Real-Time Autonomous Crowds in Graphical Worlds
Abstract

This is a report on theory and implementation of crowds in 3D computer worlds. Different algorithms and theories are discussed as why they were used or why they were rejected. The main focus of the work has been to be able to have groups and non-controlled agents that appear as natural and non-invasive as possible while still being easy to control.

The goal platform has been games and other applications where a strict simulation is not as important as a realistic facade. In the end, a sector-based path finding engine was found to be most efficient coupled together with an easy scripting language and hardwired primitives for group forming, following and behaviors. Actions and behaviors are represented generically and can be interpreted differently by different visualization media. In fact, the engine can be used to represent groups in both non-graphical as well as 2D environments aside from the originally planned 3D interaction.

Keywords: crowd behavior, agents, artificial intelligence, computer graphics, real-time
# Content

Abstract ................................................................................................................................... 2  
Content .................................................................................................................................. 3  
1 Introduction .......................................................................................................................... 6  
1.1 Background ...................................................................................................................... 6  
1.2 Purpose and Question at Issue .......................................................................................... 7  
1.3 Theory .............................................................................................................................. 7  
1.4 Methods, Tools and Software Used .................................................................................. 8  
1.5 Definition of Terms .......................................................................................................... 8  
2 Crowds in 3D-Graphics ....................................................................................................... 9  
2.1 Usage Areas ..................................................................................................................... 9  
2.2 Related Work ................................................................................................................... 9  
3 Theory .................................................................................................................................. 12  
3.1 Movement ......................................................................................................................... 12  
3.1.1 Path Finding ................................................................................................................. 13  
3.1.1.1 Finding the Next Sector .......................................................................................... 13  
3.1.1.2 Finding a Point on the Portal .................................................................................. 14  
3.1.1.3 Bézier Curves ........................................................................................................ 15  
3.1.2 Collision Avoidance .................................................................................................... 15  
3.1.2.1 Decoupled Planning ............................................................................................... 15  
3.1.2.2 Speed Variations ...................................................................................................... 16  
3.1.2.3 Detouring ................................................................................................................ 16  
3.1.2.4 Detecting Collisions .............................................................................................. 16  
3.1.2.5 Path Planning ......................................................................................................... 18  
3.1.3 Spatial Subdivision ....................................................................................................... 18  
3.1.4 Group Movement ....................................................................................................... 19  
3.1.5 External Objects ........................................................................................................ 20  
3.2 Scripting & Behavior ....................................................................................................... 20  
3.2.1 Example of Uses ......................................................................................................... 21  
3.2.1.1 Walking ................................................................................................................ 21  
3.2.1.2 Events and Actions ............................................................................................... 22  
3.2.1.3 Continuing Actions ............................................................................................... 23  
3.2.2 Objects in the World ................................................................................................. 23  
3.2.2.1 Agents .................................................................................................................. 24  
3.2.2.2 Groups .................................................................................................................. 24  
3.2.2.3 Event Points ......................................................................................................... 25  
3.2.3 Spawning and Deleting .............................................................................................. 25  
3.2.4 Controlling Walking ................................................................................................. 26  
3.2.5 Behaviors and Actions .............................................................................................. 27  
3.2.6 Conditionals .............................................................................................................. 28  
3.2.7 Control versus Autonomy ......................................................................................... 28  
3.2.8 Textual Representations of Behavior ........................................................................ 29  
3.2.9 Randomness .............................................................................................................. 29  
3.3 Level of Detail .................................................................................................................. 30  
3.3.1 Calculating Level of Detail ......................................................................................... 30  
3.3.2 View ........................................................................................................................... 30  
3.3.3 Population Control .................................................................................................... 33  
3.3.4 Reducing Calculations .............................................................................................. 33  
4 Implementation .................................................................................................................... 34  
4.1 System Overview .......................................................................................................... 34  
4.2 Movement ....................................................................................................................... 35  
4.2.1 The World .................................................................................................................. 36  
4.2.2 Finding the Next Portal .............................................................................................. 36  
4.2.3 The Grid ..................................................................................................................... 36
List of Figures

Figure 1 An area of passable space consisting of sectors 1-3 connected by two portals A, B.................................................................13
Figure 2 Path without reference points.................................................................14
Figure 3 Path with reference points r1 and r2.............................................................15
Figure 4 A collision would occur if the velocities were reversed..................................17
Figure 5 A collision has occurred...............................................................................17
Figure 6 A collision will occur at $t_{\text{min}}$................................................................17
Figure 7 A collision is detected at time $t = t_{\text{min}}$..........................................................17
Figure 8 The subregions keep track of the agents that are currently inside them.........19
Figure 9 An agent has to avoid collisions against agents in neighboring subregions......19
Figure 10 Agents within the groupradius copy the leader’s velocity...............................20
Figure 11 An advanced example of desired walking.....................................................22
Figure 12 Fleeing in a street..........................................................................................23
Figure 13 The starting state of sectors and portals.........................................................31
Figure 14 The first two vectors from the first portal.......................................................31
Figure 15 New vectors constructed from the portals’ corners and intersections.............32
Figure 16 The newly constructed vectors applied..........................................................32
Figure 17 A new direction is calculated to avoid collision.............................................38
Figure 18 Simple walk point example............................................................................39
Figure 19 The mall example.........................................................................................40
Figure 20 An example with an explosion.......................................................................41
Figure 21 A picture from the 2D version.......................................................................41
Figure 22 Walking in 3D..............................................................................................51
Figure 23 A sector where problems can arise...............................................................53
1 Introduction

This is a report on a work on crowds in graphical worlds, specifically real-time simulated 3D worlds. It was written at Lunds Tekniska Högskola (Lund’s Institute of Technology) [31] with the help of the Computer Science department, especially the section for computer graphics.

It is about how agents, program entities representing an actor in a virtual world, and how they interact in groups, sets of agents. The primary goal has been to create an engine for realistically behaving groups and agents to be used in games and other applications where facade and looks are more important than a stringent simulation. It needs to be emphasized that this is in no way a simulation, and as such is not suitable for usage in simulating emergency drills. Note that no work has gone into actual graphics like textures or animation systems.

Four main areas have been predominant – path finding, grouping, behavior and control. Our goal was that the group system should be simple to use and control while being able to present a convincing behavior from the groups and agents it controls.

One of the major downfalls of many applications which simulate for human environments, such as cities, is the fact that using many computer controlled agents simultaneously can be very resource heavy. By concentrating on the areas that make humans identify entities as humanoid and making sure these are effective and fast a system that can create and maintain a very large amount of well behaving agents can be created.

1.1 Background

As computer power gets more and more evolved and more advanced virtual worlds are created the need for systems capable of handling large crowds arises. Some systems have been created to support large crowds for pre-generated movies and simulations. The only systems for real-time realistically looking behavior have been either games which are lacking heavily in the realistic area, and a few research projects.

We foresee a need in many applications to come where a system for handling and controlling large crowds are needed. An example for this is games taking place in areas with many non-user-controlled people such as towns or large outdoor areas. Another example is town planning or other areas where a natural looking collection of people would ease the design and construction planning.

When more of the processing load for advanced graphics applications has been shifted to the GPU more of the CPU is available for other tasks in an application, especially artificial intelligence, which in turn increases the interest for larger and more advanced systems.

Traditionally, few games or programs employed more than a few entities active at the same time and even then with very straightforward objectives and behavior. Even fewer programs had some kind of non-interacting bystanders. Most of this came from a lack of resources, but also a lack of need. Games were simple and did not try to reflect reality and therefore had no need of well behaving autonomous crowds.

Lately games like Grand Theft Auto 3 [22] and True Crime [23] have made use of larger crowds in the size of up to 50 simultaneous entities, but with a surprisingly simple artificial intelligence with obvious glitches. Some obvious improvements over these titles
are pin-pointed spawning/deleting as to not create the “disappearing pedestrian”-problem and better path finding so they don’t walk into and through each other.

1.2 Purpose and Question at Issue

By nature of software engineering the question that was at the core of our work is of an empirical nature;

- How does one design a system to handle a large number of computer-controlled agents in real-time while still having agent behavior realistic enough to make the agents indistinguishable from a user-controlled entity?

The purpose is to maximize the resources given to the program by the computer to enable a maximum number of agents while still maintaining a level of realism and behavior as to be able to make a convincing humanoid to a user. We believe that although this is an area explored to some level in systems dealing with pre-generation, such as the Massive system [16], it has not been investigated to any greater depth despite being more and more common in computer applications, especially games.

We believe that by using individually simple components an emergent behavior can be created, complex enough to fulfill the criteria of realistic behavior. Because our time and resources are limited we do not always delve very deeply or implement some topics but often note what we have concluded or researched in that area.

1.3 Theory

It is our belief that large crowds can be handled in real-time and by “cheating”. The impression of even larger crowds can be created without the need for blatantly removing or adding agents in plain view for the user. We also believe that it is possible to have crowds that move so naturally that a user or player moving an entity of their own will blend in and not look out of place, assuming their behavior and movement is at least somewhat normal.

What separates our goal from that of simulating crowds is the before-mentioned “cheating”. This means we can and will remove and add agents as needed, simplify their behavior or take shortcuts in path finding and collision avoidance where applicable.

Our basic theory on how to accomplish this is that the most important attribute if you want something to appear human is how it walks and moves, especially true when you have large groups of entities. Humans do not walk straight forward until they bump into something and then turn left, nor do they use something like A* [1] to find their path. Instead human walking patterns are a complex game of body language, eye contact and social patterns. Often small course adjustments are made to ensure you pass on a suitable distance next to another person.

These kinds of patterns gets yet more complex when people are in groups and not only adjust their course based on everyone else, but also tries to keep group coherence. Due to this the main focus of our work has been on a path finding engine suitable for these behaviors while still being so fast as to allow a large number of agents.

With this in mind, another important aspect was that it is often desirable to be able to do extra-ordinary activities and direct the groups within an application. To allow for this there has to be an easy way to control and direct agents behavior which facilitates some kind of control interface. We believe a script language which can be loaded and executed dynamically is the best solution to this.
Coupled with the script language, built-in behavior types and definable points of action a general behavior can be created and controlled to a satisfactory level.

1.4 Methods, Tools and Software Used

Several other works within the realm of simulated behavior was divulged before work commenced. Of these especially the works of Thalmann and his research staff at the VRLab [17] in Lausanne, Switzerland and the project for densely populated urban environments [5] at the University College London was of note. These works seemed to share some of our main points, especially the ViCrowds [6] project at the VRLab. However we believed that since the subject is only still at its infancy, a research from the bottom was motivated. Although many documents were studied before the work began, few external resources were needed or used once work had begun.

Several 3D engines were examined and two of them tested; Nebula [18] and Crystal Space [29]. After some problems with Nebula combined with the character animation library Cal3D [30] it was decided to use Crystal Space despite its sub par performance. It was decided from the beginning to use Cal3D for character animation due to previous experience; additionally ODE [19] was to be used if the need for rigid body physics aroused. To handle the connection between Crystal Space and Cal3D we used another work from LTH [31], the Cal3D plug in [7] by Arton Grajqevci.

Our main development platform was Visual Studio.NET 2003 [20] for its nice Crystal Space integration as well as good debugging options. Hammer [21], with some additional scripts were used to create test worlds.

1.5 Definition of Terms

Here are some terms frequently used and what they represent.

- **Crowd-system** – A name to signify the complete system handling crowds.
- **Executing environment** – The application that uses the crowd-system as a part of its execution. For example, a game engine.
- **World** – The world in which the agents move.
- **Agent** – A single entity in the world.
- **Group** – A set of agents.
- **Behavior** – Everything an agent does except for movement.
2 Crowds in 3D-Graphics

As noted above, traditionally, crowds have not been a large part of games and applications. Despite this, their use has increased the last couple of years and some usage areas and previous implementations exist.

2.1 Usage Areas

To begin with, it should be noted that what we talk about here are systems that handle crowds that give an image of being real, not that act real. This means that these kinds of systems are not suitable for simulating emergency evacuations or other similar usages. What they are suitable for are

- Games
- Bringing a sense of life to graphical models

The main focus is games or possibly other real-time engines with weight placed on realistic behavior.

2.2 Related Work

Most notable implementations with real-time crowds are in games such as:

- Grand Theft Auto 3 [22]
- True crime [23]
- City of Heroes [24]

Grand Theft Auto 3 [22] is a very popular game but the crowds in the game do not give a completely realistic behavior. Most notable is that even agents that are within the view are removed very fast. This is unfortunately very obvious to the player. We strive to support a large enough number of agents to avoid this – we only want to remove agents far away from the player.

The other games also have their deficiencies such that in City of Heroes [24], if a player’s avatar is standing still the agents representing the crowd will start to just “push” the avatar in front of them if it is in their path. True Crimes [23] is the one of these three games (and most other games) that we think have solved many problems best. The player’s avatar will literally bump and push agents out of his way when they can’t get out of the way quick enough (something we try to mimic) and agents aren’t spawned or deleted in plain view. However it still lacks consistency in the world. If a person is standing around in an area and the player leaves out of sight and then returns the person is gone.

There is also some research in the area that uses different methods to simulate the crowd.

Table 1 has been modified from its original work [8] and the ViCrowd [6] version to show various approaches in the field, including our crowd-system.
Table 1 A comparison between different kinds of group systems

As Parent [8] mentions particle-, flocking- and behavioral systems are categories that have found their way into the literature, nothing prevents mixing attributes from the categories.

Particle systems and flocking systems have an emergent behavior; a global effect is generated by local rules. Gloor, Cavens, Lange, Nagel and Schmid have simulated virtual pedestrians in large scale applications using particle systems [11]. The agents follow pre-made paths and no control is possible during execution, it is not suitable for games but they do establish that 2D graphics are very practical for debugging, which we have used greatly throughout our project. Reynolds uses a generalization of particle systems to simulate flocks in combination with a behavioral model to decide the motion of the flock [12]. Reynolds focus is on flocks consisting of birds that flies in 3D and is not suitable for human movement. Reynolds states general rules for flocking behavior that we have used for controlling the movement of groups of agents.

The particle system approaches are able to handle a large number of participants. Force fields are used to control the agents’ paths. Whereas this suitable for some usage areas it does not give the individual control of the agents that is desirable for games.

Loscos, Tecchia and Chrysanthou address the problem of shadow computation for large numbers of dynamic objects [5]. In their crowd-system agents outside of the view is immediately removed and new agents are created when the view changes. We do not want to remove the agents as quickly but it inspired us to that several calculations, such as collision avoidance, can be neglected for agents outside of the view. Other interesting work in the graphics area has been made by Perlin and Goldberg who use procedural techniques to create non-repetitive motions of agents [13].

Tu and Terzopolous have used behavioral animation for creating for creating artificial life [14]. The behaviors of the agents rely on their perception of the environment. Behavioral systems generally do not support many participants and the control mechanism is too complicated to be suitable for a game application.

Fuertey simulates the collision avoidance behavior of pedestrians using (x, y, t) space [10]. He established that the density of agents influences the walking speed as well as the distance at which agents start to detour. Some of Fuertey’s ideas are used in our movement-system.
Tsai-Yen Li and Hsu-Chi Chou addresses the problem of motion planning for a crowd of robots [9]. They establish that when the path for each robot is generated at different times decoupled planning is more appropriate than a centralized approach. We are using decoupled planning in our movement-system.

Musse and Thalmann simulate crowds of humans in real time in ViCrowd using a hierarchy of crowds, groups and individuals [6]. ViCrowd offers good control on different levels and is able to handle many agents. ViCrowd is a large project that supports several different usages. Our work is largely inspired by ViCrowd and the likeness can be seen in Table 1. ViCrowd supports the definition of obstacles in two ways, either by declaration of the obstacles or by declaration of the areas where the crowd can walk. We are using the second approach. ViCrowd describes crowd motion by interest points (locations where the crowd must pass through) and action points (locations where the crowd can if necessary go and on arrival must perform an action). Our scripting language is inspired by this. ViCrowd aims to generate human crowds based on groups instead of individuals, whereas the groups contains goals, emotions, knowledge and intentions the individuals are just able to avoid collision with obstacles and other agents. Although our system to a higher degree treats groups and individuals alike, the basic group design is inspired from ViCrowd along with the behaviors that can be programmed in the script language, such as: flocking, following, attraction and repulsion.

Our main contribution is our focus on games. Due to this focus, optimizations can be applied that is not possible in the more general ViCrowd system. ViCrowd does not implement level of detail, which we believe is necessary in games today. We investigate spawning and deletion of agents which we believe must be done in a realistic way. Also in games the crowd must take consideration of user controlled players, we have integrated this in our crowd-system.
3 Theory

In this section the theory that our implementation is based on is described. The problems are acknowledged and explained. We present our solutions as well as other possible approaches. The advantages and drawbacks with our solutions are discussed.

In the movement section the theory and algorithms involved with agents in motion are explained and discussed. This includes path finding and collision avoidance, the basic fundamentals of the movement-system. A sector based path finder is presented as well as simple collision avoidance algorithm that works in (x, y, t) space. The use of spatial subdivision to enhance speed is discussed and a solution presented. How groups and externals objects are handled is also explained.

In the scripting section it is examined what possible uses of the scripts are and what kind of behavior and actions can be enacted. Descriptions of different methods and possibilities to accomplish certain types of behaviors are discussed and all the objects that can exist in a script are gone through. In the end of this section a small discussion about representing behaviors in computer systems is held.

In the level of detail section the use of level of detail to enhance speed is discussed. How it can be calculated and used to reduce calculations in the movement-system is explained. A scheme for building a BSP-tree [4] that can be used to determine if an agent is inside of the view, and how this can be used when adding or deleting agents is presented.

3.1 Movement

The movement of agents is a basic fundament of a crowd-system; it is also one of the computationally most expensive due to the often complex algorithms. An agent must have the ability to move through the world while avoiding collision with static objects (such as walls) and dynamic objects (such as other agents). It is desirable that this is done in as natural looking way as possible. What looks natural is large dependant on what kind of objects the agents represents. For example human walking patterns differ enormously from how cars move. We have only concentrated on agents representing humans. Human walking patterns are very complex and only the most basic behaviors can be simulated fast enough. We focus on the aspects that are most important to give an adequate movement. This largely involves avoiding collisions before they occur. Since collisions are to be avoided no collision detection should be needed for the agents. This is similar to how ViCrowd [6] handles collision avoidance.

In this project we will only consider 2D movement which is less complex than 3D movement and sufficient for many of the games on the market today. This is a normal simplification, ViCrowd [6] also only handles 2D movement. However several particle systems [12] handle 3D movement. Many of the ideas presented here can however easily be transferred to 3D movement.

Within the movement field we include two, partially independent, areas.

- **Path finding**: The task of the pathfinder is to find a path from point X to point Y avoiding static objects that may be blocking the way.
- **Collision avoidance**: Collision avoidance is the task to avoid collisions between dynamic objects. This assumes we have a path for the object.

An agent has a 2D position (x, y) but it still occupies space. How this space is represented is an important factor. We represent this space by a radius. This is the simplest and fastest
solution and it works perfectly well for agents that can be approximated by a circle. People fall in this category, cars do not. Henceforth in the movement-algorithms agents will be thought of as circles.

3.1.1 Path Finding

In order to find a path between two points and avoiding static objects all the static objects must be known to the pathfinder. We have chosen to use an approach where all passable space is known to the pathfinder. This approach is also used by ViCrowd [6]. The task is hence not to avoid objects, but to find a path that solely passes through passable space.

We are dividing this passable space in sectors. A sector represents a rectangular area of passable space. Sectors are connected by portals. A portal is a vertical or horizontal border between sectors. This simple division of the passable space can be extended to non-rectangular sectors which would allow more complex structures, but we believe rectangular sectors is a sufficient simplification for this project as it still offers good possibilities of map building. The idea with a sector is that an agent can walk in a straight line to any other point in the sector. Hence no path finding is needed within the sector.

![Figure 1 An area of passable space consisting of sectors 1-3 connected by two portals A, B.](image)

An agent moving from sector 1 to sector 3 in Figure 1 will pass through portals A, B and sector 2. The problem of finding a path for an agent can then be reduced to:

1. Finding the next sector to move to, and a portal that leads to it.
2. Finding a point on the portal and move straight towards it

These steps can be repeated until the agent is in the last sector where it can move straight towards its goal.

3.1.1.1 Finding the Next Sector

The problem is to find the next sector for the agent to move to. For this we assume that the current sector and the final sector are known. The next sector to move to should be chosen as the neighboring sector with closest distance to the final sector. This is the problem of finding the shortest path. There are many algorithms to solve this problem; in this project we are using a pre-calculated table that for any two sectors gives the next portal.

The advantage of this is speed. A drawback is that sectors can not be changed during execution; neither can dynamic aspects such as current agent density in the area be weighted in the algorithm. To create the pre-calculated table the all-pairs shortest-paths problem has to be solved, we are using the Floyd-Warshall [3] algorithm to do this.
Table 2 This is the pre-calculated table for the sectors and portals in Figure 1.

<table>
<thead>
<tr>
<th>Sector</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>A</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>-</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>B</td>
<td>-</td>
</tr>
</tbody>
</table>

3.1.1.2 Finding a Point on the Portal

The easiest solution to this problem is that an agent moves towards the closest point on the portal that is at least the agent’s radius distance from the portal edge. This point can be found fast and easy. This approach leads to a nice path in the current sector. However it does not take in account the next sector which leads to unnatural turning at the portals. We found that this unnatural turning was very obvious to a spectator and unfortunately too large a simplification.

To solve this we use a reference point that approximates where the agent will be moving after it passes through the portal. For the last portal this reference point will be the goal position. By counting backwards from this point we can obtain an exact reference point for all the portals, this would be time consuming. However we found that a very simple approximation of the reference point as the midpoint of the next portal gives adequate results.

The reference point is used so that instead of choosing the closest point on the portal, the portal point will be chosen so that the path to the reference point that goes through the portal point will be the shortest possible. How this is done is described in the implementation part 4.2.4.
Figure 2 and Figure 3 show the differences when using an approximated reference point. Though sharp turning is still done at the portals we find that it looks realistic enough to a spectator.

3.1.1.3 Bézier Curves

Bézier curves [2] can be used to make a smooth path without sharp turns. The portal points can be used to create the Bézier curves and the tangent of the curve can be used to determine the agent facing. We decided however that this would be worth neither the extra implementation time nor the extra processor time; our goal is to make a fast engine that is adequately realistic. Bézier curves would especially make a huge negative speed impact on the collision avoidance.

3.1.2 Collision Avoidance

When we have several agents the task of collision avoidance is very important. The objective is to avoid agents overlapping each other without using computationally expensive physics algorithms. That is, we want to avoid collision before it occurs. It is important that this is done in a way that looks natural.

We have based our collision avoidance on some general ideas inspired by Franck Feurtey [10] and Tsai-Yen Li and Hsu-Chi Chou [9]

- **Decoupled planning**: All agents do not plan their paths at the same time; when an agent plans its path it accommodates to all other agents that already have a planned path.
- **Speed variations**: To avoid collision an agent may temporarily reduce its speed.
- **Detouring**: If speed reduction is not applicable to avoid collision, the agent can change its planned path within the sector.

3.1.2.1 Decoupled Planning

Tsai-Yen Li and Hsu-Chi Chou establish that when the paths are generated at different times decoupled planning is more appropriate than a centralized approach [9]. The decoupled planning differs from normal human movement; people generally are not concerned of which path was planned first. Precedence is instead normally based on who reaches the crossing point first. The effect of this is that an agent may at times give precedence when it would look more natural if it did not. This decoupled planning is however very practical. It is very important for reducing the number of calculations as already planned paths will not have to be re-planned when new paths are added. We also found that it does not particularly affect the overall impression.
3.1.2.2 Speed Variations

Feurtey suggests speed variations and detouring for avoiding collisions [10]. Avoiding collision by speed reduction is very common for real people. It is therefore natural to have it as a part of the collision avoidance. To simplify the calculations and the implementation we have settled with reducing the speed to zero, meaning the agent either moves at its preferred speed or is standing still. This is a simplification that is only possible for agents that can accelerate and decelerate very fast, for example it is not possible for cars. Sometimes it is not reasonable to wait for the other object:

- **Zero velocity:** If the other dynamic object has zero velocity we might have to wait forever
- **Wrong angle:** If the angle between the velocities of the objects is too small or too large the agent may have to wait for an unreasonable long time. Two agents moving straight towards each other would have to wait for infinity.

In these cases the agent will have to detour from its current path.

3.1.2.3 Detouring

When speed variations are not enough to avoid collision the agent will have to detour from its original path. We are doing this by letting the agent move in another direction for some time and then try to find a straight path towards its goal again. This is a natural approach, also used by ViCrowd [6]. The direction is chosen so that the agent will pass close to the left or right of the blocking agent. Which side it passes on depends on which side that requires less deviation from the original path.

When the distance between the agents is small it is natural that they pass close to each other. When the distance is larger people tend to avoid getting other people to close, hence the distance at which they pass should be larger. Likewise the distance at which the detouring starts should be dependent of the agent density as Feurtey [10] mentions. We have been experimenting with this trying to find a good balance. We found that varying these distances only makes a small visual effect. This partially depends on our spatial division that prevents agents from noticing each others at large distances, due to which detouring is natural as soon as a future collision is detected.

3.1.2.4 Detecting Collisions

In order to prevent a collision it must first be detected. This is the most costly part of the movement system. Several algorithms with varying level of complexity exist. A common approach is that collision avoidance is done in (x, y, t) space. In (x, y, t) space all objects are static and the problem becomes that of finding a 3D collision. This can also be used to select good speed variations and detouring as described by Feurtey [10]. As described above we are using simple solutions for speed variations and detouring that are computationally fast, but not as exact as a more complex (x, y, t) space algorithm. Due to this we can use a simple algorithm to detect future collisions. For this ViCrowd uses a procedural method that is based on the intersection of two lines [6]. We have compiled another method.

What we basically want is to find if there will be a collision and when this collision will occur.

The distance between two dynamically moving points with velocities \( \mathbf{u}, \mathbf{v} \) can be described as
\[ d(t) = w(t) = w_0 + t(u - v) \]
\[ w_0 = P_0 - Q_0 \]

Where \( P_0, Q_0 \) are the positions at time \( t = 0 \) and \( w(t) \) is the vector between the points. This is explained by Dan Sundy [15].

We want to calculate the time \( t \) at distance

\[
\begin{align*}
    d(t) &= \text{distance} \\
    d(t)^2 &= \text{distance}^2 \\
    d(t)^2 &= w(t) \cdot w(t) = (u - v) \cdot (u - v)t^2 + 2w_0 \cdot (u - v)t + w_0 \cdot w_0 \\
    \Rightarrow (u - v) \cdot (u - v)t^2 + 2w_0 \cdot (u - v)t + w_0 \cdot w_0 - \text{distance}^2 &= 0
\end{align*}
\]

This is a regular 2nd degree equation. It can be used to check for collision between two agents if \( \text{distance} \) is set to the sum of the agents’ radiiuses. If the equation has real solutions, \( t = t_{\text{min}}, t_{\text{max}} \) we have found a collision. If \( t_{\text{min}} > 0 \) this collision will occur in positive time and we must prevent it using speed variations or detouring. If \( t_{\text{min}} < 0 \) and \( t_{\text{max}} > 0 \) the agents are already too close, a collision has occurred and must be handled.

These calculations can be done fast, \( t_{\text{min}} \) gives the point in time when the agent must wait and in addition to that \( t_{\text{max}} - t_{\text{min}} \) gives a good approximation for how long time the agent must wait.
3.1.2.5 Path Planning

Assuming no external influence, all agents could plan their entire path without running the risk of having to re-plan it. Normally this will not be the case due to user controlled objects, such as a player, and due to new movement orders to the agents. Planning a long collision free path is naturally more time consuming that planning a short path, if the path has to be re-planned due to external event the time spent planning the long path will be wasted. In our solution an agent only plans a collision free path until its next change of velocity. An agent might plan to keep its current velocity for 5 seconds and then wait for another agent that is temporarily blocking its path.

Every time an agent changes its velocity, collision avoidance against other agents is checked for. Using this method there will be more collision avoidance checks than when planning longer paths, but they will be easier and faster to calculate and therefore re-planning due to external events will not be as bad.

3.1.3 Spatial Subdivision

When an agent changes its path it has to check for collision against all other agents. This is very costly and to reduce the number of collision checks that have to be made spatial subdivision is needed. By dividing the space an agent only has to avoid collisions against nearby agents. None of the crowd-systems we have looked at describes how they handle this. We have considered two basically different methods of doing this.

- **Static subregions**: The subregions are static and have constant areas. The number of agents in each subarea can vary and cause many collision checks in crowded regions.
- **Dynamic subregions**: To avoid a huge number of collision checks in crowded areas the number of agents in each subregion is kept constant or below a maximum. This is done by have dynamic subregions.

We investigated the approach of dynamic subregions. This can be realized by a quadtree [4]. The drawback of this method is that the quadtree can not be pre-computed; it has to be updated as the agents move. After close consideration we decided on using static subregions due to some characteristics of the crowd:

- **Movement**: A thesis of ours is that agents on the move looks more natural, we want to keep the crowd almost constantly moving, making tree updating more expensive.
- **Dispersal**: It is desirable that the crowd is dispersed evenly as it often gives a realistic impression and reduces problems of a natural looking movement when agents are very crowded. Hopefully the number of agents in each static subregion should therefore be fairly equal.
- **Level of detail**: As discussed in the level of detail section, 3.3.4, agents far away from the viewer or outside the field of view do not have to avoid collision. Updating the quadtree for these agents is therefore unnecessary work.

Although the position of the agent is guaranteed to be inside the agent’s current subregion this is not the case for the circle that is the agent’s space. To avoid agents in different subregions overlapping each other, collision avoidance checks must be done against agents in the agent’s current subregion and the neighboring subregions.
Every time an agent enters a new subregion it has to check its path for collisions, hence the size of the subregions can not be too small or the number of collision avoidance checks will start to rise. The optimal size of the subregions is depending on how crowded the area is. More agents make smaller subregions more preferable.

3.1.4 Group Movement

People often walk in groups and it is desirable that this behavior can be simulated in a crowd-system. People in groups walk with approximately the same velocity and when they have found a group coherency that they are satisfied with they try to keep it. In order to imitate this behavior, agents must be aware of the actions of the other members in the group. In a group consisting of a set of agents we have one of the agents as a leader, the rest are followers. The leader directs the movement of its group. This is a definition of a group similar to ViCrowd’s [6]. Reynolds states the following rules in order of decreasing precedence for simulating flocks [12]:

1. Collision avoidance: avoid collision with nearby flockmates
2. Velocity matching: attempt to match velocity with nearby flockmates
3. Flock centering: attempt to stay close to nearby flockmates

We are basically using these rules for directing the followers in the group. To decide when rule 3 should have precedence over rule 2 a group also has a group-radius; the followers will try to stay within group-radius distance from the leader. When moving, the following considerations are made for groups:

- **Leader:** If possible the leader will pass through portals in such a way that it is at least group-radius distance from the portal edge.
- **Followers:** If a follower is within group-radius it will try to copy the leader’s velocity, moving parallel to the leader (rule 2). If a follower is outside the group-radius or unable to copy the leader’s velocity due to obstacles it will move towards the next point the leader is moving to so that it will come within the group-radius (rule 3).
3.1.5 External Objects

Not all the objects in the world will be under the control of the crowd-system. In games the agents must interact with external objects such as a player in a natural way. This mainly includes collision avoidance. This is not investigated in ViCrowd [6] and many games have lousy collision avoidance between agents and the player. Our approach is that agents should treat the external objects as they treat each other. For this the crowd-system needs to know the position and radius of all external objects. By comparing the new position to the old, the velocity of the object can be calculated. The velocity can be used for the collision avoidance algorithm. External objects are in this way handled very much like agents. The main difference is that whereas agents will do their best to avoid external objects there is no guarantee that external objects will try to avoid agents.

3.2 Scripting & Behavior

Even though the goal of the project is to have semi-autonomous groups and agents there is a need to control and decide behavior. Additionally logistics of creating, deleting and handling agents should be taken care of outside of the code itself. We decided that all of this was to be handled by some kind of scripting language interpreted by the engine. The areas and needs that are required can be categorized into three fairly distinct groups of functions:

- **Logistics** – Creation, deletion and handling of agents and groups. This include specifying which agents initially are included in a group, setting up event points for deletion and spawning and naming the groups and agents that need known names.
- **Behavior** – Specifying different types of behaviors and actions and applying these to agents. In this category, objects like walk points and patterns are included.
- **Immediate** – Events and actions that are applied at once. This can be making a group or an agent walk somewhere, begin a walk pattern or make an agent, group or event point execute a one-time action.
One question that immediately arise is how finely grained control of the agents is desirable. The quick answer is enough to gain sufficient control over them while keeping the amount of scripting needed to a minimum. However, this is of course harder to achieve in practice than it is to describe.

Since the project aims to make it as simple as possible to facilitate large crowds, especially background crowds which aren’t actively interacting with a user, it is of great importance that it is easy to set up large anonymous crowds following a certain behavior and at a later point be able to interact with these without specifically knowing which agents should be affected by an area effect.

This also raises the question of what kind of actions and behaviors are interesting. A more in-depth answer follows in the part about behaviors, a short answer is that we have found that only a few different kinds of built-in behaviors and actions are necessary.

Some of these basic building blocks are the ability to cause fleeing, attraction and execute individual generic text-actions to be parsed by the executing environment. For example, when an explosion is triggered in the application, the engine is told to make all agents nearby flee from the explosion and also to set their action as “Scream”. This text can then be parsed by different graphics-systems. A full-fledged 3D-engine [25] might show the agents opening their mouths and playing a sound file, while a simpler engine might just print the text “Aaaaah!” above their heads.

### 3.2.1 Example of Uses

To give an idea of what the scripting language can be used for, a few examples will be given. In this, the theory-section, only general situations are given and something said about what needs to be accomplished. In the implementation section 4.3.1 more detailed examples will be given, with script code to show how to accomplish this kind of situations too.

#### 3.2.1.1 Walking

The first and most basic thing that one may want agents to do in a virtual world is to walk around. A simple example is this town square. Here we might want some different things for different people. Some should walk to the ice-cream stand to buy ice-cream. Some should walk around randomly but plausible. Perhaps someone should go to a specific point to wait for a friend. People might walk in groups or alone.

To be able to facilitate this we have three kinds of walking in this example; “Walk along this walk pattern”, “Walk to this named position” and “Walk to a specified point”. All of this can be seen in the following picture.
Figure 11 An advanced example of desired walking

Here we see several agents (green circles), some in groups and some alone. The red dotted line denotes that this group is heading to a named position, namely the “Ice-Cream”-point, where the ice-cream stand is. The yellow slash-line shows that this lone agent is heading to a position, specified with an (x,y) notation. And lastly the network of black lines shows a walking pattern. Agents can begin following this pattern. When they come to an area where one line ends, and one or more other extends from, they’ll select one of the new lines and follow it. This is similar to the points the ViCrowd system [6] uses called “Interest Points”.

This means that for example at the point “P1”, when an agent comes here following the line in from the left, it can choose three of the possible directions extending from this walk point. Either it goes up along the square, down to the road or into the store. Walk patterns means that the agents will walk randomly, but still within logical limits and paths.

3.2.1.2 Events and Actions

But walking is usually not everything that is interesting when it comes to crowd. It might also be practical and useful to be able to have continuous or triggered events and actions. One such example might be when a person pulls out a pistol in the middle of a street, causing panic.
Figure 12  Fleeing in a street

Here we see a person pulling out a pistol (the light orange dot) which causes everyone else (the green dots) to run away from the person with the gun. When the action is performed to take out the guns, a fleeing motion is applied to all the other agents, while the person with the gun sends a text message to its graphical representation that it should show the gun.

These too are inspired by the ViCrowd [6] project, especially their action points and their abilities to cause attraction, repulsion and parsed actions such as in their form of definable action parameters.

3.2.1.3 Continuing Actions

Sometimes more subtle actions are needed. It can be something simple as changing posture while walking. To accomplish this, agents can have behaviors with actions that are triggered with a certain chance at certain intervals. One such thing could then be to trigger a “Posture” action every 120 seconds with a 40% chance of triggering. When the graphics-engine gets told the action has triggered, it changes the agent’s posture.

3.2.2 Objects in the World

What are actually represented in the world are three basic types of entities. These are the objects that can get behavior applied to them, get told to execute actions and walk patterns (only agents and groups can move). All interaction in the world goes through these by giving different orders or behaviors.

- **Agents** – The individual agents. These can be part of a group or be free entities. They are the basic part of the engine and the only objects that move even though the groups move by having the agents in them moving.
- **Groups** – Groups. The groups consist of agents and can be directed around and made to perform actions just like a single agent. A group can exist in the world
even though it is empty of agents for reasons that will become apparent later, it can however not move as long as no agents are included.

- **Event points** – An event point is a position in the world coupled with a radius of effect. Event points can be used for such things as spawning agent, deleting agents, executing an action on all agents in range.

One thing to note here is that there exists no kind of object that allows for generic moving objects. The only moving objects that can get actions applied to them are agents or groups by extension; event points are stationary. However, it should also be noted that since event points can be created and deleted dynamically they can change position during the course of execution, they just can’t move continually.

The idea of using points as focal for many events comes from the ViCrowd [6] system, but instead of coupling them with regions as they do we use a radius on the point. By using many semi-overlapping points with different radius but the same kind of event similar regions can be created in a simple manner.

### 3.2.2.1 Agents

Agents are the basic unit of all scripting and behavior. One agent represents one entity in the world. In this prototype work an agent primarily represents a human, or at least a humanoid. Together with other agents they make up the groups that are the lifeblood of the crowd-system. Since we decided that a group always moves together an agent can only be a member of one group.

In the end all of the behaviors, actions and walking is applied on the agents. As said before, it’s our belief that some very simple basic “axioms” of behavior are needed to mimic more advanced patterns. The only basic information that an agent holds is a position, a velocity and additional information for the pathfinder, a possible behavior, where it’s going and when it last performed its actions.

### 3.2.2.2 Groups

Groups are a collection of agents that try to stick together and walk together. At most times the agents in the group also share behavior. This work has been targeted towards applications where most of the agents are part of a group and not free agents.

From a behavioral point of view a group always tries to stick together and any action or behavior applied to the group is normally applied to all of the agents in the group. A group basically holds no information which its members are.

One question was the definition of a group, since it can be defined in many ways. Was a group other agents you walked with? Shared behavior with? Occupied the same area? For example, should all the people walking on a street be considered one large group with possible subgroups or should we just use the actual groups, such as friends walking together, et cetera.

In the end it was decided to go for the outlook that a group is people walking together fairly coherently. The reasoning behind it was that the benefits from larger group could be compensated for or achieved through other means with the smaller more coherent groups. Applying the same behavior for crowds consisting of many smaller groups is easy due to the way behaviors are specified separate from the groups that use them. Group-relative path finding could be compensated with the help of more interacting agents and groups in the path finder.
In hindsight, it might have been a good idea to have a hierarchy above the group stage, such as the crowd-object used by the ViCrowd [6] system. This because there might occur situations when it’s desirable to apply a behavior or action to a large group of people even if they’re not walking together or members of the same small group. However, any effect that could be done like this can be done with the current system, it just requires more work.

Also of note is that our systems “atom”, basic unit is the agent while ViCrowd has groups as the basis of the system.

3.2.2.3 Event Points

Points are static positions in the world, fulfilling many needs. They specify a radius of influence and a rate in which they should be activated. Event points can be an action point creating certain behavior in the agents around it, causing them to flee, making them perform some kind of action, et cetera. It can be a walk point which is a part of walking patterns or it can be a point to spawn or delete agents.

Event points that move require a separate path finding engine that doesn’t use social models to guide movement. For example a car does not path find based on how it is looked at by other cars. Most of the things it could be used for can also be achieved by having a specially tagged agent. One situation where moving event points could be useful is when there is a need to activate an event on an external actor in the world, one which is not handled by the crowd-system.

3.2.3 Spawning and Deleting

One of the more important parts of a system for semi-autonomous entities is the ability to easily create agents. We chiseled out four different ways it could be interesting and useful to spawn agents in.

- **Single, named, agent.** Creating a single agent with a specified name gives absolute control over where it will be created, how it will behave and handled.
- **Anonymous agent spawned together with a group.** This enables creation of a large number of agents as part of a group.
- **A spawning point, spawning an agent every X milliseconds while activated.** This enables a seemingly endless supply of agents. One use is to have agents spawn out of sight for the user, then walk past and then be deleted when out of sight. This gives the impression of unique and endless flow of people.
- **Spawn an anonymous agent alone in a position.** This can be used if you just need anonymous, non-active agents, for example as spectators to some sport, or if you need to spawn more anonymous agents into an existing group.

Similarly, there exists two distinct ways to delete objects from the world.

- Explicitly deleting an agent, group or event point by referring to it by name.
- Deleting an agent because it gets within the radius of influence of a delete point when the point is triggered.

Once again, we see here the focus on groups. A once spawned anonymous agent can only be deleted or get its behavior changed as a part of a group or by being within the active radius of a point. Most of the methods of spawning allows for an agent to immediately join a group.

As is realized by this, only the named free agents and groups can be active actors in the world. More important entities in the world should therefore preferably be named.
The simplicity of spawning a new agent is important, to be able to keep an area filled with people. Many games and applications use extremely simple ways of adding or removing agents, often fading them in and out in plain sight. This gives a rather surreal and unrealistic feeling to the peoples and crowds. We hoped to facilitate more advanced spawning and deletion schemes to alleviate this.

An idea was to use a control of how far from an active user the agent is and if it is out of view it can be deleted and then re-spawned when the user approached but before it is within sight, we will cover concept more thoroughly in the Level of Detail chapter.

Here we also differ from many system such as GTA3 [22] were spawning and deleting is done in plain view of the user and without any concern for consistency. We also differ from the ViCrowd [6] system in that we have a strategy in spawning and deleting whereas their groups just plain exist.

3.2.4 Controlling Walking

When you have a crowd, you most often still want them to move somewhere. We decided on two methods of guiding paths.

- **A direct method.** Simply telling an agent or group where to go next.
- **An indirect method.** By using networks of walk points a group or agent can walk around the world without need for direct guidance.

The direct method is rather straightforward. By specifying a position and a speed an agent or group can be made to move to the position.

The indirect way is more interesting. It was conceived because we wanted a way to allow for getting agents to follow other agents or walk in certain areas without requiring constant interaction from the executing environment constantly giving them new walk-directives.

A single walk point is very simple. It is defined with a position and a list of other walk points and the chance to walk to these points. An agent or a group can then be told to walk to a specific walk point by name. Once there they will go through the list of linked points and if a random number is lower than the chance of that point they will start walking towards it. When they arrive the procedure begins anew. Note that it is possible to link nowhere or giving the last point a chance of less than 100 percent units thus gathering agents that end up there.

By having long circular networks of these points agents can walk seemingly at random while still staying within an area or certain paths. One can also put walk points in all the interesting points on a map and have them linked to enable agents to walk freely between these points.

The advantage of this over just having the agents actually walking randomly is that they will appear to always be headed to somewhere sensible. Real humans seldom walk at random and it is often easy for us to notice when a computer controlled entity is. Instead by letting the entities walk between known good points they appear more realistic.

On top of these ways to control walking, actions can trigger fleeing or attraction in agents, which causes them to go to other locations, more on this below.
3.2.5 Behaviors and Actions

Behaviors and actions control agents on top of their movement. Some actions can cause changes in their movement patterns. Behaviors decide the overall acting of an agent or group. It controls how easily startled or attracted they are and can also consist of several actions.

An action in turn consists of a specific action. This can be to cause flee or attract, to cause joining or leaving a group or to set a text-action on agents. An action can be part of an agent’s behavior, it can be triggered as a one time event or it can be induced by being within range of an action point. We will deal with all of this in more depth.

To begin with behaviors, they are applied to either a lone agent or a group and have three basic qualities

- **Flee-modifier.** This modifies the chance of group or agent to flee when exposed to something inducing a flee reaction.
- **Attract-modifier.** This modifies the chance of a group or agent to be attracted when something is tempting them.
- **A list of actions,** specified with the chance and rate of occurrence. Each time a certain amount of time has passed (the rate) indicating it’s time to evaluate an action again, the group or agent with the behavior executes the action if a random number is lower than the chance. For a group this should be done individually for each member.

Although this is very simple and might be expanded for real-world usage it works quite well. One thing to note that especially outdoors, in cities or other areas where large crowds are common it is often rare to do anything more advanced than walk around. Except for actions such as shopping at a merchant in a town square or equivalent the only actions taken are in unusual situations, such as in fear or interesting happenings which the flee and attract modifiers covers nicely.

The actions are mainly thought to be used as “monotone” breakers to increase distinctiveness in different agents. Since an action can contain a free text component it has very versatile uses. They can for example be used for changing posture or something similar at certain intervals.

Actions have a bit more qualities than behaviors, but some of these are grouped and not allowed simultaneously.

- **To cause flee, attract or deletion.** When executed the action can cause agents in the area around the executor to flee from the executor, to be attracted or to be deleted. These are mutually exclusive.
- **Join or part a group.** When executed either the agent, the group or all agents in the area joins a specified group or part their own. How this is applied depends on how the action is executed.
- **A parsed action.** This means that either the executor self or agents in the area all get a certain text set as their parsed action. This action can then be retrieved by an executing environment and parsed as appropriate. The text can specify which physical action should be performed like “Waving”. It can also specify something more intangible like that the agent has a bad headache that should result in changed behavior.

Actions can be executed in many ways. We opted for three different ways that we thought covered many of the usages one could have for actions. These are:
• **As a continuing part of behavior.** As specified above an action can be part of a behavior and be executed at certain intervals.

• **As a one time event.** An action can be executed as a one time event. Either on an agent or group, applying its effect to these agents or by being applied to an event point and thereby affecting all agents within the range of the point.

• **As a continuing point.** Certain points can have actions executed at certain intervals just like with behaviors only the action applies to all agents within the range of influence.

By creative use of these three different methods and the options given by an action, especially the parsed text action, most events than can be thought of can be constructed and easily executed.

What we have tried to make is an as possible generic interface. By giving simple but powerful building blocks, more advanced emergent behavior can be created. With the parsed part of actions the actions and events can be interpreted as best suites the environment it is used in, whether this is text, 2D or 3D.

As mentioned above in the examples, much of the inspirations for what needs to be present in an behavioral system comes from the ViCrowd [6] system. However instead of dealing with ratios and divisions within groups (such as “80% happy” means that 80% of the crowd is happy) we deal with statistics and probabilities. By indicating an 80% chance of happiness, on an average, 80% of the group will be happy but there’s not guarantee that at a certain point in time, 80% of the group will be happy.

### 3.2.6 Conditionals

We choose not to include any kind of conditionals or program flow in the script part, instead seeing it more as description of what was instead of what could be. This can of course be debated, and in the long run a move towards a more ordinary script-language might be an idea.

This differs from the ViCrowd [6] system which has been our main inspiration point for the scripting part. A large reason for our divergence here are partly the reasons above but also time constraints.

### 3.2.7 Control versus Autonomy

When it comes to systems such as this, there is always a fine balance between being able to exert control over agent behavior and computer-control which brings simplicity. We’ve tried to strike a good balance between these. The behavior and destinations of the agents can be controlled but the computer handles the lowdown and gritty stuff like planning an exact path.

Having the more fine details being handled by the computer also means that it can more effectively make changes to agents and thereby optimizing resources and speed.

As stated above, our view of the scripts is that they are a specification or description of the crowds at “the now”. After they have been described with behavior, numbers and group-coherency they should be able and allowed to handle themselves. Entities requiring much tighter control such as the user’s companions in a computer role-playing game are better served if they are controlled by another system since they require more detailed behavior and control than this project has.
However, we think the system would work perfectly for the villagers and monsters in such as a game. When the need arises for more interaction with the user, like talking to a villager or attacking a monster, the agents representing those could be removed from the crowd-system and be handled by another engine. This could be done transparently for the user who would never notice anything.

### 3.2.8 Textual Representations of Behavior

At some time, all of these behaviors and actions have to be represented in computer parsable text. It is very easy for a human to react to, adapt to and otherwise interact with different behaviors because we are social creatures with thousands of years of evolution and experience behind us.

Something that is much harder is to quantify and express these behaviors in writing and descriptions. We have in fact several professions just to deal with these, such as psychologists. To put down a persons behavior in a few words, understandable for a computer is hard, very hard.

Any behavior that can easily be described in a script engine like ours will by nature be simplistic. We have tried to focus our energy on the parts where less than perfect behavior is easily identified to a human as wrong. The most important part of the behavior we could identify was how the entities move. Humans can spot strange walking very quickly and on a long distance. The path finder engine tries to emulate human walking while still being fast. As an additional bonus it is easy to put descriptions of movement in text, it is just a matter of telling an agent where to go.

What we would like to have said is that a large part of creating a good system for handling crowds probably has to deal with being able to put behavior into text. Examples of this can be seen in the ViCrowd [6] system.

### 3.2.9 Randomness

As a finishing point on the theory of behavior we want to discuss the randomness present in the scripting.

Humans do very few things at random, probably nothing; instead what might seem random to an observer is most likely the result of a very large amount of different impulses creating emergent patterns hard to spot. However, the computer does not have the resources to simulate that much emergent behavior for one person, let alone for many.

Because of this, something that looks like it is emergent behavior but isn’t, needs to be used. So this is where the element of randomness comes in as substitute. Humans are very good at finding patterns, we find patterns everywhere, even where they do not exist. So even with randomness, as long as there is some kind of underlying patterns, as with the interval of actions, humans will probably spot it sooner or later. This is unfortunate, and perhaps some more obfuscation could be applied at the cost of computer power but it is not within the scope of this project.

Until computers are powerful enough to simulate some very advanced internal impulses and thereby emergent behaviors for many agents we will probably have to rely on an element of randomness.
3.3 Level of Detail

In a huge 3D-world normally a very small part of the world will be displayed on the screen. Since unlike other crowd-systems we are not doing a program for simulation but for games, we want to avoid doing heavy calculations on agents that will not affect the viewer’s impression. This does not mean that all agents outside of the view can be ignored. In several games, such as Grand Theft Auto 3 [22], agents outside and even in the view are removed, giving an unrealistic impression when the viewer turns back and notices that they are gone. Agents far away from the viewer can however be removed without disturbing the impression. Apart from agent removal, several calculations can be reduced when agents are far away or outside of the view. Collision avoidance is not necessary outside of the view as long as agents are not standing in each other when they come into view again. Even when in view agents far away from the viewer do not have to walk very natural to give a realistic impression.

3.3.1 Calculating Level of Detail

For calculating the level of detail for any point a reference point is needed. We use an external object, thought of as the player, as reference point. The level of detail for a point \((x, y)\) is calculated as

\[
LOD = \frac{1}{distance^2}
\]

Where \(distance\) is the distance between \((x, y)\) and the reference point.

The level of detail has to be calculated continuously. We do however not calculate level of detail for individual agents, instead we calculate it for the subregions and then the agents inherit the level of detail from the subregion they are currently in. This is faster than calculating level of detail for individual agents but mostly it is very practical that all agents in a subregion have the same level of detail. Especially when considering if collision avoidance should be used. This way no problems will arise from some agents using collision avoidance whilst the agents they are trying to avoid are not. When calculating level of detail this way it will not be continuous. As stated this is an advantage for the movement algorithms but could be a serious drawback for the graphics. Often the subregions will be quite small, lessening this drawback. If this is still not sufficient the graphics could calculate its own level of detail using another model.

3.3.2 View

For many decisions regarding level of detail it is of great importance if an agent is within view of the user. To be able to determine this quickly there is a need to somehow be able to pre-determine what areas are visible from a certain point. As previously stated we do not care for individual agents, so what we are interested in is what sectors can be seen from a specific sector.

By using a variant of a potentially visible set (PVS)-algorithm [4] we can easily calculate which sectors can be seen from a given sector. With the help of this we build a small BSP-tree [4] for each sector that can be used in real-time to decide which sectors can be seen considering the current facing of the user.

The algorithm used to create these trees can be explained as follows:
For each sector $S$
For each portal $P$
  Add the new sector $O$ to $S$'s list of visible sectors
  Create two vectors, one along each wall out from the portal
  For all portals in the new sector
    If the portal is at least partially within the two vectors
      Use the old portals corners/intersect points and the new to create two new vectors. Recurs into the next sector with these vectors.

We'll give an example to clarify this. Our starting situation looks like this:

![Figure 13 The starting state of sectors and portals](image1.png)

The pink lines are denoting portals, and the named squares sectors. Say we are going to find which sectors we can see from A. We begin by going through all of A’s portals, in this case just one. Here we put two vectors, one straight left and one straight right.

![Figure 14 The first two vectors from the first portal](image2.png)

The reason the vectors are chosen like this is because we want every portal in B to fall within the two vectors, since by standing on the portal-line between A and B, a user in A
can see all the portals in B. For each portal in B, we draw lines from the AB portal to the
new portal points.

**Figure 15 New vectors constructed from the portals’ corners and intersections**

The method to choose which of the portal points to use and how to build the vectors will
be dealt with later. We can already now say that they are built to ensure they encompass
the whole area that can be seen in the next sector. To complete this example, we’ll
concentrate on the BD portal. We take the newly calculated vectors and position them at
the portal points. Note that we do not really care about the vectors lengths, only their
direction.

**Figure 16 The newly constructed vectors applied**

Now we once again go through all the portals in the active sector, D. As we can see, the
one existing portal is not within the vectors, so recursion stop and D is the last sector
added to this branch of the recursion. The full recursion will also add C. So in the end we
get that from sector A we can see B, C and D but not E.

This is not the normal way of applying a PVS-algorithm where normally several random
points are chosen in each sector, and then the view is rendered in 6 orthogonal views.
However we find that the method we use is exact enough since we only need which
sectors we can see and is much faster and easier.
These sectors are then, as they are found, added to A’s BSP-tree. Sectors are inserted into
the tree with a normal BSP-tree partitioning strategy with the exception that only strictly
vertical or horizontal partitions are made. At the moment partitions are made in the middle
between two of the walls between sectors, a wiser choice would probably be along one of
the walls.

When choosing the points on which to base the next vectors all the different combinations
between the four points (in reality, only two combinations are possible, with the vectors
crossing or without) are tried. The longest of them is chosen. We haven’t mathematically
proven that this yields the correct result, but empirical studies seem to confirm it.

3.3.3 Population Control

The main usage of the view is the ability to spawn and delete agents when the user isn’t
watching. The thought is that when the user is over a certain distance away and a sector is
out of view, the system can delete the agents in the area and thereby conserve resources.
As the user approaches, but before he comes within view, the system can spawn agents
anew, and thereby the user will never notice that the sector was empty. Another check that
could be useful is to not spawn more agents even if some have been removed if the area
already contains over a certain amount.

Unfortunately, due to time constraints, the last part of this spawn and delete system was
never implemented and tried in practice, however sector visibility and BSP-tree building
is complete and all the building blocks for it available and tested.

3.3.4 Reducing Calculations

Agents outside of the view or far away do not have to act as natural as agents with a
higher level of detail. They still have to continue moving so that the viewer’s environment
is not static. However, they do not have to avoid collisions as they will not be noticed by
the viewer. Neither do they have to be updated as often as agents with a higher level of
detail.

When agents that have not been using collision avoidance are about to come into view
they must be pushed apart so that they are not standing in each other. Using our collision
avoidance algorithm a collision is detected when an agent checks its path for collision
avoidance. All agents that come into view must check their paths as they will start using
collision avoidance and this also gives the current collisions. We then use a simple
collision handler that moves the agents apart.

Updating of agents can also benefit from using level of detail. The viewer will not notice
small movement by agents far away and it is not necessary to update these agents or the
agents outside of the view as often as agents with a higher level of detail. Since these
agents are not using collision avoidance the resource benefits of not updating them are
however relatively small.
4 Implementation

In this chapter the code and implementation will be gone through. First there will be a short introduction how the general system works and updates and then delve into more specialized sections.

Section 4.2 is devoted to movement and how the path finder works. It is discussed how the world is represented internally and how the agents navigate within its constraints. Also discussed is how the collision avoidance is solved in code to be within reasonable speed and memory constrictions.

Section 4.3 deals with scripting and applying behavior and actions to agents and other objects. Script syntax is explained and exemplified to show how it works and why it is constructed the way it is.

In section 4.4 the details of how level of detail can be implemented and what needs to be handled to get satisfactory results are described. It is also mentioned how the BSP-trees built for viewing can be used to determine if a sector is within sight.

Section 4.5 is about how the crowd-system communicates with an executing environment and how information is passed between them.

The last section, 4.6, is about problems that have been encountered concerning the implementation and eventual solutions to these problems found.

4.1 System Overview

The crowd-system should be as independent as possible. It should not be dependent on the implementation of graphics that is used to represent the agents, nor should it be dependent on too many external libraries. In this we have somewhat succeeded. In the communication with the program only a few classes are used, all of them either implemented by us or being a part of STL [27]. In the crowd-system we do use some minor Crystal Space [29] classes and methods. The idea is that private replacements of these classes and methods should be implemented; this has not been done yet.

An application using the crowd-system communicates mainly with the crowd-system through means of scripting, however, one function is very important to call during execution. This is the Update function.

The program that uses the crowd-system has to call Update with the elapsed time since last call, for the crowd-system to update itself. When update is called the crowd-system works as follows:

The scripting is updated, this is dealt with below

For each subregion with agents in
  Calculate LOD for subregion
  For each agent in subregion
    If needed
      Update agent path
      Update agent position
    If agent position is in new subregion
      Set agents current subregion to the new subregion
      If the new subregion was empty
        Add the new subregion to the update-list
      If the old subregion is empty
        Remove the old subregion from update-list
The elapsed time is used for updating behaviors and walk patterns and for calculating how far the agents have moved with their current velocities. Updating of agents is done in subregion order; all agents in a subregion are updated before the agents in the next subregion are updated. To not have to call update on empty subregions an update-list of all subregions with agents in are maintained.

4.2 Movement

The most important consideration when implementing the movement is speed. In theory path finding and collision avoidance are clearly distinguished areas, in our implementation they are not as separated. Henceforth, when discussing implementation we will include all the parts of the movement in the word pathfinder.

The goal of the pathfinder is to control the paths of all the agents in the system. To do this, information of all dynamic objects are needed; agents as well as external objects. Agents and external objects do have several things in common such as a position and a velocity. We therefore have a base class for dynamic objects. This class is inherited by separated classes for agents and external objects.

The classes have all the information that the pathfinder needs to find a good path. This includes position, velocity and the time to keep the velocity. These are not the classes that represent agents and external objects in the external bindings, for this we have separate classes that act as a link between the environment and the system. This way we prevent external sources from changing the internal variables.

If the agent has a new final position to walk to
    Find the last sector
    Find the next portal
    If next portal is found
        Find a point on the portal to walk to
    Else
        Move to final position
If the agent has orders to wait
    Check if object still is blocking
    If agent has to wait
        Agent velocity = 0
        Calculate the time to wait
    If agent is moving
        Calculate the velocity to reach the goal
Collision avoidance check
If agent will collide with an object
    If agent is close enough to goal
        Stand still here
    Else Do
        If angle between velocities is not to small
            Walk as far as possible then wait
        Else If object is blocking the point on the portal
            Find new point on the portal
            Calculate velocity to reach this point
        Else Do
            Find a detour from original path
            Check so detour is valid
            Until detour is valid
            Collision avoidance check
        Until no collision detected

If agent is moving to new point
    Inform all followers
4.2.1 The World

To represent the passable space we use sectors connected with portals. A sector is represented by a class that contains the area of the sector and a vector with links to all of its portals. The portal class consists of a line and links to the two sectors that it binds together.

The positions of the sectors and portals are read from a file before any path finding can be done. First all sectors are added to the world-object. Every sector receives an ID starting at 0 and increasing. Then when the portals are added, based on the portal’s position we find the two sectors that it binds together and add the links between the portal and the sectors.

4.2.2 Finding the Next Portal

It is the responsibility of the world-object to find the next portal given a start sector and an end sector. To do this a 2D table $P$, which gives the next portal, is calculated using the Floyd-Warshall algorithm. We start by creating a 2D array $W$, in which the value $(i, j)$ gives the weight between the sectors with ID $I, J$. If the sectors have a common portal we use

$$\text{weight}(i, j) = |\text{Sector}_{I_{\text{diagonal}}}| \text{ otherwise } \text{weight}(i, j) = \infty$$

When sectors $I, J$ have a common portal we also insert a link to this portal at $P(i, j)$. The weight from sector $I$ to sector $J$ is the longest possible distance from a point in $I$ to $J$. Other weights are possible; we find that this works fine. The Floyd-Warshall [3] algorithm is then:

```java
for(int k = 0; k < sectors.size(); k++) {
    for(int i = 0; i < sectors.size(); i++) {
        for(int j = 0; j < sectors.size(); j++) {
            if(W[i][j] > W[i][k]+W[k][j]) {
                W[i][j] = W[i][k] + W[k][j];
                P[i][j] = P[i][k];
            }
        }
    }
}
```

After this $P(I, J)$ will contain a link to the next portal to use when moving from sector $I$ to sector $J$. The Floyd-Warshall algorithm is $O(n^3)$ but it only has to be done once. Finding the next portal between sectors $I, J$ is $O(1)$. One consequence of how we find the next portal is that if there are two portals between sectors $I, J$ the same portal will always be chosen. This can easily be avoided by splitting one of the sectors into two.

4.2.3 The Grid

For dividing the world into subregions we are using a grid. It is implemented as a 2D array with pointers to square-objects. Every square in the grid represents a rectangular area. The grid has two functions

- **Dividing agents:** Every square keeps track of which dynamic objects that is in the area currently. This can be used to find all agents in an area fast and is used when checking for collision avoidance.
- **Finding sectors:** Every square also keeps track of the sectors that are in the area. This can be used to find in which sector a point $(x, y)$ is located.
Since the grid is static and all the squares are of the same size it is very fast to find the square in which a point \((x, y)\) is located. When an agent is ordered to move to point \((x, y)\), to find the next portal using the world-object the sector containing the point \((x, y)\) must first be located. This can be done with the grid by finding the adequate square and then looping through the sectors that is in that area. In theory, squares could be very large and sectors very small, making this slow. In practice, sectors are usually larger than squares so that a square normally contains only one sector and at maximum four.

### 4.2.4 Finding a Point on the Portal

To avoid passing outside of the portal, the part of the portal that is used is actually lessened by the agent’s radius at both portal ends. When groups walk through portals it is desirable that the group members do not have to change their relative positions. The leader must therefore choose a portal point that allows the group to pass through the portal on both sides. For the leader the used part of the portal is therefore lessened by the `groupradius + leaderradius`.

How we find a reference point is described in the theory section 3.1.1.2. When the reference point is located the intersection between the lines Portal and reference point – agent position is located. This is done using simple methods in crystal space. If the intersection is not located on the portal the closest path to the reference point is through passing the portal at the edge closest to the intersection.

If another dynamic object is standing close to the portal point, blocking the way, we must find a replacement point. This is easily done by choosing a point that is sufficiently far away from the blocking agent.

### 4.2.5 Collision Avoidance Check

When checking for future collision between two dynamic objects we use the method described in the theory section 3.1.2.4. The collision avoidance check between two agents can be implemented very fast. When an agent changes its velocity this check has to be carried out against all nearby dynamic objects. An agent checks for collisions like this:

```plaintext
timeUntilWait = timeOfAction - timeOfNextAction
For each neighbor subregion including this
  For each dynamic object in subregion
    Check for future collision against the dynamic object
    If tmax > 0 and tmin > 0 and tmin < timeUntilWait
      //future collision detected
      waitingFor = the dynamic object
      timeUntilWait = tmin
    Else If tmax > 0 and tmin < 0
      //collision at present time detected
      Handle physics
After this check waitingFor and timeUntilWait will contain the dynamic object, if any, that is blocking the agent’s path and the time until the collision will occur. The agent can keep its current velocity for this period of time before it will have to wait. This test is also carried out when an agent stops as it then changes its velocity to zero. When the agent has zero velocity it can not avoid other dynamic objects, instead it will have to warn the agents it will collide with that they has to update their paths. Similarly, the pathfinder has no control over the movement of external objects. When external objects are moved the same algorithm as above is used to warn agents that they must update their paths to avoid collision with the external object.
```
4.2.6 Waiting

An agent detects if, when and for whom it has to wait in the collision avoidance check. When it is time to wait, a new control is carried out against the dynamic object to wait for. The reason for this is that the object might have changed its path making waiting unnecessary.

If the angle between the velocities of the agents is too small or too large, waiting does not work well. When this is the case, as when the blocking object has zero velocity the agent will have to detour.

4.2.7 Detouring

When detouring, we calculate a new direction in which the agent can move for a while and then try to move straight for its goal again. To find this direction we first calculate the position of the blocking object at the time for the collision. As we know the velocity of the blocking agent as well as the time for the collision this is done easily. The new direction is then calculated towards a point perpendicular to a line between the agent and the future position of the blocking object, located so that the agent passes the blocking agent at a distance that is larger than their radiuses.

![Figure 17 A new direction is calculated to avoid collision.](image)

Sometimes several objects will be blocking the way. When this is the case, situations may arise where the agent will first try to detour left of the first object and then right of the second, having to detour for the first object again. To avoid this we have introduced drifting. Basically if an agent detours to the right of an object it will continue detouring to the right of all objects until it has found a straight path towards its goal again. The same applies when the agent detours to the left. This effectively solves the problem of several blocking agents.

When an agent detours, a check must be carried out so that the agent not detours outside of the passable space. We have implemented this check so that an agent does not detour outside of its current sector. It would be better if the agent could detour as long as it is inside any sector but with our implementation this leads to problems with keeping track of the current sector. The drawback of this is that detouring options close to portals are more limited.

4.2.8 Flee/Follow

To support orders somewhat more advanced than “move to point (x,y)” we have implemented methods for agents to flee from points and to follow other dynamic objects.
When an agent is ordered to flee from a point, a position located in the other direction is found and the agent is ordered to move there.

When an agent is ordered to follow another agent the same methods that are used for groups are applied. When an agent is ordered to follow an external object these methods can not be applied with very good results, as players often walk quite unpredictable. Instead when the external object moves away from its current position the agent will try to move to the new position of the external object.

4.3 Scripts and Behavior

The implementation of the script handling and behavior control as described in the theory chapter is split into two distinct parts.

- **Loader and parser.** This part loads a block of script commands to parse and execute.
- **Update.** This part handles the continual update of behaviors and actions and is called at frequent intervals during execution.

The parser is seldom run, and then only to parse a new piece of information, for example the definition of a new group, on the other hand the update part is supposed to be called frequently during execution. To understand how these work we have to begin with some examples and their corresponding scripts.

4.3.1 Usages and Examples

To give an idea of how these basic axioms of behavior can be used, some different scenarios and examples will be given. We’ll begin with a simple walk point example containing three walk points.

![Figure 18 Simple walk point example](image)

The script setting up this layout of walk points look as follows:

```plaintext
DEFINE WALKPOINT Alpha
POSITION 10,90
POINT 90 Beta
```
DEFINE WALKPOINT Beta
POSITION 50, 10
POINT 30 Alpha
POINT 100 Gamma
END DEFINE

DEFINE WALKPOINT Gamma
POSITION 90, 50
POINT 100 Alpha
END DEFINE

This means that at Alpha, it’s a 90 percent chance to go to Beta and 10 percent to stop there. Once at Beta there’s a 30 percent chance to go to Alpha and 70 percent chance to go to Gamma((1.0 – 0.3) * 1.0). An agent arriving at gamma always walks to alpha.

To set an agent walking on the trail a command like DO WALKPOINT TheAgent Alpha 10.0 2.0 could be used. This would start by getting the agent walking towards alpha, and then once there it would be given a new walk point.

In a more advanced example we have a position that should resemble a normal street. At the pavements side is the entrance to a mall. As people walk by, some agents should walk into the mall and get a shopping behavior previously defined.

![Figure 19 The mall example](image)

The setup consists of four points; three walk points marked in red and an action point marked in blue. The green circle denotes the action points’ field of influence. It works by having the agents walk by outside on the pavement. Some of them will come within the action point’s influence. These will leave their current group and join a group of “MallVisitors” and be given an order to walk to the M0 walk point. Once there, they begin their travels through the mall.

As noted above at the explanation of action syntaxes, the ability for actions to cause agents to walk to a point would be a good thing to have. Unfortunately this does not exist which creates the need for the outer program to call a DO WALKPOINT on the new arrivals every now and then. The script for the whole construction looks like this.

```plaintext
# The walk points

DEFINE WALKPOINT M0
```
POSITION 10,50
POINT 50 M1
POINT 100 M2
END DEFINE

DEFINE WALKPOINT M1
  POSITION 10, 10
  <some other points, deeper in the mall>
END DEFINE

DEFINE WALKPOINT M2
  POSITION 10, 90
  <some other points, deeper in the mall>
END DEFINE

# The group for mall visitors
GROUP MallVisitors
  BEHAVIOR ShoppingSpree # We assume this behavior has been previously defined
END GROUP

# The action for the action point
DEFINE ACTION JoinMallVisitors500
  PARSEDACTION GoesIntoStore
  JOIN MallVisitors
  WALKTO M0 # Make them start walking into the store
END DEFINE

In one last example we’ll see a one-time action. We have a town square with some barrels standing; when the barrels explode people should flee and scream.

![Figure 20 An example with an explosion](image)

This is accomplished by having an event point in the middle of the barrels without an attached event but with a large radius. When we want the explosion to happen, we trigger an action once on the point, which then gets applied to all agents within range. The script for this looks like this

POINT Barrels 35, 65 45 0

DEFINE ACTION Explosion 4000 # The 4000 means people will scream for 4 seconds
  CAUSEFLEE
  PARSEDACTION Scream
END DEFINE
4.3.2 Script Syntax

We begin with an inclusion of all the script syntax and then walk through it one statement at a time.

- `<variable>` indicates that a text, number or equivalent should be in that place.
- `[^<variable>]` indicates that what follows within the [ ] is optional. These can be followed with a star (*) indicating that the optional parts can be repeated 0 to 8 times.
- An expression of the form (`<option1> | <option2>`) indicates that one of the two or more options should be chosen.
- Anything following a # to the end of the line is a comment.
- Any one line can be no longer than 512 characters.

AGENT `<name>`
STARTPOS `<position>`
[BEHAVIOR `<behavior name>`]
END AGENT

GROUP `<name>`
[AGENTS `<name1>`, `<name2>`, ...]
[SPAWN `<number of agents>` `<position>`]
[BEHAVIOR `<behavior name>`]
END GROUP

POINT `<name>` `<position>` `<radius>` `<milliseconds between activation>`
[<action, spawn or delete name>]
DO POINT `<name>` (ENABLE | DISABLE)
DO PERFORM `<name>` `<action name>`
DO SPAWN `<position>` [GROUP name]
DO DELETE `<group name>` | `<name>`
DO GOTO `<name>` `<position>` `<speed>` `<acceptable distance>`
DO WALKPOINT `<name>` `<walk point>` `<speed>` `<acceptable distance>`
DEFINE WALKPOINT `<point name>`
POSITION `<position>`
[POINT `<chance` `<name>`]`
END DEFINE

DEFINE BEHAVIOR `<behavior name>`
[FLEE `<modifier>`]
[ATTRACT `<modifier>`]
[UNDELETABLE]
[<action name> `<chance` `<milliseconds between activation>`]
END DEFINE

DEFINE ACTION `<action name>` `<length>`
[<PARSED ACTION `<text>`]
[JOIN `<group name>` | PART]
[WALKTO `<walk point>`]
END DEFINE

DEFINE SPAWN `<spawn name>`
[JOIN `<group name>`]
END DEFINE
As can be seen there are three major groups of statements:

- **Object definitions.** This includes the Agent, Group and Point statements.
- **Intangible definitions.** This includes definitions of Actions, Behaviors, Spawns, Deletes and Walk points.
- **Immediate statements.** This consists of commands such as setting an event point’s state, performing an action, spawning or deleting objects and finally making agents and groups walk.

We will deal with each of these groups in separate sections where their usage and results are explained. General things to note is that all scripts are evaluated linearly, which means that a group can not use named agents that have not already been specified.

### 4.3.2.1 Reasons Behind the Language Design

The reasons behind how the script language was designed are simple:

- It should be quick to implement.
- It should be able to control what we need.

These were the only two criteria we had. The language should be seen as a prototype and in a real-world application it should probably be incorporated into an existing script language such as Python or Lua. Since the language has been added to and remodeled as work progressed it is also somewhat inconsequential in parameter-passing and syntax.

The actual look of the language was vaguely inspired by the ViCrowd [6] system but we choose not to directly use a similar system due to several considerations. The foremost of these were the disparities between how our and the ViCrowd system handles and groups agents in groups and crowds especially the fact that a group in our crowd-system always walk together.

However, some inspirations were derived from the ViCrowd system and are mentioned where applicable.

### 4.3.2.2 A Note about Syntax of Positions

Positions in the script can be specified in a multitude of ways. The decimal-separator is “.”, any one of “([,;])” can be used to separate the two different numbers making up the position. Some examples of positions are

- 4.0 -3.3
- (9, 27)
- [-32; 8.4]

### 4.3.2.3 Objects (Agents, Groups and Points)

This group of script commands defines what exists in the world. This consists of the objects that exist; *Agent, Group, Points*. We begin with the most basic unit of the system, the agents. A single named agent can be defined by using the `AGENT` command, structured as follow

```plaintext
AGENT <name>
STARTPOS <position>
```
The `<name>` specifies the name of the agent. It should preferably be unique, if it is not, unpredictable events may occur. `STARTPOS` is always required and should be given a position. If wanted a `BEHAVIOR` can be specified by name. This behavior must be previously defined.

This will create a lone free agent in the position specified. If this position is occupied the system will attempt to place it as nearby as possible until it is too far away from the starting point.

Groups are defined by using the `GROUP` directive. This is the only way to create a group.

```
GROUP <name>
  [AGENTS <name1>, <name2>, . . .]
  [SPAWN <number of agents> <position>]
  [BEHAVIOR <behavior name>]
END GROUP
```

As with agents the name should be unique. Additionally just as with agents a behavior can be specified, it will apply to all agents in the group. What are new compared to agents are the `AGENTS` and `SPAWN` directives. The `AGENTS` keyword allows one to supply a list of agents names (these agents needs to be previously specified) which are added to the group.

The `SPAWN` keyword allows one to give a number and a position and that number of anonymous agents will be spawned and join the group. This allows for large groups without the need for specifying each agent individually.

The last of the objects that can be defined are points. A `POINT` keyword is used to specify this kind of objects.

```
POINT <name> <position> <radius> <milliseconds between activation>
  [<action, spawn or delete name>]
```

As can be seen the `POINT` keyword requires more required variables than the previous two. As before `<name>` should be unique. `<position>` specifies the point where the point itself is located, `<radius>` gives a range of influence for the points events. The next argument gives how many milliseconds should pass before each triggering of the point event. The last argument is optional, and should be the name of a previously defined action, spawn or delete event.

### 4.3.2.4 Intangibles (Action, Behavior, Spawn, Delete & Walk Point)

Intangibles are the group controlling behaviors of agents and groups and the ability to create and delete agents in thin air. The first one of these is `ACTION`.

```
DEFINE ACTION <action name> <length>
  [CAUSEFLEE | CAUSEATTRACT | CAUSEDELETE]
  [PARSEDATION <text>]
  [JOIN <group name> | PART]
  [WALKTO <walk point>]
END DEFINE
```

As before the name should be unique. The length sets how long the action runs. When that time has passed since the action was activated, it will be reset. Note that this just affects the parsed action text; all the others happen one time at activation and are immediate.
CAUSEFLEE and CAUSEATTRACT do exactly what they sound like. CAUSEDELETE causes agents around the object with the action to be deleted, but not the object itself or its eventual group. Setting the part argument causes any agent subjected to the action to leave its’ group, the join argument causes an agent subjected to join the group specified. WALKTO sets the objects subjected to the action walking towards the walk point.

DEFINE BEHAVIOR <behavior name>
[ FLEE <modifier>]  
[ ATTRACT <modifier>]  
[ UNDELETABLE]
[ ACTION <action name> <chance> <milliseconds between activation>]*
END DEFINE

The name is self-explanatory by now. The flee- and attract modifiers should be in the range of 0 to 100 and specifies a chance in percent units to flee or be attracted when subtracted to a force of that kind. If not specified it will be 100. UNDELETABLE specifies if the object with the behavior can be deleted by actions / delete points. Even with this set it can be deleted with a DO DELETE and a specified name. A behavior can have 0 to 8 actions attached to it, with a chance and a rate specified. Each time the number of milliseconds specified has passed, a check will be made against the chance, if it passes, the action will be executed on the agent or group that has the behavior.

As specified, when the length of an action has run out, the parsed action text on the agents will be reset. If a behavior has several actions running, the shortest one will be used to determine reset time. An alternative to this would be to have each agent keep a list of all actions currently executed on it. This could easily be changed if desired.

DEFINE WALKPOINT <point name>
 POSITION <position>
[ POINT <chance> <name>]*
END DEFINE

This defines a walk point. It is given a name and a position, and then a possible list of 0 to 8 other walk points together with a chance and their name. The list of walk points is evaluated somewhat interestingly. An example:

DEFINE WALKPOINT Alpha
 POSITION 0,0
 POINT 50 Beta
 POINT 50 Gamma
END DEFINE

A point like this does not mean that from Alpha the agents walk either to Beta or Gamma with an equal chance. Instead what happens is that the system begins with looking at the first point, Beta. This point does indeed have \( P(gotoBeta) = 0.5 \), the next point however is evaluated only after the first point has been selected or not, so it gets the chance of \( P(1.0 - P(gotoBeta) \cdot P(goto Gamma)) \) which gives 0.25. This leaves a \( P(standstill) \) of 0.25.

DEFINE SPAWN <spawn name>
[ JOIN <group name>]  
END DEFINE

Finally we have the two definitions for defining an abstract spawn or delete that can be assigned to a point. Both needs to be supplied a name. A spawn point can also be given a group that all spawned agents join.
4.3.2.5 Immediate (Do Goto, Delete, Perform, Point, Spawn, Walk Point)

These commands do not as much define things as make previously existing objects to execute some kind of action or event.

DO POINT <name> (ENABLE | DISABLE)

This command simply enables or disables a point. A disabled point will not have its event triggered.

DO PERFORM <name> <action name>

A perform makes either an agent, a group or a point perform an action. In the case of agents and groups the action is performed on themselves. On points, the action is applied to all agents within point range.

DO SPAWN <position> [<group name>]

Spawns an agent at the specified position. If a group name is supplied, the agent will join that group.

DO DELETE <name>

Deletes an agent, a group or a point. Only named agents can be deleted this way if they’re not part of a group.

DO GOTO <name> <position> <speed> <acceptable distance>

Makes a group or agent go to the specified position at the specified speed. <acceptable distance> gives the distance from <position> where it’s acceptable to stop if the path is blocked.

DO WALKPOINT <name> <walk point> <speed> <acceptable distance>

Works the same way as a Goto except instead of a position, the name of a walk point is given.

4.3.3 Code Implementation

The code that handles scripting and behaviors consists of one large main class representing the script manager and several small objects representing different objects used by the scripting. These are

- **ScriptManager**, handles the loading and parsing of scripts, and also updating them during execution.
- **ScriptObject**, from which all objects that are used by the scripting inherits. Was originally thought to be used just for agents and groups but shifted focus during development.
- **ScriptAgent**, describes an agent. It holds a link to the agent’s behavior, when it last performed its actions and the next walk point. All other information, especially for the pathfinder, is retained by a pointer to the **AgentState**.
- **ScriptGroup** is just like the agent except it holds a **GroupState** instead.
- **ScriptAction**, **Behavior, Delete, Point, Spawn and Walkpoint** basically holds just the information that can be specified for their type as an appropriate data type. For example **CAUSEFLEE** is represented as a Boolean in **ScriptAction**.
4.3.4 How a Script is Loaded andParsed

Scripts are loaded one line at the time, and parsed by tokenizing the line and passing through the parse-system. This system uses a recursive switch system based on the current token. As such, both the scripting language and parser are incredibly simple, this is mostly because the need for a prototype language outweighed the need for a proper language and parser.

4.3.5 How Updating Behaviors Work

When update on the ScriptManager is called it enters into a loop to update all behaviors. Three trees are used to hold the script objects, namely one for agents, one for groups and one for all other script objects. The update runs something like this:

Run through the tree holding agents
   If the agent has a temporary action and its length is past, reset it
   If the agent has a behavior then
      For each action in the behavior
         If it's been more than the rate since last performed
            If a random number is less than the chance
               If action has a parsed part, set it
               If action causes flee or attract then
                  Instill flee or attract in all surrounding agents
               Set time this action was performed
         If the agent is not in a group and has a next walk point set then
            If agent isn’t moving
               If the agent is close to the next walk point
                  Look up the new next walk point and set it
               Else, we have stopped for some reason away from walk point so
                  Start moving towards the walk point again
   After this has been done for the agents, it is done for the groups in almost the exact same way. Lastly the points are gone through as follows.

Run through the tree holding script objects besides agents and groups
   If the object is a point
      Check if it is time to execute the points event, if is then
         Check the point event type
            If it is an action point
               Execute the action, affecting all agents in range
            If it is a spawn point
               Spawn an agent
            If it is a delete point
               Delete all agents in range

When this is completed, all of the behaviors and actions that should be updated are finished.

4.4 Level of Detail

As mentioned in the theory section 3.3.1 agents inherit the level of detail from their current subregion. We are calculating the level of detail value for a subregion just before updating the agents in it. Agents therefore always have the correct value. When calculating the level of detail as \( \frac{1}{distance^2} \) no square root is needed making it faster to compute. The distance from the reference point to the subregion is naturally measured to the midpoint of the subregion.
4.4.1 Reducing Calculations

The level of detail value can be used to reduce calculations by avoiding unnecessary calculations for agents that are far away or outside of the view. We believe that when doing this many problems are avoided by having the same level of detail for all agents in the same subregion. Our results have however showed that the pathfinder also without these optimizations is able to handle a lot more agents than the graphics-engine. We have therefore not implemented them but concentrated on other areas.

4.4.2 View

The creation of the view is implemented almost exactly as the theory goes, so there is not much to say about that. We don’t use the BSP-tree [4] for what it is primarily used, hidden surface removal, but instead to determine which sectors are in view.

The normal application for this is to cull polygons against the view frustum but we will instead be culling sectors. At every node the view should be compared against the plane that partitions the tree. If the culling fulfills certain requirements whole branches of the tree can be discarded as not seen.

By applying the partitioning plane and the view frustum in the following formula
$$\mathbf{N}_v \cdot \mathbf{N}_p \geq \cos^{-1}\left(\frac{\pi}{2} - \theta\right) [4]$$
where $\mathbf{N}_v$ is the view direction, $\mathbf{N}_p$ the normal of the partitioning plane and $\theta$ half the view angle (in many popular games this is 45 degrees). This formula is simple enough to be applied at each update. This particular formula is if the viewer is on the positive side of the partitioning plan. If the viewer instead is on the negative side an analogous formula can be created.

4.5 External Pathways

For the system to be usable there have to be ways of communications between the crowd-system and an external environment. All communication passes through some points of contact in the following classes

- **System** – Creates and manages agents and the world either through direct methods or through loading scripts.
- **AgentState** – This class works as a direct link to the agents. Orders can be given and information about the agent’s current position and action can be retrieved.
- **ExternalState** – This class is to enable the external environment to inform the crowd-system of what the external objects are doing.
- **GroupState** – Used to control groups. Work similar to AgentState.
- **GraphicsObject** – Help keeps a link between an agent and its graphical representation.

We will delve into each of these in somewhat more detail, beginning with System. The main purpose of System is to be the vessel through which communication is opened. This is accomplished by several methods. The most important one of these is the ability to load a script file and have it parsed and executed by the crowd-system. By using scripts all the details of creating and handling agents is handled internally.
However, it is not always desirable to use scripting. The system then offers methods to directly create or delete agents and groups. Used in this manner, the methods to create an agent return the agent’s AgentState so the external program can keep track of it.

When an agent is created or deleted, even if it is done by scripting, the correct callback is called. This enables the graphical environment to attach a link to the agent’s AgentState, with the graphical representation of the agent held in an object that inherits from GraphicsObject. This is done so there is no need to search and match between agents and their graphical representation.

All of the states act as direct links between objects in the world and the external environment. They can be used to give direct orders and to get information about the agents, groups or externals that the states represent. If an agent is created manually, the AgentState is returned, however the best way of getting the AgentStates is to query the system for all states or all states in an area. These are returned in an STL List [27].

Generally, most of the data held by the system exist in common STL data structures.

### 4.6 Problems and Solutions

Some problems in the implementation arose. Most of them were pretty easy to solve, and none of them worthy of any lengthy discussion so we will cover them briefly.

An object-oriented problem was that the ScriptAgents needed to keep a track of the associated AgentState but also vice-versa which creates an ugly dual-link. However this is by far the easiest and fastest way to do it.

We did not want agents to expose too much to external programs yet the path finder needed access to many of the agents’ inner workings. This created the need for the different states such as AgentState which very nicely filled the gap between external programs and agents.

In a similar vein, we wanted the agents to have some kind of link to the graphics that represented them without tying the crowd-system to any particular graphics-engine. Therefore the GraphicsObject was created together with the callbacks that trigger when an agent is created or deleted.

Most of our real problems have been associated with Crystal Space, Cal3D or the Cal3D plug-in. Particularly the fact that the Cal3D plug-in is created for an older version of Crystal Space and an older version of Cal3D, which has forced us to make some changes in the plug-in for it to even compile. Some of the smaller problems with the Cal3D-plugin can come from this. To get around the limitations this created in number of simultaneous agents, we used simpler models and the 2D-graphics implementation.
5 Graphical Representation

We decided early that the crowd-system should be independent of the graphics used to display it. This enables the use of several graphical representations as well as total graphical freedom for the program that uses the crowd-system. To do this in an effective way we use an empty class called GraphicsObject. Every agent has a link to its GraphicsObject. This class can then be used as base class for the graphical representations the program wants to use. More of this in the external pathways section 4.5.

Agents in the crowd-system perform actions that should be represented graphically. These actions are represented by text-strings leaving the graphical interpretation to the graphics-engine and freeing the crowd-system to having to know anything of the graphics-engine.

Although sectors are used in the crowd-system to represent passable space there is no actual connection between the sectors and the graphical representation of the world. It is the responsibility of the program that uses the crowd-system to make sure that the sectors agrees with what is shown on the screen.

We have implemented both 2D and 3D graphics to test the soundness of our design. We will discuss some specifics with those implementations in this chapter, but first it is important to know why and how graphics should be used in conjuncture with the crowd-system to create realistic-looking behavior.

5.1 Help for Realistic Behavior

Graphics can greatly help how realistic agents are perceived to be. The first and easiest way to accomplish this is to have the graphical representation of an agent’s face the direction it is moving. The second way is to have the agent do a walking or running motion suitable for the speed it is keeping.

Other ways is to have a multitude of different looks for the agents so they do not appear to be the same person. Since we spawn many agents, sometimes at random, a method to create unique but still reasonably looking clothes, hair and attribute for agents would be useful.

In reality, almost all humans look unique so a good system to generate individually looking agents greatly increases how realistic agents will be perceived to be. With the advent of MMORPG [26] there are many advanced systems for generating unique characters.

Lastly, having a visual and or audio interpretation of all the possible actions that an agent can display, both for parsed actions and for fleeing and such, would greatly increase how credible they look.

5.2 Aiding Level of Detail

Besides creating realistic behavior, graphics can be important when it comes to support the rendering of as many agents as possible. Level of detail should be taken maximum advantage of. The idea is to not use unnecessary detail; an object that will only occupy a few pixels on the screen does not need to be excessively described. The problem of mesh simplification involves reducing the number of polygons in the mesh to an adequate level. We will not look further into this but note that it is desirable to have a continuous level of detail in graphics. As mentioned the value for level of detail provided by the crowd-
system is not continuous, but since it is calculated fast there should not be much waste of processor time if the graphics-engine calculates its own level of detail approximation.

5.3 Graphics Implementations Used

We tested our solution with two different graphics-engines, a plain OpenGL 2D-implementation and one in the Crystal Space 3D-engine with Cal3D for character animation support. By having a function to give the z-value at a certain point we get a 2.5D implementation that works just as well in 2D as in 3D. It does prevent agents from being able to be in the same spot but different height, but this is very unusual in a city or outdoor environment normally, the only bigger exception being bridges.

5.3.1 2D Graphics

For 2D graphics we used OpenGL [28]. It is simple and easy to use. We see 2D graphics only as a substitute for 3D graphics which is the real goal. Gloor, Cavens, Lange, Nagel and Schmid uses 2D graphics for debugging their crowd-system [11], and 2D graphics is undoubtedly very practical when developing a crowd-system as it gives a good overview. We represent agents with colored circles. Actions are represented by text-strings over the circles.

![2D Graphics](image)

**Figure 21 A picture from the 2D version**

5.3.2 3D Graphics and Character Animation

For 3D graphics we used the graphics-engine Crystal Space [29]. Crystal space is unfortunately not very fast; neither does it give the best graphical results. However it is free of charge.
For character animation we use Cal3D [30], a skeletal based 3D character animation library. To use Cal3D with Crystal Space we use the Cal3D plug-in. From the Cal3D plug-in we get the character models as Crystal Space meshes. In order to quickly link the agents to their corresponding meshes we extend the GraphicsObject with a class containing the mesh.

Many of the graphical speedups such as view culling and portals are handled by Crystal Space. It is important not to minimize the reloading of textures to the graphic-card. Since we are not concentrating on the graphical aspect we are only using a single character model for all agents so textures should not have to be reloaded.

Cal3D has support for creating a core model that is used for all other models. Unfortunately we have not managed to get this to work with the Cal3D plug-in and Crystal Space. Not using this means that every agent will have its own set of textures that will have to load to the graphic-card every time that agent is drawn. Partially due to this we have not been able to use a large number of Cal3D models in our program, however Cal3D is slow in Crystal Space and not really suitable to use for the rendering of many characters. Instead we currently use rectangular blocks to represent agents.

Figure 22 Walking in 3D

5.3.3 Level of Detail in Cal3D

It is important to make maximal use of the level of detail to enhance the graphical results. Cal3D has support for level of detail by reducing the number of polygons used for the character models which to some extent works with the Cal3D plug-in. Unfortunately when used with the Cal3D plug-in the reduction of polygons does not seem to speed up the graphics as much as making the model unrecognizable. One option we have considered is to use 2D billboards for agents with low level of detail. Since Cal3D does only support very few agents the billboards would have to be used at very close distances to give the proper rendering speed and we have therefore not implemented it as the rectangular blocks we use work better.
6 Unsolved Problems

6.1 Cal3D Plug-in

As mentioned above and in the graphics parts of various chapters, Cal3D [30] in combination with Crystal Space and the Cal3D plug-in is slow and does not accomplish what we want. Possible real solutions would be major changes in the Cal3D plug-in or more drastically, changing graphics-engine.

6.2 Boundary-checks on Agents

Our basic assumption that an agent always can walk in a straight line between two portals in a sector is not always true. An example of this can be seen in the following image

![Figure 23 A sector where problems can arise](image)

If the agent in Figure 23 moves straight towards the X, part of the agent’s space will be outside of the sector. When represented graphically this might lead to agents standing in wall or other static objects. We believe this can be solved by better checks in the pathfinder but that the best solution is to design the sectors so that similar situations do not occur. An algorithm that divides unsuitable sectors into several smaller sectors could be added to the project.

6.3 Agent Rotation

When an agent changes the direction of its velocity it has to be rotated. This rotation should be done smoothly. We initially ignored this problem convinced that it would be quite easy to solve later. Unfortunately this was not the case.

The path finder does not support that the agent is standing still to rotate; this would have to be implemented as a part of waiting and would require more collision avoidance checks to be done.

Besides, rotation on the spot does not look natural. Good rotations can be accomplished by using Bézier curves or some other algorithm for smoothing curves and the use the tangent as direction for the agent. This creates a smooth rotation during the whole movement.

With our current way of creating paths as straight lines we do not know any method to create a good rotation. Since any rotation towards the new direction before we actually switch direction will create unnatural strafing. Neither have we been able to find someone else with a good idea on how to solve this. Implementations that we have seen, ViCrowd [6] especially, use the same method as we.
7 Discussion and Final Comments

We believe we have given one answer to our original question

- How does one design a system to handle a large number of computer-controlled agents in real-time while still having agent behavior realistic enough to make the agents indistinguishable from a user-controlled entity?

There are of course several other approaches and answers to this question, but we believe we have investigated our line of answer to a satisfying degree.

We think that it is possible for a player to blend in with the agents in the crowd-system if the player behaves realistically. It is our experience that players often do not use natural walking patterns in games. A situation where the agents can be distinguished by moving more natural than the player is possible. This is only relevant in multiplayer games when witnessed by other players.

We believe as crowds will become more and more common in real-time 3D applications, the users will have to start moving more realistically since that is the most practical way of moving through a crowd (at least while not waving a gun).

We also believe that we can handle an adequate amount of agents at the same time, at least while not being hindered by a slow graphics-engine. With OpenGL in 2D several hundred agents are handled with ease.

Our initial thought that constant movement is necessary to get a realistic look on movement for the agents turned out to be true since this greatly enhances how well they are perceived as humans. Even in a 2D world with small circles representing the agents, a good pathfinder and constant movement goes a long way to give the impression of human, not computer, control over the agents.

As a side note, we would like to heartily recommend the use of 2D as a debugging platform even when developing with the aim of 3D worlds.

Of course, some things could be changed. We have learned that it probably would have been a good idea with another level of hierarchy representing complete crowds. If we have a busy commerce street it would be very practical to have two crowds, one for each direction, so they can have a common walk command and direction while still consisting of many smaller groups, which walk together and act together.

The time to implement other methods of path finding would have been very interesting. We would like to test if they worked better and faster or worse. However we do think that the path finder works well at directing agents in a realistic manner while still being very fast, in fact, it is so fast it will probably be a long time before graphics-speed catches up even with current development.

As we quickly discovered with Cal3D, we are unable to have more than 5 to 10 agents with reasonable speed. With rectangular blocks in their place Crystal Space can handle upwards 200. In 2D OpenGL even more than this is possible. This means that even a 200% increase in Cal3D efficiency would allow only upwards 30 agents, which is far too few.

This is one of the main reasons we have not made much inroads towards level of detail for the artificial intelligence except for agent spawning and deleting, since any optimization there would be wasted compared to level of detail in the graphics engine. Unfortunately,
for level of detail in the graphics-engine to be worth the trouble it will probably have to be faster than Crystal Space to begin with and a good and working implementation of both graphics-engine and character animation library is needed.

This means the initial focus on level of detail were somewhat put to shame. Even despite this the system handles a very large amount of agents.

As a conclusion, we believe that large autonomous crowds will become a larger and larger part of new games and applications as the current development is already showing. We believe that our approach to path finding and behavior is fast enough while being realistic enough to be acceptable in virtually all real-time applications without the need for stringent simulation. Finally, we believe that as more and more of the work load of a game is put on the GPU, more and more research and development will be needed to utilize the CPU for more advanced artificial intelligence and improved worlds more similar to our own.
8 References


[31] Lunds Tekniska Högskola - http://www.lth.se