Honours Project Report

University of Cape Town
Department of Computer Science

# Environmentally Aware Game Bots

*Author:*
Gina Morris
ginamorri@gmail.com

*Supervisor:*
Patrick Marais
patrick@cs.uct.ac.za

*Co-supervisor:*
Simon Perkins
sperkins@cs.uct.ac.za

| | Category | Min | Max | Chosen |
|---|---|---|---|---|
| 1 | Software Engineering/System Analysis | 0 | 15 | |
| 2 | Theoretical Analysis | 0 | 25 | |
| 3 | Experiment Design and Execution | 0 | 20 | |
| 4 | System Development and Implementation | 0 | 15 | |
| 5 | Results, Findings and Conclusion | 10 | 20 | |
| 6 | Aim Formulation and Background Work | 10 | | 10 |
| 7 | Quality of Report Writing and Presentation | 10 | | 10 |
| 8 | Adherence to Project Proposal and Quality of Deliverables | 10 | | 10 |
| 9 | Overall General Project Evaluation | 0 | 10 | |
| **Total marks** | | **80** | | **80** |

November 6, 2009

**Abstract**

Effective use of the environment in making strategic decisions is an important aspect in the design of bots for computer games and is garnering increased attention from game designers. Previously level designers have created the illusion of environmental awareness through careful scripting and the placement of 'hints' on level maps. As games begin to use more dynamic and random maps, these carefully constructed means for instructing bots become less effective and so adaptable systems need to be established that can provide bots with similarly informative information calculated dynamically.

Using a basic spatial analysis framework developed by Simon Perkins [16] this project aims to create bots that decide on actions by analysing such spatial features as width and curvature in addition to normal contextual game information. These bots operate in teams and implement pathfinding as well as flocking. Their behaviour is presented to the user through a graphical interface that allows for insights into the bots' behaviour and hence enables refinement of the rule system. The aim is to test the utility of Perkins' framework for developing simple rule-based bots such as one finds in games.

Our experimentation found no statistically significant data to suggest that knowledge of environmental variables is advantageous to bots in this implementation. Nevertheless the inclusion of the environmental data shows promise and further research should be done in this area to further explore the possibilities.

# Acknowledgements

Thanks to Patrick for being our *super*visor.
Simon, thanks for giving us all of your code and then thanks for fixing it when we broke it. and thanks for fixing it when we broke it again.

Mike. Wesley. Time to sleep.

# Contents

# List of Figures

# Chapter 1

# Introduction

Many computer games involve a user playing against opponents that are either controlled by another user or by the computer. Computer controlled opponents are known as bots and these bots are required to perform such that a human user is sufficiently challenged by them in the game. In order for a bot to display some seemingly intelligent behaviour it needs to "know" certain things about the world that it is playing in.

Games have previously been released with a finite set of maps representing the game worlds. These maps contain *clues*, placed by the map designers, that give bots additional knowledge about the world. These *clues* allow the bots to behave very convincingly on a particular map, but this approach does not create a bot that performs convincingly on any map (i.e. the bots are not level independent).

For many games with randomly generated, or user created maps the above method is not effective as it requires expertise and is a time consuming procedure. Hence it is important to create systems that can speedily gather information about a game world without input from level designers. Some other relevant points are raised in [22] respecting the need to develop more dynamic bots.

Many types of information can be gathered from the game world and one such example is spatial information, such as is described in [16]. The outcome of this project is useful for game designers who may not want to use predefined maps or want to use less storage for bot logic. As the framework does not require a designer to explicitly specify the spatial properties on a map it could conceivably provide level independent bots.

## 1.1 Problem Outline

The project as a whole is to develop a tool that will allow a user to design and refine bots and maps that make use of environmental information from a spatial analysis framework[16]. The tool presents all of the information

necessary for the user to make informed alterations to the environment or to the bots and allows the user to test the performance of their creations.

This part of the projects seeks to answer the following two questions:

1. Does awareness of spatial features (specifically width and curvature) significantly improve the tactical behaviour of bots?
2. Is there some rule combination that is significantly more effective for improving bots' tactical behaviour than other similarly complex combinations?

Two key success factors are extracted from the above aims, these are (numbered accordingly):

1. The awareness of spatial features (specifically width and curvature) significantly improves the effectiveness of bots.
2. A specific rule set results in significantly more effective bots than other similarly complex combinations.

These key success factors will be assessed by playing bots that use different rule sets (with and without environmental awareness) against each other and recording how many times each rule set wins. The bots that win the most will be deemed more effective and statistical analysis will be done to determine if the differences between rule sets is significant.

## 1.2 Division of Work

This project was a collaborative effort between Wesley King, Michael Talbot and myself. Each part is developed and tested independently prior to final integration. Work is divided as follows:

**Engine and Renderer - Wesley King** The engine controls user interactions as well as the rate at which the simulator and renderer update (i.e. the game loop). The graphical user interface allows the user to observe the *game* and make insights into the bots' behaviour. This in turn enables refinement of the rule system. The world is presented in an intuitive way such that a user is provided with a lot of information that is suitable to the situation and easy to use and understand.

**Simulator and Bot Control - Michael Talbot** The game simulator keeps track of all elements within the game such as which team has the flag and where all of the bots are. This section also implements low-level bot actions such as the behaviours, field D* pathing and flocking. The simulator interfaces with the Bot Logic component in order to update the behaviours and targets of the bots.

**Bot Logic - Gina Morris** The main aim of this section is to determine whether the spatial awareness provided by [16] can result in more effective bot behaviour. The system implements bot planning using analysis of spatial features such as width and curvature as well as normal contextual game information. This module is responsable for calculating targets and behaviours for bots using a simple rule-based system.

## 1.3 Report Outline

This report concerns the implementation of a testbed to determine the significance of incorporating a spatial analysis framework[16] into the logic of game bots. Chapter 2 is a literary investigation into existing approaches to bot design and other environmental analysis frameworks used in games. This is followed by Chapter 3 which describes and justifies the technological design of the system. The implementation process is discussed in Chapter 4, followed by testing, and analysis and discussion of results in Chapter 5.

## 1.4 Ethical Considerations

As this project does not make use of user tests it was not necessary to get ethical permissions.

A lot of code that is used in this project was written by Simon Perkins and permission was granted to use and reproduce much of it in whole or in part. Other such references have been acknowledged in the code where it, or its algorithms, are used.

This work is my own and all sources have been referenced in the Bibliography. Informal sources such as websites are given in footnotes.

# Chapter 2

# Background and Theory

There is a shift of attention in the game development industry towards the improvement of game bots [7]. This project was proposed in order to test the utility of Simon Perkins' spatial analysis framework[16] in computer games. Specifically, to determine whether spatial information can be used for developing effective yet simple, rule-based game bots for a scenario requiring teams and strategy. The framework has previously only been tested in much simpler cases; a racing scenario and a very basic war scenario.

To examine the utility of the framework it was proposed that a tool be developed enabling a designer to incorporate spatial features into a bot's reasoning process, and then observe the performance of the bot in a customizable game world. This tool is a testbed, a platform with all the elements necessary to conduct tests on the framework.

The remainder of this chapter looks at the relationship that current game bots have with their environment as well as introducing terms and concepts neccessary to understanding the project.

## 2.1 Bots in Games

The term *bot* is often used interchangeably with *agent* which has many interpretations. Wooldridge and Jennings describe an agent as "autonomous, proactive, reactive and socially able"[25]. Hayes-Roth states that "Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions."[10]. As the term *agent* is rather loaded in the artificial intelligence (AI) community we shall only be using the former term *bot*.

The Bot Logic referred to throughout this report is the 'thinking' part of the bot, it performs reasoning to determine actions from information about the game world. The way a bot behaves is determined by the the state

it is in. Some common states that the bot could be in include defensive, attacking or sneaking states.

In this report the term bot refers predominantly to the computer controlled opponents in games of the 'shooter' genre (including tactical shooters, first person shooters, etc) such as Quake[1].

Types of games that are better to demonstrate the effects of enhanced bots are such tactical shooters, deathmatch games and capture the flag (CTF) style games. Tactical shooters require realistic combat and team support while agents in deathmatch games just aim to kill as many unfriendly units as possible. Capture the flag is normally implemented following the rules explained in [1] but could also be run with each team aiming to steal the flag from their opponent's base or aiming to possess a single central flag.

In video games the best measure of how effective a bot is is ultimately the level of enjoyment[9] or challenge experienced by the users playing the game. This is sometimes acheived by making bots behave in realistically human ways[21], this only requires that the bot "act humanly" and not that it "think humanly"[19]. As this project is to develop a tool for testing different rules, it is not necessary for users to 'play' against the bots. Bot effectiveness is rather to be determined from the results of lower-level goals. By pitting teams of bots against each other we can deem those that win more frequently and in less time more effective.

## 2.2 Bot AI in Games

Bots do not need to use complicated artificial intelligence (AI) to appear intelligent; in fact the simpler approaches seem to be preferred by many[15]. This is because they are easier to implement and do not necessarily appear less intelligent in game. Infact, the increased complexity can sometimes make characters behave worse from the human players' perspective[14]. Another factor limiting bot design to simpler techniques is that bots need to process information quickly, this is especially important in real-time games. There are two distinct programming styles used for bots, namely *dynamic* and *static* [18]. Bots can also be implemented as a hybrid of *dynamic* and *static* styles.

Static bots constantly refer back to a pre-processed representation of the world (usually using waypoints or pathnodes) that shows which areas are suitable for which tasks and helps bots select good paths. Static bots are commonly used as they require less computation and still provide a good gameplay experience. The main disadvantage of using purely static bots is that they cannot be used on all maps. An example of static bots are those discussed in [24].

[1] `http://www.idsoftware.com/games/quake/quake/`

Dynamic bots on the other hand, dynamically analyse and possibly even learn the levels as they play [4]. The advantage of using dynamic bots is that they can be used on any map. Dynamic bots are also more commonly associated with emergent behaviour because they do not stick to the same predictable patterns.

Reactive architectures are defined in [25] "to be one that does not include any kind of central symbolic world model, and does not use complex symbolic reasoning". Examples of this type of control mechanism are Brooks' subsumption architecture [3] and Rosenschein and Kaelbling's situated automata [13].

The simplest system that is suitable for the scope of this project is a basic rule-based system derived from a traditional expert system[12] in that it has a knowledge base (kb), a rule base(rb) and an inference engine. The knowledge base holds information that is gathered by the bot from the game, such as it's own health level and which enemies it can see. The rule base holds all of the rules that the bot will abide by, these rules represent possible game situations and the resulting actions that the bot must take. The bot uses an inference engine to infer from the knowledge base which rules are true, it does this by finding a rule that is made up of only true pieces of knowledge. This can be seen in Example 2.1.

If the kb contains:

```
cheese is visible
cat is visible
```

And the rb contains rules:

```
if cheese is visible and cat is notVisible then eat cheese
if cat is visible then hide
```

Then the inference engine will test each rule against the current knowledge and will come to the conclusion that the mouse must hide because the cat is visible.

**Example 2.1:** *A simple rule-based system for a mouse*

## 2.3 The Game Environment

Bots in games are all aware of the environment to some degree although it is usually not in a way that bears any likeness to how a human player perceives the game world. When talking about the *environment* we are referring to the 'physical' properties of the game world. Relying more on information about the environment can help bot developers in creating bots that are "more adaptive to new situations, harder to game, less predictable, and

more variable"[8]. This "environmental awareness" could greatly enhance the effectiveness and reusability of bots albeit at the cost of simplicity[9].

Agents are usually designed with access to perfect information because limiting them to human-like perceptions would advantage human players who are better able to "fill in the blanks" from incomplete data[2]. Perfect information means that the bots know everything about the game world and hence designers need to be selective about what information they use.

There are a few common ways to provide bots with environmental information, these methods can be classified as *level independent* (the information is worked out dynamically, as the game progresses), or *level dependent* (the information is determined beforehand and saved for use within a specific world). level independent approaches are synonymous with dynamic bots and level dependent methods with static bots. A prevalent level dependent approach is the placement of "hints" in the world that cannot be seen by human players, only by bots. These are placed by level designers to inform a bot of where to go or what to do when it sees the hint. Pathfinding is a simple example of level independent environmental information gathering[9].

Some environmental information and ways of gathering it are investigated below. This project focuses on the spatial properties, width and curvature.

**Visibility** There are two interpretations of visibility, one is 'what an agent can see' and the other is 'the degree to which an agent can be seen'.

Line of sight (LOS) is an efficient approach but does not always provide optimal behaviour[5]. Another approach is the use of visibility graphs[6] which represents intervisible locations as a graph (these are also commonly used to perform pathfinding).

Increased awareness of how visible an agent is at point in the virtual world would greatly assist their ability to sneak and hide making them more efficient. This property can be calculated using Perkins' spatial analysis framework[16] which is discussed in more detail below.

Figure 2.1 shows where on the map the red dot is visible from (green region), this is the same as the region that it can see.

**Terrain** More steeply sloped terrain often implies a slower path across the terrain. This is commonly implemented by dividing the world up in some manner (e.g. triangulating the world) and then weighting the nodes to represent steepness. This deters pathfinding algorithms from choosing a path through that terrain and so the bot would be displaying seemingly intelligent behaviour. Pottinger notes that "The simplest and most brute force approach to terrain analysis is pathfinding"[17]. Waypoints are a level dependent and frequently used way to implement pathing. They are like virtual signposts erected by level designers and they tell bots where to go.

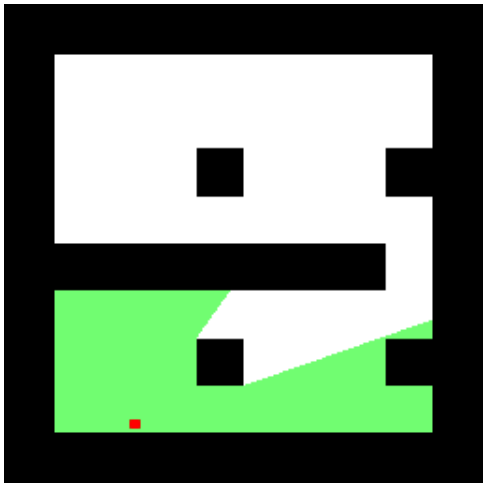


**Figure 2.1:** *Visibility at a point.*

Waypoints provide a high level of control over the bots but adding them can be "laborious, time consuming and error-prone"[9].

Pathfinding can also be used to determine connectivity of spaces. An alternative for determining higher-order connectivity is proposed in [16] and discussed in section 2.4.

**Spatial properties** An example of a useful spatial feature to consider when strategising is that of a chokepoint or bottleneck[24] (Figure 2.2). This is where a wide area narrows thereby possibly forcing a concentration of players, hence it can provide a strong point of defence.

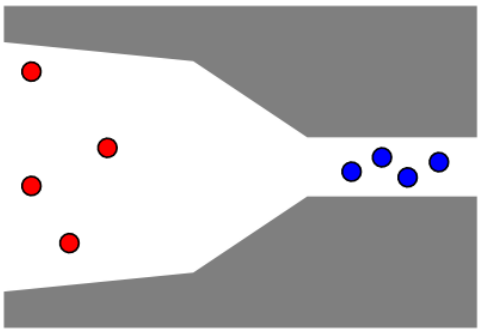


**Figure 2.2:** *Example of a bottleneck: note how the red bots can spread out while the blue bots are forced close together.*

The above example is just one case where spatial properties of a map can be useful to players (both human and bot). However, human players are capable of instantaneously evaluating spaces due to an ability to piece

together information[2] while bots rely on designer placed hints or embedded frameworks.

Areas can be explicitly labeled as having certain spatial properties (basically these are hints as discussed already) but this requires a great deal of work and understanding from the designer. Another approach discussed in [24] demonstrates the use of existing waypoints, LOS and line-of-fire information to analyse areas.

The approach of [16] differs from existing methods as it examines the intrinsic properties of a space, namely width (the openness of a space) and curvature (how sharply a space curves). These properties can potentially be used to make bots' behaviour vary to suit different types of spaces. For example, running or checking behind oneself in wide open spaces or sneaking close to walls in narrow spaces with sharp corners.

## 2.4 Spatial Analysis Framework

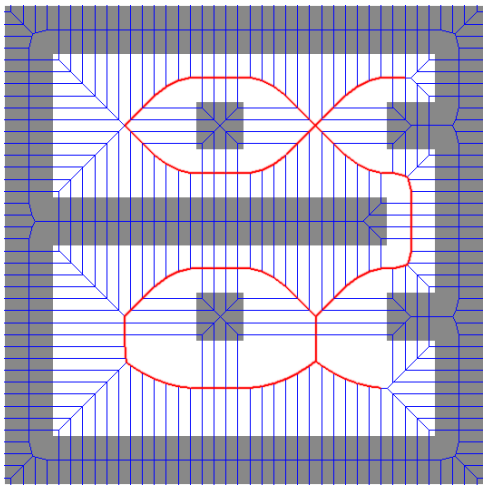


**Figure 2.3:** *The Skeleton (red) is created from a Voronoi Tesselation (blue lines) between polygons in the world*

As this project primarily investigates the utility of Perkins' framework it is important that the framework be described in more detail. For a detailed explanation of how the framework works see [16].

The framework uses Voronoi Diagrams to create a skeleton in the centre of the walkable part of the map (walkable means that there are no obstacles there, the bot can literally *walk* in that region). Figure 2.3 shows the voronoi tesselation drawn in blue and the resulting skeleton through the world (in red). Using the skeleton, one can divide the walkable area of the map into polygons (Figure 2.4) which are stored in a quad tree,allowing quick look-up

of polygons containing a point. A skeleton can be extracted from any map which makes it a level independent resource for gathering information.

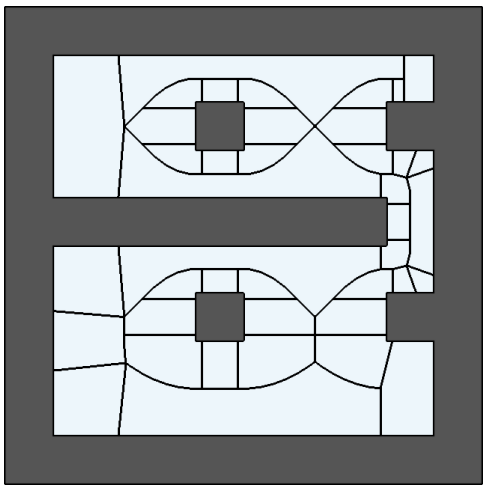


**Figure 2.4:** *Walkable world broken into Polygons*

The framework has many possible uses and a few are outlined below.

The framework can be used for pathing and determining connectivity. To do this one need only identify logical paths between points using the skeleton. It can easily be determined if a space leads to a dead end because the skeleton is folded in on itself (Figure 2.5).

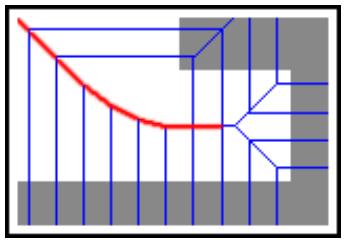


**Figure 2.5:** *Folded in section of a skeleton*

Using a skeleton framework an agent can be implemented to recognize areas of potentially higher traffic which may be ideal for "camping" (waiting in an area for an opponent to come to you, rather than actively seeking them out[2]). These areas would be points where a relatively large number of skeleton intersections could be seen as more intersections mean a higher probability that the enemy will pass within range.

The main use of the skeleton for the scope of this project is to determine the width and curvature of spaces. The width of a space is defined as the

[2] `http://en.wikipedia.org/wiki/Camping_(video_gaming)`

distance from one side of a space to the other and the curvature is defined as the dgree to which the skeleton bends. The exact details of how width and curvature are calculated are discussed in chapter 4.

## 2.5 Quadtrees

In order to use the spatial analysis framework it will be important to be able to find a point in the space. An efficient way of doing this is with region quadtrees[20].

A quadtree is a tree structure that partitions a rectangular, usually square space. Each node represents a rectangular region and subdivides this region into 4 equally sized subregions which are the children of the node. This is a region quadtree because it is being used to partition space occupied by structures (the obstacles in the world). Other quadtree's may hold buckets of objects (e.g. points) in their child nodes.

## 2.6 Summary

It is evident that designing bots to experience the world like people do is overly complex and not required for computer games. Rather, creating bots that appear to have human-like behaviour is all that is needed. Environmental awareness will only be useful to bots if they can use it without compromising their effectiveness hence allowing bots to "cheat" is acceptable.

Some environmental factors need not be made more complicated than they already are while other factors are relatively unexplored. The challenge of creating bots that do not rely on level designers placing extra area information into the map is going to be tackled here. Bots are to be able to automatically extract information from the environment and use existing game information to inform their actions. The spatial analysis framework in [16] is to be tested.

A simple rule-based system is sufficient to implement for this project.

# Chapter 3

# Design

This chapter discusses how the final solution was arrived at and offers motivations for design decisions made in terms of the 'key success factors'.

This project component is tasked with the creation of some test to determine whether the use of spatial awareness values provided by [16] results in more effective performances of game bots in a team-strategy scenario. The system makes use of spatial features as well as the normal contextual game information.

The key success factors for this section as proposed in the initial project proposal are:

- The awareness of spatial features (specifically width and curvature) significantly improves the effectiveness of bots. This is measured by the proportion of games won by environmentally aware bots versus similarly complex bots that do not make use of the spatial features provided by [16].
- A specific rule set is found to result in significantly more effective bots than other similarly complex combinations. This is measured by analysing the results of all rule sets (with and without environmental awareness), and determining if one wins a significantly large proportion of the time.

The following design goals must be ensured in order to test the validity of the 'key success factors'.

**Environmental Variables:**

- Environmental variables must be recoverable from the spatial analysis framework during the running of the program.
- Environmental variables must be incorporated into rules so that they directly influence the inference process and result in environmentally aware behaviour.

**Rules:**

- New rules must be easy to create.
- Existing rules must be easy to change.
- Rules must be straightforward and easily understandable.

**Analysis**

- Rule sets must not be difficult to analyse to determine how effective they make bots.

## 3.1 Overview

The final project is a combination of three individual components which each tackled different areas of the problem, these sections are Engine and Renderer, Bot Simulation and Control and the Bot Logic. The problem is separated this way so that each group member can focus on their section without waiting for the other sections to be complete. Separation also allows for independent testing and validation before final integration. Figure 3.1 shows an overview of the system architecture, more specifically, it highlights the Bot Logic section which is the focus of this report.

## 3.2 Interfaces

The three components communicate at only two points as can be seen in Figure 3.1, these points are:

- Between the Engine and the Simulator
- Between the Bot Control and the Bot Logic.

This means that only two interfaces are needed. This minimisation of communication between sections allows less room for error and improves performance. Sections are also more independent and able to be tested and validated independently. Independent testing is necessary to determine if the key succes factors have been met.

The Engine/Simulator interface is relatively simple with the engine passing all information relevant to that particular simulation when the simulator is created. From there the engine simply calls the simulators update function for each update step and the information required by the engine and renderer is available for reference in the simulator.

The Bot Control/Logic interface is also constantly accessed. Figure 3.2 illustrates the loop of the interface. Each bot queries the Bot Logic whenever

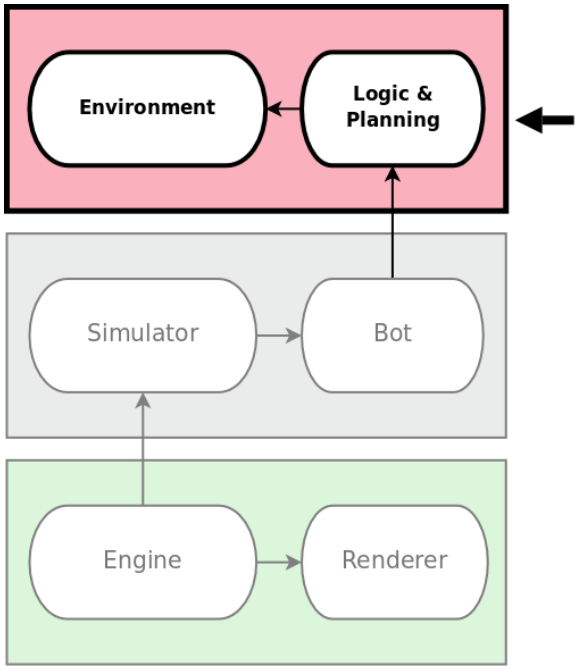



**Figure 3.1:** *System architecture overview*

a relevant game event occurs (See Triggers). The bot sends game information to the Bot logic which can determine a new target and behaviour for the bot.

## 3.3 World Data Structures

### 3.3.1 Skeleton

One of the main aims of this project is to determine whether using the spatial analysis framework developed by [16] can enhance the performance of game bots. To do this the environmental variables need to be calculated using the framework during the running of the program. This framework has previously been tested using a racing scenario and a simple war scenario. This project examines the framework using a more complicated team-strategy scenario and hence requires certain extensions and deviations from the provided framework.

The skeleton framework is explained in more detail in chapter 2 and in [16]. The extraction of width and curvature from the spatial framework is discussed in chapter 4.

[your comment: You need to say a bit mroe about waht tehse structures contain and why they are useful, and how they are used, and comment on

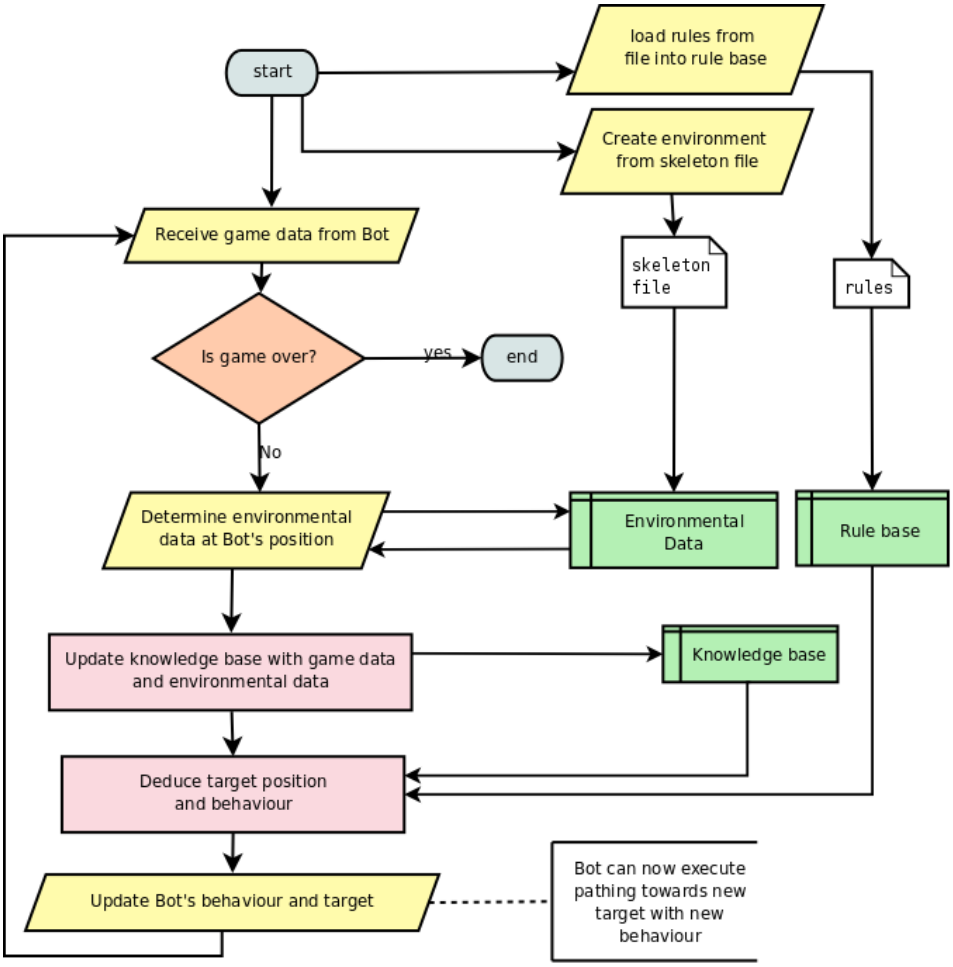


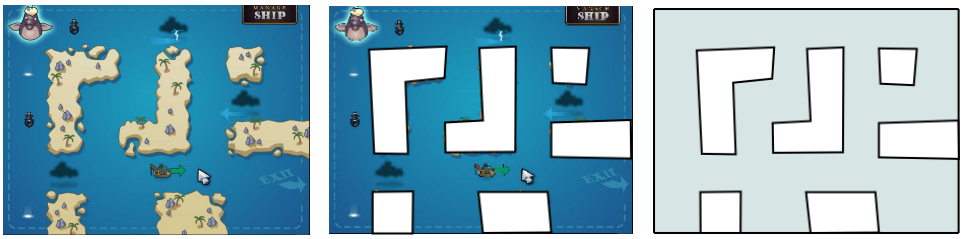

**Figure 3.2:** *Flowchart showing logic processes*

how this impacts on what you are trying to do. Ditto for Maps.
*Information on the skeleton is provided in background chapter. should I discuss more here anyway?*]

### 3.3.2 Maps

The game world is effectively two-dimensional (2D) but each map is converted to a heightmap and rendered in 2.5D to give the appearance of a three-dimensional (3D) world. This is used because it enriches the visual appeal of the graphics without being computationally expensive.

The part of the world that the bots are able to walk on is represented as 2D in memory regardless of the heightmap because using a 3D world representation is not relevant to the problem. A 2D world simplifies pathfinding operations and is also more appropriate for the spatial framework (which



**(a)** *Existing game world* **(b)** *Polygons drawn from world* **(c)** *Polygon map*

**Figure 3.3:** *Example of how one could extract a polygon based map of obstacles from an existing game world*

currently only supports a 2D world).

The spatial analysis framework uses a skeleton file which stores the skeleton and other data associated with the skeleton. This skeleton file is created from a map of polygons representing the obstacles in the environment which makes it simple. Maps can also be created for existing game worlds, this is demonstrated in Figure 3.3. (original image[1])

## 3.4 Environmental Bot Logic

The Bot Logic is required to determine new targets and behaviours for bots each time a game event (trigger) occurs. When calculating these targets and behaviours the system will take into account all triggers that have occurred and the current game context. Some rule sets will make use of extra knowledge of spatial properties (width and curvature) at the bots position. Testing rules that don't use the environment is the control as it allows one to observe the relative performance of environmentally aware bots.

The rules that are used in each rule set should make sense in the context of the game. This means that the rules should not tell bots to do counter-productive things like attack their own team members or take the flag to the enemy base. An example of a comprehensive rule set is given in Appendix A.1.

**[Patrick, I'm not quite sure what to do in this part. I'm really struggling with linking it up to the goals and success factors. I know why it links up but I can't seem to get it on paper. Also, the planning section below, you said I should "present how I have chosen to implement planning..." But I'm not really sure what you mean.]**

[1] `http://www.onetonghost.com/irondukes/`

### 3.4.1 Planning

There are many AI methods available (e.g. neural networks, genetic algorithms, production system, subsumption architectures, etc.) that could be used to determine an appropriate behaviour, but for the scope of this project it is not necessary to implement anything more than a simple rule-based system. This is because the game world is very simple and because games do not use the rigourous AI that is par for the course in academic pursuits[9]. Behaviours are determined by rules that take into account various environment variables and game triggers. The rules must be simple yet lead to intelligent behaviour in the game world. Rules follow the form:
***IF*** *relationship* ***AND*** *relationship* ***AND*** *...* ***THEN*** *[behaviour] [target]*

Many rule sets have to be tested in order to determine a more effective set, hence it is imperative that the rules can be read from a file at runtime. As this is fundamentally a testbed it makes sense that the system uses strings to represent rules. String comparisons are slower than comparing other data types but the speed is sufficient for this project. String representations allow users not only to write new rules, but also to extend the rule system quite easily.

### 3.4.2 Triggers

Triggers are game events that possibly require the bot to change its behaviour. Each is chosen because it should represent a large shift of tact. Triggers in the system are as follows:

**A team captures the flag**
: When the flag is captured the bots will want to change their approach. If the other tean has the flag the bots will want to try and steal the it away while, if the bot's team has the flag, they will need to return it to their base in order to win the game.

**The flag is dropped**
: This is important for similar reasons to above. In this case, when the flag is dropped both teams will want to grab it as this is necessary to win the game.

**Visibility information changes**
: Information concerning which bots can see which other bots changes throughout the pame, for example two bots of opposing teams come within visibility range of each other and the visibility array changes. This is important because when a bot sees it's enemy it must take action to kill or escape, otherwise the bot may itself be killed.

**A bot reaches it's target**
: When a bot reaches the target position specified by its previous call

to the Bot Logic it has accomplished a "mini goal". It then wants to reassess the situation and decide what to do next.

**A bot dies**
The death of a bot is a necessary trigger as the rest of the bots can reassess what they are doing. For instance, if a bot is shooting at an opponent and that opponent dies the bot must not continue shooting at the dead.

### 3.4.3 Behaviours

Behaviours are determined by the Bot Logic and implemented by the Bot Control component thus maintaining a clear distinction between the logical processes and implementation. The behaviours display sufficient variety so that more outcomes are possible and the solution space can be more comprehensively explored in order to acheive the second key success factor. Behaviours are made whole by the use of a suitable target. Behaviours implemented in the game are as follows:

**Attack** Pursue and attack a visible enemy.
Our game is not just a 'race to the flag' because it involves conflict. Attacking is a common offensive conflict behaviour encountered in games that implies a bot engaging with a specific opponent.

**Flee** Retreat towards a point, walking backwards.
When a player encounters an enemy and wants to get away it is not sensible to turn away from the opponent to flee as they can then just shoot you in the back. This is why fleeing bots will face the direction they are moving from in order to continue shooting any opponents that pursue. This can be interpreted as defensive conflict.

**Move** Walk towards a point.
The ability to move around the map taking no distinctive actions is encountered in most games. The behaviour that emerges from using Move is determined more by the target that the bot is moving towards. Different targets (outlined in subsection 3.4.4) can change the result of using Move from an exploring type action to a retreating action (like Flee but walking forward)

**Camp** Scan the surrounding area from a point.
Camping is often frowned upon in games, perticularly in deathmatch-type games. However, in CTF-type games it can be very useful for players to camp and cover team members when attempting to get the flag. Base camping is also a familiar practice where bots camp at a base, be it their own or the enemy's. Camping prevents a bot from being snuck up on as it is always checking in other directions.

**Sneak** Walk towards an enemy without shooting at them.
Sneaking can be used to approach an enemy without them detecting you. An example is if an enemy is guarding the flag, instead of drawing attention to itself and possibly being killed a bot can stealthily grab the flag. Sneaking is only sensible if the bot has not already been seen by the enemy.

**Defend** Protect an allied bot by keeping close and shooting enemies that come within range, walk backwards.
Defending is also a common behaviour encountered in computer games involving teams. It allows the team to rally around a member to prevent harm coming to that member.

### 3.4.4 Targets

Bots are given targets that are relevant to the behaviour they have. A target can either be a position on the map or a reference to another bot. The range of target options is chosen to provide the above behaviours with the most possible resulting actions. Possible targets are as follows:

**Random Enemy** A completely random bot from the enemy team.
Useful for Attack and Sneak behaviours

**Random Ally** A completely random bot from the current bot's team.
Can be used for Defending or just for clumping a team together.

**Flag Carrier** The bot that is carrying the flag. Useful to either Attack the flag carrier if it is an enemy or to Defend if they are in the same team.

**Base position** The position of this bot's base. Useful for base camping or retreating (Flee or Move).

**Enemy Base Position** The position of the enemy bot's base. Useful for base camping.

**Current Position** This bot's current position in the world. This can be used to ensure that a bot maintains it's position in the world. A bot can Camp at it's current position to ensure it is not being snuck up on.

**Random Position** A random position in the walkable part of the world. This is for when a bot is exploring the world using Move. In some cases bots may start off with knowledge of where the flag is but other times it may be desireable for the bot to explore the world. Random positions can also be used to make bots scatter in different directions. (Move or Flee)

**Flag Position** The position of the flag. This is used when a bot is aware of where the flag is and can then Move directly towards it.

# Chapter 4

# Implementation

This chapter discusses the project implementation. It examines the use of data structures and algorithms, adaptations to existing code, as well as describing problems and their solutions

## 4.1 Implementation Details

### 4.1.1 Platform

It was agreed that developing all of the components in the same environment would allow simpler integration. Linux was the preferred Operating System (OS) of all group members as it has many open source libraries that could be used for development. Use of Linux also greatly simplified the integration of Perkins' framework[16] as the code supplied was developed in a very similar environment.

The project was implemented and tested in Ubuntu 8.10.

Libraries used during implementation (directly and indirectly) were: libqhull-dev; CGAL 3.4; Boost 1.35; bgl-python; libsdl1.2-dev; libsdl-image1.2-dev; OpenGL; gtk+; gtkglext; libglade; libexpat

### 4.1.2 Language

C++ was the language chosen as it is stable, time efficient, allows openGL (for the rendering component) and allows efficient interfacing between components using pointers. Additionally, all group members are able to code satisfactoraly in C++.

Some python code was provided by Simon Perkins and this was used in it's original form wherever reasonable to save time.

## 4.2 World Data Structures

The data structures that make up the spatial analysis framework are encapsulated in the Environment class. The Environment class also contains other information and methods necessary for creating and interfacing with the spatial framework. This class is instantiated once by the simulator during the program's initialization. The retrieval and use of these data structures is expanded below.

### 4.2.1 Raw Map Files

The different components of the system all use maps to acheive their key success factors.

- The spatial analysis framework requires a Skeleton file which is extracted from polygon representations of the obstacles of the world (This is explained in detail in the following section)
- The pathing implementation requires a .dia file representing the walkable regions of the map as triangulated polygons.
- The renderer uses a heightmap in order to render the world in 2.5D. The darker a pixel in the image is, the higher that point is in the rendered world

Each of these maps is required to be a different format and so a process was developed that allows the creation of input files from a singular initial map file (Figure 4.1). This is automated by using the python script in Appendix B.1 but sometimes additional changes need to be made manually.

Using this process ensures that data is consistent between components.

The biggest problem encountered when regulating the maps was the discrepancies of the coodinate systems used. The renderer uses the dimensions of the heightmap image while the bot components both use the dia map coordinates. This was solved by converting all points that pass between the engine and simulator to the correct value. The bot logic component is not required to perform and point conversions as it communicates only with the simulater component which uses the same coordinates.

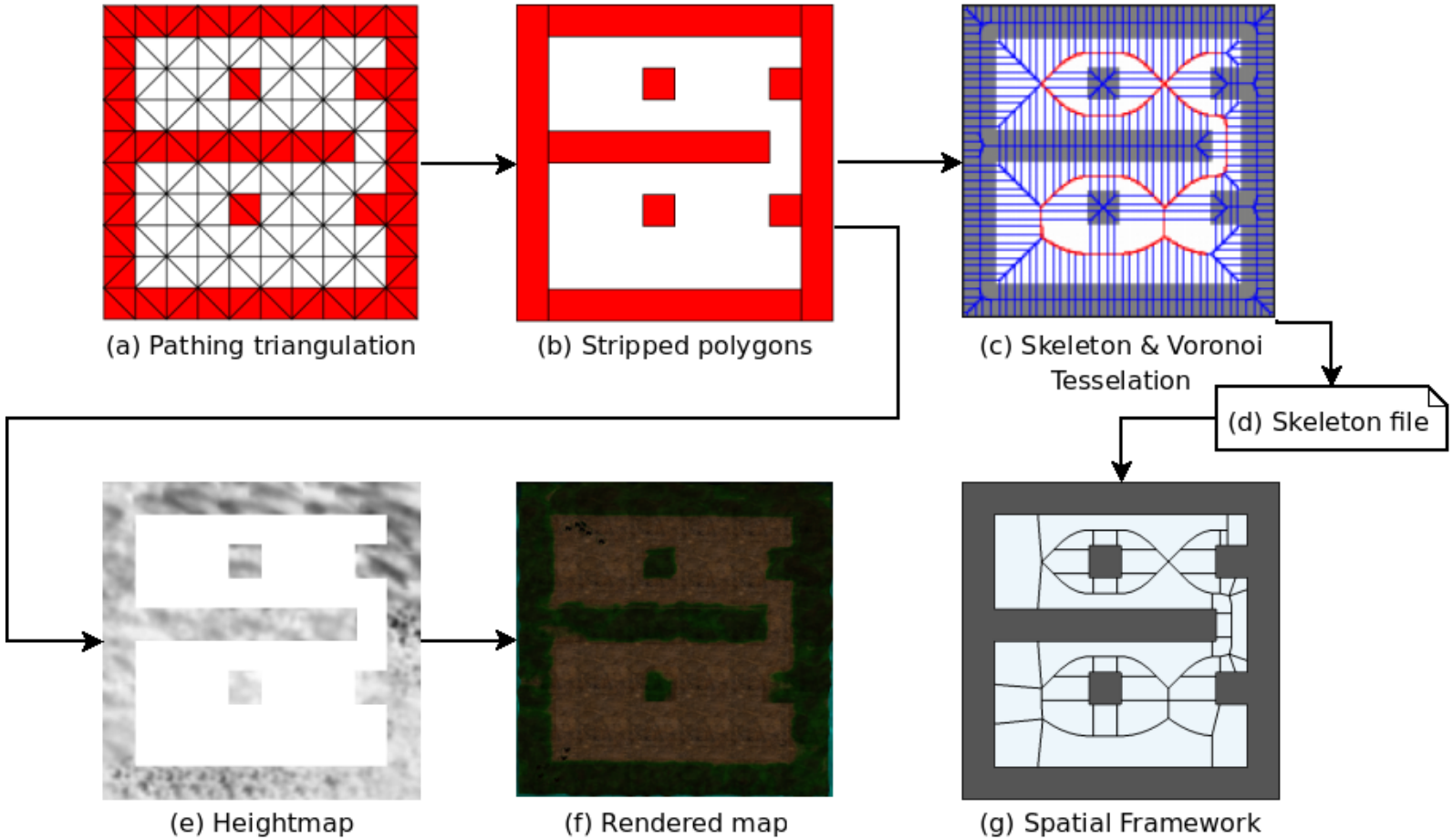



**Figure 4.1:** *The pathing triangulation map (a) is stripped by colour to only contain higher order polygons representing the obstacles in the world (b). The height map (e) is created by shading obstacles which are then rendered (f). The skeleton of the world (c) is extracted from the stripped polygon map and a skeleton file (d) is created. The skeleton file is the input necessary to build the spatial framework (g).*

### 4.2.2 Extracting the Skeleton

The code used to extract the skeleton from the stripped map was provided by Perkins. It is written in python and makes use of many of the libraries mentioned previously. This program can also produce visibility information such as that shown in figure 2.1 but this information was not used because of the limited scope of the project due to time constraints.

Initially the code would not compile on any available machine but this was eventually corrected. There were a few slight changes to the format of the output skeleton file throughout the implementation. These changes generally only required slight alterations to the code used for parsing the skeleton file, this is covered in the next section.

### 4.2.3 Creating the Spatial Analysis Framework

The framework code provided was written in python by Simon Perkins who was unwilling to code a suitable interface between the python and the C++.

The reasons for this were:

1. He is not a full resource on this project.
2. Interfacing to the python code would have created even more dependencies on python libraries.

It was decided that the required framework code be *re*written in C++.

In addition to simpler integration, rewriting the code increases familiarity with the data structures and allows customizations that are more suitable for the scenario. Environmental information is also accessed more efficiently when the data structures are more readily accessable.

C++ is somewhat more sophisticated and less intuitive or forgiving than Python and so the C++ implementation requires thought despite primarily consisting of converted code.

The skeleton data is read from the skeleton file at the start of the program and parsed into a series of data structures. These data structures are maintained in memory and are fundamentally the basis of the framework.

The following objects are used to construct the framework.

**CollatedMappingData** A vector of these objects make up the game world. CollatedMappingData objects contain a ParameterisedLine representing the skeleton and another representing the boundary; an ordered list of linear mappings and a list of OppositeMappingData objects associated with this CollatedMappingData object.

**BoundaryToSkeletonSection** BoundaryToSkeletonSection objects are linear mappings that map a section of skeleton to a section of the boundary.

**OppositeMappingData** This object associates a section of the skeleton with a section of skeleton from an opposite CollatedMappingData object.

**ParameterisedLine** Parameterised lines are list of points defining a piecewise linear manifold.

### 4.2.4 Mapping a Point to the Skeleton

The environmental information at a point needs to be accessable throughout the running of the program. In order to get the environmental data at point $P$, one needs to know which objects in the spatial analysis framework are associated with $P$. To make this possible, the linear mappings of the framework are converted to polygons. The polygons are inserted into a quadtree based on their position in the world and then the point in question is checked against the polygons.

The following objects are used to find points in the framework:

**QuadTree** Holds the root node of the quad tree.

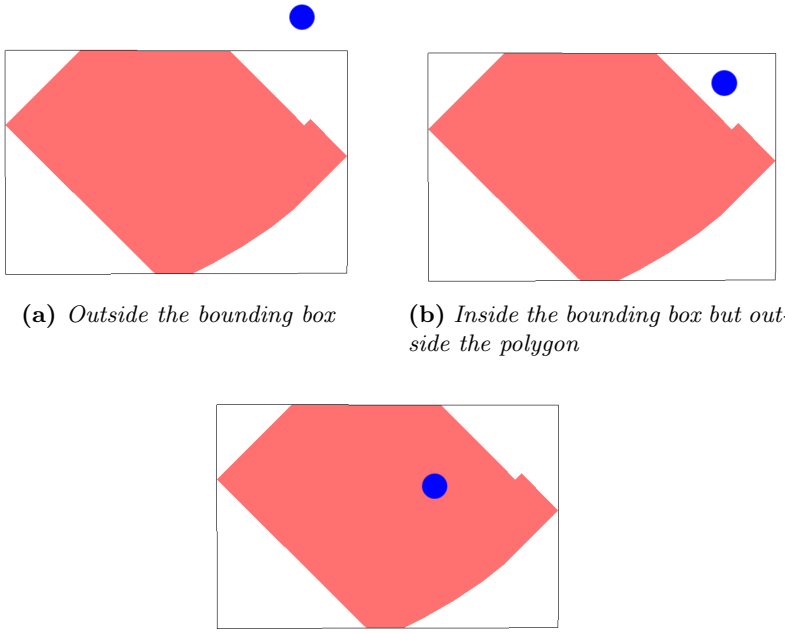
**QuadTreeNode** Contains a list of QuadTreeItem objects that fit into the space represented by this node, links to up to four children nodes that further divide the space.

**QuadTreeItem** Encapsulates a Polygon object and also maintains its bounding box.

**Polygon** A polygon chunk of the world represented by an ordered list of points that make up the boundary of the polygon. All Polygon objects have references back to the CollatedMappingData and BoundaryToSkeletonSection objects that they are associated with.

```
pick(node, point):
  item_list = []
  for items in node do
    if item.bounding_box.contains(point):
      item_list.add(item)
    end if
  end for
  for child in node do
    pick(child, point)
  end for
  return item_list
```

**Listing 4.1:** *Pseudocode of algorithm used to find the polygons that contain a point within their bounding boxes*



**(a)** *Outside the bounding box*

**(b)** *Inside the bounding box but outside the polygon*

**(c)** *Inside the polygon*

**Figure 4.2:** *Determining if a point is inside a polygon using a bounding box.*

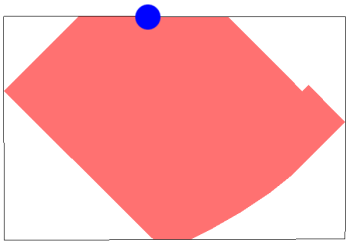


**Figure 4.3:** *A point on the boundary of a Polygon*

```
bool onEdge = false;
if ((prev_point.y == next_point.y) &&
    (test_point.y == prev_point.y) {
  onEdge = true;
}
return onEdge;
```

**Listing 4.2:** *Code to check if point is on horizontal edge*

To locate a point in the world one begins by querying the quad tree using the `pick()` method. This recursive method outlined in Listing 4.1 returns a list of QuadTreeItems where the point is within the bounding box (see figure 4.2b and figure 4.2c). The point is then checked against the Polygons in

the returned QuadTreeItems using the Polygon's `containsPoint()` method. `containsPoint()` is an implementation of the winding number point in polygon strategy[11]. The point can technically only be in one Polygon so when one is found the program stops looking.

A problem encountered when checking if a Polygon contains a point was if the point lay directly on a horizontal edge of the polygon as in figure 4.3. To combat this a check was added to `containsPoint()` to see if a point lies on any horizontal boundaries the polygon may have (Listing 4.2.

Once the correct Polygon has been found it is possible to map the point onto the appropriate skeleton using the associated linear mapping.

To debug the framework,we required a visualisation of the world structure. We wrote a method (`makeOff()`, Appendix B.2)to write a list of Polygons to an off file which can then be viewed using GeomView[1]. GeomView This not only simplified debugging, but also allowed detailed exploration of the behaviour of the spatial framework and enhanced our understanding of it.

### 4.2.5 Mapping a point to the Linear Mapping

It is also necessary to be able to map a point on the skeleton to a point on the linear mapping.

As per the original work a linear mapping is established between a section of skeleton and boundary. This mapping allows us to obtain a parameterised value on the boundary given a parameterised value on the skeleton, and vice-versa. In this sense, a linear mapping can be viewed as defining a set of lines over the space between the skeleton and boundary. Given a point within this space, we minimise the distance between the point and a line defined by a particular skeleton parameter value.

A line is obtained from a given skeleton parameter value, t, by using the the linear mapping to determine a boundary parameter value s. The point on the skeleton at t and the point on the boundary at s form the endpoints of the line.

Thus given a search point, p, and a skeleton parameter value, t, we define a function f(t,p) to obtain the distance between p and the line defined by skeleton parameter t. We minimise f(t,p) using x iterations of the golden section search to determine the line that p lies on and thereby, the skeleton parameter t defining the line. [16]

### 4.2.6 Extracting Width and Curvature

At this point it is possible to work out the width and curvature using the knowledge of the position on the skeleton.

[1] `http://www.geomview.org/`

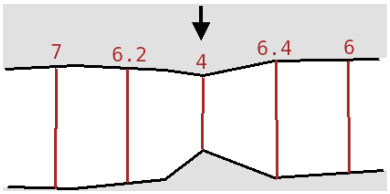



**Figure 4.4:** *Calculating width using average width*

Both width and curvature are calculated as mean values over a length of skeleton making the values more representative of the space. For example, in figure 4.4 the width at the marked position is 4. However this is not an accurate representation of the space because it as a sudden narrowing which then widens again. If one calculates the average with a few samples one gets a better representation:

$$\begin{aligned}\overline{width} &= \frac{\text{sum of widths}}{\text{number of samples}} \\ &= \frac{7 + 6.2 + 4 + 6.4 + 6}{5} \\ &= \frac{29.6}{5} \\ &= 5.92\end{aligned}$$

**Width**

The width at a point on the skeleton is calculated by finding the point on it's associated boundary and also the point on the opposite boundary. The point on the opposite boundary can be found by finding the boundary point of the OppositeMappingData object associated with that point on the skeleton. The width is the sum of the distances from each boundary point to the skeleton point.

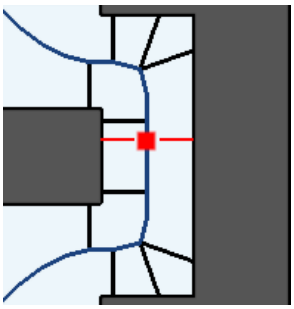


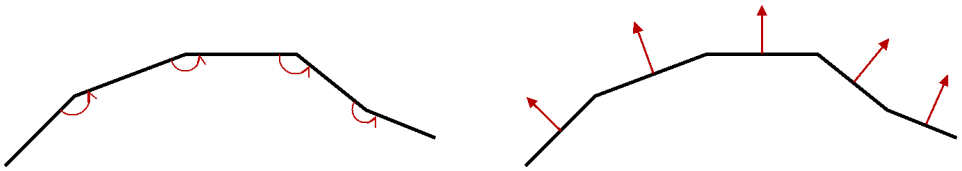
**Figure 4.5:** *Width*

Dead-ends, or sections of skeleton that end in an end-point (see Figure

2.5) do not have any associated OppositeMappingData as they are discontinuities on the skeleton. In these cases, the width is set to be twice the value of distance between the boundary and the skeleton. This is an appropriate solution since as the skeleton approaches the dead-end the width tends towards this value as the end-point of the skeleton is equidistant from all points on the boundary.

**Curvature**

Curvature is assentially the degree to which a space curves. It is possible



**(a)** *Angles* **(b)** *Normals*

**Figure 4.6:** *Curvature can either be determined using angles between adjacent line segments (a) or using the dot product of adjacent surface normals(b).*

to estimate the curvature of a space by examining how the skeleton bends, this is because the skeleton is *literally* the backbone of the space.

Initially attempts were made to calculate curvature as the average of the angles between the line segments of the skeleton (Figure 4.6a). This provided a sub-optimal measure of curvature since it only describes discrete changes in skeleton direction rather than an averaged measure of curvature over a section of skeleton.

2D surface normals were eventually implemented as an improvement over the angles method (Figure 4.6b). The curvature is calculated as the average $\overline{k}$ where $k$ is calculated as follows:

Suppose $\alpha$ is the unit normal vector of line segment AB, and $\beta$ is the unit normal vector of line segment BC.
Therefore, k at vertex B is $1.0 - \alpha \cdot \beta$.

Figure 4.7 shows how the dot product works and this shows that tighter curves would have higher $\overline{k}$ and flatter curves would have lower $\overline{k}$ as demonstrated in Figure 4.8.

## 4.3 Environmental Bot Logic

The Brain class encapsulates the objects required for the functioning of the bot logic. Each team of bots in the game has one brain associated with it because all bots in a team use the same set of rules. The brain has methods

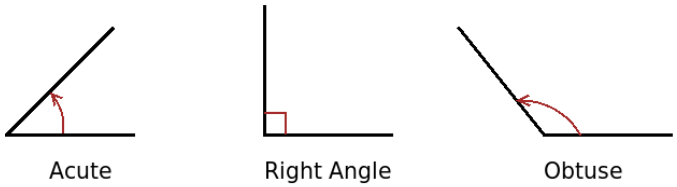



**Figure 4.7:** $|a \cdot b| = |a||b| \cos\theta$*; hence, if* $a$ *and* $b$ *are both unit vectors then:* $|a \cdot b| = \cos\theta$*. This means that acute angles will have* $|a \cdot b| > 0$*, right angles will have* $|a \cdot b| = 0$ *and obtuse angles will have* $|a \cdot b| < 0$

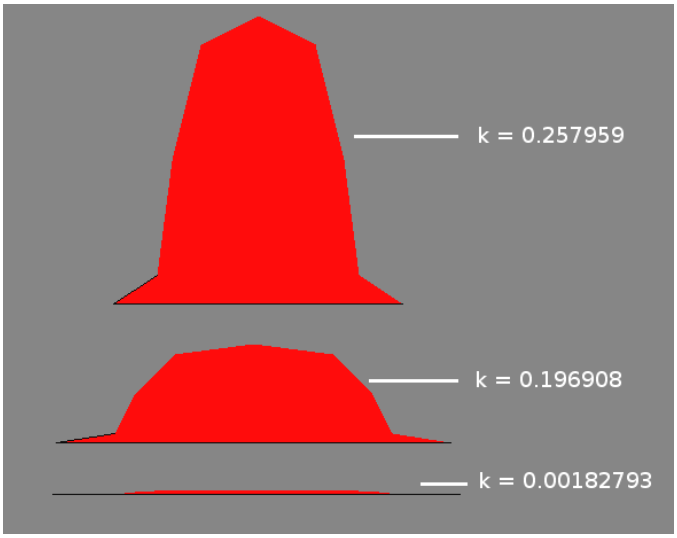



**Figure 4.8:** *Example* $\overline{k}$ *values*

for gathering information about the game (`getKnowledge()`), reading rules in from a file (`readInRules()`) and for updating the behaviour and target of a bot (`updatePlan()`).

### 4.3.1 Accessing Game Information

In order for the Bot Logic to determine behaviour modes and targets from rules it first needs to gather raw information about the environment and the game. The game information is sent from the simulator to the brain as a parameter of the `updatePlan()` method which is called for each bot whenever a trigger occurs. The game information is sent in a PlanningData struct (Listing 4.3) and contains all necessary data for the bot logic. Environmental information is gathered by querying the bot's position in the world as described in section 4.2.6.

The PlanningData struct was easy to adapt whenever new data became available in the simulator or required by the brain.

```
struct PlanningData{
    Bot* thisBot;
    const vector<Bot*>* bots; // this bot's team
    const vector<Bot*>* enemyBots;  // enemy bot team
    Bot* flagCarrier; // bot carrying flag
    int teamSize;

    const Point flagPos;
    const Point basePos;
    const Point enemyBasePos;

    // Visibility arrays representing which bots can see each
        other
    bool*** visibility;

    int teamWithFlag; // -1 if no team

    // phases change between neutral and captured
    bool gamePhaseChange;
    bool flagMove;  // has the flag moved
    bool beenShot;  // has thisBot been shot

    // a pointer to the environment so that width and curvature
         can be extracted.
    Environment* environment;
};
```

**Listing 4.3:** *PlanningData struct*

### 4.3.2 The Bot Logic Rule System

The rule system is made up of a rulebase (rb), a knowledgebase (kb) and an inference engine (ie). The use of these components to create a working rule-based system is explained below.

**Relationships and the KnowledgeBase**

Relationships are 'pieces of information' about the game world. They are represented by strings in this implementation as it permits straightforward changes. Relationships look like this: `thisBot is carryingFlag`. The three parts are classified A=`"thisBot"`, relation=`"is"` and B=`"carryingFlag"`.

Knowledge Relationships are created from the data gathered about the game world and placed inside a KnowledgeBase object (kb). The kb uses the stl map container to store Relationships because it implements the use of keys, like a dictionary. The Efficient look-up of relationships in the kb is improved by using a special RelationKey (`A, relation`) rather than just a single valued key.

In order to use the map container I had to overload the <operator but there is no definitive ordering for Relationships and so I implemented it

as just a string comparison of the As and then a string comparison of the relations.

If the kb contains:
`gouda is cheese`
`cheese is yellow`
`gouda is hard`

Then using deduction will result in an extra rule being added to the kb:
`gouda is cheese`
`cheese is yellow`
`gouda is yellow`
`gouda is hard`

**Example 4.1:** *KnowledgeBase Deduction*

The kb also is able to perform deduction in order to deduce any relationships that might not be explicitely stated, example 4.1 demonstrates this. This is not used in the project so far but it could possible be incorporated into more complex systems.

**Rules and the Rulebase**
Rules are designed in the form
`IF [Relationship] AND [Relationship] AND ... THEN [behaviour] [target].`
Rules are read in from a rule file, the Relationships are parsed and added to a set, the behaviour is read and converted to a behaviours enum and the target is used to determine the thenRule. The Rules are stored in a rulebase (rb) which is an stl set container of Rules. Using the set container was the best choice as it uses the elements themselves as the key for look-up and doesn't allow duplicate entries.

The Relationships are the conditions that need to be true in order for the entire Rule to be true.

The `THEN` part of the rule was difficult to implement because changing the behaviour and targets of a bot required access to the bot as well as access to certain environmental data. To solve this the `THEN` part was stored in the rule as a function pointer RuleFunction. A RuleFunction is a void function pointer that takes the desired behaviour and a PlanningData struct as parameters.

The RuleFunction is set by the target variable and so a finite collection of targets are permitted as described in section 3.4.2. In these functions the target being set is either a target position on the map, a target enemy bot or a target ally bot. When one of these variables is set the other two are set to NULL so that there is no confusion when the behaviours are being

interpreted by the bots. One of these functions is demonstrated in Listing 4.4.

```
void target_basePoint(Bot::behaviours behaviour,
    PlanningData* planningData)
{
  planningData->thisBot->plan = behaviour;
  planningData->thisBot->targetPos = planningData->basePos;

  // clear the other target variables
  planningData->thisBot->targetEnemy = NULL;
  planningData->thisBot->targetAlly = NULL;
}
```

**Listing 4.4:** *Rulefunction to set the target position to this bot's base*

One of the target positions is a random position. To calculate this, it is desirable to find a point that is on the walkable part of the map and secondly, not change target position too frequently. Changing the target position too randomly results in the bots walking in random directions and they take considerably longer to achieve anything in the game. It has been implemented in such a way that there is about a 30% chance of a bot receiving a new random point, other times the bot keeps it's existing target position.

### Inference

```
for (rules in rb) {
  bool allTrue = true;

  // iterate through relationships in rule. break if one is not
      true.
  for (relationships in rule) {
    if(relationship is not in kb) {
      allTrue = false;
      break;
    }
  }

  if (allTrue) {
    fire RuleFunction;
    return; // only want to fire one rule
  }

  // does nothing if no rule is true
}
```

**Listing 4.5:** *Infer() method pseudocode*

Once a KnowledgeBase has been built, the InferenceEngine is used to work out which Rule from the rulebase should be selected. This is done in the

inference method ( `infer()` ) which is outlined in listing 4.5. The appropriate RuleFunction will be fired when a true rule is found and this will change the behaviour to an appropriate one.

### 4.3.3 Writing Rulesets

When writing rulesets it is important to write the relationships correctly so that the parsers create rules that have actual outcomes and also so that the KnowledgeBase relationships will have counterparts in the rulebase. The strings need to be exact in order to equate relationships. The options for each `'A is'` are shown in Listing 4.6:

```
*A*   *relation*  *B*
width      is  [ narrow | wide | open ]
curvature is  [ tight | normal | loose ]

teamWithFlag  is  [ thisTeam | otherTeam | noTeam ]
enemyBot    is  [ visible | notVisible ]
enemyBot    is  [ shooting | notShooting ]
thisBot     is  [ carryingFlag | notCarryingFlag ]
thisBot     is  [ healthy | weak ]
flagBot     is  [ visible | notVisible ]
```

**Listing 4.6:** *Relationships variables*

In order to make sure that the variables were always consistently named I created a spreadsheet that only allows the entry of valid strings. Additionally a text editor syntax file for our rule system was created to assist in finding errors during the creation of rule files.

Rulesets can be easily debugged by running the simulation with the renderer enabled, the visualisations are very informative when trying to determine what bots are doing. A screenshot of the visualiser is presented in figure 4.9


**Figure 4.9:** *Screenshot of visualisation with walked and planned paths showing*

# Chapter 5

# Testing & Results

The Key success factors of the project specify the metric that is used for each:

- The awareness of spatial features (specifically width and curvature) significantly improves the effectiveness of bots. This is measured by the *proportion of games won by environmentally aware bots versus similarly complex bots that do not make use of the spatial features provided by [16].*
- A specific rule set is found to result in significantly more effective bots than other similarly complex combinations. This is measured by *analysing the results of all rule sets (with and without environmental awareness), and determining if one wins a significantly large proportion of the time.*

To test their validity (or invalidity) one needs to design an experiment to gather results as specified by the metric. The results of these experiments are then analysed and discussed.

## 5.1 Experiment Design

Each experiment will consist of a set of *games* between bot teams with each team using a different ruleset. Experiment variables are detailed below.

**Dependent variables**

The dependent variables in each game are

- The winning team.
- The number of gameloops before a winner is determined (This *times* the game).
- The number of bots still alive in each team at the end of a game.

**Independent variables**

The independent variables are:

- Rulesets used by each team.
- The game map.
- Flocking settings.
- Random number seed.
- Flag and base positions.

**Controls**

The game map, random number seed, flag position, base positions and flocking options all remain constant for each experiment to ensure that the results pertain to only the rulesets.

The matches will include games between two teams using the same ruleset as the control. This will prove that the ruleset is capable of finishing a game against an equally matched opponent.

**Possible rule combinations**

As there are too many combinations of rules to test all possible rulesets we use a subset of rules that make logical sense. As we do not want rules that are too similar we can choose approximately 10 rules for each experiment and make sure that they are different.

**Making observations**

The games were run in batches without a renderer so as to minimize the run time. Results were output to a csv file in the form:

*mapfile, rulefile[0], rulefile[1], winners index, gameloop count, bots alive[0], bots alive[1]*

| | | Blue Team | | |
|---|---|---|---|---|
| | | Team A | Team B | Team C |
| Red Team | Team A | Results of A vs A | Results of A vs B | Results of A vs C |
| | Team B | Results of B vs A | Results of B vs B | Results of B vs C |
| | Team C | Results of C vs A | Results of C vs B | Results of C vs C |

**Example 5.1:** *Abridged representation format of scores*

The output holds all dependant variable information for generating scores and analysing. Scores are calculated and then plotted in the form demonstrated in example 5.1.

$$\begin{aligned}\text{Score} = &+1 \text{ for each enemy bot killed,}\\ &+1 \text{ for each bot still alive on your team,}\\ &+5 \text{ for winning the game.}\end{aligned}$$

Rulesets can be divided into two high-level bins, rulesets *with* environmental relationships and rulesets *without* environmental relationships. Each ruleset can also be placed into an individual bin to allow one to compare performance.

## 5.2 Testing

Raw experimentation output is available in Appendix C. Also available is mean and standard-deviation data for each ruleset and group ( with or without environmental data). The experimentation was done using particle flocking and 'Map5' as shown in figure 5.1.

The random number is constantly seeded as 0 so that the results can be exactly reproduced. Seeding the random number also reduces the variation of a singular rulesets performance over one run. A more rigorous experiment would be to run the same inputs with different seeds so as to receive more data on each ruleset.

81 independant games were played between nine rulesets for this experiment, five using the environment and four using only the game information.

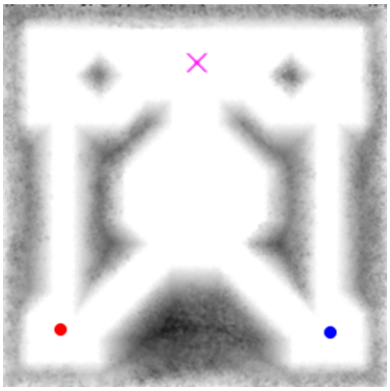


**Figure 5.1:** *Map used for experiments with red and blue bases marked and flag position marked by a pink 'x'*

## 5.3 Discussion of Results

The determination of a winner in a game appears to be quite reliant on the starting position of the teams. This is apparent from the bar graph in

| Ruleset [0] \ Ruleset [1] | rules1 | rules2 | rules3 | rules4 | rules1w | rules2w | rules3w | rules4w | rules5w | total: |
|---|---|---|---|---|---|---|---|---|---|---|
| rules1 | 0 | 0 | 0 | 14 | 2 | 0 | 0 | 3 | 3 | 22 |
| rules2 | 1 | 0 | 0 | 13 | 14 | 0 | 10 | 3 | 2 | 43 |
| rules3 | 0 | 0 | 0 | 13 | 14 | 0 | 14 | 3 | 3 | 47 |
| rules4 | 13 | 2 | 11 | 12 | 14 | 13 | 14 | 12 | 12 | 103 |
| rules1w | 0 | 0 | 2 | 2 | 14 | 0 | 13 | 2 | 1 | 34 |
| rules2w | 0 | 0 | 2 | 15 | 13 | 0 | 1 | 14 | 8 | 53 |
| rules3w | 0 | 0 | 0 | 4 | 1 | 0 | 12 | 2 | 10 | 29 |
| rules4w | 13 | 13 | 14 | 12 | 14 | 13 | 13 | 12 | 7 | 111 |
| rules5w | 13 | 8 | 8 | 9 | 14 | 8 | 13 | 14 | 1 | 88 |
| total: | 40 | 23 | 37 | 94 | 100 | 34 | 90 | 65 | 47 | **530** |

**Figure 5.2:** *Table showing Scores awarded to team[0] in each game.*

| Ruleset [0] \ Ruleset [1] | rules1 | rules2 | rules3 | rules4 | rules1w | rules2w | rules3w | rules4w | rules5w | total: |
|---|---|---|---|---|---|---|---|---|---|---|
| rules1 | 15 | 15 | 15 | 1 | 13 | 15 | 15 | 12 | 12 | 113 |
| rules2 | 14 | 15 | 15 | 2 | 1 | 15 | 5 | 12 | 13 | 92 |
| rules3 | 15 | 15 | 15 | 2 | 1 | 15 | 1 | 12 | 12 | 88 |
| rules4 | 2 | 13 | 4 | 3 | 1 | 2 | 1 | 3 | 3 | 32 |
| rules1w | 15 | 15 | 13 | 13 | 1 | 15 | 2 | 13 | 14 | 101 |
| rules2w | 15 | 15 | 13 | 0 | 2 | 15 | 14 | 1 | 7 | 82 |
| rules3w | 15 | 15 | 15 | 11 | 14 | 15 | 3 | 13 | 5 | 106 |
| rules4w | 2 | 2 | 1 | 3 | 1 | 2 | 2 | 3 | 8 | 24 |
| rules5w | 2 | 7 | 7 | 6 | 1 | 7 | 2 | 1 | 14 | 47 |
| total: | 95 | 112 | 98 | 41 | 35 | 101 | 45 | 70 | 88 | **685** |

**Figure 5.3:** *Table showing Scores awarded to team[1] in each game.*

figure 5.4 where most rules have higher scores when starting as team [1]. The average cumulative score for a team starting at [0] in this scenario is only 58.89 while it is 76.11 for teams starting at [1]. As the starting points are positioned symmetrically on a symmetric map it is reasonable to assume that the cause of the discrepency is in the order of game play. This could be solved by implementing a multithreaded version of the game.

The distribution of the scores between the environmentally aware group and the not-environmentally aware group appears to be normal and so we can use the one-way between groups ANOVA test. This test is relatively stable so it will be able to handle any outliying data points that may be encountered. The ANOVA one-way between groups analysis determines if there is a statistical significance of variance between groups.

The results of the ANOVA test are shown in Figure 5.5. If the significance (p) value is less than 0.05 then it means that there is a significant difference between the groups[23]. As can be seen from the results in figure 5.5 the two groups tested have a p value of 0.44, therefore the difference in scores between the bots is not significant. This invalidates our first key success factor of determining that the bots are improved by adding environmental data from Perkins'[16] framework.

However this is most likely as a result of other shortcomings in the implementation of the project as a whole and so further testing is encouraged.

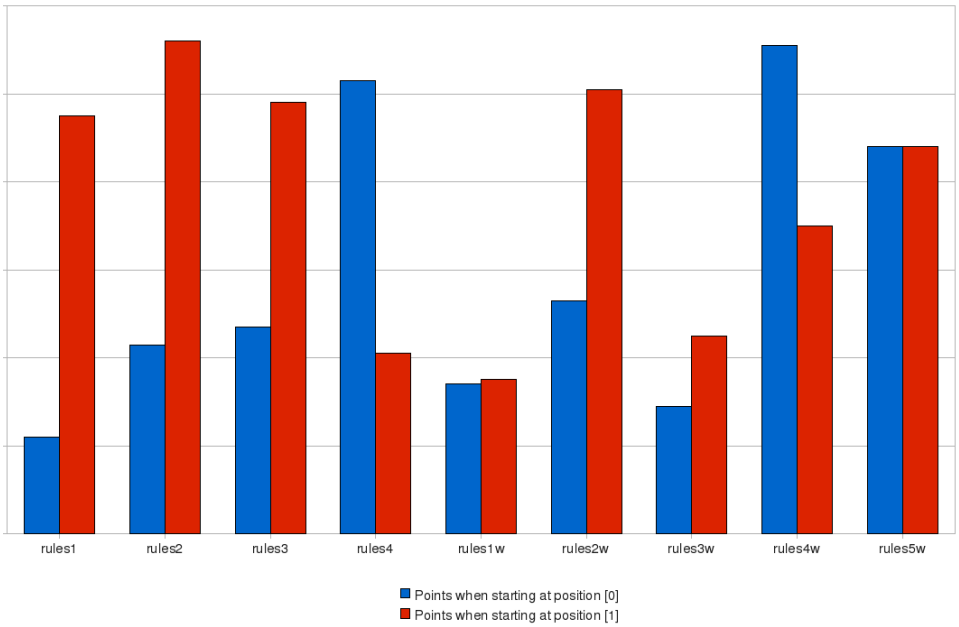



**Figure 5.4:** *This graph shows the score a ruleset acheives when starting from position [0] compared to that when starting from position [1].*

| | Sum of Squares | df | Mean Square | Fisher F-value | Significance (p) |
|---|---|---|---|---|---|
| Between Groups: | 21.22 | 1 | 21.22 | 0.6 | 0.44 |
| Within Groups: | 2786.88 | 79 | 35.28 | | |
| Total: | 2808.1 | 80 | | | |

**Figure 5.5:** *ANOVA test to show variance between the environmentally aware bots and the* not *environmentally aware bots.*

The relationship between the complexity of a ruleset and its mean score is plotted in Figure 5.6. The regression analysis shows that as rulesets become longer and more complex (i.e. more relationships per rule), the mean score increases implying that the most complex rules may lead to more effective behaviour and higher scores, however, there may be a point where too complex a file is very inefficient. It is apparent from the graph that the environmentally aware bots improve at a faster rate than the normal bots. This may be because they have more variables available to them but it may be because the environmental information creates better gameplay.

Nevertheless, this information leads us to the conclusion that with more time for testing an 'elite' ruleset can be written that makes use of the environment.

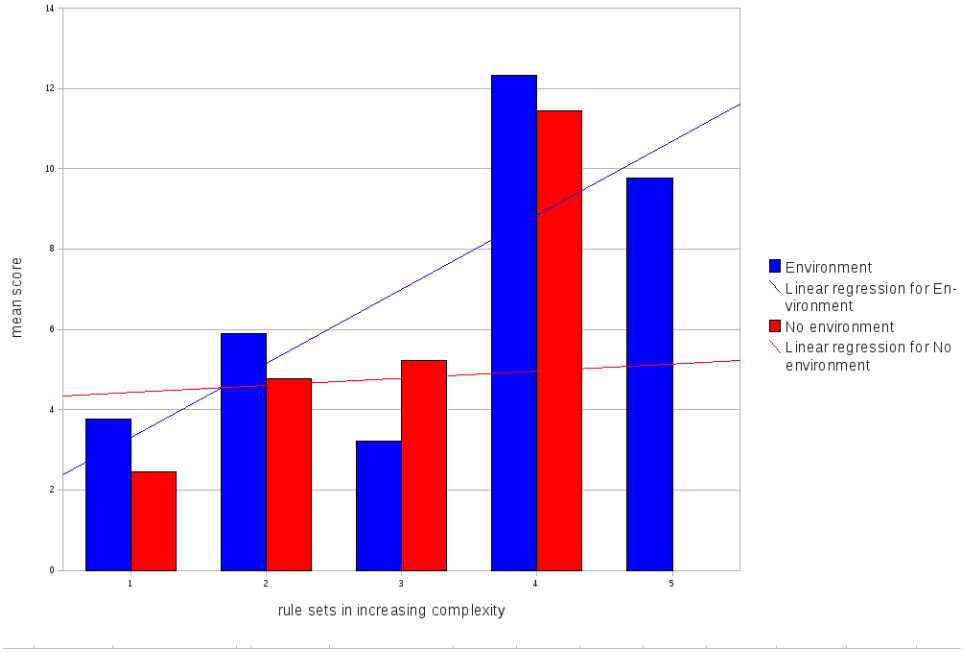



**Figure 5.6:** *Regression analysis of rule complexity.*

# Chapter 6

# Conclusion

This project was proposed in order to develop a tool that can test the utility of Perkins' spatial analysis framework in a game world scenario involving more complex bots than have been tested with his framework before. This concept is relevant to the game world today as a shift is occurring towards more dynamic worlds and an increase in user created content.

The chosen 'bots playing capture the flag' scenario is a good one as it allows bots to engage in a larger variety of behaviours in order to acheive a win (either by killing all of the enemy or 'capturing' the flag). This is compared to a deathmatch where the bots would only be aiming to kill without being killed.

It was shown that environmental data for use in the game world can easily be extracted from the skeleton despite initially seeming a near impossible task. The environmental data is suitably efficient in terms of both time and space. In fact, the program's running speed is only slowed by the expensive pathing that is implemented.

The questions posed in the introduction and repeated here for convenience were the key to maintaining focus on the right aims of the project.

1. Does awareness of spatial features (specifically width and curvature) significantly improve the tactical behaviour of bots? and,

2. Is there some rule combination that is significantly more effective for improving bots' tactical behaviour than other similarly complex combinations?

These questions have yet to be answered as the results from the testing are inconclusive, likely as a result of other shortcomings in the implementation of the project as a whole. Despite not being able to gather conclusive results, we are confident that the spatial framework can be a viable and effective tool for use in game worlds and so further testing in this direction should continue.

# Bibliography


[1] Atkin, M., Westbrook, D., and Cohen, P. Capture the Flag: Military simulation meets computer games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games* (1999), AAAI Press, pp. 1–5.

[2] Back, D. Agent-Based Soldier Behavior in Dynamic 3D Virtual Environments. Master's thesis, Naval Postgraduate School, 2002.

[3] Brooks, R. A robust layered control system for a mobile robot. *IEEE journal of robotics and automation 2*, 1 (1986), 14–23.

[4] Carmel, D., and Markovitch, S. Learning models of intelligent agents. In *Proceedings of the National Conference on Artificial Intelligence* (1996), pp. 62–67.

[5] Darken, C., Anderegg, B., and McDowell, P. Game AI in Delta3D. In *IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007* (2007), pp. 312–319.

[6] De Berg, M., Schwartskopf, O., Overmars, M., and Van Kreveld, M. *Computational geometry*. Springer-Verlag Berlin, 2000, ch. 15, pp. 307–317.

[7] Dignum, F., Westra, J., van Doesburg, W., and Harbers, M. Games and Agents: Designing Intelligent Gameplay. *International Journal* (2009).

[8] Diller, D., Ferguson, W., Leung, A., Benyo, B., and Foley, D. Behavior modeling in commercial games. In *Proceedings of the Thirteenth Conference on Behavior Representation in Modeling and Simulation* (2004).

[9] Funge, J. *Artificial intelligence for computer games: an introduction*. AK Peters, Ltd., 2004.

[10] Hayes-Roth, B. An architecture for adaptive intelligent systems. *Artificial Intelligence 72*, 1-2 (1995), 329–365.

[11] Heckbert, P. *Graphics gems IV*. Morgan Kaufmann, 1994.

[12] Jackson, P. *Introduction to expert systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1998.

[13] Kaelbling, L. An architecture for intelligent reactive systems. In *Reasoning about actions & plans: proceedings of the 1986 workshop* (1986), Morgan Kaufmann Publishers, pp. 395–410.

[14] Millington, I. *Artificial intelligence for games*. Morgan Kaufmann, 2006.

[15] Moyer, C. How Intelligent is a Game Bot, Anyway. Tech. rep., The College of New Jersey, Ewing, NJ, 2000.

[16] Perkins, S., Jacka, D., Gain, J., and Marais, P. A spatial awareness framework for enhancing game agent behaviour. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games* (2008), ACM New York, NY, USA, pp. 15–22.

[17] Pottinger, D. Terrain analysis in realtime strategy games. In *Proceedings of Computer Game Developers Conference* (2000).

[18] Rander. The Bot FAQ. URL: `http://members.cox.net/randar/botfaq.html`, 1998. Accessed on: 07 May 2009.

[19] Russell, S., Norvig, P., Canny, J., Malik, J., and Edwards, D. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, NJ, 1995.

[20] Samet, H. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR) 16*, 2 (1984), 187–260.

[21] Scott, B. The illusion of intelligence. *AI Game Programming Wisdom* (2002), 16–20.

[22] Straatman, R., van der Sterren, W., and Beij, A. Killzone's AI: dynamic procedural combat tactics. In *Proceedings of the Game Developers Conference* (2005), Citeseer, pp. 7–11.

[23] Tabachnick, B., Fidell, L., and Osterlind, S. Using multivariate statistics.

[24] van der Sterren, W. Terrain reasoning for 3D action games. *Game Programming Gems 2* (2001), 307–323.

[25] Wooldridge, M., and Jennings, N. Intelligent agents: Theory and practice. *Knowledge engineering review 10*, 2 (1995), 115–152.

# Appendices

# Appendix A

# Rules



```
IF teamWithFlag is noTeam AND enemyBot is notVisible AND enemyBot is notShooting THEN Move randomPoint
IF teamWithFlag is noTeam AND enemyBot is notVisible AND enemyBot is shooting THEN Camp currentPoint
IF teamWithFlag is noTeam AND enemyBot is visible AND enemyBot is shooting THEN Attack randomEnemy
IF teamWithFlag is noTeam AND enemyBot is visible AND enemyBot is notShooting THEN Attack randomEnemy
IF teamWithFlag is thisTeam AND thisBot is notCarryingFlag AND enemyBot is notVisible THEN Defend
    flagCarrier
IF teamWithFlag is thisTeam AND thisBot is notCarryingFlag AND enemyBot is visible THEN Attack myBase
IF teamWithFlag is thisTeam AND thisBot is carryingFlag THEN Flee myBase
IF teamWithFlag is otherTeam AND flagBot is visible THEN Attack flagCarrier
IF teamWithFlag is otherTeam AND flagBot is notVisible AND enemyBot is visible THEN randomEnemy
IF teamWithFlag is otherTeam AND enemyBot is notVisible AND enemyBot is shooting THEN Camp currentPoint
IF teamWithFlag is otherTeam AND enemyBot is notVisible AND enemyBot is notShooting THEN Camp enemyBase
```

**Listing A.1:** *<rules1>- Ruleset that doesn't use environmental variables*

# Appendix B

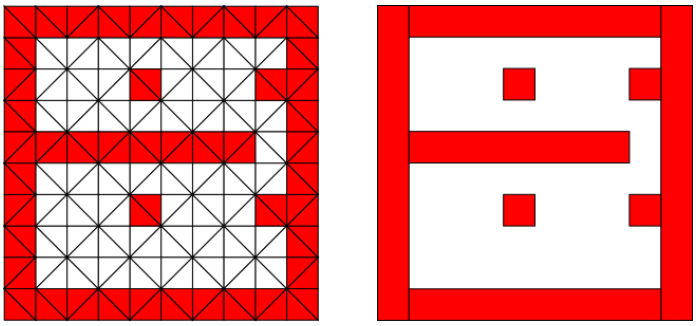
# Maps

## B.1 Dia file splitter

```
import sys
from xml.dom import minidom
import xml

class DiaXmlParser:
  def __init__(self, filename, inner_colour):
    self.filename = filename
    self.inner_colour = inner_colour

  def removePolys(self, newfilename):
    xmldoc = minidom.parse(self.filename)

    print xmldoc.toxml()

    self.recurse(xmldoc.firstChild)

    nf = open(newfilename, 'w')
    nf.write(xmldoc.toxml())

    print xmldoc.toxml()

  def isPolygonNode(self, node):
    if node.nodeType != xml.dom.Node.ELEMENT_NODE:
      return False

    if not node.tagName == 'dia:object':
      return False

    if node.hasAttribute('type') and node.getAttribute('type')
        == 'Standard - Polygon':
      return True

    return False

  def isValidNode(self, node):
```

```
    if not self.isPolygonNode(node):
      return True

    if self.getInnerColour(node) == self.inner_colour:
      return False

    return True

  def getInnerColour(self, node):
    for child_node in node.getElementsByTagName('dia:attribute'
        ):
      if not child_node.hasAttribute('name') or child_node.
          attributes['name'].value != 'inner_color':
        continue

      for child_child_node in child_node.getElementsByTagName('
          dia:color'):
        if child_child_node.hasAttribute('val'):
          return child_child_node.attributes['val'].value
    return None

  def recurse(self, node):
    if not node.hasChildNodes():
      return

    for child_node in node.childNodes:
      if self.isValidNode(child_node):
        self.recurse(child_node)
      else:
        node.removeChild(child_node)

if __name__ == "__main__":
  inner_colour = infile = outfile = None

  print len(sys.argv)

  if len(sys.argv) < 3:
    print "Usage: python dia_poly_splitter.py <infile> <outfile
        > <inner_color_val>"
    sys.exit()

  infile = sys.argv[1]
  outfile = sys.argv[2]

  if len(sys.argv) >= 4:
    inner_colour = sys.argv[3]

  print inner_colour

  xml_parser = DiaXmlParser(infile, inner_colour)
  xml_parser.removePolys(outfile)
```

**Listing B.1:** *<dia_poly_splitter.py>- S. Perkins*



**(a)** *before* **(b)** *after*

**Figure B.1:** *A map triangulation (a) is stripped of all walkable polygons leaving only the obstacles (b). Obstacles can then be made into higher-level polygons which are used as input to the skeleton framework.*

## B.2 makeOff() method

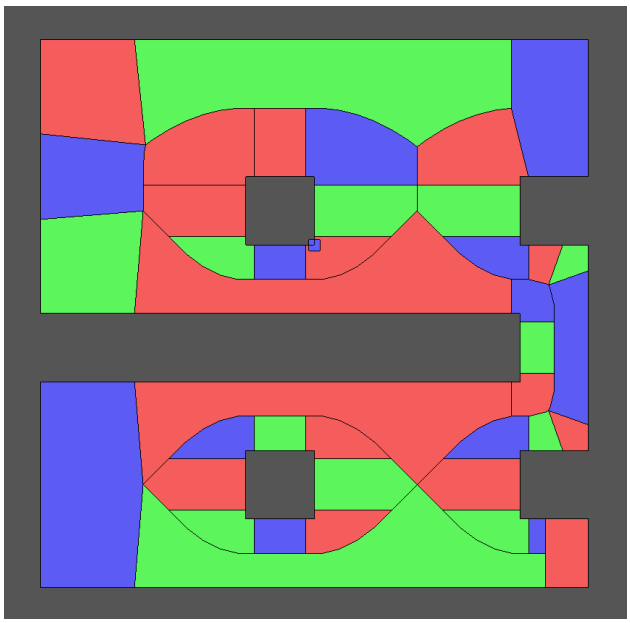


**Figure B.2:** *Example of MakeOff() output*

```
//Outputs polygons to an .off file which can then be viewed in
    GeomView See: (http://www.geomview.org/)

void Environment::makeOff(const char* filename, vector<Polygon
    *> popo) {
    ofstream off( filename, ofstream::out );
    int totpoints = 0;
    for (int i = 0; i<popo.size();i++)
  totpoints = totpoints + popo[i]->point_list.size();


    off<<"OFF"<<endl;
    off<<totpoints<<" "<<popo.size()<<" "<<popo.size()<<endl<<
        endl;

    for (int i = 0; i<popo.size();i++) {
  off<<"# points for polygon "<<i<<endl;
  for (int j = 0; j < popo[i]->point_list.size(); j++) {
      off << popo[i]->point_list[j].x <<"\t"<<popo[i]->
          point_list[j].y<<"\t0.0<<endl;
  }
    }
    int cum = 0;
    float r=1.0, g=0.0, b=0.0,a=0.75;

    //now sorting out the vertices
    for (int i = 0; i<popo.size();i++) {
  off<<popo[i]->point_list.size();
  for(int j = 0; j<popo[i]->point_list.size(); j++) {
      off <<" "<<cum;
      cum++;
  }
  off <<" "<< r <<" "<< g <<" "<< b << " " << a << endl;

  // switches the colour between red, green and blue
  int grr = i%3;
  if( grr == 0) {
      r=1.0, g=0.0, b=0.0,a=0.75;
  }
  else if( grr == 1) {
      r=0.0, g=1.0, b=0.0,a=0.75;
  }
  else if( grr == 2) {
      r=0.0, g=0.0, b=1.0,a=0.75;
  }

    }
}
```

**Listing B.2:** *makeOff() method for writing polygons to a more graphical format*

# Appendix C

# Experiment Data

| w rulesets | |
|---|---|
| point mean: | 7 |
| point stdev: | 5.85 |
| sample count: | 45 |

| plain rulesets | |
|---|---|
| point mean: | 5.97 |
| point stdev: | 6.05 |
| sample count: | 36 |

**Figure C.1:** *These are the mean scores for the environmentally aware rules (w) and the other rules.*

| | |
|---|---|
| ruleset: | ./ruleAISource/rules1 |
| point mean: | 2.44 |
| point stdev: | 4.53 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules1w |
| point mean: | 3.78 |
| point stdev: | 5.59 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules2 |
| point mean: | 4.78 |
| point stdev: | 5.85 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules2w |
| point mean: | 5.89 |
| point stdev: | 6.58 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules3 |
| point mean: | 5.22 |
| point stdev: | 6.46 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules3w |
| point mean: | 3.22 |
| point stdev: | 4.63 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules4 |
| point mean: | 11.44 |
| point stdev: | 3.68 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules4w |
| point mean: | 12.33 |
| point stdev: | 2.12 |
| sample count: | 9 |

| | |
|---|---|
| ruleset: | ./ruleAISource/rules5w |
| point mean: | 9.78 |
| point stdev: | 4.24 |
| sample count: | 9 |

**Figure C.2:** *These are the mean scores per ruleset.*

| Map file | Ruleset [0] | Ruleset [1] | winner | time | Bots alive [0] | Bots alive [1] | Score |
|---|---|---|---|---|---|---|---|
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules4w | 1 | 2462 | 2 | 4 | 3 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules2 | 1 | 983 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules2w | 1 | 1032 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules3w | 1 | 1065 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules1 | 1 | 1048 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules5w | 1 | 3958 | 2 | 4 | 3 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules1w | 1 | 2691 | 0 | 3 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules4 | 0 | 1257 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1 | ./ruleAISource/rules3 | 1 | 1063 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules5w | 1 | 3913 | 1 | 5 | 1 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules3 | 1 | 2462 | 1 | 4 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules2w | 1 | 2450 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules1 | 1 | 1264 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules1w | 0 | 1299 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules4w | 1 | 2474 | 2 | 5 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules4 | 1 | 2476 | 2 | 5 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules3w | 0 | 1289 | 3 | 0 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules1w | ./ruleAISource/rules2 | 1 | 1175 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules3 | 1 | 984 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules4 | 0 | 2454 | 4 | 1 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules3w | 0 | 2435 | 3 | 3 | 10 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules1 | 1 | 1034 | 0 | 4 | 1 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules5w | 1 | 4019 | 1 | 4 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules2w | 1 | 981 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules2 | 1 | 1006 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules1w | 0 | 2379 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2 | ./ruleAISource/rules4w | 1 | 2462 | 2 | 4 | 3 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules4w | 0 | 1198 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules1 | 1 | 1048 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules3 | 1 | 2478 | 1 | 4 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules2w | 1 | 1009 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules3w | 1 | 1160 | 0 | 4 | 1 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules2 | 1 | 983 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules4 | 0 | 1164 | 5 | 0 | 15 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules5w | 0 | 2441 | 2 | 4 | 8 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules2w | ./ruleAISource/rules1w | 0 | 2434 | 4 | 1 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules2w | 1 | 974 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules3 | 1 | 1031 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules2 | 1 | 1047 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules3w | 0 | 2432 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules1w | 0 | 2379 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules1 | 1 | 1050 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules4 | 0 | 2454 | 4 | 1 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules5w | 1 | 3962 | 2 | 4 | 3 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3 | ./ruleAISource/rules4w | 1 | 2462 | 2 | 4 | 3 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules3w | 0 | 2419 | 3 | 1 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules2 | 1 | 1047 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules5w | 0 | 2448 | 3 | 3 | 10 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules3 | 1 | 1031 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules4 | 1 | 2538 | 2 | 3 | 4 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules2w | 1 | 974 | 0 | 5 | 0 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules4w | 1 | 2477 | 2 | 5 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules1w | 1 | 1176 | 0 | 4 | 1 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules3w | ./ruleAISource/rules1 | 1 | 1064 | 0 | 5 | 0 |

| Map file | Ruleset [0] | Ruleset [1] | winner | time | Bots alive [0] | Bots alive [1] | Score |
|---|---|---|---|---|---|---|---|
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules3 | 0 | 2483 | 4 | 3 | 11 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules2w | 0 | 2437 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules1w | 0 | 2435 | 5 | 1 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules4 | 0 | 2434 | 4 | 2 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules5w | 0 | 2486 | 4 | 2 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules4w | 0 | 2434 | 4 | 2 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules1 | 0 | 2475 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules2 | 1 | 2637 | 0 | 3 | 2 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4 | ./ruleAISource/rules3w | 0 | 2437 | 5 | 1 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules5w | 1 | 3982 | 3 | 1 | 7 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules4 | 0 | 2434 | 4 | 2 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules1w | 0 | 2435 | 5 | 1 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules3 | 0 | 2682 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules2 | 0 | 2477 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules3w | 0 | 2603 | 3 | 0 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules1 | 0 | 2477 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules4w | 0 | 2434 | 4 | 2 | 12 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules4w | ./ruleAISource/rules2w | 0 | 2459 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules2 | 0 | 4250 | 1 | 3 | 8 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules4w | 0 | 4019 | 5 | 1 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules1w | 0 | 2176 | 4 | 0 | 14 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules3 | 1 | 2475 | 5 | 2 | 8 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules2w | 1 | 2474 | 5 | 2 | 8 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules1 | 0 | 4209 | 5 | 2 | 13 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules5w | 1 | 4331 | 1 | 5 | 1 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules4 | 0 | 4101 | 2 | 3 | 9 |
| ./Maps/Map5Stripped.png | ./ruleAISource/rules5w | ./ruleAISource/rules3w | 0 | 4137 | 5 | 2 | 13 |

**Figure C.3:** *Raw Results*