Lab 2. Drawing basic shapes in OpenGL

Introduction

OpenGL can draw only a few basic shapes, including points, lines, and triangles. There is no built-in support for curves or curved surfaces; they must be approximated by simpler shapes. The basic shapes are referred to as primitives. A primitive in OpenGL is defined by its vertices.

The code we are going to write in the following will be based on the template file (framework) we created during the last lab (file template.py). More particularly, our task is to write the code that draws the scene in the user defined `render_Scene()` function.

Drawing points

There is only one kind of point primitive: GL_POINTS. This will cause OpenGL to interpret each individual vertex in the stream as a point. The function related to drawing points in OpenGL are:

- `glVertex2f()` - specify a vertex. The parameters are x-coordinate and y-coordinate on the 2D Plane.
- `glVertex3f()` - specify a vertex. The parameters are x-coordinate, y-coordinate and z-coordinate on the 3D Plane.
- `glColor3f()` - Sets the color of the point. The parameters are red, green and blue (RGB).
- `glPointSize()` - Sets the size of the point.

The basic code that allows to draw a single point is the following:

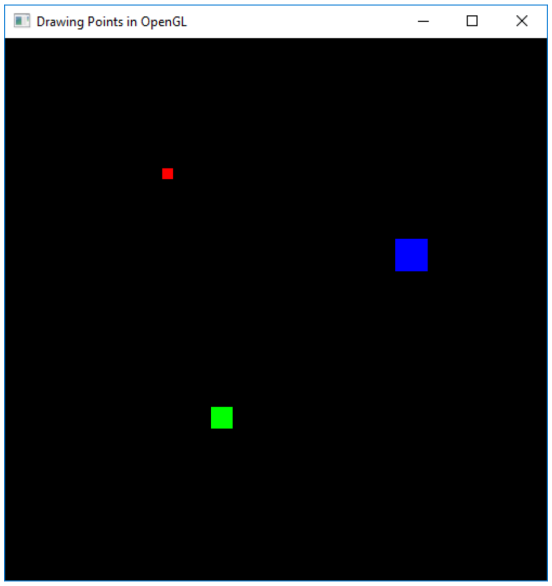
```
glBegin(GL_POINTS)
glVertex2f(0.0, 0.0)
glEnd()
```

To draw several points, the previous bloc of code must be repeated as many times as necessary.

For example, the code that allows to draw three points with different positions, colors and sizes is given below (render_Scene() function).

```
def render_Scene():

    # Draw the first point (red)
    glColor3f(1.0,0.0,0.0)
    glPointSize(10.0)
    glBegin(GL_POINTS)
    glVertex2f(-0.4, 0.5)
    glEnd()

    # Draw the second point (green)
    glColor3f(0.0,1.0,0.0)
    glPointSize(20.0)
    glBegin(GL_POINTS)
    glVertex2f(-0.2, -0.4)
    glEnd()
    # Draw the third point (blue)
    glColor3f(0.0,0.0,1.0)
    glPointSize(30.0)
    glBegin(GL_POINTS)
    glVertex2f(0.5, 0.2)
    glEnd()
```

The corresponding output is:



Drawing a circle using points

We can draw more complex shapes using points.

For example, we can draw a circle by drawing its individual points within a loop after specifying their coordinates using the well-known following formulae (and changing the value of the angle α in the interval [0,360] to scan a whole circumference)

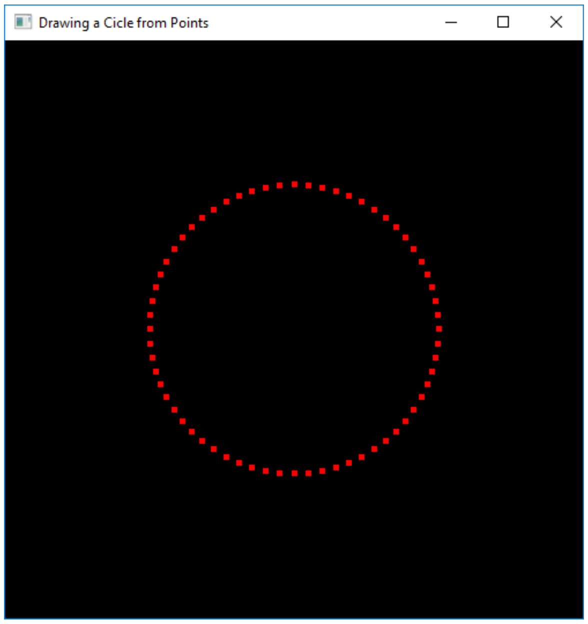
$$x = r\cos(\alpha)$$

$$y = r\sin(\alpha)$$

The corresponding Python code is the following (render_Scene() function):

```
def render_Scene():
    # Set current color to red
    glColor3f(1.0,0.0,0.0)
    glPointSize(5.0)

    for i in range(64): # We draw 64 points
        angle = 2*math.pi * i / 64
        x = 0.5 * math.cos(angle)
        y = 0.5 * math.sin(angle)
        glBegin(GL_POINTS)
        glVertex2f(x, y)
        glEnd()
```

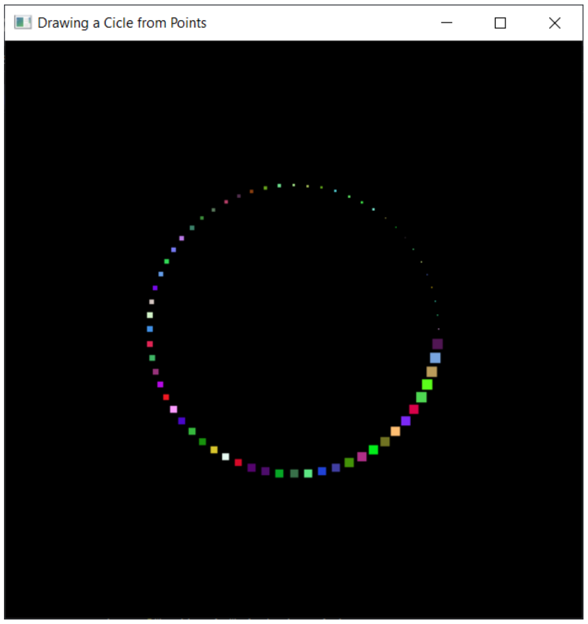
The corresponding output is:



Exercise

Modify the previous code so that the sizes of the points increase progressively from 1 (first point) to 10 (last point) and their colors are randomly chosen.

An example of the expected output is:



Drawing lines

There are three primitives for drawing line segments: GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP. GL_LINES draws disconnected line segments; specify two vertices for each segment that we want to draw. The other two primitives draw connected sequences of line segments. The only difference is that GL_LINE_LOOP adds an extra line segment from the final vertex back to the first vertex.

- `glVertex2f()` - specify the vertices of the segment. The parameters are x-coordinate and y-coordinate on the 2D Plane. Two vertices must be specified for each segment.
- `glVertex3f()` - specify a vertex. The parameters are x-coordinate, y-coordinate and z-coordinate on the 3D Plane. Two vertices must be specified for each segment.
- `glColor3f()` - Sets the color of the line. The parameters are red, green and blue (RGB).
- `glLineWidth()` — specify the width of rasterized lines

The basic code that allows to draw a line segment is the following:

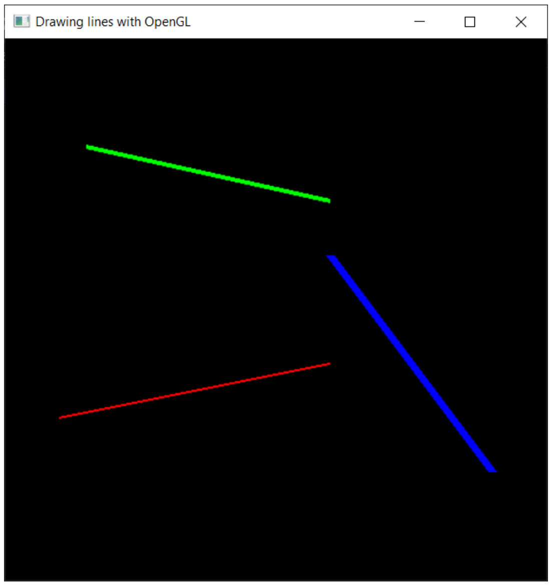
```
glBegin(GL_LINES);
glVertex2f(-0.4, 0.6);
glVertex2f(0.2, 0.2);
glEnd();
```

To draw several lines, the previous bloc of code must be repeated as many times as necessary.

For example, the code that allows to draw three lines with different positions, colors and widths is given below (render_Scene() function).

```
def render_Scene():
    # Draw the first line (in red)
    glColor3f(1.0,0.0,0.0);
    glLineWidth(2.0);
    glBegin(GL_LINES);
    glVertex2f(-0.8, -0.4);
    glVertex2f(0.2, -0.2);
    glEnd();

    # Draw the second line (in green)
    glColor3f(0.0,1.0,0.0);
    glLineWidth(4.0);
    glBegin(GL_LINES);
    glVertex2f(-0.7, 0.6);
    glVertex2f(0.2, 0.4);
    glEnd();

    # Draw the third  line (in blue)
    glColor3f(0.0,0.0,1.0);
    glLineWidth(8.0);
    glBegin(GL_LINES);
    glVertex2f(0.8, -0.6);
    glVertex2f(0.2, 0.2);
    glEnd();
```

The corresponding output is:



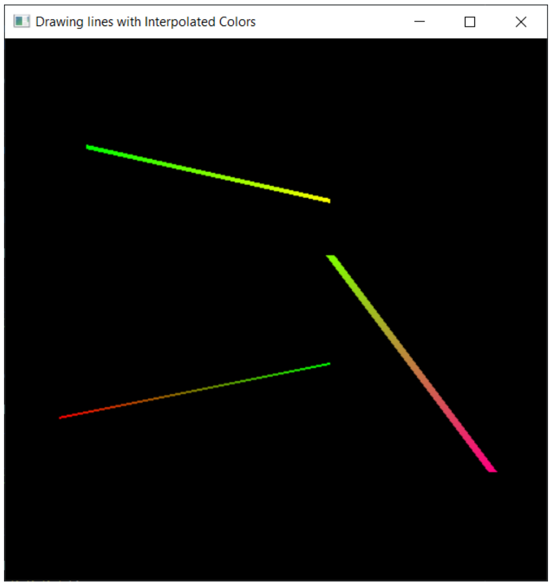
Colors interpolation when drawing lines

When the colors chosen for the two points of the line are different, OpenGL perform an interpolation to draw the line. That is, given two points, OpenGL will determine the set of pixels that the line would cross through and then change their color based on the color of the line. Bresenham's algorithm is a famous algorithm for drawing lines, and it (or some extension of it) is probably the one used internally by OpenGL.

For example, the following code draws the same lines of the previous paragraph choosing different colors for the vertices corresponding to each line (`render_Scene()` function).

```
def render_Scene():
    # Draw the first line
    glLineWidth(2.0);
    glBegin(GL_LINES);
    glColor3f(1.0,0.0,0.0);
    glVertex2f(-0.8, -0.4);
    glColor3f(0.0,1.0,0.0);
    glVertex2f(0.2, -0.2);
    glEnd();

    # Draw the second line
    glLineWidth(4.0);
    glBegin(GL_LINES);
    glColor3f(0.0,1.0,0.0);
    glVertex2f(-0.7, 0.6);
    glColor3f(1.0,1.0,0.0);
    glVertex2f(0.2, 0.4);
    glEnd();
    # Draw the third line
    glLineWidth(8.0);
    glBegin(GL_LINES);
    glColor3f(1.0,0.0,0.5);
    glVertex2f(0.8, -0.6);
    glColor3f(0.5,1.0,0.0);
    glVertex2f(0.2, 0.2);
    glEnd();
```

The corresponding output is the following:



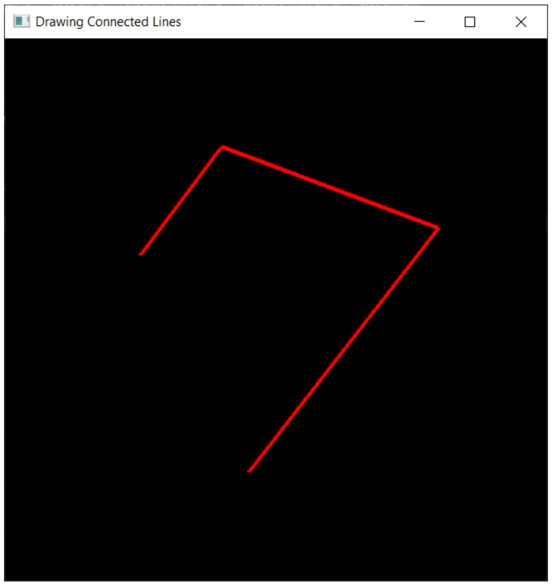
Draw connected sequences of line segments

To draw connected sequences of line segments, we must use the GL_LINE_STRIP primitive and then precise the set of vertices to connect.

An example of code is given below (`render_Scene()` function)

```
def render_Scene():
    glColor3f(1.0,0.0,0.0);
    glLineWidth(4.0);
    glBegin(GL_LINE_STRIP);
    glVertex2f(-0.5,0.2);
    glVertex2f(-0.2,0.6);
    glVertex2f(0.6,0.3);
    glVertex2f(-0.1,-0.6);
    glEnd();
```

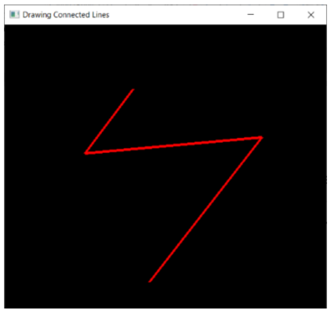
The corresponding output is the following:



N.B.

The order of adding the vertices matters in the form of the shape to be drawn. For example, if we change the order of appearance of the same four vertices of the previous paragraph, we will have another shape different than the previous one.

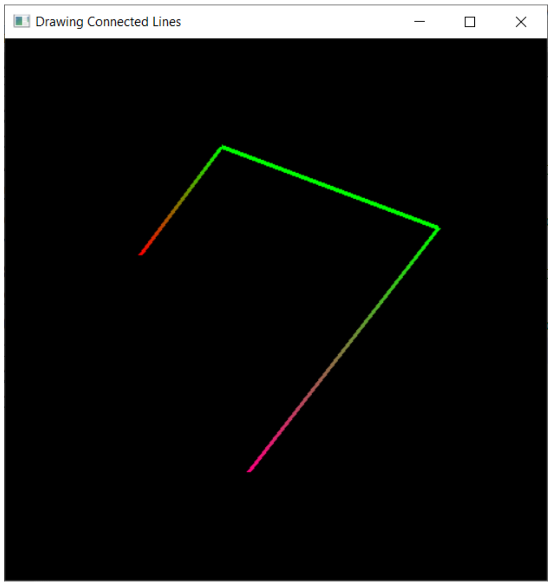
```
def render_Scene():
    glColor3f(1.0,0.0,0.0);
    glLineWidth(4.0);
    glBegin(GL_LINE_STRIP);
    glVertex2f(-0.2,0.6);
    glVertex2f(-0.5,0.2);
    glVertex2f(0.6,0.3);
    glVertex2f(-0.1,-0.6);
    glEnd();
```



The code version with different colors chosen for the vertices of each line is given below (`render_Scene()` function):

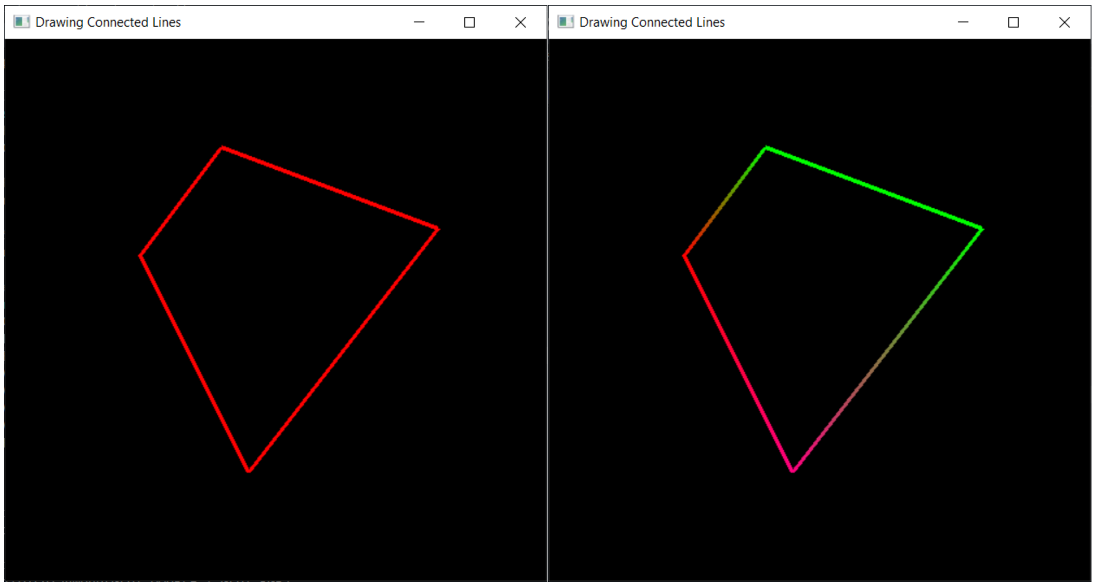
```
def render_Scene():
    glLineWidth(4.0);
    glBegin(GL_LINE_STRIP);

    glColor3f(1.0,0.0,0.0); # First point
    glVertex2f(-0.5,0.2);
    glColor3f(0.0,1.0,0.0); # Second point
    glVertex2f(-0.2,0.6);
    glColor3f(0.0,1.0,0.0); # Third point
    glVertex2f(0.6,0.3);
    glColor3f(1.0,0.0,0.5); # Fourth point

    glVertex2f(-0.1,-0.6);
    glEnd();
```

The corresponding output is:



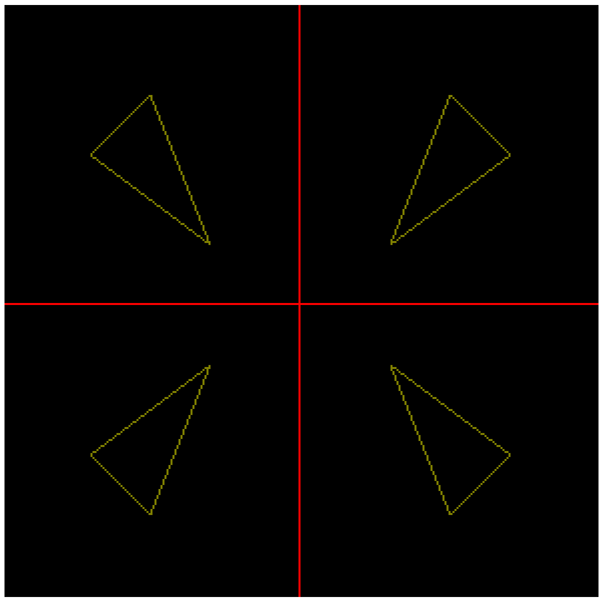
To allow drawing a connected group of line segments from the first vertex to the last, then back to the first, we must use the GL_LINE_LOOP primitive and then precise the set of vertices to connect.

The same examples of the previous paragraph replacing GL_LINE_STRIP with GL_LINE_LOOP will give the following outputs:



Exercises:

1. Draw an X-Y axis system using lines (in the middle of the drawing window)
2. Draw a triangle using closed-loop connected lines (GL_LINE_LOOP primitive) built around the following coordinates:
   x1, x2, x3 = 0.3, 0.7, 0.5
   y1, y2, y3 = 0.2, 0.5, 0.7
   Note that the triangle is located in the first quadrant of the X-Y plane
3. Draw the reflection of the triangle in all the remaining 3 quadrants of the X-Y Plane
   You should have a figure similar to the following one:



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 them. Since it 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, you 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.

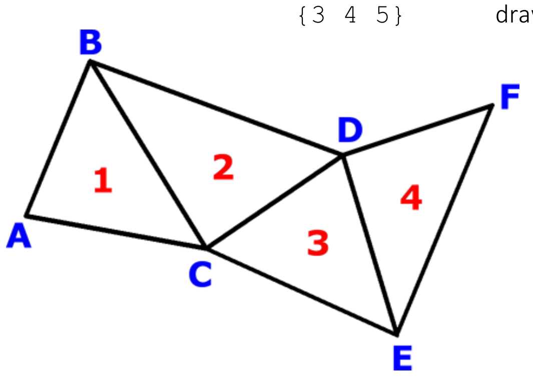
Here are some examples that can be more illustrative:

GL_TRIANGLES:

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

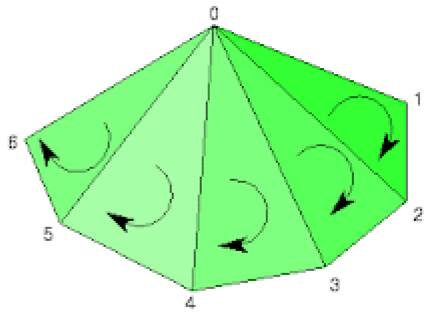
GL_TRIANGLE_STRIP:

```
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}
                   {3 4 5}         drawing order is (4 3 5) to maintain proper winding
```



GL_TRIANGLE_FAN:

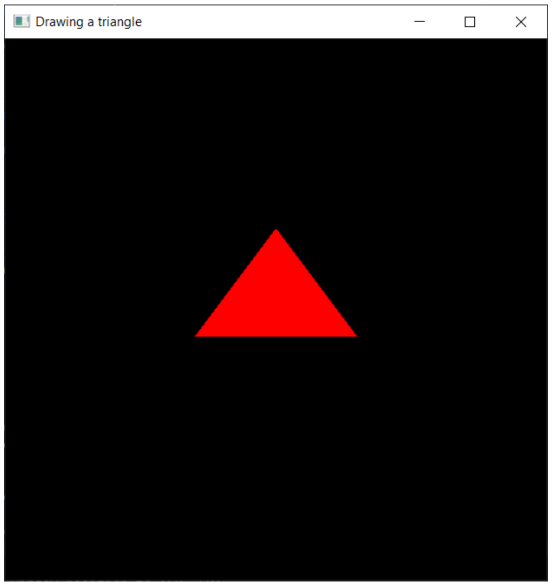
```
Indices:      0 1 2 3 4 5 6...
Triangles:  {0 1 2}
            {0} {2 3}
            {0}   {3 4}
            {0}     {4 5}
```



An example of the code that allows to draw a triangle using dedicated primitives is the following:

```
def render_Scene():
    glColor3f(1.0,0.0,0.0);
    glBegin(GL_TRIANGLES);
    glVertex3f(-0.3,-0.1,-0.8);
    glVertex3f(0.3,-0.1,-1);
    glVertex3f(0,0.3,-0.9);
    glEnd();
```

The corresponding output is the following:

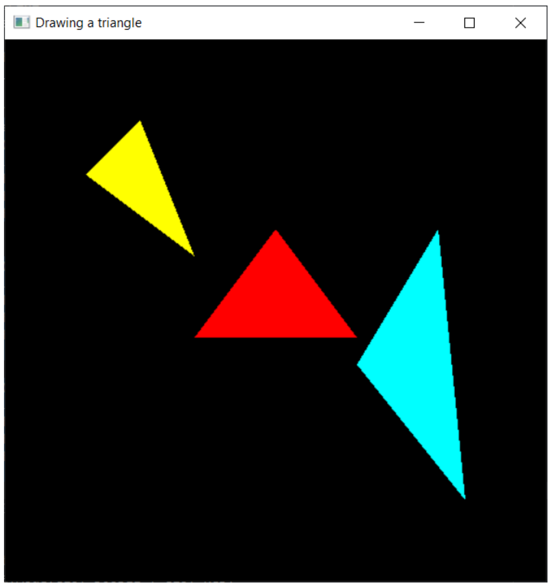


We can draw several triangles in the same windows by adding the bloc (glBegin/glEnd) as many times as needed.

For example, the following code allows to draw 3 triangles with different colors in the same window:

```
def render_Scene():
     # Draw the first triangle (in red)
   glColor3f(1.0,0.0,0.0)
    glBegin(GL_TRIANGLES)
    glVertex3f(-0.3,-0.1,-0.8)
    glVertex3f(0.3,-0.1,-1)
    glVertex3f(0,0.3,-0.9)
    glEnd()

    # Draw the second triangle (in yellow)
    glColor3f(1.0,1.0,0.0)
    glBegin(GL_TRIANGLES)
    glVertex3f(-0.3,0.2,0.0)
    glVertex3f(-0.7,0.5,0.0)
    glVertex3f(-0.5,0.7,0.0)
    glEnd()

    # Draw the third triangle (in cyan)
    glColor3f(0.0,1.0,1.0)
    glBegin(GL_TRIANGLES)
    glVertex3f(0.3,-0.2,0.0)
    glVertex3f(0.7,-0.7,0.0)
    glVertex3f(0.6,0.3,0.0)
    glEnd()
```

The corresponding output is the following:

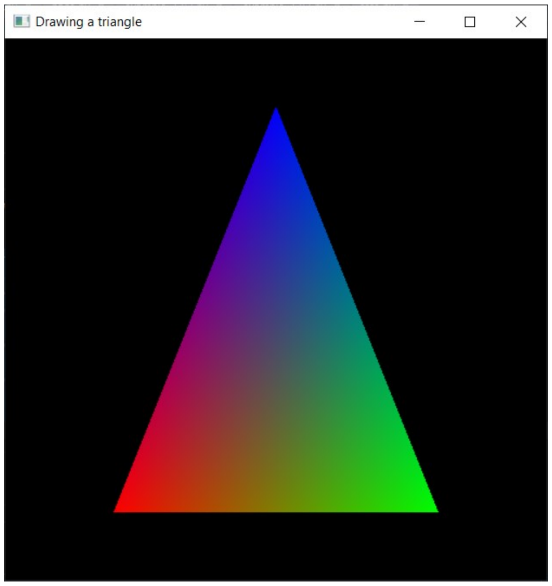


When using a different color for each of the vertices of the triangle, the colors of the inner pixels of the triangle will be determined using a barycentric interpolation.

The following example of code illustrates the principle:

```
def render_Scene():
    glBegin(GL_TRIANGLES);
    # Draw the first vertex (in red)
    glColor3f(1, 0, 0)
    glVertex3f(-0.6, -0.75, 0.5)
    # Draw the second vertex (in green)
    glColor3f(0, 1, 0)
    glVertex3f(0.6, -0.75, 0)
    # Draw the third vertex (in blue)
    glColor3f(0, 0, 1)
    glVertex3f(0, 0.75, 0)
    glEnd()
```

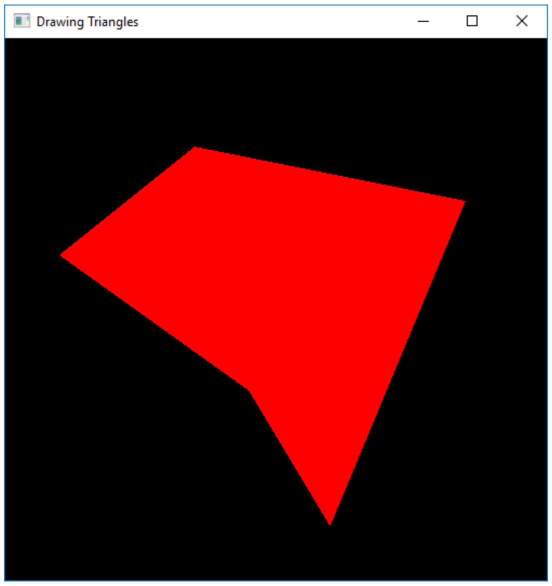
The corresponding output is the following:



Using the directive GL_TRIANGLE_STRIP with the same color for all vertices

```
def render_Scene():
    glBegin(GL_TRIANGLE_STRIP);
    glColor3f(1.0,0.0,0.0)
    glVertex3f(-0.8,0.2,0)
    glVertex3f(-0.3,0.6,0)
    glVertex3f(-0.1,-0.3,0)
    glVertex3f(0.7,0.4,0.0)
    glVertex3f(0.2,-0.8,0)
    glEnd()
```

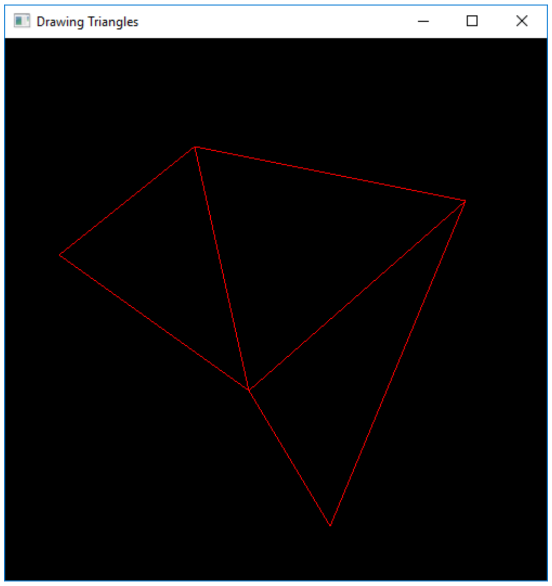
The corresponding output is:



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

```
def render_Scene():
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)
    glBegin(GL_TRIANGLE_STRIP)
    glColor3f(1.0,0.0,0.0)
    glVertex3f(-0.8,0.2,0)
    glVertex3f(-0.3,0.6,0)
    glVertex3f(-0.1,-0.3,0)
    glVertex3f(0.7,0.4,0.0)
    glVertex3f(0.2,-0.8,0)
    glEnd()
```

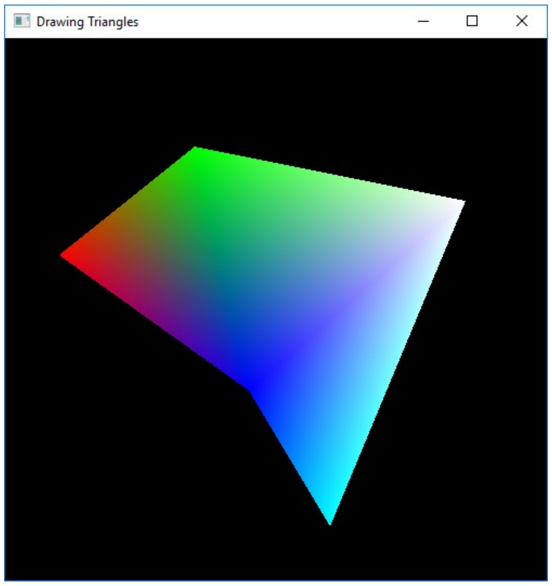
The corresponding output is:



Using the directive GL_TRIANGLE_STRIP with different colors for the vertices

```
def render_Scene():
    glBegin(GL_TRIANGLE_STRIP);
    glColor3f(1.0,0.0,0.0)
    glVertex3f(-0.8,0.2,0)
    glColor3f(0.0,1.0,0.0)
    glVertex3f(-0.3,0.6,0)
    glColor3f(0.0,0.0,1.0)
    glVertex3f(-0.1,-0.3,0)
    glColor3f(1.0,1.0,1.0)
    glVertex3f(0.7,0.4,0.0)
    glColor3f(0.0,1.0,1.0)
    glVertex3f(0.2,-0.8,0)
    glEnd()
```

The corresponding output is:

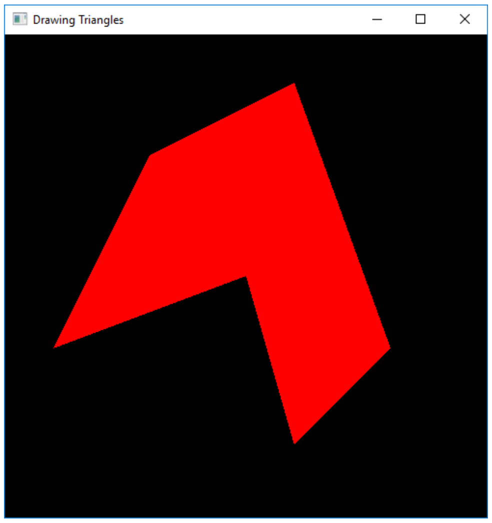


Using the directive GL_TRIANGLE_FAN with the same color for all the vertices (filled triangles)

Code:

```
def render_Scene():
    glBegin(GL_TRIANGLE_FAN);
    glColor3f(1.0,0.0,0.0)
    glVertex3f(0.0,0.0,0.0)
    glVertex3f(-0.8,-0.3,0.0)
    glVertex3f(-0.4,0.5,0.0)
    glVertex3f(0.2,0.8,0.0)
    glVertex3f(0.6,-0.3,0.0)
    glVertex3f(0.2,-0.7,0.0)
    glEnd()
```

Output:

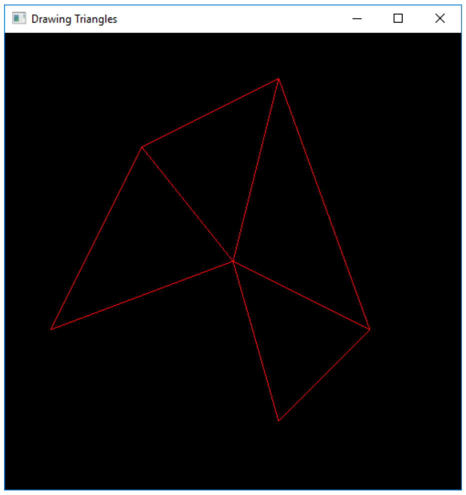


Using the directive GL_TRIANGLE_FAN with the same color for all the vertices (empty triangles)

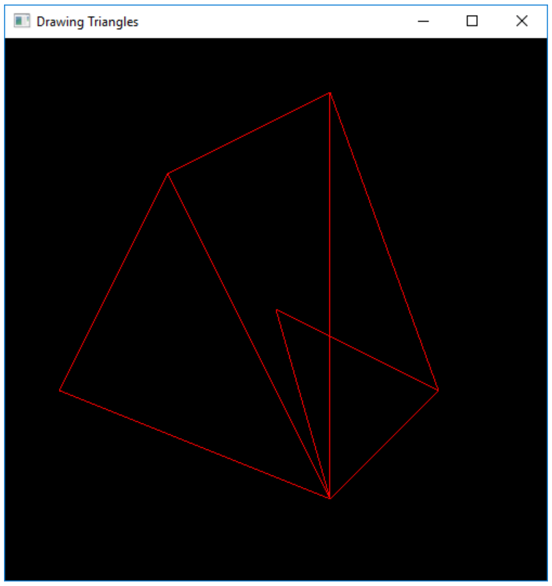
Code:

```
def render_Scene():
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)
    glBegin(GL_TRIANGLE_FAN)
    glColor3f(1.0,0.0,0.0)
    glVertex3f(0.0,0.0,0.0)
    glVertex3f(-0.8,-0.3,0.0)
    glVertex3f(-0.4,0.5,0.0)
    glVertex3f(0.2,0.8,0.0)
    glVertex3f(0.6,-0.3,0.0)
    glVertex3f(0.2,-0.7,0.0)
    glEnd()
```

Output:

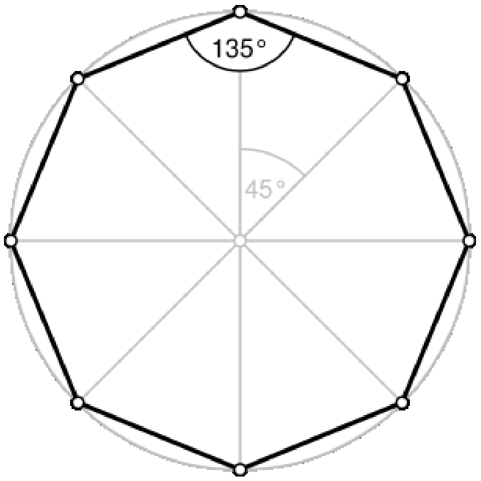


Be careful of the choice of the central point as well as the order of the points. For example, if we interchange the orders of the first and the last points in the previous example we will get the following output:



Exercise:

Draw a regular octagon (see figure below) using 8 triangles with the directive GL_TRIANGLE_FAN (the central point will be (0,0)



Other drawing primitives in OpenGL

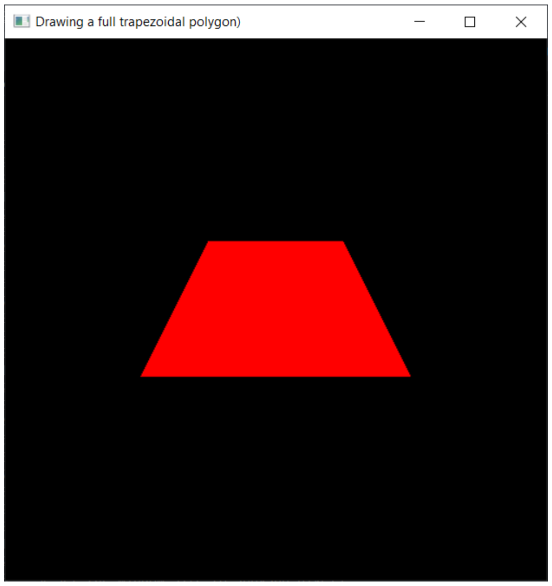
GL_QUADS

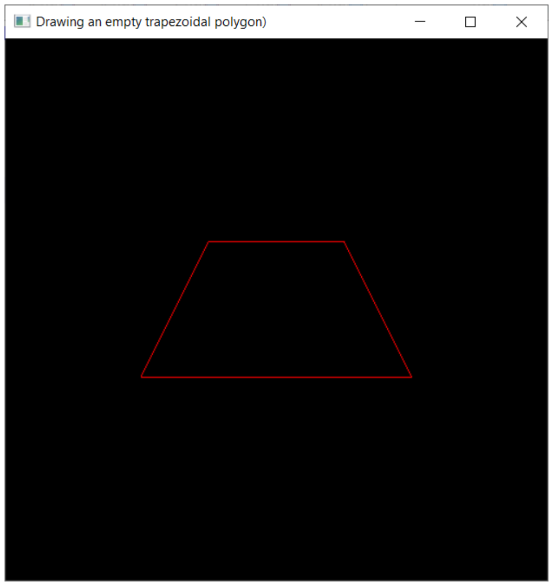
OpenGL has a special primitive called GL_QUADS that allows to draw a quadrilateral polygon knowing its four vertices.

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)`).

```
def render_Scene():
    glColor3f(1, 0, 0);
    glBegin(GL_QUADS);
    glVertex2f(-0.25, 0.25);
    glVertex2f(-0.5, -0.25);
    glVertex2f(0.5, -0.25);
    glVertex2f(0.25, 0.25);
    glEnd();
```

Output:



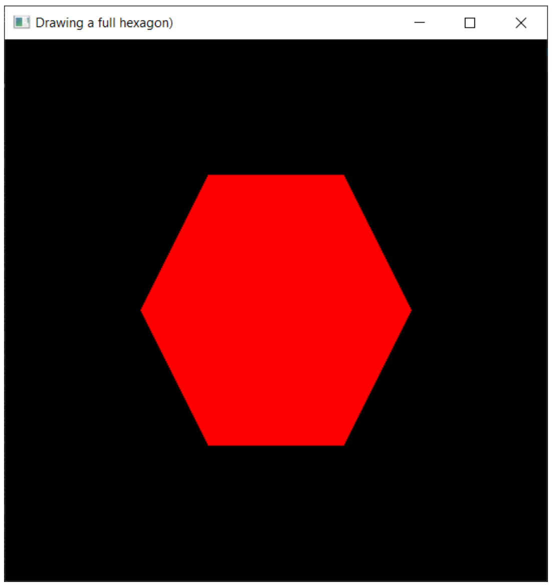


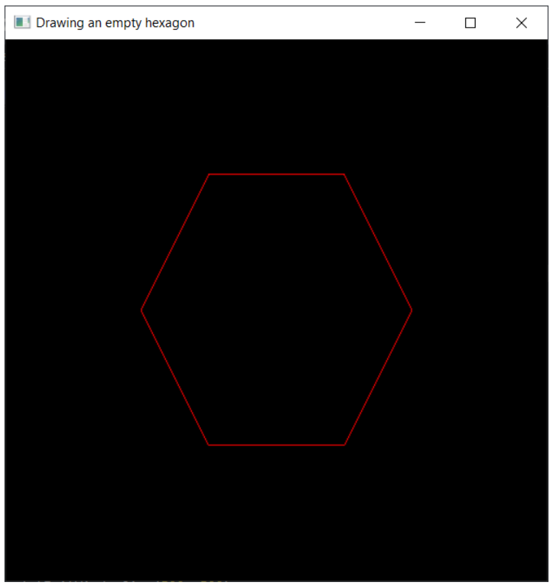
GL_POLYGON

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)`).

```
def render_Scene():
    glColor3f(1, 0, 0);
    glBegin(GL_POLYGON);
    glVertex2f(-0.25, 0.5);
    glVertex2f(-0.5, 0);
    glVertex2f(-0.25, -0.5);
    glVertex2f(0.25, -0.5);
    glVertex2f(0.5, 0);
    glVertex2f(0.25, 0.5);
    glEnd();
```





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(1,50,50)`)

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

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)`)

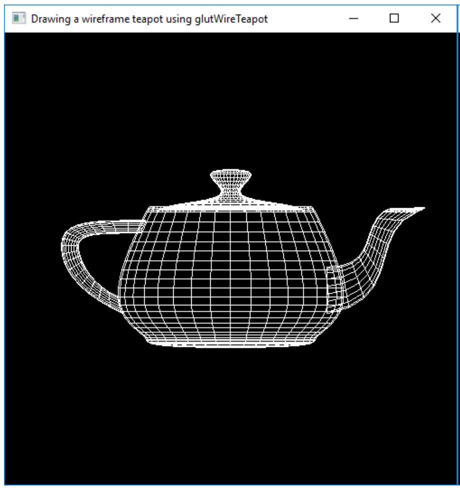
glutWireCube: renders a wireframe cube (e.g. `glutWireCube(1)`)

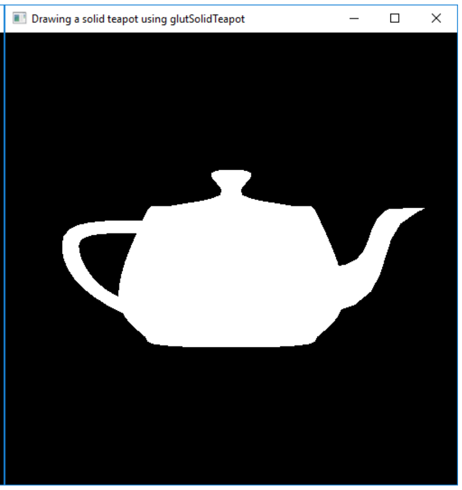
glutSolidCone: renders a solid cone (e.g. `glutSolidCone(1,2,50,10)`)

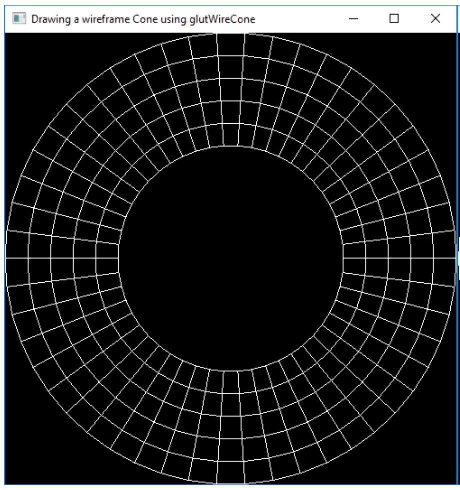
glutWireCone: renders a wireframe cone (e.g. `glutWireCone(1,2,50,10)`)

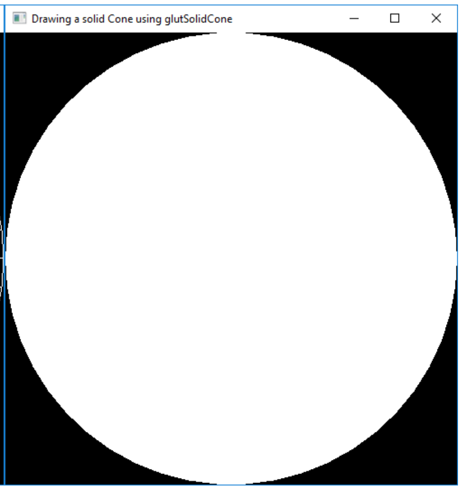
N.B. A part of the wireframe cone was clipped because his height is greater than 1.

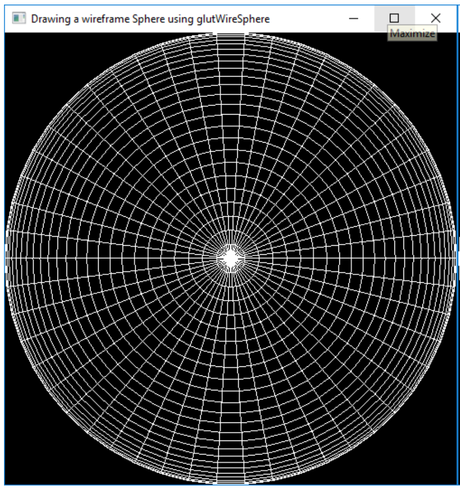
Some outputs:

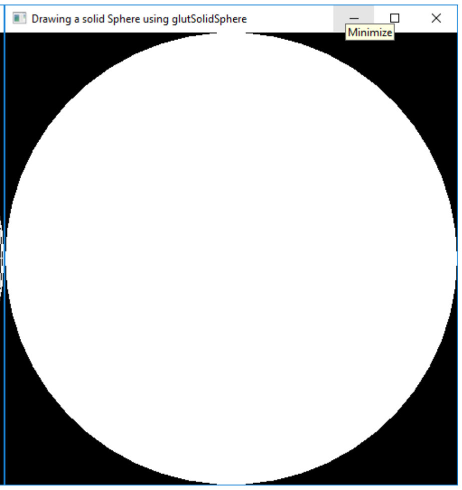

Drawing a wireframe teapot using glutWireTeapot


Drawing a solid teapot using glutSolidTeapot


Drawing a wireframe Cone using glutWireCone


Drawing a solid Cone using glutSolidCone


Drawing a wireframe Sphere using glutWireSphere
Maximize


Drawing a solid Sphere using glutSolidSphere
Minimize