1 Perspective Transformation

Imagine you are inside the unit box from -1 to 1 in x, y, and z, with your eye at $(0, 0, 0)$ looking down the negative z-axis at a plane n (near) distant ($z = -n$) and a plane f (far) distant ($z = -f$). You can only see out the back square: $-1 \le x \le 1$, $-1 \le y \le 1$, $z = -1$. Because your line of sight spreads with distance, you can see a larger square $-n \le x \le n$, $-n \le y \le n$ on the near plane $z = -n$ and an even larger square $-f \le x \le f$, $-f \le y \le f$ on the far plane $z = -f$. The zone you can see between these two squares is a *frustum*: a truncated pyramid.

1.1 Projective Transformation

What does the frustum look like to your eye? It looks like a cube, head on, in orthogonal projection. Right? The square in the $z = -n$ plane exactly covers up the square in the $z = -f$ plane. If we think of the $z = -1$ square as a screen, the $(x, y, -1)$ point corresponds to $(nx, ny, -n)$ and $(fx, fy, -f)$ and all the points on the segment between them. So if we transform $(nx, ny, -n)$ to $(x, y, -1)$ and $(fx, fy, -f)$ to $(x, y, 1)$ and similarly the points between them, then the result will look like an orthogonal projection of the clip box, which is exactly what the graphics card generates for us!

Two line segments in the frustum which both "point to" $(0, 0, 0)$ are transformed into parallel line segments. Ordinarily, a linear transformation cannot transform intersecting lines into parallel lines. However, a linear transformation of *homogeneous* coordinates can.

We want all the squares scaled to the dimensions of the box. Since the squares are different sizes, we cannot use a single linear scale factor. Instead, we use the power of homogeneous coordinates. If we set $x' = x$, $y' = y$, and $w' = -z$ (let's deal with z' and z later), then the homogeneous coordinates $[x', y', z', w']$ has actual coordinates $(x'/w', y'/w', z'/w')$. So points on the near plane $z = -n$ are divided by $w' = -z = n$ and those on the far plane are divided by $w' = -z = f$. In both cases, the squares are mapped to the unit square $-1 \le x \le 1$, $-1 \le y \le 1$.

We would like to map the near plane square to the back of the box (actually the front of the clip box because its z goes the other way) $-1 \le x \le 1$, $-1 \le y \le 1$, $z = -1$. So we should map the far plane square to the front of the box $-1 \le x \le 1$, $-1 \le y \le 1$, $z = 1$. That way the inside of the frustum is mapped to the inside of the box. The coordinate that does this is z'/w'. We already have decided $w' = -z$. z' needs to be a linear function in x, y, z, w because we want to implement it with a matrix. It can't depend on x or y because changing x of y does not change z'/w'. So $z' = az + bw$.

For $z = -n$ and $w = 1$, we want $z'/w' = -1$. Since $z' = az + bw$ and $w' = -z$, this means $(az + bw)/ - z = (a - n + b1)/ - (-n) = -1$ or $-n = a(-n) + b$.

For $z = -f$ and $w = 1$, we want $z'/w' = 1$ (map to front face). This leads to $f = a(-f) + b$.

Solving for a and b, we get

$$a = -\frac{f + n}{f - n} \qquad \text{and} \qquad b = -\frac{2fn}{f - n}.$$

The equations $x' = x$, $y' = y$, $z' = ax + bw$, and $w' = -z$, can be written in matrix form,

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

We can call this matrix *view2proj*. How can we calculate its inverse?

1.2 Proj to View

We know $w' = -z$ and so $z = -w'$. Since $z' = az + bw$, $z' = -aw' + bw$ or

$$w = \frac{1}{b} z' + \frac{a}{b} w'.$$

If we let

$$a' = \frac{1}{b} = -\frac{f-n}{2fn} \qquad \text{and} \qquad b' = \frac{a}{b} = \frac{f+n}{2fn},$$

then the inverse matrix is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & a' & b' \end{bmatrix}$$

1.3 Proj to Clip

The clip box is square, and so the portion of the view that we display is square. BUT the canvas is not necessarily square. If it is wider than it is tall, such as a 16 by 9 HD display, then the view will look stretched in the x-direction. Back in the day, we bought an HD-ready set before our cable company had HD content (or maybe we were too cheap to upgrade). SD is only 4 by 3. HD is 30% wider. So we wouldn't "waste" our new screen, we set it to stretch the picture, making everyone look somewhat dumpy.

It would be nice if we could change the x and y dimensions of the clip box, but we can't. So what we do is "squish" the wider x into the clip by scaling it by 9/16, the reciprocal of the aspect ratio. This is *proj2clip*.

Because the view is spreading as it leaves the back of the box, the frustum is much larger than the unit cube. Since we display the entire frustum, the cube looks small. So we need to zoom. To zoom by a factor of two, scale both x and y by two. Of course, this is on top of the scaling to fix the aspect ratio. In general, if the cube is distance d from the eye/camera, the zoom factor should be d.

2 View Coordinate Frame

The view coordinate frame is the coordinate frame of the viewer or camera. As the viewer, you look forward along the negative z-axis. The positive z-axis points out the back of your head. The x-axis points out of your right ear, and the y-axis points out the top of your head.

2.1 Looking at an object

Suppose your eye is at position e and the object is at position o and your eye is "looking at" the object. That means that the z-axis vector v_z is a unit vector pointing from o to e.

Looking at the object orients your head somewhat, but it can still *roll*, rotate around your (the view) z-axis. That's what a dog does when it looks confused. It is natural for you to keep your head "level" meaning your ears are the same height off the ground and your x-axis is parallel to the ground, which means your x-axis vector v_x is *perpendicular* to the world's y-axis vector w_y (which points up from the floor). How can we choose v_x that is perpendicular to both the world y-axis $w_y = (0, 1, 0)$ and your own z-axis v_z? Obvious, right? Don't forget to unitize. Once you have v_z and v_x, how do you get v_y, which is perpendicular to both?

Next, you can create a matrix with v_x, v_y, v_z, and e as its columns using

```
new Mat(c1, c2, c3, c4)
```

This matrix times the origin $(0, 0, 0, 1)$ is e, and times the vector $(a, b, c, 0)$ is $av_x + bv_y + cv_z$. So this is indeed the transformation *view2world*.

How do we get its inverse *world2view*? If we call new Mat with just v_x, v_y, and v_z, then the fourth column will default to $0, 0, 0, 1$ and the matrix will be a pure rotation since its translational column is the origin. Call that matrix R. We can also create the matrix T which is a pure translation by e. The matrix *view2world* is the product of R and T. Which order? (Hint: which one transforms the origin correctly?) To take the inverse of the product, use the fact that the inverse of a rotation matrix is its transpose.

2.2 Moving the camera

If we translate or rotate the camera or you, the viewer, how do *world2view* and *view2world* change? If we step to the right, applying translation matrix S to our current position/orientation, what do we see? From our point of view, the world does the opposite and moves to the left. In our view, the world is transformed by S^{-1}.

Let p be a point in the world. We see it at $world2view \cdot p$ in our view coordinate system. If this is transformed by S^{-1}, then the new view position is

$$S^{-1} \cdot (world2view \cdot p) = (S^{-1} \cdot world2view) \cdot p$$

by the associate property of matrix multiplication. So what is the new value of *world2view*? Of *view2world*?

3 Zooming

Just multiply *view2proj* by a scale matrix.

4 Dragging the cube with the mouse

Dragging acts as if it is dragging the center of the object. If we drag from a_{canvas} to b_{canvas}, the we should add the vector $b_{\text{world}} - a_{\text{world}}$ to the current translation of the object and update *rotated2world* (and *world2rotated*). But what does a_{canvas} mean? It's where we click, but are we clicking at $z = -1$ or $z = 1$ or somewhere in between. Because of the perspective transformation, this makes a difference because $z = -1$ cooresponds to the "near square" which is smaller than the "far square" at $z = 1$ once they are transformed to world coordinates.

The answer is to translation the center o_{world} to o_{canvas} and use its z coordinate.

5 Rotating the cube with the mouse

If we click a face of the cube or indeed anywhere, we want to see the cube move with the mouse. We have already implemented that. (Just make sure you homogenize now.) However, if we click a *vertex* of the cube, we want to see the cube *rotate* with the mouse.

First, we have to figure out if the mouse down has clicked "on", say within 4 pixels of, a vertex of the cube.

- Let d_{canvas} be the down point in the canvas (with z equal zero).
- For each vertex point p_{model} in the cube (model) coordinates, transform p to its canvas coordinates p_{canvas}.
- Set the z-coordinate of p_{canvas} to zero.
- If the distance from p_{canvas} to d_{canvas} is less than 4, that is the vertex you have clicked on.

Now let's drag that vertex.

- Let d_{canvas} be the drag point in the canvas. This time create f_{canvas} and b_{canvas} with the same x and y as d_{canvas} but $z = -1$ and $z = 1$. f and b stand for *front* and *back.*
- Transform these points into the rotated frame. Just call them f and b here. Don't forget to homogenize.
- Let u be the unit vector parallel to the line fb.

- Let o be the center of the model and v be the vector from o to the vertex p being dragged as before.
- Set w to the vector from o to the point p on line fb that is closest to o. We know this point is closest if w is perpendicular to u.
- To find w, express it as $w = f + us - o$, use the fact that the dot product $u \cdot w = 0$ (and $u \cdot u = 1$). Solve for s and plug in.
- If w is longer than v, we scale it down to have the same magnitude as v.
- If w is shorter than v, add a multiple of u to make it the same magnitude as v: $(w + ut)^2 = v^2$, expand and solve for t. Remember $u \cdot w = 0$.
- The square root can have two signs. Pick the sign of t that makes $w + ut$ closest to v.
- Set w equal to that vector.

So we need to rotate v to w. About what axis? The axis should be perpendicular to both so v rotates into w.

- Set v_x and w_x to v and w unitized.
- Set v_z and w_z equal to a unit vector perpendicular to both v and w. We will rotate about this vector.
- Set v_y and and w_y to make v_x, v_y, v_z and w_x, w_y, w_z a right-handed coordinate system. (Remember the "circle of life".)
- We need a transformation that takes the v's to the w's. Start with the transformations
  - V = new Mat(v_x, v_y, v_z)
  - W = new Mat(w_x, w_y, w_z)
- V takes vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ to what three vectors? What about W?
- What matrix does the opposite?
- What matrix (product) takes the v's to the w's?