Matrix transformations

Oct. 14, 2012

Introduction

In the previous tutorial we changed the Wireframe to use matrices. Now we need to update the code for displaying and transforming wireframes so they work with matrices. In this tutorial we will:

- Fix the display to show the new wireframe object
- Convert the transformating functions into matrix transformations

By the end of the tutorial we should be back where we started two tutorials ago, but our code will be a lot more efficient.

Translation matrix

As described previously, to translate an object means to add a constant to every one of its x-, y- or z-coordinates. A translation matrix is a 2D matrix that looks like this (where dx is the number of units you want to translate the object in the x-coordinate, dy in the y-coordinate and so on):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

Matrix multiplication

The reason for defining a matrix like this is so that when we multiple the node matrix by this matrix, the transformation occurs. If you're not familiar with matrix multiplication, then Wikipedia should help. Briefly, the value (i, j) in the resulting matrix is first value in the ith row times the first value in the jth column plus the second value in the ith row times the

first matrix must equal the number of rows in the second matrix, so all our transformation matrices must have 4 rows.

For example, if we have two nodes and we multiply by the transformation matrix, the first term in the result matrix (which is the x value of the first node) is $(1 \cdot x) + (0 \cdot y) + (0 \cdot z) + (1 \cdot dx)$, which is $x + dx$. This is the reason why we have the row of ones in the node matrix.

If you want to explore the results of multiplying matrices, I've made a basic matrix multiplier that will accept simple variables, such as x and y (but not x1 - you have to use just letters).

Original nodes (2 x 4) · Translation matrix (4 x 4) = Final coordinates (2 x 4)

$$\begin{bmatrix} X_0 & Y_0 & Z_0 & 1 \\ X_1 & Y_1 & Z_1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix} = \begin{bmatrix} X_0+dx & Y_0+dy & Z_0+dz & 1 \\ X_1+dx & Y_1+dy & Z_1+dz & 1 \end{bmatrix}$$

So this might seems an overly complicated way to do things, but in programming terms it is quite straightforward and will make more complex transformations a lot easier later on. Also, NumPy is very efficient at multiplying matrices, so it's fast. Real 3D graphics programs use the GPU is basically designed to do lots of matrix multiplications all at once.

The code

We can can give our **Wireframe** class a function for applying any matrix:

```
28.  def transform(self, matrix):
29.      """ Apply a transformation defined by a given matrix. """
30.
31.      self.nodes = np.dot(self.nodes, matrix)
```

This uses the numpy function **dot()**, which multiplies two matrices. We now write a function in wireframe.py to create a translation matrix:

```
5.
6.      return np.array([[1,0,0,0],
7.                       [0,1,0,0],
8.                       [0,0,1,0],
9.                       [dx,dy,dz,1]])
```

Our object can now be translated in any direction with:

```
1.  matrix = translationMatrix(-10, 12, 0)
2.  cube.transform(matrix)
```

Scaling matrix

A scaling matrix can be defined like this:

```
11.  def scaleMatrix(sx=0, sy=0, sz=0):
12.      """ Return matrix for scaling equally along all axes centred on the point
     (cx,cy,cz). """
13.
14.      return np.array([[sx, 0,  0,  0],
15.                       [0,  sy, 0,  0],
16.                       [0,  0,  sz, 0],
17.                       [0,  0,  0,  1]])
```

If you work through the result of multiplying this matrix with some nodes, you will see that each x value is multiplied by sx, each y value by sy and each z value by sz.

Rotation matrices

The rotation matrices are given below. If you work through the multiplication of these, you'll see that, for example, rotating about the x-axis, does not affect the x-coordinates, but the y- and z-coordinates are changed by a function of both the y- and z-values.

```
20.
21.     c = np.cos(radians)
22.     s = np.sin(radians)
23.     return np.array([[1, 0, 0, 0],
24.                      [0, c,-s, 0],
25.                      [0, s, c, 0],
26.                      [0, 0, 0, 1]])
27.
28. def rotateYMatrix(radians):
29.     """ Return matrix for rotating about the y-axis by 'radians' radians """
30.
31.     c = np.cos(radians)
32.     s = np.sin(radians)
33.     return np.array([[ c, 0, s, 0],
34.                      [ 0, 1, 0, 0],
35.                      [-s, 0, c, 0],
36.                      [ 0, 0, 0, 1]])
37.
38. def rotateZMatrix(radians):
39.     """ Return matrix for rotating about the z-axis by 'radians' radians """
40.
41.     c = np.cos(radians)
42.     s = np.sin(radians)
43.     return np.array([[c,-s, 0, 0],
44.                      [s, c, 0, 0],
45.                      [0, 0, 1, 0],
46.                      [0, 0, 0, 1]])
```

Applying transformations

Now out **Wireframe** object have a **transform()** method and we've defined our transformation matrices we just need to update how **Wireframe** display works.

First we need to make a slight change to the **key_to_function** mapping:

```
10.   pygame.K_RIGHT:(lambda x: x.translateAll([ 10, 0, 0])),
11.   pygame.K_DOWN: (lambda x: x.translateAll([0,  10, 0])),
12.   pygame.K_UP:   (lambda x: x.translateAll([0, -10, 0])),
```

Instead of defining a direction as an axis letter and a magnitude, we define a vector, so moving 10 units along the x-axis is defined as [10, 0, 0]. Then to apply the translation, we need to create the relevant matrix and apply it:

```
72.  def translateAll(self, vector):
73.      """ Translate all wireframes along a given axis by d units. """
74.
75.      matrix = wf.translationMatrix(*vector)
76.      for wireframe in self.wireframes.itervalues():
77.          wireframe.transform(matrix)
```

If you're surprised by the ***vector** command, all it's doing is converting the list of values in the vector (e.g. [10, 0, 0]) into three separate values, so when the translation matrix is made, they fill the dx, dy and dz parameters. Instead we could have written the more verbose:

```
1.  matrix = wf.translationMatrix(vector[0], vector[1], vector[2])
```

Alternatively, we could have created the four different translation matrices to start with and wrote a **translateAll()** function to pass them directly to **wireframe.transform()**, which would have been more efficient, but less flexible.

Comments (8)

Anonymous on May 29, 2013, 6:15 a.m.

Hi Peter,

I can't figure out how to set up numpy. I've downlaoded it but when I try to import it to a program I get an import error, "No module named numpy". Do I need to do something to install numpy or do you think I just don't have it saved in the right place?

You need to install it once you have downloaded it. How you do this depends on your operating system. This might help: http://stackoverflow.com/questions/7562834/installing-numpy

---

Anonymous on May 29, 2013, 5:11 p.m.

I have a mac and all the instructions for instalation appear to be incredibly complicated and require me to download compilers for C and fortran. Am I just overthinking this or is the installation procces really just this involved.

Abhijeet on June 3, 2013, 10:29 a.m.

Hi Peter,

I would like to write a VERY simple program for analysis of framed structures. This would need to display node-edge type structures with zooming, panning, selection of edges by clicking/ rubber banding. I thought Pygame would be right for this until I realised that it's only 2-D. Isn't there any 3-D library to handle the typical stuff like display, selction, etc. ?

Thanks

---

peter on June 3, 2013, 5:58 p.m.

I think there are a few Python 3D graphics libraries, but I've not tried any. I have heard panda3D is quite good: http://www.panda3d.org/

Yang on April 25, 2016, 5:04 p.m.

你好， Peter,

Matrix is useful to operate 3D projection. I have finish matrix transform, scale and rotate via following your tutorial by using matrix. Now I want want some complex model, just like the teapot model in your YouTube vidow. Can you tell me where to find points and edg data for complex model, I think it would be hard to draw it by myself. hahaha...

António on Jan. 9, 2018, 4:48 a.m.

Hi, I know this is an old tutorial, but i can't seem to be able to put it to work.

" File "C:\Python36\opengl\3D\wireframe3.py", line 97, in transform

self.nodes = np.dot(self.nodes, matrix)

ValueError: shapes (8,) and (4,4) not aligned: 8 (dim 0) != 4 (dim 0"

Can anyone help me, or send me the working .py files to this tutorial?

Tks

Matthew B on March 4, 2019, 3:19 a.m.

I wrote this code. I found the code of this corner was missing. here is. https://github.com/matthewb2/captain.git

---

Leave a comment

Name:

Comment:

Post Comment Preview

Using matrices 3D Graphics with Pygame