



OPENPYFLYSIM

A Flight Simulator written in Python and OpenGL

Adrian Draber

22018721 Woodhouse College

Table of Contents

<u>1 Analysis</u>	2
1.1 Introduction	2
1.2 Existing products	3
YSFLIGHT	3
FlightGear	7
Conclusions	10
1.3 End user research	10
1.4 Objectives	12
1.5 Modules	14
1.6 Implementation Research	16
Transformations	16
Culling	18
Camera System	19
1.7 Prototyping	21
Prototype 1	22
Prototype 2	23
Prototype 3	25
<u>2 Documented Design</u>	28
2.1 Swimlane diagram	28
<u>3 Implementation</u>	32
<u>4 Technical Solution</u>	32
<u>5 Testing</u>	32
<u>6 Evaluation</u>	32
<u>7 References</u>	33
<u>8 Appendix</u>	33
Prototype 1	33
Prototype 2	36
Prototype 3	40

1 Analysis

1.1 Introduction

Flight simulators are a genre of program where the user assumes control over an aircraft, such as a plane. Usually, they strive for realism, particularly because they are used to train pilots and other jobs involved in the air industry such as air traffic controllers. From a technical perspective, they present an interesting challenge due to the requirement to render three-dimensional environments and replicate the physics of real aircraft.

In this project, I aim to build a virtual flight simulator from the ground up, focusing on achieving accurate rendering of three-dimensional environments as well as replicating a very basic system of aircraft controls. The project should primarily aim to demonstrate the ability to use mathematics to project environments onto a two-dimensional screen and do so in an effective and optimised manner. Additionally, the program should also function as a learning tool for – primarily amateur or recreational – pilots, and thus possess systems such as measuring the G-force on the cockpit, or the ability to take off/land. These are important as making mistakes while flying a real

plane could be potentially dangerous, so it could assist in helping amateurs recognise dangerous manoeuvres.

1.2 Existing products

The following is an analysis of some existing flight simulators:

YSFLIGHT

YSFLIGHT^[1] is a Flight Simulator written in C++ using OpenGL by Soji Yamakawa, with its first release in 1999, and it has received regular updates ever since.

According to Yamakawa, the program was written for the purposes of “*(1) writing my own flight simulator, and (2) writing a software used by hundreds of thousands of people over the world. I am always so happy to receive encouraging emails about YSFLIGHT.*”. Initially created as a school project, it was turned into software to help people learn to understand aircraft physics and mechanics: “*Microsoft Flight Simulator is a great piece of work, but I also believe it is nice to have a flight simulator that everyone can casually play during the lunch break. That has been the concept of YSFLIGHT. But, I put many elements that I learned from my flight training in YSFLIGHT. I do use YSFLIGHT for practicing IFR approaches in a Cessna for myself (of course I'm not logging time for it though.) I hope YSFLIGHT serves you well for the future!*”^[2]

Booting up YFLIGHT, we are treated to this menu where we choose a starting location and model of plane, among other factors:



Figure 1: The screen before starting a flight.

YFLIGHT is primarily designed for joystick control; however, the mouse and keyboard can be used to simulate joystick input. Upon entering the cockpit, several features of the control scheme stand out:



Figure 2: After starting; plane is on the runway.

- There is a virtual joystick on screen, seen on the left of the cockpit in the image. This allows the pilot to always see the status of the joystick, which is useful for mouse controls.
- There are physical flaps and spoiler indicators (bottom right of the UI). These are the controls used on real planes and are responsible for turning motions. Having physical bars showing the orientation of each helps with visibility of the controls substantially.
- There is a virtual plane on the left side of the UI, showing its orientation relative to the ground - this is useful because YS Flight primarily employs a camera perspective from inside the cockpit.
- However, one drawback is that controls aren't easily explained, and you can't access the control menu from inside the simulator.

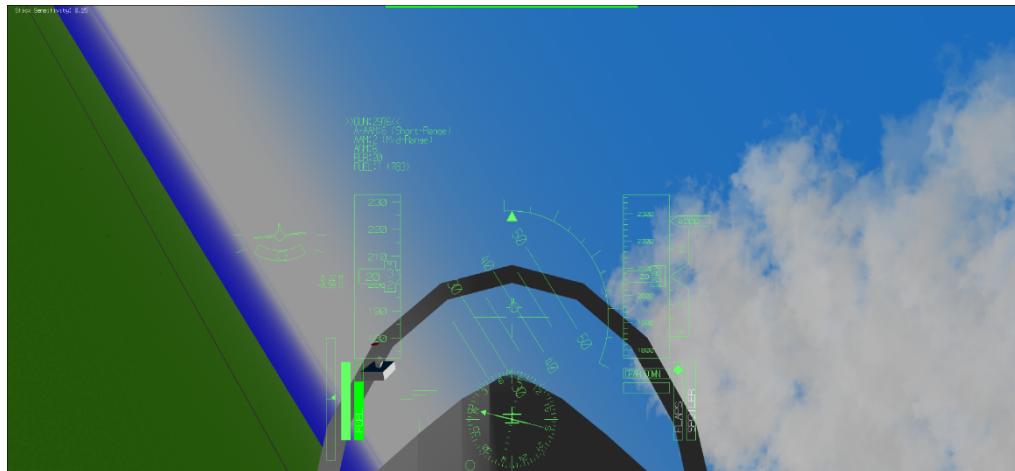


Figure 3: The screen while in flight

The simulator also contains several pre-built scenarios, such as trying to land a plane at a recreation of London Heathrow. After playing around with the flight controls for an hour, these are some interesting features I noticed:

- As noted in the Introduction, it is important for flight simulators to accurately convey dangerous manoeuvres to teach aspiring pilots in an environment where no risks are posed regarding their safety. YS Flight achieves this by colouring the entire screen red or black when a high G-force is reached, which functions as an easy-to-read indicator of pilot danger.
- Sideways motion of the joystick controls roll (sideways rotation of the plane) while up/down motion controls pitch (up-down rotation). Yaw, the rotation that corresponds to the direction on the map the plane is facing, is controlled with the ZXC keys (which physically control the “rudder” at the back of the plane).
- Pressing the “M” key reveals a second camera view, behind the cockpit. This is useful in the case of YS Flight since it can be used as a military dogfight simulator in addition to its main use as an amateur flight sim.
- In addition to visual indicators, various bleeps and other noises are used to communicate information to the pilot. These serve as easily recognisable warnings that recommend pilots to look at their gauges - for instance, they might indicate that fuel is not flowing to the engine during a turning manoeuvre due to the forces involved.

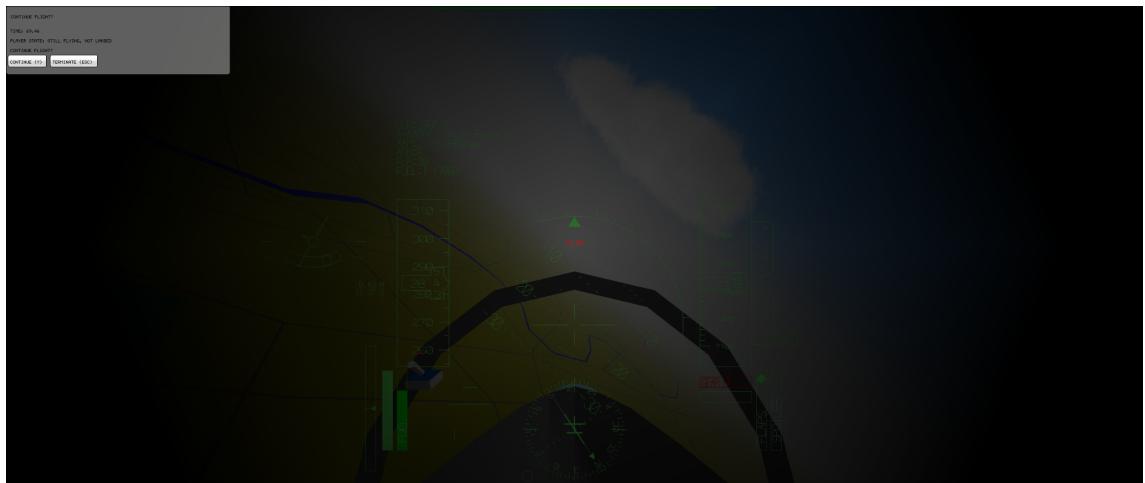


Figure 4: The screen turns black as I hit 12.4G during a manoeuvre. In real life, this amount of force would kill the pilot.

FlightGear

FlightGear is another free, open source flight simulator. It is written in C/C++ and renders itself using OpenGL. It was produced by Dave Murr as a response to flight simulators such as Microsoft Flight Simulator, which were closed source. FlightGear could be distributed to many users for free, and its open source nature meant that users could modify its code to add features they desperately wanted. [3]

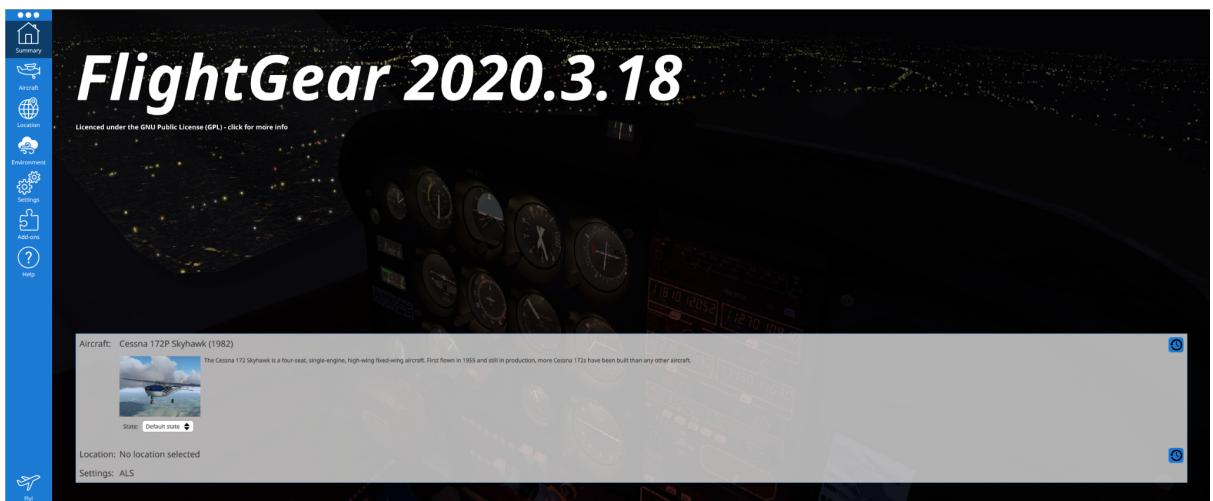


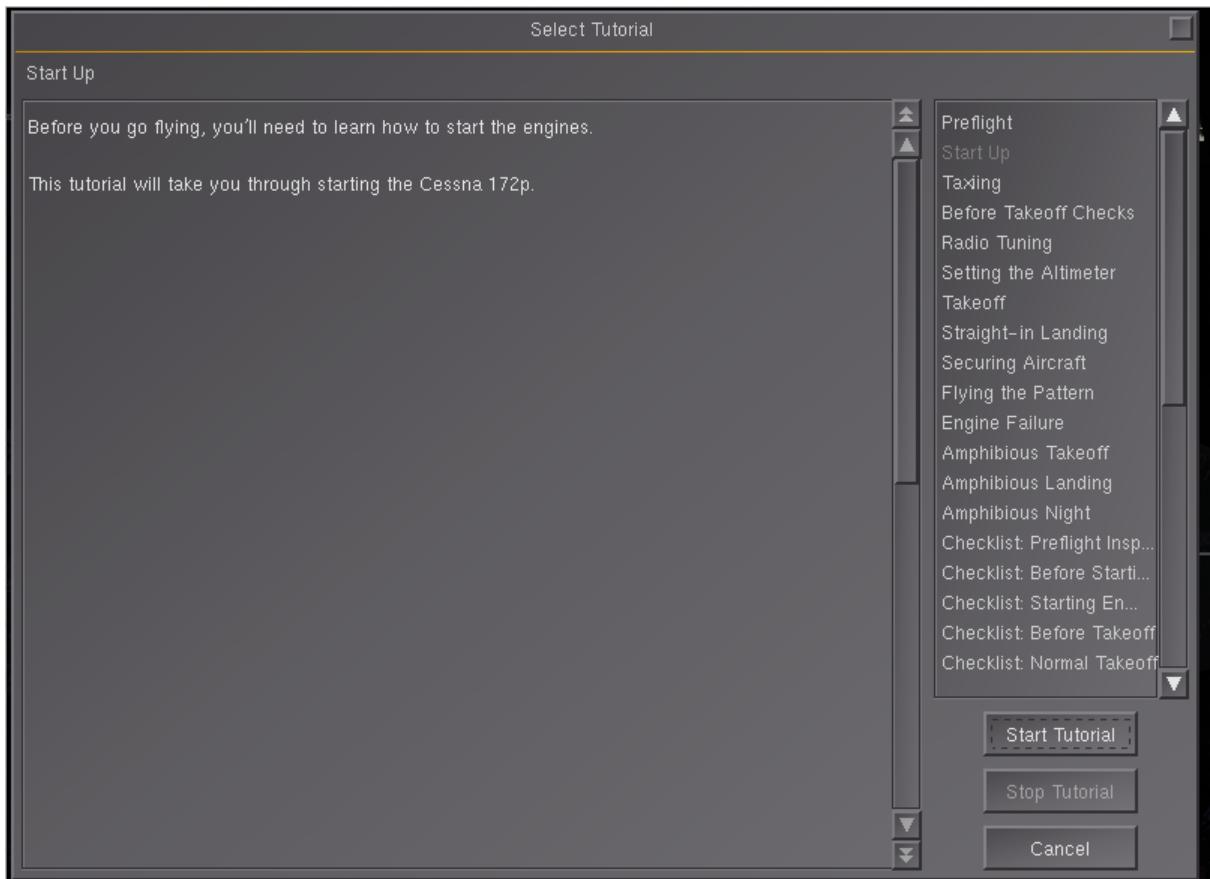
Figure 5: The main menu of FlightGear

The main menu and UI of FlightGear was much more intuitive than that of YSFlight. On the other hand, the controls were at first more unintuitive - for instance, the arrow keys control the opposite surface relative to the direction in which they are located (down arrow key raises elevator up, etc). It is clear that, where YSFlight was built for a joystick, FlightGear was built for more realistic controls regarding a keyboard. Between the two, I came to prefer FlightGear's control scheme as it felt more intuitive, and will be what I base my controls on in my NEA.



Figure 6: The cockpit in FlightGear

FlightGear also had an excellent set of in-depth tutorials teaching every topic needed to understand how to fly a plane. Dozens of these tutorials exist, each narrated by a computerised voice, and I found many of them helpful and inspiring regarding how my controls should work. Implementing a tutorial would be useful to my goals considering my project's audience, but I doubt I would be able to implement such an ambitious feature within the scope of my project.



Figures 7 and 8: The selection screen for tutorials, and me learning how to take off my plane from the runway.

Another thing I found impressive was the weather system, as well as the effective use of sound effects (like in YSFlight) to communicate

various events, such as the tail of my plane hitting the runway as I tried to take off.

Conclusions

- There exists a demographic for people uninterested in the offerings of big corporations like Microsoft due to their proprietary nature, and people who look for both more casual flight simulators, and those who can learn how to fly from them.
- It is important to consider the physical surfaces (rudder, ailerons etc) when it comes to controlling a plane
- Good sound effects and UI can go a long way in helping the user understand what is happening
- Plane controls can be quite unintuitive because of the mechanics regarding three-dimensional control.

1.3 End user research

As previously identified, the target audience for such a program would primarily be hobbyist/amateur pilots, particularly those who can't practise by flying a real plane. In order to understand the demands of this community, which will be used to inform my objectives, I contacted various flight enthusiast communities and asked them what they enjoyed about their particular flight simulator. I was able to get 12 responses, which I categorised into five broad categories:

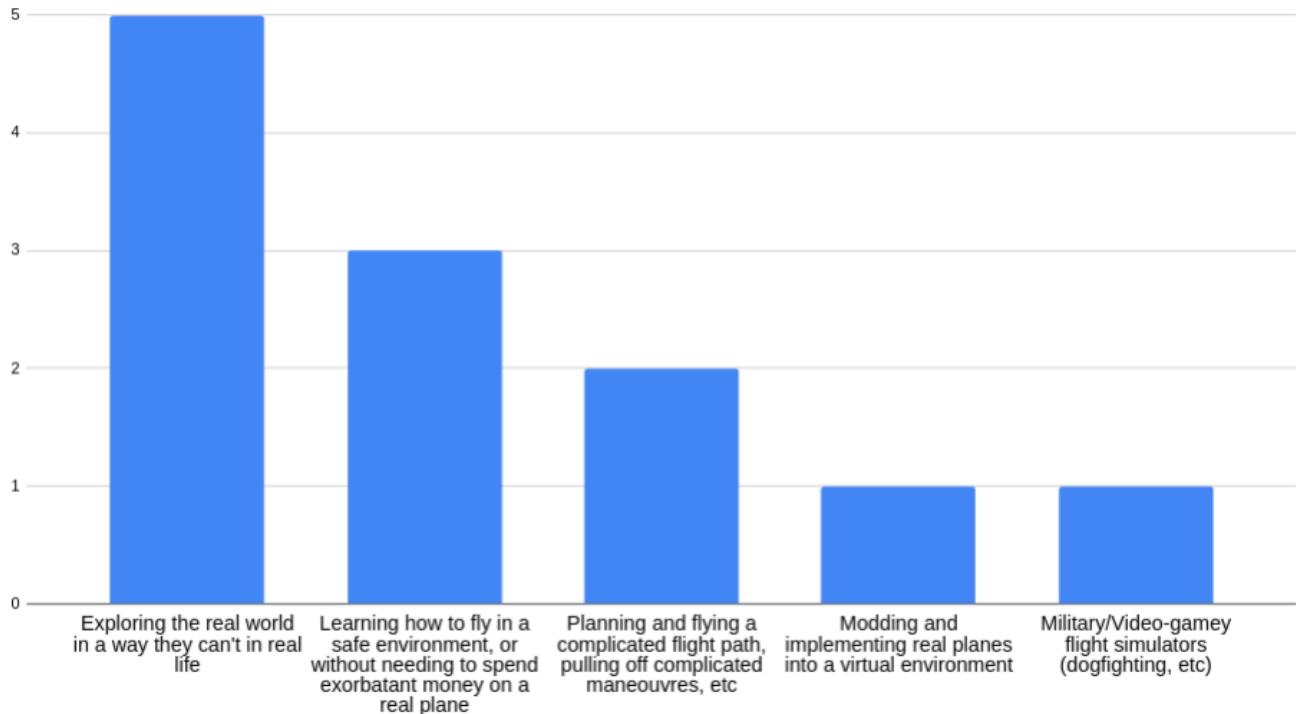


Figure 9: The 5 categories of the 12 responses I got by asking people what they enjoyed about <x> flight simulator. Respondents are kept anonymous.

It is important to emphasise that many different subcommunities exist within the “flight simulator” genre. Some people prefer realistic, expensive flight simulators, which allows them to feel like a pilot and explore the real world. As seen in the survey, this was the most representative group. This makes sense as I collected part of this data in a community dedicated to Microsoft Flight Simulator. There is also a large community however dedicated to modding these simulators and implementing their own hyper realistic plane models. There also exist many video games based around aeroplanes which mostly use air combat and dogfighting as a source of appeal. These include War Thunder and Ace Combat.

From my survey however, it appears that exploration and the idea that the entire world is at one's fingertips is one of the most prevalent reasons for picking up flight simulation as a hobby, particularly as a hobby. For instance, one user told me:

"[It] lets me explore different places without actually going there. It's also got good graphics and attention to detail which makes it a chill way to virtually travel and take in the sights".

Taking this into account, since this program is meant to follow a similar philosophy as Soji Yamakawa's YSFLIGHT, I will focus primarily on the first two appeals - the ability to virtually explore the world, and the ability to be a pilot without heavy investment in flight school and aircraft. It also appears that customisation is also an interest of the community, so I will add the ability to import custom maps. However, some features - such as air combat or a fully detailed and realistic map of the real world - will need to be discarded as they are far beyond the scope of this project.

1.4 Objectives

The investigation asks for the following requirements to be satisfied:

O1: Load in a world from a map. This can be a real map of a real environment, or fictional.

(1) Use a .bmp file or similar to store a heightmap of some terrain. The brightness values on this heightmap will correspond to the actual in-program height of the terrain at that location. This will be done by a module which reads the brightness values and assigns them to a matrix that determines the y value of the vertices on the terrain matrix.

(2) Allow for import of custom maps.

O2: Render the terrain on-screen according to the location of the player's camera, and the direction in which it is pointing.

(1) Orient the camera in the correct direction, and update its position according to velocity.

(2) At the start of the program, define all vertices in the world map, complete with colour, elevation, etc

(3) Determine the vertices within the player's current viewport and their distance. Then, only render the vertices within the render distance. Make use of trigonometry to also make sure that vertices behind the camera aren't rendered either.

(4) Define a water level. Any vertices below this level will be underwater, so draw a water plane.

(5) Use back-face culling technique so that only triangles facing the camera are being rendered

(6) Draw 2D triangles between the on-screen vertices.

(7) Update this process for every frame.

O3: Render other props

(1) Render the plane model.

(2) Render terrain details - trees, etc

O4: Allow player input, such as the ability to shift the rudder and
ailerons in order to perform turning manoeuvres.

(1) Register user input from the keyboard and mouse.

(2) Update velocity and angles as determined by input, using
various trigonometric functions.

(3) Check to see if the player is colliding with terrain.

O5: Make use of JIT to speed up the program's performance.

O6: Implement taking off and landing as mechanisms.

(1) Implement landing and take-off strips on the map

(2) Allow flight only after a certain velocity has been reached on
the ground, to mirror real life physics.

O7: Implement a title screen and UI (similar to the one in YSFlight) to
allow control of initial conditions.

1.5 Modules

We will make use of the following Python modules:

- OpenGL^[4] – OpenGL is a graphics API for rendering vector
graphics. This will be the primary method we will use to draw what
is going on screen. Additionally, to extend OpenGL's functionality,

we will utilise the OpenGL Utility Library (GLU) and OpenGL Utility Toolkit (GLUT)^[5]. In Python, this will require installation of the pyOpenGL and pyOpenGL-accelerate modules, as well as of a GLUT library such as freeGLUT to the PC/Virtual Environment.

The justification for choosing this library is its ease of use and support with most PC systems, which is what led to it being considered the “standard” library for amateur graphics projects.

- NumPy^[6] – This is a library that extends Python’s mathematical features. For instance, it allows us to use arrays, whereas base python only allows for linked lists – this is useful because arrays are significantly more efficient than linked lists. Given the nature of graphics needing to be updated many times per second, it is crucial that we write an efficient program.
- PyGame^[7] – We will use PyGame in order to handle both our display and input of controls. While PyGame provides many other uses, such as drawing content directly onto the screen, this will be handled by OpenGL in our application. Additionally, the pygame.joystick module of pygame allows for reading joystick controls, such as from a gamepad, which is considered the “default” controller for flight sims and is preferred over mouse and keyboard (as seen in the “existing products” section).
- Numba^[8] - Numba is a Just-In-Time compiler for Python - while a script is being interpreted, it will compile any function given the

@njit decorator, allowing it to run faster. The downside is that a compiled function cannot make use of foreign modules or refer to other functions in the program. Despite this, including this module to compile some of our python functions may lead to great improvements in performance.

1.6 Implementation Research

Based on my research of existing flight simulators, I was able to create a list of objectives. In order to check their feasibility, I created a series of prototypes regarding their implementation. I first performed research into how many of my intended features will be implemented.

Transformations

OpenGL functions primarily on the principle of vertices, which have lines, triangles and squares (“quads”) drawn between them. We can move these vertices via the use of matrix transformations:

$$\begin{bmatrix} \text{Transform_XAxis.x} & \text{Transform_YAxis.x} & \text{Transform_ZAxis.x} & \text{Translation.x} \\ \text{Transform_XAxis.y} & \text{Transform_YAxis.y} & \text{Transform_ZAxis.y} & \text{Translation.y} \\ \text{Transform_XAxis.z} & \text{Transform_YAxis.z} & \text{Transform_ZAxis.z} & \text{Translation.z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 10: The transformation matrix in OpenGL.^[9]

Here, we have a 3x3 matrix representing the points themselves, as well as a fourth row and column that allows us to perform translations. Some examples of specific uses to achieve certain matrix transformations:

$$\begin{bmatrix}
 1 & 0 & 0 & \text{Translation.x} \\
 0 & 1 & 0 & \text{Translation.y} \\
 0 & 0 & 1 & \text{Translation.z} \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

$$\begin{bmatrix}
 \text{Scale.x} & 0 & 0 & 0 \\
 0 & \text{Scale.y} & 0 & 0 \\
 0 & 0 & \text{Scale.z} & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

$$\begin{bmatrix}
 1 & 0 & 0 & 0 \\
 0 & \cos(\theta) & -\sin(\theta) & 0 \\
 0 & \sin(\theta) & \cos(\theta) & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

Figure 11: Example of a translation; scaling; and rotation around the x axis.^[9]

One thing to note is that matrices are non-commutative. This means that the order of operations does matter. In particular, matrices are always modified by operations behind them. The order of operations of a matrix is hence often “reversed”. We can also merge matrices into a single transformation:

$$\begin{bmatrix}
 1 & 0 & 0 & 1.5 \\
 0 & 1 & 0 & 1.0 \\
 0 & 0 & 1 & 1.5 \\
 0 & 0 & 0 & 1
 \end{bmatrix} \times \begin{bmatrix}
 1 & 0 & 0 \\
 0 & \cos(180) & -\sin(180) \\
 0 & \sin(180) & \cos(180) \\
 0 & 0 & 1
 \end{bmatrix} \times \begin{bmatrix}
 \cos(90) & 0 & \sin(90) & 0 \\
 0 & 1 & 0 & 0 \\
 -\sin(90) & 0 & \cos(90) & 0 \\
 0 & 0 & 0 & 1
 \end{bmatrix} = \begin{bmatrix}
 0 & 0 & 1 & 1.5 \\
 0 & -1 & 0 & 1.0 \\
 1 & 0 & 0 & 1.5 \\
 0 & 0 & 0 & 1
 \end{bmatrix}$$

Figure 12: If performed separately, we'd consider the rotation around y to be performed first, followed by the rotation around x and finally the translation. Here we can also simplify this to a single transformation.^[9]

While OpenGL does not require us to manually input matrices, it does still follow the rules of matrix mathematics – for instance, in OpenGL, transformations will appear to be performed in the “reversed” order compared to their definition in the code. Additionally, since matrix transforms are applied to all vertices, we need to make use of a stack so that only specified sets of vertices are available at any time to be

transformed – else, every time we perform any sort of transform, everything in the world is transformed.

Culling

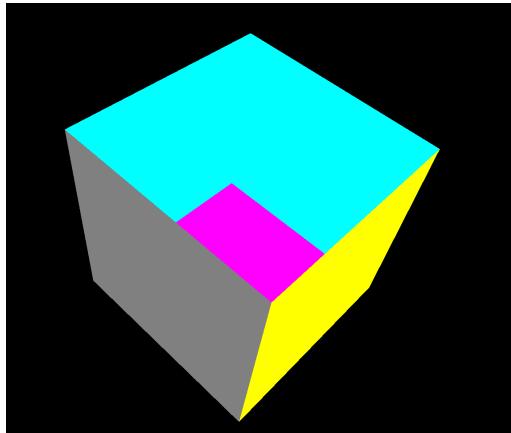


Figure 13: An image from one of my prototypes. No culling algorithm is being used; we can see that the magenta face clips through the cube, since it is drawn after the cyan face (but before the grey and yellow faces).

The culling method used by OpenGL involves discarding faces that are “facing away” from the viewport, in a process called Face Culling. This is achieved via winding order. When triangles are drawn, the vertices are defined in an order. The order of the vertices is used to define whether a triangle’s vertices are defined as counterclockwise or clockwise. By default, any triangle with counterclockwise vertices is considered to be seen from the front, while triangles with clockwise vertices are seen from the back.

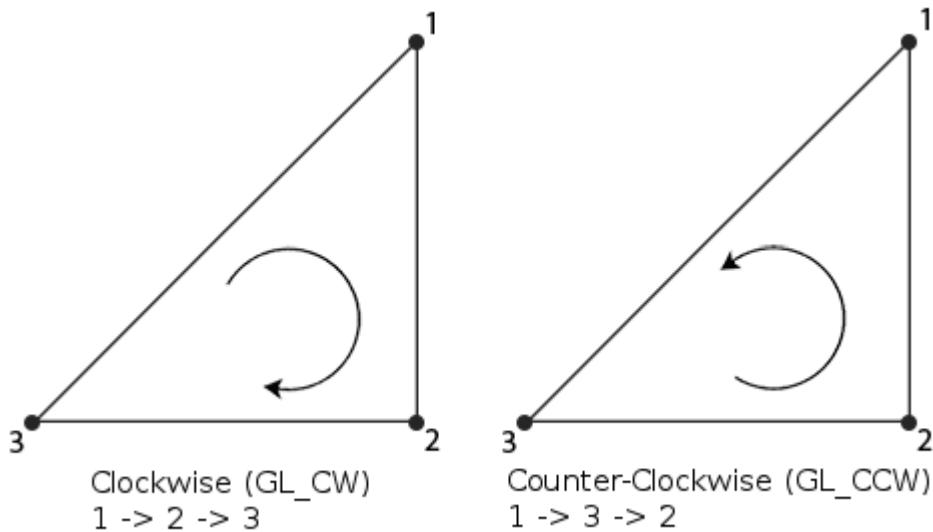


Figure 14: Two triangles, drawn with clockwise and counterclockwise vertices. OpenGL considers the clockwise triangle to be seen from the back and it would be culled.^[10]

OpenGL also allows us to change whether front or back faces are culled. For instance, we can set it so faces considered to be front facing are culled instead.

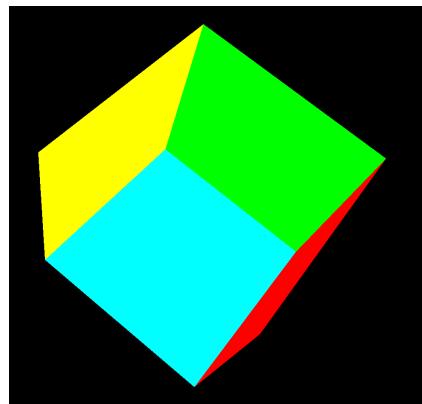
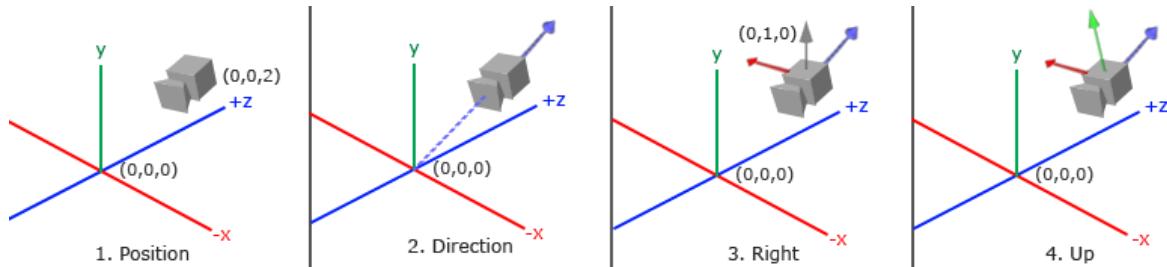


Figure 15: A cube with all front-facing faces culled.

Camera System

OpenGL has an inbuilt `gluLookAt` function which takes the current position, a “look at position” and an “up” position and orients the camera to match all three.

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Figures 16 and 17: How the `gluLookAt` function works. In the top matrix R is the “right” vector, U is the “up” vector, D is the “direction” vector and P is the camera’s position. The bottom image shows a visual representation of these four vectors on the camera.^[11]

While this simplifies the amount of maths needed substantially, we still need to start by defining the three Euler angles. These look as follows:

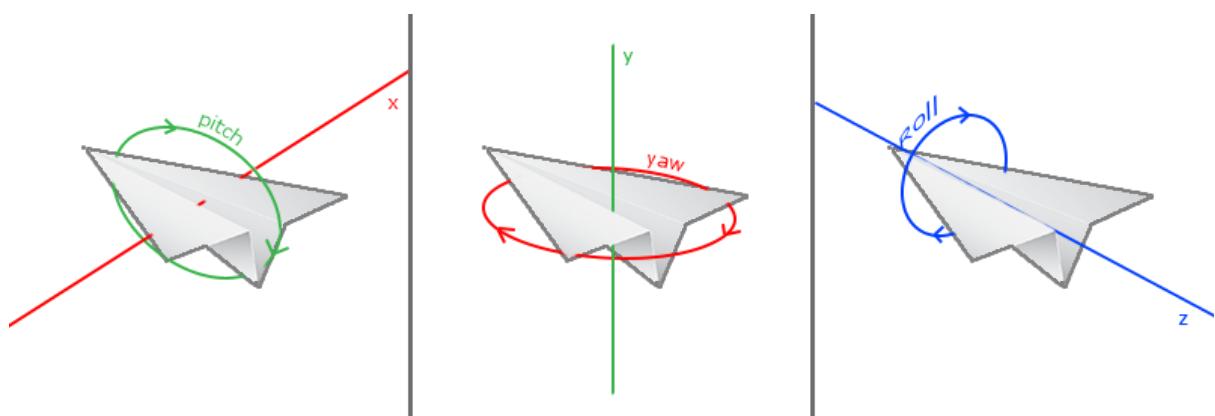


Figure 18: The three Euler angles.^[11]

These angles can then be updated by the program, in a way corresponding to real plane controls. In order to move the camera, we must consider the x and y components of our rotations. For pitch and roll, this looks as follows:

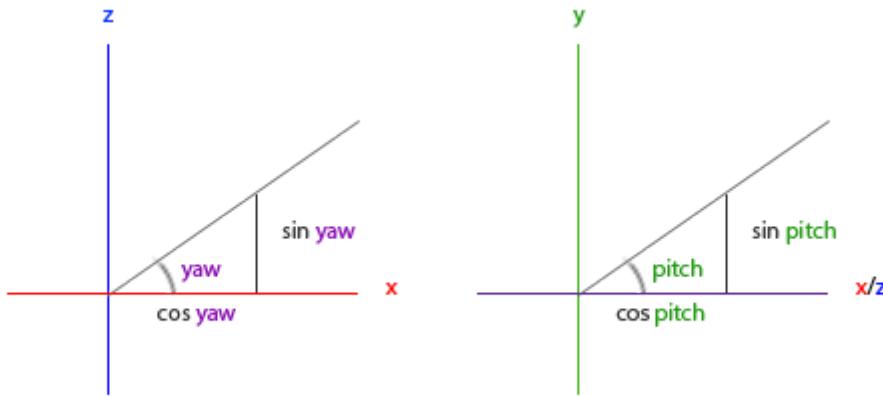


Figure 19: Determining the x and y components of yaw and pitch.^[11]

Using this, we can construct a “front” coordinate that is 1 unit away from the camera, which represents our direction vector. By taking the cross product of direction and up relative to the world (ie. $(0,1,0)$) we can get a right vector. The cross product is simply defined as the vector that is perpendicular to both vectors, and thus in the plane normal to the plane containing them. Taking the cross product of right and direction, we get our relative up vector, which is the third vector we need to rotate our camera.

Once all three are implemented, we will have a working camera system that can look at all 360 degrees around itself.

1.7 Prototyping

The code for all three prototypes is available in the appendix section of the document.

Prototype 1

The goal of Prototype 1 was to familiarise myself with OpenGL. I set out to achieve a basic task - rendering a 3d wireframe cube. After installing all the required modules, I followed an online tutorial^[12] to learn the basics of pyOpenGL, and copied the code while taking notes and writing comments to better understand it. While I had some trouble getting the screen to work, which was done through PyGame, the code itself was simple to understand for me and I was able to run it effortlessly.

The wire cube works by defining 8 vertices in a tuple, and then creating 12 edges that contain two numbers indexing the vertices.

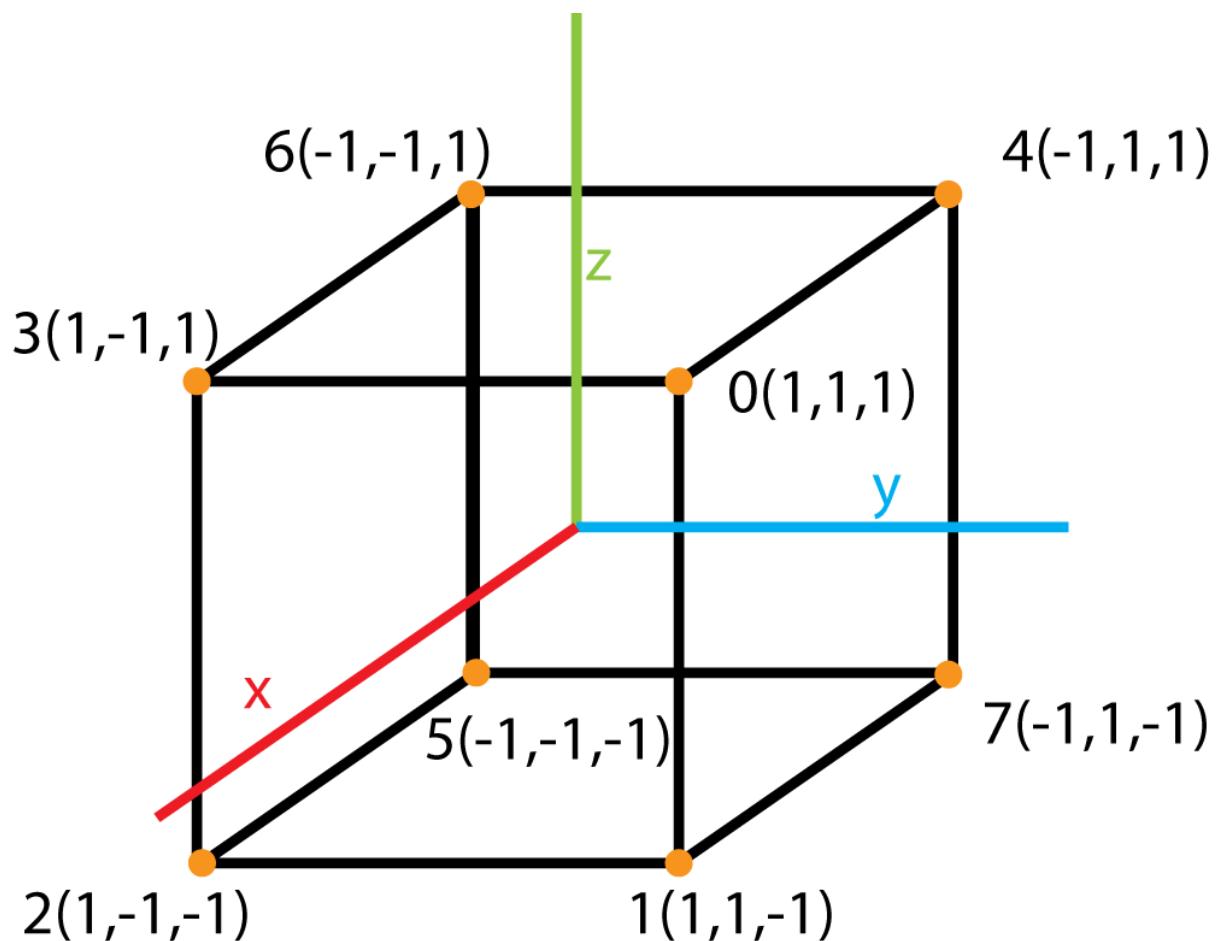


Figure 20: A diagram of the points constituting the cube.^[11]

```

13 #Principles of rendering:
14 #Every object in OpenGL is a collection of points; lines that connect pairs of points; triangles composed of three points; quads composed of four points etc etc
15 cubeVertices = ((1,1,1),(1,1,-1),(1,-1,1),(1,-1,-1),(-1,1,1),(-1,1,-1),(-1,-1,1),(-1,-1,-1)) #Points of a 2x2x2 cube
16 #every edge is given a number, from 0 through to 7, based on its position in our array. To join them:
17 cubeEdges = ((0,1),(0,3),(0,4),(1,2),(1,7),(2,5),(2,3),(3,6),(4,6),(4,7),(5,6),(5,7))
18 #Finally, connect the vertices via their number to also make a cube.
19

```

Figure 21: The definition of the vertices and edges in the code.

As the program is run, a subroutine is constantly executed which updates the cube and draws the lines via the GL_Lines parameter.

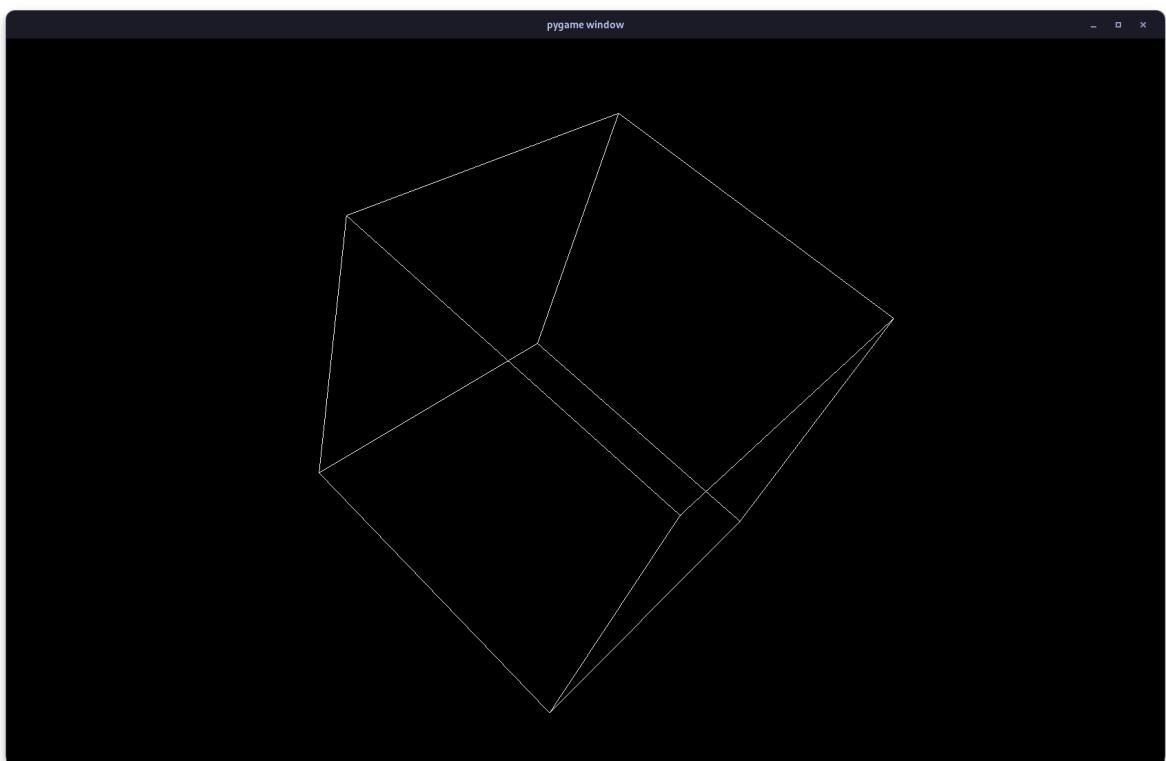


Figure 22: Prototype 1.

Prototype 2

In order to challenge myself, and put my knowledge of culling and transformations to use, I wanted to expand upon Prototype 1 with my own code to create a cube with 6 coloured faces that can be rotated by the user.

Similar to how the edges for the wire cube were defined as indexes of a tuple of vertices, a tuple of indexes was used to make 6 faces. A separate tuple then contained the colours of each face.

I had to make sure that each face had its vertices defined in the correct order as per the culling system, which uses counterclockwise-defined faces as front faces. This took a substantial amount of trial and error before I got it to work right.

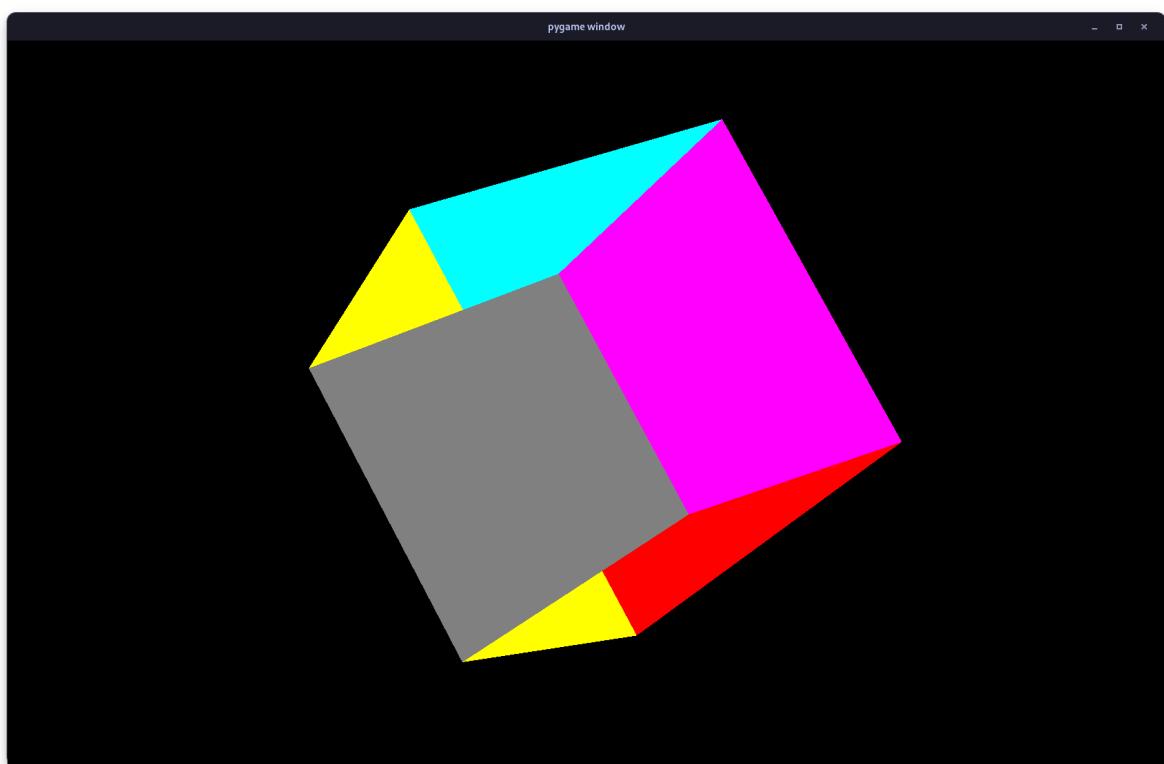


Figure 23: Improperly defined faces, causing them to clip through each other.

Adding user controls was less difficult. Pygame has an inbuilt `pg.key.get_pressed()` method, which can be referenced to check what keys have been pressed. I chose to use the QAWSED keys to control the three axes of rotation.

```
#Matrices are done in REVERSE
if keys[K_w]:
    glRotatef(1, 0, 0, 1) # angle, x, y, z
if keys[K_e]:
    glRotatef(-1, 0, 0, 1)
if keys[K_s]:
    glRotatef(-1, 0, 1, 0)
if keys[K_d]:
    glRotatef(1, 0, 1, 0)
if keys[K_q]:
    glRotatef(-1, 1, 0, 0)
if keys[K_a]:
    glRotatef(1, 1, 0, 0)
#glPopMatrix()
```

Figure 24: The code behind the rotation of the cube.

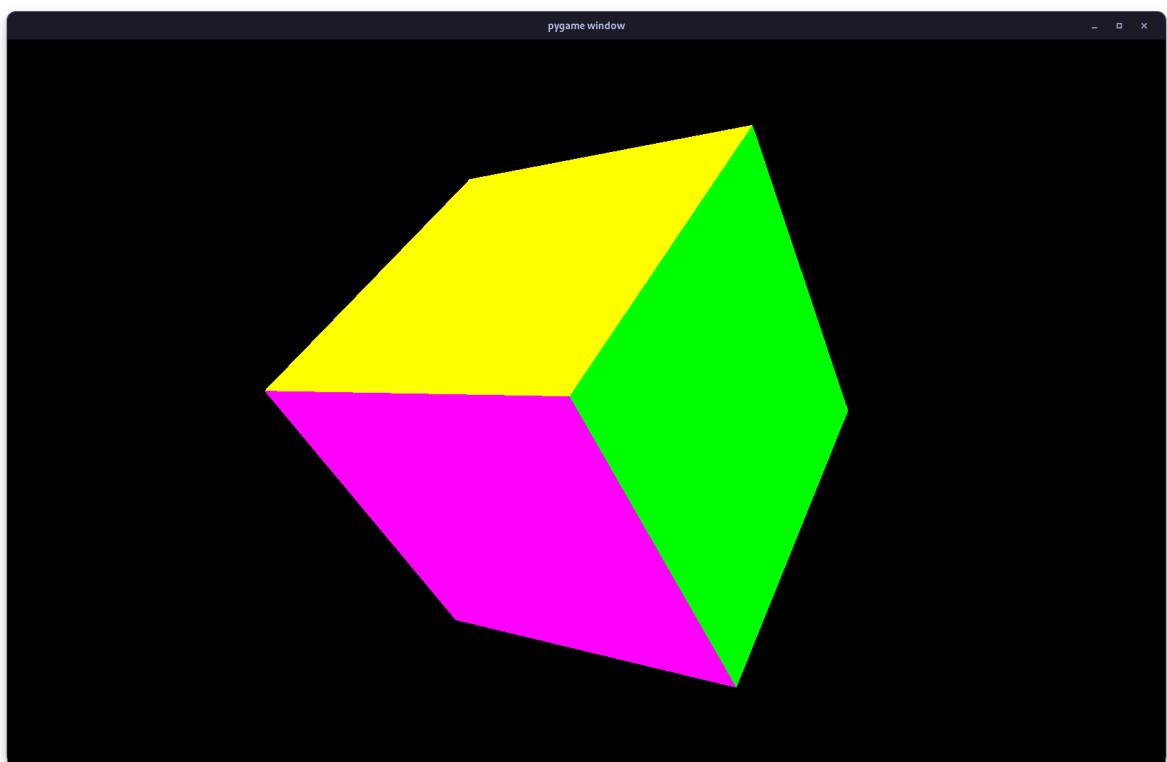


Figure 25: Prototype 2.

Prototype 3

Using what I had learnt from the previous two prototypes, I decided to start from scratch and write a program that would generate a plane of

triangles with random height. This would serve as a prototype for the ground terrain of the final product.

In order to draw the plane, I defined an x and y height. For every vertice in this plane, two triangles were drawn as so:

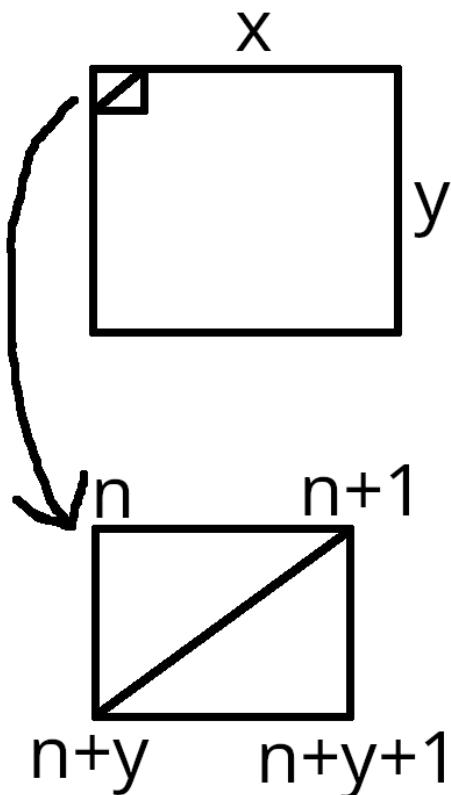


Figure 26: A plane of $2xy$ triangles and how they are rendered. The values at each corner are then used to index an array of points with randomly generated y values.

At the start of each instance, every single vertex was defined in a list, together with a randomly generated y value and colour.

One issue I encountered was the extremely poor performance. I dealt with this by fixing several bugs and replacing the linked list of vertices with a NumPy array.

Another issue was triangles seemingly generating across the entire grid:

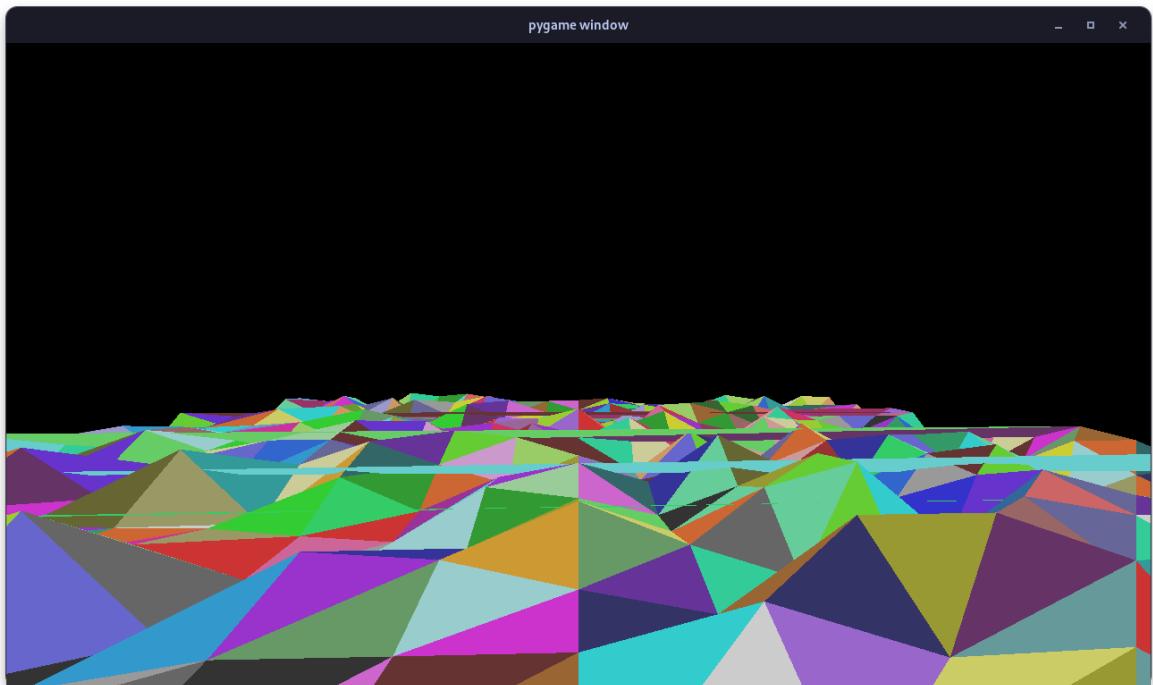


Figure 27: Notice the triangles at the edge of the plane generating across the entire map.

The fix for this turned out to be trivial when I revisited my diagrams of how the plane is generated. At the edge of every column of the plane, $n+1$ would be at $x=0$ on the next row down - on the other side of the entire plane. To prevent this from happening, I added this check in the code:

```
if i%xlength == xlength-1:
    pass
```

Figure 28: Checks if the vertex is at the end of its respective column. If its location would mean drawing a triangle to the next row down, ignore it.

I also used `glTranslatef` to add a basic WASD movement system, although the ability to turn the camera wasn't yet implemented due to the greater complexity of the camera system. Using `glutBitmapString`, I

also implemented an FPS counter so that I could keep track of the performance of my program.

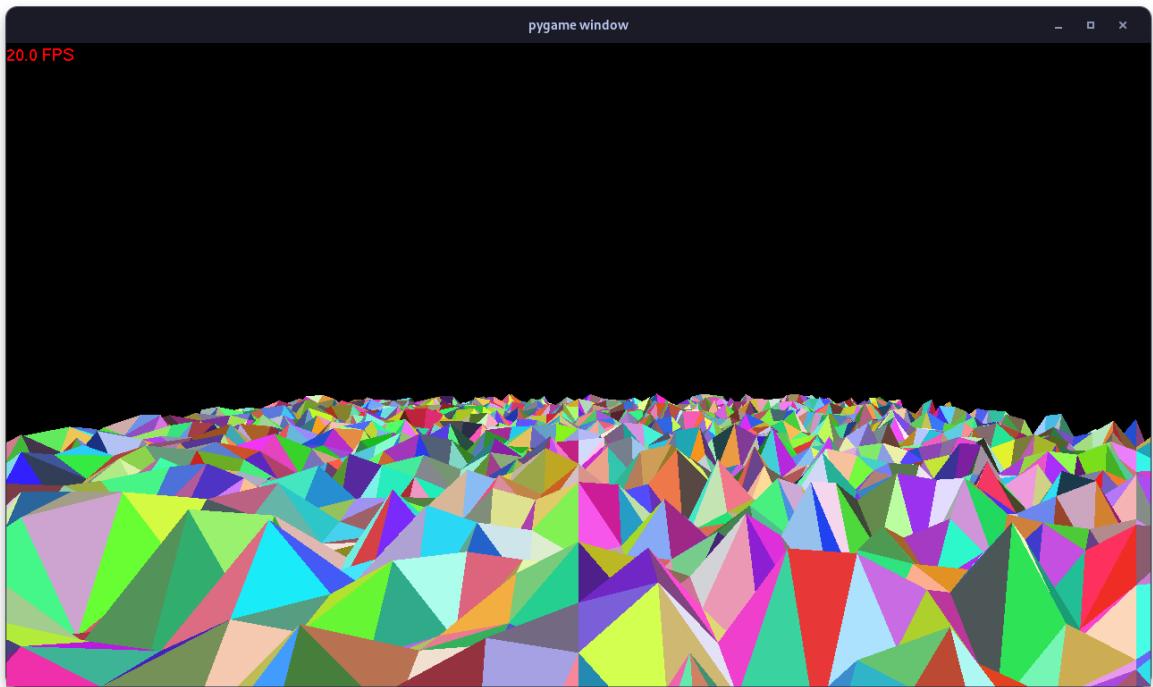


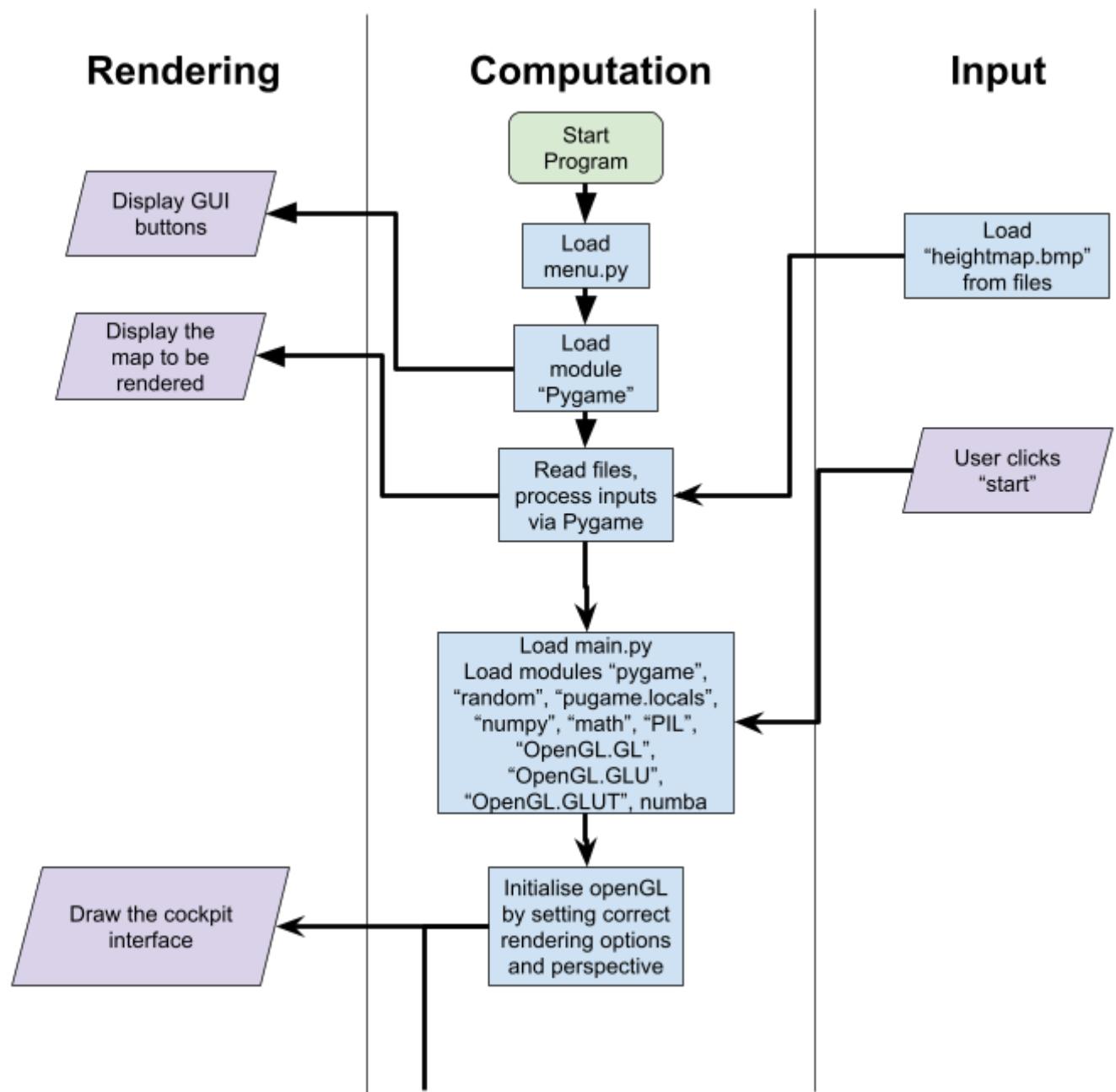
Figure 29: Prototype 3. 5,000 triangles drawn at 20fps.

2 Documented Design

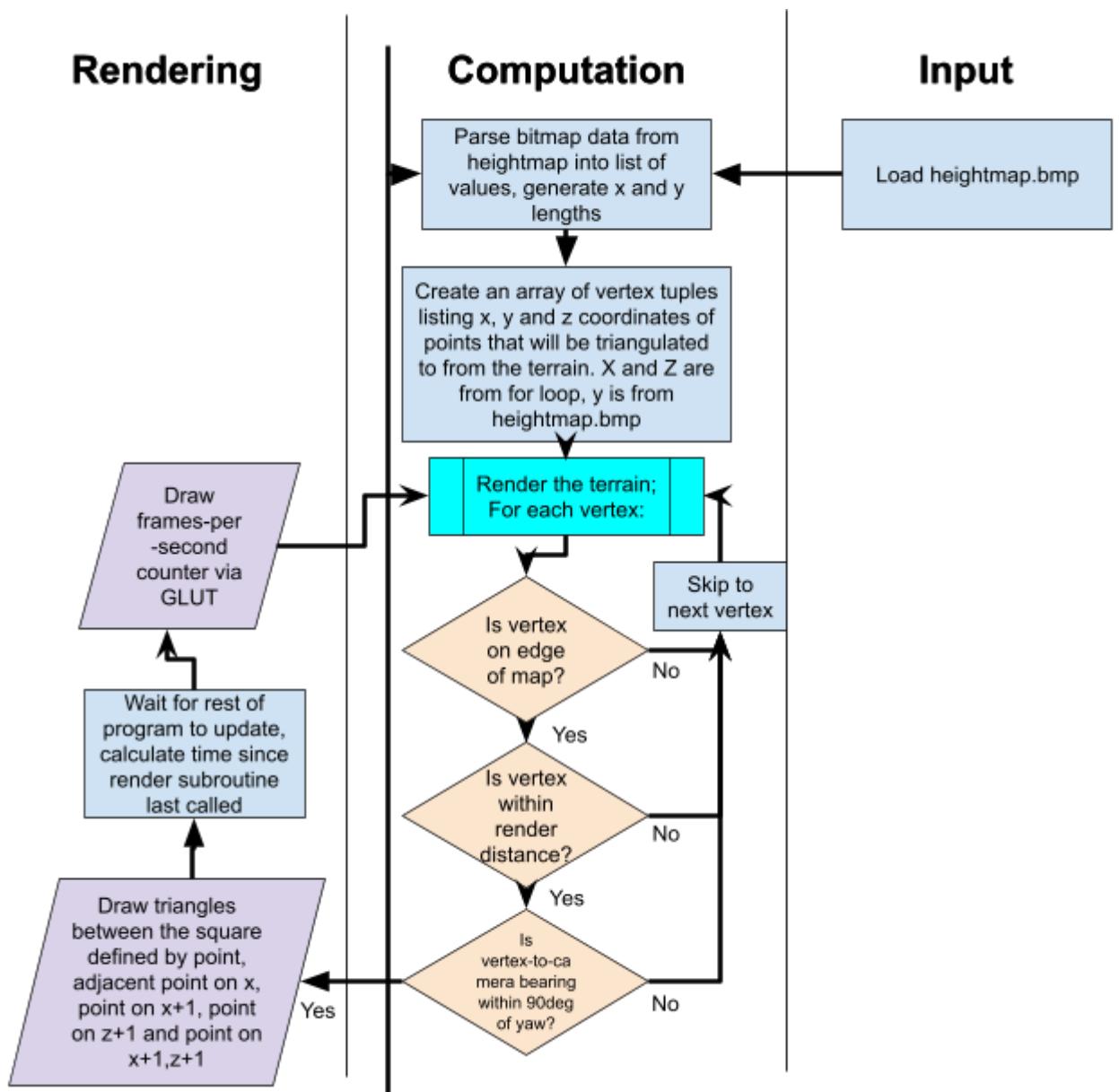
2.1 Swimlane diagram

A swimlane diagram documents how various aspects of a program may interact in a flow chart. Here, I have separated the program into three main processes: input, rendering and computing. Input regards the physical keystrokes given by the player to the program, as well as external files that are loaded into the program; computing regards how the actual program is run, and rendering concerns itself with the output of the program onscreen.

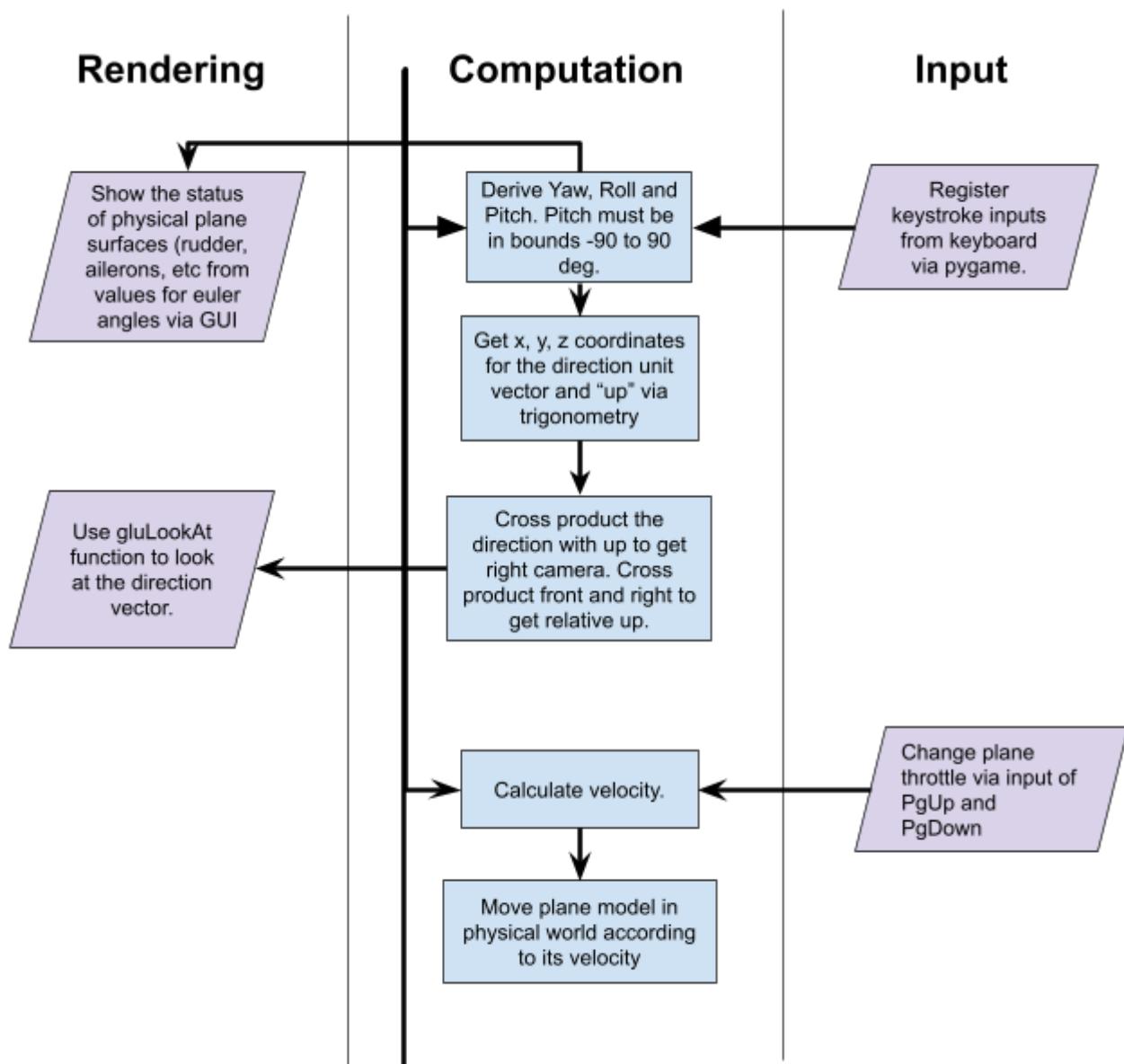
Menu

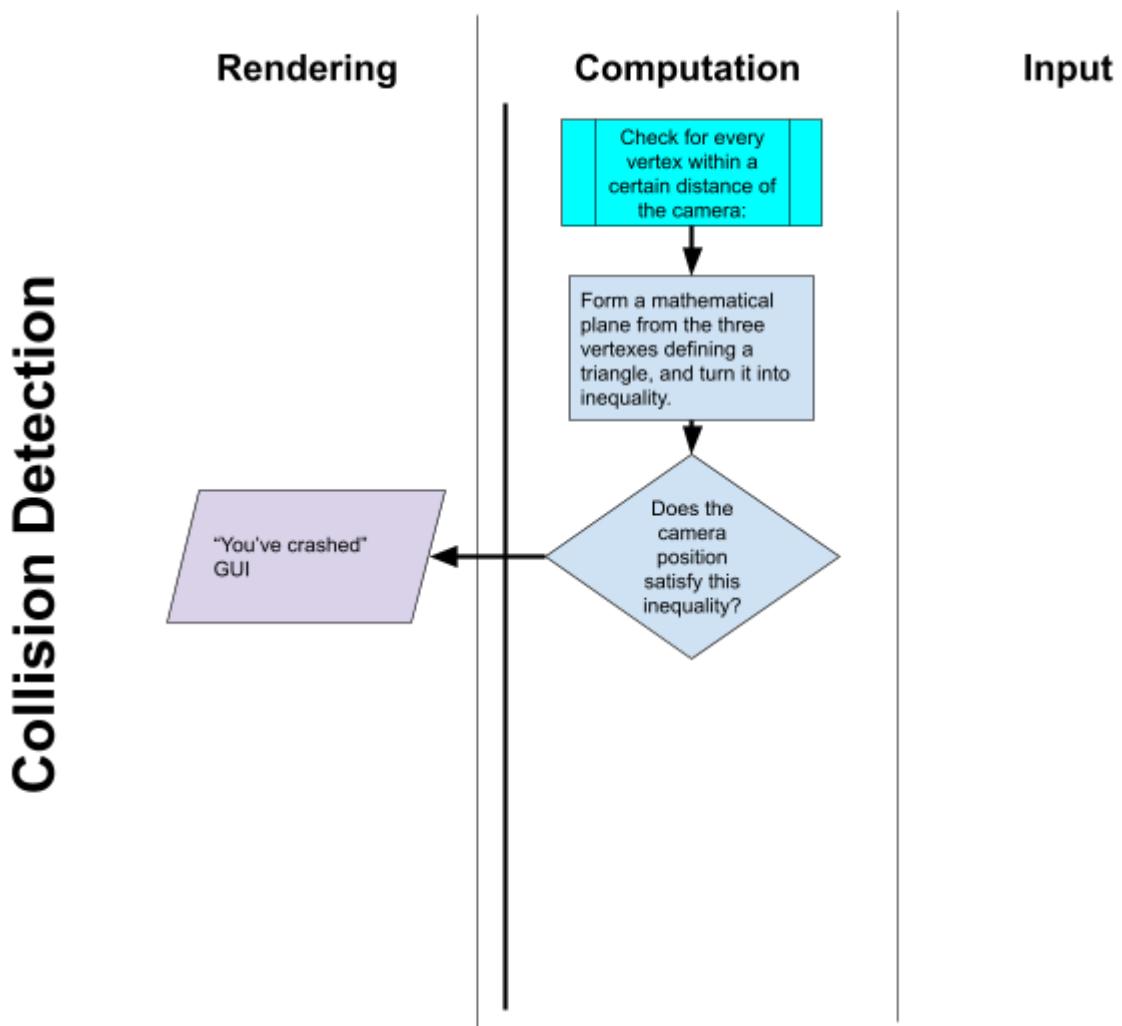


Map Rendering



Camera System





2.2 File Structure

- OpenPyFlightSim
 - menu.py
 - main.py
 - heightmap.bmp

2.3 Description of Key Algorithms

2.3.1 Parsing a .bmp image

The Bitmap image format stores image data as an array of pixels, and is a common uncompressed format. Images are stored with

colour values that correspond to colours / values as according to a colour table. For my project, I will choose a grayscale colour table of 2^8 bits (values 0-255), with each value representing a height. The advantage of this over a format such as 16-bit-per-pixel, which uses no colour table, is that I can save space on the program, and reduce rendering time.

2.3.2 Rendering terrain

2.3.3 Camera system

2.3.4 Collision detection

3 Implementation

4 Technical Solution

5 Testing

6 Evaluation

7 References

Analysis

1. <https://ysflight.org/>, accessed 13 July 2023
2. <https://github.com/captainys/YSFLIGHT>, accessed 13 July 2023
3. https://www.usenix.org/legacy/events/usenix04/tech/sigs/full_papers/perry/perry.pdf, USENIX Association, accessed 13 September 2023
4. <https://www.opengl.org/>, accessed 13 July 2023
5. https://www.opengl.org/resources/libraries/glut/glut_downloads.php, accessed 13 July 2023
6. <https://numpy.org/>, accessed 13 July 2023
7. <https://www.pygame.org/>, accessed 13 July 2023
8. <https://numba.pydata.org/>, accessed 29 August 2023
9. Images of matrices courtesy of
http://www.codinglabs.net/article_world_view_projection_matrix.aspx, written by Marco Alami, accessed 13 July 2023
10. Image courtesy of https://www.khronos.org/opengl/wiki/Face_Culling, accessed 13 July 2023
11. Image and information courtesy of
<https://learnopengl.com/Getting-started/Camera>, accessed 29 August 2023.
12. <https://stackabuse.com/advanced-opengl-in-python-with-pygame-and-pyopengl/>, accessed 27 June 2023

8 Appendix

Prototype 1

```
#thank you to

https://en.wikibooks.org/wiki/OpenGL\_Programming/Installation/Linux for
helping me install OpenGL

# resources used:

# https://stackabuse.com/advanced-opengl-in-python-with-pygame-and-pyopengl/

# Init:

import pygame as pg
from pygame.locals import *

from OpenGL.GL import *
from OpenGL.GLU import *

print("Packages successfully loaded.")
```

```

#Principles of rendering:

#Every object in OpenGL is a collection of points; lines that connect pairs
of points; triangles composed of three points; quads composed of four points
etc etc

cubeVertices =
((1,1,1), (1,1,-1), (1,-1,-1), (1,-1,1), (-1,1,1), (-1,-1,-1), (-1,-1,1), (-1,1,-1))

#Points of a 2x2x2 cube

#every edge is given a number, from 0 through to 7, based on its position in
our array. To join them:

cubeEdges =
((0,1), (0,3), (0,4), (1,2), (1,7), (2,5), (2,3), (3,6), (4,6), (4,7), (5,6), (5,7))

#Finally, connect the vertices via their number to also make a cube.

#It helps to draw a diagram at this point. The page I'm following here has
this diagram:

#
https://stackabuse.s3.amazonaws.com/media/advanced-opengl-in-pygame-and-pyopengl-2.png

cubeQuads = ((0,3,6,4), (2,5,6,3), (1,2,5,7), (1,0,4,7), (7,4,6,5), (2,3,0,1))

# draw wire cube

def wireCube():

    glBegin(GL_LINES) # begin drawing. GL_LINES tells us we'll be drawing
lines

        for cubeEdge in cubeEdges:
            for cubeVertex in cubeEdge:
                glVertex3fv(cubeVertices[cubeVertex])
                """
What does this command mean?
glVertex: defines a vertex

```

```

glVertex3: 3-coordinate vertex
glVertex3f: of type GLfloat
glVertex3fv: put inside a vector (tuple) as opposed to
glVertex3fl which uses a list of arguments instead
So, we are drawing out all our vertices one by one.

"""
glEnd() #stop drawing

#following similar logic:

def solidCube():
    glBegin(GL_QUADS)
    for cubeQuad in cubeQuads:
        for cubeVertex in cubeQuad:
            glVertex3fv(cubeVertices[cubeVertex])
    glEnd()

def main():
    pg.init()
    display = (1680, 1050)
    pg.display.set_mode(display, DOUBLEBUF|OPENGL)

    gluPerspective(45, (display[0]/display[1]), 0.1, 50.0)
    glTranslatef(0.0, 0.0, -5)

    while True: #allows us to actually leave the program
        for event in pg.event.get():
            if event.type == pg.QUIT:
                pg.quit()
                quit()

```

```

# Transformation matrix is T1 before this block of code

#use an inbuilt openGL stack, allowing us to render multiple things:

#gPushMatrix()

#Matrices are done in REVERSE

glRotatef(1, 1, 0, 1)

wireCube()

#gPopMatrix()

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

#solidCube()

wireCube()

pg.display.flip() #update window with active buffer contents

pg.time.wait(10)

if __name__ == "__main__":
    main()

```

Prototype 2

```

#thank you to

https://en.wikibooks.org/wiki/OpenGL\_Programming/Installation/Linux for
helping me install OpenGL

# resources used:

# https://stackabuse.com/advanced-opengl-in-python-with-pygame-and-pyopengl/

# Init:

import pygame as pg

from pygame.locals import *

from OpenGL.GL import *

```

```

from OpenGL.GLU import *

print("Packages successfully loaded.")

#Principles of rendering:

#Every object in OpenGL is a collection of points; lines that connect pairs
of points; triangles composed of three points; quads composed of four points
etc etc

cubeVertices =
((1,1,1),(1,1,-1),(1,-1,-1),(1,-1,1),(-1,1,1),(-1,-1,-1),(-1,-1,1),(-1,1,-1))

#Points of a 2x2x2 cube

#every edge is given a number, from 0 through to 7, based on its position in
our array. To join them:

cubeEdges =
((0,1),(0,3),(0,4),(1,2),(1,7),(2,5),(2,3),(3,6),(4,6),(4,7),(5,6),(5,7))

#Finally, connect the vertices via their number to also make a cube.

#It helps to draw a diagram at this point. The page I'm following here has
this diagram:

#
https://stackabuse.s3.amazonaws.com/media/advanced-opengl-in-python-pygame-and-pyopengl-2.png

cubeQuads = ((0,3,2,1),(5,7,1,2),(0,1,7,4),(2,3,6,5),(3,0,4,6),(4,7,5,6))

colours =
((0,0,1),(0,1,0),(1,0,0),(0,1,1),(1,0,1),(1,1,0),(0.5,0.5,0.5),(1,1,1))

#pygame

# draw wire cube

def wireCube():

    glBegin(GL_LINES) # begin drawing. GL_LINES tells us we'll be drawing
lines
        for cubeEdge in cubeEdges:

```

```

for cubeVertex in cubeEdge:

    glVertex3fv(cubeVertices[cubeVertex])

"""

What does this command mean?

glVertex: defines a vertex
glVertex3: 3-coordinate vertex
glVertex3f: of type GLfloat
glVertex3fv: put inside a vector (tuple) as opposed to
glVertex3fl which uses a list of arguments instead

So, we are drawing out all our vertices one by one.

"""

glEnd() #stop drawing

#following similar logic:

def solidCube():

    x=0

    glBegin(GL_QUADS)

    for cubeQuad in cubeQuads:
        x+=1

        for cubeVertex in cubeQuad:
            glColor3fv(colours[x])
            glVertex3fv(cubeVertices[cubeVertex])

    glEnd()

def main():

    pg.init()

    display = (1680, 1050)
    pg.display.set_mode(display, DOUBLEBUF|OPENGL)

```

```

#gLEnable(GL_CULL_FACE)
#gLCullFace(GL_FRONT)

gluPerspective(45, (display[0]/display[1]), 0.1, 50.0)
glTranslatef(0.0, 0.0, -5)

while True: #allows us to actually leave the program
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
            quit()

# Transformation matrix is T1 before this block of code
#use an inbuilt openGL stack, allowing us to render multiple things:
#glPushMatrix()

keys=pg.key.get_pressed()

#Matrices are done in REVERSE

if keys[K_w]:
    glRotatef(1, 0, 0, 1) # angle, x, y, z

if keys[K_e]:
    glRotatef(-1, 0, 0, 1)

if keys[K_s]:
    glRotatef(-1, 0, 1, 0)

if keys[K_d]:
    glRotatef(1, 0, 1, 0)

if keys[K_q]:
    glRotatef(-1, 1, 0, 0)

if keys[K_a]:
    glRotatef(1, 1, 0, 0)

```

```

#glPopMatrix()

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

glDepthMask(GL_TRUE)
glDepthFunc(GL_LESS)
glEnable(GL_DEPTH_TEST)
glEnable(GL_CULL_FACE)
glCullFace(GL_BACK)
glFrontFace(GL_CCW)
glShadeModel(GL_SMOOTH)
glDepthRange(0.0,1.0)

solidCube()
#wireCube()

pg.display.flip() #update window with active buffer contents
pg.time.wait(10)

if __name__ == "__main__":
    main()

```

Prototype 3

```

import pygame as pg

from pygame.locals import *
from random import *
import numpy as np

from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *

```

```

from numba import njit

print("Packages successfully loaded.")

#####
#length = 50
xlength = 50

def mapGen():

    #Create our matrix for both the 20x20 surface and the colours
    vertList = []
    coloursList = []

    for zcord in range(-zlength//2,zlength//2):
        for xcord in range(-xlength//2,xlength//2):

            vertList.append((xcord/5,uniform(-1,-0.5),zcord/5)) #enables
            steps of 0.1m

    coloursList.append(((uniform(0.1,1)),(uniform(0.1,1)),(uniform(0.1,1))))
    #define random RGB values for all corresponding vertices

    coloursList.append(((uniform(0.1,1)),(uniform(0.1,1)),(uniform(0.1,1))))
    #every vertice has 2 triangles associated with it

    return np.array(vertList), np.array(coloursList)

def triangulate(p1,p2,p3,colour):

    #remember: counter clockwise rotation

    glBegin(GL_TRIANGLES)
    glColor3fv(colour)
    glVertex3fv(p1)

```

```

glVertex3fv(p2)

glVertex3fv(p3)

glVertex3fv(p1)

glEnd()

def genTerrain(mapMatrix, coloursList):

    try:

        for i in range(len(mapMatrix)):

            #This stops vertices at the edge from rendering triangles - this
            previously led to triangles being rendered across the entire map

            if i%xlength == xlength-1:
                pass
            else:
                #the two triangles adjacent to any vertex

                triangulate(mapMatrix[i+1],mapMatrix[i],mapMatrix[i+xlength],coloursList[2*i]
)
                triangulate(mapMatrix[i+1],mapMatrix[i+xlength],mapMatrix[i+xlength+1],colour
sList[(2*i)-1])

            except IndexError: #invalid triangle, avoid crashing
                pass

#Define a text rendering framework:

def text(x, y, color, text):

    glColor3fv(color)

    glWindowPos2f(x, y)

    glutBitmapString(GLUT_BITMAP_HELVETICA_18, text.encode('ascii'))

def main():

```

```
pg.init()
pg.font.init()
glutInit()
glutInitDisplayMode(GLUT_RGBA)

my_font = pg.font.Font('freesansbold.ttf', 32)

display = (1280, 720)

screen = pg.display.set_mode(display, DOUBLEBUF|OPENGL)

gluPerspective(60, (display[0]/display[1]), 0.1, 50.0) #fov, aspect,
zNear, zFar

glTranslatef(0.0, 0.0, -5)

mapMatrix, coloursList = mapGen()

while True: #allows us to actually leave the program
    for event in pg.event.get():
        if event.type == pg.QUIT:
            pg.quit()
            quit()

        #regenerate the map
        if event.type == pg.KEYDOWN:
            if event.key == pg.K_r:
                mapMatrix, coloursList = mapGen()
                pg.time.wait(1)
```

```
timeTaken=pg.time.get_ticks()

keys=pg.key.get_pressed()

#clear buffer

glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)

if keys[K_w]:
    glTranslatef(0, 0, 0.2) # angle, x, y, z

if keys[K_s]:
    glTranslatef(0, 0, -0.2)

if keys[K_a]:
    glTranslatef(0.2, 0, 0) # angle, x, y, z

if keys[K_d]:
    glTranslatef(-0.2, 0, -0)

if keys[K_o]:
    glRotatef(1, 0, 0, 1) # angle, x, y, z

if keys[K_l]:
    glRotatef(-1, 0, 0, 1)

if keys[K_i]:
    glRotatef(-1, 0, 1, 0)

if keys[K_k]:
    glRotatef(1, 0, 1, 0)

if keys[K_u]:
    glRotatef(-1, 1, 0, 0)

if keys[K_j]:
    glRotatef(1, 1, 0, 0)

#culling

glDepthMask(GL_TRUE)
```

```
glDepthFunc(GL_LESS)
 glEnable(GL_DEPTH_TEST)
 glEnable(GL_CULL_FACE)
 glCullFace(GL_BACK)
 glFrontFace(GL_CCW)
 glShadeModel(GL_SMOOTH)
 glDepthRange(0.0,1.0)

genTerrain(mapMatrix, coloursList)
timeTaken=1/((pg.time.get_ticks()-timeTaken)/1000)
text(0, 700, (1, 0, 0), str(round(timeTaken,1))+' FPS')

pg.display.flip() #update window with active buffer contents
pg.time.wait(10) #wait a bit, avoids speed of simulation from being
speed of execution of code

if __name__ == "__main__":
    main()
```

heightmap.bmp

