

Introduction to Computer Graphics

Toshiya Hachisuka

Lecturer

- Toshiya Hachisuka
- Leading the computer graphics group
- Office: I-REF 503
- <http://www.ci.i.u-tokyo.ac.jp/~hachisuka>

Today

- Introduction to ray tracing
- Basic ray-object intersection



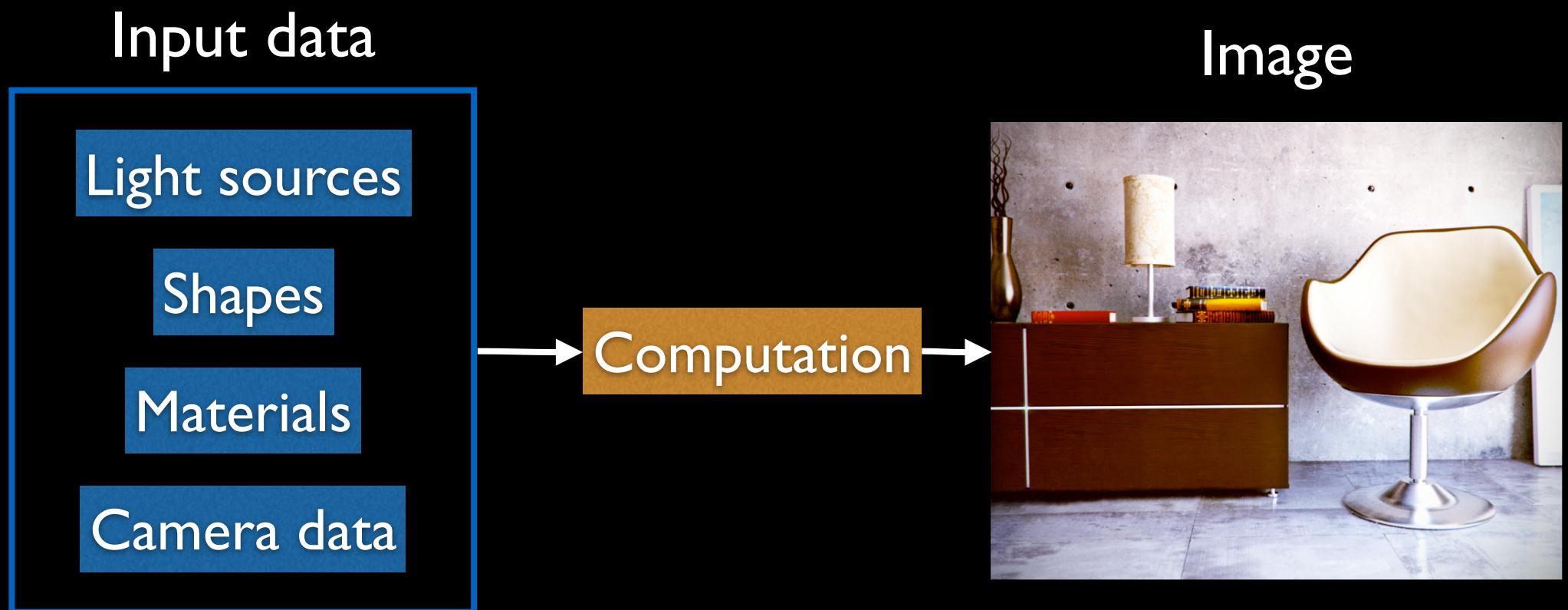
LuxRender



LuxRender

How can we generate realistic images?

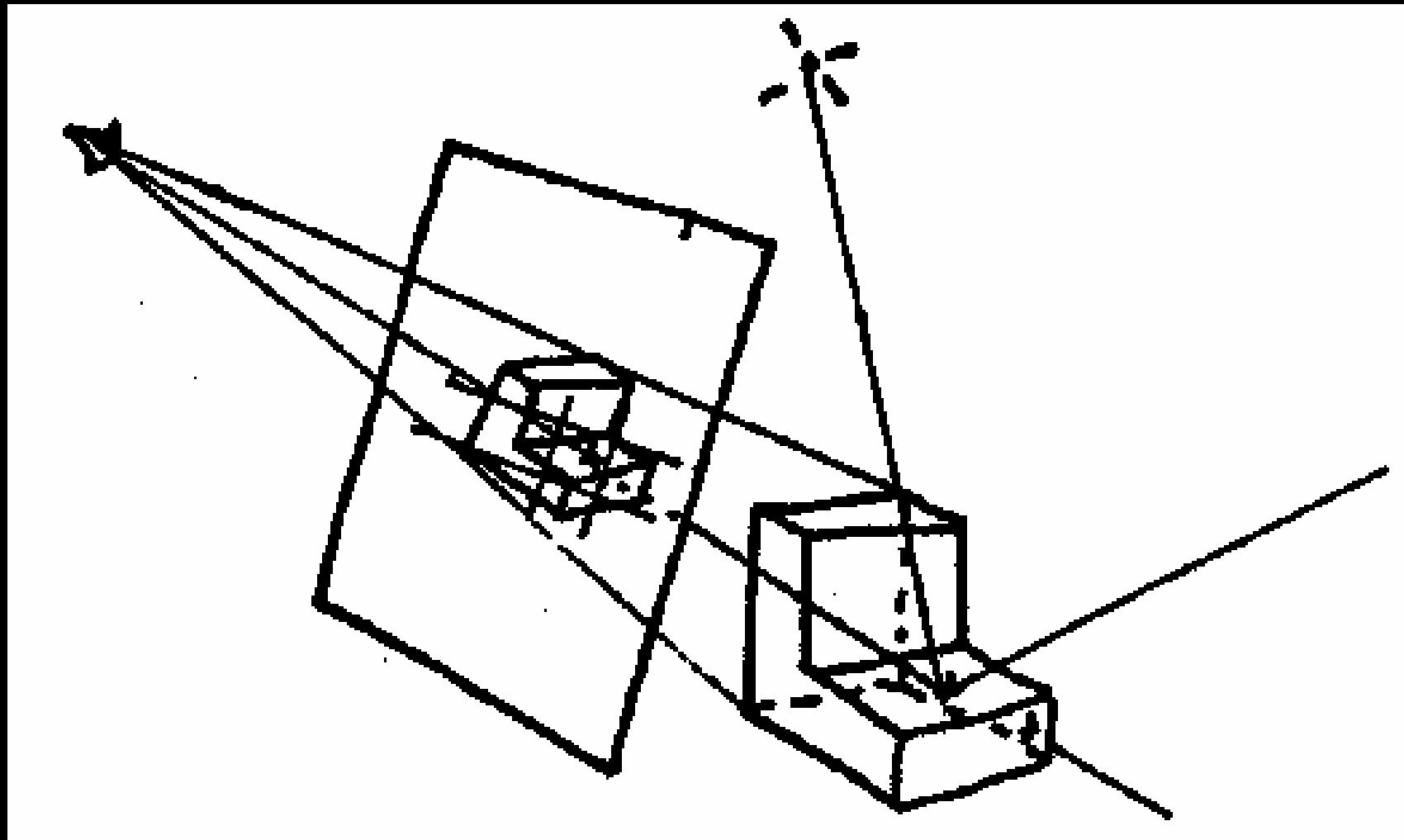
Rendering



Interdisciplinary Nature

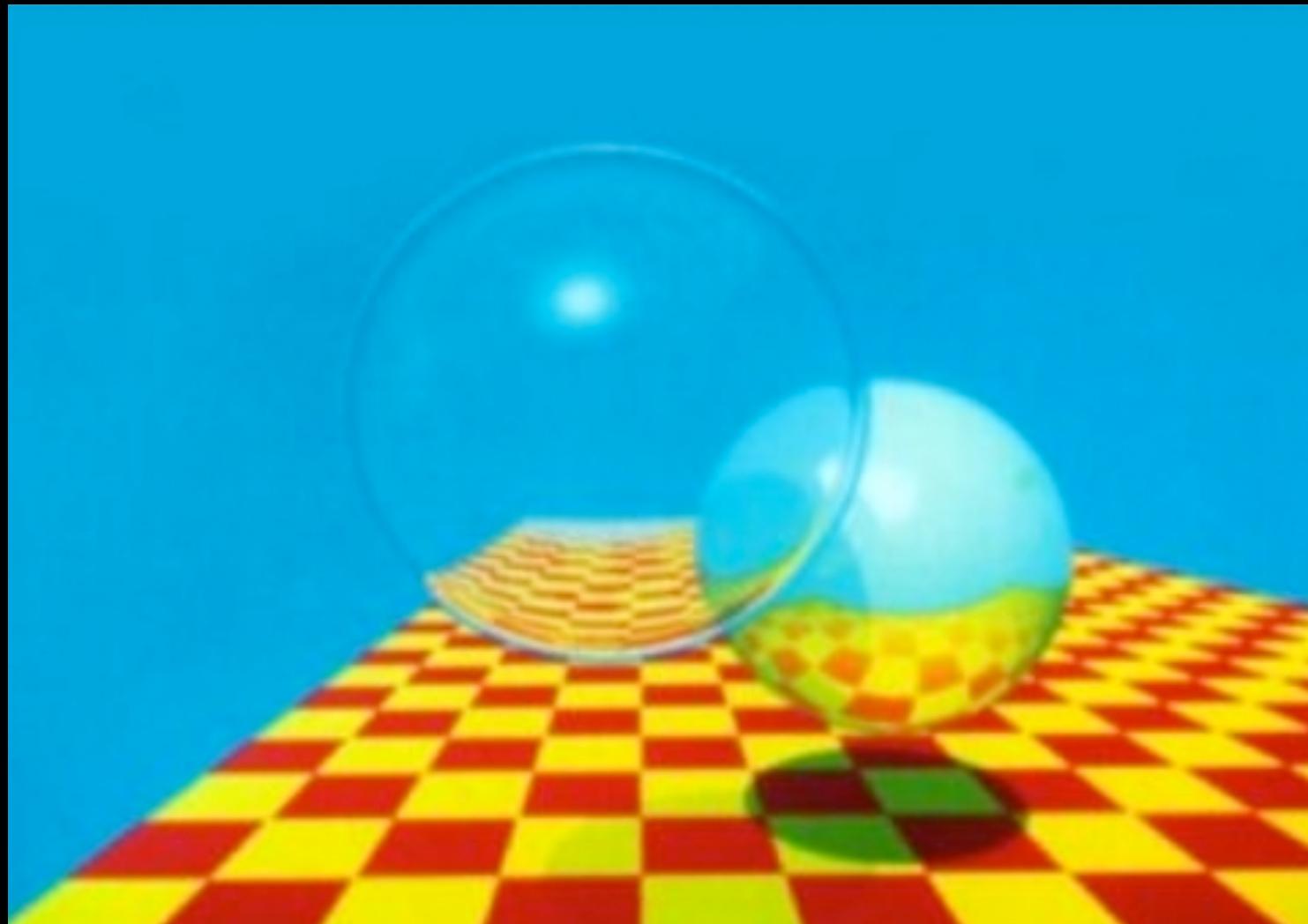
- Computer Science
 - Algorithms
 - Computational geometry
 - Software engineering
- Physics
 - Radiometry
 - Optics
- Mathematics
 - Algebra
 - Calculus
 - Statistics
- Perception
- Art

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

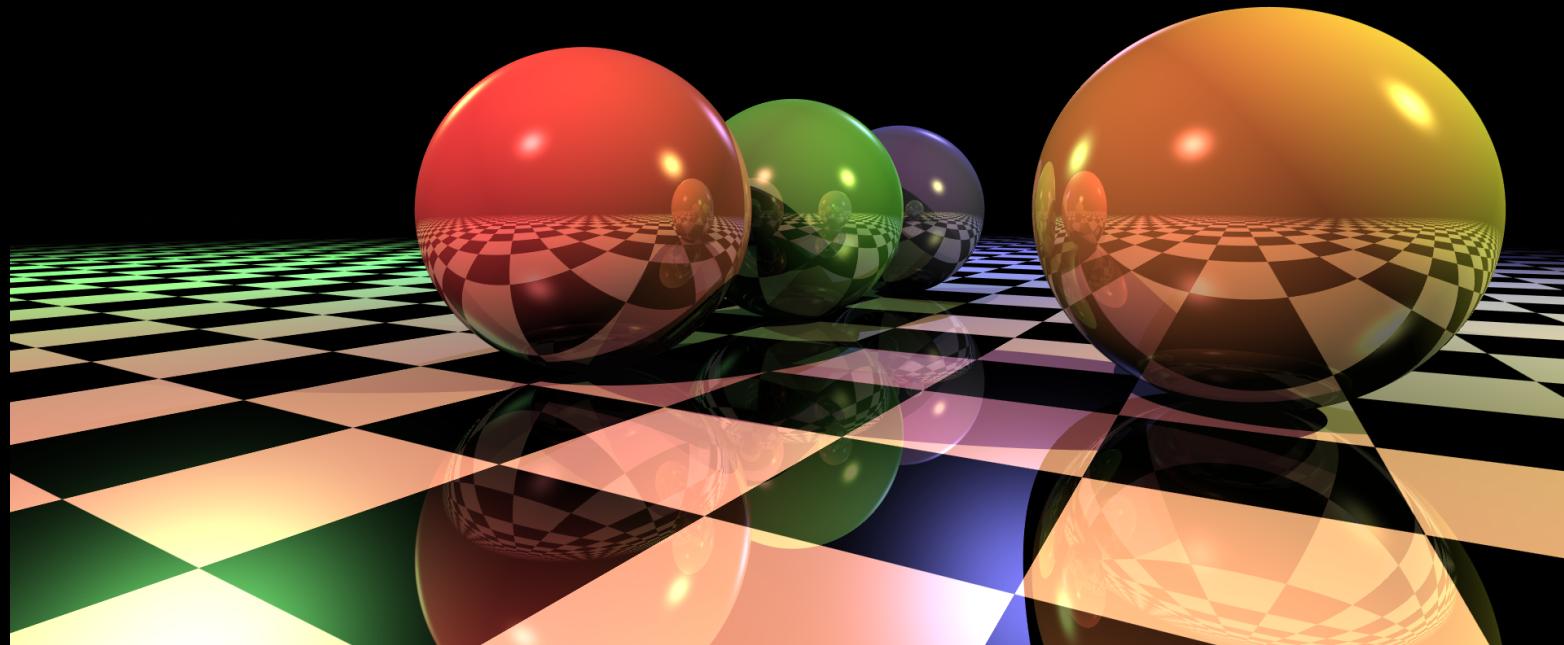
- Runs realtime on a GPU!



Screenshot of a demo using NVIDIA's OptiX

Whitted Ray Tracing Today

- Runs realtime on a GPU!



Whitted Ray Tracing Today

- Runs realtime on a GPU!

You are going to implement something like this!



Ray Tracing - Pseudocode

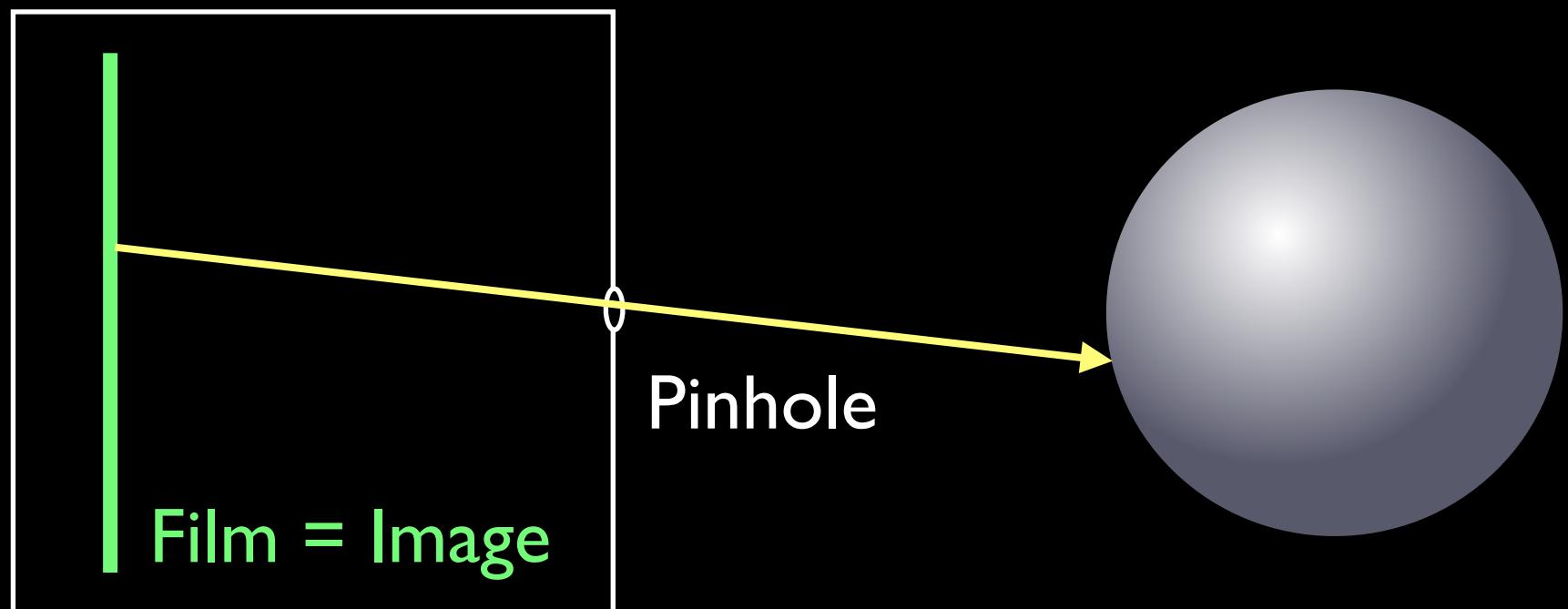
```
for all pixels {
    ray = generate_camera_ray( pixel )
    for all objects {
        hit = intersect( ray, object )
        if "hit" is closer than "first_hit" {first_hit = hit}
    }
    pixel = shade( first_hit )
}
```

Ray Tracing - Data Structures

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

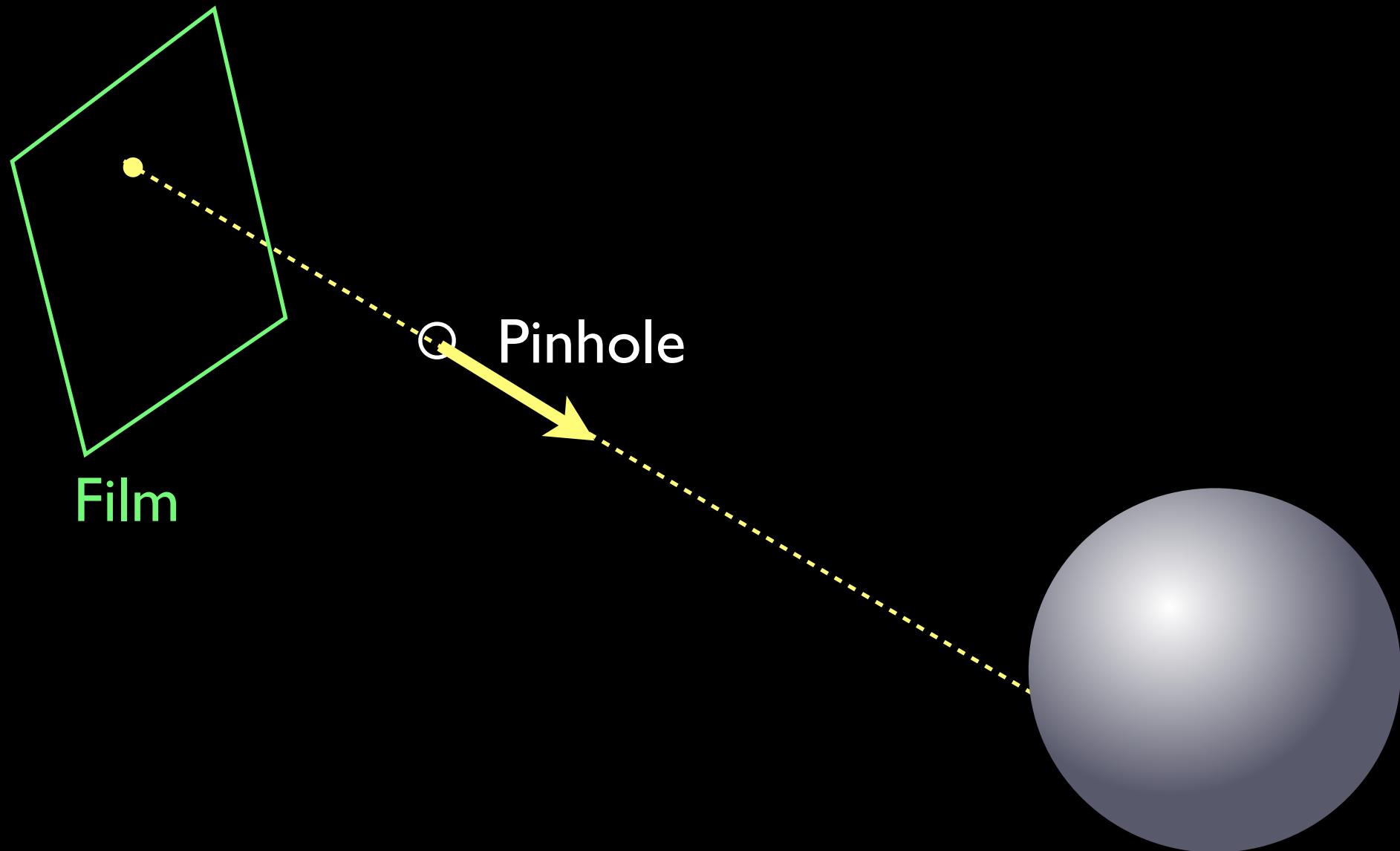
```
class ray {  
    vector origin  
    vector direction  
}
```

Pinhole Camera

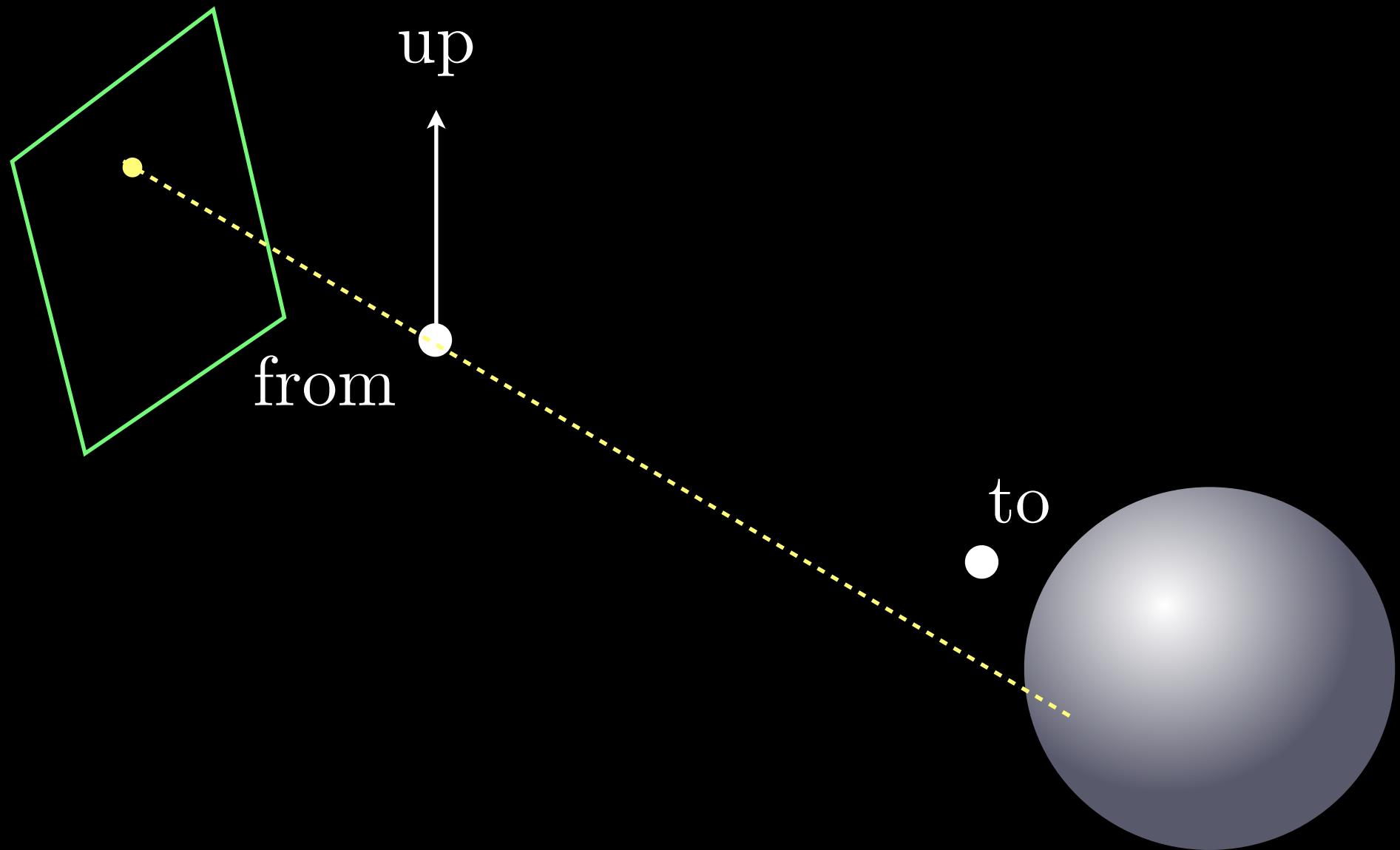


(the image is flipped)

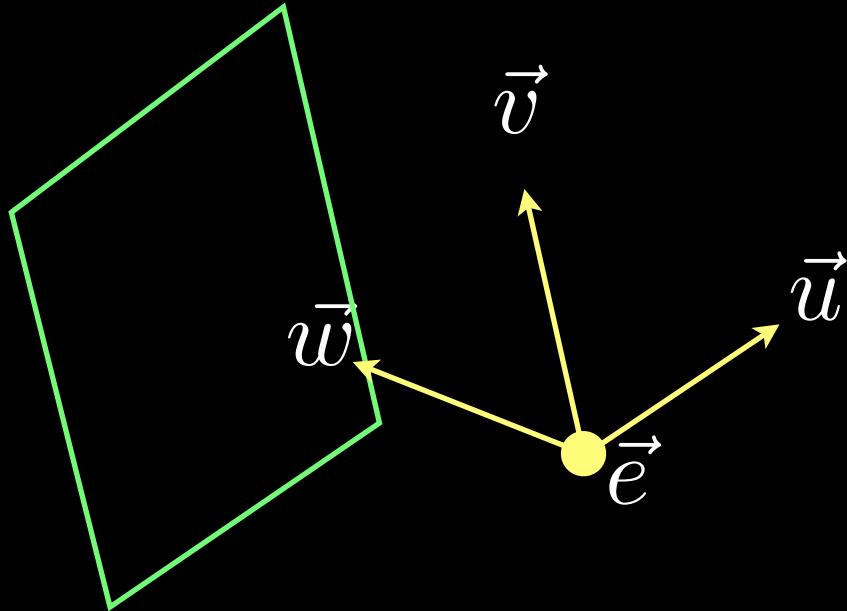
Pinhole Camera



Camera Coordinate System



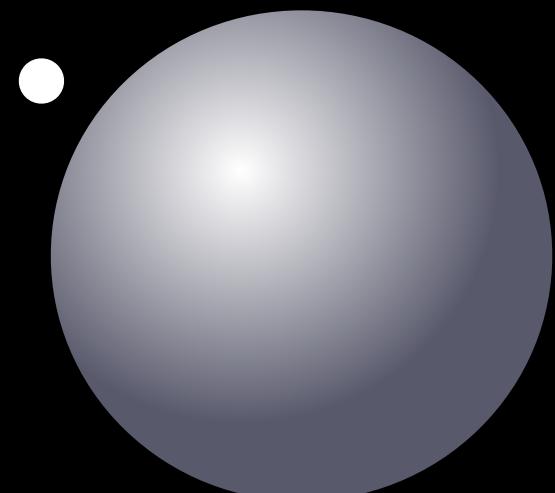
Camera Coordinate System



Orthonormal basis

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

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



Camera Coordinate System

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

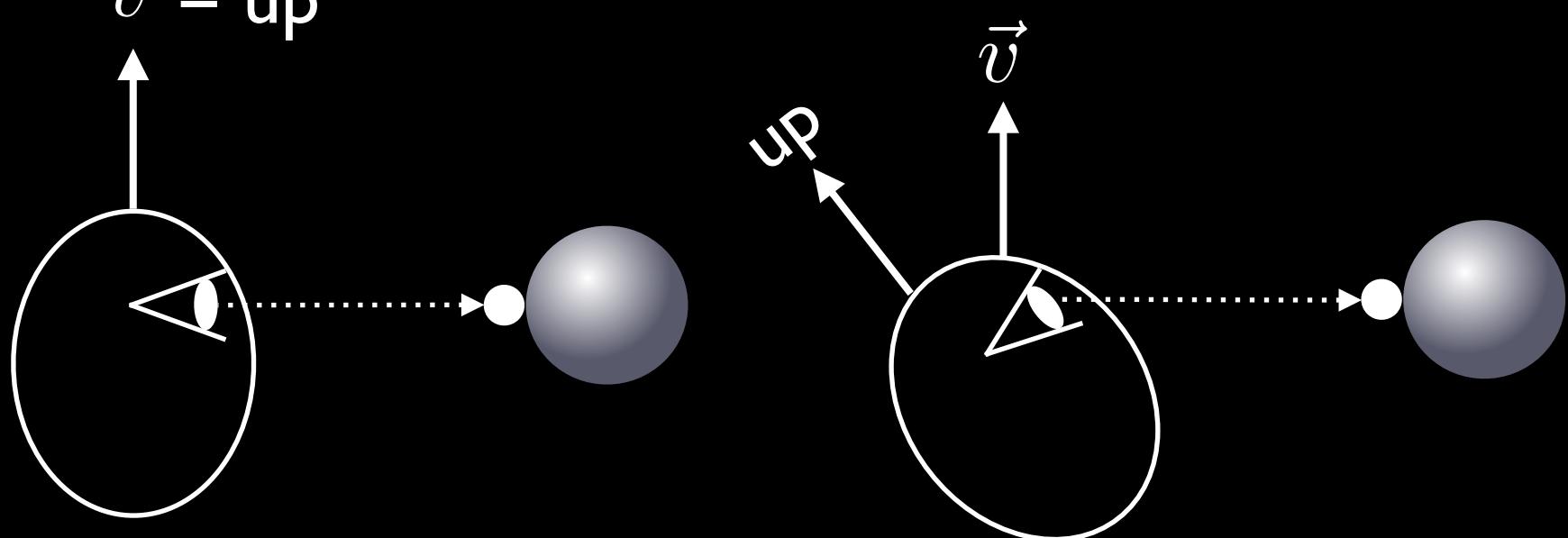
$$\vec{e} = \vec{C}_{\text{from}}$$

Origin

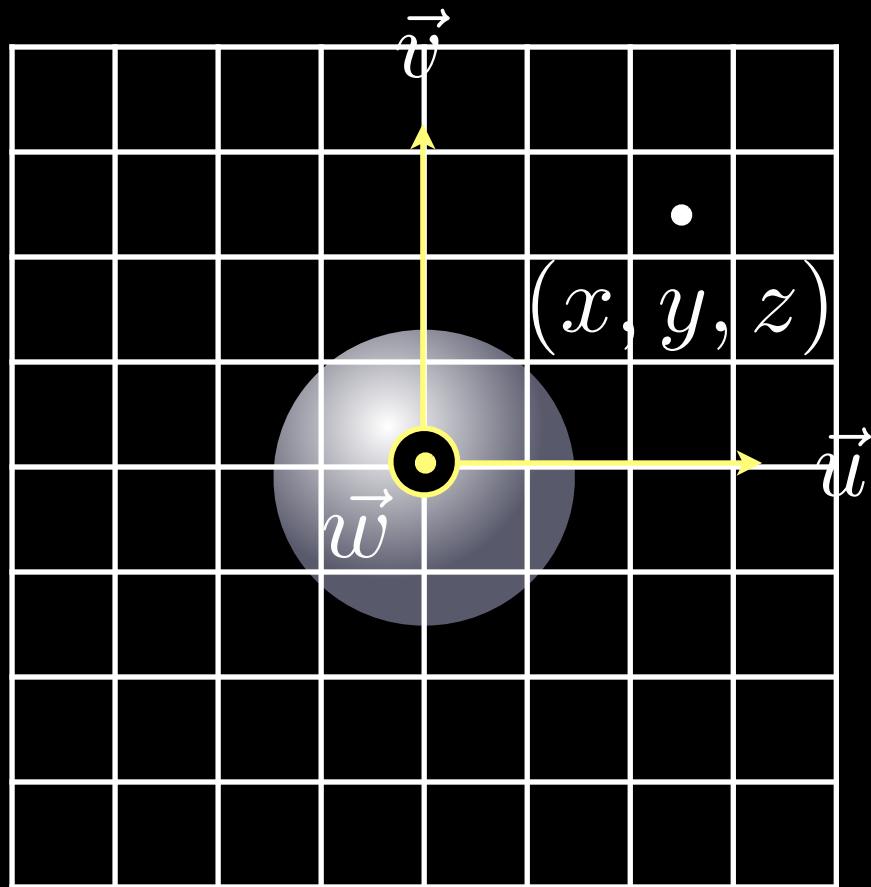
Up vector?

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

$$\vec{v} = \text{up}$$



Generating a Camera Ray



$$x = \text{film_w} \frac{\text{pixel_i} + 0.5}{\text{res_x}}$$

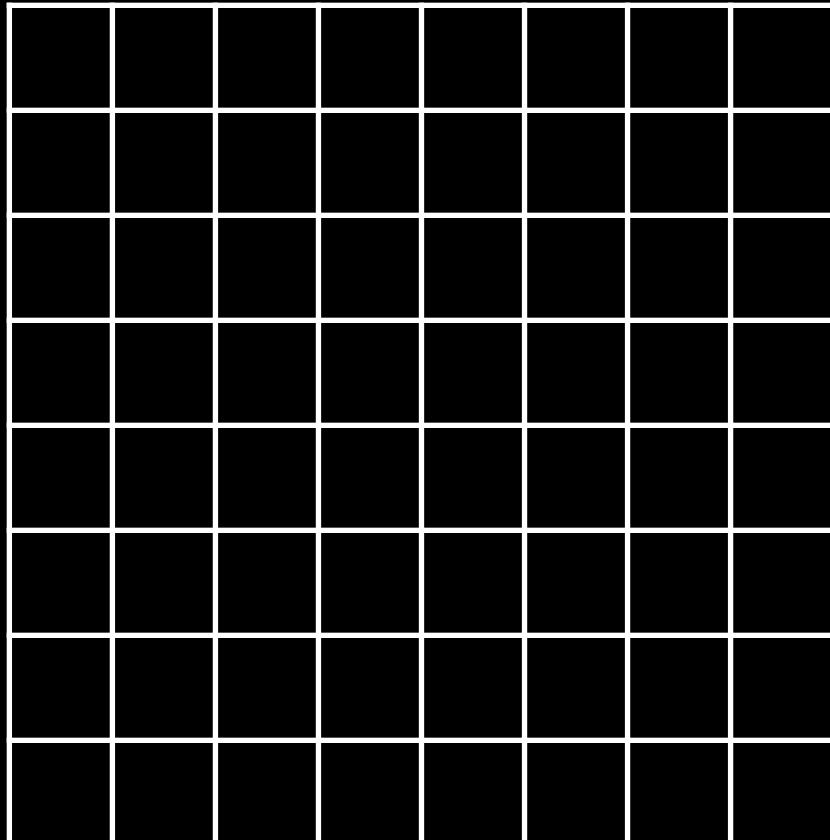
$$y = \text{film_h} \frac{\text{pixel_j} + 0.5}{\text{res_y}}$$

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

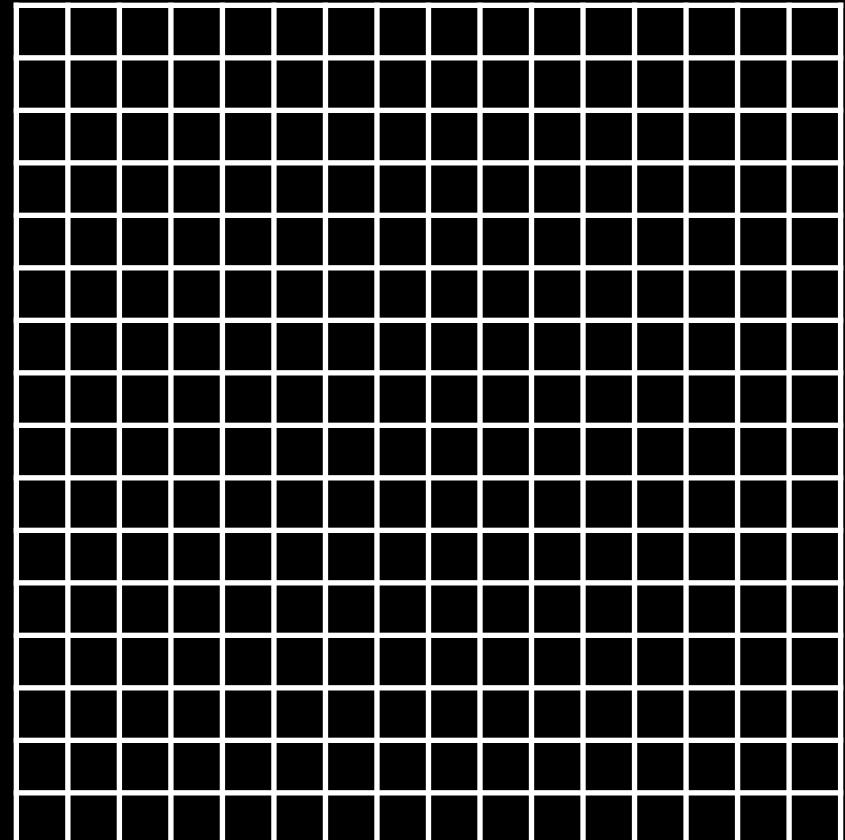
Pixel location in the camera coordinates

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:

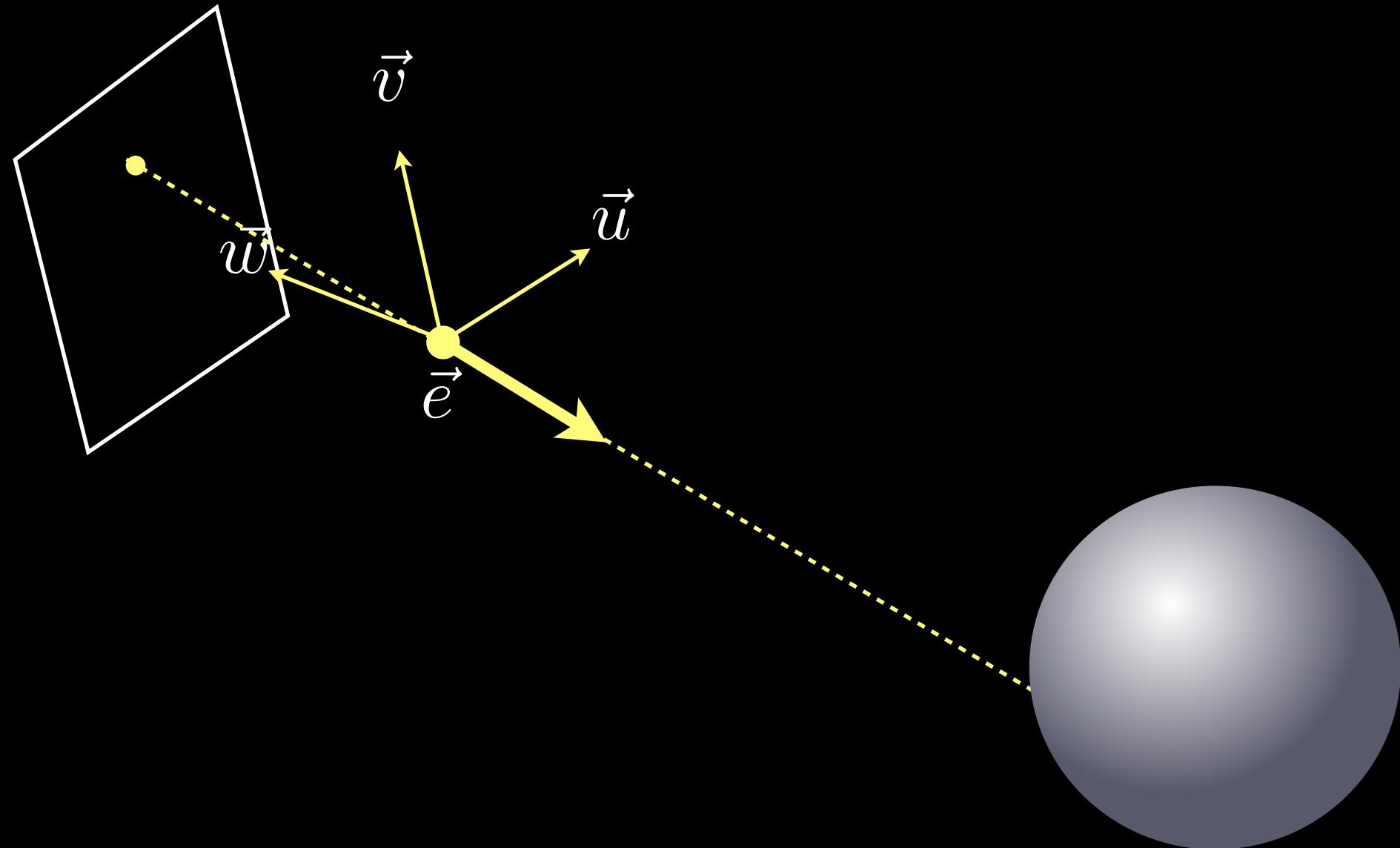
$$\text{pixel} = x\vec{u} + y\vec{v} + z\vec{w} + \vec{e}$$

- 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.]

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



Full Simulation

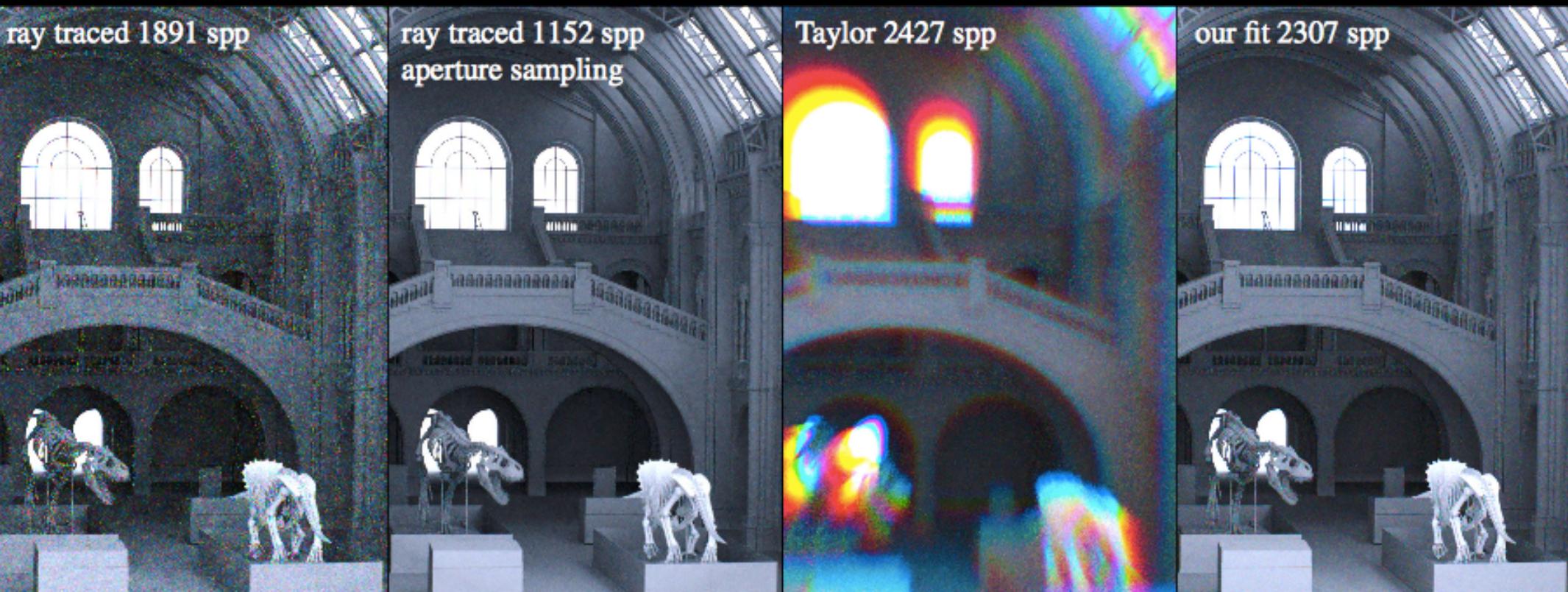


Thin Lens Approximation

More Realistic Cameras

[Hanika et al.]

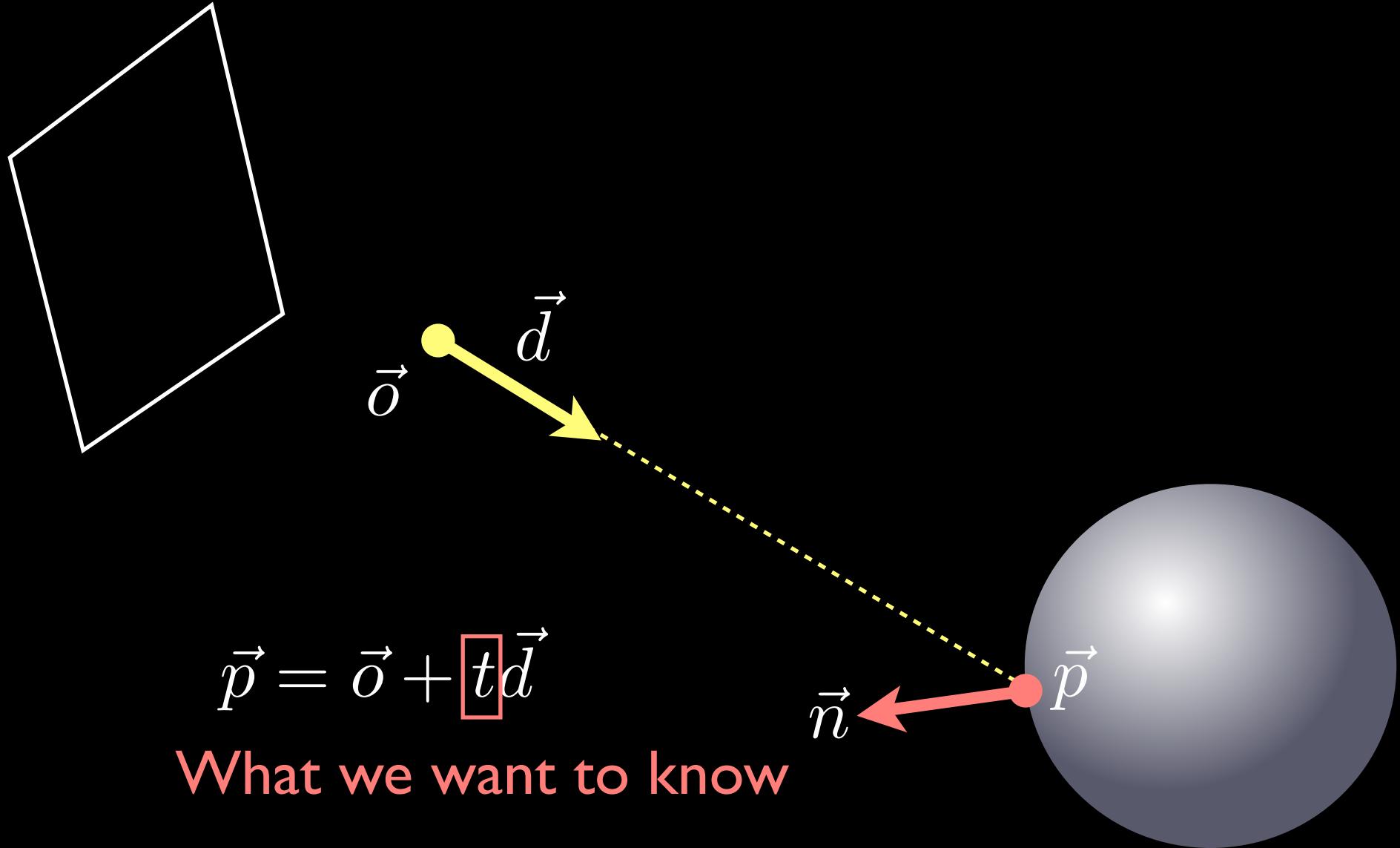
- “Efficient Monte Carlo Rendering with Realistic Lenses”
 - Polynomial approximation of a lens system



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 )
}
```

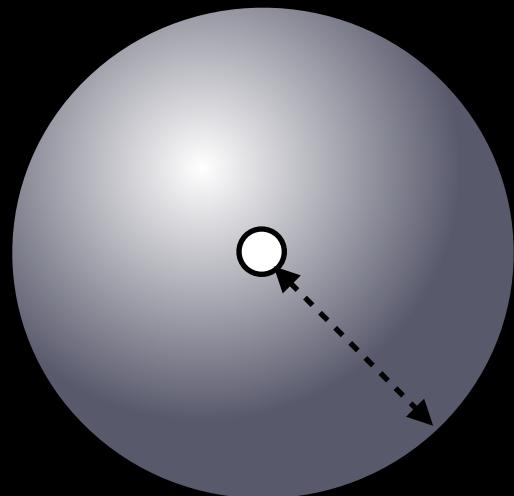
Goal



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

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

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

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

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

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

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

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

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

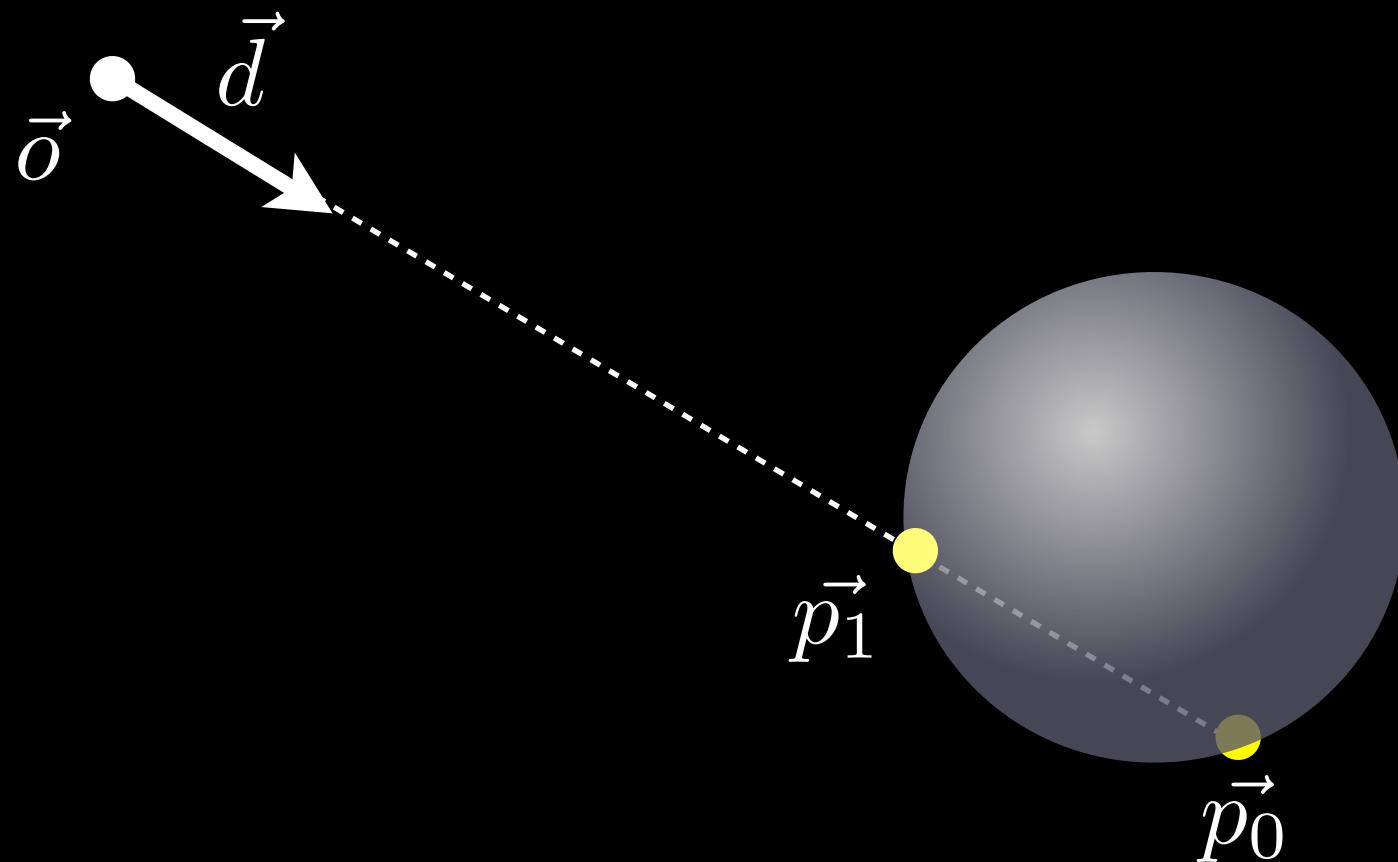
Quadratic equation of $t \longrightarrow$ Solve for t

Ray-Sphere Intersection

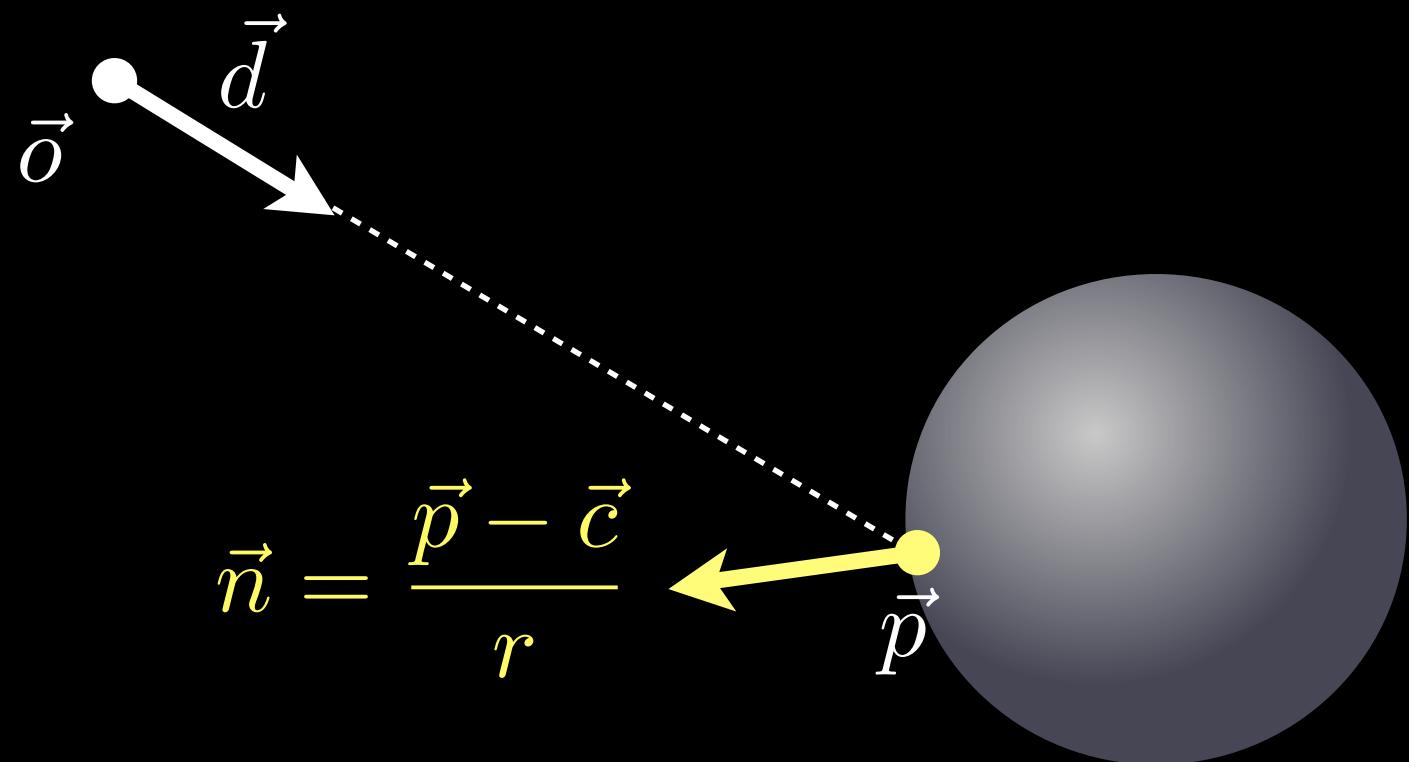
- t can have (considering only real numbers)
 - 0 solution : no hit point
 - 1 solution : hit at the edge
 - 2 solutions
 - two negatives : hit points are behind
 - two positives : hit points are front
 - positive and negative : origin is in the sphere

Ray-Sphere Intersection

- Two hit points - take the closest



Normal Vector



$$\vec{n} = \frac{\vec{p} - \vec{c}}{r}$$

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

Ray-Implicit Surface Intersection

- Generalized to any implicit surface

Intersection point:

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

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

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

- Julia set



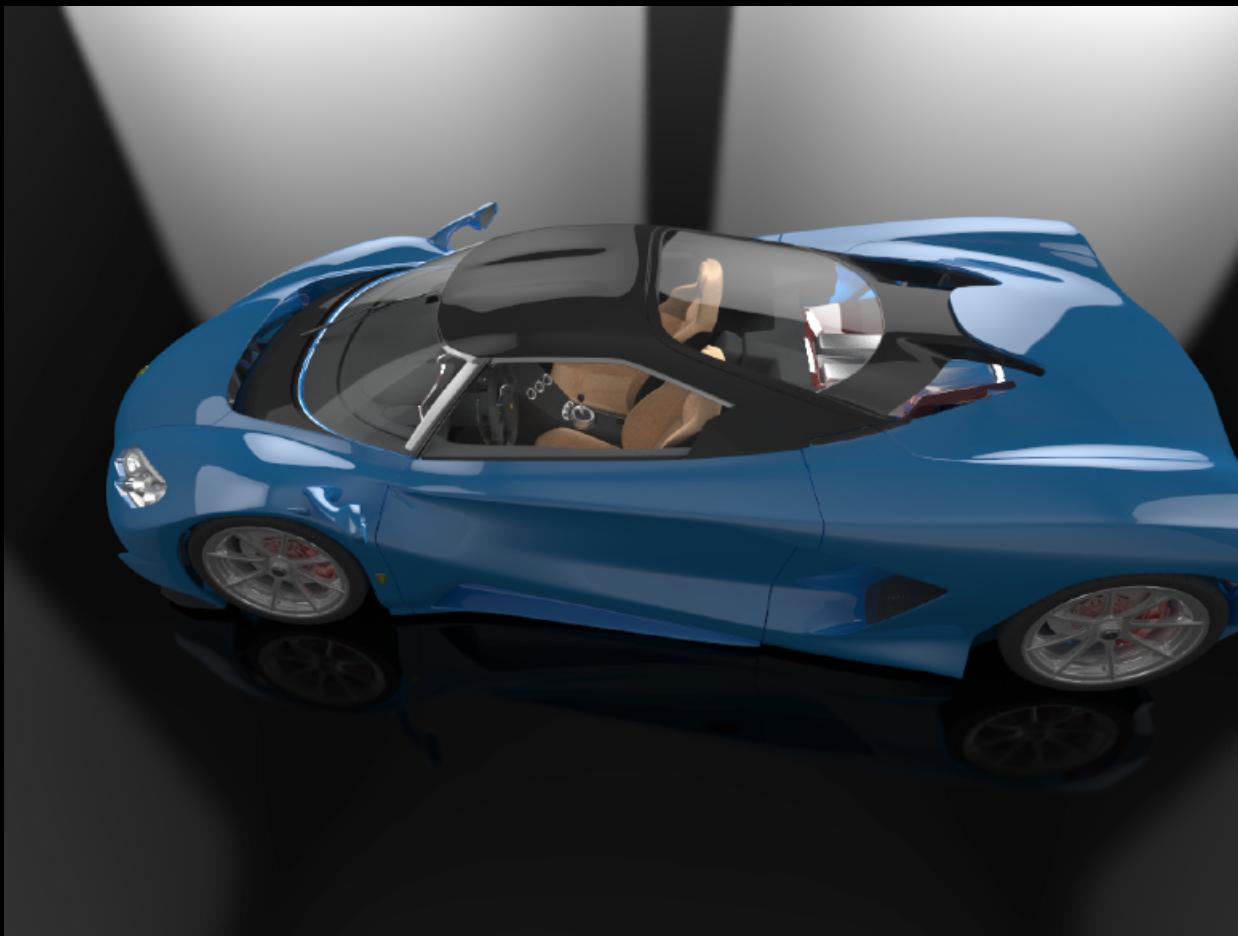
Ray-Implicit Surface Intersection

- Procedural geometry



Ray-Implicit Surface Intersection

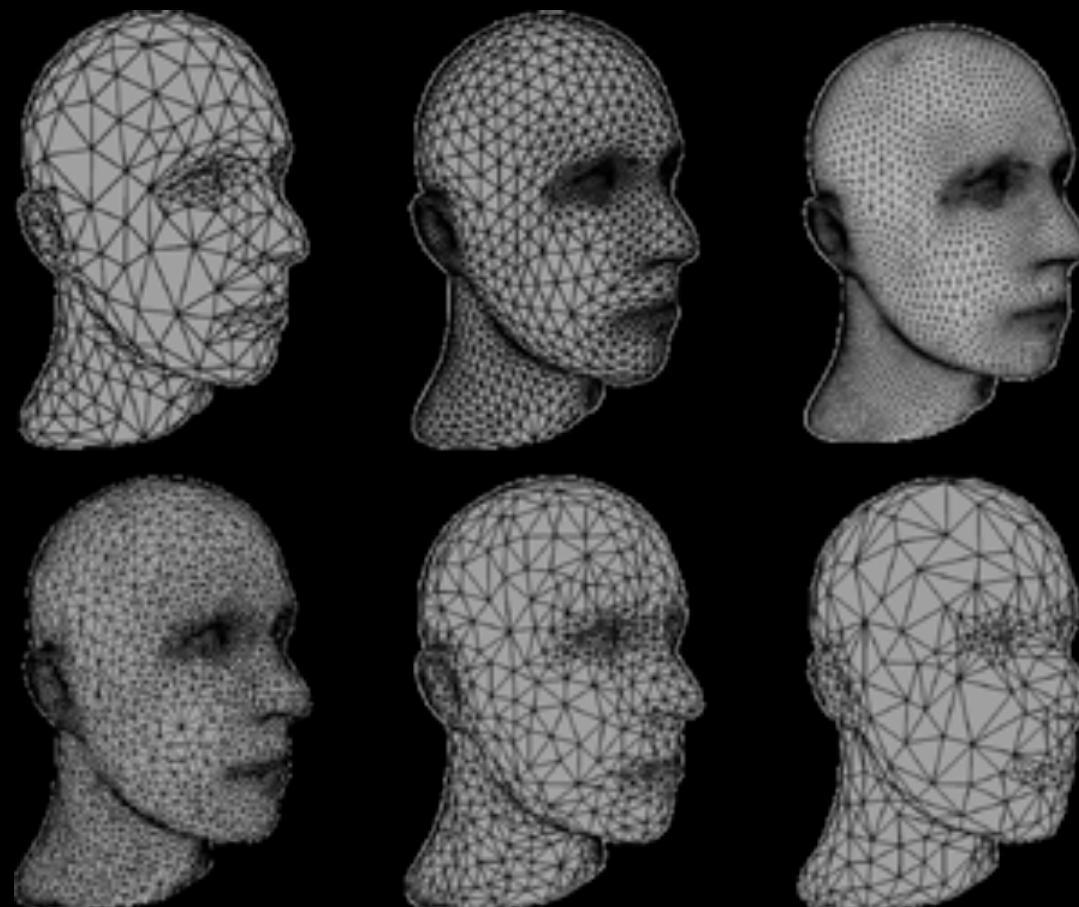
- Subdivision surfaces



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

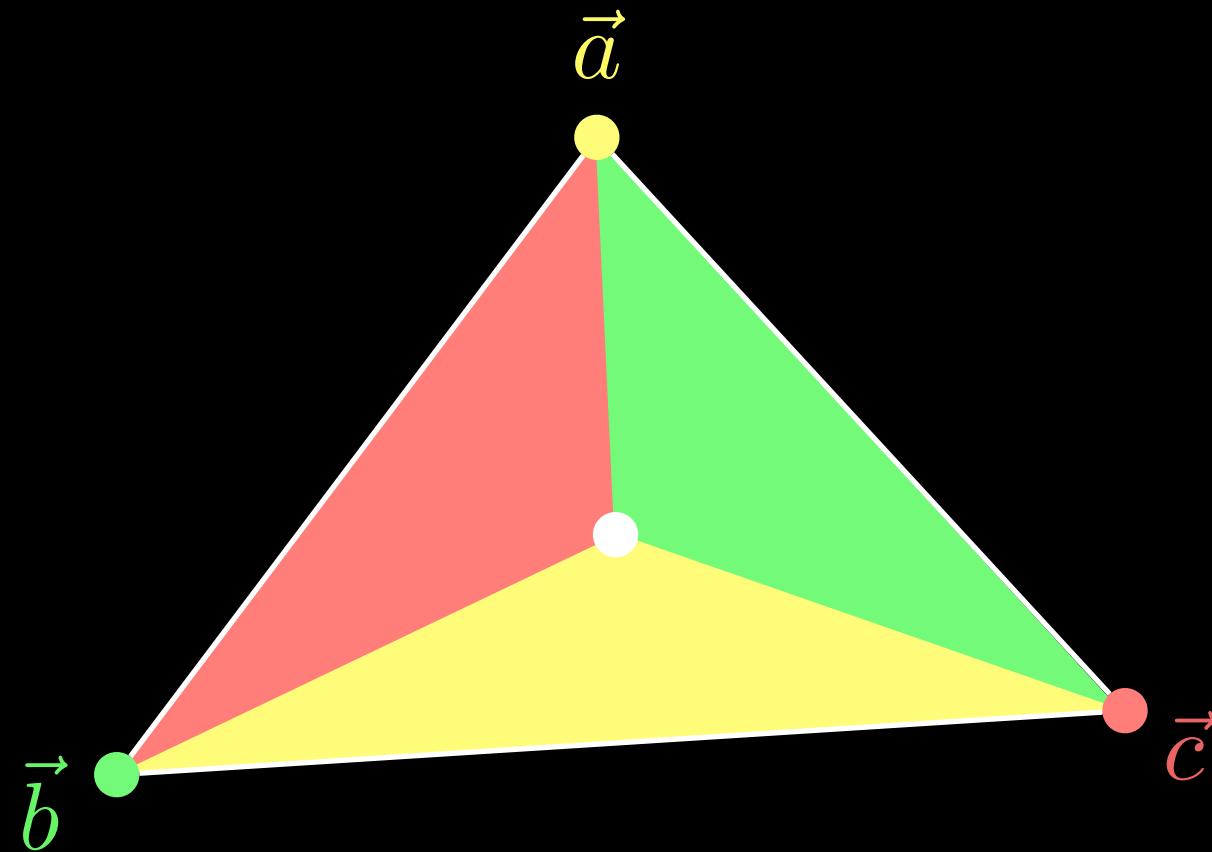
Triangle Mesh

- Approximate shapes with triangles



Barycentric Coordinates

- Ratios of areas of the sub-triangles



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

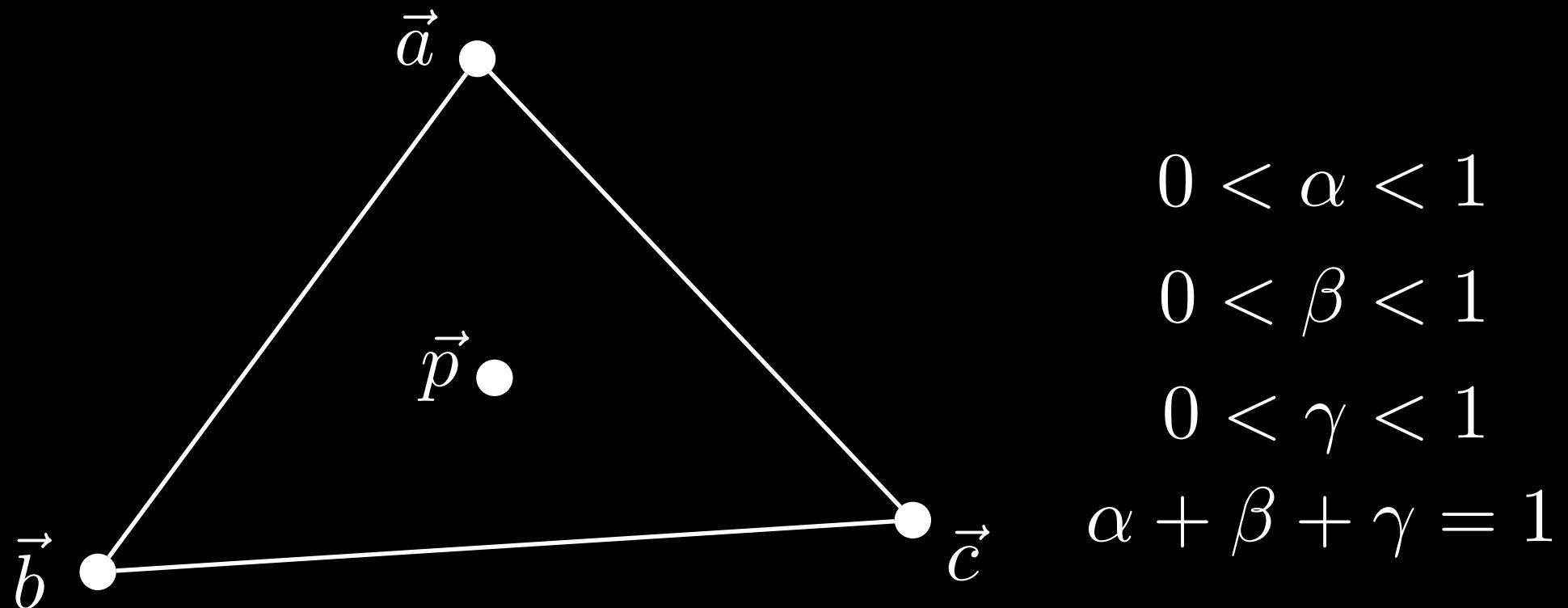
$$\beta = \frac{A_b}{A_a + A_b + A_c}$$

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

Barycentric Coordinates

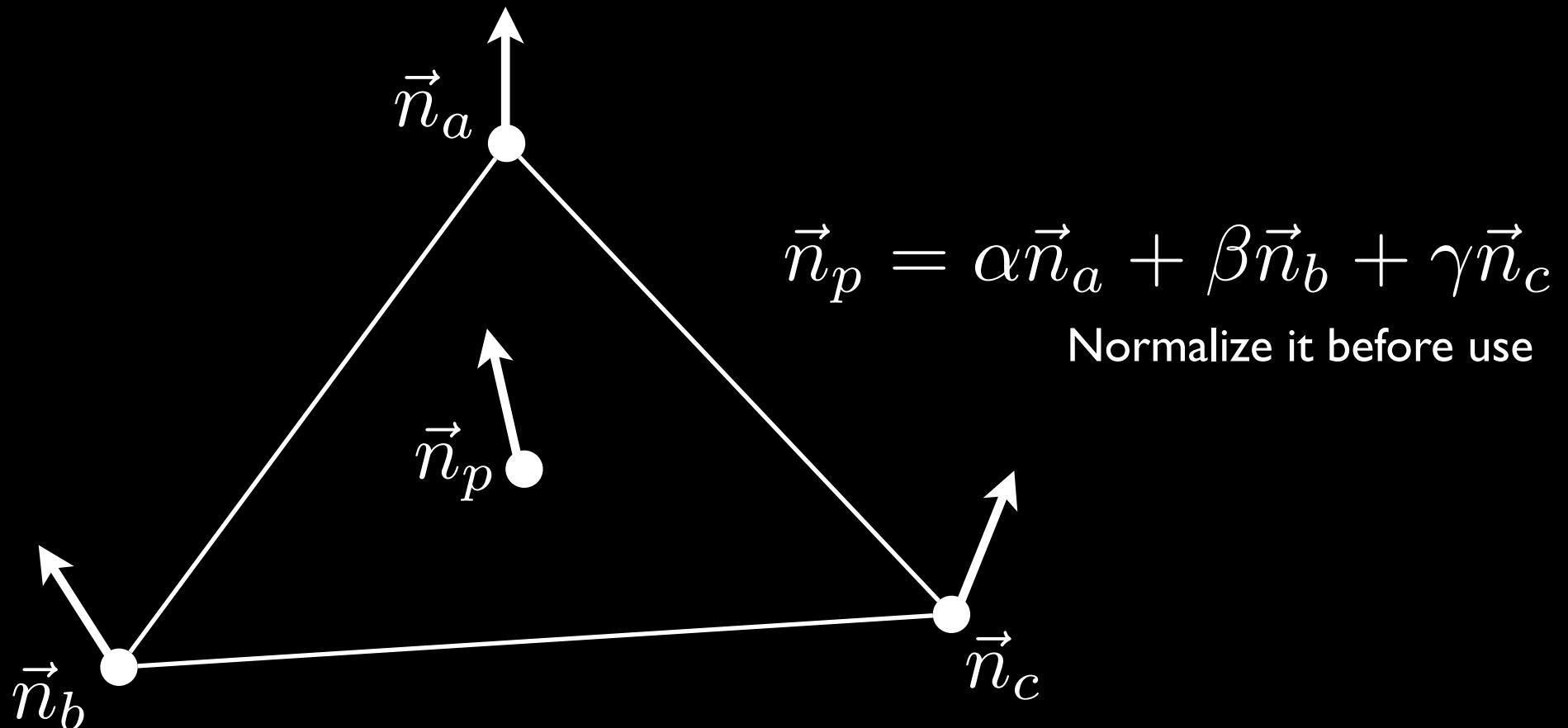
- Parametric description of a point in a triangle

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



Barycentric Coordinates

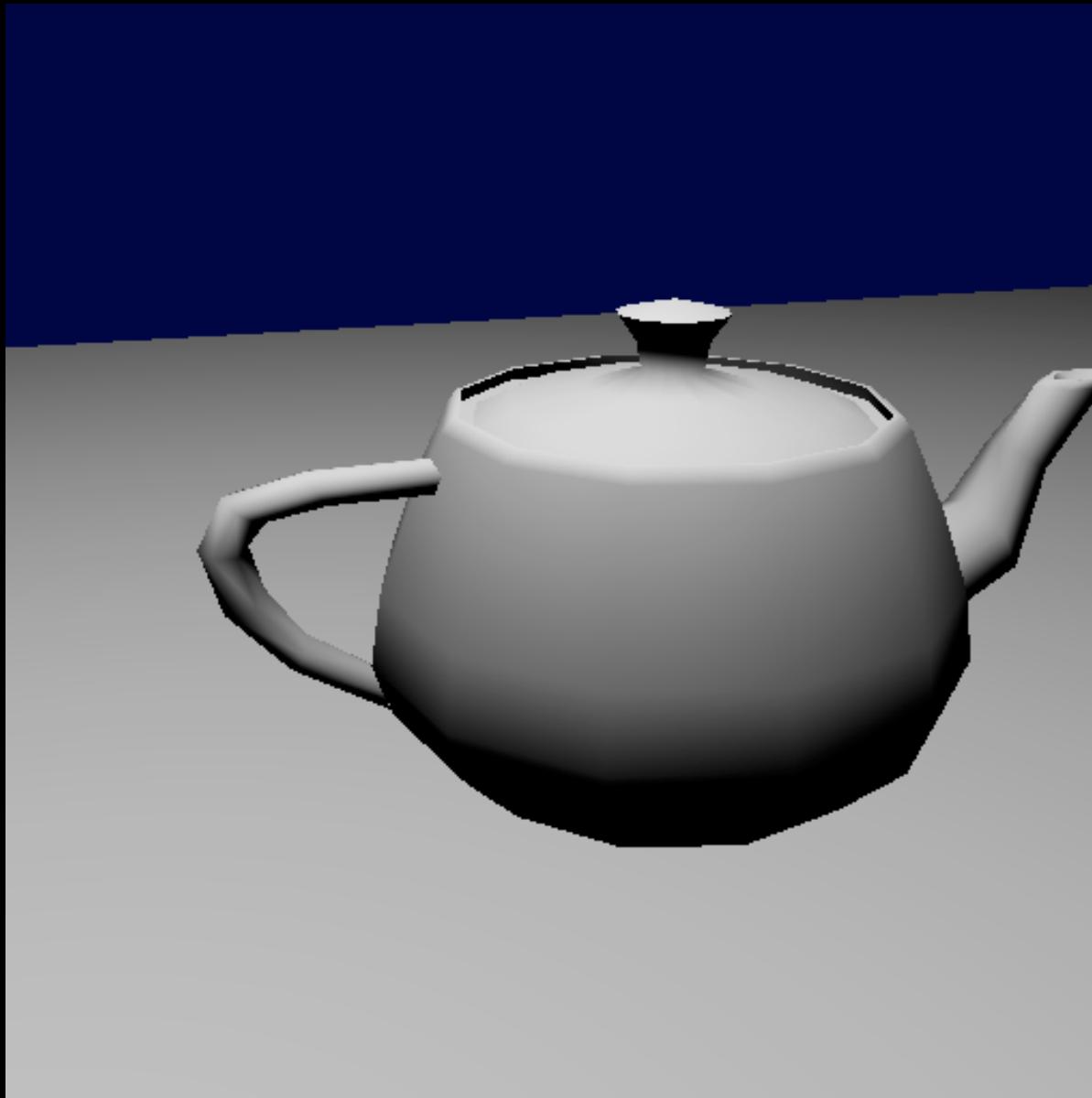
- Interpolate values at the vertices



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

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

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

<http://www.cs.utah.edu/~aek/research/triangle.pdf>

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

hide code

2

compiled successfully

fullscreen

gallery

- 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

```
1 // Copyright (c) 2013 Andrew Baldwin (baldand)
2 // License = Attribution-NonCommercial-ShareAlike (http://creativecommons.org/licenses/by-nc-sa/3.0/deed.en\_US)
3
4 // "Mirror Cube"
5 // A simple ray tracer and timer for GLSLsandbox
6 // Gigatron for glslsandbox ./
7 #ifdef GL_ES
8 precision mediump float;
9#endif
10
11 uniform float time;
12 uniform vec2 mouse;
13 uniform vec2 resolution;
14
15
16
17 const vec3 up = vec3(0.,1.,0.);
18
19
20 float intersectfloor(vec3 ro, vec3 rd, float height, out float t0)
21 {
22     if (rd.y==0.0) {
23         t0 = 100000.0;
24         return 0.0;
25     }
26
27     t0 = -(ro.y + height)/rd.y;
28     t0 = min(100000.0,t0);
29     return t0;
30 }
31
32 float intersectbox(vec3 ro, vec3 rd, float size, out float t0, out float t1, out vec3 normal)
33 // Calculate intersections with origin-centred axis-aligned cube with sides length size
34 // Returns positive value if there are intersections
35 {
36     vec3 ir = 1.0/rd;
37     vec3 tb = ir * (vec3(-size*.5)-ro);
```

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 )  
}
```