A Generic Extensible Steering Library Built on Influencers

Adam Thacker

Bachelor of Science in Computer Science with Honours/The University of Bath
April 2008
This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
A Generic Extensible Steering Library Built on Influencers

Submitted by: Adam Thacker

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:
Abstract

Steering, and in particular flocking, is an area that has been studied for many years. In a computer simulation different steering behaviours control the direction and/or speed an entity should go. These steering behaviours can be combined to create emergent behaviour and have a wide range of applications from animal modelling, crowd simulation, film animation and game AI to military UAVs (Unmanned Aerial Vehicles). This project aims to create a generic and extensible steering library that is not domain specific so that it could easily be used for any one of the applications previously noted. In order to help allow easy creation of steering behaviours we shall provide building blocks in the form of ‘InfluenceZones’, an extension of ‘Force Fields’. We will then build some steering behaviours ourselves to test whether these ‘InfluenceZones’ help or hinder their development. We find that they do indeed provide an excellent construction tool for creating steering behaviours but they alone are not sufficient to completely replace specific behaviours.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 System Architecture</td>
<td>14</td>
</tr>
<tr>
<td>4.3 “Flocker” Architecture</td>
<td>14</td>
</tr>
<tr>
<td>4.4 InfluenceZone</td>
<td>16</td>
</tr>
<tr>
<td>4.5 Behaviours</td>
<td>17</td>
</tr>
<tr>
<td>4.6 Simulations</td>
<td>19</td>
</tr>
<tr>
<td>4.7 Summary</td>
<td>19</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>20</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>20</td>
</tr>
<tr>
<td>5.2 Tools Used</td>
<td>20</td>
</tr>
<tr>
<td>5.3 BoidBrain</td>
<td>21</td>
</tr>
<tr>
<td>5.4 InfluenceZone</td>
<td>23</td>
</tr>
<tr>
<td>5.4.1 Zone</td>
<td>23</td>
</tr>
<tr>
<td>5.4.2 Influencer</td>
<td>24</td>
</tr>
<tr>
<td>5.5 Behaviour</td>
<td>26</td>
</tr>
<tr>
<td>5.6 Helpers</td>
<td>36</td>
</tr>
<tr>
<td>5.7 Summary</td>
<td>38</td>
</tr>
<tr>
<td>6 Evaluation and Testing</td>
<td>39</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>39</td>
</tr>
<tr>
<td>6.2 Simulation 1 - Flocking</td>
<td>39</td>
</tr>
<tr>
<td>6.3 Simulation 2 - Point Following</td>
<td>41</td>
</tr>
<tr>
<td>6.4 Simulation 3 - Flocking, Point Following and Follow the Leader</td>
<td>44</td>
</tr>
<tr>
<td>6.5 Simulation 4 - Flocking and Collision Avoidance</td>
<td>45</td>
</tr>
<tr>
<td>6.6 Ease of Use</td>
<td>47</td>
</tr>
<tr>
<td>6.7 Influence Zones</td>
<td>48</td>
</tr>
<tr>
<td>6.8 Summary</td>
<td>49</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>50</td>
</tr>
<tr>
<td>7.2 Review of Achievements</td>
<td>50</td>
</tr>
<tr>
<td>7.3 Critique of Process</td>
<td>52</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Basics of Steering ...................................................... 4

6.1 Simulation 1 - Start .................................................. 40
6.2 Simulation 1 - Start of Flocking .................................. 41
6.3 Simulation 1 - One Minute in ..................................... 42
6.4 Simulation 2 - Start ................................................... 42
6.5 Simulation 2 - Zones being drawn ............................... 43
6.6 Simulation 3 - Start ................................................... 44
6.7 Simulation 3 - Low Separation Weight ......................... 45
6.8 Simulation 4 - Start ................................................... 46
6.9 Simulation 4 - Start of Flocking ................................. 46
6.10 Simulation 4 - Flocking Around the Obstacles ............... 47
Acknowledgements

I would like to thank my project supervisor, Dr Claire Willis, for her help and feedback during this project and my father, Mr John Thacker, for proof reading this document.
Chapter 1

Introduction

1.1 Overview

In any graphical computer simulation, there is a need to be able to move entities around the simulation ‘world’. A common way of doing this is to use ‘steering behaviours’. A specific behaviour will tell its entity what velocity to go at, based on the logic of the behaviour and certain variables, and the entity can then act on this by moving itself (using some sort of animation system). More than one steering behaviour can be combined to create ‘emergent’ behaviour - complex behaviour that is created from multiple simple behaviours combined - which can help create much more natural simulations although they are harder to control and test.

Past steering behaviour research has often just looked individual steering behaviours and how they can be implemented (Reynolds, 1999). We intend to take a look at steering behaviours as a whole and see whether there is a strong common thread between them that can be used to help speed up and ease development. In effect what we are asking is whether we can create a generic steering behaviour system. Of course the answer to this is immediately ‘yes’ because we could create a system so generic it could be used for anything but this system would be completely empty and totally useless. What this means is that as well as trying to create a generic system, we also need to create a system that actually helps with the creation of steering behaviours.

Using a single concept to create steering behaviours is something that has been done before. In the 1980’s a film called ‘Eurhythmy’ was created (Amkraut and Girard, 1990) that used nothing but ‘force fields’ to control a flock of birds flying around avoiding each other and obstacles. This work shows that it is possible to create a steering system built on a single concept.

Because of the wide use of steering behaviours - from animal modelling, crowd simulation, film animation and game AI to military UAVs - creating a generic, easy to use steering system could be of wide benefit. Having a system that is built on a single concept would
CHAPTER 1. INTRODUCTION

mean greater code reuse which would likely mean faster development of individual steering
behaviours and easier testing as each ‘building block’ can be testing once in isolation.
This would in turn leave more time for the more time consuming, and important, part of
steering behaviours - testing and tweaking of in use steering behaviours to create emergent
behaviour that forms your specific simulation.

1.2 Project Framework

This document is broken into the following sections:

Literature Survey
In this chapter we shall look at the steering behaviours in depth. What are ‘steering
behaviours’? What past work has been done on them? What are they used for? What
software solutions exist for them? The aim of this is to understand the steering behaviour
domain and how our work will further it.

Project Objectives
With our research into past work done, in this following chapter we will list our objectives
and aims for the project. These will be used to form the basis of our design, and thus
implementation, and will be used to evaluate our final results.

Design Analysis
This chapter looks at the high level design of our steering library. We shall take a look
at the overall architecture of our system and discuss what needs to be done in order to
meet the aims of our project. We shall also design the different steering behaviours we will
implement using our library.

Implementation
In this chapter we will look at our steering library at a lower level than the design section.
This will include looking at how the key parts of the library were implemented and why they
were implemented that way. During this chapter we will also discuss how the ‘Influencer’
concept has helped or hindered the development of some steering behaviours.

Evaluation and Testing
In this chapter will use four different simulations that use our library to test and evaluate
some of our steering behaviours, and our library as a whole. Included in this will be an
evaluating of the ease-of-use of our library and evaluation of the ‘Influencer’ concept in
general (as opposed to the specific usefulness of it that we discuss in our Implementation
section).

Conclusion
The final chapter will summarise our project in general, including a review of the project’s
achievements with respect to the project objectives, the process of the project’s develop-
ment, a look at possible improvements and future work and analysis of what we learnt
whilst undertaking this project.
Chapter 2

Literature Survey

2.1 Introduction

In this chapter we will look at past research into steering behaviours and the surrounding fields. We will first look at what ‘steering’ means and what exactly it is used for. We will then look at some important steering behaviours before finally talking about some existing steering libraries.

2.2 Steering

Steering, at its most basic level, is the ‘desire’ for an entity to head in a certain direction, possibly at a certain speed. This entity could be an animal, such as a bird or fish, or a vehicle such as a car. The entity would thus be defined by certain properties such as a minimum and maximum speed, maximum turning angle and so on, depending on the application. The entity would then be controlled in some way, usually by higher level AI or A-life behaviours, by using different steering behaviours. It is important to note that steering is a ‘desire’ and not a kind of ‘force’ that pushes or pulls the entity - the entity is in full control of its own movement. Figure 2.1 demonstrates the basics of how an entity is ‘steered’.

Steering behaviours are actually an offshoot of flocking (Reynolds, 1987) however flocking is itself a steering behaviour. Reynolds published the original paper on steering behaviours (Reynolds, 1999) and as well as discussing the topic in general, goes into some examples of different steering behaviours, including Seek, Flee, Path Following, Collision Avoidance and Flocking. On their own these behaviours are not very interesting but by combining them together we can create ‘emergant behaviour’. For instance we can combine flocking, collision avoidance and flee together too create a simulation of birds that flock together, avoid collision with trees and flee when a bird of prey gets close to them.
CHAPTER 2. LITERATURE SURVEY

2.3 What is Steering used for?

Steering behaviours are incredibly useful for a whole range of applications. This includes both pure research work and the control of real world machines. In this section we shall take a high level look at some of these different applications. This is vital because if we want to create a generic and extensible steering library we want it to be able to apply to as many different areas as possible.

2.3.1 Model Animals

A very common use for steering behaviours is in animal modelling/simulation (Kunz and Hemelrijk, 2003), (Stephens, Pham and Wardhani, 2003). Different steering behaviours can be assigned to different animals depending on the purpose of the simulation. For instance, the ‘flocking’ steering behaviour can obviously be used to model flocking animals such as birds or fish. Other behaviours can be created to make different simulations, such as a ‘pursue’ behaviour for a bird of prey to hunt other birds and a ‘flee’ behaviour for the birds being hunted when the bird of prey gets close.

2.3.2 Animation and Films

The naive way to animate a creature would be to manually move it in each frame. Steering behaviours allow the animator to program the creature with a behaviour, such as following a path or another character, which can save a huge amount of time (Anderson, McDaniel and Chenney, 2003). In particular creating flocks of creatures used to be a long process but using ‘flocking’ behaviour techniques flocks can be created much faster, and be more realistic, as the animator only needs to script a path for the flock as a whole. One of
the first films to do this was ‘Eurythmy’ (Amkraut and Girard, 1990) which was a film purely showing flocking techniques. A more famous example is the wildebeest stampede in Disney’s ‘The Lion King’ which was also done using flocking behaviour techniques. Flocking techniques can be used for more than just creating wildlife though. They can also be used for other groups of entities moving together, such as large crowds. This has been used successfully to create large organic armies in films such as Lord of the Rings which used ‘Massive’ software (Massive Software, 2008).

2.3.3 Games

Like films, games have also made a lot of use of steering techniques (Bourg and Seemann, 2004). It can be used to control the movement of simple entities, such as following a path, but again, like film, it is the flocking steering behaviour that is perhaps the most useful. It has been used for the obvious reason - to create wildlife - but also, as in films, to control other groups of entities (Chen and Klie, 2002). For instance it has been used to control groups of soldiers that have to move in formation across a virtual world, often filled with obstacles. The big difference between games and films is that games are running in real time and thus the animation cannot be carefully and fully controlled meaning that the algorithms must be more robust. The fact that it is running in real time means that the steering algorithms must be very fast and it is often common to only use the more expensive behaviours, like flocking, for key parts of the game when needed and use similar techniques, such as swarming, for less important parts of the game (Scutt, 2002).

2.3.4 Crowds

Over recent years there has been a large amount of work looking at modelling crowds (Musse, Babski, Capin and Thalmann, 1998), (Treuille, Cooper and Popovic, 2006), (Reynolds, 2006). The simulation of crowds branched off from flocking and while it shares many similar ideas as flocking one of the main differences has been the interaction of crowds which is not something greatly looked at in flocking research. The uses of crowd simulation have ranged from simulating emergency evacuations from buildings on fire to its use in creating crowds for use in films and games (as noted above).

2.3.5 Military

Unmanned Aerial Vehicles (UAVs) have seen widespread use in modern militaries. As these drones are unmanned they are either controlled remotely or by autonomous control logic. In order to get groups of UAVs to fly together flocking techniques are often used along with other similar techniques such as swarming, and these papers often draw directly on Reynolds 1987 work. This work is often first simulated of course, such as the work done by Corner (Corner and Lamont, 2004), but is the most practical use of flocking seen.
CHAPTER 2. LITERATURE SURVEY

2.4 Important Steering Behaviours

We have seen that steering behaviours have a wide range of use in different applications. At the steering level they are all very similar and a path following behaviour in one area is likely to do exactly the same thing as in another area. As such we shall now look at the some key individual steering behaviours to get an idea of how they work.

2.4.1 Goal Seeking

Goal seeking is perhaps the simplest of steering behaviours. It is the behaviour to move towards a certain point and nothing else and thus can be used in a variety of scenarios. For instance, if you are modelling a flock in order to study birds, you will typically want them to move about from food source to nesting locations and so on. This sort of control tends to be indirect and is governed by the AI driving the creatures. If using flocking for animation on the other hand you will want direct control over the flock, such as telling the flock to follow a fixed path at a certain time, in order to create the animations that you want.

In Reynolds 1987 paper (Reynolds, 1987) he touches briefly on goal seeking and scripted flocks, but he goes into little detail. Before flocking techniques, it was necessary for animations to script every single entity in a flock and this was a time consuming process. Flocking allowed them to script the flock as a whole, meaning they could create just one path to follow and the flock would move, naturally as a group, along it. This was typically done by scripting one of the entities and the rest of flock would obey their normal flocking behaviour rules and would thus, in theory, end up following the scripted entity (usually called the ‘leader’). As you can see this new ‘following the leader’ behaviour is actually just a goal seeking behaviour with the goal attached to a moving entity, probably with an offset. However the problem with this was that there was no guarantee that the flock would follow the leader and this often meant that extra rules needed to be written to ensure the flock followed the leader. As part of their work on creating flocking animations, Anderson et al looked at this problem and, as part of the general solution, succeeded in creating flocks that followed certain paths while still following their basic flocking rules (Anderson et al., 2003). Anderson’s work differed somewhat from Reynolds’ in that they were only interested in creating specific animations while Reynolds was more interested in natural and emergent behaviour.

2.4.2 Flocking

It was Emlen first who talked about animals flocking together due to some sort of impulse to move towards each other (Emlen Jr, 1952) and then in 1987 Reynolds created a model of flocking by having the different entities feel a need to move towards or away from other flock mates (Reynolds, 1987). In his paper Reynolds coined the term ‘boids’ for the flock individuals and decided upon three basic behaviours for each of the boids to follow in order
to create a flock:

2. Flock Centreing: attempt to stay close to nearby flock mates.
3. Velocity Matching: attempt to match velocity with nearby flock mates.

However applying these three behaviours naively is unlikely to create a good simulation. They will usually need to be weighted and this could either be controlled explicitly or could depend on the circumstances. For instance, if the boid is at risk of imminent collision with another boid it should put all of its ‘effort’ behind collision avoidance and completely forget about the others until it has avoided the other boid. It is Reynolds’ paper that has formed the basis of all flocking work that has come after it.

One question is what constitutes a ‘nearby’ flock mate? In Reynolds’ original work, and in most work since, flock mates are considered ‘nearby’ if they are within a certain distance from the boid. This is quite a simple approach but produced good results and was usually extended so that boids could only see nearby flock mates within a certain arc (i.e. they couldn’t see, and would take account of, flock mates behind them). However more recent research has shown that birds, or at least starlings, instead use a ‘topological’ approach, as opposed to the Reynolds ‘metric’ one, to find nearby flock mates (Ballerini, Cabibbo, Candelier, Cavagna, Cisbani, Giardina, Lecomte, Orlandi, Parisi, Procaccini, Viale and Zdravkovic, 2007). What this means is that they look at a fixed number of nearby flock mates, regardless of where they are, instead of all flock mates that are within a certain range. Ballerini et al found that the starlings typically kept track of six to seven flock mates and that this topological approach allowed birds to form quickly back into flocks after separation (such as from going around an obstacle).

An important part to flocking is collision avoidance. Reynolds discussed this in his 1987 paper, and he wrote another paper the following year that looked just at collision avoidance (Reynolds, 1988). Reynolds discusses various different obstacle avoidance techniques all of which can be applied to flocking. Firstly he discusses techniques based on geometrical models all of which are based on moving away from the obstacle when you near it. These include the ‘force field’ approach, pushed away when you get within a certain distance of the object, the ‘steer away from centre’ approach, turn away from the centre of the obstacle when near, the ‘steer along surface’ which is where the entity detects an obstacle directly in front of them and turns away and finally ‘steer towards silhouette edge’ which involves using vision to find the edges of the obstacle. These different approaches tend to be useful in different situations, but it is the force field approach that has seen the widest use. We shall look at this in greater detail in the section below.

The second group of techniques he discusses are vision based, and although these are perhaps the most realistic techniques when creating flocking simulations they are of course much more complex and would not be fast enough for real time simulations with lots of entities. As this is not a vision project, these techniques are outside the scope of the
project. Finally he mentions techniques based on artificial intelligence, but these are more appropriate to robots and image processing systems and are also beyond the scope of this project.

The next major work in flock modelling was done in the film ‘Eurhythmy’ which depicted a flock of birds flying around a building. This work done by Amkraut and Girard used a force field approach for the entire flock simulation (Amkraut and Girard, 1990). In this simulation every bird and static object was given a force field that would repel birds away from it in order to avoid collisions.

One area which few researchers have looked at is the simulation of how flocking can come about. In 1993 Werner (Werner and Dyer, 1993) looked at how herding could evolve in a group of animals in a virtual world populated with food and predators. This is very different from most simulations where flocking behaviours are hard-coded into the animals at the start of the simulation. In this simulation Werner was able to create animals that naturally herd together as it provides bonuses to finding food, finding mates and avoiding predators. Although many papers have looked at animal evolution, we know of no others that have looked at how herding/flocking itself could evolve.

Reynolds continued his flocking work, and in 2000 published a paper about interaction with flocks (Reynolds, 2000). Specifically he modelled a group of pigeons feeding in a park and are ‘scared’ away when a toy car moves towards them or makes a noise. In any kind of simulation it is of great importance how the creatures react to their environment and other creatures and in fact this is often the reason for doing simulations. Although the objective of this project is not to directly create A-life, it is about creating a system that could be used to create A-life so this idea of creatures reacting to other creatures is important and must be considered.

As well as flock simulation, flock modelling has been used a great deal in pure animation such as that needed for films and adverts. In 2003 Anderson et al wrote a paper that dealt with just this (Anderson et al., 2003). In it they discussed a way to create finely controlled animations that were guaranteed to work as opposed to flock simulations whose flock ‘animations’ evolve naturally and with no control. As their basis they used Reynolds’ work so each bird was governed by the three basic rules described above, along with a new rule that told the bird to match a certain speed. Although the birds followed these rules, they were constrained to follow other rules over and above these rules, such as moving to an exact point at an exact time. These allowed them to have fine control over the flock and allowed them to create animations such as a flock following a line or the flock forming into an arrow shape. However it was a tricky balance to ensure they got the animation they wanted but at the same time the flock obeyed the other rules so that it looked natural. Although this work is important for pure practical animation it has little use for ‘natural’ flocking simulations.

Flocking techniques are of course not just used for bird simulations, they can be used for any ‘grouping’ of creatures such as herds of sheep and schools of fish. In 2003 Kunz et al looked at creating a detailed and realistic school of fish (Kunz and Hemelrijk, 2003). With
regards to flocking the important part was the different ‘bands’ they created around each fish. What this meant was that if a fish ‘saw’ another fish on its outer band it would move towards it, if it saw a fish in the middle band it would align with it and if it was in its inner band it would move away from it. This created the ‘three behaviours’ as created by Reynolds in 1987 and thus the fish flocked together. They were then able to look at effects of school size, body size and body form on schools of fish, but because modelling fish was the focus of the project they did not try to advance flocking forward at all, merely used it. Another very similar paper was written in the same year which also looked at how schools of fish react to a predator (Stephens et al., 2003).

This work has been extremely diverse. Some have based their work directly on what Reynolds did, such as having much finer control of the flocks in order to create animations (Anderson et al., 2003) or creating extremely detailed flocks of animals in order to study other aspects of their flocking behaviour (Kunz and Hemelrijk, 2003) or as part of a larger simulation (Werner and Dyer, 1993). There has been a lot of work on crowds (Musse et al., 1998), (Treuille et al., 2006), (Reynolds, 2006) and in fact it appears that most ‘flocking’ research currently being done is on crowds. Indeed there has been relatively little work done on directly advancing Reynolds’ work, instead most recent research has been using what Reynolds did or moving it into a different research area, such as crowds.

2.4.3 Collision Avoidance

Most interesting virtual environments are typically filled with various objects, some static some not. Entities that exist in these environments need to be able to avoid collisions with these obstacles as obviously in the real world creatures try not to run into obstacles such as trees.

One obstacle avoidance technique is the ‘force field’ approach (Reynolds, 1987), (Xiao and Hubbold, 1998), (Reynolds, 1988). This technique creates some sort of zone around the obstacle that when entered by an entity, pushes the entity away from it. There are problems with this, such as an entity moving directly towards the obstacle would only cause the entity to slow down rather than turn away and the zone having an effect even when the entity was just passing by it and not towards it. In order to get around these problems the force fields are normally more complex, such as only causing the entity to turn away when it is actually heading for it.

As we mentioned in the above section, Reynolds discussed force fields in his 1987 paper and about their use in collision avoidance (Reynolds, 1987). However more interesting is the 1990 film ‘Eurhythmy’ which used force fields to create the entire simulation and not just obstacle avoidance (Amkraut and Girard, 1990). This shows that although force fields are considered to be relatively simple and perhaps don’t create the most realistic simulations, they are in fact able to form the basis for an entire simulation.

A detailed paper on using force fields was written in 1998 by Xiao et al (Xiao and Hubbold, 1998). They looked at using force fields to ‘guide’ navigation around an environment
and around obstacles. They found that it allowed much easier movement by humans around a cluttered environment and at the same time made the simulation seem more real even though they were guided by invisible forces. Although they didn’t look at computer controlled entities the same basic principles apply and it is encouraging to see that they were able to create force fields around all objects, even very complex ones such as stairs.

A second approach is called ‘steer to avoid’ (Reynolds, 1987), (Reynolds, 1988) and this more closely resembles actual creatures. The entities effectively ‘look’ for obstacles and when they see that they are heading towards one and will collide with it in the near future they try to steer around it. In Reynolds’ papers he talks about finding the nearest silhouette of the obstacle and then steering towards it, however the most important part is detecting that a collision will occur shortly and then dealing with it. You could use the silhouette technique, but you could instead use some other technique, such as a version of the force field approach.

2.5 Existing Software

As steering behaviours are used so widely, there exist some software tools and libraries to help create steering behaviours, two of which we will look at here. These solutions are interesting to look at as they would be a good comparison to the software we will create.

**OpenSteer**

OpenSteer is a C++ library designed to help create steering behaviours (OpenSteer, 2008). It provides a simple framework, some helper methods, such as finding nearby flock mates, an application that demonstrates the use of the library and some pre-built steering behaviours. The main purpose of the library is to help programmers prototype new steering behaviours although the library can also be integrated into their main piece of software to provide the actual steering framework.

Although OpenSteer provides a solid framework to help build and test new steering behaviours, it does not provide any ‘building blocks’ that can be used to help create individual steering behaviours. Instead it is up to the programmer to completely program the logic of a new steering behaviour with just a few helper methods that are only helpful in very specific steering behaviours.

**Massive**

From their website (Massive Software, 2008):

“Massive is the premier 3D animation system for generating crowd-related visual effects for film and television. Using Massive, an animator or TD designs characters with a set of reactions to what is going on around them.”

Massive is very much more than a steering library. It is a complete software package to create animated characters for film and thus combines AI, steering behaviours, and an animation system. Naturally the price of this software is very high and as it commercial software details on its inner workings are extremely sparse.
2.6 Conclusion

In this literature review we started off looking at what exactly ‘steering’ is and what it is used for. We have also looked at three important areas of interest to flocking, and this project, namely goal seeking, flocking and obstacle avoidance.

Goal seeking and other ‘simple’ steering behaviours are of great importance to any A-life simulation and especially for animation. The simple nature of these behaviours means that, individually, they haven’t been researched much however they are applied a great deal and have ranged from being used in predator-prey simulations to creating finely controlled animations.

Flocking is by far the biggest research area of steering behaviours and we looked at past work that has been done on flocking from as far back as 1952. We saw that flocking was initially the study of how birds and other animals such as fish, possibly formed and moved as flocks. Reynolds in 1987 wrote what is now considered the key flocking paper where he identified three basic behaviour rules that are enough to create realistic flocks. Since that time these rules have been used to create flocking simulations on birds, fish and other animals. Some work has purely been about studying flocking animals while others have gone further, even looking at how flocking itself may evolve from groups of animals. Flocking has seen wide spread use not just in research simulations but also for creating flock animations for films and other media.

We have seen that there have been several different obstacle avoidance techniques developed and used over the years, each with different uses, ranging from simple ‘force field’ approaches to image processing. The ‘force field’ approach is particularly of interest to this project as it may become the basis for the entire system, and the large amount of work that has been done it, along with surrounding areas, will be of immense value.
Chapter 3

Project Objectives

The intention of this project is to build a simple, easy to use, generic and extensible steering library built on 'influencers', as the name of the project suggests. So what are these 'Influencers'? In our literature survey we discovered that steering behaviours are at the basic level all the same - simply a desire for an entity to go in a certain direction. We also saw the idea of 'Force Fields' that provide this attraction or repulsion desire and saw that force fields are used with some steering behaviour implementations. Indeed we saw that a flocking film, 'Eurythmy', had been created entirely with force fields. We shall take this one step further and use an extended version of force fields to help create this steering library.

So, the objectives of this project are:

1. Design and implement a steering library that is:
   (a) Stand alone - the library must not require, for instance, other libraries for it to work. This includes not being dependant on a specific 3D engine for it to function.
   (b) Generic - the steering library must not just be built to enable the creation of specific steering applications only, but instead be a 'general' library.
   (c) Extensible - the library must be easy to extend. This includes being able to easily add new features to the library but also allow users to extend the library when using it.

2. Design and implement the concept of 'Influencers'.

3. Implement several different steering behaviours using 'Influencers'.

4. Implement a few different simulations to test and demonstrate implemented steering behaviours.
5. Evaluate how useful ‘Influencers’ were in the construction of steering behaviours and how easy it was to create the different simulations with regards to using our steering library.

In the next chapter, Design Analysis, we will use objectives 1 to 3 to create the high level design of our system. Following that we will use the design and objectives 1 to 4 to implement our system, which we will discuss in the Implementation chapter. Finally we shall evaluate and test our system based on objectives 4 and 5.
Chapter 4

Design Analysis

4.1 Introduction

In this section we shall discuss the high-level design of the system, including the process of design itself. We shall start by looking at the overall architecture of the system, then at the framework of the key components, experimental ‘Influence Zones’, various different steering behaviours that we will implement in the library and finally look at the different simulations we will create to test and demonstrate our steering library.

4.2 System Architecture

The aim of this project is to build a standalone steering library which means we will also need to build an executable application that uses it, in order to test and demonstrate it. Specifically our system will consist of three components:

1. The steering library - this is the focus of this project.
2. A 3D engine - to allow rendering of the simulations.
3. A test application - this will be something developed by us that uses the steering library and 3D engine to test and demonstrate our steering library.

As this project is purely focused on the steering library, we will not go into much detail regarding the 3D engine but we will discuss the simulation application later on.

4.3 “Flocker” Architecture

In this section we will look at what the architecture of the steering library ("Flocker") needs to be so that it is generic, extensible and easy to use.
In any application that was to use this steering library, there will almost certainly be some class that represents entities in the world - for example a ‘Vehicle’ or ‘Bird’ class - that will move around guided, at least in part, by the steering library. As we want our library to be easy to use, a good idea would be to have a class that does the steering ‘thinking’ for the entity. Each frame the world entity can call on this ‘thinking’ object to re-calculate what direction and what speed it should be doing based on the different behaviours that this entity should be obeying. We shall call this the BoidBrain class.

Some steering behaviours, notably the ‘Flocking’ behaviour, require that there is a group of entities in order for the behaviour to function. As a result, we shall have a Flock class that contains a list of entities that belong to it and every BoidBrain shall have a reference to the Flock object they belong to.

As discussed in our literature survey, we shall take the idea of ‘Force Fields’ a step further and create ‘Influencers’. These influencers can then be used as building blocks to help build different steering behaviours. A simple ‘Force Field’ is typically made up of two components - the area it effects (its ‘field’) and its repulsion strength (its ‘force’). Thus we shall break our influencers into three components:

1. InfluenceZone - this represents the ‘force field’ as a whole and is made up of two components.
2. Zone - the area (field) of effect.
3. Influencer - the direction and/or speed desire for the entity.

By separating the two components - the Zone and the Influencer - it allows a greater flexibility and reuse of code. Specifically, we can create different zones and different influencers and then combine them together to create an InfluenceZone.

Originally this was the design of the steering library - a BoidBrain would be created for each entity using the steering library and would be filled with various InfluenceZones to control it. However early development showed that this was not enough to create anything more than simple seek and flee steering behaviours and lacked ease of use as to create more complex steering behaviours you would need to control the InfluenceZones yourself. As a result, we decided that we needed a further concept called Behaviour. It would be these Behaviours that the BoidBrains would store and act upon, rather than the InfluenceZones. Instead you can use the InfluenceZones to help create Behaviours.

In summary, the key parts of the steering library are:

- **BoidBrain** - Represents the brain of an entity and, based on its list of Behaviours, calculates the new velocity of the entity each frame (but does not directly control the entity).
- **Flock** - Represents a group of entities that have BoidBrains. This is for management purposes and allow the application of a Behaviour to an entire group of entities.
• **InfluenceZone** - Represents a Zone and Influencer that have been put together.

• **Zone** - Represents an area in world space. For example a sphere or box.

• **Influencer** - Represents a desire to move in a certain direction and/or go at a certain speed, possibly based on the entity’s current properties (such as position).

• **Behaviour** - Represents a steering behaviour. For example Seek and Flock.

And the flow of control is as follows:

1. A *Flock* is created.
2. An entity in the simulation is created that contains a *BoidBrain*.
3. The entity is added to the *Flock*.
4. *Behaviours* are created and added to the entity’s *BoidBrain* and/or its *Flock*.
5. The entity in question is ‘updated’ (where it performs its logic such as moving).
6. The *BoidBrain* contained in this entity is provided with relevant data (such as the entity’s current position) and asked to calculate what velocity the entity should go at.
7. The *BoidBrain* iterates through the list of its *Behaviours* and the *Behaviours* of its *Flock* asking each one to provide it with the direction and/or speed it wants the entity to do.
8. The *BoidBrain* then combines these together to get a final velocity and returns this to its entity.
9. The rest of the simulation will then update and draw itself before returning to step 5.

### 4.4 InfluenceZone

As the aim of the project is to try and create a steering library using these InfluenceZones this is perhaps the most important part of the library. As we mentioned before, an InfluenceZone will be made up of both a Zone and an Influencer. The InfluenceZone will, itself, do very little. Instead it will have wrapper methods that then call its Zone or Influencer. Specifically it will need a method to check whether a given BoidBrain is in its Zones area and methods that provide the actual influence, provided by its Influencer, for a given BoidBrain.

From this description it is clear that Zones must have methods to check for intersection between itself and a given entity and that Influencers must have methods that provide their direction and/or speed influence given an entity.
From the base *Zone* and *Influencer* classes we will then build some specific classes, though they will depend on the behaviours we build so will be discussed in the Implementation section.

### 4.5 Behaviours

The base *Behaviour* class will, like the *Zone* and *Influencer* classes, be abstract and very simple and will basically provide an interface so that objects that use them (such as *Boid-Brain* objects and *InfluencerZone* objects) don’t need to be concerned with the actual details.

The following is the design for the different steering behaviours that we had time to implement. As the aim of this project is to see if *InfluenceZones* can be used to help create steering behaviours the design will be very high level so that we not influenced by the *InfluenceZone* concept - i.e. *InfluenceZones* should not limit the different steering behaviours we can create. In the Implementation section we shall go into detail about how *InfluenceZones* helped/hindered the development of these different steering behaviours.

In general the basic design of these steering behaviours are based on steering behaviours described by Reynolds’ in his 1999 paper (Reynolds, 1999).

**Collision Avoidance**

As we discussed in the Literature Survey chapter, there are various methods ranging from the very simple ‘force field’ to much more complex vision techniques. For our collision avoidance behaviour we will create something in between. The collision avoidance behaviour class will hold a list of ‘collision meshes’ (a class we will create) where each one will represent a set of bounding objects, in the same sort of way one can do collision detection. The behaviour will shoot a ray out of the given boid and test if it hits any of the collision meshes and find the closest, as long as any of them are ‘near’ the boid (i.e. the boid is close to an object to avoid and is heading towards it). The desired direction calculated should point in the opposite direction from the collision mesh, relative to the boid’s position, causing the boid to turn away and avoid the object.

**Flee**

The flee behaviour causes the entity to move away from a specific point or area in global space. The desired direction would thus point in the opposite direction from the flee point, relative to the entity’s position.

**Flocking**

Flocking is arguably the most interesting steering behaviour and the most researched. As a result we discussed it heavily in our Literature Survey so please see that section for an in depth look at flocking - here we will just look at the basics needed for implementing it. Flocking is split into three different behaviours:

2. Cohesion: attempt to stay close to nearby flock mates.

3. Alignment: attempt to match velocity with nearby flock mates.

Thus we shall actually have four different behaviours for flocking: Separation, Cohesion, Alignment and Flocking which shall be made up of the other three.

The flocking behaviours are calculated by looking at the position and/or velocity of other nearby flock mates so we need a way to ‘find’ these ‘nearby’ flockmates. In our Literature Survey we saw that there are two different approaches to this. The first, which has been the standard for many years, is the ‘metric’ approach which works by considering all flock mates that are within a certain range of the entity. The second, and newer, approach is the ‘topological’ approach which considers the nearest flock mates up to a certain number. We shall implement both of these approaches in our flocking behaviour.

**Follow the Leader**

The follow the leader behaviour is a type of dynamic seek behaviour. A seek point is attached to an entity (the leader) and then positioned at an offset behind it. Entities using this behaviour will be attracted towards this point as long as they are not already very close to it. A common use of this behaviour is to set an entity as a leader, have it follow a path and set the rest of its flock to follow it while still under the flock behaviour. This allows one to make a flock follow a path without have to control the exact movement of every entity.

**Point Follow**

The point follow behaviour is a type of path follow behaviour. The point follow behaviour will consist of a number of points in space, in effect creating a path, that the entity will follow. There should be the ability to set it so after reaching the end of the path it starts again or loops back (follows the path in reverse) as if it was patrolling.

**Seek**

The seek behaviour causes the entity to move towards a specific point or area in global space. The desired direction would this point in the direction towards the seek point, relative to the entity’s position.

**Basic Behaviour**

This behaviour is not a normal steering behaviour and is an exception to the above note about not mentioning *InfluenceZones*. It will just contain a list of *InfluenceZones* and apply their influences. This behaviour’s main use is to help test different *InfluenceZones* but could also be used if *InfluenceZones* are enough to create a desired behaviour.

As you can see we implementing a range of different steering behaviours, although this is certainly not an exhaustive list. However the point of this steering library is to allow others to easily create their own steering behaviours. Indeed implementing these different steering behaviours will in fact be a test of the flexibility and ease of use of our steering library - most importantly whether *InfluenceZones* make it easier to create new behaviours.
4.6 Simulations

In order to help test, evaluate and demonstrate our steering library we need to create a few different simulations. Initially we will create different simulations to test new Influence-Zones and Behaviours as and when we need to so we will not detail them here. At the end of the project though we shall implement the following simple simulations to help evaluate and demonstrate our library and to provide samples for those wishing to see an example of how our steering library can be/is used:

1. Sim 1: Flocking - This first simulation will demonstrate the flocking behaviour and will allow the user to create new flocks during the simulation with different properties, such as using metric or topological techniques to find nearby flock mates.

2. Sim 2: Point Following - The second simulation will demonstrate simple path following which will show how InfluenceZones can be used dynamically.

3. Sim 3: Follow the Leader, Point Following and Flocking - The third simulation will demonstrate the use of multiple behaviours to create emergent behaviour. Specifically it will show one boid following a set path, with the others following it while still maintaining a flock.

4. Sim 4: Collision Avoidance and Flocking - The final simulation will demonstrate the collision avoidance behaviour with a scene consisting of a flock (or flocks) flying around avoiding collisions while maintaining their flock.

4.7 Summary

In this chapter we have seen exactly what we aim to built and what its high level structure is. The concept of ‘Influence Zones’ has been made concrete and care has been taken to ensure they are generic and extensible. We have designed the different steering behaviours with enough detail to be able to implement them but ensuring that the InfluenceZone concept did not influence/limit their design in any way. Finally we designed the different simulations we would create in order to test, evaluate and demonstrate our steering library.

In the next chapter we look in detail at the implementation of our system including the key classes, how we kept our system generic and extensible but at the same time easy to use and how InfluenceZones helped or hindered the development of specific steering behaviours.
Chapter 5

Implementation

5.1 Introduction

In the above section we looked at the high level design of our steering library and also at the simulations we would create. In this section we shall look at detail at the implementation of the system, in particular how InfluenceZones were used with regards to the different Behaviours we implemented. We will also take a brief look at the tools and techniques we used when creating our system.

5.2 Tools Used

The first choice in the implementation of any system is to decide on the programming language, APIs and any tools that will be used. We will now take a brief look at our choices with regards to these issues and the reasons for them.

Language Choice and Graphics API

As there is very limited time to complete this project, it will be important to choose a programming language that is, at least for us, fast to develop with and also to use a ready made 3D engine, so that we do not have to spend time developing it. Because of this, we have chosen to develop our steering library in C#.NET and the XNA Framework (a 3D game development API) for the following reasons:

- We have experience with both C#.NET and the XNA Framework.
- It is fast to develop with.
- It has fast execution speed.
We have previously developed a simple 3D engine ("BOB XNA Engine") with C#.NET and XNA meaning we do not have to spend time developing one now, we already know how to use it and can easily modify it if needed.

Integrated Development Environment

As we have chosen to use C#.NET and XNA the natural choice of an IDE to use is Visual Studio. Specifically we will use ‘Microsoft Visual Studio 2005 Professional Edition’. The IDE’s features such as code colouring, intellisense, file and asset management, build system and auto-refactoring will greatly ease the development of our system.

3D Engine

As stated above, we have previously developed a simple 3D engine using C#.NET and XNA called the "BOB XNA Engine". This will more than provide the necessary 3D rendering/framework abilities that we will need to test and demonstrate our steering library.

5.3 BoidBrain

The BoidBrain class represents the brain of the entity being steered but its implementation is actually very simple. Its primary method is the CalcInfluence which we will now step through.

The method itself takes in the current position of the entity as well as its bounds as a radius (BoidBrain stores a bounding sphere that contains its entity for use in intersection tests), returns the newly calculated velocity as well as passing out how much it has rotated by and on what axis:

Listing 5.1: BoidBrain CalcInfluence Method Input and Output

```csharp
public virtual Vector3 CalcInfluence(Vector3 position, float boundsRadius, out float rotationAngle, out Vector3 rotationAxis)
```

After updating its stored position and bounding sphere, it then iterates through all its Behaviours and its Flock’s Behaviours and adds up their combined direction and speed influences:

Listing 5.2: BoidBrain CalcInfluence Method Behaviour Iteration

```csharp
// Iterate through all the behaviours and add up their total 'influences'
foreach (Behaviour behaviour in m_Behaviours.Values) {
    if (behaviour.DirectionEnabled) {
```
After this it then calculates the angle and axis its entity needs to rotate in order to move from its current direction to the ‘desired’ one, clamps this value with a maximum, and then rotates its local forward, up and right vector directions. It then calculates the average ‘desired’ speed or uses a ‘preferred’ speed if there is no desire to go at a certain speed.

Finally it calculates the final desired velocity and returns this:

```
Listing 5.3: BoidBrain CalcInfluence Method Velocity Calculation

// Calculate the new velocity (clamping it by min/max speeds)
m_Velocity = MathsHelper.Clamp(m_ForwardDirection * desiredSpeed, m_MinSpeed, m_MaxSpeed);

// Return the rotated angle and new velocity
rotationAngle = angle;
rotationAxis = axis;
return m_Velocity;
```

The code above for calculating a new velocity is very simple and does not consider mass, thrust or acceleration. However for the purposes of this project, it is sufficient.

A BoidBrain object will typically be attached to an entity object in the simulation, such as a bird or car, as is probably apparent by its name. To help with this we created an interface called IBoid. Classes that want to have a BoidBrain should implement this interface which just insures that there is a way to access its BoidBrain (using a C# Property). You might
wonder what the point of this is if all it does is ensure there is a BoidBrain - after all why not just pass the BoidBrain around? Indeed for most of our methods you will see BoidBrains being used but this is purely because these are methods being called, directly or indirectly, by a BoidBrain itself and thus has to pass itself as it knows nothing of the object that is using it. Instead the usefulness of the IBoid interface is only apparent in the Behaviours that need to use a specific BoidBrain and not just the one that is calling it asking for its influence. This is useful because it provides a way to help users manage their entities. For instance, the FollowLeaderBehaviour takes in an IBoid as the leader. If it only took in a BoidBrain, which is all it needs, users of the behaviour would not be able to ask for a reference to the entity which is being followed, instead they would just get the BoidBrain which would not be as useful.

5.4 InfluenceZone

The InfluenceZone class is purely a management class that combines a given Influencer and a given Zone. All of its methods (barring its constructors) are just wrappers for the Influencer and Zone that it contains so we shall just look at these two classes.

5.4.1 Zone

The Zone class is an abstract class that represents some area in space and contains a few abstract methods:

Listing 5.4: Zone Methods

public abstract Vector3 Position
{
    get;
    set;
}
public abstract bool Intersects(BoundingBox box);
public abstract bool Intersects(BoundingSphere sphere);
public abstract bool Contains(BoundingBox box);
public abstract bool Contains(BoundingSphere sphere);
public abstract float CalcScaledWeight(BoidBrain boid, float minClamp,
    float maxClamp, bool inverse);

As you can see, every Zone must have a changeable position and some intersection/containment tests but more interesting is the ‘CalcScaledWeight’ method. This method allows a Zone class to set how weight percentage is scaled based on the BoidBrain object - typically this will be the boid’s position. We will see an example of this in the SphereZone.

With this base class finished, we created some basic Zones:

Sphere Zone

This Zone represents a sphere in space and uses the built in ‘BoundingSphere’ class from
CHAPTER 5. IMPLEMENTATION

the XNA API to do intersection and containment tests. Its implementation of the ‘CalcScaledWeight’ method is obvious - the percentage tends towards zero the closer to the centre of the sphere the boid is, and at the edge of the sphere, and beyond, it is at 100% (1.0f). If the inverse flag is set it is the other way around (i.e. higher percentage the closer to the centre of the sphere the boid is, 0% if on the outside).

Box Zone
This Zone represents an axis-aligned box in space and uses the built in ‘BoundingBox’ class from the XNA API to do intersection and containment tests.

Infinite Zone
This Zone represents a ‘shape’ that contains everything in the ‘world’. Regardless of where the boid is it will also be contained inside an InfiniteZone. At first it may seem that the position of this Zone is irrelevant however we shall see that this can be important as some Influencers use the Zones position when calculating their influence.

These three Zones are just the ones that we decided to make. The Zone class is both generic and extensible meaning that anyone could create new Zones and use them immediately with the InfluenceZone. For instance, it would be perfectly reasonable and easy to create a ‘Plane Zone’ where an intersection could be considered ‘true’ if the boid is on a particular side on the plane.

Our only concern with this class is that the methods and method names might not be appropriate or generic enough. At the moment they are adequate enough as the only thing that calls their methods are the InfluenceZones and user built Behaviours and they only use the bounding sphere version of the ‘Intersection’ method, to test whether they intersect with a boid. An improvement would be to have better method names and to perhaps have it so that Zones can do intersection checks on each other, or to create a new class that represented lower level 3D shapes (such as sphere and box). This could then also be used, and added to and extended, by users of the library. It would also allow easier porting of the steering library as any of the built in XNA 3D shapes that we use, such as bounding spheres and boxes, would be neatly wrapped.

5.4.2 Influencer

The second half of an InfluenceZone is an Influencer. As its name implies, an Influencer is responsible for calculating some sort of influence to apply to a given BoidBrain. It does not apply this ‘desire’ itself, just returning it to the calling object which will typically be an InfluenceZone.

The base Influencer is, like the base Zone class, an abstract class to allow creation of new Influencers. Its methods are:

Listing 5.5: Influencer Methods

```csharp
public virtual Vector3 CalcDirection(Zone zone, BoidBrain boid)
public virtual float CalcDirectionWeight(Zone zone, BoidBrain boid)
public virtual Vector3 CalcWeightedDirection(Zone zone, BoidBrain boid)
```
public virtual float CalcSpeed(Zone zone, BoidBrain boid)  
public virtual float CalcSpeedWeight(Zone zone, BoidBrain boid)  
public virtual float CalcWeigthedSpeed(Zone zone, BoidBrain boid)  

You will notice that these are virtual and not abstract methods. Each of these methods returns either 0.0f (zero) or a 3-dimension vector containing zeros, depending on their return type. This is so that new Influencers that inherit from this base class do not have to implement any of these methods if they have no wish to use them. For instance, a new Influencer may only wish to apply a direction desire so there is no point in forcing the programmer to implement the speed methods.

You will also notice that there are three methods for each desire type (direction, speed). The InfluenceZone class has a wrapper method for each of these methods and the Behaviour class, as we shall see, has wrapper methods for these methods. Looking back at the BoidBrain class you will see that it only actually uses three of these methods but by having them all it allows greater flexibility and uniformity.

With this base class finished, we created some basic Influencers:

BasicInfluencer  
This class is, as you would expect, the simplest possible Influencer.

Listing 5.6: BasicInfluencer  
public BasicInfluencer(Vector3 direction, float directionWeight, float speed, float speedWeight)

It just stores a vector for the direction influence and a float for the speed influence along with respective weights. When asked to calculate the influences it returns the relevant stored values.

AttractorInfluencer  
This Influencer calculates a vector from the given BoidBrain’s current position towards the given Zone’s position creating an ‘attraction’ towards the Zone:

Listing 5.7: AttractorInfluencer  
public override Vector3 CalcDirection(Zone zone, BoidBrain boid)  
{
    Vector3 direction = zone.Position - boid.Position;
    if (direction != Vector3.Zero)  
    {  
        direction.Normalize();
    }

    return direction;
}

The constructor takes in a float as a ‘strength’ value which is used to weight the influence.

RepulserInfluencer  
This Influencer is the opposite to the AttractorInfluencer in that it calculates a vector from
the given Zone’s position towards the given BoidBrain’s position creating a ‘repulsion’ away from the Zone:

Listing 5.8: RepulserInfluencer

```csharp
public override Vector3 CalcDirection(Zone zone, BoidBrain boid)
{
    Vector3 direction = boid.Position - zone.Position;
    if (direction != Vector3.Zero)
    {
        direction.Normalize();
    }

    return direction;
}
```

Like the AttractorInfluencer it has a float ‘strength’ to weight the influence. You may notice that it is possible to make an AttractorInfluencer repulse and a RepulserInfluencer attract by using a negative strength.

Again, these are just an example of the different Influences one could create but we found that these were all we needed to create our different Behaviours. Some other examples of Influencers that could be created include an ‘orbiter’ that causes the given BoidBrain to orbit around a point or Zone (remember that an ‘influence’ is a ‘desire’ and not a ‘force’) and an Influencer that causes boid’s to slow down the closer they are to the centre of the Zone which could be used to help create an ‘Arrival’ Behaviour. Indeed the Zone method ‘CalcScaledWeight’ could be of use here.

Like the Zone class this is a simple class but when combined to create an InfluenceZone or Behaviours it becomes a very useful tool and piece of a more complex system. The base Influencer class has allowed easy creation of new Influencers and provides a generic interface. However there are perhaps some more things that could be added to allow greater ease of use and make it even more generic. This could include ‘weight/strength’ fields and modification which at the moment has to be implemented by each new Influencer. This would make it easier to use and more generic but at the cost of flexibility as each new Influencer may have different needs when it comes to influence weightings/strength. The balance between ease or use, generic base classes, extensibility and flexibility is a difficult one to manage and although in general we think we have struck a good balance, only extensive use of the steering library will be able to fully test this.

### 5.5 Behaviour

In order to test whether the InfluenceZone concept helps create steering behaviours, we created several different steering behaviours. These behaviours also serve as samples for how to create new behaviours using our library as well as being behaviours that can directly be used in an application.
Here we shall look at the low level implementation of each of these behaviours and look at how we used InfluenceZones to help create them. They are listed in the approximate order they were implemented in:

**Seek**
The seek behaviour is able to use the AttractorInfluencer in an InfluenceZone without any modifications. The first constructor uses an InfiniteZone as the Zone so that the entity is always drawn to it.

The second constructor is more interesting. This constructor allows the user to specify their own Zone and whether the entity should seek the position of the Zone or the Zone itself. If seek the point is chosen it means the entity will only seek when inside the Zone while if seeking the Zone is chosen the entity will only seek it when outside the Zone. This doesn’t require any extra work in the seek behaviour as the InfluenceZone class has a ‘InverseZone’ option which means that the intersection/containment test method booleans are flipped (i.e. if the entity intersects the Zone the returned value is false rather than true):

Listing 5.9: SeekBehaviour Second Constructor

```csharp
public SeekBehaviour(string name, Zone zone, float directionWeight, bool seekPoint)
    : base(name)
{
    m_InfZone = new InfluenceZone(new AttractorInfluencer(directionWeight), zone, true, false);
    m_InfZone.InverseZone = !seekPoint;
    SpeedEnabled = false;
}
```

When the behaviour is asked for its directional influence all it has to do is use its InfluenceZone:

Listing 5.10: Seek CalcWeightedDirection

```csharp
public override Vector3 CalcWeightedDirection(BoidBrain boid)
{
    if (m_InfZone.Intersects(boid.BoundingSphere))
    {
        return m_InfZone.CalcWeightedDirection(boid);
    }
    else
    {
        return Vector3.Zero;
    }
}
```

You can see that the InfluenceZone concept fits perfectly with the seek steering behaviour and it allowed extremely fast implementation of it. The InfluenceZones features, such as
inverting intersection tests, also meant we could create a more interesting and flexible seek behaviour without any extra work.

BasicBehaviour
This behaviour does not have any specific logic as it is a generic behaviour that just contains a list of InfluenceZones that are added by the user. When asked for the direction or speed influence it iterates through all the behaviours and combines their influences together:

Listing 5.11: BasicBehaviour CalcWeightedDirection

```csharp
public override Vector3 CalcWeightedDirection ( BoidBrain boid )
{
    Vector3 direction = Vector3.Zero;
    foreach ( InfluenceZone infZone in m_InfluenceZones )
    {
        if ( infZone.DirectionEnabled &&
            infZone.Intersects ( boid.BoundingSphere ) )
        {
            direction += infZone.CalcWeightedDirection ( boid );
        }
    }
    return direction ;
}
```

This behaviour was mostly used for testing new influencers and for seeing whether InfluenceZones could provide some steering behaviours on their own without extra logic.

Flee
The flee behaviour is identical to the SeekBehaviour except that it uses a RepulserInfluencer rather than an AttractorInfluencer. The relationship between the flee and seek behaviours is the same as the relationship between the RepulserInfluencer and AttractorInfluencer in that using a negative weight/strength causes it to behaviour like the other.

Flocking
As the flocking behaviour is made up of three separate behaviours we split it thus so and created a FlockingBehaviour class that contains all three behaviours. This means that the three flocking behaviours can be re-used in other behaviours, or on their own, but if a standard flocking behaviour is needed then FlockingBehaviour can be used.

All three of the behaviours (CohesionBehaviour, SeparationBehaviour, AlignmentBehaviour) have the same direction influence methods. Depending on whether the user of the behaviour has chosen to find nearby flock mates using a metric or topological system, nearby flock mates are found and then used to calculate any influence:

Listing 5.12: CohesionBehaviour CalcWeightedDirection

```csharp
public override Vector3 CalcWeightedDirection ( BoidBrain boid )
{
    List<BoidBrain> flockmates ;
    if ( m_FlockmateFinder == FlockmateFinder.Metric )
    {
        flockmates . Add ( boid , 0.0f );
        for ( int i = 0 ; i < flockmates . Count ; i++ )
        {
            Vector3 direction = Vector3 . Zero ;
            foreach ( InfluenceZone infZone in m_InfluenceZones )
            {
                if ( infZone.DirectionEnabled &&
                    infZone.Intersects ( flockmates [ i ] . BoundingSphere ) )
                {
                    direction += infZone.CalcWeightedDirection ( flockmates [ i ] ) ;
                }
            }
            return direction ;
        }
    }
    return Vector3 . Zero ;
}
```
CHAPTER 5. IMPLEMENTATION

flockmates = SteeringHelper.GetCloseFlockmates(boid, boid.Flock, m_NeighbourRange);
}
else // Topological
{
    flockmates = SteeringHelper.GetClosestFlockmates(boid, boid.Flock, m_NeighbourCount);
}
ReCalcInfluenceZone(flockmates, boid);

return m_InfZone.CalcWeightedDirection(boid) * mDireccionWeight;
}

Because finding nearby flock mates can be an expensive operation, we also provided methods that do not calculate nearby flock mates but instead accept a list of nearby flock mates as a parameter. It is these methods that are used by the FlockingBehaviour class as this class finds nearby flock mates itself and then passes this information to the three sub-flocking behaviours meaning that nearby flock mates are only calculated once and not three times:

Listing 5.13: FlockingBehaviour CalcWeightedDirection

class FlockingBehaviour
{
    public override Vector3 CalcWeightedDirection(BoidBrain boid)
    {
        List<BoidBrain> flockmates;
        if (m_FlockmateFinder == FlockmateFinder.Metric)
        {
            flockmates = SteeringHelper.GetCloseFlockmates(boid, boid.Flock, m_NeighbourRange);
        }
        else // Topological
        {
            flockmates = SteeringHelper.GetClosestFlockmates(boid, boid.Flock, m_NeighbourCount);
        }

        Vector3 steeringWeightedDirection =
            m_CohesionBehaviour.CalcWeightedDirection(flockmates, boid) +
            m_SeparationBehaviour.CalcWeightedDirection(flockmates, boid) +
            m_AlignmentBehaviour.CalcWeightedDirection(flockmates, boid);

        return steeringWeightedDirection * m_Weight;
    }
}

It is obviously in the 'ReCalcInfluenceZone' method of the three flocking behaviours where they differ. The CohesionBehaviour finds the average position of nearby flock mates and moves its InfluenceZone's SphereZone there which contains an AttractorInfluencer to influence the boid to move to that position. This behaviour also makes use of the Zone 'CalcScaledWeight' ability to scale the weight of the influence based on the position of the
boid:

Listing 5.14: CohesionBehaviour ReCalcInfluenceZone Weight Scaling

```csharp
// Calc strength (based on distance)
float str = 0.0f;
switch (m_ScaleWeight)
{
    case ScaleWeight.DontScale:
        str = 1.0f;
        break;
    case ScaleWeight.Scale:
        str = m_InfZone.Zone.CalcScaledWeight(boid, 0.0f, 1.0f, false);
        break;
    case ScaleWeight.InverseScale:
        str = m_InfZone.Zone.CalcScaledWeight(boid, 0.0f, 1.0f, true);
        break;
}
```

This ability of the SphereZone class proved to be very useful as it allowed us to ‘balance’ the cohesion and separation flocking behaviours so that boids would generally flock together at a consistent distance from each other (at the point where the weighted influence of the separation and cohesion behaviours were both at half strength).

The SeparationBehaviour works by calculating a direction away (repulsion) from each of the flock mates, relative to the boid in question, and then combines them together to form the directional influence of the BasicInfluencer that the InfluenceZone of the behaviour uses. It also, like the CohesionBehaviour, uses the ‘CalcScaledWeight’ ability of the SphereZone class. In the case of the flock mate being on top of the boid, which would result in a repulsion in no direction, a random direction is created.

We could have implemented the SeparationBehaviour by creating an InfluenceZone containing a RepulserInfluencer for each of the nearby flockmates and combining their repulsion influence together to form the final directional desire. This perhaps would have made more use of the Influence Zones and RepulserInfluencer and would have been more robust. Our solution though, which calculates the repulsions itself and combines them on the go, uses less resources but is perhaps not in keeping with the idea of using Influence Zones as much as we can.

The third and final flocking behaviour is AlignmentBehaviour and is one of the simplest behaviours. When asked for its influence, it just finds the average direction of the boid’s flock mates and returns it:

Listing 5.15: AlignmentBehaviour ReCalcInfluenceZone

```csharp
private void ReCalcInfluenceZone(List<BoidBrain> flockmates, BoidBrain boid)
{
    // Add up all the flockmate directions
    Vector3 avgDirection = Vector3.Zero;
```
foreach (BoidBrain flockmate in flockmates)
{
    avgDirection += flockmate.ForwardDirection;
}

// Calc average direction of flockmates
if (avgDirection != Vector3.Zero && flockmates.Count != 0)
{
    avgDirection = (avgDirection / (float)flockmates.Count) - 
    boid.ForwardDirection;
    if (avgDirection != Vector3.Zero)
    {
        avgDirection.Normalize();
    }
}

((BasicInfluencer)m_InfZone.Influencer).Direction = avgDirection;

m_InfZone.Zone.Position = boid.Position;

The ‘normal’ implementation of the alignment behaviour is to find the average velocity, and
you will notice that we just find the average direction and ignore the magnitude (speed).
This is simply because we implemented this behaviour early on when we only interested
in direction and not speed, which we keep separate. However it would be very easy to
create an alignment behaviour that also calculated the average speed. Indeed one would be
able to inherit from the AlignmentBehaviour class and just implement the speed influence
calculation methods and if they were using the FlockingBehaviour class do the same there.
Setting both of these behaviour’s ‘SpeedEnabled’ flags would then cause the BoidBrain
class to ask for a speed influence.

The InfluenceZone concept was a natural fit for the flocking behaviours and the flexibility
of the Behaviour class was also a big help in terms of being able to use a behaviour class
(in this case the FlockingBehaviour class) to manage and combine other behaviours. The
implementation of the system means that the BoidBrain class, and anyone who uses the
FlockingBehaviour class, is completely oblivious to this - as it should be. A common theme
in the flocking behaviours is a need to calculate averages - such as average flock mate
position and direction - so it may be worth looking into the possibility of adding standard
methods of these sort of things to the library helper classes (see details of these classes
below).

Follow the Leader
The follow the leader behaviour doesn’t actually use an InfluenceZone directly, instead
it uses the SeekBehaviour. The behaviour takes an IBoid as the ‘leader’ and an offset
distance and every time it is asked for the directional influence it moves the SeekBehaviours
destination point to just behind the leader boid:

Listing 5.16: FollowLeaderBehaviour ReCalcInfluenceZone

private void ReCalcInfluenceZone(BoidBrain boid)


{  
    m_SeedBehaviour.Destination = m_Lleader.BoidBrain.Position –  
    (m_Lleader.BoidBrain.ForwardDirection * m_FollowDistance);
}

There are slightly more complex versions of ‘follow the leader’ though we found this implementation was enough to create effective following. This behaviour doesn’t show the usefulness of InfluenceZones though it does show the flexibility of Behaviours in that they can be built on top of each other.

**Point Follow**

This behaviour contains a list of points in space (3D vectors) that can be added to or cleared at any time. An InfluenceZone consisting of an AttractorInfluencer is used to draw the entity to each point in turn. Once we have reached the current point, we move onto the next one:

Listing 5.17: PointFollow ReCalcInfluenceZone Method - Moving onto the next point

```csharp
// If we are at the current point, move onto the next one
if (m_InfZone.Intersects(boid.BoundingSphere))
{
    if (!m_ReverseFollow)
    {
        m_CurrentIndex++;  
    }
    else
    {
        m_CurrentIndex--;  
    }
}
```

Here you can also see one of the options - ‘Reverse Follow’ - which, when true, means that the entity will follow the list of points in reverse order.

If the point we are moving to next is valid, we just carry one, otherwise we have reached the end of the path. What we do next is dependant on the ‘RestartWhenDone’ enumeration option, which allows us to set whether we just stop when we get to the end of the path, start again, or loop:

Listing 5.18: PointFollow ReCalcInfluenceZone Method - At End of Path

```csharp
// If we have somewhere else to go...
if (m_CurrentIndex >= 0 && m_CurrentIndex < m_Points.Count)
{
    // Set zone position to current destination point
    m_InfZone.Zone.Position = m_Points[m_CurrentIndex];
}
else
{
    // We have nowhere else to go
```
switch (m_RestartWhenDone)
{
    case RestartWhenDone.Yes:
        if (!m.ReverseFollow)
        {
            m_CurrentIndex = 0;
        }
        else
        {
            m_CurrentIndex = m_Points.Count - 1;
        }
        break;
    case RestartWhenDone.No:
        m_Finished = true;
        break;
    case RestartWhenDone.Loop:
        m ReverseFollow = !m.ReverseFollow;
        if (!m.ReverseFollow)
        {
            m_CurrentIndex = 0;
        }
        else
        {
            m_CurrentIndex = m_Points.Count - 1;
        }
        break;
}

m_InfZone.Zone.Position = m_Points[m_CurrentIndex];

This behaviour was able to utilise the InfluenceZone concept in a similar way to the seek behaviour. However this behaviour required quite a bit of behaviour specific logic in order to be implemented. There are certainly some improvements to the InfluenceZone concept that could have further helped the implementation of this behaviour, such as the ability to program a Zone to move when it is intersected. This would obviously directly help with this behaviour but we think it would also be a useful ability in general.

It should also be noted that this behaviour updates itself as the boid moves from point to point which means that, unlike most of the other behaviours, care must be taken when adding an object of this behaviour to boid(s). If added to a flock, rather than a boid, or to multiple boids, the InfluenceZone will move to the next point as soon as any one of the boids gets to the current point. This isn’t to say that the implementation of this behaviour is wrong or that a new instance of it should be given to each path following boid, just that its behaviour is different depending on how it is used.

**Collision Avoidance**

This behaviour consists of two classes. The first is the CollisionAvoidanceBehaviour class that inherits from the Behaviour class. When asked for its directional influence by a
CHAPTER 5. IMPLEMENTATION

*BoidBrain* it recalculates where its *InfluenceZone*, which consists of a *RepulserInfluencer* and an *InfiniteZone*, should be and whether there is a risk of any collisions:

Listing 5.19: Collision Avoidance Behaviour CalcWeightedDirection Method

```csharp
public override Vector3 CalcWeightedDirection(BoidBrain boid)
{
    ReCalcInfluenceZone(boid);
    if (m_CollisionApproaching)
    {
        return m_InfZone.CalcWeightedDirection(boid);
    }
    else
    {
        // No risk of collision at this time
        return Vector3.Zero;
    }
}
```

As you can see, if there is a risk of a collision, it applies the *InfluenceZone*.

The ‘ReCalcInfluenceZone’ method is where the interesting logic happens. It starts off by firing a simple ray from the *BoidBrain* in the direction it is travelling:

Listing 5.20: Collision Avoidance Behaviour ReCalcInfluenceZone Ray Firing

```csharp
// Calculate the ray shooting out from the boid
m_BoidRay = new Ray(boid.Position, boid.ForwardDirection);
```

It then iterates through all of the *CollisionAvoidanceMesses*, which is the second class mentioned above, and checks whether the ray intersects any of them, and, if so, which is the closest:

Listing 5.21: Collision Avoidance Behaviour ReCalcInfluenceZone Collision Checks

```csharp
// Go through all the collision meshes and find the closest intersecting one
CollisionAvoidanceMesh closestMesh = null;
bool firstHit = true;
float closestDist = 0;
for (int i = 0; i < m_CollisionMeshes.Count; i++)
{
    float? dist = m_CollisionMeshes[i].Intersects(m_BoidRay);
    if (dist != null)
    {
        // Is this the closest collision so far that is within the min avoidance range?
        if ((dist < closestDist || firstHit) && dist <=
        m_CollisionMeshes[i].AvoidanceRange)
        {
            firstHit = false;
        }
    }
}```
If a potential collision is found, the *InfluenceZone’s Zone* is moved to the position of that *CollisionAvoidanceMesh* so that the boid is influenced away from it:

**Listing 5.22: Collision Avoidance Behaviour ReCalcInfluenceZone Zone Movement**

```java
// If there is a potential collision approaching, set the repulsor influence zone to it
if (closestMesh != null)
{
    m_InfZone.Zone.Position = closestMesh.BroadBound.Center;
    m_CollisionApproaching = true;
}
else
{
    m_CollisionApproaching = false;
}
```

Typically with collision detection each collidable object in the simulation will have a collision mesh made out of simple objects, such as spheres or boxes, which are used for collision checks. This method is used because trying to do exact collision checks using all the polygons that make up the model of the object would be incredibly expensive. We are using a similar system in our collision avoidance behaviour using the *CollisionAvoidanceMesh* class we mentioned above. A *CollisionAvoidanceMesh* should be created for each object in the simulation that the user wishes entities to avoid and added to the *CollisionAvoidanceBehaviour* object that their entities are using. Each *CollisionAvoidanceMesh* consists of a ‘Broad’ sphere which should enclose the object and a list of ‘Narrow’ spheres which more closely match the object. These two groups are used to help speed up the ray intersection tests by first checking against the larger, and all encompassing, ‘Broad’ sphere:

**Listing 5.23: CollisionAvoidanceMesh Intersects Method**

```java
public float? Intersects(Ray ray)
{
    // Check if we hit the broad sphere
    float? closestDist = m_BroadBound.Intersects(ray);
    if (closestDist != null)
    {
        // We do, so check if we hit any of the narrow (inner) spheres
        bool firstNarrowHit = true;
        for (int i = 0; i < m_NarrowBound.Count; i++)
        {
            float? dist = m_NarrowBound[i].Intersects(ray);
            if ((dist < closestDist || firstNarrowHit)
            {
```
The collision avoidance behaviour is one of the more complex behaviours we implemented. The InfluenceZone concept certainly helped, but there was still a fair amount of behaviour specific logic to write. One idea we had towards the start of the project, that was scrapped when we introduced ‘Behaviours’, was to be able to have collections of Zones and InfluenceZones. In this way, an influence could have an effect not just when the entity was inside a specific Zone but in one of a number of Zones. This could help with this behaviour by extending the Zone intersection tests to include tests for ray intersections. You would then have little need for the CollisionAvoidanceMesh as its purpose will already have been more or less provided for.

This behaviour itself could be improved by projecting a cylinder out from the boid, rather than a simple ray, because at the moment it is possible for the ray to miss a potential collision that the entity may hit with its ‘side’.

### 5.6 Helpers

In order to help with the creation of the steering library, and to avoid repetition of code, two helper classes were created:

- **MathsHelper**
  This static class contains some mathematics methods ranging from truncating a vector to finding the angle between two vectors. If the steering library was ever extended, this would be a useful place to add new mathematics methods.

- **SteeringHelper**
  This static class contains some methods to help with steering behaviours. Specifically it contains two methods to help find nearby flock mates - one that does it topologically:

  Listing 5.24: SteeringHelper Topological Method

```java
public static List<BoidBrain> GetClosestFlockmates(BoidBrain boid, Flock flock, int number) {
    List<DistToFlockmate> closestFlockmates = new List<DistToFlockmate>();

    // Add the flockmates to the sorted list, with the key being their distance to the given boid
```
for (int i = 0; i < flock.Boids.Count; i++)
{
    if (flock.Boids[i].BoidBrain != boid)
    {
        DistToFlockmate flockmate;
        flockmate.Flockmate = flock.Boids[i].BoidBrain;
        flockmate.Distance = Vector3.DistanceSquared(boid.Position,
            flockmate.Flockmate.Position);
        closestFlockmates.Add(flockmate);
    }
}

// Sort the list
closestFlockmates.Sort(new
    Comparison<DistToFlockmate>(CompareDistances));

// Get rid of distant flockmates if we have too many
while (closestFlockmates.Count > number)
{
    closestFlockmates.RemoveAt(closestFlockmates.Count - 1);
}

// Put all the flockmates into a list for returning
List<BoidBrain> returnValue = new
    List<BoidBrain>(closestFlockmates.Count);
foreach (DistToFlockmate flockmate in closestFlockmates)
{
    returnValue.Add(flockmate.Flockmate);
}
return returnValue;

And one that does it metrically:

Listing 5.25: SteeringHelper Metric Method

public static List<BoidBrain> GetCloseFlockmates(BoidBrain boid, Flock flock, float radius)
{
    // Create a bounding sphere from the given radius for intersection testing
    BoundingSphere area = new BoundingSphere(boid.Position, radius);
    List<BoidBrain> closeFlockmates = new List<BoidBrain>();

    // Find all the flockmates in the flock that are within the radius
    foreach (IBoid flockmate in flock.Boids)
    {
        if (flockmate.BoidBrain != boid)
{  
    // Is the flockmate within the radius?  
    if (area.Intersects(flockmate.BoidBrain.BoundingSphere))  
    {  
        closeFlockmates.Add(flockmate.BoidBrain);  
    }  
}  
return closeFlockmates;  
}

These methods are used in the FlockingBehaviour class as well as its related three Behaviours. This means that when using the flocking behaviour the programmer can choose whether to use the topological or metric technique to find nearby flock mates.

Again, this would be a useful class to extend with more methods to help with creating new steering behaviours.

5.7 Summary

In this section we have seen how and what we implemented in our system. First we saw what tools and technologies we used to help implement our system and why we choose to use them.

We then looked at the BoidBrain class, the most important ‘framework’ class, which gave a good look at the ‘flow’ of our steering library and how to use it. Following this we saw how we implemented the ‘Influence Zone’ concept, specifically how we created it to be flexible with Influences and Zones that can be implemented separately and generically and then merged together to form an InfluenceZone. We also discussed how we used this in practice with examples of some Influencers and Zones we had created ourselves.

In order to see how useful the InfluenceZone concept was for helping to create steering behaviours, we created a range of behaviours ourselves. We found that they did indeed help with the implementation of steering behaviours in that they saved programming time, increased code re-use and speeded up testing of behaviours.

In the next section we will look at the different simulations we created in order to test and demonstrate our library. This includes both how easy it was to use our library in an application and the steering behaviours using our InfluenceZones.
Chapter 6

Evaluation and Testing

6.1 Introduction

In the previous section, Implementation, we discussed, among other things, how the InfluenceZone concept helped/hindered the development of some steering behaviours. This evaluation of InfluenceZones was only concerned with how useful they were in programming new behaviours. In this section we will discuss four different simulations we created to test and demonstrate the steering library and we will evaluate our steering library as a whole, including InfluenceZones.

6.2 Simulation 1 - Flocking

This first simulation demonstrates the FlockingBehaviour. Using the console command line in the application, you can create new flocks at any point and specify:

1. The flock size.
2. The cohesion behaviour weight.
3. The separation behaviour weight.
4. The alignment behaviour weight.
5. The cohesion scale range.
6. The separation scale range.
7. Whether the flock should use a metric or topological technique for finding flock mates.
8. The neighbour range (when using metric) or the neighbour count (when using topological).
In order to test out the flocking behaviour, we ran a simulation where we created four different flocks. Each flock was of size 30 and its behaviour weights were all one:

1. A flock with scale ranges of 200 that used the topological technique and a neighbour count of 7.
2. A flock with scale ranges of 200 that used the metric technique and a neighbour range of 200.
3. A flock with scale ranges of 50 that used the topological technique and a neighbour count of 7.
4. A flock with scale ranges of 500 that used the metric technique and a neighbour range of 100.

At the start of the simulation all the boids were randomly placed in a 1000x1000x1000 cube area (see 6.1). Very quickly the boids using the topological technique were grouping together and flock 3, with a scale range of 50, were already a lot closer together than flock 1, with a scale range of 200.

Within a few seconds all the boids in flock 1 were flocking together in a loss group, while all the boids in flock 3 were flocking in a very tight group. The boids in flock 2 were mostly flocking together at this point in the same sort of loose grouping as flock 1 (see 6.2). Indeed it can be observed from this that when the boids flock together the point at which they find balance is when the cohesion and separation behaviours are having an equal effect. This
will be the mid-point between the cohesion and separation scale ranges, assuming that the cohesion and separation weights are the same (which they are in this simulation run). In the cases of flock 1 and flock 3, each boid appears, and logically is, an average of 100 units (half of 200) from its flock mates. Flock 2 on the other hand has much closer grouping - average spacing of 25 (half of 50). Flock 4, with its metric method only ‘looking’ for boids within 100 units, has formed no recognisable flock. In fact it is unlikely to do so as its scale range is 500, meaning its ‘balance’ point is at 250 units - which is outside its ‘nearby flock mate’ range. So as soon as any of the flock 4 boids do get close enough to start flocking with each other, the separation behaviour makes them move away from each other. See 6.3 for a closer look at the flock spacing.

Figure 6.2: Simulation 1 - Start of Flocking

This simulation has shown that the flock behaviour we have implemented not only works, but provides flexible options to define how the boids flock. In particular the results we got were what we expected based on the options we used. This is important as being able to control flocking, or indeed any steering behaviours, especially when they are combining to create emergent behaviour, without directly animating the entities, is difficult but extremely useful. We think this shows the flexibility of both our flocking behaviour implementation as well as our InfluenceZone concept which was used to build our flocking behaviour.

6.3 Simulation 2 - Point Following

The second simulation consists of a single boid that is following the PointFollowBehaviour. At the start of the simulation there is a single boid in the centre of the ‘world’ and the
Figure 6.3: Simulation 1 - One Minute in

`PointFollowBehaviour` that is assigned to it consists of eight points in a diamond shape (see 6.4).

Figure 6.4: Simulation 2 - Start

When the simulation starts the boid is drawn to each point in turn, before starting again when it reaches the end. The boid is set with a low turning circle and speed meaning that
it easily navigates between the points. If it had a large turning circle and/or high speed
it would not follow the path very well and indeed it may find it impossible to even reach
each point as it may end up just circling it.

You will notice that all the points are being drawn (see 6.5). Our steering library does
not contain any drawing code but because the behaviours are made up of InfluenceZones,
which are made up of Zones, it is very easy to write some simple drawing code to draw
these Zones. This can greatly help with debugging, especially as trying to debug emergent
behaviour can be very difficult. It is up to the programmer of each Behaviour to allow some
access to any InfluenceZones it contains, and the type of Zones it uses, which we have done
in our Behaviours. If a programmer was using our steering library in their application and
creating new steering behaviours it would be even easier for them to draw debug code as
their behaviours would have access to their drawing system (each of their behaviours could
have a ‘DrawDebug’ method).

Figure 6.5: Simulation 2 - Zones being drawn

Although we have shown that it is quite simple to draw the Zones to provide debug informa-
tion there is certainly a lot more that could be done to help with debugging. Although our
steering library does not, and should not, contain any drawing code it could provide infor-
mation to help to do this. Perhaps the best place for this would be in the BoidBrain class.
This class could contain several debug information methods that would provide information
such as its current axis (forward, up, right) and its latest desired speed and direction. This
sort of information is already available but it could be provided in a ‘ready-to-use’ manner
along with examples in our applications/simulations on how to use draw/present this data
in a useful manner.
6.4 Simulation 3 - Flocking, Point Following and Follow the Leader

The third simulation demonstrates three behaviours, but more importantly shows behaviours working together to create emergent behaviour. The simulation consists of a flock of 31 boids, the first of which is the leader and has a *PointFollowBehaviour* attached to it with points in diamond formation, the same as simulation 2. The remaining 30 boids have two behaviours attached to them - a *FlockingBehaviour* and a *FollowLeaderBehaviour* with the first boid as their leader.

When starting the simulation the leader boid, which is a different colour to the others, begins to follow the path point by point. The other boids follow the leader while still maintaining a flock, as seen in 6.6.

![Figure 6.6: Simulation 3 - Start](image)

As the simulation continues it appears that the flock, as a whole, is following a path when in fact it is only the leader. With the correct weight settings (in this case we found setting all weights to 1.0f except for follow the leader and separation both set to 5.0f) the flock maintains their flock while still following the leader. Setting the *FollowLeaderBehaviour* weight to 5.0f ensured that the flock was heavily drawn to the leader but it caused the flock to become extremely clumped together, as can be seen in 6.7. We found that the separation behaviour needed extra weight to balance their desire to get as close to the back of the leader as they can.

It is clear from this simulation that providing the ability to change the weight/strength of different behaviours (and sub-behaviours in the case of the *SeparationBehaviour* as it is
CHAPTER 6. EVALUATION AND TESTING

45

Figure 6.7: Simulation 3 - Low Separation Weight

contained within a FlockingBehaviour is vital to creating the emergent behaviour that you want. Looking back at simulation 1 it is also clear that other weight/strength properties, such as weight that scales based on distance, is very useful.

6.5 Simulation 4 - Flocking and Collision Avoidance

The final simulation tests one of the more complex behaviours - CollisionAvoidanceBehaviour. The simulation is set up so there is one flock to which you can add more boids and you can create new avoidance meshes (just simple spheres) anywhere in the scene. We created a scene with a flock of twenty boids and four collision meshes as can be seen in 6.8.

At the start the boids actually start inside one of the collision meshes so they immediately try to get out with almost no regard to their flocking behaviour (the weight/strength of the collision avoidance behaviour is far greater than that of their flocking behaviour). Once they have left the avoidance mesh the collision avoidance behaviour no longer has any effect and the separation behaviour of the flocking behaviour takes over and they spread out quickly (see 6.9).

They then begin to flock around the obstacles and appear to flock in a normal manner. They are quite ‘willing’ to fly right alongside the obstacles but as soon as one of them begins to head towards one they move around it and this can have a knock-on effect with the rest of the flock due to the separation and alignment behaviours (6.10).

Although the collision avoidance behaviour, and the emergent behaviour of it combined
with the flocking behaviour, seem to be working well it can be difficult to tell if a boid ever enters any of the avoidance meshes. A useful debugging tool would be able to have hooks in the behaviour so that if a boid ever did enter a `CollisionAvoidanceMesh` an event would be called. This idea could be further extended, and generalised, to being included
as standard in the Zone class. However on the face of it our CollisionAvoidanceBehaviour appears to be working well.

### 6.6 Ease of Use

Our simulations allowed us to test and evaluate the steering library itself but the process of creating these simulations also meant we could evaluate how easy it is to actually use our library. We found that using our library to construct the simulation application was easy and straight forward although this will in part be due to us building the steering library and thus we know it very well.

For all of our simulations we used the same Bird class that inherited from IBoid and thus contained a BoidBrain. To set up the BoidBrain required just one line of code and using it to calculate a new velocity required just a few lines of code:

```
Listing 6.1: BoidBrain Being used by Bird

public override void Update(GameTime gameTime)
{
    if (Enabled)
    {
        // Ask the BoidBrain for a new velocity and rotation amounts
        float rotationAngle = 0.0f;
        Vector3 rotationAxis = Vector3.Zero;
        m_Velocity = m_Brain.CalcInfluence(Position, out rotationAngle, out rotationAxis);
    }
```
CHAPTER 6. EVALUATION AND TESTING

48

if (rotationAngle != 0 & rotationAxis != Vector3.Zero)
{
    PointTo(Position + BoidBrain.ForwardDirection,
             BoidBrain.UpDirection);
}

base.Update(gameTime);

You can see that only a few lines of code are needed to use BoidBrain. You may also notice that we don’t really use, or need, the ‘rotationAngle’ or ‘rotationAxis’ out parameters but is just because the 3D engine we are using has a ‘PointTo’ rotation method so these two parameters certainly should not be removed.

We found that, for each simulation, the process of creating new behaviour objects, boids and flocks was both very easy and clear and only required a few lines of code. We certainly feel that our steering library is easy to use and understand, as all software libraries should be, although it can only truly be fully evaluated by third parties.

6.7 Influence Zones

Objective number five says that we must evaluate how useful our ‘Influencer’ concept was and we discussed this in our Implementation chapter when we were creating our behaviours. In summary we found that the ‘Influencer’ concept was indeed very useful in helping to create new steering behaviours. The actual concept fitted in logically with the concept of steering behaviours and the implementation of this concept proved successful. We initially had planned, and implemented as such, that InfluenceZones would be used directly by the BoidBrain but found this only really allowed for simple behaviours, such as seek. It would have been possible to include greater logic in the construction of Influencers to help create more complex behaviours however this would have massively reduced the re-usability of Influencers. Instead we created the Behaviour class instead that would be for creating specific steering behaviours and would use InfluenceZones to help. This ‘help’ would consist of code re-use and faster debugging but we feel also help steering behaviour novices to construct, and understand, steering behaviours. We certainly found this as we had little experience creating steering behaviours and in the early prototyping found it difficult to create steering behaviours from scratch. InfluenceZones proved a base, or building-block, to work from so that you are not working from ‘nothing’. Because of this we feel that our steering library would be of greatest use to steering behaviour novices, such as when the focus of an application/research project isn’t on steering behaviours but
requires it, for example the study of animal flocking.

6.8 Summary

In this chapter we have looked at the simulations we created to test and evaluate our steering library. We have seen that our library works and is easy to use. Further, we have evaluated our experimental ‘Influencer’ concept and seen that it has indeed proved useful in creating steering behaviours and that it, along with our library as a whole, would be very useful when creating steering applications.

In the next chapter, Conclusions, we will take a final look at our project as whole - where we started, where we went and where we ended up.
Chapter 7

Conclusions

7.1 Introduction

In this project we have reviewed past research into flocking, planned and designed a steering library built on the concept of InfluenceZones, implemented said steering library and finally used our steering library to create some simulations to test and evaluate our system. We will now take a step back and look at our project and system as a whole and see both what we actually have and how we got to where we are.

7.2 Review of Achievements

We started off this project with five objectives. The first objective was to design and implement a steering library that was stand alone, generic and extensible. Our library has been built so that it depends on no other libraries, such as a 3D engine, and only contains ‘pure logic’ meaning that it is very much stand alone and could be used with any other application. Obviously it is limited to being used with other C#.NET XNA applications but it has been designed and built in such a way as to minimise the use of XNA. Indeed only a few built-in XNA types and mathematical methods have been used meaning that it would be easy to port to any other language and/or API, such as C++ and openGL or DirectX.

As we saw in our Literature Survey, steering behaviours are used in a wide range of different areas and the second part of this objective was that our library would be generic. We believe we have met this requirement. Just looking at the structure of our code you can see that it is only concerned with steering an entity in a certain direction and there is no specific code to deal with ‘crowds’ or other areas. However although the behaviour system and InfluenceZones are very generic we feel that the more ‘management’ classes, such as the BoidBrain and Flock classes, are perhaps not generic enough. Thankfully because these are just ‘management’ classes there is no requirement for applications using the library to
use them and could instead create their own management classes that use the Behaviour and InfluenceZone concepts.

The third and final part of this objective was that our library must be extensible and this meant both being able to easily add new parts to the library, and to extend the library when being used in an application. Our simple design and the ‘layers’ and wrapper methods in our library certainly means that it is easy to extend and modify the library - we found this while developing it. The second part of this, being extended by applications, is of vital importance and ties back to the ‘generic’ objective. Because our library was meant to be generic, if it was not extensible its usefulness would be very limited. The different parts of the library, such as Zones, Influencers and Behaviours, have all been built as abstract classes meaning that anyone can create new sub-classes of these and of pre-existing ones (such as SphereZone). This allows users to create application specific sub-classes, such as new Behaviours, that they can program to do whatever they want and still be useable with the rest of the steering library. The fact that our library is built on ‘layers’ (Zones and Influencers making up InfluenceZones that can create new Behaviours) helps with this extensibility in that applications can extend our library in different parts, such as new Zones, and then use them to extend other parts such as Behaviours. This also greatly helps with code re-use and testing. Indeed a new project using our library could start off by extending the Zone and Influencer by creating new sub-classes and then use these to create their new application specific Behaviours. They could even then use their new Zones and Influencers in other projects, if they are built in a generic way as ours are, saving even more time and effort.

Our second main objective was to design and implement the concept of ‘Influencers’. We did this with the Zone, Influence and InfluenceZone classes. By splitting it like this, with Zone and Influence as abstract classes, with concrete sub-classes forming InfluenceZones, it allowed extensibility, flexibility and code re-use. It also meant that the Zone and Influence parts could be used on their own in Behaviours if the other one wasn’t needed.

The third objective was to implement several steering behaviours in our library using the ‘Influencer’ concept in order to test our library, test and evaluate the ‘Influencer’ concept and to provide useful behaviours, or at least behaviour samples, for general use. We created nine steering behaviours and one generic behaviour that is made up of a list of, settable, InfluenceZones. We discussed the use of the ‘Influencer’ concept in creating steering behaviours in the Implementation chapter. We found that the ‘Influencer’ concept was very helpful in creating steering behaviours and that it fits naturally with steering behaviours in general.

The fourth and fifth objectives were to use our library to create some simulations and then to use these to test and evaluate our library as a whole, including how easy it is to use. Please see the previous chapter for details on this.

We believe we have met the objectives for this project and found that our ‘Influencer’ concept does indeed help create new steering behaviours and that our steering library as a whole is useful tool for creating applications that use steering behaviours as a key
CHAPTER 7. CONCLUSIONS

component. That is not to say there are no improvements to be made - indeed there are many - and we will discuss those in the sections below.

7.3 Critique of Process

The initial stages of this project were spent, not on researching steering behaviours, but researching flocking. It was during this early research that we decided to build a generic, extensible flocking library (hence the name “Flocker” for the library) rather than a library that was specific to controlling flocks to do certain things. We then moved towards an even more generic system by making it a general steering library rather than just a flocking library. This change meant we widened our research scope to include other steering behaviours and looked at what fields used steering behaviours. This work included reading journals, proceedings from conferences, websites and articles in books, and the authors of these works ranged from computer graphics academics to animal modelling expects to computer game AI developers. We feel that this part of the project, arguably the most important part, was our weakest. We were slow to begin in-depth research and our slight change of projects at the start also slowed us down. We would have preferred to finish the research part of this project about two weeks before we did which would have given us greater time to experiment with and improve our steering library.

After the research phase, we did a small prototype project to help us gain some understanding of steering behaviours and how they are implemented. This work was in 2D and although it meant we gained some experience the time probably would have been better spent doing more in-depth research and starting the project proper.

When we did start the implementation our understanding of what we were trying to built was actually very limited. We spent a while developing the framework of the library and system as a whole and this was probably the easiest part of the project. Moving on to the creation of the ‘Influence Zone’ concept we spent a while trying different implementations of it and we are very glad we spent a long time on this. The time we had allowed us to scrap parts of it and even start from scratch again and we feel this resulted in a much better realisation of the ‘Influence Zone’ concept. With this done we could then spend time testing it and building behaviours confident that the underlying framework and ‘Influence Zone’ concept worked well.

We spent some time developing different steering behaviours, perhaps too much time, and then also creating some simulations to test and evaluation our library. Although we developed several different steering behaviours it may have been better, and more interesting, to have created fewer behaviours and instead created some more complex ones or spent the time refining them. However we feel that the simulations we created were done well in that as well as helping us test and evaluate our system, they also provide a good sample of how to use our library.
7.4 Possible Implementation Improvements

There are certainly a number of implementation improvements that could be made to our steering library and the nature of the experimental ‘Influencer’ concept means that it would be better thought of as a prototype.

The BoidBrain class is an important class but it is actually very simple and only offers a few ways to customise it, such as setting maximum and minimum speeds for the boid. Indeed this class is not very generic or extensible unlike the rest of the library. The main method, ‘CalcInfluence’ is quite long and could certainly do with being split up so that users who inherit from the BoidBrain class can override different parts of it as they choose. We propose that the BoidBrain class be more like the InfluenceZone class in that it should be made up of different parts - concrete classes that inherit from generic and extensible abstract classes. In the same way that new Zones and Influencers can be created (and re-used) so it should be for different parts of the BoidBrain class. For instance, at the moment the BoidBrain uses a very simple piece of code to calculate its new velocity - it multiplies its desired direction and speeds together and clamps them by the minimum and maximum speeds - and a far better solution would be to have a BoidEngine class. This would be an abstract class and would allow applications to create their own specific engine code that could be as simple as ours or represent a more realistic engine, such as having a ‘thrust’ component and calculating acceleration.

In our research we saw that some flocking implementations used ‘bands’ to decide whether a nearby boid would cause cohesion, separation or alignment (Kunz and Hemelrijk, 2003). This concept of ‘bands’ to decide influence has proved useful in steering behaviours and it could be something that could be built into our library. You could have a collection of ‘stacked’ InfluenceZones, perhaps called an ‘InfluenceZoneGroup’, where the influence calculated and returned to the behaviour would be the influence from the lowest stacked InfluenceZone that the entity was in. For instance, you could have a stack of SphereZones each one slightly larger than the one ‘beneath’ it and all at the same position in space. As well as being useful for behaviours, like flocking, that consist of multiple behaviours that need to ‘balance’ it could also be used as almost an AI/A-life control system. As an example, you could use an ‘InfluenceZoneGroup’ to help in a crowd simulation by placing the Zones not necessarily stacked on each but placed around the world. There would be some Zones that cover large areas, including one covering the entire world, and some that contain smaller Zones. The larger Zones would cause ‘normal’ crowd steering behaviour but when an entity moved into a smaller Zone, say one that covered a park, it would change to follow the more specific Influencer. In this example the entity could go from walking briskly along the street, then walk into the park where the Influencer here causes the entity to walk in a more casual manner. This is all possible at the moment with the flexibility of Behaviours but part of the point of the ‘Influencer’ concept is to help create new behaviours as easily as possible. Another example would be stacking some SphereZones on top of each other, the inner and outer ones influencing an entity to move out/in while a middle InfluenceZone has no influence. This would create the effect of orbiting.
CHAPTER 7. CONCLUSIONS

7.5 What We Learnt

This project has, as one would expect, taught us a great deal about steering behaviours. We have learnt that steering behaviours are used in a wide variety of different fields including animal modelling, crowd simulation, films and games - basically anywhere that requires an entity in a computer simulation to ‘move’ without being explicitly animated. This project has also given us the opportunity to research and implement specific steering behaviours, including the heavily used ‘flocking’ behaviour as well as letting us experience how to create (and control) emergent behaviour.

Undertaking this project has also meant the development of academic writing skills, research skills, the ability to learn from, analyse and critique academic papers and research and presentation skills.

Although we have developed much larger software in the past, the implementation of this project has certainly given us greater experience in software development and programming especially in using the natural generic and extensible features of object-oriented programming languages. We also gained valuable experience in the difficult field of 3D mathematics as our library and simulations are ‘3D in nature’ and not 2D.

7.6 Possible Future Work

As we stated above, we believe that our steering library is more like a prototype with regards to the experimental ‘Influencer’ concept. Certainly our library could be greatly enhanced with the implementation suggested above and would be even more enhanced if other steering concepts were included in our library to increase flexibility and its usefulness. Perhaps there could also be the development of new sections in the library that are specific for certain steering applications - for instance a ‘crowds’ section - that could be added to and built up over time.

As we have discussed previously, steering behaviours have no ‘intelligence’, and just form the middle layer of the ‘action-¿steering-¿locomotion’ concept (Reynolds, 1999). It would be interesting if libraries in the same spirit as ours - generic, extensible and layers that can be built up and combined - were to be constructed, perhaps even designed to work alongside our library (although not necessarily forced). In particular we would be very interested to see this sort of work done on the ‘action’ (AI) layer.

This actually brings up an interesting point of how our library interacts with the two layers beside it. At the moment it has no interaction. It is up to the application to provide it and our library has no knowledge of them or how they might work. In our simulations we have next to no AI and the locomotion part is just the moving and rotating of simple 3D models. Perhaps something else that could be looked at is enabling our steering library to more easily ‘hook’ into the action and locomotion layer, particularly the action/AI layer. For instance, a simple improvement would be to add event hooking to the Zones so that
when a boid entered the Zone methods hooked to the event would be fired and these event methods could then decide that because the boid is in that area, it should now do something else. These could also be hooked into Behaviours although this is easily possible now with custom application built Behaviours.
Bibliography


Appendix A

Code

In this appendix is a listing of some of the classes from the steering behaviour library “Flocker”. The classes included are the primary classes (BoidBrain, InfluenceZone, Zone, Influencer, Behaviour) and a concrete example of a Zone (SphereZone), an Influencer (AttractorInfluencer) and a Behaviour (SeparationBehaviour). For the complete source code of the library, the simulation application and the 3D engine (“BOB XNA Engine”) used in the project, please see the attached CD.
A.1 File: BoidBrain.cs

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Flocker.Behaviours;
using Flocker.Influencers;

namespace Flocker
{

    public interface IBoid
    {
        BoidBrain BoidBrain { get; }
    }

    /// <summary>
    /// Represents the steering brain of an entity. Each entity should inherit from the IBoid interface
    /// and contain its own BoidBrain. Steering Behaviours should be added to it or its flock. On each 'update' you
    /// should call 'CalcInfluence' to calculate the newly desired velocity to go.
    /// </summary>

    public class BoidBrain
    {
        private Vector3 m_Position;
        private Vector3 m_Velocity;

        // Movement/3D Space
        private Vector3 m_FowardDirection;
        private Vector3 m_UpDirection;
        private Vector3 m_RightDirection;
        private float m_MinSpeed = 0.0f;
        private float m_MaxSpeed = 5.0f;
        private float m_PreferredSpeed = 1.0f;
        private float m_MaxTurningAngle = MathHelper.ToRadians(1.0f);

        // Influence
        private Flock m_Flock;

        // BoundingSphere
        private BoundingSphere m_BoundingSphere;

        // Dictionary of Behaviours
        private Dictionary<string, Behaviour> m_Behaviours;

        #region Properties
        public Vector3 Position
        {
            get { return m_Position; }
            set
            {
                m_Position = value;
                m_BoundingSphere.Center = value;
            }
        }
        public Vector3 Velocity
        {
            get { return m_Velocity; }
        }
        public float MinSpeed
        {
            get { return m_MinSpeed; }
            set { m_MinSpeed = value; }
        }
        public float MaxSpeed
        {
            get { return m_MaxSpeed; }
            set { m_MaxSpeed = value; }
        }
        public float PreferredSpeed
        {
            get { return m_PreferredSpeed; }
            set { m_PreferredSpeed = value; }
        }
        // }
    }

}
public BoidBrain(Flock flock, Vector3 position, float boundingRadius, Vector3 forwardDirection, Vector3 upDirection, Vector3 rightDirection, float minSpeed, float maxSpeed, float preferredSpeed, float maxTurningAngle)
{
    m_Flock = flock;
    m_Position = position;
    m_BoundingSphere = new BoundingSphere(position, boundingRadius);
    m_ForwardDirection = forwardDirection;
    m_UpDirection = upDirection;
    m_RightDirection = rightDirection;
    m_MinSpeed = minSpeed;
    m_MaxSpeed = maxSpeed;
    m_PreferredSpeed = preferredSpeed;
    m_MaxTurningAngle = maxTurningAngle;

    m_Behaviours = new Dictionary<string, Behaviour>();
}

#region Behaviour Management
/// <summary>
/// Adds the given Behaviour to this boid’s list of Behaviours, using the Behaviour’s name as its key
/// </summary>
public void AddBehaviour(Behaviour behaviour)
{
    m_Behaviours.Add(behaviour.Name, behaviour);
}
#endregion
public bool RemoveBehaviour(string name) {
    return m_Behaviours.Remove(name);
}

public Behaviour GetBehaviour(string name) {
    Behaviour behaviour = null;
    m_Behaviours.TryGetValue(name, out behaviour);
    return behaviour;
}

public virtual Vector3 CalcInfluence(Vector3 position, out float rotationAngle, out Vector3 rotationAxis) {
    // Update position and bounding sphere
    m_Position = position;
    m_BoundingSphere.Center = position;

    // Influences (desired direction and desired speed)
    Vector3 desiredDirection = Vector3.Zero;
    float desiredSpeed = 0.0f;
    float speedWeights = 0.0f;
    bool noDesiredSpeed = true;

    // Iterate through all the behaviours and add up their total 'influences'
    foreach (Behaviour behaviour in m_Behaviours.Values) {
        if (behaviour.DirectionEnabled) {
            desiredDirection += behaviour.CalcWeightedDirection(this);
        }
        if (behaviour.SpeedEnabled) {
            desiredSpeed += behaviour.CalcWeightedSpeed(this);
            speedWeights += behaviour.CalcSpeedWeight(this);
            noDesiredSpeed = false;
        }
    }

    // Do the same for the flock behaviours
    foreach (Behaviour behaviour in m_Flock.Behaviours.Values) {
        if (behaviour.DirectionEnabled) {
            desiredDirection += behaviour.CalcWeightedDirection(this);
        }
    }

    // the boid should use to rotate itself
    rotationAngle = Math.Max(0.0f, Math.Min(180.0f, angle) - (m_Angle - angle));
    rotationAxis = desiredDirection;

    // the boid wants to travel at, clamped by its min and max speeds and max turning angle. Rotates its axis by the amount it wants to (and can) rotate by and passes this information out
    Vector3 offset = rotationAxis * Vector3.Cross(rotationAxis, Vector3.Cross(rotationAxis, desiredDirection)).Length();
}

/// <param name="name">The name of the Behaviour to remove</param>
/// <returns>True if successfully found and removed, false otherwise</returns>
public void RemoveBehaviour(string name) {
    m_Behaviours.Remove(name);
}
if (behaviour.SpeedEnabled)
{
    desiredSpeed +=
        behaviour.CalcWeightedSpeed(this);
    speedWeights +=
        behaviour.CalcSpeedWeight(this);
    noDesiredSpeed = false;
}

// Direction
float angle = 0.0f;
Vector3 axis = Vector3.Zero;
if (desiredDirection != Vector3.Zero)
{
    // Calculate the angle between current forward direction and the direction we 'want' to go
    // Angle
    angle =
        MathsHelper.AngleBetween(m_FORWARD_DIRECTION, desiredDirection);
    // Clip the angle by our max turning angle
    if (angle > m_MAX_TURNING_ANGLE)
    {
        angle = m_MAX_TURNING_ANGLE;
    }
    // Axis
    axis =
        MathsHelper.RotationAxis(m_FORWARD_DIRECTION, desiredDirection);
    // 180 Degrees case
    if (axis == Vector3.Zero && angle ==
        MathHelper.ToRadians(180))
    {
        // Axis should be 90 Degrees to the vectors
        axis = m_UP_DIRECTION;
    }
    // Rotate our directions
    RotateDirections(axis, angle);
}

// Speed
if (!noDesiredSpeed && speedWeights != 0.0f)
{
    desiredSpeed = desiredSpeed / speedWeights;
}
else
{
    desiredSpeed = m_PREFERRED_SPEED;
}

// Calculate the new velocity (clamping it by min/max speeds)
Vector3 velocity =
    MathsHelper.Clamp(m_FORWARD_DIRECTION *
        desiredSpeed, m_MIN_SPEED, m_MAX_SPEED);

// Return the rotated angle and new velocity
rotationAngle = angle;
rotationAxis = axis;
return m_VELOCITY;

/// <summary>
/// Rotates the BoidBrain's forward, up and right directions
/// </summary>
/// <param name="axis">The axis to rotate on</param>
/// <param name="angle">The angle, in radians, to rotate by</param>
public void RotateDirections(Vector3 axis, float angle)
{
    Quaternion rot =
        Quaternion.Normalize(Quaternion.CreateFromAxisAngle(axis, angle));
    m_FORWARD_DIRECTION =
        Vector3.Transform(m_FORWARD_DIRECTION, rot);
    m_FORWARD_DIRECTION.Normalize();
m_UpDirection = Vector3.Transform(m_UpDirection, rot);
m_UpDirection.Normalize();

m_RightDirection = Vector3.Transform(m_RightDirection, rot);
m_RightDirection.Normalize();

A.2 File: InfluenceZone.cs

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Flocker.Influencers;
using Flocker.Zones;

namespace Flocker
{
    /// InfluenceZone represents an area in space (the Zone) that has some sort of steering influence (the Influence).
    /// Has methods to check whether a boid is in its Zone and methods to calculate any steering influence on said boid.
    ///
    public class InfluenceZone
    {
        private Influencer m_Influencer;
        private Zone m_Zone;
        private bool m_InverseZone = false;
        private bool m_DirectionEnabled = true;
        private bool m_SpeedEnabled = true;

        #region Properties
        public Influencer Influencer
        {
            get { return m_Influencer; }
        }
        public Zone Zone
        {
            get { return m_Zone; }
        }
        public bool InverseZone
        {
            get { return m_InverseZone; }
            set { m_InverseZone = value; }
        }
        public bool DirectionEnabled
        {
            get { return m_DirectionEnabled; }
            set { m_DirectionEnabled = value; }
        }
        public bool SpeedEnabled
        {
            get { return m_SpeedEnabled; }
            set { m_SpeedEnabled = value; }
        }
        #endregion

        /// Creates a new InfluenceZone object consisting of the given Influencer and Zone.
        /// Both Direction and Speed influencing are enabled
        ///
        /// <param name="influencer">The Influencer this InfluenceZone should use</param>
        /// <param name="zone">The Zone this InfluenceZone should use</param>
        public InfluenceZone(Influencer influencer, Zone zone)
        {
            m_Influencer = influencer;
            m_Zone = zone;
        }
    }
}
/// enabled status must be specified
/// </summary>
/// <param name="influencer">The Influencer this InfluenceZone should use</param>
/// <param name="zone">The Zone this InfluenceZone should use</param>
/// <param name="directionEnabled">Whether directional influence is enabled</param>
/// <param name="speedEnabled">Whether speed influence is enabled</param>
public InfluenceZone (Influencer influencer, Zone zone, bool directionEnabled, bool speedEnabled)
{
    m_Influencer = influencer;
    m_Zone = zone;
    DirectionEnabled = directionEnabled;
    SpeedEnabled = speedEnabled;
}

/// Whether the given bounding sphere intersections with this InfluenceZone's Zone. If the 'InverseZone'
/// flag is set to true then the return value will be opposite of normal
/// </summary>
/// <param name="sphere">The BoundingSphere to check against</param>
/// <returns>True if the given sphere intersects with the Zone, false otherwise</returns>
public virtual bool Intersects (BoundingSphere sphere)
{
    if (m_InverseZone)
    {
        return !m_Zone.Intersects (sphere);
    }
    else
    {
        return m_Zone.Intersects (sphere);
    }
}

/// Calculates the directional influence
/// </summary>
/// <param name="boid">The boid asking for the 'desire'</param>
/// <returns>The calculated directional desire</returns>
public virtual Vector3 CalcDirection (BoidBrain boid)
{
    return m_Influencer.CalcDirection (m_Zone, boid);
}

/// Calculates the directional influence weight
/// </summary>
/// <param name="boid">The boid asking for the 'desire' weight</param>
/// <returns>The calculated directional desire weight</returns>
public virtual float CalcDirectionWeight (BoidBrain boid)
{
    return m_Influencer.CalcDirectionWeight (m_Zone, boid);
}

/// Calculates the directional influence weighted
/// </summary>
/// <param name="boid">The boid asking for the 'desire' weighted</param>
/// <returns>The calculated directional desire weighted</returns>
public Vector3 CalcWeightedDirection (BoidBrain boid)
{
    return m_Influencer.CalcWeightedDirection (m_Zone, boid);
}

/// Calculates the speed influence
/// </summary>
/// <returns>The calculated speed influence</returns>
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace Flocker.Zones
{
    /// Represents an 'area' in 3D space.
    /// Typically used to form part of an 'InfluenceZone'
    public abstract class Zone
    {
        #region Properties
        public abstract Vector3 Position
        {
            get;
            set;
        }
        #endregion

        /// Checks whether the given BoundingBox intersects with this Zone
        /// <param name="box">The BoundingBox to check intersection with</param>
        /// <returns>True if there they do intersect, false otherwise</returns>
        public abstract bool Intersects(BoundingBox box);

        /// Checks whether the given BoundingSphere intersects with this Zone
        /// <param name="sphere">The BoundingSphere to check intersection with</param>
        /// <returns>True if there they do intersect, false otherwise</returns>
        public abstract bool Intersects(BoundingSphere sphere);
    }
}

A.3 File: Zone.cs
A.4 File: SphereZone.cs

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace Flocker.Zones {
    public class SphereZone : Zone {
        private BoundingSphere m_BoundingSphere;

        #region Properties
        public override Vector3 Position {
            get { return m_BoundingSphere.Center; }
            set { m_BoundingSphere.Center = value; }
        }
        public float Radius {
            get { return m_BoundingSphere.Radius; }
            set { m_BoundingSphere.Radius = value; }
        }
        #endregion

        public SphereZone(Vector3 center, float radius) {
            m_BoundingSphere = new BoundingSphere(center, radius);
        }
    }
}
/// Checks whether the given BoundingBox intersects with this Zone
public override bool Intersects(BoundingBox box)
{
    return m_BoundingSphere.Intersects(box);
}

/// Checks whether the given BoundingSphere intersects with this Zone
public override bool Intersects(BoundingSphere sphere)
{
    return m_BoundingSphere.Intersects(sphere);
}

/// Checks whether the given BoundingBox is fully contained within this Zone
public override bool Contains(BoundingBox box)
{
    return (m_BoundingSphere.Contains(box) == ContainmentType.Contains);
}

/// Checks whether the given BoundingSphere is fully contained within this Zone
public override bool Contains(BoundingSphere sphere)
{
    return (m_BoundingSphere.Contains(sphere) == ContainmentType.Contains);
}

/// Scales the weight based on the distance of the boid from the center of the sphere. As the boid moves
/// towards the center of the sphere, the percentage tends towards zero, and at the very edge of the sphere
/// (and beyond) it is at 100% (1.0f)
public override float CalcScaledWeight(BoidBrain boid, float minClamp, float maxClamp, bool inverse)
{
    float str = (Vector3.Distance(Position, boid.Position) / Radius);
    if (inverse)
    {
        str = 1 - str;
    }
    str = MathHelper.Clamp(str, minClamp, maxClamp);
    return str;
A.5 File: Influencer.cs

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Flocker.Zones;

namespace Flocker.Influencers
{
    /// Represents some sort of directional and/or speed influence/desire.
    /// Typically used to form part of an 'InfluenceZone'.
    public abstract class Influencer
    {
        /// Returns zero. Override this with Influencer specific code to calculate the directional influence.
        /// <param name="zone">The Zone this influencer is 'paired' with</param>
        /// <param name="boid">The boid asking for the 'desire'</param>
        /// <returns>The calculated directional desire weight</returns>
        public virtual float CalcDirectionWeight(Zone zone, BoidBrain boid)
        {
            return 0.0f;
        }

        /// Returns zero. Override this with Influencer specific code to calculate the speed influence.
        /// <param name="zone">The Zone this influencer is 'paired' with</param>
        /// <param name="boid">The boid asking for the 'desire'</param>
        /// <returns>The calculated speed desire</returns>
        public virtual float CalcSpeed(Zone zone, BoidBrain boid)
        {
            return 0.0f;
        }
    }
}
A.6 File: AttractorInfluencer.cs

```csharp
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;
using Flocker.Zones;

namespace Flocker.Influencers {
    /// Represents an Influencer whose influence is always towards the centre of the 'paired' (given) Zone relative to the given boid's position. Has no speed influencer
    public class AttractorInfluencer : Influencer {
        private float m_Strength;

        #region Properties
        public float Strength {
            get { return m_Strength; }
            set { m_Strength = value; }
        }
        #endregion

        /// Creates a new AttractorInfluencer object
        public AttractorInfluencer(float strength) {
            Strength = strength;
        }

        /// Calculates the directional influencer - direction towards the centre of the given Zone relative to the given boid's position
        public virtual float CalcSpeedWeight(Zone zone, BoidBrain boid) {
            return 0.0f;
        }

        public virtual float CalcWeightedSpeed(Zone zone, BoidBrain boid) {
            return 0.0f;
        }
    }
}
```
public override Vector3 CalcDirection (Zone zone, BoidBrain boid)
{
    Vector3 direction = zone.Position - boid.Position;
    if (direction != Vector3.Zero)
    { 
        direction.Normalize();
    }
    return direction;
}

public override float CalcDirectionWeight (Zone zone, BoidBrain boid)
{
    return m_Strength;
}

public override Vector3 CalcWeightedDirection (Zone zone, BoidBrain boid)
{
    return CalcDirection (zone, boid) * 
             CalcDirectionWeight (zone, boid);
}

A.7 File: Behaviour.cs

using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.Xna.Framework;

namespace Flocker.Behaviours
{
    /// Represents a steering Behaviour. Abstract class that must be inherited to instantiate
    public abstract class Behaviour
    {
        private string m_Name;
        private bool m_Enabled = true;
        private bool m_DirectionEnabled = true;
        private bool m_SpeedEnabled = true;

        #region Properties
        public string Name
        {
            get { return m_Name; } }
        }
        public bool Enabled
        {
            get { return m_Enabled; } 
            set { m_Enabled = value; }
            }
        }
/// <summary>
/// Whether the direction AND enabled flags are true. Only sets the direction enabled flag though
/// </summary>
public bool DirectionEnabled {
    get { return m_DirectionEnabled && m_Enabled; }
    set { m_DirectionEnabled = value; }
}

/// <summary>
/// Whether the speed AND enabled flags are true. Only sets the speed enabled flag though
/// </summary>
public bool SpeedEnabled {
    get { return m_SpeedEnabled && m_Enabled; }
    set { m_SpeedEnabled = value; }
}

/// <summary>
/// Creates a new Behaviour object
/// </summary>
/// <param name="name">
/// The name of the behaviour (used for look-up)
/// </param>
public Behaviour(string name) {
    m_Name = name;
}

/// <summary>
/// Returns zero. Override this with Behaviour specific code to calculate the directional influence
/// </summary>
/// <param name="boid">
/// The boid asking for the 'desire'
/// </param>
/// <returns>The calculated directional desire weighted</returns>
public virtual Vector3 CalcDirectionWeight(BoidBrain boid) {
    return 0.0f;
}

/// <summary>
/// Returns zero. Override this with Behaviour specific code to calculate the directional influence weighted
/// </summary>
/// <param name="boid">
/// The boid asking for the 'desire'
/// </param>
/// <returns>The calculated directional desire weighted</returns>
public virtual float CalcSpeed(BoidBrain boid) {
    return 0.0f;
}
namespace Flocker.Behaviours
{
    /// <summary>
    /// Represents a separation steering behaviour. Influen
ces boids to move away their nearby flock mates. If boids
    /// are ever found to be directly on top of each other, a random directional influence is calculated
    /// </summary>
    public class SeparationBehaviour : Behaviour
    {
        private FlockmateFinder m_FlockmateFinder;
        private int m_NeighbourRange; // Metric
        private int m_NeighbourCount; // Topological
        private float m_DirectionWeight;
        private ScaleWeight m_ScaleWeight;
        private InfluenceZone m_InfZone;

        private static Random s_Rand;

        #region Properties
        public FlockmateFinder FlockmateFinder
        {
            get { return m_FlockmateFinder; }
            set { m_FlockmateFinder = value; }
        }
        public int NeighbourRange
        {
            get { return m_NeighbourRange; }
            set { m_NeighbourRange = value; }
        }
        public int NeighbourCount
        {
            get { return m_NeighbourCount; }
            set { m_NeighbourCount = value; }
        }
        public float DirectionWeight
        {
            get { return m_DirectionWeight; }
            set { m_DirectionWeight = value; }
        }
        public ScaleWeight ScaleWeight
        {
            get { return m_ScaleWeight; }
            set { m_ScaleWeight = value; }
        }
    }
}
get { return m_ScaleWeight; }
set { m_ScaleWeight = value; }
}
public float WeightScaleRange
{
    get { return ((SphereZone)m_InfZone.Zone).Radius; }
    set { ((SphereZone)m_InfZone.Zone).Radius = value; }
}

m_InfZone = new InfluenceZone(new
    BasicInfluencer(Vector3.Zero, 1.0f), new
    SphereZone(Vector3.Zero, 1.0f));

SpeedEnabled = false;
if (s_Rand == null)
{
    s_Rand = new Random();
}

/// <summary>
/// Creates a new SeparationBehaviour object
/// </summary>
/// <param name="name">The name of the behaviour</param>
/// <param name="flockmateFinder">Whether to use a metric or topological technique to find nearby flock mates</param>
/// <param name="neighbourRange">The distance to consider flock mates as 'nearby' when using the metric technique</param>
/// <param name="neighbourCount">The number of neighbours to consider as 'nearby' when using the topological technique</param>
/// <param name="directionWeight">The directional influencer weight of this behaviour</param>
public SeparationBehaviour(string name, FlockmateFinder flockmateFinder, int neighbourRange, int neighbourCount, float directionWeight)
: base(name)
{
    m_FlockmateFinder = flockmateFinder;
    m_NeighbourRange = neighbourRange;
    m_NeighbourCount = neighbourCount;
    m_DirectionWeight = directionWeight;
    m_ScaleWeight = ScaleWeight.DontScale;
}
public SeparationBehaviour(string name, FlockmateFinder flockmateFinder, int neighbourRange, int neighbourCount, float directionWeight, ScaleWeight scaleWeight, float weightScaleRange) : base(name) {
    m_FlockmateFinder = flockmateFinder;
    m_NeighbourRange = neighbourRange;
    m_NeighbourCount = neighbourCount;
    m_DirectionWeight = directionWeight;
    m_ScaleWeight = scaleWeight;
    m_InfZone = new InfluenceZone(new BasicInfluencer(Vector3.ZERO, 1.0f), new SphereZone(Vector3.ZERO, weightScaleRange));
    SpeedEnabled = false;
    if (s_Rand == null) {
        s_Rand = new Random();
    }
}

/// <summary>
/// Re-calculates the influence of this behaviour. Influence is away from the
/// positions of the nearby flock mates. A random directional influence is used
/// if a boid is ever at the exact same position of a flock mate
/// </summary>
/// <param name="flockmates">The 'nearby' flock mates</param>
/// <param name="boid">The boid asking for the 'desire'</param>
private void ReCalcInfluenceZone(List<BoidBrain> flockmates, BoidBrain boid) {
    // Calc a repulsion force from each nearby flockmate and combine them
    Vector3 totalRepulsion = Vector3.ZERO;
    Vector3 repulsion;
    foreach (BoidBrain flockmate in flockmates) {
        repulsion = boid.Position - flockmate.Position;
        if (repulsion != Vector3.ZERO) {
            repulsion.Normalize();
        } else {
            // The boids have same position (on top of each other) so generate a random direction
            repulsion = new Vector3(s_Rand.Next(-100, 100), s_Rand.Next(-100, 100), s_Rand.Next(-100, 100));
            if (repulsion != Vector3.ZERO) {
                repulsion.Normalize();
            }
        }
    }

    // Set the position of the zone to the flockmates position and calc stregth of repulsion
    m_InfZone.Zone.Position = flockmate.Position;
    switch (m_ScaleWeight) {
        case ScaleWeight.DontScale:
            totalRepulsion += repulsion;
            break;
        case ScaleWeight.Scale:
            totalRepulsion += repulsion * m_InfZone.Zone.CalcScaledWeight(boid, 0.0f, 1.0f, false);
            break;
        case ScaleWeight.InverseScale:
            totalRepulsion += repulsion * m_InfZone.Zone.CalcScaledWeight(boid, 0.0f, 1.0f, true);
            break;
        default:
            break;
    }

    // ...
/// With all repulsions combined to create
/// final direction, normalize and set it as
/// influence
if (totalRepulsion != Vector3.Zero)
{
    totalRepulsion.Normalize();
}
((BasicInfluencer)m_InfZone.Influencer).Direction
    = totalRepulsion;

/// Finds 'nearby' flock mates of the given boid
/// (using metric or topological technique) and
/// calculates
/// a repulsion desire away from each of them
/// (weight)
/// <returns>The calculated directional desire weight</returns>
public override float CalcDirectionWeight(BoidBrain boid)
{
    List<BoidBrain> flockmates;
    if (m_FlockmateFinder ==
        FlockmateFinder.Metric)
    {
        flockmates =
            SteeringHelper.GetCloseFlockmates(boid,
                boid.Flock, m_NeighbourRange);
    }
    else // Topological
    {
        flockmates =
            SteeringHelper.GetClosestFlockmates(boid,
                boid.Flock, m_NeighbourCount);
    }
    ReCalcInfluenceZone(flockmates, boid);
    return m_InfZone.CalcDirectionWeight(boid) *
        m_DirectionWeight;
}

/// Finds 'nearby' flock mates of the given boid
/// (using metric or topological technique) and
/// calculates
/// a repulsion desire away from each of them
/// <returns>The calculated directional desire
/// weight</returns>
public override Vector3 CalcDirection(BoidBrain boid)
{
    List<BoidBrain> flockmates;
    if (m_FlockmateFinder ==
        FlockmateFinder.Metric)
    {
        flockmates =
            SteeringHelper.GetCloseFlockmates(boid,
                boid.Flock, m_NeighbourRange);
    }
    else // Topological
    {
        flockmates =
            SteeringHelper.GetClosestFlockmates(boid,
                boid.Flock, m_NeighbourCount);
    }
    ReCalcInfluenceZone(flockmates, boid);
    return m_InfZone.CalcDirection(boid);
```csharp
public override Vector3 CalcWeightedDirection(BoidBrain boid)
{
    List<BoidBrain> flockmates;
    if (m_FlockmateFinder == FlockmateFinder.Metric)
    {
        flockmates = SteeringHelper.GetCloseFlockmates(boid, boid.Flock, m_NeighbourRange);
    }
    else // Topological
    {
        flockmates = SteeringHelper.GetClosestFlockmates(boid, boid.Flock, m_NeighbourCount);
    }
    ReCalcInfluenceZone(flockmates, boid);
    return m_InfZone.CalcWeightedDirection(boid) * m_DirectionWeight;
}

/// <summary>
/// Finds 'nearby' flock mates of the given boid (using metric or topological technique) and calculates
/// a repulsion desire away from each of them (weight)
/// <param name="flockmates">The 'nearby' flock mates</param>
/// <param name="boid">The boid asking for the 'desire'</param>
/// <returns>The calculated directional desire weight</returns>
/// </summary>
public float CalcDirectionWeight(List<BoidBrain> flockmates, BoidBrain boid)
{
    ReCalcInfluenceZone(flockmates, boid);
    return m_InfZone.CalcDirectionWeight(boid) * m_DirectionWeight;
}

/// <summary>
/// Finds 'nearby' flock mates of the given boid (using metric or topological technique) and calculates
/// a repulsion desire away from each of them (weight)
/// <param name="flockmates">The 'nearby' flock mates</param>
/// <param name="boid">The boid asking for the 'desire'</param>
/// <returns>The calculated directional desire weighted</returns>
/// </summary>
public Vector3 CalcDirection(List<BoidBrain> flockmates, BoidBrain boid)
{
    ReCalcInfluenceZone(flockmates, boid);
    return m_InfZone.CalcDirection(boid) * m_DirectionWeight;
}
```