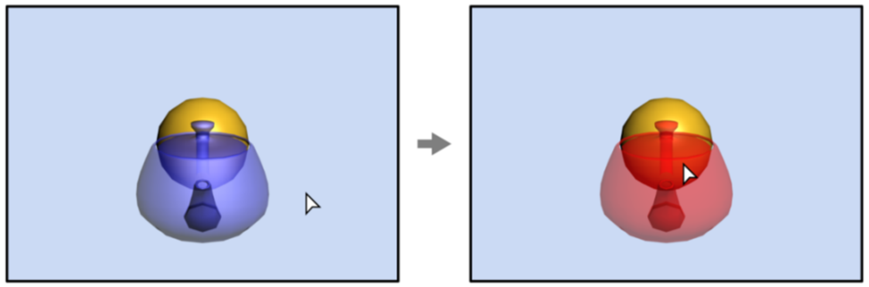
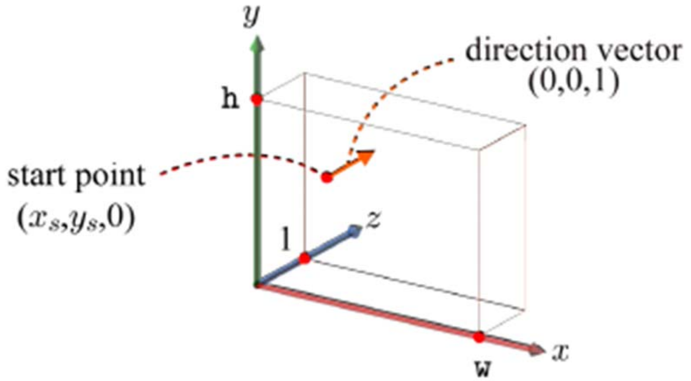
Screen-space Object Handling

Object Picking

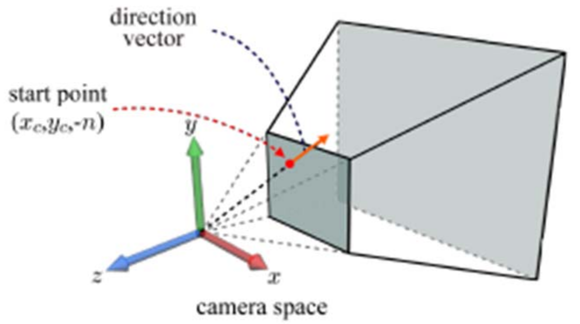
- On touchscreen, an object can be picked by tapping it with a finger. In PC screen, mouse clicking is popularly used for the same purpose.
- Mouse clicking simply returns the 2D pixel coordinates (x_s,y_s). Given (x_s,y_s), we can consider a *ray* described by the start point (x_s,y_s,0) and the direction vector (0,0,1).





- We need ray-object intersection tests to find the object *first* hit by the ray, which is the teapot in the example. However, it is not efficient to compute the intersection in the screen space because the screen-space information available to the GPU is about the triangles of each object, not the object itself.
- Therefore, the test is taken on by CPU. As the CPU has every object in the pre-transformed state, i.e., in the object space, the screen-space ray should be transformed back to the object space.

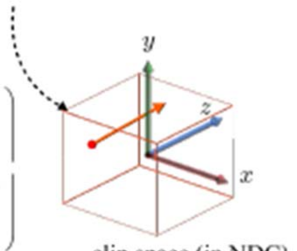
From the Screen Space to the Camera Space



projection transform

$$\begin{pmatrix} m_{11} & 0 & 0 & 0\\ 0 & m_{22} & 0 & 0\\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n}\\ 0 & 0 & -1 & 0\end{pmatrix}$$

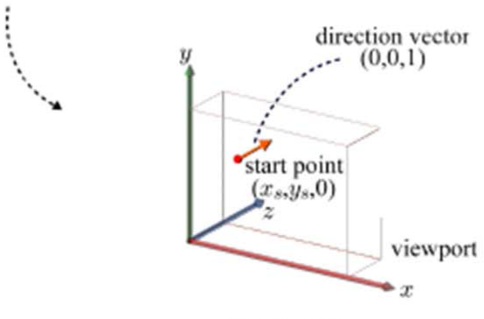
$$\boxed{\begin{matrix}\dfrac{cot(\frac{fovy}{2})}{aspect} \rightarrow m_{11}\\ cot(\frac{fovy}{2}) \rightarrow m_{22}\end{matrix}}$$



clip space (in NDC)

viewport transform

$$\begin{pmatrix} \frac{W}{2} & 0 & 0 & \frac{W}{2}\\ 0 & \frac{h}{2} & 0 & \frac{h}{2}\\ 0 & 0 & \frac{1}{2} & \frac{1}{2}\\ 0 & 0 & 0 & 1\end{pmatrix}$$



screen space

$$\begin{pmatrix} m_{11} & 0 & 0 & 0\\ 0 & m_{22} & 0 & 0\\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n}\\ 0 & 0 & -1 & 0\end{pmatrix}\begin{pmatrix} x_c\\ y_c\\ -n\\ 1\end{pmatrix} = \begin{pmatrix} m_{11}x_c\\ m_{22}y_c\\ -n\\ n\end{pmatrix} \rightarrow \begin{pmatrix} \frac{m_{11}x_c}{n}\\ \frac{m_{22}y_c}{n}\\ -1\\ 1\end{pmatrix}$$

$$\begin{pmatrix} \frac{W}{2} & 0 & 0 & \frac{W}{2}\\ 0 & \frac{h}{2} & 0 & \frac{h}{2}\\ 0 & 0 & \frac{1}{2} & \frac{1}{2}\\ 0 & 0 & 0 & 1\end{pmatrix}\begin{pmatrix} \frac{m_{11}x_c}{n}\\ \frac{m_{22}y_c}{n}\\ -1\\ 1\end{pmatrix} = \begin{pmatrix} \frac{W}{2}(\frac{m_{11}x_c}{n}+1)\\ \frac{h}{2}(\frac{m_{22}y_c}{n}+1)\\ 0\\ 1\end{pmatrix}$$

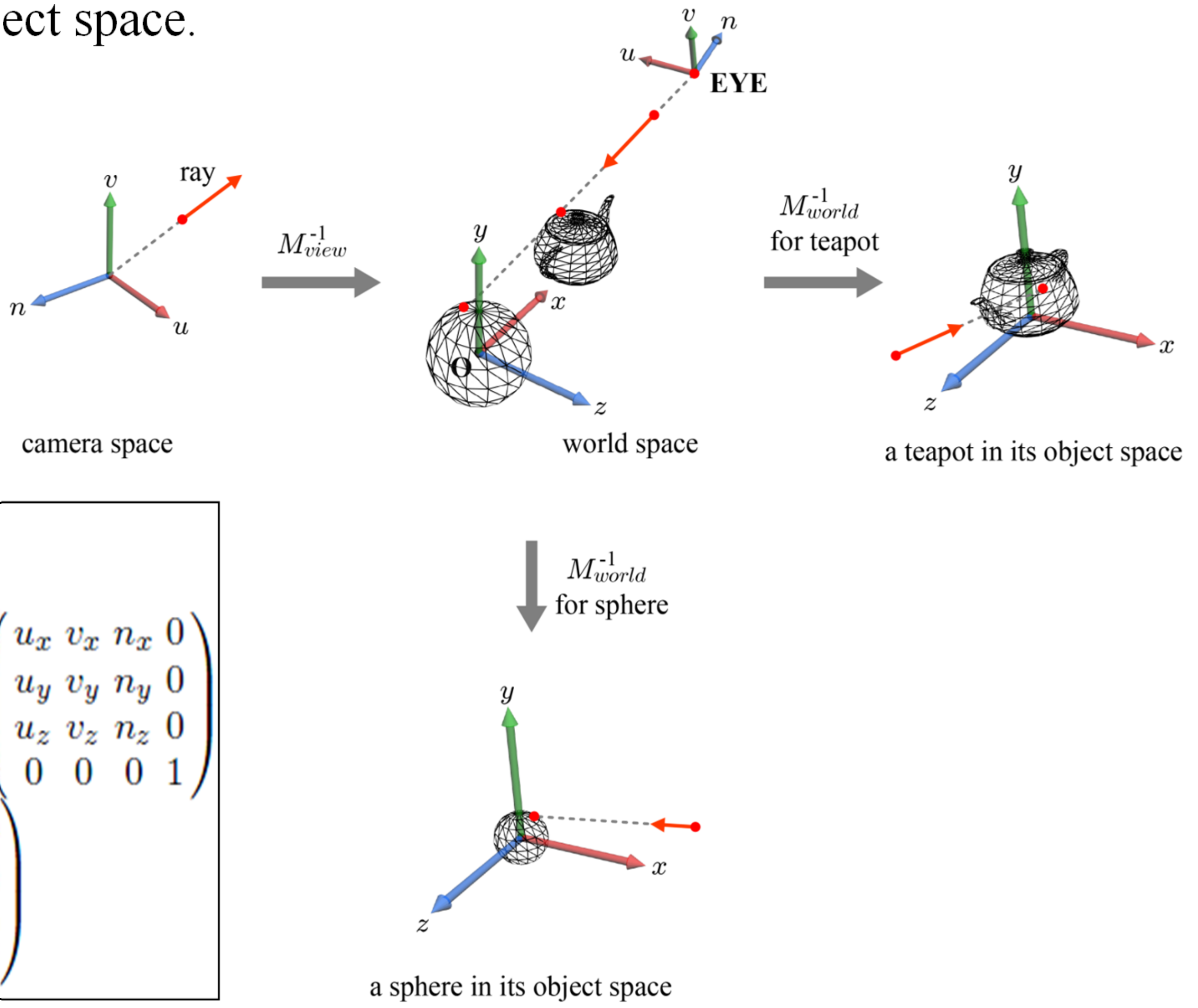
$$\begin{pmatrix} x_c\\ y_c\\ -n\\ 1\end{pmatrix} = \begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1)\\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1)\\ -n\\ 1\end{pmatrix}$$

$$\begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1)\\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1)\\ -n\\ 1\end{pmatrix} - \begin{pmatrix} 0\\ 0\\ 0\\ 1\end{pmatrix} = \begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1)\\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1)\\ -n\\ 0\end{pmatrix}$$

direction vector of the camera-space ray ➡ $\begin{pmatrix} \frac{1}{m_{11}}(\frac{2x_s}{W}-1)\\ \frac{1}{m_{22}}(\frac{2y_s}{h}-1)\\ -1\\ 0\end{pmatrix}$

From the Camera Space to the Object Space

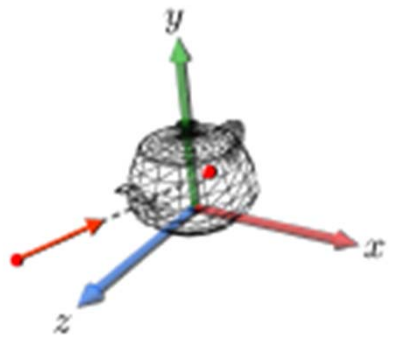
- It is straightforward to transform the camera-space ray into the world space and then to the object space.

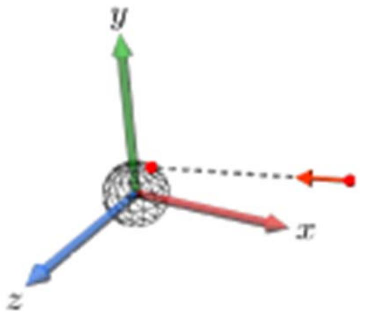


$$
\begin{aligned}
M^{-1}_{view} &= (RT)^{-1} \\
&= T^{-1}R^{-1} \\
&= \begin{pmatrix} 1 & 0 & 0 & \mathbf{EYE}_x \\ 0 & 1 & 0 & \mathbf{EYE}_y \\ 0 & 0 & 1 & \mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}
\begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} u_x & v_x & n_x & \mathbf{EYE}_x \\ u_y & v_y & n_y & \mathbf{EYE}_y \\ u_z & v_z & n_z & \mathbf{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
$$

Ray Intersection

- In order to determine if a ray hits an object represented in a polygon mesh, we have to perform a ray-triangle intersection test for every triangle of the object. If there exists at least a triangle intersecting the ray, the object is judged to be hit by the ray.

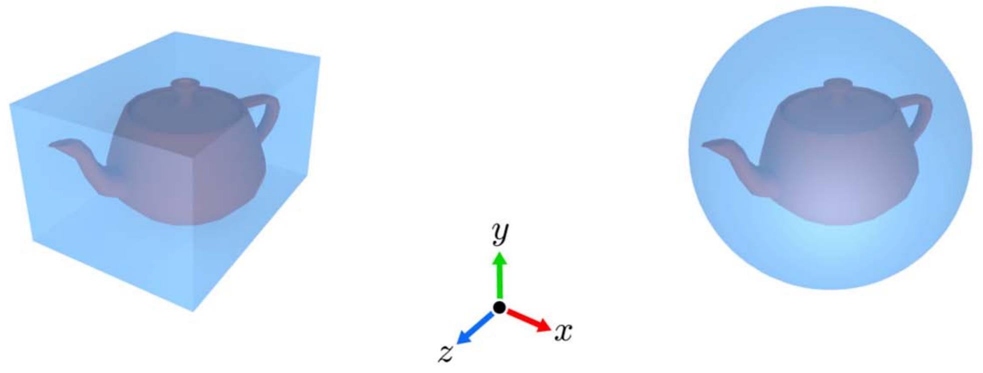




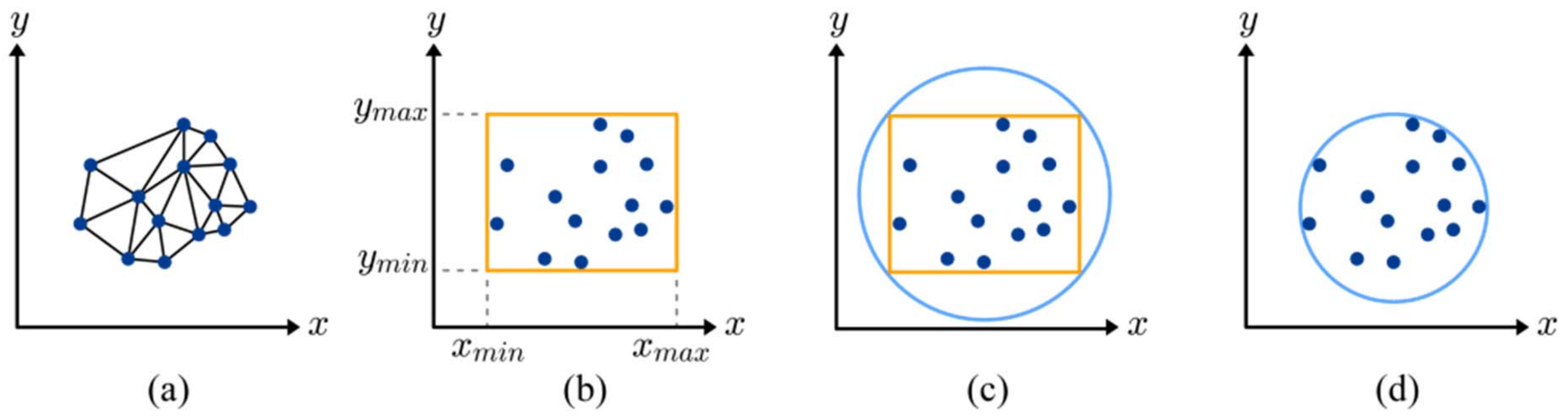
- Unfortunately, processing every triangle would be costly. A faster but less accurate method is to approximate a polygon mesh with a bounding volume (BV).

Bounding Volumes

- Bounding volumes

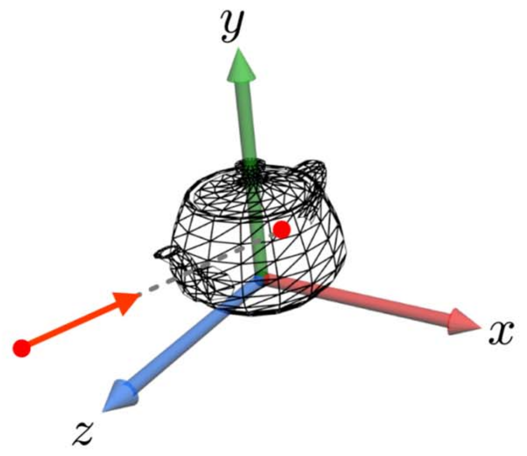


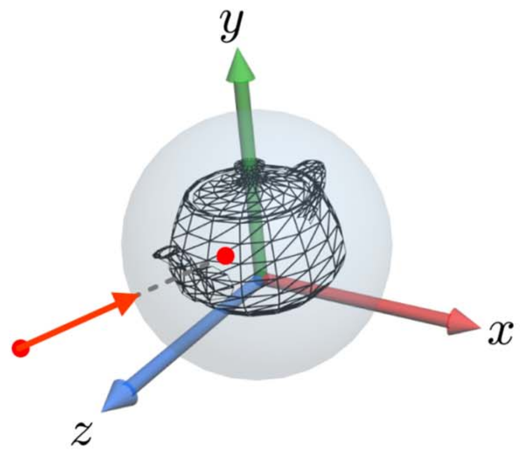
- Bounding volume creation



Ray Intersection (cont'd)

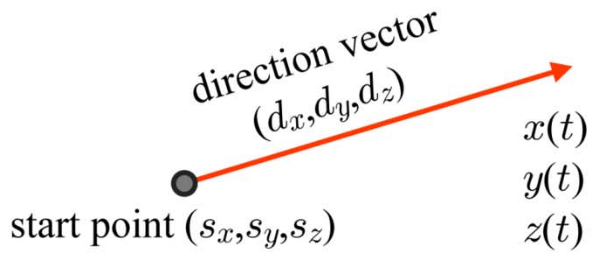
- Intersection tests with bounding volumes vs. polygon meshes.





Ray Intersection (cont'd)

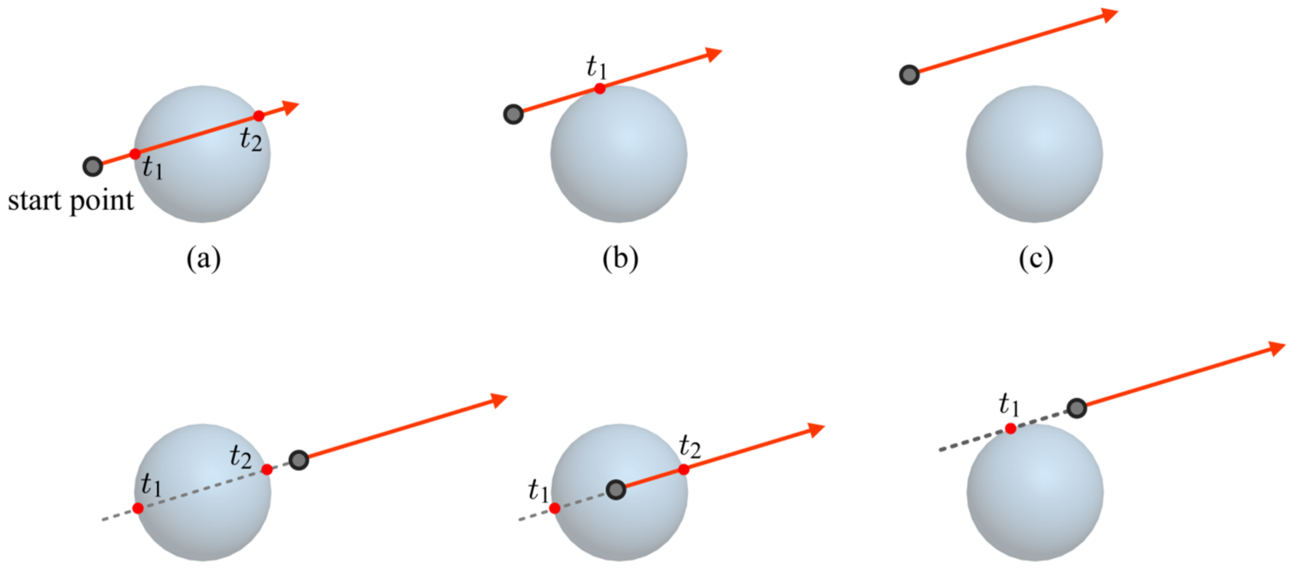
- For the ray-sphere intersection test, let us represent the ray in a parametric representation.



$$x(t) = s_x + td_x$$
$$y(t) = s_y + td_y$$
$$z(t) = s_z + td_z$$

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2 \Rightarrow at^2 + bt + c = 0 \Rightarrow t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Collect only positive *t*s. (Given two positive *t*s for a sphere, choose the smaller.)

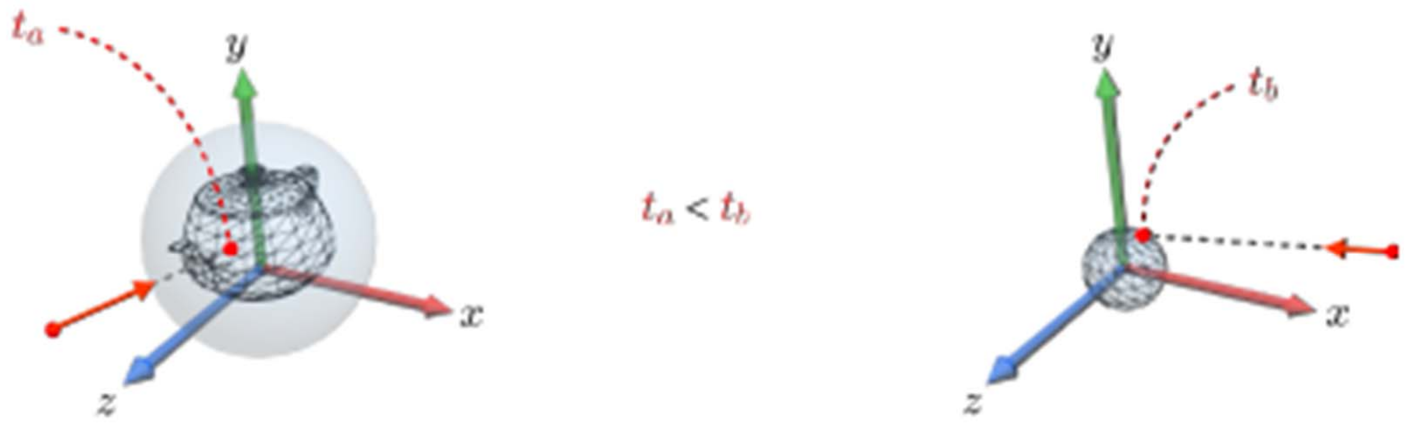


(a) (b) (c)

(d) (e) (f)

Ray Intersection (cont’d)

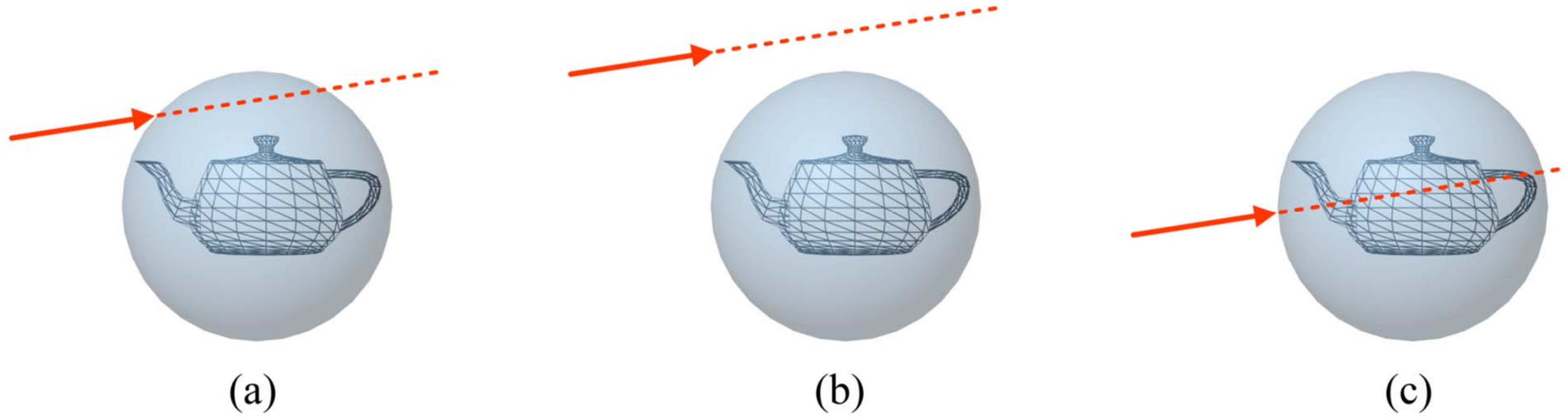
- The bounding sphere hit first by the ray is the one with the smallest *t*.



a teapot in its object space a sphere in its object space

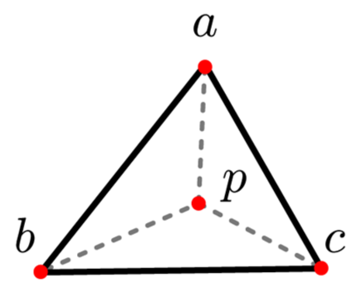
Ray Intersection (cont'd)

- Ray-sphere intersection test is often performed at the preprocessing step, and discards the polygon mesh that is guaranteed not to intersect the ray.



Ray Intersection (cont'd)

- A point in a triangle



$$p = ua + vb + wc$$

$$\boxed{u = \frac{area(p,b,c)}{area(a,b,c)}, v = \frac{area(p,c,a)}{area(a,b,c)}, w = \frac{area(p,a,b)}{area(a,b,c)}}$$

- The weights (u,v,w) are called the *barycentric coordinates* of *p*.
- The triangle is divided into three sub-triangles, and the weight given for a control point is proportional to the area of the sub-triangle "on the opposite side."
- Obviously, $u+v+w=1$, and therefore w can be replaced by $(1-u-v)$, i.e., $p=ua+vb+(1-u-v)c$.

Ray Intersection (cont'd)

- Computing the intersection between a ray, *s* + *td*, and a triangle, <*a*, *b*, *c*> is equivalent to solving the following equation:

$$s + td = ua + vb + (1 - u - v)c$$

$$td + u(c - a) + v(c - b) = c - s$$

$$td + uA + vB = S$$

$$\begin{aligned} td_x + uA_x + vB_x &= S_x \\ td_y + uA_y + vB_y &= S_y \\ td_z + uA_z + vB_z &= S_z \end{aligned}$$

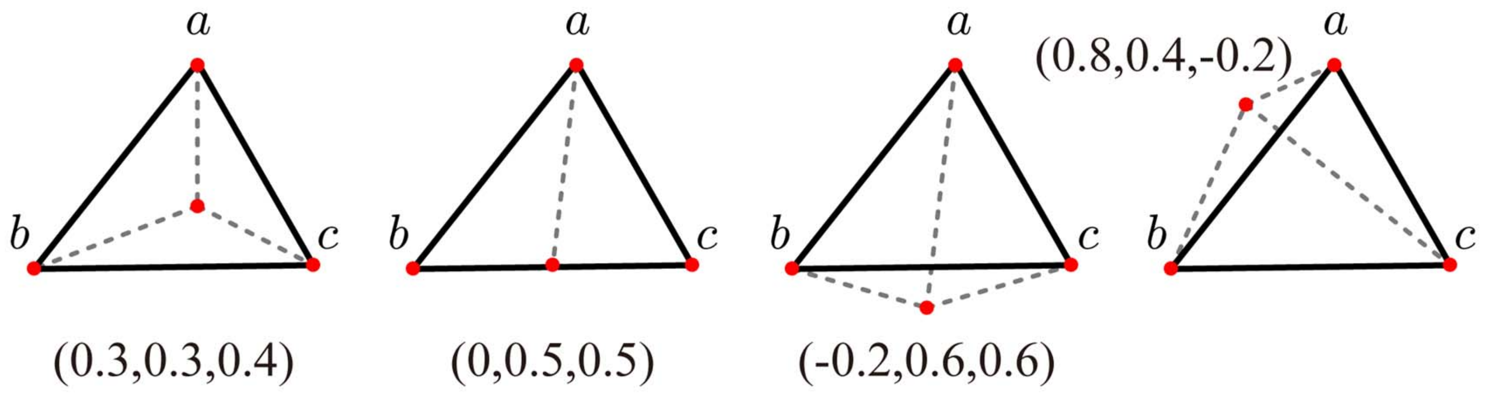
- The solution to this system is obtained using Cramer's rule:

$$t = \frac{\begin{vmatrix} S_x & A_x & B_x \\ S_y & A_y & B_y \\ S_z & A_z & B_z \end{vmatrix}}{\begin{vmatrix} d_x & A_x & B_x \\ d_y & A_y & B_y \\ d_z & A_z & B_z \end{vmatrix}}, u = \frac{\begin{vmatrix} d_x & S_x & B_x \\ d_y & S_y & B_y \\ d_z & S_z & B_z \end{vmatrix}}{\begin{vmatrix} d_x & A_x & B_x \\ d_y & A_y & B_y \\ d_z & A_z & B_z \end{vmatrix}}, v = \frac{\begin{vmatrix} d_x & A_x & S_x \\ d_y & A_y & S_y \\ d_z & A_z & S_z \end{vmatrix}}{\begin{vmatrix} d_x & A_x & B_x \\ d_y & A_y & B_y \\ d_z & A_z & B_z \end{vmatrix}}$$

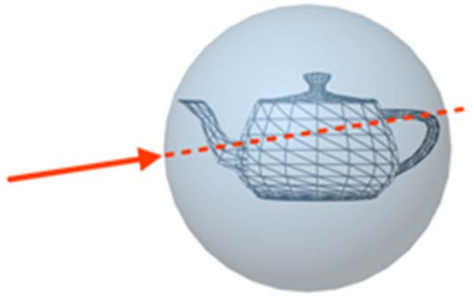
- The intersection point is computed by inserting *t* into the ray equation, *s* + *td*, or by inserting *u* and *v* into *ua*+*vb*+(1-*u*-*v*)*c*.

Ray Intersection (cont'd)

- In order for the intersection point to be confined to the triangle, the following
- condition should be satisfied: $u \geq 0$, $v \geq 0$, and $u+v \leq 1$.

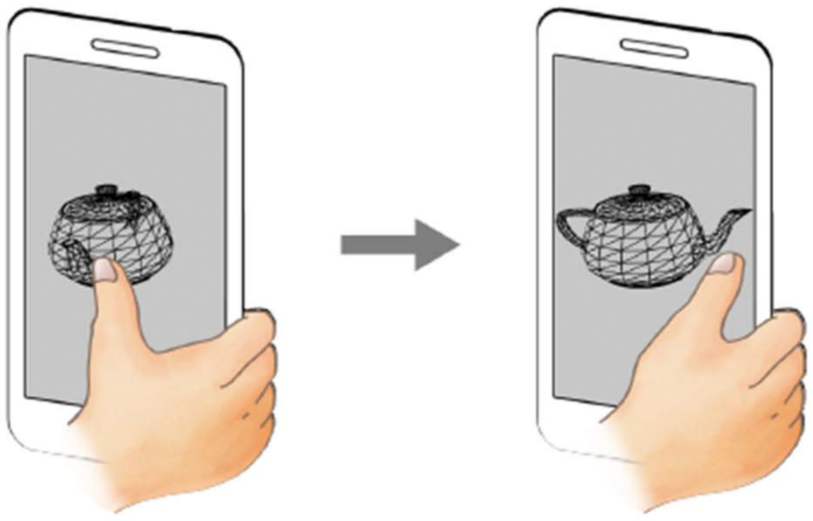


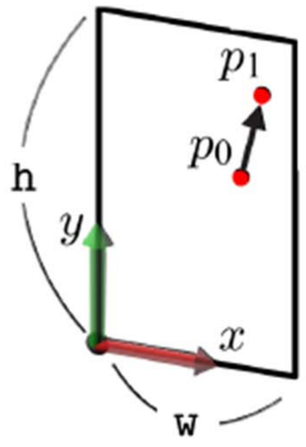
- When every triangle of a mesh is tested for intersection with the ray, multiple intersections can be found. Then, we choose the point with the smallest positive *t*.



Object Rotating

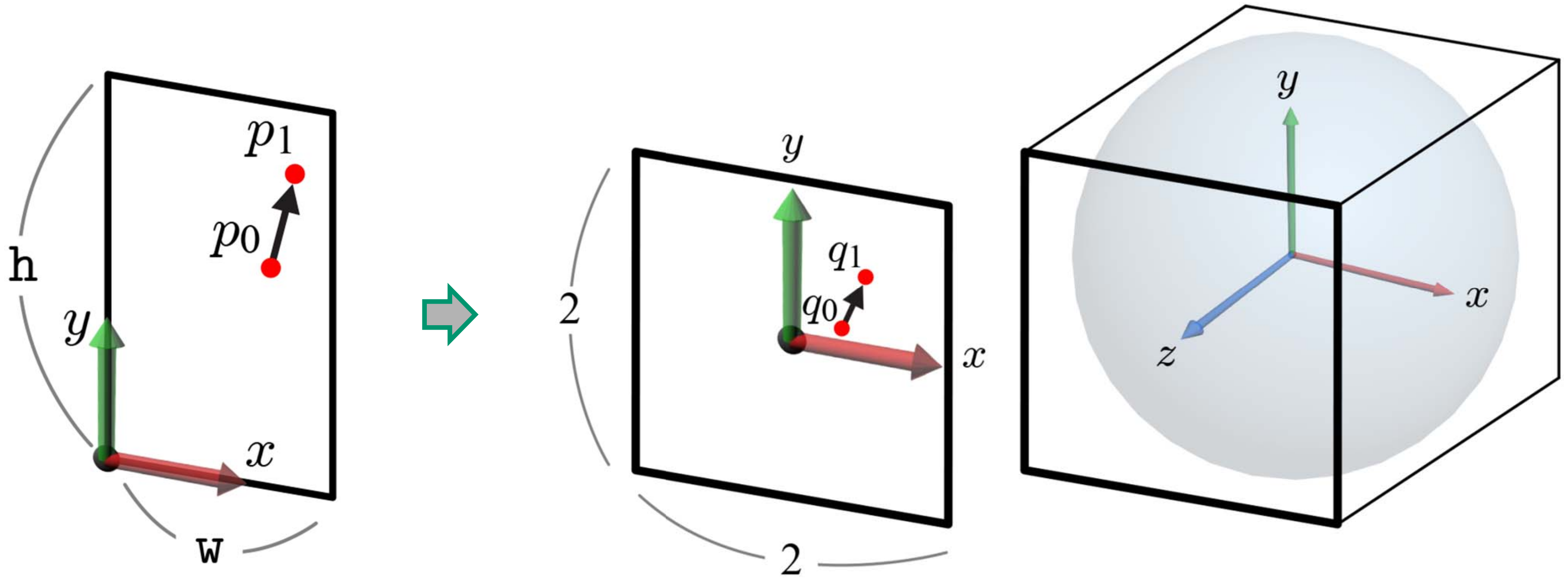
- Especially useful for touchscreen.
- The input to the object rotating function is a sequence of 2D screen coordinates for the tracked finger's positions. It is not straightforward to convert such a 2D motion of a finger to 3D rotation.





Arcball

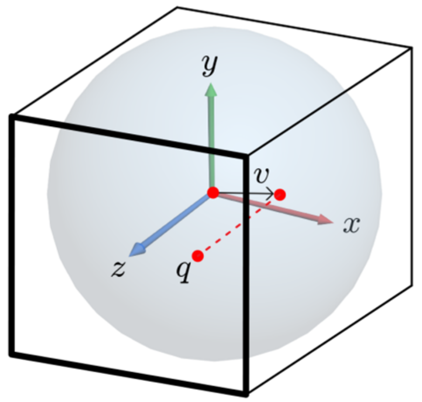
- The arcball is a virtual ball that is located behind the screen and encloses the object to rotate. The sliding finger rotates the arcball and the same rotation applies to the object.
- Use a unit sphere (of radius one) for the arcball. For compatibility with the arcball, the 2D screen and the finger positions need to be normalized.

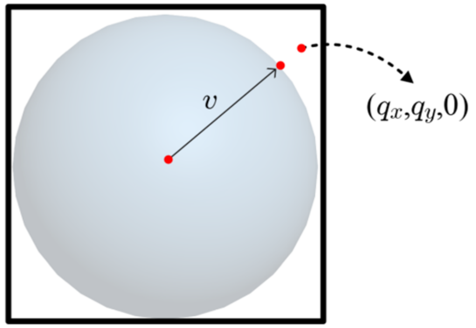


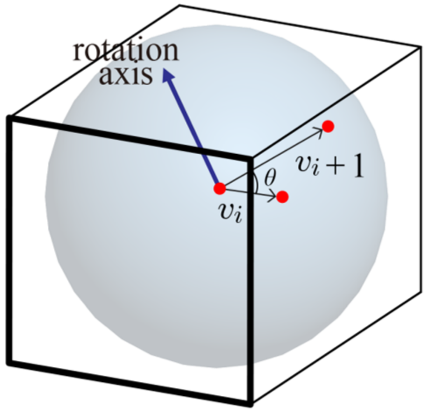
$$\left(2\frac{x}{\mathtt{w}}-1,\ 2\frac{y}{\mathtt{h}}-1\right)$$

Arcball (cont'd)

- The normalized position *q* is orthographically projected onto the surface of the arcball. Let *v* denote the vector connecting the arcball's center and the projected point. Note that $v_x=q_x$, $v_y=q_y$, and $v_z=\sqrt{1-v_x^2-v_y^2}$.
- If the projection is out of the arcball, *v*=normalize(q_x,q_y,0).





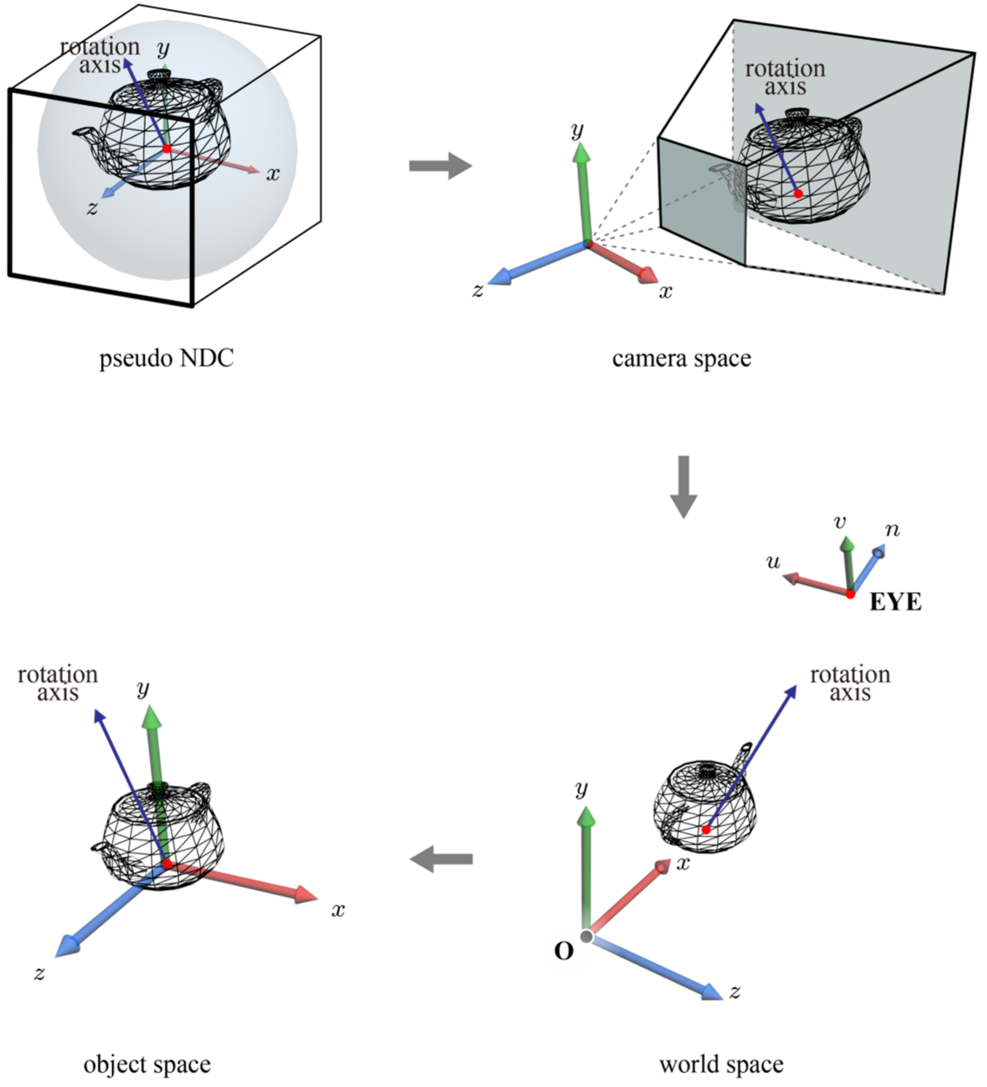


- Given v_i and v_{i+1} we can compute the rotation axis and angle.
  - The rotation axis is obtained by taking the cross product of v_i and v_{i+1}.
  - The rotation angle is obtained from the dot product of v_i and v_{i+1}.

$$v_i \cdot v_{i+1} = \|v_i\|\|v_{i+1}\| cos\theta$$

Object-space Rotation Axis

- Pseudo NDC!
- Our rotation should apply to the object in its object space so that the rotated object enters the GPU rendering pipeline. The rotation axis needs to be transformed to the object space so as to make up a *quaternion* together with the rotation angle.
- Take the rotation axis in the pseudo NDC as that in the camera space.
- We know how to transform from the camera space to the object space.


rotation axis
y
x
z
pseudo NDC
rotation axis
y
x
z
camera space
v
n
u
EYE
rotation axis
y
x
z
object space
rotation axis
y
x
O
z
world space