


Tutorial 4

CS3241 Computer Graphics (AY23/24)

September 20, 2023

Wong Pei Xian

 e0389023@u.nus.edu

Recap

Lecture 4:

- Matrices (translation, rotation, scale)
- Matrix stacks (Current Transformation Matrix or CTM)

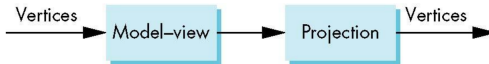
Lecture 5:

- View transformation
- Projection
- `GL_MODELVIEW` and `GL_PROJECTION` in context of CTM

Recap

CTMs in OpenGL

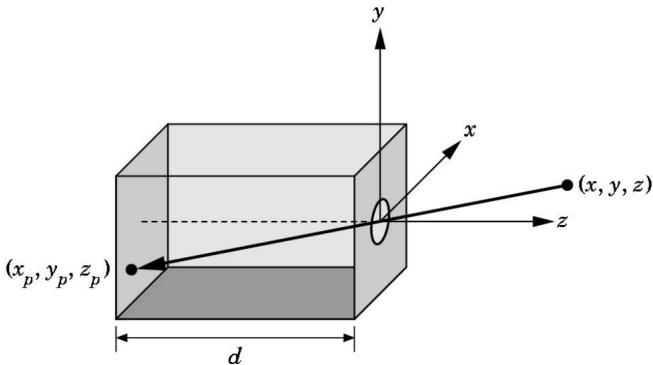
- OpenGL has a **model-view** and a **projection** matrix in the pipeline



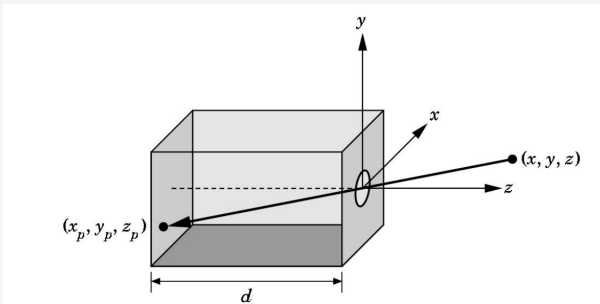
- Each has a CTM and can be manipulated by first setting the correct matrix mode

Question 1a

Referring to Lecture 1 Slide 31. If an imaginary image plane is d unit distance in front of the pinhole camera, what are the coordinates of the **projection** (on the imaginary image plane) of the 3D point (x, y, z) ?



Question 1a



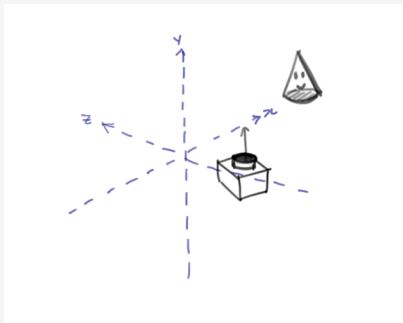
$$\frac{x}{x'} = \frac{y}{y'} = \frac{z}{z'} \text{ and by definition } z' = d$$
$$x' = \frac{dx}{z} \quad y' = \frac{dy}{z} \quad z' = d$$

To project simply scale the ray to hit the surface.

Question 1b

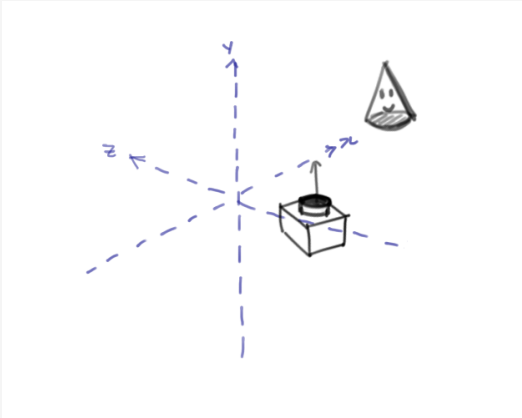
In the above setup, the camera's center of projection is conveniently located at the origin of the "world" coordinate frame, and pointed in the z direction. If the camera's center of projection is not located at the origin, and the camera is pointed in an arbitrary direction, the calculation of the projection becomes very messy. How would you make it less messy?

Explanation

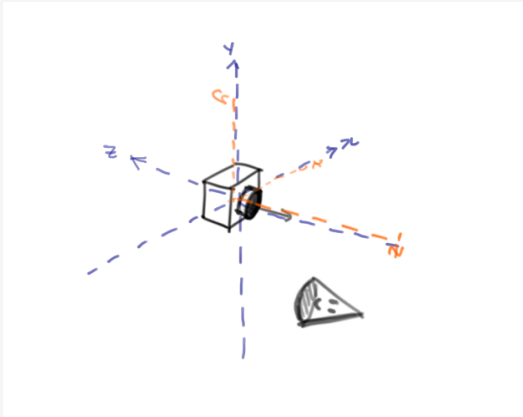


- **Camera axes** and **world axes** are not not equivalent
- The cone is represented in **world space coordinates**.
- We undo the transformation by **translating everything by the camera's distance from the origin**,
- and then **rotating everything by the camera's rotation**.
- And now the **camera axes** and **world axes** are aligned.

Q1b



Q1b



Question 1b

Reorient the world **with respect to the camera's rotation and translation.**

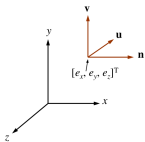
Visualization: <https://imgur.com/a/sXuYgaM>

Question 2

Why do we want to perform view transformation?

- Suppose the camera has been moved to the location $[e_x, e_y, e_z]^T$, and its x_c, y_c, z_c axes are the unit vectors $\mathbf{u}, \mathbf{v}, \mathbf{n}$, respectively, then

$$\mathbf{M}_{\text{view}} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- Note that $[e_x, e_y, e_z]^T$ and $\mathbf{u}, \mathbf{v}, \mathbf{n}$ are all specified w.r.t. to the **world frame**

$$M_{\text{view}} = RT$$

Benefits

1. Can reorient camera position within the world without manually changing all vertices' coordinates.
2. Can perform **perspective projection**
 - o Why do we need perspective projection?

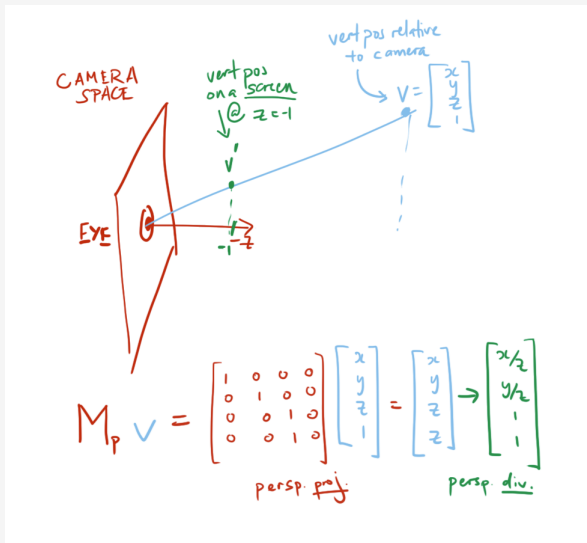
Perspective projection matrix (simplified)

$$M_p v = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

Here M_p is a basic projection matrix that simply projects any light ray from the pinhole (eye) through the point onto a virtual plane, setting up the homogenous coordinate such that **perspective division scales the resulting image into what would be captured at distance 1 from eye.**

The actual perspective division matrix is more complex as it must account for the transformation to NDC space.

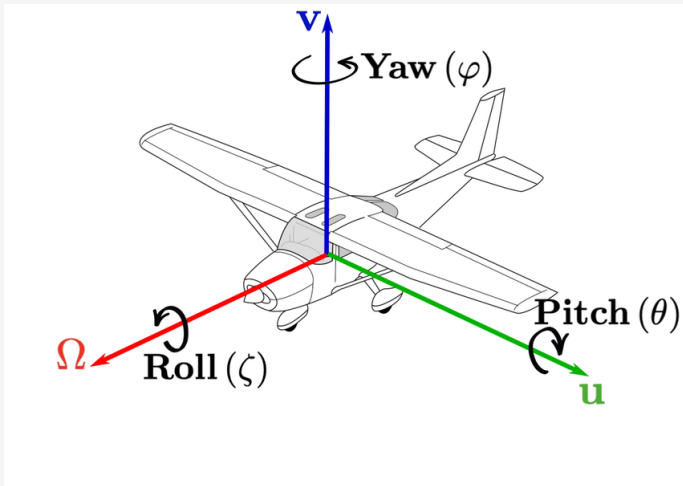
Perspective projection matrix (simplified)



Question 3

Explain the purpose of the “up-vector” provided to the `gluLookAt()` function.

To prevent the camera from 'rolling'



By defining the "up-vector" we establish a vertical plane for the y and z axes of the camera coordinates.

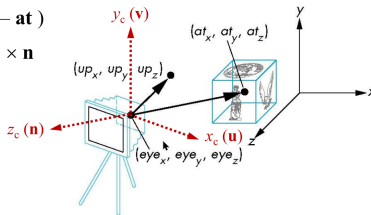
Question 3b

Why does the “up-vector” not need to be perpendicular to the view direction?

We can derive our 3 axes as such:

■ The vectors \mathbf{u} , \mathbf{v} , \mathbf{n} can be derived as follows

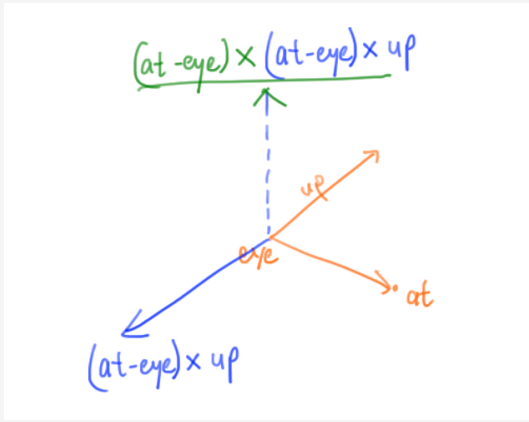
- $\mathbf{n} = \text{normalize}(\text{eye} - \text{at})$
- $\mathbf{u} = \text{normalize}(\mathbf{up}) \times \mathbf{n}$
- $\mathbf{v} = \mathbf{n} \times \mathbf{u}$



18

As long as the up-vector is **not parallel** to the view direction and **is not zero vector**, it already **uniquely defines the y-axis** of the camera.

We can derive our 3 axes as such:



Question 4

Replace the following `gluLookAt()` function call with one or more calls to `glRotated()` and `glTranslated()`.

When using `glRotated()`, you are allowed to rotate about the x-axis, y-axis and z-axis only.

```
gluLookAt( ex, ey, ez, ex, ey, ez+1, 0, -1, 0 );
```

Analysis of gluLookAt

$$\text{eye} = (e_x, e_y, e_z)$$

$$\text{at} = (e_x, e_y, e_z + 1)$$

$$\text{eye} - \text{at} = (0, 0, -1)$$

$$\text{up} = (0, -1, 0)$$

z axis: $n = \text{eye} - \text{at} = (0, 0, -1)$ (camera looks in the -*z* direction!)

x axis: $u = \text{norm}(\text{up}) \times \text{norm}(n) = (1, 0, 0)$

y axis: $v = \text{norm}(n) \times \text{norm}(u) = (0, 0, -1) \times (1, 0, 0) = (0, -1, 0)$

camera position = (e_x, e_y, e_z)

▫ It is made up of a **translation first, then a rotation**

▪ $M_{\text{view}} = \mathbf{R} \mathbf{T}$

- The translation \mathbf{T} moves the camera position back to the world origin
- The rotation \mathbf{R} rotates the axes of the camera frame to coincide with the corresponding axes of the world frame

1. Translate the world towards camera: `glTranslate(-ex, -ey, -ez);`
2. Rotate the world to align with camera:
 - Notice that the camera z and y coordinates are flipped
 $z_c = n = -(0, 0, 1)$ and $y_c = v = -(0, 1, 0)$
 - `glRotated(180, 1, 0, 0)`
 - `glRotated(180, 0, 1, 0); glRotated(180, 0, 0, 1);`

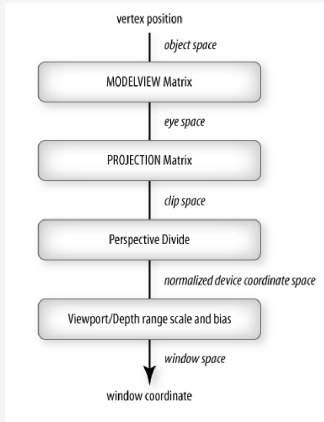
Question 5

A vertex, whose camera coordinates are (4, 6, -6), is being projected using the following OpenGL orthographic projection:

```
glOrtho( -10, 10, -10, 10, 0, 8 );
```

What will be the vertex's Normalized Device Coordinates (NDC)?

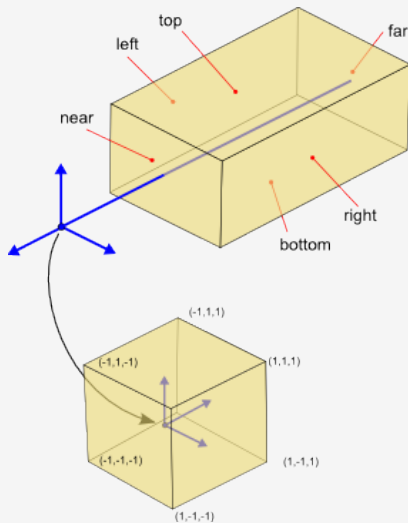
Coordinates through pipeline



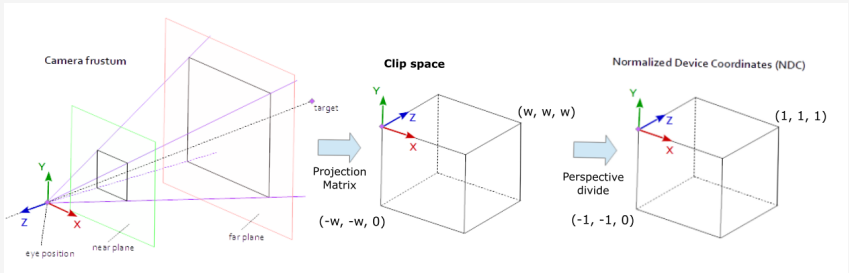
Camera coordinates to NDC space:

1. If vertex is within the clipping region, it is mapped in NDC space
2. NDC space is scaled to a $2 \times 2 \times 2$ **volume**

Coordinate spaces: Orthographic Projection



Coordinate spaces: Perspective Projection



Orthographic projection

- The mapping can be found by
 - First, **translating** the view volume to the origin
 - Then, **scaling** the view volume to the size of the canonical view volume

$$\mathbf{M}_{\text{ortho}} = \mathbf{S} \left(\frac{2}{\text{right} - \text{left}}, \frac{2}{\text{top} - \text{bottom}}, \frac{2}{\text{near} - \text{far}} \right) \cdot \mathbf{T} \left(\frac{-(\text{right} + \text{left})}{2}, \frac{-(\text{top} + \text{bottom})}{2}, \frac{(\text{far} + \text{near})}{2} \right)$$

- Note that $z = -\text{near}$ is mapped to $z = -1$,
and $z = -\text{far}$ to $z = +1$

Orthographic projection

```
glOrtho( l, r, b, t, n, f );
```

$$\mathbf{T} = T\left(\frac{-(10 - 10)}{2}, \frac{-(10 - 10)}{2}, \frac{8 + 0}{2}\right)$$

$$= T(0, 0, 4)$$

$$\mathbf{S} = S\left(\frac{2}{10 - (-10)}, \frac{2}{10 - (-10)}, \frac{2}{0 - 8}\right)$$

$$= S(0.1, 0.1, -0.25)$$

$$\mathbf{Mv} = \mathbf{ST}(4, 6, -6)$$

$$= \mathbf{S}(4, 6, -2)$$

$$= (0.4, 0.6, 0.5)$$

Perspective Projection Matrix (full)

Recall our basic perspective projection matrix

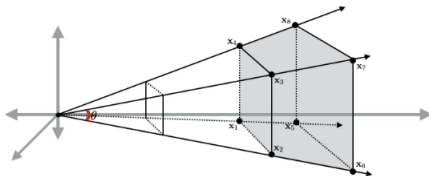
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix}$$



$$\begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

objects shrink
in distance

Full perspective matrix takes geometry of view frustum into account:



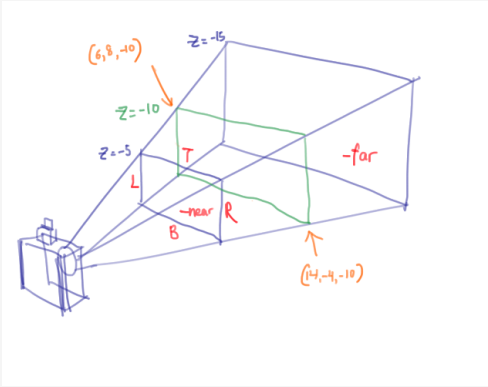
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

l = left b = bottom n = near
 r = right t = top f = far

Question 6

A rectangle has vertices A: (6, -4, -10), B: (14, -4, -10), C: (14, 8, -10), D: (6, 8, -10) in the camera space.

Write a `glFrustum` function call to set up a view frustum that will maximize the image size of the rectangle, and the entire rectangle must appear in the image. The near and far plane distances should be set as 5 and 15 respectively.



```
glFrustum( 3, 7, -2, 4, 5, 15);
```

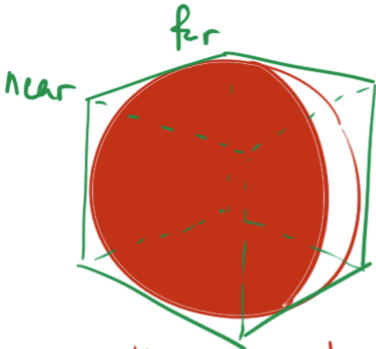
Question 7

A viewpoint at (v_x, v_y, v_z) is looking at the center (c_x, c_y, c_z) of a sphere of radius R . Complete the following OpenGL program to set up a view transformation and an orthographic projection so that the entire sphere appears as big as possible in a square viewport.

```
double PI = 3.141593;
double R = ...; // radius of sphere.
double cx, cy, cz; // center of sphere.
double vx, vy, vz; // viewpoint position.
...
double D = Distance( cx, cy, cz, vx, vy, vz );

// Write your code below.
```


Visualization



1. the entire sphere in NDC
2. the entire front face is nearest to near plane

Code

```
glMatrixMode(GL_PROJECTION); // Camera coordinates
glLoadIdentity(); // Always reset the matrix
// we are already looking at the camera's center,
// with the top/bottom/left/right points of the circle touching the clipping boundaries
// near = front most point on z-axis, far = furthest point on z-axis
glOrtho(-R, R, -R, R, D-R, D+R);

glMatrixMode(GL_MODELVIEW); // World Coordinates
glLoadIdentity(); // Always reset the matrix
// eye, at, up
gluLookAt(vx, vy, vz, cx, cy, cz, 0, 1, 0);
```

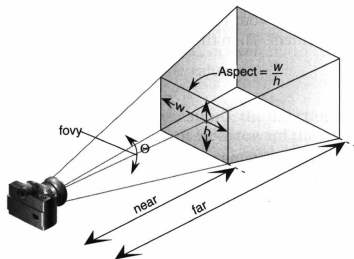
Question 8

Re-implement the `gluPerspective()` function by using the `glFrustum()` function. You can make use of the tangent function `tan()`, which takes an angle parameter (in radians).

```
void gluPerspective(  
    double fovy, double aspect,  
    double near, double far) {  
    const double PI = 3.141592;  
}
```

Question 8

```
gluPerspective( fovy, aspect, near, far );
```



$$\text{left} = -\frac{h}{2}, \text{right} = \frac{h}{2}, \text{bottom} = -\frac{w}{2}, \text{top} = \frac{w}{2}.$$

Let aspect ratio be $a = \frac{w}{h}$.

Let fovy be θ .

By trigonometry, $h = 2 \tan\left(\frac{\theta}{2}\right) \times \text{near}$.

By definition, $w = ah$.

```
void gluPerspective( double fovy, double aspect,
                    double near, double far )
{
    // Note that fovy is in degrees.
    const double PI = 3.141592;

    double h2 = near * tan( fovy/2.0 * PI / 180.0 );
    double w2 = aspect * h2;
    glFrustum( -w2, w2, -h2, h2, near, far );
}
```

Thanks! Get the slides here after the tutorial.



<https://trxe.github.io/cs3241-notes>