Project Volare

Toby Scamman

Candidate Number - 4168
Centre Number - 46123

Contents

1 Analysis **4**
1.1 Current Scenario . . . 4
1.2 End Users . . . 4
1.3 Solution . . . 4
1.4 Objectives . . . 4
1.5 Research . . . 5
1.5.1 Senior interviews . . . 5
1.5.2 Cadet interviews . . . 5
1.5.3 Programming implementation . . . 5
1.5.4 Alternative Flight Simulators . . . 7
1.6 Modelling . . . 8
1.6.1 Aircraft Type . . . 8
1.6.2 Keyboard Input Controls . . . 8

2 Design **10**
2.1 Objects . . . 10
2.2 Data Structures . . . 10
2.3 Physics . . . 10
2.3.1 Forces . . . 11
2.4 Torques . . . 11
2.4.1 Motion . . . 12
2.5 Aerodynamics . . . 13
2.5.1 Thrust . . . 13
2.5.2 Lift . . . 14
2.5.3 Weight . . . 15
2.5.4 Drag . . . 15
2.5.5 Balancing Moment . . . 16
2.6 Aerodynamic Coefficients . . . 16
2.6.1 Option 1: Lookup Tables . . . 16
2.6.2 Option 2: Real Time Calculation . . . 17
2.6.3 Aerodynamic Parameters . . . 18
2.6.4 Aerodynamic Coefficient Calculations . . . 20
2.7 Delta Time . . . 25
2.8 Bounding Boxes . . . 26
2.8.1 Axis Aligned vs Oriented Bounding Boxes . . . 26
2.9 Making OBBs . . . 28
2.9.1 From AABBs . . . 28
2.9.2 Complex OBB creation . . . 28
2.9.3 Finding the centre . . . 28
2.9.4 Covariance Matrix . . . 29
2.9.5 Eigendecomposition . . . 29
2.9.6 Solving the cubic equation . . . 30
2.9.7 Depressing Cubics . . . 30
2.9.8 Cardano's formula . . . 31
2.9.9 Complex Cube Root . . . 31
2.9.10 Back to Cardano's formula . . . 32
2.9.11 Finding Eigenvectors . . . 32
2.9.12 Finding the dimensions of the OBB . . . 33
2.10 Intersection testing . . . 34
2.10.1 Intersection Testing and Separating Axis Theorem . . . 34
2.10.2 OBB and OBB Intersection testing . . . 35
2.10.3 Triangle and OBB Intersection . . . 39
2.11 Collision Response . . . 41
2.11.1 Impulses . . . 42
2.12 Terminal Velocity . . . 44
2.12.1 Drag Coefficient (again) . . . 44

2.13 Frontal Area . . . 45
2.13.1 What is 'Frontal Area'? . . . 45
2.13.2 Problem discussion . . . 45
2.13.3 Projection . . . 52
2.13.4 Convex Hull Algorithms . . . 53
2.13.5 Graham Scan . . . 54
2.13.6 Union . . . 55
2.13.7 The Greiner-Hormann Clipping Algorithm [26] . . . 55
2.13.8 Edge Intersection . . . 55
2.13.9 Threshold Designation . . . 56
2.13.10 Union Creation . . . 57
2.13.11 Edge Cases . . . 57
2.13.12 Recursive Polygon construction . . . 58
2.13.13 Shoelace Formula . . . 58
2.13.14 Graphics design . . . 59

3 Technical Solution **60**
3.1 Important features . . . 60
3.2 script.js . . . 61
3.3 physical_interactions.js . . . 65
3.4 constants.js . . . 104
3.5 terminal_velocity . . . 125

4 Testing **138**
4.1 Objectives . . . 138
4.2 End User & Client Testing . . . 139
4.3 Examples . . . 139
4.4 Testing Aids . . . 140

5 Evaluation **140**
5.1 Objectives . . . 140

6 Appendices **143**
6.I Meeting Log . . . 143
6.I.A Meeting 1 - 18/03/2025 . . . 143
6.I.B Meeting 2 - 21/05/2025 . . . 144
6.II Bibliography . . . 145

1 Analysis

Overview

As a sergeant in the RAF Cadets at my school, I will make a flight simulator to teach cadets about the basics of flying, which they can access whenever they want.

1.1 Current Scenario

We have three computers with flight simulators (Microsoft Flight Simulator and Digital Combat Simulator) installed and set up in the RAF hut, each of which has accompanying peripherals. However, to access these few computers, cadets must share between themselves whilst supervised by a Senior Non-Commissioned Officer during certain lunchtime sessions only twice a week. In addition, cadets who go to a different school cannot access them at all, and are unable to learn about or experience flight simulation.

1.2 End Users

My client will be Flight Lieutenant Williams, who is in charge of the RAF cadets. We will have meetings every few weeks, on Tuesday afternoons after session.

My end users will be young cadets who have varying levels of experience - from those who have used flight simulators before to those who are completely new to them. It can be assumed that they don't have much programmatical experience, so programming knowledge should not be necessary in order to use my program.

1.3 Solution

I will make a flight simulator which is accessible to all cadets whenever they want. This will involve 3D models of aircraft which are likely to be 3MB to 10MB [1], and scenery models, although these will be generated in-engine and are therefore of a negligible file size.

Cadets will access this program via their own personal computers, so the program should have a low computer specification requirement, to allow cadets to access it at home with their current hardware.

1.4 Objectives

I have collated a list of requirements that the program will need to fulfill in the form of specific, measurable objectives. These are based off my own research, my own ambitions and expectations as well as my first discussion with my client, Flight Lieutenant Williams [Appendix 6.I.A].

1. Receive user inputs which control the aircraft
   (a) **A** and **D** keys will control the ailerons, allowing the aircraft to roll clockwise and anticlockwise.
   (b) **W** and **S** keys will control the elevators, allowing the aircraft to pitch up and down.
   (c) **Up** and **Down** will control the throttle, allowing the aircraft to accelerate and decelerate.
   (d) The mouse will control the camera position.
2. Calculate the impact of forces on the aircraft
   (a) Forces applied at the centre of gravity should provide translational acceleration for the aircraft.
   (b) Forces applied elsewhere should generate torque and therefore angular acceleration for the aircraft.
   (c) Each frame, the acceleration should be applied to the velocity.
   (d) Each frame, the velocity should be applied the aircraft's position
3. Simulate physics and calculate the forces which are applied to the aircraft:
   (a) **Thrust** - the acceleration of the aircraft.
   (b) **Drag** - the friction and air resistance opposing the motion of the aircraft.
   (c) **Lift** - the upwards force allowing the aircraft to take off and remain airborne.
   (d) **Weight** - the result of gravity acting on mass which applies downwards on the aircraft.

(e) **Terminal Velocity** - A velocity limit of the aircraft.

4. Display the 3D model of the aircraft
   (a) Display an aircraft model on the screen using a 3D graphics library such as OpenGL.
   (b) Simulate light and shadow on the aircraft.
5. Render the scene with respect to a camera.
   (a) The camera will change position based on the mouse to provide the user with their desired perspective.
6. Background graphics
   (a) Render the airfield to allow the user to see where they should take off from.
7. Airplane collisions
   (a) Incorporate collision detection, so that intersection tests are carried out between the aircraft and the ground.
   (b) Respond to the collision by applying a reaction force against the aircraft.
   (c) The collision should also generate friction.

1.5 Research

1.5.1 Senior interviews

In order to explore the current state of flight simulator access, I spoke with Flight Sergeant Knowles, who has been overseeing the flight simulator use during lunch times. He told me that the flight simulators were only accessible on a Tuesday and Thursday lunchtime, and that cadets who go to a different school are not allowed to come down during lunch and therefore never get a chance to use them. Therefore, he believed that making my own flight simulator would greatly benefit the cadets.

I then spoke with my client, Flight Lieutenant Williams, about this problem [Appendix 6.I.A]. He also agreed that cadets from the other school did not have sufficient access to the flight simulators, as well as suggesting that this would be an effective learning tool, a great demonstration of the potential of programming and a chance for the cadets to have some fun.

1.5.2 Cadet interviews

I spoke to various younger cadets, who all expressed an enthusiasm for the project. All the cadets who went to a different school told me of their lack of experience with flight simulators.

I then interviewed some cadets who went to this school but did not have much experience with flight simulators. They praised the professionalism of the current flight simulator citing good graphics and immersive controls. However, they said that they didn't have the time and they hadn't been taught well enough to be confident using a flight simulator, so they would appreciate and learn from the flight simulator that I would make.

Finally, I interviewed a cadet with a lot of flight simulator experience. He said that he was not as interested in using my program to learn how to control a flight simulator, but to teach other people. He also informed me of current alternatives to the one which I would make, namely **GeoFS** [2], which I will discuss later.

1.5.3 Programming implementation

When I was considering which programming language to use, there were three main options which appealed to me: Python, C++ and JavaScript. All of them have a 3D library (OpenGL/WebGL) which I would use for the 3D rendering, but each language has different benefits and drawbacks.

Python Python is a user-friendly, object-oriented programming language.

Advantages

- I am most confident programming in Python of all the three languages, having done my GCSE in it.
- I already have experience with 3D rendering in Python.

Disadvantages

- Python is by far the slowest of the three languages[3], due to being an interpreted language. This is undesirable for 3D rendering.
- To distribute my program, I would have to export and provide `.py` files to the **End Users**, who we have established may not be confident with programming at all. `.py` files would require some additional knowledge for the cadets to use, as well as a way of running them (such as an IDE). In addition, the required libraries for 3D rendering such as OpenGL would need to be installed on each user's machine, making the program inaccessible to most.
- Even if I used tools such as PyInstaller, I would have to distribute the program as a `.exe` file. Many modern email systems and especially school systems prevent the distribution of `.exe` files, so I would have to share the program using physical means like a USB drive, which defeats the purpose of having an accessible flight simulator for all the cadets.

JavaScript JavaScript is a language which provides web pages with interactivity and can manipulate and access HTML and CSS.

Advantages

- I have programmed 2 websites with JavaScript, so I have some experience with it.
- JavaScript is very commonly used as a client-side language for web pages, so using JavaScript would allow me to distribute my program more easily on a website.
- JavaScript is faster than Python because it uses a 'Just In Time' (JIT) compiler[3].

Disadvantages

- JavaScript is slower than C++ [3], which is still not ideal for 3d Graphics rendering, as a high speed and consequently a high framerate is desirable.

C++ C++ is a low-level language that introduces object-oriented programming into C.

Advantages

- C++ is famously fast due to its use of a compiler rather than an interpreter - by far the fastest out of the three listed here [3]. This makes it ideal for intensive graphics rendering.

Disadvantages

- I have by far the least experience in C++ out of any of the languages here, which would slow down the speed at which I could work.
- In my experience, setting up the environment for C++ has been much more difficult than any of the other two languages, which would delay the start of the **Technical Solution** and reduce my productivity.
- C++ compiles into a `.exe` file, which, as mentioned previously, many modern email systems prevent the distribution of, reducing the accessibility of my program.

Programming Language Decision

Ultimately, I chose JavaScript, because, according to my research[3]:

> "Choosing a language for your application simply because its 'fast' is the ultimate form of premature optimization.".

I believe that choosing C++ over JavaScript would be an example of this, even though performance is important for 3D rendering, as, for this project, ease of distribution and accessibility should be prioritised. Additionally, I believe that my inexperience in C++ would prevent me from utilising its speed, and would cause me to program less efficiently. For Python, the difficulty of distributing a Python project compared to a JavaScript one made me choose the latter.

1.5.4 Alternative Flight Simulators

There are obvious alternatives to making my own flight simulator, as other flight simulators already exist. The ones which I am going to cover will be Microsoft Flight Simulator [4] and Digital Combat Simulator [5], both of which the RAF cadets can access as aforementioned while supervised in the RAF hut, as well as GeoFS [2], a flight simulator which runs in browser.

Each of these have their own strengths, which I can observe and take inspiration from, as well as weaknesses, which is why I have chosen to make my own.

Microsoft Flight Simulator 2024 [4]

Strengths

- Incredible graphics, delivering an immersive flight simulator experience.
- Very popular, so has a lot of community support.

Weaknesses

- Very graphically demanding, as it requires a minimum of 16GB of RAM, 50GB of storage space, an AMD Ryzen 5 2600X or Intel Core i7-6800K processor and a Radeon RX 5700 or GeForce GTX 970 graphics card [4]. This is far beyond what I'd expect most cadets to possess.
- The program costs anywhere from £70 to £200 [4], but the cadets should be able to access a flight simulator for free.
- There are a plethora of in-app purchases for aircraft models, which the cadets might lose money on.

Digital Combat Simulator [5]

Strengths

- Incredible combat.
- Free to play.

Weaknesses

- Very graphically intense, as it requires a minimum of 16GB of RAM, 200GB of storage space, an Intel Core i3 at 2.8 GHz or AMD FX processor and a NVIDIA/AMD 6GB graphics card [5]. The colossal 200GB storage requirement alone is far beyond what a cadet could be reasonably expected to own.
- The program also has in-app purchases, which I wouldn't want to expose cadets to.

GeoFS [2]

Strengths

- Very accessible - can be played in browser.
- Free to play.

Weaknesses

- From my research, the program struggles when unable to use WebGL2. When I have tried it on a computer which is unable to support WebGL2, the program throws an error and is unable to run at all.
- The program again has an in-app purchase - fewer than the previous two, but I would still prefer none.
- Without the in-app purchase, the website is plagued by advertisements.
- The physics can get very inconsistent and unrealistic.
- The controls are poorly communicated and unwieldy to use.

1.6 Modelling

1.6.1 Aircraft Type



Figure 1: A Grob Tutor [6]

For aircraft type, I will choose a Grob Tutor, as it is a simple aircraft which many cadets will actually fly in themselves, adding to the educational aspect of the program.

1.6.2 Keyboard Input Controls

As previously mentioned, the aircraft will be controlled by specific keys which have been selected to resemble the direction of a joystick used in flight simulators. The following is a demonstration of what that would look like [7]:

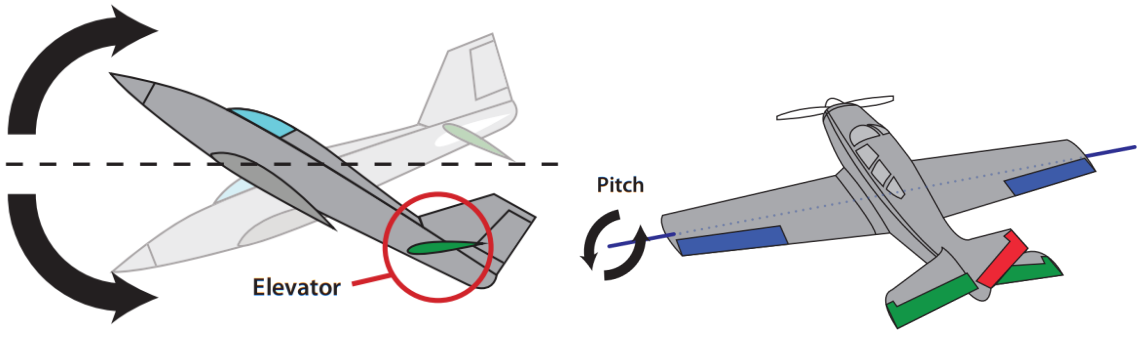


Figure 2: Pitch

The **Elevators** (green) move up and down, causing the aircraft to **'pitch'**. This will be controlled by the **Up** and **Down** keys: the **Up** key will cause the elevators to angle down, making the aircraft pitch down, and the **Down** will cause the aircraft to pitch up.

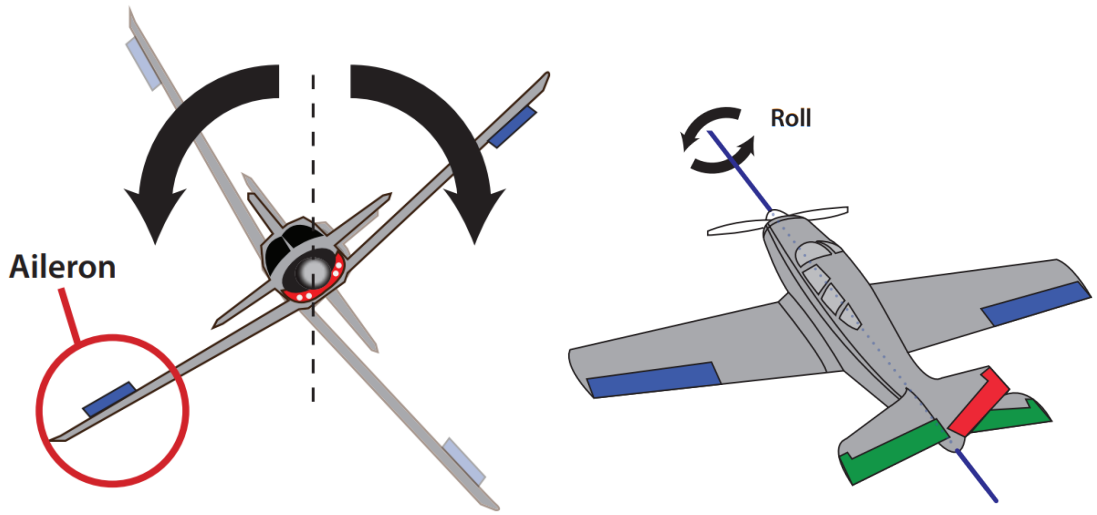


Figure 3: Roll

Each **Aileron** (blue) moves in the opposite direction to each other, causing the aircraft to **'roll'**. This will be controlled by the **Left** and **Right** keys: the **Left** key will cause the left aileron to angle up and the right aileron to angle down, making the aircraft roll anticlockwise. The **Right** key does the opposite and will make the aircraft roll clockwise.

2 Design

2.1 Objects

Objects will be a crucial part of this program.

The main group of objects are those related to the aircraft.

Firstly, we have the Model class, which loads a 3D model.

Then the Aircraft class inherits from that and includes many of its own functions, such as receiving keyboard input and calculating its position based on its velocity and acceleration.

Once the aircraft model has loaded, the Aircraft object creates new objects, Aircraft_Parts which each represent a smaller part of the aircraft. They are responsible for collision response and generating aerodynamic forces.

Each Aircraft_Part has a Collision_Bounding_Box object as well, which is instrumental in finding the frontal area of the aircraft and ensuring that their own objects, OBBs, are of the correct size.

Finally we arrive at the last object of the group: Oriented Bounding Boxes. They encapsulate the aircraft's shape and are pivotal for intersection testing.

Those were just a few examples of the objects in the program, demonstrating how important objects are.

2.2 Data Structures

Binary Tree

I will use a binary tree (`OBB_tree`) to hold all the OBBs of the aircraft. The header node has an OBB as an attribute, as well as two pointers to other nodes. Each node's OBB is made to contain all the aircraft's OBBs underneath it. This structure continues until the pointers are no longer other nodes but the OBBs themselves generated from the geometry of the aircraft's parts instead. This allows for efficient intersection testing: if the terrain doesn't intersect with the largest OBB - the header node's OBB - then it won't intersect with any of the aircraft's OBBs beneath it.

The OBB_tree_nodes have a subdivide method, which splits the collision objects which it encapsulates into two groups, and then creates two OBBs around them. This is how the tree is made.

The terrain is also made from an OBB tree, so the program must navigate two trees simultaneously to find intersections.

Polygonal Circular Doubly Linked List

[H] The `Polygonal_Circular_Doubly_Linked_List` is so called because it uses a linked list, of which the nodes point both forward and back, including the header, to represent a polygon.

The `Polygonal_Circular_Doubly_Linked_List`object contains a pointer to its header node, through which it executes most of its functions, including pushing, popping, cloning, accessing nodes by index and returning an array representation of itself (the last two are for testing purposes).

The `Polygon_Node` makes up the linked list. They have the coordinates of the point they represent as an attribute and contains pointers in both directions. They also have a `next_polygon` attribute which can have another `Polygonal_Circular_Doubly_Linked_List` assigned to it. This provided the linked list with matrix functionality as necessary, to represent disjunct polygons. Each node has an index by which it can be individually accessed if desired, which can be used to identify the root node. An important distinction should be made between intersections and the so-called 'proper nodes'. As each list is used, new nodes which represent intersections are created (see Page 55 for details). These nodes have an `is_intersection` attribute set to true, amongst other attributes. The distinction between them is significant: sometimes the original polygon needs to be traversed ignoring additional intersections, and sometimes intersections are specifically sought out. These intersections also have a pointer to a corresponding intersection in a different `Polygonal_Circular_Doubly_Linked_List`, to form a sort of bridge between them.

2.3 Physics

My second objective contains details about how the aircraft should be physically interacted, specifying 3 important areas: forces, torques and motion.

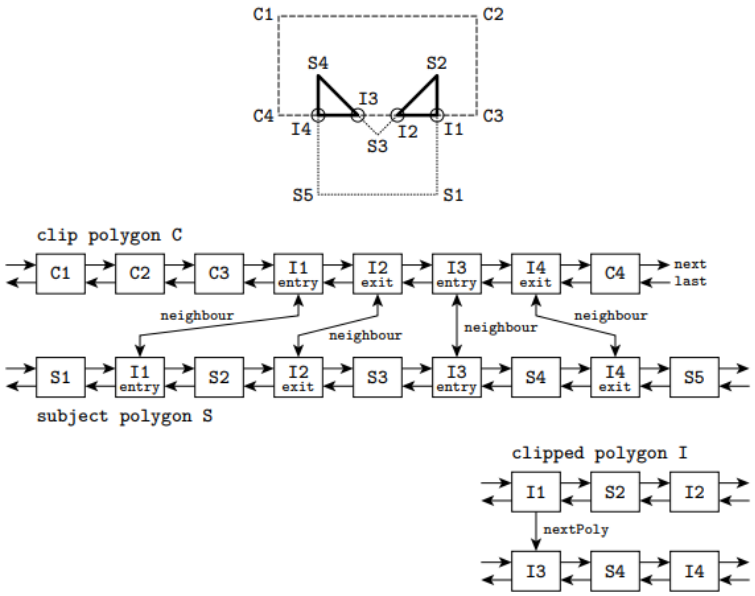


Figure 4: A diagram of a Polygonal Circular Doubly Linked List

2.3.1 Forces

Forces are very simple to incorporate, as one can use Newton's second law to calculate the acceleration which results from their application:

$$F = ma \tag{1}$$

where

F = resultant force, N,

m = mass, kg,

a = acceleration, ms^{-2}

2.4 Torques

Before we can consider anything else about torques, we need to find the centre of gravity for the aircraft. I found the position of the centre of gravity for the aircraft along its longitudinal axis (2.72 metres from the propeller) using the Grob G 115's type certificate data sheet [8].

Back to torques: torques are more challenging to implement. There are two stages to them: finding the torque from a force at a position, then applying the resultant torque to the aircraft.

For the first stage, we can use the formula below to calculate the torque based on a force and a position:

$$\tau = r \times F \tag{2}$$

where

τ = torque, Nm

F = force, N,

r = position relative to the centre of gravity at which the force is applied, m

The direction of the cross product gives the axis of rotation, and the magnitude gives the magnitude of rotation.

The application of the torque to the aircraft is more challenging, as we have to use this equation:

$$\alpha = I^{-1}\tau \tag{3}$$

where

α = angular acceleration, rads^{-2},

I^{-1} = reciprocal moment of inertia,

τ = torque, Nm

The moment of inertia is a matrix which represents how resistant an object is to spinning around the x, y or z axes. We can calculate for each aircraft part's OBB, then use **Parallel Axis Theorem** to find the moment of inertia for the entire aircraft.

The moment of inertia can be calculated for each OBB by using the formula below:

$$I = \frac{m(a^2 + b^2)}{12} \tag{4}$$

where

I = moment of inertia about a chosen axis of the OBB,

a and b = the length of the edges which are not parallel to the chosen axis of the OBB, m

m = mass, kg

However, we want to find the moment of inertia for the entire aircraft, not just the individual OBBs. To do this, we can use **Parallel Axis Theorem**:

$$I = I_{\text{cm}} + md^2 \tag{5}$$

where

I = moment of inertia about a chosen axis of the entire aircraft,

I_{m} = moment of inertia about the chosen axis of the OBB which goes through its centre of gravity,

m = mass of the OBB, kg,

d = perpendicular distance between the two chosen axes, m,

We can use this to add up all of the moments of inertia for every OBB around every axis, and then form a matrix (or a vector, as I prefer) to represent the reciprocals of the moments of inertia for each axis, which we can use in Equation 3 to find the angular acceleration.

2.4.1 Motion

Once we have determined the acceleration, calculating the motion is simple with the correct equations:

$$v = u + at \tag{6}$$

where

v = final velocity, $ms^{-}1$,

u = initial velocity, $ms^{-}1$,

a = acceleration, ms^{-2},

t = time, s

$$s = ut + \frac{1}{2}at^2 \tag{7}$$

where

s = displacement, m,

u = initial velocity, ms^{-1},

a = acceleration, ms^{-2},

t = time, s

$$\omega = \omega_0 + \alpha t \tag{8}$$

where

ω = final angular velocity, $\text{rad}s^{-1}$,

ω_0 = initial angular velocity, $\text{rad}s^{-1}$,

α = angular acceleration, $\text{rad}s^{-2}$,

t = time, s

$$\theta = \omega_0 t + \frac{1}{2}\alpha t^2 \tag{9}$$

where

θ = angular displacement (the rotation of the object), rad,

ω_0 = initial angular velocity, $\text{rad}s^{-1}$,

α = angular acceleration, $\text{rad}s^{-2}$,

t = time, s

2.5 Aerodynamics

The fundamentals of aerodynamics and by extension aviation are commonly shown with a force diagram such as the following [9]:

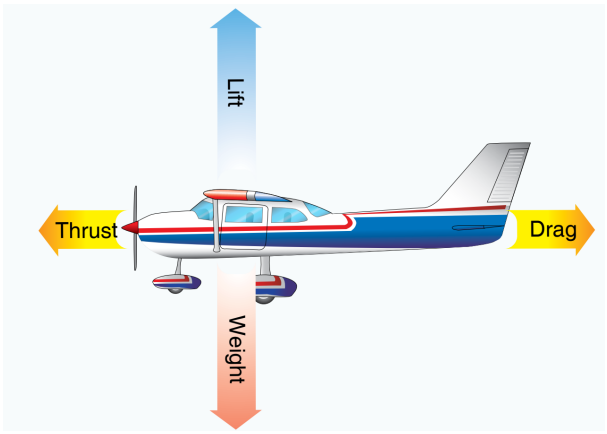


Figure 5: The forces which act on an aircraft

2.5.1 Thrust

Thrust is a force generated by the engine which acts in the direction that the aircraft is pointing. The component of the thrust which is controlled by the user input is known as the throttle. For the purposes of this simulation, the throttle will vary between 0 and 1. 0 means no engine power and 1 means maximum engine power, with every intermediate value being linearly interpolated. The engine output can be found by this equation:

$$P = tP_{max} \tag{10}$$

where

P = power produced by the engine, W,

t = throttle, 0 to 1,

P_{max} = maximum power produced by the engine.

The maximum power of the Lycoming AEIO-360-B1F/B engine which is used by the Grob Tutor is 130kW [10].

The equation for thrust is as follows [11]:

$$T = \frac{P\eta_{prop}}{v + 0.1} \tag{11}$$

where

T = thrust force, N,

P = power produced by the engine, W,

η_{prop} = propeller efficiency,

v = airspeed, ms^{-1}

The propeller efficiency typically ranges from 0.7 to 0.9 [12], so we will approximate it as 0.8. 0.1 has been included in the denominator to prevent zero division errors when the aircraft's speed is zero.

Equation (11) provides us with a value for thrust, which we can apply to the direction which the aircraft is facing to simulate thrust.

2.5.2 Lift

Lift is an aerodynamic force generated by different pressures above and below the wing, causing a net force. It is what allows the aircraft to fly, and acts perpendicularly to the direction that the aircraft is pointing.

Unlike the thrust, lift is generated by several different components, such as the two wings and the horizontal stabiliser. Therefore, I will need to calculate the lift at each different part of the aircraft.

The equation for lift is as follows[13]

$$L = \frac{1}{2}C_L\rho v^2 A \tag{12}$$

where

L = lift force, N,

C_L = lift coefficient,

ρ = air density, kgm^{-3},

v = airspeed, ms^{-1},

A = planform area, m^2

- **Lift Coefficient**
  The lift coefficient is an absolute pain to calculate, as detailed in the Aerodynamic Coefficients section on Page 16.
- **Air Density**
  I will use the value of $1.225kgm^{-3}$ for the air density at sea level [14].
- **Airspeed**
  I will calculate the relative airspeed of the aircraft part by combining translational and angular velocity.

- **Planform Area**

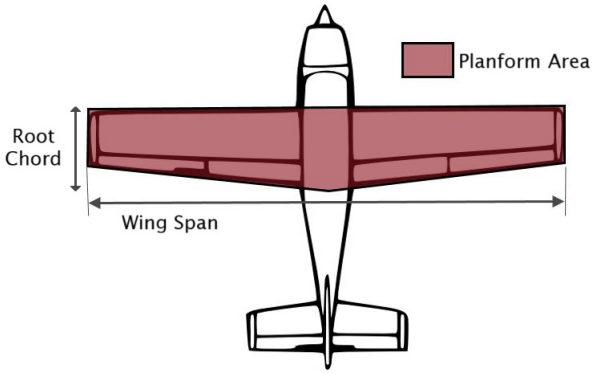


Figure 6: The planform area

I will approximate the planform area by multiplying the span by the root chord. I will get both of those values from the OBB of the aircraft part.

2.5.3 Weight

Weight is the resulting force of gravity acting on the mass of the aircraft. It can be easily calculated with this equation:

$$\mathbf{W} = m\mathbf{g} \tag{13}$$

where

$\mathbf{W}$ = Weight, N,

m = mass, kg,

$\mathbf{g}$ = acceleration due to gravity (9.81), ms^{-2}

It will always act downwards and does not change direction as a result of the aircraft.

2.5.4 Drag

Drag is the another aerodynamic force - similar to lift - which is the result of air resistance, and acts in the opposite direction to motion.

The equation for drag is as follows [15]:

$$D = \frac{1}{2}C_D\rho A v^2 \tag{14}$$

where

D = drag force, N,

C_D = drag coefficient,

ρ = air density kgm^{-3},

A = planform area, m^2,

v = airspeed, ms^{-1},

All of these values can be found in the same way as those involved in the lift equation.

2.5.5 Balancing Moment

There are three aerodynamic forces/moments I need to account for: lift, drag and the pitching moment. The moment is calculated as a balancing pitching moment created by the interaction between the aerodynamic surfaces and the air, and is found with the following equation [16]:

$$M = \frac{1}{2} C_M \rho A L v^2 \tag{15}$$

where

D = moment, Nm,

C_M = moment coefficient,

ρ = air density kgm^{-3},

A = planform area, m^2,

L = chord length, m,

v = airspeed, ms^{-1},

I will find the chord length from the bounding box of the aircraft component for which I am calculating the moment.

2.6 Aerodynamic Coefficients

There are three aerodynamic coefficients that I need to find: the lift coefficient (C_L, as seen in Equation (12)), the drag coefficient (C_D, as seen in Equation (14)) and the moment coefficient (C_M, as seen as in Equation (15)) .

These three coefficients are all functions of the angle of attack and control surface deflection (control surface deflection is how much part of the aircraft which can be moved by the pilot, such as the elevators, is rotated). Essentially, these aerodynamic coefficients incorporate the pilots actions and thereby give the pilot agency over the aircraft's motion. They are crucial to making an effective flight simulator, so I need a way of finding them accurately.

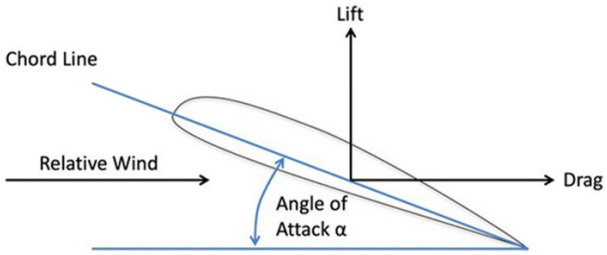


Figure 7: The angle of attack is the angle between the relative wind and the chord line of the aerodynamic surface [17]

2.6.1 Option 1: Lookup Tables

The easiest option is to precalculate the coefficients and store them in a table, which can be quickly accessed based on current parameters like angle of attack.

The program I downloaded to calculate these coefficients was JavaFoil [18], which did aerodynamic calculations based on a given airfoil (the cross-section of a wing or other aerodynamic surface).

I could not find sufficient data about the Eppler 696 airfoil [10] which the Grob Tutor's wings used, nor any of the other airfoils used by other aerodynamic surfaces. Therefore, I took a screenshot of the airfoil and recreated it in JavaFoil.



Figure 8: A screenshot of the airfoil, taken from the 3D model I am using

I ran the calculations and JavaFoil presented me with this lift coefficient graph:

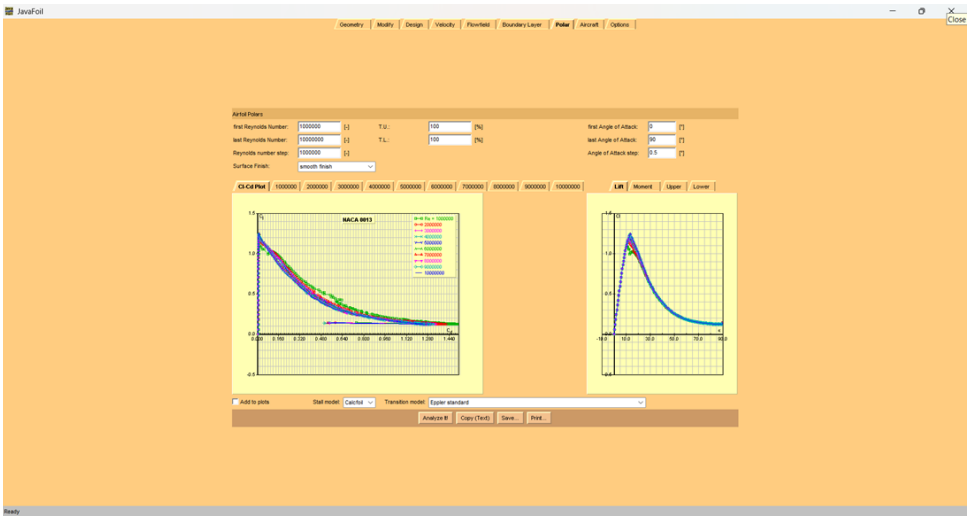
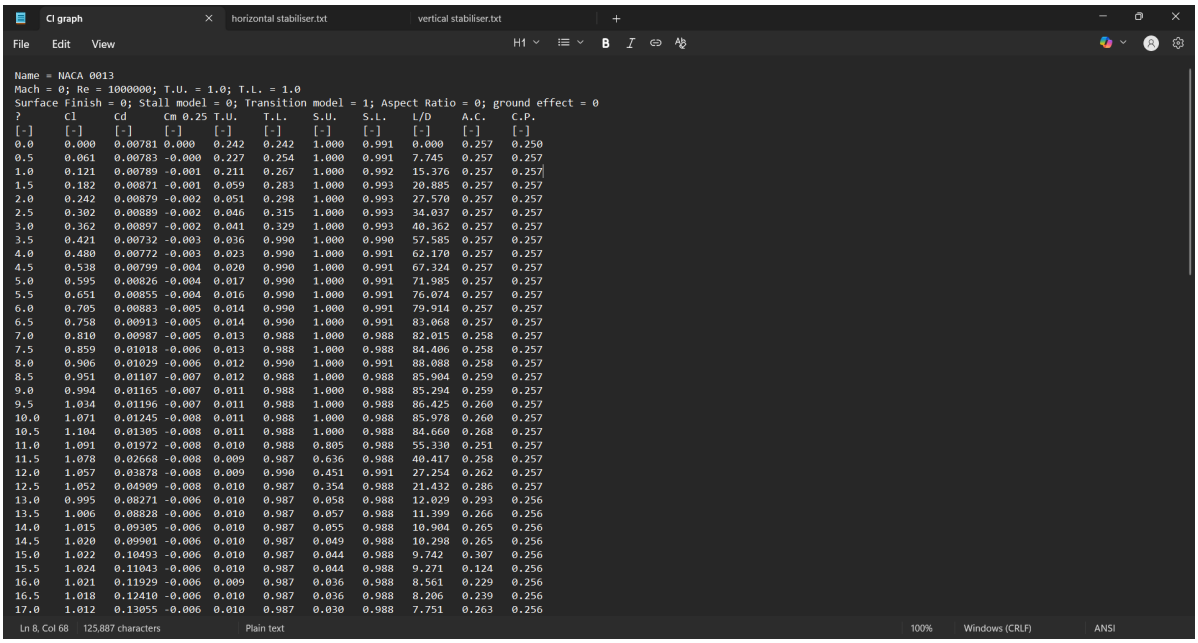


Figure 9: A screenshot of the lift to angle of attack curve (right) generated by JavaFoil

I exported that curve and other data such as drag and moment coefficient into a text file, which I could have the transferred into the SQL database that I had set up, if I had decided to proceed with this option.



Cl graph — horizontal stabiliser.txt — vertical stabiliser.txt

File Edit View

```
Name = NACA 0013
Mach = 0; Re = 1000000; T.U. = 1.0; T.L. = 1.0
Surface Finish = 0; Stall model = 0; Transition model = 1; Aspect Ratio = 0; ground effect = 0
?       Cl      Cd      Cm 0.25 T.U.    T.L.    S.U.    S.L.    L/D     A.C.    C.P.
[-]     [-]     [-]     [-]     [-]     [-]     [-]     [-]     [-]     [-]     [-]
0.0     0.000   0.00781 0.000   0.242   0.242   1.000   0.991   0.000   0.257   0.250
0.5     0.061   0.00783 -0.000  0.227   0.254   1.000   0.991   7.745   0.257   0.257
1.0     0.121   0.00789 -0.001  0.211   0.267   1.000   0.992   15.376  0.257   0.257
1.5     0.182   0.00871 -0.001  0.059   0.283   1.000   0.993   20.885  0.257   0.257
2.0     0.242   0.00879 -0.002  0.051   0.298   1.000   0.993   27.570  0.257   0.257
2.5     0.302   0.00889 -0.002  0.046   0.315   1.000   0.993   34.037  0.257   0.257
3.0     0.362   0.00897 -0.002  0.041   0.329   1.000   0.993   40.362  0.257   0.257
3.5     0.421   0.00732 -0.003  0.036   0.990   1.000   0.990   57.585  0.257   0.257
4.0     0.480   0.00772 -0.003  0.023   0.990   1.000   0.991   62.170  0.257   0.257
4.5     0.538   0.00799 -0.004  0.020   0.990   1.000   0.991   67.324  0.257   0.257
5.0     0.595   0.00826 -0.004  0.017   0.990   1.000   0.991   71.985  0.257   0.257
5.5     0.651   0.00855 -0.004  0.016   0.990   1.000   0.991   76.074  0.257   0.257
6.0     0.705   0.00883 -0.005  0.014   0.990   1.000   0.991   79.914  0.257   0.257
6.5     0.758   0.00913 -0.005  0.014   0.990   1.000   0.991   83.068  0.257   0.257
7.0     0.810   0.00987 -0.005  0.013   0.988   1.000   0.988   82.015  0.258   0.257
7.5     0.859   0.01018 -0.006  0.013   0.988   1.000   0.988   84.406  0.258   0.257
8.0     0.906   0.01029 -0.006  0.012   0.990   1.000   0.991   88.088  0.258   0.257
8.5     0.951   0.01107 -0.007  0.012   0.988   1.000   0.988   85.904  0.259   0.257
9.0     0.994   0.01165 -0.007  0.011   0.988   1.000   0.988   85.294  0.259   0.257
9.5     1.034   0.01196 -0.007  0.011   0.988   1.000   0.988   86.425  0.260   0.257
10.0    1.071   0.01245 -0.008  0.011   0.988   1.000   0.988   85.978  0.260   0.257
10.5    1.104   0.01305 -0.008  0.011   0.988   1.000   0.988   84.660  0.268   0.257
11.0    1.091   0.01972 -0.008  0.010   0.988   0.805   0.988   55.330  0.251   0.257
11.5    1.078   0.02668 -0.008  0.009   0.987   0.636   0.988   40.417  0.258   0.257
12.0    1.057   0.03878 -0.008  0.009   0.990   0.451   0.991   27.254  0.262   0.257
12.5    1.052   0.04909 -0.008  0.010   0.987   0.354   0.988   21.432  0.286   0.257
13.0    0.995   0.08271 -0.006  0.010   0.987   0.058   0.988   12.029  0.293   0.256
13.5    1.006   0.08828 -0.006  0.010   0.987   0.057   0.988   11.399  0.266   0.256
14.0    1.015   0.09305 -0.006  0.010   0.987   0.055   0.988   10.904  0.265   0.256
14.5    1.020   0.09901 -0.006  0.010   0.987   0.049   0.988   10.298  0.265   0.256
15.0    1.022   0.10493 -0.006  0.010   0.987   0.044   0.988   9.742   0.307   0.256
15.5    1.024   0.11043 -0.006  0.010   0.987   0.044   0.988   9.271   0.124   0.256
16.0    1.021   0.11929 -0.006  0.009   0.987   0.036   0.988   8.561   0.229   0.256
16.5    1.018   0.12410 -0.006  0.010   0.987   0.036   0.988   8.206   0.239   0.256
17.0    1.012   0.13055 -0.006  0.010   0.987   0.030   0.988   7.751   0.263   0.256
```

Ln 8, Col 68 125,887 characters Plain text 100% Windows (CRLF) ANSI

Figure 10: The data I exported from JavaFoil

2.6.2 Option 2: Real Time Calculation

There was one major flaw with using a lookup table: whilst they could accurately provide data for the coefficients based on angle of attack, the data that I had access to did not account for control surface deflection - the only way for the pilot to control the direction of the aircraft. This meant that I unfortunately had to abandon the lookup table plan and calculate the coefficients in real time.

I will divide the aircraft into different aerodynamic surfaces (e.g separating each wing into one part with the aileron and one part without), then calculate the aerodynamic coefficients for each aerodynamic surface.

To calculate the aerodynamic coefficients, I will heavily consult 'Real-Time Modeling of Agile Fixed-Wing UAV Aerodynamics' [19], a paper by Waqas Khan and Meyer Nahon. The paper primarily concerns

the modelling of highly manouverable unmanned aerial vehicles (remotely controlled aircraft), but includes the calculation of the aerodynamic coefficients in a way that applies to larger aircraft.

Equipped with this paper, all I need to do is find the specified aerodynamic parameters and use them in the given equations along with the angle of attack and control surface deflection to generate values for the three aerodynamic parameters which I seek. I made the diagram below to illustrate this process:

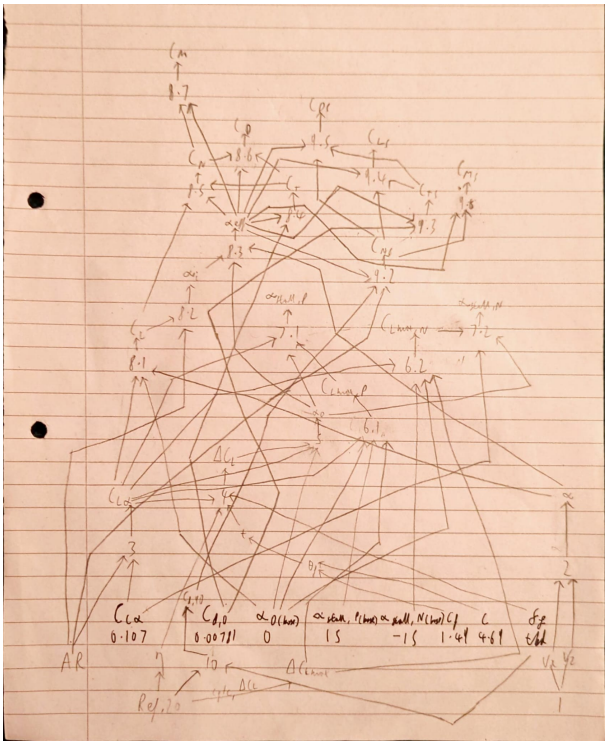


Figure 11: A diagram which shows how various aerodynamic parameters can be used to generate the lift, drag and moment coefficients

2.6.3 Aerodynamic Parameters

There are nine different aerodynamic parameters I need to find before I can start using the equations: angle of attack, control surface deflection, 2D lift curve slope, skin friction coefficient, zero lift angle of attack, positive and negative stall angle, flap to chord ratio and aspect ratio.

Angle of Attack

The equation for angle of attack given in the paper is:

$$\alpha = \tan^{-1}\left(\frac{V_y}{V_x}\right) \tag{16}$$

where

α = angle of attack, °,

V = relative velocity, ms^{-1}

Relative velocity is defined by the axes of the aircraft, as the following diagram shows:

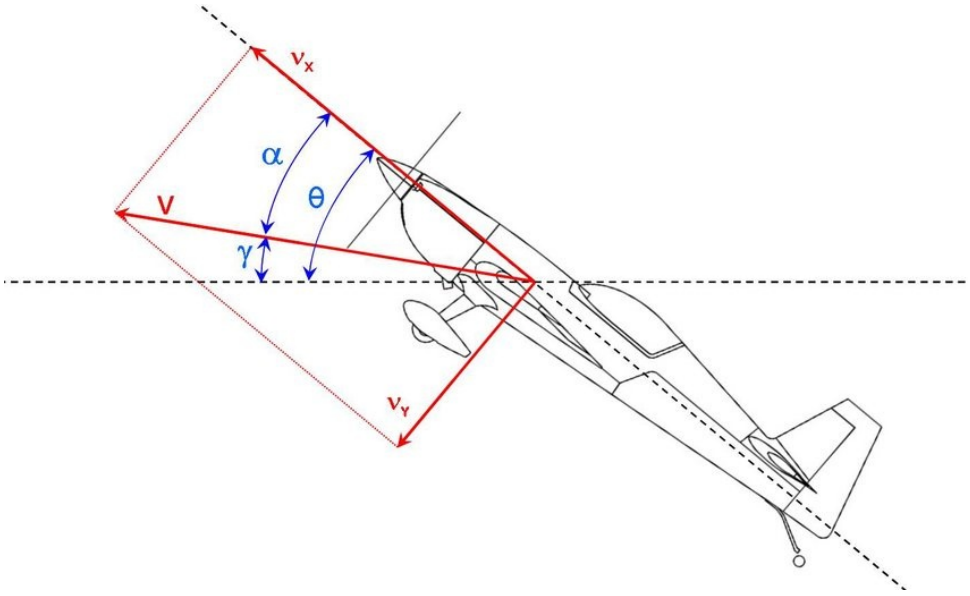


Figure 12: The angle of attack is calculated from relative velocity, not world velocity

Control Surface Deflection

This is one of the more straightforward of the parameters. I will just use the angle at which each control surface model is rotated as the value for the control surface deflection (δ_f).

2D Lift Curve Slope

As it turns out, JavaFoil will come in handy after all. The 2D lift curve slope ($C_{l\alpha}$)is the pre-stall lift coefficient gradient, which we can calculate from the data I collected from JavaFoil in Figure 10.

After gathering the data, I wrote the simple Python program below to calculate the gradient of the line of best fit for the data. For example, the wing had a 2D lift curve slope of 6.152033224533429 (converted to radians).

```
def mean(mean_list):
    return sum(mean_list) / len(mean_list)

def standard_deviation(sd_list):
    return math.sqrt(sum(map(lambda x: x ** 2, sd_list)) / len(sd_list) - mean(sd_list) ** 2)

def PMCC(x_list, y_list):
    x_mean = mean(x_list)
    x_sd = standard_deviation(x_list)
    y_mean = mean(y_list)
    y_sd = standard_deviation(y_list)
    total = 0
    for i in range(len(x_list)):
        total += ((x_list[i] - x_mean) / x_sd) * ((y_list[i] - y_mean) / y_sd)
    return total / len(x_list)

def gradient(correlation_PMCC, x_list, y_list):
    return correlation_PMCC * (standard_deviation(y_list) / standard_deviation(x_list))

def y_intersect(gradient, x_mean, y_mean):
    return y_mean - x_mean * gradient

def equation(x_list, y_list):
    x_mean = mean(x_list)
    y_mean = mean(y_list)
    correlation_PMCC = PMCC(x_list, y_list)
```

```
27      correlation_gradient = gradient(correlation_PMCC, x_list, y_list)
28      correlation_y_intersect = y_intersect(correlation_gradient, x_mean, y_mean)
29      return correlation_y_intersect, correlation_gradient, correlation_PMCC
```

Skin Friction Coefficient

The skin friction coefficient ($C_{d,0}$) is the drag coefficient at the angle of attack where no lift is produced. This can be found from the JavaFoil data in Figure 10 again: all you need to do is find the angle at which the lift coefficient is 0, then find the drag coefficient at that angle. The wing's skin friction coefficient is 0.00781.

Zero Lift Angle of Attack

The zero lift angle of attack ($\alpha_{0(\text{base})}$ is the angle at which the lift coefficient is 0, and therefore the total lift is too. It is based on the geometric shape of the airfoil. For symmetrical airfoils like ours, the zero lift angle of attack is generally 0. It can be found by looking for the angle of attack at which the lift coefficient is 0. For the wing, the zero lift angle of attack is 0 (as expected).

Positive and Negative Stall Angle

When an aerodynamic surface exceeds a certain critical value of angle of attack, the flow begins to separate, the lift coefficient decreases and the aircraft stalls. You can see this effect in Figure 9, as the lift curve slope increases, then it hits a maximum, and then it decreases. This is called the positive stall angle ($\alpha_{\text{stall,P(base)}}$) is generally around 15°. We can find this angle by determining the maximum lift coefficient, and then finding the corresponding angle. The wing's positive stall angle is 15°.

We can find the negative stall angle($\alpha_{\text{stall,N(base)}}$) by using the same strategy but for negative values. For a symmetrical airfoil, this is generally the same as the positive stall angle but negative. The negative stall angle of the wing is -15.

Flap to Chord Ratio

The flap to chord ratio (c_f/c) represents how much of the root chord length (see Figure 6 for root chord definition) is taken up by the control surface's length. I will calculate this in engine by dividing the length of the OBB of the control surface by the root chord length of the aerodynamic surface. The flap to chord ratio of the wing is $1.49050807147378 \div 4.690120788210841 = 0.31779737426386623$.

Aspect Ratio

Calculating the aspect ratio (A_R) is very similar to calculating the area (see Figure 6). However, we need to divide the span by the root chord rather than multiply them. The aspect ratio of the wings are 8.2, but we will actually use different values for the different aerodynamic surfaces which make up the wing, as they are of different sizes.

2.6.4 Aerodynamic Coefficient Calculations

And just like that, we have found all the aerodynamic parameters that we need in order to begin calculating the aerodynamic coefficients.

Most of the following equations are from Khan and Nahon's paper [19].

First, we need to find the lift curve slope for the aerodynamic surface with a finite aspect ratio ($C_{L\alpha}$, rather than with the infinite aspect ratio that $C_{l\alpha}$ represents.

$$C_{L\alpha} = C_{l\alpha}\left(\frac{A_R}{A_R + \frac{2(A_R+4)}{A_R+2}}\right) \tag{17}$$

Then we need to determine how much the lift coefficient changes due to the control surface deflection.

$$\theta_f = \cos^{-1}(2c_f/c - 1) \tag{18}$$

$$\tau = 1 - \frac{\theta_f - \sin\theta_f}{\pi} \tag{19}$$

$$\Delta C_L = C_{L\alpha}\tau\eta\delta_f \tag{20}$$

where

θ_f = the flap effectiveness angle, which determines the flap effectiveness due to the flap to chord ratio,

τ = the flap effectiveness angle, which directly determines how much of an impact the control surface has on the lift coefficient due to the flap effectiveness angle,

ΔC_L = the lift coefficient increment (how much the lift coefficient changes as a result of variation in the control surface deflection),

η = an additional factor to account for air viscosity

If the control surface changes, then the camber (the 'middle' of the airfoil) also changes:

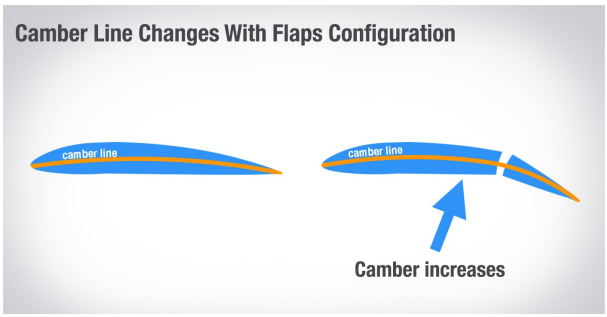


Figure 13: The camber is dependent on the control surface deflection

As previously mentioned, the zero lift angle of attack depends on the geometry of the airfoil. Because the geometry is changed by the control surface deflection, then the zero lift angle of attack must change too:

$$\alpha_0 = \alpha_{0(\text{base})} - \frac{\Delta C_L}{C_{L\alpha}} \tag{21}$$

where

α_0 = the zero lift angle of attack due to the shifted geometry of the airfoil caused by the control surface deflection,

$\alpha_{0(\text{base})}$ = the zero lift angle of attack due solely to the original geometry of the airfoil.

It is worth noting that, if we substitute the equation for the lift coefficient increment - Equation (20) - into Equation (22), we get this:

$$\alpha_0 = \alpha_{0(\text{base})} - \tau\eta\delta_f \tag{22}$$

which more clearly shows the link between the zero lift angle of attack and the control surface deflection.

The zero lift angle of attack is not the only angle that changes as a result of control surface deflection: the stall angles do too, as shown by this graph from Khan and Nahon's paper [19]:

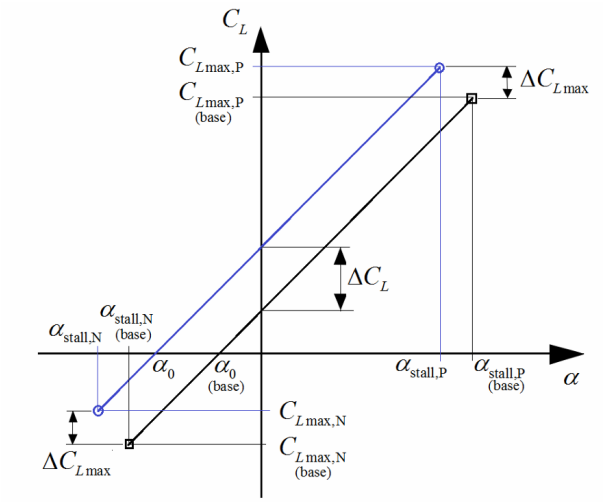


Figure 14: The effect of control surface deflection on the lift curve in the linear region. The black and blue lines represent no deflection, and positive deflection respectively.

We want to find the new stall angles from this graph, so we'll start by focusing on $\alpha_{\text{stall,P}}$. As the gradient of the graph is $C_{L\alpha}$ as we discussed earlier, we can use the classic $y - y_0 = m(x - x_0)$ equation for straight line graphs to form an equation for $\alpha_{\text{stall,P}}$:

$$C_{L\text{max,P}} - 0 = C_{L\alpha}(\alpha_{\text{stall,P}} - \alpha_0) \tag{23}$$

$$\alpha_{\text{stall,P}} - \alpha_0 = \frac{C_{L\text{max,P}}}{C_{L\alpha}} \tag{24}$$

$$\alpha_{\text{stall,P}} = \alpha_0 + \frac{C_{L\text{max,P}}}{C_{L\alpha}} \tag{25}$$

where

$C_{L\text{max,P}}$ = the new maximum lift coefficient, due to the shifted control surface,

$\alpha_{\text{stall,P}}$ = the new stall angle, due to the shifted control surface

The only one of these values that we do not know is $C_{L\text{max,P}}$, so we need to use the graph further to find an equation for it.

$$C_{L\text{max,P}} = C_{L\text{max,P(base)}} + \Delta C_{L\text{max}} \tag{26}$$

$$C_{L\text{max,P(base)}} = C_{L\text{max,P}} - \Delta C_{L\text{max}} \tag{27}$$

$$C_{L\text{max,P(base)}} - 0 = C_{L\alpha}\left(\alpha_{\text{stall,P(base)}} - \alpha_{0\text{(base)}}\right) \tag{28}$$

$$C_{L\text{max,P}} - \Delta C_{L\text{max}} = C_{L\alpha}\left(\alpha_{\text{stall,P(base)}} - \alpha_{0\text{(base)}}\right) \tag{29}$$

$$C_{L\text{max,P}} = C_{L\alpha}\left(\alpha_{\text{stall,P(base)}} - \alpha_{0\text{(base)}}\right) + \Delta C_{L\text{max}} \tag{30}$$

Now we have an equation for $C_{L\text{max,P}}$, except that we've taken one value that we don't know, and put it in terms of another value that we don't know: $\Delta C_{L\text{max}}$.

To find a value for $\Delta C_{L\text{max}}$, we must follow the guidance of the paper. It says

"$\Delta C_{L\text{max}}$ can be determined from [20] for a given ΔC_L (from Eq. (4)) and c_f/c ratio."

[20] refers to B. W. McCormick's 1995 book, 'Aerodynamics, Aeronautics, and Flight Mechanics' [20]. Deep within the many pages of that tome, I found this diagram:

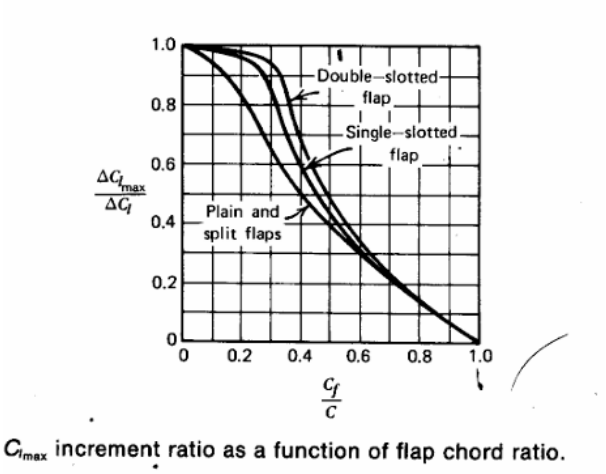


Figure 15: A graph of $\frac{\Delta C_{L\max}}{\Delta C_L}$ against c_f/c ratio.

This is *nearly* a linear graph; it's close enough for my standards, at least. If we interpret it as a linear graph, from $(0, 1)$ to $(1, 0)$, we can form the following equations:

$$\frac{\Delta C_{L\max}}{\Delta C_L} = 1 - c_f/c \tag{31}$$

$$\Delta C_{L\max} = \Delta C_L(1 - c_f/c) \tag{32}$$

This gives us our equation for $\Delta C_{L\max}$, which we can substitute into (30) which gives us $C_{L\max,\mathrm{P}}$, which we can in turn substitute into (25) to find $\alpha_{\mathrm{stall,P}}$.

We can do the same process but for $\alpha_{\mathrm{stall,N}}$, and we get similar equations:

$$C_{\mathrm{Lmax,\ N}} = C_{L\alpha}\left(\alpha_{\mathrm{stall,N(base)}} - \alpha_{0(\mathrm{base})}\right) + \Delta C_{L\max} \tag{33}$$

$$\alpha_{\mathrm{stall,N}} = \alpha_0 + \frac{C_{\mathrm{Lmax,N}}}{C_{L\alpha}} \tag{34}$$

Now we arrive at a fork in the road: stall. The following equations are dependent on whether the angle of attack is pre or post-stall.

Pre-stall

If $\alpha_{\mathrm{stall,N}} \leq \alpha \leq \alpha_{\mathrm{stall,P}}$ (α is the actual angle of attack which we calculated all the way back in Equation (16) on Page 18), then the aerodynamic surface is pre-stall and we use the following equations:

We can use $y - y_0 = m(x - x_0)$ again to find the lift coefficient, which is linear pre-stall:

$$C_L = C_{L\alpha}(\alpha - \alpha_0) \tag{35}$$

According to the paper,

> "a finite surface induces a downwash reducing the angle of attack by α_i, called the induced angle of attack."

We therefore need to account for this by adjusting the angle of attack by a_i. Firstly, we need to calculate a_i with the equation that they provide:

$$\alpha_i = \frac{C_L}{\pi A_R} \tag{36}$$

Then we need to adjust the angle of attack by a_i. Given that we always use the current angle of attack relative to the zero lift angle of attack α_0, we might as well create a new angle, an 'effective' angle of attack (α_{eff}) which accounts for both α_0 and α_i:

$$\alpha_{\mathrm{eff}} = \alpha - \alpha_0 - \alpha_i \tag{37}$$

We can now use α_{eff} whenever we need do any calculations with the angle of attack. Before we can take the final steps to calculate the drag and moment coefficient, we need to calculate two other coefficients: the tangential and normal force coefficients (C_T & C_N).

$$C_T = C_{d,0} \cos \alpha_{\text{eff}} \tag{38}$$

The tangential force coefficient (C_T) is the coefficient of the tangential force which acts along the airfoil. It depends on the skin friction coefficient $C_{d,0}$, which represents the friction force experienced by the air as it travels past the airfoil.

We can now use C_T to calculate the normal force coefficient.

$$C_N = \frac{C_L + C_T \sin \alpha_{\text{eff}}}{\cos \alpha_{\text{eff}}} \tag{39}$$

The normal force coefficient consists of the components of the lift coefficient and tangential force coefficient which are in the direction of the normal of the airfoil.

Now we can use C_T and C_N in Equations (45) and (46) to calculate the drag and moment coefficients, but first, we should consider the other scenario: the airfoil is beyond stall.

Post-stall

If $\alpha < \alpha_{\text{stall,N}}$ or $a > \alpha_{\text{stall,P}}$, then the aerodynamic surface is post-stall, and we use the following equations.

The first thing to acknowledge is that there is no equation for the induced angle of attack (a_i) beyond stall. Instead, a_i is linearly tapered to 0 at 90°. Therefore, the value for a_i can be linearly interpolated between its maximum value in Equation (36) and 0 at 90°.

Once we have found the value for a_i, we will once again calculate an effective angle of attack:

$$\alpha_{\text{eff}} = \alpha - \alpha_0 - \alpha_i \tag{40}$$

Now, before we can jump into another equation, we need to find the '2D drag coefficient of the airfoil normal to the flow' ($C_{d,90}$ - in other words, the skin friction coefficient when the airfoil is perpendicular to the airfoil, rather than parallel to it.

This is calculated from data from 'Aerodynamics, Aeronautics, and Flight Mechanics' [20], which was curve fitted and linearised into the following equation:

$$C_{d,90} = -4.26 \times 10^{-2} \delta_f^2 + 2.1 \times 10^{-1} \delta_f + 1.98 \tag{41}$$

We now have all that we need to find the normal force coefficient again, but the equation is very different post stall:

$$C_N = C_{d,90} \sin \alpha_{\text{eff}} \left[\frac{1}{0.56 + 0.44 \sin \alpha_{\text{eff}}} - 0.41 \left(1 - e^{\frac{-17}{AR}} \right) \right] \tag{42}$$

You can see that it is proportional to the $C_{d,90} \sin \alpha_{\text{eff}}$, which approaches $C_{d,90}$ as α_{eff} approaches 90°.

We can use C_N to find the post-stall tangential force coefficient, C_T:

$$C_T = 0.5 C_{d,0} \cos \alpha_{\text{eff}} \tag{43}$$

which approaches the value of half of the skin friction coefficient as α_{eff} approaches 0°

Finally, we can calculate the lift coefficient using the other force coefficients that we have calculated so far:

$$C_L = C_N \sin \alpha_{\text{eff}} - C_T \sin \alpha_{\text{eff}} \tag{44}$$

Drag and Moment Coefficients

At last, we have everything we need to calculate the drag and balancing pitch moment coefficient. The equations are both identical either pre or post-stall.

$$C_D = C_N \sin \alpha_{\text{eff}} + C_T \cos \alpha_{\text{eff}} \tag{45}$$

$$C_M = -C_N \left[0.25 - 0.175 \left(1 - \frac{2\alpha_{\text{eff}}}{\pi} \right) \right] \tag{46}$$

And with that, we have found the lift, drag and moment coefficients which Equations (12), (14), (**??**)

2.7 Delta Time

My project will have two issues that need solving which are both related to time:

Issue 1 - Physics calculations
Many equations I was planning to use, for example, $s = vt$ (speed = distance × time), required a value for time, and finding this value was not immediately obvious to me: if I assumed it to be equal to 1, then I would suffer from the second problem:

Issue 2 - Frame Rate Dependence
Throughout the duration of the program, the frame rate is likely to fluctuate. Additionally, as this program is intended for cadets to use at home, the power of the user's device will vary, leading to different frame rates.

The issue that different frame rates cause is that the main loop will run as many times as the frame rate allows. If we assume t to be 1, as suggested earlier, then we would be using values in metres per second every frame. Consequently, the displayed velocity would be dependent on the frame rate. The following equation expresses the problem:

$$v_f = u \tag{47}$$
$$\therefore v = \psi u \tag{48}$$

where

v_f = displayed velocity, mf^{-1},

v = displayed velocity, ms^{-1},

ψ = frame rate, fs^{-1},

u = intended velocity, ms^{-1}

Solution - Delta Values
Ideally, the solution to this would solve both issues simultaneously; it would provide a value for t during physics calculations and would counter the issue of frame rate dependence.

To solve the latter problem, we need to convert the intended velocity from metres per second to metres per frame. This can be achieved by dividing the intended velocity by the frame rate, as shown below:

$$v = \psi \frac{u}{\psi} \tag{49}$$
$$\therefore v = u \tag{50}$$

As ψ is measured in frames per second, it follows that the reciprocal $\frac{1}{\psi}$ would be measured in seconds per frame. This can be rewritten as time elapsed since the previous frame, which is conventionally referred to as **Delta Time (Δt)**, and can be found by finding the time difference between the previous two frames.

$$\Delta t = t_{f_{n-1}} - t_{f_{n-2}} \tag{51}$$

where

Δt = delta time, s

$t_{f_{n-1}}$ = time between the start of the program's execution and last frame, s

$t_{f_{n-2}}$ = time between the start of the program's execution and the second-to-last frame, s

The final equation is therefore as follows:

$$v = \psi \Delta t u \tag{52}$$
$$\therefore v = u \tag{53}$$

And, applying delta time to Equation 47, we get this:

$$v_f = \Delta t u \tag{54}$$

This solves the issue of frame rate independence, and, because delta time represents the time interval between each frame, provides a value of t to be used in physics calculations.

Implementation

The conclusion of this is that, every frame, I should multiply any value which is measured per second by delta time, so they accurately reflect the current frame rate, providing consistent movement, and frame rate independence.

Due to my use of separate files for different parts of the program, this is unfortunately not quite as simple as creating a variable which I update every frame. Instead, I will create a **delta_time** object which has a **delta_time** attribute which can be used in necessary calculations, and an **update** method to recalculate **delta_time**. This allows delta time to be up to date whenever it is used by referencing an object instead of a variable.

2.8 Bounding Boxes

2.8.1 Axis Aligned vs Oriented Bounding Boxes

According to SketchFab[1], the website whence I sourced the model, there are 69,500 triangles in the model I am using. Therefore, it is not reasonable to attempt collision detection for every triangle in the model. Instead, we must approximate the shape of the model with a group of shapes. Conventionally, this is done with cuboids which completely encase the geometry of the model, known as Bounding Boxes, of which there are two types.

Axis Aligned Bounding Boxes

Axis Aligned Bounding Boxes (AABBs) are bounding boxes which are strictly defined by the 3 axes of the 3D space. Each of their edges is aligned with one of these axes.

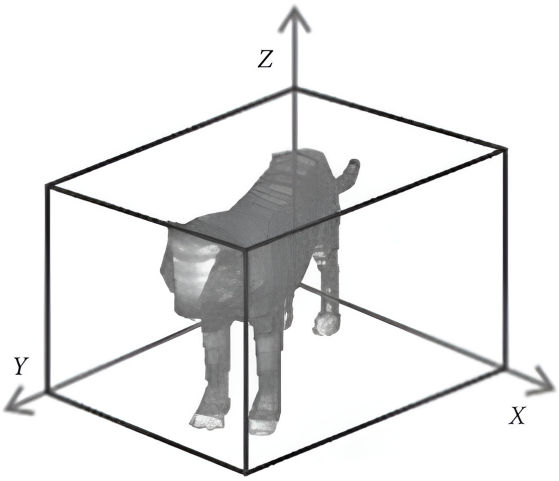


Figure 16: The Axis Aligned Bounding Box of a 3D model

This allows collision to be detected simply with another AABB, because, if any of the vertices of one AABB have coordinates between the x, y and z ranges of the other, then they must be intersecting. In fact, Three.js natively supports intersection testing between two box objects using this, so it would be trivial to implement.

However, AABBs do have shortcomings, most prominently, inaccuracy. As seen in Figure 16, the AABB does not provide a tight fit for the model. This inaccuracy greatly increases the likelihood of a false-positive intersection result, as there is a lot of space where an object could be which would be within the bounds of the AABB, but not actually touching the object.

AABBs are an especially inappropriate choice for this project, as aircraft are very manoeuvrable and tend to rotate a lot. However, as previously mentioned, AABBs must be aligned with the world axes, so the size of the AABB will vary significantly as the aircraft rotates, despite the aircraft remaining the same size. This example exacerbates the inaccuracy issue, and convinced me to select the other type of bounding boxes.

Oriented Bounding Boxes

Oriented Bounding Boxes (OBBs) are bounding boxes which are not strictly defined by any axis. They can adopt whichever orientation suits the model best.

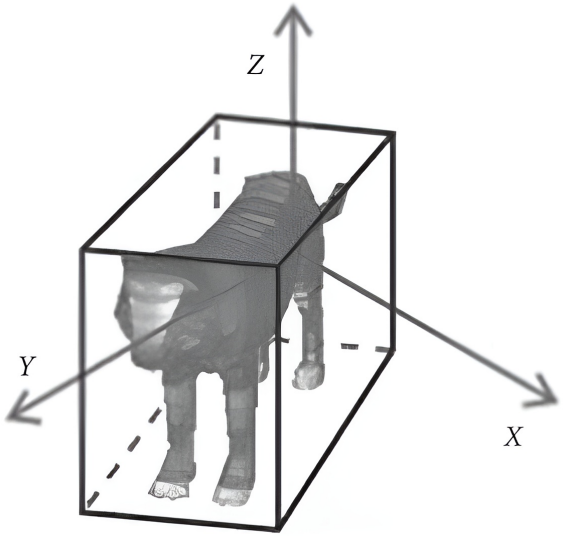


Figure 17: The Oriented Bounding Box of a 3D model

The above OBB clearly represent the model better and provides a tighter fit than the AABB did in figure 16, leading to more accurate collision checks.

But going further, a key benefit of OBBs is not just that they are rotated, but that they can be further rotated as necessary. This is ideal for the aircraft model, as the size of the OBB will remain constant regardless of the orientation of aircraft. The below comparison clearly demonstrates this difference.



Figure 18: **Left**: The AABB of the right wing. **Right**: The OBB of the right wing

Unfortunately, this OBBs also come with a downside: their collision tests are far more complicated. We can no longer just test if the vertices lie within the bounds of the other OBB when doing OBB on OBB collision checks, as it's very possible that two edges of the two OBBs intersect, with no vertices involved. Therefore, we need to be a little bit smarter about how we go about intersection tests. But that is a discussion for another time. Now we need to figure out how we're going to make our OBBs.

2.9 Making OBBs

2.9.1 From AABBs

The simplest way to make an OBB is to base it on an AABB and rotate it as necessary. This is the strategy that I will use to create the bounding boxes for the aircraft parts. Because the aircraft has been divided into smaller parts, the AABB is an appropriate representation of the part.

Once we have found the centre and extents of the AABB for each part, we can create an OBB based on it, and update the position and rotation as necessary.

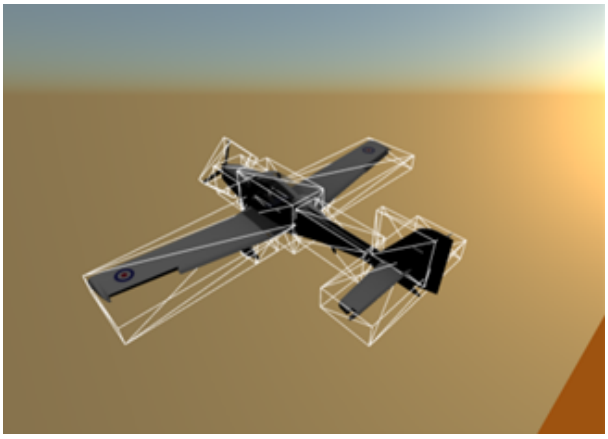


Figure 19: These OBBs were made from AABBs and updated to follow the aircraft

2.9.2 Complex OBB creation

As I explained in the binary tree section on Page 10, it is inefficient to check every OBB or triangle in the scene for collision with each other. Instead, if we could generate OBBs for the collision objects, then we would only need to check for the the collision between the overarching OBB of the aircraft and the overarching OBB of the terrain. If they intersect, then we could check for intersection between the four children of the two trees, and continue to do this until we find or exhaust the possibilities of a collision.

However, we need an algorithm which generates OBBs for collision objects. Ideally, this will be polymorphic, accepting either triangles or OBBs, so it can be used for either the aircraft's OBBs or the terrain's triangles.

My research led me to David Eberly's brilliant 1999 paper 'Dynamic Collision Detection using Oriented Bounding Boxes' [21], which provided an effective, if difficult to decipher, solution. He suggested three ways of creating OBBs, two of which interested me: creating an OBB from triangles and creating an OBB from points (this could be the vertices of an OBB or of a triangle). Both of these methods follow the same steps, which he vaguely alluded to:

1. Find the centre of the objects.
2. Create a covariance matrix out of the vectors from the centre to the objects.
3. Perform eigendecomposition on the covariance matrix to find the eigenvalues and more importantly the eigenvectors which we can use as axes.
4. Calculate the centre and extents of the OBB.
5. Create the new OBB.

2.9.3 Finding the centre

This works by summing all of the coordinates of the points and dividing the resulting vector by the number of points - fairly standard. The only thing of note is that triangles work slightly differently to points: the centres of the triangles are used, and they are multiplied by the area of their respective triangle. This provides some weighting to the OBB, so that a massive triangle has priority over a small one. This is the main benefit of using the OBB creation algorithm specifically for triangles.

2.9.4 Covariance Matrix

Simply put, a covariance matrix describes how variables relate to each other. We can use a covariance matrix to represent the spread of points from the previously worked out centre and their relations in regard to the world axes. Fortunately, forming the covariance matrix is easier done than said:

$$\text{covariance matrix} = \begin{bmatrix} \mathbf{d}_x\mathbf{d}_x & \mathbf{d}_y\mathbf{d}_x & \mathbf{d}_z\mathbf{d}_x \\ \mathbf{d}_x\mathbf{d}_y & \mathbf{d}_y\mathbf{d}_y & \mathbf{d}_z\mathbf{d}_y \\ \mathbf{d}_x\mathbf{d}_z & \mathbf{d}_y\mathbf{d}_z & \mathbf{d}_z\mathbf{d}_z \end{bmatrix} \tag{55}$$

where

$\mathbf{d}$ is the vector from the global centre to the object

All we now need to do is add up all of the covariance matrices for the objects, and then we arrive at the final covariance matrix.

2.9.5 Eigendecomposition

Now comes the difficult part. This whole process relies upon the idea that the optimal axes for the OBB to have are the eigenvectors of the covariance matrix we just found. The definition of an eigenvector of a matrix is a vector which remains on its span (in the same direction) after being transformed by the matrix. For a detailed explanation of them, I highly recommend 3Blue1Brown's video on the topic [22].

The equation for eigenvectors is as follows:

$$A\vec{v} = \lambda\vec{v} \tag{56}$$

$$\tag{57}$$

Essentially, if we apply the matrix A to vector , the vector remains pointing the same direction and is instead scaled by λ.

$$A\vec{v} = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \vec{v} \tag{58}$$

$$A\vec{v} = \lambda I\vec{v} \tag{59}$$

$$A\vec{v} - \lambda I\vec{v} = \mathbf{0} \tag{60}$$

$$\vec{v}(A - \lambda I) = \mathbf{0} \tag{61}$$

The result of this is that $\vec{v}$, when transformed by the matrix $(A - \lambda I)$, is a zero vector. Because we want to find eigenvectors, we assume $\vec{v}$ to not be a zero vector.

And in that case, the only way for the product of a matrix and a non-zero vector to be zero is if the determinant of the matrix is equal to 0.

$$\therefore \det(A - \lambda I) = \mathbf{0} \tag{62}$$

where

A is the transformation matrix (in our case, the covariance matrix),

$\vec{v}$ is the eigenvector,

λ is the eigenvalue, the value by which the eigenvector is scaled,

I is the identity matrix

Now we need to solve for λ, the eigenvalues, from which we can find the eigenvectors. We can do this by combining the matrices into one and finding its determinant

$$\text{Let } A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{63}$$

$$A - \lambda I = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \tag{64}$$

$$= \begin{bmatrix} a-\lambda & b & c \\ d & e-\lambda & f \\ g & h & i-\lambda \end{bmatrix} \tag{65}$$

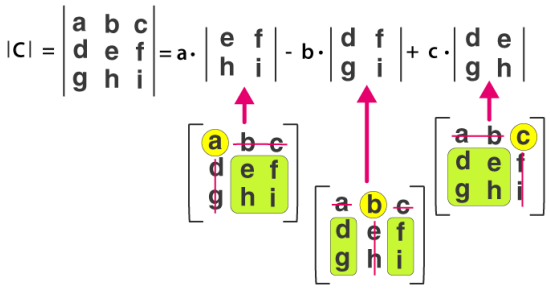


Figure 20: A demonstration of how the determinant of a 3×3 matrix is found

$$\begin{aligned}
\det(A-\lambda I) &= \det\left(\begin{bmatrix} a-\lambda & b & c \\ d & e-\lambda & f \\ g & h & i-\lambda \end{bmatrix}\right) \\
&= (a-\lambda)\cdot\begin{vmatrix} e-\lambda & f \\ h & i-\lambda \end{vmatrix} - b\cdot\begin{vmatrix} d & f \\ g & i-\lambda \end{vmatrix} + c\cdot\begin{vmatrix} d & e-\lambda \\ g & h \end{vmatrix} \\
&= (a-\lambda)((e-\lambda)(i-\lambda)-fh) - b(d(i-\lambda)-fg) + c(dh-(e-\lambda)g) \\
&= -\lambda^3 \\
&+ (a+e+i)\lambda^2 \\
&+ (-ae-ai-ei+fh+bd+cg)\lambda \\
&+ (aei-afh-bdi+bfg+cdh-cge) \\
&= 0
\end{aligned}$$

Because we have previously figured out what a-i are from the covariance matrix, what remains is a cubic equation.

2.9.6 Solving the cubic equation

There are several potential ways of solving cubic equations. I initially tried to use the cubic formula, but instead opted to depress the cubic and subsequently use Cardano's formula.

2.9.7 Depressing Cubics

Cardano's formula provides a simple way of solving cubic equations, but it has one large caveat: it only works on depressed cubics. This means cubics which don't have an x^2 term. Fortunately, any cubic can be converted into a depressed cubic.

If we have a cubic in the form $ax^3 + bx^2 + cx + d = 0$, then we can substitute $x = y - \frac{b}{3a}$ to remove

the x^2 term.

$$a(y-\frac{b}{3a})^3+b(y-\frac{b}{3a})^2+c(y-\frac{b}{3a})+d=0 \tag{66}$$

$$ay^3-by^2+\frac{b^2}{3a}y-\frac{b^3}{27a^2}+by^2-\frac{-2b^2}{3a}y+\frac{b^3}{9a^2}+\frac{b^3}{9a^2}+cy-\frac{bc}{3a}+d=0 \tag{67}$$

$$ay^3+\left(c-\frac{b^2}{3a}\right)y+\left(d+\frac{2b^3}{27a^2}-\frac{bc}{3a}\right)=0 \tag{68}$$

$$y^3+\left(\frac{-b^2+3ca}{3a^2}\right)y+\left(\frac{2b^3-9abc+27a^2d}{27a^3}\right)=0 \tag{69}$$

$$y^3+py+q=0 \tag{70}$$

2.9.8 Cardano's formula

Cardano's formula uses a cubic in the form $y^3+py+q=0$, and is as follows:

$$y=\sqrt[3]{-\frac{q}{2}+\sqrt{\frac{q^2}{4}+\frac{p^3}{27}}}+\sqrt[3]{-\frac{q}{2}-\sqrt{\frac{q^2}{4}+\frac{p^3}{27}}} \tag{71}$$

However, the 'discriminant' in the square root can be negative. This results in an imaginary number, which we will add to or subtract from $-\frac{q}{2}$, creating a complex number with a real and imaginary component (I will represent complex numbers as a `Vector2`). Then, we need to cube root that potentially complex number add it to the other cube root of a potentially complex number to get y.

2.9.9 Complex Cube Root

A complex number might look something like this:

$$a+b\mathbf{i} \tag{72}$$

where

$\mathbf{i}=\sqrt{-1}$

We can convert the complex number into polar form like this:

$$a+b\mathbf{i}=r(\cos\theta+i\sin\theta) \tag{73}$$

where

$r=\sqrt{a^2+b^2}$

$\theta=\tan(\frac{b}{a})$

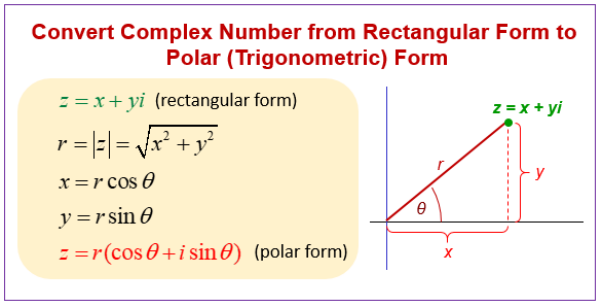


Figure 21: Conversion from the rectangular to polar complex form [23]

Now we can use De Moivre's Theorem to find the cube root like so.

$$r(\cos\theta+\mathbf{i}\sin\theta)^{\frac{1}{3}}=r^{\frac{1}{3}}\left(\cos\frac{\theta}{3}+\mathbf{i}\cos\frac{\theta}{3}\right) \tag{74}$$

$$=r^{\frac{1}{3}}\cos\frac{\theta}{3}+r^{\frac{1}{3}}\cos\frac{\theta}{3}\mathbf{i} \tag{75}$$

The final form of this equation is back in rectangular form, which allows us to put it back into a `Vector2`.

However, there are always 3 cube roots for a number, and this only provides one. We can find the others by adding 2π radians to θ, which essentially rotates all the way round so it is the pointing in the same direction it was initially. If we add 2π again, we get our final cube root.

2.9.10 Back to Cardano's formula

Now we have found the cube roots of the potentially complex $-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}$, we can substitute one of them into Cardano's formula. However, that only gives us one answer, so you can either substitute the others in or do what I did and multiply the first and second cube root by one of the cube roots of unity. These are the cube roots of 1, so 1, $-0.5 + \frac{\sqrt{3}}{2}\mathbf{i}$ and -0.5 + $\frac{-\sqrt{3}}{2}\mathbf{i}$ are all cube roots of unity. Multiplying with these finally gives us our three solutions to the cubic equation, which we will use as our eigenvalues.

2.9.11 Finding Eigenvectors

To recap, we have found the values of λ for which the determinant of $A - \lambda I = 0$. Now we need to find the values of x, y, and z for which $\vec{v}(A - \lambda I) = \mathbf{0}$.

where

$$\vec{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\text{Let } \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} = \begin{bmatrix} a-\lambda & b & c \\ d & e-\lambda & f \\ g & h & i-\lambda \end{bmatrix} \tag{76}$$

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{77}$$

$$Ax + By + Cz = 0 \tag{78}$$

$$Dx + Ey + Fz = 0 \tag{79}$$

$$Gx + Hy + Iz = 0 \tag{80}$$

Now, this might look simple to solve, but they way in which we formed them means that we can't 'solve' these equations per se; after all, there are technically an infinite number of eigenvectors, each with a different length but in the same direction. Therefore, there are an infinite number of possible valid values for x, y and z. We just need to find one valid vector and normalise it for each of the three eigenvectors.

Therefore we can choose one of x, y and z to equal a value of our choice (for simplicity, I chose 1). This gives us three simultaneous equations (I will only use the first two) for two variables which each have a single solution.

$$\text{Let } x = 1 \tag{81}$$

$$By + Cz = -A \tag{82}$$

$$Ey + Fz = -D \tag{83}$$

$$z = \frac{-A - By}{C} \tag{84}$$

$$Ey + F\frac{-A - By}{C} = -D \tag{85}$$

$$CEy - BFy - AF = -CD \tag{86}$$

$$(CE - BF)y = AF - CD \tag{87}$$

$$y = \frac{AF - CD}{CE - BF} \tag{88}$$

$$\tag{89}$$

We can substitute the value for y into (84) to get a value for z.

However, we cannot always assume that $x = 1$. We can choose whatever number we want and still get a parallel vector in every circumstance except one: $x = 0$. But overcoming this issue is not as simple as merely substituting in $x = 0$.

$$\text{Let } x = 0 \tag{90}$$
$$y = \frac{AFx - CDx}{CE - BF} \tag{91}$$
$$y = 0 \tag{92}$$
$$z = \frac{-A - By}{C} \tag{93}$$
$$z = 0 \tag{94}$$

This gives us a zero vector, which as previously mentioned is not what we want. Instead, we can let y different variable equal one.

$$\text{Let } y = 1 \tag{95}$$
$$Ax + Cz = -B \tag{96}$$
$$Dx + Fz = -E \tag{97}$$
$$z = \frac{-B - Ax}{C} \tag{98}$$
$$Dx + F\frac{-B - Ax}{C} = -E \tag{99}$$
$$CDx - AFx - BF = -CE \tag{100}$$
$$(CD - AF)x = BF - CE \tag{101}$$
$$x = \frac{BF - CE}{CD - AF} \tag{102}$$

So if x = 0, we can use the above equations instead. However, there is one more possibility: both the x and y values are 0, and the eigenvector is [0, 0, 1]. This can probably be accounted for more simply, but for consistency, I completed the trio of equations:

$$\text{Let } z = 1 \tag{103}$$
$$Ax + By = -C \tag{104}$$
$$Dx + Ey = -F \tag{105}$$
$$y = \frac{-C - Ax}{B} \tag{106}$$
$$Dx + E\frac{-C - Ax}{B} = -F \tag{107}$$
$$BDx - AEx - CE = -BF \tag{108}$$
$$(BD - AE)x = CE - BF \tag{109}$$
$$x = \frac{CE - BF}{BD - AE} \tag{110}$$

We can now set either x, y or z as 1, and find our eigenvector. We need to do this twice more for the other eigenvalues (as the cubic equation gives us three values for λ) using the same method, and then we will have found the axes of the OBB. Because we eigendecomposed a symmetric covariance matrix, the resulting vectors are orthogonal, we all we need to do is normalise the eigenvectors to find the axes.

2.9.12 Finding the dimensions of the OBB

Now we have the axes, we can find the dimensions of the box by projecting all the vertices of the objects, whether they be OBBs or triangles, onto each axis, and recording the most extreme result. Because the axes are normalised, we can simply project using the dot product between the axis and the vector of each vertex relative to the centre of the vertices.

After finding the range of points on each axis, we can find the extents along each axis by halving the distance between the most extreme projections for each axis, and we can find the centre by taking the average between the two extremes for each axis. However, doesn't actually give us the centre: it gives us

the centre in terms of the eigenvalue axes, relative to the previous centre that we calculated. To find the actual centre, we must convert it so it is in terms of the world axes, and then add the previous centre.

Now we have found the centre, the axes and the extents of the OBB, we can instantiate and new `OBB` object, which accurately represents the spread of the points it covers.

2.10 Intersection testing

2.10.1 Intersection Testing and Separating Axis Theorem

One of the disadvantages of using OBBs instead of AABBs is the additional complexity which comes with the intersection testing.

Instead of doing simple bounds testing, we need to use a more intricate collision detection algorithm. The basis for OBB intersection testing is the following quote:

> "Two nonintersecting convex polyhedra can be separated by a plane that is either parallel to a face of one of the polyhedra or that contains an edge from each of the polyhedra." [21]

This is called the **Separating Axis Theorem**, and the idea is that if you can fit a plane (imagine a sheet of paper) between two objects and the plane separates the two objects then they do not intersect. This seems fairly obvious, but we can use this to perform intersection testing between various objects, such as two OBBs or an OBB and a triangle.

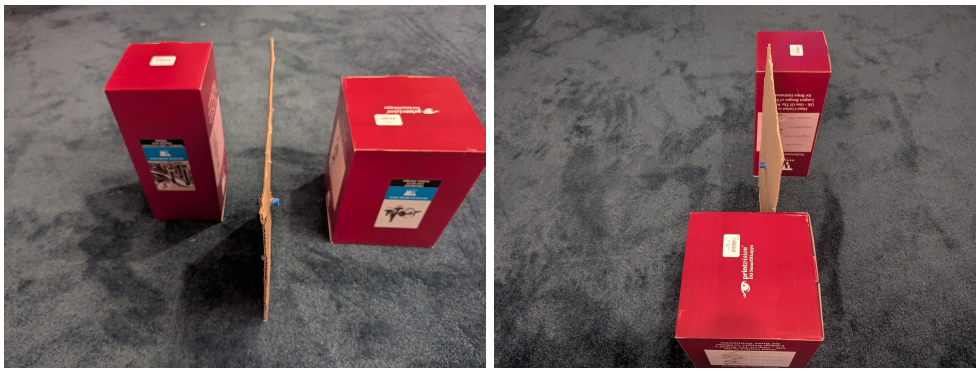


Figure 22: The left plane separates the boxes into two distinct sections, whereas the right plane does not

To find out if the plane separates the boxes, we will do something akin to testing for intersection with spheres: if the distance between centres of the spheres is greater than the sum of the two radii, then the spheres do not intersect.

In the context of OBBs, we can project the points onto the axis of the plane's normal. If the distance between the projected centres of the OBBs is greater than the sum of the two maximum projected points of the OBB, then the OBBs do not intersect in that axis, the plane can separate the OBBs and the OBBs do not intersect each other.

The axes that we need to check depends on the objects for which we are doing intersection tests. This process can be optimised by stopping the testing after a separating axis has been found. But if two objects do intersect, we need to exhaust every possible axis that could separate them before we can stop.

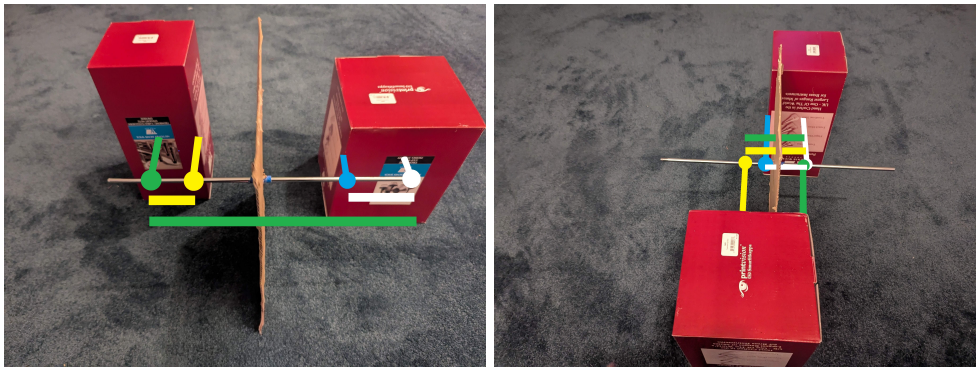


Figure 23: On the left plane, the sum of the maximum extents of the boxes projected to the plane's normal (the yellow and white line) is less than the distance between the centres of the boxes (the green line). On the right plane, the sum of the maximum extents is larger than the distance between the two boxes' centres

2.10.2 OBB and OBB Intersection testing

Let C_0 = the centre of this OBB,
Let C_1 = the centre of the intersecting OBB,
Let $D = C_1 - C_0$,
Let A_i = the axes of this OBB, where $i \in \{0, 1, 2\}$,
Let B_i = the axes of the intersecting OBB, where $i \in \{0, 1, 2\}$,
Let a_i = the extents (half of each side length) of this OBB, where $i \in \{0, 1, 2\}$,
Let b_i = the extents of the intersecting OBB, where $i \in \{0, 1, 2\}$,
Let L = the potential separating axis that we are testing

There are 15 potential axes that we need to check for intersection.

- The 3 axes which represent this OBB's edges, A_i,
- The 3 axes which represent the intersecting OBB's edges, B_i,
- The 9 axes which are the cross products of each axis from this OBB's edges and each axis from the intersecting OBB's edges $A_i \times B_j$.

Together, these axes form an effectively exhaustive list of possible separating axes - if these don't separate the OBBs, nothing will.

We can project a vector onto the potential separating axis L by finding the dot product of the potential separating axis and the vector we are projecting (scaled if necessary) and dividing it by the dot product of the potential separating axis with itself - essentially the square of its length - to ensure that the vector is not affected by the length of L. For each axis, there are three vectors that we need to project:

1. The vector from this OBB's centre to the the intersecting OBB's centre, R.

$$R = \frac{|L \cdot D|}{L \cdot L} \tag{111}$$

2. The vector from this OBB's centre to the furthest extent of this OBB, R_0.

$$R_0 = \sum_{i=0}^{2} \frac{|a_i L \cdot A_i|}{L \cdot L} \tag{112}$$

3. The vector from the intersecting OBB's centre to the furthest extent of the intersecting OBB, R_1.

$$R_1 = \sum_{i=0}^{2} \frac{|b_i L \cdot B_i|}{L \cdot L} \tag{113}$$

Once we have found R, R_0, and R_1, we can do the simple test $R > R_0 + R_1$ (is the distance between the two centres greater than the sum of the maximum projected lengths of the OBBs). If the result is true, then the OBBs are not intersecting, and we can stop doing further collision testing.

As all of these terms have $L \cdot L$ as the denominator, we can ignore it, and just compare the denominators.

Finding R, R_0 and R_1

The methods we must employ to find R, R_0 and R_1 vary depending on what L is. We'll use the above equations to find them.

- $\mathbf{L = A_i}$
  - **R**
    We simply need to project the vector from this OBB's centre to the intersecting OBB's centre onto the potential separating axis: $|A_i \cdot D|$.
  - $\mathbf{R_0}$
    Because the potential separating axis is one of this OBB's axes, and the other axes are perpendicular, then the maximum projected extent is simply a_i, the extent along the axis of OBB which is being used as a separating axis.
  - $\mathbf{R_1}$
    We need to find out how much of the intersecting OBB's axis is projected onto the potential separating axis. To do this, we project each scaled axis onto the potential separating axis and add them all together: $b_0|A_i \cdot B_0| + b_1|A_i \cdot B_1| + b_2|A_i \cdot B_2|$.
- $\mathbf{L = B_i}$ This is the same as the previous, except in terms of B_i rather than A_i
  - **R**
    This is the same as the previous R, just a simple projection: $|B_i \cdot D|$
  - $\mathbf{R_0}$
    As with R_1 previously, we need to project the each of the scaled axes of this OBB onto the potential separating axis and add them together: $a_0|A_0 \cdot B_i| + a_1|A_1 \cdot B_i| + a_2|A_2 \cdot B_i|$.
  - $\mathbf{R_1}$
    As with R_0 previously, this will just be the extent of the axis which is being used as the potential separating axis: b_i.
- $\mathbf{L = A_i \times B_j}$ This is somewhat more complicated than the previous two, as the potential separating axis is none of the axes of the OBBs.
  - **R**
    This is the similar to the the previous R; we need to to a projection, except this time onto a cross product: $|(A_i \times B_j) \cdot D|$
  - $\mathbf{R_0}$
    We can use the same strategy as before, but with a few tweaks: because the potential separating axis is perpendicular to A_i (that's what a cross product is after all), then the dot product of the two vector will be 0, so we can just ignore the term. Therefore we end up with this:

$$\begin{aligned}
&\text{Let } L = A_0 \times B_j \\
&R_0 = a_1|A_1 \cdot (A_0 \times B_j)| + a_2|A_2 \cdot (A_0 \times B_j)| \\
&\text{Let } L = A_1 \times B_j \\
&R_0 = a_0|A_0 \cdot (A_1 \times B_j)| + a_2|A_2 \cdot (A_1 \times B_j)| \\
&\text{Let } L = A_2 \times B_j \\
&R_0 = a_0|A_0 \cdot (A_2 \times B_j)| + a_1|A_1 \cdot (A_2 \times B_j)|
\end{aligned}$$

  - $\mathbf{R_1}$
    This is the same as R_0, including the exclusion of the perpendicular vector, except this in

terms of the intersecting OBB's axes:

$$\begin{aligned}
&\text{Let } L = A_i \times B_0 \\
&R_0 = b_1|B_1 \cdot (A_i \times B_0)| + b_2|B_2 \cdot (A_i \times B_0)| \\
&\text{Let } L = A_i \times B_1 \\
&R_0 = b_0|B_0 \cdot (A_i \times B_1)| + b_2|B_2 \cdot (A_i \times B_1)| \\
&\text{Let } L = A_i \times B_2 \\
&R_0 = b_0|B_0 \cdot (A_i \times B_2)| + b_1|B_1 \cdot (A_i \times B_2)|
\end{aligned}$$

These values provide us with this table:

L	$\mathbf{R}_0$	$\mathbf{R}_1$	**R**
A_0	a_0	$b_0\|A_0 \cdot B_0\| + b_1\|A_0 \cdot B_1\| + b_2\|A_0 \cdot B_2\|$	$\|A_0 \cdot D\|$
A_1	a_1	$b_0\|A_1 \cdot B_0\| + b_1\|A_1 \cdot B_1\| + b_2\|A_1 \cdot B_2\|$	$\|A_1 \cdot D\|$
A_2	a_2	$b_0\|A_2 \cdot B_0\| + b_1\|A_2 \cdot B_1\| + b_2\|A_2 \cdot B_2\|$	$\|A_2 \cdot D\|$
B_0	$a_0\|A_0 \cdot B_0\| + a_1\|A_1 \cdot B_0\| + a_2\|A_2 \cdot B_0\|$	b_0	$\|B_0 \cdot D\|$
B_1	$a_0\|A_0 \cdot B_1\| + a_1\|A_1 \cdot B_1\| + a_2\|A_2 \cdot B_1\|$	b_1	$\|B_1 \cdot D\|$
B_2	$a_0\|A_0 \cdot B_2\| + a_1\|A_1 \cdot B_2\| + a_2\|A_2 \cdot B_2\|$	b_2	$\|B_2 \cdot D\|$
$A_0 \times B_0$	$a_1\|A_1 \cdot (A_0 \times B_0)\| + a_2\|A_2 \cdot (A_0 \times B_0)\|$	$b_1\|B_1 \cdot (A_0 \times B_0)\| + b_2\|B_2 \cdot (A_0 \times B_0)\|$	$\|(A_0 \times B_0) \cdot D\|$
$A_0 \times B_1$	$a_1\|A_1 \cdot (A_0 \times B_1)\| + a_2\|A_2 \cdot (A_0 \times B_1)\|$	$b_0\|B_0 \cdot (A_0 \times B_1)\| + b_2\|B_2 \cdot (A_0 \times B_1)\|$	$\|(A_0 \times B_1) \cdot D\|$
$A_0 \times B_2$	$a_1\|A_1 \cdot (A_0 \times B_2)\| + a_2\|A_2 \cdot (A_0 \times B_2)\|$	$b_0\|B_0 \cdot (A_0 \times B_2)\| + b_1\|B_1 \cdot (A_0 \times B_2)\|$	$\|(A_0 \times B_2) \cdot D\|$
$A_1 \times B_0$	$a_0\|A_0 \cdot (A_1 \times B_0)\| + a_2\|A_2 \cdot (A_1 \times B_0)\|$	$b_1\|B_1 \cdot (A_1 \times B_0)\| + b_2\|B_2 \cdot (A_1 \times B_0)\|$	$\|(A_1 \times B_0) \cdot D\|$
$A_1 \times B_1$	$a_0\|A_0 \cdot (A_1 \times B_1)\| + a_2\|A_2 \cdot (A_1 \times B_1)\|$	$b_0\|B_0 \cdot (A_1 \times B_1)\| + b_2\|B_2 \cdot (A_1 \times B_1)\|$	$\|(A_1 \times B_1) \cdot D\|$
$A_1 \times B_2$	$a_0\|A_0 \cdot (A_1 \times B_2)\| + a_2\|A_2 \cdot (A_1 \times B_2)\|$	$b_0\|B_0 \cdot (A_1 \times B_2)\| + b_1\|B_1 \cdot (A_1 \times B_2)\|$	$\|(A_1 \times B_2) \cdot D\|$
$A_2 \times B_0$	$a_0\|A_0 \cdot (A_2 \times B_0)\| + a_1\|A_1 \cdot (A_2 \times B_0)\|$	$b_1\|B_1 \cdot (A_2 \times B_0)\| + b_2\|B_2 \cdot (A_2 \times B_0)\|$	$\|(A_2 \times B_0) \cdot D\|$
$A_2 \times B_1$	$a_0\|A_0 \cdot (A_2 \times B_1)\| + a_1\|A_1 \cdot (A_2 \times B_1)\|$	$b_0\|B_0 \cdot (A_2 \times B_1)\| + b_2\|B_2 \cdot (A_2 \times B_1)\|$	$\|(A_2 \times B_1) \cdot D\|$
$A_2 \times B_2$	$a_0\|A_0 \cdot (A_2 \times B_2)\| + a_1\|A_1 \cdot (A_2 \times B_2)\|$	$b_0\|B_0 \cdot (A_2 \times B_2)\| + b_1\|B_1 \cdot (A_2 \times B_2)\|$	$\|(A_2 \times B_2) \cdot D\|$

Table 1: The unoptimised equations for R, R_0 and R_1, which can be used in the intersection test $R > R_0 + R_1$

As you are probably getting a sense of, all this can be quite computationally expensive. This is exacerbated by the fact that we need to test the OBBs of the aircraft and all the terrain as well. Using an OBB tree does limit the number of intersection tests necessary rather than trying every possible combination, but we should still try to optimise the intersection test algorithm as much as possible.

Optimisation

As you can see in Table 1, there are two key operations that happen frequently which it is important to optimise: the dot product and the cross product.

Optimising the Dot Product

The dot product can be fairly simply optimised by precalculating the dot product of every pair of axes in this OBB and the intersecting OBB. This can be stored in a matrix and accessed as necessary:

$$C = \begin{bmatrix} A_0 \cdot B_0 & A_0 \cdot B_1 & A_0 \cdot B_2 \\ A_1 \cdot B_0 & A_1 \cdot B_1 & A_1 \cdot B_2 \\ A_2 \cdot B_0 & A_2 \cdot B_1 & A_2 \cdot B_2 \end{bmatrix} \tag{114}$$

$$= \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} \tag{115}$$

So instead of calculating the dot products every time, we can use the matrix instead.

Optimising the Cross Product

The cross product is the more time expensive operation of the two, so it is the most important one to optimise.

We could precalculate them like we do with the dot products, but I would prefer to get rid of them altogether.

Every time we use the cross product, we then find its dot product with another vector.

For example, in Table 1, Row 7, Column R_0 we see:

$$A_1 \cdot (A_0 \times B_0) \tag{116}$$

This form is called the **Triple Scalar Product**, which has several properties, the relevant one here being that you can cyclically rearrange it:

$$A_1 \cdot (A_0 \times B_0) = B_0 \cdot (A_1 \times A_0) = A_0 \cdot (B_0 \times A_1) \tag{117}$$

The second of these equations are important, because it includes the cross product of A_1 and A_0. The cross product creates a vector which is perpendicular to the two original vectors. But because A_1 and A_0 are both orthonormal axes, then we already know about a vector perpendicular to both of them: A_2. So we can substitute A_2 instead of $A_1 \times A_0$ (and put it in an absolute function to account for the potentially incorrect sign). We can then use our precalculated dot products to efficiently compute the result. This gives us the following:

$$\begin{aligned} A_1 \cdot (A_0 \times B_0) &= B_0 \cdot (A_1 \times A_0) && (118)\\ &= |B_0 \cdot A_2| && (119)\\ &= |c_{20}| && (120) \end{aligned}$$

We can use this to replace every cross product in the R_0 and R_1 column, but the R column is slightly more involved. If we look at Row 7 again, Column R, we have this:

$$(A_0 \times B_0) \cdot D \tag{121}$$

Cyclically rearranging this doesn't get us anywhere, because none of the the vectors are intrinsically linked. Instead, we can transform the basis of B_0 into the axes of this OBB by projecting it onto each of this OBB's axes.

$$\begin{aligned} B_0 &= (A_0 \cdot B_0)A_0 + (A_1 \cdot B_0)A_1 + (A_2 \cdot B_0)A_2 && (122)\\ &= c_{00}A_0 + c_{10}A_1 + c_{20}A_2 && (123) \end{aligned}$$

Now we can substitute this back into Equation 121:

$$\begin{aligned} (A_0 \times (c_{00}A_0 + c_{10}A_1 + c_{20}A_2)) \cdot D &= ((c_{00}(A_0 \times A_0) + c_{10}(A_0 \times A_1) + c_{20}(A_0 \times A_2)) \cdot D && (124)\\ &= (c_{10}A_2 - c_{20}A_1) \cdot D && (125)\\ &= c_{10}A_2 \cdot D - c_{20}A_1 \cdot D && (126) \end{aligned}$$

The cross product of a vector with itself is a zero vector, so we can ignore $A_0 \times A_0$. We can use the orthonormality yet again to convert $A_0 \times A_1$ and $A_0 \times A_2$ into A_2 and $-A_1$ respectively (we need to multiply A_1 by -1 to negate the cross product - because we are going 'down' from A_0 to A_2, rather than 'up' from A_2 to A_0.

We can use Equation (120) and Equation (126) to replace every cross product in the table, giving us a new table, the one provided in [21]:

L	R_0	R_1	R
A_0	a_0	$b_0\|c_{00}\| + b_1\|c_{01}\| + b_2\|c_{02}\|$	$\|A_0 \cdot D\|$
A_1	a_1	$b_0\|c_{10}\| + b_1\|c_{11}\| + b_2\|c_{12}\|$	$\|A_1 \cdot D\|$
A_2	a_2	$b_0\|c_{20}\| + b_1\|c_{21}\| + b_2\|c_{22}\|$	$\|A_2 \cdot D\|$
B_0	$a_0\|c_{00}\| + a_1\|c_{10}\| + a_2\|c_{20}\|$	b_0	$\|B_0 \cdot D\|$
B_1	$a_0\|c_{01}\| + a_1\|c_{11}\| + a_2\|c_{21}\|$	b_1	$\|B_1 \cdot D\|$
B_2	$a_0\|c_{02}\| + a_1\|c_{12}\| + a_2\|c_{22}\|$	b_2	$\|B_2 \cdot D\|$
$A_0 \times B_0$	$a_1\|c_{20}\| + a_2\|c_{10}\|$	$b_1\|c_{02}\| + b_2\|c_{01}\|$	$\|c_{10}A_2 \cdot D - c_{20}A_1 \cdot D\|$
$A_0 \times B_1$	$a_1\|c_{21}\| + a_2\|c_{11}\|$	$b_0\|c_{02}\| + b_2\|c_{00}\|$	$\|c_{11}A_2 \cdot D - c_{21}A_1 \cdot D\|$
$A_0 \times B_2$	$a_1\|c_{22}\| + a_2\|c_{12}\|$	$b_0\|c_{01}\| + b_1\|c_{00}\|$	$\|c_{12}A_2 \cdot D - c_{22}A_1 \cdot D\|$
$A_1 \times B_0$	$a_0\|c_{20}\| + a_2\|c_{00}\|$	$b_1\|c_{12}\| + b_2\|c_{11}\|$	$\|c_{20}A_0 \cdot D - c_{00}A_2 \cdot D\|$
$A_1 \times B_1$	$a_0\|c_{21}\| + a_2\|c_{01}\|$	$b_0\|c_{12}\| + b_2\|c_{10}\|$	$\|c_{21}A_0 \cdot D - c_{01}A_2 \cdot D\|$
$A_1 \times B_2$	$a_0\|c_{22}\| + a_2\|c_{02}\|$	$b_0\|c_{11}\| + b_1\|c_{10}\|$	$\|c_{22}A_0 \cdot D - c_{02}A_2 \cdot D\|$
$A_2 \times B_0$	$a_0\|c_{10}\| + a_1\|c_{00}\|$	$b_1\|c_{22}\| + b_2\|c_{21}\|$	$\|c_{00}A_1 \cdot D - c_{10}A_0 \cdot D\|$
$A_2 \times B_1$	$a_0\|c_{11}\| + a_1\|c_{01}\|$	$b_0\|c_{22}\| + b_2\|c_{20}\|$	$\|c_{01}A_1 \cdot D - c_{11}A_0 \cdot D\|$
$A_2 \times B_2$	$a_0\|c_{12}\| + a_1\|c_{02}\|$	$b_0\|c_{21}\| + b_1\|c_{20}\|$	$\|c_{02}A_1 \cdot D - c_{12}A_0 \cdot D\|$

Table 2: The optimised equations for R, R_0 and R_1, which can be used in the intersection test $R > R_0 + R_1$

2.10.3 Triangle and OBB Intersection

The principles of triangle and OBB intersection testing are similar to those of OBB and OBB intersection testing. However, the triangle doesn't really have a natural 'centre', so we instead have to compare the projected distance from the OBB's centre to its furthest projected extent with the minimum projected distance of all the vectors from the OBB's centre to each of the triangle's vertices. If the vertex closest to the OBB's centre of the triangle is closer than the OBB's furthest extent, then the objects intersect in that axis.

Let C = the centre of the OBB,
Let A_i = the axes of the OBB, where $i \in \{0, 1, 2\}$,
Let a_i = the extents (half of each side length) of the OBB, where $i \in \{0, 1, 2\}$,
Let U_i = the vertices of the triangle, where $i \in \{0, 1, 2\}$,
Let E_i = the edges of the triangle, where $E_0 = U_1 - U_0, E_1 = U_2 - U_0, E_2 = E_1 - E_0$
Let N = the normal of the triangle, $E_0 \times E_1$
Let $D = U_0 - C$
Let L = the potential separating axis that we are testing

There are 13 potential axes that we need to check for intersection.

- The triangle's normal, N
- The 3 axes which represent the OBB's edges, A_i,
- The 9 axes which are the cross products of each axis from the OBB's edges and each edge of the triangle $A_i \times E_j$.

For each axis, there are four vectors that we need to project (division by $L \cdot L$ has been omitted for the same reason as before):

1. The vector from the OBB's centre to the furthest extent of the OBB R.

$$R = \sum_{i=0}^{2} |a_i L \cdot A_i| \tag{127}$$

2. The vector from the OBB's centre to the first vertex of the triangle, p_0.

$$p_0 = L \cdot (U_0 - C) = L \cdot D \tag{128}$$

3. The vector from the OBB's centre to the second vertex of the triangle, p_1.

$$p_1 = L \cdot (U_1 - C) = L \cdot (D + E_0) = p_0 + L \cdot E_0 \tag{129}$$

4. The vector from the OBB's centre to the third vertex of the triangle, p_2.

$$p_2 = L \cdot (U_2 - C) = L \cdot (D + E_1) = p_0 + L \cdot E_1 \tag{130}$$

Once we have found R, p_0, p_1 and p_2 we can do the simple test $\min(p_0, p_1, p_2) > R$ (is the vertex closest to the OBB's centre of the triangle further away than the OBB's furthest extent, then the objects intersect in that axis.). If the result is true, then the objects are not intersecting, and we can stop doing further collision testing.

Finding R, p_0, p_1 and p_2

The methods we must employ to find R, p_0, p_1 and p_2 vary depending on what L is. We'll use the above equations to find them.

- $\mathbf{L = N}$
  - **R**
    As we often did in the OBB and OBB testing, we need to project each of the scaled axes of the OBB onto the normal of the triangle: $R = a_0|N \cdot A_0| + a_1|N \cdot A_1| + a_2|N \cdot A_2|$.
  - $\mathbf{p_0}$
    Using Equation (128), all we have to do is project D onto N: $p_0 = N \cdot D$.
  - $\mathbf{p_1}$ **&** $\mathbf{p_2}$
    All of the vertices of the triangle project onto the same point on the normal, so $p_1 = p_2 = p_0$.
- $\mathbf{L = A_i}$
  - **R**
    Given that we are projecting onto one of the OBB's own axes, all we need is the extent of that axis: a_i.
  - $\mathbf{p_0}$
    Once again, we simply have to project D onto A_i: $p_0 = A_i \cdot D$
  - $\mathbf{p_1}$
    We can use Equation (129) to adjust p_0 so it represents p_1: $p_1 = p_0 + A_i \cdot E_0$
  - $\mathbf{p_2}$
    Similar to p_1, we can use Equation (130) to find p_2: $p_2 = p_0 + A_i \cdot E_2$
- $\mathbf{L = A_i \times E_j}$
  - **R**
    To find R, we can use Equation (127) to project each scaled axis onto $A_i \times E_j$: $R = a_0|A_0 \cdot (A_i \times E_j)| + a_1|A_1 \cdot (A_i \times E_j)| + a_2|A_2 \cdot (A_i \times E_j)|$ As before, we can remove some terms, as the dot product of a vector with its cross product is 0:

$$\begin{aligned}
&\text{Let } L = A_0 \times E_j \\
&R_0 = a_1|A_1 \cdot (A_0 \times E_j)| + a_2|A_2 \cdot (A_0 \times E_j)| \\
&\text{Let } L = A_1 \times E_j \\
&R_0 = a_0|A_0 \cdot (A_1 \times E_j)| + a_2|A_2 \cdot (A_1 \times E_j)| \\
&\text{Let } L = A_2 \times E_j \\
&R_0 = a_0|A_0 \cdot (A_2 \times E_j)| + a_1|A_1 \cdot (A_2 \times E_j)|
\end{aligned}$$

From there, we can use the previous optimisation techniques to remove the cross products:

$$\begin{aligned}&\text{Let } L = A_0 \times E_j\\&R_0 = a_1|A_2 \cdot E_j| + a_2|A_1 \cdot E_j|\\&\text{Let } L = A_1 \times E_j\\&R_0 = a_0|A_2 \cdot E_j| + a_2|A_0 \cdot E_j|\\&\text{Let } L = A_2 \times E_j\\&R_0 = a_0|A_1 \cdot E_j| + a_1|A_0 \cdot E_j|\end{aligned}$$

- **p_0**
  This is the simplest one to find of the four, we just project D onto $A_i \times E_j$: $p_0 = A_i \times E_j \cdot D$
- **p_1**
  We can use the same equation as usual, but we will have to adjust the result based on the edge which E_j represents: $p_1 = p_0 + (A_i \times E_j) \cdot E_0$.
  We can use the **Triple Scalar Product** again: $p_1 = p_0 + A_i \cdot (E_j \times E_0)$.
  $E_j \times E_0$ is different based on what E_j is:
  If $E_j = E_0$, then the cross product returns a zero vector, and $p_1 = p_0$.
  If $E_j = E_1$, then the cross product is $E_1 \times E_0$. The normal of the triangle is defined as $E_0 \times E_1$, so $E_1 \times E_0$ is just the normal negated. Therefore $p_1 = p_0 - A_i \cdot N$.
  If $E_j = E_2$, then the cross product is $E_2 \times E_0$. This creates a vector perpendicular to the triangle - the normal. However, if we use the right-hand rule, we can see that it is again in the opposite direction to the normal. Therefore $p_1 = p_0 - A_i \cdot N$.

- **p_2**
  p_2, as always, is similar to p_1. Using (130), we get $p_2 = p_0 + (A_i \times E_j) \cdot E_1$.
  We can turn this into $p_2 = p_0 + A_i \cdot (E_j \times E_1)$ We the interpret this based on what E_j is:
  If $E_j = E_0$, then the cross product is $E_0 \times E_1$. This simply is the normal, so $p_2 = p_0 + A_i \cdot N$.
  If $E_j = E_1$, then the cross product is $E_1 \times E_1$. This is a zero vector, so $p_2 = p_0$.
  If $E_j = E_2$, then the cross product is $E_2 \times E_1$. This is the negated normal, which means that $p_2 = p_0 - A_i \cdot N$.

After all that, we can generate another table, this time for triangle and OBB intersection: Once we have

L	p_0	p_1	p_2	R
N	$N \cdot D$	p_0	p_0	$a_0 \lvert N \cdot A_0 \rvert + a_1 \lvert N \cdot A_1 \rvert + a_2 \lvert N \cdot A_2 \rvert$
A_0	$A_0 \cdot D$	$p_0 + A_0 \cdot E_0$	$p_0 + A_0 \cdot E_1$	a_0
A_1	$A_1 \cdot D$	$p_0 + A_1 \cdot E_0$	$p_0 + A_1 \cdot E_1$	a_1
A_2	$A_2 \cdot D$	$p_0 + A_2 \cdot E_0$	$p_0 + A_2 \cdot E_1$	a_2
$A_0 \times E_0$	$A_0 \times E_0 \cdot D$	p_0	$p_0 + A_0 \cdot N$	$a_1 \lvert A_2 \cdot E_0 \rvert + a_2 \lvert A_1 \cdot E_0 \rvert$
$A_0 \times E_1$	$A_0 \times E_1 \cdot D$	$p_0 - A_0 \cdot N$	p_0	$a_1 \lvert A_2 \cdot E_1 \rvert + a_2 \lvert A_1 \cdot E_1 \rvert$
$A_0 \times E_2$	$A_0 \times E_2 \cdot D$	$p_0 - A_0 \cdot N$	$p_0 - A_0 \cdot N$	$a_1 \lvert A_2 \cdot E_2 \rvert + a_2 \lvert A_1 \cdot E_2 \rvert$
$A_1 \times E_0$	$A_1 \times E_0 \cdot D$	p_0	$p_0 + A_0 \cdot N$	$a_0 \lvert A_2 \cdot E_0 \rvert + a_2 \lvert A_0 \cdot E_0 \rvert$
$A_1 \times E_1$	$A_1 \times E_1 \cdot D$	$p_0 - A_1 \cdot N$	p_0	$a_0 \lvert A_2 \cdot E_1 \rvert + a_2 \lvert A_0 \cdot E_1 \rvert$
$A_1 \times E_2$	$A_1 \times E_2 \cdot D$	$p_0 - A_1 \cdot N$	$p_0 - A_1 \cdot N$	$a_0 \lvert A_2 \cdot E_2 \rvert + a_2 \lvert A_0 \cdot E_2 \rvert$
$A_2 \times E_0$	$A_2 \times E_0 \cdot D$	p_0	$p_0 + A_0 \cdot N$	$a_0 \lvert A_1 \cdot E_0 \rvert + a_1 \lvert A_0 \cdot E_0 \rvert$
$A_2 \times E_1$	$A_2 \times E_1 \cdot D$	$p_0 - A_2 \cdot N$	p_0	$a_0 \lvert A_1 \cdot E_1 \rvert + a_1 \lvert A_0 \cdot E_1 \rvert$
$A_2 \times E_2$	$A_2 \times E_2 \cdot D$	$p_0 - A_2 \cdot N$	$p_0 - A_2 \cdot N$	$a_0 \lvert A_1 \cdot E_2 \rvert + a_1 \lvert A_0 \cdot E_2 \rvert$

Table 3: The equations for R, p_0, p_1 and p_2, which can be used in the intersection tests $\min(p_0, p_1, p_2) > R$ or $\max(p_0, p_1, p_2) < -R$

found the values, we can do $\min(p_0, p_1, p_2) > R$ or $\max(p_0, p_1, p_2) < -R$. If the test returns true, then the objects do not intersect on that axis or at all.

2.11 Collision Response

Once we have found an intersection between an aircraft part's OBB and a triangle of the terrain, we need to react to it. Initially I planned to add a force which would nullify the component of the velocity

which acted in the direction of the terrain's normal. However, I also needed to incorporate the torque, for which I would need to find the force at each intersecting aircraft part and then calculate the torque about the aircraft's centre of gravity. I used representational abstraction to strip out the unnecessary parts of the scenario, replacing the aircraft with a beam and the wheels with pillars. This led me to develop what has been dubbed the 'Three Pillar Problem'.

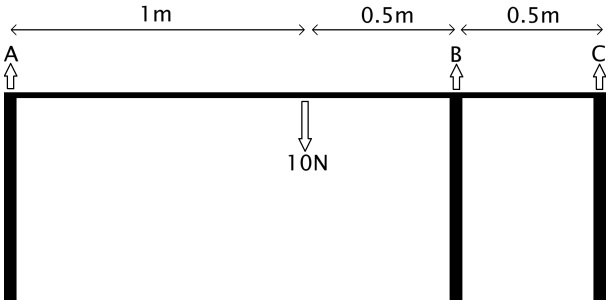


Figure 24: A diagram of the three pillar problem

The Three Pillar Problem

The problem states that there is a uniform beam of weight 10 Newtons and length 2 metres. It is supported by 3 pillars: one on its left end, one 0.5 metres from its right end and one on its right end. The problem asks: what are the normal reaction forces (shown by A, B and C) at each pillar?

From this problem, we can derive two equations, one from equating the upwards and downwards forces:

$$A + B + C = 10N \tag{131}$$

and one from taking moments about the centre:

$$A = 0.5B + C \tag{132}$$

However, in order to solve a simultaneous equation with 3 unknowns (A, B and C), one needs three distinct equations which link each unknown. This is where the issue lies. We only have two: (131) and (132). Therefore, no discrete value for A, B or C can be found.

This produces the somewhat counterintuitive conclusion that there is no answer to the problem. There is no way of telling what the normal reaction force would be at these pillars. For me, this is especially odd, given that this problem could theoretically be reproduced in real life, and there would be values for the normal reaction forces, yet we cannot with certainty predict them.

After pondering this problem for a while, I considered finding the average position of the locations where the collisions wer taking place and apply the torque and force from there.

Later, I did some research on the topic and concluded that applying a force in response to a collision was not the best strategy. This was for many reasons, one of which was because I would have to calculate the resultant force, of the aircraft, apply the force, calculate the updated position of the aircraft, do collision tests, calculate the new reaction forces, resolve all the forces, apply them and finally update the position of the aircraft. That sounds like a lot.

But the main reason is because it seemed that impulses were far more effective.

2.11.1 Impulses

> "An instantaneous change in velocity, without first integrating from acceleration with regard to time, is known as an impulse."

This is a quote from a paper regarding collision response by the University of Newcastle [24]. The paper details how one can calculate and use an impulse to simulate collision responses.

They discuss collision between two objects with velocity and mass, but for the purposes of the flight simulator, we only need to concern ourselves with the velocity and mass of the aircraft (the velocity and mass of the terrain is assumed to be $\mathbf{0}$ and ∞ respectively).

Firstly, we need to define the impulse (often denoted with J). We can initially represent it as some force, F, applied over some time, t:

$$J = F\Delta t \tag{133}$$

We can substitute Newton's second law, $F = ma$ into this equation:

$$J = ma\Delta t \tag{134}$$

$$J = m\frac{\Delta v}{\Delta t}\Delta t \tag{135}$$

$$J = m\Delta v \tag{136}$$

This tells us that J is a change in velocity proportional to the mass of the object.

To implement this, we first need to find the component of the aircraft's velocity which is in the direction of the intersecting triangle's (normalised) normal. This can easily be done with a dot product.

$$v_j = v \cdot \hat{n} \tag{137}$$

where

v_j = the speed of the collision, ms^{-1},

v = the velocity of the intersecting aircraft part, ms^{-1},

$\hat{n}$ = the normalised normal of the intersecting triangle

However, we need to account for the potential elasticity of the collision. We do this with the 'coefficient of restitution', e which represents how 'bouncy' the collision: 0 = totally inelastic, 1 = perfectly elastic. We can then adjust the equation to include this:

$$v_j = -(1 + e)v \cdot \hat{n} \tag{138}$$

We can substitute this into Equation (136):

$$J = \frac{v_j}{m^{-1}} \tag{139}$$

NB: A multiplication by mass is conventionally represented by a division of the reciprocal of the mass, to allow for more convenient manipulation of infinite masses.

The above equation gives us a value for J, but it does not account for angular velocity:

$$\theta = I^{-1}(r \times \hat{n}) \times r \tag{140}$$

where

I^-1 = the reciprocal of the moment of inertia of the aircraft,

r = the position of the collision relative to the centre of gravity of the aircraft,

$\hat{n}$ = the normalised normal of the intersecting triangle

This gives us the component of the angular velocity which is moving in the direction of the normal. We can combine this with Equation (139) to form a complete equation for J:

$$J = \frac{-(1 + e)v \cdot \hat{n}}{m^{-1} + (I^{-1}(r \times \hat{n}) \times r) \cdot \hat{n}} \tag{141}$$

This gives us a scalar value for J, which we can use to apply impulses to the translational and angular velocity of the aircraft:

$$v_f = v + m^{-1}J\hat{n} \tag{142}$$

where

v_f = final velocity

$$\omega_f = \omega + I^{-1}(r_1 \times J\hat{n}) \tag{143}$$

where

ω = initial angular velocity

ω_f = final angular velocity

Applying these impulses will resolve the collisions, allowing the aircraft to rest on the ground or collide with objects.

2.12 Terminal Velocity

In order to prevent the aircraft from reaching unrealistically high velocities, I need to calculate the terminal velocity of the aircraft with which I will limit the velocity.

The terminal velocity is usually considered the velocity at which the force due to drag is equal to the gravitational force of the falling object, or in equation form:

$$F_D = mg \tag{144}$$

where

F_D = drag force, N,

m = mass, kg,

g = acceleration due to gravity, ms^{-2}

If we combine this equation with Equation (14) (substituting for F_D), we get the following:

$$mg = \frac{1}{2}C_D\rho A v_T^2 \tag{145}$$

$$\frac{2mg}{C_D\rho A} = v_T^2 \tag{146}$$

$$v_T = \sqrt{\frac{2mg}{C_D\rho A}} \tag{147}$$

where

C_D = drag coefficient,

ρ = fluid density kgm^{-3},

A = frontal area, m^2,

v_T = terminal velocity, ms^{-1}

This gives us the equation for terminal velocity, (147). However, we can modify this equation to apply to all scenarios by replacing the acceleration due to gravity with the acceleration of the aircraft.

$$v_T = \sqrt{\frac{2ma}{C_D\rho A}} \tag{148}$$

where

a = the acceleration of the aircraft

This is the equation we will use to find the velocity limit of the Grob Tutor.

However, there are two important factors in Equation 148 which need a lot of consideration: C_D (drag coefficient) and A (frontal area).

2.12.1 Drag Coefficient (again)

Yes, we have already discussed the drag coefficient. A lot. But it's different this time: we need the drag coefficient of the entire aircraft. We can find this by simply summing all the drag coefficients of the individual aerodynamic components to get an approximation for the total drag coefficient aircraft.

2.13 Frontal Area

2.13.1 What is 'Frontal Area'?

Now, observant readers may have noticed that the frontal area mentioned in Equation (148) is not the same as the surface area previously mentioned in Equation (14) (page 15). That referred to the planform area of the aerodynamic surface, but the frontal area is something different entirely.

The frontal area of the aircraft is the area of the aircraft's shape projected onto a plane which has a normal of the aircraft's velocity. This is best understood with a diagram:

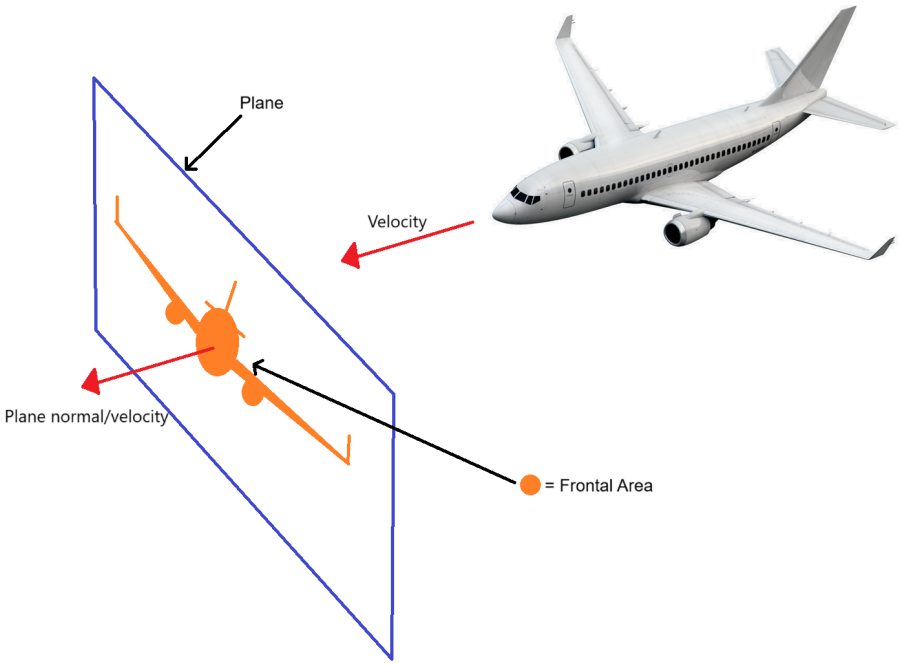


Figure 25: Frontal area

Think of it like the area that the drag acts on. It won't always be the 'front' of the aircraft: for instance, if the aircraft is falling straight down, it would represent the bottom of the aircraft instead. Therefore, we can't use a constant precalculated value, but we need to recalculate it every frame based on the aircraft's updated rotation and velocity. However, this is very inefficient to do accurately, as I would have to project every vertex (of 43,000) of the aircraft onto the plane each frame. Instead, I will make use of components which we have already talked about at length: bounding boxes. Each OBB takes thousands of the model's vertices and represents them using just eight. This drastically cuts down the processing needed to calculate the area each frame, at the cost of accuracy.

2.13.2 Problem discussion

It took me a while to figure out what to do at this stage. How do I find the frontal area of a group of bounding boxes?

Raycasting

This was one idea I briefly considered: from a point along the vector of the aircraft's velocity, I would cast rays to each outer vertex and find the corresponding area. This didn't seem like a good or substantial idea, so I soon abandoned it,

Rasterising and Monte Carlo Simulation

Instead, I could project each vertex onto a plane with the normal of the aircraft's velocity. Then, I would create a rectangle around the convex hull of the vertices from which I could find the area by rasterising or Monte Carlo simulation.

Rasterising

The process of rasterising would subdivide the square into much smaller squares. After that, it would find the number of the smaller squares which are contained within the hull and multiply that number by the area of each square to find an approximation for the total frontal area.

Monte Carlo Simulation

This method includes placing lots of random points around the square, and testing to see if each point is inside the hull. Then, we would multiply the proportion of points inside the hull by the area of the large square to find an approximation for the frontal area.

Hulls

Both the rasterising and Monte Carlo methods require some sort of outline of the aircraft's projected shape, which is called a hull, because they both involve checking for intersection. However, finding the hull comes with some difficulties. This is because lots of projected vertices are within the outline, and are therefore irrelevant to finding the area, so must be removed.

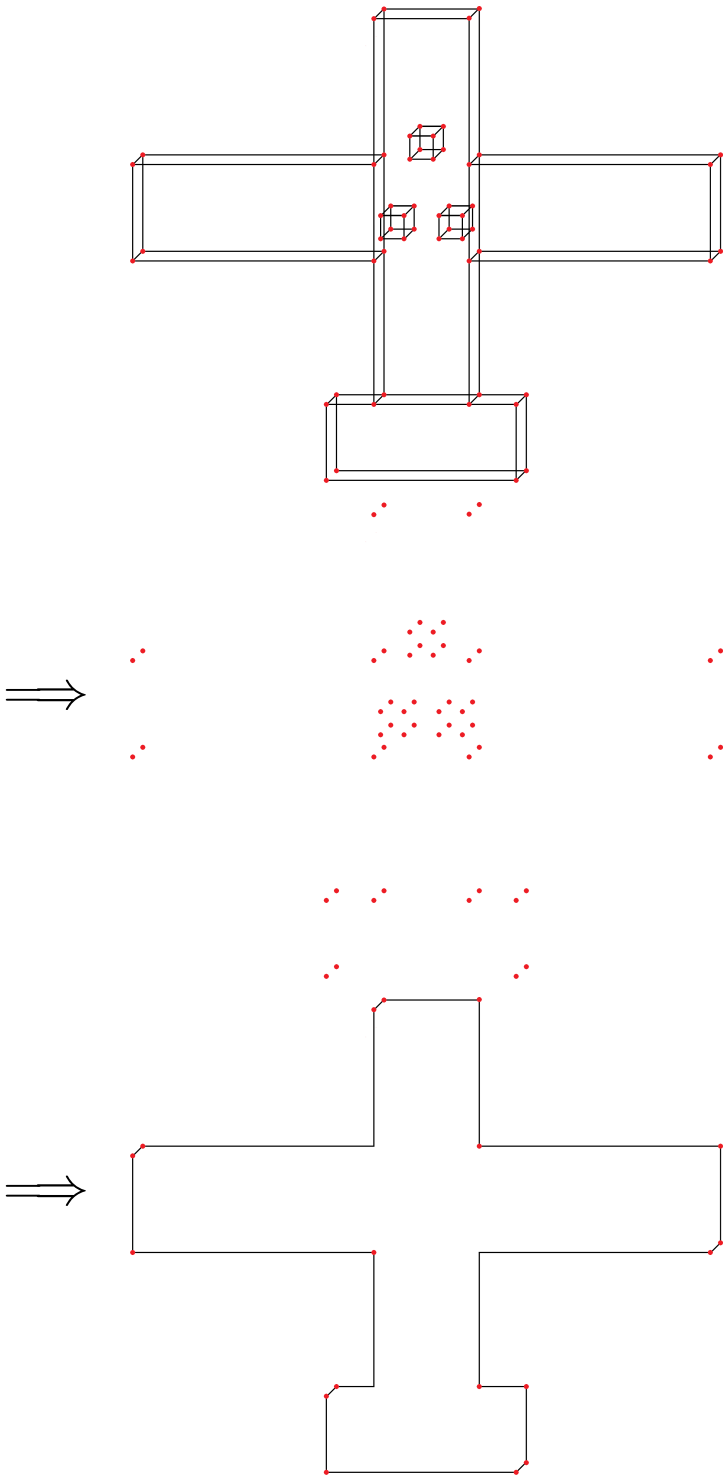


Figure 26: The ideal sequence from bounding boxes to hull

In most cases, algorithms are applied to remove the unnecessary points and generate one of two main types of hull. Unfortunately, the hull shown in the final image of Figure 26 is neither of the two main types of hull, and to understand why, we need to look at the hull types.

Convex hull

This is the most common type of hull, and it includes all of the outer points of a group. Think of them as wrapping a rope around a set of pegs. This works well for convex shapes. Unfortunately, the aircraft's frontal area is concave, so the convex hull is inappropriate.

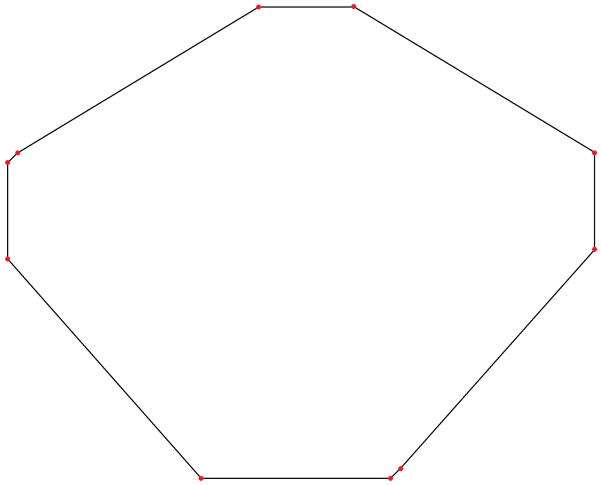


Figure 27: A convex hull for the projected points

In Figure 27 we try to use it for the entire aircraft's shape, and the resultant shape is clearly not representative.

Concave hull

The other option is a concave hull, which is far less reliable and more heuristic, subjective and complicated. Each concave hull includes inner points the concave hull wouldn't and seeks to maximise or minimise a different variable based on the implementation. Due to this inconsistency, each shape has many potential concave hulls:

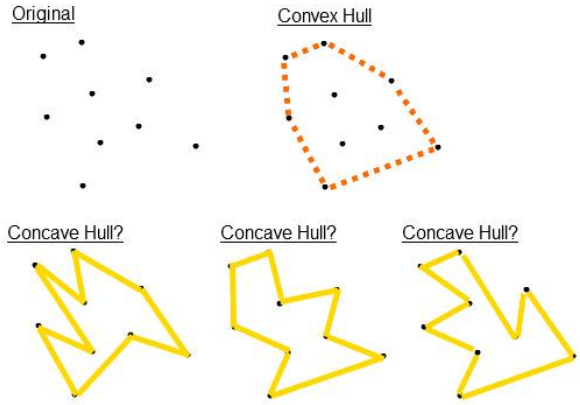


Figure 28: A comparison of a convex vs concave hull, and a demonstration of the variety of possible concave hulls

I downloaded a program which generates concave hulls, and tried to replicate the projected points as seen in Figure 26:

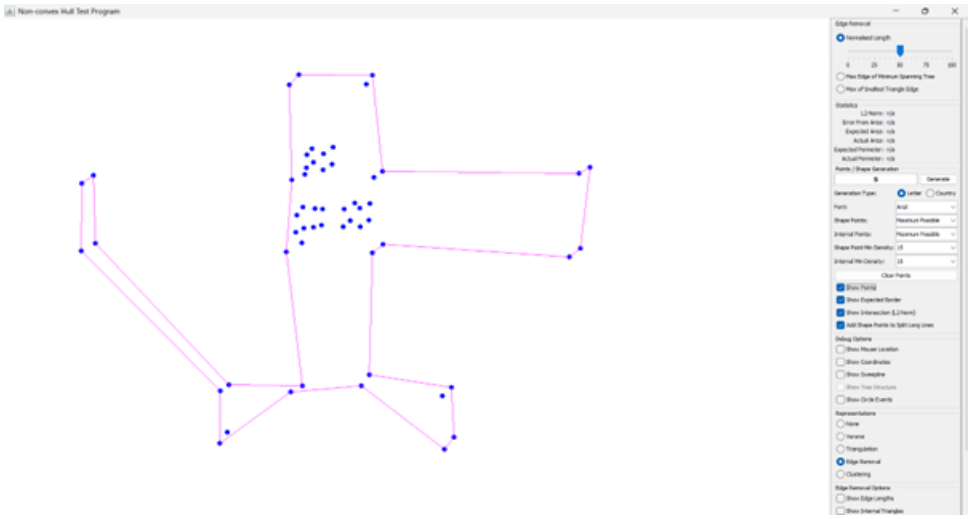


Figure 29: A potential concave hull for the projected points

Figure 29 clearly doesn't represent the shape either, so using a concave hull doesn't seem like a good idea either.

Why neither work

Another issue with both of these approaches is that they are both limited by the points they are trying to represent. If we look at the desired hull, we can see that some of the vertices of the hull aren't actually designated by any of the projected vertices of the bounding boxes but instead at the intersection where two bounding boxes meet.

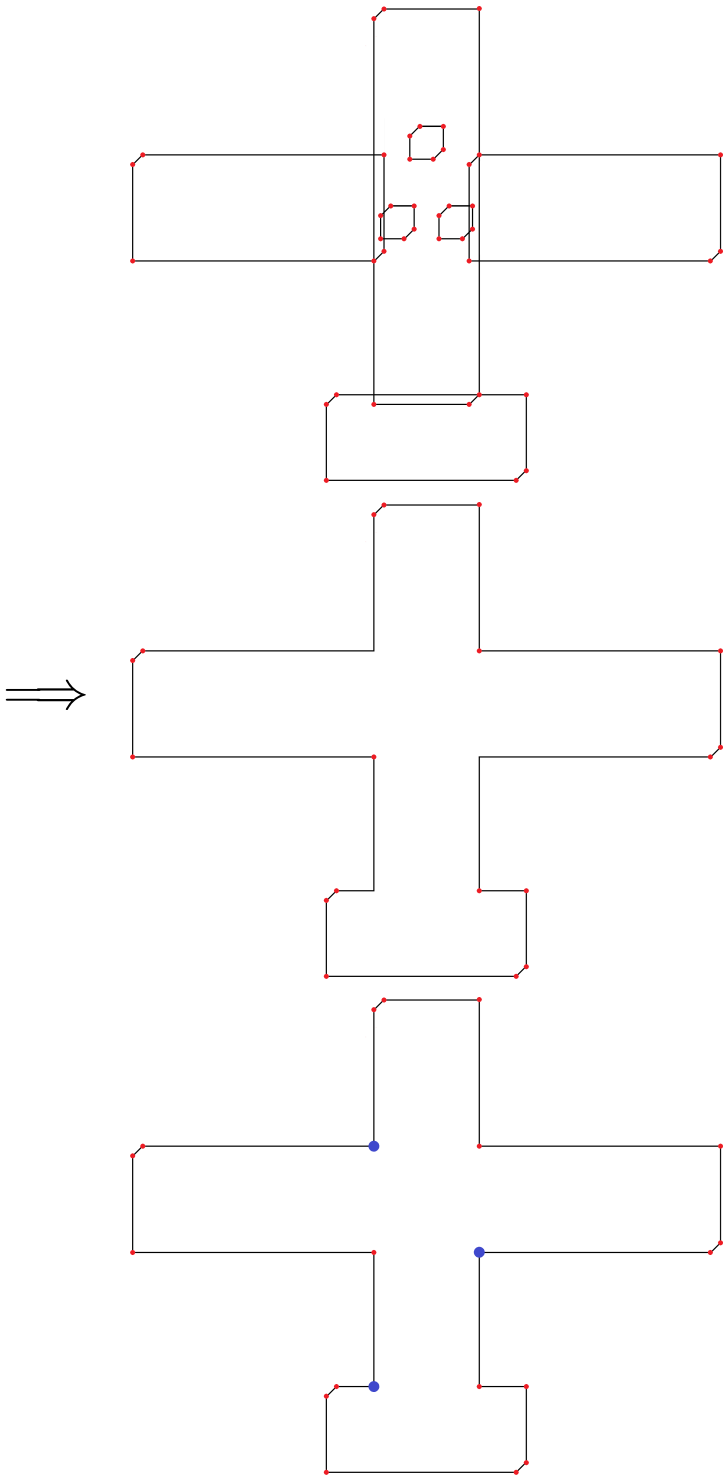


Figure 30: Some vertices of the desired hull (blue) must be created, not merely projected

The Plan

Ultimately, the issue with the previous approaches of Rasterising or Monte Carlo Simulation seems to be that we can't form an accurate hull from a simple collection of every projected vertex of the bounding boxes to use in those algorithms.

To use either of those two methods, I would need to firstly find a way of creating an accurate hull. And if I do that, there would be no point in pursuing either of the two suggested approaches, as at that point I could simply calculate the area of the hull using geometry.

I also considered finding the area for each bounding box individually, and adding them together. The

convex hull works well for this, as the projection of an OBB will always be a convex shape. Unfortunately, the bounding boxes frequently overlap, so the simple addition of areas will not do the trick.

Instead, I will need to employ a rather more sophisticated process, the entirety of which is subsequently documented:

1. Project each OBB's vertices onto the plane.
2. Create a convex hull for each OBB's projected vertices.
3. Form a union of two of the hulls.
4. Repeat Step 3 until only one hull remains: the hull of the entire aircraft.
5. Find the area of the hull using geometry.

This process is visualised in Figure 31.

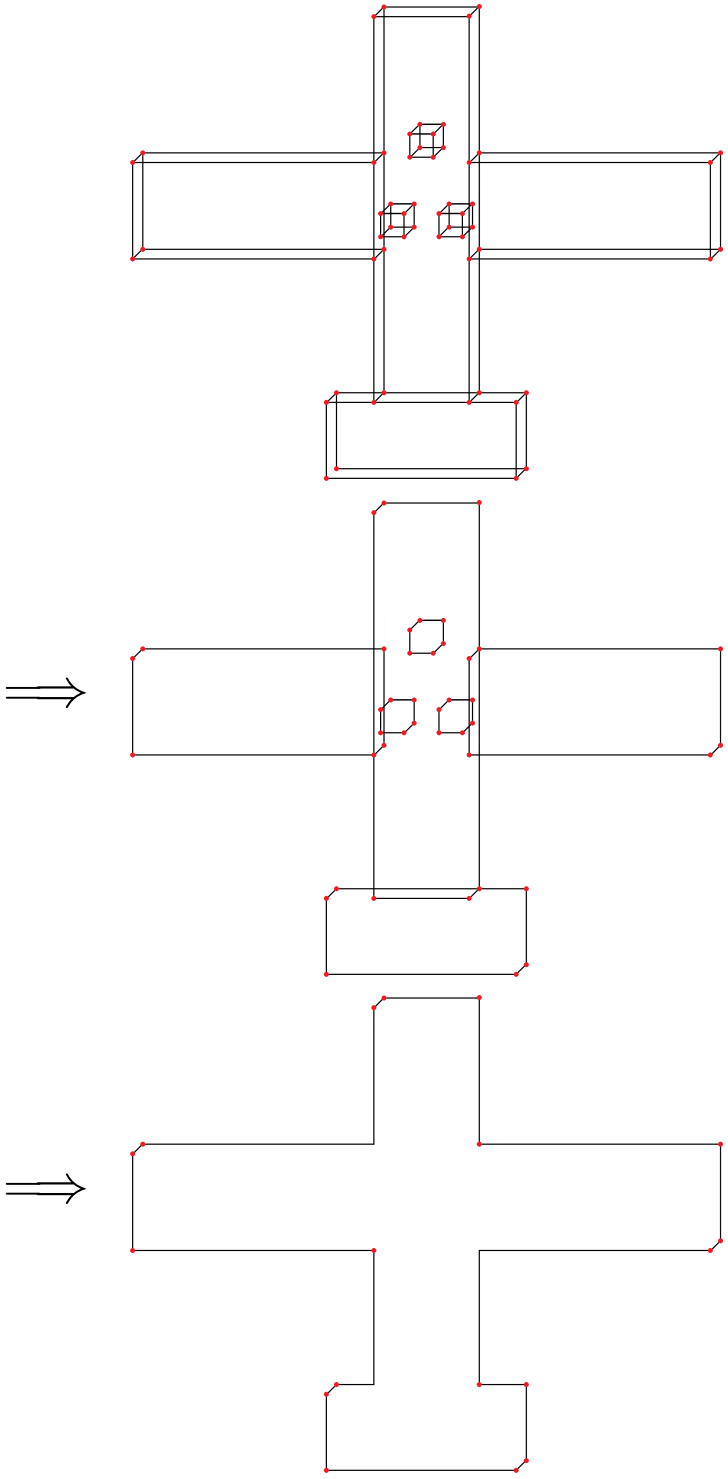


Figure 31: The planned sequence from bounding boxes to hull

2.13.3 Projection

The first stage of all this process is to project the vertices of an OBB onto a plane with the normal of the aircraft's velocity. Three.js provides a plane object, which can be assigned to an attribute of the aircraft model. The normal of this plane can then be updated every frame so that it is consistent with the aircraft's velocity.

The projection step of this process is quite trivial: given a point and the origin point and normal of a plane, one can find multiply the plane's normal by the dot product of the normal and the vector from the origin to the point, then subtract the resulting vector from the point to find the position of the point

projected onto the plane. Three.js has a built in `Plane.projectPoint()` method, which I decided to use for simplicity.

But this is where I found my first of many obstacle when finding the frontal area: how to use these points. These points are all projected onto a plane and therefore coplanar, so they can all be represented as 2D coordinates. 2D coordinates are a lot simpler (as all the algorithms I plan to use will utilise 2D coordinates) and less time-intensive to use, so I should use them over 3D coordinates wherever possible.

Therefore I need a way to convert the coplanar 3D points to 2D points. My first idea was to convert the coordinates to use a polar coordinate system, using the plane's origin as the coordinate system's origin. The pole was easy to find, as I could just find the distance between the origin and the point by using Three.js' Pythagorean method, `Vector3().length()`. However, finding the polar angle presented more of a challenge. My idea to project an arbitrary vector onto the plane and find the angle between that and the point's pole with trigonometry. When I tried this, it caused consistent inaccuracies and inconsistencies in points, especially near the world's origin.

Instead, I found that I would use go with a classic Cartesian style coordinate system. If you think about it, the Cartesian coordinate system is just a representation of how far a point is along the x and y axes. This is exactly what the dot product represents - how far one vector is along another. Therefore, I could create my own x and y axes - both orthogonal and coplanar vectors - and represent each point by how far they are along these axes. I created the x axis by projecting one of the world axes (e.g [1, 0, 0]) onto the plane, then normalising it. I would then get the y axis by finding the cross product of the x axis and the plane's normal. Now, which world axis should I choose? If I chose one, and that one happened at some point to be parallel to the aircraft's velocity (i.e. the aircraft was travelling perfectly in the direction of the x, y or z axis), than it's projected vector would be [0, 0], obviously inappropriate for an axis. Therefore, I will choose the world axis which is represents the smallest component of the aircraft's velocity. In practice, this mean that if the velocity was [4, 7, 2], I would choose [0, 0, 1] as the world axis to be projected, as the z component of the velocity is the smallest. This avoids any chance of the world axis being parallel to the aircraft's velocity.

I found this method of representing the axes to be much more consistent, as it had no issues around the world origin.

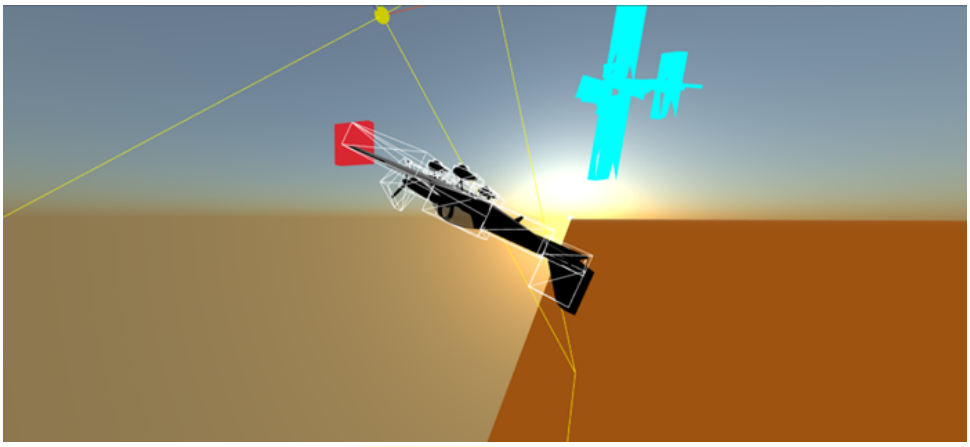


Figure 32: An in-engine demonstration of the frontal projection (cyan) and 2D conversion

2.13.4 Convex Hull Algorithms

Now we have a group of 2D points which represent the frontal shape of the OBB. However, as mentioned in the **Hulls** section on Page 46, this isn't quite what we need, because some superfluous projected vertices will be inside the convex hull of the shape, and will therefore interfere with the subsequent union and area calculations. The consequences of this can be seen in Figure 32, as the projected shape has gaps where it should have full shapes. To combat this, we must employ a convex hull algorithm. (As a reminder, we are looking to find the convex hull of the OBB's projected shape because it is convex, whereas the aircraft's projected shape is concave.)

My research led me to three options: **Graham Scan**, **Jarvis March** and **Chan's algorithm**. Out of these, the Graham Scan appealed to me the most because of its recursive potential.

2.13.5 Graham Scan

Before we can get to the recursive part, we need to order the points in a specific way. To do this, I will find the lowest point of the group, and order the other points anticlockwise from that point.

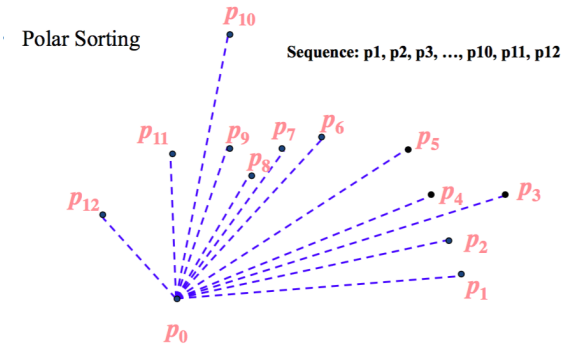


Figure 33: The order of points before the Graham Scan [25]

At this point, the Graham Scan will navigate through the points and select the ones which are included in the convex hull. I think the best course of action is, rather than to create a recursive algorithm which does this, is to convert the points from an array into a Polygonal Circular Doubly Linked List. This both provides an elegant way to navigate the points, allows each point to easily access the previous and subsequent nodes, and also produces a linked list which is vital for the later stages of forming a convex hull.

The algorithm

The Graham Scan algorithm traverses the points and discards a node if path has to turn clockwise to go to the next node.

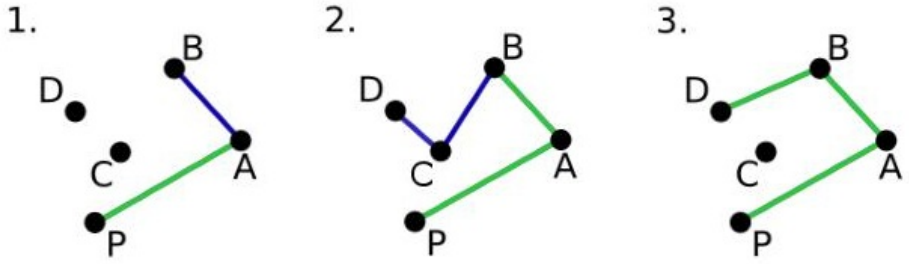


Figure 34: A demonstration of part of a Graham Scan

The following pseudocode represents it.

Algorithm 1 GRAHAM SCAN

```
1: if this node is not the root node then
2:     if next node is clockwise from this node then
3:         kill this node
4:         return this node is deleted
5:     else
6:         if next node is not the root node then
7:             if next node is deleted by graham scan then
8:                 graham scan next node
9:             end if
10:        end if
11:    end if
12: else
13:    if next node is deleted by graham scan then
14:        graham scan next node
15:    end if
16: end if
```

This recursively navigates the linked list and deletes any internal points, leaving a linked list which represents the convex hull of the projected OBB. This leads us on to the next stage.

2.13.6 Union

So now you have two linked lists which represent the convex hulls of two OBB's projected vertices. Next we need to find the union between them. To do this, I will use the Greiner-Hormann Clipping Algorithm.

2.13.7 The Greiner-Hormann Clipping Algorithm [26]

The Greiner-Hormann Clipping Algorithm consists of three steps:

Step 1: Find intersections
Traverse each edge of the polygon and check if the edge intersects with an edge from the other polygon. If so, create a new node at the point of intersection.

Step 2: Designate thresholds
Find out if the root node of the polygon is inside the other polygon or not. Consequently, traverse the polygon and designate if each point of intersection is an entry or an exit of the other polygon.

Step 3: Construct union
Create a new linked list by navigating the polygon and swapping to navigate the other polygon at an intersection. The direction of traversal is specified by the entry/exit state of each intersection.

The result of this is a new linked list which contains all the nodes from the union of the projected OBBs.

2.13.8 Edge Intersection

As discussed on Page 49, what truly limits the simple convex/concave hull algorithms is that they just account for the nodes and don't account for the intersections between the edges. The Greiner-Hormann Clipping Algorithms resolves this by finding all the intersections of the two intersecting polygons.

For each node, the algorithm tests if each edge of the other polygon intersects with the edge between this edge and the next edge, using the following test:

Let $\overrightarrow{OA}$ = the position vector of this node

Let $\overrightarrow{AB}$ = the vector from this node to the next node

Let $\overrightarrow{OC}$ = the position vector of the other polygon's node

Let $\overrightarrow{CD}$ = the vector from the other polygon's node to its next node

Then the initial equation to solve is:

$$\text{coordinates of intersection} = \overrightarrow{OA} + t\overrightarrow{AB} = \overrightarrow{OC} + s\overrightarrow{CD} \tag{149}$$

where

t = the proportion of $\overrightarrow{AB}$ at which the intersection occurs,

s = the proportion of $\overrightarrow{CD}$ at which the intersection occurs

As we're working with vectors, this can be expanded thus.

$$\begin{pmatrix} A_x \\ A_y \end{pmatrix} + t \begin{pmatrix} B_x - A_x \\ B_y - A_y \end{pmatrix} = \begin{pmatrix} C_x \\ C_y \end{pmatrix} + t \begin{pmatrix} D_x - C_x \\ D_y - C_y \end{pmatrix} \tag{150}$$

which can be separated into two simultaneous equations:

$$A_x + t(B_x - A_x) = C_x + s(D_x - C_x) \tag{151}$$
$$A_y + t(B_y - A_y) = C_y + s(D_y - C_y) \tag{152}$$

Equation (151) can be rearranged into

$$s = \frac{A_x - C_x + t(B_x - A_x)}{D_x - C_x} \tag{153}$$

which can be substituted into Equation (152) to produce

$$t = \frac{(C_y - A_y)(D_x - C_x) + (A_x - C_x)(D_y - C_y)}{(B_y - A_y)(D_x - C_x) - (B_x - A_x)(D_y - C_y)} \tag{154}$$

Using this method, we can also find s as

$$s = \frac{(A_y - C_y)(B_x - A_x) + (C_x - A_x)(B_y - A_y)}{(D_y - C_y)(B_x - A_x) - (D_x - C_x)(B_y - A_y)} \tag{155}$$

The reason we want both t and s in this form, rather than substituting t into Equation (153), is to prevent the zero division error which would otherwise happen if C's and D's x coordinates were equal.

The result of all this is two values, t and s which represent how far along each edge the intersection. This is useful for a number of reasons: it allows us to calculate the coordinates of the intersection using Equation (149), and it allows us to test if there actually is an intersection - if both values are between 0 and 1, then the intersection occurs between the bounds of the edges. If either value is below 0 or above 1, then the intersection occurs beyond the bounds of the vector, so it essentially doesn't exist.

Once we have found the intersection between the edges (if there is one, of course), then we can insert a new node between the current node and the next one and the we do the same for the other polygon. Those intersection nodes are both given a pointer to the other to allow for traversal between polygons later on.

2.13.9 Threshold Designation

Now that we've added all the intersections between the nodes, we need to designate whether each intersection is an entrance into the other polygon or an exit. This allows us to traverse the polygons in such a way that we create the union, rather than the intersect.

Ray Casting

Firstly, we need to find out if the root node is inside the other polygon or not. To do this, we generate a ray in a random direction and test for how many times it intersects with the other polygon. We can use the intersection test detailed in the last section, except this time it is not necessary for t to be less than 1, as the ray should extend infinitely. However, t should still be larger than 0, as the ray begins at the root node and does not extend backwards. These conditions are necessary if we want to use the even-odd rule.

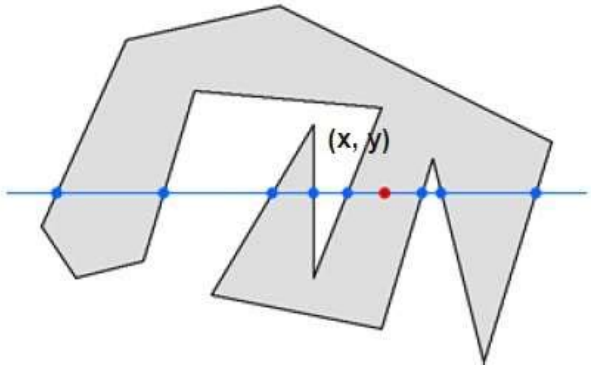


Figure 35: A demonstration of the Even-Odd rule

Even-Odd Rule

The Even-Odd rule states that whether a point is inside a polygon or not can be determined by how many times a ray emitted from it intersects with the edges of a polygon: if the number is odd, then the point is inside the polygon; if it is even, the point is outside the polygon. This is because the ray must always travel in and out of the polygon if it's outside the polygon, but it need only travel out if it's inside the polygon.

Designating Thresholds

Through the combination of ray casting and the Even-Odd rule, we can determine if the root node is inside or outside of the other polygon. We then traverse the polygon until we find an intersection point. An intersection point signifies either that the edge is heading into the other polygon or out of it. If the root node is outside of the other polygon, then the next intersection must signify that the edge is heading into the other polygon, so the intersection must be an entrance. Conversely, if the root node is inside the other polygon, then the next intersection must be an exit.

We can apply this principle further. If the previous intersection was heading out of the shape, it follows that the next one must be heading in, vice versa. Therefore, we can pass a boolean argument through a `designate_thresholds()` method which represents whether the next intersection should be an entrance or an exit. Once it has been used, we can negate it for the next intersection.

2.13.10 Union Creation

Once all the intersections have been designated as an entrance/exit, then we finally have enough information to create the union shape.

During traversal, each node is marked with a visited attribute to provide a base case to stop the traversal algorithm. Additionally, each node is added to a new linked list, which will make up the union of the two polygons.

We begin at the first intersection, because we can't be sure that the root node is part of the union, but we can be sure that each intersection will be.

At each intersection, the algorithm moves to the corresponding intersection which is in the other polygon (as mentioned on the previous page). This allows it to traverse along the outside of both polygons as necessary.

If the intersection is an exit, then the algorithm moves to the next node. If the intersection is an entrance, the algorithm moves to the previous node. Both of these move the algorithm out of the other polygon ensuring that the algorithm correctly traces along the outside of the shape. The current direction (previous/next node) is passed as an argument to ensure that the algorithm continues to move in the correct direction.

Eventually, the algorithm will come across a node which has already been visited. At this point, a linked list representing all the nodes of the union is returned.

2.13.11 Edge Cases

In order to produce a robust and reliable union algorithm, there a few edge cases where there are no intersections which ought to be anticipated and mitigated.

Edge Case 1: This polygon completely contains the other polygon
As there are no intersections in this scenario, the algorithm will not traverse the polygons. However, because the root node has previously been shown to be outside of the shape (see Threshold Designation), we can simply return a clone of this polygon, which is equivalent to the union of the two polygons, because the other polygon is entirely contained within this polygon.

Edge Case 2: The other polygon completely contains this polygon
Once again, the algorithm doesn't traverse the polygons. But in this scenario, this polygon is inside the shape, so we cannot return a clone of this polygon. Instead, we can use the `find_stragglers()` method, which checks if the other polygon is outside this polygon. Because it is, the method returns a clone of the other polygon.

Edge Case 3: The polygons do not intersect
This is the problem that technically turns the Polygonal Circular Doubly Linked List into Polygonal Circular Doubly Linked Matrix. In this case, both of the previous cases occur, because they both don't intersect and have root nodes outside the alternate polygon, as they would if they completely contained the alternate polygon. Because of this fact, both a clone of this polygon and a clone of the other polygon are returned. As these shapes have been proven to be disjunct, there is no way of reconciling them through a combination. Instead, the other polygon is assigned to a `next_polygon` attribute of the root node. This means that the root node contains both a pointer to its fellow nodes and an entire other linked list. This complicates many operations, so I will make sure to account for there being another linked polygon in all of them.

2.13.12 Recursive Polygon construction

Now that we have made the union of two polygons, we can apply the same techniques to all of the projected OBBs. After that, we can again combine the resultant polygons. We continue this process until we have only one remaining polygon, which accurately represents the hull of the entire aircraft's projected shape. Now all we need to do is find the area.

2.13.13 Shoelace Formula

Fortunately, there is a very convenient formula for this, which also works well recursively: the Shoelace Formula.

The Shoelace Formula calculates the area of a polygon from the coordinates of its points. It is as follows:

$$A = \frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + \ldots + x_ny_1 - x_1y_n) \tag{156}$$

In words, it is the x coordinate of each point times the y coordinate of the next point subtract the x coordinate of the next point times the y coordinate of this point, all over 2.

I will implement this by returning the area value of each node through the list, adding to the total as it goes through, and then finally dividing by 2.

And thus we have found the frontal area.

2.13.14 Graphics design

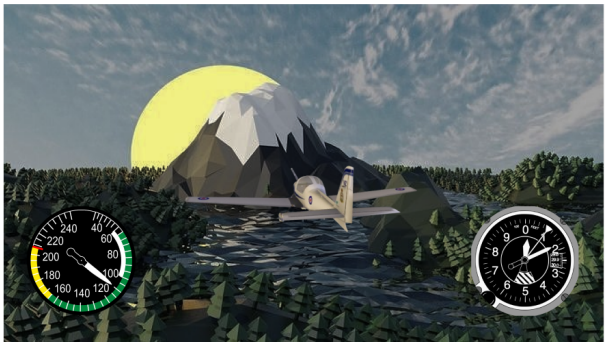


Figure 36: A mockup of the flight simulator page

3 Technical Solution

3.1 Important features

- Aerodynamic coefficient calculations - Page 78
- OBB Intersection Testing - Page 84,
- OBB tree nodes: subdivide and intersects - Page 91,
- OBB creation - Page 95,
- Union creation - Page 126

Now these are only the absolute highlights - there are many other noteworthy features.

3.2 script.js

```
//-----------------------------------------------------------------------------------------------\\
//                                          PREPARATIONS                                         \\
//-----------------------------------------------------------------------------------------------\\


import * as THREE from "three";
import { Sky } from "three/addons/objects/Sky.js";
import {Aircraft, OBB, Point, Triangle, Terrain} from "./physical_interactions new.js";
import {scene, clock, GROB_FILE_PATH, GROB_INFORMATION, GROB_PARTS, TERRAIN_JSON} from "./constants.js";


// Initialising the renderer:
const renderer = new THREE.WebGLRenderer({antialias:true});
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.setPixelRatio(window.devicePixelRatio);
renderer.shadowMap.enabled = true;
renderer.shadowMap.type = THREE.PCFSoftShadowMap;

document.body.appendChild(renderer.domElement);


scene.fog = new THREE.Fog("#dddddd", 100, 3000);

// Representing the sunlight:
scene.add(new THREE.HemisphereLight("#ddddff", "#24601c", 0.7));

// Sky initialisation. The sky is imported.
const sky = new Sky();

sky.scale.setScalar(450000);
let sun_position = new THREE.Vector3().setFromSphericalCoords(1, 1.5, Math.PI);
sky.material.uniforms.sunPosition.value = sun_position;

scene.add(sky);



const grob = new Aircraft(GROB_FILE_PATH, GROB_INFORMATION, GROB_PARTS)

```

```
40
41 const camera_man = {
42     camera: new THREE.PerspectiveCamera(60, window.innerWidth/window.innerHeight, 0.1, 10000),
43     focus: grob,
44     offset: grob.CENTRE_OF_GRAVITY,
45     radius: 10, // The distance which the camera orbits the focus at.
46     sensitivity: 125, // Larger number = less sensitive.
47     update: function() {
48         if (keys.has('=')) {
49             this.radius -= 0.05;
50         }
51         if (keys.has('-')) {
52             this.radius += 0.05;
53         }
54         this.focus_position = this.focus.model.localToWorld(this.offset.clone());
55         // Orbit calculations:
56         let theta = (mouse.clientY - window.innerHeight / 2) / this.sensitivity + Math.PI / 2;
57         let phi = (mouse.clientX - window.innerWidth / 2) / this.sensitivity;
58         this.camera.position.x = this.focus_position.x +  this.radius * Math.cos(phi) * Math.sin(theta);
59         this.camera.position.y = this.focus_position.y + this.radius * Math.cos(theta);
60         this.camera.position.z = this.focus_position.z + this.radius * Math.sin(phi) * Math.sin(theta);
61         // To stop the camera from clipping through the ground:
62         if (this.camera.position.y < 0.1) {
63             this.camera.position.y = 0.1;
64         }
65         this.camera.lookAt(this.focus_position);
66     }
67 }
68
69
70 // A gaffer is a person who controls the lighting on a film set.
71 // The gaffer is necessary here to make sure that the shadow camera follows the aircraft.
72 const gaffer = {
73     light: new THREE.DirectionalLight(0xffffdd, 1),
74     focus: grob,
75     range: 8,
76     update: function() {
77         this.light.position.copy(this.focus.model.position);
78         this.light.position.addScaledVector(sun_position, 5);
79         this.light.target.position.copy(this.focus.model.position);
80         this.light.target.updateMatrixWorld();
```

```
81      },
82      initialise: function() {
83          this.light.castShadow = true;
84          this.light.shadow.camera.left = -this.range;
85          this.light.shadow.camera.right = this.range;
86          this.light.shadow.camera.top = this.range;
87          this.light.shadow.camera.bottom = -this.range;
88          this.light.shadow.camera.near = 1;
89          this.light.shadow.camera.far = 1000;
90          this.light.shadow.camera.updateProjectionMatrix();
91      }
92  }
93
94  gaffer.initialise();
95  scene.add(gaffer.light);
96
97
98  const terrain = new Terrain(TERRAIN_JSON);
99
100
101 //-------------------------------------------------------------------------------------------\\
102 //                                          MAIN LOOP                                        \\
103 //-------------------------------------------------------------------------------------------\\
104
105
106 let mouse = {clientX: 0, clientY:0};
107
108 window.addEventListener("mousemove", (event) => {
109     mouse = event;
110 });
111
112
113 let keys = new Set();
114 window.addEventListener("keydown", (event) => {
115     keys.add(event.key);
116 });
117
118 window.addEventListener("keyup", (event => {
119     keys.delete(event.key);
120 }));
121
```

```
122
123 function run() {
124
125     if (grob.model) {
126         grob.update(keys, terrain.tree.header);
127
128         camera_man.update();
129         gaffer.update();
130     }
131
132     renderer.render(scene, camera_man.camera);
133     clock.update()
134     requestAnimationFrame(run);
135 }
136
137 run();
```

3.3 physical_interactions.js

```
import * as THREE from "three";
import {GLTFLoader} from "three/examples/jsm/Addons.js";
import {graham_scan, remove_duplicate_vector2s} from "./terminal_velocity.js";
import {GRAVITY, AIR_DENSITY, VISCOSITY_FACTOR, scene, clock} from "./constants.js";


const loader = new GLTFLoader();


class Model {
    constructor(file_path) {
        loader.load(file_path, (loaded_gltf) => {
            // This runs when the model loads.
            scene.add(loaded_gltf.scene);
            this.model = loaded_gltf.scene;
            this.model.traverse((object) => {
                if (object.isMesh) {
                    object.castShadow = true;
                    object.receiveShadow = true;
                }
            });
            this.loaded();
        } );
    }

    loaded() {
        // This will be overwritten by each child class.
    }
}

export class Aircraft extends Model {
    constructor(file_path, aircraft_information, parts_dictionary) {
        super(file_path);

        this.parts_dictionary = parts_dictionary;


        this.acceleration = new THREE.Vector3();
        this.velocity = new THREE.Vector3();
```

```
40
41          this.angular_acceleration = new THREE.Vector3();
42          this.angular_velocity = new THREE.Vector3();
43
44          this.resultant_force = new THREE.Vector3();
45          this.resultant_torque = new THREE.Vector3();
46
47          this.reaction_velocity = new THREE.Vector3();
48          this.reaction_angular_velocity = new THREE.Vector3();
49
50          this.throttle = 0;
51
52
53          this.MASS = aircraft_information.MASS;
54          this.FUEL_MASS = aircraft_information.FUEL_MASS;
55          this.TOTAL_MASS = this.MASS + this.FUEL_MASS;
56          this.WEIGHT = GRAVITY.clone().multiplyScalar(this.TOTAL_MASS);
57
58          this.DRAG_CORRECTIONAL_CONSTANT = aircraft_information.DRAG_CORRECTIONAL_CONSTANT;
59          this.CENTRE_OF_GRAVITY = aircraft_information.CENTRE_OF_GRAVITY;
60
61
62          this.plane = new THREE.Plane();
63          }
64
65
66      loaded() {
67          this.parts = new Array();
68          for (let part_dictionary_index in this.parts_dictionary) {
69              // This allows each part to be accessed by name from this, e.g. this.propeller.
70              this[this.parts_dictionary[part_dictionary_index].name] = new Aircraft_Part(this, this.parts_dictionary[
                    part_dictionary_index]);
71
72              this.parts.push(this[this.parts_dictionary[part_dictionary_index].name]);
73          }
74
75
76          this.drag_parts = this.parts.filter((part) => part.drag);
77          this.OBBs = this.parts.filter((part) => part.collision).map((part) => part.collision_box.OBB);
78          this.masses = this.parts.filter((part) => part.mass != null);
79          this.fuel_masses = this.parts.filter((part) => part.mass == 1);
```

```
80          this.aerodynamic_surfaces = this.parts.filter((part) => part.aerodynamic_parameters);
81
82
83          //-------------------------------------------------------------------------------------------\\
84          //                                    MOMENT OF INERTIA CALCULATIONS                         \\
85          //-------------------------------------------------------------------------------------------\\
86          this.volume = sum(this.masses.map((part) => part.find_volume()));
87          this.density = this.MASS / this.volume;
88          this.fuel_volume = sum(this.fuel_masses.map((part) => part.find_volume()));
89          this.fuel_density = this.FUEL_MASS / this.fuel_volume;
90
91          this.moments_of_inertia = [0, 0, 0];
92          for (let axis = 0; axis < 3; axis++) {
93              for (let mass of this.masses) {
94                  this.moments_of_inertia[axis] += mass.find_moment_of_inertia(axis) + mass.find_mass() * this.
                        CENTRE_OF_GRAVITY.clone().sub(mass.centre_of_gravity).setComponent(axis, 0).lengthSq();
95              }
96          }
97
98          this.reciprocal_moments_of_inertia_vector = new THREE.Vector3().fromArray(this.moments_of_inertia.map((
                moment_of_inertia) => 1 / moment_of_inertia));
99
100         //-------------------------------------------------------------------------------------------\\
101
102         this.update();
103
104
105         this.tree = new OBB_tree(this.OBBs);
106
107
108         this.model.position.set(0, 3, 0);
109     }
110
111
112     update(keys, terrain) {
113         let thrust_force = new THREE.Vector3(-this.throttle * 130000 * 0.8 / (this.velocity.length() + 0.1), 0, 0);
114         thrust_force.applyQuaternion(this.model.quaternion);
115
116         this.resultant_force = new THREE.Vector3().add(thrust_force).add(this.WEIGHT);
117
118         this.aerodynamic_surfaces.map((aerodynamic_surface) => aerodynamic_surface.find_forces());
```

```
119
120
121         this.plane.normal = this.velocity.clone().normalize();
122
123         let velocity_array = [Math.abs(this.velocity.x), Math.abs(this.velocity.y), Math.abs(this.velocity.z)];
124         this.plane_axis_x = new THREE.Vector3().crossVectors( // plane_axis_x and plane_axis_y are unit vectors, like x and
                y but in 3D space.
125             new THREE.Vector3().setComponent(velocity_array.indexOf(Math.min(Math.abs(this.velocity.x), Math.abs(this.
                    velocity.y), Math.abs(this.velocity.z))), 1), // Ensures that u is not identical to the velocity, or too
                    close
126             this.plane.normal
127         ).normalize();
128
129         this.plane_axis_y = new THREE.Vector3().crossVectors(this.plane_axis_x, this.plane.normal).normalize();
130
131         for (let drag_part of this.drag_parts) {
132             drag_part.update();
133         }
134
135         if (this.velocity.lengthSq() > 0) {
136             // Terminal velocity calculations
137             let area = this.calculate_drag_shape(this.drag_parts).area();
138             this.velocity.clampLength(0, Math.sqrt(2 * this.TOTAL_MASS * this.acceleration.length() /
139                 (this.wing_left_0.find_aerodynamic_coefficients()[1] +
140                 this.wing_left_1.find_aerodynamic_coefficients()[1] +
141                 this.wing_right_0.find_aerodynamic_coefficients()[1] +
142                 this.wing_right_1.find_aerodynamic_coefficients()[1] +
143                 this.horizontal_stabiliser.find_aerodynamic_coefficients()[1] +
144                 this.vertical_stabiliser.find_aerodynamic_coefficients()[1]) * area * this.DRAG_CORRECTIONAL_CONSTANT) *
                        AIR_DENSITY);
145         }
146
147         if (keys) {
148             this.keyboard_input(keys);
149         }
150
151         this.propellor.rotate(this.throttle);
152
153         this.acceleration = this.resultant_force.clone().divideScalar(this.TOTAL_MASS);
154
155         this.velocity.add(this.acceleration.clone().multiplyScalar(clock.delta_time));
```

```
156
157
158         let this_position = this.model.position.clone();
159         let next_position = this_position.clone().add(this.velocity.clone().multiplyScalar(clock.delta_time).add(this.
                acceleration.clone().multiplyScalar((clock.delta_time ** 2) * 0.5)));
160         this.model.position.copy(next_position);
161
162         this.model.updateMatrixWorld();
163
164         //-----------------------------------------------//
165         //           DO COLLISION CHECKS HERE            //
166         if (this.tree) {
167             for (let intersection of this.tree.intersects(terrain, this)) {
168                 intersection.aircraft_obb.apply_reaction_force(intersection.terrain_triangle.normal.clone().normalize());
169             }
170
171             this.velocity.add(this.reaction_velocity);
172             this.angular_velocity.add(this.reaction_angular_velocity);
173
174             this.reaction_velocity = new THREE.Vector3();
175             this.reaction_angular_velocity = new THREE.Vector3();
176         }
177         //-----------------------------------------------//
178
179
180         next_position = this_position.clone().add(this.velocity.clone().multiplyScalar(clock.delta_time).add(this.
                acceleration.clone().multiplyScalar((clock.delta_time ** 2) * 0.5)));
181         this.model.position.copy(next_position);
182
183
184         this.angular_acceleration = this.resultant_torque.clone().multiply(this.reciprocal_moments_of_inertia_vector);
185         this.angular_velocity.add(this.angular_acceleration.clone().multiplyScalar(clock.delta_time));
186
187         this.angular_velocity.clampLength(0, 5);
188         this.angular_velocity.multiplyScalar(0.95);
189
190         this.rotate_from_torque(this.angular_velocity.clone().multiplyScalar(clock.delta_time).add(this.
                angular_acceleration.clone().multiplyScalar((clock.delta_time ** 2) * 0.5)));
191
192         this.resultant_torque = new THREE.Vector3();
193     }
```

```
194
195
196     keyboard_input(keys) {
197         if (keys.has('w')) {
198             if (this.elevators.angle > -0.5) {
199                 this.elevators.rotate(-3.5 * clock.delta_time);
200             }
201         }
202
203         if (keys.has('s')) {
204             if (this.elevators.angle < 0.5) {
205                 this.elevators.rotate(3.5 * clock.delta_time);
206             }
207         }
208         if (!keys.has('w') && !keys.has('s')) {
209             this.elevators.rotate(this.elevators.angle * - 4 * clock.delta_time);
210         }
211
212         if (keys.has('a')) {
213             if (this.aileron_left.angle < 0.5) {
214                 this.aileron_left.rotate(3.5 * clock.delta_time);
215                 this.aileron_right.rotate(-3.5 * clock.delta_time);
216             }
217         }
218
219         if (keys.has('d')) {
220            if (this.aileron_left.angle > -0.5) {
221                 this.aileron_left.rotate(-3.5 * clock.delta_time);
222                 this.aileron_right.rotate(3.5 * clock.delta_time);
223             }
224         }
225
226         if (!keys.has('a') && !keys.has('d')) {
227             this.aileron_left.rotate(this.aileron_left.angle * - 4 * clock.delta_time);
228             this.aileron_right.rotate(this.aileron_right.angle * - 4 * clock.delta_time);
229         }
230
231         if (keys.has("ArrowUp")) {
232             this.throttle += 0.5 * clock.delta_time;
233         }
234
```

```
235         if (keys.has("ArrowDown")) {
236             this.throttle -= 0.5 * clock.delta_time;
237         }
238 
239         if (this.throttle > 1) {
240             this.throttle = 1;
241         } else if (this.throttle < 0) {
242             this.throttle = 0;
243         }
244 
245         if (keys.has("ArrowLeft")) {
246             if (this.rudder.angle > -0.5) {
247                 this.rudder.rotate(-3.5 * clock.delta_time);
248             }
249         }
250 
251         if (keys.has("ArrowRight")) {
252             if (this.rudder.angle < 0.5) {
253                 this.rudder.rotate(3.5 * clock.delta_time);
254             }
255         }
256 
257         // THESE INPUTS ARE FOR TESTING PURPOSES ONLY
258         /*
259         if (keys.has(' ')) {
260             this.throttle = 0;
261         }
262 
263         if (keys.has('.')) {
264             new Point(this.model.position);
265             console.log(this.model.position);
266         }
267         */
268     }
269 
270 
271     calculate_drag_shape(parts) {
272         if (parts.length > 1) {
273             let return_union = this.calculate_drag_shape(parts.slice(0, Math.floor(parts.length / 2))).union(this.
                    calculate_drag_shape(parts.slice(Math.floor(parts.length / 2))));
274             return return_union;
```

```
275         } else {
276             if (parts[0]) {
277                 let projection_polygon = parts[0].collision_box.terminal_velocity_calculations();
278                 return projection_polygon;
279             } else {
280                 return null;
281             }
282         }
283     }
284 
285 
286     rotate_from_torque(torque) {
287         let original_pivot_position = this.model.localToWorld(this.CENTRE_OF_GRAVITY.clone());
288 
289         this.model.quaternion.multiply(new THREE.Quaternion().setFromAxisAngle(torque.clone().normalize(), torque.length())
                );
290         this.model.updateMatrixWorld();
291 
292         let adjusted_pivot_position = this.model.localToWorld(this.CENTRE_OF_GRAVITY.clone());
293         original_pivot_position.sub(adjusted_pivot_position);
294 
295         this.model.position.add(original_pivot_position);
296 
297     }
298 }
299 
300 
301 export class Aircraft_Part {
302     constructor(parent, part_json) {
303         this.parent = parent;
304         this.name = part_json.name;
305         this.collision = part_json.collision;
306         this.drag = part_json.drag;
307         this.mass = part_json.mass;
308         // Centre of gravity is initially relative to the geometric centre of the bounding box of the aircraft part.
309         this.centre_of_gravity = part_json.centre_of_gravity;
310 
311         this.rotation_axis = part_json.rotation_axis;
312         this.pivot_point = part_json.pivot_point;
313         this.angle = 0;
314 
```

```
315         this.control_surface = part_json.control_surface;
316 
317         this.aerodynamic_parameters = part_json.aerodynamic_parameters;
318 
319         this.model = this.parent.model.getObjectByName(this.name);
320 
321 
322         if (this.collision || this.drag || this.rotation_axis) {
323             this.collision_box = new Collision_Bounding_Box(this, this.parent, this.model);
324 
325         }
326 
327 
328         if (this.mass != null) {
329             // The centre of gravity will now be relative to the centre of gravity of the aircraft.
330             this.centre_of_gravity.add(this.collision_box.AABB_centre).sub(this.parent.CENTRE_OF_GRAVITY);
331         }
332     }
333 
334 
335     rotate(angle) {
336         if (this.pivot_point && angle != 0) {
337             this.angle += angle;
338             this.model.position.sub(this.pivot_point);
339             this.model.position.applyAxisAngle(this.rotation_axis, angle);
340             this.model.position.add(this.pivot_point);
341             this.model.rotateOnAxis(this.rotation_axis, angle);
342         }
343     }
344 
345 
346     update() {
347         this.collision_box.update(this.parent);
348     }
349 
350 
351     find_volume() {
352         return this.collision_box.find_volume();
353     }
354 
355 
```

```
356     find_mass() {
357         return this.find_volume() * (this.parent.density + this.mass? this.parent.fuel_density: 0);
358     }
359 
360 
361     find_moment_of_inertia(axis) {
362         return this.find_mass() * (this.collision_box.AABB_size.getComponent((axis + 1) % 3) ** 2 + this.collision_box.
                AABB_size.getComponent((axis + 2) % 3) ** 2) / 12;
363     }
364 
365 
366     find_flap_chord_ratio() {
367         if (this.control_surface) {
368             if (this.control_surface.slice(0, 7) == "aileron") {
369                 return ((this.collision_box.AABB_centre.x + this.collision_box.AABB_size.x / 2) - (this.parent[this.
                        control_surface].collision_box.AABB_centre.x - this.parent[this.control_surface].collision_box.AABB_size
                        .x / 2)) / this.collision_box.AABB_size.x;
370             } else {
371                 return (this.parent[this.control_surface].collision_box.AABB_centre.x + this.parent[this.control_surface].
                        collision_box.AABB_size.x / 2 - this.parent[this.control_surface].pivot_point.x) / this.collision_box.
                        AABB_size.x;
372             }
373         } else {
374             return 0;
375         }
376     }
377 
378 
379     find_dimensions() {
380         if (this.control_surface) {
381             if (this.control_surface.slice(0, 6) == "rudder") {
382                 // The rudder is heavily slanted, so the aspect ratio must be calculated differently to the other aircraft
                        parts.
383 
384                 // This calculates the vertical height of the vertical stabiliser, from the bottom of the vertical
                        stabiliser to the top of the rudder.
385                 return [(this.parent[this.control_surface].collision_box.AABB_centre.y + this.parent[this.control_surface].
                        collision_box.AABB_size.y / 2 -
386                 (this.collision_box.AABB_centre.y - this.collision_box.AABB_size.y / 2)),
387                 //-------------------------------------------------
388 
```

```
389                 // This calculates the average depth of the vertical stabiliser, taking the average between the depth of
                        the vertical stabiliser and the rudder.
390                 (((this.parent[this.control_surface].collision_box.AABB_centre.x + this.parent[this.control_surface].
                        collision_box.AABB_size.x / 2 -
391                 (this.collision_box.AABB_centre.x - this.collision_box.AABB_size.x / 2)) +
392                 this.parent[this.control_surface].collision_box.AABB_size.x) / 2
393                 //-------------------------------------------------
394                 )];
395             } else {
396                 return [Math.max(this.collision_box.AABB_size.y, this.collision_box.AABB_size.z), (this.parent[this.
                        control_surface].collision_box.AABB_centre.x + this.parent[this.control_surface].collision_box.AABB_size
                        .x / 2 - (this.collision_box.AABB_centre.x - this.collision_box.AABB_size.x / 2))];
397             }
398         } else {
399             return [Math.max(this.collision_box.AABB_size.y, this.collision_box.AABB_size.z), this.collision_box.AABB_size.
                    x];
400         }
401     }
402
403     find_relative_velocity() {
404         let relative_velocity = this.parent.velocity.clone().add(this.parent.angular_velocity.clone().cross(this.
                centre_of_gravity)).negate();
405         return relative_velocity;
406
407     }
408
409     find_aerodynamic_coefficients() {
410         this.aerodynamic_parameters.control_surface_deflection = this.control_surface? this.parent[this.control_surface].
                angle: 0;
411         return find_aerodynamic_coefficients(this.aerodynamic_parameters);
412     }
413
414     find_forces() {
415         if (!this.aerodynamic_parameters.flap_chord_ratio) {
416             this.aerodynamic_parameters.flap_chord_ratio = this.find_flap_chord_ratio();
417             this.dimensions = this.find_dimensions();
418             this.aerodynamic_parameters.aspect_ratio = this.dimensions[0] / this.dimensions[1];
419             this.aerodynamic_parameters.area = multiply_array(this.dimensions);
420         }
421
422         let world_flow_velocity = this.find_relative_velocity();
```

```
        let local_flow_velocity = world_flow_velocity.clone().applyQuaternion(this.parent.model.quaternion.clone().invert()
            );
        local_flow_velocity.setZ(0);

        let pressure = AIR_DENSITY * local_flow_velocity.lengthSq() / 2;
        this.aerodynamic_parameters.angle_of_attack = degrees(Math.atan2(local_flow_velocity.y, local_flow_velocity.x));

        let aerodynamic_coefficients = this.find_aerodynamic_coefficients();

        let drag_direction = local_flow_velocity.clone().normalize().applyQuaternion(this.parent.model.quaternion);
        let lift_direction = drag_direction.clone().cross(new THREE.Vector3(0, 0, -1).applyQuaternion(this.parent.model.
            quaternion)).normalize();
        let torque_direction = new THREE.Vector3(0, 0, -1).applyQuaternion(this.parent.model.quaternion);

        let lift = lift_direction.clone().multiplyScalar(aerodynamic_coefficients[0] * pressure * this.
            aerodynamic_parameters.area);
        let drag = drag_direction.clone().multiplyScalar(aerodynamic_coefficients[1] * pressure * this.
            aerodynamic_parameters.area);
        let torque = torque_direction.clone().multiplyScalar(aerodynamic_coefficients[2] * pressure * this.
            aerodynamic_parameters.area * this.dimensions[1]);

        let total_force = lift.clone().add(drag);
        let total_torque = this.centre_of_gravity.clone().cross(total_force);

        this.parent.resultant_force.add(total_force);
        this.parent.resultant_torque.add(total_torque);

    }

    apply_reaction_force(normal) {

        // The following code sets the collision point to be the deepest of the intersecting points (or average of the
            deepest)
        // However, I found this to be a lot more jitter, so I abandoned it.

        /*
        let vertex_depth = 0;
        let vertex_depths = [];
        for (let vertex of this.collision_box.OBB.get_vertices()) {
            let current_depth = vertex.dot(normal);
            if (current_depth < vertex_depth) {
```

```
458             vertex_depth = current_depth;
459             vertex_depths = [vertex];
460         } else if (current_depth == vertex_depth) {
461             vertex_depths.push(vertex);
462         }
463     }
464
465     let collision_point = new THREE.Vector3();
466     for (let vertex of vertex_depths) {
467         collision_point.add(vertex);
468     }
469     collision_point.divideScalar(vertex_depths.length);
470     */
471
472
473     let relative_velocity = this.parent.velocity.clone().add(this.parent.angular_velocity.clone().cross(this.
        centre_of_gravity));
474
475     let relative_normal_velocity = relative_velocity.dot(normal);
476
477     // If the part is actually going into the ground (otherwise the aircraft could not take off)
478     if (relative_normal_velocity < 0) {
479
480         // e is the 'coefficient of restitution' - basically how bouncy the collision is (0 = inelastic, 1 = elastic).
481         let e = 0.1;
482
483         let collision_velocity = -(1 + e) * relative_velocity.dot(normal);
484         let collision_angular_velocity = this.centre_of_gravity.clone().cross(normal).multiply(this.parent.
            reciprocal_moments_of_inertia_vector).cross(this.centre_of_gravity);
485
486         let impulse = collision_velocity / ((1 / this.parent.TOTAL_MASS) + collision_angular_velocity.dot(normal));
487
488         // mu = coefficient of friction - basically how rough the ground is
489         let mu;
490         if (relative_velocity.length() < 20) { // Static Friction
491             // mu varies from 0.7 to 1
492             mu = 0.7 - 0.6 * relative_velocity.length() / 20;
493         } else { // Dynamic Friction
494             mu = 0.1;
495         }
496
```

```
497             // The relative velocity without the component parallel to the normal.
498             let friction_tangent = relative_velocity.clone().addScaledVector(normal, -relative_velocity.dot(normal));
499             if (friction_tangent.length() > 0.3) {
500                 let friction_direction = friction_tangent.clone().normalize().negate(); // Negated as friction opposes
                        motion.
501                 let friction_impulse = mu * impulse;
502
503
504                 this.parent.reaction_velocity.addScaledVector(friction_direction, friction_impulse / this.parent.TOTAL_MASS
                        );
505                 this.parent.reaction_angular_velocity.add(this.centre_of_gravity.clone().cross(friction_direction.clone().
                        multiplyScalar(friction_impulse)).multiply(this.parent.reciprocal_moments_of_inertia_vector));
506             }
507
508             this.parent.reaction_velocity.addScaledVector(normal, impulse / this.parent.TOTAL_MASS);
509
510             this.parent.reaction_angular_velocity.add(this.centre_of_gravity.clone().cross(normal.clone().multiplyScalar(
                    impulse)).multiply(this.parent.reciprocal_moments_of_inertia_vector));
511         }
512
513     }
514
515
516 }
517
518 function radians(degrees) {
519     return degrees * Math.PI / 180;
520 }
521
522 function degrees(radians) {
523     return radians * 180 / Math.PI;
524 }
525
526 function find_aerodynamic_coefficients(aerodynamic_parameter_json) {
527     // Eq. 3
528     const lift_curve_slope_finite = radians(aerodynamic_parameter_json.lift_curve_slope_2d * (aerodynamic_parameter_json.
            aspect_ratio / (aerodynamic_parameter_json.aspect_ratio + (2 * (aerodynamic_parameter_json.aspect_ratio + 4)) / (
            aerodynamic_parameter_json.aspect_ratio + 2))));
529
530     // Eq. 4
531     const flap_effectiveness_angle = Math.acos(2 * (aerodynamic_parameter_json.flap_chord_ratio) - 1);
```

```
532     const flap_effectiveness_factor = 1 - (degrees(flap_effectiveness_angle) - Math.sin(flap_effectiveness_angle)) / Math.
            PI;
533     const lift_coefficient_increment = lift_curve_slope_finite * flap_effectiveness_factor * VISCOSITY_FACTOR *
            aerodynamic_parameter_json.control_surface_deflection;
534
535     // Ref [20]
536     const max_lift_coefficient_increment = lift_coefficient_increment * (1 - (aerodynamic_parameter_json.flap_chord_ratio))
            ;
537
538     // Eq. 5
539     const zero_lift_angle = aerodynamic_parameter_json.zero_lift_angle_base - (lift_coefficient_increment /
            lift_curve_slope_finite);
540
541     // Eq. 6
542     const max_lift_coefficient_positive = lift_curve_slope_finite * (aerodynamic_parameter_json.stall_angle_positive_base -
             aerodynamic_parameter_json.zero_lift_angle_base) + max_lift_coefficient_increment;
543     const max_lift_coefficient_negative = lift_curve_slope_finite * (aerodynamic_parameter_json.stall_angle_negative_base -
             aerodynamic_parameter_json.zero_lift_angle_base) + max_lift_coefficient_increment;
544
545     // Eq. 7
546     const stall_angle_positive = zero_lift_angle + max_lift_coefficient_positive / lift_curve_slope_finite;
547     const stall_angle_negative = zero_lift_angle + max_lift_coefficient_negative / lift_curve_slope_finite;
548
549     let lift_coefficient, normal_force_coefficient, tangential_force_coefficient, induced_angle_of_attack,
            effective_angle_of_attack;
550     if (stall_angle_negative < aerodynamic_parameter_json.angle_of_attack && aerodynamic_parameter_json.angle_of_attack <
            stall_angle_positive) {
551         // Eq. 8
552         lift_coefficient = lift_curve_slope_finite * (aerodynamic_parameter_json.angle_of_attack - zero_lift_angle);
553
554         induced_angle_of_attack = lift_coefficient / (Math.PI * aerodynamic_parameter_json.aspect_ratio);
555
556         effective_angle_of_attack = aerodynamic_parameter_json.angle_of_attack - zero_lift_angle - induced_angle_of_attack;
557
558         tangential_force_coefficient = aerodynamic_parameter_json.skin_friction_coefficient * Math.cos(radians(
                effective_angle_of_attack));
559
560         normal_force_coefficient = (lift_coefficient + tangential_force_coefficient * Math.sin(radians(
                effective_angle_of_attack))) / Math.cos(radians(effective_angle_of_attack));
561     } else {
562         // Eq. 9
```

```
563         if (stall_angle_positive < aerodynamic_parameter_json.angle_of_attack && aerodynamic_parameter_json.angle_of_attack
                < 90) {
564             induced_angle_of_attack = (lift_curve_slope_finite * (stall_angle_positive - zero_lift_angle)) / (Math.PI *
                    aerodynamic_parameter_json.aspect_ratio * (stall_angle_positive - 90)) * (aerodynamic_parameter_json.
                    angle_of_attack - stall_angle_positive) + (lift_curve_slope_finite * (stall_angle_positive - zero_lift_angle
                    )) / (Math.PI * aerodynamic_parameter_json.aspect_ratio);
565         } else if (-90 < aerodynamic_parameter_json.angle_of_attack && aerodynamic_parameter_json.angle_of_attack <
                stall_angle_negative) {
566             induced_angle_of_attack = (lift_curve_slope_finite * (stall_angle_negative - zero_lift_angle)) / (Math.PI *
                    aerodynamic_parameter_json.aspect_ratio * (stall_angle_negative - 90)) * (aerodynamic_parameter_json.
                    angle_of_attack - stall_angle_negative) + (lift_curve_slope_finite * (stall_angle_negative - zero_lift_angle
                    )) / (Math.PI * aerodynamic_parameter_json.aspect_ratio);
567         } else {
568             induced_angle_of_attack = 0;
569         }
570 
571         effective_angle_of_attack = aerodynamic_parameter_json.angle_of_attack - zero_lift_angle - induced_angle_of_attack;
572 
573         const skin_friction_coefficient_beyond_stall = -0.0426 * aerodynamic_parameter_json.control_surface_deflection ** 2
                + 0.21 * aerodynamic_parameter_json.control_surface_deflection + 1.98;
574 
575         normal_force_coefficient = skin_friction_coefficient_beyond_stall * Math.sin(radians(effective_angle_of_attack)) *
                (1 / (0.56 + 0.44 * Math.sin(radians(mod(effective_angle_of_attack, 180)))) - 0.41 * (1 - Math.pow(Math.E, (-17
                / aerodynamic_parameter_json.aspect_ratio))));
576 
577         tangential_force_coefficient = 0.5 * aerodynamic_parameter_json.skin_friction_coefficient * Math.cos(radians(
                effective_angle_of_attack));
578 
579         lift_coefficient = normal_force_coefficient * Math.cos(radians(effective_angle_of_attack)) -
                tangential_force_coefficient * Math.sin(radians(effective_angle_of_attack));
580     }
581 
582     const drag_coefficient = normal_force_coefficient * Math.sin(radians(effective_angle_of_attack)) +
            tangential_force_coefficient * Math.cos(radians(effective_angle_of_attack));
583 
584     const moment_coefficient = -normal_force_coefficient * (0.25 - 0.175 * (1 - Math.abs(effective_angle_of_attack) / 90));
585 
586     return [lift_coefficient, drag_coefficient, moment_coefficient];
587 }
588 
589 function mod(dividend, divisor) {
```

```
590     let quotient = dividend / divisor;
591     return (quotient - Math.floor(quotient)) * divisor;
592 }
593 
594 class Collision_Bounding_Box {
595     constructor(part, parent, object_to_bound) {
596         this.part = part;
597         this.parent = parent;
598         this.object_to_bound = object_to_bound;
599 
600         this.AABB = new THREE.Box3(); // Axis Aligned Bounding Box - cannot be rotated, because it is defined by the world
                axes.
601         this.AABB.setFromObject(this.object_to_bound);
602 
603         this.AABB_size = new THREE.Vector3();
604         this.AABB.getSize(this.AABB_size); // The initial size of the AABB is recorded and assumed to be the desired one. (
                The size varies with rotation of the bounded object.)
605 
606         this.AABB_centre = new THREE.Vector3();
607 
608         this.AABB.setFromObject(this.object_to_bound);
609         this.AABB.getCenter(this.AABB_centre);
610 
611 
612         // Oriented Bounding Box - can be rotated, not constrained by world axes.
613         this.OBB = new OBB(
614             this.AABB_centre,
615             new THREE.Vector3(1, 0, 0).applyQuaternion(this.parent.model.quaternion),
616             new THREE.Vector3(0, 1, 0).applyQuaternion(this.parent.model.quaternion),
617             new THREE.Vector3(0, 0, 1).applyQuaternion(this.parent.model.quaternion),
618             this.AABB_size.x / 2,
619             this.AABB_size.y / 2,
620             this.AABB_size.z / 2,
621             this.part
622         );
623     }
624 
625     update(parent) {
626         this.OBB.track(parent);
627     }
628 
```

```
629     terminal_velocity_calculations() {
630         let projected_points = project_shape_onto_plane(this.OBB.mesh, this.parent.plane);
631         let cartesian_points = new Array();
632 
633         for (let i in projected_points) {
634             cartesian_points.push(convert_coplanar_points_to_2d_coordinates(projected_points[i], this.parent.plane_axis_x,
                    this.parent.plane_axis_y));
635         }
636 
637         let grahams_points = graham_scan(remove_duplicate_vector2s(cartesian_points));
638         return grahams_points;
639     }
640 
641     find_volume() {
642         return multiply_array(this.AABB_size.toArray());
643     }
644 }
645 
646 function project_shape_onto_plane(shape, plane) {
647     let shape_vertices = shape.geometry.attributes.position;
648     let return_vertices = new Array()
649     let shape_vector = new THREE.Vector3().fromBufferAttribute(shape_vertices, 0);
650 
651     for (let i = 0; i < 8; i++) {
652         // Formula I made for accessing the binary value of a at index b: (a % (2 ** (b + 1)) - a % (2 ** b)) / (2 ** b).
653         // (x - 0.5) * 2 maps {0,1} to {-1, 1}.
654         // The combination of these means that I can access every vertex.
655         let vertex_vector = new THREE.Vector3(
656             shape_vector.x * (((i - i % 4) / 4 - 0.5) * 2),
657             shape_vector.y * (((i % 4 - i % 2) / 2 - 0.5) * 2),
658             shape_vector.z * (((i % 2) - 0.5) * 2)
659         );
660         vertex_vector.applyMatrix4(shape.matrix);
661         let projected_point = new THREE.Vector3();
662         plane.projectPoint(vertex_vector, projected_point);
663         return_vertices.push(projected_point);
664     }
665     return return_vertices
666 }
667 
668 function convert_coplanar_points_to_2d_coordinates(point, u, v) {
```

```
669         return new THREE.Vector2(point.dot(u), point.dot(v));
670     }
671 
672 
673     export class Point { // A class which visually represents a single point in space for testing purposes only.
674         constructor(position=new THREE.Vector3(0, 0, 0), collection=scene) {
675             this.position = position;
676 
677             this.mesh = new THREE.Mesh(
678                 new THREE.SphereGeometry(5),
679                 new THREE.MeshBasicMaterial({color:"#cccc00"})
680             )
681 
682             this.set(this.position);
683 
684             collection.add(this.mesh);
685         }
686 
687         set(position) {
688             this.mesh.position.copy(position);
689         }
690 
691     }
692 
693     export class OBB {
694         constructor(centre = new THREE.Vector3(), axis_0 = new THREE.Vector3(1, 0, 0), axis_1 = new THREE.Vector3(0, 1, 0),
                axis_2 = new THREE.Vector3(0, 0, 1), extent_0 = 1, extent_1 = 1, extent_2 = 1, part = null) {
695             this.original_centre = centre.clone();
696             this.centre = centre.clone();
697 
698             this.original_axis_0 = axis_0.clone();
699             this.original_axis_1 = axis_1.clone();
700             this.original_axis_2 = axis_2.clone();
701             this.original_axes = [this.original_axis_0, this.original_axis_1, this.original_axis_2];
702 
703             this.axis_0 = axis_0.clone();
704             this.axis_1 = axis_1.clone();
705             this.axis_2 = axis_2.clone();
706             this.axes = [this.axis_0, this.axis_1, this.axis_2];
707 
708             this.extent_0 = extent_0;
```

```
709         this.extent_1 = extent_1;
710         this.extent_2 = extent_2;
711         this.extents = [this.extent_0, this.extent_1, this.extent_2];
712 
713         this.part = part;
714 
715         this.mesh = new THREE.Mesh(
716             new THREE.BoxGeometry(this.extent_0 * 2, this.extent_1 * 2, this.extent_2 * 2),
717             new THREE.MeshBasicMaterial({color: "#ffffff", side:THREE.DoubleSide, wireframe:true, visible:true})
718         );
719         let rotation_matrix = new THREE.Matrix4();
720         rotation_matrix.makeBasis(this.axis_0, this.axis_1, this.axis_2);
721         this.mesh.applyMatrix4(rotation_matrix);
722 
723         this.mesh.position.set(this.centre.x, this.centre.y, this.centre.z);
724         scene.add(this.mesh);
725     }
726 
727     set rotation(new_quaternion_rotation) {
728         this.axis_0 = this.original_axis_0.clone().applyQuaternion(new_quaternion_rotation);
729         this.axis_1 = this.original_axis_1.clone().applyQuaternion(new_quaternion_rotation);
730         this.axis_2 = this.original_axis_2.clone().applyQuaternion(new_quaternion_rotation);
731         this.axes = [this.axis_0, this.axis_1, this.axis_2];
732     }
733 
734     set position(new_centre) {
735         this.centre.copy(new_centre);
736     }
737 
738     intersects(intersecting_object, parent) {
739         if (parent) {
740             this.track(parent);
741         }
742 
743 
744         if (intersecting_object instanceof Triangle) {
745 
746             let D = intersecting_object.vertex_0.clone().sub(this.centre);
747 
748             // Intersection test for the normal of the triangle (N).
749             if (Math.abs(intersecting_object.normal.dot(D)) > this.extent_0 * Math.abs(intersecting_object.normal.dot(this.
```

```
    axis_0)) + this.extent_1 * Math.abs(intersecting_object.normal.dot(this.axis_1)) + this.extent_2 * Math.abs(
    intersecting_object.normal.dot(this.axis_2))) {
750     return false;
751 }
752
753 // Intersection tests for the axes of the OBB (axis_0, axis_1 and axis_2).
754
755 for (let axis = 0; axis < 3; axis++) {
756     if (this.axes[axis].dot(D) > this.extents[axis]) {
757         if (this.axes[axis].dot(intersecting_object.edge_0) >= 0) {
758             if (this.axes[axis].dot(intersecting_object.edge_1) >= 0) {
759                 return false;
760             } else if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_1) > this.extents[
                    axis]) {
761                 return false;
762             }
763         } else if (this.axes[axis].dot(intersecting_object.edge_1) <= this.axes[axis].dot(intersecting_object.
                edge_0)) {
764             if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_1) > this.extents[axis])
                {
765                 return false;
766             }
767         } else {
768             if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_0) > this.extents[axis])
                {
769                 return false;
770             }
771         }
772     } else if (this.axes[axis].dot(D) < -this.extents[axis]) {
773         if (this.axes[axis].dot(intersecting_object.edge_0) <= 0) {
774             if (this.axes[axis].dot(intersecting_object.edge_1) <= 0) {
775                 return false;
776             } else if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_1) < -this.extents
                    [axis]) {
777                 return false;
778             }
779         } else if (this.axes[axis].dot(intersecting_object.edge_1) >= this.axes[axis].dot(intersecting_object.
                edge_0)) {
780             if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_1) < -this.extents[axis])
                {
781                 return false;
```

```
782                 }
783             } else {
784                 if (this.axes[axis].dot(D) + this.axes[axis].dot(intersecting_object.edge_0) < -this.extents[axis])
                        {
785                     return false;
786                 }
787             }
788         }
789     }
790
791     // Intersection tests for the cross products between the edges of the triangle (edge_0, edge_1 and edge_2) and
        the axes of the OBB (axis_0, axis_1, axis_2).
792
793     for (let axis = 0; axis < 3; axis++) {
794         for (let edge = 0; edge < 3; edge++) {
795             let p = this.axes[axis].clone().cross(intersecting_object.edges[edge]).dot(D);
796
797             // (x + 1) % 2 - x maps {0, 1, 2} to {1, -1, -1}. I could have used x^2 - 3x + 1 instead, but I did not
                    for computational efficiency.
798             let d = ((edge + 1) % 2 - edge) * this.axes[axis].dot(intersecting_object.normal);
799
800             //((x + 1) % 2 - x) / 2 + 0.5 maps {0, 1, 2} to {1, 0, 0} and (x % 2 - x) / 2 + 2 maps {0, 1, 2} to {2,
                    2, 1}
801             let R = this.extents[((axis + 1) % 2 - axis) / 2 + 0.5] * Math.abs(this.axes[(axis % 2 - axis) / 2 +
                    2].dot(intersecting_object.edges[edge])) +
802                 this.extents[(axis % 2 - axis) / 2 + 2] * Math.abs(this.axes[((axis + 1) % 2 - axis) / 2 +
                        0.5].dot(intersecting_object.edges[edge]));
803
804             if (p > R) {
805                 if (d >= 0) {
806                     return false;
807                 } else if (p + d > R) {
808                     return false;
809                 }
810             } else if (p < -R) {
811                 if (d <= 0) {
812                     return false;
813                 } else if (p + d < -R) {
814                     return false;
815                 }
816             }
```

```
817             }
818         }
819         return true;
820 
821     // Else, the intersecting_object is an OBB or OBB_tree_node.
822     } else {
823         if (intersecting_object instanceof OBB_tree_node) {
824             intersecting_object = intersecting_object.OBB;
825         }
826         // D is centre of the intersecting OBB  relative to this OBBs centre.
827         let D = intersecting_object.centre.clone().sub(this.centre);
828 
829         // C is a matrix containing the dot products between all of the axes of the two OBB, so they do not need to be
                recalculated.
830         let C = new THREE.Matrix3();
831         for (let i = 0; i < 9; i++) {
832             C.elements[i] = this.axes[Math.floor(i / 3)].dot(intersecting_object.axes[i % 3]);
833         }
834 
835         // Intersection tests for the axes of this OBB (this.axis_0, this.axis_1 and this.axis_2).
836         for (let this_axis = 0; this_axis < 3; this_axis++) {
837             let R = Math.abs(this.axes[this_axis].dot(D));
838 
839             let R_0 = this.extents[this_axis];
840 
841             let R_1 =
842                 intersecting_object.extent_0 * Math.abs(C.elements[3 * this_axis]) +
843                 intersecting_object.extent_1 * Math.abs(C.elements[3 * this_axis + 1]) +
844                 intersecting_object.extent_2 * Math.abs(C.elements[3 * this_axis + 2]);
845 
846             if (R > R_0 + R_1) {
847                 return false;
848             }
849         }
850 
851         // Intersection tests for the axes of the intersecting OBB (intersecting_object.axis_0, intersecting_object.
                axis_1 and intersecting_object.axis_2).
852         for (let intersecting_axis = 0; intersecting_axis < 3; intersecting_axis++) {
853             let R = Math.abs(intersecting_object.axes[intersecting_axis].dot(D));
854 
855             let R_0 = intersecting_object.extents[intersecting_axis];
```

```
856
857                 let R_1 =
858                     this.extent_0 * Math.abs(C.elements[3 * intersecting_axis]) +
859                     this.extent_1 * Math.abs(C.elements[3 * intersecting_axis + 1]) +
860                     this.extent_2 * Math.abs(C.elements[3 * intersecting_axis + 2]);
861
862                 if (R > R_0 + R_1) {
863                     return false;
864                 }
865             }
866
867             // Intersection tests for the cross products between the axes of this OBB (this.axis_0, this.axis_1 and this.
                    axis_2) and the axes of the intersecting OBB (intersecting_object.axis_0, intersecting_object.axis_1 and
                    intersecting_object.axis_2).
868             for (let this_axis = 0; this_axis < 3; this_axis++) {
869                 for (let intersecting_axis = 0; intersecting_axis < 3; intersecting_axis++) {
870                     let R =
871                         Math.abs(C.elements[((this_axis + 1) % 3) * 3 + intersecting_axis] * this.axes[(this_axis + 2) %
                            3].dot(D) -
872                         C.elements[((this_axis + 2) % 3) * 3 + intersecting_axis] * this.axes[(this_axis + 1) % 3].dot(D));
873
874                     let R_0 =
875                         this.extents[((this_axis + 1) % 2 - this_axis) / 2 + 0.5] * Math.abs(C.elements[((this_axis % 2 -
                            this_axis) / 2 + 2) * 3 + intersecting_axis]) +
876                         this.extents[(this_axis % 2 - this_axis) / 2 + 2] * Math.abs(C.elements[(((this_axis + 1) % 2 -
                            this_axis) / 2 + 0.5) * 3 + intersecting_axis]);
877
878                     let R_1 =
879                         intersecting_object.extents[((intersecting_axis + 1) % 2 - intersecting_axis) / 2 + 0.5] * Math.abs
                            (C.elements[this_axis * 3 + (intersecting_axis % 2 - intersecting_axis) / 2 + 2]) +
880                         intersecting_object.extents[(intersecting_axis % 2 - intersecting_axis) / 2 + 2] * Math.abs(C.
                            elements[this_axis * 3 + ((intersecting_axis + 1) % 2 - intersecting_axis) / 2 + 0.5  ]);
881
882                     if (R > R_0 + R_1) {
883                         return false;
884                     }
885                 }
886             }
887             return true;
888         }
889     }
```

```
890
891
892     get_vertices() {
893         let vertices = new Array();
894         let edge_0 = this.axis_0.clone().multiplyScalar(this.extent_0);
895         let edge_1 = this.axis_1.clone().multiplyScalar(this.extent_1);
896         let edge_2 = this.axis_2.clone().multiplyScalar(this.extent_2);
897         let new_vertex;
898         for (let i = 0; i < 8; i++) {
899             new_vertex =
900                 // Formula I made for accessing the binary value of a at index b: (a % (2 ** (b + 1)) - a % (2 ** b)) / (2
                        ** b).
901                 // x - 0.5 * 2 maps {0,1} to {-1, 1}.
902                 edge_0.clone().multiplyScalar(((i % 2) - 0.5) * 2).add(
903                 edge_1.clone().multiplyScalar(((i % 4 - i % 2) / 2 - 0.5) * 2)).add(
904                 edge_2.clone().multiplyScalar(((i - i % 4) / 4 - 0.5) * 2)).add( // i % 8 is not necessary as i < 8.
905                 this.centre);
906             vertices.push(new_vertex);
907         }
908         return vertices;
909     }
910
911     track(parent) {
912         // This track method changes the position and rotation of mesh, then sets the centre and axes to be match it.
913         this.mesh.rotation.set(0, 0, 0);
914         let rotation_matrix = new THREE.Matrix4();
915         rotation_matrix.makeBasis(this.original_axis_0, this.original_axis_1, this.original_axis_2);
916         this.mesh.applyMatrix4(rotation_matrix);
917         this.mesh.position.copy(this.original_centre);
918         this.mesh.applyMatrix4(parent.model.matrix);
919         this.position = this.mesh.position;
920         this.rotation = this.mesh.quaternion;
921     }
922
923 }
924
925 class OBB_tree {
926     constructor(collision_objects) {
927         this.header = new OBB_tree_node(collision_objects);
928     }
929
```

```
930     track(parent) {
931         this.header.OBB.track(parent);
932     }
933 
934     intersects(intersecting_object, parent) {
935         return this.header.intersects(intersecting_object, parent);
936     }
937 }
938 
939 class OBB_tree_node {
940     constructor(collision_objects) {
941         this.OBB = make_OBB(collision_objects);
942         this.collision_objects = collision_objects;
943 
944         this.left_node = null;
945         this.right_node = null;
946 
947         this.subdivide();
948     }
949 
950     subdivide() {
951         if (this.collision_objects.length == 1) {
952             this.left_node = this.collision_objects[0];
953         } else {
954             let longest_axis = this.OBB.axes[this.OBB.extents.indexOf(Math.max(...this.OBB.extents))];
955             let projection_array = new Array();
956             for (let collision_object of this.collision_objects) {
957                 projection_array.push([collision_object, longest_axis.dot(collision_object.centre)]);
958             }
959 
960             projection_array.sort((a, b) => a[1] - b[1]); // Negatives are not sorted correctly unless specified.
961             let left_projections = projection_array.slice(0, Math.floor(projection_array.length / 2)).map((element) =>
                element[0]);
962             let right_projections = projection_array.slice(Math.floor(projection_array.length / 2)).map((element) =>
                element[0]);
963 
964             if (left_projections.length > 1) {
965                 this.left_node = new OBB_tree_node(left_projections);
966             } else if (left_projections.length == 1) {
967                 this.left_node = left_projections[0];
968             }
```

```
969
970             if (right_projections.length > 1) {
971                 this.right_node = new OBB_tree_node(right_projections);
972             } else if (right_projections.length == 1) {
973                 this.right_node = right_projections[0];
974
975             }
976         }
977     }
978
979     intersects(intersecting_object, parent) {
980         if (this.OBB.intersects(intersecting_object, parent)) {
981             if (intersecting_object instanceof Triangle) {
982                 let left_intersection, right_intersection;
983                 if (this.left_node) {
984                     left_intersection = this.left_node.intersects(intersecting_object, parent);
985                 }
986                 if (this.right_node) {
987                     right_intersection = this.right_node.intersects(intersecting_object, parent);
988                 }
989
990                 // The following === are necessary to ensure that left_intersection === true is only triggered which
                        left_intersection is the boolean 'true', not an array.
991                 if (left_intersection === true) { // If the left_node is an OBB which does intersect the triangle:
992                     left_intersection = [{"aircraft_obb": this.left_node.part, "terrain_triangle": intersecting_object}];
993                 } else if (left_intersection === false) { // If the left_node is an OBB which does not intersect the
                        triangle:
994                     left_intersection = [];
995                 }
996
997                 if (right_intersection === true) { // If the right_node is an OBB which does intersect the triangle:
998                     right_intersection = [{"aircraft_obb": this.right_node.part, "terrain_triangle": intersecting_object}];
999                 } else if (right_intersection === false) { // If the right_node is an OBB which does not intersect the
                        triangle:
1000                    right_intersection = [];
1001                }
1002
1003                return [...left_intersection, ...right_intersection];
1004
1005            } else { // The intersection is with a bounding box from the terrain tree.
1006                return [...interpret_collision(this.left_node, intersecting_object.right_node, parent), ...
```

```
                    interpret_collision(this.right_node, intersecting_object.left_node, parent), ...interpret_collision(this
                    .left_node, intersecting_object.left_node, parent), ...interpret_collision(this.right_node,
                    intersecting_object.right_node, parent)];
            }
        } else {
            return [];
        }
    }
}

function interpret_collision(this_object, intersecting_object, parent) {
    let collision = this_object.intersects(intersecting_object, parent);
    if (collision === true) {
        if (intersecting_object instanceof Triangle) {
            collision = [{"aircraft_obb": this_object.part, "terrain_triangle": intersecting_object}]; // This is passed
                through as JSON to prevent the values from being separated by the spread operator (...)
        } else {
            collision = [...interpret_collision(this_object, intersecting_object.left_node, parent), ...interpret_collision
                (this_object, intersecting_object.right_node, parent)] ;
        }
    } else if (collision === false) {
        collision = [];
    }
    return collision;
}


export class Triangle {
    constructor(vertex_0, vertex_1, vertex_2, material = new THREE.MeshStandardMaterial({color: "#ff0000", roughness: 0.9,
        metalness: 0.1})) {

        this.vertex_0 = vertex_0;
        this.vertex_1 = vertex_1;
        this.vertex_2 = vertex_2;
        this.vertices = [this.vertex_0, this.vertex_1, this.vertex_2];

        this.edge_0 = this.vertex_1.clone().sub(this.vertex_0);
        this.edge_1 = this.vertex_2.clone().sub(this.vertex_0);
        this.edge_2 = this.edge_1.clone().sub(this.edge_0)
        this.edges = [this.edge_0, this.edge_1, this.edge_2];

```

```
1042        this.normal = new THREE.Vector3().crossVectors(this.edge_0, this.edge_1);
1043
1044        this.mesh = new THREE.Mesh(
1045            new THREE.BufferGeometry().setFromPoints([this.vertex_0, this.vertex_1, this.vertex_2]),
1046            material
1047        );
1048        this.mesh.castShadow = true;
1049        this.mesh.receiveShadow = true;
1050        scene.add(this.mesh);
1051        this.mesh.geometry.computeVertexNormals();
1052    }
1053
1054    check_collision(model) {
1055        for (let part in model.parts) {
1056            if (model.parts[part].collision) {
1057                if (model.parts[part].collision_box.OBB.intersects(this)) {
1058                    model.parts[part].apply_reaction_force(this.normal.clone().normalize());
1059                    break;
1060                }
1061            }
1062        }
1063    }
1064
1065    move(new_position) {
1066        this.vertex_0.add(new_position);
1067        this.vertex_1.add(new_position);
1068        this.vertex_2.add(new_position);
1069        this.vertices = [this.vertex_0, this.vertex_1, this.vertex_2];
1070
1071        this.edge_0 = this.vertex_1.clone().sub(this.vertex_0);
1072        this.edge_1 = this.vertex_2.clone().sub(this.vertex_0);
1073        this.edge_2 = this.vertex_2.clone().sub(this.vertex_1)
1074        this.edges = [this.edge_0, this.edge_1, this.edge_2];
1075
1076        this.normal = new THREE.Vector3().crossVectors(this.edge_0, this.edge_1);
1077        this.mesh.position.add(new_position)
1078    }
1079
1080    area() {
1081        return new THREE.Vector3().crossVectors(this.edge_0, this.edge_1).length() / 2;
1082    }
```

```
1083
1084     get centre() {
1085         let centre = new THREE.Vector3();
1086         for (let i = 0; i < 3; i++) {
1087             centre.setComponent(i, (
1088                 this.vertex_0.getComponent(i) +
1089                 this.vertex_1.getComponent(i) +
1090                 this.vertex_2.getComponent(i)
1091             ) / 3)
1092         }
1093         return centre;
1094     }
1095
1096     get_vertices() {
1097         return this.vertices;
1098     }
1099
1100 }
1101
1102 export class Terrain {
1103     constructor(terrain_json) {
1104         let triangles = [];
1105         for (let triangle of terrain_json) {
1106             triangles.push(new Triangle(...triangle.vertices.map((coords) => new THREE.Vector3().fromArray(coords)),
                    triangle.material));
1107         }
1108
1109         this.tree = new OBB_tree(triangles);
1110     }
1111 }
1112
1113 //-------------------------------------------------------------------------------------------\\
1114 //                              USEFUL ARRAY MANIPULATION FUNCTIONS                          \\
1115 //-------------------------------------------------------------------------------------------\\
1116
1117 function sum_matrices(matrix_a, matrix_b) {
1118     return new THREE.Matrix3().fromArray(sum_arrays(matrix_a.toArray(), matrix_b.toArray()));
1119 }
1120
1121 function sum_arrays(array_a, array_b) {
1122     let total_array = new Array();
```

```
1123     for (let item in array_a) {
1124         total_array.push(array_a[item] + array_b[item]);
1125     }
1126     return total_array;
1127 }
1128
1129 function sum(array) {
1130     return array.reduce((total, current) => total + current);
1131 }
1132
1133 function multiply_array(array) {
1134     return array.reduce((total, current) => total * current);
1135 }
1136
1137
1138 //-------------------------------------------------------------------------------------------\\
1139 //                                  OBB CREATION FUNCTIONS                                   \\
1140 //-------------------------------------------------------------------------------------------\\
1141
1142 export function make_OBB(collision_objects) {
1143
1144     let object_vertices = new Array();
1145     for (let object of collision_objects) {
1146         object_vertices.push(...object.get_vertices());
1147     }
1148     if (collision_objects[0] instanceof OBB) {
1149         collision_objects = object_vertices;
1150     }
1151     let objects_centre = global_centre(object_vertices);
1152
1153
1154     let objects_covariance_matrix = covariance_matrix(object_vertices, objects_centre);
1155
1156     let eigenvalues = find_eigenvalues(objects_covariance_matrix);
1157     let eigenvectors = find_eigenvectors(objects_covariance_matrix, eigenvalues);
1158
1159     let extents = find_extents(object_vertices, eigenvectors, objects_centre);
1160     let final_extents = extents.map((element) => (element[1] - element[0]) / 2); // The distance between the extents,
         halved, as these extents are necessary for OBB creation.
1161     let final_centre = extents.map(((element) => (element[1] + element[0]) / 2)); // The middle point of the extents.
1162
```

```
1163     let rotation_matrix = new THREE.Matrix4().makeBasis(...eigenvectors);
1164
1165     // The current final_centre is a representation of how far along each eigenvector axis the final_centre is, relative to
             objects_centre.
1166     // However we want to find how far along each world axis the final_centre the centre of the OBB is, relative to the
             world origin.
1167     // So create a matrix which converts the final_centre into world-defined coordinates, and add the objects_centre to
             find the final world centre of the OBB.
1168     final_centre = new THREE.Vector3(...final_centre).applyMatrix4(rotation_matrix).add(objects_centre);
1169     return new OBB(final_centre, ...eigenvectors, ...final_extents);
1170
1171 }
1172
1173 function global_centre(objects) {
1174     let centre_vector = new THREE.Vector3();
1175     if (objects[0] instanceof Triangle) {
1176         let object_areas = objects.map((object) => object.area());
1177         let total_object_area = sum(object_areas);
1178         let object_centres = objects.map((object) => object.centre);
1179         for (let i = 0; i < 3; i++) {
1180             let component_sum = 0;
1181             for (let object in objects) {
1182                 component_sum += object_areas[object] * object_centres[object].getComponent(i);
1183             }
1184             centre_vector.setComponent(i, component_sum / total_object_area);
1185         }
1186         return centre_vector;
1187     } else if (objects[0] instanceof THREE.Vector3) {
1188         let centre_vector = new THREE.Vector3();
1189         for (let i = 0; i < 3; i++) {
1190             let component_sum = 0;
1191             for (let object_point of objects) {
1192                 component_sum += object_point.getComponent(i);
1193             }
1194             centre_vector.setComponent(i, component_sum / objects.length);
1195         }
1196         return centre_vector;
1197     }
1198 }
1199
1200 function covariance_matrix(objects, centre_vector) {
```

```
1201    let object_matrix_sum;
1202    if (objects[0] instanceof Triangle) {
1203        for (let object in objects) {
1204            let displacement_from_centre = objects[object].centre.clone().sub(centre_vector);
1205            let object_matrix = new THREE.Matrix3(
1206                displacement_from_centre.x * displacement_from_centre.x, displacement_from_centre.y *
                        displacement_from_centre.x, displacement_from_centre.z * displacement_from_centre.x,
1207                displacement_from_centre.x * displacement_from_centre.y, displacement_from_centre.y *
                        displacement_from_centre.y, displacement_from_centre.z * displacement_from_centre.y,
1208                displacement_from_centre.x * displacement_from_centre.z, displacement_from_centre.y *
                        displacement_from_centre.z, displacement_from_centre.z * displacement_from_centre.z,
1209            );
1210            object_matrix.multiplyScalar(objects[object].area());
1211            if (object_matrix_sum) {
1212                object_matrix_sum = sum_arrays(object_matrix_sum, object_matrix.toArray()); // THREE.Matrix3 does not have
                        an add method, so we have to be slightly more creative.
1213            } else {
1214                object_matrix_sum = object_matrix.toArray();
1215            }
1216        }
1217        return new THREE.Matrix3().fromArray(object_matrix_sum);
1218    } else if (objects[0] instanceof THREE.Vector3) {
1219        for (let object_point of objects) {
1220            let displacement_from_centre = object_point.clone().sub(centre_vector);
1221            let object_matrix = new THREE.Matrix3(
1222                displacement_from_centre.x * displacement_from_centre.x, displacement_from_centre.y *
                        displacement_from_centre.x, displacement_from_centre.z * displacement_from_centre.x,
1223                displacement_from_centre.x * displacement_from_centre.y, displacement_from_centre.y *
                        displacement_from_centre.y, displacement_from_centre.z * displacement_from_centre.y,
1224                displacement_from_centre.x * displacement_from_centre.z, displacement_from_centre.y *
                        displacement_from_centre.z, displacement_from_centre.z * displacement_from_centre.z,
1225            );
1226            if (object_matrix_sum) {
1227                object_matrix_sum = sum_arrays(object_matrix_sum, object_matrix.toArray()); // THREE.Matrix3 does not have
                        an add method, so we have to be slightly more creative.
1228            } else {
1229                object_matrix_sum = object_matrix.toArray();
1230            }
1231        }
1232        return new THREE.Matrix3().fromArray(object_matrix_sum);
1233    }
```

```
}

function find_eigenvalues(matrix_to_decompose) {
    let [a, b, c, d, e, f, g, h, i] = matrix_to_decompose.toArray();
    return cubic_roots(
        -1,
        (a + e + i),
        (- a * e - a * i - e * i + f * h + b * d + c * g),
        (a * e * i - a * f * h - b * d * i + b * f * g + c * d * h - c * g * e)
    );
}

function cubic_roots(a, b, c, d) {
    const CUBE_ROOTS_OF_UNITY = [new THREE.Vector2(1, 0), new THREE.Vector2(-0.5, Math.sqrt(3) / 2), new THREE.Vector2
        (-0.5, -Math.sqrt(3) / 2)]
    let solutions = new Array();
    let [p, q] = depress_cubic(a, b, c, d);

    let discriminant = (q / 2) ** 2 + (p / 3) ** 3;
    if (discriminant < 0) {
        discriminant = new THREE.Vector2(0, Math.sqrt(Math.abs(discriminant)));
    } else {
        discriminant = new THREE.Vector2(Math.sqrt(discriminant), 0);
    }
    let positive_discriminant_complex = discriminant.clone();
    positive_discriminant_complex.x += (-q / 2);
    let negative_discriminant_complex = discriminant.clone().negate();
    negative_discriminant_complex.x += (-q / 2);

    let positive_discriminant_complex_cube_root = complex_cube_root(positive_discriminant_complex)[0];
    let negative_discriminant_complex_cube_root = complex_cube_root(negative_discriminant_complex)[0];

    let solution_positive_discriminant, solution_negative_discriminant, solution;
    for (let i = 0; i < 3; i++) {
        solution_positive_discriminant = imaginary_foil(positive_discriminant_complex_cube_root.clone(),
            CUBE_ROOTS_OF_UNITY[i]);
        // 2 - ((i + 2) % 3) maps (0, 1, 2) to (0, 2, 1).
        solution_negative_discriminant = imaginary_foil(negative_discriminant_complex_cube_root.clone(),
            CUBE_ROOTS_OF_UNITY[2 - ((i + 2) % 3)]);
        solution = new THREE.Vector2().addVectors(solution_positive_discriminant, solution_negative_discriminant);
        solution.x += (- b / (3 * a));
```

```
        if (Math.abs(solution.y) < 0.0001) {
            solutions.push(solution.x);
        } else {
            solutions.push(NaN);
        }
    }

    return solutions;
}

function depress_cubic(a, b, c, d) {
    let p =
        ( - (b ** 2) + 3 * a * c)
                    /
                (3 * (a ** 2))
    ;

    let q =
        (2 * (b ** 3) - 9 * a * b * c + 27 * (a ** 2) * d)
                                /
                        (27 * (a ** 3))
    ;
    return [p, q];
}

function complex_cube_root(complex_number) {
    let r = complex_number.length();
    let theta = Math.atan(complex_number.y / complex_number.x);
    let cube_root_r = Math.cbrt(r);
    let complex_cube_roots = new Array();
    for (let index = 0; index < 3; index++) {
        let cube_root_theta = (theta + index * 2 * Math.PI) / 3;
        complex_cube_roots.push(new THREE.Vector2(Math.sign(complex_number.x) * cube_root_r * Math.cos(cube_root_theta),
            Math.sign(complex_number.x) * cube_root_r * Math.sin(cube_root_theta)));
    }
    return complex_cube_roots;
}

function imaginary_foil(vector_a, vector_b) { // Complex First-Outside-Inside-Last (FOIL) - a method of expanding brackets.
    return new THREE.Vector2(vector_a.x * vector_b.x - vector_a.y * vector_b.y, vector_a.x * vector_b.y + vector_a.y *
        vector_b.x);
```

```
}

function find_eigenvectors(eigenvector_matrix, eigenvalues) {
    let eigenvectors = new Array()
    for (let eigenvalue in eigenvalues) {
        if (eigenvalues[eigenvalue] == NaN)  {
            eigenvectors.push(NaN);
        } else {
            let subtracted_matrix = sum_matrices(eigenvector_matrix.clone(), new THREE.Matrix3().multiplyScalar(-
                eigenvalues[eigenvalue]));
            let [a, b, c, d, e, f, g, h, i] = subtracted_matrix.toArray();
            let x, y, z;
            x = 1;
            y = ((a * f - d * c) / (e * c - b * f));
            z = (-a - b * y) / c;
            if (isNaN(y) || !isFinite(y)) {
                y = 1;
                x = (b * f - c * e) / (c * d - a * f);
                z = (-b - a * x) / c;
            }
            if (isNaN(z) || !isFinite(z)) {
                z = 1;
                x = (c * e - f * b) / (d * b - a * e);
                y = (-c - a * x) / b;
            }
            eigenvectors.push(new THREE.Vector3(x, y, z).normalize());
        }
    }
    return eigenvectors;
}
// The Gram Schmidt process was an algorithm I used to ensure that 3 vectors were orthonormal, but I ended up making the
    find_eigenvectors() method more robust instead.
/*
function gram_schmidt_process(vectors) {
    let [initial_vector_0, initial_vector_1, initial_vector_2] = vectors.map((vector) => vector.clone());
    let final_vector_0 = initial_vector_0;
    let normalised_vector_0 = final_vector_0.normalize();
    let final_vector_1 = initial_vector_1.sub(project(initial_vector_1, final_vector_0));
    let normalised_vector_1 = final_vector_1.normalize();
    let final_vector_2 = initial_vector_2.sub(project(initial_vector_2, final_vector_0)).sub(project(initial_vector_2,
        final_vector_1));
```

```
    let normalised_vector_2 = final_vector_2.normalize();
    return [normalised_vector_0, normalised_vector_1, normalised_vector_2];
}
*/

function project(this_vector, onto_this_vector) {
    return onto_this_vector.clone().multiplyScalar(onto_this_vector.dot(this_vector) / (onto_this_vector.lengthSq()))
}

function find_extents(vertices, eigenvectors, centre) {
    let extents = [[null, null], [null, null], [null, null]];
    let current_extent;
    for(let vertex of vertices) {
        for (let i = 0; i < 3; i++) {
            current_extent = eigenvectors[i].dot(vertex.clone().sub(centre));
            if (extents[i][0] == null || current_extent < extents[i][0]) {
                extents[i][0] = current_extent;
            }
            if (extents[i][1] == null || current_extent > extents[i][1]) {
                extents[i][1] = current_extent;
            }
        }
    }

    return extents;
}

// I considered trying to find eigenvectors by turning the matrix into row reduced echelon form
/*
function find_row_reduced_echelon_form(matrix3_to_reduce) {
    let reduced_matrix3 = matrix3_to_reduce.clone()
    console.log(matrix3_to_reduce);
    swap_rows(reduced_matrix3, 0, 2);
    if (reduced_matrix3.elements[0] == 0) {
        if (reduced_matrix3.elements[1] == 0) {
            if (reduced_matrix3.elements[1] == 0) {
                return reduced_matrix3;
            } else {
                reduced_matrix3 = swap_rows(reduced_matrix3, 0, 2);
            }
        } else {
```

```
1390            reduced_matrix3 = swap_rows(reduced_matrix3, 0, 1);
1391        }
1392    }
1393
1394
1395    for (let i = 2; i > -1; i--) { // This is done in reverse to make sure that reduced_matrix3.elements[0] can be used
        multiple times to divide the other values in the first row.
1396        reduced_matrix3.elements[i * 3] /= reduced_matrix3.elements[0];
1397    }
1398
1399    for (let i = 2; i > -1; i--) {
1400        reduced_matrix3.elements[i * 3 + 1] -= reduced_matrix3.elements[i * 3] * reduced_matrix3.elements[1] /
            reduced_matrix3.elements[0];
1401    }
1402
1403    for (let i = 2; i > -1; i--) {
1404        reduced_matrix3.elements[i * 3 + 2] -= reduced_matrix3.elements[i * 3] * reduced_matrix3.elements[2] /
            reduced_matrix3.elements[0];
1405    }
1406
1407
1408
1409    for (let i = 2; i > 0; i--) {
1410        reduced_matrix3.elements[i * 3 + 1] /= reduced_matrix3.elements[4];
1411    }
1412
1413    for (let i = 2; i > 0; i--) {
1414        reduced_matrix3.elements[i * 3] -= reduced_matrix3.elements[i * 3 + 1] * reduced_matrix3.elements[3] /
            reduced_matrix3.elements[4];
1415    }
1416
1417    for (let i = 2; i > 0; i--) {
1418        reduced_matrix3.elements[i * 3 + 2] -= reduced_matrix3.elements[i * 3 + 1] * reduced_matrix3.elements[5] /
            reduced_matrix3.elements[4];
1419    }
1420
1421
1422
1423    for (let i = 2; i > 0; i--) {
1424        reduced_matrix3.elements[i * 3 + 2] /= reduced_matrix3.elements[8];
1425    }
```

```
1426
1427     for (let i = 2; i > 0; i--) {
1428         reduced_matrix3.elements[i * 3] -= reduced_matrix3.elements[i * 3 + 2] * reduced_matrix3.elements[6] /
                 reduced_matrix3.elements[8];
1429     }
1430
1431     for (let i = 2; i > 0; i--) {
1432         reduced_matrix3.elements[i * 3 + 1] -= reduced_matrix3.elements[i * 3 + 2] * reduced_matrix3.elements[7] /
                 reduced_matrix3.elements[8];
1433     }
1434     console.log(reduced_matrix3);
1435     return reduced_matrix3;
1436 }
1437
1438 function swap_rows(matrix3_to_swap, row_a, row_b) {
1439     let swapped_matrix3 = matrix3_to_swap.clone();
1440     for (let i = 0; i < 3; i++) {
1441         swapped_matrix3.elements[i * 3 + row_a] = matrix3_to_swap.elements[i * 3 + row_b];
1442         swapped_matrix3.elements[i * 3 + row_b] = matrix3_to_swap.elements[i * 3 + row_a];
1443     }
1444     return swapped_matrix3;
1445 }
1446 */
1447 //-------------------------------------------------------------------------------------------\\
```

3.4 constants.js

```
import * as THREE from "three";

//-----------------------------------------------------------------------------------------------\\
//                                              OBJECTS                                          \\
//-----------------------------------------------------------------------------------------------\\


export const scene = new THREE.Scene();


export const clock = {
    delta_time: 0.05,
    last_frame: performance.now(),
    update: function () {
        this.delta_time = (performance.now() - this.last_frame) / 1000;
        this.last_frame = performance.now();
    }
};


function print(this_array) {
    let print_string = "";
    for (let i in this_array) {
        //print_string += "\n" + String(this_array[i].x) + ";" + String(this_array[i].y);
        if (this_array[i].isVector2) {
            print_string += "\n" + "(" + String(this_array[i].x) + "," + String(this_array[i].y) + ")";
        } else {
            print_string += "\n" + "(" + String(this_array[i].x) + "," + String(this_array[i].y) + "," + String(this_array[
                i].z) + ")";

        }
    }
    console.log(print_string);

}

//-----------------------------------------------------------------------------------------------\\
//                                              CONSTANTS                                        \\
//-----------------------------------------------------------------------------------------------\\
```

```
39
40
41 export const GRAVITY = new THREE.Vector3(0, -9.81, 0);
42 export const AIR_DENSITY = 1.225;
43 export const VISCOSITY_FACTOR = 0.7;
44
45
46 export const GROB_FILE_PATH = "grob_tutor.glb";
47
48 export const GROB_INFORMATION = {
49     MASS: 670, // The empty mass of the Grob G 115 aircraft is  670kg.
50     FUEL_MASS: 72, // The maximum mass of fuel in a Grob G 115 aircraft is 72kg.
51     DRAG_CORRECTIONAL_CONSTANT: 12.21 / 15.613733935552318, // The wings should have an area of 12.21, but by my
           calculations they have an area of 15.6
52     CENTRE_OF_GRAVITY: new THREE.Vector3(-3.9075379371643066 + 2.480 + 0.243, -0.15, 0)
53 }
54
55 // This JSON contains information about how each part of the aircraft should be interpreted, which is used by the
       Aircraft_Part object.
56 export const GROB_PARTS = [
57     // "fuselage" refers to the entire fuselage of the aircraft. The subsequent fuselages are sections of the fuselage.
58     // This is so that the entire fuselage can be used for frontal area calculations (using too many aircraft parts for
           this hits the maximum recursion depth)
59     // and the individual fuselage sections can be used for collision detection and more precise collision response.
60     {
61         name: "fuselage",
62         collision: false,
63         drag: true,
64         rotation_axis: null,
65         pivot_point: null,
66         mass: null,
67         centre_of_gravity: null,
68         control_surface: null,
69         aerodynamic_parameters: null,
70     },
71     {
72         name: "fuselage_0",
73         collision: true,
74         drag: false,
75         rotation_axis: null,
76         pivot_point: null,
```

```
77          mass: 0,
78          centre_of_gravity: new THREE.Vector3(0.2, 0, 0),
79          control_surface: null,
80          aerodynamic_parameters: null,
81      },
82      {
83          name: "fuselage_1",
84          collision: true,
85          drag: true,
86          rotation_axis: null,
87          pivot_point: null,
88          mass: 0,
89          centre_of_gravity: new THREE.Vector3(0, -0.1, 0),
90          control_surface: null,
91          aerodynamic_parameters: null,
92      },
93      {
94          name: "fuselage_2",
95          collision: true,
96          drag: false,
97          rotation_axis: null,
98          pivot_point: null,
99          mass: 0,
100         centre_of_gravity: new THREE.Vector3(0.2, 0, 0),
101         control_surface: null,
102         aerodynamic_parameters: null,
103     },
104
105     {
106         name: "horizontal_stabiliser",
107         collision: true,
108         drag: true,
109         rotation_axis: null,
110         pivot_point: null,
111         mass: 0,
112         centre_of_gravity: new THREE.Vector3(0, 0.1, 0),
113         control_surface: "elevators",
114         aerodynamic_parameters: {"lift_curve_slope_2d": 4.044040292177923, "skin_friction_coefficient": 0.001161, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 17, "stall_angle_negative_base": -17},
115     },
116     {
```

```
117         name: "vertical_stabiliser",
118         collision: true,
119         drag: true,
120         rotation_axis: null,
121         pivot_point: null,
122         mass: 0,
123         centre_of_gravity: new THREE.Vector3(0.2, -0.1, 0),
124         control_surface: "rudder",
125         aerodynamic_parameters: {"lift_curve_slope_2d": 3.648601659088268, "skin_friction_coefficient": 0.00935, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 10, "stall_angle_negative_base": -10},
126     },
127
128     {
129         name: "wheel_mount_front",
130         collision: true,
131         drag: true,
132         rotation_axis: null,
133         pivot_point: null,
134         mass: 0,
135         control_surface: null,
136         centre_of_gravity: new THREE.Vector3(-0.1, -0.2, 0),
137         control_surface: null,
138         aerodynamic_parameters: null,
139     },
140     {
141         name: "wheel_mount_left",
142         collision: true,
143         drag: true,
144         rotation_axis: null,
145         pivot_point: null,
146         mass: 0,
147         centre_of_gravity: new THREE.Vector3(-0.2, -0.2, 0.25),
148         control_surface: null,
149         aerodynamic_parameters: null,
150     },
151     {
152         name: "wheel_mount_right",
153         collision: true,
154         drag: true,
155         rotation_axis: null,
156         pivot_point: null,
```

```
157         mass: 0,
158         centre_of_gravity: new THREE.Vector3(-0.2, -0.2, -0.25),
159         control_surface: null,
160         aerodynamic_parameters: null,
161     },
162     // Similar to the fuselage, the entirety of each wing is used for the drag
163     // whereas the subsequent wing sections are used for collision detection
164     // and chiefly physics, as wing_left_0 does not have an aileron, whereas wing_left_1 does,
165     // thereby varying the respective aerodynamic coefficients.
166     {
167         name: "wing_left",
168         collision: false,
169         drag: true,
170         rotation_axis: null,
171         pivot_point: null,
172         mass: null,
173         centre_of_gravity: null,
174         control_surface: null,
175         aerodynamic_parameters: null,
176     },
177     {
178         name: "wing_left_0",
179         collision: true,
180         drag: false,
181         rotation_axis: null,
182         pivot_point: null,
183         mass: 1,
184         centre_of_gravity: new THREE.Vector3(0, 0, -0.1),
185         control_surface: null,
186         aerodynamic_parameters: {"lift_curve_slope_2d": 6.152033224533429, "skin_friction_coefficient": 0.00781, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 10.5, "stall_angle_negative_base": -10.5},
187     },
188     {
189         name: "wing_left_1",
190         collision: true,
191         drag: false,
192         rotation_axis: null,
193         pivot_point: null,
194         mass: 1,
195         centre_of_gravity: new THREE.Vector3(0, 0, -0.1),
196         control_surface: "aileron_left",
```

```
197         aerodynamic_parameters: {"lift_curve_slope_2d": 6.152033224533429, "skin_friction_coefficient": 0.00781, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 10.5, "stall_angle_negative_base": -10.5},
198     },
199
200     {
201         name: "wing_right",
202         collision: false,
203         drag: true,
204         rotation_axis: null,
205         pivot_point: null,
206         mass: null,
207         centre_of_gravity: null,
208         control_surface: null,
209         aerodynamic_parameters: null,
210     },
211     {
212         name: "wing_right_0",
213         collision: true,
214         drag: false,
215         rotation_axis: null,
216         pivot_point: null,
217         mass: 1,
218         centre_of_gravity: new THREE.Vector3(0, 0, 0.1),
219         control_surface: null,
220         aerodynamic_parameters: {"lift_curve_slope_2d": 6.152033224533429, "skin_friction_coefficient": 0.00781, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 10.5, "stall_angle_negative_base": -10.5},
221     },
222     {
223         name: "wing_right_1",
224         collision: true,
225         drag: false,
226         rotation_axis: null,
227         pivot_point: null,
228         mass: 1,
229         centre_of_gravity: new THREE.Vector3(0, 0, 0.1),
230         control_surface: "aileron_right",
231         aerodynamic_parameters: {"lift_curve_slope_2d": 6.152033224533429, "skin_friction_coefficient": 0.00781, "
                zero_lift_angle_base": 0, "stall_angle_positive_base": 10.5, "stall_angle_negative_base": -10.5},
232     },
233
234     {
```

```
        name: "propellor",
        collision: true,
        drag: false,
        rotation_axis: new THREE.Vector3(1, 0, 0),
        pivot_point: new THREE.Vector3(0, 0, -0.178),
        mass: 0,
        centre_of_gravity: new THREE.Vector3(0, 0, 0.178),
        control_surface: null,
        aerodynamic_parameters: null,
    },

    // I based the following pivots point of off the control surface pivots  of the aircraft in this video: https://youtu.
        be/lEDbuLzkFFY?t=112.
    {
        name: "aileron_left",
        collision: false,
        drag: false,
        rotation_axis: new THREE.Vector3(-0.086, -1, 0.07).normalize(),
        pivot_point: new THREE.Vector3(-0.49, 0, -0.6),
        mass: null,
        centre_of_gravity: null,
        control_surface: null,
        aerodynamic_parameters: null,
    },
    {
        name: "aileron_right",
        collision: false,
        drag: false,
        rotation_axis: new THREE.Vector3(0.086, -1, -0.07).normalize(),
        pivot_point: new THREE.Vector3(-0.49, 0, -0.6),
        mass: null,
        centre_of_gravity: null,
        control_surface: null,
        aerodynamic_parameters: null,
    },

    {
        name: "elevators",
        collision: false,
        drag: false,
        rotation_axis: new THREE.Vector3(0, 1, 0).clone().negate(),
```

```
275         pivot_point: new THREE.Vector3(3.1, 0, 0),
276         mass: null,
277         centre_of_gravity: null,
278         control_surface: null,
279         aerodynamic_parameters: null,
280     },
281 
282     {
283         name: "rudder",
284         collision: false,
285         drag: false,
286         rotation_axis: new THREE.Vector3(0, 0, 1),
287         pivot_point: new THREE.Vector3(3.6, 0, 0),
288         mass: null,
289         centre_of_gravity: null,
290         control_surface: null,
291         aerodynamic_parameters: null,
292     },
293 
294     {
295         name: "wheel_front",
296         collision: true,
297         drag: false,
298         rotation_axis: null,
299         pivot_point: null,
300         mass: 0,
301         centre_of_gravity: new THREE.Vector3(0, 0, 0),
302         control_surface: null,
303         aerodynamic_parameters: null,
304     },
305     {
306         name: "wheel_left",
307         collision: true,
308         drag: false,
309         rotation_axis: null,
310         pivot_point: null,
311         mass: 0,
312         centre_of_gravity: new THREE.Vector3(0, 0, 0),
313         control_surface: null,
314         aerodynamic_parameters: null,
315     },
```

```
316     {
317         name: "wheel_right",
318         collision: true,
319         drag: false,
320         rotation_axis: null,
321         pivot_point: null,
322         mass: 0,
323         centre_of_gravity: new THREE.Vector3(0, 0, 0),
324         control_surface: null,
325         aerodynamic_parameters: null,
326     },
327 
328     {
329         name: "internals",
330         collision: false,
331         drag: false,
332         rotation_axis: null,
333         pivot_point: null,
334         mass: null,
335         centre_of_gravity: null,
336         control_surface: null,
337         aerodynamic_parameters: null,
338     }
339 ];
340 
341 
342 const asphalt = new THREE.MeshStandardMaterial({color: "#807E78", roughness: 0.7, metalness: 0.3});
343 const straw =  new THREE.MeshStandardMaterial({color: "#d8cd89", roughness: 0.9, metalness: 0.1});
344 const grass = new THREE.MeshStandardMaterial({color: "#669b2f", roughness: 0.9, metalness: 0.1});
345 const steel = new THREE.MeshStandardMaterial({color: "#CED2D7", roughness: 0.1, metalness: 0.9,});
346 
347 // I calculated where each of the following triangles should go:
348 export const TERRAIN_JSON = [
349     {
350         vertices: [
351             [-800, 0.001, -20],
352             [-800, 0.001, 20],
353             [200, -0.001, 20],
354         ],
355         material: asphalt,
356     },
```

```
    {
        vertices: [
            [-800, 0.001, -20],
            [200, 0.001, 20],
            [200, -0.001, -20],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [0, 0.001, -20],
            [40, 0.001, -20],
            [30, -0.001, -240],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [40, 0.001, -20],
            [70, 1, -220],
            [30, -0.001, -240],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [10, 0.001, -250],
            [130, 0.001, -190],
            [50, -0.001, -330],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [130, 0.001, -190],
            [170, 0.001, -270],
            [50, -0.001, -330],
        ],
        material: asphalt,
    },
    {
```

```
398         vertices: [
399             [-760, 0.001, -710],
400             [40, 0.001, -310],
401             [50, -0.001, -330],
402         ],
403         material: asphalt,
404     },
405     {
406         vertices: [
407             [-750, 0.001, -730],
408             [-760, 0.001, -710],
409             [50, -0.001, -330],
410         ],
411         material: asphalt,
412     },
413     {
414         vertices: [
415             [-50, 0.001, -380],
416             [-30, 0.001, -370],
417             [70, -0.001, -620],
418         ],
419         material: asphalt,
420     },
421     {
422         vertices: [
423             [-30, 0.001, -370],
424             [90, 0.001, -610],
425             [70, -0.001, -620],
426         ],
427         material: asphalt,
428     },
429     {
430         vertices: [
431             [90, 0.001, -610],
432             [100, 0.001, -630],
433             [-630, -0.001, -970],
434         ],
435         material: asphalt,
436     },
437     {
438         vertices: [
```

```
            [-630, 0.001, -970],
            [100, 0.001, -630],
            [-620, -0.001, -990],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [-750, 0.001, -730],
            [-730, 0.001, -720],
            [-630, -0.001, -970],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [-730, 0.001, -720],
            [-610, 0.001, -960],
            [-630, -0.001, -970],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [-370, 0.001, -540],
            [-410, 0.001, -860],
            [-390, -0.001, -550],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [-370, 0.001, -540],
            [-390, 0.001, -850],
            [-410, -0.001, -860],
        ],
        material: asphalt,
    },
    {
        vertices: [
            [40, 0.001, -20],
```

```
            [200, 0.001, -20],
            [200, -0.001, -80],
        ],
        material: straw,
    },
    {
        vertices: [
            [40, 0.001, -20],
            [200, 0.001, -80],
            [49, -0.001, -80],
        ],
        material: straw,
    },
    {
        vertices: [
            [-100, 0.001, -80],
            [0, 0.001, -20],
            [8.19, -0.001, -80],
        ],
        material: straw,
    },
    {
        vertices: [
            [-100, 0.001, -20],
            [0, 0.001, -20],
            [-100, -0.001, -80],
        ],
        material: straw,
    },
    {
        vertices: [
            [-50, 0.001, -80],
            [8.19, 0.001, -80],
            [-50, -0.001, -240],
        ],
        material: grass,
    },
    {
        vertices: [
            [-50, 0.001, -240],
            [8.19, 0.001, -80],
```

```
            [30, -0.001, -240],
        ],
        material: grass,
    },
    {
        vertices: [
            [5, 0.001, -240],
            [30, 0.001, -240],
            [10, -0.001, -250],
        ],
        material: grass,
    },
    {
        vertices: [
            [49, 0.001, -80],
            [200, 0.001, -80],
            [130, -0.001, -190],
        ],
        material: grass,
    },
    {
        vertices: [
            [49, 0.001, -80],
            [130, 0.001, -190],
            [70, -0.001, -220],
        ],
        material: grass,
    },
    {
        vertices: [
            [-850, 0.001, 20],
            [200, 0.001, 50],
            [200, -0.001, 20],
        ],
        material: grass,
    },
    {
        vertices: [
            [-850, 0.001, 20],
            [-850, 0.001, 50],
            [200, -0.001, 50],
```

```
562         ],
563         material: grass,
564     },
565     {
566         vertices: [
567             [130, 0.001, -190],
568             [200, 0.001, -80],
569             [200, -0.001, -270],
570         ],
571         material: grass,
572     },
573     {
574         vertices: [
575             [130, 0.001, -190],
576             [200, 0.001, -270],
577             [170, -0.001, -270],
578         ],
579         material: grass,
580     },
581     {
582         vertices: [
583             [200, 0.001, 50],
584             [250, 0.001, 50],
585             [200, -0.001, -270],
586         ],
587         material: grass,
588     },
589     {
590         vertices: [
591             [250, 0.001, 50],
592             [250, 0.001, -270],
593             [200, -0.001, -270],
594         ],
595         material: grass,
596     },
597     {
598         vertices: [
599             [170, 0.001, -270],
600             [250, 0.001, -270],
601             [-30, -0.001, -370],
602         ],
```

```
603         material: grass,
604     },
605     {
606         vertices: [
607             [-30, 0.001, -370],
608             [250, 0.001, -270],
609             [100, -0.001, -630],
610         ],
611         material: grass,
612     },
613     {
614         vertices: [
615             [-30, 0.001, -370],
616             [250, 0.001, -270],
617             [100, -0.001, -630],
618         ],
619         material: grass,
620     },
621     {
622         vertices: [
623             [250, 0.001, -270],
624             [250, 0.001, -630],
625             [100, -0.001, -630],
626         ],
627         material: grass,
628     },
629     {
630         vertices: [
631             [100, 0.001, -630],
632             [250, 0.001, -630],
633             [-620, -0.001, -990],
634         ],
635         material: grass,
636     },
637     {
638         vertices: [
639             [-620, 0.001, -990],
640             [250, 0.001, -630],
641             [250, -0.001, -990],
642         ],
643         material: grass,
```

```
644     },
645     {
646         vertices: [
647             [-370, 0.001, -540],
648             [-50, 0.001, -380],
649             [70, -0.001, -620],
650         ],
651         material: grass,
652     },
653     {
654         vertices: [
655             [-370, 0.001, -540],
656             [70, 0.001, -620],
657             [-390, -0.001, -850],
658         ],
659         material: grass,
660     },
661     {
662         vertices: [
663             [-730, 0.001, -720],
664             [-390, 0.001, -550],
665             [-410, -0.001, -860],
666         ],
667         material: grass,
668     },
669     {
670         vertices: [
671             [-730, 0.001, -720],
672             [-410, 0.001, -860],
673             [-610, -0.001, -960],
674         ],
675         material: grass,
676     },
677     {
678         vertices: [
679             [-760, 0.001, -710],
680             [-620, 0.001, -990],
681             [-760, -0.001, -990],
682         ],
683         material: grass,
684     },
```

```
685     {
686         vertices: [
687             [-760, 0.001, -710],
688             [5, 0.001, -240],
689             [40, -0.001, -310],
690         ],
691         material: grass,
692     },
693     {
694         vertices: [
695             [-760, 0.001, -710],
696             [-50, 0.001, -240],
697             [5, -0.001, -240],
698         ],
699         material: grass,
700     },
701     {
702         vertices: [
703             [-760, 0.001, -710],
704             [-50, 0.001, -80],
705             [-50, -0.001, -240],
706         ],
707         material: grass,
708     },
709     {
710         vertices: [
711             [-760, 0.001, -710],
712             [-100, 0.001, -80],
713             [-50, -0.001, -80],
714         ],
715         material: grass,
716     },
717     {
718         vertices: [
719             [-760, 0.001, -710],
720             [-100, 0.001, -20],
721             [-100, -0.001, -80],
722         ],
723         material: grass,
724     },
725     {
```

```
726         vertices: [
727             [-760, 0.001, -710],
728             [-760, 0.001, -20],
729             [-100, -0.001, -20],
730         ],
731         material: grass,
732     },
733     {
734         vertices: [
735             [-850, 0.001, -990],
736             [-850, 0.001, -20],
737             [-760, -0.001, -20],
738         ],
739         material: grass,
740     },
741     {
742         vertices: [
743             [-850, 0.001, -990],
744             [-760, 0.001, -20],
745             [-760, -0.001, -990],
746         ],
747         material: grass,
748     },
749     {
750         vertices: [
751             [-850, 0.001, -20],
752             [-850, 0.001, 20],
753             [-800, -0.001, 20],
754         ],
755         material: grass,
756     },
757     {
758         vertices: [
759             [-850, 0.001, -20],
760             [-800, 0.001, 20],
761             [-800, -0.001, -20],
762         ],
763         material: grass,
764     },
765     {
766         vertices: [
```

```
            [-850, 0.001, -20],
            [-800, 0.001, 20],
            [-800, -0.001, -20],
        ],
        material: grass,
    },
    {
        vertices: [
            [-850, 0.001, -1050],
            [-850, 0.001, -990],
            [250, -0.001, -990],
        ],
        material: grass,
    },
    {
        vertices: [
            [-850, 0.001, -1050],
            [250, 0.001, -990],
            [250, -0.001, -1050],
        ],
        material: grass,
    },
    {
        vertices: [
            [-30, 0, -60],
            [-30, 0, -100],
            [-50, 50, -80],
        ],
        material: steel,
    },
    {
        vertices: [
            [-30, 0, -60],
            [-50, 50, -80],
            [-70, 0, -60],
        ],
        material: steel,
    },
    {
        vertices: [
            [-70, 0, -60],
```

```
            [-50, 50, -80],
            [-70, 0, -100],
        ],
        material: steel,
    },
    {
        vertices: [
            [-30, 0, -100],
            [-70, 0, -100],
            [-50, 50, -80],
        ],
        material: steel,
    },
    {
        vertices: [
            [10000, -5, 10000],
            [10000, -5, -10000],
            [-20000, -5, 0],
        ],
        material: grass,
    },

];

export const NORMALISED_X_VECTOR = new THREE.Vector3(1, 0, 0);
export const NORMALISED_Y_VECTOR = new THREE.Vector3(0, 1, 0);
export const NORMALISED_Z_VECTOR = new THREE.Vector3(0, 0, 1);
```

3.5 terminal_velocity

```
    import * as THREE from "three";

class Polygonal_Circular_Doubly_Linked_List {
    constructor(vertices) {
        if (vertices && vertices.length > 0) {
            this.header = new Polygon_Node(vertices[0], null);

            for (let vertex = 1; vertex < vertices.length; vertex++) {
                this.push_node(vertices[vertex]);
            }
        } else {
            this.header = null
        }

        this.thresholds_designated = false;
        this.is_next_transition_an_entry = null;

    }

    intersect_polygon(intersecting_polygon) {
        this.header.intersect_edge(intersecting_polygon.header);
        if (this.header.next_polygon) {
            this.header.next_polygon.intersect_polygon(intersecting_polygon);
        }
    }

    push_node(coords) {
        if (this.header) {
            this.header.push_node(coords);
        } else {
            this.header = new Polygon_Node(coords, null);
        }
    }

    pop_node() {
        if (this.header) {
            if (this.header.next_node == this.header) {
                this.header = null;
            } else {
```

```
                this.header.pop_node();
            }
        }
    }

    designate_thresholds(intersecting_polygon) {
        if (!this.thresholds_designated) {
            this.thresholds_designated = true;
            let ray = new THREE.Vector2(Math.random() - 0.5, Math.random() - 0.5).normalize();

            let ray_intersects = this.header.cast_ray(ray, intersecting_polygon.header);
            if (ray_intersects % 2 == 0) { // If the header is outside the intersecting polygon:
                this.is_next_transition_an_entry = true;
            } else {
                this.is_next_transition_an_entry = false;
            }
            this.header.designate_thresholds(this.is_next_transition_an_entry);

            if (!intersecting_polygon.thresholds_designated) {
                intersecting_polygon.designate_thresholds(this);
            }
            if (this.header.next_polygon) {
                this.header.next_polygon.designate_thresholds(intersecting_polygon);
            }
        }

    }

    union(intersecting_polygon) {

        if (intersecting_polygon) {

            this.intersect_polygon(intersecting_polygon);

            this.designate_thresholds(intersecting_polygon);

            let this_polygon = this.sub_union();

            if (this_polygon) {
                this_polygon.next_polygon_push(intersecting_polygon.find_stragglers());
            } else {
```

```
                this_polygon = intersecting_polygon.find_stragglers();
            }
            return this_polygon;
        } else {
            return this.clone()
        }
    }

    sub_union() {
        if (!this.header.visited) {
            let new_polygon = this.traverse();
            if (this.header.next_polygon) {
                let new_next_polygon = this.header.next_polygon.sub_union();
                if (new_next_polygon) {
                    if (new_polygon) {
                        new_polygon.next_polygon_push(new_next_polygon);
                    } else {
                        new_polygon = new_next_polygon;
                    }
                }
            }
            return new_polygon;
        } else {
            if (this.header.next_polygon) {
                return this.header.next_polygon.sub_union()
            } else {
                return null;
            }
        }

    }

    find_stragglers() {
        let return_polygon = null;
        if (!this.header.visited && this.is_next_transition_an_entry) {
            return_polygon = this.clone()
        }
        if (this.header.next_polygon) {
            let new_next_polygon = this.header.next_polygon.find_stragglers();
            if (new_next_polygon) {
                if (return_polygon) {
```

```
                    return_polygon.next_polygon_push(new_next_polygon);
                } else {
                    return_polygon = new_next_polygon;
                }
            }
        }
        return return_polygon
    }

    traverse() {
        let next_intersection = this.header.find_unvisited_node(true);
        if (next_intersection) { // If there is an intersection:
            return next_intersection.traverse(next_intersection.is_entry); // The next intersection is guaranteed to be in
                the union, whereas the header might not be.
        } else if (this.is_next_transition_an_entry) { // If this polygon fully contains the intersecting polygon or does
            not overlap at all:
            return this.clone();
        } else { // If this polygon is fully inside the intersecting polygon:
            return null;
        }
    }

    next_polygon_push(new_next_polygon) {
        if (this.header.next_polygon) {
            this.header.next_polygon.next_polygon_push(new_next_polygon);
        } else {
            this.header.next_polygon = new_next_polygon;
        }
    }
    area() {
        if (this.header.next_polygon) {
            return 0.5 * Math.abs(this.header.area()) + this.header.next_polygon.area();
        } else {
            return 0.5 * Math.abs(this.header.area());
        }
    }

    access(index) {
        return this.header.access(index);
    }

```

```
    clone() {
        let clone_polygon = new Polygonal_Circular_Doubly_Linked_List()
        this.header.clone(clone_polygon);
        return clone_polygon;
    }

    to_array() {
        let array_to = new Array();
        this.header.to_array(array_to);
        return array_to
    }


}

class Polygon_Node {
    constructor(coords, previous_node, alpha = 2, intersection_neighbour = null) {

        this.coords = coords;
        this.x = coords.x;
        this.y = coords.y;
        this.alpha = alpha;

        if (previous_node) { // If this is not the first item in the list
            this.set_previous_node = previous_node;
            this.set_next_node = this.previous_node.next_node;
            this.next_node.set_previous_node = this;
            this.index = previous_node.index + 1;
        } else {
            this.set_previous_node = this;
            this.set_next_node = this;
            this.index = 0;
        }

        this.is_intersection = this.alpha != 2;
        this.intersection_neighbour = intersection_neighbour;
        if (this.intersection_neighbour) {
            this.intersection_neighbour.intersection_neighbour = this;
        }

        this.is_entry = undefined;
```

```
202         this.visited = false;
203         this.next_polygon = null;
204     }
205 
206     set set_next_node(next_node_to_set) {
207         this.next_node = next_node_to_set;
208         if (!next_node_to_set.is_intersection) {
209             this.next_proper_node = next_node_to_set;
210         }
211     }
212 
213     set set_previous_node(previous_node_to_set) {
214         this.previous_node = previous_node_to_set;
215         if (!previous_node_to_set.is_intersection) {
216             this.previous_proper_node = previous_node_to_set;
217         }
218     }
219 
220     push_node(coords) { // Adds a node to the end of the list.
221         if (this.next_proper_node.index == 0) { // If this node is at the end of the list:
222             this.set_next_node = new Polygon_Node(coords, this);
223         } else {
224             this.next_proper_node.push_node(coords);
225         }
226     }
227     pop_node() {
228         if (this.next_node.index == 0) { // If this node is at the end of the list:
229             this.kill()
230         } else {
231             this.next_proper_node.pop_node();
232         }
233     }
234 
235     insert_node(coords, alpha = 2, intersection_neighbour = null) { // Adds a node next to this one
236         this.set_next_node = new Polygon_Node(coords, this, alpha, intersection_neighbour);
237     }
238 
239     access(index) {
240         if (index == 0) {
241             return this
242         } else if (index < 0) {
```

```
            return this.previous_proper_node.access(index + 1);
        } else if (index > 0) {
            return this.next_proper_node.access(index - 1);
        }
    }
    add_intersection(coords, alpha, intersection_neighbour=null) {
        if (this.next_node.alpha >= alpha) {
                this.insert_node(coords, alpha, intersection_neighbour);
                return this.next_node;
            } else {
                return this.next_node.add_intersection(coords, alpha, intersection_neighbour);
            }
    }

    find_intersection(A, B, C, D, raycasting = false) { // Finds the intersection between A-B and C-D.
        // I worked out the simultaneous equation for the point of intersection. t represents how far along the edge
            between this node and the next node the point of intersection is.
        let A_B = new THREE.Vector2().subVectors(B, A);
        let C_D = new THREE.Vector2().subVectors(D, C);

        if (Math.abs(A_B.x) == Math.abs(C_D.x) || Math.abs(A_B.y) == Math.abs(C_D.y) || A_B.x + A_B.y == 0 || C_D.x + C_D.y
             == 0) { // If A-B and C-D are parallel or equal to (0,0):
            return [NaN, NaN, null];
        }


        let t =
            ((C.y - A.y) * (C_D.x) + (A.x - C.x) * (C_D.y))
                                    /
            ((A_B.y) * (C_D.x) - (A_B.x) * (C_D.y));

        let s =
            ((A.y - C.y) * (A_B.x) + (C.x - A.x) * (A_B.y))
                                    /
            ((C_D.y) * (A_B.x) - (C_D.x) * (A_B.y));

        /* I decided not to use the following, as, if C_D is vertical, s = Infinity, which is a little bit messy.
            let s = (A.x - C.x + t * (A_B.x))
                                /
                        (C_D.x);
        */
```

```
282
283         if (raycasting) {
284             return [t, s, null];
285         } else {
286             let intersection_coords = null
287             if (t >= 0 && t <= 1 && s >= 0 && s <= 1) { // If the point of intersection lies between within the edge`s
                    bounds:
288                 intersection_coords = new THREE.Vector2().addVectors(A, A_B.clone().multiplyScalar(t));
289             }
290             return [t, s, intersection_coords];
291         }
292     }
293
294     intersect_edge(intersecting_node) {
295         const [t, s, intersection_coords] = this.find_intersection(
296             this.coords,
297             this.next_proper_node.coords,
298             intersecting_node.coords,
299             intersecting_node.next_proper_node.coords,
300         );
301         if (intersection_coords) {
302             intersecting_node.add_intersection(intersection_coords, s, this.add_intersection(intersection_coords, t));
303         }
304
305         if (intersecting_node.next_proper_node.index == 0) { // If the intersecting node is the final node of the other
                polygon
306             if (this.next_proper_node.index == 0) { // If this node is the final one of this polygon:
307                 if (intersecting_node.next_proper_node.next_polygon) { // If there is a disjoint polygon in the
                        intersecting shape:
308                     this.next_proper_node.intersect_edge(intersecting_node.next_proper_node.next_polygon.header);
309                 }
310                 // The end of the intersection testing for this node.
311             } else {
312                 this.next_proper_node.intersect_edge(intersecting_node.next_proper_node); // An intersect test between the
                        next node and the header of the other polygon.
313             }
314
315         } else {
316             this.intersect_edge(intersecting_node.next_proper_node);
317         }
318     }
```

```
319
320     cast_ray(ray, intersecting_node, collisions = 0) {
321         const [t, s] = this.find_intersection(
322             this.coords,
323             new THREE.Vector2().addVectors(this.coords, ray),
324             intersecting_node,
325             intersecting_node.next_proper_node,
326             true
327         );
328
329         let total_collisons = (t >= 0 && s >= 0 && s <= 1)? collisions + 1: collisions;
330
331         if (intersecting_node.next_proper_node.index == 0) {
332             if (intersecting_node.next_proper_node.next_polygon)
333                 return this.cast_ray(ray, intersecting_node.next_proper_node.next_polygon.header, total_collisons);
334             else {
335                 return total_collisons;
336             }
337         } else {
338             return this.cast_ray(ray, intersecting_node.next_proper_node, total_collisons);
339         }
340     }
341
342     designate_thresholds(is_next_transition_an_entry) {
343         if (this.is_intersection) {
344             this.is_entry = is_next_transition_an_entry;
345             is_next_transition_an_entry = !is_next_transition_an_entry;
346         }
347         if (this.next_node.index != 0) {
348             this.next_node.designate_thresholds(is_next_transition_an_entry);
349         }
350     }
351
352     find_unvisited_node(needs_to_be_intersection=false) {
353         if (!this.visited && (!needs_to_be_intersection || this.is_intersection)) {
354             return this;
355         } else if (this.next_node.index == 0) {
356             if (this.next_node.next_polygon) {
357                 this.next_node.next_polygon.header.find_unvisited_node(needs_to_be_intersection);
358             } else {
359                 return null;
```

```
360             }
361         } else {
362             return this.next_node.find_unvisited_node(needs_to_be_intersection);
363         }
364     }
365
366
367     traverse(current_is_entry, new_polygon = new Polygonal_Circular_Doubly_Linked_List()) {
368         if (this.visited) {
369             return new_polygon;
370         } else {
371             this.visited = true;
372             if (!this.is_intersection || this.intersection_neighbour.visited) {
373                 new_polygon.push_node(this.coords);
374                 return current_is_entry? this.previous_node.traverse(current_is_entry, new_polygon) : this.next_node.
                        traverse(current_is_entry, new_polygon);
375             } else {
376                 return this.intersection_neighbour.traverse(this.intersection_neighbour.is_entry, new_polygon);
377             }
378         }
379     }
380
381     area() {
382         if (this.next_proper_node.index != 0) {
383             return this.x * this.next_proper_node.y - this.next_proper_node.x * this.y + this.next_proper_node.area();
384         } else {
385             return this.x * this.next_proper_node.y - this.next_proper_node.x * this.y;
386         }
387     }
388
389     graham_scan() {
390         if (this.index > 0) {
391             // We can use the right hand rule to determine if the next node is clockwise or not.
392             let anticlockwise = this.coords.clone().sub(this.previous_proper_node.coords).cross(this.next_proper_node.
                    coords.clone().sub(this.coords)) > 0;
393             if (!anticlockwise) {
394                 this.kill();
395                 return true;
396             }
397             if (this.next_proper_node.index != 0) {
398                 if (this.next_proper_node.graham_scan()) {
```

```
                    this.graham_scan()
                }
            }
        } else {
            if (this.next_proper_node.graham_scan()) {
                this.graham_scan()
            }
        }
    }

    kill() {
        this.next_node.set_previous_node = this.previous_node;
        this.previous_node.set_next_node = this.next_node;
    }

    clone(clone_polygon) {
        clone_polygon.push_node(this.coords);
        if (this.next_proper_node.index != 0) {
            this.next_proper_node.clone(clone_polygon);
        }
    }

    to_array(array_to) {
        array_to.push(this.coords);
        if (this.next_proper_node.index != 0) {
            this.next_proper_node.to_array(array_to);
        }
    }

    distance_to_next_node() {
        return new THREE.Vector2().subVectors(this.next_proper_node.coords, this.coords).length();
    }
}


export function remove_duplicate_vector2s(vector2_array) {
    let cleaned_vector2_array = new Array();
    let rounded_vector2_array = new Array();
    for (let vector2 in vector2_array) {
        let rounded_coords = new THREE.Vector2(vector2_array[vector2].x.toFixed(10), vector2_array[vector2].y.toFixed(10));
        cleaned_vector2_array.push(vector2_array[vector2]);
```

```
440         rounded_vector2_array.push(rounded_coords);
441         if (rounded_vector2_array.length > 1) {
442             for (let rounded_vector2 in rounded_vector2_array.slice(0, -1)) {
443                 if (rounded_coords.x == rounded_vector2_array[rounded_vector2].x && rounded_coords.y ==
                        rounded_vector2_array[rounded_vector2].y) {
444                     cleaned_vector2_array.pop();
445                     rounded_vector2_array.pop();
446                     break;
447                 }
448             }
449         }
450     }
451     return cleaned_vector2_array;
452 }
453
454 export function graham_scan(points) {
455
456     let lowest_point = points[0];
457
458     for (let point = 1; point < points.length; point++) {
459         // If this point is lower or the same height but further to the left than the current lowest point:
460         if (points[point].y < lowest_point.y || (points[point].y == lowest_point.y && points[point].x < lowest_point.x)) {
461             lowest_point = points[point];
462         }
463     }
464
465     points.splice(points.indexOf(lowest_point), 1);
466
467     // The following code sorts the points anticlockwisely.
468     points.sort((a, b) => grahams_order_evaluation(a, b, lowest_point))//(a, b) => angle_between(lowest_point, a) -
            angle_between(lowest_point, b));
469     points.unshift(lowest_point); // unshift adds lowest_point to the start of the array.
470
471     let convex_hull_polygon = new Polygonal_Circular_Doubly_Linked_List(points);
472     convex_hull_polygon.header.graham_scan();
473     return convex_hull_polygon;
474 }
475
476 function grahams_order_evaluation(a, b, lowest_point) { // This deals with cases where the angle between the lowest point
        and the two points a and b are the same.
477     let a_b_angle = angle_between(lowest_point, a) - angle_between(lowest_point, b)
```

```
478     if (a_b_angle == 0) { // If a, b and lowest point are colinear:
479         // The closer point will come first.
480         return new THREE.Vector2().subVectors(a, lowest_point).lengthSq() - new THREE.Vector2().subVectors(b, lowest_point)
                .lengthSq()
481     } else {
482         return a_b_angle;
483     }
484 } // If a is first, return a negative number. If b is first, return a positive number.
485 
486 
487 function angle_between(anchor, point) {
488 
489     if (point.x == anchor.x) {
490         return Math.PI / 2;
491     } else if (point.y == anchor.y) {
492         if (point.x < anchor.x) {
493             return Math.PI;
494         } else {
495             return 0;
496         }
497     }
498 
499     let atan_value = Math.atan2(point.y - anchor.y, point.x - anchor.x);
500     while (atan_value <= 0) {
501         atan_value += Math.PI;
502     }
503     return atan_value;
504 }
```

4 Testing

I have provided a video to accompany this log, which is vital proof of the testing.

4.1 Objectives

1. **Receive user inputs which control the aircraft**
   At 00:29 in the video, I clearly demonstrate that the program can receive all the user inputs specified in the objectives.

2. **Calculate the impact of forces on the aircraft**
   Throught the video, but especially at 00:59 and 01:16, the impact of forces and torques can be clearly seen.

3. **Simulate physics and calculate the forces which are applied to the aircraft**
   At 01:16, thrust, lift, weight and drag are explicitly shown.

   At 3:15, the frontal area is shown (despite the lag in the video) to be very consistent in keeping with what the expected values should be.

4. **Display the 3D model of the aircraft**
   The 3D model is constantly on display during the video



Figure 37: Both the model, the lighting and shadow are in full display here.

5. **Render the scene with respect to a camera.**
   At 00:33, the camera is clearly show to move with the mouse position.



Figure 38: A variety of camera perspectives

6. **Background graphics**
   At 03:24, the airfield is clearly displayed.



Figure 39: This is the airfield being rendered

7. **Airplane collisions** The OBBs were demonstrated beginning from 02:04, and the collision was strenuously tested at 03:05.

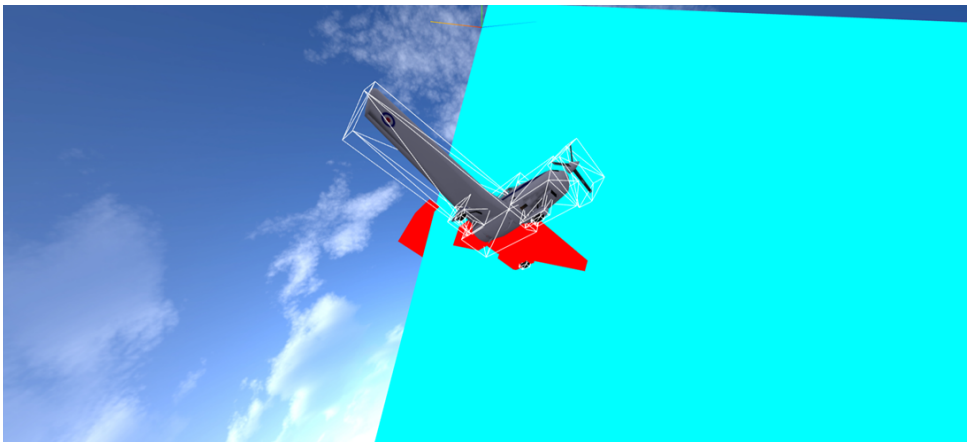


Figure 40: The red OBBs demonstrate the collision being registered

4.2 End User & Client Testing

When I showed my client my project, he was impressed. We went through all of the objectives and he was satisfied with all of them. He was particularly impressed by the graphical fidelity of the program, including the shadow of the aircraft.

I then showed it to some cadets, who stated that they found it helpful because it was a simplified version of the flight simulators that they had previously used, and it was therefore more accessible and easy to learn from.

4.3 Examples

As one would expect, I thoroughly tested my program throughout development. I thought I'd highlight the testing process for the Greiner-Hormann Clipping algorithm:

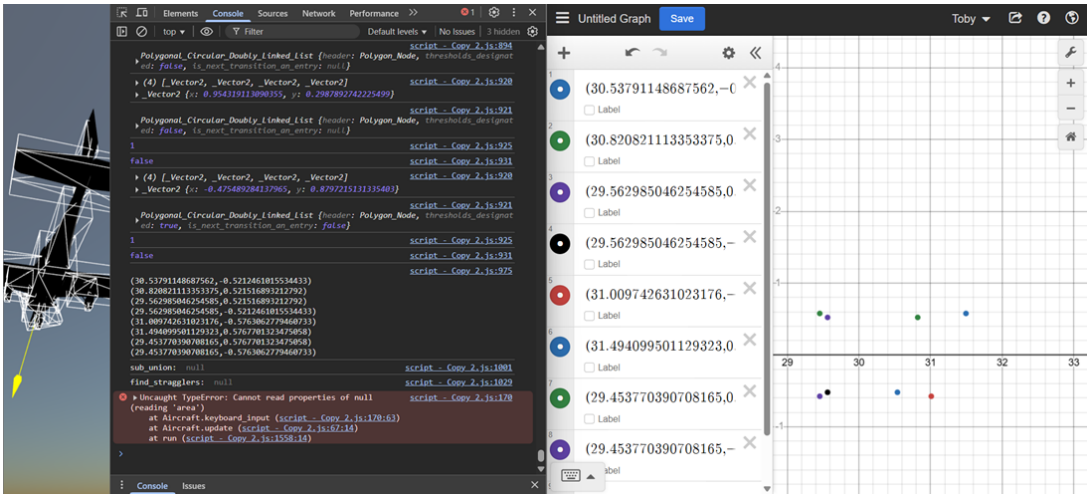


Figure 41: A screenshot of my testing process

Whenever I ran into an error when making this algorithm, I would use the print function that I had made and log all the offending points into the console. From there I would be able to see what had gone wrong and do more targeted, precise debugging.

4.4 Testing Aids

I created several testing aids, such as the print function which can be found in constants.js, or the Point object, which visualised a point in 3D space.

5 Evaluation

5.1 Objectives

1. **Receive user inputs which control the aircraft**
   The control surfaces (such as the ailerons and elevators) moved in response to the user's keyboard inputs, and the camera changed perspective based on the mouse position. Therefore, I would consider this objective met.

2. **Calculate the impact of forces on the aircraft** I gave the aircraft a resultant force and torque attribute to resolve forces and torques, I used Newton's second law to calculate the acceleration and consequently the velocity and position and I calculated the moment of inertia using parallel axis theorem, and from that I calculated angular acceleration and velocity. Overall, I would say that I did a very good job on this objective, and set myself up for success with the later physics objectives.

3. **Simulate physics and calculate the forces which are applied to the aircraft**
   I actually did a lot in terms of physics, despite this objective clearly being the weakest:

   - I used delta time to make sure the physics was being applied at the correct rate.
   - I accurately modelled the thrust using various equations as I have previously specified.
   - I accurately modelled the weight as the force of gravity acting on mass.
   - I found the frontal area of the aircraft to find the terminal velocity, and then incorporated checking to ensure that the velocity never exceeded the terminal velocity.
   - I used a vast set of aerodynamic equations to accurately generate lift, drag and moment coefficients.

   However, I was not able to realistically incorporate the lift, drag or moment. For me, even though I have done so many other physics related objectives, the lift is too crucial to ignore - it is an aircraft after all. Therefore I do not believe that I have met this objective.

But there is a reason that I failed. A symptom of this can be seen in the first few sentences of one of the books I used for research:

> "This book was written for aeronautical, aerospace, andlor mechanical engineers. The book is intended primarily as a textbook for an engineering class that is taught as an upper-division undergraduate or lower-division graduate class."

As someone who doesn't even study Physics, grappling with university level aeronautics was far beyond my comfort zone, and far beyond what could be expected from any A-Level. Therefore, I take some consolation in knowing that, for the objective which I failed the most was, it was hardly to be expected that I could succeed.

So overall, I believe I have fulfilled this objective to an A-Level standard.

4. **Display the 3D model of the aircraft**
   As someone who had had very little experience with 3D modelling, I put this down as an objective, anticipating some difficulty. As it turned out, this was fairly trivial with the tools that Three.js provides, through which I made a class specifically to load 3D models. The more challenging aspects of this were the getting the shadows to work (I had to set cast shadows to true for every submodel of the aircraft and create a gaffer object to ensure the aircraft always casted shadows.) and actually working with the 3D model in Blender, and dividing up the model into different parts.

   Overall, I believe I have met this objective.

5. **Render the scene with respect to a camera.**
   To achieve this objective, I created a camera man object which followed the aircraft round, pointing at its centre of gravity. The most difficult part of this was figuring out how to make the camera orbit the aircraft in a sphere based on the 2D mouse position. I solved this problem by using the parametric definition of a sphere.

   Once again, I have achieved this objective.

6. **Background graphics**
   This was a laborious and regrettably uncreditable task, but I had to describe every triangle of the airfield manually, using $y = mx + c$ and other pieces of maths. However, Flight Sergeant Knowles, whom I mentioned earlier, was quite impressed by this airfield layout, so I would consider this objective a success. However, the terrain is very flat, which could be improved upon. However, I did not consider it a suitable demonstration of my programmatically abilityor use of my time.

7. **Airplane collisions**
   Of all the objectives, this is the one I am most impressed by, perhaps because it spent so long not working. The collision detection is very accurate, and the impulse created in response is realistic. The one thing that I could do better though is to incorporate an OBB tree for the triangles which make up the terrain, as I currently have just a massive triangle which represents the ground. That would also allow me to do collisions with upright objects.

   But ignoring that, I would definitely consider this a success.

Overall, I think that I successfully achieved an introductory demonstration to flight simulators. However, the fact that I didn't fully meet the aerodynamic objectives does limit its success, and I cannot honestly say that I feel that I have successfully created a flight simulator, even if the objective in question is beyond A Level standard. So in conclusion, I do not believe that I have achieved my overall goal of making a flight simulator. Ironically, the crash physics are more realistic than the flight physics, so it has sort of become a crash simulator instead.

Despite my reservations, my client was pleased with it, and thought that it would be a very effective learning tool, if it could be distributed as a website to the cadets.

There are a few things I would like to incorporate if I had more time

- Realistic aerodynamic forces
- Yaw and the Rudder
- Different aircraft and better terrain
- Some UI, like a Airspeed indicator

- A Database for lookup tables
- Touch screen to allow it to be used on phones.

I believe that the outcome could be improved with some additions of some if these thing, like realistic lift generation, but there is a more significant way in which I would improve my project: I need to work on my approach.

I had a consistent approach for working on my project: do very limited research on a subject which I wanted to include, spot something that interested me about it, do a lot of research and spend a week or two down a rabbit hole, create hundreds of lines of code for this problem, then realise that this wasn't really a very important part of the project. It would seem that I struggled with scope and having a balanced approach to this project. For example, I implemented gravity. However, its accelerative nature made it so the aircraft fell into the void within seconds. As it would later turn out, this could be fixed by implementing delta time. However, my immediate instinct was to incorporate terminal velocity. So I looked it up (the first result was from WikiHow), and the equation it gave required the projected area of the aircraft onto the plane of its velocity. Fast-forward two weeks, and I have achieved this. However, my hope to use this in the lift, drag and moment equations would be dashed when I realised that the area specified in those equations was in fact the planform wing area, not the projected frontal area. These are the kind of issues my lack of research caused. Fortunately, I remembered that my original intention was to use it for terminal velocity. However, that is a very minor part of the physics system, and two weeks is quite a long time, even if the algorithms and data structures used are impressive and recursive. If I hadn't remembered to use it for terminal velocity, I might have been forced to delete over 600 lines of code. To rectify this in future projects, I will take a top-down view of the project, and list things in order of priority. That would mean making a functional physics system implementing moments and control surface deflections before launching into esoteric odysseys as I am prone to doing.

Ultimately, it was foolish to hope that I could rival the works of those so much more experienced and talented than I, but alas, I tried.

6 Appendices

6.I Meeting Log

Overview

This is a log of all of my meetings with my client, Flt. Lt. Williams. For each meeting, I created agenda items and have included the accompanying minutes.

6.I.A Meeting 1 - 18/03/2025

Item 1 - The purpose of making our own flight simulator

1. It would allow people who are part of the RAF cadets but don't go to this school to have the experience of using a flight simulator.
2. It could be used as a teaching or assessing tool, especially for cadets without access to the flight simulators.
3. As programming and cyber skills are emerging and important areas in the RAF, it could serve as a demonstration of the possibilities of coding.
4. Using a flight simulator is one of the most fun aspects of being an RAF cadet, so allowing more cadets to have access to one increase their enjoyment of the section and make them more likely to stay with the RAF cadets in the future.

Item 2 - How should we distribute it?

1. Distributing the program as a `.exe` file would allow us to easily control who has access to it, and would make the program more exclusive.
2. However, it would be difficult to distribute to a `.exe` file, as many email services understandably prevent attaching `.exe` files to emails.
3. Ultimately, a website would be the best and most flexible option, as cadets could easily access it from home.

Item 3 - What should the User Interface and Experience be like?

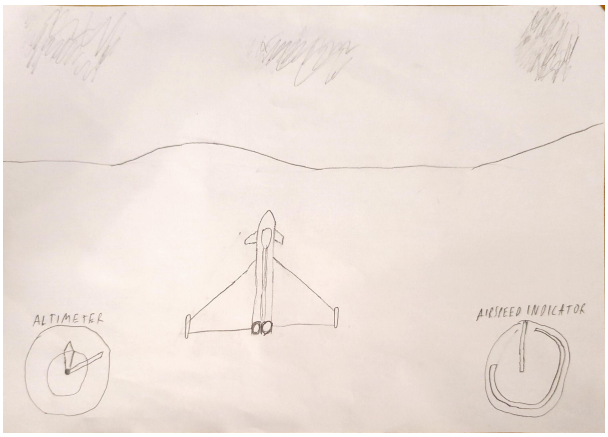


Figure 42: The mock-up I made to demonstrate what the flight simulator and UI could look like.

1. Useful meters and indicators - like an Airspeed Indicator or an Altimeter should be on screen, to allow cadets to understand more of the statistics behind flying and for cadets to react to when flying.
2. A variable camera position - from a cockpit view to an exterior view - was initially considered. This was dismissed by the client after seeing Figure 42, and preferring it to the cockpit view. This decisions reduces the complexity in order to prioritise other more important aspects such as accurate aircraft controls, whilst providing the user with an clear view of the aircraft.

3. A 2D, gamified version of a flight simulator with a side-perspective would be more unique and perhaps more engaging and enjoyable for the cadets. This idea was suggested by the client, however, it was rejected as it would not give cadets a realistic experience of controlling an aircraft.
4. I asked my client about the control scheme: whether the aircraft's pitch and roll should be controlled by the arrow keys, or by the mouse's position. He decided that using the keyboard would provide a more realistic experience over a mouse, allowing the mouse to control the camera position instead.

Item 4 - The overall scope of the project.

4 key objectives were decided upon and considered a priority.

1. Allow the aircraft to be accurately controlled and manoeuvered by the user.
2. Subject the aircraft to realistic physics to simulate actual flight.
3. Render an airfield and surrounding terrain to increase realism.
4. Teach the user how to control the aircraft through a tutorial.

Incorporating realistic aircraft noise was also suggested, but was considered a lower priority.

6.I.B Meeting 2 - 21/05/2025

Item 1 - Control Scheme

1. I asked Flt. Lt. Williams if he thought the keyboard controls (WASD for ailerons and elevator control) were suitable, or if he would prefer a mouse based system where the aircraft moves towards the location of the mouse pointer.
2. He said that he thought keyboard controls would work better, as they are more representative of the actual control scheme of an aircraft.

Item 2 - Camera

1. I asked Flt. Lt. Williams if he thought the current camera configuration (the camera orbited the aircraft based on the location of the mouse pointer) was suitable, or if he would prefer a more fixed system where the camera can still orbit the aircraft, but its position adjusts to follow the aircraft round as it rotates.
2. He said that he preferred the former option, the free orbit system, because it would allow the user to be able to view the aircraft more freely.
3. I also asked him if he was happy with the external camera view, rather than an internal one which displays the cockpit.
4. He said that, whilst the potential for an internal view would be nice and useful for cadets to see how the internals of the aircraft appear, how the aircraft moves and reacts to inputs from the user is more important and should be prioritised.

Item 3 - Terrain

1. I asked Flt. Williams if the terrain should be procedurally generated or predetermined.
2. He said that it should be predetermined, allowing for more thoughtful design and a more consistent experience. This would also allow cadets to familiarise themselves.

Item 4 - Practical Use and Implementation

1. The aim of this project, rather than being an especially realistic flight simulator, is to give cadets an impression of how aircraft are controlled and how user input can affect control surfaces.
2. The flight simulator should therefore contain a variety of tutorials to make it accessible to inexperienced cadets.
3. It could also be used to explain the theory of flight.
4. A benefit of this is that we could get everyone to it at once in a computer room, so it would therefore be less restrictive than the three flight simulators in the RAF Cadet Hut.

Item 5 - Use on phones

1. Ideally, this program should be accessible to cadets on phones.
2. As phones are ubiquitous in the modern age, this would extend the usefulness of the program.
3. This means that I have to make the program responsive and able to receive touchscreen input.

6.II Bibliography

References

[1] Helijah. A 3D model of a Grob Tutor. URL: https://sketchfab.com/3d-models/grob-g-115-64bc0e98d56a4f88a2833dbefa2ca975.

[2] Xavier Tassin. *GeoFS - Free Online Flight Simulator*. URL: https://www.geo-fs.com/.

[3] David Lion et al. "Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?" In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 835–852. ISBN: 978-1-939133-29-40. URL: https://www.usenix.org/conference/atc22/presentation/lion.

[4] Microsoft. *Microsoft Flight Simulator 2024*. URL: https://store.steampowered.com/app/2537590/Microsoft_Flight_Simulator_2024/.

[5] Eagle Dynamics. *DCS World 2.9*. URL: https://www.digitalcombatsimulator.com/en/downloads/world/stable/.

[6] RAF. An image of a Grob Tutor. URL: https://www.raf.mod.uk/sites/raf-beta/cache/file/B6186D49-7039-4394-AA76A04A06DA6529_1200x800.jpg.

[7] NASA. *Principles of Flight: Axes / Control Surfaces*. URL: https://www.nasa.gov/wp-content/uploads/2023/06/axes-control-surfaces-5-8.pdf.

[8] *TYPE-CERTIFICATE DATA SHEET*. URL: https://www.caa.co.uk/Documents/Download/3934/ca692457-336e-4190-840e-cd26cf498935/2596.

[9] Federal Aviation Administration. *Aerodynamics of Flight*. URL: https://www.faa.gov/sites/faa.gov/files/07_phak_ch5_0.pdf.

[10] Munson K Jackson P Peacock LT. *Jane's All the World's Aircraft, 2004-2005. 95th ed.* Jane's Information Group, 2004.

[11] Manuel Soler Arnedo. *Propeller propulsion equations*. URL: https://eng.libretexts.org/Bookshelves/Aerospace_Engineering/Fundamentals_of_Aerospace_Engineering_(Arnedo)/06%3A_Aircraft_propulsion/6.01%3A_The_propeller/6.1.01%3A_Propeller_propulsion_equations.

[12] Snorri Gudmundsson. "Chapter 14 - The Anatomy of the Propeller". In: *General Aviation Aircraft Design*. Ed. by Snorri Gudmundsson. Boston: Butterworth-Heinemann, 2014, pp. 581–659. ISBN: 978-0-12-397308-5. DOI: https://doi.org/10.1016/B978-0-12-397308-5.00014-3. URL: https://www.sciencedirect.com/science/article/pii/B9780123973085000143.

[13] NASA. *Lift Equation — Glenn Research Centre — NASA*. URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/lift-equation/.

[14] *Sea level conditions*. URL: https://www.aerodynamics4students.com/properties-of-the-atmosphere/sea-level-conditions.php.

[15] NASA. *Drag Equation — Glenn Research Centre — NASA*. URL: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/drag-equation-2/.

[16] Jump Trajectory. *Realistic Aircraft Physics for Games*. URL: https://www.youtube.com/watch?v=p3jDJ9FtTyM.

[17] James Albright. An diagram of the angle of attack.

[18] Martin Hepperle. URL: https://www.mh-aerotools.de/airfoils/javafoil.htm.

[19] Waqas Khan and Meyer Nahon. "Real-Time Modeling of Agile Fixed-Wing UAV Aerodynamics". In: June 2015. DOI: 10.1109/ICUAS.2015.7152411.

[20] B. W. McCormick. *Aerodynamics, Aeronautics, and Flight Mechanics*. 2nd. New York: John Wiley & Sons, 1995.

[21] David Eberly. "Dynamic Collision Detection using Oriented Bounding Boxes". In: *Geometric Tools* (1999).

[22] 3Blue1Brown. *Eigenvectors and eigenvalues — Chapter 14, Essence of linear algebra*. URL: `https://youtu.be/PFDu9oVAE-g`.

[23] OnlineMathLearning.com. *Complex Plane: Rectangular and Polar Form*. URL: `https://www.onlinemathlearning.com/complex-plane-hsn-cn4.html`.

[24] Newcastle University. *Physics Tutorial 5: Collision Response*.

[25] NCCU Information Engineering Wiki. *acm/course/Convex Hull*. URL: `https://wiki.csie.ncku.edu.tw/acm/course/Convex%20Hull`.

[26] Günther Greiner and Kai Hormann. "Efficient clipping of arbitrary polygons". In: *ACM Transactions on Graphics* 17(2) (April 1998), pp. 71–83.