MULTIRESOLUTION FORMATION PRESERVING PATH PLANNING IN 3-D VIRTUAL ENVIRONMENTS

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

BY

CAN HOŞGÖR

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

SEPTEMBER 2011

Approval of the thesis:

MULTIRESOLUTION FORMATION PRESERVING PATH PLANNING IN 3-D VIRTUAL ENVIRONMENTS

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

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. Göktürk Üçoluk
Computer Engineering Dept., METU _____

Prof. Dr. Faruk Polat
Computer Engineering Dept., METU _____

Prof. Dr. İ. Hakkı Toroslu
Computer Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU _____

Asst. Prof. Dr. Mehmet Tan
Computer Engineering Dept., TOBB ETÜ _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: CAN HOŞGÖR

Signature :

ABSTRACT

MULTIRESOLUTION FORMATION PRESERVING PATH PLANNING IN 3-D VIRTUAL ENVIRONMENTS

Hoşgör, Can

M.Sc., Department of Computer Engineering

Supervisor : Prof. Dr. Faruk Polat

September 2011, 51 pages

The complexity of the path finding and navigation problem increases when multiple agents are involved and these agents have to maintain a predefined formation while moving on a 3-D terrain. In this thesis, a novel approach for multiresolution formation representation is proposed, that allows hierarchical formations of arbitrary depth to be defined using different referencing schemes. This formation representation approach is then utilized to find and realize a collision free optimal path from an initial location to a goal location on a 3-D terrain, while preserving the formation. The proposed metod first employs a terrain analysis technique that constructs a weighted search graph from height-map data. The graph is used by an off-line search algorithm to find the shortest path. The path is realized by an on-line planner, which guides the formation along the path while avoiding collisions and maintaining the formation. The methods proposed here are easily adaptable to several application areas, especially to real time strategy games and military simulations.

Keywords: multi-agent systems, path finding, navigation, multiresolution, formation control

ÖZ

ÜÇ BOYUTLU HARİTALARDA ÇOK ÇÖZÜNÜRLÜKLÜ FORMASYONLARIN KORUNARAK YOL PLANLANMASI

Hoşgör, Can
Yüksek Lisans, Bilgisayar Mühendisliğ Bölümü
Tez Yöneticisi : Prof. Dr. Faruk Polat

Eylül 2011, 51 sayfa

Yol bulma ve navigasyon problemi 3-B arazi üzerinde yapıldığında ve öntanımlı bir formasyonu koruması gereken birden fazla etmen içerdiğinde oldukça karmaşık hale gelmektedir. Bu tezde, hiyerarşileri istenen derinlikte kurulabilen ve farklı referans verme şekilleriyle tanımlanabilen çok çözünürlüklü formasyonlara izin veren yeni bir yaklaşım sunulmuştur. Bu formasyon tanımlama yaklaşımından, 3-B bir arazide bir başlangıç ve bitiş noktası arasında çarpışmasız ve optimal yolun bulunmasında ve formasyonun korunarak gerçeklenmesinde yararlanılır. Sunulan metod, bir arazi analizi tekniği kullanarak yükseklik-haritası verisinden bir ağırlıklı çizge meydana getirir. Bu çizge üzerinden en kısa yol hesaplanır. Bulunan yol, formasyon şekli korunacak ve etmenlerin engellerle ve birbirleriyle çarpışmaları önlenecek şekilde gerçeklenir. Burada sunulan metodlar gerçek zamanlı strateji oyunları ve askeri simülasyonlar gibi bir çok uygulama alanına kolaylıkla uyarlanabilir.

Anahtar Kelimeler: çok-etmenli sistemler, yol bulma, navigasyon, çok çözünürlüklü, formasyon kontrolü

To my Family

ACKNOWLEDGMENTS

I would like to thank my superviser Prof. Dr. Faruk Polat for his encouragement and guidance in this work. I also would like to thank Ali Galip Bayrak for establishing the basis this study is built upon.

I am grateful to all my friends, especially Okan, Utku, Kerem, Elvan, Merve and Gökdeniz for their support.

Finally, I would like to thank examining committee members, Prof. Dr. Göktürk Üçoluk, Prof. Dr. İ. Hakkı Toroslu, Assoc. Prof. Dr. Halit Oğuztüzün and Asst. Prof. Dr. Mehmet Tan for their valuable comments.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

1.1 Subject

Path-planning is one of most prominent research areas in artificial intelligence. It has many application domains ranging from computer games, robotics and simulations. Over the years, the breadth of the problem has expanded greatly because of varying requirements and limitations imposed by these application domains. This resulted in a large amount of research dedicated to solving specific variants of the path-planning problem. Path-planning becomes especially interesting when there are multiple agents are involved. In this thesis, we tackle the multiresoultion formation preserving path-planning problem.

In the context of this work, a formation is defined as predefined alignment of agents. Preserving the shape of the formation means each agent tries to keep its alignment in accordance with its place in the formation. The formations are especially used in the military during navigation to protect valuable assets, prevent ambushes, or better utilize fire power of units. In Figure 1.1, four common types of formations used in the military are shown. These are, from left to right as follows:

(a) Line formation agents are aligned horizontally.

(b) Column formation agents are aligned front to back.

(c) Diamond formation agents are aligned to the corners of a diamond shape

(d) Wedge formation agents are aligned as a ”V” shape.

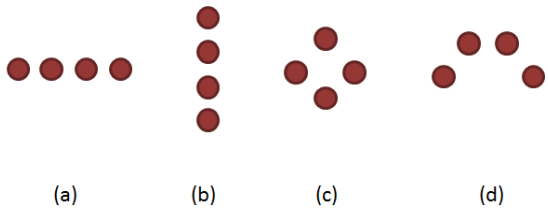

Figure 1.1: Common Types of Formations

1.2 Problem Definiton

The aim is to do path-planning for a number of agents in a 3-D terrain, by first finding an optimal path between a given start and goal locations, then moving agents from the start location to goal location. Obstacles and collisions with other agents must be avoided. And each agent must preserve its alignment in the formation as much as possible. Formation can be multiresoultion, in other words it can include sub-formations, and they can be nested to an arbitrary depth. Formation must be treated as a hierarchy rather than a flat list, thus the alignments must be preserved in a hierarchical manner.

In order to accomplish this task, we developed a two phased approach. The first one, called the off-line planning part, is run before the simulation starts and agents start their motion. It involves subtasks such as terrain analysis, search graph construction, and path finding. In the second part, which we call the on-line planning part, actual motion simulation is carried out. Using the results obtained from the off-line planner, this part runs at the start of each simulation time step to move agents to their goal locations.

1.3 Outline

The contents of this thesis are organized as follows.

- In Chapter 2, we present the related work on the subject.
- In Chapter 3, the off-line part of this method will be presented. we elaborate the environment properties and explain the environment representation used in this work. In this chapter, the terrain analysis process that transforms the height map and obstacle

list into a search graph suitable for use by agents is explained in detail. After that, the off-line part of the path finding problem is explained. Finally, we talk about the path smoothing process that makes the path found by the off line search more suitable for navigation.

- Chapter 4, we build upon the findings of the previous chapter and introduce the on-line part of the method. First we begin by explaining the formation representation method used in this work. Next, we explain the coordinate system used in formation representation. We introduce the formation referencing schemes supported by this work, which are different ways of representing the same formation. After that we explain the methods that determine the waypoints of each agent along the path, move agents between these waypoints and adjust agents speeds in order to preserve the formation.

- In Chapter 5, we evaluate the performance of off-line and on-line parts of the method. First the experimental setup is discussed then the off-line part is evaluated with both computer generated and real life test data. Next, the on-line part is evaluated for each formation referencing scheme for different formation configurations.

- In Chapter 6, we discuss conclusions reached throughout this work and give some ideas on possible enhancements to the methods in future work section.

CHAPTER 2

RELATED WORK

Path finding and navigation are probably the most fundamental problems autonomous agents need to solve. This problem has been studied extensively for the single agent case and several well performing approaches that are either generic or specific to a certain problem domain have been proposed. On the other hand, the complexity of the path finding and navigation problem increases when multiple agents are involved and they have to preserve a predefined formation while moving on the terrain.

Because of this, coordinated motion control of mobile agents has been an active research subject especially in recent years [1]. Due to the breadth of the application areas, this problem attracts attention from several fields. For example, in computer games characters that are not controlled by human players need to behave autonomously to solve problems particular to the game and one of the most common problems agents face is path finding and navigation. In real-time strategy games, many agents run path queries and use the obtained results to navigate the world at the same time. Occasionally, these agents must also coordinate their motions as a tactical formation to protect valuable units or to better utilize fire power. In addition to computer games, military simulations that are used for tactical training need to model warfare scenarios as realistically as possible. In simulations such as ModSAF (Modular Semi-Automated Forces) [2] and SWARMM [3], units can be ordered to navigate from one point to another while preserving a designated formation. Finally in robotics, autonomous robots are designed to carry out tasks like search and rescue, surveillance and exploration [4]. Due to the limited sensor and actuator abilities of individual robots, they need to align with each other to create a formation to increase their cumulative field of view or carry large objects that are beyond limits of a single robot.

Path finding and navigation problem is generally addressed in three parts [5]. The first part is generally referred to as terrain analysis [6], and search graph construction. The second part is conducting path queries on the resulting search graph and the final part is realizing the path found in the previous step.

The first part of the path finding and formation control problem is deciding on the environment representation. In older computer games, the size of levels is kept small, and level data is available as a grid. In this setting, the search graph for path finding consists of the whole grid, with the grid cells treated as vertices and connections between neighbour cells as edges. Since the size of the search graph is known and deliberately kept small beforehand, efficiency and scalability is not the main concern of this representation. Yet, in both military simulations and computer games, there is an ongoing demand in more realism. This resulted in an increase in the complexity of virtual environments, and inadequacies in existing environment representation techniques have become apparent. In order to cope with the challenges posed by this complexity, several approaches have been proposed under the title of terrain analysis. The majority of these approaches begin with the full description of the environment and aim to transform it into a simpler form so that irrelevant environment features are removed but features that are important for path planning and navigation are preserved. Another aim is to provide cues to the agents that will aid them in their reasoning process. This is done by annotating the environment. After the environment is processed, the path finding and navigation modules operate on these newly generated information, which allows them to reason about the environment both faster and more intelligently.

One of the most popular approaches in terrain representation is converting grid data to quad trees. In [7], the environment is taken as a two dimensional grid. Then it is converted to a quad-tree representation by subdividing each region into four equal sized square regions. The regions are subdivided recursively until they reach a lower bound in size (generally same as the size of an individual cell), or all the cells in a region are either completely traversable or completely occupied. After the quad-tree is constructed, a standard graph search algorithm is run starting from the leaf node containing the start position, and ending in the leaf node containing the goal position.

Another approach is preprocessing the grid data to find connected components in it. In [8], the environment is treated as a two dimensional grid consisting of boolean cells, where each

cell has a truth value depending on whether this cell is traversable or not. This data is then preprocessed to determine polygonal regions and choke points. Regions are connected components in which the agents can move freely and choke points are the points that connect these components. This representation closely resembles a graph, and it can be used to plan routes between points or to compute the size of an open area which is useful in tactical operations.

The next approach implements a flow restriction idea inspired by real-life road networks [9]. Instead of reducing the number of grid points, it works by annotating the existing grid data, with traffic flow information, by means of laying out virtual roads onto the grid. In order to avoid head-to-head collisions between agents, they limit the allowed movement direction in each row or column of the grid. A road-map consisting of lanes and intersections is constructed.

The methods discussed above are designed with the assumption of environment being two dimensional. Although this assumption is true for most real-time strategy games, these methods are not directly applicable to real-life terrains. Most of these techniques can be extended to support 3-D environments. But due to the nature of the problems they were originally designed to tackle, most of the assumptions made by these algorithms cannot be made in the 3-D case and they need heavy modifications to be usable.

The techniques for representing 3-D environments are mostly heightmap or point cloud based, depending on whether the terrain data is sampled at regular or irregular intervals. Height-map based techniques can be thought of as extended versions of the grid methods, with the addition of a height value associated with each cell. Because of their simplicity the majority of existing grid based approaches can be extended to support height-maps.

In addition to grid based methods, techniques involving polygonal meshes generated from terrain data are also popular [10]. These methods include Voronoi diagrams and Visibility Graphs [11]. Visibility Graphs treats the terrain as a polygonal mesh containing polygonal obstacles. This technique computes visibility between every pair of vertex in the polygon, and adds an edge between them if the two vertices can see each other. Voronoi diagram splits the terrain into regions called Voronoi sites, centered at the vertices of the terrain mesh. Each point on the terrain belongs to the site whose center is closest to it. After all the Voronoi sites are computed, edges are added between neighbour sites if it is possible to navigate from one to another. Delaunay triangulation is also used, which is dual to the Voronoi diagram. In [10],

Delaunay triangulation is used to produce an Irregular Triangular Mesh (ITM) from a point cloud data set.

After the search graph is constructed, the next step is finding the shortest path from start to goal locations. This problem can be solved by many well-known graph search algorithms, each having different characteristics. If the distance between a given node to the goal location can be estimated without computing the real distance, using an informed search technique like A* [12] or Focused D* [13] will yield better results. Otherwise, an uninformed search algorithm such as Dijkstra's algorithm or D* [14] should be used.

The most well-known form of graph search is A* algorithm. It is a best-first search technique that finds the lowest cost path from a start node to goal node. At each iteration, it expands the most promising node which is determined by the node having the lowest f value. The f value of a node is defined as its actual distance from the start node $g(x)$, plus its estimated distance to the goal node, $h(x)$. As long as the heuristic function is admissible, i.e., it does not overestimate a nodes actual distance to the goal, A* guarantees to find the optimal path. There are many other search techniques that are tailored for a specific problem domain, one of which is Iterative Deepening A* (IDA*) [15] that uses iterative deepening to keep memory usage lower than A*. However, it can be stated that most of these algorithms are based on the ideas of A*.

The final step of the process is realizing the path. This is done by moving agents from the start location to goal location by following the path nodes found in the previous step. For the single agent case, this task can be accomplished either by moving the agent between consecutive path points at a constant speed, or by providing steering manoeuvres to the agent at each time step [16]. Navigation between consecutive path points can be modelled using potential fields that attract the agent to goal location [17].

For the multi agent case involving formations of multiple agents, this problem poses more challenges. In [18], the rules that govern agents actions are divided into two separate parts based on their dependence on local or global knowledge about the environment. The part that depends on global knowledge is mostly concerned on moving the agents along the global path, and the part that depends on local knowledge is concerned about avoiding collisions with obstacles and maintaining the formation. Several strategies were proposed, to resolve conflicts between local and global actions, some of which favour global knowledge over local

and vice versa. In their work, it is concluded that the best results are obtained by using local control augmented by a global goal and more complete global information.

In the research reported in [19], the methodology is similar to [18]. Both of these works evaluate the line formation as an evaluation metric. Yet, [19] extends the findings of [18] by evaluating additional formation shapes and providing new formation representation types. In this work, the same formation can be represented in three different ways. In the first one, called unit-center approach, the positions of agents in the formation are given with respect to a fixed point, which is generally chosen as the centroid of the formation shape. In the second one, the position of each agent is specified with respect to a leader agent. In the final one, the position of each agent is specified with a neighbour agent. The third one can be thought of as a generalization of the second one. This representation method is also used in [5], which allow an agent to position itself with respect to any agent as long as the referenced agent has higher priority than the referring agent. The findings in this work state that unit-center approach gave the best results among three representations.

In [20], the formation of agents are treated as particles constituting to a rigid body. Path planning is done for the whole body and then the body is moved along the trajectory while the agents adjust their positions and orientations with respect to their positions in the virtual structure. There is a bidirectional flow of control between the part concerned with the position of virtual structure and the part concerned with the position of robots, where each part tries to adapt itself to the other. This results in a very good fault tolerance, where the virtual structure can easily adapt when one or more of the robots fail. The methods proposed in this research allows very precise motion and almost perfect formation maintenance. Although, this work does not consider flexible or deformable formations in its current state.

Most of the research done in formation control is limited to single level formations. On the other hand, there is a present demand in robotics, military simulations and computer games, for techniques that deal with multi-resolution formations [6].

CHAPTER 3

TERRAIN ANALYSIS AND SEARCH GRAPH CONSTRUCTION

3.1 Properties of the Environment

The environment that the agents live is a 3-D world, that contains a terrain and some natural and man made obstacles residing on that terrain. Based on the application domain, these obstacles can be trees, buildings, rivers, or areas where agents are not allowed to enter. Threat zones where there is a possibility that agents might be attacked by an enemy can also be modelled using obstacles.

In our study, terrain is represented as a height map, which is a surface where there is exactly one height value for any location. Although height maps cannot be used to model caves and overhangs, these terrain features are rarely used in navigation. Obstacles that prevent the agents from passing are represented with polygons.

The environment is fully observable, each agent has complete knowledge about the terrain, obstacles and other agents at all times. The sensors and actuators on agents are fully deterministic. There is no sensor noise and actuators are able to perform perfectly as long as the desired action is within the capabilities of agents. In order to reach their goals, it is assumed that the agents are able to perfectly communicate with each other. The communication is two-way, each agent is equipped with the ability to send and receive information.

Agents start at an initial location and aim to reach a goal location on the terrain, without colliding with obstacles or other agents, moving with respect to their physical capabilities, and trying to maintain the formation assigned to them while keeping a designated average

speed. Occasionally, the actions required to reach these aims might be conflicting with each other, or might be impossible. At such cases, the most sensible action at hand is chosen, based on the implicit action priorities defined by the algorithm.

The simulation runs in discrete time steps. At each time step, each agent senses the environment and changes its acceleration based on the information it receives about its surroundings. Afterwards, positions and velocities of every agent are computed using Euler integration, and these are provided as sensory inputs at the following time step. The simulation keeps on running until all agents are considered to be in a final state.

The formation is organized as a hierarchical group, which can include other groups and individual agents. Individual agents are holonomic and mobile. They have a circular shape with a predefined radius assigned to them. In addition to that, each agent has a set of physical capabilities it has to abide by during its motion. These capabilities determine the limits of an agent's mobility, such as its maximum speed or maximum and minimum slopes it can ascend and descend. The capabilities of a group are aggregated from the capabilities of agents and sub-groups in it, such that every capability chosen for the group is viable for every member in the group. For example, the maximum speed of a group is the minimum value of its members maximum speeds. The shape of a group is the rotated bounding box that surrounds all its members. Because of non-homogeneity among its members and terrain features limiting navigation, a group can rarely behave holonomic.

3.2 Environment Representation

The most straightforward way of representing the environment is treating it as a grid. In this approach, a virtual grid is laid over the terrain that divides it to cells of equal size. The height of each cell is sampled from the height at the midpoint of the cell and each cell is marked as navigable or not based on its height value and obstacles that intersect with it.

A major drawback of this representation is the difficulty of choosing an appropriate cell size. Because of low resolution, a grid with a very large cell size will be incapable of representing small features on the terrain. This leads to path finding algorithm finding a very rough path, or worse failing to find a path even if there exists one. A grid with a very small cell size will be capable of representing small features, at the cost of increasing the total number of cells.

This leads to quadratic increases in the search space of the path finding algorithm, without a significant quality gain on the resulting path.

In order to overcome these issues, we have come up with a method that is able to represent small features without using excessive number of cells. In natural terrains, height values change continuously among adjacent grid cells, therefore only a subset of the grid cells is relevant to path finding most of the time. Our algorithm exploits this property by first determining which points are important for path finding and then constructing a search graph using only these points as vertices. The resulting search graph contains fewer number of vertices and edges, this allows us to do path finding faster and using less space. In the following figure, a grid and the corresponding graph created from the same height-map is shown.

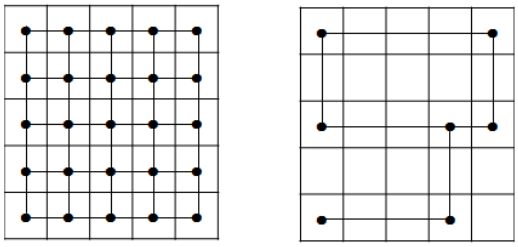

Figure 3.1: Sample grid and resulting search graph

In order to find important points, we treat the height map as a 3-D mathematical surface and analyse some of its mathematical properties to find which points are important for the search graph. After we determine the important points, these are used as vertices in the search graph. Then, we iterate over all pair of vertices (u, v), and insert an edge (u, v) between two vertices if the following conditions hold:

- u and v stay on the same row or same column on the grid, and
- There are no vertices between u and v, and
- The agents are able to move from u to v.

The resulting graph will be a directed graph because the agents might be able to move from one vertex to another in one direction but not in the reverse direction. An example to this case occurs when the agents capability for ascending and descending slopes is different. That

is, the agents might be capable of descending a slope downwards, but cannot climb the same slope upwards.

A natural result of this asymmetry is that, edge costs between two vertices in different directions should not be equal as most agents are able to move downwards easier while climbing is more difficult for them, therefore we take this important property into account while computing edge costs. The algorithm for computing edge costs will be discussed later.

The most prominent properties of a mathematical surface are its local extrema points, saddle points and singular points. These points combined with the points along obstacle borders carry enough information to represent most of the features of the terrain.

3.3 Terrain Analysis

Terrain analysis begins the same way as the naive grid approach. Terrain is divided to equal sized cells, and height values for these cells are computed. The ideal size of a cell largely dependent on the application domain. However, a good heuristic estimate for cell size can be the radius of the smallest agent in the group. As our method significantly reduces the number of nodes in the search graph, it is possible to choose a very small value for cell size. After the grid is prepared, we begin analysing the terrain by first finding local extrema points, using Algorithm 1. A local extremum point is marked as local maxima if its height value is greater than all its neighbour points, and local minima if its height value is less than all its neighbour points. Neighbourhood is defined with respect to 4-connectivity, two points are neighbours if there are no points between them, and they reside either in the same row or the same column on the grid. All local extrema points found at this step are added to the set of vertices of the search graph.

If the terrain contains a lot of unevenness there might be too many local extrema points. Some of these points may be considered unnecessary for the search graph as they correspond to very small hills and cavities which are not very important features for the terrain. In order to eliminate such small extrema points, we use Algortithm unmarkSmallPoints. We iterate over the set of extrema points, and remove the points whose absolute height difference between its neighbours is less than a given threshold. The threshold should be chosen according to the application domain. It should be the height difference which the agents can pass over without

Algorithm 1 markExtremaPoints(*grid*)

1: **for all** p **in** *grid* **do**
2: **if** $\forall q \in \{N, S, E, W\}\ z_p > z_q$ **then**
3: $grid.markAsMaxima(p)$
4: **else if** $\forall q \in \{N, S, E, W\}\ z_p < z_q$ **then**
5: $grid.markAsMinima(p)$
6: **else if** $\forall q \in \{N, S\}\ \forall r \in \{E, W\}\ z_p < z_q \wedge z_p > z_r$ **then**
7: $grid.markAsSaddle(p)$
8: **else if** $\forall q \in \{N, S\}\ \forall r \in \{E, W\}\ z_p > z_q \wedge z_p < z_r$ **then**
9: $grid.markAsSaddle(p)$

difficulty.

Algorithm 2 unmarkSmallPoints(*grid*, *treshold*)

1: // $nearestMarked(a, b)$ gives the point that is closest to a in direction b
2: **for all** p **in** *grid* **such that** p is marked **do**
3: **if** $\forall direction \in \{N, S, E, W\}\ |z_{nearestMarked(p,direction)} - z_q| < k$ **then**
4: $grid.unmark(p)$

The set of extrema points correspond to points whose first derivative is zero. In mathematical sense, there might also be some points where the first derivative is not well-behaved. For example, at some points the first derivative might approach positive or negative infinity and existence of such points indicates discontinuities on the surface. In our case, these points correspond to points from which it is impossible to move to a neighbour, that is, the slope at that point is too big to ascend or descend. In order to find such points, we iterate over all points in the grid. If the height difference between a point and its neighbour going beyond agents capabilities, that point is marked as a singular point and added to the set of vertices of the search graph.

The next step is to mark the borders of obstacles on the terrain so that no edge will pass through an obstacle in the search graph. In our study, obstacles are represented as an ordered set of points. Obstacle borders are marked by drawing a line between each of the consecutive points of the obstacle. Since we are using a discrete grid, the line is rasterized using a standard line drawing algorithm such as Bresenham's. The standard version of Bresenham's algorithm draws 8-connected lines. Since we are using 4-connectivity, a slightly modified version that

will output 4-connected lines is used.

After the points on obstacle borders are marked, the desired start and goal points of the path are marked. This makes sure that start and goal points are always included in the search graph and a valid path will be found even if no other points were marked. In some terrain configurations, a point might be the only marked point in its row or column. Such a point is called an *alonepoint*. As we are using 4-connectivity during the construction of the search graph, alone points might cause the search graph to become disconnected. To prevent alone points, we mark additional points on the grid to connect alone points with the rest of the graph. The algorithm for preventing alone points works by iterating all the rows in the grid. When it encounters an alone point in a row, it marks the point that is on the same column with the encountered alone point and same row with the last marked point on the previous rows. This makes sure that the alone point is reachable from the previous rows by walking over the newly marked point.

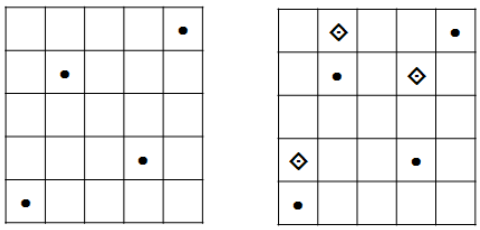

Figure 3.2: Grid points before and after marking alone points

It is enough to represent the most important features of the terrain using the steps taken so far. In other words, if we consider only the marked points up to this point, it is guaranteed to find a path from start to goal location, if there exists one. However, these marked points constitute to the bare minimum number of vertices for the search graph. As a result of this, the path obtained from this search graph may not be natural. In order to find more natural paths, we augment the search graph with additional vertices, by marking additional points between points that we marked in the previous steps. First, we define a relation called ”adjacency” such that two marked points, that are either on the same horizontal or vertical line are adjacent if there are no obstacles or marked points between them. For each adjacent point u and v, we draw an imaginary line from u to v, and set the last marked point to u. For each point w encountered along the line, if the absolute height difference between w and the last marked

point on the line is greater than or equal to a predefined value called ”height difference factor”, we mark w, and update the last marked point to w, so that the algorithm considers w for height different comparison at the next iteration. Choosing a reasonable value for height difference factor is largely dependent on the application domain. In our experiments, we found that choosing this value as half of the maximum slope agents can climb yielded the best results.

The algorithms for determining important points loop over each grid point a constant number of times, and require constant space per grid point. Therefore the overall algorithmic complexity of these algorithms is $O(N)$, where N denotes the total number of points on the grid. In the worst case of an unnaturally rough terrain, all the grid points will be marked as important and our representation degenerates to the regular grid, requiring $O(N)$ space. In the best case of a completely flat terrain, only the start and goal locations, and a point connecting these two will be marked, and this will require $O(1)$ space. In the average case, the space requirements will be between these two extremes, but it is important to note that our approach never performs worse than the grid approach.

3.4 Search Graph Construction

After the set of vertices are determined, the next step is to loop over the set one more time determine the edges of the search graph. The edges are determined as follows. For each vertex u of the search graph, we take the nearest vertex v in each of the four directions (North, South, East and West) and add an edge from u to v, if v is accessible from u. It is possible to consider more than four directions at this step, but since we took measures to keep the graph 4-connected, considering just these four directions is enough to make the graph connected as long as the underlying terrain is connected. The vertex v is accessible from u, if the terrain profile between these two points does not contain a slope that exceeds agents physical capabilities, and there is no obstacle between these two points.

To determine the edge cost, we divide the line connecting u and v to small line segments, each of which are unit length. The cost of an individual line segment is initially the euclidean distance between the two end points of the line. This value is then multiplied by a function that depends on the slope of the line segment and whether the slope is ascending or descending. This calculation is needed to make ascending a steep slope more expensive than descending a

moderate slope. The total cost of the edge depends on the sum of the costs of these line segments and whether the agents can maintain their formation along the edge. In order to decide if the agents can maintain their formation while moving from u to v, multiple approaches can be taken. The most accurate approach is placing the formation at the start point and simulating movement using the on-line planner. Since this step will be repeated for all the edges in the search graph, it will be computationally infeasible for even moderately sized search graphs. The second approach is treating the formation as a box and checking if the box fits at all the points along the edge. Although this approach is faster than the first one, it is still computationally costly. Thus, instead of considering all points on the edge, we place the formation at the start and end points of the edge, and check if the formation can fit at these points. This approach is not as accurate as the other two, but provides a nice trade-off between accuracy and speed.

The algorithm that checks whether the formation can be maintained at the start and end points of the edge is given in Algorithm 3. This algorithm first finds two points called $leftPoint$ and $rightPoint$ such that, $leftPoint$ and $rightPoint$ are the furthest points accessible from $startPoint$ along the line perpendicular to the edge. If the 2-D euclidean distance between these two points is less than formation width, the algorithm returns false. This return value indicates that the passage between $leftPoint$ and $rightPoint$ is too narrow for the formation to pass without shrinking. Since it is not desirable to break formation during moving, we impose an extra cost to this edge by multiplying the current edge cost with a function of the distance between $leftPoint$ and $rightPoint$ such that smaller the distance, higher the cost will be. If the return value is true, it means the formation can pass freely from this point so we don't need to impose any extra cost. It is important to note that, while finding $leftPoint$ it is enough to consider at most $formationWidth$ number of points to the left of $startPoint$ since the formation can pass freely through everywhere that is wider than the formation width and it will not expand beyond formation width. The same is true for $rightPoint$, therefore the total algorithmic complexity of this algorithm is $O(formationWidth)$.

We finish search graph construction after computing the costs of all edges. Since an individual edge can span all the points on a grid row or column, to compute all edge costs, we might loop over all the grid points once in horizontal and once in vertical direction. In addition to that, we will loop over $2 * formationWidth$ grid points to determine if the formation can pass through an individual edge. Therefore the total complexity of this step is $O(E * formationWidth + N)$

Algorithm 3 Can Formation Fit at Waypoint

Require: $\vec{p}$: waypoint, *direction*: direction of formation at $\vec{p}$, $formationWidth$, $formationDepth$

$\vec{rightNormal} \leftarrow \vec{unit}(d - \frac{\pi}{2})$

$\vec{rightPoint} \leftarrow \vec{p}$

for $i = 1$ **to** $formationWidth$ **do**

  if $accessible(\vec{p}, \vec{p} + i * \vec{rightNormal})$ **then**

    $\vec{rightPoint} \leftarrow point + i * \vec{rightNormal}$

  else

    break

$\vec{leftNormal} \leftarrow -\vec{rightNormal}$

$\vec{leftPoint} \leftarrow \vec{p}$

for $i = 1$ **to** $formationWidth$ **do**

  if $accessible(\vec{p}, \vec{p} + i * \vec{leftNormal})$ **then**

    $\vec{leftPoint} \leftarrow \vec{p} + i * \vec{leftNormal}$

  else

    break

$\vec{backNormal} \leftarrow \vec{unit}(d - \pi)$

$\vec{backPoint} \leftarrow \vec{p}$

for $i = 1$ **to** $formationDepth$ **do**

  if $accessible(\vec{p}, \vec{p} + i * \vec{backNormal})$ **then**

    $\vec{backPoint} \leftarrow \vec{p} + i * \vec{backNormal}$

  else

    break

return $\vec{leftPoint}, \vec{rightPoint}, \vec{backPoint}$

where E is the total number of edges and N is the total number of grid points.

3.5 Off-line Path Finding

After the search graph is constructed, the next task is to find the optimal path from start to goal location on this graph. For this task, we use the standard $A*$ search algorithm. $A*$ guarantees to find the shortest path on the graph and is optimal, as long as the heuristic function is admissible.

3D Euclidean distance is chosen to be the heuristic function for $A*$, because it gives a reasonable approximation on the actual distance and it is admissible. At the first step of computation of edge costs, we divided each edge into smaller segments and summed their lengths to obtain the final edge cost. So, the cost of an individual edge became equal to the length of the polygonal path connecting the end points of this edge. An edge's cost cannot be smaller than the length of straight line connecting its end points because of triangle inequality property. Therefore using Euclidean distance as the heuristic function never overestimates the actual distance, making it admissible. The extra cost assigned to an edge when the formation cannot fit it also reinforces this property.

3.6 Path Smoothing

$A*$ algorithm will return a path if there exists one. However, the path returned will be lacking diagonal components because of the 4-connected structure of the search graph. Due to the lack of diagonal components, agents will move in a staircase like pattern and this will ultimately result in an unnatural looking movement along the path. In order to deal with this shortcoming, a smoothing process should be applied to the path. Although this process is optional, it will make the path more natural and easier for the agents to navigate.

$A*$ returns an ordered set of vertices, containing at least 2 and at most N number of elements provided that the start and goal locations are different. At this step the aim is to choose a subset of these vertices such that, the path will still be valid and possible for the agents to navigate and the total cost of the path will not exceed the original cost too much. We obtain the smoothed path by looking at each vertex p_i in the original path and removing it from the

set there is a direct line-of-sight from its predecessor point p_{i-1} to its successor point p_{i+1}. And the cost of movement from v_{i-1} to v_{i+1} doesn't exceed the cost of movement from p_{i-1} to p_i and then to p_{i+1} by a margin of ϵ. This ϵ constant is determined by the user depending on the application domain. The steps of this algorithm is shown in Algorithm 4. The fragment that performs line-of-sight test is omitted.

Algorithm 4 Smooth Path

Require: p: list of path points, ϵ: maximum allowed increase in path cost

1: $i \leftarrow 1$
2: **while** $i < p.count - 1$ **do**
3: **if** $lineOfSight(p_{i-1}, p_{i+1})$ **and**
$cost(p_{i-1}, p_i) + cost(p_i, p_{i+1}) > cost(p_{i-1}, p_{i+1}) - \epsilon$ **then**
4: $p.remove(p_i)$
5: **else**
6: $i \leftarrow i + 1$

After the path is smoothed, the next step is to determine agent positions at each point in the path. Since this task requires information about the formation representation, it will be explained in the following chapter.

CHAPTER 4

MULTIRESOLUTION FORMATION PRESERVING PATH PLANNING

4.1 Formation Representation

Agents in a formation are organized as a hierarchy in which a formation can contain one or more agents or sub-formations. The most natural way to represent this hierarchy is treating the formation as a tree data structure. In this sense, intermediate nodes of the tree correspond to sub-formations and leaf nodes correspond to individual agents. A sample formation represented using this tree structure is illustrated in Figure 4.1. We label each node in the tree as *group* or *agent* depending on whether that node is a sub-formation or individual agent. If the formation contains only one agent then the root node of the tree is an *agent* node, otherwise the root node is a *group* node. Every node in the tree has a common set of properties, therefore most of the algorithms here can treat the nodes uniformly without depending on whether the node is a *group* or *agent*. One of the most important benefits this representation provides is that querying properties of the formation or assigning tasks to it can be done using simple depth-first traversals on the tree.

Properties of *agent* nodes can be computed immediately without considering other nodes in the tree. These properties either correspond to inherent features that agent has or dynamic information about the agent known at current time step. Inherent properties are supplied by the user. Most of the time these properties don't change during the agents lifetime. They are used to specify the agents physical capabilities, and its place in the hierarchy. Dynamic properties are computed on the fly. They are retain information like the agents current position or rotation.

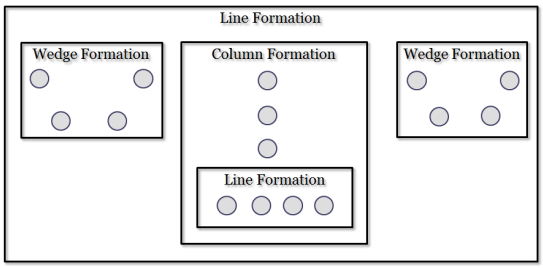

Figure 4.1: A Sample Multiresolution Formation

Properties of *group* nodes are computed by looking at the properties of their descendant nodes. For example, to compute the bounding box of a *group* node, all the bounding boxes of all its child nodes must be taken into account. Some of the properties of *group* nodes are aggregated from unchanging properties of their children, therefore these values are computed when they are needed for the first time and the computed values are cached so that subsequent queries for these properties can be completed immediately by returning the cached value.

A *group* node holds information about how the agents and other groups in it are organized as formation. The *group* keeps references to its children inside an ordered sequence, where each child is assigned a positive integer index denoting its place in the sequence. The leader of the group is the one with the lowest index.

4.1.1 Bounding Boxes

In our work, we compute bounding boxes to be able to tell how much the shape of the formation has changed at each time step. This information is used by agents when adjusting their speeds to compensate for the deformations. How much the formation deformed is also used in Chapter 5 as an evaluation metric.

In general sense, the bounding volume of a group of objects is the smallest volume that entirely contains the objects. Bounding volumes are utilized in computer graphics and computational geometry applications in order to make geometrical operations run more efficiently. Since there are faster algorithms for dealing with simple geometry rather than arbitrarily complex geometry, bounding volumes are usually chosen to be simpler than the real geometry of objects. Most of the geometrical operations first work on bounding volume geometry, consid-

ering the real geometry only when it is really required to.

There are several alternatives of bounding volume types, each with their own costs and benefits. The major types that we considered in this work are bounding spheres, axis aligned bounding boxes, oriented bounding boxes and convex hulls. Bounding spheres can be computed in linear time using the algorithm defined in [21] and once computed they don't change when the objects rotate. They are good when the objects are scattered evenly, but they become too large if the contained geometry is very tall or thin. Because of this, they are not ideal for representing column and line formations. Axis aligned bounding boxes are good for representing objects that are aligned with the world axes, but otherwise they suffer the same problem with bounding spheres. They can be computed in linear time by computing the minimum and maximum values of points for each axis. However, due to the fact that formations will usually change their orientation, they will rarely be aligned with the world axes and axis aligned bounding boxes will not be an ideal representation. Oriented bounding boxes can also be computed in linear time, with an algorithm similar to axis aligned bounding boxes. We found oriented bounding boxes consistent with the approach taken throughout the rest of the work, specifically with our formation representation, which are essentially oriented rectangles. At this point, although we considered convex hulls as an alternative, we found that the accuracy they provide doesn't compensate for the fact that they are computationally more expensive ($O(n \log n)$) than the other approaches.

In this work, we define a bounding box with four edges called front, rear, left and right. All the edges are right angled to each other. The front edge is the foremost edge with respect to the agents current movement direction (orientation). The edges are represented using four points: front-left, front-right, rear-left and rear-right. The edges and points of a sample bounding box are illustrated in Figure 4.2.

The algorithms for computing bounding boxes are different for individual agents and groups. Individual agents are circle shaped. The center of the circle overlaps with the position of the agent. For the bounding box, we return a square that has the same orientation with the agent, and completely encompasses the circle. Algorithm 5 takes the agents position, orientation and radius as input and returns the four points that define the bounding box.

In order to compute a groups bounding box, we use the method in Algorithm 6. The steps

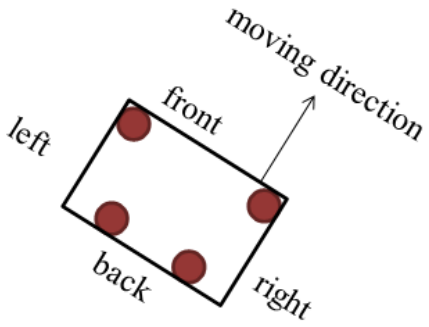

Figure 4.2: Bounding Box of a Formation

of this algorithm are as follows. First we, compute unit vectors that give the rear-to-front, and right-to-left directions of the bounding box using the orientation of the group. A group's orientation is found by taking the orientation of the leader agent in the group. After the unit vectors are computed, we loop over each child and take its bounding box. The points returned at this step are projected over the direction vectors. In order to find the extents of the bounding box, we update the minimum and maximum values along these axes as we loop. After the loop is finished and minimum and maximums are computed, the points that define the bounding box of the group are calculated using these values. For example, the front-left point is found by multiplying the maximum value on rear-to-front axis (*maxF*) with the rear-to-front unit vector (*dirF*) and summing this with the multiplication of *maxL* and *dirL*. Other three points of the bounding box are calculated in a similar manner.

Algorithm 5 Compute an Agent's Bounding Box

Require: ($\vec{p}$: position of agent, d: direction of agent, r: radius of agent

1: $\vec{frontLeft} \leftarrow \vec{p} + r * \sqrt{2} * \vec{unit}(d + \frac{\pi}{4})$

2: $\vec{frontRight} \leftarrow \vec{p} + r * \sqrt{2} * \vec{unit}(d - \frac{\pi}{4})$

3: $\vec{rearLeft} \leftarrow v + r * \sqrt{2} * \vec{unit}(d + \frac{3\pi}{4})$

4: $\vec{rearRight} \leftarrow \vec{p} + r * \sqrt{2} * \vec{unit}(d - \frac{3\pi}{4})$

5: **return** $(\vec{frontLeft}, \vec{frontRight}, \vec{rearLeft}, \vec{rearRight})$

Algorithm 6 Compute A Groups Bounding Box

Require: d: direction of group

$\vec{front} \leftarrow \vec{unit}(d)$

$\vec{left} \leftarrow \vec{unit}(d + \frac{\pi}{2})$

$min_{front}, min_{left} \leftarrow \infty$

$max_{front}, min_{left} \leftarrow -\infty$

for all c **in** $group$ **do**

  if *child* is a group **then**

    Compute bounding box of agent *child* into $fl,\ fr,\ rl,\ rr$

  else

    Compute bounding box of group *child* into $fl,\ fr,\ rl,\ rr$

  $min_{front} \leftarrow \min(min_{front}, \vec{fl} \cdot \vec{front}, \vec{fr} \cdot \vec{front}, \vec{rl} \cdot \vec{front}, \vec{rr} \cdot \vec{front})$

  $max_{front} \leftarrow \max(max_{front}, \vec{fl} \cdot \vec{front}, \vec{fr} \cdot \vec{front}, \vec{rl} \cdot \vec{front}, \vec{rr} \cdot \vec{front})$

  $min_{left} \leftarrow \min(min_{left}, \vec{fl} \cdot \vec{left}, \vec{fr} \cdot \vec{left}, \vec{rl} \cdot \vec{left}, \vec{rr} \cdot \vec{left})$

  $max_{left} \leftarrow \max(max_{left}, \vec{fl} \cdot \vec{left}, \vec{fr} \cdot \vec{left}, \vec{rl} \cdot \vec{left}, \vec{rr} \cdot \vec{left})$

$\vec{frontLeft} \leftarrow max_{front} \cdot \vec{front} + max_{left} \cdot \vec{left}$

$\vec{frontRight} \leftarrow max_{front} \cdot \vec{front} + min_{left} \cdot \vec{left}$

$\vec{rearLeft} \leftarrow min_{front} \cdot \vec{front} + max_{left} \cdot \vec{left}$

$\vec{rearRight} \leftarrow min_{front} \cdot \vec{front} + min_{left} \cdot \vec{left}$

return $\vec{frontLeft}, \vec{frontRight}, \vec{rearLeft}, \vec{rearRight}$

4.2 Referencing Schemes

It is natural to define a formation more than one way. In this thesis, we covered all three referencing schemes mentioned in [19] and [5] namely Unit-center scheme, Single-leader scheme and Neighbor scheme. The common idea is that each agent's ideal position is specified with respect to a reference point or reference agent. Ideal positions are specified in (*width*, *depth*) pairs. These values are encoded in a rotation invariant coordinate system in order to eliminate differences due to the rotation of the formation. The vector that is perpendicular to the groups moving direction pointing right becomes the *width* axis, and the vector that is parallel to the groups moving direction pointing backwards becomes the *depth* axis. This local coordinate system is illustrated in Figure 4.3.

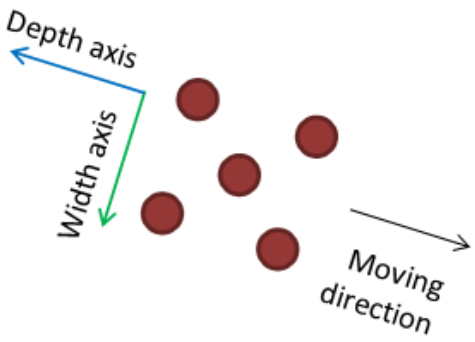

Figure 4.3: Width and Depth Coordinate System

- In Unit-center Scheme (Fig. 4.4a), the width and depth of agents are specified with respect to a central point, which may or may not belong to the interior of agent.
- In Single-leader Scheme (Fig. 4.4b), the width and depth of agents are specified with respect to a single agent in the group.
- In Neighbor Scheme (Fig. 4.4c), the width and depth of agents are specified with respect to agents with higher priorities.

Although the main idea of all three schemes is common, there are some slight differences especially in terms of the way $\Delta width$ and $\Delta depth$ of agents are specified. Figure 4.4 illustrates a sample wedge formation defined using all three schemes.

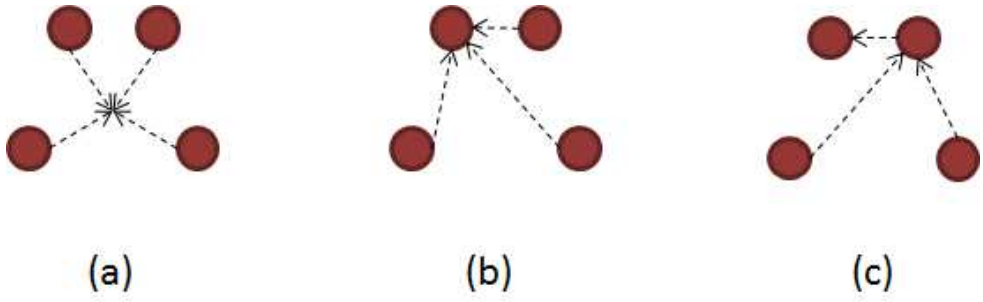

Figure 4.4: A Wedge Formation Defined Using Different Referencing Schemes

4.2.1 Unit-Center Scheme

In this scheme, there is no designated leader. The position and direction of the formation is the average of the position and direction of agents in it. That means, each agent has equal vote on the position and direction of the overall formation. When adding agents to the group, the rules that specify $\Delta width$ and $\Delta depth$ of agents are given with respect to the unit-center, which is initially at $(0, 0)$. Since the unit-center is the center-of-mass of the formation, it moves while the agents are added.

Instead of computing the center-of-mass after adding each agent, we reset the unit-center back to $(0, 0)$ after all the agents are added to the group. This algorithm finds the average $\Delta width$ and $\Delta depth$ in the group. Then subtracts this averages from agents original $\Delta width$ and $\Delta depth$ values to move the unit center back to $(0, 0)$

In order to determine each agents ideal width and depths, Algorithm 7 is used. This algorithm iterates over all the rules to determine the center-of-mass of the formation for the ideal case. Center-of-mass is found by averaging all the width and depth values in the rule set. Afterwards, the widths and depths of agents in the group are adjusted according to the rules and the center-of-mass obtained in the previous step.

4.2.2 Single-leader and Neighbor Schemes

Single-leader and Neighbor schemes are covered in one section because of the fact that, Single-leader is a special case of Neighbor scheme. In this scheme, the information about

Algorithm 7 Determine Widths and Depths of Agents For Unit-center Scheme

Require: *rules*: list of rules, *group*: group containing agents

1: $w_{avg} \leftarrow 0$
2: $d_{avg} \leftarrow 0$
3: **for all** $(agent, \Delta width, \Delta depth)$ **in** $rules$ **do**
4: $\quad w_{avg} \leftarrow w_{avg} + \Delta width$
5: $\quad d_{avg} \leftarrow d_{avg} + \Delta depth$
6: $w_{avg} \leftarrow w_{avg}/rules.count$
7: $d_{avg} \leftarrow d_{avg}/rules.count$
8: **for all** *agent* in $group$ **do**
9: $\quad w_{agent} \leftarrow \Delta width - w_{avg}$
10: $\quad d_{agent} \leftarrow \Delta depth - d_{avg}$

each agents position relative to other agents in the formation is encoded as a set of rules. Each rule is a quadruple consisting of (*lead*, $follow$, $\Delta width$, $\Delta depth$) components. A rule defines a distance based relation between two agents, that are called *lead* and $follow$. In order to maintain the formation, the $follow$ agent adjusts its position with respect to the position of the *lead* agent. The relative position of $follow$ is specified in $\Delta width$ and $\Delta depth$ components.

In order to determine each agents ideal position in a group, Algorithm 8 is used. This algorithm iterates over all rules to determine ideal width and depth values for all child agents contained in the group. It begins by first initializing width and depth of the first agent to zero. Then for each rule, it computes the width and depth of the $follow$ agent by adding the width and depths of *lead* agent to $\Delta width$ and $\Delta depth$ values specified in the rule. If the newly computed values are different than $follow$ agent's current values, it updates them to new values and sets the *changed* flag to true. This flag is used in the outer loop to check when to stop iterating over rules. In other words, the algorithm ends when agents don't change their *width*, and $depth$ values anymore. If the rules are ordered, i.e., that an agent other than the group leader doesn't appear in a *lead* position before it appears in a $follow$ position, the algorithm completes in just one pass. Otherwise it will take multiple passes to place every agent in their correct position. In order to eliminate circular references, we restrict that the index of *lead* is always less than the index of $follow$.

Since the rules will not change throughout a single run, it is sufficient to call Algorithm 8

Algorithm 8 Determine Widths and Depths of Agents For Single-Leader or Neighbor Schemes

Require: *rules*: list of rules, *group*: group containing agents

1: $w_0 \leftarrow 0$
2: $d_0 \leftarrow 0$
3: **repeat**
4: $changed \leftarrow false$
5: **for all** $(lead, follow, \Delta width, \Delta depth)$ **in** *rules* **do**
6: $newWidth \leftarrow w_{lead} + \Delta width$
7: $newDepth \leftarrow d_{lead} + \Delta depth$
8: **if** $newWidth \neq w_{follow}$ **or** $newDepth \neq d_{follow}$ **then**
9: $w_{follow} \leftarrow newWidth$
10: $d_{follow} \leftarrow newDepth$
11: $changed \leftarrow true$
12: **until not** *changed*
13: compute minimum and maximum of widths of agents into w_{min} and w_{max}
14: **for all** *agent* in *group* **do**
15: $w_{agent} \leftarrow w_{agent} - (w_{max} + w_{min})/2$

once. Afterwards, the ideal width and depth values of all agents, and the total width and depth of the formation will be stored into fields of the nodes inside the tree.

4.3 Determining Waypoints for Each Agent

The formation navigates from start to goal location, passing over each path point along the way. It starts motion from p_0, moving in a straight line to p_1, and then changes its movement direction to p_2 when it reaches p_1. This goes on until the formation reaches the final point on the path, which is the goal location. This scenario holds if the formation is treated as a single entity or it contains just one agent. In all but the trivial cases, the formation will be a composite entity that includes multiple agents and sub-groups. When the formation is placed on a path point, each agent will be placed at a different position, depending on its place in the formation. This arises the need to find each agents position for each point on the path. The position of an agent when the formation is placed on a path point is called a waypoint.

Every agent has to have a waypoint for each path point on the path. An agent navigates using its own list of waypoints. It starts motion from the first waypoint and moves to the next waypoint until the final waypoint is reached. After all agents reach their final waypoints, the formation has reached the goal location.

We use Algorithm 9 to determine each agents waypoint in a group. Since an agent might end up in an inaccessible position if we were to put it directly according to its place in the formation, we need to determine a valid position by scaling the width and depth of the group as necessary. This is done by using Algorithm 3 to find the furthest accessible points from p. After that, the group is scaled in width axis so that it can fit between $leftPoint$ and $rightPoint$ returned from Algorithm 3. The group is scaled in the depth axis in a similar manner. After the group is scaled, positions of each agent is computed. We compute two points a and b for each agent. a corresponds to the agents ideal position while passing the waypoint, and b corresponds to the agents position if the group was moving in a column formation. If an agent passing from point a cannot reach p, then the agents waypoint is chosen as b. In other words, if the terrain is too rough or there are obstacles limiting the agents motion from a to p, then the agent should move as close as possible to the original path, as if it was passing a very narrow passage. Otherwise, the agents waypoint is chosen as a. The group calls this

algorithm recursively for each group it encounters in the child list in order to propagate this waypoint to its descendants.

Algorithm 9 Add Waypoint

Require: *group*: the group, $\vec{w}$: waypoint, $\vec{p}$: actual path point, d: direction

1: Compute $\vec{left}$, $\vec{right}$ and $\vec{back}$ using Algorithm 3.
2: $scale_w \leftarrow \min(1, |\vec{left} - \vec{right}|)$
3: $scale_d \leftarrow \min(1, |\vec{w} - \vec{back}|)$
4: $\vec{mid} \leftarrow$ mid point of $\vec{left}$ and $\vec{right}$
5: $\vec{dir}_w \leftarrow scale_w * \vec{unit}(d - \frac{\pi}{2})$
6: $\vec{dir}_d \leftarrow scale_d * \vec{unit}(d - \pi)$
7: **for all** *agent* in *group* **do**
8: $\quad \vec{a} \leftarrow \vec{mid} + w_a * \vec{dir}_w + d_a * \vec{dir}_d$
9: $\quad \vec{b} \leftarrow \vec{mid} + d_a * \vec{dir}_d$
10: $\quad$ **if not** *accessible*$(\vec{a}, \vec{p}$ **then**
11: $\quad\quad \vec{a} \leftarrow \vec{b}$
12: $\quad$ **if** *agent* is a group **then**
13: $\quad\quad$ call Add Waypoint with $\vec{a}$, $\vec{p}$ and d
14: $\quad$ **else**
15: $\quad\quad$ add a to *agent*'s waypoint list

4.4 Moving Agents Between Waypoints

Moving agents between waypoints involves a few steps. First of all, the new position of an agent must be computed using its current position, current velocity and current direction. If there is no obstacle or other agent in this direction, the agent updates its position, and then checks to see if it has reached its destination, i.e. the next waypoint. If that is the case, the agent chooses the next waypoint as target and updates its direction to face the new target. If there is no next waypoint, then the agent has reached its goal and stops.

The main difference between approaches for moving agents between waypoints is how they handle targets and obstacles. For example, in [17] target waypoints are modelled as potential fields that attract the agents, and obstacles and other agents are modelled as potential fields that repel the agents. The strength of the potential field depends on the distance, so immediate

Algorithm 10 Move Agent Between Waypoints

Require: *state*: current state, $\vec{p}$: current position, *s*: current speed, *dt*: time step, $\vec{prev}$: previous waypoint, $\vec{next}$: next waypoint

while $\vec{p}$ is very close to $\vec{next}$ **do**
  if $\vec{next}$ was the last waypoint **then**
    $state \leftarrow 4$
    return
  update $\vec{next}$ to next waypoint
$d \leftarrow$ the direction of the vector from $\vec{p}$ to $\vec{next}$
if $state = 1$ **then**
  if $tryToMove(\vec{p}, d, s, dt)$ **or** $tryToMove(\vec{p}, d + \frac{\pi}{4}, s, dt)$ **or** $tryToMove(\vec{p}, d - \frac{\pi}{4}, s, dt)$ **or** $tryToMove(\vec{p}, d + \frac{\pi}{2}, s, dt)$ **or** $tryToMove(\vec{p}, d - \frac{\pi}{2}, s, dt)$ **then**
    return
  $state \leftarrow 2$
if $state = 2$ **then**
  $side \leftarrow 1$ **if** $\vec{p}$ is on the right of the line from $\vec{prev}$ to $\vec{next}$ **else** -1
  if $tryToMove(\vec{p}, d, s, dt)$ **or** $tryToMove(\vec{p}, d + side * \frac{\pi}{4}, s, dt)$ **then**
    $state \leftarrow 1$
    return
  else if $tryToMove(\vec{p}, d + side * \frac{\pi}{2}, s, dt)$ **or** $tryToMove(\vec{p}, d + side * \frac{3\pi}{4}, s, dt)$ **then**
    return
  $state \leftarrow 3$
  $wait \leftarrow$ random amount of time
if $state = 3$ **then**
  $wait = wait - dt$
  if $wait \leq 0$ **then**
    $state \leftarrow 2$
    return
if $state = 4$ **then**
  return

Algorithm 11 Try to Move

Require: $\vec{p}$: current position, d: desired direction, s: current speed, dt: time step

1: $\vec{p'} \leftarrow \vec{p} + \vec{unit}(d) * s * dt$
2: $slope \leftarrow$ the slope of line segment from $\vec{p}$ to $\vec{p'}$
3: $\vec{p'} \leftarrow \vec{p} + \vec{unit}(d) * s * dt * \cos(slope)$
4: **if not** $accessible(\vec{p}, \vec{p'})$ **then**
5: **return false**
6: **else if** another agent occupies $\vec{p'}$ **then**
7: **return false**
8: **else**
9: update agents current position to $\vec{p'}$
10: update agents current direction to d
11: **return true**

obstacles apply a larger repelling force.

In our approach we used a state-machine based algorithm. According to this, the behavior of the agent is determined by its current state. We call the direction from an agents current position to its next waypoint the ”front” direction, and state changes are mostly governed whether the agent can move in the front direction or not.

Agents are in the first state most of the time. In this state, the agent first tries to move in the front direction. If it cannot move due to some obstacle or terrain roughness, it tries to move in the front-left, and front-right, left and right directions. If the agent cannot move in either of these directions, it transitions to second state.

Agents in the second state, first try to move in the front direction. If it fails, they try to get to the line connecting the previous and next waypoints as close as possible. They do this by first determining which side of the line they are, and then trying to move perpendicular to the front direction from their side. If that is not possible, they try to move in the rear-left or rear-right directions as a last resort. If that fails too, then the agents transition to third state.

Agents in the third state wait for a random amount of time and then transition to second state again. This allows the agents to wait for temporary obstructions to the path created by other agents. If the path was blocked by another agent, it will most likely be cleared after the

waiting time. The randomness also ensures that two agents moving head to head will not get in a deadlock situation by entering and leaving the third state at the same time.

The fourth and final state is the final state. Agents transition to this state when they reach their goal and stop moving. The algorithm for moving agents between waypoints is given in Algorithm 10. The algorithm that checks to see if the agent can move from its current position with a given angle, and updates the position and direction of the agent if the move is possible is shown in Algorithm 11.

4.5 Tuning Speeds

Since each agent will travel along its own set of waypoints, the distance to be travelled by each agent is different. For example, when the formation is making a turn, the agents on the inner side of the turn will travel less than the ones on the outer side. If the agents in the formation never changed their speeds, preserving the shape of the formation would be impossible since agents that travel shorter distances will quickly pass the ones that travel longer distances.

In order to keep the relative depths of agents in a group close to their ideal depths, groups call Algorithm 12 in regular intervals. Each group is responsible for tuning the speeds of its own child agents. The algorithm will assign one of three actions to each agent, that are slow down, speed up and normalize speed. After the algorithm finishes, the speed of agents will be adjusted depending on the action assigned to each of them.

This algorithm computes a depth vector, and relative depth of each agent depending on the formation referencing scheme used.

- If Unit-center scheme is used, depth vector is computed by taking the inverse of the groups moving direction (which is the average of all agents directions in the group), then the relative depth is computed by projecting the difference between the agents current position and groups center-of-mass onto this depth vector.
- If Single-leader or Neighbor scheme is used, depth vector is computed by taking the inverse of the direction of the agent, that current agent is following. After that, relative depth is computed similarly, but taking the difference between the agents current position and the leading agents current position.

If the agent is too close or too far to its ideal depth, it is assigned an action to make its depth approach its ideal depth in the following time steps. In order to eliminate fluctuations in speed especially when passing a rough terrain, we tolerate a certain percentage of error in depth. The speed of an agent is normalized if its error in depth does not exceed this percentage. In this context, normalizing speed of an agent means making its speed approach its designated average speed.

Algorithm 12 Tune Speeds

Require: *group*: The group, *p*: the percentage up to which errors in depth are tolerated

1: **for all** *agent* in *group* **do**
2: $action \leftarrow normalizeSpeed$
3: **if** the formation is defined using unit-center scheme **then**
4: $\vec{d} \leftarrow \vec{unit}(group.dir - \pi)$
5: $depth \leftarrow (\vec{agent}.pos - \vec{group}.pos) \cdot \vec{d}$
6: **else**
7: $\vec{d} \leftarrow \vec{unit}(leader(agent).dir - \pi)$
8: $depth \leftarrow (\vec{agent}.pos - leader(\vec{agent}).pos\ cdot\vec{d}$
9: **if** $agent.idealDepth < 0$ **then**
10: **if** $depth > agent.idealDepth * (1 - p)$ **then**
11: $action \leftarrow speedUp$
12: **else if** $depth < agent.idealDepth * (1 + p)$ **then**
13: $action \leftarrow slowDown$
14: **else**
15: **if** $depth < agent.idealDepth * (1 - p)$ **then**
16: $action \leftarrow slowDown$
17: **else if** $depth > agent.idealDepth * (1 + p)$ **then**
18: $action \leftarrow speedUp$

CHAPTER 5

EXPERIMENTAL EVALUATION AND SAMPLE RUNS

5.1 Test Environment

In order to conduct experiments and and evaluate the effectiveness of the methods proposed in this thesis, a software implementation was realized. Because of its widespread use in simulation software, the implementation language is chosen as C++. The software is separated to three modules, the first one being the simulation itself, the second one is the data collection module that records statistical information required for evaluation and the third one is the visualization module that allows the effectiveness of the algorithms to be inspected real-time. For the visualization module, OpenGL and GLUT cross-platform graphics libraries were used.

The software is compiled using GCC 4.5.2 running under MinGW32. The experiments are run and the timings are recorded on a 2.27 GHz Intel i5 M430 notebook PC with 4 GB of physical memory. The operating system is Windows 7 Home Premium 64 bit. Although the platform is Windows in this setup, the implementation is cross-platform and the same experiments were also conducted on 64 bit Ubuntu Linux with similar results.

5.2 Data Set

The data sets used throughout the experiments is similar to the one in [6]. For the terrain data set, we generated three $512 * 512$, three $1024 * 1024$ and three $2048 * 2048$ height-maps. Among the generated terrain data, we also used one real world terrain data which is also $2048 * 2048$.

We used a subdivision based algorithm to generate height maps. This algorithm subdivides a coarse height-map, introducing a controllable level of randomness to the terrain data at each iteration. In [22], a height-map is recursively divided into four squares at each step. A point's elevation is computed by averaging the elevation of points at its corners in a square shape and adding or subtracting a random value [23]. This step is repeated, but this time considering the points that are at the corners of a diamond shape instead of a square. After that, the operation is repeated recursively for four squares obtained by subdividing in a quad-tree like manner. The degree of randomness is reduced at each subdivision, hence there is less variation in elevation of features that are spatially close than features that are far. As this property preserves local similarities it adds to the realism of generated height-maps.

In this chapter, performance of the on-line and off-line planners are evaluated using two separate set of experiments. Experimental results and their evaluation will be covered in the following two sections.

5.3 Off-line Planner

In this section, performance of the off-line planner is explained. For this set of experiments, we first constructed a search graph from each of the terrains. The behavior of the algorithms that determine vertices of the search graph depend on threshold values. In the following tables these values are labeled as,

$h1$ decides whether an extrema point is an insignificant terrain feature to be included in the graph.

$h2$ decides whether a point is singular (not accessible from one of its neighbors).

$h3$ the maximum allowed elevation difference between adjacent points.

We run these experiments for different values of $h1$, $h2$ and $h3$, and once without them. That is, the effect of algorithms that make use of these values is disabled. For comparison, we also included the grid method, which includes all the points in the search graph. After the graph is constructed, we ordered to search a path starting at the south-west corner and ending at the north-east corner of the terrain.

In these experiments, we recorded the total number of vertices included in the search graph, the time required to construct the search graph, the time required to find the shortest path, the number of nodes expanded during search, the cost of the path found, and the cost of the path after the path smoothing algorithm is applied.

The results of these experiments indicate that, the proposed terrain analysis methods significantly reduce the complexity of the search graph. The actual number of vertices depends on the threshold values. By experimenting with different values for h_1, h_2 and h_3, we found that best results are obtained when $h3 = 2h_2 = 4h_1$.

The number of vertices decreases with increasing values of h values. The bare minimum number of vertices required to construct a complete graph can be obtained when the algorithms that use these thresholds are disabled. In these minimal graphs, the number of vertices is only 1% of the original graph.

Graph construction time increases with decreasing values of h values. As explained earlier, when smaller thresholds are used more points are included in the search graph, hence computing the costs of edges that are incident with these points introduce some overhead. It is important to note that, graph construction time is a one-time operation. Subsequent searches on the same graph will use the existing data. Therefore, the time spent at this phase will be amortized when the graph is used for many searches. Also, search graphs can also be serialized to a storage medium, and loaded when required. This property can be useful in computer games and simulations, where the terrain data is mostly static and lots of searches will be performed during the execution of the program.

The time spent for A* depends on the complexity of the graph. In our experiments we used a mostly non-optimized implementation of this algorithm to better illustrate how the number of vertices in the graph affects search performance. The performance penalty for search quickly becomes infeasible for a grid method, even for moderately sized terrains. This points out the necessity of terrain analysis techniques such as the one used in this thesis. Especially in experiments when the thresholds are not used, searches are almost instant.

The cost of the path is an indicator of the quality of the path. To make a comparison, the costs returned from the grid search is useful because of the fact that they correspond to the real distances the agents will travel by following the path. In accordance with this property,

the quality of the path increases with increasing number of points included in the graph.

As a result of these experiments we observed that the algorithms are able to construct graphs that are very close to ideal (grid method). When appropriate values for thresholds are supplied, the quality of paths found by running the search these graphs are no worse than 10% of the ideal case. The major advantage is that this quality is achieved without sacrificing search performance.

Table 5.1: First $512 * 512$ Terrain

Thresholds (h_1, h_2, h_3)	1, 3, 6	1, 6, 12	None	Grid
Number of Vertices	14864	7344	3583	262144
Graph Construction Time	0.05 sec	0.04 sec	0.01 sec	0.23 sec
A* Search Time	0.28 sec	0.07 sec	0.005 sec	8.79 sec
Expanded Nodes	12139	6372	310	226407
Path Cost	1785.17	2005.58	2976.37	1568.11
Smoothed Path Cost	1544.71	1564.12	2916.85	1390.30

Table 5.2: Second $512 * 512$ Terrain

Thresholds (h_1, h_2, h_3)	1, 2, 4	1, 3, 8	None	Grid
Number of Vertices	53844	32963	13111	262144
Graph Construction Time	0.09 sec	0.07 sec	0.04 sec	0.22 sec
A* Search Time	1.55 sec	0.85 sec	0.13 sec	5.13 sec
Expanded Nodes	37197	23062	11095	149195
Path Cost	1728.88	1792.81	2635.27	1549.78
Smoothed Path Cost	1543.73	1702.26	2513.83	1490.17

Table 5.3: Third $512 * 512$ Terrain

Thresholds (h_1, h_2, h_3)	1, 3, 8	1, 4, 12	None	Grid
Number of Vertices	24339	16948	6143	262144
Graph Construction Time	0.06 sec	0.05 sec	0.02 sec	0.23 sec
A* Search Time	0.52 sec	0.2 sec	0.01 sec	4.78 sec
Expanded Nodes	16700	9585	2846	151131
Path Cost	2035.38	2087.51	3386.9	1743.5
Smoothed Path Cost	1663.83	1833.38	3315.2	1573.88

Table 5.4: First $1024 * 1024$ Terrain

Thresholds (h_1, h_2, h_3)	1, 1, 2	1, 2, 4	None	Grid
Number of Vertices	16152	3945	418	1048576
Graph Construction Time	0.1 sec	0.05 sec	0.02 sec	1.06 sec
A* Search Time	0.2 sec	0.03 sec	0.02 sec	65.17 sec
Expanded Nodes	16135	3948	418	986374
Path Cost	2904.72	3003.18	3990	2639.9
Smoothed Path Cost	2681.2	2914.9	3467.01	2677.44

Table 5.5: Second $1024 * 1024$ Terrain

Thresholds (h_1, h_2, h_3)	2, 2, 4	4, 4, 8	None	Grid
Number of Vertices	68721	32208	7843	1048576
Graph Construction Time	0.25 sec	0.19 sec	0.05 sec	1.07 sec
A* Search Time	1.46 sec	0.44 sec	0.02 sec	73.17 sec
Expanded Nodes	32372	16296	3064	750097
Path Cost	2851.11	3012.3	4078.95	2645.1
Smoothed Path Cost	2641.47	2940.25	3472.62	2675.29

Table 5.6: Third $1024 * 1024$ Terrain

Thresholds (h_1, h_2, h_3)	3, 4, 6	4, 6, 12	None	Grid
Number of Vertices	33878	11689	2812	1048576
Graph Construction Time	0.20 sec	0.12 sec	0.03 sec	1.12 sec
A* Search Time	1.3 sec	0.2 sec	0.02 sec	76.95 sec
Expanded Nodes	30757	11041	1243	816162
Path Cost	3167.86	3567.11	7717.36	2826.46
Smoothed Path Cost	2868.29	3310.61	7712.36	2702.2

Table 5.7: First 2048 $*$ 2048 Terrain

Thresholds (h_1, h_2, h_3)	4, 8, 16	5, 10, 20	None	Grid
Number of Vertices	81865	61134	7505	4194304
Graph Construction Time	1.271 sec	1.215 sec	0.792 sec	3.431 sec
A* Search Time	9.832 sec	5.874 sec	0.121 sec	317.58 sec
Expanded Nodes	70938	53140	4368	2066343
Path Cost	5653.5	5752.08	7423.2	4667.28
Smoothed Path Cost	5076.47	5569.22	6282.79	4692.24

Table 5.8: Second 2048 $*$ 2048 Terrain

Thresholds (h_1, h_2, h_3)	4, 7, 14	4, 8, 16	None	Grid
Number of Vertices	101405	65046	9869	4194304
Graph Construction Time	1.357 sec	1.192 sec	0.808 sec	3.413 sec
A* Search Time	17.244 sec	8.698 sec	0.324 sec	650.821 sec
Expanded Nodes	98589	64171	7633	3301482
Path Cost	5855.92	6053.01	6805.94	5074.07
Smoothed Path Cost	5282.83	5879.8	6391.19	5105.84

Table 5.9: Third 2048 $*$ 2048 Terrain

Thresholds (h_1, h_2, h_3)	4, 7, 14	5, 10, 20	None	Grid
Number of Vertices	105210	67325	4875	4194304
Graph Construction Time	1.395 sec	1.308 sec	0.735 sec	3.581 sec
A* Search Time	18.329 sec	10.416 sec	0.094 sec	735.403 sec
Expanded Nodes	104581	67200	1521	3492519
Path Cost	6383.5	6505.37	8462.43	5421.58
Smoothed Path Cost	5928.24	6504.71	7027.77	5449.66

Table 5.10: Real World 2048 $*$ 2048 Terrain

Thresholds (h_1, h_2, h_3)	4, 7, 14	4, 8, 16	None	Grid
Number of Vertices	59346	31559	12158	4194304
Graph Construction Time	1.292 sec	1.213 sec	1.030 sec	3.296 sec
A* Search Time	16.262 sec	4.530 sec	0.096 sec	413.966 sec
Expanded Nodes	51899	25226	742	1366792
Path Cost	5384.54	5619.49	7443	4347.81
Smoothed Path Cost	5063.43	5451.9	6152.68	4419.54

5.4 On-line Planner

For the evaluation of the On-line Planner, we designed four formations and tested each of them using the four formation definition schemes.

The first scheme is called Unit-center. In this scheme, agents adjust their alignments with respect to the center-of-mass of the formation.

In the second test, the formation is defined using the Single-leader scheme. In this scheme, agents adjust their alignments with respect to the pre-defined leader in the group. The agent that is on the front-most of the formation is chosen as the leader.

In the third test, the formation is defined using the Neighbor scheme. In this scheme, the relative width and depth of each agent is specified with respect to its closest neighbor.

The formations we used are illustrated in Figures 5.1, 5.2, 5.3 and 5.4. Their corresponding evaluation results are illustrated in Tables 5.11, 5.12, 5.13 and 5.14. To interpret the results, we used similar evaluation metrics as [18], [19] and [5]. At each time step during the simulation, we recorded the positions, orientations and speeds of the agents. We took the ratio of path traveled by the agent to the original path. If the path is relatively straight, and the formation could be maintained successfully, this value should be close to 1. For the single leader and neighbor formation schemes, the path ratio is displayed in two columns for each agent and its leader.

In addition to the path ratio, we computed the normalized error value in $\Delta width$ and $\Delta height$ values. This value indicates how much the shape of the formation was modified during its motion, averaged for each time step. As a direct consequence of these metrics, we also compute the percent of time an agent stays out of formation. We chose to tolerate errors that are within 10% of correct value. Since the value of this threshold is in accordance with [19], the results of these two works are comparable to each other.

Apart from the metrics defined by the earlier works, we also considered the velocity of the formation as an evaluation metric, as it is desirable for the agents to maintain a designated velocity.

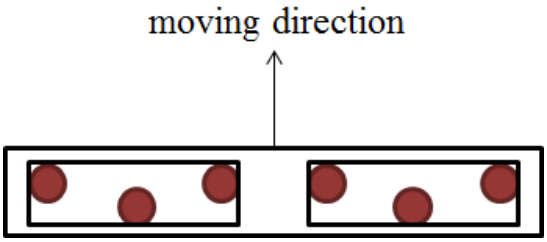

Figure 5.1: First Formation

Table 5.11: Evaluation Results for First Formation

Unit-center Scheme	Time out of formation (width)	6.957%
	Time out of formation (depth)	15.507%
	Average error in $\Delta width$	0.053 (σ = 0.099)
	Average error in $\Delta depth$	0.299 (σ = 0.845)
	Average speed	0.939 (σ = 0.315)
	Distance travelled	238.947
	Path ratio	1.105
Single-leader Scheme	Time out of formation (width)	24.633%
	Time out of formation (depth)	16.259%
	Average error in $\Delta width$	0.080 (σ = 0.089)
	Average error in $\Delta depth$	0.153 (σ = 0.189)
	Average speed	0.997 (σ = 0.290)
	Distance travelled	238.919
	Path ratio	1.104
Neighbor Scheme	Time out of formation (width)	22.166%
	Time out of formation (depth)	18.788%
	Average error in $\Delta width$	0.094 (σ = 0.113)
	Average error in $\Delta depth$	0.125 (σ = 0.149)
	Average speed	0.996 (σ = 0.300)
	Distance travelled	238.905
	Path ratio	1.104

The test results indicate that, all three formation schemes are able to maintain the formation most of the time. Among the three formation schemes, the Unit-center scheme is able to preserve formation better than the other two. This is because of the fact that each agent has equal vote on the ideal position of the formation and therefore an error made by one agent will be corrected by the others. Additionally, the errors in $\Delta width$ and $\Delta depth$ values of one agent will not propagate to other agents by means of leader-follower relationship.

One disadvantage of unit-center scheme is that it is sometimes too strict on formation main-

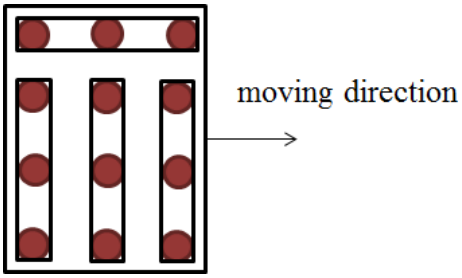

Figure 5.2: Second Formation

Table 5.12: Evaluation Results for Second Formation

Unit-center Scheme	Time out of formation (width)	0.000%
	Time out of formation (depth)	11.812%
	Average error in $\Delta width$	0.000 (σ = 0.000)
	Average error in $\Delta depth$	0.535 (σ = 2.114)
	Average speed	0.954 (σ = 0.283)
	Distance travelled	239.166
	Path ratio	1.084
Single-leader Scheme	Time out of formation (width)	0.000%
	Time out of formation (depth)	18.239%
	Average error in $\Delta width$	0.000 (σ = 0.000)
	Average error in $\Delta depth$	0.111 (σ = 0.178)
	Average speed	0.935 (σ = 0.365)
	Distance travelled	239.215
	Path ratio	1.085
Neighbor Scheme	Time out of formation (width)	0.000%
	Time out of formation (depth)	9.090%
	Average error in $\Delta width$	0.000 (σ = 0.000)
	Average error in $\Delta depth$	0.068 (σ = 0.140)
	Average speed	0.899 (σ = 0.195)
	Distance travelled	269.121
	Path ratio	1.220

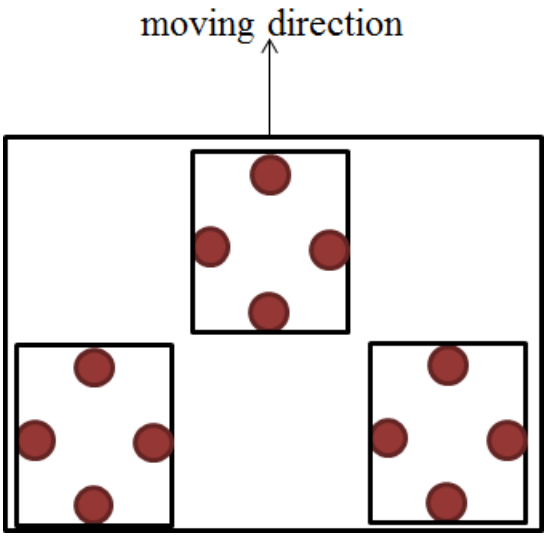

Figure 5.3: Third Formation

Table 5.13: Evaluation Results for Third Formation

Unit-center Scheme	Time out of formation (width)	2.053%
	Time out of formation (depth)	2.223%
	Average error in $\Delta width$	0.023 (σ = 0.044)
	Average error in $\Delta depth$	0.022 (σ = 0.055)
	Average speed	0.724 (σ = 0.199)
	Distance travelled	179.328
	Path ratio	1.118
Single-leader Scheme	Time out of formation (width)	12.289%
	Time out of formation (depth)	9.657%
	Average error in $\Delta width$	0.047 (σ = 0.054)
	Average error in $\Delta depth$	0.096 (σ = 0.152)
	Average speed	0.759 (σ = 0.160)
	Distance travelled	179.523
	Path ratio	1.119
Neighbor Scheme	Time out of formation (width)	17.039%
	Time out of formation (depth)	10.277%
	Average error in $\Delta width$	0.074 (σ = 0.087)
	Average error in $\Delta depth$	0.098 (σ = 0.156)
	Average speed	0.760 (σ = 0.186)
	Distance travelled	179.767
	Path ratio	1.121

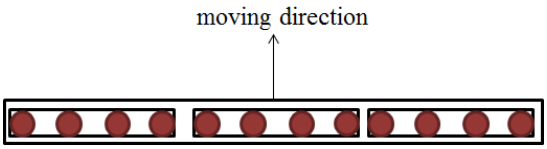

Figure 5.4: Fourth Formation

Table 5.14: Evaluation Results for Fourth Formation

Unit-center Scheme	Time out of formation (width)	5.282%
	Time out of formation (depth)	0.000%
	Average error in $\Delta width$	0.051 (σ = 0.093)
	Average error in $\Delta depth$	0.000 (σ = 0.000)
	Average speed	0.561 (σ = 0.285)
	Distance travelled	180.690
	Path ratio	1.054
Single-leader Scheme	Time out of formation (width)	16.904%
	Time out of formation (depth)	0.000%
	Average error in $\Delta width$	0.059 (σ = 0.067)
	Average error in $\Delta depth$	0.000 (σ = 0.000)
	Average speed	0.746 (σ = 0.218)
	Distance travelled	180.162
	Path ratio	1.053
Neighbor Scheme	Time out of formation (width)	17.925%
	Time out of formation (depth)	0.000%
	Average error in $\Delta width$	0.061 (σ = 0.069)
	Average error in $\Delta depth$	0.000 (σ = 0.000)
	Average speed	0.746 (σ = 0.226)
	Distance travelled	180.100
	Path ratio	1.051

tenance. Formations designed using this scheme detect errors in $\Delta width$ and $\Delta depth$ more often than the other two. This has two consequences. First, the agents travel longer distances to reach their ideal positions, and second agents change their speeds frequently to restore the ideal alignment. Although this is a desired behavior, very frequent changes in speed results in oscillations in the positions of agents. This disadvantage can be mostly overcome by using a larger tolerance in the speed tuning algorithm.

Another observation is that, in this scheme, the average speed of the agents were the lowest among three schemes, especially if the *width* of the formation is fairly greater than its *depth* resembling a line formation. On the other hand, the average speed is very close to the ideal speed of 1 if the formation has a *depth* close to its *width*.

We found out that, the performance of Single-leader scheme is very similar to Unit-center. This is because of the fact that in both of these schemes, agents align themselves according to a single point, which is either the position of the leader agent or the center-of-mass. Unit-center scheme is less strict on formation maintenance, so the average speed is generally better, and the distance travelled by the agents is less. The performance of Single-leader scheme varies in turns. Sharp turns that are on the leaders side are handled better than turns on the opposite side of the leader. This is because all other agents will detect their positional errors and try to correct them as soon as the leader starts turning. However, in opposite side turns, the leader will be the last one to start turning and thus, have the largest positional error. Since the leaders always assume their position is correct, it will make no corrective manoeuvres while the other agents will move backwards to keep their alignment with the leader. Due to this asymmetry, Single-leader scheme loses its advantage over others in right-angled turns.

Neighbor scheme was the least performing among three, in terms of strictness in formation maintenance. The major drawback of this scheme is that positional errors made by one agent will propagate to others that directly or indirectly refer to it. This drawback is obvious if the terrain is rough and agents need to deviate from the waypoints they are assigned to, due to the limited traversability of the terrain. Neighbor scheme was able to maintain the best average speed and got the best path ratio.

It is concluded that, preserving the shape of the formation and maintaining a constant speed are conflicting goals most of the time. Each of the schemes presented here favors one over other. Among the three formation schemes, Unit-center provided the best results in terms

of formation preservation. Single-leader and Neighbor schemes are viable alternatives in some application domains, especially when a two-way communication between agents in the formation is not always possible, such as robot formations.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we built upon the foundations in [5] and proposed a method that introduces multiresolution formation support and new formation referencing schemes. Since the findings obtained throughout this research are fairly straightforward to implement and verify, they are easily adaptable to many application domains like computer games and simulations. In order to both evaluate the performance of the proposed method, and to visualize the behavior of agents, a simulation software was created along with this study.

The method explained here first starts by preprocessing a height map to build a search graph. As shown in Chapter 5, this preprocessing step vastly reduced the number of vertices and provided the basis for path finding. The search graph is then used to find an optimal path from a start to a goal location, using the well known shortest path algorithm A*. Afterwards, the position of agents at each path point is determined, and the formation begins its motion passing through each path point along the way.

We allowed the formations to be defined using any of the three supported formation definition schemes. In order to support formations consisting of smaller formations, we have come up with an approach for treating agents and formations uniformly inspired by the composite pattern that is used in working with tree-like data structures. Thanks to this approach, hierarchical formations of arbitrary depths are supported.

We evaluated the performance of the proposed method using the experimental setup given in Chapter 4 and obtained satisfying results. The performance of three formation referencing schemes varied during experiments. The metrics we used in evaluation measured different and sometimes conflicting qualities of the schemes. For example, going fast and preserving the formation were conflicting goals and each formation referencing scheme favored one over

another. Overall, we concluded that Unit-center approach gave the best performance since it gave a balance between conflicting goals. Formations designed using Unit-center scheme were especially successful in preserving the shape of the formation throughout the path, with a slight trade-off in average speed. However, since it is not always possible to maximize all goals in all cases an appropriate formation scheme must be chosen depending on the priority of these goals imposed by the application domain.

6.1 Future Work

Currently, only static obstacles (i.e. non moving) are considered and the planner needs to be re-run in order to compensate for the changes in the environment. As a future work, A* algorithm used in off-line path planning can be replaced with a dynamic variant that supports changing the path to repair newly invalidated sections or exploit newly introduced passages. Also, the on-line planner can be extended to handle dynamic obstacles in the environment.

Another possible improvement is introducing the ability to switch from one formation configuration to another on the fly. This will be especially useful in scenarios where different formation configurations are more suitable in different sections along the path. This improvement can be handled by modifying the part of the on-line planner that determines each agents positions along the path.

REFERENCES

[1] N. Sahli and B. Moulin. Ekemas, an agent-based geo-simulation framework to support continual planning in the real-word. *Applied Intelligence*, 31(2):188–209, 2009.

[2] A. Ceranowicz. Modular semi-automated forces. In *Proceedings of the 26th conference on Winter simulation*, pages 755–761. Society for Computer Simulation International, 1994.

[3] D. McIlroy and C. Heinze. Air combat tactics implementation in the smart whole air mission model (swarmm). In *Proceedings of the First International SimTecT Conference*. Citeseer, 1996.

[4] A.K. Das, R. Fierro, V. Kumar, J.P. Ostrowski, J. Spletzer, and C.J. Taylor. A vision-based formation control framework. *Robotics and Automation, IEEE Transactions on*, 18(5):813–825, 2002.

[5] A.G. Bayrak and F. Polat. Formation preserving path finding in 3-d terrains. *Applied Intelligence*, pages 1–21.

[6] I. Millington and J. Funge. *Artificial intelligence for games*. Morgan Kaufmann, 2009.

[7] S. Kambhampati and L. Davis. Multiresolution path planning for mobile robots. *Robotics and Automation, IEEE Journal of*, 2(3):135–145, 1986.

[8] L. Perkins. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[9] K.H.C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS)*, 2008.

[10] J.N. Bakambu, P. Allard, and E. Dupuis. 3d terrain modeling for rover localization and navigation. In *Computer and Robot Vision, 2006. The 3rd Canadian Conference on*, pages 61–61. IEEE, 2006.

[11] M. De Berg, O. Cheong, and M. Van Kreveld. *Computational geometry: algorithms and applications*. Springer-Verlag New York Inc, 2008.

[12] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[13] A. Stentz. The focussed dˆ* algorithm for real-time replanning. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 1652–1659. Citeseer, 1995.

[14] I.S.C. Mellon. Optimal and efficient path planning for unknown and dynamic environments. 1993.

[15] R.E. Korf. Depth-first iterative-deepening* 1:: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[16] C.W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, volume 1999, pages 763–782. Citeseer, 1999.

[17] Y.K. Hwang and N. Ahuja. A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on*, 8(1):23–32, 1992.

[18] L.E. Parker. Designing control laws for cooperative agent teams. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 582–587. IEEE, 1993.

[19] T. Balch and R.C. Arkin. Behavior-based formation control for multirobot teams. *Robotics and Automation, IEEE Transactions on*, 14(6):926–939, 1998.

[20] K.H. Tan and M.A. Lewis. Virtual structures for high-precision cooperative mobile robotic control. In *Intelligent Robots and Systems' 96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, volume 1, pages 132–139. IEEE, 1997.

[21] N. Megiddo. Linear-time algorithms for linear programming in r3 and related problems. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 329–338. IEEE, 1983.

[22] G.S.P. Miller. The definition and rendering of terrain maps. *ACM SIGGRAPH Computer Graphics*, 20(4):39–48, 1986.

[23] R.M. Smelik, K.J. de Kraker, T. Tutenel, R. Bidarra, and S.A. Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009.