Computer graphics - lab exercise 1

Make sure you've properly installed Visual Studio Code or you programming environment of choice. Install the packages PyOpenGL and pygame.

Make a new .py file that you may call TGRA_Lab1, or whatever fits your own naming scheme.

Start by adding the following code to the top of the file and make sure there are no errors or warnings:

```
import pygame
from pygame.locals import *

from OpenGL.GL import *
from OpenGL.GLU import *

import random
from random import *
```

Next make 4 functions:

```
def init_game():
   pass

def update():
   pass

def display():
   pass

def game_loop():
   update()
   display()
```

And finally ad this code to the bottom of the file:

```
if __name__ == "__main__":
   init_game()
   while True:
       game_loop()
```

This is done in this particular way so that we can isolate code that updates positions and orientations of game objects from code that displays to the screen. It's also good practice to keep all code that checks for input in a single place as well.

Let's start by initializing pygame, thus setting up a window that our graphics will be displayed in. Edit your init_game() operation thus:

```
def init_game():
   pygame.display.init()
   pygame.display.set_mode((800, 600), DOUBLEBUF|OPENGL)
```

This opens a window that is 800 pixels wide and 600 pixels high. You can run your program to see what happens.

Let's get to know the pygame event handling a little bit by adding code at the top of our game_loop function. It will now look like this:

```
def game_loop():
   for event in pygame.event.get():
       if event.type == pygame.QUIT:
           pygame.quit()
           quit()
       elif event.type == pygame.KEYDOWN:
           if event.key == K_ESCAPE:
               pygame.quit()
               quit()

   update()
   display()
```

These are two events. One is the event that the program is quitting for some reason, which is a good time to shut down pygame. The other one we will use a lot more, the KEYDOWN event. In this case the ESC key has been pressed, so we also shut pygame down.
Check the program again to see if these things work.

Now we want to make our graphics appear. We will use a very simple method of sending vertices into OpenGL. We will only use this method in 2D so that we can postpone our use of shaders a little bit, but it is nice to get things going in a simple manner now so we can start the graphics and motion right away.

Let’s start with the background. In our init_game function add:

```
   glClearColor(0.0, 0.0, 0.0, 1.0)
```

Then change display to:

```
def display():
   glClear(GL_COLOR_BUFFER_BIT)
```

Run the program now and see if it has changed.
See what happens if you change the values in glClearColor. These values range from 0.0 (0%) to 1.0 (100%).

Now add a bit of code to display:

```
   glMatrixMode(GL_PROJECTION)
   glLoadIdentity()
   glMatrixMode(GL_MODELVIEW)
   glLoadIdentity()

   glViewport(0, 0, 800, 600)
   gluOrtho2D(0, 800, 0, 600)
```

The top part is about initializing matrices and we’ll ignore that for a while. It just needs to be there.
The bottom part is setting up a viewport (actual dimensions on the screen) and our orthographic viewing window (arbitrary dimensions that we can decide). Here they are the same dimensions, 800x600, but the Orthographic window can be anything we want. It can pay off to experiment with that once we start drawing images.
This code should, for now, always be in the beginning of display()

The following code should be at the end of display():

```
   pygame.display.flip()
```

This tells our graphics hardware that we are done drawing into one of the frame buffers and that the screen can now display what we drew, while we will continue drawing into the other frame buffer.

Other code that we add to display() should come between the two.

Make sure everything runs correctly and then let’s start drawing shapes. Add this to display():

```
    glBegin(GL_TRIANGLES)
    glVertex2f(100, 100)
    glVertex2f(100, 200)
    glVertex2f(200, 100)
    glEnd()
```

This tells the OpenGL pipeline that will begin sending vertices into the pipeline and that they should be interpreted as separate triangles. Then we just send three vertices in (2f means we’re just using x and y coordinates, z is ignored (or always 0)) so there will only be one triangle.
Try adding another triangle so that this will be a square. You don’t need to call begin and end again, just add the other triangle to the vertices.

Now you can try, instead of GL_TRIANGLES, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN.
You can edit the order in which the vertices are used to better understand the difference between these operations. Add vertices and experiment, experiment, experiment.

If nothing is displaying or you want to change the color of your drawing, add:

```
    glColor3f(1.0, 1.0, 1.0)
```

Make sure you add it before you draw the vertices. Every change you do to variables in the pipeline affect all vertices that go through it from then on, not the ones already sent in.

Now you can repeat glColor() and glBegin()/glEnd() as many times as you wish in your code, change the coordinates of the vertices and draw shapes all over the screen.

Get things moving

Now let's go back to our game_loop operation and the keyboard events. Add:

```
           elif event.key == K_q:
               glClearColor(random(), random(), random(), 1.0)
```

Next let's add a positional variable and a boolean switch.
Add a variable that can be called x_pos and another that can be called moving_left so that they are accessible by all the functions. You can make a class that all the operations are part of or you can initialize the variables at the top and then use the global keyword.
Top:

```
going_left = False
x_pos = 100
```

In any function that needs the variables:

```
def update():
   global x_pos
   global going_left
```

This is just one way to do this, so feel free to design your code the way you like.

Now, add the following to the events in game_loop:

```
           elif event.key == K_LEFT:
               going_left = True
       elif event.type == pygame.KEYUP:
           if event.key == K_LEFT:
               going_left = False
```

And add to update():

```
   if going_left:
       x_pos -= 0.5
```

Now the variables should be changing, but they're not affecting the drawing, so change at least one of your shapes so that their vertices are affected by the variable:

```
   glBegin(GL_TRIANGLES)
   glVertex2f(x_pos, 100)
   glVertex2f(x_pos, 200)
   glVertex2f(x_pos + 100, y_pos)
   glEnd()
```

Now try the mouse button event:

```
        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                x_pos = pygame.mouse.get_pos()[0]
                y_pos = pygame.mouse.get_pos()[1]
```

You can use different variables or the same. Edit your shape so that the y_pos also affects it.

Try other keys and other buttons. Try setting up more shapes. Try setting up lists of positions so that you can draw multiple boxes in a single loop.

Try drawing a circle. Set up loop that increments a variable from 0 to 6.283 (2*PI, 360°) and then apply cos() and sin() to that variable and use as x and y coordinates for vertices. Use GL_LINE_LOOP or GL_TRIANGLE_FAN. Note the size. If the coordinates are not multiplied the radius of the circle is 1. How big is that in your game?

Can you change the dimensions of the orthographic Window and see the results. It is the coordinate frame that you are drawing into. The Viewport is the area on the actual screen. See how these work and how you can manipulate them.

Can you draw other mathematical functions.
Increment a variable n linearly and use n as x-coords and sin(n) as y-coords. Change the gluOrtho2D variables so that they represent an area that the function fits in. or you can multiply the results to fit into 800x600.

Experiment, EXPERIMENT, ***EXPERIMENT!!!***