Air Traffic Control Simulator

Nathaniel Taulbut

23rd February 2023

Contents

List of Figures

Listings

1 Introduction

Figure 1: A radar display at Swanwick ATC centre (NATS, CC BY-NC-ND 2.0)

My project is an air traffic control simulator. Air Traffic Control (ATC) is a service provided by air traffic controllers, who issue instructions and provide information to aircraft on the ground and in the air. Air traffic controllers monitor the position of aircraft using radar, as shown in Figure 1, and communicate with pilots using radio[1].

2 Analysis

There are three main types of air traffic controllers: Aerodrome, Area, and Approach.

Aerodrome Controllers issue clearances to take off and land and route aircraft around the airfield; Area Controllers are responsible for aircraft in the climb, descent and en-route phases of flight; and Approach Controllers manage aircraft approaching an airport, putting them into the most efficient sequence to land[2].

An air traffic controller is responsible for a particular section of airspace. Aircraft will arrive into their area of responsibility from certain points, and they will have to guide the traffic to the next controller's area of responsibility. For example, a plane enters an approach control area descending from cruise, and the controller must guide them to an approach for a certain runway, where they are transferred to the tower controller who is responsible for the runways and the area in close proximity to the airport.

There are existing solutions for real-world training of air traffic controllers, and for public use. Because the solutions for real-world controllers require certification, my solution will focus on public use. My solution will aim to improve on the areas that are lacking in the existing publicly available solutions.

2.1 Stakeholders

2.1.1 Air Traffic Control organisations

The UK's National Air Traffic Services (NATS) organisation has a basic air traffic control game on their website for people to test their skills and determine if becoming an air traffic controller is right for them. My solution could better solve that problem by providing a more realistic simulation, creating a more accurate test. Other ATC organisations could promote the solution and make use of it to get people interested in air traffic control, making them more likely to take it up as a career, which is necessary as some ATC organisations struggle to recruit enough controllers[3].

2.1.2 Hobbyists

Providing air traffic control is challenging and high-pressure. It involves reacting to novel situations; thinking and planning ahead; and executing to move aeroplanes as safely and quickly as possible[4]. For this reason, many people find it enjoyable to play the role of ATC in a simulator – the Virtual Air Traffic Simulation Network (VATSIM)[1] has over 100,000 active members as of 2023. These users likely have an interest in air traffic control generally or want to become a controller in real life. They will make use of the solution for entertainment as well as personal training in skills such as multi-tasking and problem solving. Their needs include realism and ease of use, which my solution will be able to provide. Because of their interests, they are likely to own a desktop or laptop computer.

Representing this group is Freddy. He is a sixth form student who is interested in and knowledgeable about Air Traffic Control, wanting to become an RAF Air Traffic Controller in the future. He has used multiple air traffic control simulators including Tower3D Pro and Endless ATC. He can therefore provide very useful feedback on my solution.

2.2 Existing solutions

2.2.1 ATC-SIM

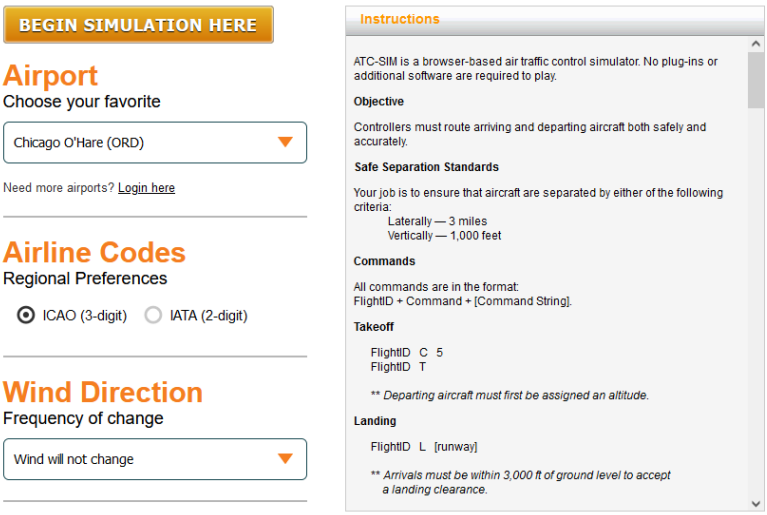

Figure 2: ATC-SIM Main Menu

ATC-SIM[2] is a browser-based air traffic control simulator which focuses on approach and tower control. On the menu screen shown in Figure 2, the user can select which airport they would like to control at and other preferences such as how much the wind will change. It also shows instructions for the main part of the game. The user can then press 'begin simulation here' to start, which takes them to the gameplay screen. This serves its purpose well.

[1] https://vatsim.net
[2] https://atc-sim.com/

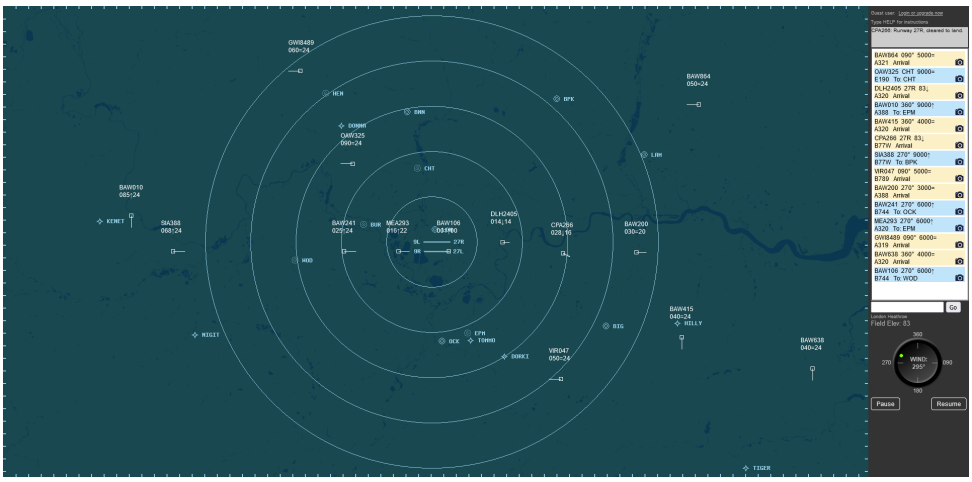

Figure 3: ATC-SIM Gameplay Screen

The user can issue instructions to aeroplanes by entering abbreviated text commands into a text box, e.g. ”BAW123 C 6” is an instruction for that aeroplane (”BAW123”) to climb or descend to 6,000ft. As seen in Figure 3, a list of aeroplanes under the user's control is on the right-hand side of the screen. On the central screen, text next to each aeroplane displays their current altitude, heading, and speed. A display on the right-hand side shows the current wind speed and direction. A background image shows the terrain and areas of water.

The view cannot be panned or zoomed and if the size of the browser window changes, the background moves but the waypoints don't, resulting in a visual mismatch (see Figure 13). Only a limited number of waypoints are available for the user to direct the aeroplane to. There is no visual indication of the approach path, which makes it impossible for the user to accurately guide aeroplanes in. Overall this is a reasonable solution for some but will disappoint more knowledgeable users.

2.2.2 Endless ATC

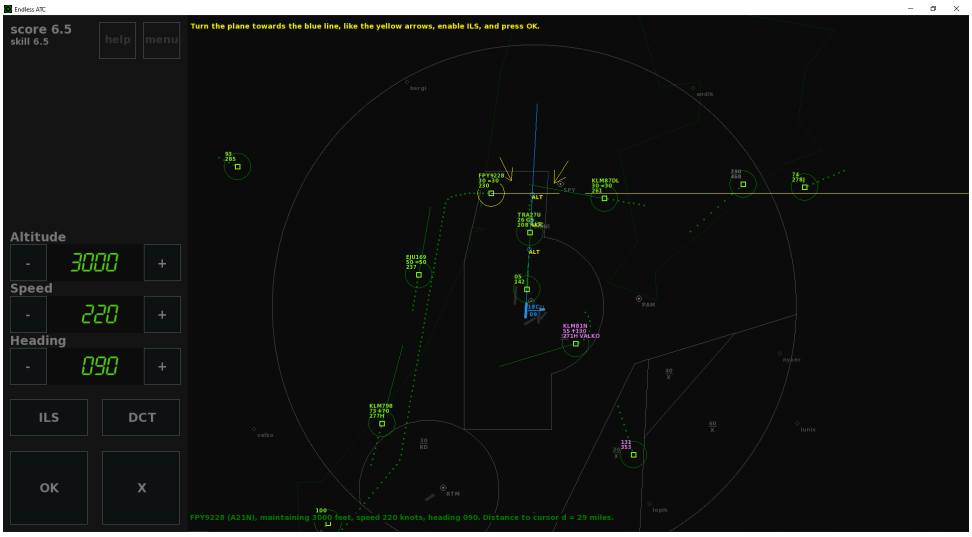

Figure 4: Endless ATC

Endless ATC[3] is a desktop and mobile air traffic control simulator focused on approach control. The user can issue headings to aeroplanes by clicking or tapping on them and then moving their cursor in the direction of the desired heading. The view can be panned around with the mouse and zoomed in with the scroll wheel. Buttons on the side allow the user to change altitude, speed, and heading, or they can use various keys such as the scroll wheel to quickly change these values.

[3] https://startgrid.itch.io/endlessatc

Once values have been changed, a button must be pressed to confirm or cancel the instructions. A problem is that an instruction is also confirmed when the user clicks away or onto another aeroplane, potentially leading to confirming an instruction the user didn't mean to. Text next to each aeroplane displays their current altitude, heading, and speed. A limited number of waypoints are available which aeroplane can be directed to by clicking and dragging to them. The coastline is shown with overly simple lines. A limited number of customisation options are available, such as changing the aeroplane icons. Another issue is that the user can only give altitude instructions in increments of thousands of feet, whereas it is very common in reality for controllers to give instructions involving hundreds of feet.

2.3 Features

2.3.1 Simulated aeroplanes

The aim of my solution is to simulate the role of an air traffic controller, and the user will therefore need some traffic to control. Whilst some solutions require other people to 'fly' the aeroplanes, mine should allow someone to use the simulation without any additional people, since if it did require them it would reduce the flexibility of the use of the simulator to when they could be organised. Therefore the air traffic will have to be simulated and computer controlled. This will involve repeatedly calculating the aeroplane's movement many times a second based on their speed and direction, and other factors such as wind. This can be achieved with an algorithm and mathematical calculations which are suited to being run on a computer, allowing many aeroplanes to be simulated quickly and concurrently.

These aeroplanes will be abstracted from reality, only simulating the variables that are necessary to appear realistic on a radar screen. For example, lift does not have to be simulated — instead it can just be stated that the aeroplane is at a particular altitude. To make them behave realistically however, it is necessary to simulate abstractions of e.g. the pitch of the aeroplane, because this affects how long it takes for the aeroplane to change altitude. Because the pitch required to maintain a certain flight path angle changes dependent on factors that would be complicated to simulate, such as lift and speed, I can instead model vertical motion only by the flight path angle, which allows for the same realistic behaviour without having to calculate the actual pitch.

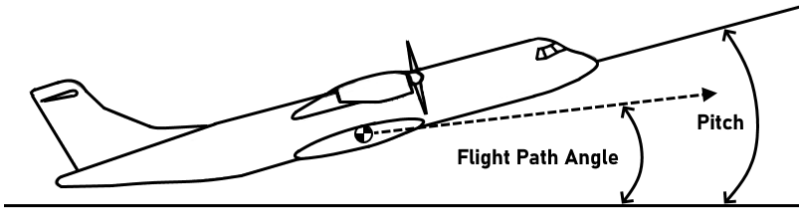

Figure 5: Pitch and FPA

A simulation of the takeoff procedures of the aeroplane, such as its acceleration on the ground and initial rotation, will not need to be included because the solution is focussed on approach control. Aeroplanes landing and flaring can also be largely ignored because below a certain altitude they will not be visible on the radar anyway.

The following success criteria must be met to ensure sufficient realism of the aeroplane simulation as required by the stakeholders:

- To simulate the manoeuvering characteristics of a typical airliner, it should take a certain amount of time for an aeroplane to change direction and speed
- The aeroplane should take a realistic amount of time to change altitude and heading
- Timescales should always be accurate, independent of the performance of the simulation
- Fifteen or more aeroplane should be able to be simulated simultaneously while maintaining performance, as this is how many a controller is typically required to control in real life

2.3.2 Giving instructions to aeroplanes

In reality, air traffic controllers instruct aeroplane pilots verbally over radio. The aeroplanes in my solution will be simulated, so there will be no pilot to hear and parse an audio instruction, e.g. ”Air France 354, descend 5,000 feet, QNH 1018”. Therefore an alternative method of giving these aeroplanes instructions will be needed. The essential instructions necessary to guide an aeroplane to a landing are for it to:

- Fly on a heading
- Fly to a waypoint
- Change altitude
- Travel at a certain airspeed
- Perform an instrument approach

The solution will create an abstraction of the verbal instructions which hides some detail, for example instead of using the text ’[callsign], climb/descend altitude 5,000 feet’, the user could type a number into the corresponding box and the plane will decide whether it needs to climb or descend if its current altitude is above or below the entered value.

The solution should allow the user to give instructions as quickly or quicker than they would be able to verbally in real life. To make the experience more fluid, it should also require as few button presses or mouse clicks as possible.

2.3.3 Wind drift

Controllers often direct aeroplanes by telling them to fly on a certain heading. The heading of an aeroplane describes the direction it is pointing in which, largely because of wind, is not necessarily the same as the direction it is travelling in. Therefore controllers have to account for this when giving headings so that the aeroplane’s actual track is in the intended direction. This is a big part of the controller’s job, so it will be necessary to simulate this. The input to this system will be a wind direction and a wind speed, live data for which could be obtained from an internet API.

2.3.4 Displaying aeroplanes

A representation of the aeroplan will need to be shown. It will need to show their assigned heading or waypoint, altitude, and speed; their current heading or targeted waypoint, altitude, and speed; and their callsign. As with real life radar displays, it should show a trail of dots or other symbols behind the aeroplan to illustrate its path and a line in front to show its current direction of travel. It should be visually similar to real life for additional realism and immersion. This can only be accomplished with a computer because of its unique ability to render custom, moving and changing graphics to display on an output device such as a monitor.

2.3.5 Waypoints

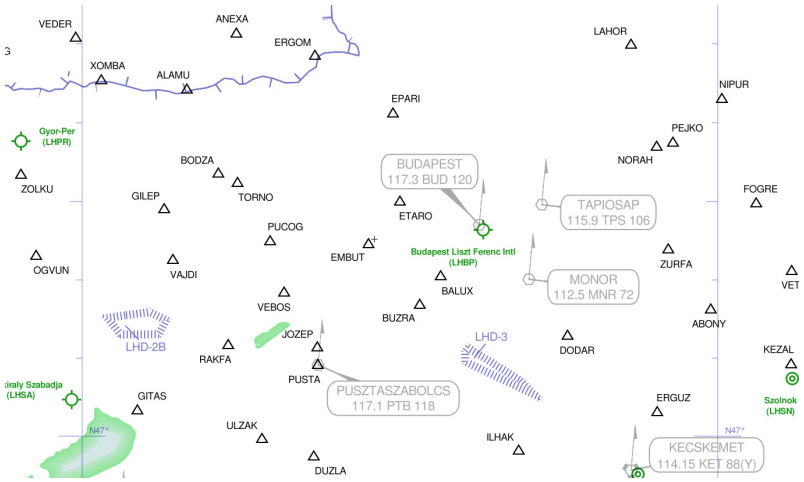

Figure 6: Map showing waypoints around Budapest International Airport

Waypoints are points, defined by latitude and longitude and given five-letter names, which are used for air navigation. They are created specifically for that purpose and usually have no connection to features of the real world[5]. A commercial airliner's route from one airport to another — its flightplan — is largely defined as a series of waypoints which it will fly between. Air traffic controllers often take advantage of waypoints to direct traffic, instructing them to 'fly direct' to a named waypoint, for example "BAW123, route direct WILLO". Waypoints are indispensable to air traffic controllers, therefore it is necessary to include them in my solution. This will require data to be sourced with the names and GPS coordinates of the waypoints. The waypoints should be shown on the screen in the correct position with a symbol corresponding to their type, and their name. As many as 30 or more waypoints should be able to be displayed at once, since this is how many are present around some airports.

2.3.6 Pan and Zoom

The user should be able to pan and zoom their view of the simulated radar screen. This way they can more clearly see important areas where the majority of traffic is concentrated, rather than being limited to a fixed view. This is especially useful where the approach control airspace area is very large and it otherwise would not be possible to simulate because there would not be enough room on the screen. Being able to pan and zoom is also a feature of most modern real world radar screens. To make interacting with the solution as enjoyable as possible and as professional as the real world equivalents, panning and zooming should be smooth; at a fixed speed and in fixed steps; and the camera should zoom towards the centre of the screen. This will also create a familiar experience which is intuitive.

2.4 Limitations

A few limitations must be stated to prevent the solution from becoming overly complex and to limit the scope. They are as follows:

- The simulator will only load information such as waypoints and terrain in a small area around the airport — it will not allow the user to pan around the whole world at once. Doing so would require streaming the data in as the user moved around, which would be unnecessarily complex to implement because the user is only controlling aeroplanes in one geographical area at a time anyway.
- The simulator will only simulate traffic under instrument flight rules (IFR)[4] because their behaviour is predictable. Aeroplanes under visual flight rules (VFR) are flown by the pilot with reference to visual landmarks which are not simulated. This is a fairly major limitation because at some airports VFR traffic is very common, however the existing solutions researched also have this limitation.

[4] https://en.wikipedia.org/wiki/Instrument_flight_rules

- Different types of aircraft, e.g. helicopters, will not be simulated because the way they behave is very different to fixed-wing aircraft and in most airports it is not common to see them. The maneuvering characteristics used will be an approximation of an average airliner, rather than having different data for different types.
- Airspace boundaries will not be simulated because it is too difficult to obtain the data defining them; and even if it could be sourced, it would also be difficult to calculate whether an aircraft was within a certain area of airspace because their areas are defined by a series of points. It is also not very necessary because at most airports there is a large open area of airspace, so considering it does not enter into the workload of the controller much; so excluding it would not decrease realism much.

2.5 Requirements

In order to make the solution accessible to as many people as possible, the program should not require any additional software to be downloaded to the user's device for it to function. However there are two problems that mean my solution will be limited to a desktop or laptop computer. Firstly, the complex inputs necessary for directing air traffic, which rules out devices without the facilities for a keyboard and mouse such as consoles. Secondly, the large area of information which has to be displayed, which rules out devices with small screens such as smartphones. However this will not present an issue, because the majority of stakeholders in the solution will have a computer.

To the same end, it should function well on a computer with very minimal specifications, for instance a 2GHz dual-core processor with integrated graphics, 4GB of RAM, and a 120GB hard drive. The program should function on Windows, Linux, and macOS systems.

3 Design

Following the design of ATC-Sim, the simulator will have two main screens: a main menu where the user can select which airport they want to control at, and a separate gameplay screen they will be taken to where that will take place. Separating these problems means that the code for the menu does not need to know how the simulation works, it just gives it the necessary data, and vice versa the simulation does not know about the menu. This will make development easier because changes to one system will largely not affect the way the other works.

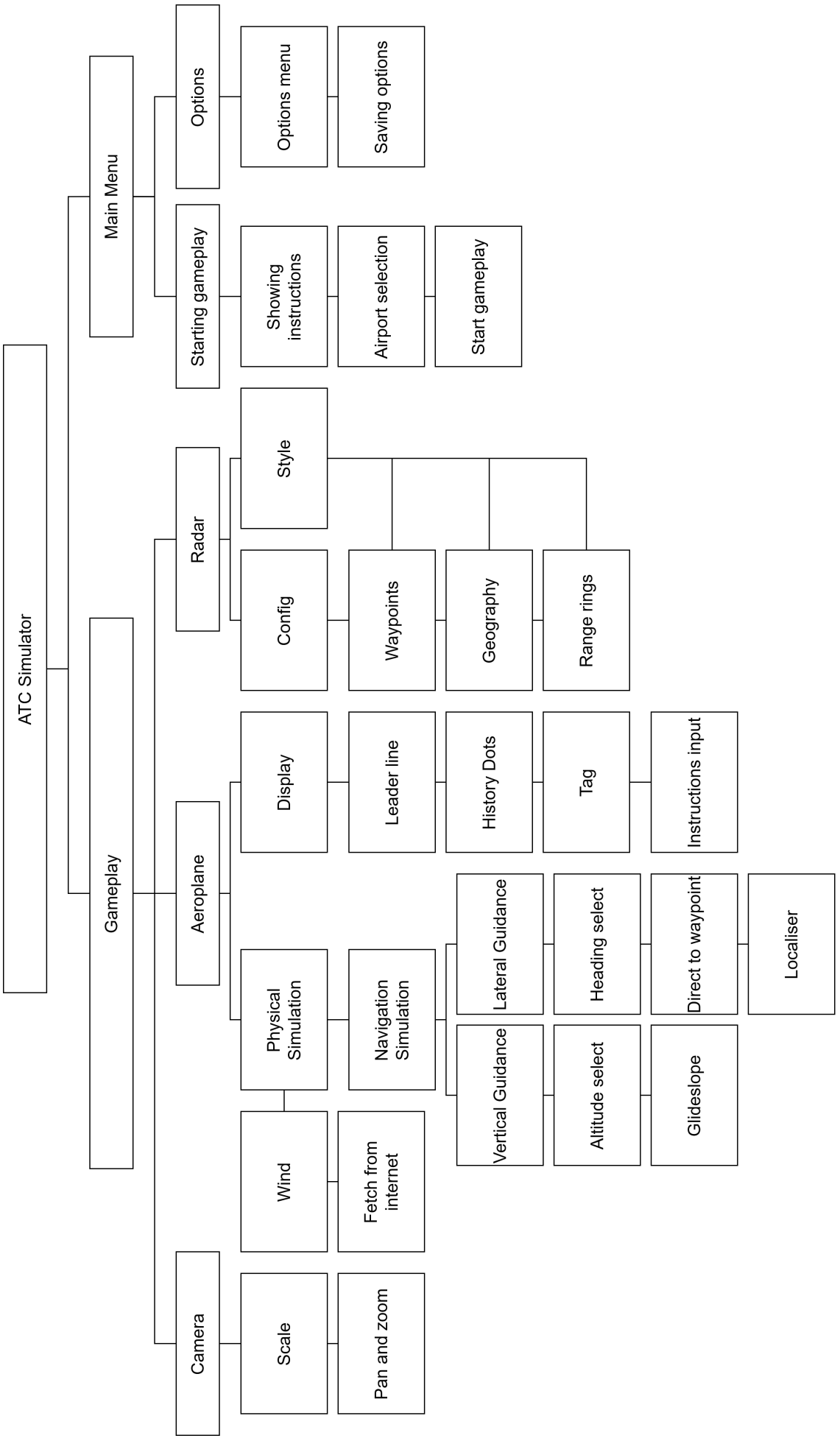

Figure 7: Solution diagram

3.1 Data

Many problems and of the solution shown in figure 7 require data to be stored. This includes waypoints, geography, airports, and more. The Godot game engine provides a feature called *resources*, which are data containers that allow you to define the names and data types of fields that the resource should store; then the engine manages writing this to the disk in an appropriate place as a text file. This is very convenient because it abstracts away the details of handling files on different operating systems and allows an object-oriented style approach to data storage where fields can contain references to other resource types, for example a RadarConfig resource will contain an array of references to Waypoint resources. It also provides data types such as vectors. Below is the initial design for the data required to be stored about each feature:

Waypoint	• Vector2 LatitudeLongitude • string Name
Airport	TODO

3.2 Displaying aeroplanes

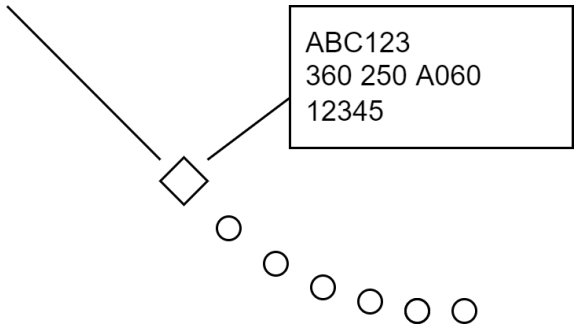

Figure 8: Design of the aeroplane display

Figure 8 shows the four main features of the aeroplane interface.

- The blip: a dot or other symbol showing the last known position of the aeroplane, which the other features are centred around.
- The leader line: a line pointing in the last known direction of travel.
- History dots: a series of dots or other symbols showing the previously received positions of the aeroplane, illustrating its motion.
- The tag: a box which can be dragged around relative to the blip showing information about the aeroplane; it will also be used to give instructions to the aeroplane.

3.3 Giving instructions to aeroplanes

As described in section 2.2, there are various approaches to this. One is to use text input. The problem with this is that, especially for users who are slow at typing, it may be slow to use; also the user could make a typo which would require them to type the instruction out again. Another approach is to use mouse movements. This would be faster and more intuitive to use but would compromise on realism and might not enable me to offer the complex set of instructions necessary. Therefore to strike the best balance between usability and realism, I will use a system in which the user enters values into input boxes on the tags next to the aeroplane, and more complicated options will be presented through drop-down menus on the tag.

3.4 Aeroplane physical simulation

TODO

3.5 Aeroplane navigation simulation

In order to maintain physical consistency (e.g. if an aeroplane is already in a turn to the right, it will take longer to start turning to the left than if it was not turning), the navigation system will be isolated from the physical simulation and will interact with it by commanding a yaw rate in degrees per second and a flight path angle in degrees. At all times, a lateral and vertical guidance mode will be engaged. Optionally, other lateral and vertical guidance modes can be *armed*, meaning at each simulation step they will be asked if they want to activate — at which point they will take over the active mode and be removed from the armed modes list. This mirrors how autopilots work in reality. Every simulation step, the update method of the active modes will be called to receive their new command. The modes will each be classes which will inherit from a parent class such as LateralMode, overriding e.g. the RollCommand method to implement their own functionality.

3.5.1 Vertical guidance

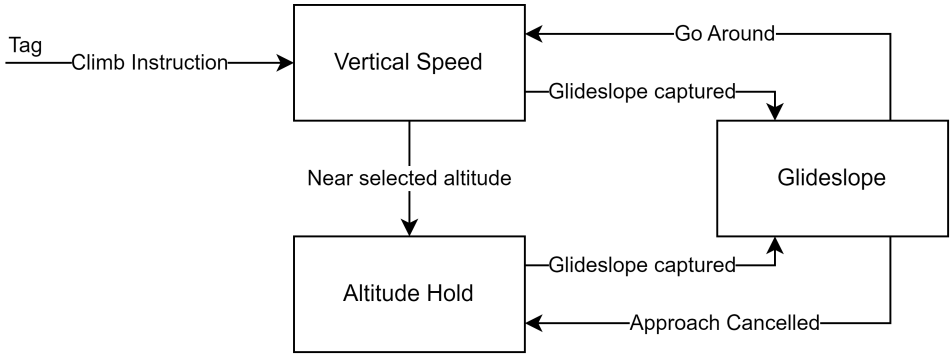

Figure 9: State transitions between vertical modes

Mirroring reality, when an instruction to climb or descend is given an altitude change mode such as vertical speed will be engaged and the altitude hold mode armed. The vertical speed mode will control the commanded flight path angle to achieve the desired vertical speed, and then as the aeroplane nears the instructed altitude the altitude hold mode will activate itself and level the aeroplane off without over or undershooting.

When the aeroplane is cleared for an ILS approach, the glideslope capture mode will be armed, and then as the aeroplane approaches an appropriate distance from the glideslope, the mode will activate itself and command a pitch down to match the glideslope angle. This will require calculating the current height of the glideslope beam at the aeroplane's position based on the position and elevation of the transmitter; and the angle of the glideslope.

3.5.2 Lateral guidance

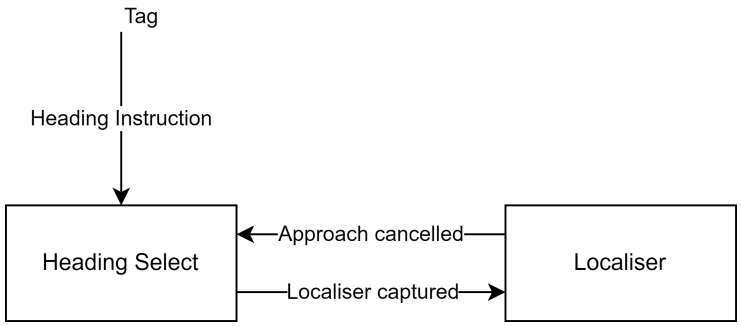

Figure 10: State transitions between lateral modes

3.6 Waypoints

Figure 11: Waypoint design

Waypoints should appear on the screen as in the diagram, with a symbol depending on their type and text with their name next to them.

3.7 Usability features

TODO

4 Development

I have chosen to use the Godot game engine[5] to build my simulator. It works using a system of nodes, which I can use to hold each module of my solution.

4.1 Mapping the real world onto the screen

Several features of my solution need to work with real-world points, such as waypoints, defined by latitudes and longitudes. To make these easier to work with, I decided to convert their positions from latitudes and longitudes into Cartesian coordinates, with the screen centre as (0, 0). To do this, I would need to assign the centre of the screen a latitude and longitude, and then calculate the point's lateral and vertical distances from it. If I wanted to display the point on the screen, I would then need to scale the real-world distances to pixels.

To enable this I created a class, 'Geo', to provide re-usable geographical functions. For example whilst the function shown in listing 1 was initially created to enable displaying waypoints on the screen, it can also be used to for example display geographical features.

4.1.1 Calculating distance along the Earth

I chose to use the spherical law of cosines formula to calculate distance, since with modern floating point numbers, it does not have significant rounding errors for the distances seen in my solution. It states that where ϕ_1, λ_1 and ϕ_2, λ_2 are the latitudes and longitudes of two points, and $\Delta\phi, \Delta\lambda$ are their absolute differences, the central angle between them is given by:

$$\Delta\sigma = \arccos\left(\sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos(\Delta\lambda)\right)$$

Because the inputs are given in radians, the result is in radians. The simple sector length formula $L = r\theta$ can then be used to find the distance, where r is the radius of the Earth[6].

```
1 // Average radius of the Earth in nautical miles
2 private const float EarthRadiusNm = 3438.175f;
3
4 public static float GetDistanceNm(float latitude, float longitude, float
      otherLatitude, float otherLongitude)
5 {
6     float centralAngle = Mathf.Acos(Mathf.Sin(latitude) * Mathf.Sin(
      otherLatitude) + Mathf.Cos(latitude) * Mathf.Cos(otherLatitude) * Mathf
      .Cos(Mathf.Abs(longitude - otherLongitude)));
7     return EarthRadiusNm * Mathf.DegToRad(centralAngle);
8 }
```

Listing 1: Calculating great-circle distance using the spherical law of cosines

After testing, I found that this broke somehow. So I used the haversine formula instead:

$$\mathrm{hav}(\theta) = \mathrm{hav}(\Delta\phi) + \cos(\varphi_1)\cos(\varphi_2)\,\mathrm{hav}(\Delta\lambda)$$

$$d = r\,\mathrm{archav}(h)$$

```
1 // Convert to radians
2 float lat = Mathf.DegToRad(latitude);
3 float lon = Mathf.DegToRad(longitude);
4 float otherLat = Mathf.DegToRad(otherLatitude);
5 float otherLon = Mathf.DegToRad(otherLongitude);
6 // Use the haversine formula
7 float haversineTheta = Haversine(otherLat - lat) + Mathf.Cos(lat) * Mathf.
      Cos(otherLat) * Haversine(otherLon - lon);
8 return EarthRadiusNm * Archaversine(haversineTheta);
```

Listing 2: Calculating great-circle distance using the haversine formula

[5] https://godotengine.org/

```
public static float Haversine(float x)
{
  return Mathf.Pow(Mathf.Sin(x / 2f), 2);
}

public static float Archaversine(float x)
{
  return 2f * Mathf.Asin(Mathf.Sqrt(x));
}
```

Listing 3: The haversine functions

4.1.2 Calculating position as a cartesian coordinate

By using an imaginary point with the latitude of the screen centre and the longitude of the point, I can individually calculate the horizontal and vertical distances. Because the distance will always be positive, I have to calculate the quadrant the point lies in relative to the screen centre and flip the sign accordingly for it to be a coordinate.

```
public static Vector2 RelativePositionNm(Vector2 position, Vector2 relativeTo)
{
    float horizontalComponent = GetDistanceNm(relativeTo.x, relativeTo.y, relativeTo.x, position.y);
    horizontalComponent = position.y < relativeTo.y ? -horizontalComponent : horizontalComponent;
    float verticalComponent = GetDistanceNm(relativeTo.x, position.y, position.x, position.y);
    verticalComponent = position.x < relativeTo.x ? -verticalComponent : verticalComponent;
    return new Vector2(horizontalComponent, verticalComponent);
}
```

Listing 4: Calculating the position of a point relative to another

4.1.3 Scaling

Because the size of the game window will change based on the resolution of the user's monitor and if they resize it, the scale factor has to continually change if a fixed real-world distance is to be displayed, which is necessary for my solution. It may also be useful to be able to fix the distance by either height or width. Calculating the scale factor given a desired real-world distance to display is a simple matter of dividing the screen size by that distance.

```
public static new float Scale(Rect2 viewportRect)
{
    _viewportRect = viewportRect;
    float zoomValue = Mathf.Pow(1 + ZoomSpeed, Zoom);
    return RadarConfig.FixedBy switch
    {
        RadarConfig.DisplayFixedBy.Width => (viewportRect.Size.x / RadarConfig.WidthNm) * zoomValue,
        RadarConfig.DisplayFixedBy.Height => (viewportRect.Size.y / RadarConfig.HeightNm) * zoomValue,
        RadarConfig.DisplayFixedBy.Compromise => (viewportRect.Size.x / RadarConfig.WidthNm + viewportRect.Size.y / RadarConfig.HeightNm) / 2 * zoomValue,
        _ => throw new NotImplementedException()
    };
}
```

Listing 5: Calculating the scale

The position in nautical miles of a point can then simply be multiplied by this value to get its final screen position.

```
[Signal] public delegate void ScaleChangedEventHandler();

public override void _Process(double delta)
{
    float scale = Scale(_viewportRect);
    if (scale != _previousScale)
    {
        EmitSignal(nameof(ScaleChanged));
    }
    _previousScale = scale;
}
```

Listing 6: Notifying other modules of a change in scale

4.2 Aeroplane physical simulation

The heading of an aeroplane describes the direction it is pointing in, which is not necessarily the same as the direction it is travelling in. In stable flight the main contributor to this discrepancy is wind, see figure 12.

```
[Export] public float TrueHeading
{
    get => _heading;
    set => _heading = Mathf.Wrap(value, 0, 360);
}
```

Listing 7: Wrapping true heading value

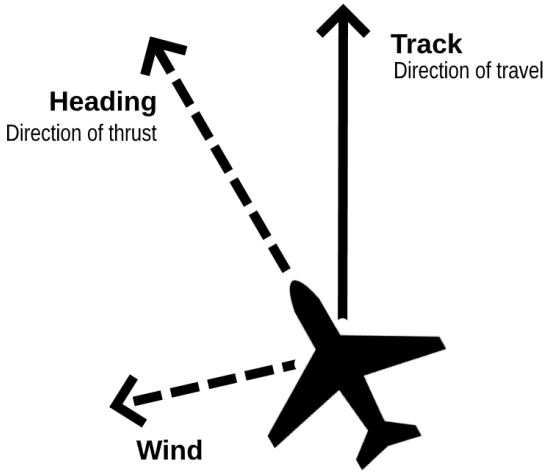

Figure 12: The wind triangle

To calculate a vector representing the movement of an aeroplane over the ground the heading vector and wind vector are added, where the heading vector is the aeroplane's direction vector multiplied by its airspeed and the wind vector is the wind direction vector multiplied by the wind speed.

```
Vector2 airVector = HeadingToVector(TrueHeading) * TrueAirspeed;
Vector2 windVector = HeadingToVector(_simulator.WindDirection) * _simulator
    .WindSpeed;
Velocity = (airVector + windVector) / SecondsInAnHour * (float)delta;
PositionNm += Velocity;
```

Listing 8: Extract of the aeroplane physics process

4.2.1 Converting a heading to a direction vector

To convert a heading to a direction vector it is necessary to determine which quadrant the heading lies in. The angle the heading makes with the y axis can then be calculated. Then the sine rule

can be used to calculate the positive x and y components of a unit vector with that angle. Knowing which quadrant the heading lies in, the sign of the components can be changed appropriately by multiplying each by 1 or -1.

```
private Vector2 HeadingToVector(float heading)
{
  Vector2 quadrant = new Vector2(1, 1);
  float theta = heading;
  if (heading > 270)
  {
    theta = 360 - heading;
    quadrant = new Vector2(-1, 1);
  }
  else if (heading > 180)
  {
    theta = heading - 180;
    quadrant = new Vector2(-1, -1);
  }
  else if (heading > 90)
  {
    theta = 180 - heading;
    quadrant = new Vector2(1, -1);
  }
  return new Vector2(Mathf.Sin(Mathf.DegToRad(theta)), Mathf.Sin(Mathf.
    DegToRad(90 - theta))) * quadrant;
}
```

Listing 9: Converting a heading to a vector

4.3 Aeroplane navigation simulation

4.3.1 Glideslope

Because it would be difficult to add the debugging features necessary to the main project, I chose to create a python program which would run a simplified simulation and allow me to more quickly iterate on the glideslope vertical mode parameters and see the result. I initially chose to use a PID controller[7] because it is probably similar to what is done in real life, however it was difficult to get it to perform accurately. Therefore I chose to use a simpler solution which would, based on the time necessary to pitch down to match the glideslope's angle, calculate the point at which to command pitch down in order to align with the glideslope's path.

4.4 Aeroplane display

TODO

4.4.1 Tag

```
public override void _Draw()
{
    // Drawing the line from the blip to the tag

    // Get the inner rect, adjusting its position because its otherwise
    local to its parent
    Rect2 innerRect = _innerControlArea.GetRect();
    innerRect.Position += TagDisplay.Position;
    // Intersect with the outer rect when hovering and the inner rect when
    not
    Rect2 tagRect = Hovering ? TagDisplay.GetRect() : innerRect;

    // Define start and end points as blip position and tag centre
    Vector2 end = innerRect.GetCenter(); //tagRect.GetCenter();
    Vector2 start = Position + Position.DirectionTo(end) * 10;

    // Define line from blip to tag centre
    float m = (start.y - end.y) / (start.x - end.x);
```

```
    float c = start.y - m * start.x;

    // Calculate intersection points for all four lines
    float x = innerRect.Position.x;
    Vector2 p_a = new(x, m * x + c);
    float y = innerRect.Position.y;
    Vector2 p_b = new((y - c) / m, y);
    x = innerRect.Position.x + tagRect.Size.x;
    Vector2 p_c = new(x, m * x + c);
    y = innerRect.Position.y + tagRect.Size.y;
    Vector2 p_d = new((y - c) / m, y);
    List<Vector2> points = new() { p_a, p_b, p_c, p_d };

    // Draw line from start to closest intersection point
    // Do not draw the line if the tag is over the blip
    if (!tagRect.HasPoint(start))
    {
        DrawLine(start, points.MinBy(point => point.DistanceSquaredTo(end - end.Normalized() * 10f)), Colors.White, 1, true);
    }
}
```

Listing 10: Drawing the tag line

4.5 Waypoints

TODO

```
d
```

Listing 11: Waypoints generation

```
public partial class Waypoint : Sprite2D
{
    public WaypointData WaypointData;
    public Vector2 PositionNm;

    public override void _Ready()
    {
        // Set icon coressponding to waypoint type
        Texture = WaypointData.Basis switch
        {
            WaypointData.Type.RNAV => Simulator.RadarConfig.Style.RNAVTexture,
            WaypointData.Type.VOR => Simulator.RadarConfig.Style.VORTexture,
            WaypointData.Type.VORDME => Simulator.RadarConfig.Style.VORDMETexture,
            WaypointData.Type.NDB => Simulator.RadarConfig.Style.NDBTexture,
            _ => throw new NotImplementedException()
        };

        // Set name
        GetChild<Label>(0).Text = WaypointData.ResourceName;

        // Calculate position relative to screen centre
        PositionNm = Geo.RelativePositionNm(WaypointData.LatLon, Simulator.RadarConfig.LatLon);
    }

    public override void _Draw()
    {
        Position = Simulator.ScaledPosition(PositionNm, GetViewportRect());
    }

```

```
30      public void OnScaleChanged()
31      {
32          QueueRedraw();
33      }
34  }
```

Listing 12: Waypoint class

4.6 Geographical features

```
1  public override void _Ready()
2  {
3      // Read GeoJSON polylines
4      var polyLines = JSON.ParseString(Simulator.RadarConfig.GeoLines.data).
       AsGodotArray();
5      foreach (var polyLine in polyLines)
6      {
7          // Get the relative position of each point in the polyline
8          List<Vector2> points = new();
9          foreach (float[] point in polyLine.AsGodotArray())
10         {
11             Vector2 PointNm = Geo.RelativePositionNm(new Vector2(point[1],
       point[0]), Simulator.RadarConfig.LatLon);
12             points.Add(PointNm);
13         }
14         _polyLines.Add(points);
15     }
16 }
17
18 public override void _Draw()
19 {
20     GD.Print("Drawing terrain");
21     foreach(List<Vector2> polyLine in _polyLines)
22     {
23         // Scale the points in the line
24         Vector2[] scaledPolyline = polyLine.Select(PointNm => Simulator.
       ScaledPosition(PointNm, GetViewportRect())).ToArray();
25         DrawPolyline(scaledPolyline, Simulator.RadarConfig.Style.
       CoastlineColour);
26     }
27 }
28
29 public void OnScaleChanged()
30 {
31     QueueRedraw();
32 }
```

Listing 13: Drawing Terrain

4.7 Camera pan and zoom

TODO

```
1  private void SetReference()
2  {
3      _initialMousePosition = GetViewport().GetMousePosition();
4      _initialCameraPosition = Position;
5  }
6
7  public override void _Input(InputEvent @inputEvent)
8  {
9      if (inputEvent.IsActionPressed("Pan Camera"))
10     {
11         SetReference();
12     }
```

```
    else if (@inputEvent.IsActionPressed("Zoom In"))
    {
        if (Simulator.Zoom < MaxZoom)
        {
            Simulator.Zoom++;
            // Move the camera to compensate for stretching of distances
            Position *= 1f + Simulator.ZoomSpeed;
            SetReference();
        }
    }
    else if (@inputEvent.IsActionPressed("Zoom Out"))
    {
        if (Simulator.Zoom > MinZoom)
        {
            Simulator.Zoom--;
            // Move the camera to compensate for stretching of distances
            Position /= 1f + Simulator.ZoomSpeed;
            SetReference();
        }
    }
    else if (@inputEvent.IsActionPressed("Reset Camera"))
    {
        // Reset position and zoom
        Simulator.Zoom = 0;
        Position = Vector2.Zero;
        SetReference();
    }
}

public override void _Process(double delta)
{
    // Panning
    // Move the camera in the opposite direction to mouse movement, from
    the reference point
    if (Input.IsActionPressed("Pan Camera"))
    {
        Vector2 mouseDelta = GetViewport().GetMousePosition() -
    _initialMousePosition;
        Position = _initialCameraPosition + new Vector2(-mouseDelta.x, -
    mouseDelta.y);
    }
}
```

Listing 14: Camera pan and zoom

4.8 Radar definitions

TODO

4.9 Main menu

TODO

5 Evaluation

5.1 Function and robustness testing

5.2 Usability testing

5.3 Conclusion

References

[1] U.K. CAA. “How air traffic control works”. In: (2022). URL: https://www.caa.co.uk/Consumers/Guide-to-aviation/How-air-traffic-control-works/.
[2] NATS. “Trainee Air Traffic Controllers”. In: (2023). URL: https://www.nats.aero/careers/trainee-air-traffic-controllers/.
[3] Gaurav Joshi. “India Is Facing A Shortage Of Air Traffic Controllers”. In: (2022). URL: https://simpleflying.com/india-shortage-air-traffic-controllers/.
[4] NATS. “Being a Controller - The buzz”. In: (2019). URL: https://vimeo.com/363827878.
[5] Wikipedia. “Waypoint”. In: (2022). URL: https://en.wikipedia.org/wiki/Waypoint.
[6] Wikipedia. “Great-circle distance”. In: (2022). URL: https://en.wikipedia.org/wiki/Great-circle_distance.
[7] Wikipedia. “PID Controller”. In: (2022). URL: https://en.wikipedia.org/wiki/PID_controller.

Appendices

A Other Code

B Other Images

Figure 13: Airport appears to be in the sea in ATC-SIM