Thesis

Motion Planning for Autonomous Vehicles in Ludic Simulation Environments

James Blair

N00162202

Report submitted in partial fulfilment of the requirements for the BSc (Hons) in Creative Computing at the Institute of Art, Design and Technology (IADT).

Table of Contents

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by some-one else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Signed: James Blair_____

Date: 23/04/2021_____

Failure to complete and submit this form may lead to an investigation into your work.

Abstract

Path following behaviour is an area of emerging significance in the field of computer autonomy and, as such, this paper presents an exploration into its applications to real world data through the development of an air traffic control simulation with Dublin Airport's arrival pattern as the accompanying data. The simulation is developed with the utilisation of several key game development techniques and aims to explore the underutilised potential of applications that exist in the narrow spectrum between game and simulation.

The main body of the document is structured to begin with the performed research on the relevant areas, followed by a study of the application's feasibility and technological requirements. The application's design is then discussed to outline the initial plan for both functionality and aesthetic. Finally, implementation and testing of the application is explained.

The findings, as collected through testing of both the technology's performance as well as user performance, suggest that the applications in this space may benefit mutually from aspects of both game and simulation and that the effectiveness of autonomous agency and steering behaviours on fulfilling the goals of applications such as these is very significant.

Acknowledgements

I would like to express my deepest appreciation to my thesis supervisor John Montayne who has, throughout every stage of this project, provided valuable insight. His passion for aviation and deep understanding of user aware systems saved me from many pitfalls. Without his constant encouragement to strive for better than I believed myself capable, this project would not be what it eventually became.

I would also like to extend my gratitude to my second reader, Cyril Connoly, who contributed meaningful direction to the project early in development.

Finally, I would like to thank my partner, Sarah Connaghan, without whose constant love, support and academic guidance I could not have made it this far.

1. Introduction

In video games, it is often essential to navigate objects from one position to another. In these scenarios, paths are typically calculated from one point to another using an algorithm. Autonomous agents, those that may be affected by other forces while attempting to navigate a path, can create challenges in requiring constant re-calculation of the most suitable path to follow and, as such, are instead given behaviours that will attempt to follow paths within their own unique physical constraints. The proposed project attempts to develop an application which implements some of the described behaviours (steering, pathfinding) on a set of real world data objects, in this case air traffic control and node network positional data.

The application will also explore the validity of building simulation training tools informed by design principles used in game development in order to improve comprehension of complex systems like those used in air traffic control. In doing so the project aims to demonstrate the benefits and limitations of programs that exist in this space.

2. Research / Literature Review

In the development of an application which intends to simulate the behaviour and movement of air traffic, it became evident that it was necessary to employ techniques to handle each individual object's means of navigation. 'Motion planning' or 'path planning' are terms used to describe many methods by which agents may navigate their environments and are frequently referred to in research surrounding video games, robotics and autonomous vehicles. Most typically this can be described as moving an object from a specific point in space to another. This, however, often presents many challenges such as the object's physical limitations or a change in the environment the object intends to navigate. This research document intends to explore with the aim of uncovering the most effective possible motion planning solutions with regards computation speed and processing efficiency for the specific use case presented by the application.

The use case presented by the air traffic control simulation can be broken down as follows: The environment will contain no obstacles as it simulates an air space, the environment will contain a series of nodes which will be key points of navigation, there will be a multitude of objects in the environment at once requiring real time pathing information, the objects must not be permitted to collide with one another. To understand the most effective means of moving objects in this environment, this thesis will examine solutions presented in works outlining path finding and path following behaviour as well as determining the most effective means of translating rules of an air traffic system by means of gamification to a digital environment.

2.1 Pathfinding Algorithms

Within motion planning, a common sub category includes the concepts of 'path finding' which usually describes the means of calculating the shortest path between two points in an environment containing obstacles. There are a variety of path finding algorithms each with strengths and weaknesses pertaining to their processing efficiency and the type of environment they are being applied to. The algorithms A* is the most commonly represented as this provides the optimal solution for path finding when compared to other search algorithms (Zafar, A., Agrawal, K. K., & Anil Kumar, Wg. C. (2018)) and as such the literature examined to determine its effectiveness are comparative of this algorithm

and others such as Djikstra, Breadth-First Search (BFS), Depth-First Search (DFS) and Best-First Search.

In Permana, Bintoro, Arifitama, & Syahputra (2018). 'Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game' a test is carried out in order to evaluate users' performance within a game 'Maze Runner' in which the player attempts to compete with an object which is attempting to navigate an environment using a pathfinding algorithm. The players place blocks which attempt to intercept the current path and force the object to recalculate.

The objective as stated for this research was to determine the shortest and most efficient route search. To accomplish this three levels were played by each researcher with each level containing a different algorithm, these being A*, Djikstra and BFS. This was repeated three times with an increasing level of complexity added in each iteration. The data recorded included processing efficiency, computation speed and number of steps taken.

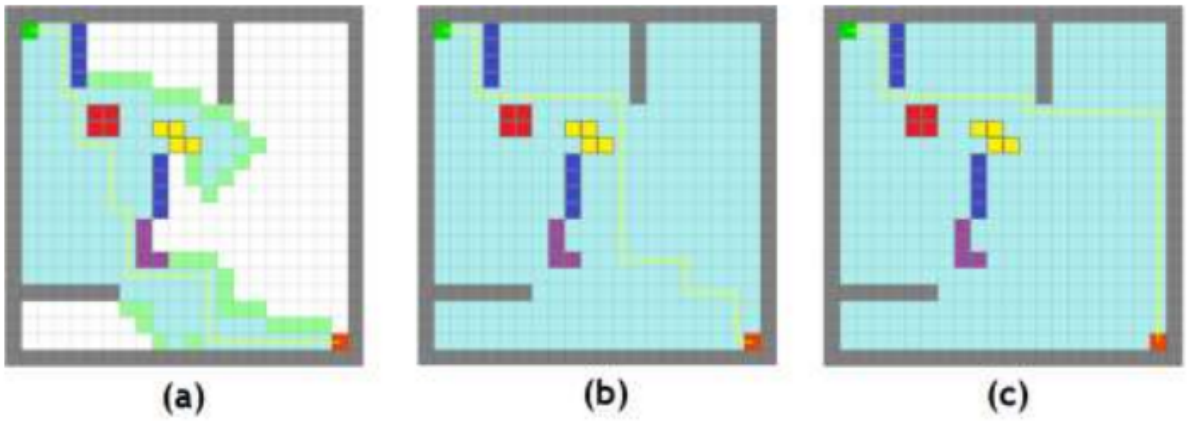


Figure.1 Pathfinding of Maze Runner Game at Level 1 using A* (a), Djikstra (b), and BFS (c) algorithms - (Source: Permana, Bintoro, Arifitama, & Syahputra, 2018)

From the computation results presented in figure.1 it is evident that the A* algorithm appears to be acting more efficiently as the method by which it searches the environments nodes reduces the quantity of iterations required. The results however determined that it calculated in this test at a slower speed than Djikstra's algorithm despite this algorithm checking every node in the environment as opposed to A* which only checked

approximately 50%. It is not until the environment becomes more complex in later tests that the A* algorithm manages to perform faster. In all cases however the BFS algorithm is less efficient.

Harika Reddy's PATH FINDING - Dijkstra's and A* Algorithm's takes a narrower approach to the subject matter by analysing just two of the algorithms utilized in the previous paper. From the results demonstrating the seeming redundancy of BFS in the comparative research carried out, it is apparent that this is appropriate. In this paper the author goes into greater detail regarding the history and inner functionality of these algorithms. The goal of this research appears to be a breakdown and comparison of the uses and pitfalls of each algorithm however no specific testing is performed in order to demonstrate this.

Reddy is able to convey that the Djikstra's algorithm uses a 'greedy' in that it repeatedly selects unselected vertices (or nodes) nearest to the starting location until it finds the shortest distance to the target. Using Big-O notation Reddy evaluates the efficiency of the algorithm and is able to determine its pitfalls. The issues with this algorithm lie in it's inefficiency of direction. The algorithm attempts to find the target without eliminating unlikely directions and with a limited view of only its current node's nearest neighbours, thus resulting in a far higher number of checks to return a result. Reddy describes this as a blind search which she concludes consumes necessary resources and wastes time. (Reddy, H 2013 p.9)

Taking a similar detailed approach to the A* algorithm, it is then outlined that the structure by which A* operates is that "As A* traverses the graph, it follows a path of the lowest known heuristic cost" (Reddy, H 2013 p.7). This is similar to the operation of the Djikstra's algorithm however A* benefits from ensuring it does not re-test nodes that have already been travelled upon.

The only notable pitfall pointed out in this paper of the A* algorithm is the necessity of having access to the entire grid of traversable nodes before beginning the calculation. This can cause complications when the quantity of agents attempting to navigate increases as the computational burden grows exponentially with each iteration.

The first paper manages to demonstrate this through a testing method which provides a tangible quantitative result while Reddy's paper tends to discuss more in depth how the algorithms work in a more literal sense using pseudo code examples and historical background as a basis for explanation. Reddy's paper could be seen as more accessible due to the nature of the language used but is less equipped than Permana's paper in delivering information which facilitates decision making for readers who may be developing an application.

From both of these papers it is apparent that the consensus is that in most practical cases A* is the preferred algorithm however there exist niche cases where the Djikstra's algorithm may be preferred, specifically cases with very large spaces and very few obstacles. It is possible that the proposed application using an open space to simulate air traffic may be a niche case which may benefit from using Djikstra's algorithm; however the method of implementation of the nodes representing the air transport network will determine whether or not this is the case.

2.1.2 Path Smoothing

In order to more closely emulate the movement of the objects proposed in the application, aeroplanes, it is essential that the objects move according to the limitations of these vehicles. To this avail a common solution to the paths generated by grid based pathfinding algorithms pixilation is to smooth the resultant paths prior to applying them to the object in the environment.

The Open Source Software (OSS) project: PythonRobotics is a collection of robotics algorithms implemented in the Python programming language. The focus of the project is on autonomous navigation, and the goal is for beginners in robotics to understand the basic ideas behind each algorithm" (Sakai, et al., 2018). While this project explores many solutions to motion planning as a whole, it moves past traditional pathfinding methods and explores the generation of curves by means of the generation of splines and bezier curves in particular making reference of 'A comparison study on path smoothing algorithms for laser robot navigated mobile robot path planning in intelligent space'(Zhao, et al., 2011). This approach sits in contrast to the approaches outlined in 'Game Path Planning' (Ceipek, 2010) which examines a method of path smoothing typically used in video games and robotics which involves the conversion of an

environment into graphs of nodes and generating paths between these nodes in fewer, and smoother patterns.

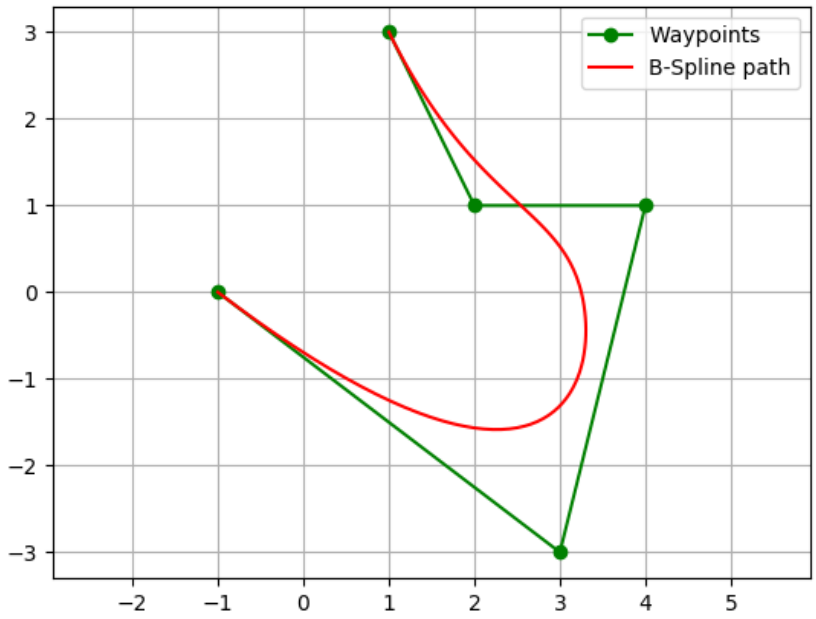


Figure.2 B-Spline Planning PythonRobotics. (Source: Sakai, et al., 2018)

In Song, Tien and Zhao's research an explanation into the mathematics and functionality of several parametric curves is presented including Hermite interpolation, Cubic spline, Bezier curve and B-Spline. The purpose of which is to compare their performance under a single specific laboratory map and to select a path planning technique for use with a mobile robot in this environment. The paths generated for the purpose of smoothing using these curves is Djikstra's algorithm and a Voronoi diagram. While they are able to determine that for the purposes of their implementation the bezier curve presented the most efficient path, it is possible that the proposed airspace environment that skipping arriving at certain nodes could result in a greater risk of congestion and therefore may not be the most effective curve for this case.

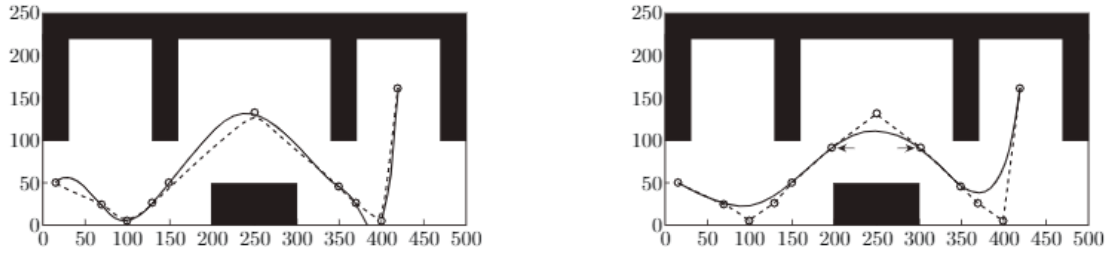


Figure 3. Left: Parametric Curve planning vs Right: Bezier Curve Planning (Source: Zhao, et al., 2011)

The approach taken by Ceipek in his more causally presented article on game path planning discusses the use of navigation meshes (Nav Mesh) in video games to create more efficient paths and a means of iterating further upon this strategy using the 'funnel' algorithm. Ceipek presents his work as a tutorial, describing the construction of the nav mesh as a graph of weighted nodes identical to descriptions used prior in this paper describing heuristic weighting in path finding algorithms.

Ceipek presents that the common use of nav meshes is often used to navigate objects between points but that this methodology is flawed. "Waypoint representations of space are easy to create and think about, but they are not very good for pathfinding. Although they use very little memory, they result in inefficient, unrealistic paths."(Ceipek, 2011). The article then demonstrates the use of A* to calculate the shortest path between two points on a nav mesh. Following this an implementation of the funnel algorithm as outlined in 'If you never fall, you are not really trying' (Jon, 2006 p.52), is implemented. This involves triangulating points along the path and determining the shortest path along the points of these triangles and then examining the triangles cross product to determine the order of the steps along these points.

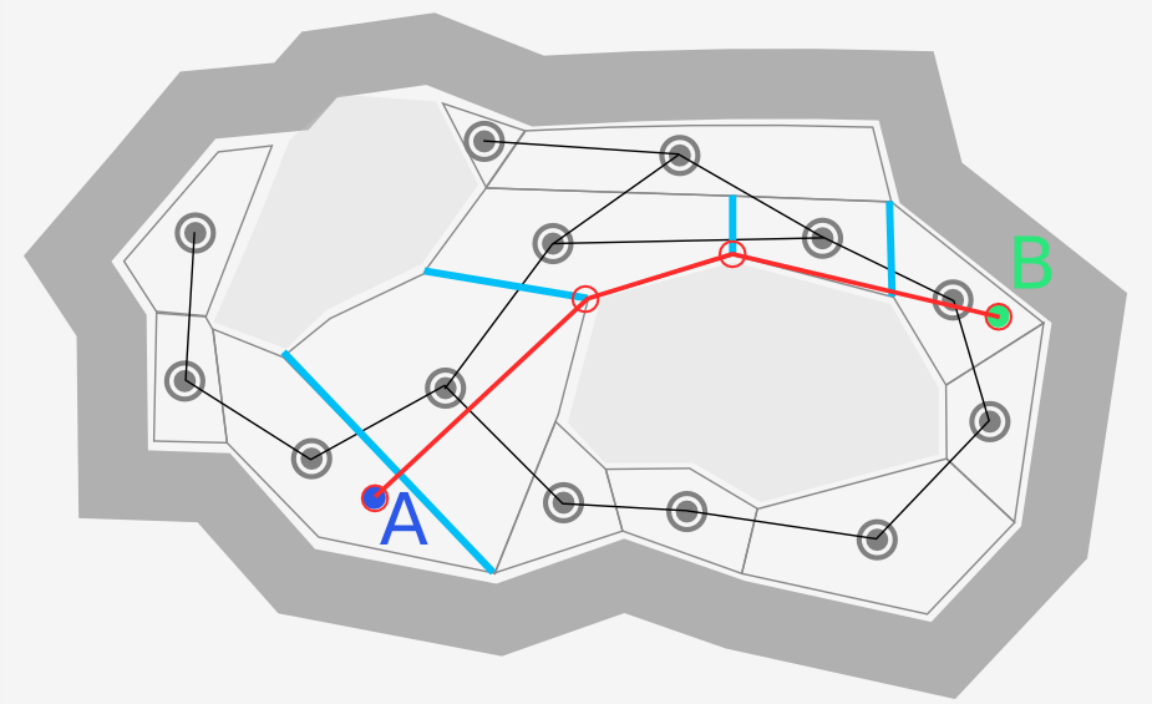


Figure.4 In Black: Nav Mesh, In Red: Proposed path using funnel algorithm (Source: Zhao, et al., 2011)

Although the resultant paths presented by this method are not curved it is possible this could be improved upon by calculating a larger sample of control points. In the application proposed for air traffic control simulation it may be necessary to implement a combination of these methods and perform testing to determine the best method.

While the subject matter of path smoothing is one for which there are many solutions it is evident that the use case scenario for each implementation will dictate not only which types of paths will be most efficient but even what may be feasible from a computational standpoint.

2.2 Path Following

Regardless of the methods used to generate paths for objects in a simulation environment or video game, it is often necessary to allow the objects to handle a quantity of the processing burden by allowing the objects to do more than simply move a specific number of steps along the path with each iteration. Instead, it is proposed by Craig Reynolds in his work in ‘Steering Behavior for Autonomous Characters’(1999) and

iterated further upon in ‘Autonomous Behaviors for Interactive Vehicle Animations’(Go, et al., 2004) that objects in a simulation space (‘Agents’) may instead be given some degree of agency over how they can follow a path. This behaviour, entailing steering, seeking, arriving, evading and queuing among others allows agents, particularly in environments containing multiple agents, to attempt to produce more realistic simulations by interacting more dynamically with one another and the environment. Reynolds describes this behaviour as navigating around their “world in a life-like and improvisational manner”.

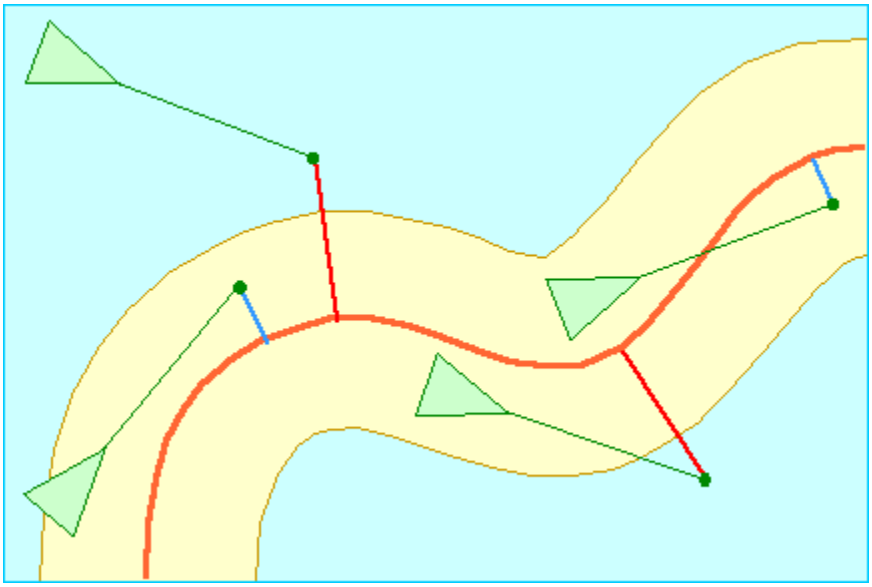


Figure.5 Example of path following behaviour of autonomous characters along a curved path (Source: Reynolds, 1999)

Reynolds’ paper discussed the levels of motion behaviour of autonomous characters in animation and video games. It defined these three levels of motion behaviour in a hierarchy as ‘Action Selection (strategy, goals, planning) Steering (path determination), Locomotion (animation, articulation)’ (Reynolds, 1999). Reynold’s paper presents definitions of terms and a limited literature review on related works. The most useful information contained was in relation to the structure he outlines for core principles of what defines steering behaviour. In particular his final section on the topic of combining behaviours. This section explains the means by which autonomous objects can be offered

a low level decision making process to determine which behaviours to prioritize, and to what degree, based on external factors and neighbours as opposed to sequentially switching between different behaviours.

In ‘Autonomous Behaviors for Interactive Vehicle Animations’(Go, et al., 2004) an adaptation of Reynolds’ work to allow for vehicles with further, more complex movement behaviours in both two dimensional and three dimensional space. The primary method presented by the article to achieve this goal of extending Reynolds’ model is to combine the behaviours with “with online path planning techniques to yield more visually realistic synthesized vehicle animations.”(Go, et al., 2004). The primary benefit of the improvements made to the previous model of steering behaviour is in the development of a control model in which a calculation of a fixed number of trajectories based on the specific parameters of the vehicle is made. The paper outlines a method by which a vehicle with limited controls can have many of its current, most likely trajectories mapped ‘offline’ before initializing the simulation for later use during ‘online’ simulation.

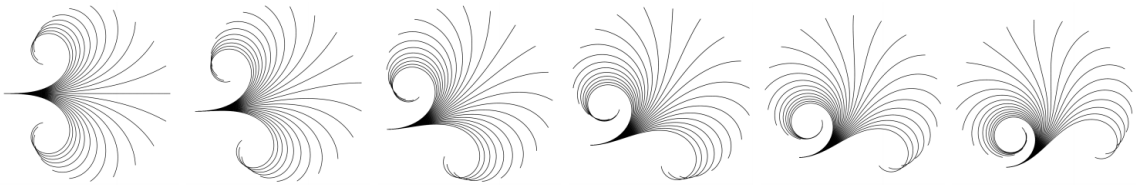


Figure.6 Trajectory traces for an autonomous animated spacecraft in 2D with only yaw control inputs. 6 different values for controls (Source: Go, Vu, & Kuffner, 2004)

This work by Go, Vu and Kuffner presents a solution to broaden the applications of steering behaviours by the use of this trajectory pre-calculation and further enables its use by demonstrating how it can be used to combine behaviours for collision avoidance by sampling large quantities of paths at once using three different methods. These methods allow for varying quantities of accuracy however certain iterations can result in large amounts of memory usage depending on the vehicle in question and the variability of its possible movements.

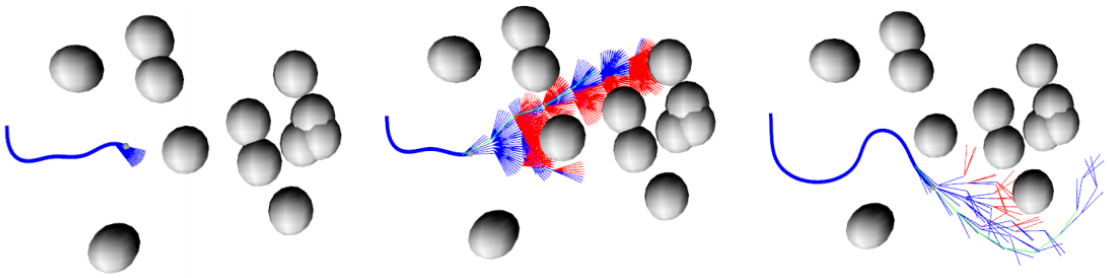


Figure.7 From left to Right: Single step sampling, uniform sampling and adaptive sampling (Source: Go, Vu, & Kuffner, 2004)

Limitations of Reynolds' paper were in the shallow depth it went into terms, however the work presented in the paper is reinforced by numerous demonstrations of each behaviour's functionality on his website (Reynolds, 2004). The article is also perhaps limited by its age. The demonstration of potential improvements to this concept provided in 'Autonomous Behaviors for Interactive Vehicle Animations'(Go, et al., 2004) only five years following its publishing highlights the possibility that many such improvements in this area may be present today. Go, Vu and Kuffner's solution provides additional areas of application for the strong basic model laid out by Reynolds however it is pointed out in their article that the solution presented contains several issues, most notably the generation of what the authors call 'path aliasing' which is the inability of the presented solution to provide straight paths due to constant recalculations of the next trajectory. The article itself, though, provides significant depth and context for the discussed subject matter and is able to convey its thesis effectively as a result.

The applications of the presented concepts of these papers to the proposed air traffic control application are numerous. Allowing vehicles in the simulation to move autonomously presents the opportunity to reduce the necessary complexity of the path finding algorithm which, in real time simulation, may be necessary to produce an application which can dynamically change due to user input. Specifically the behaviours governing collision avoidance, path following and queuing may provide particular relevance to the simulation of a congested airspace. The additional models outlined in

Go, Vu and Kuffner's work also presents interesting possibilities to this application. Due to the focus of their work seeming to be in the enabling of particularly complex vehicles, it is still undetermined if the aeroplanes modelled in the application will benefit from this degree of trajectory pre-sampling.

2.3 Simulation and Gamification of Rules

When creating a simulation of a real world environment whether for the intent of analysis or entertainment it is usually pertinent to closely examine the rules governing this environment in reality and to replicate them to as close a margin as is feasible. It is inevitable however that at certain points in this process it will be necessary to disregard or simplify certain variables that are either too complex or too random to predict. Furthermore in the development of video games, the factor which drives these decisions is often alternatively founded in whether the simulated variable contributes to the enjoyment or immersion of the user instead. In addition to the emulation of specific factors there is also consideration given to the intent of a piece of software. This being whether the software is intended as a purely entertainment based product or as a tool for creation or training. To this effect this section will examine the work of Brock Dubbels In his paper titled 'Gamification, Serious Games, Ludic Simulation, and other Contentious Categories' (2013) in which a framework is employed to further understand this process. This will be compared to the work of James R. Parker and Katrin Becker titled 'The Simulation–Game Controversy: What is a Ludic Simulation?'(2014) which, attempts to alternatively more closely explore the concept of ludic simulation and the relationship between work and play within the scope of software applications.

Dubbels' paper is divided into two distinct sections. The first outlines the design of a conceptual framework he titles the spectrum of gamification, by which software can be categorized on the spectrum of game and simulation through many factors, these being work vs play, consequence vs ambiguity, mimesis vs diegesis and narrative vs story. Dubbels outlines an area within the three axes which determines games. The second section of this paper makes use of this framework to perform "feature analysis of traditional categories known as models, simulations, and games to compare with hybrid categories known as ludic simulations, gamification, and serious games"(Dubbels , 2013).

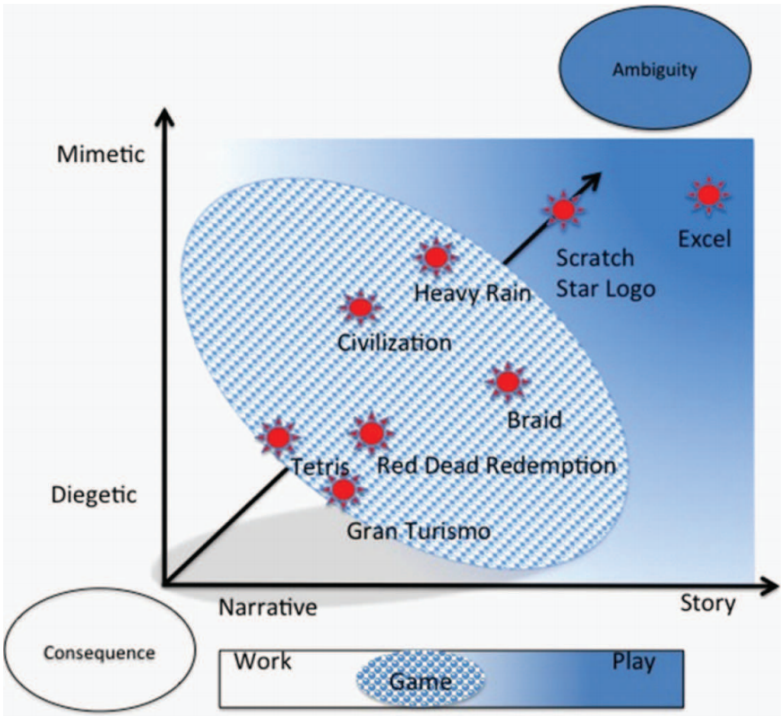


Figure.8 Gamification Spectrum with Examples (Source: Dubbels , 2013)

In figure.8 examples of Dubbels categorization of notable games and tools can be seen to demonstrate the use of this system. Despite clear outlines provided in the paper for the descriptions of each category and the reasoning for the placement of many of the examples provided, the framework has no means by which a checklist is used to quantify exactly where a subject may be placed and is therefore mostly a means of approximation. The descriptions regarding these reasonings for categorisation are perhaps the most beneficial section of the paper as they provide a means of distinguishing key differences in games, simulations and tools. This can be observed in Dubbels' discussion of RAC's (Reward-Action Contingencies) which are the means by which games "provide clear signals that increase the player's sense of empowerment, knowledge, control and potential"(Dubbels , 2013). This is something that most tools, training and even simulations do not provide within the software, often opting alternatively to provide context through the use of external training literature or documentation.

Parker and Becker's work takes a similar approach in outlining the understood definition of, in particular, games and simulations. Following these initial brief explanations the

article moves to examine the relationship between the two, coming eventually to the statement that " games are simulations but not all simulations are games."(Parker & Becker, 2014). This is particularly significant as the paper moves closer to describing the thesis' question regarding ludic simulation, they are able to outline that the only core difference between the game and simulations is the presence of a goal. That a simulation may have graphics, sound, interactivity or even a game-like scoring mechanism does not define it as a game unless there is a central goal to the purpose of the user's interaction.

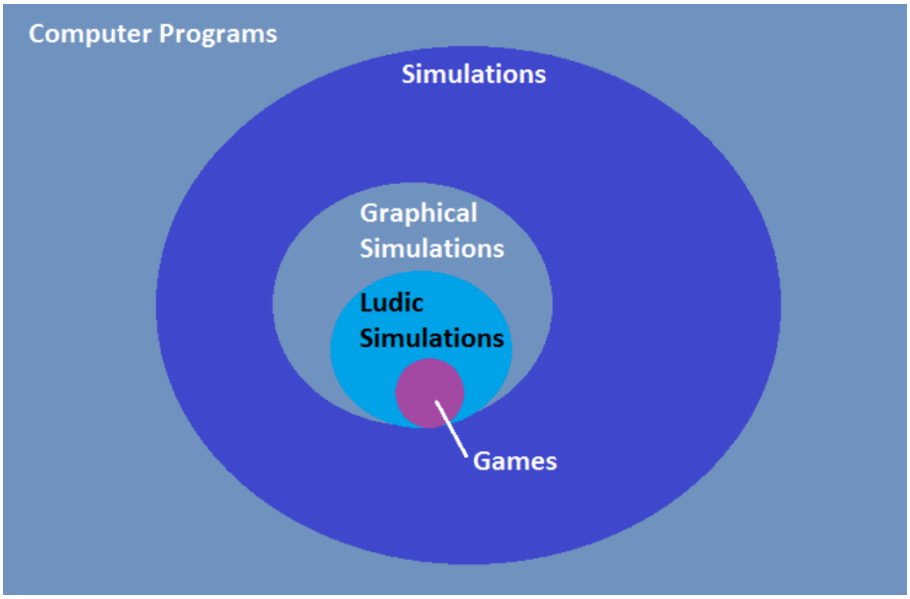


Figure.9 Relationship between games and various types of simulations (Source: Parker & Becker, 2014)

Parker and Becker conclude that "The lack of a clear delineation between levels of the hierarchy must not be allowed to impact design, development, or analysis of simulations"(2014). They describe that the two closely related topics can benefit greatly from one another through use of the strengths of each.

Dubbel's paper provides a valuable context to understand means of classifying software and the most commonly observed identifiers of these categories. This is achieved through use of accessible language and relevant examples however the paper is perhaps limited by the vagueness of its intent. While the framework provided offers a solution to

understanding existing software it does much less so to help determine choices a designer may take to insert their own project into this spectrum nor to provide a reason to do so. In contrast Parker & Becker are able to deconstruct the definitions in a way which allows these sorts of evaluations to be made. They place more emphasis on the strengths and weaknesses of both games and simulations and how the design and development of each may be bolstered through comprehension of their definitions.

This area of study is one that is of particular relevance to the proposed air traffic control simulation as at this stage in the design, the degree to which gamified elements will be included is still vague. Including systems by which there is a progressive introduction of more complex features of the application over time to the player can only serve to enhance the user experience however requires a more modular approach to the component design of the application, with each instance of functionality having the capacity to be either automated or user controlled.

2.4 Summary of Literature Review

This literature review has examined principles surrounding the key technological area the proposed project intends to implement in the form of both path finding and path following. From research into these topics the review was able to determine that it is essential to fully factor the use case and perform a variety of tests to determine the most efficient method of motion planning for a given project. It also revealed that motion planning can be a somewhat layered approach, with each technique adding additional quantities of efficiency, definition or sampling to existing calculated paths. Finally, from this research it became evident the importance of understanding the intent of a simulated piece of software with regards to gamification and user experience. These factors as explored in the previous chapter unveiled the significance of identifying where on the spectrum of game to simulation the proposed project will fall.

3. Feasibility Study & Requirements

3.1 Requirements Analysis

The initial process undertaken to plan and prepare the relevant research for the was the undertaking of a requirements and feasibility study. This process involves the collection of a broad range of relevant information for the planned project as well as demographic analysis and risk assessment. It is the objective of this study to create a clearer set of short term goals for the project and lay a foundation for the plan in terms of both design and implementation.

3.1.1 Existing applications

The model of steering and following behaviour often used or replicated in game development is perhaps best represented in Craig Reynolds work on flocking and avoidance behaviour. In particular his C++ library OpenSteer for autonomous characters in video games and animations which was developed with assistance from Sony Computer Entertainment of America. This library allows game developers to prototype, visualize and annotate steering behaviours during game development with the intention of then being reconstructed in the developer's native game engine.

The application which this project intends to develop presents a complete environment that informs its own behaviours independent of a separate library as well as the simulation of air traffic in real time. While OpenSteer is a tool which enables developers to create behaviours similar to the behaviours this project aims to develop, this project's goals are conversely in the analysis of the effectiveness of the technique of utilizing steering behaviours for the specific use case of air traffic control simulation.

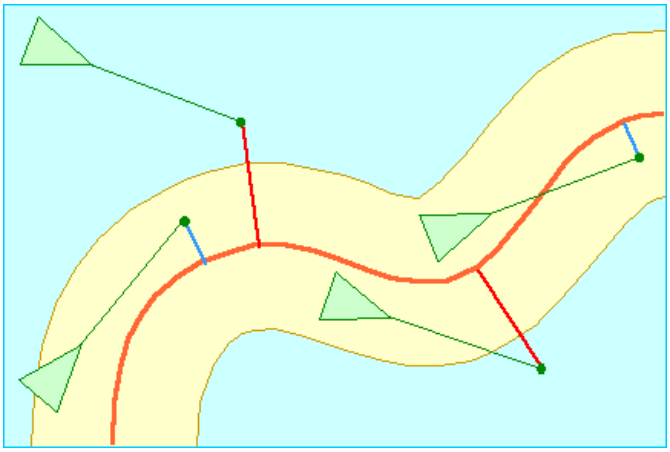


Figure.10 - Steering Behaviors for Autonomous Characters Example (Source: Reynolds, 1999)

With regards to the game and simulation space the application will occupy, it can be compared to similar gamified air traffic control applications such as Global ATC Simulator (*Global ATC Simulator*, n.d.). While many examples of such games exist, they often do not provide real traversal information of the vehicles, opting instead to update the positions by a fixed amount over time. The proposed application would also include features such as the capacity to create new air spaces from correctly formatted data and as such presents a much more significant capacity for scalability.

Figure.11 Global ATC Simulator, (Source: *Global ATC Simulator*, n.d.)

Professional, non-commercially available systems used in actual air traffic control such as the proposed solution by WEYTEC pictured in fig.3 are immensely complex and require several years of training to effectively operate. The application attempts to present a much more accessible system which can be adjusted to in a much shorter period of time through the use of gamification albeit with less depth with regards to micro-mechanics handled in these sorts of systems.

Figure.12 WEYTEC proposed application based ATC solution (Source: Greenky, 2019)

3.1.2 User Profile

The typical user of this application is an aviation hobbyist wishing to gain an entry level perspective of air traffic control systems. The reason such a person may seek out this program over others is the proposed high level of accessibility due to presentation using game systems as well as the capacity to have specific air spaces included through its modular design. This user is typically in search of a tool which will enable them to gain a greater understanding of a complex subject and as such their likely desires have been outlined:

- A User Interface that is easy to navigate and comprehend
- An application which is clear in its presentation of information and the significance of that information.
- An application which has familiar air spaces
- An application that is potentially enjoyable to use.

3.1.3 Personas

The creation of user personas documents was carried out in order to establish a clear picture of the user profile. It was in creating these that specific wants of these fictional individuals which had, of yet, not been considered in the development plan, would become apparent.

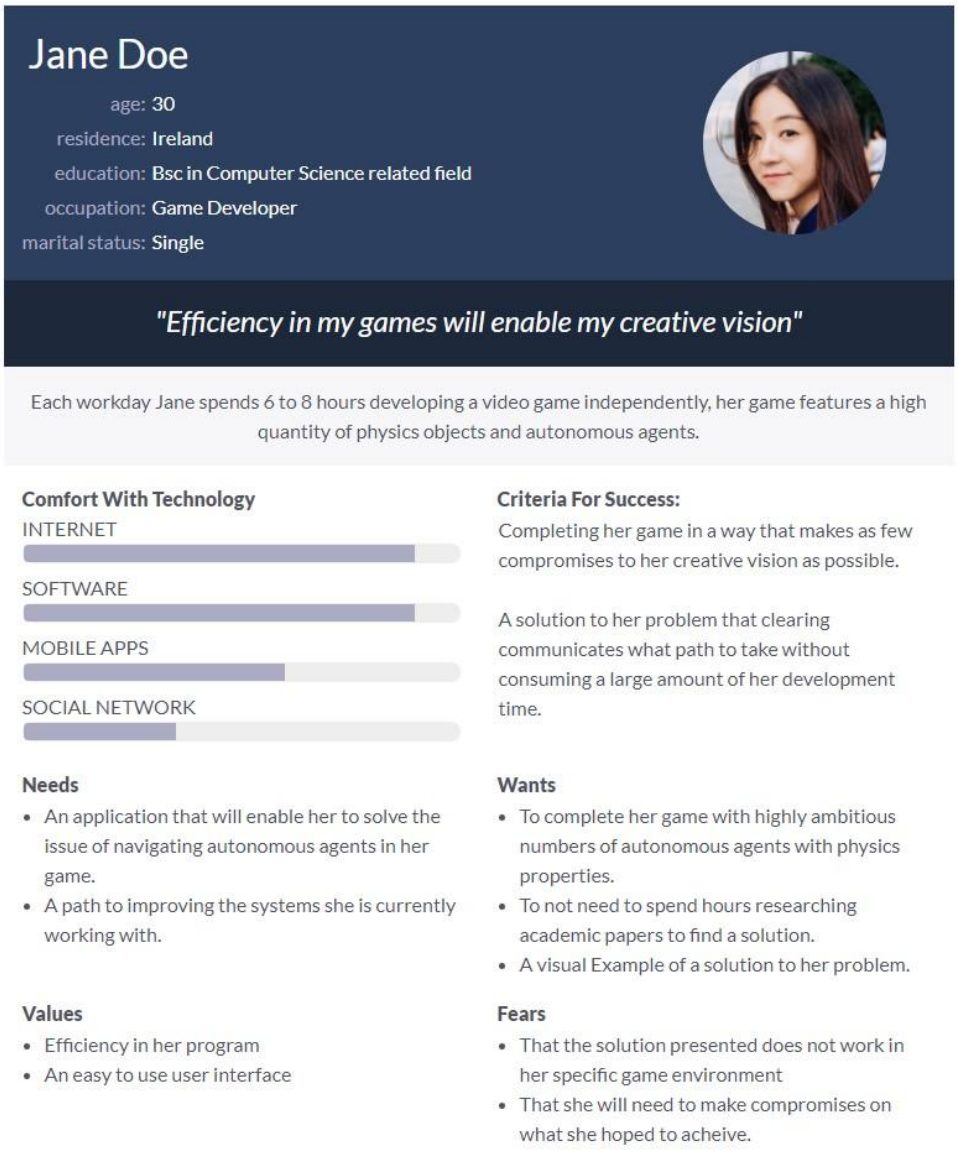


Figure.13 Jane Doe Persona - Developer

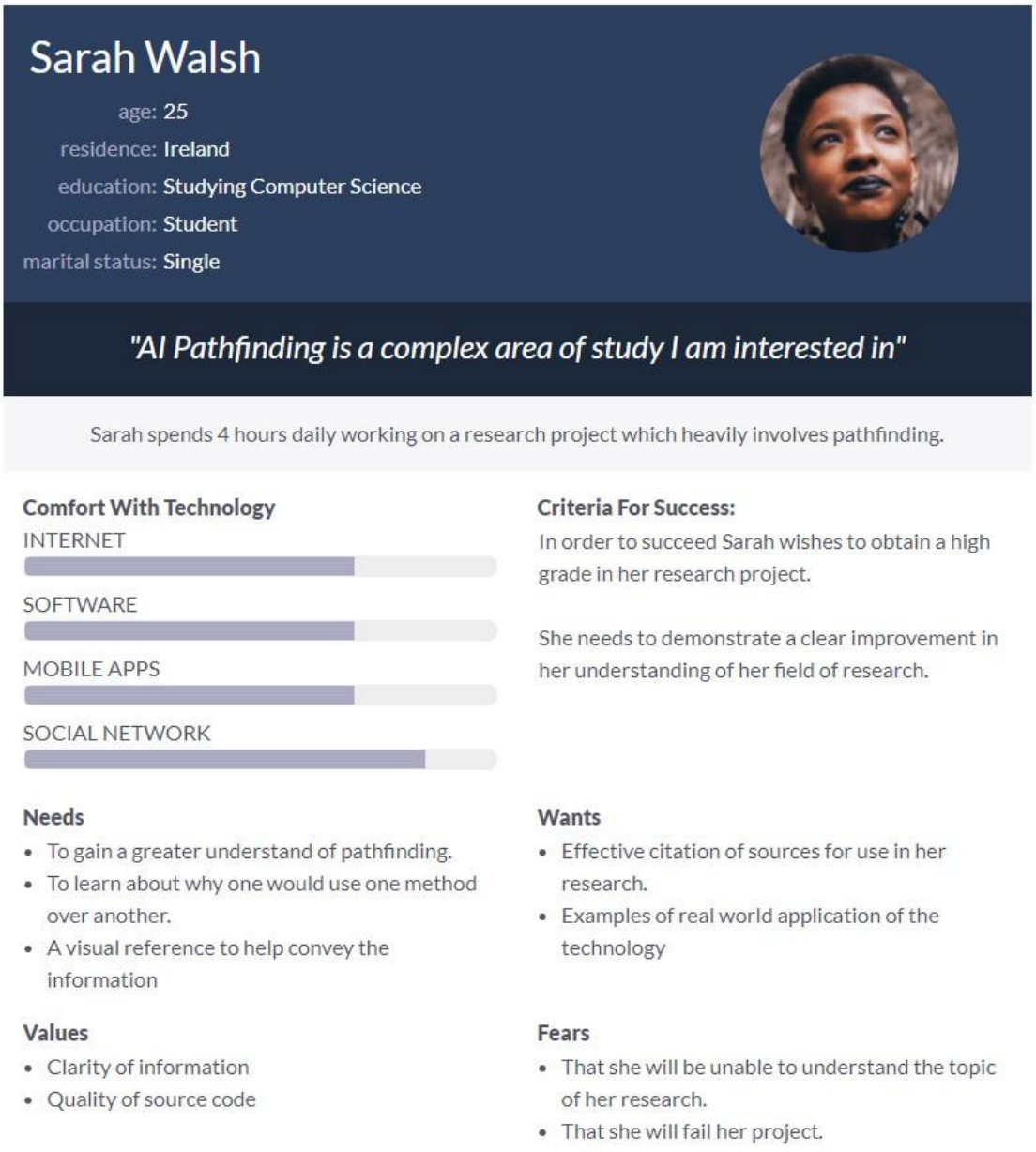


Figure.14 Sarah Walsh Persona - Student

3.2 Requirement Modelling

3.2.1 Functional Requirements

The System must:

- Display clear user interface elements
- Train the user to use the program
- Receive data from a data source
- Visually communicate through use of shape and colour the motion of the vehicles

3.2.2 Non-Functional Requirements

The System must be:

- Efficient and effective in its execution of code multiple object interaction
- Highly useable and responsive to user input
- Able to support a variety of user experience levels meaningfully

Use Case 1 : Training

Primary Actor: User (Aviation Hobbyist)

Success Scenario:

1. Actor open application
2. Actor Selects training program
3. Actor completes program and proceeds to data driven model
4. System records progress through scoring mechanics
5. Actor plays until failure
6. System displays performance

Extensions:

2.a Actor plays through each of the three training scenarios

5.a System scales difficulty of game until failure

Use Case 2 : Only Game

Primary Actor: User (Game Enthuthiast)

Success Scenario:

1. Actor open application
2. Actor omits completion of training and heads straight to full game scenario
3. Actor likely fails sooner than expected due to not being proficient
4. Actor retries until satisfied

Extensions:

4.a Alternatively actor quits due to dissatisfaction

It is evident from this example that the second user profile, the game enthusiast is less likely to have their needs met by the application if they decide not to engage in the training program. This may be addressed by making the user interface as intuitive as possible.

3.3 System Model and System Requirements

The systems involved in the construction of the application are broken down into the following:

- The Simulation system
    - Game Movement and Behaviors
    - Path Following and Game Logic
- The Data system
    - Pulls Data from user json
    - Informs object positions
- The UI System
    - Receives and interprets user input
    - Displays changes on screen

These systems will work in conjunction with one another to deliver the full application. A diagram of this workflow as a user interacts with it can be seen in Fig 3.0.

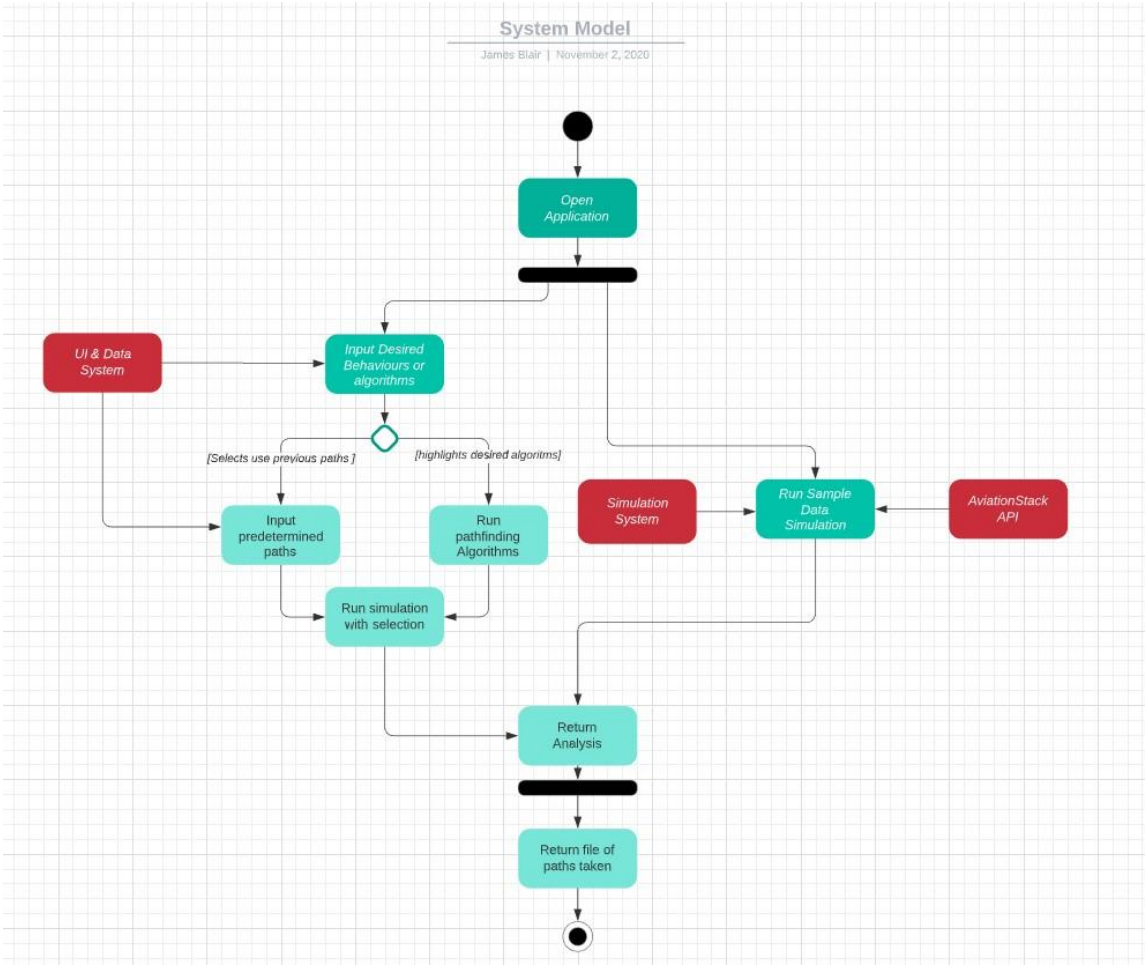


Figure.15 Sample System Model

3.4 Feasibility Study

In order to establish the risks associated with implementing the proposed project a feasibility study was carried out. This allowed technical and project management issues to be identified and the solutions to these obstacles to be outlined prior to their arising.

Sequential Project Completion Events:

1. Selection of technologies
2. Implementation of simulation
3. Implementation of User Interface Controls
4. Implementation of position data system

3.4.1 Selection of Technologies

This stage involves the assessment of benefits and downsides to the available technologies with which the project could be implemented.

Potential Challenges:

- The software may have limitations as of yet unidentified until later on in the development cycle.
- The software may have compatibility issues with target platforms or other technologies the project intends to use in conjunction.

Potential Solutions:

- Predetermine the core functions of the minimum viable product and ensure that selected technologies have the capability to deliver these.
- Test the intended functionality on a small scale or find examples of successful existing projects which use both technologies.

In the case of this project the selection of technologies for a javascript library has been an important factor to consider. P5.js and D3.js present similar solutions to simulating visual data with the latter putting a stronger emphasis on data and the former on design. Alternatively game engines may provide a more complete and robust set of tools with the drawback of a potential reduction of control in certain areas.

3.4.2 Implementation of Simulation

This stage involves the implementation of the main body of the functionality of the project. This being the movements of planes in a flight path network.

Potential Challenges:

- Development of class and object structure needs to be highly robust as it will require all objects to have access to all other objects at all times.
- Solutions need to be highly efficient as multiple calls may be made for many objects each frame.

Potential Solutions:

- Explore viability of solutions to this such as a hierarchy and script execution order.
- Move as much code as possible to single scripts not being called per object.

The simulation component of the application is the most significant as regards functionality and as such, will require the best solutions to be selected for issues that may arise.

3.4.3 Implementation of User Interface Controls

This stage describes the user input handling and processing as well as the layout and functionality regarding user interface elements.

Potential Challenges:

- User interface requires large number of different items reducing comprehension
- Inputs are registered slowly due to underperforming code

Potential Solutions:

- Explore options for grouping or combining UI elements in order to keep the screen clear of clutter.
- Create exceptions for user inputs which are processed with priority.

The user interface represents the most relevant component of the project for comprehension by the user and as such should be at the forefront of consideration when implementing any feature throughout development.

3.3.4 Implementation of Data System

This stage refers to the acquisition and implementation of a data resource to inform locations in the simulation environment.

Potential Challenges:

- Format of selected data requires translation to usable state.
- Environment scale and size needs to be mapped to match latitude and longitude data

Potential Solutions:

- Find resources for converting data to the desired format and confirm compatibility
- Investigate projection mapping options such as mercator projections

The .json file type is the likely desired format for the final data although it is unlikely such a resource in this format already exists and as such it is likely that a process will need to be undertaken in order to address this.

3.5 Project Plan

3.5.1 Research and analysis

In order to assess the topics for the previously included research component of the project, a list of short form topics were researched to determine the likeliest candidates for full research and analysis. For each of the sections outlined above the following topics were researched:

Technology research:	Pathfinding research:
● P5.js ● D3.js ● jquery ● React ● Unity	● BFS ● DFS ● Djikstra ● A* ● Multi-Agent Pathfinding ○ M* ○ CBS ● Game Implementations ○ Standard grid

	○ Visibility Graph ○ Navigation Mesh ○ Path smoothing using string pulling and funnel algorithm

Following and Steering Behaviour research:	**Javascript Web Application Research:**
● Autonomous Agents ○ Rule trees ○ AI ● Methods of Navigation ● Methods of Avoidance ○ Speed adjustment vs Path adjustment ● Simulating real world factors such as ATC rules ○ ATFM - Flow ○ ATC - Separation	● Canvas Limitations ● Physics handling

The preliminary research that enabled the project conception involved determining sources of information for the above areas of study and confirming the viability of utilising that information.

3.5.2 Outline Design

The application will initiate with an option to select if the user would like to run the training program or a mode called endless. If they select the training program they will be taken through three consecutive scenes each describing the fundamentals of how to use

the application. The endless mode will be a program which simulates air traffic using data for Dublin airport's arrival pattern.

Training stage one will address the holding and landing component of the controls, with users simply needing to organize the order the planes land in.

Training stage two is the steering and navigation component with users being required to move the camera and give planes directions.

Training stage three explains the scoring system with users needing to adjust speed and altitude to gain points.

The endless mode combines all of these things in a real world scenario and will continue indefinitely until the user fails due to a plane colliding with another. The application will increase the rate that planes appear to be landed throughout to scale the difficulty of the simulation. Once completed this will then present a score to the user based on their performance.

3.6 Test Plan

In order to facilitate the mitigation of errors while progressing from each project stage to the next, a test plan has been created and will be followed throughout the development process. The test plan follows the standardized method as outlined in Fig 16.

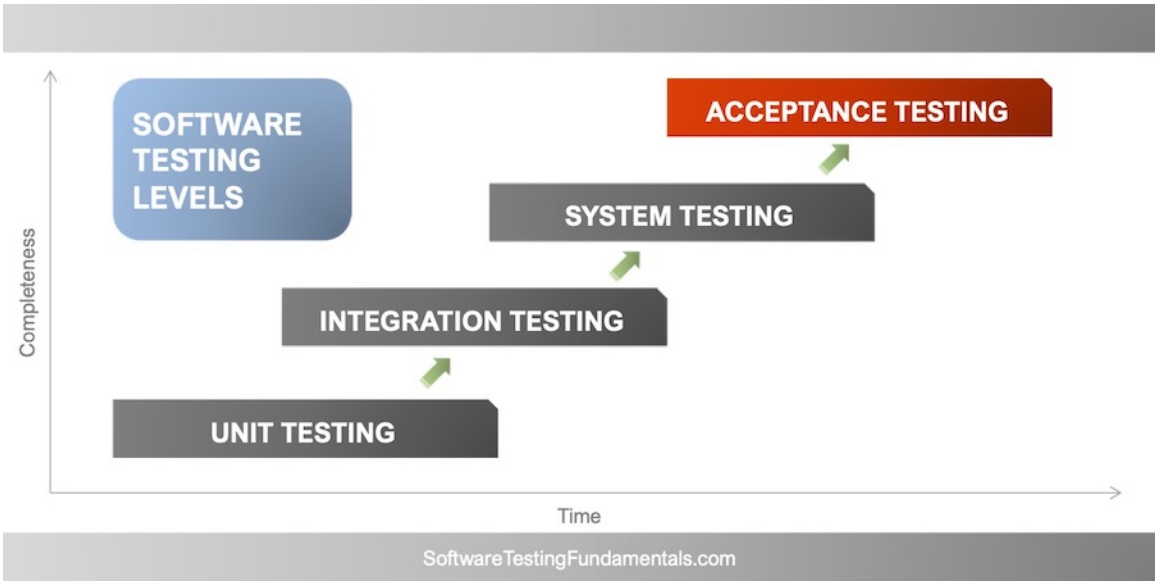


Figure.16 Test Plan Model (Source: *Software Testing Levels*, 2011)

Testing will begin with unit testing which concerns the smallest testable components of the project. This may refer to specific objects or javascript functions which will likely be the smallest unit of software included in the project. The unit testing will be executed by implementing the white box testing method which is done by supply the test unit with valid and invalid inputs and observing if the outputs as anticipated.

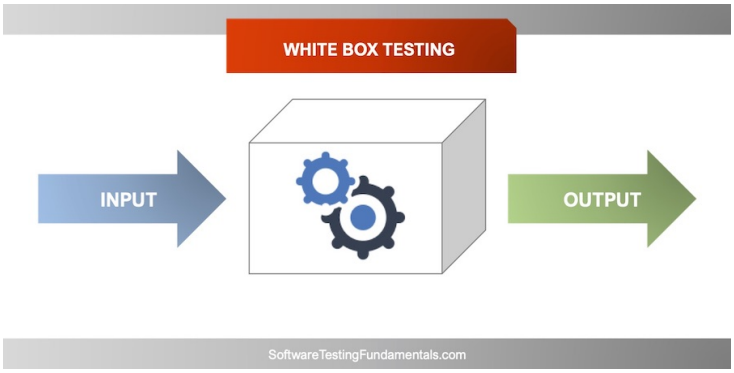


Figure.17 White Box Testing (Source: *Software Testing Levels*, 2011)

Following unit testing, integration testing will be performed. This is when the tested functional units will be executed in combination to determine if the desired outcome is being produced under more elaborate conditions. The method for testing integration will be grey box testing which is when the internal structure of the test unit is partially known. In this project integration testing will be performed on entire scripts to determine that the internal functions are executing successfully.

System testing is then initiated to determine the integrated system meets the requirements specified. This is carried out manually by performing black box testing, in which the internal structure of the integrated system is unknown.

4. Design

4.1 Design Introduction

The application for this project is a ludic simulation for air traffic control. In order to demonstrate the processes involved in the design of this system, this document will outline the core elements of that design. These design elements are broken down into program design, being the design of the system and functionality of the application, game design, which entails the process of designing gamified elements and training systems and finally user interface design which explains the process of designing the interactable and visual elements present in the project the user is expected to interact with. As each design phase relies on a comprehension of the technologies present in the application and as such this document will first outline the general workflow and pertinent features of the unity game engine.

4.2 Technologies

The Unity game engine is the primary focus of the technologies present in this project and as such this chapter will outline the structure and workflow of this development environment and its tools as they relate to the project. The IDE used in conjunction with this engine is Microsoft Visual Studio Code.

The project also relies on the use of an online repository for version control. To this end Github is being utilized to facilitate reliable access to older versions of the software in the event of technical issues occuring.

Finally the website ‘trello’ allows the project to succinctly manage task status and organise the workflow that should be undertaken by establishing a system of task priority and time management.

4.2.1 Scene Structure

A scene in unity is a specific configuration of asset positions and properties in a space. Scenes can be loaded and unloaded during runtime to enable access to widely varying situations for users. Within a scene will be a hierarchy of the objects the developer has placed or instantiated from their assets folder.

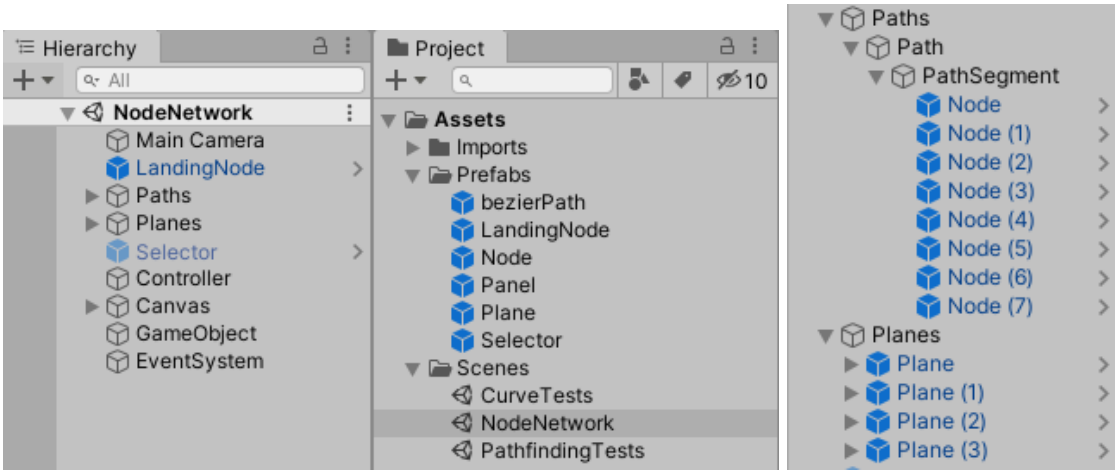


Figure.18 Unity folder structure

Each object in the hierarchy has been placed into the scene environment while the project folders contain the available assets the application may access during runtime via code. The blue objects are instances of prefabs which are comparable to objects of a class while the grey boxes denote objects that are unique single instances. In this structure it can be observed that the 'Paths' and 'Planes' GameObjects have been used as containers to hold multiple other objects so that they may be accessed more effectively using code.

Another common practice in the design of unity scenes present here is the 'Canvas' which contains all of the necessary UI elements in a scene and a 'Controller' object which holds a script which acts as a mid-point to other scripts wishing to access variables in foreign objects and a camera which determines the viewpoint the user will see when loading the application.

This structure is common to most Unity projects and enables a consistent reliable access to the information as well as a fast and effective means of testing functionality in a variety of possible situations.

4.2.2 Object Structure

An object in unity refers to an instance of a unity class called a GameObject and upon initial creation will contain just one component being a 'Transform' which determines its position in three dimensional space. A game object may be given a tag so that it can be

more easily located via code without needing to find it by its name and may also be given a layer to determine which group of objects it is permitted to interact with.

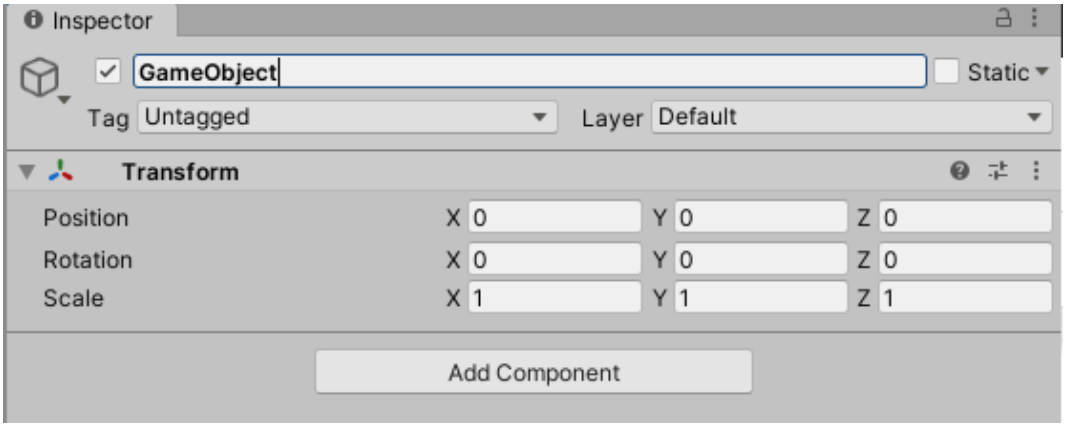


Figure.19 Unity GameObject Inspector with Transform Component

A GameObject may have any number of Unity's components attached to it. There are a large number of components which serve many purposes such as a 'rigidbody' component which allows physics forces such as gravity to affect the object or a 'collider' component which allows collisions with other objects to be tracked. These components can also be scripts written by the developer which may be configured to enable any desired form of interaction.

4.2.3 Engine Architecture

The unity engine compiles an application in a predetermined sequence which can only be modified by the user at the highest level. The user defined components and scripts can be sequenced by the user in any desired order however the structure of scene to GameObject to components is essential.

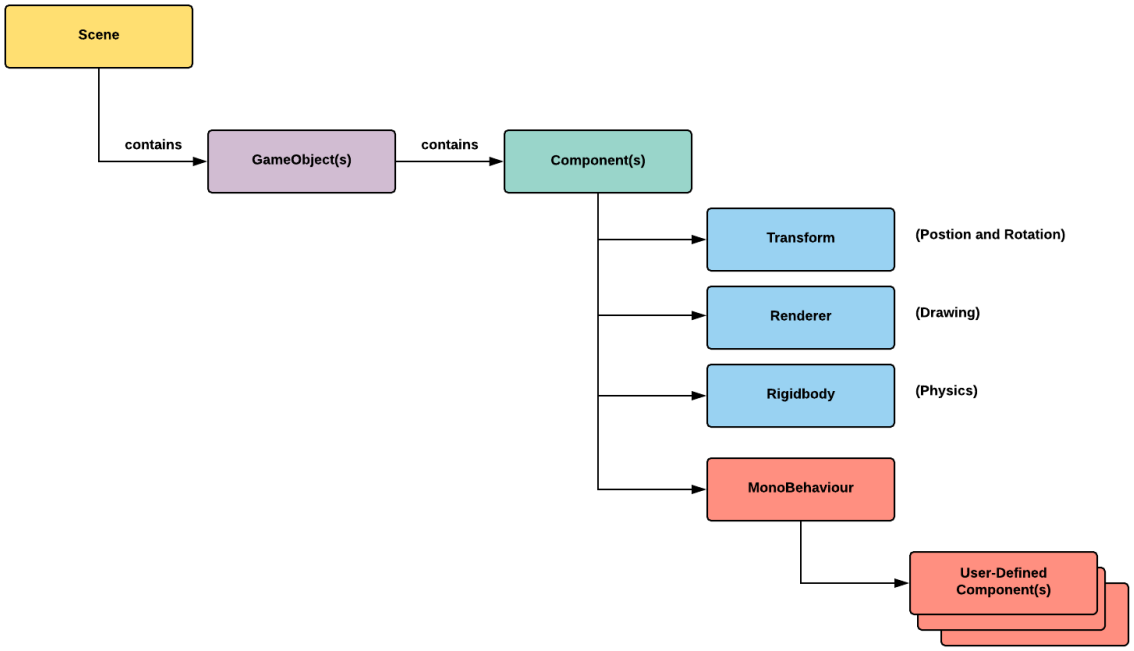


Figure.20 Unity Execution Priority (Source: Unity Technologies, n.d.)

In the below example, I have determined that the bezier script must execute 100ms before the path script on each frame. This is necessary because the path script depends on certain variables being established/updated in the bezier script. The other present items in execution order regard the UI system which performs more effectively when initialized before runtime.

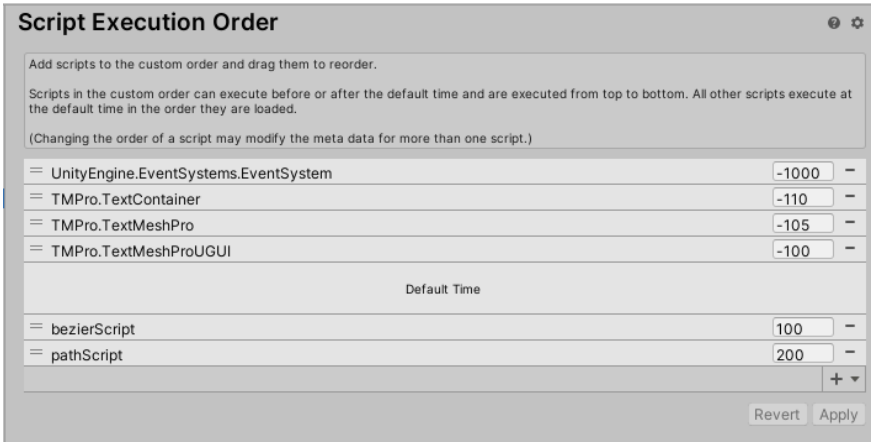


Figure.21 Unity Execution Order Editor

Making use of all of the above systems the application is able to be designed to make use of a structure and sequence for every component of the application with complete control.

4.3 Preliminary Application Design

In order to develop an initial design structure for the application system several diagrams were established to create a basic idea of the components that would interact in order to accomplish the application's goals.

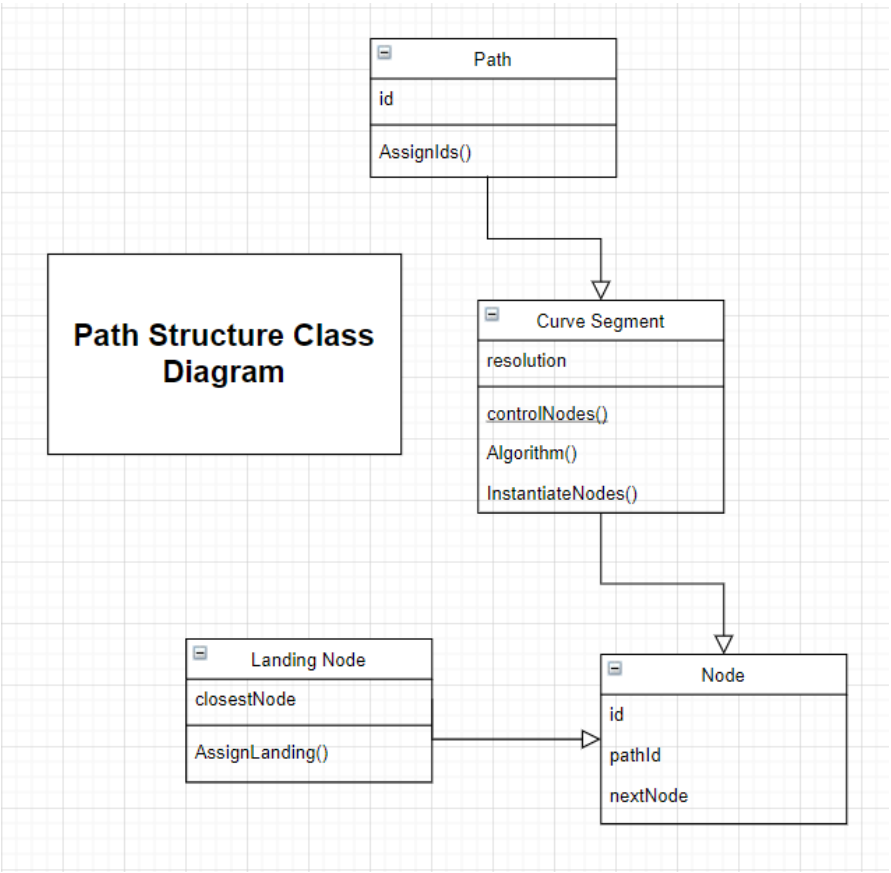


Figure.22 Path Structure Class Diagram

The first component designed to lay the foundation was the system of nodes and paths which would allow the agents to navigate logically within the environment. These nodes will make use of unity's scene hierarchy in order to lay out the sequence in which they should be navigated. Following this it was necessary to verify the integrity of some of the key formulae that would power the more complex components of the system's operation.

In the following example an early iteration of the DeCastell algorithm was established in pseudocode.

```
loop
   input start and end point of curve
        vector = start point of curve * resolution
        insert new node at current vector
        increment resolution to find next data piont
                determine if at final point
                        break if true
end loop
```

Figure.23 Early example of pseudocode for curve algorithm

During the development of this algorithm and it's conversion to C# it was important to understand which aspects of this code would need to be interacted with beyond the initial generation of the curve. Having control over the positions of control points in order to create primary nodes which would handle arrival time logic as well as curve resolution so as not to overburden the application with too many nodes were key considerations in the design.

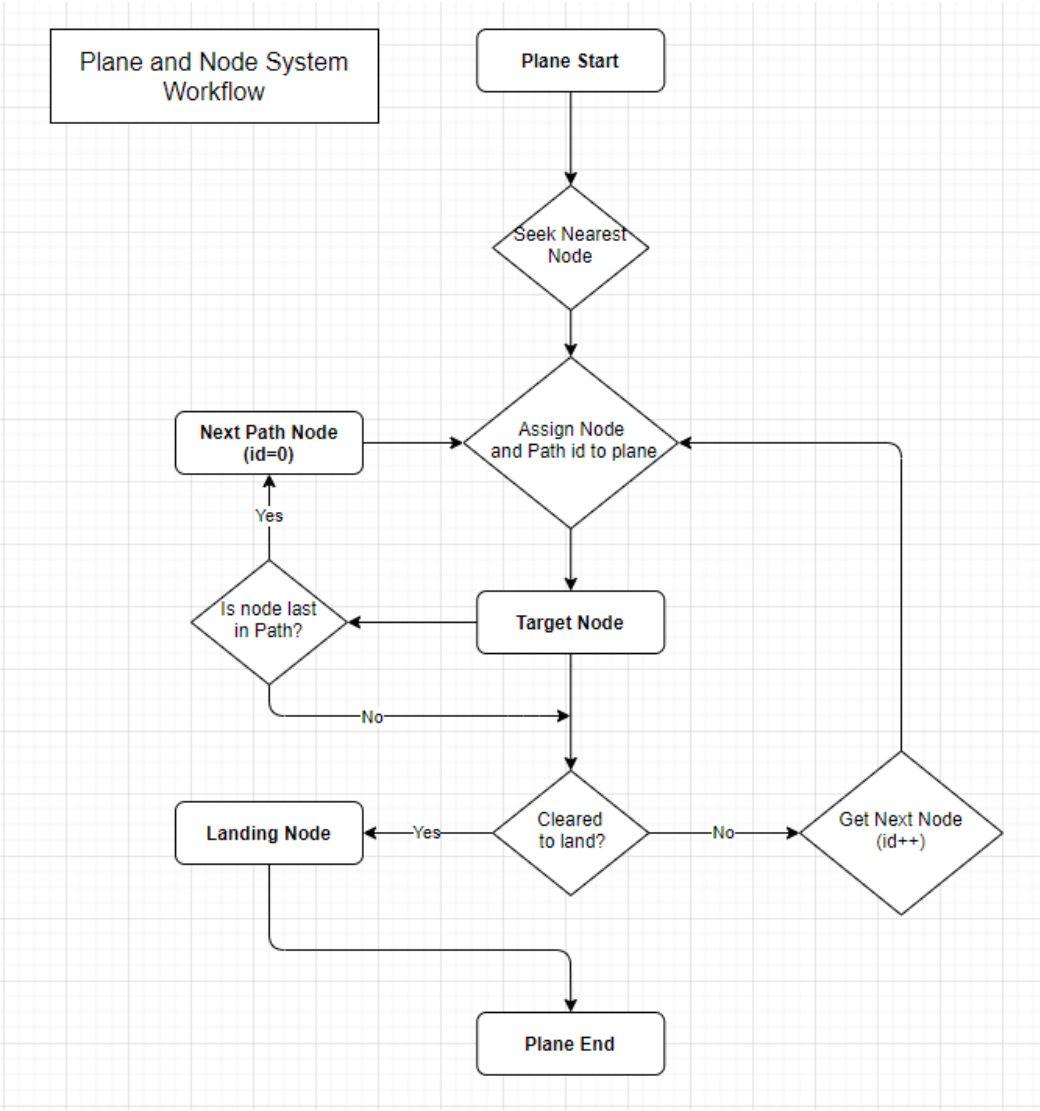


Figure.24 Plane and Node System Flow Chart

The above flowchart outlines the basic structure of behaviours for objects of the two primary classes: nodes and planes. The flowchart demonstrates how they will make use of the scene hierarchy to assign a sequence of nodes which are parented to path segments which themselves are parented to paths. This design allows planes to navigate through the variety of non grid-structured nodes in the correct order without over reliance on pathfinding calculations. This sequence of logic will be executed independently by each plane in the scene simultaneously. The reasoning for a system whereby each plane is able to, without reliance on information from a central system, decide where it should go next

is that it is an important foundation for developing more complex decision making processes for these agents.

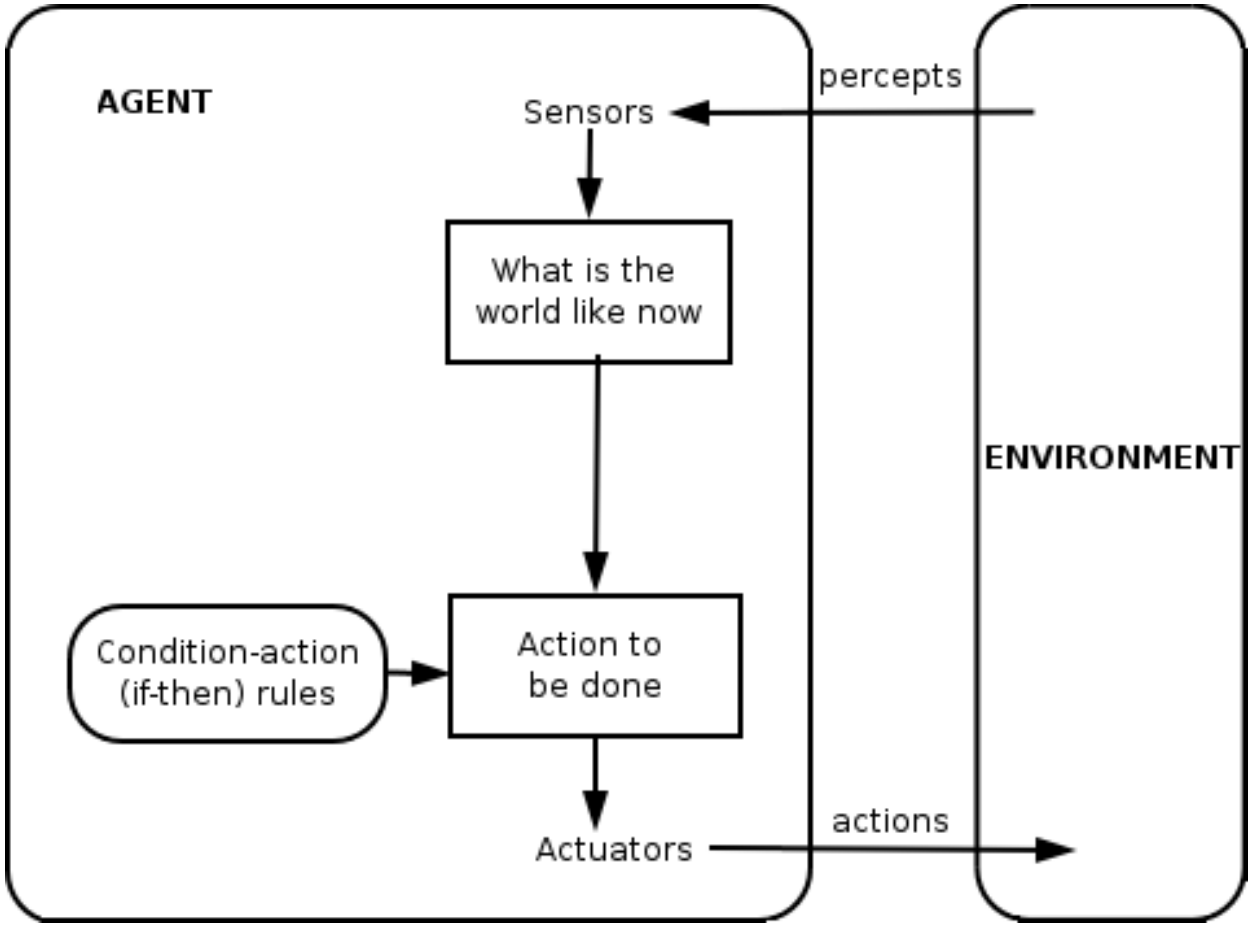


Figure.25 Generic Intelligent Agent Behaviour Flowchart (Source: Russell & Norvig, 2010)

The plane's structure of operation relies on inputs from the environment at each decision making stage. This structure is central to the concept of an intelligent agent and allows the object to perceive factors which lead to a variety of action taking results as outlined in the above.

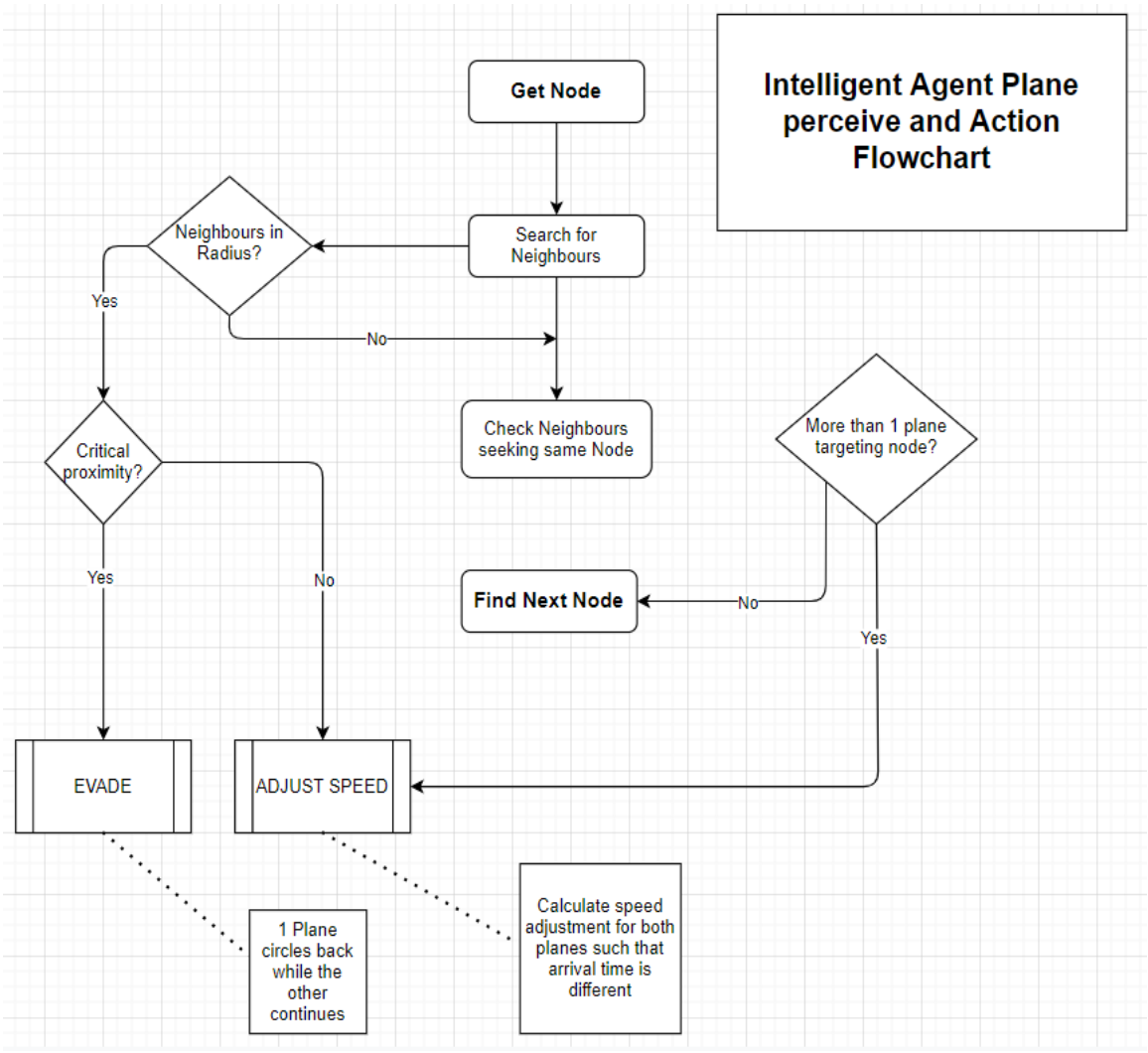


Figure.26 Plane Agent Behavior Flowchart

From this next flowchart it can be observed that the plane will act to perceive specific environmental factors then choose from an array of actions. For reference a ‘neighbour’ in this case refers to other planes than this specific agent. This example includes an evade action which would select one of two vehicles in a conflict and alter its course. The adjusted speed action would slow or speed up one or both vehicles to prevent an evasion action needing to be taken. As these features are implemented upon it may be necessary to add additional actions to this system however the structure relying on perception and action and the systems which enable this are firmly established here.

4.4 Feature Development Plan

Initially the design for this project was sequenced to include the core elements which would enable a testing environment for the development process. These core objects are as follows:

- Aircraft
    - Moves forward towards assigned target
    - Components:
        - Rigidbody
        - Script
        - Collider
        - Renderer
- Node
    - Has an ID which denotes its sequence in a series of nodes
    - Draws lines between neighbours in sequence
    - Components:
        - Script
        - Collider
        - Renderer
        - Line Renderer
- Landing Node
    - System by which planes and nodes can be flagged to change behaviour if approved to land
- Curves and Paths
    - Adjustable curve which creates a series of nodes along its body
    - Path object which connects these sections of nodes as desired

Each of these elements will be stored as a prefab within the unity engine structure and instantiated into scenes using code. The majority of functionality the application will depend on is structured around the relationships between these objects and as such it has been determined essential that this element is fully functional prior to development on subsequent features. These objects and the systems which enable their functionality is the first phase of development.

The next set of features include the remaining core functionality for a minimum viable product which meets the acceptable goals the project intends to meet.

These are as follows:

- Data Integration
    - Pulls longitude and latitude data for flight paths from a .json resource
    - Converts these values to a relative transform in the environment
    - Populates the positions with paths
- Advanced Aircraft Behaviour
    - Evasion of other planes through system of desired vs actual
    - Adjustment of arrival time at nodes via node logic
- User Interface Implementation

By clearly defining the priority of tasks and breaking them down further into subtasks the project aims to be succinct in defining the scope of features that are achievable within the timeframe.

4.4.1 Network Design

The networking component of the project makes use of a file system to return relevant information in the '.JSON' file format to allow for effective implementation in the game engine. In order to interpret .JSON files in the appropriate format the application will be utilising a plugin called simpleJSON (*SimpleJSON - Unify Community Wiki*, n.d.) which allows for effective parsing and building of JSON files.

Using this framework the project will create serialized objects from the data and enable the design of various features of the application to be informed by tangible data.

4.4.2 Modular Design

In order to facilitate the training system not requiring bespoke code, it is essential that each feature added to the functionality of the application be entirely modular. This will mean that the feature can be enabled or automated based on the particular scenario or user settings.

An example of this type of design could be observed in the intended implementation of a difficulty setting. Depending on this state the method governing whether the agents will automatically adjust their speed to avoid collision or simply ignore this functionality would need to be accounted for in the design.

Rather than registering these states in simple boolean variables, however, it will be more efficient to have a central data center in the form of an interface containing all of the player settings and for each feature to have access to this data to determine how to execute their functionality. This game data will also be saved locally using a generic file path which Unity can access on subsequent launches of the game to recall previous user settings.

The challenge with this design structure however will be in maintaining a non-reliance on specific functionality which may or may not be currently active for static features. For example the user interface will need to adjust which interactable elements are displayed based on which states are active and may also be required to hide default elements which have no relevance such as a ‘land plane’ button if the planes are currently being manually adjusted and landed.

4.4.3 Gameplay and Mechanics Design

This game intends to strike a balance between actual simulation and a video game with varying challenges. As a result the mechanics need to engage players more actively than typical ATC simulation and dually serve the purpose of training the user gradually. In order to outline the design process this implies a list of the actions the user can and should perform has been outlined:

- Plane Selection
    - Select a plane to see a UI hover over it

    - Select between adjust speed or Change Path
- Path Selection
    - After selecting change path a visualisation of all available paths will stem from that plane
    - Hovering over a selected path will highlight it
    - Once selected a the plane will then attempt to change to that path
- Vehicle Class
    - Different planes will adjust to follow paths at varying speeds
    - Identifying the vehicle type and understanding how maneuverable it is will be essential to success
- Landing
    - A plane's probability to land successfully will be larger dictated by how long the plane was able to descend in a straight line before landing
    - This will be communicated by an interpolation of color from red to green.
    - An airport will also need time to clear the landing zone for subsequent landings
- Circling
    - To increase odds of success in landing players will need to queue planes to circle an airport to provide enough time for others to land.
- Scoring
    - A player will be awarded points for 2 Factors of their gameplay
    - Speed and Altitude limits while approaching destination
    - Risk is assessed by proximity to other planes during flight

Prior to implementation and testing the most logical approach to designing the game elements of the application is to start with only what is necessary for the player initially and subsequently introduce new elements. It is proposed that in the first stage the player will only be required to tell a plane when to land. Following landing responsibilities would be introduced per stage in the following sequence:

1. Circling and Landing
2. Plane and Path Selection

3. New Vehicle Classes
4. Scoring System
5. Additional Hazards

The scoring system introduction at this stage suggests that prior to this the player is not penalized for being non-punctual or risk averse prior to this. This design choice will likely be subject to adjustment based on the outcome of user experience testing. Which elements are effective at improving the user experience and added to the application will depend very heavily on the results of this testing. The additional hazards mentioned imply that once the application is complete additional features such as weather warnings or fuel shortages may be introduced in order to increase pressure on the player and add variety to the experience if the current model is overly monotonous.

4.5 User Interface Design

The approach taken to user interface design for the purposes of this application echoes the rest of the design. It is necessary that in order to fulfil the purpose of simulation that the look and feel is similar enough to actual air traffic control interfaces however it is also necessary that it be readable by an entry level user.

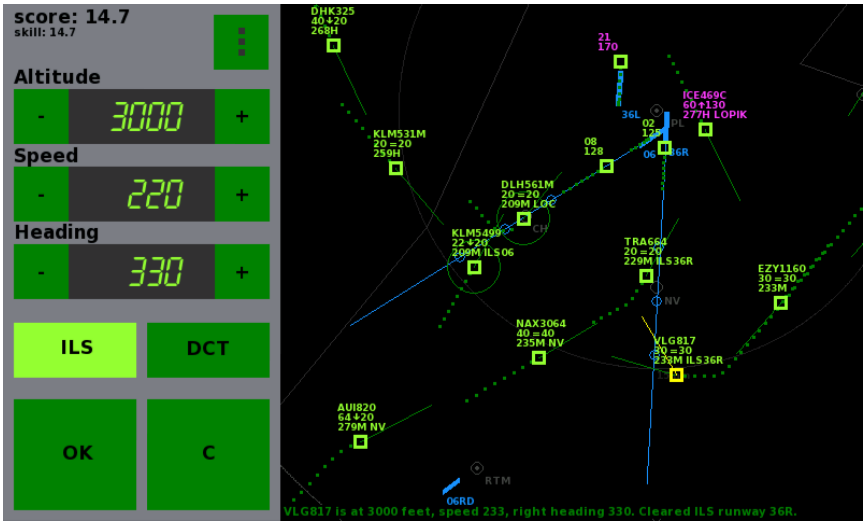


Figure.27 A Mobile Game Interpretation - *(Source: statgrid, 2015)*

From Fig.27 it can be observed a mobile game simplification of the kinds of systems used in reality. The user interface in this example is fairly simple and intuitive while extensive training is typically required in order to interpret systems used in an official capacity.

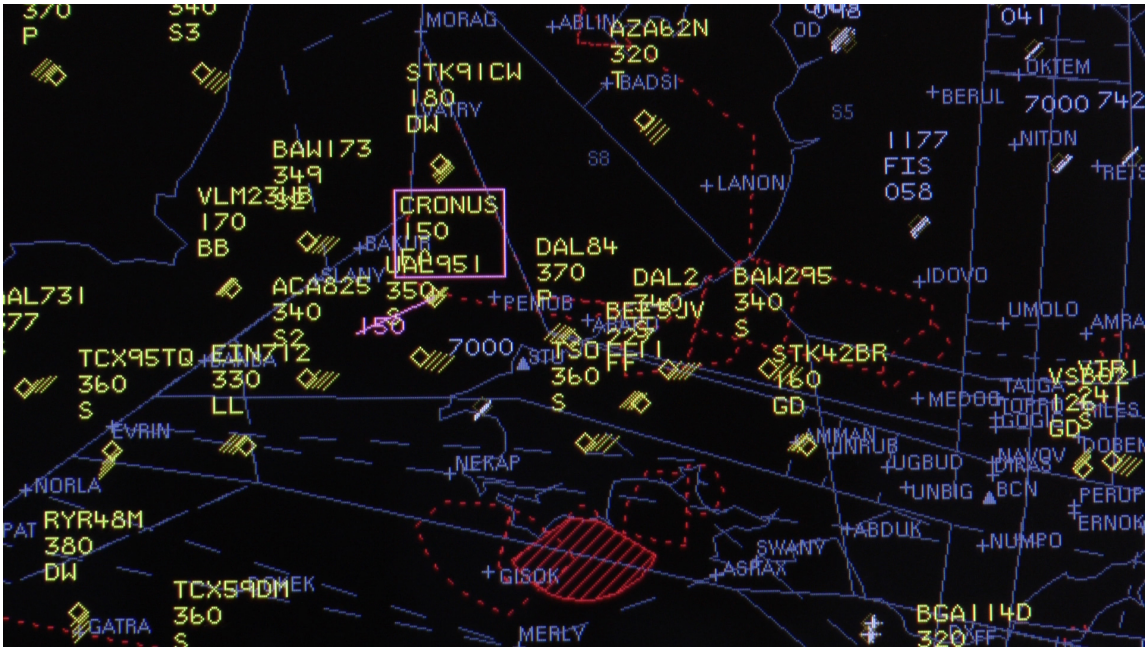


Figure.28 Professional ATC Software - Watchkeeper Unmanned Aircraft System *(UAS)* (Source: *EDR, 2015*)

The proposed design for the main module of this project's application will make use of systems implemented in video games such as minimaps, panning and zooming as well as screen edge indicators in order to bolster a UI similar to that which was demonstrated in Endless ATC for mobile. The colour scheme and typeface will also maintain the look of these examples with a large priority placed on visual clarity above aesthetic appeal.

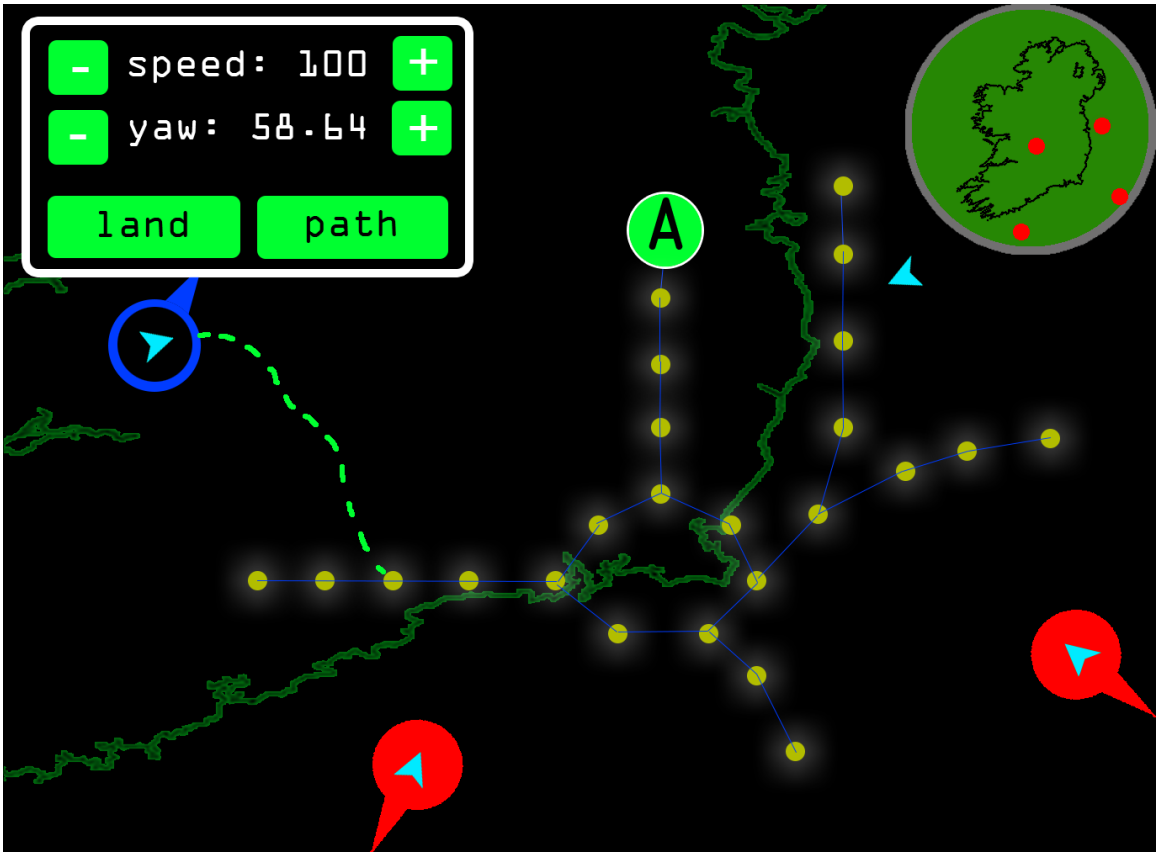
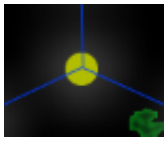


Figure.29 Early Mockup of Design Language for UI

				
Airport	**Selected Plane**	**Edge Indicator**	**Minimap**	**Node**
User's objective	Displays this plane's controls	Indicates off screen plane	Displays whole environment	Path joint

From the above user interface suggestion one of the key differences is the use of symbol language to communicate rather than text where possible. It is essential to understand that in this iteration a player can use the mouse or arrow keys to navigate freely the area their viewport examines or even zoom in or out as appropriate. The colours used in this example are to differentiate elements and are not necessarily representative of the final

scheme which will likely lend heavily from the green and black color scheme heavily associated with these kinds of applications.

The font chosen for this application was heavily influenced by in-depth research performed by Helena Reed on the subject matter specifically regarding fonts chosen for air traffic control applications.. (Reed, 2017) This research presents ‘verdana’ as a viable choice for its comprehension and visual clarity.

In order to facilitate the likely high quantity of controls and on screen information it is proposed the user HUD (‘heads up display’) be broken down into a series of panels that may be toggled in and out of view of the screen. These sliding panels will allow the user to have a clearer view of the detailed environment without being impeded by the user interface elements that are not currently pertinent to them. By offering the user agency over the contents of the display it is believed that the user will not allow themselves to be overburdened by visual noise.

The three proposed panels are as outlined below.

Info Panel

This panel, which is the smallest, will contain non-interactable information and is intended to merely inform the user of elapsed time and progress. By conveying the score the user may gain a better understanding of when they have done something of merit however if the user would rather close this panel to attain greater clarity the option will exist.

Control Panel

This panel, which will display once a plane has been selected, will display all of the relevant information for that plane such as its callsign, speed and altitude. Below this display will be buttons and handled sliders which allow users to adjust the plane's trajectory by any means.

List Panel

This panel displays a list of all inbound aircraft which need to be landed. As planes arrive or are landed the list will dynamically adjust its length. Each of the listed planes may be interacted with in order to smoothly navigate the camera to center on the plane.

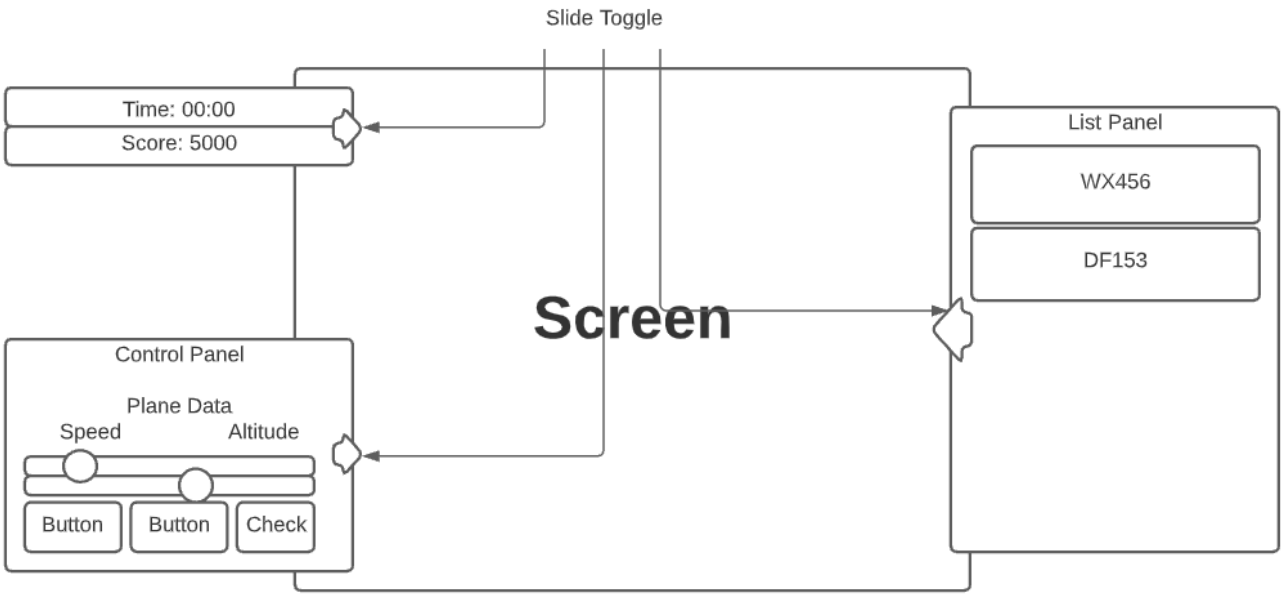


Figure.30 User Interface Layout with Sliding Panels

5. Implementation

This chapter will discuss the method undertaken in the implementation of each component of the system. The application's components are defined in three parts, the data processes which refers to all data preparation and the means by which it is accessed, the object logic which refers to the scripts which define the relationships and interactions between the various objects and the means by which resultant actions are taken, and finally the game logic which governs the systems handling user interaction, user interface navigation and scoring.

5.1 Components

5.1.1 Data Processes

The data required to inform the positions of the node network of Dublin Airport was resourced at the Irish Aviation Authority's website in the form of a .pdf file (see Appendix A, p.110-113). In order to use this data in the application, it was ideal for the data to be arranged in a .json structure so as to enable the nodes to be treated as objects with properties. The first stage in this conversion process was to convert the file to an excel file using simplifyPDF (Simplify PDF, n.d.). The result of this was unfortunately imperfect as the data in the .pdf table was unclear in its structure. Within Excel the data then needed to be reformatted into a form that could be converted to .json using beautifyTools (*Excel To Json Converter - BeautifyTools.com*, n.d.). In order to modify the table structure, excel formulae were written to normalize relevant data.

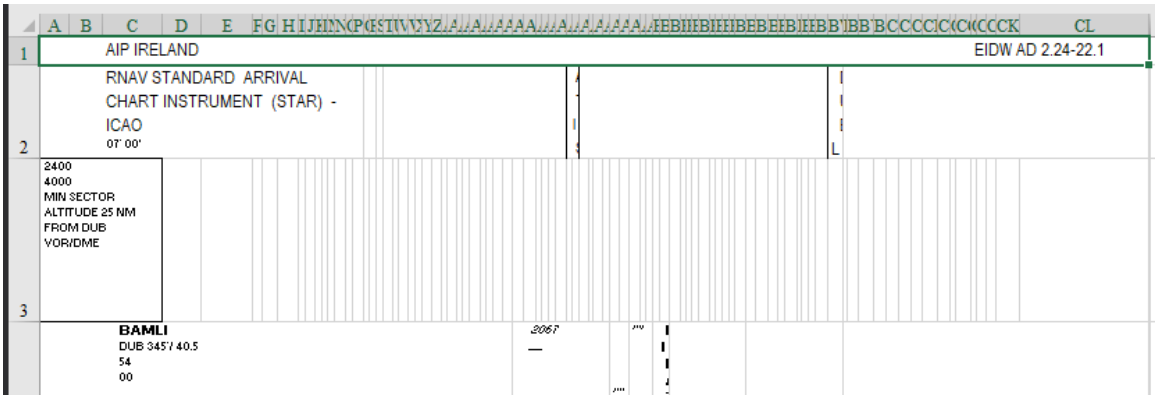


Figure.31 Original PDF to Excel Conversion

	A	B	C	D	E	F	G
1	PathName	NodeName	Latitude	Longitude	SpeedLimit	UpperAlt	LowerAlt
2	ABLIN	ABLIN	52.4658000000	-4.5933000000	0	1800	1500
3	ABLIN	IRKUM	52.5948000000	-5.2239000000	0	0	900
4	ABLIN	LIPGO	53.0350100000	-5.3000000000	240	0	0
5	ABLIN	PEKOK	53.0739300000	-5.3400800000	0	800	800
					230	800	800

Figure.32 Final formatted data

Imported Object

paths


```
{
  "path": [
    {
      "PathName": "ABLIN",
      "NodeName": "ABLIN",
      "Latitude": "52.4658000000",
      "Longitude": "-4.5933000000",
      "SpeedLimit": "0",
      "UpperAlt": "1800",
      "LowerAlt": "1500"
    },
```


Figure.33 Example of resultant object in paths.JSON

The resulting .json file would then need to be prepared for use with the unity engine's c# scripts. The solution to this was implemented in the JSONReader.cs script. The initial step was to use Unity's Serializable system to create a struct which could accept the data and convert to an object many times.

```
[System.Serializable]
  public class Path
  {
      public string PathName;
      public string NodeName;
      public float Latitude;
      public float Longitude;
      public float SpeedLimit;
      public float UpperAlt;
      public float LowerAlt;
  }
```

```
    [System.Serializable]
    public class PathList
    {
        public Path[] path;
    }

    public PathList pList = new PathList();
```

Figure.34 Structure of node object as defined by JSONReader.cs

```
     pList = JsonUtility.FromJson<PathList>(textJSON.text);

        foreach (object o in pList.path)
        {
```

Figure.35 How the objects are populated for each object in the .json file

Once the objects are created with all of the necessary components the next significant task is in the conversion of latitude and longitude data to x and y coordinates. In unity the x and y coordinates for an object in world space are stored within the first two parameters of a Vector3 property located in the object's transform component. In order to account for the curvature of the Earth when moving latitude and longitude data to a flat surface the most typical approach is to make use of a mercator projection. To this end google map services SDK was imported to the project as it contains a convenient function in its library 'LatLngToVector3()' which performs this projection.

```
const latLngToVector3 = (latLng, radius) => {
  const phi = Math.PI * (0.5 - (latLng.lat / 180));
  const theta = Math.PI * (latLng.lng / 180);
  const spherical = THREE.Spherical(radius || latLng.radius || 1, phi,
theta);
  return new THREE.Vector3().setFromSpherical(spherical);
};
```

Figure.36 Three.js approximation of Google Maps SDK LatLngToVector3 function which is inaccessible

The use of this function initially provided the desired results however as development continued under this implementation it became apparent for the use case that the google

SDK protects many parts of it's functionality from view by users. This resulted in a loss of precise control in the development process such as being unable to define the desired size of the space used for projection and instead being given pre-defined map chunks. An alternative simpler method was used for the specific use case of developing the Dublin air space which does not account for the Earth's curvature. A mapping function which allows for the input of upper and lower limits for the provided value and returns a new value based on new limits was implemented.

```
public float Map(float s, float a1, float a2, float b1, float b2)
    {
        return b1 + (s - a1) * (b2 - b1) / (a2 - a1);
    }
```

Figure.37 Mapping function for redefining value from one range to another

This function allows the minimum and maximum longitude and latitude to be mapped to the minimum and maximum x, y coordinates respectively. The result is then provided an artificial buffer to the values in order to adjust any discrepancies.

The size of the air space being projected has an exponential effect on the relative accuracy of the positions being mapped using this method. The smaller the portion of earth being projected the less significant the inaccuracies appear using such a method. For the specific use case of Dublin airport this accuracy is still within a very high margin however in order to scale for larger potential air spaces such as in central Europe a more permanent solution for mercator projection which allows for a larger number of controls may be required.

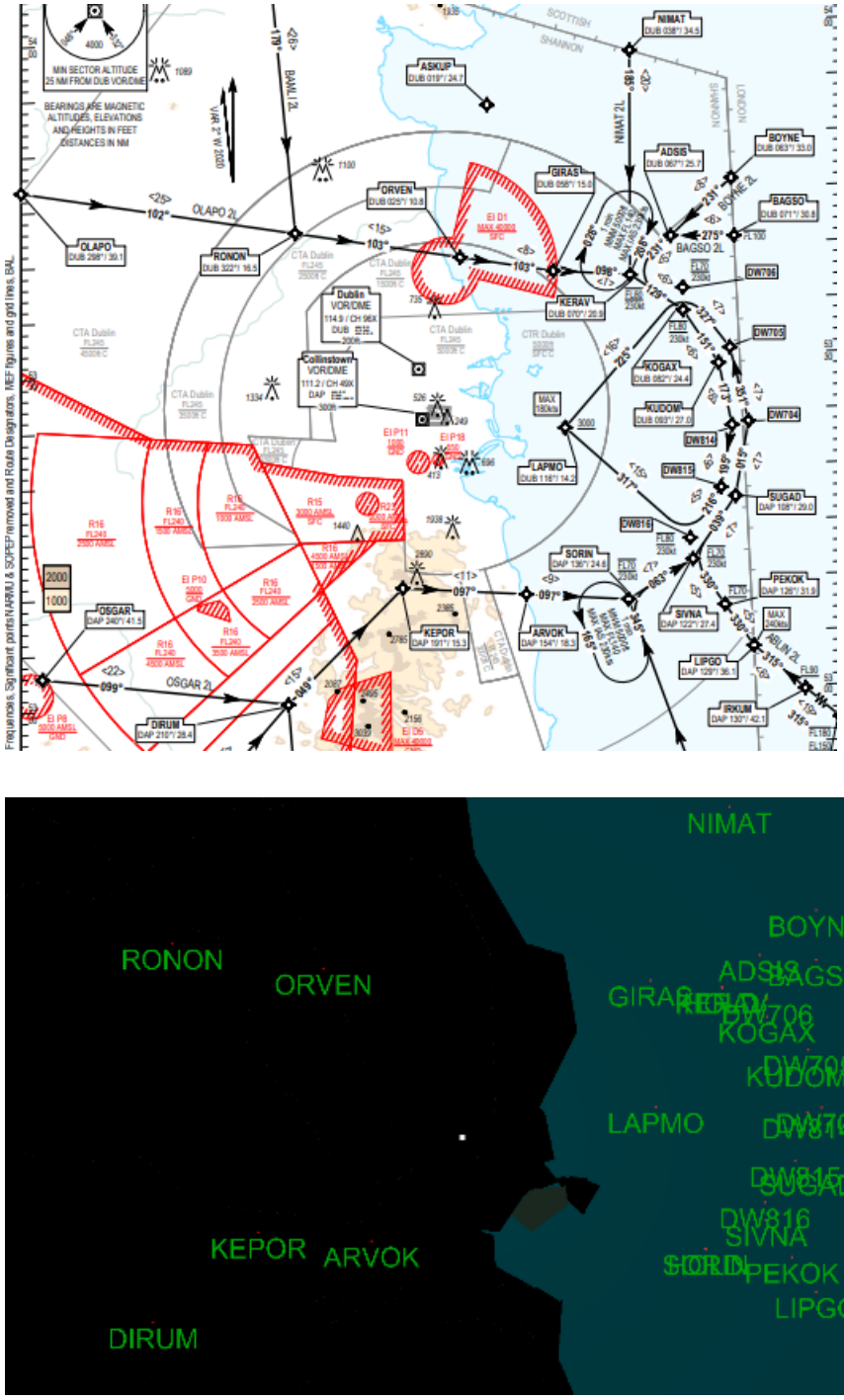


Figure.38 Google Maps SDK projection with space limits and original pdf image above.

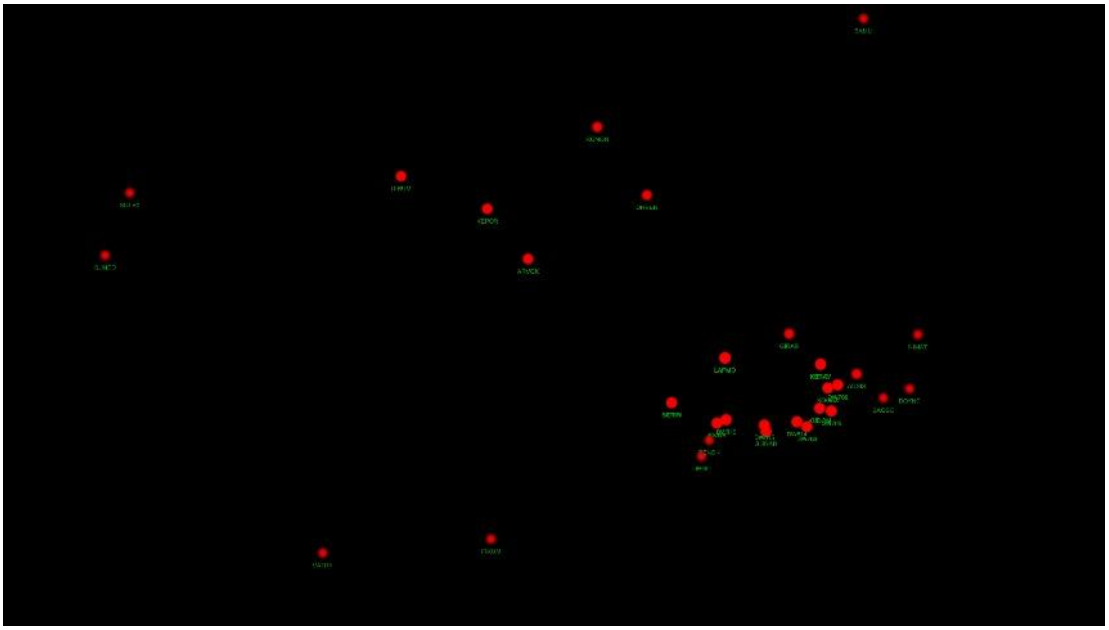


Figure.39 Highly inaccurate projection prior to mapping function using no mapping method

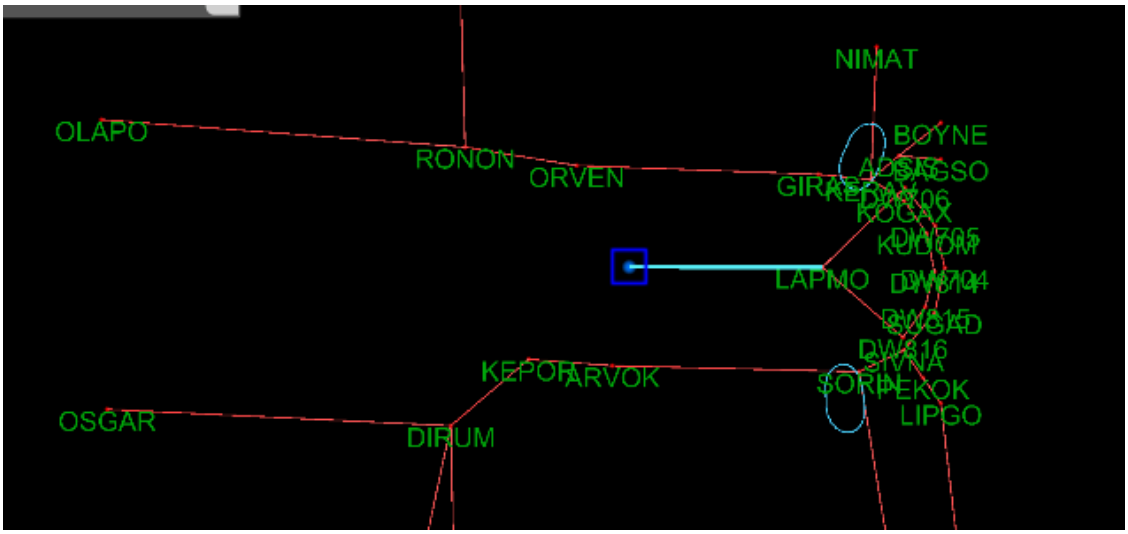


Figure.40 Final accurate projection with controllable world space and objects grouped into paths

Overall the implemented solution was sufficient for the scope of the application and succeeds in the goal of allowing access to various complex data types to a functional format for a two dimensional space. There remain potential improvements to this aspect of the project in the form of hosting the data in an API and potentially retrieving it during

runtime to save on memory if the user intends to use multiple air spaces. The scalability this would provide to the project would be highly valuable however there is a large quantity of administrative labour involved in the process of converting incomplete pdf data to their usable format despite attempts to automate it and as such was determined to fall outside the scope of the project's primary intent.

5.2 Object Logic

The objects, their functionality and their interactions with one another are defined here as object logic. The three primary object types are planes, nodes and paths with the basic structure being such that nodes are children of paths and have their positions informed by the path structure. Planes then use nodes to navigate the environment. The primary method that objects are identified in code is through the use of unity 'tags'. This allows objects to be efficiently searched for by type using one of Unity's many in-built functions.

```
var GameObject[] nodes; // nodes is declared as an array of game objects
nodes = GameObject.FindGameObjectsWithTag("Node"); // gets all nodes
```

Figure.41 Example of Search for other objects by tag

Each object described in this section has an instance of one or many scripts attached to it, some of which execute each frame. For this reason it is essential that searches for other objects and actions taken to modify other objects are performed only when necessary and that unnecessary checks are avoided in order to save processing time. Objects may modify variables or execute functions in scripts belonging to other objects through the use of the 'GetComponent<scriptname>()' family of functions provided they are publicly declared rather than static. This is a fast and effective way of accessing not just scripts but other functional components of objects such as renderers.

```
closest =
GameObject.Find("Controller").GetComponent<Controller>().FindClosestNode(tra
nsform)
```

Figure.42 Example of GetComponent

5.2.1 Nodes and Paths

In order to structure the order that planes travel between nodes, a structure is defined using the path objects. Path objects are manually created empty game objects which hold a script - 'pathScript.cs', which sorts all of the child objects belonging to it and assigns them identification numbers according to their order in the hierarchy. The paths lead into one another according to ids that are manually assigned in the engine using the interface. This hierarchical structure allows for the most efficient approach to creating a directional network.

```
public class pathScript : MonoBehaviour
{
    public int id = 0;

    void Awake() //on awake
    {
        int i = 0;

        foreach (Transform child in transform) //searches in grandchildren
for nodes
        {
            foreach (Transform subChild in child)
            {
                if (subChild.gameObject.tag == "Node")
                {
                    subChild.GetComponent<nodeScript>().pathId = id;
//assigns the node's path id
                    subChild.GetComponent<nodeScript>().id = i; //assigns
the node's id
                    i++;
                }
            }
        }
    }
}
```

Figure.43 pathScript.cs id assignment loops

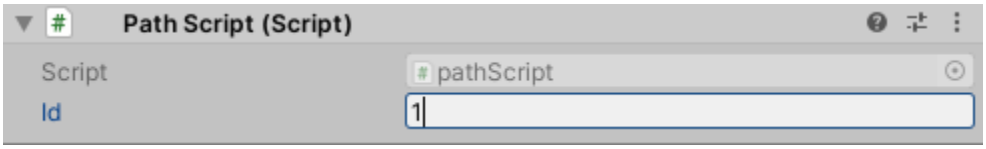


Figure.44 Example of Interface being used to assign path ids

The node object is instantiated from a prefab at each defined latitude and longitude point upon initial load. The object itself is displayed as a small coloured circle and has many attached components including two scripts, one for all of it's behaviours and another to hold its object data. Once all the nodes have been sorted into paths and the path has assigned all the nodes and id the nodes then perform a function FindNextNode() to find the node in its path with the next numerical id to their own. This is achieved by searching for all objects with the node tag and storing them in an array. This array is then filtered for nodes sharing a path id with the node in question and then returning the next node in the sequence. With this information the node stores the it's next node as a variable to be used later by plane objects and using a line renderer component, draws a line to this location. Exception cases have been included in this function to handle the first and last nodes in a path which have slightly differing behavior. They instead point to the first node in the next path in the sequence or if this is the final path, to the landing node.

```
private void Start() // executes only once on program start
{
    if (nextNode == null && !isLandingNode)
    {
        nextNode = FindNextNode(); // assign next in sequence
        x = 0;
    }
}
void Update()
{
    if (!isLandingNode)
    {
        //assigns id of the parent path
        pathId = transform.parent.parent.GetComponent<pathScript>().id;
        //draws straight line to next node
        line = GetComponent<LineRenderer>();
        line.SetPosition(0, transform.position);
        line.SetPosition(1, nextNode.transform.position);
    }
}
```

Figure.45 nodeScript.cs, find next node in path, assign parent path id and draw line

The node objects aside from having the attached line renderer component also have a collider attached to allow for detection of objects that enter or leave it's airspace. The collision events regarding this component are handled by the plane object.

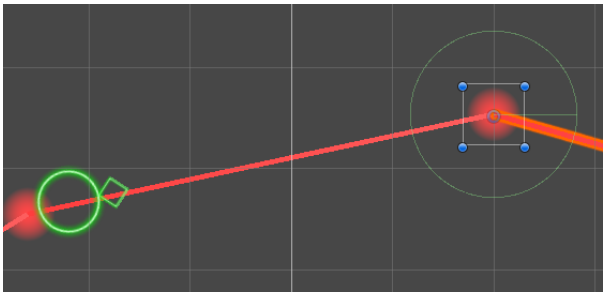


Figure.46 Node collider is the green circle surrounding the joint.

An alternate version of this node and path system exists in the application in the form of curved paths. These paths, generated by the bezierScript.cs create allow for the creation of paths with curvature and automatically generate their own nodes with adjustable resolution of nodes. This is necessary when creating holding patterns for planes to enter and follow indefinitely while waiting for permission to land. Using Unity's 'Gizmos' allows for modification of the handles of these curves in the editor allowing for precision when being designed.

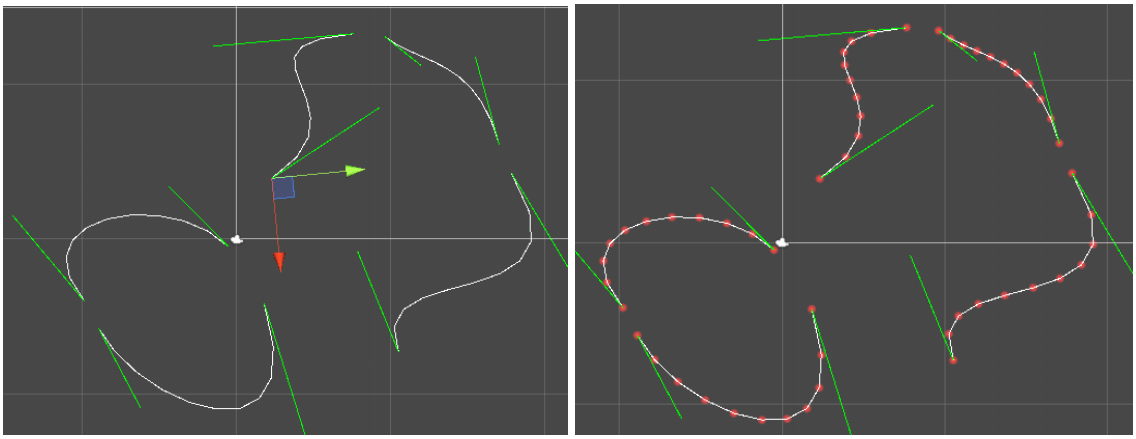


Figure.47 & 48 Bezier paths being created in editor on left and the resultant node structure on right - bezierScript.cs

The effect of these curves is achieved by utilising the DeCasteljau's algorithm which involves acquiring the midpoints of the start and end points of each curve as well as its handles iteratively in layers so that nodes can be placed at a specific percentages along the path according to the desired resolution using a Catmull-Rom spline. The version of this algorithm in effect here is a modified version of its transposition to C#.

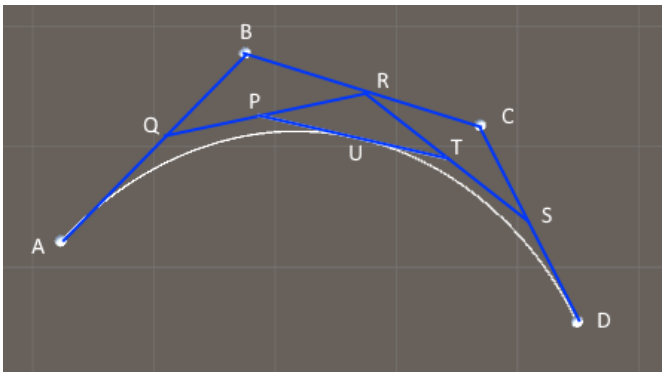


Figure.49 Example of Catmull-Rom Spline using CastelJau's Algorithm to solve for midpoints (Source: *Everything about interpolation in Unity with C# code - Bezier curves | Habrador*, n.d.)

5.2.2 Planes

The plane object is the only moving component in the application and as such is the trigger for the majority of events that occur during runtime. This means it has been a priority to ensure that the method of motion for these vehicles is as authentic as possible. To this effect an interpolation is used to angle the plane's forward direction toward it's target over time according to it's turning speed. This results in a gradual movement which softens as it arrives at its desired angle. The second component of this rotation refers to locking the object to the two dimensional plane while preserving a Vector3 object as there is lesser functionality for physics available for Vector2 transforms.

```
          if (target != null)
        {
            transform.up = Vector3.Lerp(transform.up,
(target.transform.position - transform.position), turnSpeed); //rotate
towards target node
            transform.eulerAngles = new Vector3(transform.eulerAngles.x, 0,
transform.eulerAngles.z); // locks y axis rotation
        } else
        {
            transform.up = Vector3.Lerp(transform.up, transform.forward,
turnSpeed); //Angle forward
            transform.eulerAngles = new Vector3(transform.eulerAngles.x, 0,
transform.eulerAngles.z); // locks y axis rotation
        }
```

Figure.50 Plane turning logic, angles toward target

```
rb.AddForce(transform.up * thrust);
```

Figure.51 Acceleration using the rigidbody component to apply thrust using the physics engine.

The thrust applied to the vehicles is informed by the rigidbody component attached to the plane which allows Unity's physics engine to apply forces to it. AddForce() was used here rather than translating or interpolating the object's position as this allows for a reliable method of collision detection and applies appropriate restrictions to the possible methods of movement due to the nature of velocity.

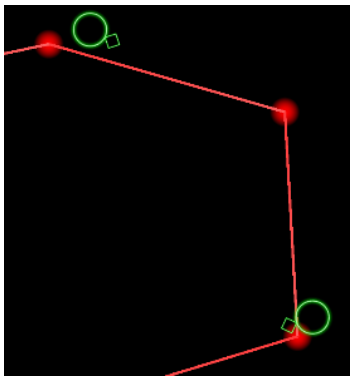


Figure.52 Planes following a path of nodes, automatically adjusting heading

As the plane makes contact with the colliders attached to other objects in the environment a set of behaviours defined within Unity's in-built OnTriggerEnter2D() function are initiated. Depending on the tag associated with the collided object, different groups of logic are executed. The most frequent of these is when a plane enters the exterior circular area of a node and is assigned a new target node; however scoring and landing logic are also handled here and are associated with collisions with other planes and the landing node/airport.

```
void OnTriggerEnter2D(Collider2D col) //returns collider of other object
    {
        if (col.gameObject.tag == "Node" && col.gameObject == target &&
!col.GetComponent<nodeScript>().isLandingNode)
        { //node collision logic
            if(col.GetComponent<nodeScript>().nextNode != null)
            {
```

```
            target = col.GetComponent<nodeScript>().nextNode;
```

Figure.53 Portion of commands executed upon collision with node

0

```
foreach (GameObject node in nodes)
      {
          Vector3 diff = node.transform.position - position;
          float curDistance = diff.sqrMagnitude;
          if (curDistance < distance)
          {
              closest = node; // assigns closest node
              distance = curDistance;
          }
      }
      foreach (Transform child in closest.transform)
      {
          foreach (Transform subChild in child)
          {
              if (subChild.gameObject.tag == "Node")
              {
                  n = subChild.gameObject;
              }
          }
      }
      return n;
```

Figure.54 Distance calculation in 'StartHold()' function in planeScript.cs

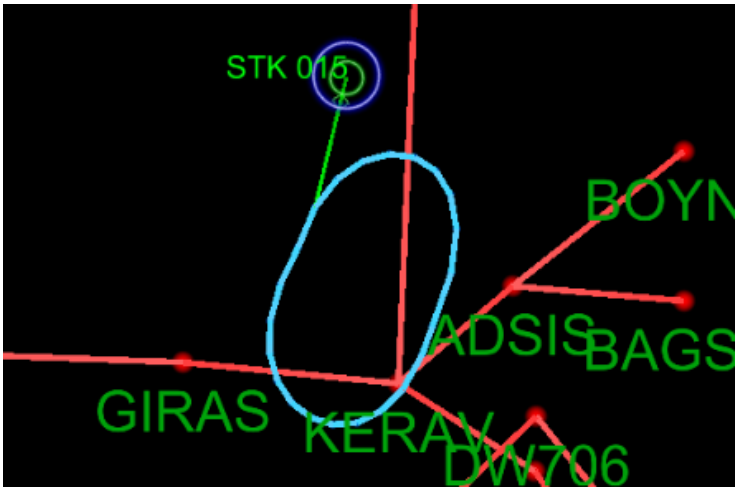


Figure.55 Plane STK 015 enters holding pattern(in light blue)

As is apparent in the above example the planes are also assigned a call sign according to Dublin airport naming convention. This is generated automatically to be unique upon the plane's instantiation by drawing from an array of these standard codes and concatenating them into a string and comparing to existing callsigns to exclude duplicates.

```
    string GenerateName()
    {
        string[] prefix = {"BCY", "EIN", "IBK", "RYR", "SZS", "STK" };
        return prefix[Random.Range(0, prefix.Length)] + " " +
(Random.Range(0, 9)).ToString() +
(Random.Range(0, 9)).ToString() +
(Random.Range(0, 9)).ToString();
    }
```

Figure.56 Callsign name generation function - planeScript.cs

The plane, node and path relationship is robust and reliable in it's purpose of informing the vehicle's direction automatically. It contains an easily traced sequence of events to follow for the implementation of additional features or logic at any stage in the sequence which enables the game structure to be built on top of it. With regards the complexity of the automation component of this network, it was originally designed with the intention of deeper levels of complexity however upon the implementation of user controls the costs involved with a more reactive AI only served to detract from the amount of interactivity present in the application.

5.3 Game Logic

The Game logic pertains to the elements of interactivity that are present in the application as well as the structure of the user interface objects and the gamified scoring system. In order to organise the code into various places without creating an overabundance of small scripts a system was developed to break down the non-object functionality below a certain size. The 'Controller.cs' script handles the player input and interaction as well as user interface element manipulation while the 'GameManager.cs' script is responsible for the majority of game flow management such as score and time keeping as well as keeping and object instantiation. There are several large modules however such as the camera and edge indicators which are contained in their own scripts in order to be more easily accessible due to their complexity and the frequency with which they interact with other game objects.

5.3.1 Sliding Panels

Following the design specification for the user interface, there are several panels implemented with varying functionality; however the first stage of implementation of these features was the means by which the UI elements are accessed. The panels can be clicked to slide in and out of view of the screen enabling a clear view of the game environment is possible at all times.

To achieve this functionality the application makes use of an external library called 'LeanTween' (Dented Pixel, n.d.). This allows the panels to simply interpolate their positions based on their current state between two predefined points. This is a necessary alternative to using standard unity features for this functionality as usually any animation taking place on Unity's UI canvas causes a complete redrawing of every asset on the plane each frame. With LeanTween we can move the objects using programming and avoid the use of key frame animation and state machines.

```
public void SlidePanel(GameObject p)
    {
        if (selected != null || p.name != "PlanePanel") {
            var pd = p.GetComponent<PanelData>();

            if (p.GetComponent<RectTransform>().position.x <= pd.inPos) {
                LeanTween.moveX(p, pd.outPos,
0.4f).setEase(LeanTweenType.easeOutQuad);
                p.GetComponent<Image>().sprite = pd.inSprite;
                pd.state = true;
            } else
            {
                LeanTween.moveX(p, pd.inPos,
0.4f).setEase(LeanTweenType.easeOutQuad);
                p.GetComponent<Image>().sprite = pd.outSprite;
                pd.state = false;
            }
        }
    }
```

Figure.57 Slide Panel function in Controller.cs

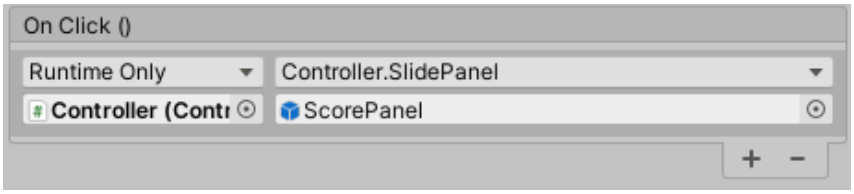


Figure.58 Describing Unity button behaviour

In order to avoid writing multiple functions for each panel, the panel itself holds data for the positions it means to target in a script 'Panel-Data.cs'. These are then passed to the function as parameters when the button is pressed and the panel is animated accordingly based on its current state which is stored as a boolean. The opportunity is then taken to reverse the direction of the arrow on the panel by replacing it's sprite.

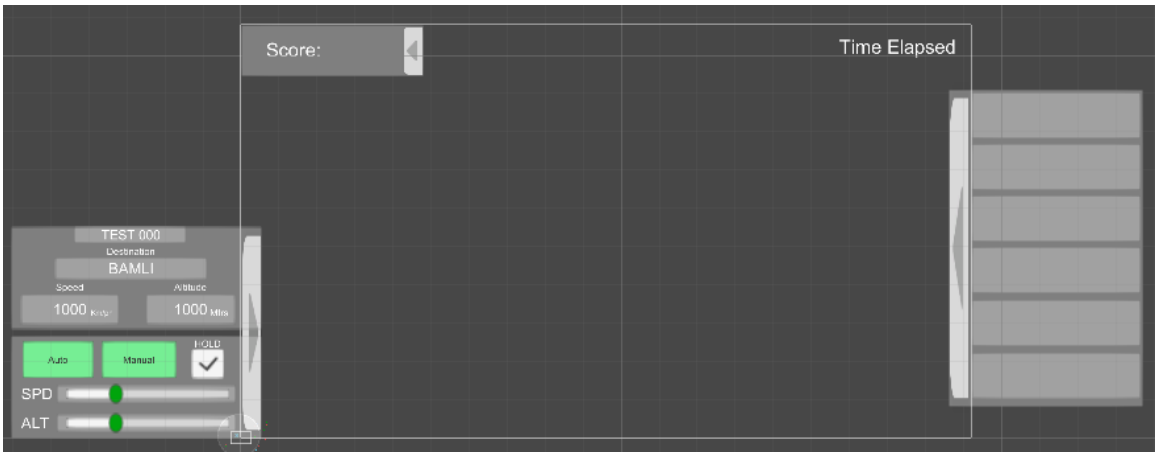
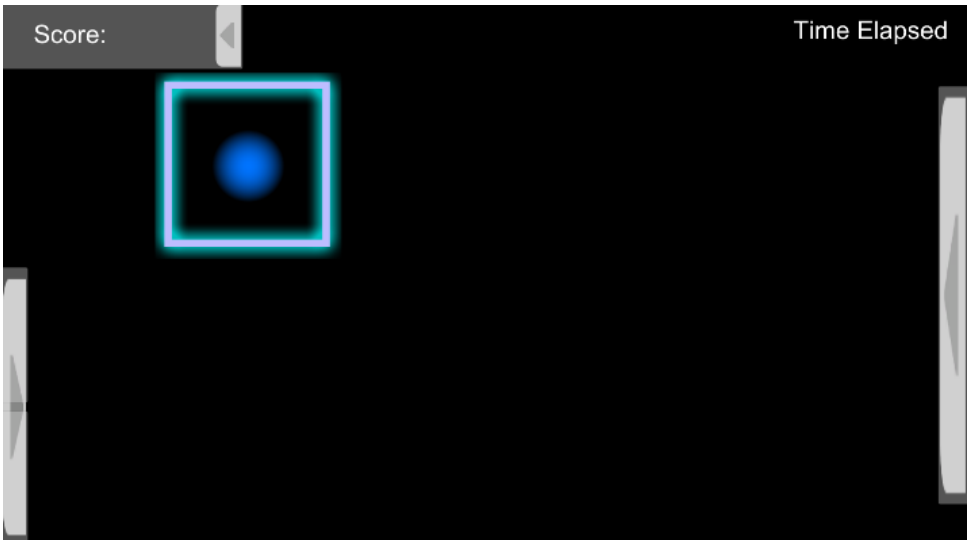


Figure.59 & 60 Above panels slid outside of the viewport. Below game view of the above



5.3.2 Input

In order to establish a means of player interactivity in the application it was expected that the player would be using a mouse. To get information from a mouse click in Unity the application makes use of raycasting. When the mouse is clicked a hypothetical laser is drawn from the camera point that was clicked forward into space until it collides with an object. This point of intersection is stored in the controller script as the variable ‘rayhit’. The ‘rayhit’ object contains information regarding the gameobject that its intersection occurred on and can therefore be used to initiate various functionality.

```
if (Input.GetMouseButtonDown(0)) //on click
        {
            RaycastHit2D rayHit =
Physics2D.GetRayIntersection(Camera.main.ScreenPointToRay(Input.mousePositio
n)); // object reference to intersection point

//on click plane
            if (!angleSelectMode && !nodeSelectMode && rayHit.transform !=
null && rayHit.transform.gameObject.tag == "Plane") // if rayhit object is
plane
            {
                SelectPlane(rayHit.transform.gameObject);
            }
```

Figure.61 Mouse input with on plane click Controller.cs

The first piece of relative interactivity was to allow users to select a plane in order to see information regarding it and control it. In order to achieve this the controller stores the plane in a gameObject variable. This calls a function called ‘SelectPlane()’ executing several commands. A ring which exists out of the game bounds to that plane’s position and childed to it to maintain its relative position to the plane indicating that it has been selected, the plane draws a line to show its intended path and then the control panel slides out and the plane's variables are fed into the information section for display. The sliders in the control panel beneath it also adjust their fill amount to the appropriate level for the selected plane.

```
  public void SelectPlane(GameObject r)
   {
       if (selected != null) //confirms first click
       {
```

```
            PanelAnim(panel); // toggle slider position
            selected.GetComponent<LineRenderer>().enabled = false; //turn
off renderer of previous
        }
        else
        {
            selected = r;
            panel.GetComponent<Button>().onClick.Invoke(); //activates panel
button
        }

        GameObject plane = r;
        selected = plane;
        slider.value = plane.GetComponent<planeScript>().thrust;
        selector.SetActive(true);
        selector.transform.position = selected.transform.position; //move ui
hover to plane
        selector.transform.SetParent(selected.transform);
    }
```

Figure.62 SelectPlane() function in Controller.cs

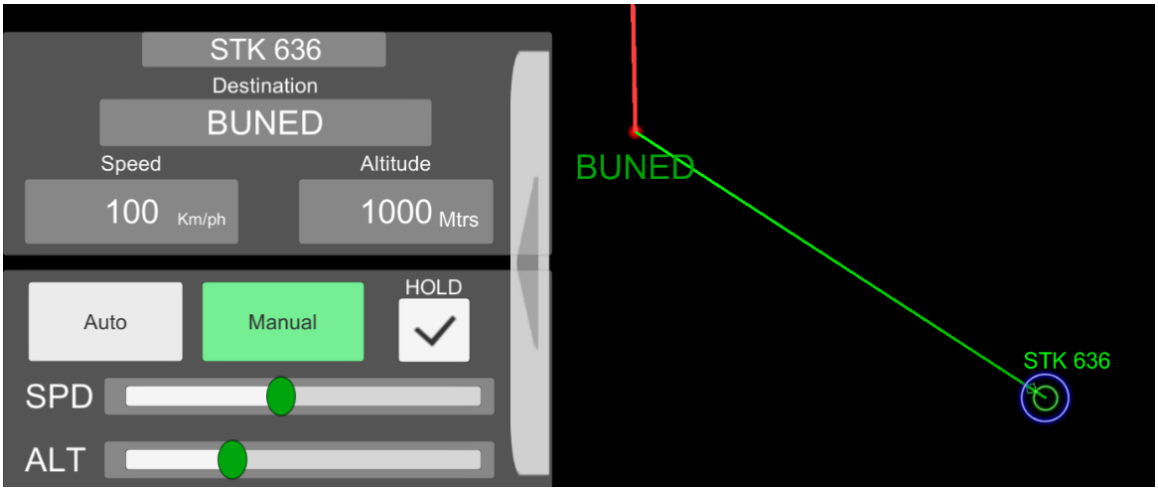


Figure.63 Selected Plane with panel displaying its data. Blue ring to indicate selection and green projected path to show intent

This same click functionality is used with nodes to activate and deactivate a dropdown menu containing information about the speed and altitude information on that node.

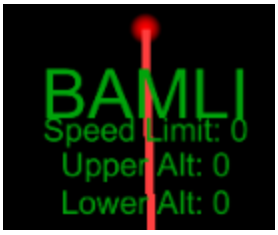


Figure.64 Clicked node displaying dropdown data

The control elements on the panel itself are made up of Unity's sliders and button components . The slider's simply adjust the variables of the selected plane while the buttons execute specific functions which change the behaviour of clicks. The auto button activates a variable named 'NodeSelectMode' which causes the next click, should it target a node to set that node as the selected plane's target while the manual button activates 'AngleSelectMode' which instantiates an invisible object at the location the the next click and sets that as the planes target. These buttons allow the user to control the plane's path either by choosing its automated path or alternatively giving it a specific heading.

```
            if (nodeSelectMode)
            {
                if (rayHit.transform != null &&
rayHit.transform.gameObject.tag == "Node")
                {
                    selected.GetComponent<planeScript>().target =
rayHit.transform.gameObject;
                    nodeSelectMode = false;
                }
            }

            if (angleSelectMode)
            {
                    selected.GetComponent<planeScript>().target =
tempObject.transform.gameObject;
                    angleSelectMode = false;
            }
```

Figure.65 The angle and node select mode clauses for click behaviour

Finally there is a checkbox element which simply toggles a boolean used to determine if a plane should enter a holding pattern or proceed to land when encountering a 'HoldingSwitch' node.

The final user interactable element of the interface is the list panel. This panel allows the user to not only see the callsigns of the incoming planes but also to click any of the listed names to lerp to camera to center on the corresponding plane's position.

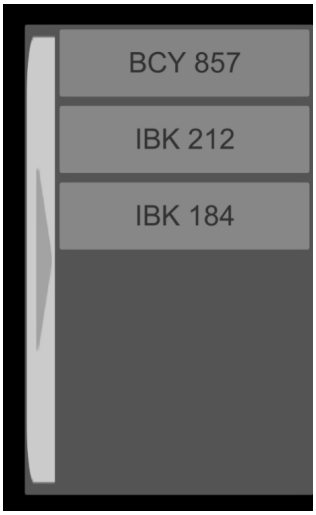


Figure.66 example of info panels

In order to accomplish this the list was established as an array 'infoPanels', which is not rendered unless it has text content. Text content is then assigned by looping through each of the buttons in the panels for each plane in the scene, assigning that plane's callsign as the text. This way the panels automatically enable and disable themselves for the quantity of planes in the scene.

```
        //handles info list panel
        planes = GameObject.FindGameObjectsWithTag("Plane");

        foreach (GameObject i in infoPanels)
        {
            if (i.GetComponentInChildren<Text>().text == "")
            {
                i.GetComponent<Image>().enabled = false;
            } else
            {
                i.GetComponent<Image>().enabled = true;
            }
```

```
        }

        int smallerArray = infoPanels.Length;

        if (planes.Length < infoPanels.Length)
        {
            smallerArray = planes.Length;
        }

        for(int i = 0; i < smallerArray; i++) // informs info panels
        {
            infoPanels[i].GetComponentInChildren<Text>().text =
(planes[i].GetComponent<planeScript>().call);
        }
```

Figure.67 code in function UpdateUI which handles the info panel - Controller.cs

5.3.3 Camera

The behaviour for the purposes of user control is that it may be panned about the environment using arrow or 'wasd' keys as well as zoomed in and out using the mouse scroll wheel. This was implemented in the script 'CameraMove.cs' which utilizes the unity input axes system to translate the camera's position according to a variable; speed. The use of input axes rather than listening for specific key presses allows for not only a large reduction in the required code but also allows for scalability should the project eventually include input from devices other than mouse and keyboard as the inputs have a generic set of expected inputs depending on the device in use.

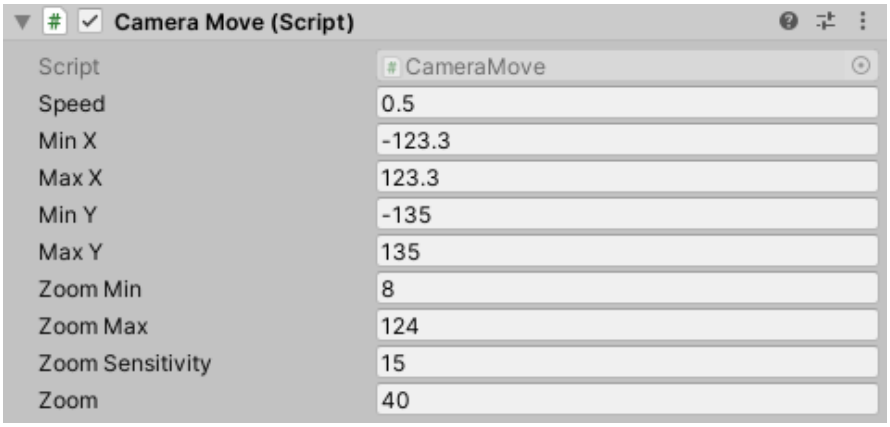


Figure.68 Camera's publicly accessible variables for in-editor adjustment

The camera zoom makes use of the ‘Mathf.Clamp’ function to set a floor and ceiling for the size of the camera’s viewport by accessing the camera’s ‘orthographicSize’ component.

```
        float xAxisValue = Input.GetAxis("Horizontal") * Speed;
        float yAxisValue = Input.GetAxis("Vertical") * Speed;
        scrollValue = Input.GetAxis("Mouse ScrollWheel") * zoomSensitivity;


        transform.position = new Vector3(
            Mathf.Clamp(transform.position.x + xAxisValue, minX, maxX),
//limits bounds
            Mathf.Clamp(transform.position.y + yAxisValue, minY, maxY),
            transform.position.z);

        zoom -= scrollValue;
        zoom = Mathf.Clamp(zoom, zoomMin, zoomMax);
        Camera.main.orthographicSize = zoom;
```

Figure.69 Camera controls as executed each frame according to inputs

This script also adjusts the scale over text and UI elements as the screen is zoomed in order to maintain readability of user interface elements in the game environment as the screen becomes larger or smaller. This is achieved for text by simply adjusting the font size to be a factor of the zoom value after each change while with more complex graphical elements such as the screen edge indicators, there are zoom amount breakpoints which rescale the objects to appropriate sizes.

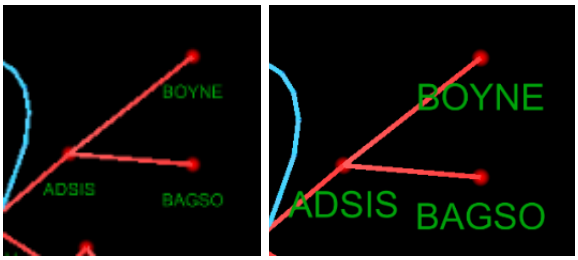


Figure.70 Two versions of highly zoomed out image of small section of screen with no rescaling on left

5.3.4 Other UI elements

It became apparent during gameplay implementation that there were difficulties keeping track of off screen planes and their movements and as such a system was developed to subtly highlight the direction and distance of off screen planes. These edge indicators are placed at the edge of the screen nearest their corresponding plane and display the distance to that plane as an integer.

```
         float tempx = Mathf.Clamp(plane.transform.position.x,
leftLimitation+objectWidth, rightLimitation-objectWidth);
        float tempy = Mathf.Clamp(plane.transform.position.y,
upLimitation+objectWidth, downLimitation-objectWidth);

        transform.position = new Vector3(tempx, tempy,0);
```

Figure.71 Snippet of 'EdgeInicators.cs' showing the movement attempted by these UI elements

This is achieved by having each plane's edge indicator only render while that plane is not currently being rendered by the camera which is determined using unity's 'visible' property. The edge indicator itself is clamped to the screen bounds and is in a constant state of attempting to move towards it's plane within these bounds.

Figure.72 Edge Indicators in red circles showing the approximate location of off-screen planes

The hypothesized minimap was implemented using an additional omnipresent camera which displayed an overall view of the environment; however this was an expensive

module as it caused all objects to be rendered twice as well as providing significant clutter to the user interface layout and as such was removed from the final version of the application.

Figure.73 Minimap module in test environment

5.3.5 Game Manager and Scoring

The 'GameManger.cs' script is responsible for the time and event management during game run time. This script is an instance and therefore does not exist on a specific object, allowing it to be referenced from other scripts with ease as an object reference is not required.

The 'InvokeRepeating()' function is used to recurrently call a function which creates new planes at appropriate time intervals at appropriate locations. These locations known as spawn points are stored in an array of out of bounds objects which can be adjusted conveniently in the editor to create multiple scenarios for gameplay variety. The rate at which the planes spawn is also dynamically adjusted according to in-game performance and time elapsed with the quantity and frequency increasing as time goes by.

```
private void Start()
{
   InvokeRepeating("SpawnPlane", spawnTime, spawnInterval); //called
recursively at every interval
}
```

Figure.74 'GameManager.cs' InvokeRepeating() function accepting adjusted variables.

Due to the already established nature of the collision based event structure of the in-game objects it was convenient to simply alter the score variable stored here by amounts based on the type of event. Each time a plane successfully reaches a node a score is given based on several factors such as if the plane is at the appropriate speed limit and altitude for that node as well as upon landing. Score is decremented for the amount of time planes are within an unacceptable range and altitude of one another also. The time taken to land a plane is also considered and any time less than four minutes results in additional points earned.

5.3.6 Menus

The final necessary feature of the application is a series of menus that may be navigated to determine which components of the model to load. This was achieved using Unity's scene management library in a script named 'SceneManager.cs'. This script simply contains functions which load scenes according to an array index which is organized in the build settings. To access these functions, new scenes are created containing buttons which directly call to this script.

```
public void MainMenu()
{
    UnityEngine.SceneManagement.SceneManager.LoadScene(0);
}
```

Figure.75 Example of function called by button to return to the main menu at scene index 0 'SceneManager.cs'

The result of this structure is a highly intuitive and functional set of navigation tools so that the user can access their intended portion of the application.

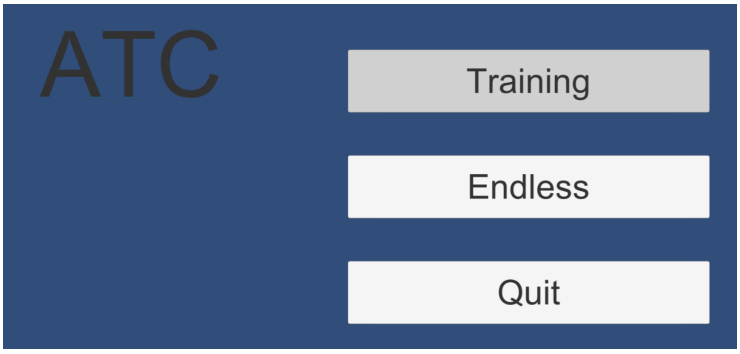


Figure.76 Basic main menu structure

5.3.7 Deployment

Building the application for the target platform of Windows and Mac was achieved using unity's build tools. This allowed me to produce a high performance standalone application with only the necessary files in their most compressed form. The building tools also permit for the inclusion features such as splash art when launching the application as the selection of a desktop icon.

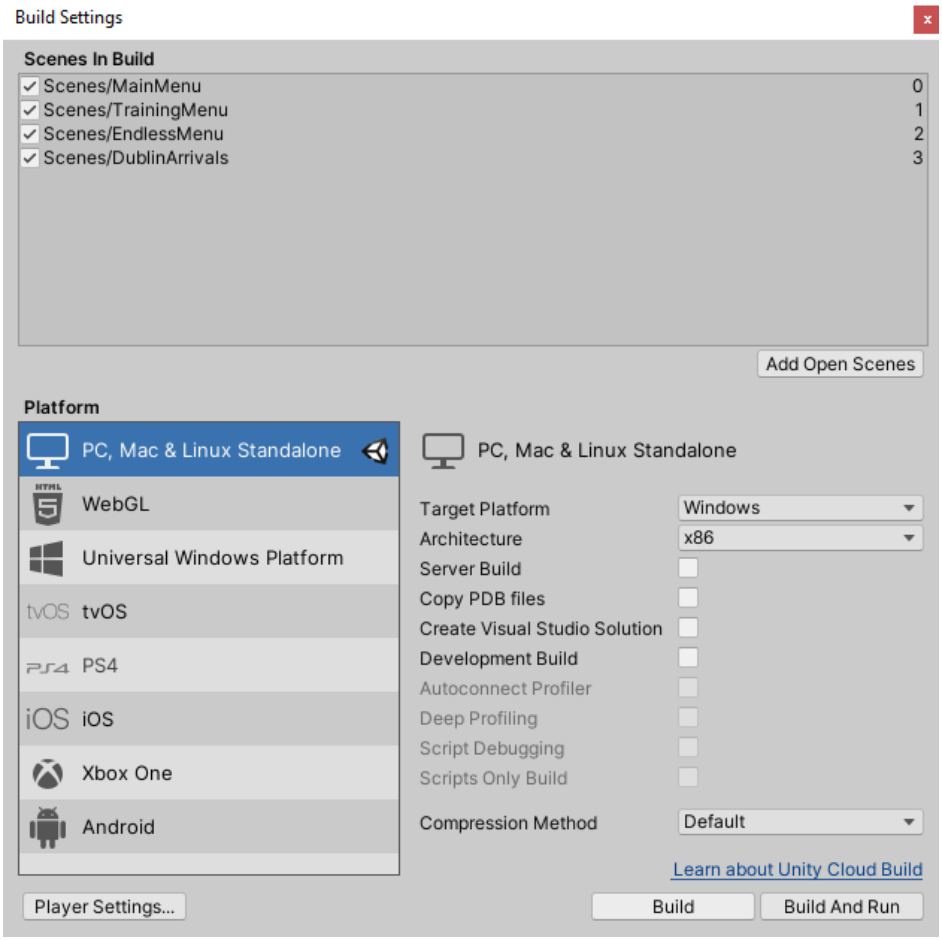


Figure.77 Unity's build setting menu

The scenes and the order they are defined in unity's build settings are finalized during this process as well as the target resolution among other presentation options. The resultant group of files produced includes a convenient '.exe' file which launches the game on execution.

6. Testing

6.1 Introduction

It was essential to perform testing of the application at all stages of development in order to assure the application was meeting the expectations of the design specifications and not causing issues either in the generation of unintended behaviour and in overall performance. To this end the test plan as outlined in 'Requirements and Feasibility' was carried out.

6.2 Unit and Integration

Unit tests are performed in unity using the test runner window which allows test scripts to be executed which provide assembly references to the script being tested. These are then, for the majority of cases within the project, tested by comparing expected values to those that are returned. This system allowed the development process to be iterated upon without concern regarding foundational code units behavior.

```
namespace Tests
{
    public class PlaneTest
    {

        [UnityTest]
        public IEnumerator NewTestScriptWithEnumeratorPasses()
        {
            var gameObject = new GameObject();
            gameObject.AddComponent<planeScript>();

            var n =
gameObject.GetComponent<planeScript>().FindClosestNode();
            yield return new WaitForSeconds(1f);

            Assert.AreEqual(expected: "Node", actual: n.tag);
        }
    }
}
```

Figure.78 Example of Unity PlayMode Unit Test for Plane function

In this example the ‘FindClosestNode()’ function is tested by determining if the return object is in fact an object with the node tag by using the ‘Assert.AreEqual’ method which accepts an expected and actual value as parameters to compare.

By testing many components in conjunction in this way, the integrated parts were confirmed to be functional. This testing framework provided a stable structure for tracing and debugging issues throughout the development process.

6.3 System Testing

Performance testing over the overall system was carried out at regular intervals in order to determine that the new components were not resulting in unacceptable frame per second values.

The method through which performance was monitored in Unity was the profiler tool. This records the usage of CPU and GPU resources per frame and produces detailed graphs and reports on the execution times of each component and script.

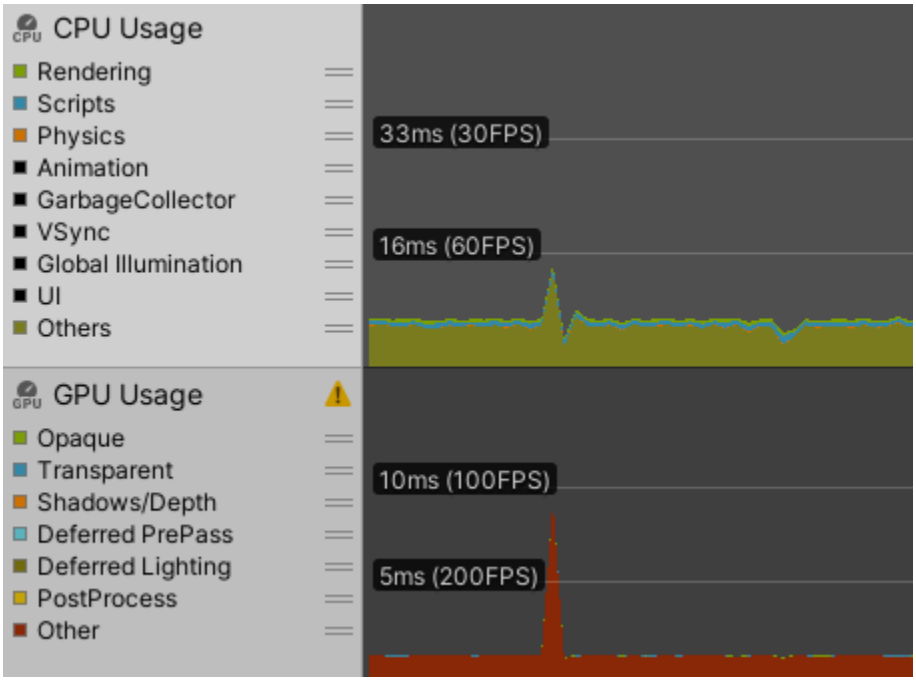


Figure.79 CPU and GPU usage spike in Unity Profiler for completed application

In this graph displaying CPU and GPU usage of the application during runtime it can be observed that spikes to approximately 60 FPS occur. These spikes occur at the beginning of each frame as the next frame is processed using the latest positional data. The custom scripts written by me can be seen in blue and make up less than 8% of the overall process per frame.

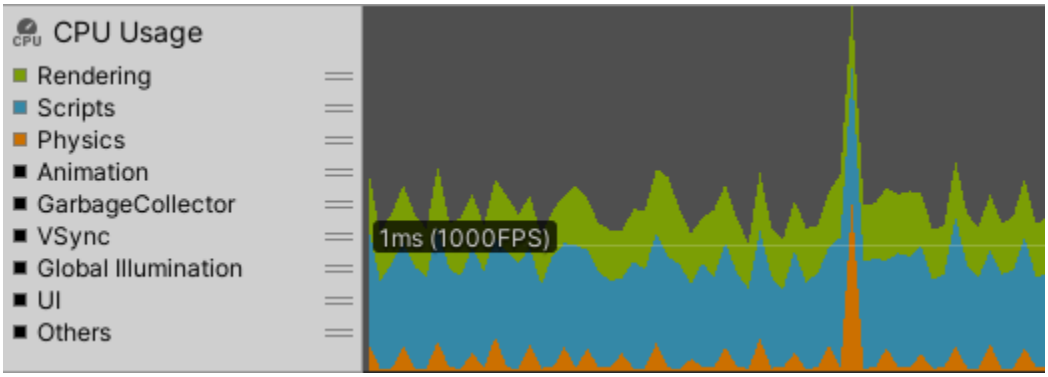


Figure.80 Breakdown of CPU usage spike not including standard processes demonstrating less than 1ms response time

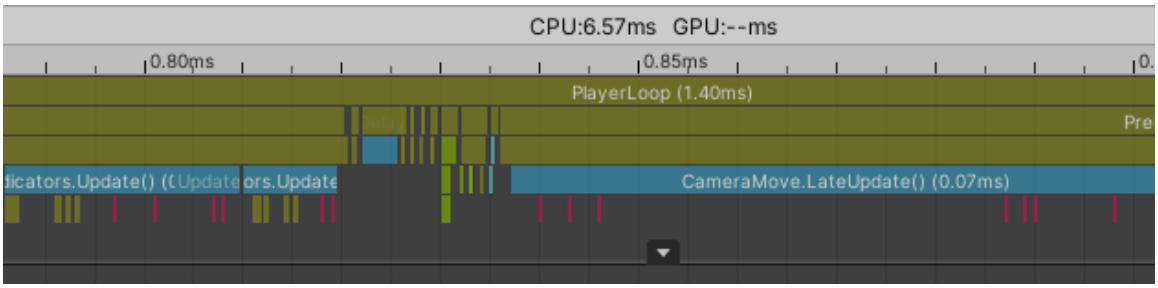


Figure.81 Closeup of processes in execution

As the bulk of processing is made up of the Unity Engine's standard rendering processes it stands to reason that of the custom scripts written for the application, the slowest response time belongs to the camera movement script as this trigger re-renders upon usage. These response times are still well within tolerable levels, however there were points in development such as with the inclusion of the minimap module which contained a second camera that the performance fell outside of these limits.

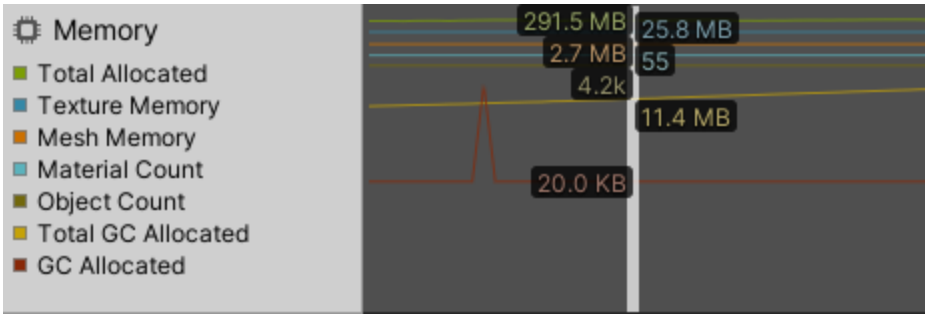


Figure.82 Memory usage graph during runtime

Memory usage also does not exceed the automatically allotted 315mb and is stable throughout gameplay, the reason for this being that the application is designed such that assets are, with the exception of changes in scene, not making calls to load different assets into RAM.

6.4 User Testing

In order to assess users' comprehension of the application and the intuitiveness of the user interface system, five users were asked to use the application and play the endless mode for five minutes, record their score and then repeat this process twice more. The users score at the end of each round as well as the number of planes landed was recorded and have been graphed below using tableau software Fig 83. The scores awarded are primarily earned through landing planes however additional opportunities to earn points through effective use of the node network as well as time efficiency exist. For reference the highest score recorded at this stage of game balancing inside this time frame was 17,740 with this being achieved by a user very familiar with the application. Prior to testing users were shown an image of the interface and provided a brief description of how these elements would function and what their goals would be onced the simulation began.

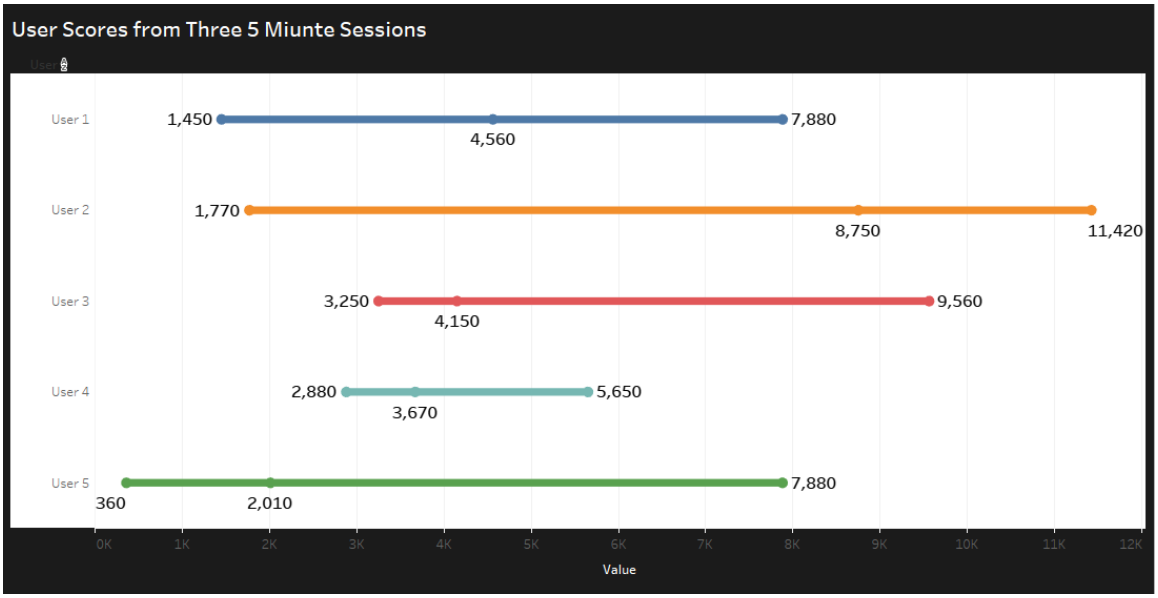


Figure.83 User testing graph of scores from three 5 minute sessions

From this graph it can be observed that the users' scores always trended upwards with no user ever producing a lesser score than the previous attempt. This analysis also demonstrated the variability in user competency for the application with there being a potentially quite steep learning curve for certain users.

This testing was performed in order to assess the requirements of the training portion of the application through the determination of which components of the program were least intuitive and how best users could be introduced to them. It was highlighted by many users that, in particular the method holding patterns and clearing to land was the least intuitive portion of the design and as such will be addressed among other valuable insights. While the methodology for the testing was informal and therefore does not produce any reliable outcomes, the primary objective of its undertaking was in the improvement of the applications features and future development.

7. Results and Analysis

7.1 Development Analysis

The results of this project were in the successful creation of an air traffic control software through the implementation of motion planning techniques. The components that were implemented in order to fulfill this object were a simulation environment occupied by objects with interaction behaviours, a user interface system which would allow for intuitive interactions and user agency and a data system for interpreting real world data and converting that to a simulated environment.

The project also succeeded in delivering an application which could handle a large number of simulated objects in real time without loss in any significant performance or integrity. This allowed the experience of the application to be unhindered by these factors, leading to more consistent results.

At the outset of the project there were several components proposed that were tested such as the inclusion of several user interface elements like a minimap or a user accessible data input system. These components were either removed due to incompatibility with the project's main goals or falling outside the scope of these goals.

7.2 Ludic Simulation Analysis

This application, as outlined as one of the key aims, incorporated the gamified elements in it’s user incentive design through a scoring system and user interface style. This was achieved while adhering as closely as possible to the design as outlined by traditional air traffic control systems. The combination of both of these elements was successful in increasing the accessibility of otherwise very advanced features, while highlighting the difficulty in creating a middle-ground between entertainment and functionality.

It was observed throughout development, the tendency to lean more heavily on the side of game or simulation. The benefits of the game format being increased user engagement through the evocation of adrenaline or catharsis whereas the benefits of simulation seem to lie more in the realm of immersiveness via realism and accuracy. It required rigorous planning and constant readjustment to remain on the course of creating an application between these spaces so as to observe the potential benefits.

It could be argued that ludic simulations such as these are niche in their potential application areas as many softwares would prefer to be constructed with the more rigid model of game or simulation, however in this instance, with the project goal being to create a training tool to a potentially inaccessible subject matter, that it was appropriate. The air traffic control application that was developed is likely to give users the greatest possible understanding of the operations and responsibilities carried out by air traffic controllers without alienating the users with the assumption of necessary prerequisite knowledge.

8. Conclusion

It was the intention of this project to explore the possibility of use cases for motion planning in the creation of simulation applications. To this effect the project broke the development into three key stages. The preparation of data which was accomplished through various conversion processes. The development of the simulation environment and object behaviours which was achieved through utilization of the Unity game engine and finally the development of an intuitive and highly modular user interface system.

This project proposes that there is value in the use of motion planning techniques in the development of any partially automated software which involves a simulation environment as it was the research and implementation of technologies in this area that most significantly enabled the application's goals.

The secondary project aim was to explore the potential of software developed in the nebulous space between game and simulation dubbed 'ludic simulation'. This was executed through the use of gamification techniques in the scoring system, user interface style and general object relationship model present in the application.

The implications of the use of this type of technology in creating applications lie primarily in the increase in accessibility for the introduction of complex systems to the general user. It is proposed that when considering the development of a simulation software that the inclusion of gamified elements at the expense of realism may only be valid if the intended users experience level of the subject matter is already relatively low. With users highly familiar with the simulated space being more likely to gain more from a more traditional simulation.

The application was developed with scalable systems that could be expanded on to greatly improve the potential value of the product. With larger data sets the current model of local .json storage could be pulled from an API or a feature could be implemented to allow users to create and edit their own preferred node network layouts. These features alongside general aesthetic improvements may lead to greater reach and applicability for this project in the future.

Finally, it should be noted the potential to modify the model used in this application and apply it to other traffic systems using node based networks and autonomous vehicles such

as road or satellite traffic may be possible as very little of the existing object structure would require large quantities of modification to enable this.

9. References (APA)

Abd Algfoor, Z., Sunar, M. S., & Kolivand, H. (2015). A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *International Journal of Computer Games Technology*, *2015*, 1–11. https://doi.org/10.1155/2015/736138

Ceipek, J. (2010). Game Path Planning. Retrieved December 8, 2019, from Jceipek.com website: http://jceipek.com/Olin-Coding-Tutorials/pathing.html

Dented Pixel. (n.d.). *LeanTween | Animation Tools | Unity Asset Store*. Assetstore.unity.com. Retrieved April 25, 2021, from https://assetstore.unity.com/packages/tools/animation/leantween-3595

Dubbels, B. (2013). Gamification, Serious Games, Ludic Simulation, and other Contentious Categories. *International Journal of Gaming and Computer-Mediated Simulations*, *5*(2), 1–19. https://doi.org/10.4018/jgcms.2013040101

EDR. (2015, October 18). *Thales's Watchkeeper achieves another first in aviation history*. EDR Magazine. https://www.edrmagazine.eu/thaless-watchkeeper-achieves-another-first-in-aviation-history

Everything about interpolation in Unity with C# code - Bezier curves | Habrador. (n.d.). Www.habrador.com. Retrieved April 25, 2021, from https://www.habrador.com/tutorials/interpolation/2-bezier-curve/

Excel To Json Converter - BeautifyTools.com. (n.d.). Beautifytools.com. Retrieved April 25, 2021, from https://beautifytools.com/excel-to-json-converter.php

Global ATC Simulator. (n.d.). Download Full Games on DFGames.net. Retrieved April 25, 2021, from https://dfgames.net/6664-global-atc-simulator.html

Go, J., Vu, T., & Kuffner, J. J. (2004). Autonomous behaviors for interactive vehicle animations. *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '04*. https://doi.org/10.1145/1028523.1028525

Greenky, S. (2019, October 21). *WEYTEC and Raytheon Team Up to Enhance Air Traffic Management Technology*. Www.businesswire.com. https://www.businesswire.com/news/home/20191021005591/en/WEYTEC-and-Raytheon-Team-Up-to-Enhance-Air-Traffic-Management-Technology

javascript - how to convert latitude/longitude to pixels in 3d using three.js. (n.d.). Stack Overflow. Retrieved April 25, 2021, from https://stackoverflow.com/questions/32657688/how-to-convert-latitude-longitude-to-pixels-in-3d-using-three-js

Jon, D. (2006). *If you never fall, you are not really trying*. Retrieved from https://skatgame.net/mburo/ps/thesis_demyen_2006.pdf

Parker, J., & Becker, K. (2014). *The Simulation-Game Controversy: What is a Ludic Simulation? Choosing and Using Digital Games in the Classroom -A Practical Guide View project*. https://doi.org/10.4018/jgcms.2013010101

Permana, S. D. H., Bintoro, K. B. Y., Arifitama, B., & Syahputra, A. (2018). Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game. *IJISTECH (International Journal of Information System & Technology)*, *1*(2), 1–8. https://doi.org/10.30645/ijistech.v1i2.7

Reed, H. (2017). EUROPEAN ORGANISATION FOR THE SAFETY OF AIR NAVIGATION EUROCONTROL EUROPEAN AIR TRAFFIC MANAGEMENT PROGRAMME Font Requirements for Next Generation Air Traffic Management Systems.

Reddy, H. (2013). PATH FINDING - Dijkstra's and A* Algorithm's. Retrieved December 3, 2020, from studylib.net website: https://studylib.net/doc/8098164/path-finding---dijkstra-s-and-a--algorithm-s

Reynolds, C. (1999). *Steering Behaviors For Autonomous Characters*. Retrieved from https://www.red3d.com/cwr/papers/1999/gdc99steer.pdf

Reynolds, C. (2004). Website Demonstrating Steering Behaviors For Autonomous Characters. Retrieved December 6, 2020, from www.red3d.com

Russell, S., & Norvig, P. (2010). *Artificial intelligence : a modern approach*. Pearson.

Sakai, A., Ingram, D., Dinius, J., Chawla, K., Raffin, A., & Paques, A. (2018). PythonRobotics: A Python code collection of robotics algorithms. 1-8. Retrieved December 2, 2020, from https://arxiv.org/pdf/1808.10703.pdf.

SimpleJSON - Unify Community Wiki. (n.d.). Wiki.unity3d.com. Retrieved April 25, 2021, from http://wiki.unity3d.com/index.php/SimpleJSON

Simplify PDF. (n.d.). *Convert PDF to Excel - SimplyPDF*. Simplypdf.com. Retrieved April 25, 2021, from https://simplypdf.com/Excel

Software Testing Levels. (2011, January 5). SOFTWARE TESTING Fundamentals.https://softwaretestingfundamentals.com/software-testing-levels/statgrid. (2015, December 11). *Endless ATC*. https://store.steampowered.com/app/666610/Endless_ATC/

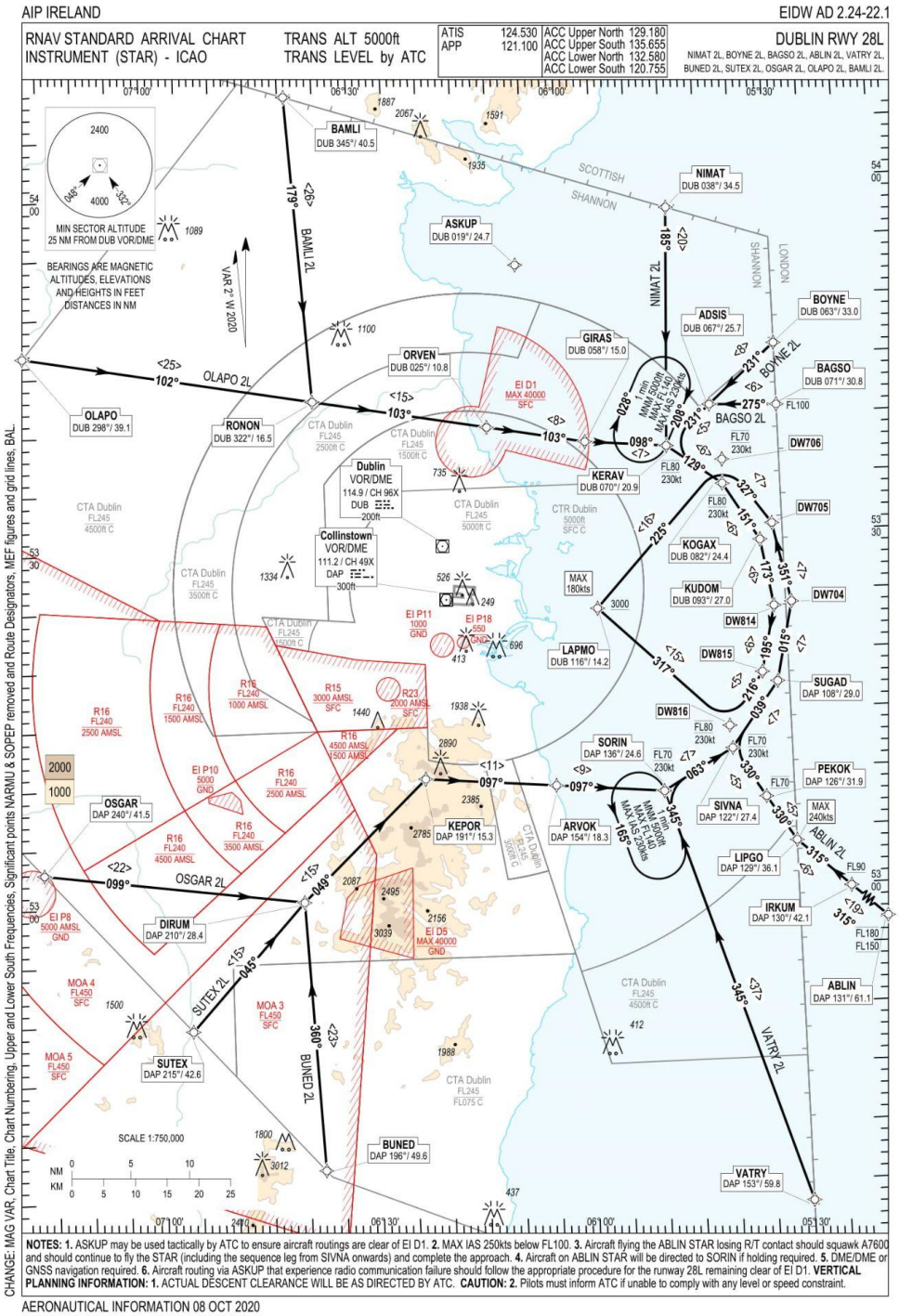
Song, B., Guohui Tian, & F. Zhou. (2010, December). A comparison study on path smoothing algorithms for laser robot navigated mobile robot path planning in... Retrieved December 6, 2020, from ResearchGate website: https://www.researchgate.net/publication/290161139_A_comparison_study_on_path_smoothing_algorithms_for_laser_robot_navigated_mobile_robot_path_planning_in_intelligent_space

Technologies, U. (n.d.). *Unity - Manual: Order of execution for event functions*. Docs.unity3d.com. https://docs.unity3d.com/Manual/ExecutionOrder.html

Zafar, A., Agrawal, K. K., & Anil Kumar, Wg. C. (2018). Analysis of Multiple Shortest Path Finding Algorithm in Novel Gaming Scenario. *Advances in Intelligent Systems and Computing*, 1267–1274. https://doi.org/10.1007/978-981-10-5903-2_132

10. Appendices

10.1 Appendix A



10.2 Appendix B

OLAPO2L STAR RWY 28L
OLAP2L

Nav. Spec.	WPT Name	Latitude (N)/ Longitude (W)	Path Term	Fly-By Fly-over	True track / Mag track	Distance (NM)	Upper limit / Lower limit	Speed Limit (kts)	Remarks
RNAV1	OLAPO	534649.0 / 0071740.6	IF	-	-	-	-	-	-
RNAV1	RONON	534233.9 / 0063619.2	TF	Fly-by	099.6 / 102	24.9	-	-	-
RNAV1	ORVEN	533953.5 / 0061129.8	TF	Fly-by	100.1 / 103	15.0	-	-	-
RNAV1	GIRAS	533821.0 / 0055733.2	TF	Fly-by	100.4 / 103	8.4	-	-	-
RNAV1	KERAV	533742.7 / 0054557.3	TF	Fly-by	095.2 / 098	6.9	FL080 / FL080	230	Turn L
RNAV1	KOGAX	533418.6 / 0053814.1	TF	Fly-by	126.5 / 129	5.7	FL080 / FL080	230	Turn R
RNAV1	KUDOM	532925.8 / 0053314.3	TF	Fly-by	148.6 / 151	5.7	FL080 / FL080	230	Turn R
RNAV1	DW814	532347.5 / 0053141.1	TF	Fly-by	170.6 / 173	5.7	FL080 / FL080	230	Turn R
RNAV1	DW815	531812.9 / 0053346.6	TF	Fly-by	192.7 / 195	5.7	FL080 / FL080	230	Turn R
RNAV1	DW816	531346.9 / 0053844.4	TF	Fly-by	213.9 / 216	5.3	FL080 / FL080	230	Turn R
RNAV1	LAPMO	532411.0 / 0055644.1	TF	Fly-by	314.1 / 317	15.0	- / A3000	180	Turn R

OSGAR2L STAR RWY 28L
OSGA2L

Nav. Spec.	WPT Name	Latitude (N)/ Longitude (W)	Path Term	Fly-By Fly-over	True track / Mag track	Distance (NM)	Upper limit / Lower limit	Speed Limit (kts)	Remarks
RNAV1	OSGAR	530257.9 / 0071612.8	IF	-	-	-	-	-	-
RNAV1	DIRUM	530009.7 / 0063940.0	TF	Fly-by	097.0 / 099	22.2	-	-	-
RNAV1	KEPOR	531016.5 / 0062200.7	TF	Fly-by	046.3 / 049	14.7	-	-	Turn L
RNAV1	ARVOK	530919.0 / 0060335.1	TF	Fly-by	094.8 / 097	11.1	-	-	Turn R
RNAV1	SORIN	530829.3 / 0054822.5	TF	Fly-by	095.1 / 097	9.2	FL070 / FL070	230	-
RNAV1	SIVNA	531152.3 / 0053827.7	TF	Fly-by	060.3 / 063	6.9	FL070 / FL070	230	Turn L
RNAV1	SUGAD	531722.5 / 0053139.8	TF	Fly-by	036.5 / 039	6.9	FL070 / FL070	230	Turn L
RNAV1	DW704	532403.7 / 0052910.1	TF	Fly-by	012.6 / 015	6.9	FL070 / FL070	230	Turn L
RNAV1	DW705	533046.6 / 0053126.4	TF	Fly-by	348.6 / 351	6.9	FL070 / FL070	230	Turn L
RNAV1	DW706	533621.3 / 0053806.9	TF	Fly-by	324.6 / 327	6.9	FL070 / FL070	230	Turn L
RNAV1	LAPMO	532411.0 / 0055644.1	TF	Fly-by	222.5 / 225	16.5	- / A3000	180	Turn L

SUTEX2L STAR RWY 28L
SUTE2L

Nav. Spec.	WPT Name	Latitude (N)/ Longitude (W)	Path Term	Fly-By Fly-over	True track / Mag track	Distance (NM)	Upper limit / Lower limit	Speed Limit (kts)	Remarks
RNAV1	SUTEX	524927.7 / 0065549.3	IF	-	-	-	-	-	-
RNAV1	DIRUM	530009.7 / 0063940.0	TF	Fly-by	042.3 / 045	14.5	-	-	-
RNAV1	KEPOR	531016.5 / 0062200.7	TF	Fly-by	046.3 / 049	14.7	-	-	Turn R
RNAV1	ARVOK	530919.0 / 0060335.1	TF	Fly-by	094.8 / 097	11.1	-	-	Turn R
RNAV1	SORIN	530829.3 / 0054822.5	TF	Fly-by	095.1 / 097	9.2	FL070 / FL070	230	-
RNAV1	SIVNA	531152.3 / 0053827.7	TF	Fly-by	060.3 / 063	6.9	FL070 / FL070	230	Turn L
RNAV1	SUGAD	531722.5 / 0053139.8	TF	Fly-by	036.5 / 039	6.9	FL070 / FL070	230	Turn L
RNAV1	DW704	532403.7 / 0052910.1	TF	Fly-by	012.6 / 015	6.9	FL070 / FL070	230	Turn L
RNAV1	DW705	533046.6 / 0053126.4	TF	Fly-by	348.6 / 351	6.9	FL070 / FL070	230	Turn L
RNAV1	DW706	533621.3 / 0053806.9	TF	Fly-by	324.6 / 327	6.9	FL070 / FL070	230	Turn L
RNAV1	LAPMO	532411.0 / 0055644.1	TF	Fly-by	222.5 / 225	16.5	- / A3000	180	Turn L

VATRY2L STAR RWY 28L
VATR2L

Nav. Spec.	WPT Name	Latitude (N)/ Longitude (W)	Path Term	Fly-By Fly-over	True track / Mag track	Distance (NM)	Upper limit / Lower limit	Speed Limit (kts)	Remarks
RNAV1	VATRY	523316.0 / 0053000.0	IF	-	-	-	-	-	-
RNAV1	SORIN	530829.3 / 0054822.5	TF	Fly-by	342.6 / 345	37.0	FL070 / FL070	230	-
RNAV1	SIVNA	531152.3 / 0053827.7	TF	Fly-by	060.3 / 063	6.9	FL070 / FL070	230	Turn R
RNAV1	SUGAD	531722.5 / 0053139.8	TF	Fly-by	036.5 / 039	6.9	FL070 / FL070	230	Turn L
RNAV1	DW704	532403.7 / 0052910.1	TF	Fly-by	012.6 / 015	6.9	FL070 / FL070	230	Turn L
RNAV1	DW705	533046.6 / 0053126.4	TF	Fly-by	348.6 / 351	6.9	FL070 / FL070	230	Turn L
RNAV1	DW706	533621.3 / 0053806.9	TF	Fly-by	324.6 / 327	6.9	FL070 / FL070	230	Turn L
RNAV1	LAPMO	532411.0 / 0055644.1	TF	Fly-by	222.5 / 225	16.5	- / A3000	180	Turn L

Hold Identification – EIDW AD 2.24-22.1

Holding Fix	Latitude (N) / Longitude (W)	Inbound True Track (degrees)	Inbound Mag Track (degrees)	Maximum Indicated Airspeed (kts)	Minimum Holding Altitude (ft)	Maximum Holding Level (FL)	Outbound time (min)	Direction of Turn
KERAV	533742.7 / 0054557.3	205.6	208	230	A5000	FL140	1	R
SORIN	530829.3 / 0054822.5	342.4	345	230	A5000	FL140	1	L