Pygame 3D Graphics Tutorial

In this tutorial I hope to explain the basics of 3D graphics using Python and Pygame. Pygame is not really designed for 3D graphics, so if you want to create a game with 3D graphics, you'd be better off using something else where all the basics, such as shading, are done automatically. The aim of this tutorial is primarily to explain the mathematics of 3D graphics. I'm using Pygame to display the results simply because it's convienient. Hopefully, however, some of the programs might be useful in their own right.

I haven't finished the tutorial, but the most up-to-date code which was used to create some of the examples below can be found here.

A few of examples of what I've made with Pygame and hope to cover:

Some simple animation

Reading .obj files

Shading

Some simple physics

Nodes and Edges

In this first tutorial, we will create our first 3D wireframe. We will:

- Define Node and Edge objects
- Define a Wireframe object consisting of Nodes and Edges
- Define a Wireframe cube

The full code for this tutorial is attached as a text file at the bottom of this page.

In order to display and manipulate objects on the screen, we have to describe them mathematically. One way to represent a simple shape is with a list of nodes, edges and faces.

- Nodes are points with three coordinates, x, y and z.
- Edges (sometimes called vertices) are lines connecting two nodes.
- Faces are surfaces bounded by multiple edges.

To start with we'll work with wireframe objects, which are only made up of nodes and edges. That way we won't have to worry about shading and figuring out which parts of the object are in front of which others until later.

Objects

So let's start by creating some objects (in a file called wireframe.py) with the appropriate properties:

```
class Node:
    def __init__(self, coordinates):
        self.x = coordinates[0]
        self.y = coordinates[1]
        self.z = coordinates[2]

class Edge:
    def __init__(self, start, stop):
        self.start = start
        self.stop  = stop

class Wireframe:
    def __init__(self):
        self.nodes = []
        self.edges = []
```

We should also give the Wireframe class a couple of methods to make it easy to define the nodes and edges of a Wireframe object:

```
def addNodes(self, nodeList):
    for node in nodeList:
        self.nodes.append(Node(node))

def addEdges(self, edgeList):
    for (start, stop) in edgeList:
        self.edges.append(Edge(self.nodes[start], self.nodes[stop]))
```

Both these methods take lists, which I think is the most versatile approach. The addNodes function takes a list of coordinates. The addEdges function takes a list of pairs of node indices to join. For example, we can add three nodes and create an edge between the second two like so:

```
my_wireframe = Wireframe()
my_wireframe.addNodes([(0,0,0), (1,2,3), (3,2,1)])
my_wireframe.addEdges([(1,2)])
```

Now would be a good time to add a couple of output functions to the Wireframe class to make debugging easier later on:

```
def outputNodes(self):
    print "\n --- Nodes --- "
    for i, node in enumerate(self.nodes):
        print " %d: (%.2f, %.2f, %.2f)" % (i, node.x, node.y, node.z)

def outputEdges(self):
    print "\n --- Edges --- "
    for i, edge in enumerate(self.edges):
```

```
        print " %d: (%.2f, %.2f, %.2f)" % (i, edge.start.x, edge.start.y, edge.start.z),
        print "to (%.2f, %.2f, %.2f)" % (edge.stop.x,  edge.stop.y,  edge.stop.z)
```

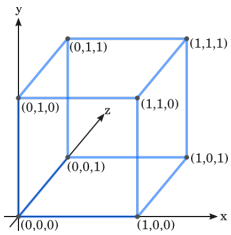
The Cube

I think the simplest shape to start working with is a cube. Although a tetrahedron has fewer sides, its sides aren't orthogonal, which makes things a bit trickier. When I tried to work out the coordinates of a regular tetrahedron, it turned out to be more complicated than I anticipated.

The nodes of a unit cube can be easily defined with a list comprehension:

```
if __name__ == "__main__":
    cube_nodes = [(x,y,z) for x in (0,1) for y in (0,1) for z in (0,1)]
```

If you're not familiar with list comprehensions, then now might be a good time to learn how they work, because I think they're one of the best things in Python and so use them whenever I can. The above list comprehension creates a list of every possible 3-tuple of the digits 0 and 1, thus defining the points of a unit cube.



(In this diagram, I represent the z-axis increasing as you move into the screen, but you could equally have it decreasing as you move into the screen. Similarly, the y-axis increases as you move up the screen, which standard for Cartesian axes, but not how computer graphics are generally displayed. I'll discuss this more in the next tutorial, but for now it's not really important.)

We can now create the nodes of our cube:

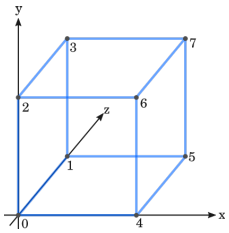
```
cube = Wireframe()
cube.addNodes(cube_nodes)
```

The edges are a little trickier to define. I find it easiest to connect them in groups that are parallel. For example, if you look at the diagram below, you can see that the edges parallel to the x-axis connect the node pairs (0,4), (1,5), (2,6) and (3,7), so we can define them with a list comprehension:

```
cube.addEdges([(n,n+4) for n in range(0,4)])
```

The remaining edges can be added like so:

```
cube.addEdges([(n,n+1) for n in range(0,8,2)])
cube.addEdges([(n,n+2) for n in (0,1,4,5)])
```



Verify the cube has the properties you expect with:

```
cube.outputNodes()
cube.outputEdges()
```

Now we have a cube object, but it only exists as an abstract object. In the next tutorial, we'll display it.

Attachment	Size
wireframe1.txt	1.41 KB

Projecting 3D objects

In the previous tutorial we created a three-dimensional cube object, now we want to display it on a two-dimensional screen. In this tutorial, we will:

- Create a simple Pygame window
- Project an image of our 3D object onto the 2D window

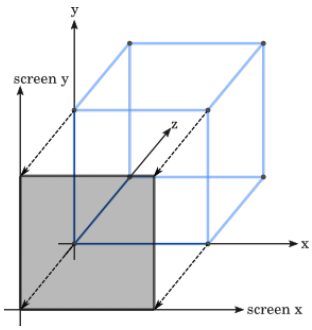
As before, you can find the final code is at bottom of the page as a text file.

3D Projections

In order to display our cube we need to convert 3D coordinates, (x, y, z), into 2D coordinates (screen_x, screen_y). This mapping from a 3D coordinate system to a 2D coordiante system is called a projection. You can imagine that we're shining a light from behind our 3D object and looking at the shadow it casts on a 2D screen. In fact, since our retinas are essentially 2D, all we ever see are projections of objects (albeit stereoscopic projections).

So to trick our brain into thinking that the 2D shape on the screen is actually 3D, we need to work out what 2D shapes would form on our retina when the 3D object is projected on to it.

There are many different way to project a 3D object onto a screen (see types of projection on Wikipedia), corresponding to viewing the object from different angles and perspectives. The simplest projection is to imagine that we're looking at our cube head on (so our "line-of-sight" is parallel to, or along, the z-axis). In this case, the z-axis contributes no information to what we see and we can simply ignore it. Since we're using a wireframe model, we don't need to pay attention to the order of elements along the z-axis.



In terms of vector transformations, we are using the linear transformation: T(x,y,z) → (x,y).

Basic Pygame program

In order to keep our code tidy, we'll put the code dealing with displaying wireframes in a separate file. This will allow us to use alternative code to display wireframes if we prefer. So in a new file called wireframeDisplay.py or something similar, import pygame and our wireframe module:

```
import wireframe1 as wireframe
import pygame
```

The projection viewer

Now let's create an class to deal with displaying projections of wireframe objects. It will contain all the variables concerned with how objects are displayed, such as the screen dimensions, the colours used and whether to display the nodes and/or edges. It also contains an empty dictionary which will contain the wireframes.

```
class ProjectionViewer:
    """ Displays 3D objects on a Pygame screen """

    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.screen = pygame.display.set_mode((width, height))
        pygame.display.set_caption('Wireframe Display')
        self.background = (10,10,50)

        self.wireframes = {}
        self.displayNodes = True
        self.displayEdges = True
        self.nodeColour = (255,255,255)
        self.edgeColour = (200,200,200)
        self.nodeRadius = 4
```

Hopefully you are familiar with the basics of Pygame. If not, you can look through the first couple of tutorials in my Pygame physics tutorial. We now add a run() function to the ProjectionViewer which will display a pygame window.

```
def run(self):
    """ Create a pygame screen until it is closed. """

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        self.screen.fill(self.background)
        pygame.display.flip()
```

We can now create a ProjectionViewer object and run it. This should create a 400 x 300 pixel window with a deep blue background ready for our wireframe.

```
if __name__ == '__main__':
    pv = ProjectionViewer(400, 300)
    pv.run()
```

In order to display wireframe, we need to be able to add them to the `ProjectionViewer`, so let's add a method for just that:

```
def addWireframe(self, name, wireframe):
    """ Add a named wireframe object. """
    self.wireframes[name] = wireframe
```

By using a dictionary, we can add multiple wireframes and then manipulate them separately (rotating one for example). We can now create wireframe cube as before (only a bit more compactly) and add it to a `ProjectionViewer` object.

```
cube = wireframe.Wireframe()
cube.addNodes([(x,y,z) for x in (0,1) for y in (0,1) for z in (0,1)])
cube.addEdges([(n,n+4) for n in range(0,4)]+[(n,n+1) for n in range(0,8,2)]+[(n,n+2) for n in (0,1,4,5)])

pv = ProjectionViewer(400, 300)
pv.addWireframe('cube', cube)
pv.run()
```

Displaying wireframes

The code still doesn't actually display the wirefames, so let's now add a display method to `ProjectionViewer`:

```
    def display(self):
        """ Draw the wireframes on the screen. """

        self.screen.fill(self.background)

        for wireframe in self.wireframes.values():
            if self.displayEdges:
                for edge in wireframe.edges:
                    pygame.draw.aaline(self.screen, self.edgeColour, (edge.start.x, edge.start.y), (edge.stop.x, edge.stop.y), 1)

            if self.displayNodes:
                for node in wireframe.nodes:
                    pygame.draw.circle(self.screen, self.nodeColour, (int(node.x), int(node.y)), self.nodeRadius, 0)
```

The `display()` method fills the background then draws all the wireframe nodes as circles at the (x, y) coordinates of the node, ignoring its z coordinate and draws edges as anti-aliased lines between the relavant nodes' (x, y) coordinates. We call the display() method in the run() method's loop where we were previously just drawing the background:

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

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

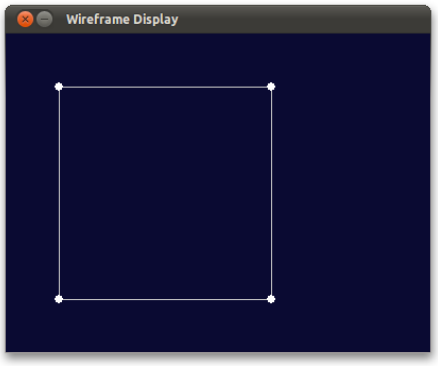
Fixing the coordinates

If you run the program now you will see a bit of a circle in the top left corner because we're currently drawing the circles for the nodes at (0,0), (1,0), (0,1) and (1,1). We can create a more sensible cube by changing its nodes to:

```
cube.addNodes([(x,y,z) for x in (50,250) for y in (50,250) for z in (50,250)])
```

Note that we don't actually have to change the z coordinates for moment, but we might as well. In the next tutorial we'll add methods for zooming and panning the display so we can view our unit cube (with 0 and 1 coordinates). Another issue is that we are viewing our cube 'upside-down' since the y-axis actually starts at the top of the screen and goes down. We'll deal with this problem in later tutorial.

Now if you run the program you should see something like this:



This is what our cube looks like when we view it directly end on. It might seem like we've cheated. If you're familiar with Pygame then I'm sure you could have drawn a square and a few circles with a lot less effort. However, in the next two tutorials we'll deal with various transformations of the cube including rotations, which will hopefully convince you that we're actually looking at a 3D object.

Attachment	Size
displayWireframe1.txt	1.93 KB

Basic 3D transformations

In the previous tutorial we displayed a static cube wireframe, which appeared as a square. In order to get a sense of its three dimensions, we must be able to move it in three dimension. But first, we'll introduce some basic transformations, which don't require a third dimension. In this tutorial, we will:

- Add the ability to translate wireframes
- Add the ability to scale wireframes
- Apply these transformations using the keyboard

By the end of the tutorial, we'll be able to move and scale the square as shown in the video below. The final code is at the bottom of the page as text files.

Translation in 3D

The simplest transformation is a translation: moving the wireframe along an axis by adding a constant to the x, y or z coordinate of every node. For this, we add the following method to the `Wireframe` class:

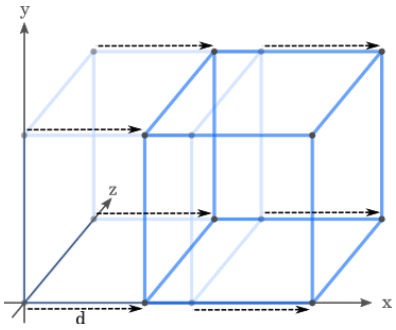
```
def translate(self, axis, d):
    """ Add constant 'd' to the coordinate 'axis' of each node of a wireframe """

    if axis in ['x', 'y', 'z']:
        for node in self.nodes:
            setattr(node, axis, getattr(node, axis) + d)
```

The `translate()` method takes the name of an axis and distance that the wireframe should be translated. It adds the distance, d, to the relevant coordinate of every node in the wireframe. We use `getattr()` and `setattr()` so we can easily define which attribute we want to change.

For example, to move our cube 100 pixels to the right, we get and set the attribute 'x':

```
cube.translate('x', 100)
```



Because the y-axis of the screen starts at the top and points down, to move our cube up 40 pixels, we call:

```
cube.translate('y', -40)
```

Translating along the z-axis will have no noticeable effect at the moment.

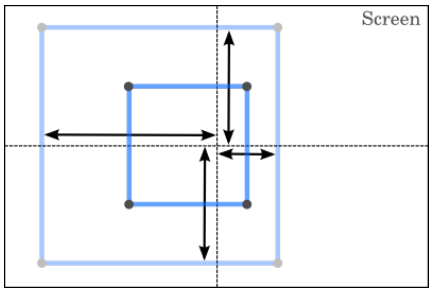
Scaling in 3D

Scaling is also relatively straightforward. We could simply multiply the x, y and z values of each node by a scaling factor, which would have the effect of scaling the cube centred on the origin. However, for more flexibility we can scale from any centre. using the following method:

```
def scale(self, (centre_x, centre_y), scale):
    """ Scale the wireframe from the centre of the screen """

    for node in self.nodes:
        node.x = centre_x + scale * (node.x - centre_x)
        node.y = centre_y + scale * (node.y - centre_y)
        node.z *= scale
```

For example, if we know the centre of the screen then the function scales the distance of each node from the centre (ignoring the z coordinate). If we assume that the screen is at z=0, then nodes behind the screen move closer as we scale down, and further away as we scale up.



For example, to scale our cube by three quarters, centred on the screen's centre:

```
cube.scale((200, 150), 0.75)
```

Keyboard controls

In order to easily use these transformations we can associate them with keys and call them (with some arbitrary value) in response to key presses. We've arranged our code so that the display of the wireframes using Pygame is in a file called displayWireframe2.py. This code should also handle keyboard inputs using Pygame. First we'll add some methods to the ProjectionViewer class to transform all its wireframes (and to calculate the centre of the screen):

```
def translateAll(self, axis, d):
    """ Translate all wireframes along a given axis by d units. """

    for wireframe in self.wireframes.itervalues():
        wireframe.translate(axis, d)

def scaleAll(self, scale):
    """ Scale all wireframes by a given scale, centred on the centre of the screen. """

    centre_x = self.width/2
    centre_y = self.height/2

    for wireframe in self.wireframes.itervalues():
        wireframe.scale((centre_x, centre_y), scale)
```

We can then associate these methods with keys with a dictionary that maps keys to lambda function. I've written in more detail on how this code works here. You can use whichever set of keys you find most logical.

```
key_to_function = {
    pygame.K_LEFT:   (lambda x: x.translateAll('x', -10)),
    pygame.K_RIGHT:  (lambda x: x.translateAll('x',  10)),
    pygame.K_DOWN:   (lambda x: x.translateAll('y',  10)),
    pygame.K_UP:     (lambda x: x.translateAll('y', -10)),
    pygame.K_EQUALS: (lambda x: x.scaleAll(1.25)),
    pygame.K_MINUS:  (lambda x: x.scaleAll( 0.8))}
```

Finally we need to check whether a key is pressed, and if it is, and it's one that's in key_to_function, we call the relevant function. We do this by adding the following code into the loop in the ProjectionViewer's run() method:

```
elif event.type == pygame.KEYDOWN:
    if event.key in key_to_function:
        key_to_function[event.key](self)
```

This is a slightly indirect way to do things - the function is called, sending the ProjectionViewer object (refered to by self), to the lambda function, which then calls the translate or scale function of ProjectionViewer. The advantage of this method is that it makes it easier to add or change key commands.

We can now manipulate our cube wireframe to a degree, but it still looks like a square. In the next tutorial, we'll introduce another transformation - rotation - which will finally allow us to see another side to our square and see that it really is a cube.

Attachment	Size
wireframe2.txt	1.96 KB
displayWireframe2.txt	2.95 KB

Rotation in 3D

In the previous tutorial, we added the ability to apply some basic transformations to our wireframe cube, but it still looked like a square. In this tutorial, we will:

- Add the ability to rotate wireframes about three axes

Rotations are more complex that the previous two transformations but more interesting. Translations add a constant to coordinates, while scaling multiples a constant by the coordinate, so both preserve the shape of the projection (i.e. square). Rotations on the other hand change the values of two coordiates e.g. x and y, by a function of both those coordinates. This means that the coordinates effectively interact, so the z-coordinate will come into play by influencing the value of the x and y coordinates. Thus we will finally see a different side of our 3D object. Hopefully this will become clear with an example.

Defining an axis

Rotations are defined by an angle and a vector through which the rotation occurs. The easiest example is to rotate the cube through an axis parallel to the z-axis. For example, as we look at our cube end on, we rotate the square we see about its centre. As we are rotating about the z-axis, only the x and y coordinates will change, so we won't see our z-coordinates just yet. So it is essentially a 2D problem.

First let's create a method for Wireframe to find its centre. The centre is just the mean of the x, y and z coordinates. In the Wireframe class add:

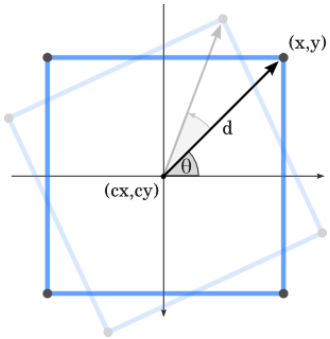
```
def findCentre(self):
    """ Find the centre of the wireframe. """

    num_nodes = len(self.nodes)
    meanX = sum([node.x for node in self.nodes]) / num_nodes
    meanY = sum([node.y for node in self.nodes]) / num_nodes
    meanZ = sum([node.z for node in self.nodes]) / num_nodes

    return (meanX, meanY, meanZ)
```

Converting from Cartesian to a polar coordinates

Since we are going to rotate points about an angle, it's easier to switch to using polar coordinates. This means rather than refer to a point as being x units along the screen and y units up the screen, we refer to it as being an angle and distance from the point of rotation. For example, below, we convert (x, y) to (θ, d).

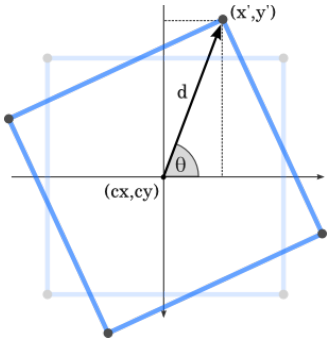


We can find the angle, θ (in radians) using a handy function from the math module called `atan2()`, which also deals with orthogonal situations. The angle measured, is between the vector and the x-axis. This isn't particularly important, so long as we're consistent when we convert back to a coordinate later. The distance, d, in the diagram is calculated as the hypotenuse of the triangle formed by the vector:

```
import math
d     = math.hypot(y-cy, x-cx)
theta = math.atan2(y-cy, x-cx)
```

Converting from polar to a Cartesian coordinates

Now we have an angle, θ, we add the angle of our rotation, then convert back to Cartesian coordinates.



The new x-coordinate is the width of the triangle (the distance between cy and y), which is, by simple trigonometry, d.cos(θ). The new y-coordinate is the height of the triangle, which is d.sin(θ). The complete Wireframe method should look like this (remember to `import math` at the start of the program):

```
def rotateZ(self, (cx,cy,cz), radians):
    for node in self.nodes:
        x      = node.x - cx
        y      = node.y - cy
        d      = math.hypot(y, x)
        theta  = math.atan2(y, x) + radians
        node.x = cx + d * math.cos(theta)
        node.y = cy + d * math.sin(theta)
```

We can now rotate the projected square about its centre with:

```
cube.rotateZ(cube.findCentre(), 0.1)
```

We still haven't yet seen into the third dimension, but by analogy to a rotation around a vector parallel to the z-axis, we can also rotate our object about vectors parallel to the x- and y-axes with the following methods:

```
def rotateX(self, (cx,cy,cz), radians):
    for node in self.nodes:
        y      = node.y - cy
        z      = node.z - cz
        d      = math.hypot(y, z)
        theta  = math.atan2(y, z) + radians
        node.z = cz + d * math.cos(theta)
        node.y = cy + d * math.sin(theta)

def rotateY(self, (cx,cy,cz), radians):
    for node in self.nodes:
        x      = node.x - cx
        z      = node.z - cz
        d      = math.hypot(x, z)
        theta  = math.atan2(x, z) + radians
        node.z = cz + d * math.cos(theta)
        node.x = cx + d * math.sin(theta)
```

Now - finally - we can see all aspects of our cube. Having three separate rotation methods is not the most efficient way to do things - in a later tutorial I demonstrate how matrices can deal with tranformations more efficiently.

Key Controls

Next we need to add a rotate function to the ProjectionViewer class. The method below rotates all wireframes about their centres and a given axis. Depending on what you want, it may be more useful to rotate all objects about the origin or about the centre of the screen.

```
def rotateAll(self, axis, theta):
    """ Rotate all wireframe about their centre, along a given axis by a given angle. """

    rotateFunction = 'rotate' + axis

    for wireframe in self.wireframes.itervalues():
        centre = wireframe.findCentre()
        getattr(wireframe, rotateFunction)(centre, theta)
```

Finally we can update our `key_to_function` dictionary to bind these new function calls to keys. For example:

```
key_to_function = {
...
    pygame.K_q: (lambda x: x.rotateAll('X',  0.1)),
    pygame.K_w: (lambda x: x.rotateAll('X', -0.1)),
    pygame.K_a: (lambda x: x.rotateAll('Y',  0.1)),
    pygame.K_s: (lambda x: x.rotateAll('Y', -0.1)),
    pygame.K_z: (lambda x: x.rotateAll('Z',  0.1)),
    pygame.K_x: (lambda x: x.rotateAll('Z', -0.1))}
```

So at last we can rotate our cube and see it from every angle. In the next tutorial we'll switch to using matrices so we can carry out transformations faster so we can look at object more complex than a cube.

Attachment	Size
displayWireframe3.txt	3.62 KB
wireframe3.txt	3.26 KB

Matrix transformations

[Not finished but I published anyway.] 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 discussed 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. Breifly, the value (i, j) in the resulting matrix is first value in the i^{th} row times the first value in the j^{th} column plus the second value in the i^{th} row times the second value in the j^{th} column and so on. This is the dot product of the vectors given by the i^{th} row and the j^{th} column. Note that this requires that the number of columns in 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.x + 0.y + 0.z + dx.1, or 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 multiplier matrices, I have an online matrix multipler that will accept simple variables, such as x and y (but not x1 - you have to use just letters): http://petercollingridge.appspot.com/matrix_multiplier

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 quite quick. Real 3D graphics programs use the GPU is basically designed to do lots of matrix mutliplications all at once.

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

```
def transform(self, matrix):
    """ Apply a transformation defined by a given matrix. """

    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:

```
def translationMatrix(dx=0, dy=0, dz=0):
    """ Return matrix for translation along vector (dx, dy, dz). """

    return np.array([[1,0,0,0],
                     [0,1,0,0],
                     [0,0,1,0],
                     [dx,dy,dz,1]])
```

Our object can now be translated in any direction with:

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

Scaling matrix

A scaling matrix can be defined like this:

```
def scaleMatrix(sx=0, sy=0, sz=0):
    """ Return matrix for scaling equally along all axes centred on the point (cx,cy,cz). """

    return np.array([[sx, 0,  0,  0],
                     [0,  sy, 0,  0],
                     [0,  0,  sz, 0],
                     [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.

```
def rotateXMatrix(radians):
    """ Return matrix for rotating about the x-axis by 'radians' radians """

    c = np.cos(radians)
    s = np.sin(radians)
    return np.array([[1, 0, 0, 0],
                     [0, c,-s, 0],
                     [0, s, c, 0],
                     [0, 0, 0, 1]])

def rotateYMatrix(radians):
    """ Return matrix for rotating about the y-axis by 'radians' radians """

    c = np.cos(radians)
    s = np.sin(radians)
    return np.array([[ c, 0, s, 0],
                     [ 0, 1, 0, 0],
                     [-s, 0, c, 0],
                     [ 0, 0, 0, 1]])

def rotateZMatrix(radians):
    """ Return matrix for rotating about the z-axis by 'radians' radians """

    c = np.cos(radians)
    s = np.sin(radians)
    return np.array([[c,-s, 0, 0],
                     [s, c, 0, 0],
                     [0, 0, 1, 0],
                     [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 Wireframedisplay works.

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

```
key_to_function = {
 pygame.K_LEFT: (lambda x: x.translateAll([-10, 0, 0])),
 pygame.K_RIGHT:(lambda x: x.translateAll([ 10, 0, 0])),
 pygame.K_DOWN: (lambda x: x.translateAll([0,  10, 0])),
 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:

```
def translateAll(self, vector):
    """ Translate all wireframes along a given axis by d units. """

    matrix = wf.translationMatrix(*vector)
    for wireframe in self.wireframes.itervalues():
        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:

```
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.

Using matrices

Our wireframe object is currently defined by a list of Node objects and a list of Edge objects, which hopefully makes things easy to understand, but it isn't very efficient. That's not a problem if we're just making cubes, but will be if we want to create more complex objects. In this tutorial, we will:

- Convert our list of nodes to a numpy array
- Simplify how edges are stored
- Create a cube using the new system

By the end of this and the following tutorial our program should function exactly the same as before, but will be more efficient.

NumPy

If you're not familiar with NumPy, then this tutorial might take a bit of work to understand, but I think it's worth the effort. You can do everything

without using matrices, but it does actually simplify things in the long run and your program will be a lot quicker. I'll do my best to explain NumPy, but you might also want to look at the official tutorial.

The first thing is to download NumPy if you haven't already done so. Then import it in our wireframe.py module (the as np is a common shortcut which saves a bit of typing later):

```
import numpy as np
```

Since NumPy includes a lot of mathematical functions, we can use it to replace the math module, thus replace math.sin() with np.sin().

NumPy arrays (matrices)

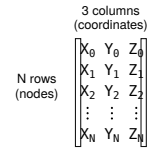
Our program currently defines nodes using the Node object, which is fine when you only have eight, but if you want thousands then it will quickly become very time and memory inefficient. A node is really just three numbers, so we could convert the list of nodes to a list of lists, each containing three numbers. However, if we use a NumPy array, we get a lot of built-in mathematical functions which will prove useful later.

Thus, we can delete our Node object and change the Wireframe class nodes attribute to:

```
self.nodes = np.zeros((0, 4))
```

This creates a NumPy array with 0 row and 4 columns. This would be filled with zeros, but since there are no rows, there are no values. There are no rows because, to start with there are no nodes. There are four rather than three columns because it makes some transformations easier as I'll explain when we come to them. Note that we are using the NumPy array class and not the matrix class because it's easier to work with and does everything that matrices do. From a mathematical point of view they can still be considered matrices.

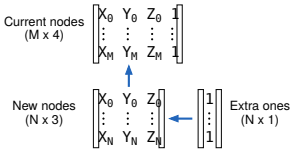
Next we need to change the Wireframe class addNodes() function because it currently takes a list of 3-tuples and converts each into a Node object. We want to change it to take a N x 3 NumPy array, in which each of the N rows is a vector of 3 coordinates (x, y and z).



For example, we would define the nodes of a unit square like this:

```
square = Wireframe()
nodes = np.array([[0, 0, 0],
                  [1, 0, 0],
                  [1, 1, 0],
                  [0, 1, 0]])
square.addNodes(nodes)
```

In order to add this N x 3 array of nodes to the array of current nodes, we first need to add a N x 1 column of ones to get a N x 4 array. Then we add the new nodes as additional rows to the current node array.



We create a N x 1 array of ones by using np.ones(N, 1). We could work out the number of rows we need (N) by looking at the shape attribute of the new node array. For example, you can try:

```
print nodes.shape
>>> (4, 3)
```

Alternatively we can use the len() function, which returns the number of rows of an array, as though it were a list. Once we have a column of ones we horizontally stack onto the array of nodes, using np.hstack(). We then vertically stack that array onto the array of current nodes with np.vstack(). So we change our Wireframe addNodes() method to:

```
def addNodes(self, node_array):
    ones_column = np.ones((len(node_array), 1))
    ones_added = np.hstack((node_array, ones_column))
    self.nodes = np.vstack((self.nodes, ones_added))
```

Simplifying edges

Just as the Node object as basically three numbers, the Edge object is basically two numbers. We could also replace all the edges with a simple NumPy array, but in this case, I think it's easier to use a list of lists. Once we've defined the edges we never need to change the values, so we don't need the matrix functions available for working with arrays.

We can therefore remove the Edge object simplify the addEdges() method to:

```
def addEdges(self, edgeList):
    self.edges += edgeList
```

Testing the new system

To check that our addNodes() and addEdges() methods are working as we expect, we should update the outputNodes() and outputEdges() methods.

```
def outputNodes(self):
    print "\n --- Nodes --- "
    for i, (x, y, z, _) in enumerate(self.nodes):
        print "   %d: (%d, %d, %d)" % (i, x, y, z)
```

Here we loop through the nodes, getting their x, y and z coordinates. We can ignore the final value as this will always be 1. We should update outputEdges() too.

```
def outputEdges(self):
    print "\n --- Edges --- "
    for i, (node1, node2) in enumerate(self.edges):
        print "   %d: %d -> %d" % (i, node1, node2)
```

We can now create a cube object in a similar way as before:

```
cube = Wireframe()
cube_nodes = [(x,y,z) for x in (0,1) for y in (0,1) for z in (0,1)]
cube.addNodes(np.array(cube_nodes))

cube.addEdges([(n,n+4) for n in range(0,4)])
cube.addEdges([(n,n+1) for n in range(0,8,2)])
cube.addEdges([(n,n+2) for n in (0,1,4,5)])

cube.outputNodes()
cube.outputEdges()
```

Fixing the display

Finally we should change the ProjectionViewer class's display() function to work with the new Wireframe nodes and edges. We update how nodes are displayed by changing node.x to node[0] and node.y to node[1]:

```
if self.displayNodes:
    for node in wireframe.nodes:
        pygame.draw.circle(self.screen, self.nodeColour, (int(node[0]), int(node[1])), self.nodeRadius, 0)
```

To fix how the edges are displayed we change the code to:

```
if self.displayEdges:
    for n1, n2 in wireframe.edges:
        pygame.draw.aaline(self.screen, self.edgeColour, wireframe.nodes[n1][:2], wireframe.nodes[n2][:2], 1)
```

This loops through the edges, which are a list of lists containing two numbers, which we call n1, and n2. These refer to the start and end nodes of the edges, so we get those nodes, and then extract their x- and y-coordinates, which are the first two, hence the [:2] index.

To test the code, we have to import numpy as np like before:

```
import numpy as np
```

Then change the addNodes() call to:

```
cube.addNodes(np.array(cube_nodes))
```

The one difference between creating a cube with this new Wireframe object, is that we must first convert the list comprehension into a NumPy array. Now we should find that we have succesfully created a cube object. In the next tutorial we'll fix the transformation functions.

Attachment	Size
displayWireframe4.txt	3.65 KB
wireframe4.txt	1.56 KB