Computer Graphics Laboratory
Lab – 02
3D Object Creation and Applying Transformations
Version-1.0

Today we will discuss about creation of 3 dimensional objects and their transformations in model world. Also we will discuss about viewing coordinate system and clipping of 3D scene.
In the last tutorial we perform modelview transformations and projections of 2D scene. As 2D scene has only one plane (XY) plane, the viewer has no need to move around and thus viewer is fixed and placed in the origin of modeling world. Therefore, in 2D we need not to explicitly define the viewing coordinate system. We only defined the clipping volume. But for 3D scene, we need to define the viewing coordinate system also as the viewer can be placed in any place in modeling world.

At first, let's create a 3D object in object space. This is similar like 2D object creation. Here we going to create a pyramid like below:

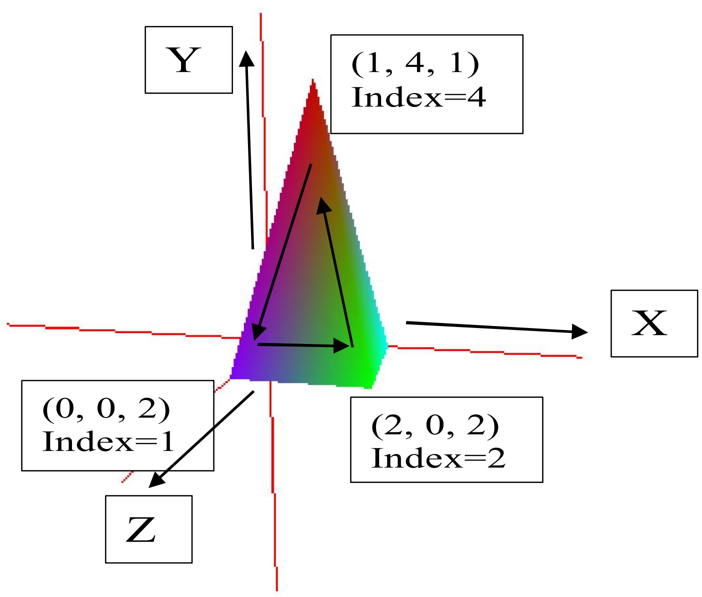


Fig: pyramid

Today, we create our desired pyramid in little different way than we did in 2D object creation. Here we see that the pyramid is a combination of four triangular plane and one rectangular plane. Creating four triangle and one rectangle requires 16 points. But here we see that we need only five points and we will reuse that points to create four triangular and one rectangular plane.
Therefore, at first we store our all required points (coordinate values of point) in an array. We also store the color for the points in another array defined in global section of our program like below:

```
static GLfloat v_pyramid[5][3] = {
                                        {0.0, 0.0, 0.0},  //point index 0
                                        {0.0, 0.0, 2.0},  //point index 1
                                        {2.0, 0.0, 2.0},  //point index 2
                                        {2.0, 0.0, 0.0},  //point index 3
                                        {1.0, 4.0, 1.0}   //point index 4
                                 };

static GLubyte p_Indices[4][3] = {
                          {4, 1, 2}, // indices for drawing the triangle plane 1
                          {4, 2, 3}, // indices for drawing the triangle plane 2
                          {4, 3, 0}, // indices for drawing the triangle plane 3
                          {4, 0, 1}  // indices for drawing the triangle plane 4
                 };
```

```
static GLubyte quadIndices[1][4] ={
                    {0, 3, 2, 1} };  // indeces for drawing the quad plane

static GLfloat colors[5][3] = {
                         {0.0, 0.0, 1.0},  //color for point index 0
                         {0.5, 0.0, 1.0},  //color for point index 1
                         {0.0, 1.0, 0.0},  //color for point index 2
                         {0.0, 1.0, 1.0},  //color for point index 3
                         {0.8, 0.0, 0.0}   //color for point index 4
                    };
```

Here above, the array p_Indices and quadIndices are important for creating triangular and rectangular planes. In p_Indices the first three values are the indices of array v_pyramid which points to three coordinate points to create the first triangular plane. But be careful to add the indices value in p_Indices array. Because there ordering is vital. **The ordering of the indices must follow the anti-clockwise rotational sequence for drawing a plane like shown in fig above.** That is the points must have to draw in either (4,1,2) or (1,2,4) or (2,4,1) order. Ordering is important for specifying the surface or plane normal which is used for applying material properties and lighting effects (we will see in next tutorial). Joining the points in anti-clockwise direction for creating a plane surface identifies that its normal will be in outward direction.

Therefore, in this tutorial we will draw a pyramid by drawing its five planes and also we calculate the normal of each plane when we draw the plane. The code segment is given below:

```
static void getNormal3p
             (GLfloat x1, GLfloat y1,GLfloat z1, GLfloat x2, GLfloat
y2,GLfloat z2, GLfloat x3, GLfloat y3,GLfloat z3){
             GLfloat Ux, Uy, Uz, Vx, Vy, Vz, Nx, Ny, Nz;

             Ux = x2-x1;
             Uy = y2-y1;
             Uz = z2-z1;

             Vx = x3-x1;
             Vy = y3-y1;
             Vz = z3-z1;

             Nx = Uy*Vz - Uz*Vy;
             Ny = Uz*Vx - Ux*Vz;
             Nz = Ux*Vy - Uy*Vx;

             glNormal3f(Nx,Ny,Nz);
}

void drawpyramid()
{
             glBegin(GL_TRIANGLES);
             for (GLint i = 0; i <4; i++) {
               getNormal3p(v_pyramid[p_Indices[i][0]][0],
v_pyramid[p_Indices[i][0]][1], v_pyramid[p_Indices[i][0]][2],
v_pyramid[p_Indices[i][1]][0], v_pyramid[p_Indices[i][1]][1],
v_pyramid[p_Indices[i][1]][2], v_pyramid[p_Indices[i][2]][0],
v_pyramid[p_Indices[i][2]][1], v_pyramid[p_Indices[i][2]][2]);

               glVertex3fv(&v_pyramid[p_Indices[i][0]][0]);
               glVertex3fv(&v_pyramid[p_Indices[i][1]][0]);
               glVertex3fv(&v_pyramid[p_Indices[i][2]][0]);
             }
  glEnd();
```

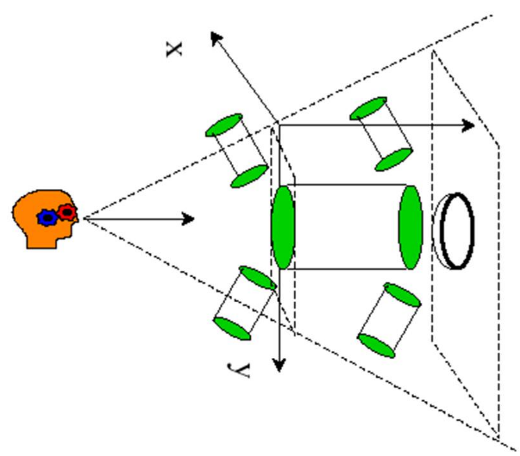
```
 glBegin(GL_QUADS);
                  for (GLint i = 0; i <1; i++) {
                    getNormal3p(v_pyramid[quadIndices[i][0]][0],
v_pyramid[quadIndices[i][0]][1], v_pyramid[quadIndices[i][0]][2],
v_pyramid[quadIndices[i][1]][0], v_pyramid[quadIndices[i][1]][1],
v_pyramid[quadIndices[i][1]][2], v_pyramid[quadIndices[i][2]][0],
v_pyramid[quadIndices[i][2]][1], v_pyramid[quadIndices[i][2]][2]);

                    glVertex3fv(&v_pyramid[quadIndices[i][0]][0]);
                    glVertex3fv(&v_pyramid[quadIndices[i][1]][0]);
                    glVertex3fv(&v_pyramid[quadIndices[i][2]][0]);
                    glVertex3fv(&v_pyramid[quadIndices[i][3]][0]);
 }
 glEnd();


}
```

Here getNormal3p function is used to create normal of each plane and normalize the created normals using glEnable(GL_NORMALIZE) in main() function which will check and if necessary renormalize all your surface normal. glVertex3fv is a vector function which accepts vector as parameter shown above. Use a common color for the pyramid by using glColor3f(1,0,0) in the display() function.

At this stage, our 3d object creation is complete. Now we need to define the viewing coordinate system as the viewer can be placed in any place in modeling world.
We will define the viewing coordinate system using viewer's eye position, look at point (where the viewer looking at) and head direction. We need to use a glu function called **gluLookAt(2,3,10, 2,0,0, 0,1,0) that defines the viewing coordinate matrix and multiplied with modelview matrix.**
Here the

i) first three parameters are the x,y,z coordinate values for the viewer's eye position.
ii) Second three parameters are the x,y,z coordinate values for the lookat point where the viewer is looking at.
iii) Third three parameters are the x,y,z coordinate values for the viewer's head up direction. This is vector defines the head direction of the viewer.

You can use this function in your display function after enabling modelview matrix.
Finally we need to define the clipping or view volume for the scene like below:



This volume depends on the viewer's position and look at direction and defines which portion of the model world the viewer will see. In 2D scene, clipping area is defined by function glOrtho2d() function.

But in 3D scene, we can define the view volume using three function depending on our requirements are describe below:

1. glFrustum(Xmin, Xmax, Ymin, Ymax, near, far) - This function creates a perspective matrix that produces a perspective asymmetric projection. The eye is assumed to be located at (0,0,0) if not defined.
   Parameters are:
   - 1 & 2:Coordinates for the left and right clipping planes.
   - 3 & 4:Coordinates for the bottom and top clipping planes.
   - 5 & 6:Distance to the near and far clipping planes. Both of these values must be positive.

2. gluPerspective(fovy, aspect ratio, near, far) - This function creates a matrix that describes a viewing symmetric frustum in world coordinates. The aspect ratio should match the aspect ratio of the viewport (specified with glViewport).
   Parameters are:
   - 1.The field of view in degrees, in the y direction.
   - 2.The aspect ratio. This is used to determine the field of view in the x direction. The aspect ratio is x/y.
   - 3 & 4:The distance from the viewer to the near and far clipping plane. These values are always positive.

3. glOrtho(Xmin, Xmax, Ymin, Ymax, near, far) - This function describes a parallel clipping volume. This projection means that objects far from the viewer do not appear smaller
   Parameters are:
   - 1.The leftmost coordinate of the clipping volume.
   - 2.The rightmost coordinate of the clipping volume.
   - 3.The bottommost coordinate of the clipping volume.
   - 4.The topmost coordinate of the clipping volume.
   - 5.The maximum distance from the origin to the viewer.
   - 6.The maximum distance from the origin away from the viewer.

You can see the effects of these projection functions executing projection.exe file in tutor folder. Press P for perspective, F frustum and O for ortho function.
You have to use any one of these functions in your display function after enabling projection matrix.

Remember, you must have to use all the other functions used in display() and main() function in last tutorial.

Finally, we are going to develop our own transformation functions. As we know the general form of transformation matrices given below:

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation matrix   Scale matrix   Rotation (about Z axis) matrix

We just need to develop a function for a specific transformation matrix and define (using parameter value) and store the matrix in a one dimensional array. Then the multiply the matrix with modelview matrix. A sample translation function is given below.

```
void ownTranslatef(GLfloat dx, GLfloat dy, GLfloat dz){
        GLfloat m[16];

        m[0] = 1;    m[4] = 0;   m[8] = 0;   m[12] = dx;
        m[1] = 0;    m[5] = 1;   m[9] = 0;   m[13] = dy;
        m[2] = 0;    m[6] = 0;   m[10] = 1;  m[14] = dz;
        m[3] = 0;    m[7] = 0;   m[11] = 0;  m[15] = 1;

        glMatrixMode(GL_MODELVIEW);
        glMultMatrixf(m);
}
```

Here, the translation matrix is stored in a one dimensional array and each sequential 4 indices represent the corresponding columns of the matrix. glMultMatrixf(m) function is used to multiply the translation matrix with the modelview matrix.

Now try this ownTranslatef() function to translate your object(s).

Solve class work 01and 02 at this stage.