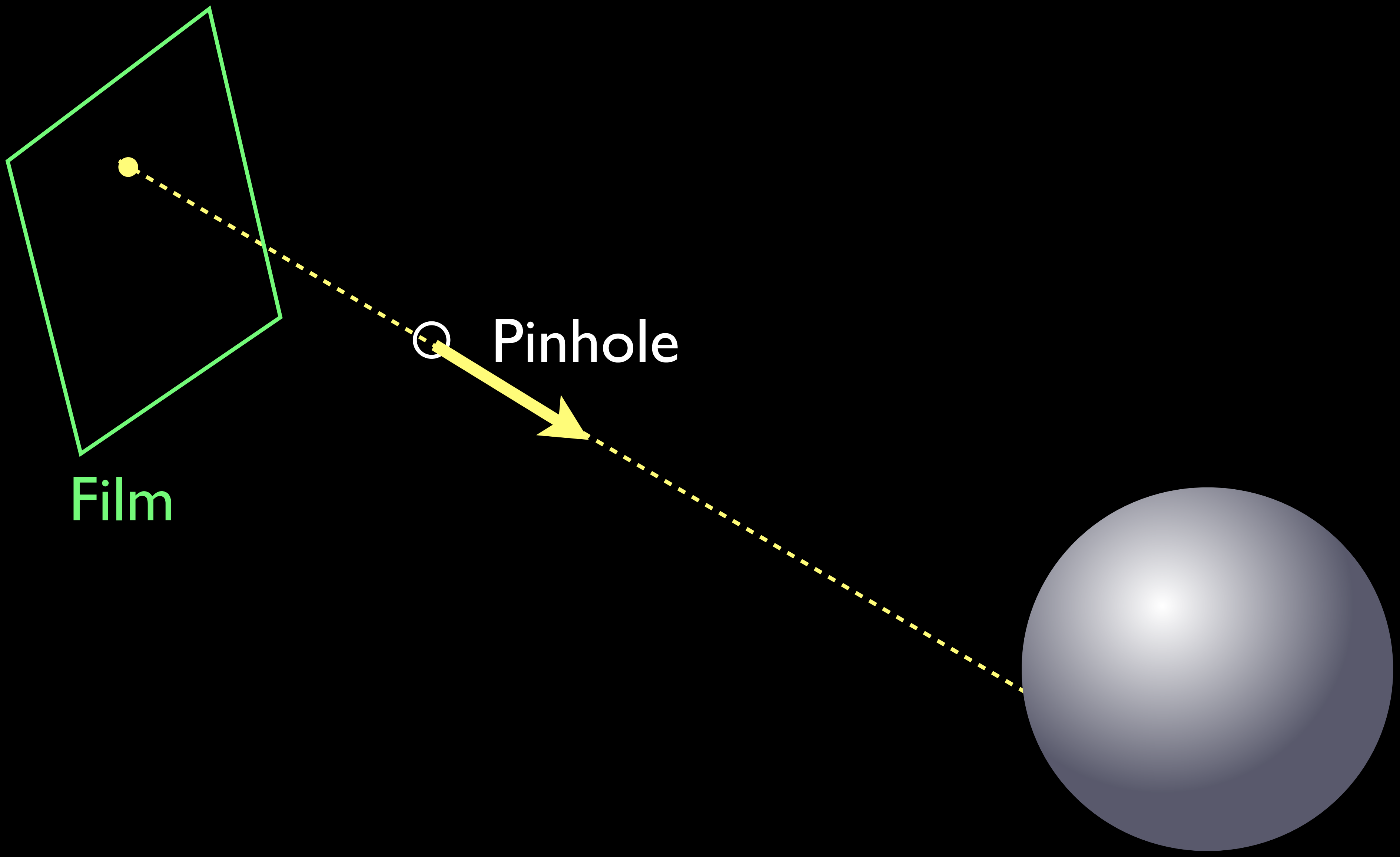
(the image is flipped)

Pinhole Camera

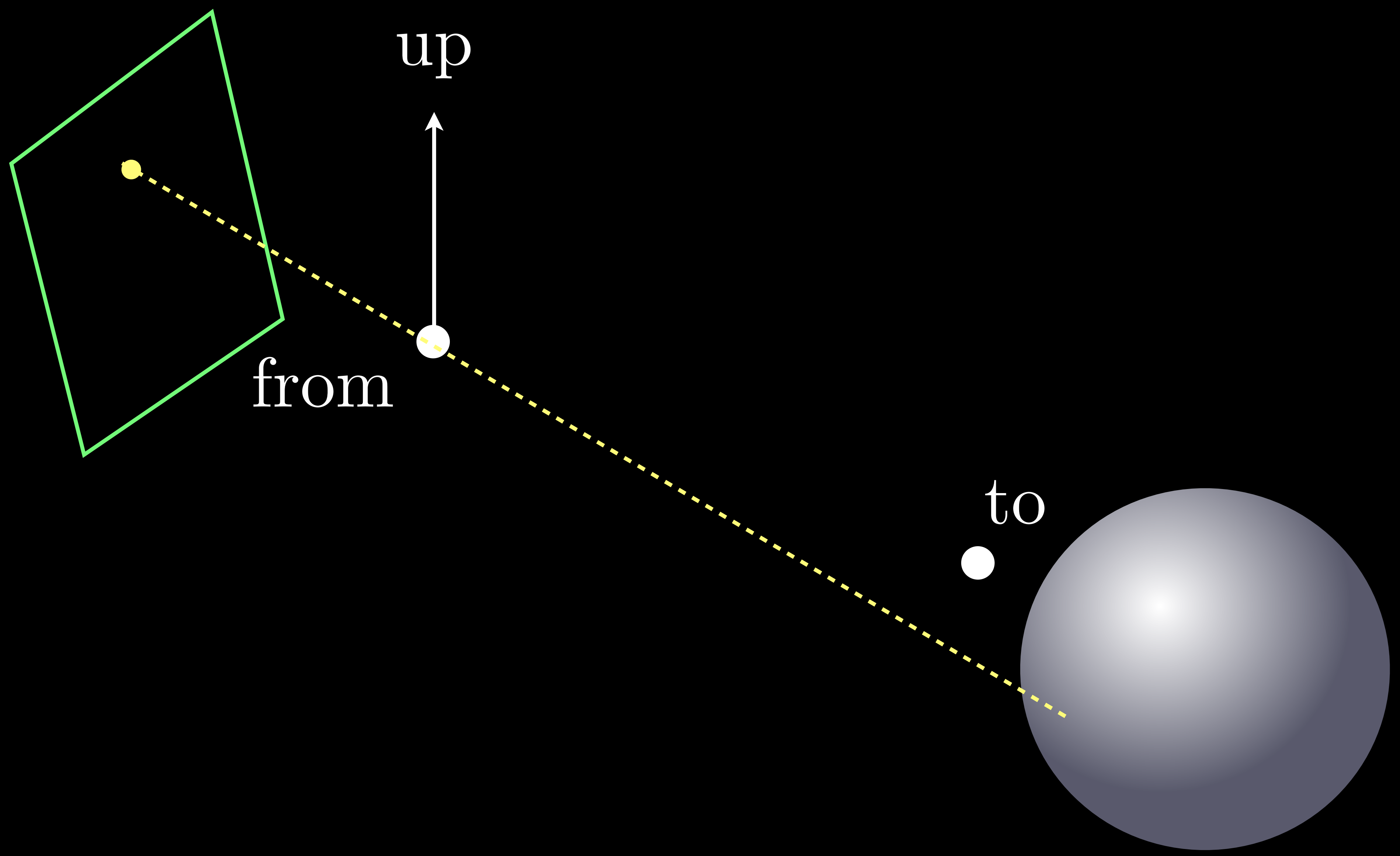
Modern Camera



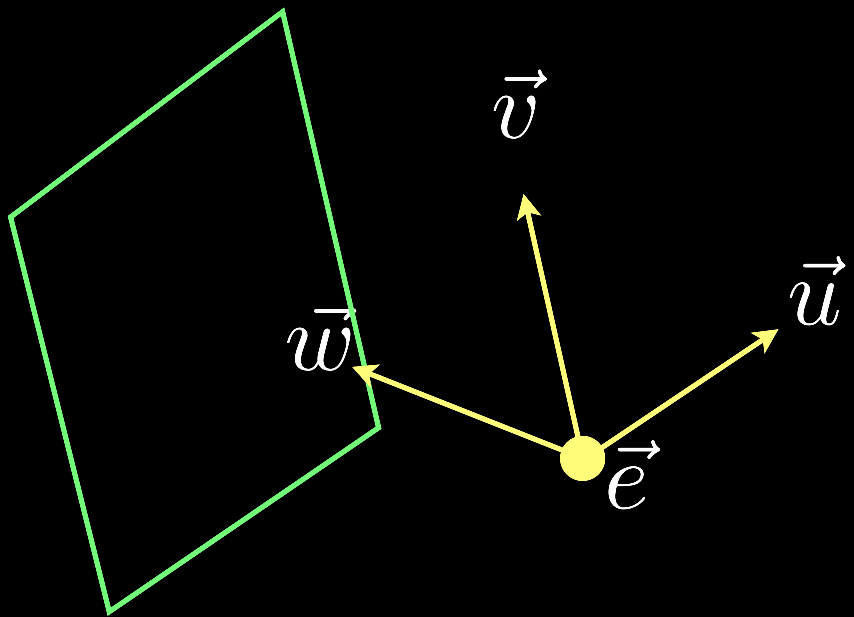
Pinhole Camera

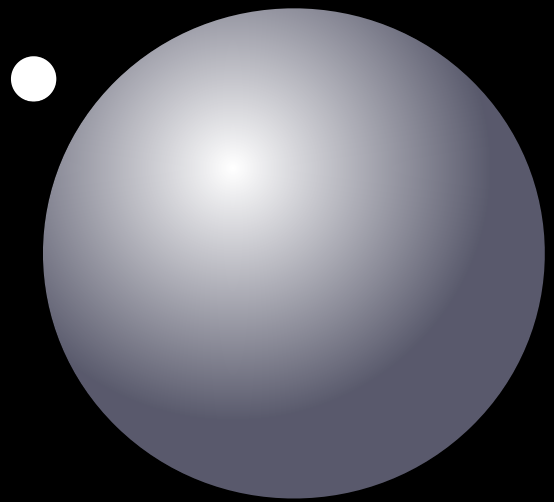


Camera Coordinate System



Camera Coordinate System





Orthonormal basis

$$\boxed{\vec{u}\cdot\vec{v}=\vec{v}\cdot\vec{w}=\vec{w}\cdot\vec{u}=0}$$

$$\boxed{||\vec{u}||=||\vec{v}||=||\vec{w}||=1}$$

Camera Coordinate System

- Given $\vec{C}_{\text{up}}, \vec{C}_{\text{from}}$, and $\vec{C}_{\text{to}}$

$$\vec{w} = \frac{\vec{C}_{\text{from}} - \vec{C}_{\text{to}}}{||\vec{C}_{\text{from}} - \vec{C}_{\text{to}}||}$$

$$\vec{u} = \frac{\vec{C}_{\text{up}} \times \vec{w}}{||\vec{C}_{\text{up}} \times \vec{w}||}$$

$$\vec{v} = \vec{w} \times \vec{u}$$

Axes

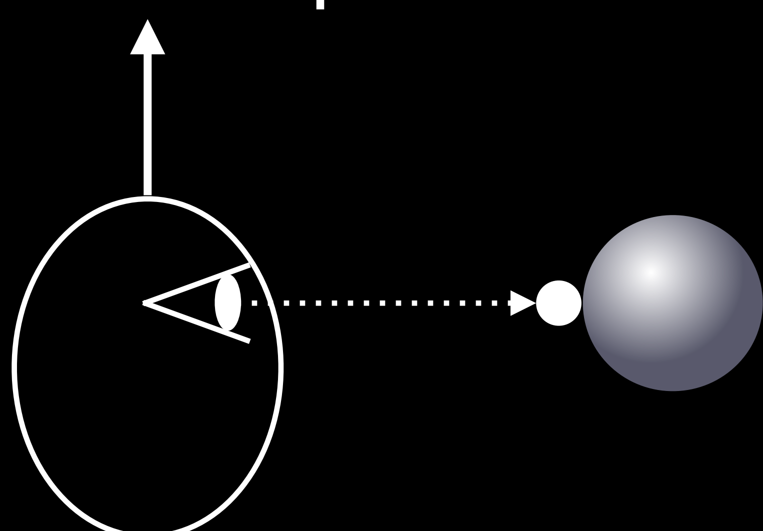
$$\vec{e} = \vec{C}_{\text{from}}$$

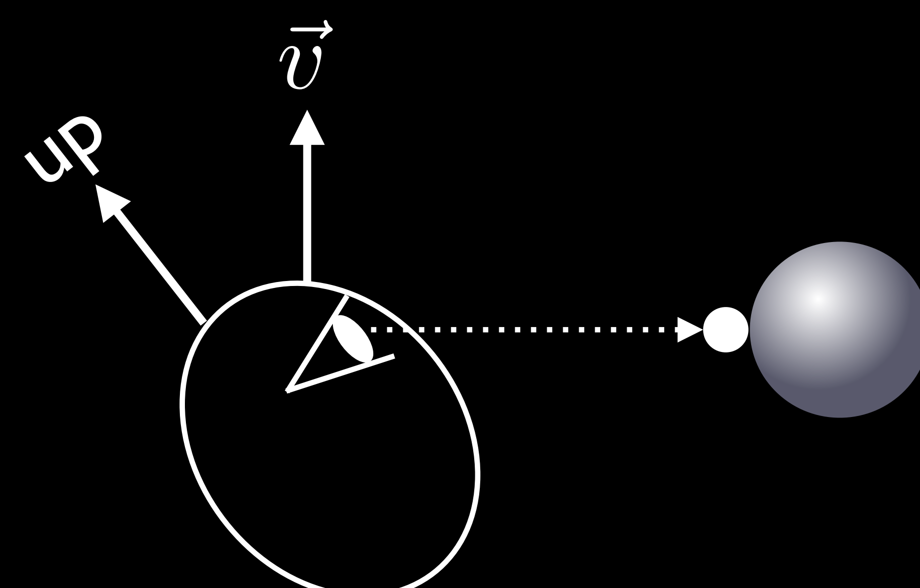
Origin

Up vector?

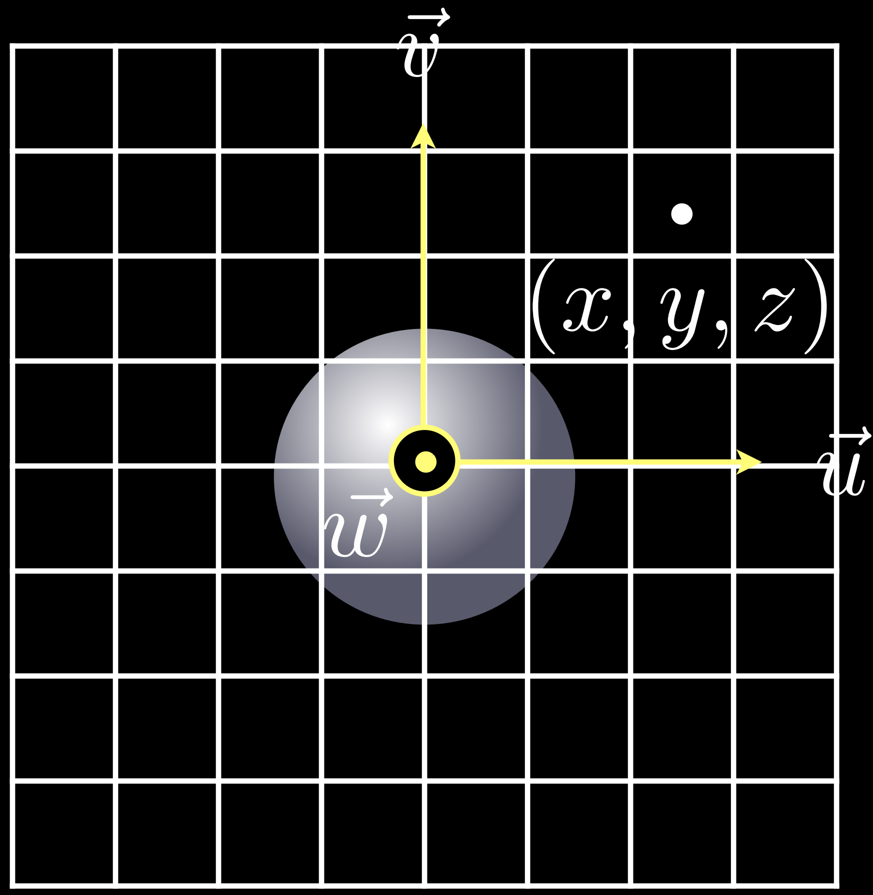
- Imagine a stick on top your head
  - The stick = up vector
  - Up vector is not always equal to $\vec{v}$

$\vec{v}$ = up





Generating a Camera Ray



$$x = \text{film\_w}\,\frac{\text{pixel\_i} + 0.5}{\text{res\_x}}$$

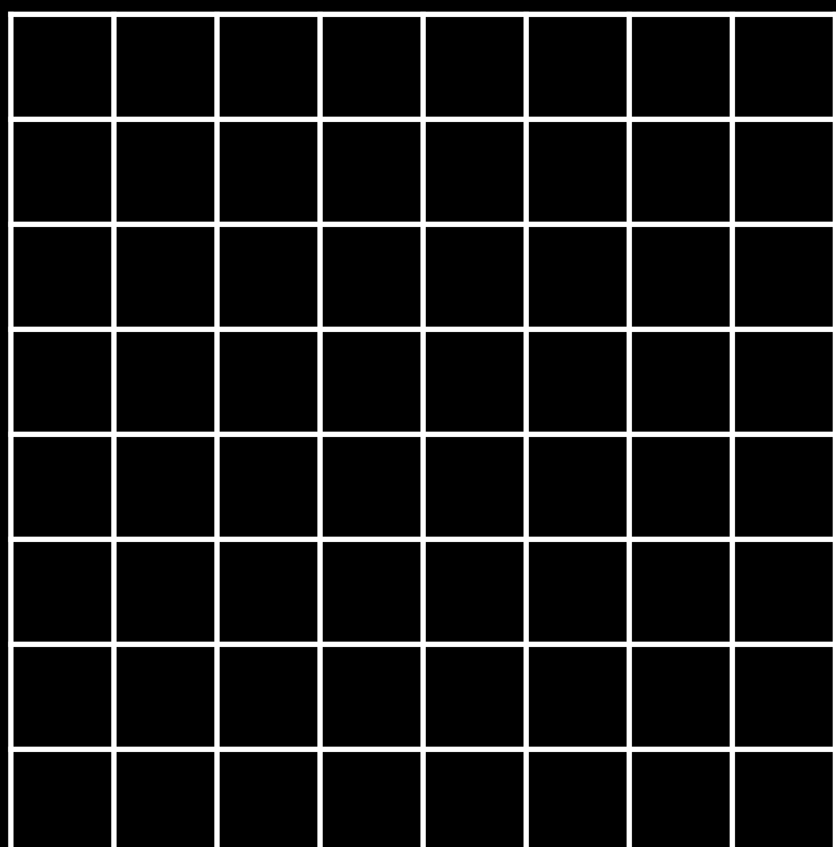
$$y = \text{film\_h}\,\frac{\text{pixel\_j} + 0.5}{\text{res\_y}}$$

$$z = \text{distance to film}$$

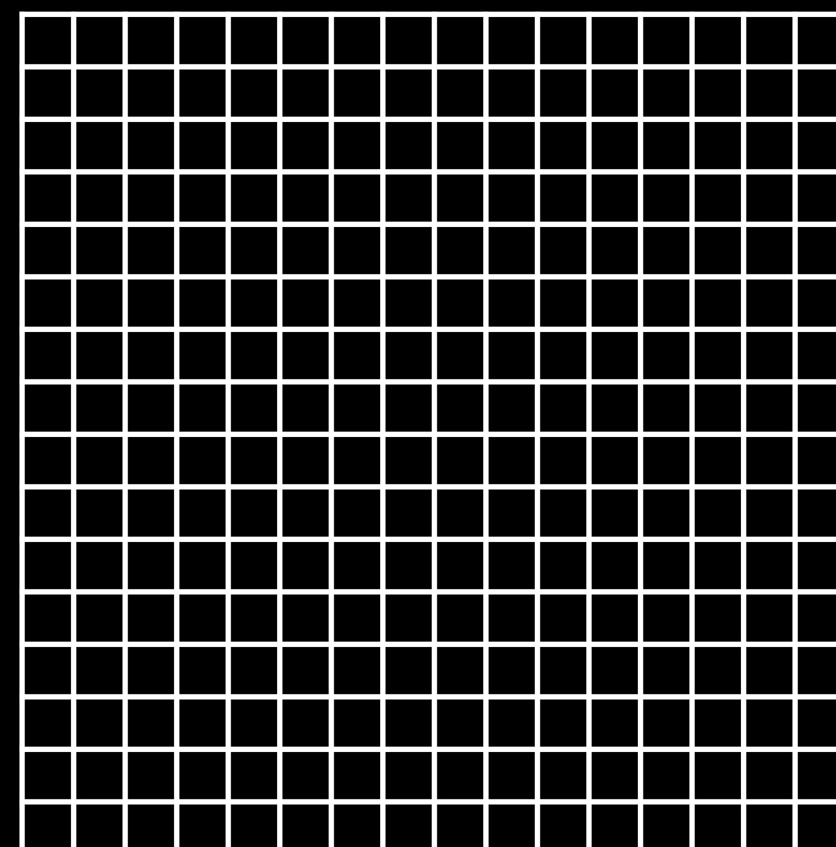
Pixel location in the camera coordinates

Generating a Camera Ray

- Film size is not equal to image resolution!



Film with 8^2 resolution



Same film with 16^2 resolution

Generating a Camera Ray

- Pixel location in the world coordinates:

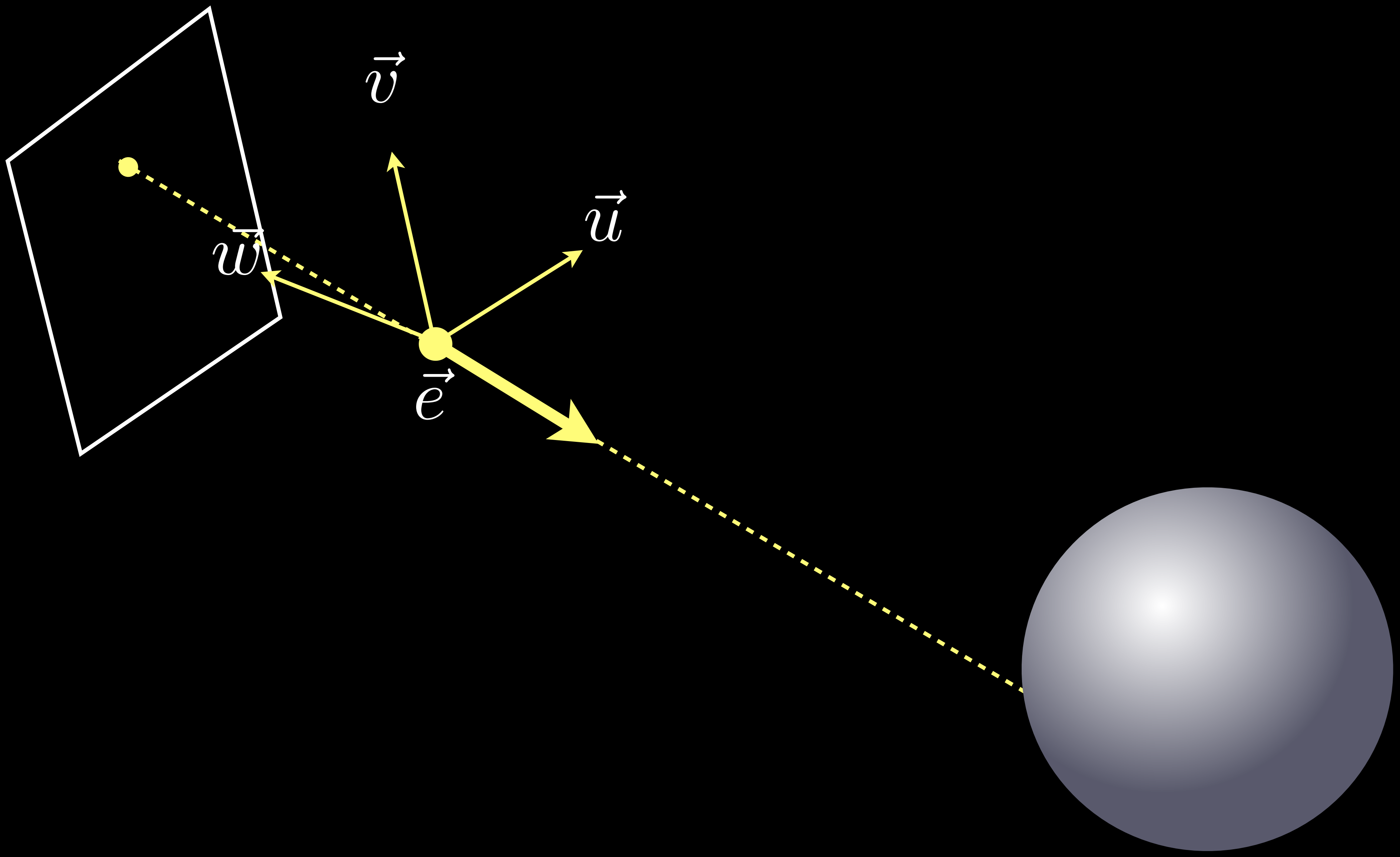
$$\text{pixel} = x\vec{u} + y\vec{v} + z\vec{w} + \vec{e}$$

- Camera ray in the world coordinates:

$$\text{origin} = \vec{e}$$

$$\text{direction} = \frac{\text{origin} - \text{pixel}}{||\text{origin} - \text{pixel}||}$$

Generating a Camera Ray



More Realistic Cameras

[Kolb et al. 1995]

- "A realistic camera model for computer graphics"
  - Ray tracing with actual lens geometry
  - Distortion

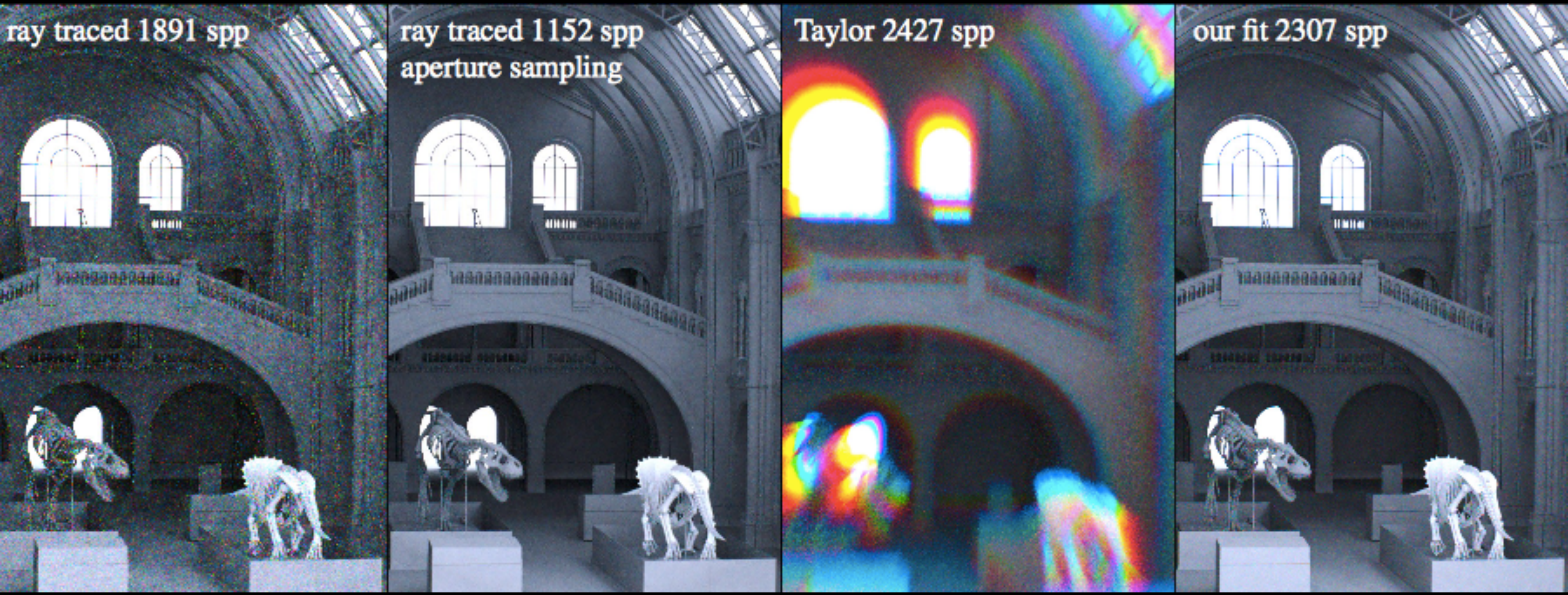
Full Simulation

Thin Lens Approximation

More Realistic Cameras

[Hanika et al. 2014]

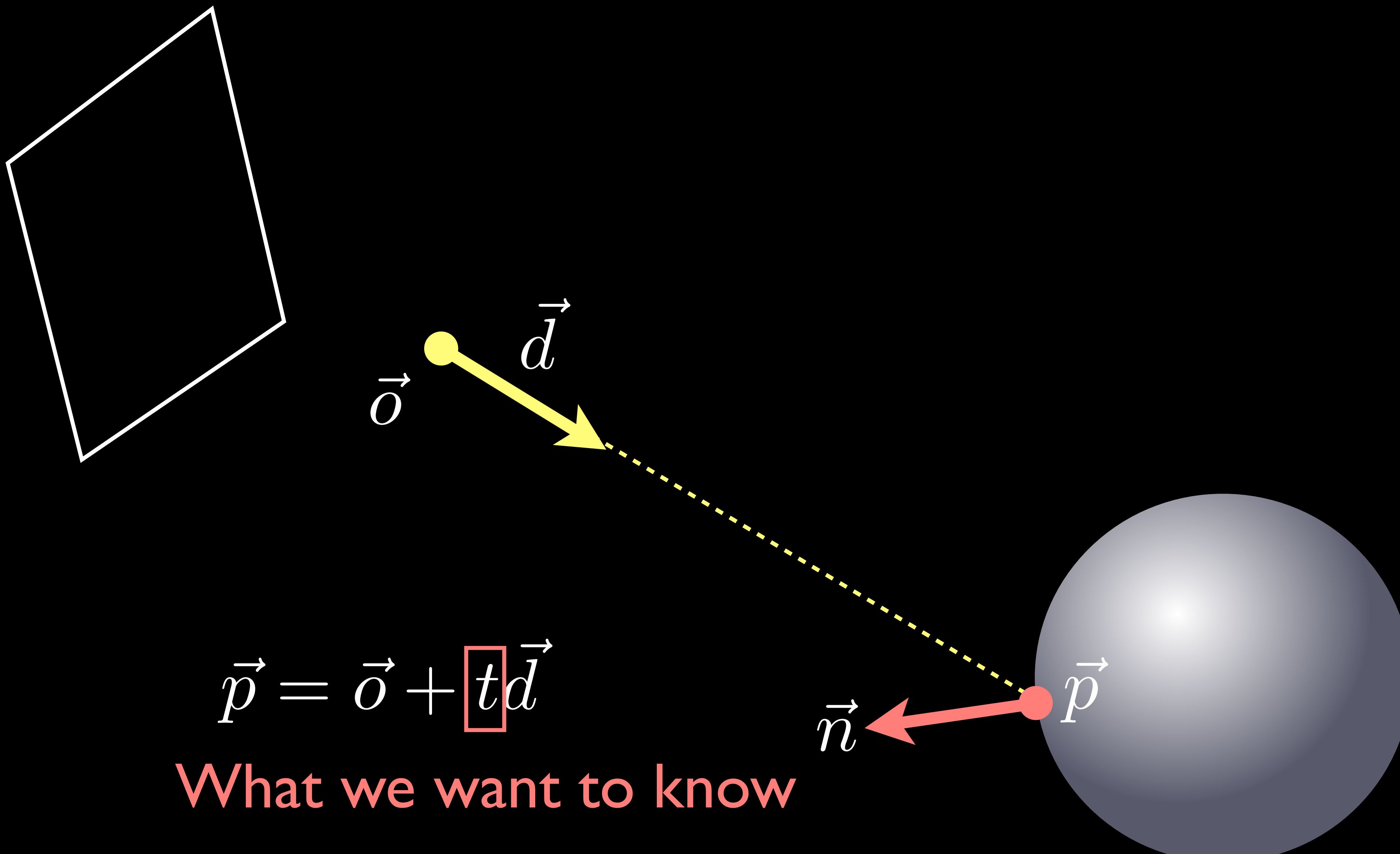
- "Efficient Monte Carlo Rendering with Realistic Lenses"
  - Polynomial approximation of a lens system



Ray Tracing - Pseudocode

```
for all pixels {
    ray = generate_camera_ray( pixel )
    for all objects {
        hit = intersect( ray, object )
        if "hit" is closer than "first_hit" {first_hit = hit}
    }
    pixel = shade( first_hit )
}
```

Goal



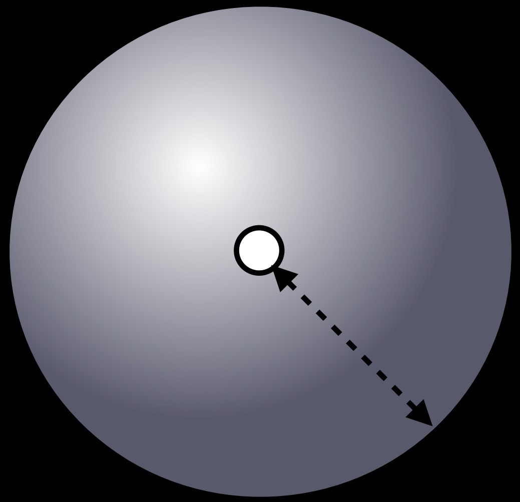
$$\vec{p} = \vec{o} + \boxed{t}\vec{d}$$

What we want to know

Ray-Sphere Intersection

- Sphere with center $\vec{c} = (c_x, c_y, c_z)$ and radius r

$$||(\vec{p} - \vec{c})||^2 = r^2$$



Ray-Sphere Intersection

- Sphere with center $\vec{c} = (c_x, c_y, c_z)$ and radius r

$$||(\boxed{\vec{p}} - \vec{c})||^2 = r^2$$

Substitute $\vec{p} = \vec{o} + t\vec{d}$

Ray-Sphere Intersection

- Sphere with center $\vec{c} = (c_x, c_y, c_z)$ and radius r

$$||(\vec{p} - \vec{c})||^2 = r^2 \qquad \boxed{||\vec{v}||^2 = \vec{v} \cdot \vec{v}}$$

Substitute $\vec{p} = \vec{o} + t\vec{d}$

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

Ray-Sphere Intersection

- Sphere with center $\vec{c} = (c_x, c_y, c_z)$ and radius r

$$||(\vec{p} - \vec{c})||^2 = r^2 \qquad ||\vec{v}||^2 = \vec{v} \cdot \vec{v}$$

Substitute $\vec{p} = \vec{o} + t\vec{d}$

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

$$\vec{d} \cdot \vec{d}t^2 + 2\vec{d} \cdot (\vec{o} - \vec{c})t + (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2 = 0$$

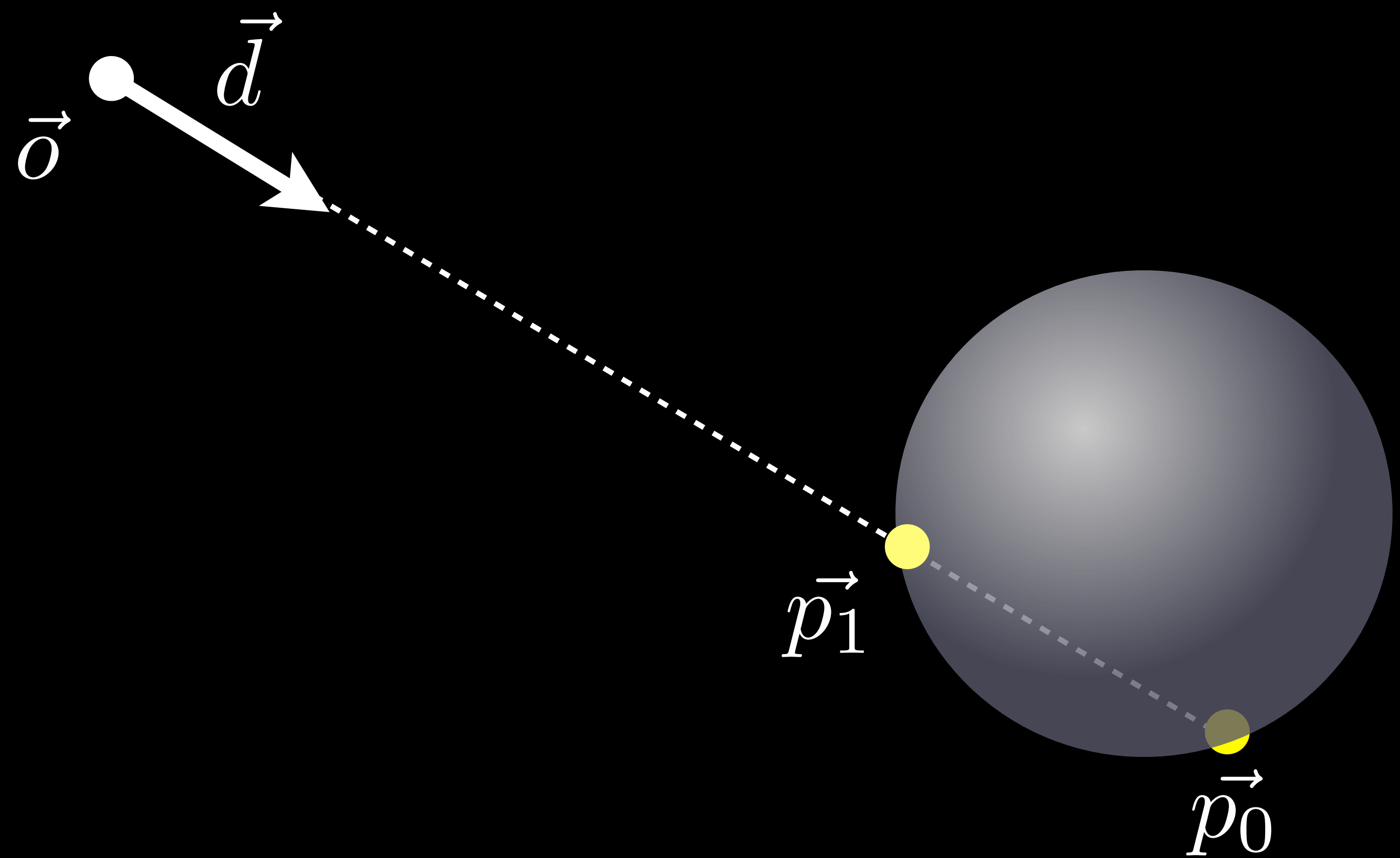
Quadratic equation of t $\longrightarrow$ Solve for t

Ray-Sphere Intersection

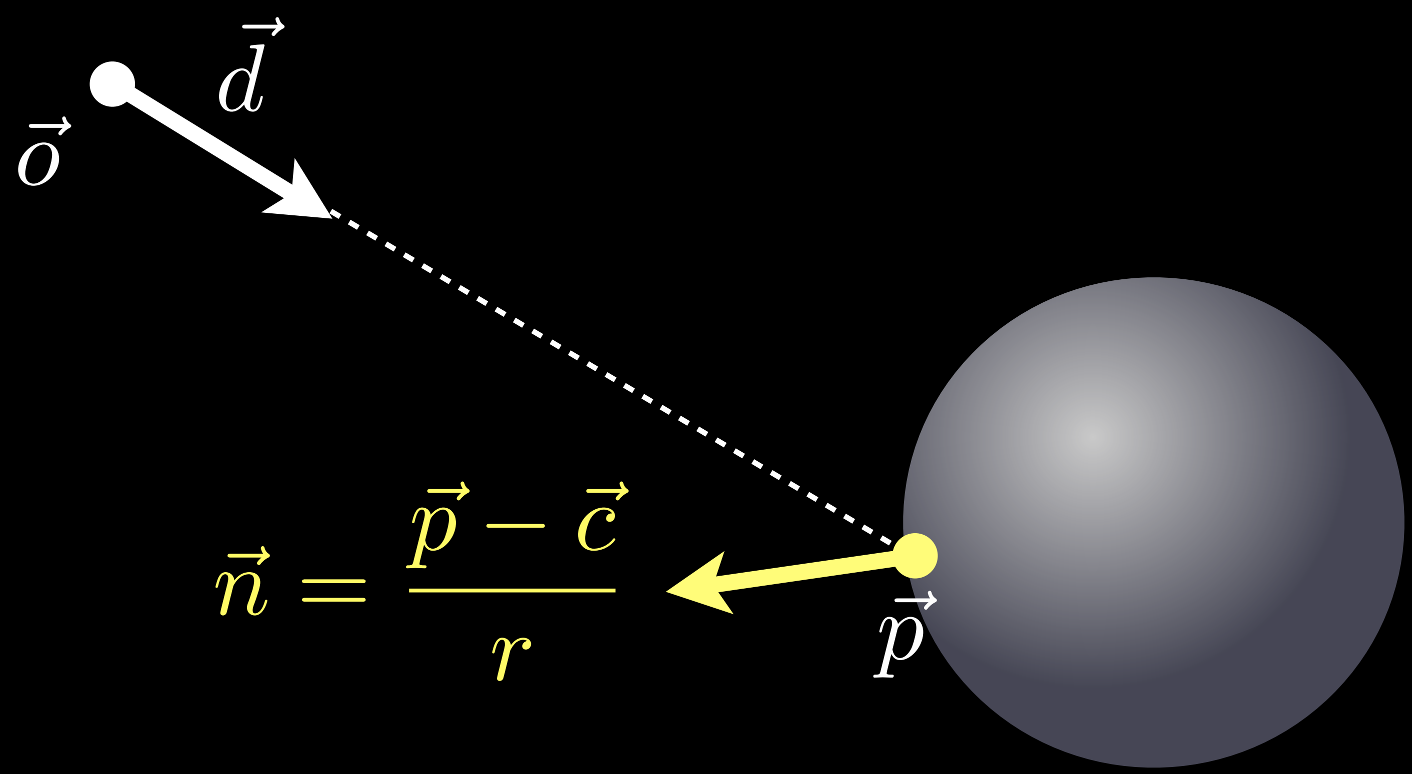
- t can have (considering only real numbers)
  - 0 solution : no hit point
  - 1 solution : hit at the edge
  - 2 solutions
    - two negatives : hit points are behind
    - two positives : hit points are front
    - positive and negative : origin is in the sphere

Ray-Sphere Intersection

- Two hit points - take the closest



Normal Vector



$$\vec{\nabla} \cdot \left((\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) - r^2 \right) = 2(\vec{p} - \vec{c})$$

Ray-Implicit Surface Intersection

- Generalized to any implicit surface

Intersection point:

Solve $f(\vec{p}(t)) = 0$

e.g., $||(\vec{p}(t) - \vec{c})||^2 - r^2 = 0$

Normal vector:

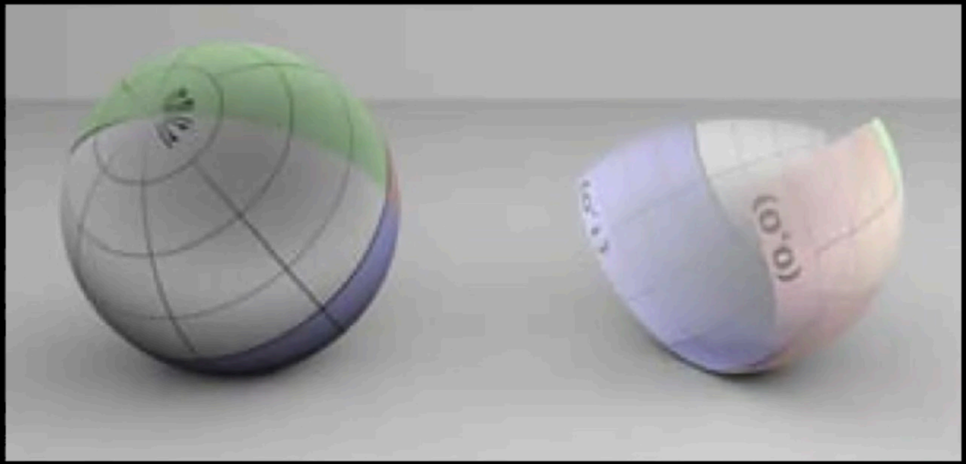
$$\vec{n} = \frac{\vec{\nabla} \cdot f(\vec{p}(t))}{||\vec{\nabla} \cdot f(\vec{p}(t))||}$$

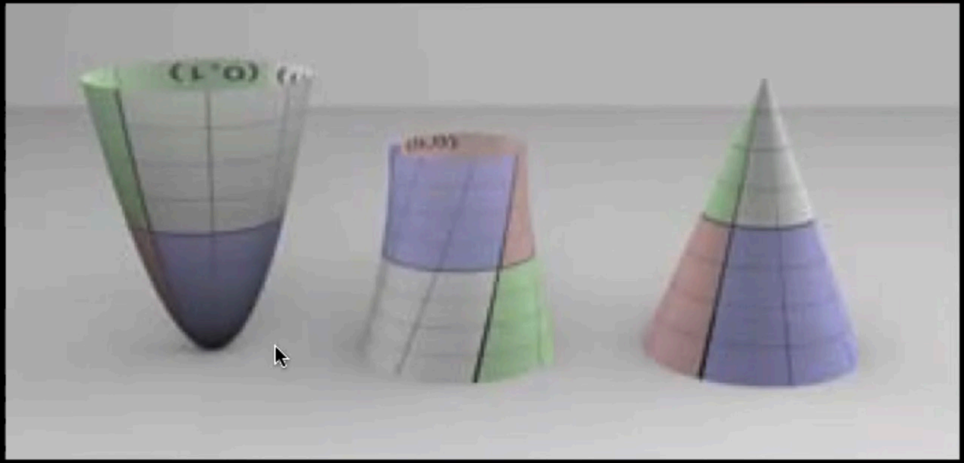
Ray-Implicit Surface Intersection

- $f(\vec{p}(t)) = 0$ can be
  - Linear: Plane
  - Quadratic: Sphere
  - Cubic: Bézier (cubic)
  - Quartic: Phong tessellation
  - ...and anything

Ray-Implicit Surface Intersection

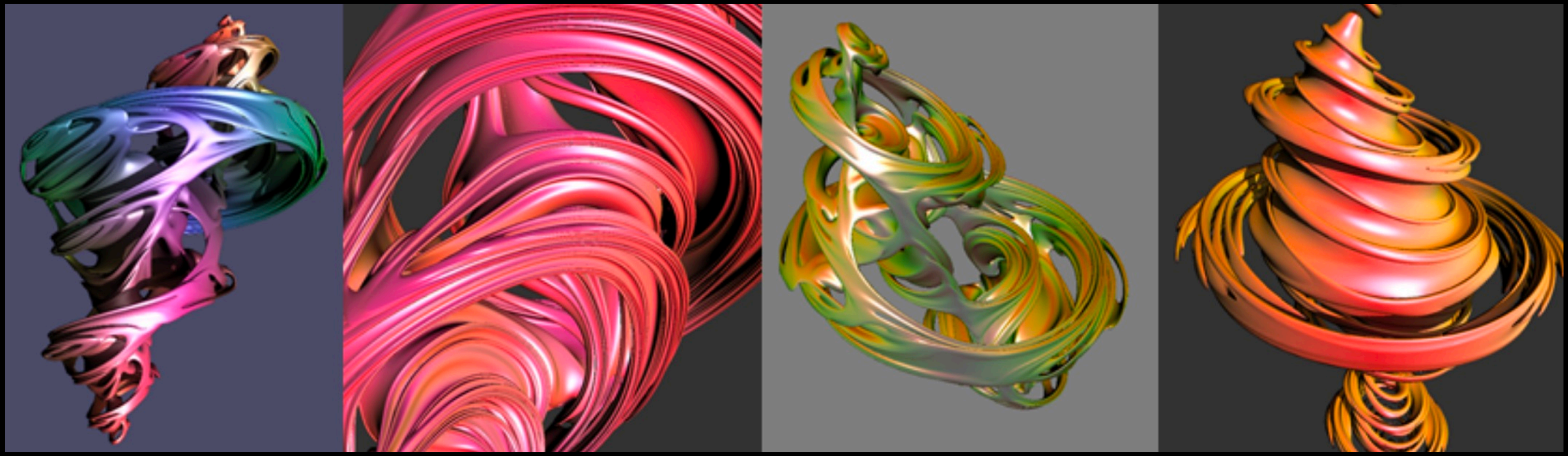
- Quadratic





Ray-Implicit Surface Intersection

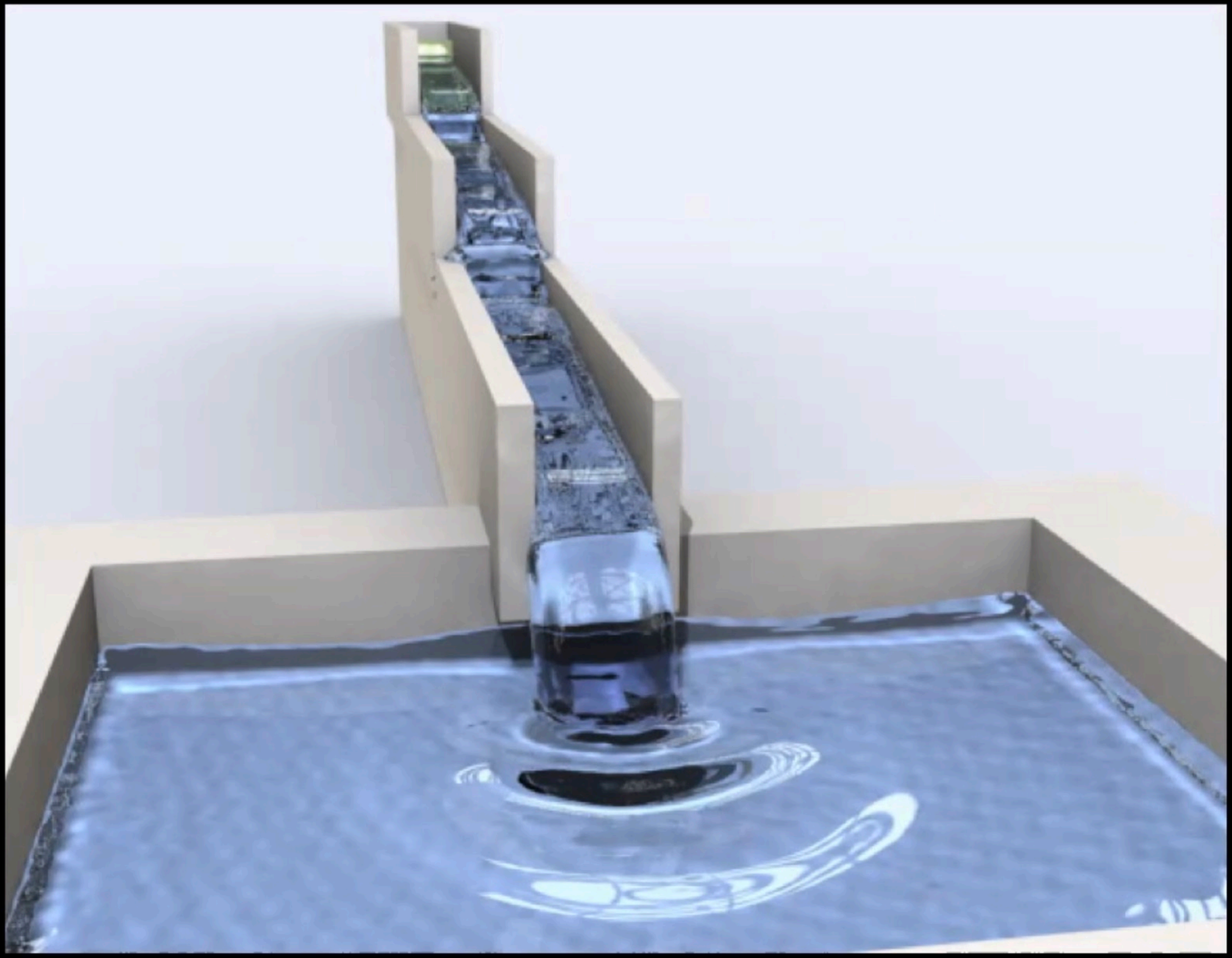
- Julia set



“Ray Tracing Quaternion Julia Sets on the GPU” [Crane 2005]

Ray-Implicit Surface Intersection

- Fluid simulation

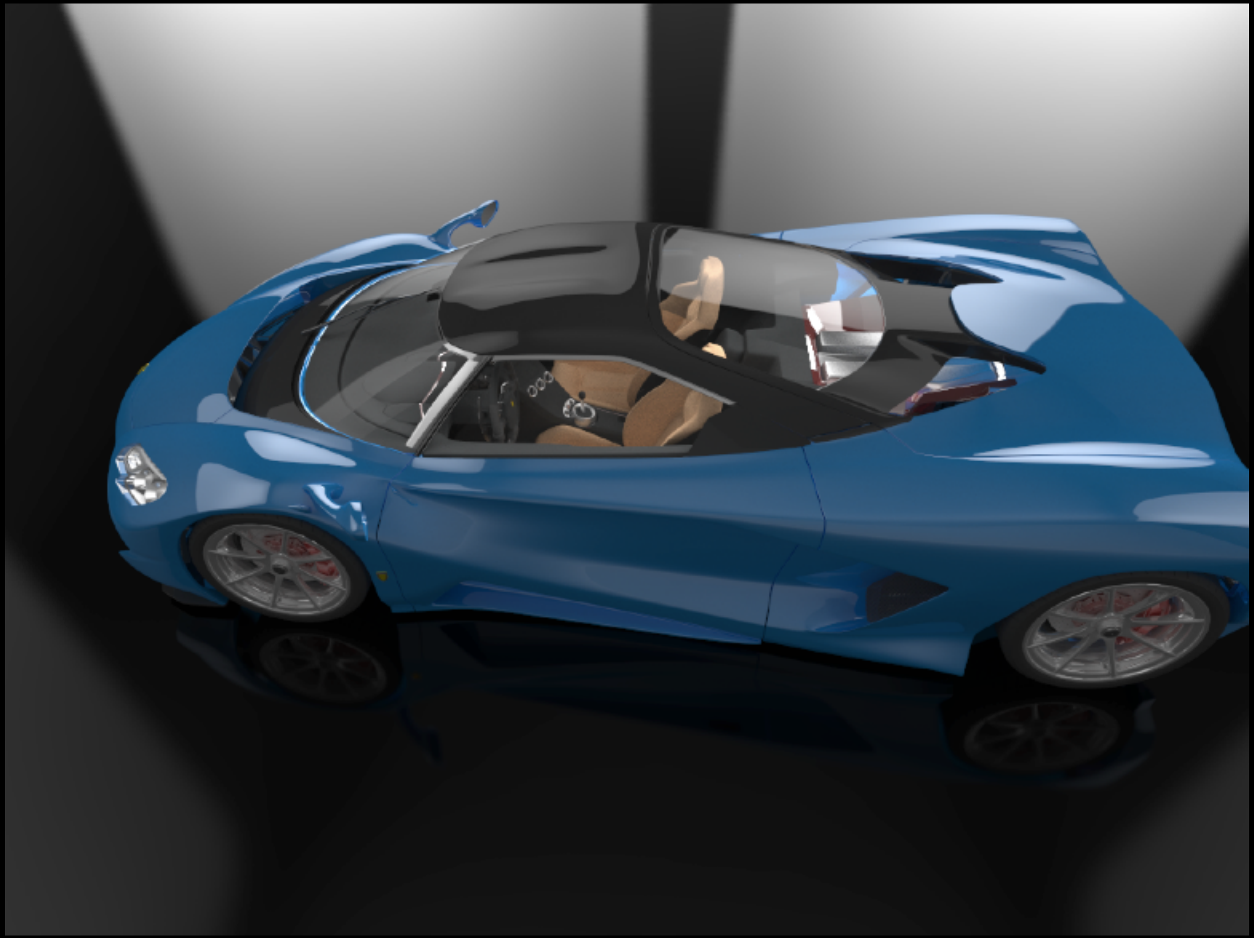


Ray-Implicit Surface Intersection

- Procedural geometry

Ray-Implicit Surface Intersection

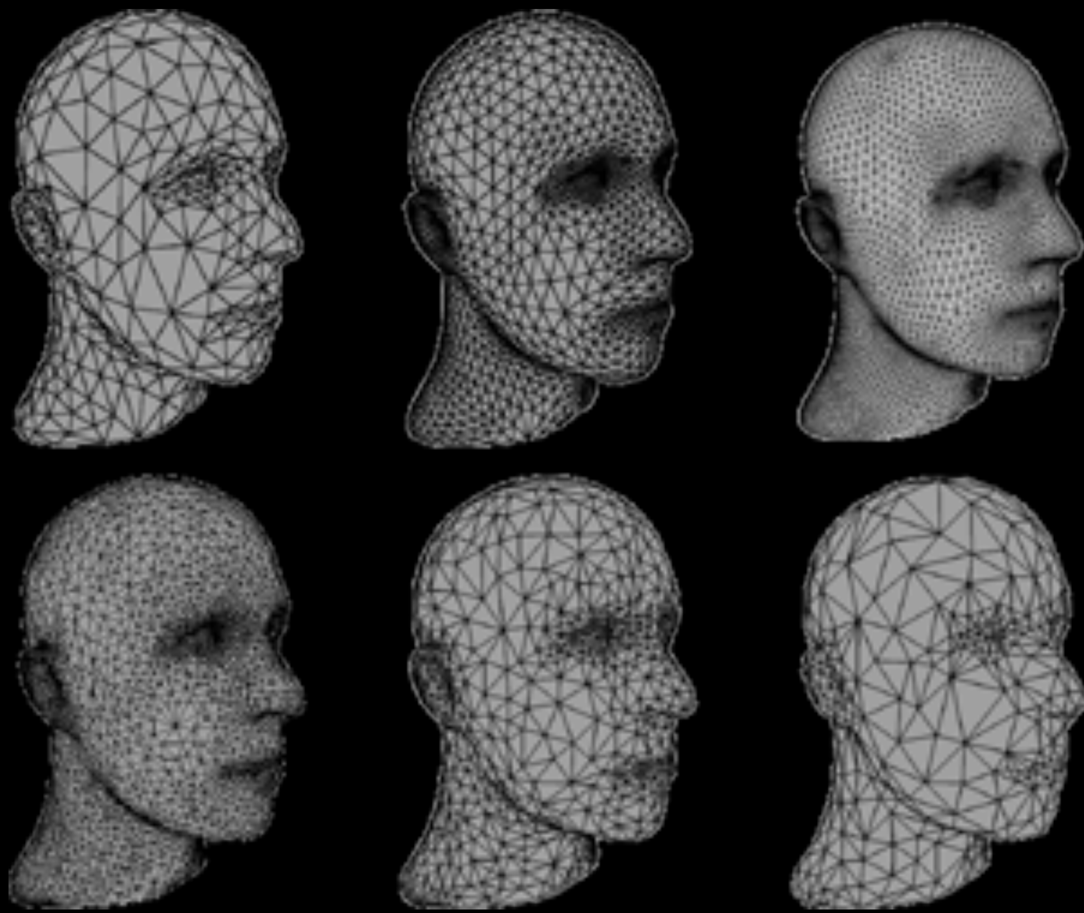
- Subdivision surfaces



“Direct Ray Tracing of Full-Featured Subdivision Surfaces with Bezier Clipping”

Triangle Mesh

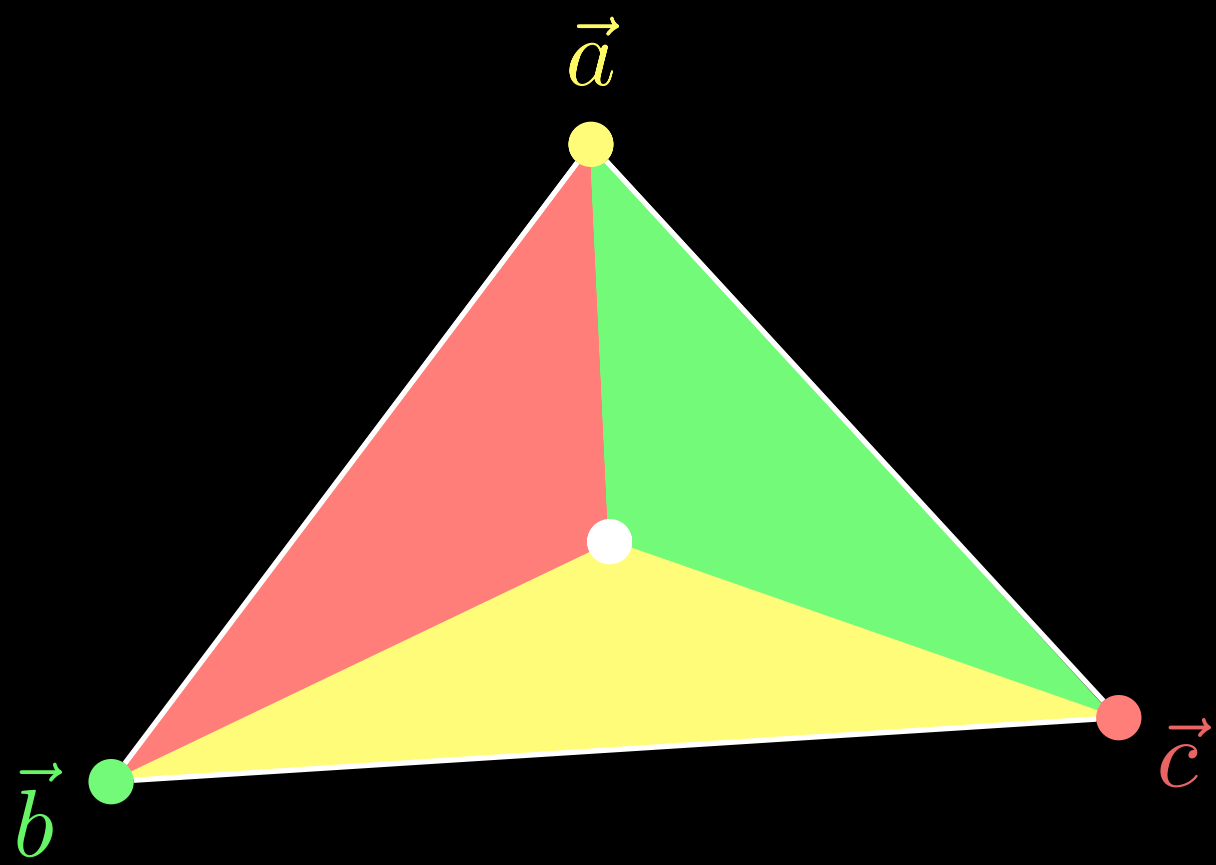
- Approximate shapes with triangles



“Multiresolution Hierarchies on Unstructured Triangle Meshes” [Kobbelt et al., 1999]

Barycentric Coordinates

- Ratios of areas of the sub-triangles



$$\alpha = \frac{A_a}{A_a + A_b + A_c}$$

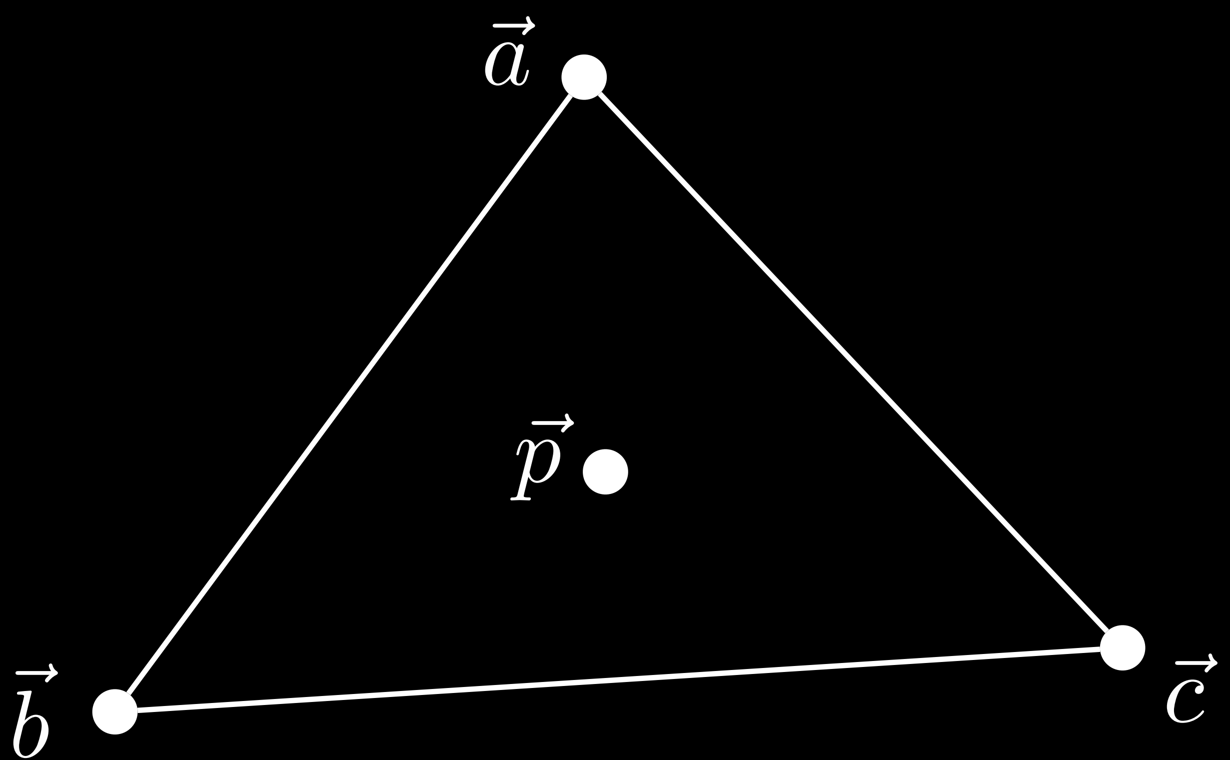
$$\beta = \frac{A_b}{A_a + A_b + A_c}$$

$$\gamma = \frac{A_c}{A_a + A_b + A_c}$$

Barycentric Coordinates

- Parametric description of a point in a triangle

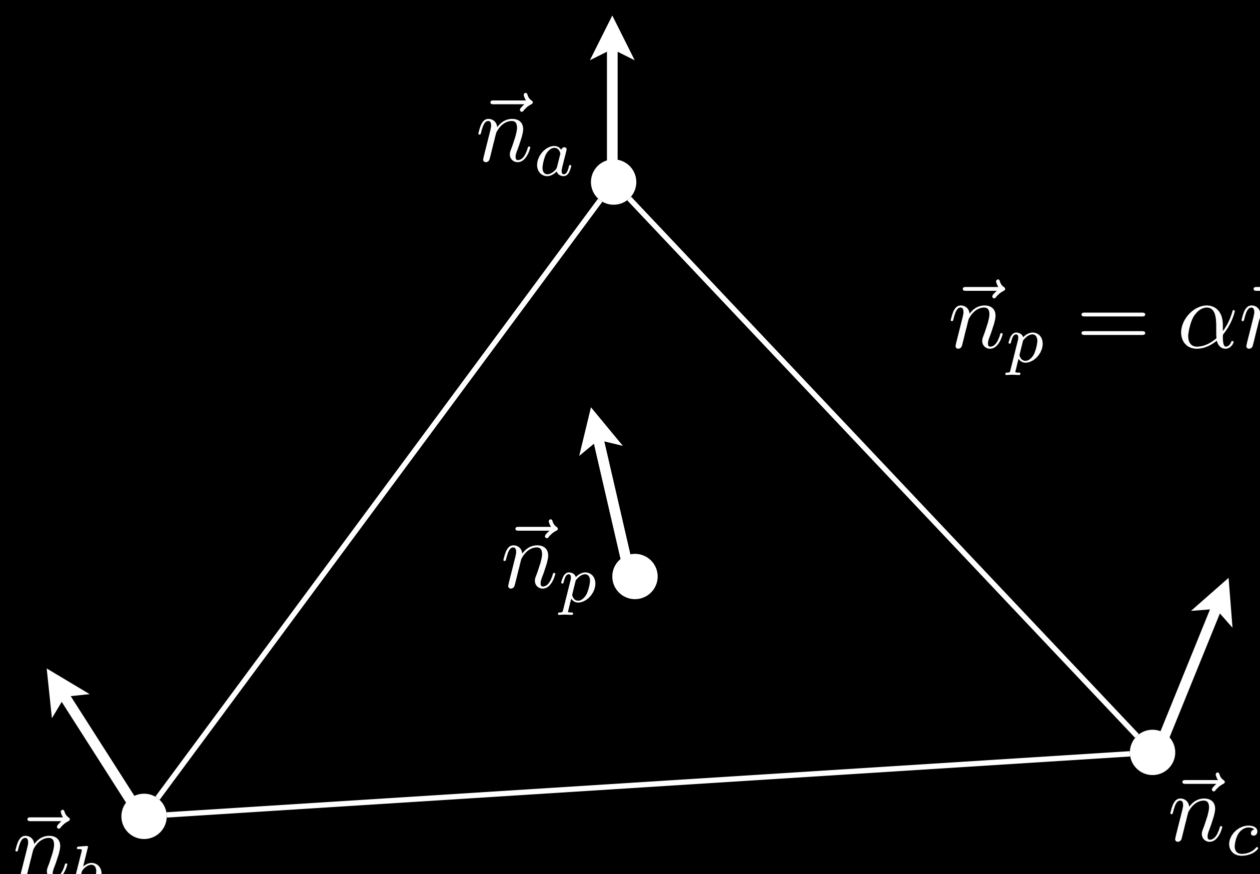
$$\vec{p} = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$



$$0 < \alpha < 1$$
$$0 < \beta < 1$$
$$0 < \gamma < 1$$
$$\alpha + \beta + \gamma = 1$$

Barycentric Coordinates

- Interpolate values at the vertices

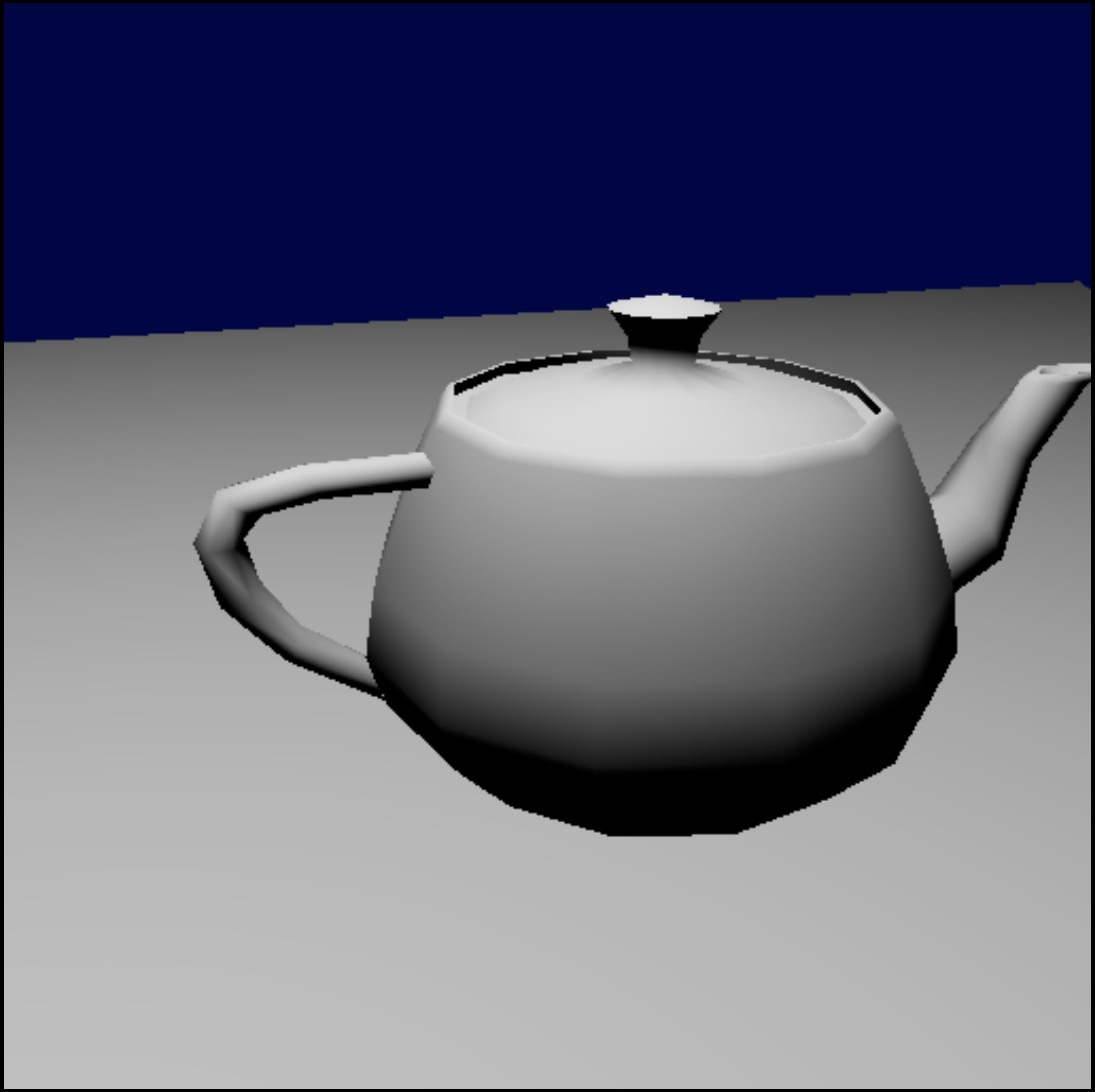


$$\vec{n}_p = \alpha\vec{n}_a + \beta\vec{n}_b + \gamma\vec{n}_c$$

Normalize it before use

Interpolation

Interpolation



Ray-Triangle Intersection

- Calculate $(t, \alpha, \beta, \gamma)$ as fast as possible

- Modification of ray-plane intersection
- Direct methods
  - Cramer's rule
  - Signed volumes

Ray-Triangle Intersection Cramer's Rule

$$\vec{p} = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

Ray-Triangle Intersection Cramer's Rule

$$\vec{o} + t\vec{d} = \alpha\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

Ray-Triangle Intersection Cramer's Rule

$$\vec{o} + t\vec{d} = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

Ray-Triangle Intersection Cramer's Rule

$$\vec{o} + t\vec{d} = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

$$o_x + td_x = (1 - \beta - \gamma)a_x + \beta b_x + \gamma c_x$$
$$o_y + td_y = (1 - \beta - \gamma)a_y + \beta b_y + \gamma c_y$$
$$o_z + td_z = (1 - \beta - \gamma)a_z + \beta b_z + \gamma c_z$$

3 equations for 3 unknowns

Ray-Triangle Intersection Cramer's Rule

$$\vec{o} + t\vec{d} = (1 - \beta - \gamma)\vec{a} + \beta\vec{b} + \gamma\vec{c}$$

$$\begin{bmatrix} a_x - b_x & a_x - c_x & d_x \\ a_y - b_y & a_y - c_y & d_y \\ a_z - b_z & a_z - c_z & d_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} a_x - o_x \\ a_y - o_y \\ a_Z - o_z \end{bmatrix}$$

Solve the equation with Cramer's Rule

$$\det\left(\vec{a}, \vec{b}, \vec{c}\right) = \left(\vec{a} \times \vec{b}\right) \cdot \vec{c}$$

Ray-Triangle Intersection Cramer's Rule

- Accept the solution only if

$$t_{\text{closest}} > t > 0$$

$$1 > \beta > 0$$

$$1 > \gamma > 0$$

$$1 > 1 - \beta - \gamma > 0$$

t_{closest} : the smallest positive t values so far

Ray-Triangle Intersection

- There are many different approaches!
  - Numerical precision
  - Performance
  - Storage cost
  - SIMD friendliness

- Genetic programming for performance

“Optimizing Ray-Triangle Intersection via Automated Search” [Kensler 2006]

GLSL Sandbox

- Interactive coding environment for WebGL
- You write a program for each pixel in GLSL
  - Automatically loop over all the pixels
  - Uses programmable shader units on GPUs

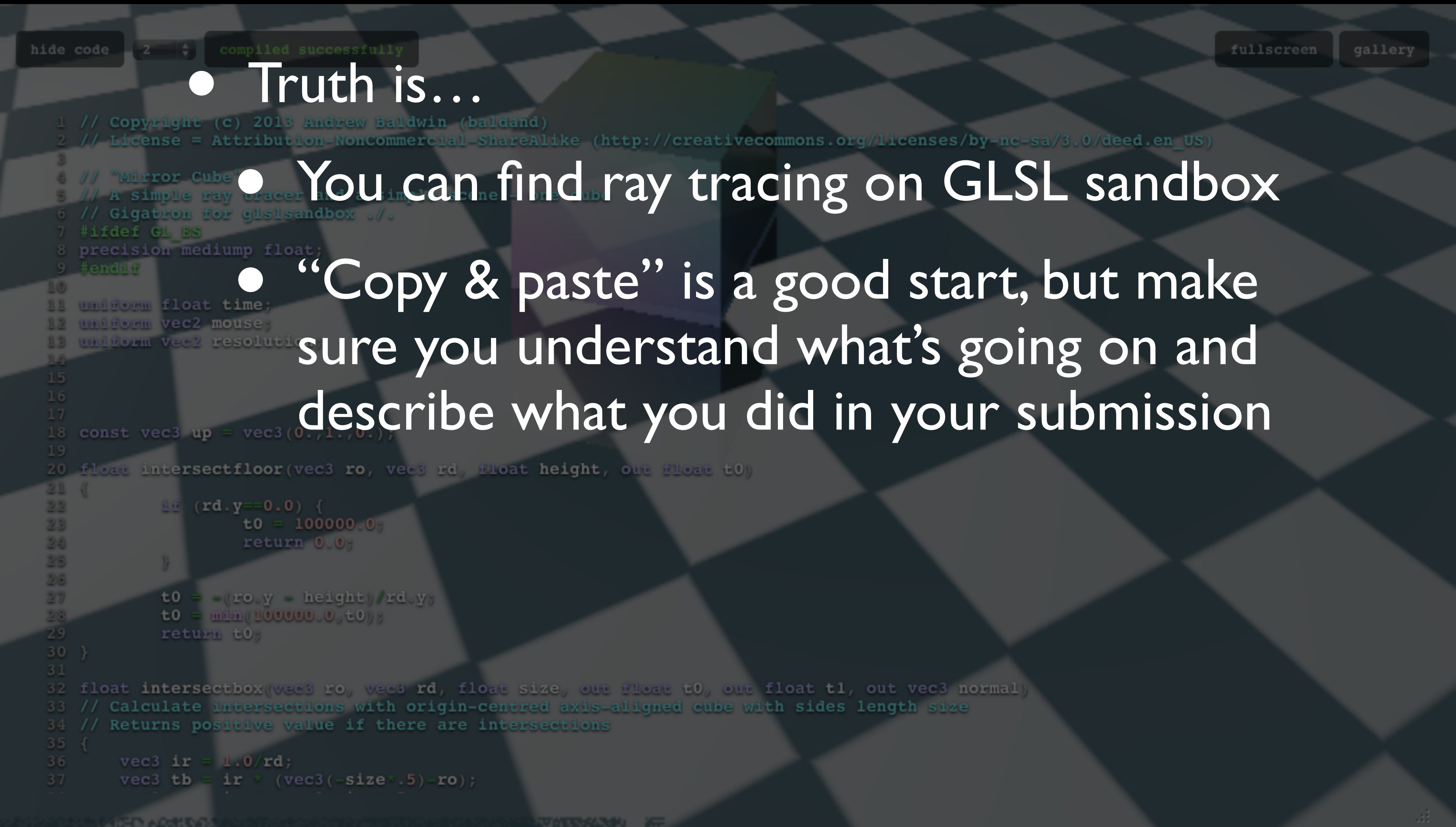
http://glslsandbox.com

GLSL implementation

```
for all pixels {
    ray = generate_camera_ray( pixel )
    for all objects {
        hit = intersect( ray, object )
        if “hit” is closer than “first_hit” {first_hit = hit}
    }
    pixel = shade( first_hit )
}
```

GLSL implementation

- Truth is…
  - You can find ray tracing on GLSL sandbox
  - “Copy & paste” is a good start, but make sure you understand what’s going on and describe what you did in your submission



Next Time

```
for all pixels {
    ray = generate_camera_ray( pixel )
    for all objects {
        hit = intersect( ray, object )
        if "hit" is closer than "first_hit" {first_hit = hit}
    }
    pixel = shade( first_hit )
}
```