Computer Graphics Lab 3 – 3D Objects, Transformations and Projections

In this lab you will learn how to do the following:

- Create a simple 3D object yourself by specifying the vertices
- Transform this model using the openGL functions glTranslate, glRotate and glScale
- Understand the use of glPushmatrix and glPopmatrix for returning to an earlier state (useful for multiple objects or composite objects)
- Understand the use of the different matrix modes (MODEL_VIEW and PROJECTION) and the movement of the camera
- Render the scene using perspective and orthographic projections

Here are the tasks to do:

1. Construct the Object

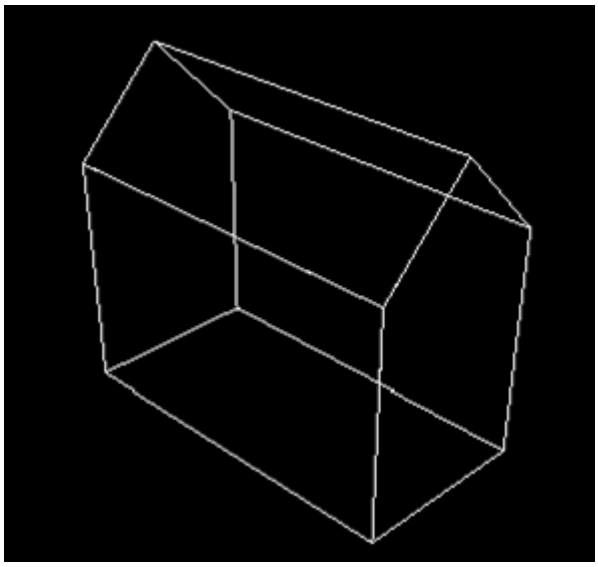
On paper write down the 3D coordinates for each face of a barn/house type object such as the one below. Here are the dimensions:

Length = 16

Height = 14 (to barn apex and 10 to height of side walls)

Width = 8

The object should be centred at the origin with the length along the x-axis, the width along the z-axis and the height along the y-axis. The floor should be at y=0.



You can then use GL_LINE_LOOP to draw a wireframe representation of the object. Draw each face of the barn separately but in the same function. Here we are specifying 3D points so we need to use glVertex3f(x,y,z) instead of glVertex2f. When listing the vertices, do so in a counter clockwise fashion

when looking from the outside to inside. This is important for later operations such as lighting and depth tests. Put the drawing code for the object into its own function or method.

Here is some code that would draw a single face

```
//front side
gl.glBegin(gl.GL_LINE_LOOP);
      gl.glVertex3f(8,10,4);
      gl.glVertex3f(-8,10,4);
      gl.glVertex3f(-8,0,4);
      gl.glVertex3f(8,0,4);
gl.glEnd();
```

2. Position and Direct the Camera/Viewer

Now we have an object in our scene that we want to render. We need to specify a position to view the object from and a position to look at, we use the function gluLookAt for this. Note that this function is a glu function. This stands for graphics library utility (GLU) which is an existension of openGL that groups lower level GL functions together to give higher level functions. To use it in JOGL you need these lines of code:

```
import javax.media.opengl.glu.GLU;
GLU glu = new GLU();
```

Before we call the gluLookAt function we should be in model_view mode. OpenGL maintains two matrices for controlling the geometry and the projection. The model_view matrix is for controlling the scene and camera geometry and the projection matrix controls the projection (for example how big the view volume should be).

To switch between matrix modes you call the following function

gl.glMatrixMode(GL_MODELVIEW); or gl.glMatrixMode(GL_PROJECTION);

It is common practice to initialise the matrices also before you work with them, you initialise them to the identity matrix with the following function call

gl.glLoadIdentity();

So if we were in MODEL_VIEW mode and we called the above function it would set the MODEL_VIEW matrix to be the identity matrix (remember these are 4x4 matrices).

Now you can call gluLookAt to position the camera and set where it is pointing/directing. The syntax and explanation of this function is as follows:

gluLookAt(Gldouble eyex, Gldouble eyey, Gldouble eyez,Gldouble atx, Gldouble aty, Gldouble atz, Gldouble upx, Gldouble upy, Gldouble upz)

- The first three parameters define the eye point of the camera, the position of the camera in space

- The second three parameters define the viewing direction of the camera, that is the point the camera is to look at
- The third three parameters define the direction for the camera to consider as up, if the camera display plane is parallel to the x-y plane and you want the camera to consider up the same direction as the positive y axis the values for the up vector are (0,1,0). This is the case a lot of the time
- Remember, you should be in model view matrix mode when using this function

To do: Use gluLookat to position and orient the camera. Now update some of the parameters into the function so the camera rotates around the object.

3. Project the scene using (a) an orthographics projection and (b) a perspective projection.

Now we want to project our object, first we will perform an orthographic projection. An orthographic projection of the barn is done using the function glOrtho. Here we need to be in projection matrix mode so make sure you switch to this and initialise the projection matrix. The syntax for the function glOrtho is given below:

glOrtho(Gldouble left, Gldouble right, Gldouble bottom, Gldouble top, Gldouble near, Gldouble far) sets up an orthographic projection matrix and defines a rectangular viewing volume with the parameters.

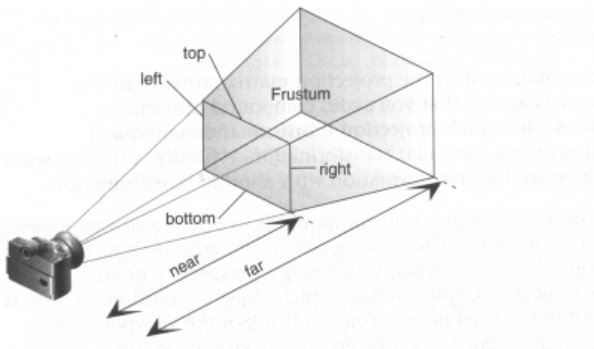
To do: Set up an orthographic projection view volume containing the object

A perspective projection of the barn is performed by switching to the projection matrix and setting up a frustum type of view volume.

There are two functions in openGL for doing this, the first is glFrustum() and the other is gluPerspective()

glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble nearVal, GLdouble farVal)

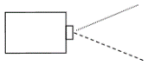
This creates the following viewing volume for our camera. The first five arguments specify the coordinates of the near clipping plane and the last one specifies the extent of the viewing volume.



Perspective Viewing Volume Specified by glFrustum()

• We can create a frustum like this in openGL using the function ***gluPerspective(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far)***

• fov is an angle that specifies the field of view of the virtual camera

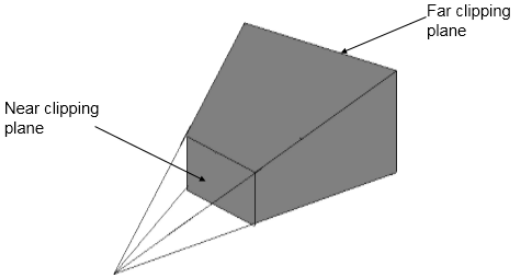


45 degree fov

• aspect sets the aspect ratio of the clipping planes (should always be 1 for our applications)

• near and far are the distances to the near and far clipping planes from the camera

• the default location of the camera in openGL is at the origin of the viewing coordinate system pointing in the negative z direction



So the following code will set up the projection matrix:

```
glMatrixMode(GL_PROJECTION);
gluPerspective(45,1,1,25);
```

This code creates a frustum viewing volume in the negative x direction clipped by planes at z=1 and z=25. Obviously this viewing volume will not contain everything in our coordinate system so how do we look at objects not in this viewing volume we move the virtual camera using gluLookAt mentioned above.

To do: Setup the view volume using gluPerspective as described above such that it contains the object you created earlier. Now try and use glFrustum instead of gluPerspective to create the same volume.

4. Perform some transformations of the object using the openGL functions, glRotate, glTranslate, glScale

Translation:

```
glTranslate*(dx, dy, dz);
```

where [**dx, dy, dz**] is the translation vector.

The effect of calling this function is to concatenate the translation matrix defined by the parameters [dx, dy, dz] to the global model view matrix:

$$M_{modelview} = M_{modelview} * T(dx, dy, dz);$$

Where $T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$

In general, a new transformation matrix is always concatenated to the global matrix from the right. This is often called **post-multiplication**.

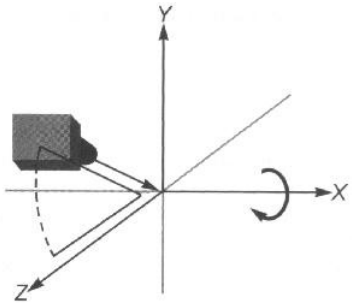
Rotation:

```
glRotate*(angle, x, y, z)
```

Where **angle** is the angle of counterclockwise rotation in degrees, and **x, y** and **z** define a vector(originating at the origin) to rotate about

Typically we rotate about only one of the major axes, which simplifies **x, y** and **z** to be a unit vector.

For example, if we want to rotate about the x-axis, then **x=1, y=0,** and **z=0**.



The effect of calling a rotation matrix is similar to translation. For example, the function call:

```
glRotatef(a, 1, 0, 0);
```

will have the following effect:

$M_{modelview} = M_{modelview} * R_x(a);$

Where $R_x(a)$ denote the rotation matrix about the x-axis for degree a: $R_x(a) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotations about the y-axis or z-axis can be achieved respectively by functions calls:

```
glRotatef(a, 0, 1, 0); // rotation about the y-axis
glRotatef(a, 0, 0, 1); // rotation about the z-axis
```

Scaling

```
glScale*(sx, sy, sz);
```

where **sx, sy** and **sz** are the scaling factors along each axis with respect to the local coordinate system of the model. The scaling transformation allows a transformation matrix to change the dimensions of an object by shrinking or stretching along the major axes centered on the origin.

Example: to make the wire cube three times as high, we can stretch it along the y-axis by a factor of 3 by using the following command:

```
// make the y dimension 3 times larger
glScalef(1, 3, 1);
// draw the cube
glutWireCube(1);
```

- It should be noted that the scaling is always about the origin along each dimension with the respective scaling factors.
- The effect of calling this function to the global model view matrix is similar to translation and rotation.

Note: All the above transformation effect the MODEL_VIEW matrix so you should be in this mode when applying them. Also the order the transformations are applied is the opposite to the order they appear in your program. This is due to the way openGL does the matrix multiplication.

To do: Use these functions in an update() function or method to make the 3D object rotate, scale and translate. So when the user presses the key R, the object should rotate, when they press T, it should translate and when they press S it should scale.

Where to put the function calls:

Have a look at the SquareControl program on Moodle if you have not done so already. In this program there is a constructor (which sets a lot of global variables like the GLCanvas object) that is called inside of main. This is probably a better approach that having everything declared in main.

When you create the Animator object it creates a new thread that repeatedly calls the display function so this is where you call your function to render the bard object (the one with all the GL_LINE_LOOPs in it).

You can have an update function also to move the object or the camera (i.e. make changes to the MODEL_VIEW matrix).

The init() function is called once when the openGL context is first created so is a good place to perform initialisations such as setting the clear colour, setting up lights, setting the state of openGL in terms of performing depth test etc. You can also set up the projection matrix here using the functions mentioned earlier (gluPerspective and glFrustum).

The position of the camera defaults to (0,0,0) as we said earlier so you are required to move it outside of the house object. Do this in an update function that is then called for the display function like before.