Transformation

Steve Canvas

yqykrhf@163.com

1 3D Transformation

1.1 3D rotaion

Rotaion around x-,y-,or z-axis in matrix method:

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

About the positive angle or negative angle: Whether right-hand axis or left-hand axis you choose, the approach to judge the positive direction of rotation is similar. Take right-hand axis for example, your right thumb points in the positive direction of the axis of rotation, then the other four fingers bend in the positive direction of the rotation.

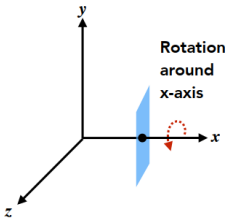

Figure 1: right-hand axis

Note: $\mathbf{R}_y(\alpha)$ is a little different from two others, that's because
we can compose any 3D rotation from R_x, R_y, R_z

$$\mathbf{R}_{xyz}(\alpha, \beta, \gamma) = \mathbf{R}_x(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_z(\gamma)$$

How to rotate around random axis? we assume the axis goes through the origin and the case that axis doesn't go through the origin will be discussed later. we normlize the axis to unit basis $\mathbf{n} = (\mathbf{n_x}, \mathbf{n_y}, \mathbf{n_z})^{\mathbf{T}}$ firstly, so $n_x^2 + n_y^2 + z_z^2 = 1$. we denote the original point as $\mathbf{v}$, and rotated point as $\mathbf{v}'$ (based on right-hand axis).

However, we want the form of matrix multiplication for calculation. Denote that matrix as $\mathbf{R_n}(\theta)$, rotation angle is θ, $\mathbf{n}$ is denoted as the axis of rotation. The derivation is as follows:

$$\mathbf{v}' = (\mathbf{n} \cdot \mathbf{v})\mathbf{n} + cos(\theta)[\mathbf{v} - (\mathbf{n} \cdot \mathbf{v})\mathbf{n}] + sin(\theta)(\mathbf{n} \times \mathbf{v})$$

A point $v = (x_v, y_v, z_v)^T$ can be describe like this:

$$\begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} = \mathbf{o} + x_v\vec{\mathbf{x}} + y_v\vec{\mathbf{y}} + z_v\vec{\mathbf{z}}$$

$$= \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + x_v \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + y_v \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + z_v \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\mathbf{R_n}(\theta) \begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} = x_v R_n(\theta) \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + y_v R_n(\theta) \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + z_v R_n(\theta) \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Make $\mathbf{v}$ equal to $\vec{\mathbf{x}}, \vec{\mathbf{y}}, \vec{\mathbf{z}}$ separately. Then we will obtain

$$\begin{pmatrix} \vec{\mathbf{x}'} & \vec{\mathbf{y}'} & \vec{\mathbf{z}'} \end{pmatrix}$$

$$= \begin{pmatrix} n_x^2(1-\cos\theta) + \cos\theta & n_x n_y(1-\cos\theta) - n_z\sin\theta & n_x n_z(1-\cos\theta) + n_y\sin\theta \\ n_x n_y(1-\cos\theta) + n_z\sin\theta & n_y^2(1-\cos\theta) + \cos\theta & n_y n_z(1-\cos\theta) - n_x\sin\theta \\ n_x n_z(1-\cos\theta) - n_y\sin\theta & n_y n_z(1-\cos\theta) + n_x\sin\theta & n_z^2(1-\cos\theta) + \cos\theta \end{pmatrix}$$

$$= \mathbf{R_n}(\theta)$$

If the axis doesn't go through the origin point. we need to translate firstly. e.g. Given two points(x_1, y_1, z_1) and (x_2, y_2, z_2). so the axis of rotaion $\mathbf{u} = (\mathbf{x_1} - \mathbf{x_2}, \mathbf{y_1} - \mathbf{y_2}, \mathbf{z_1} - \mathbf{z_2})^{\mathbf{T}}$ as shown Figure 2. Nomalize $\mathbf{u}$ to get $\mathbf{n} = \frac{\mathbf{u}}{||\mathbf{u}||}$.

If a 3D point $(x_v, y_v, z_v)^T$ rotate θ degree around $\mathbf{u}$ axis, the transformaion is as follows:

$$\begin{pmatrix} x'_v \\ y'_v \\ z'_v \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & x_2 \\ 0 & 1 & 0 & y_2 \\ 0 & 0 & 1 & z_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} R_n(\theta) \begin{pmatrix} 1 & 0 & 0 & -x_2 \\ 0 & 1 & 0 & -y_2 \\ 0 & 0 & 1 & -z_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_v \\ y_v \\ z_v \\ 1 \end{pmatrix}$$

In fact, Given a rotation angle and axis, we can express rotation. That's so-called **Euler angles**. Euler angesl is often used in flight simulators:pitch(x), yaw(y), roll(z). As shown in Figure 3

Unfortunately, Euler angels will cause singular expression called gimbal lock in some specifal cases.

Gimbal Lock

Let's describe this phenomenon with matrix method.

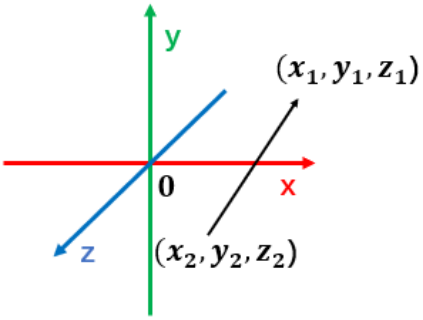

Figure 2: rotation around the axis

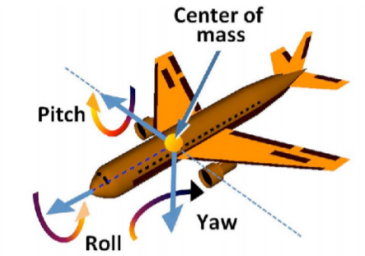

Figure 3: pitch(x), yaw(y), roll(z)

$$E(pitch, yaw, roll) = R_z(roll)R_y(yaw)R_x(pitch)$$

Every rotation is called gimbal. Whatever you change the rotation angle, this rotation order is unchangeable. In most cases, these three matrix can describe rotation around **three different axes**. But sometimes, we will lose one degree of freedom in special cases. Note: the degree of freedom is from the world view, not the object view itself. Here's a example.

1. rotate random angle around x axis. shown in Figure 4.
2. rotate $\Pi/2$ around y axis. shown in Figure 5.

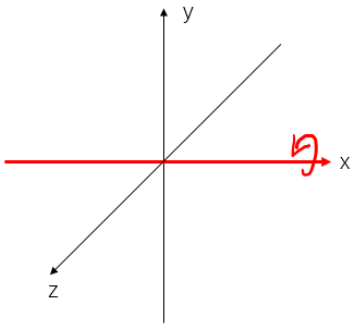

Figure 4: rotate random angel around x axis(pitch)

we follow the order: pitch(x), yaw(y), roll(z) to rotate. So far, we have made use of tow matrix $R_y(yaw)R_x(pitch)$, and $R_z(roll)$ left. If we rotate angle around z axis, we lose the blue degree of freedom as shown in Figure 6. The most

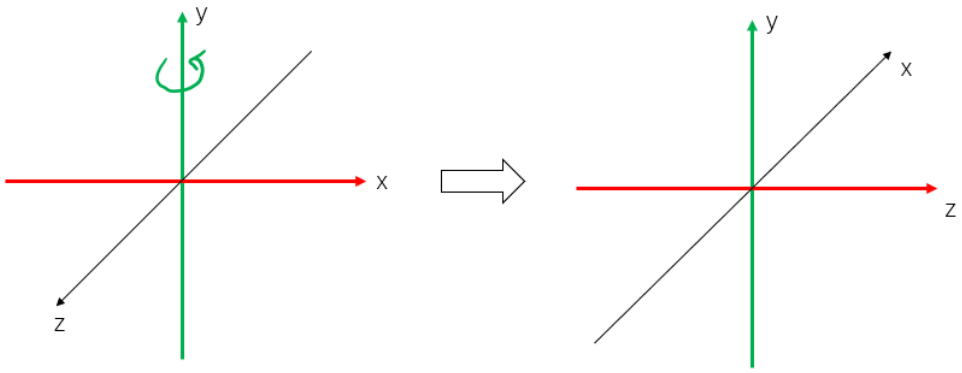

Figure 5: rotate $\Pi/2$ around y axis

important thing, we should

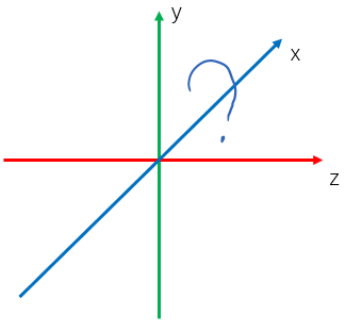

Figure 6: lose one DOF

Mathematically speaking,

$$
\begin{aligned}
E(\alpha, \pi/2, \beta) &= R_z(\beta) R_y(\pi/2) R_x(\alpha) \\
&= \begin{pmatrix} 0 & \sin(\alpha-\beta) & \cos(\alpha-\beta) \\ 0 & \cos(\alpha-\beta) & -\sin(\alpha-\beta) \\ -1 & 0 & 0 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha-\beta) & -\sin(\alpha-\beta) \\ 0 & \sin(\alpha-\beta) & \cos(\alpha-\beta) \end{pmatrix} \\
&= R_y\left(\frac{\pi}{2}\right) R_x(\alpha-\beta)
\end{aligned}
$$

Perhaps, you will think rotate angle γ around X axis after $R_z(\beta)R_y(\pi/2)R_x(\alpha)$. i.e.

$$E(\alpha, \pi/2, \beta, \gamma) = R_x(\gamma) R_z(\beta) R_y(\pi/2) R_x(\alpha)$$

As the matter of fact, you just decrease the probability of generaton of gimbal lock. It's not totally solved that because rotaion around other axis except y axis may cause other two or three axis coincidence.

The core of gimbal lock is the fixed rotation order

1.2 Transforming normal vectors

Denote a point of a surface as $\mathbf{p}$, and transformation matrix is denoted as $\mathbf{M}$, so the point after transformation is $\mathbf{M} \cdot \mathbf{p}$. a vector $\mathbf{t}$ the is tangent to the surface and is multiplied by M willbe tangent to the transformed surface. But, a surface normal vector $\mathbf{n}$ that transformed by $\mathbf{M}$ may not be normal to the transformed surface. It's obviously that:

$$n^T t = 0$$

We denote the desired transformed vectors as $\mathbf{t_M} = \mathbf{Mt}$ and $\mathbf{n_N} = \mathbf{Nn}$.

- the goal is to find $\mathbf{N}$ such that $\mathbf{n_N^T t_M} = 0$

The following is the derivation.

$$\begin{aligned} n^T I t &= n^T M^{-1} M t = 0 \\ n_N^T &= n^T M^{-1} \\ n_N &= (M^{-1})^T n \end{aligned}$$

so we get the transformation matrix for normal vector.

$$\mathbf{N} = (\mathbf{M^{-1}})^\mathbf{T}$$

1.3 Windowing Transformation

If we want to map a box $[x_l, x_h] \times [y_l, y_h] \times [z_l, z_h]$ to a new box $[x'_l, x'_h] \times [y'_l, y'_h] \times [z'_l, z'_h]$, we need translation and scaling matrix.

$$\begin{aligned} M &= T_2 R T_1 \\ &= \begin{pmatrix} 1 & 0 & 0 & x'_l \\ 0 & 1 & 0 & y'_l \\ 0 & 0 & 1 & z'_l \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 & 0 \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 & 0 \\ 0 & 0 & \frac{z'_h - z'_l}{z_h - z_l} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_l \\ 0 & 1 & 0 & -y_l \\ 0 & 0 & 1 & -z_l \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{x'_h - x'_l}{x_h - x_l} & 0 & 0 & \frac{x'_l x_h - x'_h x_l}{x_h - x_l} \\ 0 & \frac{y'_h - y'_l}{y_h - y_l} & 0 & \frac{y'_l y_h - y'_h y_l}{y_h - y_l} \\ 0 & 0 & \frac{z'_h - z'_l}{z_h - z_l} & \frac{z'_l z_h - z'_h z_l}{z_h - z_l} \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Here a small math tip:

$$\begin{pmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & x_t \\ a_{21} & a_{22} & a_{23} & y_t \\ a_{31} & a_{32} & a_{33} & z_t \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.4 Coordinate Transformation

We need origin $\mathbf{p}$ and a set of orthonormal basis $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$. Then we can describe a point in u, v, w coordiantes like this:

$$\mathbf{p} + u\mathbf{u} + v\mathbf{v} + w\mathbf{w}$$

we often write a point as ordered array (x_p, y_p, z_p). we set $\mathbf{o} = (0,0,0), \mathbf{x} = (1,0,0)^T, \mathbf{y} = (0,1,0)^T, \mathbf{z} = (0,0,1)^T$. Actually it's full expression is:

$$\mathbf{p} = (x_p, y_p, z_p) \equiv \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y} + z_p\mathbf{z}$$

we often set $\mathbf{o} = (0,0,0), \mathbf{x} = (1,0,0)^T, \mathbf{y} = (0,1,0)^T, \mathbf{z} = (0,0,1)^T$

Now, we have another coordinate origin $\mathbf{e}$ with basis $\mathbf{u}, \mathbf{v}, \mathbf{w}$. we express the same point $\mathbf{p}$:

$$\mathbf{p} = (u_p, v_p, w_p) \equiv \mathbf{e} + u_p\mathbf{u} + v_p\mathbf{v} + w_p\mathbf{w}$$

$$\begin{pmatrix} x_p \\ y_p \\ z_p \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & x_e \\ 0 & 1 & 0 & y_e \\ 0 & 0 & 1 & z_e \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_u & x_v & x_w & 0 \\ y_u & y_v & y_w & 0 \\ z_u & z_v & z_w & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_p \\ v_p \\ w_p \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} x_u & x_v & x_w & x_e \\ y_u & y_v & y_w & y_e \\ z_u & z_v & z_w & z_e \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_p \\ v_p \\ w_p \\ 1 \end{pmatrix}$$

To make it clearer, we write matrix like this:

$$\mathbf{p_{xyz}} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{p_{uvw}}$$

2 OpenGL Camera

OpenGL doesn't explicitly define neither camera object nor a specific matrix for camera transformation. Instead, OpenGL transforms the entire scene (including the camera) inversely to a space, where a fixed camera is at the origin (0,0,0) and always looking along -Z axis. This space is called **eye (camera) space**.

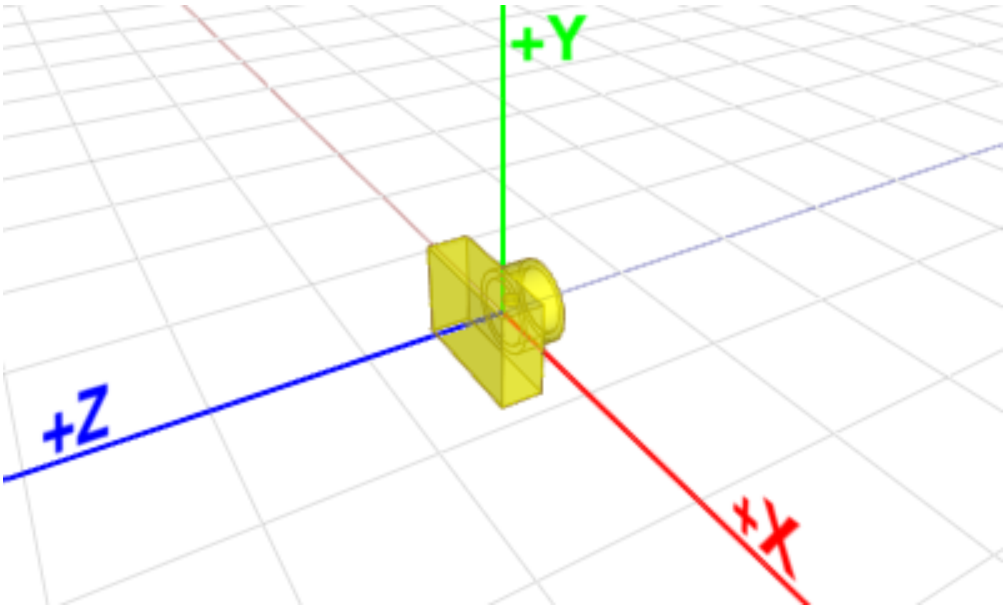

Figure 7: OpenGL camera is always at origin and facing to Z in eye space

Thus, OpenGL utilize a single *GL_MODELVIEW* matrix for both object transformation to world space and camera (view) transformation to **eye space(camera space)**.

You may break it down into 2 logical sub matrices; $M_{modelView} = M_{view}M_{model}$

That is, each object in a scene is transformed with its own M_{model} first, then the entire scene is transformed reversely with M_{view} The next section will discuss the detailes.

3 Viewing

we need three transformaiton to poject points in canonical space to screen.

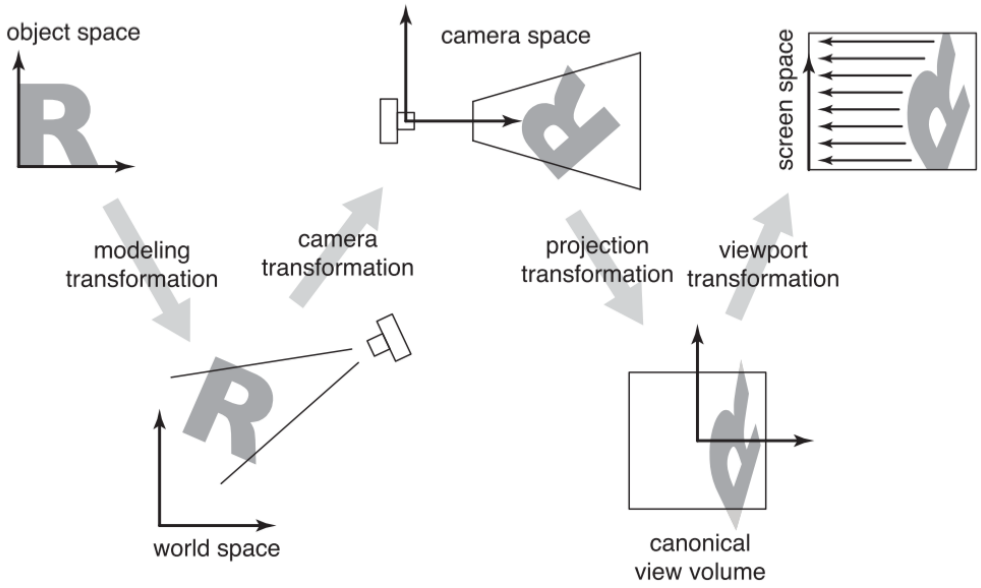

Figure 8: transformations that gets objects from their origin coordinates into screen space

1. The **camera transformaion** converts points in canonical coordinates (or world system) to *camera coordinates* or places them in *camera space*.
2. The **projection transformation** moves points from camera space to the *canonical view volume*.
3. The **viewport transformation** maps the canonical view volume to *screen space*

Other names: camera space is also eye space and the camera transformation is sometimes the viewing transformation; the canonical view volume is also clip space or normalized device coordinates; screen space is also pixel coordinates.

3.1 Camera(viewing) Transformation

Define a camera in virtual space by three vectors

- position $\vec{e}$
- look-at/gaze direction $\vec{g}$
- up direction $\vec{t}$

The initial postion is (0,0,0), and camera look at -z axis. As shown in Figure 9. So if we need to set model in specific position and get the projection of model from a specific angle, we need to do the following:

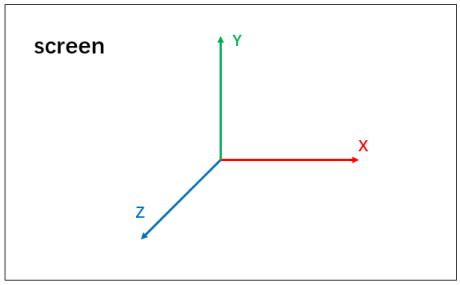

Figure 9: axis on screen

1. translate $\vec{e}$ to origin(0,0,0)
2. rotate $\vec{g}$ to -z axis
3. rotate $\vec{t}$ to y axis
4. rotate $\vec{g} \times \vec{t}$ to x axis

However, that's difficult to write. Actually, we could decompose M_{view} into $R_{view}T_{view}$ i.e translation and rotation.

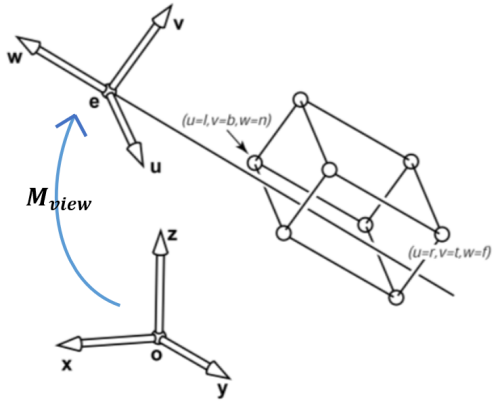

Figure 10: camera transformation

Obviously, we could get the T_{view} simply.

$$\mathbf{T}_{view} = \begin{pmatrix} 1 & 0 & 0 & -x_{\vec{e}} \\ 0 & 1 & 0 & -y_{\vec{e}} \\ 0 & 0 & 1 & -z_{\vec{e}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

According the base vector transformation rule, we get the R_{view}^{-1}

$$\mathbf{R}_{view}^{-1} = \begin{pmatrix} x_{\vec{g}\times\vec{t}} & x_{\vec{t}} & x_{\vec{-g}} & 0 \\ y_{\vec{g}\times\vec{t}} & y_{\vec{t}} & y_{\vec{-g}} & 0 \\ z_{\vec{g}\times\vec{t}} & z_{\vec{t}} & z_{\vec{-g}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And the rotaion matrix is orthogonal matrix.

$$\mathbf{R}_{view} = \begin{pmatrix} x_{\vec{g}\times\vec{t}} & y_{\vec{g}\times\vec{t}} & z_{\vec{g}\times\vec{t}} & 0 \\ x_{\vec{t}} & y_{\vec{t}} & z_{\vec{t}} & 0 \\ x_{\vec{-g}} & y_{\vec{-g}} & z_{\vec{-g}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{0} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

So far, we have got the matrix M_{view}.

$$\begin{aligned} \mathbf{M_{view}} = R_{view}T_{view} &= \begin{pmatrix} x_{\vec{g}\times\vec{t}} & y_{\vec{g}\times\vec{t}} & z_{\vec{g}\times\vec{t}} & 0 \\ x_{\vec{t}} & y_{\vec{t}} & z_{\vec{t}} & 0 \\ x_{\vec{-g}} & y_{\vec{-g}} & z_{\vec{-g}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_{\vec{e}} \\ 0 & 1 & 0 & -y_{\vec{e}} \\ 0 & 0 & 1 & -z_{\vec{e}} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \end{aligned}$$

Note:

$$\begin{pmatrix} R & & & t \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} R^{-1} & & & -R^{-1}t \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The coordinates of the model are stored in terms of the canonical (or world) origin o and the x-, y-, and z-axes

$$\mathbf{P_{uvw}} = \mathbf{M_{view}}\mathbf{P_{xyz}}$$

And we transform the objects together with camera using M_{view} which also known as **ModelView** transformation.

Note: In programming like *gluLookAt(eyex, eyey, eyez, centerx, centery, centerz, upx, upy, upz);*, the up vector(center - eye) is not orthogonal with up vector. so, we should adjust slightly as follows before construct the $\mathbf{M_{view}}$:

$$\begin{aligned} \mathbf{w} &= -\frac{\mathbf{g}}{\|\mathbf{g}\|} \\ \mathbf{u} &= \frac{\mathbf{t}\times\mathbf{w}}{\|\mathbf{t}\times\mathbf{w}\|} \\ \mathbf{v} &= \mathbf{w}\times\mathbf{u} \end{aligned}$$

3.2 Projection Transformaion

Project points from camera space so that all visible points fall in the range in -1 to 1 in x and y. It depends only on the type of projection desired. There are two kinds of approaches—perpective and orthographic projection.

3.2.1 Orthographic projection

The *canonical view volume* is a cube with side of length two centered at the origin shown Figure 12.

Of course, we usually want to render geometry in some region of space other than the canonical view volume. Our first step in generalizing the view will **keep the view direction and orientation fixed looking along z with +y**

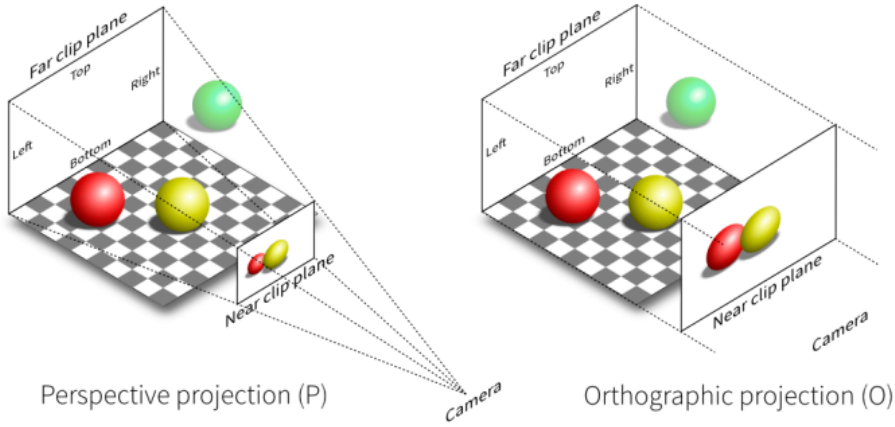

Figure 11: Image source

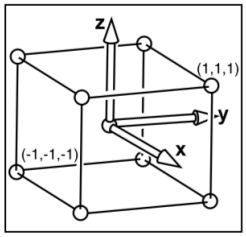

Figure 12: The canonical view volume

up, but will allow arbitrary rectangles to be viewed. Under these constraints, the view volume is an **axis-aligned box**, and well name the coordinates of its sides so that the view volume is $[l, r] \times [b, t] \times [f, n]$ shown in Figure 13.

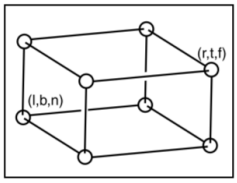

Figure 13: The orthographic view volume

$$l \equiv \text{left plane}, \qquad r \equiv \text{right plane}$$
$$b \equiv \text{bottom plane}, \qquad t \equiv \text{top plane}$$
$$n \equiv \text{near plane}, \qquad f \equiv \text{far plane}$$

That vocabulary assumes a viewer who is looking along the minus z-axis with his head pointing in the y-direction. This concept is shown in Figure 14.

As a matter of fact, the transform from orthographic view volume to the canonical view volume is a windowing transform. So we can use equation in 1.3(Windowing Transformaion).

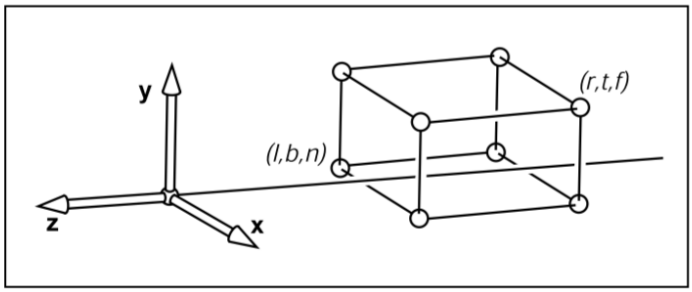

Figure 14: The orthographic view volume is along the negative z-axis, so n > f

$$\mathbf{M}_{orth} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

n > f is not intuitive, so in some API like OpenGL, the canonical volue is left-hand axis, as shown in Figure 15

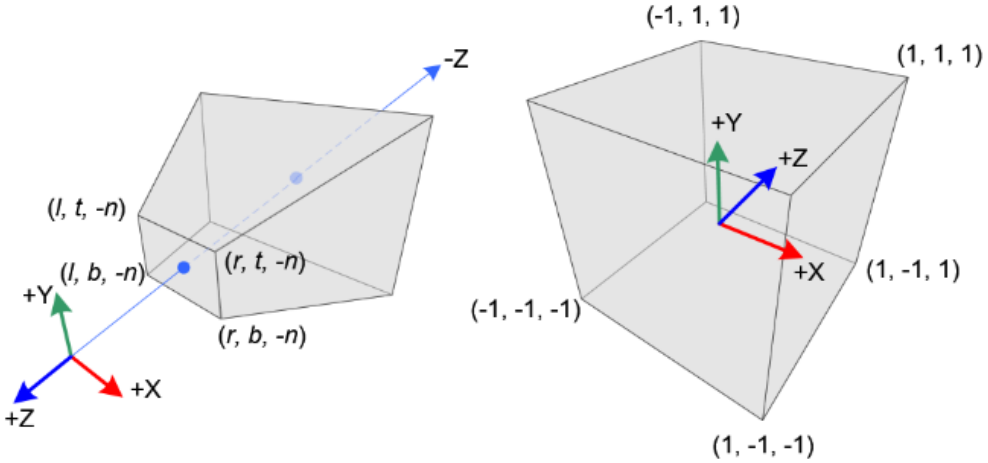

Figure 15: Perspective Frustum and Canoical View Volume(NDC) in OpenGL

3.2.2 Perspective projection

In homogeneous coordiantes

- (x,y,z,1), (kx,ky,kz,k!=0), $(\mathbf{zx}, \mathbf{zy}, \mathbf{z^2}, \mathbf{z! = 0})$ all represent the same point (x,y,z) in 3D

How to do perspective projection

1. First "squish" the frustum into a cuboid $(n- > n, f- > f)$ $(\mathbf{M_{persp-ortho}})$
2. Do orthographic projection ($\mathbf{M_{ortho}}$, already known!)

There are three rules when we "squish" the frustum:

1. Central point on the near plane will not change
2. Any point on the near plane will not change
3. Any point'z on the far plane will not change

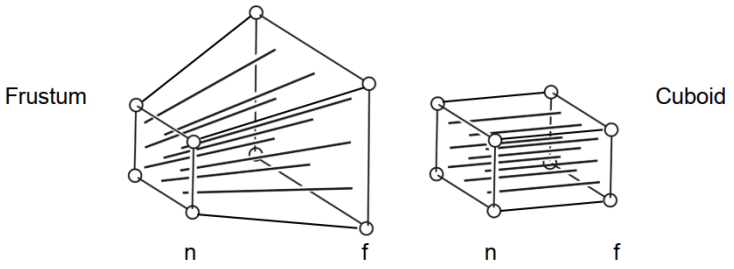

Figure 16: "squish the frustum"

We denote the transformed points by $\mathbf{M}_{\mathbf{persp-ortho}}$ as (x', y', z') and the origin points as (x, y, z) From the

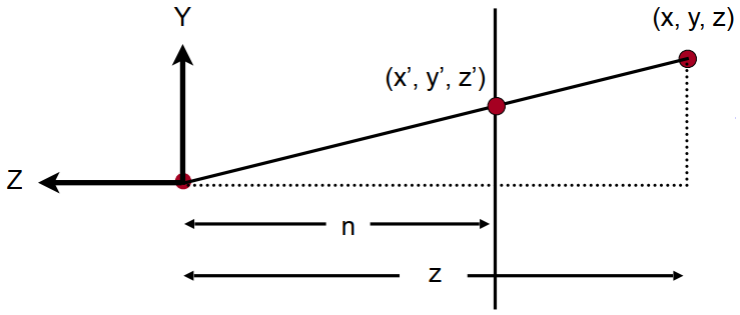

Figure 17: similar triangle

similar triangle, we can easily got two equation below.

$$y' = \frac{n}{z}y$$
$$x' = \frac{n}{z}x$$

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} nx/z \\ ny/z \\ \text{unknown} \\ 1 \end{pmatrix} \begin{matrix} \text{mult.} \\ \text{by } z \\ = \end{matrix} \begin{pmatrix} nx \\ ny \\ \text{still unknown} \\ z \end{pmatrix}$$

According to derivation above, we can deduce the part items in $\mathbf{M}_{\mathbf{persp-ortho}}$.

$$\mathbf{M}_{\mathbf{persp-ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} \begin{matrix} \text{mult.} \\ \text{by } n \\ = \end{matrix} \begin{pmatrix} nx \\ ny \\ n^2 \\ n \end{pmatrix}$$

$$\mathbf{M_{persp-ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} \begin{matrix} \text{mult.} \\ \text{by } f \\ = \end{matrix} \begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix}$$

$$\begin{cases} Af + B = f^2 \\ An + b = n^2 \end{cases} \Rightarrow \begin{cases} A = f + n \\ B = -nf \end{cases}$$

At last, we finally achieve the construction of $\mathbf{M_{persp-ortho}}$.

$$\mathbf{M_{persp-ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Note: there is defference of our intuition about the middle point's z in frustum when "squish" the frustum. e.g. $P_m = (x, y, \frac{n+f}{2})$ After $\mathbf{M_{persp-ortho}}$,

$$P'_m = \begin{pmatrix} nx \\ ny \\ \frac{n^2+f^2}{2} \\ \frac{n+f}{2} \end{pmatrix} \equiv \begin{pmatrix} nx \cdot \frac{2}{n+f} \\ ny \cdot \frac{2}{n+f} \\ \frac{n^2+f^2}{n+f} \\ 1 \end{pmatrix}$$

$\frac{n^2+f^2}{n+f} > \frac{n+f}{2}$, so the the absolute value of middle point'z will be greater after "squish".

$$\begin{aligned} \mathbf{M_{persp}} &= \mathbf{M_{ortho}}\mathbf{M_{persp-ortho}} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -\frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{aligned}$$

Sometimes, we use vetical field-of-view θ and aspect ratio(width / height). (assume symmetry l = -r, b = -t). Some APIS is as follows:

Matrix4f get_projection_matrix(float eye_fov, float aspect_ratio, float zNear, float zFar)

$$l = -r, b = -t$$
$$aspect\ ratio = \frac{r}{t}$$
$$tan(\frac{\theta}{2}) = \frac{t}{|n|}$$

In this case,

$$\mathbf{M_{persp}} = \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -\frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} -\frac{atan(\theta/2)}{ratio} & 0 & 0 & 0 \\ 0 & -atan(\theta/2) & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & -\frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

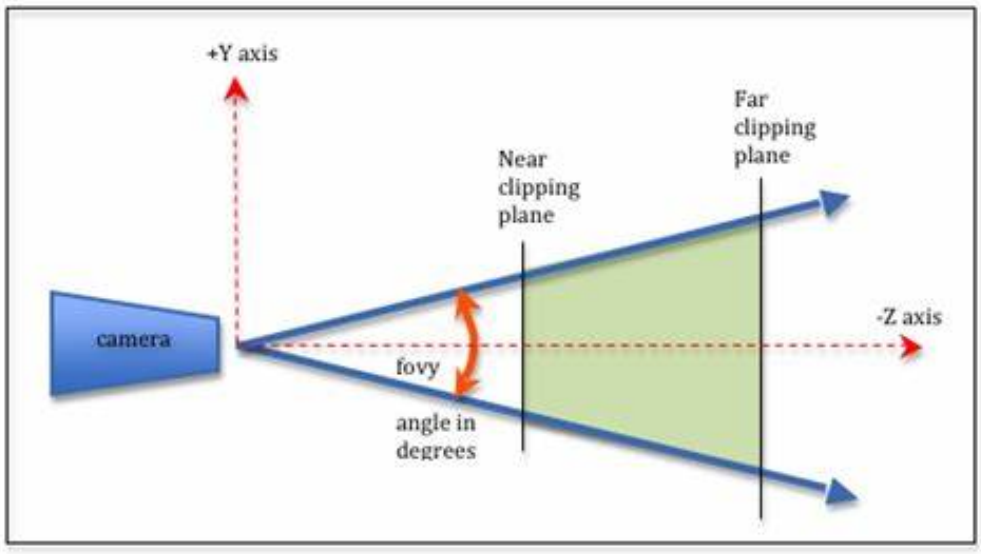

Figure 18: Vetical field-of-view

3.3 Viewport Transformation

The canonical view volume is the cube containging all 3D points whose Cartesian coordiantse are between -1 and +1, i.e. $(x, y, z) \in [-1, 1]^3$.

With the purpose of display, we need to project 3D points to 2D pixels in screen. we define the screen width and height. we temporally drop z axis. Because a points distance along the projection direction doesnt affect where that point projects in the image.

we need to map the square $[-1, 1]^2$ to the rectangle $[0, width] \times [0, height]$

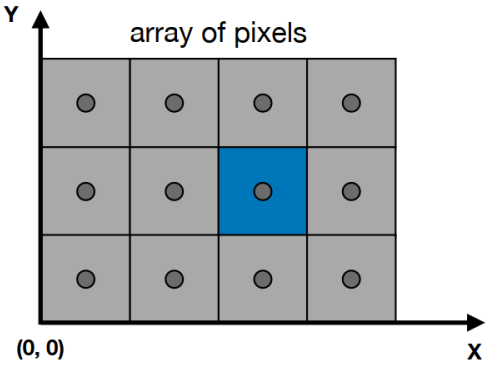

Figure 19: Window Coordinate System

$$\mathbf{M_{viewport}} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

4 Render Pipeline

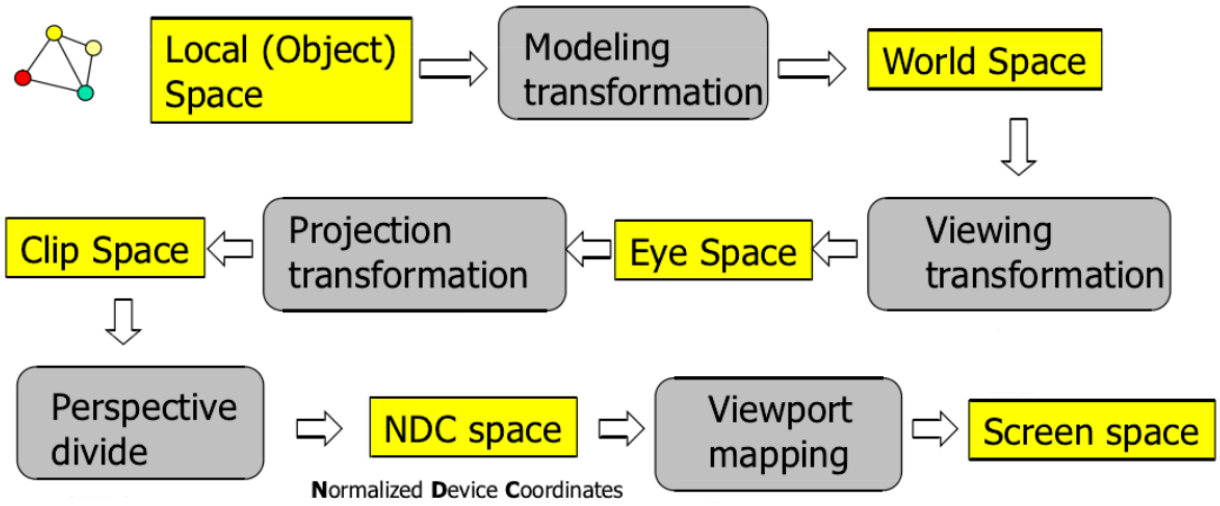

Figure 20: OpenGL Render Pipeline

Perspective divide switches homogeneous to nonhomogeneous(x/w,y/w,z/w)

Note: Screen Coordinate and Window Coordinate are different, as shown in Figure 22

In OpenGL, we can use *void glViewport(GLint Sx, GLint Sy, GLsizei width, GLsizei height)* and *glDepthRangef(GLclampf ns, GLclampf fs)*

If a point (x_{ndc}, y_{ndc}) in NDC space whose corresponding window coordinates is (x_s, y_s), the transformation is as follows:

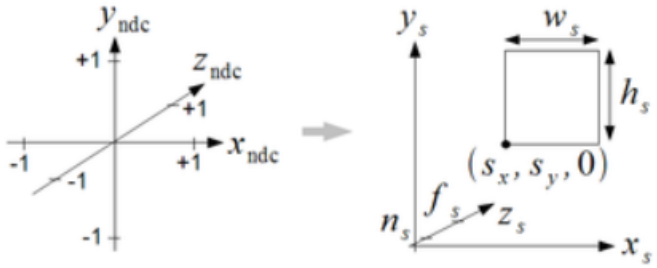

Figure 21: NDC to Screen Space(not screen coordiantes)

$$x_s = \frac{width}{2} * x_{ndc} + Sx + \frac{width}{2} * x_{ndc} \tag{1}$$

$$y_s = \frac{height}{2} * y_{ndc} + Sy + \frac{height}{2} * y_{ndc} \tag{2}$$

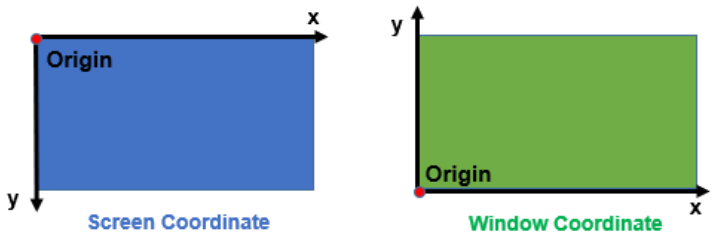

Figure 22: Comparsion Visualization

5 Reference

- *Fundamentals Of Computer Graphiccs – Peter Shirley, Steve Marshner 3rd version*
- MVP Transformaion in OpenGL.
- Modern opengl
- OpenGL Projection Matrix
- MIT Linear Algebra Course Notes
- OpenGL Camera
- Learnopengl
- Homogeneous Space Clipping
- Viewport Transformation