Computer Graphics Laboratory
Lab – 01
2D Object Creation and Applying Transformations
(Basic OpenGL Programming)
Version-1.0

Objective of graphics programming

- To create synthetic image such a way that it is close to reality.
- To create artificial animation that emulates virtual reality.
- But why? We have already tools like camera to create image of natural scene and video camera for creating natural movies!

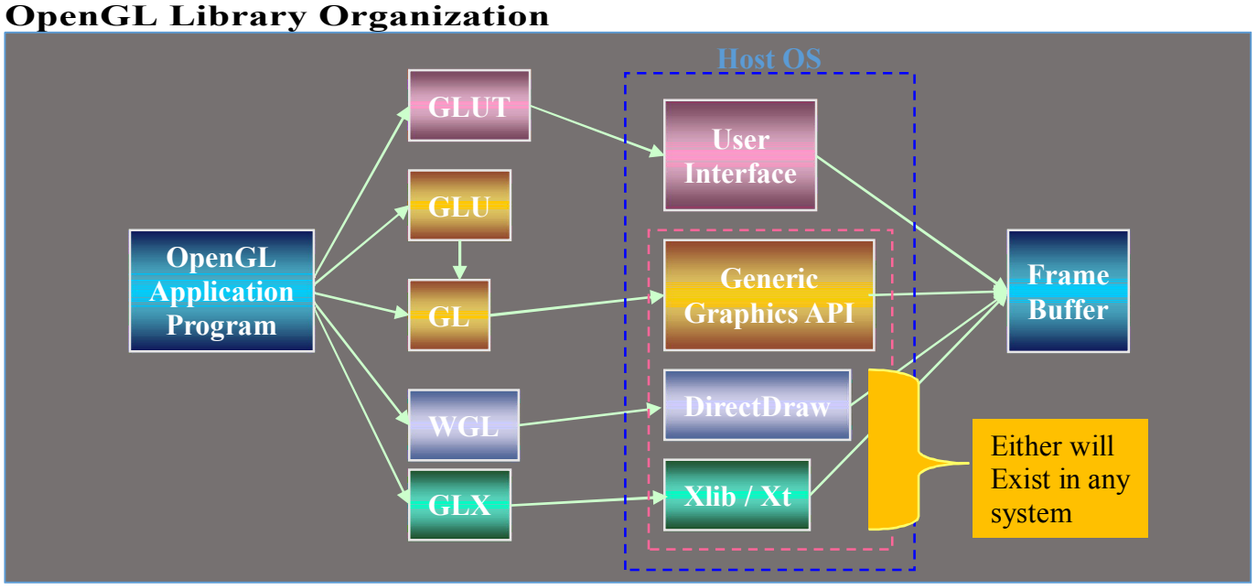
OpenGL – Open Graphics Library

- What it is?
  - ✓ a software interface to graphics hardware
  - ✓ a graphics programming library
  - ✓ Dominant industry "standard" for 3D graphics
  - ✓ Consists of about 150 basic commands
  - ✓ Platform independent
- What it is not?
  - a windowing system (no window creation)
  - a UI system (no keyboard and mouse routines)
  - a 3D modeling system (Open Inventor, VRML, Java3D)

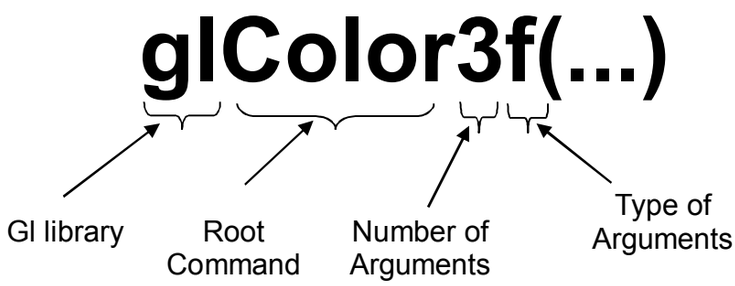
OpenGL Related Libraries

- GL – *gl.h* – *Opengl32.lib*
  - ✓ Provides basic commands for graphics drawing
- GLU (OpenGL Utility Library) – *glu.h* – *glu32.lib*
  - ✓ Uses GL commands for performing compound graphics like
    - viewing orientation and projection specification
    - Polygon tessellations, surface rendering etc.
- GLUT (OpenGL Utility Toolkit) – *glut.h* – *glut.lib*
  - ✓ is a window system-independent toolkit for user interaction

OpenGL Library Organization



Convention of function naming

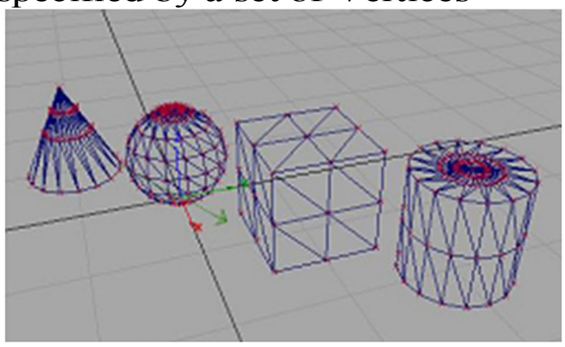


OpenGL APIs

- Primitive functions
  - ✓ glBegin(type);
  - ✓ glVertex(…);
  - ✓ glVertex(…);
    - …
  - ✓ glVertex(…);
  - ✓ glEnd();
- Attribute functions
  - ✓ glColor3f(…);
- Transformation functions
  - ✓ glRotate(…); glTranslate(…);
- Viewing functions
  - ✓ gluLookAt(…);
- Input functions
  - ✓ glutKeyboardFunc(…);
- Control functions
- Inquiry functions

Primitives

- Primitives: Points, Lines & Polygons
- Each object is specified by a set of Vertices

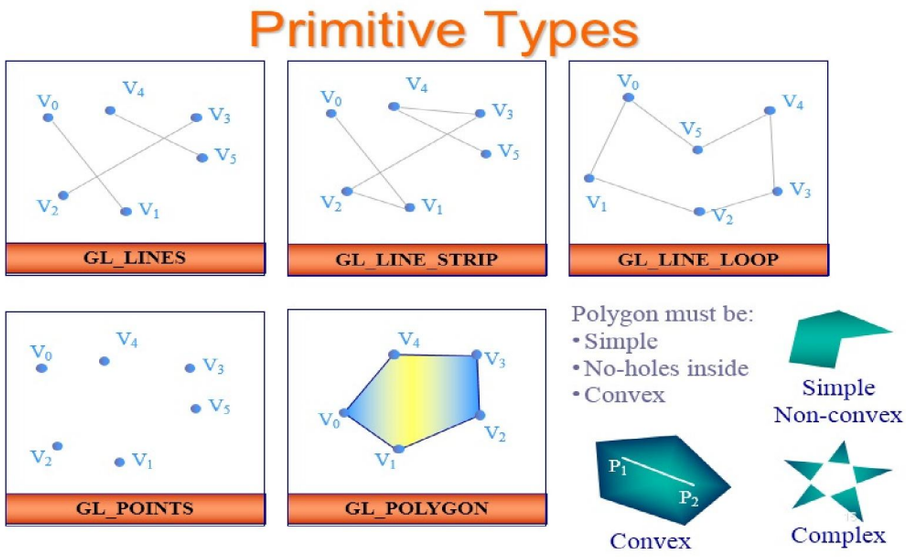


  - ✓ Grouped together by glBegin & glEnd

```
glBegin(type)
        glVertex*( )
        glVertex*( )
        …
glEnd( );
```

  - ✓ type can have 10 possible values
- To specify attributes of last Vertex drawn:
  - ✓ ***glColor*()/ glIndex*()*** current vertex color
  - ✓ ***glNormal*()*** current vertex normal (lighting)
  - ✓ ***glMaterial*()*** current material property (lighting)
  - ✓ ***glTexCoord*()*** current texture coordinate
  - ✓ ***glEdgeFlag*()*** edge status (surface primitives)

Primitive Types



Build your First openGL Program

Developing programs using openGL library is very simple. openGL has reach library functions that help you to develop your own project.

Follow the how create a openGL project in CodeBlocks.pdf file for details information about creating an open GL project in CodeBlocks IDE.

After creating a project, there will be a default program. Run the project and you will see some graphics objects in yout output. Don't need to analyze the program. Simply **delete all the code from main.cpp** file.

Now let's try to develop our first openGL simple program. What you needs to do first is to add the following header files in the top of your main.cpp file.

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
```

Here you already know about last three header files. First three are the required header file for GL, GLU and GLUT library functions.

Suppose we want to create a simple triangle and want to apply some animation on it. To create an object, it is a better practice to develop a user define function for that object. We will just call that function whenever the object is required. So let's create a function named **triangle** in you main.cpp file as like below;

```
void triangle()
{
    glBegin(GL_TRIANGLES);//Denotes the
beginning of a group of vertices that define
one or more primitives.
            glColor3f(1.0,1.0,1.0);
            glVertex2f(2.0,2.0);
            glColor3f(0.0,1.0,0.0);
            glVertex2f(2.0,0.0);
            glColor3f(0.0,1.0,0.0);
            glVertex2f(0.0,0.0);
          glEnd();//Terminates a list of
vertices that specify a primitive initiated by
glBegin.

}
```

Here we create triangle polygon in its own object space like below

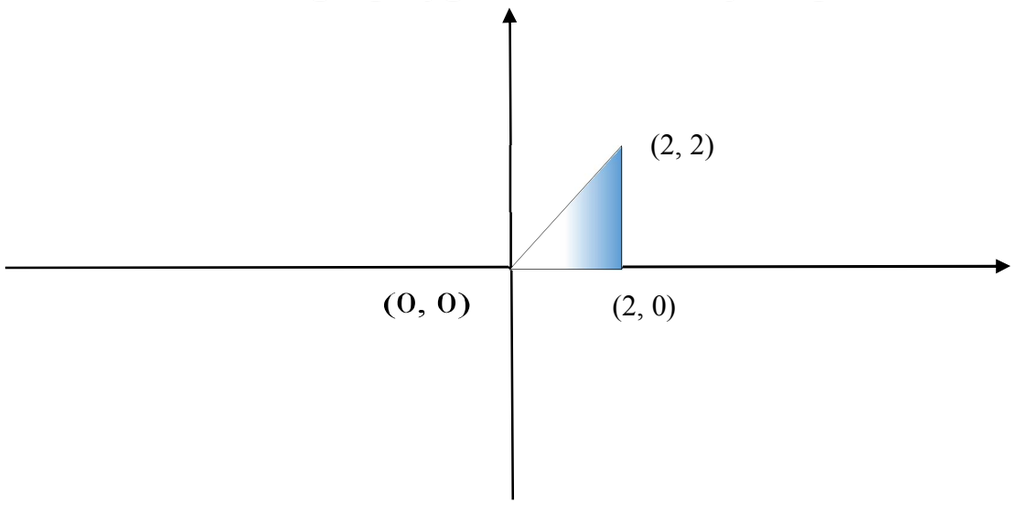


Fig: sample object drawing in object space.

Whose primitives type is `GL_TRIANGLES` that forms filled triangles from given set of vertex points. `glVertex2f(x, y)` function is used to define a vertex point in 2D object space and the attribute function `glColor3f(R, G, B)` is used to apply a particular color in its vertex. Each of the three parameters very between 0 to 1. 0 means 0% of that color and 1 means 100% of that color. Combination of the three colors will apply to that vertex. See here the colors of the vertices are interpolated among the triangle depending on the shade model (will discuss later).

Now our task is to display the object. To do this we develop another user defined function called display(), that includes some openGL library functions along with our defined triangle function like below;

```
void display(void)
{
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluOrtho2D(-3, 3, -3, 3);

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    glViewport(0, 0 ,windowWidth ,windowHeight);

    triangle();

    glFlush();
    glutSwapBuffers();
}
```

Let's introduce with the above library functions.
`glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)` – consists of the bitwise OR of all the buffers to be cleared. Color buffer stores the color of each pixel of the projected scene. And depth buffer stores the depth information of each pixel for multiple overlapping objects. Both needs to cleared before creating new scene.

`glViewport(0, 0 ,windowWidth ,windowHeight)` – sets the region within a window that is used for mapping the clipping volume coordinates to physical window coordinates. First two parameters are the x and y coordinate values of the lower left corner of the window and the last two are the width and height of that window, must be less than or equal to the original window width and height. **Remember,** don't forget to declare two global variable called

`windowWidth ,windowHeight` set their initial value. The figure below illustrate the above function

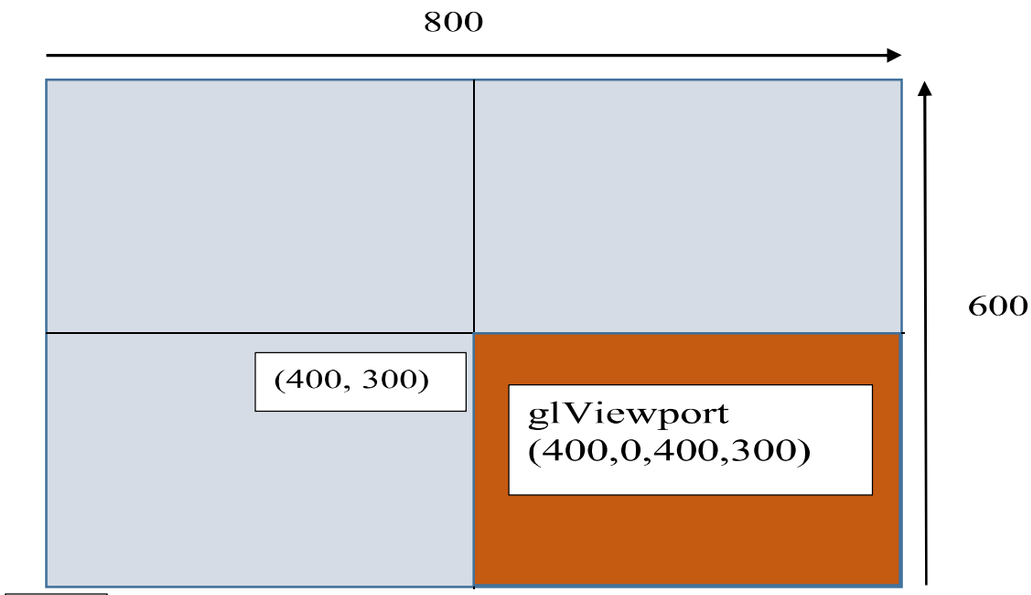


Fig: Sample window and viewport in it.

Here we can see that, if we want we can create more than one viewport in one window, will be demonstrated later. But for now we create a viewport equal to the original window size.

`glMatrixMode()` – This function determines which matrix stack (GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE) is used for matrix operations.

`GL_MODELVIEW`	Matrix operations affect the modelview matrix stack. (Used to move objects around the scene.)
`GL_PROJECTION`	Matrix operations affect the projection matrix stack. (Used to define clipping volume.)
`GL_TEXTURE`	Matrix operations affect the texture matrix stack. (Manipulates texture coordinates.)

Basically openGL works by representing the objects, their transformation, projection etc. in matrix form. Everything is done using any of the three above matrix stack. Mainly modelview and project matrix stack is used. Every operations of each category are stored by multiplying the current operation matrix with the stored matrix.

The projection matrix stack is used store the clipping volume defined by `gluOrtho2D(-3, 3, -3, 3)` function. Therefore `glMatrixMode()` function selects the required matrix stack and then `glLoadIdentity()` function replaces the current transformation matrix with the identity matrix of the stack.

`gluOrtho2D(Xmin, Xmax, Ymin, Ymax)` defines a rectangular clipping volume from world space specifies by the parameters Xmin, Xmax and Ymin, Ymax. This actually specifies which portion of the world coordinate (space) system will displayed in the scene. It defines kind of eye window for a viewer. Comparing with the actual world scenes, a viewer only can see the objects only that are in between its eye boundary. So, the scene outside the boundary of the window is clipped and will not displayed. You can examine this by changing the window size in `gluOrtho2D()` function. Finally, this clipping window is mapped with the viewport window and display the scene in the specified viewport.

`glMatrixMode(GL_MODELVIEW);` and `glLoadIdentity();` are also used to select the modelview matrix stack and replace the current matrix with the identity matrix. Modelview matrix actually store the multiple transformations in multiplication of matrix form that will be apply on particular object(s) for different kind of transformation of the object(s). Therefore, **creating an object and applying transformations on that object must have to be done after selecting the modelview matrix stack**.

Then we draw the triangle by calling `triangle()` function. This triangle will be represented in matrix form and stored in modelview matrix stack, multiplied by the identity matrix in modelview matrix.

Finally, `glFlush()` is used to ensure the drawing commands are actually executed rather than stored in a buffer waiting additional OpenGL commands.
`glutSwapBuffers()` performs a buffer swap on the layer in use for the current window. Specifically, glutSwapBuffers promotes the contents of the BACK BUFFER of the layer in use of the current window to become the contents of the FRONT BUFFER. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after glutSwapBuffers is called.

By now, our display function is ready. At this stage all we need is a main function to execute the defined functions. Here we have a special main() function for this project;

```
int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB
| GLUT_DEPTH);

    glutInitWindowPosition(100,100);
    glutInitWindowSize(windowWidth,
windowHeight);
    glutCreateWindow("Traingle-Demo");

    glShadeModel( GL_SMOOTH );
    glEnable( GL_DEPTH_TEST );

    glutDisplayFunc(display);

    glutMainLoop();

    return 0;
}
```

Now let's know about the new functions used here:
`glutInit(&argc, argv)` - This initializes the GLUT library, passing command line parameters (this has an effect mostly on Linux/Unix).
`glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH)` - Initial parameters for the display. In this case, we are specifying a RGB display (GLUT_RGB) along with double-buffering (GLUT_DOUBLE), so the screen won't flicker when we redraw it. GLUT_DEPTH specifies a 32-bit depth buffer.
`glutInitWindowPosition(100,100)` – sets the original display window start position in your display device. Remember, display screen coordinates starts from upper left corner of the screen.
`glutInitWindowSize(windowWidth,windowHeight)` - Initial window size - width, height

`glutCreateWindow("Traingle-Demo")` - Creates and sets the window title.
`glShadeModel( GL_SMOOTH )` - Sets the default shading to flat or smooth. OpenGL primitives are always shaded, but the shading model can be flat (GL_FLAT) or smooth (GL_SMOOTH).
`glEnable( GL_DEPTH_TEST )` - glEnable enables an OpenGL drawing feature. Here it enables the depth test features for depth buffer.
`glutDisplayFunc(display)` - This function tells GLUT which function to call whenever the windows contents must be drawn. This can occur when the window is resized or uncovered or when GLUT is specifically asked to refresh with a call to the glutPostRedisplay function (will be discussed later).
`glutMainLoop()` - This function begins the main GLUT event-handling loop. The event loop is the place where all keyboard, mouse, timer, redraw, and other window messages are handled. This function does not return until program termination.

At last we are ready to execute our program. This simple program will display a triangle in the display window.

Here in this program, let's verify and observe some effects:

1. Change the viewport's position and size and try to understand the effect of viewport transformation.
2. Now change clipping window size and try understand the effect varying the objects' size.

Till now, we are able to draw and display a particular object. Next task is to apply some transformations on that object to place the object in the world space in an appropriate fashion. In openGL, there are three basic transformation functions;

1. `glTranslatef(Xval,Yval,Zval)` – this function forms a translation matrix that translate (simply adds) the current object's x, y, z coordinate values by Xval, Yval, Zval respectively. This matrix is multiplied with modelview matrix stored in modelview matrix stack.
2. `glScalef(Xval,Yval,Zval)` – this function forms a scale matrix that scale (simply multiply) the current object's x, y, z coordinate values by Xval, Yval, Zval respectively. This matrix is multiplied with modelview matrix stored in modelview matrix stack.
3. `glRotatef(theta,Xval,Yval,Zval)` – this function forms a rotation matrix that rotate the current object's x, y, z coordinate values by theta degree in anti-clockwise direction with respect to vector formed by Xval, Yval, Zval and centered at origin. This matrix is multiplied with modelview matrix stored in modelview matrix stack.

Now let's apply these functions on our created triangle.
Suppose, we want to translate the triangle 1unit in right and 1unit down to its current position. Then we need to call the `glTranslatef(1,-1,0)` function just before the `triangle()` function in our `display()` function. In openGL, the transformations you want to apply, must need to define before the object. All the transformation functions defined before the object will affect the object.
If you want more than one transformations on object(s), for example you want a translation first and then a rotation on the triangle. Then both the transformation functions must called before the triangle, but `glTranslatef(1,-1,0)` needs to be called immediately before

`triangle()` and then `glRotatef(45,0,0,1)` before the `glTranslatef(1,-1,0)`. That is in the following order;

```
glRotatef(45,0,0,1);
glTranslatef(1,-1,0);
triangle();
```

This will translate the triangle first and then rotate 45 degree anti-clockwise with respect to Z axis.
Remember, the order of placing the transformation function is important. Because, the output of first translation, then rotation is not the same for the output of first rotation, then translation. For justification replace the above function by the following:

```
glRotatef(45,0,0,1);
glTranslatef(1,-1,0);
triangle();
```

Solve class work 01 at this stage.

Now, we want to display the same triangle twice. Very simple, call the `triangle()` function again just after the first one. But in output you will see only one triangle (same as previous output). Why?
Actually the second triangle is also drawn along with first one and they completely overlapped. But why they overlap? We did not apply any transformations for the second triangle.
Remember, all the transformations defined before the object will affect the object (mentioned above).
But, we don't want to apply the transformations on the second triangle. What could be the solution? You can push the transformation matrices along with first triangle in the modelview matrix stack and pop the matrix when to display using glPushMatrix() and glPopMatrix() like below:

```
glPushMatrix();
    glTranslatef(1,-1,0);
    glRotatef(45,0,0,1);
    triangle();
glPopMatrix();

glPushMatrix();
    triangle();
glPopMatrix();
```

Now see the output. There will be two triangles in different shape. That is second triangle is now out of the effect of the transformations. Here `glPushMatrix()` - pushes the current matrix onto the current matrix stack. This function is most often used to save the current transformation matrix so that it can be restored later with a call to glPopMatrix().
`glPopMatrix()` - Pops the current matrix off the matrix stack.

Now try some different transformations for the second triangle and you will see that affects only the second one.

Now we will introduce with glut keyboard and idle functions. First, the purpose of keyboard function is to listen a key press and perform an action with respect to that key press. It needs to define a user defined function, where we define the actions for a specific key press. Say, we want to translate the first triangle left, right, up and down using key L, R, U and D respectively. So the defined function should be look like below:

```
void myKeyboardFunc( unsigned char key, int x, int
y )
{
            switch ( key )
            {
```

```
            case 'r':
                Txval+=0.2;
                break;
            case 'l':
                Txval-=0.2;
                break;
            case 'u':
                Tyval+=0.2;
                break;
            case 'd':
                Tyval-=0.2;
                break;
            case 27:    // Escape key
                exit(1);
        }
        glutPostRedisplay();
}
```

Here the function parameters are predefined and cannot be changed. Parameter `key` receive the pressed key value and `x` and `y` are the mouse position if mouse action is to be listened. Also we use two global variables named Txval and Tyval. We just changing these variables value with respect to any key press.

Our task is to translate the first triangle left, right, up and down using key L, R, U and D respectively. So, we must use these global variables in the parameters of glTranslatef() function like below:

```
glPushMatrix();
    glTranslatef(Txval,Tyval,0);
    glRotatef(45,0,0,1);
    triangle();
 glPopMatrix();
```

This is not yet done. You must add one function in your keyboard function called `glutPostRedisplay()`.

This function informs the GLUT library that the current window needs to be refreshed. Multiple calls to this function before the next refresh result in only one repainting of the window. Therefore, without this function you cannot see the keyboard actions.

Now you need to call your keyboard function (`myKeyboardFunc`) from the main() function using function `glutKeyboardFunc(myKeyboardFunc)` which sets the keyboard callback function for the current window.

Finally, the last function for this tutorial is to apply an idle function in your program (optional). Suppose you want to apply some transformation continuously until you stop it. Say, we want to scale the second triangle continuously until key press. For this we need to add some codes in our keyboard function and define a function called animate() like below:

```
        case 'S':
              flagScale=true;
              break;
        case 's':
              flagScale=false;
              break;
```

add the above codes into the switch case.

```
void animate()
{
        if (flagScale == true)
        {
          sval+= 0.005;
          if(sval > 3)
              sval = 0.005;
        }
        glutPostRedisplay();
```

```
}
```

Here the global variable is used in the function in display function like below:

```
glPushMatrix();
     glScalef(sval,sval,1);
     triangle();
    glPopMatrix();
```

Now you need to call your animate function from the main() function using function `glutIdleFunc(animate)` which is particularly useful for continuous animation or other background processing.

At this stage, let's play with more than one viewports. Modify you display function with the following code segments below:

```
     glPushMatrix();
     glViewport(400, 300 ,400 ,300);
     glTranslatef(Txval,Tyval,0);
     glRotatef(45,0,0,1);
     triangle();
     glPopMatrix();

     glPushMatrix();
     glViewport(0, 0 ,400,300);
     glScalef(sval, sval,1);
     triangle();
     glPopMatrix();
```

Here we add specific viewport for each matrix stack omitting the single general viewport function. In this way you can create more than one viewport in your output window and display specific objects in that viewport.

Solve class work 02 at this stage.