# FORMATION PRESERVING NAVIGATION OF AGENT TEAMS IN 3-D TERRAINS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALİ GALİP BAYRAK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2008

Approval of the thesis:

**FORMATION PRESERVING NAVIGATION OF AGENT TEAMS IN 3-D TERRAINS**

submitted by **ALİ GALİP BAYRAK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

**Date:** 05.08.2008

|  |  |  |
|---|---|---|
| Name, Last name | : | Ali Galip Bayrak |
| Signature | : |  |

# ABSTRACT

FORMATION PRESERVING NAVIGATION OF AGENT TEAMS IN 3-D TERRAINS

Bayrak, Ali Galip
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Faruk Polat

August 2008, 45 pages

Navigation of a group of autonomous agents that are needed to maintain a formation is a challenging task which has not been studied much in especially 3-D terrains. This thesis presents a novel approach to collision free path finding of multiple agents preserving a predefined formation in a 3-D terrain. The proposed method could be used in many areas like navigation of semi-automated forces (SAF) at unit level in military simulations and non player characters (NPC) in computer games. The proposed path finding algorithm first computes an optimal path from an initial point to a target point after analyzing the 3-D terrain data from which it constructs a weighted graph. Then, it employs a real-time path finding algorithm specifically designed to realize the navigation of the group from one way point to the successive one on the optimal path generated at the previous stage, preserving the formation and avoiding collision both. A software was developed to test the methods discussed here.

Keywords: path finding, formation control, coordination, multi-agent systems

# ÖZ

ETMEN TAKIMLARININ 3 BOYUTLU HARİTALARDA FORMASYON KORUYARAK İLERLEMELERİ

Bayrak, Ali Galip
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Faruk Polat

Ağustos 2008, 45 sayfa

Bir grup etmenin belli bir formasyonu koruyarak ilerlemeleri, özellikle 3 boyutta daha önce çok fazla çalışılmamış zor bir problemdir. Bu tez, bir grup etmenin 3 boyutlu arazide öntanımlı bir formasyonu koruyarak çarpışmadan yol bulmalarını sağlayan yeni bir yaklaşım sunmaktadır. Sunulan metodlar, askeri simulasyonlardaki birlik seviyesindeki yarı-otonom kuvvetlerin ve bilgisayar oyunlarındaki bilgisayar tarafından kontrol edilen karakterlerin ilerlemeleri gibi birçok alanda kullanılabilir. Sunulan yol bulma algoritması, ilk olarak, başlangıç noktasından bitiş noktasına, ağırlıklı bir çizge oluşturduğu arazi verisini kullanarak optimal bir yol bulur. Daha sonra, aynı zamanda hem formasyonu koruyarak hem de çarpışmaları engelleyerek, bir önceki aşamada bulunmuş olan optimal yol üzerindeki ardışık noktalardan sırayla her seferinde bir sonrakine grubun ilerlemesini sağlayan gerçek zamanlı bir yol bulma algoritması uygulanır. Buradaki metodların test edilmesi için bir yazılım geliştirilmiştir.

Anahtar Kelimeler: yol bulma, formasyon koruma, koordinasyon, çok-etmenli sistemler

# ACKNOWLEDGEMENTS

I especially would like to thank my advisor, Prof. Dr. Faruk Polat, for his guidance and cooperation throughout my MSc studies.

I wish to thank Prof. Dr. İ. Hakkı Toroslu and Prof. Dr. Göktürk Üçoluk for their guidance for my future academic career.

I also would like to thank my friends, for their great support and friendship. Especially, my previous and new housemates Umut, Olcay, Bulut and Deniz, and friends from college Okan, Berk, Anıl, Aykut, Can, Deniz, Merve, Çağrı, Görkem, Çağla, Ebru, Alev, Elvan, İlker, Yunus and Kazım, and many others which is hard to list all here.

I wish to thank TÜBİTAK for their financial support.

The greatest thanks are to my family, for their support throughout my whole life.

To My Mother..

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF SYMBOLS

| | |
|---|---|
| **SAF** | Semi-Automated Forces |
| **NPC** | Non-Player Characters |
| **2-D** | 2-Dimensional |
| **3-D** | 3-Dimensional |

# CHAPTER 1

# INTRODUCTION

## 1.1 The Subject

Navigation of a group of mobile agents in coordination is a popular problem studied in different areas such as robotics, computer games and military simulations. The problem is to move a team of agents from a given initial location to a given final location preserving a predefined formation on a 3-D terrain. Formation is the tactical arrangement of agents in a team, like column, line and wedge (see Figure 1.1), used especially in military forces.

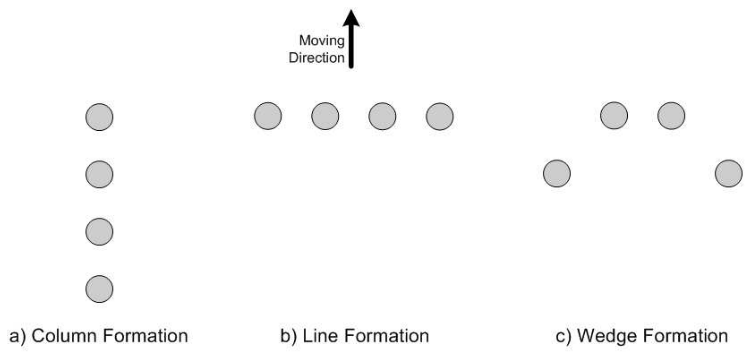


Figure 1.1: Some Common Formations

Several path finding algorithms were proposed for single and multiple robots [2] [5] [7] [9] [10]. These algorithms were generally developed for navigation in 2-D environments hence restricted as they can not be used or efficiently adopted to 3-D terrains. Also, these algorithms mainly focus on the physical capabilities of the robots, which is not the case in computer simulations, which is our primary interest in this thesis. In computer games, especially in strategy games, algorithms are designed for moving a group of soldiers, or vehicles in the simulated battlefields [6] [13] [14]. In military simulations, there has been growing interest in modelling behaviors at both individual and unit level to simulate decision making and tactics used in real-life for simulation based training, analysis and acquisition (OneSAF, TeBAT, ModSAF, SWARMM). In that respect, navigation of individual soldiers/vehicles and that of a unit require efficient algorithms to be embedded in Semi-Automated Forces (SAF) [11]. There are only a few published works for the so-called path finding task in the areas of military simulations and computer games. Motivated by this, we developed algorithms for formation preserving navigation task of multiple autonomous agents in 3-D terrains and tested them on real 3-D maps.

## 1.2 Scope and Objective

In this thesis, we developed a novel algorithm for finding an efficient path between two points for a group of agents that are also required to maintain a pre-defined formation and avoid collisions with each other and with obstacles in 3-D terrains. We divided the problem into three main parts; constructing a search graph, finding the path and maintaining the formation while moving [3].

We first construct a directed weighted graph where nodes corresponds to way points some of which will define the route, edges to the accessibility of way points connected, weights to the cost between the way points connected, after analysing and identifying important terrain features. This graph is constructed once at the beginning of the simulation, before the team actually starts to move.

Then, with the help of an off-line planner, we determine a path for the team on the constructed search graph. The planner, first, uses an informed search technique, actually A*, and finds an optimal path (a sequence of way points in the order they will be visited). Then it uses a smoothing algorithm in order to smooth the found path, which may be jagged. And finally, the planner determines each agent's goal position at each of the way points on the path. Note that, the off-line planner is also done before the agents start to move.

The last step is to move the agents in real-time using on-line planner. The agents, in parallel, follows the path found by the high-level planner, by using a real-time algorithm that plans and navigates agents between two successive way points on the solution path, avoiding collisions and maintaining the formation. Also, team is able to rearrange the formation, when an agent loses its mobility, which may be faced in games and military simulations.

## 1.3 Outline

In Chapter 2, related works are discussed. In Chapter 3, we define the environment and describe how terrain features are extracted from the 3-D terrain data and how the search graph is constructed. Chapter 4 contains an overview of our formation preserving path finding method. In the chapter, first, steps of off-line path planning phase is given. Then, the on-line path finding algorithm is given. Experimental results and sample runs are given in Chapter 5. Finally, conclusions and future research directions are given in Chapter 6.

# CHAPTER 2

# RELATED WORK

There is a growing interest in autonomous agents and multi-agent systems in computer simulations. In most of the games, there are non-player characters that act autonomously to overcome some real-life problems, either in cooperation with or against the player. While single-agent decision making mechanism is more popular in games, in some sort of the games, especially in real-time strategy games, multi-agent systems are used, e.g. agents can work on a cooperative task in order to beat the player. Also, in military simulations, soldiers and vehicles, either individually or at unit level can simulate the decision making and tactics used in real-life.

Since mostly the agents are mobile, the most frequently faced problem in such simulations is the path finding problem. There have been many works on single agent path finding problem. However, there is not so much published work on multi-agent case. Multi-agent formation preserving path finding is used especially in military simulations and games. In military simulations like OneSAF (One Semi-Automated Forces) and ModSAF (Modular Semi-Automated Forces) and games like Force21 [14] units are needed to move on 3-D terrain, preserving formation.

The path finding problem mainly consists of three parts: how to represent the environment and construct a search graph, how to search and find a path on the constructed graph and how to realize the found path in real-time.

For representation of environment, there are many approaches. However, most of them are designed for 2-D and not very efficient in 3-D domains. The representations generally focus on two important factors. One is the number of vertices it generates for the search graph and the other is the reliability of the generated graph. The number of vertices is very important, such that generally the higher the number of vertices, the more time it takes while searching on the graph. Reliability is a key factor, such that the constructed graph should preserve the connectedness of vertices. For example, if one point is reachable from the other, then on the search graph it should be so, and vice versa.

The most popular environment representation is grid representation. The environment is divided into equal sized square cells, each of which is either reachable or not, and these cells are considered as the vertices of graph. The edges of the graph are between the reachable neighboring cells. This representation can be implemented in such a way that it preserves reliability, but efficiency is not the main concern of the representation.

Another popular representation is visibility graph [4]. It considers the environment as a huge polygon, in which there are small polygons representing the unreachable areas. Then, each corner of these polygons are considered as vertices of the graph and the visibility of a pair to each other determines the edges. This representation is efficient in 2-D environments.

Delaunay triangulation can also be used for environment representation [4]. It divides the environment into triangles, using the corners of edges. Centers of these triangles can be used as vertices and edges are inserted between neighboring ones if they are accessible. Voronoi diagram is another method, which is dual to Delaunay triangulation in graph theoretical approach.

The second important phase in path finding is the search on the constructed graph. Since time efficiency is one of the most important factors real-time applications, generally informed search techniques, especially A* is used. A* is the most widely-known form of best-first search [12]. A*, at each step during the search, expands the node having the lowest $f(n)$ value, where $f(n) = g(n) + h(n)$, $g(n)$ is the path cost from start node to node $n$ and $h(n)$ is the heuristic function which is the estimated cost of the cheapest path from node $n$ to goal node. A* is optimal if $h(n)$ is an *admissible heuristic*, that is, provided that $h(n)$ never overestimates the cost to reach the goal.

The last step in path finding is the realization of found path. In single agent case, this problem is easy to handle. The agent can move on the line between successive way points along the path and simple algorithms can be used in order to avoid collisions. However, multi-agent path finding problem brings many additional constraints compared to single-agent case, especially if run on 3-D terrains. Agents should consider the mobility, position and velocity of other agents. Representation of environment is an important factor at this point. *Virtual structure*, introduced in [9] by Lewis and Tan, considers the agent group as a rigid body and planning is done for this body. Most common method is to let each agent decide its own path according to its relative position with respect to other agents. Desai et al presented a graph theoretical approach where each agent determines its location according to the locations of other agents and there is a leader agent who does not follow any other agent but leads the group [7]. The relation between two agents consists of relative distance and orientation between them. There are several methods that make use of priorities among agents to describe a formation in a team [10].

# CHAPTER 3

# DESCRIPTION AND REPRESENTATION OF THE ENVIRONMENT

## 3.1 Properties of the Environment

The environment is a 3-D virtual world made up of a terrain and objects placed on it. Terrain is represented as a height map. The natural and man-made objects such as trees and buildings are represented with polygons and they are treated as obstacles. There are autonomous agents in the environment and they can collaborate to achieve the task. The environment is considered to be fully observable, but the proposed methods can also be employed in partially observable environments, with some little improvements. The problem is discrete-time and continuous-state; percepts and the actions of agents are handled at discrete time steps and agents can move to a range of continuous points on the terrain at each time step.

Agents are mobile and holonomic[1] and aim to reach to a specified target point with a predefined average speed. In order to maintain formation and avoid collisions, agents may tune their speeds. Agents in a team are considered to be physically identical and they are restricted by their physical capabilities, such as maximum/minimum slope they can ascend/descend, maximum positive/negative acceleration and maximal speed they can reach. The communication between agents of the same team is considered to be perfect.

[1]A holonomic agent is an agent which can rotate at the same place with a turning radius of zero [5]

## 3.2 Representation of the Environment and Construction of the Search Graph

Concerning the representation of the environment, the most common approach is the *grid* representation where the environment is divided into a number of square cells of the same size, where each cell is considered either as obstacle, or not. Search algorithms for finding a path from one cell to another, may consider the set of midpoints of these cells. The drawback of this method is that cell size should be determined carefully. Figure 3.1 shows some examples of the grid representation with different cell sizes. If the size is small, the number of points in the search space will be huge, as shown in Figure 3.1(b). If the size is big, the homogeneity (obstacle/not) of the cells will decrease since each cell is considered either obstacle or not, and even it may result in failure of the search as shown in Figure 3.1(d). The ideal cell size for this case is shown in Figure 3.1(c).

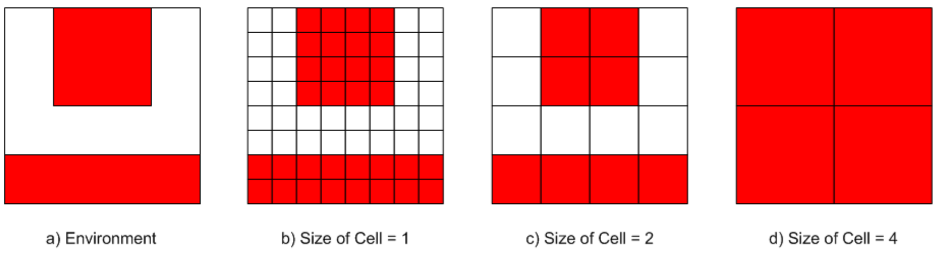


Figure 3.1: Grid Representation

There are some other methods like triangulation, Voronoi diagrams and visibility graphs [4]. However, in 3-D terrains having lots of contour lines, these methods may result in huge number of points to be considered in the search. So, it may be better to modify the grid representation, that minimizes the number of points and preserves homogeneity. In [8], Kambhampati and Davis propose a multi-resolution representation with quad trees. In this thesis, we employ a simpler but successful method that analyzes the terrain height map to extract useful features from which it identifies way points to be used by the path planner.

Concerning the representation, we consider the terrain as a 3-D mathematical surface and find its critical and singular points, and construct a search graph using these points as vertices of the graph. Since the terrain height map data is discrete, heights of some equidistance points are extracted from this data and interpolation is used for finding the height of any other point.

First, we divide the map into small square cells and take the center points of these cells, as in the grid method. The size of a cell should be as small such that it could be considered as homogeneous. Let *sizeOfCell* denote the size of a cell. Then, determine the critical points of the terrain as described in Algorithm 1. The extremum points of the terrain are considered critical points of it. The basic idea of the algorithm is as follows. First, *mark* the points that are *local maxima* and *local minima* according to their height. The *mark*ing process adds the point as a vertex to the set which will be used in the search (i.e, search graph). A point can be considered *local maxima* if its height is greater than any of its neighbors (4 or 8 neighborhood can be assumed for simplicity, we take 4 neighborhood for this study but the method is easily adaptable to 8 neighborhood) and *local minima* if its height is less than any of its neighbors. Also, mark the points that are *partially extrema*. A point can be considered as *partially extrema* if it is either maxima, or minima on every opposing pair and it is not local maxima or local minima. For example, if the height of a point is greater than the height of its south and north neighbors and less than the height of its west and east neighbors it is *partially extrema*. The points that are marked (local maximum, local minimum and partially extremum points) will form the set of extremum points. Figure 3.2 shows examples for each of maxima (a), minima (b) and partially extrema (c) respectively. In the figure, the numbers in the cells represent the heights and the points in the center cells are the extremum points.

The terrain may be rough and hence there may be some very small hills or cavities. To eliminate such extremum points, the condition in Formula 3.1 is checked against every marked point $p$. Note that, $nearestMarked(p, z)$ denotes the marked point that is nearest to the point $p$ in direction $z$. The condition checks four nearest marked points in four neighboring directions and returns whether the height differences between $p$ and each of these four points are within a threshold. If the condition

**Algorithm 1** DetermineCriticalPoints(terrainGridData) : SetOfPoints

```
1: // neighbor($x$, $dir$) : The point that is $sizeOfCell$ unit far from $x$ (i.e. neighbor of $x$) in the direction $dir$. $dir$ is one of the following: NORTH, SOUTH, WEST, EAST
2: // height($x$) : The height of point $x$
3: $S \leftarrow \emptyset$ // Let $S$ denote the set of points to be returned
4: for all point $p$ in terrainGridData do
5:   if $\forall z$ ($z \in$ {NORTH, SOUTH, EAST, WEST} $\Longrightarrow$ height($p$) $>$ height(neighbor($p$,$z$))) then
6:     $S \leftarrow S \cup \{p\}$ // $p$ is local maxima, add it to the set $S$
7:   else if $\forall z$ ($z \in$ {NORTH, SOUTH, EAST, WEST} $\Longrightarrow$ height($p$) $<$ height(neighbor($p$,$z$))) then
8:     $S \leftarrow S \cup \{p\}$ // $p$ is local minima, add it to the set $S$
9:   else if $\forall z$ ($z \in$ {NORTH, SOUTH} $\Longrightarrow$ height($p$) $>$ height(neighbor($p$,$z$))) $\wedge$ $\forall z$ ($z \in$ {EAST, WEST} $\Longrightarrow$ height($p$) $<$ height(neighbor($p$,$z$))) then
10:    $S \leftarrow S \cup \{p\}$ // $p$ is partially extrema, add it to the set $S$
11:  else if $\forall z$ ($z \in$ {EAST, WEST} $\Longrightarrow$ height($p$) $>$ height(neighbor($p$,$z$))) $\wedge$ $\forall z$ ($z \in$ {NORTH, SOUTH} $\Longrightarrow$ height($p$) $<$ height(neighbor($p$,$z$))) then
12:    $S \leftarrow S \cup \{p\}$ // $p$ is partially extrema, add it to the set $S$
13:  end if
14: end for
15: return $S$
```

evaluates to true for $p$, then we unmark $p$. $k$ can be considered as the height that an agent can pass over easily.

$$\forall z(z \in \{NORTH, SOUTH, EAST, WEST\} \Longrightarrow |height(p) - height(nearestMarked(p, z))| > k) \tag{3.1}$$

Up to this point, we finished determining local extremum points. Next step is to determine and *mark* the *singular points*. In mathematics, a point $p$ of a curve $c$ is considered as singular, if $c$ is not *well-behaved* in some manner, such as differentiability, at $p$. Generally, the term is used for the points where the curve is not differentiable.

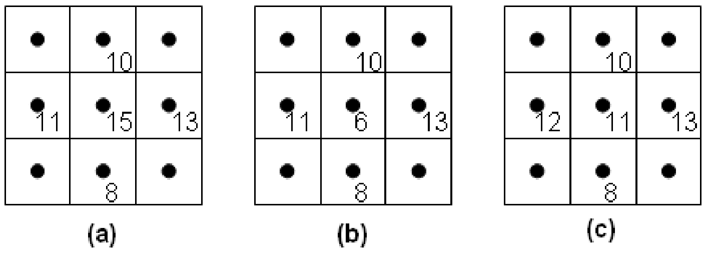


Figure 3.2: Extremum Points

In our work, we classified a point as singular, if it is not accessible from at least one of its neighbors. A point is inaccessible from its neighbor if the slope between them prevents the agent movement (because of agent's physical capability). Algorithm 2 returns the set of singular points.

**Algorithm 2** DetermineSingularPoints(terrainGridData) : SetOfPoints

```
1: $S \leftarrow \emptyset$ // Let $S$ denote the set of points to be returned
2: for all point $p$ in terrainGridData do
3:     if $\exists z$ ($z \in$ {NORTH, SOUTH, EAST, WEST} $\wedge$ inaccesible($p$, neighbor($p$,$z$))) then
4:         $S \leftarrow S \cup \{p\}$ // $p$ is singular, add it to the set $S$
5:     end if
6: end for
7: return $S$
```

Figure 3.3 contains a sample terrain where local maxima and minima are shown with small black circles and singular points are shown with small black squares.

Next step is to mark the borders of the obstacles. If a point is on the border of at least one obstacle, then it is marked. The borders of obstacles can be determined by using *Bresenham's line drawing algorithm* or any other line drawing algorithm. Figure 3.4 contains an example where dark squares represent the marked points.

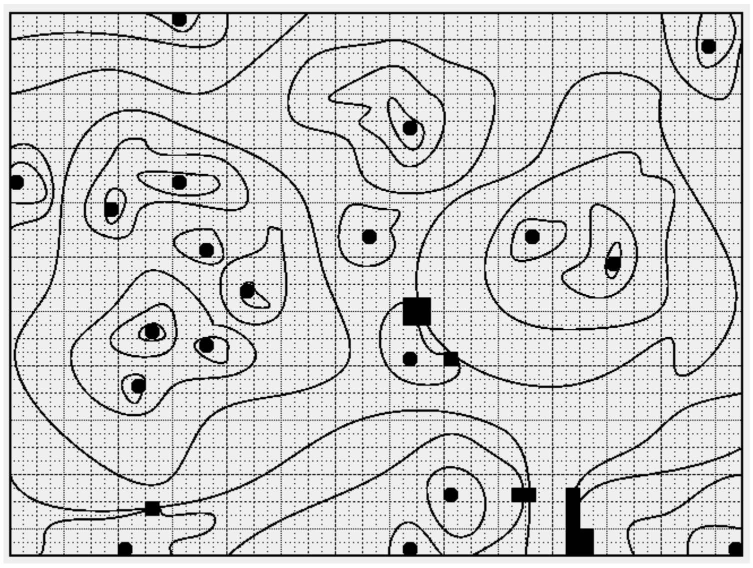


Figure 3.3: Sample Environment

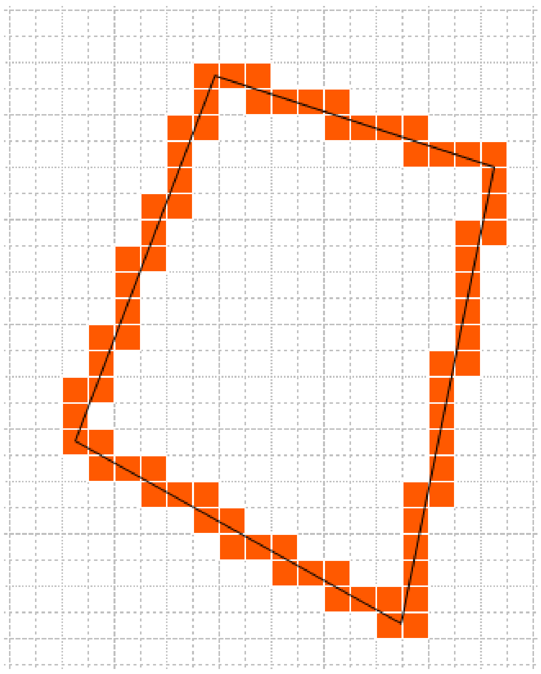


Figure 3.4: Obstacle Borders

Then, we mark the initial and goal locations of agents.

The points marked up to this point can express the characteristic properties of the terrain. However, since some of these marked points might be the only marked point in its row/column, search might fail because of disconnectedness of these points. To prevent this, we mark some additional points using Algorithm 3. Let's call a point *alone*, if it is the only marked point in its row. The algorithm, starting from the first row, traverses all the rows and if finds an *alone* point $p$ on a row, it marks the point that is on the same column with $p$ and on the same row with the point that is previous *alone* point (if $p$ first *alone* point, then do not mark any point). Figure 3.5.a shows an example of this step. Similar process is done for the *alone* points in columns as in Figure 3.5.b.

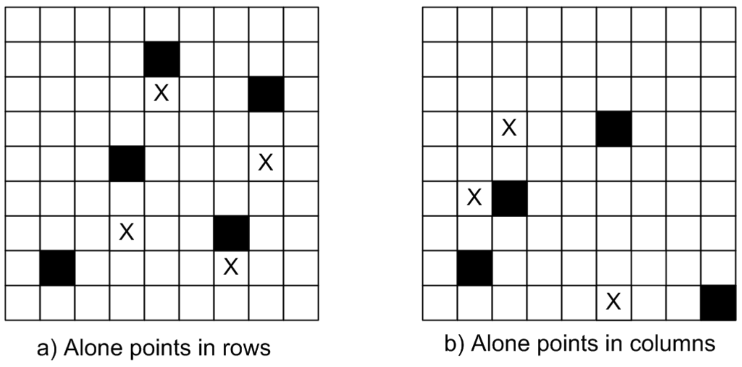


Figure 3.5: Preventing *alone* Points

If the agent only considers the points marked so far in determining the path to its destination, the search will guarantee to find a path if there is one, but the path realized may not be natural since it only considers critical and singular points. In order to prevent this and find more natural paths not only considering extremum and singular points, we mark some additional points as follows. Let's call two marked points *adjacent* if they are in the same row or column and there are no other marked points or obstacles in between them. For each adjacent point pairs, we draw a line joining them. We mark such points on this line that, two consecutive recently marked

**Algorithm 3** AlonePoints(terrainGridData)

```
1: $lastAloneRow \leftarrow 0$
2: for $i$=1 to $n$ /*$n$ denotes the number of rows*/ do
3:     $numberOfMarkedPoints \leftarrow 0$, $lastMarked \leftarrow 0$
4:     for $j$=1 to $m$ /*$m$ denotes the number of columns*/ do
5:         if isMarked($i$, $j$) then
6:             $numberOfMarkedPoints++$
7:             if $numberOfMarkedPoints > 1$ then
8:                 break
9:             else
10:                $lastMarked \leftarrow j$
11:            end if
12:        end if
13:    end for
14:    if $numberOfMarkedPoints == 1 \wedge lastAloneRow \neq 0$ then
15:        mark($lastAloneRow$, $lastMarked$)
16:        $lastAloneRow \leftarrow i$
17:    end if
18: end for
19: $lastAloneColumn \leftarrow 0$
20: for $i$=1 to $m$ do
21:    $numberOfMarkedPoints \leftarrow 0$, $lastMarked \leftarrow 0$
22:    for $j$=1 to $n$ do
23:        if isMarked($j$, $i$) then
24:            $numberOfMarkedPoints++$
25:            if $numberOfMarkedPoints > 1$ then
26:                break
27:            else
28:                $lastMarked \leftarrow j$
29:            end if
30:        end if
31:    end for
32:    if $numberOfMarkedPoints == 1 \wedge lastAloneColumn \neq 0$ then
33:        mark($lastAloneColumn$, $lastMarked$)
34:        $lastAloneColumn \leftarrow i$
35:    end if
36: end for
```

points on the line has height difference greater than or equal to a predefined value called *height difference factor*. Two consecutive recently marked points on the line are considered *adjacent* from now on. Figure 3.6 contains an example where *height difference factor* is 1. The points shown with 'X' are recently marked.

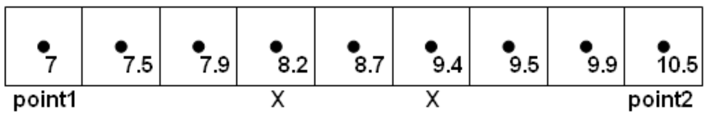


Figure 3.6: Marking Additional Points

The marked points so far will form the vertices of the graph on which the search will be performed. Since each of the marking algorithm traverses each point of the grid at most constant times, the total complexity of the algorithms used so far is $O(N)$, where $N$ denotes the number of points in the grid. The algorithm decreases the number of vertices significantly compared to grid method, especially if the terrain is not very rough.

The edges of the graph are determined as follows. For each marked point (i.e., vertex) $u$, if any adjacent point $v$ is accessible from $u$, an edge incident from $u$ to $v$ is created. A point is accessible from another if the slope between them is in the range that is determined by the physical capability of the agent and there is no obstacle in between them.

The cost of an edge is calculated as follows. We divide the edge into small pieces each of which has the length *sizeOfCell*, and then sum the costs of all these small pieces. Cost of a small piece is initially assumed to be the Euclidean distance between the two corners of the piece. Then, we multiply this cost with a function of slope such that the higher is the slope the higher is the cost. The last factor that affects the cost of an edge is whether the desired formation can be maintained along it. Note that, this is just a heuristic for calculating the costs of edges of the search graph and the calculation is done before the team actually starts to move. Considering the formation as a rectangular box and checking whether this box can move along the edge can be a solution. However, moving along the whole edge for all edges will be computationally costly. So, we used a computationally cheap method given in Algorithm 4, which checks whether this box can pass through the two vertices incident with the edge. This does not always give the desired result but is a good approximation. Figure 3.7 shows an example for Algorithm 4 to determine whether the formation can pass through a way point. The algorithm finds two points, *leftPoint* and *rightPoint*, on the line perpendicular to the edge, denoting the left and right boundaries of the line accessible from the way point and checks whether the formation can fit into the particular area between them.

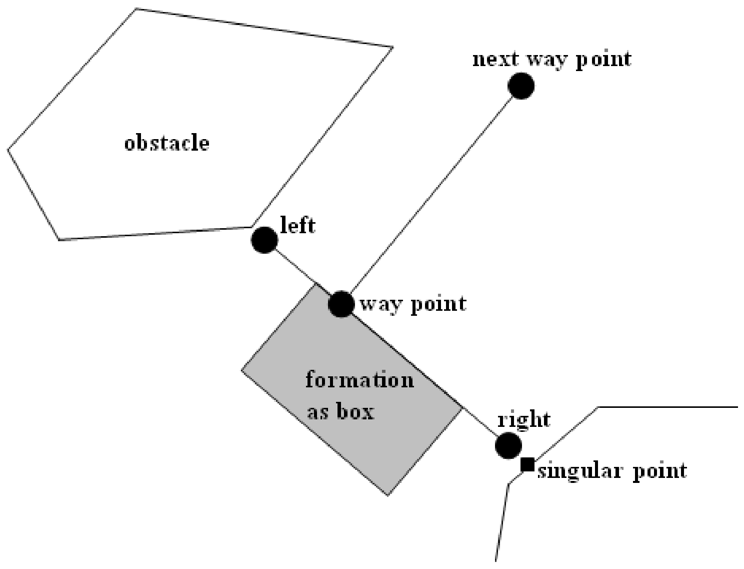


Figure 3.7: Checking Whether the Formation Can Pass Through a Point

If this algorithm returns false, we multiply the cost of the edge with a function of the distance between *leftPoint* and *rightPoint*, such that, the smaller is the distance the higher is the cost. Repeat the same procedure for the end point of the edge. After this step, construction of the search graph is completed. The algorithmic complexity

**Algorithm 4** CanFit(formationWidth, edge) : Boolean

```
1: $startPoint \leftarrow edge.start$
2: $lineP \leftarrow$ a line perpendicular to the $edge$ and passing at $startPoint$
3: $leftPoint \leftarrow startPoint$
4: for $i = 1$ to $formationWidth$ do
5:    if inaccesible($startPoint$, $leftPoint$) then
6:       break
7:    end if
8:    $leftPoint \leftarrow$ point that is $i$ * $sizeOfCell$ far from $startPoint$ on the line $lineP$
      and to the left of the $startPoint$
9: end for
10: $rightPoint \leftarrow startPoint$
11: for $i = 1$ to $formationWidth$ do
12:    if inaccesible($startPoint$, $rightPoint$) then
13:       break
14:    end if
15:    $rightPoint \leftarrow$ point that is $i$ * $sizeOfCell$ far from $startPoint$ on the line
       $lineP$ and to the right of the $startPoint$
16: end for
17: if EuclideanDistance($leftPoint$, $rightPoint$) $\geq formationWidth$ then
18:    return true
19: end if
20: return false
```

of edge cost detection using above algorithm is $O(E * formationWidth + N)$, if we traverse all grid points once in north-south direction and once in east-west direction and consider at most $2 * formationWidth$ points for each edge, where $E$ is the number of edges and $N$ is the number of grid points. Adding up this complexity with the complexity of vertex determination, total algorithmic complexity of the graph construction is found as $O(E * formationWidth + N)$.

# CHAPTER 4

# THE PROPOSED FORMATION PRESERVING PATH PLANNING METHOD

In this chapter, we will discuss our formation preserving path planning method. An overview of method is given in Figure 4.1. We first construct the search graph as described in previous chapter. Then, we use an informed search technique (i.e., A*) to find an optimal path in this search graph. If there exists a path, we use a smoothing algorithm in order to make the found path smooth, which may be jagged because of the number of neighborhood considered during graph construction. After that, for each way point along the path, we determine each agents' position at that way point, assuming that the team is able to arrive at the way point. All these steps are executed off-line before the team actually starts to move. At this point, we have a path (sequence of way points) for each individual agent. Then, in real time, each agent moves along its own path in coordination with its teammates maintaining the formation and avoiding collision. During on-line path finding, the group is able to reorganizes itself in case some agent loses its mobility. When all the team members reach their goal points, the task is accomplished. In the following sections, we will describe each of the steps of off-line and on-line path planning in details.

## 4.1 Off-line Path Finding

The number of points to be considered in the search in a realistic application can be huge, because of the terrain size. That's why there is a need for using an informed

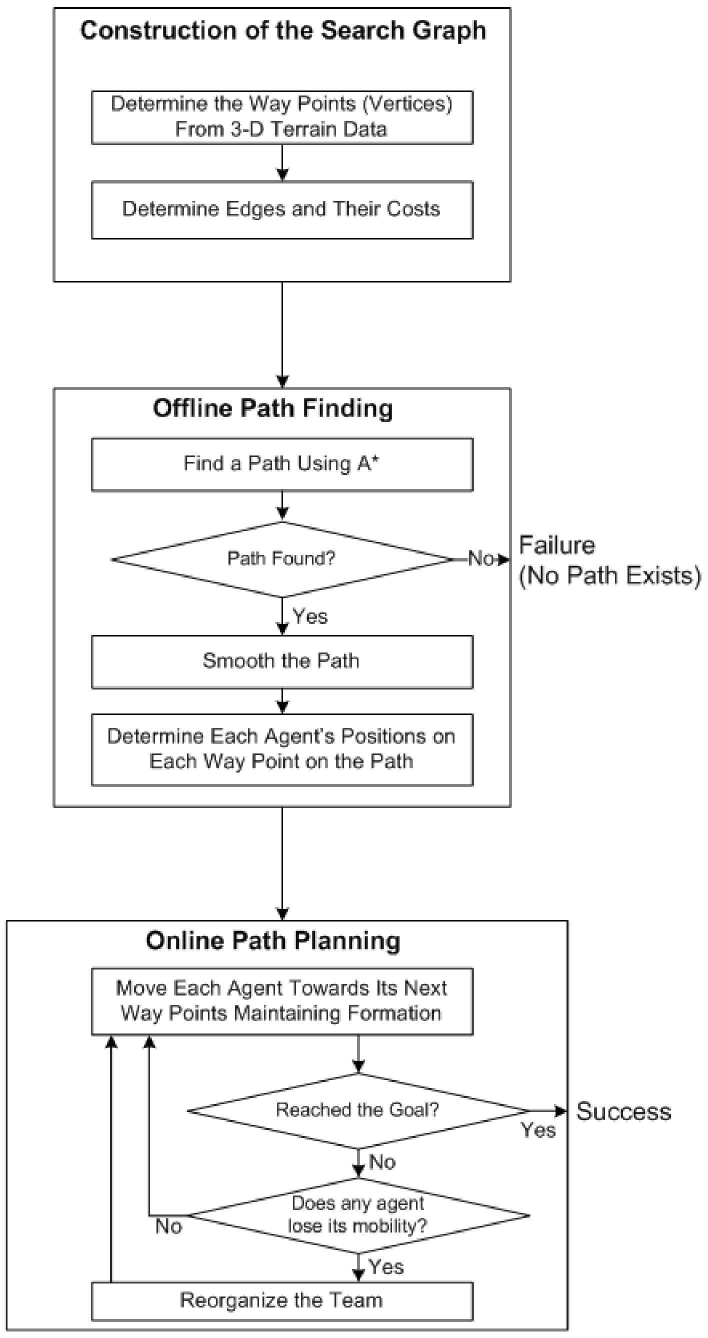

Construction of the Search Graph
Determine the Way Points (Vertices) From 3-D Terrain Data
Determine Edges and Their Costs
Offline Path Finding
Find a Path Using A*
Path Found?
No
Failure (No Path Exists)
Yes
Smooth the Path
Determine Each Agent's Positions on Each Way Point on the Path
Online Path Planning
Move Each Agent Towards Its Next Way Points Maintaining Formation
Reached the Goal?
Yes
Success
No
Does any agent lose its mobility?
No
Yes
Reorganize the Team

Figure 4.1: Steps of the Proposed Formation Preserving Path Finding Method

search technique to plan the path. We employ A* algorithm on the constructed weighted graph to produce an optimal path. Since 4 neighborhood is considered during graph construction phase, Manhattan distance[1] is used as heuristic function (namely $h(.)$). If 8 neighborhood was considered, Euclidean distance might have been used. For actual distance traveled from source(namely $g(.)$), we use the cost function mentioned in the previous chapter to determine edge costs. The heuristic function is admissible, since the Manhattan distance between any two points could never overestimate the actual cost between them.

The path generated by A* may be jagged because of the number of neighborhood used during graph construction. In order for the path to be more realistic, it should be smoothed. For path smoothing, we used the method described in [10] given in Algorithm 5, with some little changes. The basic idea of their algorithm is that we remove a point from the path if its predecessor is visible to its successor. For visibility of two points to each other, we can use any Line-of-Sight algorithms. Figure 4.2 shows an example of the smoothing process.

**Algorithm 5** SmoothPath(listOfWayPoints) : SequenceOfWayPoints

```
1: $start \leftarrow 1$
2: $returnList.add(listOfWayPoints[1])$
3: for $end = 2$ to $listOfWayPoints.length() - 1$ do
4:   if invisible($listOfWayPoints[start]$, $listOfWayPoints[end+1]$) then
5:     $returnList.add(listOfWayPoints[end])$
6:     $start \leftarrow end$
7:   end if
8: end for
9: $returnList.add(listOfWayPoints[listOfWayPoints.length()])$
10: return $returnList$
```

This algorithm is effective in 2-D. Because of the triangle inequality, the cost of smoothed path can not be larger than the cost of original path. Since the terrain is 3-D and cost function does not only consider the Euclidean distance, the cost

[1]Manhattan distance between two points is the sum of the absolute values of differences in each coordinate axis (i.e., in 2-D, $|x_1 - x_2| + |y_1 - y_2|$)

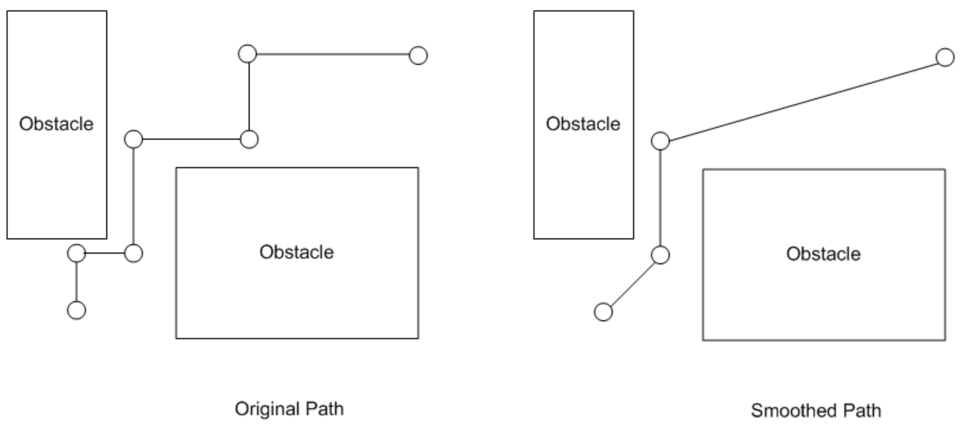


Figure 4.2: Path Smoothing Process

may sometimes can grow during this smoothing process. In order to make the path smoother but prevent the cost to grow much, step **4:1.** of the algorithm is revised as follows:

**4:1.** If the end node is not the goal, check whether *cost(start node, end node)* $\leq$ *cost(start node, immediate precedent of end node)* + *cost(immediate precedent of end node, end node)* + $\epsilon$ holds. If it holds, then choose the node successive to the end node as the new end node. Otherwise, record the immediate precedent of the end node as a new way point and choose end node as the start node with its successive as the end node. Go to step 2

### 4.1.1 Formation Representation

The spatial structure of a formation should be represented precisely. In [9], Lewis and Tan introduced the concept of *virtual structure*, considering the agent group as a rigid body and all the planning is done for this body. Another popular method is to let each agent decide its own path according to its relative position with respect to other agents. In [7], Desai et al presented a graph theoretical approach where each agent determines its location according to the locations of other agents and there is a leader agent who does not follow any other agent but leads the group. The relation between two agents consists of relative distance and orientation between them. There are several methods that make use of priorities among agents to describe a formation in a team [10].

In our work, we defined a formation by specifying relative positions of agents and their priorities. Agents are given ID's from 1 to $n$ where $n$ denotes the number of agents. The agent with lowest ID (i.e., 1) has the highest priority and is called the *leader*. The lower the ID, the higher the priority. The priority is mainly used to determine the order of movement within each discrete time step in on-line path planning. The leader agent is given an ID of 1 first. An agent's relative position is given with respect to that one of the other agents having smaller ID. Relative position of agent $a$ with respect to agent $b$ is defined with two variables, $\Delta$*depth(a,b)* and $\Delta$*width(a,b)*. $\Delta$*depth(a,b)* is the distance between $a$ and $b$ in the movement direction of the group. $\Delta$*width(a,b)* is the distance between $a$ and $b$ in the axis perpendicular to the movement direction of the group. In addition, the numbering of agents brings about another constraint that if agent $a$ is closer to the front of the group in *depth* compared to agent $b$, $a$ has lower ID than $b$'s. Note that, one can design any formation using our representation and Figure 4.3 shows how some commonly used formations are represented. In the figure, circles are the agents and numbers in the circles are ID's of the agents. A directed edge from agent $a$ to agent $b$ means that position of $a$ is described with respect to position of $b$ in the formation. $a$ is called predecessor of $b$ and $b$ is called successor of $a$.

### 4.1.2 Determining Agent Positions at Every Way Point of the Path

At any way point of the path found by off-line planner, we use Algorithm 6 to determine each agent's position in accordance with the pre-defined formation. The method described in the algorithm is as follows. First, by using the same method as in Algorithm 4, for a way point, we determine the two points $leftPoint$ and $rightPoint$, considering the edge between the way point and its successor. Then, determine the farthest accessible point to the way point, called $backPoint$, that is on the line passing through the way point and its successor and is to the back of the $wayPoint$ considering facing direction of group as front. Figure 4.4 shows an example of $leftPoint$, $rightPoint$ and $backPoint$.

Then, we determine the midpoint of *leftPoint* and *rightPoint* and superpose the

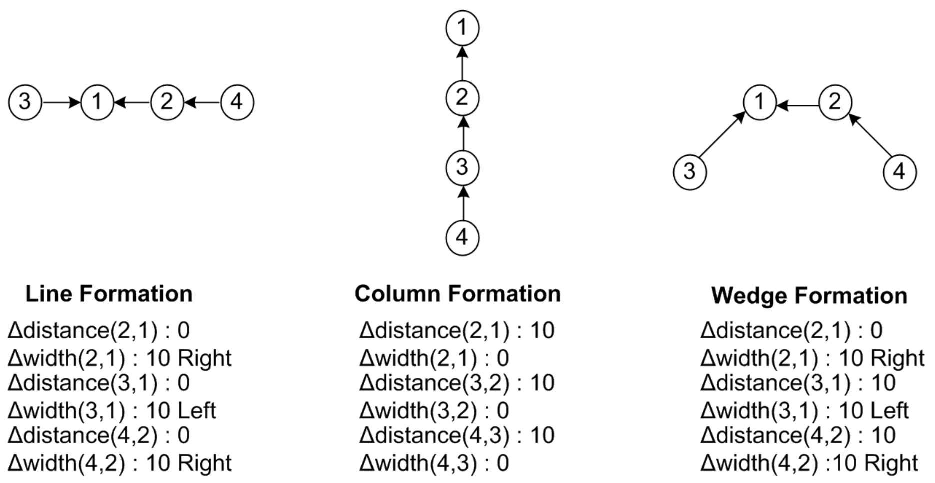


Figure 4.3: Formation Representation

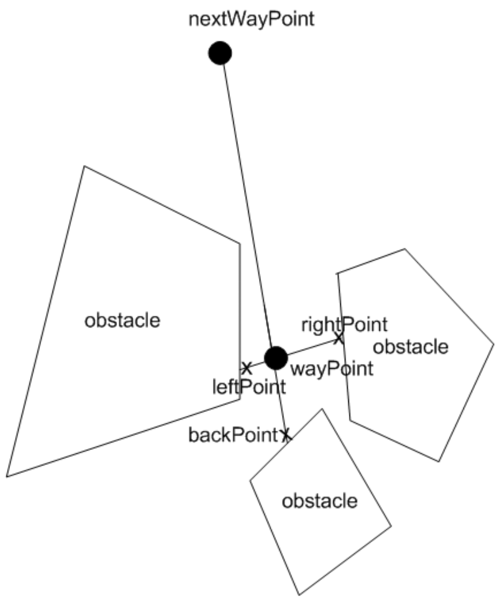


Figure 4.4: $leftPoint$, $rightPoint$ and $backPoint$

front midpoint of *bounding rectangle* of the formation on this midpoint. *Bounding rectangle* of the formation can be defined as follows. Draw a line passing through the leader agent which is perpendicular to facing direction of the formation. Then,

**Algorithm 6** PlaceAgents(formation, wayPoint, nextWayPoint)

```
1: $lineM \leftarrow$ the line passing through $wayPoint$ and $nextWayPoint$
2: $lineP \leftarrow$ the line perpendicular to the $lineM$ and passing at $wayPoint$
3: Find $leftPoint$ and $rightPoint$ as in Algorithm 4, with these arguments:
   $CanFit(formation.width$, edge($wayPoint$, $nextWayPoint$))
4: $backPoint \leftarrow wayPoint$
5: for $i = 1$ to $formation.depth$ do
6:   if inaccessible($wayPoint$, $backPoint$) then
7:     break
8:   end if
9:   $backPoint \leftarrow$ the point that is $i$ * $sizeOfCell$ far from $wayPoint$ on the line $lineM$ and to
     the back of the $wayPoint$ considering facing direction as front (see Figure 4.4)
10: end for
11: $midPointLR \leftarrow$ midpoint of $leftPoint$ and $rightPoint$ (see Figure 4.5.a)
12: $midPointBR \leftarrow$ front midpoint of bounding rectangle of formation (see Figure 4.5.b)
13: Put the bounding rectangle of formation in such a way that $midPointLR$ and $midPointBR$ are
    superposed (see Figure 4.5.c)
14: Scale bounding rectangle in left-right direction such that it can fit between $leftPoint$ and
    $rightPoint$ (see Figure 4.6.a)
15: Scale bounding rectangle in front-back direction such that it can fit between $midPointBR$ and
    $backPoint$ (see Figure 4.6.b)
16: for $i = 1$ to $n$ /*$n$ denotes the number of agents*/ do
17:   $pointOfAgent \leftarrow$ the place of agent in the bounding rectangle found above (see Figure 4.7)
18:   $lineOfAgent \leftarrow$ line parallel to $lineP$ and passing through $pointAgent$
19:   $pointOfReference \leftarrow$ the point that is intersection of $lineOfAgent$ and $lineM$
20:   if inaccessible($pointOfReference$, $pointOfAgent$) then
21:     update the place of agent in the bounding rectangle as $pointOfReference$
22:   end if
23: end for
```

draw a parallel line passing through the agent that is farthest from the first line. Finally, draw two perpendicular lines to these lines passing through the leftmost and the rightmost agents with respect to the moving direction. These 4 lines will form the bounding rectangle. This superposing is illustrated by an example in Figure 4.5.

After that, the *bounding rectangle* is scaled in both left-right and front-back directions in order to fit to the specific area near the way point, if there are inaccessible points near it. If the distance between $leftPoint$ and $rightPoint$ is not smaller than

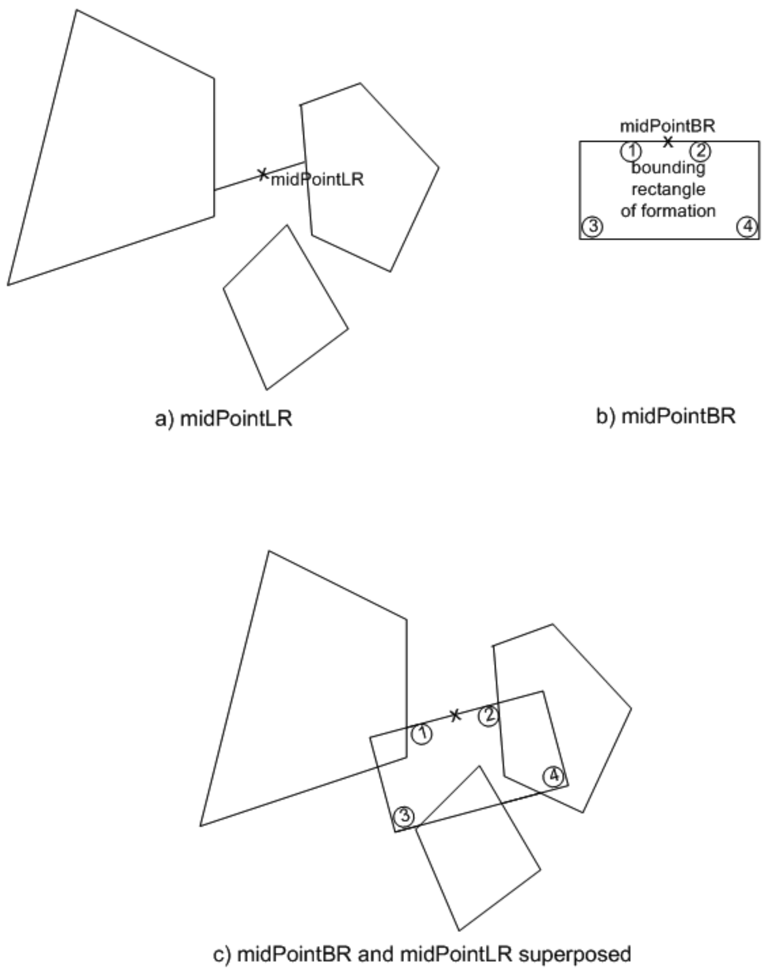

midPointLR
a) midPointLR
midPointBR
1
2
bounding rectangle of formation
3
4
b) midPointBR
1
2
3
4
c) midPointBR and midPointLR superposed

Figure 4.5: Superposing $midPointLR$ and $midPointBR$

width of formation, then there is no need to scale in left-right direction. Likewise, if the distance from $backPoint$ to the front of *bounding rectangle* is not smaller than depth of formation, then there is no need to scale in front-back direction, too. Figure 4.6 shows an example of scaling process.

Finally, for each of the agents' positions in the *bounding rectangle* formed so far, a line passing through the position and parallel to the line passing through $leftPoint$ and $rightPoint$ is drawn and intersection of this line with the line passing through the way point and successor of the way point is determined. If this intersection point is inaccessible from the position, the position of agent is shifted to the intersection point. This is because this intersection point is necessarily accessible from the way

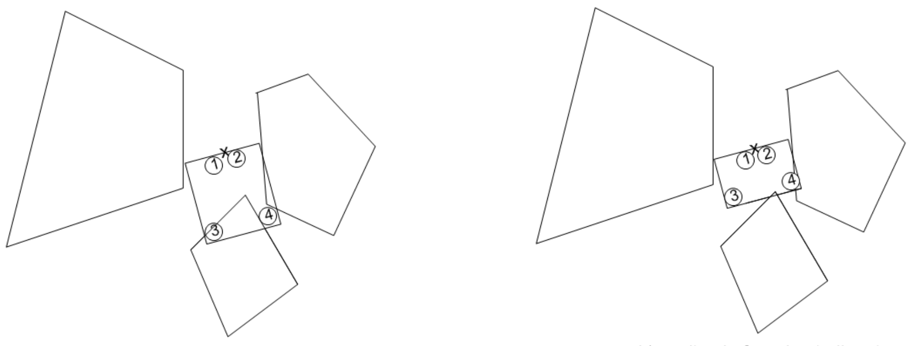

1
2
3
4
a) scaling in left-right direction
b) scaling in front-back direction

Figure 4.6: Scaling Bounding Rectangle to Fit into an Area

point and it is important for each agent's position to be accessible from the way point, in order to move in coordination. An example of shifting process is given in Figure 4.7.

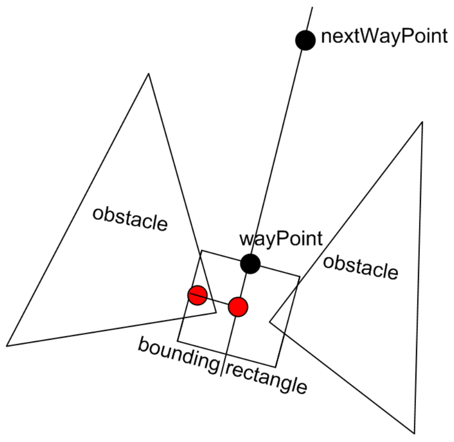


Figure 4.7: Shifting Inaccessible Points

Using this algorithm, agent positions at any way point are determined and off-line path finding step is completed. Next step is to move the agents in real time by using on-line path planning method given in next section.

## 4.2 On-line Path Finding

Having the positions calculated with the help of Algorithm 6 for each agent in the team at any way point, each agent navigates between its own way points in real time, avoiding the collisions. Agents plan their next moves and execute them at discrete time steps where the time between two successive time steps is very small. At any time step, agents plan and execute their moves in the order of their priorities (i.e., ID's).

We introduced an online path finder algorithm, low level planner, given in Algorithm 7, that navigates an agent from one way point to another which may take more than one time step. Note that, at the beginning of each time step, all the agents tune their speeds in order to maintain formation using Algorithm 8.

The basic idea behind the Algorithm 7 is to move the agent towards the next way point at each time step and if this can not be accomplished, get closer to the line from last visited way point to the next way point since it is guaranteed that the next way point is accessible from the last visited one. The reason why all agents do not move along this line is to move in formation.

In order to move smoother, the agents can change their direction a few steps before facing obstacles. This can be done by changing the step **2.1** as follows:

**2.1:** Determine whether the agent can move with its current speed in the direction of line, without colliding with any other agent in 1 unit time and without colliding with any obstacles during the next t unit times (where t determines the lookahead value, if it is huge the efficiency is affected negatively, if it is very small (i.e., 1) it is same as main algorithm)

Agents tune their speeds at each time step in order to preserve the formation while moving by using Algorithm 8. According to the distance between an agent and its predecessor, it tunes its speed and tells the predecessor to do so when the distance exceeds a threshold.

If an agent gets more than one contradictory requests, the decelerate request has higher priority over accelerate request. However, an accelerate request has higher priority over a normalize (tuning to average speed) request.

**Algorithm 7** OnlinePathFinder(lastVisitedWayPoint, nextWayPoint)

```
1: If the current location of the agent is near to the *nextWayPoint*, finish this procedure
2: Draw a line from the current location to *nextWayPoint* and let's call the direction of this line as upper direction
   1. Determine whether the agent can move without colliding with any other agent or obstacle with its current speed in upper direction
   2. If it can move, then move the agent and go to step 1 for next time step
   3. If it will collide with an agent, then wait for it in this time step and repeat step 2 in next time step
   4. Otherwise, apply these steps (i.e., steps 2.1 - 2.3) for upper-left, upper-right, left and right directions instead of upper direction, respectively
3: If step 2 failed (agent cannot move in mentioned directions), check whether the agent is on the right side or the left side of the line drawn from $lastVisitedWayPoint$ to $nextWayPoint$
   1. If the result is right, try the procedure in step 2.1 for upper, upper-left, left, lower-left directions respectively
   2. Else, try that procedure for upper, upper-right, right, lower-right directions respectively
   3. If move is possible, first move to the direction found above (3.1 or 3.2), then if result is upper, upper-left or upper-right, go to step 1 for next time step; else repeat this step (i.e., step 3) in next time step
   4. Else if it will collide with any other mobile agent, then wait for it in this time step and repeat step 3 in next time step
   5. Otherwise, wait a predefined amount of time (because the path may have been unavailable for an amount of time), and then go to step 2
```

### 4.2.1 Rearrange Formation

In the case of mobility loss of an agent, the remaining agents should rearrange their role according to the Algorithm 9. The team re-organizes by making each agent get the role (relative position and ID) of its predecessor. This is only the state change for

**Algorithm 8** TuneSpeeds(positionsOfAgents, directionOfAgents)

```
1: // $p$ is a pre-defined threshold (a real number between 0 and 1)
2: // that specifies to what degree the relative position of any agent
3: // wrt its predecessor can be altered.
4: for $i = 2$ to $numberOfAgents$ do
5:   $dist \leftarrow$ distance in $depth$ between the agent $i$ and agent $predecessor(i)$ according to the $direction(predecessor(i))$
6:   if $dist < \Delta depth(i, predecessor(i)) * (1 - p)$ then
7:     $decelerate(i)$
8:   else if $dist > \Delta depth(i, predecessor(i)) * (1 + 2 * p)$ then
9:     $decelerate(predecessor(i))$
10:  else if $dist > \Delta depth(i, predecessor(i)) * (1 + p)$ then
11:    $decelerate(i)$
12:  else
13:    $normalizespeed(i)$
14:    $normalizespeed(predecessor(i))$
15:  end if
16: end for
```

the team, the formation will be restored in time as the team moves. Figure 4.8 shows an example, where the team is moving in column formation and the agent with ID 2 has lost its mobility.

**Algorithm 9** RearrangeTeam(agent, remainingFormation)

```
1: $current \leftarrow agent$
2: while $current$ has successor in $remainingFormation$ do
3:     $child \leftarrow$ the lowest numbered successor of the $current$
4:     add $child$ to list $L$
5:     $current \leftarrow child$
6: end while
7: $current \leftarrow agent$
8: for $i = 1$ to $lengthOfList(L)$ do
9:     replace $current$ with $L[i]$
10:    $current \leftarrow L[i]$
11: end for
```

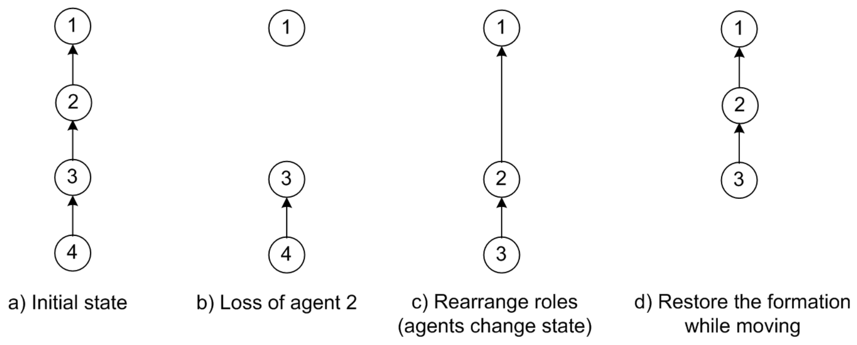

1
2
3
4
a) Initial state
1
3
4
b) Loss of agent 2
1
2
3
c) Rearrange roles (agents change state)
1
2
3
d) Restore the formation while moving

Figure 4.8: Rearrange Formation

# CHAPTER 5

# EXPERIMENTAL RESULTS AND SAMPLE RUN

The experimental results of the proposed algorithms are given in this chapter. In the first section, the testing platform and experimental setup are given. In the second section, performance of off-line planner is discussed. Section 3 contains experimental results of on-line planner. Finally, some sample screenshots from software are given in last section.

## 5.1 Experimental Setup

All the given algorithms are implemented using C++ programming language. To visualize the 3-D environment, OGRE (Object-Oriented Graphics Rendering Engine) API is used. All the codes were written platform independently and software was both tested in Linux and Windows platforms. Tests were run on a PC, which has Intel Core2 1.80GHz CPU and 1GB memory.

We randomly generated 9 terrain data and one real world terrain to test the algorithms. Randomly generated data have the size 2000x2000, 1000x1000 and 500x500, three from each. Real world data is also 2000x2000. Algorithm 10 is used to generate random *heightmaps*[1], which is converted to 3-D mesh to represent terrain. The algorithm first creates a height map whose all pixels are initialized to 0. Then, it randomly creates *oblate semi-spheroids*[2] at randomly selected points and having random

[1] *Heightmap* is an image used for storing terrain elevation data

[2] *Oblate spheroid* is a special type of ellipsoid, formed by rotating an ellipse around its minor axis

but bounded major axis and minor axis lengths. Finally, it adds up the elevations of all these spheroids at each pixel.

**Algorithm 10** GenerateHeightMap(terrainSize, numberOfSpheroids, maxMajorAxis, maxMinorAxis) : HeightMap

```
1: Create a height map $H$ with size $terrainSize$ x $terrainSize$ and initialize all
   pixels as 0
2: for $i = 1$ to $numberOfSpheroids$ do
3:     $x \leftarrow$ random($terrainSize$) // Assume that $random(i)$ generates an integer
       between 1 and $i$
4:     $y \leftarrow$ random($terrainSize$)
5:     $r \leftarrow$ random($maxMajorAxis$)
6:     $h \leftarrow$ random(min($maxMinorAxis$,$r$))
7:     Create an oblate semi-spheroid which is centered at ($x$,$y$), with major axis of
       length $r$ and minor axis of length $h$ and add the elevation of this spheroid at
       each pixel to the corresponding pixels of $H$
8: end for
9: return $H$
```

By increasing the number of spheroids or *minor axis*/*major axis* ratio, more rough terrains can be generated. One of the three randomly created terrains of the same size is relatively plain, one is relatively rough and the other is very rough. Tested terrains of size 1000x1000 are given Figure 5.1, Figure 5.2 and Figure 5.3.

## 5.2 Performance Evaluation of Off-line Planner

In this section, experimental results of off-line planner will be discussed. 9 randomly generated data and one real world data are used for testing. Results are given in Table 5.1, Table 5.2, Table 5.3 and Table 5.4. For each terrain, 4 methods are tested. These are grid method, our proposed method with *height difference threshold* 2 (HDT=2), 5 (HDT=5) and not using *height difference threshold* (No HDT). Details

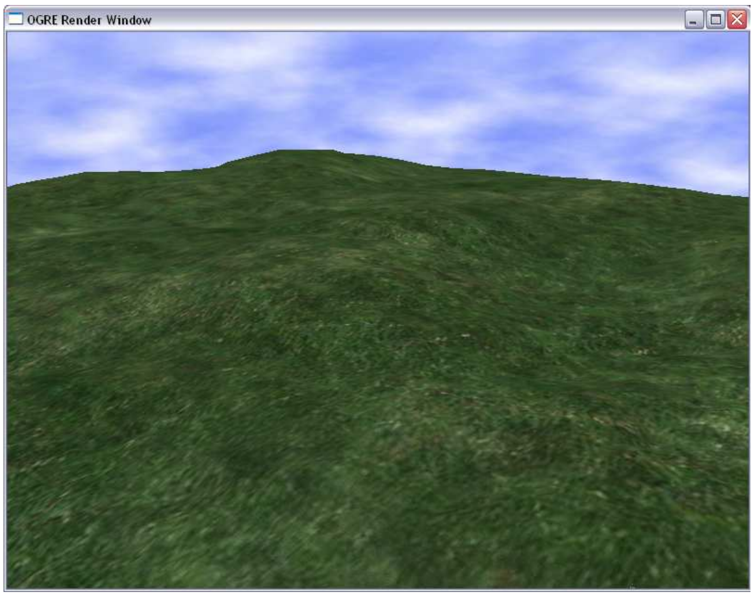


Figure 5.1: Sample Terrain 1

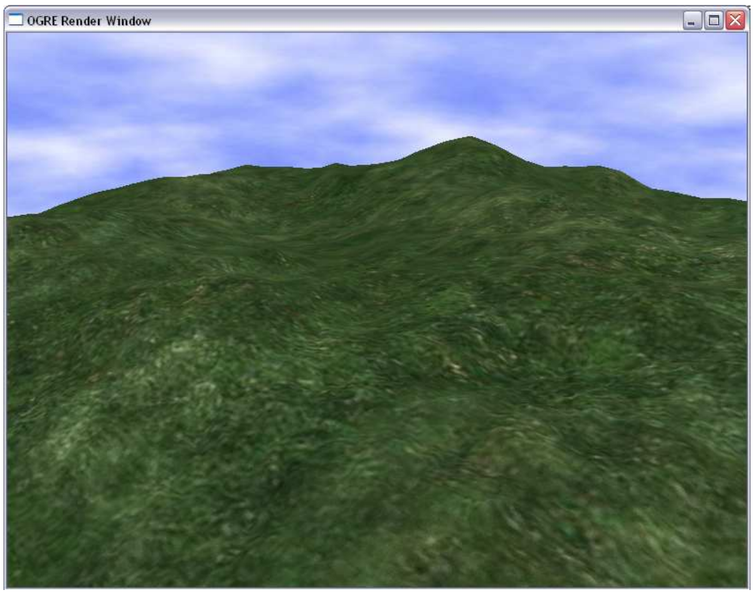


Figure 5.2: Sample Terrain 2

of the *height difference threshold* is given in Chapter 3. For each method, number of points (i.e, vertices) in the constructed search graph, time consumed for graph

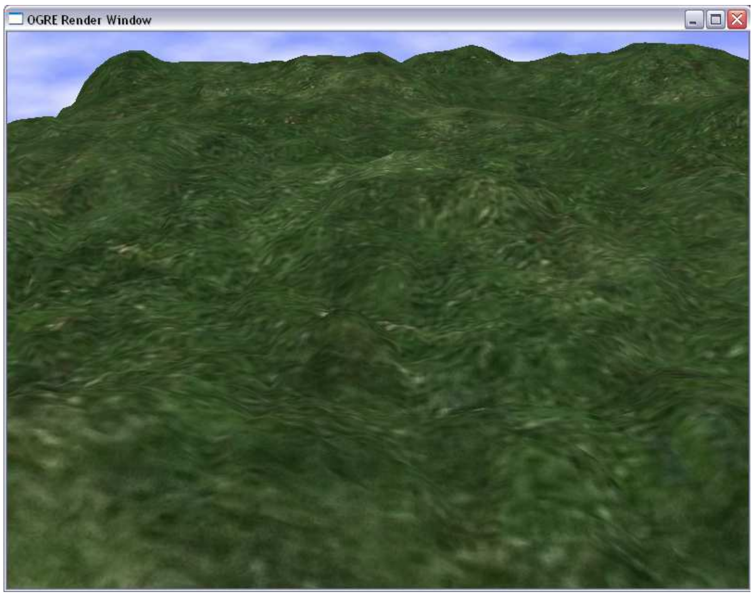


Figure 5.3: Sample Terrain 3

construction in seconds, time used for finding a path on constructed graph (time for A* + time for path smoothing), cost of the path found after running A* and cost of the path after smoothing are given. In each test data, team's mission is to find a path from one corner of terrain to opposing corner.

From these results, first, we can conclude that our method is very effective in time, especially the method named $NoHDT$. As can be seen in all terrains, the method decreases the number of points steeply and as a result, search can finish in a very short time period compared to grid method. Even if the problem is to find a path once in a given terrain, the method is very effective since *time constructing graph* + *time running A** is small compared to grid method. But in real life applications, e.g. games and military simulations, the search graph is only constructed at the beginning of the simulation once, which can be used for lots of navigation tasks throughout the simulation. So, efficiency of the *time running A** is the key criteria in such applications. We can gain up to 100 times better results in this criteria if we use the method $NoHDT$.

Second criteria for the search is the cost of path. We can conclude from the results that, the lengths of the paths found using our proposed methods are approximately

Table 5.1: Experiment on 2000x2000 Terrains

| Height Difference Threshold | Number Of Points (Vertices) | Time Constructing Graph | Time running A* | Cost of path found by A* | Cost of Smoothed Path |
|---|---|---|---|---|---|
| First Terrain | | | | | |
| HDT=2 | 308047 | 19.63 | 5.68 | 3902 | 2937 |
| HDT=5 | 119164 | 19.4 | 4.87 | 3906 | 2907 |
| No HDT | 2837 | 7.24 | 0.2 | 5783 | 2907 |
| Grid | 4000000 | 16.32 | 16.5 | 4035 | 3309 |
| Second Terrain | | | | | |
| HDT=2 | 608702 | 29.4 | 8.36 | 4091 | 3353 |
| HDT=5 | 255044 | 19.93 | 4.73 | 4085 | 3259 |
| No HDT | 30768 | 19.27 | 0.95 | 4172 | 3500 |
| Grid | 4000000 | 16.23 | 8.19 | 4290 | 3646 |
| Third Terrain | | | | | |
| HDT=2 | 667764 | 19.05 | 16.58 | 4236 | 3636 |
| HDT=5 | 306255 | 19.89 | 10.54 | 4259 | 3607 |
| No HDT | 99102 | 17.15 | 1.04 | 4343 | 4146 |
| Grid | 4000000 | 16.64 | 16.24 | 4367 | 3564 |

equal to, generally better than the one found using grid method.

## 5.3 Performance Evaluation of On-line Planner

Experimental results of on-line planner will be discussed in this section. These results are obtained from the on-line path planner during the team is moving on the path found by off-line planner using our method (No HDT) on 2000x2000 terrains and the real world terrain used above. Results can be seen in Table 5.5. On each of the four terrains, three common formations (column, line, wedge) are tested for a team of four agents, which use the path found by off-line planner. Formations are defined as in Figure 4.3 in Chapter 4. As previously mentioned, the formation is specified

Table 5.2: Experiment on 1000x1000 Terrains

| Height Difference Threshold | Number Of Points (Vertices) | Time Constructing Graph | Time running A* | Cost of path found by A* | Cost of Smoothed Path |
|---|---|---|---|---|---|
| First Terrain | | | | | |
| HDT=2 | 77661 | 6.89 | 2.78 | 1822.38 | 1336.71 |
| HDT=5 | 27774 | 6.73 | 1.09 | 1823.71 | 1336.71 |
| No HDT | 1192 | 1.66 | 0.03 | 1878.03 | 1336.71 |
| Grid | 1000000 | 3.75 | 3.17 | 1864.63 | 1476 |
| Second Terrain | | | | | |
| HDT=2 | 161407 | 6.98 | 4.17 | 1901.5 | 1513.29 |
| HDT=5 | 66539 | 4.74 | 1.3 | 1914.43 | 1504.8 |
| No HDT | 7119 | 3.02 | 0.07 | 2157.72 | 1854.16 |
| Grid | 1000000 | 4.19 | 3.58 | 1965.35 | 1567.92 |
| Third Terrain | | | | | |
| HDT=2 | 172522 | 4.59 | 3.43 | 1987 | 1590.36 |
| HDT=5 | 78881 | 6.92 | 2.08 | 2010.55 | 1600.45 |
| No HDT | 31574 | 4.24 | 0.29 | 2087.38 | 2005.66 |
| Grid | 1000000 | 3.82 | 18.6 | 2054.35 | 1642 |

by each agent's relative position with respect to its predecessor. Remember that, the relative position of $a$ with respect to its predecessor $b$ is defined with two variables $\Delta depth(a, b)$ and $\Delta width(a, b)$. $\Delta depth(a, b)$ is the distance between $a$ and $b$ in the movement direction of the group and $\Delta width(a, b)$ is the distance between $a$ and $b$ in the axis perpendicular to the movement direction of the group. In on-line planner, fluctuation of these $\Delta depth$ and $\Delta width$ values up to 20% are considered in range and up to 40% are tolerable. The results show the average absolute values of distance errors in depth and in width. Note that, relative distances in formations are set 10, so values up to 2 are considered in range and up to 4 are considered tolerable.

Results show that, agents are successful in preserving their relative position with respect to their predecessor (especially in width), since the errors in both are in

Table 5.3: Experiment on 500x500 Terrains

| Height Difference Threshold | Number Of Points (Vertices) | Time Constructing Graph | Time Running A* | Cost of Path Found by A* | Cost of Smoothed Path |
|---|---|---|---|---|---|
| First Terrain | | | | | |
| HDT=2 | 16348 | 1.11 | 0.22 | 741.554 | 540.507 |
| HDT=5 | 5378 | 1.05 | 0.07 | 746.535 | 540.507 |
| No HDT | 373 | 0.35 | 0.02 | 1093.96 | 540.507 |
| Grid | 250000 | 0.69 | 1.24 | 760.892 | 540.507 |
| Second Terrain | | | | | |
| HDT=2 | 43812 | 1.12 | 0.52 | 787.862 | 632.598 |
| HDT=5 | 18053 | 1.57 | 0.32 | 782.584 | 608.401 |
| No HDT | 2621 | 0.82 | 0.02 | 852.89 | 703.412 |
| Grid | 250000 | 0.94 | 1.11 | 825.305 | 632.04 |
| Third Terrain | | | | | |
| HDT=2 | 44818 | 1.11 | 0.54 | 796.319 | 630.541 |
| HDT=5 | 20370 | 1.1 | 0.25 | 812.369 | 718.666 |
| No HDT | 9804 | 0.98 | 0.06 | 903.353 | 840.836 |
| Grid | 250000 | 0.7 | 0.86 | 842.652 | 625.592 |

Table 5.4: Experiment on Real World Terrain

| Height Difference Threshold | Number Of Points (Vertices) | Time Constructing Graph | Time Running A* | Cost of Path Found by A* | Cost of Smoothed Path |
|---|---|---|---|---|---|
| HDT=2 | 199765 | 19.5 | 8.3 | 3942.27 | 3125.81 |
| HDT=5 | 95703 | 19.65 | 2.73 | 3953.93 | 3145.27 |
| No HDT | 62933 | 19.52 | 1.08 | 3977.06 | 3188.32 |
| Grid | 4000000 | 16.3 | 7.86 | 4075.39 | 3147.34 |

Table 5.5: Results of On-line Planner

| | | Average Error in Depth | | | | Average Error in Width | | | |
|---|---|---|---|---|---|---|---|---|---|
| Terrain | Formation | A1 | A2 | A3 | A4 | A1 | A2 | A3 | A4 |
| 1 | Column | - | 1.78 | 3.04 | 3.04 | - | 0.25 | 0.21 | 0.21 |
| | Line | - | 1.79 | 1.76 | 1.70 | - | 1.10 | 1.01 | 1.06 |
| | Wedge | - | 2.46 | 1.10 | 2.71 | - | 1.21 | 0.95 | 1.37 |
| 2 | Column | - | 3.99 | 3.99 | 3.93 | - | 0.28 | 0.33 | 0.23 |
| | Line | - | 3.21 | 2.96 | 2.94 | - | 1.02 | 1.22 | 1.55 |
| | Wedge | - | 2.59 | 1.72 | 3.04 | - | 1.33 | 1.09 | 1.46 |
| 3 | Column | - | 3.54 | 3.26 | 3.68 | - | 0.32 | 0.28 | 0.45 |
| | Line | - | 3.26 | 3.12 | 3.29 | - | 2.30 | 1.32 | 2.41 |
| | Wedge | - | 3.58 | 3.08 | 3.63 | - | 2.09 | 1.39 | 2.59 |
| Real | Column | - | 2.52 | 2.60 | 2.65 | - | 0.18 | 0.20 | 0.22 |
| | Line | - | 1.44 | 1.45 | 1.41 | - | 1.11 | 1.22 | 1.39 |
| | Wedge | - | 2.54 | 2.48 | 2.60 | - | 1.21 | 1.28 | 1.38 |

tolerable range. Because of the terrain features (e.g. roughness), it becomes hard, sometimes impossible, to preserve the distances and that's why the errors increase in 2nd and 3rd terrains.

## 5.4 Sample Runs

Screenshots from sample runs of three common formations, line, column and wedge, are given in Figure 5.4, Figure 5.5 and Figure 5.6, respectively. In figures, the red lines show the paths travelled by agents. Each agent forms a red line by putting small red spheres to its instant position at every time step during simulation.

Also, a sample run in which the team in line formation passing through a passage is given. In Figure 5.7, four screenshots from the sample run is given. The paths realized by each agent during the run can be seen in Figure 5.8.

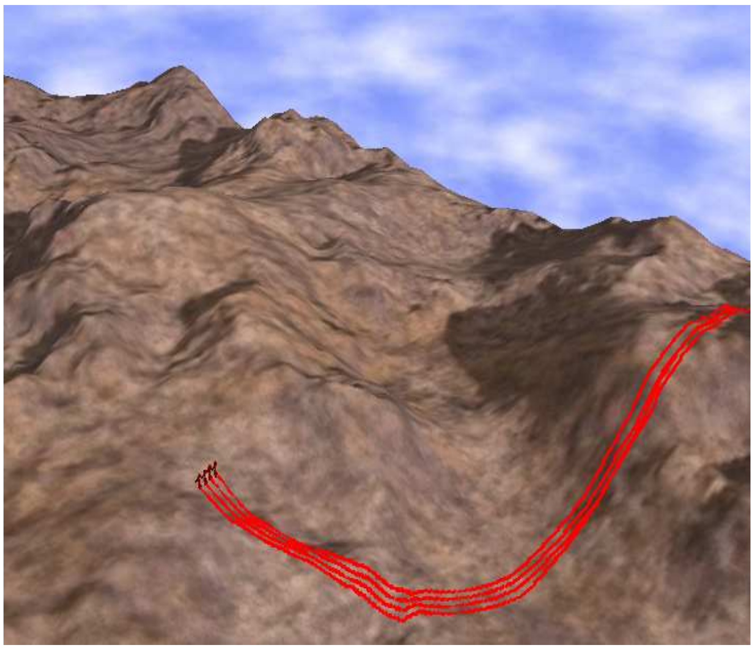


Figure 5.4: Team in Line Formation

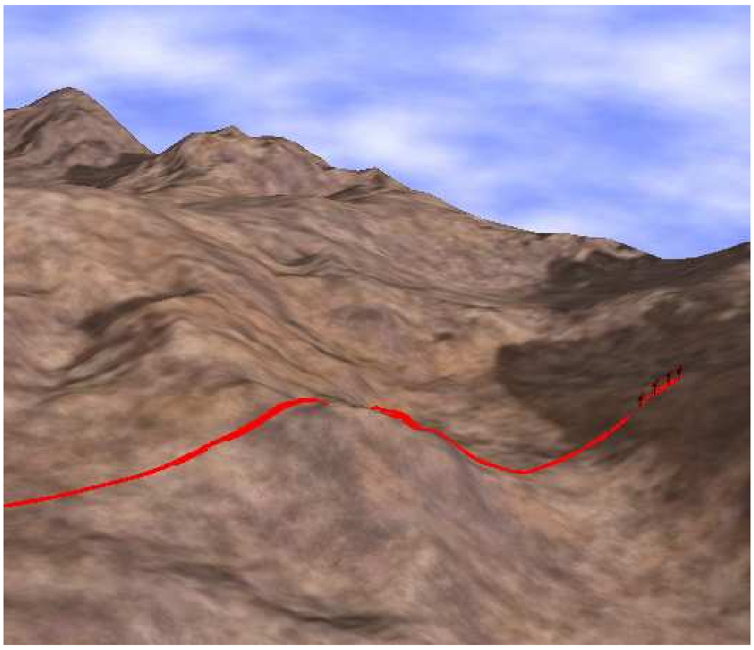


Figure 5.5: Team in Column Formation

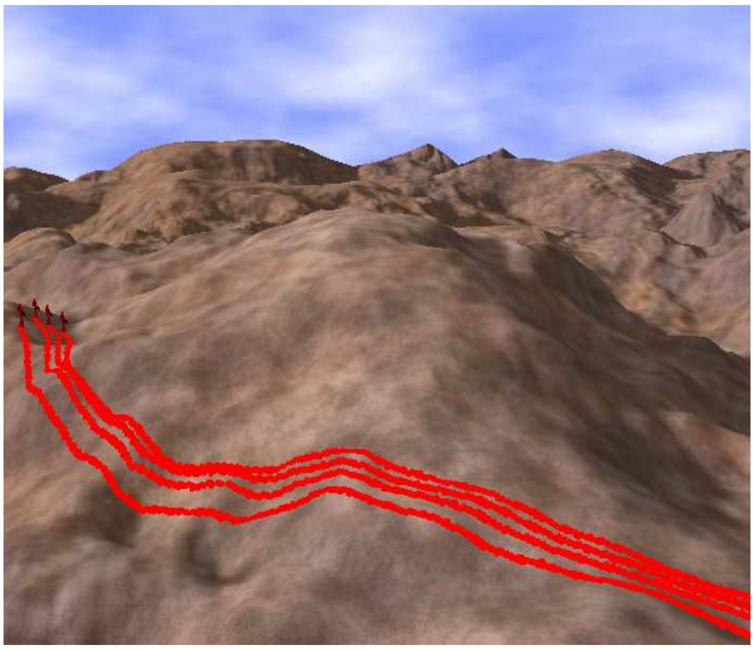
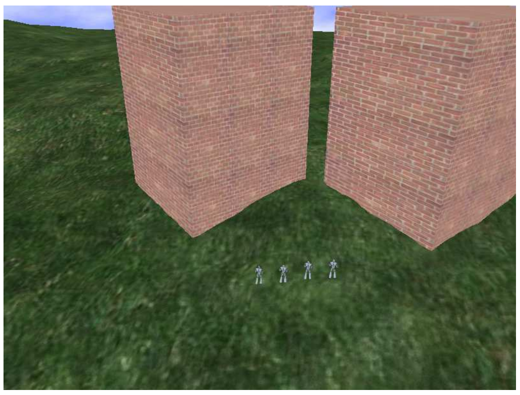
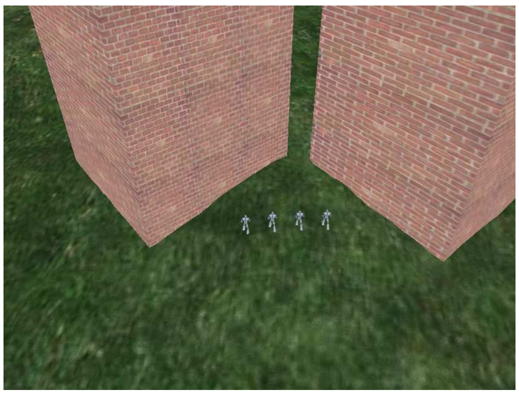


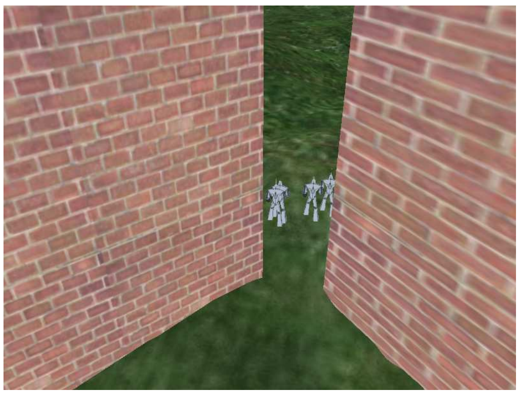
Figure 5.6: Team in Wedge Formation



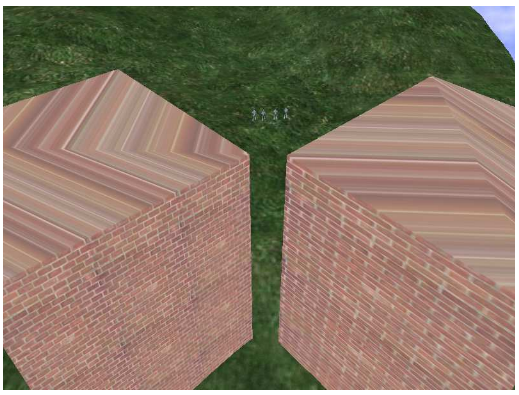
(a) At the Beginning



(b) Near to the Passage



(c) Passing the Passage



(d) After Passing

Figure 5.7: Team Passing a Passage - Screenshots

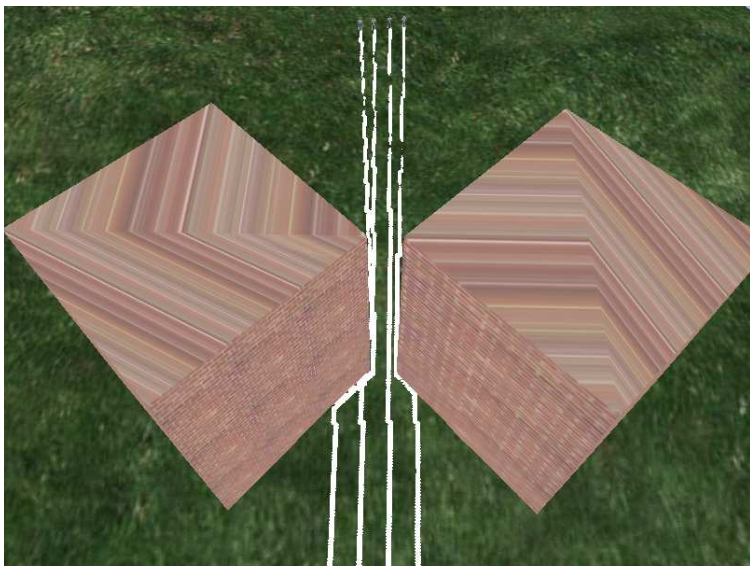


Figure 5.8: Team Passing a Passage - Paths

# CHAPTER 6

# CONCLUSIONS AND FUTURE WORK

In this work, we developed an algorithm for planning a path for a group of autonomous agents that need to move in a specified formation in 3-D terrains. We come up with a software in order to test the proposed methods and visualize the environment and behavior of agents in 3-D. Our method can be used especially in computer games and military simulations.

The proposed method, considering the group as a rigid body, first constructs a search graph by analyzing and identifying important terrain features from height map. Then a high level planner that uses A* algorithm determines an optimal path from an initial location to a target location. Then with the help of a low level on-line planner, each agent in the team navigates between way points on the solution path, avoiding collision with each other and with environmental objects. We defined a representation for group formation on which algorithms were developed to maintain formation while moving from one way point to another. The proposed method has also ability to repair formation when an agent or some agents in the team loses mobility, which is quite possible in real-life applications.

We tested our proposed algorithms both on randomly generated and real-world terrains. The off-line planner has brought a significant gain in time performance, which might be the most crucial factor in real-life applications especially in games, over grid based terrain representation. Results obtained from on-line planner were also very satisfying. The team maintained the formation through most of the path, and when it is necessary to break the formation, e.g. a team in line formation passing through a narrow passage, the team recovered the formation in a very short time period.

The algorithms were implemented and tested on static environments but they are easily adaptable to dynamic environments. Reconstructing only the changed part of the search graph and using D*[1] like algorithms would be a solution. Also, it is considered that the environment is known at the beginning of the simulation. In the same way, by considering the observability of a region as dynamism of the environment, this problem can also be handled. As a future work, methods will be adapted to dynamic and partially observable environments.

Another future research is the formation preserving navigation of hierarchical agent teams. For example, each of the fireteams[2] will move in wedge formation, while the squad[3] will move in column formation. By making some changes in formation representation, our method may also be used for this task.

[1]D* is a A* like search algorithm especially used in dynamic environments

[2]Fireteam is the smallest unit in military

[3]Squad is one level higher unit of fireteam

# REFERENCES

[1] E. Bahceci, O. Soysal, and E. Sahin. A review: Pattern formation and adaptation in multi-robot systems. Technical report, Robotics Institute, Carnegie Mellon University, October 2003.

[2] T. Balch and R. C. Arkin. Behavior-based formation control for multirobot teams. *IEEE Transactions on Robotics and Automation*, vol. 14:926–939, December 1998.

[3] A. G. Bayrak and F. Polat. Formation preserving navigation of agent teams in 3-d terrains. In *Proceedings of Industrial Simulation Conference (ISC)*, pages 148–155, June 2008.

[4] M. D. Berg, M. V. Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[5] H. Chia, H. Hsu, and A. Liu. Multiagent-based multi-team formation control for mobile robots. *Journal of Intelligent and Robotic Systems*, vol. 42:337–360, 2005.

[6] C. Dawson, editor. *AI Game Programming Wisdom.* 2002.

[7] J. P. Desai, V. Kumar, and J. P. Ostrowski. Control of changes in formation for a team of mobile robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1556–1561, May 1999.

[8] S. Kambhampati and L. Davis. Multiresolution path planning for mobile robots. *IEEE Journal of Robotics and Automation*, vol. 2:135–145, September 1986.

[9] A. Lewis and K. H. Tan. High precision formation control of mobile robots using virtual structures. *Autonomous Robots*, vol. 4:387–403, 1997.

[10] V. T. Ngo, A. D. Nguyen, and Q. P. Ha. Toward a generic architecture for robotic formations: Planning and control. In *Proceedings of the Sixth International Conference on Intelligent Technologies*, 2005.

[11] D. A. Reece. Movement behavior for soldier agents on a virtual battlefield. *Presence*, vol. 12:387–410, August 2003.

[12] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall, 2003.

[13] S. L. Tomlinson. The long and short of steering in computer games. *International Journal of Simulation*, vol. 1-2:33–46, 2004.

[14] J. V. Verth, V. Brueggemann, J. Owen, and P. McMurry. Formation-based pathfinding with real-world vehicles. In *Proceedings of the Game Developers Conference*, 2000.