

A comparison of formation movement techniques in aerial flight

**Name: Daniel Luke Bailey**

**Submission Date: [Date]**

**Supervisor: Chris Janes**

This report is submitted in partial fulfilment of the requirement for the degree of **BSc (Hons) Games Programming** by **Daniel Luke Bailey**.

All content quoted in this paper from other authors publications have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this paper have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this could be considered plagiarism and will be considered grounds for failure in this project and the degree as a whole.

Daniel Luke Bailey

# Abstract

This paper aims to compare most effective method for aerial formation movement from existing implementations found through research. This will be achieved by exploring the available methods for both ground and air-based AI and choosing a selection which will then be built in order to be put through a series of tests. These test results will then be used in order to find the strengths and weaknesses of each implementation and determine which is the most effective overall.

Contents

[Abstract 3](#_Toc70782366)

[Chapter 1: Introduction 1](#_Toc70782367)

[Background 1](#_Toc70782368)

[Research Focus 1](#_Toc70782369)

[Aims and objectives 1](#_Toc70782370)

[Outline chapters 1](#_Toc70782371)

[Chapter 2: Literature Review 3](#_Toc70782372)

[General Background 3](#_Toc70782373)

[Games Using formation movement 3](#_Toc70782374)

[Formation types 3](#_Toc70782375)

[Formation Movement Techniques 4](#_Toc70782376)

[Steering Behaviors 5](#_Toc70782377)

[Which of these systems will be compared? 6](#_Toc70782378)

[Chapter 3: Implementations 7](#_Toc70782379)

[Introduction 7](#_Toc70782380)

[Consistent features and implementation 7](#_Toc70782381)

[Formation Manager 7](#_Toc70782382)

[Formation Base Class 8](#_Toc70782383)

[Individual implementations 9](#_Toc70782384)

[Implementation 1: Blind Formation 9](#_Toc70782385)

[Implementation 2: Autonomous Behaviour 10](#_Toc70782386)

[Implementation 3: Remove the Leader 12](#_Toc70782387)

[Conclusion 12](#_Toc70782388)

[Chapter 4: Testing and Analysis 13](#_Toc70782389)

[Introduction 13](#_Toc70782390)

[Performance Testing 13](#_Toc70782391)

[Experience Testing 14](#_Toc70782392)

[Conclusion 15](#_Toc70782393)

[Chapter 5: Conclusion 16](#_Toc70782394)

[References 17](#_Toc70782395)

[Appendix: 18](#_Toc70782396)

[Appendix 1 18](#_Toc70782397)

[Appendix 2 18](#_Toc70782398)

[Appendix 3 26](#_Toc70782399)

[Appendix 4 26](#_Toc70782400)

[Appendix 5 31](#_Toc70782401)

[Appendix 6 31](#_Toc70782402)

[Appendix 7 33](#_Toc70782403)

[Appendix 8 40](#_Toc70782404)

[Appendix 9 41](#_Toc70782405)

[Appendix 10 42](#_Toc70782406)

[Appendix 11 49](#_Toc70782407)

[Appendix 12 49](#_Toc70782408)

# Chapter 1: Introduction

## Background

It is the authors belief that cohesive AI movement and interaction is a key aspect for creating an enjoyable gameplay experience. Most existing aerial combat games have AI opponents which act completely independently from one another at all times, rarely moving together in a co-ordinated manner. This research paper aims to address the lack of methods for creating formation movement for aerial AI by examining and adapting methods which exist for ground-based movement. There are two key terms this research will use, the first is “formation movement” which refers to a set group of AI units which use pre-set logic in order to move through the level as a co-ordinated group which maintains a desired shape. The second key term is “Steering behaviours”, which is used to refer to an additional set of logic which can be used in addition to an AI’s movement in order to guide or change the result of the base movement logic.

## Research Focus

The primary focus of this research is to examine what impact specific formation AI techniques have on performance and player experience. The research will focus on exploring existing methods for formation movement and how they can be adapted for aerial movement, along with what impact to both the games performance and the player experience each method has. The impact of these implementations will be measured through both testing from players and tests conducted by the author in order to gain a range of perspectives on each implementations impact.

## Aims and objectives

This research has three main objectives which focus on the exploration, development, and analysis of existing methods for formation movement within game AI.

The first objective of the overall aim is to find and explore existing methods of formation movement across both ground and aerial units. This will involve research to establish a wide range of currently available methods and detail the logic required to recreate them, along with discussing how any ground-based implementations could be adapted into aerial movement. The purpose of this stage will be to use this research when selecting which implementations to use in overall analysis.

The second objective is to build a selection of the explored methods, while adapting them to aerial movement where required. Using the previously mentioned research, a selection of the existing implementations which show the most potential for differing behaviours will be created within an existing aerial combat game, using the basic principles found from the research as a guide to the logic behind the formations movement and overall decision making.

The final objective is to carry out tests on the built implementation in order to determine which is most and least effective method. These tests will be split between how the implementation impacts the overall performance of the game and how it impacts the players experience of the game. This will allow for conclusions about the effectiveness of each implementation to be made from both aspects, along with an overall conclusion, rather than from a technical or player only perspective.

## Outline chapters

The literature review aims to detail the research undertaken into existing methods of formation movement and how they operate. Focus will be placed on how the units within the formation find their target position within the world based on their assigned formation spot and how units within the formation are designed to react to any obstacles in the formations path. In addition it will also explore other existing methods for enhancing AI movement behaviours. The purpose of this addition will be to outline how these additional methods can be used to improve or alter an implementation in order to yield different results that may not be achievable.

This is followed by the implementations chapter, which aims to outline how the implementations selected from the literature review have been created. The main focus of this detailed exploration is to cover how each implementation differs in order to create their unique movement behaviours. However the chapter will also cover the key systems shared across each implementation which help to form the complete flight behaviour.

The testing and evaluation chapter will bring together the results of tests carried out on the previously mentioned implementations, covering what conclusions can be drawn from the results. This chapter will be where each implementations effectiveness is compared against the others in order to determine which is most effective and to identify any strengths and weaknesses of individual implementations

Finally the conclusion chapter will discuss if the aims and objectives mentioned above have been met, detailing how each has been completed. In addition, this chapter will also cover what the author has learned from completing the research, in order to help guide similar research in the future.

# Chapter 2: Literature Review

## General Background

The Encyclopedia Britannica defines formation flight as “two or more aircraft traveling and maneuvering together in a disciplined, synchronized, predetermined manner” (Gary, 2020). Encyclopedia Britannica traces the idea of moving in a formation first being developed during the First World War, where pilots noticed an increase in victories when pairs of aircraft engaged in combat rather than working alone. This then continued into the Second World War where the techniques were refined, and pilots experimented with aspects such as distance and positions. However, in the modern-day flight formations are primarily used at air shows to display a group of pilot’s skills at controlling the aircraft. (Gary, 2020)

Naturally, formation flight eventually emerged in video games as developers aimed to have their aircraft act like real fighter squadrons, likely to improve a player’s immersion in the game, with games such as *Star Wars Battlefront 2* (Dice, 2017) starting missions with aircraft flying in formation until engaged in combat, at which point the formation is broken and a dogfight sequence begins.

## Games Using formation movement

Formation movement exists in a wide variety of games and for a range of reasons, spanning from mimicking military tactics for realism to visual appeal during set sequences. While most games do not share the specific implementation for their formation movement, in some cases it is clear to players what rules a formation is following.

For example, in *“Total War: Warhammer 2”* (Creative Assembly, 2017) units begin in a large squad formation which moves when instructed while maintaining the overall shape of the squad formation. However, while individual units staying within the squad formation bounds they are not directly locked into position, this leads to individual troops falling out of line while en-route to the selected location but reforming the formation either at the goal position or after the squad’s movement allows. In addition, squad formations will also resize automatically depending on the number of units left in the squad, thus leading to a more realistic look during combat as a unit compensates for losses and fills gaps in the formation.

In contrast *“Star Wars Battlefront 2”* (Dice, 2017) uses aerial formation movement in the mission “Inferno” to add to the cinematic nature of the gameplay experience. The section of the mission has several groups of TIE fighters moving in formation around a Star Destroyer with the player following for a set amount of time until combat starts, at which point the formation breaks to give players an experience similar to an aerial dogfight. Unlike *Total War* (Creative Assembly, 2017), the player has no control over how the formation moves as the section is scripted until combat begins, making the formation entirely for the visual appeal and immersion of the player. However, it is still a useful example of showing how fighters might move in formation while on patrol around a larger objective, as well as how aerial fighters adapt ground-based formations to an aerial space.

## Formation types

From research undertaken for this project the author has determined that the positioning of units is a key aspect of a formation; without defined positions a maneuver is more difficult to co-ordinate and distinguish from regular flight. Dawson (2002, p.273) lists 7 common formations, shown in (Dawson, 2002, p.273, fig 5.6.1), which offer a range of advantages while in flight until combat. These can then be further diversified and adapted with small rotational adjustments in order to enhance the advantages of moving in a select formation. For example, the “Wedge” and “Box” formation offers greater protection from the front and sides to the units in the centre of formation, thus making them useful for escorting important units with maximum protection. Mars and Chanut (2015, p.219) note that it is not always best to assign units to their closest slot when creating a formation, this is especially important when creating a formation for a purpose such as the one previously mentioned. Without slot assignment in these formations key units may be in vulnerable positions and thus be in more danger.

The positions a unit should take in a formation are also highly dependent on the speed of the unit compared to that of the overall formation, which Mars and Chanut (2015, p.221) call Velocity Correction. When a unit’s speed is less than that of the most common unit in the formation the overall group must match that speed to ensure that the formation holds, otherwise slower moving units can quickly fall out of formation. The same is true for units which are faster than the most common unit, as this can also lead to the formation being broken and units left exposed.

Most formations are easy to add additional units to as in most cases a formation will either expand in a specific direction or add extra “ranks” as suggested in Dawson (2002, p.277). This allows for formations to not be size dependant and instead shrink and expand when needed, with the formations pathfinding also adjusting for the correct number of units.

From this research the project will aim to ensure that the formations compared will have the ability to adapt based on the size of the formation and the features of the units it contains, while also maintaining one of the common formation shapes in order to limit the number of variables between formations and implementations. Doing so will keep test results fair as in each implementation the number of formations, shapes and sizes will be maintained in order to fairly test the performance impact of the specified implementation.

## Formation Movement Techniques

From the research the author has also determined that the implementation of formation movement depend heavily on the complexity requirements of the formation. In some scenario’s units in a formation a will require come control over their behavior, whereas others will simply need to move towards their slot.

Mars and Chanut (2015, p.219) refer to the most basic example formation movement as a “Blind” formation, which sees the units within a formation lose almost all individual behaviour. Instead the movement of the formation is handled as an entire group with individual units only able to calculate for movement towards their assigned slot position within the formation and unable to react as individual units to obstacles. This leads to formations moving to avoid obstacles in the path of any unit as a group to prevent the formation being broken. They also note a key potential improvement to this system, which has units calculate the velocity needed to reach their slot position and apply that velocity as long as it does not exceed the unit’s maximum. This is a useful improvement to make to the system as it prevents units from moving too far over their target position or prevents units from being unable to reach the target in a single simulation update, thus preventing what they refer to as “motion jolts”.

However, Mars and Chanut (2015, p.220) suggest that in many cases it is more beneficial for units to retain some of their own behaviours which are influenced rather than overwritten by other units in the formation. For this purpose, they suggest a method referred to as “Autonomous Formation following“, which allows units of a formation to handle their own collision detection and break away from the formation in order to avoid obstacles, rather than forcing the whole formation to avoid an obstacle which will only block a single unit. Doing this allows for formations to look more realistic when adapting to overcome obstacles instead of a strict rule followed by all units not matter the situation. The units within the formation can then also be assigned “sub-goals” to complete while completing the movement in order to give a formation more ‘personality’ and to make the units within seem individual to those around it.

In contrast, Millington (2009, p.148- 149) suggests an implementation which he refers to as “Removing the Leader”. This implementation does not have a selected fighter as the “leader” of the formation and instead has an invisible formation leader which moves and updates the formations slots. This invisible leader does not need to avoid obstacles and as such the obstacle avoidance is left to the individual units within the formations. Similar to other methods discussed this avoids large formations moving to avoid an obstacle which would only impact the path of a single unit and also avoids the “leader” unit dictating the avoidance for all other units within the formation. Millington (2009, p.149) notes that this method usually leads to a simpler implementation as the invisible formation follows its own pathing and the unit’s behaviour is kept separately but is heavily guided by the formation.

Alternatively, Pottinger (1999, p.34) suggests a method of “Halving and Re-joining” in order to avoid obstacles within a formation. This implementation involves creating 2 smaller formations from the main formation and each smaller formation separating to navigate around the obstacle before re-joining with the other. In this method the split occurs at the unit with the obstacle in its path and aims to have units retake their previous slots after the obstacle has been successfully navigated. Pottinger (1999, p.34) states that this addition greatly increases the visual appeal of a formation, likely due to its elimination of units moving alone to avoid any obstacles.

Each implementation discussed varies in how the overall formation will move throughout a scene and with each difference units will behave differently. These methods serve as a useful starting point for this project, as each is explored in more detail before developing a select range to compare with the others. With that further detail the implementations which will yield the best comparisons will be more apparent.

## Steering Behaviors

When referring to an AI agent, the term ‘Steering Behavior’ is commonly used to refer to the way in which an agent generates and then follows its path through a level. As Millington and Funge discuss (2002, p.55- 95), steering behavior implementations are highly dependent on their use, but most have a similar structure. Millington and Funge (2002, p.55- 56) note that most steering behaviors take a character and target input, which depends on the function of the agent, and use the data along with the behaviors set of rules to generate or influence a unit’s movement.

Seemann and Bourg (2004, p.7) state that the most common steering behavior found across game AI is that of ‘Chasing and Evading’, which involves AI agents taking the role of either the predator or prey. Seemann and Bourg note that this steering behavior involves both agents path generation and steering being heavily influenced by the location of the other group of agents, with ‘predator’ agents being influenced to guide them towards prey agents as they attempt to reach the ‘prey’s’ location, while the ‘prey’ agents are influenced to move away from any nearby predator agents towards what the agent deems to be safety. Such behavior could be applied to an overall formation to keep the formation moving in an expected way, either towards or away from a target position or unit.

An alternative method suggested by Bjore (2014, p.289) is to steer a formation using a method known as “steering circles”. This method uses circles to generate a turn radius for a formation’s slots based on the entrance and exit points of the circle, along with the maximum turn size units in the formation can make. Bjore (2014, p.290) states that using circles to calculate a formations path first requires an entrance, exit and centre for the circle which is then used to calculate the curve needed in order to smoothly rotate from the entrance to exit. Once this calculation is complete the curve is then passed into the units within the formation starting with the first row, this row then begins to carry out the turn and the implementation then continues row by row. For all rows but the first the formation must calculate which way the row is required to turn based on their position relative to the previous row and its turning direction, once this is established the innermost unit based on the turn direction is rotated towards its column counterpart in the row ahead and thus creating a fluid curve across the entire formation.

This method could then be mixed with the idea of predicting positions discussed by Pottinger (1999, p.37 - 38), which highlights the usefulness of predicting a unit’s movement path for collisions or errors before undertaking the path generated. Pottinger (1999, p.39) also suggests a method similar to steering circles in this section, further enforcing the potential use for both systems at once.

Reynolds (1999, p.11) suggests moving units in a ‘flocking’ pattern as a more advanced steering behavior, within which agents use the locations of the units around them in order to determine where to steer towards. Reynolds discusses how a combination of several steering behaviors can lead to a more varied movement within a group of units, with each factor being weighted to the influence over the unit’s movement and thus leading to a flocking behavior rather than a strict behavior. This would lead to more fluid movement within a formation as units would not be completely rigid to their slot position and would instead have limited free movement and the potential to change slots with another.

For situations with lots of AI agents, Buckland (2004, p.126) suggests a method known as ‘Spatial Partitioning’, which divides the game world into ‘cells’ and allows units to only check for collisions in the ‘cells’ around the agent. He notes that this method can lead to large speed improvements as units check the obstacles around their own ‘cell’ rather than every obstacle in the scene. This would also greatly improve the speed for formation implementations as it limits the number of obstacles for the formation, or units within, to check against when calculating obstacle avoidance.

## Which of these systems will be compared?

In order to cover a wide range of implementations the author believes it is important to consider how similar implementations are to others and which can be combined for potentially more effective results. With this in consideration three implementations have been selected based on their difference to each other and overall complexity. The three chosen are: “Blind Following”, “Autonomous Formation Following” and “Removing the Leader” as they offer not only a range in the complexity of the implementation but also a wide range of unit dependence on the formation. For example, in “Blind Following” the units within a formation are highly dependent on the formation when considering their movement. This is in contrast “Removing the Leader” where units are almost entirely self-dependant for the majority of behaviours and only rely on the formation when no obstacles or other key individual criteria is met.

# Chapter 3: Implementations

## Introduction

This chapter will discuss the approaches used in the implementation of formation movement for aerial / space-based craft. Three approaches are implemented; Blind, Autonomous and Remove the Leader, each with their own way of allowing units to move together in a formation.

While the movement behaviour of each formation will change across the different implementations, there are certain features which will be consistent throughout which will also be covered in this chapter. The aim in doing so being to minimize any impacts from systems outside the implementations requirements, such as spawning formations or reforming them.

## Consistent features and implementation

The implementation built to test each method has two main components; the formation manager and the formation base class. Both contain feature that remain the same across implementations.

### Formation Manager

The formation manager is responsible for creating, monitoring, and updating all active formations within the scene, it achieves this through a list containing every formation the manager has created along with a maximum number of formations the scene is allowed to contain. The function runs within Unity’s “FixedUpdate” as part of the movement process involves adding a RigidBody force to each fighter, which Unity recommends doing within “FixedUpdate” (Unity Technologies, no date). Within this loop the formation manager checks the number of active formations compared to the maximum, if the number is less than the maximum then the manager adds the number of required formations. The manager then calls the “UpdateFormation” function for every formation base class in its list of active formations (See Appendix 1).

The other key feature handled by the formation manager which will be consistent through all implementations is formation generation and spawning. Generation and spawning are called both by the start function and when the manager needs to replace a formation once its units have been defeated. While the code to behind creating the formation is placed in a function called by the formation base class’s “Start” function (See Appendix 2, line 57), it is only ever triggered by the formation manager and as such the manager becomes responsible for this feature. This covers the creation of new formations as well as placing them in the level (See Appendix 3), with the key aspect being the creation of slots for the number of units set to be in the formation. In the method created, the first slot is assigned to one of the random start positions around the area, which is then marked as taken to prevent other formations from choosing it as a start location. The other slot positions are then calculated from this base point and the formations pre-set slot offset, which is added to either the base position or the last position on the specified side of the formation (See Appendix 2, line 92).The resulting formation takes an arrow shape but can be modified by changing the pre-set offset and how each slot chooses which existing slot to add the offset to. In some implementations the base position and the unit assigned to it become marked as the formations “lead unit”, while in the “remove the leader” implementation this position simply serves as the first slot in the formation. It is important to keep this feature the same across all implementations as if the generation of formation differs then it could lead to anomalous results that are not caused by the formations behaviour itself and are instead caused by how the formation has been created.

### Formation Base Class

The base class is responsible for holding all data on a specific formation, this includes: all the formations slot positions, a list of active fighters in the formation and if the formation is in combat. The formation base class contains several functions which are important for the formation’s overall behaviour, the main function is referred to as “UpdateFormation” (See Appendix 2, line 180), which updates the formations position and is where the majority of the changes occur between each implementation. This function is responsible for updating the entire formation with both the different behaviours created by the differing implementations and the consistent features.

How the implementation handles breaking for combat and reforming after combat has ended is a key aspect which will influence how players will perceive the effectiveness of a formation, poor implementation can lead to unengaging dogfights or issues when reforming. When a unit within the formation is damaged the formation ceases its usual update function, instead the “UpdateFormation” function (See Appendix 2, line 180) begins to call the update function for each unit’s individual behaviour and allows them to move as free units rather than as a formation, in turn allowing for more dogfight style combat encounters.

Upon the formation breaking a timer is also set which updates each time a unit within the formation takes damage, regardless of if the formation is currently separated. If this timer is able to pass without being interrupted by another unit taking damage then the formation is set to no longer in combat and the “UpdateFormation” function (See Appendix 2, line 180) returns to updating the units as a formations, thus allowing them to being to move them back into a position and eventually as a formation again.

Having this as a consistent feature of all implementations is important as how a formation behaves when attacked will impact a player’s perception of how well the implementation is functioning. Thus, by maintaining the way in which a formation reacts to being attacked, the impact of behavioural differences is limited to when the units are moving as part of the formation, rather than the unit’s individual movement or how the formation responds to combat.

In addition, the formation base class also contains a function for responding to a unit within the formation being defeated. In the overall implementation, a formations response to losing a unit is divided into two distinct responses, if the formation has more than one unit remaining then it will remove the lost unit from its list of fighters and also remove the last slot position in the list of available positions (See Appendix 2, line 216). Doing so will move any units later in the formation down the list of active units, leaving an empty slot on the end which is then removed.

If the formation loses a unit which leaves a single fighter remaining a function is called which checks to find the nearest active formation. Once this has been established the unit remaining is added to the located formation as a single unit, generating a new slot position and adding the unit to the found formations list of units. The formation then removes itself from the formation managers list of active formations and then deletes the attached game object (See Appendix 2, line 252).

For this behaviour to work well the formation base class also contains a function for adding a fighter to the formation, which generates a new slot position on the desired formation and adds the unit passed into the function the list of active fighters for that formation (See Appendix 2, line 292). This is done in a similar way as when the formation is first adding units upon being generated in order the new slot to be created in the same way and fit with the formation.

Similar to when the unit responds to combat, it is important that all implementations respond to losing a unit in the same way in order to avoid any changes in performance which would occur due to the difference in method and interfere with the results of the test.

Certain behaviours within the formations specific movement will also be consistent across each implementation, such as the random movement behaviour (See Appendix 2, line 365). This movement involves selecting a point within the bounds of the level (See Appendix 2, line 383) and rotating either the lead ship or each fighter individually, accounting for their offset, to face that location. This location changes on a random interval between a minimum and maximum delay or when the formation has reached the chosen point, thus allowing the formation to continually move around the playable area.

## Individual implementations

The majority of changes across each implementation occur within the “UpdateFormation” function in the formation base class, for easy identification when building and reviewing code these changes have been added to a separate function referred to as “FormationSpecific” where possible. This function is then called during the “UpdateFormation” function where required depending on if the formation is in combat.

### Implementation 1: Blind Formation

The primary stage in this implementation’s movement is to carry out a series of logic which will determine how the lead unit behaves (See Appendix 4, line 395) while the rest simply follow the lead unit as stated by Mars and Chanut (2015, p.219). For this, the formation first carries out a function which checks for any obstacles which may be in front of each unit (See Appendix 4, line 432). This involves performing a raycast from front of each ship in formation heading towards the ship’s forwards direction for a set distance. If the ray hits another game object the function uses the objects tag to identify the obstacle and then acts depending on the tag.

If the obstacle is the player ship the function checks the distance between the formation and the player, when the distance is less than a set amount then the player is too close to the formation and the lead ship will calculate a point facing away from the player and will begin to turn towards it (See Appendix 4, line 474). During this behaviour, the formation also sets a bool referred to as “CanSpotPlayer” to false, which forces the formation to fail all sight checks on the player regardless of if sight is established. A co-routine called after this sets the bool back to true, thus allowing the formation to spot the player again after 10 seconds. This prevents the formation from turning to avoid the player and immediately regaining sight, which leads to formations constantly circling the player after the first line of sight is made (See Appendix 4, line 641). When the distance is greater than the avoidance distance and less than the formations pre-set attack range, each ship in the formation will run its own attack function, thus allowing them to fire projectiles towards the player regardless of their position in the formation (See Appendix 4, line 486). If neither of these conditions are met then the formation simply continues on its current path, aiming to reduce the distance to the player and allow the ships to attack.

If the object identified is not the player then the formation uses the same calculation as when avoiding the player to turn the lead unit away from the obstacle and marks the formation as “avoiding” (See Appendix 4, line 497). While marked as “avoiding” the lead unit will continue to turn to the desired rotation until it has been reached (See Appendix 4, line 509), thus allowing the turn to continue even after the obstacle is not directly in front of the formation but may still collide with the edge of a unit. Once the desired turn has been completed the formation is set to no longer be marked as avoiding and will continue its usual behaviour.

If no obstacle is found, and the formation is not currently avoiding, the next function checks for a line of sight on the player through another raycast for a set “sight” distance (See Appendix 4, line 529). If the ray hits an object the function checks if the hit objects tag identifies it as the player and if successful, will turn the lead unit towards the player ship (See Appendix 4, line 553).

If neither of the previous conditions are true then the lead unit will perform the random “wander” movement mentioned previously. Thus allowing the formation to continue to move around the playable space until another condition is met (See Appendix 2, line 365).

Finally a RigidBody force is applied to the lead unit in order to move the unit in the desired direction (See Appendix 4, line 395). The forces direction is created from the units forward direction multiplied by a slightly reduced formation maximum move speed, which allows for other units to use the full move speed to catch up when falling behind after turns.

With the lead unit’s behaviour complete, the next stage of the formation’s implementation is to update the positions of each slot in the formation (See Appendix 4, line 560). To update each individual slot the function uses a list of each slots offset relative to the lead unit which is stored when the formation is generated, with each offset relating to the slot in the same index of the stored slots list. These offsets are added to the lead unit’s current position and then rotated around the lead position using the lead unit’s rotation (See Appendix 4, line 595). This results in each slot being moved to the new correct position based on its required offset and the rotation of the lead unit, thus creating the idea aimed for by Mars and Chanut (2015, p.219) that all other units are “blind” and simply following the lead unit.

Finally the formation moves each unit to its assigned slot based on its index in the formations list of stored units, with each index relating to the slot in the same index of the slot positions list (See Appendix 4, line 600). First each unit is rotated to face its target slot position, pointing the unit at its desired target. A calculation is then completed to determine how much of the distance between the ship and its slot can be covered in the current physics update with the formations maximum speed. If the distance required is greater than the coverable distance, a force is added with the direction being the ships forward vector multiplied by the maximum speed of the formation. This allows any ship that cannot reach its target slot to move faster than the lead unit and catch up to its assigned slot. If the distance required is less than the coverable distance a second calculation determines the exact force required to move the ship to its assigned slot, thus preventing ships from overshooting their target and maintaining the formation.

Overall the implementation meets the description and functionality set out by Mars and Chanut (2015, p.219). While the units within generated formations are able to detect obstacles in the path, they are unable to react to the obstruction and instead pass the information to the lead unit while the other units move in reference to the lead. In addition units within the slot use the same velocity correction principle suggested in order to prevent motion jolts.

### Implementation 2: Autonomous Behaviour

Much of the structure for the “Autonomous formation following” remains the same as in the “blind” formation implementation, with the key differences between the two being how the formations overall logic behaves and how the units within the formation responds to obstacles in their path.

With the main addition of this implementation being to allow units to respond to their own obstacles, it became necessary to create a custom class structure which could hold key information about each fighter in the formation and allow them to behave semi-independently. For this purpose, a custom class referred to as “FighterStruct” (See Appendix 5) is created and stored in a list for every ship in the formation during generation (See Appendix 6). This class contains a reference to the ship GameObject it is linked to, a Boolean which designates if the ship is currently avoiding an obstacle, a Boolean which designates if the ship is returning to face its slot after avoiding and a Quaternion to store the ships specific avoidance rotation for comparison while avoiding an obstacle.

In the modified “FormationSpecific” function (See Appendix 7, line 436) the newly created list of “FighterStruct”s is used to loop through the active ships in a formation, along modified versions of the “CheckForObstacle” (See Appendix 7, line 486) and “IdentifyObstacle” (See Appendix 7, line 514) functions, to check if there is an obstacle in the path of each fighter. If the “CheckForObstacle” function hits an object which the “IdentifyObstacle” function finds to be an avoidable object, the current “FighterStruct” is marked as avoiding an obstacle and the same calculation as the “blind” implementation sets the structs avoid rotation. While marked as “avoiding” ships will no longer rotate towards their assigned slot during the “MoveUnitsToSlots” function (See Appendix 7, line 707) and will instead rotate towards the assigned avoidance rotation until this rotation is met. This leads to the desired implementation set out by Mars and Chanut (2015, p.220) as units become able to avoid their own obstacle, while still following a lead unit if needed. Furthermore, if the current struct is linked to the lead unit of the formation, a second Boolean is set to true which marks the formations lead unit as avoiding and as such locks out the “CheckForSight” and “RandomMovement” behaviours which would otherwise be run in the “FormationSpecific” function (See Appendix 7, line 436). Doing so prevents the lead unit’s rotation being overwritten by these functions after all fighters have been scanned for obstacles as it could cause the lead unit to continue into an obstacle which it should be avoiding.

Once a ship has finished is marked avoidance turn the ship is set to no longer be avoiding and is instead marked as “returning” and as such will run the “ReturnToForm” function (See Appendix 7, line 619) in the new “FormationSpecific” loop. Similarly to avoiding an obstacle, within this function a calculation to find the rotation needed to face its slot’s current location is completed in the same way as calculating an avoidance rotation, the fighter is then rotated to face this point and will continue to be until the rotation has been complete. While marked as returning ships will also not turn towards the specific slot position during the “MoveUnitsToSlots” Function (See Appendix 7, line 707) as it could overwrite the rotation needed to avoid the obstacle in the ships path.

In addition to these modifications the “FormationSpecific” (See Appendix 7, line 436) changes to obstacle avoidance, a further change has been made to the section which establishes if the formation has a line of sight on the player. In the modified implementation, a line of sight can only be established when lead unit is not avoiding obstacles in order to avoid overwriting an avoidance rotation with one that would lead to collision.

With the addition of the new “FighterStruct” to the implementation, a small change to the “RemoveFighter” function (See Appendix 8) was also required. The change ensures that when a fighter is defeated the “FighterStruct” it links to is also removed from the list of structs in the formation. In a similar way the “AddFighter” function (See Appendix 9) was also modified to create and add a “FighterStruct” to this list when adding a fighter from another formation.

As suggested by Mars and Chanut (2015, p.220) this implementation expands upon the base provided by the “blind” formation and gives each unit within the formation a semi-independence from the lead unit. While the implementation created mainly aims to give units their own collision avoidance capabilities rather than relying on the lead unit, it can be easily modified further to allow units to have individual goals or behaviours stored in the “FighterStruct” class to further add to “personality” of each ship.

### Implementation 3: Remove the Leader

The previously created “Autonomous formation” served as a useful basis for the “Remove the Leader” implementation, with both formation types allowing units to handle collision avoidance and have their own sub-goals. The primary difference between the two is that while slots within an “Autonomous formation” are updated based on the “lead” units’ position, slots within a “Remove the Leader” implementation update based on an invisible leader which does not collide or avoid obstacles.

The first step in converting to a “Remove the leader” format was to change the formations code to use the empty game object which holds the “Formation Base” script as “invisible leader” instead of a designated lead unit. This involved adding a RigidBody to the empty game object so the same force that was previously applied to the lead unit could instead be applied to the empty object, thus allowing it to move at the correct speed and direction. In addition, the “TurnToPlayer” (See Appendix 10, line 601) and “RandomMovement” (See Appendix 11) functions were also changed to adjust the rotation of the empty game object rather than the designated lead unit of the formation in order to completely replace the lead unit features with the “invisible leader” and allow all units within the formation to act independently.

With the formation now updating based on the “invisible leader” the “UpdateSlotPositions” function (See Appendix 10, line 608) also had to be changed to use the “invisible leader’s” position rather than the lead unit. Additionally, changes also had to be made to the “MoveUnitsToSlots” function (See Appendix 10, line 649), which changed the position that the slots use as a basis to add to their offset and update the relevant slot. This is in order to allow for the previously lead unit to behave as standard unit within the formation.

Finally several functions, including the “GenerateFormation” (See Appendix 2, line 92), “FormationSpecific” (See Appendix 10, line 409) and “RemoveFighter” (See Appendix 2, line 216) functions, were changed in order to remove references to the lead unit and its related variables, where they have not been replaced with references to the “invisible leader”, as they were no longer required for the implementation to function.

The implementation created adheres to the principals laid out by Millington (2009, p.148- 149) as it “removes any responsibility for guiding the formation from the leader” and does not require any obstacle avoidance, instead leaving it to the fighters within the formation to handle individually.

## Conclusion

To summarise, each implementation fulfils the requirements set out by the research conducted and has led to different behavioural outcomes from the overall formations. Each implementation builds upon the previous rather than completely rewriting the code for each. This is in order to ensure that any changes to performance or player experience found from the tests in the following chapter are from the key aspects of the implementation and not due to changes made to generic features.

# Chapter 4: Testing and Analysis

## Introduction

Each implementation has undergone a series of tests in order to determine which is the most effective, as well as to identify specific areas where one implementation may be better than others. These tests can be split into two distinct categories: “performance” and “experience” testing.

## Performance Testing

The “performance” tests carried out used the “Unity Profiler” (Unity Technologies, no date) to check the time taken to complete a frame update with each implementation during key actions. The key actions which were tested during this process are: turning during random movement, obstacle avoidance which affects all ships in the formation and obstacle avoidance which affects some ships. While the implementation for finding a location to wander towards was the same across different implementations, the way in which the implementation handled reaching this point was likely to yield different results. These actions were chosen as they are key actions which change between implementations and are logic intensive, meaning they are more likely to cause a drop in performance. The tests were completed in a controlled environment rather than inside the main game level to avoid any outside interference on the average frame count, such as calculating the players movement or rendering other objects within the scene. This controlled environment contained a spawn gate for the testing formation to use and a single plane object for tests which required an obstacle, formations were limited to containing 5 ships in order to maintain variables during the tests. In addition, where obstacles were required they were always placed in the same position in the scene and set to the same scale across each implementations test.

The first test carried out measured the average time taken to complete a frame update for the first 30 seconds of the formation’s movement through the test scene with no obstacle present. The expected outcome from this test would be that, as the simplest implementation, the “blind” implementation would have a shorted update time than “autonomous” and “remove the leader”, with the latter taking the longest due to its complexity. After the test was completed, it was found that while the “blind” formation had the shortest update time, as expected, the “autonomous” implementation had the longest frame update time. However, the difference between all the updates times was less than 1ms (See Appendix 11), meaning that while the “blind” implementation was faster it would not be noticeable to players.

The second test measured the time to complete a frame update while the formation responds to an obstacle which would otherwise collide with all ships. This involved placing a flat plane object in front of the formation as it spawned and measuring the frame update time from the frame the formation started avoiding the obstacle, to the frame it finished avoiding the obstacle. Similarly to the previous test, it was expected that the “blind” formation would update the fastest, as it is designed to calculate the turn required for the lead ship to avoid the others and none of the other ships. It was also expected that the “remove the leader” implementation would update slowest, as it calculates the turns required for each ship as an individual. The test results showed that all implementations completed the turns with the same average update time of 4ms (See Appendix 11), showing small variations and fluctuations which averaged out to the same value.

The final test measured the time taken to complete a frame while the formation responds to an obstacle which would only collide with a single ship. This involved moving the plane mentioned previously to a point where only one ship would detect it as an obstacle and measuring the average frame update time from the frame the formation started avoiding the obstacle, to the frame it finished avoiding the obstacle. The expected results for this test were that all implementations would update at a similar speed, due to each requiring to only calculate the turn for a single ship, with “blind” only calculating the turn for the lead ship, while “autonomous” and “remove the leader” only calculate the turn for the affected ship. Similarly to the first test, it was found that the “blind” implementation updated with the fastest time while “autonomous” updated the slowest. However the difference between the times was again less than 1ms (See Appendix 11), making it unnoticeable to players.

Overall, while tests revealed each implementations small impact on the performance of the game, these impacts are unnoticeable to players and irrelevant within the scope of a larger game.

## Experience Testing

The “experience” testing section aimed to evaluate how each implementation affects the players overall experience while playing the game. This was measured through two sets of tests, the first of which was a set of videos showcasing a formation undertaking the same actions as those in the performance tests. This test aimed to allow players to evaluate the visual appeal of each implementation completing the manoeuvre without the distractions that come with a regular gameplay experience. The second section had players complete a 10 min gameplay session in order to further evaluate how each implementation impacts the performance in regard to the wider gameplay experience. During most of the question’s testers were asked to consider not only the visual appeal of the formation’s movement but also how efficiently they complete any turns and if the movement seems like that of a real formation.

The first aspect the questionnaire (See Appendix 12) aimed to determine was the overall impact each implementation had on the formations visual appeal to players. This was an important factor to research as a formation which performs well but does not look visually appealing can reduce a player’s enjoyment of the game as enemies do not appear to move in realistic ways. The expected result of these questions would be that the most complex formation, in this case “remove the leader” would be most popular as it has the most logic designed to make the units act in interesting ways. It was also predicted that the “blind” formation would be least popular as it is not only the simplest implementation, but also creates partially unrealistic formations as the ships within have no independent actions aside from the lead ship. The questionnaire results found that testers enjoyed the “remove the leader” implementation the most. The most common reason for this was that testers enjoyed the visual variety that occurred when avoiding obstacles and how the units within the formation reformed afterwards. In addition, the questionnaire also found that testers enjoyed the “blind” implementation the least, with the most common reasons being that the formations did not falter from their formations and as such looked boring or dull to testers.

The second aspect of the questionnaire (See Appendix 12) was designed to determine which implementation was most visually appealing specifically during the formations regular wandering” manoeuvres, asking testers to select a favourite and least favourite based only on their view of the formation’s wandering behaviour through the provided videos and gameplay build. The purpose of narrowing down the criteria was to discover if the results would differ from the overall visual favourite and least favourite, which would inform on if any implementation were more visually appealing than the others in this specific area alone. In contrast to the previous prediction, it was expected that the “blind” implementation would be the favourite for general movement, while “remove the leader” was expected to be the worse. This was again due to the complexity of each implementation, however in this case the complexity of “remove the leader” was expected to make general movement less fluid, while the simplicity of the “blind” implementation was expected to produce more fluid turns. Results from the questionnaire found that testers enjoyed the wandering behaviour created by the “blind” implementation, the most common reason for this was that testers enjoyed the overall smoothness of the unit’s movements while turning as a complete unit. In contrast players least enjoyed the results produced by the “autonomous” implementation as many found issues with ships wobbling during the turning process, as ships attempted to maintain position.

Next testers were asked to consider how each implementation handles obstacle avoidance. Similarly to the previous question set, these questions (See Appendix 12) aimed to determine if the results would differ from the overall favourite and least to find if players preferred a different implementations method of avoiding obstacles. For this, testers were again asked to choose a favourite and least favourite implementation using the previously mentioned videos and gameplay build. For reasons similar to the first question set, it was expected that the “remove the leader” implementation would be the most popular due to its complexity allowing for a wide range of responses to obstacles blocking the formation. It was also predicted that the “autonomous” implementation would be least popular when avoiding obstacles as it contains aspects of both “blind” and “remove the leader” which may cause issues with more complex manoeuvres. Testing results showed that testers enjoyed how the “remove the leader” implementation handled obstacles avoidance, with the most common reason being how the formation is able to scatter and reform effectively. Testing also showed that testers enjoyed the “autonomous” implementation the least, as many found issues with ships turning to avoid obstacles and colliding with the rest of the formation as they attempt to do so.

Finally testers were asked if they encountered any performance issues during the gameplay build, however none of the testers reported any issues.

To conclude, the “experience” testing found that overall most tester enjoyed the “remove the leader” implementation, noting how each ship acting as an individual under most situations rather than a block unit was more enjoyable. In addition, testing showed that players enjoyed the “autonomous” implementation least, mentioning how units within these formations would sometimes fail to consider the rest of the formation when avoiding obstacles or moving.

## Conclusion

In summary, the performance tests completed found small performance impacts which would be unnoticeable by players, while the “experience” tests found that players found the “remove the leader” implementation more appealing. This reveals that when selecting a method of formation movement to implement, focus should be placed on how the implementation will impact the players experience of the game, as performance issues can be improved upon during the building process.

# Chapter 5: Conclusion

## Introduction

This chapter will bring together overall results of the research conducted, discussing if the aims of the research have been met and what results the research has found, in order to fully summarise the papers goals and findings. In addition, the chapter will also cover what the author has learned by completing this research and how it could be improved in the future.

## Were the research aims met?

The aims of the research have been successfully met as the research conducted enabled the creation and testing of different methods of formation movement, culminating in one of the created implementations being deemed most effective and one being deemed least effective. The implementations created accurately represent the chosen methods outlined, which then allowed for the tests to accurately determine which is most and least effective based on both the performance of each implementation and player opinion.

The first key aim of this research was to compile existing methods for formation movement and explore how each is used to create a different variation of formation movement. The literature review outlined a variety of existing methods, each with their own level of complexity and expected results. In addition, the chapter also explored how steering behaviours, can be used to aid or alter the results of each implementation. Furthermore the existing methods were also explored to discover how they could be adapted into formation movement for aerial AI rather than ground based.

Exploring these methods then formed the basis of the second research aim, which was to build a selection of existing methods, using those which present the most potential for yielding differing behaviours from the formation and as such, different results when tested. From the methods explored, 3 differing methods were chosen and adapted for formation movement within an existing project. A framework was created to build each implementation within in order to easily allow for some features, which were key to the behaviour of the AI but not key to the formation’s implementation, to remain the same. This included behaviours such as how the formation establishes a line of sight on the player or how the formation responds to losing a unit, in order to minimise any external influence on the results of the tests which followed.

A series of tests were then carried out on each implementation which set out to determine which is most and least effective in the overall performance and which is most and least effective in a player’s experience of the overall game. The results from these tests were used to discover which implementation was able to maintain the highest average framerate and which had the lowest, thus revealing any performance impacts generated by each implementation. The player experience tests used a set of videos, where the implementations completed the same scenarios as the performance testing, and a playable build of each implementation in order to find which implementation players enjoyed the visuals and gameplay aspects of. The findings from both sets of tests were then combined to determine which method is most and least effective overall. As covered previously, it was discovered that none of the three implementations created generated any noticeable difference in the game’s overall performance (See Appendix 11) and players preferred the “remove the leader” implementation most and were unsatisfied by the “autonomous” implementation. The lack of difference in performance found during testing was a surprising result, as it had been predicted that the more complex behaviours of “autonomous” and “remove the leader” would cause a decrease in performance due to the extra logic required to achieve the implementations specific behaviours. In addition the overall least effective method was not as predicted, as it had been predicted that testers would enjoy the simplest “blind” implementation the least and as such it would likely be the least effective when combined with the results of the performance testing. This differed from the final result, which showed the “autonomous” behaviour to be the least popular amongst tester and indifferent to performance. However, the overall testing result for the most effective implementation concluded as expected, with the most complex behaviour of “remove the leader” being found as the most effective.

## What have I learned from the research?

### What went well?

I believe that during the research undertaken to find existing methods of formation movement I was able to find a wide range of existing implementations and explore how the logic behind each functions in order to create the behaviours desired. This has not only served as useful research for this paper but would also allow for the creation and testing of additional implementations using the breakdowns of logic covered in the literature review.

In addition, the implementations which were chosen to be tested have been adapted well from focusing on ground-based unit movement to aerial unit movement. When creating each implementation, the principals of the formation’s movement had to be adapted to account for the change between ground units which are affected by gravity to aerial units which are unaffected by gravity and have freedom of movement across all axis. The methods used to do so could also be applied to other ground-based movement techniques in future research.

Furthermore, the implementations created were successful in matching the descriptions outlined by their respective core texts. From the methods created, each used the behaviours outlined correctly and achieved the varying results expected from each. For example, the “blind” implementation was successful at ensuring that only the lead unit responded to external factors which would affect the formation and all other ships only updated their position within relation to the lead ship, regardless of any external factors that should be influencing them.

### What could be changed in the future?

In future research I would like to create and test more of the implementations explored in the literature review, in order to further determine which method is most effective. In addition to methods found in this research, it would also be possible to explore other methods of formation movement which may have been missed or developed in the time since this research in order to allow for a wider range of comparisons. This would allow for further comparison between implementations created and the others found in this research in order to determine if there is a more effective method than those already explored.

In addition to testing more methods for formation movement, it would also be beneficial to add more complex steering behaviours, such as those covered in the literature review, in order to create an even wider variety results from the formations behaviour and as such, a wider range of possible outcomes from testing. These behaviours could serve as interesting additions to each implementation and could explore how steering behaviours affect different implementations, comparing the resulting behaviour and test results to determine if these additions increase an implementations effectiveness.

Finally in future research I would aim to add further polish to the implementations created during this research, making key improvements such as allowing units to consider the positions of others within the formation when turning to avoid obstacles or allowing a formation to accurately detect when all ships are blocked by an obstacle at different ranges. This would create the potential for different results from the experience testing as implementations would be more refined and as such less likely to contain small issues which were present in this research. These changes could then also be included in any additional implementations created during future research, allowing for those implementations to also not contain the same issues and enable a more balanced comparison.

# References

Gary, D. (2020). *Formation Flying* Avaliable at: <https://www.britannica.com/technology/formation-flying> (Accessed: 17th November 2020)

Dice (2017) *Star Wars Battlefront 2* [Video Game], EA, Available at: <https://www.ea.com/games/starwars/battlefront/star-wars-battlefront-2> (Accessed: 17th November 2020)

Creative Assembly (2017) *Total War: Warhammer 2* [Video Game], Sega, Available at: <https://www.totalwar.com/games/warhammer-ii/> (Accessed 27th November 2020)

Dawson, C. (2002) ‘Formations’, in Rabin, S. (ed.) *AI Game Programming Wisdom*, Hingham: Charles River Media. p.273- 281

Mars, C. Chanut, J. (2015) ‘Hierachial Architecture for Group Navigation Behaviours’. In Rabin, S. (ed.) *Game AI Pro 2*. Boca Raton: CRC Press, Taylor & Francis Group. p.209- 222

Millington, I. and Funge, J.D. (2009*) Artificial intelligence for games*. 2nd edn. San Francisco, Calif.; Oxford: Morgan Kaufmann; Elsevier Science. p.148- 149

Pottinger, C. (February, 1999) ‘Coordinated Unit Movement’. In Miller, P. *Game Developer Magazine*. San Francisco: UBM Tech. p.27-35

Millington, I. and Funge, J.D. (2009*) Artificial intelligence for games*. 2nd edn. San Francisco, Calif.; Oxford: Morgan Kaufmann; Elsevier Science. p.55- 95

Seemann, S. and Bourg, M. (2004) *AI For Game Developers*. O'Reilly Media, Inc. p.7- 9

Bjore, S. (2014) ‘Techniques for Formation Movement Using Steering Circles’ in Rabin, S (ed.) *Game AI Pro 1.* Boca Raton, Florida: CRC Press. p.289- 295

Reynolds, C. (1999) *Steering Behaviors For Autonomous Characters*. p.11

Buckland, M. (2004) *Programming Game AI By Example* p.126- 129

Unity Technologies (no date). *MonoBehaviour.FixedUpdate()*. Available at: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html> (Accessed: 22/04/21)

Unity Technologies. (no date) *Profiler Overview*. Accessed at: <https://docs.unity3d.com/Manual/Profiler.html> (Accessed: 22/04/21)

# Appendix:

## Appendix 1

//Calls on Fixed Update due to Update Formation using RigidBody Physics

void FixedUpdate()

{

//Check for missing formations

if (SpawnedFormations.Count < MaxFormationNumber)

{

AddFormation();

}

//Call update for each formation

foreach(GameObject Formation in SpawnedFormations)

{

Formation.GetComponent<FormationBase>().UpdateFormation();

}

}

## Appendix 2

005: public class FormationBase : MonoBehaviour

006: {

007: [Header("Setup Variables")]

008: public GameObject UnitPrefab;

009: public int UnitTotalCount;

010: private GameObject[] SpawnPoints;

011: private bool FlippedGate;

012:

013: [Header("Formation Variables")]

014: public int ActiveUnitCount;

015: public Vector3 SlotOffset;

016: public bool InCombat;

017: public List<Vector3> SlotPositions;

018: private List<Vector3> SlotOffsets;

019: private List<GameObject> FightersInFormation;

020: private FormationManager Manager;

021: public float ReformTime;

022: public float ReformDelay;

023: private Quaternion LeadRotation;

024: private Vector3 LeadPosition;

025: private FriendlyMarker ThisHUDMarker;

026:

027: [Header("Movement Variables")]

028: //General Movement

029: private float RotationSpeed;

030: private float MoveSpeed;

031: //Random Movement

032: public Vector3 RandomTarget;

033: private float MinDirChange;

034: private float MaxDirChange;

035: private float NextChangeTime;

036:

037: [Header("Collision Variables")]

038: public float CollisionRange;

039: public float PlayerEngageRange;

040: public float PlayerAvoidRange;

041: private Quaternion AvoidRotation;

042: private bool CurrentlyAvoiding;

043:

044: [Header("Sight Variables")]

045: public float SightRange;

046: private GameObject PlayerOBJ;

047: private bool CanSpotPlayer;

048:

049: [Header("Used in some implementations")]

050: public GameObject LeadUnit;

051: public Rigidbody LeadBody;

052:

053: public AverageFrames TestScript;

054:

055: //----------------------------------------------- Formation Generation Functions ----------------------

056:

057: private void Start()

058: {

059: CanSpotPlayer = true;

060: //Get all possible spawn points

061: SpawnPoints = GameObject.FindGameObjectsWithTag("ThaalianJumpgate");

062: //Set Delay to reform after damage

063: ReformDelay = 60f;

064: //Get Formation manager from parent object

065: Manager = this.transform.parent.GetComponent<FormationManager>();

066: //Create empty list to store slots

067: SlotPositions = new List<Vector3>();

068: SlotOffsets = new List<Vector3>();

069: //Create empty list to store fighters

070: FightersInFormation = new List<GameObject>();

071: //Get Player OBJ

072: PlayerOBJ = GameObject.FindGameObjectWithTag("Player");

073: //Set Movement Variables

074: RotationSpeed = 40f;

075: MoveSpeed = 10f;

076: MinDirChange = 10f;

077: MaxDirChange = 45f;

078: RandomTarget = new Vector3(0, 0, 0);

079: NextChangeTime = Time.time + Random.Range(5f, 10f);

080: AvoidRotation = new Quaternion();

081:

082: //Call Formation Generation

083: GenerateFormation();

084: this.transform.position += this.transform.forward \* 2;

085:

086: RandomTarget = this.transform.position + (this.transform.forward \* 400);

087: ThisHUDMarker = this.GetComponent<FriendlyMarker>();

088:

089: TestScript = GameObject.FindGameObjectWithTag("MissionControl").GetComponent<AverageFrames>();

090: }

091:

092: private void GenerateFormation()

093: {

094: //Move Formation to randomly selected Spawn Gate

095: Transform LeadTransform = ChooseNewGate();

096: //Make copy of transform and rotation

097: LeadPosition = LeadTransform.position;

098: LeadRotation = LeadTransform.rotation;

099: //Set this position and rotation

100: this.transform.position = LeadPosition;

101: this.transform.rotation = LeadRotation;

102:

103: //Bool for Reverse Gates (prevents formations spawning backwards on half gates

104: FlippedGate = false;

105:

106: //Add LeadPosition

107: SlotPositions.Add(LeadPosition);

108: SlotOffsets.Add(new Vector3(0, 0, 0));

109:

110: //Rotate formation depending on spawn gate

111: if (LeadTransform.rotation.eulerAngles.y == 180)

112: {

113: LeadRotation.eulerAngles = new Vector3(0, 180f, 0);

114: FlippedGate = true;

115: }

116: else

117: {

118: LeadRotation.eulerAngles = new Vector3(0, 0f, 0);

119: FlippedGate = false;

120: }

121:

122: //Spawn Lead fighter

123: LeadUnit = Instantiate(UnitPrefab, LeadPosition, LeadRotation);

124: LeadBody = LeadUnit.GetComponentInChildren<Rigidbody>();

125: LeadUnit.GetComponentInChildren<FighterHealth>().ThisFormationBase = this;

126: //Add Fighter to list

127: FightersInFormation.Add(LeadUnit);

128:

129: //For each fighter to add minus 1 for lead unit

130: for (int i = 0; i < UnitTotalCount - 1; i++)

131: {

132: Vector3 NewSlotPosition = new Vector3();

133: //Calculate Slot position for offset

134: if (i < 2)

135: {

136: //Check if gate is flipped

137: if (FlippedGate)

138: {

139: NewSlotPosition = LeadPosition - SlotOffset;

140: }

141:

142: else

143: {

144: NewSlotPosition = LeadPosition + SlotOffset;

145: }

146: }

147: else

148: {

149: //Check if gate is flipped

150: if (FlippedGate)

151: {

152: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] - SlotOffset;

153: }

154:

155: else

156: {

157: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] + SlotOffset;

158: }

159: }

160:

161: SlotOffset.x \*= -1;

162: //Add slot to list

163: SlotPositions.Add(NewSlotPosition);

164: //Store offset in list (Useful for updating slot position while in motion

165: Vector3 Newoffset = this.transform.position - NewSlotPosition;

166: SlotOffsets.Add(Newoffset);

167: //Spawn Fighter at position

168: GameObject SpawnedFighter = Instantiate(UnitPrefab, NewSlotPosition, LeadRotation);

169: //Give fighter reference to this formation

170: SpawnedFighter.GetComponentInChildren<FighterHealth>().ThisFormationBase = this;

171: //Add Fighter to list

172: FightersInFormation.Add(SpawnedFighter);

173: }

174: //Set Unit Count

175: ActiveUnitCount = UnitTotalCount;

176:

177: }

178:

179:

180: public void UpdateFormation()

181: {

182: //Check if formation is currently in combat

183: if (InCombat)

184: {

185: //Check if enough time has passed to start reforming

186: if(Time.time >= ReformTime)

187: {

188: //Set to no longer in combat so formation will activate next update

189: InCombat = false;

190: ThisHUDMarker.MyMarker.enabled = true;

191: for (int i = 0; i < FightersInFormation.Count; i++)

192: {

193: GameObject Fighter = FightersInFormation[i];

194: Fighter.GetComponentInChildren<FriendlyMarker>().MarkerActive = false;

195: }

196: }

197:

198: //Else update each ship with it's solo behaviour

199: else

200: {

201: foreach(GameObject Fighter in FightersInFormation)

202: {

203: Fighter.transform.GetChild(1).GetComponent<FighterMovement>().SoloMovement();

204: }

205: }

206: }

207:

208: //Else Update Position as

209: else

210: {

211: FormationSpecific();

212: }

213: }

214:

215:

216: public void RemoveFighter(GameObject ToRemove)

217: {

218: //Check if lead unit needs replacing

219: if(ToRemove == LeadUnit)

220: {

221: //Lead unit is always index 0, so set new lead to index 1

222: LeadUnit = FightersInFormation[1];

223: }

224: //Remove Fighter from list

225: //All units in slots above removed ship will then move down a slot

226: FightersInFormation.Remove(ToRemove);

227: //Decrease number of active fighters

228: ActiveUnitCount -= 1;

229: //Remove last slot in formation as it is now unoccupied

230: SlotPositions.RemoveAt(SlotPositions.Count - 1);

231: SlotOffsets.RemoveAt(SlotOffsets.Count - 1);

232:

233: //Check if there is only 1 unit remaining

234: if (ActiveUnitCount == 1)

235: {

236: //Move fighter to closest formation and destroy this formation

237: MoveFighterToOtherForm();

238: }

239: //Else enable individual ship markers

240: else

241: {

242: for (int i = 0; i < FightersInFormation.Count; i++)

243: {

244: GameObject Fighter = FightersInFormation[i];

245: Fighter.GetComponentInChildren<FriendlyMarker>().MarkerActive = true;

246: }

247: ThisHUDMarker.MarkerActive = false;

248: }

249: Destroy(ToRemove);

250: }

251:

252: private void MoveFighterToOtherForm()

253: {

254: Debug.Log("Consolidating Ship to nearest");

255: //Locate Nearest formation

256: float NearestDistance = float.MaxValue;

257: GameObject NearestFormation = null;

258: foreach (GameObject OtherFormation in Manager.SpawnedFormations)

259: {

260: if (OtherFormation != this)

261: {

262: if (Vector3.Distance(OtherFormation.transform.position, this.transform.position) < NearestDistance)

263: {

264: //Set new nearest distance

265: NearestDistance = Vector3.Distance(OtherFormation.transform.position, this.transform.position);

266: //Set nearest game object

267: NearestFormation = OtherFormation;

268: }

269: }

270: }

271:

272: //Check Nearest formation isn't somehow == null

273: if (NearestDistance != float.MaxValue)

274: {

275: //Add fighter into formation

276: FormationBase FoundFormation = NearestFormation.GetComponent<FormationBase>();

277: FoundFormation.AddFighter(FightersInFormation[0]);

278: }

279:

280: //Otherwise allow fighter to fly solo

281: else

282: {

283: FightersInFormation[0].transform.SetParent(null);

284: FightersInFormation[0].transform.GetChild(1).GetComponent<FighterMovement>().enabled = true;

285: }

286:

287: //Destroy formation

288: Manager.SpawnedFormations.Remove(this.gameObject);

289: Destroy(this.gameObject);

290: }

291:

292: public void AddFighter(GameObject ToAdd)

293: {

294: //Use similar code as formation generation for new slot position

295: Vector3 NewSlotPosition = new Vector3();

296: //Check if formation has more than 3 fighters

297: if (ActiveUnitCount < 3)

298: {

299: //Check if gate is flipped

300: if (FlippedGate)

301: {

302: NewSlotPosition = SlotPositions[0] - SlotOffset;

303: }

304:

305: else

306: {

307: NewSlotPosition = SlotPositions[0] + SlotOffset;

308: }

309: }

310: else

311: {

312: //Check if gate is flipped

313: if (FlippedGate)

314: {

315: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] - SlotOffset;

316: }

317:

318: else

319: {

320: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] + SlotOffset;

321: }

322: }

323:

324: SlotPositions.Add(NewSlotPosition);

325: SlotOffset.x \*= -1;

326: //Store offset in list (Useful for updating slot position while in motion

327: Vector3 Newoffset = this.transform.position - NewSlotPosition;

328: SlotOffsets.Add(Newoffset);

329: //Add fighter to list of formation units

330: FightersInFormation.Add(ToAdd);

331: //Increase number of fighters in formation

332: ActiveUnitCount += 1;

333: }

334:

335: private Transform ChooseNewGate()

336: {

337: bool AcceptableGate = false;

338: GameObject SelectedGate = null;

339: while (!AcceptableGate)

340: {

341: int RandomIndex = Random.Range(0, SpawnPoints.Length - 1);

342: SelectedGate = SpawnPoints[RandomIndex];

343: SpawnGateBase SelectedBase = SelectedGate.GetComponent<SpawnGateBase>();

344: if (SelectedBase.GateTaken == false)

345: {

346: SelectedBase.GateTaken = true;

347: AcceptableGate = true;

348: StartCoroutine(SelectedBase.TakenDelay());

349: }

350: }

351: MeshRenderer GateCollider = SelectedGate.transform.GetChild(1).GetComponent<MeshRenderer>();

352: StartCoroutine(EffectDisable(GateCollider));

353: return SelectedGate.transform;

354: }

355:

356: private IEnumerator EffectDisable(MeshRenderer Chosengate)

357: {

358: Chosengate.enabled = true;

359:

360: yield return new WaitForSeconds(5);

361:

362: Chosengate.enabled = false;

363: }

364:

365: private void RandomMovement()

366: {

367: //Check if time to choose random movement point

368: if(Time.time >= NextChangeTime || Vector3.Distance(LeadUnit.transform.position, RandomTarget) <= 10)

369: {

370: //Update Change Time

371: NextChangeTime = Time.time + Random.Range(MinDirChange, MaxDirChange);

372: //Update Random Position

373: RandomPosition();

374: }

375: Vector3 VectToTarget = RandomTarget - LeadUnit.transform.position;

376: if (Vector3.Angle(LeadUnit.transform.forward, VectToTarget) != 0)

377: {

378: var q = Quaternion.LookRotation(VectToTarget);

379: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, q, RotationSpeed \* Time.deltaTime);

380: }

381: }

382:

383: private void RandomPosition()

384: {

385: //Get Random point somewhere in bounds of level

386: RandomTarget.x = Random.Range(-750, 750);

387: RandomTarget.y = Random.Range(-750, 750);

388: RandomTarget.z = Random.Range(-750, 750);

389: }

## Appendix 3

Diagram

Description automatically generated

## Appendix 4

392: public class FormationBase : MonoBehaviour

393: {

394: //See Appendix 2 for common functionality

395: private void FormationSpecific()

396: {

397: GameObject ObstacleOBJ;

398: Vector3 HitPoint;

399:

400: //Obstacle Detection

401: if (CheckForObstacle(out ObstacleOBJ, out HitPoint))

402: {

403: IdentifyObstacle(ObstacleOBJ, HitPoint);

404: }

405: //Else check if currently making turn to avoid obstacle

406: else if (CurrentlyAvoiding)

407: {

408: ContinueAvoiding();

409: }

410:

411: //Check for line of sight on player

412: else if (CheckForSight())

413: {

414: TurnToPlayer();

415: }

416:

417: //Random Movement

418: else

419: {

420: RandomMovement();

421: }

422:

423: LeadBody.AddForce(this.transform.forward \* (MoveSpeed - 1), ForceMode.Impulse);

424:

425: //Update Slot positions

426: UpdateSlotPositions();

427:

428: //Update Unit positions

429: MoveUnitsToSlots();

430: }

431:

432: private bool CheckForObstacle(out GameObject HitObstacle, out Vector3 HitPoint)

433: {

434: RaycastHit HitOut = new RaycastHit();

435:

436: //Raycast from each fighter

437: foreach(GameObject Ship in FightersInFormation)

438: {

439: //Get point in front of ship

440: Vector3 ShipFront = Ship.transform.position + Ship.transform.forward \* 5;

441: //Check for obstacle within set range

442: if (Physics.Raycast(ShipFront, Ship.transform.forward, out HitOut, CollisionRange))

443: {

444: //Check if not already avoiding or if obstacle is too close anyway and not another enemy fighter

445: if((CurrentlyAvoiding == false || HitOut.distance <= 75) && !HitOut.collider.transform.root.CompareTag("EnemyFighter"))

446: {

447: //Set HitObstacle Output to object hit

448: HitObstacle = HitOut.collider.gameObject;

449: HitPoint = HitOut.point;

450: //Return True when obstacle

451: return true;

452: }

453: }

454: }

455: //If no fighter returns a hit

456: HitObstacle = null;

457: HitPoint = Vector3.zero;

458: return false;

459: }

460:

461: private void IdentifyObstacle(GameObject ObstacleOBJ, Vector3 HitPoint)

462: {

463: //Get Obstacle Tag

464: string ObstacleTag = ObstacleOBJ.tag;

465: float DistanceToObstacle = Vector3.Distance(this.transform.position, ObstacleOBJ.transform.position);

466:

467: //Act accordingly based on tag

468: switch (ObstacleTag)

469: {

470: //If hit object is player

471: case "Player":

472: Debug.Log("Player is obstacle, check if in range to attack!");

473: //Check if Distance is too close

474: if (DistanceToObstacle <= PlayerAvoidRange)

475: {

476: CanSpotPlayer = false;

477: //Set Avoid Rotation

478: AvoidRotation = Quaternion.LookRotation((LeadUnit.transform.root.position - HitPoint) \* 0.5f);

479: //Set currently avoiding

480: CurrentlyAvoiding = true;

481: //Start rotation

482: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

483: StartCoroutine(SpotPlayerDelay());

484: }

485: //Else check if in range to engage

486: else if (DistanceToObstacle <= PlayerEngageRange)

487: {

488: foreach (GameObject Ship in FightersInFormation)

489: {

490: Ship.GetComponentInChildren<FighterFiring>().FireCannons(ObstacleOBJ.transform.position);

491: }

492: }

493: //Otherwise formation will continue to move towards Player

494: break;

495:

496: //If obstacle is other

497: default:

498: //Set Avoid Rotation

499: AvoidRotation = Quaternion.LookRotation((LeadUnit.transform.root.position - ObstacleOBJ.transform.position) \* 0.5f);

500: //Set currently avoiding

501: CurrentlyAvoiding = true;

502: //Start rotation

503: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

504:

505: break;

506: }

507: }

508:

509: private void ContinueAvoiding()

510: {

511: //Check if arrived at rotation

512: if (LeadUnit.transform.rotation == AvoidRotation)

513: {

514: //If yes, set currently avoiding to false

515: CurrentlyAvoiding = false;

516: //Update Change Time

517: NextChangeTime = Time.time + Random.Range(MinDirChange, MaxDirChange);

518:

519: //Update Random Position

520: RandomPosition();

521: }

522: else

523: {

524: //Continue to rotate towards

525: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

526: }

527: }

528:

529: private bool CheckForSight()

530: {

531: RaycastHit HitOut = new RaycastHit();

532: //Check each fighter for line of sight

533: foreach (GameObject Ship in FightersInFormation)

534: {

535: Vector3 ShipFront = Ship.transform.position + Ship.transform.forward \* 8;

536: Vector3 VectToPlayer = PlayerOBJ.transform.position - ShipFront;

537: //Raycast towards player

538: if (Physics.Raycast(ShipFront, VectToPlayer, out HitOut, SightRange) && CanSpotPlayer)

539: {

540: //Check if hit is player

541: if (HitOut.collider.gameObject.transform.root.CompareTag("Player"))

542: {

543: //Return True if line of sight made

544: return true;

545: }

546: }

547:

548: }

549: //Return false otherwise

550: return false;

551: }

552:

553: private void TurnToPlayer()

554: {

555: Quaternion PlayerRotation = Quaternion.LookRotation(PlayerOBJ.transform.position - LeadUnit.transform.position);

556: //Turn Formation towards Player

557: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, PlayerRotation, RotationSpeed \* Time.deltaTime);

558: }

559:

560: private void UpdateSlotPositions()

561: {

562: //Check slot positions exist

563: if(SlotPositions.Count != 0)

564: {

565: Vector3 SlotHolder = new Vector3();

566: //Update Lead Position

567: SlotPositions[0] = LeadUnit.transform.position;

568: this.transform.position = LeadUnit.transform.position;

569: this.transform.rotation = LeadUnit.transform.rotation;

570:

571: if (FlippedGate)

572: {

573: //Loop through other positions and set to Lead Position + Relevant Offset

574: for (int i = 1; i < SlotPositions.Count; i++)

575: {

576: SlotHolder = SlotPositions[0] + SlotOffsets[i];

577: //Rotate slot around center to match current rotation

578: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, LeadUnit.transform.position, LeadUnit.transform.eulerAngles);

579: }

580: }

581: else

582: {

583: //Loop through other positions and set to Lead Position + Relevant Offset

584: for (int i = 1; i < SlotPositions.Count; i++)

585: {

586: SlotHolder = SlotPositions[0] - SlotOffsets[i];

587: //Rotate slot around center to match current rotation

588: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, LeadUnit.transform.position, LeadUnit.transform.eulerAngles);

589: }

590: }

591:

592: }

593: }

594:

595: public Vector3 RotatePointAroundPivot(Vector3 point, Vector3 pivot, Vector3 angles)

596: {

597: return Quaternion.Euler(angles) \* (point - pivot) + pivot;

598: }

599:

600: private void MoveUnitsToSlots()

601: {

602: for(int i = 1; i < SlotPositions.Count; i++)

603: {

604: //Get variables required for specific ship and slot

605: GameObject Ship = FightersInFormation[i];

606: Rigidbody ShipBody = Ship.GetComponentInChildren<Rigidbody>();

607: if (ShipBody != null)

608: {

609: Vector3 TargetSlot = SlotPositions[i];

610: float ShipMoveSpeed = MoveSpeed;

611: Quaternion q = new Quaternion();

612:

613: //Turn to face slot position

614: q = Quaternion.LookRotation(TargetSlot + (LeadUnit.transform.forward \* 5) - Ship.transform.position, LeadUnit.transform.up);

615:

616: Ship.transform.rotation = Quaternion.RotateTowards(Ship.transform.rotation, q, RotationSpeed \* Time.deltaTime);

617:

618:

619: //Use Calculate force required to reach point with max move speed

620: //Allows for units on outside of curve to catch up to formation and prevents units on inside from overshooting point

621: float NeededSpeed = 0;

622: Vector3 VectToPoint = TargetSlot - Ship.transform.position;

623:

624: if (Vector3.Distance(TargetSlot, Ship.transform.position) < MoveSpeed / 100)

625: {

626: NeededSpeed = (MoveSpeed - 1) / Vector3.Distance(TargetSlot, Ship.transform.position);

627: }

628: else

629: {

630: NeededSpeed = MoveSpeed / Vector3.Distance(TargetSlot, Ship.transform.position);

631: }

632:

633: if(NeededSpeed < MoveSpeed)

634: {

635: ShipBody.AddForce(VectToPoint \* NeededSpeed, ForceMode.Impulse);

636: }

637: }

638: }

639: }

640:

641: private IEnumerator SpotPlayerDelay()

642: {

643: yield return new WaitForSeconds(10f);

644: CanSpotPlayer = true;

645: }

646: }

## Appendix 5

55: public class FighterStruct

56: {

57: public GameObject FighterOBJ;

58: public bool Avoiding;

59: public bool ReturningToSlot;

60: public Quaternion AvoidRotation;

61: }

## Appendix 6

100: private void GenerateFormation()

101: {

102: //Move Formation to randomly selected Spawn Gate

103: Transform LeadTransform = ChooseNewGate();

104: //Make copy of transform and rotation

105: LeadPosition = LeadTransform.position;

106: LeadRotation = LeadTransform.rotation;

107: //Set this position and rotation

108: this.transform.position = LeadPosition;

109: this.transform.rotation = LeadRotation;

110:

111: //Bool for Reverse Gates (prevents formations spawning backwards on half gates

112: FlippedGate = false;

113:

114: //Add LeadPosition

115: SlotPositions.Add(LeadPosition);

116: SlotOffsets.Add(new Vector3(0, 0, 0));

117:

118: //Rotate formation depending on spawn gate

119: if (LeadTransform.rotation.eulerAngles.y == 180)

120: {

121: LeadRotation.eulerAngles = new Vector3(0, 180f, 0);

122: FlippedGate = true;

123: }

124: else

125: {

126: LeadRotation.eulerAngles = new Vector3(0, 0f, 0);

127: FlippedGate = false;

128: }

129:

130: //Spawn Lead fighter

131: LeadUnit = Instantiate(UnitPrefab, LeadPosition, LeadRotation);

132: LeadBody = LeadUnit.GetComponentInChildren<Rigidbody>();

133: LeadUnit.GetComponentInChildren<FighterHealth>().ThisFormationBase = this;

134: LeadUnit.GetComponentInChildren<FighterHealth>().FormationIndex = 0;

135: //Add Fighter to list

136: FightersInFormation.Add(LeadUnit);

137:

138: //Create Lead fighter struct

139: FighterStruct LeadStruct = new FighterStruct();

140: LeadStruct.FighterOBJ = LeadUnit;

141: LeadStruct.Avoiding = false;

142:

143: FighterStructList.Add(LeadStruct);

144:

145: //For each fighter to add minus 1 for lead unit

146: for (int i = 0; i < UnitTotalCount - 1; i++)

147: {

148: Vector3 NewSlotPosition = new Vector3();

149: //Calculate Slot position for offset

150: if (i < 2)

151: {

152: //Check if gate is flipped

153: if (FlippedGate)

154: {

155: NewSlotPosition = LeadPosition - SlotOffset;

156: }

157:

158: else

159: {

160: NewSlotPosition = LeadPosition + SlotOffset;

161: }

162: }

163: else

164: {

165: //Check if gate is flipped

166: if (FlippedGate)

167: {

168: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] - SlotOffset;

169: }

170:

171: else

172: {

173: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] + SlotOffset;

174: }

175: }

176:

177: SlotOffset.x \*= -1;

178: //Add slot to list

179: SlotPositions.Add(NewSlotPosition);

180: //Store offset in list (Useful for updating slot position while in motion

181: Vector3 Newoffset = this.transform.position - NewSlotPosition;

182: SlotOffsets.Add(Newoffset);

183: //Spawn Fighter at position

184: GameObject SpawnedFighter = Instantiate(UnitPrefab, NewSlotPosition, LeadRotation);

185: //Give fighter reference to this formation

186: SpawnedFighter.GetComponentInChildren<FighterHealth>().ThisFormationBase = this;

187: SpawnedFighter.GetComponentInChildren<FighterHealth>().FormationIndex = i + 1;

188: //Add Fighter to list

189: FightersInFormation.Add(SpawnedFighter);

190:

191: FighterStruct CurrentStruct = new FighterStruct();

192: CurrentStruct.FighterOBJ = SpawnedFighter;

193: CurrentStruct.Avoiding = false;

194: FighterStructList.Add(CurrentStruct);

195: }

196: //Set Unit Count

197: ActiveUnitCount = UnitTotalCount;

198:

199: }

## Appendix 7

432: public class FormationBase : MonoBehaviour

433: {

434:

435: //See Appendix 2 for common functionality

436: private void FormationSpecific()

437: {

438: //Loop through each fighter to check for obstacles

439: for(int i = 0; i < ActiveUnitCount; i++)

440: {

441: GameObject ObstacleOBJ;

442: Vector3 HitPoint;

443: FighterStruct Fighter = FighterStructList[i];

444:

445: //Obstacle Detection

446: if (CheckForObstacle(Fighter, SlotOffsets[i].z, out ObstacleOBJ, out HitPoint))

447: {

448: //Identify obstacle and act accordingly

449: IdentifyObstacle(Fighter, ObstacleOBJ, HitPoint);

450: }

451:

452: //Else check if currently making turn to avoid obstacle

453: else if (Fighter.Avoiding)

454: {

455: ContinueAvoiding(Fighter);

456: }

457:

458: //Else check if fighter needs to turn back to slot after avoiding

459: else if (Fighter.ReturningToSlot)

460: {

461: ReturnToForm(Fighter, SlotPositions[i]);

462: }

463: }

464:

465: //Check for line of sight on player and lead unit not currently avoiding

466: if (CheckForSight() && LeadCurrentlyAvoiding == false)

467: {

468: TurnToPlayer();

469: }

470:

471: //Random Movement

472: else if(LeadCurrentlyAvoiding == false)

473: {

474: RandomMovement();

475: }

476:

477: LeadBody.AddForce(this.transform.forward \* (MoveSpeed - 1), ForceMode.Impulse);

478:

479: //Update Slot positions

480: UpdateSlotPositions();

481:

482: //Update Unit positions

483: MoveUnitsToSlots();

484: }

485:

486: private bool CheckForObstacle(FighterStruct CurrentShip, float SlotBackOffset, out GameObject HitObstacle, out Vector3 HitPoint)

487: {

488: RaycastHit HitOut = new RaycastHit();

489:

490: float RelevantCollisionRange = CollisionRange /\*+ Mathf.Abs(SlotBackOffset)\*/;

491:

492: //Get point in front of ship

493: Vector3 ShipFront = CurrentShip.FighterOBJ.transform.position + CurrentShip.FighterOBJ.transform.forward \* 5;

494: //Check for obstacle within set range

495: if (Physics.Raycast(ShipFront, CurrentShip.FighterOBJ.transform.forward, out HitOut, RelevantCollisionRange))

496: {

497: //Check if not already avoiding or if obstacle is too close anyway and not another enemy fighter

498: if ((CurrentShip.Avoiding == false || HitOut.distance <= 75) && !HitOut.collider.transform.root.CompareTag("EnemyFighter"))

499: {

500: //Set HitObstacle Output to object hit

501: HitObstacle = HitOut.collider.gameObject;

502: HitPoint = HitOut.point;

503: //Return True when obstacle

504: return true;

505: }

506: }

507:

508: //If no fighter returns a hit

509: HitObstacle = null;

510: HitPoint = Vector3.zero;

511: return false;

512: }

513:

514: private void IdentifyObstacle(FighterStruct CurrentShip, GameObject ObstacleOBJ, Vector3 HitPoint)

515: {

516: //Get Obstacle Tag

517: string ObstacleTag = ObstacleOBJ.tag;

518: float DistanceToObstacle = Vector3.Distance(this.transform.position, ObstacleOBJ.transform.position);

519:

520: //Act accordingly based on tag

521: switch (ObstacleTag)

522: {

523: //If hit object is player

524: case "Player":

525: //Check if Distance is too close

526: if (DistanceToObstacle <= PlayerAvoidRange)

527: {

528: CanSpotPlayer = false;

529: //Set Avoid Rotation

530: AvoidRotation = Quaternion.LookRotation((CurrentShip.FighterOBJ.transform.root.position - HitPoint) \* 0.5f);

531: //Set currently avoiding

532: CurrentShip.Avoiding = true;

533: //Check if obstacle is blocking lead unit

534: if (CurrentShip.FighterOBJ == LeadUnit)

535: {

536: //Set lead to avoiding

537: LeadCurrentlyAvoiding = true;

538: //Start rotation

539: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

540: }

541: else

542: {

543: //Set Ship to avoid

544: CurrentShip.Avoiding = true;

545: //Set Ships avoid rotation

546: CurrentShip.AvoidRotation = Quaternion.RotateTowards(CurrentShip.FighterOBJ.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

547: }

548:

549: StartCoroutine(SpotPlayerDelay());

550: }

551: //Else check if in range to engage

552: else if (DistanceToObstacle <= PlayerEngageRange)

553: {

554: foreach (FighterStruct Ship in FighterStructList)

555: {

556: //If ship is not avoiding

557: if(Ship.Avoiding == false && Ship.ReturningToSlot == false)

558: {

559: //Call firing

560: Ship.FighterOBJ.GetComponentInChildren<FighterFiring>().FireCannons(HitPoint);

561: }

562: }

563: }

564: //Otherwise formation will continue to move towards Player

565: break;

566:

567: //If obstacle is other

568: default:

569: //Calculate Avoidance Turn

570: AvoidRotation = Quaternion.LookRotation((CurrentShip.FighterOBJ.transform.root.position - HitPoint) \* 0.5f);

571: //Set Ship to avoid

572: CurrentShip.Avoiding = true;

573: //Set Ships avoid rotation

574: CurrentShip.AvoidRotation = AvoidRotation;

575: GameObject FighterOBJ = CurrentShip.FighterOBJ;

576: //Check if ship is lead unit

577: if (CheckIfLead(CurrentShip))

578: {

579: //Set lead to avoiding

580: LeadCurrentlyAvoiding = true;

581: }

582:

583: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

584:

585: break;

586: }

587: }

588:

589: private void ContinueAvoiding(FighterStruct Fighter)

590: {

591: GameObject FighterOBJ = Fighter.FighterOBJ;

592: //Check if arrived at rotation

593: if (FighterOBJ.transform.rotation == Fighter.AvoidRotation)

594: {

595: //If Lead Unit avoiding

596: if(CheckIfLead(Fighter))

597: {

598: //If yes, set currently avoiding to false

599: LeadCurrentlyAvoiding = false;

600: //Update Change Time

601: NextChangeTime = Time.time + Random.Range(MinDirChange, MaxDirChange);

602: //Update Random Position

603: RandomPosition();

604: }

605: //Otherwise set to returning to slot position

606: else

607: {

608: Fighter.ReturningToSlot = true;

609: }

610: Fighter.Avoiding = false;

611: }

612: else

613: {

614: //Continue to rotate towards

615: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, Fighter.AvoidRotation, RotationSpeed \* Time.deltaTime);

616: }

617: }

618:

619: public void ReturnToForm(FighterStruct Fighter, Vector3 ShipSlot)

620: {

621: GameObject FighterOBJ = Fighter.FighterOBJ;

622: Quaternion ToSlot = Quaternion.LookRotation(ShipSlot + (LeadUnit.transform.forward \* 5) - FighterOBJ.transform.position, LeadUnit.transform.up);

623: //Check if arrived at rotation

624: if (FighterOBJ.transform.rotation == ToSlot)

625: {

626: Fighter.ReturningToSlot = false;

627: }

628: else

629: {

630: //Continue to rotate towards

631: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, ToSlot, RotationSpeed \* Time.deltaTime);

632: }

633: }

634:

635: private bool CheckForSight()

636: {

637: RaycastHit HitOut = new RaycastHit();

638: //Check each fighter for line of sight

639: foreach (GameObject Ship in FightersInFormation)

640: {

641: Vector3 ShipFront = Ship.transform.position + Ship.transform.forward \* 8;

642: Vector3 VectToPlayer = PlayerOBJ.transform.position - ShipFront;

643: //Raycast towards player

644: if (Physics.Raycast(ShipFront, VectToPlayer, out HitOut, SightRange) && CanSpotPlayer)

645: {

646: //Check if hit is player

647: if (HitOut.collider.gameObject.transform.root.CompareTag("Player"))

648: {

649: //Return True if line of sight made

650: return true;

651: }

652: }

653:

654: }

655: //Return false otherwise

656: return false;

657: }

658:

659: private void TurnToPlayer()

660: {

661: Quaternion PlayerRotation = Quaternion.LookRotation(PlayerOBJ.transform.position - LeadUnit.transform.position);

662: //Turn Formation towards Player

663: LeadUnit.transform.rotation = Quaternion.RotateTowards(LeadUnit.transform.rotation, PlayerRotation, RotationSpeed \* Time.deltaTime);

664: }

665:

666: private void UpdateSlotPositions()

667: {

668:

669: //Check slot positions exist

670: if(SlotPositions.Count != 0)

671: {

672: Vector3 SlotHolder = new Vector3();

673: //Update Lead Position

674: SlotPositions[0] = LeadUnit.transform.position;

675: this.transform.position = LeadUnit.transform.position;

676: this.transform.rotation = LeadUnit.transform.rotation;

677:

678: if (FlippedGate)

679: {

680: //Loop through other positions and set to Lead Position + Relevant Offset

681: for (int i = 1; i < SlotPositions.Count; i++)

682: {

683: SlotHolder = SlotPositions[0] + SlotOffsets[i];

684: //Rotate slot around center to match current rotation

685: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, LeadUnit.transform.position, LeadUnit.transform.eulerAngles);

686: }

687: }

688: else

689: {

690: //Loop through other positions and set to Lead Position + Relevant Offset

691: for (int i = 1; i < SlotPositions.Count; i++)

692: {

693: SlotHolder = SlotPositions[0] - SlotOffsets[i];

694: //Rotate slot around center to match current rotation

695: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, LeadUnit.transform.position, LeadUnit.transform.eulerAngles);

696: }

697: }

698:

699: }

700: }

701:

702: public Vector3 RotatePointAroundPivot(Vector3 point, Vector3 pivot, Vector3 angles)

703: {

704: return Quaternion.Euler(angles) \* (point - pivot) + pivot;

705: }

706:

707: private void MoveUnitsToSlots()

708: {

709: for(int i = 1; i < ActiveUnitCount; i++)

710: {

711: //Get variables required for specific ship and slot

712: FighterStruct CurrentShipStruct = FighterStructList[i];

713: GameObject Ship = FighterStructList[i].FighterOBJ;

714: Rigidbody ShipBody = Ship.GetComponentInChildren<Rigidbody>();

715: Vector3 TargetSlot = SlotPositions[i];

716: float ShipMoveSpeed = MoveSpeed;

717: Quaternion q = new Quaternion();

718: Vector3 VectToPoint = new Vector3();

719: float NeededSpeed = 0;

720:

721: //If ship is not avoiding its own obstacle

722: if (CurrentShipStruct.Avoiding == false && CurrentShipStruct.ReturningToSlot == false)

723: {

724: //Turn to face slot position

725: q = Quaternion.LookRotation(TargetSlot + (LeadUnit.transform.forward \* 5) - Ship.transform.position, LeadUnit.transform.up);

726:

727: Ship.transform.rotation = Quaternion.RotateTowards(Ship.transform.rotation, q, RotationSpeed \* Time.deltaTime);

728:

729: //Use Calculate force required to reach point with max move speed

730: //Allows for units on outside of curve to catch up to formation and prevents units on inside from overshooting point

731: VectToPoint = TargetSlot - Ship.transform.position;

732:

733: if (VectToPoint.magnitude < MoveSpeed / 100)

734: {

735: NeededSpeed = (MoveSpeed - 1) / VectToPoint.magnitude;

736: }

737: else

738: {

739: NeededSpeed = MoveSpeed / VectToPoint.magnitude;

740: }

741:

742: if (NeededSpeed < MoveSpeed)

743: {

744: ShipBody.AddForce(VectToPoint \* NeededSpeed, ForceMode.Impulse);

745: }

746: }

747: else

748: {

749: //Set to use same speed as Lead

750: NeededSpeed = MoveSpeed - 1;

751: //Apply max speed force to ship forward vector

752: ShipBody.AddForce(Ship.transform.forward \* NeededSpeed, ForceMode.Impulse);

753: }

754: }

755: }

756:

757: private bool CheckIfLead(FighterStruct FighterIn)

758: {

759: if(FighterIn.FighterOBJ == LeadUnit)

760: {

761: return true;

762: }

763: else

764: {

765: return false;

766: }

767: }

768:

769: private IEnumerator SpotPlayerDelay()

770: {

771: yield return new WaitForSeconds(10f);

772: CanSpotPlayer = true;

773: }

774: }

## Appendix 8

201: public void RemoveFighter(GameObject ToRemove, int InIndex)

202: {

203: Debug.Log("Removing at: " + InIndex);

204: //Remove Fighter from list

205: //All units in slots above removed ship will then move down a slot

206: FightersInFormation.Remove(ToRemove);

207: //Check if lead unit needs replacing

208: if (ToRemove == LeadUnit)

209: {

210: LeadCurrentlyAvoiding = false;

211: //Lead unit is always index 0, so set new lead to index 1

212: LeadUnit = FightersInFormation[0];

213: LeadBody = LeadUnit.GetComponent<Rigidbody>();

214: }

215: //Remove last slot in formation as it is now unoccupied

216: SlotPositions.RemoveAt(SlotPositions.Count - 1);

217: SlotOffsets.RemoveAt(SlotOffsets.Count - 1);

218: FighterStructList.RemoveAt(InIndex);

219: //Decrease number of active fighters

220: ActiveUnitCount -= 1;

221: //Check if there is only 1 unit remaining

222: if (ActiveUnitCount == 1)

223: {

224: Debug.Log("FormationDestroyed");

225: MoveFighterToOtherForm();

226: //Destroy formation

227: Manager.SpawnedFormations.Remove(this.gameObject);

228: Destroy(this.gameObject);

229: }

230: //Otherwise update all indexes for ships in formation

231: else

232: {

233: for(int i = 0; i < ActiveUnitCount; i++)

234: {

235: GameObject Fighter = FightersInFormation[i];

236: Fighter.GetComponentInChildren<FighterHealth>().FormationIndex = i;

237: Fighter.GetComponentInChildren<FriendlyMarker>().MarkerActive = true;

238: }

239: }

240: }

## Appendix 9

288: public void AddFighter(GameObject ToAdd)

289: {

290: //Use similar code as formation generation for new slot position

291: Vector3 NewSlotPosition = new Vector3();

292: //Check if formation has more than 3 fighters

293: if (ActiveUnitCount < 3)

294: {

295: //Check if gate is flipped

296: if (FlippedGate)

297: {

298: NewSlotPosition = SlotPositions[0] - SlotOffset;

299: }

300:

301: else

302: {

303: NewSlotPosition = SlotPositions[0] + SlotOffset;

304: }

305: }

306: else

307: {

308: //Check if gate is flipped

309: if (FlippedGate)

310: {

311: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] - SlotOffset;

312: }

313:

314: else

315: {

316: NewSlotPosition = SlotPositions[SlotPositions.Count - 2] + SlotOffset;

317: }

318: }

319:

320: SlotPositions.Add(NewSlotPosition);

321: SlotOffset.x \*= -1;

322: //Store offset in list (Useful for updating slot position while in motion

323: Vector3 Newoffset = this.transform.position - NewSlotPosition;

324: SlotOffsets.Add(Newoffset);

325: //Add fighter to list of formation units

326: FightersInFormation.Add(ToAdd);

327:

328: //Create New Fighter struct

329: FighterStruct CurrentStruct = new FighterStruct();

330: CurrentStruct.FighterOBJ = ToAdd;

331: CurrentStruct.Avoiding = false;

332: FighterStructList.Add(CurrentStruct);

333:

334: //Increase number of fighters in formation

335: ActiveUnitCount += 1;

336: }

## Appendix 10

406: public class FormationBase : MonoBehaviour

407: {

408: //See Appendix 2 for common functionality

409: private void FormationSpecific()

410: {

411: //Loop through each fighter to check for obstacles

412: for(int i = 0; i < ActiveUnitCount; i++)

413: {

414: GameObject ObstacleOBJ;

415: Vector3 HitPoint;

416: FighterStruct Fighter = FighterStructList[i];

417:

418: //Obstacle Detection

419: if (CheckForObstacle(Fighter, SlotOffsets[i].z, out ObstacleOBJ, out HitPoint))

420: {

421: //Identify obstacle and act accordingly

422: IdentifyObstacle(Fighter, ObstacleOBJ, HitPoint);

423: }

424:

425: //Else check if currently making turn to avoid obstacle

426: else if (Fighter.Avoiding)

427: {

428: ContinueAvoiding(Fighter);

429: }

430:

431: //Else check if fighter needs to turn back to slot after avoiding

432: else if (Fighter.ReturningToSlot)

433: {

434: ReturnToForm(Fighter, SlotPositions[i]);

435: }

436: }

437:

438: //Check for line of sight on player and lead unit not currently avoiding

439: if (CheckForSight())

440: {

441: TurnToPlayer();

442: }

443:

444: //Random Movement

445: else

446: {

447: RandomMovement();

448: }

449:

450: ThisBody.AddForce(this.transform.forward \* (MoveSpeed - 1), ForceMode.Impulse);

451:

452: //Update Slot positions

453: UpdateSlotPositions();

454:

455: //Update Unit positions

456: MoveUnitsToSlots();

457: }

458:

459: private bool CheckForObstacle(FighterStruct CurrentShip, float SlotBackOffset, out GameObject HitObstacle, out Vector3 HitPoint)

460: {

461: RaycastHit HitOut = new RaycastHit();

462:

463: float RelevantCollisionRange = CollisionRange /\*+ Mathf.Abs(SlotBackOffset)\*/;

464:

465: //Get point in front of ship

466: Vector3 ShipFront = CurrentShip.FighterOBJ.transform.position + CurrentShip.FighterOBJ.transform.forward \* 5;

467: //Check for obstacle within set range

468: if (Physics.Raycast(ShipFront, CurrentShip.FighterOBJ.transform.forward, out HitOut, RelevantCollisionRange))

469: {

470: //Check if not already avoiding or if obstacle is too close anyway and not another enemy fighter

471: if ((CurrentShip.Avoiding == false || HitOut.distance <= 75) && !HitOut.collider.transform.root.CompareTag("EnemyFighter"))

472: {

473: //Set HitObstacle Output to object hit

474: HitObstacle = HitOut.collider.gameObject;

475: HitPoint = HitOut.point;

476: //Return True when obstacle

477: return true;

478: }

479: }

480:

481: //If no fighter returns a hit

482: HitObstacle = null;

483: HitPoint = Vector3.zero;

484: return false;

485: }

486:

487: private void IdentifyObstacle(FighterStruct CurrentShip, GameObject ObstacleOBJ, Vector3 HitPoint)

488: {

489: //Get Obstacle Tag

490: string ObstacleTag = ObstacleOBJ.tag;

491: float DistanceToObstacle = Vector3.Distance(this.transform.position, ObstacleOBJ.transform.position);

492:

493: Debug.Log(ObstacleTag);

494:

495: //Act accordingly based on tag

496: switch (ObstacleTag)

497: {

498: //If hit object is player

499: case "Player":

500: //Check if Distance is too close

501: if (DistanceToObstacle <= PlayerAvoidRange)

502: {

503: CanSpotPlayer = false;

504: //Set Avoid Rotation

505: AvoidRotation = Quaternion.LookRotation((CurrentShip.FighterOBJ.transform.root.position - HitPoint) \* 0.5f);

506:

507: //Set Ship to avoid

508: CurrentShip.Avoiding = true;

509: //Set Ships avoid rotation

510: CurrentShip.AvoidRotation = Quaternion.RotateTowards(CurrentShip.FighterOBJ.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

511:

512: StartCoroutine(SpotPlayerDelay());

513: }

514: //Else check if in range to engage

515: else if (DistanceToObstacle <= PlayerEngageRange)

516: {

517: foreach (FighterStruct Ship in FighterStructList)

518: {

519: //If ship is not avoiding

520: if (Ship.Avoiding == false && Ship.ReturningToSlot == false)

521: {

522: //Call firing

523: Ship.FighterOBJ.GetComponentInChildren<FighterFiring>().FireCannons(HitPoint);

524: }

525: }

526: }

527: //Otherwise formation will continue to move towards Player

528: break;

529:

530: //If obstacle is other

531: default:

532: //Calculate Avoidance Turn

533: AvoidRotation = Quaternion.LookRotation((CurrentShip.FighterOBJ.transform.root.position - HitPoint) \* 0.5f);

534: //Set Ship to avoid

535: CurrentShip.Avoiding = true;

536: //Set Ships avoid rotation

537: CurrentShip.AvoidRotation = AvoidRotation;

538: GameObject FighterOBJ = CurrentShip.FighterOBJ;

539:

540: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, AvoidRotation, RotationSpeed \* Time.deltaTime);

541: break;

542: }

543: }

544:

545: private void ContinueAvoiding(FighterStruct Fighter)

546: {

547: GameObject FighterOBJ = Fighter.FighterOBJ;

548: //Check if arrived at rotation

549: if (FighterOBJ.transform.rotation == Fighter.AvoidRotation)

550: {

551: Fighter.ReturningToSlot = true;

552: Fighter.Avoiding = false;

553: }

554: else

555: {

556: //Continue to rotate towards

557: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, Fighter.AvoidRotation, RotationSpeed \* Time.deltaTime);

558: }

559: }

560:

561: public void ReturnToForm(FighterStruct Fighter, Vector3 ShipSlot)

562: {

563: GameObject FighterOBJ = Fighter.FighterOBJ;

564: Quaternion ToSlot = Quaternion.LookRotation(ShipSlot + (this.transform.forward \* 5) - FighterOBJ.transform.position, this.transform.up);

565: //Check if arrived at rotation

566: if (FighterOBJ.transform.rotation == ToSlot)

567: {

568: Fighter.ReturningToSlot = false;

569: }

570: else

571: {

572: //Continue to rotate towards

573: FighterOBJ.transform.rotation = Quaternion.RotateTowards(FighterOBJ.transform.rotation, ToSlot, RotationSpeed \* Time.deltaTime);

574: }

575: }

576:

577: private bool CheckForSight()

578: {

579: RaycastHit HitOut = new RaycastHit();

580: //Check each fighter for line of sight

581: foreach (GameObject Ship in FightersInFormation)

582: {

583: Vector3 ShipFront = Ship.transform.position + Ship.transform.forward \* 8;

584: Vector3 VectToPlayer = PlayerOBJ.transform.position - ShipFront;

585: //Raycast towards player

586: if (Physics.Raycast(ShipFront, VectToPlayer, out HitOut, SightRange) && CanSpotPlayer)

587: {

588: //Check if hit is player

589: if (HitOut.collider.gameObject.transform.root.CompareTag("Player"))

590: {

591: //Return True if line of sight made

592: return true;

593: }

594: }

595:

596: }

597: //Return false otherwise

598: return false;

599: }

600:

601: private void TurnToPlayer()

602: {

603: Quaternion PlayerRotation = Quaternion.LookRotation(PlayerOBJ.transform.position - this.transform.position);

604: //Turn Formation towards Player

605: this.transform.rotation = Quaternion.RotateTowards(this.transform.rotation, PlayerRotation, RotationSpeed \* Time.deltaTime);

606: }

607:

608: private void UpdateSlotPositions()

609: {

610:

611: //Check slot positions exist

612: if(SlotPositions.Count != 0)

613: {

614: Vector3 SlotHolder = new Vector3();

615: //Update Lead Position

616: SlotPositions[0] = this.transform.position;

617: this.transform.position = this.transform.position;

618: this.transform.rotation = this.transform.rotation;

619:

620: if (FlippedGate)

621: {

622: //Loop through other positions and set to Lead Position + Relevant Offset

623: for (int i = 1; i < SlotPositions.Count; i++)

624: {

625: SlotHolder = SlotPositions[0] + SlotOffsets[i];

626: //Rotate slot around center to match current rotation

627: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, this.transform.position, this.transform.eulerAngles);

628: }

629: }

630: else

631: {

632: //Loop through other positions and set to Lead Position + Relevant Offset

633: for (int i = 1; i < SlotPositions.Count; i++)

634: {

635: SlotHolder = SlotPositions[0] - SlotOffsets[i];

636: //Rotate slot around center to match current rotation

637: SlotPositions[i] = RotatePointAroundPivot(SlotHolder, this.transform.position, this.transform.eulerAngles);

638: }

639: }

640:

641: }

642: }

643:

644: public Vector3 RotatePointAroundPivot(Vector3 point, Vector3 pivot, Vector3 angles)

645: {

646: return Quaternion.Euler(angles) \* (point - pivot) + pivot;

647: }

648:

649: private void MoveUnitsToSlots()

650: {

651: for(int i = 0; i < ActiveUnitCount; i++)

652: {

653: //Get variables required for specific ship and slot

654: FighterStruct CurrentShipStruct = FighterStructList[i];

655: GameObject Ship = FighterStructList[i].FighterOBJ;

656: Rigidbody ShipBody = Ship.GetComponentInChildren<Rigidbody>();

657: Vector3 TargetSlot = SlotPositions[i];

658: float ShipMoveSpeed = MoveSpeed;

659: Quaternion q = new Quaternion();

660: Vector3 VectToPoint = new Vector3();

661: float NeededSpeed = 0;

662:

663: //If ship is not avoiding its own obstacle

664: if (CurrentShipStruct.Avoiding == false && CurrentShipStruct.ReturningToSlot == false)

665: {

666: //Turn to face slot position

667: q = Quaternion.LookRotation(TargetSlot + (this.transform.forward \* 5) - Ship.transform.position, this.transform.up);

668:

669: Ship.transform.rotation = Quaternion.RotateTowards(Ship.transform.rotation, q, RotationSpeed \* Time.deltaTime);

670:

671: //Use Calculate force required to reach point with max move speed

672: //Allows for units on outside of curve to catch up to formation and prevents units on inside from overshooting point

673: VectToPoint = TargetSlot - Ship.transform.position;

674:

675: if (VectToPoint.magnitude < MoveSpeed / 100)

676: {

677: NeededSpeed = (MoveSpeed - 1) / VectToPoint.magnitude;

678: }

679: else

680: {

681: NeededSpeed = MoveSpeed / VectToPoint.magnitude;

682: }

683:

684: if (NeededSpeed < MoveSpeed)

685: {

686: ShipBody.AddForce(VectToPoint \* NeededSpeed, ForceMode.Impulse);

687: }

688: }

689: else

690: {

691: //Set to use same speed as Lead

692: NeededSpeed = MoveSpeed - 1;

693: //Apply max speed force to ship forward vector

694: ShipBody.AddForce(Ship.transform.forward \* NeededSpeed, ForceMode.Impulse);

695: }

696: }

697: }

698:

699: private IEnumerator SpotPlayerDelay()

700: {

701: yield return new WaitForSeconds(10f);

702: CanSpotPlayer = true;

703: }

704: }

Appendix 11

380: private void RandomMovement()

381: {

382: //Check if time to choose random movement point

383: if(Time.time >= NextChangeTime || Vector3.Distance(this.transform.position, RandomTarget) <= 10)

384: {

385: //Update Change Time

386: NextChangeTime = Time.time + Random.Range(MinDirChange, MaxDirChange);

387: //Update Random Position

388: RandomPosition();

389: }

390: Vector3 VectToTarget = RandomTarget - this.transform.position;

391: if (Vector3.Angle(this.transform.forward, VectToTarget) != 0)

392: {

393: var q = Quaternion.LookRotation(VectToTarget);

394: this.transform.rotation = Quaternion.RotateTowards(this.transform.rotation, q, RotationSpeed \* Time.deltaTime);

395: }

396: }

## Appendix 11

## Appendix 12

Formation Flight Questionnaire

1) Which of the formations did you find most visually appealing? And why?

2) Which of the formations did you find least visually appealing? And why?

3) Which formation do you believe handled turning best? And why?

4) Which formation do you believe handled turning the worst? And why?

5) Which formation avoided obstacles in the most appealing way? And why?

6) Which formation avoided obstacles in the least appealing way? And why?

7) Did you experience any performance issues with any formations?

7.5) If yes, which formation(s) caused the issues?