Raytracer Topics: Cameras and Raycasting

Adam Mally
CIS 560 Spring 2015
University of Pennsylvania

Basic Structure of a Raytracer

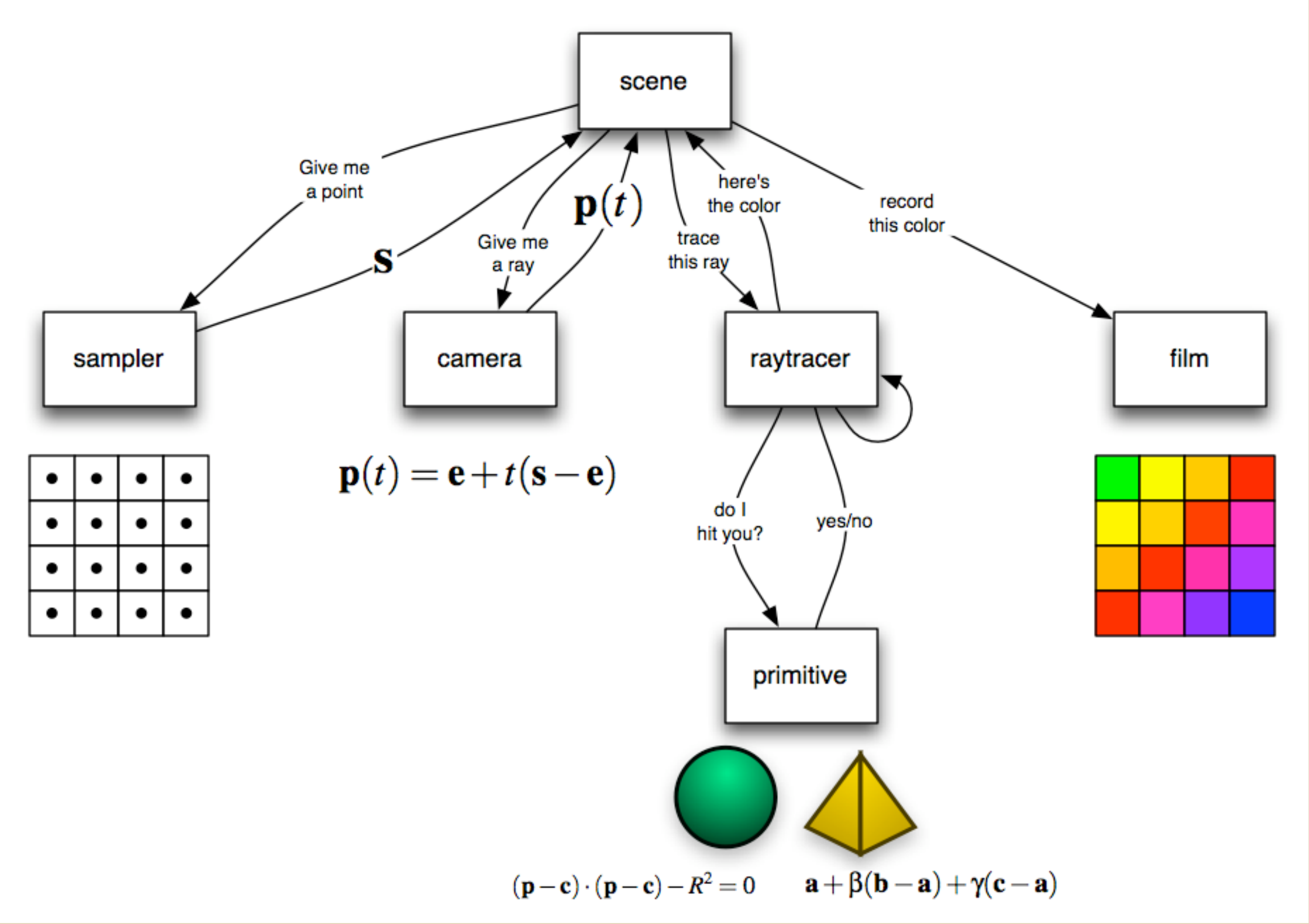


Image source: UC Berkeley CS184

Today’s Topics

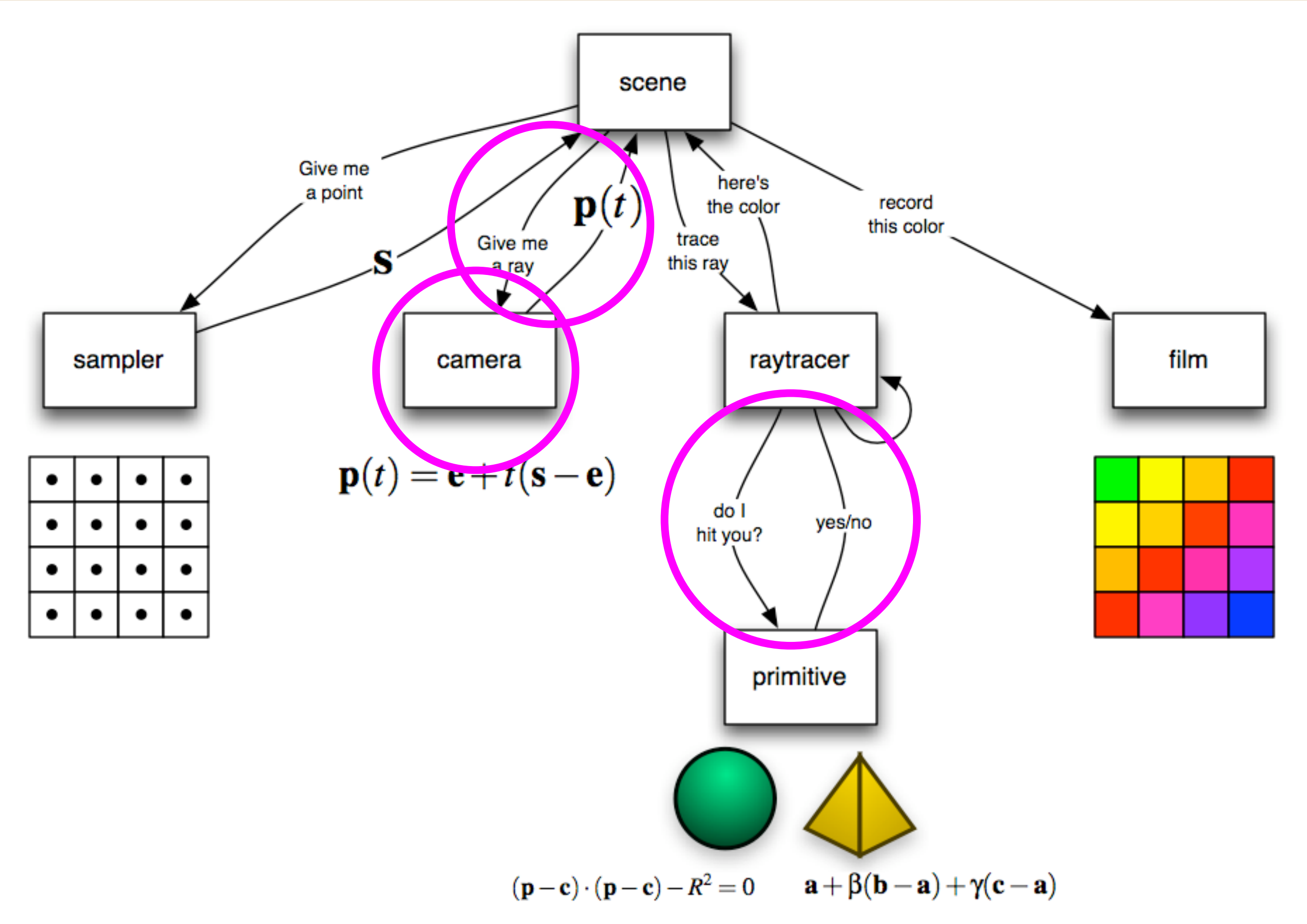
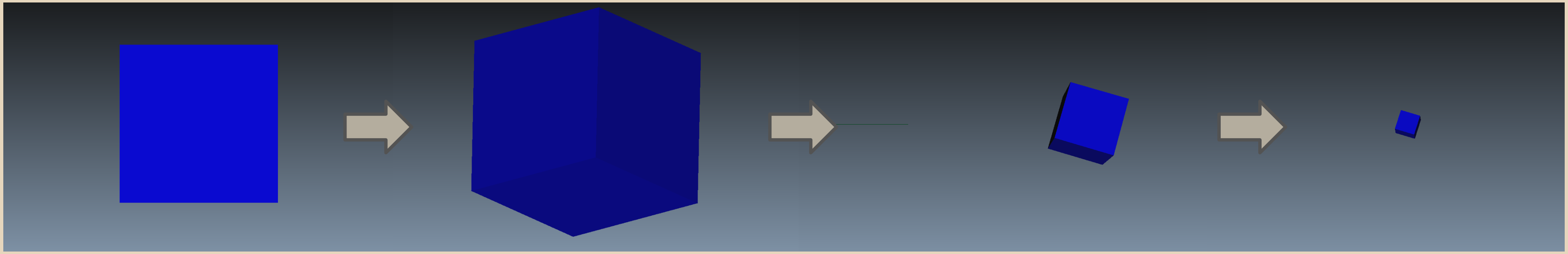


Image source: UC Berkeley CS184

Viewing a scene

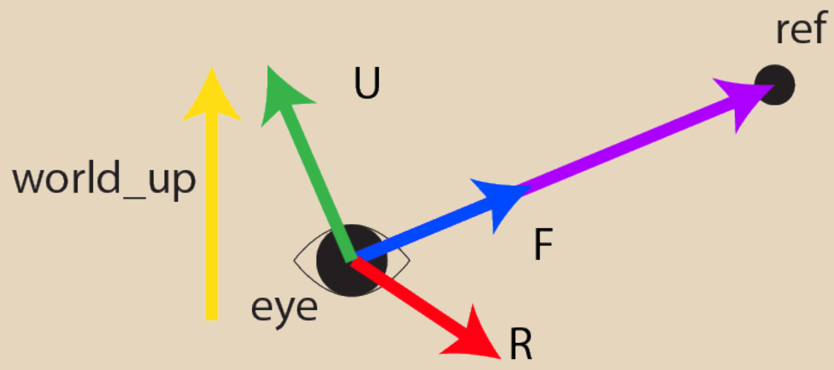
- Concatenate three matrices with your geometry to view your scene
- Projection_Mat * View_Mat * Model_Mat * Geometry
- Model matrix:
  - Object space -> world space
  - Scale, rotation, and translation in any combination
- View matrix:
  - Set up your camera's position and orientation
- Projection matrix:
  - Project objects from 3D space into the screen's 2D space
  - Screen space is in Normalized Device Coordinates



No matrices Model matrix View matrix Projection matrix

Camera: View matrix parameters

- Position in world coordinate system (commonly referred to as the eye position)
- (Optional) A point at which to look, commonly referred to as the reference point
- Look direction (F) - which way is the camera facing? Represents the local Z-axis
- An "up direction" in world coordinates, used for computing the camera's local axes
- Local right (R) - A direction in world coordinates that represents the direction that is "rightward" in the camera's local coordinates. Represents local X-axis
    - Perpendicular to the look direction
- Local up (U) - A direction in world coordinates that represents the direction that is "upward" in the camera's local coordinates. Represents local Y-axis
    - Perpendicular to both local right and look direction

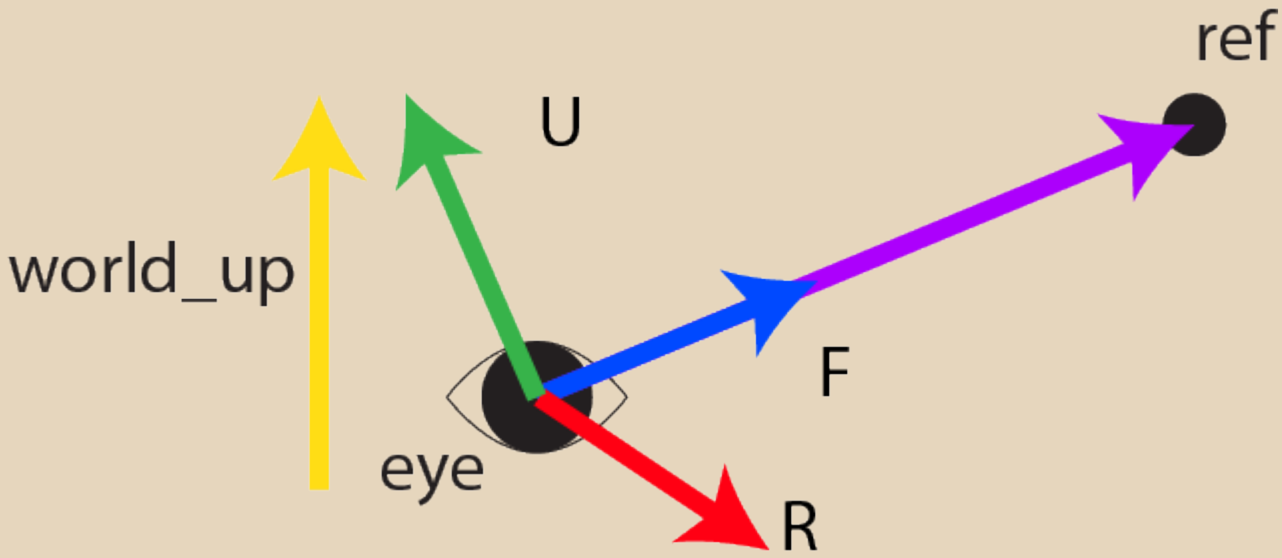


Camera: Axis computation

F = normalize(ref − eye)

R = normalize(F ✖ world_up)

U = normalize(R ✖ F)



Camera: View matrix computation

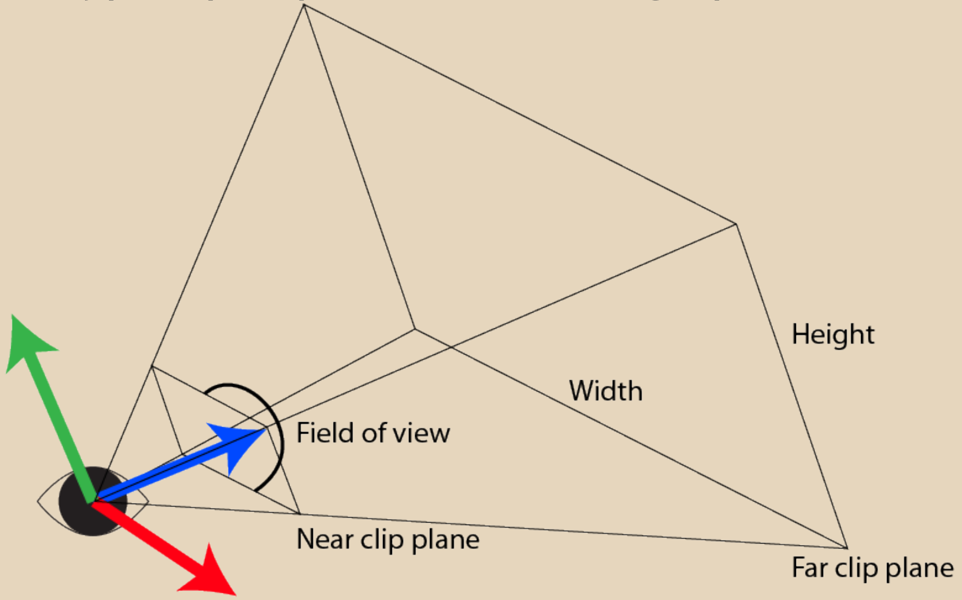
- View matrix transforms geometry from world space into camera space
  - The camera itself is not moving; the scene's geometry is transformed in the opposite direction and orientation of how you want the camera to move
  - Strictly speaking, the "camera" does not exist; you are always transforming the geometry in your scene
- View matrix is composed of two sub-matrices
  - Orientation matrix (O)
  - Translation matrix (T)

- $O = \begin{matrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$
- $T = \begin{matrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{matrix}$

- View matrix = O * T

Camera: Projection matrix

- Projection matrix transforms geometry from camera space into screen space (remember: NDC)
- Built from several camera components
  - Field of view - the angle that determines how quickly the frustum grows as it extends from the camera
  - Aspect ratio - screen width / screen height
  - Near clip and far clip planes
  - Projection type - perspective or orthographic



Camera: Viewing frustums

- Geometry outside the frustum is not visible to the camera
- Frustum shape affects how geometry is projected
- Since perspective projection frustums grow in size as they extend from the camera, geometry that is further from the camera appears to be smaller than geometry closer to the camera
  - The geometry gets further from the frustum bounds as they grow
- Orthographic frustums do not grow with distance, so a sphere 1 foot away and a sphere 10 feet away will look the same

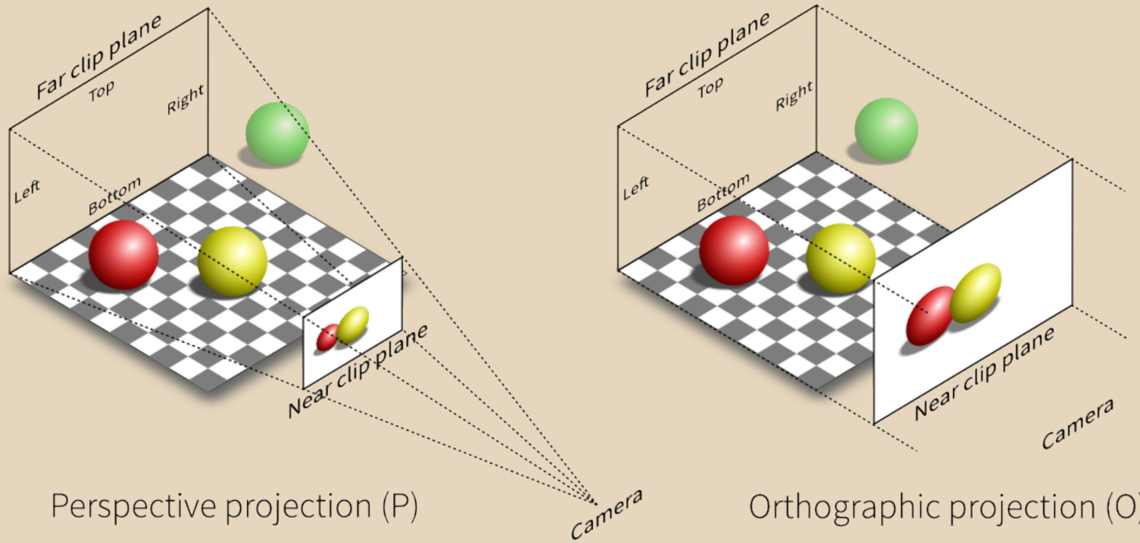


Image source: http://www.labri.fr/perso/nrougier/teaching/opengl/

Camera: Orthographic projection

- Given a near clip, far clip, top clip, bottom clip, and aspect ratio:

$$\begin{aligned} \text{bottom} &= -\text{top} \\ \text{right} &= \text{top} \times \text{aspect} \\ \text{left} &= -\text{right} \end{aligned}$$

$$P = \begin{pmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{1}{\text{far}-\text{near}} & -\frac{\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Maps (left, bottom, near) to (-1, -1, 0) and (right, top, far) to (1,1,1)

Camera: Orthographic projection

- Let's do some matrix multiplication to see how the clipping planes work
- Near clip: 1
- Far clip: 10
- Top: 10
- Aspect: 1:1
- Camera position: [0,0,0]
- Look vector: [0,0,1]

$$\begin{aligned} \text{bottom} &= -\text{top} \\ \text{right} &= \text{top} \times \text{aspect} \\ \text{left} &= -\text{right} \end{aligned}$$

$$P = \begin{pmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{1}{\text{far}-\text{near}} & -\frac{\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
      0.1  0    0      0
      0    0.1  0      0
P =   0    0    0.111  -0.111
      0    0    0      1
```

- Note how the upper-right element and the element below that always zero out, since they're `(right + left)/(numbers)` and `(top + bottom)/(numbers)`, which are equivalent to `(right - right)/(numbers)` and `(top - top)/(numbers)`

Camera: Orthographic projection

$$\begin{vmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.111 & -0.111 \\ 0 & 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} -2 \\ 5 \\ 0.5 \\ 1 \end{vmatrix} = \begin{vmatrix} 0.1*-2 + 0*5 + 0*0.5 + 0*1 \\ 0*-2 + 0.1*5 + 0*0.5 + 0*1 \\ 0*-2 + 0*5 + 0.111*0.5 + -0.111*1 \\ 0*-2 + 0*5 + 0*0.5 + 1*1 \end{vmatrix}$$

$$= \begin{vmatrix} -0.2 \\ 0.5 \\ -0.0555 \\ 1 \end{vmatrix}$$

- Since the Z range of the screen is [0,1], we cannot see this point as its Z coordinate is negative
- Remember, though: the X and Y ranges are [-1, 1]
- Similarly, a point that is past the far clip plane would have a Z coordinate greater than 1 after being transformed by this matrix
- Overall, any point in space outside the frustum is mapped to a point outside the range <-1,-1,0> to <1,1,1>

Camera: Perspective projection

- Given a near clip, far clip, FOV, and aspect ratio:

$$
\begin{aligned}
\text{top} &= \text{near} \times \tan(\text{FOV}/2) \\
\text{bottom} &= -\,\text{top} \\
\text{right} &= \text{top} \times \text{aspect} \\
\text{left} &= -\,\text{right}
\end{aligned}
$$

$$
P = \begin{pmatrix}
\frac{2\times\text{near}}{\text{right}-\text{left}} & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0 \\
0 & \frac{2\times\text{near}}{\text{top}-\text{bottom}} & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 \\
0 & 0 & \frac{\text{far}}{\text{far}-\text{near}} & -\frac{\text{far}\times\text{near}}{\text{far}-\text{near}} \\
0 & 0 & 1 & 0
\end{pmatrix}
$$

- Maps (left, bottom, near) to (-1, -1, 0) and (right, top, near) to (1,1,0)

Camera: Perspective projection

- Let’s do some matrix multiplication to see how this scales geometry
- Near clip: 1
- Far clip: 10
- FOV: 90
- Aspect: 1:1
- Camera position: [0,0,0]
- Look vector: [0,0,1]

$$
\begin{aligned}
\text{top} &= \text{near} \times \tan(\text{FOV}/2) \\
\text{bottom} &= -\text{top} \\
\text{right} &= \text{top} \times \text{aspect} \\
\text{left} &= -\text{right}
\end{aligned}
$$

$$
P = \begin{pmatrix}
\frac{2\times\text{near}}{\text{right}-\text{left}} & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} & 0 \\
0 & \frac{2\times\text{near}}{\text{top}-\text{bottom}} & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 \\
0 & 0 & \frac{\text{far}}{\text{far}-\text{near}} & -\frac{\text{far}\times\text{near}}{\text{far}-\text{near}} \\
0 & 0 & 1 & 0
\end{pmatrix}
$$

```
     1   0   0       0
     0   1   0       0
P =  0   0   1.111  -1.111
     0   0   1       0
```

Camera: Perspective projection

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1.111 & -1.111 \\ 0 & 0 & 1 & 0 \end{vmatrix} * \begin{vmatrix} -1 \\ 1.5 \\ 2 \\ 1 \end{vmatrix} = \begin{vmatrix} 1*-1 + 0*1.5 + 0*2 + 0*1 \\ 0*-1 + 1*1.5 + 0*2 + 0*1 \\ 0*-1 + 0*1.5 + 1.111*2 + -1.111*1 \\ 0*-1 + 0*1.5 + 1*2 + 0*1 \end{vmatrix}$$

$$= \begin{vmatrix} -1 \\ 1.5 \\ 1.111 \\ 2 \end{vmatrix} = \begin{vmatrix} -0.5 \\ 0.75 \\ 0.556 \\ 1 \end{vmatrix}$$

- In order to maintain homogeneous coordinates, we must divide the entire vector by its *w* component so *w* remains 1
- Note how the last row of the matrix has a 1 in the Z component, which causes the *w* component of the resultant vector to scale with the Z coordinate of the point we’re transforming

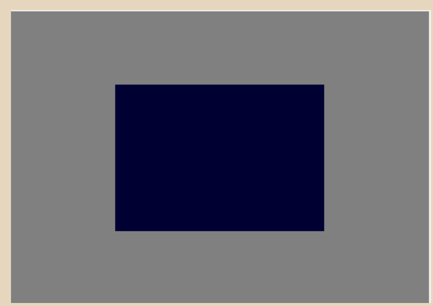
Camera matrices

What would we see if we were to draw a 1x1x1 cube centered at the origin with neither view nor projection matrices?

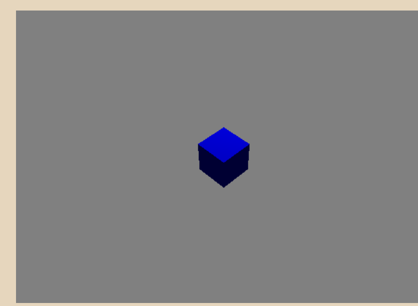
1

2

3

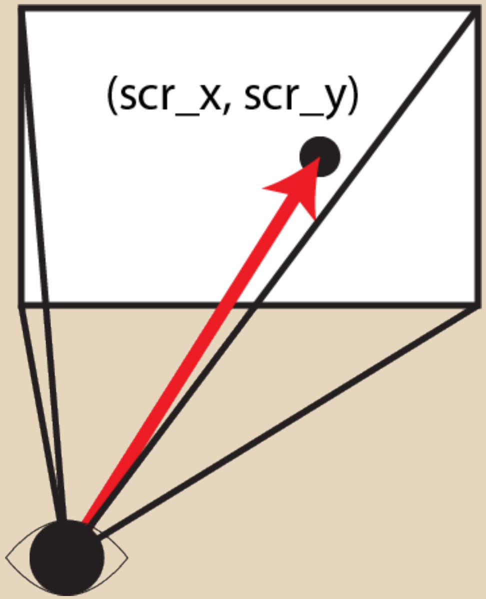


4



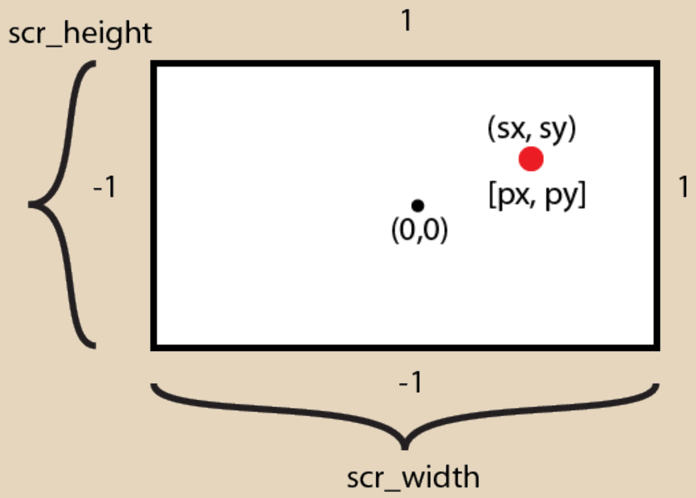
What is raycasting?

- Creating a line that passes through the viewing frustum and travels from the eye to some endpoint on a slice of the frustum (e.g. the far clip plane)
- The line's endpoint is determined by the pixel on our screen from which we want to raycast



Normalized device coordinates

- Recall that your GL window ranges from -1 to 1 on both the X and Y axes
- We can convert to NDC from any given pixel



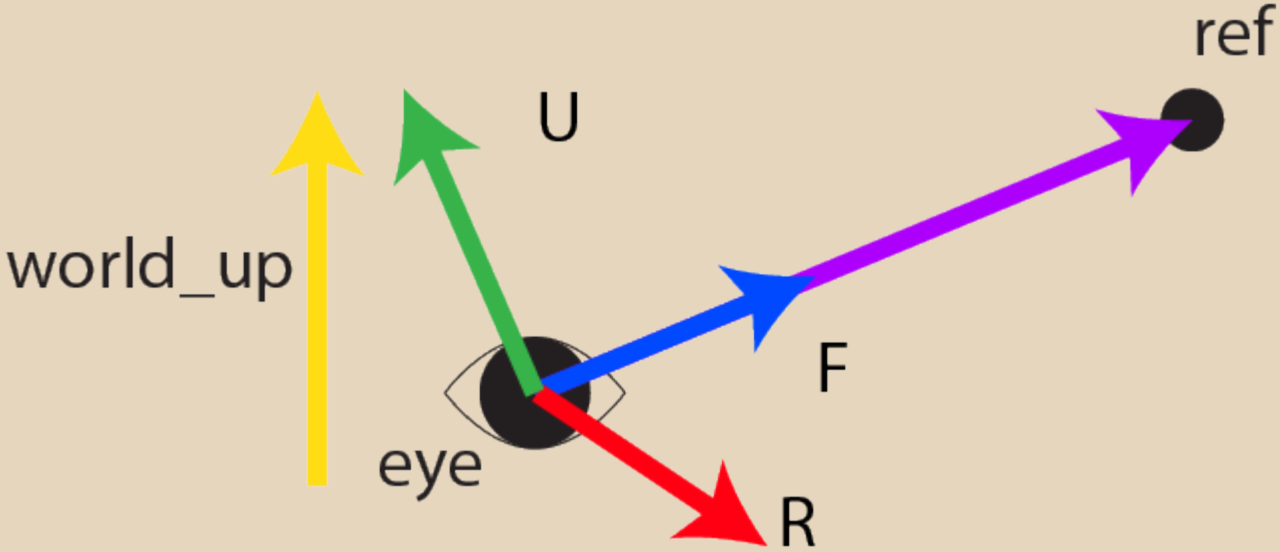
- sx = (2 * px/scr_width) – 1
- sy = 1 – (2 * py/scr_height)
- px and py are the given pixel’s x and y coordinates

Recap: Camera axes

F = normalize(ref – eye)

R = normalize(F x world_up)

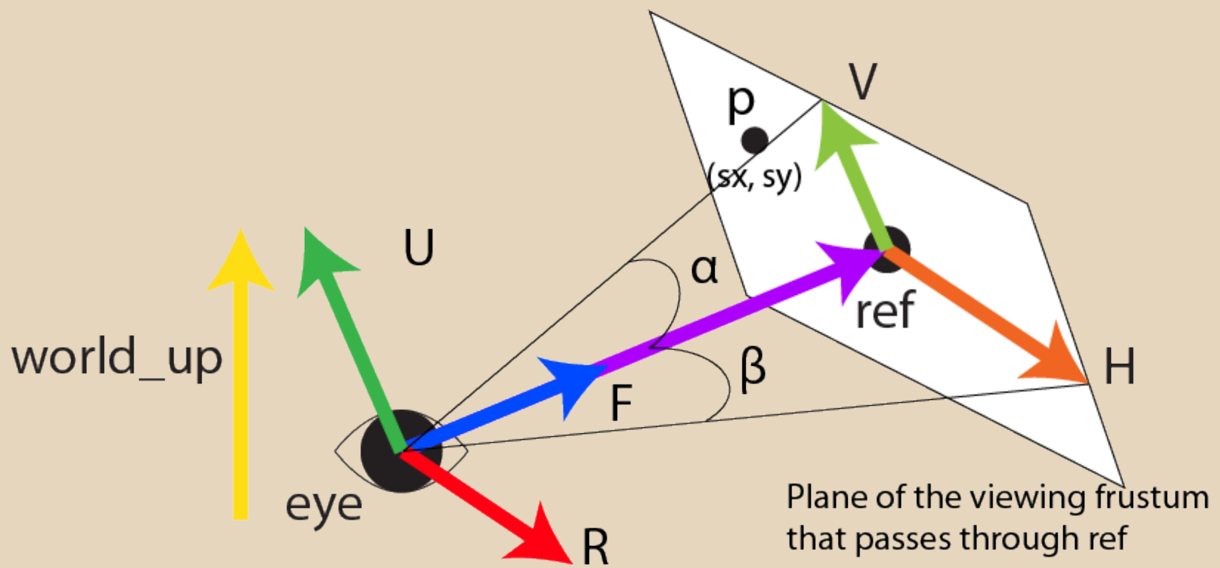
U = normalize(R x F)



Screen point to world point

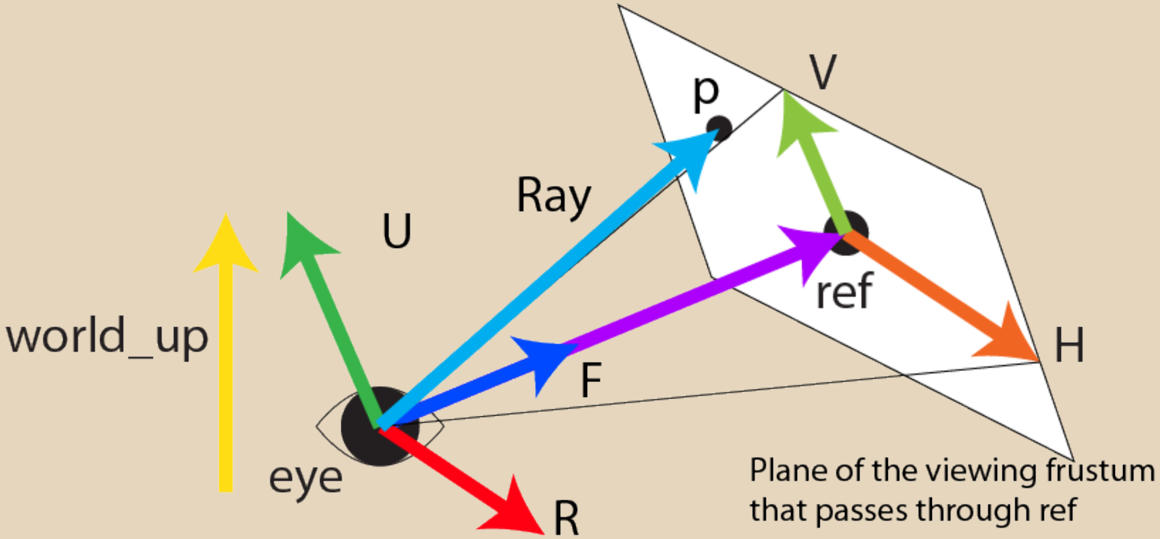
- len = |ref - eye|
- V = U*len*tan(α)
- H = R*len*aspect*tan(α)
- α = FOVY/2

- p = ref + sx*H + sy*V
- sx, sy are in NDC



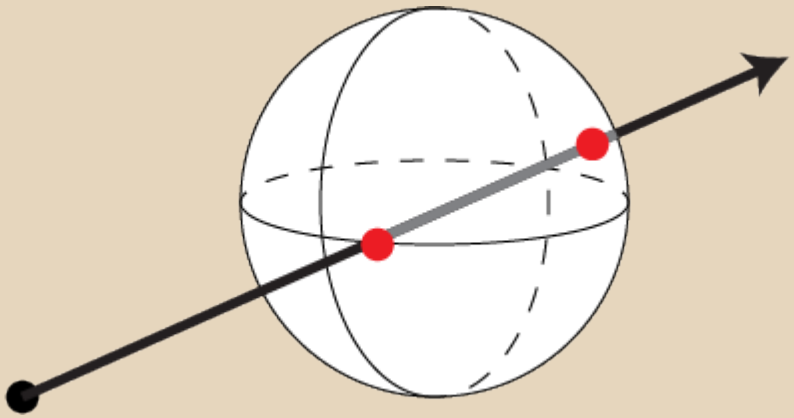
Getting a ray from the world point

- `ray_origin = eye`
- `ray_direction = normalize(p - eye)`
- `Arbitrary point on ray = eye + t*ray_direction`



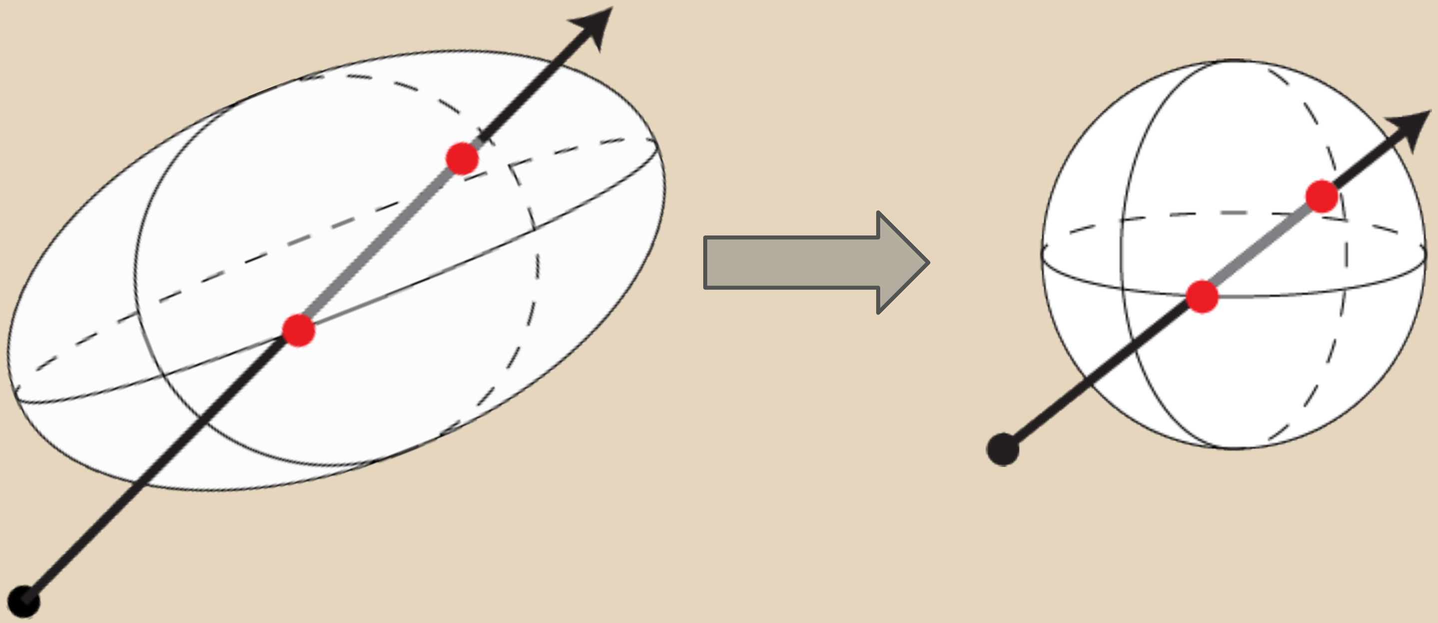
Ray-polygon intersection

- Most common intersection test is ray-triangle
- We'll also cover ray-sphere and ray-cube
    - All three are commonly used in basic raytracer testing



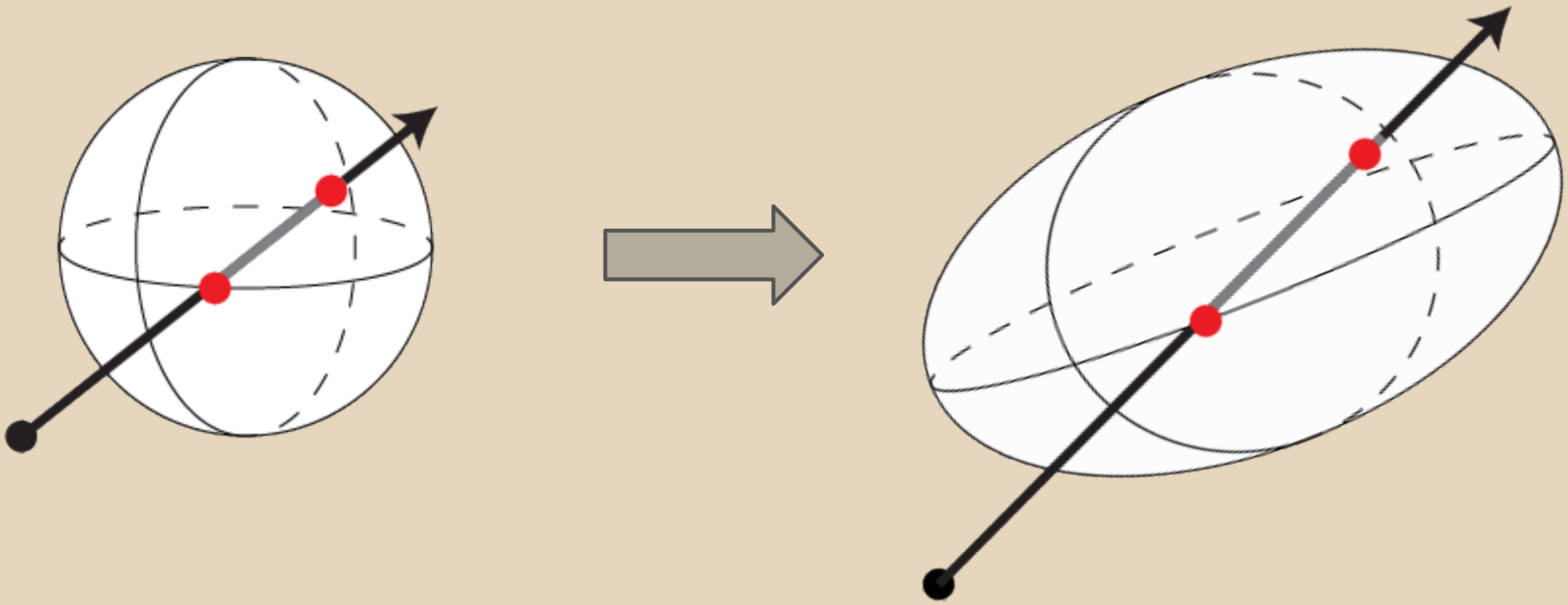
Frames of reference

- Before you try to test a ray against primitive geometry, you must first transform the ray so from its perspective, the geometry in question **is** primitive
- Simply transform the ray's direction and origin by the *inverse* of the geometry's model matrix



Frames of reference

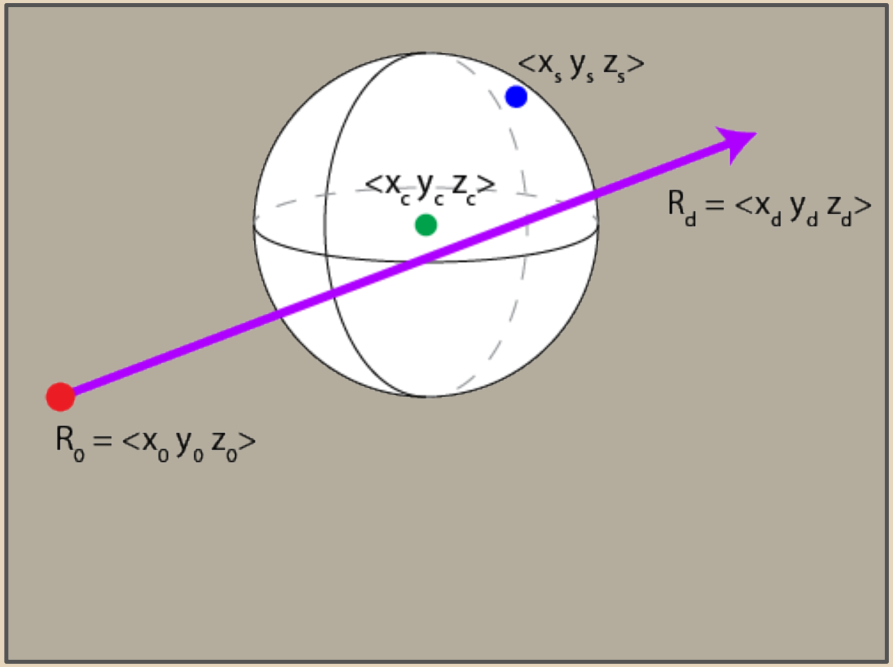
- Similarly, make sure you transform the results of your intersection test back into world space (e.g. the point of intersection, the surface normal at the intersection, etc.)



Ray-sphere intersection

- Sphere defined as $(x_s - x_c)^2 + (y_s - y_c)^2 + (z_s - z_c)^2 = r_s^{\,2}$
  - Sphere center = $<x_c\ y_c\ z_c>$
  - All points on the sphere surface = $<x_s\ y_s\ z_s>$
  - r_s is the sphere’s radius

- Ray defined as: $R_0 + t * R_d$
  - $R_0 = <x_0\ y_0\ z_0>$
  - $R_d = <x_d\ y_d\ z_d>$
  - t is a parameterization of R_d (i.e. a float)



Ray-sphere intersection

- Substitute $\langle x_s y_s z_s \rangle$ for the ray equation:

$$(x_0 + t*x_d - x_c)^2 + (y_0 + t*y_d - y_c)^2 + (z_0 + t*z_d - z_c)^2 = r_s^2$$

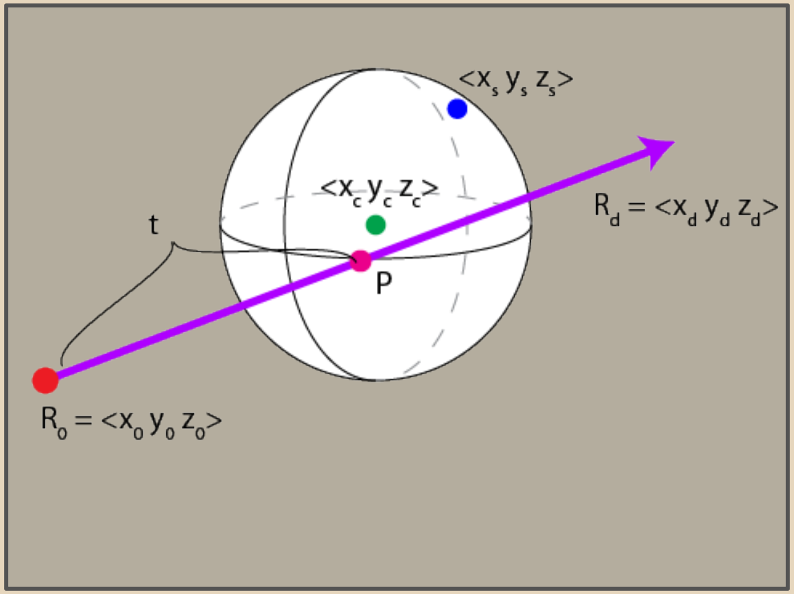
- Can also be written as:
- $At^2 + Bt + C = 0$
  - $A = x_d^2 + y_d^2 + z_d^2$
  - $B = 2(x_d(x_0 - x_c) + y_d(y_0 - y_c) + z_d(z_0 - z_c))$
  - $C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r_s^2$
- Note that we now have a quadratic equation
  - We can solve for t using the quadratic formula!

Ray-sphere intersection

- t_0 , t_1 = $(-B \pm \sqrt{(B^2-4AC)})/(2A)$
  - t_0 is for the - case and t_1 is for the + case
- Remember: if the discriminant is negative, then there is no real root and therefore no intersection
  - Discriminant = B^2-4AC
- If t_0 is positive, then we're done. If not, then compute t_1.

Ray-sphere intersection

- Once we have t, we can plug it into our ray equation to find the closest point of intersection on our sphere.
  - If all we care about is whether or not we hit the sphere, we can just check:
  - near_clip < t < far_clip
- $P = R_0 + t * R_d$



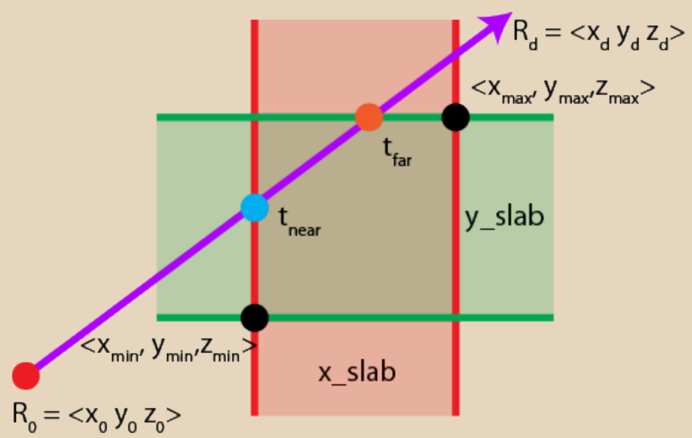
Converting from local to world space

- It is important to remember that the t value we have found is in **object space** (i.e. only valid for an untransformed sphere)
- To find a t value in world space, we can use our object-space t to find the point of intersection P_o on our unit sphere
- We can then transform P_o by our sphere's transformation matrix to find the point of intersection in world space P_w
- We can use P_w to compute our world-space t value as follows:
  - world_t = length(P_w - camera.eye)
  - Remember that camera.eye is the origin of your untransformed ray in world space

Ray-cube intersection

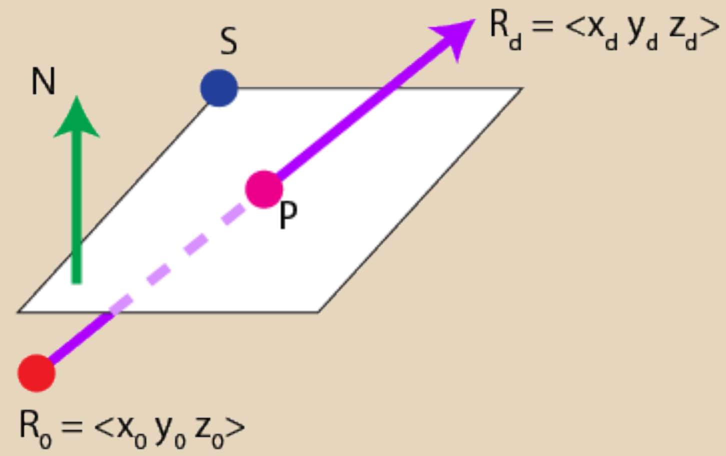
- Begin by storing t_{near} = -infinity and t_{far} = infinity
- For each pair of planes associated with the X, Y, and Z axes (the example uses the X “slab”):
  - If x_d is 0, then the ray is parallel to the X slab, so
    - If $x_0 < x_{min}$ or $x_0 > x_{max}$ then we miss
  - $t_0 = (x_{min} - x_0)/x_d$
  - $t_1 = (x_{max} - x_0)/x_d$
  - If $t_0 > t_1$ then swap them
  - If $t_0 > t_{near}$ then $t_{near} = t_0$
  - If $t_1 < t_{far}$ then $t_{far} = t_1$
  - Repeat for Y and Z
  - If $t_{near} > t_{far}$ then we miss the box

We can’t see the z_slab because it’s coming directly out of the screen



Ray-plane intersection

- Plane defined as: dot(N,(P-S)) = 0
  - N is the plane's normal
  - S is some point on the plane
  - P is the point of intersection
- Ray defined as: R_0 + t * R_d
- Substitute P for ray:
  - dot(N,(R_0 + t * R_d - S)) = 0
- Solve for t:
  - t = dot(N,(S - R_0)) / dot(N,R_d)



Point-in-triangle

- Use **barycentric coordinates** to test if P is within the bounds of a triangle
    - The **barycenter** of a triangle is its center of mass, often given unequal weighting to its vertices
- $S = \text{area}(P_1, P_2, P_3)$
- $S_1 = \text{area}(P, P_2, P_3)/S$
- $S_2 = \text{area}(P, P_3, P_1)/S$
- $S_3 = \text{area}(P, P_1, P_2)/S$
- Therefore, $P = S_1P_1 + S_2P_2 + S_3P_3$
- So, P is within the triangle if:
    - $0 \leq S_1 \leq 1$
    - $0 \leq S_2 \leq 1$
    - $0 \leq S_3 \leq 1$
    - $S_1 + S_2 + S_3 = 1$