Coordinate frames play a big role in computer graphics. Here are the frames that matter in the next assignment.

model The model of the cube is 1 by 1 by 1 with one vertex at the origin and the rest in the first octant.

object The object is a 1 by 1 by 1 cube with the origin at its *center*. This is just a simple example. We could change its size or even make it 1 by 2 by 3.

rotated The rotated object.

world The rotated and translated object, now somewhere in the world.

view The world from your point of view. Your x points out your right ear, your y points out the top of your head, and your z points out the back of your head. Where is the object in THAT coordinate system? The cube's coordinates change in the view coordinate system when YOU move. That's relativity.

proj The coordinates after applying orthogonal or perspective projection. We only see the partion of the world between $z = -n$ (near) and $z = -f$. The projection transformation takes $z = -n$ to $z = -1$, the "front" of the clip box, and $z = -f$ to $z = 1$, the "back" of the clip box. Orthogonal projection otherwise leaves x and y the same. For prog04, we will use near plane $z = 1$ (yes, that's behind us!) and far plane $z = -1$, so orthogonal projection just sets z to $-z$. We will learn about perspective projection later.

clip The coordinates in the clip box. We need to squish the x coordinate by the reciprocal of the aspect ratio because the clip box is 2 by 2 (by 2) but the canvas isn't.

canvas Just to be fancy, I have made it 960 by 540 (16/9 aspect ratio).

For each pair of adjacent frames, we will create matrices (Mat) which transform coordinates in one to the other. Hence model2object and object2model, etc. These must be inverses of each other. For debugging, whenever you set a pair, you should print out (console.log) their product to make sure it is the identity.

Mat does not have a general inverse() method. Instead you will use the fact that the inverse of a rotation, translation or scale matrix is a rotation, translation or scale matrix. If your matrix is a product, then use the fact that

$$(AB)^{-1} = B^{-1}A^{-1}$$

What is the inverse of

- Mat.rotation(i, a)?
- Mat.translation(v)?
- Mat.scale(s)?
- Mat.rotation(i, a) · Mat.translation(v) · Mat.scale(s)?

Here are the matrices you will need:

model2object translates the center of the model cube to the origin.

object2rotated is a rotation matrix, initially the identity. If we want to apply the rotation R, should the new matrix be R times object2rotated or rotated2object times R? Well, if p is a corner of the object, its current position is object2rotated * p, so if we rotate that by R, we get R * (object2rotated * p) which equals (R * object2rotated) * p because matrix multiplication is associative. (It is NOT commutative.) So the new matrix is R * object2rotated.

rotated2world similarly applies a translation matrix T that is set by clicking the z+ or z- button or dragging. For dragging, T translates by the translation that takes where the mouse was last down to where to mouse is currently. You need to transform this translation to world coordinates.

world2view is the identity for now.

view2proj just scales z by -1 for now.

proj2clip scales x by the reciprocal of the aspect ratio.

clip2canvas translates to take (-1,1) to (0,0) and then scales to take (1,-1) in the clip to (width,height) in the canvas. What is the translation matrix T (easy)? What is the scale matrix S? Be careful: we are not scaling (1,-1). We are scaling what (1,-1) becomes after translation. Finally is clip2canvas S * T or T * S?

If p is a point of the model, where is it located in the clip box?

proj2clip * (view2proj * (world2view * (rotated2world * (object2rotated * (model2object * p)))))

right? (Sorry for the small font.) So what the single matrix model2clip that takes p from the model to the clip box?

model2clip = proj2clip * view2proj * world2view * rotated2world * object2rotated * model2object

We load the points of the model into a buffer and attach them to the attribute variable vPosition in the vertex shader. We pass the matrix model2clip to the vertex shader to apply to each point to get its position in the clip box. Whenever you update any of the matrices above, you need to recompute model2clip.

The standard technique is to have only three matrices

model M = rotated2view * object2rotated * model2object

view V = world2view

projection P = proj2clip * view2proj

so one passes P * V * M to the vertex shader. As I have explained, I like to break things up into more "bite sized" pieces and let the computer do the multipling for us!