THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Game AI: Reasoning About Space

Daniel Alexander Jeffery (3220069)

Bachelor of Computer Science (Honours)

Supervisor: Dr Malcolm Ryan

Assessor: Dr Maurice Pagnucco

October 19, 2010

Contents

Abstract

Game Artificial Intelligence, for a long time now, has been implemented using a small subset of the approaches found in the traditional/modern artificial intelligence field. Now, as computer game technology has advanced, players are more savvy and 'realism' is now expected from a game. The NPCs (non-playing characters) should also be realistic; they should be able act intelligently, be able to reason and make informed decisions. Based on the want for more realistic game artificial intelligence, I executed an experiment and evaluation of an implementation of realistic game artificial intelligence for a pursuit-evasion game. I extensively modified a game engine designed by Jimmy Kuriwan, "Art Of Stealth", to implement a working particle filter and achieved some interesting behaviours supplemented by subtle scripting to create a workable and interesting set of artificial intelligent agents.

Acknowledgements

I would like to thank Dr Malcolm Ryan, my supervisor, for all his help and encouragement throughout the research and development of my thesis. Without his assistance I surely would not have been able to have achieved as much as I did or have learnt so much in the process.

I would also like to thank Dr William Uther for sitting down with me early in my studies and providing me with the information and explanations I required in order to tackle this topic.

Thank you to all the games-labbers and playtesters for their support, time, energy, great ideas and inspiration.

Finally a big thank you to Jimmy Kurniawan for allowing me to work with his original game engine "Art Of Stealth".

Chapter 1

Introduction

The artificial intelligence used in computer games is inferior and far less powerful than modern, established techniques. The techniques used to implement game AI have not evolved very much since their beginning. The technology has improved but the same scripted, finite state machine based approach is prevalent throughout most modern titles. This is because the computation and resources involved in implementing and running traditional/modern artificial intelligence techniques are incredibly taxing on real-time systems, especially when the system is not only trying to simulate a reasoning process but also process input from a user, display graphical output and manage the inner state of the game. Consequently, many game AI developers generally use less expensive techniques to simulate or produce the illusion of logic, reasoning and planning. Since the goal of computer games is to provide entertainment it is considered overkill to implement higher artificial intelligence techniques. This is partly because the majority of the processing time is already devoted to running the game engine. Additionally implementing higher artificial intelligence is seen to be more complicated than using 'tricks', scripted actions and finite state machines. The question is then asked, which is simpler "having to create the brain of an actor playing the role of a character instead of the brain of the character in the first place?"[3].

For computer games there is a requirement that the NPCs make interesting opponents. There is no enjoyment gained by beating an opponent that the player does not view as a challenge. On the other hand, the opponents must be challenging but beatable. The rule of thumb for game AI design is "as long as the player has the illusion that a computer-

controlled character is doing something intelligent, it doesn't matter what AI (if any) was actually implemented to achieve that illusion"[6]. Generally game AI is implemented as simple rule-based systems using scripted behaviours without any sophisticated artificial intelligence. Much of the time these implementations are quite stupid and designers make up for these faults by allowing the NPCs to cheat and provide them with information about the game state that should not be accessible in terms of a realistic depiction of the scenario. "Cheating in this manner is common and helps give the computer an edge against intelligent human players. If it is obvious to the player that the computer is cheating, the player likely will assume his efforts are futile and lose interest in the game. Also, unbalanced cheating can give computer opponents too much power, making it impossible for the player to beat the computer"[2]. I find this is a major problem with current game artificial intelligence and I believe that instead of giving the computer opponents an edge, we should instead strive to design agents that are able to remain competitive and realistic through means free of deception.

The process of making the game state fully observable to the NPC(s) is fine for the majority of the time, but it is not always effective or believable, especially in pursuit-evasion games. The pursuit-evasion scenario is where one or more pursuers (the NPCs) chase the player while the player attempts to escape or hide. Due to this trade-off between knowledge and intelligence the NPCs tend to lack the ability and the ingenuity to be able to act convincingly. They exhibit behaviours such as following the player indefinitely, finding the player wherever the player hides and acting with overly predictable behaviour. This mixture of stupid and seemingly brilliant behaviours creates an unrealistic opponent. To overcome this lack of consistency in behaviours I plan to use more advanced techniques taken from other fields of artificial intelligence.

Solutions to these types of issues may be solved by using techniques implemented in the field of robotics. Many computer games have features and issues similar to that of the robotic environment, i.e. a continuous, partially observable, non-deterministic environment with agents having to plan and make decisions in real-time. My project involved the design and implementation of realistic, challenging and enjoyable artificial intelligent agent(s) using some techniques adapted from the field of robotics. I looked at a number of techniques,

but in particular I experimented with particle filtering as a method of tracking, explored a number methods for decision making and motion planning.

During my experiments I intended to demonstrate how a modifiable and tunable particle filter implementation can be used to achieve behaviours with varying levels of difficulty. I also intended to show how this can be used to develop intelligent NPCs which are not constrained by the limitations of rule-based systems and cheating. Using a combination of particle filtering and efficient and fast decision methods can provide a realistic, adjustable and intelligent game AI supplemented with a smaller level of underlying scripting that could replace a completely rule-based, scripted systems. By the end of this project I also had hoped to have developed a parameterised, modifiable and adjustable framework for the control of an artificially intelligent pursuer. I believe I have made the first steps towards this goal.

At the completion of my thesis I have created an competently performing artificially intelligent set of agents which are competitive and relatively realistic.

The following is a outline of what each section of my thesis will entail:

- **Background**
  - **Theory**: Definitions and ideas that need to be explained to understand the context of the following work
  - **Literature Review**: A summary of some previous work done in the area of pursuit-evasion game study
- **Design**: Documentation on the work undertaken and techniques that could be used to implement the three major features of my pursuit agents: motion planning, tracking and decision making
- **Evaluation**: An summary of the work undertaken and an assessment of its success.
- **Conclusion**: A summary of achievements and failures

Chapter 2

Background

The first part of this section will explain the basic concepts and theories related to the literature review and design section, sections 2.2 and 3 respectively. The second part of this section will describe some of the work previously done on implementing artificial intelligent agents in pursuit-evasion scenarios. Some of the work has been explored in robotics and also in simulated environments (including games).

2.1 Theory

This section defines and describes theory and related techniques referenced in literature review and the design, sections 2.2 and 3 respectively. The techniques and ideas covered set the context of the following research about the design of artificially intelligent agents. Three main aspects of designing an agent for a pursuit-evasion senario are discussed here. They are motion planning, tracking and decision making.

The following definitions are obtained mainly from *Probabilistic Robotics*[9] and *Artificial Intelligence: A Modern Approach*[8].

2.1.1 Motion Planning

Motion planning could generally be described as detailing a path into a discrete set of motions. Planning in a continuous space can be an incredibly difficult task. To minimize the errors and issues entailed in pathing in a continuous space we can map the environment

into a discrete set of nodes and edges; connecting them in order to provide 'safe' navigation.

Configuration Space

The configuration space is the space of a robot's states defined by location, orientation and joint angles. Path planning involves finding a path from one configuration to another in the configuration space. There are a number of techniques to do this but I will cover the two major families - 'cell decomposition' and 'skeletonization (Roadmap methods)'. Definitions are obtained from *Artificial Intelligence: A Modern Approach*[8].

Cell Decomposition

Cell decomposition is the process of converting continuous, free space into a finite set of continuous regions, known as 'cells'. Common cells shapes are square and triangle polygons. The discretization of space in this way transforms the problem of pathfinding in a near infinite space into a simple graph search problem in which movement involves simply moving to a valid adjacent cell. This is a simplistic and fast method of decomposing the configuration space into something we can work in acceptable time.

This approach suffers in high-dimensional environments as the number of cells increase exponentially with the number of dimensions. Another issue with cell decomposition is that in most real world environments we cannot adequately cover the entire space. In any decomposition there is a good chance that there will be 'left over' space not encompassed by the distribution. Thus, cell decomposition does not provide a complete coverage of the configuration space, while it may be simple and efficient, there is the possibility if it is not performed accurately the path planner may be unsound.

Cell Decomposition is a simple and efficient way of discretizing an environment, but it leaves the mapped space incomplete. Having a incomplete search space would allow for unrealistic exploits to be made available to the player and also consequential "stupidity" on behalf of the agent. The agent would not be 'allowed' to search the entire space for the evader simply because of the restrictions placed on them by a grid-based decomposition. Another impact of using cellular decomposition is that it has an unavoidable tendency to create inefficient and unrealistic movement through the mapped space. For example, if

we have a grid based world, the only movement available is north, south, east or west. Having only these four options will create staircase patterns when the agent needs to move diagonally.

Skeletonization (Roadmap) methods

Skeletonization is the process of mapping the configuration space into a one-dimensional space consisting of a set of vertexes and edges, i.e. the skeleton. In doing this the problem of planning in a continuous environment is now a simple graph search. Generally there are two main ways of generating this skeleton: using a Voronoi graph or generating a probabilistic roadmap.

A Voronoi graph is obtained by taking the set of all points equidistant between two or more obstacles and joining them into a path. Doing so creates a clean path around all obstacles assuming that the distance between every obstacle is large enough for the agent to move between (if not some routes may need to be discarded). The issue with this approach is that the number of paths generated is quite small, and the agents' available bounds of motion are heavily constricted.

A probabilistic road map is generated by taking a large number of random points across a map, discarding points that lie on inaccessible places and joining the rest with their nearest neighbours. This defeats the problem associated with Voronoi graphs but presents new problems. Since it is a random distribution across the map, the distribution of points has to be large enough to provide a complete mapping of the entire space.

Skeletonization maps the space in a more realistic fashion than cell decomposition. Even though skeletonization methods don't necessarily map the space into a series of nodes and edges which are entirely explorable, it provides a graph in which the space can be explored without relying too heavily on collision detection. The graph generated by skeltonization techniques also do not incur unrealistic movement patterns.

2.1.2 Tracking

There are a large number of different methods for tracking in a continuous, partially observable environment. A lot of tracking used in robotics is specializations of Bayes filtering

such as Kalman filters and particle filters.

Belief State

The state of some system is a description about it in a vector of variables. The state, if described correctly, should contain all relevant information required to represent the system at any given point in time, under the constraints of the environment. A vector x, can be used to describe the system at some time t. For example the state of an agent tracking its position in a two dimensional space may be represented as $< X, Y, \theta >$ with X and Y being coordinates and θ being the agent's orientation. A possible state could be $x_t =< 25, 55, 43^\circ >$ at some time t. In terms of robotics, we have a continuous environment both spatially and in time, but we can discretize both of these. For time (t), we represent it discretely making each step in time as a uniform interval.

When the environment is fully observable the entire state vector, x, is known. In robotics we never have a fully observable environment, as such, we keep what is known as a 'belief state'. A belief state reflects the robot's internal knowledge about the system state. Using the above example, if a robot is tracking its position we may infer $x_t =< 25, 55, 43^\circ >$ at time t from the information obtained but this may not be completely indicative of its true position. A belief state represents the unknown state as a probability distribution over possible states.

Markov Assumption

Let X be a set of states and let x_t be a state variable at time t such that;

$$P(X = x_t) = p(x_t)$$

A stochastic process is Markov if the conditional probability distribution of all future states of the process depend only upon the present state and observation. This means that in order to predict the next system state we do not need to use the complete history of all actions and observations. So, if a process is Markovian then we have;

$$p(x_{t+1}|x_t, x_{t-1}, .., x_0) = p(x_t|x_{t-1}) \text{ when } t = 1..n \text{ and } p(x_0) \text{ when } t = 0$$

Thus, if we assume Markov, we assume that conditional probability of a new state can be found by only having to look at the current state.

Bayes Filter

Bayes filtering (also known as recursive Bayesian estimation) is a relatively simple probabilistic approach to estimating an unknown probability density function. When we are looking at a robotic system we take some input from the world through a set of sensors. This observation we define as o_t at time t. o_t contains all relevant information obtained from the sensors. The Bayes Filter takes advantage of the Markov property in order to predict future states.

The general equation for a Bayes Filter is;

$$p(x_t|o_t) = \eta \cdot p(o_t|x_t) \int p(x_t|x_{t-1}) p(x_t|o_{t-1})\, dx_{t-1}$$

The result values obtained for the previous equation are generally not probabilities since they may not integrate to 1, so we use a normalising constant η. The general Bayes filter algorithm, algorithm 1,is obtained from *Probabilistic Robotics*[9].

Algorithm 1 Algorithm Bayes_Filter(x_{t-1}, o_t)

for all x_t **do**
  $\bar{x}_t = \int p(x_t|x_{t-1}) p(x_t|o_{t-1})\, dx_{t-1}$
  $x_t = \eta \cdot p(o_t|x_t) \bar{x}_t$
end for
return x_t

Particle Filtering

Particle filters are a Monte Carlo (repeated random sampling) based implementation of Bayesian filtering. Particle filters provide an approximation of multimodal, non-linear probability density function. This is done by representing the belief state x_t as a set of random samples drawn from this belief state. This set of N samples are known as particles. Each particle with a weight w_i for particle i, indicates the likelihood/probability of the thing being tracked existing at the location of the particle. Like a Bayes filter it relies on the Markov property to propagate forward new belief states. In the case of pursuit-evasion, the pursuer will use a particle filter with each particle representing a possible position for the evader. Particle filtering is used in robotics because it is a fast, efficient and simple

method of approximating a probability density function. The computational complexity is near linear so it is perfect for real-time processing.

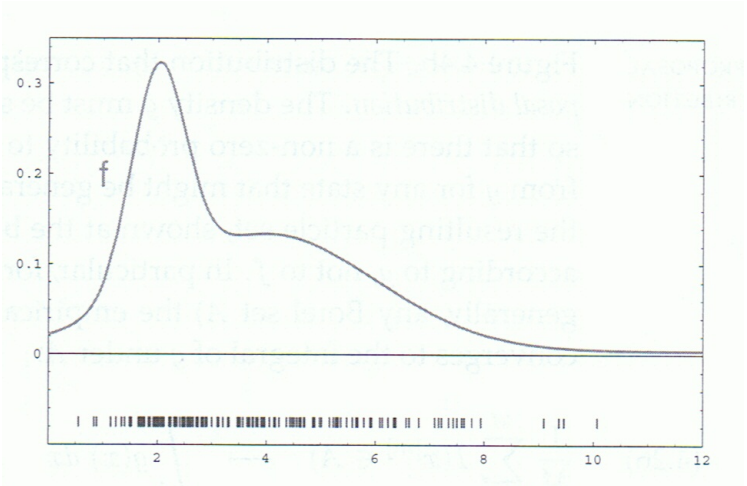

Figure 2.1: A sample distribution of particles approximating the target density f[8]

For the particle filter we have a motion model, $p(x|x_{t-1})$ and an observation model $p(o_t|x_t)$ for state at time t. The motion model essentially predicts the movement of a particle, and in turn predicts the way the evader is moving. In terms of our pursuit-evasion situation it can be expressed as: "Based on where we thought the evader was before, where will they now be at the next timestep?". The motion model would generally predict by applying some Gaussian noise to the current input and applying any assumptions we have about how the target will move. The observation model says; "Based on what I can see, update what I believe about the state". This can be implemented in a number of different ways - specifically it should assign weights to each particle based on how confident we are that the particle accurately represents where our target is located. To maintain this distribution, the update cycle consists of three major steps. This update cycle is repeated for each time step. An explanation of each step is taken from *AI: A Modern Approach*[8];

1. Each sample is propagated forward by sampling the new state value, for new time step t, x_t given the current value x_{t-1} for the sample, and using the motion model $p(x_t|x_{t-1})$.
2. Each sample is weighted by the likelihood it assigns to the new evidence, based on the observation model $p(o_t|x_t)$

3. The population is resampled to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

Algorithm 2 Weighted sample with replacement algorithm

repeat
  For N particles in P, generate a random number R such that $0 \leq R \leq \sum_{i=1}^{N} w_i$
  while $R > 0$ **do**
    $R = R - w_i x_i$
  end while
  Add the particle last deducted to $\bar{P}$
until $|\bar{P}| = N$
$P = \bar{P}$
$sum = \sum_{i=0}^{N} w_i$
for all x_t in P **do**
  $w_i = \frac{w_i}{sum}$
end for
return P

Particle filtering worked very well in Curt Bererton's experiments[1] for tracking a single evader in a two-dimensional space. The distribution is easily maintained, predicted and updated. It is also a simple, efficient and fast algorithm which can run in real-time without requiring too much computational power. It appears that the biggest issue with particle filters is they are quite difficult to share. This issue of sharing raises a few questions: how many particle filters do we need (one for each agent, or a single shared one), how do we merge the gathered information, and how do we get the agents to work together using the knowledge they obtain have obtained?

Not only is particle filtering fast, simple and efficient, I hypothesized that differing behaviours can be achieved by implementing a parametric particle filter featuring the modification of three main features:

- Number of Particles
- Motion Model
- Observation Model.

The downfall of this method was that a maintained distribution is not as easily shared as its sister algorithms such as the Kalman filter, which just requires the multiplication of two Gaussian distributions in order to merge the belief states. As part of my research I hoped to experiment with different methods of sharing and also different methods of collaboration using the information available, but this did not come to fruition due to time constraints.

2.1.3 Decision Making and Planning

Decision making and planning is where the current state of the game is assessed and the agent plans a strategy, or at least its next move. Generally in game AI these are known as behaviours and are either scripted or implemented using simplistic methods like low-depth game trees or finite state machines. These simplistic methods are preferred by game developers because they provide predictable, efficient game play. If a behaviour is too predictable the game becomes too easy and uninteresting. In this instance a game developer may choose to add more opponents, increase the strength an enemy, their health or even the amount of damage they cause. Instead of changing the game play, it would make more sense to imbue them with better reasoning skills.

Greedy Methods

Greedy methods are a reasonably simple approach to decision making. Generally greedy methods don't provide an optimal solution, but a good one for the present situation. This is because greedy methods suffer from local-minimum/maximum. Greedy methods appear to be a great, simple solution and efficient solution to decision making in game AI. Game AI doesn't need to perform optimally, it just needs to perform well enough to remain competitive.

It appears that from both Curt Bererton's work[1] and also the results from the BEAR project[10] that greedy methods, while not optimal in many cases, provide quite good results in pursuit-evasion games. They are fast, easy to implement and relatively efficient (especially if good heuristics are developed) and pose a suitable solution to simple decision making in a complex, continuous and stochastic environment.

Game Trees

Game trees are a method for adversarial search in either cooperative or competitive turn based games. They are used to find the best set of moves (or a plan) based on a current game state and subsequent, plausible states. They are used often in artificial intelligence to find and choose an optimal move. The most common form of game tree search is a 'minimax' algorithm in which we assume the opposition plays optimally for each possible turn, for a finite period of steps. "Given a game tree, the optimal strategy can be determined by examining the minimax value of each node"[8]. The game tree is set out as such; when it is your turn to move, you take the maximum expected value (the utility of a node), when it is the opponents turn to move, you take the minimal expected value (assuming the opponent wishes to choose a move which reduces your expected reward for the game). The tree is explored and computed by recursive depth first search starting from the current game state (the root of the tree) branching out by the resulting game states of every possible move made by you, then from those states, every possible move made by your opponent and so forth. When a final state is reached, the results are propagated back up through the tree.

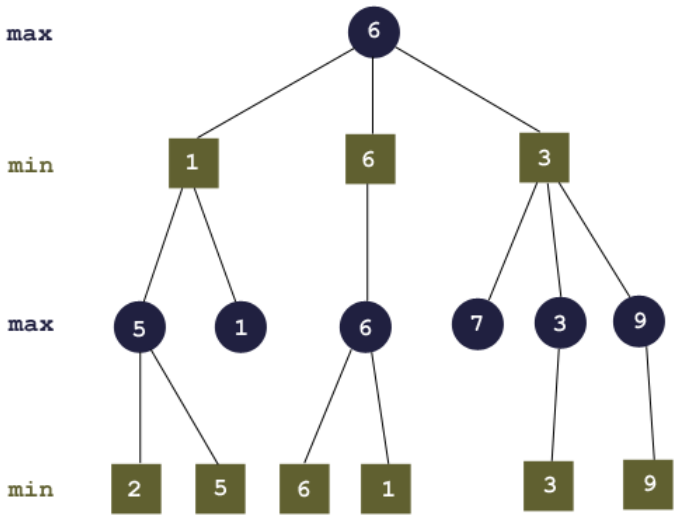

Figure 2.2: An example of a possible, computed minimax game tree of two turns

2.2 Previous Work

2.2.1 State Estimation for Game AI Using Particle Filters

Curt Bererton[1] explored tracking in a pursuit-evasion game using techniques from robotics research, namely particle filters. Bererton suggested that instead of using omniscient information in order to provide challenging game play, we should look at ways of making the agent both appear and actually be intelligent. He suggested that to make the NPCs be smart, they must "make observations about the game world with senses that a player feels are fair" and "take actions based only on the information that they have received".

Bererton proposed that techniques used in robotics would also work well in computer games as much of the same constraints apply. He also suggested that these techniques were somewhat 'tunable' - given more time to run and/or more information they will be 'smarter' or given less time they will be 'dumber'. The same approach can also be used with particle filters by increasing or decreasing the number of particles used to represent the probability density function. The more particles used to represent the distribution, the better the approximation will be - which leads to more accurate tracking. As such, Bererton proposes that these techniques could be used in games to adjust the intelligence of NPCs without resorting to the usual tricks.

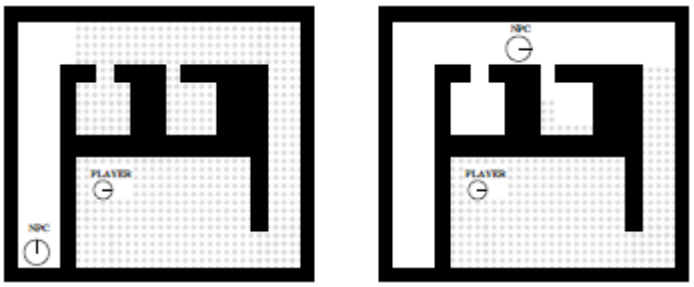

Figure 2.3: The image depicts a search guided by the use of a particle filter. Left: Particles (the small shaded circles) represent possible locations of the player. Right: As the NPC searches, particles in the NPC's line of sight when the player is not present are removed[1]

Bererton's experiment used two teams of agents on a two dimensional map. One team was controlled by NPCs or a human player, while the other was always controlled by the NPCs. The final experiment used 31 particle filters, being maintained simultaneously, tracking 31 agents. 10 laser range finders are were used per NPC for observations. Bererton

found that depending on the map, representing the distribution using between 200 and 500 particles was sufficient. The experiment used a very simple control system, the agents simply navigated to the mean of the posterior distribution (the middle of the cloud of particles). In further experiments the number of particles was reduced, and in doing so, the behaviour of the agents became more random.

This experiment was a great exploration of the possible uses of robotic techniques in a computer games setting. The tuning of the particle filter could have been extended past just simple modification of the number of particles used to represent the probability density function. The transition model used appeared to be static, i.e. no prediction was made about the movement of the target being tracked and the observation model was optimal, i.e. no stochasticity - either the target is seen or it is not seen.

2.2.2 BErkeley AeRobot (BEAR) project

The BEAR project[10] is a research effort at UC Berkeley. It makes use of Unmanned Aerial Vehicles (UAVs) and Unmanned Ground Vehicles (UGVs) to create a test bed for a number of different applications. The aim of BEAR is to "to integrate multiple autonomous agents with heterogeneous capabilities into a coordinated and intelligent system that is modular, scalable, fault-tolerant, adaptive to changes in task and environment, and able to efficiently perform complex missions". In several of these scenarios the test bed was used to enact a pursuit-evasion situation. A team consisted of one UAV and two UGVs as pursuers and a single UGV evader.

For the purpose of the experiments, the problem of both map building and pursuit-evasion were treated as a single problem. The framework consisted of representing the environment as a finite, two-dimensional set of cells with an unknown number of fixed obstacles. To move between the cells, the cell must be empty of obstacles and other agents. As the area is explored, the agents build up a map. Each cell stores the probabilistic likelihood of the evader being in that cell at time t. The BEAR project looks at building on research [5] done by some members of the BEAR group, namely Sastry and Kim, which proposed a "greedy policy to control a swarm of autonomous agents in the pursuit of one or several evaders". Two greedy pursuit polices were explored in the BEAR testbed as

local-max and *global-max*. Both policies plan based on the assumption that the evader is moving randomly.

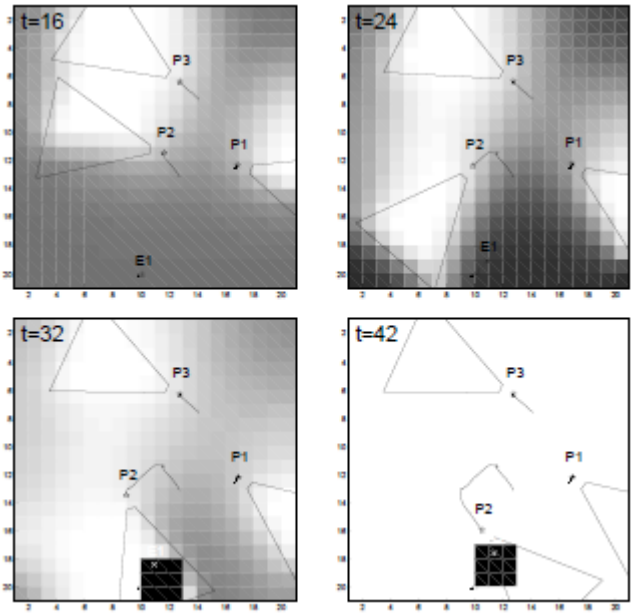

Figure 2.4: One of the BEAR experiments depicting the use of a probabilistic grid space. P1, P2 and P3 are pursuers and E1 is the evader.

The *local-max* policy for each pursuer involves a greedy decision to move to the cell which is adjacent and reachable with the highest probability of the evader being present over all evader maps. This approach is scalable and computationally efficient independent of the size of the maps. The problem with this approach is it's not persistent on average (the probability of the pursuers finding the evaders in a finite period of time is not equal to 1).

The *global-max* policy searches over the entire map to find the action that maximizes the probability of finding the evader. While this approach is persistent on average, it is computationally dependant on the size of the map and hence is quite intensive and inefficient and does not scale.

The evader was tested moving both randomly and intelligently using a *local-min* and/or *global-min* policies. They computed much like local-max and global-max but with the intent on minimizing the probability of being captured at a time step t.

During the experiments they tried a number of different approaches looking at the effectiveness of the pursuit policies when modifying different elements of the scenario. They

experimented with differing visibility regions, speeds for both pursuers and evaders, and also the evasion methods. The conclusions drawn suggest that the best performing pursuit policy was the greedy global-max. They also proved that for this policy "there exists an upper bound on the expected capture time which depends on the size of the area, and the speed and sensing capabilities of the pursuers"[10].

2.2.3 Strategy Generation in Multi-Agent Imperfect-Information Pursuit Games

This paper was written by a team of computer scientists and mechanical engineers at the University of Maryland[7]. It describes a formalism and algorithms for game tree search of a partially observable, continuous, obstacle filled space pursuit-evasion senario. A Multi-agent pursuit scenario is considered with a team of pursuers and a single, moving evader. It is given that each pursuer will be sharing all information they have with all other pursuers and that the map is known to both the pursuer and the evader. The authors propose using a game-tree search with a limited-depth lookahead within a decomposed configuration space to compute the best possible moves to maximise the chances of seeing/finding the evader in subsequent time steps (assuming the amount of possible moves by the evader in a continuous space is infinite, they needed to reduce the state space). To make this algorithm valid it is also assumed that for each move the evader wishes to maximise the 'volume' of its possible locations while the pursuers wish to minimise this 'volume'.

To reduce the amount of possible states the algorithm uses a 'paranoid' model of the evader's behaviour. This means that each purser assumes that the evader knows for each time step the strategy and location of the purser, and as such will make moves that are worse for the pursuer, i.e. choose a move which maximizes the 'volume' of its possible locations. A number of heuristic functions are used depending on the state of a game in order to approximate the minimax value. These heuristics are used to evaluate a game state at any time interval, depending on what point of play the game is at. Using a set of heuristics to approximate the minimax value at a node in the game-tree dramatically speeds up the computation required to make decisions, and in doing so becomes a more feasible solution.

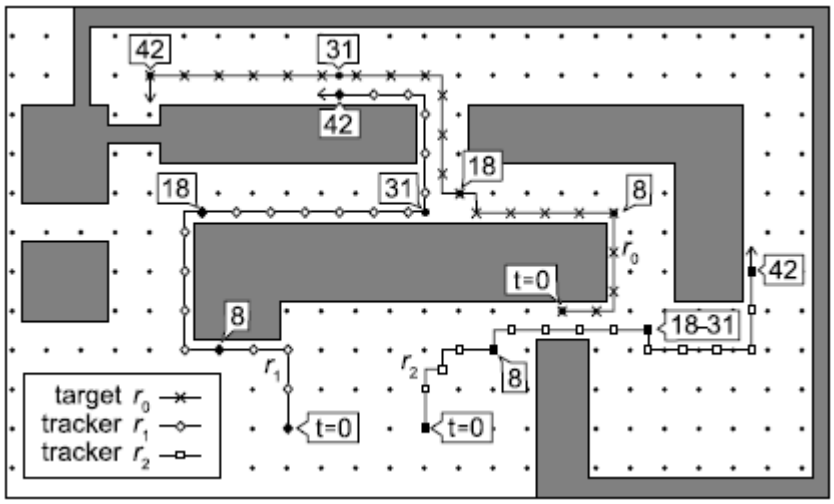

Figure 2.5: Example of a strategy generated by the team's algorithm on a simple tracking problem. Note how trackers r_1 and r_2 move to block two[7]

This paper is very interesting as it puts forward a possible solution to generating more complex strategies for pursuit in a continuous world. The game tree is made searchable by making heavy assumptions about the game state at each time step, using limited depth exploration of the game tree and also using sensible heuristic functions to estimate the game state at each time step. It focuses mainly on the issue of keeping the evader in sight once it has been seen and reasoning about the landscape, rather than methods of finding the target as quickly as possible through efficient exploration and tracking. This paper is markedly different from the majority of work I have examined. The conclusions and results suggest that a game tree search can produce reasonable results if enough assumptions and decompositions are made in order to reduce the total searchable state space.

Chapter 3

Design

The following is a discussion of the main techniques I used for the design and implementation of the pursuing agent(s) in "Art Of Stealth". This chapter describes the design and work involved in the establishment of:

- the platform
- the motion planning
- the tracking method - the particle filter, and
- the decision making and planning.

3.1 Platform

To begin my work I needed a platform upon which to implement the artificial intelligent agents. Since it was my first game project I decided that it would be best to work with an existing project rather than creating my own. The platform I chose to work with is Jimmy Kurniawan's game "Art Of Stealth". It is a 2-dimensional stealth game in a simple polygon world. The player takes the role of a thief evading a set of guards and attempting to steal a jewel. Once the jewel is stolen, the thief must continue to evade the guards until the timer runs out and a goal appears. Once the goal appears the player must navigate to it in order to win the game.

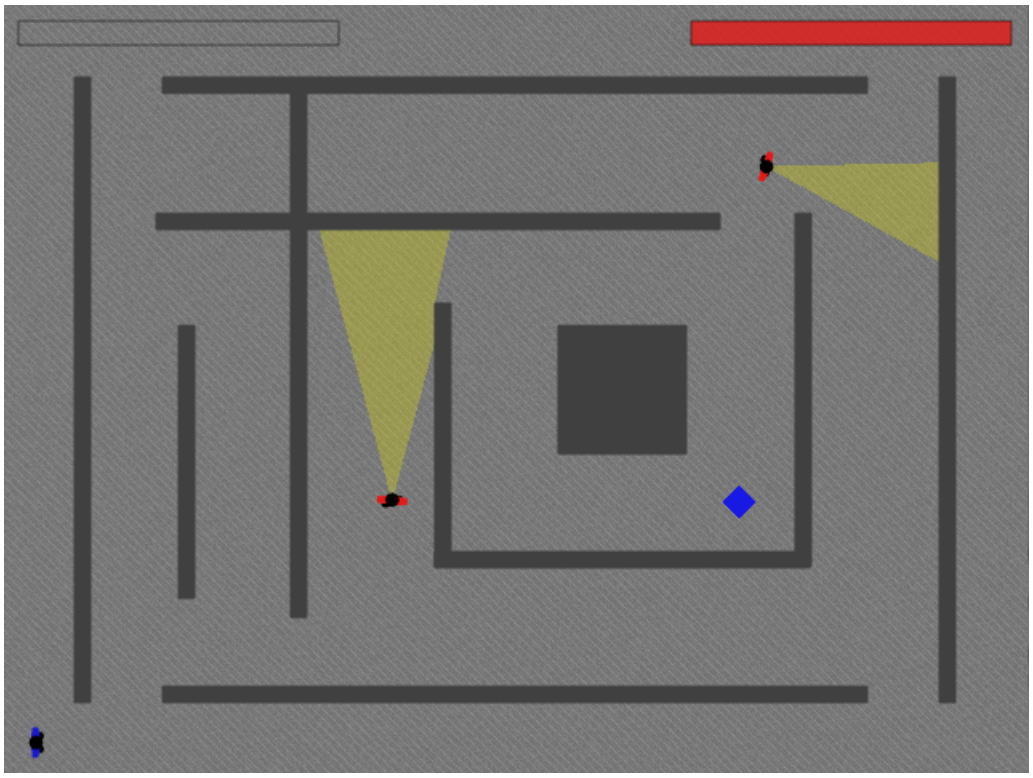

Figure 3.1: Scene from Art Of Stealth. Player (evader) is in blue, NPCs (pursuers) in red.

The game engine uses the C++ library CGAL to do the majority of the geometric calculations, create the game objects and define the pathing information. It is also used to do the collision detection in the game. The graphics are all rendered using openGL from the SDL C++ libraries. Every object in the game is a simple, vector polygon. This simplicity is preferred because I have very little artistic talent and am not familiar with openGL.

Much of the first few months were spent modifying the original game engine to establish a more suitable platform with which to work. Much of the work on the game engine involved minor scripting, correcting logic errors and attempting to increase efficiency in some of the computationally taxing calls to the geometry library.

In my initial design I wanted threaded AI processes. I was not able to do this because the project was already too established and interlinked to easily perform the necessary separations of logic. Traditionally, the computation for each agent's decision processes and updates would be maintained in different threads. This allows the decision process to be controlled somewhat outside of the main line. Results are returned from the thread when

ready and the game itself does not have to slow down to compensate for the background logic to complete. In a threaded environment we can separate out the logic for each individual intelligent agent and allow each thread to be executed at the discretion of CPU. Possible benefits were discussed by Champarndard[4]:

- Reduce the time taken to figure out which new behaviour should be executed in the current context, if there is currently no behaviour running
- Decrease the cost of checking alternative courses of actions to see if they are of higher priority, or guaranteeing that the current behaviour is up-to-date, and
- Speed up the actual execution of behaviours too, as long as they offer a certain level of parallelism (e.g. procedurally animating different body parts).

While I was unable to separate these processes into threads when I was so far into the project, I attempted to simulate this behaviour by interleaving the particle filter updates and the decision making processes. This was done by keeping a timer for each agent, for both the particle filter update and the decision process update. For each guard I calculated an incremental difference between them such that when I had two guards operating they would never update their filters or their decision process in the same timestep. By doing this I achieved a large increase in speed, reduced lag and gained faster responsiveness. This process, while more efficient, was still not an optimal solution and the speed required had to be balanced with the cost of execution. This balance changes from system to system. Finding this equilibrium for my own system was difficult and finding it for every system would be a huge if not impossible task.

Allowing players to play "Art Of Stealth" throughout the development process enabled me to actively improve and modify my game during the development cycle. In terms of contribution to the platform and general gameplay this was quite influential on how the final version was designed. From my original game play there were a few major changes:

- **The score counter became a progress bar**

  Players assumed that the counter I provided them with was indicative of their score on a level. As such they tended to stay in the map for lengthy periods of time in order to gain as many points as possible. While this encouraged competition between the

players, it was not the direction I wanted the game to go. I modified the counter into a progress bar that would fill up as the player evaded the guards and on completion would open an exit door. Without the counter being present the players were less inclined to hang around pointlessly in the maze.

- **Players given both a run and a walk speed**

  Watching my playtesters try the first version of the game was interesting, they all enjoyed messing with the guards and proving they how unintelligent they were. They would run behind the guards and follow them, run away and repeat. I felt that this behaviour was not something I wanted; the guards were supposed to be the enemy, not toys. Consequently, I separated the movement into "a run and a walk" and attached a penalty to running; the guards would be alerted to the "sound" of the footsteps. This implementation will be discussed later in section 3.3.2. This reduced the tendency of the players to mess around with the guards and instead encouraged them to proactively hide from them.

- **Players given health regeneration**

  Players had a tendency to give up if they were injured past a certain point. Once their life dropped below 20% they were inclined to give up and start over. This was again not behaviour I wanted to encourage; it resulted only in frustration and anger in the players and the thrill of evading capture was lost. I subsequently added a regeneration to the players health if they weren't seen for some preset period of time. By doing this when the players were injured, they were more likely to attempt to hide and recover.

3.2 Motion Planning

To create pathing for my agents I required some method to decompose the configuration space. To plan the individual motion of an agent can be a difficult task, it is far easier to restrict their movement and reduce the planning complexity. To do this, I had three methods from the skeletonization family in mind. They were as follows: Delaunay triangulation, random roadmap methods and hand drawn triangles (polygon mesh). I finished the

implementation using Delaunay triangulation after finding it the most suitable technique for a constantly changing environment. This made it suitable for a project in development which was frequently undergoing significant modifications.

During my own personal experimentation if found that the random roadmap method was too unwieldy and unpredictable to be used on the fly. The paths it generates tend to be unsuitable to be actively used. Some points are too concentrated in some areas and far too sparse in others, leading to incomplete coverage and insufficient connectivity. The times it did feel suitable were for large open spaces but the majority of the areas I wanted to construct were relatively tight. Due to these issues I decided that this technique was unsuitable for application in the maps I was trying to cover and I did not feel that it was worth pursuing any further.

While hand drawn paths would be the optimal choice to implement, it is far too time consuming with an unfinished project. During development I attempted to create some of my own pathing, for a single map, but I found this exercise was both very slow to return adequate results and time consuming to establish. If I had more time I would have experimented more with this technique, but to sit down and design exactly how you want the pathing graph to be connected is incredibly tedious.

3.2.1 Delaunay Triangulation

You can take the points of all obstacles in a map, along with the map corners, and run Delaunay triangulation over this set of points to generate a series of triangles. The Delaunay triangulation algorithm attempts to maximise the minimum angle in all triangles. The consequence of this is the creation of larger, wider triangles better used to efficiently map a space into a set of finite regions. Upon the creation of these triangles you can take the centre of these triangles, connect them, removing any edges that pass through obstacles, and you have a set of paths that map around all obstacles in the map.

The CGAL Libraries provided me with an easy method of setting up this skeletonization. All I had to do was define every corner of both the game obstacles and the boundaries of the map and add them into a 2 dimensional Delaunay triangulation. I then removed any connections that intersected with the game obstacles to restrict the range of movement of

the agents to valid areas. In doing so, there is no need to implement collision detection as the agents are only allowed to move along valid paths. Precomputing valid paths saves time that would be wasted testing whether every intentional move is valid.

I found this to be easiest method of producing adequate paths. Once the algorithm was set up it was applicable to most sensible levels I designed. At some points the paths did cut a bit close but with a reasonable tolerance. While Delaunay triangulation did not provide complete coverage in all cases it did provide adequate, simple and effective pathing in the majority of instances. As such, I decided to use this in preference to the other methods.

3.3 Particle Filtering

To implement intelligent tracking for my agents I looked for a method from the field of robotics. This was because robots are required to react, operate and plan in a real-time, partially observable environment. The same requirements apply to that of most AI in computer games. This method had to be: fast, efficient, low cost, flexible, and adaptable.

From my research, particle filters appeared to be the most appropriate tracking method for implementation in my game, not just for their speed and efficiency but also for their customisability. I felt that I would be able to implement particle filtering for tracking the evader based on my understanding of the technique as well as its previous success in Bererton's research[1].

During my research I planned to implement a particle filter framework which was parameterised, adjustable and modifiable to achieve different behaviours, complexity and game play. This became lesser priority due to the difficulty of creating a particle filter that was more capable than a randomly searching, scripted agent. Being a novice at implementation of higher order artificial intelligence techniques, it was incredibly difficult for me to get an adequate solution operational. I spent much of my time fine tuning and tweaking my code in order to get certain behaviours operating correctly. With this inherent difficulty, I realised that the later goals would have to wait, and the completion of a set of competent agents was far more important.

The particle filter was written by making extensive use of the CGAL library. Each particle was a simple datastructure consisting of a 2 dimensional point from the CGAL

library (x) and a double precision floating point as the weight (w). The filter was then just a container for these particles.

On initialising the particle filter, two parameters were passed: the level and the initial number of particles (this was also the maximum number of particles that can be resampled and maintained). I passed the complete level to ensure that any collision detection done was completely contained within the filter class itself. It was not an ideal method as it attributed to redundant data storage, but it was the least complicated method of ensuring some amount of encapsulation while reducing the visibility of the game logic.

When the initial particle count was set, that number of particles were randomly generated such that they didn't lie on the bounded side of any game obstacles. This was important to ensure that any future transition of such a particle would not occur within, or from, an area where the player cannot access (based on the game's rules). For the particle filter to function properly with realistic and sensible predictions, each particle must be transitioned in accordance with the rules of the object it is tracking. If predictions are not made in accordance, the filter would fail to produce transitions with any semblance to the movement of the object being tracked, and hence be relatively useless overall. Each new particle which was generated in the initialisation of the particle filter was given a uniform weight, i.e. $w_i = \frac{1}{N}$ for all $i \in P$ where N is the total number of particles and P is the particle filter. A uniform weighting needs to be set on initialisation to avoid introducing any probabilistic bias from the beginning.

I allowed for enough flexibility in my filter design to either increase or decrease the particle count on the fly. Through experimentation I found increments of 50 particles were reasonable. After an increase or decrease to the maximum number of particles being maintained, the filter resets by removing all particles and randomising a new set. I found that by not doing this I would unintentionally affect the current distribution, and as such, it was easier just to resample the distribution and start over.

There were three main stages to my particle filter update (which are mentioned on page 8). These three stages were:

- Apply the motion model
- Apply the observation model, and

- Perform Weighted resampling.

The following sections will explain both how I performed the application of the motion model and the observation model. Details of the weighted resampling algorithm (algorithm 2) I used can be found in the description on page 10.

3.3.1 Motion Model

I found modification of the motion model, $P(x_t|x_{t-1})$, could be used to achieve different behaviours. Since the motion model is responsible for propagation of the particles into the next state, it could also be seen that the movement of the particles is dependant on the implementation of the motion model. When trying to design an optimal particle filter, we do the best we can to provide the most accurate approximation about the way target's location will change between states.

Generally when implemented in robotics we use the particle filter to track the robot itself, when gaming, we may want to track the opposition. To compensate for this difference, instead of developing a motion model probabilistically indicative of the agent's own movement, we instead generalise a motion model representative of how our target is predicted to move. Each particle transition, predicted by the motion model, had to be tested in the same way as if the player was located at the particle. Without performing these collision checks, particles would be allowed to pass through walls, entering areas which would be forbidden to the player (based on the game rules). Without enforcing these rules the motion model would not be predicting the movement of the target effectively or realistically, and hence provide inadequate tracking.

I hypothesised that instead of trying to provide an optimal prediction, we could exploit this idea and manipulate the tracking behaviour of an agent by the changing or giving the motion model certain assumptions to apply to the distribution during propagation. I found that instead, this varying nature was necessary to create an accurate prediction and a near optimal motion model was required for the agents to remain competitive. I required a combination of the following predictions to keep the guards competitive:

- **No prediction**

  This would cause the particle to not shift in any way. Keeping a particle static allowed

for the possibility that the player was standing still. It was also required to keep the probability distribution from moving too quickly.

- **Assuming random movement**

  Assuming completely random movement and applying it would cause a particle distribution to spread outwards slowly, distributing itself around the available area. This was required to keep the distribution expanding realistically. Without any noise added to the motion model the particles clump and shift together in resampling.

- **Predicted movement**

  If we assume that the evader will always run in some direction the the cloud will then drift in this direction. This was perhaps the most useful addition to the particle filter as it allowed for some interesting prediction, not just tracking. I used three separate predictions to be chosen based on the situation:

  - **Towards the goal**

    This caused the particles to drift towards the player's target and subsequently made the guards occasionally check the area and make sure the player wasn't there. This helped significantly with predictive movement as it realistically modelled the movement of the player.

  - **Outwards from the position last seen**

    This was perhaps was one of the most effective options for tracking as it allowed the guard's to predict where the player was likely to be after escaping from the guards sight. This was fundamental for the guards to actually be able to effectively pursue the player. I found if I wasn't doing this the guards would quickly erase all the particles in the area and continue searching elsewhere, allowing the player to easily escape.

  - **Away from the guard**

    This allowed the guards to realistically predict the actions of the player if they were in close proximity to the guards. Hence, the guards would then be able to react and chase accordingly.

- **'Teleporting'**

  Allowing the particles to teleport means that sometimes particles will be (possibly) redistributed in the previously explored space, causing the pursuer to double back on himself or at least redistribute some of the distribution. I found that doing this was good for getting the particle field to move around a bit more. Setting the chance of teleporting too high caused the cloud to jump around the map too frequently and render the tracking relatively useless.

During my research I found that it was simple to modify the behaviour of the particles, but getting them to behave as I wanted was far more difficult. Finding the correct probabilities used to determine which prediction would be used was quite a tedious process. For instance, too little random movement and the clouds would not expand, too much and they wouldn't drift. As such, I spent significant amounts of my research trying to find a decent motion model that could accurately track and predict while remaining competitive. My research quickly became less about experimenting with behaviours and more about establishing an appropriate motion model that allow the guards to pose a challenge.

Algorithm 3 is a generalised, pseudocode representation of the final version of my motion model. I found the algorithm quite useful both in predicting both player movement and tracking. Subtle changes to the probabilities can achieve varying behaviours but through both personal experimentation and early trials I felt it was the most suitable.

While this motion model is not completely indicative nor 100% accurate, it provided a means by which the agents could track and predict reasonably well. Results of playtesting undertaken to test the effectives of this predictive agent against a random agent can be found in section 4.4.1.

3.3.2 Observation Model

In the final design each guard had a sight radius and the ability to hear the player running from a certain distance away. While the player was in the agents sight radius they took damage. Any particles within this sight radius were evaluated as being seen in that timestep.

Modifying the observation model, $P(o_t|x_t)$, may be used to achieve different game dy-

Algorithm 3 Generalised version of final motion model

```
Let x be a random floating point between 0 and 1
if x < 0.1 then
  No transition
else if x < 0.5 then
  if The player has been seen recently then
    Predict movement outwards from last position
  else if Guard's proximity is close to the particle then
    Predict movement away from guard
  else if Players goal is present then
    Predict movement towards the goal
  end if
else if x < 0.999 then
  Assume random movement
else
  Teleport the particle randomly
end if
```

namics. In an optimal simulation, the observation would be 100% accurate; if we see the target, we know the target is at that location. In terms of a realistic implementation, especially in robotics, we may choose to have an observation model with a slight amount of noise to account for the possibility of inaccurate or distorted input. This idea can be applied to this pursuit-evasion scenario. Maybe the thief is standing in some shadows when a guard sees him, or maybe the guard is just a little blind. In adjusting the probabilities of producing false positives and false negatives we can modify how the agent will process the information coming from their vision, and hence adjust how they maintain their belief state because it directly affects the reweighting process. Again this can attributes to adding more depth to the game.

	Thief seen at x	Thief $\neg$ seen at x
Thief at x	1	0
Thief $\neg$ at x	0	1

Table 3.1: Optimal Observation Model

	Thief seen at x	Thief $\neg$ seen at x
Thief at x	0.8	0.2
Thief $\neg$ at x	0.1	0.9

Table 3.2: Example of a 'blurry' Observation Model

Algorithm 4 is a pseudocode representation of the final observation model I used for the guards sight.

Algorithm 4 Generalised version the observation model

```
for Each particle i ∈ P do
  if Player has been seen then
    if Particle is within guards sight then
      if Player position == x_i then
        w_i = w_i * 1
      else
        w_i = w_i * 0
      end if
    else
      w_i = w_i * 0
    end if
  else
    if Particle is within guards sight then
      w_i = w_i * 0
    else
      w_i = w_i * 1
    end if
  end if
end for
```

A problem I encountered with the application of this observation model was the case when a particle was not present at the player's location. To solve this issue, when the player was seen I added a particle at the players position with a weighting of 1, i.e. $w_i = 1$. This was a means by which I could have the particle filter working effectively and not disclose any information which would be seen as outside of the guards senses. It is a sensible and simple solution to the problem, which in turn, forces the particle filter to track far more accurately.

In addition to simple sight, I found I needed to extend the agent's sensors in order for them to operate realistically. Simulated hearing seemed to be the most sensible addition. My playtesters preferred having some repercussions for running around in a stealth game instead of taking the quiet approach. Having the guard be able to hear the player run, if the player was within a certain distance, and calculate the probability of them being at the position as such equates to $P(X) = 1 - d * (1/maxD)$, where d is the distance of the player from the guard and $maxD$ is the maximum distance at which the guard can hear. With the

addition of hearing, the guards could perform far more accurately without necessarily being given information which should be hidden. This method not only increased the predictive accuracy of the particle filter but also added a lot more depth to the game itself. Since there was some disadvantage to running everywhere, the player was more likely to attempt to sneak around the maze as intended. Having the ability to alert guards to a certain area also allowed for the players to create distractions.

3.3.3 Number Of Particles

Modifying the number of particles changes the amount of information accessible in the model. Increasing the number of particles increases the intelligence of agent. This is because having more particles gives a more accurate approximation of the probability density function, thus allowing for a more informed search. Conversely, the less particles maintained, the poorer the approximation of the probability density function. Having a poorer representation of the probability density distribution causes a more randomised search.

Based on my playtesting I found that a particle count of around 200-250 was a decent distribution in terms of speed, efficiency and cost. The amount of particles maintained per filter became a matter of finding an equilibrium between cost and speed. As stated earlier in section 3.3.1, each particle transition required the same testing as the players movement would undergo in order to retain realistic predictions. It can be seen that the addition of more particles would lead to more calculations, and conversely, subtraction of particles would lead to less.

Another consequence of the maintenance of more particles is an increase in the amount of information to be analysed during the decision making processes. This carries the same effect (i.e. more particles, more calculations). These issues concerning both the motion model calculations and the decision process cost will be further discussed in sections 4.4.1 and 4.5 respectively.

An evaluation of the number of particles and their relation to the perception of the agent's intelligence will be discussed in section 4.4.3.

3.3.4 Sharing Information

Another major part of my planned research was to look at different methods of sharing information between the agents. There were three possibilities that I wanted to look at: a single shared particle filter, multiple unshared particle filters and multiple shared particle filters. I first experimented with a single shared particle filter. I found this to be inadequate for the collaboration and decision making I wanted to use. I then moved on to multiple unshared particle filters, which I found were far more useful. I did not get to try multiple shared particle filters because of time constraints. The following is a short discussion on my experiences with the first two.

A single shared particle filter

There is a single particle filter for all the guards to maintain and update. Some advantages of this is that it requires one update per time step and requires less memory to maintain. This being so, there is the possibility of using more particles for a better approximation. The downfall to this approach is that the observations must be combined into a single update and in terms of the grand scheme of things. Each of the guards updates will have less of an effect in total.

I found during my research that this approach to sharing information was quite poor. All agents converged on the same spots and finally ending up uselessly following each other around the map.

Although the amount of management was dramatically reduced, the effective use of the information gathered was quite minimal. I was unable to adequately handle and make use of this shared information so I moved on to multiple, unshared particle filters. I abandoned this approach and moved on to multiple individual unshared particle filters

Multiple individual unshared particle filters

Every agent maintains their own particle filter. No sharing is done so each guard keeps all the information to themselves. While this simplifies the problem of sharing data, such that it doesn't exist, you also lose much of the information that could have been used to assist in powerful and intelligent collaboration. One positive is that since each agent

maintains their own distribution, said maintenance is a lot simpler to carry out. As such, each distribution is tailored to a guard's expectations and can most effectively be used by themselves without the risk of having their effective information being diluted.

During my experimentations, multiple, unshared particles filters appeared to be the far more effective than the single particle filter. It was far simpler to control and manage each particle filter while using the information to its greatest effective. All information was only applicable to a single guard, as such, all information was relevant. This made it far simpler to use this information for decision making. The real downfall I found when experimenting was that since there were more particle filters there was also more particles to manage, and hence, far more calculations to perform. Consequently, having too many guards performing too many updates slowed the game dramatically. Finding a balance between the number of obstacles, number of particles and number of guards became paramount.

3.4 Decision Making and Planning

The majority of my research was on particle filtering, but I attempted to explore some possible decision making and planning techniques. Not only is it important to have a reliable, intuitive and intelligent tracking system but there also needs to be an intelligent way of analysing the gathered data. This data needs to be assessed and a plan constructed about how to best approach the understood situation. Having never worked with a particle filter before I focused mainly on greedy methods which appeared far simpler to implement. These methods were explored during my own experimentation. When I came to the last method, the "most dense waypoint", I decided to stop and use it in my final design.

3.4.1 Naïve and Greedy Methods

There were four main approaches I wanted to examine. Each method focuses on exploiting some aspect of the data. They provide and are based on some level of reasoning about the data. In this scenario there needed to be a balance between 'exploration' and 'exploitation'. Exploitation is making a decision with the highest expected payoff and exploration is making decisions to gather more information about the state. Each will have their own

effect and considerably change the way the agents reason, contributing to the prevalence of behaviours. The algorithms I experimented with are as follows:

- **Pick a particle randomly from the distribution to explore to**

  Making a random choice will probabilistically result with the selection of a particle that comes from a large cloud, making discovery more probable. Due to the stochastic nature of the selection, there is not a bias either on exploration or exploitation and should in fact provide a relatively balanced approach to the solution. A pursuer using this naive method would be wandering around the map relatively randomly.

  This turned out to work completely as expected, the movement was incredibly random, and hence quite useless. This method works excellently when you have only a few particles on the map but as you build up a better distribution, it becomes far less effective and the guards seem to resort to acting completely randomly. All in all this was a suitable jumping off point to start working with, and could even could be used quite efficiently when there are only a few particles being monitored.

- **Pick the closest particle the distribution to explore to**

  By exploiting spacial closeness we look at the particles in the distribution and choose to explore to a particle that is closest in reach. This can be considered a very simple explorer. This naive method can be seen to be simple exploration until the target has been sighted.

  As suspected, this method provides quite basic exploration which easily gets confused. It is hard to accurately judge the "real" distance of a particle since you may be comparing a particle through a wall. Consequently, this method was about as useful as the random selection. Nothing noteworthy to discuss was achieved using this method.

- **Navigate to the mean of the posterior distribution**

  This method is entirely exploitative. Looking at the distribution, we choose to navigate to the direct middle of the entire cloud. The idea behind this is the denser the cloud the more likely the target may be there, so when we have a relatively dense cloud the middle of this mass has a good chance of representing the player's location.

As such, this method can be seen to be globally greedy. This method obviously will fall short as the particle cloud becomes sparse.

As suspected this method worked quite well when the particle clouds were dense. The guards easily explored to the middle of the distribution and (hopefully) saw a number particles along the way. This worked well for exceptionally open spaces. This method's downfall was when it was used in levels that were relatively closed and small. Since the mean was being tracked there was no discernible way to check whether the mean was indicative of where the particles actually were. Also when the fields were quite spare, the guard had a tendency to explore to the middle of the map (also the mean point) and just sit. This was because the cloud was so evenly distributed the mean of the cloud was a relatively useless measure.

To counteract this problem I developed a 'Pseudo-mean' method which a handful of particles is taken and analysed instead of the entire collection. In this way I avoided a lot of the problems that come with only exploring to the middle. This was a pretty successful method for quite a while, it works nicely for large open spaces and reasonably for more confined spaces. However, once I implemented the "most dense waypoint" algorithm, this method became obsolete.

- **Navigate to the most dense waypoint**

  If we examine the location of every particle and calculate where they mostly lie, we can see when the most dense region of the map. Hence, we find the most probable point. This is because where the distribution is most dense is where the agent pictures the player to be most likely based on current observations. It should be obvious that this technique would be far more accurate at pinpointing the position of an escaping player rather than the previously mentioned techniques.

  This method is quite expensive in terms of number of instructions, but was also one of the best performing. For each particle I found its closest waypoint and added the weighting of it to a count. After each particle was analysed, I took the waypoint of maximal weighting, which was also the waypoint where the player was most likely to be found around, and chose to explore to it. Although this method was far from efficient, it did provide the most accurate planning and successful chase mechanism.

Because of this inefficiency it is very time consuming to run the algorithm, and so I tried to apply the same 'Psuedo' trick to it, but results were relatively poor.

The most dense waypoint, being the most effective technique I experimented with, was selected for use in the final design. While it tended to be slow and costly to run, it generated the most effective plans during my own experimentation.

Chapter 4

Evaluation

The following will be an assessment of the experimentation undertaken and the resulting product, "Art Of Stealth". Many of the goals I had originally set for myself were not met and some experimentation I had planned went untested. Despite this, I will attempt to provide a balanced assessment of the work I did, the quality of the final game and the degree at which I met the aims I completed. Firstly I will discuss how I went about playtesting my game in order to receive feedback on my design. In further sections I will evaluate the design with respect to both my personal experimentation and also to the findings of these playtests.

4.1 Playtesting

I found it difficult to adequately evaluate the results of my thesis as it is problematic to test a range of different players, determine their competency and analyse their input. I went through three stages of playtesting through the development of my thesis. The first was very informal. I had a number of friends and colleagues come in and play the first version I had established and asked them to comment on it, assess how intelligent they felt the guards were and tell me what the game was lacking. The second and third series of tests were more formal with questionnaires and multiple versions of the game, each with a change to the logic. I attempted to get my playtesters to assess the intelligence of the AI. I attempted to find a rating system by which I could generalise their answers, but I feel I

cannot properly evaluate the resulting intelligence of the game via these methods. I was able to extrapolate a general consensus about the grading of the different test versions.

4.1.1 First Stage

The following is a breakdown of the findings from the first round of playtesting. The informal testing mostly focussed on cosmetic and mechanic issues. Most of these results that made it into the design are explained in section 3.1.

Players also felt the game of "evasion and then escape" was a little dull. They suggested that since the player is a theif, they should steal something from the guards. To make things more interesting for my players I changed the gameplay so the player first had to steal a gem before being allowed to escape.

Players wanted the game to move a little faster; they felt the guards were searching too slowly and should be more alert. As a result of this I increased the speed of the guards and implemented a run/walk mode for the player. The guards would also then become alert for a period of time having seen the player or realising the gem had been stolen. While alert they would move around 1.5 times faster.

I noticed during the first play that players were confused by the counter which told them they when they could escape. They perceived it as a "score" and attempted to score the most points possible by sitting in the maze and waiting. To correct this behaviour I changed the counter into a progress bar, which worked very well.

As part of this first round of playtesting I also asked the players to design a level that they would enjoy playing. I asked them not to make it too complex or difficult, but something that they would like to see in my game. I received some good feedback on this and some interesting designs. Some of the viable and interesting designs made it into the final version of the game with a few modifications. This being my first game project, as stated earlier, I did not have any experience with level design. Receiving these contributions from my playtesters so early in the design cycle allowed me to create a far more interesting environment for them to test.

Other suggestions mostly pertained towards the look and feel of the game. Some comments collected referred to theme or setting changes, addition of new mechanics and changes

to the animation and graphics.

It is because of this first round of playtesting that I developed a far more interesting game than the initial game engine featured. Had this been a test bed there would be no issue, but being a project that was intended for play it was a goal for it to be entertaining. My first round of testers were incredibly helpful in this respect with design and the final result if better for that support.

4.1.2 Second and Third Stages

A more formal approach was taken in the second and third stages of testing. Surveys were given out to help me assess the difference between three versions of my game: the first version with only random prediction, the second version with a low particle count and full prediction, and the third version with high particle count and full prediction. The aim of these playtests was to evaluate the differences between what I hypothesised were the inferior versions (the first and second) and compare them with what I believed was my final version (the third).

The most helpful input on the artificial intelligence aspect of the evaluation was from both academics and colleagues also studying computer science. They were insightful and enjoyed playing with the particle filter enabled on screen so they could watch the computer "think". Many comments and reflections were taken into account from the second playtest and implemented in the third. The third and final round of playtesting was mostly just for final evaluation and development of ideas for future implementations.

Players acquired skill mastery during the first few play throughs. This would in turn have effected how challenging they found the subsequent versions. To compensate I tried to mix around the order in which the players attempted the challenges, but I'm not sure how successful I was. This style of playtesting is incredibly hard to analyse and employ. Had another option of testing been available I probably would have attempted it. The majority of my evaluation was not obtained through the surveys but through actually talking to my playtesters after the game, or listening to their vocalised thoughts and frustrations as they played.

Interesting results were gained from the playtesting. There is not enough testing for

them to be statistically valid but there were general patterns emerging. 11/17 players correctly realised that the random version was just predicting random movement. 11/17 players realised that the guards in the 2nd version were acting with some purpose. 14/17 players felt that version 3 had guards also acting with some purpose, or at least, could see they were attempting to defend objectives. 8/17 players found version 2 more enjoyable to play. 7/17 players found version 3 to be the most enjoyable. Only 1 player felt that version 1 was more fun and 1 player couldn't tell the difference.

As far as determining which version was the most enjoyable to play with, the even split between version 2 and 3 tells me (along with players feedback), that the predictiveness incorporated into the motion model was fundamentally needed for a challenging and fun opponent. Many players were consistently surprised with zeal at which their opponents guarded the players objectives. Perhaps the most encouraging feedback was having my playtesters told me that the game was actually difficult. Seeing players lose, but not giving up, was also a sign that the design of competitive guards was successful.

Feedback also indicated the enjoyment would occasionally be overshadowed by increasing degrees of latency associated with version 3, and hence the 2nd version would be favoured. Because of this I had to find a number of particles which allow the guards to retain a reasonable amount of intelligence, while having the game relatively uninterrupted with latency caused by frequent and costly calculations. While a good degree of intelligence, insight and ingenuity was needed to create an interesting, enjoyable opponent, this fun can also be interrupted by a lack of flow. A player will not easily suffer through a game rife with latency. Players would obviously prefer to play a version in which they felt they had more control. As stated in the design section 3.3.3, I found that maintaining around 200-250 particles per particle filter gave me a reasonable result.

Other less notable insights, but still important, were related to cosmetic aspects or additional ideas to create more interesting gameplay. This playtesting gave me insight into how to improve this project for future reference.

4.2 Platform

The following is an evaluation of the advantages and disadvantages of the platform I worked on for the duration of my thesis, i.e. Jimmy Kurniawan's original "Art Of Stealth".

In hindsight, in some respects, the selection of this platform was not the best choice for such an implementation. It was discussed at the start of my selection whether I would prefer to write my own engine or take on the work of Jimmy Kurniawan. I had chosen to take up Jimmy's incomplete game engine under the belief that, this being the first game I have ever had a hand in writing, I would not have the skills necessary to have completed something of this magnitude without a commencement point. I still maintain my belief that my skills and knowledge would have been inadequate to begin working on this project without the initial help. Thanks to Jimmy I had a working (albeit incomplete) platform I could easily start modifying. Jimmy's work provided me with a system I could demonstrate my ideas on, work with the mechanic I had chosen, and end up producing a game that met a fair few of my initial design goals.

Working with Jimmy's game engine allowed me to jump right into development of the AI. There being an already existing platform for me to start working with was an incredible initial boost. After sitting down with Jimmy and reading through the code I found it relatively easy to understand, although through the development cycle certain issues did arise. Most of the animation, although simple, was already in place and all I had to do was modify it slightly for my uses. Many algorithms for the geometry were also implemented, and this reduced the background reading I had to do in order to fully grasp the fine details of some of the techniques.

This being said, using the existing infrastructure resulted in a lot of calculations which could have been avoided if the code was written around the particle filter, rather than the opposite way. The addition of the particle filter on top of the existing game did not make for efficient processing. I will discuss the difficulties of using this approach in the particle filter section 4.4. Had I chosen to write my own platform there would have been much more thought put into the construction of a game based around, working with and assisting the central mechanic, rather than the mechanic trying to cope and work with the game in order to be useful.

In section 3.1 described how I simulated threading by interleaving. The results of this technique varied, it had both advantageous and disadvantages. The consistent cost of running the artificial intelligence was reduced dramatically. This was because it was only run ever few timesteps. While a lag sometimes occurred at small intervals, the overall speed of execution was increased. Since I wasn't running the computationally intensive code every timestep, accumulators had to be kept to count the amount of time passed between executions. Finding the right interval at which to update was a matter of trial and error. If the particle filter was updated too slowly the results were less useful as they became less indicative of the player's movement. This was because the motion was made to make real-time predictions and constraining it to update on intervals reduced its overall effectiveness. As the observations had to be synchronised with the particle filter, too much time between updates caused issues with the agents sensors which were obviously meant to operate in real-time. This was much the same for the decision system, if the update was run too infrequently guards would either exhibit erratic behaviour or respond too slowly to stimuli.

Coming with both positives and negatives, Jimmy's game engine was a much needed starting point in the development of my thesis. Without it I would not have been able to produce the work I now have to such a sound level.

4.3 Motion Planning

Motion planning was quite a simple task when I left it up to the algorithm. I only did a little personal experimentation with it as I found Delaunay triangulation was the easiest to use based on speed, efficiency, coverage and effort (both in terms of the amount of work required to specify these paths and also the number of calculations required). Figure 4.1 is an example of the pathing created automatically by Delaunay triangulation. The following is an evaluation of Delaunay triangulation as a motion planning technique throughout the entirety of the construction of "Art Of Stealth".

Delaunay triangulation was the simplest method of motion planning. Using the CGAL libraries to conduct both the selection of waypoints and the creation of the paths allowed me focus on the particle filtering and tracking. In terms of its application in computer

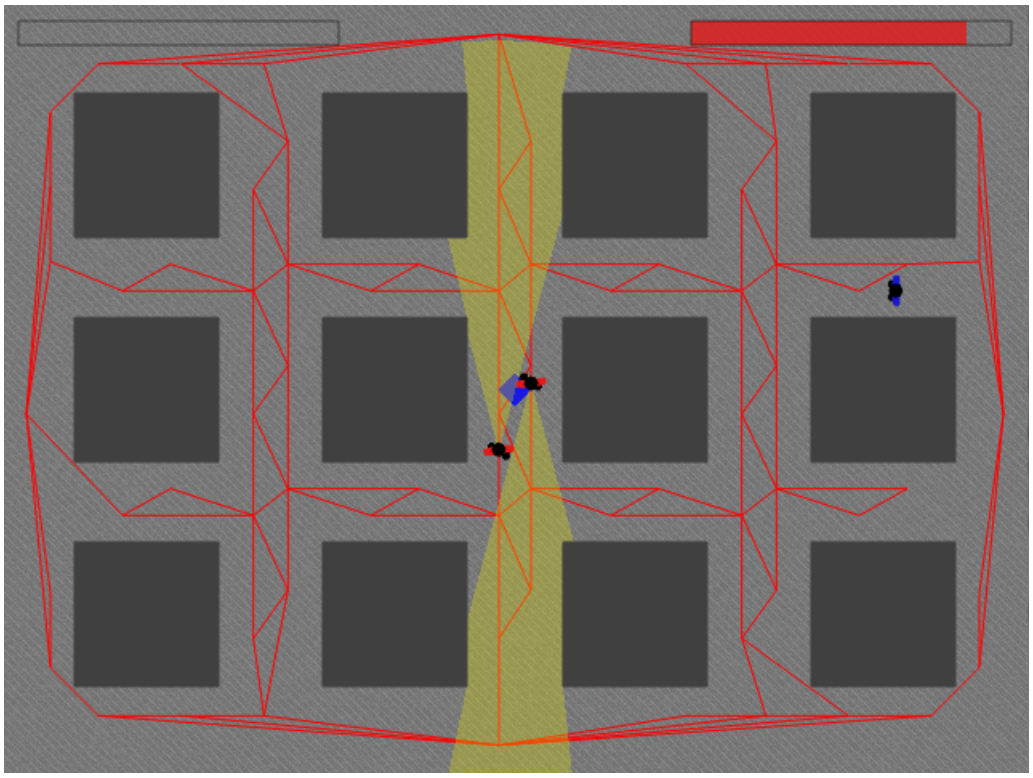

Figure 4.1: Path construction using Delaunay Triangulation in Art Of Stealth

games as opposed to robotics, it has both negatives and positives.

The Delaunay triangulation worked best when used with a lot of objects on the field. If the number of objects was too small, the paths created to maneuver around the area were quite sparse. This incomplete, which can be seen in figure 4.2, became quite a big problem as it forced me to create more objects and alter their configuration to achieve adequate coverage and completeness. This had the adverse effect of increasing the number of calculation required to run any collision checks.

The process of adding more objects to construct interesting spaces became a balancing act. In order to adequately map the entire area I had to make sure that objects I placed would allow for such configuration based both on size and placement. Broken paths would be established if objects were added in a configuration that wasn't feasible based on the constraints, as seen in figure 4.3. This was a large drawback as it meant a lot of the levels that would have had quite interesting configuration had to be modified to allow the algorithm to accurately perform its job.

The Delaunay Triangulation tended to create far too many unnecessary waypoints.

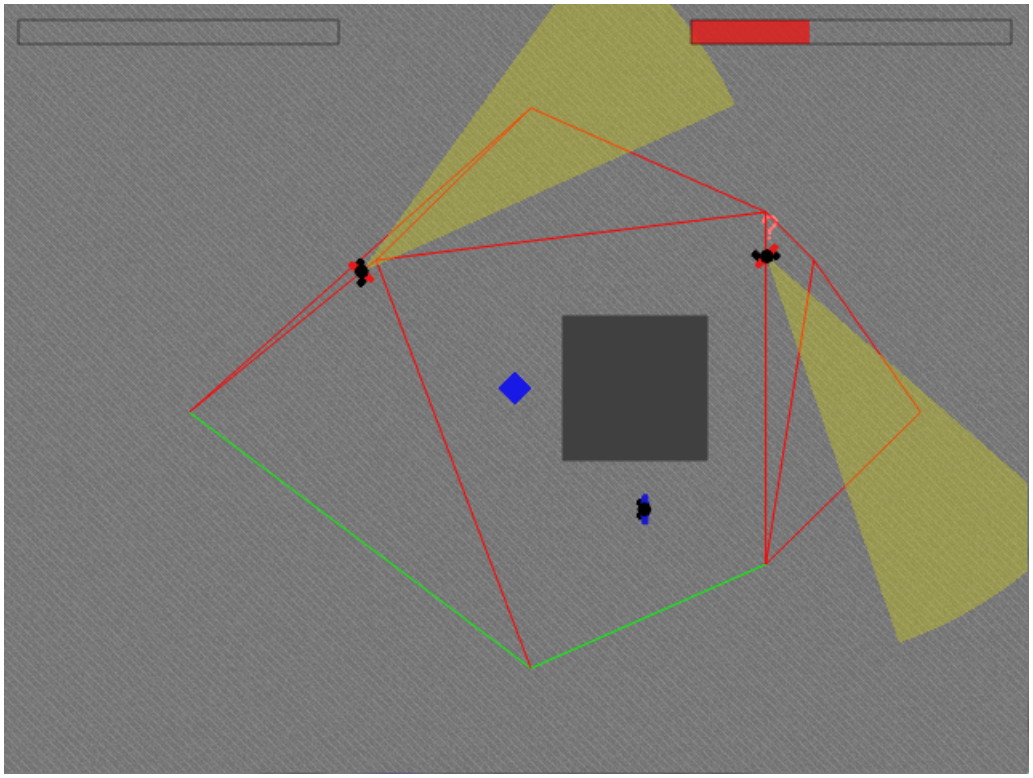

Figure 4.2: Incomplete path construction using Delaunay Triangulation

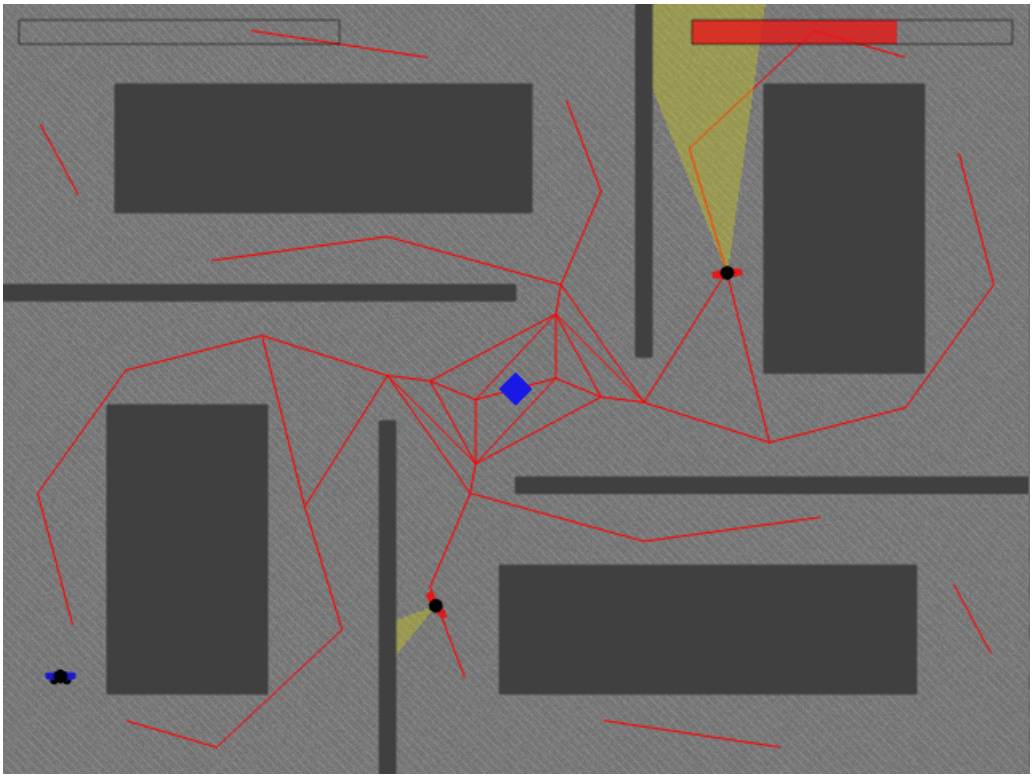

Figure 4.3: Inadequately performed path construction using Delaunay Triangulation

While this did provide some interesting pathing patterns, it also negatively impacted on the amount of calculations required for certain decision making techniques.

Once the algorithm was set up all the paths could be generated without any assistance. This made testing incredibly simple because the work would be done for me no matter which configuration the level was in. The automation of path creation was quite worthwhile, especially as I was constantly making changes to the levels. In terms of a final implementation, I feel a well thought out, meticulously planned, hand drawn approach would work far better. Where the game is at this point, still somewhat incomplete and unoptimised, Delaunay triangulation is an adequate choice although it has issues and tends to produce a fair amount of redundant output.

While this method wasn't perfect for the implementation, it served incredibly well for a work in progress. The ability to automatically generate pathing without any specific input, apart from the ever changing level design, streamlined testing and development tremendously.

4.4 Particle Filter

The particle filter, when designed effectively and efficiently as possible resulted in an interesting and useful method of tracking. The more accurate the player model, the better the prediction becomes. The issues with inefficiency stem from the particle filter operating in an environment not specifically designed to enable it to reach its absolute potential.

The implementation I produced was a relatively naive approach. I did a lot of research into how a particle filter should operate, the mathematics behind the computation and where they would be most effectively used but implementation details were quite scarce. I talked to a few academics in order to get a feel for what I should be doing in order get the most of my implementation, but in some respects I fell short due to this being my first try. If I was to implement a particle filter again, I would ensure that the mechanics coordinated well with the structure and geometry of the world in which it was operating. The majority of issues about efficiency, cost and time mostly stemmed from an inconsistent base. An underlying base which facilitated the operation of the particle filter as a mechanic would lead to far better results in terms of its predictions, cost, speed and efficiency.

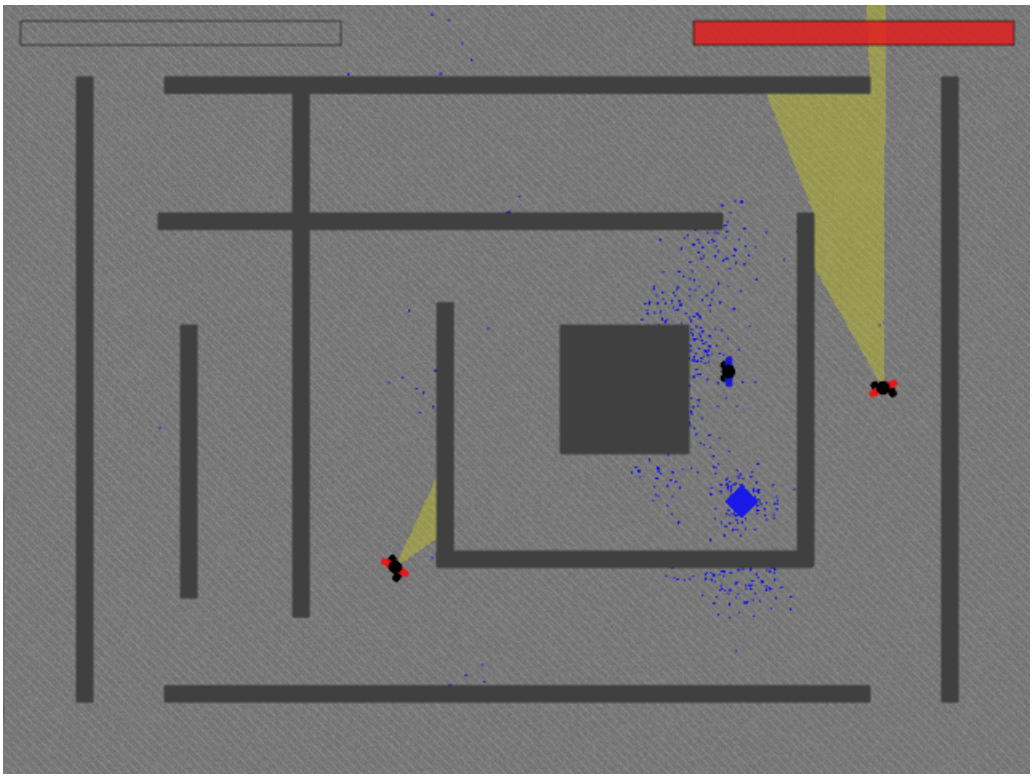

Figure 4.4: Particle Filter actively tracking player in Art Of Stealth

4.4.1 Motion Model

The majority of the work in my thesis was to not only design a working particle filter but also demonstrate that this mechanic could be used in place of traditional scripting. The construction of a predictive, intuitive and realistic motion model played the largest part in this exercise. The motion model was key for both predicting player activities and providing the decision process with intelligent data.

I feel that the particle filter with an intuitive motion model was an adequate and realistic replacement for standard scripting that would be used predicting a player's movement. The final version of the motion model could have been far more complex and robust offering a greater level of predictability. However, due to the amount of calculations required to maintain this prediction and tracking, some trade-offs may have had to been made, further slowing down the game. For the level of accuracy and the speed of the game, my final version of the motion model performs relatively successfully, modelling realistic reasoning and offering competitive play.

Some interesting behaviours emerged from the motion model's predictions in combi-

nation with the decision processes. The most notable perhaps was a defensive, guarding behaviour. I never programmed the guard to explicitly observe the gem nor the exit. In the motion model I included a prediction that the player would attempt to move towards the goal, whether that be the gem or the exit, if either were present on the field at the time. Due to this prediction many of the particles would be transitioned towards these points. As such, when the decision process was run, and the most dense waypoint was found, the guards would tend to navigate toward these goals. They would tend to check these areas, erase the particles in the vicinity, and return to their search. A number of factors affected the frequency of this behaviour, but it was apparent in every level. This was a very large success, having demonstrated that emergent behaviours are possible from just using intelligent techniques, they could possibly serve as a replacement for traditional scripted methods.

This intelligent and intuitive motion model appeared to be the defining measure of intelligence in the agents. My playtesters, not being able to see the underlying logic of the particle filter, judged the intelligence of the guards based on how well the guards seem to predict their movements. Having a motion model that encompasses the basics of the player's movements and can predict them with some accuracy made a huge difference to the player's perception of the guards intelligence. During playtesting, players could mostly distinguish between two separate models - one with almost all random prediction and the other with predictive movement. Out of my playtesting nearly all testers could pick the difference between the random agent and the predictive agent. As such, I believe that the predicative motion model was indeed the most intuitive, inherently intelligent and greatest factor contributing to the agent's intelligence. The number of objects directly affected the worst case number of collision checks by the number of existing particles, per filter update. Every transition of a particle had to be tested as if it were the player. The particle filter tracks the predicted movement of the player, as such, the predicted movements must be consistent with the movements of the player. With 12 game objects and 500 particles there were 6,000 intersection checks per run of the algorithm. This large number of calculations being executed for every update caused a noticeable amount of latency which detracted noticeably from the flow of the game.

The motion model is incredibly sensitive to small changes in the probability of transition. By changing the model probabilities by even a few percent, very different predictions can be seen. My original hypothesis that the motion model could exist as an avenue for the creation of unique and interesting behaviours was correct, but the creation of these behaviours is quite complicated and sensitive. To even obtain a reasonable, accurate player model took a lot of study, self testing and playtesting in order to encapsulate the basic gameplay. To alter this to emphasise certain aspects of play would be relatively easy, but tuning the model to provide the experience being sort after is another issue. An example of this issue that I encountered was when the predictive random movement was not occurring frequently enough. When the motion model was applied to the distribution, it resulted in large amounts of clumping. Clumping offered none of the required uncertainty which should be prevalent in a stealth game, and resulted in an incredibly centred search. The opposite occurred when the probability of random movement being applied was set too high; the level of realistic, accurate prediction decreased and the distribution expanded outward. Hence, the search became far more randomised.

Although I had initially planned to have the motion model not operating optimally, this preconception was dispatched with almost immediately. The motion model had to be predicting as accurately as possible for the guards to act as competitive opponents. As it was, the game was too easy for a lot of players because of factors external to the operation of the particle filter (i.e. the gameplay, level design, core mechanics, etc). The game is simple enough to be accessible by a large range of players. With competitive, relatively intelligent adversaries who operated as optimally as possible on the data given, both non-gamers and gamers alike would be able to beat the game after only a few tries. While difficult, my playtesters took a lot of pleasure in being able to outwit the guards and only saw it as an increasing challenge when they were beaten. Few of my playtesters stated that the AI was either too intelligent, or conversely too stupid, in its final state. Any ease or difficulty encountered was mostly attributed to level design and chance.

4.4.2 Observation Model

The observation model was not a large part of my initial design goals but attributed to adding depth to the game and assisting in the emergence of new behaviours.

The observation model, being only slightly experimented with, is hard to evaluate. An optimal observation model made sense in the setting with the already established mechanics. If I was to implement "dark" areas in which the guards found it harder to see it may have made more sense to have a slightly "blurry" observation model. I believe that a method like this, with a particle filter being adequately implemented in a game environment, could achieve the behaviours I was looking for. This approach would also require a large amount of supportive scripting. It is not enough for the guards to inadequately recognise you - the player must also acknowledge that this is happening, otherwise they just question the guard's intelligence. The possibility of adding dark areas (or something of the sort) would possibly impart the experience for which I was looking. Unless the player actively recognises what is happening, the effect is quickly lost on them. This being such, my experimentation with the observation model was cut short as I felt without extensive modification to the game engine any work done or ground made would be lost upon the players, and hence, irrelevant.

While a optimal sight observation model made sense in this respect, a suboptimal auditory model made the gameplay far more interesting. Giving the guards the ability to hear the player running made the game feel a little more realistic and allowed for the player to play with the agents in order to trick them. This feature subsequently added a lot more depth to the game while reinforcing the stealth elements of the game by having a negative repercussions for playing the game "incorrectly". This addition was a suggestion from the playtesters based on a desire for faster gameplay and my wish for players to maintain that covert playing style.

I think in terms of its contribution, the current observation model is a reasonable and intelligent solution to the issue of visual and 'sound' detection for the current state of the game. None of the playtesters felt that this ability to hear the player run was unrealistic or unbalanced; all players who ran knew they were doing so at their own risk and took that into account during their play. Comments were made that the hearing may be a little too

good in some respects, but it was accepted as is.

In conclusion, the final version of the observation model provides a relatively simple and efficient sensory model and works reliably and realistically for the current implementation. A change in the lighting mechanics would warrant a change to observation model.

4.4.3 Number of Particles

My hypothesis was that an increase in the information allowed to be maintained by the guards (i.e. more particles per filter) would increase the intelligence of the agent while making the guard a more difficult opponent. Instead of being able to determine whether this was true, I found some interesting results.

My results experimenting with this aspect of the particle filter were mixed. It was found that the playtesters could recognise the difference between an agent tracking a large distribution (about 400) to one tracking a small distribution (about 50) but for different reasons. My playtesting involved trialling two different versions of the game, each tracking a different number of particles per guard: in one version each guard tracked a distribution of 50 particles, in the other version each guard tracking a distribution of around 400.

Some of my playtesters felt on average that the version with 50 particles was far more responsive, quicker to react and better at prediction. Most players felt that the version with 400 particles was far better than the version with random prediction but also saw that it was quite slow compared to the version with less particles being maintained. This was true - the time and cost to maintain the larger distribution of particles took far longer to both update, observe and resample, but to also query as part of the decision process.

After I recognised this recurring pattern I questioned the players about it. Most agreed that the speed of the version tracking less particles was more interesting to play as it reacted far faster to subtle changes in the game. Although it couldn't predict as well, it was a more challenging opponent in some respects because it was able act on the information gained during the game far more quickly.

Another possibility for the different of opinion is that although the guard can retain a lot more information and analyse a far more complex probability distribution, the methods it used to query that underlying knowledge base were far weaker in terms both efficiency

and complexity. The greedy methods I implemented could not optimally take advantage of all the information gained, and as such wasted a lot of computing time and power deriving fairly similar results independent of the number of particles being maintained.

Maintaining a too low particle count would cause problems as well. Using the weighted resampling method, if the majority of the particles were erased from a sector, particles remaining there would be of higher weight. Since the particles with higher weight are more likely to be resampled, I found with a small particle count areas which should have been classified as clear became repopulated with particles on the next timestep. This resulted in the guards checking in a single area indefinitely. With a larger particle count, this behaviour was less likely too occur.

Also attributed to the speed of the game was the general responsiveness of the graphics. The version with the higher particle count had a much lower FPS (frames per second) count. The players testers felt that playing the version in which the lag was minimal was far more satisfying. Players like to have a realistic, real-time experience when they play; having to wait for the logic to catch up to the rendering becomes frustrating and breaks the flow of the game.

To deal with these speed/cost difficulties I had to attempt to find a suitable particle count. The number of particles needed to be chosen such that reduced the lag caused by numerous, costly calculations while providing a decent prediction with an adequate knowledge base. As stated in 3.3.3, 200-250 seems appropriate.

4.5 Decision Making and Planning

While decision making and planning was not a focus of my thesis there still needed to be some experimentation with a number of techniques, if only personal. The algorithm I implemented for the final version of "Art Of Stealth" was the "most dense waypoint" greedy method. While not optimal, it provided reasonable planning, especially when the distribution was mostly unimodal. The following is an evaluation based on my personal experimentation with the methods mention in the design section 3.4.

Methods which generated a new point based on the existing distribution (i.e. navigation to the mean of the posterior distribution) were relatively useless. It worked for small particle

distributions, especially if the probability density function had only a single peak. When the distribution was split into a multimodal approximation, the technique fell apart - agents stood in the middle of the map walking between two points not evaluating any particles because they were unable to reach them. Upon reflection I can see why this method was destined to fail. Assuming we had only a single peak in the distribution, this technique may perform adequately (even better if the space was large and open). However, by using a particle filter which allows for multimodal approximation, there is no easy way to discern more than two particle clouds from each other. As the clouds are not easily separable, the creation of a point being the mean of all clouds results with a point somewhere in the middle of all of them. This point was normally in a place that does not accurately reflect the information gathered or the distribution from which it was inferred. This being such, it is obvious that a point created to act as a representation of all data obtained, which is inaccurate, is not a suitable choice upon which to base any planning assumptions. When these results were observed I attempted to try the "psuedo-mean" version, but this resulted with similar decisions. The algorithms were subsequently scrapped both in terms of cost/effectiveness and inaccuracy.

The most effective algorithm I experimented with was the "most dense waypoint". Each particle is compared with every waypoint and whichever waypoint it is closest to has the weight of the particle added to it. After every particle has been looked at, the waypoint with the largest weight is set as the point most likely for the player to be found at. Currently for a particle count of 500 with a modest waypoint count of 30 would require 15,000 distance checks per run of the "most dense waypoint" algorithm. If run every timestep, it would create far too much lag for the game to be playable. With this attributed cost to the algorithm, it can be seen that using less particles and/or having a configuration space of less particles, means less calculations. This caused the same balancing act between cost and time as I have experienced so much throughout this project.

My solution was to interleaved the running of this algorithm so that it only recalculated the agents route every 8th of a second. This seemed to perform relatively well and only created a little latency. This method, while slightly uninformed and simple, functioned relatively well in terms of allowing for prediction and tracking. It may be slow and sometimes

inaccurate, but out of all the methods tested it consistently performed the best. Again, with an unsupportive decomposition of the configuration space, this algorithm could neither be run to its full potential nor an efficient speed. With a grid space it would be far easier to calculate the most dense square, using simple modular arithmetic, and set that as the target.

A lesson learned in this area is that to efficiently make use of the employed mechanics of the game, the game engine itself must be easily accessible and designed to efficiently assist and compensate for these mechanics. While the "most dense waypoint" performed a reasonable job, having a more supportive game engine would have assisted greatly in terms of speed and cost.

Another lesson learned, touched on in section 4.4.3, is that the method used for analysing a set of data should be able to handle said data's complexity. A major issue with implementing these greedy methods was that they were unable to make adequate sense of a multimodal distribution. Using "the most dense waypoint", by only looking for the most concentrated point, it ignores the complexity of the data and focuses on the global-maximum. While this worked for demonstrating the usage of a particle filter as a method for tracking, much of the relevant data gathered goes unused. A method I thought about, but never managed to implement, is discussed in 5. Using an A* search with the number of particles seen per waypoint would be a far more efficient and comprehensive planning technique.

Chapter 5

Future Work

If I was to attempt implementing this mechanic again I would take a different approach to number of activities. Some of the more important actions I would take if I were given the opportunity to attempt this again:

- write the platform from scratch to ensure I knew the code comprehensively
- use a grid based approach for motion planning
- design the game engine around the particle filter to make the most of its power, increase both its efficiency and effectiveness
- use multithreading to process the game AI
- have a supportive layer of scripting to animate the behaviours
- develop a reasonable method of sharing information, and
- emphasise not only more powerful decision making algorithms, but also collaborative approaches.

When I started work on this project I had no real understanding of how to establish a game engine, nor how to design an interesting game. Now I feel I have both of those to a more significant degree, and if I was to attempt this project again, writing the game engine from scratch would be the best option to ensure it fully integrates and supports the particle filtering process.

If I was to attempt this project again I would do the motion planning using different techniques. Instead of skeletonization techniques I would attempt a cellular approach using a grid world. I believe that a grid space construction, not just for motion planning but also for level design, would be a much more appropriate approach. In a grid world obstacles and walls could easily be mapped out, collision detection issues would be dramatically reduced and the efficiency of running planning algorithms would also be increased. Using a grid world would also offer a much easier avenue to share data by converting the information collected from the particle filters into probabilistic cells. A grid world would constrain the design of the levels to mainly blocks, but given that the design of the game levels consisted mostly out of squares and rectangles, this would not be a problem.

A major issue from the start was the lack of threaded AI. Generally for a game like this you would have the AI's control process being executed in a separate thread to the main line. This is because you want absolute responsiveness for the player and their world, but the artificial intelligent agents may require a higher cost and more time to complete whatever underlying planning is required from them to act. In a threaded environment we can separate out the logic for each intelligent agent and allow each thread to be executed at the discretion of CPU. With this method the logic of each agent can be executed and maintained outside of the mainline. This would result in reducing the total cost of executing it as the now the mainline will not have to complete the AI's logic process for every timestep. This is a more efficient use of the CPU's resources and also provides real-time output to the player while reducing the overall lag caused by the execution of high cost decision making process. One the other hand, the AI will not have an update for every timestep. This is not a significant problem because the AI doesn't have to plan as fast as the mainline executes, because you will generally precompute a strategy, then reassess, observe and update this strategy on the next execution.

Players had trouble determining the intelligence of the AI by just playing with them. I believe an assisting layer of "barking" would be incredibly helpful in demonstrating the underlying intelligence. "Barking" is the act of a game AI shouting out or communicating its thought process so that the player can understand their oppositions intentions. "I think I hear something?" or "What was that?" are two common dialogue options used by game

AI to let the player know that the AI is aware of their presence. Barking can also be used to broadcast planned moves like "I'm checking the bedroom", and thus the player is then aware where the AI is going and should prepare accordingly.

One algorithm which I wanted to try would be an additional calculation of the "most dense waypoint" algorithm. I would have liked to have performed an A* search across the weight calculated waypoints using the number of particles at each waypoint as a heuristic. In doing so I think a more informed plan could be achieved. A path would be calculated which maximised the amount of particles seen while traversing to the most dense waypoint. This approach would provide a far more efficient search path, but would be subject to the same disadvantages as the most dense waypoint algorithm.

One thing I was not able to do was to find a reasonable way of allowing the guards to share information with each other. If I was to attempt this project again, I would take advantage of a grid based world by converting the distribution to a probabilistic grid and share the information between guards. I feel that by doing this their intelligence would rise greatly and would thoroughly assist in collaboration.

Collaboration is another aspect of the pursuit-evasion scenario that would require a lot of experimentation and research. It is very dependent on both the decision and reasoning algorithms used as well the information sharing. Multi-agent collaboration is a huge problem and far too large to take tackle with the intent on finding an optimal solution. I intended to look at how I could have developed reasonable method(s) of multi-agent collaboration, or at least, reduction of conflicts in planning. While this was a planned part of my research, time constraints meant it became an unfulfillable objective. This problem was far too large to tackle as part of my thesis as much of the initial part of my thesis was spent struggling to implement an effective particle filter.

Chapter 6

Conclusion

The aim of my thesis was to create realistic, challenging and intuitive agents to play a pursuit-evasion game. These agents were to be constructed using higher artificial intelligence techniques without relying on scripted, rule-based systems so common in the field of game AI.

The use of a particle filter, while not too efficient in the given setting, contributed to some interesting behaviours and competitive gameplay. None of the guards are ever allowed to access the internal players state, nor were given too much information about the game state. Any intelligence displayed is solely due to intuitive design and rational predictions and planning.

While this is a reasonable outcome it leaves a lot of opportunity for further experimentation. The main goal of my thesis was achieved. I believe I demonstrated that higher level artificial intelligence techniques from the field of robotics can be adapted to serve as a replacement for traditional, simple rule-based and/or scripted systems.

Bibliography

[1] C. Bererton. State estimation for game AI using particle filters. In *AAAI workshop on challenges in game AI*, 2004.

[2] David M. Bourg and Glenn Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004.

[3] Alex J. Champandard. Game AI: beauty and the beast. http://aigamedev.com/open/editorial/beauty-beast/, August 2007.

[4] Alex J. Champandard. Hierarchical logic and multi-threaded game AI. http://aigamedev.com/open/articles/hierarchical-logic-multi-threading/, August 2007.

[5] J. P Hespanha, H. J Kim, and S. Sastry. Multiple-agent probabilistic pursuit-evasion games. In *IEEE Conference on Decision and Control*, volume 3, page 2432–2437, 1999.

[6] L. Lidén. Artificial stupidity: The art of intentional mistakes. *AI Game Programming Wisdom*, 2:41–48, 2003.

[7] E. Raboin, D. Nau, U. Kuter, S. K Gupta, and P. Svec. Strategy generation in Multi-Agent Imperfect-Information pursuit games, 2010.

[8] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, 2009.

[9] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[10] R. Vidal, O. Shakernia, H. J Kim, D. H Shim, and S. Sastry. Probabilistic pursuit-evasion games: theory, implementation, and experimental evaluation. *IEEE Transactions on Robotics and Automation*, 18(5):662–669, 2002.

List of Figures