Examensarbete
LITH-ITN-MT-EX--04/028--SE

REACT

Crowd Simulation System for Visual Effects

Fredrik Limsäter

2004-03-30

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--04/028--SE

REACT

Crowd Simulation System for Visual Effects

Examensarbete utfört i Medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Fredrik Limsäter

Handledare: Pete Medrow, Cinesite (Europe) Ltd.
Examinator: Anders Ynnerman

Norrköping 2004-03-30

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2004-03-30

Språk
Language

☐ Svenska/Swedish
■ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
■ D-uppsats

☐ _____

ISBN

ISRN LITH-ITN-MT-EX--04/028—SE

Serietitel och serienummer **ISSN**
Title of series, numbering

URL för elektronisk version

http://www.ep.liu.se/exjobb/itn/2004/mt/028/

Titel
Title

REACT – Crowd Simulation System for Visual Effects

Författare
Author

Fredrik Limsäter

Sammanfattning
Abstract

By using existing knowledge from the game community, which have had a long experience from game artificial intelligence, and new research from the field of artificial intelligence I have implemented REACT, a crowd simulation system for visual effects. REACT is based on high-level behaviour that uses an underlying layer of low-level behaviour. The high-level capabilities gives the digital character means to reasoning about how to achieve certain goals based on a knowledge base of rules and facts that are present in the virtual world. This gives the digital character a degree of autonomous intelligent behaviour.

REACT is designed to integrate directly into the 3D animation package Maya as a plug-in. This means that the animators can continue to animate their characters via their animation package of choice, rather than having to learn a new technology. In addition, many animators are already familiar with the workflow of Maya, so learning curves are reduced.

REACT is already in use in the visual effects industry where it has proven itself to be a worthy competitor to the existing systems on the market.

Nyckelord
Keyword

Crowd simulation, artificial intelligence, steering behaviours, animation system, visual effects

REACT[1]

Crowd Simulation System for Visual Effects

Fredrik Limsäter

March 30, 2004

[1] Reasonable Embodied Agents for Crowd Simulation

Abstract

By using existing knowledge from the game community, which have had a long experience from game artificial intelligence, and new research from the field of artificial intelligence I have implemented REACT, a crowd simulation system for visual effects. REACT is based on high-level behaviour that uses an underlying layer of low-level behaviour. The high-level capabilities gives the digital character means to reasoning about how to achieve certain goals based on a knowledge base of rules and facts that are present in the virtual world. This gives the digital character a degree of autonomous intelligent behaviour.

REACT is designed to integrate directly into the 3D animation package Maya as a plug-in. This means that the animators can continue to animate their characters via their animation package of choice, rather than having to learn a new technology. In addition, many animators are already familiar with the workflow of Maya, so learning curves are reduced.

REACT is already in use in the visual effects industry where it has proven itself to be a worthy competitor to the existing systems on the market.

Contents

List of Figures

Chapter 1

1 Introduction

What is a crowd simulation and why do we need it? To answer these questions we need to understand what a crowd is, what defines a crowd and what are the characteristics of a crowd.

> *crowd*
> *- a large number of things or people considered together*

To understand the characteristics of a crowd we need to look at the field of social psychology. From the psychological point of view the word or expression "crowd" assumes quite a different signification to the statement above. Gustave Le Bon [17] means that under certain given circumstances, and only under those circumstances, an accumulation of men presents new characteristics very different from those of the individuals composing it. The sentiments and ideas of all the persons in the gathering take one and the same direction, and their conscious personality vanishes. A collective mind is formed, doubtless transitory, but presenting very clearly defined characteristics. The gathering has thus become what Le Bon call an organised crowd, or, a psychological crowd. It forms a single being, and is subjected to the "Law of the mental unity of crowds".

By understanding what a crowd is we can explain what a crowd simulation is and why we need it. A crowd simulation is a computer simulation of a crowd of digital characters with a certain artificial intelligence. The individual characters can act on their own, they do not need to be pre-programmed or scripted, the characters respond on their environment without external input. They will respond to changes in their virtual worlds, for example sound, obstacles and enemies. If one character acts

naturally so will the rest of the crowd and since they all are affected by the same virtual world they will form what Le Bon calls an organised crowd.

So why do we need these crowd simulations? Large crowd scenes, in particular battle scenes require a sheer number of extras which make them extremely expensive and their violent nature makes them very dangerous to do in real life. Given the complexity and danger of such scenes, it is clear that filming these crowds with real actors would be, if not impossible, very expensive and a logistical nightmare.

Different approaches have been taken and the results can be seen in films like Ben Hur[2] 1959, Gladiator 2000 and the Lord of the Rings trilogy 2001-2003 where they visualized from a couple of hundred characters to hundred of thousands characters.

While some of these approaches has been very simple like particle systems, motion capture or just cloning of pre-filmed clips, others has been more advanced with an architecture based on both low-level reactive behaviour (like automatically avoid obstacles and other characters etc.), and high-level reasoning behaviour. While the latter produces very realistic behaviour which enhances the visual appearance, it is still a lot of work for the animator or artist to set up the scene or simulation. And they often work in a program separated from their animation package of choice.

1.1 The REACT System

This thesis will address some of the short comings of other systems and the result is implemented as a complex animation system called REACT, Reasonable Embodied Agents for Crowd Simulation. REACT is meant to reduce the time to create complex animations of crowds compared to existing techniques.

The REACT system consists of several different modules, see 3.1.1, but this thesis will focus on the behavioural part, the artificial intelligence and the integration into existing 3D software.

[2] They actually had more than 10 000 extras in the same scene, but increased that by cloning filmed clips

1.2 Overview

The reader of this thesis is expected to have some knowledge of computer graphics and artificial intelligence. A good reference and introduction to artificial intelligence can be found in “Artificial Intelligence - A Modern Approach” [13].

Chapter 2 will discuss the theoretical foundations of my work and previous work from which I obtain the basic concepts. Attention will be paid to low- and high-level behaviours, AI-architectures and data structures used in REACT.

Chapter 3 will first discuss the theoretical implementation of REACT and then continue to discuss the technical implementation.

Chapter 4 concludes the thesis, with a look at what has been accomplished and points out some promising directions for future work.

Chapter 2

2 Theoretical Basis

In this chapter I will discuss the theoretical foundations of my work. I will start by explaining some fundamental concepts of AI that are required to follow the discussion in this chapter. Then I will discuss low-level behaviours and finally conclude the chapter by returning to the field of artificial intelligence and discuss high-level behaviours.

2.1 Fundamental AI concepts

To many people, AI is only a buzzword for technology used in sci-fi films like "AI", "Star Trek" or the famous HAL in "2001 - A Space Odyssey". But AI is used in many fields like computer games, search engines on the Internet and industrial robots to mention a few. But what is the meaning of artificial intelligence? There exist many different definitions but Kurzweil [18] points out one possible meaning.

> *Artificial intelligence is the art of creating machines that perform functions that require intelligence when performed by people.*

What will the purpose be for the AI system in REACT? There are three important features that the AI must provide. These are:

- *Primitive Behaviour* such as performing purposeful gestures.
- *Movement* in the virtual world, and dealing with obstacles, barriers and avoiding other characters.
- *Decision making* on a high-level to decide which actions are necessary for the character to accomplish his task and in what order.

2.1.1 Autonomous Agent and Multi-Agent System

In AI, a smart entity is known as an *agent* and this definition will be used in the remains of this thesis instead of digital character. According to Stuart Russel and Peter Norvig [13] the agent is called *non autonomous* agent if the agent's actions are based completely on built-in knowledge, meaning that it pay no attention to its percept. But if the agent's behaviour is determined by its own experience it is called an *autonomous agent*. If a system can handle more than one agent in coordination it is known as a *multi-agent system*.

2.1.2 Embodied and situated

Agents can be embodied. This is a different way of dealing with agents and it means that the intelligence is separated from the body. This may not sound revolutionary, but it will actually lead to more genuine behaviour. The embodied agents are subject to the constraints of its environment where it is *situated*, which means that the actions of their brains are limited. In general this means that the possible actions that can be executed by the body - and hence the AI - are restricted to the subset of actions consistent with the laws of the simulation. However, embodiment does not limit what the AI can achieve; it just restricts how it is done.

To understand the difference between these two approaches an example from Alex Champandard [8] is used:

> *A non-embodied agent can change its position itself to reach any point in space. An embodied agent needs to move itself relatively to the current position, having to actually avoid obstacles. It will not even have the capability to update its position directly.*

By simulating the body you can add biologically plausible errors to the interaction with the virtual world. For example, agents could have difficulty perceiving if it is a friend or enemy in the distance.

2.2 Low-level behaviours

Even the simplest living creatures can move and movement can be re-created relatively simple. This part will explain how we can give agents the ability to navigate around their world in a life-like and improvisational

manner. The abundance of existing work from Craig Reynolds [1] on the subject will come in handy as guidance.

The low-level behaviours of an agent can be divided into a hierarchy of three layers. These are action-selection, steering and locomotion.

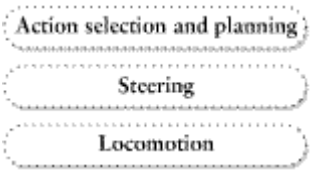

Figure 2.1 Hierarchy of low-level behaviours

For simplicity we assume 2D but it generalizes to 3D.

2.2.1 Locomotion

The *locomotion* layer represents the embodiment explained above. It will transform the data sent from the steering layer into motion of the body. Since we want to consider steering behaviours independent from the underlying locomotion scheme Reynolds introduce a simple *idealized vehicle model.*

A simple vehicle model

For the simplicity of this chapter I assume that the vehicle is based on a point mass approximation. The point mass is defined by a position, mass and velocity property. The point mass velocity is modified by applying forces. The vehicle has a limit in the force that can be applied and it has a maximum speed. Finally it has an orientation vector.

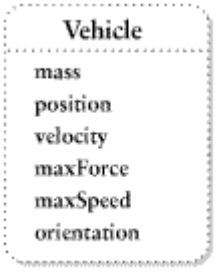

Figure 2.2 Simple vehicle model

The physics of the vehicle is based on forward Euler integration which means that at each simulation step, calculated steering forces are applied to the vehicle's point mass. This produces acceleration equal to the steering force divided by the vehicle's mass. This acceleration is added to the old velocity to produce a new velocity and finally, the velocity is added to the old position.

2.2.2 Steering behaviours

This discussion of steering behaviours assumes that locomotion is implemented by the simple vehicle model described above. It also assumes that it is parameterized by a single steering force vector. Therefore the steering behaviours are described in terms of the geometric calculation of a vector representing a desired steering force.

Wander

The wander behaviour is implemented using random steering forces. One easy way to implement it is to add a random force to the overall steering force for each frame of the animation. But the resulting movement from this action would not look very realistic; the steering force would change direction almost instantaneously and don't produce any sustained turns. A solution that Reynolds proposes is to constrain the deviation of the force to a circle right in front of the vehicle, see Figure 2.3. To produce a new steering force, a random displacement is added to the previous force and the sum is then constrained to the circle's boundary. This approach will allow the vehicle that in one frame be turning up and to the left, and in the next frame still be heading in almost the same direction but slightly offset.

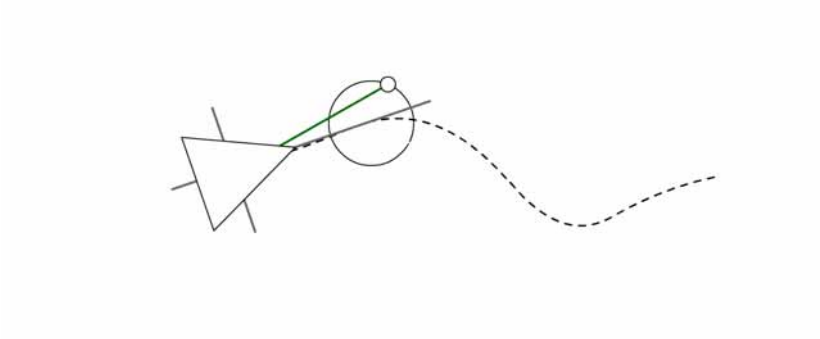

Figure 2.3 Wander behaviour

This behaviour can be adjusted by using different values of the radius and the position of the circle. See Figure 2.4

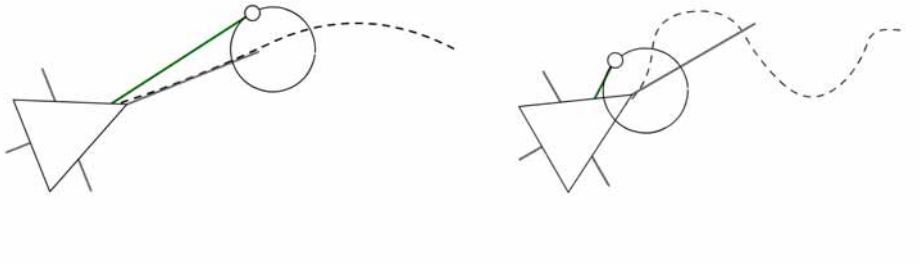

Figure 2.4 Different parameters for wander behaviour

If the circle is placed near the vehicle, the changes in the direction of movement will be faster. If it is placed farther away, the resulting movement will be more linear. Another approach to implement wander behaviour could be to use coherent Perlin Noise [9].

Seek

The seek behaviour is used to steer a vehicle towards a static target position. This behaviour makes the vehicle's velocity vector point toward the target position, see Figure 2.5

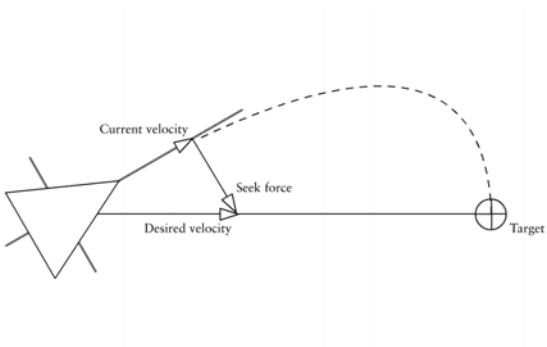

Figure 2.5 Seek behaviour

The implementation is very simple. The steering force is the difference between the velocity vector from the vehicle to the goal position and the vehicles current velocity vector.

$$\vec{V}_{seek} = \left(\overrightarrow{PQ}\right) - \vec{V} \qquad (2.1)$$

where P is the vehicles current position, Q is the target position and V is the vehicles current velocity. The problem with this behaviour is that when the vehicle reaches its goal it will just move right through it and attempt to approach it again, like a moth dancing around a light source. To come around this you can implement arrival behaviour.

Arrival

Arrival behaviour starts out the same way as in seek behaviour. But when the vehicle reach a certain distance from the target, the force that drive the vehicle forward will be ramped down linear until it come to a full stop at the target position. See Figure 2.6

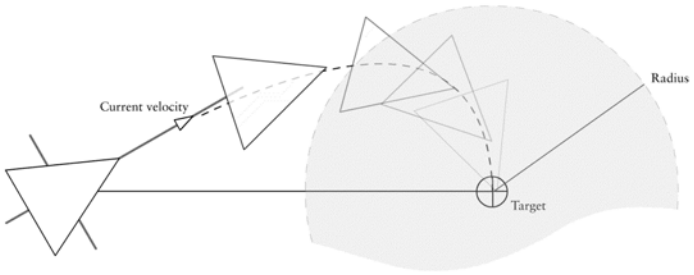

Figure 2.6 Seek with arrival behaviour

Some examples of this behaviour could include a pedestrian stopping for red light; or a bird coming in for landing.

$$s_{ramped} = \left|\vec{V}_{\max}\right| \left(\frac{\left|\vec{V}_{seek}\right|}{d} \right) \tag{2.2}$$

$$s_{clipped} = \min\left(s_{ramped}, \left|\vec{V}_{\max}\right|\right) \tag{2.3}$$

$$\vec{V}_{arrive} = \left(\frac{s_{clipped}}{\left|\vec{V}_{seek}\right|} \right) \vec{V}_{seek} \tag{2.4}$$

where V_{seek} is the velocity generated from the seek behaviour, $|V_{max}|$ is the vehicles maximum allowed speed, S is a scalar, d is a constant that defines the distance from the target to where to start slowing down. Finally, V_{arrive} is the new steering force.

Flee

Flee is very similar to seek, but instead of steering the vehicle towards the target, flee steers away from it. See Figure 2.7.

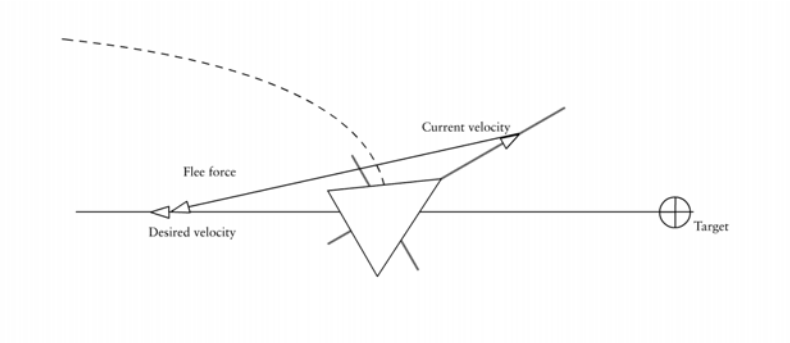

Figure 2.7 Flee behaviour

The desired velocity points in the opposite direction from the target. The steering force is the difference between the vector opposite to the velocity vector from the vehicle to the goal position and the vehicles current velocity vector.

$$\vec{V}_{flee} = \left(\overrightarrow{QP}\right) - \vec{V} \qquad (2.5)$$

where P is the vehicles current position, Q is the target position and V is the vehicles current velocity.

Pursuit

Pursuit is very similar to seek behaviour, but instead of seeking a static target, pursuit seeks another moving vehicle. See Figure 2.8.

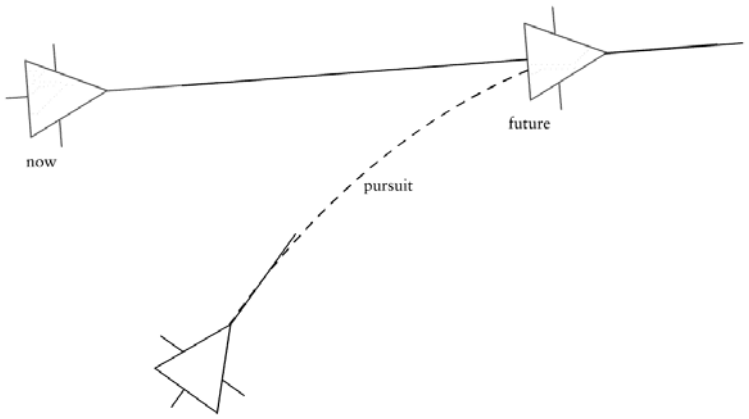

Figure 2.8 Pursuit behaviour

To be able to pursue and intersect another vehicle you need to know what its future position will be and a predictor see equation 2.6, is used for this task. The predictor looks at the targets current velocity times a time constant to get the future position. Of course, this implies that the target doesn't turn, which is likely to happen. But, as Reynolds mention in his paper, the resulting prediction will only be in use a limited time, in our case only 1/24 of a second.

$$\vec{P}_{predicted} = \vec{V} * t \qquad (2.6)$$

The resulting steering force is obtained using seek behaviour on the predicted future position.

Evasion

Evasion is analogous to pursuit. The difference is that flee is used instead of seek to evade from the targets future position. See Figure 2.9.

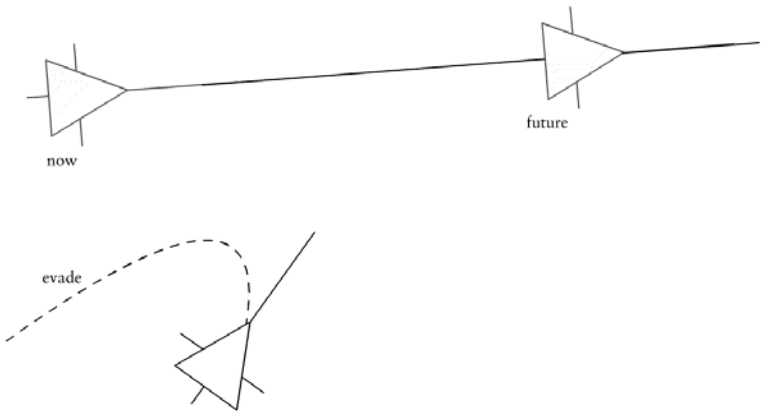

Figure 2.9 Evasion behaviour

Path following

Curve segments are used to steer the vehicle in path following. But instead of fixating the agent to the curve it can deviate from it. I use a radius attached to the curve and this creates a tube around it for the vehicle to move along. See Figure 2.10. The vehicle will try to move along the path while staying inside this tube.

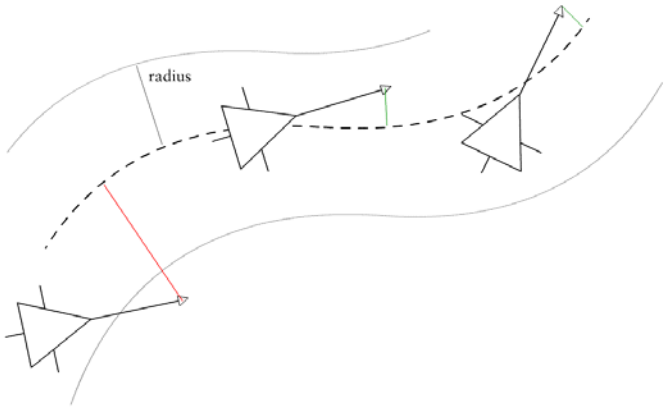

Figure 2.10 Path following

To see if the vehicle is following the path you predict its future position. Then you project this point to the nearest point on the curve and if the distance between them is less then the specified radius no corrective steering is required. If the vehicle is wandering away from the curve you use seek behaviour to seek closest point on path.

Flow field following

In flow field following the vehicle is affected by a flow filed or a vector field on the surface that the vehicle is moving on. See Figure 2.11

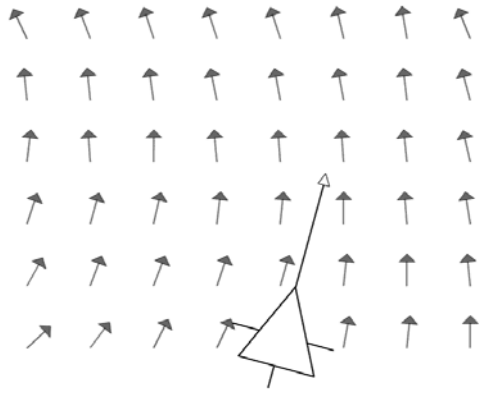

Figure 2.11 Flow field following

The vehicle will try to align it self with the local tangent of the closest vector, or a sum of nearby vectors.

Obstacle avoidance

To give vehicles ability to move in an arbitrary scene they must be able to handle obstacle avoidance.

The simplest representation of an obstacle is a circle, or a sphere in three dimensions. See Figure 2.12. For long walls and barriers a circle can be a very bad approximation, so I separate these two obstacles into different types. See Barrier avoidance. The advantage in using a circle or a sphere as approximation is the simple calculations to determining intersections.

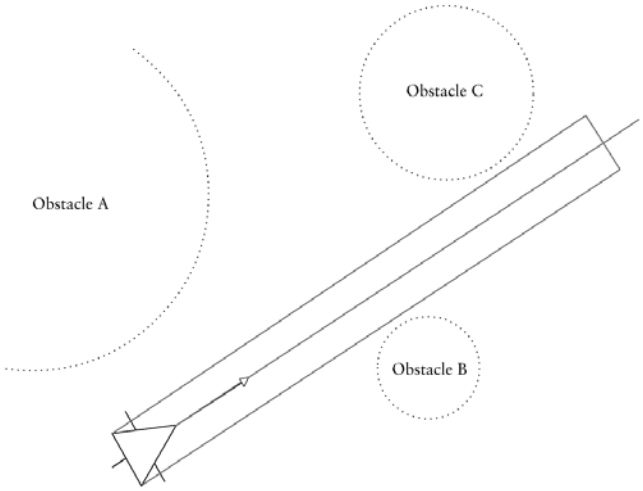

Figure 2.12 Obstacle avoidance

To implement this behaviour you place an imaginary cylinder in front of the vehicle. It is placed along the vehicles forward axis and has the same diameter as the vehicles bounding circle. The length of this cylinder is based on a constant, for example, minimum time to collision, and an obstacle further away than this distance is not a threat to the vehicle. The obstacle avoidance behaviour does a neighbourhood search to get a list of obstacles and determining if the cylinder intersects any of them. Reynolds method for calculate possible intersection is to project the local obstacle centre to the side-up plane, if the 2D distance from that point to the local origin is greater than the sum of the radii of the vehicle and the obstacle, then there is no potential collision. For any remaining obstacles Reynolds use a line – circle intersection calculation. The most threatening obstacle is the one that intersects the forward axis nearest the vehicle. The steering force needed to avoid the obstacle is calculated negating the side-up of the obstacle centre.

Barrier avoidance

For barriers and walls we need to adopt a different technique than using circles as boundary. One technique is to use the object itself as a bounding

box. Of course, this implies that the geometric shape should be rather simple, preferable rectangular. Instead of using just one testing vector for intersection test, two additional vectors are added resulting in; one vector testing the front, one to the left and one to the right. This way the vehicle can approach a wall and then maintain a certain offset from it. To implement this behaviour you predict the future position of the vehicle and see if it will intersect the box. If it does, continue by calculating the surface normal of the intersecting edge and use the component of the surface normal which is perpendicular to our forward direction as the corrective lateral steering. See Figure 2.13.

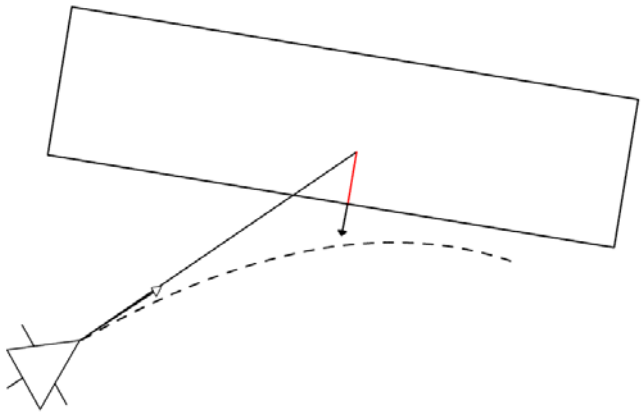

Figure 2.13 Barrier avoidance

Test every edge of the box for intersection. This is done using properties from linear algebra. If the two scalars

$$\left(\overrightarrow{AB} \times n\right) \bullet \overrightarrow{AP} \tag{2.7}$$

$$\left(\overrightarrow{AB} \times n\right) \bullet \overrightarrow{QA} \tag{2.8}$$

has the same sign there has been an intersection of that edge. Here A and B are the edge positions, P is the vehicles current position, Q is the predicted future position and n is the world up-vector (0,1,0). With this algorithm you don't need to continue calculating intersection for the remaining edges since this will return false for back faced edges, i.e. the first intersection is the correct one. Of course, this procedure will have to be repeated for the side vectors as well. To get the perpendicular vector, or the surface normal, to the edge you will have to solve

$$\vec{V} = \left(\vec{A} - \vec{B}\right)\perp \tag{2.9}$$

Agent avoidance

Agent avoidance is used to prevent vehicles from running into each other. This is done in two passes; the first pass will look for close vehicles within a certain radius from the original vehicle. If there is no direct threat it will continue and look for future possible collisions based on a time constant. See Figure 2.14.

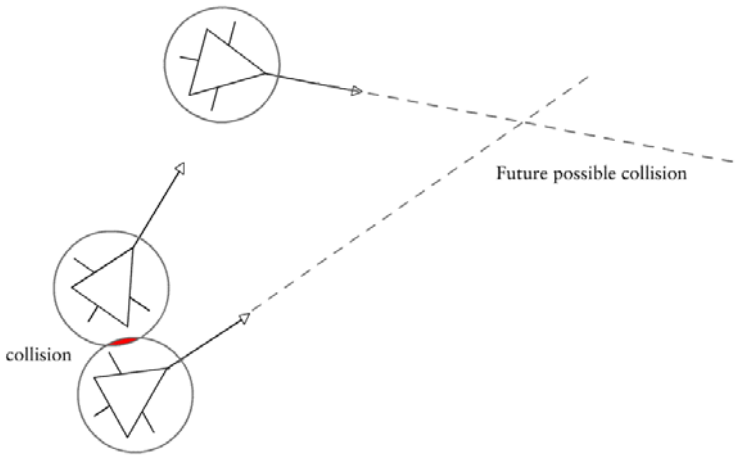

Figure 2.14 Agent avoidance

To compute agent avoidance you have to do a search in the vehicle's neighbourhood to get nearby vehicles. The first pass will iterate through this list to see if there are any vehicles within a minimum distance. This minimum distance is the sum of radii for the two vehicles plus a minimum separation distance. The distance between the vehicles is calculated;

$$d = \left|\overrightarrow{PQ}\right| \tag{2.10}$$

where P is our position and Q is the nearby vehicles position. If *d* is less than the minimum distance then corrective steering is required, but this is only necessary if the vehicle is in front of us. To see if it is in front of us we calculate the following property:

$$\vec{V}_{direction} * \overrightarrow{PQ} > 0 \tag{2.11}$$

where

$$\vec{V}_{direction} = \frac{\vec{V}_{velocity}}{\left|\vec{V}_{velocity}\right|} \tag{2.12}$$

The first term is the normalized velocity, i.e. the vehicle's direction. The second term is the offset vector to the other vehicle. If the threat was in front of us a new steering force is calculated;

$$\vec{V} = \left(\left(-\overrightarrow{PQ}\right) - \left(\left(\left(-\overrightarrow{PQ}\right) * \vec{V}_{direction}\right) * \vec{V}_{direction}\right)\right) \tag{2.13}$$

What I do here is calculating the perpendicular component to the vehicles current forward direction. This will make the vehicle turn away from the threat.

If no corrective steering was required the behaviour will continue to the second pass. For every vehicle in our neighbourhood we calculate the nearest approach time. First we calculate the relative velocity between the vehicles.

$$\vec{V}_{relativeVelocity} = \vec{V}_{velocityNeighbour} - \vec{V}_{velocity} \tag{2.14}$$

Next step is to calculate the relative speed.

$$s_{relativeSpeed} = \left|\vec{V}_{relativeVelocity}\right| \tag{2.15}$$

Now consider the path of the other vehicle in this relative space, a line defined by the relative position and velocity. The distance from the origin (our vehicle) to that line is the nearest approach. Take the unit tangent along the other vehicle's path

$$\vec{V}_{relativeTangent} = \frac{\vec{V}_{relativeVelocity}}{s_{relativeSpeed}} \tag{2.16}$$

Find distance from its path to origin (compute offset from other to us, find length of projection onto path)

$$\vec{V}_{relativePosition} = \overrightarrow{QP} \tag{2.17}$$

$$s_{projection} = \vec{V}_{relativeTangent} * \vec{V}_{relativePosition} \quad (2.18)$$

Now we calculate predicted time until nearest approach of our vehicle and the other vehicle.

$$s_{time} = \frac{s_{projection}}{s_{relativeSpeed}} \quad (2.19)$$

For that vehicle that had the shortest time you continue by predicting it's future position. The future distance between our vehicle and the other is calculated

$$\vec{V}_{futureOffset} = \vec{P}_{future} - \vec{Q}_{future} \quad (2.20)$$

$$d_{futureDistance} = \left|\vec{V}_{futureOffset}\right| \quad (2.21)$$

If this distance is less than a user specified threshold a potential collision was found and we have to compute a steering force to avoid it. First we check how their paths are aligned.

$$s_{parallelness} = \vec{V}_{direction} * \vec{V}_{neighbourDIrection} \quad (2.22)$$

Possible outcomes are parallel, perpendicular or anti-parallel and we will handle these cases below.

If $s_{parallenes}$ is less than *-sin (45º)* we are on anti-parallel head on paths. We calculate new steering force

$$\vec{V}_{steeringForce} = \overrightarrow{PQ_{future}} \tag{2.23}$$

$$s_{sideDot} = \vec{V}_{steeringForce} * \left(\vec{V}_{direction} \perp\right) \tag{2.24}$$

If $s_{parallenes}$ is greater than *sin (45º)* we are on parallel path. We calculate new steering force

$$\vec{V}_{steeringForce} = \overrightarrow{PQ} \tag{2.25}$$

$$s_{sideDot} = \vec{V}_{steeringForce} * \left(\vec{V}_{direction} \perp\right) \tag{2.26}$$

The last case will handle perpendicular paths. We will try to steer behind the other vehicle, but only if our speed is less then the other vehicle.

$$s_{sideDot} = \vec{V}_{neighbourVelocity} * \left(\vec{V}_{direction} \perp\right) \tag{2.27}$$

The final steering force for agent avoidance is:

$$\vec{V}_{steer} = \begin{Bmatrix} -1 & s_{sideDot} > 0 \\ 1 & s_{sideDot} < 0 \end{Bmatrix} \tag{2.28}$$

$$\vec{V}_{finalSteeringForce} = \left(\vec{V}_{direction} \perp\right) * \vec{V}_{steer} \quad (2.29)$$

Separation

Separation behaviour is used to give vehicles ability to keep a certain separation distance from each other. This prevents them from crowding together.

To compute separation behaviour you have to do a search in the vehicle's neighbourhood. See 2.5 for more information about neighbourhood search. This search will return a list of all the vehicles in the neighbourhood. A repelling force is now created for all the vehicles in the domain and summed up to get the overall force.

$$\vec{V} = \sum_{i=0}^{n} \overrightarrow{PQ_i} \quad (2.30)$$

That is, subtracting the position of our vehicle from the position of the other vehicles position, Q_i. The resulting vector is normalized and scaled to $1/r$ where r is the distance between the vehicles. The distance between the two objects will affect the force of the vector. It will be stronger if the distance is low and softer if it is farther away.

Cohesion

Cohesion behaviour is used to keep members of a group together or approach and form a group with other nearby vehicles.

As in the case with separation, a list of neighbouring vehicles is used. Average position of these vehicles are then computed, that is, the offset vector to each of the vehicles in the list are summed up and then divided by the number of vehicles in the domain.

$$\vec{P}_{avg} = \frac{\sum_{i=0}^{n} \overrightarrow{PQ_i}}{n} \quad (2.31)$$

The resulting position is the new target for the vehicle and a steering force is calculated using seek behaviour.

Alignment

Alignment behaviour is used to make nearby vehicles align themselves to each other.

As in the case with separation, a list of neighbouring vehicles is used. The velocities of these vehicles are retrieved and then summed up and divided by the number of vehicles to form an average direction.

$$\vec{V}_{avg} = \frac{\sum_{i=0}^{n} \vec{V}_i}{n} \quad (2.32)$$

The new steering force is difference between this average and our vehicles current velocity.

$$\vec{V} = \vec{V}_{avg} - \vec{V}_{current} \quad (2.33)$$

Leader following

Leader following behaviour will make a vehicle follow another moving vehicle. This behaviour combines separation, see Separation, and arrival behaviour, see Arrival. The vehicle will seek a point slightly behind the leader and slow down, i.e. activate arrival behaviour when it is near. If the agent finds him self in front of the leader he will try to steer away from the leader's path.

2.3 High-level decisions

By giving the agent capabilities to make high-level decisions he can reasoning about what to do in a certain situation. This part will discuss knowledge representation (KR), different approaches to store and use KR like rule-based system and decision trees.

2.3.1 Knowledge Representation

In order to make our agent intelligent we must have some smart way to present knowledge to him. This is called knowledge representation and it is used to express knowledge in a form so it can be interpreted by a computer. Fundamentally, the task of defining an interface for the agent to interact through is a matter of knowledge representation; how should the information passed to the AI module be encoded? A KR language is defined by two aspects:

- *Syntax* - specifying the structure of sentences.
- *Semantics* - defining their meaning

It is important to design a flexible representation that is convenient for computer implementation, this is called KR formalism. Many different low-level programming constructs can be used for knowledge representation. Examples from Alex Champandard [8] are used to illustrate different KR-formalisms.

Symbols

A symbol represents an object as any primitive data type, for example integers, floats, Boolean or character array:

[leftWallDistance 15.3]

[rightWall "unknown"]

The problem with using symbols is that each concept of the problem will need its own variable.

Object-Attribute-Value

This paradigm resolves the previous problem. By allowing objects to have several variables associated with them we limit the number of objects.

```
distance (leftWall, 15.3)

presence (rightWall, "unknown")
```

This approach is similar to classes and structures in programming.

Frames

A frame defines an object by specifying its current state and relationship to other frames.

```
frame-left-wall
distance (15.3)
present (true)
entity (frame-back)
```

Accessing the information about any concept is made very easy with this approach. But modifying it is more difficult because consistency must be maintained. This is called the frame-problem and it is beyond the scope of this thesis, John Funge [3] has a discussion and a solution to this problem. Transferred to programming the frame approach could be understood as pointers or references to other instances.

Semantic network

A semantic network stores relationships between the objects in a graph-like structure. Every node is a concept and the links describe the relationship, see Figure 2.15.

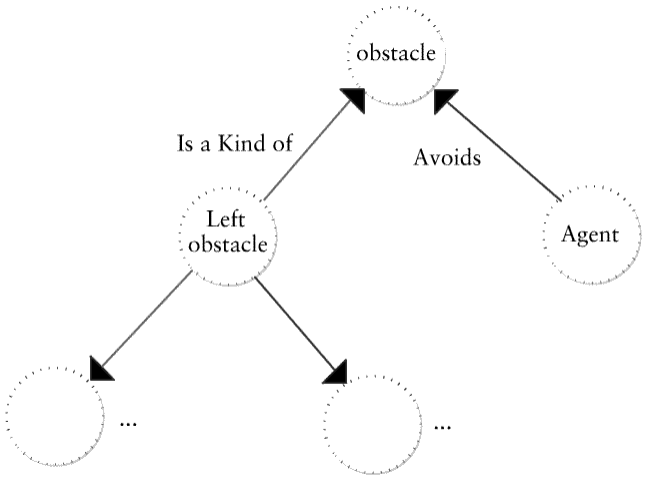

Figure 2.15 Semantic network

Relationships include *is-a*, *has-a*, and *instance-of*. Transferred to programming this corresponds to inheritance or dependencies.

2.3.2 Rule based system

A rule-based system (RBS) is a simple but successful AI technique. An RBS is a knowledge-based-system, dealing with the processing of information, for example sensory data from an agent. The RBS consists in a collection of "if this is true, do this". It can be compared to if-then-else statements in programming. In REACT we use RBS for problem solving, that is, reaching a conclusion from initial facts, and control, producing a sequence of actions in a simulation. An RBS can be called an expert system, that is, the rules base in the system is crafted by a human, or a domain expert.

An RBS consists of three components, the memory, the rule base and an interpreter, see Figure 2.16.

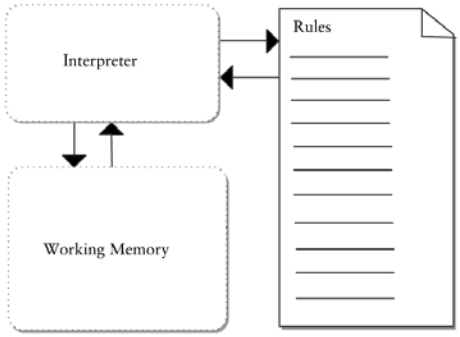

Figure 2.16 Rule base system

The memory is a collection of fact about the world and it is known as the internal representation. The memory could just be an array of symbols or a database. The second component, the rule base contains a collection of rules like:

```
IF <condition> THEN <action>
```

See knowledge representation described above. An important factor when creating a rule base is the data structure chosen for internal storage. We do not want the computational power to scale linear with the number of rules and I will show an optimized implementation in 3.2.3 that is both efficient and flexible. Finally, the interpreter is the part of the program that controls the RBS. It decides which rules match the current state and how to execute them. The interpreter is also an inference engine, that is, it allows knowledge to be inferred from the declarations.

There are two different kinds of inference mechanisms, forward and backward chaining. See Figure 2.17 and Figure 2.18. This describes how rules can be applied to the memory to solve problems. Forward chaining starts with a set of assertions, and repeatedly attempts to apply rules until the desired result is reached. Backward chaining starts with the hypothesis and attempts to verify it by returning to the current state. For an in-depth study of forward and backward chaining and a more elaborate discussion

about rule-based systems I recommend the reader to see Stuart Russels and Peter Norvigs book about artificial intelligence [13].

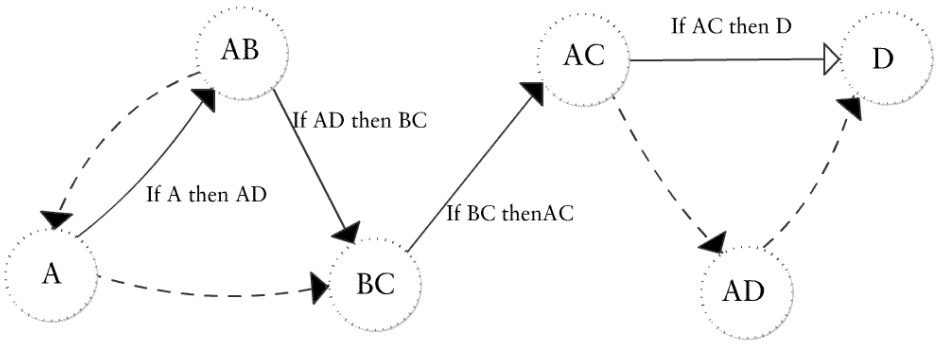

Figure 2.17 Forward chaining

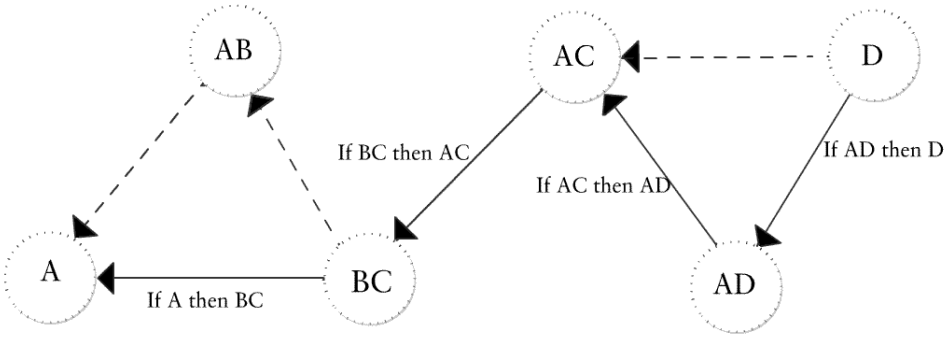

Figure 2.18 Backward chaining

The advantages with RBS are that it is simple to develop, extremely fast at runtime, flexible and easily extended. But at the same time it has some drawbacks like poor expressiveness in the rules and symbols aren't suited to modelling some problems and expert knowledge is not always available. Most of these drawbacks can be resolved in one way or another; one way is to use a decision tree.

2.3.3 Decision trees

A decision tree (DT) is a tree structure in which each branch node represents a choice between a number of alternatives, and each leaf node represents a decision. See Figure 2.19.

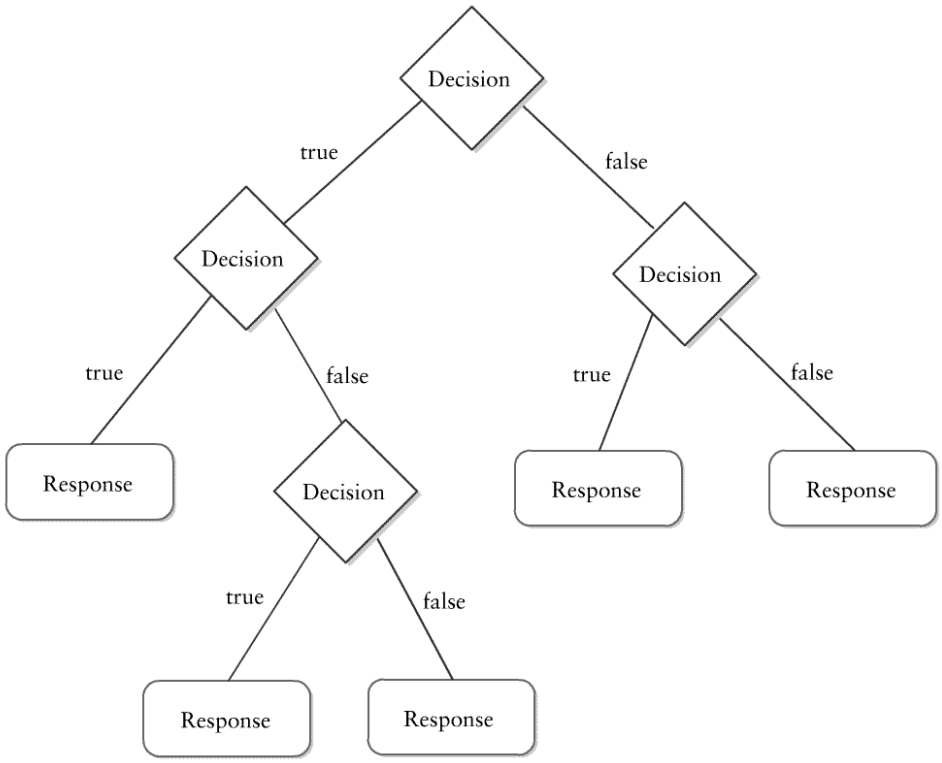

Figure 2.19 Decision tree

A decision tree starts with a root node on which it is for the system or the user to take actions. From this node, the system split each node recursively according to a decision tree learning algorithm that I will explain later. The final result is a decision tree in which each branch represents a possible scenario of decision and its outcome.

A decision tree must be learned from sample data, or examples, so it's necessary to gather it beforehand. An example is a set of attributes, or predictor variables. These attributes can represent just about anything

conceptually; in the context of weapons for soldiers, properties such as weight and damage it will cause to the victim etc.

In these examples there is an additional attribute with a special meaning known as a response variable, or dependant variable. This is the criteria for making decisions on each of the examples.

To illustrate the use of examples, we use a classic "play tennis" example.

Attribute	Possible values
Outlook	sunny, overcast, rain
Temperature	hot, mild, cool
Humidity	high, normal
Windy	true, false
Decision (response)	true, false

Figure 2.20 Attrributes and possible values

Sample set (learning examples)

Outlook	Temperature	Humidity	Windy	Decision
Sunny	Hot	High	False	False
Sunny	Hot	Normal	False	True
Overcast	Cool	High	True	True
Rain	Mild	High	False	True

Figure 2.21 Decision tree sample set

In the DT each level is a decision, and every node is test, a condition that has some sort of outputs. A node can have a large range of outputs but in REACT we use two, true or false, making it a binary decision tree. Each branch from the node corresponds to true or false. Figure 2.22

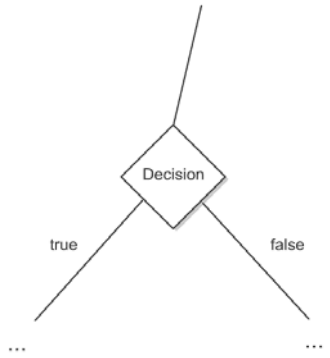

Figure 2.22 Binary decision tree node

One of the great things with decision trees is that we can estimate the response variable for unknown examples, based on the value of the attributes only. To process the data we traverse through the tree. We start at the root, and at each level of the tree, a decision is made based on the values of the attributes. The conditional test in each node is used to evaluate predictor variables. One unique branch from the node will match the result of this evaluation. The traversal branch to the next node where a new decision is made, see Figure 2.23.

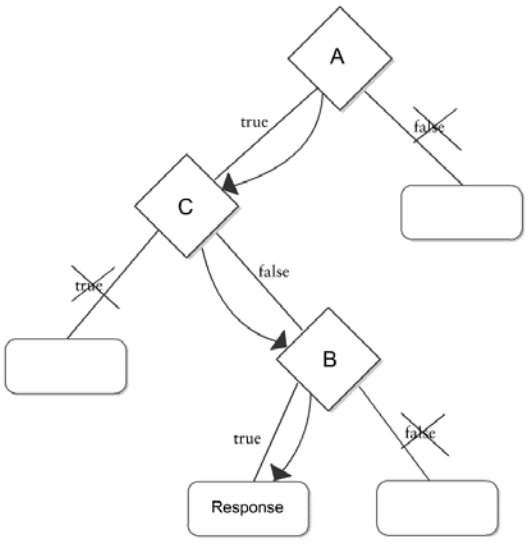

Figure 2.23 Traversal of a decision tree

The traversal will stop when the end of the branching is reached. Each leaf in the tree corresponds to a response.

Algorithm for decision tree traversal

```
currentNode = root
repeat
    result = node.evaluate(sample)
    for each branch from node
        if branch.match(result)
            node = branch.child
        end if
    end for
until node is a leaf
return value of leaf
```

It is possible to build these trees by hand, but decision trees have one great advantage over other techniques, they can be learned automatically. To build the tree from our set of examples we use a tree learning algorithm from J. R Quinlan [20]. This algorithm is called Recursive Partitioning and works by recursively partitioning the data set, building up the tree incrementally. The concepts of the algorithm are dividing and conquer and the goal is to build a tree for categorizing data samples. This is done by splitting the data set into roughly classified subsets, and try again until the classification is good enough.

Algorithm for Recursive Partitoning

```
function partition (dataSet, node)
    if not createDecision(dataSet, node)
        return
    end if
    for each sample in dataSet
        result = node.evaluate(sample)
        subSet[result].add(sample)
    end for
    for each result in subSet
        partition (subSet, child)
        node.add(branch, result)
    end for
end function
```

The amount of recursion is primarily determined by the complexity of the problem. The function will terminate quickly for simple data sets. It is important to point out that the same conditional test will never be used twice in the tree created. We need to explain the call createDecision in the algorithm above.

Given a data set, what would be the criteria for partitioning the set? The attributes are often chosen in a greedy fashion. Statistical analysis can reveal the attribute that does the best split. Splitting along this attribute generates subsets of minimal error. But we don't know if the whole tree will be optimal. What we need to do is to scan all the samples, and measure the impurity for each attribute. An impure set means that it contains a wide variety of response variables. What we want to do is to identify the impure attributes to make them pure, specifically to reduce the variety of response variables.

Entropy is used to measure impurity. The definition of impurity for Boolean/binary values from a set S is:

$$Entropy(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-) \qquad (2.34)$$

where p_+ is the ratio of true values and p_- is the ratio of false values. More generally, for sets with *n* number of classes:

$$Entropy(S) = \sum_{i=1}^{n} - p_i \log_2(p_i) \qquad (2.35)$$

$$p_i = \frac{|S_i|}{|S|} \qquad (2.36)$$

Entropy is defined as the sum of the weighted logarithms for each class i. p_i is defined as the proportion of samples in class *i*. But entropy is not sufficient to determine the best attribute to choose for partitioning. Information gain measure the expected decrease in entropy if an attribute is used for the split.

$$Gain(S, A) = Entropy(S) - \sum_{i=1}^{n} p_i Entropy(S_i) \qquad (2.37)$$

This expresses the information gain as the total entropy of a set, but subtracting the entropy of the subsets that are created during the split. See pseudo code below.

Function for finding best attribute for partitioning a Data Set

```
function createDecision(dataset, node)
    max = 0
    # find the impurity for the entire training data
    entropy = computeEntropy(dataSet)
    for each attribute in dataset
        # split and compute total entropy of subset
        e = entropy-computeEntropySplit(attribute, dataset)
        # find the best positive gain
        if e > max
            max = e
            best = attribute
        end if
    end for

    # create the test if there's a good attribute
    if best
        node.evaluate = createTest(attribute)
        # otherwise find the value of leaf node
    else
        node.class = findClass(dataset)
    end if
end function
```

2.4 Agent architectures

One simple approach for building an intelligent agent is to implement a finite state machine. A finite state machine is a model of computation with a limited amount of memory. Each machine has only a finite number of possible states, see Figure 2.24. A transition function determines how the state changes over time, according to the inputs to the finite state machine.

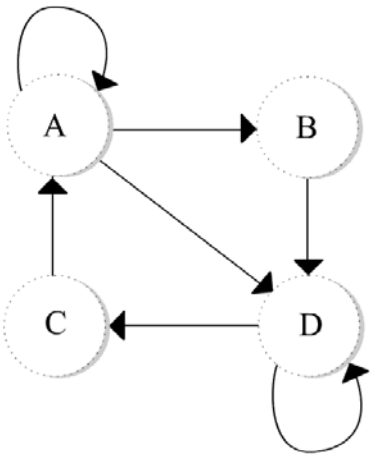

Figure 2.24 Finite state machine

Another approach is to make use of the growing popularity of agent-based architectures and methodologies in the area of artificial intelligence. This gives us an opportunity to apply recent advances in AI to our crowd system. The cycle in most of these architectures is "Sense Think Act", a standard approach to building systems which have to communicate with the environment and it is similar to the way humans perceive and act upon information. Three popular architectures are John Funge's Cognitive Agents [3], Michael Winikoff's Simple Agent Concept [6] and the Free Will architecture [4]. Free Will is in fact based on both Funge's and Winikoffs approach. I will explain each of these architectures as my work is based on these different approaches.

2.4.1 Cognitive Agents

The cognitive architecture (FCA) from Funge defines an agent as “a character that, during the course of a computer game or animation, can decide how to behave on its own.” [3]. He propose a new approach to high-level control in which the user gives the character a behaviour outline, or “sketch plan”. This “sketch plan” is constructed using a logical language, known as the situation calculus. The agent views its world as a sequence of what Funge calls “snapshots”. These snapshots are known as situations. The “sketch plan” will give the agent an understanding of how the world can change from one situation to another and describing what the effect of performing each given action would be.

Funge means that the cognitive layer is only one of many building an agent, see Figure 2.25. A developer trying to build a virtual scene inhabited by intelligent agents will be faced with many problems apart from those of artificial intelligence. The architecture involve the whole process of building a virtual character like forward and inverse kinematics, rigid body motion and dynamics.

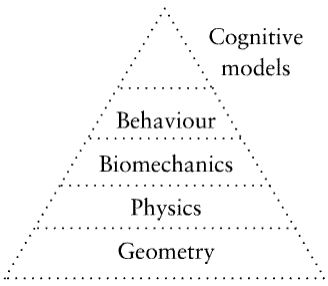

Figure 2.25 FCA architecture

The cycle that each agent go through is “Sense Think Act”. For low-level behaviours like “avoiding collisions” and basic locomotive capabilities such as “go to a particular location”, Funge recommends the use of a rule base system.

For high-level reasoning he suggest using trees, or pruned trees for better performance. Each node in this tree is an axiom from the “sketch plan” and the reasoning engine implemented in Funge’s approach becomes a theorem prover.

2.4.2 Simple Agent Concepts

The Simple Agent Concepts (SAC) approach extends the BDI[3] architecture [19] to make it more accessible as a software engineering development framework. Winikoff's approach is much more implementable than the original BDI architecture and he focuses more on goals, events and construction of plans. See Figure 2.26. SAC use the same cycle for the agent as in the FCA, that is, "Sense Think Act".

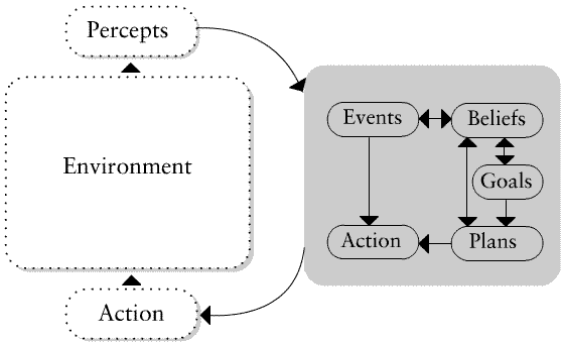

Figure 2.26 SAC architecture

SAC agent execution model

1. Percept are interpreted (using beliefs) to give events
2. Beliefs are updated with new information from percept
3. Events yield reflexive actions
4. Goals are updated, including current, new and completed goals.
5. If there is no selected plan for the current goal, or if the plan has failed, or if reconsideration of the plan is required (due to an event) then a plan is chosen.
6. The chosen plan is expanded to yield an action
7. Action(s) are scheduled and performed

It is the developer's responsibility to identify all of the agent's goals. The developer identifies the main goal and possible sub goals which are used in achieving main goals. Additionally the agent may be equipped with a plan library from which it can select appropriate plans when necessary.

[3] Belief Desire Intention

2.4.3 Comparison between FCA and SAC

FCA and SAC are not the solution to all problems involved when creating a virtual world with intelligent agents. Adam Szarowicz and Peter Forte [5] make a comparison between the FCA and SAC architectures where they address some of the advantages and disadvantages of the architectures.

- Both architectures stress the importance of goals, reactive and proactive behaviour, and the need for some representation of the character's knowledge (beliefs).

- They are both based on the sense-think-act cycle.

- In FCA the sensing process will always update the character's world model but in SCA it will give rise to internal events that can update the agent's beliefs.

- FCA's focus is on animation while SAC is more of a general-purpose architecture for designing and implementing situated agents. This means that FCA includes animation concepts like kinematics and geometry.

- The SAC approach builds upon BDI agents and that states that the agents should maintain a list of predefined plans, so SAC includes the concept of plan libraries. FCA allows pre-defining behaviour (rule base system or state machines) but for the goal-directed higher level reasoning it proposing the use of situation trees.

- FCA does not provide any explicit mechanism to allow interaction between agents

2.4.4 Free Will

Free Will's aim is to create intelligent and realistic animation in the form of crowd scenes. Free Will contains both elements found in animation driven systems and distributed multi-agent systems. The agents are implemented as modified SAC agents with a body layer for the visual part of the agent, that is, the agent is embodied and situated. See Figure 2.27.

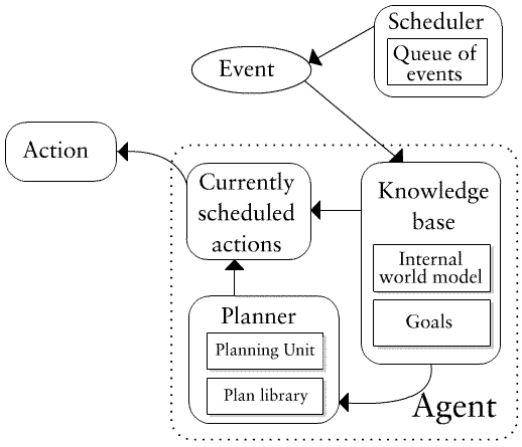

Figure 2.27 Free Will architecture

Free Will architecture is created with the intention to connect to an existing animation package. They show an example where they use Discreet's 3D Studio Max as animation package, but any existing 3D package should work. In this example there is no real time feedback. The result is visible first after render.

The internal world model that the agent has is a substantial part of the agent knowledge and is updated through sensing, similar to the approach in FCA.
The concepts from SAC give them means to build the "mind" of the agent and they store plan libraries instead of creating plans from scratch every time a new plan is needed, new plans are constructed only if there is no existing template. The interaction between agents is modelled by reaction to events.

The different between SAC agents and Free Will agents are that events to which the agent responds to are created external in contrast to SAC's internal creation. Free Will use a global scheduler for event handling and the agent query this scheduler when it wants to update its knowledge or perform an action.

Free Will agent execution cycle:

1. a sensing action is executed by the agent
2. agent's beliefs get updated
3. current plan is evaluated, if there is a need to perform a reflexive action the current action queue is cancelled and a new action gets submitted
4. goals get updated if necessary
5. plan is updated if necessary
6. last action is chosen and submitted to the scheduler, when the agent is asked to execute the action it also submits a new sensing action to be put in the event queue.

2.5 Spatial data structure

In a multi-agent system it is unavoidable that agents will have to interact with each other. To make agents look realistic they have to avoid other agents and then they need to know where these agents are. One way of doing this is to iterate over all the agents in the system to query for there position, but when the agents increase, this will be a big performance hit for the system. The complexity for these kind of proximity queries is $O(n^2)$, that is, if you double the number of agents you will quadruple the amount of time needed to do proximity queries. A significant point is that no matter how fast each query can be done, the cost of the proximity queries will eventually dominate all other computational costs. What we need is some sort of spatial data structure. A spatial data structure is one that organizes geometry in some n-dimensional space. The organization of spatial data structures is usually hierarchical. This means, that the topmost level encloses the level below that level, and so on.

The only thing that we are interested in is to find the close neighbours to our agent. One way to do that is to store all the agents in the spatial data structure mentioned above. This structure will keep the agents "pre-sorted" based on the location in space. There are many different techniques for spatial data structures, for example bin-lattice spatial subdivision [2] or binary space partitioning (BSP) trees [15]. The ideal solution would be to

supply a position in space and a radius from this position to get the neighbours.

2.5.1 KD-Trees

Kd-trees [16] are effective data structures for small and moderate numbers of dimensions. See Figure 2.28. The purpose of kd-trees, or k-dimensional trees, is to hierarchically decompose space into a relatively small number of cells such that no cell contains too many input objects. The inputs are a set of n-points in k-dimensions. This provides a fast way to access any input object by position. We traverse down the hierarchy until we find the cell containing the object and then scan through the few objects in the cell to identify the right one.

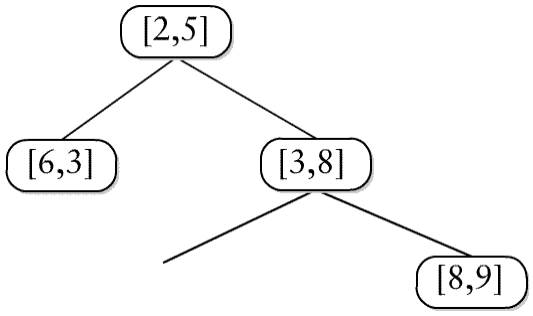

Figure 2.28 A 2d-tree of four elements.

The typical algorithm constructs the tree by partitioning point sets. Each node in the tree is defined by a plane through one of the dimensions that partitions the set of points into left and right sets, each with half the points of the parent node. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after *log n* levels, with each point in its own leaf cell. See Figure 2.29

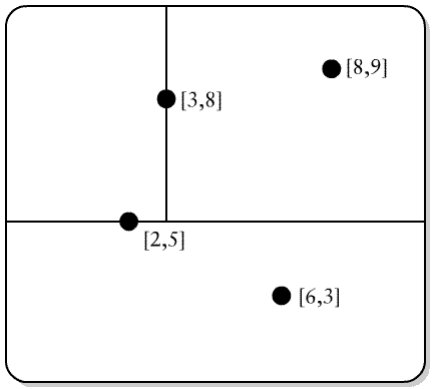

Figure 2.29 How the tree of figure 2.28 splits up the x,y plane

The cutting planes along any path from the root to another node define a unique box-shaped region of space, and each subsequent plane cuts this box into two boxes. Each box-shaped region is defined by 2k planes, where k is the number of dimensions. In any search performed using a kd-tree, we maintain the current region defined by the intersection of these half-spaces as we move down the tree. The building process has the complexity *O(n log n)*

The implementation of the Kd-tree structure used in REACT includes a nearest neighbour search and a range search.

Nearest neighbour search

To find the point in the set S closest to a query point q, we perform point location to find the cell c containing q. Since c is bordered by some point p, we can compute the distance d(p,q) from p to q. Point p is likely very close to q, but it might not be the single closest neighbour. Suppose q lies right at the boundary of a cell. Then q's nearest neighbour might lie just to the left of the boundary in another cell. Thus we must traverse all cells that lie within a distance of d(p,q) of cell c and verify that none of them contain closer points. With nice, fat cells, very few cells should need to be tested.

Range search

This will see which points lie within a query box or region. Starting from the root, check to see whether the query region intersects or contains the cell defining the current node. If it does, check the children; if not, none of the leaf cells below this node can possibly be of interest. We quickly prune away the irrelevant portions of the space.

Chapter 3

3 Implementation

This chapter will discuss the implementation of REACT and it is based on the theory explained in the previous chapter. I will discuss the overview of REACT and how the different parts are built up in a theoretical way. Finally I will discuss the integration into an existing animation package.

3.1 Overview of REACT

To make REACT work in a professional film environment it is based on several different sub-systems. I will show the complete pipeline of REACT and break it down to the simulation engine to give the reader a clear overview of the system before we continue to discuss the REACT agent.

3.1.1 Pipeline

The pipeline of REACT consists of seven different modules or sub-systems. See Figure 3.1

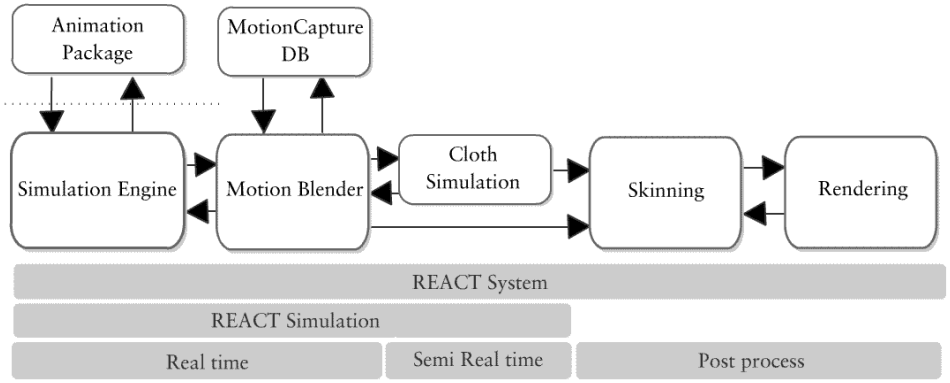

Figure 3.1 REACT pipeline

The first and the one discussed in this thesis is the simulation engine. The simulation engine is responsible for the main simulation and it is the module that handles the communication with the existing animation package. It also sends and retrieves data from the motion blender so that the user can preview smooth motion in the animation package before rendering it. When the user is satisfied with the result he sends the final simulation, that is, the motion path that each individual agent has produced during the simulation, to the motion blender which blends the motion and simulates any cloth that might have been used in the simulation. The next step in the pipeline is skinning the agent. In this process the skeleton used in the simulation gets a skin, or a body. The final step is rendering and REACT uses Pixars Renderman as rendering system.

3.1.2 Simulation engine

The simulation engine is tightly connected with the animation system used as host. But it is also responsible for the communication with the rest of the system discussed above. The simulation engine can be broken down further to show the different sub-systems used inside.

REACT's simulation engine can be broken down in five large parts. These are:

- *Main simulation node*, which is responsible for everything that happens. It will know which agents that are connected, obstacles and barriers that are present and handle the communication with the animation system and the rest of the pipeline.
- *I/O interface*, responsible for reading and writing files to disc.
- *Animation system integration*, responsible for communication with the animation system.
- *Agents*, the visual part of REACT. The agents will be discussed in detail below.
- *Objects*, this can be obstacles, barriers, terrains etc.

3.2 The REACT Agent

The REACT agent is an independent autonomous character capable of making his own decisions and has a capability for adapting to new environments. The design concepts of the REACT agent are sound and represent much of what I have discussed in the theoretical basis. Further on the agent is embodied and situated in a virtual world. The agent uses senses to gather local information, similar to the way humans interact with their

environment. It will react to stimuli from the environment, sound for example.

3.2.1 Architecture

Implementing a finite state machine (FSM) would be an easy and efficient approach. There is research comparing FSM and agent concepts [7] and it shows that there is a linear increase in cycle time for both models when the number of agents increases, though the overhead is much higher for the agent concept. But the code complexity for the FSM increases quadratically as the system scales up with more behaviours compared to linear for agent concepts. Following the discussion above and in the theory chapter about agent architectures I have chosen to base my agent concept on the Free Will architecture. Although it differs in some major aspects the foundation is the same. While Free Will creates external events, the REACT agent will create events internally that can update the agents beliefs. This is more in line with the SAC architecture.

Execution cycle for REACT agents.

1. percept are interpreted (using beliefs) to give events
2. agent's beliefs get updated
3. current plan is evaluated, if there is a need to perform a reflexive action the current action queue is cancelled and a new action gets submitted
4. goals get updated if necessary
5. plan is updated if necessary
6. last action is scheduled and performed

3.2.2 The knowledge base

The knowledge base (KB) stores all the knowledge that is relevant for the agent like goals and facts. The facts are a set of rules that consists of symbols. These symbols represent Boolean values and the syntax of these rules is:

IF SYMBOL THEN SETACTION

These facts are used by the Rule Base System. The knowledge base also contains pre-calculated decision trees. These are used by the agent to make high-level decisions.

It is up to the user or the "knowledge engineer" to create the knowledge for the agent. The user is also responsible for supplying main and sub goals for the agent. The main goal can be "go to this position" and sub goal can be "avoid enemies".

3.2.3 Control reactive steering behaviours

One example of a reactive behaviour is obstacle avoidance. As soon as an agent realizes that he is about to collide with an obstacle he will clear his event queue and try to avoid the obstacle.

Reactive behaviour has one major advantage that they are fully deterministic, that is, the exact output is known given any input pattern. The time complexity for determining the output is generally constant. There is no thinking or deliberation, the answer is a reflex from the system, available almost immediately.

Reactive behaviour can be controlled in different ways. One quick and dirty solution would be to just add some if and else statements directly in the program code but this would be unmanageable when the number of behaviours exceeds two digits. Two other solutions are to use a finite state machine or a Rule Base System. FSM has already been discussed and the complexity when adding more behaviour rules out this approach. Left is Rule Based System and this is the approach taking in REACT.

3.2.4 High-level decisions

For efficiency, the high-level decisions that the REACT agent does is structured in decision trees with each internal or non–leaf node applying one rule so that the leaf nodes contain the list of actions to be selected. The agent has access to these trees from the knowledge base. The agents can have an unlimited number of decision trees. Once a decision has been made based on the rules, there is a list of actions to be performed. Typically, this is the activation of behaviours and animations. See Figure 3.2 for a simple example of a REACT decision tree.

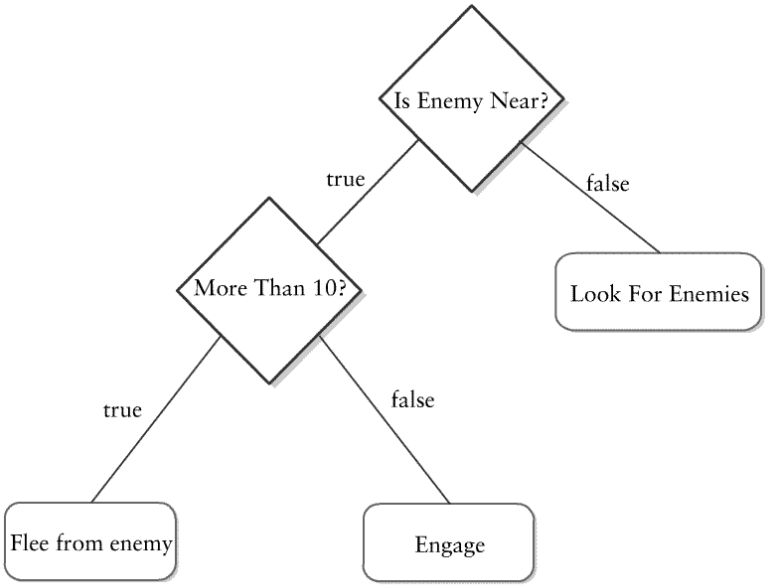

Figure 3.2 Example of REACT decision tree

3.3 Implementation in Maya

Given that the commercial animation package Alias Maya is so feature rich and mature that studios build their entire production pipelines on top of it, REACT is designed to transparently integrate directly into it as a plug-in.

Maya is node based which means that an "object", such as a sphere, is built from several nodes: a *creation node* that records the options that created the sphere, a *transform node* that records how the object is moved, rotated, and scaled, and a *shape node* that stores the positions of the spheres control points. See Figure 3.3.

Figure 3.3 Nodes

3.3.1 Maya API

To be able to write a plug-in that transparently integrates into Maya you need internal access to Maya. This is provided through Maya API which is a C++ API. This means that the core of REACT is completely written in C++. This gives a speed advantage compared to other programming languages that is necessary when working with complex 3D graphics. The objects created using Maya API will appear as nodes within Maya. This means that Maya have access to the data stored in this object and this gives us an important advantage. Maya is equipped with a scripting language that can access this data during run time.

3.3.2 MEL

MEL is Maya's scripting language and stands for Maya Embedded Language. It is deeply integrated with Maya, and allows you to do anything from open a window to total customization of the Maya interface. In short, anything that you can do with the C++ API you can do with MEL. The only disadvantage is that it is slower since it is a script and has to be interpreted.

MEL gives us the possibility to create a customized graphical user interface inside Maya that can connect directly to REACT to set and get attributes and execute commands inside the REACT node.

3.3.3 REACT in Maya

As discussed earlier Maya works with nodes. This means that REACT is built up from nodes. A typical setup can be found in Figure 3.4.

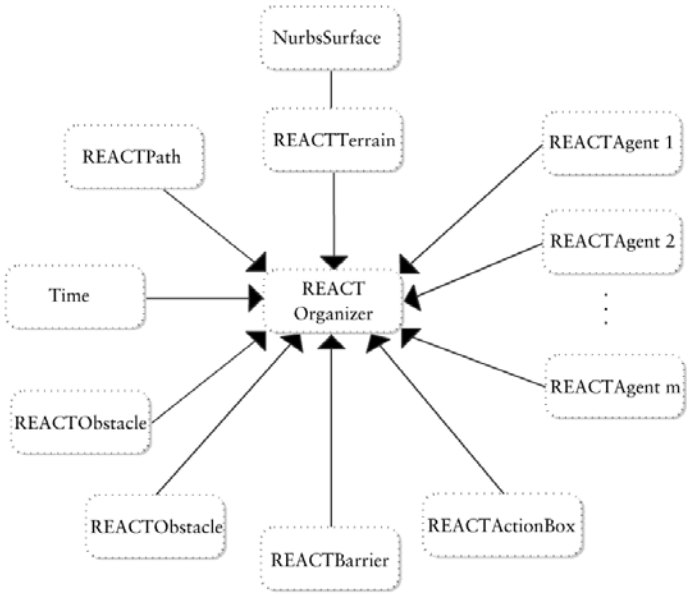

Figure 3.4 Nodes in a REACT simulation

REACT consists of eight different nodes. These nodes are discussed in details below but I will give a brief explanation to each of them here.

- *REACTOrganizer* is the node that has the main responsibility in REACT. Everything that is used in the simulation is connected to this node.

- *REACTAgent* is by far the most complex part in REACT. This is the actual visible agent and it is here you find the body and the brain.

- *REACTTerrain* is recognized by the agents as a surface and they will follow the topology of this surface.

- *REACTPath* is used to make agents follow a path.

- *REACTFlowfield* is used to guide the agents over a flow field or a vector field.

- *REACTActionBox* is a node that works as stimuli for the agent. It can change the agent's current state; give him a certain style etc.

- *REACTObstacle* is recognized by the agent as an obstacle that you can go around.

- REACTBarrier is recognized by the agent as a wall that you can follow or go around.

3.3.4 REACT Organizer

In almost all simulations there is a main program controlling the flow and in REACT this program is called REACT Organizer (RO) and is a node within Maya. Every other REACT node present in the simulation is connected to this organizer. RO keeps track of all the agents, terrains, obstacles, barriers, paths, flow fields and actions boxes.

The agents are "pre-sorted" in a Kd-tree based on the location in space. The supported search methods include a nearest neighbour search and a range search.

The execution cycle for RO is:

1. See which agents are connected to this node and update Kd-tree.
2. Find connected obstacles and update list.
3. Find connected barriers and update list.
4. Find connected action boxes and update list;
5. See if the terrain has changed, in that case update terrain.
6. Are their any paths connected, in that case update path list?

One important aspect with Maya is that it will only try to update the attribute if the attributes dirty flag is set to true. Thus, a dirty flag set to true means that the attributes value has changed. But instead of updating all nodes it will only update the nodes that are affected by this attribute. This saves lot of time during run time.

When RO has updated all the values the agent can query for neighbours, obstacles barriers etc.

3.3.5 REACT Agent

As mentioned above REACT Agent is by far the most complex part in REACT. It is in agent that the real artificial intelligence exists, with low-level behaviours, high-level decisions and complex data structures for making these decisions.

Knowledge

When the agent is created he will read in his knowledge from a file resident on the hard drive. All the read and write to disc is handled through XML so the knowledge is saved in an XML file.

In the beginning of the knowledge file there is a definition of symbols. These symbols represent some of the attributes that exists in the agents "working memory". The syntax of these symbols is shown below:

```
<memory>
    <symbol name=”barrierInFront”/>
    <symbol name=”barrierInLeft”/>
    <symbol name=”obstacleInFront”/>
    <symbol name=”turnRight”/>
</memory>
```

When the definition has been loaded by the agent we bind these symbols to sensor or affector functions in agent. This is done using functors[11]. Functors are basically function pointers and are a good way of implementing callbacks in C++. If we bind the symbol "barrierInFront" to a sensor function in agent and `turnRight` to an affector function we can set and get attributes of the agent using the symbols. What this means is that if we query the symbol `barrierInFront` for a value we actually query the sensor function connected to this symbol. It will become clearer when I show the next step in the procedure.

The first knowledge to be read is for the Rule Base System. The XML syntax for the REACT rule base is shown below.

```
<rulebase>
    <rule>
        <conditions>
            <symbol name=”barrierInFront” value=”true”/>
            <symbol name=”barrierLeft” value=”true”/>
        </coonditions>
        <actions>
            <symbol name=”turnRight” value=”true”/>
        </actions>
    </rule>
</rulebase>
```

The agent will parse this rule and when he test the conditions he will query the function connected to `barrierInFront` to see if that function returns true. If the condition is true he will execute the action which in this case is a symbol called `turnRight` and that are connected to an affector function in agent that will make the agent turn right.

The second knowledge part is the decision tree data. This is a pre-calculated tree where each node has the following syntax:

```
<node>
    <decision> ... </decision>
    <branch>
        <match> ... </branch>
        <node> ... </node>
    </branch>
</node>
```

The approach with binding functions is similar to the Rule Base knowledge. A detailed discussion about loading and binding Rule base and decision tree knowledge can be found in “AI Game development – Synthetic Creatures with Learning and Reactive Behaviors” [8].

Interpret percept

In order to update the internal world model the agent query REACT Organizer for information about the neighbourhood. The input is the current position and the response is a list of agents in that neighbourhood, (RO queried the Kd-tree with a range search where the range is the agent’s vision cone). Further on the agent gets a list of *REACTObstacles* and *REACTBarriers* that are present in the simulation.

Reactive behaviour

A reactive behaviour rule out all other behaviours. A reactive behaviour in REACT is when the agent is about to collide with an obstacle, barrier or another agent. There is no need for any knowledge in these situations. Humans do not think when avoiding obstacles, it is done automatically, so the agents just imitate this behaviour. The rule base can be used to force a complex behaviour instead of a singe reactive behaviour. If the agent is about to collide with an obstacle and has a group of agent on the right side the user can construct a rule that tell the agent to turn right in these situations.

Control movement, behaviour, states and styles

A director for a movie would most certainly want to control the agents just like extras. This is called explicit control, the director sets up the story line. The other approach is called implicit control; the agents have full control to do what they want. I have chosen a combination of these two approaches. You can let the agents control the situation but I also give the director the possibility to stimulate the agents to do certain things. This could be changing behaviour, changing state or changing style of a motion.

One powerful approach is that you can paint a flow field using *REACTFlowfield* on the surface that will affect the agent. This way the director can point out how the agents should move over the surface to match his ideas for the shoot.

Another way of controlling the movement of the agents is to use *REACTPath*. This node will create a path that the agent or agents can follow. The user can set how much the agent can deviate from it and how much it attracts the agent. It gives a very dynamic result when you have a leader that follow the path and the rest of the agents follow the leader.

The most powerful method of controlling the agents is through *REACTActionBoxes*. These boxes are capable of changing almost any attribute of the agent. They can change the state, like changing from march to charge, they can add a style to a state, like banging the sword against the shield, they can change the behaviour, for example from path following to seek behaviour and they can change all attributes of the agent, like radius o mass etc. As soon as the agent enters one of these boxes it will try to change the agent's attributes.

The final option to control the agent is through MEL. MEL can access all the attributes of the agent and it gives the user endless possibilities to create

customized user interface, automated scripts or a completely new node that is created to control REACT.

Control the body

The movement of the body is based on motion capture. Every state is basically recorded motion from real humans. Example of states is idle, march, charge, die etc. Styles will add additional motion on top of the state. That can for example be banging the sword against the shield while marching. Both state and styles can be controlled by the user. The user can also control different parts of the agent body. This can be the head, arms and legs, hip etc. This is a powerful way of making the agent more believable. The head can be scripted to follow the leader if he walks by, or that the agent starts turning his head before the body turns to see if it is clear.

3.3.6 REACT Terrain

The agent will follow the topology of any surface connected to the organizer. But if the user connects a nurbs surface *REACTTerrain* will create a proxy. This proxy is a polygon surface and the reason for that the calculations for the height field are done much faster on polygon data than on nurbs data.

Another advantage with REACTTerrain is that you can connect a very large and detailed nurbs surface but use a proxy with less resolution and with a size that just cover the area around the agents. This will be a major speed advantage compared to the full resolution nurbs surface.

Chapter 4

4 Conclusion

In this, the final chapter, I will try to summarize the previous chapters and point out some promising directions for future work. Finally I will conclude this thesis.

4.1 Summary

The discussions in this thesis cover a broad area of artificial intelligence and computer graphics.

After introduction I outlined the underlying theoretical basis for REACT in chapter 2. We discussed fundamental AI-concepts, low-level behaviours including locomotion and steering behaviours. We continued with high-level decisions and agent architecture and finally I talked about spatial data structures.

I moved on to discuss the implementation of REACT. First in a theoretical way and then the integration into an existing animation package. I discussed the different sub-systems of REACT and how REACT works in Maya as nodes. I moved on to discuss the important parts of REACT, the main node, the organizer and the agents.

4.2 Conclusion

By using existing knowledge from the game community, which have had a long experience from game artificial intelligence and new research from the field of artificial intelligence I have implemented REACT. REACT is based on high-level behaviour that uses an underlying layer of low-level behaviour. The high-level capabilities gives the agent means to reasoning about how to achieve certain goals based on a knowledge base of rules and facts that are present in the virtual world. This gives the agent a degree of

implicit and explicit control for the agents. The director can still direct the agents just like extras where he think that it is necessary and let the agent be autonomous everywhere else.

Agent's knowledge is scriptable via XML which make it very extensible to other systems or programs.

REACT is designed to integrate directly into Maya as a plug-in. This means that the animators can continue to animate their characters via their animation package of choice, rather than having to learn a new technology. In addition, many animators are already familiar with the workflow of Maya, so learning curves are reduced.

REACT is already in use in the visual effects industry where it has proven itself to a worthy competitor to the existing systems on the market.

4.2.1 Example 1

Example of obstacle avoidance:

Figure 4.1 Agents avoiding obstacles

4.2.2 Example 2

Example of barrier avoidance:

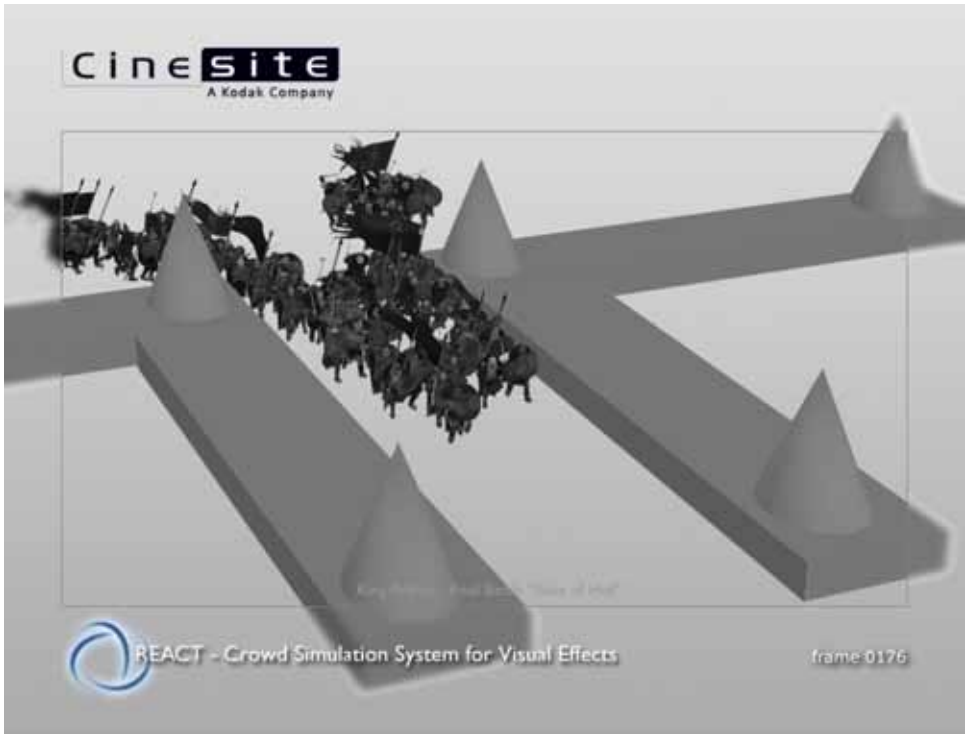

Figure 4.2 Agents avoiding barriers

4.2.3 Example 3

Example of changing state:

Figure 4.3 Agents changing state

4.3 Future Work

REACT is not a perfect system and there is much room for improvements. The logic used in REACT can only model crisp values, for example true or false. With fuzzy logic it can be mostly false or mostly true. The fuzzy approach provides shades of grey between black and white so the next step is to implement Fuzzy Logic to make the decisions more human like.

Another interesting field is genetic algorithms. Genetic algorithms can optimize parameters in multi dimensional space. You could use these algorithms to adjust weapon strategies for soldiers for example.

Finally I would like to develop the agent senses a bit further. Make the agent see the world instead of getting a list with objects and agents in his neighbourhood. I would also like to implement sound awareness for the agent.

Chapter 5

5 Acknowledgements

I would like to thank Cinesite (Europe) Ltd. for their trust in me and specially my supervisor Pete Medrow and Douglas Harsch, head of technical direction. I would also like to thank my girlfriend Linda Sointio for her support during this thesis and my academic supervisor Professor Anders Ynnerman.

Bibliography

[1] Craig W. Reynolds, Steering Behaviors for Autonomous Characters Conference Proceedings of the 1999 Game Developers Conference, pages 763-782.

[2] Craig W. Reynolds, Interaction with Groups of Autonomous Agents. Conference Proceedings of the 2000 Game Developers Conference pages 449-460.

[3] John David Funge, Make Them Behave – Cognitive Models for Computer Animation (1998) Thesis PhD.

[4] Peter Forte, Adam Szarowicz, The Application of AI Techniques for Automatic Generation of Crowd Scenes, The Eleventh International Symposium on Intelligent Information Systems, June 3-6, Sopot, Poland pages 209-216, 2002

[5] Peter Forte, Adam Szarowicz, Combining Intelligent Agents and Animation AIxIA 2003 - Eighth National Congress on AI, September 22-26, Pisa, Italy, 2003

[6] Michael Winikoff, Lin Padgham, James Harland, Simplifying the development of Intelligent Agents, Technical Report TR-01-3. 14th Australian Joint Conference on Artificial Intelligence, Adelaide 2001

[7] Arran Bartish, Charles Thevathayan, BDI Agents for Game Development AAMAS 2002, pages 15-19.

[8] Alex J, Champandard, AI Game development – Synthetic Creatures with Learning and Reactive Behaviors. New Riders 2004, ISBN 1-5927-3004-3

[9] Ken Perlin, An Image Synthesizer, SIGGRAPH 1985 Proceedings, pages 287-296.

[10] Yannis Smaragdakis, Don Batory, Mixin-Based Programming in C++, Net.Objectdays 2000

[11] Lars Haendel, The Function Pointer Tutorials http://www.function-pointer.org/

[12] BOOST, http://www.boost.org/

[13] Stuart Russel, Peter Norvig, Artificial Intelligence - A Modern Approach, Prentice Hall 1995, ISBN 0-13-360124-2

[14] David Gould, Complete Maya Programming. Morgan Kaufmann 2003 ISBN 1-55860-835-4

[15] Tomas Akenine-Moller, Eric Haines, Real-Time Rendering Second Edition. A K Peters 2002 ISBN 1-56881-182-9 pages 346-356

[16] Andrew Moore, An introductory tutorial on kd-trees, Technical Report No. 209, Computer Laboratory, University of Cambridge, Robotics Institute, Carnegie Mellon University, 1991

[17] Gustave Le Bon, The Crowd, Transaction Publishers 1995, ISBN 1-56000-788-5

[18] R. Kurzweil. The Age of Intelligent Machines, MIT Press Cambridge Massachusetts, 1990

[19] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-Architecture. Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference 1991, pages 473–484.

[20] J. R Quinlan, Induction of Decision Trees, Machine Learning 1, 1986, pages 81-106