ENSIM, 5A INFO IPS

Image Synthesis, Lab 2. :

2.1-Drawing polygons

Drawing Triangles

A triangle is a primitive formed by 3 vertices. It is the 2D shape with the smallest number of vertices, so renderers are typically designed to render triangles.

Since a triangle is created from only 3 vertices, it is also guaranteed to be planar.

We can draw triangles in OpenGL using the “hard way” by drawing closed-loop connected lines (GL_LINE_LOOP primitive), but it would be better to use special OpenGL primitives dedicated to this kind of tasks.

There are 3 kinds of triangle primitives, based again on different interpretations of the vertex stream:

GL_TRIANGLES: Vertices 0, 1, and 2 form a triangle. Vertices 3, 4, and 5 form a triangle. And so on.

GL_TRIANGLE_STRIP: Every group of 3 adjacent vertices forms a triangle. The face direction of the strip is determined by the winding of the first triangle. Each successive triangle will have its effective face order reversed, so the system compensates for that by testing it in the opposite way. A vertex stream of n length will generate n-2 triangles.

GL_TRIANGLE_FAN: The first vertex is always held fixed. From there on, every group of 2 adjacent vertices form a triangle with the first. So with a vertex stream, we get a list of triangles like so: (0, 1, 2) (0, 2, 3), (0, 3, 4), etc. A vertex stream of n length will generate n-2 triangles.

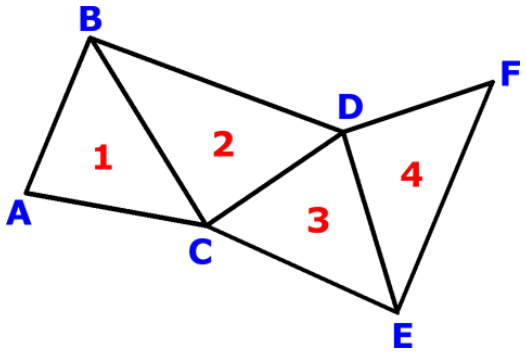
Here are some examples that can be more illustrative:

GL_TRIANGLES:

```
Indices:     0 1 2 3 4 5 ...
Triangles:  {0 1 2}
                  {3 4 5}
```

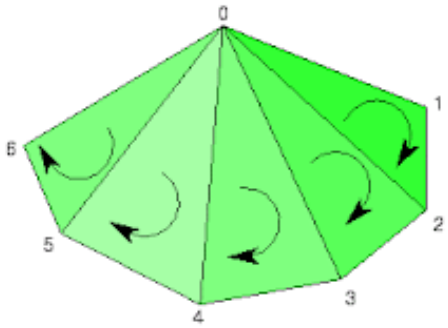
GL_TRIANGLE_STRIP:

```
Indices:     0 1 2 3 4 5 (points A, B, C, D, E and F)
Triangles:  {0 1 2}
              {1 2 3    drawing order is (2 1 3) to maintain proper winding
                {2 3 4}          drawing order is (3 2 4) to maintain proper winding
                  {3 4 5}                  drawing order is (4 3 5) to maintain proper winding
```



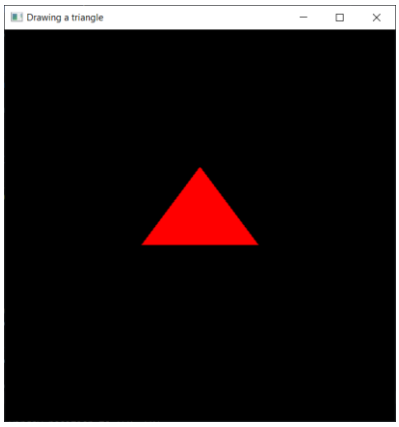
GL_TRIANGLE_FAN:

```
Indices:      0 1 2 3 4 5 6...
Triangles:  {0 1 2}
            {0} {2 3}
            {0}   {3 4}
            {0}     {4 5}
```



Write the code that allows to draw a red

The corresponding output is the following:



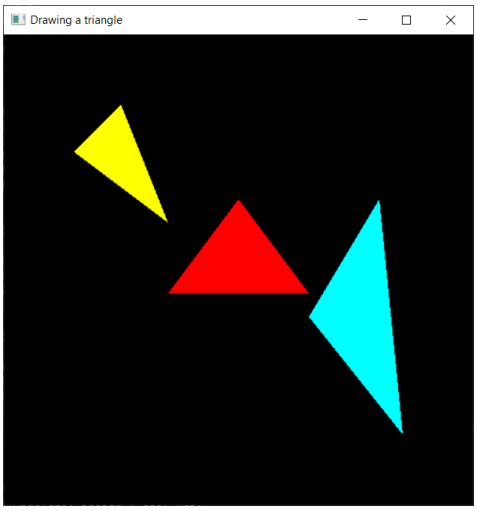
We can draw several triangles in the same windows using a single block (glBegin/glEnd) for all the added points/vertices (their number should be a multiple of 3) or by adding the block (glBegin/glEnd) as many times as needed.

Give the code that allows to draw 3 triangles with different colors in the same window:

Version 1. using a single block (glBegin/glEnd) for all the added points

Version 2. using a block (glBegin/glEnd) for each triangle

The corresponding output is the following:

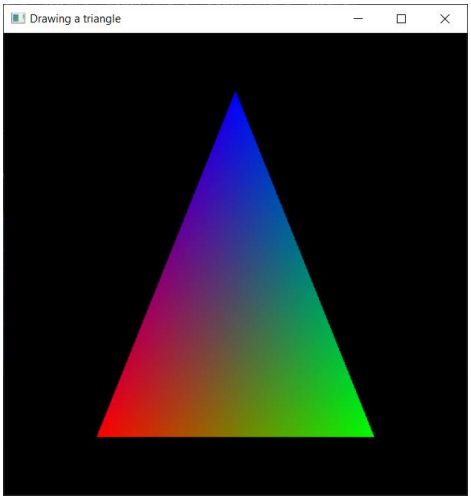


Using a different color for each vertex

When using a different color for each of the vertices of the triangle, the colors of the inner pixels of the triangle will be determined using a barycentric interpolation.

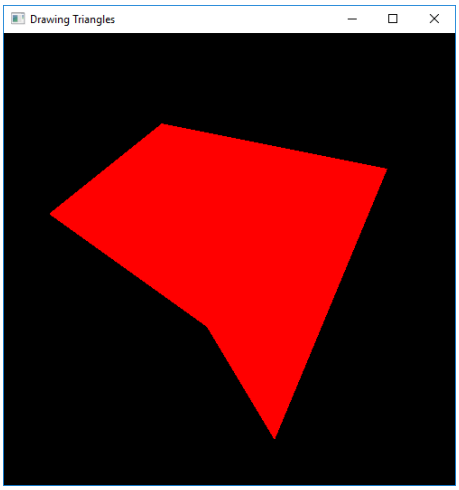
Write the code that illustrates this principle:

The corresponding output could be:



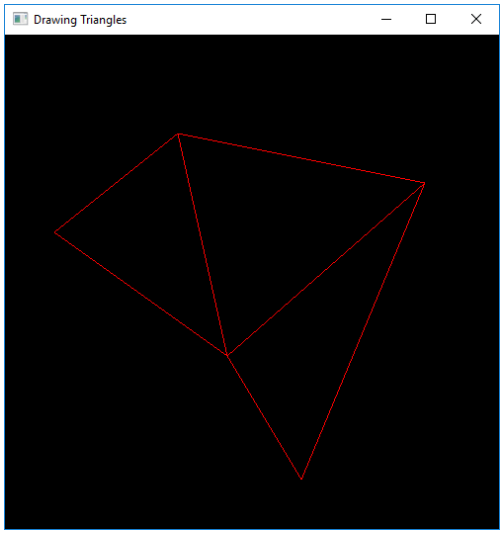
Using the directive GL_TRIANGLE_STRIP with the same color for all vertices

The corresponding output is shown



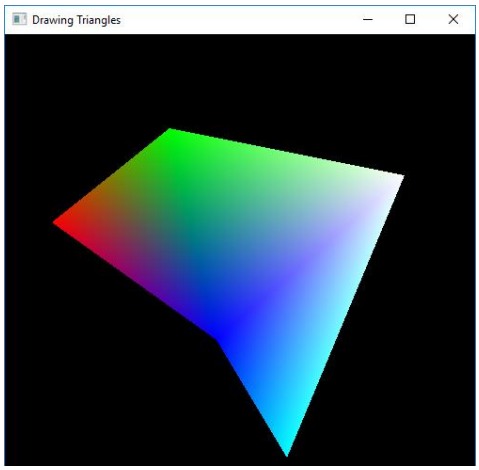
We can use the function `glPolygonMode` to precise that we want an empty shape (triangle in our case): `glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)`

The corresponding output is:



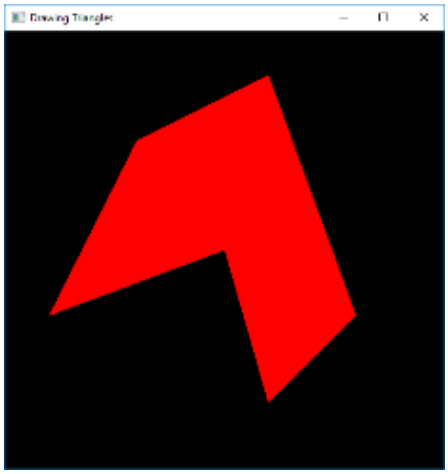
Using the directive GL_TRIANGLE_STRIP with different colors for the vertices

The corresponding output is shown to the right:



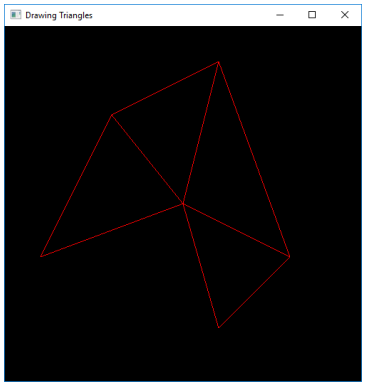
Using the directive GL_TRIANGLE_FAN with the same color for all the vertices (filled triangles)

Output shown

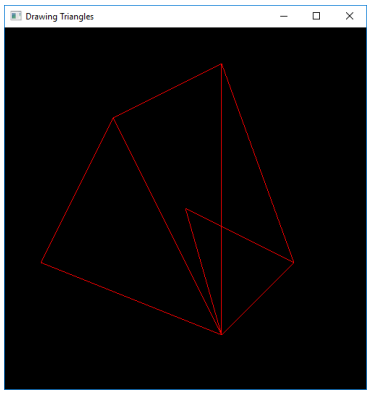


Using the directive GL_TRIANGLE_FAN with the same color for all the vertices (empty triangles)

Output shown to the right:

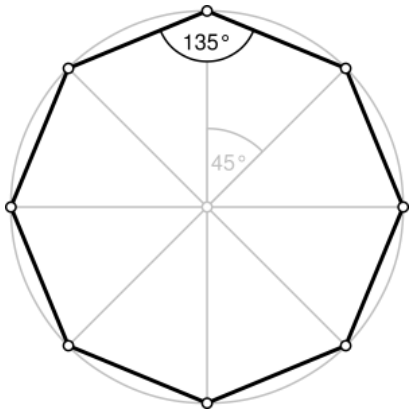


Be careful of the choice of the central point as well as the order of the points. For example, if we interchange the orders of the first and the last points in the previous example we will get the following output:



Exercise

Draw a regular polygon (e.g. an octagon) (see figure to the right) using a certain number of triangles with each of the directives GL_TRIANGLES, GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN. The center of the polygon will be (0,0) and the number of triangles should be fixed at the beginning of the code.



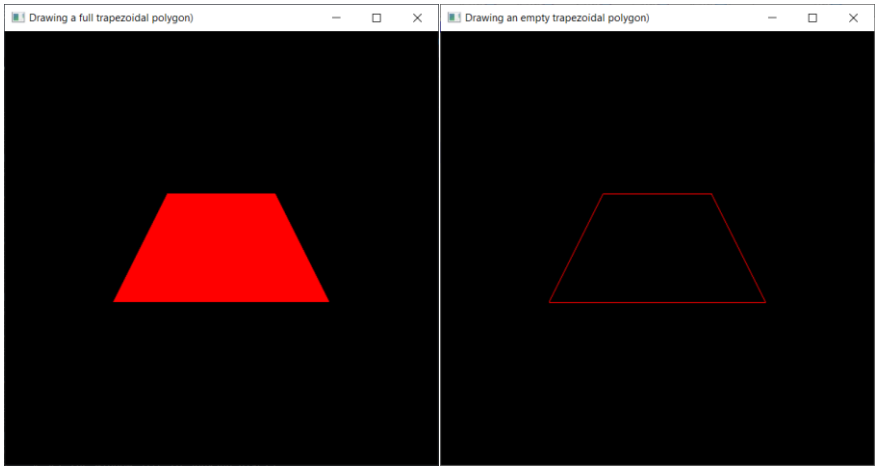
Other drawing primitives in OpenGL

GL_QUADS

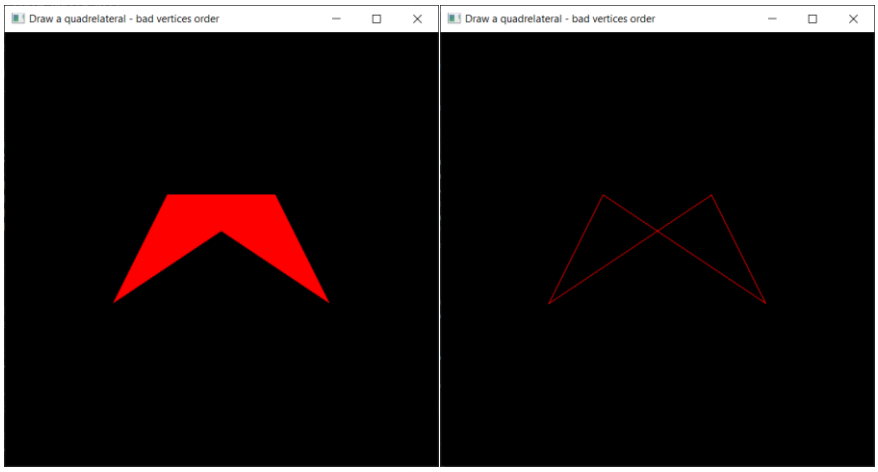
OpenGL has a special primitive called GL_QUADS that allows to draw a quadrilateral polygon knowing its four vertices.

For example, the following code allows to draw a full red trapezoidal polygon using GL_QUADS (we can draw an empty version by adding `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`).

Output:



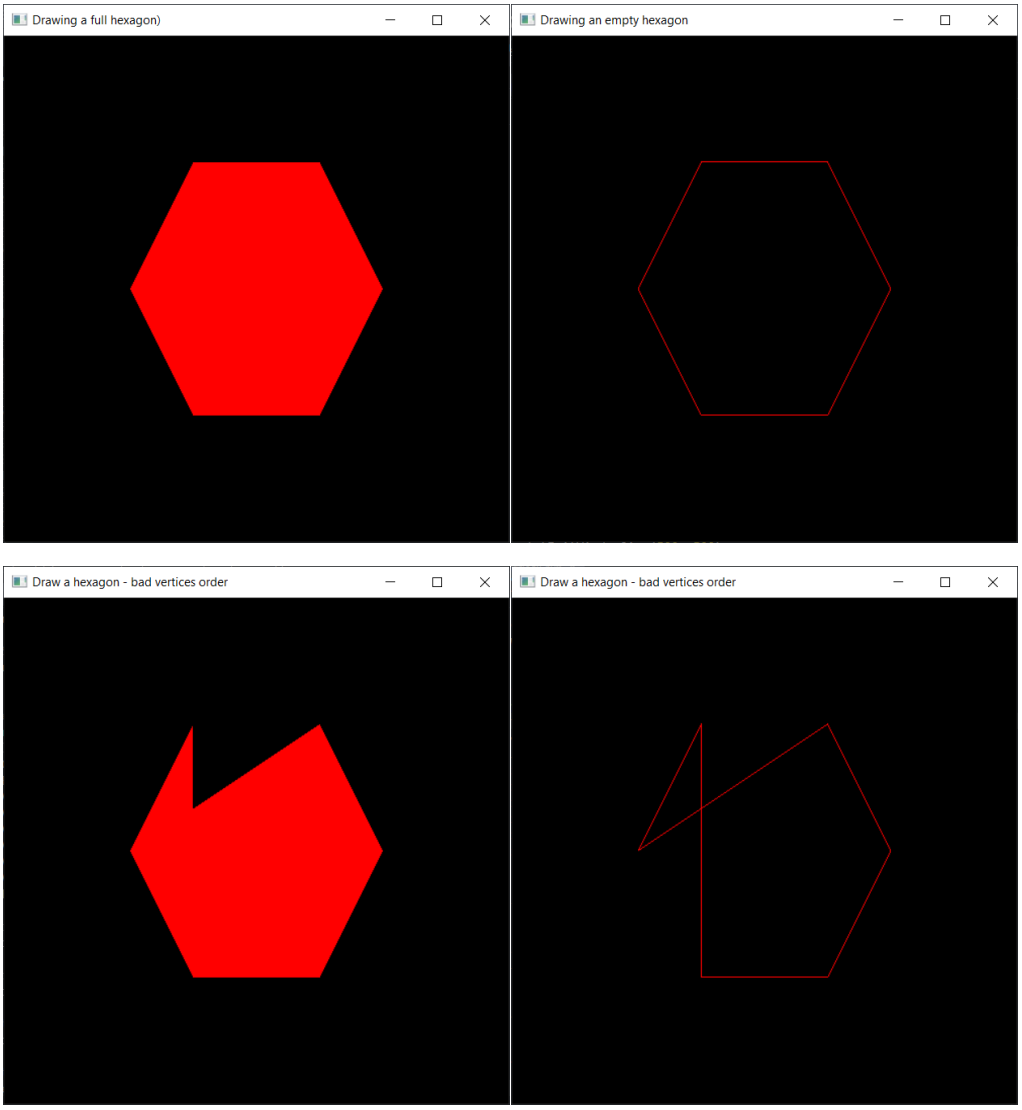
Be careful of respecting the **order of the points/vertices** of the polygone. For example, if we interchange the order of the first two points in the previous quadrilateral rendering code we obtain the following outputs:



GL_POLYGON

OpenGL has a special primitive called GL_POLYGON that allows to draw an arbitrary polygon knowing its vertices (more or equal to 3).

For example, the following code allows to draw a full red hexagon using GL_POLYGON (we can draw an empty version by adding `glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)`). Be careful of respecting the order of the points/vertices of the polygon.



Rendering Primitive Shapes in PyOpenGL

There are some primitive shapes (objects) in PyOpenGL which can be drawn easily using some predefined functions. These functions are part of the OpenGL Utility Toolkit (GLUT).

Two variants of these functions exist and allows to draw a solid and a wireframe version of the object respectively. Some of these functions are:

`glutSolidSphere`: renders a solid sphere (e.g. `glutSolidSphere(1,50,25)`)

`glutWireSphere`: renders a wireframe sphere (e.g. `glutWireSphere(1,50,25)`)

`glutSolidTeapot`: renders a solid teapot (e.g. `glutSolidTeapot(0.5)`)

`glutWireTeapot`: renders a wireframe teapot (e.g. `glutWireTeapot(0.5)`)

`glutSolidCube`: renders a solid cube (e.g. `glutSolidCube(1)`)

`glutWireCube`: renders a wireframe cube (e.g. `glutWireCube(1)`)

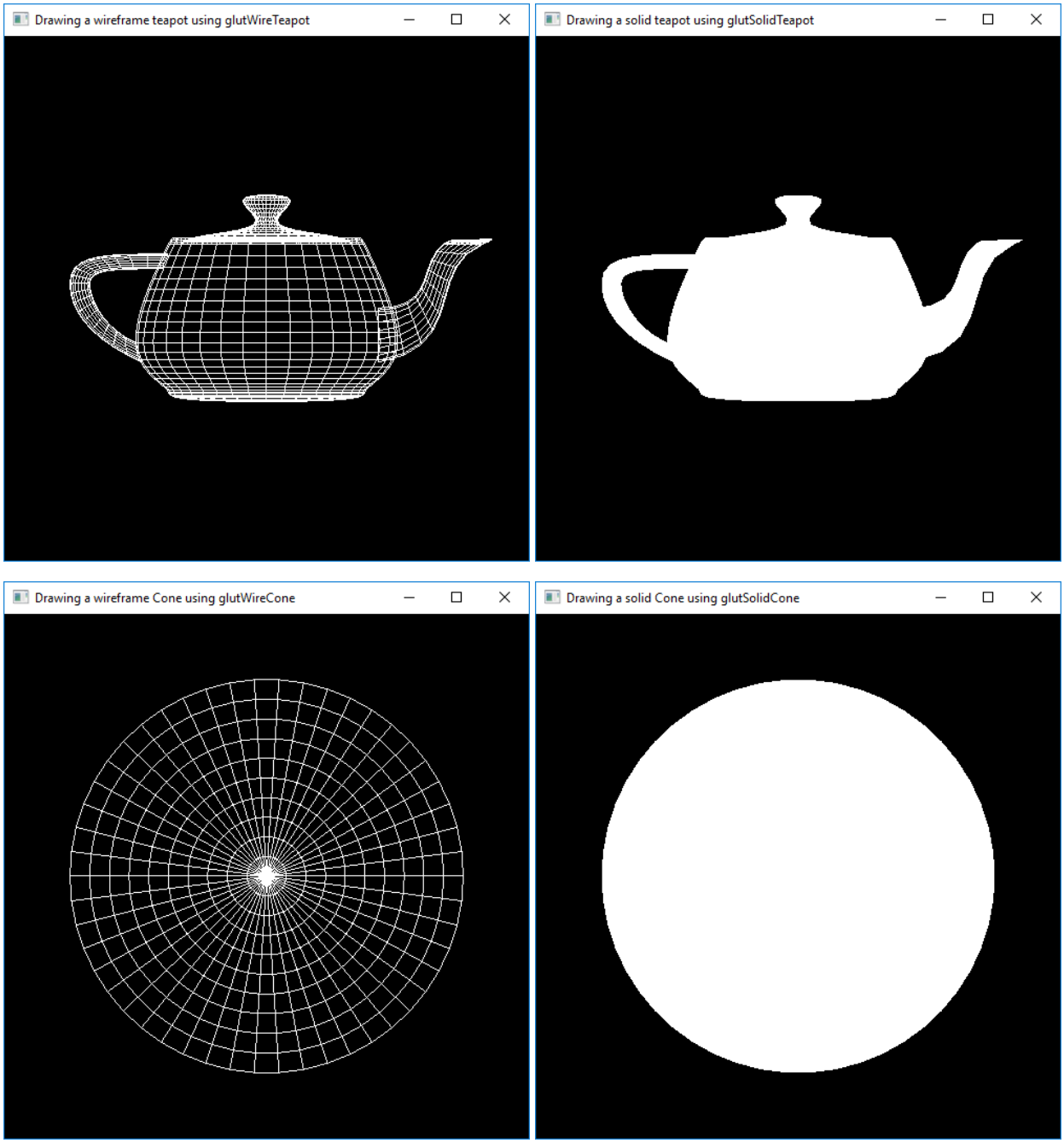
`glutSolidCone`: renders a solid cone (e.g. `glutSolidCone(0.75,0.75,50,10)`)

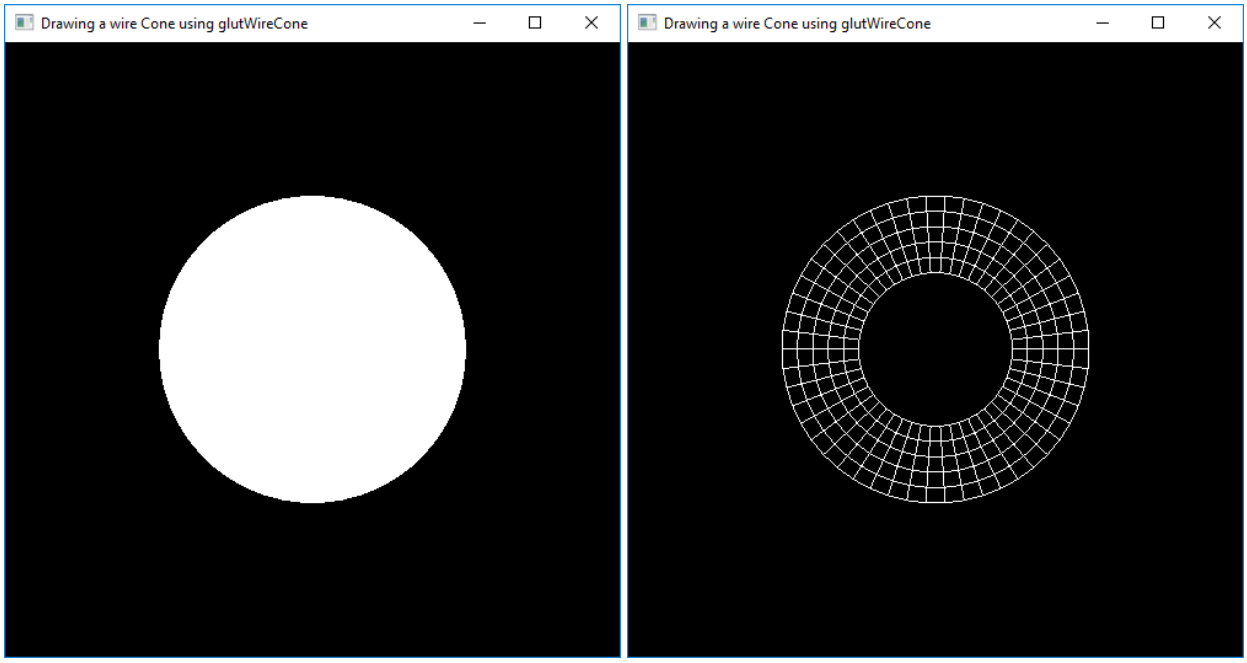
`glutWireCone`: renders a wireframe cone (e.g. `glutWireCone(0.75,0.75,50,10)`)

`glutSolidTorus`: renders a solid torus (e.g. `glutSolidTorus(0.25,0.5,50,10)`)

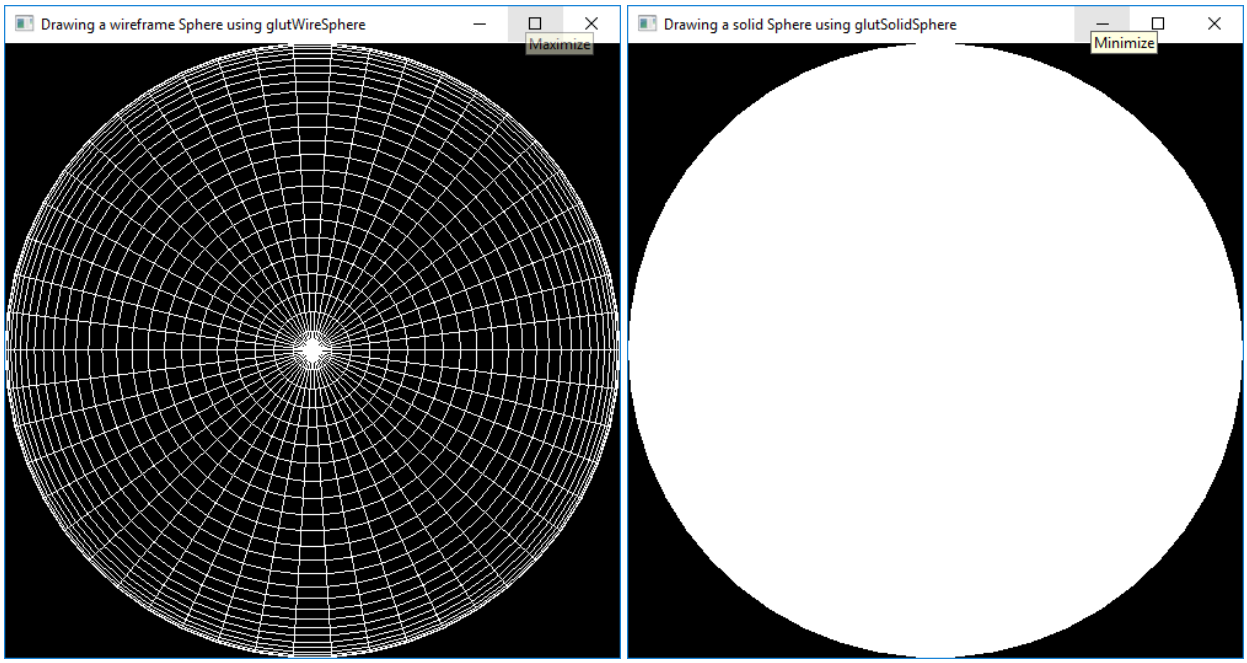
`glutWireTorus`: renders a wireframe torus (e.g. `glutWireTorus(0.25,0.5,50,10)`)

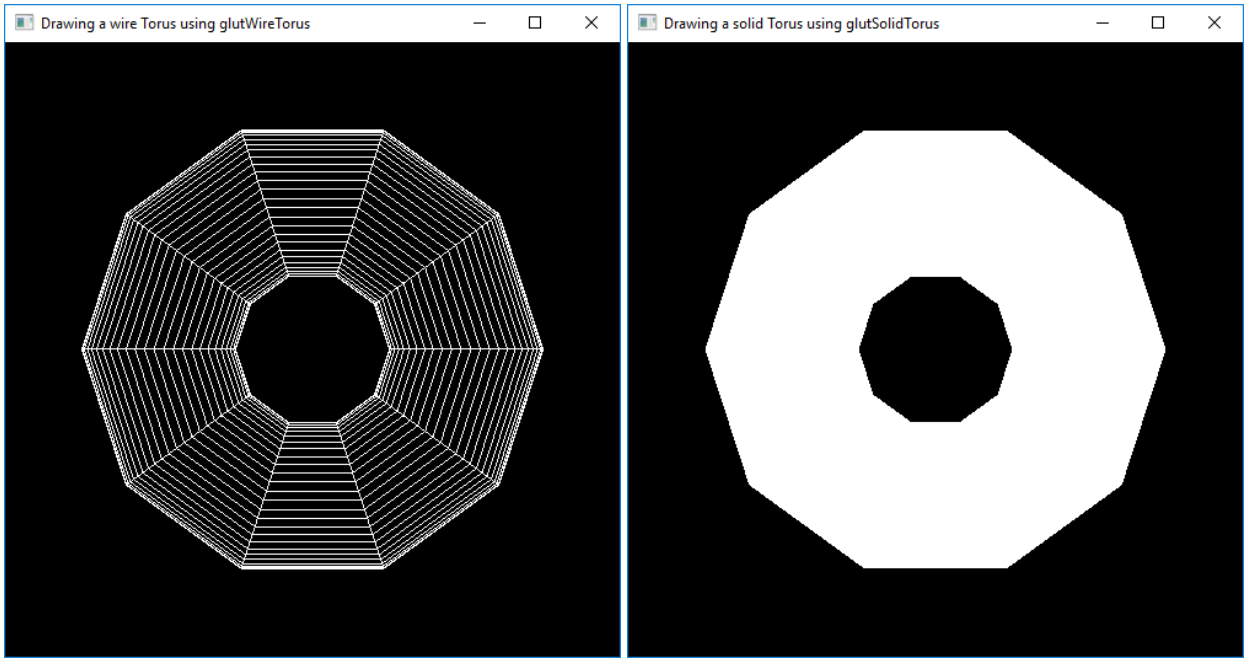
Some outputs:





N.B. A part of the wireframe cone was clipped because his height is greater than 1.





`glutWireSphere(radius, slices, stacks)`

`radius`: The radius of the sphere.
`Slices`: The number of subdivisions around the Z axis (similar to lines of longitude).
`Stacks`: The number of subdivisions along the Z axis (similar to lines of latitude).

`glutWireTeapot(size)`

`size`: Relative size of the teapot (a double value between 0 and 1)

`glutWireCube(size)`

The cube is centered at the modeling coordinates origin with sides of length `size`.

Size is double and must be less than 2 in order to fit in the NDC square.

`glutSolidCone(radius, height, slices, stacks)`

`base`: The radius of the base of the cone.
`Height`: The height of the cone.
`Slices`: The number of subdivisions around the Z axis.
`Stacks`: The number of subdivisions along the Z axis.
`glutWireTorus(innerRadius, outerRadius, nsides, rings)`

`innerRadius`: Inner radius of the torus.
`outerRadius`: Outer radius of the torus.
`Nsides`: Number of sides for each radial section.
`Rings`: Number of radial divisions for the torus.

2.2 Transformations (translation, rotation and scaling)

Introduction

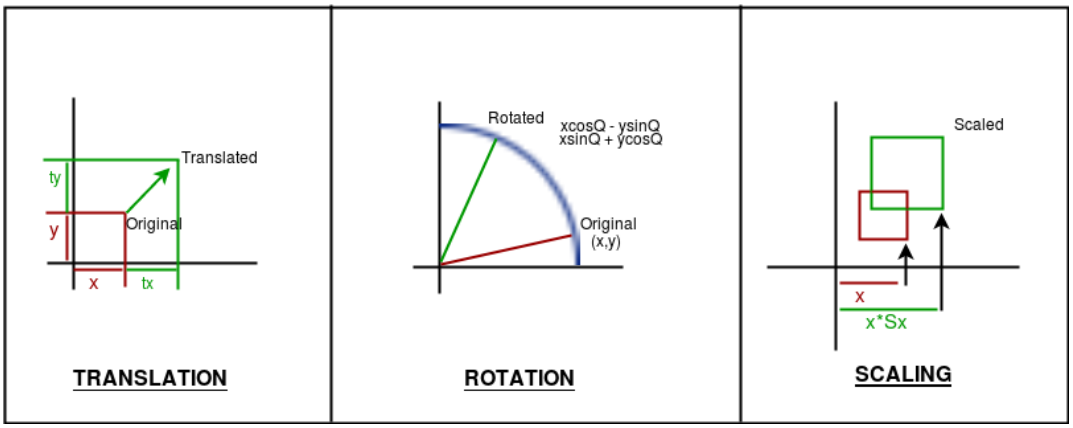
Transformations play a very important role in manipulating objects on screen. We can implement the algorithms corresponding to different types of transformation but we will prefer to use the OpenGL built-in functions instead.

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:

The following outputs show the triangle before and after translation:

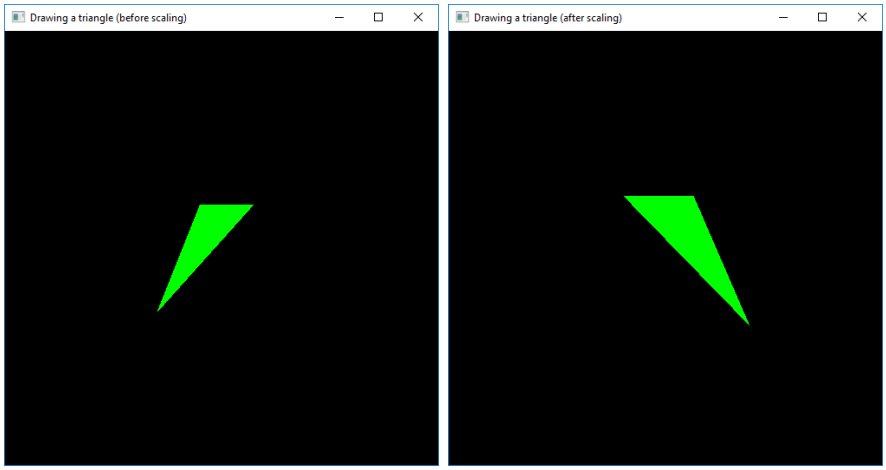
Question: what happens if we click or move or resize the OpenGL window? And why?

Example no. 2:

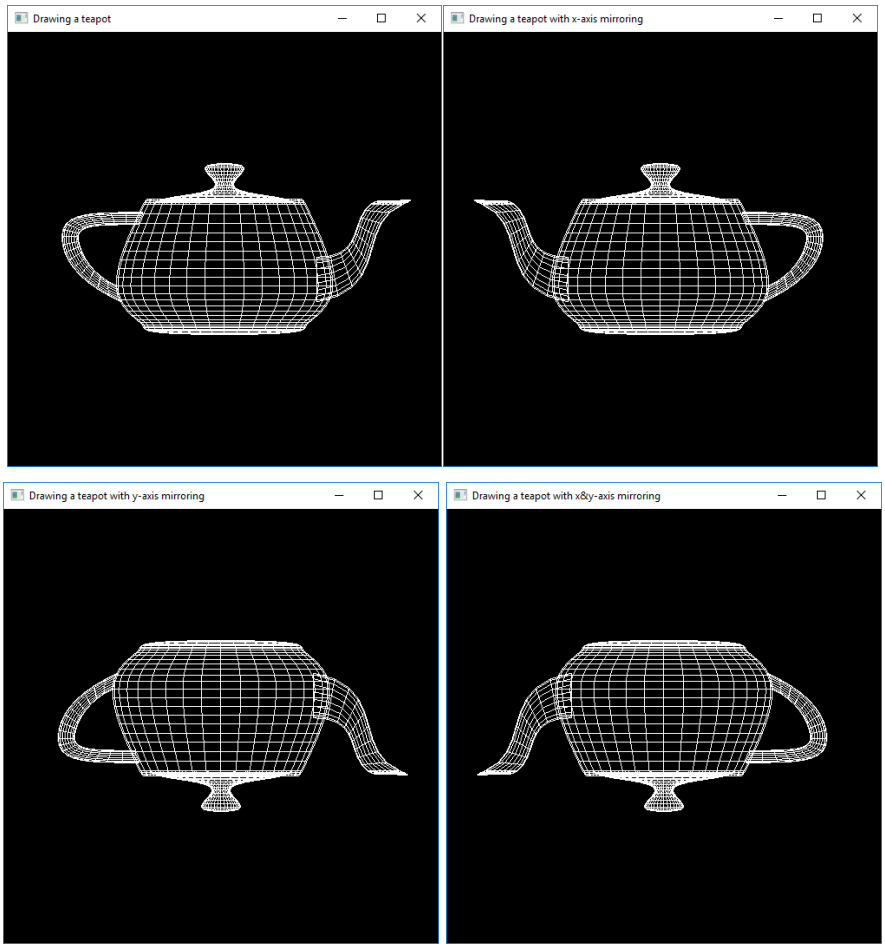
The following code draws a triangle, then scales it with the scaling factors across x, y and z axis equal to 1.05, 1.2 and 1 respectively, using `glScale3f` function:

The following outputs show the triangle before and 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 -1.3, 1.2 and 1 respectively will give the following output:

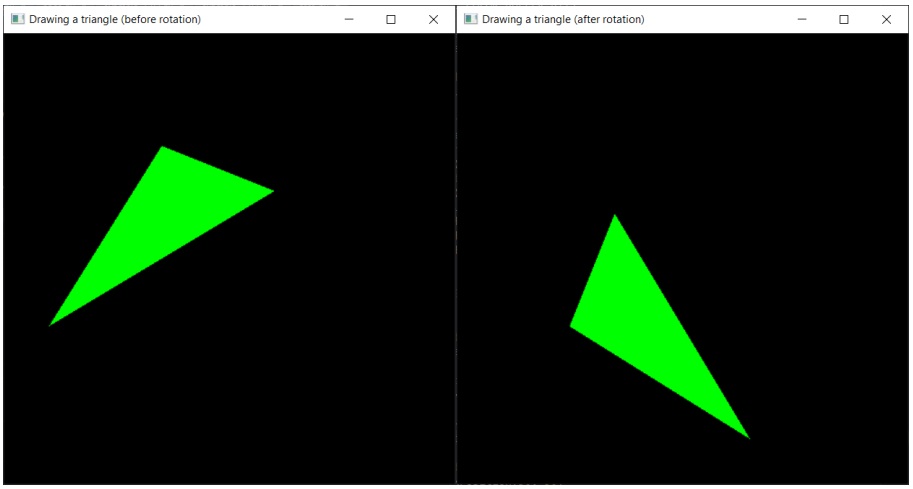


We can use negative scaling to perform shape mirroring with respect to a given axe/plane. For example we can mirror a teapot with respect to x-axis (-1,1,1), y-axis (1,-1,1) or x&y-axis simultaneously (-1,-1,1)



Example no. 3:

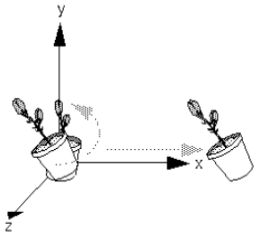
The following code draws a triangle then rotates it 90 degrees (anticlockwise rotation) about the z-axis (vector (0,0,1)) using `glRotate3f` function:



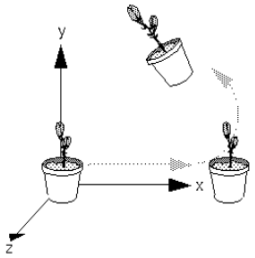
Note that the rotation angle may be negative. For example, the rotation of the same previous triangle with an angle of -90 degrees (clockwise rotation) about the z-axis (vector (0,0,1)) will give the following output:

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.



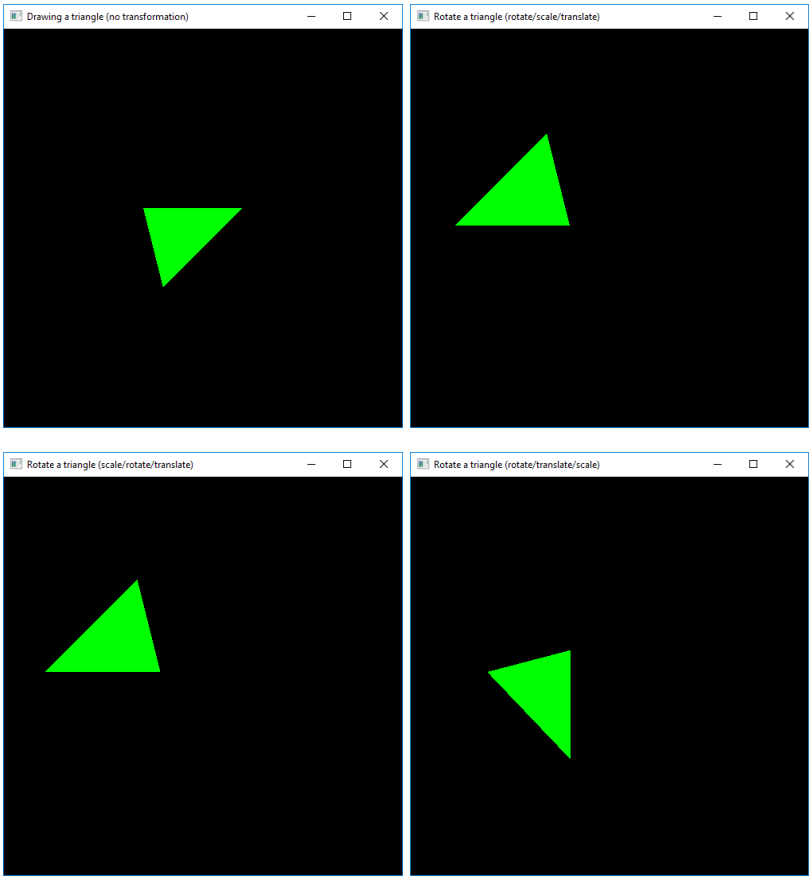
Rotation first



Translation first

N.B. Transformations are stacked (LIFO)

Outputs:



Practice

Modify the previous code so that the three transformations can be made separately in each call to the `render_Scene()` function.

Hint: Use a global switching variable, updated during each call to the `render_Scene()` function, to choose one among the three transformations.

Animations using transformations

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.

Write 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:

Practice

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 code given above 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), it changes its direction (from the right to the left) and vice versa.

Exercice

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.

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).

Multiple shapes with multiple transformations

In the previous sections, if we make a transformation, it will be applied to all the shapes in the scene (which is the default behavior).

In order to make different transformations for each shape/object in the scene, we must push the matrix of the transformation in the stack of matrices using the function `glPushMatrix()`, then we call the function `glLoadIdentity()` to "reset" the current matrix by replacing it with the identity matrix before popping the pushed transformation matrix from the stack using the function `glPopMatrix()`.

Example 1: both red and yellow triangles will be translated

Write the code that draw 2 triangle and only the first/red triangle will be translated

Modify the code given on the white board that the first/red triangle shifts to the bottom of the window and the second/yellow triangle shifts to its top.