Transformation

Transformation is generally a change applied to a certain object. For example, change in size of an object, change in orientation (Rotation).

On this exercise, we will see a simple rotation of the box we drew on Exercise 1.

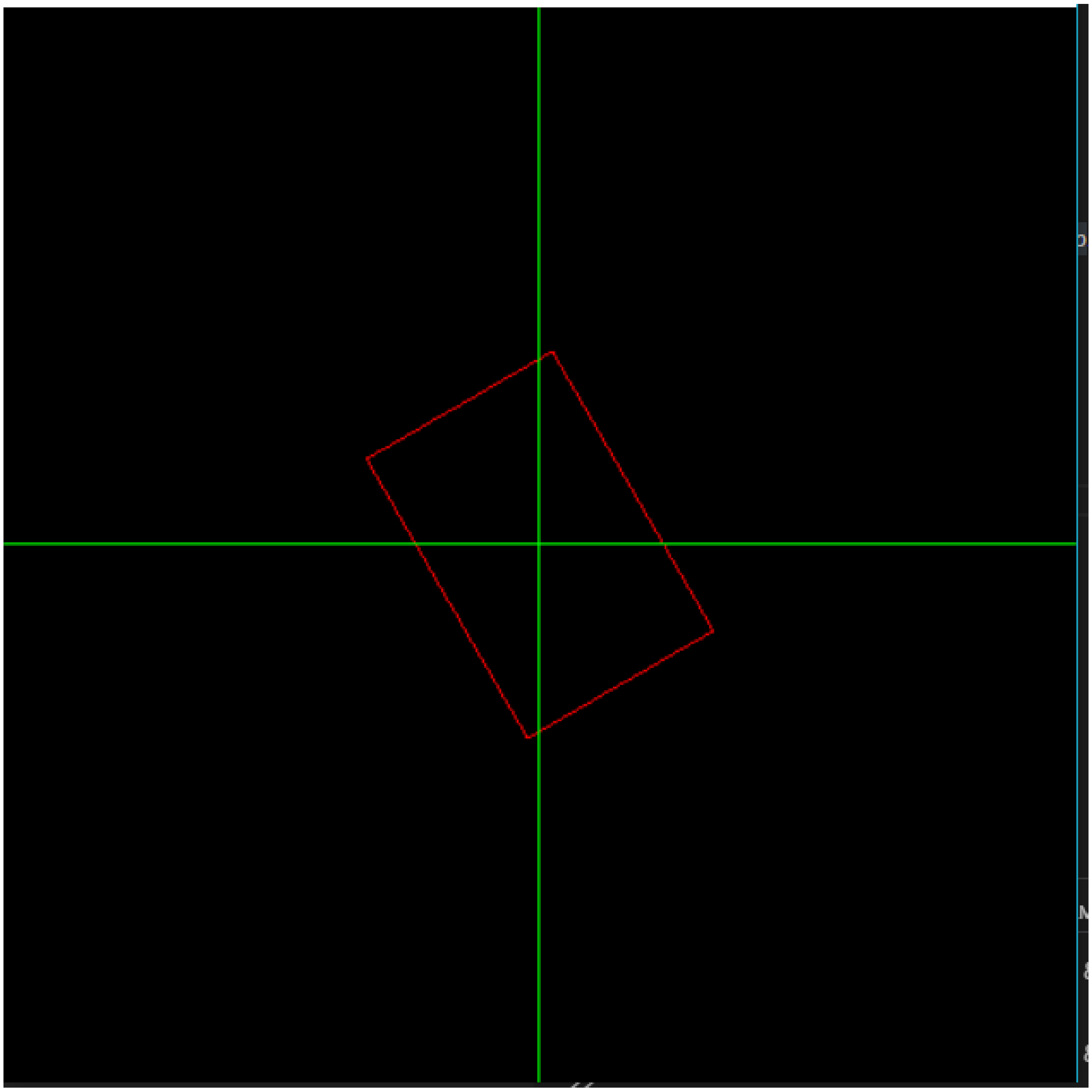
Import everything from transorm.py in your exercise 1 python file.

In transform.py we have a function rotationMat function which returns a rotation matrix. A rotation matrix is a matrix that rotates an object along an axis when it is multiplied with the object’s vertices.

Example. Let P be a point and M be a rotation matrix.
P X M = P rotated

So to rotate an object, we simply need to multiply all our points with a rotation matrix.
The rotationMat function in transform.py accepts parameters of degree by which we rotate an object.

Exercise 2:
Rotate the box you have made on Exercise 1
a. Create a variable mat and assign it to rotation matrix of 60’
b. Using np.dot function multiply the points of the box with mat
c. Add .4 to all points and see what happens.



DRAWING A CUBE (3D)

So, the way OpenGL works is you just specify the objects within space. For a cube, for example, you specify the "corners." Corners are referred to as vertices (plural).

Once you define the vertices, you can then do things with them. In this example, we want to draw lines between them. Defining the vertices is done with a simple list or tuple in Python. You can then pre-define some rules like what vertices make up a "surface" and between

Once you have everything, go ahead and open up IDLE and type in:

```
import pygame
import OpenGL
```

If you can type those statements and run them without any errors, then you are ready to proceed. If you are getting errors, something went wrong. Most of the time, the error is either you've downloaded the wrong Python version of PyGame or OpenGL, or the wrong bit version.

If you still have the import pygame and import OpenGL code, erase that and start completely blank.

First, we're going to do some imports:

```
import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *
```

We're importing all of PyGame here, and then all of the PyGame.locals. This is some typical PyGame code.

Next, we import OpenGL.GL and OpenGL.GLU. OpenGL.GL is just your typical OpenGL functions, then OpenGL.GLU is some of the more "fancy" OpenGL functions.

```
vertices= (
    (1, -1, -1),
    (1, 1, -1),
    (-1, 1, -1),
    (-1, -1, -1),
    (1, -1, 1),
    (1, 1, 1),
    (-1, -1, 1),
    (-1, 1, 1)
    )
```

Here, we've defined the location (x,y,z) of each vertex. I think it is best to envision this in "units." Try to think of these locations "spatially." With a cube, there are 8 "nodes" or vertices.

Next, we're ready to define the edges:

```
edges = (
    (0,1),
    (0,3),
    (0,4),
    (2,1),
    (2,3),
    (2,7),
    (6,3),
    (6,4),
    (6,7),
    (5,1),
    (5,4),
    (5,7)
    )
```

Each of the above tuples contains two numbers. Those numbers correspond to a vertex, and the "edge" is going to be drawn between those two vertices. We start with 0, since that's how Python and most programming languages work (the first element is 0). So, 0 corresponds to the first vertex we defined (1, -1, -1)... and so on.

Now that we've got that, let's work on the required code to work with OpenGL to actually generate a cube:

```
def Cube():
    glBegin(GL_LINES)
    for edge in edges:
        for vertex in edge:
            glVertex3fv(vertices[vertex])
    glEnd()
```

First, we start off our function as we would any other function.

Next, since this is just a function containing OpenGL code, we go ahead and open with a glBegin(GL_LINES), this notifies OpenGL that we're about to throw some code at it, and then the GL_LINES tells OpenGL how to handle that code, which, in this case, means it will treat the code as line-drawing code.

From there, we say for edge in edges, which corresponds to each pair of vertices in our edges list. Since each edge contains 2 vertices, we then say for vertex in edge, do glVertex3fv(vertices[vertex]), which performs the glVertex3fv OpenGL function on the [vertex] element of the vertices tuple.

As such, what ends up being passed through OpenGL with the constant of GL_LINES is:

glVertex3fv((1, -1, -1))

glVertex3fv((1, 1, -1))

...and so on. OpenGL, knowing that we're drawing lines here will draw lines between those points.

After running through all edges, we're done, so we call glEnd() to notify OpenGL that we're done telling it what to do. For each "type" of OpenGL code that you plan to use, you will need opening and closing GL commands like this.

That's it for our cube function. This function will create the cube, but now we want to display the cube and specify our perspective in the environment:

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

This is mostly typical PyGame code.

The only major difference here is we're adding another "parameter" looking thing after "display" in the pygame.display.set_mode. These are actually constants, notifying PyGame that we're going to be feeding it OpenGL code, as well as DOUBLEBUF, which stands for double buffer, and is a type of buffering where there are two buffers to comply with monitor refresh rates. Take note that pipe ("|") that is used to separate constants. It will be used again later to separate constants.

Next, within this main() function:

```
   gluPerspective(45, (display[0]/display[1]), 0.1, 50.0)
```

gluPerspective is code that determines the perspective, as it sounds. The first value is the degree value of the field of view (fov). The second value is the aspect ratio, which is the display width divided by the display height. The next two values here are the znear and zfar, which are the near and far clipping planes.

MORE ON GLUPERSPECTIVE

void gluPerspective(GLdouble fovy,

```
          GLdouble aspect,

          GLdouble zNear,

          GLdouble zFar);
```

Parameters

fovy

Specifies the field of view angle, in degrees, in the y direction.

aspect

Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).

zNear

Specifies the distance from the viewer to the near clipping plane (always positive).

zFar

Specifies the distance from the viewer to the far clipping plane (always positive).

Description

gluPerspective specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in gluPerspective should match the aspect ratio of the associated viewport. For example, aspect = 2.0 means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

What in the heck is a clipping plane? If you're like me, that means nothing to you at this point. Basically, a clipping plane is at what distance the object appears/disappears. So the object will only be visible between these two values, and both values are supposed to be positive, because they are in relation to your perspective, not in relation to your actual location within the 3D environment.

So, we're having the close clipping happening at 0.1 units and the far clipping plane as 50.0 units away. This will make more sense later, once we've displayed the cube and we can control where we are in the 3D environment, then you will see the clipping planes in action.

Next up, we have:

```
glTranslatef(0.0,0.0, -5)
```

glTranslatef, officially "multiplies the current matrix by a translation matrix." OK cool, again that means nothing to me. So, in layman's terms, this basically moves you, and the parameters are x, y and z. So above, we're moving back 5 unites. This is so we can actually see the cube when we bring it up. Otherwise, we'd be a bit too close.

Now let's write our typical event loop for PyGame. Again, if you want to learn more, check out the aforementioned tutorial.

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
```

This is a simple PyGame event loop that is only checking for any exit, which is only looking for the literal "x" out. Continuing under this "while" statement:

```
    glRotatef(0, 0, 0, 0)
     glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
     Cube()
     pygame.display.flip()
     pygame.time.wait(10)
```

glRotatef multiplies the current matrix by a rotation matrix. The parameters here are angle, x, y and z.

Then we have glClear, which is like any other clearing function. We specify a couple of constants here, which is telling OpenGL what exactly we're clearing.

Once we have a clean "canvas" if you will, we then call our Cube() function.

After that, we call pygame.display.flip(), which updates our display.

Finally we throw in a short wait with pygame.time.wait(10).

TRY:

```
     glRotatef(1, 3, 1, 1)
```