A third-person shooter game

Final Report for CS39440 Major Project

Author: Jerzy Krzysztof Leszczynski (jkl1@aber.ac.uk)
Supervisor: Dr. Bernard Tiddeman (bpt@aber.ac.uk)

21th April 2013
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in
Computer Graphics, Vision and Games (G450)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature .....

Date .....

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature .....

Date .....

Acknowledgements

I would like to thank my supervisor, Dr Bernard Tiddeman, for organising frequent meetings monitoring the progress made and general help regarding the major project. I have greatly benefited from his guidance and support throughout my final year.

I would also like to thank my friends who have helped me during this project. In particular, I would like to thank Adrianna Glowacka for her support and encouragement, and Karen Jones for her help with proofreading.

Abstract

Over the years, computer and console games have achieved a high level of photorealism and sophistication of the behaviour of non-player characters. However, due to the hardware limitations of mobile devices, there is a noticeable lack of games featuring intelligent enemies in mobile game markets.

In order to develop a highly competitive game, a number of artificial intelligence techniques have been implemented and optimised specifically for devices of lower computational power to create realistically acting agents. These techniques include simple, as well as more complex methods, based on relatively recent publications and approaches used by advanced and well-known computer and console games. The agents' decision-making process is controlled by finite state machines and involves the use of fuzzy logic and terrain analysis. Moreover, the game implements a robust navigation system capable of moving agents under various conditions and coordinated enemy formations using a method inspired by the Markov process for danger assessment.

Due to the time constraints, it was not the objective of this project to develop a fully complete game consisting of multiple maps, but rather one basic level featuring all the created agents which will be extended in the future.

Contents

List of Figures

List of Tables

Chapter 1

Background & Objectives

1.1 Computer games

Video games throughout the years have become one of the most popular applications of computer technology. Despite the fact that the game industry is relatively small compared to other software industries, many recently developed games have production costs comparable to the cost of Hollywood movies [1].

Although games mainly contribute to the entertainment industry, there are game genres developed specifically for other purposes, such as educational games or serious games. It has been also suggested that games have become a new form of art [2].

Games are demanding of computer technology. In order for them to be appealing to the end users, they strive for:

- photorealism achieved by advanced lightning and shading techniques and high polygon number of 3D models,
- realistic interactions between objects usually simulated by a physics engine approximating interaction of objects in the real world,
- and intelligent enemies produced using various techniques inspired by or directly drawn from the field of artificial intelligence.

Due to the technological advancement of the past two decades, the first two aspects have been accomplished to the extent of satisfying the players. While the first games (*Arkanoid*, *Pong*) contained simple graphics based on rectangular shapes, the most recent games offer realistic virtual worlds containing breathtaking sceneries that begin to resemble the real world (*Crysis 2* [3]). However, as characters begin to look more realistic, players expect them to behave more credibly. This effect is commonly known as uncanny valley [1]. Once a visual anthropomorphism of characters is achieved, players may become distracted if the complexity of their behaviour does not match their appearance. For this reason, more emphasis has been placed on artificial intelligence in the development of contemporary games.

1.2 Artificial intelligence in games

1.2.1 Current situation in the video game industry

Many recently released PC and console games contain complex enemies acting similarly to real players. To make intelligent decisions, various techniques are used including: decision trees,

finite-state machines, rule-based systems and fuzzy logic [4]. In order for the enemies to move through the world avoiding obstacles, games implement various kinds of navigation systems, using different representation of search space (such as waypoint grid or navigation mesh) and search algorithms finding the shortest path between two points [5].

More advanced games [6] often use a concept of terrain analysis, a method evaluating various aspects of the terrain (such as provided cover or the distance from dangerous objects) in order for the enemies to act more intelligently, for instance by, choosing the best sniping location or using suppression fire.

Enemies are often capable of complex collaboration by making decisions as a group and individually (emergent cooperation) or shaping various formations. Some of the games even adapt real military tactics derived from training manuals of U.S. and other NATO countries military.

The combination of these techniques, produces a single complex emergent behaviour. This approach is referred to as multi-tier AI [4], where behaviours are structured in a multi-level hierarchy.

Most recent games often use some kind of dynamic difficulty balancing. This process ensures that a game is not too easy or too difficult (to prevent players from becoming bored or frustrated). For instance, *Left 4 Dead* (Valve) [7] contains an AI director estimating the emotional intensity of the players depending on their injuries or death of their team members. Based on this value, the number of spawned creatures is adjusted to maximise drama in the game.

Less often, but with equally interesting results, game developers implement some kind of learning techniques [8]. Examples include *Colin McRae Rally 2* (Codemasters), which uses a neural network to train opponents to drive realistically and *The Sims* (Electronic Arts) where social behaviour of non-player characters is based their interactions.

1.2.2 Mobile games and similar work

Due to the recent spread and technological advancement of mobile devices, the mobile game market has expanded to a size comparable to the console and PC game industry. However, due to the hardware constraints, mobile games are less advanced than their console and PC equivalents. The primary goal of game developers is the creation of games of high visual appeal. While the most recent mobile games feature graphics comparable to the graphics of the PC games (especially when viewed on the smaller resolution of mobile phones), there is often little processing power and memory left to be allocated for computations involving artificial intelligence.

Guerrilla Bob (Angry Mob Games) [9] is one of the most popular mobile third-person shooters. The game contains polished graphics and an interesting storyline. However, enemies exhibit simple behaviours, mostly based on following the player. While there is no information available about the techniques developers have used to implement this behaviour, it appears to be based on a simple state machine containing two states (handling attacking and following). It also contains an imprecise navigation system, as in some cases the enemies are unable to avoid simple obstacles.

Shadowgun [10], a third-person shooter developed by Madfinger Games, is widely considered to be the most advanced game released on mobile platforms. It contains highly polished graphics, animations, cut scenes, various weapons and a number of soundtracks. It also features fairly advanced artificial intelligence, involving enemies capable of mutual cooperation and seeking cover spots. However, some critics [11] point out that while the enemies' complexity is standing out from other mobile games, their actions are based on one pattern and are easily predictable.

1.3 Objectives

The aim of this project was to produce a cross-platform third-person shooter game. It was developed under a Software Alliance Wales programme and will be released on Android and IOS markets once it is finished.

As there is a noticeable lack of games featuring complex enemies in the mobile market, in order for the game to be highly competitive, a focus has been put specifically on this aspect. The game contains a number of enemy types, based on multi-tier AI structure, exhibiting different behaviours varying in complexity. While some of them have been implemented using simple approaches, others use more advanced techniques drawn from recent computer games and researches, adapted and optimised in order to work on mobile devices.

Since the project is mainly focused on the development of artificial intelligence of the enemies, the game consists of only one short level. However, when the core functionality of the game is finished (player and camera control, shooting, etc.) other levels will be easily and quickly added by reusing the existing functionality (for instance, by creating multiple instances of enemies and placing them throughout a map). Also, the game uses free, low quality 3D models, which will be replaced by a graphic artist in the later stages of the development.

Chapter 2

Development Process

2.1 Introduction

It is important for games to be innovative, as it is one of the factors affecting their sales volume. To achieve this, the development time should be as short as possible, otherwise a product may not be up to date with current industry trends. Moreover, the development cycle should allow and facilitate changes in requirements, as the industrial trends may change during the development, which requires introducing quick modifications to the game design. Additionally, due to the expansion of mobile platforms, the game projects are small in scale. For this reason, agile methodologies tend to be more appropriate than other, more traditional approaches, which are better suited for larger projects.

To develop the project, an agile methodology Scrum [12] has been used. As it meets the requirements stated above, it is considered to be one of the most well-suited methodologies for game development [13].

2.2 Scrum outline

The main artifact in Scrum is product backlog: a sorted list of product requirements. They are ordered based on various aspects, including risk, complexity, importance and date needed. These values are approximated to a single story point value usually using a rounded Fibonacci sequence.

Iterations in Scrum are called sprints. They usually have a fixed duration of two to four weeks. A sprint is preceded by a planning meeting, where the features for the current iteration (Sprint backlog) are chosen and evaluated in terms of their difficulty. It is followed by a retrospective meeting reviewing the progress made and reflecting on any possible improvements.

Scrum involves three core roles:

- Product owner - A person representing the customer. He or she is striving for the highest return on the investment by prioritising essential features in a product backlog.
- Development team - A group of 3 to 9 employees possessing skills required to achieve the objectives set out for a sprint. They are self-managing, as they decide what and how much work will be done in an iteration.
- Scrum Master - A person ensuring the work is done according to the Scrum principles by educating the members of the development team about the Scrum process.

2.3 Modifications

As, Scrum is a methodology involving a number of roles it needed to be adapted to suit a one person project. The developer looked at the work from a perspective of not only a development team, but also a product owner and a Scrum Master. Moreover, since the Scrum process involves a number of team meetings (e.g. sprint planning meeting, daily stand-up meeting, and review and retrospective meetings) instead of conducting them, the developer reflected on the features to be implemented and the work already done.

2.4 Implementation tools

As, the development of a 3D game is a time consuming process and it is not practical to build it without using any supporting tools, most game developers are using existing game engines (software specifically designed for the creation of games). Its functionality typically includes a rendering engine, physics engine and collision detection.

After evaluating a number of the most popular game engines, the one meeting all of the project requirements (such as support for mobile platforms and inexpensive license) has been chosen: Unity3D [14] .

2.4.1 Game engine description

Unity3D is a cross-platform game engine and development environment. It supports a number of platforms, including Andriod, IOS, Windows and game consoles. It provides various graphical features typically found in other game engines, such as occlusion culling, particle system, lightmapping, shadow generation and shader support. For physics simulation it uses a Nvidia PhysX engine. Despite its inexpensive license, it is considered to be one of the top quality engines. Unity has been used to develop a number of popular games (including Guerrilla Bob and Shadowgun), and it is used in other areas, such as serious game development [15] and robot design [16].

2.4.2 Implementation characteristics

Unity structure has been designed according to the game object component architecture [17] widely used in game development. This approach emphasises the idea of modularity, a separation of functionality into independent components that leads to a clean design, high interoperability and easy reuse of code. It has a different structure from a traditional object-oriented approach, as entities are mainly connected with a ”has a” relationship instead of an ”is a” relationship typical for inheritance.

Every entity in the scene is an instance of GameObject class. A game object contains a number of components, each providing some general kind of functionality. The most commonly used components are transform (storing position and rotation), rigidbody (physical attributes), animation and collider. Each component may contain a number of behaviours: scripts implementing more specific functionality. All behaviours derive from MonoBehaviour class and usually override its Awake function which is called when a component is activated and the Update function called in every game cycle.

2.4.3 Peculiarities

For scripting, Unity uses UnityScript [18], a language with syntax inspired by (but not conforming to) the ECMAScript specification. It is quite different from JavaScript (e.g. it has classes and global variables). It does not support abstract or generic classes. Moreover, since Unity is based on Mono, an open-source implementation of the .NET framework, it provides an access to a number of .NET libraries. It is useful, but causes confusion and results in inconsistencies in the naming convention.

Unity code may often appear to contain numerous, seemingly unnecessary, public member variables. This is due to three reasons:

- Parameter customisation - In order to tune certain details of a game, such as physical properties of objects or parameters of the enemy AI, some of the variables are made public and as a result are accessible from the Unity Inspector panel. It enables programmers and designers to change them in real time during the game execution.
- Component caching - As computational efficiency in game development is often an issue, it is important to use component caching. Instead of accessing components of objects in game through an accessor variable directly, the reference to the component is stored in a member variable to gain performance.
- Component dataflow - Components often need to exchange data with each other by accessing their fields. To facilitate this, components providing data store a reference to a component receiving data, and therefore have a direct access to the relevant variables.

Chapter 3

Game structure

The structure of all characters in the game (the player, as well as the enemies) has been based on character controller and motor movement components. Such an approach is widely used in Unity development [19], as can be seen in the sample project provided by Unity Technologies [20].

3.1 Component description

Main components:

- Character controller - A component managing character movement and actions. For example, to determine the direction of movement, a player controller simply reads keyboard input (arrow keys), while an enemy controller is driven by a state machine.
- Motor movement - In order for the characters to move realistically, it is not enough to determine the direction of movement. A character speed and rotation needs to be adjusted to make the movement smooth. For this reason, a character controller passes a direction vector to a motor movement component, which performs the actual translation and rotation of a controlled game object. As in some cases, characters need to be able to walk in other directions than they are facing (for example, while shooting and moving back from the player), movement and facing directions are stored separately.

As the player character requires different kind of control mechanism than enemy characters, and each enemy exhibits different behaviour, the project contains a number of controller and movement classes corresponding to individual character types. This structure is shown in Figure 3.1.

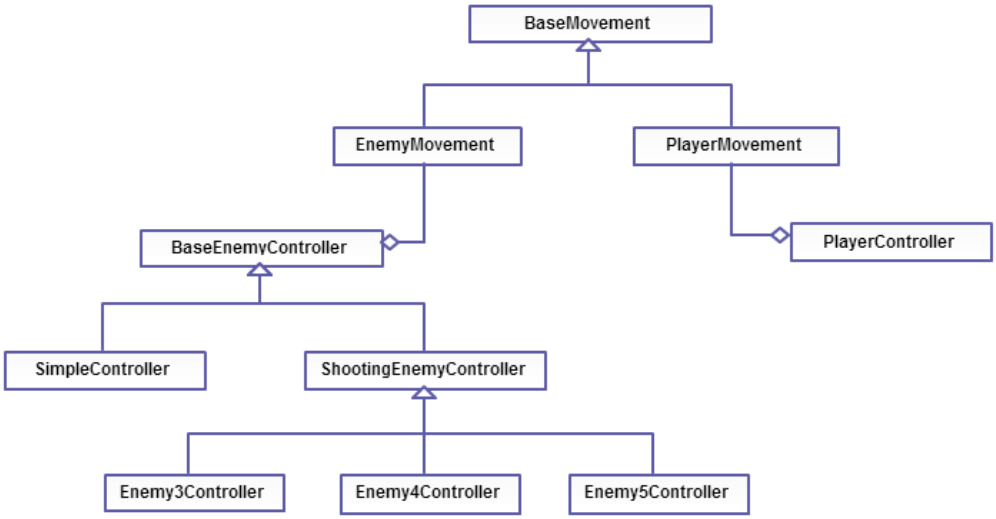

Figure 3.1: A diagram presenting relationships between character controller and motor movement classes. The names of variables and functions have been omitted to improve the readability of the diagram.

Other components:

- Health component - Monitors the level of a character's health-points. Depending on this value, it executes specific actions. For instance, if the number of health-points is equal to zero, the character is removed from the scene.
- Glow component - Smoothly changes the colour of a character model to a given value (red by default) and then changes it back to its original value. It creates a 'glow' effect used mainly when a character is being hit by a bullet. It is activated by a health component.
- Detector component - Used by enemy characters. It activates the character controller when the player is in close proximity.
- Destroy component - Removes the character from the scene. It is activated by a health component.

Component structure of the player character is very similar to the enemy structure, with few exceptions (for example, it does not use a detector component). The controller and movement scripts have been based on the code from the Unity sample project. The controller component, depending on platform moves the player based on the input from the keyboard and mouse (PC), on-screen thumbsticks (Android and iOS) or a console controller (XBOX 360 and PS3). To manage animations as the player character is temporarily represented using a 3D model and animations from the sample project, the game uses an animation script from that project.

3.2 Communication

Components communicate with each other in two ways:

- Direct variable change - A component continuously passing data to another component is given direct access to a relevant field to reduce the performance overhead. For instance the

character controller is constantly changing the variable storing movement direction vector of the motor movement component.

- Messaging system - In a situation where a component needs to trigger an event or activate another component, an event based messaging system is used. Its implementation is based on SignalSender class from the Unity sample project.

The component diagram shown in Figure 3.2 represents a typical component structure of an enemy character.

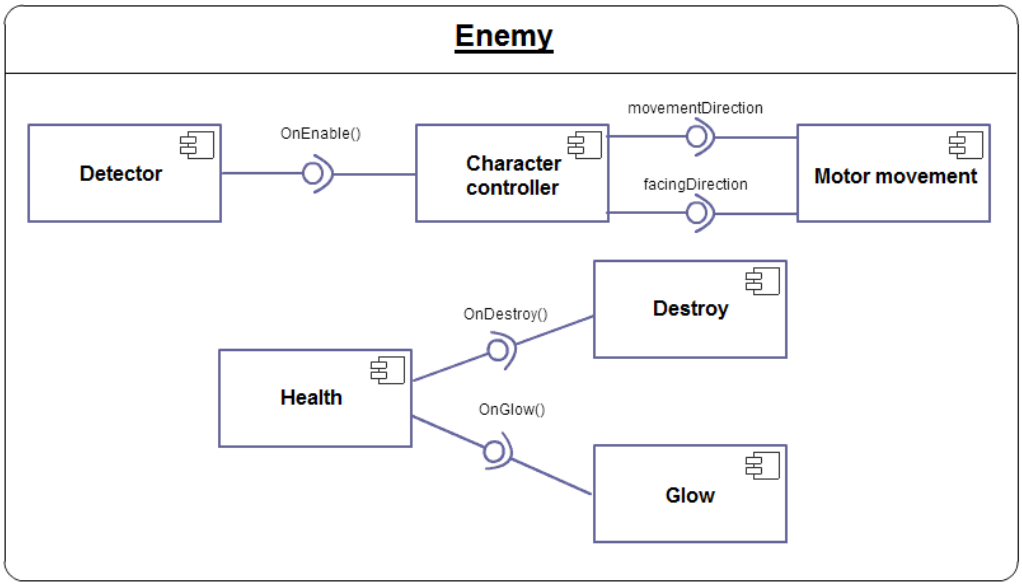

Figure 3.2: A component diagram presenting relationships between different components used by characters in the game. The annotations above the connectors indicate a method or variable that is used for data exchange.

3.3 Weapons

Characters also use a number of components related to weapons and shooting. However, in accordance to the Unity conventions, they are not attached to character game object directly, but rather to its 3D model (more specifically to the right hand). Their implementation has been partially based on similar components from the Unity sample project. The overall structure of the weapon system is shown in Figure 3.3.

All weapon-related components are contained in a Weapon slot game object:

- Weapon Handler - A component providing access to basic weapon functions, such as weapon changing or firing.
- TriggerOnMouse - A component reading data from input controller (e.g. mouse) and calling a weapon handler to initialise shooting.

Weapon slot may store a number of weapon game objects representing different weapons (e.g. pistol, rocket launcher, etc.), each containing the following components:

- Auto-Fire - A component instantiating bullets and applying damage to characters hit by them.
- Weapon Info - Stores various information about a weapon including, the initial amount of ammunition, the amount of ammunition left, the amount of time it takes for the weapon to load, and the frequency of firing bullets.
- Per frame raycast - A script detecting if there is any object in the line of fire. It is executed during every frame.
- Muzzle flash - A script creating a muzzle flash effect when a bullet is being fired.

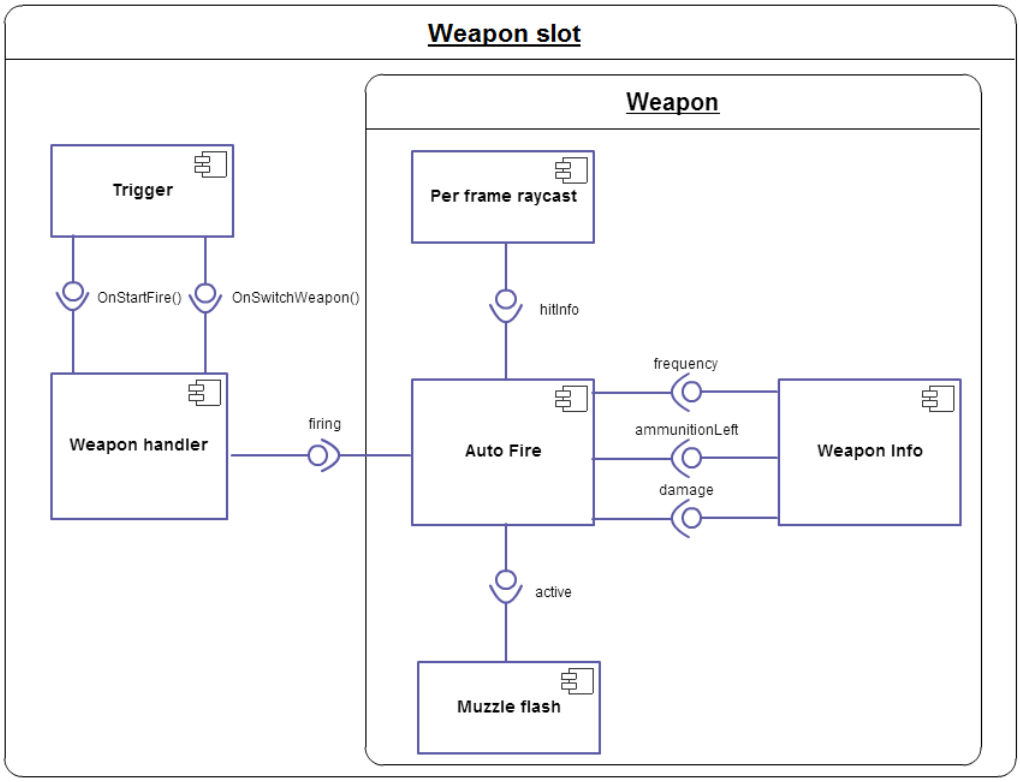

Figure 3.3: A component diagram presenting a structure of a weapon system used by each character.

3.4 Visual style

Since the project is not focused on the development of visual aspects, the game uses free models obtained from Asset Store (a facility built-in Unity Environment) and the Unity sample project.

The figure below presents 3D models representing player character, the first and the second enemy type. Similarly to the second enemy model the other enemy models are copied versions of the player model with changed colour.

Figure 3.4: Models representing the player character (a), first enemy type (b) and second enemy type (c).

The game level is based on a seaside scenery built from simple models of rocks, water and sandy terrain as shown in the figure below.

Figure 3.5: An image presenting a part of the game scenery.

Chapter 4

Artificial intelligence

4.1 Overview

The game contains five enemy types of different difficulties. Behaviour of each enemy is based upon a finite-state machine. Initially, each enemy is inactive. When the player is discovered by a detector component, an enemy controller is activated and its finite-state machine is started. The game includes following enemies:

- Simple enemy

  The only enemy that is unable to shoot. It follows the player, and when it is very close, it starts to attack.

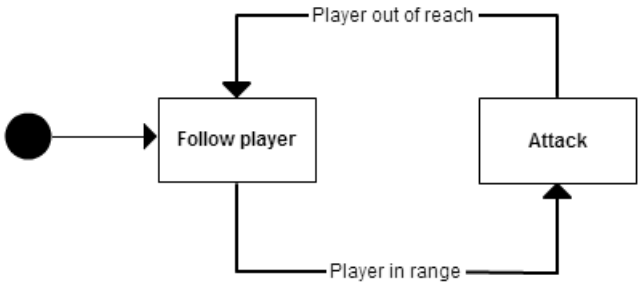

Figure 4.1: A state diagram presenting a state machine controlling the first enemy type.

- Shooting enemy

  As the first enemy, it follows the player until it is in a certain range and begins to attack (in this case by shooting). If the player is too close, it moves back.

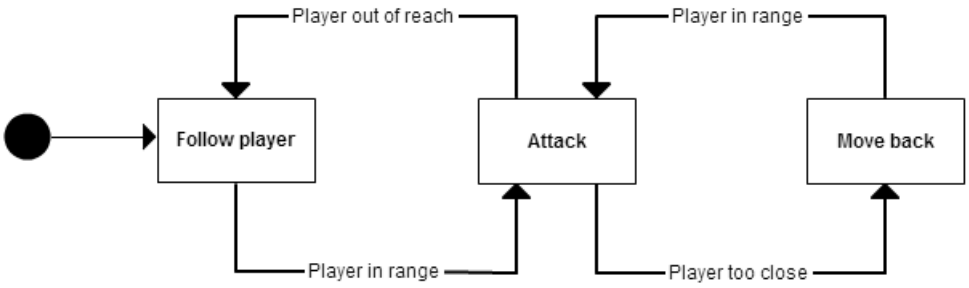

Figure 4.2: A state diagram presenting a state machine controlling the second enemy.

- Enemy dodging bullets

  This enemy is essentially the same as the previous one with one addition - when it is hit by a bullet, it moves out from the line of fire.

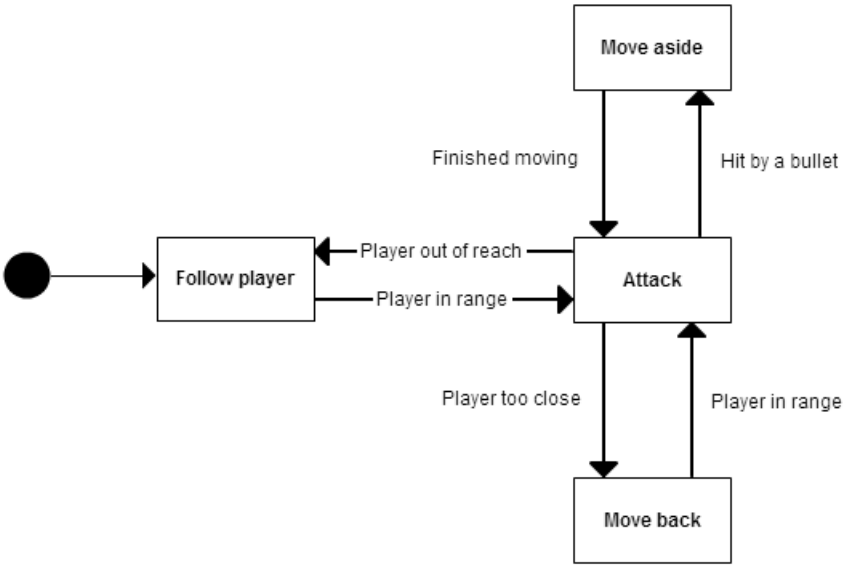

Figure 4.3: A state diagram presenting a state machine controlling the third enemy type reacting to the player attack.

- Enemy with fuzzy logic based weapon management

  The structure of this enemy state machine is similar to the structure of the second enemy state machine. However, its attack state involves more complex functionality implemented using fuzzy logic. As this enemy has two weapons, the attack state chooses an appropriate weapon for a given situation. It also calculates shooting accuracy based on such parameters as an enemy's health and distance to the player.

- Enemy based on terrain analysis

  The enemies listed above exhibit engaging yet quite simple behaviour following easily predictable patterns. Their actions are closely linked with the player's actions. For instance,

when the player steps back, the enemy automatically moves forward (towards the player). In this regard, they resemble agents based on the reactive paradigm, widely used in robotics where actions are directly induced by sensory data. This enemy appears to act more intelligently. Its action are based on spatial analysis of the nearby environment which takes into account such details as distance from dangerous objects and provided cover.

The enemy starts by searching for a safe location (hide state). Then it loads the weapon (wait state), leaves the safe spot to track the player (track state) and starts to shoot (attack state). When a weapon needs to be reloaded, the enemy hides again.

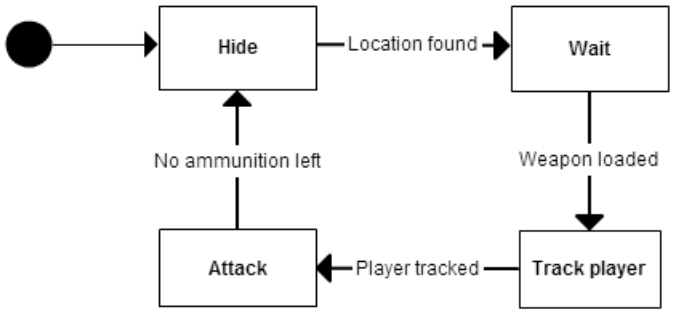

Figure 4.4: A state diagram presenting a state machine controlling the enemy based on terrain analysis.

4.2 Finite-state machines

4.2.1 Introduction

A concept of a finite-state machine is widely used in game development to manage behaviour of entities. It consists of a number of states connected by transitions. Each transition has a set of conditions that when met, result in the activation of the transition's target state. The active state defines actions performed by a character.

State machines have a number of advantages. Thanks to their simple structure, they are easy to implement and debug. As their logic is based on hard-coded conditions, they are easy to process what decreases the computational overhead. Moreover, their functionality can be extended with little effort by adding new states or transitions.

4.2.2 Design and implementation

There is no single correct implementation of a state machine used in games, as its design is dependent on specific game needs. In this project a design proposed in Chapter 10 of the book by Buckland [21] has been used.

Each state is represented by an individual class and contains Execute function defining state-specific behaviour and determining the state transitions. State classes also contain Enter and Exit functions invoked when a state is being switched on or switched off.

A StateMachine class keeps track of a currently active state in the currentState field. It uses the Update function to invoke its Execute function during every frame. It also contains a Configure function initialising the state machine with a given state, and a ChangeState function changing the

active state. StateMachine class also stores a reference to a controller of an agent using the state machine, and passes it to the active state as a parameter of the Execute function. It provides states with access to information related to the controlled character, which is needed for evaluation of transition rules. The state machine is stored in a character controller, which invokes its Update functions during every frame.

This implementation is based on the state design pattern. As the states are encapsulated as objects with embedded transition logic, state machines are self-contained and independent from other parts of the system. This approach is much more scalable than incorporating character behaviour and transition logic in a monolithic switch statement.

As the game may contain a number of agents using state machines, they are likely to share the same states. For this reason, individual states have been implemented as singleton classes. A singleton pattern restricts the instantiation of a class to one object. Each state class has a static function Instance providing an access to a variable storing the reference to the instance of the object. Upon the first invocation of the Instance function, a class is instantiated. Subsequent invocations result in returning a reference of a previously created object. The use of the singleton pattern reduces a performance overhead, as it prevents new objects from instantiating when a state change is made. A class diagram presenting the implementation of state machines is shown in the figure below.

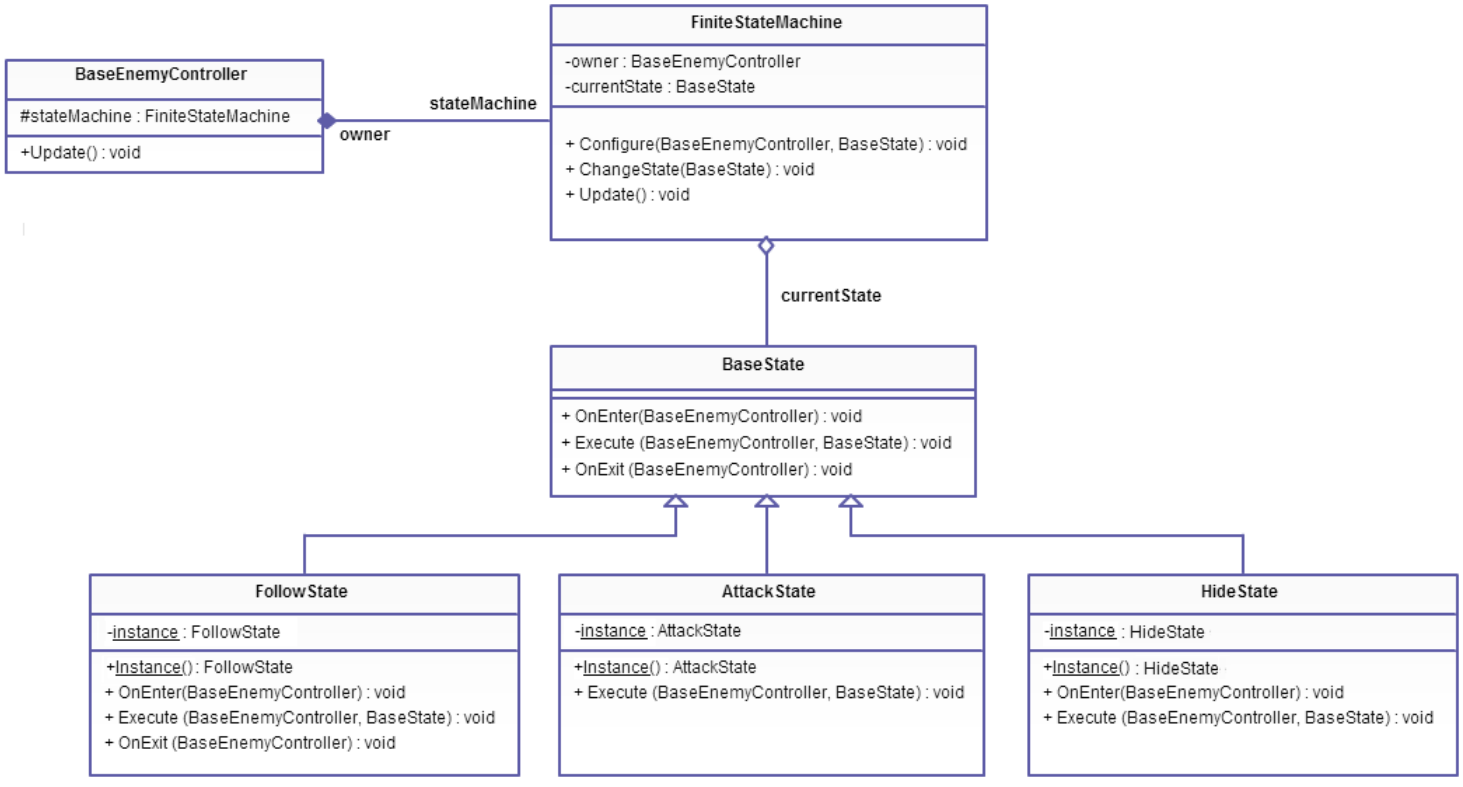

Figure 4.5: A class diagram presenting classes implementing the state machines. The diagram is simplified. The game contains 12 different state classes and presenting all of them in one diagram would make it difficult to read. Also, variables and functions of BaseEnemyController class, which are not related to state machines have been omitted.

4.2.3 Design considerations

Since the transition logic is hard-coded in the form of if-statements, the scalability of this implementation is decreased. When a certain state needs to be used with a different set of transition rules, a new state class needs to be created with partially duplicated code. To avoid this, another approach has been proposed in Chapter 5 of the book by Millington [4]. It solves this problem by separating the transition logic from the state classes by creating classes representing transitions. However, this results in a performance overhead and requires implementing a set of classes supporting different kinds of transition rules, which outweighs the advantages that could have been gained.

Both of the proposed designs present a state machine based on polling. This solution is not optimal, as transition rules of an active state are evaluated in every game cycle. Moreover, it is difficult to change the state due to external events. In such a case, an event-driven state machine would be more suitable. However, at this point, this project does not require such functionality.

4.3 Navigation

In order for agents to appear intelligent, it is important for them to realistically navigate through the game world. This task consists of three stages: pathfinding, path following and local obstacle avoidance.

4.3.1 World representation

Pathfinding is a task of finding a route between two points. To accomplish this, a pathfinding function needs to take into account obstacles present in the game world. For this reason, a suitable world representation needs to be chosen to indicate the ”walkable” space. It is stored in a graph data structure, traversable by a search algorithm. If a path between two points exists, an algorithm returns a number of nodes (usually points in space) making up a route.

A correct choice of a search space is important as it has a direct influence on the robustness of the navigation system. Generally, the search spaces used for pathfinding are represented by a graph (a data structure consisting of nodes connected by edges). The larger the game world is, the more nodes and edges the graph contains. As the memory usage is proportional to the number of nodes, it might be problematic to store a graph representing a large world, especially on mobile devices. Moreover, the size of a graph affects the computational performance of a system. To find a path in a large graph, a search algorithm needs to traverse more nodes resulting in a slower search. For these reasons, the search space needs to be represented by as few nodes as possible.

Another important factor to consider is the way in which the search space is created. It can either be generated automatically or created manually by a designer. As the automatic generation is often difficult to implement or too computationally expensive, many games use the second approach. However, it has several drawbacks. A designer may place the nodes inaccurately or omit some important locations, especially if the game world is large. It may possibly introduce bugs in the AI, for instance it may limit an agent’s walking space or result in awkward movement. Moreover, it reduces the scalability of a game. Each time a world structure is changed (by adding or removing obstacles), nodes need to be updated as well. It may be an issue, especially if the world can be modified in real-time during game execution or if it is generated automatically.

Three different approaches described below have been considered during the development of this project.

Grid is one of the simplest representations of search space. It consists of closely placed cells (nodes), as can be seen in Figure 4.6. Their major advantage is random access lookup which is the ability to determine a cell occupying a given location (in this case in 2D coordinate system) in a constant time. For this reason, it is widely used in 2D strategic games, where the world itself is grid based.

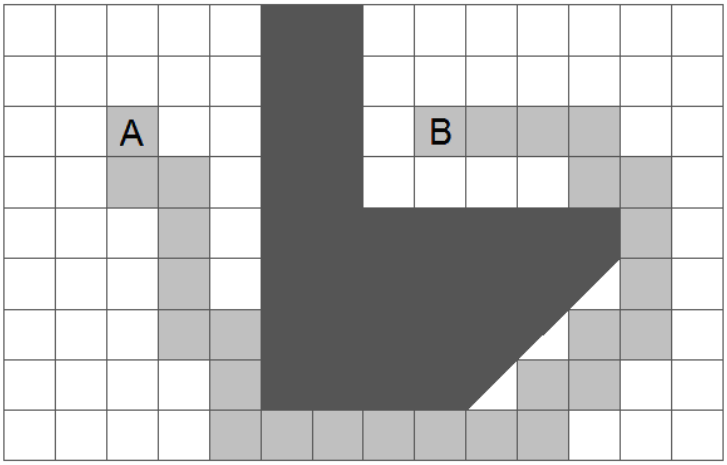

Figure 4.6: An example of a grid-based representation. A path from point A to point B is represented by light grey squares. Dark grey nodes represent an obstacle.

In most cases, it is a naive approach. To cover the whole game world, nodes need to be placed densely, which results in a massive graph. As its generation has a quadratic time complexity, which is not efficient especially, with such large number of nodes, the generation process may take a long time. To make it less computationally exhaustive, the node density (grid resolution) may be decreased, resulting in a reduced number of nodes. However, it may cause gaps in the search space, as shown in the figure below.

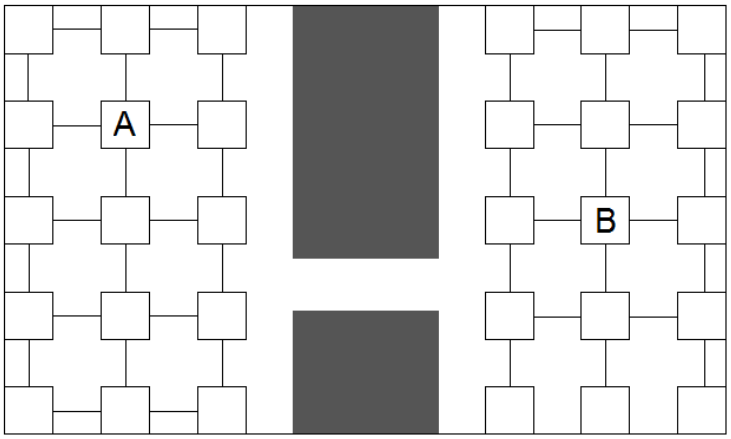

Figure 4.7: A world represented by a grid consisting of sparsely placed nodes (the graph edges are represented by thin lines). As the gap between the obstacles is small, there is no node placed there, thus a route between points A and B does not exist.

Another world representation is a waypoint graph, consisting of nodes (waypoints) manually placed by a designer. An example of such a graph is presented in the figure below.

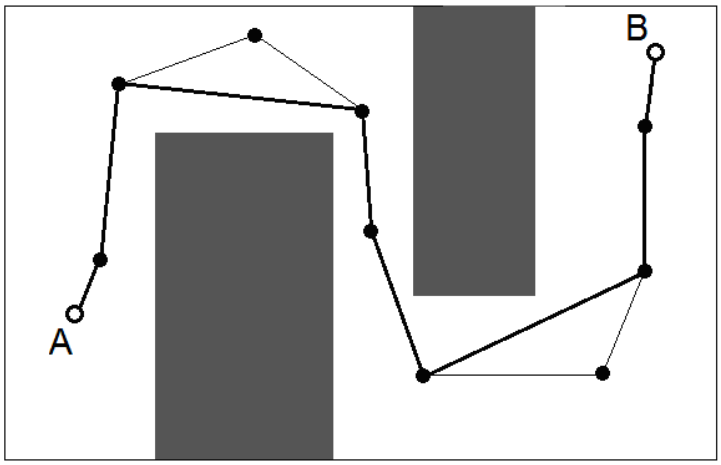

Figure 4.8: An example of a waypoint graph. Thin lines represent the graph edges and a thick line represents found route from point A to B.

A number of nodes in a waypoint graph is significantly lower than in a grid that results in quicker graph traversal and lower memory usage. However, this approach has a number of disadvantages related to the manual generation, which have been mentioned above. In addition, as the nodes are placed manually, they are not connected and therefore require further processing. To connect all the nodes, each possible pair of nodes needs to be checked to determine if they are in the line of sight. This is an extensive test, as it has quadratic time complexity.

The third considered representation is a relatively new approach in games development, a navigation mesh. In contrast to other approaches, it does not represent a space as a set of connected points. Instead, it consists of a number of convex polygons defining the ”walkable” space, as shown below.

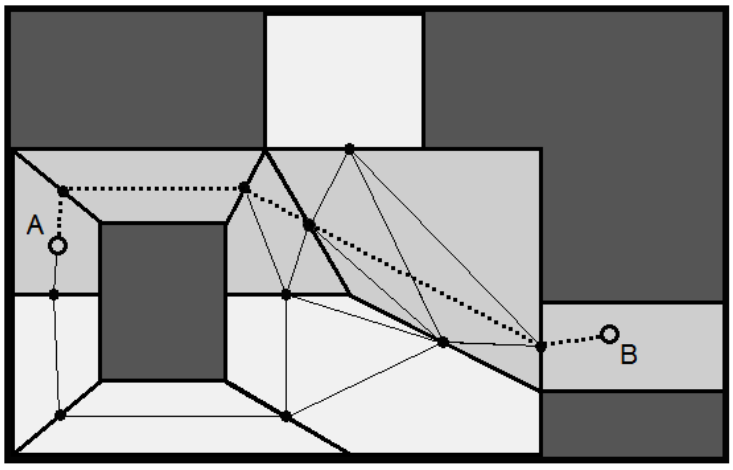

Figure 4.9: An example of a navigation mesh. To facilitate pathfinding, polygons are connected by the midpoints of their edges (there are other approaches, such as connecting polygons by their vertices and centroids). A path between point A and B is represented by the dotted line and the highlighted polygons. Graph edges are represented by the thin lines.

A convex polygon is a polygon such that every line segment between any two of its vertices

remains within its bounds. This property is important as it allows game developers to introduce certain optimisations not attainable in other representations. The fact that the polygons are convex guarantees that the agent can walk in a straight line to any point inside the polygon it is currently in. It significantly improves the performance of a navigation system, as the number of searches is reduced. Moreover, it is simple to determine whether a certain point can be accessed, as it is sufficient to check whether it lies within any of the polygons, as it can be seen in the figure below.

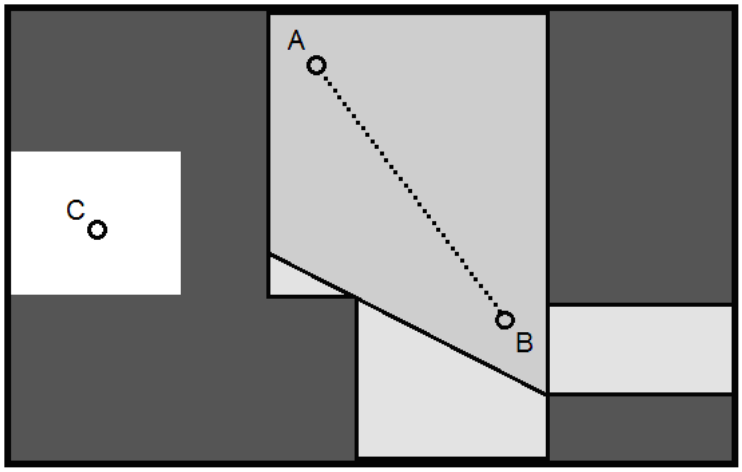

Figure 4.10: As points A and B are contained within the same polygon there is a direct connection between them. Point C is not contained by any polygon and therefore it is inaccessible.

There are two main types of navigation meshes, triangle based and N-sided polygon based. Triangle based meshes have been successfully used in popular games such as *Resident Evil* [22]. They can be created using the Delaunay [23] triangulation, which generates a number of triangles by connecting corners of objects in the game world. However, as such meshes consist only of triangles, they contain more nodes than meshes constructed from higher order polygons, as shown below (the triangle based mesh from figure 4.11 has fifteen, nodes while the mesh shown in Figure 4.9, representing the same game world, has only nine).

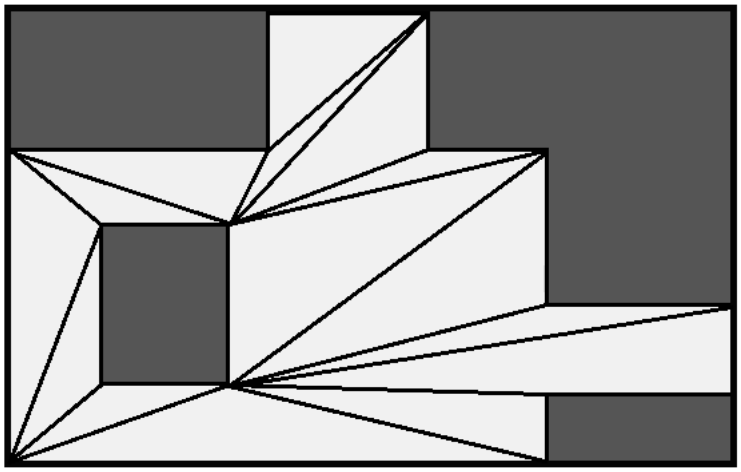

Figure 4.11: Game world decomposed into triangle based mesh.

N-sided polygon based meshes also have been widely used in a number of recently released games. This is because, this type of mesh is implemented by a very popular commercial game engine Unreal Engine 3 [24]. An example of such mesh can be seen in figures 4.9 and 4.10. As this mesh type contains a low number of polygons compared to other approaches, it has been chosen over the triangle based mesh type to represent the search space in this game.

4.3.2 Overview of automatic mesh generation methods

The automatic generation of an optimal mesh is not trivial. It has been shown that this problem is NP-Hard [23], therefore all existing approaches are only sub-optimal. There are three main factors that need to be taken into account: number of generated polygons, coverage provided and computation time.

The Delaunay triangulation provides a full coverage and can be computed in a reasonable time (implemented using an appropriate algorithm it has linearitmic time complexity). However, it generates a large number of polygons (figure 4.11 shows a mesh comparable to the results obtained by the Delaunay triangulation). This flaw can be mitigated using the Hertel-Mehlhorn algorithm [23]. It works by removing an edge joining two polygons that, when combined together, shape another convex polygon. This process is repeated until there are no edges that can be removed. It results in a significantly decreased number of polygons, and hence higher quality of decomposition, as shown in Figure 4.12.

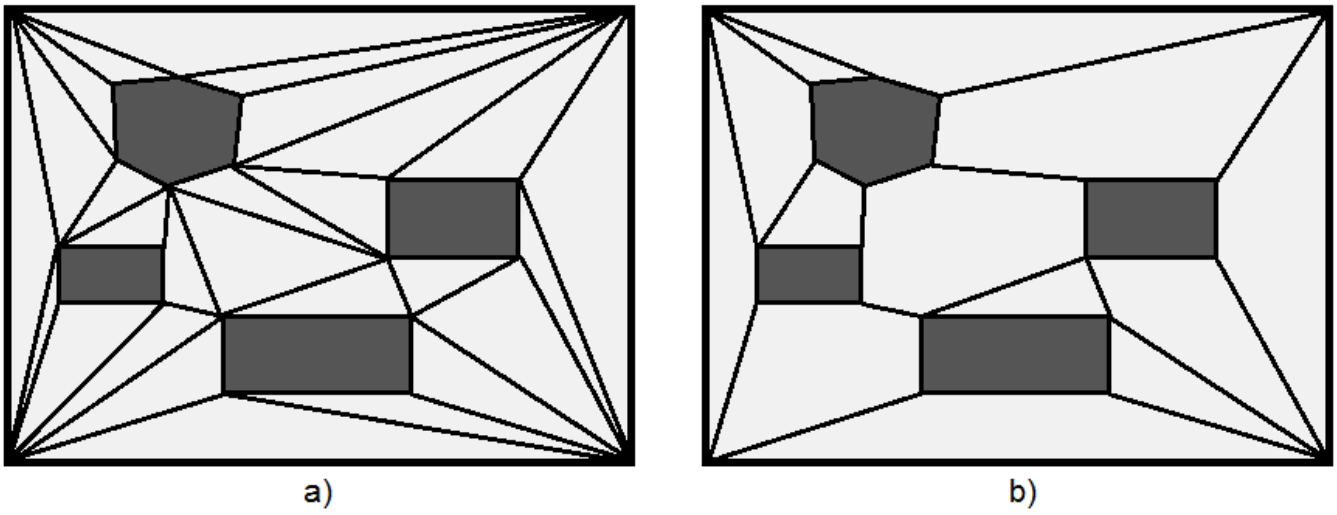

Figure 4.12: Game world represented by a triangle based mesh (a) and the same mesh after removing unnecessary edges (b).

This approach, however, suffers from an inherent flaw of triangle based meshes, in that there are numerous polygons with vertices originating from the same point (in particular from corners as can be seen in the figure 4.12). As a result, characters standing in this point are contained by more than one region, which makes it difficult to determine the correct path. This issue aggravates when the geometry of the objects in the game world is approximating the shape of a circle (as the object in the upper left corner of the figure 4.12). Another way of generating a navigation mesh is the space-filling volumes algorithm proposed by Tozour [25]. This approach is growth based; the algorithm begins by placing a number of squares at constant intervals across the map. The squares are then scaled in all four directions. Once one of the edges of a square collides with an obstacle,

the growth in that direction stops. A result of such a process can be seen in the figure below.

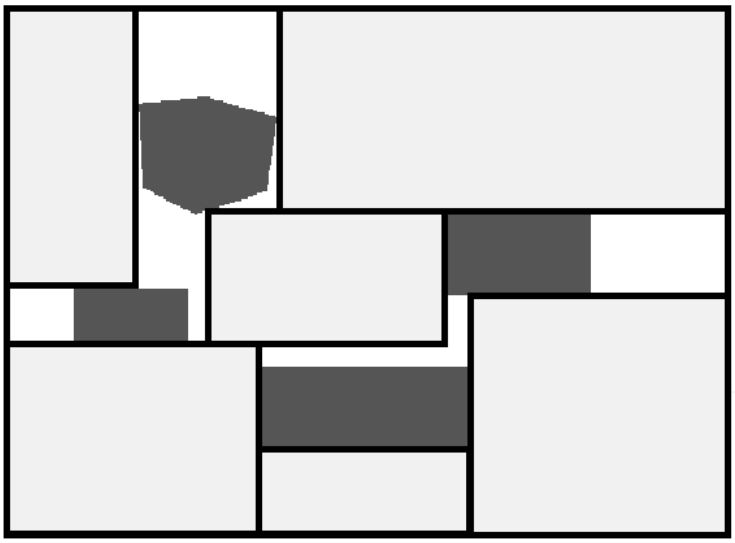

Figure 4.13: Example of the game world decomposed by space-filling volumes algorithm.

This approach results in a very low number of polygons and performs well in the worlds where the objects are placed at regular intervals. However, it does not provide a full coverage of the map. It can be improved by increasing a number of squares initially placed, but it still does not guarantee a complete, coverage especially if the geometry of the objects is not axis-aligned (that is whose edges are not parallel to the coordinate axes of the world space, as the object in the upper left corner of the figure above).

This approach has been improved by recently developed DEACCON (Decomposition of Environments for the Creation of Convex-region Navigation-meshes) algorithm [26]. It starts in the same way by distributing a number of squares over the game world, however, its growth process is different. The algorithm distinguishes three types of possible collisions between growing squares and objects in the game world, presented in Figure 4.14 reproduced from the original paper [26].

The first considered case is the simplest one, where a rectangle as well as the object hit are axis-aligned (base case in the Figure 4.14). The second collision type occurs when one of the object's vertices is inside the growing region, as can be seen in Figure 4.14 (b). In both cases, the growth in the direction of the normal of the edge intersecting with the obstacle is ceased. The third collision type takes place, if one of the region vertices lies within the encountered obstacle. In this case, such a vertex is replaced by two new vertices. The growth process continues, but those two vertices are moved according to the line equation of the intersecting edge, as shown in Figure 4.14 (c).

Once all the regions have been expanded as far as possible, the algorithm performs "seeding". It places new seed regions in all areas adjacent to the previously expanded regions, which has not been covered. Then the new regions are expanded in the same manner as described above. The whole process is repeated until a complete coverage is achieved. In addition, to decrease the node number, the algorithm merges all polygons that when combined, still maintain the convexity property.

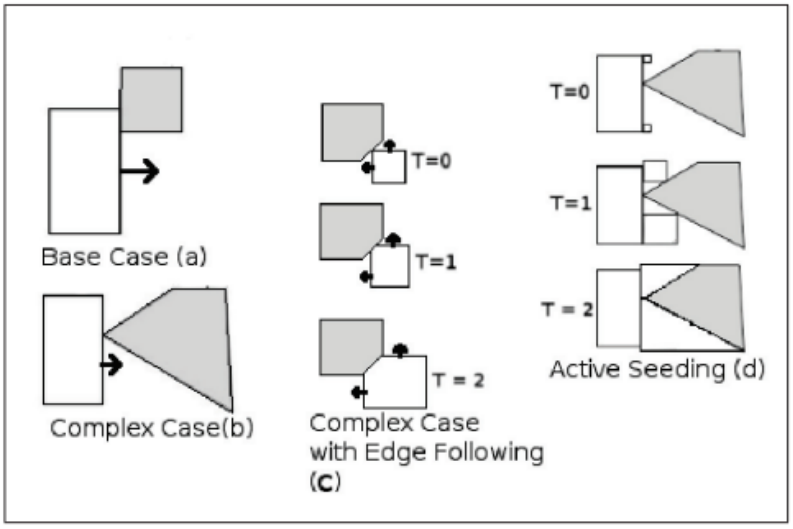

Figure 4.14: Three different cases of collision between growing regions (white polygons) and objects in the game world (grey polygons): base case (a) and two complex cases (b and c). The combination of seeding and region expansion with edge following guarantees a full coverage (d).

The DEACCON algorithm is superior to the space-filling volumes algorithm as it results in a full coverage of the game world, as can be seen in the figure below (adapted from the original paper).

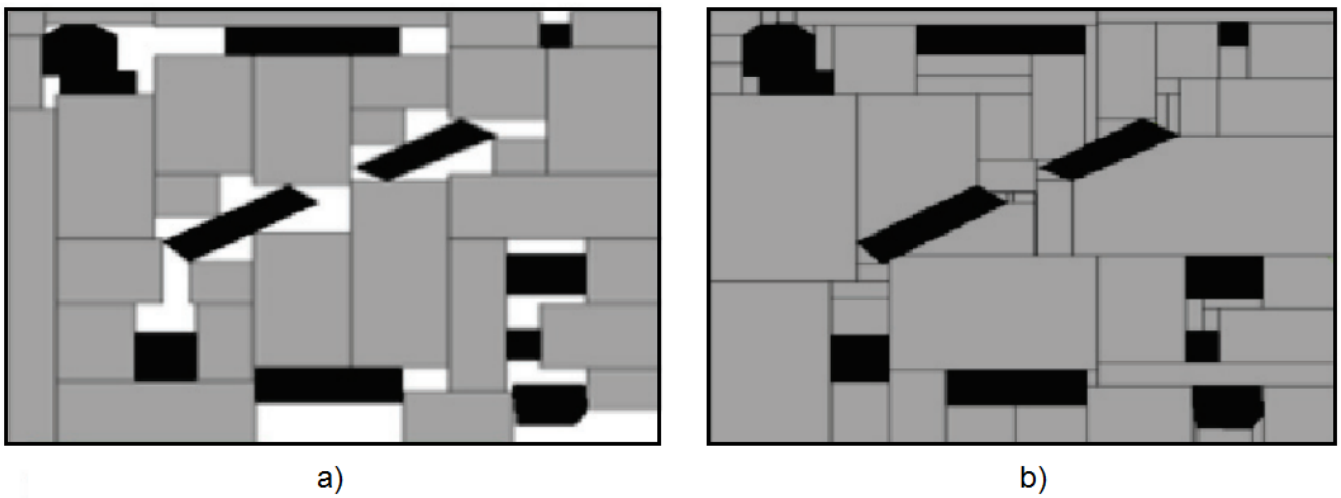

Figure 4.15: Game world of a relatively complex structure, decomposed using space-filling volumes (a) and DEACCON(b) algorithms.

4.3.3 Mesh generation method used

The DEACCON algorithm provides a complete coverage of the game world while maintaining low number of polygons and therefore it was initially intended to be used in this project. However, during its implementation, several issues affecting performance have been identified.

These problems are related to the second complex collision case, which involves the edge following process. It turned out to be computationally expensive to determine which edge of the object hit intersects with the expanded region and then to expand this region further with respect to the line equation of that edge. Despite the fact that these calculations are quite simple (they involve determining and solving 2D linear equations), when carried out repeatedly they produce a large computational overhead especially noticeable on mobile devices. While the intersecting edge is identified only once for each collision, the growth process is performed gradually by expanding the region by small amounts. This issue could be mitigated by generating the mesh once, during the development phase, as the generation time in such a case would not matter. The model of the mesh would be stored in a file and loaded on the game launch. Unfortunately, it would impose certain limitations on features that could be added in the future, such as procedurally generated maps or a game world whose structure could be modified by the player.

Another issue arising from this method is that it increases the number of vertices of the polygons. While this is generally a benefit, as it results in a lower number of polygons, there are certain advantages of rectangle based meshes over meshes constructed from higher order polygons.

In order to construct a graph, the polygons need to be connected with each other in a way that is usually accomplished by joining them by the midpoints of their edges and their centres. While it is easy to determine a centre of a rectangle, it is more computationally expensive to calculate the centroid of a polygon.

Another benefit is that it is easy to determine which mesh node (rectangle) is occupied by the player or an agent, what is required to find a path between them. If the rectangles are axis-aligned, to determine whether a rectangle contains a given point, it is sufficient to check whether the point coordinates are smaller than the top right and bigger than bottom left corner of the rectangle. Testing whether a point lies within a polygon is more complex and therefore requires special algorithms. Even though in this specific case it can be simplified by taking advantage of the polygon convexity, it is still incomparably more expensive. This issue is especially important, as it will be present regardless of how the mesh will be generated, since these tests need to be performed in real-time during game execution.

For these reasons, this project uses a simplified version of the DEACCON algorithm generating a rectangle based mesh. It begins by placing a number of squares in the game world (seeding phase). Then the squares are gradually expanded as much as possible. At this point, the game world is partially covered by a number of rectangles. To increase the coverage, the locations adjacent to each edge of the rectangles are checked if they contain any gaps. All gaps found are then seeded with new squares that are then grown in the same manner as described above. The whole process is repeated until there are no gaps left to be filled. A pseudocode snippet presenting this method is shown on the next page. An example of such decomposition is shown in Figure 4.16.

Algorithm 1 Modified DEACCON algorithm

function GenerateMesh()
$rectangles$ = null
$seeds$ = null
$noGapsDetected$ = false
// World is seeded with initial rectangles
GenerateSeeds($rectangles$)
// Initial rectangles are expanded
GrowRectangles($rectangles$)
$rectanglesToSeed$ = $rectangles$
repeat
  // Seeds are placed in locations adjacent to the region edges which are // not covered
  $noGapsDetected$ = DetectGaps($rectanglesToSeed$, $seeds$)
  // Adding the newly distributed rectangles to the navigation mesh
  rectangles.Add(*seeds*)
  // Expanding new rectangles
  GrowRectangles($seeds$)
  // Newly placed rectangles need to be seeded
  $rectanglesToSeed$ = $seeds$
until $noGapsDetected$
// Calculating centers and edge midpoints of the rectangles
CalculateGraphNodes()

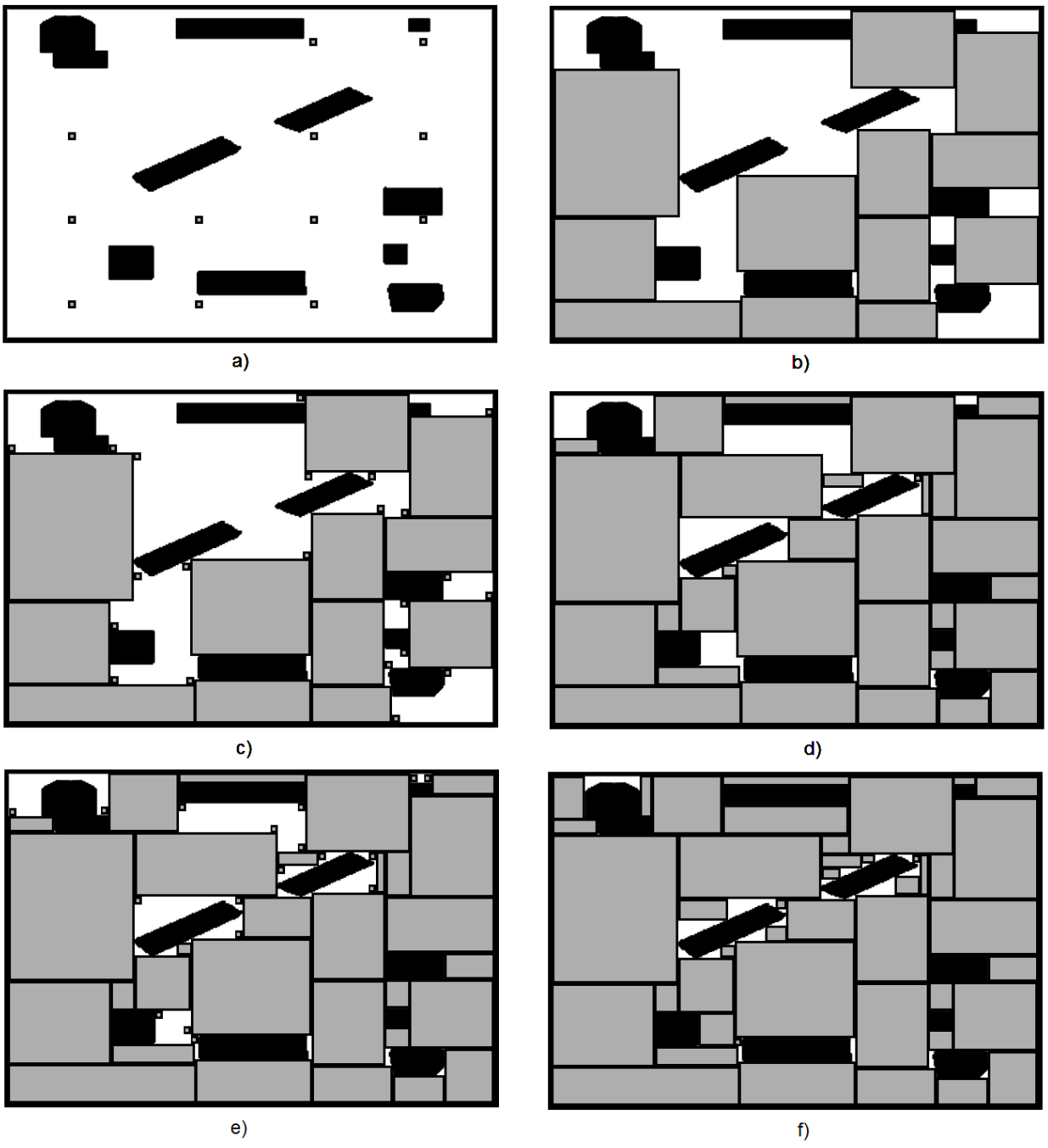

Figure 4.16: Game world decomposed using the simplified DEACCON algorithm. Images present the following stages: initial seeding (a), growth of the initially placed rectangles (b), filling not covered locations with additional squares (c, e), and their growth (d, f).

Despite the fact that this method does not guarantee the game world will be filled completely, the coverage level is quite high. In fact, in situations where the structure of the world is not excessively complex, the algorithm can be terminated after approximately three iterations, as the remaining gaps are small and their presence does not affect the functioning of the pathfinding system.

4.3.4 Dynamic extension for mesh generation

Thanks to the optimisations made, the method described above generates mesh in a short time, as shown in Appendix B. However, the generation time is highly dependent on the game world size. While the method performs well when used on the current map, if the map was substantially larger, it would take an unacceptably long time for the mesh to be generated. Another problem with this approach is that once the mesh is created, it cannot be changed. This constraint would be an issue if the user was allowed to modify the game world structure (for example, destruction of a wall could create a new passage that would require adding new polygons to the mesh).

Authors of the DEACCON algorithm have also identified this issue and therefore they developed a dynamic extension called the Dynamic Adaptive Space Filling Volumes (DASFV) [27]. It is a method of updating a mesh when objects are either added to or removed from the game world.

When a new object is added, all polygons occupied by the object are found and removed from the mesh. Then the polygons previously connected to these removed regions are identified. Their connectivity information is reset and in order to fill the empty space, new polygons are placed in free locations adjacent to their sides. Newly placed polygons are then expanded in the same way as during the mesh generation, as can be seen in Figure 4.17, reproduced from the original paper. The pseudo code snippet below presents how this method has been implemented in this project.

Algorithm 2 Handling addition of new object using DASFV algorithm

```
function AddObject(object, rectangles)
rectanglesOccupied = null
rectanglesToSeed = null
// Identifing rectangles occupied by the added object
for each rectangle in rectangles do
    if rectangle.IntersectsWith(object) then
        // Storing adjacent rectangles to seed them later
        rectanglesToSeed.Add(rectangle.AdjacentRectangles)
        // Disconnecting adjacent rectangles from the found rectangle
        rectangle.ResetConnections()
        rectanglesOccupied.Add(rectangle)
    end if
end for
// Removing occupied rectangles from the navigation mesh
RemoveRectangles(rectanglesOccupied)
noGapsDetected = false
seeds = null
repeat
    // Seeds are placed in locations adjacent to the region edges which are // not covered
    noGapsDetected = DetectGaps(rectanglesToSeed, seeds)
    //Expanding the new rectangles
    GrowRectangles(seeds)
    //Newly placed rectangles need to be seeded
    rectanglesToSeed = seeds
until noGapsDetected
// Calculating centers and edge midpoints of the rectangles
CalculateGraphNodes()
```

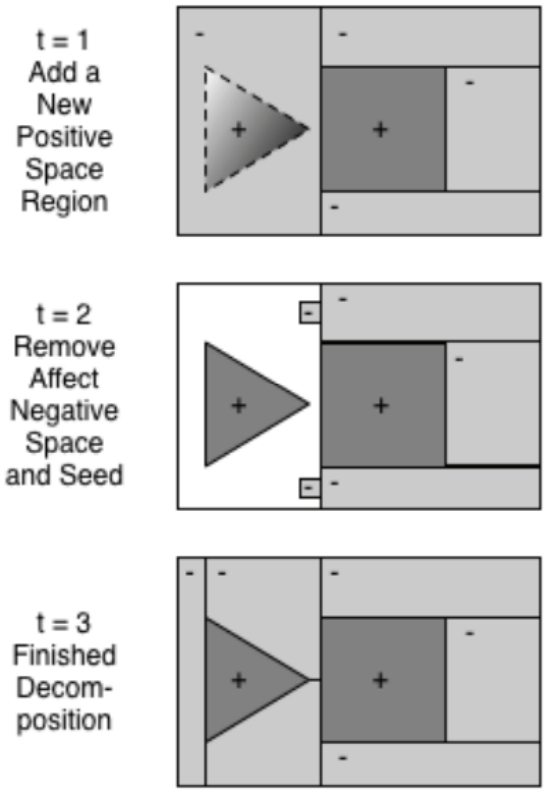

Figure 4.17: Three stages of addition of a new object (marked with dotted outline). Authors of the paper refer to the objects in the world as 'positive space regions' therefore, in the figure, they are marked with the plus signs, as opposed to regions constructing the mesh, denoted by the minus signs. Firstly, the polygon containing the object is identified. Then it is removed and new polygons are placed in the neighbourhood of the polygons previously connected to it. Once they are expanded and other remaining gaps are filled, the game world is completely covered again.

This method allows for updating the existing mesh at runtime, but as it still needs to be generated beforehand, the performance problem mentioned above still persists. However, a few modifications need to be made to solve it. Shortly after publishing the DASFV algorithm, its creators presented a method of building a mesh dynamically during game play called Navigation-Mesh Automated Discovery (NMAD) [28], which has been implemented in this game. This algorithm allows for creating a mesh with absolutely no previous knowledge about the objects present in the game world. This method is partially inspired by the process of map learning used in robotics. It starts by building an inaccurate model of a mesh which is later incrementally refined by an agent's discoveries.

To begin with, the algorithm makes an assumption that there are no objects in the game world, and thus creates a navigation mesh consisting of one polygon covering the entire map. When the agent traversing the world encounters an obstacle, it is added to the list of discovered objects and the navigation mesh is updated, as shown in Figure 4.18. As the agent moves through the world and comes across other objects, the accuracy of the world model represented by the navigation mesh is increasing.

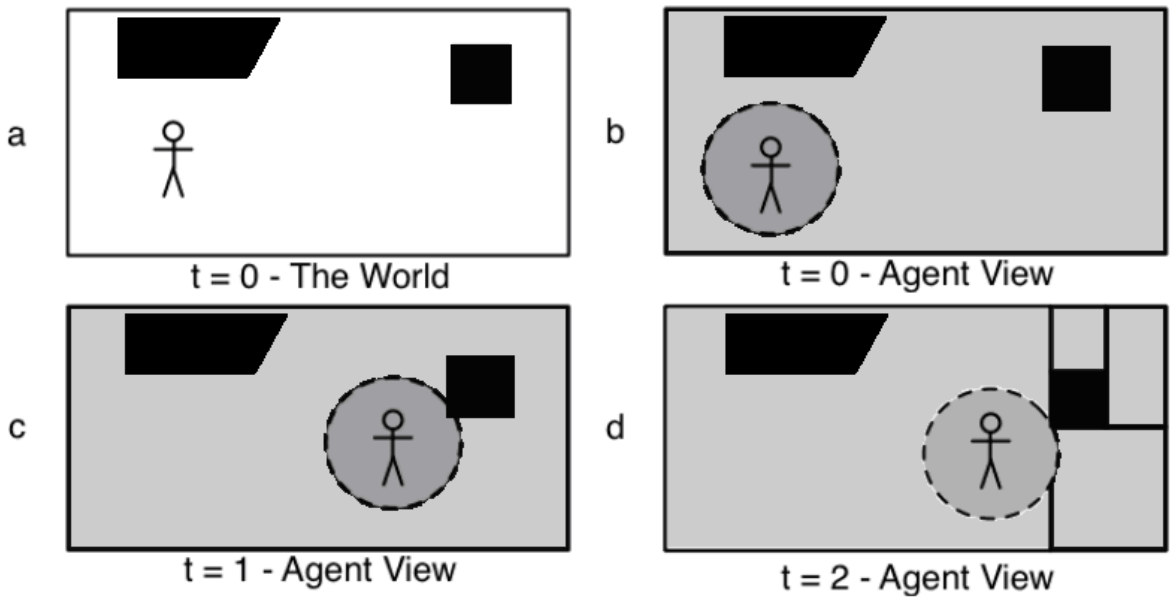

Figure 4.18: Figure adapted from the original paper. The first image (a) shows the actual world structure consisting of one agent and two obstacles. Since the NMAD algorithm initially assumes that there are no objects in the game world, the entire map is covered by a single polygon (b). The dotted circle represents the agent's detection area. The next image (c) shows that the agent has discovered one of the present objects. The last image (d) presents the mesh updated with information about the encountered object. The other obstacle (in the upper left corner) has not been detected by the agent and therefore is not taken into account by the method generating the navigation mesh.

4.3.5 Graph construction

In order to take advantage of the navigation mesh, a graph connecting the polygons needs to be constructed. As in most games, this project uses an undirected graph, which is a graph whose edges have no orientation, for instance an edge connecting node A to node B is identical to the edge connecting B to A.

There are several possible methods of connecting nodes. The most popular include connecting the polygon's edges, edges' midpoints and centroids of polygons, as shown in the figure below reproduced from [29].

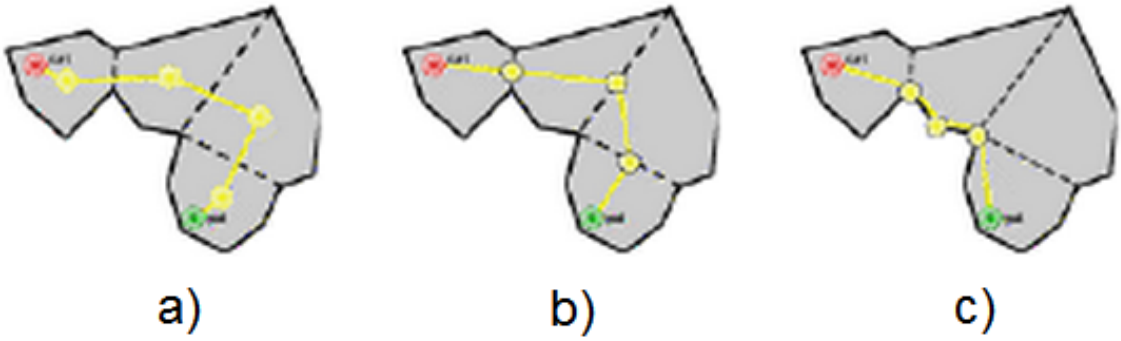

Figure 4.19: Three different ways of connecting polygons. The graph nodes are placed on: polygon centroids (a), edge midpoints (b) and obstacle corners (c).

None of the methods gives optimal results, even when used in environments of relatively simple structure. The first two methods, in some cases, result in path overlapping objects in the game world, as shown in Figure 4.20. When using the third method, the agents tend to walk too closely to the obstacle walls. Therefore, in this project, a combination of connecting polygon centroids (in this case rectangle centres) and edge midpoints has been used.

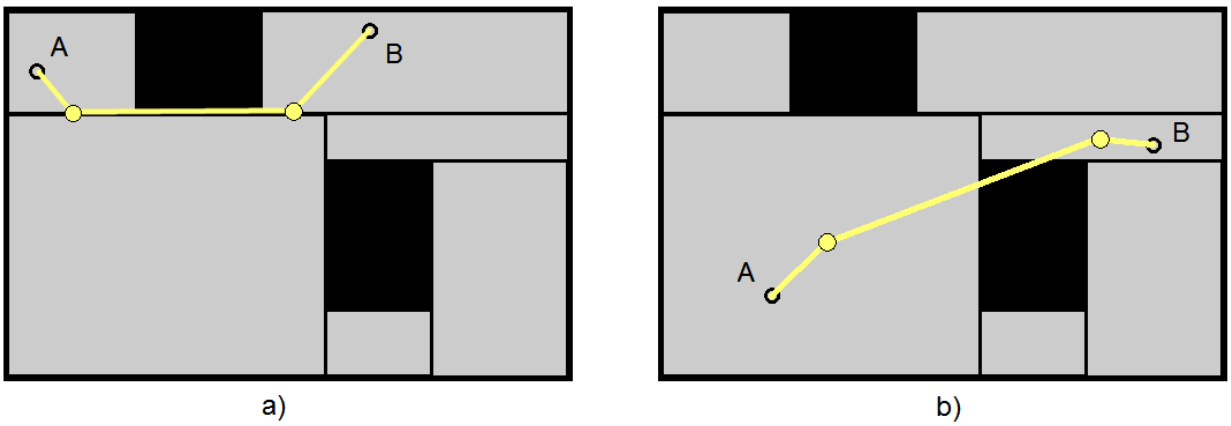

Figure 4.20: Rectangles connected by their edge midpoints (a) and centres (b). Both methods have generated paths overlapping the obstacles in the game world.

4.3.6 Search algorithm

4.3.6.1 Description

The navigation system in this project uses the A* [5] algorithm to find the shortest path between two points. It is the most widely pathfinding algorithm used in modern computer games. If a game uses a different algorithm, it is likely to be a modified variation of the A* (such as D* Lite [30]).

To find the shortest route, the A* algorithm uses a best-first search. It explores the graph by choosing nodes that appear to be best, according to some specified criteria. To achieve this the A* uses a cost function estimating a distance to a given node. This function is a sum of two costs: the known distance from the start node to the current node and the estimated cost of getting from the current node to a goal, commonly referred to as heuristics.

During the graph traversal, the A* maintains two sets of nodes: an open set and a closed set. The open set stores the nodes that have not been expanded yet. At each stage of the algorithm, the node with the lowest cost is removed from the open set, and nodes in its neighbourhood are added to the set with their cost values. The explored node is then added to the closed set, to mark it as already traversed. Once the goal node is found, the algorithm returns all nodes that led from the start node to the goal. An example of a path calculated using the A* algorithm is shown in Figure 4.21.

To ensure that the A* always finds the shortest path, the heuristic used must be admissible: it cannot overestimate the distance to the goal. In this game, the heuristic is calculated using the Euclidean distance, the straight-line distance between two points.

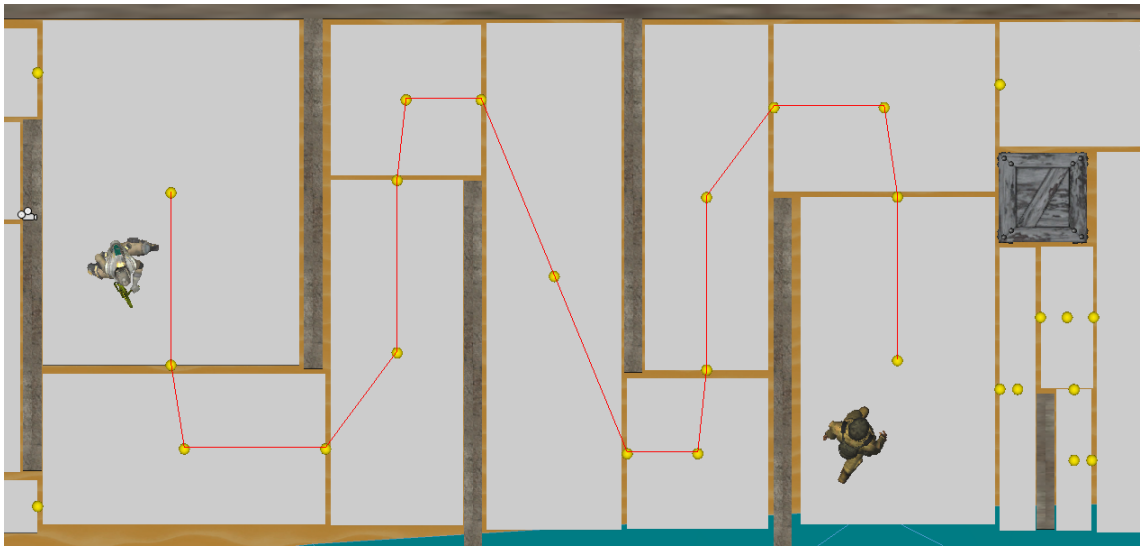

Figure 4.21: An image taken during the game execution showing a portion of the game world, additionally displaying the navigation mesh (grey rectangles) and the found path from the enemy to the player (red line) using the A* algorithm. The yellow circles represent the graph nodes.

4.3.6.2 Considered alternative

There are numerous more advanced search algorithms, however, nearly all of them are more complex versions of the A* specifically designed to handle some special circumstances. For instance, there is a number of incremental search algorithms (such as D* Lite), which can optimise search in situations where the world is not completely known or its structure changes dynamically. When a moving agent encounters a new object, such algorithm uses information about previously performed searches to replan a new path quicker than using the A*, as shown in the Figure 4.22 adapted from [30]. For this reason, they have been widely used in robotics to guide robots in unexplored areas. Such situations are similar to the process of generating the navigation mesh using the NMAD algorithm, and therefore such an algorithm could prove to be useful in this game.

However, as the navigation mesh ensures that the number of graph nodes is very low, no performance issues have been identified while using the regular A* algorithm and thus the implementation of an incremental search algorithm was not necessary.

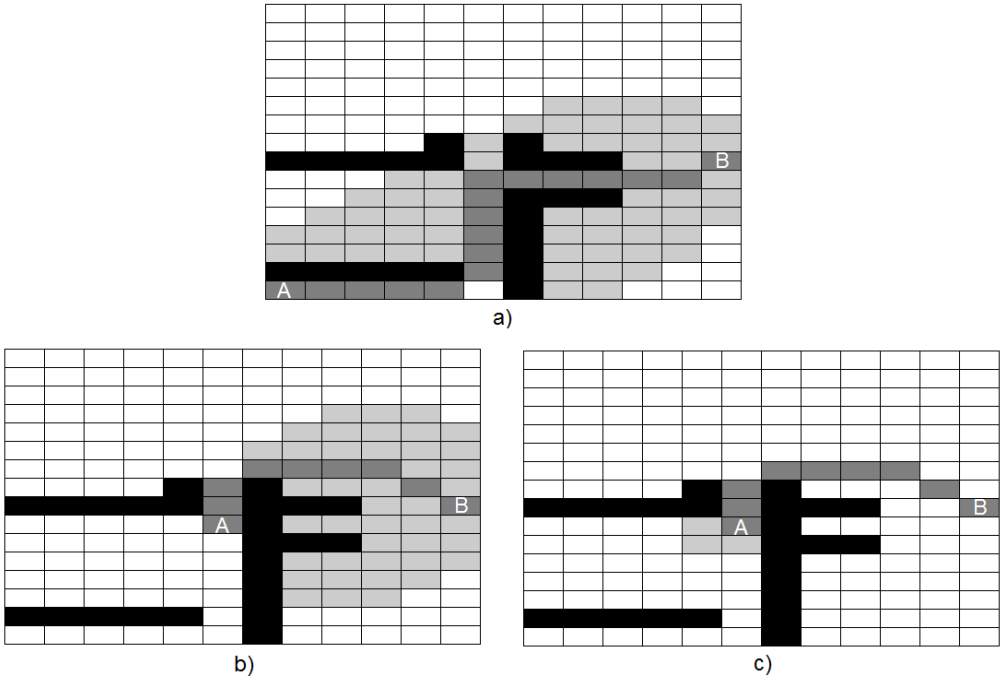

Figure 4.22: Path planning in an unknown environment. a) shows a path found from the point A to B (dark grey cells) and notes which were expanded during the search (light grey cells). Such a result is the same for both A* and D* Lite. The agent following the path has encountered a new obstacle obstructing the route and therefore it needs to be replanned. Images b) and c) show the replanning done by A* and D* Lite, where the D*, taking advantage of the previously planned path, has expanded a substantially lower number of nodes resulting in a quicker search.

4.3.7 Path following

4.3.7.1 Introduction

Using the methods described above, a shortest path between two points can be found. However, such a path is only a general outline of how the final path should look like. As it can be seen in Figure 4.21, if a character would simply follow each node consecutively, its movement would look odd. The found route is the shortest route between two nodes in the underlying graph, not the shortest route possible.

A more optimal route can be generated using a string-pulling or funneling [29] algorithm which eliminates the unnecessary nodes and moves them with regard to the obstacle corners, as shown in the figure below.

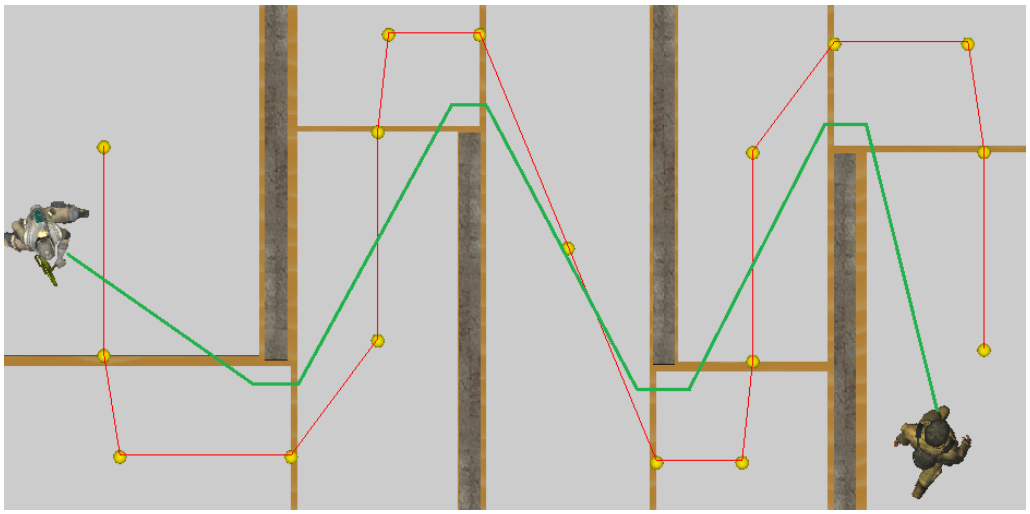

Figure 4.23: An example of a shorter path (green lines) than the one found by connecting the graph nodes (red lines).

Yet, the path shortening alone, does not result in a realistic movement of agents. The characters will be able to reach the goal more quickly, but in a very unnatural way. When a character reaches a node, it suddenly turns to the direction of the next node. It is usually mitigated using some kind of path smoothing technique. However, using such techniques is inefficient when the path needs to be frequently recalculated. This is the case for this game, as the player moving through the game world in a short period of time can visit a number of mesh rectangles.

4.3.7.2 Reactive path following

The developers of a popular game, *Left 4 Dead* (Valve), have introduced a concept of reactive path following [7]. Using this method, instead of following each node from the first to the last one, it looks for the nodes farther down the path and chooses more distant ones. As this method suits this game best, it has been chosen over the approaches mentioned above. To implement it, the nodes constructing the path are checked in reverse order (from the goal to the first node), to determine if they can be accessed. It is done using sphere casting, which casts a sphere of a specified radius in a given direction and informs whether any object has been hit. It is a modified version of ray casting, which works in the same way but casts a ray. Sphere casting is more useful in this case, as certain ways can be passable only by the characters of a smaller size and the sphere radius can be used to indicate the size of the agent. An example of a final path followed by an enemy can be seen in the figure below.

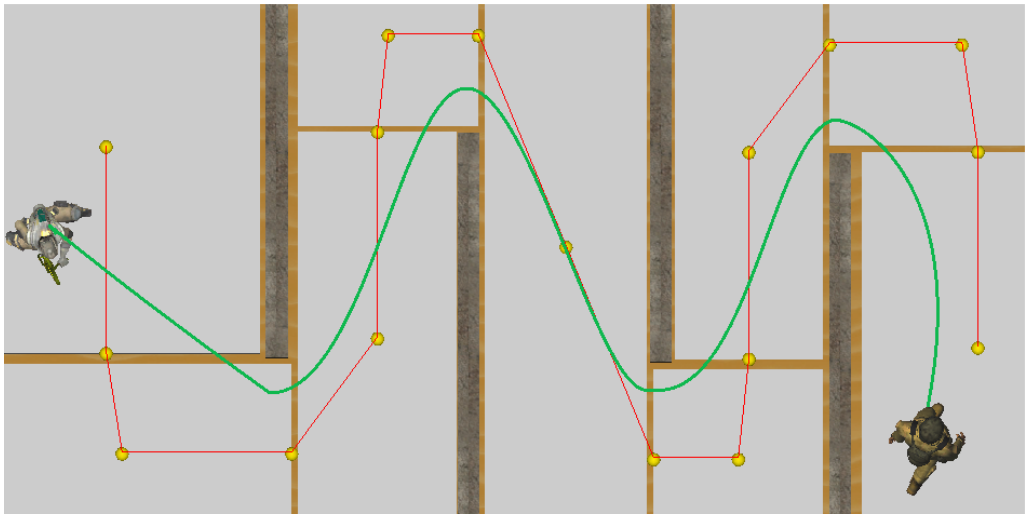

Figure 4.24: An example of a path followed by the agent using the reactive path following (the green line). The smooth turns are not only a result of using this path following technique, but also of a steering performed by the movement motor component that smooths the agent's rotation and handles local obstacle avoidance described in the following section.

4.3.8 Handling inaccessible locations

The agents in the game are generally led to locations that are accessible (such as the location of the player). However, there are cases where the accessibility of the goal cannot be guaranteed. For instance, when an enemy is too close to the player, it moves back. Since neither the state machine controlling the character's behaviour nor the "MoveBack" state giving specific commands has a knowledge about the nearby environment, the character can be led to a location that cannot be reached, as it can be seen in Figure 4.25. Such situation is noticed by the navigation system, as there is no rectangle in the navigation mesh containing the goal. In such case, movement of the character is stopped.

Figure 4.25: An enemy (purple character) tries to move back, as it is too close to the player character. However, the target location is inaccessible (the end of the white line) and therefore its movement is stopped.

4.3.9 Local obstacle avoidance

Pathfidning is a planning process. It plans a route between two points, which due to the simplified underlying world representation, is only an approximation of how a real route should look like. It may result in the problems discussed in the graph generation subsection, involving the generated path overlapping objects in the game world.

There is also an issue related to the objects that are dynamic. Such objects (e.g. a large barrel that can be moved) are usually not taken into account by pathfinding and therefore the navigation mesh considers them to be a free space. Since this project happens to support a real-time mesh generation (using the NMAD algorithm), such objects theoretically could be classified by the mesh as obstacles. In practice, however, it would not be feasible, as the dynamically moved objects would require instant recalculations of the mesh structure. The NMAD algorithm can achieve this in a short amount of time (usually less than one second as shown in Appendix B) which is acceptable as it is done in a separate thread and therefore is unnoticeable by the end user. Yet, it is not fast enough to be performed in a frequency of nearly every game cycle that would be required when the object would be moved. In addition, some dynamic objects explicitly cannot influence the structure of the navigation mesh, as they use it themselves (the agents). One could imagine a navigation system classifying agents as obstacles in its world representation, but not only would it suffer from the problem mentioned above, but also its inefficiency would increase proportionally to the number of added agents, as each moving agent would be constantly modifying the navigation mesh. For this reason, a method handling the avoidance of dynamic object needs to be separated from the pathfinding system.

Thus, to mitigate these problems rather than performing complex planning, a more reactive approach is needed. It has been achieved by a simple alteration of the agent movement direction, taking place when an obstacle in front of this direction is detected. To navigate around the obstacle, a new direction is determined to be in between the original direction and the direction perpendicular (normal) to the wall of the obstacle, as shown in the Figure 4.26. To separate this method from the pathfinding system, it has been put in the component that is at the lowest level of the multi-tier AI hierarchy: the motor movement controller. It is a simple approach; as it is purely reactive, it is possible that the agent, while avoiding one obstacle, will walk straight into another one. However, in an environment that is not excessively complex, this method performs very well.

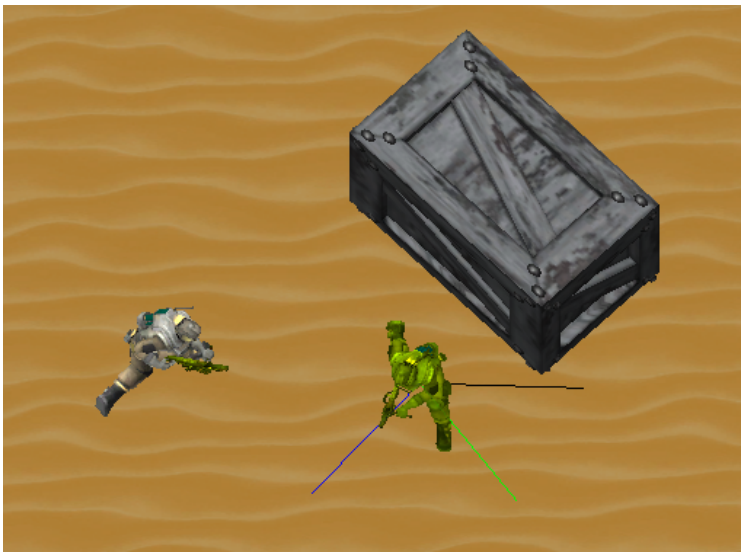

Figure 4.26: The enemy (green character) is forced by the player to move back (in the direction represented by the black line), but it encounters an obstacle. To avoid it, the normal of the obstacle wall is identified (blue line) and a new direction is calculated (green line).

4.3.10 Design

In short, the navigation system consists of the following components:

- Navigation mesh: A component creating a representation of the game world in the form of a navigation mesh. The world model can be either complete from the beginning, generated at the game launch or it is initially incomplete and incrementally refined during the game play.
- MeshNode: A class stornig information about a mesh rectangle such as its size and location. It also contains a number of methods performing intersection tests. For this reason, it uses the Utils class, which contains various static utility methods used throughout the whole project.
- A* Helper: A class using the A* algorithm to calculate the shortest path between two points.
- Pathfinder: A component managing the functionality related to pathfinding. In order to find a path, it maps the specified start point and the goal from the world coordinates to their graph equivalents (nodes). Then it feeds the graph nodes of the navigation mesh together with the start and goal nodes into the A* Helper component. It also performs initial checks allowing an earlier termination of the pathfinding process: checking whether the starting point and the goal are contained in the same rectangle (they are within a line of sight) or if there is no rectangle containing the goal (inaccessible location).
- Enemy controller: As the enemy controller specifies an agents behaviour, it contains a Follow function (invoked by the FollowState) that navigates agents to a given point. This function calls the Pathfinder component to find a path between the agents position and the given goal. It also contains a function handling path following that chooses a node to follow. Finally, based on the node chosen, it sets the characters movement direction and faced direction through the motor movement controller.

- Motor movement: Eases the movement and facing directions to make the character behaviour appear more natural. It also handles the local obstacle avoidance.

The whole structure can be seen on the class diagram below. To make it more readable, variables and method signatures have not been included.

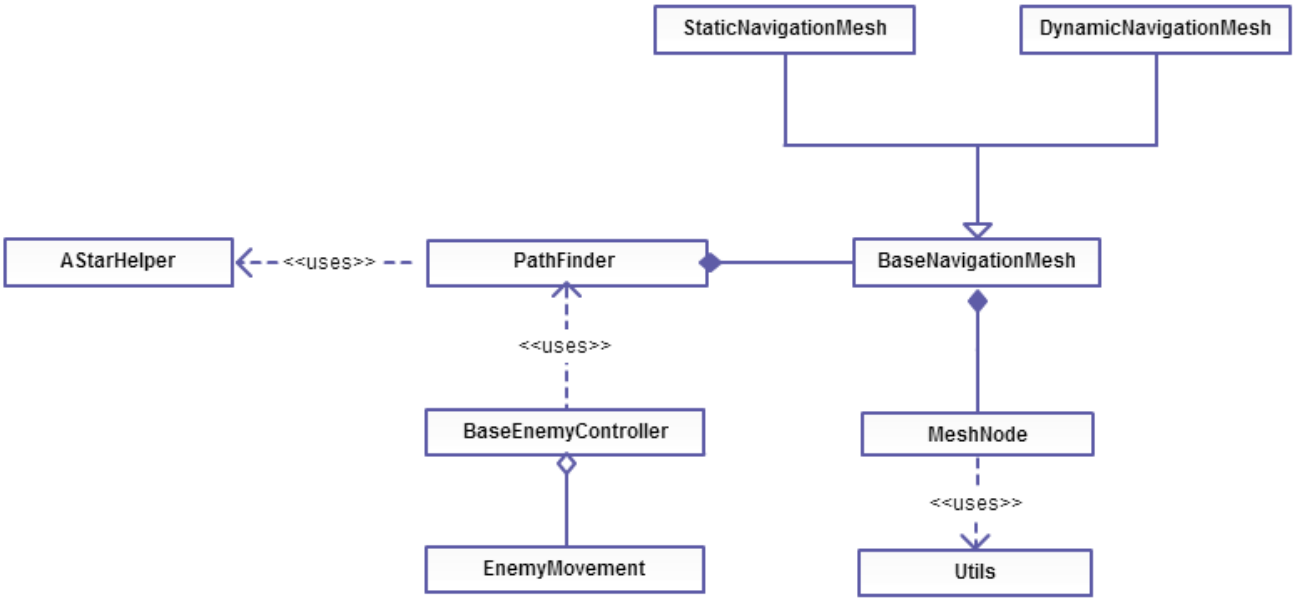

Figure 4.27: A class diagram presenting the structure of the navigation system. The names of variables and function have been omitted to improve the readability.

4.3.11 Implementation

4.3.11.1 General algorithm

The simplified process of pathfinding and path following is described by the following pseudo code snippet. As these tasks are handled by different components, in practice this algorithm is divided into separate classes and methods.

Algorithm 3 Navigation

```
function GoTo(goal)
// Find rectangles containing the start and goal locations
startRectangle = GetRectangle(start)
goalRectangle = GetRectangle(goal)
// The target location is inaccessible
if goalRectangle = null then
    // The movement is stopped
    movementDirection = Vector.zero
    facingDirection = GetPlayerDirection()
    return
end if
// Locations are in the same rectangle therefore the goal can be reached //directly
if startRectangle = goalRectangle then
    facingDirection = GetDirectionTo(goal)
    movementDirection = GetDirectionTo(goal)
    return
end if
// Find path using A* algorithm
path = AStarHelper.CalculatePath(start, goal)
// Reactive path following
for each node in path.Reverse() do
    // If node is in the line of sight
    if CanSee(node) then
        // Go in its direction
        facingDirection = GetDirectionTo(node)
        movementDirection = GetDirectionTo(node)
        return
    end if
end for
```

4.3.11.2 Oriented bounding box intersection

During the growth of the navigation mesh rectangles, two intersection test methods are used to determine whether a rectangle can be expanded further. The rectangle expansion is stopped when it encounters another rectangle (all rectangles are axis-aligned therefore the intersection test is simple, as described in subsection 4.3.3) or an object in the game world. To increase the speed of the mesh generation, the shape of the object is approximated to its bounding box (the smallest possible rectangle that the object can fit into) to simplify the intersection test. However, since the objects in the game world are often rotated, they are represented by oriented bounding boxes that require more complex intersection test than axis-aligned bounding boxes.

To determine whether two convex polygons intersect, the separating axis theorem can be used [31], which states that if two convex shapes do not collide, then there is a line separating them. Such line can be easily found, as when two rectangles do not collide, one of their edges will act as the separating line, as shown in Figure 4.28. Therefore, it is sufficient to check if all corners of one rectangle are on the same side of any edge of the second rectangle and vice versa. To determine on which side of the edge the given point is, a dot product of the edge normal and the vector specified by any point on the edge and the given point needs to be calculated. Depending on which side of

the edge the point is, it returns either a positive or a negative value.

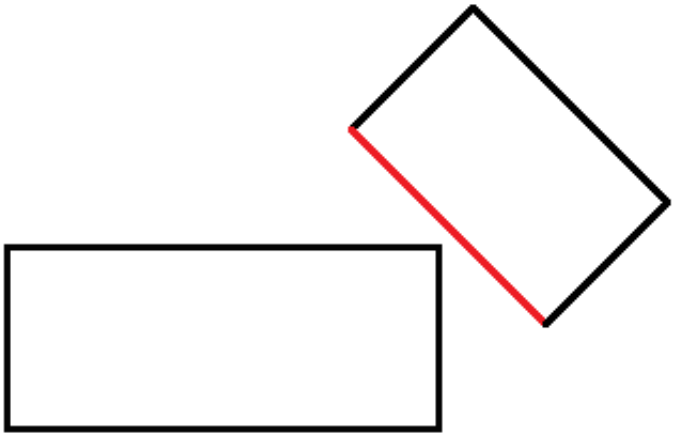

Figure 4.28: The red edge is the line separating the rectangles, as every corner of the other rectangle is on the same side of that edge.

4.3.11.3 Issues encountered

During the implementation of the navigation system, several issues related to the navigation mesh generation and Unity API have been encountered.

As described above, this process requires performing intersection tests between the mesh rectangles and the objects in the game world. In order to do that, four corners of the object's oriented bounding box need to be specified. Even though the Unity API provides access only to the object's axis-aligned bounding box, it can be easily rotated by the angle specified by the object's rotation around the vertical (Y) axis. However, such a solution does not work if the 3D model is not axis-aligned itself when imported by Unity. In such a case, the model needs to be rotated using 3D modeling software.

Another problem has been encountered during the implementation of the dynamic navigation mesh. Thanks to the NMAD algorithm, the dynamic mesh update takes a very short time. However, the end user can easily notice, if the game execution is halted for even a fraction of a second. For this reason, this process needs to be carried out in a separate thread. Unfortunately, the Unity API is not thread-safe, and therefore cannot be used in a such way, what prevents from accessing basic methods and fields, such as an object's position. To bypass this issue, before performing a mesh update, all relevant data (such as positions and bounding boxes) of the objects that have already been discovered is stored in a list that is later accessed by the methods expanding the rectangles and performing the intersection checks. This solution works correctly, however as additional data is stored the performance is slightly decreased and the design of the code is affected.

4.4 Fuzzy logic

4.4.1 Introduction

The transition rules used in state machines to change states, are based on Boolean logic. The conditions that need to be fulfilled for a state to be changed (antecedents) can be either true or

false. Similarly, the output of such a rule (consequent) is fixed to these two values. For instance, a transition rule switching the FollowState to AttackState can be represented by the following code:

IF enemy.DistanceTo(Player) <AttackRange THEN stateMachine.ChangeState(AttackState)

The antecedent is always either true or false. Similarly, the consequent is constrained to these two values, as the transition is either activated or it is not. In the real world, however, humans tend to use more vague terms. As it would be impossible to evaluate the exact distance between two points with the naked eye, the distance would rather be close or far to a certain degree. To evaluate rules in such a way, fuzzy logic can be used. This process starts with fuzzification, which is the mapping of the crisp input (real world values) to fuzzy input. The mapping is performed using a degree membership function defined by a fuzzy set specific for each variable. Then the fuzzy input is combined using fuzzy rules to generate fuzzy output. The generated output can be then turned into a crisp value (defuzzification).

Fuzzy logic has three main advantages over regular if statements used in state machines. As the rules are represented in a more natural form, they can be easily written not only by programmers but also by game designers in the later stages of development. Moreover, as the representation of a certain state is not restricted to two values, it can be described more realistically, what makes the actions performed by agents more natural. Finally, such logic representation can be more easily extended. For example, if new variables would be introduced, such as certain enemy characteristics (S.W.A.T. 2 [32] uses fuzzy logic to model agent personality using such characteristics as aggression, courage and intelligence), it would be much simpler to integrate them in the game using fuzzy rules than large if statements.

4.4.2 Fuzzy sets

Currently, the game distinguishes four variables describing the agent's state, including the amount of ammunition, character tiredness (increases when a heavy weapon is used), the distance to a player and health points. Those factors are evaluated using the collections of mutually exclusive fuzzy sets presented in the figure below.

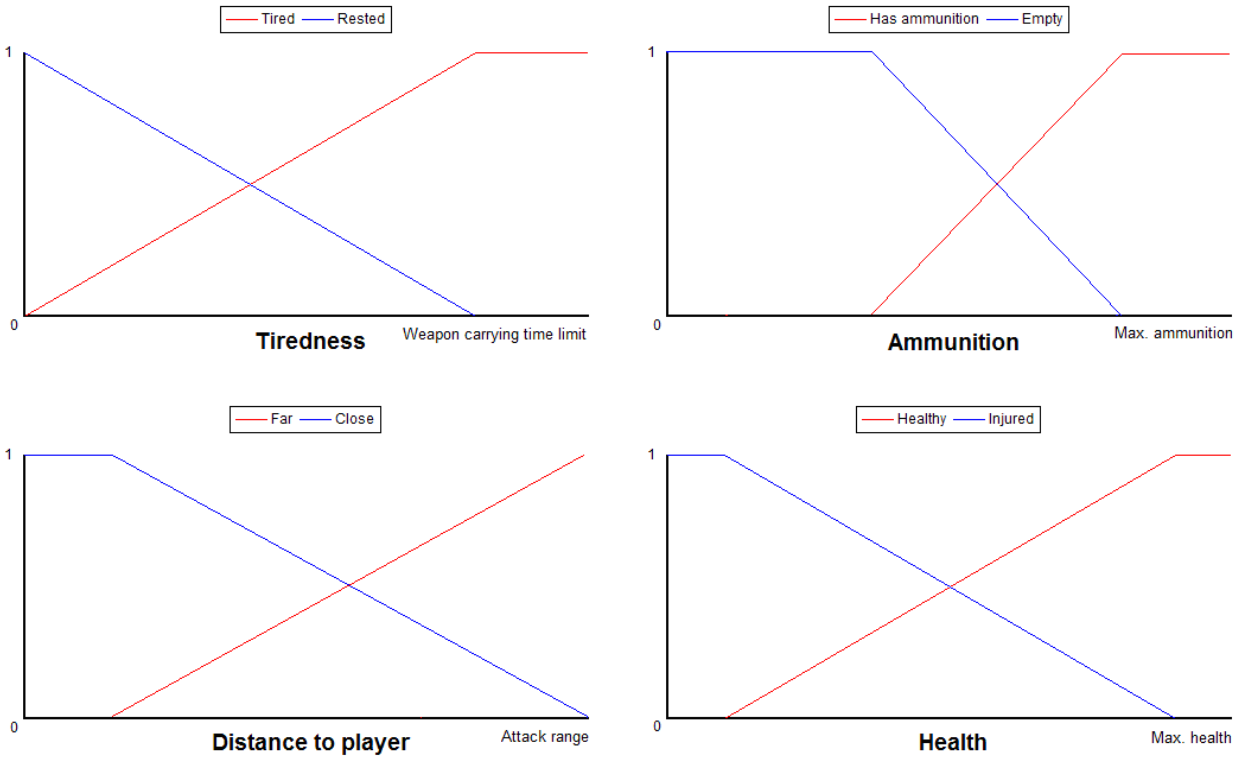

Figure 4.29: Membership functions describing four different agent characteristics.

To facilitate adjusting membership functions, the game uses two fuzzy hedges (operators changing value of degree of membership): ”Very” (square) and ”Fairly” (square root). The effect of applying these operators is shown in the figure below.

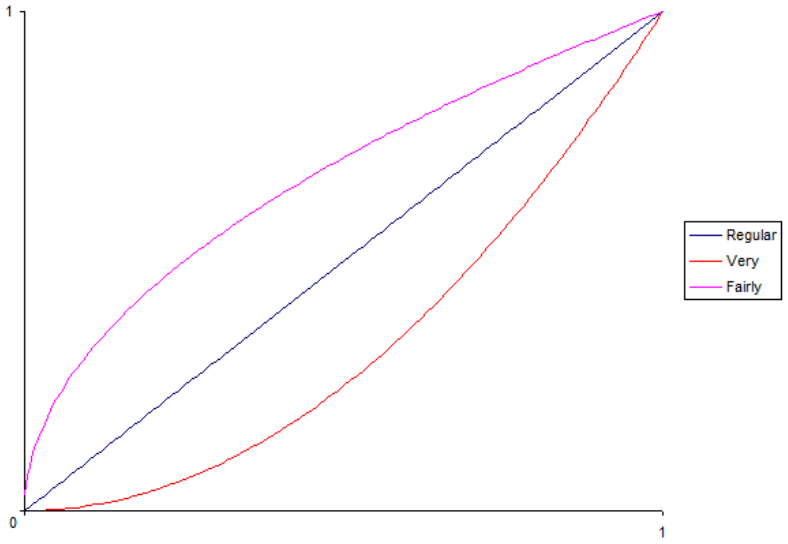

Figure 4.30: Very and Fairly hedges applied on a linear membership function.

4.4.3 Weapon selection

As previously mentioned, one of the enemies uses fuzzy rules to choose one of the two weapons, a pistol or a rocket launcher. A similar method has been proposed in the book by Buckland [21]. As there are three variables, each described by two sets, to cover all possible combinations, twelve rules have been created:

1. IF has ammunition AND Fairly(far) THEN rocket launcher
2. IF empty AND Fairly(far) THEN pistol
3. IF empty AND close THEN pistol
4. IF has ammunition AND close THEN pistol
5. IF rested AND Fairly(far) THEN rocket launcher
6. IF rested AND close THEN rocket launcher
7. IF tired AND Fairly(far) THEN rocket launcher
8. IF tired AND close THEN rocket launcher
9. IF rested AND has ammunition THEN rocket launcher
10. IF rested AND empty THEN rocket launcher
11. IF tired AND has ammunition THEN rocket launcher
12. IF tired AND empty THEN rocket launcher

As facts in fuzzy logic, in contrast to traditional logic, are represented by a number and are not limited to a true or false value, they require a different type of logical operators. For this reason, AND operator is defined by a minimum of two values, while OR operator is defined by a maximum of two values.

In this case, the consequents are in a crisp form, and therefore defuzzification is unnecessary. The rules are evaluated using a simple method: choosing the highest membership value by combining all the rules using an OR operator.

4.4.4 Shooting accuracy

Once a weapon type has been chosen, an enemy's shooting precision is calculated depending on its health and the distance to the player. It is defined by the sets shown in the figure below.

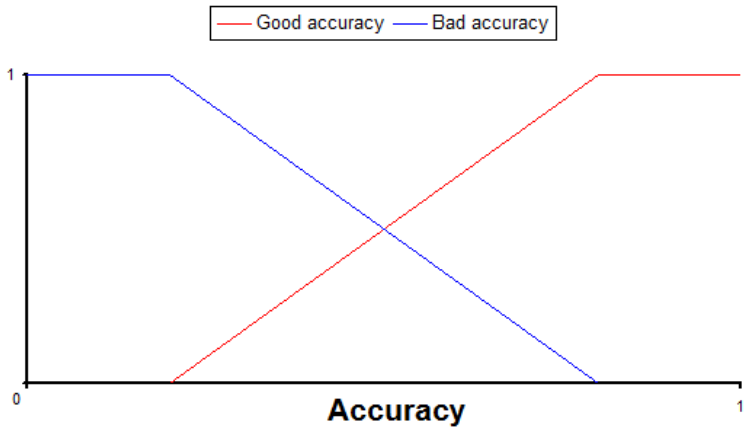

Figure 4.31: Fuzzy sets defining shooting accuracy.

As only two variables are used, the rules can be expressed using the fuzzy associative matrix:

	healthy	unhealthy
close	Good accuracy	Bad accuracy
far	Bad accuracy	Bad accuracy

Table 4.1: A matrix representation of four rules determining shooting accuracy.

To obtain a single value from membership values of the sets, the defuzzification process needs to be carried out. There are many defuzzification techniques, varying in precision and computational complexity. This game uses a method called Average of maxima. It provides a good trade-off between efficiency and precision, thanks to its simplicity and results close to the ones obtained using more advanced techniques [21].

Rules sharing the same consequent (in this case, Bad accuracy) are combined using an OR operator. The Average of maxima method calculates the crisp output using the weighted mean of membership values of each consequent. The weights are specified by the representative values of each consequent which is the middle the set's plateau (where membership equals 1).

$$V = \frac{\sum_{i=1}^{n} d_i r_i}{\sum_{i=1}^{n} d_i}$$

Where:

V: Crisp value

d: Degree of membership

r: Representative value

The calculated output is passed directly to the motor movement component which handles the enemy rotation affecting the shooting accuracy.

4.4.5 Design and implementation

The fuzzy weapon management system is contained by the FuzzyAttackState class. The implementation of fuzzy logic in this project is inspired by the approach presented in the book by

Millington [4] and based on the FuzzyModule class. It contains a method handling rule evaluation using AND and OR operators. It also performs defuzzification. Rules are represented by a FuzzyRule class. The antecedents can be combined either using AND or OR operators, therefore there are two subclasses of the FuzzyRule class: FuzzyORRule and FuzzyANDRule. The fuzzy sets are represented in a similar manner. The sets currently used in the game have their plateau in either the left or the right-hand side. For this reason, they are defined by LeftShoulderSet and RightShoulderSet classes deriving from the FuzzySet class. They implement a CalculateDOM method, calculating the degree of membership of a given value depending on a set type. The separation of rules and sets from the main module enables easy addition of new enemy characteristics.

The diagram below represents the structure of this system. The fact that FuzzyAttackState derives from the BaseState has been omitted to make the diagram more readable.

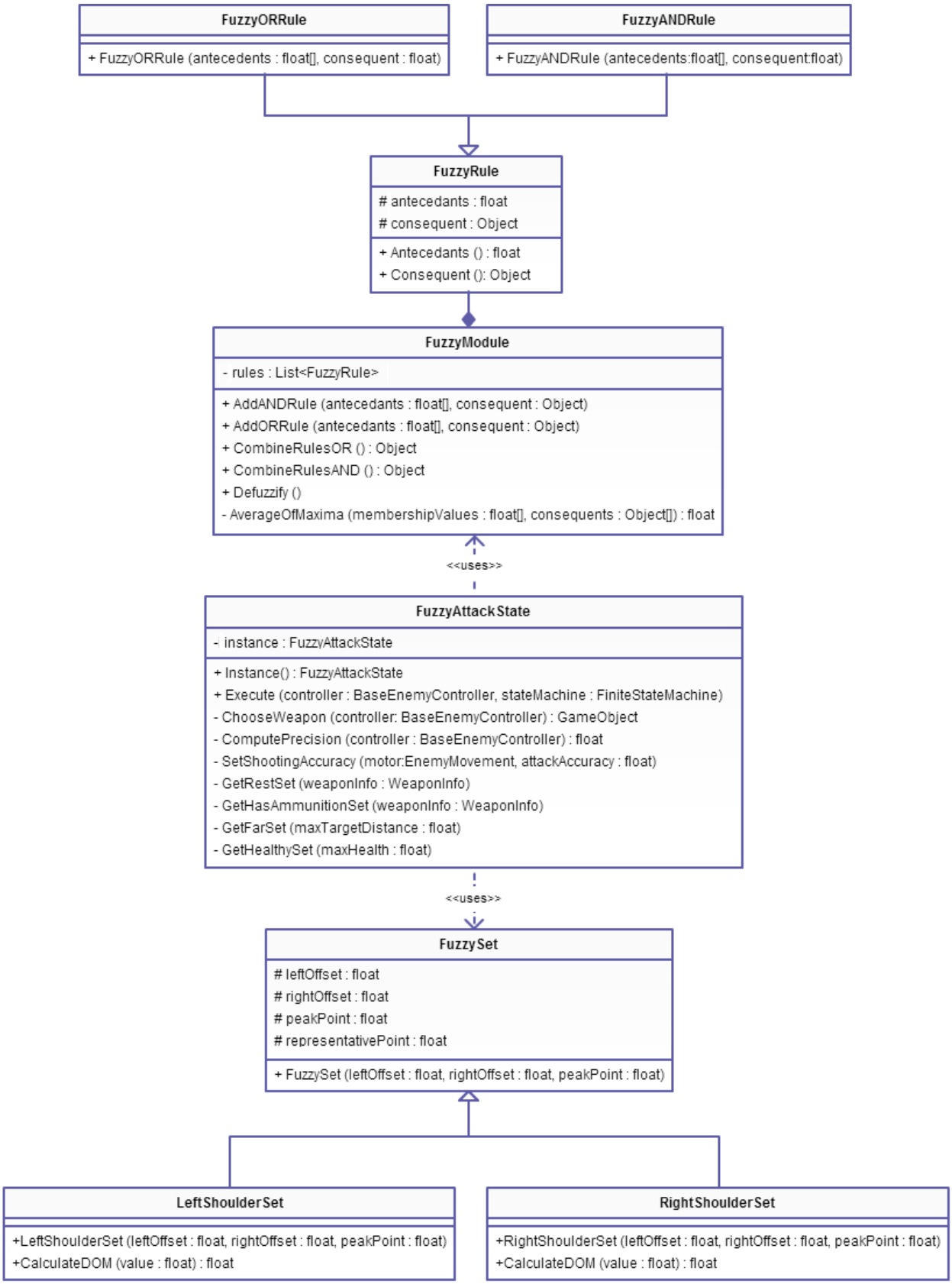

Figure 4.32: A class diagram presenting the structure of the fuzzy logic system implemented in the project.

4.5 Formations

4.5.1 Introduction

Developing different enemy types and modelling their individual behaviours is a fundamental aspect of this project. To make the game even more immersive, agents' intelligence can be developed beyond that by grouping them together into formations. It is possible to simply place several enemies in close proximity; once they detect the player, they will attack together, as a group. However, to make such an attack more realistic, an illusion of teamwork is required, which can be achieved by coordinated movement.

In this game it is accomplished using fixed formations. Each enemy is assigned a specified slot in a formation to follow. The slots are placed to form a fixed, predefined shape. Moreover, they are positioned in relation to a certain target, on which the formation is focused. The formations presented below are easily customisable, as their size and shape can be adjusted and they can include any number of enemies of any type.

4.5.2 Formation types

The game uses two formation types. The first presented formation is a line formation based on two line segments. The slots are placed along those segments, separated by a given distance. The angle between these lines can be adjusted to create a V-shaped formation. This formation is based on a concept of an invisible leader: an entity that cannot be seen or destroyed, guiding members of a formation. Line segments originate from the same point, specified by leader's position, as can be seen in the Figure 4.33.

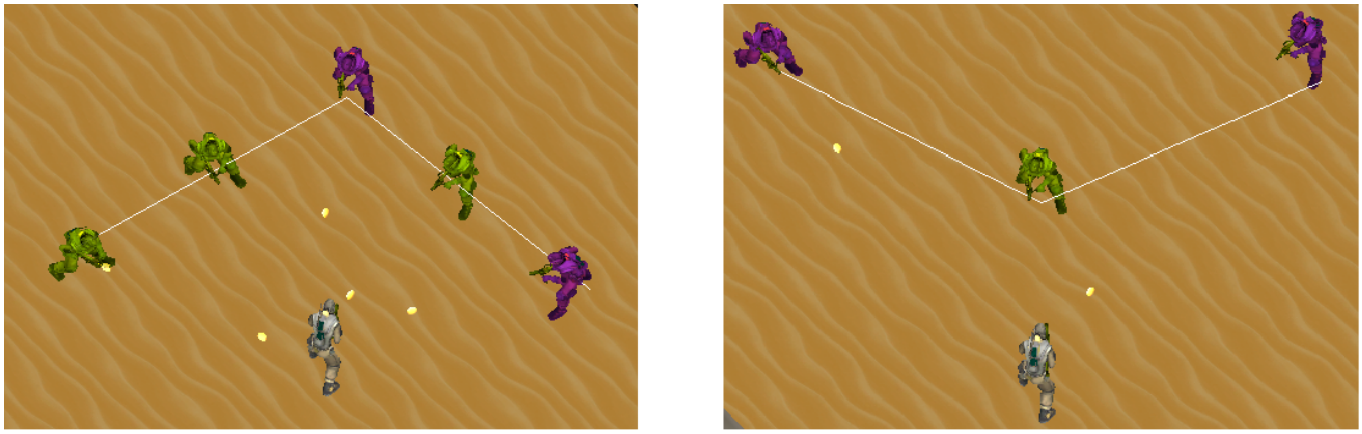

Figure 4.33: Images showing formations created using different angles, consisting of two enemy types (green and purple characters) attacking the player. The white line segments specifying positions of slots are not visible in the game.

In addition, once one of the formation members is destroyed, the formation immediately regroups to fill the missing position. Each enemy is reassigned to a new slot, which is the closest one to its old slot, what makes the regrouping process appear more natural.

The second formation positions enemies around a circle of a specified radius, as can be seen in the figure below. In this case, formation slots are automatically adjusted depending on a number of formation members. Therefore the more enemies compose a formation, the closer they are placed.

It is similar to the line formation, except it does not have a leader; target is simply specified by the player's position.

Figure 4.34: Image showing a circle formation consisting of five enemies of different types.

4.5.3 Safety estimation

As the enemies' positions are strictly confined to the shape of a formation, their movement may seem to be unnatural and static. This issue has been mitigated by introducing a simple tactical analysis method evaluating safety levels of formation slots. Each slot has a value determining safety at its position, initially equal to 1. A slot's safety diminishes, every time the enemy occupying it is hit by a bullet. Depending on a deviation from the initial value, the enemy position is adjusted. If the slot's safety level is higher than the default value, an enemy is moved forward, otherwise it is moved backward. In other words, if an enemy "feels" more exposed to the player's fire it moves back to become less vulnerable.

The safety levels are calculated using the method presented in the book by Millington [4], inspired by the Markov process. In mathematics a Markov process, is generally a process where the prediction for the future can be made based only on its current state. The current state is represented by a state vector containing a set of probabilities of the future events. To calculate the next state, this vector is multiplied by a transition matrix determining how the probabilities will change. For this project, however, it is sufficient to examine a current state of affairs. To create an illusion of enemy cooperation, a change in a safety level of one slot will alter the safety of the other slots. The transition generating the next state decreases the safety of the hit enemy, but increases safety of other enemies. Therefore, if an enemy is hit by a bullet, it moves back but the

other enemies, ”feeling” less exposed as the player is not focused on them, move forward.

Initially, the safety values of each slot stored in a state vector are equal to 1. For instance, a state vector storing the safety values of five enemies would be represented by:

$$V = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Once the enemy in the middle has been shot, a transition vector is calculated to decrease its safety and increase the safety of the other enemies (the further the enemies are from the attacked slot, the safer their slot will become).

$$T = \begin{bmatrix} 2 \\ 1.1 \\ 0.6 \\ 1.1 \\ 2 \end{bmatrix}$$

After multiplication, a new state is generated and enemy positions are updated accordingly as it can be seen in the figure below.

a)

b)

Figure 4.35: Images showing the state of a formation before (a) and after the enemy in the middle has been shot (b).

4.5.4 Design and implementation

Formations are represented by a Formation class. It contains references to the enemies involved in the formation. A reference to the formation component is stored in the enemy controller. This way, the formation is informed if any of its enemies has been hit. When this happens, the controller invokes the formation's OnEnemyHit method, updating the state vector. To represent the actual formation types, two classes deriving from the Formation class have been created. The logic handling enemy positioning, according to a formation shape, and safety levels is contained in UpdateMembersPosition functions. The enemy positions are calculated using basic vector arithmetic (vector addition, multiplication and dot product). The whole structure is presented on the class diagram below.

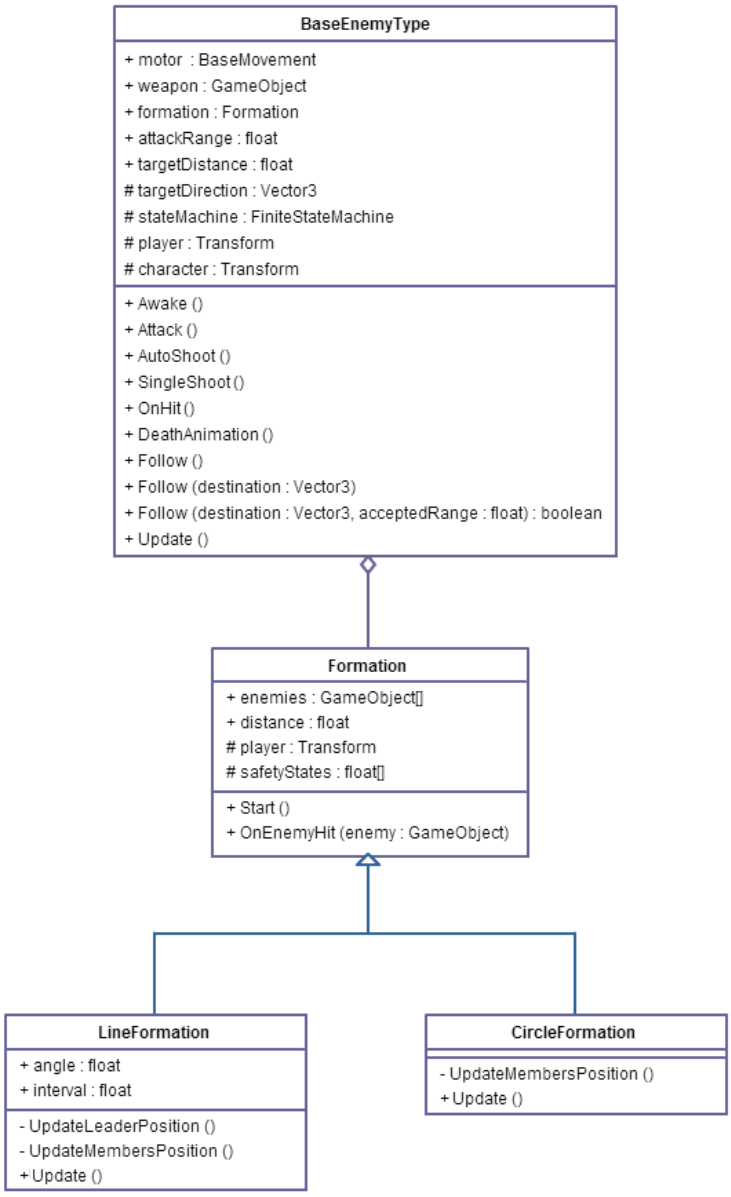

Figure 4.36: A class diagram presenting the structure of the formation system implemented in the project.

As formations' slots are placed strictly according to a predefined pattern (shape), they do not account for obstacle avoidance. However, a formation does not position its enemies directly; it sets their target position, which they follow. As a result, the obstacle avoidance is handled by the navigation system, as can be seen in Figure 4.37.

Figure 4.37: The enemies are not placed strictly according to the shape of the formation (white line segments), but are aligned by the navigation system handling obstacle avoidance.

4.6 Terrain analysis

4.6.1 Introduction

The formations presented above used safety estimation to adjust enemy position. It is a form of tactical analysis, however it is a very basic one, as there is little planning involved. The formation members implicitly change their position once they are attacked by the player. This chapter will describe an agent which uses more complex tactics based on the analysis of spatial data. This analysis evaluates the surrounding environment in terms of two factors: cover provided from the player's attack and distance from dangerous objects or entities to find safe locations.

The concept of tactical analysis in games originate from influence maps presented by Tozour [33], widely used in strategy games. Influence maps indicate locations that are important for the process of decision making, for example the distribution of enemy and friendly forces or positions of dangerous objects (such as explosives). Apart from only showing where those objects are placed, they show what influence those objects have on the nearby environment. For example, if an empty house is surrounded by the enemy forces, locations inside the house are marked as dangerous. However, the strategy games typically use influence maps representing the entire game world, which can be memory consuming. This approach has been modified to be more suitable for shooting games by Sterren et al. in the first-person shooter game *Killzone* [6]. Instead of maintaining one large influence map, it is generated only when an enemy needs to evaluate the nearby environment. It also takes into account the surrounding terrain, not the whole game world. Not only has this method been adapted for this particular game type, but it also has been greatly enhanced by the use of visibility maps indicating which locations provide cover from the enemy

attack. Due to the limited computational power resulting from the high number of enemies, the visibility information is precomputed and stored in a look-up table and is approximated to only eight directions, as it can be seen in Figure 4.38, reproduced from [6]. As it would not be possible to store visibility information between every pair of points in the game world, the terrain analysis method used in *Killzone* relies on its underlying world representation: a waypoint graph.

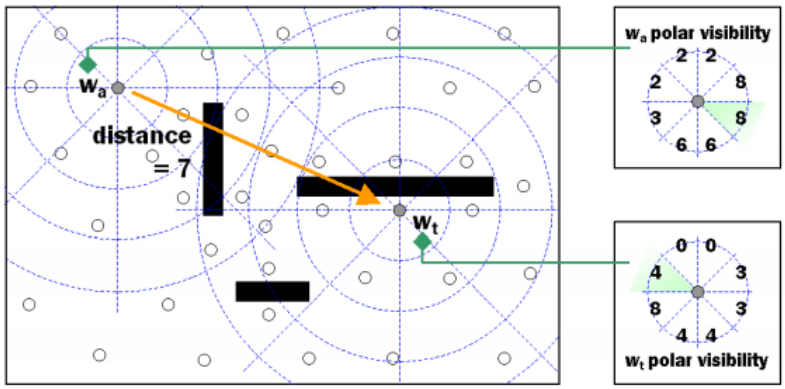

Figure 4.38: The waypoint based visibility map used in *Killzone*. The visibility is approximated to eight directions using polar representation.

The game world presented in this project does not involve a high number of enemies at a time, as due to the camera view typical to the third-person shooters they would not fit into a screen. Therefore, the visibility information does not need to be stored in a precomputed table and it is not limited to a number of directions. As it can be seen Figure 4.38, the fact that the visibility map is based on waypoints decreases the precision of the analysis as the position of the characters are approximated to the waypoints' positions. For this reason, the terrain analysis in this project uses its own separate representation: a cell based map. While a navigation mesh can be used for influence mapping [34], it is not suited for visibility maps. The polygons constructing a mesh are often too large to represent a single spot. For instance, the obstacles shown in Figure 4.38 have a couple of waypoints behind and in front of them providing a possible cover from the enemy attack. In a navigation mesh, the polygons placed in these locations could be larger than the obstacle and therefore would provide inaccurate visibility information. It is possible to mitigate this issue, as it has been done in *Resident Evil* [22]: from each polygon a number of points is chosen to create a waypoint map similar to the one used in *Killzone*. It may be more accurate depending on the density of the waypoint placement, but such a map itself is a representation separate from the navigation mesh.

4.6.2 The method

Currently, the influence map used in this project indicates objects that are dangerous to an enemy, such as the player and explosive barrels. However, it will be simple to add the support for friendly objects in the later stages of the development. Both influence and visibility maps are represented by cell based grids. The safety or danger level of a cell is determined by a value between 0 and 1. As in *Killzone*, influence maps are created on demand (when an enemy needs to determine a safe location to hide), and cover only the enemy's neighbourhood. To begin with, such an influence map is initialised and the locations of all dangerous objects in its range are indicated on it. At this stage, each object is represented by only one cell. To demonstrate the influence these objects have on the near environment, the map is processed with a Gaussian blur filter. It is a widely used,

image processing technique of blurring images. It works by affecting the value of each pixel by values of its adjacent pixels. As a result, the values of individual pixels are distributed over larger areas. In this case, pixels are represented by map cells. After applying this filter, the values of cells indicating objects' positions are propagated, marking the influence these objects have on its neighbourhood, as shown in Figure 4.39. The maps presented in the figures below are not normally displayed during game execution.

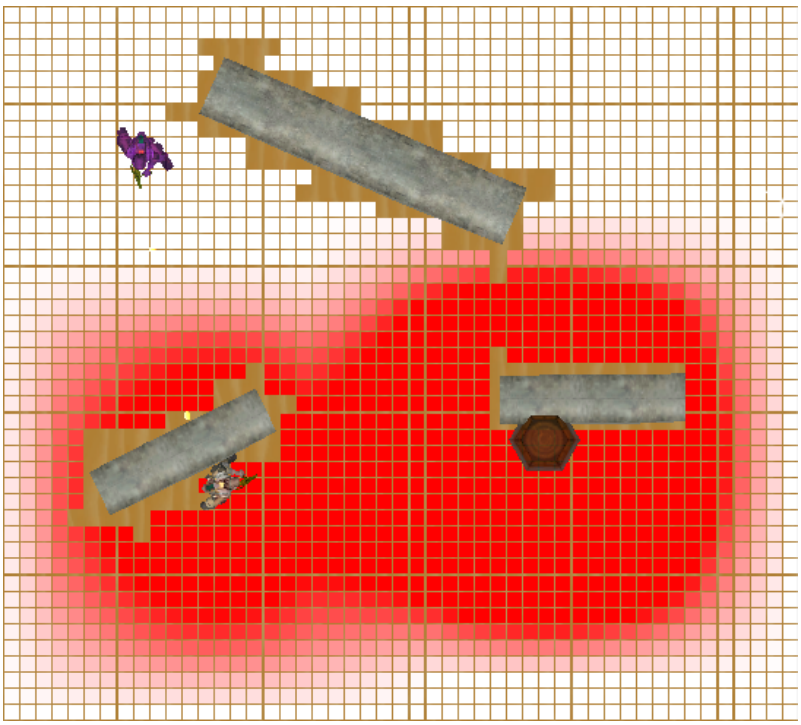

Figure 4.39: An example of an influence map indicating influences of the player and an exploding barrel. The amount of red colour indicates the cell's danger level.

A simpler method of spreading influences is the linear drop off formula, which starts from the object's position with the highest influence and decreases it over a given radius. However, as the influence is strictly limited to the specified radius this method has one disadvantage: small influences do not add up over larger distances. For example, if two exploding barrels are separated by a certain distance, no influence will be indicated in between them.

Influence maps only indicate the locations that should be avoided, and therefore they cannot be used alone to find safe places. To find locations providing cover from the player's attack, the visibility map is generated. It is based on the same cell based grid as the influence map, however to facilitate its further processing, it indicates safety of the locations, not their danger. After it is initialised, a number of line of sight tests are performed between every cell and the player. If a cell cannot be seen by the player, it is marked as safe. Its safety value, however, decreases with the distance between the cell and an obstacle providing the cover, as can be seen in Figure 4.40.

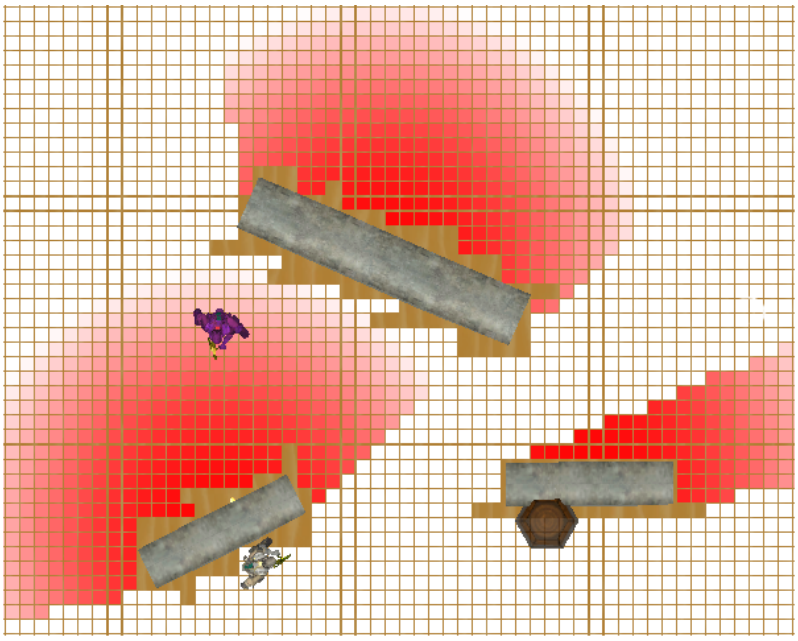

Figure 4.40: An example of a visibility map indicating locations (red cells) that provide cover from the player's line of sight.

To take advantage of both maps, a final map is generated by subtracting the values stored by the influence map (dangerous locations) from the visibility map (safe locations), as shown in the Figure 4.41 (a). This map indicates a large number of secure spots; since such precision is not necessary, the game uses maps of lower resolution, as can be seen in Figure 4.41 (b). With a smaller number of cells to process, the analysis is performed much faster.

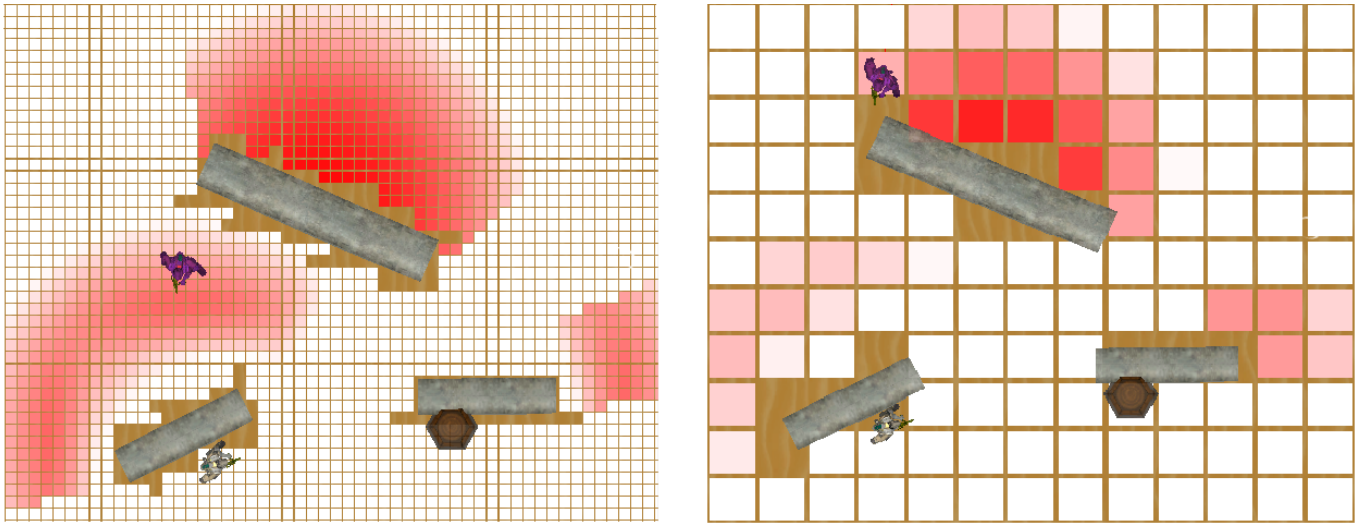

Figure 4.41: The first image (a) shows the final map taking into account both visibility and influence information. The second image (b) presents the same terrain analysed with a map of a lower resolution.

4.6.3 Design and implementation

The results of this analysis (safe locations) are used by an agent to hide from the player, as described in section 4.2. The analysis is performed by the TerrainAnalyzer class, which is added to an enemy as a component. It contains the FindCover method, accessible by the enemy, returning the best cover location. When it is called, the influence and visibility maps are generated and then subtracted to obtain the final map. Finally, a cell with the highest safety value is returned.

The cell based grid is represented using TerrainMap class. It handles map initialisation (the density of cell placement depending on a set cell and map size) and disabling cells that cannot be accessed (those that are either occupied by an object or placed outside of the game world). To represent a single cell, a Cell class is used, which stores cell value, position and information whether it can be accessed. The influence and visibility analysis is performed by InfluenceFilter and VisibilityFilter classes respectively. They contain static methods taking an initial map as an input and returning a processed map with added safety or danger levels.

This structure is presented in the diagram below. At this stage, the TerrainAnalyzer class only contains functionality indicating safe locations, but since terrain analysis can be used for a number of other purposes, such as grenade throwing or suppression fire [6], it will be extended in the future.

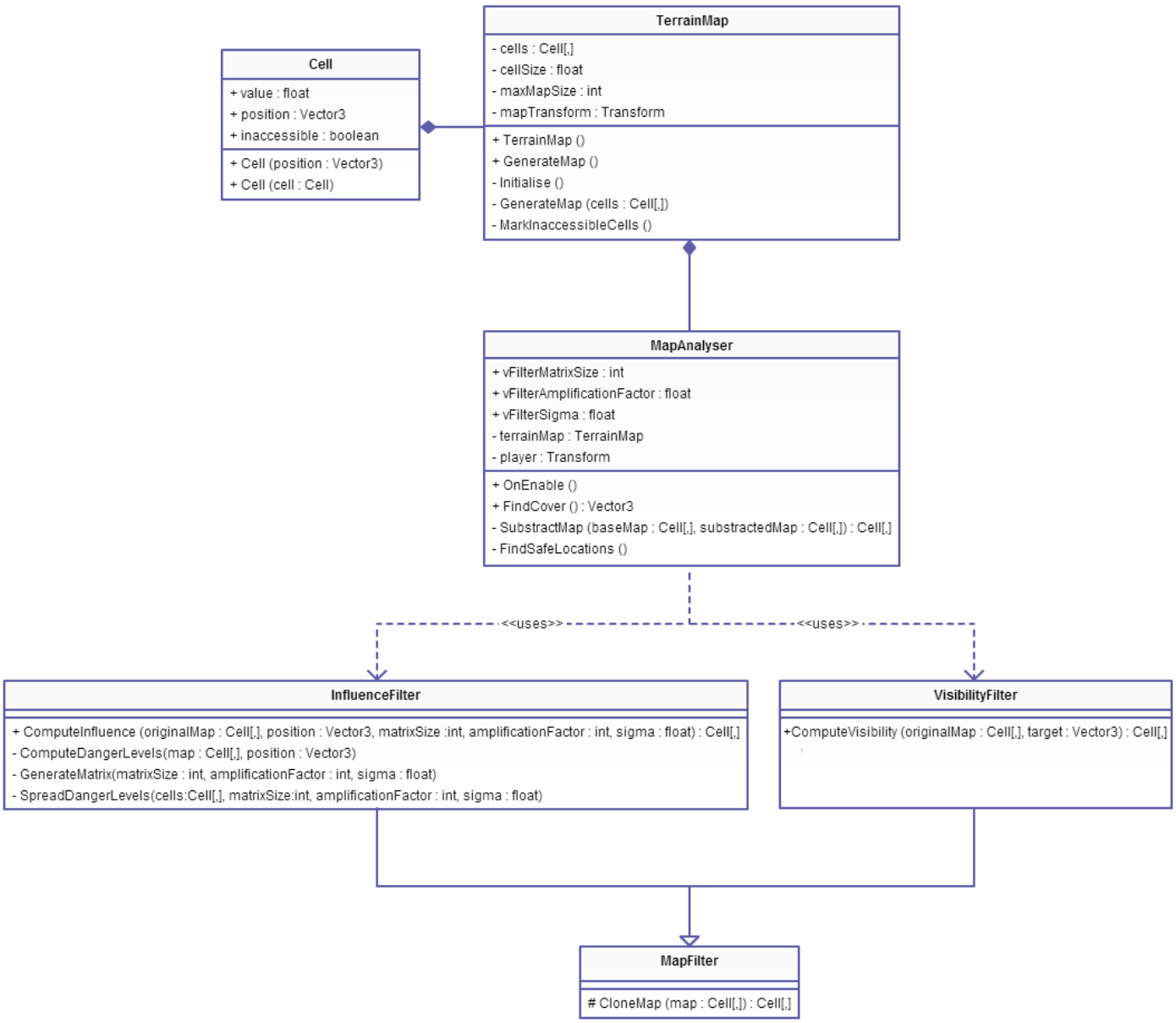

Figure 4.42: A class diagram presenting the structure of the implemented terrain analysis system. As the MapAnalyser is added as a component to the enemy game object, rather than to the enemy controller class, it has not been indicated in the diagram.

4.7 Conclusion

All the methods mentioned above form a greater multi-tier hierarchy with behaviours at multiple levels. This structure is shown in the figure below. The whole process begins by determining a strategy. If an agent is a part of a formation, the formation determines a target to follow. It is then passed onto a state machine and based on a distance to this target, the transition rules of a currently active state determine whether the state should be changed. Otherwise it determines the

agent's actions. The state may take advantage of the terrain analysis to obtain detailed information about the near environment. Then if the agent needs to reach a specified location, a route to this target is planned. Once it is calculated, the character proceeds to move towards the goal by following the route. If it encounters an object not taken into account by the path planning process, the local obstacle avoidance will temporarily change its direction to navigate around the object. Finally, an agent's rotation is adjusted by its motor movement component, which results in a smooth movement.

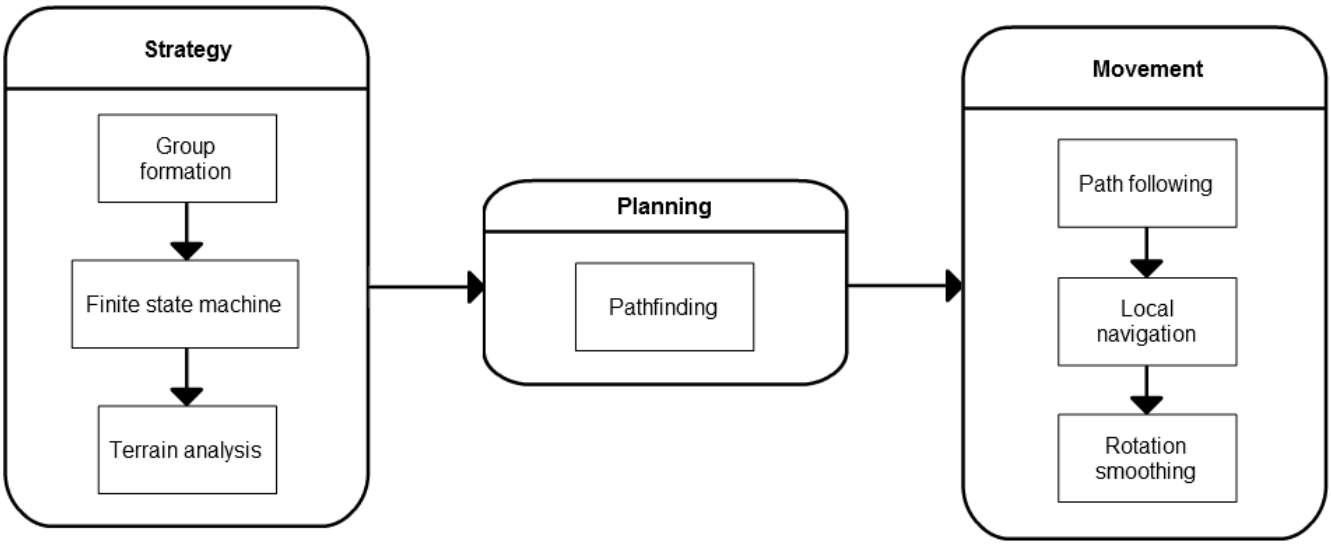

Figure 4.43: The multi-tier AI structure used for the characters in the game. It is a general overview; currently agents in a formation do not have the capability of using terrain analysis.

Chapter 5

Testing Strategy

5.1 Overall Approach to Testing

In accordance to the Scrum methodology [13], the tests were a part of every sprint, ensuring that newly implemented functionality works correctly. Two types of tests have been carried out: automated and system tests.

Due to the nature of this project automated testing of its components was limited. As in most computer games, it is often not possible to determine through code, whether certain functions work properly. For instance, when the player fires a bullet, it cannot be determined procedurally whether the bullet is actually displayed on the screen (unless an access to advanced computer vision library is provided). The second reason includes the constraints imposed by the Unity API. As previously mentioned, every object in the game world performing any kind of action has a component containing a behaviour class (MonoBehaviour) determining the activity. Unfortunately, these classes cannot be instantiated and therefore cannot be procedurally tested. On the other hand, a large part of this project is related to functionality involving artificial intelligence, which is typically based on processing data and therefore should be suited for testing. However, the processed data is often dependent on the game world objects. Therefore, to test such classes, a whole scene would have to be generated. Moreover, results of such data processing cannot be easily evaluated (for instance, placement of polygons in a navigation mesh). For this reason, more emphasis has been placed on non-automated testing.

5.2 Automated Testing

For the reasons mentioned above, only tests of the very basic functions can be carried out automatically. Since no components have been identified that could have been tested as a group, automated integration testing has not been performed and only a set of unit tests has been written. The results of the unit tests are presented in Appendix C.

To carry out the unit tests, an empty game object has been created with a Tester component attached to it. This component executes following tests:

- A* search test - A test checking whether the implementation of A* algorithm works properly by invoking CalculatePath function from the AStarHelper class on a sample graph.
- Object oriented intersection test - A test of a function determining whether two rectangles intersect (used for the navigation mesh generation).

- Fuzzy logic - A set of tests written for classes related to fuzzy logic, including the FuzzyModule, FuzzySet and FuzzyRule classes.

5.3 System Testing

To ensure that despite the limited number of unit tests the system works properly, after each iteration a thorough system testing was performed. These tests verified the following functionality:

- Basic player behaviour (walking, shooting),
- weapon management (weapon changing, weapon types),
- navigation mesh generation (static and dynamic),
- pathfinding and path following,
- finite-state machines (state changing),
- enemy formations,
- fuzzy logic weapon management,
- and terrain analysis.

The results of those tests are presented in Appendix C. The tests above were performed under regular circumstances. To ensure that the system will be able to work in more difficult conditions in the future, a couple of the tests mentioned above were extended by additional stress testing:

- Navigation mesh generation has been executed in more a complex environment (numerous, not axis-aligned objects) to examine the obtained world coverage and generation time. Additionally, as the mesh generated under such circumstances contains a higher number of polygons, the performance of pathfinding and path following was tested. Quantitative results of navigation mesh generation are presented in Appendix B.
- A number of enemies has been increased to examine whether it would result in lower performance or other unforeseen issues. This test also involved creating large enemy formations.

To ensure that the game works on different devices, a compatibility testing has been carried out. The tests mentioned above have been repeated on a computer with lower computational power (Windows 8, different than the computer used for development), and on a tablet (Android). These tests were successful as no performance issues were identified.

Additionally, an acceptance testing has been carried out by a person not involved in the game development. Since the game is not yet fully developed, it was not a final acceptance test; the user mainly assessed the game in terms of the enemy complexity. The user remarked that while the enemies behave intelligently, due to the low quality of animations of the 3D models, they move unnaturally. As this project was mainly focused on aspects related to artificial intelligence, the visual features of the game will be enhanced at later stages of the development. The user also pointed out that intelligence of certain enemies needs further adjustments such as increasing attack inaccuracy of the enemy with fuzzy logic weapon handling.

Chapter 6

Evaluation

6.1 Objectives achieved

The project has achieved its principal aim of producing a game involving a number of enemies exhibiting behaviours of various complexity. The acceptance testing has confirmed that despite the need of further adjustments, the enemies are challenging for an end-user and therefore the game provides an immersive experience. The stress testing demonstrated that the developed components work correctly in various conditions and therefore can be used in future development of this project. Additionally, their implementation has been designed to be scalable in order to facilitate further development of the game. Although, there is a number of features which has not been implemented (described below), the game contains the core functionality. Therefore, given the time available, the whole project can be regarded as successful.

6.2 Issues

However, during the development of the project, certain issues have been identified that could not be solved due to the time constraint. As mentioned in the section describing the implementation of finite-state machines, the scalability of the project can be improved by introducing hierarchical finite-state machines. Another improvement that could be introduced would be to modify the state machine implementation to be event-driven as opposed to the current polling based approach.

Initially, the game maps were intended to be simple and planar, as in many other mobile games [9]. Therefore, the implemented navigation system is capable only of 2D navigation, and it cannot lead agents to locations placed higher or lower than its underlying navigation mesh. Extending the system to cope with 3D geometry should not be problematic, as such a process has been presented by the creators of the algorithm on which the current navigation mesh generation is based [26]. Another part of the navigation system that could be improved is pathfinding. As the path following method involves frequent recalculations of the followed path, rather than computing it each time using A* algorithm, a lookup table could be implemented, to store the paths that have already been calculated.

During the development it turned out that the initial choice of the programming language was not optimal. Apart from UnityScript used in this project, the Unity engine supports C# which is considered to be more difficult, but contains useful features not available in UnityScript, such as support for generic and abstract classes. Unfortunately the disadvantages of the chosen language have been identified in late stages of the development and switching to C# would require rewriting several dozen source code files. As it would be too time consuming, the language has not been

changed.

6.3 Future improvements

Moreover, there is a number of enemy-related features that have not been implemented, due to the time constraints. Currently, the enemies in the game are automatically activated when the player is noticed to be in close proximity. A more natural approach, which will be implemented in the future, will involve an enemy autonomously wandering around a fixed area or following a predefined patrol route. Once the player enters its field of view, it will begin to attack. Additionally, to make the actions performed by enemies appear less automatic, each action (change of state) will be delayed by a certain reaction time.

The terrain analysis is currently used to determine safe cover locations, but it can be extended to produce more complex behaviours, such as intelligent grenade throwing, suppression fire and tactical pathfinding [6]. In such a case, enemies will take advantage of existing fuzzy logic modules to choose the most appropriate behaviour. Additionally, the behaviour of the enemy using the terrain analysis will need to be improved as it currently is not context-sensitive. For instance, if a hiding enemy is suddenly attacked by the player, instead of running away, it continues to wait until its weapon is loaded.

Since the only way of detecting user input on mobile devices is through a touch screen, additional icons need to be designed to create interface enabling weapon change, which currently is possible only on PC version.

Naturally, this project is only a basis for further development. A fully completed game will include several weapons, visual effects (such as explosions), a number of larger game scenes, higher quality 3D models, sound effects and music. Additionally, as previously mentioned, the behaviour of the currently implemented enemies will not only be extended, but also adjusted.

Appendices

Appendix A

Third-Party Code and Libraries

The project was developed in Unity3D game engine. As previously mentioned, the game uses following components taken from Unity sample project [20]:

- PlayerController,
- PlayerMovement,
- TriggerOnMouseOrJoystick (taking input from mouse and keyboard or touch screen),
- Spawner (facilitating instantiation of object during game execution),
- PlayerAnimation
- and SignalSender (component messaging system).

The models currently used in the project have been derived from the Unity sample project and Unity Asset Store (build-in the Unity environment). They are free to use in both personal and commercial purposes.

Additionally, for breaking objects into pieces a the project uses SimpleFracture script published on Unity message board [35].

Appendix B

Navigation mesh generation results

This appendix presents results obtained using the implemented navigation mesh generation method.

2.1 Parameter adjustment

Static mesh generation has two main parameters which need to be manually chosen: number of initially placed rectangles and number of iterations of the algorithm (determining how thorough the filling process is). Figure B.1 presents results obtained with 12 initial rectangles with 3 iterations (a) and 1 initial rectangle with 10 iterations (b). Both methods provided similar coverage levels (95.50% to 94.87%). Filling map using only one initial rectangle, generates larger rectangles and therefore results in lower number of overall rectangles (26 to 31). However it is slightly slower (0.21 seconds to 0.19 seconds) and it is difficult to choose the appropriate number of iterations to fully fill the game world. It is also worth noting that the generated rectangles are not just larger, but often longer, which may lead to problems related to path following described in the graph construction subsection. For this reason, the game uses the first approach.

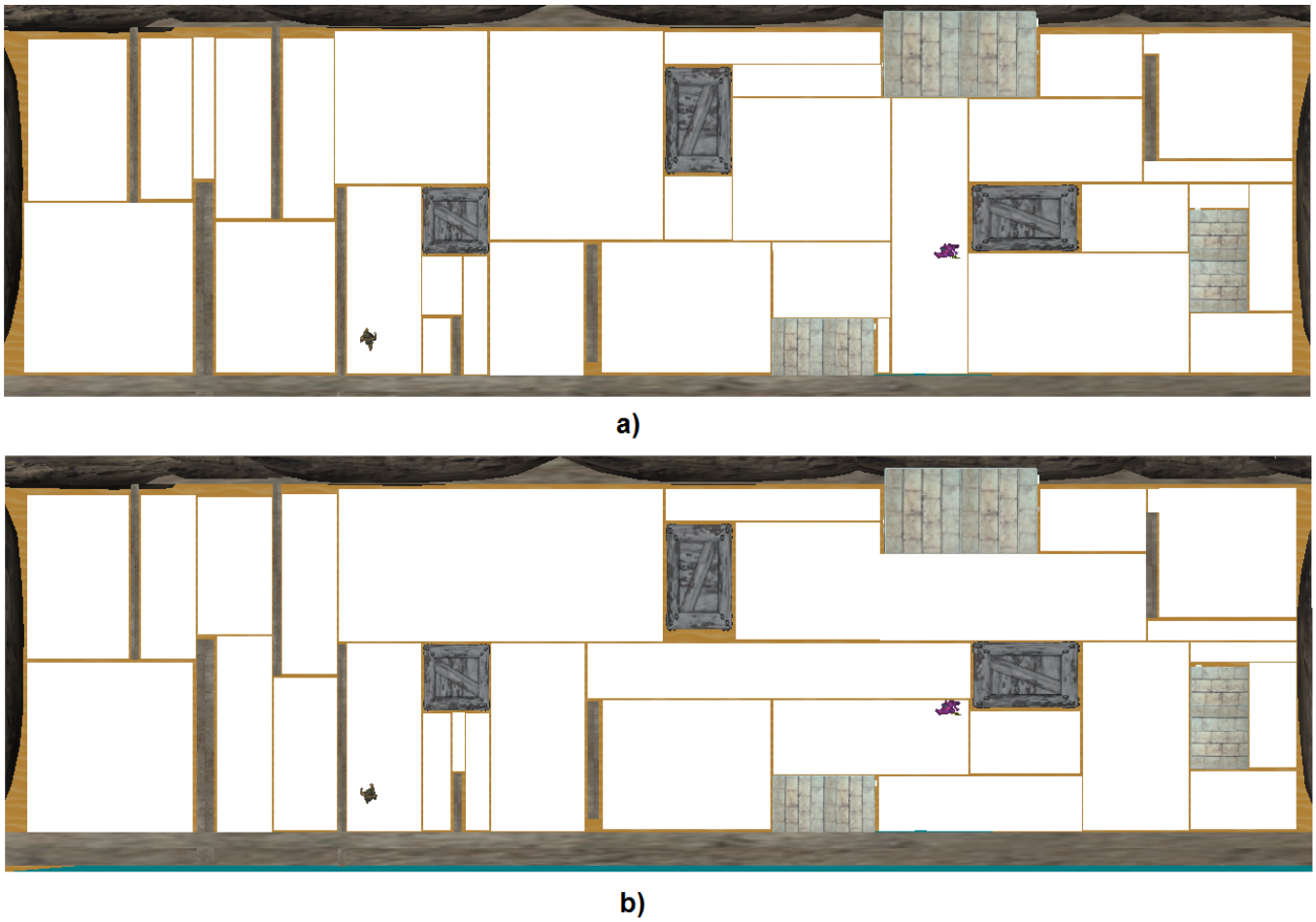

Figure B.1: Game world decomposed using a high number of initial rectangles (a) and one initial rectangle (b).

2.2 Complex environment

Due to the previously mentioned problems regarding non-axis-aligned objects, to ensure that the navigation mesh will cover the map to a satisfactory degree the generation has been tested on a game world containing such objects, as can be seen in Figure B.2. The generation time was slightly longer (0.23 seconds), however the number of rectangles was substantially higher (89). Moreover, the game world was covered only in 90.68%. On the other hand, since all possible passages have been covered such mesh could have been used successfully for navigation. Additionally, the presented game world is overly complex. One of the reasons for modifying the original algorithm providing full world coverage, was that the game maps are not intended to contain many non-axis aligned objects.

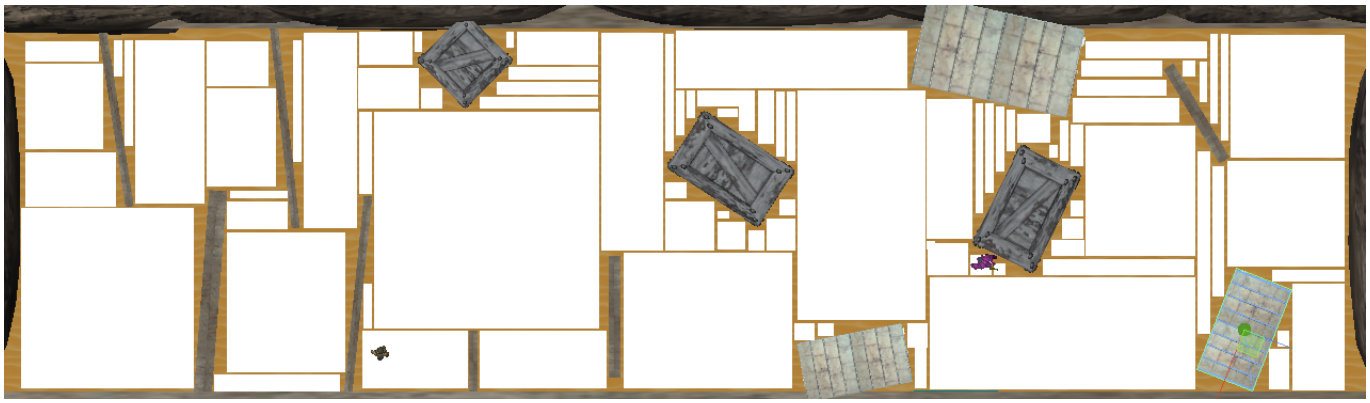

Figure B.2: Navigation mesh generated for a game world consisting of non-axis aligned objects.

2.3 Dynamic mesh generation

Figure B.3 presents four stages of dynamic mesh generation. The process starts with a mesh consisting of one rectangle. As the agent discovers obstacles, the mesh is updated to be more accurate. An average time of each update was 0.05ms.

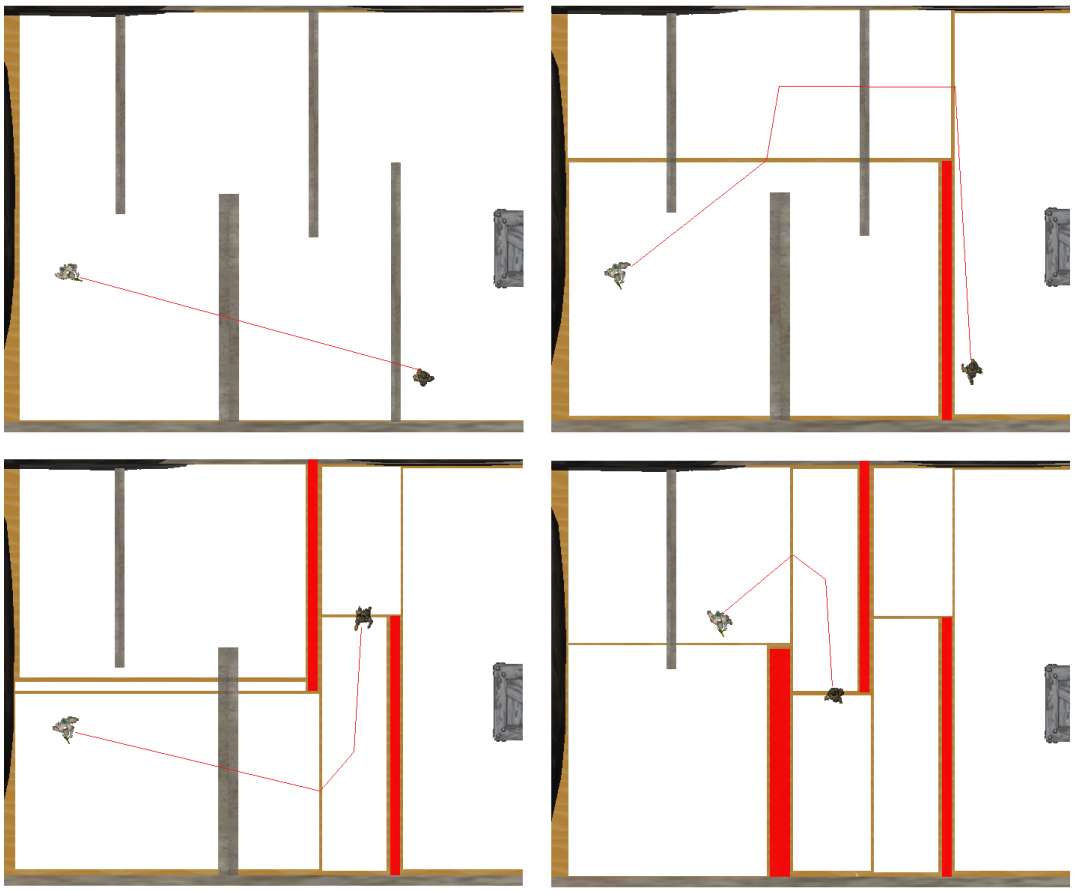

Figure B.3: Dynamic mesh generation using NMAD algorithm. The obstacles discovered by the enemy are marked in red. The red line represents the path followed by the enemy.

Appendix C

Testing results

3.1 Unit testing

Description	Fixture	Input	Output	Pass/Fail
AStarHelper CalculatePath	Graph (n1, n2, n3, n4)	n1, n4	n1, n2, n3, n4	Pass
LeftShoulderSet CalculateDom	LeftShoulderSet (peak point = 0, right offset = 1)	0.5	0.5	Pass
RightShoulderSet CalculateDom	RightShoulderSet (peak point = 1, left offset = 1)	0.5	0.5	Pass
FuzzyORRule constructor	-	[0, 1, 2, 3, 4], 0	4	Pass
FuzzyANDRule constructor	-	[0, 1, 2, 3, 4], 0	0	Pass
FuzzyModule CombineOR	FuzzyModule object	[1], 2 [2], 1	1	Pass
FuzzyModule CombineAND	FuzzyModule object	[1], 2 [2], 1	2	Pass
FuzzyModule Defuzzify	RightShoulderSet(1, 0, 1) LeftShoulderSet(0, 1, 0)	ANDRule(0, set1); ANDRule(1, set2)	0	Pass
FuzzyAND	-	0, 1	0	Pass
FuzzyOR	-	0, 1	1	Pass
FuzzyFairly	-	9	3	Pass
FuzzyVery	-	9	81	Pass
OBBIntersection Test 1	rectangle1([0,0], [10,10]) rectangle2([5,5]; [2,2])	rectangle1, rectangle2	true	Pass
OBBIntersection Test 2	rectangle1([0,1], [1,1], [2,1], [0,0]) rectangle2([0,0], [0,1], [1,1], [1,0]);	rectangle1, rectangle2	true	Pass
OBBIntersection Test 3	rectangle1([10,10], [11,10], [11,11], [10,11]) rectangle2([0,0], [0,1], [1,1], [1,0]);	rectangle1, rectangle2	false	Pass

Table C.1: Results of unit testing.

3.2 System testing

Function tested	Action	Results	Pass/Fail (PC1)	Pass/Fail (PC2)	Pass/Fail (Tablet)
Player movement	Use arrow keys (or on-screen controls on tablet) to move the player in different directions.	The player moves towards indicated directions.	Pass	Pass	Pass
Shooting	Press left mouse button (or tablet screen).	The player shoots.	Pass	Pass	Pass
Weapon changing	Use mouse scroll to switch between weapons.	Weapons change.	Pass	Pass	-
Weapon types	Press left mouse button (or tablet screen) to initiate shooting. Then change the current weapon and shoot again.	The frequency of fired bullets is different depending on a weapon type.	Pass	Pass	-
Pathfinding (no obstacles)	Approach an enemy not separated by any obstacle and wait.	The enemy walks towards the player.	Pass	Pass	Pass
Pathfinding and path following	Approach the enemy separated by obstacles and wait.	The enemy finds a way around the obstacles to reach the player.	Pass	Pass	Pass
State changing	Approach any of the enemies to activate it and wait.	The enemy follows the player and begins to attack.	Pass	Pass	Pass
Navigation mesh generation (static)	Launch the game. Approach the enemy separated by obstacles and wait.	The game launches instantly. The enemy finds a way around the obstacles to reach the player.	Pass	Pass	Pass
Navigation mesh generation (dynamic)	Approach an enemy separated by obstacles and wait.	The enemy finds a way around the obstacles to reach the player. The game frame rate is not affected.	Pass	Pass	Pass
Formation's shape	Approach line or circle enemy formation.	The enemies align and move according to the predefined formation's shape.	Pass	Pass	Pass
Safety of formation's slots	Approach line or circle enemy formation and shoot one of its enemies.	The enemy shot moves backwards, while other enemies move forwards.	Pass	Pass	Pass
Enemy weapon management	Approach an enemy carrying multiple weapons. Once it is close, move back.	The enemy changes weapon depending on a distance to the player.	Pass	Pass	Pass
Terrain analysis	Approach the last enemy type.	The enemy hides behind an obstacle, far from dangerous objects.	Pass	Pass	Pass

Table C.2: Results of system testing.

Annotated Bibliography

[1] S. M. Lucas, "A New IEEE Transactions," *Computational Intelligence and AI in Games*, vol. 1, pp. 1–3, 2009.

An article briefly covering the topic of artificial intelligence in games and motivating its further development.

[2] Wikipedia, "Video games as an art form," http://en.wikipedia.org/wiki/Video_games_as_an_art_form, 2011, accessed April 2013.

An article describing the concept of video games as a new form of art. It lists highly acclaimed games, which may provide inspiration in further development of visual aspects of the game.

[3] Wikipedia, "Crysis 2," http://en.wikipedia.org/wiki/Crysis_2, 2012, accessed April 2013.

An article about an advanced first-person shooter game Crysis 2, describing its visual aspects.

[4] I. Millington, *Artificial Intelligence for Games*. Elsevier/Morgan Kaufmann, 2006.

A comprehensive book describing various artificial intelligence techniques used in games. The presented implementations of some of these techniques (for instance, fuzzy logic) were considered during the development of this project.

[5] Wikipedia, "A* search algorithm," http://en.wikipedia.org/wiki/A*_search_algorithm, 2012, accessed April 2013.

An article describing the A* search algorithm. It also includes a pseudo code snippet which was especially useful while implementing the A* algorithm in this project.

[6] R. Straatman and A. Beij, "Killzones AI: dynamic procedural combat tactics," in *Game Developers Conference*, 2005.

A detailed description of terrain analysis used in first-person shooter game Killzone which inspired the implementation of a similar method used in this project. Moreover presents various methods of using the information obtained by such analysis including intelligent grenade throwing, suppression fire and tactical pathfinding.

[7] M. Booth, "The AI Systems of Left 4 Dead," in *Artificial Intelligence and Interactive Digital Entertainment Conference*. The AAAI Press, 2009.

A detailed description of the artificial intelligence techniques used in first-person shooter game Left 4 Dead, including reactive path following implemented in this project.

[8] D. Charles and S. McGlinchey, "The past, present and future of artificial neural networks in digital games." in *Proceedings of the 5th International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004, pp. 163–169.

A thorough overview of methods implementing agent learning using neural networks in contemporary games. It was one of the considered approaches of increasing the enemy difficulty in this game.

[9] Angry Mob Games, "Guerrilla bob," http://www.angrymobgames.com/guerrilla-bob.html, 2011, accessed April 2013.

A website containing information about a very popular mobile third-person shooter Guerrilla Bob, similar to the game developed in this project, in terms of structure and size of the game world (typical for mobile third-person shooters).

[10] Madfinger Games, "Shadowgun," http://www.madfingergames.com/g_shadowgun.html, 2013, accessed April 2013.

A developer website containing information about advanced mobile third-person shooter Shadowgun. It briefly describes the featured enemies and actions that they are capable of taking.

[11] T. Wimberly, "Shadowgun. bounty hunting has never looked this good," http://androidandme.com/2011/10/games/shadowgun-bounty-hunting-has-never-looked-this-good/, 2011, accessed April 2013.

A review of an advanced third-person shooter game Shadowgun. Apart from its advantages, it describes its disadvantages which include predictable enemy actions.

[12] Wikipedia, "Scrum (development)," http://en.wikipedia.org/wiki/Scrum_(development), 2012, accessed April 2013.

An article describing an agile software development methodology Scrum, used throughout the development of this project.

[13] C. Keith, *Agile Game Development with Scrum*. Addison-Wesley Professional, 2010.

A comprehensive book describing Scrum methodology and motivating its use in game development.

[14] Unity Technologies, "Unity," http://unity3d.com/unity/, 2013, accessed April 2013.

A website describing in detail Unity3D, the 3D game engine used for development of this project.

[15] P. Petridis, I. Dunwell, S. de Freitas, and D. Panzoli, "An engine selection methodology for high fidelity serious games," in *Proceedings of the Second International Conference on Games and Virtual Worlds for Serious Applications, VS-GAMES*. IEEE Computer Society, 2010, pp. 27–34.

A comparison of the most popular game engines in terms of their applicability in serious game development. As it is a very demanding field, this comparison influenced the choice of the game engine used in this project.

[16] W. A. Mattingly, D.-J. Chang, R. Paris, N. Smith, J. Blevins, and M. Ouyang, "Robot design using unity for computer games and robotic simulations," in *Proceedings of the 17th International Conference on Computer Games, CGAMES*. IEEE Computer Society, 2012, pp. 56–59.

A paper presenting an approach of designing robots in Unity3D game engine. It also contains a brief overview of its structure and interface.

[17] C. Stoy, "Game object component system," in *Game Programming Gems 6*, M. Dickheiser, Ed. Charles River Media, 2006, pp. 393–403.

An overview of game object component architecture commonly used in computer games.

[18] Unity Technologies, "Scripting overview," http://docs.unity3d.com/Documentation/ScriptReference/, 2012, accessed April 2013.

A general documentation of scripting in Unity game engine. It was the main source of knowledge about the UnityScript language throughout the development of this project.

[19] W. Goldstone, *Unity 3.X Game Development Essentials*. Packt Publishing, 2011.

An extensive book about game development in Unity3D engine. It presents commonly used approaches in Unity development such as structuring characters using player and enemy controllers.

[20] Unity Technologies, "Demo projects," http://unity3d.com/gallery/demos/demo-projects, 2013, accessed April 2013.

A website listing a number of Unity demo projects, including AngryBots whose components has been used in development of this project.

[21] M. Buckland, *Programming Game AI By Example*. Wordware Publishing, Incorporated, 2005.

An extensive book about artificial intelligence techniques commonly used in game development. It influenced the implementation of finite-state machines used in this project.

[22] N. Bamford, "Situational Awareness: Terrain Reasoning for Tactical Shooter AI," in *Game Developers Conference*, 2005.

A description of navigation and terrain analysis techniques used in *Resident Evil: Operation Raccoon City* game, considered during the development of this project.

[23] D. H. Hale, "A growth-based approach to the automatic generation of navigation meshes," Ph.D. dissertation, The University of North Carolina at Charlotte, 2011.

A Ph.D thesis written by author of the DEACCON algorithm. It contains a comprehensive comparison of methods used for autimatic mesh navigation generation in computer games.

[24] E. Games, "Navigation mesh reference," http://udn.epicgames.com/Three/NavigationMeshReference.html, 2012, accessed April 2013.

Description of navigation mesh used in Unreal Engine 3. It presents an approach which was considered while implementing a mesh generation method in this project.

[25] P. Tozour, "Search space representations," in *AI Game Programming Wisdom 2*. Charles River Media, 2004, pp. 85–102.

An overview of different search space representations, including the space-filling volumes.

[26] D. H. Hale, G. M. Youngblood, and P. N. Dixit, "Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. The AAAI Press, 2008.

A paper introducing the DEACCON algorithm and presenting its advantages over other approaches commonly used for mesh generation.

[27] D. H. Hale, G. M. Youngblood, and N. S. Ketkar, "Using intelligent agents to build navigation meshes," in *Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2010.

A paper introducing the DASFV algorithm handling dynamic addition and removal of object from navigation meshes.

[28] D. H. Hale and G. M. Youngblood, "Dynamic updating of navigation meshes in response to changes in a game world," in *Proceedings of the Twenty-Second International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2009.

A paper introducing the NMAD extension to the DASFV algorithm enabling dynamic creation of navigation mesh during game playing.

[29] X. Cui and H. Shi, "An overview of pathfinding in navigation mesh," *International Journal of Computer Science and Network Security*, vol. 12, no. 12, p. 48, 2012.

An overview of methods constructing graphs based on a navigation mesh. It also presents a funnelling algorithm which optimises a path connecting two polygons.

[30] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

A paper introducing the D* Lite search algorithm, greatly improving replanning process in dynamic environments.

[31] Wikipedia, "Hyperplane separation theorem," http://en.wikipedia.org/wiki/Hyperplane_separation_theorem, 2012, accessed April 2013.

An article describing the separating axis theorem and its use in collision detection. It was used in this project to implement a test detecting intersection between non-axis-aligned objects.

[32] D. Johnson and J. Wiles, "Computer games with intelligence," in *Proceedings of the 10th IEEE International Conference on Fuzzy Systems*. IEEE, 2001, pp. 61–68.

An article describing various artificial intelligence techniques used in popular games which were considered during development of this project.

[33] P. Tozour, "Influence mapping," in *Game Programming Gems 2*. Charles River Media, 2001, pp. 287–297.

An article describing and motivating the use of influence maps in computer games.

[34] F. W. P. Heckel, G. M. Youngblood, and D. H. Hale, "Proceedings of the 4th international conference on foundations of digital games," in *FDG*. ACM, 2009, pp. 79–85.

A paper written by authors of the DEACCON algorithm, describing a method of influence mapping in navigation meshes.

[35] BTM, "Unity forums, simple fracture script [v 1.01]," http://forum.unity3d.com/threads/57994-Simple-Fracture-Script-v-1-01, 2010, accessed April 2013.

A website containing link to a script facilitating shattering 3D models during game execution, used in this project.