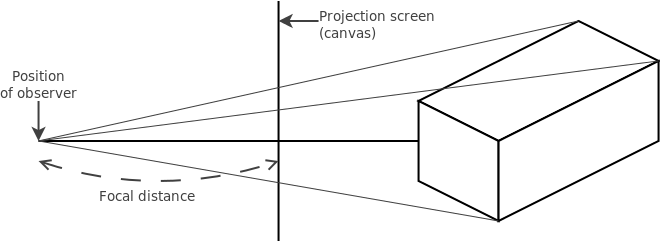
**3D simulation in Java**

This is a small program to showcase vector approaches to 3D rotation and translation in a 3D space, projected on a screen at some focal distance of the observer.

**Projection**

To start with some assumptions, we have the observer fixed at (0, 0, 0). That is us. The projection screen, on which the objects are drawn, is supposed to be at a certain distance from the observer, in the XY plane. This distance is the focal distance.

[](https://github.com/wthielen/3dsim/raw/master/img/projection.png)

In this example I am creating a world that is projected on a screen of 600x600, put on a focal distance of 300 pixels from the observer:

World w = new World(600, 600, 300);

**Objects**

Every object is derived from a BaseObject class. This BaseObject has a placement vector p, and its own set of XYZ axis vectors axis[3]. The coordinates of the vertices needed to draw this object, are calculated with respect to this set of XYZ axis vectors. These axis vectors are placed at the position of the placement vector of this object.

So a vertix on position v = (v.x, v.y, v.z) will actually be placed in the 3D space on position (p.x + axis[X] x v, p.y + axis[Y] x v, p.z + axis[Z] x v) where x is the dot product operation. This calculation is done in the mapVertices function that returns an array of X and Y coordinates for every vertix mapped on the projection screen.

As you can deduct from the projection sketch, the X and Y coordinates on the projection screen are the fractions of the real X and Y coordinates with the focal distance divided by the real Z coordinate:

x = v.x \* focus / v.z;

y = v.y \* focus / v.z;

**Translation**

Translation is moving the object along a displacement vector v. This is pretty easily done by adding the displacement vector to the position vector:

this.p.add(v);

This way, all the vertices for this object are also translated, without having to recalculate them separately. This is done in the mapVertices function.

**Rotation**

Rotation is performed using rotation vectors. Classical mechanics teach us that the cross product of the rotation vector with the position vector gives us the positional displacement, or actually the speed vector from that position. This speed vector or displacement is called delta here:

delta = p x omega;

The x operator in this case is the cross product. But this displacement alone is not enough: we also need to maintain the rotation radius, which is the length of the original position vector.

p = (p + delta) \* |p| / |p + delta|;

The vertical bars here |v| mean the length of the vector v.

This calculation is done on the axis[3] vectors of the BaseObject in the rotate(Vector3d omega) function. But to maintain the property that the three axes are perpendicular to each other, the function also carries out some cross products to ensure that the axes are perpendicular. Here is the order:

1. Rotate the X axis
2. Rotate the Y axis
3. Calculate the Z axis as the cross product of the rotated X and Y axes. The resulted Z axis will be equally rotated
4. Ensure that the Y axis is perpendicular to the X axis, by calculating the cross product of the Z and X axes (order is important here!)

By rotating the object's axes, we have also rotated all the vertices for this object, without having to recalculate them, because the vertices' coordinates are with respect to the axes we just rotated.

**Rotation speed**

The length of the rotation vector is the rotation speed in radians per second. But the World presentation is redrawn at a rate of World.fps frames per second. So we need to rescale the displacement vector to get the vector responsible for the current frame's displacement. This is done by:

delta.cross(p, omega);

delta.scale(1.0/this.w.fps);