SOLVING THE TAKE DOWN AND BODY HIDING PROBLEMS

by

Jorge Emmanuel Morales Díaz

School of Computer Science
McGill University, Montreal

November, 2019

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Stealth games are a genre that challenges players to complete tasks while remaining unnoticed by enemies. To create this type of challenge, designers usually add movement mechanics to enemies. Players need to analyze these behaviours to find a path towards a goal while remaining hidden from the enemies' sensory systems, where vision is one of the most common features included. While movement is the main tool used by players to solve this challenge, modern games often provide players with the ability to silently eliminate enemies, such as putting them to sleep or dealing a lethal blow from behind. By adding this, challenges can acquire a puzzle-like nature in which the player must find not only a path to the goal but also the order in which the enemies must be eliminated in order to ensure the path remains hidden. This "take down" problem is frequently accompanied by a need to hide the enemy body in order to leave no evidence behind and avoid creating suspicion in the remaining characters. However, by adding all of these possibilities, the realm of possible scenarios becomes larger and analyzing its design to determine if it is suitable for players to solve becomes a more complex task. In this work, we present a version of a sample-based search algorithm to analyze a level and find paths that respect the stealth constraints. This approach allows for the addition of *take down* and *body hiding* mechanics as well as variations in the conditions imposed by them in dynamic scenarios. These conditions are analyzed and formalized in order to introduce a series of improvements that increase the success rate of the search algorithm. Afterwards, we present different methods used in games to introduce zones where a body can be hidden and how they affect the search. Our work shows that the take down problem can be incorporated into the search space with reasonable success, more affected by some design parameters than others. In addition, we show that the body hiding problem has the surprising impact of often improving the success rate, despite the additional steps involved. A full analysis of the stealth problem including these kinds of variations is important to understanding the difficulty of a game design and can be useful to designers.

Résumé

Les jeux furtifs sont un genre qui incite les joueurs à effectuer des tâches tout en restant inaperçus des ennemis. Pour créer ce type de défi, les concepteurs intègrent des mécanismes de déplacement dans les personnages du jeu. Les joueurs doivent donc, en utilisant leur vision, analyser ces comportements pour trouver leur chemin tout en restant cachés des systèmes sensoriels mis en place par les concepteurs. Bien que le mouvement soit le principal outil utilisé par les joueurs pour résoudre ce problème, les jeux modernes offrent souvent aux joueurs la possibilité d'éliminer en silence leurs ennemis, par exemple en les endormant ou en leur infligeant le coup fatal. En ajoutant cela, les défis acquièrent une dimension de type «puzzle» : le joueur doit trouver non seulement un chemin vers le but, mais également, l'ordre dans lequel les ennemis doivent être éliminés afin que son parcours reste indétectable. Ce problème de "Take down" s'accompagne souvent d'un besoin de cacher le corps de l'ennemi afin de ne laisser aucune trace derrière lui et d'éviter de créer des soupçons dans les personnages restants. Cependant, en ajoutant toutes ces possibilités, le nombre de scénarios possibles s'élargit et l'analyse de sa conception pour déterminer s'il convient aux joueurs devient une tâche plus complexe. Dans ce travail, nous présentons une version d'un algorithme de recherche basé sur des exemples pour analyser un niveau du jeu et vérifier que les chemins respectent les contraintes de furtivité. Cette approche permet d'ajouter des mécanismes de *"Take down"* et de *"Body hiding"*, ainsi que des variations des conditions qu'ils imposent dans des scénarios dynamiques. Ces conditions sont analysées et formalisées afin d'introduire une série d'améliorations qui augmentent le taux de réussite de l'algorithme de recherche. Ensuite, nous présentons différentes méthodes utilisées dans les jeux pour introduire des zones dans lesquelles un corps peut être caché et comment elles affectent la recherche. Nos travaux montrent que le problème de *"Take down"* peut être intégré à l'espace de recherche avec un succès raisonnable, davantage affecté par certains paramètres de conception que par d'autres. De plus, nous montrons que le problème de *"Body hiding"* a l'impact surprenant d'améliorer souvent le taux de réussite, malgré les étapes supplémentaires que cela implique. Une analyse complète du problème de furtivité incluant ces types de variations est importante pour comprendre la difficulté de la conception des jeux et, pour cette raison, peut être utile aux concepteurs.

Acknowledgements

First of all, to my supervisor Clark Verbrugge, you have my most sincere gratitude for all your support, patience and guidance since the very first day. This thesis would have never been possible without you.

To all my family, for always being supporting and cheering for me. To my father for giving me strength in the hardest times, my mother for being that light that guides me, Albina for all the kindness and love. This is all yours, thanks for showing me how to make my dreams true.

To all the people that walked with me during this journey. Angel, thanks for your help and words, Gabriela for always having time to hear me, Nash for all your care, Cary for showing me what I am capable of, for challenging me and for all those amazing coins, Daniela for all the laughs we shared. Thank you all for making winters less harsh.

I would also like to recognize the CONACyT for the help provided to carry out my studies.

Special thanks to Mandy for always putting a smile in my heart. And finally, a special thanks to Margarita and Daniel, for never leaving me alone, for all your love, and for showing me the purest love there is. You are always in my heart and my heart will always be with you.

Contents

List of Figures

List of Algorithms

List of Symbols

n ..... Number of dimensions.
S ..... Coordinate space.
A ..... Starting point.
B ..... Ending point.
x ..... Any state in S.
P ..... Solution path..
m ..... Maximum number of nodes in the RRT.
s ..... State in the space.
O ..... Set of static obstacles in the coordinate space.
o_i ..... Point in the coordinate space that represents an obstacle.
S_{free} ..... Free/walkable space in the state space..
T ..... Edges or connections of nodes.
P ..... Solution path for the stealth problem.
E ..... Set of enemies.
O_s ..... Set of obstacles in a stealth problem.
F_e ..... Field of View associated to the enemy e.
t ..... Time.
k ..... Solution length.
O_{ko} ..... Obstacles in a take down problem.
$E_a(t)$ ..... Enemies alive at time t.
$E_{ko}(t)$ ..... Enemies taken down at time t.
r ..... Knockout radius.
P_{ko} ..... Solution for a take down scenario.
$e_{bh}(t)$ ..... Enemy position at time t in a body hiding problem.

$t_{ko}(e)$ ..... Time of take down of enemy e.
$t_d(e)$ ..... Time at which the body of enemy e is dropped..
P_{bh} ..... Solution path for a body hiding problem..
v_p ..... Player speed.
u ..... Game unit of distance.
f ..... Timestep.
e_p ..... Enemy position.
$\hat{e_f}$ ..... Enemy Forward.
θ ..... Field of view angle.
d_F ..... Field of view distance.
e_s ..... enemy state.
S_e ..... Domain of enemy states.
W_m ..... Movement waypoint for an enemy behaviour.
v_m ..... Waypoint movement speed.
W_r ..... Rotation waypoint for the enemy behaviour.
$\hat{w}$ ..... Waypoint direction vector.
v_r ..... Rotation speed of a waypoint.
W_w ..... Waiting waypoint.
t_w ..... Waiting time of a waypoint.
O_m ..... Movement-only obstacles..
O_v ..... Vision-only obstacles..
H_s ..... Safe spots for body hiding.
H_e ..... Exposed spots for body hiding..
Σ ..... Search space.
E_d ..... Enemy state representation.
g ..... Number of enemies.
k_p ..... Take down variable.
b_d ..... Body drop variable.
p_{ko} ..... Take down probability.
p_b ..... Bias probability.
r_h ..... Hiding radius.

t_g . . . . . Granularity factor.

Chapter 1
Introduction

Part of the appeal of modern stealth games lies in the challenge they present: the player is given the task of accomplishing certain goals while avoiding being seen by the enemies. This basic challenge, however, is frequently modified by adding more features to both player and enemies in order to make games more interesting. One of the features that are most commonly added to players is the ability to knock enemies out. Adding this feature allows a game to present stealth challenges that may seem impossible to clear only by hiding, but which turn out to be rather simple once a couple of enemies are removed. In this way, the challenge presented by stealth games becomes a puzzle in which the solution comes from finding the correct order and time to *take down* enemies.

While the reasoning behind selecting the correct movements to stay hidden in a scenario could be simply described as studying the foe's movements and timing the actions to move at their back, translating this into an automated process implies a series of considerations including the spatial representation of the scene, the sight of enemies, the dynamics of the elements, the prediction of collisions and the path planning towards a goal. Furthermore, every single feature added to either players or enemies increase the complexity of the problem by allowing or blocking some actions, thus increasing the conditions for its solutions.

In the present work, we address the take down problem by generating a model that identifies the main components of it, creating a representation that allows enemies to follow different routes over time and adding a field of view for each of them. We combine this with different sets of obstacles that can hinder either the visibility or movement of characters to create scenarios that can emulate modern stealth game spaces. As with previous works on exploring this game genre [1], [2], we make use of the *Rapidly Exploring Random Tree (RRT)* algorithm in order to analyze the stealth

puzzles, in our case adding features to handle instances where staying out of sight is not sufficient to solve the challenge. We additionally expand the RRT state space to add the enemy states, to explore the different orders in which they can be taken out. This model is applied to examine different strategies to find a solution, as well as alternatives that reduce the amount of ahead-of-time knowledge assumed of the game level, including a full-knowledge strategy in which the search knows and can predict where the enemies will be at any given time, a partial-knowledge strategy in which the search is based only in the current and previously observed states, and a corrective strategy in which the search ignores the enemies until they obstruct the path of the player and then tries to modify the path into a solution.

Additionally, we expand on the core take down problem by allowing for the enemies to stay in the scenario once neutralized. Enemies taken down act as passive or dead bodies with no movement or actions, but introduce further complexity in completing the goal: a stealth solution should also ensure a body is not subsequently seen by a live enemy. We thus allow a player to move dead bodies after the take down, adding a *body hiding* task to the take down problem.

We present our experimental work by executing the proposed solutions in two different test levels to examine the performance under different parameter settings, and then testing the best settings in three more complex scenarios which include 2D simplifications of 3D real game mechanics.

Specific contributions of this work include:

- A formalization of the take down and body hiding problems and the important features of a stealth scenario. This extends the core stealth model developed in previous works [1].

- We describe several variations on search-based solutions to handle both the take down and body hiding puzzles. Our solutions improve upon a naive search, biasing search with varying levels of prediction.

- We provide an approach to representing features included in 3D scenarios of stealth games into 2D approximations. This avoids the need for a full 3D representation and state exploration, while still faithfully representing properties of the 3D space.

- We identify the different parameters involved in the solution search and experiment on the impact they have on the success rate of the search.

- Using a non-trivial implementation in the Unity framework, we show feasibility in applying our techniques to models of stealth puzzles found in actual stealth games.

This thesis includes five more chapters which are organized as follows:

- Chapter 2 provides the information and base theory, as well as discussing previous works concerning the problems treated in this thesis and the use of RRT for the exploration of game levels.
- Chapter 3 discusses the basic theory and procedures involved in our formalization of the stealth problem, including our extensions and basic integration of RRT as a search strategy.
- Chapter 4 expands our approach to incorporate solutions to the *take down* and *body hiding* problems into the RRT search.
- Chapter 5 presents experimental data on both simple 2D test scenarios, as well as 2D versions of more complicated 3D scenes adapted from popular stealth games.
- Chapter 6 concludes our work. We also provide different options to extend on this work in the future.

Chapter 2
Background and Related Work

In this work, we will tackle the problems posed by stealth games when a take down or body hiding puzzle is given. We adapt the Rapidly-exploring Random Tree algorithm (RRT) to solve both of these problems. Thus, in this chapter, we will provide the necessary background on stealth games and pathfinding. Specifically, we will describe the basics of covert pathfinding which is a specific technique for pathfinding that is associated with stealth behaviour. Finally, we will get into the details of the RRT algorithm and how can it be used to analyze game scenarios.

2.1 Video Games

Games are one of the applications of computer science that directly reflects the evolution of new technologies. Several studies have been conducted using games as their targets and from the early days, two main areas have been identified among games: board games and video games [3].

Board games have been one of the oldest study cases in the games field, being of significant importance not only for computer science but for other disciplines like math and psychology as well [4]. An early example of this was the success of Gerald Tesauro's TD-Gammon back in 1992, which lead scientists to reconsider learning strategies that were previously considered unorthodox [5].

The interest in games comes from the fact that they requires one to "think" in order to play them, as well as they being based on a series of simple rules that can be easily modeled by math. Determining winning strategies, or optimal moves is a central focus in analyzing such games, however, in practice, the difficulty of making the optimal move does not come from following

this rules alone but also in how this optimality is defined. That is one of the main reasons why early research on games was focused on creating game-playing programs for games like chess [6], tic-tac-toe [7], checkers [8], among others. Innovations on this topic have led to systems that have been able to beat world champions and show even superhuman capabilities like IBM's Deep Blue for chess or Google DeepMind's Alpha Go for the game of Go [9], [10].

Early AI research in video games focused on playing games as effectively as possible. Some of these efforts were demonstrated in games like *Pong* (Atari 1972) and *Pac-Man* (Midway Games West, Inc., 1979). The later is of singular importance to the AI field by being one of the first games many people remember presenting enemies that show intelligent and individual behaviour. This was achieved by a technique considered simple nowadays: state machines [11]. This shows another of the earliest focus points in the field: creating Non-Playable Characters (NPCs).

Early games commonly illustrate one of the approaches followed by game designers to handle the difficulty of levels. As more enemies appear in one scenario, the more difficult it seems to the player since it will be necessary to be aware of multiple characters at the same time. This is usually paired with an increase in the speed of their movements as the game progress. This means that in order to increase the difficulty the variety of the enemies must be increased, and therefore to preserve their intelligent appearance more individual behaviours for pathfinding must be added while the engine must also be able to generate them faster as levels progress [12].

According to the literature, it was not until the early 2000s with Laird's and van Lent's article [13] that the foundations for AI in games were established and the field started to branch in a number of different applications, from narratives to procedural generation and inspiring the video game industry to integrate more sophisticated techniques in their products [3].

With the increasing capabilities in modern gaming systems more and more AI techniques become available to incorporate into games, and it is important for developers to be able to appeal to different audiences. Developers can now apply analysis techniques to their games to be sure they are suited to the players, they can test the challenge they set to be sure it can be solved, and in order to do that they can simulate player behaviours to ensure the correct design of the scenarios or to create characters that players can sympathize with. For example, several modern games have led to the creation of well-known planners like *STRIPS*, *HTN* or reactive models like *Behaviour Trees* [14], which have some analogy to the approaches in our work.

Two modern examples of how AI has been a growing interest for game developers to achieve

are the AI created for the companion NPC characters in *Bioshock Infinite* (Irrational Games, 2013) and of *The Last of Us* (Naughty Dog, 2013): Elizabeth and Ellie. The former was defined by Irrational Games' team as a companion that was designed to both follow and lead. A whole team was put in charge of designing this character which resulted in a companion that can interact with the game world, can participate in combat while avoiding blocking the player, includes a set of emotions that change her behaviour, and most important of all ensure that the player notices all of this, turning her into a character that players feel involved with [15].

Figure 2.1 – Pre-alpha footage of the procedural cover generation for Ellie in The Last of Us as shown in the 2014 Game Developer Conference. Image taken from the panel recording [16].

Ellie, Naughty Dog's character, was designed to be useful but human at the same time. It was given the ability to wander around the map, show emotions, interact with the objects and engage in combat, staying as close as possible to the player — all of these while trying to find cover spots. The latter was improved by letting the character find its own cover spots in the same way as a player would do instead of only having preset information about them, showing a procedural cover generation (Figure 2.1). In words of the development team *"...her actions are no more stupid than the player's."* [17].

2.1.1 Stealth Games

The stealth genre in games has evolved through time while presenting the same core idea for challenging the player: the goal is to avoid detection by the enemies while completing a task. This concept has been introduced in other game genres like role playing games (RPGs), first person shooters (FPS), adventure, among others, although in some cases it is not entirely clear whether the ability to avoid combat is part of the intention.

The stealth genre can also be seen as a subgenre of the action games in which combat is typically incorporated into the mechanics, with the difference that the stealth player is unfit to take on enemies in direct combat. Thus, the stealth genre could be seen as an action game where the player is limited either by low combat capabilities, low health or a disadvantage in numbers or strength compared with the enemies. Therefore, these are effectively combat games where the player is forced to find an alternative way to complete the assigned tasks. Furthermore, stealth missions often appear as segments of combat games where the aforementioned conditions force the player to change strategies for a while.

A good example of this can be seen in the game *Batman: Arkham City* (Rocksteady Studios, 2011). While in this action game Batman's abilities let the player take on almost any enemy by direct combat, sometimes he is forced to take them down by surprise when facing a horde of enemy soldiers or when the enemy is stronger, like the battle against Mr. Freeze. In this boss battle, the enemy is unaffected by any hits, thus, the player has to stay hidden and use all his weapons and surroundings to damage the enemy.

There are two main actions that the player must consider while solving a stealth scenario: movement and hiding. Movement is required to enable the player to get around a scenario to complete its goals. On the other hand, hiding is the essential activity required in order to ensure the player remains undetected by enemies.

The complexity of this game genre lies in the abilities that the game introduces for both the player and enemies. While some authors treat *Pacman* as one of the earlier stealth games for presenting a challenge where the goal is to avoid enemies [18], this game cannot be grouped into the modern stealth genre as its mechanics consist only in avoiding the enemies but does not care about detection, since enemies are always aware of the player. Clearer examples of mechanics of this genre are in modern games like *Dishonored*, *Thief* or *Assassin's Creed* which introduces

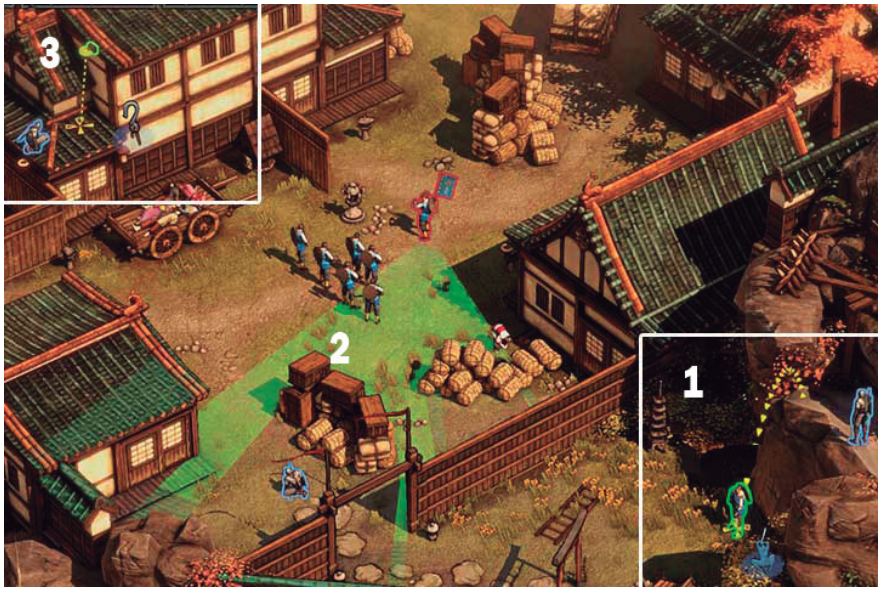


Figure 2.2 – A stealth puzzle in *Shadow Tactics: Blades of the Shogun* (Mimimi Productions, 2016). Image taken from Gamepressure's walkthrough: `https://guides.gamepressure.com/shadowtactics/guide.asp?ID=37921`

abilities that allow one to kill, knockout, or "take down" enemies without directly entering combat. Therefore, one way to stay hidden is to kill the enemies before they can notice the presence of the player or ring an alarm to other enemies. On the other hand, they gave to the enemies the ability to notice the player when discovering a dead body in the scenario, forcing the player to weigh the relative value of using take down skills over sneaking.

Given all the components of a stealth game, developers aim to create scenarios where players are faced with a puzzle that challenges one to discover the correct actions, order, and timing to perform them in order to get a clear path towards its goal. In Figure 2.2 an example of this puzzle-like behaviour is shown. The player, highlighted in blue must sneak into the enemy camp (at the top of the screen) without being noticed. Different tasks must be completed in order to achieve this. He begins in the bottom right corner of the scenario (1) where he can get behind the enemy and kill it, afterwards he has to sneak behind a bunch of boxes (2) and out of the enemy field of view (FoV). Finally, he can get to the roofs (3) from where he is able to get behind the patrolling enemies and either take them down or sneak to the goal.

In this specific genre, one of the most relevant research topics is the improvement of guard behaviours [19]. While *finite state machines* were found to be extensively used for defining guard behaviours, these were also found not to be maintainable when the behaviour was extended. A better performance was found when substituting them by *behaviour trees* that encompass modular aspects of the behaviour, while also allowing the consideration of environmental aspects in decision making. By improving the modularity of behaviour, the detection methods could be improved by using *occupancy maps*, which allow for creating probabilistic distributions of the position of the player when it is out of sight, resulting in a more realistic detection and awareness behaviour of guards.

2.2 Pathfinding

One of the most visible applications of AI in video games is agent movement or motion planning, finding a route to go from one coordinate in the game world to another (referred as start and ending or goal points, respectively).

The pathfinding problem is one of the most popular AI problems in the industry [20]. With the inclusion of non-playable characters (NPCs) in games, the need to grant them an intelligent behaviour was also born. This inherently included the need for making NPCs move across the game map and thus, the creation of paths for them to travel.

As a solution to this problem, a pathfinder will define a route through a virtual world to solve a given set of constraints. These constraints could be of different types, like ensuring the shortest path, avoiding obstacles or considering terrain specific constraints. For early games where the scenarios had a 2D basis and many of them were restricted to movement in a single axis, i.e. movement was defined by a distance and speed; the task of pathfinding might seem simple, however, with the introduction of more dimensions to the game scenarios, the variety in movement paths became a sign of individuality and thus of intelligence of the agents. This led the game designers to turn their interest into the creation of paths that could be generated by the NPCs instead of being manually set during design.

Pathfinding has had many solutions proposed, and while no standard way of categorizing these methods exists, some efforts on classifying them based on algorithm characteristics can be found in the literature. Graham et al. [21], for example, broadly classify them as either *undirected* or

directed, where undirected algorithms do not spend time planning the next move but rather go blindly around the map, while directed approaches have a method for assessing the progress of all adjacent nodes before picking one to continue. Under these definitions, the *RRT* algorithm (described in section 2.3) would be classified as a directed approach as it assesses the possible connections that can be made with a sampled node based on the distance to its neighbors.

Another important classification is based on the world geometry the approach requires. This world geometry is a simplified representation of the game map used by the pathfinder to determine the values needed to find the solution. The most common geometries are **Navigation Meshes** and **Waypoints**. A navigation mesh is a collection of convex polygons describing the *walkable* surface, while the waypoint geometry is a graph representation of the world where each node represents an important point of the map (like the corners) and the edges are safe routes to travel between nodes [22].

One of the most relevant algorithms for pathfinding in video games is the A* algorithm [23]. This directed, graph-based pathfinding algorithm was the first known algorithm to incorporate a heuristic function to estimate the distance to the goal while moving across a search graph and is proven to find a path between any two points if it exists, regardless of the obstacles, and also ensures that the solution will provide the shortest path possible. This is achieved by a composed evaluation function F that checks for the distance from the initial position G and the heuristic H that evaluates the remaining distance to the goal (Figure 2.3). Thus, this algorithm combines both Dijkstra's algorithm and a Greedy Best First Search. One of the features that make this algorithm one of the preferred approaches for developers is the fact that it is proven that no search that uses the same heuristic will expand fewer nodes than A* [20].

Although A* is commonly the standard for modern games motion planners, it also inspired different variations to improve its performance. One of the main algorithms derived from this is *D**, an algorithm created in the field of robotics which deals with the case of unknown or partially known environments by calculating a path with the initial information and then fixes it as it discovers more data from the environment [25].

On the other hand, this algorithm has also been found inefficient in some cases. One of the principal disadvantages is the time needed to spend on processing the game map to create the graph representation which makes this algorithm harder to implement on continuous spaces and may require special attention to the memory management. This issue becomes more relevant

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figure 2.3 – G values for each node on the A* algorithm for a grid world. The starting and goal points are shown in green and blue respectively [24].

when the scenarios include dynamic objects, that is, the scenario presents changes over time, like characters or objects moving around. In this case, reapplying A* every time the path is blocked could become expensive in memory usage [21], [26]. Another disadvantage of this approach arises when the game needs to present natural paths, this is, paths that present realistic turns and not robotic movements. In this case, A* needs to incorporate another kind of optimizations for turns and path smoothing. For example, Niederberger et al. [27] incorporated a *cone-of-sight* algorithm to the A* postprocessing to get a path with minimal number of nodes and allowed the agent to divert from the path up to certain degree according to the terrain slope to generate more natural paths. On the other hand, when the game needs only a solution but not exclusively the shortest path, A* might not be the best option [20].

2.2.1 Interception in Pathfinding

A specific type of dynamic scenarios are those where the goals change their position over time. An example of this is the *take down problem* where the objectives (enemies) may be moving around the scenario; in this case, the problem turns into an interception task. In this topic, previous work has been done that relates directly to ours, among them an RRT variation was created as a planning algorithm for Unnamed Ground Vehicles (UGV) for surveillance purposes [28]. This work aims to modify the *Risk-RRT algorithm*, a variation of RRT designed to operate under uncertain environ-

ments focused on avoiding moving obstacles. Taking advantage of its probabilistic representation the algorithm is adapted to intercept moving objects instead of avoiding them. For this purpose, while the original Risk-RRT algorithm calculates the probability of colliding with either a static or dynamic obstacle, the modified version uses this to search for the path with the highest probability of colliding with the dynamic goal.

The interception task has also been explored as part of human behavior simulation. One of the ways in which this problem has been tackled is by making a bot learn a reward function based on a set of human player gameplays in First Person Shooters (FPS) [29]. Using inverse reinforcement learning techniques, both navigation and interception behaviours for human-like bots were studied. For the specific case of interception, the approach is focused on making a bot generate a feature-based reward model from a set of policies observed in a set of example traces obtained from human players actions when performing multiple matches against the same opponent. That way the learned reward model is combined with particle filters allowing the bots to generate an interception plan that tries to eliminate the possible hiding spots of the opponents. It is important to notice that although this model shows a higher tracking accuracy than simple prediction methods, their work does not perform any evaluation on the likelihood of the generated model and human behaviour.

2.2.2 Covert Pathfinding

While the interception task is important for the take down problem (which is the main focus of this work), it also includes a task diametrically opposed to this. The core of the stealth problem is to find a path while staying out of the sight of foes in the scenario; this is the *covert pathfinding problem*. We can define it as the search for a route were the entity moving along the path can avoid or minimize its exposure to observers like stationary surveillance cameras or dynamic sentries moving along the terrain.

This problem comes from the field of robotics with the objective of adapting robots to act on high-risk missions where stealth is required, like military operations. A number of different approaches have been studied for this field, which had been even compiled into surveys on the topic [30]. These works have been focused on different goals which include covert pathfinding in initially known environments, using elevation of the terrain as cover, treating all areas visible to the observers as obstacles, the uncertainty of watchers positions, among many other. Some of the

approaches used to solve this problem are potential field algorithms [31], Voronoi diagrams [32], distance transform algorithms [33], and visibility maps among many others.

However, this problem is also relevant to the games field since some of them include mechanics where players must stay hidden or take cover from shots. These applications are categorized as *covert pathfinding* or stealth-based path planning [34], and its goals show a clear parallel to the challenge posed by stealth games as discussed previously.

In covert path finding all decisions and actions are based on probabilities, predictions or assumptions about the observer's positions. This means that due to the uncertainties and the nature of the problem, a solution might not be always guaranteed [30].

Important for this problem was the introduction of visibility maps [35], which are discretizations of scenarios where each position is given a value as a function of the number of other positions from where it is possible to see it [36]. Thus, a covert path or the path where the explorer has less probability of being seen can be found by choosing the spaces with the lowest values. This led to the introduction of the Dark Path algorithm (DP) which uses visibility maps in conjunction with Distance Transform (DT) algorithms to calculate the shortest, most covert path to a goal.

These visibility maps have been also combined with other techniques like *corridor maps*, which are based on Voronoi regions that produce a skeleton of the scene and a set of collision-free corridors in which it is possible to move freely. This is useful to produce both visibility polygons as well as a path from one point to another. Combining these approaches with A* search it is possible to find a path that minimizes the visibility values [34].

Among the reviewed literature, the earliest work found in covert pathfinding focused on intercepting a target (which is one of the goals of the take down problem) tackles the problem of a robot in a security system, where it is important to capture an invader moving inside a forbidden area. Furthermore, in order to be viable for a security system, the route followed by the robot must stay out of the sight of the invader, to avoid causing the target to change direction [37]. To solve this problem, they present the Active Prediction Planning Execution (APPE) strategy, which assumes that the motion of the target is predictable. Thus, a point in its path can be selected at which the robot will be sent to perform the interception. The task is divided in two parts: first, the robot has to move as close as possible to the target in a stealthy manner up to the point of the first detection, which is preferred to be as close to the interception point as possible, while the second part includes the interception of the target. To address these tasks the approach first selects an interception point

among the candidates using roadmaps and then calculates a covert path that allows for the robot to get there on time. To do that, the authors introduce *Detection Maps* as a technique that calculates a covert path by computing all the visible regions from the target position at each timestep. From this, safely-reachable intervals are calculated to get a path that avoids detection. It is important to note that this solution works in a discretized time environment.

When applied to games, the game map and its components are often known to the algorithm and it has access to the location of every observer at any given time step, although this information can be left out of the algorithm in order to add more realism to the simulation [38].

2.3 Rapidly-exploring Random Tree (RRT)

To solve the problems posed by the optimal path algorithms like A*, several alternatives were developed for pathfinding. Among them, the RRT algorithm originally presented by LaValle [39] [40] became the basis of this work.

The Rapidly-exploring Random Tree is a data structure used for incremental randomized sample-based pathfinding. This algorithm has been studied because of the characteristics it possesses. First of all, it is a simple and fast pathfinding approach when exploring high dimensional spaces compared with algorithms such as A* which has the need to search among the whole state-space. Furthermore, the RRT approach has been proved to be "probabilistically complete", meaning that it will always find a solution eventually although it may benefit from restarting. It can be augmented to add more properties and restrictions while its randomness will provide different solutions under the same conditions; in contrast, algorithms like A* will always find the same optimal solution unless changes in the algorithm or scenario are performed. This randomness is also useful to simulate the individual preferences or choices that two or more intelligent beings might prefer.

Given the $\mathbf{n}$ dimensional space $\mathbf{S}$ where a path from the starting point $\mathbf{A}$ to the ending point $\mathbf{B}$ must be found, and considering each state inside the state-space $\mathbf{S}$ and as a node $\mathbf{x}$; the RRT algorithm can be conceptually described as a process of iteratively sampling a new state $\mathbf{x}$ at random from $\mathbf{S}$ which will grow the RRT tree by connecting it to the nearest neighbour $\mathbf{x_{near}}$ in the current tree. The algorithm terminates when the goal state $\mathbf{B}$ is reachable from the latest sampled state [40]. This algorithm's pseudocode is shown in Algorithm 1.

After executing the function described, we can find the solution path $\mathbf{P}$ by traveling backwards

Algorithm 1 RRT Algorithm

Require: $A, B \in S$
1: **procedure** GENERATEPATH(S, A, B)
2: $RRT \leftarrow A$
3: **while** True **do**
4: $x \leftarrow RANDOM(S)$
5: $x_{near} \leftarrow$ NEAREST_NEIGHBOR(RRT, x)
6: **if** !IS_VALID(x, x_{near}) **then** // Verifies constraints.
7: **continue**
8: **end if**
9: $x.parent \leftarrow x_{near}$
10: RRT.ADD(x)
11: **if** CAN_CONNECT(x, B) **then** // Goal is reachable.
12: $B.parent \leftarrow x$
13: **break**
14: **end if**
15: **end while**
16: **end procedure**

along the RRT structure starting from point $\mathbf{B}$ which will eventually take us back to $\mathbf{A}$. It is important to note that the algorithm previously described has no boundaries or limits as to how big the RRT structure should grow or about the validity of the chosen nodes. However, these restrictions can be simply set by tweaking the algorithm a bit. This is, if a finite loop is used instead of the infinite one in the code, the RRT can be bounded to $\mathbf{m}$ nodes. On the other hand, any number of checks can be performed after picking the nearest neighbour to add any constraint about the distance, the place or the values of the chosen state.

Because of its random behaviour, the RRT algorithm first sparsely explores the whole extent of the available connected space before starting to increase the density of nodes. This has a high probability of an increased density around the first nodes selected, due to the connection step. Thus, the algorithm benefits from restarting, as for every new search the algorithm will choose an entirely different potential path [41]. This is useful to analyze the complexity of scenarios as each path could represent a different player, taking different actions and exploring different spaces. The diversity of solution paths for a scenario could be an indicator of more freedom players will have to solve it; on the other hand, the less diverse the paths the more precise actions players will have

to perform to find a solution [2].

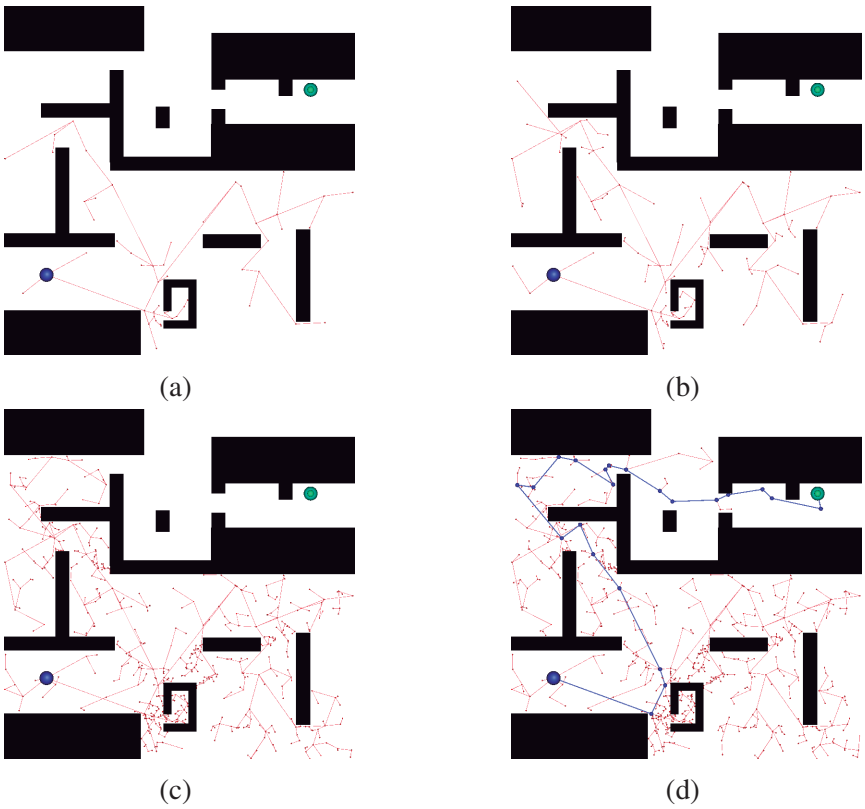


Figure 2.4 – Example of the expansion of the RRT structure for (a) 50, (b) 100, (c) 500 and (d) 642 nodes, while solving a pathfinding problem. The starting position (root) is shown as the biggest blue dot and the goal is shown in green.

Figure 2.4 shows an example of the expansion of the RRT structure in a test scenario where obstacles are included. To do so, the search shown does not include any limit about the distance between the nodes, however, all the randomly sampled nodes must belong to the free space (shown in white) and the connection to the nearest neighbor cannot intersect with an obstacle in order to be added to the RRT. In the last part of the expansion in Figure 2.4, the solution path is shown in blue.

In the case of stealth games, the RRT algorithm has been shown to be useful due to its exploration behaviour, which ensures eventual exploration of the entire space. Furthermore, the granularity of this exploration can be set up by fixing the number of iterations to execute. Additionally,

the RRT has been shown to perform correctly on high dimensional spaces, which is convenient to study stealth games that include dynamic obstacles or enemies. Finally, the random behaviour of this algorithm is useful to study the number of different paths that can be followed to solve a single scenario.

While the RRT approach comes from the field of robotics and was proposed for holonomic, nonholonomic and kinodynamic planning problems, it has been successfully implemented in other areas like bioinformatics for deriving binding peptides [42] and has been previously used for analyzing game scenarios by creating a graph representation using clusters of RRT paths [43], as well as for more direct pathfinding in real strategy games.

For example, Naveed, Kitchin, and Crampton presented two approaches for pathfinding in Real Time Strategy games (RTS) [44], comparing an Upper Confidence Tree (UCT) planning system [45] with a variation where RRT is used as the sampling method. The UCT is a Monte-Carlo planner which, for each iteration (also called *rollout*) expands a node in the search tree, creating child nodes representing the results of each possible action. Once expanded, these nodes are evaluated based on some heuristic, and a corresponding reward is propagated up the tree. After a number of rollouts, the action with the highest predicted reward is applied. The authors used the open RTS game engine (ORTS) for executing the planners for a fixed amount of time. They were able to conclude that although simple UCT is faster at finding solutions,the RRT performs better in terms of the game scores obtained by the solutions because of the larger portion of the state space explored.

Bauer and Popović analyzed platform games using RRT to study the reachability of the scenarios and their goals [43]. By executing the RRT algorithm using the whole scenario as the state space, and performing a random action to generate a new state, they were able to obtain a graph representation of the game levels covering all the spots a player can reach. Applying a clustering method to the RRT representation, they were able to obtain an approximation of the complexity of reaching each location in the scenario, based on the fact that the hardest places to reach are those where more precise movements are needed, and therefore the RRT algorithm will add those nodes less frequently. Furthermore, during the clustering step, the edges of the clustered graph were weighted based on the number of edges in the original RRT structure that were joined to create each cluster edge, which led the authors to hypothesize about the relationship between the edge weight and the difficulty of the movement. This method was applied to the platformer game:

Treefrog Treasure developed in the University of Washington. The authors opted for incorporating their work into a level editor to allow the developer to visualize how changes in the scenarios affect its complexity, allowing for them to obtain real-time feedback about the reachability in its static scenarios.

A similar work in platformers presented a verification tool for the level scenarios using pathfinding algorithms [46]. A comparison between approaches using A*, Monte-Carlo Tree Search (MCTS) and RRT was performed. It is important to notice that during this work 2D based A* was expanded into a third dimension adding the time component to analyze dynamic environments. However, in the case of RRT, it was only studied in 2D searches. Furthermore, the RRT algorithm was also modified to use either A* or MCTS to find routes that can join a sampled node with the nearest neighbor. Through their experiments using *Unity*, a set of static and dynamic scenarios (with moving obstacles and platforms) were analyzed in order to verify the capabilities of the algorithms in finding not only a solution but a set of them to compare how diverse they were.

An important aspect tackled by the previous work is that of dynamic environments. This feature is of relevance for pathfinding systems since the changes over time in the scenario could allow or invalidate a solution at different time frames, or may represent a need for re-planning; while, on the other hand, it is a feature often presented in modern games.

Finally, it is important to review two works that are not only relevant but became a basis for ours. The first work presents an approach for generating a tool that can help stealth level designers to validate the scenarios by analyzing the possible paths a player can take, as well as the visibility in the scenario [1]. This was achieved by implementing an RRT search to generate a set of paths over a scenario with static and dynamic obstacles. After finding them a clustering step is performed that allows the creation of a heat map of the routes available to players, and from which one can generate a visibility map based on the routes. This map shows potential safe spots that allow for the designer to check if the expected behaviour is achieved or there were flaws a player can exploit. Furthermore, this safe spot analysis is one of the tasks involved in the body hiding problem presented in this work. It is important to notice some of the characteristics of the tool presented in this work. First, a discretized form of the 2D space is needed for the analysis of the scenario and more important, all the scenarios presented in the experiments of this work present the simple stealth problem where there is at least one path that can take the player to the goal without facing any of the sentries. The same authors, later extended their design to incorporate combat mechanics

to solve this stealth problem [47]. This allows one to eliminate enemies by engaging in combat with them once the RRT samples a node inside a field of view, thus, clearing other areas in the map where a path to the goal can be found. However, in contrast to our work, in order to be capable of solving a stealth problem that permits combat as well, a system of health, strength and even items had to be implemented.

Based on the previous works, one proposed approach to solve the basic stealth problem was focused on *distractions* [2]. Distractions are often found in stealth games, allowing players to create a noise or visual disturbance that attracts guards away from their normal route, and thus expose additional opportunities for player movement. This was implemented by defining certain triggers which modify the normal path of the sentries in the scenario when activated. In this way, an RRT algorithm can explore the use of such distractions and might be able to turn the stealth problem into a solvable pathfinding for a period of time. Many of the approaches presented in our work are based on this approach, however, it is important to note that some improvements were made to the algorithms presented to allow for more accurate collision detection in an approximated continuous time dimension.

Chapter 3
Methodology

In this chapter, we elaborate on the process followed to model and generate a solution for the stealth problem. We first present a formalization of the problem and its variations. After this, we discuss the approaches taken to reproduce its elements and characteristics in our experiments. Finally, we focus on the search algorithm implemented to solve the stealth problem and introduce the elements related to the take down and body hiding problems.

3.1 Problem Analysis

This work focuses on two of the tasks players might found in a stealth game level: the *Take Down* and the *Body Hiding* problem. Both of them are augmentations of the general goal of a stealth game, which is to get from point A to point B without being detected by foes.

The take down problem arises when the player cannot complete such task because every path that can take him from A to B puts him in a position where he can be detected by at least one foe. Therefore, he needs to get rid of at least one of them to complete his goal. The body hiding problem augments this even more by making each taken down enemy leave a trace that must not be detected by the others before the player reaches his goal, however, it allows the player to move this *evidence*. Given these two problems we can identify a number of factors that must be considered to find a solution and that will impact in its performance.

- **Scenario space**: This affects directly the pathfinding algorithm search space as it can increase or decrease the state space.

- **Agent representation**: This is the way we represent a player or enemy in space (e.g., represented as points, regular volumes, bounding boxes, etc.). It directly affects the complexity of the algorithm by defining how operations such as collision detection must be performed.
- **Movement**: This is a primary factor that determines the kind of pathfinding algorithm that can be implemented.
- **Obstacles**: The representation of obstacles inside the scenario impacts the complexity of the pathfinding by making it more or less complex to validate if a position can be occupied by an agent or not.
- **Detection mechanic**: Another important aspect that also affects the solution algorithm, as it determines when the player has failed in its task. For example, a mechanic where an enemy can only detect a player if he is at a distance less than a defined threshold requires less effort to validate than a mechanic that emulates detection by hearing the player, which would require modeling the way the sounds behaves.
- **Take down behaviour**: This is the conditions required to perform a take down action, its consequences and the way they will be introduced during the solution computation. These constraints add a direct complexity to the pathfinding as they must be validated every time a take down action is meant to be performed. To understand it better we can take as example a take down behaviour where a take down will be attempted at every step during the pathfinding with the only condition of having no obstacle between the player and an enemy (i.e. take down by long range weapon). This case will be easier to validate than a take down behaviour where the action will be attempted only after finding a path to the goal but such path can be detected by an enemy and where the take down condition is that the player and enemy must be close to an obstacle (e.g. pushing the enemy into a trap).
- **Body hiding activation**: Similar to the previous factor the moment in which a body hiding task must be performed affects the complexity of the search, e.g., requiring to hide a body after every take down or allowing to skip the task changes the size of the solution paths.
- **Hidden zones**: The nature of the valid places to hide a body strongly changes the search algorithm complexity, since having preset zones turns this into another pathfinding subtask,

On the other hand, requiring the solution to compute these zones adds a different problem that must be addressed by the algorithm.

In the following sections we will formalize each of these factors and at the end of Chapter 4 an analysis of the complexity introduced by each of them on our proposed solution will be presented.

3.2 Formalization of the Stealth Problem

Following the analysis of the previous section and in order to understand the involved factors better, we will formalize each of them in a more technical way. Figure 3.1 exemplifies a simple stealth problem and its solution; we present this to help in identifying its components.

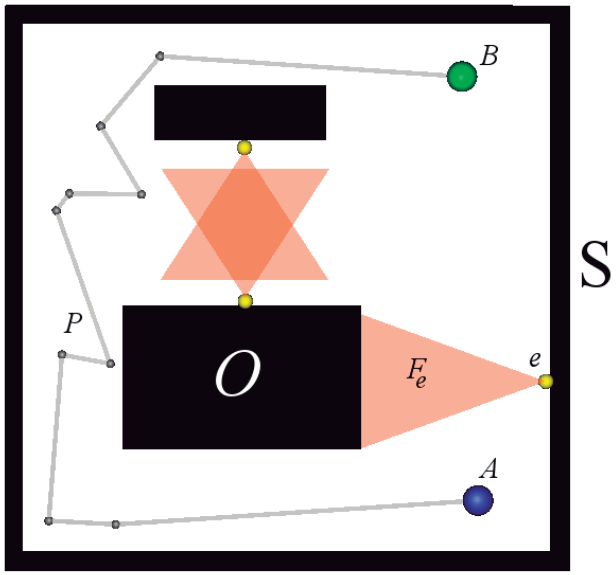


Figure 3.1 – Basic stealth problem solution.

3.2.1 Pathfinding Formalization

A first step towards the formalization of a stealth problem is to identify the elements inherited from a simple pathfinding problem and mapping them into a game scenario. These elements are:

- A **coordinate space** $\mathbf{S}$ of $\mathbf{n}$ dimensions. Where $\mathbf{s}$, is any state in the space, i.e. $\mathbf{s} \in \mathbf{S}$ which is named a **node**. It is represented in the figure as the whole space.

- A set $\mathbf{O} = \{o_1, o_2, ..., o_k\}$ where $\mathbf{o_i}$ is a node with coordinates $(o_{i_1}, o_{i_2}, ..., o_{i_n})$ that represents an **obstacle**. Such node $\mathbf{o_i}$ cannot be included in the solution of the problem and is shown in the figure as black spaces.

- A subset $\mathbf{S_{free}} \subseteq \mathbf{S}$ that defines the free space that can be explored by the player inside the state space. It can be defined as $\mathbf{S_{free}} = \mathbf{S} - \mathbf{O}$ and is shown as non-black space in the image.

- A set $\mathbf{T}$ of valid transitions between points (nodes) in the coordinate space, this is: $\mathbf{T} \subseteq \{\{s_1, s_2\} | s_1, s_2 \in \mathbf{S}\}$

- A **starting node** $\mathbf{A} \in \mathbf{S_{free}}$ with coordinates $(a_1, a_2, ..., a_n)$. Shown as a blue dot in the figure.

- An **ending node** $\mathbf{B} \in \mathbf{S_{free}}$ with coordinates $(b_1, b_2, ..., b_n)$. Shown as a green dot in the figure.

Given the previous elements, we can define a pathfinding solution as:

$$P = \{(p_0, p_1, ..., p_m) \,|\, p_i \in S_{free} \ \wedge \ \{p_i, p_{i+1}\} \in T \ \wedge \ p_0 = A \ \wedge \ p_m = B\} \tag{3.1}$$

That is, *the **solution for a pathfinding problem** is a sequence* $\mathbf{P}$ *of points in the free space* $\mathbf{S}_{free}$ *that begins in point* $\mathbf{A}$ *and ends in point* $\mathbf{B}$*, such that, for each pair of successive points* (p_i, p_{i+1}) *in* $\mathbf{P}$*, a transition in* $\mathbf{T}$ *exists between them.*

3.2.2 Static Stealth Problem

In order to turn a pathfinding problem into a stealth problem we need to consider the existence of a **set of enemies** $\mathbf{E}$ blocking our way to the goal. Each enemy is defined by its position inside the space. Figure 3.1 shows enemies as yellow dots at specific locations. In this static version of the problem, enemies are assumed to be unmoving, with a fixed orientation.

The goal here is for the solution path to avoid the enemies, and thus we can treat enemies as static obstacles, i.e. $\mathbf{O_s}= \mathbf{O} \cup \mathbf{E}$. However, they have an added feature that makes them different than simple obstacles: the **Field of View (FoV)**. The FoV is an area inside the space that the enemy is aware of and thus, cannot be part of the solution since the main goal is to be unnoticed by any enemy.

Formally, this FoV element for enemy $e \in E$ can be defined as:

$$F_e \subseteq S \tag{3.2}$$

This is, *each* ***enemy*** $\mathbf{e} \in \mathbf{E}$ *is associated with an area* $\mathbf{F_e}$ *named* ***the Field of View (FoV) of*** $\mathbf{e}$ *which is a subset of points from the coordinate space* $\mathbf{S}$. Figure 3.1 illustrates this FoV as orange triangular areas projected from the associated enemy position.

If each point in the FoV is treated as a normal obstacle we have:

$$O_s = O \cup \bigcup \{F_e \mid e \in E\} \tag{3.3}$$

Which defines the set $\mathbf{O_s}$ of **obstacles in a stealth problem** as *the union of the* ***static obstacles in the game environment*** $\mathbf{O}$ *and the set of points inside all of the* ***FoVs*** *of the enemies* $\mathbf{F_e}$ in the scenario. Given this, we can use the previous definition of $\mathbf{P}$ to get a solution for the most basic stealth problem with static enemies and obstacles.

Going back to figure 3.1, where an example of the basic stealth problem is illustrated, we can define its elements in the domains that this work focuses on. That is, the state space for a game scenario like this belongs to that of a continuous two-dimensional space $\mathbf{S} = \mathbb{R}^{\mathbf{2}}$. Therefore, for this work $\mathbf{n = 2}$.

Since we are dealing with a continuous real state space we must substitute the definition of available transitions $\mathbf{T}$ to consider the characteristics of this space. Therefore, we consider that for any two states $\mathbf{s_1}$ and $\mathbf{s_2}$ in the space, a valid transition from $\mathbf{s_1}$ to $\mathbf{s_2}$ exists if both states can be connected by a continuous line segment $\overline{\mathbf{s_1 s_2}}$ such that, all points in such segment belongs to the **free space** $\mathbf{S_{free}}$.

The subset of obstacles $\mathbf{O}$ for any of the game scenarios in this work is shown as black portions of the space as illustrated in figure 3.1. The enemies $\mathbf{e_i} \in \mathbf{E}$ are represented as yellow dots, identified by the coordinate of its center.

The solution path $\mathbf{P}$ for the example is illustrated in grey. In this case, since we are working in the continuous space $\mathbb{R}^{\mathbf{2}}$, and because of the nature of the transitions stated above we can formalize the solution as:

$$P_{\mathbb{R}^2} = \{(p_0, p_1, ..., p_m) \mid \overline{p_i p_{i+1}} \in S_{free} \, \forall \, 0 \leq i < m \ \wedge \ p_0 = A \ \wedge \ p_m = B\} \tag{3.4}$$

This means, the solution path $\mathbf{P}_{\mathbb{R}^2}$ can be represented as a set of successive line segments, where the visited states $(\mathbf{p_0}, \mathbf{p_1}, ..., \mathbf{p_m})$ belong to the free space $\mathbf{S_{free}}$ and are those points that describe each of the segments $\overline{\mathbf{p_i p_{i+1}}}$. Furthermore, the path can be summarized only by the beginning and end of each of these segments i.e. $\mathbf{P} = (\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_m})$.

3.2.3 Dynamic Stealth Problem

Although the previous definitions present a formalization to the stealth problem and its solution, it only encompasses static behaviours for the enemies, while in modern games it is sometimes the dynamic behaviour of them what makes stealth games challenging. Thus, it is imperative to include this in the formalization.

The dynamic nature of the enemies can be defined as the capability of them to change their position and direction around the free space of the scenario over time. Due to this, in order to observe and deal with the dynamic aspect, a time dimension must be added to the model. To do this, we follow the same approach as previous works [2], [47] extruding a two-dimensional polygonal scenario into a third dimension which represents the time. Figure 3.2 shows the same level presented in figure 3.1 but extruded into the time dimension. It is important to note that although the original obstacles of the scenario in figure 3.2 seems to be unaffected by the extra dimension, this is only for illustration purposes as they are considered to stay in the same positions at any time, which would generate polyhedrons similar to those of the enemy FoVs.

A successful solution in this model is thus a path from start to finish that avoids both physical obstacles and obstacles representing enemy FoVs while following Equation 3.4. However, by adding the time dimension we must also constrain the solution by the nature of this new parameter. First of all, for the solution we aim at, time values can only change in one direction (time can only go forward) and we cannot have negative values for it. Thus, a path of length k in this extruded domain can be expressed as a function of time:

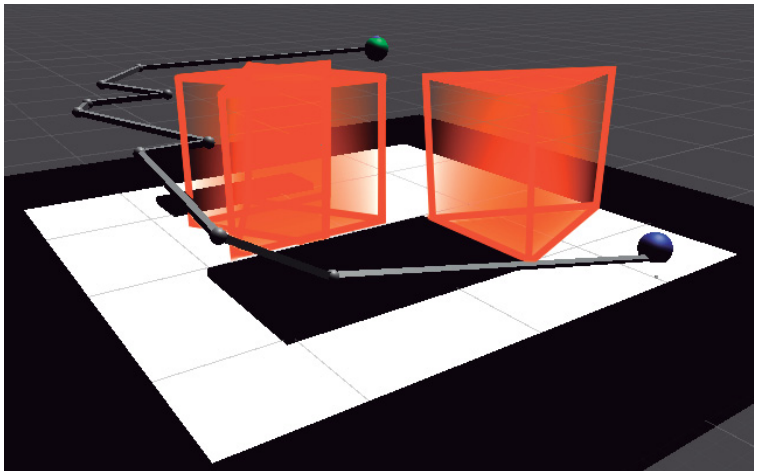


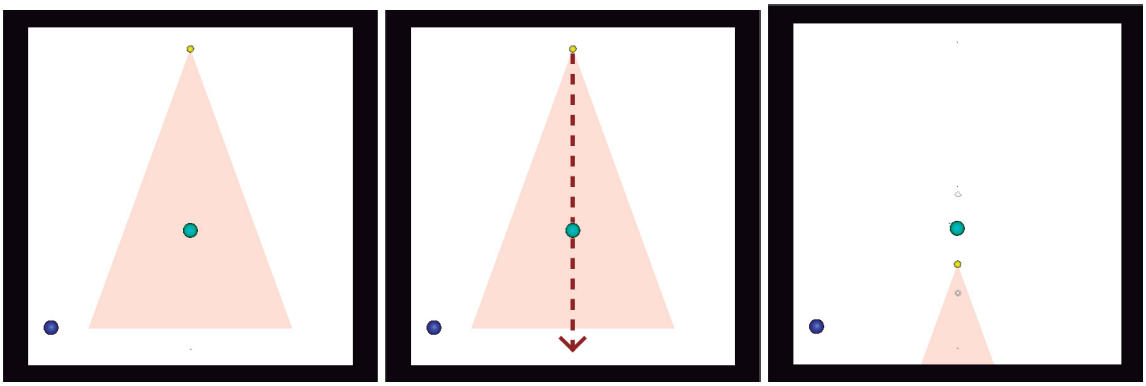
Figure 3.2 – State space representation of a stealth problem where a 2D scenario is augmented with an extra dimension representing the time. The grid in the image represents the XZ plane while the vertical axis represents the extruded time dimension.

$$\begin{gathered} P(t) : \mathbb{R}^+ \cup 0 \rightarrow S_{free} \\ P(t_i) = p_i(x_i, z_i, t_i) \quad s.t. \quad p_i \in S_{free} \, \forall \, 0 \leq i \leq k \ \wedge \ p_0 = A \ \wedge \ p_k = B \end{gathered} \tag{3.5}$$

This is, the **solution path** $\mathbf{P}$ is a continuous function over **time** $\mathbf{t}$, such that, for any given positive time value $\mathbf{t_i}$ from 0 to $\mathbf{k}$ there is a state $\mathbf{p_i}$ inside the free space $\mathbf{S_{free}}$ that follows a polygonal path between the start point $\mathbf{A}$ and ending point $\mathbf{B}$. We call this upper bound $\mathbf{k}$ the **solution length.** A shown in the previous equation we redefine our state space to consider this additional dimension. Since any state $\mathbf{s}$ in the state space $\mathbf{S} = \mathbb{R}^\mathbf{2}$ was defined by its two coordinates $(\mathbf{x}, \mathbf{z})$ (we use $\mathbf{z}$ instead of $\mathbf{y}$ for convenience, as these are the coordinates used by the Unity engine when working on 2D spaces), for the extruded state space $\mathbf{S} = (\mathbb{R}^\mathbf{2}, \mathbb{R}^+)$ we now have that any state $\mathbf{s}$ can be represented by its coordinates plus the added time value $(\mathbf{x}, \mathbf{z}, \mathbf{t})$.

Given the previous definition, a solution can be found for scenarios with similar settings like the one shown in figure 3.2. In this scenario, the enemy behaviour is extruded over time. Since they are all static, that is, they stay at the same $\mathbf{x}, \mathbf{z}$ coordinates during all the execution of the scenario, so does their field of view, while, at the same time, travelling along time. If we superimpose all the FoV positions over time we end up with a polyhedron similar to a triangular prism for each of the enemies. Given these shapes, the problem is only to find a path in the three dimensions avoiding

all of these shapes, a similar approach to considering them as a simple obstacle.



(a) (b) (c)

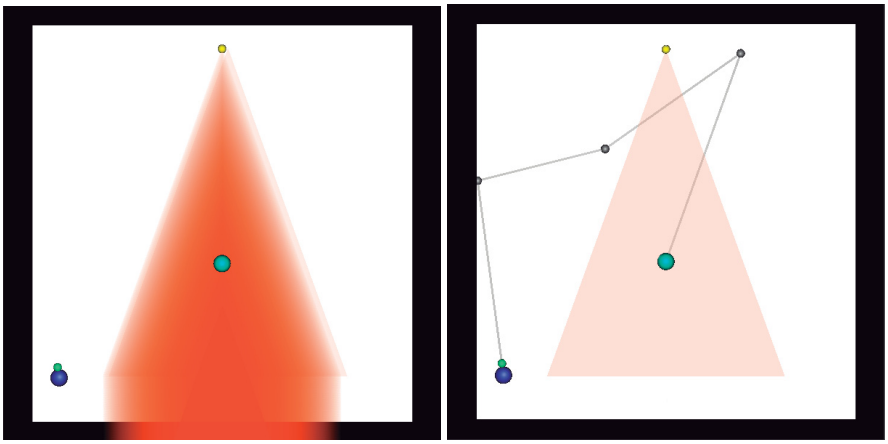
Figure 3.3 – Example of a dynamic stealth scenario. (a) Initial setup at $t = 0$. (b) Enemy movement. (c) Final enemy position at $t = k$.

Enemy Movement

Although the example in figure 3.2 illustrates how to handle a scenario that considers the time dimension it is not useful if the scenario does not present any change through time. Furthermore, one of the main characteristics of stealth games is that they create challenging puzzles by giving the enemies behaviours that interfere with the goal of the player. One of the most elemental of these behaviours is the movement. Enemies that can travel along the scenario pose a harder challenge to players as they must be aware of the location and direction of the FoV at any time since a safe spot at a certain time might be observed in the future.

Figure 3.3 illustrates a simple scenario showing this movement mechanic. In 3.3a the initial setup is shown; the starting point is shown as a big blue circle at the bottom left corner, the ending point is shown as a big circle in the middle of the scene and the enemy is shown as a yellow circle at the top section with its field of view shaded in orange (this will be the representation used for all of the scenarios in this work). In part 3.3b we can see a representation of the movement the enemy will perform during the execution of the scenario, it will travel in a straight line from the top to the bottom of the screen.

As it can be observed, the enemy in its initial setup covers the goal position with its FoV, if



(a) (b)

Figure 3.4 – Example of a dynamic stealth scenario. (a) Visible zones over time. (b) Solution path in the 2D space.

we tried to solve the scenario with the definitions given until now and considering only this initial setup, we will not be able to find a solution path. However, if we executed the scenario for a given amount of time, let us say $\mathbf{k}$, at some point this goal would not be inside the enemy FoV as shown in figure 3.3c where a valid path from the beginning to the goal can be seen. Therefore, the problem for this scenario is expanded not only to find a path between start and finish but a path with the right timing.

Although finding this path might seem straightforward it is important to be careful with the movement mechanic and how it is treated. For example, a covert pathfinding approach could try to analyze the scenario prior to its execution and calculate the whole space that will be under a given FoV during execution. Then, it could try to find a path using only the space that is never or least often seen. Figure 3.4a shows a representation of this approach applied to the scenario in figure 3.3, where orange areas represent coordinates that are seen by the enemy during its movement, and the colour intensity represents the frequency of this. As it can be observed, this kind of approach will find the zones near to the goal to have a high probability of being seen and this might affect the performance of a search using this kind of heuristic.

Figure 3.4b shows a solution path for the example scenario, in this case, if we take a look at it in the original state space it might seem like the solution is incorrect as it does not satisfy the

condition of the solution path definition which states that every state $\mathbf{p_i}$ in the solution path must be part of the free space $\mathbf{S_{free}}$, that is, no point in the polygonal path should intersect with a FoV or obstacle as it does in the illustration. However, when we consider the time dimension as a third axis in the state space, we can see a more accurate representation of the solution path for a scenario with dynamic enemies, as shown in figure 3.5. As we observe here, the solution path avoids the polyhedron created by the FoV over time, this means that for any time value the path will only explore a single state in the free space at such time.

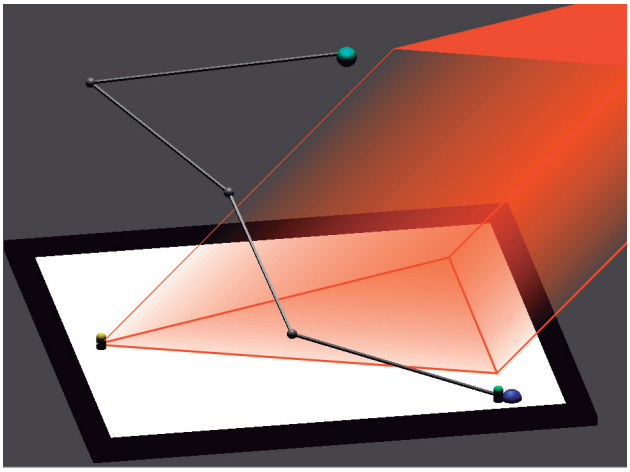


Figure 3.5 – Solution for the example scenario in the extruded state space.

Some of the important characteristics of the solutions for the dynamic stealth problem can be observed in the illustration. First, the path should always describe a continuous and ascendant path with respect to the time, this because the agent cannot travel backwards in time, neither can he occupy two states at the same time value since we are not dealing with mechanics that allows for instant teleportation. Thus, we can extend the definition in Equation 3.5 with the condition:

$$\forall \{p_i(x_i, z_i, t_i),\ p_j(x_j, z_j, t_j)\} \in P(t) \text{ if } j > i \text{ then } t_j > t_i \tag{3.6}$$

Another important fact is that the goal state has no restriction on the time at which it can be reached. That is, as long as the path achieves the x and z coordinates of the goal, the time at which this happens is irrelevant.

Another aspect worthy of notice in the extruded scenario are the different slopes of the path segments. These are due to the speed of the player while traveling each segment; since the vertical axis represents the added time dimension, the closer a segment is to horizontal the faster the speed. A completely horizontal segment would mean to travel from one point to another in no time, a teleportation behaviour, which will not be allowed for the scenarios explored in our work. On the other hand, the closer to vertical a segment is, the lower the speed of the player; an actual vertical segment would mean that the player stays in the same position for a given time interval.

Enemy Rotation

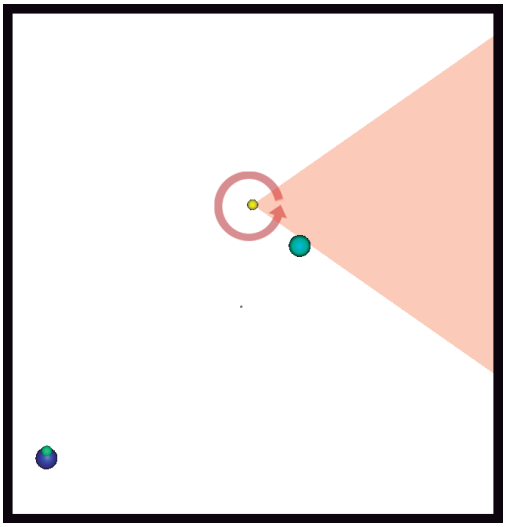


Figure 3.6 – Basic scenario with a rotating enemy.

While the extruded model shown in the previous section allows us to solve scenarios where enemies describe more complex polygonal trajectories, if the movement described goes in more than one direction the design must additionally define the direction the enemy is looking at (we will call this the **view direction**). For example, for human models, most of the time the direction observed is the same as the direction of the movement. This indicates that a change in the position of the FoV with respect to the enemy can be performed when it moves in a different direction.

Another case is that of enemies modeled after fixed observers like security cameras. In this case, although they cannot move around the scenario, they do normally present the ability to ro-

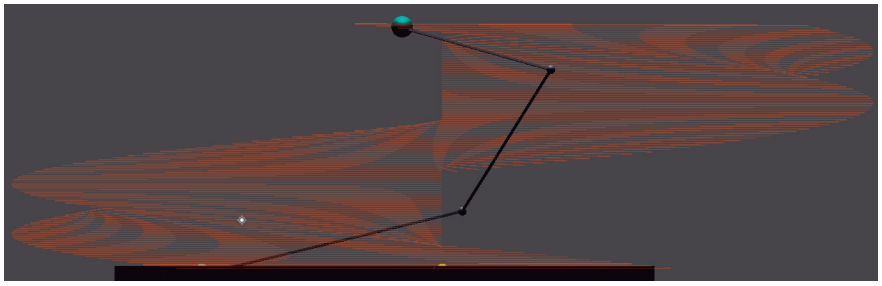
tate around their own axis to cover a greater space. This rotation movement is important to be implemented in any design related with stealth games, as in modern games, players can normally take advantage of the blind spots of the enemies—the zones outside of their sight that represent a weakness in their perception and thus an opportunity to deal with them.

A single scenario representing this motion is shown in figure 3.6. In this 2D representation, a single enemy is given the ability to rotate in a counter-clockwise direction during the execution. Similar to the case presented in the previous section this prevents the player from finding a path to the goal (shown in cyan color). Analyzing this scenario, we can deduce that every single coordinate in the 2D scenario will be covered by the FoV of the enemy at some time. Although this may sound like a setting where the player will be unable to find a place where it will not be seen, this is only partially true if we only consider the visibility over the 2D space.

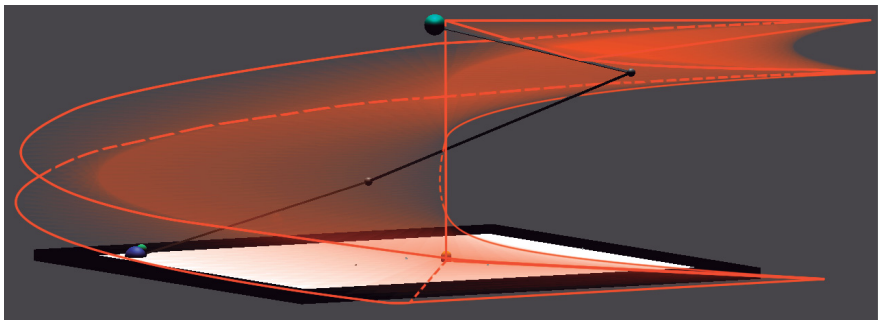
Once again, by introducing the time dimension, we can get another outlook of the motion. Figure 3.7 presents the same scenario extruded into a third dimension. In this case, the FoV of the enemy changes its position over time, shown as orange slices over the vertical axis. In the illustration we can see that the union of all of these slices creates a helicoidal shape, which shows open spaces that are not covered by it, thus, allowing for a solution to the task. This solution can be found by applying the definition in Equation 3.5 and is shown as dark-grey polygonal segments traveling through the empty spaces of the helicoidal shape and reaching the goal position at the top of it. Figure 3.7b shows the same solution seen from a different perspective, where the FoVs have been abstracted as a single and continuous helicoidal form representing an obstacle that must be avoided by the path.

Given the behaviors presented above, it is possible to modelmore complex scenarios using these two motions, furthermore, these are two of the behaviours for enemies shown to be used in modern stealth games to create the challenges of this genre. Figure 3.8 shows a sample scenario combining both the rotation and movement behaviour. In this scenario, the player is situated at one end of a corridor and is given the task of reaching the other end, which is guarded by an enemy. As shown in figure 3.8a the enemy will travel along the corridor from one end to another, then it will rotate 180°and repeat. Although the FoV of the enemy blocks the player from getting through the enemy, there are several alcoves along both walls of the corridor where the enemy FoV cannot reach, this may be used by the player to hide when the enemy is near.

Figure 3.8b shows the solution for the sample scenario as seen in the 2D state space, while



(a) FoV behavior during a rotation movement as seen projected into the vertical axis.



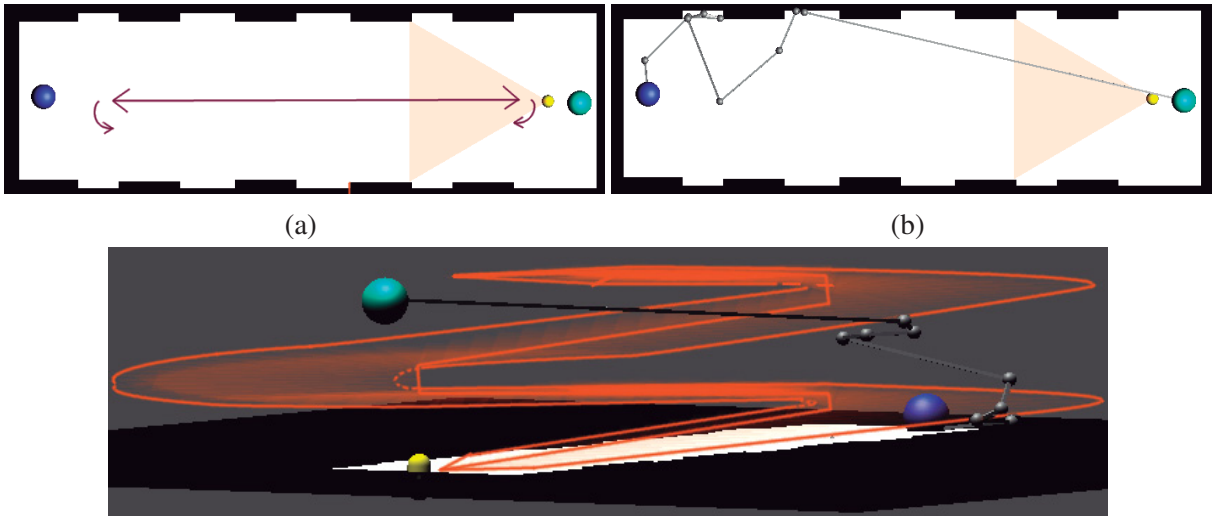
(b) The helicoidal shape created by the superposition of the FoVs over time.

Figure 3.7 – Solution to the basic scenario with a rotating enemy.

figure 3.8c shows this same solution in the extruded state space, as well as an approximation to the shape generated by the FoV superposition. Both representations show how the player is forced to wait in the spaces on the wall until the enemy has passed by and until it has enough advantage to aim for the goal without being caught by the enemy FoV.

3.2.4 Take Down Behaviour

While we have formalized the stealth problem as well as its solution path for both the static and dynamic scenarios, the main focus of this work are those scenarios that cannot be solved only with such definitions. That is, the scenarios shown in the previous sections have all shown to have at least one path that can get from start to end without being intercepted by an enemy, or for the case of the dynamic scenario, there is a certain time at which the player can move towards the goal without being caught. However, although this already presents a challenge for the player,



(a) (b)

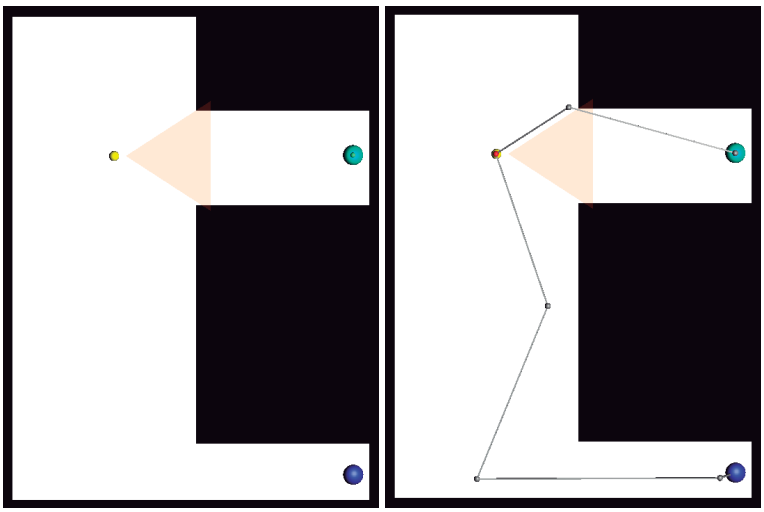
(c)

Figure 3.8 – Example scenario combining both movement and rotation. (a) Initial configuration. (b) Solution path. (c) Solution in the extruded state space.

its behaviour is restricted to simply moving around. Furthermore, there are some configurations of the elements presented up to this point that create scenarios that cannot be solved by a player with such restrictions. Figure 3.9a shows an example of such a configuration. In the example, the player is faced with the task of traveling from a room into another. However, this goal room is being guarded by a static enemy: the enemy does not change through time and there is no spot in the entrance that is not covered by its FoV. In this case, if we apply definition 3.5 we will find that there is no function $\mathbf{P}$ that can get to the goal location going only through the free space $\mathbf{S}_{\mathbf{free}}$ because of the enemy blocking all the states in the only entrance to the room.

Although this might be taken as bad level design, a more interesting stealth game should allow for the players to deal with scenarios like this. An example of this could be the introduction of the combat mechanic as shown in previous work [47]. However, in stealth games where the goal is to stay hidden as much as possible, combat mechanics may contradict this. On the other hand, another type of mechanic commonly seen in stealth games is that of silently taking down the enemies by quickly attacking them in a lethal or non-lethal way without them noticing, approaching them from a spot out of their sight, which is commonly their back. This way enemies can be eliminated leaving free states that were blocked by them before and thus, increasing the free space $\mathbf{S}_{\mathbf{free}}$.

It is important to note, that although sharing some characteristics, the combat and stealth mechanics have some fundamental differences. First of all, in order to engage in combat with an enemy, the player does normally need to be at a certain distance from the target regardless of the direction. On the other hand, for a stealth mechanic, it is required to approach the enemy from a direction out of its sight, as the goal is to remain undetected. This detection factor could be not considered in the combat mechanic or even be used as a trigger for it. A second difference is that a stealth take down is considered to be instantaneous; that is, the moment it is triggered, the outcome is reflected in the state of the scenario, while a combat mechanic requires a model for performing this action, which might consider factors like attack, health, speed, among others, with the outcome only reflected after the battle is completed. In this sense the stealth and combat mechanics could be seen as complementary to each other. Furthermore, in modern games, it is usual to have two options to complete a task, either by engaging in direct combat with all the enemies or taking them down stealthily to avoid damage. Some games even present additional challenges based on requiring either of them.



(a) (b)

Figure 3.9 – Sample scenario where take down is needed to be solved. (a) The configuration of the scenario does not allow for the player to sneak through the enemy and reach the goal. (b) The scenario can be solved if the player is allowed to take down the enemies.

Figure 3.9b shows how allowing for the player to take down an enemy can modify the setting

of the scenario and create the correct state space to find a solution. In this case, a red dot in the path indicates the action of the player getting close enough to an enemy without being noticed and thus able to eliminate it. After doing this, the entrance to the room will not be blocked anymore and the player can reach the goal. Figure 3.10 illustrates this in a better way than the 2D model as by showing the time dimension. Here we can observe that while the entrance is blocked by the polyhedron created by the enemy FoV this is later cut off at the timestep where the take down is performed, and then the path can go over it and get to the goal.

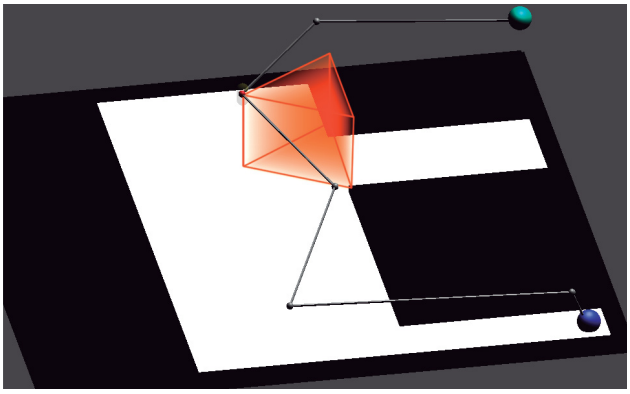


Figure 3.10 – Sample scenario where take down is needed to be solved. The configuration of the scenario does not allow for the player to sneak through the enemy and reach the goal. However, it can be solved if the player is allowed to take down the enemy.

To formalize this, we must first observe that the definition of the free space $\mathbf{S_{free}}$ needs to be changed, as the states inside the FoV of the enemies can change according to not only the enemy behaviour but also to the path up to certain timestep. That is, the take down can be performed at different timesteps, and thus the free space becomes dynamic as it can change at any time. To handle this, we can define the obstacles as a function of time:

$$\begin{aligned} O_{ko}(t) = O \cup \bigcup \{F_e \mid e \in E_a(t)\} \quad &\Rightarrow \\ E_a(t) = E - E_{ko}(t) \quad &\Rightarrow \\ e \in E_{ko}(t) \rightarrow \exists\, t_i :\ |P(t_i)e(t_i)| \leq r \ \wedge\ t_i \leq t \end{aligned} \tag{3.7}$$

This is, *the* ***obstacles in a take down problem*** $\mathbf{O_{ko}}$ *is a function over* ***time*** $\mathbf{t}$ *defined as the union of the* ***static obstacles*** $\mathbf{O}$ *(walls, objects, etc.) and the* ***fields of view*** $\mathbf{F_e}$ *of all the* ***enemies*** $\mathbf{e}$ *in the*

set $\mathbf{E_a(t)}$. We are giving to this set $\mathbf{E_a}$ the name of **enemies alive** which is also a function of time and complement of the set $\mathbf{E_{ko}(t)}$ **enemies taken down**. An enemy can only be in this set at time $\mathbf{t}$ if there exists a previous timestep $\mathbf{t_i}$ such that, the distance between the point on the solution path $\mathbf{P}$ and the position of the enemy at time $\mathbf{t_i}$ is less than or equal than $\mathbf{r}$. We call this distance the **knockout radius** since it describes a circular area around the enemy within which it can be taken down by the player. This radius helps to model different take down behaviours in stealth games; when it is close to zero, we require the player to be right next to the enemy to be able to perform the take down, this might simulate the action of strangling or striking the enemy on the head to put it to sleep, as seen in games like *Dishonored* (Bethesda Softworks, 2012). An $\mathbf{r = 0}$ simulates a scenario where take down cannot be performed as this would require the player to be in the exact same position as the enemy, which is usually not allowed to avoid unrealistic physical mechanics. On the other hand, a large $\mathbf{r}$ value might simulate the use of long-range weapons such as bows, knives, guns, etc., a mechanic seen in games like the *Assassin's Creed series* (Ubisoft, 2007).

With the obstacles in the scenario defined we can now apply the definition 3.5 to solve a stealth take down scenario:

$$
\begin{aligned}
&P_{ko}(t) : \mathbb{R} \rightarrow S_{free}(t) \Rightarrow \\
&P_{ko}(t) = \{p_i \mid p_i \in S_{free}(t) \ \wedge \ p_0 = A \ \wedge \ p_k = B \quad \forall\, 0 \leq t_i \leq k\}
\end{aligned} \tag{3.8}
$$

where

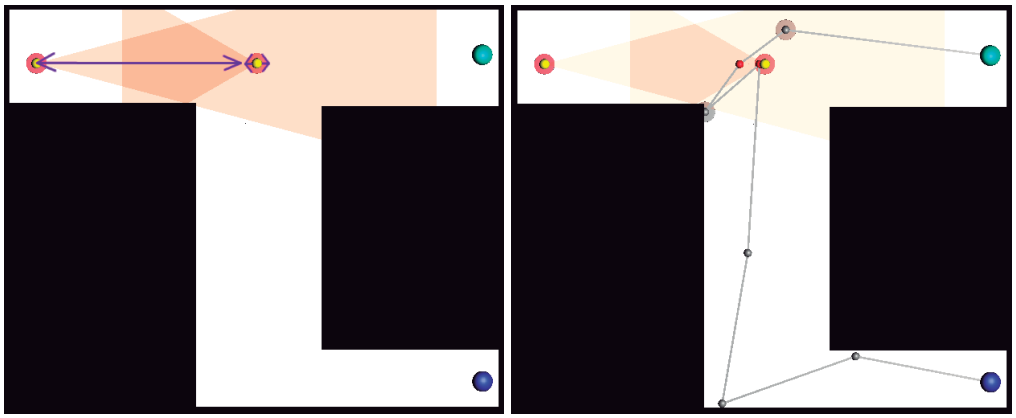
$$S_{free}(t) = S - O_{ko}(t)$$

That is, *the* ***Solution for a take down scenario*** $\mathbf{P_{ko}}$ *is a continuous function over* ***time*** $\mathbf{t}$*, such that, for any time value* $\mathbf{t_i}$ *it describes a polygonal path between the* ***starting point*** $\mathbf{A}$ *and the* ***goal*** $\mathbf{B}$ *in which all states of the path belong to the* ***free space*** $\mathbf{S_{free}}$*, defined as all the states in the state space that at time* $\mathbf{t}$ *do not belong to the obstacles in the scenario* $\mathbf{O_{ko}}$ *as defined in Equation 3.7.*

The previous definition can also be expanded by adding the condition $E_a(k) = \emptyset$ to the solution definition, this means that when reaching the goal state, all enemies must have been taken down. Adding this condition to the definition allows us to model scenarios where the player is forced not only to sneak to the goal but to also eliminate all enemies. This creates two different types of stealth scenarios, the first in which the goal is mainly to sneak from one place to another and the

second where the goal is to eliminate the enemies, a task which could be implemented in a game to simulate challenges with neutral and hostile NPCs.

3.2.5 Body Hiding Behaviour



(a) (b)

Figure 3.11 – Sample scenario with body hiding mechanic. (a) The initial configuration of the scenario does not allow for the player to sneak through both the enemies and reach the goal. (b) Solution to the scenario where body hiding is used.

The second stealth mechanic derives from the take down behaviour we explored in the previous section. Once taken down, the unconscious or dead body of the enemy may still exist, with other enemies observing the corpse interpreting it as a cause for alarm, implying and exposing the existence of the player. *Body hiding* is thus another frequent stealth mechanic, presenting an increased challenge to the player of having to dispose or hide the evidence of an enemy taken down. To compensate for this added challenge, games usually give the player the ability to carry an enemy body once eliminated and drop it in another place in the scenario, hiding it to avoid discovery by other, living enemies.

To formalize this problem, we must begin by analyzing the constraints involved in this problem. The first one requires we give the enemies the ability to notice the bodies of other characters once taken down. Second, we need a constraint on the solution so that no enemy should be in the FoV of another after being taken down. That is:

$$\forall\, 0 \leq t \leq k \quad \forall\, e \in E_{ko}(t) \quad e(t) \in S_{free}(t) \tag{3.9}$$

*For any **time value** $\mathbf{t}$ from the beginning ($\mathbf{t = 0}$), until the end of the solution ($\mathbf{t = k}$), the **position** $\mathbf{e(t)}$ of each enemy $\mathbf{e}$ in the set $\mathbf{E_{ko}(t)}$ must be inside the **free space** $\mathbf{S_{free}(t)}$ of that time. Where $\mathbf{E_{ko}(t)}$ is a function over **time** $\mathbf{t}$ that defines the set of all **enemies that have been taken down up to that time**.*

The previous definition includes the position of the enemy as one of its components. However, for the body hiding problem, this position is not simply defined by the predefined behaviour of the character, due to the added mechanic that allows for the player to carry the body of an enemy. Because of this, the position of the enemy must be now considered as a piecewise function:

$$e_{bh}(t) = \begin{cases} e(t) & e \in E_a(t) \\ P_{bh}(t) & t_{ko}(e) < t \leq t_d(e) \\ P_{bh}(t_d(e)) & t_d(e) < t \leq k \end{cases} \tag{3.10}$$

This means that the **enemy position in a body hiding problem** $\mathbf{e_{bh}(t)}$ will be given by three different functions:

- The normal behaviour of the enemy if it is still alive.

- The position will equal that of the player if t is between the **time of the taken down** $\mathbf{t_{ko}(e)}$ and the **time when the body is dropped** $\mathbf{t_d(e)}$. That is, the time interval in which the player is carrying the body.

- The position will be that of the player at the time of dropping the body $\mathbf{t_d(e)}$ for any value of $\mathbf{t}$ greater than that.

Note that this design assumes a body is not moved again after being hidden. Given the previous we can apply the same definition as we did for the take down problem in Equation 3.8 and add the condition in Equation 3.9 to get our solution function for the body hiding, this is:

$$P_{bh}(t) = \{p_t \mid p_t \in S_{free}(t) \ \wedge \ p_0 = A \ \wedge \ p_k = B \wedge \ \forall \, e \in E_{ko}(t) \, e(t) \in S_{free}(t)\} \, \forall \, 0 \leq t \leq k \tag{3.11}$$

This means that the ***solution for a body hiding problem*** $\mathbf{P_{bh}}$ *is a continuous function over* ***time*** $\mathbf{t}$*, that describes a polygonal path in the* ***free space*** $\mathbf{S_{free}}$ *from the* ***starting point*** $\mathbf{A}$ *to the* ***ending point*** $\mathbf{B}$ *such that, for any value of* $\mathbf{t}$*, the positions of all enemies taken down up to that time are in the* ***free space***.

To illustrate this definition, figure 3.11 shows an example scenario with the body hiding mechanic, with the starting and ending points shown in dark and light blue respectively. The level presents two enemies on the hallway at the top watching each other. The enemy on the left will walk up to the other enemy, then make an instant turn and go back to its original position faster than the first walk. The other enemy will keep its position and just instantly turn 180°every time the first enemy does it. This gives the player a small chance to kill one of the enemies when they have their back turned on each other, however, there is not enough time to get to the goal and once the other enemy turns he will see the body of its partner and notice the presence of the player. Thus, the player must take that time to hide the body instead of going to the goal.

Figure 3.11b shows a solution path for the scenario, the spots where the take downs are performed are shown with red nodes and the places where the bodies are dropped are shown as gray circular areas around the nodes.

This solution example shows a common tactic used in stealth games. Typically the player would wait until they have a reliable picture of the enemies' behaviour. The player can try to take down the closer enemy and hide somewhere out of reach of the other foe, waiting until it gets really close to where the player is hiding to perform the second take down and then go freely to the goal.

Figure 3.12 shows the extruded dimension on the solution for this scene, along with the polyhedra created by the FoVs of both enemies (denoted in purple and orange), as well as the solution path and the spots where a take down was performed (shown in red) and where bodies are dropped, shown with the silhouettes of the enemies in yellow.

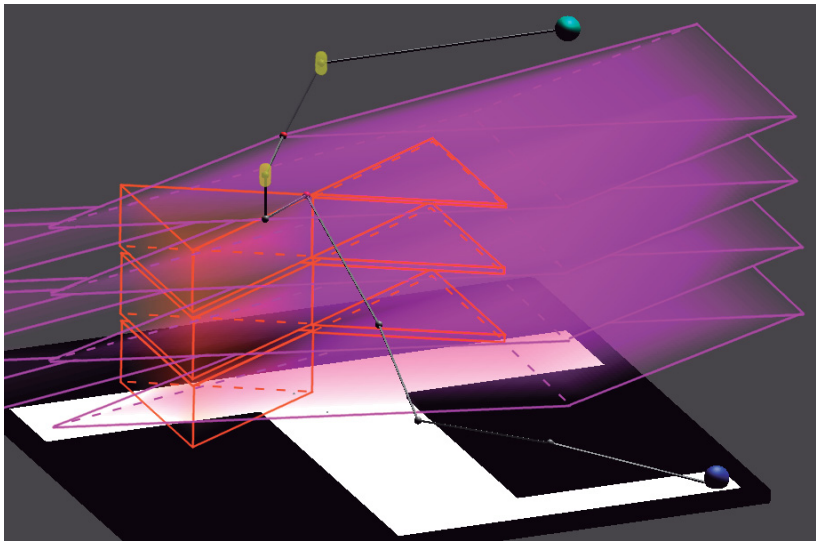


Figure 3.12 – Body hiding problem solution represented in the extruded time dimension. In this case, the points where the take downs are performed are shown as red circles and the points where the bodies are dropped are shown with the silhouette of the enemy.

3.3 Modelling Stealth Games

In this section, we will present the approach taken to translate the formalizations presented in the previous chapter into a computer model that can allow for us to test search algorithms to find a solution to the stealth challenges introduced by the take down and body hiding mechanic. This section is divided into four parts: We begin by presenting the tools and approaches used to model the elements involved in the stealth scenarios previously discussed. After this, we present the search algorithms used to find a solution that satisfies the correspondent definitions. Afterwards, we present the improvements made to the basic search algorithm in order to tackle the take down problem and finally, we do the same for the body hiding problem.

3.3.1 Level Definition

Previously we have presented the different elements involved in the stealth problem. However, in order to deal with them, we must first translate those definitions using tools that can help to model such definitions in a computer system. This enables use to explore different solution approaches

on those models.

Our models are based on elements available in the Unity 3D framework to define scenarios for games. Here we include the approaches taken to create the basic scenario, enemies, and the player, along with appropriate behaviours.

Scenario

In previous sections we have already presented examples of stealth scenarios. Figure 3.1 on page 22 illustrates how the scenario reflects a significant part of the state space. Specifically, a scenario includes a representation of the **free space** $\mathbf{S_{free}}$ and the **static obstacles** $\mathbf{O}$. To do this we present the entire space with a two-dimensional plane shown in white, representing the two-dimensional Euclidean portion of the state space $\mathbb{R}^2$, this plane is the basis for the scenarios. Unless otherwise specified, we assume dimensions of 50×50 game units.

The obstacles, on the other hand, are modeled by rectangular three dimensional objects denoted with black color. We only care about the X and Z coordinates of the rectangles to define the occupied space. These elements can be freely positioned in the scenario to simulate boundaries, holes or walls among the space through which the player nor the enemies can pass or see. Furthermore, these rectangular objects can overlap each other to simulate more complex polygonal obstacles. Then the **free space** $\mathbf{S_{free}}$ will be the Euclidean space among the plane that is not occupied by these polygonal obstacles.

Figure 3.13a shows an example model of a scenario. Here the white space represents the **free space** $\mathbf{S_{free}}$ and its bounds with a total boundary size of $\mathbf{50 \times 50}$ units. The black elements represent the **static obstacles** $\mathbf{O}$. A unit element is shown as reference in the bottom left corner in pink.

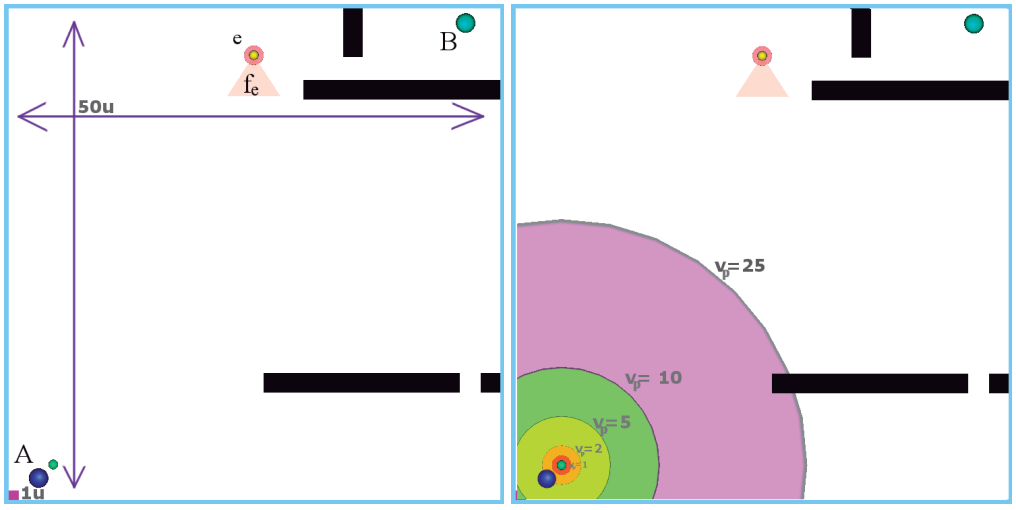
Every scenario needs to provide a **starting** $\mathbf{A}$ and **ending** $\mathbf{B}$ point which will become key points of the solutions. These will be modeled as points but illustrated as big circles of dark and light blue color.

Player

The player in our scenarios is modelled as a point but illustrated as a spherical green object . The size of it as well as its shape or boundaries are of no interest for the model.

Players have one important parameter in the model which is the **player speed** $\mathbf{v_p}$. This speed defines the boundaries for the solution search. That is, this speed defines the distance a player can travel in a single timestep, we measure this distance based in the unit used by Unity 3D to measure distances, which will be referred as **game units** and denoted by the symbol $\mathbf{u}$. Thus, the speed will be measured in $\frac{\mathbf{units}}{\mathbf{timestep}}$ and denoted as $\mathbf{u/f}$, where $\mathbf{f}$ represents the frame duration.

Figure 3.13b shows a representation of how this speed affects the search space in the scenario, the colorized areas around the player positions represent the area in which the search can locate the player in the next timestep based on a speed value of 1, 2, 5, 10 or $25^{u}/_{f}$. Conversely, these areas represent the locations the player can reach in 1, 2, 5, 10 or 25 timesteps at a speed of $\mathbf{1^{u}/_{f}}$.



(a) (b)

Figure 3.13 – (a) A sample scenario including the basic elements: The starting point A, the ending point B, an enemy e with its FoV f_e and the take down radius shown as a red zone around it. (b) A visualization of the valid movement distance for different speed values.

Enemies

Once the static elements have been defined in the scenario it is important to model an enemy. In our case, we are not concerned with the shape or size of the enemy body, and thus enemies will also be modeled as points represented by small yellow circles, as shown on top in figure 3.13a with symbol e.The spatial definition of an enemy, in contrast to that of the player includes both a

coordinate point and a unit vector to represent enemy orientation: $\mathbf{e} = (\mathbf{e_p}, \hat{\mathbf{e}_f})$. We refer to these components as the **enemy position** $\mathbf{e_p}$ and the **enemy forward** $\hat{\mathbf{e}_f}$. An enemy can thus be seen as a vector whose tail is located at $\mathbf{e_p}$ and direction given by $\hat{\mathbf{e}_f}$.

Following this, a feature that is important for the enemies in the scenario is the FoV. This is modelled after the cone of vision of a person or lens; in the 2D space we simplify it into a triangular shape denoted by two parameters, its **angle** θ, and its **distance** $\mathbf{d_F}$. The FoV is modelled as an isosceles triangle where its apex, is defined by the **enemy position** $\mathbf{e_p}$, while the **enemy forward** $\hat{\mathbf{e}_f}$ bisects the apex angle measuring $\mathbf{2}\theta$. That is, each of the equal sides of the FoV form an angle θ with vector $\hat{\mathbf{e}_f}$. In figure 3.13 the FoV is denoted as an orange triangle originated from the enemy position, in this case, the values for θ and $\mathbf{d_F}$ are set to $33°$ and 5 respectively.

The take down problem implies another enemy feature, the **knockout radius r**, which defines the area within which the player is able to take down the enemy. This area is modelled with a single decimal value giving the radius of a circular area centered in the enemy position. Figure 3.13 shows this as a red area around the enemy position, for this example the value of the radius is set to 1.

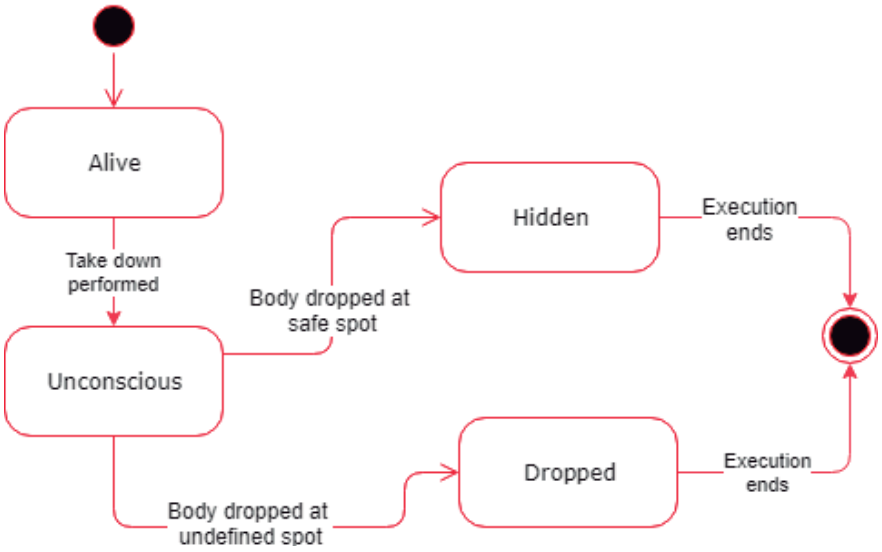


Figure 3.14 – Enemy states and their transitions.

The modelled enemies also include an **enemy state** $\mathbf{e_s}$ parameter, which indicates if an enemy is still alive, has been taken down or is hidden during the execution of the solution search. The set of all the enemy states will be represented in the present work as $\mathbf{S_e}$. Figure 3.14 illustrates

the different transitions between the states which also affect the enemy behaviour, as shown in table 3.1:

- **Alive:** This is the initial state for the enemies when the scenario is executed, here the enemies are able to move and patrol following their default behavior, and although their bodies can be seen, it is not important for the body hiding mechanic.
- **Unconscious:** An enemy enters this state at the moment a take down is performed on them. Once in this state, they cannot move or monitor any area according to normal behaviour. The FoV is thus of no importance, and a take down cannot be performed again on an enemy in this state. For the body hiding mechanic, unconscious enemies being carried are in this state, and so they can still change their position while being carried and bodies can be seen by other enemies.
- **Dropped:** To enter this state a player must drop a carried body in any place that is not considered a safe spot (this will be defined in the following sections). In this case, the enemy cannot change its position any more; its FoV has no meaning and no take down can be performed on it. However, its body can still be noticed by the enemies alive.
- **Hidden:** The last state for the enemies is attained when a player drops the body in a place considered to be a safe spot. In such case its position cannot change and neither its FoV nor its body has any further interest in the scenario.

	Move	**Sight**	**Take down**	**Be seen**
Alive	Yes	Yes	Yes	-
Unconscious	By player	No	No	Yes
Dropped	No	No	No	Yes
Hidden	No	No	No	No

Table 3.1 – Behaviours of the enemy for each state.

Movement

In the previous section we referred to the enemy behaviours as well as the mechanics related with it, such as movement and rotation. These mechanics have to be modelled in order to tackle the

dynamic stealth problem. There are three types of motion mechanics inside our model, these are: movement, rotation and waiting. To model these mechanics, we use the waypoint system presented in Borodovski's work [2].

The **movement** is the motion performed for an enemy to go from one point to another, to model this we define a **waypoint** $\mathbf{W_m}$=$(\mathbf{x}, \mathbf{z}, \mathbf{v_w})$ whose position in the scenario indicates the target position for the motion, it includes a **speed parameter** $\mathbf{v_m}$ similar to the player model to indicate how many game units per timestep the enemy will move when targeting the waypoint position. This model only defines movement in straight-line, however, it is possible to create paths with more complex trajectories using multiple waypoints in sequence.

The **rotation** motion is modeled in a similar way as the movement. However, this is not defined by coordinates as the rotation will be performed in the place where the enemy is and the rotation will be performed with respect to its own axis. Instead, we use a **rotation waypoint** $\mathbf{W_r}= (\hat{\mathbf{w}}, \mathbf{v_r})$ which defines the **direction vector** $\hat{\mathbf{w}}$ which will be the direction the **forward vector** $\hat{\mathbf{e_f}}$ of the enemy will point to after the rotation. The second parameter of the rotation waypoint is the **rotation speed** $\mathbf{v_r}$ which defines the number of degrees an enemy will rotate in a single timestep. Since our scenarios are modelled using the *Unity 3D* framework, this angle will be calculated using its Vector libraries, thus, it will never be greater than $180°$

Finally, the **waiting** mechanic is used to model time intervals in which the enemy stays still, this third type of waypoint $\mathbf{W_w}= \mathbf{t_w}$ which is defined by a single parameter corresponding to the **waiting timesteps** $\mathbf{t_w}$.

This waypoint system is expressed in the scenarios as non-visible objects placed in the space at the locations where the motion happens. This objects also define a **next waypoint** parameter that allows chaining different motions to model more complex paths. For the present work, we will indicate the motions of the enemies using arrows in the same way as we have done in previous figures.

Additional Elements

Although the previously mentioned elements are enough to define most of the scenarios we aim to analyze, we also added some elements that are not strictly related with the game levels but instead are used to analyze some of the proposed approaches to solving the stealth problem or to abstract

features of three-dimensional scenes into a 2D model in order to study their effect in the models.

Firstly, we present two types of obstacles that although similar to the static obstacles presented before, they include different features than a simple wall-like obstacle. Then we will present two abstract elements related to the body hiding mechanic.

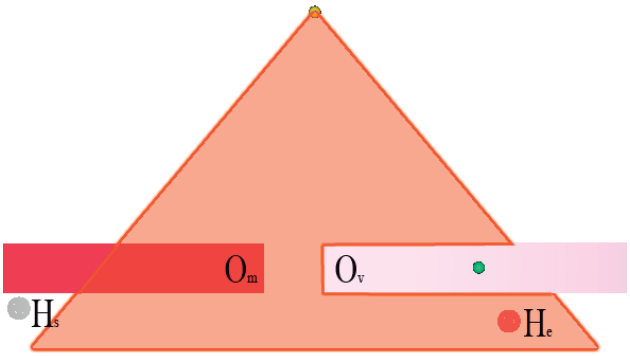


Figure 3.15 – Added elements on an example scene. The pink rectangle shows a vision obstacle. A motion obstacle is shown as a red rectangle, while the safe and exposed spots are shown as gray and red circles respectively.

Movement-only Obstacles These elements present similar characteristics to that of the static obstacles in the scenarios, however, they differ in that a movement-only obstacle does not occlude the FoV of enemies. Thus, the player can be noticed by an enemy trying to hide behind them but characters cannot travel through them, and therefore these obstacles are not inside the **free space** $\mathbf{S_{free}}$. This type of obstacle is useful to model elements like tables, cages, and other smaller objects that offer imperfect occlusion. Figure 3.15 shows this kind of obstacle as a red rectangle which as illustrated, does not affect the FoV of enemies. In following sections, we will refer to this type of objects simply as **movement obstacles** $\mathbf{O_m}$

Vision-only Obstacles In contrast with the movement obstacles O_m which mainly hinders the player, the vision-only obstacles, which we will refer to as **vision obstacles** $\mathbf{O_v}$ only affects the enemies by acting similar to a static obstacle, inhibiting their sight in the space occupied by these elements but allowing for the characters to travel through them. This is $\mathbf{O_v} \in \mathbf{S_{free}}$. However,

their behaviour is a little more complex as the enemies can look through them. Then, an enemy will notice a player standing behind these obstacles but it will be unable to notice the player if its position is inside the area of the obstacle. This is illustrated in figure 3.15 as a pink rectangle where the player is located. This kind of obstacles are useful to model elements of modern 3D stealth games such as pipelines, wires, suspended platforms, tunnels, among others that players can use to travel through the scenarios without being noticed.

Safe and Exposed Spots These elements are specific for body hiding scenarios where one of the main concerns are the places where body are dropped. In the following sections, we will explore the different approaches for defining these places. The **safe spots $\mathbf{H_s}$** define areas where a body can be hidden from enemies; in these zones bodies are not noticed by enemies even if they are inside their FoV. In contrast, the **exposed spots $\mathbf{H_e}$** define zones where hiding a body is not possible as it will be noticed at some point by an enemy. Figure 3.15 shows both elements: a grey circular area for the **safe spot $\mathbf{H_s}$**, and a red circular area for the **exposed spot $\mathbf{H_e}$**.

The following Table 3.2 summarizes the characteristics of each of the additional elements defined above.

	Move through	**See used space**	**See through**	**Hides bodies**
Motion-only obstacle	No	Yes	Yes	No
Vision-only obstacle	Yes	No	Yes	No
Safe spot	Yes	Yes	Yes	Yes
Exposed spot	Yes	Yes	Yes	No

Table 3.2 – Characteristics of the additional elements in the scenes.

3.4 Solution Search with RRT

In order to find a solution for a stealth puzzle, the player is forced to explore the area and create a movement trajectory around the **free space $\mathbf{S_{free}}$** to achieve the goal, or for taking down the enemies in the level while remaining unnoticed.

While several options for path planning can be found in literature, we decided to explore use of a *Rapidly-exploring Random Tree (RRT)*. This option has advantages in being an incremental and sample-based algorithm, with a stochastic behavior that enables rapid generation of multiple solution paths, arguably better approximating potential human behavior. RRT has also been previously used to explore both combat and stealth games [1]

In this section we will describe important details about the RRT implementation used to solve stealth scenarios. This includes the state space stored in each node, the node selection, connectivity validation as well as some improvements over them.

3.4.1 State Space

In order to solve the stealth scenarios previously defined using an RRT, it is important for each of the nodes included in the data structure to provide the values sampled from a whole search space. This search space has to span across all the possible configurations in the scenarios to be explored. To better explain this we can observe what happens to the state space as it gets extended to different scenarios.

The most basic case for exploring a scenario with RRT would be solving a pathfinding problem in a static 2D scenario. In this case, the search space would only include the coordinates of the nodes as they get sampled from the coordinate space, thus our **search space** $\mathbf{\Sigma}= \mathbb{R}^{\mathbf{2}}$ and a sampled RRT node is represented by $\langle \mathbf{x}, \mathbf{z} \rangle$.

When considering to expand the RRT for a dynamic scenario as defined in subsection 3.2.3 we are concerned with the changes in the scenario over time, thus our search space must be extended to add the extruded time dimension which can only contain positive real values $\mathbf{\Sigma} = \mathbb{R}^{\mathbf{2}} \times \mathbb{R}^{+}$, therefore, a sampled node will be defined by $\langle \mathbf{x}, \mathbf{z}, \mathbf{t} \rangle$.

Up to this point we have dealt with the stealth problem, however, by adding the take down mechanic we must again expand the search space to add the status of the enemy as we must consider the sampling to include the possibility of performing or not a take down. To do this, we must generate an **enemy state representation** $\mathbf{E_d}$ which provides the information of the enemies in the scenario for a single RRT state. This representation must include the **state of the enemy** $\mathbf{e_s} \in \mathbf{S_e}$ taken from the **enemy state domain** $\mathbf{S_e}$ as defined in Table 3.1 on page 44. Thus, for a scenario with $\mathbf{g}$ enemies, the **enemy state representation** will be $\mathbf{E_d} = \langle \mathbf{e_{s1}}, \mathbf{e_{s2}}, ..., \mathbf{e_{sg}} \rangle \in \mathbf{S_e}^{\mathbf{g}}$.

Then, a sampled node for the take down problem will be represented by:

$$\langle x, z, t, k_p, E_d \rangle \in \mathbb{R}^2 \times \mathbb{R}^+ \times \{0, 1\} \times {S_e}^g$$

Where $\mathbf{k_p}$ is a boolean variable that indicates if at time $\mathbf{t}$, a take down action was performed.

Given the previous search space, we can expand it for a body hiding scenario, first by expanding the **enemy state representation $\mathbf{E_d}$** to include its **body positions** for keeping track of where the bodies are being carried or dropped. Thus the enemy representation will be defined by $\langle \mathbf{e_s}, \mathbf{x}, \mathbf{z} \rangle \in \mathbf{S_e} \times \mathbb{R}^{\mathbf{2}}$. Therefore, a node for a body hiding RRT is represented by

$$\langle x, z, t, k_p, b_d, E_d \rangle \in \mathbb{R}^2 \times \mathbb{R}^+ \times \{0, 1\} \times \{0, 1\} (\times S_e \times \mathbb{R}^2)^g \tag{3.12}$$

Where $\mathbf{b_d}$ is a boolean variable that indicates if a body drop is performed in a given node. This representation is the one that will be used for all the RRT searches in the following sections.

3.4.2 RRT Algorithm

The RRT search used for solving the stealth problem is outlined in algorithm 2, its creation and expansion involves several steps that will be detailed in this section, we will follow the algorithm to explain how the structure is created, the selection of the nodes, their validation and the connectivity check during execution.

Data Structure

An important aspect to consider when building a solution using RRT is that its performance will be affected by the search executed when inserting a new node, as the RRT structure will need to select the closest nodes among all the collection. Following the previous implementation of RRT in games by Bodorovski [2] we stored the RRT in a KDTree structure [48], which offers a convenient way to store and search among a multi dimensional Euclidean space. Specifically, we use the implementation of KDTree by Levy Heckel, and Alvarez [49].

RRT nodes are included in a multi-dimensional space as defined by equation 3.12. Inside the KDTree, however, they are only stored in a three-dimensional space using its x and z coordinates, as well as the time value t. This is done in consideration that the main purpose will be to search for

Algorithm 2 Stealth RRT

Require: start A and end $B \in$ scenario S. Set of enemies E, iterations m_{max}, time bound k_{max} and player speed v_p.

```
1:  procedure ComputeStealthPath(S, A, B, E, m_max, k_max, v_p)
2:      RRT ← A                                          // Initialize RRT structure
3:
4:      for i = 0 to m_max do
5:          x ← selectNode(A,S,k_max,v_p)                // Node Selection
6:          parent ← findClosestNode(RRT, x)             // Parent Selection.
7:
8:          if !validateNode(x, parent) then             // Node Validation.
9:              continue
10:         end if
11:
12:         if obstacleCollision(x, parent) then         // Collision check.
13:             continue
14:         end if
15:
16:         if enemyCollision(x, parent, E) then
17:             continue
18:         end if
19:
20:         x.parent ← parent                            // Connects node to the tree
21:         RRT.add(x)
22:                                                      // Body hiding can be performed here
23:         if goalIsReachable(x, B) then                // Goal is reachable.
24:             B.parent ← x                             // Connects coal
25:             break
26:         end if
27:     end for
28: end procedure
```

the nearest neighbors and to calculate the distance between two nodes, during which we are only interested in such Euclidean coordinates as will be explained in following sections.

Node Selection

The RRT is built iteratively by adding one node at a time. In order to perform an efficient search we aim to select the nodes that will be added by uniformly sampling the whole search space. However, as defined in equation 3.12 the search space includes parameters that should define its value based on several conditions or even based in the previous nodes of the tree; for example, the **take down variable** $\mathbf{k_p}$ can only have the value of true if the player is close enough to an enemy and is not being seen by anyone. Uniformly sampling the values for this variable will thus result in an excessive number of cases where the constraints of the problem will be broken. Another example is that of the **body positions** which cannot be defined only by sampling the possible values, as they are conditioned both on being part of a branch where a take down on the enemy has been performed, on the fact that the body can only be dropped at a position the player has also been.

Our sampling will thus be based on the dimensions we are interested in exploring to form a path: the coordinate and the time dimensions. During the node selection shown at line 5 in Algorithm 3 we proceed by randomly sampling the x and z coordinates of the node. The time is also randomly sampled, but upper bounded by the **maximum timesteps value** $\mathbf{k_{max}}$ (defined in the input of the algorithm), and lower bounded by *the time it would take for the player to get from the starting point* $\mathbf{A}$ *to the sampled coordinates by moving at maximum speed* $\mathbf{v_p}$.

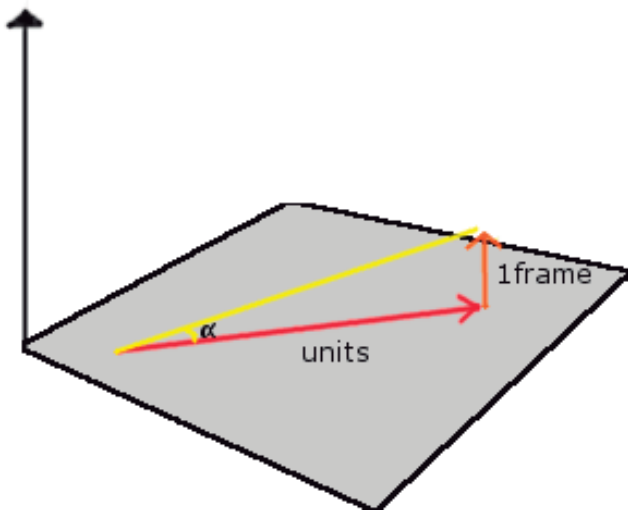


Figure 3.16 – Player speed representation in the extruded coordinate space.

To determine the lower bound we calculate the angle generated by the **maximal player speed** $\mathbf{v_p}$. As this is given in units per timestep, and the third dimension represents the time, the angle can be computed from a vector in the XZ plane with a magnitude $\mathbf{v_p}$ and the unit vector in the third axis, as shown in figure 3.16. Thus we can calculate this angle as $\alpha = \mathbf{arctan}(\frac{\mathbf{1}}{\mathbf{v_p}})$. To calculate the minimum time to get from A to the point X we use $\mathbf{t_{min}} = \mathbf{d}(\mathbf{A}, \mathbf{X})\mathbf{tan}(\alpha)$. The full procedure to sample nodes is shown in algorithm 3.

Algorithm 3 Node Selection

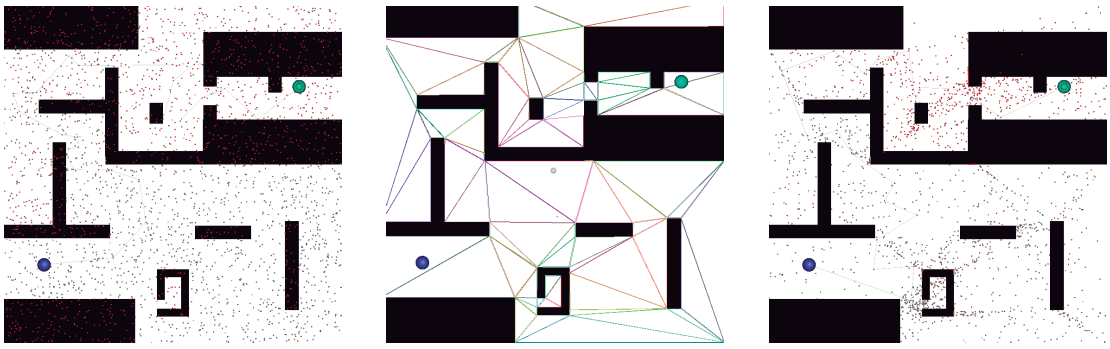
Require: start point $A \in$ scenario S, time bound k_{max} and player speed v_p

```
tanα ← 1/v_p

procedure SELECTNODE(A, S, k_max, v_p)
    node.x ← Random(S.x_min, S.x_max)
    node.z ← Random(S.z_min, S.z_max)
    t_min ← d(A, node) * tanα
    node.t ← Random(t_min, k_max)
    return node
end procedure
```

This sampling method selects points from the full state space $\mathbf{S}$. This space, however, includes obstacles that will lead to cases where collisions that invalidate the sampled state. Figure 3.17a highlights this problem showing that many of the sampled nodes are taken from the spaces occupied by obstacles which will result in invalid cases. To reduce this problem, we change the naive sampling to be performed only over the free space $\mathbf{S_{free}}$, to do this it is necessary to reduce the irregular polygonal shape of the free space in the scenario to more simple figures which will help during sampling.

Triangulation In order to sample nodes from the free space exclusively we first decompose the scenario into smaller polygons. As all the scenarios are restricted to include only polygonal obstacles, this will result in the free space having also a polygonal shape, which can be divided into triangles that together span the entire space. The triangulation method is a "brute force" approach, the same as the one used by Borodovski [2]. Summarizing this previous work, it first merges all the overlapped obstacles into single polygonal shapes to create a set of vertices that will be used



(a) (b) (c)

Figure 3.17 – Difference in node sampling. (a) shows the sampling without triangulation. (b) shows the triangulation for the level and (c) shows the sampling using such triangulation. The red dots represent the nodes that result in invalid cases after validation while the grey dots represent valid cases.

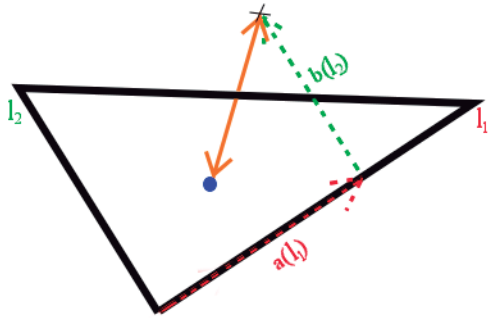


Figure 3.18 – Random point sampling inside a triangle.

for the triangulation; afterwards, by iterating over all pairs of vertices it generates a set of internal "diagonals" that do not intersect with any obstacle (or other diagonal), and these lines are then grouped to form the triangles. Finally, the generated triangles are evaluated to generate a Delaunay [50] triangulation to improve triangle shapes. Figure 3.17b shows an example of the triangulation applied to a basic scenario.

Once the triangulation is obtained we can consider this list of triangles as our **free space** $\mathbf{S_{free}}$. We sample it by simply by changing the random naive sampling of the x and z coordinates to first select a random triangle from the list and then pick a random point inside this triangle. To do this random sampling inside a triangle we select two of the lines that forms the triangle l_1 and l_2,

considering them as vectors, we select two random values $a, b \in [0, 1]$, then, using the vertex that connects l_1 and l_2 as the origin we sample the point $x = al_1 + bl_2$ this point will always be in the quadrilateral formed by the selected triangle and its symmetric to l_3, then if the point is outside the selected one we sample the symmetric point with respect to the middle point of l_3. This process is depicted in figure 3.18. The process of checking if a point is inside a triangle is done by checking the number of times a ray generated from the point intersects the triangle edges. Following the *Jordan Curve Theorem*, if it is an odd number then the point is inside it [51].

Figure 3.17c shows a comparison of sampling random points in a scenario with and without triangulation. As we can observe, the cases where points will be discarded for being on the space occupied by an obstacle are reduced considerably. Point distribution, however, is also altered, since triangle area can vary. Algorithm 4 presents the pseudo-code of the random point selection using triangulation.

Algorithm 4 Node Selection with Triangulation

Require: Start point and set of obstacles $A, O \in$ scenario S, time bound k_{max} and player speed v_p

$Triang \leftarrow delaunayTriang(S, O)$
$tan\alpha \leftarrow 1/v_p$

procedure SELECTNODE($A, Triang, k_{max}$)
    $randT \leftarrow Random(Triang)$ // Picks a random triangle
    $a \leftarrow Random(0, 1)$
    $b \leftarrow Random(0, 1)$
    $node \leftarrow a * randT.l_1 + b.randT.l_2 + randT.vertex(l_1, l_2)$
    if $!isInside(node, randT)$ **then**
        $node \leftarrow symmetric(node, randT.l_3)$
    end if
    $t_{min} \leftarrow d(A, node) * tan\alpha$
    $node.t \leftarrow Random(t_{min}, k_{max})$
    return node
end procedure

Parent Selection

After sampling a node, the node must be added to the RRT tree by creating a single connection to another node already in the structure. Thus, it is necessary to select from all the previous node the one that will be connected to the current one; in other words, we must select the parent of our sampled node.

A simple RRT would select the closest node by coordinates. However, when adding the time dimension it is important to consider the constraint that no node can have a time value smaller or equal than that of the parent as this would translate into the player going back in time or teleporting. This means that some times, the geographically closest node might not be valid to make a connection to the sampled node.

To handle the time constraint, we use the KDTree structure to search for the **c** closest nodes in the tree. We then iterate over this subset discarding those with a time value higher than the sampled node, taking the first node that follows the constraint as the possible parent. Algorithm 5 shows the pseudo code for the parent selection.

Algorithm 5 Parent Node Selection

Require: $node, RRT, c$

```
1: procedure FINDCLOSESTNODE(RRT, node, c)
2:     candidates[ ] ← RRT.knn(c)                    // Picks c closer nodes.
3:
4:     for each x in candidates do
5:         if x.t < node.t then
6:             return x
7:         end if
8:     end for
9: end procedure
```

Node Validation

Although we ensure sampled nodes exist in the future relative to their parent, not all nodes lead to valid transitions that follow the constraints imposed by the stealth problem. Thus, after sampling a node it is important to verify if it should be added and connected to the nodes in the RRT according to the player speed, the obstacles and enemy positions.

In algorithm 2 (on page 50) the node validation is shown from lines 8 to 16, and involves 3 checks. The first check involves validating the parent, which involves first verifying that the node has not been added previously; if so, we return the previously stored node instead of creating a new one, otherwise we actually create the node.

Algorithm 6 Node Validation

Require: $node, parent, RRT$, player speed v_p
1: $\alpha \leftarrow tan^{-1}(1/v_p)$
2: **procedure** VALIDATENODE($node, parent, RRT$)
3: **if** $node \in RRT$ **then return false**
4: **end if**
5: $vector \leftarrow node - parent$
6: $2DVector \leftarrow vector$
7: $2DVector.t \leftarrow 0$
8: $angle \leftarrow angle(vector, 2DVector)$
9: **if** $angle < \alpha$ **then return false**
10: **end if**
11: **return true**
12: **end procedure**

In this same check we ensure that the player speed is not surpassed. To do this we calculate the angle generated by the vector that goes from the parent to the sampled node and the XZ plane, if the angle is less than α (calculated in Figure 3.16) then the connection is not valid. Algorithm 6 shows the pseudocode for the node validation which corresponds with line 8 of the general RRT algorithm on page 50.

Obstacle Collision

Once the node has been verified to create a path with its parent that follows the time constraints we obtain the motion path the player will be performing in the time interval that goes from the parent node's t value to the sampled t value. Although we have taken care to both coordinates are not inside the static obstacle space, this does not guarantee that the path between those points satisfies the same property. Figure 3.19 shows an example of this, where the closest node might be a valid coordinate but is on the other side of a wall. To avoid this, it is important to validate the

path segments generated by linearly interpolating the coordinates of two nodes and verifying lack of any collision.

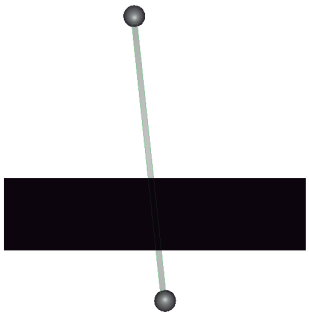


Figure 3.19 – Example of obstacle collision when sampling nodes that are in S_{free}

We divide this task based on the type of obstacles. Remembering the scenario definitions we have two types of them: **static** which include all the polygonal objects representing holes, walls and boundaries, and **dynamic** which basically are the enemies in the level.

Static obstacles by definition never change their position: if extruded into the time dimension they will cover the same space at any time value. Checking for collisions against them in the extruded dimensions can thus be performed by validating the same on a 2D space. This can be done by creating a line segment between both points with the same, fixed time value, and then checking the intersection of this segment against each of the edges of the polygonal object to detect a collision.

Algorithm 7 Obstacle Collition

Require: $node, parent, obstacleLayers,$

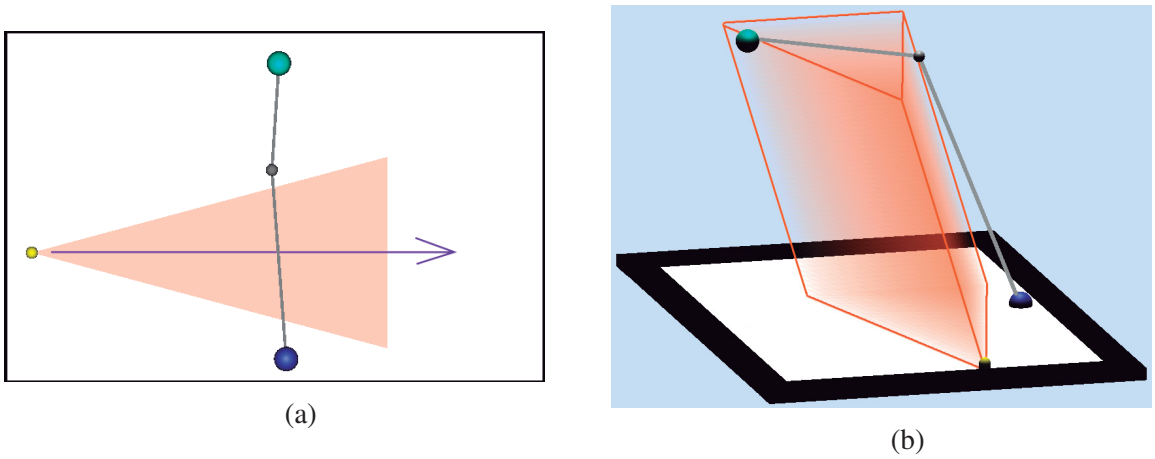
1: **procedure** OBSTACLECOLLISION($node, parent$)
2: $start \leftarrow Vector(parent.x, 0, parent.z)$ // Projects the points on the XZ plane
3: $end \leftarrow Vector(node.x, 0, node.z)$
4: $isHit \leftarrow Linecast(start, end, obstacleLayers)$
5: **return** $isHit$
6: **end procedure**

We used the tools provided by Unity *3D* to detect this kind of collisions. We set all our obstacles into a special layer of game objects in the *Unity 3D* editor. In this way we can use the Unity

linecasting feature to throw a ray from one node to another, and using a filter to only considerate the objects in a fixed layer we are able to detect the collisions. Algorithm 7 shows the pseudocode for the static obstacle collision.

Enemy Collision

While collision detection with static obstacles has a straightforward solution using the predefined *Unity 3D* tools, collisions with dynamic objects and enemies require a more complex analysis. These collisions cannot be reduced to a 2D plane due to the fact that the space occupied by dynamic objects is not the same at different time values.



(a) (b)

Figure 3.20 – Problem detecting an enemy collision in 2D. (a) shows how a solution path for a dynamic scenario might be incorrectly detected as invalid due to an intersection with the enemy FoV when seen in a 2D coordinate space. (b) Shows the same scenario and solution path seen in an extruded 3D space where the vertical axis represents time.

To illustrate this, figure 3.20 shows a comparison on evaluating collisions in a two-dimensional space and doing the same for an extruded three-dimensional space considering the time dimension. Figure 3.20a shows a scenario with one dynamic enemy (yellow circle on the left side) following the movement indicated by the purple arrow. Here, the gray line at the center of the images shows a solution path going from the starting point shown in dark blue at the bottom of the image to the ending point shown in light blue at the top of the image. If we were to evaluate if the path intersects at some point with the enemy FoV considering **only** the information of a 2D space it

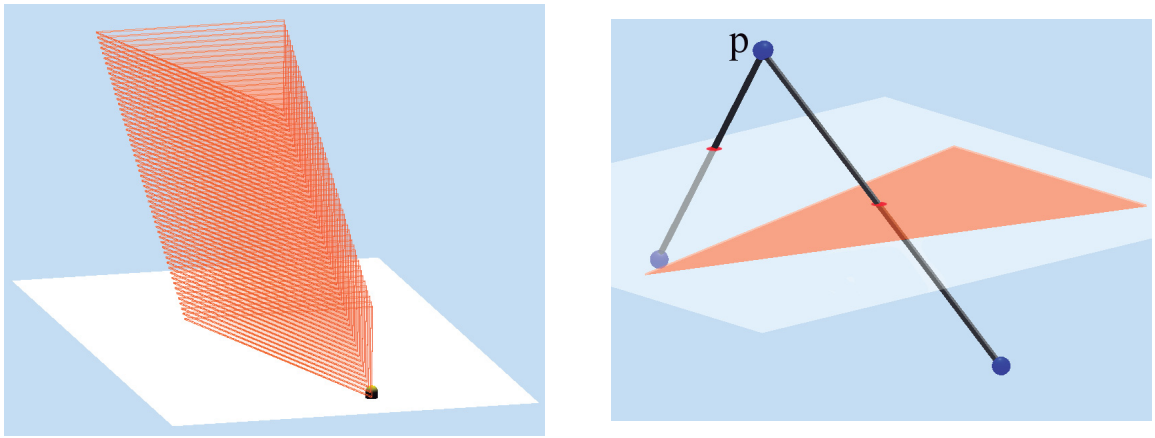
would be complicated as the only information we have are the starting and ending point of the enemy and the player trajectories, there is no information about how the space occupied by the dynamic obstacle (the FoV of the enemy) changes as it moves. Furthermore, just by looking at the illustration one could assume that a collision exists since the path runs over a big portion of the FoV.

Figure 3.20b however, shows the same scenario and solution when incorporating the time dimension. Here the vertical axis shows the time value and it is possible now to map the position of the FoV over time, therefore, the dynamic obstacle can be treated as a static obstacle in this space represented as a three-dimensional volume created by the movement of the FoV through space. In this representation we can see that the player path actually does not intersect with the FoV solid.

Although a three dimensional space provides a better way of detecting collisions it comes with the drawback of incorporating volumes which have to be calculated from the 2D shape that generates them and the trajectory they follow. This itself represents a complex task, and becomes even more complex when adding rotation motions that generate curves and helicoidal shapes like the one shown in figure 3.7 on page 32. Computing these volumes, as well as the intersection with the path becomes a problem on its own which will not be addressed in this work.

To avoid computing complex shapes and their intersections, we use a discretized version of this problem, in which the volumes can be decomposed into planar *time slices* of the same shape as the original FoV but with different coordinates. For any value of time, there exists a *time slice* for each enemy representing the position of the FoV at that time. If we were to calculate all the *time slices* for a single enemy and place them right on top of the *slice* for the previous time value, we would get an approximation of the volume created by the movement. Figure 3.21 illustrates the reconstruction of the FoV volume for the scenario of figure 3.20b.

With this approximation, it is possible to reduce the task of detecting a collision with a FoV volume to simply detecting an intersection between 2D elements. To do this we perform a validation, checking each *time slice* between an interval. Since all the *time slices* have the same shape as the FoV, detecting a collision is the same as detecting if a line segment in three dimensions intersects with a triangular area on the XZ plane. Since the path is a function that maps all the time values to coordinates in space, and the FoV is in a plane orthogonal to the time axis, the intersection point between the plane and the line is that of the coordinate mapped by the plane at the time value corresponding to the *time slice*. Furthermore, since the enemy collision check is performed



(a) (b)

Figure 3.21 – Decomposition of the FoV volume in time slices. (a) Shows how the union of the time slices approximates the volume generated y the FoV. (b) Shows the intersection of two different paths towards point P and their intersection with the time slice plane.

when we connect two RRT nodes, the point at the time value corresponding to the *time slice* can be calculated by interpolating the coordinates of the two nodes. To verify if a point is inside the FoV triangle we used the same method of counting the number of intersections with edges, as described in the triangulation section on page 52.

Since the enemy collision is based on *time slices* it is important to notice that the interval between them is important as it will define the granularity of the detection. To illustrate this, figure 3.22 shows two collision detections performed on the same path segment. In the example the algorithm aims to join node b to a, a movement performed in the interval from time t_1 to t_4. Figure 3.22a shows the position of the enemy at the beginning and end of the time interval. In this case, if we were to select slices too big we might be missing important data along the movement. That is, if we were to check only the beginning and end of the movement we would determine that no collision happens along the path as it does not intersect with either of both slices. However, if we choose a smaller interval between the slices, such as the one shown in figure 3.22b we would determine that actually there is a collision during the movement since the path intersects both slices at t_2 and t_3. While one might simply decide to select the smallest possible granularity, it is important to consider that these checks will be performed at each iteration of the RRT algorithm, and the number of validations is inversely proportional to the granularity selected.

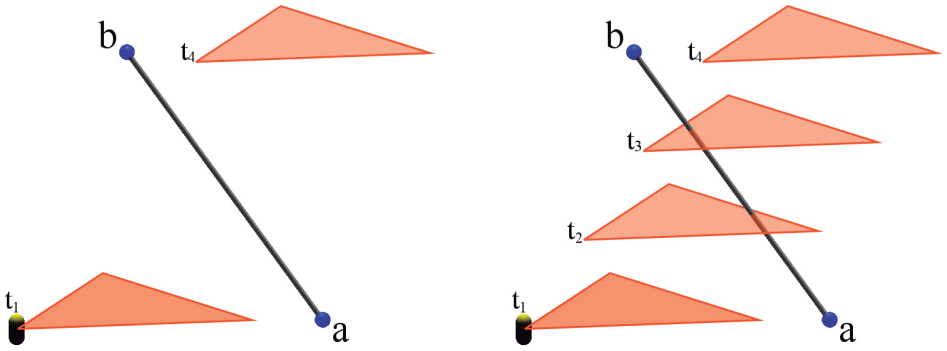
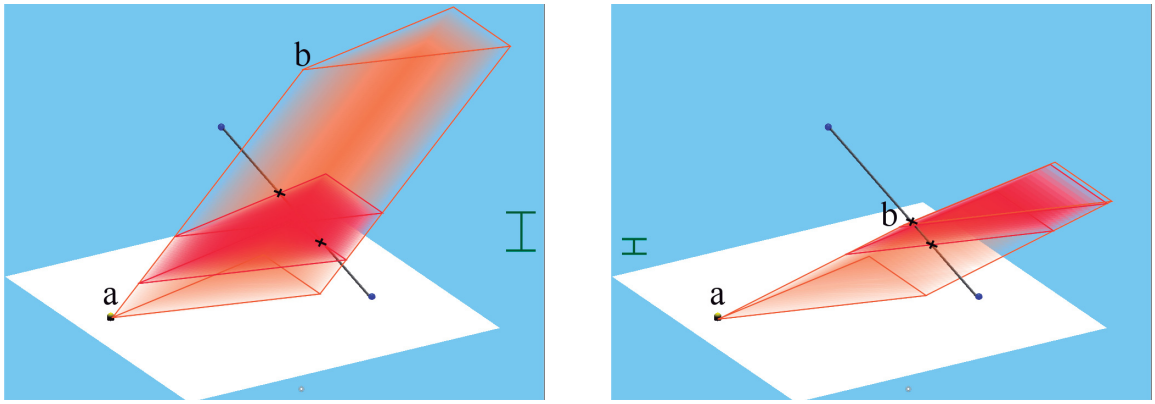


Figure 3.22 – Different granularity for enemy collision detection. With higher speeds, a granularity too big (a) might overlook collisions on the path while small granularity (b) involves a more accurate detection but at the cost of an increased number of validations.

Another important consideration for enemy collision detection is the speed of the movements. To illustrate this let us consider two enemies moving between the same two points, the second one moving way faster than the first one. If the same path intersects with both enemies during the motion, the interval in which the path points are inside the FoV of the enemies will be different. Figure 3.23 illustrates this. Here the movement of the enemies goes from the XZ coordinates of point a to b, but because the movement of one enemy is faster than the other the volumes generated by the motion of the FoV will be different for both, with the slower one generating a volume that spans for a longer interval in the vertical axis (shown in figure 3.23a) compared with the faster one (shown in figure 3.23b). This leads to a smaller interval time in which the path crosses the volume, which can be appreciated as the red interval shown in the figure or the green ruler shown between the two volumes.

When choosing the granularity for the enemy collision detection it is thus important to follow the rule that the granularity should be inversely proportional to the speed of the elements in the scenario, while keeping in mind that the number of operations in the RRT algorithm is inversely proportional to the granularity. Algorithm 8 presents the pseudocode of the enemy collision for our RRT. We added two validations before actually performing the collision check by first verifying that the enemy that the collision is checked against is still alive, otherwise no collision can happen against it. The second one avoids some extra computations by first verifying that the point of the



(a) (b)

Figure 3.23 – Comparison of collisions with enemy FoV at different speeds. The green polygon is generated by the enemy FoV when moving from point a to b. The red section is the interval in which the enemy path intersects the polygon. (a) shows an enemy that travels slower than (b), thus, generating a greater intersection interval.

path at a given time is not farther from the enemy than the FoV distance, otherwise, it would mean that no intersection can happen between the FoV and the player.

One final validation is performed when an intersection with a FoV is detected. Since our algorithm does not perform any computation of the occlusion of the FoV by other elements in the scenario, it could be the case that the player is inside a FoV but the line of sight between the enemy and the player is blocked by an obstacle as exemplified in figure 3.24. To handle this cases after detecting an enemy collision we perform an obstacle collision detection as defined in section 3.4.2 between the enemy position at the time the intersection was found and the coordinate mapped by the path for the same time value. Since this obstacle collision detection is performed between points using a linecast, if a collision is detected then no intersection with the enemy is produced by the algorithm.

Ending Condition

The RRT algorithm iterates repeating the procedures described above until an ending condition is fulfilled. A first ending condition is reaching the goal: the algorithm stops and gathers the solution path by traversing backwards from the ending node and going through all the parent nodes until

Algorithm 8 Enemy Collision

Require: $node, parent$ set of enemies E, $interval$

```
1:  procedure ENEMYCOLLISION(node, parent, E, interval)
2:      moveVector ← node.coordinates − parent.coordinates
3:      slices ← (node.t − parent.t)/interval
4:      for i from 0 to slices do
5:          time ← parent.t + i ∗ interval
6:          point ← parent.coordinates + moveVector ∗ i/slices
7:          for each enemy in E with E.state = ALIVE do
8:              if distance(enemy.position, point) > enemy.d_F then
9:                  continue
10:             end if
11:             slice ← enemy.FoV(time)
12:             collision ← IsPointInsideTriangle(slice, point)
13:             if collision == True then
14:                 if !ObstacleCollision(enemy.position(time), point) then
15:                     return True
16:                 end if
17:             end if
18:         end for
19:     end for
20:     return False
21: end procedure
```

reaching the root node of the RRT structure. Since it is unlikely that we would sample the exact ending point coordinates at random, we check this after each iteration in which a node was added to the RRT structure, performing the same collision validations as the ones used for node sampling but this time by trying to connect the last sampled node and the ending point. This process is shown in the algorithm 9.

A second ending condition added to the algorithm is that of sampling $\mathbf{m}$ nodes. In this case, by reaching this limit the algorithm would have failed in finding a solution path. This condition also avoids continuing to expand a tree that is already dense. Although RRT will eventually find a solution if one exists, it benefits from restarting rather than just growing, as each restart allows it to explore a different expansion and routes without being biased by the existing nodes and their connections.

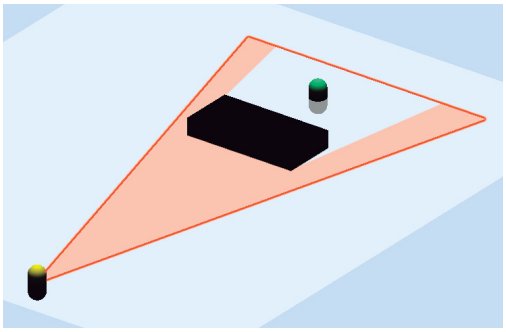


Figure 3.24 – Enemy FoV occlusion by an obstacle. While an intersection between the player (green) and the enemy FoV (orange) is detected, the line of sight of the enemy is blocked by an obstacle, thus no detection should be produced.

Both of these checks are performed after completing each iteration. These processes are shown in lines 4 and 23 of the general algorithm 2 on page 50.

Algorithm 9 Ending Condition

Require: $node$, ending point B, player speed v_p, set of enemies E

1: **procedure** GOALISREACHABLE($node, B, v_p, E$)
2: $goalNode \leftarrow B$
3: $goalNode.t \leftarrow node.t + \mathrm{d}(B, node)/v_p$
4:
5: **if** obstacleCollision($goalNode, node$) **then** // Collision check.
6: **return** False
7: **end if**
8:
9: **if** enemyCollision($goalNode, node, E$) **then**
10: **return** False
11: **end if**
12: **return** True
13: **end procedure**

Chapter 4
Proposed Solutions

In this chapter we analyze the problems related with the take down and body hiding mechanics in a stealth scenario, as well as a solution using an augmented version of the search algorithm presented in the previous character. We begin by reviewing the stealth problem and the elements involved; afterwards, we present the modifications needed in the RRT algorithm to solve this problem. Finally, we present different improvements that can be used to improve the sampling step of the RRT algorithm for this problem. In the second part of this chapter, we follow the same structure focused on the body hiding problem.

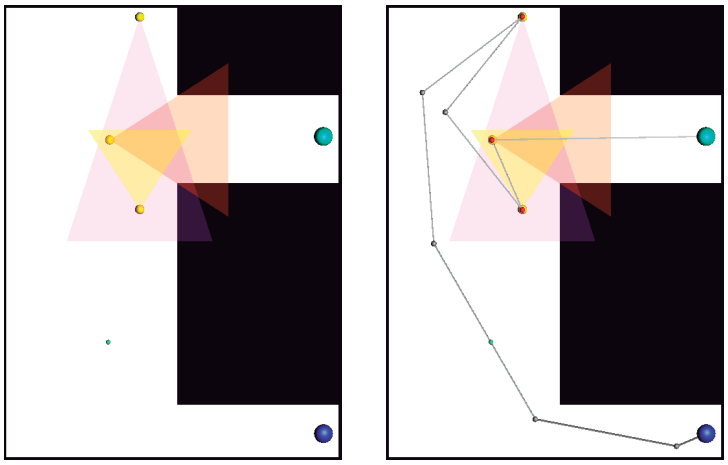
4.1 Take Down Model

In previous sections we have explored the modeling of a solution for the stealth problem using RRT, however, the solution presented up to this point only deals with navigating a scenario while avoiding the FoV of the enemies, and thus it is unable to deal with scenarios where enemies block the path towards the goal. In this section, we present the modifications to the basic stealth algorithm in order to deal with this situation by giving the player the ability to take down enemies.

We will begin with a recapitulation of the take down problem, providing a description of the solution proposed for handling these scenarios as well as some improvements over the naive approach.

4.1.1 Take Down Problem

Take down problems are typically motivated by the inability of a player to reach the goal position without observation by an enemy agent. In such cases, it becomes necessary to provide the player with abilities to either engage in (violent or non-violent) combat to eliminate an enemy, or employ *distractions* [2] to cause enemies to change their behaviour.



(a) (b)

Figure 4.1 – Take down problem example. (a) In this case, it is impossible for the player to reach the goal without being noticed by any of the enemies. Even if the player has the ability to eliminate the enemies this cannot be done in an arbitrary order since engaging with some of them will require to enter the field of view of another enemy. (b) A possible solution to the problem when the player is given abilities to take enemies down.

A player engaging in combat or trying to take an enemy down must still avoid alerting other agents who may observe the act. The puzzle-like nature of the problem then becomes more complex when enemies are observing each other. Figure 4.1a shows an example. The challenge is to reach the goal on the right-hand side, which is protected by the guard with orange FoV. However, in order to eliminate (take down) this guard, the player must first remove the guard with yellow FoV, which itself requires removing the guard with the pink FoV. The chain of dependencies is the basis for the take down problem.

Figure 4.2 shows a 3D rendering of a solution for the previous scenario where the take downs are outlined as red nodes along the path. Since an enemy that has been eliminated cannot no-

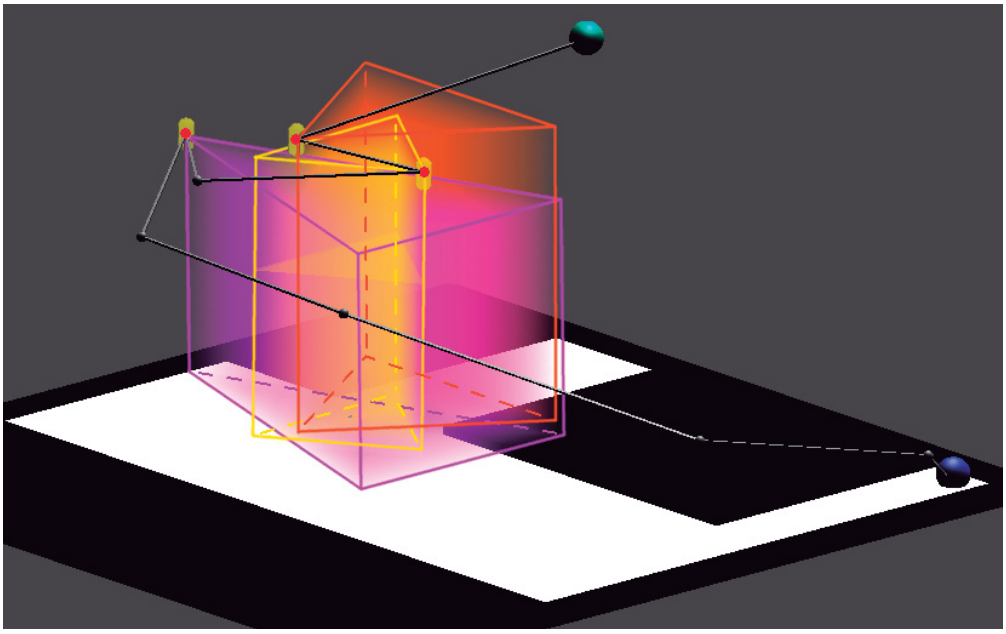


Figure 4.2 – Take down problem solution representation. In this case, The points where the enemies are taken down are shown as red circles that cut the volumes generated by the FoVs.

tice the presence of the player its FoV does not act as an obstacle anymore, and thus in future representations the volumes generated by the FoVs will be cut at the moment of a take down.

4.1.2 Naive Take Down Using RRT

A basic, **naive approach** to the take down problem is entirely based in a random sampling of the RRT nodes. Since a take down can only be performed when the player is at a smaller distance to the enemy than the **take down radius r**, this means that the sampling has to select a point inside this area in order to perform a take down.

In order to generate a solution for the take down problem, we augmented the RRT nodes to incorporate information about the state of each enemy (defined in table 3.1 on page 3.1). While performing the RRT algorithm, we create the root node with the status of all enemies set to *alive*. Then, a sampled node assumes that the statuses of the enemies are the same as in its closest neighbor (parent node). This ensures that any sampled node will inherit the changes in the status of the enemies that happened in any of its ancestor nodes.

Another factor involved in the take down problem is its puzzle-like nature: the order in which the enemies are neutralized can be important. In early experimentation we noticed that the closest enemy to the starting point is frequently chosen first for take down because the speed restriction tends to make it easier to connect nodes sampled close to the RRT. To increase exploration of different orders in taking enemies down, we add a parameter that defines the **take down probability** $\mathbf{p_{ko}}$: when the player reaches a position where a take down can be performed it might decide not to do so.

Algorithm 10 Naive Take Down

Require: $node, parent$, set of enemies E, take down probability p_{ko}, take down radius r

```
1:  procedure NAIVETAKEDOWN(node, parent, E, p_ko, r)
2:      for each enemy in E do
3:                                          // Cannot take down an unconcious enemy
4:          if node.getState(enemy) != Alive then continue
5:          end if
6:                                          // Take down based on probability
7:          if Random.value > p_ko then continue
8:          end if
9:
10:         if distance(enemy, node) > r then
11:             continue        // If the player is not in a right location to perform a take down,
12:         end if                          a bias can be performed to correct the position
13:                                         instead of continuing. See subsection 4.1.3
14:         node.setState(enemy, Unconscious)
15:         node.k_p ← True
16:         break
17:     end for
18: end procedure
```

Whenever a take down is performed, the corresponding node is updated so that the state of the enemy taken down is changed in this node and therefore all the branches of the RRT generated from this node will consider the enemy as unconscious. Algorithm 10 shows the pseudocode for this naive search strategy.

4.1.3 Improvements to the Model

While the naive approach can find a correct solution for the take down problem, it relies entirely on randomness to perform a take down and to step into a zone where it can be performed. This means that the smaller the **take down radius** $\mathbf{r}$, the less likely a correct solution will be found. Furthermore, if we restrict the take down behaviour to only be performed when the player is right next to the enemy, it will be highly unlikely for the random search to select the exact position of the enemy. Therefore, we explore some improvements to the naive approach in order to handle this problem.

The improvements proposed are based on the amount of knowledge that we allow the algorithm to access. A *full-knowledge* approach can predict or calculate correctly the position of an enemy at any given time value, either greater or lower than the actual one. On the other hand, *partial-knowledge approaches* only have access to the position of the enemies up to the current time value.

4.1.4 Bias Approach

The first proposed approach is a *full-knowledge* method in which the algorithm has access to the position of all enemies for a given time value. We name it the **bias approach** as it is based on biasing the sampled nodes to a correct position for a take down.

The process of biasing a node comes right after a sampled node fails to perform a take down in a naive manner (even if the node is inside the **free space** $\mathbf{S_{free}}$), as indicated in line 11 of algorithm 10. In case the take down fails due to the sampled point being outside the **take down radius** $\mathbf{r}$, we try to modify the sampled node by checking whether the parent node can be connected to another random point outside the enemy's FoV but inside the **take down radius** $\mathbf{r}$. If a node with these characteristics is found, then we use that point instead of the randomly sampled node.

The process for biasing a node is shown in algorithm 11. An important part of the bias is related to choosing a point inside the **take down radius** $\mathbf{r}$ of the enemy with the same time value as the failed node. We call this area the *blind zone*, and a point inside this zone is referred to as *blind spot*. Since we are using a circular zone around the enemy we can extract the angle of the FoV from the blind zone and choose a random point in the remaining area using a random angle and radius inside such bounds.

This approach implies the need for additional checks. The paths implied by connecting nodes in

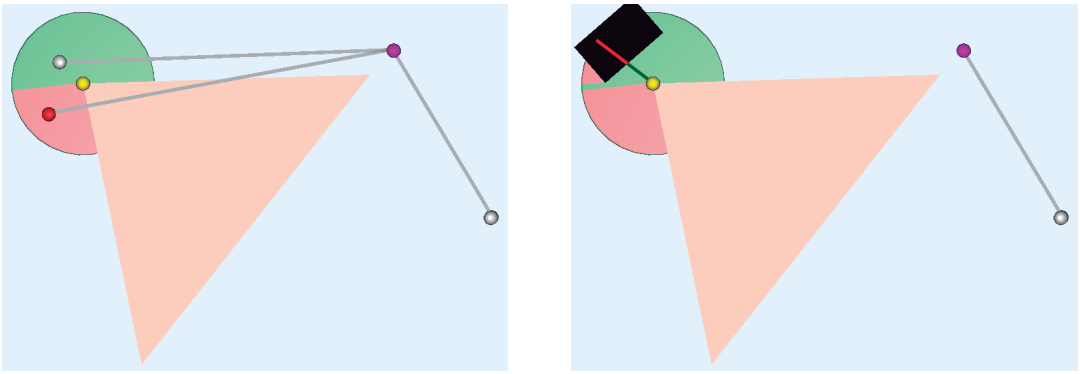
Algorithm 11 Bias Take Down

Require: $failedNode, parent, enemy$, take down radius r

```
1:  procedure BIASTAKEDOWN(failedNode, parent, enemy)
2:      biasNode ← enemy.getBlindSpot(failedNode.t, r)
3:      if !validateNode(biasNode, parent) then return          // See Algorithm 6
4:      end if
5:      if obstacleCollision(biasNode, parent) then continue    // See Algorithm 7
6:      end if
7:      if enemyCollision(biasNode, parent) then continue       // See Algorithm 8
8:      end if
9:      biasNode.setState(enemy, Unconscious)
10:     biasNode.k_p ← True
11:     biasNode.parent ← parent
12:     RRT.add(biasNode)
13: end procedure
```

the RRT must be straight line trajectories, and while we have verified that the parent node feasibly connects to the *failed node*, our *biased node* does not trivially inherit that property: a portion of the straight path to the parent may intersect the FoV, as shown in figure 4.3a. This can result in the bias process failing for a large number of random blind spots, particularly when the player is traveling from one side of the enemy to a random blind spot on the other side, and the wider the angle of the FoV, the less freedom we have to select valid spots. We improve this by selecting only points that are on the "opposite" side of the FoV (180° from the forward vector of the enemy), and we ensure that any node on the right or left side of the FoV have access to this angle without crossing the FoV.

Although this solution handles the problem of crossing the FoV it can still result in invalid nodes when an obstacle is present near to the radius. This is illustrated in figure 4.3b, showing that valid points in terms of avoiding crossing the FoV may be invalid in terms of being occluded by the presence of an obstacle near to the enemy, a problem that becomes worse with a larger **take down radius r**.Our implementation checks for this as well, but if an enemy is placed with its "back" close to a wall or obstacle, it will have a success rate of performing a take down very similar to a naive approach.



(a) (b)

Figure 4.3 – Blind spot sampling problems. When sampling a random blind spot inside an enemy (yellow) take down radius (green) we must consider that (a) some blind spots might create an intersection with the FoV (orange) when connected to a parent node (purple). (b) when using a fixed blind point among the radius, it might become invalid if the enemy is near to an obstacle which reduces the valid area to choose (green).

4.1.5 Prediction Approach

Our second approach reduces the amount of global knowledge required. Although the underlying RRT search itself still assumes knowledge of enemy positions and geometry, the local knowledge assumed in selecting a take down location is reduced, and instead of being based on the exact enemy future positions it is estimated based on the enemy motion. After sampling a valid node (that is, a valid connection to the RRT that traverses only the **free space** $\mathbf{S_{free}}$), there is a probability of trying to bias the search; if it is triggered, the algorithm checks the actual and previous position of the enemy and assumes that the same trajectory observed between those two points will be performed by the enemy over a similar time period. This is then used to guess a suitable take down location.

As shown in Figure 4.4, the algorithm uses the motion observed from the parent node to the sampled node. This gives us a prediction of the future position of the enemy after that same time interval, taking that position and time as the next sampled node for the RRT algorithm. In case it is not a valid node, the RRT algorithm will simply discard it and continue as normal. If it is a valid node, then the knock down can be performed as the enemy will be intercepted at that point, thus, providing a solution for that problem.

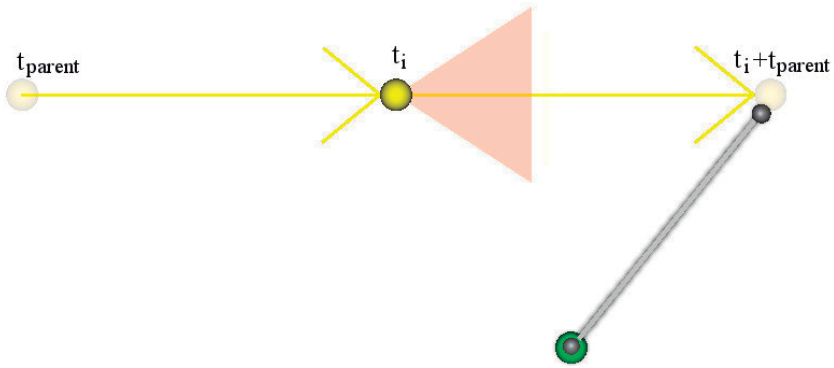


Figure 4.4 – Prediction approach. When biasing the search, the algorithm checks the difference between the enemy position in the sampled node and the position in the parent node. With this information, it assumes the same trajectory will be followed covering the same distance in the same time and samples a new node for this trajectory.

Algorithm 12 shows the pseudocode for the prediction take down approach. As with our bias approach, the predicted node has to be validated again to ensure it does not collide with an obstacle in the scenario or the FoV of this or any other enemy. The latter is a more complex check, since the prediction is based only on the behaviour of one single enemy and does not itself analyze the other elements of the scenario.

4.1.6 Correction Approach

The last approach for biasing the take down search is a *partial knowledge* approach, where the least information about the enemies is provided to the algorithm in order to effect a take down.

Our previous approaches aimed to bias a node by extrapolating from nodes already known to be in the **free space** $\mathbf{S_{free}}$. This final approach focuses on taking advantage of the nodes that would normally be discarded for falling inside an enemy FoV. This approach is inspired by personal experience while playing stealth games, where sometimes the player does not know about the location of the enemies and thus tries to perform certain movements only to find that an enemy is really near and blocking its way. Skilled players in such situations may be able to quickly react and change their motion to try to take down the enemy before being intercepted: they *react* to a potential failure. In the same way, our **correction approach** does not provide any information about the enemies to the take down part of the algorithm unless a node is sampled inside a FoV

Algorithm 12 Prediction Take Down

Require: $node, parent, enemy$, take down radius r

1: **procedure** PREDICTIONTAKEDOWN($node, parent, enemy$)
2: $movement.x \leftarrow node.x - parent.x$
3: $movement.z \leftarrow node.z - parent.z$
4: $interval \leftarrow node.t - parent.t$
5: $predictedNode.x \leftarrow node.x + movement.x$
6: $predictedNode.z \leftarrow node.z + movement.z$
7: $predictedNode.t \leftarrow node.t + movement.t$
8:
9: **if** $!validateNode(predictedNode, parent)$ **then return** // See Algorithm 6
10: **end if**
11: **if** $obstacleCollision(predictedNode, parent)$ **then return** // See Algorithm 7
12: **end if**
13: **if** $enemyCollision(predictedNode, parent)$ **then return** // See Algorithm 8
14: **end if**
15: **if** $distance(enemy, node) > r$ **then return**
16: **end if**
17:
18: $predictedNode.setState(enemy, Unconscious)$
19: $predictedNode.k_p \leftarrow True$
20: $predictedNode.parent \leftarrow node$
21: $RRT.add(predictedNode)$
22: **end procedure**

of the enemy. In a naive approach this would simply result in discarding the node, while in this approach we interpret it as an opportunity to react—the player has seen an enemy in their way, giving the search algorithm access to the position of an enemy.

The algorithm discards the node but checks the point in which the player path intersects the FoV of the enemy. Since the FoV is represented by a triangle, it assumes that one of the vertices of the edge where the intersection occurs is connected to the enemy; an example is shown in figure 4.5. The algorithm then follows this edge to find a point just outside the FoV but within the action radius that would allow for the player to take down the enemy. This point becomes a newly sampled node for the RRT algorithm; if it can be validated and connected to the RRT tree then it is a solution for that take down, and otherwise, the algorithm continues normally.

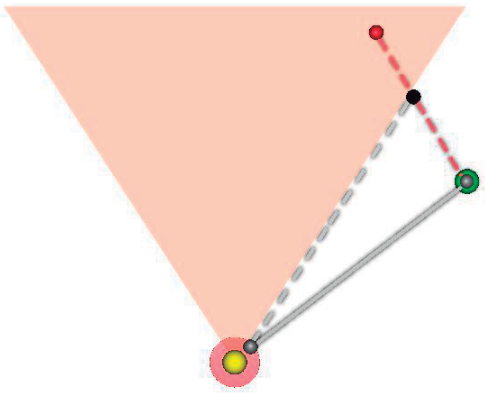


Figure 4.5 – Partial knowledge approach. When a node (red) is sampled inside the FoV of an enemy (orange), the intersection point of it with the path from the parent node (green) is calculated (black). Then, using the FoV edge as a reference it tries to aim for a point where the enemy can be taken down (gray), and if it exists, it is sampled for the RRT.

Note that this approach gives us one of two opportunities to find a take down spot, either from the left or right of the FoV. We do not process a node that intersects the FoV on the edge that it is not connected to the enemy (top in figure 4.5) as this would mean being intercepted by the enemy head-on, and it is improbable that a node can be projected near to the enemy without going through the FoV again.

Unlike previous approaches that perform the node bias after the validations have resulted in a positive outcome, this method performs it when the enemy collision check fails—line 16 of the algorithm 2 on page 50. After attempting the node bias we discard the node sampled naively and instead go back to the parent selection and validation of the corrected node as if this had been sampled naively. The pseudocode of this approach is shown in algorithm 13; here the computation of the intersection point and the FoV, as well as the verification of the line segment to which the point belongs to, are performed by a line segment to line segment intersection check of the path with each of the three triangle edges.

Algorithm 13 Correction Take Down

Require: $node, parent, enemy$, take down radius r

```
1:  procedure CORRECTIONTAKEDOWN(node, parent, enemy)
2:      path ← node − parent
3:      intersection ← TriangleLineIntersection(path, enemy.FoV)
4:      edge ← getEdge(intersection, enemy.FoV)
5:
6:      if distance(edge.vertex1, enemy) < r then
7:          // Samples edge.vertex1
8:      else if distance(edge.vertex2, enemy) < r then
9:          // Samples edge.vertex2
10:     end if
11: end procedure
```

4.2 Body Hiding Model

Up to this point, we assumed that taking down an enemy will simply delete it from the environment. However, in modern games this is rarely seen, and normally after knocking down an enemy the body is left behind or at least some evidence of the combat like blood or weapons is present. In many games this evidence can alert other enemies if they come to notice it and can even become a failure for the player.

This presents a new challenge for the player, in the form of trying to hide the evidence from other enemies, either by knocking down the enemies where no other can see, knocking all the enemies before they notice the evidence, or by hiding the evidence. Assuming evidence is in the form of the enemy body, there are then three main variations on the *body hiding problem*, depending on when a hiding location can be calculated:

- **Preset**. The evidence must be deposited in predefined "safe spots" set within the scenario by the level designer for this purpose, such as containers or other disposal areas. In *Dishonored*, for example, large trash containers serve this purpose. These sites can be even set without caring about the movement of enemies.

- **Calculated**. The evidence must be placed in spots where no field of view of the enemies can reach. This may include permanently shadowed areas, as well as areas that are sim-

ply not covered by enemy FoVs. Computing these areas requires full knowledge of enemy movement and their FoV as well a preprocessing of the scenario before its use in the search algorithm. This implies that the player will learn the movement of enemies before performing an action.

- **Learned**. The evidence can be left at any point that the player decides, as long as it is not seen before solving the problem. In contrast with the previous method which requires finding permanently unseen areas indifferent to the events during the search process, this version computes safe spots incrementally, adding or reducing regions as enemies are eliminated.

In general, body hiding induces a separate pathfinding problem after a take down event. In many games, this search requires the use of a different movement model. Since only one body can be carried at a time, this often also reduces mobility or adds action limitations. In this section, we present approaches to solving a body hiding problem using either the **Preset** or **Learned** hiding spots. The case of the **Calculated** hiding spots, although not addressed in this work, can be reduced to a **Preset** hiding spots problem after computing the dark areas of the scenario, a problem that has been studied in related works such as Tremblay's [52].

Another important factor to consider is that a solution is not guaranteed for all the possible configurations of a body hiding problem. For the **Preset** and **Calculated** approaches, it depends on the design of the levels, and requires the designer ensures hiding spots are always accessible. For the case of the **Calculated** approach, this will depend on the enemy behaviour, since calculating the unseen areas of the scenario before any take down is performed means that all the areas covered at some point by a FoV will be considered unsuitable for a body drop. For scenarios where all the areas are covered by the enemies, no dark spot will be found and thus, no body drop could be performed. On the other hand, the **Learned** approach defines the hiding spots at runtime, and thus the existence of a solution will only be known after some execution.

In the following sections, we will present two approaches for the **Preset** and **Learned** hiding spots in a body hiding problem. For its design, we must consider how the RRT algorithm is modified in order to incorporate elements and events related to the body problem.

4.2.1 Body Hiding RRT

In order to solve the body hiding problem using RRT, it is necessary to include the elements related to hiding spots that were described on page 47 of section 3.3.1. These elements are stored on a different layer than the obstacles of the scenario as hiding zones are not considered to restrict movement.

Another element adapted to the body hiding problem is the enemy movement behaviour. Up to this point the behaviour of the enemies is ruled by the defined waypoints until a take down is performed, after this, the algorithm does not care about the eliminated enemy and simply ignores it. However, for the body hiding problem that is not the case, and we are interested in the position of the body after the take down. Here, the body hiding model implemented assumes the following constraints:

- An unconscious enemy gets carried by the player instantly after the take down.
- The enemy position while being carried is exactly the same as the player position.
- While an enemy is being carried, the player cannot perform another take down.
- After a body drop is performed, the enemy cannot be relocated again (a single body drop for each take down).
- The goal cannot be reached if the player is carrying an enemy.

For the RRT to handle these constraints, we expand the nodes to include information of the enemy body positions as described in equation 3.12 on page 49. This way, for a node where an enemy has the status of *Unconscious* (Table 3.1 on page 44), we will forbid the take down action, and thus no node can contain two unconscious enemies. Furthermore, we will update the body position of such an enemy to be equal to that node coordinates, which means that the player has carried the enemy up to that node. On the other hand, when a node is validated to be inside a hiding zone, a body drop can be performed, which will update the enemy status to *Hidden*. However, for the **Learned** approach, it will update it to *Dropped* as it is just a potential safe spot.

As it can be inferred, the body hiding solutions depend entirely on the player getting into a hiding zone or around a potential safe spot after performing a take down, similar to the way the

take down problem depends on the node sampling choosing a coordinate inside the take down radius of an enemy. Thus a similar approach can be used in which a validation will be performed on every sampled node to check if it is within a zone where a body drop can be performed, and we control this action using a parameter for the **body drop probability** $\mathbf{p_d}$. This is illustrated in algorithm 14 which will be performed after validating and connecting a node to the RRT; that is, the body hiding will be added at line 22 of algorithm 2 on page 50.

Algorithm 14 Naive Body Hiding

Require: $node, parent, p_d$

```
1: procedure NAIVEBH(node, parent, p_d)
2:     enemy ← node.getEnemyUnconcious()
3:     if enemy == null then return
4:     end if
5:     node.setBodyPosition(enemy, node.coordinates)
6:     if isInsideHidingZone(node) && Random.value < p_d then
7:         node.setState(enemy,' Hidden')
8:     end if
9: end procedure
```

4.2.2 Preset

When the hiding spots are defined by the level design and are specifically set for that purpose we treat them as permanent zones where the bodies can be dropped and assume that being part of the scenario design, no body will be seen in these places.

It is important to notice that the main challenge of the body hiding problem does not come from the action of dropping the bodies, but rather from trying to avoid having the bodies be seen by the enemies that are still alive. Since our constraints assume that a body cannot be seen in a valid hiding zone, and also, the body position of an unconscious enemy is the same as the player's, then a node that is validated to create a path traveling only through the **free space** $\mathbf{S_{free}}$ also ensures that no body is noticed. This means, that the enemy collision validation performed on the basic stealth RRT is enough to check that all bodies are hidden for this approach. Therefore, simply by adding the procedure in algorithm 14 to any of the take down approaches, we will be able to solve the body hiding problem in a naive way.

Algorithm 15 Preset Body Hiding Bias

Require: $node, parent, p_d$

```
1:  procedure PRESETBH(node, parent)
2:      enemy ← node.getEnemyUnconcious()
3:      if enemy == null then return
4:      end if
5:      if Random.value > p_b then return
6:      end if
7:      spot ← getHidingSpot()
8:      biasedNode = sampleNodeInside(spot)
9:      if !validateConnection(node, biasNode) then return
10:     end if
11:     biasedNode.setState(enemy, Hidden)
12:     biasedNode.parent ← node
13:     RRT.addNode(biasNode)
14: end procedure
```

However, as previously seen in the take down problem this naive search depends entirely on the probability of selecting the correct sequence of nodes. While this by itself is hard to accomplish for the take down mechanic, adding a new process that depends on this randomness again further reduces the success probability of the algorithm. This probability will also be affected by the size of the hiding zones compared with the total size of the space. Therefore, it is useful to have a method for biasing this search in a way that can help the search to accomplish its goals. In this case, since the hiding zones are static elements, the biasing does not need any kind of prediction over its position as the enemies did require.

Biasing proceeds much like the take down bias. After taking down an enemy and adding the node to the RRT, the next step will be to check if any of the hiding spots are reachable from the sampled node. If so, the hiding spot position will be the next sampled node for the RRT, and otherwise the RRT continues as normal adding the same check after every new sample that includes an unconscious enemy.

Algorithm 15 presents the pseudocode to perform a body drop on a preset hiding zone. Here we decided to use a function that selects at random the hiding zone that the algorithm will try to bias towards; it could, however, also iterate over all the hidden zones to check if any can be connected

to the node. We also sample a random point inside the hidden zone in case there are some portions of it that are less accessible than others. To leverage this sampling, we use only rectangular hiding zones for this approach. To control the degree of bias in the algorithm we use a parameter for the **bias probability** $\mathbf{p_b}$.

4.2.3 Learned

The learned solution assumes that no hiding spots are present in the scenario and its task is to discover which places in the scenario are out of the sight of the enemies either permanently or for a period of time, while at the same time looking for a solution for the scenario. It tries to avoid any calculation previous to the execution and takes advantage of the validations performed by the stealth algorithm to check for collisions during the naive sampling.

In order to learn where to hide the bodies the **body drop probability** $\mathbf{p_b}$ is used for each sampling. Again, it assumes that the body hiding action is started immediately after the take down. The drop probability allows for the player to drop the body at any randomly sampled node that is in the **free space** $\mathbf{S_{free}}$ at certain time value, without knowing if it will be in the sight of an enemy at a later time. In case it is reached at some point during the RRT execution, then, any attempt to expand the RRT branch that goes through that node will fail after the time value where it is seen. Furthermore, if during execution, a collision validation fails for that node, it will be stored as an unsafe spot with an added **hiding radius** $\mathbf{r_h}$, assuming that if a specific location can be reached by the enemies, those close to it can be reached too. These unsafe spots will increment as the algorithm iterates creating bigger unsafe areas every time. On the other hand, if the algorithm is able to find a solution for the scenario, once the goal is reached, all the coordinates where bodies were dropped will be stored as potential hidden spots. However, it is important to notice that the unsafe areas will also depend on the order and timing of the take downs since a coordinate might be classified as safe because the enemy able to reach that spot was already neutralized. Algorithm 16 shows the process to increase the learned hidden spots.

This body hiding approach also adds a new value for the status of the enemies: now they can be either *Alive, Unconscious, or Dropped* (See Table 3.1 on page 44). The state *Hidden* is not used for this approach since we assume that the algorithm cannot know for sure if a spot is never seen for any time value. Due to this, it is now necessary to additionally check on every sampling

Algorithm 16 Learned Hidden Spots

Require: $endingNode, E$

```
1: procedure LEARNHIDDENSPOTS(endingNode, E)
2:     if Solution was not found then return
3:     end if
4:     for each enemy in E do
5:         if endingNode.getState(enemy) == Dropped then
6:             hiddenSpots.add(endingNode.getBodyPosition(enemy))
7:         end if
8:     end for
9: end procedure
```

if no enemy body is caught inside the FoV of another enemy. This was not needed for the **preset** approach since there, we assume that the scenario designer guarantees that the hidden zones are objects that do not allow for an enemy to see a body inside them, while in this approach we assume there are no such objects.

Algorithm 17 shows the process to check if, for a given connection of a pair of nodes, no body is seen by other enemies. This validation uses the enemy interception check presented in the naive RRT search. In case a body is intercepted, if the coordinate was previously considered a hidden spot, it will be removed from that set and added to the exposed spots set.

When this approach is combined with the search bias towards a hidden zone we can expect the algorithm to perform better after a first solution is found naively and keep on improving, potentially reaching performance close to that ofan approach where all the places permanently out of sight had been calculated before execution.

4.3 Analysis of the Proposed Solution

In Section 3.1 on page 20, we presented a series of factors that impact directly on the complexity of a solution algorithm for the take down and body hiding problems. After presenting our proposed solution in this chapter we will refer to such factors and analyze the complexity introduced by each of them.

The first factor is the representation of our scenarios. Since most modern games use 2D and 3D

Algorithm 17 Validate Hidden Bodies

```
Require: node, parent, E
1:  procedure ValidateHiddenBodies(node, parent, E)
2:      for each enemyD in E do
3:          if parent.getState(enemy)==Alive then continue          // Ignores enemies alive
4:          end if
5:          body ← node.getBodyOf(enemyD)
6:          for each enemyA in E do
7:              if parent.getState(enemy)!=Alive then continue      // Dead enemies cannot see
8:              end if
9:              if enemyCollision(body, enemyA, node, parent) then  // See Algorithm 8
10:                 hiddenSpots.remove(body)
11:                 exposedSpots.add(body)
12:                 return True
13:             end if
14:         end for
15:     end for
16: end procedure
```

scenarios we aimed for a representation that allows reducing both of them in 2D, this allows for our search space to be reduced by one dimension without losing the original mechanics. Furthermore, applying a pathfinding algorithm in a 3D space could lead to a higher number of invalid coordinates since 3D scenarios are often still quite constrained—in most 3D games, for example, the player is not allowed to walk in the air or to go below the floor. By approximating 3D mechanics in a 2D space using the elements presented in section 43 on page 45 we can use pathfinding algorithms without the strict need to discretize the scene space or process the scene to find a navigation surface.

The RRT algorithm, as a random sample-based algorithm with probabilistic completeness, allows us to search the space without exhaustively exploring all possibilities, but rather explore a small subset of them every time. Furthermore, being a non-deterministic solution it allows for the exploration of different solutions in a single scenario.

By implementing a KD-Tree as the data structure for our RRT algorithm we have the advantage of querying the nearest neighbour in an expected $O(log\ n)$ time [48]. As the most expensive step on the basic RRT algorithm, and being bound in our solution to n nodes, we can expect an execution with time complexity of $O(n\ log\ n)$ as the basis.

Before getting into the validation steps we must first address other factors that defined how such validations were performed. These are the models for the agents, movement and obstacles. In a video game, even though objects could have elaborate 3D models, their position is often defined by a single point. With this, we also turn our collision steps into point-in-polygon problems for which several solution methods are known in the literature.

The movement in our model was defined as a linear movement in any direction at any step which allows us to keep a continuous coordinate space with Euclidean distance. In the case of enemies, the movement was controlled by a waypoint system in which each waypoint defines the speed of the movement. Since our model was implemented in *Unity 3D* we defined this speed using a frame as our time unit.

In the case of obstacles, we modelled them as polygonal 2D objects in our scenario. This way, validating if a node of the RRT process can be connected only required checking if the movement vector from the nearest neighbour overlaps these polygons. In Unity 3D we solved this by ray-cast using the movement vector and filtering the operation by checking for collisions only in the obstacles layer.

Having these three factors the validation step of our RRT algorithm was divided into three phases: checking the speed constraint, obstacle collision and enemy detection. For the speed constraint step, we validated that the fixed speed for the player was not surpassed, this was calculated by the difference in the time t between the node to connect and the nearest neighbour as well as the distance between them, adding a single arithmetic operation to the complexity of the algorithm. For obstacle collision as we mentioned before we based the detection in 2D ray-polygon intersection. In this case, the time dimension was ignored since all obstacles in the scene were static. On the other hand, the enemy collision detection had more complexity since their movement projected in an extruded 3D space (with the third coordinate being the time) created a volume, while the player movement created a line, turning this into a volume-line intersection problem. However, given the rotation of enemies, the volumes created present helicoidal shapes which by itself made this intersection detection task more complex as shown in section 18 on page 30. In order to avoid this, we discretized this movement by *"slicing"* it at time intervals (See section 3.4.2 on page 58).

Checking for enemy detection on these intervals was then reduced to point-in-polygon tests which were made using the ray intersection algorithm. This has an $O(N)$ complexity where N is the number of edges of the polygon [53]. In our case the FoV of enemies is always triangular,

making this computation have constant complexity for each test. We introduced the **Granularity factor $\mathbf{t_g}$**, to define the number of slices that will be tested for intersection. As explained in section 3.4.2 on page 58, the greater t_g is, the more accurate the detection, but also the complexity will increase. The number of intersection tests to perform after sampling point X and having found its nearest neighbour X_0 is $(t_X - t_{X_0}){t_g}^{-1}$. For our algorithm where there is an upper limit for the time value t_{max}, and the worst case is having to validate a node at the maximum time to be connected to the starting point ($t = 0$), in which case the number of tests will be $t_{max} \cdot {t_g}^{-1}$. This validation step is performed after each sampling and for each enemy, therefore the worst-case complexity of the validation step which will be performed for each node is $O(|E|\ t_{max}\ {t_g}^{-1})$, where E is the set of enemies in the scene.

The take down problem adds onto the previous complexity in almost the same way with the difference based on the computation of the biased node. First of all, we introduced the take down probability p_{ko} this will define how often the steps for taking down an enemy will be added to the validation step; in the case where $p_{ko} = 1$ it will be added after each node sampling. For our model the take down is only based on being close to an enemy up to certain threshold; this condition requires computing the Euclidean distance to each of the enemies alive, and thus taking down enemies adds a complexity of $O(|E|)$ to the validation step of the algorithm.

The other factor introduced was the bias of the nodes, this bias step will be triggered according to the bias probability p_b. Again, when $p_b = 1$ this step will be introduced for every validation step. In this case the added complexity changes according to the bias method used. For the case of the bias approach it will try to check if any enemy can be reached from the current position. For this it will add another validation step to check if the position of any enemy can be connected to the current node, adding the same complexity as the validation step but checking all the enemies positions, i.e. $|E| \cdot O(|E|\ t_{max}\ {t_g}^{-1})$. On the other hand, both the correction method and the prediction method calculate the biased node based on the current FoV of a single enemy or its current position respectively. This computation does not validate if the new node can be connected or if it collides with obstacles, therefore they add a constant complexity $O(1)$. From this we can see that although the bias method ensures that a biased node will be valid to perform a take down (therefore, helping the search to reach its goal), it is more expensive than the prediction or correction methods. The latter two, however, do not ensure the take down is feasible and still need to be validate it like a randomly sampled node on the next iteration.

In the case of the body hiding problem, this action was fixed to be started after every take down. For this we have two methods, the preset method and the learned method. Again, we introduce probabilistic variables to generate different option paths, in this case the body drop variable b_d indicates when a body drop will be attempted, adding this step to the validation phase of the algorithm. Once triggered we check if we are in a hidden zone, these zones were modeled as polygons in a separate layer in Unity 3D, and we use a point-in-polygon test to check if we are in such zones. With the help of Unity ray utility this step only adds a constant complexity to the validation step.

The body hiding problem also adds a bias probability p_b, in this case to bias the search towards a random hidden zone in the scene. After selecting the zone we simply add a random point inside it as the next sampled node, therefore adding a constant complexity to the problem.

Finally, the most expensive part of the body hiding problem comes from validating that no enemy detects a body after dropping it. In the case of the preset zones this step adds no complexity at all since the way we modelled the player carrying a body is giving to the body the same position as the player and because his position is already validated for detection, and the hidden zones in this method are ensured to be out of sight. For the learned method, the player is able to drop the body at any location if no hidden zone is known. Since that is not guaranteed to be out of sight, the validation step must check that no body is seen. The way we do that is by performing the same validation we did for checking that the player is not seen but this time for each of the bodies, giving the detection step a worst-case complexity of $|E| \cdot O(|E|\ t_{max}\ {t_g}^{-1})$. From this we can see that the preset approach requires less computation time than the learned approach, however, the former requires additional work from the level designer to set the zones.

This analysis shows clearly that the proposed solution complexity is mainly impacted by the detection step which introduces the highest time complexity and is used in many validations of the algorithm. Therefore, an improvement of this intersection test method will positively impact the overall performance of the solution.

Chapter 5
Experimental Results

In this chapter we present the design and results of the experiments performed to test the proposed approaches, we created a set of scenarios following different stealth configurations to first be able to test the configuration of the different parameters involved in the solution and how they affect the performance of the algorithm. We first present an overview of the settings and configurations of both the scenarios and the testing environment; afterwards we present the results of the tests for the different parameter configurations and finally compare the different approaches proposed.

5.1 Scenario Description

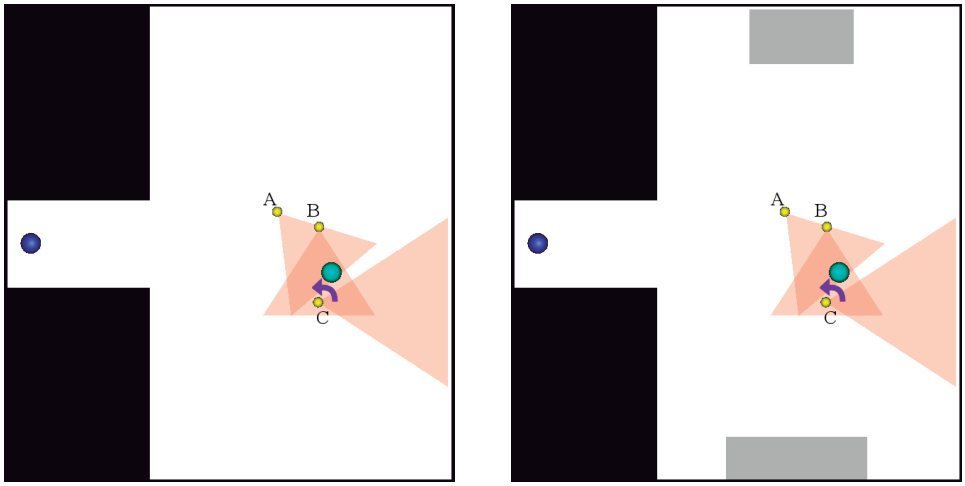
The experimental suite used to test our approaches include five different scenarios, based on different situations a player might face while playing a stealth game. Some of the scenarios are based on actual stealth game levels while others are abstractions of stealth challenges in them. We classify these scenarios into two groups: 2D scenarios which present basic situations and static obstacles that can be translated into a 2D world space, and 3D level approximations where we include elements that allow us to simulate a 3D environment in a two-dimensional space. Since these elements raise the complexity of the scenarios we use them to show the impact of this on the success rate.

To design these scenarios, we used the elements provided by the *Unity 3D* framework and tried to generate a basic model from game scenes without any kind of aesthetic decoration but keeping the general shape of the objects that affect the movement of an agent.

5.1.1 2D Test Scenarios

These tests scenarios were created in order to present different puzzles to the algorithm, the first was created just with the goal of investigating dependencies in take down order. The second one was based on a 2D representations of a real stealth game which was simplified to present the same puzzle as the original while abstracting the basic features of it: enemy behaviour, obstacles and goals.

Test Scenario



(a) (b)

Figure 5.1 – Test Scenario. (a) Configuration for the take down problem. (b) Configuration for the body hiding problem.

Figure 5.1 shows the design of the most basic scenario used for our tests. In this level, we can see three enemies, A, B, and C,represented as yellow circles. The starting and ending points are shown as big blue and green circles on the left side and center of the scenario respectively. The level presents a forced dependency on the order in which the enemies must be taken down to solve the puzzle.

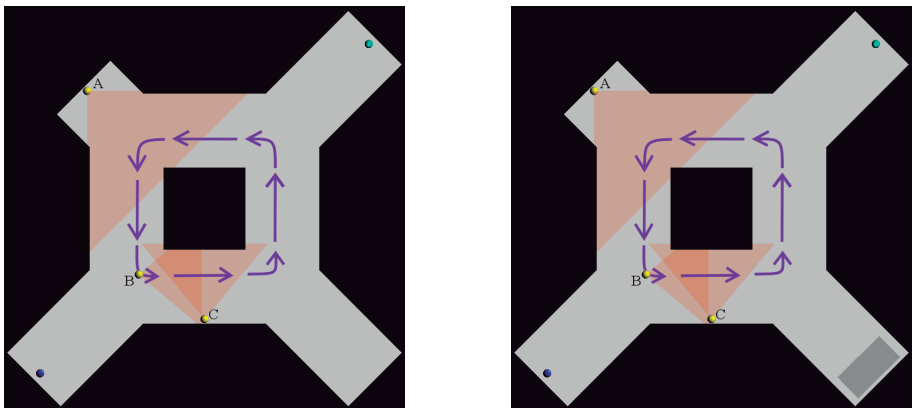
The enemies A and B are static enemies, thus, they do not move at all. Enemy C however, begins with its line of sight on the horizontal axis pointing to the right. After 10 timesteps it

performs a counterclockwise rotation of $90°$ up to a point where its FoV covers the other two enemies. After rotating, enemy C will wait for 10 timesteps before going back to its original position.

As shown, the ending point in the scenario is always under the FoV of enemy B, forcing the player to take out that enemy. However, enemy B is always inside the FoV of A, which at the same time is under the FoV of C at some time intervals. Furthermore, for some periods, both enemy A and the ending point come inside the FoV of enemy C, while this is inside the FoV of enemy B making it impossible to get rid of any of them. Thus, in order to solve the level, the player must take down the enemies in the order $A \rightarrow B \rightarrow C$ while avoiding the FoV of C

For the case of the body hiding problem using the preset hiding zones we set two areas where bodies can be dropped, these are shown as grey rectangular zones on the upper and bottom sections of the scenario.

Thief Scenario



(a) (b)

Figure 5.2 – Thief Scenario. (a) Configuration for the take down problem. (b) Configuration for the body hiding problem.

This scenario is an adaptation of the experimental level used by Borodovski to test their application of an RRT algorithm with distractions in a configuration that cannot be solved by plain stealth mechanics [2]. It is a reconstruction of the *Old Cathedral Courtyard* scenario in the *Hidden City* mission of the game *Thief* (Square Enix, 2014), illustrated in figure 5.3. Borodovski built this

level based on a gameplay video clip [54]; we used this reconstruction and removed the distraction elements to transform it into a stealth mission that requires at least one take down.

Figure 5.3 – Screens of a portion of the scenario for the *Hidden City* mission of the game: *Thief*. Images taken from a walkthrough video on Youtube [55]

The scenario (shown in figure 5.2) presents a puzzle in which the player has to move from the starting point at the bottom left corner and get to the top right corner of the scenario while facing two stationary enemies A and C and enemy B, who is patrolling along the route shown with purple arrows. Enemy B moves in a straight line on a rectangular route, rotating $90°$ after reaching each corner. It is easy to figure out that enemies B and C can be taken down, however, enemy C cannot be reached because its FoV watches over the whole access to its alcove, turning this section into an unreachable area of the scenario. For the case of the body hiding scenario with preset hiding zones, we set a hiding area in one of the corners of the scenario, as these are the places used by the player in the gameplay video to remain unnoticed by the enemies.

5.1.2 3D Tests Scenarios

As modern stealth games often rely on mechanics based on the environment to solve their puzzles (attacking from the rooftops or underground, teleportation, hidden passages, etc.) it is important to explore the behaviour of the proposed algorithm with scenarios that incorporate these features. However, in order to explore mechanics based on higher or lower heights, it is necessary to explore scenarios in three dimensions. To do this we explored an approach to simulate this third dimension behaviour inside a 2D representation based on different attributes for some portions of the scenarios.

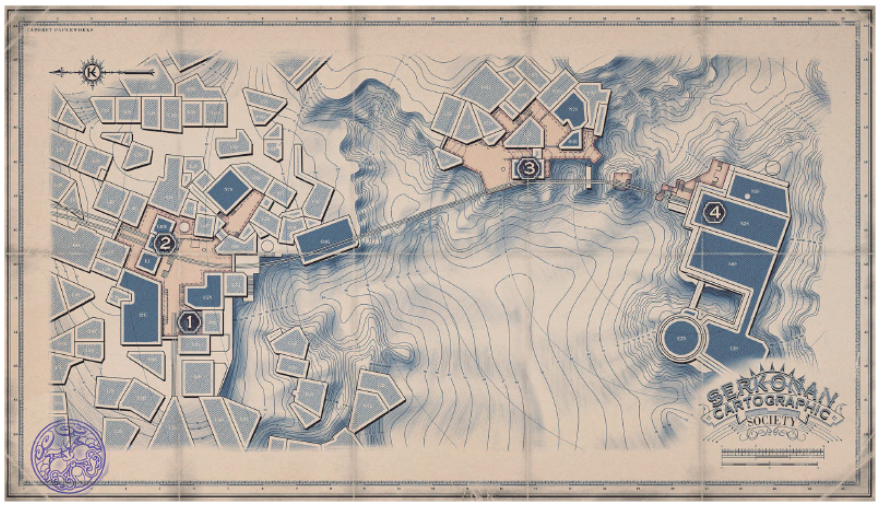


Figure 5.4 – Aventa District Map as shown in Dishonored 2. Image taken from [56]

These scenarios present a higher complexity, and require more precise actions in order to be solved; thus, its purpose is to show the impact of this increase in difficulty on the success rate of the algorithms.

Aventa Station Scenario

This scenario is a recreation of the environment used for the mission *The Clockwork Mansion* in the game *Dishonored 2* (Bethesda Softworks, 2016). A map of this area is shown in figure 5.4. Our focus here is on the area indicated by numbers 1 and 2. Here, a station takes the central place of the scenario with enemies patrolling around its entrance and an alley on the back of the station, which the player can also use to sneak into the station.

Due to the size of this scenario and the number of elements we divide this area in two places as the original map does. The first scenario, which we named simply *Aventa Station* deals with the first section.

Here an underground watercourse is the starting point of the player who has to deal with the enemies watching the entrance of the station as shown in 5.5. The enemies are watching on two levels of the ground, the first one elevated by a set of small stairs over the second one. Since the enemies on both of the floors can see each other we represent this difference in height simply using motion obstacles in the portions of the railing of the stairs, to force the player to take the stairs in

(a) (b)

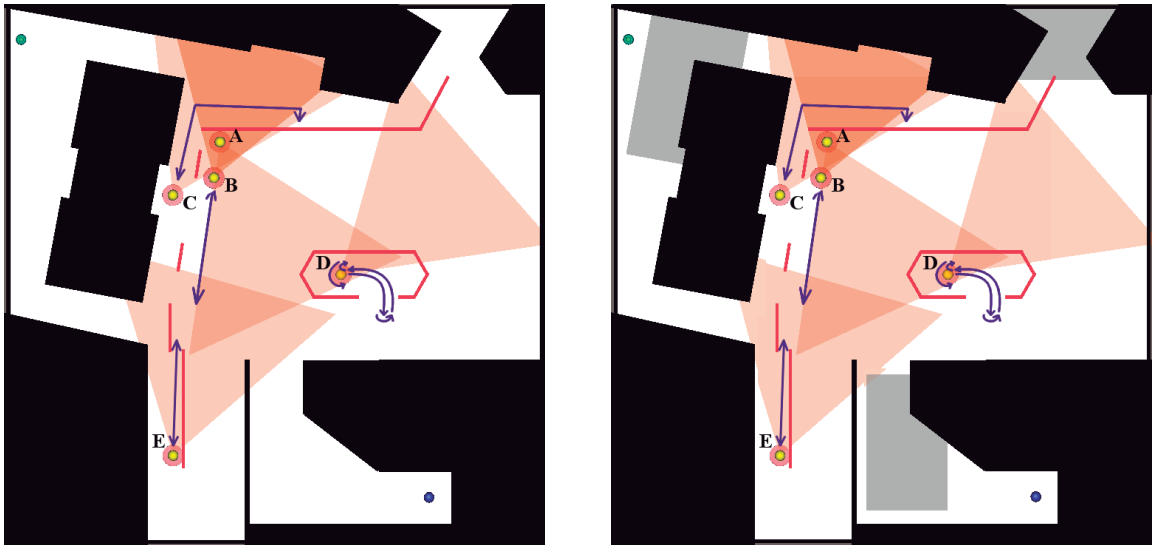
Figure 5.5 – Screens of the front section of the *Aventa Station* scenario in *Dishonored 2*. Images taken from a walkthrough video [57].

order to have access to the enemies in higher ground. In the bottom floor at the center of the screen, represented by enemy D on our reconstruction in figure 5.6 there's a guard inside a metal cabin watching over the plaza and the other enemies. Although the enemy is inside the cabin, it has sight of all the other enemies. To access this cabin, the player must either wait until the enemy steps outside of it or enter by an open portion of the cabin. To model these elements we use again motion obstacles that do not occlude the enemies sight but blocks the movement.

Figure 5.6 shows the scenario used in our experiments as well as some annotations about the behaviour of enemies. Here, enemy A is static, watching the bottom right corner of the scene up to the enemy inside the cabin. Enemy B and E watch over the bottom and top floor respectively moving repeatedly up and down along the stairs. Enemy C watches over the entrance of the station and the top section of the level going up and turning to the left until the midsection of the railing where he turns to watch over the entire plaza for 10 timesteps before going back to his original position.

The scenario also features an open section on the top right portion of the level which in the original game connects to another area of the district and in which the enemies in this scenario are never interested. To clear the scenario, the player must reach the back of the station building on the top left corner of the scenario.

For the body hiding problem with preset hidden zones, we place three of them, the first one is on the entrance of the level which leads towards the watercourse where the player begins and



(a) (b)

Figure 5.6 – Aventa Station scenario. (a) Configuration for the take down problem. (b) Configuration for the body hiding problem.

is covered by a building and a wall of, thus making it a section that no enemy ever inspects. The second one is on the open section on the top right which leads to another area that the enemies in the station cannot reach, and finally one on the back section of the station in the top left which represents a small alley that players use to get to the next section of the scenario and is somewhat of a hidden passage.

Aventa Alley Scenario

This scenario is the complement of the *Aventa Station* scenario as it covers the area at the back of the station, shown with a number 2 in the map in figure 5.4. Here, the player enters into an alley, where he can climb to sneak into the station. Figure 5.7 shows the original scenario in *Dishonored 2* while figure 5.8 shows our recreation which rotates the scenario.

The player can access this scene from both sides of the station to reach an area where two enemies C and D watch over a rectangular area below a set of ledges on the three buildings that circle the alley. While the player objective is to gather the resources at the end of the alley, both entrances are guarded by enemies. The first entrance (at the bottom of figure 5.8) is the

Figure 5.7 – Screens of the back section of the *Aventa Station* scenario in *Dishonored 2*. Images taken from a walkthrough video [57].

main entrance, as it is clearly signalled in the game, however, an enemy E placed behind a trash container watches the entrance ready to attack the player as soon as he turns the corner. The other entrance, on top, presents an open gate with two pillars. Assuming that the player will come from the main entrance, two enemies A and B are placed behind the pillars waiting to attack the player. In the original gameplay, this entrance is hidden and it is covered by traps that the player can easily dismantle. When facing the gate, the game presents A and B wrapped in a conversation where A loses the sight of B and thus, presents an opportunity for the player to take down both of them.

All the enemies in this scenario are modelled as static enemies, as in the original game their only behaviour is to wait until the player is caught in its ambush. A set of vision obstacles are placed around the level to simulate ledges on the buildings, this was done after realizing that in a number of gameplays, players tend to use this high ground to avoid any combat with the enemies.

Albarca Baths Scenario

This scenario presents the highest complexity in our test suite. It is a reconstruction of the *Albarca Baths Kennel* scenario of the game *Dishonored: Death of The Outsider* (Bethesda Softworks, 2017). A reference map taken from the game is shown in figure 5.9, our focus is on the section signalled with the number 2. Our reconstruction, shown in figure 5.10 presents the same scene rotated $90°$ counter-clockwise. Here, the player begins in a corridor on the left of the scenario which is connected with the central room by two entrances, on the top and left side. The top

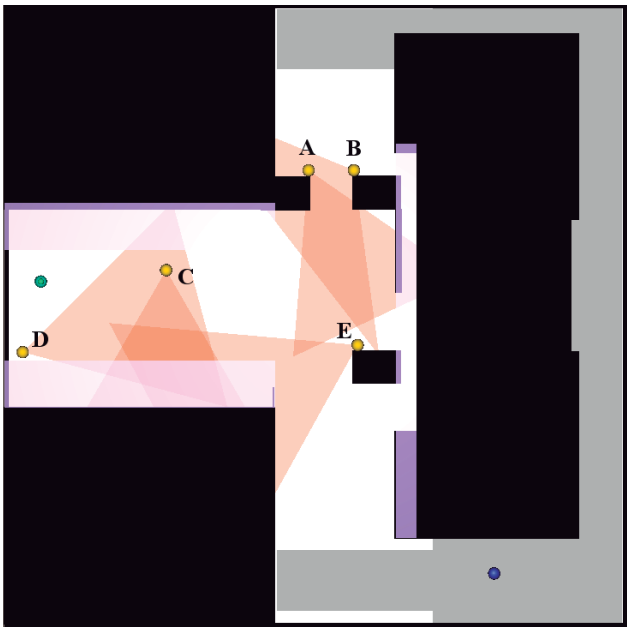


Figure 5.8 – Aventa Station Alley scenario.

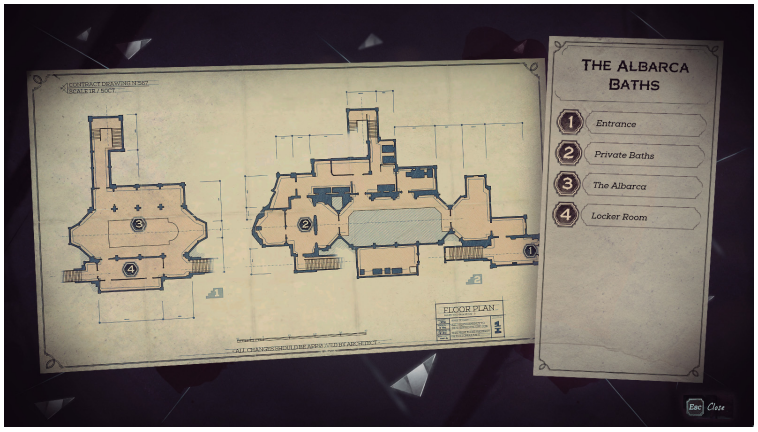


Figure 5.9 – Albarca Baths Map as shown in the game *Dishonored: Death of The Outsider*. Image taken from a Youtube video [58]

entrance in the original game is a window that leads to an empty bathroom, while the left entrance is a door that is originally closed but the player can look through the lock. In the main room, there are initially two enemies A and B talking with each other on the other side of the room. The player can hide either in the bathroom or behind a pile of rubble (near the left entrance) while the enemies are talking to each other. It is important to notice that while the enemies are talking, they are face to face, therefore no take down can be performed and the scenario cannot be solved as their FoVs block the entrance to the room where the goal is.

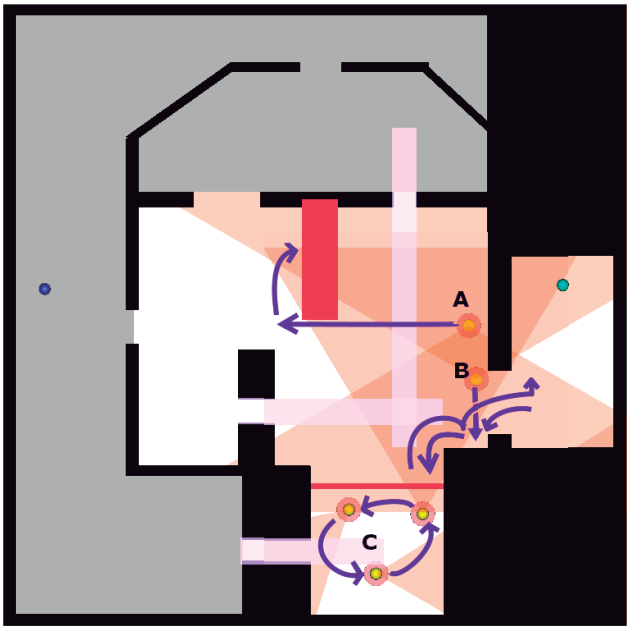


Figure 5.10 – Albarca scenario.

After a period of time (200 timesteps in our model) both enemies finish their conversation and start moving. Enemy A turns to the left and sits behind a desk facing the entrance of the goal room. Since the enemy can see anything beyond the desk, we model this element as a motion obstacle (shown as a big red rectangle), as he cannot move through it. B, on the other hand. turns facing the south and starts walking in a cycle, going to take a look at a cage containing 3 enemies, signalled by the letter C and the room where the goal is.

The enemies C inside the cage, however, are not human. In the original game they are hounds just walking inside the cage and watching over the other two enemies. Therefore, if the player gets

close to them they will alert the two guards. These three enemies present all the same behaviour, just walking around the cage, which we modelled as a cyclic path inside the southernmost alcove with an entrance covered by a motion obstacle that simulates the fact that the cage can be seen through but cannot be walked over.

(a) (b)

Figure 5.11 – Screens of the *Albarca Baths Kennels* scenario in *Dishonored: Death of the Outsider*. Images taken from a walkthrough video [58] [59].

The difficulty with this level lies in the need for the player to take down enemy B as its FoV covers the goal and evade enemy A while both are being watched over by the hounds C inside the cage.

To do this, the player has to make use of the environment of the scene. As shown in the screen of figure 5.11a. The main room presents a set of hanging pipes, which the player can use to climb on it and travel around the room without being noticed by A or B. Enemies C however, being hounds have an enhanced smell sense and if the player comes near to the cage, they will notice him, even if the player is on the pipes. We model this set of pipes as two vision obstacles shown in light purple, one coming straight from the bathroom and close to the hound's cage and the other starting near to the rubble pile and intersecting the first one on the other side of the room. These obstacles, do not allow for the enemies to notice the player, as long as he is inside their area, however, to take down any of them the player will have to step outside of them.

In the gameplays we used as our reference, some players are able to clear the scenario by good timing taking out B when A and C are not looking at him, while a number of other players took advantage of the elements in the scenario of the next room. This room is placed over the hound's cage and on one of the walls it presents a small vent situated right on the center of the cage, as

shown in figure5.11b. This vent while being too small to move through it, is big enough to shoot at the dogs when they are in the right place or to drop *hook mines* at the cage which can silently kill the hounds.

We modelled this element as a connection between the corridor and the cage alcove blocked by a vision obstacle that hinders the hounds FoV and ends at the center of the cage. Since these elements do not affect the motion behaviours, it is possible for the player to take down each hound by waiting for them to step *"below"* the obstacle. On the other hand, we modelled this element thin enough to present a difficulty for the player (RRT) to go through it, which, in a sense simulates the size of the original vent.

For the body hiding problem with preset hiding zones, we set all the corridor and bathroom space as the hidden zones as no gameplay shows the enemies being able to notice any of these spaces. Furthermore, players are able to stay in the bathroom space, right next to the enemy on the desk without being noticed by him.

5.2 Results and Discussion

In this section, we present the different set of tests that we performed on the scenarios previously described, and discuss the result. We split them into two sets, the first one performed on the 2D scenarios in order to study the impact of the different parameters and the second set ran on the 3D scenarios to analyze the success rate of each approach on a single configuration. All the experiments were run on a Windows 10 machine, using an Intel i5-2320 CPU operating at 3.0GHz, with 8 GB of RAM using Unity 3D 5.6.1f1.

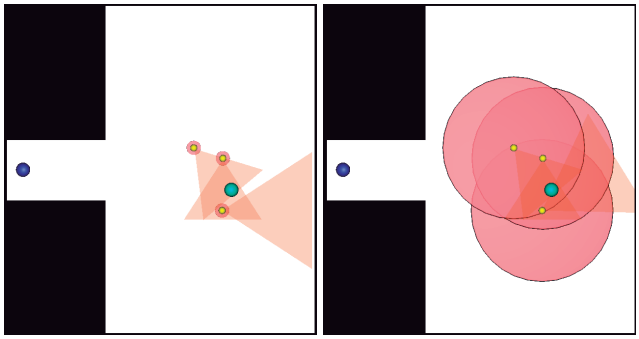
It is important to note that because our main interest is in studying the take down and body hiding behaviours, as well as the effect of each of the improvement approaches over the naive solution, all the tests shown in this section were performed applying a value of 100% for both the **take down probability** $\mathbf{p_k}$ and the **bias probability** $\mathbf{p_b}$. This means that whenever a node is sampled in a location that is valid to perform a take down, it will execute such action and every other node will try to bias towards a valid location, according to the take down method that is executed.

5.2.1 Take Down Parameter Tests

The initial test set was aimed at understanding under which configurations we could solve the basic scenarios, one created with a simple puzzle challenge and the second one based on the previous work by Borodovski [2], to compare the solution of its work based on distractions and our solution. The second objective of our test is to analyze the effect the two parameters that affect the take down mechanic: the radius in which the action can be performed (**take down radius** $\mathbf{r}$) and the **speed of the enemies** $\mathbf{v_e}$. The first is supposed to increase the ability to eliminate enemies and the second is supposed to increase the difficulty of performing such action.

Test Scenario

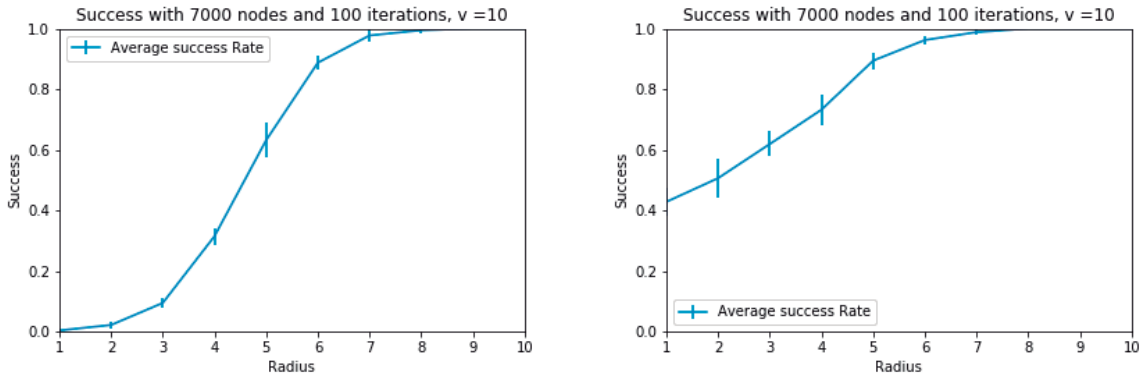
For the parameter tests on the Test Scenario we first analyze the impact of the **take down radius** $\mathbf{r}$ by running the naive search and all the proposed improvement approaches over the scenario with the speed set at a value of 10, which means that each enemy will move 10 times as fast as the player. We made 100 iterations of the RRT algorithm, each with a maximum of $7,000$ nodes. We examine radius values from 1 to 10 in steps of 1. Figure 5.12 illustrates the difference between both extreme values used for the test.



(a) $r = 1$ (b) $r = 10$

Figure 5.12 – Take down radius configurations for the parameter tests on the Test Scenario. The pink circles around each enemy show the size of the take down radius.

On these tests we measure the success rate of the different approaches over the 100 iterations. To better approximate success rate value we ran this test set 10 times. The results of the tests are shown in figure 5.13 for the *naive* and the *correction* approach.



(a) Naive approach

(b) Correction approach

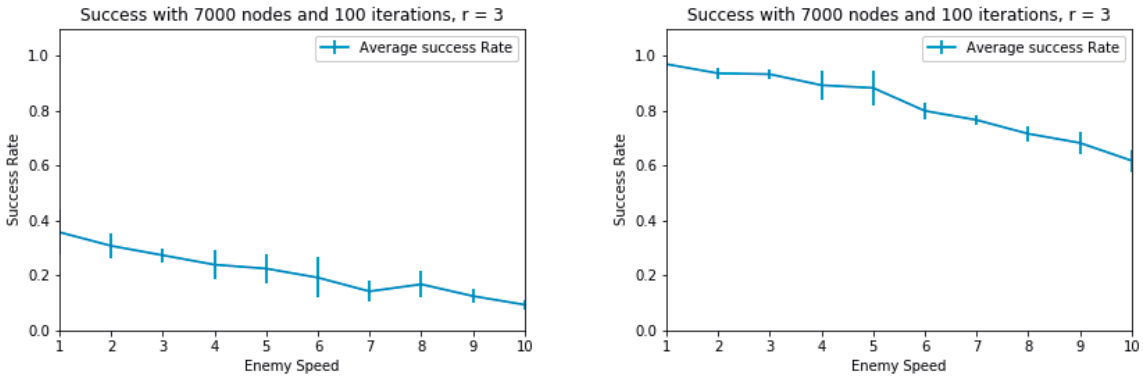
Figure 5.13 – Success rate for the different take down approaches on the Test Scenario with different take down radius settings.

Along with the average result for the test set, we show the standard deviation of the results as error bars.

As we can appreciate from this result, the *naive* approach is affected significantly by the size of the area where nodes have to be sampled to perform a take down action while the *correction* approach presents an improvement on the success rate of near 50 percent for a small action area, although this difference decreases as the value of the radius starts increasing, reaching a 100% success rate at a value of 7 compared with a value of 8 for the naive approach. However, it is important to remember that this test was performed with an enemy speed at its maximum value, presenting the greatest challenge possible for the scenario with respect to this value. It is important to note that we omit the results for both the *bias* and *prediction* approaches as they showed a 100 success rate invariant to all the configurations.

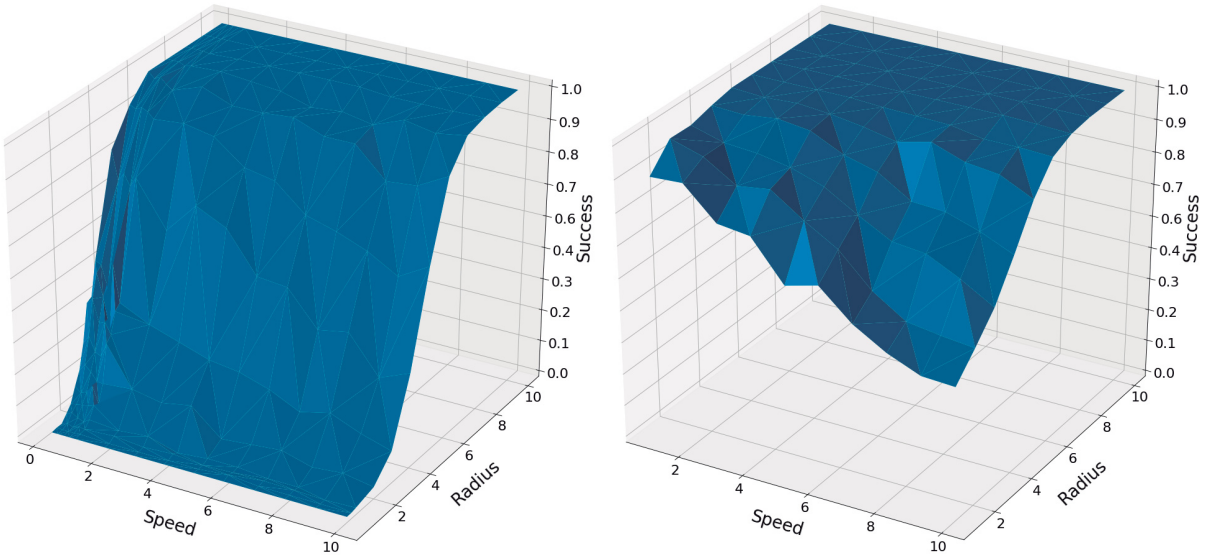
Given these results, we performed the opposite test by fixing a value for the **take down radius** r and iterating over the configurations for the enemy speed that ranges from 1 which is the same speed as the player and up to a value of 10 in steps of 1. We selected a radius value of $r = 3$ that representsan interesting increase in difficulty to the algorithms to solve the scenario. We performed the same set of tests obtaining the results shown in figure 5.14.

As illustrated, the speed of the enemies affects the take down algorithms by covering a greater area of the scenario with their FoV for a given time interval, thus requiring the algorithm to sample



(a) Naive approach (b) Correction approach

Figure 5.14 – Success rate for the different take down approaches on the Test Scenario with different enemy speed settings.



(a) Basic approach (b) Correction approach

Figure 5.15 – Average success for the take down approaches in the Test Scenario over the different parameter settings. Data taken over 10 sets of 100 executions of the algorithms on each configuration with a maximum of 7,000 nodes per execution.

nodes in a smaller portion of the state space in order to perform a take down. Here the speed of the enemies reduces the success of the algorithms by approximately 30% for the *naive* approach and up to 40% for the *correction* approach. This increased impact over the latter is related to the improvement being based on the FoV of a single enemy, not considering the position of the others. This causes the bias to result in an invalid node more frequently when the enemies have their take down radius guarded by another enemy. In this scenario that happens when the enemy that watches over the other two rotates faster with a higher speed value. Once again, for this sample scenario, the *bias* and *prediction* approaches shown a 100% success rate for any configuration.

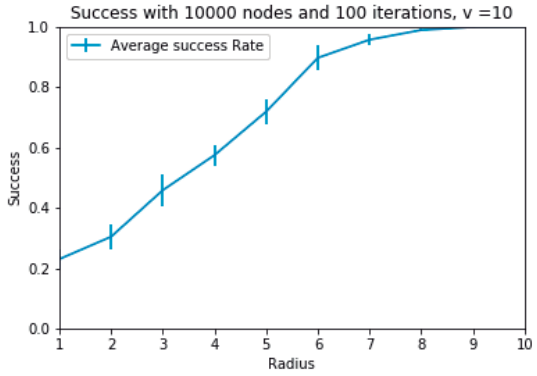
Figure 5.15 shows a 3D graph representing the results of the same test set ran over each pair of values for the **enemy speed** $\mathbf{v_e}$ and the **take down radius** $\mathbf{r}$ where we can appreciate a significant improvement over the *naive* approach.

Thief Scenario

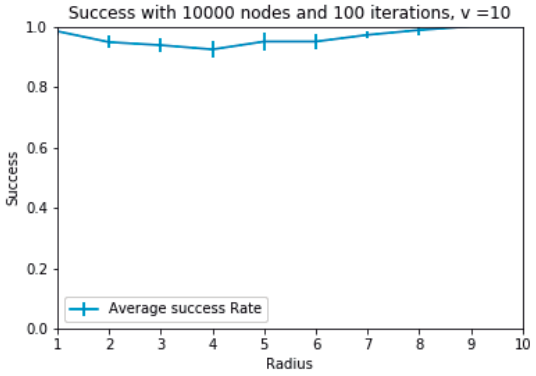
The second parameter test was aimed to analyze the effect of the same parameters discussed in the previous scenario but on an environment that is closer to what modern games present. To do this, we ran a similar set of experiments (10 sets of 100 executions of the algorithm with a limit of $10,000$ nodes for each of the parameter value), increasing only the number of nodes since the previous scenario was designed to only present a simple stealth puzzle while the others are based on real games and therefore present an increased difficulty.

While one of our goals was to compare our results to those of Borodovski [2] we did not have information about the speed of the enemies used in its experiments. However, assuming that the most basic configuration of **enemy speed** $\mathbf{v_e} = \mathbf{1}$ was used, we set our **take down radius** $\mathbf{r} = \mathbf{1}$. With this configuration, we obtained a success rate of 46% with the $naive$ approach for the test reported by Borodovski of 100 iterations with $2,500$ nodes and a single attempt per iteration. Compared with the results reported by the author for the distraction model, the success of our $naive$ approach is nearly half of it. Furthermore, our maximum time for finding a solution were $0.96s$ with an average of $0.61s$ while the original work presented an average of $0.1s$. However, for the models that build on the naive search the results show a 100% success rate for all of them with average times of $0.04s$ ($\sigma = 0.054$) for the *bias* approach, $0.06s$ ($\sigma = 0.098$) for the *prediction* approach and $0.07s$ ($\sigma = 0.113$) for the *correction* approach. It is important to note that there is a

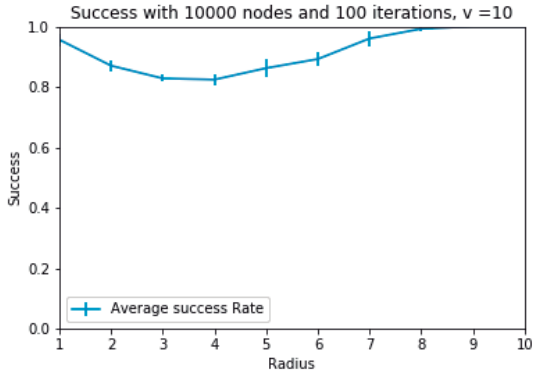
significant difference between the stealth approach we tested and that of Borodovski in which the aim was to solve the problem using distractions. The distraction model requires the algorithm to aim for static coordinates to trigger them, while the take down approach requires the algorithm to navigate towards certain areas that change position over time.



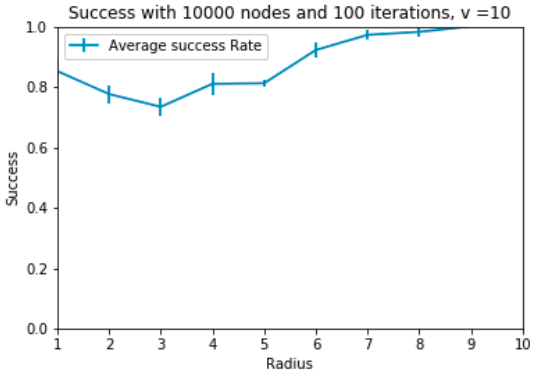
(a) Naive approach



(b) Biased approach



(c) Prediction approach



(d) Correction approach

Figure 5.16 – Success rate for the different take down approaches on the Thief Scenario with different take down radius settings.

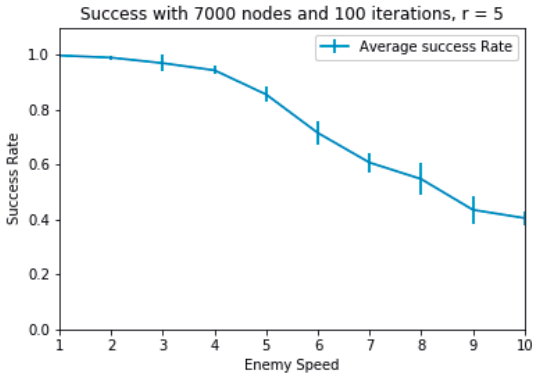
The second test set of Borodovski runs the algorithm for 100 iterations with 4000 nodes, however, this time 5 searches are attempted at each iteration. In this case, all our algorithms hit a 100% success rate, the same way as the distraction approach did. Comparing the times, Borodovski reports an average of $0.15s$ while our approaches reported $1.18s$ $(\sigma = 1.146)$ for the *naive* approach, $0.28s$ $(\sigma = 0.648)$ for the *bias* approach, $0.29s$ $(\sigma = 0.648)$ for the *prediction* approach and

$0.3s$ ($\sigma = 0.647$) for the *correction* approach. Comparing the tests of both approaches, we note that when a single search with few nodes is performed the improved approaches of the take down model achieve a 100% accuracy, better than the distraction model results. On the other hand, when more nodes and searches are allowed, both models achieve similar accuracy with the distractions model requiring less computation time. Given this result it could be hypothesized that when an automated agent is given a scenario of this nature, the fewer the number of node samples allowed, the more likely we are to find a solution by choosing to take down enemies instead of distracting.

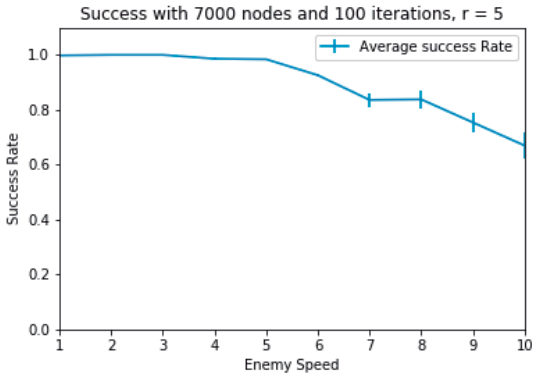
When increasing the search space to match those of the test performed in the Test Scenario we found a 100% of success in all the approaches even when the enemy speed is set at maximum. After analyzing the scenario, we hypothesized that this invariance comes from the fact that the only enemy affecting the difficulty of the scene is enemy B shown in figure 5.2 on page 88, as it is the only enemy that can block the path to take down enemy C. Since this enemy is moving around a big portion of the scene, it leaves the path for the solution open for long time intervals. To avoid using even higher values for the enemy speed that will result in enemy behaviours that are unlikely to be present in games (enemies moving too fast to be tracked by human players) we decided to tackle this by the opposite angle: reducing the speed of the player so that it will move in shorter path segments for a given time interval, thus the RRT connections will be shorter. By setting this speed to a value of $\mathbf{v_p} = \mathbf{0.5}$ we can study the parameters effect on the algorithm.

Figure 5.16 shows the results of the **take down radius r** effect on the algorithms when an enemy speed is set to the highest value. As one can appreciate, the *naive* approach success changes in proportion to the take down radius size as expected by an incremental area that increases the probability of sampling nodes where a take down action can be performed. The *prediction* and *bias* approaches however seem to be inversely affected by the radius size. This is because the enemy that is the key to solving the problem is positioned close to a wall, therefore, an increase in the size of the radius leads to a greater portion of the take down area overlapping with the level boundaries: the approaches that try to bias the search into a random point of the radius that is behind the enemy will end in an attempt to connect a node that is blocked by obstacles. This result shows that in contrast with what intuition might suggest, the greatest value for the take down radius is not always the best choice, but rather this optimal value will depend on the design of the scenario. Improvements to how points in the take down radius are sampled may mitigate this effect.

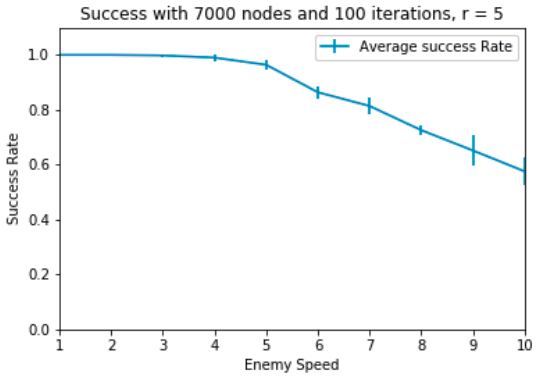
Figure 5.16d shows an interesting phenomenon derived from the search bias introduced by the *correction* approach. While it tries to bias the nodes that fall inside a FoV, in this scenario the greatest FoV area corresponds with enemy A of figure 5.2, therefore, most of the attempts to bias the search will try to find a spot to take down this enemy which by design is set up in an alcove that cannot be accessed by the player. Therefore, most of the attempts to bias the search will end up in invalid nodes. However, this approach still shows an improvement over the results presented by the *naive* approach.



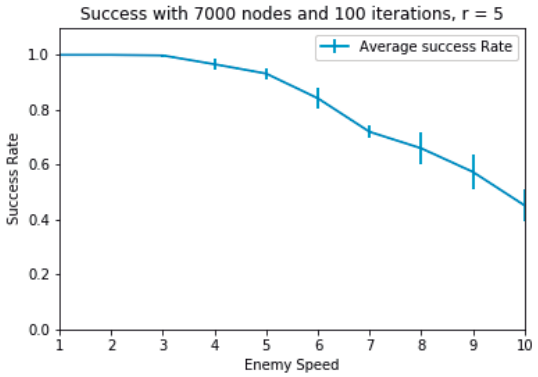
(a) Naive approach



(b) Bias approach



(c) Prediction approach

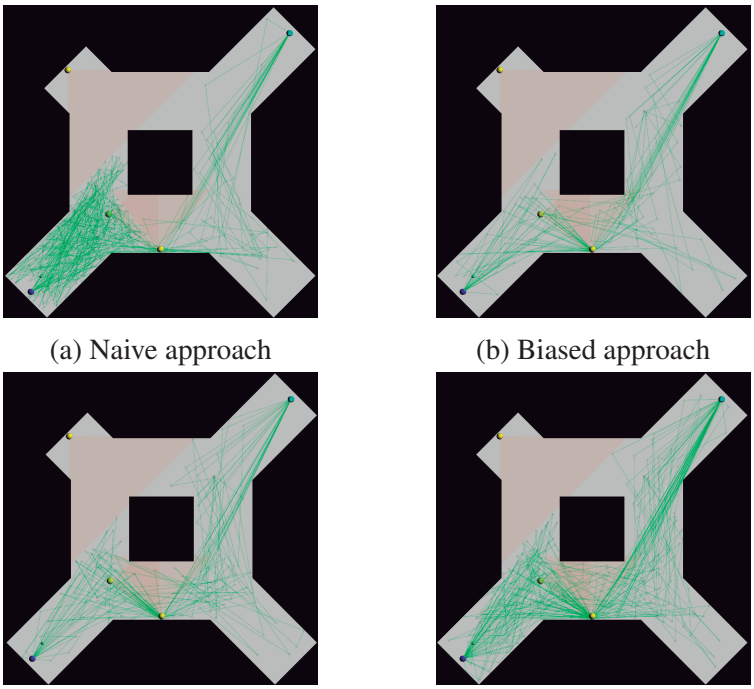


(d) Correction approach

Figure 5.17 – Success rate for the different take down approaches on the Thief based scenario with different enemy speed settings.

Given the results of these tests, we selected a fixed value for the take down radius that presented a significant success rate for the tests but not a 100% in order to study how the value of the enemy

speed can affect the ability to find a solution. We selected a value of 5 to perform the same test set over different configurations of the **enemy speed** $\mathbf{v_e}$. The results are shown in figure 5.17, where we can observe that the enemy speed only seems to reduced the success of the algorithm significantly when using high values that would represent almost 10 times the speed of the player when this has been reduced to half of the original used on other scenarios.



(a) Naive approach (b) Biased approach

(c) Prediction approach (d) Correction approach

Figure 5.18 – 30 Solution paths for the Thief Scenario using each approach.

Figure 5.18 illustrates the solution paths found for 30 iterations of the algorithm for each of the approaches. We can observe that the *naive* approach generates the densest space of all the approaches, sampling a larger number of nodes for each solution. In contrast, the *biased* approach generates the least dense space; furthermore, it also generates more similar solutions, from which we can hypothesize that this method is the one that most severely restricts the exploration on its solutions.

5.2.2 Success on Complex Scenarios

In this set of test, we aim to analyze the performance of the different RRT approaches when solving the take down problem on scenarios that include elements that emulate properties of 3D scenarios in a 2D representation. These scenarios, due to their nature and number of elements, present a higher difficulty than those used for the parameter testing. We therefore focus in this section on analyzing the success rate of the different approaches over the same configuration.

Aventa Station

To analyze the success rate on the Aventa Station Scenario we performed 10 sets of 100 executions of the RRT algorithm with a maximum of $10,000$ nodes for each of the different approaches. We set the **enemy speed** $\mathbf{v_e} = \mathbf{1}$ as well as the **take down radius** $\mathbf{r} = \mathbf{1}$ for all the iterations. After the executions, we analyze the number of solutions found for every 100 executions and observed the number of nodes the algorithm had to sample before finding a solution, as well as the time it took for each iteration to be completed.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	Std. Deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Random	19.4%	0.0293	5713	728	9998	464	2.64	0.141
Bias	47.8%	0.0487	5360	301	9969	646	2.27	0.138
Prediction	47.3%	0.0438	5615	452	9978	475	2.43	0.131
Correction	37.5%	0.0571	5256	526	9987	358	2.62	0.143

Table 5.1 – Results for the Aventa station scenario over 10 experiments with 100 iterations using a maximum of 10,000 RRT nodes

Table 5.1 shows the results of the execution, where the first point that comes out is the significant difference in the success rate of the approaches compared to the results obtained on the 2D scenarios. While the previous scenarios were solved closely to 100% of the times by some approaches, here the success is around 47% for the methods with the best performance. From the results of the 2D scenarios we hypothesized that the greatest impact on the success came from the value of the **take down radius** $\mathbf{r}$, however when testing higher values the results did not improve

in a significant way as it did for the 2D Scenarios. Therefore, we hypothesized that the reason is not because the player is unable to approach the enemies, but rather the density of enemies in the left section of the scenario (shown in figure 5.6) which creates smaller time intervals in which the player can get into the area and leave.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	std. deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Without Enemy A								
Random	78.4%	0.0471	4752	172	9980	245	1.70	0.109
Bias	98.1%	0.0132	3007	175	9638	189	0.83	0.063
Prediction	97.2%	0.0193	3197	226	9505	139	0.95	0.092
Correction	93.6%	0.0185	3646	286	9915	169	1.23	0.071
Without Enemy B								
Random	40.2%	0.0159	5256	358	9969	579	2.81	0.152
Bias	73.0%	0.0363	4366	341	9991	498	1.97	0.020
Prediction	76.8%	0.0305	4467	253	9996	842	2.05	0.070
Correction	66.6%	0.0338	4865	251	9919	311	2.45	0.095
Without Enemy C								
Random	37.4%	0.0185	5673	245	9982	555	2.51	0.063
Bias	60.2%	0.0342	4798	186	9989	812	1.93	0.077
Prediction	61.2%	0.0440	4891	344	9969	544	1.99	0.029
Correction	51.9%	0.0303	4755	352	9966	488	2.54	0.206

Table 5.2 – Results for the Aventa Station scenario when an enemy is removed from the design.

To test this hypothesis, we extracted one of the enemies in the scenario who converge in the upper left section of the scene, which is the section where more FoVs intersects; these FoVs correspond to enemies A, B and C. We performed the same set of tests on each of these new configurations, obtaining the results shown in table 5.2. As we can see, just by reducing the number of enemies by one, we are able to at least double the success rate of the search. Furthermore, by removing either A or B we obtain an increase of more than three times the original rate. This is due to both enemies guarding each other, as enemy A almost always blocks the path to get near enemy B, while the FoV of B covers A during half of its trajectory, leaving a small interval of time for the player to take down A, made even smaller when considering that its position is also

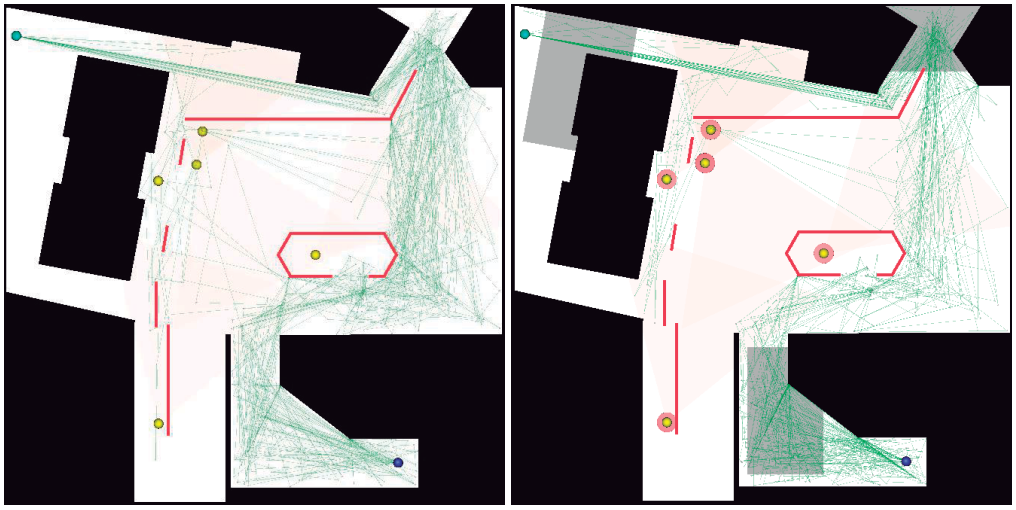
covered at some times by B or D. On the other hand, the effect of removing C is not of the same magnitude as the other cases since the path of enemy C is only guarded at some small intervals by B. Furthermore, when looking at the solutions generated by the RRT search shown in figure 5.19 we can see that the search tends to generate a considerable number of the solutions going through the corridor guarded by C, which indicates that this enemy was the easiest to deal with.

In order to keep the original structure and elements for the scenario, we decided to check the effect of lowering the enemy speed. This led to an increase in the success rate, almost two times the outcome of the initial results. While the previous hypothesis of removing an enemy required modifications to the design of the scenario, lowering the speed on the other hand, does not lead to a redesign. Games such as *Assassin's Creed* uses this lower speed configuration for generating some enemies that are not as faster as the player, to give the chance of escaping from difficult situations.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	std. deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Random	24.9%	0.0452	5662	418	9983	704	3.22	0.246
Bias	60.89%	0.0375	4864	137	9958	326	3.36	0.321
Prediction	56.4%	0.0443	5261	311	9993	474	3.01	0.163
Correction	46.5%	0.0387	5090	351	9990	530	3.21	0.099

Table 5.3 – Results for the Aventa station scenario for the Body Hiding problem over 10 experiments with 100 iterations using a maximum of 10,000 RRT nodes with the *preset* approach.

A more interesting phenomenon was observed when using the original configuration to test the body hiding problem solution with preset hiding zones. Contrary to intuition, this added challenge led to an increase in the success rate for the scenario of almost two times, as shown in table 5.3. After analyzing the solutions of the algorithms for the take down problem and the position of the hidden zones, we can see that an important portion of the solutions aim for the less guarded space in the corridor on the upper portion of the scenario, as illustrated in figure 5.19, at the same time these results overlap with the position of the hidden zones. This could be related with the bias causing the search to lead into the hiding zone, in this case as these zones are positioned in places that coincide with most of the solutions, they act as a sort of key point for generating solution paths which are more likely explored because the bias leads to these zones.



(a) Take down (b) Body hiding

Figure 5.19 – Heatmap of 30 solutions for the Aventa Station scenario for the *bias* approach

To test this hypothesis, we removed the upper right hiding zone, which led to a decrease in the success for the body hiding problem to less than 10% as in such case the player can only hide bodies near to the beginning of the level and close to the end of it. One last test was to instead of removing, trying to boost the exploration on another route by changing the upper hiding zone to be near to the area below the bottom-most enemy. This change also led to a decrease in the success rate as it does not boost the probability of dealing with the enemies in the densest area where guards keep on being guarded by each other. This is contrary to what the original configuration does by leading the player into a spot where timing only needs to be concerned with the enemy that guards the upper corridor.

These results show that it is feasible to perform a design analysis of game scenarios using a search like the one we propose, as it could show how difficult to solve a scenario could be when enemies are too dense and how placing some secondary elements in scenarios that are too complex could give a hint to the player about a possible approach to solve it.

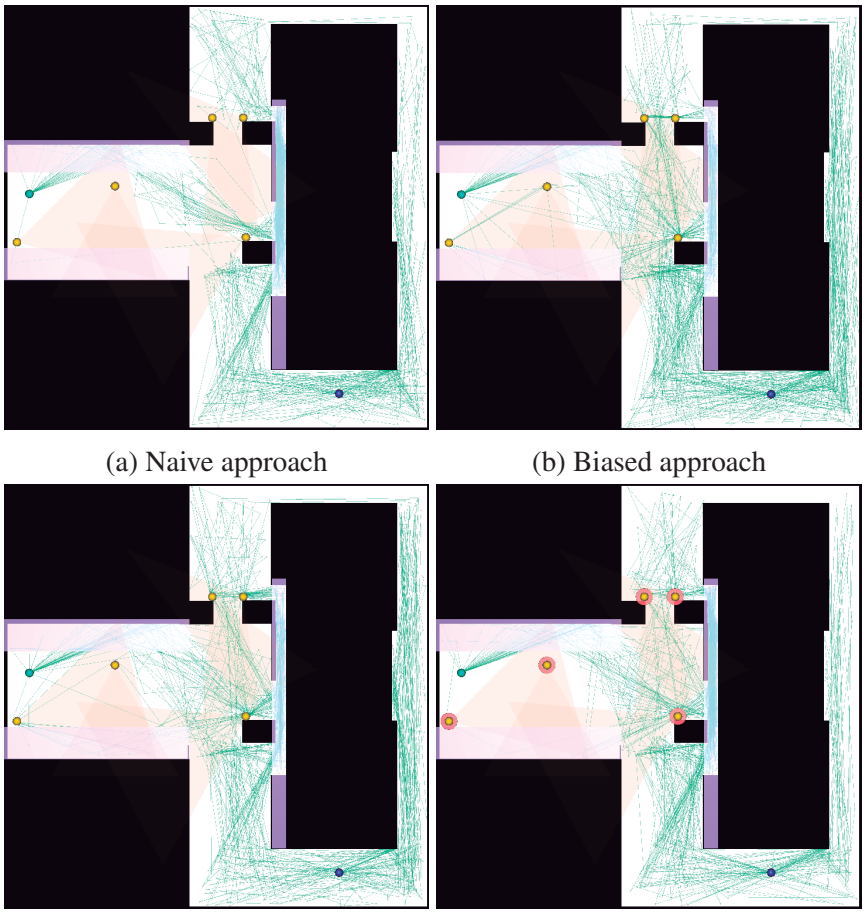
Aventa Station Alley

Although more complex than the 2D scenarios and although presenting the same number of enemies as the Aventa Station scenario, this scenario presents the opposite case with respect to the elements and behaviours. In the Aventa Station, all the 3D added elements acted as a disadvantage for the player, as they hindered actions but did not affect the enemies. In contrast, this level adds elements that act as an advantage for the player as they only modify the sight of the enemies. Therefore, although more complex, the results of the basic configuration ($\mathbf{r} = \mathbf{1}$ and $\mathbf{v_e} = \mathbf{1}$) on this level show a good performance for all the improved approaches, with the *Correction* approach most affected by this increase in the enemy density.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	std. deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Random	35.9%	0.0452	4836	253	9905	269	4.07	0.284
Bias	93%	0.0225	2590	171	9959	84	1.09	0.080
Prediction	94.4%	0.0080	2491	209	9938	69	1.03	0.081
Correction	69.2%	0.0346	3920	277	9985	107	2.58	0.128

Table 5.4 – Results for the Aventa Station Alley scenario over 10 experiments with 100 iterations using a maximum of $10,000$ RRT nodes.

As shown in table 5.4, this time the most successful approach was the *prediction* method, this is due to the enemies on the scenario all presenting a static behaviour, allowing for this method to perfectly predict the blind spots of the enemies. Figure 5.20 shows the solution paths obtained from the execution of 100 searches for each of the approaches. As we can observe, both the $naive$ and $correction$ approaches tend to be less likely to find a solution that involves taking down all the enemies, instead, they use the 3D elements to avoid the FoV of both enemies guarding the ending point and go directly to it. On the other hand, the $bias$ and $prediction$ approaches tend to explore more often the corridor on the right side to get behind the enemies on the upper portion of the scene, while the other two approaches instead get to that point using the 3D elements.



(a) Naive approach (b) Biased approach

(c) Prediction approach (d) Correction approach

Figure 5.20 – 40 Solution paths for the Aventa Station Alley scenario using each approach.

Albarca Baths

This scenario, being the most complex in our test set, presents both types of added 3D elements: obstacles that hinder the enemies vision and obstacles that block the player path. This scenario also presents a high density of enemies, similar to the previous scenarios but adding a larger FoV to all of them. Furthermore, the only way to get closer to three of the five enemies is by using a small corridor with a 3D obstacle at the bottom of the scenario. The size of this element added to the portion of the scenario taken by the corridor and the bathroom makes the player less likely to sample nodes along the 3D elements that could help to solve the scenario.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	std. deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Random	5.3%	0.0236	4519	1137	9838	129	5.21	0.142
Bias	22.7%	0.0371	5289	777	9989	172	4.49	0.193
Prediction	18%	0.0236	5524	654	9912	219	4.75	0.107
Correction	10.7%	0.0200	4396	600	9771	187	5.15	0.109

Table 5.5 – Results for the Albarca baths scenario over 10 experiments with 100 iterations using a maximum of 10,000 RRT nodes

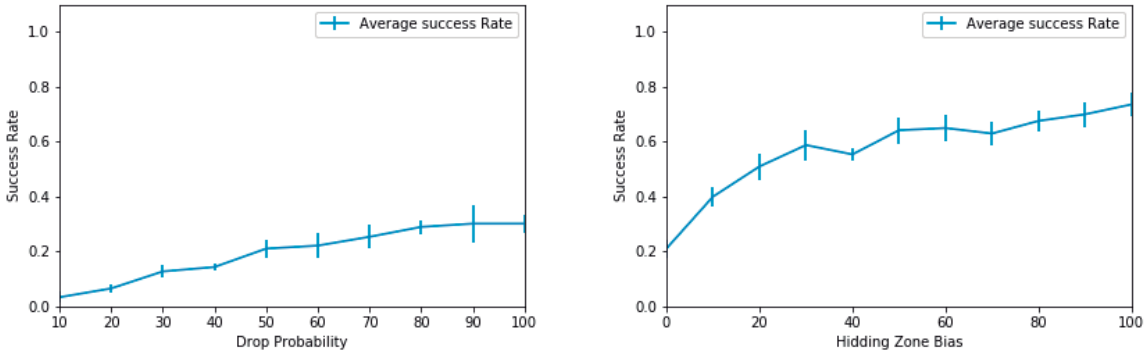
Table 5.5 shows the results of the test set applied to this scenario which included 10 executions of 100 searches with a limit of $10,000$ nodes for each of the approaches. As we can see the increased complexity on this level severely reduces the success rate of the search algorithm. However, increasing the size of the take down radius or reducing the speed of the enemies allows for the search to obtain a better success rate while still being close to the original level in the game, where the player has access to mid-range weapons such as bombs and darts and the enemies on the scene tend to wander slowly along the scene if they are unaware of the player being close.

5.2.3 Body Hiding Parameter Tests

In this section, we aim to analyze the performance of our proposed solution for the body hiding problem. While this task elaborates on the take down problem adding constraints to the solution, we aim to test the same scenarios we used for the take down problem to see how much this added

constraint impacts the success rate we obtained in the previous tests. Therefore, following the same approach as we did for the take down problem we divide the experiments into two sets, the first one focused on observing the influence of both of the parameters that control the body hiding search: the **hiding zone bias probability** $\mathbf{p_b}$ which determines the likeness of biasing the node sampling towards a hidden zone and the **body drop probability** $\mathbf{p_d}$ which controls how likely the search will be to generate paths where the player carries the bodies for longer times.

Test Scenario



(a) Body Drop Probability

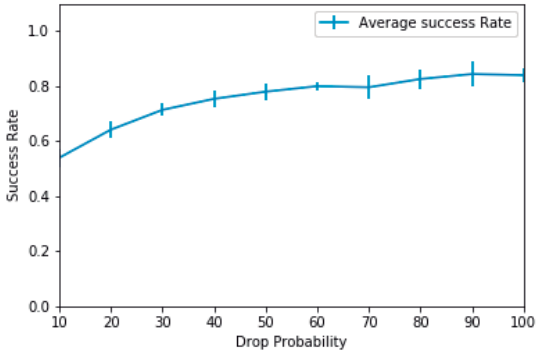
(b) Hiding Zone Bias

Figure 5.21 – Effect of the body hiding parameters in the success of the *Preset* hiding zones for the Test Scenario using a *naive* search.

Preset Zones The same way as we did for experimenting with the take down parameters, we decided to use the 2D scenarios to check the success rate of the different algorithms over a range of values for the parameters involved in the body hiding problem. Since the **enemy speed** $\mathbf{v_e}$ and **take down radius** $\mathbf{r}$ have been studied in previous tests, we set a fixed value for this two parameters in order to isolate the effect generated by the parameters related to the body hiding problem. We choose a value of 1 for the **enemy speed** $\mathbf{v_e}$ as the previous test in this scenario shown this value allows for all the take down approaches to be able to find solution paths. On the other hand, we set the value for the **take down radius** $\mathbf{r}$ to 3 to compare the results with those obtained in the experiments of the take down problem.

To test the effect of both body hiding parameters we fixed the value of one of them to isolate the effect of the others. We began by fixing the value of the **hiding zone bias probability** $\mathbf{p_b = 0}$ to have a scenario where the action of moving into a hidden zone is only based on the naive sampling of the RRT. Since the algorithm does not allow taking down a second enemy before hiding the body of the first, no other bias will be performed on the search after taking down an enemy and before hiding the body. For each of our tests, we executed 100 iterations of the algorithm with a maximum of $10,000$ nodes for each of the values of the parameters in test and for each of the take down and body hiding methods. We repeated this test 10 times to calculate the average and standard deviation of the results.

In the next tests using the *preset* hidden zones shown in figure 5.1b on page 87, we obtained results similar to those of the take down problem, in which only the *naive* approach was unable to find a solution for each execution. The results shown in figure 5.21a allows us to see that the success in a body hiding problem is improved when the player can perform a body drop more frequently, this comes from the fact that the naive sampling will not always select nodes inside a hidden zone, therefore, it is better to have a high body drop probability to take advantage of the times that such nodes are selected.



(a) Naive approach

Figure 5.22 – Effect of the body drop probability in the success of the *Learned* Body Hiding for the Test Scenario.

After the previous test, we set a fixed value for the **body drop probability** $\mathbf{p_d}= 0.5$ and executed the same tests iterating over the values for the **hiding zone bias probability** $\mathbf{p_b}$ obtaining

again a 100% success rate for all the take down approaches but the *naive* take down. The results in figure 5.21b show that, as expected, biasing the search towards a hidden zone increases the probability of success as the search does not have to wait until a sample is taken from inside these areas.

Learned Zones We repeated the same process described above but this time using *learned* hidden zones. That is, the scenario begins with no hidden zones and allows for the player to drop the bodies at any point creating new potential hidden zones. A potential hidden zone will be rejected if at some future time, when another body drop is tried at this place it is found to be intersected by an enemy FoV. For these tests, we reset the learned hidden spots after each set of 100 executions, for each experiment to begin under the same conditions.

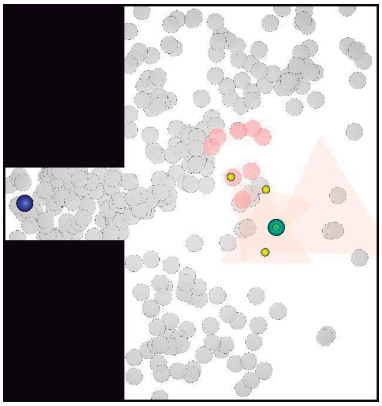


Figure 5.23 – Hidden zones found after 100 iterations in the Test Scenario.

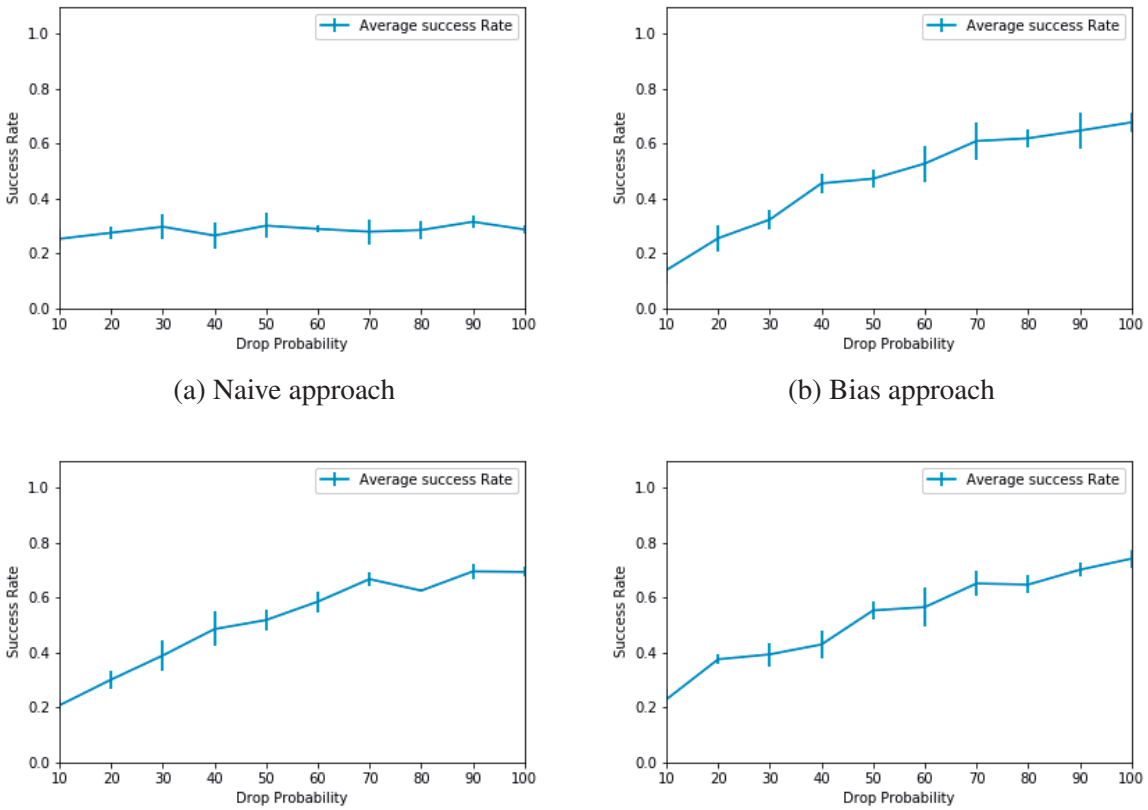
Figure 5.22 shows the results obtained when iterating over the values of the **body drop probability** $\mathbf{p_d}$. Here the preference of higher values for the parameter can be seen again even although the sizes of the hidden zones for this approach and those that were *preset* are different. These results also show how for this scenario, where there is a considerable space that is never seen by the enemies, a learned approach presents a better performance as the search does not depend on sampling a node inside a fixed zone as with the *preset* approach.

The results obtained for the second test set analyzing the effect of the **hiding zone bias probability** $\mathbf{p_b}$ show a minimal difference between the results for the different values, pointing towards this value having no effect when using *learned* hidden zones as biasing towards these zones which

are randomly discovered has the same performance as simply keeping on exploring new potential zones. However, this might only be true for this scenario because of the proportions of its free space.

Figure 5.23 shows in grey the areas detected as possible hidden zones after 100 executions of the search. The red zones represent possible hidden zones that were discarded for intersecting a FoV during an attempt to drop a body there.

Thief Scenario



(a) Naive approach

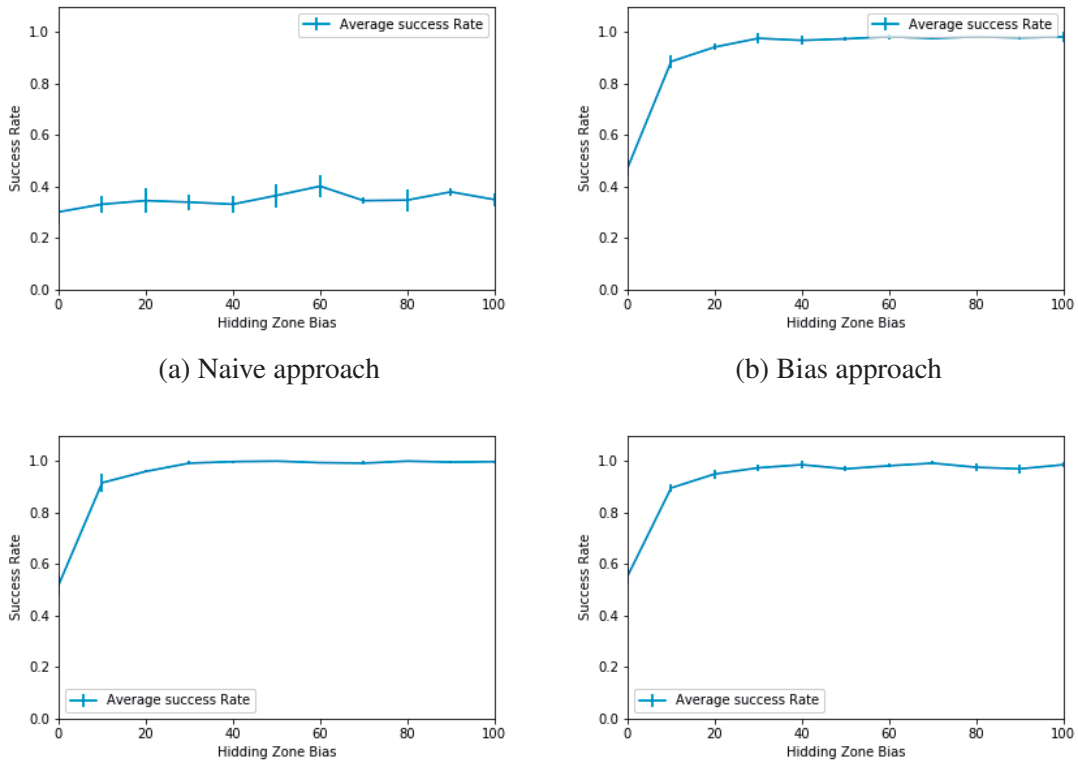
(b) Bias approach

(c) Prediction approach

(d) Correction approach

Figure 5.24 – Effect of the body drop probability in the success of the *Preset* Body Hiding for the Thief Scenario.

To confirm the effect of the body hiding parameters observed in the Test Scenario, we performed the same set of tests for the Thief scenario. This time, however, we increased the complexity by reducing the **player speed** $\mathbf{v_p = 0.7}$, the same way as we did for the parameter tests in the take down problem and increasing the **enemy speed** $\mathbf{v_e = 4}$ while setting a small value for the **take down radius** $\mathbf{r = 1}$. This is done to avoid all the approaches reaching their best performance, since using a basic setting where all the speeds are set to a value of 1 makes the scenario solvable by all the approaches 100% of the time.



(a) Naive approach

(b) Bias approach

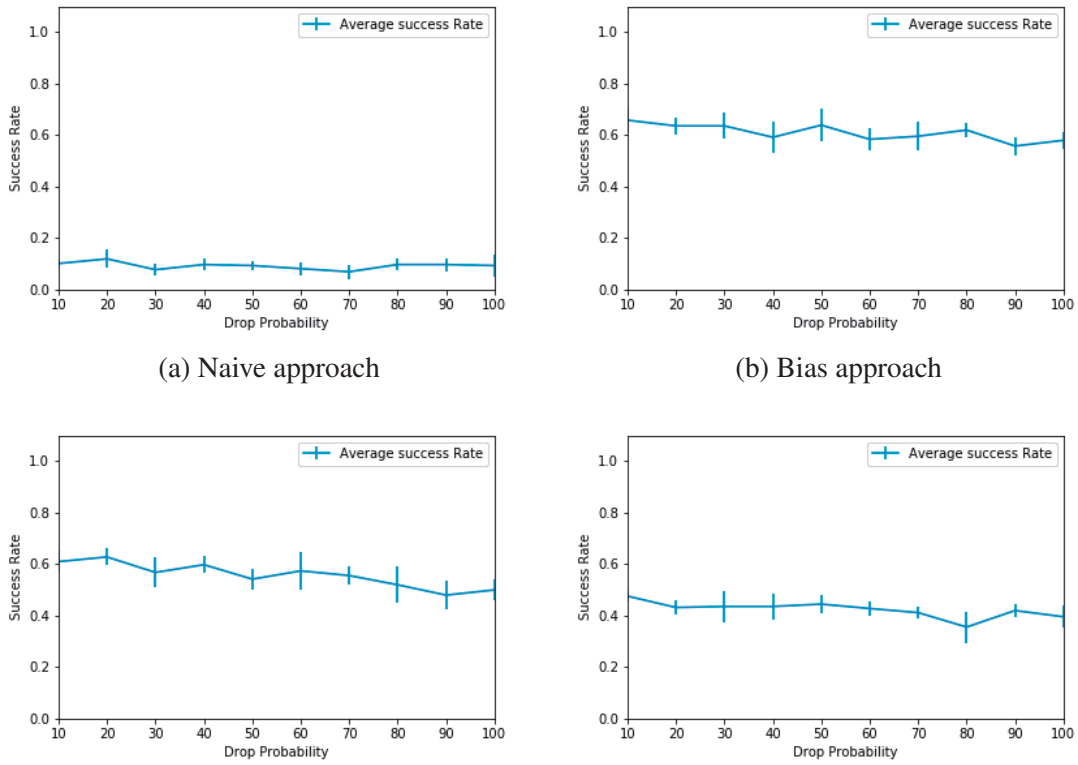
(c) Prediction approach

(d) Correction approach

Figure 5.25 – Effect of the hiding zone bias in the success of the *Preset* Body Hiding for the Thief Scenario.

Preset Zones Figure 5.24 shows the results obtained when fixing the **hiding zone bias probability** $\mathbf{p_b = 0}$ to base the navigation towards these zones entirely on the random sampling of the

RRT. It is interesting to see that while the success rate of the *naive* approach grows less than 5% as the value of the **body drop probability** $\mathbf{p_d}$ increases, all the other approaches increase their performance up to 50% when the body drop is performed as soon as a hidden zone is reached. Unsurprisingly, for a stealth scenario where the hidden zones are set in alcoves the enemies cannot access, the best action for a player is to take down and drop the enemy body immediately upon entering one the alcoves to avoid carrying it around while trying to hide from the other enemies.



(a) Naive approach

(b) Bias approach

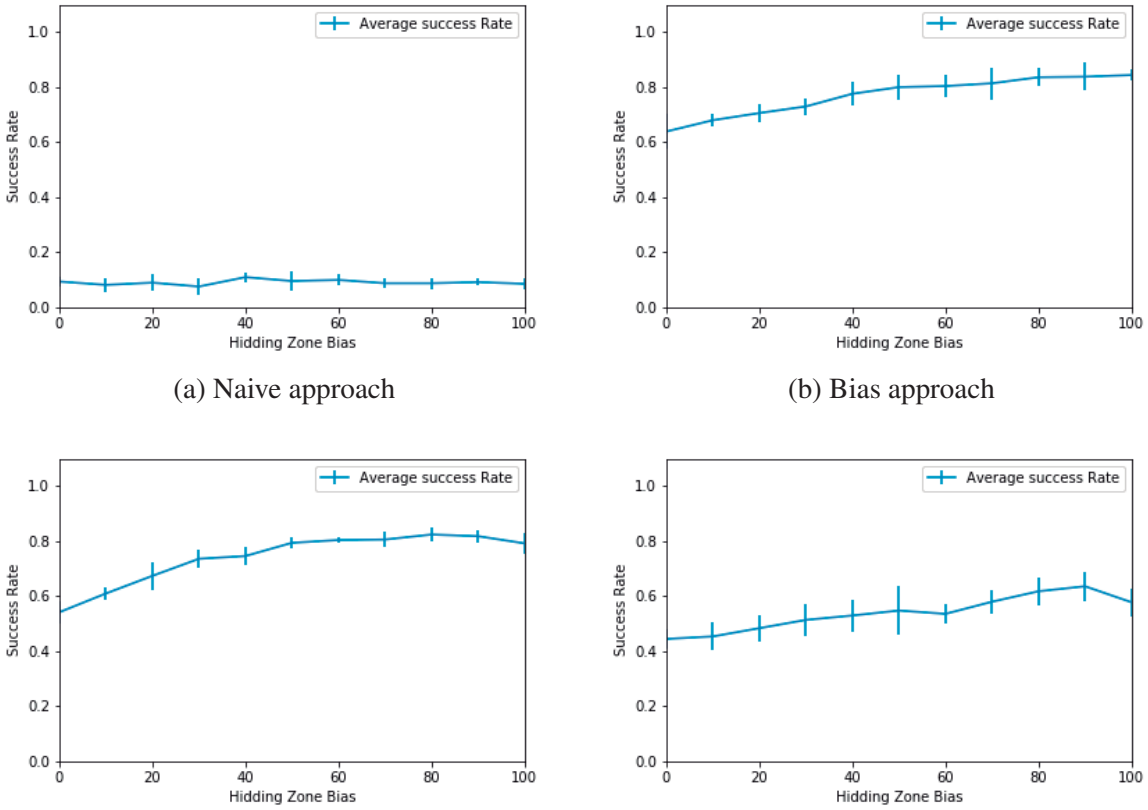
(c) Prediction approach

(d) Correction approach

Figure 5.26 – Effect of the body drop probability in the success of the *Learned* Body Hiding for the Thief Scenario.

Figure 5.25 shows results when adjusting the **hidden zone bias probability** $\mathbf{p_b}$.As shown, all the methods that build on the naive approach improve their success, with the bulk of the improvement (up to 40%) coming from a bias towards a hidden zone with a probability of 0.1. Although the hidden zone is just an alcove of the scenario, and a small area of the total level scene, even a

small bias probability has a major impact on success.



(a) Naive approach

(b) Bias approach

(c) Prediction approach

(d) Correction approach

Figure 5.27 – Effect of the hiding zone bias in the success of the *Learned* Body Hiding for the Thief Scenario.

Learned Zones When running the same tests for the *learned* hidden spots we found that the **body drop probability** p_d effect is different than how it was for the preset zone. This time for any value of the parameter, the success rate stays around the same values, due to almost all the solutions focusing on taking down the static enemy at the bottom part of the scenario and going directly to the goal, which is reachable from almost any point inside the take down radius of the static enemy. Results are shown in figure 5.26. However, we see a small reduction in the success for high values of the parameter, which comes from the fact that the higher the probability to quickly drop a body

the more likely it is to drop a body in sub-optimal places, such as a corridor that is being patrolled by the enemy, and therefore it is more likely to be detected at some point, blocking expansion of that RRT branch.

The **hidden zone bias probability** p_b on the other hand has an effect where the success rate is more proportional to its value, improving when a high probability value is combined with any of the take down approaches, as shown in figure 5.27. In the graphs however, we can see that best performance is actually reached slightly below 1.0, between 0.8 and 0.9, which leaves some probability for exploring new hidden zones further in the scenario.

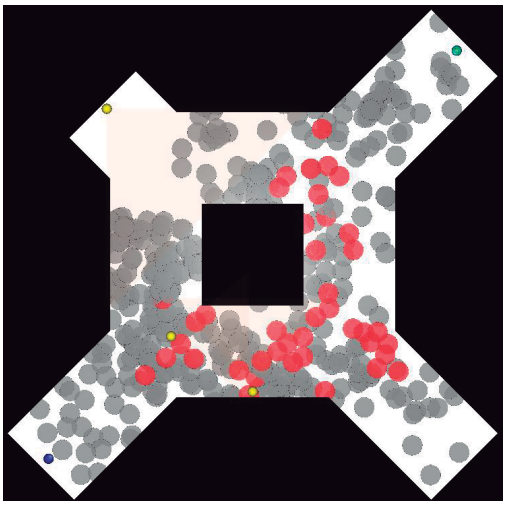


Figure 5.28 – Example of the hidden zones learned after the parameters tests in the Thief Scenario

Similar to the effect observed when using *Preset* hidden zones, the naive approach shows almost no difference in its performance independently of the value that is used for this parameter. In this scenario, success under the naive approach is dominated by the small **take down radius**

Figure 5.28 shows in grey the zones found to be safe to drop a body in the scenario. While the zones that coincide with the patrolling route of the enemy might be seen as incorrectly classified, some of them correspond with places where this same enemy can be dropped and therefore will never be detected as exposed to a FoV. On the other hand, the spots on the right side of the scenario can be connected to the ending point after dropping a body, ending the task before the enemy patrolling is able to detect a body.

5.2.4 Success on Complex Scenarios

After analyzing the effect of the body hiding parameters in the solution search we performed another set of experiments on levels that incorporate elements that emulate 3D mechanics in order to review how effective the proposed solution is to solve such scenarios. For these experiments, we run 100 executions of the search with a maximum of $10,000$ nodes with each of the take down and body hiding approaches over a fixed configuration for both the **body drop probability** $\mathbf{p_d}$ and the **hidden zone bias probability** $\mathbf{p_b}$, as well as for the take down parameters discussed in previous sections.

Aventa Station

For this scenario we ran the set of tests fixing an **enemy speed** $\mathbf{v_e = 1}$ and **take down radius** $\mathbf{r = 1}$, **hidden zone bias probability** $\mathbf{p_b = 1}$ and **body drop probability** $\mathbf{p_d = 1}$.

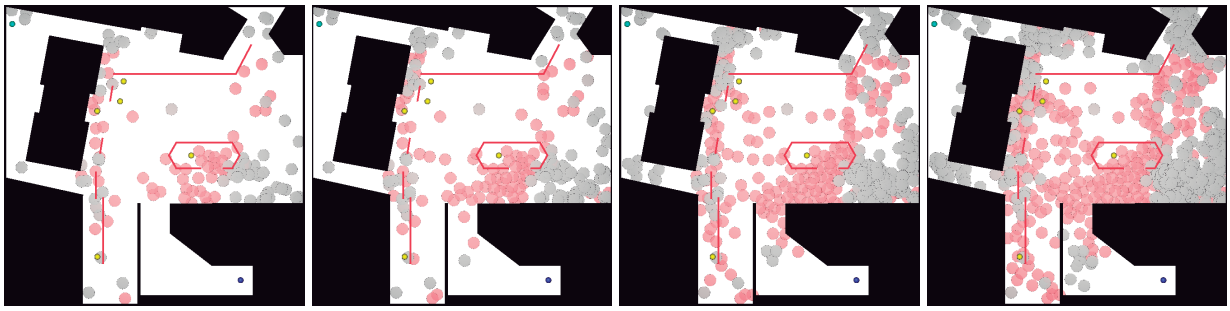
As discussed before, the results of this test set using the *preset* hiding zones (shown in table 5.3 is interesting as, contrary to intuition, it improves on the plain take down results by using the hidden zone bias to help bias the search towards the easiest route to the solution.

	Success Rate		**Nodes in Solution**				**Time per Iteration (s)**	
	mean ($\overline{x}$)	Std. Deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Random	23.6%	0.0372	5818	611	9969	506	3.01	0.148
Bias	44.7%	0.0563	5419	306	9957	589	3.04	0.123
Prediction	42.1%	0.0576	5645	338	9998	1023	3.32	0.179
Correction	34.6%	0.0482	5197	368	9999	666	3.49	0.171

Table 5.6 – Results for the body hiding problem with *learned* hidden zones in the Aventa Station scenario.

For the *learned* hidden zones, however, the improvement is only effective for the *naive* take down. The other approaches have similar performance to that shown previously in the take down problem. The results of the test set are shown in table 5.6. As expected, this average success rate is not as a good as with preset zones. The algorithm starts with no knowledge of valid hidden zones,

which makes the algorithm more prone to fail on its first iterations, until more valid safe zones are found.



(a) 100 iterations (b) 200 iterations (c) 500 iterations (d) 1000 iterations

Figure 5.29 – Convergence of the learned hidden zones in the Aventa Station scenario.

This scenario, however, better shows the evolution of the *learned* hidden zones, since it was created based on the elements of a real game that actually shows areas where bodies can be hidden. When we look at figure 5.29, which shows how the learned zones evolve through a number of iterations, we can see that the areas found as hidden by this method (shown in grey) start to converge to the same zones used in the *preset* scenario (shown in figure 5.6b. Furthermore, as iterations keep on going the algorithm starts to generate two different areas in the map, the area that is seen by enemies at some point in time (shown by the exposed spots in red) and the area that is never seen by any enemy (shown by the hidden zones in grey).

Aventa Station Alley

This scenario shows an interesting result compared with the outcome of previous scenarios. As shown in table 5.7, in this case, the success rate of the *learned* hidden zones approach improves on the results for the *preset* zones for all the take down methods, showing the greatest difference for the *biased* and *prediction* methods.

This difference is related to the nature of the enemies in the scenario: all of them are static enemies, thus the search is able to find consistent hidden spots, since once found them it is unlikely that they will be invalidated later—the only changes in the FoV areas are produced by the take downs, and therefore the only hidden spots that might be invalidated are those found inside the

	Success Rate		Nodes in Solution				Time per Iteration (s)	
	mean ($\overline{x}$)	Std. Deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Preset Hidden Zones								
Random	33.4%	0.0341	4947	361	9972	116	4.29	0.198
Bias	77.39%	0.0319	3789	261	9994	146	2.74	0.127
Prediction	79.5%	0.0316	3759	201	9984	68	2.78	0.137
Correction	57.9%	0.0496	4667	315	9962	52	3.60	0.239
Learned Hidden Zones								
Random	34.6%	0.0392	4726	659	9967	149	3.95	0.237
Bias	93.6%	0.0200	2636	246	9944	32	1.25	0.091
Prediction	93.2%	0.0241	2750	186	9926	31	1.32	0.112
Correction	69.5%	0.0329	3964	388	9977	32	2.57	0.152

Table 5.7 – Results for the body hiding problem in the Aventa Alley scenario.

FoV of an eliminated enemy, which on a following search might be inaccessible. Furthermore, for high **body drop probabilities** $\mathbf{p_d}$ (which not only have shown the better success rate in the parameter tests) the search tends to drop the bodies close to where the take downs are performed; due to the relative proximity of all enemies, this also allows it to try to perform another take down immediately rather than wandering around the scenario before aiming for the next enemy. When we analyze the scenario with *preset* hidden zones (in figure 5.8 on page 94), we observe that although the right portion of the scenario might seem like the best place to hide a body because it is away from all the enemies, it is also the zone that requires the greatest effort to reach and come back. On the other hand, hiding the bodies near to where enemies are ensures that they will not be seen, while at the same time saving on the length of the solution paths.

Albarca Baths

The Albarca Baths scenario is the most complex one we analyze, and having the lowest success rate for the take down problem it also presents the lowest success for the body hiding search. Here we again see a decrease in the success rate when using the *preset* hidden zones due to the added constraints. Table 5.8 shows there is a small improvement in its success rate when using *learned* hidden zones compared with the pure take down results in table 5.5 on page 112. This comes from

	Success Rate		Nodes in Solution				Time per Iteration (s)	
	mean ($\overline{x}$)	Std. Deviation (σ)	$\overline{x}$	σ	max	min	$\overline{x}$	σ
Preset Hidden Zones								
Random	8.29%	0.0253	3605	779	9998	194	6.2	0.231
Bias	10.6%	0.0272	5580	475	9887	404	6.73	0.255
Prediction	12.6%	0.0303	4344	733	9860	328	6.31	0.202
Correction	12.3%	0.0340	4485	622	9973	227	5.73	0.205
Learned Hidden Zones								
Random	11%	0.0204	4026	1123	9836	170	4.40	0.101
Bias	23.6%	0.0402	5038	548	9841	324	4.39	0.173
Prediction	22.1%	0.0439	4887	532	9999	308	4.49	0.158
Correction	13.4%	0.0253	3992	1003	9988	355	4.38	0.053

Table 5.8 – Results for the body hiding problem in the Albarca Bath scenario.

the same characteristics of the *preset* hidden zones as in the Aventa Alley scenario—hiding zones in distant alcoves requires added effort in creating paths that go back and forth to hide the enemy bodies, while the learned hidden spots enable reductions in time between take downs by dropping the bodies more immediately in the free space presented in the scenario. Furthermore, finding hidden zones near to the 3D elements that emulate hanging pipes allows the player to stay hidden and sneak past other enemies after dropping the bodies.

5.3 Overview of Results

The set of experiments performed to analyze the proposed approaches to solve the take down and body hiding behaviour show that while it is possible to use the RRT search to solve these problems, adding a method to bias the search towards a zone where a take down can be performed improves the results of the search. Furthermore, the success rate will be affected by the speed of the enemies in relation to the speed of the player. However, a parameter that impacts more strongly in the success of the search is the distance from an enemy within which the player can perform a take down.

While the addition of a bias to the search greatly improves the success rate of the algorithm

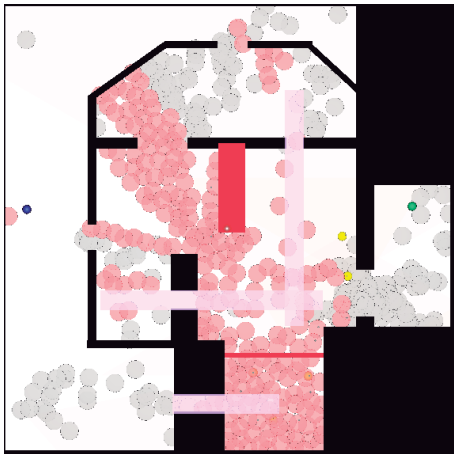


Figure 5.30 – Example of the hidden zones learned after the tests in the Albarca Baths Scenario

for 2D scenarios above 90%, for scenarios with 3D elements or mechanics this is only true when the enemies are mostly static. For scenarios with a high density of enemies, the success of all the approaches drops below 60%. Furthermore, when precise moves are required in sequence such as going through narrow corridors and timing correctly against a number of enemies, in scenarios with 3D elements, the success rate of the search drops even further.

It is interesting to note thatwhile adding a body hiding constraint to solve the level might indicate an increase in the complexity of the problem, for some cases it can actually be used to boost success.This is especially true when valid hiding zones are placed along a less complex solution path, and for scenarios where there is a considerable empty free space that is useful to explore. This could be used for analyzing the design of scenarios and for adding discrete hints about the path a player should take inside a scenario to solve it.

For the body hiding problem, however, it was observed that a bias towards a hidden zone and the probability of dropping a body can increase the success rate of the search only if the values for these parameters are set according to the scenario design. During the tests, it was observed that high body drop probability is helpful for scenarios where there is a considerable free space that is never explored by the enemies, while a moderate probability is preferred when the player is required to drop the bodies in specific zones that should be discovered. The bias towards a hidden zone, on the other hand, was observed to be useful in all cases, showing better results when this bias is not fully strict but allows for the exploration of new hiding zones.

Chapter 6
Conclusion & Future Work

Stealth game scenarios are designed as a combination of different factors that can either reduce or increase the difficulty for the player in achieving his goals. Among these factors, the mechanics granted to both the player and enemies are one of the most popular tools used to create different challenges. However, adding these mechanics increases the space of player actions and therefore, the complexity for analyzing the design and its solutions. The take down mechanic is a frequent addition in stealth games to introduce an alternative for creating scenarios where the solution space increases as the player neutralizes enemies. Furthermore, this mechanic is often accompanied by a body hiding feature to create realistic situations. In this work, we have developed a representation of both of these mechanics, as well as a formal analysis of the elements involved with them. We incorporated this representation into an RRT search to analyze a game scenario and find a solution for a number of game scenarios that can help to describe different aspects of them that can be useful for designers, such as how diverse the solution space is, the effect of the speed of characters and take down distance, the impact of the location of body hiding zones and the inclusion of 3D mechanics. We then introduced a number of improvements that build on the RRT search to increase its success rate. These improvements were demonstrated in a variety of test levels modelled from real stealth game scenarios, starting from 2D representations and building up to scenarios with emulated 3D elements where the difference between the improvements and the basic search was shown. We show how the proposed solutions are able to solve the take down and body hiding scenarios with reasonable success and presented different designs for the body hiding problem that can either improve or hinder the success rate of the search.

6.1 Future Work

Stealth is genre with a range of features yet to be explored. Previous work has focused on generating paths that stay hidden, and a few others have analyzed the inclusion of mechanics such as combat or distractions, leaving a number of other mechanics that can be considered, including concealment, inclusion of allies, luring enemies, etc. Our work presents a possible approach to the take down and body hiding behaviours, but it has not explored all possible variations on them. Variants such as having a dynamic take down radius based on available weapons, the inclusion of dynamic NPCs that occlude the movement and sight of player or enemies, hidden enemies, among others are potentially interesting to explore.

One of the problems of the RRT search lies in the validation to connect two nodes. For this we should be detecting intersection between a potential player path and enemy FoVs. The latter can be complex, however, and while our proposed approach tries to avoid calculating the precise 3D shapes generated by the enemy FoVs, using a simpler calculation based on discretized time, a more precise FoV representation and accompanying intersection test would improve accuracy, and potentially performance as well.Other improvements lie in the sampling process, which could prioritize the zones where enemies are, or the spaces closer to the goal. In the same way, the choice of parent RRT node might also be improved by prioritizing those neighbors with fewer enemies alive.

The take down process can be improved by better heuristics for choosing the blind spots of enemies, since our use of random selection can result cases where the chosen point is in an unreachable location of the scenario. Similarly, for the body hiding problem, the selection of the hidden zone can be improved by using another method than a random selection, which produces cases where the target is unreachable and the chance to bias the search at that point is discarded.

One interesting case would be the modification of the search and the RRT structure to allow the possibility of choosing either a take down or body drop action at each node, rather than having nodes dedicated to these individual actions. This could improve the performance for scenarios with a large number of enemies where the order in which they are taken down is critical. One final approach is the combination of the stealth take down and body hiding models with a distraction and combat model to allow for solutions combining these options to deal with enemies.

Bibliography

[1] J. Tremblay, P. A. Torres, N. Rikovitch, and C. Verbrugge, "An exploration tool for predicting stealthy behaviour", in *The Second Workshop on Artificial Intelligence in the Game Design Process*, 2013, pp. 34–40.

[2] A. Borodovski and C. Verbrugge, "Analyzing stealth games with distractions", in *Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Oct. 2016, pp. 129–135.

[3] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*. Springer International Publishing, 2018, ISBN: 978-3-319-63519-4. DOI: 10.1007/978-3-319-63519-4.

[4] J. Schaeffer and H. van den Herik, "Games, computers, and artificial intelligence", *Artificial Intelligence*, vol. 134, no. 1, pp. 1–7, 2002, ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00165-5. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001655.

[5] G. Tesauro, "Td-gammon: a self-teaching backgammon program", in *Applications of Neural Networks*, A. F. Murray, Ed. Boston, MA: Springer US, 1995, pp. 267–285, ISBN: 978-1-4757-2379-3. DOI: 10.1007/978-1-4757-2379-3_11. [Online]. Available: https://doi.org/10.1007/978-1-4757-2379-3_11.

[6] A. M. Turing, "Chess", in *Computer Chess Compendium*, D. Levy, Ed. New York, NY: Springer New York, 1988, pp. 14–17, ISBN: 978-1-4757-1968-0. DOI: 10.1007/978-1-4757-1968-0_2. [Online]. Available: https://doi.org/10.1007/978-1-4757-1968-0_2.

[7] R. P. Jones and D. J. Thuente, "The role of simulation in developing game playing strategies", in *1990 Eastern Multiconference. Record of Proceedings. The 23rd Annual Simulation Symposium*, Apr. 1990, pp. 89–97. DOI: 10.1109/SIMSYM.1990.717276.

[8] A. L. Samuel, "Some studies in machine learning using the game of checkers", *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, Jul. 1959, ISSN: 0018-8646. DOI: 10.1147/rd.33.0210.

[9] M. Campbell, A. Hoane, and F.-h. Hsu, "Deep blue", *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002, ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370201001291.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, no. 7587, p. 484, Jan. 2016. [Online]. Available: https://doi.org/10.1038/nature16961.

[11] I. Millington and J. Funge, *Artificial intelligence for games*, 2nd. Morgan Kaufmann Publishers, 2009, ISBN: 978-0-12-374731-0.

[12] Z. A. Algfoor, M. S. Sunar, and H. Kolivand, "A comprehensive study on pathfinding techniques for robotics and video games", *Int. J. Comput. Games Technol.*, vol. 2015, 7:7–7:7, Jan. 2015, ISSN: 1687-7047. DOI: 10.1155/2015/736138. [Online]. Available: https://doi.org/10.1155/2015/736138.

[13] J. E. Laird and M. van Lent, "Human-level AI's killer application: interactive computer games", in *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI)*, 2000, pp. 1171–1178.

[14] A. J. Champandard, *Planning in games: an overview and lessons learned*, *AIGameDev.com* article, Mar. 2013. [Online]. Available: http://aigamedev.com/open/review/planning-in-games/.

[15] J. Abercrombie, *Bringing Bioshock Infinite's Elizabeth to life: An AI development postmortem*, Video file, Game Developers Conference, Mar. 2014. [Online]. Available: https://www.gdcvault.com/play/1020831/Bringing-BioShock-Infinite-s-Elizabeth.

[16] M. Dyckhoff, *Ellie: buddy AI in The Last of Us*, Video file, Game Developers Conference, Mar. 2014. [Online]. Available: https://www.gdcvault.com/play/1020364/Ellie-Buddy-AI-in-The.

[17] ——, "Ellie: buddy AI in The Last of Us", in *Game AI Pro 2: Collected Wisdom of Game AI Professionals*, S. Rabin, Ed., Natick, MA, USA: A. K. Peters, Ltd., 2015, ch. 35, pp. 431–442, ISBN: 9781482254792.

[18] C. Alkaisy, *The history and meaning behind the 'Stealth genre'*, *Gamasutra* article, http://www.gamasutra.com/blogs/MuhammadAlkaisy/20110610/7764/The_history_and_meaning_behind_the_Stealth_genre.php, Oct. 2011.

[19] L. Moore, "Improving guard behaviour in a top-down 2D stealth game", *Game Behaviour*, vol. 1, no. 1, 2014. [Online]. Available: https://computing.derby.ac.uk/ojs/index.php/gb/article/view/8.

[20] Y. C. Hui, E. C. Prakash, and N. S. Chaudhari, "Game AI: artificial intelligence for 3D path finding", in *2004 IEEE Region 10 Conference TENCON 2004*, vol. B, Nov. 2004, 306–309 Vol. 2. DOI: 10.1109/TENCON.2004.1414592.

[21] R. Graham, H. McCabe, and S. Sheridan, "Pathfinding in computer games", *The ITB Journal*, vol. 4, no. 2, p. 6, 2003. DOI: 10.21427/D7ZQ9J. [Online]. Available: https://arrow.dit.ie/itbj/vol4/iss2/6.

[22] N. M. Wardhana, H. Johan, and H. S. Seah, "Enhanced waypoint graph for surface and volumetric path planning in virtual worlds", *The Visual Computer*, vol. 29, no. 10, pp. 1051–1062, Oct. 2013, ISSN: 1432-2315. DOI: 10.1007/s00371-013-0837-x. [Online]. Available: https://doi.org/10.1007/s00371-013-0837-x.

[23] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968, ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[24] N. Swift, *Easy A* (star) pathfinding*, Medium Corporation, Feb. 2017. [Online]. Available: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2.

[25] A. Stentz, "Optimal and efficient path planning for partially-known environments", in *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, May 1994, 3310–3317 vol.4. DOI: 10.1109/ROBOT.1994.351061.

[26] A. Nareyek, "AI in computer games", *Queue*, vol. 1, no. 10, pp. 58–65, Feb. 2004, ISSN: 1542-7730. DOI: 10.1145/971564.971593. [Online]. Available: http://doi.acm.org/10.1145/971564.971593.

[27] C. Niederberger, D. Radovic, and M. Gross, "Generic path planning for real-time applications", in *Proceedings Computer Graphics International, 2004.*, Jun. 2004, pp. 299–306. DOI: 10.1109/CGI.2004.1309225.

[28] M. Garzón, E. P. Fotiadis, A. Barrientos, and A. Spalanzani, "RiskRRT-based planning for interception of moving objects in complex environments", in *ROBOT2013: First Iberian Robotics Conference*, M. A. Armada, A. Sanfeliu, and M. Ferre, Eds., Springer International Publishing, 2014, pp. 489–503, ISBN: 978-3-319-03653-3.

[29] B. Tastan, "Learning human motion models", Department of EECS, University of Central Florida, Tech. Rep. WS-12-18, 2012. [Online]. Available: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5479/5777.

[30] M. A. Marzouqi and R. A. Jarvis, "Robotic covert path planning: a survey", in *2011 IEEE 5th International Conference on Robotics, Automation and Mechatronics (RAM)*, Sep. 2011, pp. 77–82. DOI: 10.1109/RAMECH.2011.6070460.

[31] D. L. Page, A. F. Koschan, M. A. Abidi, and J. L. Overholt, "Ridge-valley path planning for 3d terrains", in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, May 2006, pp. 119–124. DOI: 10.1109/ROBOT.2006.1641171.

[32] S. A. Bortoff, "Path planning for UAVs", in *Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334)*, vol. 1, Jun. 2000, 364–368 vol.1. DOI: 10.1109/ACC.2000.878915.

[33] M. S. Marzouqi and R. A. Jarvis, "New visibility-based path-planning approach for covert robotic navigation", *Robotica*, vol. 24, no. 6, pp. 759–773, 2006.

[34] R. Geraerts and E. Schager, "Stealth-based path planning using corridor maps", in *Computer Animation and Social Agents*, 2010.

[35] M. S. Marzouqi and R. A. Jarvis, "Covert path planning for autonomous robot navigation in known environments", in *Australasian Conference on Robotics and Automation (ACRA)*, 2003.

[36] A. Johansson and P. Dell'Acqua, "Knowledge-based probability maps for covert pathfinding", in *Motion in Games*, R. Boulic, Y. Chrysanthou, and T. Komura, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 339–350, ISBN: 978-3-642-16958-8.

[37] J. Park, J. Choi, J. Kim, and B. Lee, "Roadmap-based stealth navigation for intercepting an invader", in *2009 IEEE International Conference on Robotics and Automation*, May 2009, pp. 442–447. DOI: 10.1109/ROBOT.2009.5152560.

[38] M. R. F. Mendonça, H. S. Bernardino, and R. F. Neto, "Stealthy path planning using navigation meshes", in *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, Nov. 2015, pp. 31–36. DOI: 10.1109/BRACIS.2015.49.

[39] S. M. Lavalle, "Rapidly-exploring random trees: a new tool for path planning", Computer Science Dept., Iowa State University, Tech. Rep. TR 98-11, Oct. 1998.

[40] J. J. Kuffner and S. M. LaValle, "Rrt-connect: an efficient approach to single-query path planning", in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, Apr. 2000, 995–1001 vol.2. DOI: 10.1109/ROBOT.2000.844730.

[41] A. Borodovski, "Pathfinding in dynamically changing stealth games with distractions", Master's thesis, McGill University, Montréal, Canada, Aug. 2016.

[42] E. Donsky and H. J. Wolfson, "PepCrawler: a fast RRT-based algorithm for high-resolution refinement and binding affinity estimation of peptide inhibitors", *Bioinformatics*, vol. 27, no. 20, pp. 2836–2842, 2011. DOI: 10.1093/bioinformatics/btr498. eprint: /oup/backfile/content_public/journal/bioinformatics/27/20/10.1093/bioinformatics/btr498/2/btr498.pdf. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr498.

[43] A. Bauer and Z. Popović, "RRT-based game level analysis, visualization, and visual refinement", in *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'12, Stanford, California, USA: AAAI Press, 2012, pp. 8–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014629.3014632.

[44] M. Naveed, D. E. Kitchin, and A. Crampton, "Monte-carlo planning for pathfinding in real-time strategy games", in *Proceedings of PlanSIG 2010. 28th Workshop of the UK Special Interest Group on Planning and Scheduling joint meeting with the 4th Italian Workshop on Planning and Scheduling*, S. Fratini, A. Gerevini, D. Long, and A. Saetti, Eds., Brescia, Italy: PlanSIG, Dec. 2010, pp. 125–132. [Online]. Available: http://eprints.hud.ac.uk/id/eprint/9242/.

[45] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning", in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293, ISBN: 978-3-540-46056-5.

[46] J. Tremblay, A. Borodovski, and C. Verbrugge, "I can jump! exploring search algorithms for simulating platformer players", in *Experimental AI in Games Workshop (EXAG 2014)*, Oct. 2014.

[47] J. Tremblay and C. Verbrugge, "An algorithmic approach to decorative content placement", in *Experimental AI in Games Workshop (EXAG 2015)*, Nov. 2015, pp. 75–81.

[48] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007. [Online]. Available: http://doi.acm.org/10.1145/361002.361007.

[49] S. D. Levy, B. Heckel, and M. A. Alvarez, *KD-tree implementation in Java and C#*, https://home.wlu.edu/~levys/software/kd/, 2010.

[50] B. Delaunay *et al.*, "Sur la sphere vide", *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, no. 793-800, pp. 1–2, 1934.

[51] T. C. Hales, "The Jordan curve theorem, formally and informally", *The American Mathematical Monthly*, vol. 114, no. 10, pp. 882–894, 2007. DOI: 10.1080/00029890.2007.11920481. [Online]. Available: https://doi.org/10.1080/00029890.2007.11920481.

[52] J. Tremblay, P. A. Torres, and C. Verbrugge, "Measuring risk in stealth games", in *Proceedings of the 9th International Conference on Foundations of Digital Games*, Apr. 2014.

[53] C.-W. Huang and T.-Y. Shih, "On the complexity of point-in-polygon algorithms", *Computers & Geosciences*, vol. 23, no. 1, pp. 109–118, 1997, ISSN: 0098-3004. DOI: https://doi.org/10.1016/S0098-3004(96)00071-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098300496000714.

[54] Centerstrain01, *Thief: Stealth Walkthrough - Master - Ghost - Part 26 - Chapter 7 - The Hidden City 1/2*, Video file, Mar. 2014. [Online]. Available: https://www.youtube.com/watch?v=icyKiYFWbBE (visited on 02/22/2019).

[55] G. Gameplay, *Thief - The Hidden City - Chapter 7 Part 1 - Walkthrough*, Video file, Mar. 2014. [Online]. Available: https://www.youtube.com/watch?v=_hBIZZvX1iY (visited on 03/17/2019).

[56] Dishonored wiki contributors, *Aventa district – Dishonored Wiki*, 2016. [Online]. Available: https://dishonored.gamepedia.com/Aventa_District (visited on 03/17/2019).

[57] Zevic, *''Dishonored 2" Walkthrough (Very Hard + All Collectibles) Mission 4: The Clockwork Mansion*, Video file, Apr. 2017. [Online]. Available: https://www.youtube.com/watch?v=hmWOYwj-vM4 (visited on 03/17/2019).

[58] SporksAreGoodForYou, *Dishonored: Death of the Outsider |M1: One Last Fight |All Collectibles |StealthMercy*, Video file, Sep. 2017. [Online]. Available: https://www.youtube.com/watch?v=2pLqPJmadRE (visited on 03/17/2019).

[59] V. G. Source, *Dishonored: Death of The Outsider - One Last Fight: Jeanette Lee Assassinated, Wolfhound Hook Mine*, Video file, Sep. 2017. [Online]. Available: https://www.youtube.com/watch?v=zpKFDmvLzwk (visited on 03/17/2019).