The OpenGL viewing pipeline

PARAG CHAUDHURI
Department of Computer Science and Engineering,
Indian Institute of Technology, Delhi

1. INTRODUCTON

OpenGL is forms the obvious choice for countless graphics programmers because it is fast and astoundingly powerful. In spite of that it has a simple API. It is almost always adequate for all graphics programming needs.

The whole OpenGL rendering pipeline is quite a complicated thing. It includes processing of vertex and pixel data, texture assembly, rasterization, per-vertex operations, primitive assembly, etc. Here we look at only a very small but very important part of the whole rendering pipeline i.e. *the viewing pipeline*. Here we will try to see how the our models get interpreted in the OpenGL world before making a final appearance on a monitor window.

So here is a closer look at the inner workings of OpenGL viewing pipeline.

2. THE OPENGL VIEWING PIPELINE

2.1 The Model Coordinates and Tranformations

The coordinates we specify using the *glVertex** commands are the model coordinates. The *glRotate*, *glTranslate* and *glScale* commands are used to transform the model into the desired orientation and size. After applying the modelling transformations to the model coordinates what we get are world coordinates. The Modelling tranformations give rise to 4×4 matrices. Descirbed below are the various modelling transformations and the matrices they produce.

—A call to *glTranlate**(x,y,z) generates T, where

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

—A call to *glScale**(x,y,z) generates S, where

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

—A call to *glRotate**(a,x,y,z) generates R, as follows:

Let $v = (x,y,z)^T$ and $u = v/\|v\| = (x',y',z')^T$

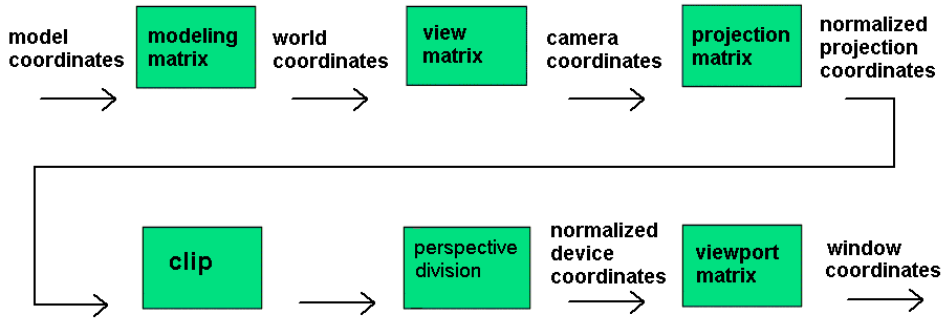

Fig. 1. The OpenGL viewing pipeline

Then, if

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \tag{3}$$

and,

$$M = uu^T + (\cos a)(I - uu^T) + (\sin a) \cdot S \tag{4}$$

Then,

$$R = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

where m represents elements from M, which is a 3×3 matrix.

2.2 The Camera Coordinates and Viewing Transformation

The next step is to convert the world coordinates into camera coordinates. This is genrally done using the *gluLookAt* command. This performs translations and rotations to transform a point from world coordinates to camera coordinates. So the matrix generated is a combination of translation and rotation matrices. So we can derive the camera matrix as a cartesian frame to frame transform. The camera frame is given by the parameters given to *gluLookAt* in the form of an eye position, a look-at point and an up vector.

Let the *eye* be given by $C = (x_c, y_c, z_c)$ and the look-at point be $A = (x_a, y_a, z_a)$ (see Figure 2). Now, if we were to define the camera frame as

$$F_{camera} = (\vec{u}, \vec{v}, \vec{w}, C) \tag{6}$$

then we get,

$$\vec{w} = \frac{C - A}{\|C - A\|} \tag{7}$$

$$\vec{u} = \frac{\vec{y} \times \vec{w}}{\|\vec{y} \times \vec{w}\|} \tag{8}$$

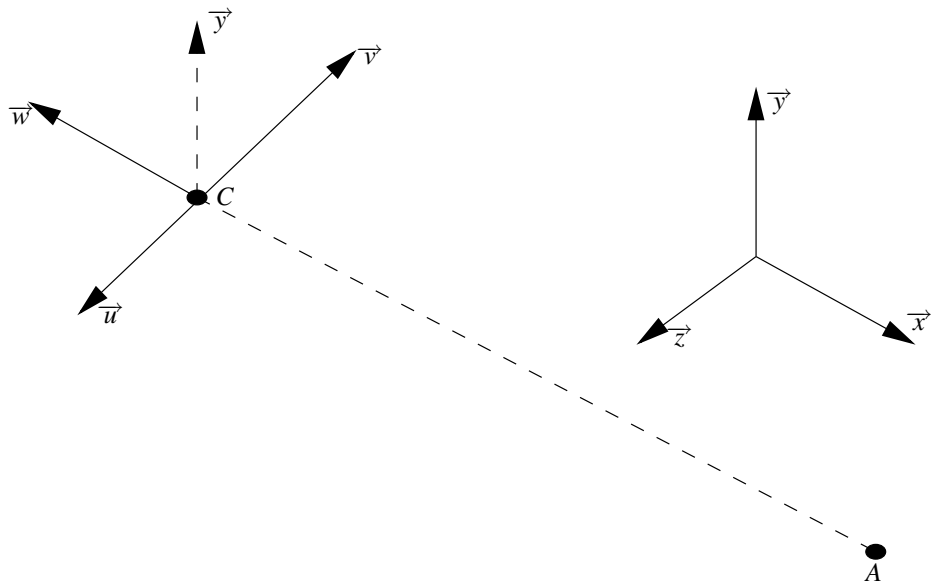

Fig. 2. Camera Coordinate Transformations

$$\overrightarrow{v} = \overrightarrow{w} \times \overrightarrow{u} \tag{9}$$

based on the assumption that the vertical direction of the camera must be in the plane defined by $\overrightarrow{w}$ and $\overrightarrow{y} = \langle 0,1,0 \rangle$.

Now we can write each of the vectors $\langle 1,0,0 \rangle$, $\langle 0,1,0 \rangle$, $\langle 0,0,1 \rangle$ as a linear combination of $\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}$ (because they are linearly independent). We can also write the vector $(0,0,0) - C$ as a linear combination of $\overrightarrow{u}, \overrightarrow{v}, \overrightarrow{w}$. So we can calculate the values $e_{i,j}$ as

$$\langle 1,0,0 \rangle = e_{1,1}\overrightarrow{u} + e_{1,2}\overrightarrow{v} + e_{1,3}\overrightarrow{w} \tag{10}$$
$$\langle 0,1,0 \rangle = e_{2,1}\overrightarrow{u} + e_{2,2}\overrightarrow{v} + e_{2,3}\overrightarrow{w} \tag{11}$$
$$\langle 0,0,1 \rangle = e_{3,1}\overrightarrow{u} + e_{3,2}\overrightarrow{v} + e_{3,3}\overrightarrow{w} \tag{12}$$
$$\langle 0,0,0 \rangle = e_{4,1}\overrightarrow{u} + e_{4,2}\overrightarrow{v} + e_{4,3}\overrightarrow{w} + C \tag{13}$$

On solving this we get the tranformation matrix for our cartesian frame to frame transformation as

$$\begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} & 0 \\ e_{2,1} & e_{2,2} & e_{2,3} & 0 \\ e_{3,1} & e_{3,2} & e_{3,3} & 0 \\ e_{4,1} & e_{4,2} & e_{4,3} & 0 \end{bmatrix}$$

But the fun is not over yet. OpenGL joins the modelling transforms and viewing transforms into a common matrix called the *Modelview Matrix*. So we get one 4x4 matrix which takes the model coordiantes to the camera coordiantes.

Note that the matrix is in *column major* order.

2.3 The Normalized Projection Coordinates and Projection Transformation

After applying the modelview matrix GL must now take the camera coordinates to the image space. This is done using the projection transformation. We specify the type of projection using the *glFrustum* or *gluPerspective* commands. These commands generally help define a viewing frustum by speficying the near and far planes and the field of view in the y-direction or the left/right and bottom/top extents of the frustum.

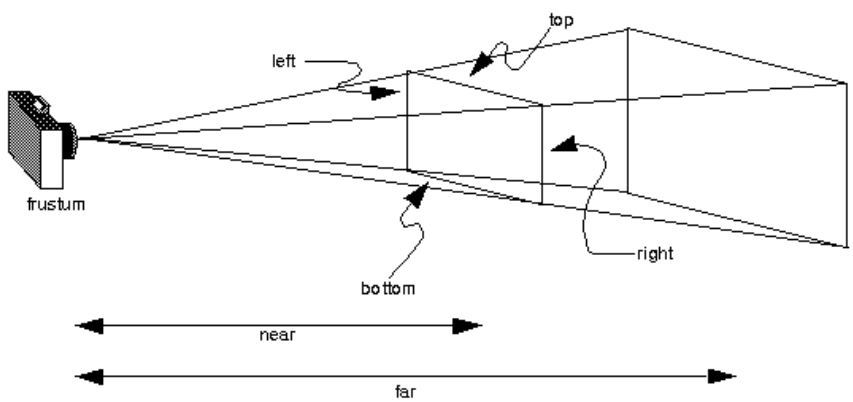

Fig. 3. The view frustum

A side view of the viewing frustum can be seen here (see Figure 4), with the u axis coming out of the paper. Note the field of view angle α.

This specification forms a viewing pyramid with the camera placed at the apex of the pyramid. The near and the far planes form a truncated pyramid or the *viewing frustum*. The projection transform this frustum into the *image space cube* $-1 \leq u, v, w \leq 1$.

To transform the frustum to a cube we start of with a transformation P of the form

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & -1 \\ 0 & 0 & b & 0 \end{bmatrix} \tag{14}$$

a and b are chosen such that the face of the frustum defined by the near plane $z = -n$ goes to the face $w = 1$ of the image space cube, and the face defined by the far plane $z = -f$ goes to the face $w = -1$. These (see Figure 5) yield the following equations

$$(0, 0, -n, 1) \cdot P = (0, 0, 1, n) \tag{15}$$

$$(0, 0, -f, 1) \cdot P = (0, 0, -1, f) \tag{16}$$

On solving we get,

$$a = (\frac{f+n}{f-n}) \tag{17}$$

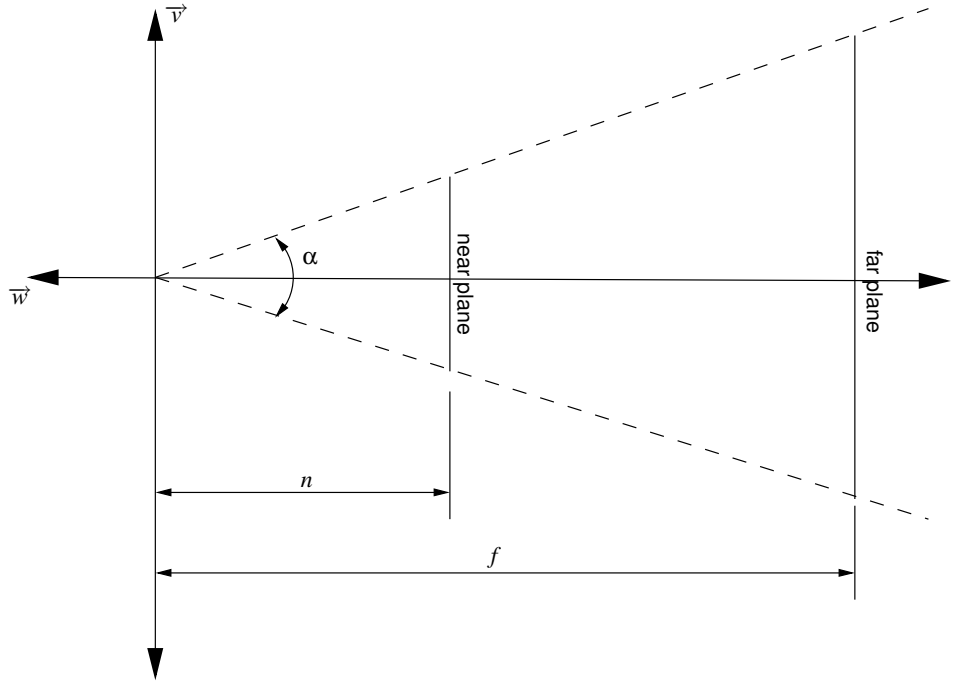

Fig. 4. The view frustum

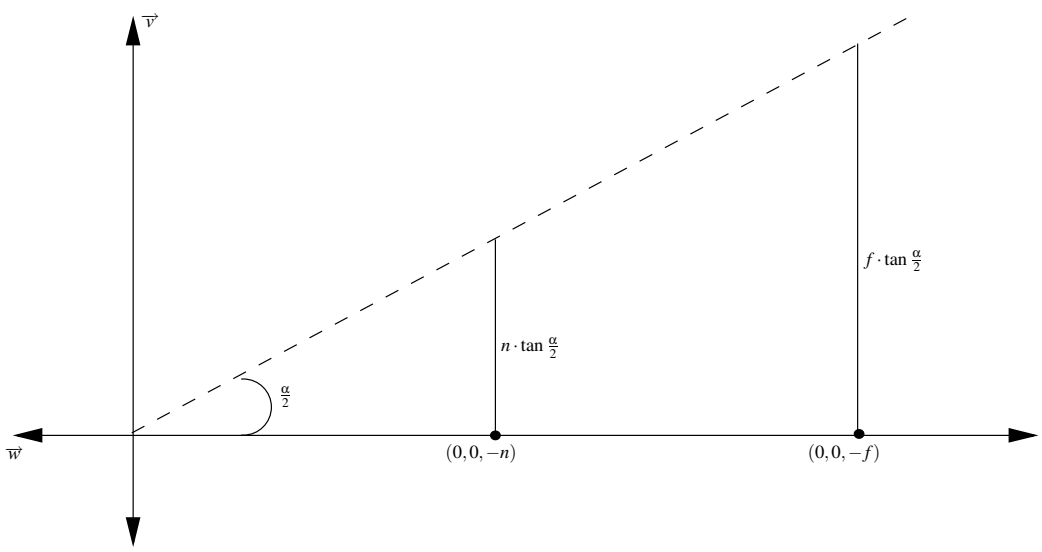

Fig. 5. Understanding the Projection Transformation

$$b = (\frac{2fn}{f-n}) \tag{18}$$

Now we need to map the rest of the image frustum to the image space cube. So consider points on the top plane that bounds the viewing pyramid in Figure 6.

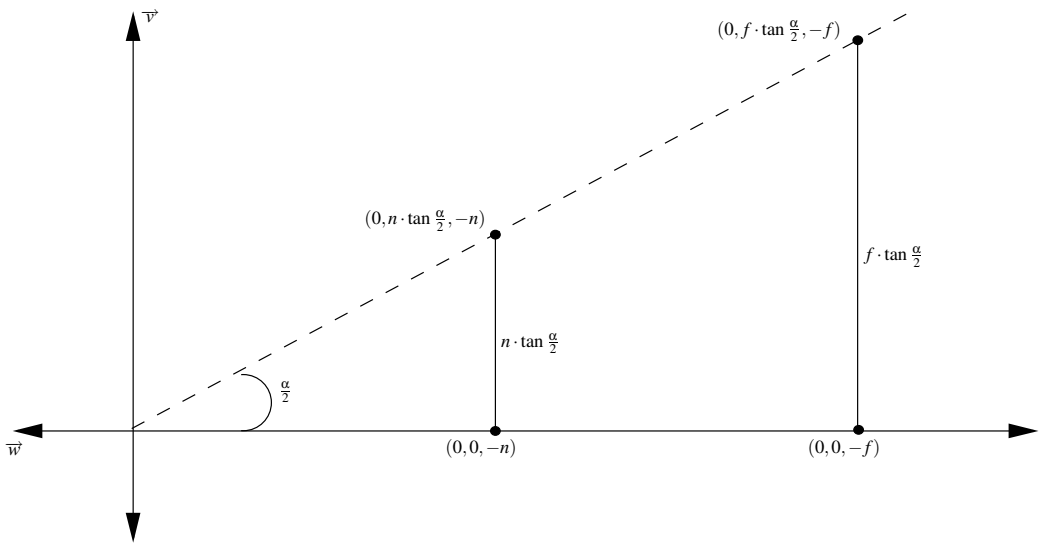

Fig. 6. Understanding the Projection Transformation

If we apply our transformation we get,

$$(0, n \cdot \tan\frac{\alpha}{2}, -n, 1) \cdot P = (0, n \cdot \tan\frac{\alpha}{2}, n, n) \tag{19}$$

$$(0, f \cdot \tan\frac{\alpha}{2}, -f, 1) \cdot P = (0, f \cdot \tan\frac{\alpha}{2}, f, f) \tag{20}$$

We can see that the v coordinates of the line $(0, -z \cdot \tan\frac{\alpha}{2}, z)$ for $w < 0$ are all mapped to $v = \tan\frac{\alpha}{2}$. Since we wish to map the v coordiantes to map to 1, we scale the x and y values by $c = \frac{1}{\tan\frac{\alpha}{2}}$.

This gives us our projection matrix as

$$P_{\alpha,n,f} = \begin{bmatrix} \frac{1}{\tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -1 \\ 0 & 0 & \frac{2fn}{f-n} & 0 \end{bmatrix} \tag{21}$$

2.4 Clipping and Perspective Divison

OpenGL clips the scene using the viewing frustum formed in the previous stage. It reconstructs the side of the clipped polygon. Then it performs perspective divison, i.e. non-homogenizing the coordinates. These are called *normalized device coordinates*. Now GL is ready to render the scene onto a window.

2.5 The Window Coordinates and Viewport Transformation

The viewport transformation determines the size of the rendered image. The windowing system is responsible for opening a window on the screen. By default the viewport is set to the entire pixel rectangle of the window. The *glViewport* command can be used to choose a smaller drawing region. In the window coordinates the lower left hand corner of a $n \times m$ pixel array is defined as the origin. Now any point (x, y) in the image space cube maps to $(\frac{n(x+1)}{2}, \frac{m(y+1)}{2})$. The z value is used to calculate the *Z-buffer*. The viewport aspect ratio should be the same as that of the view frustum or the image appears distorted.

3. CONCLUSION

This completes our discussion of how OpenGL takes a set of coordinates specified by the user from the modelling coordinate system to the final image coordinate system.