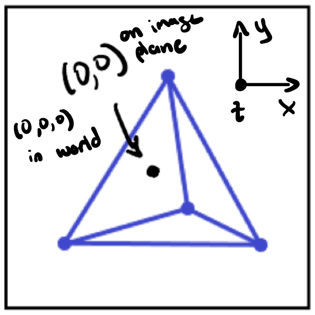
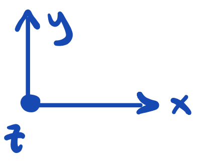


- Want to represent 3D points in 2D plane
- each 3D point defined on an XYZ coordinate system: $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$
- convert to homogenous coordinates for translation: $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$
- We seek to map this 3D point to the user's 2D window: thus, we need to know the relationship between the user's place in the world and the points place in the world, and then map this point to the 2D image plane (eventually using color and shading to represent depth).

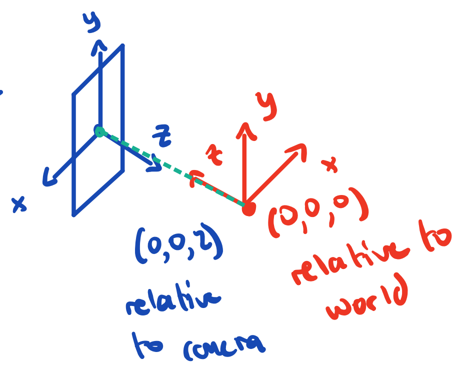
- Do this using coordinate frames:
  - world frame
  - camera frame

- Since the object is stationary, we will assign it the same coordinate frame as the world frame, namely origin at (0,0,0] and  coordinate frame.

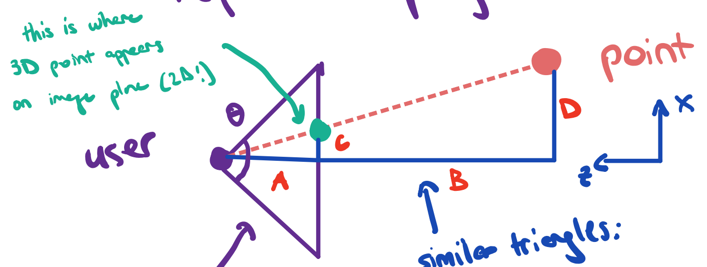
- We also have the camera coordinate frame, in some orientation. Since we are only rotating, we consider a frame that is translated an infinite distance away along the world z-axis.

- We seek to convert these world coordinate points into the camera frame's representation.

This is because we want to project them onto the camera's image plane, and in order for that it must be in the camera's coordinate frame.



represent projection like so:



some kind of FOV determined by $\theta \rightarrow \frac{1}{\tan(\frac{\theta}{2})}$

similar triangles:
have a mapping
since we know distance to image plane and position of point (since it's in our coordinate system)

$\frac{C}{A} = \frac{C+D}{B+A}$ → only unknown is C

set image plane to be 1 away.

Now we have $C = \frac{C+D}{B+A}$ → z distance in 3D
→ x distance in 3D.

Same for y!

BUG FIX: we do not want to rotate camera based on user input; Rather, rotate <u>vertices</u> of object to achieve desired shape orientation

Coding outline:

- using python and pygame : simple, less overhead

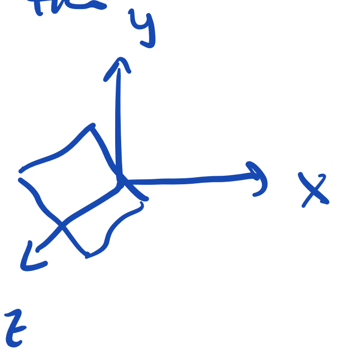
- given: vertices and faces
  - vertices ↙ relative to world frame
  - faces ↓ composed with vertices

- need object for points, camera, object
  - ↳ python class

- constantly refresh screen with updated point locations and lines

- use numpy for fast matrix operations

- to calculate shading for faces, we should look at the normal vector for each plane. Each face is composed of 3 points, so we can simply create 2 lines and then take the cross product of them. If this normal vector is parallel to



the z-axis, then taking the dot product of the two should produce the magnitude & multiply the two together. Likewise, if they are orthogonal it should produce a value of 0.

- Additionally, need to consider view of camera frame: find vector from z-axis to camera frame in order to take "triangle" culling into account.

- Also, ordering of vertices for faces should remain consistent, otherwise there will be issues w/ cross product