컴퓨터그래픽스
Computer Graphics

Screen-space Object Manipulation

부산대학교 정보·의생명공학대학
정보컴퓨터공학부

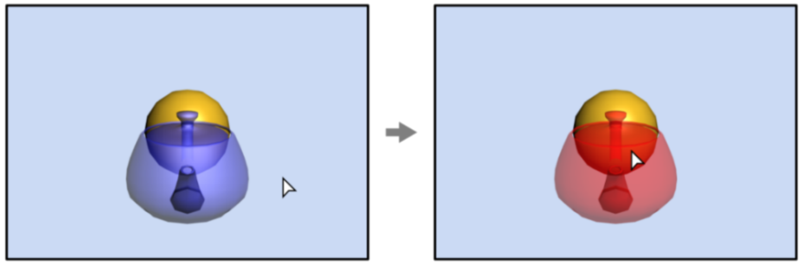
This class...

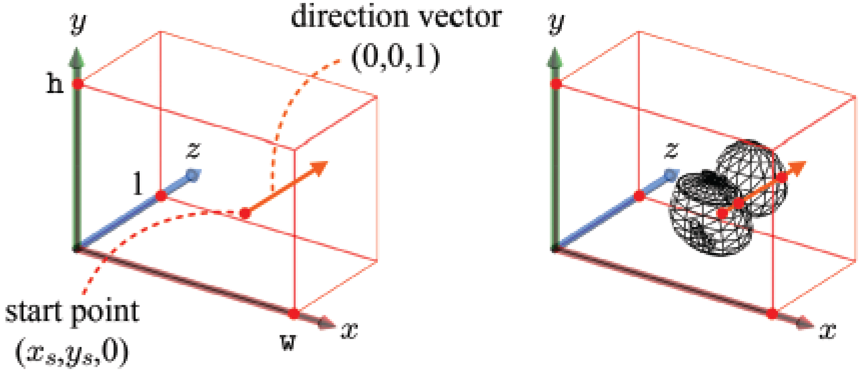
- Object picking
  - Ray in Screen space → Camera space → Object space
  - Intersection between Ray and Bounding volume/triangle

- Rotating an Object

Object Picking

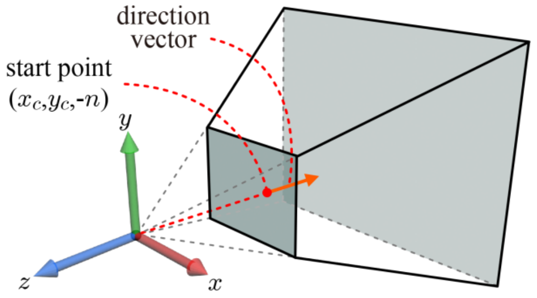
- On touchscreen, an object can be picked by tapping it with a finger. In PC screen, mouse clicking is popularly used for the same purpose. Both simply return the 2D pixel coordinates (x_s,y_s).
- Given (x_s,y_s), we can consider a *ray* described by the start point (x_s,y_s,0) and the direction vector (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 good to compute the intersections in the *screen space* because the screen-space information available to us is not about the objects.
- Our solution is to transform the screen-space ray back to the *object spaces* and do the intersection tests in the object spaces.

Camera-space Ray

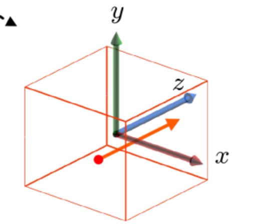


camera space

projection transform

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

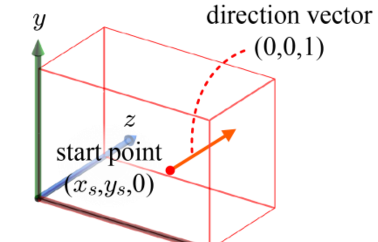
$$\boxed{\begin{array}{l} \frac{cot(\frac{fovy}{2})}{aspect} \Rightarrow m_{11} \\ cot(\frac{fovy}{2}) \Rightarrow m_{22} \end{array}}$$



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

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

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

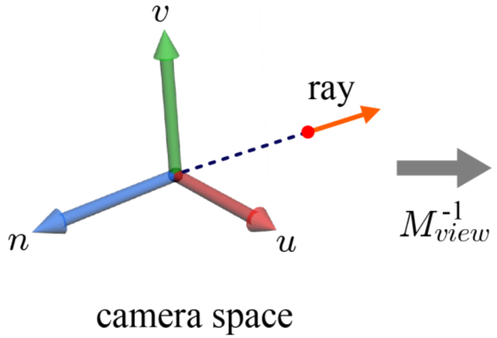
$$\begin{pmatrix} x_c \\ y_c \\ -n \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1) \\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1) \\ -n \\ 1 \end{pmatrix}$$

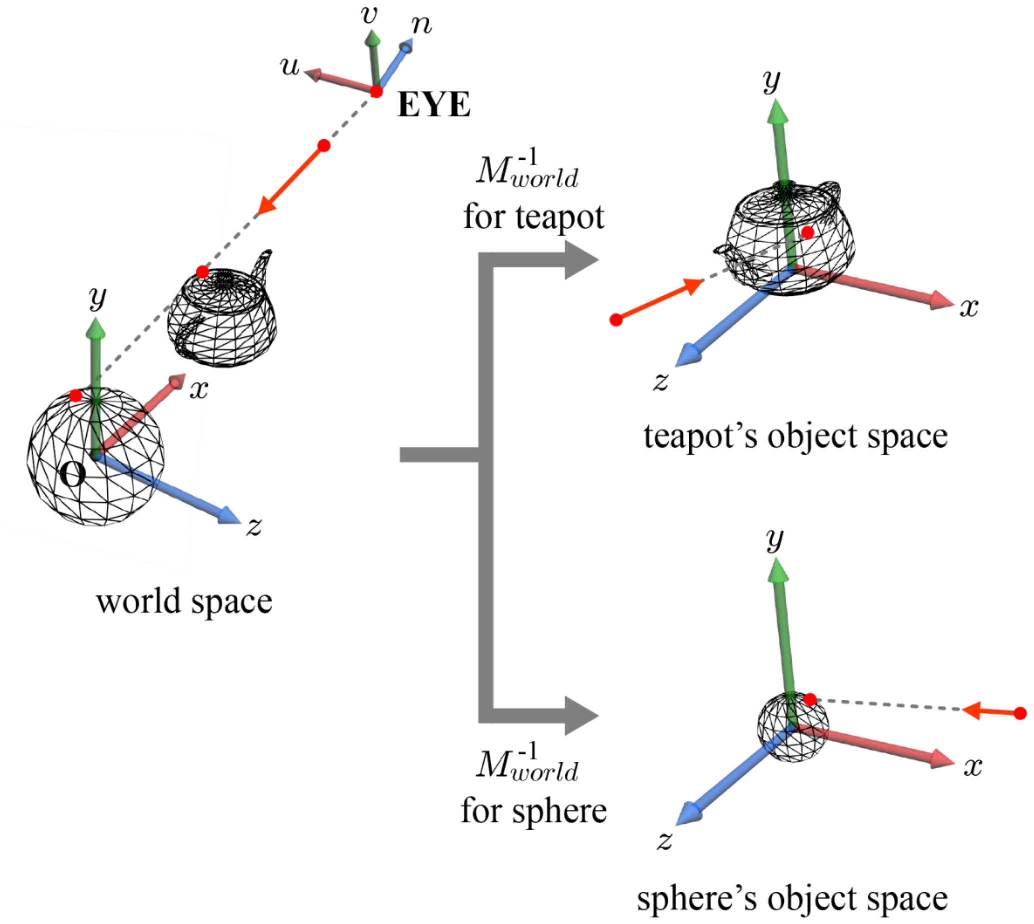
$$\begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1) \\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1) \\ -n \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{n}{m_{11}}(\frac{2x_s}{W}-1) \\ \frac{n}{m_{22}}(\frac{2y_s}{h}-1) \\ -n \\ 0 \end{pmatrix}$$

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

Object-space Ray

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



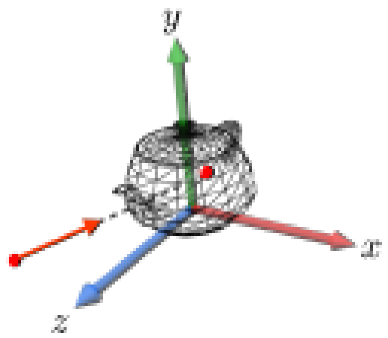


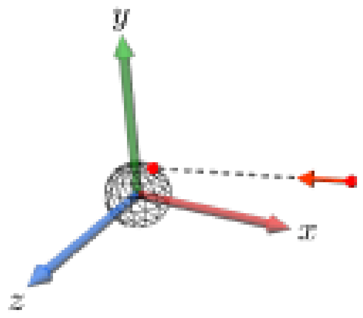
$$
\begin{aligned}
M_{view}^{-1} &= (RT)^{-1} \\
&= T^{-1}R^{-1} \\
&= \begin{pmatrix} 1 & 0 & 0 & \mathrm{EYE}_x \\ 0 & 1 & 0 & \mathrm{EYE}_y \\ 0 & 0 & 1 & \mathrm{EYE}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 &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

The teapot and sphere have their own world transforms. Their inverses are applied to the world-space ray.

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.

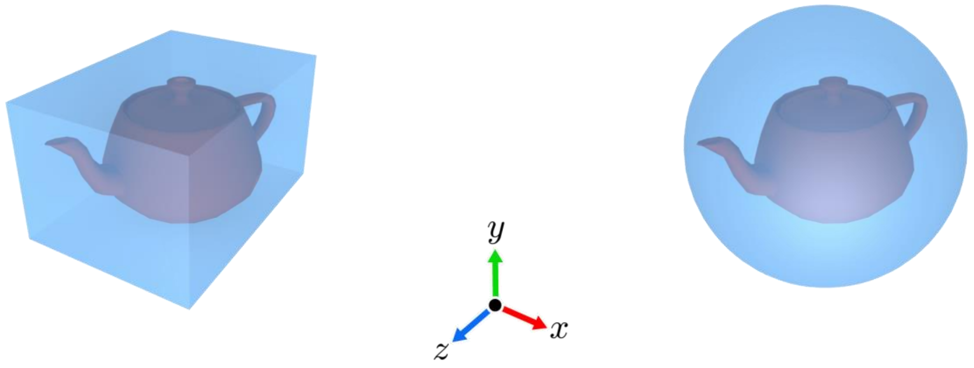




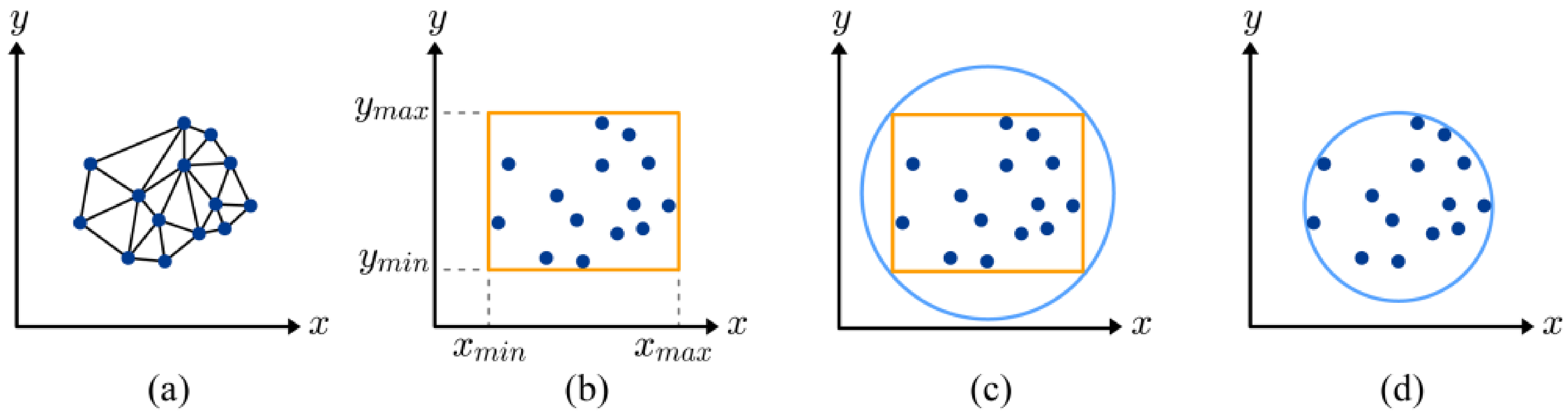
- Unfortunately, processing every triangle would be costly. A faster but less accurate method is to approximate a polygon mesh with a *bounding volume* (BV).

Bounding Volumes

- Bounding volumes: axis-aligned bounding box and bounding sphere

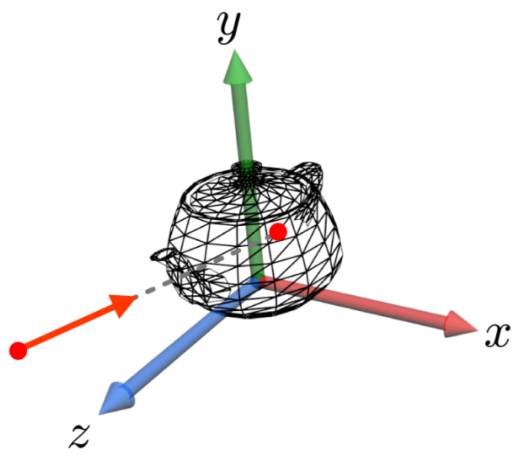


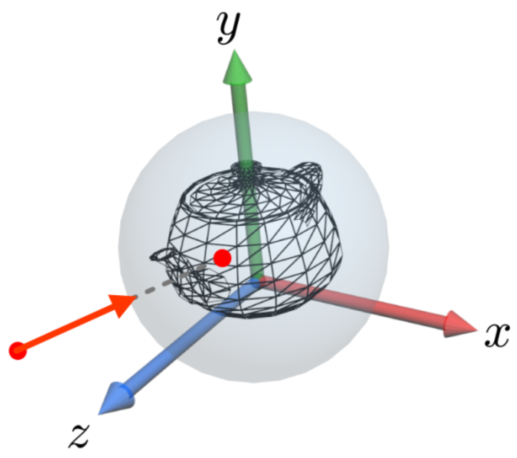
- Bounding volume creation



Ray-BV Intersection

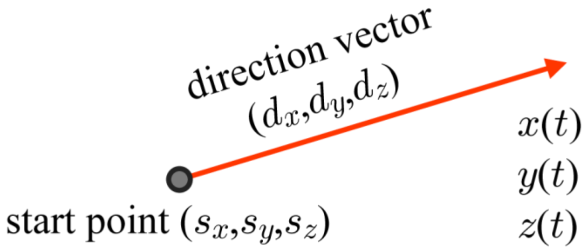
- Intersection tests with polygon meshes vs. bounding volumes.





Ray-BV Intersection (cont'd)

- For the ray-sphere intersection test, let us represent the ray in three parametric equations.

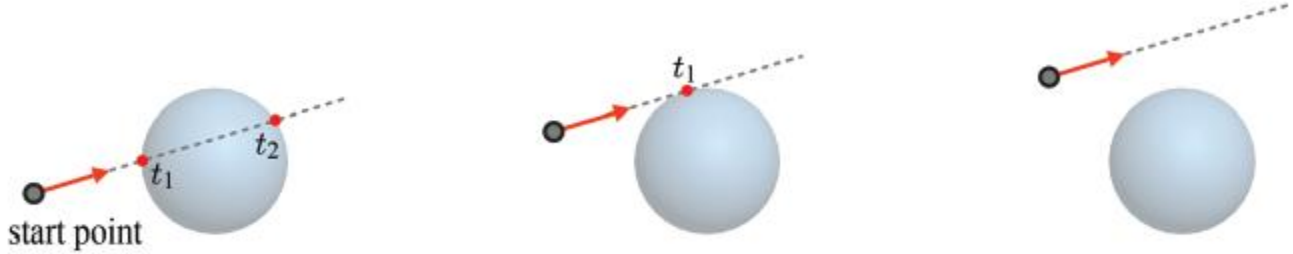


$$x(t) = s_x + td_x$$
$$y(t) = s_y + td_y$$
$$z(t) = s_z + td_z$$

- Insert *x*(*t*), *y*(*t*), and *z*(*t*) into the bounding sphere equation.

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

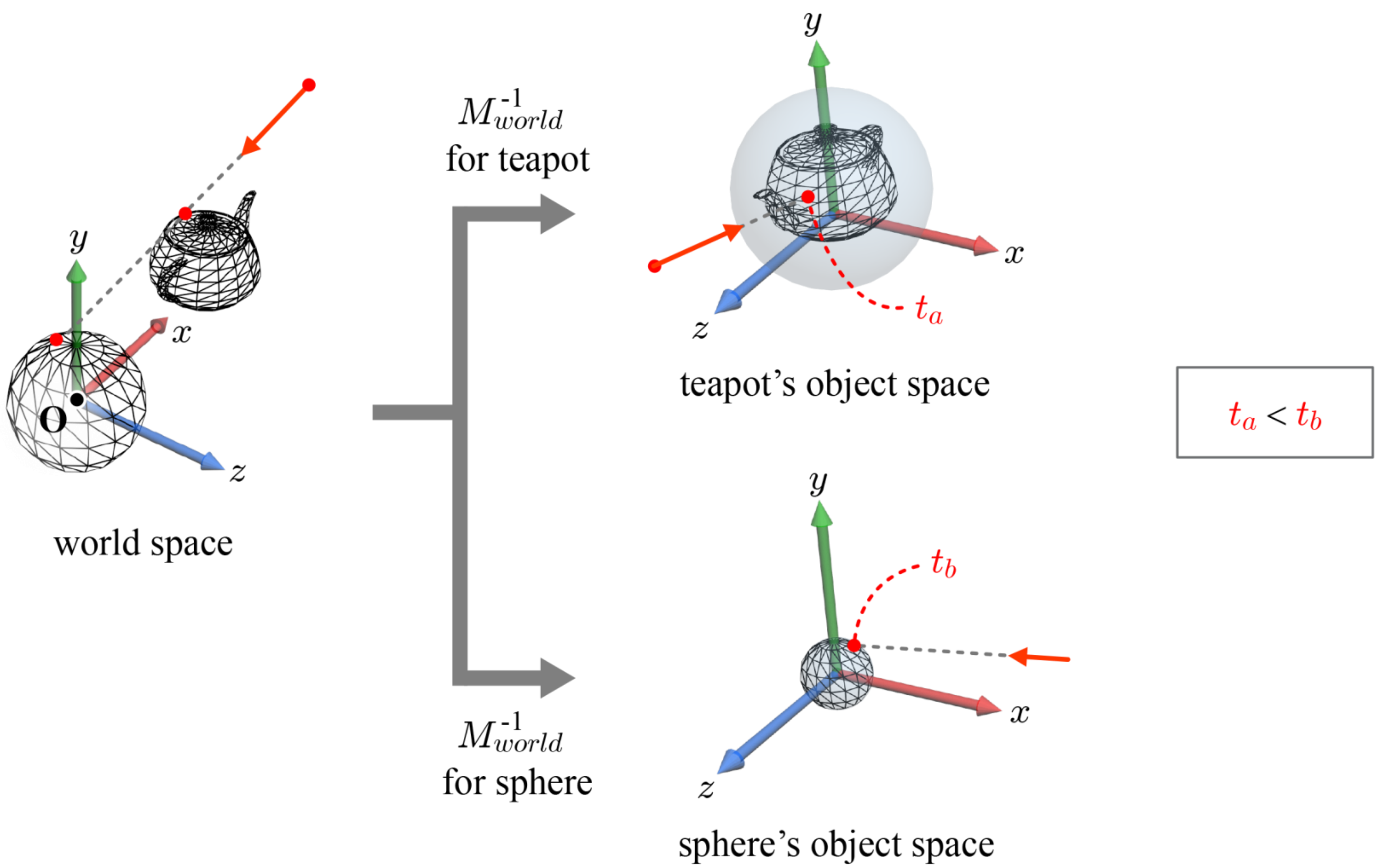
- Three cases determined by the discriminant.



- Given two distinct roots, choose the smaller.

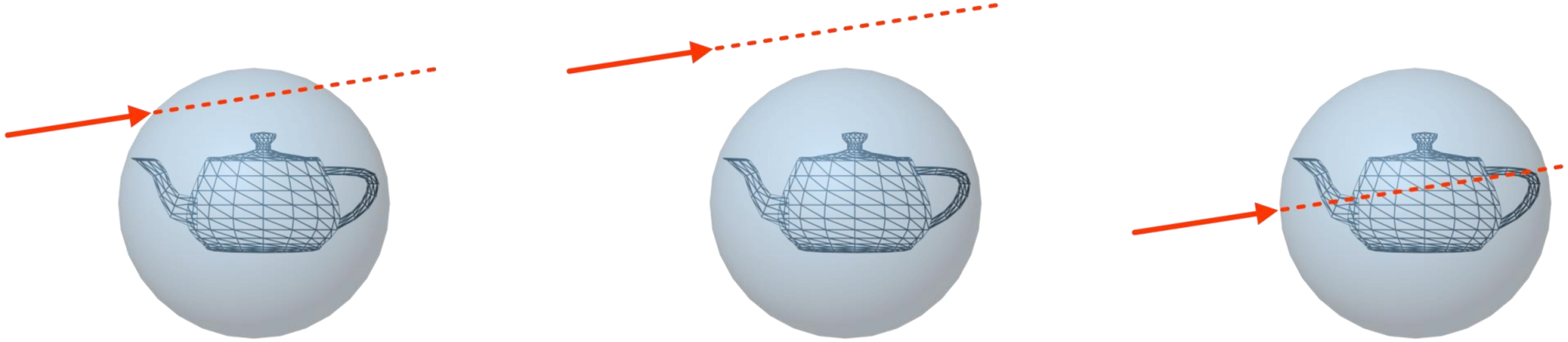
Ray-BV Intersection (cont'd)

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



Ray-BV Intersection (cont'd)

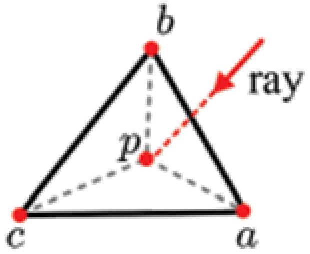
- Ray-sphere intersection test is often performed at the preprocessing step, and discards the polygon mesh that is guaranteed not to intersect the ray.



(a) (b) (c)

Ray-Triangle Intersection

- When a triangle <*a*,*b*,*c*> is hit by a ray at *p*, the triangle is divided by *p* into three sub-triangles.
- Let *u* denote the ratio of the area of <*p*,*b*,*c*> to that of <*a*,*b*,*c*>. It roughly describes how close *p* is to *a*. The closer, the larger. It can be taken as the *weight* of *a* on *p*.



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

- When *v* and *w* are similarly defined, *p* is defined as a *weighted sum* of the vertices: $p = ua + vb + wc$.
- The weights (u,v,w) are called the *barycentric coordinates* of *p*.
- 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-Triangle Intersection (cont'd)

- Compute the intersection between a ray, *s* + *td*, and a triangle, <*a*, *b*, *c*>.

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

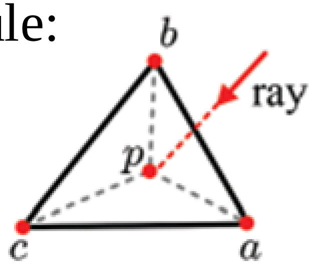
$$td + u(c - a) + v(c - b) = c - s$$

$$td + uA + vB = S$$

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

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

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



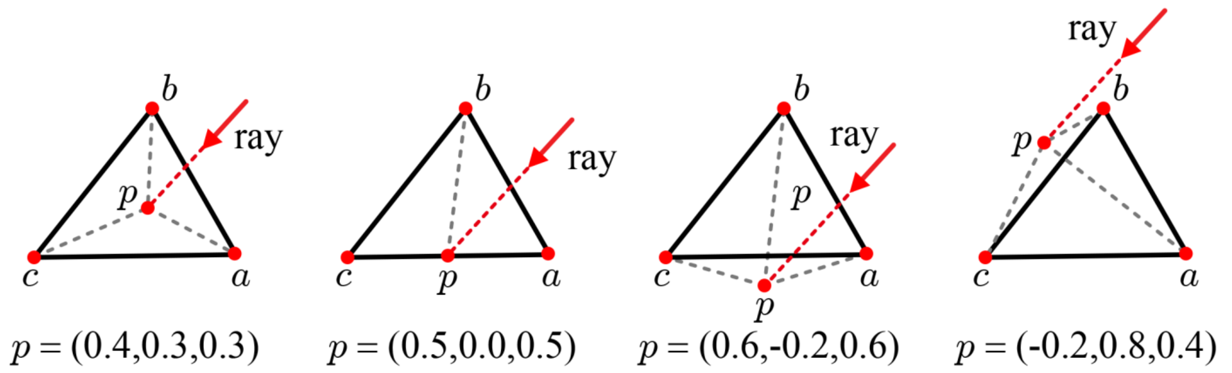
$$p = s + td$$

$$p = ua + vb + wc$$

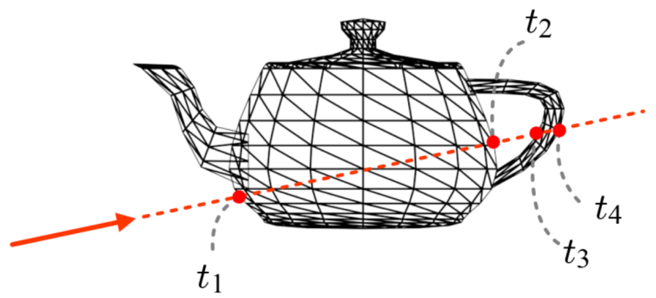
- The intersection point can be obtained by inserting *t* into *s* + *td* or by inserting *u* and *v* into *ua*+*vb*+(1-*u*-*v*)*c*. However, note that we do not need the point itself.
- More importantly, is *p* guaranteed to be inside the triangle?

Ray-Triangle 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 $u+v \leq 1$.

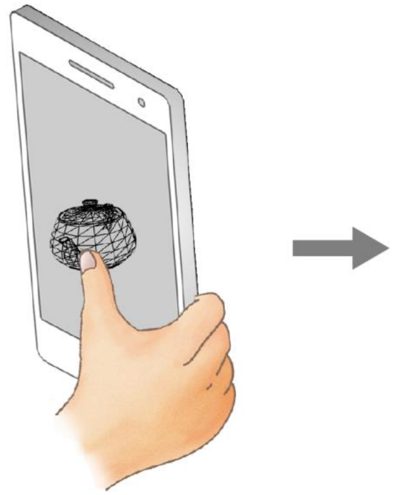


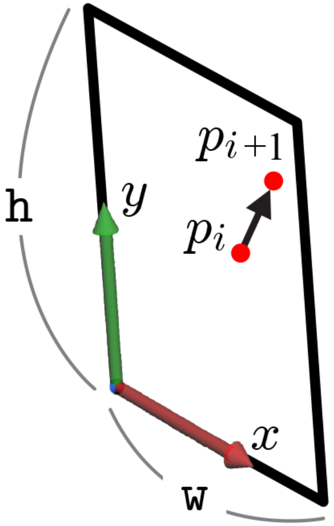
- When every triangle of a mesh is tested for intersection with the ray, multiple intersections can be found. Then, choose the point with the smallest positive *t*.



Rotating an Object

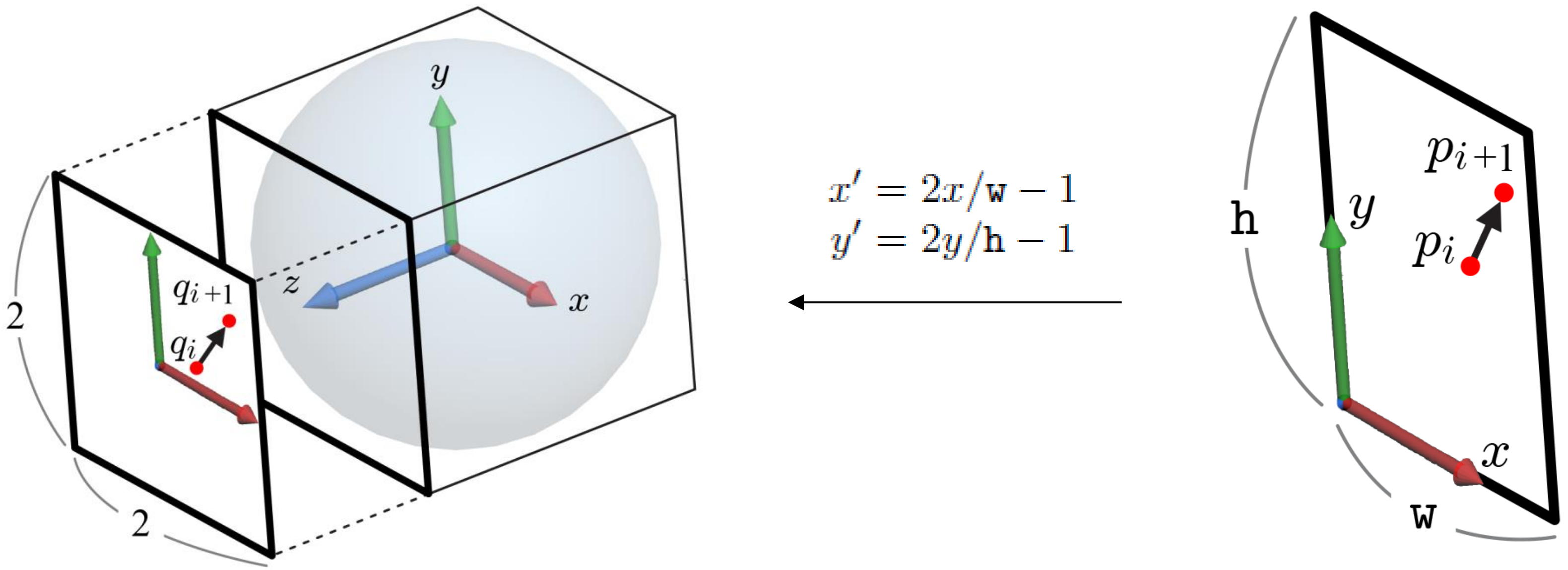
- Rotating an object is used as frequently as picking an object.
- On the 2D screen, the sliding finger's positions, $\{p_1, p_2, \ldots, p_n\}$, are tracked. For each pair of successive finger positions, p_i and p_{i+1}, a 3D rotation is computed and applied to the object.





Arcball

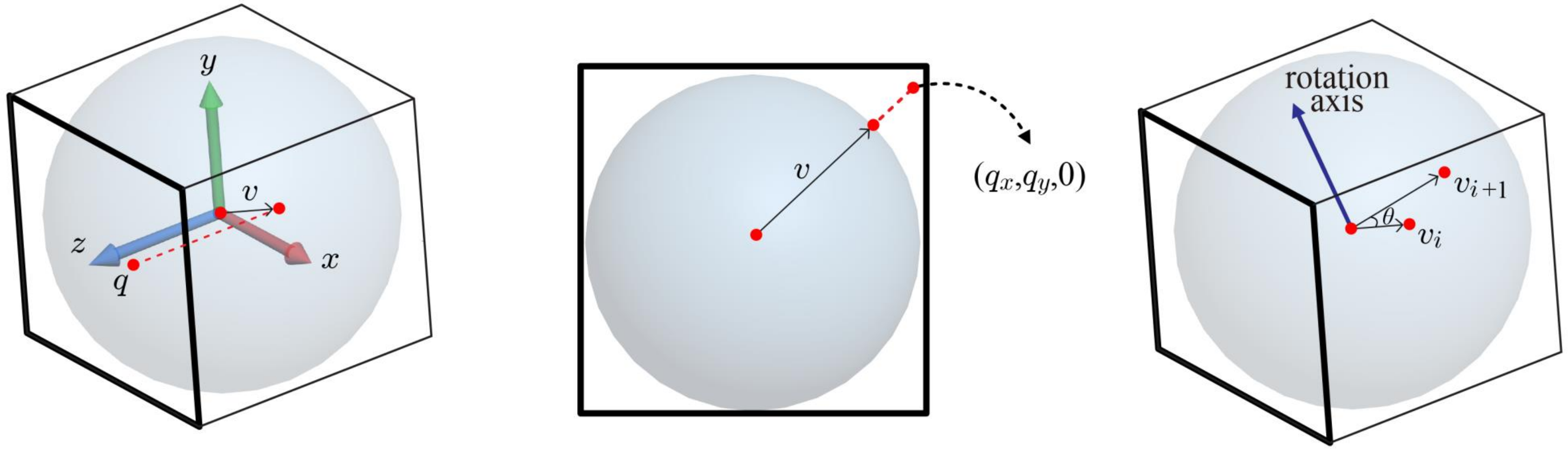
- The arcball is a virtual ball located behind the screen and encloses the object to rotate. The sliding finger rotates the arcball and the same rotation applies to the object.
- For efficient implementation, the 2D screen with dimensions `w`×`h` is normalized to the 2×2 square, and the 2D coordinates, (x,y), of the finger's position are accordingly normalized, i.e., p_i and p_{i+1} are converted to q_i and q_{i+1}, respectively, in the square.



- Note that the arcball is a unit sphere, the radius of which is one.

Arcball (cont'd)

- Then, q is orthographically projected along $-z$ 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, i.e., if $q_x^2 + q_y^2 > 1$, v = normalize(q_x,q_y,0).

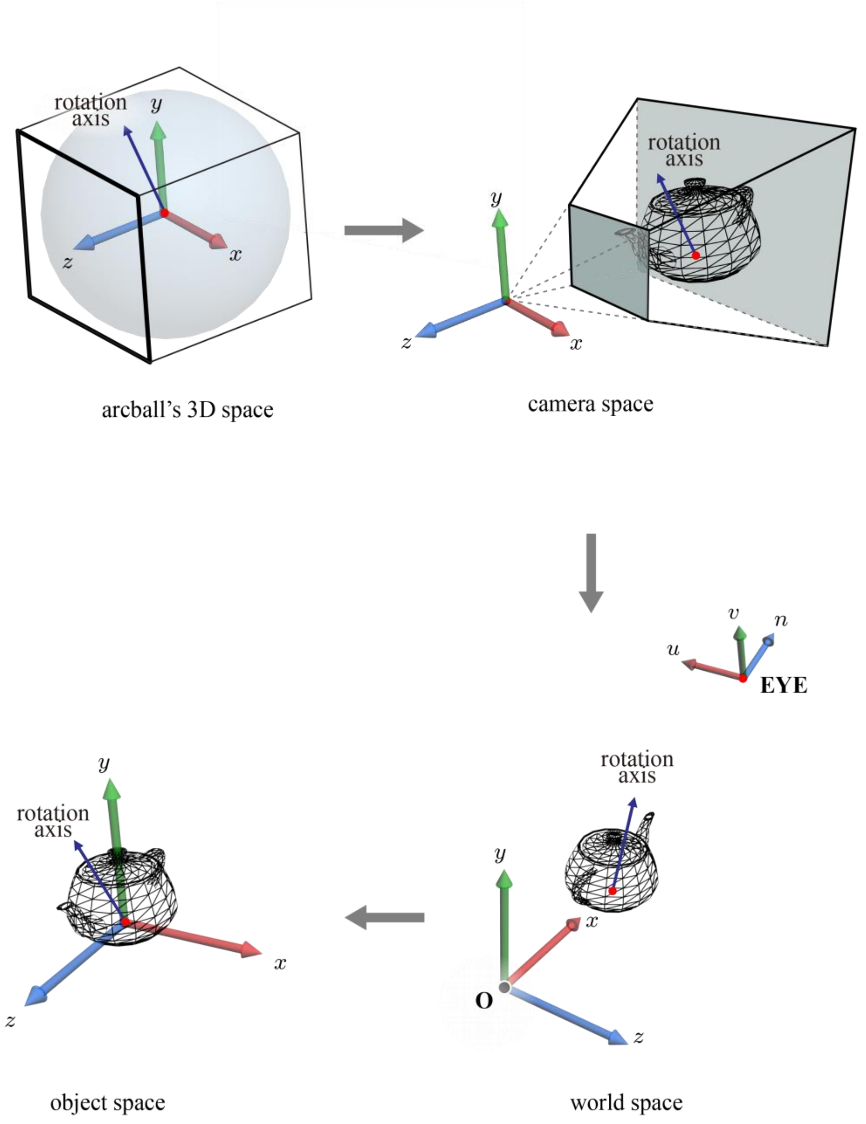


- Given v_i and v_{i+1}, we can compute the rotation axis and angle.
  - The rotation axis is obtained by taking the cross product of v_i and v_{i+1}.
  - The rotation angle is obtained from the dot product of v_i and v_{i+1}.

$$v_i \cdot v_{i+1} = \|v_i\| \|v_{i+1}\| cos\theta = cos\theta \longrightarrow \theta = arccos(v_i \cdot v_{i+1})$$

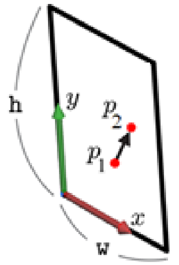
Object-space Rotation Axis

- The rotation axis should be redefined in the object space so that the rotation is made prior to the original world transform.
- We computed the rotation axis in the arcball's space. It is a temporarily devised coordinate system, which we have not encountered in the rendering pipeline.
- Our strategy is then to take the rotation axis *as is* into the camera space of the rendering pipeline.
- The rotation axis, as a vector, can be thought of as passing through the object in the camera space, and such a configuration is preserved in the object space.

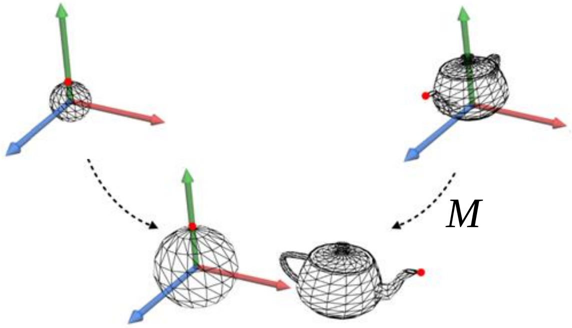


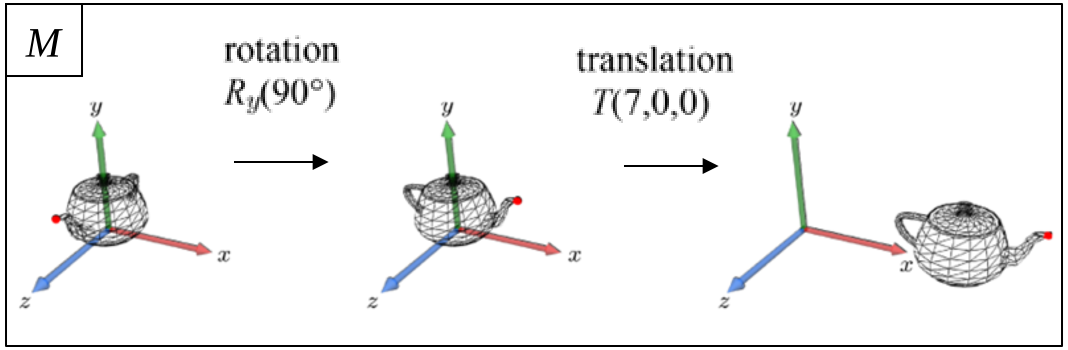
Rotating an Object - Implementation

- We have defined the rotation axis and angle. Then, the rotation matrix can be obtained:
  - Invoke `glm::rotate(angle, axis)`.
    - Object3D.rotateOnAxis(axis, angle) in Three.js
  - Define a quaternion and convert it into a matrix.



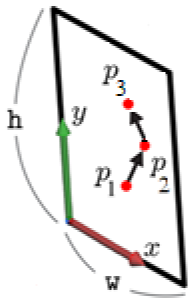
- Let M denote the current world matrix. Initially, the finger is at p_1 on the screen. When the finger moves to p_2, the rotation axis is computed using p_1 and p_2. It is transformed from the world space back to the object space using the inverse of M.
- Let R_1 denote the rotation matrix determined by the rotation axis. Then, MR_1 will be provided for the vertex shader as the world matrix, making the object slightly rotated in the screen.





Rotating an Object – Implementation (cont'd)

- Note that MR_1 presented in the previous page is the world matrix to define the screen when the finger is at p_2.



- When the finger moves from p_2 to p_3, the rotation matrix is computed using the inverse of MR_1. Let R_2 denote the rotation matrix. Then, the world matrix is updated to MR_1R_2 and is passed to the vertex shader.
- This process is repeated while the finger remains on the screen.

Thank You!

Slides are modified from

Introduction to Computer Graphics with OpenGL ES (J. Han)