3D Graphics with pi3d

Release 1.0

Paddy Gaunt

June 15, 2015

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Target Audience

It's not possible to cater for all levels of knowledge and experience in one book. Inevitably you will find sections where I cover things that you already know and you can skip ahead quickly, but there will also be parts that seem difficult where you will have to take it slowly and check things out on-line (I try to include links where they might be useful but you can always resort to google!). I assume that:

- You already know how to program (either with python to a basic level, or with another language to a high enough level that the switch to python presents few issues). Apart from things that are non-standard, such as using numpy [1], ctypes [2] or GLSL [3], I shall not explain general programming concepts or syntax. Also, although I will try to introduce pi3d concepts in a logical order, this will not necessarily correspond with the sophistication of the programming techniques used.
- Although not essential it will help if you have a reasonable (secondary school) understanding of mathematics such as basic trigonometry (*sin, cos, tan, arctan* [4], Pythagoras [5] etc.) and have at least come across *vectors and matrices* [6].
- You have an average understanding of computer images and display, in so far as they consist of pixels with red, green, blue values and that some file types (PNG, TIFF etc.) allow transparency per pixel with a fourth, alpha value.
- You want to learn a bit about the mechanisms of producing 3D graphics rather than simply find recipes for how to do things.

1.2 The Structure and how to use this book

A book on how to program, or use a module within a language, inevitably needs lots of code to look at and run (unless it sets out to be a very boring book). And code needs lots of comments and explanations because that's almost the essence of good coding. So it did occur to me that the whole book could be constructed entirely from the documentation in the example programs that accompanied it. However, although this works for generating documentation (as used for http://pi3d.github.io/html/index.html), it produces too many constraints and a rather unwieldy book structure. In the end I opted for this:

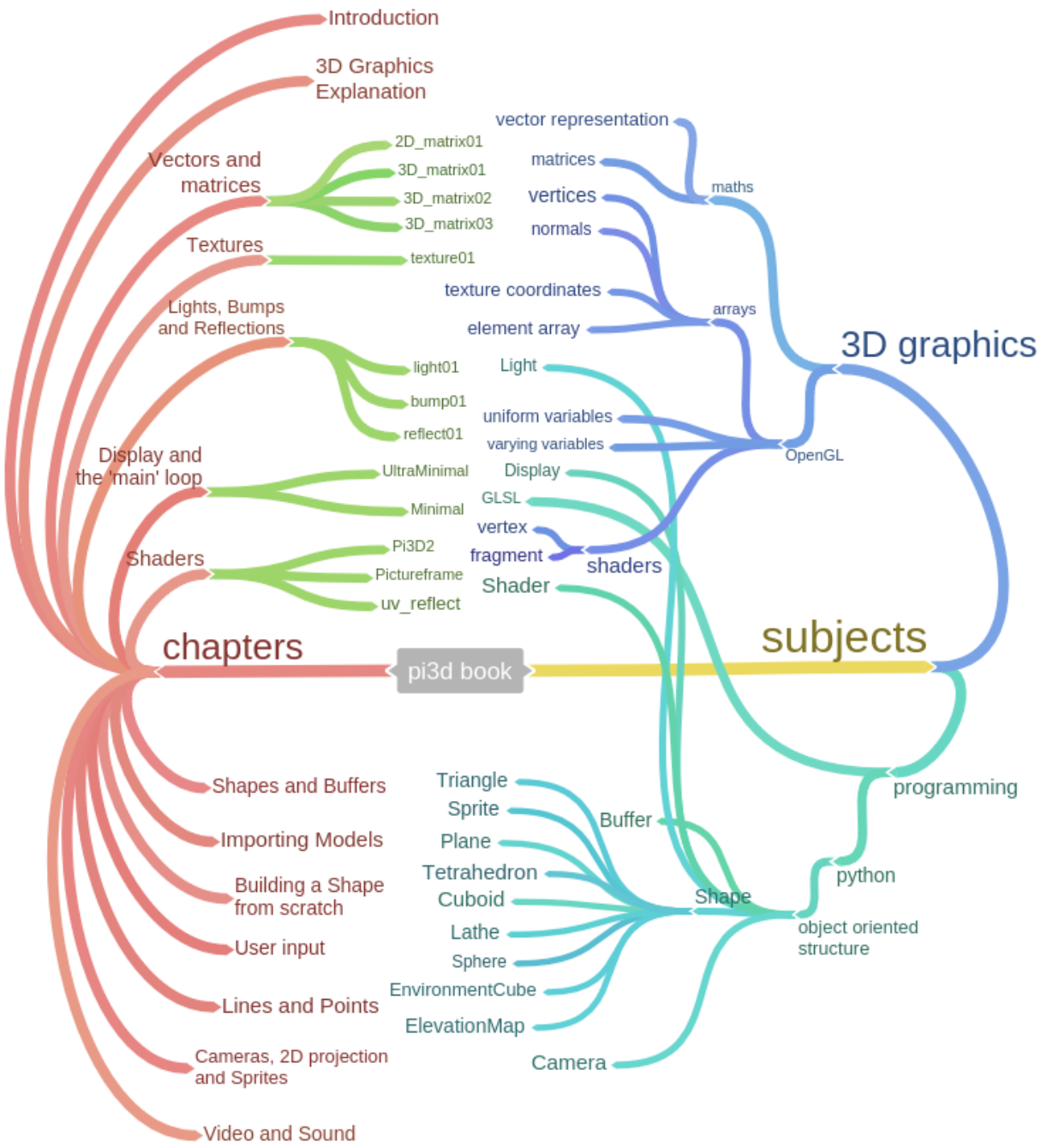
1. A narrative and overall explanation starting from the workings of the GPU and OpenGL, the use of vectors and matrix transformations, simple shapes and shaders through to complicated projects and games.

[1] http://www.numpy.org/
[2] https://docs.python.org/2/library/ctypes.html
[3] http://en.wikipedia.org/wiki/OpenGL_Shading_Language
[4] https://www.mathsisfun.com/sine-cosine-tangent.html
[5] https://www.mathsisfun.com/pythagoras.html
[6] http://www.intmath.com/vectors/vectors-intro.php

2. A set of programs that can be run to illustrate the topics covered above. Generally the illustrations will use pi3d to create the graphical output but to start with this will be glossed over as a distraction from the topic being explained.
3. A selection of the demo programs from github.com/pi3d/pi3d_demos. As familiarity and understanding grow these can become a source of ideas and boiler-plate code to modify.
4. The source code of pi3d. A key reason to use python is that it's easy to read and understand how it works. If something in pi3d doesn't work as you expect you are encouraged to open the source code with an editor and figure out exactly how it works.
5. The on-line documentation. Things like installation instructions for different platforms, arguments and return values of class methods, and FAQs belong elsewhere and do not clutter up this book!

The programs are referenced from the narrative and include comprehensive docstrings and comments. **The code and the docstrings are NOT duplicated in the narrative** so it is essential that the book is read in conjunction with the programs, and that the programs are *read* as well as run.

This is an outline mind map from Coggle.it



1.3 Installation

If you don't have pi3d set up already you need to read the relevant section here http://pi3d.github.io/html/ReadMe.html#setup-on-the-raspberry-pi and the paragraphs below, that apply to your platform.

At points in the book I will suggest that you look in various pi3d files and if you installed on Raspberry Pi or Linux these will be in `/usr/local/lib/python2.7/dist-packages/pi3d/` (or similar), on Windows try `C:\Python27\Lib\site-packages\pi3d\`. Obviously python3 would have an appropriately different path.

You also need to get a copy of http://github.com/pi3d/pi3d_demos either using git clone or download the zip and extract it.

And you need a copy of the example programs for this book github.com/paddywwoof/pi3d_book

CHAPTER

TWO

3D GRAPHICS EXPLANATION

This is a short introduction to 3D graphics from the perspective of pi3d, there will be gaps and possibly misapprehensions but it should give a reasonable general perspective of how things work! Also I intentionally skip over many of the more involved aspects such as rendering to off-screen buffers, using masks, etc.

Beneath the python module classes and functions provided by pi3d there are three "steps" necessary for control of the GPU. Two or three of these require external libraries (shared objects in linux, dlls in windows) imported [1]:

1. on the Raspberry Pi `libbcm_host.so` is used to create and manage a display surface to draw on. On linux (desktop and laptop) the surface is provided by the x11 server [2] and windows and Android use pygame (which uses SDL2 [3])

2. `libELG` is used to set up the interface between the machine or operating system window system and

3. `libGLESv2` provides access to the OpenGL language functions developed to standardise utilisation of graphics cards. Mobile devices, including the Raspberry pi use a slightly cut-down version called OpenGL ES, specifically version 2.0.

From OpenGL ESv2.0 onwards the fundamental graphics donkey work is done by 'shaders' that are defined by the developer and compiled as the program runs rather than being 'built into' the GPU. This opens up a fantastic range of possibilities but there are some fundamental limits that may not be immediately apparent.

2.1 Communication between python and the GPU

There are two parts to a shader: the vertex shader and the fragment (essentially pixel) shader which are written in a C like language (GLSL). I will give some more detail to what each actually does later but one crucial thing to appreciate is that information is passed from the CPU program (in our case python pi3d ones) to the shaders and the vertex shader can pass information on to the fragment shader, however the only output is pixels [4]. It is fundamental to the efficiency and speed of the GPU that the shaders operate on only one vertex or pixel. i.e. the vertex shader can't access information about the location of adjacent vertices and the fragment shader can't determine the colour, say, of adjacent pixels. This allows the processing to be run in parallel (massively parallel, some GPU have thousands of processing cores) but means that some operation such as blurring or edge detection have to be done as a double pass process.

Information needed to render the scene is passed to the shader in four distinct blocks:

1. *An 'element array'* that will be drawn by the call to the drawElements function. This function can be used to draw polygons (limited to triangles in OpenGL ES2.0), lines or points, and the type of drawing will determine

[1] The attempt to work out on what platform pi3d is running and what libraries to import is done in /pi3d/constants/__init__.py and the Initialization is done in /pi3d/utils/DisplayOpenGL.py

[2] X11 is the standard windowing and user-input system used on Linux systems

[3] Simple DirectMedia Layer https://www.libsdl.org/index.php

[4] It is possible to get 'output' from GPUs using sophisticated techniques that allow the parallel processing capabilities to be used elsewhere, but this is not trivial!

how the entries in the array are interpreted. Essentially each element will contain reference indices to one or more vertices. In the simple square example below this is the `triangle indices` array.

2. *An 'attribute array'* of vertex information, again the type of drawing determining how much information needs to be passed. For the most general 3D drawing in pi3d the array contains vertex x,y,z values, normal vectors and texture coordinates.
3. *'uniform' variables*. This includes things that apply to all the vertices being drawn, such as the transformation matrix (for the shape to which the vertices belong), the projection matrix to represent camera location and field of view, the location and colour of light sources, fog properties, material shades and transparency, variables to control pattern repeats or for moving patterns etc.

   A very significant part of the uniform variables are images or texture samplers to 'clothe' the object or to provide information on bumps or reflections.
4. *The program* for the GPU to run, comprising the vertex and fragment shader.

In pi3d these four categories of information are held in various objects: The element and attribute arrays are part of the Buffer and the Shader class contains the shader programs. However the uniform variables are held in Buffer, Shape, Camera, Light and Texture objects as seemed logical and appropriate. General window information, EGL and OpenGL functionality are held in pi3d globals or the Display object.

NB other 3D graphics frameworks pass essentially identical information to the GPU but use different terminology. So in threejs there are: Scene, PerspectiveCamera, WebGLRenderer, Mesh, Geometry, Material etc.

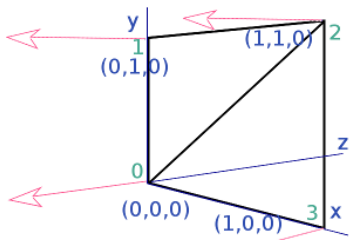
2.2 Sequence of events

3D objects are defined for use in graphics programs starting with a list of points or vertices in space each one needing x, y, z coordinates. Although not generally essential, in pi3d each vertex has a normal vector defined as well. This is effectively an arrow at right angles to the surface at that point and it also needs three values to define its magnitude in the x, y, z directions. The normal vector can be used by the shader to work out how light would illuminate a surface or how reflections would appear. If the normals at each corner of a triangular face are all pointing in the same direction then the fragment shader will treat the surface as flat, but if they are in different directions the surface will appear to blend smoothly from one direction to another. 3D models created in applications such as blender normally have an option to set faces to look either angular or smoothed by calculating different types of normal vectors. Each vertex also has two texture coordinates. These are often termed the u, v position from a two dimensional texture that is to be mapped to that vertex. Again the fragment shader can interpolate points on a surface between vertices and look up what part of a texture to render at each pixel. The crucial piece of information needed by the shader is to define which vertices to use for the corners of each triangle or element. So if I use as an example a very simple one sided square this could be defined by the attribute array:

```
"""   vertices      |    normals      |  texture
                    |                 |  coords
     x    y    z   |  x    y    z    |  u    v
"""
attribute_array = numpy.array(
  [[0.0, 0.0, 0.0,   0.0, 0.0,-1.0,   0.0, 0.0],   # 0
   [0.0, 1.0, 0.0,   0.0, 0.0,-1.0,   0.0, 1.0],   # 1
   [1.0, 1.0, 0.0,   0.0, 0.0,-1.0,   1.0, 1.0],   # 2
   [1.0, 0.0, 0.0,   0.0, 0.0,-1.0,   1.0, 0.0]])  # 3
```

and the element array of triangle indices. Note the order of corners is important. Each triangle 'faces' towards a view where the sequence is clock-wise. Normally the backs of faces are not rendered by the GPU:

```
element_array = numpy.array(
  [[0, 1, 2],
   [0, 2, 3]])
```



Here's a sketch so you can see how the system works.

The GPU uses coordinate directions x increases from left to right, y increases from bottom to top, z increases going into the screen.

The GPU has been designed to be fantastically efficient at performing vector and matrix arithmetic. So rather than the CPU calculating where about the vertices have moved and how these positions can be represented on the 2D computer screen it simply calculates a transformation matrix to represent this and passes that to the GPU. In pi3d we pass two matrices, one representing the object translation, rotation and scale and an additional one including the camera movement and perspective calculations. In the vertex shader these matrices are used to convert the raw vertex positions to screen locations and to work out where the light should come from in order to work out shadows.

Image files are converted into texture arrays that are accessed very efficiently by the GPU.

When pi3d.Buffer.draw() method is called for a 3D object the python side of the program sets the shader and necessary uniform variables to draw the given object. It then works out the 4x4 matrix combining translation, rotation, scale for the object and an additional matrix incorporating the camera movement and lens settings. The camera has two basic modes for handling perspective, the default is 'normal' where things further away are represented as smaller on the screen and the this is defined by a viewing angle between the top edge of the screen and bottom edge. If the camera is set to orthographic mode then objects do not get smaller in the distance and one unit of object dimension corresponds to a pixel on the screen. An orthographic camera can be used to do fast 2D drawing.

The glDrawElements function is then called which sets the vertex shader to work out the locations of each vertex, normal, lighting, texture in terms of screen coordinates. The vertex shader then passes the relevant information to the fragment shader which calculates what colour and alpha value to use for each pixel. The fragment shader takes into account the depth value of each pixel and doesn't draw anything that is behind something it has already drawn. This means that it is more efficient to draw opaque objects from near to far but if something is partially transparent then is must be drawn **after** anything further away that should 'show through'.

pi3d uses a double buffer system where everything is drawn onto an off-screen buffer which, when complete at the end of the frame loop, is swapped 'instantaneously' to visible. This makes the animation much smoother

CHAPTER

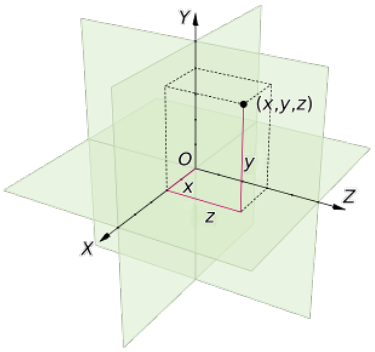
THREE

VECTORS AND MATRICES

This subject could fill many books so the coverage here will be the minimum to understand why the information passed to the GPU is in the form described in the last chapter, how the vectors and matrices are generated by the pi3d code, and what the GPU does with them. I would strongly advise you to find out more about vectors by reading about them elsewhere; they're great!

3.1 Vectors

The classic definition of a vector is something that has `Magnitude` and `Direction` - a value having only magnitude being termed `scalar`. Arrows are often used to represent vectors but, although this analogue is very easy to understand, it is also a slight distraction that can make further understanding more difficult. The crucial thing about vectors is that they have more than one **component**. So whenever a value has to be uniquely defined like (x, y, z) or even (R, G, B) then that makes it a vector.



So the way a surface "points" (the normal) if often drawn as an arrow perpendicular to the surface and this can be easily understood as a vector (as can the direction of a light "ray" hitting the surface). However positions of vertices, texture coordinates, movements and rotations are all vectors as well.

At this point it's worth thinking a little about the vector representation of rotations. A logical approach is to define the direction of an axis of rotation using three coordinates with the amount of rotation depending on the overall magnitude of the three values. However if you play around with a small box (book, mug etc), pretending it's the "camera" used to view a scene, you will see it's not so simple. For instance tilting the camera about the horizontal x axis (running from left to right) through 90 degrees so it's pointing straight down, then rotating it about the vertical y axis (in GPU terms) through 90 degrees would require Euler [1] to figure out about which axis it had rotated and by how much. What's more if order of rotation is y first then x it ends up in a different position. In pi3d a rotation vector *(A, B, C)* is interpreted as first rotate C about the z axis (roll), then rotate A about the x axis (pitch), finally rotate B about the y axis (yaw) as this produces the most intuitive results!

3.2 Matrices

Matrices are really a short-hand way of holding structured information, and from that perspective are indistinguishable from programming arrays:

[1] http://en.wikipedia.org/wiki/Euler_angles#Relationship_to_other_representations

```
M = [[1.2, 0.0, 0.0, 1.0],
     [0.0, 2.2, 1.5. 1.0],
     [0.8, 0.2, 3.2. 0.0],
     [0.0, 0.0, 0.0. 1.0]]
```

However very useful properties have been defined and implemented in maths and subsequently programming languages that enable efficient and **fast** calculations involving vectors. And, as we've just seen, vectors are the natural way to represent the components of 3D graphics.

The essential things to grasp without getting bogged down in the details are:

1. Matrices can "operate" on vectors resulting in translation (moving in some direction), scaling or rotation.
2. Matrices can "operate" on other matrices to produce a new matrix with a combined effect. So in pseudo-code:

```
# starting vector v
v = T1(v) # apply translation matrix function to v
v = R1(v) # then rotate it
v = S1(v) # then scale it (etc etc)
v = P1(v) # then convert it to 2D screen coordinates using perspective!
# which you could write as
v = P1(S1(R1(T1(v))))
# with matrix maths you can do
# M = P1 x T1 x R1 x S1 # termed "matrix multiplication"
M = P1(S1(R1(T1))) # or in our pseudo functional code
v = M(v)
```

And the reason this is useful is that we can do a relatively small amount of matrix manipulation in the python program to represent movement of shapes or the camera and "simply" pass the modified matrices to the GPU each frame for it to number crunch the actual pixel values.

3.3 Illustrations

Now is probably a good time to look at the first illustration program **2D_matrix01.py** [2] (open a copy in an editor on your computer so you can run it as well as view it)

The objective is to get an appreciation of how matrices can be used to modify vectors so, at this stage, don't worry about how pi3d is being used to display the output. Display, Camera, Shader, Lines, Keyboard, Font will be covered in later chapters. The whole process is inevitably complicated-looking as these details are the very thing that is done "behind the scenes" by pi3d or by the GPU! (Especially don't be put off by the very complicated procedure to get numbers to appear near the corners)

The main bits to look at are where there are docstring explanations. There are three types of matrix defined which you can modify by pressing the keys w,a,s,d,z,x,c,v. There is also a printout of the matrices each time you press a key, to fit them in nicely you will probably have to "stretch" the terminal window to make it wide enough. Spend a reasonable time figuring out what's happening before you move on.

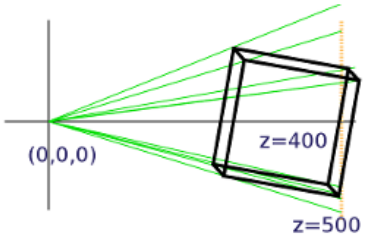
In **3D_matrix01.py** [3] there is an expansion into three dimensions so the transformation matrices become 4x4. If you are unclear why this is necessary it may be a good idea to go back and look at the first illustration.

Because the computer screen is essentially flat there has to be a method of converting the (x, y, z) vectors of the model into (x, y) vectors of the screen. The simplest way would be to just ignore the z values, and this is effectively what the "orthographic" projection does (when setting the Camera object up in line 10 I set the argument `is_3d=False`) For perspective projection there has to be a "scaling down" of x and y coordinates with distance, which is achieved using the matrix p_mat. When this operates on the vertex a scaling factor is calculated and put into the fourth "slot" of

[2] https://github.com/paddywwoof/pi3d_book/blob/master/2D_matrix01.py
[3] https://github.com/paddywwoof/pi3d_book/blob/master/3D_matrix01.py

the resultant vector. In line 67 you will see that in this manual version the x and y (and z but not needed here) values are divided by the scaling factor. On the GPU the scaling is done automatically, and this is the reason why the vertex position vectors used in the OpenGL shaders are of the form `(x, y, z, w)` i.e. four dimensional.

Note also that the perspective modifications to the x and y values are done after the x, y and z values of the vertices have been recalculated using the transformation matrices. The scaling is done from a view point at the origin (0, 0, 0) and this is why the cube has to be displaced 400 units in the z direction to be "within shot". If we want to modify the view by moving the camera as well as the objects in the scene (as in "first person view" games such as minecraft) then this is achieved by translating and rotating **everything else** in the opposite sense to the camera. i.e. in this example if the camera were to move +50 in the z direction and +50 in the x direction it would be achieved by moving the cube (-50, 0, -50). These transformations are rolled up into the camera view matrix that is passed to the GPU.



In pi3d (and 3D graphics generally) the scaling factor is calculated using a `field of view` angle, a screen width to height ratio, a `near` plane and a `far` plane. There is a nice interactive demo here http://webglfundamentals.org/webgl/frustum-diagram.html

3D_matrix02.py [4] switches from doing all the matrix operations manually to using the standard 3D functionality of pi3d and OpenGL. Ideally there should be no difference between the behaviour of this program and the last one apart from the switch to Fortran style matrices mentioned in the docstrings, however it's **much** faster though this will not be apparent with such a simple model! It's also dropped from 151 to 90 lines of code (excluding comments).

3D_matrix03.py [5] finally uses a pi3d.Cuboid object instead of constructing a skeleton from lines. In this program there are two Shaders, the one passed to the Lines objects (xaxis and yaxis) is "mat_flat" and the one passed to the Cuboid object (cube) is "mat_light". The result is that the sides of the cube behave as if illuminated by a directional light as it is rotated. The way that the shaders produce the lighting effect will be covered in a later chapter but now it's time to move away from this slightly theoretical background and start to see how the pi3d classes fit together and how they can be used in practice.

[4] https://github.com/paddywwoof/pi3d_book/blob/master/3D_matrix02.py
[5] https://github.com/paddywwoof/pi3d_book/blob/master/3D_matrix03.py

CHAPTER

FOUR

SHAPES, BUFFERS AND DISPLAY

This is a rather technical chapter with only a few examples and demos, however it takes a look inside some of the pi3d source code with two-fold aims: one is to see how the information needed by the shader is held by pi3d objects and the way it is sent when the draw() method gets called, the other aim to get used to opening up the source code of the module to figure out any problems using it.

In the the pi3d documentation ReadMe there is an ultra minimal example:

```
import pi3d
DISPLAY = pi3d.Display.create()
ball = pi3d.Sphere(z=5.0)
while DISPLAY.loop_running():
  ball.draw()
```

Which seems to be at odds with the requirement that there has to be a Camera, Light and Shader object in existence to draw any of the standard Shapes. The answer is that all these classes inherit from the DefaultInstance class as explained in the Shape.draw() description below.

If you open pi3d/shape/Sphere.py in an editor you will see that it is relatively brief. Almost all of the functionality comes from its parent class Shape and this is the case for everything in the pi3d/shape directory:

```
Building    ElevationMap     LodSprite    Sphere       Tube
Canvas      EnvironmentCube  MergeShape   Sprite
Cone        Extrude          Model        TCone
Cuboid      Helix            MultiSprite  Tetrahedron
Cylinder    Lathe            Plane        Torus
Disk        Lines            Points       Triangle
```

4.1 Shape

Have a look at the source code for Shape. *Don't be dismayed by how long it is - the majority of it is just convenience methods for*:

1. setting the uniform variables array (`self.unif`, remember uniform variables from chapter two - one of the four categories of data passed to the GPU shaders),
2. setting the uniform variables held in the Buffer list (`self.buf[0].unib`, I will explain the relationship between Shapes and Buffers below),
3. updating the matrices (see `rotateIncY()` on line 665, you've already used that method in 3D_matrix02.py and the process of writing sines and cosines into an array should be reassuringly familiar!)

However the draw() method does several important things. Firstly, on lines 163, 164 and 37 (which is in __init__() actually!) you will see the method instance() being called for Camera, Shader and Light. These three classes inherit

from the DefaultInstance class and the method will either return the first instance of that class that has been created, or if none, it will create one.

Most of the time the default Light is fine - it's a neutral directional light. The default Camera is also what you want for normal 3D viewing, but there are occasions when you need to overlay 2D objects in front of a 3D scene and this can be done by using two Camera instances and assigning each to different objects. The default Shader is much more of a fall-back position. This is because it has to be a "material" based Shader rather than one that relies on Textures being loaded. When we look inside the Buffer class you will see why a default material can be set easily but default textures would be messy.

The second thing to look at in the Shape.draw() method is the section from line 167 to 205. This is basically the matrix multiplication we did by hand in 2D_matrix01.py and 3D_matrix01.py Because this has to be done for every object in the scene in every frame it is time critical and this has been found to be the fastest combination 1) use numpy dot() [1] 2) set flags everywhere and only do the dot() when something has moved or rotated. Line 210 is where two 4x4 matrices are passed to the shader and line 214 passes twenty 3x1 vectors, the Shape.unif array.

Before we follow line 218 to the Buffer.draw() we'll just have a quick scan through the Shape.unif array which occupies lines 38 to 49 (with a comprehensive description of what it all is underneath it). The first twelve values are taken from arguments to the __init__() method and only offset should need any explanation. This allows objects to be rotated about different points than their self origin. Fog is a shade and alpha value that is "merged" by the Shader starting at a third of fog distance and increasing to 100% fog values at the full fog distance. Shape alpha allows objects to become partially or completely transparent. The Light values get stored here, in each Shape, even though there is a separate Light object. This means that it's possible to illuminate objects within a scene with different lights. Although there looks to be space for two lights for each Shape all the Shaders (so far) only calculate illumination on the basis of the first one. Lights will be discussed in a later chapter but they essentially have a flag to mark them as "point" or "directional" which determines how the x,y,z vector is interpetted, an RGB color value for the light and RGB for ambient. The final eighteen values are available for special shader effects.

N.B. If you are eagle-eyed and have been paying attention you will have noticed a "proteted" function [2] in Shape `_lathe()` that is used by the majority of the pi3d/shape classes. This will be investigated in a later chapter.

4.2 Buffer

This class gets its name because it's the object used to hold the attribute array and element array which are both created by calling the OpenGL function `glBufferData()`. The reason why it's a separate class (rather than just being part of Shape) is that one Shape can be constructed from several parts, each with its own Texture or material properties. This is particularly true of Model object i.e. Shapes that have been designed elsewhere and saved as `obj` or `egg` files.

The Buffer class is also complicated-looking and has more opengles function calls than Shape. There are a few things worth noting about this class

1. The "constructor" __init__() takes lists of vertices, normals, texture coordinates and element indices, as we would expect. However if the normals argument passed is `None` it will calculate a set of vectors at right angles to both the triangle edges that meet at each vertex [3]. It can also be made to construct smaller buffers by being passed empty lists for the texture coordinate and or the normals when these are not needed i.e. for Lines, Points or a non-texture-mapped Shape.
2. The draw() method (which is called by Shape.draw() as we saw above) passes the attribute and element arrays to the Shader on lines 283 to 292 and on line 318 four 3x1 vectors, from Buffer.unib (which I will explain in more detail below). draw() also passes the Texture samplers from line 297. NB when I say "pass" the data it is

[1] numpy http://www.numpy.org/ is an important addition to python that can dramatically improve performance. Although it's quite hard to get the hang of, it's definitely worth persisting.

[2] python doesn't have formal name-space control seen in other languages where attributes and methods are declared public, private, protected etc. However the convention is to use underscores as the first letter to indicate that a method is not intended for "external" use. Similarly pi3d adopts standard upper case names to denote global "static" variables.

[3] using cross product http://en.wikipedia.org/wiki/Cross_product

only the pointer to the data that needs to be transferred, the actual arrays were set up in the GPU memory space when the Buffer was created and just need to be switched on (which is very quick). However...

3. There is a re_init() method that can be used to alter the values of the vertex, normal or texture coordinate vectors from frame to frame. This requires more processing than simply enabling data that is already there but it is much faster than scrapping the previous Buffer object and creating a complete new one.

   Moving vertices, normals or texture coordinates isn't something that needs to be done very often but it might make an entertaining exercise in this otherwise fairly wordy chapter. Copy the example program from the start of this chapter into an editor and make sure it runs OK (there's no way of stopping it as it stands apart from Ctrl+C to break or closing the window). Then add some distortion, straight after `ball.draw()` at the same indent along the lines of:

```
bufr = ball.buf[0]         # there's only one Buffer in the list buf
b = bufr.array_buffer      # this is the array buffer!
l = len(b)                 # the length of the array buffer (195 actually)
import numpy as np         # python is clever enough not to do this every loop!
b[:,0:3] *= np.random.uniform(0.99, 1.01, (l, 3)) # numpy slicing, see below
bufr.re_init(pts=b[:,0:3]) # finally re make the buffer
```

   If you are not used to numpy you will probably be bamboozled by the fifth line. This is how numpy works: the looping is done "automatically" as a result of the slicing or the shape of the arrays involved. Using python list comprehension this would achieve the same result:

```
new_buf = [[b[i,j] * random.uniform(0.99, 1.01) for j in range(3)] for i in range(l)]
bufr.re_init(pts=new_buf)
```

   And good old straightforward, easy to understand looping:

```
new_buf = []
for i in range(l):
  new_buf.append([])
  for j in range(3):
    new_buf[i].append(b[i,j] * random.uniform(0.99, 1.01))
bufr.re_init(pts=new_buf)
```

   The reason for this apparent regression to a less obvious code format is **speed**. If you test the three alternatives with `timeit` you will find that the traditional looping takes 2.2ms, the list comprehension takes 1.95ms and numpy takes 0.08ms, a massive margin that only increases as the array gets bigger.

The Buffer.unib array of uniform variable passed to the Shader needs a bit more explanation that the equivalent array in Shape. **ntile** is used to control how many normal map Texture maps (also called bump maps) are to be tiled for each unit of texture coordinates. Normal maps will be explained fully in the chapter on Light and Texture but they are a way of adding structural detail to a surface without having to make it from millions of vertices - have a search on google if you're curious. **shiny** controls how much reflection is produced by the mat_reflect or uv_reflect Shaders. **blend** is a variable set during the draw() process depending on whether the Texture values taken from the image file are to be blended or not. If the alpha value of the pixel is below this value then the pixel will be "discarded". This will be covered in detail later but it allow sharp edges to be produced around shapes constructed from png images with large areas of transparency. **material** is the RGB values for this Buffer when drawn using a material Shader. **umult** and **vmult** control the tiling of the Texture map (the color one as opposed to normal or reflection). **point_size** is the size in pixels of vertices drawn by the Points object. **u_off** and **v_off** are the tiling offsets that go with vmult and umult. **line_width** is the size in pixels drawn by the Lines object.

4.3 Display

Although there could have been a case for making Display create a default instance of itself in the same way that Camera, Light and Shader do, there are various reasons why this would be messy, the most obvious of which relate to

how the main loop would be structured.

In pi3d we have chosen to make the loop an explicit python `while ..:` with a call to a central `loop_running()` used to do various once-per-frame tasks, tidy up GPU buffers and close things down when the program finishes. Many other user interface frameworks have adopted a more event oriented "hidden" loop style but our reasoning was that it only makes sense to use the GPU and OpenGL where frames per second are of central importance, so in pi3d the main loop is made very visible.

Open pi3d/Display.py in an editor and look though the docstrings, all of which should be self-explanatory. There are some more obscure but occasionally useful arguments to the create() function [4].

[4] something not explained very clearly elsewhere is the `samples` argument to create() this can be set to 4 and will give much better anti-aliasing i.e. prevent the stepped edges of diagonals of contrasting color. However using this will prevent pi3d from creating a Display on computers running windows. This might be because of the functionality of DirectX and the OpenGL ES emulators for windows (even Linux virtual machines under windows).

CHAPTER

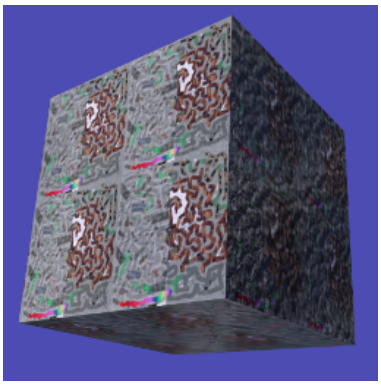
FIVE

TEXTURES, LIGHTS AND SHADERS

We've touched on the roles of these three classes previously but in this chapter I hope to give much more detail of how they fit together and how they can be used.

5.1 Textures

First of all have a look at the next illustration program **textures01.py** [1] and run it to see what it does. The code starts from **3D_matrices03.py** but replaces the yellow material of the cube with an image texture, the docstrings explain the changes.



5.2 Lights

Before looking at the next texture example it would be good to get more of an idea how Light works; so open up and run the **light01.py** [2] example. Again, much of the explanation that I would have put here is in the docstrings so read them and try the experiments suggested in the text.

Now work your way through **textures02.py** [3] which is using all the functionality available in the "standard" shaders. There are lots of variables to tweak and experiments to do with this example so work your way though it slowly and carefully.

5.3 Shaders

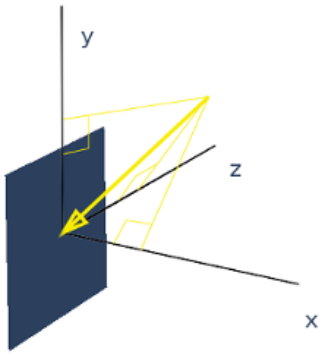
In the next illustration we will look at what the shader is doing to a) look up the texture values for a given pixel b) adjust for lighting. However the code to get the normal map and reflection map is rather complicated so I will only give an outline description of that here (if you want to look at it in detail you will have to read through the shader yourself!)

Caution the language that shaders use (GLSL) is C-like in syntax, but that in itself shouldn't be a problem, the confusing aspect is that variables can be "different shapes". Bearing in mind that in GLSL (as in C) variable types have to be explicitly defined:

[1] https://github.com/paddywwoof/pi3d_book/blob/master/textures01.py
[2] https://github.com/paddywwoof/pi3d_book/blob/master/light01.py
[3] https://github.com/paddywwoof/pi3d_book/blob/master/textures02.py

```
float a = 4.12;
vec2 b = vec2(4.12, 5.23);
vec3 c = vec3(4.12, 5.23, 7.34);
a = mod(a, 3.1416); // python equivalent would be a % 3.1416
b = mod(b, 3.1416);
c = mod(c, 3.1416);
b = mod(b, vec2(3.1416, 6.2832);
c = mod(c, vec3(3.1416, 6.2832, 9.4248);
```

You will see that generally speaking variables can be vectors which the compiled GLSL is designed to process very fast. On the other hand branching and conditional statements are very slow and this sometimes results in strange program structure and use of built in functions such as `step()` and `clamp()`.



One final bit of explanation before looking at the next example. The dot product of two vectors is often described as "the length of one times the length of the other times the cosine of the angle between them". This is reasonably easy to apprehend in two dimensions, and when we can think of the vectors as arrows. However in four dimension when the vectors represent RGBA values it's not so intuitive. A better informal description would be "how much of one vector is in the same direction as the other" it's still easy to see how this applies to light illuminating a surface but it's much easier to see that the dot function doesn't need to do any (slow) trigonometry, it is sufficient to multiply the x,y,z components together and this is very fast:

```
normal = vec3(1.0, 0.0, 0.0);   // surface facing in the same direction as x axis
light = vec3(-2.5, -2.5, -2.5); // light down, from right, out of the screen
float a = dot(normal, light);   // results in -2.5 # i.e. (1.0 * -2.5) + (0.0 * -2.5) + (0.0 * 2.5)
```

So now have a look at **shader01.py** [4] and play around with it. Any typos or errors in the two shader scripts will be hard to track down so proceed with caution (remember Ctrl-z can get you back to a working version!). Also, because the GLSL is embedded in strings in the python code, the chances are that any code formatting in your editor will not be brilliant, so here is the code again. Vertex Shader:

```
precision mediump float;
attribute vec3 vertex;           // these are the array buffer objects
attribute vec3 normal;           //  defined in Buffer
attribute vec2 texcoord;

uniform mat4 modelviewmatrix[2]; // [0] model movement [1] projection
uniform vec3 unib[4];
/* umult, vmult => unib[2][0:1] # these are defined in Buffer
   u_off, v_off => unib[3][0:1] */
uniform vec3 unif[20];
/* eye position => unif[6][0:3] # defined in Shape
 light position => unif[8][0:3] */

varying vec2 texcoordout; // these have values set in the vertex shader which
varying vec3 lightVector; // are picked up in the fragment shader. However
varying float lightFactor;// their values "vary" by interpolating between vertices
varying vec3 normout;

void main(void) {
  if (unif[7][0] == 1.0) { // this is a point light and unif[8] is location
    // apply the model transformation matrix
```

[4] https://github.com/paddywwoof/pi3d_book/blob/master/shader01.py

```
    vec4 vPosn = modelviewmatrix[0] * vec4(vertex, 1.0);
    // to get vector from vertex to the light position
    lightVector =  unif[8] - vec3(vPosn);
    lightFactor = pow(length(lightVector), -2.0); // inverse square law
    lightVector = normalize(lightVector);         // now convert to unit vector for direction
  } else {                                        // this is directional light
    lightVector = normalize(unif[8]) * -1.0;      // directional light
    lightFactor = 1.0;                            // constant brightness
  }
  lightVector.z *= -1.0;                          // fix r-hand axis
  normout = normalize(vec3(modelviewmatrix[0] * vec4(normal, 1.0))); // matrix multiplication
  texcoordout = texcoord * unib[2].xy + unib[3].xy; // offset and mult for texture coords
  gl_Position = modelviewmatrix[1] * vec4(vertex,1.0); // matrix multiplication
  /* gl_Position is a pre-defined variable that has to be set in the vertex
  shader to define the vertex location in projection space. i.e. x and y
  are now screen coordinates and z is depth to determine which pixels are
  rendered in front or discarded. This matrix multiplication used the full
  projection matrix whereas normout used only the model transformation matrix*/
}
```

and Fragment shader:

```
precision mediump float;
uniform sampler2D tex0; // this is the texture object
uniform vec3 unib[4];
/*     blend cutoff => unib[0][2] # defined in Buffer */
uniform vec3 unif[20];
/*      shape alpha => unif[5][2] # defined in Shape
          light RGB => unif[9][0:3]
  light ambient RGB => unif[10][0:3] */


varying vec3 normout; // as sent from vertex shader
varying vec2 texcoordout;
varying vec3 lightVector;
varying float lightFactor;

void main(void) {
  gl_FragColor = texture2D(tex0, texcoordout); /* look up the basic RGBA value
  from the loaded Texture. This function also takes into account the distance
  of the pixel and will use lower resolution versions or mipmaps that were
  generated on creation (unless mipmaps=False was set)
  gl_FragColor is another of the pre-defined variables, representing the
  RGBA contribution to this pixel */
  // try making it a "material" color by swapping with the line above
  //gl_FragColor = vec4(0.7, 0.1, 0.4, 0.9);
  // to allow rendering behind the transparent parts of this object:
  if (gl_FragColor.a < unib[0][2]) discard;
  // adjustment of colour according to combined normal:
  float intensity = clamp(dot(lightVector, normout) * lightFactor, 0.0, 1.0);
  // try removing the 0 to 1 constraint (with point light):
  //float intensity = dot(lightVector, normout) * lightFactor;
  // directional lightcol * intensity + ambient lightcol:
  gl_FragColor.rgb *= (unif[9] * intensity + unif[10]);
  gl_FragColor.a *= unif[5][2]; // finally modify the alpha with the Shape alpha
}
```

There is a khronos GLSL quick reference card [5] if you want to see what all the functions do.

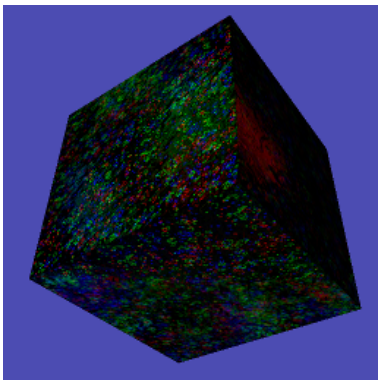
I mentioned above that I would give a general description of how the normal map and reflection map work. If you have attempted to look at the shader code "really" used in pi3d you will have found that it is structured with lots of #includes so that common sections can be re-used - this makes it quite hard to reconstruct. You may have also seen that the normal vector is not passed from the vertex to fragment shader as shown in this example. Instead the light vector is rotated in the vertex shader by a complicated process (Euler angles again) so that it is correctly oriented relative to the normal vector at that vertex *if that vector was pointing straight out of the screen* i.e. in the -ve z direction!

The reason for this complication is that it then allows the fragment shader to modify the normal vector by simply adding values from the RGB of a normal map texture. Values of red less than 0.5 make the x component of the normal negative, greater than 0.5 positive. The green values control the y component in a similar way.

The reflection map works out the vertical and horizontal angles that a line drawn from the camera to a given pixel would be reflected. The reflection uses the normal vector at each pixel adjusted by the normal map as described above. The reflection angles are then used to look up a position from a Texture where the horizontal range is -pi to +pi (+/- 180 degrees) and the vertical range is -pi/2 to +pi/2 (+/- 90 degrees) This is the standard projection used for photo-spheres.

5.4 A Final look at Textures

As a final bit of fun have a look at the **textures03.py** [6] demo. This illustrates how Texture objects can be constructed from numpy arrays (or PIL Image objects) and can be updated each loop. Although numpy or PIL are much faster than python (they are compiled modules written in C) they are not as fast as GPU shaders. However for some applications it can be very convenient to manipulate reasonably small textures in this way. The VideoWalk.py demo (on github.com/pi3d/pi3d_demos/) shows how video frames can be read using ffmpeg and used to update a Texture. This is also a way to use OpenCV images as Textures as they are already numpy arrays.



[5] https://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf
[6] https://github.com/paddywwoof/pi3d_book/blob/master/textures03.py

CHAPTER

SIX

CAMERAS, 2D PROJECTION AND SPRITES

Although pi3d, and OpenGL generally, are aimed at making 3D rendering efficient and fast they allow 2D rendering to be done equally well. In fact we have already used 2D rendering in the first few examples in the Vectors and Matrices chapter. In that chapter I mention in passing that for a 2D or `orthographic` projection the matrix multiplication simply needs to preserve the x and y coordinates - no scaling with distance needs to be done. I also mention that, in pi3d, this can be achieved by setting the Camera argument is_3d=False.

So this is the general scheme for managing different types of rendering in pi3d: Projection matrices are held in Camera objects. When a Shape is drawn either an explicitly assigned Camera or a default one will be used and the projection matrix passed to the shader as part of the `uniform modelviewmatrix` (as we saw at the end of the last chapter). Although it's possible to have as many different Cameras as you want it's normally sufficient to have just one 3D for rendering 3D objects that possibly moves around the virtual environment under user control, and one 2D that remains fixed and allows "dashboard" information to be displayed.

Have a look at the two demo programs Minimal.py and Minimal_2d.py from https://github.com/pi3d/pi3d_demos There are a couple of noteworthy differences between 3D and 2D projections:

Scale In 3D projection (with the default lens settings) 1.0 unit of x or y at a z distance of 1.1 just about fills the screen (in the vertical direction; the `field of view` is defined vertically). Try adding variations of this line after the DISPLAY creation in Minimal.py:

```
CAMERA = pi3d.Camera(lens=(1.0, 1000.0, 25, DISPLAY.width / float(DISPLAY.height)))
```

In 2D projections 1.0 unit of x or y equates to 1 pixel

z distance In 2D projections the effective distance of objects (to determine what gets drawn in front of what) is:

```
10000 / (10000 - z_2D)
```

So if you set a 2D object at z=20.0 it will be drawn in front of a 3D object at z=1.002 (effectively in front of **all** 3D objects). To draw a 2D object behind a 3D object at z=20.0 it must be moved to z=9500.0 However the relative positions of 2D objects with respect to each other work as you would expect so a 2D object drawn at z=20.0 is in front of a 2D object at z=21.0

In Minimal_2d.py try changing rotateIncZ to rotateIncX (or Y). Do you see the effect of moving some of the object in front of the `near plane`? To keep in in view you need to move it further away. In the following line `sprite.position()` increase the z value from 5.0 to 50.0.

Open the demo file Blur.py. This has various features I haven't explained yet: MergeShape, Defocus, Font and String. You can probably figure how they're being used but don't worry about that at the moment. Just look at lines 78 and 79 where the String is defined. You will see that it uses a 2D camera, try increasing the z value. The text will only coincide with the balls when you increase z above 8000

The most common way to use the 2d Camera is with the Sprite class which maps a rectangular Texture to a quad formed by two triangles. The Minimal_2d.py demo above shows the basic use via the ImageSprite class which wraps

up the Texture loading and Shader allocation. Have a look at **sprites01.py** [1] which shows why the order of drawing objects matters for blending.

The Sprite class in pi3d is an easy way to do 2D rendering of images, however it runs into processing limitations if it is used for sprites in the sense of a 2D animation or game. Where there are hundreds or thousands of images moving about (think "creeps + towers + missiles" in a tower defence game) then it is more efficient to use the OpenGL point drawing functionality which will be touched on in the chapter `Lines and Points` and demonstrated in SpriteBalls.py and SpriteMulti.py

[1] https://github.com/paddywwoof/pi3d_book/blob/master/sprites01.py

CHAPTER

SEVEN

MODELS

CHAPTER

EIGHT

CONSTRUCTING A SHAPE FROM SCRATCH

CHAPTER

NINE

USER INPUT

CHAPTER

TEN

LINES AND POINTS

CHAPTER

ELEVEN

VIDEO AND SOUND

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*