This note describes the "simple 3D viewing" system that I described at the end of class on Tuesday. I'm interested in corrections and improvements; perhaps we can lobby FvDFH to incorporate parts of it in Chapter 7, as a way to introduce the concepts and to give readers a better intuition for what's going on. This material was derived fromWatt's *Fundamentals of Three-Dimensional Computer Graphics*.

1 Viewing (M_{view})

To a first approximation, there are only two variables that can be set in our simple viewing system: the position of the viewer's eye and the distance the viewing plane is away from the eye. The viewer is always looking at the origin of the "world space," and the viewing plane is always normal to the line of sight. More about this later. We say that the eye is at position P and the origin of world space is the point O.

The position of the eye defines the origin of a new coordinate system called the "eye space". It is a left-handed coordinate system whose z axis sits on the line PO. As we look along the eye's z axis, the values are getting larger as things are farther away. The x axis of the eye-space is parallel to the x-y plane of world space.

The gist of things is that we need to find a matrix that will convert a point in world space into eye space. This matrix is completely specified by P's location in world-space. As it turns out, the matrix is easy to derive (and to write down) if we specify P in spherical coordinates rather than cartesian coordinates. So P is given by a triplet of numbers, (μ, θ, ϕ), with the following interpretation:

μ is the distance from the viewer to the origin.

θ is the angle made between the line of sight, PO, and the plane containing the x-z axes.

ϕ is the angle between the line of sight and the plane containing the y-z axes.

To check your understanding, consider the point P with $\mu = 10$, $\theta = -30$, and $\phi = 45$. The eye is somewhere in the octant with positive values for all three coordinates, and the viewer would be looking down and to his left to see the origin.

With some high school trigonometry, we find that

$$\begin{aligned} P_x &= \mu cos\theta sin\phi \\ P_y &= -\mu sin\theta \\ P_z &= \mu cos\theta cos\phi \end{aligned}$$

Note that negative sign for P's y coordinate. The negative sign is needed because θ is negative in order to "raise" a line above the xy plane. (An easy sanity check that the coordinates are reasonable is to look at the square of the distance from P to the origin using the cartesian coordinates. It should equal μ^2. This doesn't convince you the sign is right, or that we haven't swapped a cos with a sin, but it's nice to know that there's nothing to prove the coordinates are wrong.)

To make a long story short, the matrix that converts a point from world space into eye space is

$$M_{view} = S(1, 1, -1) \cdot Rx(-\theta) \cdot Ry(-\phi) \cdot T(-P_x, -P_y, -P_z)$$

where the matrices for scale, rotation, and translation are found in FvDFH, pages 214–5.

2 Projecting (M_{proj})

As mentioned, in our simple model the viewing plane is located normal to the eye space's z axis (the line from P to O), at a distance d away from P. That is, in the eye space, the plane is at $z = d$, parallel to the eye space's xy plane.

To perform a perspective projection, transform points in eye space by

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

If you ignore clipping and hidden surface elimination, you can now homogenize the resulting points. That is, you need to divide the x and y coordinates by z/d.

To perform an orthographic projection, you don't need to do anything. If, for the sake of symmetry, you'd like to do a "projection" type transformation, simply transformi your points in eye space by the identity matrix:

$$M_{orth} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To summarize, the projection matrix is M_{proj}. It is either M_{per} or M_{orth}, depending on the desired type of projection. It projects objects from eye space into "screen space."

3 Imaging (M_{image})

The final thing you need to do is to get a part of the viewing plane onto "image space" on an SRGP canvas (perhaps via a SUIT widget). How do you specify such window onto of the view plane?

In our simple model, we define a screen space coordinate system in the view plane whose origin is the spot where the line PO hits the plane. The vertical and horizontal axes, usually labeled u and v, are parallel to the x and y axis of our eye space.

Since we are always looking down at the origin, there's a good chance that we've set up the objects to be viewed such that they are around the origin. So, for the sake of keeping things simple, we'll specify a size w of the viewing window centered about the origin. That is, the viewing window goes from $(-w, -w)$ in the lower left to (w, w) in the upper right. With a regular window nicely centered about the origin, the transformation to a viewport in an SRGP canvas is simple:

$$M_{image} = T(lx, by, 0) \cdot S\left(\frac{rx - lx}{2w}, \frac{ty - by}{2w}, 1\right) \cdot T(w, w)$$

where the southwest corner of the canvas is (lx, by) and the northeast corner is (rx, ty). This is just a special case of the window-to-viewport transformation we've seen a few times in class already. You are now ready to draw the object by just ignoring the z coordinate.

4 Putting it all together

If you are not clipping or performing hidden surface elimination, you can combine the three matrices together into a single matrix, transform each point from the world space into image space, and then homogenize transformed the point. The cumulative transformation matrix is as follows

$$M_{crm} = M_{image} \cdot M_{proj} \cdot M_{view}$$

Don't forget to homogenize the transformed point!

5 Rotating the cube

Most of you have had no problems computing the matrices for rotating the cube about its main diagonal. Just in case there's any confusion, here's the scoop.

Assume the cube is centered about the origin in world space, going from $(-1, -1, -1)$ to $(1, 1, 1)$. The matrix to spin the cube by α degree forward is

$$\begin{aligned} M_{spin} = \quad & T(-1,-1,-1) \cdot Ry(-45) \cdot Rz(arcsin(1/sqrt(3))) \cdot \\ & Rx(\alpha) \cdot \\ & Rz(-arcsin(1/sqrt(3))) \cdot Ry(45) \cdot T(1,1,1) \end{aligned}$$

It is tempting to combine this matrix with the three above. Your life will be easier if you don't. Because M_{spin} rotates the cube from its current position by α degrees, you need to keep the transformed points around each time you spin the cube a bit.

In other words, the high-level structure for your program is as follows:

```
cube = initial position of cube
Mctm = Mcanvas * Mproj * Mview;
loop
    cube = Mspin * cube
    cubeP = Mctm * cube
    cubeP = homogenize cubeP
    display cubeP
end
```

If you aren't doing any clipping or hidden surface elimination, and you combine M_{canvas}, M_{proj}, and M_{view} into a single cumulative transformation matrix, M_{ctm}, you should still keep the individual matrices around so the user can change some of the parameters interactively.

For example, in my program, I allow the user to interactively set the following items:

- viewing parameters μ, θ, and α. These affect M_{view} only.
- projection parameters d, and a perspective vs. orthographic flag. These affect M_{proj} only.
- canvas parameters w, and the lx, by, rx, ty which define a location of the SUIT widget. These affect M_{canvas} only.

Of course, after changing one of the parameters, the cumulative matrix must be recomputed by matrix multiplying the three matrices that comprise it.