HONOURS PROJECT REPORT

Environmentally Aware Game Bots

MICHAEL TALBOT
mtalbot@cs.uct.ac.za
Computer Science Department
University of Cape Town

SUPERVISED BY:

PATRICK MARAIS
patrick@uct.ac.za
Computer Science Department
University of Cape Town

SIMON PERKINS
sperkins@cs.uct.ac.za
Computer Science Department
University of Cape Town

	Category	**Min**	**Max**	**Chosen**
1	Software Engineering/System Analysis	0	15	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	15
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	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	0
Total marks		**80**		**80**

November 6, 2009

Contents

List of Figures

Abstract

A simulator for team-based computer games was developed. The three main goals of the simulator were as follows: to produce an adversarial team-based game environment, to explore the benefit of using a triangulated mesh to represent the environment and using a field D* search to compute paths on this mesh, and to produce flocking behaviour in the teams of bots.

A novel approach to flocking is explored as well as the widely-used approach employing an electrostatic particle simulation.

The results obtained suggested that the novel approach to flocking is unacceptable for an adversarial environment. The electrostatic particle simulation produced more promising results. The field D* search produced better results than other commonly-used searching algorithms and produced pathing improvements above those stated in the literature.

Chapter 1

Introduction

Computer gaming is a growing industry. Gamers are interested in believable actions from AI bots. In the past, bots have been limited in their planning to waypoints and follow-the-leader. The purpose of this project is to evaluate the use of data generated by a spacial analysis tool in a game environment. The project consists of three main components:

- Engine and Renderer
- Simulator
- Environmental Planning

The focus of this report will be on the simulator component. The other components will be referenced only when affecting the design and analysis of the simulator.

There were three primary aims regarding the simulator. The first was to implement a team-based computer game. The second was to evaluate the use of a triangulated mesh and a field D* search in an adversarial game environment. The final aim was to create flocking behaviour between the bots.

Designing the team-based game evolved through iterative design. The final design consisted of two teams competing in a game mode called “capture the flag“. This provided a platform to explore and evaluate pathing and flocking techniques.

A number of pathing algorithms was explored to motivate the use of the field D* search. The particular implementation that was used made two restrictive assumptions. Much work was spent in finding suitable solutions to these restrictions. A number of subtle problems became apparent when the bots were required to move along their path, but these were overcome.

Producing flocking behaviour proved to be a non-trivial task. Two flocking techniques were explored; a novel approach and a widely-used electrostatic particle simulation approach. The adversarial setting provided a number of difficulties regarding what flocking requirements were appropriate. Each bot had its own plan of action, and these plans usually had a greater affect on the behaviour of the bots than the subtle flocking behaviour. This presented difficulties in evaluating the effectiveness of the flocking.

There were no ethical or legal issues to consider regarding this component of the project other than to acknowledge that the CGAL 3.4 open-source package [4] was used in the field D* search.

Chapter 2

Background

This chapter introduces a number of concepts that must be explained in order to understand the rest of this report. The three main concepts are flocking, pathing and some concepts from team-based computer games.

2.1 Flocking

Flocking refers to the behaviour seen when bots group together to form teams that show three specific characteristics. These characteristics are as follows:

- Collision avoidance
- Velocity matching
- Flock centering

Collision avoidance requires that, as the bots move in the group, they do not collide with each other or their surroundings. *Velocity matching* requires that all the bots move together with approximately the same velocity. Velocity refers to both speed and direction. The final requirement is *flock centering*. This requires that the group sticks together and doesn't drift apart.

These three characteristics were stated as the requirements for flocking by Reynolds [15] after he observed flocks of birds and herds of livestock. Flocking is also known as herding. He proposed that it was these three characteristics that seperate the behaviour of flocks of birds and herds of livestock from other generic groups of animals.

There are other important aspects of team structure that relate to flocking. One such aspect relates to implicit or explicit teams. Explicit teams are when each bot is a member of a team, and sticks with that team. The teams are well-defined. Implicit teams are when bots consider themselves part of a team consisting of nearby bots. Each bot looks at the state of nearby bots and computes a course of action in an attempt to produce the required behaviours. An example of an explicit flocking method was demonstrated by Cheng et al. [5], and two examples of implicit flocking were shown by Nowak et al. [14] and Reynolds [15].

Cheng et al. [5]
: This work was primarily interested in producing bots that explore a terrain in groups using flocking behaviour. To maximise a group's ability to explore, the overlap of each bot's sensors needs to be minimised. This produced groups that tended to form lines perpendicular to the direction of motion and that spread out as far as possible.

Reynolds [15]
: Reynolds was interested in simulating the motion of flocks of birds for animation. He used an adaptation of a particle simulation to produce the flocking behaviour he desired. The particle system adaptation lends itself to implicit flocking, as particles interact with each other only implicitly.

Nowak et al. [14]
: The bots in this work were unmanned air vehicles (UAVs). These UAVs needed to be able to explore a terrain and then engage an enemy if detected. The goals of these bots were situationally dependent so as to act appropriately in a given situation. Implicit teams were incorporated to accomplish this.

Another consideration is whether a team leader is employed or not. Team leaders only really make sense in explicit teams when the list of bots belonging to that team is known. For implicit teams, the team leader would not be sure who it is leading or even if it is leading any other bots at all. A benefit of employing team leaders is that it is easier to focus on the goal while maintaining team cohesion [5], since only the leader needs to make decisions and the rest of the team simply follows the leader. A problem arises in an adversarial situation if the team leader is lost or destroyed. In such a case, another team member needs to take over as the leader. This was not an issue for Cheng et al. as their work was not in an adversarial environment. Another consideration in an adversarial environment is that the team leader my not always be aware of everything that its team members are aware of. Nowak et al. [14] describe how their agents move together without a team leader until one of them sights an enemy. That agent then notifies nearby agents explicitly, which do the same to their neighbours. This should produce a quick respons without the added complexity of team leaders.

A benefit of explicit teams with leaders is that it simplifies goal seeking [5]. Implicit teams have the advantage of promoting dynamic team association, allowing teams to merge and split easily as demonstrated by Nowak et al. [14] and Reynolds [15].

The shape of the resulting team is also an important factor. The flocking produced by Cheng et al. [5] resulted in a line formation while the flocking by Reynolds [15] produced a more circular group. Cheng et al. was interested in exploration while Reynolds was interested in modelling birds. The shape needs to be picked to suit the purpose. A nice feature from Nowak et al. was that the flocking rules were situationally dependent [14].

Implicit or explicit communication can also be considered. *Explicit communication* occurs when a bot communicates by passing a message to the recipient. *Implicit communication* occurs when a bot gathers information from nearby bots. Much communication is based on implicit communication as described by Nowak et al. [14]. An example of this is when some of the nearby bots move off in a different direction. The implicit message is that the bot should move in that direction with the rest of the group. Reynolds demonstrated that all three flocking behaviour requirements can be met using only implicit communication. Nowak et al. describe how their bots use implicit communication until an enemy is located. They then send an explicit message to their neighbours who do the same to their neighbours [14]. This ensures that incorrect messages aren't being communicated implicitly when there is a threat.

The choice of these various options largely depends on the environment that the bots will be situated in. Some of the choices are imposed by the hardware available or the knowledge of the landscape. Other choices are decided by the development team to suit the purpose.

2.2 Pathing

Pathing describes the way in which a bot computes a path from point A to point B. It is an integral part of any bot in a simulated or real environment. Without pathing there is no useful mobility. Graphs are used to represent the environment that the bots operate in. The most common types of graph used to represent these environments are *2D* or *2.5D* regular grids and *triangulated meshes*. A description of each follows:

2D regular grid

A 2D regular grid is a bit matrix. It consists of evenly-spaced rows and columns. Each cell contains a boolean value describing whether that cell can be occupied by the bot or not. A 2D graph of this form produces a coordinate lattice. When referred to in this way, the vertices are referred to as lattice points.

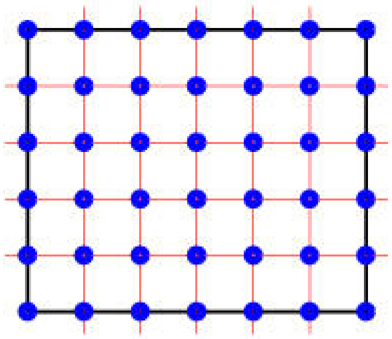

Figure 2.1: A 2D grid example
http://web.mit.edu/xcrysden_v1.4.1/www/XCRYSDEN/doc/img/genper-grid.jpg

2.5D regular grid

A 2.5D regular grid is like the 2D grid in that it is a matrix consisting of evenly-spaced rows and columns. Each cell can contain a real or integral value though. A value in a given cell often refers to the height of the environment at that point. Thus, 2.5D regular grids are often referred to as heightmaps.

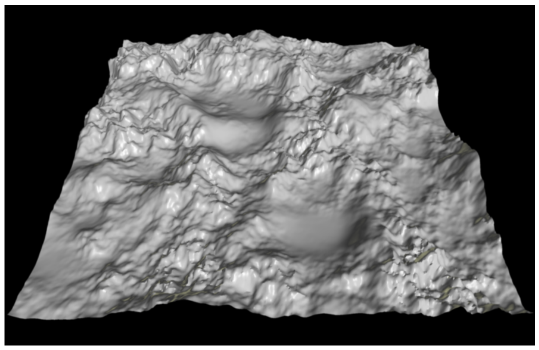

Figure 2.2: A heightmap example
http://upload.wikimedia.org/wikipedia/commons/2/2f/Heightmap_rendered.png

Triangulated mesh

A triangulated mesh is a graph where each face in the graph is a triangle. This is a far more general graph than the 2D or 2.5D grids. The triangulated mesh allows for very fine detail in some areas and more coarse detail in other areas of the same graph. This is accomplished by using small triangles to describe the finer detail and using larger triangles in areas where less detail is required. The nodes in the graph are not constrained to lie on grid lines. They can take on real-valued positions.

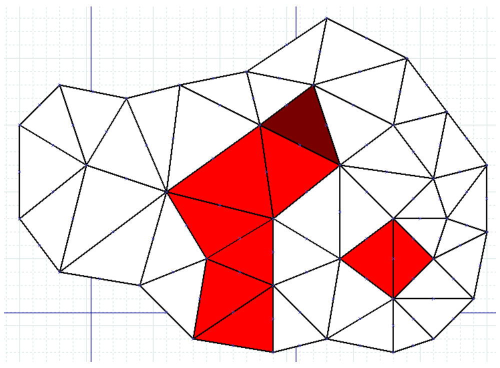

Figure 2.3: A triangulation example generated using Dia

The simplest searches that can be performed on these data-structures are the *depth-first* and *breadth-first* searches. These typically begin at the start node, and expand all possible paths outwards, until the goal node is found. With a depth-first search, the first path to find the goal node is not necessarily the shortest path, so all paths must be computed to determine the shortest path. An optimisation to the depth-first solution is to use a *branch-and-bound* method. This limits the expansion of paths that are known to be worse than the current best path to the goal node. Even with optimisations such as this, depth-first and breadth-first searches are far too slow for the common real-time requirements of bots in all but the smallest of environments.

Natural environments and many virtual environments have the added complexity that not all traversable areas are equally easy to cover. This cost of mobility must be taken into account by the pathing algorithms and data structures representing the environment. This information is not representable using a 2D or 2.5D grid. An addition to these data structures would be required to store the extra mobility cost information. With more general graph structures such as the triangulated mesh, this mobility information can be stored in the edges, or on the faces of the triangles that constitute the graph. Such a graph is called a *weighted graph*

Dijkstra's algorithm is an example of a search algorithm that takes into account non-uniform graph weightings. Dijkstra's algorithm is guaranteed to find the shortest path from the start node to the goal node. Also, the path on which the goal node is first seen is the shortest path. Pseudocode of Dijkstra's algorithm follows:

Dijkstra's Algorithm [13]
Input: A weighted connected graph $G =< V, E >$ and start vertex s
Output: The length d_v of a shortest path from s to v

```
Initialise($Q$) //an empty priority queue
for each vertex $v$ in $V$ do:
    $d_v = \infty$
    $p_v$ = null
    Insert $(Q, v, d_v)$ //initialise vertex priority in the priority queue
$d_s = 0$
Decrease $(Q, s, d_s)$ //update priority of $s$ with $d_s$
$V_T = \emptyset$
for $i = 0$ to $|V| - 1$ do
    $u^* =$ Pop Min $(Q)$ //pop the minimum priority element
    $V_T = V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) < d_u$
                $d_u = d_{u^*} + w(u^*, u)$
                $p_u = u^*$
                Decrease $(Q, u, d_u)$
```

The algorithm presented above computes the shortest path from the start node to all nodes. If one is only interested in computing a path from the start node to the end node, an early break-out condition can be introduced.

Even though Dijkstra's algorithm is a large improvement on the depth-first and breadth-first searches, it still expands nodes outwards from the start node in all directions, regardless of the direction of the goal. This knowledge of the goal node direction is wasted in such an algorithm.

Heuristic searches improve on the above-mentioned algorithms by employing information about the position of the goal node. These heuristic searches then focus the expansion of the nodes of the graph in the direction of the goal, rather than in all directions. The *A* search* is the simplest heuristic graph search algorithm and is very similar to Dijkstra's algorithm. Dijkstra's algorithm computes a cost for each vertex v based on the cost of moving from the start vertex to v. The A* also computes a cost for each vertex v based on the cost of moving from the start vertex to v, but adds the expected cost from v to the goal vertex. If the search is expanding away from the goal vertex, the nodes will be weighted with a greater cost because the expected cost to the goal node is increasing. Pseudocode for the A* algorithm follows:

A* Algorithm

Input: A weighted connected graph $G =< V, E >$, a start vertex s and a goal vertex g

Output: The length d_v of the shortest path from s to g

```
Initialise($Q$) //an empty priority queue
for each vertex $v$ in $V$ do:
    $d_v = \infty$
    $p_v =$ null
    Insert $(Q, v, d_v)$ //initialise vertex priority in the priority queue
$d_s = h(s, g)$
Decrease $(Q, s, d_s)$ //update priority of $s$ with $d_s$
$V_T = \emptyset$
for $i = 0$ to $|V| - 1$ do
    $u^* =$ Pop Min $(Q)$ //pop the minimum priority element
    $V_T = V_T \cup \{u^*\}$
        for every vertex $u$ in $V - V_T$ that is adjacent to $u^*$ do
            if $d_{u^*} + w(u^*, u) + h(u, g) < d_u$
                $d_u = d_{u^*} + w(u^*, u) + h(u, g)$
                $p_u = u^*$
                Decrease $(Q, u, d_u)$
```

An important caveat applies to heuristic searches: they are not guaranteed to produce the optimal shortest path, unless the heuristic function is *admissible*. A heuristic function is admissible if it never overestimates the cost of reaching the goal. In other words, $\forall v \in V, h(v) \leq C(v)$ where $h(v)$ is the heuristic cost from v to g and $C(n)$ is the actual cost from v to g. The main purpose of the heuristic is to speed up the search so that an adequate path can be computed fast enough to meet time requirements.

Two common choices for admissible heuristic functions are the *Manhattan distance* and the *Euclidian distance*. The Manhattan distance is defined as $D_M = \triangle x + \triangle y$. The Euclidian distance is the normal 2D distance given by $D_E = \sqrt{\triangle x^2 + \triangle y^2}$. An appropriate scaling constant is usually needed to be applied to these distances to make them admissible. The A* search often makes use of the Manhattan distance due to the low cost of calculating this distance.

Although the A* search often produces the optimal path from the start vertex to the goal vertex, the paths must always pass through the lattice points on the coordinate lattice. This means that even the optimal path produced by this algorithm is likely to be unrealistic because the bots will have to zigzag if they are moving in any direction other than parallel to an axis. Since most paths will have at least some part that requires movement in a non axis-aligned direction, the A* will always produce sub-optimal results.

The *D* search* is an improvement on the A* search in that it makes allowance for dynamic graph weightings. If the weightings of the graph change, the path can be updated without recomputing the entire path again [12]. Graph weights may change as more of the environment is explored in an unknown environment or if the actual environment is changing. The *D*-lite* search provided computational speed-ups to the D* search [12].

The problem with the A* algorithm is that the bots can only move along the lattice lines in discrete angles, usually 90°. The *field D* search* addresses this problem. It is a search algorithm that computes paths using the triangulated mesh graph. It is not suitable for the 2D and 2.5D graphs. In the searches described above, the bots are only able to move along edges, from vertex to vertex. In the field D* search, the bots move along the faces of the triangulation and are not bound to the edges. The paths are also able to change direction at any edge or vertex, rather than just at vertices as in the other search algorithms. This means that the path can take on arbitrary angle changes rather than just axis-aligned changes. Thus, bots are no longer confined to moving along the lattice lines [8]. This allows for much shorter paths as the bots need not zigzag their way to their goal, but can move directly there. The most general field D* also caters for unknown terrain as well as a moving goal.

Various additions have been made to the field D*. One such addition is the multi-resolution field D* algorithm [9] that caters to very large terrains. This method produces a very coarse grained, large scale plan from the start position to the goal across the terrain. Near the bot, a much finer grained path is produced. This stops the bot from having to generate a full fine-grained path on every step. Another update was made specifically to cater for groups of bots [11]. This variation computes a meta path that the team moves along with each bot approximately following that path with some offset from it. This helps to seperate the group. When the path goes through a narrow region, the bots bunch up, and when the region is wider, the bots spread out more. This method produced good results and seemed to help produce flocking behaviour rather than hindering it.

Pathing and flocking are both required to produce sensible bot behaviour. These two components often work against each other though. The pathing attempts to compute the shortest path between points. The flocking attempts to attract bots together so as to maintain team cohesion, which requires a deviation from the optimal path. Flocking also attempts to seperate bots if they get too close to each other, also requiring a deviation from the shortest path. Thus, the goals of pathing and flocking are naturally apposed. Joining these components in a constructive manner is non-trivial, as is suggested by the literature.

2.3 Team-Based Game

Many team-based computer games support multiple modes of play. One such common mode is called *capture the flag* (CTF). This mode typically consists of a number of competing teams each with a home base. There exists one or more flags in the map. The goal of the game is for the teams to move to the flag, pick it up and then to return the flag to their base. The first team to return the flag to their own base wins. CTF games often involve combat. Thus, teams are able to eliminate opposing team members to achieve their goal. If one team has the flag, an opposing team may eliminate the flag carrier, pick up the flag and return it to their base.

There are many particulars in the CTF game that must be decided on when creating such a game. Some of these particulars are restricted by the environment or the hardware of the bots, and some are free for the designers to decide on. These remaining free decisions are more about creating a game the designer wants, rather than overcomming technical issues. Thus, there is no correct way to decide on the particulars. These particulars are explored further in the design chapter.

2.4 Summary

Flocking is an important behaviour if believable teams are required. The three main requirements for flocking are collision avoidance, velocity matching and flock centering. A good scheme for this project seems to be to use implicit teams with implicit communication. An appropriate shape for the teams in this adversarial environment seems to be roughly circular when moving and then to change into a more linear formation or to split up when moving into confined spaces.

Pathing is essential when it comes to the mobility of bots. Simple techniques are typically very slow but give optimal results. Heuristic methods are usually employed to speed up the path-finding process, but are not guaranteed to produce optimal results unless the heuristic function is admissible. The simplest heuristic search, the A*, computes the shortest path efficiently but has a big drawback in that the paths are confined to the vertex lattice. The D* provides the ability to update the graph weights dynamically and update the path without a complete recomputation. the D*-lite provides computational speed-up to the D* search. The field D* solves the problem of the restrictive A* paths and can compute paths that are not confined to the vertex lattice. The multi-resolution field D* allows for large environments, but is not required for this exploratory project. Another improvement computed a meta-path for the team which the team members follow approximately. This method does not allow for dynamic team association though.

The capture the flag game is one of many common computer game modes. It is particularly good at promoting team play. For this project, the number of flags will be restricted to one and the number of teams will be restricted to two. A number of problems and corresponding solutions will be explored in the following chapter.

Chapter 3

Design

This chapter will outline the key success factors, discuss the design options and the decisions that were made in order to guide the project towards the success factors.

3.1 Criteria

The key success factors of this portion of the project are as follows:

- Using a triangulated mesh to describe the game environment is an uncommon method. This project is required to explore the value in using a field D* implementation on a triangulated mesh landscape in a computer game environment.
- The bots are required to exhibit flocking behaviour.
- A capture the flag game must be correctly implemented. The simulator must track the phase of the game and all the bots. The bots must be made aware of the game phase.

3.2 Interfaces

Both interfaces between this component and the other two components require information to be sent and received. The following is an outline of what information the components required from each other.

Engine and Renderer
Receive from

- A file name for the triangulation used for the map. This must be passed to the field D* implementation.

- Map settings required by the simulator. A list of these settings can be found in the appendix.
- A rasterised 2D grid describing the traversable area of the map. This is used for visibility checks.
- Two filenames; one for each rule system. The two rule systems are loaded into each team's planning system. The corresponding filename must be passed on to the correct team's planning system for loading.

Send to

- The current game phase.
- The states and attributes of all the bots in the game.
- A pointer to the flag carrier
- The position of the flag
- The dimensions of the triangulation mesh

Environmental Planning
Receive from

- A new bot state for each bot. Each bot's target ally, enemy and position must be set according to the new state the bot is in.

Send to

- The current game phase
- The cause of the request for new plans. These causes were known as *triggers* and will be described later.
- Information required to compute a new state for each bot. A list of this information can be found in the appendix.
- A table describing the vision of each bot. For each bot, a visibility test is performed for every enemy bot and the result stored in this table.

There were a number of options when considering how to neatly pass information between the components. An option is to pass the data as long argument lists to functions. This is widely accepted as being a bad way to pass many values between components. Another option is to use structs or classes to encapsulate the data and then to pass the struct or class to the component requiring the data. In the interface between the engine and the simulator, only one large data transfer occurs, which is when all the information required to initialise the simulator is sent. It did not seem valuable to have a struct type for one data transfer. Instead, the relevent variables were made public for the engine to set explicitly after the simulator had been instantiated. The requirement is that all the variables are set before the simulator is called to begin. The simulator can then be called to begin and update.

The values required by the engine were mostly limited to points and states. Structs were considered for passing information to the engine, but since there were only a small number of fields required, a struct did not seem to add value to the interface. The fields required by the engine were made public to it to simplify the interface.

The interface between the environmental planning component was somewhat different, as the planning component was a member of the simulator. To test the rule systems, an environmental planning object was required for each team. When new bot plans were required, the team planning object would be called to compute a plan for each bot. There was a relatively large number of fields required by the planning objects to produce a new bot plan. As discussed above, the option of passing each field as a parameter to a function is considered a bad option. This data transfer is required for each bot each time new plans are required. Because of the frequency, an encapsulating structure seemed appropriate. It was decided to use a struct to encapsulate this data and to pass the struct as the only parameter to the planning objects.

One parameter was a pointer to the bot requiring new plans. The planning state was made public to the planning component to set. Thus, no data was explicitly returned to the simulator.

3.3 Capture the Flag

For this project, the number of teams will be limited to two. This is to simplify the experiment by removing experimentation variables that may be introduced by having many teams. In one form of CTF, each team has a flag. The goal is to retrieve the opposing team's flag to one's base while still having control of one's own flag. This form tends to produce longer games, as an extra condition is required to win the game. For this reason, only one flag was employed. This flag starts at some neutral position in the map. The opposing teams must move to the neutral flag, pick it up and return it to their home base in order to win.

The capture the flag (CTF) game is most often an adversarial one. Thus, bots are able to eliminate opposing bots. This allows a team to re-capture the flag if an opposing bot has possession of it. In some CTF forms, when a bot is eliminated, it will be placed at its home base again with full health after a certain amount of time. This can greatly lengthen the CTF games, and if opposing teams are too evenly balanced, may cause a game to have no result and continue indefinitely. For this reason, when the bots in this game are eliminated, they will not restart. In such a situation, an alternate win condition to the game is to eliminate the opposing team. This approach was taken; when one team has been completely eliminated, the other team wins.

This set of rules constitutes a CTF game, however there are a number of problems associated with the current set of rules:

1. **Problem**
   Suppose a bot from team A obtains the flag and begins moving back to its home base. Suppose that bots from team B are a short distance behind bot A. If they are capable of moving the same speed, the flag carrier will maintain its lead and return the flag uninterrupted. This sort of game does not promote inter-team interaction.
   Potential Solution
   The only real solution to this problem is to restrict the flag carrier's movement speed in some way. For this project, it was decided to restrict the flag carrier's movement speed to 50% of the normal movement speed. This was an arbitrary decision and could have been set to some different value. This provides bots pursuing the flag carrier with more opportunity to engage in the game, and makes it possible for a team to regain the flag.
   Example
   Many first person shooter games such as Quake 3 [10] and the Unreal Tournement series [7] do not alter the flag carrier's movement speed. They commonly employ two flags for the two teams though. This means that a team can score only when both flags are present at one teams base. This makes it far harder to execute a successful flag return. In World of Warcraft [2] on the other hand, the flag carriers are limited in such a way that they are only capable of moving at 50% the speed of other players.

2. **Problem**
   Suppose bots A and B engage in combat and that it becomes clear that bot B is going to eliminate bot A. The options open to bot A are to continue fighting or to retreat. If bot A continues to fight, it will be eliminated. If it retreats, there is little benefit because it will rejoin the battle at some later stage in the same state of repair it left the previuous battle in. This makes for uninteresting game play as there is no real option once a bot is engaged in combat.
   Potential Solution
   A solution is provide a way for bots to regain health. Two common ways of doing this are to provide health packs or repair kits in variuos places in the map, or to allow the bots to passively regenerate after being out of combat for a certain amount of time. An advantage of using health packs is that the bots have a definite objective when retreating from combat. A problem though is that

the positioning of the health packs removes an element of decision that is available to bots if they passively regenerate. The benefit of passive regeneration is that the bots need to compute a position that looks beneficial to hide in. As the goal of the project as a whole is to test the usefulness of the environmental data analysis, the passive regeneration model seems to provide more scope to do this. This is decision that was taken.
Example
Many games like the Quake series [10] and the Unreal Tournement series [7] employ the use of health packs. It is a good system that has been used widely. As described above, this system reduces the scope for bots to make decisions relating to the environment, which is important considering the aims of the project. Games such as the more recent Call of Duty series [1] releases and World of Warcraft [2] use the passive regeneration system. This provides players with options when they are near death, rather than being targets at nearby health pack positions.

3. **Problem**
Suppose that bot A is the flag carrier. An opposing bot B has engaged bot A in combat and bot A is low on health. Since bot A is moving at 50% movement speed, there is no chance for escape. Bot A decides to drop the flag and retreat at full speed. Bot B then obtains the flag and the situation is reversed. Bot B then drops the flag and retreats. This can lead to unprogressive play.
Potential Solution
An option to fix this problem would be to allow the flag carrier to move at normal speed, but the above discussion rules that out. An alternative would be to let the flag carrier receive less than the normal amount of damage from opposing bots. This would act as an incentive to keep the flag. Once bot A is on low health though, even reduced damage may still be enough to cause it to decide to drop the flag, creating the same unprogressive play. Another alternative is to forbid the flag carriers from dropping the flag. The advantage is that the game necessarily progresses. The disadvantage is that it does inhibit the flag carrier's ability to make decisions. This is less important than the completion of games however. For this reason it was decided that the flag carriers are not permitted to drop the flag
Example
In World of Warcraft [2], flag carriers are permitted to drop the flag. This was done for a number of reasons. One such reason is that a particular player may be more suited to carrying the flag than another type of player. This is not really applicable for this application though as each bot on a team has identicle abilities. Most of the popular first person shooters such as the Quake series

and the Unreal Tournement series do not permit the flag carriers to drop the flag.

3.4 Behaviours

A requirement of the simulator is to produce a capture the flag game. The bots need to be able to perform basic actions in order to compete in the game. Higher level behaviours can then be created and performed as combinations of the basic actions. The planning component can then compute high-level plans in terms of these behaviours. The behaviours that the simulator is required to provide are the typical actions that one would find players performing in a CTF game.

An important decision is how to place actions in the appropriate abstraction level. The simplest of actions is the ability to move, shoot and turn. These actions are in some sense atomic in that they cannot be decomposed into simpler actions. It therefore seems appropriate to use these three actions as the fundamental building blocks of all higher-level action.

Some of the higher-level behaviours that one might observe in a CTF game would include the following:

- Attack
- Camp
- Defend the flag carrier
- Flee
- Sneak up on a target
- Explore

Many of these seem good, but two changes were made.

While exploring is a valid behaviour, it is only sensible in a situation where the map is not known to the teams at the start. This option was considered, however there is a disadvantage. The disadvantage is that it increases the time taken before inter-team interaction begins and provides no real benefit. Evaluating exploration techniques is not a goal of this project, unlike evaluating inter-team interactions. For this reason, the full map is known to the bots as well as the flag position. There is thus no need for exploration.

A behaviour that is required by the planning component is the ability to move a bot to a particular position without necessarily attacking or defending someone at that position. The above-mentioned behaviours don't allow for this. It was decided to provide a move behaviour to the planning component to provide the control required.

The game implemented by the simulator allows the bots to perform 6 explicit behaviours. These 6 behaviours are described below:

Attack
The Attack behaviour causes the bot to move towards the target enemy's position and to face in the direction of motion. Once the enemy is within shooting range, the bot will shoot at the enemy.

Flee
The Flee behaviour causes the bot to move towards a target position. The bot faces in a direction opposite to the direction of motion though, under the assumption that it is fleeing away from a pursuing enemy. If a target enemy is specified and it is in range, the bot will shoot at the enemy while fleeing.

Move
This behaviour causes the bot to move towards the target position, facing in the direction of motion.

Camp
The bot will move towards the target camping position until it is within a camping threshold distance of the target. If an enemy is specified and within shooting range, the camping bot will shoot at the enemy. If an enemy is not specified and the bot is at its camping position, the bot will turn on the spot so that nearby enemies may be seen.

Sneak
Sneak causes the bot to move towards the target enemy's position, facing the direction of motion, but does not attack the enemy. This allows the sneaking bot to inflict more damage on the first combat step, increasing its chance of winning.

Defend
The Defend behaviour causes the bot to move towards the target ally's position, facing away from the target ally, under the assumption that attackers are not between the defender and the defended bot. If a target enemy is specified and the enemy bot is in range, the bot shoots the enemy and faces towards the it.

The planning component uses these behaviours to carry out the higher-level plans.

To promote interesting results in the game, the shoot function required a random factor. If there was no random factor, then the bot to shoot first would always be the victor, which is not particularly interesting. A first attempt at a damage function was

$$d = random(0, 1) \times damage$$

In most shooting situations in real life and games, a target that is closer is easier to hit, and thus easier to damage. An improved damage equation follows:

$$d = (\frac{maxDist - dist}{maxDist})(random(0,1) \times damage)$$

A minimum damage amount seems to be beneficial to ensure that battles can only last a certain amount of time and not go on indefinitely. A scaling factor on the distance was also decided on. The final equation for damage is as follows:

$$d = (\frac{maxDist - \frac{1}{2}dist}{maxDist})((max - min) \times random(0,1) + min)$$

where max and min refer to the maximum and minimum damage amounts and $dist$ refers to the distance between the bot and its target. When $dist = 0$, $d \in (min, max)$. When $dist = maxDist$, $d \in (\frac{1}{2}min, \frac{1}{2}max)$

3.5 Pathing

The heuristic A* search is an improvement on Dijkstra's algorithm because it focusses the nove expansion in the direction of the goal node. The problem with the A* search is that the path is still confined to the lattice lines. This is unsuitable in a computer game environment, because it limits bots to zigzagging through the environment, when a straight line would be shorter and more believable.

A known solution to this problem is to use the field D* search algorithm. This requires a triangulated mesh representing the environment. This is an uncommon way for game environments to be represented. A goal of this project is to explore the value in using this environment representation and searching algorithm.

One of the resources that this project incorporates is a field D* implementation written by S. Perkins. This implementation restricted the map file format to be *dia* files. Dia is an open-source diagram tool [6] that produces dia files. Dia also allows the faces of triangles to be weighted.

This implementation of the field D* search was not the most general implementation and has two restrictions in it. These restrictions were acceptable for the requirements of the original implementation. One restriction is that this implementation requires the start position and goal position of the search to be on a vertex of the triangulation. The intermediate path is not restricted in this way, only the start and end points. The other restriction is that this implementation does not dynamically update the path when the goal node changes. The path must be recalculated.

The integration of this code into the main project was non-trivial and took a substantial amount of time. The implications of the two

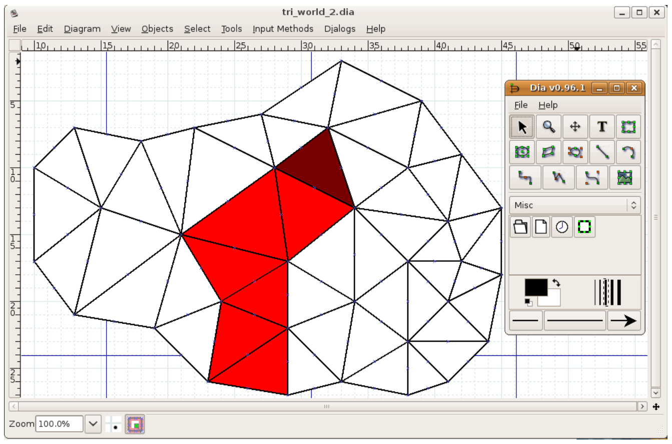
tri_world_2.dia
File Edit Diagram View Objects Select Tools Input Methods Dialogs Help
Dia v0.96.1
File Help
Misc
Zoom 100.0%

Figure 3.1: Dia v0.96.1

restrictions on this particular implementation were not clear until late in the project.

The first restriction to overcome was to be able to compute paths where the start position and goal position were not on vertices. A transformation needed to be made from the start and goal points of the bot to vertices in the triangulation. An option was to increase the resolution of the triangulation, making the maximum transformation required less. This would slow down the pathing algorithm though, because it is searching through a graph with more nodes and edges. A potential solution to this problem could have been to update the pathing less frequently. A problem associated with this method would be that on the update steps where pathing is updated, the update would be noticably slower. Staggering the pathing updates is a potential solution to that problem, but then some bots are at an advantage if they happen to have their pathing updated shortly after an important event. This line of thinking seems to have complicated solutions that don't produce very good results.

The only appropriate solution seemed to be to snap the start and goal positions to the nearest vertices in the triangulation. The movement speed of the bots was relatively small compared to the size of the triangles in the maps that were tested on. If the computed path repeatedly snapped the start position back to the nearest vertex, the bot would never move. Thus, after snapping the start position to the nearest vertex and computing the path, the first point in the path needed to be set back to the bot's current position. This allowed the bot to move off the vertex closest to its position.

A result of this solution was that, if bots were moving around an impassable area, they would move towards the corner, but when they

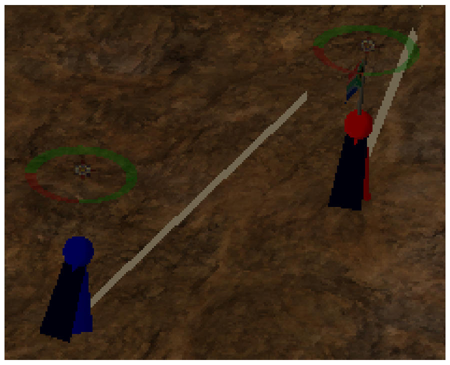

Figure 3.2: The result of snapping

got close, they would simply move over the corner of the impassable area.

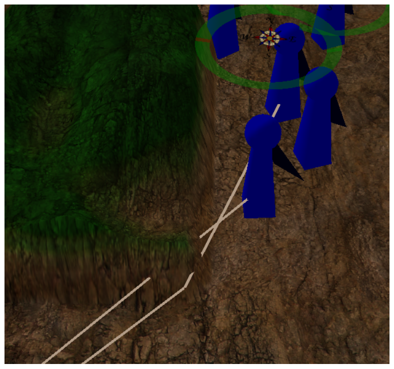

Figure 3.3: The closer bot ignores the corner

This was because after the path had been computed, the first position in the was being replaced by the bot's current position. When the current position snapped to the corner vertex, the path would return a path where the first path position was the corner vertex. This would be replaced by the bot's current position. Thus, the point the bot was moving towards was already around the corner, so the bot would simply move over the unpassable area.

A solution to this problem was to determine whether the vertex that the start position was snapping to was in front or behind the bot's current position. If the vertex was behind the bot, the first point could be safely replaced by the bot's current position. If the vertex was in front of the bot, the implication is that the bot still needs to move to that point. Thus, the bot's current position must be inserted at the front of the path.

This provided an acceptable solution at the start of the path. Similar problems were evident at the end of the path however. At this point, the path ended at the vertex that the goal position snapped to. When the minimum distance required for the bot to obtain the flag was set too low, the bots were unable to obtain the flag. This is because the distance between the nearest vertex and the goal position was larger than the threshold value.

The first intended solution was to check if the path length was 1. This indicated that the start and end positions were snapping to the same vertex. In such a case, the path updated to just contain the start position and end position. This however lead to bots moving across unpassable corners.

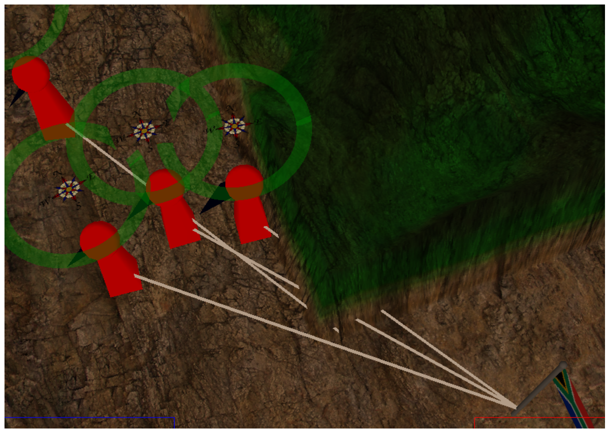

Figure 3.4: Bots close to the target ignoring corners

The reason for this is that the start and end positions could snap to the same vertex even if the goal position was round a corner. A property of triangles that is widely exploited is *convexity*. A convex polygon is a polygon such that for every two points inside the polygon, the line between the points is also contained in the polygon. The convexity of triangles can be exploited in this scenario. If the start position and end position are on the same face, then it is acceptable to change the path to consist only of the start position and end position. This is because any point between the start and end position lies inside the same triangle, and thus has the same mobility properties.

The problem still stands if the start position and goal position do not lie in the same face though. In such a situation, it is not immediately clear whether it is safe to change the path to just consist of the start and goal nodes. An option that is always safe is to insert the current bot position before the snapped vertex, and then add the goal node after the snapped vertex, noting that the path returned by the field D* implementation consists only of the snapped vertex. This produces the expected behaviour when moving around corners.

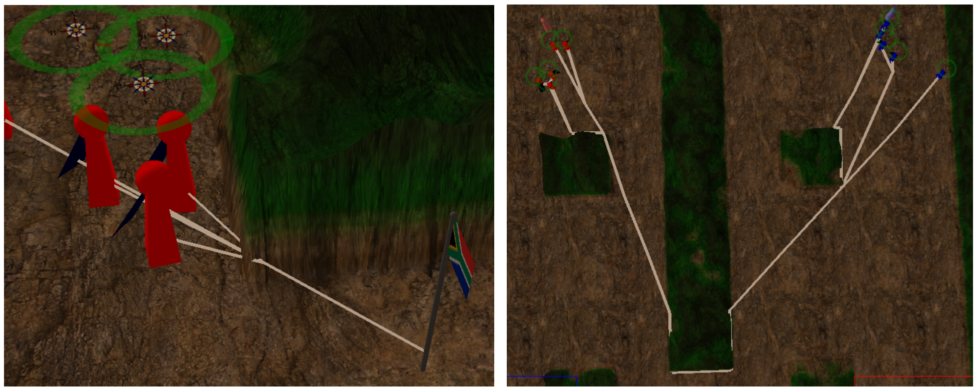
Figure 3.5: [Left] Expected behaviour close the the flag, [Right] Final pathing implementation

The problem with this solution is that, even if it is safe to change the path to consist only of the start and goal positions, the path will still require the bot to move to the snapped vertex first. An improvement to this method would be to cast a ray from the start position to the goal position to determine if it is safe to change the path to consist only of the start and goal positions. This improvement was not implemented due to time constraints.

3.6 Flocking

The three requirements of flocking are collision avoidance, velocity matching and flock centering. In an adversarial environment such as a CTF game, not all of these requirements are appropriate. An example is a group of bots defending the flag carrier. Collision avoidance, flock centering and speed matching are appropriate, but the requirement for all the bots to have an approximately uniform direction is inappropriate. The reason is because if all the defenders are facing one direction, it provides a wide angle for enemies to approach undetected. So, part of the flocking requirements are appropriate, but not all of them.

For which behaviours does flocking apply then? It seems that for sneak, camp and flee, flocking is not appropriate at all. Sneaking is typically performed by one bot, or a small number of bots, fleeing should not be a group action, and camping requires no movement. For attacking and defending, flocking applies only to the moving portion of the behaviour. In other words, collision avoidance, speed matching and flock centering apply to attack and defend, but not direction mathing.

For the Move behaviour, the full flocking requirements are appropriate, provided the group is moving towards goals in a similar region.

Since the bots all have the same movement speed, speed matching is implicit. In the case of the flag carrier moving at 50% movement speed, the defenders follow the flag carrier's position, so speed matching is retained.

If the bots are all moving to the same target position, flock centering occurs automatically, as the group converges over time as they approach the target. If the bots have different targets, then the flock may split to form a number of smaller flocks, in which these properties are also evident.

The problem then in meeting the flocking requirements is to ensure a seperation between bots, and that bots avoid obstacles in the landscape. This is a non-trivial problem, as is suggested by the literature. The most comprehensive discussion in the research that was focussed on was limited to bots avoiding circles and regular polygons. This is unsatisfactory in a computer game environment.

The first attempt at producing the seperation characteristic was a novel idea. Each bot considers its nearest neighbour. If the nearest neighbour is further away than some flocking threshold amount, do nothing. Otherwise, if bot A is in front of bot B, then bot B determines the angle θ that bot A makes with the line perpendicular to the direction of bot B. Bot B then turns away from A by the angle θ. If bot A is next to bot B, bot B will hardly change course because bot A is not obstructing the path. The more that bot A is in front of bot B, the greater θ becomes, and the more bot B attempts to avoide bot A.

Two emergent behaviours became apparent. The first is that the bots were able to form a straight line in the direction of the target, and would not move out of that configuration. This is because consecutive bots were slightly out of range of the flocking threshold, and so no flocking was being performed. The reason why this configuration was achieved was because the bot in front does not deflect, only the bot behind. This means that the bot in front can move the full distance in that update step in the direction of its target. The bot behind is deflected though, and can only move $\sin(\theta)$ times the full distance in the direction of its target. This means that the bots that start slightly behind get further and further behind until they are beyond the flocking distance threshold and flocking no longer takes place. These bots can then move at full speed again towards their target, which results in the line formation.

The other emergent behaviour is that, when many bots were defending a flag carrier, they would group tightly together, side by side. This configuration was achieved because, if bot A makes a very small angle θ with bot B, bot B will not be deflected by much. This is regardless of the distance between bot A and bot B. Thus, bot A can move very close to bot B, as long as θ remains small. In other words, bot A can

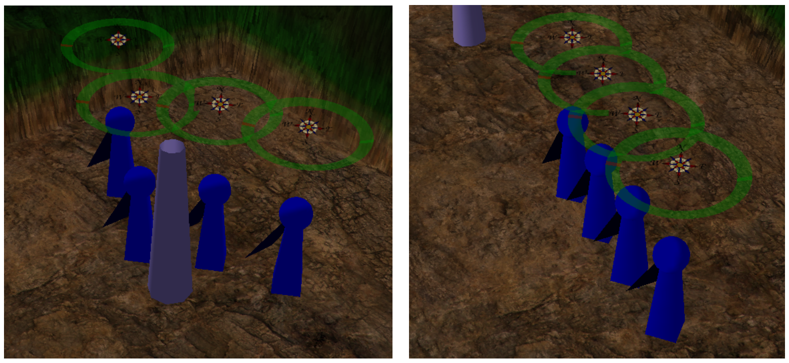

Figure 3.6: Bots lining up shortly after the start

approach bot B along the line perpendicular to the line from bot B to its target.

This novel approach therefore resulted in poor flocking behaviour, as it produced lines of bots when moving, and failed to seperate bots when defending a flag carrier. While this approach may be suitable in some applications, it is unsuitable in a game environment because a team of bots walking in a straight line provides an easy target for any adversary.

A commonly used flocking method is based on *electrostatic particle simulations*. For particles carrying equal magnitude electric charges, the magnitude of the force produced by the charges is given by the equation

$$f = \frac{k}{dist^2}$$

where $dist$ is the distance between the particles.

The flock centering and collision avoidance requirements of flocking can be modelled as forces. The collision avoidance can be modelled as repulsive forces between bots, and flock centering can be modelled as attractive forces between bots. The attraction forces draw the bots towards the centroid of the group, while the repulsion forces seperate the bots if they move to close to each other.

If the k value in the numerator is the same for the attraction and repulsion forces, they will always cancel each other. Under the assumption that the same denominator is used for attraction and repulsion, the force will be given by

$$f = \frac{k_{attraction} - k_{repulsion}}{dist^2}$$

This is not ideal as bots will always repel or attract the other bots on its team. The only solution is to have a different denominator for the two forces. The repulsion force need only take effect when the bots are in close proximity to each other. Over any distance, this force should be neglegible. The attraction force should be present across a distance that would include most of the group. The repulsion force should be much greater than the attraction force when bots are too close together, but the reverse if bots are further away. A change to the force function for repulsion given by

$$f_{repulsion} = \frac{k}{dist^3}$$

would provide the properties described above. The attraction force remains as first stated.

The bots still need a force directing them in the direction of their target. This force must be evident anywhere in the environment so that the group doesnt remain stationery. An option was to make this force scale with distance. This would have been something of the form

$$f = \frac{k}{dist}$$

Early prototypes and tests found this force to be too small to direct the bots from large distances. Instead, a constant force was employed. The weighting of this force was large enough so that the sum of the other forces was slightly less than it. This provided the bots with direction, while allowing the other forces to alter the bots' local movement.

The weightings of all these forces is non-trivial. The literature warns that much tweaking is required when trying to balance these [3]. This was found to be true when attemping to balance these forces for this projcet. An acceptable result was obtained, but for better results, a better method of iterative improvement would be required. One option would be to use a genetic algorithm to try to optimise the weightings, however this was not possible due to time constraints. The denominators were designed iteratively, as described above.

The results of this flocking method and a comparison with the first approach is discussed in the results and analysis section.

3.7 Line of Sight

It was expected that some form of visibility checking would be provided with the environmental analysis software that the planning component used. This expectation was not met, so visibility checking became a requirement of this component.

Two commonly used data structures for computing the field of view of bots in an environment are *binary space partition* trees and *quad* trees. These data structures are complicated, which reduces the benefit of using them if visibility checking is not the focus of this component.

An alternate method would be to check for each bot on team A, whether each bot in team B is in the field of view. Although this method has $O(n^2)$ complexity with n being the number of bots in each team, n is typically between 5 and 10. An optimisation to this method is to only compute the visibility check if the pair of bots is within the maximum sight range threshold. Another optimisation is to only check the visibility if the target bot is within the viewing cone of the viewer. If both of these conditions are satisfied, then a ray can be cast from the viewer to the target bot. If the ray intersects any obstacle, then the visibility check fails and the viewer cannot see the target bot. If no obstacle is intersected, then the check passes and the terget is visible.

This option seemed efficient enough for the scope of the problem and didn't require complex data structures. For this reason, this option was employed.

3.8 Triggers

The two most time-consuming portions of the update step in the simulator are the calls to the planning component and to the pathing function. These are unnecessary to call on every update step as the state of the game and the position of the bots does not change enough. A common optimisation is to only update planning and pathing once every second. This might still not be required, as the game is unlikely to change quickly at the start. During combat periods however, new plans might be required more often than once per second.

A solution is to determine all the events that warrant a pathing or planning update. These events were known as triggers. The first events that were seen to require updates were: if the game phase changed, if the set of visible enemies changed for any bot and if the flag moved. After some testing, it became apparent that if a number of bots were defending the flag carrier and the flag carrier was eliminated, the bots would not realise. Thus, in the event of a team member being eliminated, a trigger would be set. Another problem that became apparent was that if a bot was attacked from behind, and therefore unable to see the attacker, the bot would not realise, and would continue with its current plans. Thus, a trigger had to be added to report to a bot when if it entered into combat.

The final list of trigger events are as follows:

- The game phase changes
- The flag moves
- A bots set of visible enemies changes
- A team mate is eliminated
- A bot has engaged in combat

In some situations this would be faster than the common optimisation of updating pathing and planning once per second. In other situations it would be slower, as updates may be requested more than once per second. The goal of using triggers was to provide an optimisation that only required update calls to be made when necessary.

3.9 Experiment

The design of experiments is important in validating the success of a project. The two sections of this project that required experimental testing and validation were the pathing and flocking sections.

A goal of the project was to explore the utility of the field D* search in the games environment. To evaluate the success of this pathing method, a comparison must be made with a common pathing method found in games, the A* search. A comparison with the actual shortest path must also be made to show the magnitude of the benefits of using the field D* search. It is also important to remove as many unnecessary variables from the testing as possible. For this reason, only 1 bot was used when testing pathing, as the flocking behaviour introduces deviations from the path. To avoid floating point arithmetic imprecision, the co-ordinates of the bot were printed out, and processed after the completion of the experiment. They were processed using types with greater precision to avoid these arithmetic errors. The distance travelled along the bot's path was computed from the output point. Both the shortest path and the path produced using an A* algorithm were computed by hand.

The success of flocking behaviour lies in the three requirements of flocking, namely collision avoidance, velocity matching and flock centering. As discussed earlier, the direction of the bots need not match in the games context. Different bots in the group may have different plans. This means that, while half the group is moving towards the flag, the other half stop briefly to camp, before continuing. Speed matching is therefore not a useful test, as the bot's plans mayy require it to not match speed with other memebers in the group.

To test for collision avoidance, a measure of the distance to the nearest obstacle is required. If this value is much less than the flocking distance, then successful avoidance has not taken place. The measure used was the distance from each bot to its nearest neighbouring bot.

The last test concerned flock centering. A measure of the closeness of the group was required. One measurement for this is the distance from each bot to the *centroid* of the group. The centroid is the averaged sum of the positions of the bots in the group. If this value is low, the group has remained cohesive. If the value is high, the group has spread out more.

It is extremely difficult to produce a good experiment to quantify flocking results, especially in an adversarial environment. The different plans of the bots often affect the nature of the flock more than the flocking behaviour of the bots. One might consider experimenting with only one team present. The problem is that, this is not the nature of the environment in which the bots exist. One might consider testing flocking only when all the bots are moving so that none of them move away from the group to pursue another goal. This also presents a problem because the flocking behaviour is appropriate for plans other than moving. This makes experimenting a difficult task.

Chapter 4

Implementation

This chapter explores the details and difficulties associated with the implementation of this project. Most of these issues relate to caveats and optimisations.

The language in which the code was written was C++. This is the language that all the developers were most familiar with. C++ is also known for its speed, making it the only good choice.

4.1 Caveats

A number of subtle problems were discovered during the development of the project.

One such problem was introduced by the dia file parser. The field D* implementation and the environmental analysis software used the same dia file parser. It was discovered that this parser inverted the map around the y-axis. The dia files were also used to generate a rasterised heightmap for rendering. A different parser was used to load the dia files for this transformation. This parser did not reflect the map. Thus, the map needed to be flipped when received from the engine.

Floating point arithmetic is known to be imprecise. To make allowance for this error, the use of a small term ϵ can be employed. Instead of using a normal comparison such as “**if** dotProduct > 0“, one would compare as follows: “**if** dotProduct $> -\epsilon$“. A commonly accepted value for ϵ is 1×10^{-6}. This allows for floating point error, but the accepted error is limited to the value of ϵ. This value cannot be too great, as the number of false positives will increase.

4.2 Optimisations

A pointer that is frequently required for all three components is a pointer to the flag carrier. In the initial implementation, when a pointer to the flag carrier was required, the list of bots would be iterated through to find the flag carrier. This inefficiency was rectified by keeping a pointer to the flag carrier in the simulator. When the flag is obtained, the pointer is set. When the flag is dropped, the pointer is reset to **null**. This pointer can then be referenced instead of requiring a search for every reference.

Trigonometric functions are slow to compute. When working with vectors and angles, the use of dot and cross products can greatly speed up computation. When performing visibility checks, dot products were employed to optimise the process.

The last part of the process of checking visibility between a pair of bots was to cast a ray from the viewer to its target. This ray cast was performed using Bresenham's line drawing algorithm. This algorithm is well known and widely used and was the most appropriate solution for ray-casting.

Vectors were employed to contain the bot objects. There was a vector for alive bots and a vector for eliminated bots for each team. After the combat step in the update function, the vector of alive bots would need to be checked. If any bots had been eliminated that round, they would need to be removed from the vector of alive bots and placed in the vector of eliminated bots. Removing elements from the start or middle of a vector is inefficient. An alternative method of removing elements from the middle of a vector that was employed was to switch the eliminated bot with the bot at the end of the vector, and then to pop the last element of the vector. Popping the last element of a vector runs in constant time, which is faster than the linear time requirements for erasing an item in the middle of the vector. The looping variable was adjusted so that after switching the bots, the previous end bot would still be checked.

Chapter 5

Results and Analysis

This chapter will explore the results of the experiments and evaluate to what extent the goals of this project have been realised.

5.1 Pathing

For the experiment concerning the pathing, 15 tests were run on a variety of maps. For each test, two points were chosen, one for the starting position of the bot, and one for the flag position. The shortest path length and the path length given by an A* search were computed. The simulation was then run, with the bot moving from its start position to the flag position. The length of this path was then computed, as decribed in the design chapter. The results of these tests are summarised in the following table.

As was to be expected, the shortest path was in fact always shorter than the field D* and the A* searches. 14 of the 15 tests showed the field D* producing a shorter path than the A* search. The only exception was in test 7 where floating point error produced a number of bad checks when determining whether to move to the nearest node, or move on to the next node. This problem is was described in the design chapter. Apart from this case, the field D* always produced shorter paths than the A* algorithm. The nature of the boundaries along the paths affected the improvement that the field D* had over the A*. Curves and obstacles benefit the field D* path more, while straight boundaries and open spaces lead to less benefit over the A* search. Tests 1 and 10 show that, when following only axis-alligned paths, the field D* and the A* search both produce optimal paths.

On average, for the 15 tests, the field D* produced paths that were 89.99% the cost of the paths produced by the A* search. This is a significant improvement. Ferguson and Stentz claim that their field D* implementation produced paths 96% the cost of the paths produced by

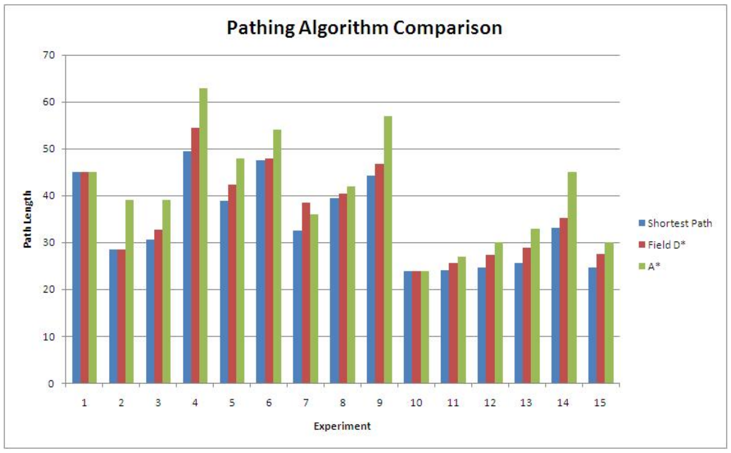

Figure 5.1: Graph comparing the path lengths of the field D* and the A* against the shortest path

a D* implementation [8]. In the case of this project, no weights were used, thus the A* and D* searches produce equivalent paths. Therefore this combination of field D* searcher and local planner produces a 6% gain on previous work with this algorithm. The best improvement recorded in the 15 tests was a path 72.96% the cost of the path produced by the A* search.

The results obtained from these tests show that on average, the field D* search produces paths approximately 90% the cost of the A* search. This is a significant improvement. It seems as though the use of triangulated meshes and the field D* search can be employed effectively in the games environment. A number of improvements are immediately apparent. One such improvement would be in using a field D* implementation that does not place the 2 requirements on searching as described in the background chapter. Another improvement would be in using triangulated meshes that provide more benefit to the field D* search. This seems promising for future work in this area.

5.2 Flocking

For the flocking tests, 10 scenarios were simulated on a variety of maps. The average distance to the centroid for each bot was recorded, as well as the distance to the nearest neighbouring bot. The results of the centroid offset test are as follows:

The first feature to account for is the spike in test 3. This particular test caused the team of bots to split into two groups, which moved in opposite directions around a large obstacle. This meant that the

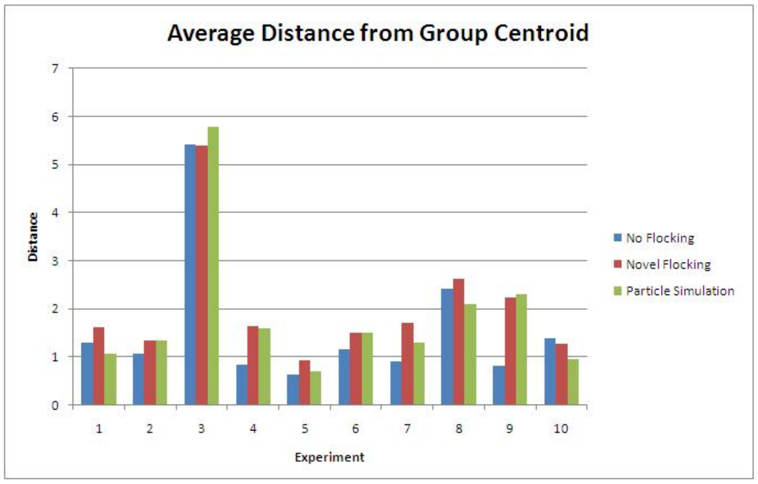

Figure 5.2: Average distance to the group centroid shown per flocking technique

centroid of the groups was approximately in between the two groups, creating large distance values between the bots and the centroid. This is as a result of the design choice to allow groups to split.

The trend of the dataset is that the tightest group formed when no flocking behaviour was present, followed by the particle simulation and then the novel approach forming the least dense group. For the tests with no flocking present, there was no collision avoidance. Bots would often move into the same space as another bot. This accounts for the low values, as the centroid would have been weighted heavily towards the groups of overlapping bots. The only factor that kept the distance from the centroid as high as it is was, was that the plans sometimes required bots to move away from the group.

8 of the 10 tests had the novel flocking produce the most sparse groups. This is because of the two problems related to this method described in the design chapter. When moving in a similar direction, the bots would form a line. This would produce a centroid in the approximate midpoint of the line. The bots at either end of the line would have a large seperation from the centroid, accounting for the high value.

The particle simulation produced a more balanced result. With more time to adjust the weightings, improved results may have been possible.

The results of the nearest neighbour test are as follows:

These results correspond strongly to the previous set of results, as does the discussion regarding them.

The tests with no flocking produced the smallest seperation 8 out of 10 times, and the tests with the novel flocking method produced

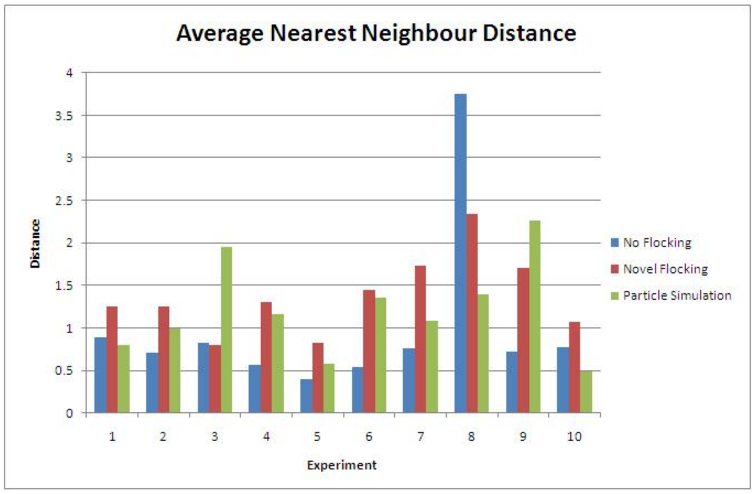

Figure 5.3: Average distance to the nearest neighbour of each bot, per flocking technique

the greatest seperation 8 of the 10 times. The reasons for this are the same as the above-mentioned reasons regarding the centroid offset test. Again, the particle simulation produced the most balanced results.

These tests suggest that the particle simulation gives the best balance between seperation and cohesion within the group. The novel approach tends to produce sparser groups.

Chapter 6

Conclusion

The goal of this project was to improve the decision-making capabilities of bots in team-based, adversarial computer games. There were three main aims of this component of the project. The first aim was to explore the utility of using a triangulated mesh to represent a game world, and a field D* search algorithm to compute paths for the bots. The second aim was to produce flocking behaviour amongst the bots. Lastly, a capture the flag game was to be implemented with a simulator that updated the game state.

It was discovered that this particular implementation of the field D* algorithm made two important assumptions. These assumptions meant that the implementation could only compute paths starting and ending at a node in the graph and that dynamic weightings could not be employed. A local planner was employed to facilitate the mobility at the end-points of the path. This produced acceptable results, although were not optimal.

A novel approach to flocking behaviour was attempted. This approach produced emergent behaviour that proved to be unacceptable as a flocking method. This was because the groups tended to form lines in the direction of motion when moving together, and to form lines perpendicular to the direction of motion when following a target. The widely-used electrostatic particle simulation method of flocking was then employed. The choice of weights proved to be a non-trivial one, which was supported by the literature. More time, or an alternative approach such as a genetic algorithm, could have been employed to produce weightings that result in better flocking behaviour, however time constraints prevented this.

The experiment to test the success of the flocking behaviour proved to be difficult to design, as the flocking behaviour, adversarial environment and plans of the bots seem inseperable. The experiment that was performed suggested that the electrostatic particle simulation approach to flocking produced the most balanced results and that the novel approach to flocking produced results that were unacceptable for this adversarial environment. This was expected after a number of design iterations.

The results of the pathing experiment proved highly positive. On average, the combination of the local planner and the field D* produced paths with a cost approximately 90% the cost of the paths produced by an A* search. This is an improvement on a figure of 96% described by Ferguson and Stentz[8]. Thus, the use of a triangulated mesh to represent the game environment, and the use of the field D* search algorithm proved to be an improvement on methods commonly employed by games.

Bibliography

[1] Activision. Call of duty. `http://www.callofduty.com/`, Nov. 2009.

[2] Blizzard Entertainment. World of warcraft community site. `http://www.worldofwarcraft.com/index.xml`, Nov. 2009.

[3] D. Bourg and G. Seemann. *AI for game developers*. O'Reilly Media, Inc., 2004.

[4] CGAL 3.4. Cgal - computational geometry algorithms library. `http://www.cgal.org/`, Nov. 2009.

[5] K. Cheng and P. Dasgupta. Coalition game-based distributed coverage of unknown environments by robot swarms. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1191–1194, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[6] Dia. Dia a drawing program. `http://projects.gnome.org/dia/`, Nov. 2009.

[7] Epic Games. Unreal tournament 3. `http://www.unrealtournament.com/uk/index.html`, Nov. 2009.

[8] D. Ferguson and A. Stentz. Field D*: An interpolation-based path planner and replanner. In *Proceedings of the International Symposium on Robotics Research (ISRR)*, pages 1926–1931. Springer, 2005.

[9] D. Ferguson and A. Stentz. Multi-resolution Field D. *Intelligent Autonomous Systems 9*, page 65, 2006.

[10] id software. id quake 4. `http://www.idsoftware.com/games/quake/quake4/`, Nov. 2009.

[11] A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. In *Proceedings of the 2004*

ACM SIGGRAPH/Eurographics symposium on Computer animation, pages 19–28. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2004.

[12] S. Koenig and M. Likhachev. Dˆ* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, pages 476–483. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[13] A. Levitin. *Introduction to the design & analysis of algorithms*. Addison-Wesley, 2003.

[14] D. J. Nowak, I. Price, and G. B. Lamont. Self organized uav swarm planning optimization for search and destroy using swarmfare simulation. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 1315–1323, Piscataway, NJ, USA, 2007. IEEE Press.

[15] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.

Chapter 7

Appendix

The list of map setting required by the simulator follows:

- seed for the random number generator
- team size
- maximum bot damage
- minimum bot damage
- sight range
- shoot range
- bot maximum health
- number of simulator steps before a bot begins regenerating health
- amount of health regenerated by the bot on each update
- distance a bot is able to move on each update
- distance required for a bot to capture the flag
- distance required for a bot to return the flag to the base
- filename for the dia map file
- the seperation distance used for the novel flocking
- which flocking method to use
- rasterised landscape
- width of the landscape
- height of the landscape

- initial flag position

The list of data required by the planning component is as follows:

- pointer to the bot needing new plans
- pointer to its team
- a pointer to the opposing team, used only for setting targets
- the flag position
- both team's base positions
- the table describing which bots are visible by each bot
- the team carrying the flag
- if the game phase has changed
- if the flag has moved
- if the bot entered combat the previous update step
- a pointer to the flag carrier
- the environment containing the spacial data
- the team size