

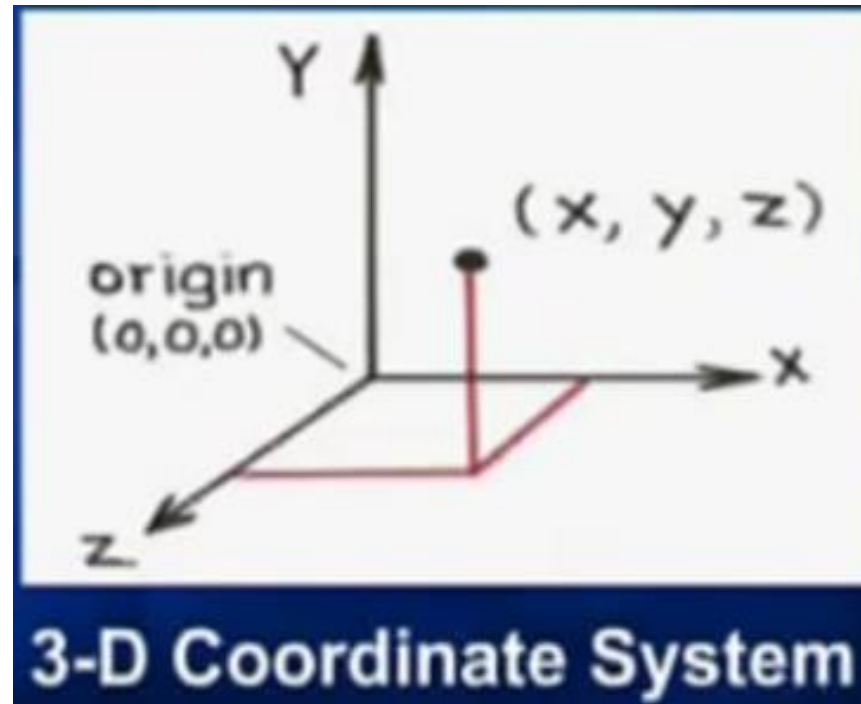
Chapter Five

3-D Graphics

3-D Geometry

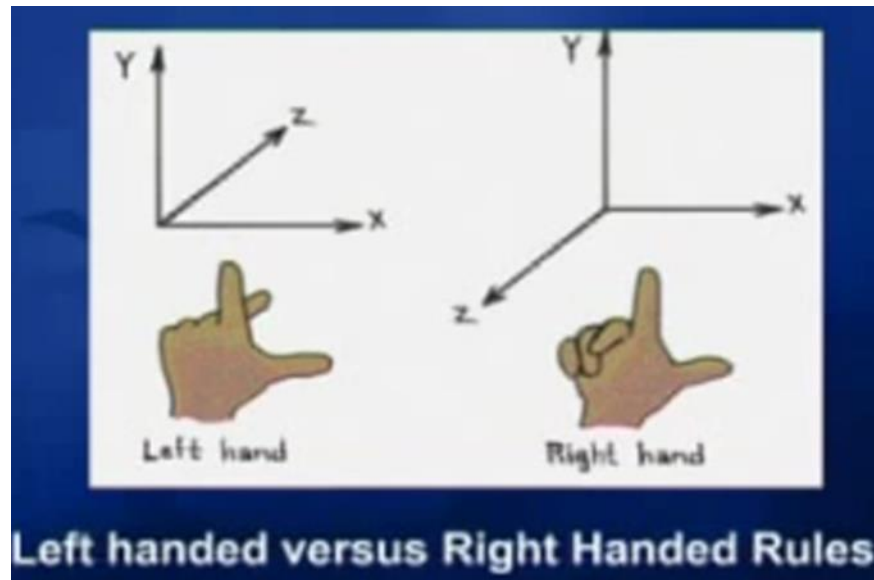
- **3D Cartesian coordinate system**

- Direction and magnitude along three axes, with reference to an origin
- Location are defined by x, y, z coordinate
- We can define points, segments, lines rays, curves, polygons (any other planar geometry) and cubes, cones, spheres, etc. (volume in space)
- Can have multiple origins (frame of reference) and transform coordinates among them



Rules

- These rules determine orientation of axes and direction of rotations
 - Thumb = pos x
 - Index up = pos y
 - Middle out = pos z
 - Most world and objects axes tend to be right handed
 - Left hand axes often are used for cameras



- Rotation

- Grasp axis with right hand with thumb oriented in positive direction, fingers will then curl in direction of positive rotation for that axis.



- Right handed Cartesian coordinate system describes the relationship of the x, y, z in the following manner
 - X is positive to the right of the origin and negative to the left
 - Y is positive to the above of the origin and negative to the below
 - Z is positive behind the origin and negative beyond

- OpenGL is a powerful 3D programming tool used to draw complex three-dimensional scenes from simple primitives.
- To draw a 3D geometry object we need to setup the following:
- Setting up the main() Function

```
int main(int argc, char* argv){  
    // Initialize GLUT and process user parameters  
    glutInit(&argc,argv);  
    // Request double buffered true color window with Z-buffer  
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

- Double buffering is a technique used in graphics programs to eliminate a problem that arises due to how images are drawn to the screen.
- Each time we redraw the scene, the display must first be erased then the new information will be drawn.
- Without double buffering you will observe a flickering effect as the screen is erased and redrawn repeatedly.

- This problem is fixed by adding a second buffer to draw to. With this method, an image is drawn to the first buffer and that buffer is shown to you.
- The next frame will be drawn to the second buffer and when that's done, the two buffers will switch places.
- We will immediately see the second buffer, but, hidden from us, the first buffer is being erased and redrawn with the third frame which will be swapped in when finished.

- We also want to enable the RGB color system in our window.
- Z-buffering is how we get the 3D effects that we want.
- OpenGL uses a three dimensional coordinate system with x, y and z axes.
- To give the effect that an object is closer to you its position on the z axis is increased, however, to make it appear further away its position on the z axis is decreased.

Enable Depth Test

- OpenGL is a strict language in that it does not assume any special features are enabled.
- For our program to properly display in 3-dimensions using the Z-buffer that we looked at earlier, we need to enable depth-test.

```
// Enable Z-buffer depth test  
glEnable(GL_DEPTH_TEST);
```

3-D Transformation

- A three-dimensional object has a three-dimensional geometry, and therefore, it requires a three-dimensional coordinate transformation.
- A right-handed coordinate system is used to carry out a 3-D transformation.
- The scaling and translation transformations are essentially the same as two-dimensional transformations.
- However, the points matrix will have a non-zero 3rd column.
- Additionally, the transformation matrices contain some non-zero values in the third row and third column, as shown below.

- A general scaling transformation matrix is given as:

$$[T_s] = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Where, s_x, s_y, s_z are scale factors along x, y, and z-axes, respectively.
- Translation Transformation matrix:

$$[T_t] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{pmatrix}$$

- The two-dimensional rotation transformation is in reality a special case of a three-dimensional rotation about the z-axis.
- We will denote it by $[Trz]$, where, the second subscript z indicates rotation about the z-axis.
- Similarly, rotation about the x and y-axes are denoted as $[Trx]$, and $[Try]$, respectively. The transformation matrices are given below.
- Matrix for representing three-dimensional rotations about the Z axis

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix for representing three-dimensional rotations about the X axis

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix for representing three-dimensional rotations about the Y axis

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```

#include<iostream.h>
#include <GL/glut.h>
void display();
void specialKeys();
// Global Variables
double rotate_y=0;
double rotate_x=0;
int main(int argc, char* argv[]){

// Initialize GLUT and process user parameters
glutInit(&argc,argv);

// Request double buffered true color window with Z-buffer
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
// Create window
glutCreateWindow("my Cube");
// Enable Z-buffer depth test
glEnable(GL_DEPTH_TEST);

glutDisplayFunc(display);
glutMainLoop();
return 0;
}

void display(){
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glRotatef( 10, 1.0, 0.0, 0.0 );
glRotatef(10, 0.0, 1.0, 0.0 );
glRotatef(10, 0.0, 0.0, 1.0 );
glBegin(GL_POLYGON);
glColor3f( 1.0, 0.0, 0.0 );    glVertex3f( 0.5, -0.5, -0.5 );    // P1
is red
glColor3f( 0.0, 1.0, 0.0 );    glVertex3f( 0.5, 0.5, -0.5 );    // P2
is green
glColor3f( 0.0, 0.0, 1.0 );    glVertex3f( -0.5, 0.5, -0.5 );    // P3
is blue
glColor3f( 1.0, 0.0, 1.0 );    glVertex3f( -0.5, -0.5, -0.5 );    // P4
is purple

```

```
glEnd();  
glBegin(GL_POLYGON);  
glColor3f( 1.0, 1.0, 1.0 );  
glVertex3f( 0.5, -0.5, 0.5 );  
glVertex3f( 0.5, 0.5, 0.5 );  
glVertex3f( -0.5, 0.5, 0.5 );  
glVertex3f( -0.5, -0.5, 0.5 );  
glEnd();
```

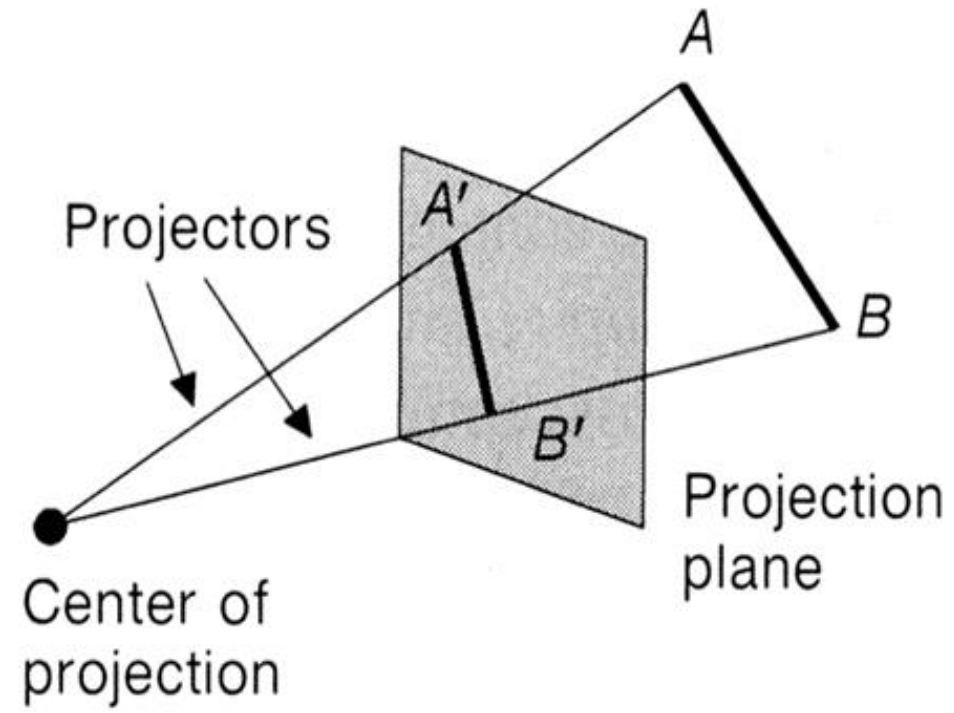
```
// Purple side - RIGHT  
glBegin(GL_POLYGON);  
glColor3f( 1.0, 0.0, 1.0 );  
glVertex3f( 0.5, -0.5, -0.5 );  
glVertex3f( 0.5, 0.5, -0.5 );  
glVertex3f( 0.5, 0.5, 0.5 );
```

```
glVertex3f( 0.5, -0.5, 0.5 );  
glEnd();  
// Green side - LEFT  
glBegin(GL_POLYGON);  
glColor3f( 0.0, 1.0, 0.0 );  
glVertex3f( -0.5, -0.5, 0.5 );  
glVertex3f( -0.5, 0.5, 0.5 );  
glVertex3f( -0.5, 0.5, -0.5 );  
glVertex3f( -0.5, -0.5, -0.5 );  
glEnd();  
// Blue side - TOP  
glBegin(GL_POLYGON);  
glColor3f( 0.0, 0.0, 1.0 );  
glVertex3f( 0.5, 0.5, 0.5 );  
glVertex3f( 0.5, 0.5, -0.5 );  
glVertex3f( -0.5, 0.5, -0.5 );  
glVertex3f( -0.5, 0.5, 0.5 );  
glEnd();
```

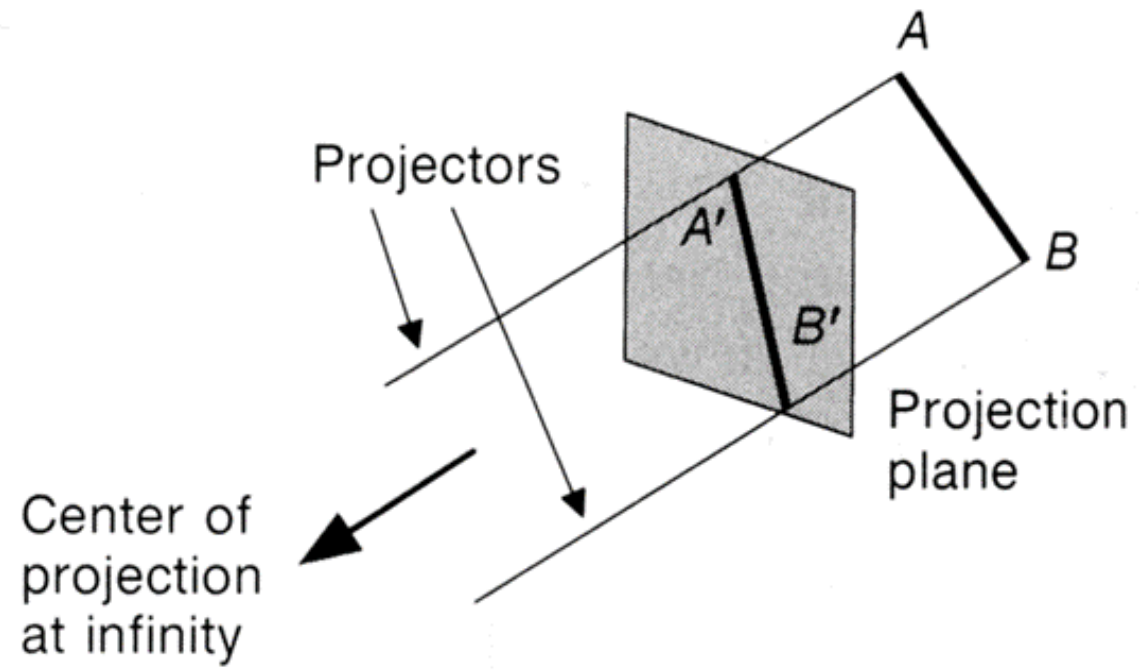
```
// Red side - BOTTOM  
glBegin(GL_POLYGON);  
glColor3f( 1.0, 0.0, 0.0 );  
glVertex3f( 0.5, -0.5, -0.5 );  
glVertex3f( 0.5, -0.5, 0.5 );  
glVertex3f( -0.5, -0.5, 0.5 );  
glVertex3f( -0.5, -0.5, -0.5 );  
glEnd();  
glFlush();  
glutSwapBuffers();  
  
}
```

3-D Display methods

- Projections use to transform 3D objects on to a 2D plane.
- There are two types of projections.
 - They are called Perspective and Parallel projections.
- In parallel projection, coordinate positions are transformed to the view plane along parallel lines.
- In perspective projection, object position are transformed to the view plane along lines that converge to a point called projection reference point (center of projection)



Perspective projection



Parallel projection

Perspective Vs Parallel

- **Perspective:**

- visual effect is similar to human visual system...
- has 'perspective foreshortening'
 - size of object varies inversely with distance from the center of projection. Projection of a distant object are smaller than the projection of objects of the same size that are closer to the projection plane.

- **Parallel:**

- It preserves relative proportion of object.
- less realistic view because of no foreshortening
- however, parallel lines remain parallel.

Orthographic projection in OpenGL

- The fundamental orthographic viewing function is `glOrtho()`, which can be used to set up the proper viewing matrix.
 - `Void glOrtho ( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`
- Sets up an orthographic projection matrix and defines a viewing volume, which is a right parallelepiped. The distances are measured from the camera, so we should have `right>left`, `top>bottom`, `far>near`
- The `glOrtho()` function changes the present matrix incrementally, so we usually start by loading the identity matrix. Hence the code for a cubic clipping volume of side length 2, centered at the origin is ;
 - `glMatrixMode(GL_PROJECTION);`
 - `glLoadIdentity();`
 - `glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);`

Example

```
#include <GL/freeglut.h>
#include<iostream>
using namespace std;
void cube();
void polygon(int , int , int ,int );
float roty=0;
float rotx=0;
float rotz=0;
GLfloat vertices[][3]={{-1.0,-1.0,1.0},{-
1.0,1.0,1.0},{1.0,1.0,1.0},{1.0,-1.0,1.0},{-1.0,-1.0,-1.0},{-1.0,1.0,-
1.0},{1.0,1.0,-1.0},{1.0,-1.0,-1.0}};
GLfloat colors[][3]={{1,0,0},{0,1,1},{1,1,0},{0,1,0},{0,0,1},{1,0,1}};
void keypress(int key, int x, int y){
    if(key==GLUT_KEY_UP){
        roty+=10;
    }
    if(key==GLUT_KEY_RIGHT){
        rotx+=5;
    }
    if(key==GLUT_KEY_LEFT){
        rotz+=5;
    }
    glutPostRedisplay();
}
```

```
void myinit()
{
    glClearColor(0.0,0.0,0.0,0.0);
}

void display()
{
    glClear( GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(1.0,1.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
    cube();
    glutSwapBuffers();
}

void polygon(int a, int b, int c,int d){
    glColor3fv(colors[a]);
    glBegin(GL_POLYGON);
        glVertex3fv(vertices[a]);
        glVertex3fv(vertices[b]);
        glVertex3fv(vertices[c]);
        glVertex3fv(vertices[d]);
    glEnd();
}
```

```
void cube()
{
    glRotatef(rotx,1,0,0);
        glRotatef(-roty,0,1,0);
        glRotatef(rotz,0,0,1);
    polygon(0,3,2,1);
        polygon(2,3,7,6);
        polygon(3,0,4,7);
        polygon(1,2,6,5);
        polygon(4,5,6,7);
        polygon(5,4,0,1);
}
```

```
void reshape(int w, int h){
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-5.0,5.0,-5.0,5.0,-10.0,10.0);
}
```

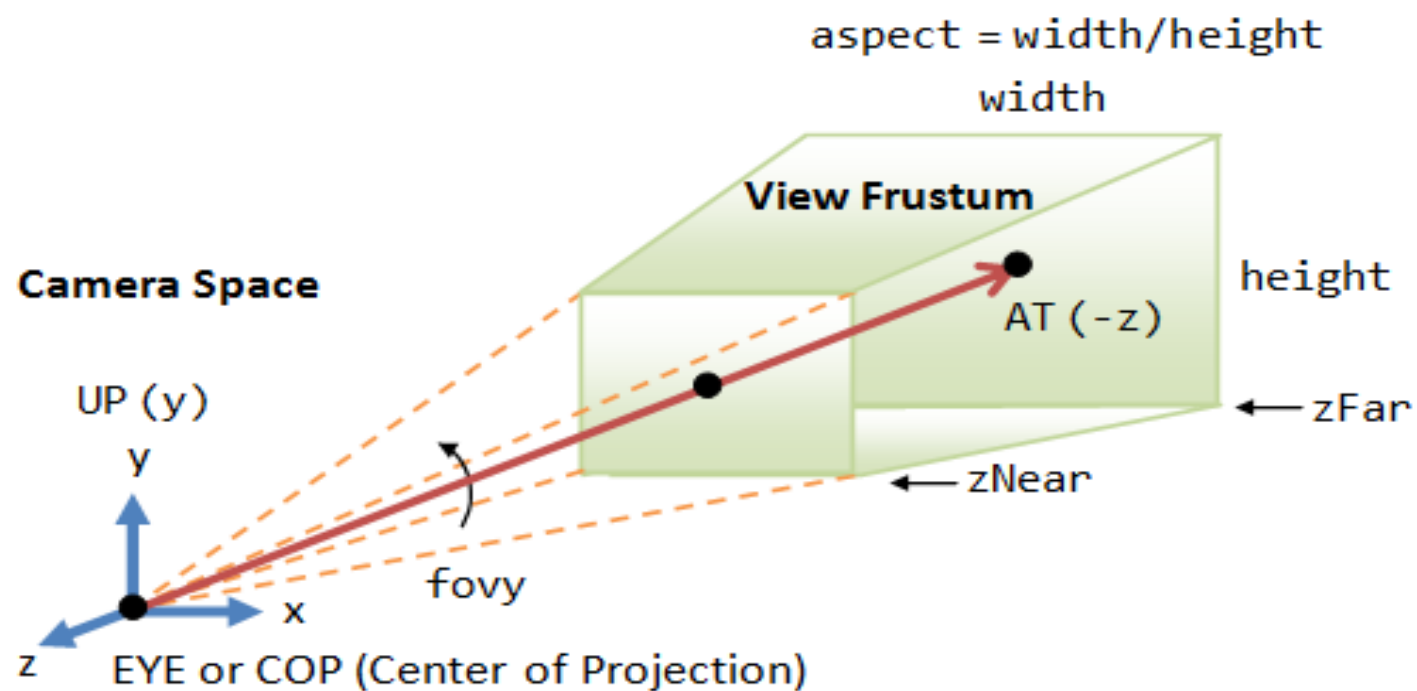
```
int main(int argc, char** argv) {
    glutInit( &argc, argv );
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize( 500, 500 );
    glutCreateWindow( "CUBE" );
    glutReshapeFunc(reshape);
    glutDisplayFunc( display );
    glutSpecialFunc(keypress);
    myinit();
    glutMainLoop();
    return 0;
}
```

Projection Transform - Perspective Projection

- Once the camera is positioned and oriented, we need to decide what it can see (analogous to choosing the camera's field of view by adjusting the focus length and zoom factor), and how the objects are projected onto the screen.
- This is done by selecting a projection mode (perspective or orthographic) and specifying a viewing volume or *clipping volume*.
- Objects outside the clipping volume are clipped out of the scene and cannot be seen.

View Frustum in Perspective View

- The camera has a limited field of view, which exhibits a *view frustum* (truncated pyramid), and is specified by four parameters: fovy, aspect, zNear and zFar.
 - Fovy: specify the total vertical angle of view in degrees.
 - Aspect: the ratio of width vs. height. For a particular z, we can get the height from the fovy, and then get the width from the aspect.
 - zNear; the near plane.
 - zFar: the far plane.
- The camera space (x_c, y_c, z_c) is renamed to the familiar (x, y, z) for convenience.

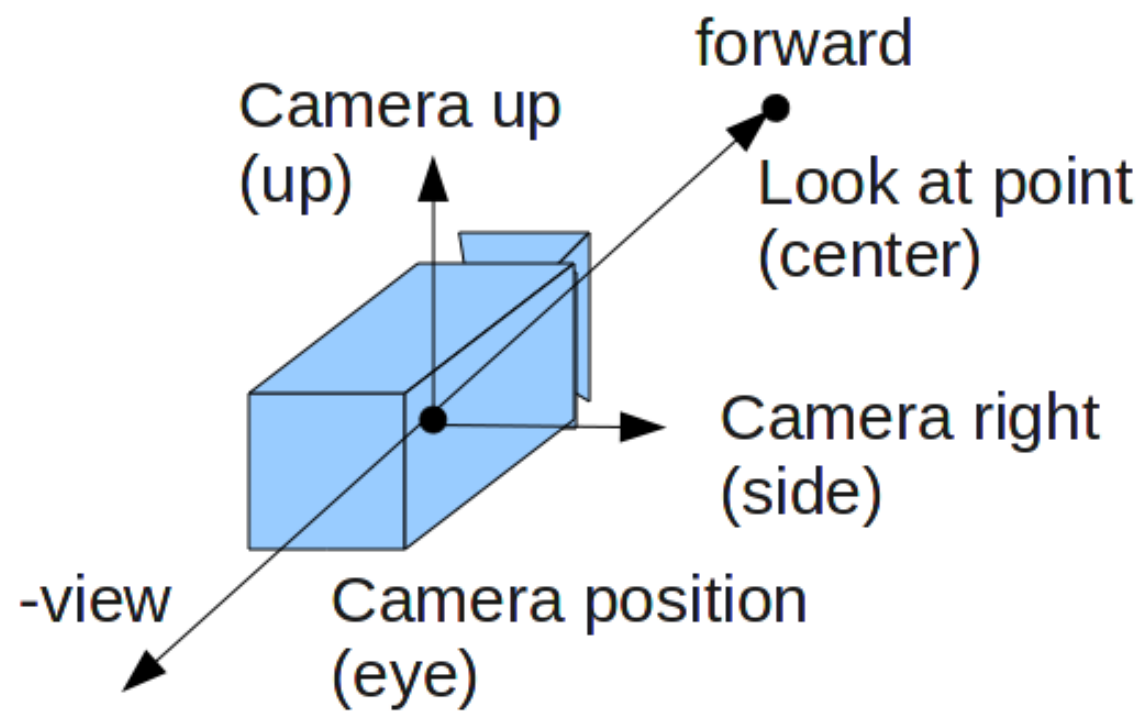


Perspective Projection: The camera's view frustum is specified via 4 view parameters: $fovy$, $aspect$, z_{Near} and z_{Far} .

- The projection with view frustum is known as *perspective projection*, where objects nearer to the COP (Center of Projection) appear larger than objects further to the COP of the same size.
- An object outside the view frustum is not visible to the camera. It does not contribute to the final image and shall be discarded to improve the performance.
- This is known as *view-frustum culling*. If an object partially overlaps with the view frustum, it will be *clipped* in the later stage.

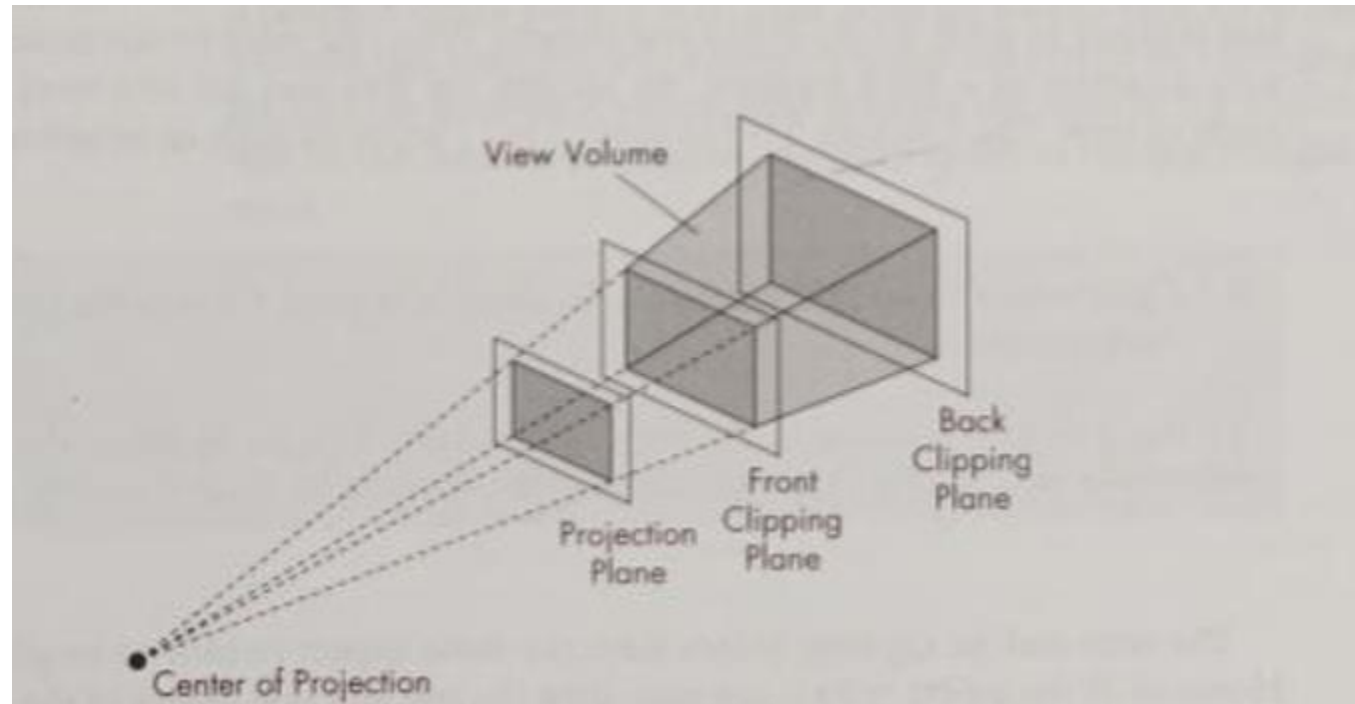
Camera - gluLookAt function

- `void gluLookAt(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ, GLdouble centerX, GLdouble centerY, GLdouble centerZ, GLdouble upX, GLdouble upY, GLdouble upZ);`
- Parameters
 - `eyeX, eyeY, eyeZ`
 - Specifies the position of the eye point.
 - `centerX, centerY, centerZ`
 - Specifies the position of the reference point.
 - `upX, upY, upZ`
 - Specifies the direction of the up vector.
- `gluLookAt` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.
- The matrix maps the reference point to the negative z axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the UP vector projected onto the viewing plane is mapped to the positive y axis so that it points upward in the viewport. The UP vector must not be parallel to the line of sight from the eye point to the reference point.



Perspective projection

- Perspective projection implements synthetic camera model of the form shown in the below figure



- The viewing volume is limited on the sides by four planes that meet at the center of projection forming an infinite viewing pyramid.
- OpenGL provides the function `glFrustum()` to create the required matrix for perspective viewing. Its parameters are the same as for `glOrtho()`.
 - `Void glFrustum(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far)`

- The problem with `glFrustum()` is that we change the distance between the near and far planes, the angles that the sides of the frustum make with respect to its center can change dramatically.
- As a user changes these distance to get the desired objects into scene, object on the sides can be lost as these angles change.
- A more natural interface is closer to what we do with a real camera, i.e `gluPerspective()`.
 - `gluPerspective(Gldouble fov, Gldouble aspect, Gldouble near, Gldouble far)`

```

#include <GL/glut.h>
void display();
float T;
void MyInit();
void Spin(){
    T=T+0.5;
    if(T>360)
        T=0;
    glutPostRedisplay();
}
int main(int argc, char* argv[]){
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB |
GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutCreateWindow("Animation");
    MyInit();

    glutDisplayFunc(display);

    glutIdleFunc(Spin);
    glutMainLoop();
}

void display(){
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(1,2,3,0,0,0,0,1,0);
    //      gluLookAt(          x, 1.0f, z,x+lx, 1.0f,  z+lz,0.0f, 1.0f,
0.0f);

    //glRotatef(T,1,1,0);
    glutSolidCube(1);
    glTranslatef(0.2,0.7,0.2);
    glutSolidTeapot(0.3);

    glFlush();
    glutSwapBuffers();
}

void MyInit(){
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1,1,-1,1,2,10);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

```