A Multi-Layered Flocking System For Crowd Simulation

Simon van den Hurk

October 2009

Supervisor: Ian Watson

A disseration submitted in partial fulfilment of the requirements for the degree of Bachelor or Science Honours in Computer Science, The University of Auckland, 2009

Abstract

The field of crowd simulation attempts to model crowd movement of both people and animals. Typical research in this field aims to develop systems which model the interaction between multiple instances of the same type of character. This dissertation examines two aspects of crowd simulation which are often not considered, the movement of crowds containing characters of vastly different sizes and the ability to allow characters to move underneath other characters when there is sufficient space to do so. To include these traits in a crowd simulation model a new system is proposed: the multi-layered flocking system. This system has a basis in the original Reynolds flocking model but includes a series of layers to represent the simulation space. Characters in the simulation are represented using one or more navigation objects which lie upon the layers in the system. This dissertation examines the different representations of characters that are possible using this system. It also examines different representations for the environment in which these characters move. Finally it describes the various types of crowd movement that can be created in the system with these character representations.

Contents

1 Introduction **1**

1.1 Introduction . . . 1

1.2 Literature Review . . . 3

1.2.1 Agents . . . 3

1.2.2 Environment Representation . . . 6

1.3 Summary . . . 7

2 Mutli-Layered Flocking System Design **9**

2.1 Introduction . . . 9

2.2 Navigation Agents . . . 10

2.3 Layered Navigation Cells . . . 11

2.4 Neighbourhood Selection . . . 13

2.5 Navigation Behaviours . . . 16

2.6 Cell and Agent Property Relations . . . 18

2.7 Summary . . . 18

3 System Capabilities and Characterization **21**

3.1 Introduction . . . 21

3.2 Types of Movement . . . 22

3.2.1 Single Layer Character Representation . . . 22

3.2.2 Multi-Layered Character Representation . . . 23

3.3 Properties . . . 24

3.3.1 AffectsBelow Property . . . 24

3.4 Environment Representation . . . 26

3.4.1 Example Properties . . . 27

3.5 Cell Layer Hierarchy . . . 28

3.6 Behaviours . . . 28

3.6.1 Standard Movement Behaviours . . . 29

3.6.2 Multi-Layered Movement Behaviours . . . 29

3.6.3 Integration with Pathfinding Systems . . . 31
3.7 Comparison with Reynolds Navigation Model . . . 31
3.8 Advantages of the Multi-Layered System . . . 32
3.9 Disadvantages of the Multi-Layered System . . . 32
3.10 Summary . . . 33

4 Conclusion 35
4.1 Conclusion . . . 35
4.2 Future Directions . . . 36

Appendix 39
.1 Appendix A: Pseudocode . . . 39
.1.1 Navigation Object Pseudocode . . . 39
.1.2 Navigation Agent Pseudocode . . . 40
.1.3 Navigation Obstacle Pseudocode . . . 41
.1.4 Navigation Cell Pseudocode . . . 42
.1.5 Navigation Cell Layer Pseudocode . . . 44

1 Introduction

1.1 Introduction

The problem of simulating the movement of large numbers of characters in an environment is relevant to both real time and rendering applications being created today. These crowds of characters are used to breathe life into the large and often varied scenes created for a variety of mediums such as games, movies and television. Furthermore the study of crowd interaction can be applied to more serious applications such as the flow of people in architecture design, the testing of transportation infrastructure and the training of military personnel.

In this dissertation two aspects of crowd simulation are examined that are often ignored and a system is proposed that further enables designers to create more complex crowd interaction by removing these limitations.

The first aspect to be examined is the interaction between the characters of a crowd when the size of the characters differs significantly. The majority of techniques described in the field of crowd simulation concern the interaction between multiple instances of the same type of character. They therefore do not consider the effect that character size will have upon the overall movement of the crowd.

The second aspect that is to be considered is the increased level of crowd interaction that can be created when characters in the crowd are able to navigate not only around other characters, but also underneath other characters when there is sufficient space.

To demonstrate these aspects an example is given involving the simulation of an African Savannah. Such a simulation would contain a range of animals from different species that represented the wildlife. Many of these animals would be grouped into flocks or herds of animals and as such would be ideally suited to be represented using a crowd simulation model. These herds of animals vary in size considerably and as such interaction between the animals does not only lie upon a single plane. Even amongst individuals of the same herd the relative size can vary considerably, especially between older and younger members of the herd. These variations in size cause typical crowd simulation models to produce a resulting crowd movement that is contrary to what one might expect. For example the young of a particular species may stay close to its parent, hiding underneath the parent when it feels threatened. With a typical crowd simulation model the child is restricted to the bounds of the parent and the simulation would not allow the child to move underneath its parent.

To allow these two aspects of crowd interaction to influence the navigation of the characters in a crowd a new system is proposed: the multi-layered flocking system. This system represents the characters and environment of a simulation by placing them upon a series of layered cells. This approach is inspired by the representation given in the original flocking paper by Reynolds[14].

The objectives of this dissertation therefore are:

- To define a navigation system that takes into account the size difference of characters within a simulation.

- To allow the characters in this navigation system to navigate not only around but also underneath other characters within a simulation.

- To examine the different representations that can be used for both characters and the environment within this navigation system.

- To examine the types of behaviour possible within this navigation system and the resulting movement which occurs when using these behaviours.

The remainder of this chapter provides an overview of the other literature in the field of crowd simulation. In the following chapter the design of the multi-layered flocking system is explained. The next chapter discusses the use of the multi-layered flocking system and describes its capabilities. The final chapter contains a conclusion for this dissertation.

1.2 Literature Review

The popularity of video games as an entertainment form has led to increased realism in successive titles. This realism includes the simulation of crowds to represent different aspects of the game. Recent games such as *Supreme Commander*[2] and the *Total War* series[21] have included thousands of characters as military units while other games such as Assassins Creed[23] have used smaller quantities of characters moving in groups in an attempt to create a more lifelike representation of a crowded market.

In the movie industry the crowd simulation tool created by Massive Software[9] is notable for producing the crowd animation in many popular movies such as *The Lord of the Rings*. In the field of serious games and training simulators the commercially available crowd simulation tool *AI.Implant* by Pregasis is available[4][13]. This tool represents the agents of a crowd by populating each agent with its own 'brain' that allows it to sense the environment, consider various strategies and them implement one of these strategies.

Character representation in real time simulations such as video games is most often performed using a combination of a coordinate position to represent the centre of the object and a radius to represent the bounds of the object. In a two dimensional simulation the coordinate position is represented by two values that describe the position on the x and y axis. In a three dimensional simulation an additional value is used to describe the position along the z axis. The radius describes either a bounding circle or bounding sphere if the simulation is two or three dimensional respectively.

1.2.1 Agents

The most widely known and popularized technique for large scale crowd movement is the flocking concept proposed by Reynolds[14]. Reynolds' paper presented a framework to simulate the flocking behaviour of animals. The examples provided by Reynolds involved bird-like flocking agents (referred to as boids) that moved through a 3-dimensional environment. The framework is also applicable to two dimensions (as in the flocking movement of sheep).

In Reynolds framework each agent is expressed using a simple point mass model, the direction of which is calculated by using a combination of different steering behaviours.

Reynolds defines three different types of steering behaviours in order to simulate the flocking motion. These behaviours are initially called *Collision Avoidance*, *Velocity Matching* and *Flock Centring* though in Reynolds later works he refers to these same behaviours under the names *Separation*, *Alignment* and *Cohesion* accordingly[15]. Each steering behaviour returns a force representing the desired acceleration of the agent in order to best fulfil the behaviour. The three behaviours are defined as following:

- *Separation* - Separation provides a steering force such that the movement of the agent avoids colliding with other flocking agents. It does not take into account avoiding collisions with obstacles which is handled separately. This steering behaviour is calculated by summing the difference between the current agent in question and its neighbours. This sum is then scaled by the inverse of the separation distance $(1/r)$ to provide a smooth force which increases as the agents get closer to each other.

- *Alignment* - Alignment is the steering behaviour that causes entities to attempt to match their directional heading with those of its neighbours. It is calculated as the average of the heading velocity of the neighbours.

- *Cohesion* - Cohesion is the third and final steering behaviour involved in creating the flocking movement. This behaviour provides a steering force towards the average position of the neighbours. It is calculated as the sum of the position of the neighbours divided by the number of neighbours. It is this behaviour that causes flocking entities to attempt to herd together.

The final steering force that will be applied to the Reynolds agent is then calculated. Reynolds provides two different combination methods and suggests a third method in his later paper[15]. These methods combine the forces provided by the different behaviours in an attempt to best produce a result that is desirable to all of the behaviours.

Each of the steering behaviours uses information about the surrounding entities. These surrounding entities are known as an agent's neighbours and the perception of these neighbours with respect to the current entity is an important part of the framework. Reynolds argues that an agent within a flock does not have complete knowledge of all the entities within the flock. He then goes on to suggest that each agent only requires knowledge of the agents that it perceives around it. In his original paper Reynolds uses a simple sphere to define the perception field of an agent but in [15] he defines both a sphere and a field of view. This creates a perception field shaped as a sphere with a cone section removed from the back. This model of the agent's perception attempts to represent the natural field of vision of a bird.

Due to the fact that agents only require a local knowledge Reynolds suggests sorting the boids into an arrangement of bins. Each agent lies within a bin and it's bin is updated as it moves throughout the simulation. This sorting of the agents allows the computational complexity of the neighbour comparison to be reduced from $O(n^2)$ to $O(binsize)$.

Reynolds further notes that both obstacle avoidance and scripted behaviour can be represented using steering forces that can simply be added to the summation of the flocking behaviours. Reynolds solution for obstacle avoidance is to poll what objects lie directly in front of the agent and then generate a steering force that turns the agent towards the nearest edge of the obstacle. Reynolds mentions that these obstacles could in fact be other behavioural characters and therefore scenes such as a flock of birds moving around a larger animal such as an elephant are possible.

One of the major advantages to using this method is that the behaviour is completely deterministic. This is a desirable attribute as the rendering during movies or the testing of games would become increasingly difficult if the agent movement was different with each instance of the application being rendered or run.

Reynolds mentions the major disadvantage of this flocking technique. He states that it is difficult to create a combination of forces that produces the desired behaviour and that fine tuning the behaviour can be a repetitive and time consuming activity.

Further work by Reynolds presents further steering behaviours that can be used within his original framework[15]. By combining these behaviours it is possible to create groups of agents that act as a flock, but also easily integrate with other behaviours that could be used when the agents state is changed. Examples of these extra behaviours include: *Flee*, *Pursuit*, *Wander* and *Leader Following*.

The concept of *Leader Following* is also addressed in the paper by Lebar Bajec et al.[5]. The technique discussed involves the use of fuzzy logic in order to make a model that defines flocking behaviour in such a way that it is easier for ethologists to comprehend.

The concept of using the boids model to simulate crowds is revisited by Lebar Bajec et al.[6]. An in depth analysis of perception throughout this paper concludes that alignment has the highest influence on flock formation, but reduces overall speed and also proposes that the best solution to avoid collisions between agents is to adjust the weights of the three behaviours such that separation and alignment are at 90% while the cohesion steering behaviour is at 10%.

As the agents in Reynolds framework only require knowledge of their local neighbours the flocking technique is easily scalable on multi-core hardware architecture. In Reynolds most recent work he discusses the implementation of the original flocking behaviour on the Playstation®3 architecture, taking into account the demand for scalability across the multi-core architecture[16]. Work by Pettré shows that in the long term a group of Reynolds agents will tend to form into a single large flock[12]. Reynolds implementation on the Playstation®3 includes an additional steering behaviour to disperse crowds of increased density in order to prevent a single flock from forming and to reduce the loads inside a single bin.

Recent work by Lee et al. also builds on the use of local neighbourhood perception to determine crowd movement[7]. Lee et al. uses aerial camcorder footage to record crowd behaviour and then develop an appropriate model of behaviour.

Scutt suggests a method to create simple swarms as an alternate approach with the aim of providing a believable swarms with minimal computational complexity[18]. Scutt's approach succeeds in reducing the required computational time to simulate the swarm, but does not guarantee the separation of the entities in the swarm. Scutt's approach is aimed at elements within a simulation that are not critical to the outcome of the simulation but are instead added to 'breathe life' into the simulation.

Treuille proposes an alternative model for crowd movement that makes use of dynamic potential fields instead of individual agent perception[22]. This framework combines the search for a global path with the search for local avoidance into a single calculation. This approach produces a crowd movement that naturally forms lanes of agents moving behind each other. Work by Berg et al. also describes a technique that creates a similar lane forming style crowd[1]. The paper by Treuille notes that the proposed framework can be used in combination with other agent models such as the framework suggested by Reynolds. The author also notes that this technique is unsuitable for agents that require individual goals. This lack of specification for individual agents is addressed by Sud et al.[20]. Sud et al. propose a framework using Voronoi diagrams and creates a technique that allows for individual agent goals. The main drawback of this framework is the increased computational requirements, the example simulation given only running at 5 frames per second for 200 agents.

1.2.2 Environment Representation

The concepts mentioned so far deal primarily with the steering behaviour of agents, rather than the representation of the environment in which these agents move.

Alternate methods of crowd simulation that have been proposed discuss the use of Voronoi diagrams in order to divide up the space on which the crowd is to traverse. Work by Pettré involved creating Voronoi diagrams around the static obstacles in the environment and then mapping a connected graph of cylinders onto this space[12]. This technique allows for movement of a crowd along corridors such that the agents do not tend towards a single line. Kamphuis et al. presents a similar approach in which the corridors of movement are created by a leading entities initial path finding, thus restricting group members to group together to form crowd movement[3].

The more recent work by Pettré et al. continues to refine the use of navigation graphs and cylinder movement spaces[11]. This work includes the concept of *Levels of Simulation* in order to simulate a crowd of 35,000 pedestrians at 10-20 frames per second. This involves updating agents that are closer to the camera more often in order to reduce the computational load while maintaining the apparent detail of the crowd. Though unusable for simulations and games this concept is highly applicable to movie renders involving crowds where the position of agents off camera is irrelevant and those of agents in the distance are unneeded at a high fidelity. Maïm et al. also discusses the use of *Levels of Simulation* in order to produce a large scale crowd simulation that runs at a desirable real-time frame rate[8]. Richmond et al. suggests a level of detail implementation that utilizes a computer's GPU in order to achieve an increase number of agents within the crowd[17].

Work by Nieuwenhuisen et al. states that while group splitting and reforming is a desirable and natural behaviour for flocks of animals it may be an undesirable behaviour for games or simulations where the crowd being represented move together in a unit[10]. Nieuwenhuisen et al. use a similar approach as Pettré et al. and Kamphuis et al. by defining the environment as a combination of navigation graph and bounding cylinders. Nieuwenhuisen et al. use these bounding cylinders to provide a corridor of acceptable space in which the agents of a unified group may move.

Silver also proposes an alternative method for path finding that is designed to cause entities in the crowd to choose paths which cooperate with the other agents around them[19]. Silver achieves this by dividing the environment into a grid, with each entity reserving a place in the grid. The entities then reserve the grids in future time steps as they move throughout the simulation and when they re-plan their path, they take into account the other cells that have already been reserved by other entities. Silver notes that this approach could be modified to account for entities of different sizes, with larger entities reserving a larger number of cells in the grid.

1.3 Summary

In this chapter the field of crowd simulation has been examined. The techniques in this field produce crowd movement where either the crowd is treated as a whole, or the individuals in the crowd each have their own behaviour whose combined affect produces a crowd simulation. Several of the approaches discuss the representation of the environment in order to best move the agents of a crowd around the environment in a realistic fashion. This dissertation aims to examine two aspects which are often ignored in the majority of crowd simulation techniques. The first is the different crowd movement that is possible when a crowd contains characters of varying sizes and the second is the increase in possible movement when the characters are able to traverse beneath other characters that are significantly larger. In the next chapter the multi-layered flocking system is formally introduced in an attempt to provide a technique to overcome these issues in crowd simulation.

2 Mutli-Layered Flocking System Design

2.1 Introduction

In this chapter a formal definition for the algorithms and data structures that compose the multi-layered flocking system is given. As this system is based upon the flocking algorithm proposed by Reynolds, a description of the different components of Reynolds algorithm is given as an introduction to the multi-layered flocking system.

Reynolds flocking algorithm can be separated into four general components. The first of these components is composed of the properties of the individual agents in the simulation that represent the characters. These properties are used by the algorithm to determine how each of the agent's behaviours should act. The second component is the representation of the simulation space. The examined technique is referred to as *dynamic spatial partitioning*, in which a grid is used to divide up the simulation space upon which these agents are located. The third is the selection of neighbours for a given agent that will be used to influence the agent's behaviour. These neighbours are chosen by defining a neighbourhood in which to examine for potential neighbours. This component represents the agents' perception of the other characters within the simulation. The final component is comprised of the behaviours that operate on these agents in order to produce their navigation. Each of these behaviours defines a force that determines the direction the agent should move in order to best produce the desired movement of the behaviour.

The multi-layered system represents an extension of a two dimensional implementation of Reynolds algorithm by taking each of these components and extending them, either through modifications, additional constraints or a more generalized abstraction. Specifically the system aims to produce a methodology to define the interaction between characters of vastly different shape or size without the requirement of individually tailored navigation systems. It is designed to allow for the efficient interaction between characters of vastly different sizes and to allow for the definition of movement in which agents can move underneath other agents in a realistic fashion.

2.2 Navigation Agents

Reynolds model uses the term boid to describe the data that encompasses a characters position and movement. Each boid represents a single animal such as a bird or fish that is to be simulated. Navigation behaviours provide forces to act upon each boid and the resulting forces for this boid describe its movement. Each boid is comprised of a series of attributes in order to simulate the locomotion of the boid around the simulation space. The attributes that a boid contains are:

- *Position*: A vector representing the boids centre. It is around this point that the visual model of the boid is drawn.
- *Heading*: A vector representing the boids direction of movement. The magnitude of this vector represents the boids current speed. Note that in a 3-dimensional simulation another vector would be required to determine the boids orientation.
- *Mass*: A scalar value representing the mass of the boid. This value is used to calculate the boids acceleration.
- *Steering Force*: A vector representing the steering force that is to be applied to the boid.
- *Perception Radius*: A scalar value representing the radius of the circle that is used to detect neighbouring boids.
- *Navigation Behaviours*: A list of all the behaviours that are to act upon this boid.
- *Cell*: A reference to the cell that this boid currently resides in.

The simulation for each boid is performed using two update methods. The first is the physical update which calculates an acceleration vector for the boid given the boids steering force and mass. In a simple simulation this is typically done using Newton's second law of motion which is that the force is equal to the mass times the acceleration ($F = m * a$). Using the calculated acceleration value a new heading and position for the boid can be obtained. This update is equivalent to a standard world update in a typical rendered animation or game simulation and can be performed with either a fixed or varied time-step. The approximation of the simulation's locomotion to that of the real world is simulation specific. Any approximation can be used that employs the forces calculated by the agent's behaviours to determine the character's locomotion.

The second update is the behavioural update. During this update each boid recalculates its neighbours and determines the forces that should act upon it for each of its navigational behaviours. Note that these two updates do not have to be simultaneous. A lower frequency update can be used for the behavioural update to match the needs of the simulation. A level of detail model could also be used to improve the efficiency for simulations where the appearance of the behaviour is only required at a high fidelity at certain parts of the simulation, e.g. around the camera.

The multi-layered flocking system abstracts this concept to provide a system that is able to represent more complex character navigation. In the place of boids each character in the multi-layered system is represented by one or more navigation objects. Navigation objects are one of two types: navigation agents and navigation obstacles. The navigation agents act in a similar fashion to a Reynolds boid. Each navigation agent is located within a single navigation cell in the same way that each boid was located within a cell. Each navigation agent also contains the same attributes as a boid, with the addition of an extra scalar value representing the bounding radius. This bounding radius is used to determine the size of an object and will be used in the interaction between objects of different sizes.

A navigation obstacle will be used to represent objects in the simulation that are either stationary or whose position is dependent on another navigation agent. For example a rock that is to be avoided would be classified as a navigation obstacle. A walking human could be represented by one navigation agent to describe its overall movement and two navigation obstacles for each of the legs. The position of the navigation obstacles would be determined by the visual representation of the character, and as such does not require all of the attributes that a navigation agent is composed of. A navigation obstacle only requires a position, reference to a cell and the scalar value representing the bounding radius. Using these attributes a navigation obstacle is able to contribute to the navigation behaviours of other objects, such as collision avoidance, even though the navigation obstacle itself does not have any navigational behaviours.

2.3 Layered Navigation Cells

In Reynolds approach he suggests the use of a single grid that is placed over the simulation space. Each boid in the simulation is associated with a single cell within this grid. Reynolds names this proposed data structure *dynamic spatial partitioning*. The use of this data structure to specify the simulation space provides for an improved efficiency in the identification of neighbouring boids with which to calculate the navigation forces.

A boid is determined to be within exactly one cell at any one point in time during the simulation. A boid lies within a cell if its position lies within the boundaries of a cell. See Figure 2.1 for an example of a single boid positioned upon a grid representing the simulation space of the environment.

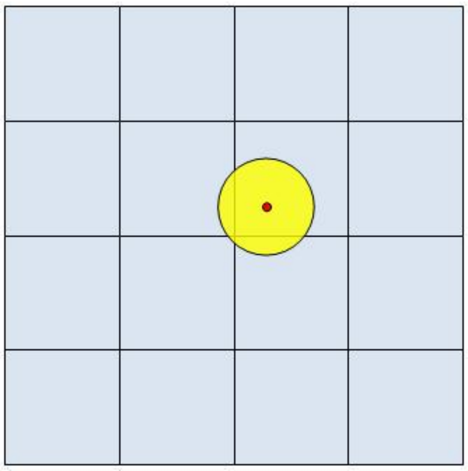
Figure 2.1: An example of a boid upon a grid of cells. The agent's position is shown as the red point and the perception radius of the agent is shown in yellow. Each blue square represents a single cell within the grid that covers the simulation space.

In the multi-layered system cells are also used to divide the simulation up, though unlike in Reynolds approach, the use of cells is compulsory and is used for more than just increased efficiency. Characters in the scene are represented in the simulation by one or more navigation objects. Each of these navigation objects lies within a particular navigation cell and each cell is associated with a single layer. A navigation layer is defined as a collection of cells that cover the entire simulation space. Navigation layers may contain a reference to a parent navigation layer. The navigation objects that lie within the cells of this parent layer, and any ancestor layer above that, will be used to influence the movement of the agents in the child layer. Furthermore the interactions between agents on different layers allows for new types of navigation interaction.

If a layer has a parent layer then all cells within that layer will have a parent cell within the parent layer. In order for a layer to be a valid parent for another layer, the cells within the parent layer must meet certain restrictions. The entire simulation space covered by the parent cell must be completely covered by its child cells. Additionally the area that each child cell covers must lie within only a single parent. In order to meet this restriction it is often necessary for a cell within a layer to be a slightly different size to the majority of cells in that layer. This is to ensure that the entire simulation space is covered without breaking one of the two restrictions above. If the simulation space is made to wrap around the x or y axis then the cell that is of a different size must be larger than the standard grid cell size. If this is not true then the agents within the simulation may not detect neighbours correctly due to having a perception radius that spans across more than one neighbouring cell. In Figure 2.2 an example is demonstrated in which a parent layer requires a slightly larger row of cells in order to guarantee that all the child cells are covered.

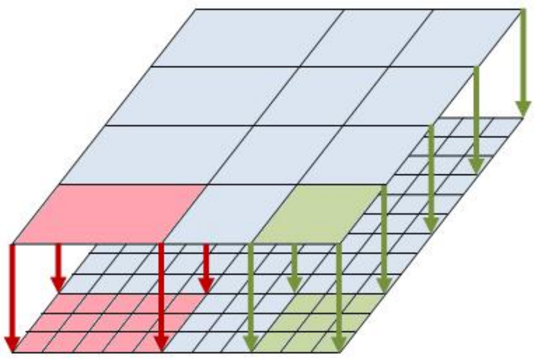

Figure 2.2: A demonstration of the mapping between the edges of parent and child cells upon two cell layers. The top layer has been created with three child cells per parent cell along both the x- and y-axis. The parent cells with green arrows map exactly to the layer below, with each parent cell lying above nine child cells. The parent cells with red arrows are required to be slightly larger in order to ensure that the six additional child cells on the far left of the x-axis are included within a parent cell. These red parent cells therefore have fifteen child cells.

The restrictions on parent size ensure that the layers are built up from the highest detail at the bottom to the lowest detail at the top. Thus the layers at the bottom will contain a larger number of smaller sized cells, while the layers at the top will contain a smaller number of larger sized cells.

The shape of the cells can be implementation specific, though a rectangular shape is most likely due to the efficient detection of a point within an axis aligned grid. Furthermore the relationship between cells means that the corners of cells in parent layers always share the corners of the cells in the child layers. As such a careful implementation can make use of this property to reduce the memory requirements to store the layers of cells.

2.4 Neighbourhood Selection

In Reynolds boids framework the neighbours of a boid determine the influencing behavioural forces. The neighbours of a boid are determined by finding the nearby boids within the simulation and then testing these boids against a perception function to determine if they will influence the boid in question.

Reynolds notes that a naive approach to determine the potential neighbours would involve a $O(N^2)$ comparison between each boid in the flock. The comparison though can be reduced using two possible techniques, one of which, *dynamic spatial partitioning*, is used as the basis for the multi-layered cell representation. In order to more efficiently obtain a list of the neighbours for a boid in Reynolds model, the simulation space is partitioned into a grid as explained in the previous section.

The neighbourhood selection algorithm returns all the boids within the selected boids cell and all those boids within the neighbouring cells. In a common 2D grid the neighbouring cells would be those cells that are above, below, right, left and diagonal to the current cell, giving a total of eight cells. Then each of the potential neighbours that lie within these cells is checked to see whether it lies within the perception radius of the current boid. In this implementation the boid considers a neighbour to be within its perception if the neighbouring boid's position is within the circle defined by the perception radius.

The approach must be modified for the multi-layered system in order to account for the neighbouring navigation objects that are of a different size. Navigation objects that are of a large size may be within the perception range of another navigation agent without the centre of the navigation object being within the perception radius i.e. only the edge of the large navigation object lies within the perception circle. Without taking this into account there could be missed interactions between navigation objects resulting in unwanted collisions or unusual navigation behaviour.

In order to avoid this problem the navigation agents in the simulation are given an additional scalar value, the bounding radius. The original radius, the perception radius, determines the distance at which an agent perceives its neighbours. The second, the bounding radius, defines a circle describing the bounds of the agent itself. The perception function for the agents is then changed such that an agent perceives a potential neighbour if any part of the bounding circle of the neighbour lies within the perception radius. Therefore a potential neighbour is considered to be within the perception distance if the distance from the centre of the agent to the neighbour agent is less than the sum of the agent's perception radius and the neighbours bounding radius.

This specified perception function is by no means the only possible method. With respect to the original boids, Reynolds suggests that the "field of sensitivity should realistically be exaggerated in the forward direction" in order to more accurately simulate a birds perception [14]. The defined perception function above is an example of a more simple representation. It is quite conceivable that more complex perception functions could be created that are tailored to specific character types, such as a particular animal's field of view, or also specific situations, such as weather effects like fog.

In the multi-layered flocking system the neighbourhood definition is extended to include some of those navigation objects that lie within the cell layers above the current agent's layer. Since the navigation objects in the layers above are within cells that are of equal or greater size there is potential that the navigation objects that are to be included have a larger bounding radius. For this reason the neighbourhood is modified to include the surrounding cells of each of the ancestors of the cell that the current agent lies within. Thus the neighbourhood of an agent is defined as the cell of this agent, plus all of its ancestors and the cells surrounding the current cell and each ancestor cell. A formal listing of the algorithm that finds all navigation objects within an agent's neighbourhood is given in listing Algorithm 1: Neighbourhood Selection.

```
Input: NavigationCell cell, NavigationAgent agent, boolean includeNeighbours,
       boolean includeParents
Output: List of NavigationObjects neighbours within cell, excluding agent,
        including neighbouring cells if includeNeighbours is true, including
        parent cells if includeParents is true.
neighbours ← ∅;
foreach NavigationObject navObj in cell do
    if navObj ≠ agent then
        if DistanceTo(agent,navObj) < (agent.perceptionRadius +
        navObj.boundingRadius) then
            Add navObj to neighbours;
        end
    end
end
if includeNeighbours then
    foreach NavigationCell nCell in neighbours do
        neighbourObjs ← NeighbourhoodSelection(nCell, agent, false, false);
        Add each NavigationObject in neighbourObjs to neighbours;
    end
end
if includeParents then
    parentObjs ← NeighbourhoodSelection(cell.parent, agent, true, true);
    Add each NavigationObject in parentObjs to neighbours;
end
```

Algorithm 1: Neighbourhood Selection

A visual example of this algorithm is shown in Figure 2.3. In it is a demonstration of a three layered navigation system and the resulting cells that are considered neighbours when the neighbourhood algorithm is run on a cell in the bottom layer.

All of the objects within these cells are compared with the perception function of the current agent to determine if they are valid neighbours, and then the navigation

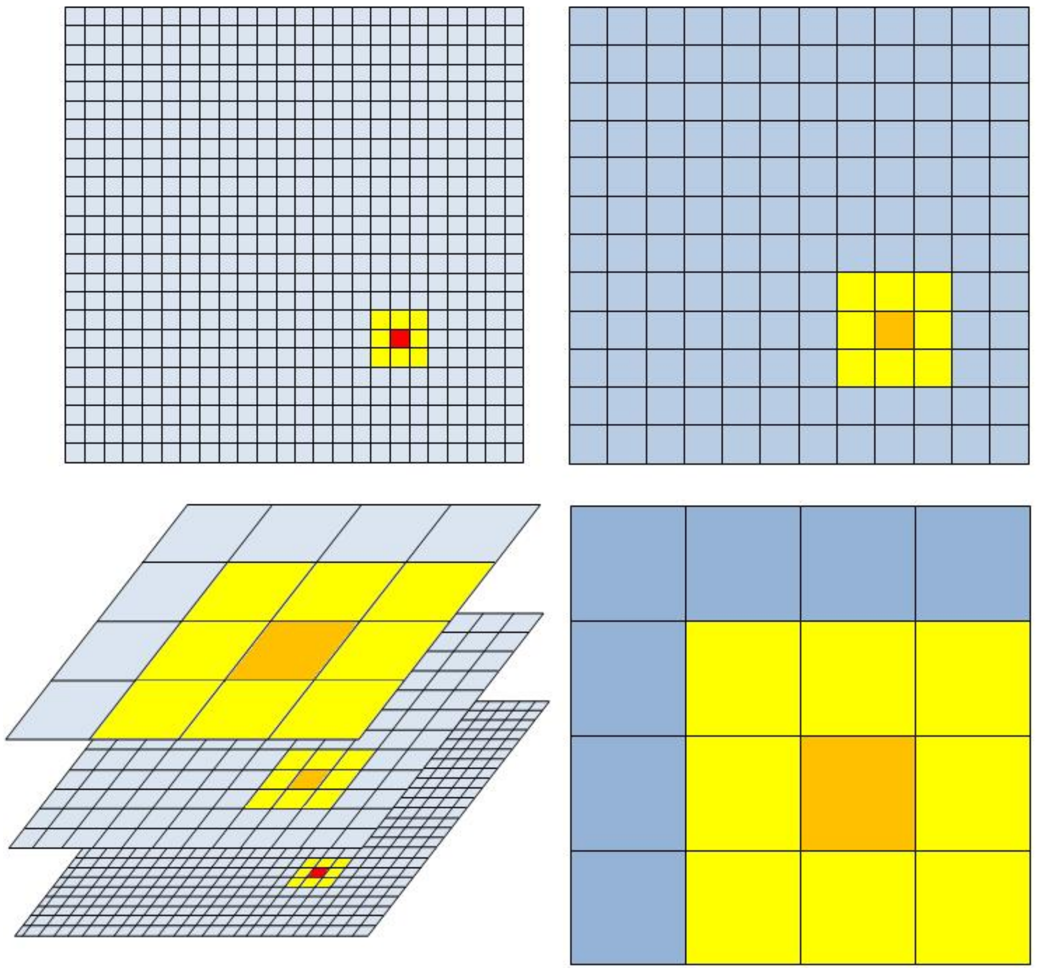

Figure 2.3: A visual example of the Neighbourhood Selection algorithm being run on a three layered navigation system. The image is divided into four sections. On the top left and top right are the first and second layers respectively. On the bottom right is the third cell layer. On the bottom left is an isometric view of all three layers. The original cell that the agent is positioned in is shown in red. Ancestor cells are shown in orange and the neighbouring cells are shown in yellow. The remaining cells in blue are not examined.

behaviours use these neighbours to produce their recommended steering forces. Figure 2.4 displays an example of a navigation agent's perception radius and the detection of neighbouring navigation objects on two cell layers.

2.5 Navigation Behaviours

Reynolds specifies a number of different navigational behaviours that allow for character movement [15]. Each steering behaviour uses the attributes of the boid and possibly the other boids that are its neighbours to decide in which direction it should go. The behaviour then provides a force that it determines will steer the boid in the right direction to achieve the desired movement. The forces of all the steering behaviours of a boid are taken and then a final steering force is calculated. The steering force of the boid is set

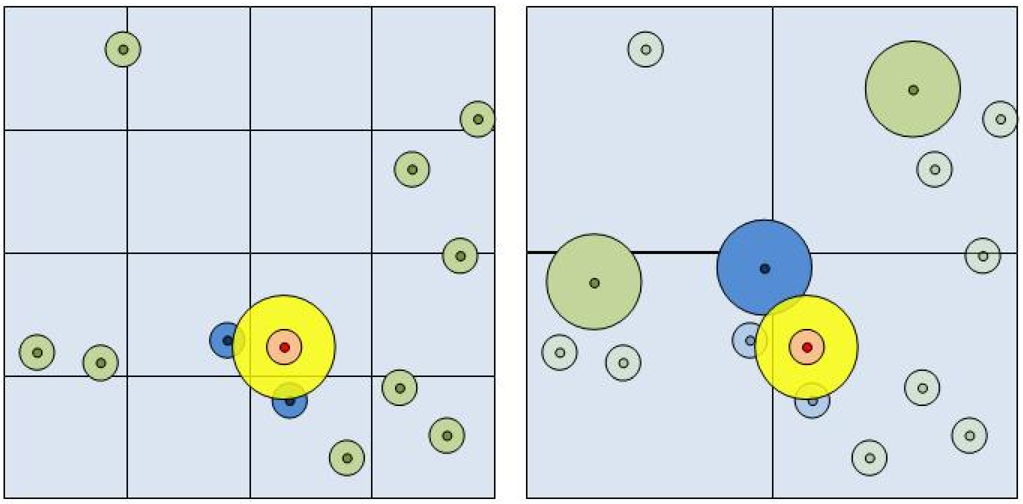

Figure 2.4: An example of an agent's perception in two layers. On the left is the agent's perception on the bottom cell layer, on the right is the perception on the top layer. The agent is represented by a position in red, bounding radius in orange and a perception radius in yellow. Other navigation objects in the simulation are shown in green if they are outside the agent's perception radius, blue if they are contained.

to this calculated value and then used during the physical update to determine a new position.

The final steering force that will be applied to the boid is calculated in one of several ways. The most simplistic form is the weighted average of each of the three forces calculated by the navigation behaviours defined above. A slight variation of this is to normalize the direction of each force returned by the steering behaviours and then multiply each force by a weighted value. This method provides weighting parameters that can be adjusted in order to modify the overall movement of a boid, allowing for a preference in the separation behaviour in order to avoid collisions. Though this model is simple and easy to compute, it leads to indecision when two of the steering behaviours produce opposite forces, resulting in a negated final steering direction. In order to address this issue the more complex method of prioritized acceleration allocation can be employed. This technique attempts to allocate the total amount of acceleration that the boid can provide. It does this by considering the steering behaviours in order of their priority. Reynolds implementation suggests giving the separation behaviour a higher priority in order to cause navigation agents to prefer avoiding collisions over forming into flocking groups. In [15] a third method is suggested for calculating the final steering force. Reynolds states that one behaviour can be evaluated randomly for a given probability for each update of the simulation, thereby spreading the computational load of calculations across multiple updates of the simulation.

For the multi-layered flocking system the same approach can be used to determine the final force. The different navigational behaviours can all be applied to the simulation, though modifications may need to be made to take into account the influence of the layers of navigation objects. For example the separation behaviour is normally calculated as the sum of the difference between the current agent in question and its neighbours. This sum is then scaled by the inverse of the separation distance ($1/r$). In a multi-layered system the difference between the current agent and the neighbours must take into account the bounding radius of the objects and can additionally be multiplied by a factor representing the difference in layer height. This additional weight can then make agents prefer to avoid collisions with agents in layers above, which in turn may create a more natural steering behaviour for the specific simulation. An example of this case is that of a small boat such as a yacht avoiding a large ship such as a tanker in preference to other smaller boats. This would occur because the tanker is represented by a navigation agent on a layer above the yacht while the other smaller boats would be represented on the same layer as the yacht.

2.6 Cell and Agent Property Relations

The closely tied relationship between cells and the navigation objects within them place certain restrictions and dependencies upon their attributes. Firstly the perception radius of the agents in a particular layer must always be less than both the width and the height of the cells in their layer. This ensures that agents on the border of a cell cannot perceive objects that lie further than one cell away (as these cells are not included in the neighbourhood selection algorithm).

The ratio between the bounding radius and the perception radius must also be considered. An agent requires that the difference between these two distances be sufficiently large such that two agents moving directly towards each other do not intersect after a single behavioural update. Since these attributes depend entirely on the simulation being created, e.g. the maximum speed of the agents in each layer, they are implementation specific. The most suitable approach would be to create the cells with a larger than expected size, allowing for a larger perception radius. This size can then be decreased in order to increase the performance until a suitable minimum sized cell value is found that allows for a perception radius that still avoids navigational errors like collisions.

2.7 Summary

In this chapter a formal description for the multi-layered flocking system was given. The multi-layered flocking system represents each character in a crowd using one or more navigation objects. These navigation objects are either navigation agents or navigation obstacles. Navigation agents examine their neighbourhood to determine those navigation objects that are close enough to influence the character they represent. The neighbours that are found are then used by one or more navigation behaviours to determine a force to apply to the character in order to create the characters locomotion. The system uses navigation cells to divide the simulation space into a series of layers. The neighbourhood of a navigation agent depends on the cells in which it is contained and those cells that surround it and lie above it. The next chapter examines the way that different characters can be represented using this system and then discusses the varying types of crowd movement that can be created.

3 System Capabilities and Characterization

3.1 Introduction

This chapter examines the different type of movement that are possible when using the multi-layered flocking system. It begins with a description of the different representations that the characters in the system can use and the ways these different representations affect the movement. It explains the use of properties in order to enhance the system and includes some examples of useful properties that produce desirable behaviours. Following this is a discussion of the representation of the environment and how this can affect the resulting navigation of agents within the simulation. This chapter then continues on to discuss the various behaviours that are applicable to the system, and the various advantages and disadvantages of the individual behaviours. It covers the use of specific behaviours in order to combine path-finding systems with the multi-layered system. It then compares the multi-layered system with the original Reynolds model before concluding with a discussion about the advantages and disadvantages of using the multi-layered flocking system.

3.2 Types of Movement

This chapter begins by explaining some of the types of movement that are possible using the multi-layered system. In order to do this a simple example involving two types of characters is presented that will be used to explain some of the different representations of characters that are possible in the multi-layered system. Consider a simple simulation that is composed of two types of characters. The first group consists of standard human sized characters; the second consists of giant humanoid characters that are significantly larger.

3.2.1 Single Layer Character Representation

The simplest way to represent these two groups of characters is to simply represent each individual character with a single navigation agent. All of these navigation agents then reside on a single cell layer that represents the traversable area in the simulation. See Figure 3.1 for a visual example of this representation.

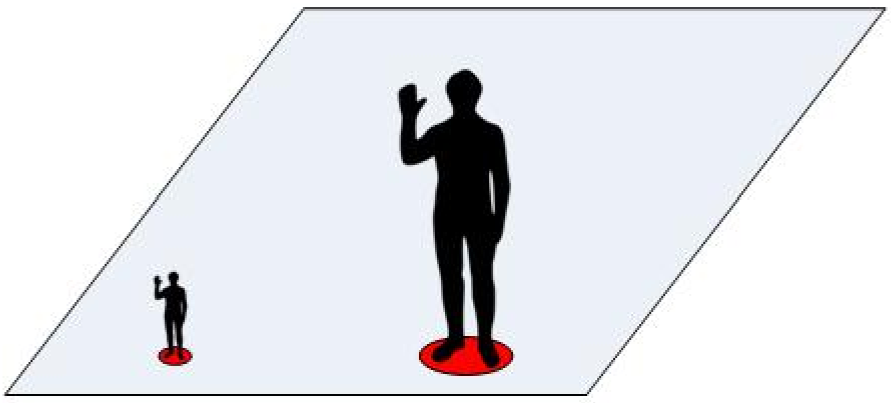

Figure 3.1: An example of single layer character representation with a human and giant. Both the human and giant characters are represented on a single cell layer, each using a single navigation agent to represent their position and bounding radius.

Using this representation the characters can be given appropriate behaviours and would traverse about the simulation without collision. The size of the cells in the cell layer would need to be quite large as it needs to be big enough to meet the conditions of the bounding radius of the giant characters. This will cause a somewhat inefficient calculation of the neighbourhood for the characters in the simulation. This is due to the fact that the required larger cell size could potentially hold a large number of the smaller human characters and as such will cause the number of objects in a neighbourhood to be large. This in turn will increase the number of comparisons required to determine the force caused by each behaviour. To avoid this problem the simulation can be extended to use another cell layer.

3.2.2 Multi-Layered Character Representation

The addition of cell layers in the multi-layered system allows for an improved efficiency while maintaining the simulation of the movement of the two groups of characters with vastly different sizes. An additional cell layer needs to be added to accommodate the giant characters. Both groups maintain their representation of a single navigation agent at their position, but the giant characters have their navigation agents placed upon the top cell layer. See Figure 3.2 for the visual example of this representation.

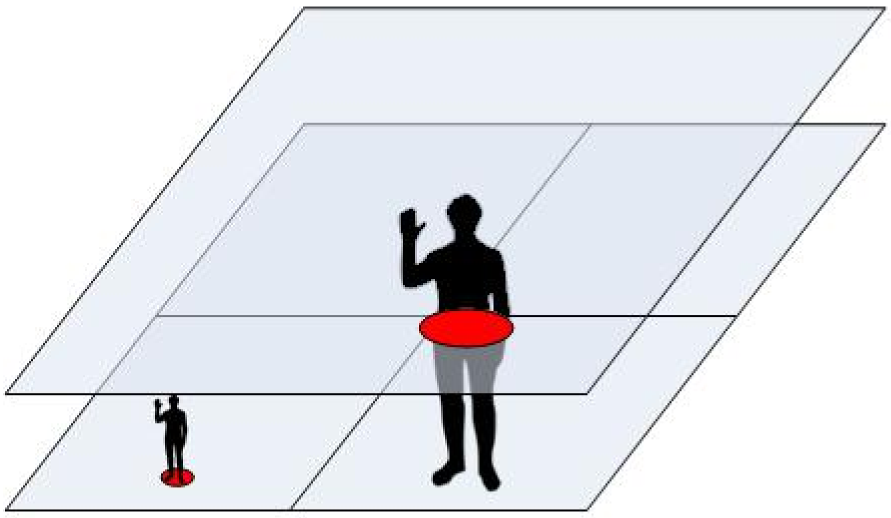

Figure 3.2: A human and giant character represented on two cell layers. The human character is represented by a navigation agent on the bottom cell layer; the giant is represented by a navigation agent on the top layer.

With this current definition the multi-layered system has divided the two groups of characters. Now the characters representing the giants lie upon the top layer and are unaware of the characters upon the bottom layer. The characters on the bottom layer will consider both the other human characters and the giant character to be within its neighbourhood and will avoid both. The cell size on the top layer will be the same as in the previous example, but the cell size on the bottom layer can be a much higher fidelity and will therefore reduce the number of comparisons to determine an agent's neighbours.

This representation has a notable flaw, which is that the characters that lie on the lower cell layer may be sufficiently small such that they can move beneath the larger characters. With the current representation such movement is not possible, as the characters on the lower level will treat the navigation agents that represent the giant characters with equal avoidance to the other neighbours. In order to address this issue the concept of specialized properties is introduced.

3.3 Properties

Properties are attributes that particular navigation objects in the simulation have in order to enhance their resulting movement. Properties are directly coupled with the behaviours associated with an agent and as such are specific to the simulation being created. A simple example of a property would be a species enumerator associated with each agent in a simulation. Using this property two behaviours can easily be created by modifying the cohesion and alignment behaviours to only conform if the neighbouring agents have the same value for the species property. This additional property is easy to add to the simulation and cleanly creates a navigation system in which particular species coordinate their movement together.

3.3.1 AffectsBelow Property

This use of properties is essential to further define complex behaviours, especially concerning the interaction between characters on different levels. As mentioned previously the multi-layered system is able to represent characters of vastly different sizes and it may be desirable to create movement where the smaller characters move underneath the body of the larger characters. In order to create this type of movement an additional property is used and the representation of larger characters is altered. This additional property is an *affectsBelow* boolean value. By default this value is true, but when it is false, the navigation object with this property is ignored by agents that lie on any cell layer below the current navigation object. This therefore allows for agents on a parent layer to interact with their neighbours on the same layer, but to not affect those agents that lie in any child layer.

Next the representation of the giant characters on the top layer is modified to use this new property. The character is now divided up into two parts, the legs and the main body. The main body is represented by a navigation agent that lies on the top layer. This main body has the *affectsBelow* property and the value of this property is set to false. The legs of the character are represented by navigation obstacles that lie on the child layer and their *affectsBelow* value is left to the default value of true. See Figure 3.3 for the visual example of this representation.

The movement of the giant characters will still be defined by the behaviours associated with the navigation agent that represents the characters body. As the character's body moves throughout the simulation the position of the navigation obstacles that represent the legs of the character are updated to match the animation of the character.

Now given this setup a new behaviour that modifies the existing separation behaviour is created so that it takes into account the additional *affectsBelow* property. The separation neighbourhood is defined as only those neighbouring navigation objects whose cell

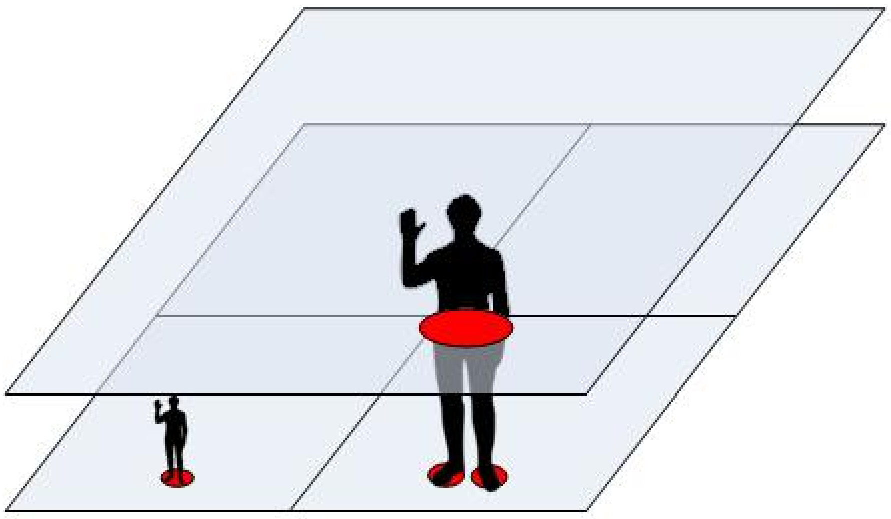

Figure 3.3: A human and giant character represented on two cell layers. The human character is represented by a navigation agent on the bottom cell layer; while the giant character is represented using multiple navigation objects on both the top and bottom cell layers.

layer is equivalent in depth or whose *affectsBelow* property is true. Using this representation of the characters the resulting movement of the human characters will be such that they will move past the agents on the parent layer by avoiding their legs but without taking a preference for moving beneath the parent layer.

This property and behaviour setup can be further extended to use a scalar value to represent the preference of a character to avoid the parent layer agents. If this scalar value is small, the parent agent will provide a weak force such that agents will choose to go around the parent agent, but could move between their legs if there is sufficient force to cause them to do so. If this scalar value is large, the resulting force would cause the agents on the child layer to avoid passing beneath the agents on the parent layer unless absolutely necessary.

The movement of the legs that lie on the lower layer needs to depend on the position of the body agent that lies on the upper layer. Furthermore it is important that the position of the leg aligns with the visual representation of the object in the simulation to ensure that the collision avoidance is consistent with the model being displayed. If the model being represented lifts its legs high enough such that the agents on the child layer could potentially pass underneath it then the navigation obstacle needs to be removed or deactivated while the leg is in the air. The obstacle can then be re-added to the child layer when the foot of the leg returns to the ground. It was found during a prototype implementation of this property that the sudden return of the navigation obstacle to the child layer caused child agents to clip through the feet of the parent agent. This can be considered a desired property if the simulation is intended to show child agents being crushed by the feet of the parent agents. If this is undesirable the navigation obstacle that represented the leg can be re-added a few seconds in advance of the foot being returned to the child layer. This causes agents to begin moving out of the way in advance of the foot being returned to the ground.

This concept of using two layers to represent a character to allow smaller characters to move underneath them is not limited to only two layers. The same approach can be extended indefinitely to allow for even more complex interactions. For example if the simulation wanted to have a third layer of characters through which both the previously defined layers moved underneath, then the third layer of characters would be defined by a body represented by a navigation agent on the third layer and navigation obstacles representing the legs on the second layer. Since the leg obstacles are within the neighbourhood of both the first and second layer, all of these characters will avoid the obstacles representing the legs and move underneath the agent.

3.4 Environment Representation

The representation of the static environment in the simulation should also be adjusted to match the agents that are in the simulation. As can be seen by the characters in the simulation there are a number of ways that objects can be represented and several different choices are examined below.

Consider again the simulation that included the two groups, one representing human sized characters, the other representing giant sized characters, and assuming they are represented upon two cell layers with the giant sized characters being split into body and legs over the two layers.

Obstacles in the environment that affect the agents on both layers should be represented by a navigation obstacle that lies on the top cell layer. Such a navigation obstacle will be included in the neighbourhood of the navigation agents in both the bottom and top cell layer and as such both these groups of agents will avoid it.

If on the other hand a piece of the environment only affected the children and was crushed by the feet of the characters whose navigation agents lie on the top layer, then the object can be represented by a navigation obstacle on the lower cell layer. The obstacle in the environment can then be destroyed when a parent agent's foot obstacle collided with it. A further type of environment, where the child agents could move through it, but the parent agents would be forced to go around can also be represented. Such an obstacle in the environment could be a building, through which the smaller human characters can enter and move about, but which the larger giant characters must navigate around. To create this type of movement the object in the environment could be represented by a series of navigation obstacles on the lower layer, and a single larger navigation obstacle on the top layer that has the *affectsBelow* property set to false. This will cause the larger agents on the top layer to navigate around the structure, while smaller agents would be able to move about the environment at the lower level. See Figure 3.4 for a visual example of this representation involving a tree.

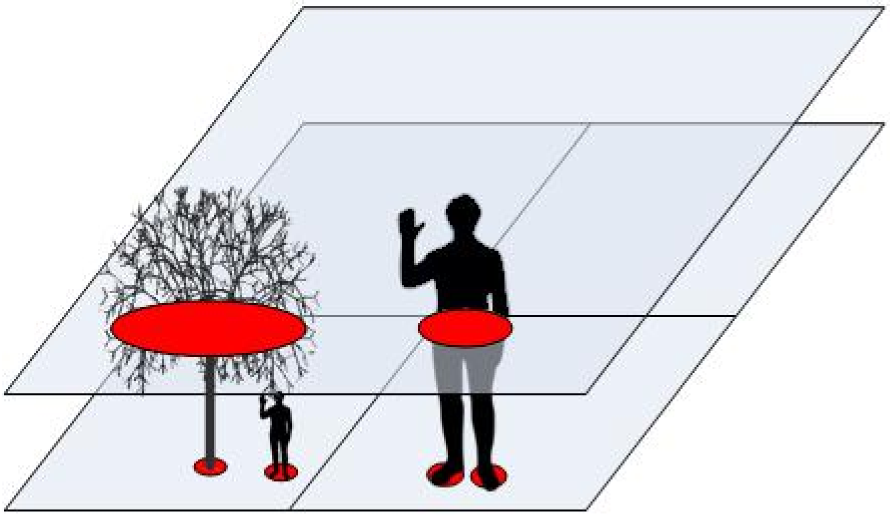

Figure 3.4: An example representation of a tree as part of the environment. The tree is represented on two layers which ensures that the smaller human sized characters are able to navigate close to the trunk while the larger giant sized characters are restricted to navigating around the branches.

3.4.1 Example Properties

Two additional properties that are generally useful in a simulation are *isActive* and *is-Blocking*.

The first, *isBlocking* is a boolean value that determines whether a navigation object should be included in the multi-layered flocking system with respect to other character's neighbourhood. By disabling this value we are able to stop this object having any influencing behaviour on other characters. This property can be used on any object in the simulation that does not wish to influence the separation behaviour of other agents. One example of using this property is to change the property to false for buildings that have been destroyed in the simulation and as such are no longer an obstacle. Another example is the representation of ethereal objects such as ghosts that could potentially be moved through by other characters.

The *isActive* property serves a related purpose. It is used to enable or disable whether a character should be influenced by the behaviours it uses. Often in a simulation a character needs to be removed from the simulation, a common example being if the particular character has died during a game. In this case the character should not be playing a part in the influence of other characters movements and updating the characters behaviours is equally unimportant. The character though may not be able to be completely removed from the simulation. The character from the previous example could be playing a death animation sequence where the character remains in the simulation. By setting the *isActive* property to false we can easily continue this animation while avoiding unnatural behaviour of the character after its death. It may also make sense in this particular example to also set the *isBlocking* property to false, as the dead character may no longer provide an obstacle to the other characters in the simulation.

3.5 Cell Layer Hierarchy

Up until this point the simulations presented have followed the format of smaller agents lying on a plane with larger and larger agents being placed in successively higher layers. Though this representation of layers in these particular examples corresponds to an increasing height value, this is not the only ordering that the layers are able to represent. Furthermore it should be noted that the examples so far have also only contained cell layers where each parent layer has a single child layer. By allowing more than one cell layer to use the same cell layer as its parent, simulations are possible in which there are smaller agents both below the larger agents and above them. Consider an extension to the above example that contained two groups of characters: humans and giants. Now assume that the simulation also required a flock of birds to be represented that would fly at the head height of the giant characters. The characters that represent these birds would be required to fly around the giant characters. To achieve this, the new bird characters would lie upon a third cell layer. This third layer would treat the cell layer that the giant characters lie upon as their parent layer. The simulation now contains two cell layers, the human layer and the bird layer, both of which have the giant layer as its parent. This representation ensures that the bird characters only need to consider each other and the giant characters when navigating and waste no time checking for collisions with the human characters in the simulation. Note that the child cells of both layers must match up to the boundaries of the parent layer to conform to the conditions described in the algorithms chapter.

3.6 Behaviours

So far this chapter has covered the different ways characters and the environment can be represented in the multi-layered system. This next section discusses the different behaviours that could typically be used in the multi-layered system, first discussing common movement behaviours and then examining the development of more complex behaviours that allow for the integration of the multi-layered system with path-finding systems.

3.6.1 Standard Movement Behaviours

The multi-layered system uses the same force combination methods as the Reynolds model. This allows the system to use any movement behaviour that can be defined as a force upon the navigation agent at a point in time in the simulation.

Using the standard weighted combination or force priority techniques mentioned in the algorithm chapter, steering behaviours can be developed that deal with the majority of standard movements required in animations and games. Reynolds provides an overview of these typical behaviours[15]. The steering behaviours that Reynolds covers include:

- *Seek*: Move towards a position.
- *Arrive*: Move towards and stop at a position.
- *Pursuit*: Follow another object.
- *Separation*: Move away from other nearby objects.
- *Cohesion*: Move towards the average position of nearby objects.
- *Alignment*: Face in the same direction as nearby objects.
- *Wander*: Move in a random direction.
- *Path Following*: Seek towards a series of points.

These behaviours can be easily adapted or extended to create movement that is more suitable to the simulation being developed. For example the *Path Following* behaviour can be modified to follow a lane on a road network, with the additional attributes that it will change lanes and stop at lights when needed.

3.6.2 Multi-Layered Movement Behaviours

By utilizing the common movement behaviours that are applicable to the multi-layered system, simulations can be made that involve a range of different movement types. Combining several of the above common behaviours allows a designer to produce varied types of movement within a simulation. The *Separation*, *Cohesion* and *Alignment* behaviours can be used to create a standard crowd. By modifying the weighting or priority of these behaviours a crowd can be made to appear slow and dense or fast and erratic. In 3.5 an example is displayed that shows a crowd of characters moving through the legs of a larger four-legged character. This simulation was created with the multi-layered flocking system by combining the standard behaviours of a Reynolds model with a *Wander* behaviour for the four-legged character. The same simulation could be modified to make the crowd flee the four-legged character, swarm beneath it, or use the four-legged character as protection from another larger character by simply adding slightly modified versions of the *Separation*, *Seek* or *Pursuit* behaviours.

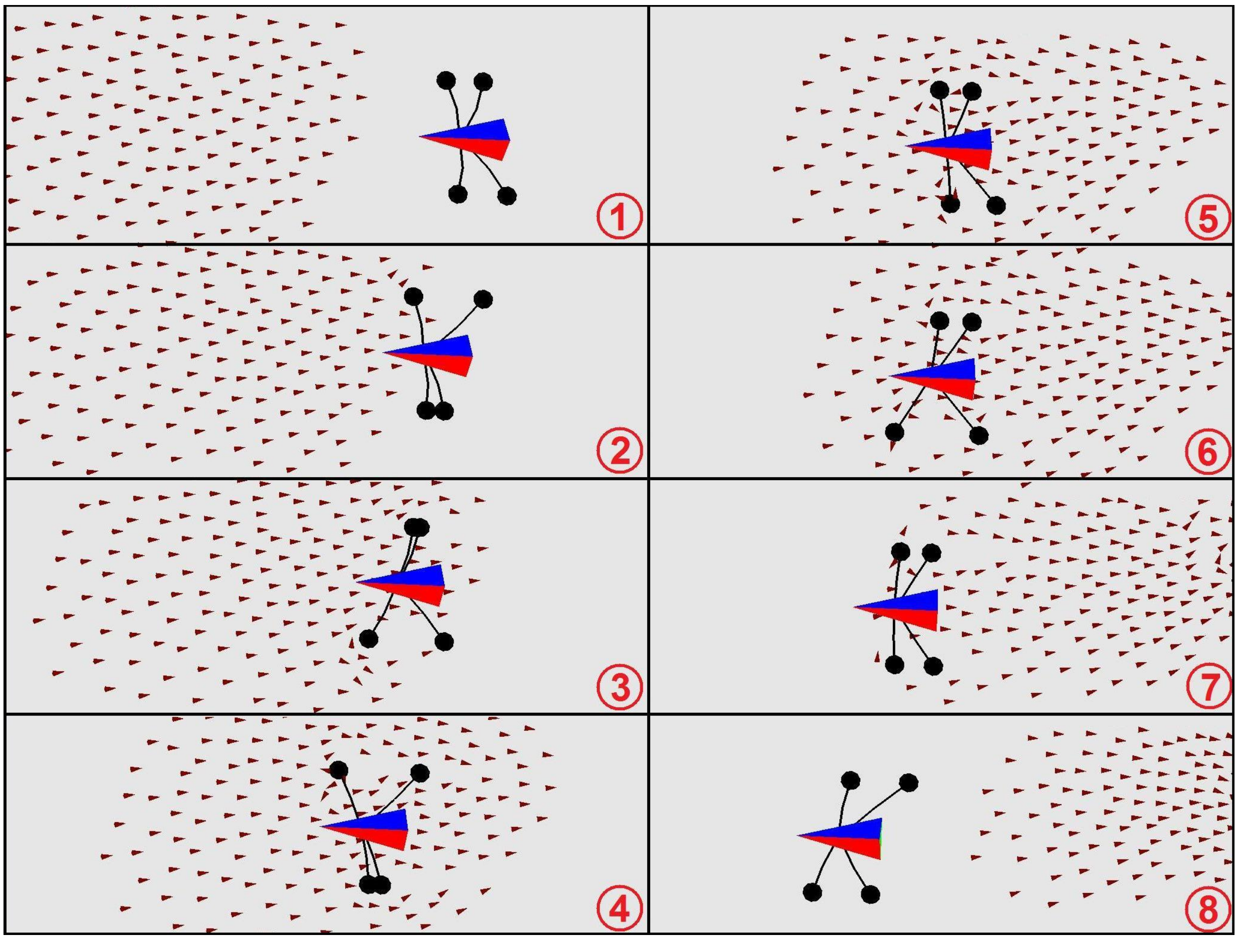

Figure 3.5: Example screenshots of a four-legged character moving through a crowd of smaller characters. The image is broken into eight sections, each representing an increase in time step in the simulation. The simulation is set up with two cell layers; the four-legged character is represented by a navigation agent on the top layer and four navigation obstacles on the bottom layer. The smaller characters are represented by a single navigation agent on the bottom layer.

An additional behaviour can be made that acts as the opposite of the *Separation* behaviour. Instead this behaviour can be used to cause characters to be drawn towards another object in the simulation. Such a behaviour can be combined with the *affectsBelow* property to create movement where child agents are attracted towards a parent agent, giving the impression that the child is hiding beneath the parent for safety. This can be applied to both navigation agents and navigation objects, thereby creating solid objects that an agent is hiding beneath as well as moving objects such as larger creatures. This behaviour is not limited to hiding beneath other agents and could also be used to attract characters towards an area, whether static or moving. For example the attraction area could represent the shade of an obstacle and objects would move towards the shade, updating their position with the movement of the sun.

3.6.3 Integration with Pathfinding Systems

Often in simulations, particularly in games in which the player controls many characters and performs tasks with them, the simulation requires more advanced pathfinding techniques that allow the characters in the simulation to navigate the environment. The multi-layered system restricts a character's knowledge to that of its local neighbourhood and as such would perform poorly in a simulation that requires complex pathfinding. Despite this, complex path-finding is easily added to the simulation by providing behaviours that act as a bridge between more complex pathfinding models and the multi-layered system. For example, consider the A* search algorithm typically used in games to find a path from one point in the simulation to another. This search can be used with the characters represented in the multi-layered system by creating a new behaviour that extends the *Path Following* behaviour to perform an A* search. This behaviour can then follow the computed path by returning a force towards the next waypoint in the path.

3.7 Comparison with Reynolds Navigation Model

Reynolds navigation model produces a resulting navigation that efficiently represents the movement of a flock in either two or three dimensions. The two dimensional version of Reynolds model can be easily recreated using the multi-layered flocking system and is essentially a subset of the possible resulting movements possible using the multi-layered system. In order to represent the Reynolds model a single cell layer would be used upon which the agents would lie.

All the agents in the layer would be given the same three defining behaviours of the navigation model: *Cohesion*, *Separation* and *Alignment*. With this setup the neighbourhood selection algorithm will return the same neighbours for each agent as they would in the Reynolds model. Computationally the process would be identical with the exception of a single additional check to determine if there is a parent layer, which would be absent. The ability of the multi-layered system to represent Reynolds model is a beneficial property as it allows scenes that do not require a complicated representation of movement to be created using the same system as more complex movement, thus providing a unified approach to all aspects of navigation in the simulation.

3.8 Advantages of the Multi-Layered System

The multi-layered system restricts the knowledge that characters in the simulation have about other characters in the simulation. By only providing the minimum amount of required knowledge to each character the system is able to reduce the computational time that each character requires to calculate its movement. This multi-layered system also maintains the advantage of partitioning the space that represents the traversable space in the simulation. Such a division allows the system to be more adaptable to the multi-core architectures that are almost standard in gaming consoles and rendering processes. The calculation of the characters behaviour is also highly parallelized due to the fact that each characters movement is only dependent on its own behaviours and its local neighbourhood.

Once the core architecture of the multi-layered system has been created the process of developing characters is relatively straightforward. Each character need only be represented by one or more navigation objects and one or more behaviours. The positioning and calculation of navigation obstacles for individual body parts such as legs could be automated by using the bounding volumes of the visual animations. Such automation would allow the designers of the simulation to rapidly prototype different types of characters, while quickly choosing the types of navigation that they wished the characters to follow. Furthermore the flexibility of the multi-layered system reduces the restrictions upon the designers, allowing them to create more interesting movement within their simulation.

3.9 Disadvantages of the Multi-Layered System

The multi-layered system allows for more complex navigation simulations to be developed but there are some disadvantages to using this approach. The problems associated with the method proposed by Reynolds are inherited into the multi-layered system. The main disadvantage is the fine tuning of varying forces to find a resulting movement that is most applicable to the simulation. If a navigation agent contains a large number of behaviours then the competition between forces may cause a resulting movement that appears contradictory to what the designer is trying to achieve. It may also not be easy to understand which particular behaviours are causing the undesirable motion. Reynolds recommendation of a prioritized combination method to choose which behaviours to enforce requires that the resulting forces of all the behaviours is scaled appropriately and does not overwhelm the possible maximum force that the navigation agent can produce in a single update.

Another issue that can potentially develop in the simulation relates to a specific property: the *isBlocking* property defined previously. Navigation agents that have this property set to false will not naturally separate from other agents on the cell layer. This therefore can lead to a large grouping of these characters in one cell on the cell layer. Since the efficiency of the system depends on the number of characters in one cell being low, a large grouping of characters can lead to a large degradation in performance. Reynolds notes this issue in his paper discussing and implementation of flocking on the Playstation®3 and recommends the use of an additional specific behaviour that ensures that large groupings of characters do not occur by providing force away from cells that are too full[16]. This method attempts to prevent overcrowding while allowing for crowds to form up to a maximum extent.

The entire structure of the multi-layered system uses the assumption that in a situation with a smaller object and a larger object, the smaller object is expected to navigate around the larger object. This structure is what maintains the efficiency of the model in the simulation. Though this structure is appropriate for the situations suggested above, it is not true of every situation.

Consider for example an adult moving about the house with the knowledge that a two year old child is playing in the living room. The parent would be represented on a cell layer that is above the cell layer that the child is upon. When the parent moves through the living room they will be careful to note the child's position and avoid stepping on the child. Consider also a situation with a simulation of fish and sharks where the representation of fish and sharks is also represented in a similar fashion. The sharks, being of significantly larger size would be placed on a cell layer that is above their prey.

With the multi-layered system the information about navigation objects flows from the top down. Navigation objects on higher levels affect those on the same or lower levels. In both the situations specified above the navigation agents on the top layer require information about the navigation agents on a lower layer in order to navigate correctly, thus the flow of information is from the bottom up. These two situations are not naturally handled by the multi-layered system. In order to facilitate this, the system can be modified to allow behaviours to have access to the cells in a layer below the layer that the agent resides in. Doing so would avoid this problem but potentially allow for large inefficiencies. This is because the agents could potentially search through a large number of navigation objects upon the lower cell layer. Behaviours that require access to a child layer require careful implementation to ensure that the complexity of these behaviours is kept to a minimum.

3.10 Summary

In this chapter the possible representations of characters and the environment have been explained. This chapter highlights the different representations, describing the resulting movement that will occur when each type is implemented. It also discusses the use of properties to further enhance the multi-layered system and define more complex behaviours. The different behaviours that can be used within the system have been explained, including the use of behaviours specifically designed to include more complex path-finding systems. Finally the chapter has discussed the advantages of the system and some of the possible disadvantages that could occur and suggests some ways to go about avoiding or reducing these problems. In the next chapter the conclusion for this dissertation is given and some future directions for further research are defined.

4 Conclusion

4.1 Conclusion

This aims of this dissertation were to define a system that allows designers to create crowds that are able to contain members of varying sizes and that allows for characters in those crowds to travel underneath larger characters. In this respect the multi-layered flocking system is successful. The resulting movement produced by the system is able to represent characters moving in groups of varying size, and these groups are able to move through, around and underneath other characters in the simulation.

The system divides the simulation space into a series of navigation cells in a similar approach to Reynolds flocking model. Several layers of these cells can be created, and using these layers the simulation space can be abstracted to allow for different sized characters to be placed within the simulation. In this dissertation the characters and environment in a simulation are represented using one or more navigation objects. The choice of navigation objects depends on the desired representation that the designer is attempting to produce. The examination of the capabilities of the system has shown that the representation is flexible enough to allow for varying types of characters and obstacles to be created, including objects that allow for smaller characters to pass beneath them.

The behaviours that are applied to each of the characters provide the characters movement. The characters in the simulation represented by this system are unique and each has its own behaviours. Thus it is possible to use this system to create both group behaviour such as crowds and combine this with specific individual goals and desires. Furthermore the system can use specific navigation behaviours to incorporate more complex navigation techniques such as pathfinding.

4.2 Future Directions

The multi-layered flocking system in this dissertation is conceptually an abstraction from the 2-dimensional version of Reynolds flocking method. It is theoretically possible to create a similar multi-layered flocking system that uses the 3-dimensional version of Reynolds flocking method as a basis. Such a model would use a 3-dimensional cell representation, which would then be layered. This structure would provide a spatial partitioning that determines the members of its neighbourhood as those within distance of an abstract volume rather than the objects that lie upon an abstract plane as is done in this dissertation. Such an implementation may prove to be complex to understand for a designer, but warrants investigation.

The prototype application built to test the multi-layered flocking system put forward in this dissertation uses somewhat standard behaviours. Though additional behaviours have been examined an in-depth study of the varying types of behaviours would provide for a useful overview to present to designers who wish to use the system. Furthermore the examination of integration of alternate techniques could also be performed in order to determine the ease at which these alternate methods can be integrated within the system.

Bibliography

[1] J. v. d. Berg, S. Patil, J. Sewall, D. Manocha, and M. Lin. Interactive navigation of multiple agents in crowded environments. In *Proc. of ACM Symposium on Interactive 3D Graphics and Games*, 2008.

[2] Gas Powered Games. *Supreme Commander*, 2007. `http://www.supremecommander.com`.

[3] A. Kamphuis and M. H. Overmars. Finding paths for coherent groups using clearance. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28. Eurographics Association Aire-la-Ville, Switzerland, Switzerland, 2004.

[4] P. Kruszewski. Real-time crowd simulation using ai.implant. In S. Rabin, editor, *Game AI Programming Wisdom 3*. Charles River, 2006.

[5] I. Lebar Bajec, N. Zimic, and M. Mraz. Simulating flocks on the wing: the fuzzy approach. *Journal of Theoretical Biology*, 233(2):199–220, 2005.

[6] I. Lebar Bajec, N. Zimic, and M. Mraz. The computational beauty of flocking: boids revisited. *Mathematical and Computer Modelling of Dynamical Systems*, 13(4):331–347, 2007.

[7] K. H. Lee, M. G. Choi, Q. Hong, and J. Lee. Group behavior from video: a data-driven approach to crowd simulation. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, page 118. Eurographics Association, 2007.

[8] J. Maïm, B. Yersin, and D. Thalmann. Real-time crowds: architecture, variety, and motion planning. In *SIGGRAPH Asia '08: ACM SIGGRAPH ASIA 2008 courses*, pages 1–16, New York, NY, USA, 2008. ACM.

[9] Massive Software. *MASSIVE*, 2009. `http://www.massivesoftware.com`.

[10] D. Nieuwenhuisen, A. Kamphuis, and M. H. Overmars. High quality navigation in computer games. *Science of Computer Programming*, 67(1):91–104, 2007.

[11] J. Pettré, H. Grillon, and D. Thalmann. Crowds of moving objects: Navigation planning and simulation. In *ACM SIGGRAPH 2008 classes*, page 54. ACM, 2008.

[12] J. Pettré, J. Laumond, and D. Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. In *First International Workshop on Crowd Simulation*, 2005.

[13] Pregasis. *AI.Implant*, 2009. `http://www.presagis.com/products/simulation/aiimplant/`.

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

[15] C. W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference. http://www. red3d. com/cwr/steer/gdc99*, 1999.

[16] C. W. Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, page 121. ACM, 2006.

[17] P. Richmond and D. Romano. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *International Workshop on Super Visualisation*, 2008.

[18] T. Scutt. Simple swarms as an alterative to flocking. In S. Rabin, editor, *Game AI Programming Wisdom*, pages 123–456. Charles River, 2002.

[19] D. Silver. Cooperative pathfinding. In S. Rabin, editor, *Game AI Programming Wisdom 3*. Charles River, 2006.

[20] A. Sud, E. Andersen, S. Curtis, M. Lin, and D. Manocha. Real-time path planning for virtual agents in dynamic environments. In *ACM SIGGRAPH 2008 classes*, page 55. ACM, 2008.

[21] The Creative Assembly. *Total War Series*, 2009. `http://www.totalwar.com`.

[22] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. In *ACM SIGGRAPH 2006 Papers*, page 1168. ACM, 2006.

[23] Ubisoft Montreal. *Assassin's Creed*, 2007. `http://www.assassinscreed.com`.

Appendix

.1 Appendix A: Pseudocode

This section includes peudocode for the fundamental classes required to implement a multi-layered flocking system. The pseudocode is written with an Object Oriented programming language in mind. The pseduocode focuses on the key aspects of the system, as described in chapter 2 and as such excludes common accessor and mutator methods. It also excludes methods which would be implementation specific such as those involving the data structure used for the NavigationCells. The classes covered are: NavigationObject, NavigationAgent, NavigationObstacle, NavigationCell and NavigationCellLayer.

.1.1 Navigation Object Pseudocode

Pseudocode describing the implementation of the NavigationObject super class. This class is extended by both NavigationAgent and NavigationObstacle.

```
class NavigationObject
{
  //Enumerator for the different types of NavigationObjects
  enum NavigationType
  {
    Obstacle,
    Agent,
  };

  //Returns the NavigationType of this NavigationObject
  virtual NavigationType GetNavigationType() = 0;

  /*...

  Common accessor/mutator methods for the variables
  used by all NavigationObjects. These are:
  - cellLayer
  - cell
  - position
  - boundingRadius

  ...*/
};
```

.1.2 Navigation Agent Pseudocode

Pseudocode describing the implementation of the NavigationAgent class. This class is a subclass of NavigationObject.

```
class NavigationAgent: public NavigationObject
{
  //The layer that this agent is located on
  NavigationCellLayer* cellLayer;

  //The cell that this obstacle is in
  NavigationCell* cell;

  Vector2 position;     //The position of this agent
  Vector2 velocity;     //The velocity of this agent
  Vector2 acceleration; //The acceleration of this agent
  Vector2 heading;      //The heading of this agent

  //The heading this agent will attempt to face each
  //update of the simulation
  Vector2 desiredHeading;

  double perceptionRadius; //The perception radius of this agent
  double boundingRadius;   //The bounding radius of this agent
  double mass;             //The mass of this agent

  //List of the behaviours which affect this agent's navigation
  vector<NavigationBehaviour*> behaviours;

  //Instance of the combinator that will combine each of the
  //agent's behaviours into a final steering force.
  BehaviourCombinator* behaviourCombinator;

  //Constructor
  NavigationAgent(/*... Agent Parameters ...*/)
  {
    /* ... Assign parameters to instance variables ... */
    //Update the reference to the cell that this agent is in
    this->cellLayer->UpdateCellReference(this);
  }

  //Update the behaviours. This will use the behaviour combinator
  //to calculate a desired heading from all the forces returned
  //by the behaviours in the 'behaviours' vector.
  void UpdateBehaviours(float currentTime, float dt)
  {
    desiredHeading = behaviourCombinator->
         GetCombinedHeading(behaviours,this,currentTime,dt);
  }

  // Returns the NavigationType of this agent
  virtual NavigationType GetNavigationType() {
    return NavigationType.Agent;
  }

  /*...

  Implementation of common accessor/mutator methods.

  ...*/
};
```

.1.3 Navigation Obstacle Pseudocode

Pseudocode describing the implementation of the NavigationObstacle class. This class is a subclass of NavigationObject.

```
class NavigationObstacle: public NavigationObject
{
  //The layer that this agent is located on
  NavigationCellLayer* cellLayer;

  //The cell that this obstacle is in
  NavigationCell* cell;

  Vector2 position;       //The position of this obstacle
  double boundingRadius; //The bounding radius of this obstacle

  //Returns the NavigationType of this agent
  virtual NavigationType GetNavigationType()
  {
    return NavigationType.Obstacle;
  }

  //Constructor
  NavigationObstacle(/*... Obstacle Parameters ...*/)
  {
    /* ... Assign parameters to instance variables ... */

    //Update the reference to the cell that this agent is in
    this->cellLayer->UpdateCellReference(this);
  }

  /*...

  Implementation of common accessor/mutator methods.

  ...*/
};
```

.1.4 Navigation Cell Pseudocode

Pseudocode describing the implementation of the NavigationCell class. This class contains a list of all the NavigationObjects whose position lies within the cell. The GetNeighbours method relates to the main neighbourhood selection algorithm described in the chapter 2.

```
class NavigationCell
{
  //The parent cell which lies above this cell
  NavigationCell* parentCell;

  //The layer this cell is on
  NavigationCellLayer* cellLayer;

  //The neighbouring cells to this cell
  vector<NavigationCell*> neighbours;

  //The bounds of this cell
  Vector2* cellBounds[4];

  //List of all the NavigationObjects in this cell
  vector<NavigationObject*> navigationObjects;

```

```
//Constructor
NavigationCell(NavigationCellLayer* cellLayer, Vector2* cellBounds)
{
  this->cellLayer = cellLayer;
  this->parentCell = cellLayer->CalculateParent(this);
  this->cellBounds = cellBounds;
}

//Returns the neighbouring NavigationObjects in a cell.
//This method excludes the NavigationObject 'agent'.
//If includeNeighbours is true then the neighbouring cells are
//also checked for NavigationObjects.
//If includeParents is true then the parent cell is also
//checked for NavigationObjects.
vector<NavigationObject*> GetNeighbours(NavigationAgent* agent,
                                        bool includeNeighbours,
                                        bool includeParents)
{
  vector<NavigationObstacle*> neighbours;

  //Get objects in this cell, excluding 'agent' if it is specified
  foreach(NavigationObject navObj in navigationObjects)
  {
    if(navObj != agent)
    {
      if(DistanceTo(agent,navObj)
              < (agent.perceptionRadius + navObj.boundingRadius)){
        neighbours.add( navObj );
      }
    }
  }

  //Get objects in neighbouring cells
  if(includeNeighbours)
  {
    foreach(NavigationCell nCell in neighbours)
      neighbours.add( nCell.GetNeighbours(agent,false,false) );
    }
  }

  //Get objects in layers above
  if(includeParents)
  {
    neighbours.add( parentCell.getNeighbours(agent,true,true) );
  }
```

```
    //Return all the NavigationObjects found
    return neighbours;
  }

  /*...

  Implementation of common accessor/mutator methods.

  ...*/
};
```

.1.5 Navigation Cell Layer Pseudocode

Pseudocode describing the implementation of the NavigationCellLayer class. This class contains a list of all the NavigationCells which make up this NavigationCellLayer. This class provides methods to determine parent-child relationships between cells and updates NavigationObjects to ensure that they referenced in the correct NavigationCell.

```
class NavigationCellLayer
{
  //The parent navigation layer, NULL if there is no parent
  NavigationCellLayer* parentLayer;

  //The cells in this navigation layer, data structure is
  //implementation specific, i.e. a grid of cells could use
  //a 2-dimensional array.
  vector<NavigationCell*> cells

  //Constructor
  public NavigationCellLayer(vector<NavigationCell*> cells)
  {
    this->cells = cells;
  }

  //Checks the navigationObject's position and updates the references
  //linking the obstacle to the cell that it is contained within.
  void UpdateCellReference(NavigationObject* navObj)
  {
    //Get a pointer to the new cell given the navigationObject's current
    //position.
    NavigationCell* newCell = GetCellAtPosition(navObj->GetPosition());
    NavigationCell* oldCell = navObj->GetCell();

    if(oldCell != NULL)
    {
      oldCell->RemoveNavigationObject(navigationObject);
    }
    navObj->SetCell(newCell);
    newCell->AddNavigationObject(navObj);
  }

  //Returns the cell that contains the position defined by 'objectPos'
  NavigationCell* GetCellAtPosition(Vector2 objectPos)
  {
    /*...

    Implementation Specific according to data structure used for
    'cells'.

    ...*/
  }

  //Returns the NavigationCell in this layer which would be the parent
  //of the NavigationCell 'childCell'
  NavigationCell* CalculateParent(NavigationCell* childCell)
  {
    /*...

    Implementation Specific according to data structure used for
    'cells'.

    ...*/
  }

  /*...

  Implementation of common accessor/mutator methods.

  ...*/
};
```