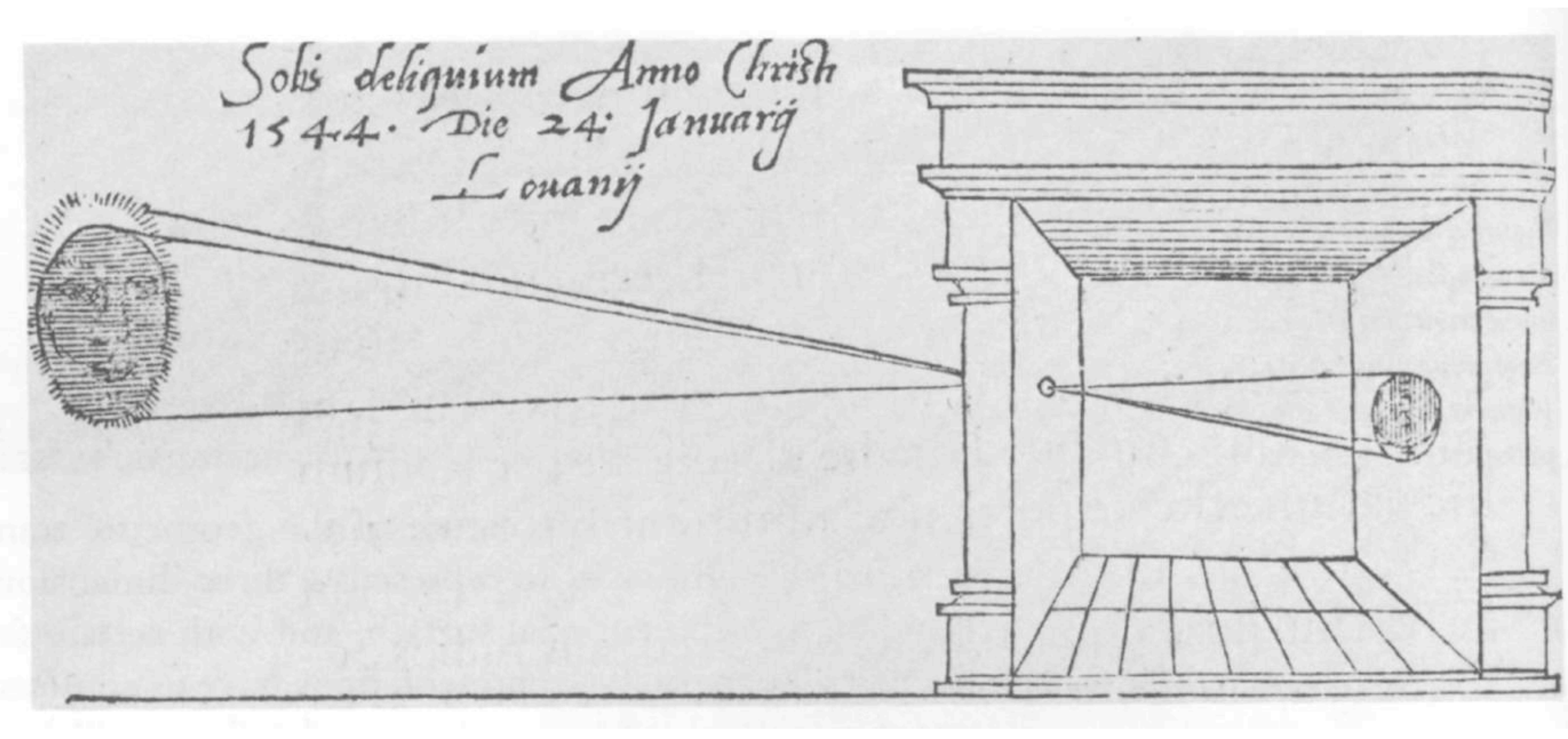
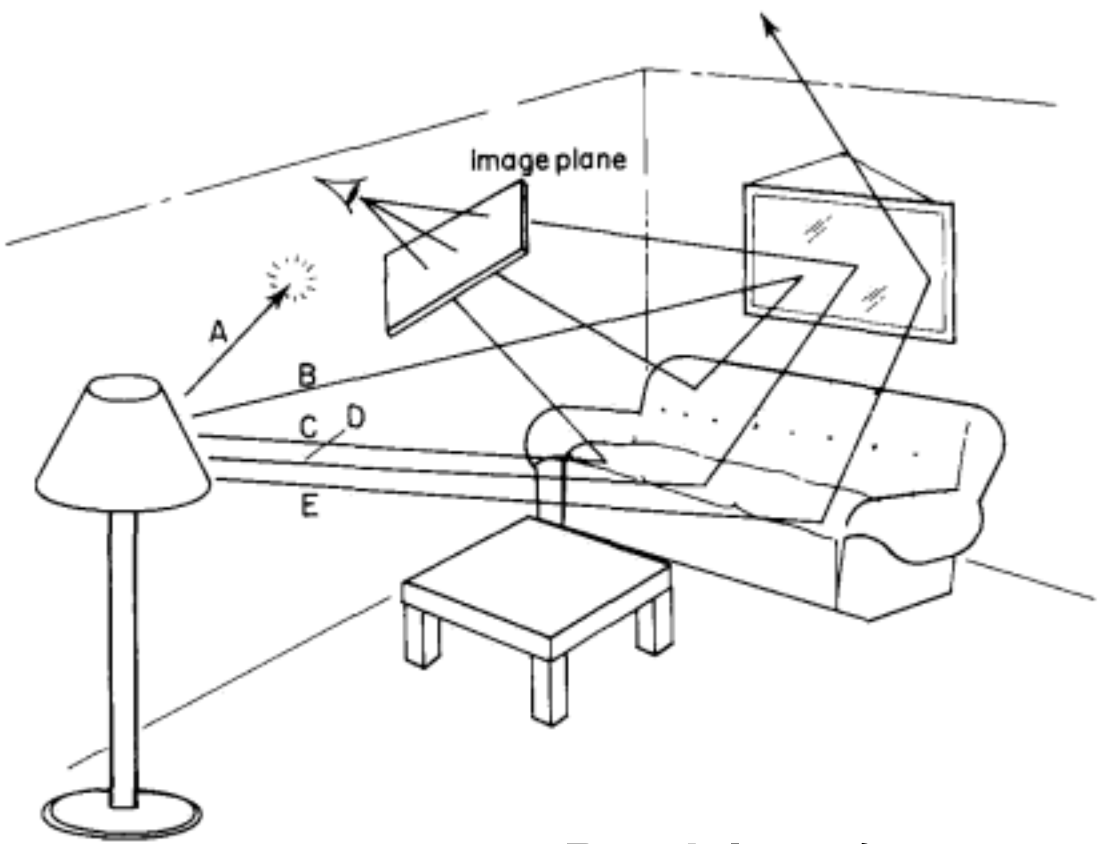
Ray Tracing Basics

CSE 681 Autumn 11
Han-Wei Shen



Forward Ray Tracing

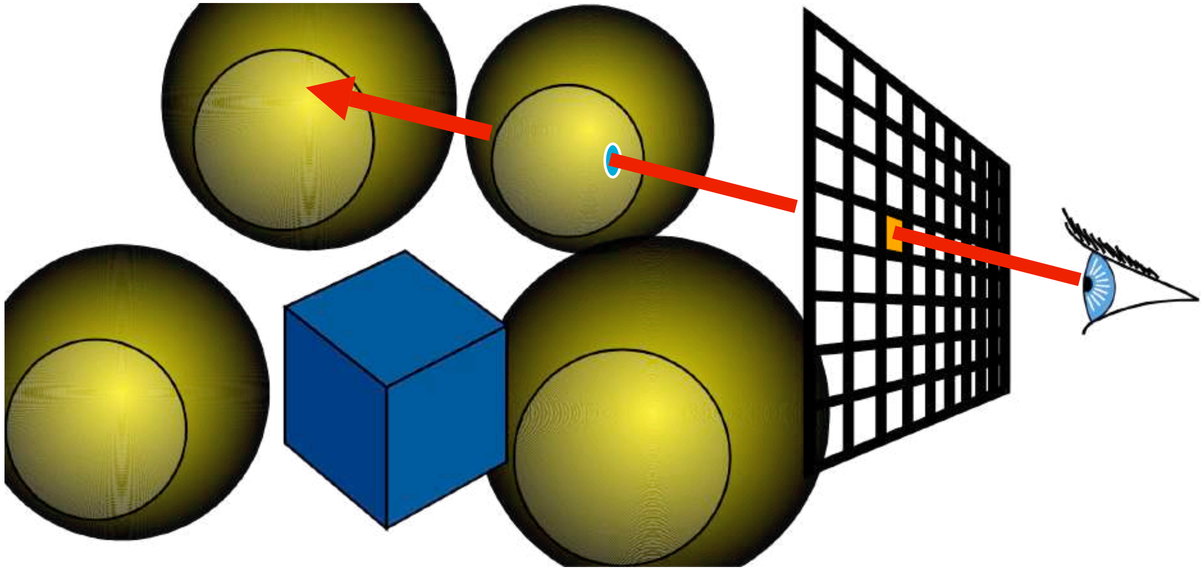
- We shoot a large number of photons



Problem?

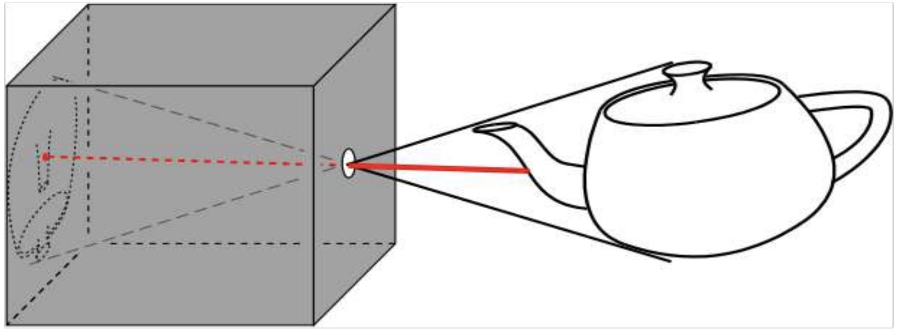
Backward Tracing

```
For every pixel
  Construct a ray from the eye
  For every object in the scene
      Find intersection with the ray
      Keep if closest
```

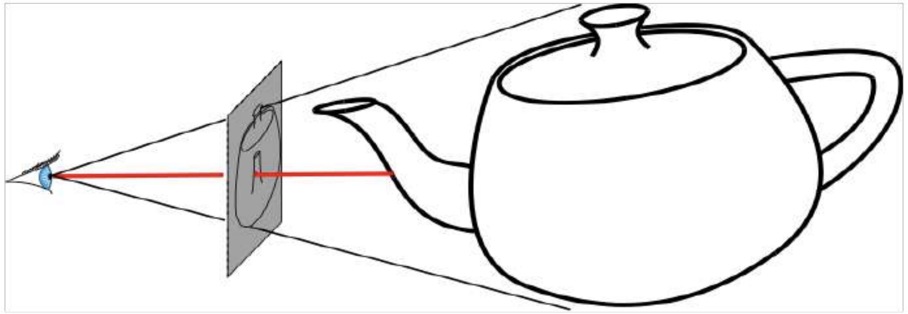


The Viewing Model

- Based on a simple Pinhole Camera model
  - Simplest lens model
  - Inverted image
  - Similar triangles
  - Perfect image if hole infinitely small
  - Pure geometric optics
  - No blurry



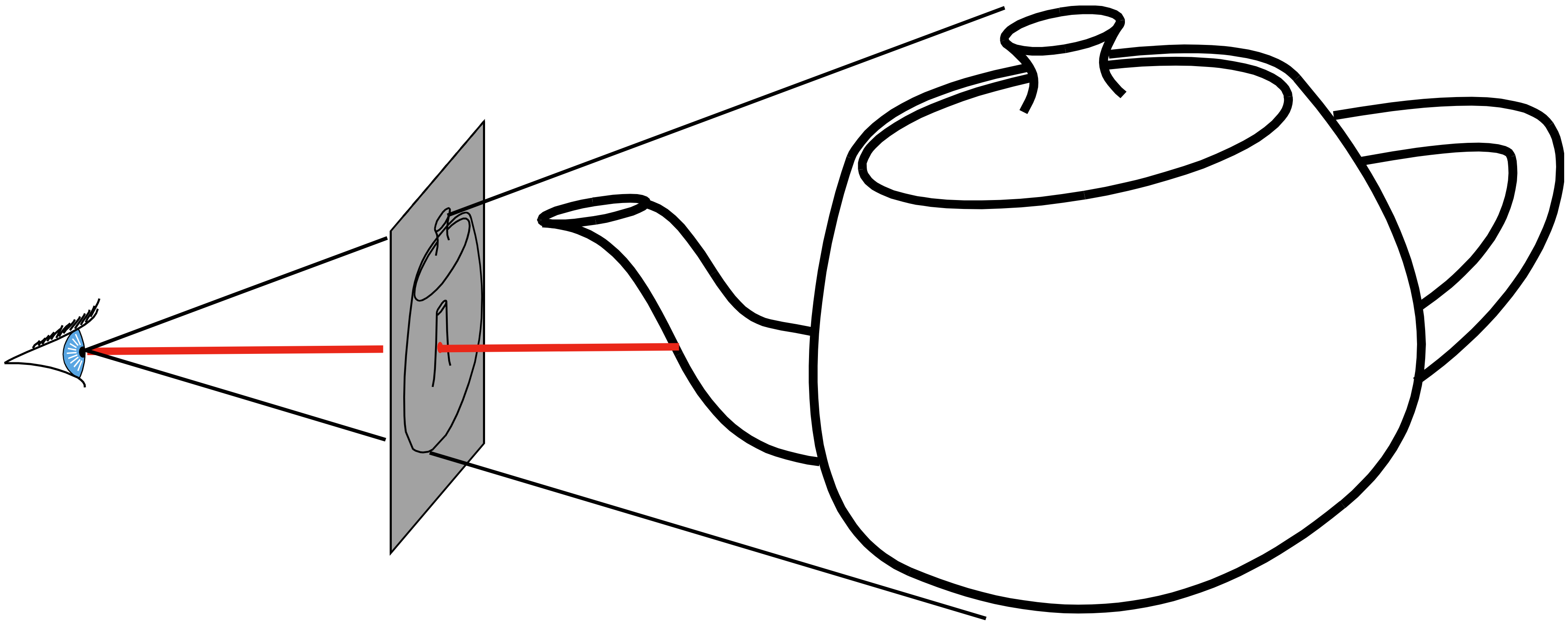
pin-hole camera



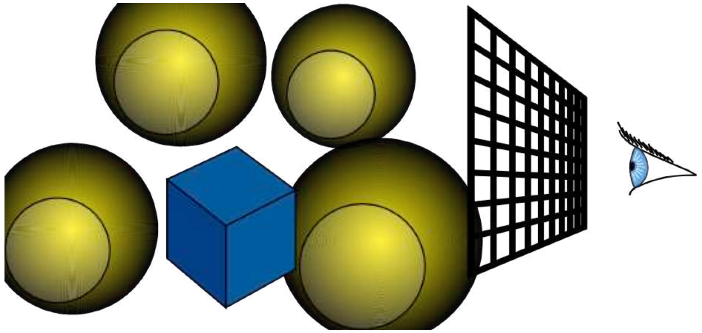
simplified pin-hole camera

Simplified Pinhole Camera

- Eye = pinhole, Image plane = box face (re-arrange)
- Eye-image pyramid (frustum)
- Note that the distance/size of image are arbitrary



Basic Ray Tracing Algorithm



```
for every pixel  {
     cast a ray from the eye
     for every object in the scene
          find intersections with the ray
          keep it if closest
      }
      compute color at the intersection point
   }
```

Construct a Ray

3D parametric line

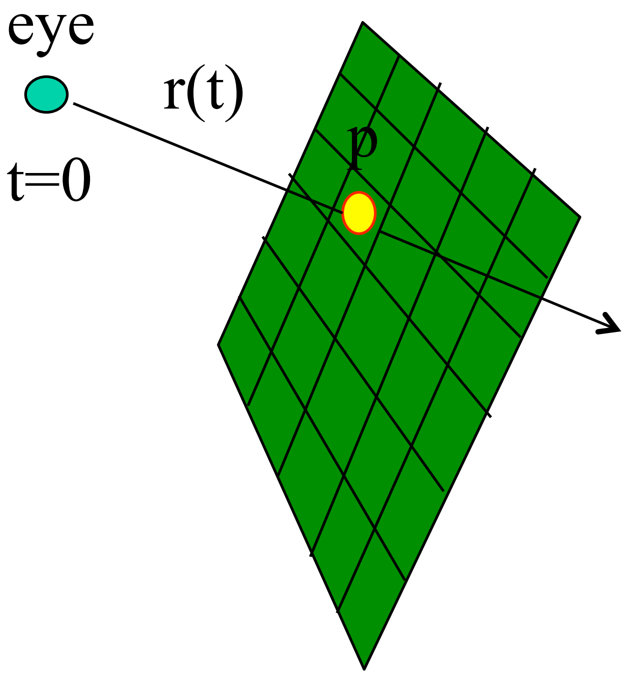
$$p(t) = eye + t\,(s - eye)$$

r(t): ray equation

eye: eye (camera) position

s: pixel position

t: ray parameter



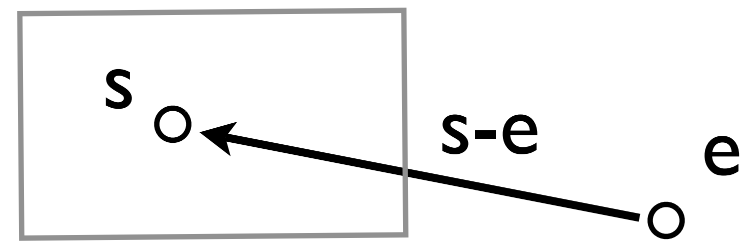
Question: How to calculate the pixel position P?

Constructing a Ray

- 3D parametric line

  p(t) = **e** + t (**s**-**e**)

  *(boldface means vector)

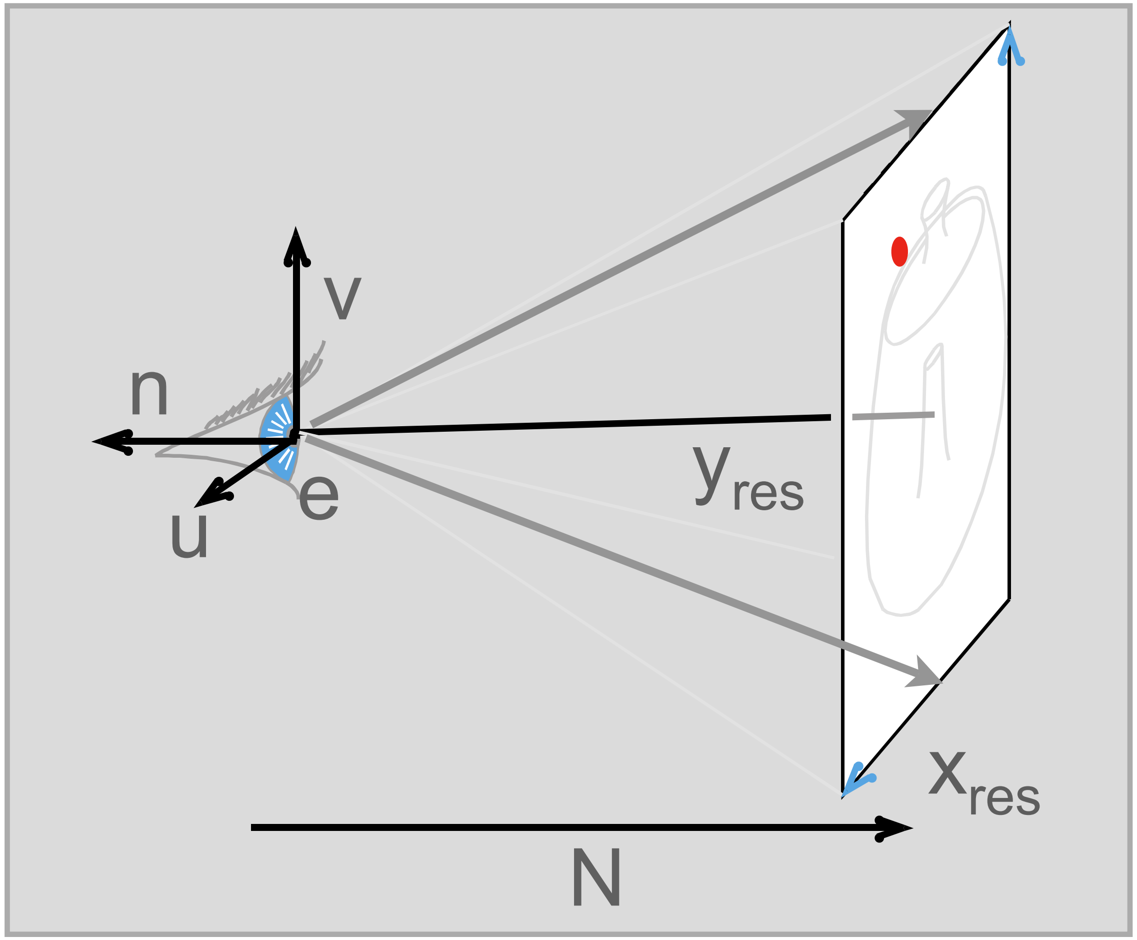


- So we need to know **e** and **s**

- What are given (specified by the user or scene file)?
  - ✓ camera position
  - ✓ camera direction or center of interest
  - ✓ camera orientation or view up vector
  - ✓ distance to image plane
  - ✓ field of view + aspect ratio
  - ✓ pixel resolution

Given Camera Information

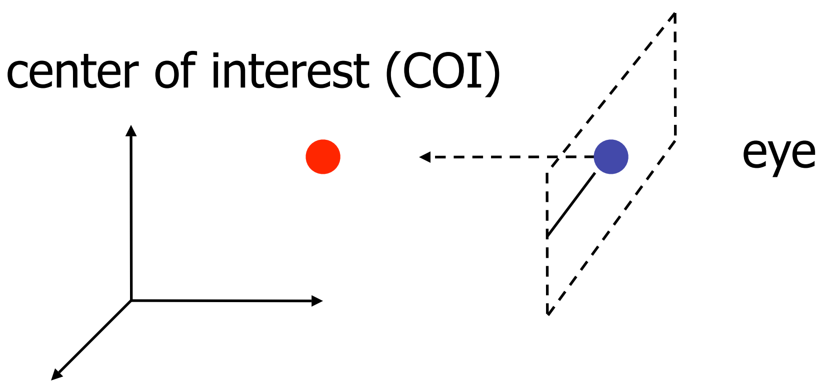
- Camera
  - Eye
  - Look at
  - Orientation (up vector)
- Image plane
  - Distance to plane, N
  - Field of view in Y
  - Aspect ration (X/Y)
- Screen
  - Pixel resolution



Construct Eye Coordinate System

- We can calculate the pixel positions much more easily if we construct an eye coordinate system (eye space) first

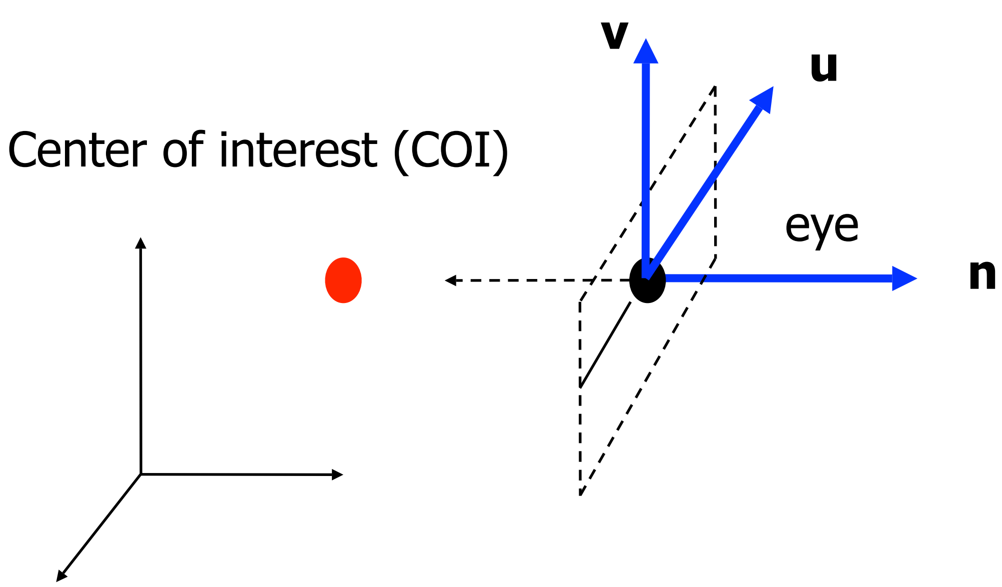
  - Known: eye position, center of interest, view-up vector
  - To find out: new origin and three basis vectors



Assumption: the direction of view is orthogonal to the view plane (the plane that objects will be projected onto)

Eye Coordinate System

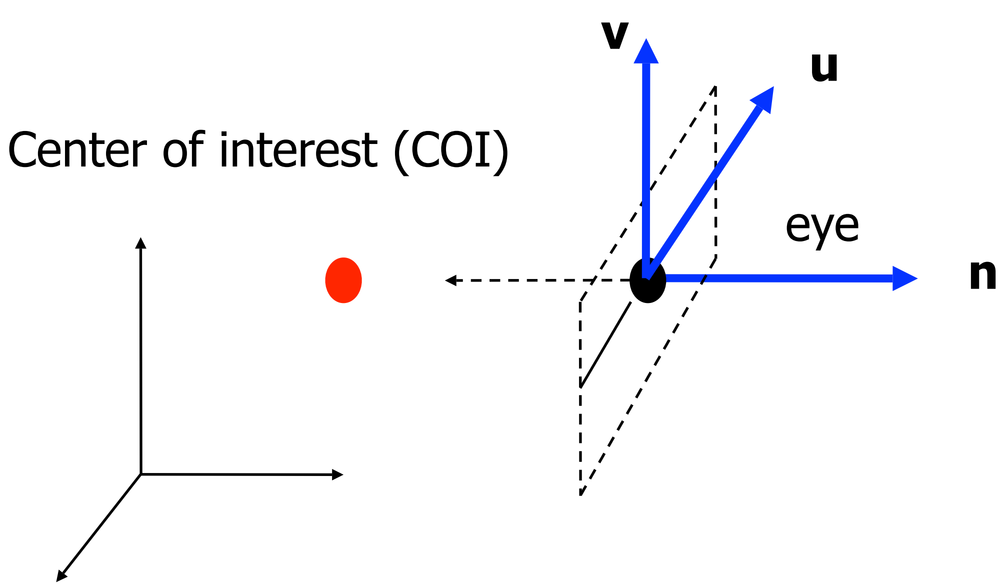
- Origin: eye position
- Three basis vectors: one is the normal vector (**n**) of the viewing plane, the other two are the ones (**u** and **v**) that span the viewing plane



(u,v,n should be orthogonal to each other)

Eye Coordinate System

- Origin: eye position
- Three basis vectors: one is the normal vector (**n**) of the viewing plane, the other two are the ones (**u** and **v**) that span the viewing plane



n is pointing away from the world because we use right hand coordinate system

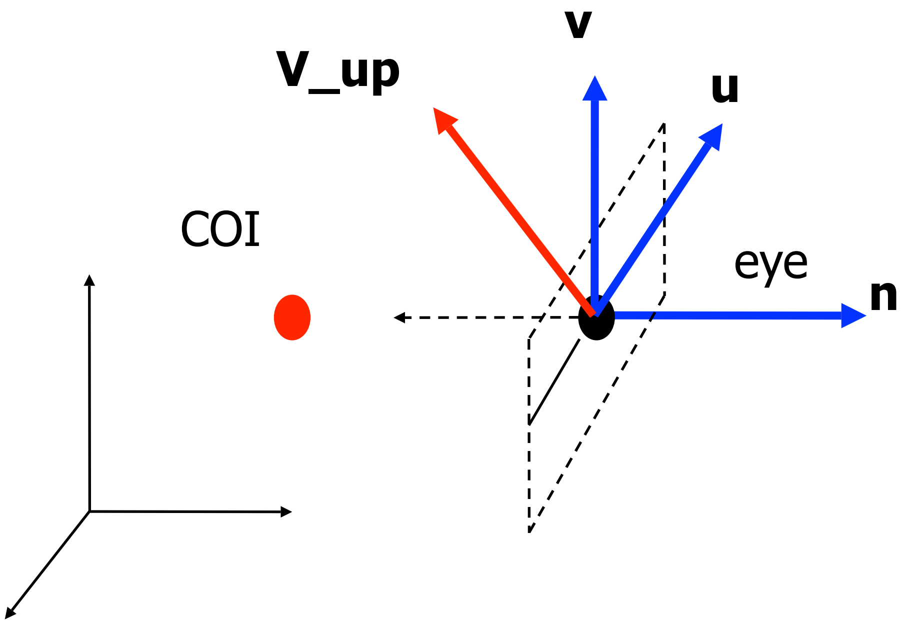
N = eye – COI
n = N / | N |

Remember **u,v,n** should be all unit vectors

(u,v,n should be orthogonal to each other)

Eye Coordinate System

- What about u and v?

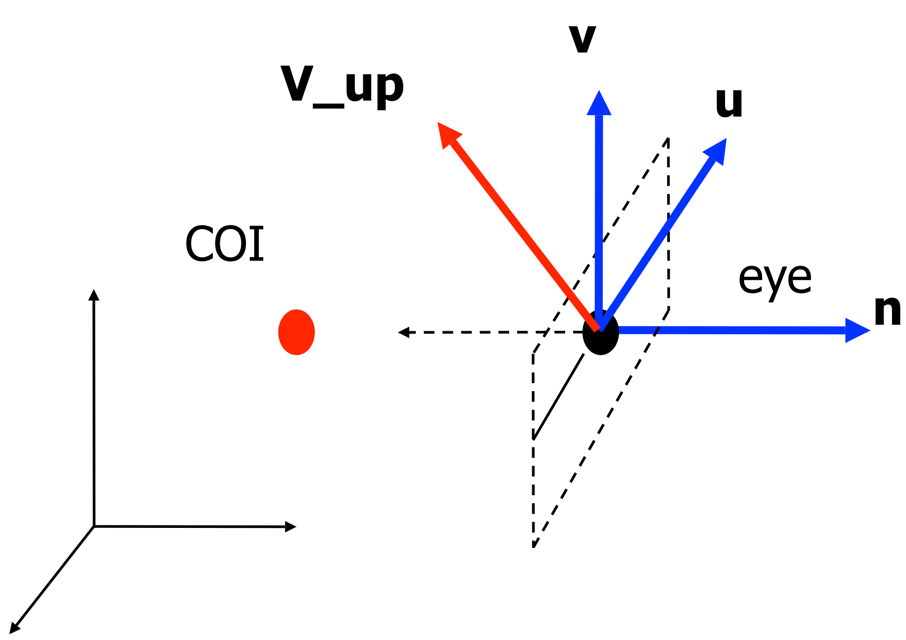


We can get u first -

u is a vector that is perpendicular to the plane spanned by N and view up vector (V_up)

Eye Coordinate System

- What about u and v?



We can get u first -

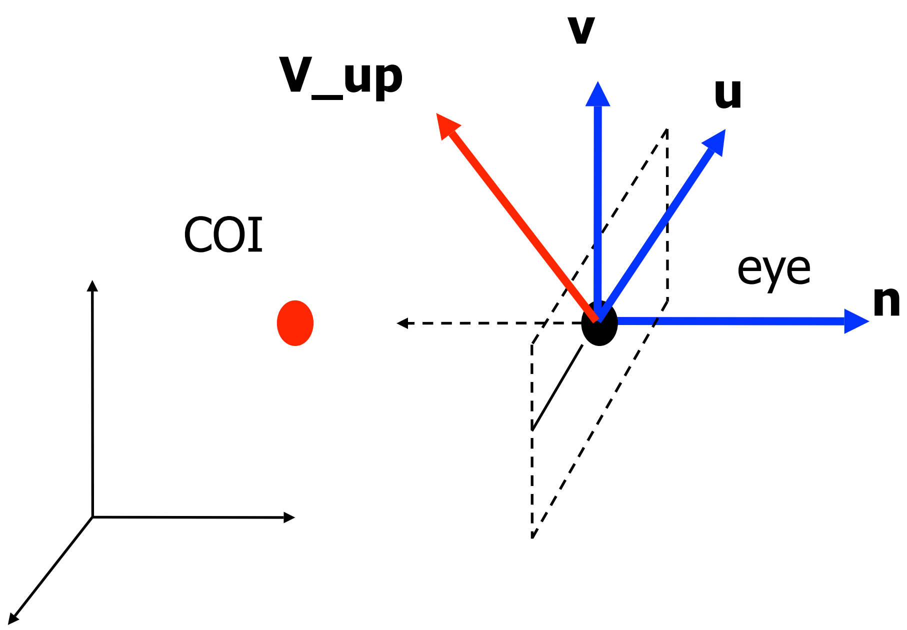
u is a vector that is perpendicular to the plane spanned by N and view up vector (V_up)

$$U = V\_up \times \mathbf{n}$$

$$\mathbf{u} = U \,/\, |U|$$

Eye Coordinate System

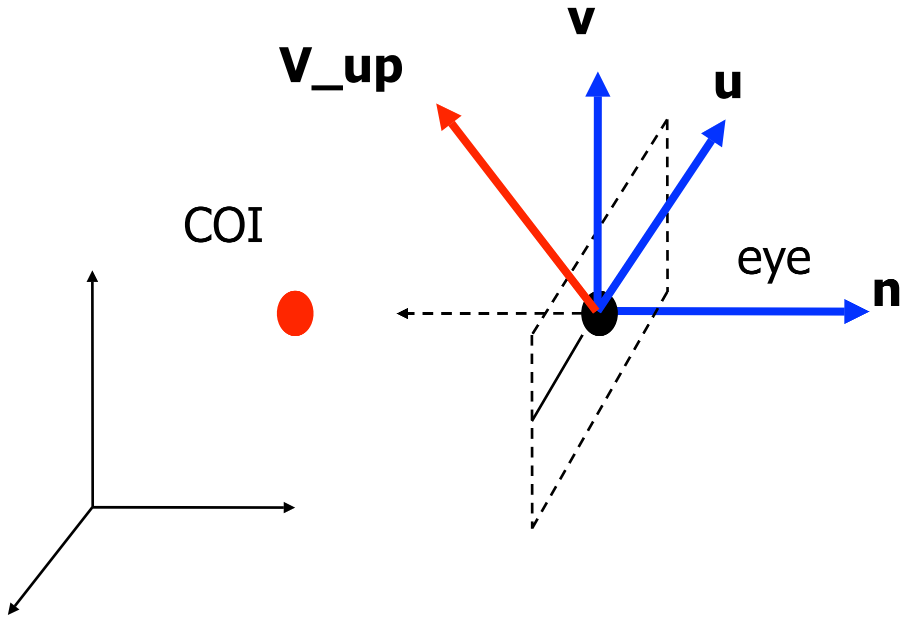
- What about v?



Knowing n and u, getting v is easy

Eye Coordinate System

- What about v?



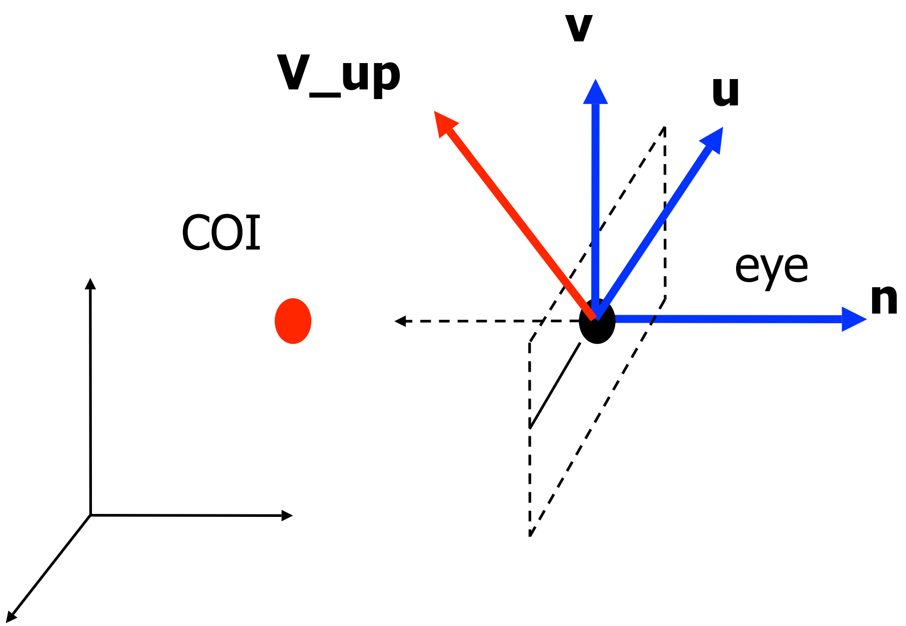
Knowing n and u, getting v is easy

v = n x u

v is already normalized

Eye Coordinate System

- Put it all together



Eye space **origin: (Eye.x , Eye.y, Eye.z)**

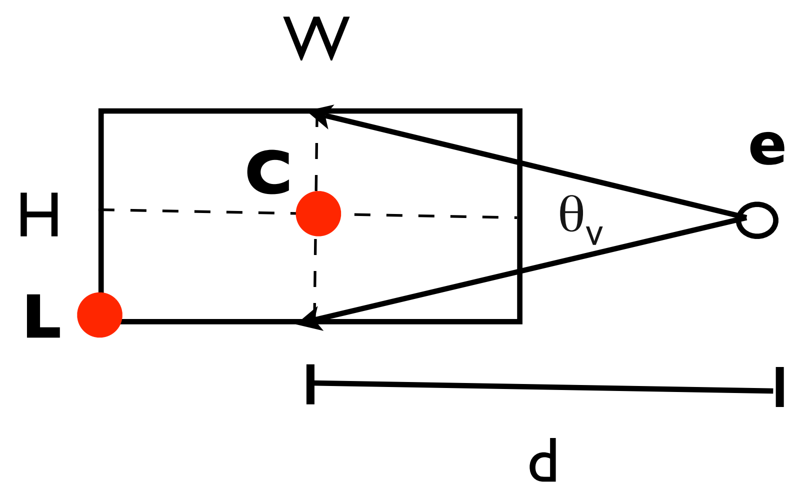
Basis vectors:

n = (eye – COI) / | eye – COI|
u = (V_up x **n**) / | V_up x **n** |
v = **n** x **u**

Next Step?

- Determine the size of the image plane
- This can be derived from
  - ✓ distance from the camera to the center of the image plane
  - ✓ Vertical field of view angle
  - ✓ Aspect ratio of the image plane
    - ★ Aspect ratio being Width/Height

Image Plane Setup



- $\mathrm{Tan}(\theta_v/2) = H / 2d$
- W = H * aspect_ratio
- C's position = **e** - **n** * d
- L's position = **C** - **u** * W/2 - **v** * H/2
- Assuming the image resolution is X (horizontal) by Y (vertical), then each pixel has a width of W/X and a height of H/Y
- Then for a pixel **s** at the image pixel (i,j) , it's location is at

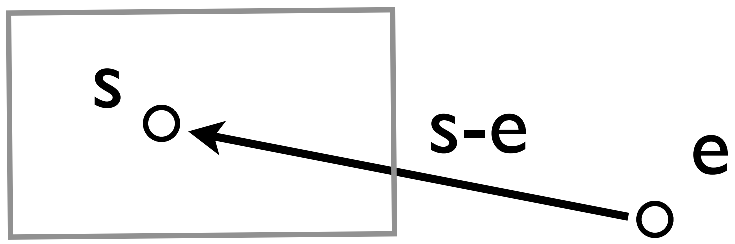
$$\boxed{\mathbf{L} + \mathbf{u} * i * W/X + \mathbf{v} * j * H/Y}$$

Put it all together

- We can represent the ray as a 3D parametric line

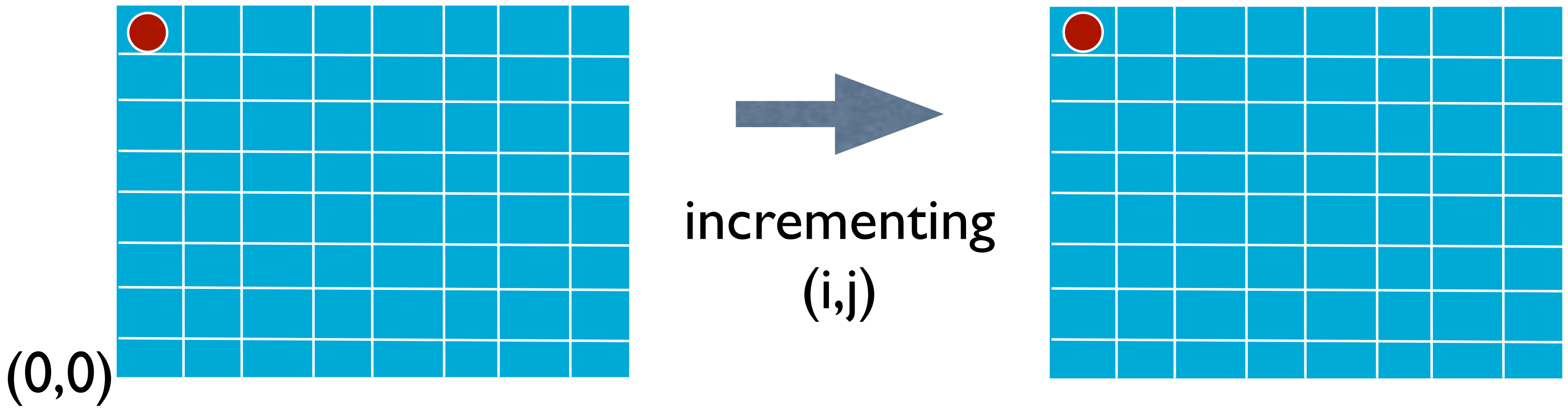
  $\mathbf{p}(t) = \mathbf{e} + t\,(\mathbf{s}-\mathbf{e})$

  (now you know how to get s and e)



- Typically we offset the ray by half

  of the pixel width and height, i.e, cast the ray from the pixel center

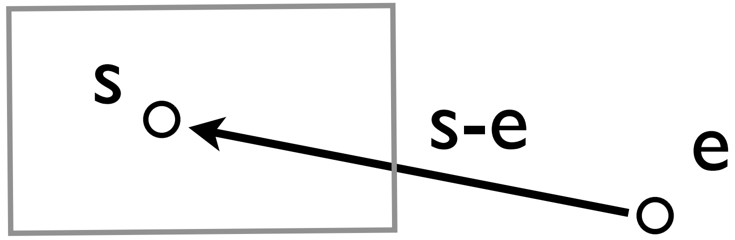


Put it all together

- We can represent the ray as a 3D parametric line

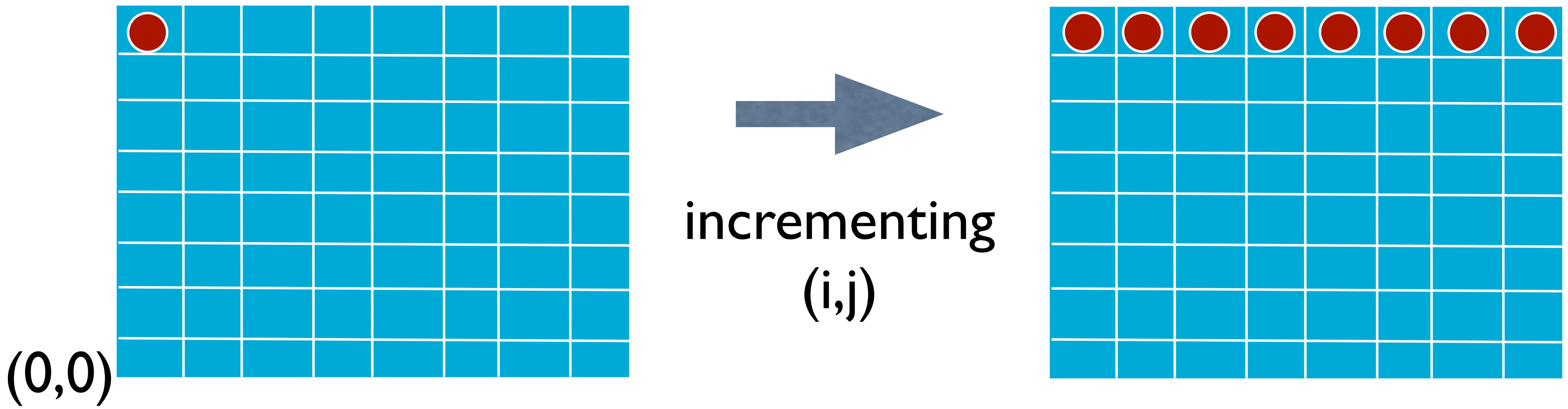
  $\mathbf{p}(t) = \mathbf{e} + t\,(\mathbf{s}-\mathbf{e})$

  (now you know how to get s and e)



- Typically we offset the ray by half

  of the pixel width and height, i.e, cast the ray from the pixel center



Ray-Sphere Intersection

- Problem: Intersect a line with a sphere
  - ✓ A sphere with center $\mathbf{c} = (x_c, y_c, z_c)$ and radius R can be represented as:

    $$(x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 - R^2 = 0$$

  - ✓ For a point **p** on the sphere, we can write the above in vector form:

    $(\mathbf{p}-\mathbf{c}).(\mathbf{p}-\mathbf{c}) - R^2 = 0$ (note '**.**' is a dot product)

  - ✓ We can plug the point on the ray $\mathbf{p}(t) = \mathbf{e} + t\,\mathbf{d}$

    $(\mathbf{e}+t\mathbf{d}-\mathbf{c}).(\mathbf{e}+t\mathbf{d}-\mathbf{c}) - R^2 = 0$ and yield

    $$(\mathbf{d}.\mathbf{d})\, t^2 + 2\mathbf{d}.(\mathbf{e}-\mathbf{c})t + (\mathbf{e}-\mathbf{c}).(\mathbf{e}-\mathbf{c}) - R^2 = 0$$

Ray-Sphere Intersection

- When solving a quadratic equation

  $at^2 + bt + c = 0$

  We have

- Discriminant $d = \sqrt{b^2 - 4ac}$

- and Solution $t_\pm = \dfrac{-b \pm d}{2a}$

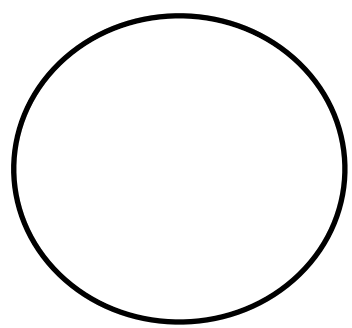
Ray-Sphere Intersection

$$d = \sqrt{b^2 - 4ac}$$

$b^2 - 4ac < 0 \Rightarrow$ **No intersection**

$b^2 - 4ac > 0 \Rightarrow$ **Two solutions (enter and exit)**

$b^2 - 4ac = 0 \Rightarrow$ **One solution (ray grazes sphere)**



- Should we use the larger or smaller t value?

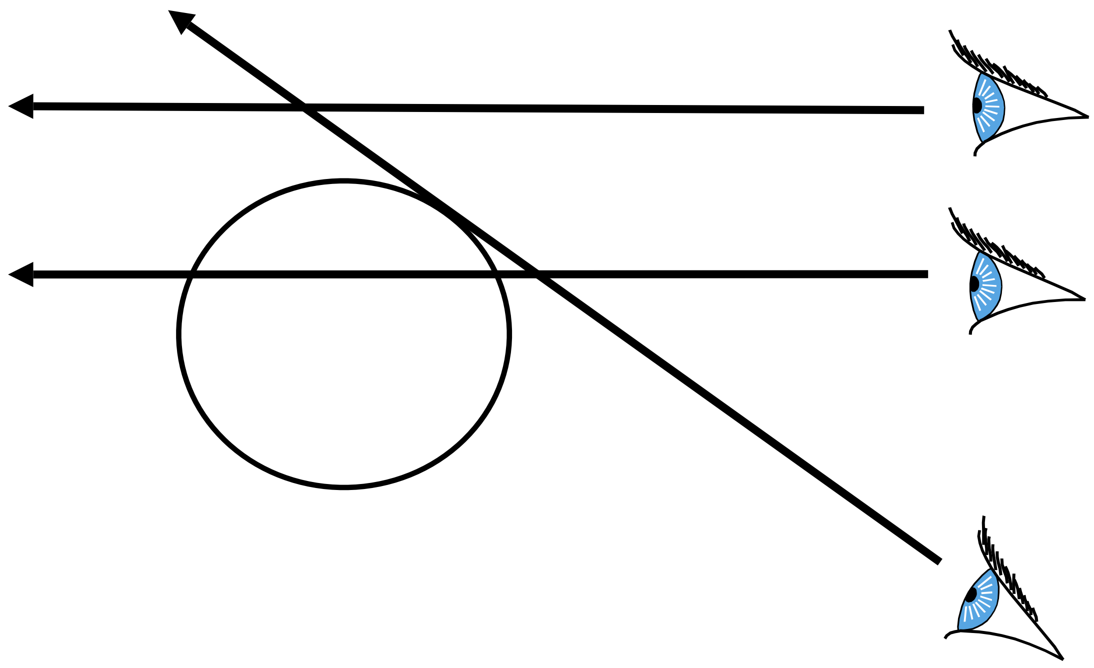
Ray-Sphere Intersection

$$d = \sqrt{b^2 - 4ac}$$

$b^2 - 4ac < 0 \Rightarrow$ **No intersection**

$b^2 - 4ac > 0 \Rightarrow$ **Two solutions (enter and exit)**

$b^2 - 4ac = 0 \Rightarrow$ **One solution (ray grazes sphere)**

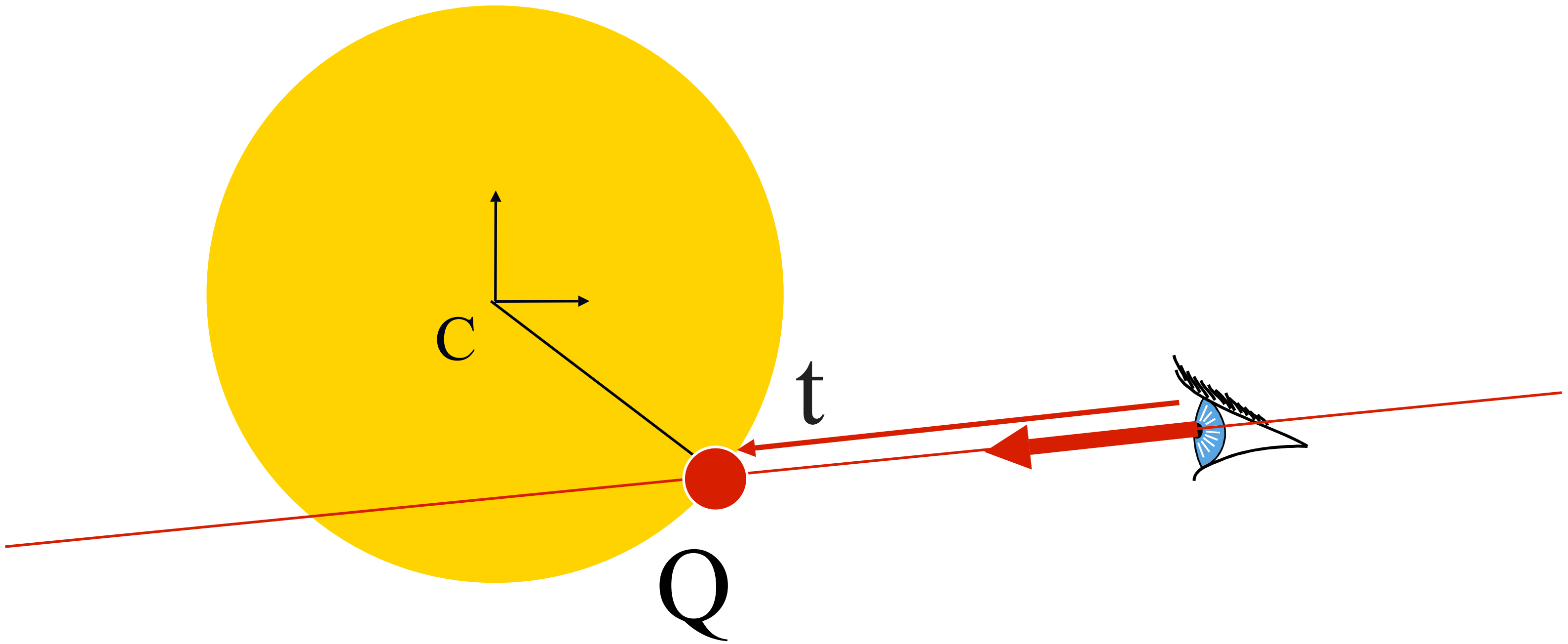


- Should we use the larger or smaller t value?

Calculate Normal

- Needed for computing lighting

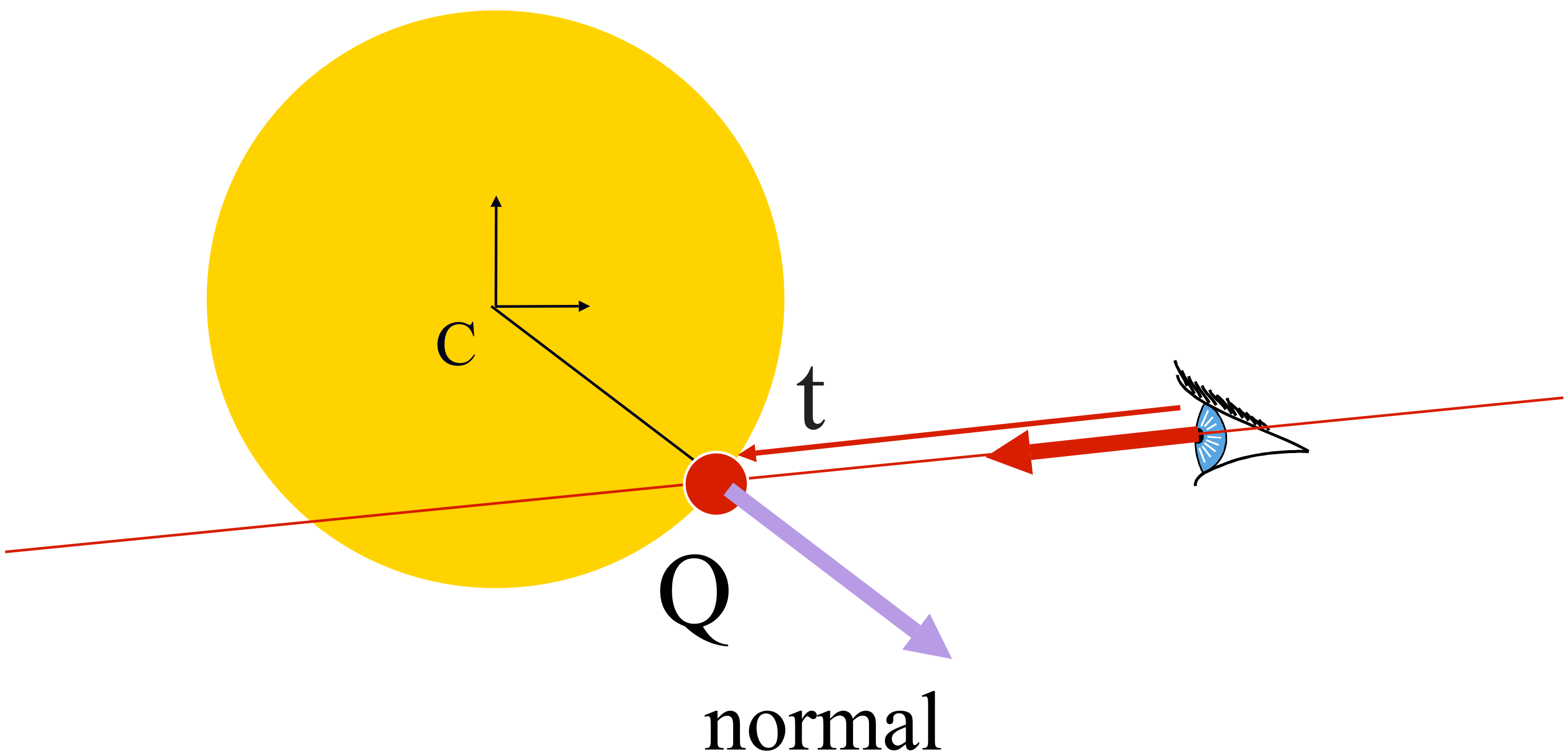
  Q = P(t) – C … and remember Q/||Q||



Calculate Normal

- Needed for computing lighting

  $Q = P(t) - C$ … and remember $Q/||Q||$



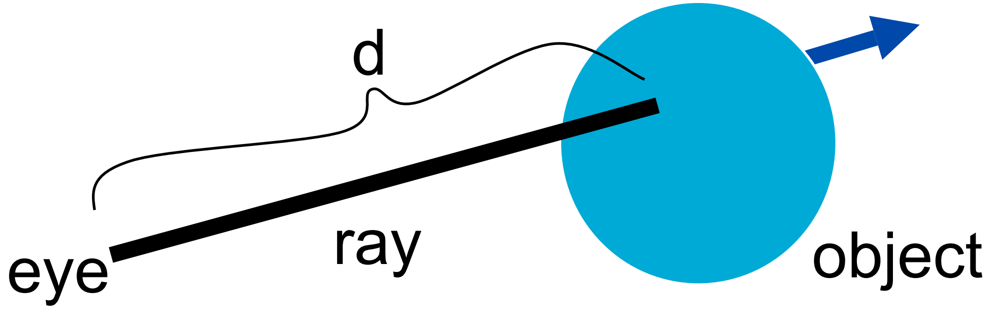
Choose the closet sphere

- Minimum search problem

```
For each pixel {
    form ray from eye through the pixel center
    t_min = ∞
    For each object {
        if (t = intersect(ray, object)) {
            if (t < t_min) {
                closestObject = object
                t_min = t
            }
        }
    }
}
```

Final Pixel Color

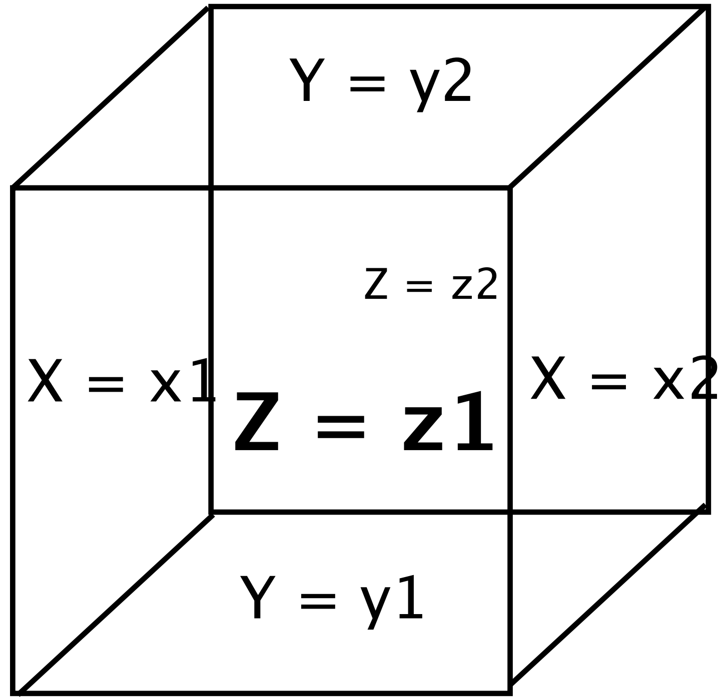
if ($t_{min} == \infty$)
    pixelColor = background color
else
    pixelColor = color of object at d along ray



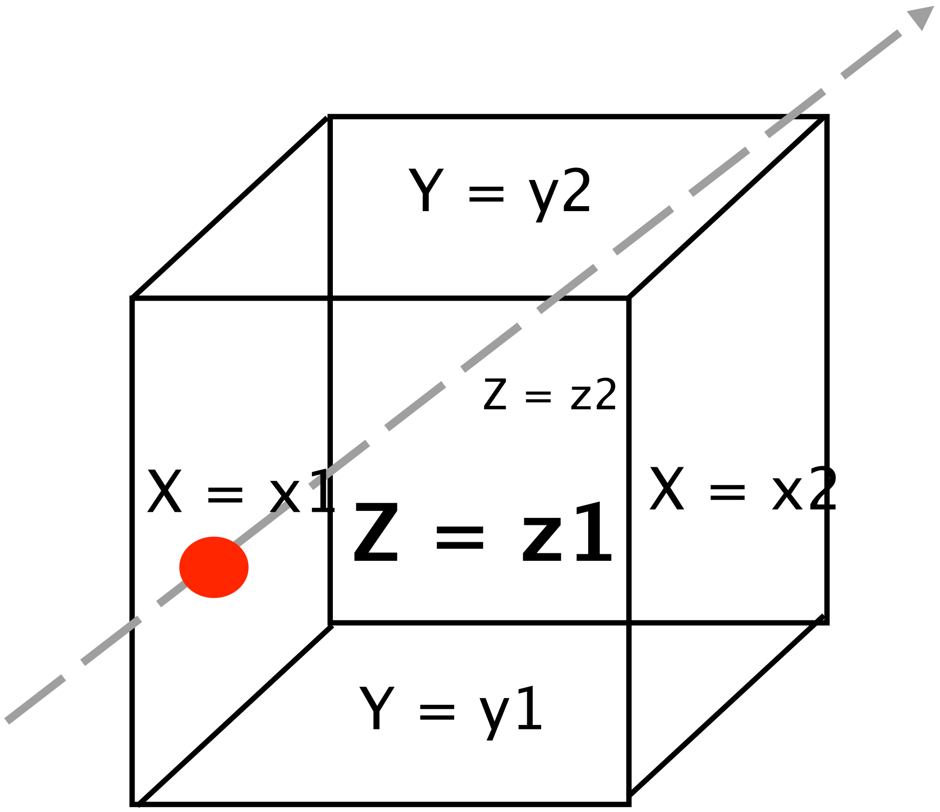
CSE 681

Ray-Object Intersections: Axis-aligned Box

Ray-Box Intersection Test



Ray-Box Intersection Test



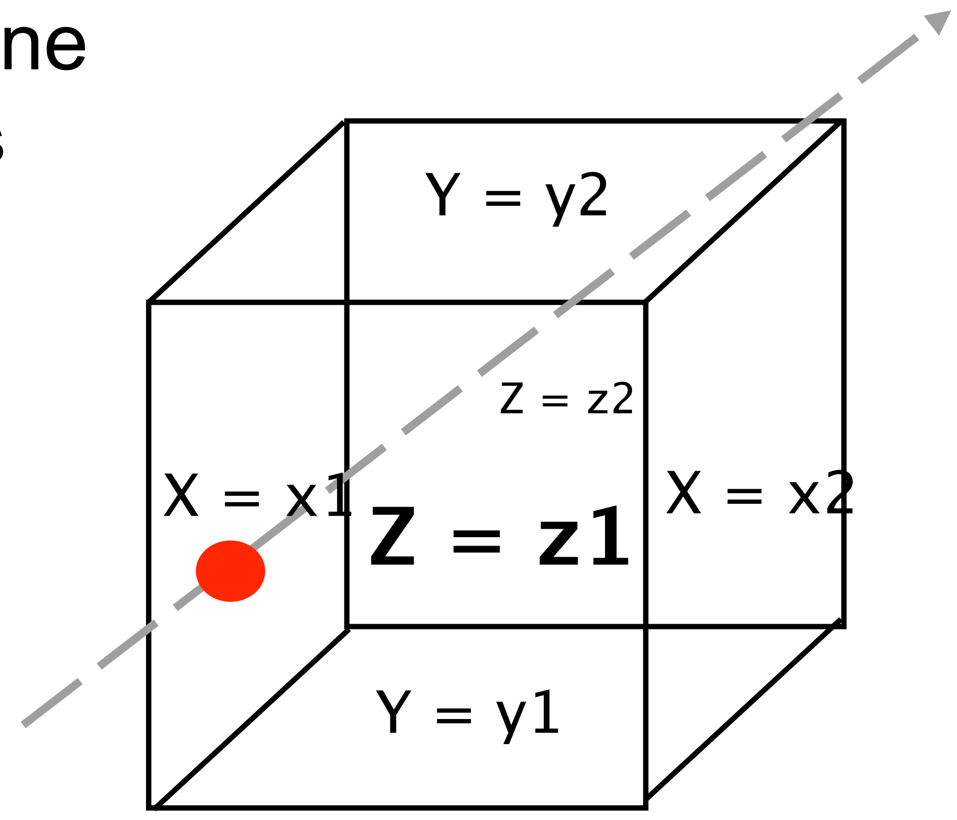
Ray-Box Intersection Test

- Intersect ray with each plane
  - Box is the union of 6 planes

    $x = x_1$, $x = x_2$

    $y = y_1$, $y = y_2$

    $z = z_1$, $z = z_2$



Ray-Box Intersection Test

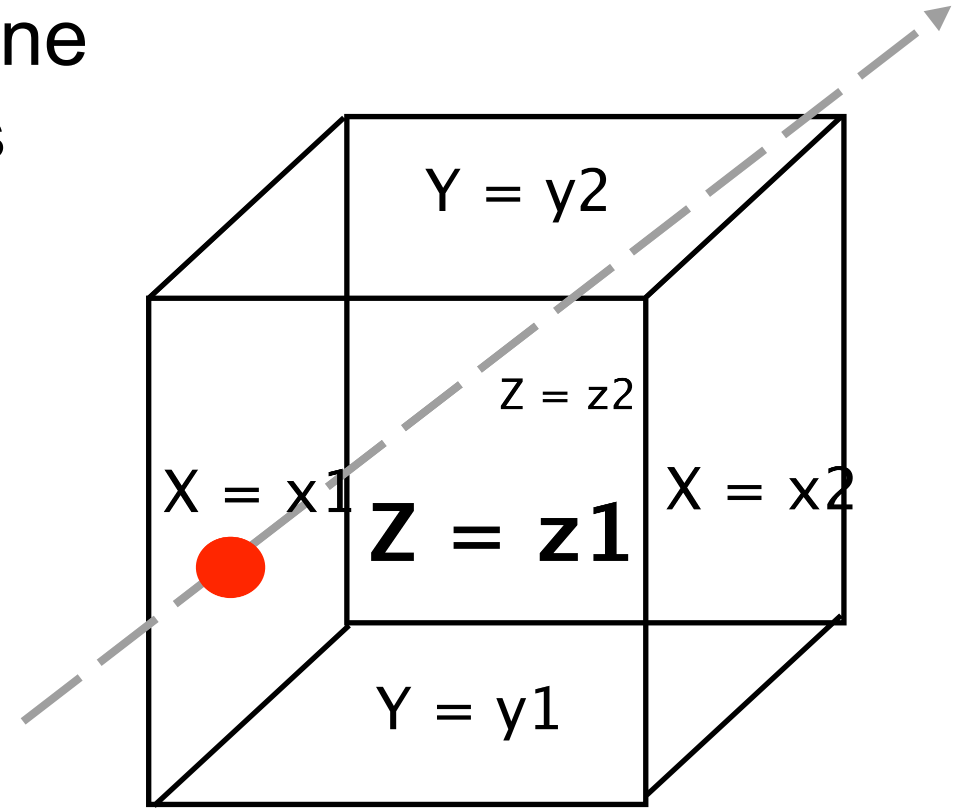
- Intersect ray with each plane
  - Box is the union of 6 planes

    $x = x_1$, $x = x_2$

    $y = y_1$, $y = y_2$

    $z = z_1$, $z = z_2$

- Ray/axis-aligned plane is easy:



Ray-Box Intersection Test

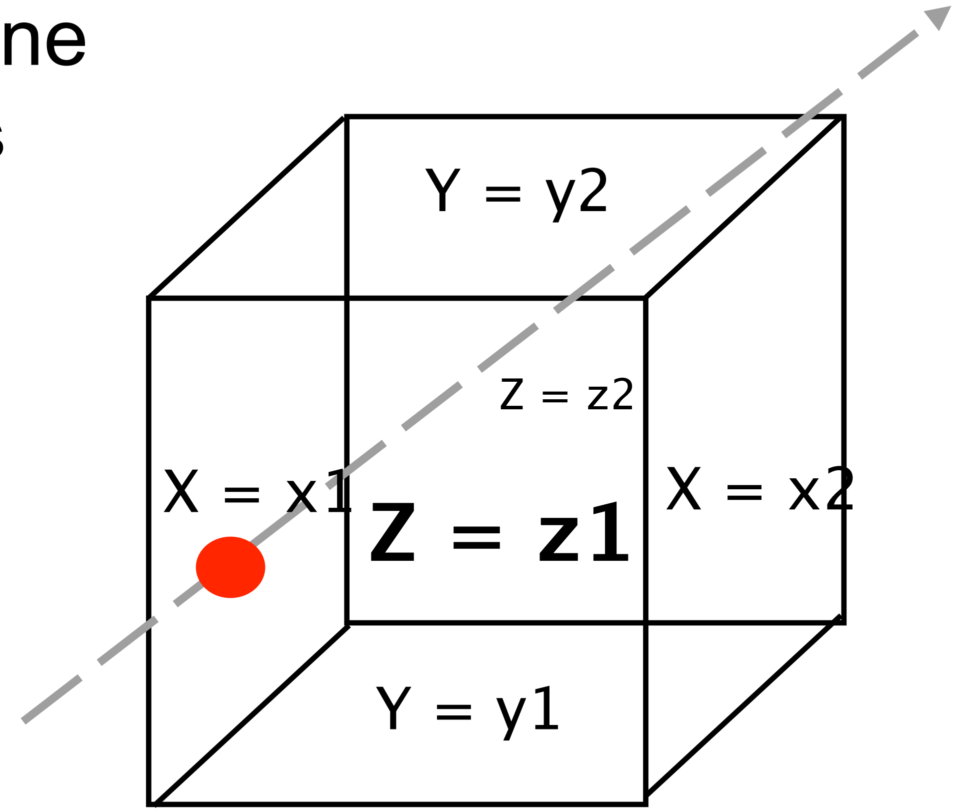
- Intersect ray with each plane
  - Box is the union of 6 planes

    $x = x_1,\ x = x_2$

    $y = y_1,\ y = y_2$

    $z = z_1,\ z = z_2$

- Ray/axis-aligned plane is easy:



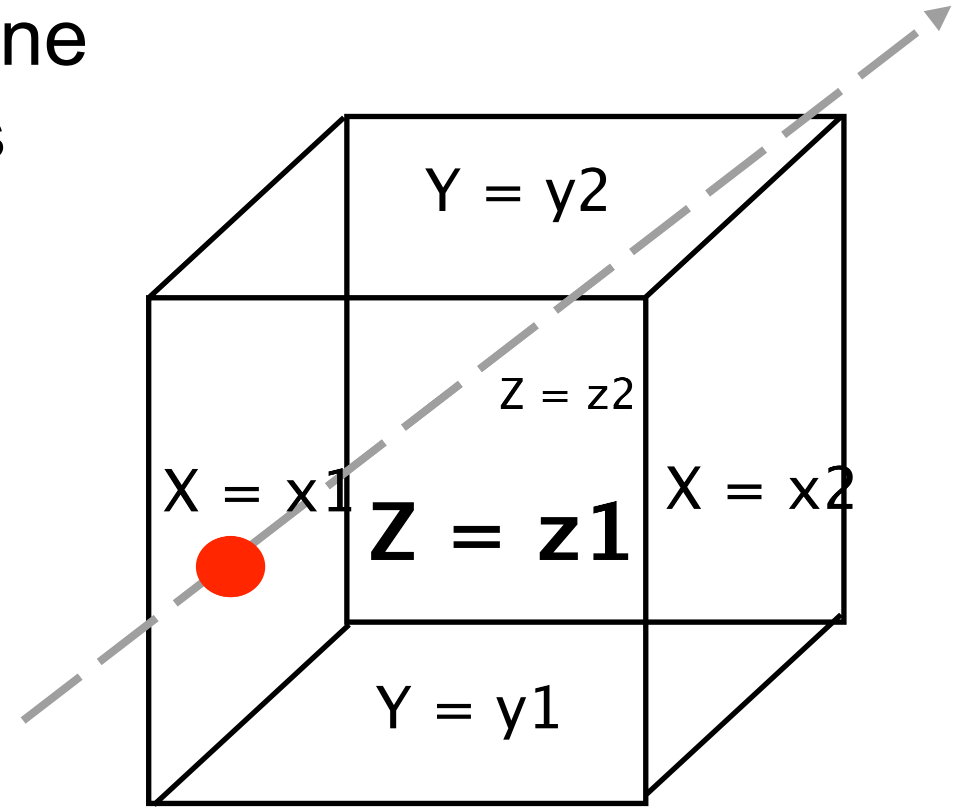
Ray-Box Intersection Test

- Intersect ray with each plane
  - Box is the union of 6 planes

    $x = x_1$, $x = x_2$

    $y = y_1$, $y = y_2$

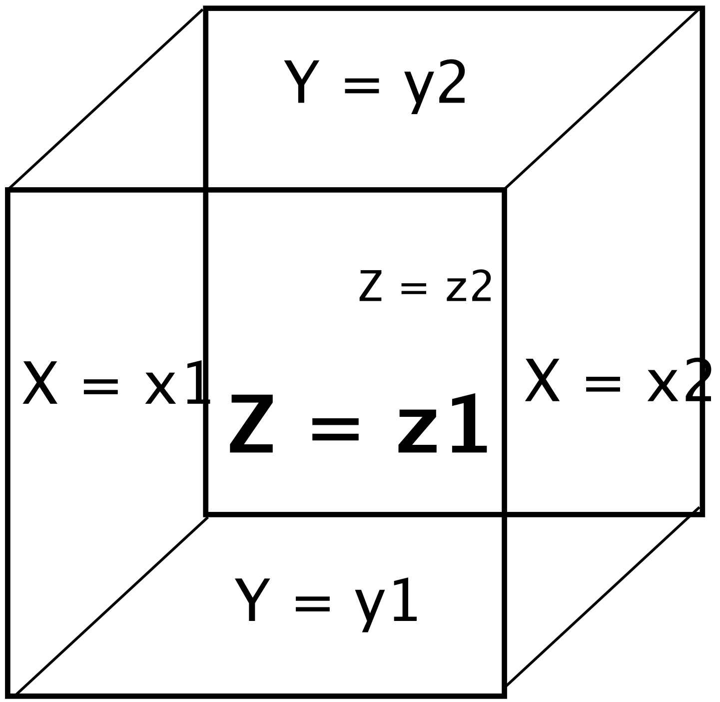
    $z = z_1$, $z = z_2$



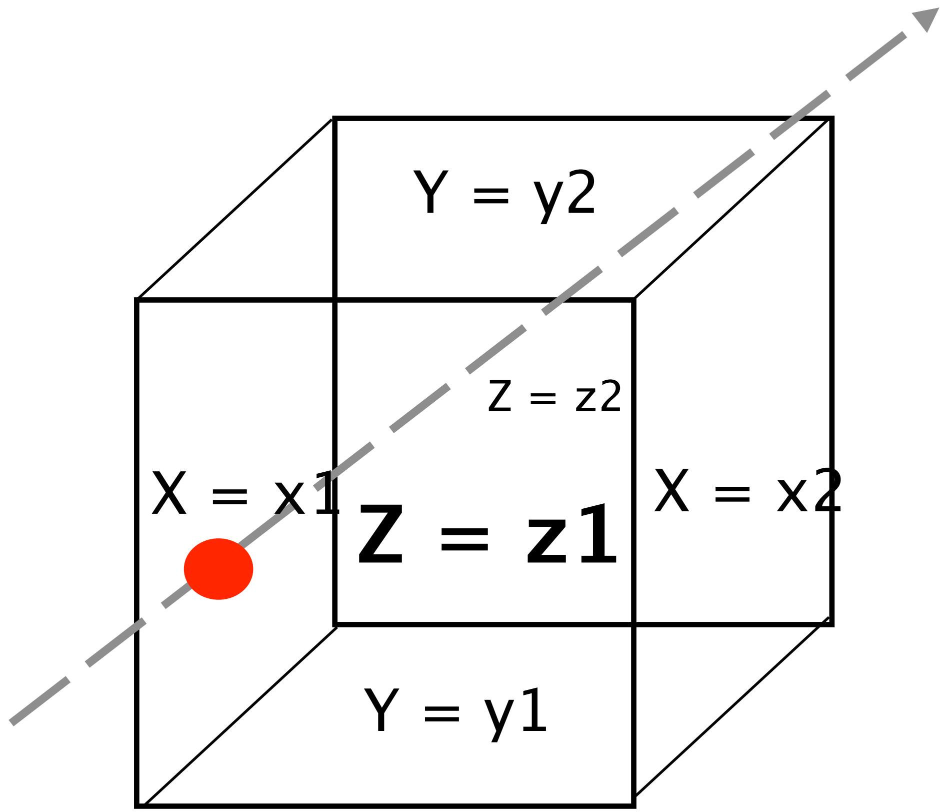
- Ray/axis-aligned plane is easy:

E.g., solve *x component:* $e_x + tD_x = x_1$

Ray-Box Intersection Test

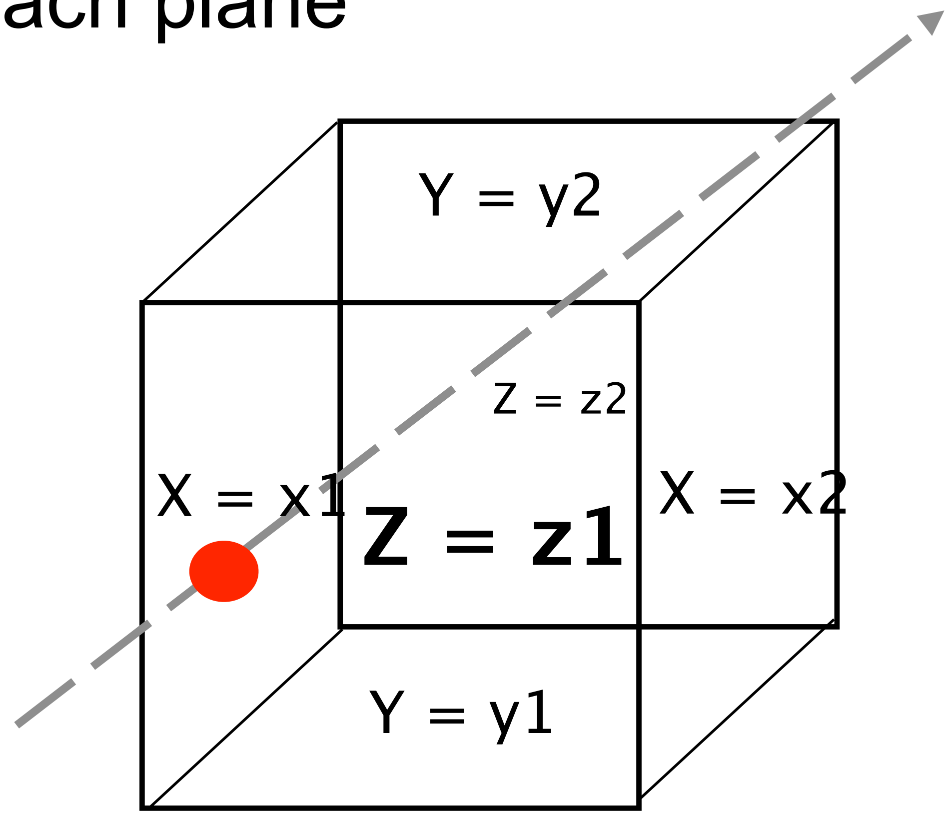


Ray-Box Intersection Test



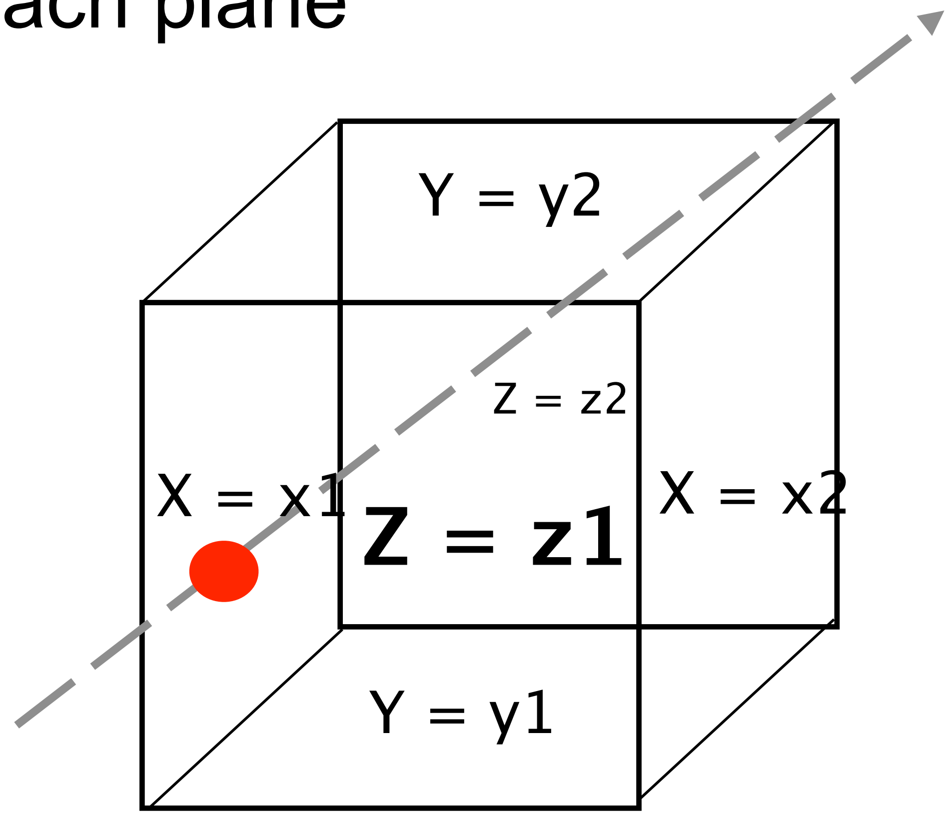
Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections



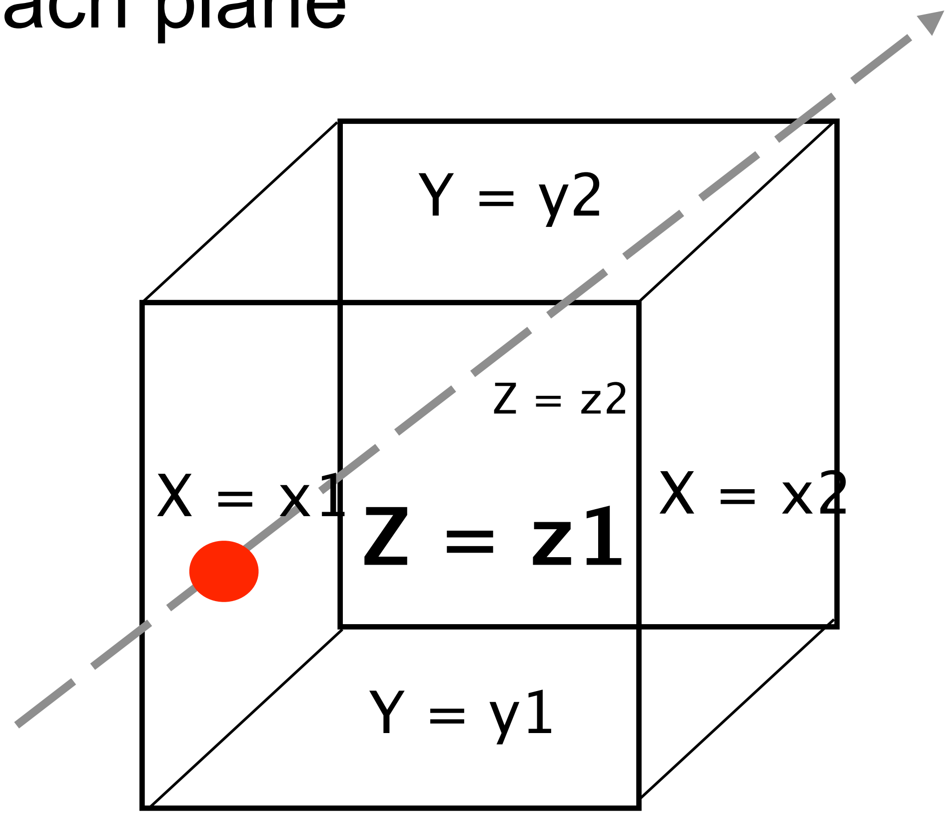
Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection



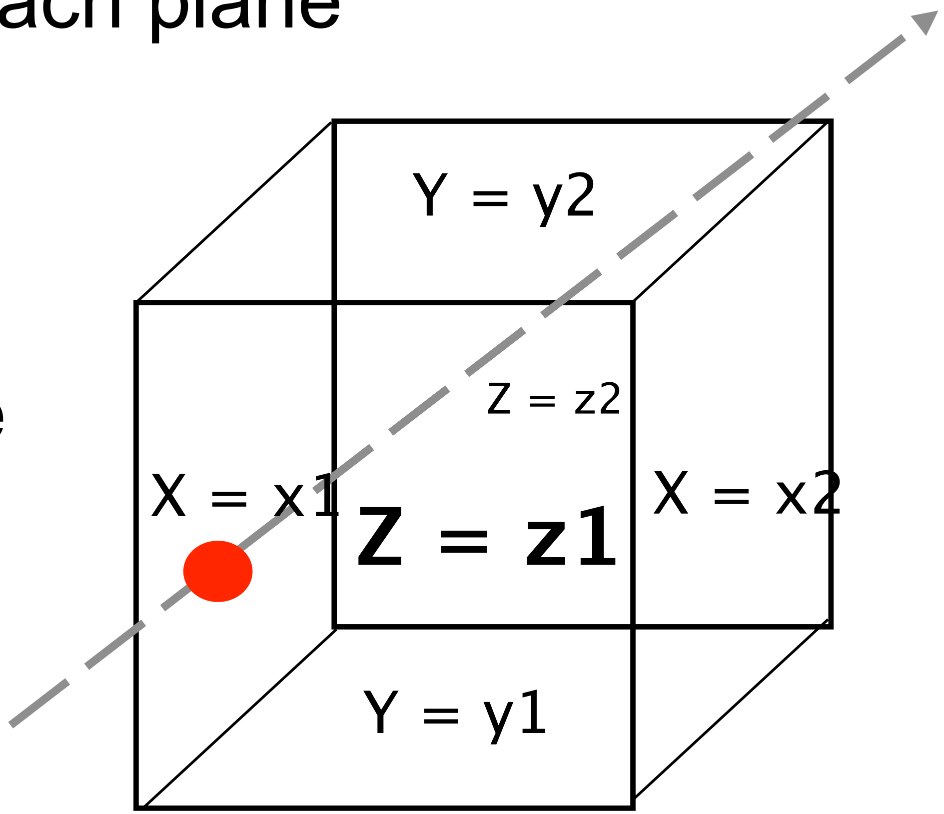
Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection
   with the smallest $t > 0$



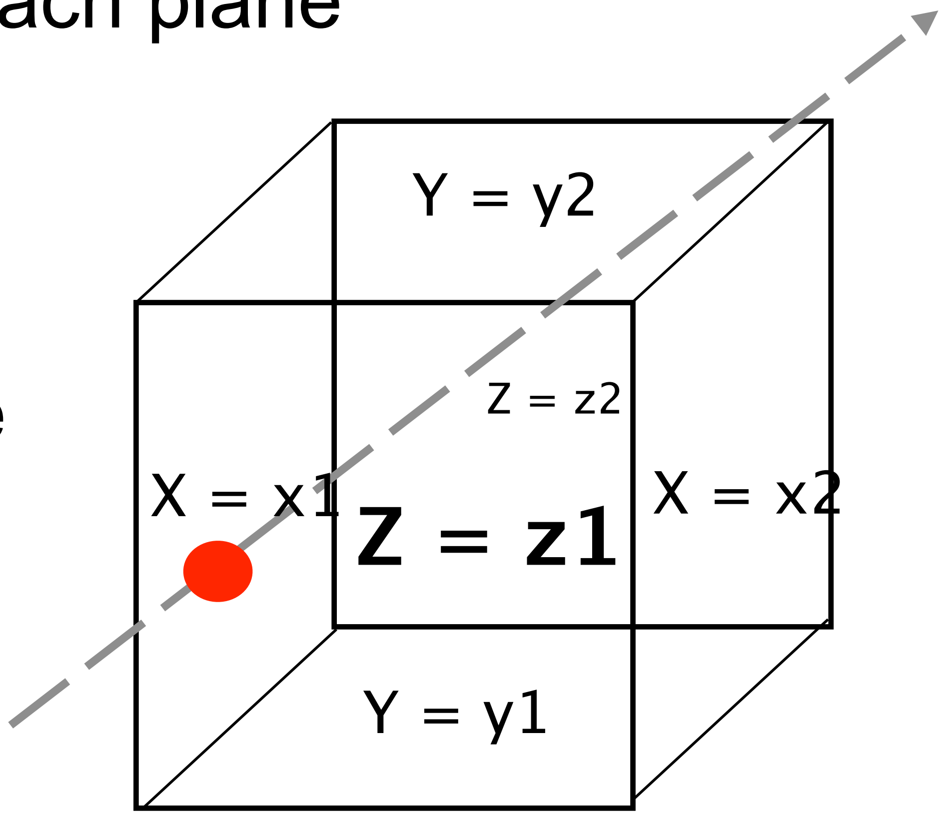
Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection
   with the smallest $t > 0$
   that is within the range



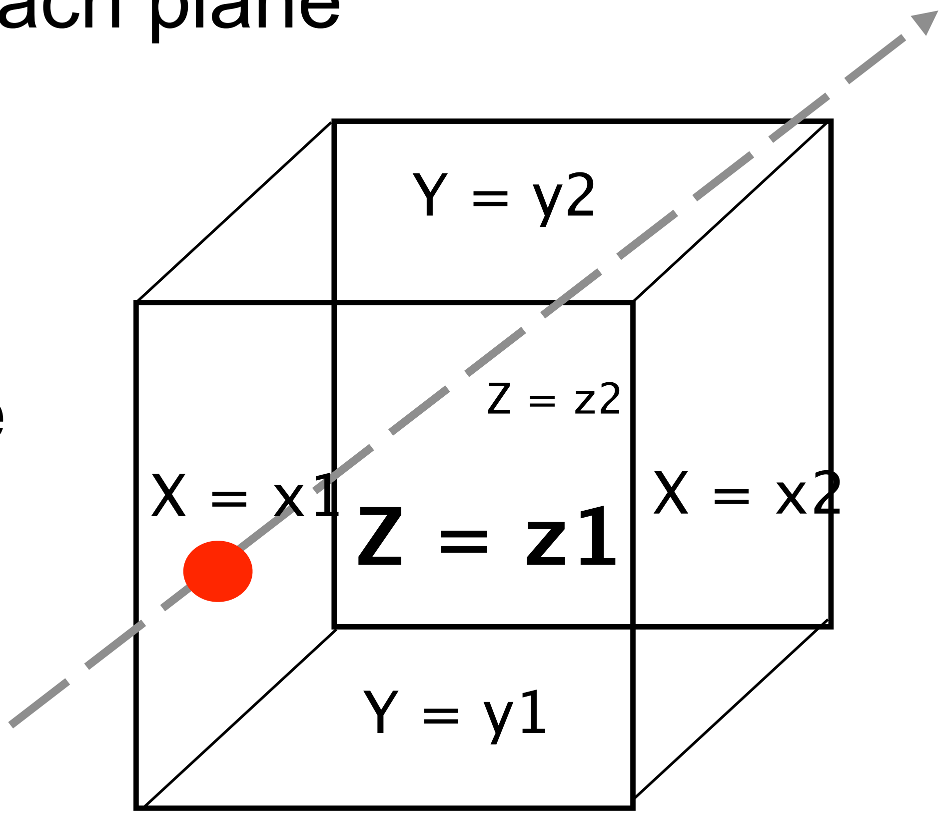
Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection with the smallest $t > 0$ that is within the range of the box



Ray-Box Intersection Test

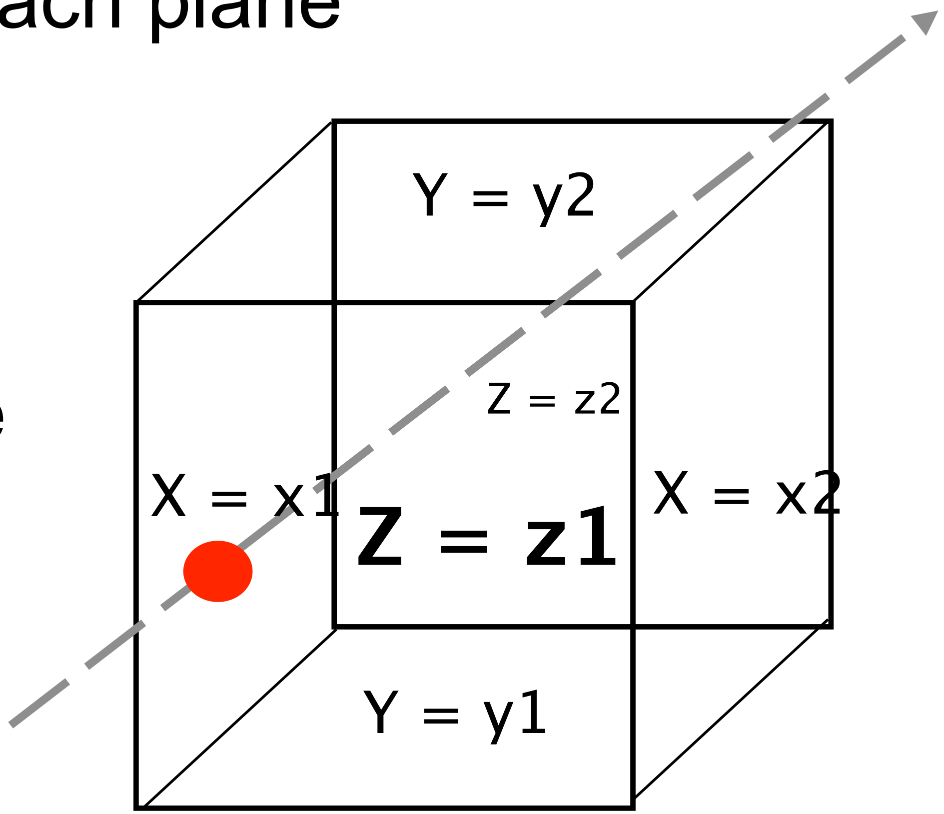
1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection
   with the smallest $t > 0$
   that is within the range
   of the box



Ray-Box Intersection Test

1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection
   with the smallest $t > 0$
   that is within the range
   of the box

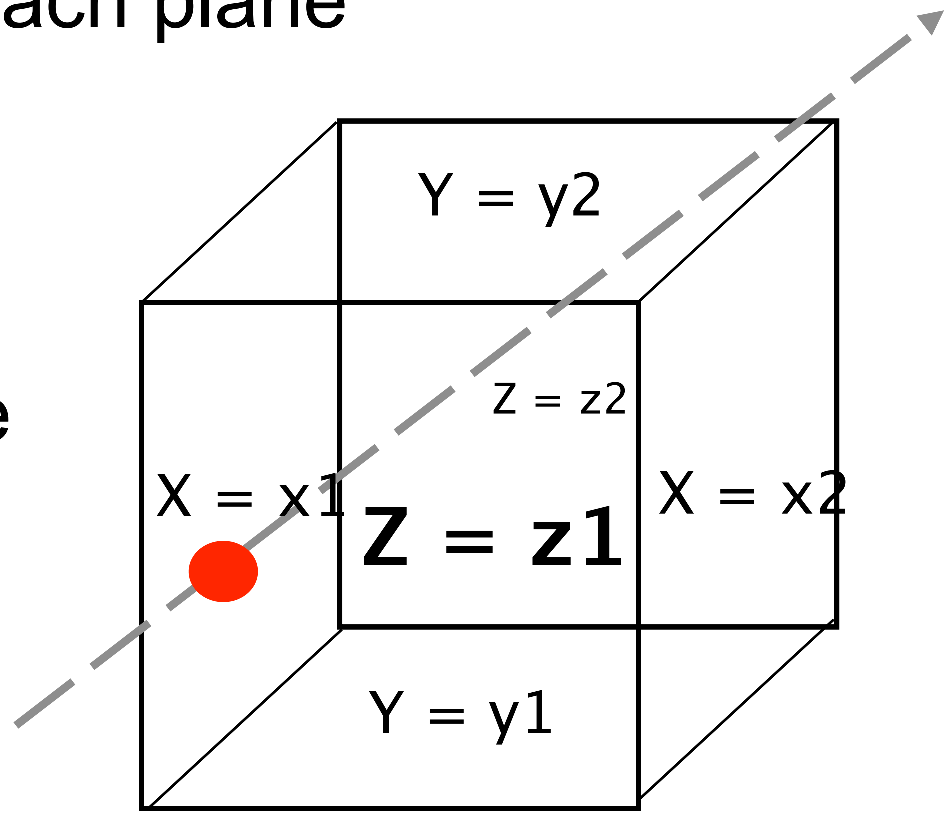
- We can do more



Ray-Box Intersection Test

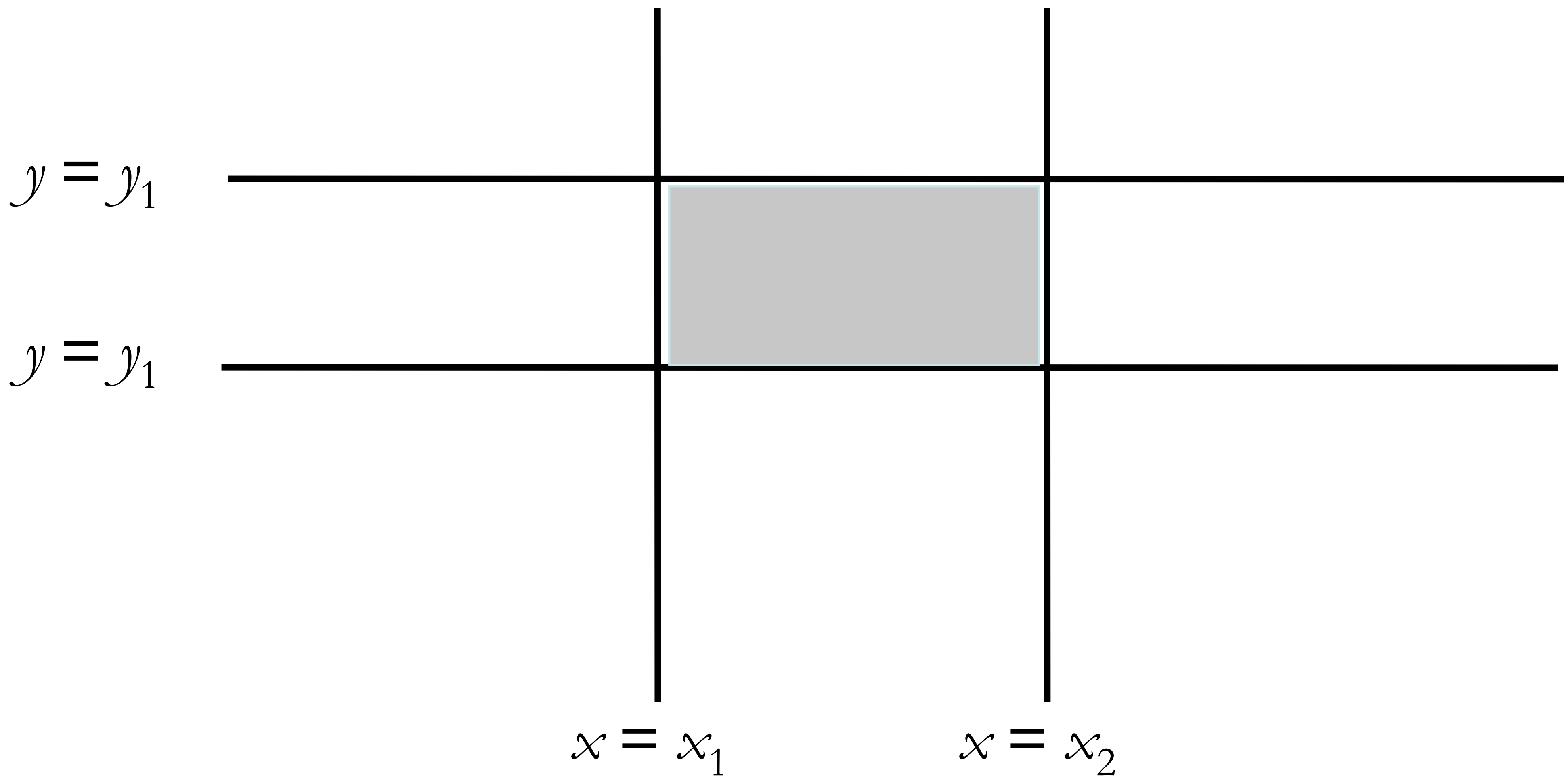
1. Intersect the ray with each plane
2. Sort the intersections
3. Choose intersection with the smallest $t > 0$ that is within the range of the box

- We can do more efficiently



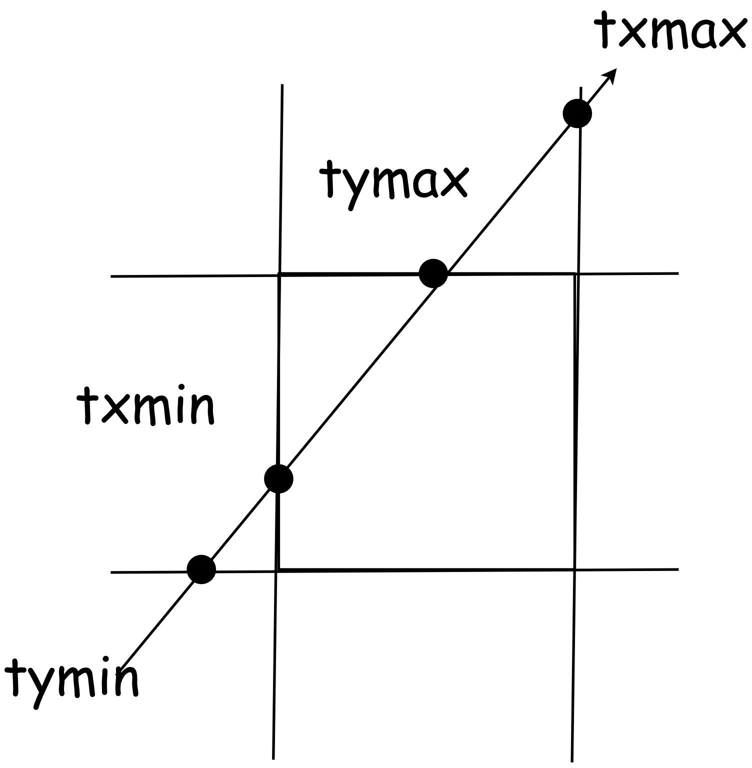
Only Consider 2D for Now

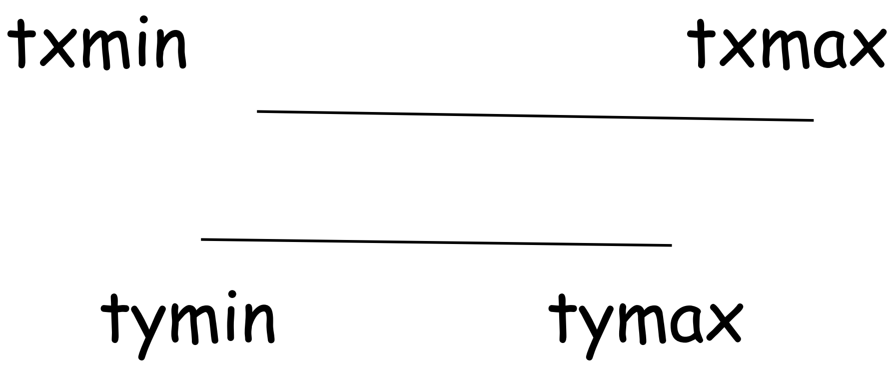
- if a point (x,y) is in the box, then (x,y) in $[x_1, x_2]$ x $[y_1, y_2]$



The Principle

- Assuming the ray hits the box boundary lines at intervals [txmin,txmax], [tymin,tymax], the ray hits the box if and only if the intersection of the two intervals is not empty





Pseudo Code

$t_{xmin} = (x_1 - e_x)/Dx$ //assume Dx >0

$t_{xmax} = (x_2 - e_x)/Dx$

$t_{ymin} = (y_1 - e_y)/Dy$

$t_{ymax} = (y_2 - e_y)/Dy$ //assume Dy >0

if $(t_{xmin} > t_{ymax})$ *or* $(t_{ymin} > t_{xmax})$

return false

else

return true

Pseudo Code

$t_{xmin} = (x_2 - e_x)/Dx$ //if Dx < 0

$t_{xmax} = (x_1 - e_x)/Dx$

$t_{ymin} = (y_2 - e_y)/Dy$ //if Dy < 0

$t_{ymax} = (y_1 - e_y)/Dy$

if $(t_{xmin} > t_{ymax})$ *or* $(t_{ymin} > t_{xmax})$

return false

else

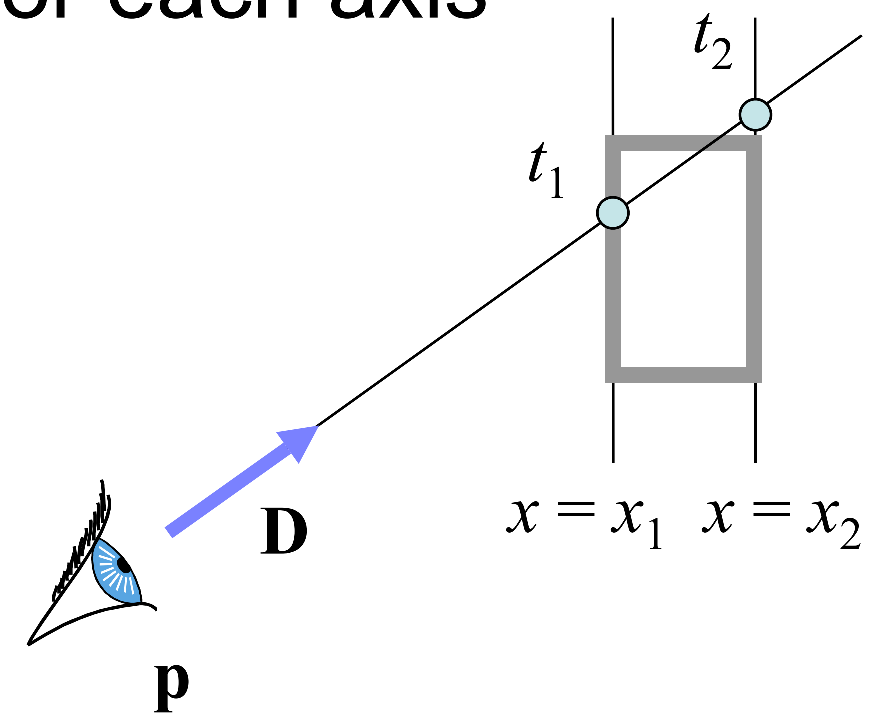
return true

Now Consider All Axis

- We will calculate t_1 and t_2 for each axis (x, y, and z)
- Update the intersection interval as we compute t1 and t2 for each axis
- remember:

  $t_1 = (x_1 - p_x)/D_x$

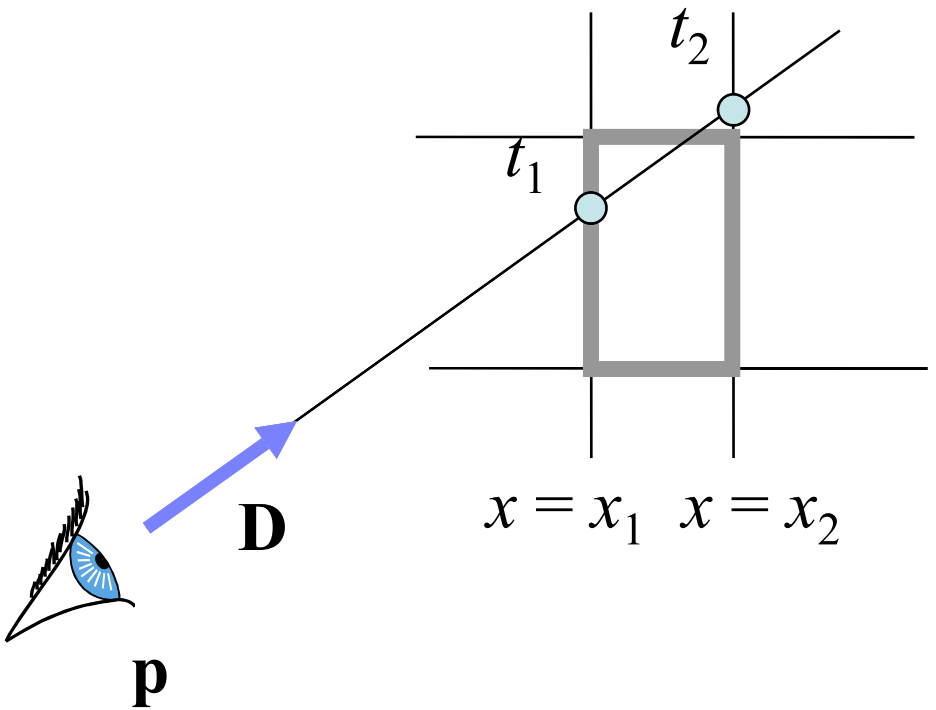
  $t_2 = (x_2 - p_x)/D_x$



Update [t_{near}, t_{far}]

- Set $t_{near} = -\infty$ and $t_{far} = +\infty$
- For each axis, compute t1 and t2
  - make sure t1 < t2
  - if $t_1 > t_{near}$, $t_{near} = t_1$
  - if $t_2 < t_{far}$, $t_{far} = t_2$

- If $t_{near} > t_{far}$, box is missed



Algorithm

Set $t_{near} = -\infty$, $t_{far} = \infty$

$R(\mathrm{t}) = p + t * \mathbf{D}$

For each pair of planes P associated with X, Y, and Z do: (example uses X planes)

if direction $\mathbf{D}_x = 0$ then

if ($p_x < x_1$ or $p_x > x_2$)

return FALSE

else

begin

$t_1 = (x_l - p_x) / \mathbf{D}_x$

$t_2 = (x_h - p_x) / \mathbf{D}_x$

if $t_1 > t_2$ then swap (t_1, t_2)

if $t_1 > t_{near}$ then $t_{near} = t_1$

if $t_2 < t_{far}$ then $t_{far} = t_2$

if $\mathrm{t}_{near} > t_{far}$ return FALSE

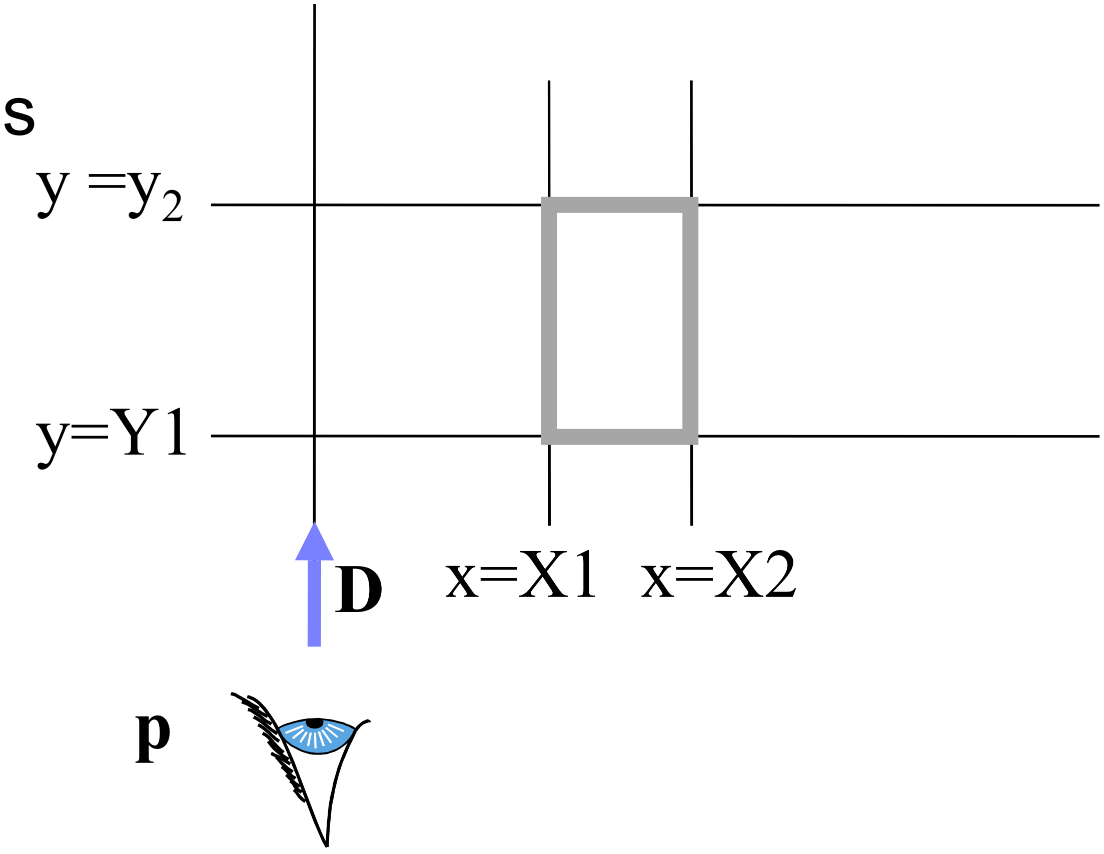
if $t_{far} < 0$ return FALSE

end

Return t_{near}

Special Case

- Ray is parallel to an axis
  - If $D_x = 0$ or $D_y = 0$ or $D_z = 0$
- $p_x < x_1$ or $p_x > x_2$ then miss



Special Case

- Box is behind the eye
  - If $t_{far} < 0$, box is behind