3D Rendering and Ray Casting

Michael Kazhdan

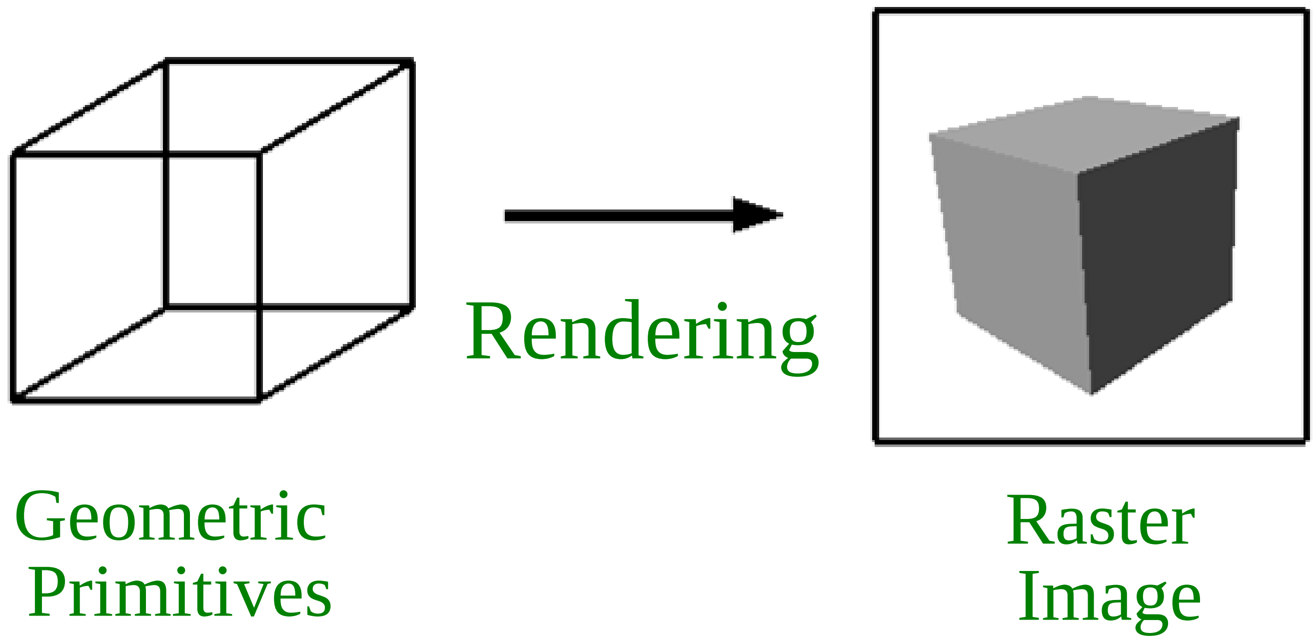
(600.357 / 600.457)

HB Ch. 13.7, 14.6

FvDFH 15.5, 15.10

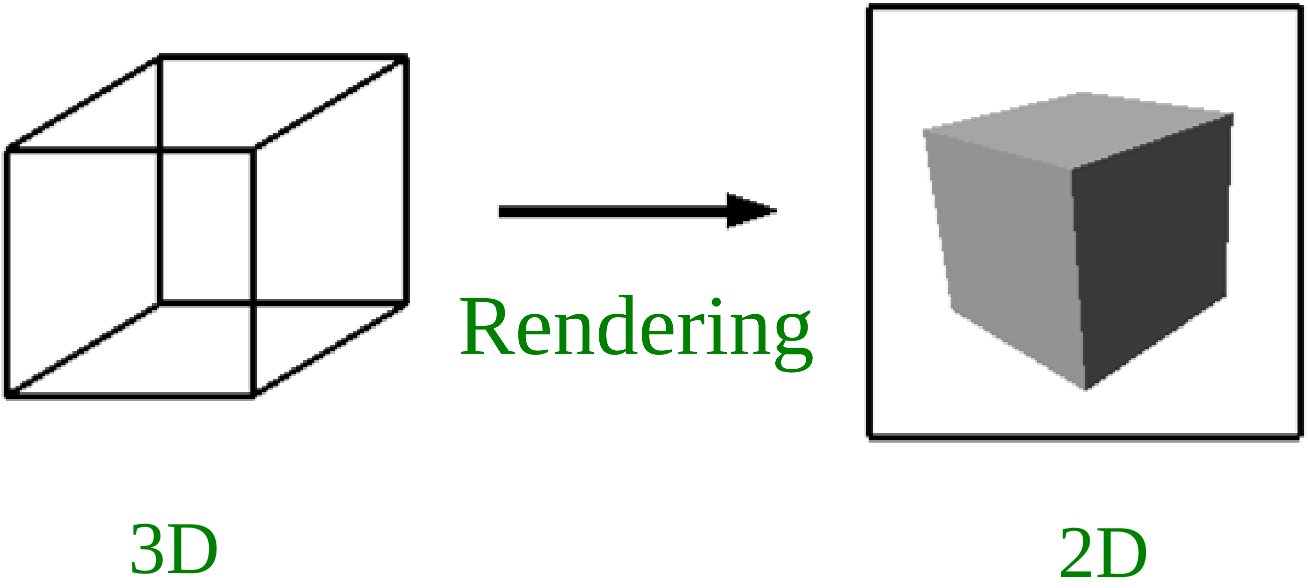
Rendering

- Generate an image from geometric primitives

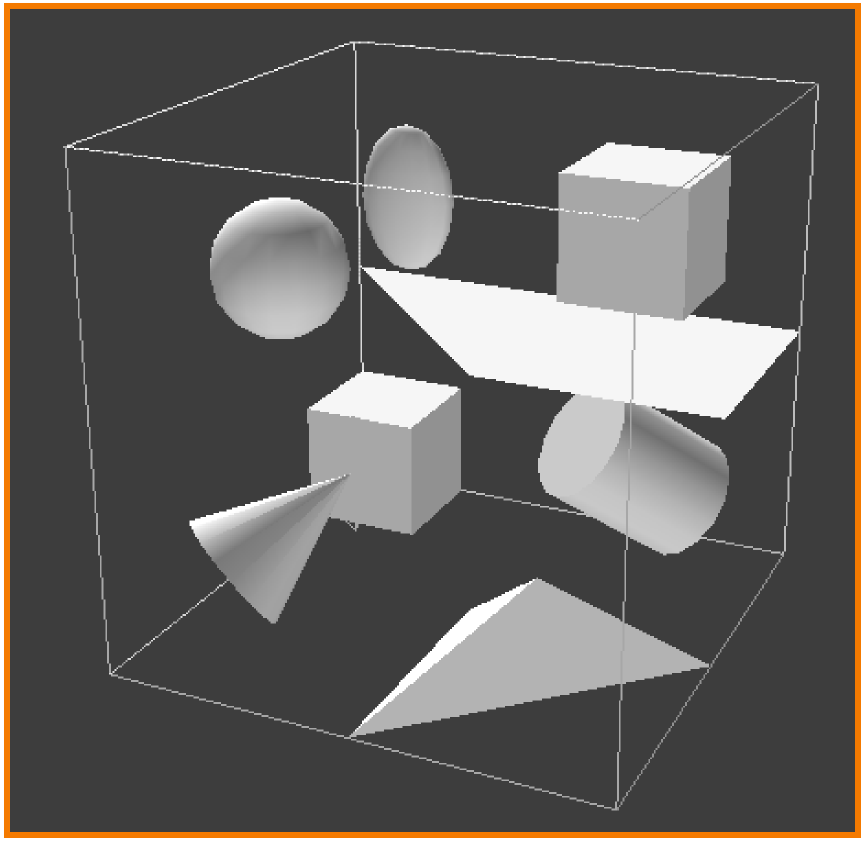


Rendering

- Generate an image from geometric primitives



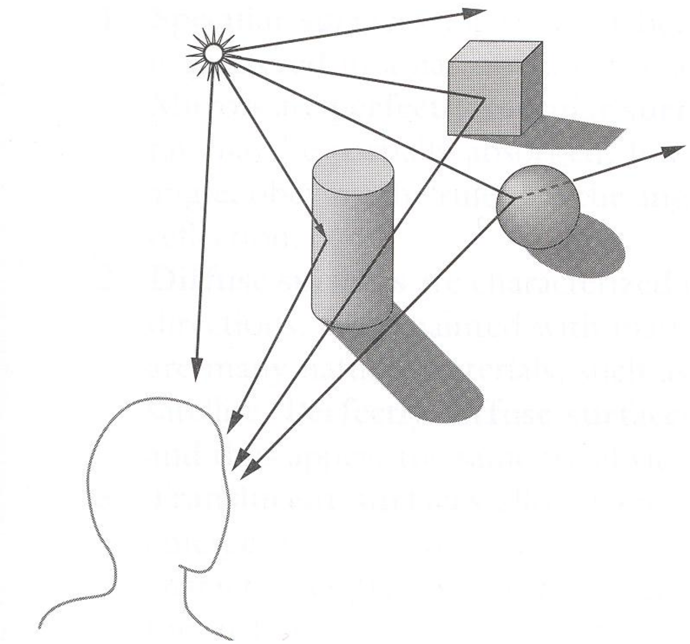
3D Rendering Example



What issues must be addressed by a
3D rendering system?

Overview

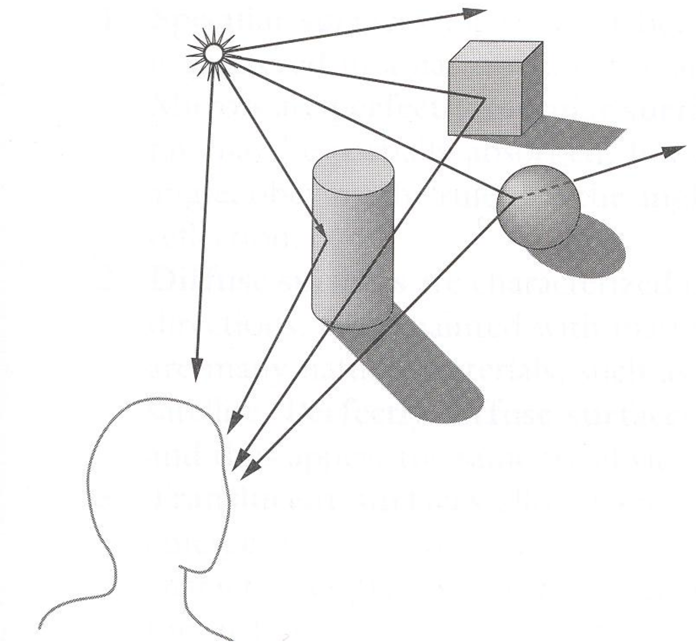
- 3D scene representation
- 3D viewer representation
- What do we see?
- How does it look?



Overview

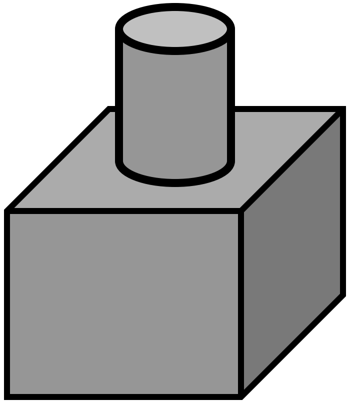
- 3D scene representation
- 3D viewer representation
- What do we see?
- How does it look?

How is the 3D scene described in a computer?



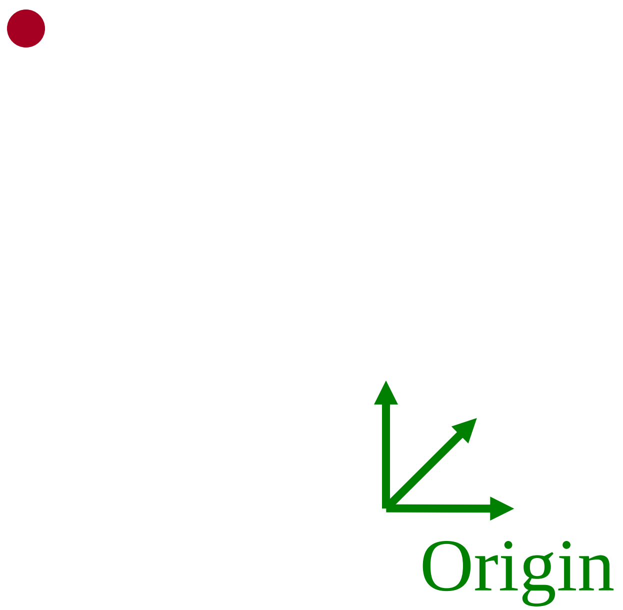
3D Scene Representation

- Scene is usually approximated by 3D primitives
  - Point
  - Line segment
  - Polygon
  - Polyhedron
  - Curved surface
  - Solid object
  - etc.



3D Point

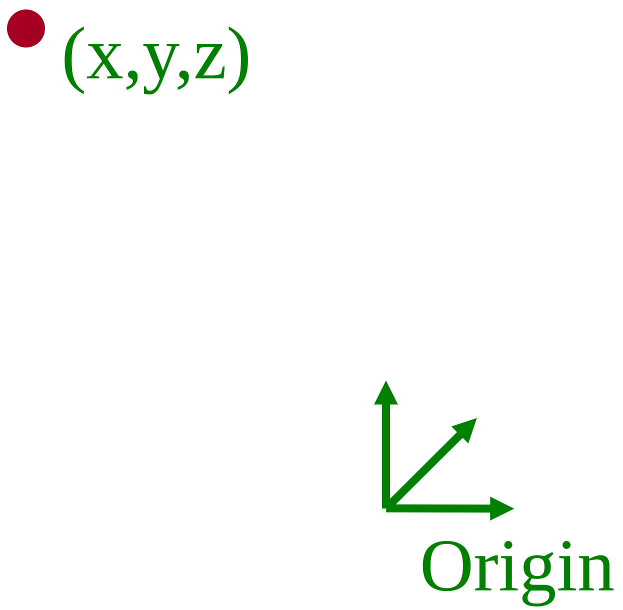
- Specifies a location



3D Point

- Specifies a location
  - Represented by three coordinates
  - Infinitely small

```
class Point3D
{
public:
    Coordinate x;
    Coordinate y;
    Coordinate z;
};
```



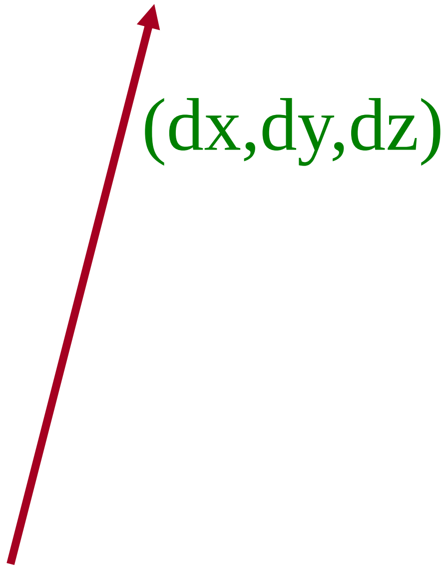
3D Vector

- Specifies a direction and a magnitude

3D Vector

- Specifies a direction and a magnitude
  - Represented by three coordinates
  - Magnitude ||V|| = sqrt(dx dx + dy dy + dz dz)
  - Has no location

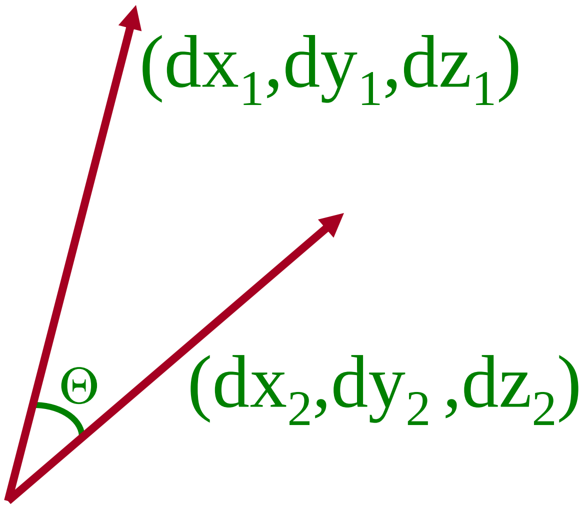
```
class Vector3D
{
public:
    Coordinate dx;
    Coordinate dy;
    Coordinate dz;
};
```



3D Vector

- Specifies a direction and a magnitude
  - Represented by three coordinates
  - Magnitude ||V|| = sqrt(dx dx + dy dy + dz dz)
  - Has no location

```
class Vector3D
{
public:
    Coordinate dx;
    Coordinate dy;
    Coordinate dz;
};
```

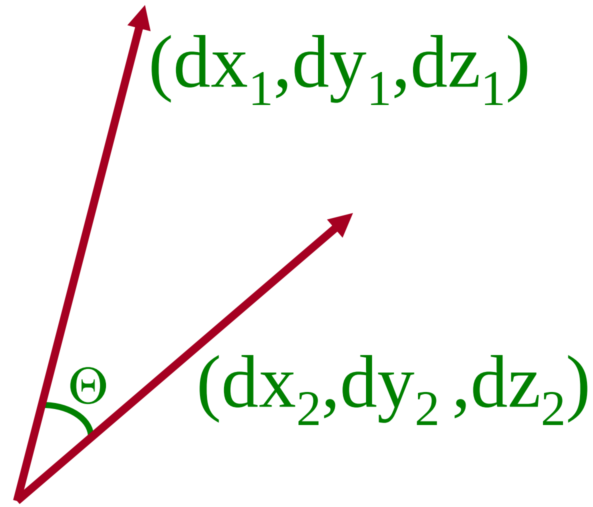


- Dot product of two 3D vectors
  - $V_1 \cdot V_2 = dx_1 dx_2 + dy_1 dy_2 + dz_1 dz_2$
  - $V_1 \cdot V_2 = ||V_1|| \, || V_2 || \cos(\Theta)$

3D Vector

- Specifies a direction and a magnitude
  - Represented by three coordinates
  - Magnitude ||V|| = sqrt(dx dx + dy dy + dz dz)
  - Has no location

```
class Vector3D
{
public:
    Coordinate dx;
    Coordinate dy;
    Coordinate dz;
};
```



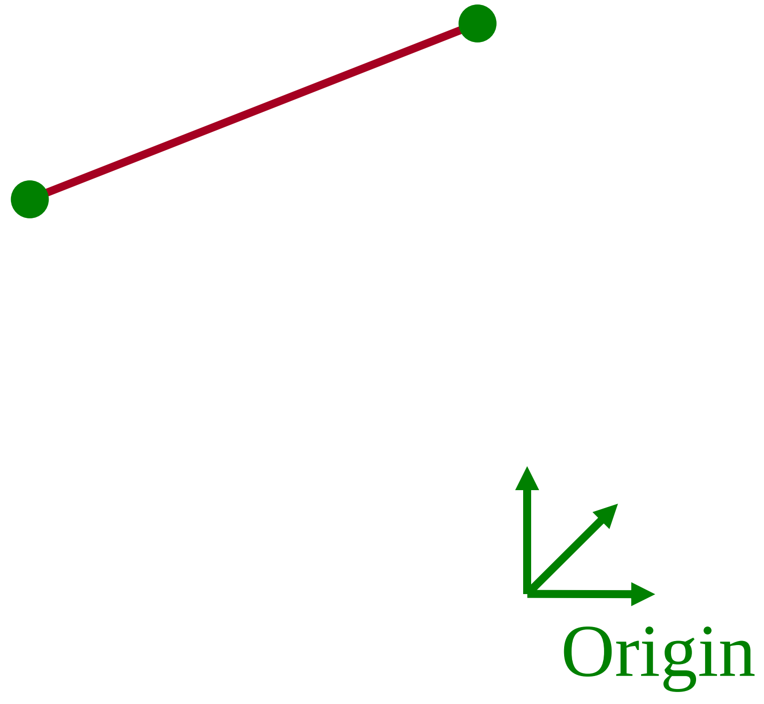
- Cross product of two 3D vectors
  - $V_1 \times V_2$ = Vector normal to plane V_1 , V_2
  - $|| V_1 \times V_2 || = ||V_1|| \, || V_2|| \sin(\Theta)$

Cross Product: Review

- Let $U = V \times W$:
  - $U_x = V_y W_z - V_z W_y$
  - $U_y = V_z W_x - V_x W_z$
  - $U_z = V_x W_y - V_y W_x$
- $V \times W = - W \times V$ (remember “right-hand” rule)
- We can do similar derivations to show:
  - $V_1 \times V_2 = \|V_1\| \, \|V_2\| \sin(\Theta)\, n$, where n is unit vector normal to V_1 and V_2
  - $\| V_1 \times V_1 \| = 0$
- http://physics.syr.edu/courses/java-suite/crosspro.html

3D Line Segment

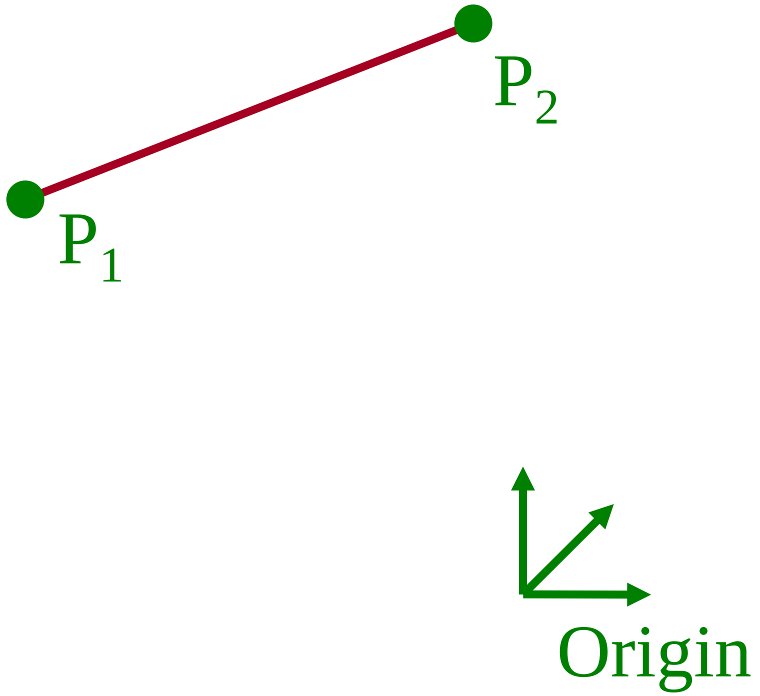
- Linear path between two points



3D Line Segment

- Use a linear combination of two points
  - Parametric representation:
    - » $P = P_1 + t\,(P_2 - P_1)$, $(0 \leq t \leq 1)$

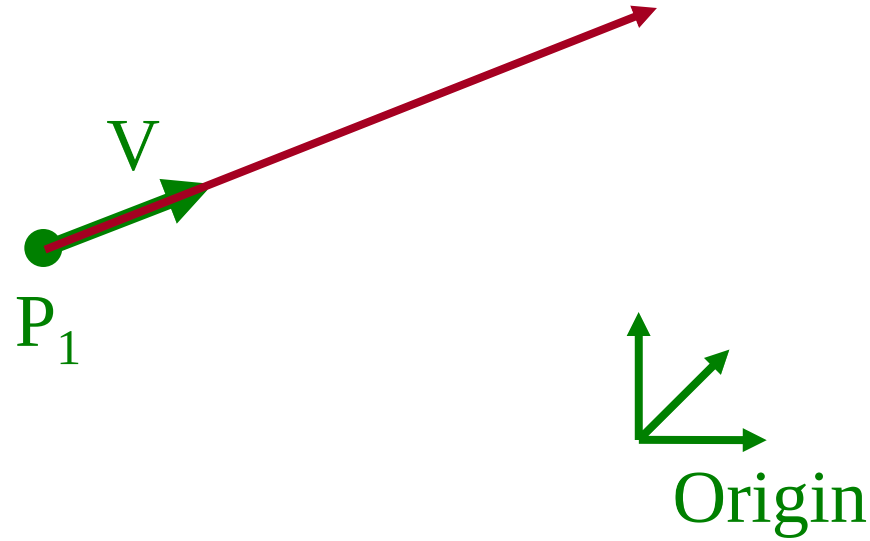
```
class Segment3D
{
public:
    Point3D P1;
    Point3D P2;
};
```



3D Ray

- Line segment with one endpoint at infinity
  - Parametric representation:
    - » $P = P_1 + t\,V, \quad (0 <= t < \infty)$

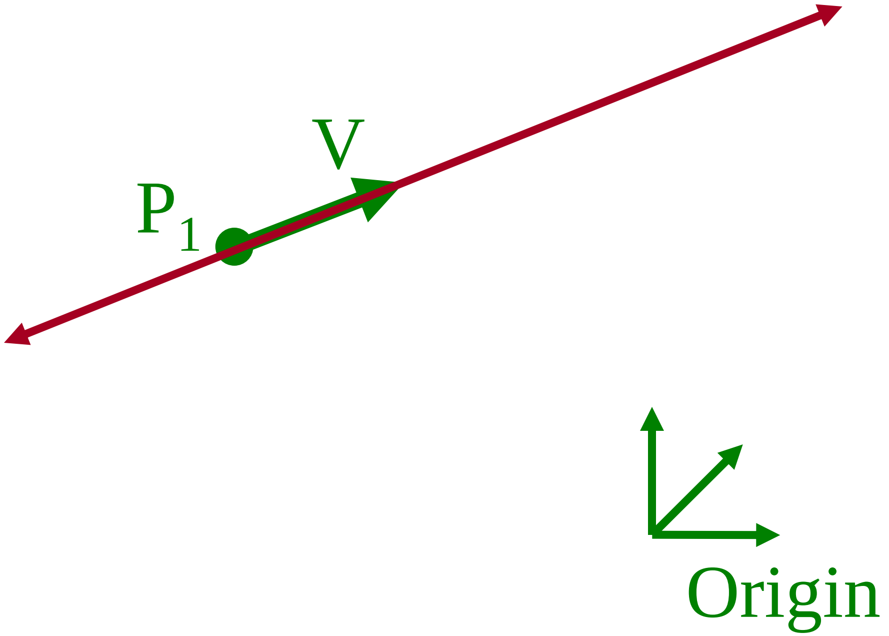
```
class Ray3D
{
public:
    Point3D P1;
    Vector3D V;
};
```



3D Line

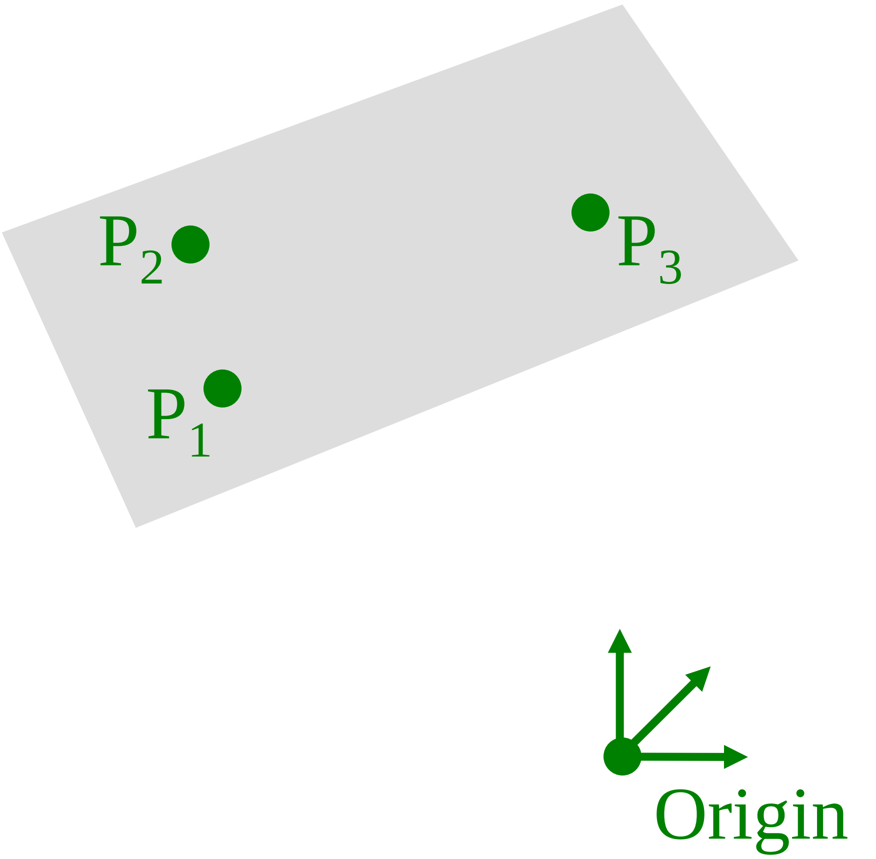
- Line segment with both endpoints at infinity
  - Parametric representation:
    - » $P = P_1 + t\,V, \quad (-\infty < t < \infty)$

```
class Line3D
{
public:
    Point3D P1;
    Vector3D V;
};
```



3D Plane

- A linear combination of three points

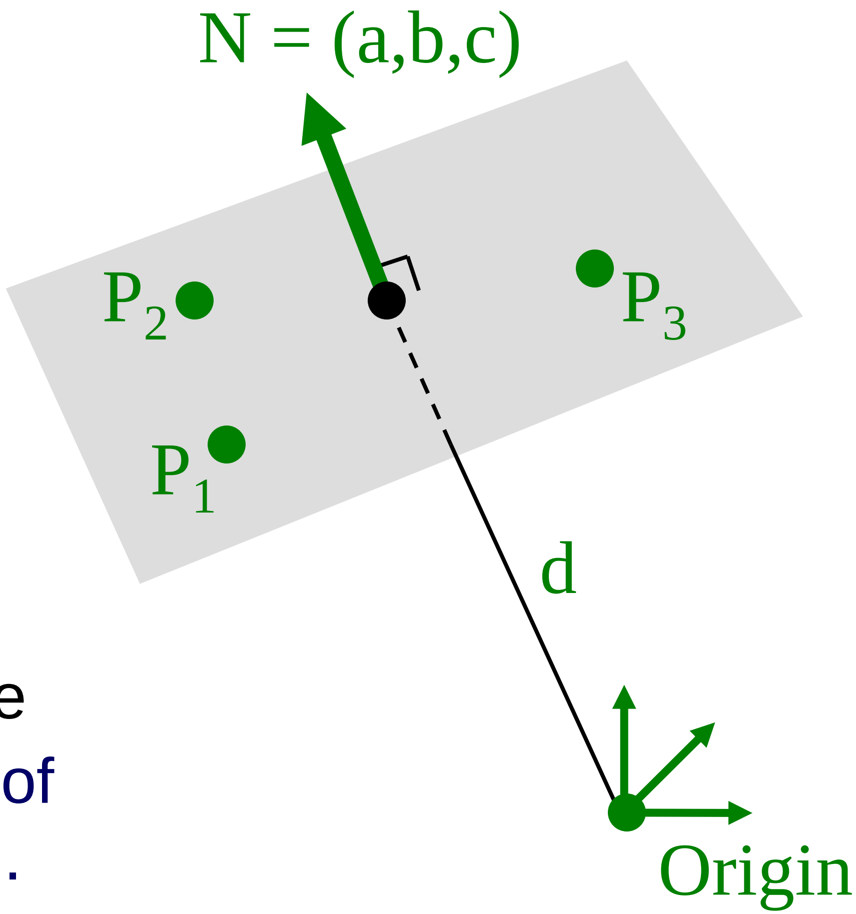


3D Plane

- A linear combination of three points
  - Implicit representation:
    - » P·N - d = 0, or
    - » ax + by + cz - d = 0

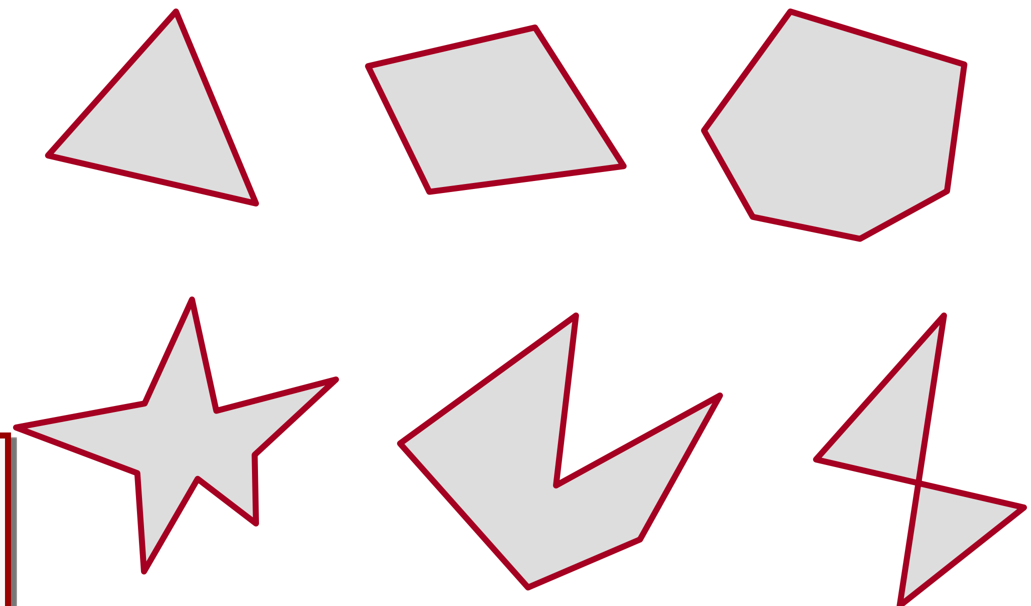
```
class Plane3D
{
public:
    Vector3D N;
    Distance d;
};
```

  - N is the plane "normal"
    - » Unit-length vector
    - » Perpendicular to plane
  - d is the signed distance of the plane from the origin.



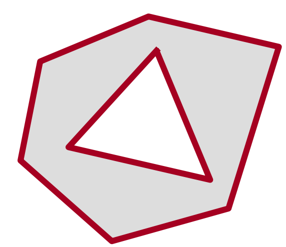
3D Polygon

- Area “inside” a sequence of coplanar points
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting



```
class Polygon3D
{
public:
      Point3D *points;
      int npoints;
};
```

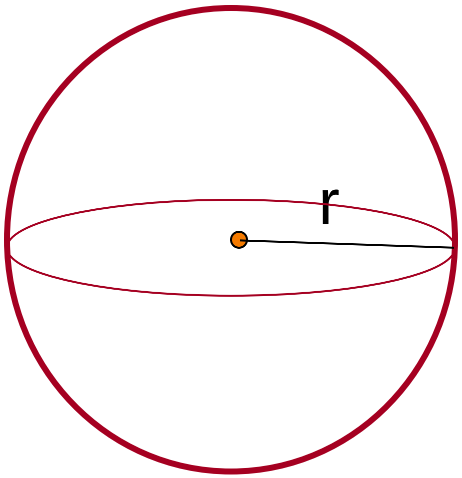
Points are in counter-clockwise order



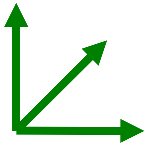
  - Holes (use > 1 polygon struct)

3D Sphere

- All points at distance "r" from point "(c_x, c_y, c_z)"
  - Implicit representation:
    - » $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$
  - Parametric representation:
    - » $x = r \cos(\phi) \cos(\Theta) + c_x$
    - » $y = r \cos(\phi) \sin(\Theta) + c_y$
    - » $z = r \sin(\phi) + c_z$




```
class Sphere3D
{
public:
    Point3D center;
    Distance radius;
};
```



Origin

Other 3D primitives

- Cone
- Cylinder
- Ellipsoid
- Box
- Etc.

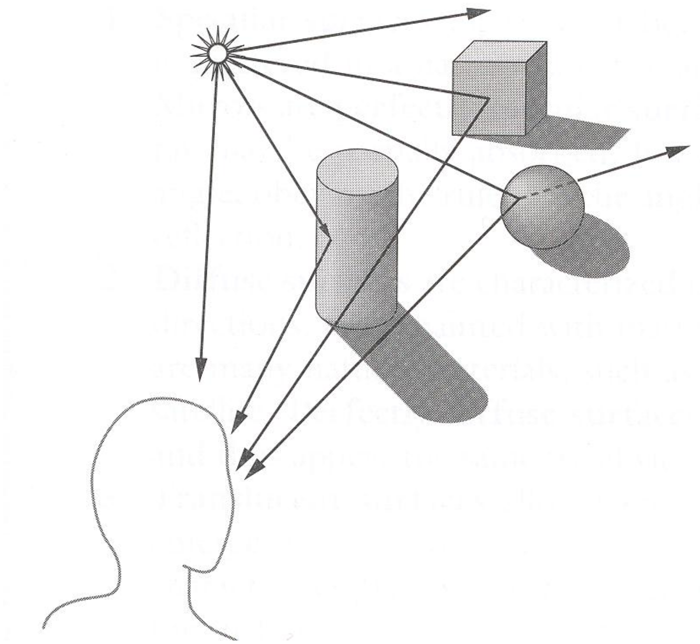
3D Geometric Primitives

- More detail on 3D modeling later in course
  - Point
  - Line segment
  - Polygon
  - Polyhedron
  - Curved surface
  - Solid object
  - etc.

Overview

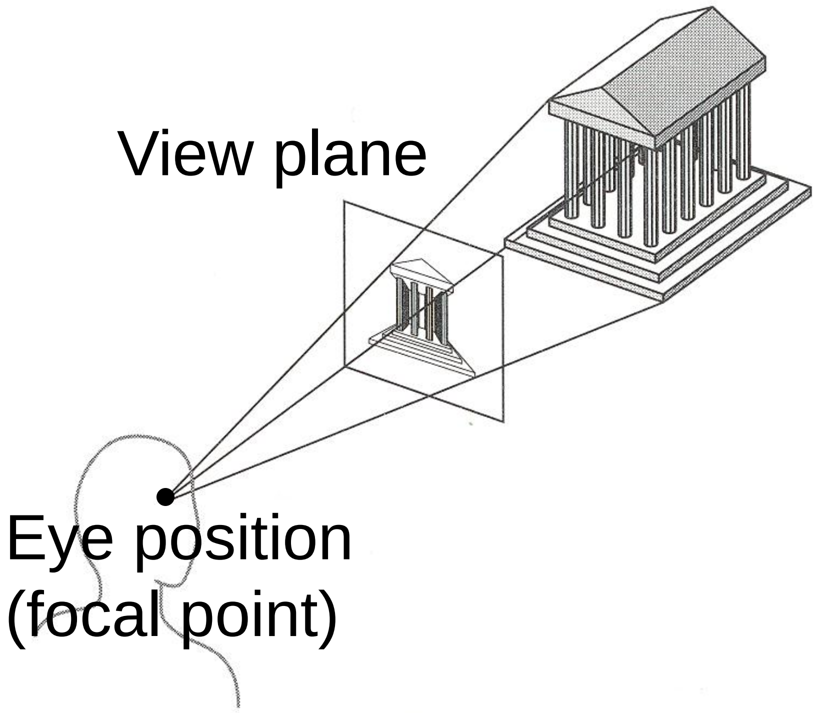
- 3D scene representation
- 3D viewer representation
- What do we see?
- How does it look?

How is the viewing device described in a computer?



Camera Models

- The most common model is pin-hole camera
  - All captured light rays arrive along paths toward focal point without lens distortion (everything is in focus)



Other models consider ...

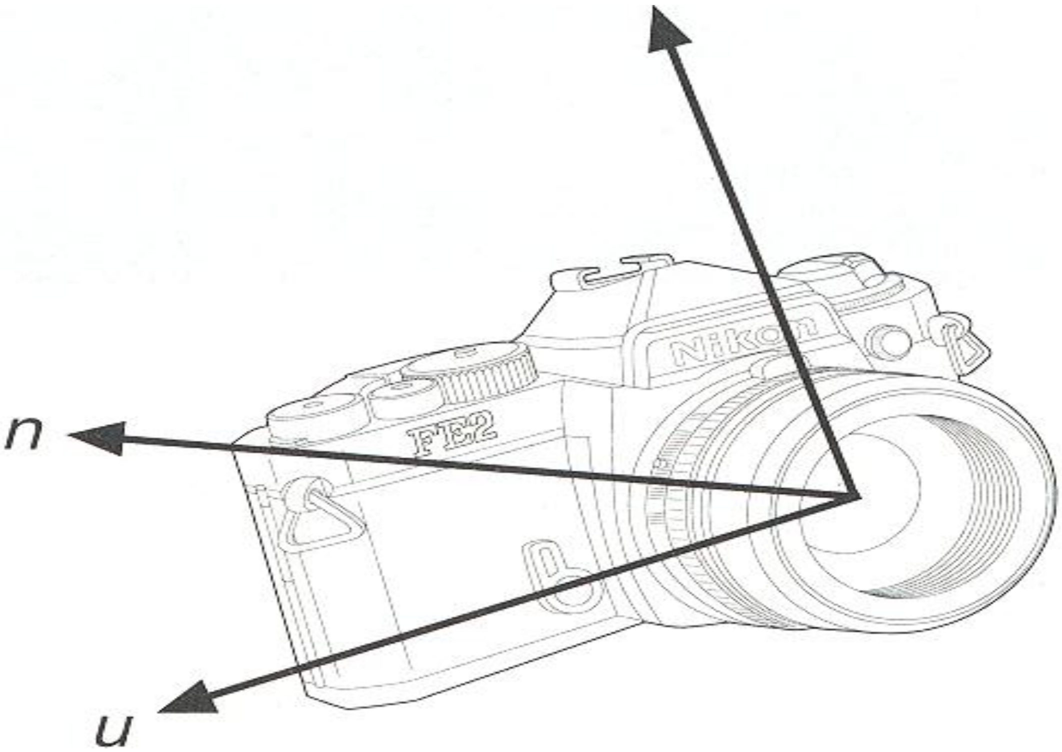
Depth of field

Motion blur

Lens distortion

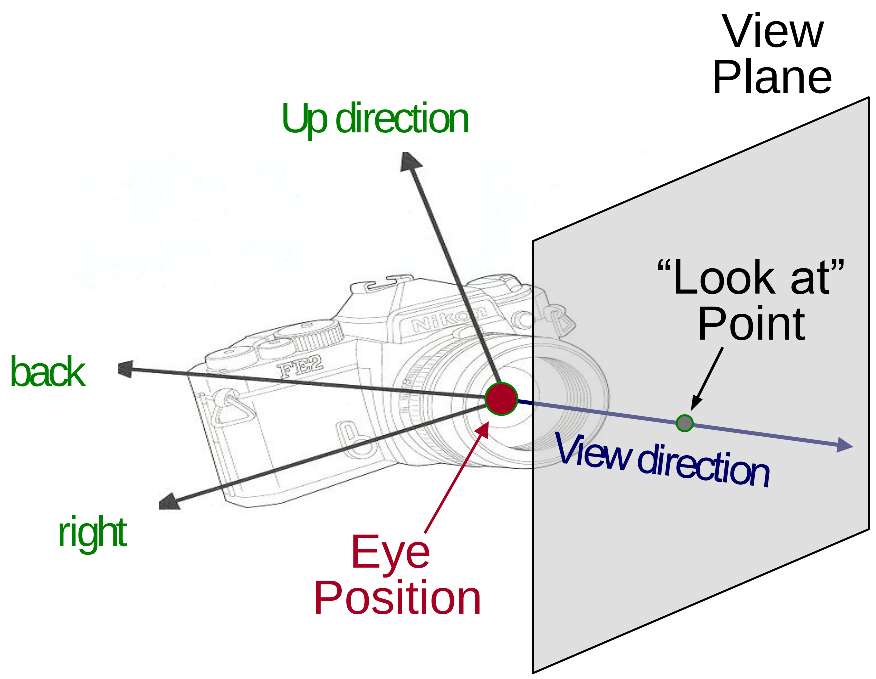
Camera Parameters

- What are the parameters of a camera?



Camera Parameters

- Position
  - Eye position (px, py, pz)
- Orientation
  - View direction (dx, dy, dz)
  - Up direction (ux, uy, uz)
- Aperture
  - Field of view (xfov, yfov)
- Film plane
  - “Look at” point
  - View plane normal



Other Models: Depth of Field

Close Focused

Distance Focused

P. Haeberli

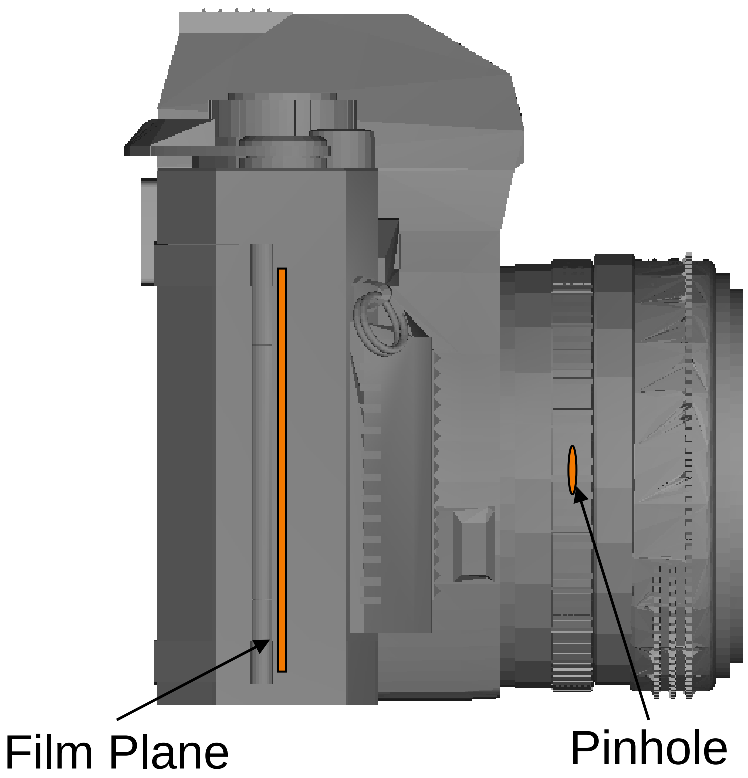
Other Models: Motion Blur

- Mimics effect of open camera shutter
- Gives perceptual effect of high-speed motion
- Generally involves temporal super-sampling

Brostow & Essa

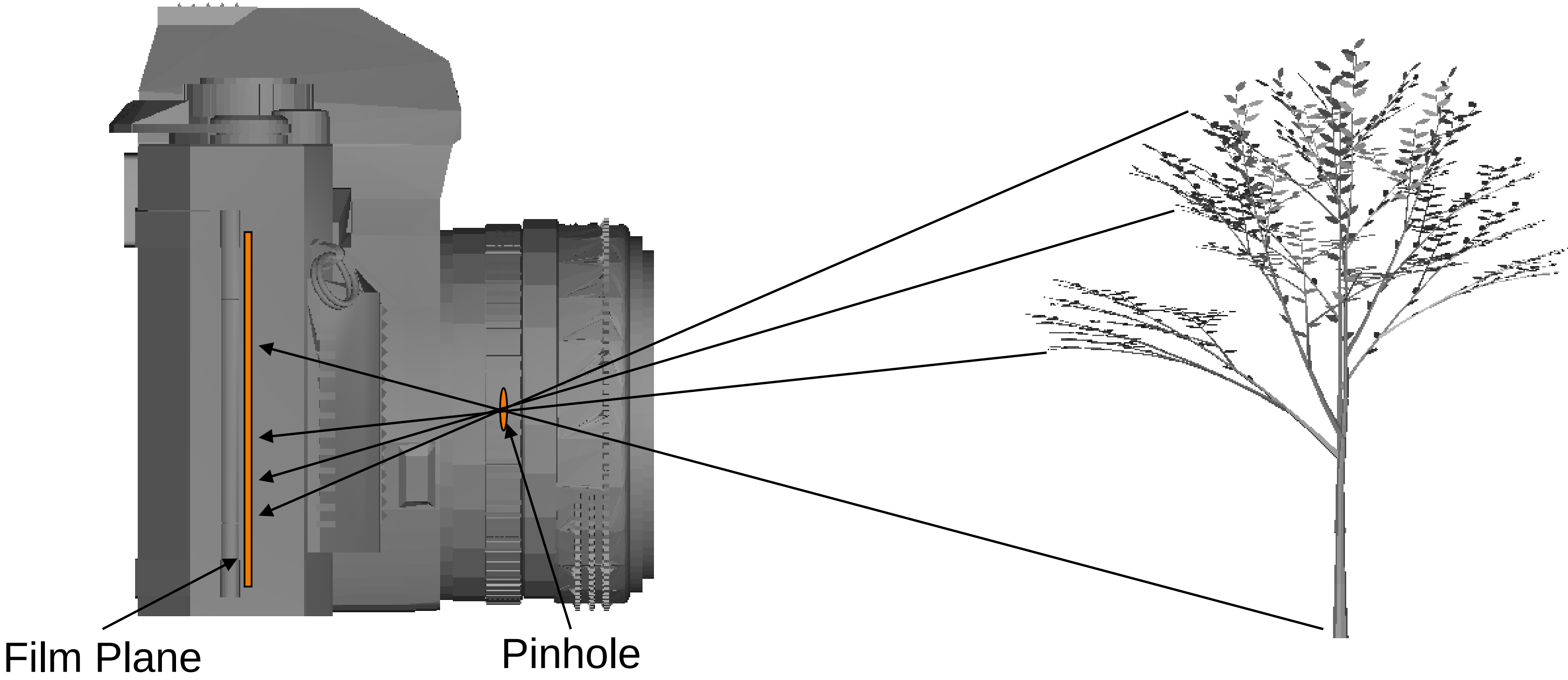
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.



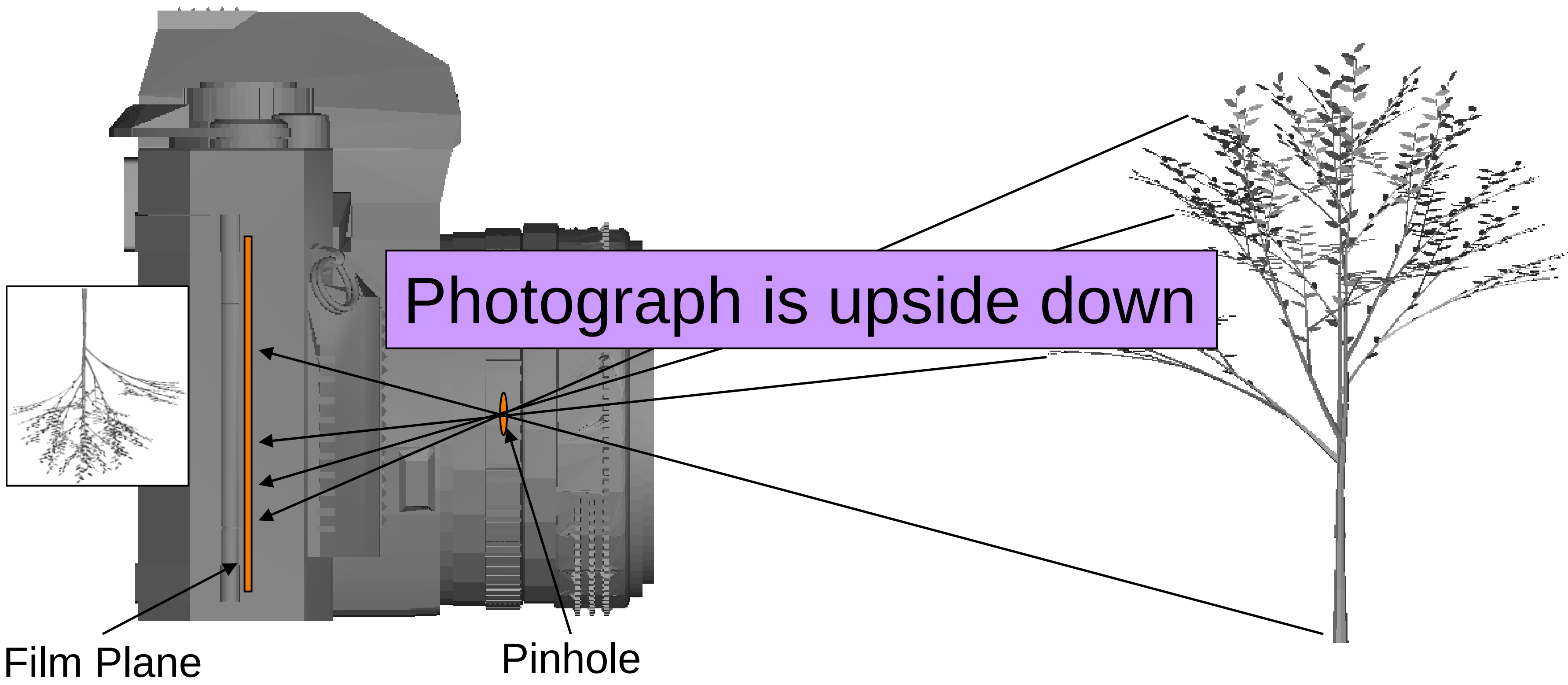
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
- Rays come in from the outside, pass through the pinhole, and hit the film plane.



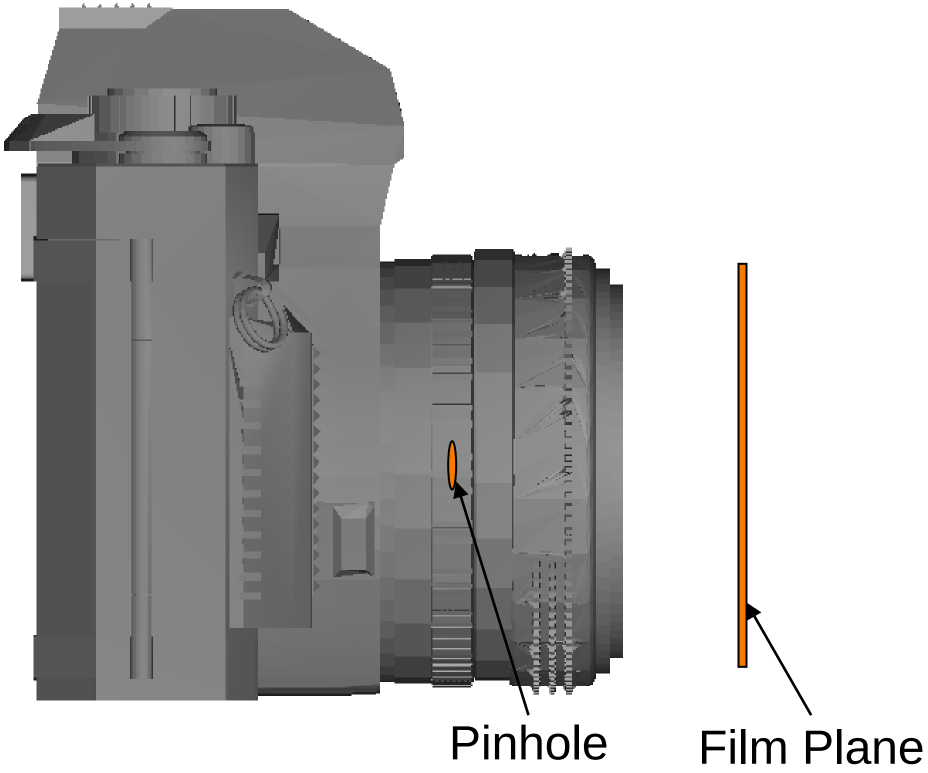
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
- Rays come in from the outside, pass through the pinhole, and hit the film plane.



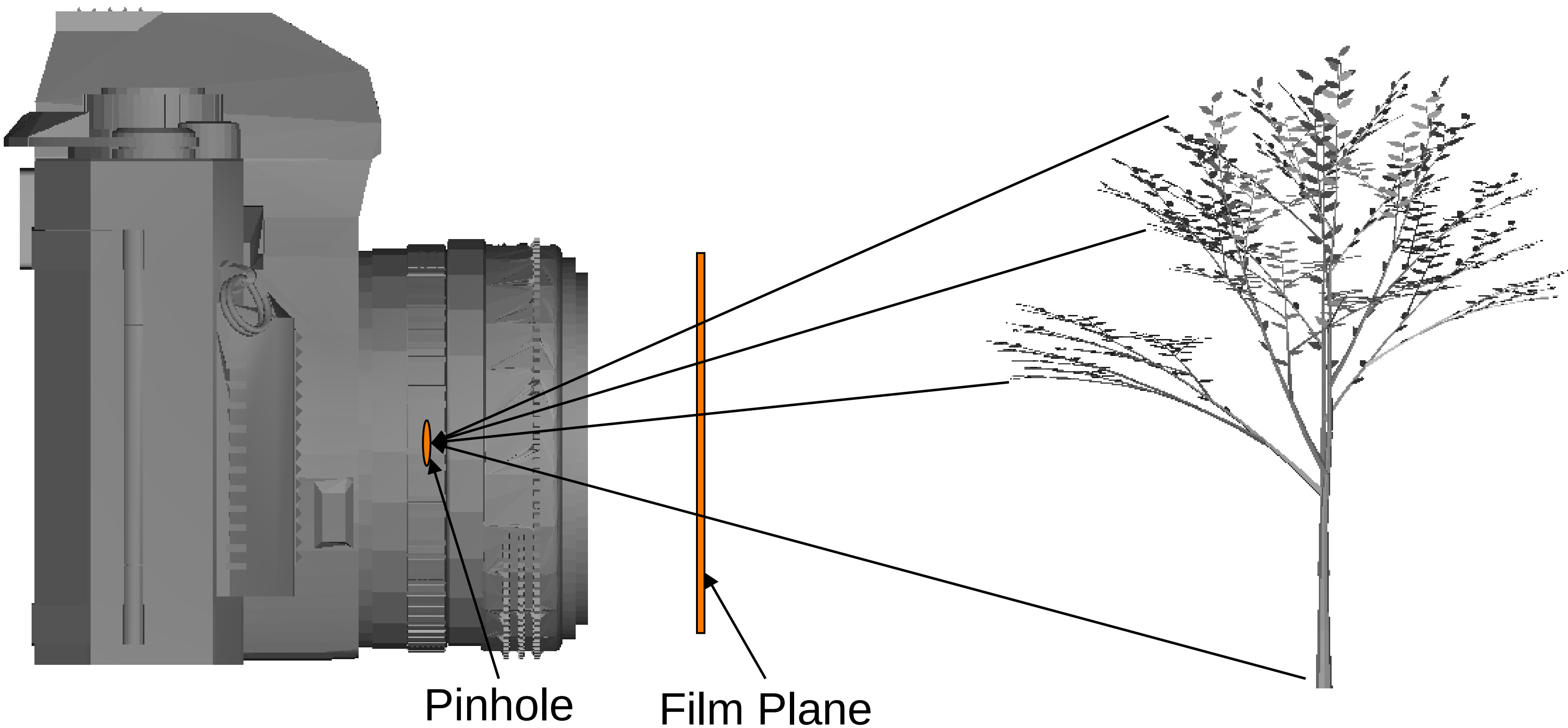
Virtual Camera

- The film sits in front of the pinhole of the camera.



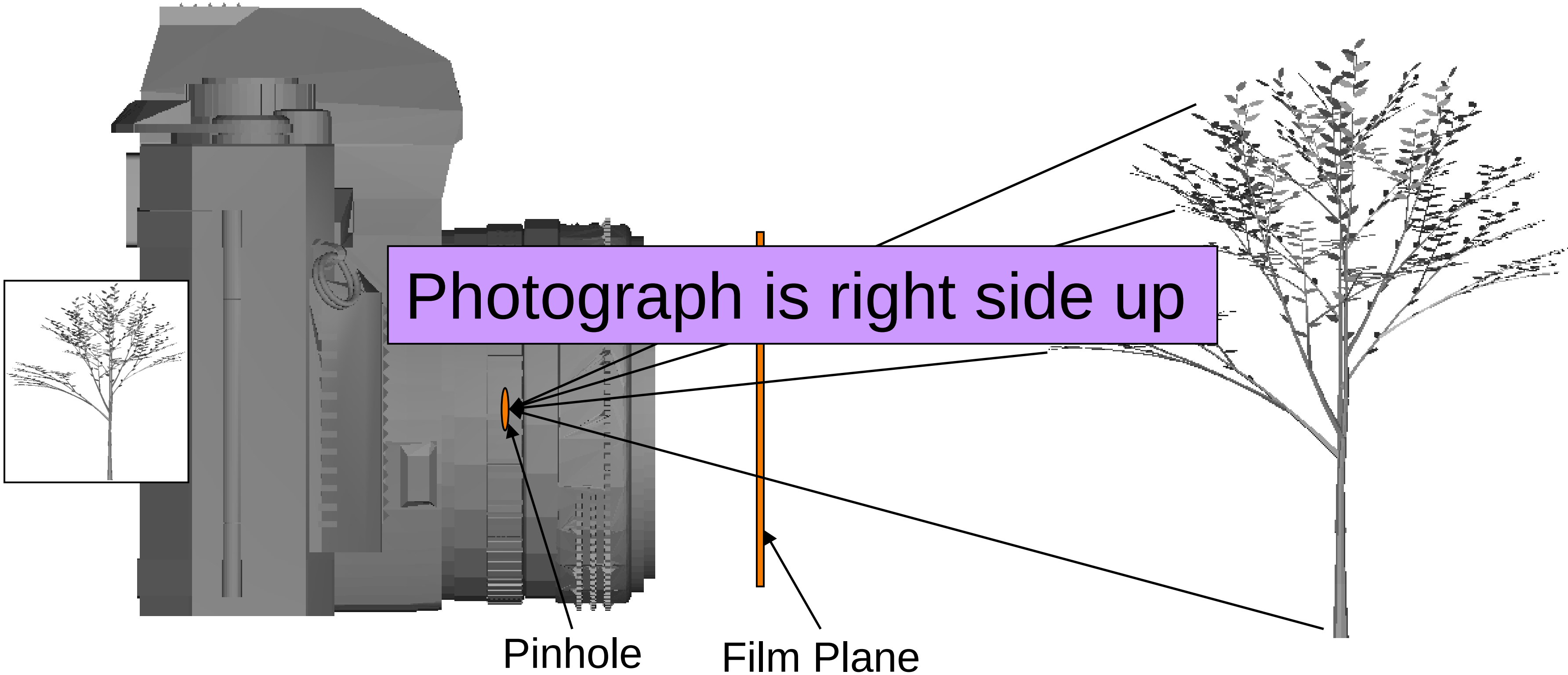
Virtual Camera

- The film sits in front of the pinhole of the camera.
- Rays come in from the outside, pass through the film plane, and hit the pinhole.



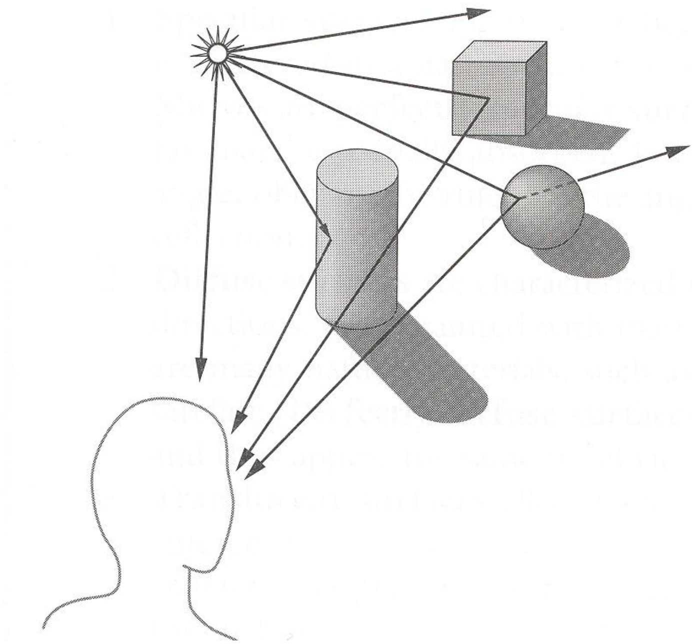
Virtual Camera

- The film sits in front of the pinhole of the camera.
- Rays come in from the outside, pass through the film plane, and hit the pinhole.



Overview

- 3D scene representation
- 3D viewer representation
- Ray Casting
  - What do we see?
  - How does it look?

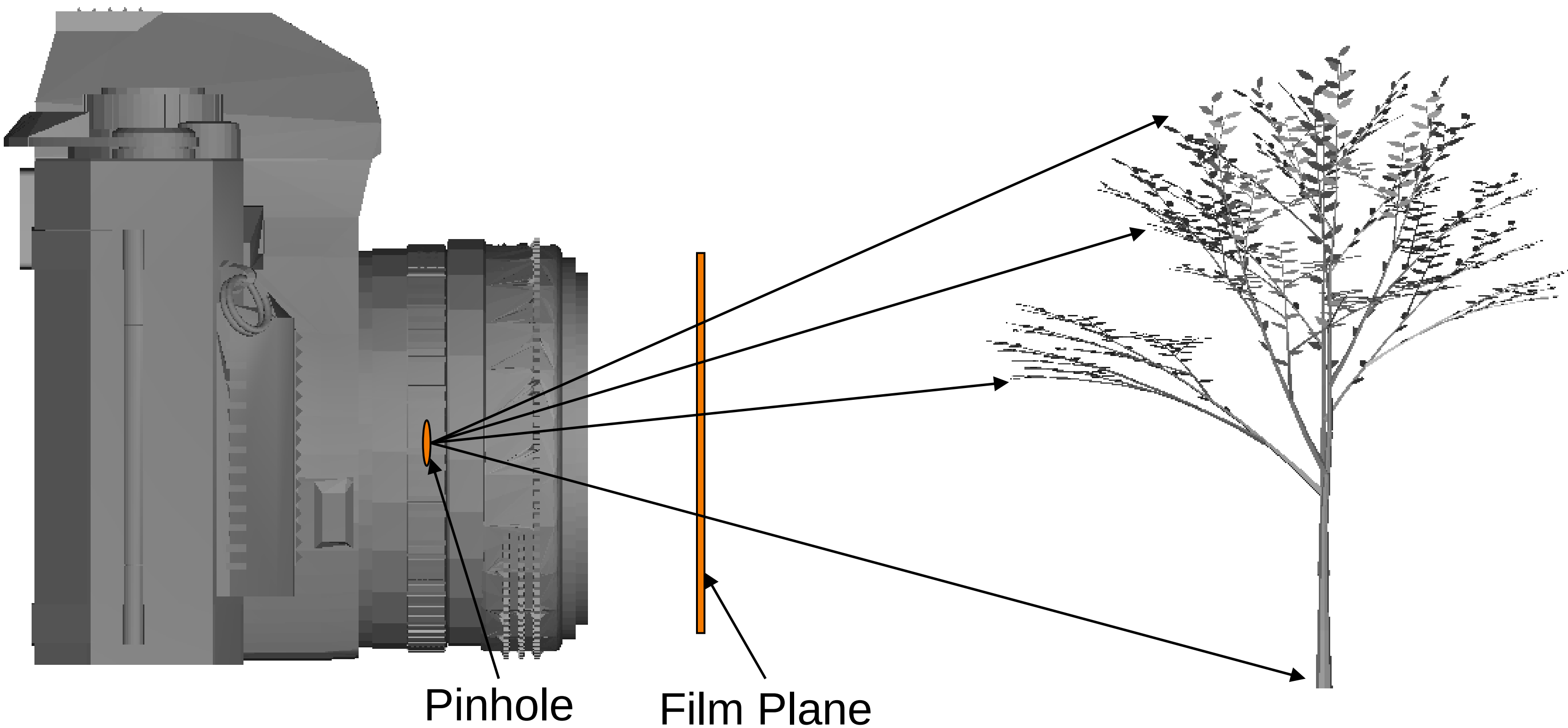


Ray Casting

- Rendering model
- Intersections with geometric primitives
  - Sphere
  - Triangle
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees

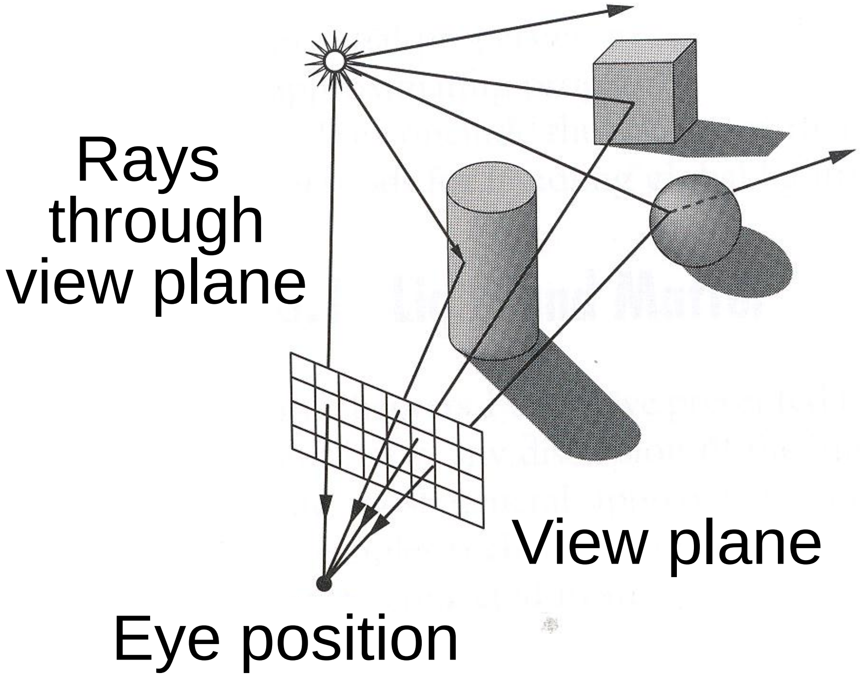
Ray Casting

- We invert the process of image generation by sending rays out from the pinhole, and then we find the first intersection of the ray with the scene.



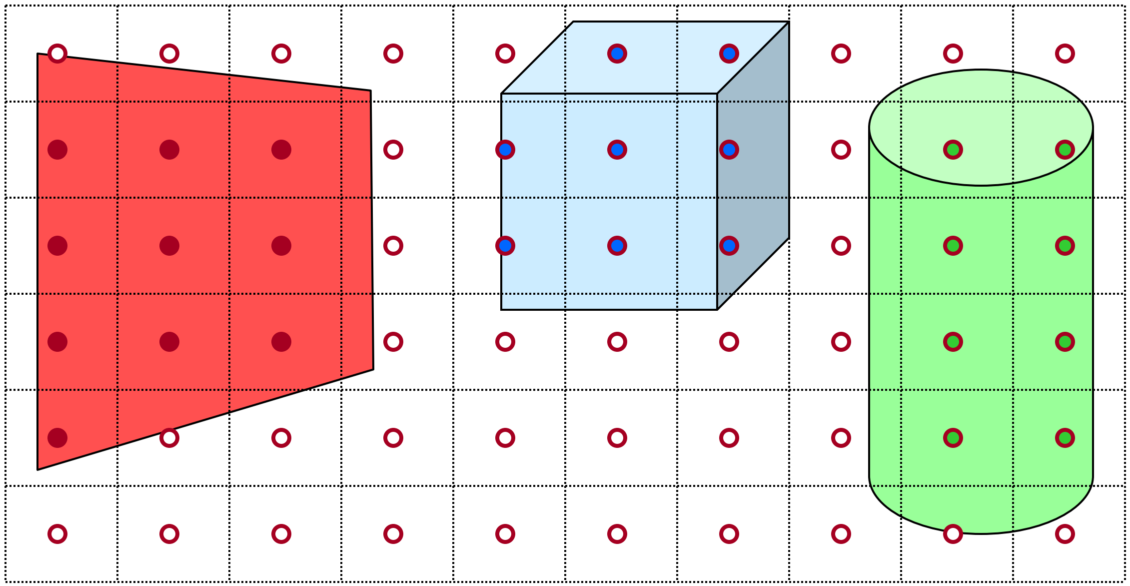
Ray Casting

- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces



Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find <u>first</u> surface intersected by ray through pixel
  - Compute color sample based on surface radiance



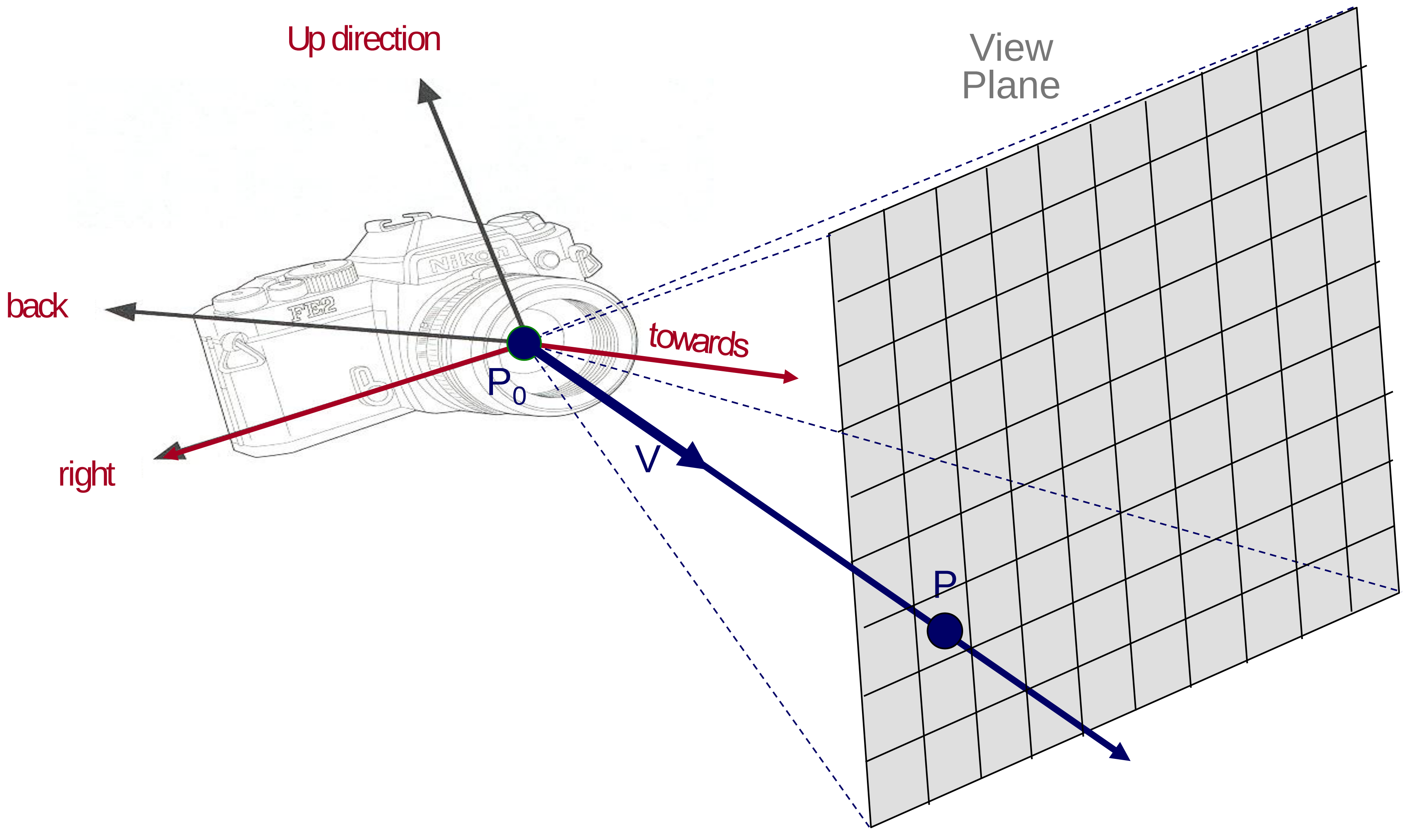
Ray Casting

- Simple implementation:

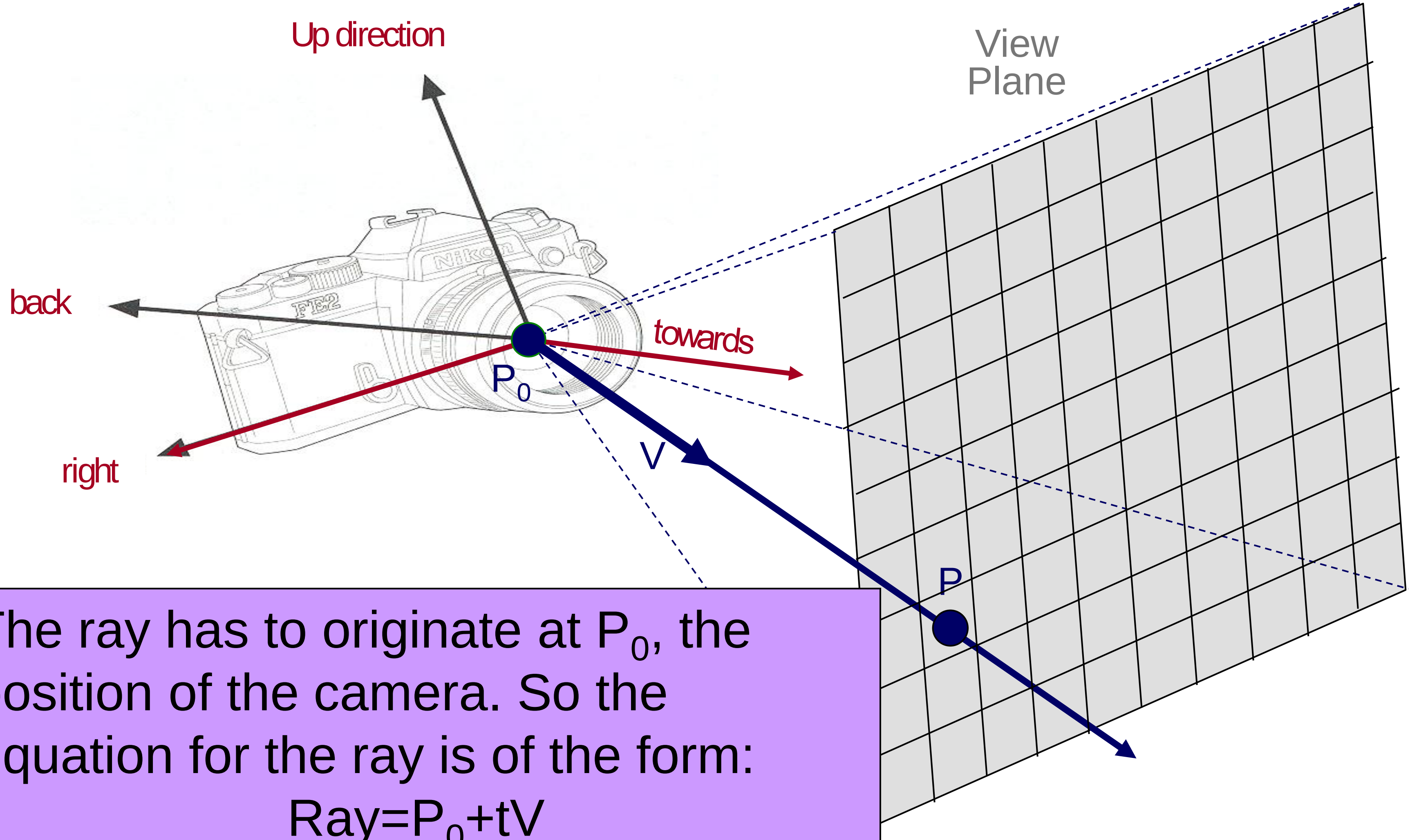
```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

- Where are we looking?
- What are we seeing?
- What does it look like?

Constructing a Ray Through a Pixel



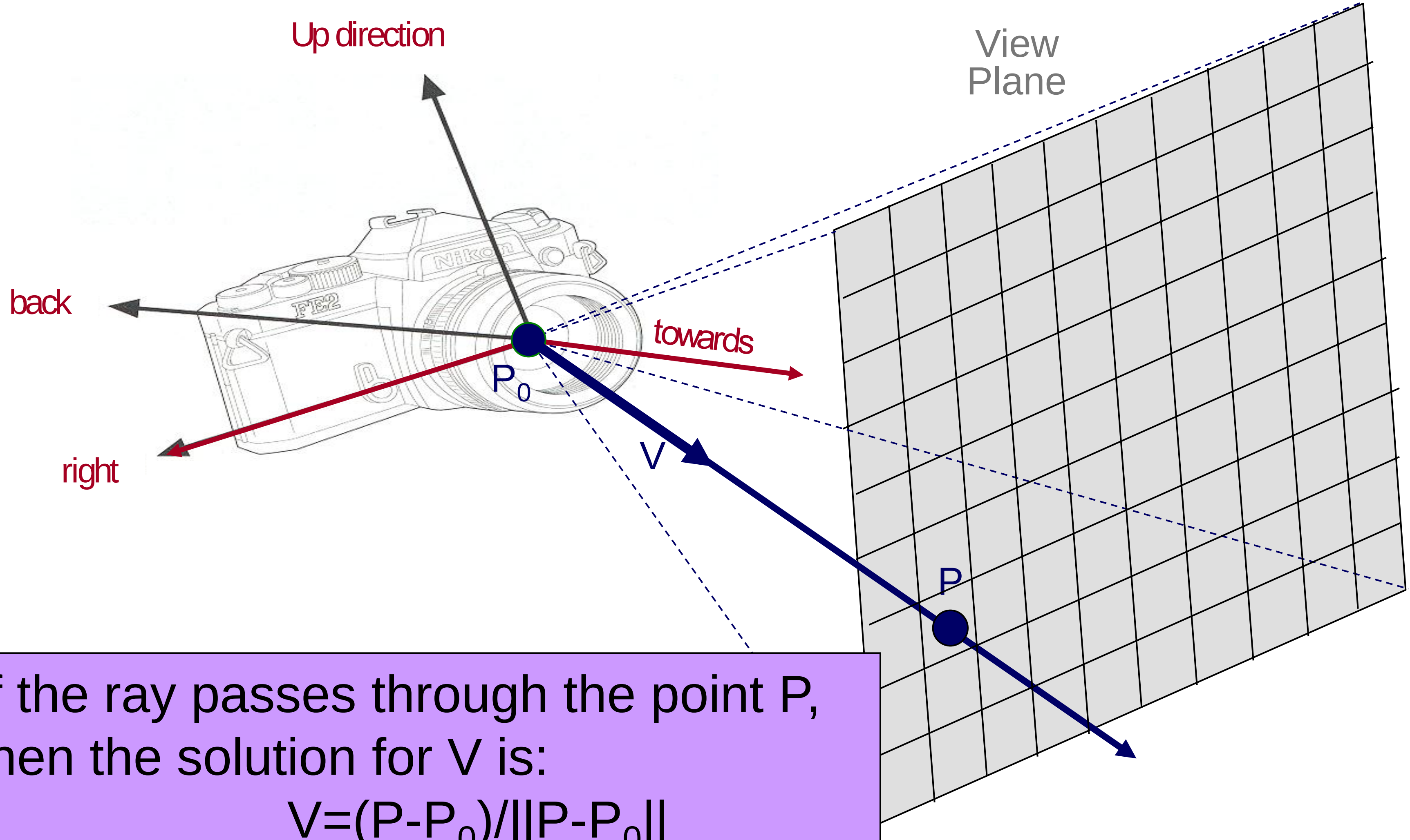
Constructing a Ray Through a Pixel



The ray has to originate at P_0, the position of the camera. So the equation for the ray is of the form:

$$Ray=P_0+tV$$

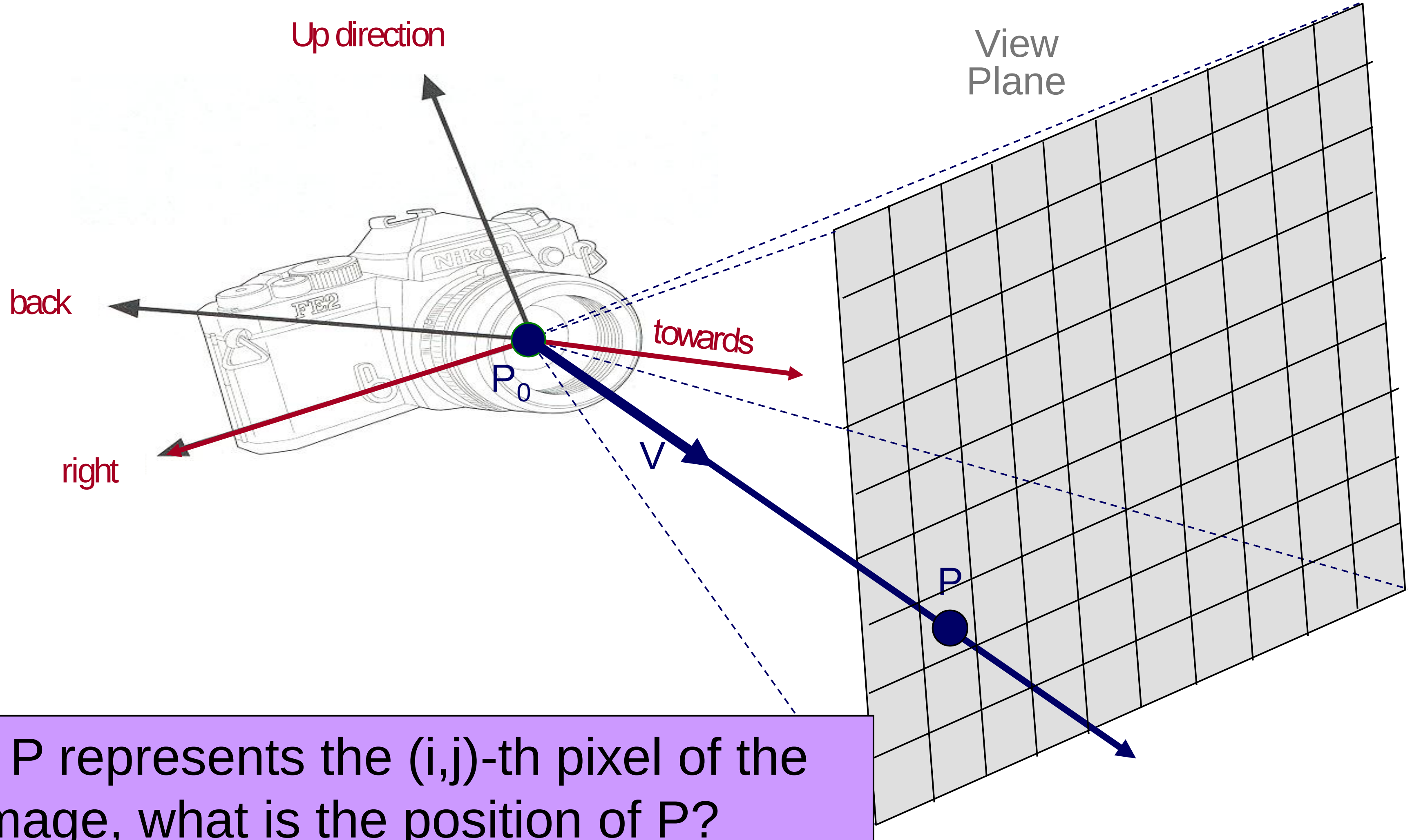
Constructing a Ray Through a Pixel



If the ray passes through the point P, then the solution for V is:

$$V=(P-P_0)/||P-P_0||$$

Constructing a Ray Through a Pixel

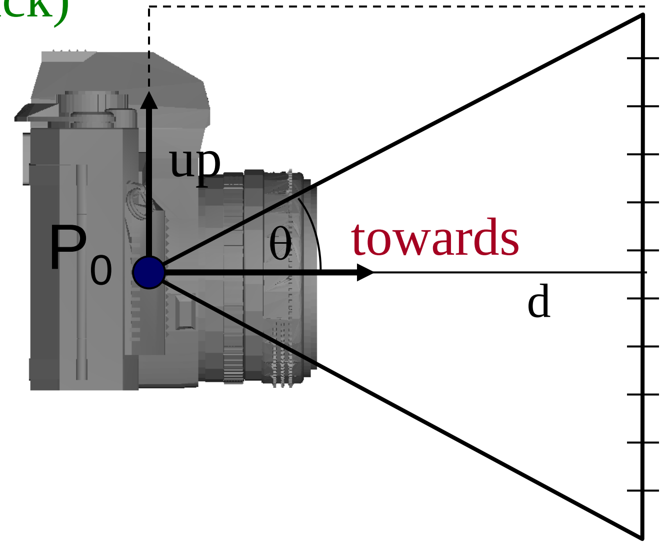


If P represents the (i,j)-th pixel of the image, what is the position of P?

Constructing Ray Through a Pixel

- 2D Example: Side view of camera at P_0
  - What is the position of the *i*-th pixel, P[i]?

θ = frustum half-angle (given), or field of view
d = distance to view plane (arbitrary = you pick)

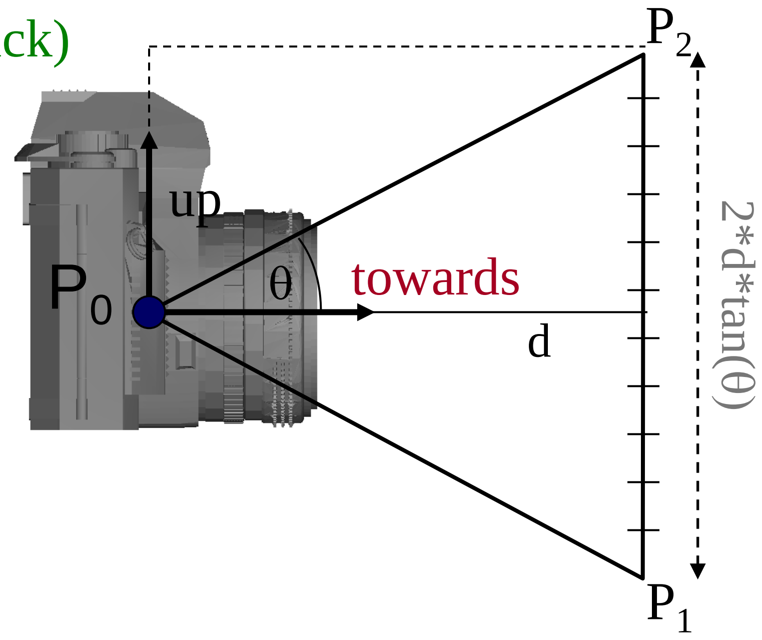


Constructing Ray Through a Pixel

- 2D Example: Side view of camera at P_0
  - What is the position of the *i*-th pixel, P[i]?

θ = frustum half-angle (given), or field of view
d = distance to view plane (arbitrary = you pick)

$P_1 = P_0 + d*\text{towards} - d*\tan(\theta)*\text{up}$
$P_2 = P_0 + d*\text{towards} + d*\tan(\theta)*\text{up}$



Constructing Ray Through a Pixel

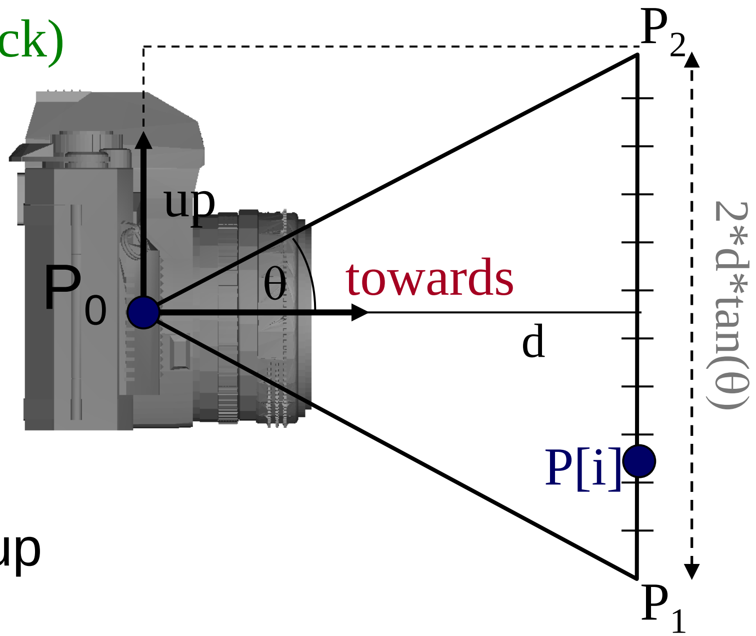
- 2D Example: Side view of camera at P_0
  - What is the position of the *i*-th pixel, P[i]?

θ = frustum half-angle (given), or field of view
d = distance to view plane (arbitrary = you pick)

$P_1 = P_0 + d*towards - d*\tan(\theta)*up$

$P_2 = P_0 + d*towards + d*\tan(\theta)*up$

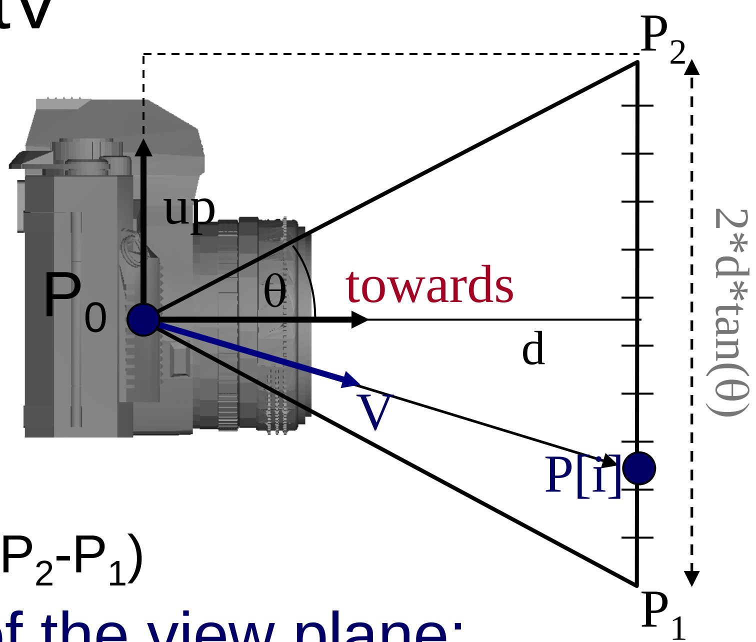
$P[i] = P_1 + ((i+0.5)/height)*(P_2-P_1)$
$\quad = P_1 + ((i+0.5)/height)*2*d*\tan(\theta)*up$



Constructing Ray Through a Pixel

- 2D Example:
  - The ray passing through the *i*-th pixel is defined by:

$$Ray=P_0+tV$$



- Where:
  - P_0 is the camera position
  - V is the direction to the *i*-th pixel: $V=(P[i]-P_0)/||P[i]-P_0||$
  - P[i] is the *i*-th pixel location:

$$P[i] = P_1 + ((i+0.5)/height)*(P_2-P_1)$$

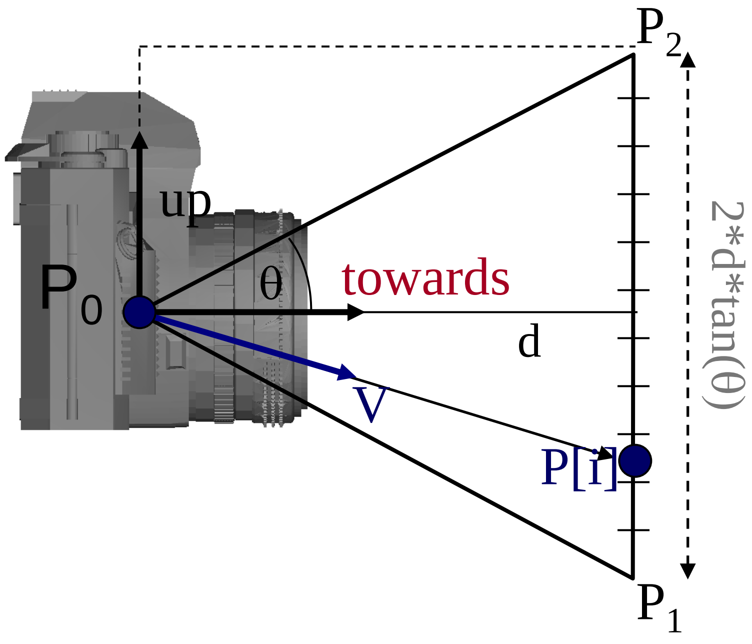
  - P_1 and P_2 are the endpoints of the view plane:

$$P_1 = P_0 + d*towards - d*tan(\theta)*up$$

$$P_2 = P_0 + d*towards + d*tan(\theta)*up$$

Constructing Ray Through a Pixel

- Figuring out how to do this in 3D is assignment 2



Ray Casting

- Simple implementation:

```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

Ray Casting

- Simple implementation:

```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
	Image image = new Image(width, height);
	for (int i = 0; i < width; i++) {
		for (int j = 0; j < height; j++) {
			Ray ray = ConstructRayThroughPixel(camera, i, j);
			Intersection hit = FindIntersection(ray, scene);
			image[i][j] = GetColor(hit);
		}
	}
	return image;
}
```

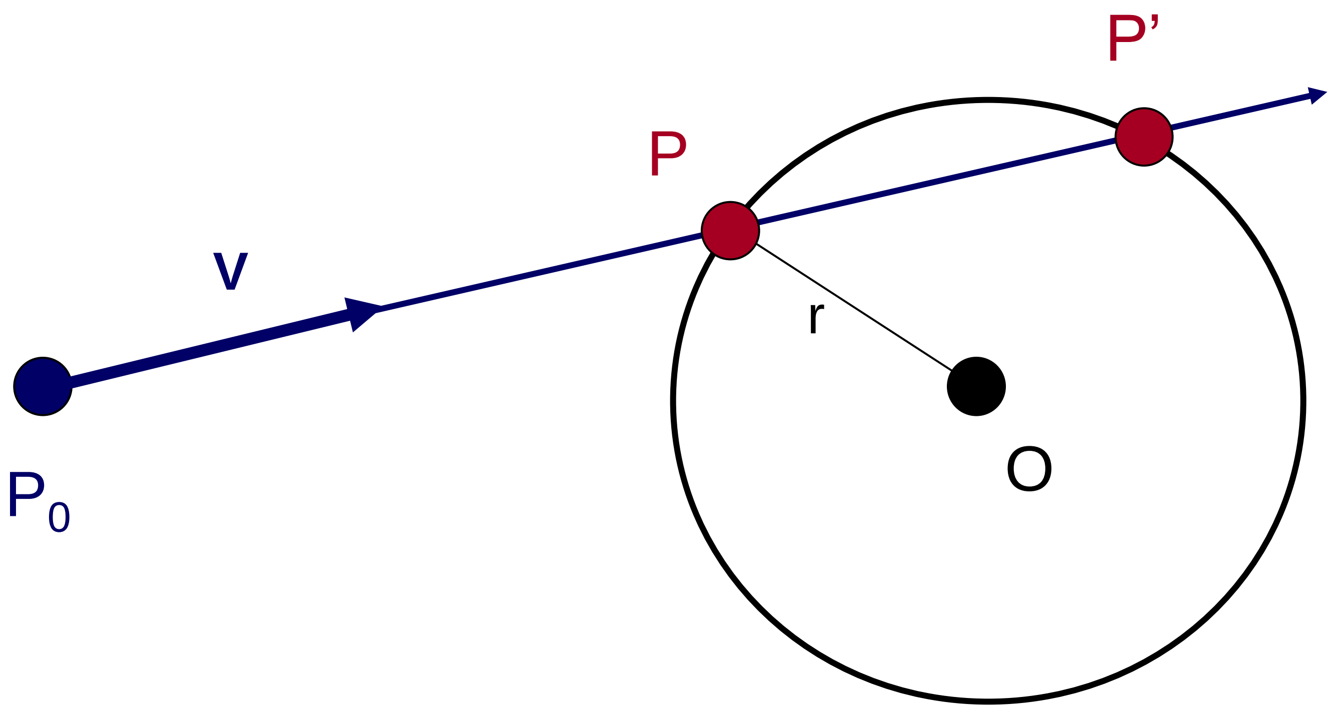
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform (Voxel) grids
    - » Octrees
    - » BSP trees

Ray-Sphere Intersection

Ray: $P = P_0 + tV$

Sphere: $|P - O|^2 - r^2 = 0$

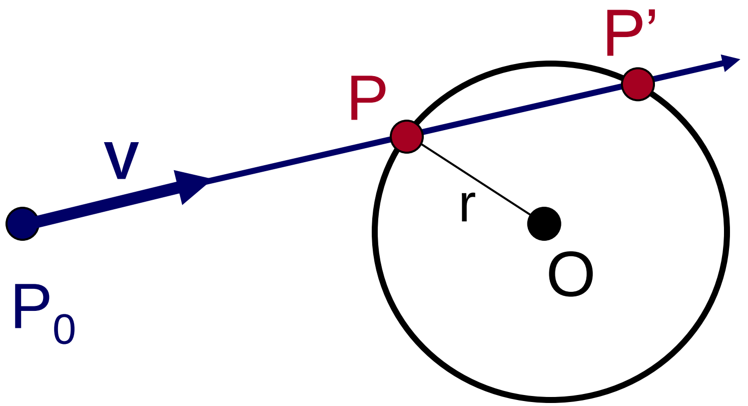


Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Substituting for P, we get:

$$|\mathbf{P_0 + tV} - O|^2 - r^2 = 0$$



Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Substituting for P, we get:

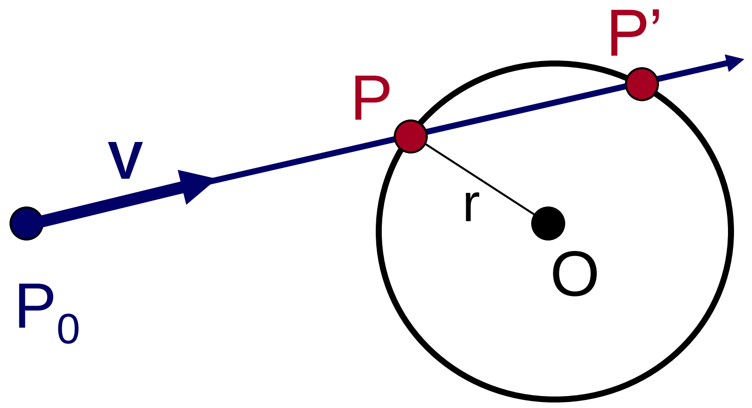
$$|\mathbf{P_0 + tV} - O|^2 - r^2 = 0$$

Solve quadratic equation:

$$at^2 + bt + c = 0$$

where:

$$a = 1$$
$$b = 2\, V \bullet (P_0 - O)$$
$$c = |P_0 - O|^2 - r^2 = 0$$



Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - O|^2 - r^2 = 0$

Substituting for P, we get:

$$|\mathbf{P_0 + tV} - O|^2 - r^2 = 0$$

Solve quadratic equation:

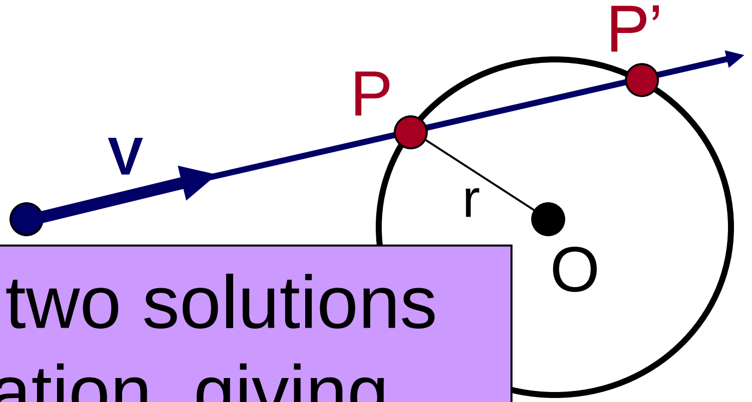
$$at^2 + bt + c = 0$$

where:

a

b

c

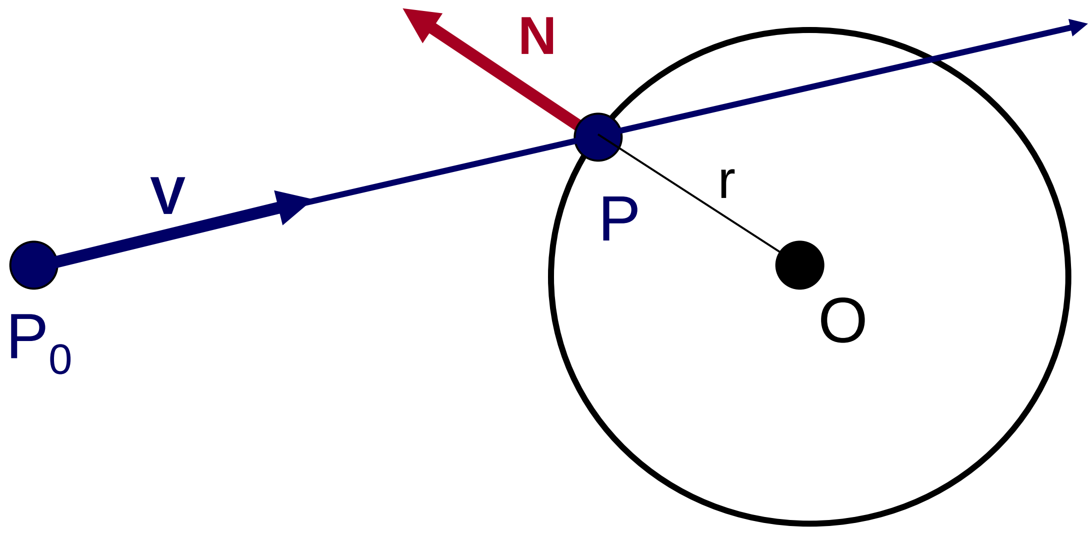


Generally, there are two solutions to the quadratic equation, giving rise to points P and P'.
You want to return the first hit.

Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

$$N = (P - O) / ||P - O||$$

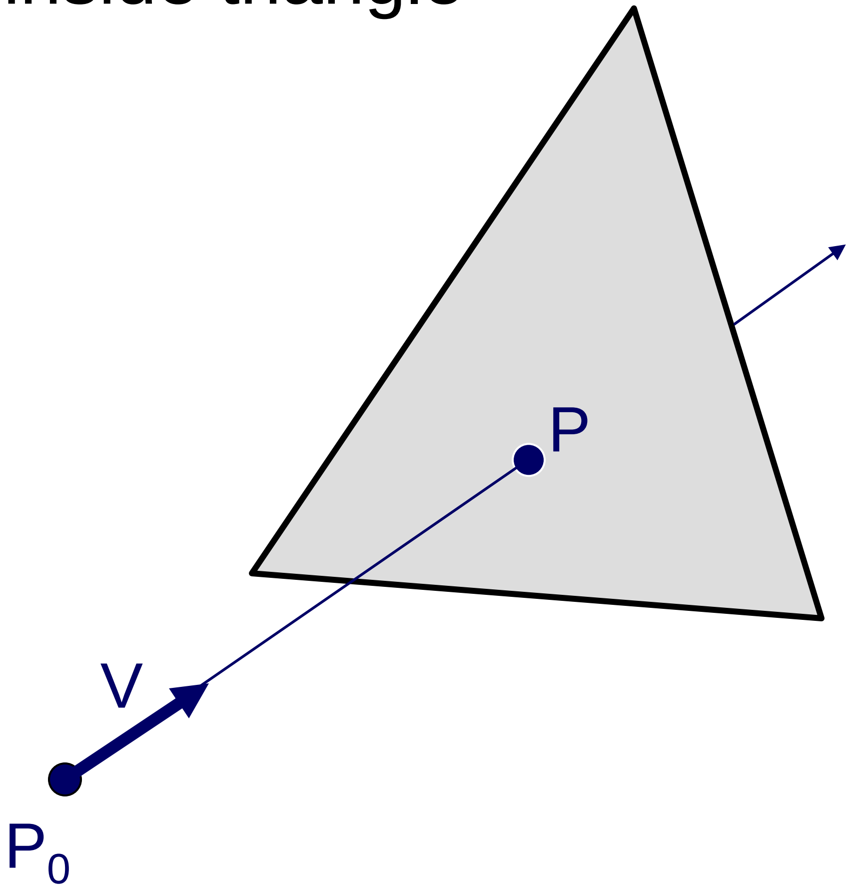


Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - » Triangle
- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - » Uniform grids
    - » Octrees
    - » BSP trees

Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle



Ray-Plane Intersection

Ray: $P = P_0 + tV$
Plane: $P \cdot N - d = 0$

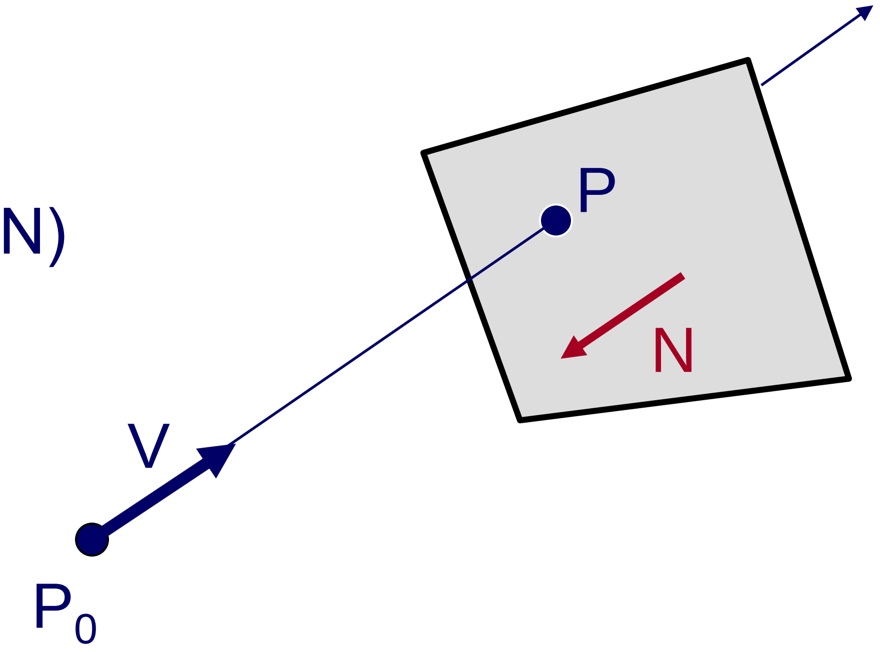
Algebraic Method

Substituting for P, we get:

$$(\mathbf{P_0 + tV}) \cdot N - d = 0$$

Solution:

$$t = -(P_0 \cdot N - d) / (V \cdot N)$$



Ray-Triangle Intersection I

- Check if point is inside triangle algebraically

For each side of triangle

$V_1 = T_1 - P_0$

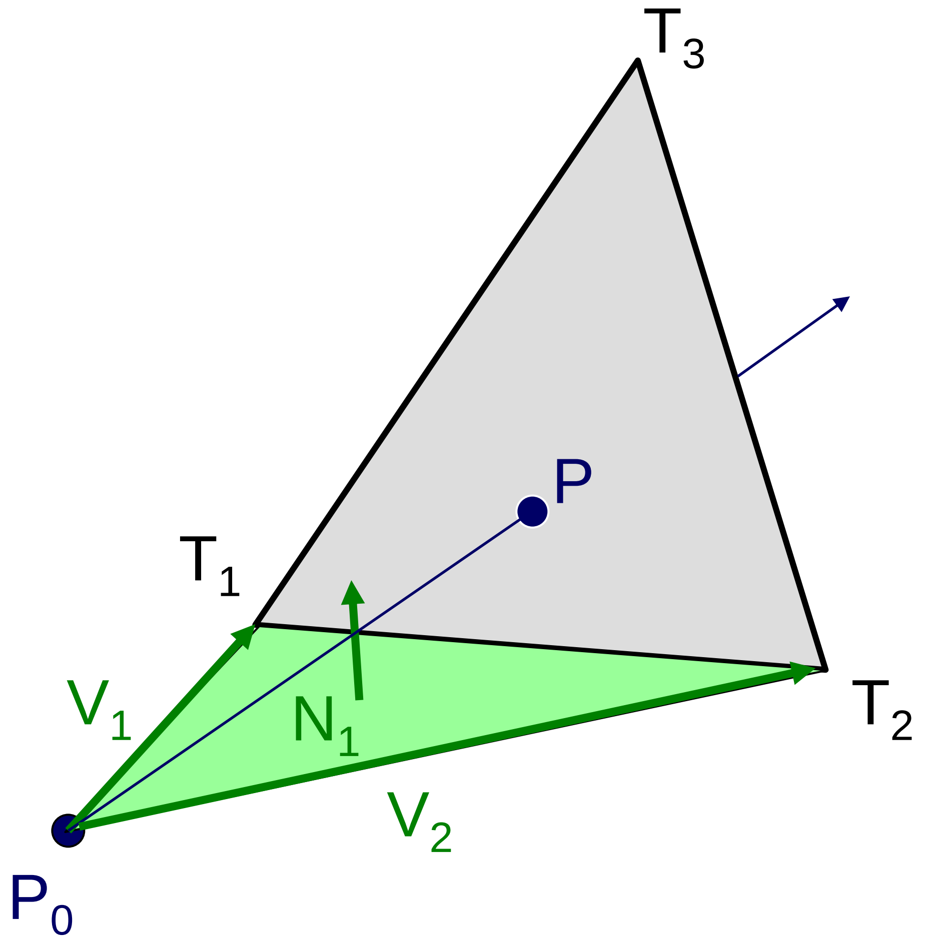
$V_2 = T_2 - P_0$

$N_1 = V_2 \times V_1$

if $((P - P_0) \bullet N_1 < 0)$

return FALSE;

end



Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

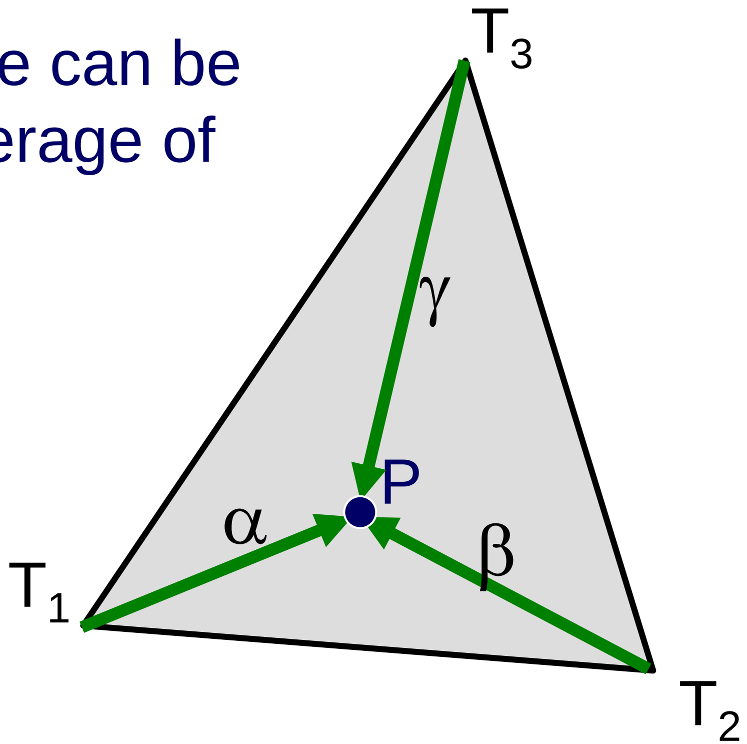
Every point P inside the triangle can be expressed as the weighted average of the corners:

$$P = \alpha T_1 + \beta T_2 + \gamma T_3$$

where:

$$0 \leq \alpha, \beta, \gamma \leq 1$$

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



Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

Solve for α, β, γ such that:

$$P = \alpha T_1 + \beta T_2 + \gamma T_3$$

And

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

Check is point inside triangle.

$$0 \leq \alpha, \beta, \gamma \leq 1$$