Bachelor's thesis

Information Technology

Game Technology

2017

Sallamari Rantanen

AI FOR WARPLANES IN A 2D SIDE-SCROLLER

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Game Technology

2017 | 33

Sallamari Rantanen

AI FOR WARPLANES IN A 2D SIDE-SCROLLER

The subject of the thesis was to develop an AI for warplanes. The main aim of the thesis was to find the most suitable algorithms for achieving the different reactions that the game's non-player character warplanes need to display. The warplane had to be able to take off, search, and follow an enemy, shoot, bomb, and land. Furthermore, the algorithms had to be as efficient as possible while being able to adhere to flow theory and satisfy end-user expectations of intelligent behavior. The thesis is part of an ongoing game project developed by a game company MansikkaMarmeladi.io.

The project was created with GameMaker 1.4 and with the built-in programming language GML. The theoretical section of the thesis goes through optimal flow theory, pathfinding, finite state machines and Big O notation. These were utilized during the planning, programming, and evaluation phases.

The end result was evaluated on the basis of difficulty level as well as the scalability of the code. Playtesting results showed that the AI created a reasonable challenge for a first level of the single player campaign. Since the full campaign is not ready yet, further testing will be conducted once it is ready to ensure that the difficulty will increase at the right pace. The scalability of the code turned out to be genuinely good as the functions with the greatest complexities were O(n). As the functions' input data will not grow exceedingly large, no scalability issues are foreseen in the future.

The end result of the thesis was an AI fulfilling the original requirements that will be utilized in the further development of the game project.

KEYWORDS:

Artificial intelligence, GameMaker, optimal flow theory, pathfinding, finite state machines, Big O notation

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Peliteknologia

2017 | 33

Sallamari Rantanen

LENTOKONEIDEN TEKOÄLY 2D-SIDE-SCROLLERISSA

Työn tarkoituksena oli kehittää tekoäly peliprojektin lentokoneille. Päätavoitteena oli löytää sopivimmat algoritmit, jotka mahdollistaisivat tekoälyltä odotetut reaktiot. Lentokoneen tuli osata nousta ilmaan, etsiä ja seurata vihollista, ampua, pommittaa ja laskeutua. Lisäksi algoritmien piti olla mahdollisimman tehokkaita noudattaen samalla flow-teorian perusteita sekä täyttäen pelaajan odotukset älykkäästä käytöksestä. Työ on osa peliyritys MansikkaMarmeladi.io:n kehitteillä olevaa peliprojektia.

Työ tehtiin GameMaker 1.4 -pelimoottorilla ja siihen sisäänrakennetulla ohjelmointikielellä (GML). Teoriaosuus käsitteli flow-teoriaa, polunetsintää, äärellisiä tilakoneita sekä Big O -notaatiota. Teoriaa hyödynnettiin niin suunnittelu-, ohjelmointi- kuin arvointivaiheessakin.

Työn lopputulosta arvioitiin sekä vaikeustason että koodin skaalautuvuuden kannalta. Pelitestin tulokset osoittivat, että tekoäly loi sopivasti haastetta yksinpelikampanjan ensimmäiselle tasolle. Koska koko kampanja ei ole vielä valmis, tarvitaan lisää pelitestejä sen valmistuttua. Tällöin varmistetaan vaikeustason kasvu sopivassa tahdissa. Koodin skaalautuvuus osoittautui erittäin hyväksi, sillä funktioiden huonoin skaalautuvuusarvo oli O(n). Koska funktioiden parametrit eivät kasva suuriksi, skaalautuvuuden ei pitäisi aiheuttaa ongelmia tulevaisuudessa.

Työn lopputuloksena saatiin tekoäly, joka täytti alkuperäiset vaatimukset. Asiakas hyödyntää työn tulosta peliprojektin jatkokehityksessä.

ASIASANAT:

Tekoäly, GameMaker, flow-teoria, polunetsintä, äärellinen tilakone, Big O -notaatio

CONTENTS

LIST OF ABBREVIATIONS 6

1 INTRODUCTION 7

2 THEORETICAL FRAMEWORK 9

2.1 Flow theory 9

2.1.1 Goals 9

2.1.2 Difficulty level 10

2.1.3 Feedback 11

2.1.4 Concentration 11

2.2 Pathfinding 12

2.2.1 Dijkstra 12

2.2.2 Greedy best first search 14

2.2.3 A* search 14

2.3 Finite state machines 14

2.4 Big O notation 16

3 BUILDING THE WARPLANE AI 19

3.1 Game Engine and language 19

3.2 Planning 20

3.3 First build-test-evaluate cycle 22

3.3.1 Client feedback 23

3.4 Second build-test-evaluate cycle 23

3.4.1 Client feedback 25

3.5 Third build-test-evaluate cycle 25

4 RESULTS AND DISCUSSION 28

5 CONCLUSION 32

REFERENCES 33

FIGURES

Figure 1. The most common algorithm complexities. 18

EQUATIONS

Equation 1. First version of the deceleration formula. 24
Equation 2. Second version of the deceleration formula. 26

PICTURES

Picture 1. Two different ways to move up the flow channel. 11
Picture 2. Example of cumulative cost. 13
Picture 3. Simple state machine. 15
Picture 4. Example code for $O(n2)$. 17
Picture 5. GameMaker’s basic layout. 20
Picture 6. State machine diagram for the AI 21
Picture 7. Examples of grids divided into cells of 25 and 10 pixels. 22
Picture 8. Code for the takeoff with turning. 30
Picture 9. Code for the normal takeoff. 31

TABLES

Table 1. Results of playtesting. 28

LIST OF ABBREVIATIONS

2D	Two-dimensional
AI	Artificial intelligence
GML	GameMaker Language
MS-DOS	Microsoft Disk Operating System
NPC	Non-player character

1 INTRODUCTION

The aim of the thesis was to create an AI for warplanes in a 2D side-scroller. The player's goal was to defeat the enemy lines with their own warplane while keeping track of ammunition and fuel. Depending on a chosen game mode, they might also need to protect important areas or resources that the enemy is trying to destroy. Enemy planes need to be able to follow the player, choose a weapon, and follow a strategy that fits the current game mode. The thesis was part of an ongoing project so the game engine and the programming languages were chosen already. The game would be created with GameMaker 1.4 and with the built-in GameMaker Language. The thesis was commissioned by a game company MansikkaMarmeladi.io.

The theoretical section of the thesis includes information about pathfinding, finite state machines, optimal flow theory and Big O notation. Optimal flow theory is examined in order to make the gameplay experience as engaging as possible. It helps to guide the development process by highlighting the design choices that need extra focus. Big O notation is used to compare and evaluate the efficiency of algorithms.

There are various methods and algorithms for AI development so there are multiple ways to approach the same problem. The complexity of the AI, the size of the game, and the possible scalability should all be considered while choosing the technique. (Kirby 2010, 1-2.) The game engine selection might affect some of the decisions as well since some of them offer tools for AI problems. For example, pathfinding can sometimes be done completely with these kind of tools or with just a few lines of code. AI has a huge impact on player experience in most games, so badly designed AI can wreck the whole game experience. This is why this thesis pays such careful attention to the game’s AI design of its non-player character (NPC) warplanes.

Regarding the difficulty, the game developed during the thesis will be quite challenging for the player. Achieving the appropriate difficulty level affects the way the AI should be planned and programmed. The enemies should create enough challenge for the player without making it too frustrating. The difficulty level is strongly affected by the accuracy of the enemy planes and their possible ability to predict the player’s next move. Getting the difficulty just right can be a challenging process since people have different amounts of experience with gaming and therefore varying skill levels. The optimal flow theory principles will help to create the right balance that will keep players engaged. The

difficulty level will also be assessed by a playtest focusing on the level of challenge created by the AI.

2 THEORETICAL FRAMEWORK

AI for games comprises several basic building blocks that allow players to experience intelligent enemies. These building blocks are mainly concerned with: (a) control, in the form of path finding; (b) structure, being the containers (or states) for the various AI behaviors; (c) user experience, determined by the achieved sense of flow; and (d) efficiency and scalability, calculated by means of the Big O notation. Careful consideration of these building blocks and how they are combined by the developer is required for the best possible play experience.

2.1 Flow theory

Psychologist Mihaly Csikszentmihaly created an idea of optimal flow. Flow can be achieved when people are engaged in a task that has a difficulty matching their skill level. When people are in a flow state, they are fully concentrated and engaged on the task at hand, which is then so enjoyable and rewarding that they do not need any external motivation to play the game in question. (Murphy 2011.) Furthermore, people experiencing flow can be so focused on the task that they lose their sense of time and awareness of physical needs (Slabinski 2013).

Flow is one of the main reasons people get into playing video games, and the game designers are trying to maximize the time that players spend in a flow state (Murphy 2011).

There are four characteristics that should be taken into consideration while designing a game in order to maximize the chance of achieving flow (Baron 2012).

2.1.1 Goals

If the player does not know their goals or how to achieve them, they will lose interest. The goals should be presented clearly in a way that the player will always know their current aim, meaning that the player's attention needs to be gained fully before the goal is given. That is, it should not be during a hectic moment in the game. Also, the player should have been able to gather the information needed to achieve these goals. If some

new gameplay mechanics are needed, they should be taught to the player first. Clear and achievable objectives give the player a sense of achievement which motivates them to continue playing. (Baron 2012.)

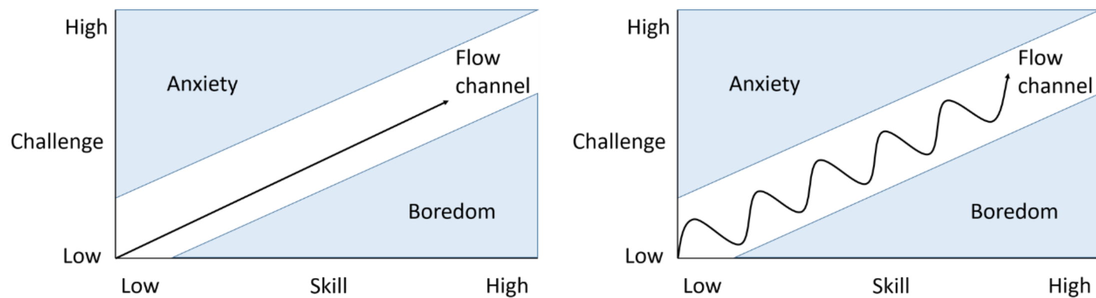
2.1.2 Difficulty level

If the goals and mechanics are clear to the player, but they just do not manage to do a required action because they are not skilled enough, they will get frustrated and stop playing. Every player has a different stress tolerance which means that some players will stop playing after a small amount of stress while others can endure a lot more. (Baron 2012.)

The tasks cannot be too easy either or the player will get bored. It is also important to note that the player's skills will improve over time, so the tasks should get more challenging at approximately the same pace. (Murphy 2011.)

When this kind of balance is achieved, the player is inside a flow channel. In order to maintain this balance, the difficulty level will have to follow the player's learning curve. Games often have different levels that get more challenging, one after the other. Having a level-based difficulty system helps to keep players with different skill levels inside the flow channel. The more experienced players will move on to the more difficult levels faster and not get bored with the easy ones. Similarly, the players with less experience have a chance to learn the game mechanics at their own pace. (Schell 2008, 120.)

Picture 1 shows that moving up the channel can be done in two ways. The straightforward approach has a constant increase in the difficulty of the game. The wavelike approach does not increase the challenge immediately, but instead lets the player to feel experienced while the difficulty level is a bit lower. After this short break, the challenge is increased again so that the player does not have a chance to get bored. Both of these approaches maintain the balance but the player might find the latter more interesting. (Schell 2008, 121-122.)



Picture 1. Two different ways to move up the flow channel.

2.1.3 Feedback

The player should always know if they have acted in the correct way or not. If the player is doing something wrong, but not given any indication that the action is not a correct one, they might get frustrated and stop playing. Since the player has no idea what they did wrong, they will not know how to change their behavior either. It is also important that the feedback is given at the right time so that there will not be any confusion on which action triggered it. (Baron 2012.)

Games use a feedback loop that consists of assessing the action, providing the conclusion to the player, and creating an outcome and opportunities for alternate actions. Feedback in games can be divided into two groups: short term and holistic. Short term gives immediate feedback to the player about their actions. For example, increasing the score when the player has picked up a collectible. Holistic feedback reflects the progression on a larger scale. Like showing how many collectibles have been collected and how many are still left. (Murphy 2011.) Holistic feedback can motivate the player a lot more, since the progress can be seen continuously, compared to getting an achievement after the winning conditions have been met (Baron 2012).

2.1.4 Concentration

In order for the player to stay in the flow, they need to be given the chance to concentrate. So while designing the game, it should be made as logical and congruent as possible. The mechanics should stay the same throughout the game, or the player needs to have time to adjust to new ones. Also, the user interface needs to be designed carefully. If the UI is full of icons and other information, it can impede the player’s concentration, learning,

and comprehension. The UI should be designed in a way that makes it easily understandable even for a new player. Any new or important information should be highlighted in some way to make sure that it gets the player's attention. (Baron 2012.)

2.2 Pathfinding

AI characters can move on a predetermined route, but sometimes more complex solutions are needed. In order to change the route dynamically it can be calculated by using pathfinding methods. This allows the AI to avoid obstacles and still move intelligently using the shortest route. There are multiple pathfinding algorithms, but the most popular one in video games is A*. (Millington 2006, 203-204.)

2.2.1 Dijkstra

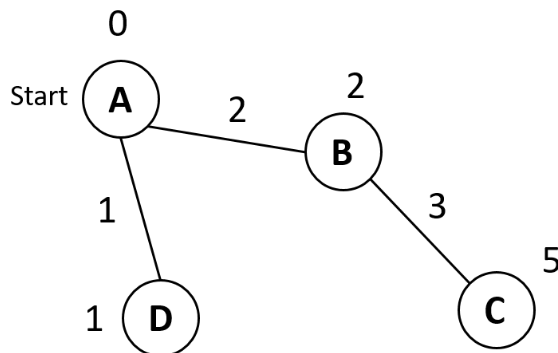
The Dijkstra algorithm, named after the creator Edsger Dijkstra, was originally developed to solve the "shortest path" problem in mathematical graph theory (Millington 2006, 211). The algorithm works by expanding in every direction from the starting point until it reaches the goal. As long as the connections do not have negative costs, the algorithm will always be able to return the shortest path. (Patel 2017.)

As it expands outwards from the starting point, it creates multiple paths which is wasteful if only the shortest path to the goal is wanted. That is why Dijkstra as itself is not usually used in games for pathfinding, but it has created the basis for more advanced algorithms. In order to understand the more efficient algorithms, it is good to start from the basic principles. (Millington 2006, 211.)

The algorithm deduces the shortest path from the start to the end node. If there are multiple paths, only one is returned. Dijkstra spreads from the start node to all of its connections while keeping track of the node and the direction it came from. This allows it to return the shortest path by going back the previous nodes once it has reached the goal. (Millington 2006, 211-212.)

Dijkstra goes through the nodes in iterations. During an iteration it follows each connection to the connected node while storing the cost of the path and the node it came from. The cost of the path is cumulative, meaning that the cost from the previous connection gets added to the connected nodes costs (Picture 2). (Millington 2006, 211-

212.) In Picture 2, the node C has a cost of 5, since the costs of the two connections leading to it gets added up.



Picture 2. Example of cumulative cost.

When a node has been visited it is either added to the “open” or “closed” list. After a node has been gone through an iteration, meaning that all of its connections have been calculated, it gets added to the closed list. And all of the nodes that were connected to that node get added to the open list. Meaning that they have been visited, but not processed fully yet. In each iteration, the node with the smallest cumulative path cost gets processed. (Millington 2006, 212-214.)

If a connected node happens to already be on the open or closed list, the path cost and the node it came from are not updated automatically. Since the current path might be worse than the one already stored, the costs have to be compared. If the current path has a lower cost, then the cost and the connection gets updated. Also the node is moved into the open list. Otherwise the current values are kept, and the node will stay on the list it is currently on. (Millington 2006, 214.)

Normally the algorithm terminates once the open list is empty, meaning it has gone through all of the reachable nodes. For pathfinding purposes only one path is needed: the shortest path from the start to the end node. That is why the algorithm can be terminated once the end node has the smallest value in the open list. At that point there will not be a shorter route through any unprocessed nodes. Once the algorithm has finished or been terminated, it returns the shortest path. It does this in the reverse order starting from the end node. Then it moves backwards towards the start node while storing the connections on a list. When the start node is reached, the list needs to be reversed since the path was started from the end node. (Millington 2006, 214-216.)

2.2.2 Greedy best first search

Greedy best first search minimizes the cost from the start to the end node. This cost is an estimate since it usually cannot be determined exactly. The cost estimates are calculated by heuristic functions, such as straight-line distance. Straight-line distance is, as its name suggests, the shortest path in a straight line between two points. (Russell & Norvig 1995, 93-97.)

Unlike Dijkstra that always returns the shortest path, greedy best first search is not guaranteed to do that. As the name states the algorithm is greedy, meaning that it will move towards the goal even if it is not the shortest path. The algorithm is only interested in finding the shortest heuristic distance each step and not about the path cost so far (the length of the path). This is why it can lead to very long and inefficient looking paths. It is faster than Dijkstra since it expands only towards the goal by determining the direction with a heuristic function. (Patel 2017.)

2.2.3 A* search

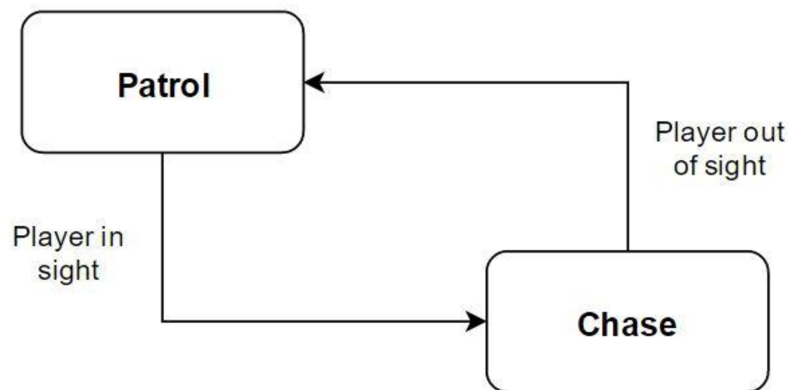
A* is a combination of Dijkstra and greedy best first search. It calculates the total cost of the path by using a cumulative cost added to the heuristic distance. Since it uses a heuristic function, it only expands to the direction of the goal, which makes A* a lot faster than Dijkstra. And unlike greedy best first search the algorithm always returns the shortest path, which makes it a better choice for pathfinding. So basically A* gets rid of the negative aspects of the two functions while combining the positives. This is why it is a popular choice for pathfinding. (Patel 2017.)

2.3 Finite state machines

Finite state machines are a collection of different states that the program can go through during its execution. These states are connected to each other with transitions. In the game development sense each state represents a certain behavior of the AI. (Kirby 2010, 43-44.) The character can only be in a one state at a time and will keep doing the same actions until a transition happens. Transition occurs once some predefined corresponding conditions are met. (Millington 2006, 318.)

Finite state machines have been used in video games for ages even though more complex methods have been developed. The long-lasting popularity is due to the positive qualities of the method. Finite state machines are simple to program and debug. They also create some flexibility since the code has been divided into smaller parts, which makes it easy to add more states and conditions. (Buckland 2005, 43.)

One simple example of a finite state machine would be having one state for patrolling the area and other for chasing intruders (Picture 3).



Picture 3. Simple state machine.

In order for the AI to change state and start chasing, the transition conditions need to be valid. In this case the AI needs to have seen the player. The chasing state will be active until the transition conditions back to the patrolling state have been met. So once the AI cannot see the player anymore, it will start patrolling again.

In a finite state machine the AI can be in a one state at a time, but it might have multiple options for exiting its current state. This kind of multiple-transition might lead into problems if more than one transition is valid at the same time. It can make the AI move to a state and then immediately move out of it, because the exit conditions were already true. This means that the character should not have transitioned to the state in the first place. The most common way to prevent this kind of unwanted action is to prioritize the transitions. This way the transitions will be checked in a fixed order. If two transitions are valid, but transition A gets checked first, transition B is ignored since the AI will switch states using transition A. (Kirby 2010, 47-48.)

The state itself consists of an entry, exit and update functions. The entry function is run once the transition has taken place. All the needed setup happens in there. For example, all the variables get assigned with their initial values. The update function is the one that contains the actions that the AI will be repeating while being inside the state. The exit function is run once the transition conditions are met. This function is used for setting everything up for the next state to work correctly. (Kirby 2010, 49-50.)

The suitability of the method needs to be verified before use. If the code cannot be divided into sensible states, some other method should be considered. The easiest way to design the states and their transitions is to create a state machine diagram. While creating the layout, it is important to make sure that the transitions make sense at all times, since the variables and other data can change significantly. The transition can work perfectly at the start of the program but stop working once the state has been changed. This means that if the conditions triggering the transitions have not been planned carefully, the AI might get stuck in one state, even though it should have moved into a different one. Creating a diagram of the whole layout reduces the chance of this kind of problem arising, since all of the states and transitions need to be fully thought through. (Kirby 2010, 44-47.)

2.4 Big O notation

The efficiency of an algorithm is measured by determining the number of steps it takes during the execution. A step being a basic unit of computation that needs to be chosen specifically for the algorithm in question. (Miller & Ranum 2011.)

A step can be for example, an arithmetic operation, an assignment or a test. If an algorithm takes the same amount of steps each time it is called, it requires constant time. This means that the size of the input will not affect the execution time. The input size is the parameter that has an impact on the amount of steps the algorithm takes. The efficiency is measured by how much the increase of the input size will affect the amount of steps. So the exact number of steps is nonessential. (Vernon 2005.)

Linear functions are written as $O(n)$ which means that time increases in direct proportion to the input size. Big O calculates the efficiency by using the scenario that will take the longest amount of time to run, also known as the worst case. (Programmer Interview 2017.)

The amount of steps taken can be shown as a function. For example, $f(n) = n$, means that function has an input size of n and the amount of steps will increase linearly related to the input. The most dominant part of the function will be the one used for comparison. This part is described by the order of magnitude function, also known as Big O notation. It is usually written as $O(f(n))$, where $f(n)$ represents the dominant part of the function. (Miller & Ranum 2011.)

The example code (Picture 4) is written out as $T(n) = 2 + 2n^2 + n$. The constant is dropped as it is insignificant as the n gets large. This leaves $T(n) = 2n^2 + n$, and it is easily seen that n^2 is the most dominant part of the function. Therefore its complexity is marked as $O(n^2)$. The coefficient of the n^2 can also be ignored as it becomes insignificant as the n gets large. (Miller & Ranum 2011.)

```
a = 10;
b = 4;

for(i=0; i<n; i++){
    for(j=0; j<n; j++){
        c = i + j;
        d = j - i;
    }
}

for(k=0; k<n; k++){
    e = k - a;
}
```

Picture 4. Example code for $O(n^2)$.

$O(\log n)$ expresses that the number of steps increase logarithmically. Logarithmic growth can be explained with binary search. Binary search is a technique for finding a specific element inside a sorted data set. The data set is halved by comparing the median of the elements to the target value. If the target value is higher, then data set half with higher values is split and another median gets compared to the target value. If the target value this time happens to be lower than the median, then the data set half with lower values is split. This continues until the correct value is found or the data cannot be halved anymore. A simple example of logarithmic growth would be a for loop that doubles its count on every iteration. This kind of loop is very fast to execute even as the n gets large.

That is why algorithms with complexity of $O(\log n)$ are really efficient with large data sets. (Bell 2017.)

The most common complexities, excluding constant time, are shown in the following graph (Figure 1). It is easy to see that an algorithm working with a small dataset might not work with a larger one. That is why it is important to determine the efficiency in order to find out if the algorithm will work with larger input values.

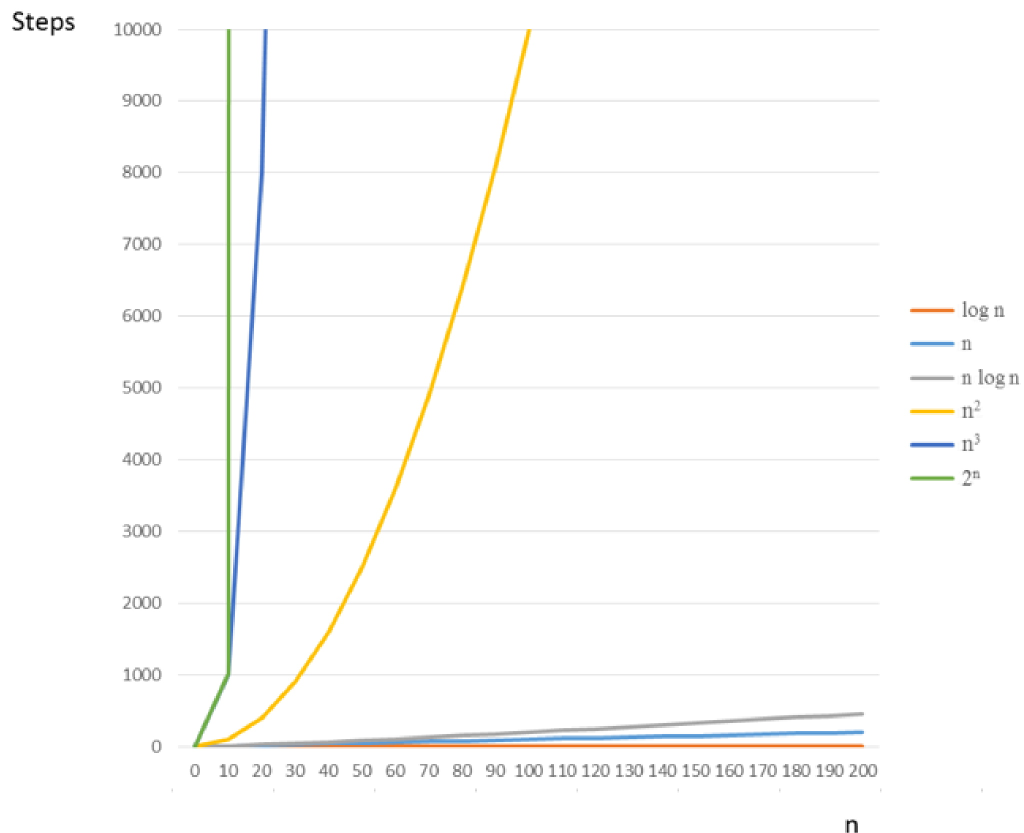


Figure 1. The most common algorithm complexities.

It is important to remember that constants and low-order terms are ignored when measuring efficiency. So two algorithms can have the same complexity with one of them still being executed faster. For example, algorithm 1 has a complexity of $O(n^2)$ and algorithm 2 is $O(3n^2 + n + 200)$. While they both have the same efficiency, algorithm 1 will be executed faster. (Vernon 2005.)

3 BUILDING THE WARPLANE AI

This thesis was a part of an ongoing game project inspired by an old MS-DOS game called Triplane Turmoil. It was released in 1996 by a Finnish game company called Dodekaedron Software Creations. The game is a 2D side-scroller in which the player's goal is to pilot their own warplane and lead the troops to victory against the enemy force. The game has both single-player and multiplayer modes.

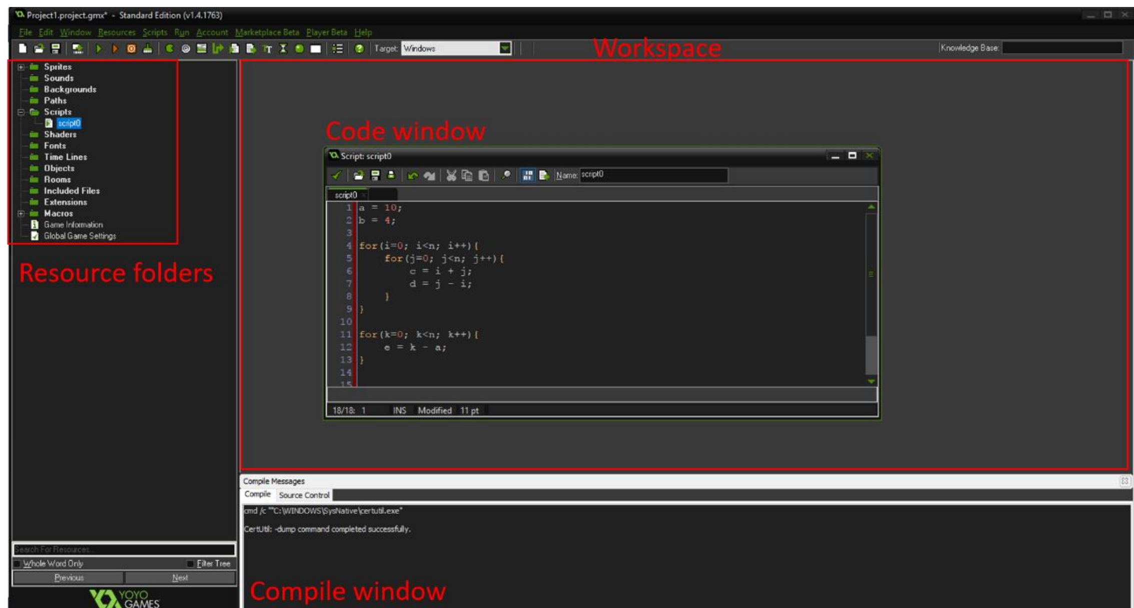
The game has 4 different types of planes to choose from. All of them have their own strengths and weaknesses. For example, lighter armor makes the plane more agile but also more vulnerable against bullets. In the single-player mode, the player has a variety of campaigns to play. Their goal is to outsmart and defeat the AI they get faced with. The aim of the thesis was to create the AI for the NPC warplanes in this mode.

The multiplayer mode supports up to 4 players. In this mode players fight against each other locally. The goal is to destroy the other players' planes with machine guns and bombs. After successfully destroying an enemy the player is awarded with a point. In case of getting shot down by an enemy or crashing their plane the player will lose a point.

3.1 Game Engine and language

The project uses GameMaker Studio 1.4, which is a 2D game engine made by YoYO Games. It has a built-in programming language (GML) based on the programming language C. GameMaker also has its own marketplace where users can create and download asset packages and extensions. (Alexander 2014.)

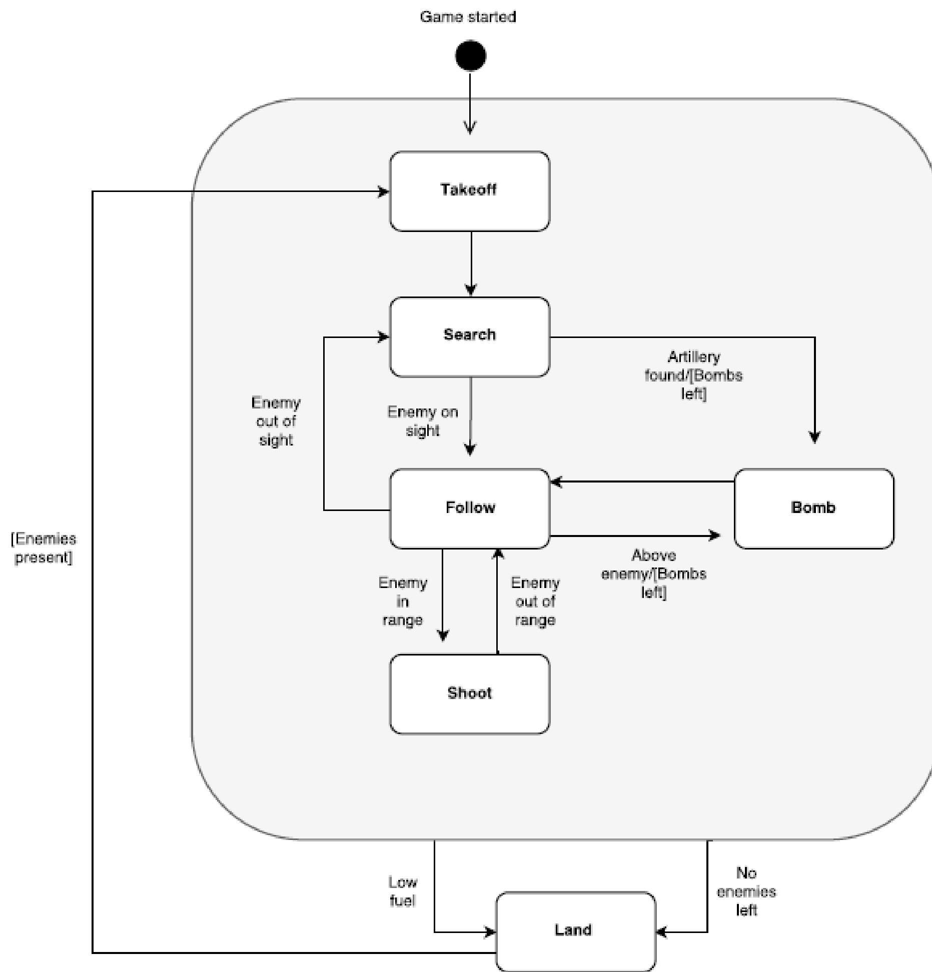
GameMaker has a quite simple layout which makes it easy to learn even for a new user (Picture 5). In the basic layout, the resource folders are on the left side. All of the objects, sprites, sounds etc. are stored in these folders. On the right is the workspace, where all of the active windows will open. For example, code files and rooms will open there. Rooms are the same as scenes in some other game engines. It is where the levels are designed and game objects get placed. The layout has a compile window at the bottom of the screen. The compile, as well as debug messages, are shown there as the game is run.



Picture 5. GameMaker's basic layout.

3.2 Planning

The goal of the thesis was to design and program an AI for one or more enemy warplanes that fight against the player, creating enough challenge for them. When the project started, the first task was to go through all of the wanted features for the AI. The client had a clear vision of what it should entail, which made the planning phase a lot easier. Once the fundamental requirements were agreed upon, the AI was divided into smaller pieces, which made up the different states of the state machine. This kind of division is easiest to do by drawing a diagram of it (Picture 6). This helped to see which states needed to be connected and where possible problems may arise.



Picture 6. State machine diagram for the AI.

The AI had 6 main states: takeoff, search, follow, bomb, shoot and land. The diagram shows a grey area around five of the states to indicate that all of those states will have a transition to the ‘land’ state. This is because the AI should be able to exit all of these five states whenever the ‘land’ conditions are met, instead of needing to go through a chain of them which could drain the fuel too much and make landing impossible.

When the game starts, an AI character is created that exits the hangar and starts taking off. After it has reached the desired height, the AI will move into a ‘search’ state. In this state it will start to look for enemy planes (the player) or other possible targets, like artillery. Once an enemy plane is spotted, the AI will start to follow it and try to get close enough to shoot it or explode it with a bomb. Once the plane is out of fuel or all enemies have been destroyed, it will land back on the airstrip it started from. This is also where it

will stock back up with bombs and machine gun ammo. Once the plane has refueled and there are enemies present, the plane will take off and move through the state machine again.

3.3 First build-test-evaluate cycle

As it was easiest and most logical, the programming was started with one state at a time beginning with the takeoff. Because the game development had already started before this thesis, there were already custom physics created along with some basic mechanics for controlling the planes. There was no need to think about how the acceleration should work or how much the air resistance or gravity would affect the plane. This is why creating the 'takeoff' state was a good introduction to the way the controlling mechanics were built and how they should be utilized.

In the 'search' state the plane had to fly from one end of the screen to the other, while being able to transition to the 'follow' state as soon as an enemy enters its field of vision. Since the plane would have needed some sort of pathfinding method in the 'follow' state, it was first tested out in the less complicated 'search' state. GameMaker has its own pathfinding functions so there was no need to implement an A* algorithm from scratch. The functions are able to calculate the shortest route from point A to point B while avoiding any obstacles in the way. This is done by dividing the room into smaller pieces with a grid. The room being the scene where the game level is created. Mp_grid_create function allows the cell count of a grid to be adjusted for the precision required. The smaller the cells, the more efficient the path is since the obstacle can be passed by more closely (Picture 7).

Picture 7. Examples of grids divided into cells of 25 and 10 pixels.

In the 'search' state, the function calculates the path to the other end of the screen, and once it is reached does the same operation back to the start point. This resulted in a

really smoothly moving plane that would stay on the shortest path between the two ends of the screen. There is however a risk with using the ready-made pathfinding functions. If the path is not destroyed before a new one is created, it may lead to memory leaks.

The advantage of using these functions in the ‘follow’ state was that the plane would avoid any obstacles while closely following the enemy. The disadvantage was that it needed to recalculate the path constantly which was not really that efficient anymore. It also raised the chance for memory leaks considerably.

3.3.1 Client feedback

As the progress was shown and discussed with the client, a decision on removing the pathfinding functions was made. This was because the plane would not utilize the basic control mechanics built into the game, which lead to the plane not being affected by the physics as planned. As a result, the plane would move more as if it was controlled by a robot than a human pilot. Also, the shortest path calculated by the function was not always the smartest way to approach the enemy. The plane should rather move according to the action it is going to take. When using a machine gun, the plane should turn to aim at the enemy as they reach the shooting distance. On the other hand, if the plane is going to use bombs, it should not move straight towards the enemy. A better tactic would be trying to get above the enemy, or to turn while releasing the bombs so that they will arch through the air.

In conclusion, the approach needed to be changed in order to make the AI act more naturally and the plane to be affected by the custom physics already implemented in the game. As was mentioned before, the chance of memory leaks affirmed this decision.

3.4 Second build-test-evaluate cycle

The problems with the paths in the first version were solved by defining boundaries for the permitted altitudes. In that way the plane would not fly too high which would cause them to come crashing down. Naturally the plane should also be able to avoid the terrain even if the enemy tries to lure them too close. By removing all of the pathfinding functions and utilizing the mechanics already in place, the plane had a more humanlike movement. Instead of following a predetermined path the AI would do correcting movements as it

reached a boundary. Once the plane got too close to the terrain or too much altitude, the AI would move up or down depending on the appropriate action.

In the second version the ‘land’ state was created. During this state, the plane would check the distance to its airstrip and with that information calculate when to start reducing speed and descending. The challenging part of this was that the formula needed to work with every type of plane which would have their own characteristics in speed and maneuverability. The first and easiest solution was to calculate at which rate the speed should decrease in order for it to be at the right level for the plane to descend safely (Equation 1).

$$divisor = \frac{|homeStrip.x - x|}{speed}$$

$$a = \frac{(1 - speed)}{divisor}$$

$$speed = speed + a$$

Equation 1. First version of the deceleration formula.

This formula was tested with every plane type and distance, and it always gave the desired results.

The ‘follow’ state underwent some major changes since the pathfinding functions were removed. The plane needed a new way to get the correct angle so that the aiming would work as it did before. As the AI starts to follow an enemy and reaches the shooting distance, it should turn to aim at them. This was done by finding out the angle from the plane to the enemy with the point_direction function. The function has input parameters for the x and y axis values of the starting point of the angle, in this case the plane, as well as of the target point (the enemy). The angle returned by this function can be used to check the position of the enemy in relation to the position of the plane. This information can then be used to make a decision of what action to make. If the direction of the plane matches the calculated angle, it means that the enemy is directly in front of the plane. If it also happens to be inside the shooting distance, it can then be shot. If the direction does not match the current angle, then the AI plane needs to correct where it is pointing, or even turn the plane around in order to be able to fire at the enemy plane.

Making the plane to turn around correctly was surprisingly troublesome. The planes needed to slow down before starting to turn or otherwise they would quickly move too high or low which resulted in crashes. Other issues surfaced once trying to correct the direction of the plane while it was facing to the right. This is because of the way the directions are defined inside GameMaker.

When the plane is moving left, it is easy to define that the plane should turn up if the direction is higher than 170. Trying the same kind of limitation by just changing the angle to 10, will not work when the object is facing right. That is because the direction will be 0 when facing straight towards right. In order to make the plane turn towards the specific direction, some extra conditions were required.

3.4.1 Client feedback

The client was happier with the movement this time around as the plane was affected by the physics. What they wanted to change this time was the way the plane landed, since the speed was directly decreased instead of letting the plane's own drag do that. It was decided to modify the formula into using the existing drag values. This meant that instead of working out the rate that the speed should be decreased, the distance needed to slow the plane down should be calculated. This new formula had to be created in a way that it would be affected by the conditions the plane was facing. For example, the higher the speed is the longer the deceleration distance needs to be, since it will take more time to get the plane slowed down enough to start the landing process. One other important factor is the angle of the plane, since it directly affects the amount of drag experienced. A larger angle means more drag since the air resistance will have more area to push against.

3.5 Third build-test-evaluate cycle

Creating a working formula for calculating the distance needed to get the plane to slow down to a desired speed was really critical. That is because without the formula the slowing distance would need to be hardcoded, which would make the code really inflexible. Even minor changes to the mechanics or physics would mean that all of it would have to be programmed again.

Variables that needed to be taken into consideration while designing, were the forces affecting the speed positively or negatively. The first task was to check the physics and see what the formula needed to entail. The physics of course include gravity which counters the lifting force. This needed to be taken into consideration since once the plane starts to slow down, the lifting force will grow smaller which leads to gravity starting to dominate. Once gravity had overpowered the other force, the plane will descend. This means that the plane would need some altitude before starting to slow down or otherwise it could crash.

Other opposite forces are the ones moving the plane forward and the ones resisting that movement. The drag is the amount of air resistance the plane is facing. The direction of the plane will affect the drag which is important to note when designing the formula. Drag is smallest as the plane is facing straight forward since the air has less surface to hit. And as the angle gets steeper the drag will increase, which leads to speed starting to reduce.

Values needed for the formula are the current speed of the plane, the desired end speed and the amount of drag the plane is facing (Equation 2). Gravity was not included in the formula itself, but it was taken into consideration by making the plane reach an ideal altitude so that it has some clearance for the descend. The formula gives the distance needed to slow down to the specific speed. SpeedDif represents the difference between the current speed and the desired end speed as the plane is moving down to the airstrip.

$$slowDownPos = \left(\frac{1}{drag}\right) * speedDif * speed - (speedDif)^2 * 0.5 * \left(\frac{1}{drag}\right)$$

Equation 2. Second version of the deceleration formula.

After the formula was tested with every type of plane and proven to be working as intended, some other fixes were done for the landing state. The code was rearranged, which lead to the formula not working anymore. This was quite unexpected and the cause of the issue was tricky to find. As was mentioned before, the angle of the plane will affect the drag considerably and because of the changes in the code structure, the drag values were checked too early. This meant that the values used in the calculation were higher than they should have been, resulting in overly short distances, which made the plane descend too fast. The plane would have too much speed and crash into the airstrip since the angle was a lot smaller than during the calculations. Once it was

ensured that the plane would be in the same angle as it is during the deceleration, the formula started to work again.

The plane’s way of turning around had some major changes as well because it was way too clumsy and slow. One of the major issues was the speed of the plane because if it became too fast it would either move too high and hit the limit or move downwards, have a too large turning circle and crash into the terrain. This was solved by alternating with the turning directions to keep the speed in a desired level, not too fast but not too slow either. If the plane lost too much speed, it would fall out of the sky since it did not have enough speed to do the turn. So if the plane gained too much speed it would start to decelerate before starting to turn. This also ensured that the plane would not lose too much speed if the enemy decides to keep circling the plane. If the speed got too slow, the plane would accelerate and gain enough speed before a turn. By modifying the turn state this way, it made the plane look more intelligent since it would turn faster and choose the turn direction based on their current speed and the position of the enemy.

At the end of the third version, all of the client’s requirements, that were illustrated in a diagram of the state machine, had been met. The plane was able to take off, search, follow, shoot, bomb and land. After this, the AI was ready for a small playtest.

4 RESULTS AND DISCUSSION

Since the single player campaigns had not been finished yet, playtesting was carried out by letting a player fight against one AI plane in the multiplayer map for about 15 minutes. In the multiplayer mode, players gain a point by eliminating an opponent and lose one when they get destroyed. All of the players ended up with a negative score because they crashed the plane multiple times before getting used to the controls (Table 1).

	Player score	AI score	Score difference	Previous experience
Player 1	-3	5	8	Yes
Player 2	-20	8	28	Yes
Player 3	-19	5	24	Yes
Player 4	-7	3	10	Yes
Player 5	-16	2	18	No
Player 6	-19	1	20	No

Table 1. Results of playtesting.

The players that had previous experience with the original game did not seem to have that much of an advantage. Players 1 and 4 got hang of the controls really quickly, which lead to them having a better score.

After playtesting, the players were asked whether the AI gave them a reasonable challenge in the first level of the game. Even though the score difference seems drastic, none of the players felt that the AI was too difficult. As the single player campaign is not ready yet, the game does not get any harder difficulty wise. So when looking at the results from the perspective of flow theory, there is no flow channel to move through at the time of this playtest. However, when considering this as a starting point for the channel, the difficulty of the AI was at an adequate level. The players had enough challenge without the game being frustrating.

It is important to note that the AI is not complete yet. It still needs some fine-tuning to make it more adaptable to different conditions as well as making sure that the difficulty level will increase at the right pace in the campaign. More playtesting will be done during the further development to make sure that the players will continue to experience flow.

When evaluating the code on the basis of scalability, most of the functions are O(1), meaning that they will not be affected by changes in the input parameters. This is because the AI does not have to gather or read through large amounts of data. If the AI would be more intelligent and learn from the player's actions, the outcome would be totally different.

Currently, the AI reacts to different condition changes and decides actions according to them. This allows it to be programmed with very low complexity since the actions and the triggering conditions are precisely defined.

Nevertheless, two functions with same complexity can still have different execution times. A good example would be the 'takeoff' state where the plane has two different ways to get to the desired altitude. This is decided by the AI checking if there is an enemy airstrip ahead of them in close proximity. If there is, it means the plane should gather some altitude before flying over the enemy's base in order to avoid getting shot down by the artillery. The AI would do this by turning the plane around once it had gathered enough speed and then turning back towards the original direction (Picture 8).

```
if(firstTurn){
    if(y > 120){
        if(direction > 0 && direction < 270){
            key_right = 1;
        }
        else{
            key_right = 0;
            if(speed > 2.9){
                firstTurn = false;
                secondTurn = true;
            }
        }
    }
    //If plane has too much altitude, turn downwards before turning up
    else{
        if(direction < 320){
            key_right = 1;
        }
        else{
            key_right = 0;
            if(speed > 2.9){
                firstTurn = false;
                secondTurn = true;
            }
        }
    }
}
//Turning back left
else{
    if(secondTurn){
        if(direction < 175 || direction > 270){
            key_right = -1;
        }
        else{
            key_right = 0;
            secondTurn = false;
        }
    }
    //Moving up
    else{
        if(y > 50){
            if(direction > 160){
                key_right = 1;
            }
            else{
                key_right = 0;
            }
        }
        else{
            takingOff = false;
        }
    }
}
```

Picture 8. Code for the takeoff with turning.

This kind of movement allows the plane to reach the desired altitude without being dangered by the enemy's artillery. If there is no enemy airstrip ahead or the artillery has been destroyed, the plane can just take off normally. These two scripts were both O(1), but they also had a different amount of lines of code. This means that one of the scripts would be executed faster than the other even though their complexity is the same. The takeoff script with turning had 24 possible steps. In the worst case the script would take 7 steps per execution and in the best case 4. The normal takeoff script has only 5 possible steps (Picture 9). In the worst case it will need 4 steps per execution and in the best case only 2. So in the worst case the normal takeoff script will take the same amount

of time as the other script in its best case. By creating two different methods for takeoff not only will the plane move more intelligently, but it will also save some execution time.

```
if(direction > 170){
    key_right = 1;
}
else{
    key_right = 0;

    if(y < 50){
        takingOff = false;
    }
}
```

Picture 9. Code for the normal takeoff.

The most inefficient AI functions are O(n), meaning that the execution time will grow linearly with the input data. A good example is the ‘search’ state, where the plane looks for the nearest enemy artillery in order to know when to release the bombs. GameMaker has its own function that returns the specified object closest to the given position. This function will go through all of the instances of that object inside the level. As the amount of artillery objects increase in the scene, so does the number of checks the function needs to go through. Even though the script would otherwise be O(1), this search function will have the most dominant complexity value and therefore give the whole script linear scalability. Since the number of artillery in one scene will not get that high, this will not create any efficiency problems. So as for scalability, there should not be any issues when the AI gets developed further.

5 CONCLUSION

The goal of the thesis was to create an AI for warplanes in a 2D side-scroller. The thesis was a part of an ongoing game project developed by game company MansikkaMarmeladi.io. The project uses GameMaker as a game engine and the code is written with the built-in programming language. The AI was created by using a finite state machine model because all of the wanted actions were easily divided into states. At the end of the thesis, the plane was able to take off, search, follow, shoot, bomb and land. This meets the first objective of the thesis where the plane should be able to react appropriately to environmental cues from the gameworld.

The code was analyzed from the perspective of scalability and the difficulty level created for the player. Most of the functions were O(1), meaning that they will have the same execution time regardless of the size of the input. The highest complexities were O(n), where the execution time will increase linearly with the input size. As a complexity value, this is still very low and will not cause any issues in the project since the size of the input will not get that large. All in all, as far as efficiency and scalability are concerned, the conclusion is that the algorithms will remain efficient as the development continues.

A small playtest was carried out to evaluate the difficulty level of the game. The results showed that there is a good starting point for further development as the players felt that the AI created a reasonable challenge for them. It is important to note that the AI is not completely finished yet. It will be developed further by the client in order to make it adapt to other possible events. Some further testing will be needed in the future so that the difficulty will stay at the right level and the players will keep experiencing flow.

Even though all the original requirements were met and the AI did what it was supposed to, some different design choices could have been made. The AI could have been designed by combining finite state machines with behavior trees. This would have made the structure more adaptable and easier to customize.

REFERENCES

Alexander, M. 2014. GameMaker: Studio 1.4 Features. Consulted 10.7.2017 https://www.yoyogames.com/blog/61/gamemaker-studio-1-4-features

Baron S. 2012. Cognitive Flow: The Psychology of Great Game Design. Consulted 9.4.2017 http://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php.

Bell, R. 2017. A beginner's guide to Big O notation. Consulted 25.4.2017 https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

Buckland M. 2005. Programming Game AI by Example. Plano: Wordware Publishing, Inc.

Kirby N. 2010. Introduction to Game AI. Andover: Course Technology / Cengage Learning.

Miller, B. & Ranum, D. 2011. Big-O Notation. Consulted 17.4.2017 http://interactivepython.org/runestone/static/pythonds/AlgorithmAnalysis/BigONotation.html

Millington I. 2006. Artificial Intelligence for Games. Boca Raton: CRC Press.

Murphy C. 2011. Why Games Work and the Science of Learning. Consulted 10.4.2017 http://www.goodgamesbydesign.com/Files/WhyGamesWork_TheScienceOfLearning_CMurphy_2011.pdf.

Patel A. 2017. Introduction to A*. Consulted 8.4.2017 http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

Programmer Interview 2017. How does Big-O Notation work, and can you provide an example?. Consulted 4.4.2017 http://www.programmerinterview.com/index.php/data-structures/big-o-notation/

Russell S. J. & Norvig P. 1995. Artificial Intelligence: A Modern Approach. New Jersey: Prentice-Hall Inc. A Simon and Schuster Company.

Schell J. 2008. The Art of Game Design. Burlington: Morgan Kaufmann Publishers.

Slabinski M. 2013. Designing Games with Flow in Mind. Consulted 9.4.2017 http://www.gamasutra.com/blogs/MarkSlabinski/20130414/190449/Designing_Games_with_Flow_in_Mind.php.

Vernon, M. 2005. Complexity and Big-O Notation. Consulted 17.4.2017 http://pages.cs.wisc.edu/~vernon/cs367/notes/3.COMPLEXITY.html