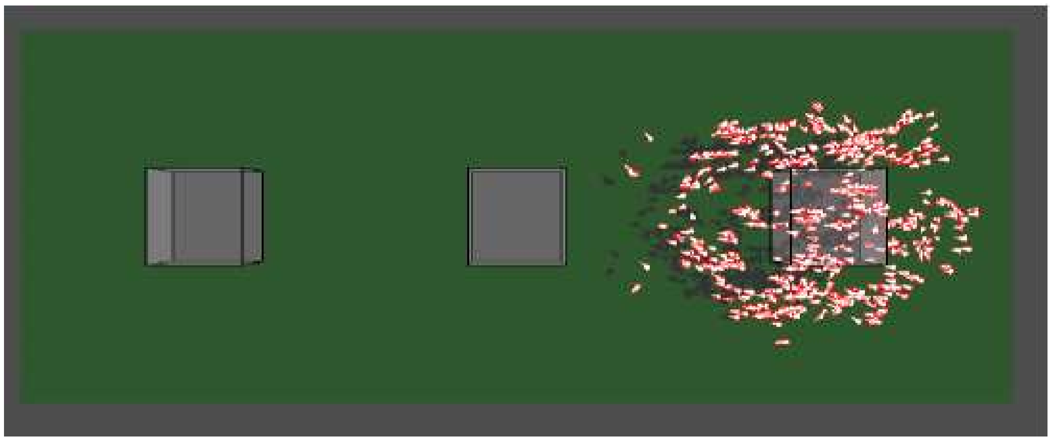
EMERGENT BEHAVIOUR PROGRAMMING LANGUAGE E.B.P.L

A DEVELOPMENT ENVIRONMENT FOR EMERGENT BEHAVIOUR



MASTERS THESIS

JONATHAN MACEY

N.C.C.A BOURNEMOUTH UNIVERSITY

21st August 2003

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This Thesis outlines the specification and design of an animation system for emergent behaviour. In particular the Emergent Behaviour Programing Language (EBPL) is presented as a solution for the development of complex behavioral animations of semi autonomous Agents.

Chapter 2 presents an short history of the control of large number of objects within animation packages as well as an overview of current multi-agent animation systems including Massive and Behaviour.

Chapter 3 introduces the concepts of a Multi-Agent System (MAS) and defines the terms Agent and Multi-Agent System with respect to the characteristics required in general and more specifically for an animation system.

Chapter 4 outlines the EBPL development environment and the relationship between the environment and the Agents.

Chapter 5 introduces the actual Agent brain programming language ebpl. This chapter discusses the design and structure of the language as well as how key aspects were implemented.

Chapter 6 Highlights some of the results of the project and list future development required for the system.

The Appendices include a short introduction to the Environment scripting language as well as the ebpl programming language itself. Appendices D and E show the results of an experiment to see if the Bin-Lattice data structure has any speed increase for the storage of Agents in a MAS over a non-spatial data structure.

Further details of the language and a series of examples as presented in the separate EBPL User Guide and Language Reference document, with a full series of animations presented on the Video.

All the source code for the project as well as HTML and pdf documentation of the *BrainCompiler* and *ebpl* programs is presented in the C.D. at the end of this document.

Chapter 2

Previous Work

2.1 Introduction

With the increasing power of modern computers the production of complex, quality animations has become a lot simpler. However managing the complexity of large scale character animations has become a major area of research [Par02]. Simple scenes with a small amount of characters can still use traditional key-framed sequences but with the film industry demanding scenes involving many thousands of characters the management of this complexity has become more of a problem.

Many solutions to this problem have been proposed with table 2.1. showing a simple taxonomy of the major group behaviour solutions.

Type of Group	Number of Elements	Incorporated Physics	Intelligence
Particles	Many	Much with Environment	None
Flocks	Some	Some with Environment and other Agents	Limited
Autonomous behaviour	Few	Little	Much

Table 2.1: Characteristics of Group Behaviour[Par02]

Each of the above solutions have advantages and disadvantages within an animation system which will be discussed in the following sections.

2.2 Particle Systems

One of the earliest method of controlling large amounts of objects in a scene is the Particle System introduced by Reeves [Ree83, Ree85] in 1983 and refined in 1985. When viewed as a whole the particles create the impression of a single, dynamic, complex object [Par02] as shown in Figure 2.1.

Particle systems do not require the elements to have any intelligence and rely upon the use of physical modelling of the environment to control the group behaviour which is useful for the modeling of organic flowing objects but not as useful for the control of humanoid forms.

2.3 Flocking Systems

In 1987 Reynolds [Rey87] extended the original Reeves particle system into a *"sub-object system"* where each particle, represented by a point in the Reeves system, is replaced by an animated object. Each of these

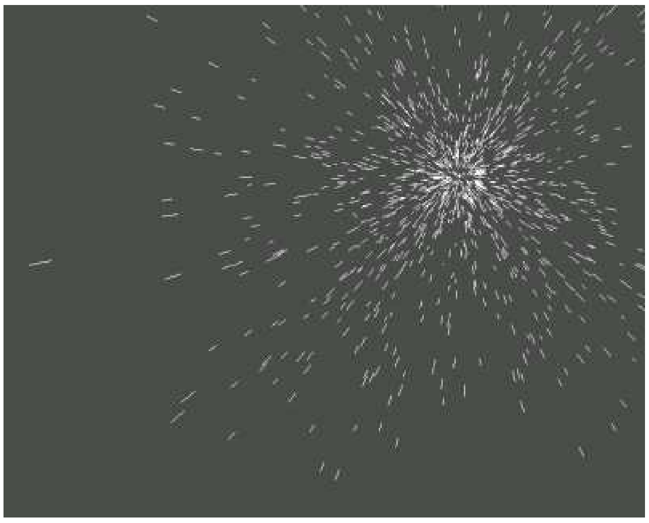


Figure 2.1: Simple Particle system Created in EBPL[1]

objects which Reynolds refers to as “Bird-oids” or “*Boids”* are given limited intelligence which when combined into a collective whole produces the flocking behaviour.

The original Boids have three simple rules :-

1. **Collision Avoidance** : avoid collisions with nearby flock mates.
2. **Velocity Matching** : attempt to match velocity with nearby flock-mates
3. **Flock Centering** : attempt to stay close to nearby flock-mates.

Combining these rules produce a simple flocking behaviour as shown in figure 2.2

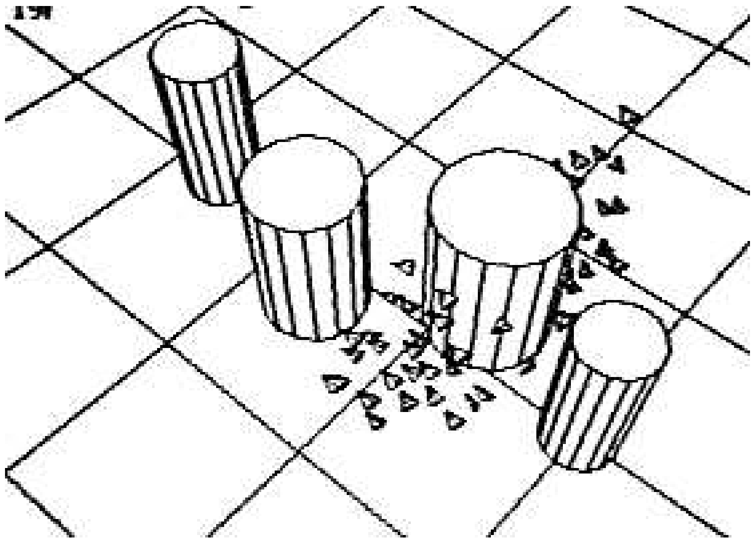


Figure 2.2: A simple Flock [Rey87]

To add to the flexibility of the system Reynolds added a simple scripting system which allows the animator to control various elements of the flock, for example the flock center can be assigned to a path to force the flock to follow a certain route, whilst still retaining the overall flocking behaviour.

Whilst giving a natural movement and good collective behaviour these systems are still very simple and deterministic, the boids have no real intelligence and have no ability to learn.

In more recent papers Reynolds has introduced slightly different rules for his flocking behaviours calling them "*steering behaviours*" [Rey99]. The new rules are as follows :-

- **Separation** : steer to avoid crowding local flock mates.
- **Alignment** : steer towards the average heading of local flock mates.
- **Cohesion** : steer to move toward the average position of local flock mates.

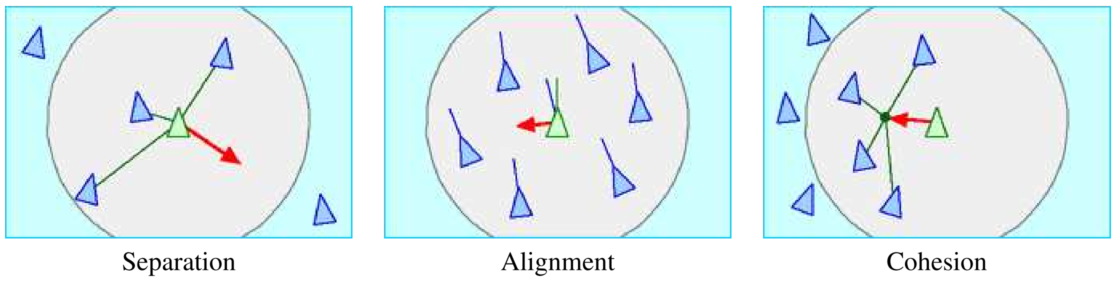


Figure 2.3: Steering Flocking rules [Rey99]

These behaviours are shown in Figure 2.3 and create a different type of flocking to the original rules. A useful resource for experimenting with these rules is the "OpenSteer" project [Rey03] which provides a test-bed for the development of steering behaviours for Agents.

2.4 Autonomous Behaviour

In recent years there has been a lot of research into the use of genetic algorithms and artificial intelligence (A.I.) to create autonomous creatures who can evolve behaviours. In 1991 Karl Sims [Sim91] produced a genetic language which allowed the simulation of a neural system for evolutionary creatures. More recent work such as Breve [Kli] has continued this process with creatures learning to walk, swim and fly based on their perception of the world, their own limbs and self. Figure 2.4 shows a simple creature learning to walk based on several genetic fitness algorithms. Each time the simulation is run several generations are created and the distance traveled is measured. At the end of each epoch the two Agents who have traveled the most are then bred to produce the new generation of Agents and the cycle begins again.

Although this process is useful for A-life research many thousands of epochs are required to produce a useful walking agent which is not particularly useful for the quick production of an animation.

2.5 Computer Animation Using Emergent Behaviour

In the last few years many commercial animation packages have been developed to allow the simulation of large groups and flocking type behaviours, some of these take the form of plugins for existing animation packages such as Maya [Ali03] and Softimage [Sof03] whilst others are standalone packages such as Massive [Mas03] and Behaviour [Sof03].

The following sections highlight some of the features of these systems in an attempt to aid the specification of an emergent animation system.

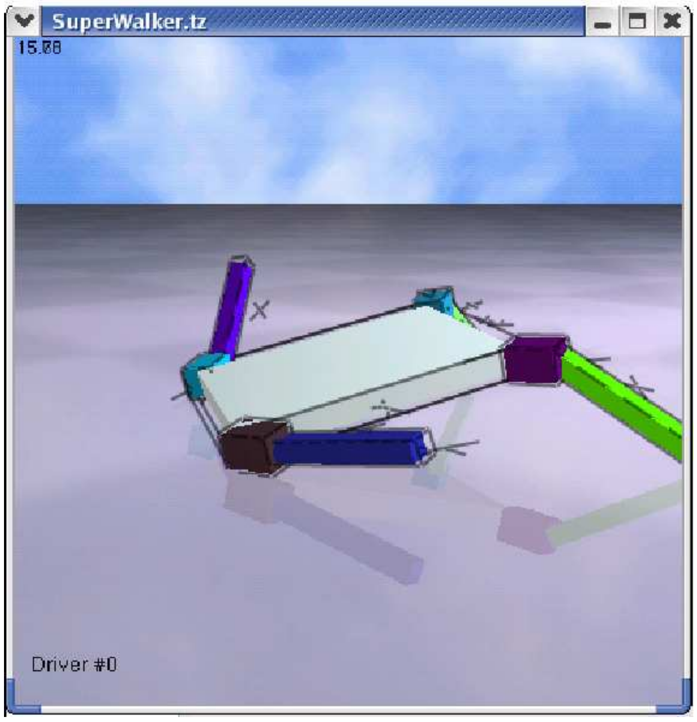


Figure 2.4: An Agent learning to Walk [Kli]

2.5.1 Behaviour

Behaviour [Sof03] from Softimage "*features an event-based state-machine programming environment, originally based on technology that combines motion synthesis of 3D characters with real-time control systems*", although it is a stand-alone program it is very reliant upon Softimage XSI for the modelling and rendering of the Agents as shown in Figure 2.5.

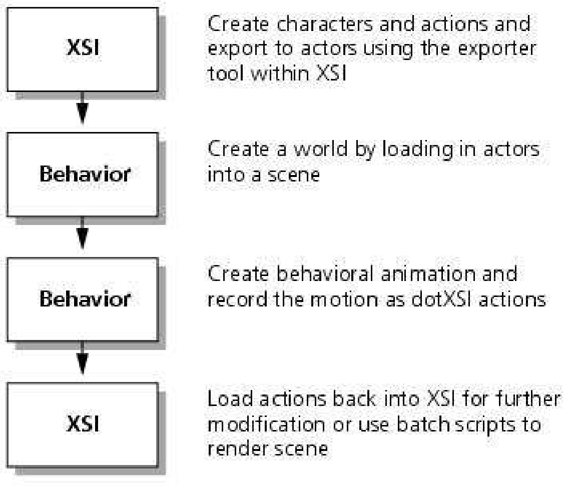

XSI
Create characters and actions and export to actors using the exporter tool within XSI
Behavior
Create a world by loading in actors into a scene
Behavior
Create behavioral animation and record the motion as dotXSI actions
XSI
Load actions back into XSI for further modification or use batch scripts to render scene

Figure 2.5: Behaviour Work Flow [Sof03]

At the heart of Behaviour is a programming language called *Piccolo* which is an object based language created for scripting purposes. It is integrated with the Hierarchical Finite State Machines (HFSMs) that Behaviour uses for its Agent control.

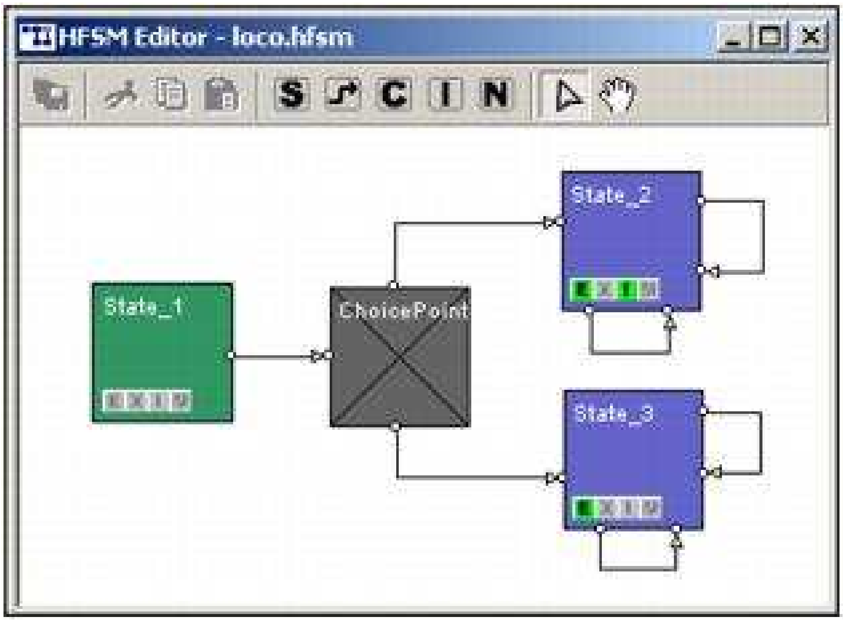


Figure 2.6: Behaviour H.F.S.M. [Sof03]

2.5.1.1 HFSM

The logic behind agent control in Behaviour is driven by a series of Hierarchical Finite State Machines as shown in Figure 2.6. These are triggered by a the agent depending upon the current state of the agent and it's interactions with the environment. Each of the HFSM's can be stacked so more complex behaviour can be created depending upon multiple agent states.

Although the HFSM structure is simple to use and configure it still leads to very predictable behaviour unlike the system employed by Massive.

2.5.1.2 Helper Functions

Behaviour / Piccolo has many built in helper functions to allow for the quick generation of crowds and flocks, functions such as *InitializeSimpleFlock* which will create a flock with a series of actors in one command. This allows for very simple programs to be generated quickly and efficiently for the novice user. The crowd shown in Figure 2.7. is created with the following code

```
function main()
  {
  engine.RunScript("rtk", nil);
  var lookpt = new DrvVector(1000, 0, 0);
  ClickClusterCircle("Gold", "happy", 8, 25, 8, lookpt);
  }
```

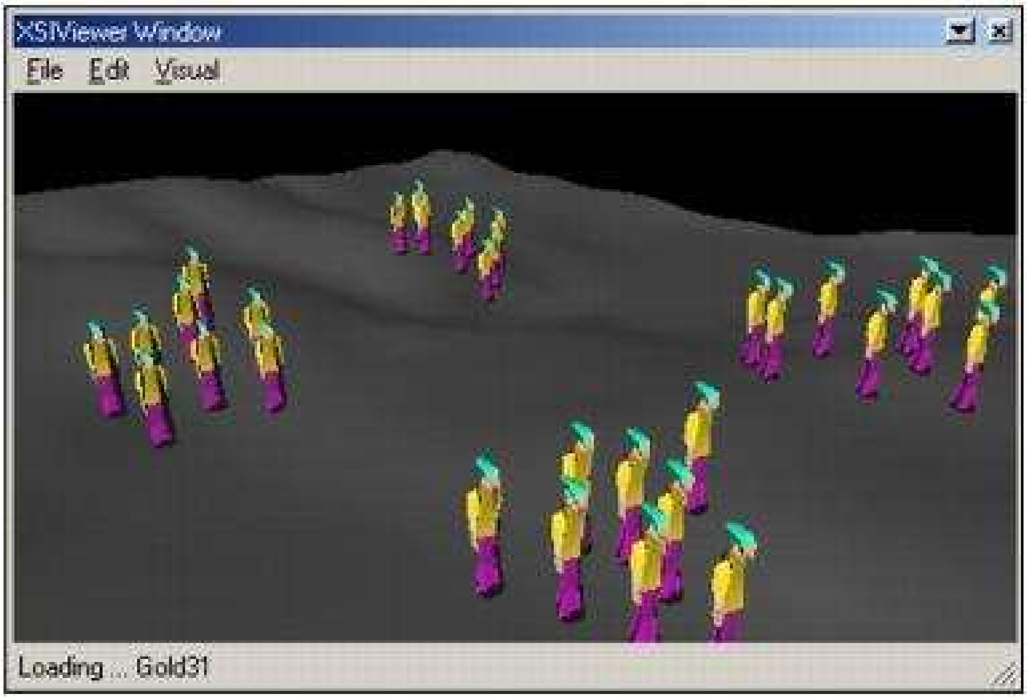


Figure 2.7: A Behaviour crowd following terrain [Sof03]

2.5.2 Massive

Unlike Behaviour, Massive [Mas03] is a fully contained simulation / rendering system for the development of complex crowd and action simulations. At the heart of the Massive system is the Agent's Brain which is responsible for the entire control of the Agent and the interactions of multiple Agents as shown in Figure 2.8.

Figure 2.8: Massive Agents Fighting [Koe02]

2.5.2.1 Agent Brains

Massive Agents function as complex beings subject to physical forces, with specific body attributes [Koe02]. The Agent has traits that range from biological (size, good eyesight etc) to behavioral (aggressive, happy etc). These are then combined together with pre-set animation cycles to determine the Agents movement, behaviour and animation cycle.

2.5.2.2 Fuzzy logic

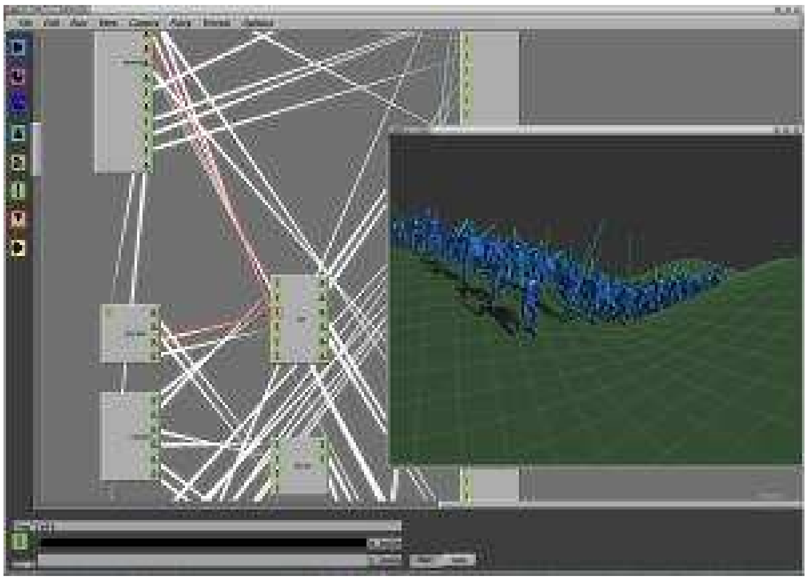


Figure 2.9: The Massive Agent Brain Editor [Mas03]

The Agent's Brain is connected by a series of fuzzy logic rules programmed by the user (Figure 2.9.), which give a "*much more human"* [Koe02] response to the environment. For example "*an archer in Massive doesn't just hit or miss a target. He adjusts his aim as each arrow follows a slightly different random trajectory. Success or failure is based on many complex and sometimes interrelated factors, such as skill level, the weather, and even his mood at the time*" [Koe02].

This approach means that the creation of a scene with massive is highly reliant upon the emergent behaviour of the system. When creating a Massive scene the Actors are placed in the environment with a specific task and the simulation is run to see what the results are. In some cases in the middle of a fight some agents have decided not to fight but to run away which they are not specifically programmed to do [Koe02].

To avoid such occurrences the Massive characters are selected and cloned to weed out ineffective agents similar to the approach used by Sims [Sim91], however no built in evolutionary algorithms are used.

More information on Massive can be found in the email from Stephen Regelous in Appendix H.

Chapter 3

Multi-Agent Systems

Multi agent systems (MAS) are finding increasing uses in many diverse fields, as shown in Figure 3.1. For an animation system only the “Multi-agent simulation” and the “Building artificial worlds” need be considered, however some of the flocking and crowd behaviours discussed are now being applied to the field of robotics [Par, SV00, HDT].

To aid with the development of an environment for emergent behaviour it is useful to determine what elements are needed to make up the environment and the Agents.

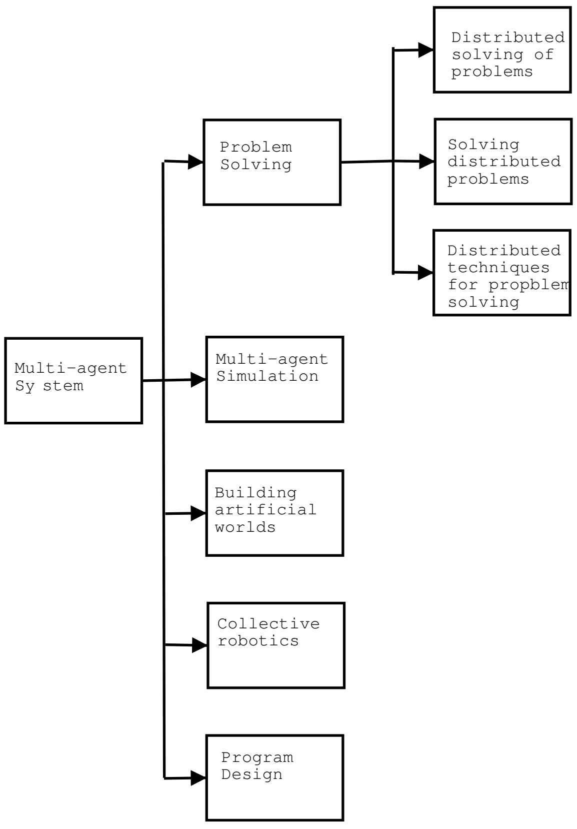

Multi-agent Sy stem
Problem Solving
Multi-agent Simulation
Building artificial worlds
Collective robotics
Program Design
Distributed solving of problems
Solving distributed problems
Distributed techniques for propblem solving

Figure 3.1: A classification of various types of application for Multi-Agent Systems. [Fer99]

3.1 Agents

Ferber [Fer99] defines an Agent by the following criteria

Definition

An Agent is a physical or virtual entity

(a) which is capable of acting in an environment,

(b) which can communicate directly with other Agents,

(c) which is driven by a set of tendencies (in the form of individual objectives or of satisfaction / survival function which it tries to optimize,

(d) which possesses resources of its own,

(e) which is capable of perceiving its environment (but to a limited extend),

(f) which has only a partial representation of this environment (and perhaps none at all)

(g) which possesses skills and offers services,

(h) which may be able to reproduce itself,

(i) whose behaviour tends toward satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representation and communications it receives.

For most animation systems this definition is over complex and can be refined to produce a simpler definition as follows :-

Definition

An Agent is a physical or virtual entity

(a) which is capable of acting in an environment,

(b) which can communicate directly with other Agents,

(c) which is driven by a set of tendencies,

(d) which is capable of perceiving its environment (but to a limited extend),

(e) which may be able to reproduce itself (but only with user intervention),

(f) whose behaviour tends toward satisfying its (very limited) objectives.

This definition is sufficient to allow the generation of a simple generic Agent capable of satisfying animation goals. As quick production of animation is a key issue for most development systems it has been decided not to add A.I. to the Agents as the training and evolution of an Agent Brain will be very time consuming and is not used in either Massive (see 73) or Behaviour.

Once the Agent has been created it needs to be situated in an environment to actually produce the MAS. In general Ferber [Fer99] defines a MAS as

Definition

The term Multi-Agent System (or MAS) is applied to a system comprising the following elements :

(1) An environment, **E**, that is a space which generally has a volume.

(2) A set of objects, **O**. These objects are situated, that is to say, it is possible at a given moment to associate any object with a position in **E**. These objects are passive, that is, they can be perceived, created, destroyed and modified by the agents.

(3) An assembly of Agents, **A**, which are specific objects ($\mathbf{A} \subseteq \mathbf{O}$), representing the active entities of the system.

(4) An assembly of relations **R**, which link the objects (and thus Agents) to each other.

(5) An assembly of operations, **Op**, making it possible for the Agents of **A** to perceive, produce, consume, transform and manipulate objects from **O**.

(6) Operators with the task of representing the application of these operations and the reaction of the world to this attempt at modification. (defined as the laws of the Universe [Fer99])

Again this definition is over complex for most animation systems and may be simplified as follows

Definition

The term Multi-Agent System (or MAS) is applied to a system comprising the following elements :

(1) An environment, **E**, which is a volume (Bounding Box)

(2) A set of objects, **O**. which include terrain. These objects are situated in **E**. These objects are passive, that is, they can be perceived, by the Agents.

(3) An assembly of Agents, **A**, which are specific objects ($\mathbf{A} \subseteq \mathbf{O}$), representing the active entities of the system.

(4) An assembly of relations **R**, which link the objects (and thus Agents) to each other.

(5) An assembly of operations, **Op**, making it possible for the Agents of **A** to perceive, objects from **O**.

(6) Operators which allow the Agents to respond to the Environment **E**.

In the above definition all references to the Agent modifying the Objects within the environment have been removed. In general an animation system requires the Agents to react and respond to the environment but not to be capable of modifying it, however the Agents are capable of modifying their own actions as well as the actions of the other Agents.

3.2 MAS Programming Languages

It is inevitable when developing a MAS that some form of programming language will be required, these can be split into different categories as shown in Figure 3.2.

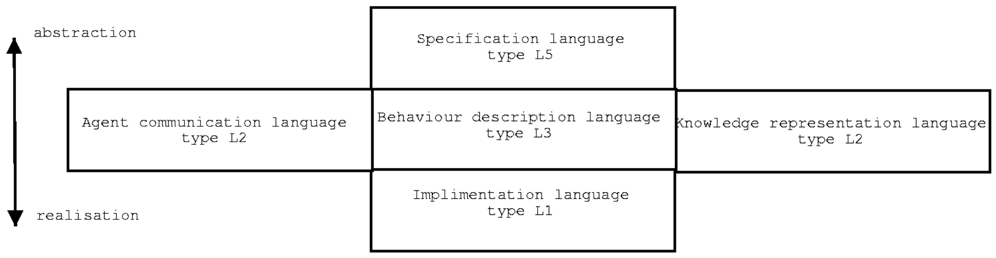

abstraction
realisation
Specification language type L5
Agent communication language type L2
Behaviour description language type L3
Knowledge representation language type L2
Implimentation language type L1

Figure 3.2: A classification of various types of application for multi-agent systems. [Fer99]

3.2.1 Type L1 Implementation Languages

These are used to develop the actual MAS and are generally high level Object Oriented languages such as C++. They are used to create both the Agents and the Environment generally from some form of lower level script.

3.2.2 Type L2 Communication Languages

This type of language provides for the communication between Agents and is generally of a lower level than the Implementation language. It can take the form of a script / dialog or simple stack based commands.

3.2.3 Type L3 Languages for describing behaviours and the laws of the environment

These languages are used to describe what is happening. They represent the way the Agents respond to each other and the environment. Again they primarily take the form of a script / dialog.

3.2.4 Type L4 Languages for representing knowledge

These are used for cognitive agents, and require some form of Neural network or Fuzzy logic system. The language must also have the ability to store the knowledge as well as adapt the programming to suit the newly learned rules.

3.2.5 Type L5 Formalization and specification Languages.

These type of language are a type of Meta language which usually take some formal mathematical approach for describing the agents behaviour. In general these are too specific for the need of an animation package and are discounted for the current application.

3.3 Creating a language for Emergent Behaviour

Using the above language definitions it is possible to create a development environment for Emergent systems, as pointed out in [Fer99] *"all these languages are obviously connected : a negotiation protocol, for example, will use a type L2 language as a carrier for conversation and a type L3 language to describe the mechanisms for the interpretation of the protocol"*.

Due to this abstraction developing a general purpose language for emergent systems is a very complex task. Some approaches such as the Breve system language “Steve” [Kli]. use a C like procedural language with access to built in functions. The Piccolo language used by Behaviour [Sof03] is very similar to Java Script, however it has many built in functions to access the external XSI api allowing the language to be at first simple, yet extensible and powerful.

When designing a language it is important to determine the scope and the application of the language. As the primary goal for EBPL is the programming of the Agent Brain the design becomes much simpler as will be discussed in Chapter 5.

Another area to be considered is the generation of the Environment for the MAS. This could be hard coded in the system or again and extensible scripting system this will be discussed in Chapter 4.

In the development of EBPL it was decided to use C++ for the main system and environment development due to it’s speed and relative portability. Java was discounted due to it’s lack of real 3D graphics support and slow speed of execution.

For the ebpl language a high-bred C scripting language was developed to allow quick development of scripts for the Agent Brain, it must also be noted that most animation packages use a C like scripting system so most animators will be in some way familiar with them.

Chapter 4

EBPL Environment

The EBPL environment loads in a simple script and runs the simulation, the format of the scripting language is very simple as outlined in Appendix C. The script is responsible for setting up the Agents, and Environment including terrain and environment objects.

The environment is responsible for containing all the Objects within the scene, as well as the calling of all the Agent's brain routines as shown in Figure 4.1.

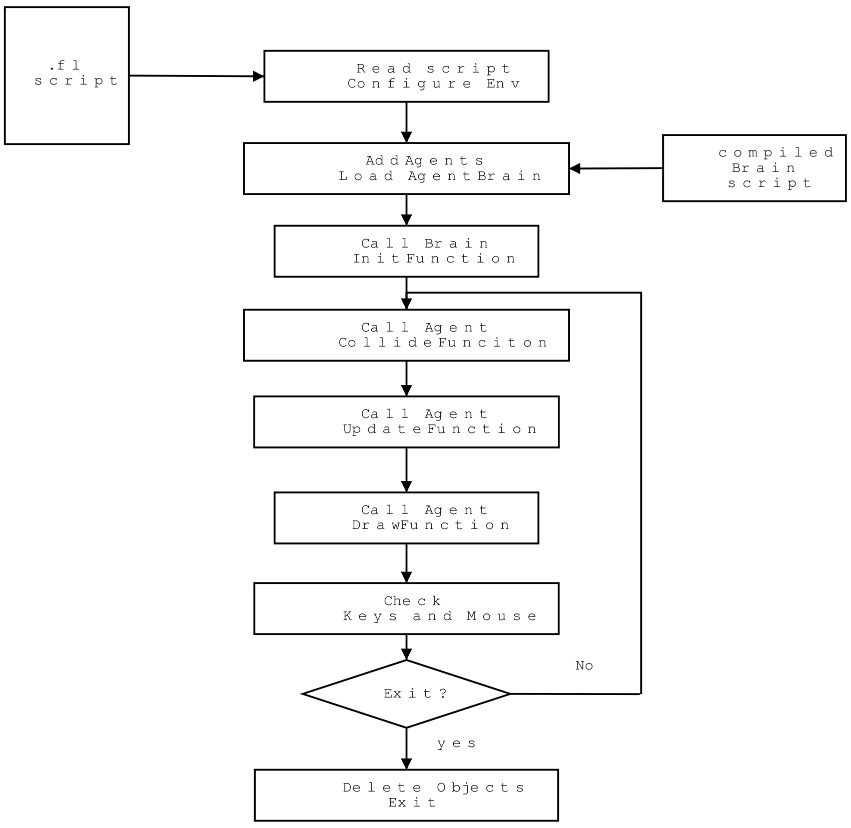


Figure 4.1: EBPL environment flow chart

The Agents must be stored within the MAS and ebpl uses an optimized spatial data-structure as outlined in the following section.

4.1 Bin Lattice Data Structure

One of the most CPU intensive elements of most flocking and emergent systems is the Agent collision detection and response. This leads to an asymptotic complexity of $O(n^2)$ where doubling the number of Agents quadruples the amount of time to do collision queries [Rey00].

By changing the way the agents are stored based on a spatial data structure this complexity can be reduced, for each iteration of the system each Agent will be stored in a new structure based on it's location in the simulation which will decrease the query time.

The bin lattice structure embeds each of the agent in a user defined lattice where each agent is a member of a bin irrespective of the emitter it belongs to Figure 4.2. shows a simple flock embedded in a 5x5x5 lattice structure.

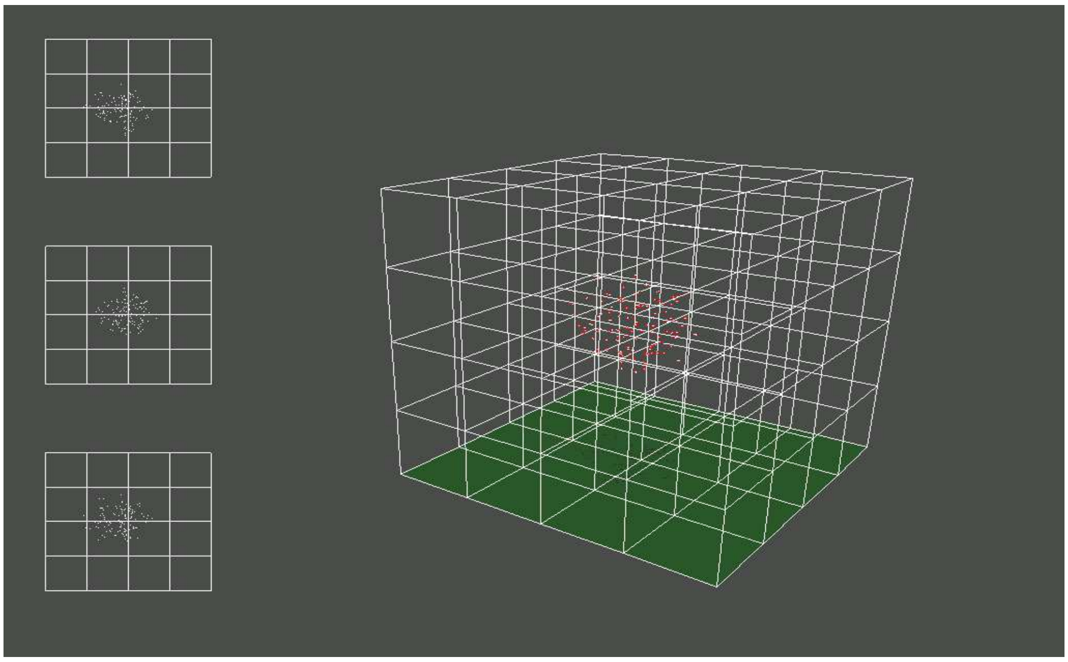


Figure 4.2: Bin Lattice Structure With Embedded Flock

When the system is started the Bin-Lattice structure is created with a user definable number of divisions for each of the lattice axis. From this a number of bins are created with each bin having a user definable maximum number of Agents, for simplicity this is usually set to the maximum number of Agents in the system but if a sparse flocking algorithm is used this may be reduced as the number of possible Agents in each Bin will be less.

At each frame the *FillBins* method is called and the Bins are filled with pointers to the Agents. Each Agent's Brain is then made aware of the Lattice-Bin structure so the Agent may query the bin and check for other Agents presence within the Bin. A full investigation of the speed increase of this structure is presented in Appendix D. with an evaluation of an non Brain flocking system.

4.2 AgentEmitters

The Agents are added to the system using an AgentEmitter, this sets up the Agents initial position and the Brain script to be loaded. Each iteration of the system updates the Emitters and this in turn runs the Brain update and collision routines for each Agent.

4.3 Terrain

For every simulation a Groundplane must be made available, EBPL uses an polymorphic GroundPlane class to allow three different types of GroundPlane. Each Agent has access to the GroundPlane Y level via a series of Opcodes and the Agent may be positioned on the GroundPlane via accessing this value.

4.3.1 GroundPlane

The base GroundPlane is a simple flat terrain with a uniform height. The height Y value is set within the .fl script and can be accessed by using the *FpushGPYleve*l EBPL opcode.

4.3.2 Image Ground plane

Simple terrain is generated by the use of a bitmap image such as that shown in Figure 4.3. Each pixel is used to generate a Height value for the terrain triangle strip. The red channel of the RGB image is used for the height value with Black representing a 0 value and White the maximum height value. In future versions of ebpl any channel will be available for the height map including the Opacity value.

The overall height is then scaled by a user definable value to allow fine tuning. The extents of the triangle strip are taken from the image for example the image in Figure 4.3. is 64 x 64 which generates a 64 x 64 triangle strip terrain.

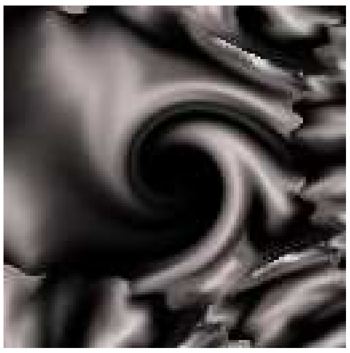


Figure 4.3: Terrain Height map

Textures may then be added to the image as shown in Figure 4.4. The texture image is usually larger than the height map image to allow for more detail.

The height value of the Agent within the terrain is calculated directly from the image height-map by using the Agents X and Z position as the offset into the Image array using the following code.

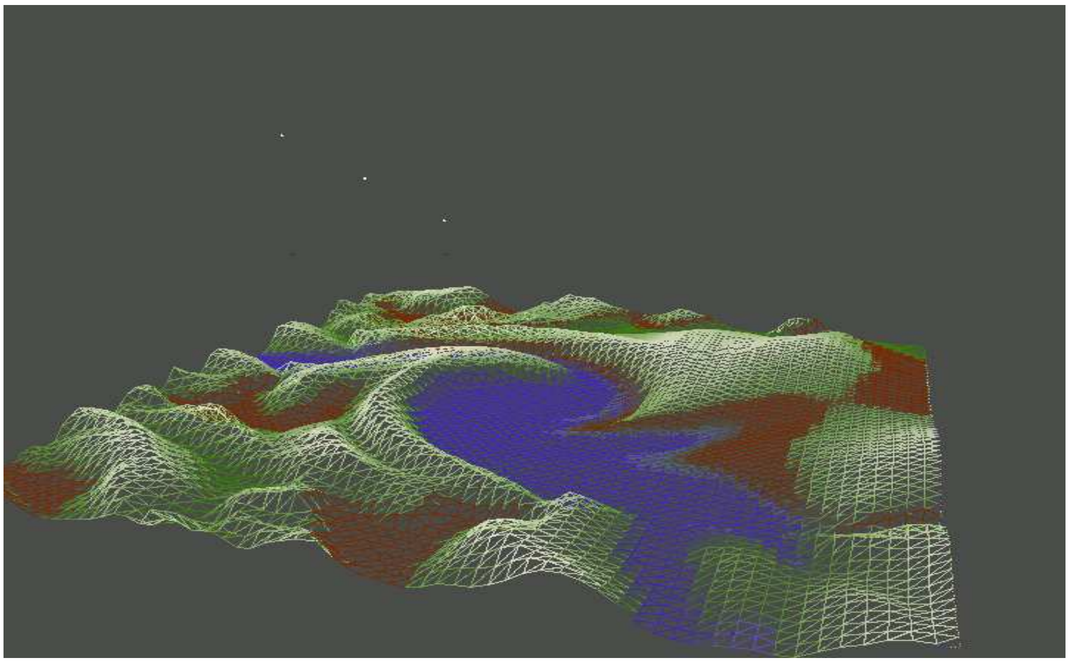


Figure 4.4: Terrain showing Triangle strips and Texturing

```
GLfloat ImageGroundPlane::GetHeight(GLfloat x, GLfloat z)
{
// if were not using an image return the normal y value
// else determine where in the terrain we are and get that value
 if( (x < MinX || x > MaxX) && (z < MinZ || z > MaxZ) )
     return MinY;
 else
  {
   GLfloat XX=rint((rint(x)-MinX)/divx);
   GLfloat ZZ=rint((rint(z)-MinZ)/divz);
   int index=(int)XX + (int)(ZZ*Xsteps);
   if(index >(Xsteps*Zsteps))
      return MinY;
   else
     return gpPoints[index].y;
  }
}
```

4.3.3 Object based terrain

Terrain my also be generated using an Alias-Wavefront .obj [Ali03] file, the terrain is assumed to be aligned with the Y axis upward and the Z axis moving in and out of the screen. At present only triangulated .obj files are processed however support for quad based files are in development.

4.3.3.1 Calculating height

To calculate the current Y value of the Agent within the terrain the GetHeight method is used. This is based on calculating the point normal for each of the faces in the obj file and determining if the Agent's x-z position is withing the triangle. The method used is modified from [jr01] as shown in Algorithm 1.

Algorithm 1 Calculating if a point is within a triangle

Given each Triangle within the .obj file extract the Points T_0, T_1, T_2 and construct the Lines b_1, b_2, b_3 by subtracting the the Triangle Points as follows :-
$b_1 = T_1 - T_0, b_2 = T_2 - T_1$ and $b_3 = T_0 - T_2$
The point Normal of each line is constructed by negating the y value and swapping it for the x so
$PN_1 = [-b_{1y}, b_{1x}, 0]$, $PN_2 = [-b_{2y}, b_{2x}, 0]$ and $PN_3 = [-b_{3y}, b_{3x}, 0]$
Next the Point to be tested P is subtracted from each of the original triangle points so
$b_1 = T_0 - P$, $b_2 = T_1 - P$ and $b_3 = T_2 - P$
To determine if the Point is within the current triangle we calculate the dot product of the triangle points with their point normals
$Test_1 = b_1 \bullet PN_1$, $Test_2 = b_2 \bullet PN_2$ and $Test_3 = b_3 \bullet PN_3$
If all the Test values are $<= 0$ then the point is within the triangle and the Y height offset value is calculated using linear interpolation of the triangle points.

4.3.4 Noise

When programming an Agent's movement, from observation, it was noticed that the movement was very uniform and unnatural. To improve this different approaches have been investigated including the use of simple random numbers added to the Agent's movement.

The most successful method of adding more natural random movement to the Agents is the addition of Perlin Noise [eta98] to the Agent's motion. A noise function is generated using the source code from the book "*Texturing and Modeling A procedural Approach*" [eta98] which allows for noise to be generated from a seed value.

There are 5 different noise functions built into the Agent at present these range from simple noise to more complex textural based systems such as Turbulence and a Marble texture function.

The noise function may be calculated within the EBPL script by passing in any of the Agent's variables such as Direction or the Agent's current position. This produces very subtle yet interesting result similar to Brownian motion. This effect when added to the Agent's calculated motion gives a more natural movement without overriding the flocking instinct of the Agents.

Chapter 5

Emergent Behaviour Programming Language (E.B.P.L.)

Much research is focused on the design and implementation of compiler / parser. For a general overview of the process the classic "Dragon Book" [AVAU86] is recommended. More specific methods of parsing and code generation are outlined in [Knu65, Spe88, Pij00]. However for the generation of the programming language EBPL a virtual machine was decided upon. This method is further supported by Ertl *et-al* [EGKP02] who state that virtual machines are popular because of :-

- Ease of implementation.
- Portability.
- Fast Edit-Compile-Run Cycle.

The system developed by Ertl *et-al* [EGKP02] is freely available and allows the user to design their own grammars and produce a virtual machine which will interpret this code and be incorporated into any other system. However this option along with the use of compiler generator such as Lex / YACC and Bision were also discounted due to the small scale of the system required. If the language is to be extended into a more complex grammar these systems will be re-evaluated and possibly used.

As the language's primary focus is the programming of the Agent's brain and the control of the Agent the language design must be based upon the definition of the Agent and it's ability to sense and react to the environment. Therefore to design the language first the Agent and the Agent Brain must be specified.

5.1 Agent Characteristics

Using the definition of an Agent on page 10 we can design the initial Agent characteristics and thus the Brain for the Agent. The initial Agent is shown in the class diagram in Figure 5.1.

The main element of the Agent is the Brain. This is a separate class which contains the Brains instructions as well as the "memory" for the Agent. This will be discussed in more detail in Section 5.2.

All the Agents must also be aware of the environment, in the system there is one global environment which is created within the main development system from the .fl script.

Although the system is designed to be flexible there are still a few built in variables for the Agent to allow quick access to the environment. The Position attribute allows the current Agents position to be stored. This may then be used to calculate the flock center within the main ebpl environment.

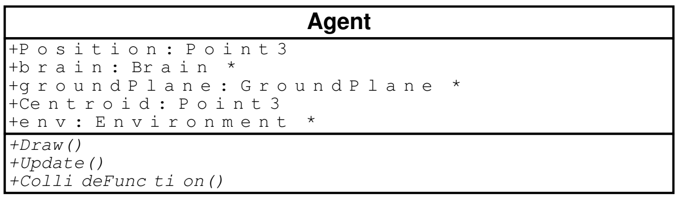


Figure 5.1: Agent Class Diagram.

The current flock center (Centroid) is calculated as the average position of all the Agents in the environment for each cycle of the system and each Agent is then made aware of this.

It is up to the user if these variables are used within a EBPL program and they do not actually have to be set.

5.2 Agent Brain

At the heart of the Agent Brain is a series of stacks and an opcode interpretor. The main function of the Brain is to execute the opcodes and produce output. The basic structure of the Brain is shown in Figure 5.2.

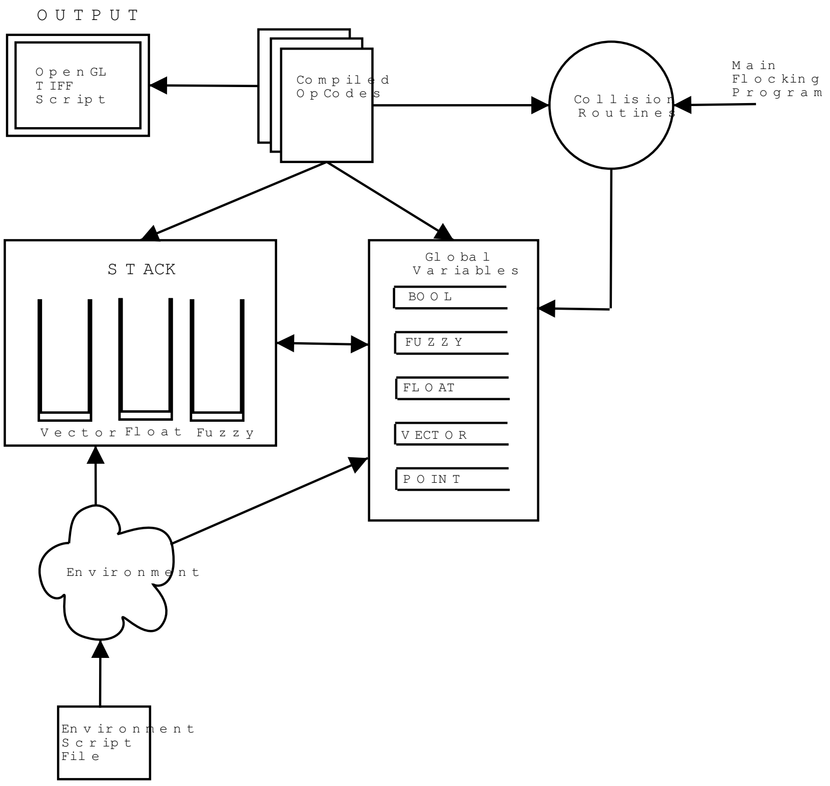


Figure 5.2: Agent Brain Structure

5.2.1 Brain Stacks

The stack used for the Agent Brain is based on a simple C++ template as outlined in [Jos99]. The use of templates allows the same instructions to be used for each of the stack but with different data types and classes as shown in Figure 5.3.

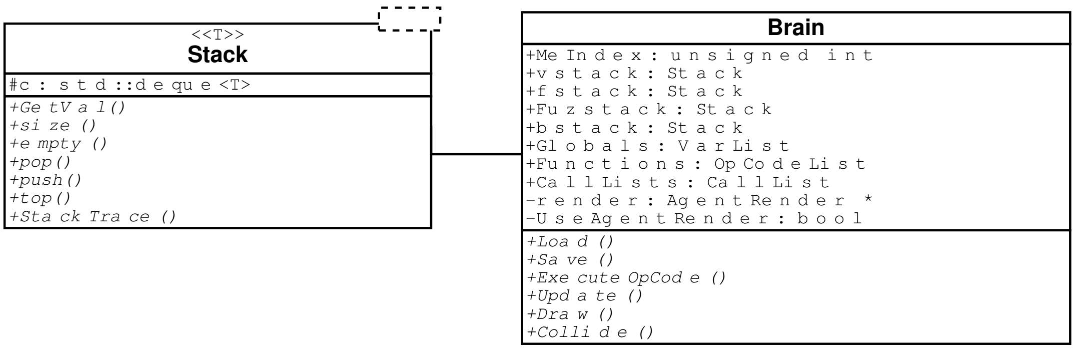


Figure 5.3: Agent Brain class diagram

The Agent Brain has four built in stacks as follows :-

Float	A floating point stack.
Vector	A stack for operation on the Vector class.
Bool	A boolean stack.
Fuzzy	A stack for operation on the Fuzzy object class.

Unlike most interpretive languages EBPL operations are not entirely dependent upon the stack architecture. Most data types can be accessed queried and modified directly, however some operation such as access to tuple data type entries still rely upon the stack.

At present there are 17 stack operations for the float stack ranging from push / pop to math functions and degrees to radians and radians to degrees conversions all the functions are outlined in Appendix A.9.

5.2.2 Global Variables

When the Brain is loaded the global variables defined in the brain script file are loaded in the form of a Variable list. Each element of this list is a class called a VarObj as shown in Figure 5.4.

This object can assume any data type and has built in methods for specialist OpenGL functions such as Vertex, Normal and Translate which only operate on the Vector and Point3 classes and are ignored by the other data types to allow polymorphic access to opcode instructions.

Once the global variables are loaded into the brain the pointer to the Global variable list is associated with the Brain making the source code re-locatable for each instance of the Brain.

5.3 Compiler / Parser

The EBPL compiler is a simple LR(1) parser [Spe88, Knu65] designed specifically for the byte-code generation for the Agents Brain. It uses a three stage parsing routine and saves the byte-code in ASCII format. It structure is shown in Figure 5.5.

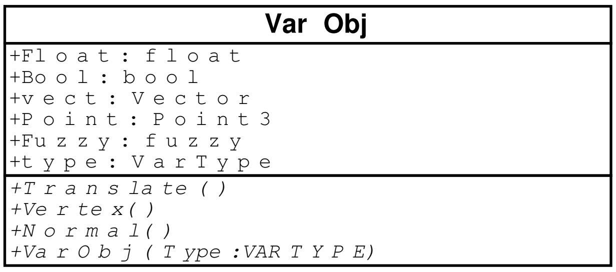


Figure 5.4: VarObj Class diagram

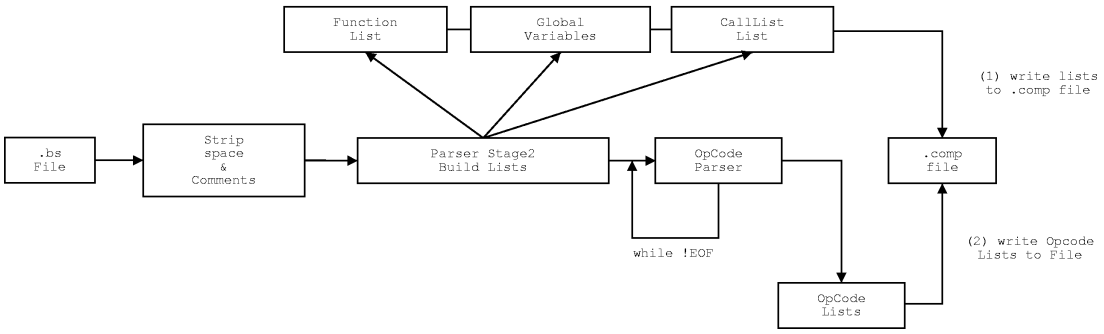


Figure 5.5: EBPL compiler structure

5.3.1 Parser Stage 1.

The first stage of the parser strips any white space from the source brain script file as well as C / C++ style comments. This makes the design and implementation of the other parsing phases easier.

Once this is done a new temporary file is created with the line number of the current instruction followed by the actual instruction code separated by a newline character “\n”. Once this is complete the new file is fed into the next stage of the parser.

5.3.2 Parser Stage 2.

The second stage of the compiler looks for variables, functions and call lists and produces look up tables for each of them containing the name of the variable, function and call list so they can be referred to in the rest of the source file using the variable or function name instead of the index generated by the compiler.

5.3.2.1 Variables

At present five variable types are supported in EBPL, as shown in Table 5.1.

The syntax of the variable definitions is tightly specified as shown in the Examples column of Table 5.1. All variables must be initialized when declared and have global scope to the brain. This means that all variables are available to all functions in the source file as well as accessible by other Agents. If integer values are required in the actual development environment (for example a case statement will only work with integer values) they are cast from a float to an int using the *(int)* C++ cast.

Variable Type	Description	Example
Float	A signed floating point value	Float CentroidWeight=50.0;
Bool	A boolean value	Bool HitAgent=false;
Vector	A four tuple vector class [x,y,z,w]	Vector Dir=[0,-1,0,0];
Point	A three tuple Cartesian point class [x,y,z]	Point Pos=[0,0,0];
Fuzzy	A full fuzzy logic class with fuzzy operators	Fuzzy NearAgent=0.2;

Table 5.1: EBPL variable types

5.3.2.2 Functions

EBPL has two types of functions *default* and *user defined*. At present there are 4 *default* functions which must be present in every EBPL script. There may be any number of *user-defined* functions which act as procedure calls within the program.

Built In Functions	Description
InitFunction	Called when the Agent brain is created and used to initialize variables etc.
UpdateFunction	Called every iteration of the system to update the Agents position
DrawFunction	Called every iteration of the system to Draw the Agent
CollideFunction	Called every iteration to do collision detection

Table 5.2: EBPL variable types

The built-in functions are shown in Table A.1. The *InitFunction* is called once when the Agent is created whilst the other built-in functions are called in sequence for each Agent in every iteration of the system as shown in Figure 4.1.

User defined functions are generated by the *Function* keyword and can be called from within any of the main built-in functions.

5.3.2.3 Call Lists

Call lists are a way of creating switchable function calls depending upon an ordinal variable value similar to the C/C++ *switch - case* statements. The creation of a call list is a two stage process as follows :-

```
DefineCallList TestList;
CallListItem TestList foo;
CallListItem TestList bar;
```

First a call list name is defined to make storage for the Function pointers to be allocated to the list. Next list items may be added to the list. To use a CallList within a function the following code is used

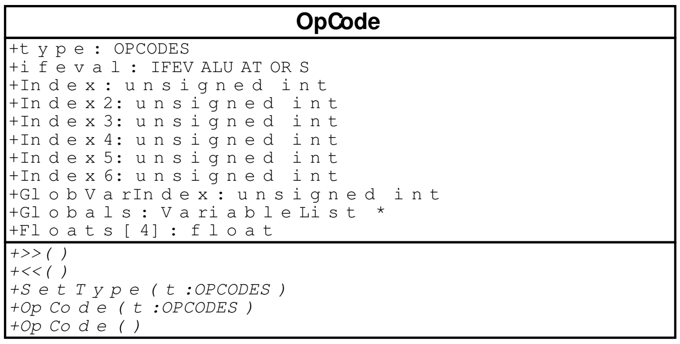
```
float ListValue=0;
// function foo called
CallList TestList ListValue;
AddD ListValue 1;
// Function bar called
CallList TestList ListValue;
```

5.3.3 Parser stage 3

The final parser stage compiles the Opcode source and produces the byte-code used by the brain. This is done by parsing each of functions using the ParseOpCode Method. This is invoked whenever on of the *built-in* functions or the *Function* keyword is encountered. This then loops until the *End;* keyword is found.

5.3.3.1 OpCode class

The Opcode Class as shown in Figure 5.6. is the main element of the whole EBPL language, every instruction in the language is given an enumerated type as shown in Appendix F. When the OpCode is encountered it is added to an OpCodeList class which is a simple container to store the OpCodes.



```
OpCode
+type : OPCODES
+ifeval : IFEVALUATORS
+Index : unsigned int
+Index2: unsigned int
+Index3: unsigned int
+Index4: unsigned int
+Index5: unsigned int
+Index6: unsigned int
+GlobVarIndex : unsigned int
+Globals : VariableList *
+Floats[4] : float
+>>()
+<<()
+SetType(t :OPCODES)
+OpCode(t :OPCODES)
+OpCode()
```

Figure 5.6: Agent Class Diagram.

Each of the OpCodes also has the ability to store directly a series of 4 floating point values and seven index values which may be used to point into the global variable list to access any of the script's variables. A separate enumerated type is also included to store the comparison operators used in the *if* and *ifelse* OpCodes.

Although this method is not particularly efficient from the point of view of Memory / data size, it does allow quick generation of byte code and easy interpretation of the byte code by the agent's brain.

5.4 EBPL language structure

The EBPL language is split into four distinct parts, these are

- Stack Based Operations.
- Variable Operations.
- Collisions.
- Drawing.

5.4.1 Stack Operations

Each stack has a series of different operations, the operations are based upon the stack data type and results from the operations are always placed back upon the stack. All stack operations are prefixed with the stack type name as shown in Table 5.3.

For example to push an element of a vector onto the floating point stack the following code would be used.

```
    Vector Dir=[0,-1,0,0];
    Fpush Dir y;
```

Operator Prefix	Description
F	Float stack operator pre-fix
FZ	Fuzzy stack operator pre-fix
B	Boolean stack operator pre-fix
V	Vector stack operator pre-fix

Table 5.3: Stack Operation Prefix

5.4.2 Variable Operations

All variables within EBPL have global scope and can be accessed within any function. They can be set directly or assigned a value based upon another variable.

The setting of individual tuple values can only be done via the float stack as outlined in Section 5.4.1.

Variables may also be used in comparison *if* and *ifelse* operations. These may only be carried out with variables of the same type and relies upon the overloaded comparison operators of Point3 and Vector classes.

As the comparison of the operators >, >=, <, <=, is ambiguous for Points and Vectors they have be defined to work for all tuple values for example the Point >= operator is defined as

```
bool Point3 :: operator>=(Point3& v)
{
if(x>=v.x && y>=v.y && z>=v.z)
     return true;
else return false;
}
```

A full list of *Point3* and *Vector* overloaded operators is shown in Appendix G. Future version of the system will use a quadrant based solution based upon the local co-ordinate frame of the environment.

5.4.3 Collisions

Each time the system is updated the brain's CollideFunction is called, this is the only time within an EBPL program that an Agent can access another Agent's data. Within the system all Agents are contained within a number of Bins (see Section 4.1. for more details). It is possible to loop through the Agents within the Bins and test for collisions. This is implemented using the *LoopBin* EBPL opcode.

Whenever the *LoopBin* structure is encountered the current Agent value is used and the rest of the Agents within the Bin are compared with the current Agent.

For convenience and speed EBPL has two built-in collision detection routines, however these and others may also be implemented within the EBPL language itself. These routines take as parameters variables from the EBPL script.

Discussion on efficient collision detection for animation can be found in [CLMP95, MW88, PG95, UOT83, KHM$^+$98], which discuss collision detection methods from simple sphere-sphere to more accurate vertex level collisions, however as only coarse collision detection is required these were discounted for simpler methods outlined in the next sections.

5.4.3.1 SphereSphereCollision

The Sphere-Sphere collision detection routine has been modified from the one presented in [Len01]. Given two spheres with radius r_1 and r_2 centered at Points $P_1$and P_2 as shown in Figure 5.7.

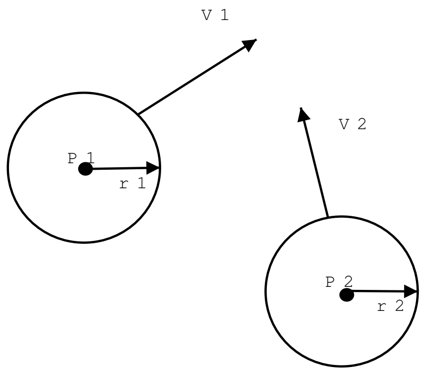


Figure 5.7: Sphere-Sphere Collision detection

A collision occurs when $dist <= minDist^2$ where $dist = relPos^2$, $relPos = P_1 - P_2$and $minDist = r_1 + r_2$ and is implemented using the following C++ code

```
bool SphereSphereCollision(Point3 Pos1, GLfloat Rad1, Point3 Pos2, GLfloat Rad2)
 // the relative position of the agent
 Point3 relPos; //
 //min an max distances of the agent
 GLfloat dist;
 float minDist;
 relPos =Pos1-Pos2;
 // and the distance
 dist=relPos.x * relPos.x + relPos.y * relPos.y + relPos.z * relPos.z;
 minDist =Rad1 +Rad2;
 // if it is a hit
 if(dist <=(minDist * minDist))
   return true;
 else
   return false;
}
```

5.4.3.2 CylinderCylinderCollision

Cylinder - Cylinder collision detection takes a similar approach to the one outlined in the Sphere-Sphere collision however only the x and y parameters of the Cylinder need to be taken into account due to the Cylinder's height. The algorithm is outlined in the following code segment

```
bool CylinderCylinderCollision(Point3 Pos1, float rad1, float height1,
                               Point3 Pos2, float rad2, float height2)
{
float dist, len_x, len_y,
real_rad=rad1+rad2;
height1 += Pos1.z;
height2 += Pos2.z;
if ((height1 - height2) < 0 || (height2 - height1) < 0)
    {
     len_x = Pos1.x - Pos2.x;
     len_y = Pos1.y - Pos2.y;
    if (rad1 <= rad2)
       real_rad = rad2 - rad1;
    if (rad2 < rad1)
       real_rad = rad1 - rad2;
    dist = ((len_x * len_x) + (len_y * len_y));
    if (dist <= rad1+rad2)
          return true;
    else return false;
    }
}
```

5.4.4 Environment Collisions

Collisions with Objects within the environment are processed by two built in functions. At present only Planes and Cubes are supported within the environment. However the system has been designed to be extensible and the addition of different types of object is a simple task.

5.4.4.1 Plane Object Collisions

A plane object is defined in the .fl script using the *Plane* command, to determine if the Agent has hit the *Plane* the Agent's bounding sphere is tested against all of the planes in the environment. This is done as follows

Algorithm 2 Sphere Plane Collision

A collision with an infinite plane P_l is calculate by determining the dot product of the Plane's normal $\mathbf{P_N}$ with the Agents Position A_P as shown below
$D = \mathbf{P_N} \bullet \mathbf{A_p}$ next the radius of the Agents bounding sphere is added to D.
If there is a collision the value of D will be <=0.
To determine if the Agent has hit a finite Plane we must now determine if the Agent Position is within the Range of the Plane's extents and return true.

5.4.4.2 Env Object Collisions

EnvObjects as shown in Figure 5.8. are represented as a cubes with a bounding sphere. When the *SphereEnvObjectCollision* Opcode is used the parameters passed to the routine are checked against all of the EnvObjects contained within the environment.

Detection of collision is a two stage process, first the Agent is tested using Sphere-Sphere collision as shown in Section A.3.0.1. to determine if the Object has been hit. Next Sphere-Plane collision as shown in Section 5.4.4.1. is used to determine which face of the object has been hit. At present the Normal of this face is returned to the Agent and collision avoidance is calculated from this value.

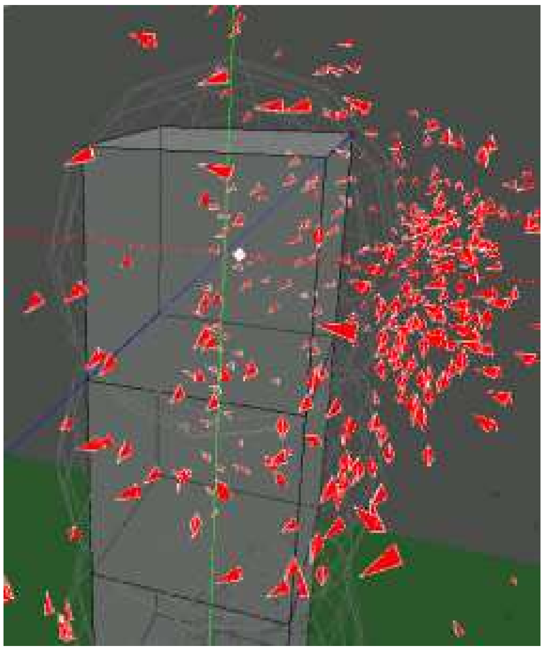


Figure 5.8: Agents avoiding an EnvObj

5.4.5 Drawing

The EBPL DrawFunction is a built in function to allow the rendering of Agents for previewing of the simulation. The system has two main methods for drawing either using a cut-down version of OpenGL or a built in AgentRender system.

5.4.5.1 Mini OpenGL

A small sub-set of OpenGL [Ltd03, MW99]. has been implemented to allow the drawing of simple polygonal shapes to visualize the Agents. At present the mini GL implementation allows for the following commands

- Push / Pop Matrix - store the OpenGL transformation stack.
- Drawing Primitives - Points Lines Quads and Line Loops.
- Colour - set the current drawing colour
- Line / Point size - change the current size of pixels
- Primitives - Sphere, cone and cube
- Affine Transforms - Rotation in X,Y and Z, Scaling and Translation using Point or Vector data types.

Figure 5.9. shows the mini GL in action rendering both the Boid type agents and the ground shadows the code for this example is shown in Algorithm 3.

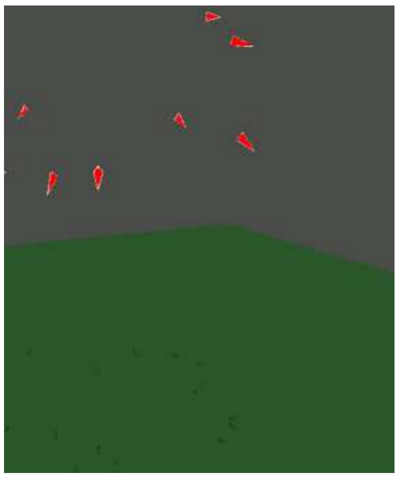


Figure 5.9: Mini GL drawing

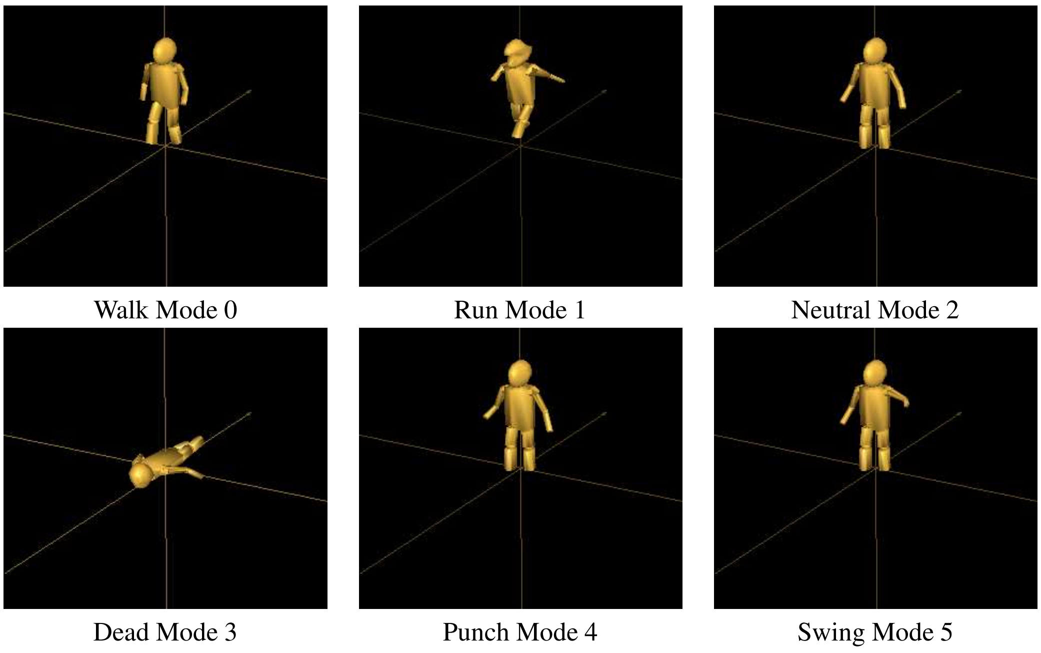


Figure 5.10: AgentRender module with 6 animations cycles

5.4.5.2 AgentRender

The AgentRender module allows for a series of key-framed Alias-wavefront .obj [Ali03] files to be loaded as separate animation sequences which can be triggered within the system.

At present the AgentRender module is hard coded with the six animation cycles shown in Figure 5.10. However in future versions of the system this will be fully scripted within the .fl script system. Each time the AgentRender module is loaded a series of OpenGL display lists are created to speed up the execution of the renderer. If a non-accelerated OpenGL graphics card is used the system will resort back to software rendering which slows down the loading and rendering of Agents.

When the AgentRender module is created the six animation cycles are created by loading in two .obj models

Algorithm 3 MiniGL Drawing Example

```
DrawFunction
 PushMatrix;
 // Translate to the Agents Position
    Translate Pos;
        //Rotate the Agent to the correct orientation
        RotateX  xrot;
        RotateY  yrot;
        RotateZ  zrot;
        //Set the Colour
        Colour 1.0 0.0 0.0;
       //Now draw the agent poly's
      Polygon ;
            Vertexf 0.5 -0.2 -0.2;
            Vertexf 0.5 0.2 0.0;
            Vertexf -0.5 -0.2 0.0;
            Vertexf 0.5 -0.2 0.2;
            Vertexf 0.5 0.2 0.0;
            Vertexf -0.5 -0.2 0.0
       glEnd ;
 // Now draw the Poly's as Lines to give the agent an outline shape
        LineSize 1.0;
        Colour 1.0 1.0 1.0;
        LineLoop;
            Vertexf 0.5 -0.2 -0.2 ;
            Vertexf 0.5 0.2 0.0 ;
            Vertexf -0.5 -0.2 0.0 ;
            Vertexf 0.5 -0.2 0.2 ;
            Vertexf 0.5 0.2 0.0 ;
            Vertexf -0.5 -0.2 0.0 ;
        glEnd;
PopMatrix ;
// Now Draw Shadows
PushMatrix;
       // preserve y value
        Fpush Pos y;
        PushGPYlevel;
        Fpop Pos y;
        Translate Pos;
        RotateX  xrot;
        RotateY  yrot;
        RotateZ  zrot;
        Scale 0.5 0.5 0.5;
        Colour 0.2 0.2 0.2;
        Polygon ;
            Vertexf 0.5 -0.2 -0.2 ;
            Vertexf 0.5 0.2 0.0 ;
            Vertexf -0.5 -0.2 0.0 ;
            Vertexf 0.5 -0.2 0.2 ;
            Vertexf 0.5 0.2 0.0 ;
            Vertexf -0.5 -0.2 0.0 ;
        glEnd;
PopMatrix;
// restore the Y value
fPop Pos y;
End;
```

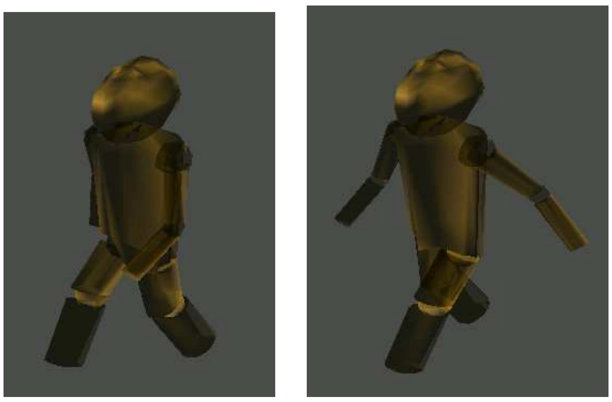
and creating a series of frames by the use of linear interpolation. Each model loaded in must have the same vertex, face and normal layout but the verticies may be in different positions to create the animation.

Figure 5.11 shows the two obj models used for generating the walk cycle. These are loaded as the start and end frames for the animation cycle then the linear interpolation function shown below is used on each Vertex to calculate the in-between frames.

```
glVertex3f( lerp(StartObj.Verts[V].x,EndObj.Verts[V].x,t),
            lerp(StartObj.Verts[V].y,EndObj.Verts[V].y,t),
            lerp(StartObj.Verts[V].z,EndObj.Verts[V].z,t));
```

Where t is the blend function set from 0, the start frame and 1 the end frame. The lerp function used is shown below

```
GLfloat lerp(GLfloat A,GLfloat B, GLfloat t)
{
   GLfloat p;
   p=A+(B-A)*t;
   return p;
}
```



Walk Start Frame Walk End Frame

Figure 5.11: Sample Agent Render model frames

Chapter 6

Conclusions and Future work

The development of the EBPL language and development environment has been very successful. Producing simple flocks is a very trivial task as is the use of ground based interactive Agents. The system has been used to produce a number of simple crowd animations as well as more complex interactions of Agents.

Unfortunately no real evaluation of the use of this system has been carried out except for the use of the system by the Author. Ideally the system and programming language needs to be beta tested by animators with specific animation goals in mind to determine the usability and ease of programming. A series of animations and example programs have been developed and these are available in the EBPL language reference and User Guide document. This document also explains in more detail the algorithms used in the development of flocks and complex Agent interactions.

The system has been developed under RedHat Linux 9.0 [Lin03] using Gnu g++ [gg03]. At present the system works on Unix platforms and versions are available for Linux, Solaris and Irix. There have been several problems with the porting the system to Microsoft Windows with Visual Studio .net not allowing several of the key classes to compile.

Also porting to Windows using the Cygwin [Lin03] version of g++ has also caused problems due to incompatibilities with the X.P. file-system. These problems are due to the Unix *\n* and the windows *\n\r* file end tags and can be resolved with a major re-write of the parser / compiler but this was considered too time consuming for the present phase of the project.

6.1 Future work

6.1.1 Multi Emitter interactions

At present the system only allows interactions with Agents from the same emitter using the same brain script. This is can be very limiting for complex animations an needs to be addressed in new versions of the system. The simplest way to implement this will be adding the name of the Agent brain script to use to the LoadARF script. This will allow different agents to respond differently within the system. To further simplify the layout and generation of the Agents a GUI based system could be used to place and initialize each of the Agents for the simulation.

6.1.2 EBPL language

At present only a small subset of the EBPL language has been developed as "proof of concept" a full implementation of the language will require more work.

At the time of writing only the floating point stack element is fully functional with opcode access, the Point, Vector and fuzzy stacks are implemented in the system but only a few opcodes have been developed. As most of the compiler structure is in place this will not require much development work.

Another control structure to be added to the system are simple loops, these would allow for the Agents to run segments of code multiple time which will be useful for more complex collision detection routines.

6.1.3 AgentRender

The AgentRender system at present works with a hard coded series of animations, this needs to be fully scriptable within the .fl file system. At present only .obj files can be loaded but using either the XSI API or Maya API different file formats can be used including the use of inverse kinematics and animation sequences.

6.1.4 Animation Curve Output

Integration with other animation applications is achieved using the outfile system where the Agents position for every frame is save as well as the rotations. This must then be interpreted and loaded into a 3rd party animation package using a script.

This method produces very large and time consuming output and a more suitable format is required. The simplest method would be to save the Agent's movement by the use of Animation curves which could then be imported into the animation system similar to that used by Behaviour.

6.1.5 Visual Programming environment

The ultimate goal of the system is the development of a fully integrated development environment with visual development of the Agents brain. This would require a higher level interpretor to allow the generation of EBPL programs from the visual system.

A simpler visual system to develop would be the Environment layout and agent placement. This would allow the Emitters, any Environment objects and the terrain to be placed interactively by the animator and the results saved to a .fl script.

Appendix A

Language Reference

The following Chapter describes the syntax of the EBPL programming language.

A.1 Variable syntax

A.1.1 Float

A floating point variable may be defined as shown below, it must be defined in the script before it used.

```
    float xrot=0.0;
```

A.1.2 Point

The *Point* data type is used to represent a 3D point for drawing, it must be defined in the script before it is used. The point is defined as a 3 tuple with *x*,*y* and *z* components

```
    Point Pos=[0,0,0];
```

A.1.3 Vector

The *Vector* data type is used to represent a 4D mathematical vector, it must be defined in the script before it is used. The point is defined as a 4 tuple with *x*,*y*,*z* and *w* components

```
    Vector Dir=[0.0,-0.2,-0.1,0.0];
```

A.1.4 Bool

The *Bool* data type is used to represent **true** and **false** values, it must be defined in the script before it is used.

```
    bool HitAgent=false;
```

A.1.5 Fuzzy

The *Fuzzy* data type represents a class of fuzzy object. It is defined as a floating point value (usually clamped between 0.0 - 1.0) and is capable of having fuzzy set operations applied to it, it must be defined in the script before it is used.

```
      Fuzzy CloseEnough=0.03;
```

A.2 Agent Global Variables

There are five built in Agent Global variables these may be used to either set or get values from the Agent Class.

A.2.1 SetGlobalPos GetGlobalPos

The global position opcodes are used to set and get the global position of the Agent. The position value is used to calculate the flock center (Centroid) and if this is to be used in the simulation the GlobalPos variable must be set. It is used in the following way :-

```
      // set the global pos flag
      Point Pos=[0,0,0];
      GetGlobalPos Pos;
      // get the global pos flag
      SetGlobalPos Pos;
```

A.2.2 SetGlobalDir GetGlobalDir

The global dir opcodes are used to set and get the global direction of the Agent. The direction value is not used within the simulation system, but the Dir value is from the arf file so getting this value can be used to configure the initial Agent direction.

```
      // get a direction value
      Vector Dir=[0,0,0,0];
      GetGlobalDir Dir;
```

A.2.3 SetGlobalCentroid GetGlobalCentroid

The global Centroid opcodes are used to set and get the global Centroid of the Agent. The Centroid is calculated every frame by finding the average position of all the Agents. If the Centroid is required the user can access the flock center by these opcodes

```
      // get the flock center
      Point Centroid=[0,0,0];
      GetGlobalCentroid Centroid;
```

A.2.4 SetGlobalCollideFlag GetGlobalCollideFlag

The global collide flags are used to indicate if an Agent has already been hit in a collision detection routine. If the global collide flag is set the collision detection will not be executed for the Agent.

```
// get the flock center
Bool TRUE=true;
// set glob collide flag
SetGlobalCollideFlag TRUE;
```

A.2.5 SetGPYlevel

The setgpylevel opcode is used to query the groundplane and set the y value of a tuple data type.

```
// get the flock center
Point Pos=[0,0,0];
// set the y value based on the gp from the agents position
GetGPYLevel Pos;
```

A.2.6 PushGPYLevel

This opcode pushes the y value of the groundplane onto the float stack. No variables are required for this function.

```
PushGPYLevel;
```

A.3 Collisions

Each time the system is updated the brain's CollideFunction is called, this is the only time within an EBPL program that the An Agent can access another Agents data. Within the system all agents are contained within a number of bins It is possible to loop through the Agents within the bins and test for collisions. This is implemented using the *LoopBin* and *LoobBinEnd* EBPL opcodes.

Whenever the *LoopBin* structure is encountered the current Agent value is used and the rest of the Agents within the bin are compared with the current Agent.

For convenience and speed EBPL has two built-in collision detection routines, however these and others may also be implemented within the EBPL language itself. These routines take as parameters variables from the EBPL script.

A.3.0.1 SphereSphereCollision

The SphereSphere collision opcode takes five parameters as outlined below

```
SphereSphereCollision bool Hit Point3 Pos1 float Rad1
                               Point3 Pos2 float Rad2 ;
```

The collision detection is calculated using the Position of the Current Agent and any other Agents in the current lattice bin. Each Agent has a Radius for the bounding sphere and if the sphere collide the Hit flag will be set to true.

A.3.0.2 CylinderCylinderCollision

The CylinderCylinder collision opcode takes seven parameters as outlined below

```
    CylinderCylinderCollision bool Hit Point3 Pos1 float Rad1 float Height1
                              Point3 Pos2 float Rad2 float Height2;
```

The collision detection is calculated using the Position of the Current Agent and any other Agents in the current lattice bin. Each Agent has a Radius and height for the bounding cylinder and if the cylinders collide the Hit flag will be set to true.

A.3.1 Environment Collisions

Collisions with Objects within the environment are processed by two built in functions. At present only Planes and Cubes are supported but the environment has been designed to be extensible and add more object types including .obj objects.

A.3.1.1 Env Object Collisions

EnvObjects are represented as a cubes with a bounding sphere. When the *SphereEnvObjectCollision* Opcode is used the parameters passed to the routine are checked against all of the EnvObjects contained within the environment.

Detection of collision is a two stage process, first the Agent is tested using Sphere-Sphere to determine if the Object has been hit. Next Sphere Plane collision is used to determine which face of the object has been hit. At present the Normal of this face is returned to the Agent and collision avoidance is calculated on this value.

The SphereEnvObjCollision opcode takes four parameters as outlined below

```
    SphereEnvObjCollision bool Hit Point3 Pos
                          float Radius2 Vector Normal;
```

The collision detection is calculated using the Position of the Current Agent and its bounding sphere radius, each EnvObject is tested in turn against the Agent and if a hit is detected the Hit flag is set to true and the Normal to the face hit is returned in the Normal parameter.

A.4 Functions

EBPL has two types of functions, there are 4 default built in functions which must be present in every EBPL script and user defined functions which act as procedure calls.

The built in functions are shown in Table A.1 and are called for every Agent in the system for each iteration of the environment. User defined functions are generated by the *Function* keyword and can be called from within any of the main built in functions.

Built In Functions	Description
InitFunction	Called when the Agent brain is created and used to initialize variables etc.
UpdateFunction	Called every iteration of the system to update the Agents position
DrawFunction	Called every iteration of the system to Draw the Agent
CollideFunction	Called every iteration to do collision detection

Table A.1: EBPL variable types

A.5 Call Lists

Call lists are a way of creating switchable function calls depending upon an ordinal variable value. The creating of a call list is a two stage process as shown below

```
DefineCallList TestList;
CallListItem TestList foo;
CallListItem TestList bar;
```

First a call list name is defined to make storage for the Function pointers to be allocated to the list. Next list items may be added to the list. To use a CallList within a function the following code is used

```
float ListValue=0;
// function foo called
CallList TestList ListValue;
AddD ListValue 1;
// Function bar called
CallList TestList ListValue;
```

A.6 Bins and Other Agent Access

The only time other Agents value may be accessed is within the LoopBin structure. There are two methods of accessing the data of other agent by use of the *SetAgentI* and *GetAgentI* methods as follows

```
// Set NDir to be the ADir of the other Agent
SetAgentI NDir ADir;
// Get the Dir of AgentI and set it to NDir
GetAgentI Dir Ndir
```

A.7 MiniGL Opcodes

A simple subset of the OpenGL drawing commands have been implemented in the script to allow the Draw function to produce images. These allow for simple line, point and 3D primitives to be drawn and are intended as a simple method to visualize the Agents behaviour.

A.7.1 RotateX

The RotateX opcode calls *glRotatef(d,1,0,0);* and rotates the current transformation matrix by degrees in the X axis. It takes as its argument either a direct floating point value or a Floating point variable.

```
RotateX xrot; // using a variable
RotateX  -90.0; //using a direct value
```

A.7.2 RotateY

The RotateY opcode calls *glRotatef(d,0,1,0);* and rotates the current transformation matrix by degrees in the Y axis. It takes as its argument either a direct floating point value or a Floating point variable.

```
RotateY yrot; // using a variable
RotateY  -90.0; //using a direct value
```

A.7.3 RotateZ

The RotateZ opcode calls *glRotatef(d,0,0,1);* and rotates the current transformation matrix by degrees in the Z axis. It takes as its argument either a direct floating point value or a Floating point variable.

```
RotateZ zrot; // using a variable
RotateZ  -90.0; //using a direct value
```

A.7.4 PushMatrix

The PushMatrix opcode calls *glPushMatrix();* to store the state of the current OpenGL transformation matrix.

A.7.5 PopMatrix

The PopMatrix opcode calls *glPopMatrix();* to re-store the state of the current OpenGL transformation matrix.

A.7.6 Translate

Translate calls *glTranslate3f(x,y,z)* where the x,y and z values come from either a Point or Vector data type. It is used as shown below

```
Translate Pos; // where Pos is a point
Translate Dir; // where dir is a vector
```

A.7.7 Polygon

The Polygon opcode calls *glBegin(GL_POLYGON);* to start drawing with polygons

A.7.8 Quad

The Quad opcode calls *glBegin(GL_QUADS);* to start drawing with quads.

A.7.9 Point

The Point opcode calls *glBegin(GL_POINTS);* to start drawing with points.

A.7.10 LineLoop

The LineLoop opcode calls *glBegin(GL_LINE_LOOP);* to start drawing a line loop

A.7.11 glEnd

The glEnd Opcode calls *glEnd()* to end the current drawing mode

A.7.12 Vertex

The vertex Opcode takes either a point or a vector data type and uses the x,y and z values to produce a vertex using *glVertex3f(x,y,z);*

```
Vertex Pos; // draw a vertex using a point
Vertex Dir; // draw a vertex using a vector
```

A.7.13 Vertexf

The vertexf opcode draws a vertex using direct floating point values for the x,y and z values as shown below

```
Vertexf -0.5 -0.2 0.0;
```

A.7.14 PointSize

The PointSize opcode sets the current point drawing size

```
PointSize 2.0; //set the point size to 2.0
```

A.7.15 LineSize

The LineSize opcode sets the current line drawing width

```
LineSize 4.0; // set the line width to 4.0
```

A.7.16 Sphere

The Sphere opcode draws a sphere using either a floating point variable or direct value for the radius of the sphere, the other two arguments are stacks and slices which can not be changed once set.

```
Sphere radius 12 12 ; // sphere using a variable value
Sphere 0.02 20 20 ; // sphere using direct float values
```

A solid version of the sphere may also be drawn using the *SolidSphere* opcode as follows

```
SolidSphere radius 12 12 ; // sphere using a variable value
SolidSphere 0.02 20 20 ; // sphere using direct float values
```

A.7.17 Cylinder

The Cylinder opcode draws a cylinder using either floating point variable of direct value for the radius and height of the cylinder, the other two arguments are stacks and slices which can not be changed once set.

```
      Cylinder Rad Height 12 14; // using variables
      Cylinder 0.4 1.0 20 20 ; // using direct values
```

A.7.18 Colour

The colour opcode sets the current drawing colour, it takes 3 floating point values for red, green and blue or a Point variable type

```
      Colour 0.3 0.2 1.0 ;
      Point colour=[0,1,0]
      Colour colour;
```

A.7.19 Lighting

At present only the *GL_LIGHT0* light is enabled in the system. This has a default position of 0,0,0 and a default colour of white.

To enable and disable the lighting in the system the following opcodes are used.

```
      // turn lights on
      EnableLights;
      // turn lights off
      DisableLights;
```

The shade models used for rendering may also be set. The default value is smooth shading which is slower. To change the shade model use the following opcodes

```
      // turn on smooth shading (slower)
      Smooth;
      // turn on flat shading (faster)
      Flat;
```

A.8 AgentRender

To use the AgentRender module both the .fl and brainscript files need to have the following instruction added

```
      UseAgentRender ;
```

This tells the environment and the brain that the AgentRender is being used. If this is not present in the scripts and any other AgentRender calls are made the system will crash. In the brain script this call is generally placed in the *InitFunction* as it only needs to be set once.

There are four Opcodes used for configuring the AgentRender as follows

All these opcodes can take either floating point variables or direct float values as shown in the example below :-

Value	Meaning
SetAnimCycle	sets the animation cycle to be drawn as indicated by the Mode values in Figure 5.10
RenderFrame	Selects which frame of the AgentRender sequence to draw
RenderAgent	Draws the Agent configured by the above two opcodes
RenderMaterial	Set the current material for the Agent (see Table A.3)

Table A.2: AgentRender Opcodes

```
    // set to agent walk cycle
    float DrawMode=0;
    float Frame=4;
    SetAnimCycle DrawMode;
    // set material to chrome
    RenderMaterial 3;
    //render the 3rd frame of the cycle
    RenderFrame Frame;
```

The material types currently used in EBPL are shown in Table A.3.

Material Number	Material Type
0	BLACKPLASTIC
1	BRASS
2	BRONZE
3	CHROME
4	COPPER
5	GOLD
6	PEWTER
7	SILVER
8	POLISHEDSILVER

Table A.3: Material Types

A.9 Float Stack Opcodes

The floating point stack operates in a First in Last Out principle, any operations on the stack use reverse polish notation for example 2+3 will be executed in the script by push 3 push 2 fadd which will result in 5 being put onto the top of the stack.

A.9.1 Fpush

The fpush opcode pushes a floating value onto the stack any variable type may be pushed onto the stack but with tuple data types which element to be pushed must be specified.

```
    Fpush Pos x; // pushes the x component of the Point pos
    Fpush yrot ; // pushes the float variable yrot;
```

A.9.2 Fpushd

The *fpushd* opcode allows a direct floating point value onto the stack as follows

```
    Fpushd 23.2; // pushes 23.2 onto the float stack
```

A.10 Fpop

The fpop opcode takes the value from the top of the stack and places it into the variable, if a tuple data type is used the x,y or z destination must be specified.

```
    fpop ypos; // places the tos value into ypos
    fpop tempPos y; places the tos value into tempPos.y
```

A.10.1 Fadd

The fadd opcode takes the top two values from the stack adds them and places the sum back onto the stack

A.10.2 Fsub

The fsub opcode takes the top two value from the stack subtracts them and places the result back on the stack. Note if the stack contains 2 and 4 the result will be 2 - 4 = -2

A.10.3 Fmul

The fmul opcode takes the two top values from the stack and multiplies them and places the product back on the stack.

A.10.4 Fdiv

The fdiv opcode takes the two top values from the stack and divides them and places the result back on the stack. Division by 0 is trapped by changing any 0 value with a 1 (and printing a warning to the console in debug mode)

A.10.5 Fdup

The fdup opcode takes the top of stack value and duplicates it.

A.10.6 Fsqrt

The Fsqrt opcode takes the top of stack value and places the square root of that value onto the stack.

A.10.7 Frad2Deg

The Frad2deg opcode is a helper function to convert radians to degrees as most C++ math operations use radians.

A.10.8 Fatan

The fatan opcode takes the two top values from the stack and calculates the arc tangent. The code used is atan2(x,y) where X is the top of stack value and y is the next value

A.10.9 Fsin

The fsin opcode takes the top most value from the stack and puts back the sine of the value

A.10.10 Fasin

The fasin opcode takes the top most value from the stack and puts back the arc sine of the value

A.10.11 Fcos

The fcos opcode takes the top most value from the stack and puts back the cosine of the value

A.10.12 Facos

The facos opcode takes the top most value from the stack and puts back the arc cosine of the value

A.10.13 Fnegate

The Fnegate opcode takes the top most value from the stack and puts back the negated value.

A.10.14 FStackTrace

The FstackTrace opcode prints out the current contents of the stack.

A.11 Variable Opcodes

Most variables can be accessed directly by the use of opcodes and values can be set and retrieved.

A.11.1 Add

The *Add* opcode works on any data type but care must be taken with the mixing of types. For example tuple data types can be added together easily but a tuple to float will cause problems. The opcode is used as follows

```
    Add Pos Dir; // add a point to a vector
    Add yrot zrot; // add two float variables
```

A.11.2 Sub

The *Sub* opcode works on any data type but care must be taken with the mixing of types. For example tuple data types can be subtracted easily but a tuple to float will cause problems. The opcode is used as follows

```
Sub Pos Dir; // subtract Dir from Pos
Sub yrot zrot; // subtract two float variables
```

A.11.3 Mul

The *Mul* opcode works on any data type but care must be taken with the mixing of types. For example tuple data types can be multiplied easily but a tuple to float will cause problems. The opcode is used as follows

```
Mul Pos Dir; // multiply Pos by Dir
Mul yrot zrot; // multiply two float variables
```

A.11.4 Div

The *Div* opcode works on any data type but care must be taken with the mixing of types. For example tuple data types can be subtracted easily but a tuple to float will cause problems. The system automatically traps division by zero errors by setting the divisor to 1 if it is a zero value. The opcode is used as follows

```
Div Pos Dir; // divide Dir from Pos
Div yrot zrot; // divide two float variables
```

A.11.5 Set

The *Set* opcode is used to assign one variable from another it may only be used with variable of the same type as follows

```
Point Pos=[1,0,1];
Point Pos2=[0,0,0]
Set Pos Pos2;  // Set Pos = Pos2;
```

A.11.6 AddD

The AddD opcode adds a value directly to a variable for example

```
// using a float
AddD yrot 180.0;
// using a point
AddD Pos 0 1 0;
```

A.11.7 SubD

The SubD opcode subtracts a value directly from a variable for example

```
      // using a float
      SubD yrot 180.0;
      // using a point
      SubD Pos 0 1 0;
```

A.11.8 MulD

The MulD opcode multiplies a value directly from a variable for example

```
      // using a float
      MulD yrot 180.0;
      // using a point
      MulD Pos 0 1 0;
```

A.11.9 DivD

The DivD opcode divides a value directly from a variable. If the divisor is zero the value will be set to 1 for example

```
      // using a float
      DivD yrot 180.0;
      // using a point
      DivD Pos 0 1 0;
```

A.11.10 SetD

The SetD opcode assigns a direct value to a variable as follows

```
      // using a float
      SetD yrot 180.0;
      // using a point
      SetD Pos 0 -1 0;
```

A.11.11 Length

The Length opcode returns the length of a Vector or Point Variable onto the stack

```
      Vector Dir=[1,0,-2,0];
      Length Dir;
```

A.11.12 Normalize

The Normalize opcode set the variable to unit size

```
      Vector Dir=[1,0,-2,0];
      Normalize Dir;
```

A.11.13 Dot

The *Dot* opcode returns the dot product of two Point or Vector data types, these types may be mixed but the return type is always a float. It is used as follows

```
Float DotResult=0.0;
Vector Dir=[1,0,-2,0];
Vector Dir2=[1,0,1,0];
// return the result DotResult = Dir • Dir2
Dot DotResult Dir Dir2;
```

A.11.14 Reverse

The Reverse opcode reverses (negates) the current variable.

```
Vector Dir=[1,0,-2,0];
Reverse Dir;
```

A.11.15 Randomize

The randomize opcode sets the variable to random values within a range based on ±seed value passed.

```
Randomize Dir 2.1 0.1 1.1;
Randomize yrot 4.0;
```

A.11.16 RandomizePos

The randomizepos opcode sets the variable to random positive value within a range based on zero to the seed value passed.

```
RandomizePos Dir 2.1 0.1 1.1;
RandomizePos yrot 4.0;
```

A.12 Structure Opcodes

The structure opcodes are used to define functions and call functions and make decisions.

A.12.1 Call

The Call opcode will call a function within the script.

```
Call CalcAngle;
```

A.12.2 Function

The Function Opcode defines the beginning of a function block. It must be terminated by use of an End opcode as shown below

```
Function CalcAngle
 fpush yrot;
 fpop yrot;
End
```

A.12.3 If

The if structure can be used on all data types and is constructed as shown below

```
if dist > MinCentroidDist
 {
  // do something
 }
```

All if statements must use the open and closed braces, however if statements may be nested. At present if only works with two defined variable types so if a comparison against a static value is required the static value must be define as a variable.

A.12.4 Ifelse

The ifelse structure can be used on all data types and is constructed as shown below

```
ifelse dist > MinCentroidDist
 {
  // do something
 }
 {
 // do something else
 }
```

All if statements must use the open and closed braces and at present ifelse statements may not be nested, however if statements may be placed within an ifelse construct. At present ifelse only works with two defined variable types so if a comparison against a static value is required the static value must be define as a variable.

A.13 Debug

The Debug opcode prints to the console the variable argument.

```
Debug Pos;
```

The *DebugOpOn* and *DebugOpOff* opcodes turn on and off the printing of the current opcode to the console.

A.14 Noise Function

Each Agent has the ability to access a noise function based on Perlin Noise [eta98]. This value is generated by a built in noise function and can be calculated from any tuple data type. There are five built in noise functions as shown in Table A.4.

Noise Type	Noise Index Value
turbulence	0
marble	1
undulate	2
noise3	3
turbulence2	4

Table A.4: ebpl Noise functions

To use noise the noise generator must be initialized as shown below

```
float NoiseType=5.0;
float NoiseScale=0.0;
RandomizePos NoiseType 5;
Randomize NoiseScale 2000;
UseNoise NoiseType NoiseScale;
```

The *GetNoiseValue* opcode is used to access the noise table, it is passed a value to return to and a seed value to generate the noise from this is shown in the following example

```
Point Pos=[0,0,0];
Vector NoiseValue=[0,0,0,0];
GetNoiseValue NoiseValue Pos;
```

A.15 CalcAngle Function

The following C++ code is used in the original flocking program to calculate the agents y and z rotation.

```
void Agent::CalcAngle(void)
{
GLfloat X,Y,Z;
X=NextPos.x-Pos.x;
Z=NextPos.z-Pos.z;
Y=Pos.y-NextPos.y;
//using spherical geometry we calculate the rotation based on the New Point
yrot=atan2(Z,X); // Now convert from radians to deg
yrot=-((yrot/TWO_PI)*360.0f); //now align the geometry to our world
yrot+=180;
// Now for the zrot
GLfloat r=sqrt(X*X+Y*Y+Z*Z);
zrot=asin(Y/r);
//convert from radians to deg
zrot=((zrot/TWO_PI)*360.0f);
}
```

The following code does the same using the Brain script language

```
Vector tmpPos=[0,0,0,0];
Vector tmpPos2=[0,0,0,0];

Function CalcAngle;
 // fist set the vectors to subtract
 Set tmpPos NextPos ;
 Sub tmpPos Pos;
 Fpush tmpPos x ;
 Fpush tmpPos z ;
 // use atan2 to calc the angle
 fatan ;
 // this gives us the angle in radians so we convert it
 Frad2deg ;
 // finally we align it with the world geometry
 Fnegate ;
 Fpop yrot ;
 AddD yrot 180.0 ;
 // now we align the z rotation
 //Y=Pos.y-NextPos.y;
 Set tmpPos NextPos ;
 Set tmpPos2 Pos ;
 // swap the y components
 Fpush Pos y ;
 Fpop tmpPos y;
 Fpush NextPos y ;
 Fpop tmpPos2 y ;
 // now subtract the two
 Sub tmpPos tmpPos2 ;
 //zrot=asin(Y/r);
 Length tmpPos ;
 Fpush tmpPos y ;
 Fdiv ;
 fasin ;
 // convert it to degrees
 frad2deg;
 Fpop zrot;
End;
```

Appendix B

Sample EBPL Scripts

```
// the Particles Position
Point Pos=[0,0,0];
// Particles Direction
Vector Dir=[0.0,0.0,0.0,0.0];
// The Particles Next Position
Vector NextPos=[0.0,0.0,0.0,0.0];
// The Lifespan of the particle
float Life=0.0;
// When the particle is to Die
float EndLife=10.0;
// Default init function for the Agent Brain
InitFunction
Call InitParticle ;
End;
// Initialize the particle
Function InitParticle ;
// Create a random direction
Randomize Dir 1.0 1.0 1.0;
// Set the particles life value to 0
SetD Life 0.0;
// Set the initial position of the particle to 0,0,0
SetD Pos 0.0 0.0 0.0;
SetD NextPos 0.0 0.0 0.0;
// Randomize the particles life span
Randomize EndLife 20;
End;
// just draw the particle as a line from current pos to next pos
DrawFunction
PushMatrix;
// Translate to the Agents Position
Translate Pos;
Colour 1.0 1.0 1.0;
LINES ;
Vertex Pos;
Vertex NextPos;
glEnd;
PopMatrix;
End;
// update the particles position based on the Dir vector
UpdateFunction
Add Pos Dir;
Set NextPos Pos;
Add NextPos Dir;
AddD Life 1.0;
// if we have reached the end of the particle's life reset it
if Life >= EndLife
{
Call InitParticle;
}
End;
// No collision detection
CollideFunction
End ;
```

Appendix C

.fl Script reference

To run the program type in the console ebpl <paramfile.fl> where <paramfile.fl> is the name of the file to load.

Once the program runs it is best to leave the agents moving in random for a while so the system can reach a suitable chaotic state.

To run full screen use ebpl <paramfile.fl> f

C.0.1 Keys

[space]	turns the flocking on and off
c	toggles the curve following for the agents
p	Pause simulation
d	Dump agent debug info to console
h	Toggle on screen menu
Mouse	rotates view Left Button x-y Right button x-z
0[zero]	toggle write agent points to file mode
ArrowKeys	Move Camera
PGUP/PGDN	Zoom Z in and out.
1	Toggle Camera follow Centroid
g	Toggle Agent Follow goals mode
p	Pause Simulation
l	Toggle Locked Centroid mode
4	Draw bin lattice structures
q	Toggle draw Environment details
t	Toggle write frames as tiff files (will be placed in the directory ../Frames/

C.1 Flocking System Script File Format

The flocking system reads a file to configure the various elements of the system. Two initial parameters must be set and after that the parameters may be set in any sequence.

All variables are assumed floats *(f)* except index values which are integer values *(i)* and the file name for the Output file *(s)*. Comments can be added to the file using the standard C comment block //. also white space on lines is also ignored.

All keywords are case insensitive and capitalizations are only used for ease of reading the script.

C.1.1 Environment Parameters

The Environment parameters are used to set up the initial parameters for the world.

C.1.1.1 BBox

The first script element of the file must be a BBox which sets up the world bounding box.

```
WorldBBox (f)Pos.x (f)Pos.y (f)Pos.z (f)Width (f)Height
(f)Depth (i)Xdiv (i)Ydiv (i)Zdiv (i)BinSize
```

Pos : sets the initial position of the bounding box center

Width, **Height Depth** : the extents of the box calculated from the center (as $\pm$Width/2 etc)

Xdiv,Ydiv,Zdiv : The subdivision of the Lattice bin structure for agent containments

BinSize : the max size of bins, this is usually set to be the number of Agents in the system

C.1.1.2 EnvObj

Any number of EnvObj can be added to the Environment, they are added sequentially from 0 and any subsequent operations of these objects refer to the index.

```
EnvObj (f)Pos.x (f)Pos.y (f)Pos.z (f)Width (f)Height
(f)Depth (f)Radius
```

Pos : sets the central position of the Object

Width Height Depth : set the extents of the Object

Radius : the radius of the object used for collision detection, it is best to set it slightly larger than the biggest extent of the object.

C.1.1.3 RotateObj

This is used to rotate an object around it's own axis in x,y or z.

```
RotateObj (i)Index (f)angle (f)xAxis (f)yAxis (f)zAxis
```

Index : the index of the object to rotate, this is based on the sequence that the objects are added in the script file starting at 0

angle : the angle to rotate in degrees

xAxis,yAxis zAxis : are flags to indicate which axis to rotate around. setting any of these to 1.0 will rotate around the axis specified, all can be set in one go i.e *RotateObj 0 25 1 1 1* will rotate 25°in all 3 axis.

C.1.1.4 Goal

Goals are timer based objects which can attract the flock. Any number can be added to the AgentEmitter they are drawn in Red in the display

```
Goal (i)index (f)Pos.x (f)Pos.y (f)Pos.z (f)TimeToActivate
```

Index : which AgentEmitter to attach the goal to

Pos : sets the central position of the goal

TimeToActivate : The time when the goal is active.

C.1.1.5 OutputFile

This sets the name for the output file to save Agent data to

```
OutputFile (s)Filename
```

Filename : file to save to

C.1.1.6 OutFileFrameSkip

The set the number of frames to skip between writes to the file. This value defaults to 5.

```
OutFileFrameSkip (i)skip
```

skip : the frame skip rate.

C.1.1.7 UpdateRate

Set how many times the simulation is update per redraw. This value will run the agent update but not display the results until all the updates have been done.

```
UpdateRate (i)rate
```

rate : how many times to update per display update

C.1.1.8 RandomSeed

The tells the simulation to set the seed to a random value (using *srand(time(NULL));*) to generate random values for the simulation, otherwise srand(2) is used to give the same values each time.

C.1.1.9 Camera

Allows the user to override the camera

```
Camera (f)eyeX (f)eyeY (f)eyeZ (f)lookX (f)lookY (f)lookZ
(f)upX (f)upY (f)upZ (i)W (i)H (f)va (f)asp (f)near
(f)far
```

eyeX,eyeY,eyeZ : eye Position

lookX,lookY,lookZ : look at point

upX,upY,upZ :a vector to indicate which axis is the up direction

W,H : the width and height of the screen area

va : the view angle

asp : the aspect ration

near,far : the near far clip planes

C.1.1.10 CamFollowCentroid

This command allows the Camera to be attached to one of the AgentEmitters centroids.

```
CamFollowCentroid (i)index (f)Xoff (f)Yoff (f)Zoff
```

Index : the index of the AgentEmitter for the camera to attach the look point to

Xoff : the X offset from the centroid for the Eye point of the camera

Yoff : the Y offset from the centroid for the Eye point of the camera

Zoff : the Z offset from the centroid for the Eye point of the camera

C.1.1.11 FrameOffset

This sets the frame offset for the output file, by default frames start at 0

```
FrameOffset (i)offset
```

offset : the offset added to the frame numbers in the output file.

C.1.1.12 RandomSeed

Sets the seed for the random number generator to the value time(NULL) which is the current system time. In theory this should make each run of the system different but this will change depending upon the random number generator used.

C.1.1.13 VectObj

Vector objects are objects in the environment which act like winds. Each vector Obj has a bounding sphere which will become active if the Agents collide with them.

```
VectObj (f)Pos.x (f)Pos.y (f)Pos.z (f)Width (f)Height (f)Depth
(f)Radius (f)X (f)Y (f)Z
```

Pos : the position of the Vector Object

Width,Height,Depth : The width height and depth of the object

Radius : The radius of the Objects bounding sphere for agent collision detection

X,Y,Z : The vector direction for the Vector object.

C.1.1.14 GroundPlane

The ground plane is the default ground level for the Agents in the Environment

```
GroundPlane (f)Level (f)Red (f)Green (f)Blue (f)Alpha
```

Level : The ground plane level

Red : The red colour component for the gp

Green : The green colour component for the gp

Blue : The blue colour component for the gp

Alpha : The transparency for the gp.

C.1.1.15 ImageGroundPlane

This allows the groundplane to be loaded from an image file. The extents of the ground plane are calculated from the WorldBBox size and the number of triangle strips used are dependent upon the size the image loaded.

```
ImageGroundPlane (f)Ylevel (s)Filename.bmp (f)Ydivisor
```

Ylevel : The level of the groundplane

Filename.bmp The name of the bmp file to load

Ydivisor : The value to scale the heightmap by.

C.1.1.16 GroundPlaneTex

This allows the groundplane to be loaded from an image file. The extents of the ground plane are calculated from the WorldBBox size and the number of triangle strips used are dependent upon the size the image loaded. The second filename is the name of the texture to use.

```
    GroundPlaneTex  (f)Ylevel  (s)Filename.bmp (s) TextureName.bmp
                    (f)Ydivisor
```

Ylevel : The level of the groundplane

Filename.bmp The name of the bmp file to load

TextureName.bmp : The name of the file to use as a texture for the groundplane.

Ydivisor : The value to scale the heightmap by.

C.1.1.17 ObjTerrain

This allows the groundplane to be loaded from an obj file with textured support from a bitmap file

```
    ObjTerrain  (f)Ylevel  (s)Filename.obj (s) TextureName.bmp
                    (f)AgentHeightOffset (f)Scale X (f) ScaleY
                    (f) ScaleZ
```

Ylevel : The level of the groundplane

Filename.obj The name of the obj file to load

TextureName.bmp : The name of the file to use as a texture for the groundplane.

AgentHeightOffset : The value to add to the Agents Y value

Scale X,Y,Z : The scale in X,Y and Z for the obj file. This pre-scales the obj file before loading.

C.1.2 Emitters

The Environment can have any number of AgentEmitter. These are the source for each of the agents and are responsible for all the agent collision detection , updates and rendering.

There must be at least one AgentEmitter in the file and this is set to index 0, each subsequent one is then incremented by one. These numeric values are then referred to by the other elements of the script to attach something to the Emitter .

C.1.2.1 AgentEmitter

The simplest emitter is shown below and is used for most systems

```
    AgentEmitter  (f)Pos.x  (f)Pos.y  (f)Pos.z  (i)NumAgents
                  (f)Width  (f)Height  (f)Depth  (f)Radius  (i)SpeciesTag
                  (b)EmitType  (s) BrainFile
```

Pos : sets the initial position of the emitter.

NumAgents : the number of Agents to Emit.

Width, **Height** ,**Depth** set the size of the agents

Radius : the bounding sphere radius of the Agent used for collision detection

SpeciesTag : indicates which species the Agents are.

EmitType: set to 0 for point emitter 1 for random distribution around the world BBox

BrainFile : The name of the compiled brain for the Agent to use.

C.1.2.2 PathFollow

This is used to set a spline curve for the agents to follow at present this is a 4 point Bezier curve.

```
PathFollow (i)Index (f)P1.x (f)P1.y (f)P1.z (f)P2.x (f)P2.y
(f)P2.z (f)P3.x (f)P3.y (f)P3.z (f)P4.x (f)P4.y (f)P4.z
```

Index : which AgentEmitter to attach the curve to

P[n].x P[n].y P[n].z : the points for the curve.

C.1.2.3 PathStep

This sets the speed at which the Centroid path step is updated each time default value 0.02

```
PathStep (f)steprate
```

steprate : The update speed for the centroid

C.1.2.4 LoadARF

This allows for the loading of an AgentResoure file generated from the Layout program.

```
LoadARF (i)Index (s)Filename
```

Index the index of the AgentEmitter to load

Filename The name of the ARF file to load.

Appendix D

BinLattice Speed Evaluation

In his paper “Interactions with Groups of Autonomous Characters” [Rey00] Reynolds reports that the bin lattice data structure when used with 1000 Agents “*gives a 16 times faster performance than the naive $O(n^2)$ implementation*”. To determine the speed increase with the developed system a series of test were carried out using the original $O(n^2)$ method and the new bin lattice method. The test were carried out using three different script files with increased number of Agents. The efficiency of the bin lattice structure was further tested by changing the number of subdivisions of the lattice.

D.0.2.5 Simple Flocking system evaluation

Figure D.1. shows the simple flocking system defined with the script file shown in Appendix E.0.6. It contains a number of Agents following a simple path with no other obstacles in the Environment.

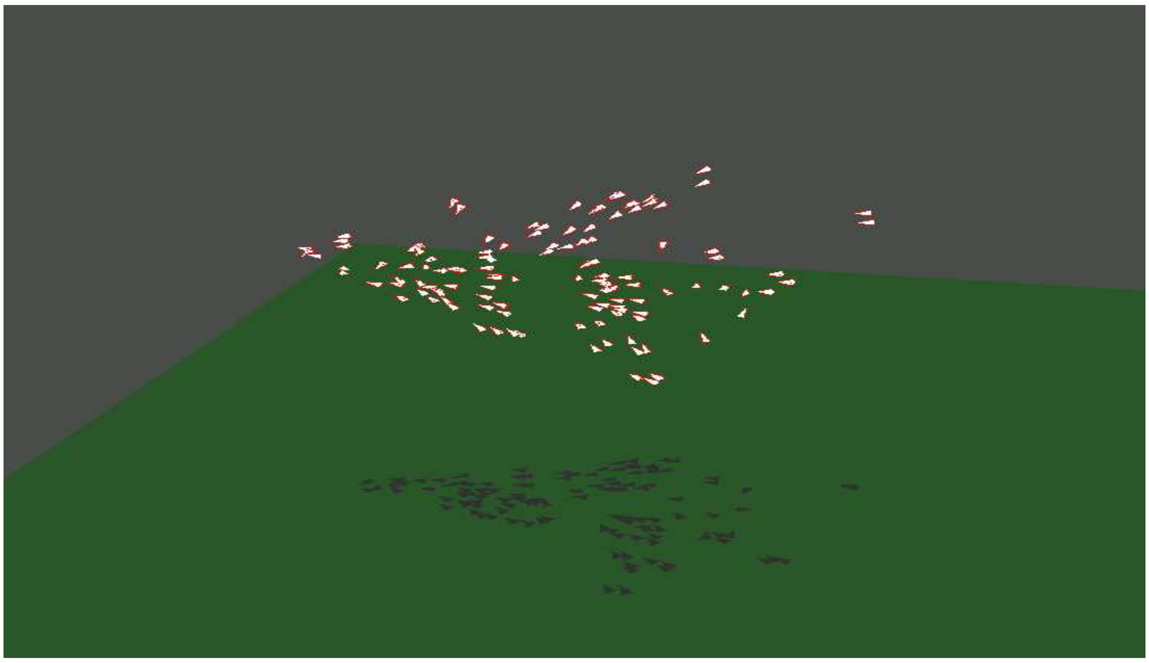


Figure D.1: Simple Flocking system

Figure D.2. shows the plot of the old system vs the new system with increased number of Agents. The new system shows an increase of at least 15 frames per second for the new system for numbers of Agents up to 500. After this the FPS drops with both systems but the new system is still consistently faster.

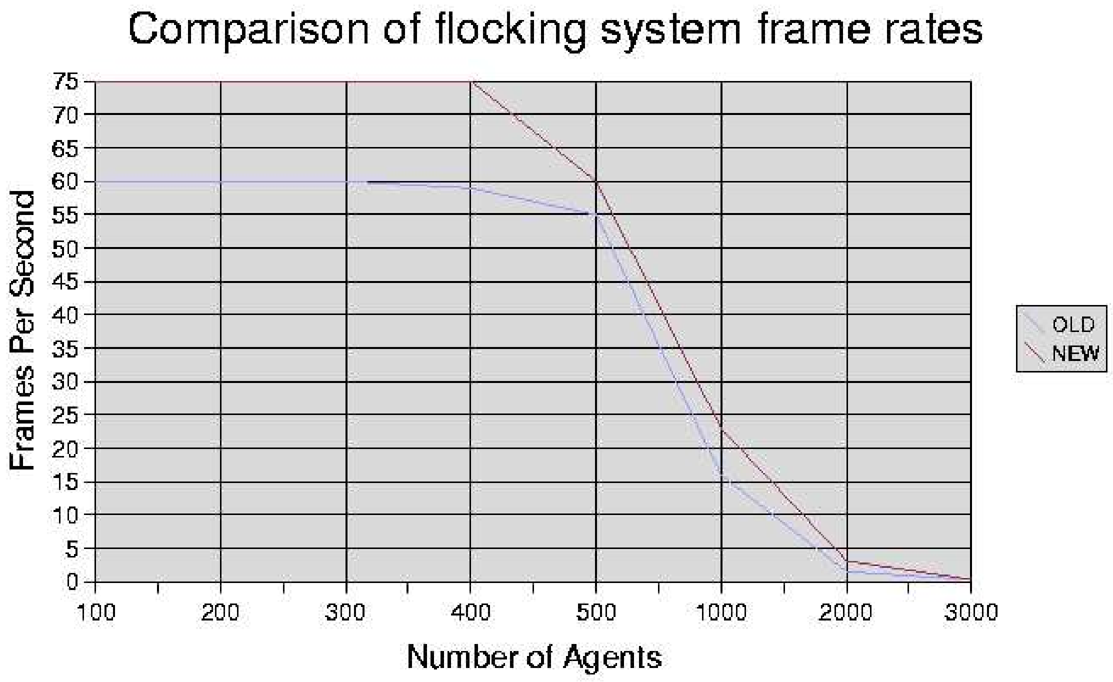


Figure D.2: Comparison of systems using Simple Flock

Figure D.3. shows a comparison of different lattice subdivisions using the simple flock script, the 5x2x5 (50 bins) subdivisions give the best results and 20x20x20 (8000 bins) giving the worst. Part of this is due to the number of bins used and the function call overhead, however comparison of the figures in Table E.2. show this not to be the only contributing factor.

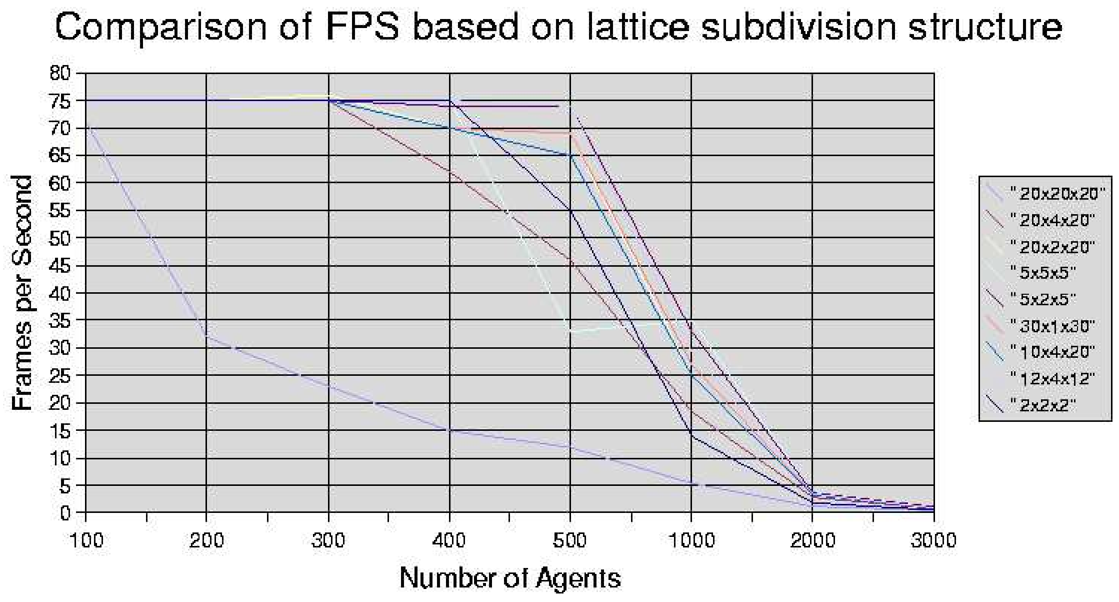


Figure D.3: Comparison of simple flock using different lattice structures

D.0.3 Object Avoidance

Figure D.4. shows the object avoid system defined with the script file shown in Appendix E.0.9. The Environment contains three objects for the Agents to avoid each of which is placed at the center of the Agents path so the Agents are compelled to hit them.

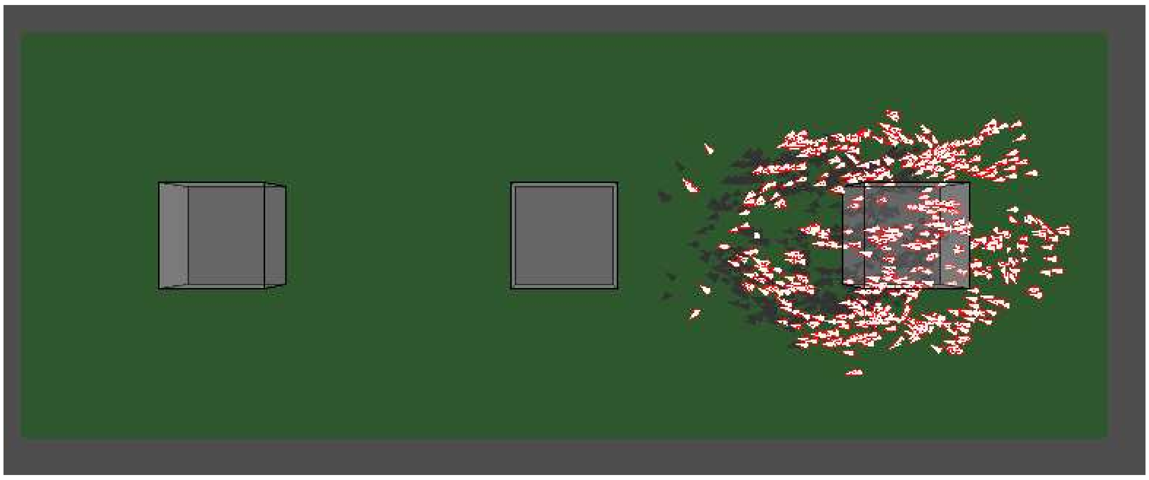


Figure D.4: Object avoid system

Figure D.5. again shows the comparison of the old and new systems with the bin lattice structure giving increased frame rates.

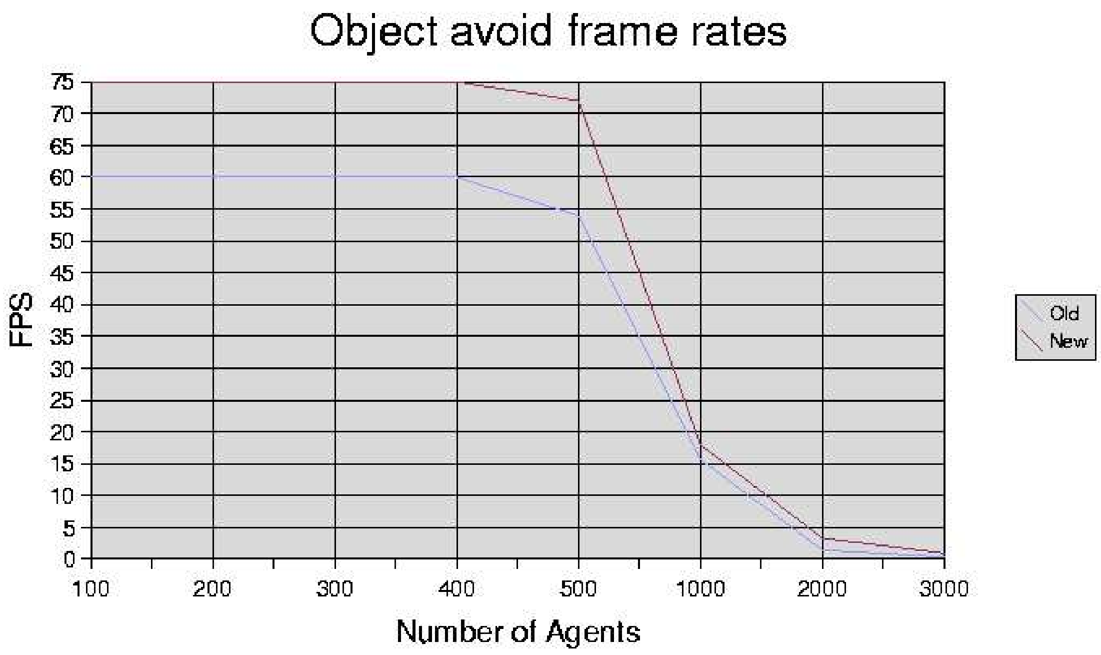


Figure D.5: Comparison of systems using object avoidance

Figure D.6. shows a comparison of different lattice subdivisions using the object avoid script script, again the number of subdivisions makes a contributing factor to the speed of the system, however the number of bins is not as significant to the speed increase as the number of subdivisions in the various axis.

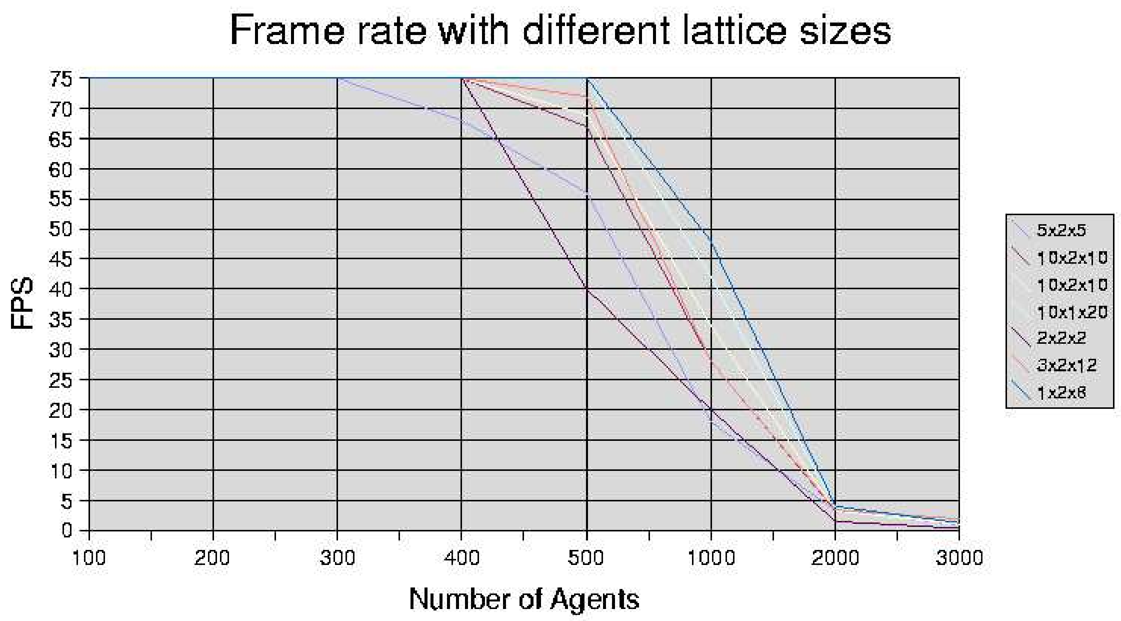


Figure D.6: Comparison of object avoidance using different lattice structures

D.0.4 Terrain Following

Figure D.7. shows the use of both Terrain following and Animated .obj files. This system is defined with the script file shown in Appendix E.0.12. The Environment contains no objects but the Agents must follow the terrain using a "use lowest route" terrain following algorithm.

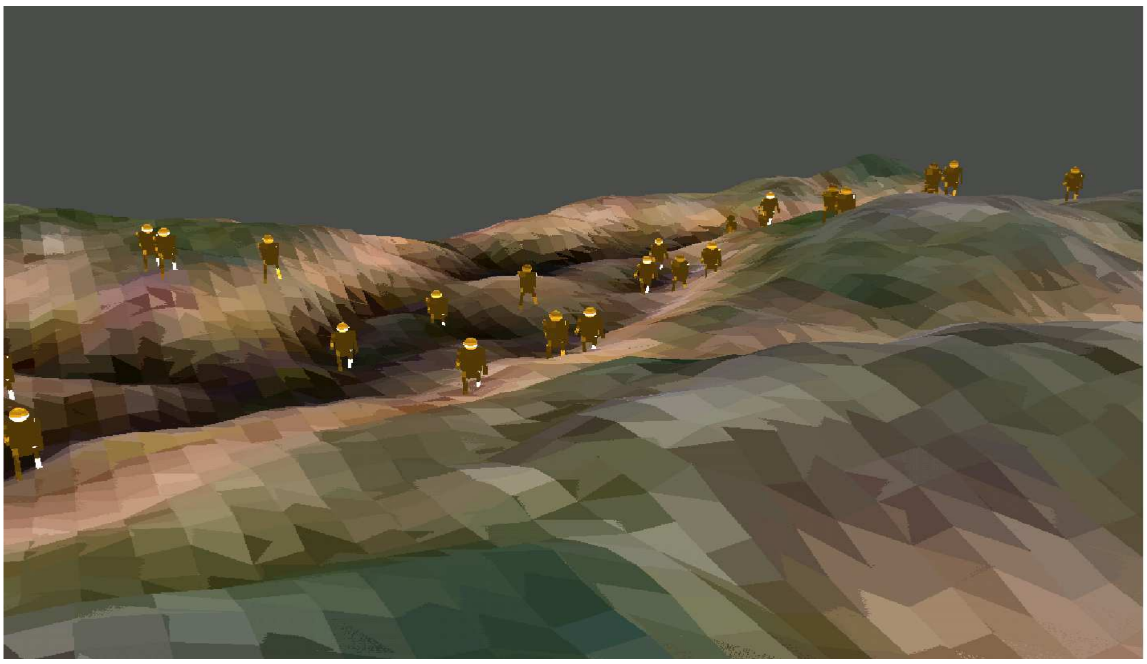


Figure D.7: Terrain Following

Figure D.8. again shows the comparison of the old and new systems with the bin lattice structure giving increased frame rates however the overall frame rate of both systems is lower due to the use of terrain, animated obj files and path following algorithms.

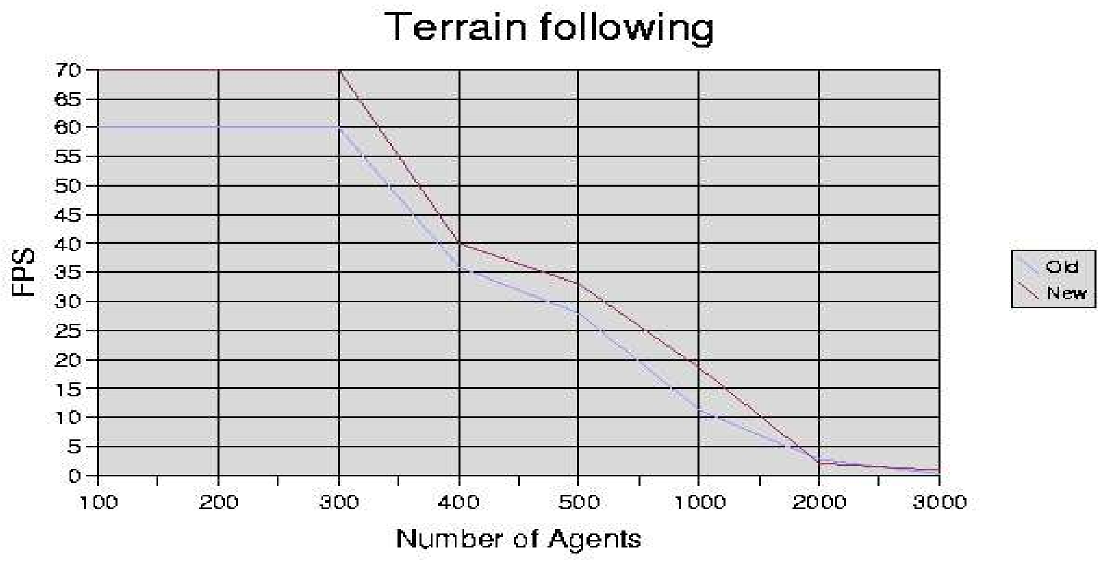


Figure D.8: Comparison of systems using terrain following

Figure D.9. shows a comparison of different lattice subdivisions using the terrain follow script, again the number of subdivisions makes a contributing factor to the speed of the system however best results were obtained by a 12x1x12 (144 bins) lattice rather than the 4x2x4 (32 bins) lattice which again indicates that the number of bins is not the major contributing factor to speed.

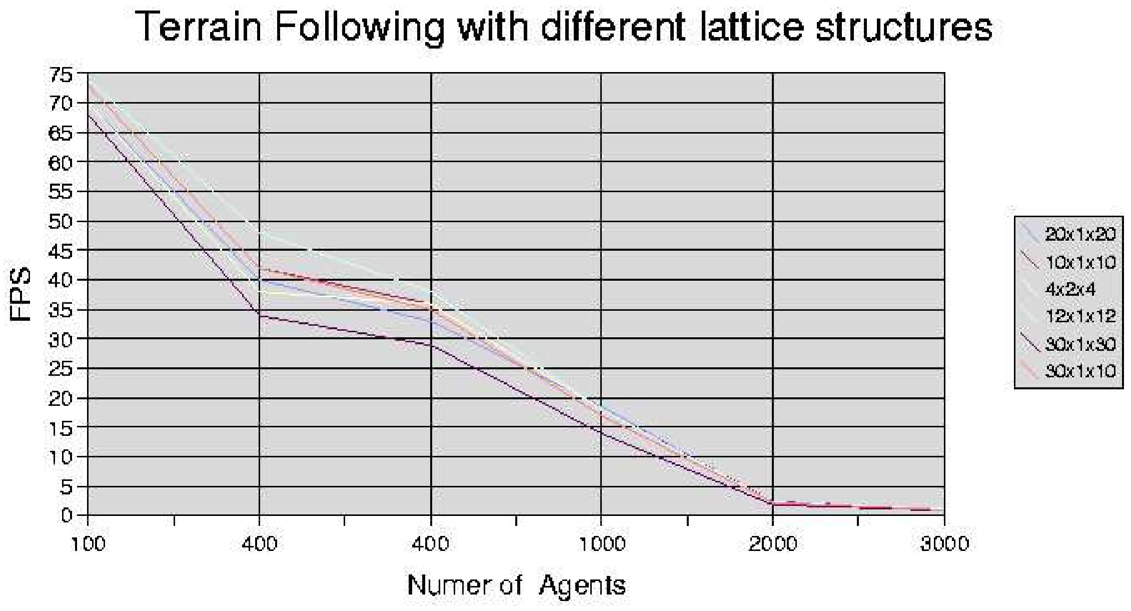


Figure D.9: Comparison of terrain following using different lattice structures.

D.0.5 Conclusions and Future work

The new system does show an improved frame rate on the original system, however the 16 times speed increase reported by Reynolds [Rey00] has not been achieved. This may be due to a number of factors.

The original system was developed using the C++ standard template library vector class which is highly optimized, and allows for fast access of elements, the system reported in the Reynolds paper [Rey00] was written in C so the storage of Agents was most likely to have used a linked list structure; therefore the introduction of the bin lattice structure would have, in this case, been more efficient than the linked list.

Reynolds also mentions other speed ups in the paper including the reduction of the number of cycles in which the Agent thinks and increasing the separation of the flock to reduce the bin membership. Neither of these methods have yet been considered in the developed system.

The use of different lattice sub-divisions is also an important factor to the speed of the system, by allowing this to be user definable in the script file the speed up may be tuned depending upon the simulation desired. A further improvement would be the use of a non-linear lattice structure with area's of interest having more sub-divisions and outlying area having a more course structure. This would be relatively simple to implement but would require a tool to be developed to design the grids and the resultant grid saved to a file and loaded into the simulation.

Another potential bottle neck is the *FillBins* method which is called each frame to determine which bin the Agents belong to. If the Agent's bin membership is changed by the Agent only when it moves from a bin this time could be reduced, however due to the structure of the present system this re-design could be problematic.

The rapid drop off in speed at around the 500 Agent mark is consistent throughout all the simulations tested an is likely to be due to the system reaching a page size in the amount of memory used and requiring the swapping of pages to physical disk.

Another potential speed up could be the introduction of a threaded multi-processor architecture with one CPU dedicated to the rendering pipeline and the other updating the Agent. This would also require a major re-design of the system with the introduction of semaphores for rendering / Agent update synchronization.

Appendix E

Bin lattice data structure test results

E.0.6 Flocking script file for Normal flocking test

```
    WorldBBox 0 0 0 80.0 80.0 80.0 10 10 10
    OutputFile test.out
    OutFileFrameSkip 1
    UpdateRate 1
    Camera 0 50 40 0 0 0 0 1 0 800 660 45.0 1.33 0.1 450.0
    Randomize
    AgentEmitter 0.0 0 0 120 0.7 0.3 0.33 0.6 0 0
    PathFollow 0 -20 0 2 -10 0 -5 0 1 -4 25 -2 -5
    FlockAvoidWeight 0 5210.02
    EnvAvoidWeight 0 20.0
    BBoxAvoidWeight 0 320.0
    MinCentroidDist 0 4.0
    CentroidWeight 0 90.0
    Noise 0 12234.1 7
    MinCentroidDist 0 3.0
    GroundPlane -20
    //AnimObj 0 ../Models/seagull1.obj ../Models/seagull2.obj 4 5
    UseOveride 0
```

E.0.7 Test results

Table E.1. shows the results of the old flock system with the new flock system using a 10 x 10 x 10 bin lattice structure

fps	100	200	300	400	500	1000	2000	3000
$O(n^2)$	60	59.98	59.98	59	55	16	1.59	0.34
$O(nk)$	75	75	74.9	75	60	23	3.2	0.96

Table E.1: Comparison of systems using simple flock

fps	100	200	300	400	500	1000	2000	3000	Number of Bins
20x20x20	71	32	23	18	12	5.37	1.20	0.49	8000
20x4x20	75	75	75	62	46	18.45	2.82	0.86	1600
20x2x20	75	75	75	70	69	25	3.28	0.98	800
5x5x5	75	75	75	75	33	35	3.73	1.14	125
5x2x5	75	75	75	74	74	33	3.69	1.15	50
30x1x30	75	75	75	70	69	27	3.28	0.98	900
10x4x20	75	75	75	70	65	25	3.33	0.98	800
12x4x12	75	75	75	75	74	29.44	3.45	0.98	576
2x2x2	75	75	75	75	55	14	1.84	0.55	8

Table E.2: Comparison of system using different lattice structures

E.0.8 Bin lattice subdivisions

E.0.9 Flocking script file for Object avoidance

```
WorldBBox 0 0 0 80.0 80.0 80.0 10 10 10
OutputFile birds.out
OutFileFrameSkip 1
UpdateRate 1
Camera 0 80 0 0 0 0 0 0 1 800 660 45.0 1.33 0.1 450.0
Randomize
GroundPlane -20
EnvObj 0.0 0.0 0.0 6.0 6.0 6.0 5.0
EnvObj -20.0 0.0 0.0 6.0 6.0 6.0 5.0
EnvObj 20.0 0.0 0.0 6.0 6.0 6.0 5.0
AgentEmitter -30.0 0 0 520 0.7 0.3 0.33 0.6 0 0
PathFollow 0 -20 0 0 -10 0 0 10 0 0 20 0 0
CentroidWeight 0 122.9
FlockAvoidWeight 0 5100.9
EnvAvoidWeight 0 6129.0
BBoxAvoidWeight 0 2.0
MinCentroidDist 0 4.0
UseOveride 0
```

E.0.10 Old system vs New

Table E.3. shows the results of the old flock system with the new flock system using a 10 x 10 x 10 bin lattice structure

fps	100	200	300	400	500	1000	2000	3000
$O(n^2)$	60	60	60	59	54	15.66	1.50	0.34
$O(nk)$	75	75	74.9	68	56	18	3.36	0.94

Table E.3: Comparison of systems using object avoid

fps	100	200	300	400	500	1000	2000	3000	Number of Bins
5x2x5	75	75	75	75	72	27	2.83	0.78	50
10x2x10	75	75	75	75	67	28	3.23	1.0	200
10x1x20	75	75	75	75	75	42	3.85	1.14	200
2x2x2	75	75	75	75	40	20	1.51	0.35	8
3x2x12	75	75	75	75	72	28	3.40	1.90	72
1x2x8	75	75	75	75	75	48	4.05	1.21	16

Table E.4: Comparison of object avoid using different lattice subdivisions

E.0.11 Bin lattice subdivisions

E.0.12 Flocking Script for Terrain following

```
WorldBBox 0 0 0 80.0 80.0 80.0 20 1 20
OutputFile birds.out
Camera 0 50 40 0 0 0 0 1 0 800 660 45.0 1.33 0.1 450.0
GroundPlaneTex -10 MountainHM.bmp MountainHMC.bmp 40
AgentEmitter 0.0 -10.0 9.0 40 0.7 0.2 0.33 0.9 0 1
HeightDiff 0 10.8
PathFollow 0 -10 -10 -10 -5 -10 -2 0 -10 2 -10 -10 10
CentroidWeight 0 225.9
MinCentroidDist 0 3.0
PredWeight 0 25219.0
PredHitLevel 0 100
PredAvoidVelocity 0 2 2 2 2
DefaultPredAvoidDir 0 0 0 0
AnimObj 0 ../Models/Droid1.obj ../Models/Droid2.obj 3 1
UseOveride 0
LockedCentroid 0 -1 -10 0
LockToGP 0
FlockAvoidWeight 0 22225.9
```

E.0.13 Old system vs New System

Table E.5. shows the results of the old flock system with the new flock system using a 10 x 10 x 10 bin lattice structure

fps	100	400	500	1000	2000	3000
$O(n^2)$	60	36	28	11.30	2.80	0.33
$O(nk)$	70	40	33	18.56	2.19	0.95

Table E.5: Comparison of systems using terrain following

E.0.14 Bin lattice subdivisions

fps	100	400	500	1000	2000	3000	Number of Bins
20x1x20	70	40	33	18.56	2.19	0.95	400
10x1x10	73	42	36	18	2.44	1.01	100
4x2x4	70	38	36	18	2.35	0.99	32
12x1x12	74	48	38	18	2.40	1.10	144
30x1x30	68	34	29	14	1.89	0.94	900
30x1x10	73	42	35	17	2.28	0.94	300

Table E.6: Comparison of terrain following using different lattice subdivisions

Appendix F

OpCode Enumerated types

```
enum OPCODES {
IF,TRANSLATE,ROTATEX,ROTATEY,ROTATEZ,VERTEX,GLEND,POLYGON,POINT,
POINTSIZE,LINESIZE,LINELOOP,QUAD,COLOUR,PUSHMATRIX,POPMATRIX,ADDDIR,
RANDOMIZE,MUL,ADD,SUB,DIV,SET,FPUSH,FPOP,FATAN,FRAD2DEG,FNEGATE,
DEBUG,LENGTH,FSQRT,FSIN,FASIN,FCOS,FACOS,FDIV,FSTACKTRACE,FSUB,FADD,
FMUL,SETDIR,FDUP,CALL,SUBDIR,REVERSE,SPHERE,VERTEXF,CYLINDER,NORMALIZE,
USEAGENTRENDER,SETANIMCYCLE,RENDERAGENT,SETGLOBALPOS,GETGLOBALPOS,
SETGLOBALCENTROID,GETGLOBALCENTROID,SPHERESPHERE,LOOPBIN,BEEP,SETAGENTI,
GETAGENTI,DIVDIR,RANDOMIZEPOS,IFELSE,PUSHGPYLEVEL,SCALE,SPHEREPLANE,
SPHEREENVOBJ,RENDERFRAME,END,OPENBRACE,CLOSEBRACE,CALLLIST,
CYLINDERCYLINDER,DEBUGOPON,DEBUGOPOFF,SETGLOBALCOLLIDEFLAG,
GETGLOBALCOLLIDEFLAG,DOT,SOLIDSPHERE,LINES,SETGLOBALDIR,GETGLOBALDIR,
MULDIR,CUBE,SETGPYLEVEL,GETNOISEVALUE,USENOISE,RENDERMATERIAL,FPUSHD,
ENABLELIGHTS,DISABLELIGHTS,SMOOTH,FLAT
};
```

```
enum IFEVALUATORS
{
  IFEQUAL,IFNOTEQUAL,IFGREATER,IFGREATEREQ,
  IFLESS,IFLESSEQ
};
```

Appendix G

Overloading of Comparison Operators

G.1 Point

```
bool Point3 :: operator==(Point3& v)
{
 if(x==v.x && y==v.y && z==v.z)
    return true;
 else return false;
}
bool Point3 :: operator!=(Point3& v)
{
 if(x!=v.x && y!=v.y && z!=v.z)
    return true;
 else return false;
}
bool Point3 :: operator>(Point3& v)
{
  if(x>v.x && y>v.y && z>v.z)
     return true;
  else return false;
}
bool Point3 :: operator>=(Point3& v)
{
 if(x>=v.x && y>=v.y && z>=v.z)
    return true;
 else return false;
}
bool Point3 :: operator<(Point3& v)
{
  if(x<v.x && y<v.y && z<v.z)
    return true;
 else return false;
}
bool Point3 :: operator<=(Point3& v)
{
  if(x<=v.x && y<=v.y && z<=v.z)
     return true;
else return false;
}
```

G.2 Vector Operator

```
bool Vector::operator==(Vector& v)
{
 if((v.x == x ) && (v.y == y) && (v.z == z) )
```

```
         return true;
      else
         return false;
    }
```

Appendix H

Massive Information

Subject:

Re: Request for Info

From:

Info <info@massivesoftware.com>

Date:

Sat, 16 Aug 2003 10:12:27 +1200

To:

Jonathan Macey <jmacey@bournemouth.ac.uk>

Hi Jonathan

Sorry it's taken me a while to get back to you. It always gets a bit busy around Siggraph.

> Thanks for your reply, I have a few questions

>

> 1. Does massive use any evolutionary type algorithms ?

Not yet. It was always my intention to use some but I haven't had the chance to do so yet.

> 2. Are the Agents programmed from scratch or are there helper flock functions such as those used in Behaviour.

There's nothing built into the software but Massive comes with demo agents. It's trivial to copy and paste modules from one agent to another which makes it easy to build agents from scratch.

>

> 3. Are Agents brains totally controlled by Fuzzy logic or are there override functions?

By designing the fuzzy logic as a directed graph the user can modify the logic in any way so there's no need to override anything. Maybe I don't understand the question, but it's pretty straightforward to disable chunks of the brain or to attach more nodes to provide a higher level of control.

>

> 4. Does the physics engine control the Agents at a higher level than the Brain i.e. do the Agents decide upon a behaviour then the physics engine stops them from doing certain things or the other way round?

The physics is only active where and when it's needed. When rigid body dynamics is active the agents can apply forces to the dynamic segments but they can't kinematicaly control a dynamic segment.

>

> 5. Do you have any metrics for the speed of generation of a single frame?

On my 900 MHz laptop I can run 10,000 Lord Of The Rings quality agents at around 20 seconds per frame. Of course some agents are more complex than others so the times can vary significantly.

>

> 6. Are the Agents stored in any type of spacial data structure (such as the bin lattice one Reynolds uses) if not are they just stored in a big list.

There are various optimizations for different purposes. For instance, sound processing uses a different system from collision detection.

- Stephen Regelous

Bibliography

[Ali03] Alias|Warefront. *www.alias.com*. Silicon Graphics Ltd, USA, 2003.

[AVAU86] Ravi Sethi Alfred V. Aho and Jeffery D Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, seconf edition, 1986.

[CLMP95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995.

[EGKP02] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.

[eta98] Ebert etal. *Texturing and Modeling A Procedural Approach*. AP Professional, second edition, 1998.

[Fer99] Jacques Ferber. *Multi-Agent Systems An introduction to Distributed Artificial Intelligence*. Addison-Wesley, first edition, 1999.

[gg03] gnu g++. *www.gnu.org*, 2003.

[HDT] Adam T. Hayes and Parsa Dormiani-Tabatabaei. Self-organized flocking with agent failure: Off-line optimization and demonstration with real robots.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library. A tutorial and reference*. Addison-Wesley, first edition, 1999.

[jr01] F. S. Hill jr. *Computer Graphics using OpenGL*. Prentice-Hall, second edition, 2001.

[KHM+98] J. Klosowski, M. Held, Joseph S. B. Mitchell, K. Zikan, and H. Sowizral. Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Trans. Visualizat. Comput. Graph.*, 4(1):21–36, 1998.

[Kli] Jon Klien. Breve : a 3d environment for the simulation of decentralized systems and artificial life. Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. volume 8 of *Information and Control*, pages 607–639, 1965.

[Koe02] Dan Koeppel. Massive attack popular science www.popsci.com. 2002.

[Len01] Eric Lengyel. *Mathematics for 3D Game Programming & Computer Graphics*. Game Development Series. Charles River Media, first edition, 2001.

[Lin03] Redhat Linux. *www.redhat.com*, 2003.

[Ltd03] Silicon Graphics Ltd. *www.sgi.com*, 2003.

[Mas03] Massive. *www.massivesoftware.com*. Massive, MASSIVE Ltd PO Box 11189 Wellington New Zeland, 2003.

[MW88] M. Moore and J. Willhelms. Collision detection and response for computer animation. *Comput. Graph.*, 22(4):289–298, August 1988. Proc. SIGGRAPH '88.

[MW99] Tom Davis Dave Shreiner Mason Woo, Jackie Neider. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1999.

[Par] L. E. Parker. Designing control laws for cooperative agent teams. pages 582–587.

[Par02] Rick Parent. *Computer Animation Algorithms and Techniques*. Morgan Kaufmann, first edition, 2002.

[PG95] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Comput. Graph. Forum*, 14(2):105–116, June 1995.

[Pij00] Wim Pijls. Lr and ll parsing: some new points of view. volume 32 of *ACM SIGCSE*, pages 24–27. ACM Press New York, NY, USA, 2000.

[Ree83] W. T. Reeves. Particle systems : A technique for modeling a class of fuzzy objects. Computer Graphics (Proceedings of SIGGRAPH 83), pages 359–376, San Fransisco, 1983.

[Ree85] W. T. Reeves. Approximate and probalistic algorithms for shading and rendering particle systems. Computer Graphics (Proceedings of SIGGRAPH 85), pages 313–322, 1985.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[Rey99] Craig Reynolds. Steering behaviors for autonomous characters. Game Developers Conference 1999, 1999.

[Rey00] Craig Reynolds. Interactions with groups of autonomous characters. Game Developers Conference 2000, pages 449–460, San Francisco, 2000.

[Rey03] Craig Reynolds. *http://opensteer.sourceforge.net/*. Sony Research and Development, 2003.

[Sim91] Karl Sims. Artificial evolution for computer graphics. *Computer Graphics*, 25(4):319–328, July 1991.

[Sof03] Softimage. *www.softimage.com/behaviour/v1*. Softimage, Avid Technology Group Europe Ltd. Pinewood Studios, Buckinghamshire. SL0 0NH England, 2003.

[Spe88] D. Spector. Efficient full lr(1) parser generation. volume 23 of *ACM SIGPLAN Notices*, pages 143–150. ACM Press New York, NY, USA, 1988.

[SV00] Peter Stone and Manuela M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[UOT83] T. Uchiki, T. Ohashi, and M. Tokoro. Collision detection in motion simulation. *Comput. & Graphics*, 7(3–4):285–293, 1983.