Lab3. Transformations (translation, rotation and scaling)

Introduction

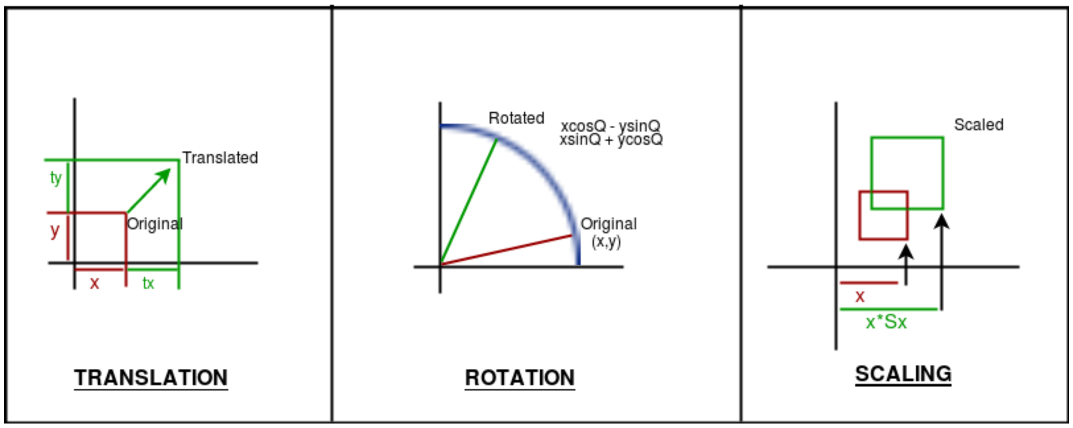
Transformations play a very important role in manipulating objects on screen. We can implement the algorithms corresponding to different types of transformation but we will prefer to use the OpenGL built-in functions instead.

There are three basic kinds of Transformations in Computer Graphics:

1. Translation: refers to moving an object to a different position on screen.

2. Rotation: refers to rotating a point or an object

3. Scaling: refers to zooming in and out an object in different scales across axes



There are mainly three functions used by OpenGL to perform the transformations mentioned above. These functions are respectively `glTranslate3f`, `glRotate3f` and `glScale3f`.

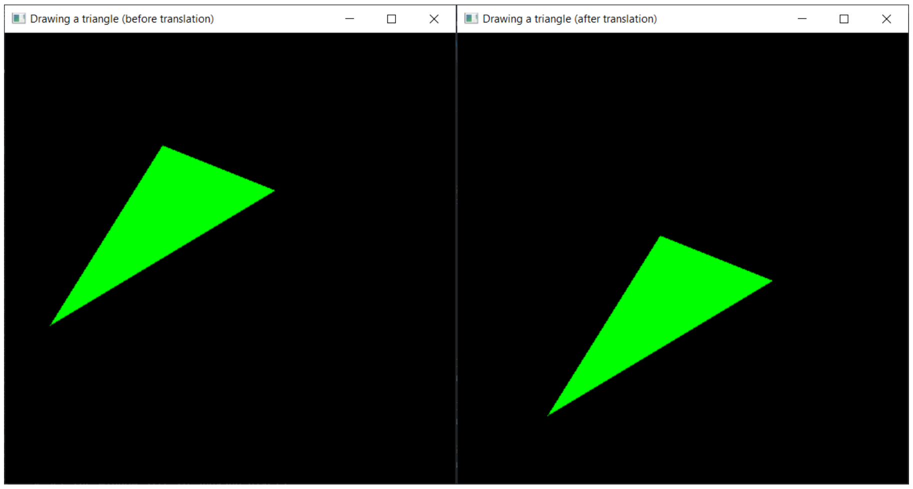
Each of the previous functions defines a matrix, which is multiplied by the current matrix to make the subsequent graphics perform the corresponding transformation (translation rotation or scaling).

Example no. 1:

The following code draws a triangle then translates it along the vector (0.2, -0.4, 0) using `glTranslate3f` function:

```
def render_Scene():
    #Draw a green triangle
    glColor3f(0,1,0);
    # translate the triangle
    glTranslatef(0.2, -0.4, 0);
    glBegin(GL_TRIANGLES);
    glVertex2f( -0.8, -0.3);
    glVertex2f( -0.3, 0.5 );
    glVertex2f( 0.2, 0.3 );
    glEnd();
```

The following outputs show the triangle before and after translation:

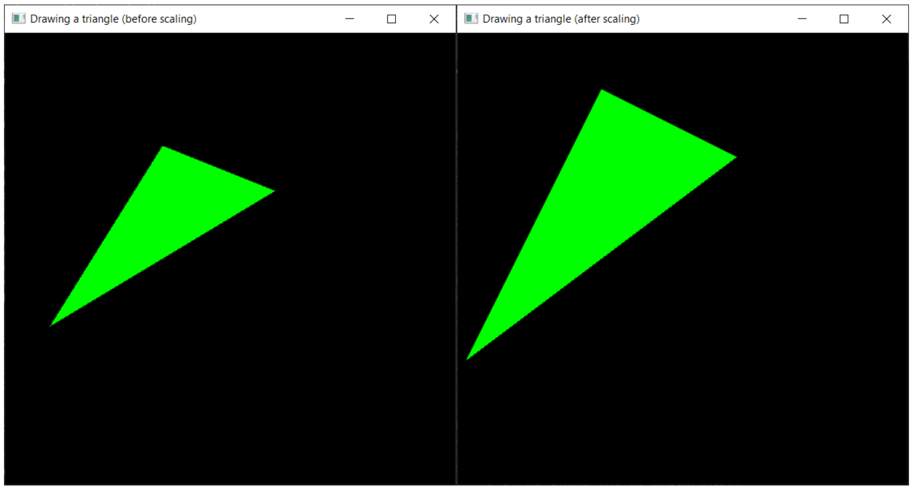

Drawing a triangle (before translation)
Drawing a triangle (after translation)

Example no. 2:

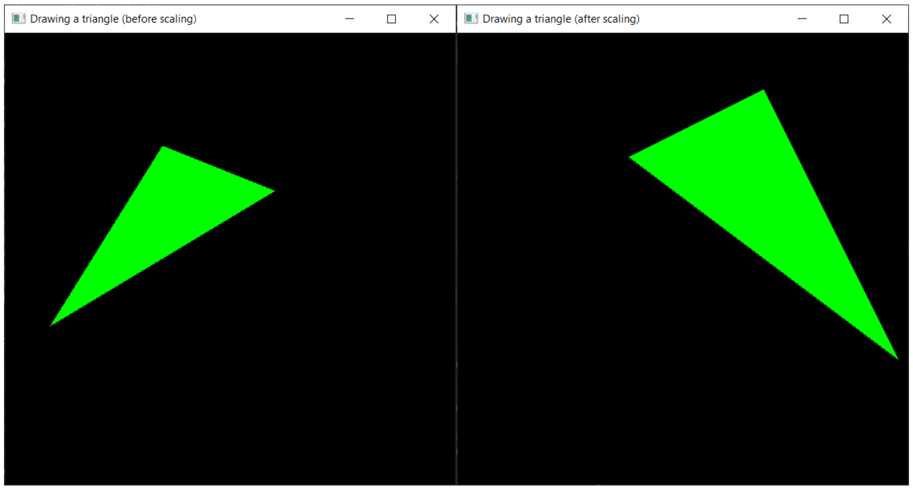
The following code draws a triangle, then scales it with the scaling factors across x, y and z axis equal to 1.2, 1.5 and 1 respectively, using `glScale3f` function:

```
def render_Scene():
    #Draw a green triangle
    glColor3f(0,1,0);
    # scale the triangle
    glScale(1.2, 1.5, 1.0);
    glBegin(GL_TRIANGLES);
    glVertex2f( -0.8, -0.3);
    glVertex2f( -0.3, 0.5 );
    glVertex2f( 0.2, 0.3 );
    glEnd();
```

The following outputs show the triangle before and after scaling:


Drawing a triangle (before scaling)
Drawing a triangle (after scaling)

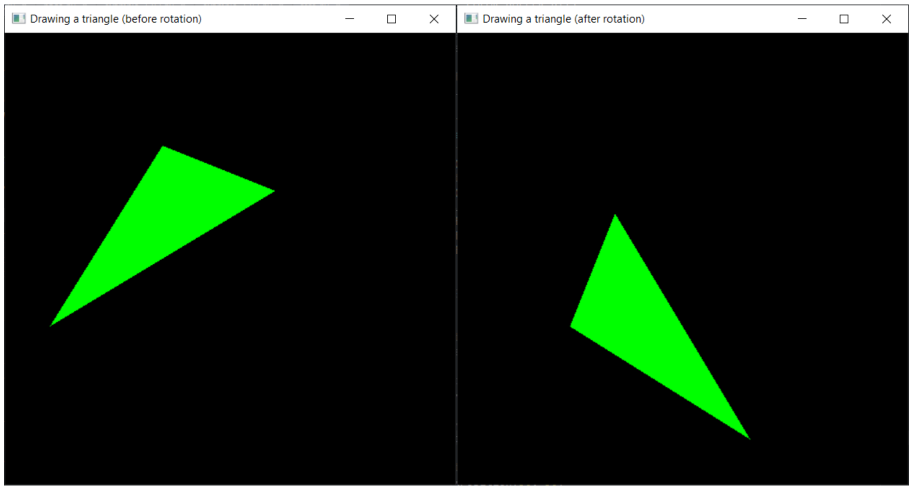
Note that the **scaling factors may be negative**. For example, the scaling of the same previous triangle with factors across x, y and z axis equal to -1.2, 1.5 and 1 respectively will give the following output:


Drawing a triangle (before scaling)
Drawing a triangle (after scaling)

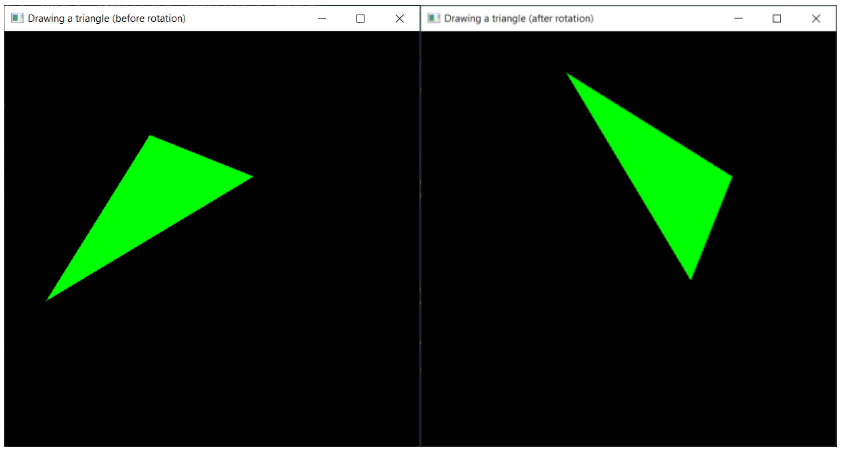
Example no. 3:

The following code draws a triangle then rotates it 90 degrees (anticlockwise rotation) about the z-axis (vector (0,0,1)) using `glRotate3f` function:

```
def render_Scene():
    #Draw a green triangle
    glColor3f(0,1,0);
    # rotate the triangle
    glRotatef(90, 0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f( -0.8, -0.3);
    glVertex2f( -0.3, 0.5 );
    glVertex2f( 0.2, 0.3 );
    glEnd();
```


Drawing a triangle (before rotation)
Drawing a triangle (after rotation)

Note that the rotation angle may be negative. For example, the rotation of the same previous triangle with an angle of -90 degrees (clockwise rotation) about the z-axis (vector (0,0,1)) will give the following output:


Drawing a triangle (before rotation)
Drawing a triangle (after rotation)

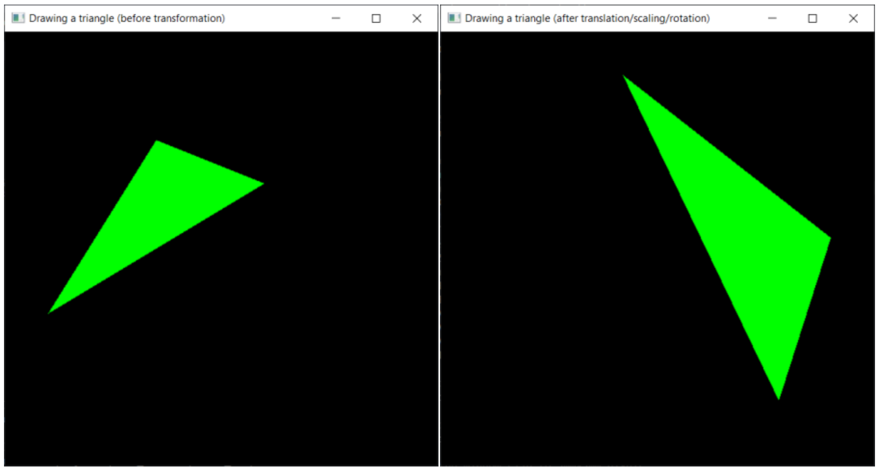
Combining transformations

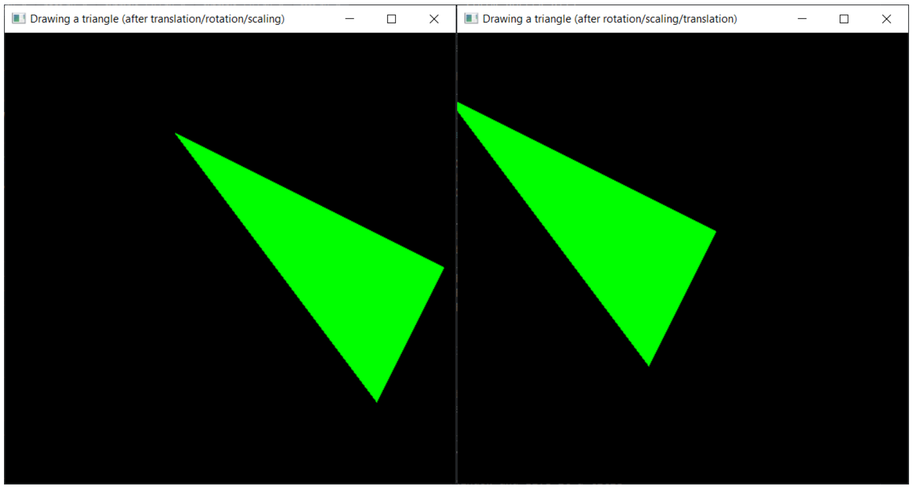
We can apply several combined transformations to the same object (translation, scaling, rotation). The **order of the transformations** to apply may influence the final form of the object.

Code:

```
def render_Scene():
    #Draw a green triangle
    glColor3f(0,1,0)
    glRotatef(-90, 0,0,1)           # rotate the triangle
    glScale(1.2, 1.5, 1.0)          # scale the triangle
    glTranslatef(0.2, -0.4, 0)      # translate the triangle
    glBegin(GL_TRIANGLES)
    glVertex2f(-0.8, -0.3)
    glVertex2f(-0.3, 0.5 )
    glVertex2f(0.2, 0.3 )
    glEnd()
```

Outputs:


Drawing a triangle (before transformation)
Drawing a triangle (after translation/scaling/rotation)


Drawing a triangle (after translation/rotation/scaling)
Drawing a triangle (after rotation/scaling/translation)

We can make animations using transformations. To do this, we must add the call to the function `glutIdleFunc` in the main code. This function sets the global idle callback function name (given as argument) so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. The code that animates the scene must then be added to the idle function that will be defined before the main code. Also in the idle callback function we must add a call to the function `glutPostRedisplay()` to mark the current window as needing to be redisplayed and thus call, after each individual animation, the display callback function to take into account the modifications added to the scene. Note that, if the animation speed is fast we can slow it down by adding some sleep period using the function sleep of the time Python module.

For example, the complete code that allows to animate an oscillatory movement of a triangle forth and back (right and left) with respect to the y-axis is given below:

```
# Importing the necessary Modules
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
import time
move = 0.3

# Disply callback function
def display():
    # Reset background
    glClear(GL_COLOR_BUFFER_BIT)
    # Render scene
    render_Scene()
    # Swap buffers
    glutSwapBuffers()

def idle():
    global move
    glTranslatef(move,0,0)
    move *= -1
    time.sleep(0.2)
    glutPostRedisplay()
```

```
# Scene render function
def render_Scene():
    glColor3f(1,0,0)
    glBegin(GL_TRIANGLES)
    glVertex2f(-0.3,-0.1)
    glVertex2f(0.3,-0.1)
    glVertex2f(0,0.3)
    glEnd()

# Initialize GLUT
glutInit()
# Initialize the window with double buffering and RGB colors
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
# Set the window size to 500x500 pixels
glutInitWindowSize(500, 500)
# Create the window and give it a title
glutCreateWindow("Animating a triangle")
# Set the initial window position to (50, 50)
glutInitWindowPosition(50, 50)
# Define display callback
glutDisplayFunc(display)

# Define idle callback
glutIdleFunc(idle)

# Begin event loop
glutMainLoop()
```

Exercises:

1. Modify the given code in order to make the oscillatory movement symmetric from either side of the y-axis.
2. Modify the previous code in order to make the oscillatory movement symmetric from either side of the x-axis (**up and down**).
3. Modify the given code so that the triangle moves continuously in one direction (from the left to the right) then, when it reaches the end of the window (the right one) and disappears, it reappears from the other end (the left one) and restarts its movement.

Homework:

OpenGL allows to handle keyboard interactions thanks to GLUT. In order to do that, we must define a special keyboard callback for the current window using the GLUT functions `glutKeyboardFunc` or `glutSpecialFunc` depending on whether the pressed key is a normal key or a function/arrow key. For example, the special keyboard callback is triggered when keyboard function (F1…F12) or directional keys (↑, ↓, →, ←) are pressed. The key callback parameter is a GLUT_KEY_* constant for the special key pressed (for details see https://www.opengl.org/resources/libraries/glut/spec3/node54.html). Then we should write in the callback routine the necessary code that allows to handle the key(s) pressed.
Your task is to modify the code corresponding to the continuously moving triangle so that the triangle moves in the 4 directions of the X-Y plane (right, left, up and down) using the corresponding arrow key (right, left, up and down).