SHEFFIELD HALLAM UNIVERSITY

Faculty of Arts, Computing, Engineering and Sciences

CONTROLLING NON-PLAYABLE CHARACTERS (NPC) IN VIDEO GAMES USING LOCAL KNOWLEDGE

BY

NIPUNA HIRANYA WEERATUNGE

Master of Science in Telecommunication and Electronic Engineering

December 1, 2016

PREFACE

This report describes project work carried out in the Faculty of Arts, Computing, Engineering and Sciences at Sheffield Hallam University between January 2016 and September 2016.

The submission of the report is in accordance with the requirements for the award of the degree of Master of Science in Telecommunication and Electronic Engineering under the auspices of the University.

"Introduce a little anarchy. Upset the established order, and everything becomes chaos. I'm an agent of chaos..."

—The Joker, The Dark Knight (2008)

ACKNOWLEDGEMENT

I would like to extend my sincerest gratitude to my parents, Ajith and Dillanjani and my sister, Wathsala for all the support, and at times the necessary push given to me during my course of study.

My sincere appreciation is extended to Dr.Chathuranga Weeraddana for supervising me throughout this work and for the unwavering support and invaluable guidance given during the course of this thesis.

A special thank you goes to Mr.Prabath Buddhika, the MSc program coordinator and to the Department of Electrical and Computer Engineering.

Without all of you, it is highly unlikely that this thesis would have ever seen the light of day!

ABSTRACT

Pathfinding is an integral part in any video game environment. Pathfinding in video games, in its simplest form, directs an in-game agent from a start position to a destination point. Typically in a video game, one or more characters will be controlled by the inputs from the player. The in-game agents that are not controlled by the player are known as *Non-Playable Characters (NPC)*. A significant component of an *Artificial Intelligence (AI)* system of a video game is reserved for pathfinding for NPC agents.

One of the most commonly used algorithms for video game pathfinding is the *A* algorithm*. From 2 dimensional *Real-Time Strategy (RTS)* games to 3 dimensional *Role-Playing Games (RPG)*, A* based approaches have been used extensively. Windowed Hierarchical Cooperative A* (WHCA*), which is an extension of the A* algorithm, is widely used for multi-agent pathfinding. Although the A* based methods can route agents in an efficient manner, the algorithm needs to have global knowledge of the video game environment in order to operate. In other words, the A* inspired algorithms needs to know the complete path before the agent can be moved. When hundreds of agents needs to be moved across a video game map, the use of global knowledge increases the computational complexity of the problem. A* inspired approaches will need to compute and store the graph for each of the agents in order to operate, imposing a heavy computational strain on hardware. In addition, video game environments are generally dynamic and can change during run time. For instance, a path calculated by the WHCA* algorithm can suddenly be blocked due to the presence of a mobile agent. In such a case, an A* based approach would have to discard the previously calculated path, partially or completely, and would have to recalculate.

In this thesis, we propose a novel pathfinding algorithm, named *LoS by Water-Filling*, that can navigate agents to a target destination in the presence of obstacles using local knowledge. This approach is based on the argument that a water stream is guaranteed to flow from the source to the destination, given that the source and destination are appropriately oriented and the necessary openings are there. The proposed method is a two-part based approach. When *no line of sight (NLoS)* can be established from start position to target destination due to the presence of obstacles, the proposed solution navigates an agent to a *line of sight (LoS)* position to the target. Once *line of sight (LoS)* can be established, the solution uses a *Particle Swarm Optimization (PSO)* based method to converge the agents on the target destination. The essence of this proposed algorithm is implemented on a real gaming environment constructed in GameMaker: Studio. Convergence of all the proposed algorithms is *guaranteed*. The convergence follows the premise that *a stream of water is guaranteed to flow from the source to the destination, in the direction of gravity, irrespective of the starting position, given that the destination is at a lower height than the source and there exists a path between the source and the destination.*

CONTENTS

ACRONYMS

AI Artificial Intelligence

GML GameMaker Language

GMS GameMaker: Studio

LoS Line of Sight

NES Nintendo Entertainment System

NLoS No Line of Sight

NPC Non-Playable Characters

PSO Particle Swarm Optimization

RPG Role-Playing Games

RTS Real-Time Strategy

WHCA* Windowed Hierarchical Cooperative A*

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The history of commercial video games runs back to the early 1970s. *Computer Space* by *Nutting Associates* which was introduced in 1971, is considered to be the first commercially made arcade game. Nonetheless, the prize for the first truly commercially successful arcade video game goes to *Pong* which was produced by *Atari* in 1972. The success of Pong spawned the inception of home video game systems such as the massively successful *Nintendo Entertainment System (NES)* which was released worldwide in 1986. The NES gave birth to classical and beloved video games such as *Super Mario Brothers* and *Donkey Kong* which were monumental in the video game revolution [1]. Even in the early days of the NES, *Non-Playable Characters (NPC)* in video games played a vital role in providing a fulfilling gaming experience to the player.

With the advances of microprocessor technology, video game developers have much more freedom and resources to work with. Compared to the 8 bit microprocessors that ran the NES and Atari game consoles, game designers today have access to 64 bit microprocessors possesing substantially more processing power. Even though these resources can essentially be allocated across the main components of a video game such as graphics/visuals, Artificial Intelligence (AI) and systems that control the inputs from the player, many of the recent games experienced faster advances in graphics/visuals compared to AI [2]. Hence, it is safe to say that systems that control NPC in video games have seen only a lesser share of the computing resources allocated to them.

Video game AI systems oversee the controlling of NPC as well as the interactions between the player and the video game. One of the fundamental aspects of the AI system is pathfinding. Pathfinding, in its simplest terms, would be to direct in-game agents from a starting position to a destination position. However, an AI system has to control the NPC in a way that exhibits some form of intelligent behavior. For instance, not many video game players would flock to play a game that haphazardly steers enemy NPC away from the players when in actuality the NPC need to be moving towards them.

1.1 Motivation

The motivation for this thesis comes from the behavior of NPC in *Real-Time Strategy (RTS)* games. In many RTS games, such as *Age of Empires* series, *World of Warcraft* series and *League of Legends* series, often a large number of agents need to be moved across the map across different types of terrain. Although the prevailing pathfinding algorithms perform relatively well in these types of scenarios, they require global knowledge of the video game environment to operate. In other words, these algorithms need to know

the complete path before an agent can be moved. When there are hundreds of agents that need to be moved across a video game map, the computational complexity of the pathfinding problem increases significantly. Algorithms that use global knowledge require the graph to be calculated and stored for each of these agents. This can impose a heavy computational strain on the used hardware, especially when a large number of agents needs to be navigated through a sizeable video game map.

Thus, there is a pressing need for algorithms that can operate using local knowledge. Essentially, a pathfinding algorithm that employs local knowledge only needs to know the target destination. It does not need to know the complete path or even the location of the obstacles and are more suited for dynamic video game environments. Hence, a pathfinding approach that employs an algorithm that uses local knowledge reduces the computational strain on hardware while better adapting to dynamically changing video game environments. This conundrum served as the motivation for designing a pathfinding algorithm that can operate using local knowledge.

1.2 Research Goal and Outline of the Thesis

The goal of this thesis is to design a novel algorithm that is capable of routing agents to a target destination in the presence of obstacles using local knowledge. The proposed solution follows the premise that a water stream is guaranteed to flow from its source to the destination, in the direction of gravity given that the necessary openings are there. We propose a two-part based solution in this thesis which is divided into the no line of sight (NLoS) instance and the line of sight (LoS) instance. The proposed solution is capable of navigating agents from a starting position to a target destination in the presence of obstacles using local knowledge.

Chapter 2 is reserved for investigating the existing video game pathfinding approaches. Mainly, the A* algorithm and one of its extensions, the Windowed Hierarchical Cooperative A* (WHCA*) algorithm will be explained in detail. In addition, areas that can be improved such as pathfinding using local information while eliminating the need for global information will be elaborated upon. We will also probe into the classic Particle Swarm Optimization (PSO) algorithm and its workings which serves as the basis for the employed LoS algorithm.

Chapter 3 will explore the main contribution of this thesis. In this chapter, we introduce the *LoS by Water-Filling* algorithm which routes agents to a line of sight (LoS) position using local knowledge when no line of sight (NLoS) can be established due to the presence of obstacles. This section will explain the necessary mechanics and the behavior of the proposed solution that is needed to control NPC in a gaming environment. The intuition that was the inspiration behind the designed algorithm will also be explained in detail.

Chapter 4 is dedicated to explaining the employed LoS algorithms which are extracted from existing literature. All of the algorithms that are presented here are based on the classic PSO paradigm. Four PSO based algorithms will be discussed as potential candidates for the LoS instance. Out of the presented four algorithms, one is chosen as the LoS algorithm.

Chapter 5 will cover the simulation results, the discussion and the testing environment. The obtained simulation results and the subsequent discussion will further solidify the rationale behind the designed algorithm. The game environment that was used for the

simulation and the simulation tool will also be briefly explained.

Chapter 6 will conclude the thesis. A summary of the main results will be presented while potential avenues for future research will also be highlighted.

In the next chapter, we will explore the existing video game pathfinding solutions as well as the mechanics of the classic PSO algorithm. It would also delve into the necessity of efficient pathfinding approaches in video games.

Chapter 2

RELATED WORK

The main objective of this chapter is to discuss pathfinding relating to a video game setting. Then, the existing pathfinding methods are investigated while identifying areas that can be further improved. Finally, the classic PSO algorithm and its working environment is presented. This chapter will essentially serve as the bridge connecting existing methods and the areas that can be potentially improved.

2.1 Pathfinding in Video Games

Pathfinding in its broadest definition, refers to finding the shortest route between two end points. As video game technology develops, pathfinding has become one of the most popular and frustrating problems in the game industry. Modern Real-Time Strategy (RTS) games and Role Playing Games (RPG) often include many scenarios that deal with sending an agent(s) to a predetermined position or towards a position determined by the player. Thus, often the most common application of video game pathfinding is to direct an agent from a starting position to a target destination while avoiding obstacles in the most efficient manner possible [3].
Pathfinding can be broken down into the following categories [4],

- Cooperative pathfinding- each agent is assumed to have full knowledge of the other agents and their planned routes
- Non-Cooperative pathfinding- agents have no knowledge of other agents' plans and must predict future movements
- Antagonistic pathfinding- agents try to get to their destination while preventing other agents from reaching theirs

This thesis deals with Non-Cooperative pathfinding where agents have no prior knowledge regarding the movements of the other agents. The following section explains the currently employed video game pathfinding algorithms.

2.1.1 Existing Pathfinding Algorithms

2.1.1.1 A* Algorithm

Probably, the most commonly used video game pathfinding algorithm is the A* algorithm [5]. It was introduced in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael in the

paper *A Formal Basis for the Heuristic Determination of Minimum Cost Paths* [4]. A* maintains two node lists, a closed list for which optimal paths are known and an open list which contains the search candidates [6].
The pseudo-code of the original A* algorithm is as follows [3]:

Algorithm: A*

---

1. Add the starting node to the open list.
2. Repeat the following steps:
   (a) Look for the node which has the lowest f value on the open list. This node will be referred to as the current node.
   (b) Switch it to the closed list.
   (c) For each reachable node from the current node
      i. If it is in the closed list, ignore it.
      ii. If it is not on the open list, add it to the open list. Make the current node the parent of this node. Record the f, g, and h value of this node.
      iii. If it is on the open list already, check to see if this is a better path. If so, change its parent to the current node, and recalculate the f and g value.
   (d) STOP when:
      i. Add the target node to the closed list.
      ii. Fail to find the target node, and the open list is empty.
3. Trace backward from the target node to the starting node. This is the path.

The A* algorithm finds a path by using a cost function. Specifically, the algorithm chooses a path that minimizes the following cost function,

$$f(n) = g(n) + h(n), \tag{1}$$

where *g(n)* is the exact cost from starting point to any point n and *h(n)* is a heuristic that estimates the cost from point n to the destination.

In order for the A* algorithm to find the most optimum or the shortest path, the used heuristic function must be admissible. This means the heuristic should never overestimate the actual cost of getting to the destination [7]. The used heuristic can be highly problem specific. For instance, for a game map that is abstracted by a square grid that allows four directions of movement, the heuristic that can be used is the *Manhattan distance*. For a square grid that allows 8 directions of movement, the *Diagonal distance* can be used [7].
A* algorithm has several beneficial properties as described below [3]:

1. A* is guaranteed to find a path from the starting node to the destination node, if a path exists.
2. If the heuristic is admissible, the found path will be the optimum one.

3. A* makes the most efficient use of the heuristic which means that no other search method exists that uses the same heuristic function to find an optimum path examining fewer nodes than A*.

Although A* is relatively straight forward to implement and has a good performance characteristic, the overall execution time depends on the problem size and complexity [8]. For instance, in a rectangular grid of 100 x 100, in order to find a diagonal path, the traditional A* algorithm must at least explore 513 nodes [7]. The biggest drawback of the A* algorithm is its need to have global knowledge to operate as well as the response time being the same as the overall time. This is because the overall solution needs to be known before an agent can start moving [8].

The next section details a variation of A*, the Windowed Hierarchical Cooperative A* (WHCA*) algorithm, which is used when it comes to directing several agents to their destinations. The main emphasis would be on the operating mechanism of WHCA* as well as areas for potential improvement.

2.1.1.2 Windowed Hierarchical Cooperative A* Algorithm

Even though the classic A* algorithm can route a single agent to its destination, multi-agent pathfinding has to be used when dealing with multiple agents. This thesis will be briefly looking at a variation of the A* called Windowed Hierarchical Cooperative A* (WHCA*), which accommodates multi-agent pathfinding. Windowed Hierarchical Cooperative A* combines several different ideas to achieve efficient cooperative pathfinding. The main aspects of WHCA* are as follows:

- Agent hierarchy - One of the important aspects of WHCA* is agent ordering. An agent is chosen, randomly or depending on a pre-arranged order and is given priority over other agents to plan its path. This priority title is varied dynamically to allow each agent to have the highest priority for the shortest possible time. Thus, this approach can provide solutions to navigation problems which otherwise would be impossible using a fixed agent ordering [4].

- Reservation table - The reservation table is shared among all of the agents and holds the path information for each of them. For instance, the agent with the highest priority places or reserves the nodes of its path in the reservation table. Any subsequent agent will check the nodes on their respective path against the reservation table. If the node is already on the reservation table, it is skipped as that node cannot be reached [6]. Once all the agents reach the end of their path, the reservation table is cleared.

- Windowing - Windowing is the limiting of the path length of the involved agents. Using the reservation table, the cooperative pathfinding can be done relatively efficiently. Nonetheless, as multiple units interact with each other, the initial cost of resolving all conflicts can become restrictively large [6]. This, however, can be mitigated by limiting or windowing the search depth. Each agent searches for a partial route to its destination and begins to follow that path. At regular intervals, the window is shifted forward and a new partial route is calculated [4].

The efficiency of the WHCA* depends on the size of the used window. A smaller window will require the least initialization cost but the algorithm needs to reroute the agents more frequently. A larger window must calculate a large section of the path but with less reroutes. Generally, window sizes of 8, 16 and 32 are used [4].

The pseudo-code for the Windowed Hierarchical Cooperative A* algorithm is as follows [9]:

Algorithm: Windowed Hierarchical Cooperative A*

---

1. Reset reservation table
2. While some agents are not at their goal
   (a) For each agent
      i. Calculate Path (Using A*)
      ii. Reserve first W steps
   (b) For each agent in parallel
      i. Move agent K steps
   (c) Reset reservation table

The approach presented by the Windowed Hierarchical Cooperative A* algorithm improves the original A* paradigm significantly. WHCA* allows multiple agents to reach the destination cooperatively by reducing the number of potential collisions that can occur. It also limits the number of nodes that needs to be searched by windowing the search space. However, WHCA* still suffers from the biggest drawback inherent to the A* approach which is the need for global information. In other words, the algorithm needs to know the complete windowed path before it can start to move the agent. WHCA* will have to compute and store the graph for each of the agents, making this approach computationally complex. When a large number of agents needs to be directed across a complex video game map, the computational strain imposed on hardware is substantial.

In addition, the agents in a video game take a certain amount of time to reach their destinations and much can occur in the game world during this time. For instance, dynamic changes to the map such as an mobile object blocking the previously calculated path can occur making the path obsolete. In such a case, the agent would have to recalculate all or part of its path [10].

Thus, A* based approaches will not be discussed further in this thesis due to the above limitations. The proposed solution aims to overcome the need for the availability of global information while being more suitable for a dynamic game map.

The next section is dedicated to describing the classic PSO algorithm and its workings. This section would act as the foundation to support the notion that a PSO based approach can act as a viable video game pathfinding solution.

2.2 Classic Particle Swarm Optimization Algorithm

Particle Swarm Optimization (PSO) was introduced in 1995 by James Kennedy, a social psychologist and Russell Eberhart, an Electrical Engineer. Initially, PSO was proposed as a method to simulate a simplified social model. However, it was discovered that this method can be used for optimization of continuous nonlinear functions [11].

PSO has its beginnings in two main component methodologies. The more obvious connection that can be associated with PSO is its ties to artificial life (A-life) in general, such as fish schooling and birds flocking. Nevertheless, it is also related to evolutionary computation and has ties to genetic algorithms and evolutionary programming [11]. These relationships will not be discussed in detail in this thesis. PSO, as described by its authors, encompass a very simple concept and paradigms that can be implemented with only a few lines of code. The PSO algorithm only employs primitive mathematical operators and is computationally inexpensive in terms of memory requirements and speed [11].

In PSO paradigm, the entities of the population, which are known as particles, are dispersed across the search space of some problem or function and each member evaluates the objective function at its current location. The search space represents the set of solutions to the used fitness function. Subsequently, each particle then determines its movements through the search space by combining some element of the history of its best found position, with those of the members of the swarm along with some random perturbations. Eventually, the swarm as a whole, similar to a flock of birds foraging for food, is likely to move closer to an optimum solution of the fitness function [12].

PSO is more than just a collection of particles. A particle, by itself, has almost no power to solve any problem. The optimization progress only occurs through particle interaction. Hence, problem solving is a population-wide phenomenon, emerging from individual behaviors of the particles through their interactions with each other [12].

According to the authors, the term *particle* was selected as a compromise. It could be argued that members of the population are mass-less and volume-less, and thus could be referred to as *points*. However, it was decided that velocities and acceleration can more appropriately be attributed to particles, even when they are defined to have arbitrarily small mass and volume [11].
The classic Particle Swarm Optimization algorithm is as follows [13]:

Algorithm: Particle Swarm Optimization

---

1. Initialization
   Set the population array of particles with random positions and velocities on D dimensions in the search space

2. For each particle, evaluate the desired optimization fitness function in D variables

3. Compare the particle's fitness evaluation with its p_{best}. If the current value is better than the p_{best}, then set p_{best} equal to the current value.

4. Identify the particle in the swarm with the best success so far and assign that value to g_{best}.

5. Change the velocity and position of the particle according to equation 2 and 3

$$V(t+1) = v(t) + \phi_1\xi_1[g_{\text{best}} - p(t)] + \phi_2\xi_2[p_{\text{best}} - p(t)] \tag{2}$$

$$P(t+1) = p(t) + v(t+1), \tag{3}$$

where ϕ_1 and ϕ_2 are spring constants with $\phi_1 = \phi_2 = 2$, ξ_1 and ξ_2 are uniformly distributed random variables between 0 and 1 and g_{best} is the global best position of the swarm while p_{best} is the personal best position of each particle.

6. If the stopping criterion (a good fitness value) is met or the maximum number of iterations have been reached, STOP. If not, go to step 2.

The current position $p(t)$ can be considered as a set of coordinates describing a point in the search space. On each iteration of the algorithm, the current position will be evaluated as a potential solution by all the particles. If the current position is better than any position that has been previously found, then its coordinates are stored as the p_{best} or personal best by each particle. This p_{best} value will be used for comparison in future iterations. Each particle communicates with others in the swarm and determines the current best position that has been found by any particle in the population. These coordinates will be stored as g_{best} or global best. The ultimate objective is to keep finding better solutions and update $p(t)$. New points are chosen by adding the velocity vector $v(t+1)$, which can be effectively seen as the step size, to the current position $p(t)$ [12].

With the use of g_{best} and p_{best}, the PSO algorithm achieves in finding a better position with each iteration. The vector $v(t+1)$ is the addition of three vectors, the inertia term $v(t)$, self confidence $h(t)$ and swarm confidence $g(t)$. In other words, each particle is steered towards a position which is in the direction of the resultant vector $v(t+1)$.

The effect of each of these vectors on the particle are as follows:

- The inertia term forces the particle towards the direction it was currently heading.
- The swarm confidence steers the particle towards the direction which has the swarm's currently best found position.
- The self confidence drives the particle towards the direction of the currently best found position of the particle itself.

Thus, in the PSO paradigm, the resultant vector of these three vectors should result in a better optimized position [13]. With each iteration of the PSO algorithm, the particles will move towards the target position. This serves as the main foundation for selecting a PSO based approach for the LoS algorithm. In addition, video game pathfinding movements needs to be visually appealing. In other words, not many players would rally to play a game that display contrived or mechanical movements. PSO, however, presents an elegant model that can exhibit lively and seamless movements. Moreover, as discussed previously, PSO only employs simple mathematical operations. Therefore, it is computationally inexpensive in terms of memory requirements and speed which is of monumental importance when it comes to video game AI. Hence, PSO presents itself as an excellent candidate that can be successfully utilized as a pathfinding algorithm in video game context.

2.3 Summary

In this chapter, existing video game pathfinding algorithms were discussed. The A* and the WHCA* algorithm, which is a descendant of A*, were presented in detail. The drawbacks of the existing algorithms such as the need for global information and having to compute and store the graph for the agents were elaborated.

Next the classic PSO algorithm was explained. The inception, mechanics and the inner workings were given in detail. The chapter then delved into the requirements of a video game pathfinding algorithm such as the need to be visually appealing while performing efficiently. Finally, the reasons for selecting PSO as a video game pathfinding algorithm was discussed as it provides an elegant and efficient movement model.

The following chapter will explore the proposed NPC navigation solution. The inner working of the algorithm as well as the intuition and reasoning behind it will also be discussed. The main emphasis would be on its ability to direct agents in the presence of obstacles using local knowledge.

Chapter 3

PROPOSED SOLUTION

In this chapter we propose a general mechanism for NPC navigation. The presented algorithm will direct an agent to a LoS position to the target destination when NLoS can be established due to the presence of obstacles. The proposed method achieves this by only taking into account local knowledge. In other words, it only needs to know the position of the target to operate.

3.1 NLoS Algorithm

In general, NPC pathfinding needs to be implemented when there is *no* LoS to the target destination. We denote the non LoS situation as NLoS. NLoS mainly exists due to obstacles in the search area of the agent. When there is NLoS, it is clearly noticed that the implementation of NPC pathfinding is challenging. The reason for such a difficulty can technically be explained by the fact that the search area of the agent becoming nonconvex. In the rest of this section, we propose an algorithm which can be used to implement NPC pathfinding, even in a NLoS situation using local knowledge.

3.1.1 Reasoning Behind the Proposed Solution

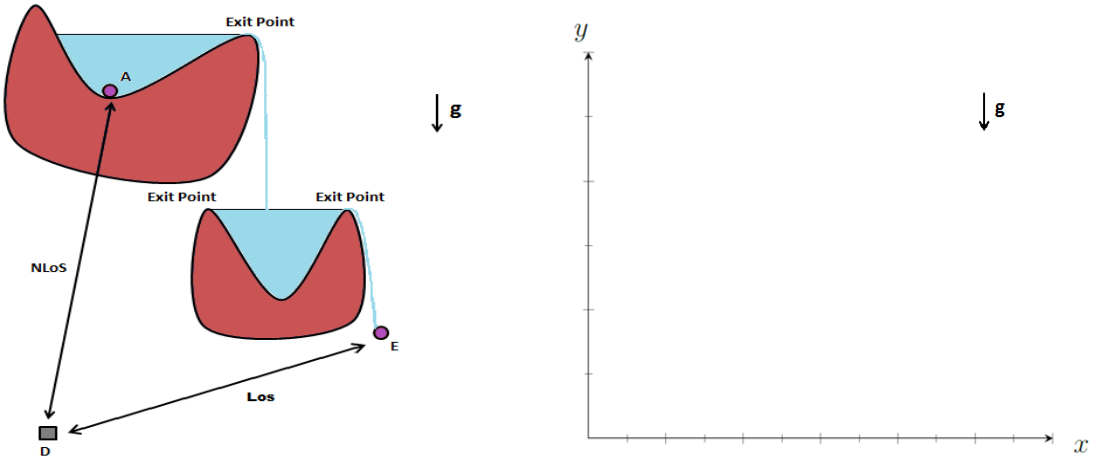
The key idea of our algorithm is based on that, *a stream of water* is guaranteed to flow in the direction of *gravity*. More specifically, irrespective of the starting point `A` of the water stream, it reaches a point `E`, which has a line of sight to any specified target destination `D`, as long as the orientation of the landscape ensures that `D` is at a lower height than `A` *and* there exists a path connecting `A` and `D`. The idea is illustrated in Figure 3.1(a), where **g** denotes the direction of gravitational force.

When applying the above idea to the dynamics of the agent's movement towards the destination, we require

- Water-Filling: an obstacle reshaping mechanism, which resembles the water-filling depicted in Figure 3.1(a) *until* an *exit point* is discovered.

At the exit point, we require

- moving along the boundary of the obstacle, followed by a vertical drop in the direction of **g** *until* another obstacle is encountered.



(a) (b)

Figure 3.1: Reasoning behind the idea: (a) Flow of a water stream originated at A to point E. ; (b) Coordinate system.

The two steps above are to be iterated until LoS between the agent and the target is established. [1]

3.1.2 Notations and Definitions

Let us define some notations to improve the clarity of the algorithm presentation. The map on which the agents and the target reside is assumed to be a rectangle, without loss of generality [2]. The lower left corner of the rectangle is considered the origin. The standard basis vectors $[1\ 0]^\intercal$ and $[0\ 1]^\intercal$ in $\mathbb{R}^2$ are used when defining the coordinates x and y, respectively. This can be seen more clearly in Figure 3.1(b). We denote by $\mathbf{x}^{\mathtt{A}} = [x_1^{\mathtt{A}}\ x_2^{\mathtt{A}}] \in \mathbb{R}^2$ the location of the agent and by $\mathbf{x}^{\mathtt{D}} = [x_1^{\mathtt{D}}\ x_2^{\mathtt{D}}] \in \mathbb{R}^2$ the location of the target. Note that, based on the considered coordinate system, we have $\mathbf{g} = [0\ -1]^\intercal$. Without loss of generality, we assume $(\mathbf{x}^{\mathtt{D}} - \mathbf{x}^{\mathtt{A}}) = \alpha\mathbf{g}$ for some positive α [3] which can be compared with the coordinate system in Figure 3.1(b).

It is assumed that there are p non-overlapping obstacles *inside* the rectangle. These obstacles are modelled with *compact* sets, denoted by $\mathcal{O}_i,\ i = 1, \ldots, p$. The boundary of the rectangle, itself is denoted by $\mathcal{O}_{p+1}$. Furthermore, we denote by $\mathcal{B}_\epsilon(\mathbf{x})$ the ball of radius $\epsilon > 0$, centered around $\mathbf{x}$. More specifically, we have

$$\mathcal{B}_\epsilon(\mathbf{x}) = \{\mathbf{y} \mid ||\mathbf{x} - \mathbf{y}||_2^2 < \epsilon\}. \tag{4}$$

This is illustrated in Figure 3.2.

Next we will introduce some definitions that would aid the comprehension of the proposed algorithm. These would be crucial when explaining the steps of the presented algorithm.

[1] Given there is LoS, the algorithms discussed in Chapter 4 can takeover.

[2] Any arbitrary shaped map can be enclosed inside a rectangle by appropriately defining more obstacles to resemble the original map.

[3] Otherwise, all the obstacles together with the agent and the target can be rotated with coordinate system being fixed to yield the required alignments pointed.

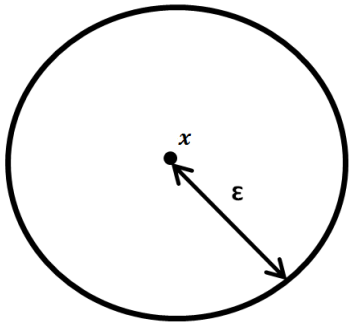


Figure 3.2: Ball

Definition 1 (Slab) *The slab of thickness ϵ_w, originated at $\mathbf{y}$ in the direction of $\mathbf{r} = [d\ 0]^\mathsf{T}$, $d \in \mathbb{R}$, is given by*

$$\mathcal{S}(\mathbf{y}, \mathbf{r}, \epsilon_w) = \{\mathbf{x} \mid [0\ 1]\mathbf{x} \geq [0\ 1]\mathbf{y},\ [0\ 1]\mathbf{x} \leq [0\ 1]\mathbf{y} + \epsilon_w,\ \mathbf{r}^\mathsf{T}\mathbf{x} \geq \mathbf{r}^\mathsf{T}\mathbf{y}\}. \tag{5}$$

Figure 3.3 represents the concept of *Slab*.

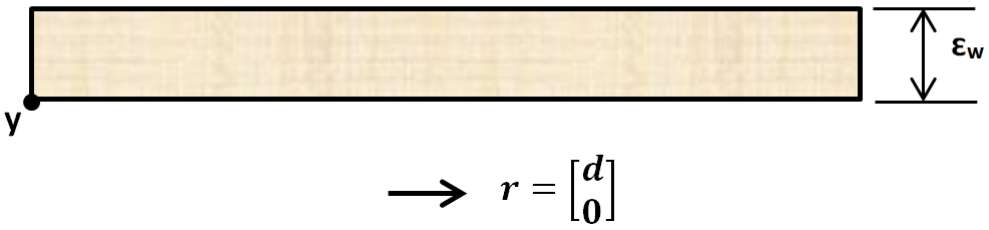


Figure 3.3: Slab

Definition 2 (Exit Set) *For an obstacle $\mathcal{O}$, an agent location $\mathbf{x}^A$, a direction $\mathbf{r} = [d\ 0]^\mathsf{T}$, $d \in \mathbb{R}$, and $\epsilon_w > 0$, the Exit Set is given by $\mathcal{E}(\mathcal{O}, \mathbf{x}^A, \mathbf{r}, \epsilon_w) = \{\mathbf{y} \mid \mathcal{O} \cap \mathcal{S}(\mathbf{x}^A, \mathbf{r}, \epsilon_w)\} \setminus \mathbf{x}^A$. Alternatively, we let $\mathcal{E}(\mathcal{O}, \mathbf{x}^A, \mathbf{r}, \epsilon_w) = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, where $\mathcal{S}_i$, $i = 1, \ldots, n$ are the elements of the Exit Set, numbered in the order they appear, as we move from $\mathbf{x}^A$ along the direction of $\mathbf{r}$.*

Definition 3 (Exit Point) *Given an Exit Set $\mathcal{E}(\mathcal{O}, \mathbf{x}^A, \mathbf{r}, \epsilon_w) = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ for an obstacle $\mathcal{O}$, an agent location $\mathbf{x}^A \in \boldsymbol{bd}(\mathcal{O})$, and positive ϵ_w, with $\mathbf{r} = [d\ 0]^\mathsf{T}$, $d \in \mathbb{R}$, we say $\mathbf{e} \in \mathcal{S}_1$ is an Exit Point if the half-space $\{x \mid \mathbf{a}^\mathsf{T}\mathbf{x} \leq \mathbf{a}^\mathsf{T}\mathbf{e}\}$, with $\mathbf{a} = [0\ 1]^\mathsf{T}$, contains the set $\mathcal{B}_\epsilon(\mathbf{e}) \cap \mathcal{O}$ for some $\epsilon > 0$.*

The concepts *Exit Set* and *Exit Point* are better in explained in Figure 3.4. In this scenario, two instances of *Exit Sets* are given. Out of the provided *Exit Sets*, the existence of *Exit Points* are investigated.

In Figure 3.4(a), using the *Slab* originating at $\mathbf{x}^A$, the *Exit Set* is found which contains elements $\mathcal{S}_1$ and $\mathcal{S}_2$. Figure 3.4(b) shows a close up of the *Exit Set* with the half-space $\{x \mid \mathbf{a}^\mathsf{T}\mathbf{x} \leq \mathbf{a}^\mathsf{T}\mathbf{e}\}$, with $\mathbf{a} = [0\ 1]^\mathsf{T}$. However, it can be seen that for the element $\mathcal{S}_1$ in *Exit*

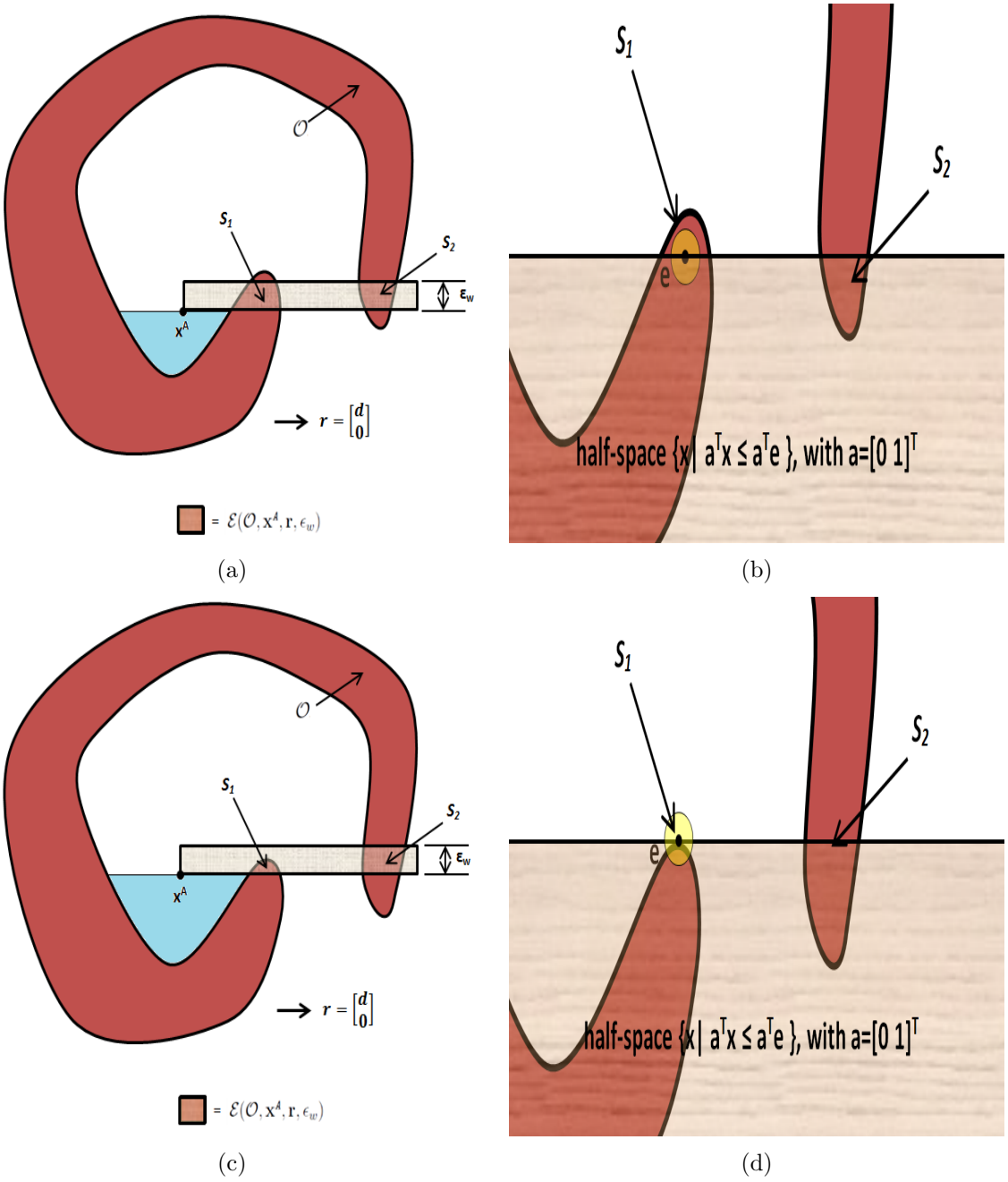


Figure 3.4: Illustrations of Exit Sets and Exit Points: (a) Exit Set 1. ; (b) Close up of Exit Set 1. ; (c) Exit Set 2. ; (d) Close up of Exit Set 2.

Set, the half-space $\{x \mid \mathbf{a}^\mathsf{T}\mathbf{x} \leq \mathbf{a}^\mathsf{T}\mathbf{e}\}$ does not contain the set $\mathcal{B}_\epsilon(\mathbf{e}) \cap \mathcal{O}$ for some $\epsilon > 0$. Hence, $\mathcal{S}_1$ is not an *Exit Point*.

Figure 3.4(c) shows another instance of an *Exit Set* found using the *Slab* originating at $\mathbf{x}^\mathsf{A}$. A close up of the *Exit Set* is given in Figure 3.4(d) with the half-space $\{x \mid \mathbf{a}^\mathsf{T}\mathbf{x} \leq \mathbf{a}^\mathsf{T}\mathbf{e}\}$. In this situation, the half-space $\{x \mid \mathbf{a}^\mathsf{T}\mathbf{x} \leq \mathbf{a}^\mathsf{T}\mathbf{e}\}$ contains the set $\mathcal{B}_\epsilon(\mathbf{e}) \cap \mathcal{O}$ for element $\mathcal{S}_1$ in *Exit Set*. Thus, $\mathcal{S}_1$ is an *Exit Point*.

Definition 4 (Upper Right Orthant) *Upper Right Orthant denoted $\mathcal{C}^R$ is the set of points of the upper-right quadrant of the coordinate system, i.e.,*

$$\mathcal{C}^R = \{\mathbf{x} \in \mathbb{R}^2 \mid \mathbf{x} = [x_1\ x_2]^\mathsf{T},\ x_1 \geq 0,\ x_2 \geq 0\}. \tag{6}$$

Definition 5 (Upper Left Orthant) *Upper Left Orthant denoted* $\mathcal{C}^L$ *is the set of points of the upper-left quadrant of the coordinate system, i.e.,*

$$\mathcal{C}^L = \{\mathbf{x} \in \mathbb{R}^2 \mid \mathbf{x} = [x_1 \ x_2]^\intercal, \ x_1 \leq 0, \ x_2 \geq 0\}. \tag{7}$$

Definition 6 (Translated Upper Right Orthant) *Translated Upper Right Orthant denoted* $\mathcal{C}^R(\mathbf{x})$*, is given by*

$$\mathcal{C}^R(\mathbf{x}) = \mathbf{x} + \mathcal{C}^R, \tag{8}$$

where $\mathbf{x}$ *is the translation introduced.*

Figure 3.5 illustrates the Translated Upper Right Orthant.

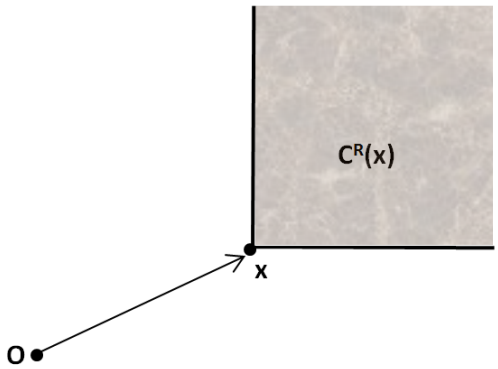


Figure 3.5: Translated Upper Right Orthant

Definition 7 (Translated Upper Left Orthant) *Translated Upper Left Orthant denoted* $\mathcal{C}^L(\mathbf{x})$*, is given by*

$$\mathcal{C}^L(\mathbf{x}) = \mathbf{x} + \mathcal{C}^L, \tag{9}$$

where $\mathbf{x}$ *is the translation introduced.*

The Translated Upper Left Orthant is shown in Figure 3.6.

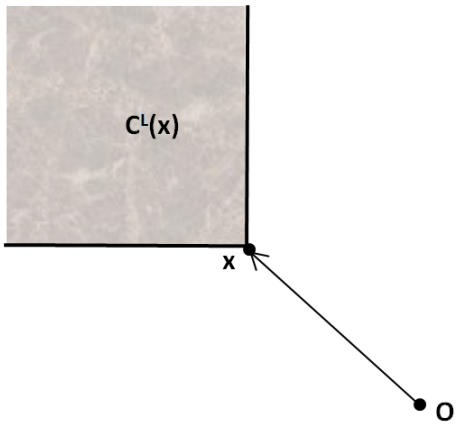


Figure 3.6: Translated Upper Left Orthant

Definition 8 (Irregular Points) *Given a point* $\mathbf{y} = [y_1\ y_2]^\mathsf{T}$*, let* $\mathcal{H}_\mathbf{y} = \{\mathbf{x} \mid [1\ 0]\mathbf{x} = y_1\}$*. The point* $\mathbf{y}$ *is called irregular, if for some* $\mathbf{v} \in \mathcal{I}_\mathbf{y} = \{\mathbf{x} \mid \mathcal{H}_\mathbf{y} \cap (\boldsymbol{bd}(\mathcal{O}_1) \cup \cdots \cup \boldsymbol{bd}(\mathcal{O}_p))\}$*, the hyperplane* $\{x \mid \mathbf{g}^\mathsf{T}\mathbf{x} = \mathbf{g}^\mathsf{T}\mathbf{v}\}$ *and* $\mathbf{y}$ *intersect only at* $\mathbf{y}$ *itself.*

Next we will look at the proposed algorithm. An explanation of the algorithm will also follow for improved clarity.

3.1.3 LoS by Water-Filling

This section will present the actual steps of the NLoS algorithm, which will be named as *LoS by Water-Filling*. The previously described notations and definitions will be used wherever needed.
The algorithm for LoS setup by Water-Filling and G-drops can be summarized as follows:

Algorithm: LoS by Water-Filling

---

1. Let the iteration index $k = 0$, $\mathbf{x}_k^\mathtt{A}$ denote the current (non irregular) position of the agent, ϵ_w denote the water-filling level in one chance, $\mathbf{r} = [1\ 0]^\mathsf{T}$ s_h denote the horizontal step length of the agent in one chance, and s_v denoted the vertical step length of the agent in one chance.

2. If LoS is established between $\mathbf{x}_k^\mathtt{A}$ and $\mathbf{x}^\mathtt{D}$, Stop. Otherwise to go to Step 3.

3. If $\mathbf{x}_k^\mathtt{A} \in \mathbf{bd}(\mathcal{O}_{i^\star})$ for some $i^\star \in \{1, 2, \ldots, p\}$, go to Step 4. Otherwise, make a vertical drop, i.e., set
$$\mathbf{x}_{k+1}^\mathtt{A} := \mathbf{x}_k^\mathtt{A} + s_v\mathbf{g}, \tag{10}$$
set $k := k + 1$ and go to Step 2.

4. Compute $\mathcal{S}_k$ from $\mathcal{S}_k = \mathcal{B}_\epsilon(\mathbf{x}_k^\mathtt{A}) \cap \mathcal{O}_{i^\star}$. If the half-space $\{x \mid -\mathbf{g}^\mathsf{T}\mathbf{x} \leq -\mathbf{g}^\mathsf{T}\mathbf{x}_k^\mathtt{A}\}$, contains the set $\mathcal{S}_k$ go to Step 9. Otherwise, go to Step 6

5. If $\mathbf{a}^\mathsf{T}\mathbf{x}_k^\mathtt{A} \geq \mathbf{a}^\mathsf{T}\mathbf{x}_{k-1}^\mathtt{A}$, with $\mathbf{a} = [1\ 0]^\mathsf{T}$, make a right horizontal step, i.e., set
$$\mathbf{x}_{k+1}^\mathtt{A} := \mathbf{x}_k^\mathtt{A} + s_h[1\ 0]^\mathsf{T} \tag{11}$$
and $k := k + 1$. Otherwise, make a left horizontal step, i.e., set
$$\mathbf{x}_{k+1}^\mathtt{A} := \mathbf{x}_k^\mathtt{A} - s_h[1\ 0]^\mathsf{T} \tag{12}$$
and $k := k + 1$. Go to Step 3.

6. If $\mathcal{S}_k \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_k^\mathtt{A}) = \emptyset$ and $\mathcal{S}_k \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_k^\mathtt{A}) \neq \emptyset$, make a right horizontal step, i.e., set
$$\mathbf{y}_k^\mathtt{A} := \mathbf{x}_k^\mathtt{A} + s_h[1\ 0]^\mathsf{T}, \tag{13}$$
set $\mathbf{x}_{k+1}^\mathtt{A}$ to the projection of $\mathbf{y}_k^\mathtt{A}$ onto the obstacle $\mathcal{O}_{i^\star}$ along $\mathbf{g}$, $k := k + 1$, and compute $\mathcal{S}_k$ from $\mathcal{S}_k = \mathcal{B}_\epsilon(\mathbf{x}_k^\mathtt{A}) \cap \mathcal{O}_{i^\star}$, and go to Step 7. Otherwise, go to Step 8.

7. If $\big(\mathcal{S}_{k-1} \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_{k-1}^\mathtt{A}) = \emptyset$ and $\mathcal{S}_k \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_k^\mathtt{A}) = \emptyset\big)$ or $\big(\mathcal{S}_{k-1} \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_{k-1}^\mathtt{A}) = \emptyset$ and $\mathcal{S}_k \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_k^\mathtt{A}) = \emptyset\big)$, go to Step 9. Otherwise go to Step 6.

8. If $\mathcal{S}_k \cap \mathcal{C}^{\mathtt{R}}(\mathbf{x}_k^{\mathtt{A}}) \neq \emptyset$ and $\mathcal{S}_k \cap \mathcal{C}^{\mathtt{L}}(\mathbf{x}_k^{\mathtt{A}}) = \emptyset$, make a left horizontal step, i.e., set

$$\mathbf{y}_k^{\mathtt{A}} := \mathbf{x}_k^{\mathtt{A}} - s_h[1\ 0]^{\mathsf{T}}, \tag{14}$$

set $\mathbf{x}_{k+1}^{\mathtt{A}}$ to the projection of $\mathbf{y}_k^{\mathtt{A}}$ onto the obstacle $\mathcal{O}_{i^\star}$ along $\mathbf{g}$, $k := k+1$, and compute $\mathcal{S}_k$ from $\mathcal{S}_k = \mathcal{B}_\epsilon(\mathbf{x}_k^{\mathtt{A}}) \cap \mathcal{O}_{i^\star}$, and go to Step 7. Otherwise go to Step 9.

9. In the directions $\mathbf{r}$ and $-\mathbf{r}$, check for an Exit-Point by using the Exit Set

$$\mathcal{E}(\mathcal{O}_{i^\star}, \mathbf{x}_k^{\mathtt{A}}, \mathbf{r}, \epsilon_w) = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}.$$

If an Exit-Point exists, set $\mathbf{x}_{k+1}^{\mathtt{A}}$ to the Exit-Point, $k := k+1$, and go to Step 5. Otherwise, go to Step 10.

10. Increase water level by a height ϵ_w. Set $\mathbf{x}_{k+1}^{\mathtt{A}}$ to an end point of the water line and set $k := k+1$. Go to Step 9.

Step 1 initializes the algorithm by setting the iteration index k to zero. Current position of the agent $\mathbf{x}_k^{\mathtt{A}}$, water-filling level in one chance ϵ_w, horizontal step length of the agent in one chance s_{h} and vertical step length of the agent in one chance s_{v} are also denoted in this step. The starting position is assumed to be a non irregular point to avoid certain technicalities. In simple terms, this checks whether the starting position is already an *Exit Point*. In reality, this assumption is almost always satisfied since it is highly unlikely that the starting position of the agent is an *Exit Point* on the obstacle.

Step 2 checks to see if line of sight can be established between the agent and the target destination. If LoS can be established, the algorithm stops and the LoS algorithm takes over. If LoS cannot be established, the algorithm moves to Step 3.

Step 3 checks to see if the current position of the agent $\mathbf{x}_k^{\mathtt{A}}$ is at a boundary of an obstacle. If so, the algorithm advances to Step 4. If the current position of the agent $\mathbf{x}_k^{\mathtt{A}}$ is not at an obstacle boundary, the agent is dropped a vertical step s_{v}, the iteration number k in incremented by one and the algorithm reverts back to Step 2.

Step 4 computes $\mathcal{S}_k$ which is the intersection of the obstacles and the ball centered around $\mathbf{x}_k^{\mathtt{A}}$. If the the half-space $\{x \mid -\mathbf{g}^{\mathsf{T}}\mathbf{x} \leq -\mathbf{g}^{\mathsf{T}}\mathbf{x}_k^{\mathtt{A}}\}$ contains the set $\mathcal{S}_k$, we move to Step 9. This essentially checks if the agent is already at an *Exit Point*. If this check fails, the algorithm skips to Step 6.

Step 5 is initiated when an *Exit Point* has been identified. This step checks the previous iteration agent position $\mathbf{x}_{k-1}^{\mathtt{A}}$ with the current agent position $\mathbf{x}_k^{\mathtt{A}}$ and directs the agent accordingly. If $\mathbf{a}^{\mathsf{T}}\mathbf{x}_k^{\mathtt{A}} \geq \mathbf{a}^{\mathsf{T}}\mathbf{x}_{k-1}^{\mathtt{A}}$, with $\mathbf{a} = [1\ 0]^{\mathsf{T}}$, the agent's previous position $\mathbf{x}_{\mathbf{k-1}}^{\mathtt{A}}$ was to the left of the current position $\mathbf{x}_{\mathbf{k}}^{\mathtt{A}}$. In other words, the agent has moved to its right in the previous iteration. Thus, it will be moved a s_{h} step to the right in this iteration as well. The same process follows if $\mathbf{a}^{\mathsf{T}}\mathbf{x}_k^{\mathtt{A}} \leq \mathbf{a}^{\mathsf{T}}\mathbf{x}_{k-1}^{\mathtt{A}}$, with $\mathbf{a} = [1\ 0]^{\mathsf{T}}$ as well, but the agent will be moved a s_{h} step to the left. After the movement, k is incremented by 1 and the algorithm restores to Step 3.

Step 6 checks to see if the $\mathcal{S}_k$ intersects with the Translated Upper Left Orthant and Translated Upper Right Orthant. If the condition ($\mathcal{S}_k \cap \mathcal{C}^{\mathtt{R}}(\mathbf{x}_k^{\mathtt{A}}) = \emptyset$ and $\mathcal{S}_k \cap \mathcal{C}^{\mathtt{L}}(\mathbf{x}_k^{\mathtt{A}}) \neq \emptyset$) is true, the obstacle is to the left of the agent position $\mathbf{x}_{\mathbf{k}}^{\mathtt{A}}$. In that case, the agent is moved to the right along the boundary of the obstacle $\mathcal{O}_{i^\star}$ in a direction of $\mathbf{y}_k^{\mathtt{A}}$ projected along $\mathbf{g}$. The iteration number k in increased by one, $\mathcal{S}_k$ is calculated at new current position and the algorithm moves to Step 7. If this condition is false, we skip to Step 8.

Step 7 essentially checks the agent's current position $\mathbf{x}_\mathbf{k}^\mathtt{A}$ is at a valley along the obstacle boundary. This is done by checking if the condition $\left(\mathcal{S}_{k-1} \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_{k-1}^\mathtt{A}) = \emptyset \text{ and } \mathcal{S}_k \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_k^\mathtt{A}) = \emptyset\right)$ or $\left(\mathcal{S}_{k-1} \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_{k-1}^\mathtt{A}) = \emptyset \text{ and } \mathcal{S}_k \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_k^\mathtt{A}) = \emptyset\right)$ is true. If either of the above statements are true, the agent's current position $\mathbf{x}_\mathbf{k}^\mathtt{A}$ is at a valley on the obstacle boundary. In that case, the algorithm skips to Step 9. If the agent is not at a valley position, we go back to Step 6.

Step 8 is very similar to Step 6. However, instead of checking if the obstacle is to the left of the agent's position $\mathbf{x}_\mathbf{k}^\mathtt{A}$, it checks the condition $(\mathcal{S}_k \cap \mathcal{C}^\mathtt{R}(\mathbf{x}_k^\mathtt{A}) \neq \emptyset \text{ and } \mathcal{S}_k \cap \mathcal{C}^\mathtt{L}(\mathbf{x}_k^\mathtt{A}) = \emptyset)$ to check the right of the agent's position. If this condition is true, the agent is moved to the left along the boundary of the obstacle $\mathcal{O}_{i^\star}$ in a direction of $\mathbf{y}_k^\mathtt{A}$ projected along $\mathbf{g}$. The iteration number k in increased by one, $\mathcal{S}_k$ is calculated at new current position and the algorithm moves to Step 7. In the case that the above condition is false, the algorithm goes to Step 9.

Step 9 checks for an *Exit Point* using the *Exit Set*

$$\mathcal{E}(\mathcal{O}_{i^\star}, \mathbf{x}_k^\mathtt{A}, \mathbf{r}, \epsilon_w) = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$$

in the directions $\mathbf{r}$ and $-\mathbf{r}$. If an *Exit Point* is found, the next position of the agent $\mathbf{x}_{k+1}^\mathtt{A}$ is set to the *Exit Point*. The iteration number k is incremented by 1 and we revert back to Step 5. If an *Exit Point* is not found, we advance to Step 10.

In Step 10, the water level is increased by a height ϵ_w and the next position of the agent $\mathbf{x}_{k+1}^\mathtt{A}$ is set to an end point on the water line. The iteration number k is incremented by 1 and the algorithm goes back to Step 9.
A flowchart depicting the condensed steps of the *LoS by Water-Filling* algorithm is given below in Figure 3.7.

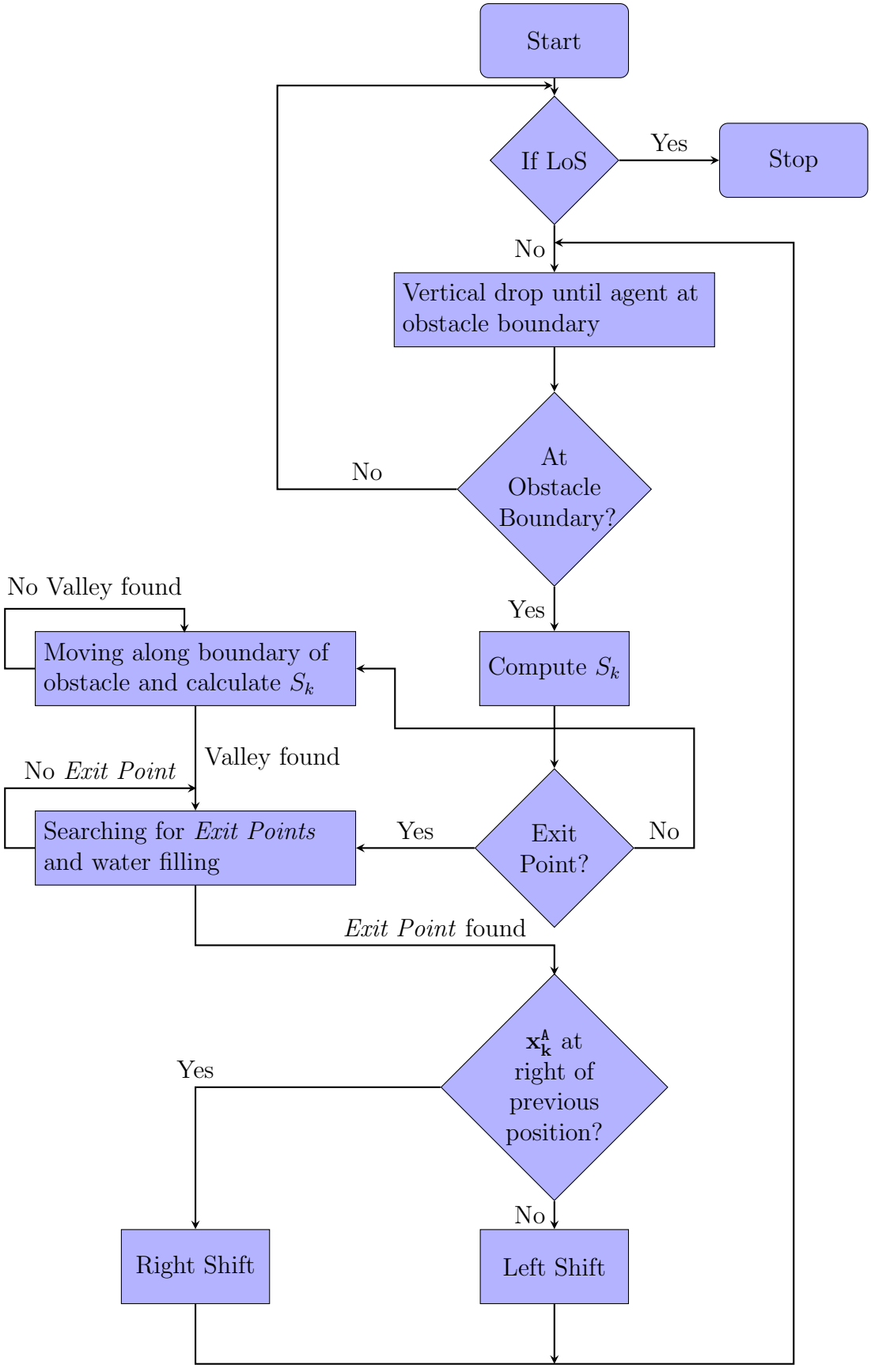


Figure 3.7: Flowchart of LoS by Water-Filling algorithm

3.2 Summary

In this section, a novel pathfinding algorithm named *LoS by Water-Filling* was proposed that can navigate agents, from a start position to a target destination, in the presence of obstacles, using local knowledge. The key idea and the reasoning behind the proposed solution was also discussed. The notations, definitions as well as the steps of the algorithm were explained in detail. *LoS by Water-Filling* is the NLoS algorithm that navigates the agents to a LoS position when obstacles are present.

In the subsequent chapter, we explore the candidates for the LoS algorithm. It will detail the inner workings and the mechanics of each of the algorithms. This will essentially serve as the foundation for selecting one of the algorithms for the LoS approach.

Chapter 4

LoS ALGORITHMS

This chapter deals with instances where *line of sight (LoS)* can be established between the agent and the target destination. Four candidates for the LoS problem are presented. All of the submitted candidates have their roots in the classic PSO algorithm. It should also be noted that the algorithms explained in this chapter are from existing literature. First, we will look into the concept of the *Constriction Coefficient* χ.

4.1 Constriction Coefficient

Each of the LoS algorithms have subtle differences when compared with each other but the one commonality they share is the use of the *Constriction Coefficient* χ. When the PSO algorithm is run without placing any velocity restraint, the velocities tend to increase to unacceptable levels within a few iterations. This, in turn can be harmful to the overall performance of the algorithm since it will cause the particles not to converge, even after the most optimum position has been found [12]. Hence the *Constriction Coefficient* χ is needed to mitigate the effect of these wayward velocities as the number of iterations increase.

The *Constriction Coefficient* χ proposed in [14] is applied as follows:

$$V_{i+1} = \chi[V_i + U(0,\phi_1)(p_i - x_i) + U(0,\phi_2)(p_g - x_i)] \ ^1 \tag{15}$$

$$x_{i+1} = x_i + V_{i+1}, \tag{16}$$

where $\phi = \phi_1 + \phi_2 > 4$ and

$$\chi = \frac{2}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \tag{17}$$

In this method ϕ is normally set to 4.1 where $\phi_1 = \phi_2$ and the *Constriction Coefficient* χ is calculated to be approximately 0.7298. As a result of χ, the previous velocity of the particle is multiplied by the constant 0.7298 and while the two (p_i-x_i) terms are multiplied by a random number that has a maximum value of 1.49618 [12]. The *Constriction Coefficient* χ is added to the PSO paradigm to ensure convergence of the algorithms.

The subsequent sections will detail out each of the LoS algorithms and their operating mechanisms. Firstly, a simplified deterministic PSO model without particle interaction

[1] U(0,ϕ_i) depicts a vector of randomly generated numbers uniformly distributed in [0,ϕ_i] at each iteration for each particle.

and constant springs will be considered and then components such as particle interaction and random springs are gradually introduced. It should be noted that without particle interaction optimization does not occur. In order to achieve optimization, interactions need to be incorporated into the model so that knowledge regarding a better position found by one particle can be communicated across the swarm [15]. The analysis of the different approaches form the basis for the selection of a LoS algorithm from the presented candidates.

4.2 PSO Version 1 without Particle Interaction and Constant Springs

This version does not take into account particle interaction and has a fixed spring constant. It essentially serves as a simplified PSO model that provides a better understanding of its inner workings. The model can be summarized as the following dynamic system [15]

$$\begin{aligned} v(t+1) &= v(t) + \phi[p - x(t)] \\ x(t+1) &= x(t) + v(t) + \phi[p - x(t)]. \end{aligned} \tag{18}$$

Since there is no particle interaction, the attractor term $(p_i - x(t))$ is fixed to $(p - x(t))$ and the spring constant is set to ϕ.

After applying velocity constriction the $v(t+1)$ is scaled by $\chi < 1$:

$$\begin{aligned} v(t+1) &= \chi\{v(t) + \phi[p - x(t)]\} \\ x(t+1) &= x(t) + \chi\{v(t) + \phi[p - x(t)]\} \end{aligned} \tag{19}$$

The given below convergence proof for this system directly follows what has already been discussed in [14] and [15].
The system can be rewritten as

$$\begin{aligned} v(t+1) &= \chi\{v(t) + \phi[y(t)]\} \\ y(t+1) &= y(t) + \chi\{-v(t) - \phi[y(t)]\}, \end{aligned} \tag{20}$$

or in matrix form as

$$P(t+1) = MP(t), \tag{21}$$

where $y(t) = p - x(t)$, $P(t) = [v(t), y(t)]^T$ and M is the 2 x 2 transformation matrix defined by the matrix equation

$$\begin{bmatrix} v(t+1) \\ y(t+1) \end{bmatrix} = \begin{bmatrix} \chi & \chi\phi \\ -\chi & 1 - \chi\phi \end{bmatrix} \begin{bmatrix} v(t) \\ y(t) \end{bmatrix}. \tag{22}$$

M is diagonalized by the similarity transform A.

$$AMA^{-1} = L = \begin{bmatrix} e_1 & 0 \\ 0 & e_2 \end{bmatrix} \tag{23}$$

Remark 1 *The convergence conditions for χ and ϕ are obtained by noting that $\|P(t)\|$ increase as $\|M^t P(0)\| = \|L^t A P(0)\|$ where $\|.\|$ is, for example, the Euclidean norm. It is shown in [14] that the eigenvalues $e_{1,2}$ are complex and of modulus $\sqrt{\chi}$ for $\phi > 4$, with χ given by*

$$\chi = \frac{2}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \tag{24}$$

Proof 1 *Convergence will follow if the constriction factor for a given spring constant is given by Eq. 24.*

We will be choosing *PSO Version 1* as the solution to the LoS problem defined above. This will be better explained in Chapter 5, *RESULTS and DISCUSSION*. It should also be noted that *PSO Version 1* is a simplified model of the classic PSO algorithm. In the subsequent sections, we incorporate stochasticity and particle interaction into this simplified model.

4.3 PSO Version 2 without Particle Interaction and Random Springs

The version explained in this section is without particle interaction but with random springs. When random springs are used, ξ_i is a randomly generated number at each iteration for each particle uniformly distributed in [0,1].
The system can be summarized as follows:

$$\begin{aligned} v(t+1) &= \chi\{v(t) + \phi\xi_i[p - x(t)]\} \\ x(t+1) &= x(t) + \chi\{v(t) + \phi\xi_i[p - x(t)]\} \end{aligned} \tag{25}$$

When the springs are randomized, we reintroduce stochasticity to the model. The effect of this randomness on the system is significant [14]. However, exploring the effect of randomness on the PSO paradigm is beyond the scope of this thesis and will not be discussed further. In the next section, we encompass particle interaction into the model.

4.4 PSO Version 3 with Particle Interaction and Constant Springs

The version given here incorporates particle interaction into the simplified model described in PSO Version 1. Since particle interaction is available in this model, it is capable of achieving optimization. However, the springs $\xi_{1,2}$ are kept at a constant value. A succinct representation of PSO Version 3 is as follows:

$$\begin{aligned} v(t+1) &= \chi\{v(t) + \phi_1\xi_1[p_i - x(t)] + \phi_2\xi_2[p_g - x(t)]\} \\ x(t+1) &= x(t) + \chi\{v(t) + \phi_1\xi_1[p_i - x(t)] + \phi_2\xi_2[p_g - x(t)]\} \end{aligned} \tag{26}$$

Personal influence and global influence which are two of the most important aspects in the PSO paradigm are included in this version. In other words, particles are affected by their personal experience as well as the experience of the swarm as a whole. The following section will detail the complete PSO model with particle interaction and stochasticity.

4.5 PSO Version 4 with Particle Interaction and Random Springs

The PSO Version 4 described here embraces the concepts of particle interaction and random springs. Essentially, this can be considered as the classic PSO algorithm with the constriction coefficient χ added to ensure convergence.
The velocity and position update rules for the system is

$$\begin{aligned} v(t+1) &= \chi\{v(t) + \phi_1\xi_1[p_i - x(t)] + \phi_2\xi_2[p_g - x(t)]\} \\ x(t+1) &= x(t) + \chi\{v(t) + \phi_1\xi_1[p_i - x(t)] + \phi_2\xi_2[p_g - x(t)]\}, \end{aligned} \tag{27}$$

where $\phi_{1,2}$ are commonly set to 2.05 and $\xi_{1,2}$ are randomly generated numbers uniformly distributed in [0,1].

4.6 Summary

Effectively, four PSO versions that are based on existing literature were discussed as potential candidates for the LoS problem. The discussion started with a simplified PSO model without particle interaction and stochasticity. As the chapter progressed, particle interaction and stochasticity was slowly reintroduced into the simplified model. Nevertheless, *PSO Version 1 without Particle Interaction and Constant Springs* was selected for the LoS approach due to its simplicity and guaranteed convergence. This will be explained in more detail in Chapter 5, *RESULTS and DISCUSSION*. *PSO Versions 2-4* will not be discussed further in this thesis.

In the next chapter, we will explore the simulation results of a modified version of the *LoS by Water-Filling* algorithm that captures its essence as well as the LoS algorithms. The simulation results for the LoS algorithms will provide the reasoning for selecting a candidate for the LoS situation. In addition, the steps that needs to be taken when implementing the *LoS by Water-Filling* algorithm in a real video game environment will be explored.

Chapter 5

RESULTS and DISCUSSION

This chapter is dedicated to the results and the discussion relating to the proposed solution. A brief description about the employed testing environment will also be given. The chapter will also delve into the challenges and issues that needs to be addressed when implementing the proposed solution in a real gaming environment.

First, we will provide a succinct explanation regarding the testing environment. Excerpts and screenshots relating to the gaming environment has been given wherever possible.

5.1 Testing Environment

The testing of the proposed algorithm was carried out using *GameMaker: Studio(GMS)* which is a proprietary game creation system created by Mark Overmars. GameMaker: Studio allows the creation of game maps, which are referred to as rooms within GMS, sprites for the in-game agents and scripts which can be used to model the behavior of these agents. All of the algorithms were coded using the native GMS scripting language, GameMaker Language (GML) which is syntaxically similar to JavaScript. Figure 5.1 shows an excerpt of the code that was used.

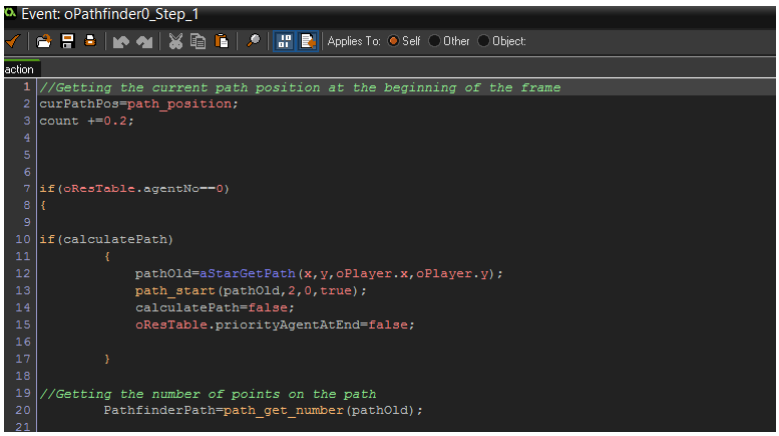


Figure 5.1: GameMaker Language

Generally, information relating to video game geometry are stored in a structure referred to as a map [16]. The following section provides details regarding the map structures that were used for the simulations.

5.1.1 Employed Video Game Map

The size of the used maps is 768 x 512 which consists of 393,216 pixels. A 24 x 16 grid was then constructed with a cell size of 32 x 32 to be superimposed over the game map. The topology of the map contains open cells that can be traversed as well as untraversable closed cells that serve as obstacles. A sample GMS map is shown below in Figure 5.2.

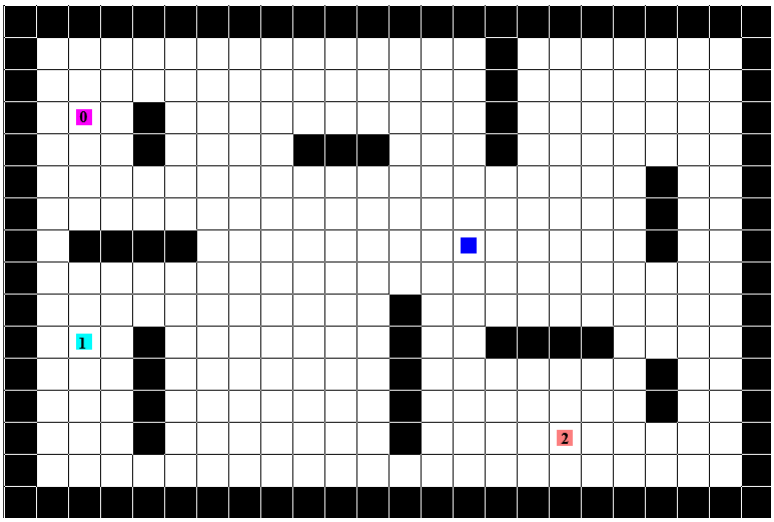


Figure 5.2: GameMaker: Studio sample map

The White cells are traversable areas while the Black cells play the role of obstacles. These maze-like environments play the role of game maps for the upcoming simulation results. Movement criteria which is another important aspect of video game pathfinding will be discussed next.

5.1.2 Movement Criteria

Movement criteria of the agents in the video game plays a vital role when it comes to agent pathfinding. Certain game environments allow only four degrees of freedom of movement while others may support up to 8 or more. In this thesis, the constructed grid in the simulation environment allows 8 directions of movement. Figure 5.3 illustrates the allowed directions in the grid.

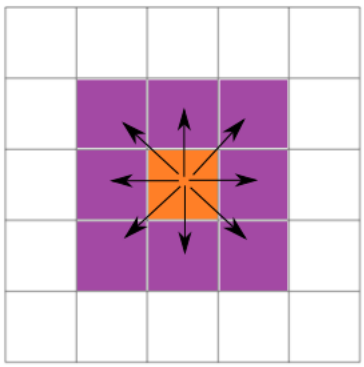


Figure 5.3: Allowed movements in the grid

If the Orange cell is the starting position of an agent, the Purple cells represent the cells that the agent can move into next. The simulation environment of this thesis employs 2D Square (Octile) grids with each center cell bordering eight neighbour cells.

The movement criteria given here imposes certain limitations when implementing the *LoS by Water-Filling* algorithm in a real video gaming environment. In the *LoS by Water-Filling* algorithm, there are no restrictions on the direction in which the agent can travel. For instance, if the agent needs to travel on a direction of 30 degrees from North, it would be difficult to implement in GameMaker: Studio due to allowed directions being multiples of 45 degrees from North. Due to this reason, we provide a simulation for a modified version which captures the essence of the *LoS by Water-Filling* algorithm.

The subsequent section presents the results that were gathered from the simulations. Based on the results and observations, a discussion will follow regarding the proposed solution. Moreover, the steps that need to be taken when the solution is applied to a real gaming environment will also be explored.

5.2 Results and Discussion

This section deals with the results of the simulations of the modified implementation of the NLoS algorithm, *LoS by Water-Filling* and the LoS algorithms. The results will be in the form of each agent's distance to target against the number of iterations of the algorithm. The measured distance is the Euclidean distance or the L^2 norm and is given in pixels.

5.2.1 LoS by Water-Filling Algorithm Applied in a Real Video Gaming Environment

In this section, we explore how the *LoS by Water-Filling* algorithm can be applied in a real video gaming environment. The game map that will be used for the explanation is given below in Figure 5.4.

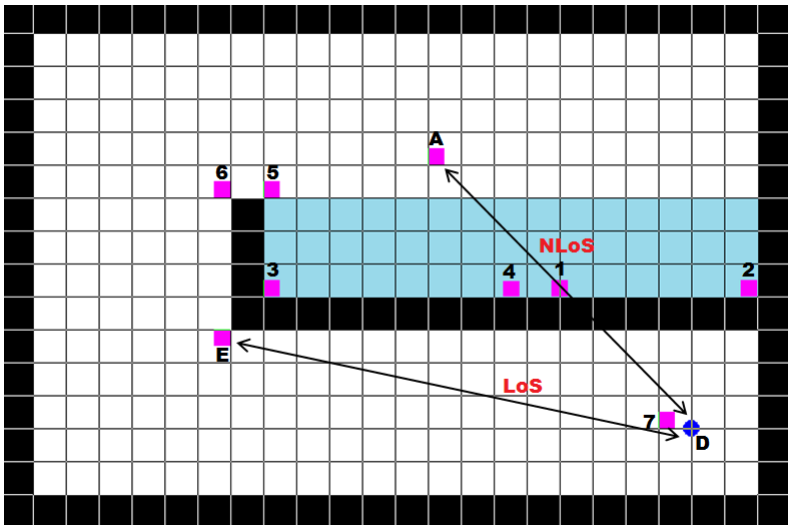


Figure 5.4: LoS by Water-Filling implemented in GMS

Position `A` denotes the start point of the agent and position `D` is the destination. Position `E` is the first line of sight position that the agent will encounter while position *6*

serves as an exit point. It is evident from the start that the agent cannot establish line of sight to the destination `D` from its start point `A`. Thus, it will essentially employ the *LoS by Water-Filling* algorithm to navigate itself to the line of sight position `E`. Next, let's run through the steps of the algorithm for this game map.

Since the agent cannot establish line of sight between `A` and `D`, it will run Step 3 of the algorithm until it reaches the boundary of the obstacle which is given by point *1*. Once at point *1*, the agent follows Steps 6 and 8 of the algorithm to move along the boundary of the obstacle while looking for an *Exit Point*. This is evident in the simulation where the agent moves to points *2* and *3* looking for an *Exit Point*. Since an *Exit Point* cannot be found along the boundary, the agent navigates to position *4*, the middle of the encountered obstacle. This is in accordance with Step 7 of the algorithm where the algorithm checks if the agent is at a valley on the boundary of the obstacle. Once at position *4*, the agent employs Step 10 of the algorithm to increase the water level and goes back to Step 9 to restart the search for an *Exit Point*. Using a combination of Steps 9 and 10, the agent finds the *Exit Point 6*. The agent ultimately reaches the *Exit Point* at position *6* by first reaching the edge at point *5*. At point *5*, the agent employs Step 5 of the algorithm which forces the agent to move to the left of its current position.

Once at exit point *6*, the agent invokes Step 3 of the algorithm to get to the line of sight position `E` by using a series of vertical drops. When the agent reaches line of sight position `E`, Step 2 comes into play and the *LoS by Water-Filling* algorithm ceases. At this point, line of sight can be established and the LoS algorithm is activated. Now employing the LoS algorithm, the agent navigates itself to position *7* and ultimately to the destination point `D`.

In the subsequent section, the steps described above will be applied to a real video game environment. Figure 5.5 represents a more complicated game map that deals with the NLoS situation due to the prevalence of obstacles. It should be noted that the agents utilize the same steps described here to navigate the obstacles and ultimately converge on the destination.

5.2.2 LoS by Water-Filling NLoS Algorithm

In this section we look at the quintessence of the *LoS by Water-Filling* algorithm described above applied to a more complicated video game map. Due to the previously described limitations of the simulation environment, a modified version of *LoS by Water-Filling* algorithm that captures its essence is implemented here. As previously stated, the NLoS agent pathfinding is complicated due to the presence of obstacles. This is more clearly evident in the game map that is used for the simulation which is given in Figure 5.5.

The Purple squares represent in-game agents while the Blue circle, named `D`, depicts their destination. In addition, White cells represent traversable area while Black cells represents obstacles. The number of agents were kept at 10. It can be seen that none of the agents can establish line of sight to the target position `D` from their starting positions. This would mean that they would essentially have to employ the steps given by the NLoS algorithm, *LoS by Water-Filling* to reach the LoS position `E` in the map. The results of the *LoS by Water-Filling* algorithm simulation can be found in Figure 5.6.

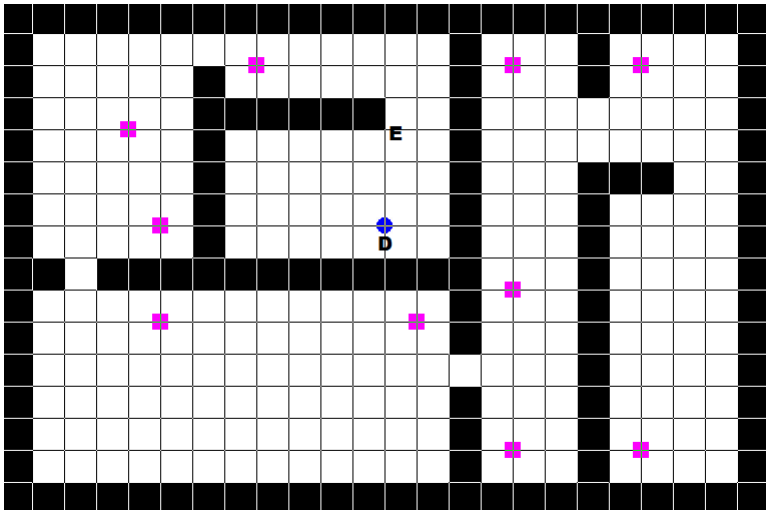


Figure 5.5: Game map for LoS by Water-Filling algorithm simulation

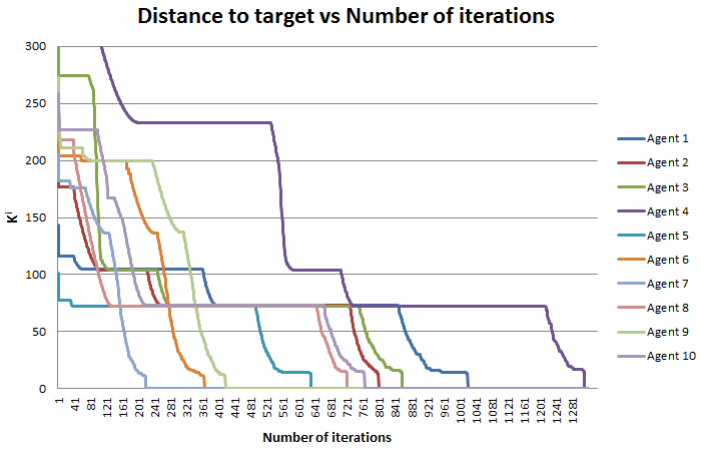


Figure 5.6: NLoS algorithm simulation result

The Figure 5.6 plots the K^i against the number of iterations of the algorithm. K^i is calculated with Eq. 28,

$$K^i = \min \|x_i - x_d\|, i \in \{1, 2, \ldots, j\} \tag{28}$$

where x_i is the current position of the agent in this iteration and x_d is the target position.

Since none of the agents can establish line of sight from their starting positions to the destination position `D`, the NLoS algorithm *LoS by Water-Filling* takes over. The objective of this algorithm is to navigate the agents through the obstacles to the LoS position `E`. At point `E`, the agents can establish line of sight with the destination position `D` and the LoS algorithm takes over. It can be seen in Figure 5.6 that by using the proposed solution all of the agents, irrespective of their starting positions have converged on the target position `D`.

In the next section, the simulation results for the presented LoS algorithms are examined. This analysis will aid in selecting an algorithm for the LoS approach.

5.2.3 LoS Algorithms

Now we present the simulation results for the LoS algorithms presented in Chapter 4. In order to achieve better comparable results, all of the algorithms were simulated with factors such as the game map and the number of agents kept constant. The game map that was used for this section is shown in Figure 5.7.

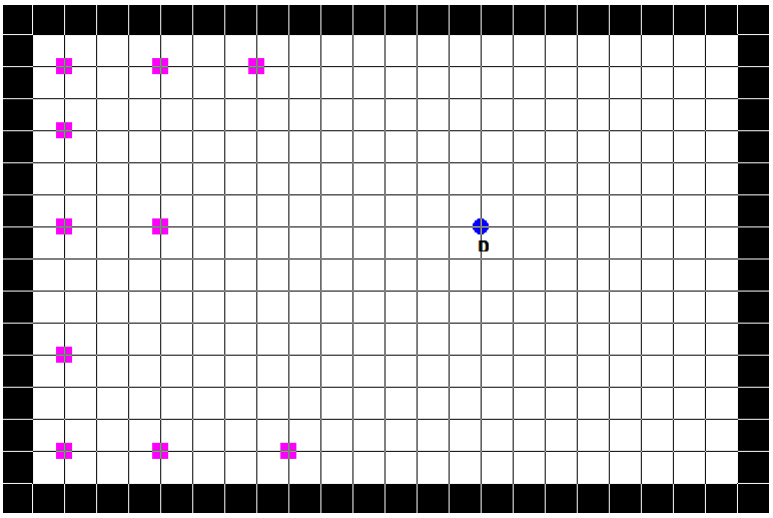


Figure 5.7: Game map for LoS algorithm simulations

The Purple squares represent in-game agents while the Blue circle, named `D`, depicts their destination. As previously mentioned in the *Testing Environment* section, White cells represent traversable area while Black cells represents obstacles. The number of agents were kept at 10 across all versions of the algorithms.

Since the LoS algorithms take over when there are no obstacles present, this can be essentially seen as a convergence analysis. Table 5.1 provides the number of iterations that are needed for each of the LoS algorithms to converge. This table depicts information for one specific agent across the LoS algorithms, but it should be noted that the results are similar for other agents also.

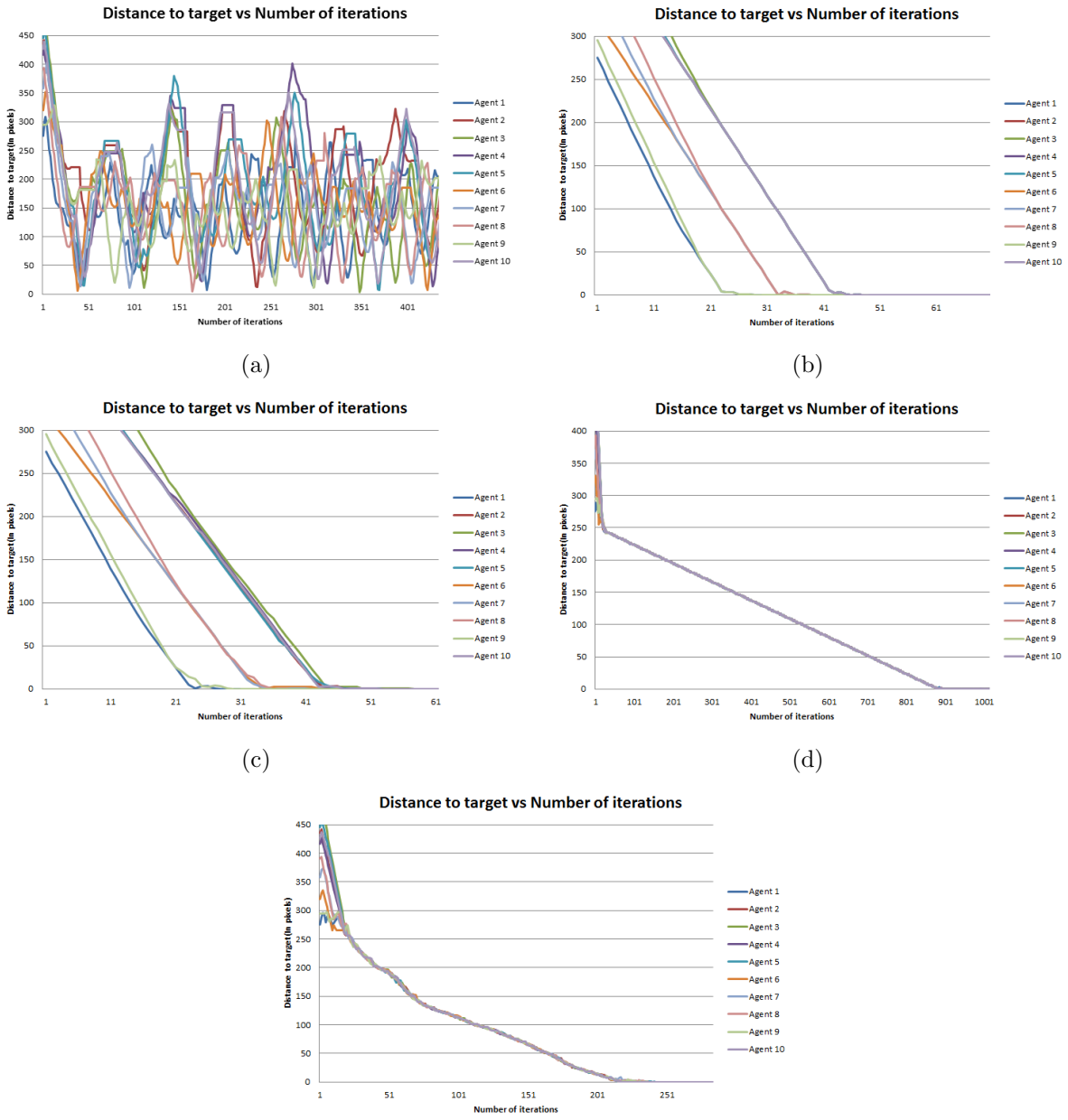


Figure 5.8: LoS algorithms simulation results: (a) Classic PSO. ; (b) PSO Version 1. ; (c) PSO Version 2. ; (d) PSO Version 3. ; (e) PSO Version 4.

Table 5.1 demonstrates the number of iterations against the distance to target for each of the LoS algorithms for one agent [1]. The definition of convergence, in this instance, would be when the agent's distance to the target has reached zero.

The classic PSO does not exhibit convergence since there is no mechanism to constrict the velocities as the number of iterations increase. This is evident in column 2 of Table 5.1 where the distance to target never reaches zero. *PSO Version 1* converges at around iteration number 46 while *PSO Version 2* converges around iteration 47. *PSO Version*

[1]The number of iterations needed for convergence given in Table 5.1 depend on the step size taken by the agent. Large step sizes will follow faster convergence. Here the maximum step size is limited to 15 pixels.

Number of Iterations	Distance to Target (In Pixels)				
	Classic PSO	PSO V1	PSO V2	PSO V3	PSO V4
1	4.45×10^2	4.45×10^2	4.45×10^2	4.45×10^2	4.45×10^2
10	3.69×10^2	3.33×10^2	3.33×10^2	3.75×10^2	3.76×10^2
20	2.47×10^2	2.26×10^2	2.26×10^2	2.50×10^2	2.59×10^2
30	1.61×10^2	1.26×10^2	1.26×10^2	2.43×10^2	2.29×10^2
40	7.80×10^1	2.60×10^1	3.40×10^1	2.41×10^2	2.03×10^2
41	6.40×10^1	1.60×10^1	2.40×10^1	2.41×10^2	2.01×10^2
42	5.10×10^1	6.00	1.40×10^1	2.40×10^2	2.00×10^2
44	2.47×10^1	3.00	5.00	2.39×10^2	2.00×10^2
45	1.45×10^1	1.00	2.00	2.40×10^2	1.99×10^2
46	1.45×10^1	0.00	1.00	2.39×10^2	1.98×10^2
47	1.32×10^1	0.00	0.00	2.33×10^2	1.96×10^2
100	1.03×10^2	0.00	0.00	2.23×10^2	1.12×10^2
150	3.48×10^2	0.00	0.00	2.10×10^2	6.64×10^1
200	2.04×10^2	0.00	0.00	1.95×10^2	1.39×10^1
210	2.68×10^2	0.00	0.00	1.92×10^2	7.07
211	2.68×10^2	0.00	0.00	1.92×10^2	7.07
242	1.95×10^2	0.00	0.00	1.83×10^2	1.41
243	2.03×10^2	0.00	0.00	1.83×10^2	0.00
300	9.29×10^1	0.00	0.00	1.67×10^2	0.00
400	3.03×10^2	0.00	0.00	1.38×10^2	0.00

Table 5.1: LoS algorithms: Number of iterations needed for convergence

4 achieves convergence at around iteration number 243. At around iteration number 400, where *PSO Versions 1,2 and 4* already have converged, *PSO Version 3* still has not converged. *PSO Version 3* will finally converge at around iteration number 893, which is not displayed in Table 5.1.

Thus, it can be seen that *PSO Version 1* would be the best solution for the LoS problem, mainly due to its simplicity and fast guaranteed convergence. Hence, for our proposed approach, *PSO Version 1* will take over when LoS can be established from the agent to the target position.

5.3 Summary

In this chapter, we explored the results for the modified implementation of the NLoS algorithm *LoS by Water-Filling* and the LoS algorithms. The results are given in the form of each agent's distance to target against the number of iterations of the algorithm. The unit of measurement for the simulations is the Euclidean distance or the L^2 norm and is given in pixels. The modified implementation of the NLoS algorithm *LoS by Water-Filling* achieved in directing all of the agents to the destination, irrespective of their starting position, in the presence of obstacles by employing local knowledge. Moreover, by analyzing the results of the LoS algorithm simulations, we selected *PSO Version 1 without Particle Interaction and Constant Springs* as the candidate for the LoS situation. This was due to its simplicity and fast guaranteed convergence.

By examining the results, the convergence of the proposed solution is *guaranteed*. The convergence proof follows the premise that *a stream of water is guaranteed to flow from the source to the destination, in the direction of gravity, irrespective of the starting position, given that the destination is at a lower height than the source and there exists a path between the source and the destination.*

In the next chapter, we summarize the findings and contributions of the thesis. It will also go into explaining potential paths of future research.

Chapter 6

CONCLUSIONS and FUTURE WORK

In this chapter a summarization of the thesis will be done highlighting the main contributions and obtained results. In addition, potential avenues for future work is also suggested.

6.1 Conclusions

The main aim of this thesis is to design a novel algorithm that can navigate agents from a starting position to a destination in the presence of obstacles using local knowledge. This was achieved by proposing a two-part based solution that can be implemented in a real gaming environment. We will now present a summary of the thesis detailing the main contributions and results.

Chapter 1 was dedicated to providing a brief introduction regarding the history of video games. The motivation behind this thesis, which is the need to create a novel algorithm that can operate using local knowledge, was also briefly explained. Finally, the aims and the outline was given to better explain the flow of the thesis.

Chapter 2 delved into a review of current video game pathfinding solutions as well as an explanation of the classic Particle Swarm Optimization (PSO) algorithm. The review of existing video game pathfinding algorithms included a condensed description regarding the A* and the Windowed Hierarchical A* (WHCA*) algorithms. The inner workings of A* and WHCA* were discussed along with a revelation on areas that can be improved. The main drawbacks of A* based approaches which is the need for global knowledge of the video game environment and having to compute and store the graph for the agents were emphasized. Finally, the classic PSO algorithm was also elaborated on to highlight its potential viability as video game pathfinding solution.

In Chapter 3, a novel video game pathfinding solution, *LoS by Water-Filling* algorithm was proposed that can navigate Non-Playable Characters (NPC) to a destination in the presence of obstacles using local knowledge. The proposed method is a two-part based solution. When no line of sight (NLoS) can be established, the NLoS algorithm *LoS by Water-Filling* navigates the agent to a position where LoS can be established to the target destination. Once line of sight (LoS) has been established, the LoS algorithm takes over. This solution is based on the principle that a stream of water will always flow in the direction of gravity when the orientation of the landscape allows it. Given that the destination is at a lower height than the starting position and there exists a path that

connects them, the algorithm will navigate the agent to the target position, irrespective of the agent's starting position. The main achievement of the *LoS by Water-Filling* is the ability to navigate the agents to a destination position in the presence of obstacles using local knowledge.

Chapter 4 detailed the line of sight (LoS) algorithms that will take over when the agent can establish line of sight with the target destination. Four candidates based on existing literature were suggested to solve the LoS situation. All of the submitted candidates have their roots in the classic PSO paradigm. Out of the submitted candidates, the *PSO Version 1 without Particle Interaction and Constant Springs* was selected as the LoS algorithm.

Chapter 5 presented the simulation results and followed it with a discussion. It also briefly described the testing environment GameMaker: Studio (GMS) that was used for the simulations. Firstly, the steps that are taken when applying the proposed solution *LoS by Water-Filling* to a real gaming environment were briefly examined. This mainly stems from the prevalent movement restrictions in the video game environment GameMaker: Studio (GMS). Due to this reason, a modified version of the proposed solution was implemented that captures the essence of the *LoS by Water-Filling* algorithm. The simulation for the *LoS by Water-Filling* algorithm consists of a complex map where none of the agents have line of sight to the target destination from their starting positions. Thus, they have to employ the *LoS by Water-Filling* algorithm to reach a line of sight position where the LoS algorithm can take over. The effectiveness of the proposed algorithm in a real gaming environment was displayed by showing that all agents converged on the target destination navigating through obstacles by employing local knowledge.

Chapter 5 then went onto provide and compare results for the five LoS algorithms. They are namely the *Classic PSO algorithm*, *PSO Version 1 without Particle Interaction and Constant Springs*, *PSO Version 2 without Particle Interaction and Random Springs*, *PSO Version 3 with Particle Interaction and Constant Springs* and *PSO Version 4 with Particle Interaction and Random Springs*. Out of the five presented candidates, *PSO Version 1* was selected as the LoS algorithm due to its simplicity and fast guaranteed convergence.

By analyzing the results obtained in Chapter 5, the convergence for the proposed solution is *guaranteed*. The convergence proof follows the premise that *a water stream is guaranteed to flow from the source to a destination, in the direction of gravity, irrespective of the starting position, given that the landscape is appropriately oriented and there exists a path that connects the source to the destination.*

6.2 Future work

There are many future avenues of potential research stemming from the presented approach. The proposed solution has many applications in robotics, especially pathfinding and maze navigation.

Although the proposed solution can effectively navigate agents to a destination in the presence of obstacles using local knowledge, it does not take collisions into account. The current implementation of the algorithm assumes the agents to be volumeless and massless and thus, having the ability to occupy the same space at the same time. Hence, if this were to be implemented on a real-world problem, a robust and efficient collision avoidance mechanism would be needed.

In addition, the proposed solution can further benefit from incorporating a social influence into the search mechanism. For instance, if one of the agents have successfully found a LoS position, it can broadcast this information to any agents in the immediate vicinity. The receiving agent can then use this information to improve its search efficiency. This element of cooperation can better improve the overall performance of the algorithm by aiding faster convergence.

A similar principle can be applied to divide the video game map into non-intersecting search areas which can be assigned to different agents. Initially, the agents search their respective regions independent of each other but can relay information to neighbouring agents, if one of them discovers a LoS position. This approach would definitely be beneficial when a large video game map needs to be solved.

There are numerous interesting applications that can benefit from the proposed solution as well as further room for improvement. One is only limited by one's own imagination.

Bibliography

[1] S. Kent, *The Ultimate History of Video Games: From Pong to Pokemon—The Story Behind the Craze That Touched Our Lives and Changed the World*, 1st ed. Three Rivers Press, 2001.

[2] K. Wang and A. Botea, "Fast and memory-efficient multi-agent pathfinding," in *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, Sydney, Australia, September 2008, p. 380.

[3] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–128, January 2011.

[4] D. Silver, "Cooperative pathfinding," in *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*. Marina del Rey, California: Association for the Advancement of Artificial Intelligence, June 2005, pp. 117–122.

[5] A. Patel, "Introduction to a*," Stanford Theory Group, [Online]. Available: http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html, accessed February 12, 2016.

[6] N. Sturtevant and M. Buro, "Improving collaborative pathfinding using map abstraction," in *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference*. Marina del Rey, California: Association for the Advancement of Artificial Intelligence, June 2006, pp. 80–81.

[7] G. Mathew and G. Malarthy, "Direction based heuristic for pathfinding in video games," *Global Journal of Computer Science and Technology: Graphics & Vision*, vol. 15, pp. 1–2, 2015.

[8] W. Lee and R. Lawrence, "Fast grid-based path finding for video games," in *26th Canadian Conference on Artificial Intelligence*. Regina, SK, Canada: Springer Berlin Heidelberg, May 2013, pp. 100–101.

[9] Z. Bnaya and A. Felner, "Conflict-oriented windowed hierarchical cooperative a*," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, p. 2.

[10] J. Hagelbäck, "Hybrid pathfinding in starcraft," *IEEE Transactions on Computational Intelligence and AI in Games*, p. 1, March 2015.

[11] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*. IEEE, November 1995, pp. 1942–1948.

[12] J. Kennedy, R. Poli, and T.Blackwell, "Particle swarm optimization: An overview," *Swarm Intelligence*, vol. 1, pp. 33–57, June 2007.

[13] R. Eberhart and Y. Shi, "Particle swarm optimization: developments, applications and resources," in *Proceedings of the 2001 Congress on Evolutionary Computation*. IEEE, May 2001, pp. 81–83.

[14] M. Clerc and J. Kennedy, "The particle swarm - explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 58–72, February 2002.

[15] T. Blackwell, "Particle swarms and population diversity," *Soft Computing*, vol. 9, pp. 793–802, November 2005.

[16] R. Graham, H. McCabe, and S. Sheridan, "Pathfinding in computer games," *ITB Journal*, vol. 4, pp. 57–59, December 2003.