PREPARING THE GRAPHICS

Posted on July 10, 2018

Table of Contents

In this section, we are going to create the visualization tool so as to understand the problems and result more intuitively. Actually, it's just to make things look a little cooler. I mean, who would like to be looking at numbers when you can have an animation! Here are the full source code for this section for your reference:

1. Python BoardDisplay Code
2. Python Wireframe Code

Oh, before I forget, the BoardDisplay code references the Wireframe code so you will need to mark the folder containing the Wireframe.py as the source root. You can do this by right clicking the folder in Pycharm, the select "Mark Directory as", then "sources root". If you do this right, the folder should be marked blue.

Creating the Wireframe

In order to display an object in Pygame, we will first need to fully define its coordinates. To make things simple, I am just going to display a rectangular block to mimic the board which I am using so all we need to do is to define the 6 faces of the block. We are going to do this in 3 steps. First, we will have a class called "Node" to define the xyz coordinates of a point, then we will have another class called "Face" which takes in 4 "Nodes" to define a face of the rectangular block. We will then have 1 last class called "Wireframe" which stored the information of the cube, and also acts as an interface between the Pygame displaying code, and the back-end calculation code.

The back-end calculation part will come in the 3rd section when we start talking about quaternions so for now, we will only concentrate on displaying the block in Pygame. Here's the code for the 3 classes stated above.

Categories

Select Category

Archives

September 2019 (3)
February 2019 (1)
January 2019 (1)
September 2018 (1)
July 2018 (4)
March 2018 (3)
February 2018 (5)
November 2017 (6)
July 2017 (2)
April 2017 (6)
March 2017 (4)
February 2017 (2)
January 2017 (9)
December 2016 (24)
November 2016 (1)

```
# Node stores each point of the block
class Node:
    def __init__(self, coordinates, color):
        self.x = coordinates[0]
        self.y = coordinates[1]
        self.z = coordinates[2]
        self.color = color

# Face stores 4 nodes that make up a face of the block
class Face:
    def __init__(self, nodes, color):
        self.nodeIndexes = nodes
        self.color = color

# Wireframe stores the details of the block
class Wireframe:
    def __init__(self):
        self.nodes = []
        self.edges = []
        self.faces = []

    def addNodes(self, nodeList, colorList):
        for node, color in zip(nodeList, colorList):
            self.nodes.append(Node(node, color))

    def addFaces(self, faceList, colorList):
        for indexes, color in zip(faceList, colorList):
            self.faces.append(Face(indexes, color))

    def outputNodes(self):
        print("\n --- Nodes --- ")
        for i, node in enumerate(self.nodes):
            print(" %d: (%.2f, %.2f, %.2f) \t Color: (%d, %d, %d)" %
                 (i, node.x, node.y, node.z, node.color[0], node.color[1], node.color[2])

    def outputFaces(self):
        print("\n --- Faces --- ")
        for i, face in enumerate(self.faces):
            print("Face %d:" % i)
            print("Color: (%d, %d, %d)" % (face.color[0], face.color[1], face.color[2]))
            for nodeIndex in face.nodeIndexes:
                print("\tNode %d" % nodeIndex)
```

I have also created some convenience functions for logging purposes. This helps to ensure that we inserted the information correctly into the class. We will be using this class in the main "BoardDisplay" code. Here is an example of how you can use the class to initialize parameters for the block.

```
def initializeBlock():
    block = wf.Wireframe()

    block_nodes = [(x, y, z) for x in (-1.5, 1.5) for y in (-1, 1) for z in (-0.1, 0.1)]
    node_colors = [(255, 255, 255)] * len(block_nodes)
    block.addNodes(block_nodes, node_colors)
    block.outputNodes()

    faces = [(0, 2, 6, 4), (0, 1, 3, 2), (1, 3, 7, 5), (4, 5, 7, 6), (2, 3, 7, 6), (0, 1,
    colors = [(255, 0, 255), (255, 0, 0), (0, 255, 0), (0, 0, 255), (0, 255, 255), (255,
    block.addFaces(faces, colors)
    block.outputFaces()

    return block
```

If you run the above code, you should get a print out of all the nodes and faces that you have created. If you used exactly what I have written above, this is the output that you should get:

— Nodes —
0: (-1.50, -1.00, -0.10) Color: (255, 255, 255)
1: (-1.50, -1.00, 0.10) Color: (255, 255, 255)
2: (-1.50, 1.00, -0.10) Color: (255, 255, 255)
3: (-1.50, 1.00, 0.10) Color: (255, 255, 255)
4: (1.50, -1.00, -0.10) Color: (255, 255, 255)
5: (1.50, -1.00, 0.10) Color: (255, 255, 255)
6: (1.50, 1.00, -0.10) Color: (255, 255, 255)
7: (1.50, 1.00, 0.10) Color: (255, 255, 255)

— Faces —
Face 0:
Color: (255, 0, 255)
Node 0
Node 2
Node 6
Node 4
Face 1:
...
...
...

I only copied the print out for Face 0 but you should get print outs of up till Face 5 since we inserted information for 6 faces in total. Let us first begin with the Nodes. We have inserted information for 8 nodes, and as you can see in the print out, they are being indexed from 0 to 7. The color information is redundant for this project because we will be showing the colors of each face instead of the color for the nodes. However, the index of the Nodes are very important because the face object refers to this information. For example, I specified Face 0 to be the face that is enclosed by Node 0, 2, 6 and 4. Below is a diagram which shows the ordering of indexes that I used. The center of the block is located at the origin of the axes.

WIREFRAME NODE INDEX

With this, we have now a fully defined block. The next step is to create the view that you see on top in Pygame.

Creating a Perspective View

Your computer screen shows a 2 dimensional image. There should be no depth because it is just a flat piece of screen. So why does the diagram in the previous section look like a 3D figure to our eyes? This is actually due to the lines that make up the object (and also the axis) which gives an illusion of depth. Here is an article on the topic of depth perception which I though was really interesting so feel free to read through it if you are interested. Creating "depth" on the computer is not that difficult but if you do not draw the object with a perspective view, for example an orthographic view, it will look distorted to our eyes even though it is actually correct.

In this section, we will only talk about a simple implementation of the one vanishing point perspective view. You can look up wikipedia for more details on other perspective views. Below is the diagram of the perspective view that I am going to create for my viewer.

ONE VANISHING POINT PERSPECTIVE VIEW

Firstly, I am going to make the origin the vanishing point, so I will have to shift the object out of the origin. In order to do this, I will shift the center of the object by "P" in the z direction so the new center of the object will now be at (0, 0, P) as shown in the diagram. Next, I am going to put my "screen" at a distance "S" on the z-axis. This screen is the Pygame screen which is 2 dimensional. What we need to do now is to project all the 8 points on the cube onto the screen. If you look closely at the projection, you'll realize that this is simply a linear scaling of the x and y coordinates based on the distance away from the screen.

Let us take Node 0 with coordinates (-1.50, -1.00, -0.10) as a example. When we shift our object out of the origin, the new coordinate of this Node will be (-1.50, -1.00, -0.10+P). In order to project this point onto the screen, we can scale it based on similar triangles. When z=0, we know that the point vanishes so x and y both becomes zero. This serves as a reference point for our calculations. If the screen is located at z=S=5, then our new projected x-coordinate will be:

$$x' = \frac{S}{z}(x) = \frac{5}{-0.10+P}(-1.50)$$

and our new projected y-coordinate will be:

$$y' = \frac{S}{z}(y) = \frac{5}{-0.10+P}(-1.00)$$

The projected depth is a little more complicated so I will not go through it here since this is not the main topic of this series of posts. If you would like more information on it, you can look up here for more information. Actually, on this point, I learnt quite a bit about computer graphics, and it is by no means an easy topic. I should have just went ahead to use the openGL libraries for a better demonstration but I chose to implement everything from scratch. As a result of this, it took me much longer to get things done, and there were some unresolved issues as well. Anyway, I hope that through this post, you will get to know more about computer graphics, and the multitude of problems that follows.

Alright, back to the point, the above equations are implemented in my python BoardDisplay code. Below is an extract of the code:

```
def projectOnePointPerspective(self, x, y, z, win_width, win_height, P, S, scaling_consta
    # In Pygame, the y axis is downward pointing.
    # In order to make y point upwards, a rotation around x axis by 180 degrees is neede
    # This will result in y' = -y and z' = -z
    xPrime = x
    yPrime = -y
    zPrime = -z
    xProjected = xPrime * (S / (zPrime + P)) * scaling_constant + win_width / 2
    yProjected = yPrime * (S / (zPrime + P)) * scaling_constant + win_height / 2
    pvDepth.append(1 / (zPrime + P))
    return (round(xProjected), round(yProjected))
```

We are now ready to draw the points onto our Pygame screen so let us move to the next section

---

Drawing in Pygame

In order to display the object, I used Pygame's draw polygon method. Here's the documentation for it if you want to check it out. Basically, we can use the `pygame.draw.polygon` method to draw a polygon by supplying it with the a list of points. Since our block is simple a six sided rectangle, we have to call this function with a list of 4 points for a total of 6 times, one time for each face, in total. However, it is impossible to show all 6 sides of the block because at any one time, we would be able to see only 3 of the faces. So how do we determine which faces are visible? We can do this using the simple painter's algorithm.

When painting landscapes, we usually start painting from the furthest landscapes, then gradually paint over those landscapes to draw objects that are closer. This is what the painter's algorithm does as well. All we have to do is to order the 6 faces in terms of its depth (at the center of the face), then draw the polygon starting from the furthest face. This is implemented in the code as follows:

```
# Calculate the average Z values of each face.
avg_z = []
for face in self.wireframe.faces:
    n = pvDepth
    z = (n[face.nodeIndexes[0]] + n[face.nodeIndexes[1]] +
         n[face.nodeIndexes[2]] + n[face.nodeIndexes[3]]) / 4.0
    avg_z.append(z)
# Draw the faces using the Painter's algorithm:
for idx, val in sorted(enumerate(avg_z), key=itemgetter(1)):
    face = self.wireframe.faces[idx]
    pointList = [pvNodes[face.nodeIndexes[0]],
                 pvNodes[face.nodeIndexes[1]],
                 pvNodes[face.nodeIndexes[2]],
                 pvNodes[face.nodeIndexes[3]]]
    print(pointList)
    pygame.draw.polygon(self.screen, face.color, pointList)
```

First, we determine the depth of the center of the face (average depth of the nodes of the face), then sort it in increasing order and draw them in order. That's it. Simple isn't it?

However, the painter's algorithm will not work for many situations. For example, you will see later that the painter's algorithm does not work perfectly in the perspective view as it is. I do not yet have a solution for this problem so if you any suggestions, please write it down in the comments section. I'll appreciate any help that I can get.

Anyway, for now, your block is stationary so you cannot see anything except for a green rectangle like this:

SIMPLE PYGAME BLOCK DISPLAY

In the next section, we will be starting on the topic of quaternion rotations so you will be able to make the stationary block above rotate.

Go to Next Section

PREVIOUS POST
Attitude Determination with Quaternion using Extended Kalman Filter

NEXT POST
Quaternion Rotation

POSTED IN ATTITUDE DETERMINATION WITH QUATERNION USING EXTENDED KALMAN FILTER

3 comments on "*Preparing the Graphics*"

HoshiMitama
July 5, 2019 at 8:30 am

really thanks very much for this project.
I come from China, you know it is not easy to break the 'WALL' and get academic information.
but it worths when I read this blog, thanks again.

rikisenia.L
July 8, 2019 at 11:52 pm

Hi HoshiMitama,

Thanks for the positive comment!
I'd glad my post helped you in some way!

Chuong Hoa Loc
May 28, 2021 at 4:32 pm

Great work!

Comments are closed.