3D Graphics with pi3d

Release 1.0

Paddy Gaunt

August 02, 2015

Contents:

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 ba-

---

[1] http://www.numpy.org/
[2] https://docs.python.org/2/library/ctypes.html
[3] http://en.wikipedia.org/wiki/OpenGL_Shading_Language

sic 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.

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.

---

[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

---

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 do what you expect you are encouraged to open the source code with an editor and figure out what the problem is.

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.

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.
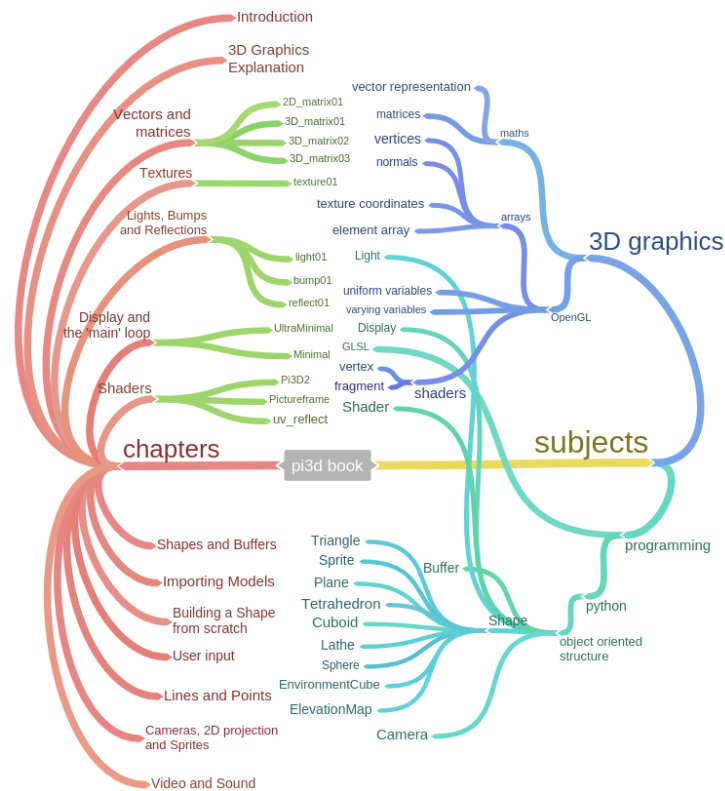
Figure 1.1: Outline mind map from Coggle.it

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

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

---

[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

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 how the entries

---

[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!

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]])
```

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

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.
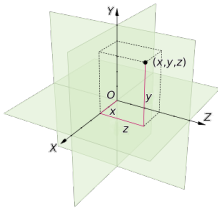
At this point it's worth thinking a little about the vector representation of rotations. A logical approach is to define the direction of an axis of rotation using three coordinates with the amount of rotation depending on the overall magnitude of the three values. However if you play around with a small 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:

```
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.

---

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

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 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

---

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

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.
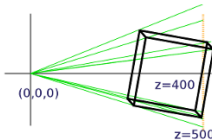
Because the computer screen is essentially flat there has to be a method of converting the (x, y, z) vectors of the model into (x, y) vectors of the screen. The simplest way would be to just ignore the z values, and this is effectively what the "orthographic" projection does (when setting the Camera 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 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

---

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

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

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 is 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:

```python
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

---

[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 con-

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 263 to 270 and on line 296 four 3x1 vectors, from Buffer.unib (which I will explain in more detail below). draw() also passes the Texture samplers from line 275. NB when I say "pass" the data it is only the pointer to the data that needs to be transferred, the actual arrays were set up in the GPU memory

---

vention 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

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]        # only one Buffer in the list buf
b = bufr.array_buffer     # this is the array buffer!
l = len(b)                # length of the array (195 actually)
import numpy as np        # python will do this just once!
b[:,0:3] *= np.random.uniform(0.99, 1.01, (l, 3)) # 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)
```

3D Graphics with pi3d, Release 1.0

> 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 mas-
> sive 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 than 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 hav-
ing 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 ma-
terial 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 de-
fault instance of itself in the same way that Camera, Light and Shader
do, there are various reasons why this would be messy, the most ob-
vious 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 vari-
ous 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).

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.
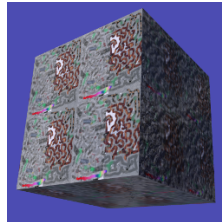
5.2 Lights

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

---

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

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:

```
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()`.

---

[3] https://github.com/paddywwoof/pi3d_book/blob/master/textures02.py

One final bit of explanation before look-
ing at the next example. The dot prod-
uct 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 appre-
hend 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:
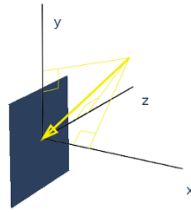
```
// surface facing in the same direction as x axis
normal = vec3(1.0, 0.0, 0.0);
// light down, from right, out of the screen
light = vec3(-2.5, -2.5, -2.5);
// results in -2.5 # i.e. (1.0 * -2.5) + (0.0 * -2.5) + (0.0 * 2.5)
float a = dot(normal, light);
```

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 work-
ing 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:

```
1  precision mediump float;
2  attribute vec3 vertex;              // these are the array buffer objects
3  attribute vec3 normal;              //  defined in Buffer
4  attribute vec2 texcoord;
5
6  uniform mat4 modelviewmatrix[2]; // [0] model movement [1] projection
7  uniform vec3 unib[4];
8  /* umult, vmult => unib[2][0:1] # these are defined in Buffer
9     u_off, v_off => unib[3][0:1] */
10 uniform vec3 unif[20];
```

------

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

```
11  /* eye position => unif[6][0:3] # defined in Shape
12   light position => unif[8][0:3] */
13
14  varying vec2 texcoordout; // these values set in vertex shader which
15  varying vec3 lightVector; // are picked up in the fragment shader.
16  varying float lightFactor;// However their values "vary" by
17  varying vec3 normout;     // interpolating between vertices
18
19  void main(void) {
20    if (unif[7][0] == 1.0) { // a point light; unif[8] is location
21      // apply the model transformation matrix
22      vec4 vPosn = modelviewmatrix[0] * vec4(vertex, 1.0);
23      // to get vector from vertex to the light position
24      lightVector =  unif[8] - vec3(vPosn);
25      lightFactor = pow(length(lightVector), -2.0); // inverse square law
26      lightVector = normalize(lightVector); // to unit vector for direction
27    } else {
28      lightVector = normalize(unif[8]) * -1.0;    // directional light
29      lightFactor = 1.0;                          // constant brightness
30    }
31    lightVector.z *= -1.0;                        // fix r-hand axis
32    // matrix multiplication
33    normout = normalize(vec3(modelviewmatrix[0] * vec4(normal, 1.0)));
34    // offset and mult for texture coords
35    texcoordout = texcoord * unib[2].xy + unib[3].xy;
36    gl_Position = modelviewmatrix[1] * vec4(vertex,1.0);
37    /* gl_Position is a pre-defined variable that has to be set in the
38    vertex shader to define the vertex location in projection space.
39    i.e. x and y are now screen coordinates and z is depth to determine
40    which pixels are rendered in front or discarded. This matrix
41    multiplication used the full projection matrix whereas normout
42    used only the model transformation matrix*/
43  }
```

and Fragment shader:

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

```
11  varying vec3 normout; // as sent from vertex shader
12  varying vec2 texcoordout;
13  varying vec3 lightVector;
14  varying float lightFactor;
15
16  void main(void) {
17    gl_FragColor = texture2D(tex0, texcoordout); /* look up the basic
18    RGBA value from the loaded Texture. This function also takes into
19    account the distance of the pixel and will use lower resolution
20    versions or mipmaps that were generated on creation
21    (unless mipmaps=False was set)
22    gl_FragColor is another of the pre-defined variables, representing
23    the RGBA contribution to this pixel */
24    // try making it a "material" color by swapping with the line above
25    //gl_FragColor = vec4(0.7, 0.1, 0.4, 0.9);
26    // to allow rendering behind the transparent parts of this object:
27    if (gl_FragColor.a < unib[0][2]) discard;
28    // adjustment of colour according to combined normal:
29    float intensity = clamp(dot(lightVector, normout) *
30                                   lightFactor, 0.0, 1.0);
31    // try removing the 0 to 1 constraint (with point light):
32    //float intensity = dot(lightVector, normout) * lightFactor;
33    // directional lightcol * intensity + ambient lightcol:
34    gl_FragColor.rgb *= (unif[9] * intensity + unif[10]);
35    gl_FragColor.a *= unif[5][2]; // modify alpha with the Shape alpha
36  }
```

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!

---

[5] https://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-
Reference-card.pdf

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
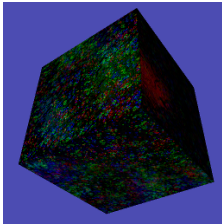
As a final bit of fun have a look at the **textures03.py** [6] demo. This illustrates how Texture objects can be constructed from numpy arrays (or PIL Image objects) and can be updated each loop. Although numpy or PIL are much faster than python (they are compiled modules written in C) they are not as 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.

---

[6] https://github.com/paddywwoof/pi3d_book/blob/master/textures03.py

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:

3D Graphics with pi3d, Release 1.0

> **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,    # near plane
                           1000.0, # far plane
                           25,     # field of view
                           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 ob-
> jects (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 ob-
> ject 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 it 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 ex-
plained yet: MergeShape, Defocus, Font and String. You can prob-
ably 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 Image-Sprite 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

SEVEN

MODELS

3D models can be made and edited using a wide variety of software and saved in nearly as large a variety of formats. In pi3d, at the moment, there are just two file types that can be parsed: wavefront **obj** and Panda3D **egg**. Of these two the obj is simpler and faster to load.
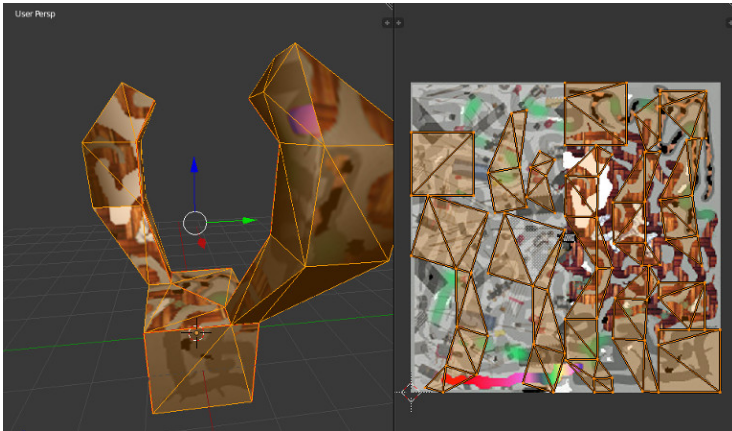

Figure 7.1: Blender screen-capture

If you have **blender** [1] installed on your PC then you can open the

---

[1] http://www.blender.org/

file **model01.blend** [2] or you can view it by running the pi3d demo **model01.py** [3] It's basically a box with horns! In the diagram above the left view is normal perspective (as with pi3d) and the right view is the uv "unwrapping" for texture mapping to the vertices.

Open the **blender01.obj** and **blender01.mtl** files and have a look at the structure of the information, it should be reasonably familiar by now with a little explanation.

Most of the obj file consists of four types of lines:

> **v vertex** lines with x, y, z coordinates.

> **vt vertex texture** lines with u, v coordinates.

> **vn vertex normal** lines with x, y, z components of normals.

> **f face** lines with a series of references to v/vt/vn lines for each corner of the face. In general there could be more than three corners and the parser function has to convert it into triangles to work with OpenGL ES2.0

Additional occasional lines are **mtllib model01.mtl** which points to material file. **o Cube** define different objects within this file. In pi3d these will each be a different Buffer object within one Shape. **usemtl Material** instructs the properties from mtllib under "Material" to be used for the following faces. **s off** and **s 1** turn smoothing off and on. Pi3d doesn't use these instruction but does use the normals. If you look at the lines:

```
s off
f 3/1/1 2/2/1 9/3/1 7/4/1
f 13/5/2 14/6/2 9/7/2 2/8/2
f 3/9/3 7/10/3 8/11/3
```

You will see three faces using vertices (4,2,9,7) (13,14,9,2) (3,7,8) with normals (1,1,1,1) (2,2,2,2) (3,3,3) i.e. all corners are facing the same direction. NB the parser will convert each of the first two quads into triangles. Later on:

---

[2] https://github.com/paddywwoof/pi3d_book/blob/master/model01.blend
[3] https://github.com/paddywwoof/pi3d_book/blob/master/model01.py

```
s 1
f 17/42/15 16/41/16 24/62/17 25/63/18
f 15/46/19 22/49/20 30/64/21 23/65/22
f 30/66/21 22/15/20 17/14/15 25/67/18
```

faces (17,16,24,25) (15,22,30,23) (30,22,17,25) have normals (15,16,17,18) (19,20,21,22) (21,20,15,18) i.e. not a flat face. And you will see that the same vertex used in different faces (i.e. vertex #17 or #25 above) has the same normal vector (#15 or #18)

In the mtl file you will see that there is a **newmtl Material** to match the usemtl line in the obj file, followed by lines specifying material properties (Ns specular exponent, Ka ambient, Kd diffuse, Ks specular (RGB values), d alpha, illum illumination model, map_kd a file to use for diffuse values) Pi3d only picks up the Kd and map_kd to use as material and Texture.

CONSTRUCTING A SHAPE FROM SCRATCH

Hopefully you've got enough idea about the way that vertices, normals, texture coordinates and faces fit together now to have a go at creating something from scratch. This isn't the way that pi3d is normally used - there are plenty of "standard" shapes, as well as the ability to import 3D models, but this exercise should be a valuable aid to understanding what the capabilities are and how to get round any limitations you might find.

The scope is intentionally ambitious: A procedurally generated city with two different building types with variable, realistic surface textures. Obviously I've spent a bit of time thinking about it but this approach certainly isn't the only way, or even the best, but this is the logic I follow in this example:

1. A function to construct a simple house given the position of its four corners. The house will be similar to the Cuboid but it will have a pitched roof (replacing two triangles with six and two extra vertices) and it doesn't need a bottom.

2. A function to construct a simple office block given the four corners. This will be essentially the same structure as for house but the roof will be nearly flat and the texture mapping will be modified so that a tall building will have more floors. This could be done using umult or vmult to make the image repeat up the side of the building, however

by using different parts of an image it will be possible to distinguish the ground floor from upper levels.

3. A function to generate roadside paving given the corners of a block

4. An algorithm that can be given N corners of a polygonal block and partition it into appropriately sized buildings. This algorithm should be able to generate a range of building sizes and types in a controllable way.

5. An algorithm to generate a pattern of city blocks.
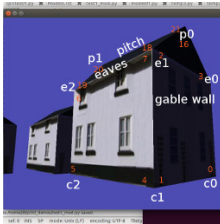
Before getting stuck into **construct01.py** [1] It's worth explaining a couple of things. First the class definition City is very similar to most of the "standard" pi3d shapes defined in the pi3d/shape directory. It inherits from Shape and calls Shape.__init__(). It also creates arrays of vertices, normals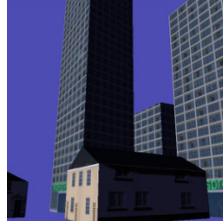, texture coordinates and faces and uses them to create a new Buffer object that is appended to the Shape.buf list. However the creation of the Buffer is done via a different method **City.make_houses()** which is provided with a lists of "specs" each containing four tuples for the coordinates of the corners and a height.

An additional difference from previous examples has been introduced here. Rather than rotating the Shape, the a,d,r,f keys rotate and tilt the Camera object. In addition the w and s keys move the Camera forwards and backwards. The x and z movements can be calculated using simple trigonometry, and this is the approached used in many of the demos in pi3d_demos, however they correspond with the values already worked out for the Camera transformation matrix so it is more processor-friendly to simply re-use the figures as is done here. Work your way though the example and try to convince yourself that you understand how the vertices, normals, texture coordinates and faces are constructed. Remember that the order of vertices in a triangle determines which is the front, there is also a similar (and not

---

[1] https://github.com/paddywwoof/pi3d_book/blob/master/construct01.py

unrelated) quirk when calculating normal directions using the cross product of two edges, namely cross(A,B) points the opposite direction to cross(B,A).

In **construc02.py** [2] most of the functionality of make_houses has been transferred to a private method _make_buildings (denoted by the _ prefix, as mentioned in the introduction, this is purely a convention to make the code easier to read, python ignores it!). This method takes two additional arguments: 1. the height of the eaves compared with the top of the pitch - i.e. how flat the roof will be 2. the whole of the tex_coords array which is quite different for offices, being a long image used for all four sides, rather than the image divided into six used for houses. There is also an additional factor added to the spec for each house or office that determines how much of texture to use in the uv map. If you play around with the *spec* values in the example program you may be able to get a feel for how this manipulation of uv values is being used.

The next illustration **construct03.py** [3] adds the nearly-flat raised areas to represent the sidewalks of each block. The code to generate these shapes is almost a re-run of houses and offices apart from the need to have different polygons to allow triangular and pentagonal blocks, also because the side surfaces are so small, a simplification has been allowed to do without a set of vertices around the edge of curb. This means that the normal direction can't be differentiated at this edge but it's not really big enough to see anyway.

The code to "populate" a block with offices and houses is one of the trickier aspects of this procedural generation exercise. It's not purely graphics related but if you're interested in the process this is the logic:

> For each side of the block calculated how many build-
> ings should be added. Then make a list of points that
> will mark the street facing corners of the buildings. For

---

[2] https://github.com/paddywwoof/pi3d_book/blob/master/construct02.py
[3] https://github.com/paddywwoof/pi3d_book/blob/master/construct03.py

the corner buildings work out the fourth corner by see-ing where two lines running in from the front corners intercept. For the intermediate buildings calculate the re-quired other two (inside) points.
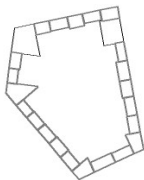
There's a little bit of algebra involved, but it's essentially just stuff you will have learned when you were 13 or 14: simulta-neous equations and the fact that the gra-dient of a line is minus one over the gra-dient of its perpendicular. The demo **con-struct04.py** [4] differs from previous ones in that it is essentially a big function for generating a list of office and house specs from a polygon representing a city block. For this reason there is a if __name__ == "__main__": section at the bottom that will be executed if you run this file but will be ignored when the file is im-ported into something else. The bottom section draws out the gener-ated block and saves it as a file "temp.png" - useful while developing the code and tweaking settings. In the next demo you may notice that there are some deficiencies caused by the allocation of corner build-ings followed by in-filling the other edge buildings: namely there is sometimes an overlap and sometimes a gap. A better approach would be to allocate the buildings from corner to corner and scale the whole lot to fit.

In the final demo **construct05.py** [5] a list of blocks is used to generate the specs. Ideally this would be done using a clever algorithm but that could take a whole book on its own! There is also a simple plane to provide the road surface and "fog" has been added to the city ob-ject. You may remember fog when you looked inside the Shape class, it's held in the unif array and is used by the shader to fade colours of objects in the distance. As the calculation is done in the vertex shader it doesn't work well for large surfaces with few vertices, like the ground plane. If you try adding a similar fog to that you will find that the effect is strongest when you are in the middle - the maximum

---

[4] https://github.com/paddywwoof/pi3d_book/blob/master/construct04.py
[5] https://github.com/paddywwoof/pi3d_book/blob/master/construct05.py

---

distance from all the vertices! One option would be to do the distance calculation in the fragment shader, but that would slow it down, the alternative approach would be to subdivide the ground surface and this is one of the benefits of using the ElevationMap class that will be covered later.
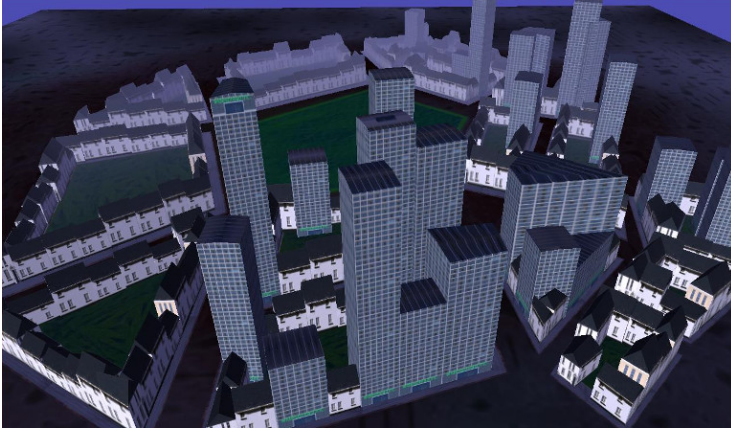
One last new feature has been added to this demo that will be explained more fully in the next chapter - mouse camera control. As you will have found, the keyboard control of the camera is very "clunky" compared with mouse movement. With mouse steering you can fly forwards and backwards using w and s keys. You will also see one of the problems that the ElevationMap helps to prevent - going through the ground surface!

There are various ways that the city could be improved: procedural street layout, more varieties of building, better image textures, more structure to the models, normal maps to give more close-up detail, reflections from windows.. The scope is almost limitless but hopefully you now have some idea how to approach and implement some of the more sophisticated aspects of 3D modelling.

USER INPUT AND ENVIRONMENT UTILITIES

9.1 Keyboard

Keyboard input has been used in all the demo programs without any real explanation so you've probably got a good idea how to use it but it might be worth explaining a little more about how it works.

One aspect of python that people often find strange is the absence of non-blocking key input as a "built-in" function. The reasons is almost certainly because of the desire to make python applications run unaltered on all platforms. In pi3d we have to provide a host of different options for keyboard input: CursesKeyboard, SysKeyboard, x11Keyboard, AndroidKeyboard (which is really just a placeholder) and WindowsKeyboard. Normally the appropriate one will be loaded for the environment you are using but this is a surprisingly non-straightforward area that may need some careful thought especially on non-standard platforms or using non-standard keyboards or even simply trying to use keys that map to variable key-codes.

9.2 Mouse

The Mouse class operates in a similar fashion to the Keyboard to some extent, but the default linux (and Raspberry Pi) mechanism reads events directly from the operating system. The disadvantage of this is that the programs have to be launched with root privileges ($ sudo ...). The advantage is that the mouse movements continue to provide input even when the cursor is outside the relevant window or even "stuck" against a screen edge. This means that it's possible to click on other things on the desktop, including the X to close the pi3d window, if needed! To achieve the same effect with the pygame mouse input (as used with Windows), the cursor is "warped" to the centre of the window each frame. There are slight differences in the behaviour of Mouse functions with the two systems but most of the demos simply use position()

Mouse.position() returns a tuple (x, y) this can be constrained by passing Mouse.__init__() an argument restrict=True (the default), in which case values can also be set for the width and height that the mouse movement is restricted.

Mouse.velocity() returns a tuple (dx, dy) which is either the distance since the last movement or since the last call of position or velocity.

Mouse.button_status() returns either Mouse.LEFT_BUTTON, RIGHT_BUTTON, MIDDLE_BUTTON or BUTTON_UP but varies a little in behaviour from platform to platform.

9.3 Events

The InputEvents class provides a very flexible method for virtally any kind of input, not just mouse and keyboard but also joysticks and game controllers. However it becomes tricky to set up on linux devices with variable input devices such as laptops with touch-pads and doesn't work on Windows at all. In pi3d_demos the Silo.py demo uses this mechanism so checking out that demo would be a good place to start if you need to use this flexibility.

9.4 3D Backgrounds

In pi3d the background scenery is provided by either the Environ-
mentCube or EnvironmentSphere classes. These are basically cubes
or sphere where the triangles are defined so that they face inwards (re-
member the order of vertices determines which way a surface faces in
OpenGL). However the Texture used to wrap onto the inside of the
shape has to conform to the perspective required for not distorting the
scene or making the seams show.

The EnvironmentSphere is simplest in this regard, using an equirect-
angular projection in which top to bottom of the image is 180 degrees
and left to right of the image is 360 degrees. If you have a modern
phone you can probably get an app to take your own "photosphere"
pictures, otherwise you can download them from various places on
the internet. see [1]

The projection for the EnvironmentCube is trickier to produce without
specialist software. There is a facility in blender to do the conversion
and there are some instructions on the FAQ [2]

Play around with the two demo programs mentioned above, trying the
different cube layouts. Look at the image files used for each version.
Notice also that the EnvironmentSphere demo uses the same image
for the reflection as the inside of the sphere and that the orientation of
the reflection correctly matches the environment - i.e. mirror image.

---

[1] sphere http://pi3d.github.io/html/FAQ.html#sphere
[2] cube http://pi3d.github.io/html/FAQ.html#cube

It is important, when using a cube or sphere to represent the distant background, to move it around as the camera moves. You will see that most of the demos do this but not the BuckfastAbbey one. If you run that demo and head out towards one of the corners you will be able to see what happens if a) the opposite corner starts to move beyond the far plane b) the camera moves outside the box.

9.5 ElevationMap

At the end of the last chapter I touched on some of the problems caused by representing the ground as a large horizontal quadrilateral (pi3d.Sprite). The ElevationMap class is used in quite a few of the demos to solve these issues, open the ForestWalk.py demo and have a look at the relevant code. There are various things worth noting:

1. The number of "facets" that the ground is divided into is defined by the constructor arguments divx and divy. The elevation is defined by a combination of height argument and a grayscale image, white pixels being taken as full height and black pixels as zero height. Notice that the number of vertices needed when dividing a grid into 32x32 is 33x33 so if you need exact heights to be read from the image you need to make the image one more pixel than divx and divy and you need to use a lossless compression format such as png. In the ForestWalk.py demo the surface has been made tileable by making the left and right,

and top and bottom, pixels identical - see the mechanism for "tiling" in lines 125-134 and 183-185.

2. There are several useful methods in ElevationMap including calcHeight() used here, but also clashTest() that checks if a sphere would intersect with the surface and return the normal vector at that point (see the Pong.py demo) and pitch_roll() that returns the X and Z rotations to apply to a shape in order for it to lie parallel with the surface at a given point (see the Tiger-Tank.py demo)
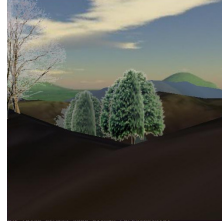
LINES, POINTS AND MERGING

In the chapter introducing 3D graphics I described the process of setting up the "attribute array" of vertex information and the "element array" essentially specifying how the vertices define the triangular faces. I then passed quickly over the details of how the OpenGL function glDrawElements() is called to draw the shape. In actual fact this function needs to be told what kind of elements to draw, not just GL_TRIANGLES but also GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN work, each of which interpret the element array differently.

In pi3d.Shape there are two methods that allow switching between triangles (the default) and points or lines.

10.1 set_line_width and set_point_size

Have a look at the source code of pi3d/Shape.py around line 420 and you will see that it is a relatively simple process of calling the OpenGL function glLineWidth() and changing the Buffer property `draw_method`. If the argument `closed` is set to True then the line will loop from the last point back to the first, If the width is set to zero then the draw_method will revert to triangles. The set_point_size method above it is slightly simpler as there is no OpenGL function to

set a global point size, rather the size is passed to the vertex shader as a uniform variable Buffer.unib[8]. This is then scaled with distance so that a point is the size specified at one unit distance from the camera, half at two units, third at three units etc. (Have a look at any of the standard vertex shaders to see how this is done). Lines are always the same width however near or far away.

Now open **linepoint01.py** [1] and try toggling between triangles, lines and points using the t, l and p keys. Check out the effect of using a flat shader or lighting dependent shader and the related effect of using uv texture mapped shader. Try the pi3d.Lines object; one thing you will find with this is that only the mat_flat shader works as expected. This is because the Buffer attribute array doesn't have normals or texture coordinates. There is also a pi3d.Points class which is very similar.
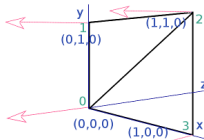
You might also notice with many of the standard shapes drawn as lines with the "strip" argument to set_line_width set True, that though the majority of the lines define quadrilaterals there are some quads that have an additional diagonal line. In fact, it might seem logical for all the diagonals to show, after all, that's how the triangles are represented when the 3D shapes are drawn as solid objects. However if the order of vertices used in the element buffer is defined carefully the line drawing can be made to (mainly) follow the edges. In the diagram, for instance, the triangles are defined as `[(0, 1, 2), (2, 3, 0)]` which when used to define a line is interpetted as 0=>1 1=>2 2=>2 2=>3 3=>0 i.e. going round the outside. In earlier versions of pi3d the triangles may have been defined as `[(0, 1, 2), (0, 2, 3)]` which would have produced a line with the diagonal drawn twice and the edge from 3=>0 missing!

In the above example the line drawing is what would happen using GL_LINE_STRIP which is the default behaviour of set_line_width. If the argument "strip=False" is used then the draw method is GL_LINES and this interprets the elements as a series of pairs defining each end of a series of lines. The result would then be 0=>1 2=>2

---

[1] https://github.com/paddywwoof/pi3d_book/blob/master/linepoint01.py

3=>0 i.e. only the first and last edge would be drawn, which you will have seen when you tried the experiments suggested in linepoint01.py. Generally the edges will be drawn more completely using the default "strip=True" however for standard shapes created with Shape.lathe() (i.e. most of them!) the faces fit together to fill in the gaps. Also for models loaded from obj files there is a tendency for extra edges to appear joining the scattered faces and these often look better using "strip=False".

It might occasionally be nice to be able to draw a 3D object as a solid, with directional lighting then overlay lines defining the edges of its faces. However there are two problems with this: 1. As we have seen in the linepoint01.py example we would need two shaders to do it nicely 2. The lines will be exactly the same distance from the camera as the edges of the faces they outline so there is likely to be z-fighting [2] or partial obscuring of lines as the object rotates.

There are at least three ways of tackling this problem. One might be to make a Shape with two Buffers, the first using GL_TRIANGLES with mat_light shader, the second Buffer being a duplicate of the first but using GL_LINES and mat_flat shader. Ideally the second Buffer would be scaled up slightly so the lines were always drawn outside the solid version. As scaling can be more easily done for a whole Shape it might be easier to achieve the same result by using two Shapes, rather than Buffers with the second Shape a child of the first one. (Look at the TigerTank.py demo to see how child objects can be used to conveniently join Shapes together but allow them to be moved relative to each other). In **linepoint02.py** [3] I use a technique of drawing the object twice each frame, once as a solid then as a wireframe. I use the Camera transformation matrix to move the shape slightly towards the view point between each draw. [4]

---
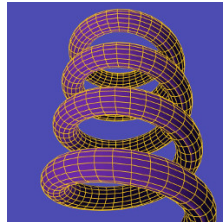
[2] https://en.wikipedia.org/wiki/Z-fighting
[3] https://github.com/paddywwoof/pi3d_book/blob/master/linepoint02.py
[4] In this case the camera isn't moving so a simple translateZ() by a fixed small amount would have sufficed, however the technique used in the example can be used

The third example in this chapter **linepoint03.py** [5] introduces the
pi3d.MergeShape class. This class can be used to combine several
Shape objects into one with the advantage that only one draw() call
will then be needed. For a large number of objects, such as the
trees in the ForestWalk.py demo, this will make a very significant
reduction in the (slow) python side matrix multiplication. Merge-
Shape has two methods: radialCopy() which can be used to pro-
duce regular patterns (see pi3d_demos/Blur.py) and cluster() which
is used to "scatter" objects randomly onto an ElevationMap, (see
pi3d_demos/ForestWalk.py).

10.2 Points using texture mapping

Finally, in the chapter on 'Cameras, 2D projection and Sprites' I
referred to an alternative sprite drawing method using points, for
fast rendering when the numbers get big. Open up the demo
pi3d_demos/SpriteBalls.py There are several features of this demo
that use numpy to do the bounce and movement calculations on large
arrays of vertices, I won't attempt to explain any of that here but I
do recommend trying to get to grips with this in the longer term. For
the moment it is sufficient to understand that the sprites are repre-
sented by the vertices of a Points Shape and that the z location is
being used to represent the size of each point (see lines 46 to 51).
Each frame the vertices move according to "physics" and the loca-
tions are revised using the Buffer.re_init() method (line 73) and the
points are drawn using a special shader. Now look at the shaders
pi3d_demos/shaders/uv_sprite.* (.vs and .fs)

In the vertex shader you will see that it simply sets gl_Position in
the normal way using the projection matrix (as neither the Shape nor
the Camera move, this matrix multiplication could have been elimi-
nated by using a different basis for the vertex coordinates). And the
gl_PointSize is set to be inversely proportional to z depth. In the frag-
ment shader a Texture2D lookup is performed on the texture loaded in

more generally.

[5] https://github.com/paddywwoof/pi3d_book/blob/master/linepoint03.py

SpriteBalls.py, however rather than using a 2D vector calcualted from the texture coordinates (which are not passed to the shader when using the Points class anyway), it uses gl_PointCoord, a variable available in the fragment shader while drawing points.

ELEVEN

STRINGS

pi3d has two mechanisms for producing Text:

The first one is the pi3d.String class which inherits from pi3d.Shape in the same way as the "standard" shapes such as Cuboid, Sphere, Sprite, Torus etc. However as it is initialised it constructs its vertices and texture coordinates to map rectangles from a special Texture - pi3d.Font or pi3d.Pngfont. Each of these classes contain an image of all the letters required and a dictionary to look up the location, width and height of each letter. pi3d.Font is generally more useful as it is generated "on the fly" from a TrueType font but it requires certain functionality from the Python Imaging Library, and where that might not be available (I couldn't get it to work on Android), pi3d.Pngfont can be used (but it's restricted to the letters on the special png image files)

The second mechanism is the pi3d.FixedString class which inherits from pi3d.Texture, a bit like the pi3d.Font class above but rather than containing the full alphabet it uses a string passed to the constructor. This means that the string can then be drawn using only four vertices and two triangles rather than that number for every letter required by pi3d.String, so it can be quite a bit faster on some platforms if there's lots of

text. Another of advantage of FixedString is that PIL [1] and numpy can be used to provide filters: BUMP will generate a normal map, EMBOSS, CONTOUR, BLUR and SMOOTH. The FixedString class can be used wherever a pi3d.Texture would be used, but it also contains an instance of pi3d.Sprite (called pi3d.FixedString.sprite) which has width, height, umult and vmult to match the dimensions of the string to the Texture. (Look back at the chapter `Shapes, Buffers and Display` if you don't remember umult and vmult)

However for varying text (such as a score, or timer) the overhead of creating a new PIL image and converting it to a Texture every frame would be large and in this case it is better to use the pi3d.String.quick_change() method. This has various restrictions as described in the documentation, the most significant of which is that the string can't be changed to a longer length than it started (of course it should be padded by spaces initially for this eventuality).

Have a play with the three examples **strings01.py**, **strings02.py** and **strings03.py** [2] There are one or two quirks to watch out for: if you use a 2D Camera (orthographic - argument is_3d=False) with a pi3d.String object, then you also need to set the argument in the String constructor to is_3d=False. Also the font passed to pi3d.String constructor is a pi3d.Font object but the font passed to pi3d.FixedString is the path to a TrueType font as a string. One odd.

With the rotating text you will have noticed the strange effect of the mipmaps which are a series of lower resolution images that automatically get swapped depending on the z depth of the fragment being rendered. The mipmaps are generated at the time of Texture creation and are generally a good thing so the default argument to Texture is mipmap=True.

You will find strings used in quite a few of the pi3d_demos so have a look at those as well.

---

[1] Python Image Library - generally Pillow https://pypi.python.org/pypi/Pillow
[2] https://github.com/paddywwoof/pi3d_book/blob/master/strings01.py etc

OFF-SCREEN TEXTURES (AND OTHER MORE COMPLICATED THINGS)

The normal result of drawing objects with pi3d is for them to appear on the display. However there can be situations where it might be useful to capture the output and do other processing on it before posting it to the screen. Reasons include: blurring, distorting, edge detection and any number of artistic post-processing effect but also collision detection, shadow casting and stereo imaging.

In pi3d there is a class OffScreenTexture that inherits from Texture. It isn't really intended to be used directly but other classes inherit from it: Clashtest, Defocus, PostProcess, ShadowCaster and StereoCam. Each of these has a similar outline work flow each frame.

1. Start the off-screen capture

2. Draw the objects in the scene, possibly with a special shader as with Clashtest or ShadowCaster but otherwise just normally i.e. PostProcess or StereoCam

3. Stop the off-screen capture

4. Process the image. Sometimes using a special shader, sometimes drawing the off-screen texture to a Sprite with a normal uv_flat shader.

Because offscreen textures can be used for a wide range of reasons the details of each one will vary greatly. However the Post-processing

application is probably the most general this is a good one to look at.

Open pi3d_demos/Post.py and, after running it to see what it does, try commenting out the lines 67, 73 and 78. This will basically cut out the capturing to off-screen texture and subsequent post processing so you can see what is being captured. For the moment ignore the fancy swirling texture on the Spheres, this is a separate complication that I will explain later. If you look inside the pi3d/util/PostProcess.py file you will see that although it can be called with all its arguments defaulting to values, in this instance we are passing a Camera instance and setting the divide argument. The reason for this is explained in the docstrings: there is a facility to only capture part of the screen to generate a lower resolution off screen texture which speeds up the whole rendering process quite a bit. In order to effect this low resolution capture the camera has to be defined with a wider field of view which is done on line 28 of pi3d_demos/Post.py (`scale` is a tidier alternative to defining a whole lens spec with larger fov) And the camera is passed to the PostProcess constructor in line 35 along with the same scaling factor, however you will notice that the camera instance is set to self.viewcam in line 51 which is then not used! The same camera is specified for myshape and mysprite on lines 42 and 48. The reason for all this camera specification is the default instance behaviour of pi3d - which will make the default camera from the first one to be created and, as a 2D camera is created in the __init__() function of PostProcess, care has to be taken to ensure that this doesn't become the default instance by accident.

In PostProcess line 72 you can see the OffScreenTexture._start() method call and some code to just render part of the screen using the glScissor function. On line 85 OffScreenTexture._end() stops the screen capture and draw() renders a simple subdivided quad self.sprite using self.shader, self.tex_list and self.camera. On line 63 you will see that self.tex_list[0] points to the PostProcess instance itself which inherits the behaviour of pi3d.Texture via pi3d.OffScreenTexture.

There are a couple of things that make this even harder to follow: 1. on line 64 and 65 there is a facility to add additional textures (such as bump and relfection) for use by the shader, 2. on lines 99 to 101 there is a facility to modify the `unif` array of self.sprite. The pi3d_demos/Post.py example doesn't use any additional textures (although some of the shaders in pi3d_demos/FilterDemo.py do) but on line 78 of pi3d_demos/Post.py you will see that post.draw() is passed a value for unif[48] that very slowly increases from 2.0 to 6.999, after which it resets to 2.0.

Now if you look in pi3d/shader/post_base.fs - the fragment shader - you will see on line 27 that use seems to be made of unif[16][0] (remember that the flat c_types.float(60) array in python becomes vec3[20] in GLSL so unif[48] in python is unif[16][0] in the shader.) But what exactly is it doing? Well the vertex shader is very simple, essentially just setting the vertex location in gl_Position and flipping the image top to bottom as it sets the uniform variable texcoordout. In the fragment shader lines 26 to 29 loop nine times to increment the eventual pixel RGBA value texc. Each loop looks up the value from the PostProcess texture using Texture2D with a slightly offset coordinate dx[] and dy[] and a weighting factor f[]. unif[16][0] is used as a multiplier for the dx[] and dy[] values in order to sample the "convolution" [1] over a wider area. If you watch the demo for long enough you will see the edges gradually get wider then suddenly jump back when the value in unif[48] wraps back to 2.0.

12.1 Other Shaders

The "star" shader used to texture the Spheres in pi3d_demos/Post.py above is another example of how you can use the GPU to do all kinds of fancy things. Look at the source code, which was contributed by Peter Hess based on www.iquilezles.org shadertoy demos. The shader works by converting the texture coordinates to a polar basis in lines 25 and 26, then applying factors that depend on an incrementing

---

[1] https://en.wikipedia.org/wiki/Kernel_(image_processing)

---

value "time" and trigonometric transformations then using the values to lookup and modify the RGBA values from the texture sampler.

It's quite fun to experiment with different formulas and values in shaders but, if you do, you will probably have to put your shaders in subdirectory of your working directory (as with pi3d_demos/shaders) and you will probably have to "expand" the `#include ...` syntax used in the main pi3d shaders as the process of figuring out the path to import from might defeat the Shader loader!